Spring Cloud译:春天的云

本帮助文档是 觉得烦死 整理--QQ:654638585
声明：
中文文档都是由软件翻译,翻译内容未检查校对,文档内容仅供参考。
您可以任意转发，但请至保留作者&出处(http://bolg.fondme.cn)，请尊重作者劳动成果,谢谢！

译:

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线）的工具。 分布式系统的协调导致锅炉板模式，使用Spring Cloud开发人员可以快速站出实现这些模式的服务和应用程序。 它们可以在任何分布式环境中运行良好，包括开发人员自己的笔记本电脑，裸机数据中心以及Cloud Foundry等托管平台。

版本：Finchley.RELEASE

1. Features译: 1.功能

Spring Cloud专注于为典型用例和可扩展性机制提供良好的即时体验，以覆盖其他人。

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Distributed messaging

Part I. Cloud Native Applications译:第一部分云本地应用程序

Cloud Native是一种应用程序开发风格，鼓励在持续交付和价值驱动开发领域轻松采用最佳实践。 一个相关的学科是建设12-factor Applications，其中开发实践与交付和运营目标相一致 - 例如，通过使用声明式编程和管理以及监控。 Spring Cloud以许多特定的方式促进了这些开发风格。 起点是分布式系统中的所有组件都需要轻松访问的一组功能。

Spring Cloud构建的Spring Boot涵盖了其中的许多功能。 Spring Cloud还提供了两个库：Spring Cloud Context和Spring Cloud Commons。 Spring Cloud Context为Spring Cloud应用程序（引导上下文，加密，刷新范围和环境端点）的`ApplicationContext`提供实用程序和特殊服务。 Spring Cloud Commons是一组用于不同Spring Cloud实现（如Spring Cloud Netflix和Spring Cloud Consul）的抽象和常用类。

如果由于“非法密钥大小”而导致异常并且您使用Sun的JDK，则需要安装Java加密扩展（JCE）无限制强制管辖权策略文件。 有关更多信息，请参阅以下链接：

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

将文件解压缩到您使用的JRE / JDK x64 / x86版本的JDK / jre / lib / security文件夹中。

Spring Cloud是在非限制性的Apache 2.0许可下发布的。 如果您想为本文档的这一部分做出贡献，或者如果您发现错误，您可以在github找到项目的源代码和问题跟踪器。

2. Spring Cloud Context: Application Context Services译: 2. Spring Cloud上下文：应用程序上下文服务

Spring Boot对如何使用Spring构建应用程序有着自己的观点。 例如，它具有常规配置文件的常规位置，并具有用于常见管理和监视任务的端点。 Spring云建立在此基础之上，并添加了一些可能系统中所有组件都会使用或偶尔需要的功能。

2.1 The Bootstrap Application Context译: 2.1 Bootstrap应用程序上下文

Spring Cloud应用程序通过创建一个“bootstrap”上下文来运行，该上下文是主应用程序的父上下文。 它负责从外部源加载配置属性，并负责解密本地外部配置文件中的属性。 这两个上下文共享一个`Environment`，这是任何Spring应用程序的外部属性的来源。 默认情况下，引导属性（不是`bootstrap.properties`但是在引导阶段加载的属性）以高优先级添加，因此它们不能被本地配置覆盖。

引导程序上下文使用不同的约定来定位外部配置，而不是主应用程序上下文。 而不是`application.yml`（或`.properties`），您可以使用`bootstrap.yml`，保持bootstrap的外部配置和主环境很好地分离。 以下列表显示了一个示例：

bootstrap.yml。

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

如果您的应用程序需要来自服务器的任何特定于应用程序的配置，则最好设置`spring.application.name`（`bootstrap.yml`或`application.yml`）。

您可以通过设置 `spring.cloud.bootstrap.enabled=false` （例如，在系统属性中）完全禁用引导过程。

2.2 Application Context Hierarchies译：2.2应用程序上下文层次结构

如果您从`SpringApplication`或`SpringApplicationBuilder`构建应用程序上下文，则引导程序上下文将作为该上下文的父级添加。 Spring的一个特性是子级上下文从父级继承属性来源和配置文件，因此与不使用Spring Cloud Config构建相同上下文相比，“main”应用程序上下文包含其他属性来源。 额外的财产来源是：

- “bootstrap”: If any `PropertySourceLocators` are found in the Bootstrap context and if they have non-empty properties, an optional `CompositePropertySource` appears with high priority. An example would be properties from the Spring Cloud Config Server. See “Section 2.6, “Customizing the Bootstrap Property Sources”” for instructions on how to customize the contents of this property source.
- “applicationConfig: [classpath:bootstrap.yml]” (and related files if Spring profiles are active): If you have a `bootstrap.yml` (or `.properties`), those properties are used to configure the Bootstrap context. Then they get added to the child context when its parent is set. They have lower precedence than the `application.yml` (or `.properties`) and any other property sources that are added to the child as a normal part of the process of creating a Spring Boot application. See “Section 2.3, “Changing the Location of Bootstrap Properties”” for instructions on how to customize the contents of these property sources.

由于属性来源的排序规则，“bootstrap”条目优先。 但是请注意，这些数据不包含`bootstrap.yml`中的任何数据，它们的优先级非常低，但可用于设置默认值。

您可以通过设置任何父上下文扩展上下文结构`ApplicationContext`你createâ€‰€”â€‰例如，通过使用自己的接口或与`SpringApplicationBuilder`方便的方法（`parent()`，`child()`和`sibling()`）。 引导程序上下文是您自己创建的最高级祖先的父级。 层次结构中的每个上下文都有自己的“底层”（可能为空）属性来源，以避免无意中将价值从父母推到其后代。 如果有配置服务器，则层次结构中的每个上下文也可以（原则上）具有不同的`spring.application.name`，因此也是不同的远程属性源。 普通的Spring应用上下文行为规则适用于属性解析：来自子上下文的属性覆盖父类中的属性，按名称和属性源名称。 （如果孩子的姓名与父母姓名相同，则父母的值不包含在孩子中）。

请注意，`SpringApplicationBuilder`可让您在整个层次结构中共享`Environment`，但这不是默认设置。 因此，兄弟情境尤其不需要具有相同的概况或财产来源，即使它们可能与其父母分享共同的价值。

2.3 Changing the Location of Bootstrap Properties译：2.3更改引导属性的位置

例如，在系统属性中，可以通过设置`spring.cloud.bootstrap.name`（默认值：`bootstrap`）或`spring.cloud.bootstrap.location`（默认值：空）来指定`bootstrap.yml`（或`.properties`）位置。 这些属性的行为与具有相同名称的`spring.config.*`变体类似。 实际上，它们用于通过在`ApplicationContext`设置这些属性来设置引导程序`Environment`。 如果有活动的配置文件（来自`spring.profiles.active`或通过您正在构建的上下文中的`Environment` API），那么该配置文件中的属性也会加载，与常规的Spring Boot应用程序相同 - 例如从`bootstrap-development.properties`获取`development`个人资料。

2.4 Overriding the Values of Remote Properties译：2.4覆盖远程属性的值

通过引导上下文添加到应用程序的属性源通常是“远程”（来自Spring Cloud Config Server的示例）。 默认情况下，它们不能在本地覆盖。 如果您想让应用程序使用自己的系统属性或配置文件覆盖远程属性，那么远程属性源必须通过设置`spring.cloud.config.allowOverride=true`（无法在本地进行设置）授予其权限。 一旦设置了该标志，两个更细粒度的设置将控制远程属性相对于系统属性和应用程序的本地配置的位置：

- `spring.cloud.config.overrideNone=true`: Override from any local property source.
- `spring.cloud.config.overrideSystemProperties=false`: Only system properties, command line arguments, and environment variables (but not the local config files) should override the remote settings.

2.5 Customizing the Bootstrap Configuration译：2.5自定义引导配置

通过在名为`org.springframework.cloud.bootstrap.BootstrapConfiguration`的密钥下向`/META-INF/spring.factories`添加条目，可以将引导程序上下文设置为执行任何您喜欢的`org.springframework.cloud.bootstrap.BootstrapConfiguration`。 这包含用于创建上下文的Spring `@Configuration`类的逗号分隔列表。 您可以在此处创建任何要用于自动装配的主应用程序上下文的bean。 `@Beans`类型的`ApplicationContextInitializer`有一个特殊的合同。 如果要控制启动顺序，可以使用`@Order`注释标记类（默认顺序为`last`）。

当添加自定义`BootstrapConfiguration`，小心你添加类不是`@ComponentScanned`误到您的A€œmainâ€应用程序上下文，这里可能并不需要它们。 为引导配置类使用单独的软件包名称，并确保该名称未被您的`@ComponentScan`或`@SpringBootApplication`注释的配置类所覆盖。

引导进程通过注入初始化到主结束`SpringApplication`实例（这是一般的弹簧引导启动序列，无论是运行作为独立应用程序或部署在应用服务器）。 首先，从`spring.factories`的类中创建引导程序上下文。 然后，所有`@Beans`型`ApplicationContextInitializer`被添加到主`SpringApplication`它开始之前。

2.6 Customizing the Bootstrap Property Sources译：2.6自定义引导属性源

通过引导进程添加的外部配置的默认属性源是Spring Cloud Config Server，但可以通过将`PropertySourceLocator`类型的`PropertySourceLocator`添加到引导上下文（通过`spring.factories`）来添加其他源。 例如，您可以从其他服务器或数据库插入其他属性。

作为一个例子，考虑下面的自定义定位器：

```
@Configuration
public class CustomPropertySourceLocator implements PropertySourceLocator {

    @Override
    public PropertySource<?> locate(Environment environment) {
        return new MapPropertySource("customProperty",
                Collections.<String, Object>singletonMap("property.from.sample.custom.source", "worked as intended"));
    }

}
```

该`Environment`传递进来的是一个为`ApplicationContext`即将被createdâ€‰€”â€‰换句话说，对于我们供应附加属性源之一。 它已经有了普通的Spring Boot提供的属性资源，因此您可以使用它们来定位特定于此`Environment`的属性源（例如，通过在`spring.application.name`上键入它，如在默认的Spring Cloud Config Server属性源定位器中所做的那样）。

如果你创建一个罐子这个类，然后添加一个`META-INF/spring.factories`包含以下，`customProperty``PropertySource`出现在任何应用程序，包括在其类路径的jar：

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPropertySourceLocator
```

2.7 Logging Configuration译：2.7记录配置

如果你打算使用Spring Boot来配置日志设置，那么你应该把这个配置放在`bootstrap。[yml |中 属性]，如果你想它适用于所有事件。

2.8 Environment Changes译：2.8环境变化

应用程序监听`EnvironmentChangeEvent`并以一些标准方式对更改作出反应（`ApplicationListeners`可以通过正常方式将其他`ApplicationListeners`添加为`@Beans`）。 当观察到`EnvironmentChangeEvent`，它具有已更改的键值列表，并且应用程序使用它们来：

- Re-bind any `@ConfigurationProperties` beans in the context
- Set the logger levels for any properties in `logging.level.*`

请注意，默认情况下，Config Client不会轮询`Environment`中的`Environment`。 一般来说，我们不会推荐用于检测更改的方法（尽管您可以使用`@Scheduled`注释进行设置）。 如果您有扩展的客户端应用程序，则最好将`EnvironmentChangeEvent`广播到所有实例，而不是让他们轮询更改（例如，通过使用Spring Cloud Bus ）。

`EnvironmentChangeEvent`涵盖了一大类刷新用例，只要您可以实际更改`Environment`并发布该事件。 请注意，这些API是公开的，并且是核心Spring的一部分）。 您可以通过访问`/configprops`端点（普通的Spring Boot执行器功能）来验证更改是否与`@ConfigurationProperties` Bean绑定。 例如， `DataSource`可以在运行时更改其`maxPoolSize`（由Spring Boot创建的默认`DataSource`是`@ConfigurationProperties` bean）并动态增长容量。 重新绑定`@ConfigurationProperties`不包含另一大类用例，您需要对刷新进行更多控制，并且需要对整个`ApplicationContext`进行原子级更改。 为了解决这些问题，我们有`@RefreshScope`。

2.9 Refresh Scope译：2.9刷新范围

当有配置更改，弹簧`@Bean`标记为`@RefreshScope`得到特殊待遇。 该功能解决了初始化时仅注入配置的有状态Bean的问题。 例如，如果`DataSource`在通过`Environment`更改数据库URL时已打开连接，则可能希望这些连接的持有者能够完成他们正在执行的操作。 然后，下一次从池中借用一个连接时，它将获得一个新的URL。

有时，甚至可能必须将`@RefreshScope`注释应用于一些只能初始化一次的bean。 如果一个bean是“不可变的”，你必须用`@RefreshScope`注释这个bean或者在属性关键字`spring.cloud.refresh.extra-refreshable`下面指定类`spring.cloud.refresh.extra-refreshable`。

刷新范围bean是使用它们进行初始化的惰性代理（即，调用方法时），范围充当初始化值的缓存。 要强制一个bean在下一次方法调用时重新初始化，必须使其缓存项无效。

`RefreshScope`是上下文中的bean，具有公共方法`refreshAll()`，通过清除目标缓存来刷新作用域中的所有Bean。 `/refresh`端点公开了此功能（通过HTTP或JMX）。 要按名称刷新单个bean，还有一个`refresh(String)`方法。

要公开`/refresh`端点，您需要将以下配置添加到您的应用程序中：

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

`@RefreshScope`（技术上）在`@Configuration`类上工作，但可能会导致令人惊讶的行为。 例如，这并不意味着`@Beans`定义的所有`@Beans`本身都在`@RefreshScope`。 具体来说，除非它本身在`@RefreshScope`，否则依赖于这些bean的任何东西都不能依赖它们在刷新时被更新。 在这种情况下，它将在刷新时重建，并且它的依赖关系会被重新注入。 此时，它们从刷新的`@Configuration`重新初始化）。

2.10 Encryption and Decryption译：2.10加密和解密

Spring Cloud拥有用于本地解密属性值的`Environment`预处理器。 它遵循与配置服务器相同的规则，并具有通过`encrypt.*`的相同外部配置。 因此，您可以使用`{cipher}*`形式的加密值，并且只要有一个有效的密钥，它们在主应用程序上下文获取`Environment`设置之前被解密。 要在应用程序中使用加密功能，您需要在您的类路径中包含Spring Security RSA（Maven协调：“org.springframework.security:spring-security-rsa”），并且您还需要完整强度的JCE扩展你的JVM。

如果由于“非法密钥大小”而导致异常并且您使用Sun的JDK，则需要安装Java加密扩展（JCE）无限制强制管辖权策略文件。 有关更多信息，请参阅以下链接：

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

将文件解压缩到您使用的JRE / JDK x64 / x86版本的JDK / jre / lib / security文件夹中。

2.11 Endpoints译：2.11终点

对于Spring Boot Actuator应用程序，可以使用其他一些管理端点。 您可以使用：

- `POST` to `/actuator/env` to update the `Environment` and rebind `@ConfigurationProperties` and log levels.
- `/actuator/refresh` to re-load the boot strap context and refresh the `@RefreshScope` beans.
- `/actuator/restart` to close the `ApplicationContext` and restart it (disabled by default).
- `/actuator/pause` and `/actuator/resume` for calling the `Lifecycle` methods (`stop()` and `start()` on the `ApplicationContext`).

如果禁用`/actuator/restart`端点，那么`/actuator/pause`和`/actuator/resume`端点也将被禁用，因为它们只是`/actuator/restart`。

3. Spring Cloud Commons: Common Abstractions译：3.春云共享：共同抽象

诸如服务发现，负载均衡和断路器等模式适合于所有Spring Cloud客户端都可以使用的通用抽象层，而与实现无关（例如，使用Eureka或Consul进行发现）。

3.1 @EnableDiscoveryClient译：3.1 @EnableDiscoveryClient

Spring Cloud Commons提供了`@EnableDiscoveryClient`注释。 这将查找与`META-INF/spring.factories`的`DiscoveryClient`接口的`META-INF/spring.factories`。 在发现客户端的实现添加配置类`spring.factories`下`org.springframework.cloud.client.discovery.EnableDiscoveryClient`关键。 的实例`DiscoveryClient`实现包括Spring Cloud Netflix Eureka， Spring Cloud Consul Discovery，和Spring Cloud Zookeeper Discovery。

默认情况下， `DiscoveryClient`实现自动注册本地Spring Boot服务器和远程发现服务器。 此行为可通过在`@EnableDiscoveryClient`设置`autoRegister=false`来禁用。

`@EnableDiscoveryClient`不再需要。 您可以在类路径上放置`DiscoveryClient`实现，以使Spring Boot应用程序向服务发现服务器注册。

3.1.1 Health Indicator 译：3.1.1健康指标

共享创建一个春天引导`HealthIndicator`是`DiscoveryClient`实现可以通过实施参与`DiscoveryHealthIndicator`。 要禁用复合`HealthIndicator`，请设置`spring.cloud.discovery.client.composite-indicator.enabled=false`。 自动配置基于`DiscoveryClient`通用`HealthIndicator`（`DiscoveryClientHealthIndicator`）。 要禁用它，请设置`spring.cloud.discovery.client.health-indicator.enabled=false`。 要禁用`DiscoveryClientHealthIndicator`的说明字段，请设置`spring.cloud.discovery.client.health-indicator.include-description=false`。 否则，它可以冒泡为`description`的卷起的`HealthIndicator`。

3.2 ServiceRegistry 译：3.2 ServiceRegistry

Commons现在提供了一个`ServiceRegistry`接口，该接口提供了诸如`register(Registration)`和`deregister(Registration)`，它们允许您提供自定义注册服务。 `Registration`是标记界面。

以下示例显示正在使用的`ServiceRegistry`：

```
@Configuration
@EnableDiscoveryClient(autoRegister=false)
public class MyConfiguration {
    private ServiceRegistry registry;

    public MyConfiguration(ServiceRegistry registry) {
        this.registry = registry;
    }

    // called through some external process, such as an event or a custom actuator endpoint
    public void register() {
        Registration registration = constructRegistration();
        this.registry.register(registration);
    }
}
```

每个`ServiceRegistry`实现都有其自己的`Registry`实现。

- `ZookeeperRegistration` used with `ZookeeperServiceRegistry`
- `EurekaRegistration` used with `EurekaServiceRegistry`
- `ConsulRegistration` used with `ConsulServiceRegistry`

如果您使用的是`ServiceRegistry`接口，则需要为您正在使用的`ServiceRegistry`实现传递正确的`Registry`实现。

3.2.1 ServiceRegistry Auto-Registration 译：3.2.1 ServiceRegistry自动注册

默认情况下， `ServiceRegistry`实现会自动注册正在运行的服务。 要禁用该行为，您可以设置：* `@EnableDiscoveryClient(autoRegister=false)`以永久禁用自动注册。 * `spring.cloud.service-registry.auto-registration.enabled=false`通过配置禁用行为。

3.2.2 Service Registry Actuator Endpoint 译：3.2.2服务注册执行器端点

Spring Cloud Commons提供了一个`/service-registry`执行器端点。 该端点依赖于Spring应用程序上下文中的`Registration` bean。 调用`/service-registry`用GET返回的状态`Registration`。 将POST用于具有JSON主体的相同端点会将当前`Registration`的状态更改为新值。 JSON正文必须包含具有首选值的`status`字段。 在更新状态和返回状态值时，请参阅`ServiceRegistry`实施的文档，以`ServiceRegistry`允许的值。 例如，Eurekaâ€™的支持状态为`UP`， `DOWN`，`OUT_OF_SERVICE`，并`UNKNOWN`。

3.3 Spring RestTemplate as a Load Balancer Client 译：3.3 Spring RestTemplate作为负载平衡器客户端

`RestTemplate`可以自动配置为使用功能区。 要创建负载平衡`RestTemplate`，请创建`RestTemplate` `@Bean`并使用`@LoadBalanced`限定符，如下例所示：

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results = restTemplate.getForObject("http://stores/stores", String.class);
        return results;
    }
}
```

Caution

一个`RestTemplate` bean不再通过自动配置创建。 个人应用程序必须创建它。

该URI需要使用虚拟主机名称（即服务名称，而不是主机名称）。 功能区客户端用于创建完整的物理地址。 见RibbonAutoConfiguration为的是如何细节`RestTemplate`设置。

3.4 Spring WebClient as a Load Balancer Client译：3.4 Spring WebClient作为负载平衡器客户端

`WebClient`可以自动配置为使用`LoadBalancerClient`。 要创建负载平衡的`WebClient`，请创建`WebClient.Builder` `@Bean`并使用`@LoadBalanced`限定符，如下例所示：

```
@Configuration
public class MyConfiguration {

 @Bean
 @LoadBalanced
 public WebClient.Builder loadBalancedWebClientBuilder() {
  return WebClient.builder();
 }
}

public class MyClass {
    @Autowired
    private WebClient.Builder webClientBuilder;

    public Mono<String> doOtherStuff() {
        return webClientBuilder.build().get().uri("http://stores/stores")
            .retrieve().bodyToMono(String.class);
    }
}
```

该URI需要使用虚拟主机名称（即服务名称，而不是主机名称）。 功能区客户端用于创建完整的物理地址。

3.4.1 Retrying Failed Requests译：3.4.1重试失败的请求

负载均衡`RestTemplate`可以配置为重试失败的请求。 默认情况下，该逻辑被禁用。 您可以通过将Spring Retry添加到应用程序的类路径来启用它。 负载均衡`RestTemplate`尊重与重试失败请求相关的某些功能区配置值。 您可以使用`client.ribbon.MaxAutoRetries`， `client.ribbon.MaxAutoRetriesNextServer`，并`client.ribbon.OkToRetryOnAllOperations`性能。 如果您希望在类路径中使用Spring重试禁用重试逻辑，则可以设置`spring.cloud.loadbalancer.retry.enabled=false`。 请参阅Ribbon documentation以了解这些属性的用途。

如果您想在重试中实现`BackOffPolicy`，则需要创建类型为`LoadBalancedBackOffPolicyFactory`的bean，并返回您希望用于给定服务的`BackOffPolicy`，如以下示例所示：

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedBackOffPolicyFactory backOffPolciyFactory() {
        return new LoadBalancedBackOffPolicyFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
          return new ExponentialBackOffPolicy();
         }
        };
    }
}
```

上述示例中的`client`应替换为您的Ribbon客户端的名称。

如果要将一个或多个`RetryListener`实现添加到重试功能中，则需要创建类型为`LoadBalancedRetryListenerFactory`的bean，并返回您希望用于给定服务的`RetryListener`阵列，如以下示例所示：

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryListenerFactory retryListenerFactory() {
        return new LoadBalancedRetryListenerFactory() {
            @Override
            public RetryListener[] createRetryListeners(String service) {
                return new RetryListener[]{new RetryListener() {
                    @Override
                    public <T, E extends Throwable> boolean open(RetryContext context, RetryCallback<T, E> callback) {
                        //TODO Do you business...
                        return true;
                    }

                    @Override
                     public <T, E extends Throwable> void close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }

                    @Override
                    public <T, E extends Throwable> void onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }
                }};
            }
        };
    }
}
```

3.5 Multiple RestTemplate objects译：3.5多个RestTemplate对象

如果你想要一个`RestTemplate`未负载平衡，营造`RestTemplate` bean并注入它。 要访问负载平衡的`RestTemplate`，请在创建`@Bean`时使用`@LoadBalanced`限定符，如以下示例所示：\

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate loadBalanced() {
        return new RestTemplate();
    }

    @Primary
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    @LoadBalanced
    private RestTemplate loadBalanced;

    public String doOtherStuff() {
        return loadBalanced.getForObject("http://stores/stores", String.class);
    }

    public String doStuff() {
        return restTemplate.getForObject("http://example.com", String.class);
    }
}
```

Important

请注意，在前面的示例中， `@Primary`注释在 `RestTemplate`声明中的使用可以消除不合格的 `@Autowired`注入的歧义。

如果您看到错误（例如 `java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.pr` ，请尝试注入 `RestOperations`或设置 `spring.aop.proxyTargetClass=true` 。

3.6 Spring WebFlux WebClient as a Load Balancer Client 译: 3.6 Spring WebFlux WebClient作为负载平衡器客户端

`WebClient`可以配置为使用 `LoadBalancerClient` 。 `LoadBalancerExchangeFilterFunction`自动配置如果 `spring-webflux`是在classpath。 以下示例显示如何配置 `WebClient`以使用负载均衡器：

```
public class MyClass {
    @Autowired
    private LoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

该URI需要使用虚拟主机名称（即服务名称，而不是主机名称）。 `LoadBalancerClient`用于创建完整的物理地址。

3.7 Ignore Network Interfaces 译: 3.7忽略网络接口

有时候，忽略某些命名的网络接口是有用的，这样可以将它们从服务发现注册中排除（例如，在Docker容器中运行时）。 正则表达式列表可以设置为使所需的网络接口被忽略。 以下配置将忽略 `docker0`接口以及以 `veth`开头的所有接口：

application.yml。

```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

您还可以使用正则表达式列表强制仅使用指定的网络地址，如下例所示：

bootstrap.yml。

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

您还可以强制使用站点本地地址，如以下示例所示：.application.yml

```
spring:
  cloud:
    inetutils:
      useOnlySiteLocalInterfaces: true
```

有关构成站点本地地址的更多详细信息，请参阅 Inet4Address.html.isSiteLocalAddress() 。

3.8 HTTP Client Factories译：3.8 HTTP客户端工厂

Spring Cloud Commons提供了用于创建Apache HTTP客户端（ `ApacheHttpClientFactory` ）和OK HTTP客户端（ `OkHttpClientFactory` ）的`OkHttpClientFactory` 。 仅当OK HTTP jar位于类路径中时，才会创建`OkHttpClientFactory` bean。 另外，Spring Cloud Commons提供了用于创建两个客户端使用的连接管理器的bean： `ApacheHttpClientConnectionManagerFactory`用于Apache HTTP客户端， `OkHttpClientConnectionPoolFactory`用于OK HTTP客户端。 如果您想定制在下游项目中如何创建HTTP客户端，则可以提供您自己的这些bean的实现。 另外，如果您提供`HttpClientBuilder`或`OkHttpClient.Builder`类型的bean，默认工厂会使用这些构建器作为构建器返回到下游项目的基础。 您还可以通过设置来禁用这些bean创建`spring.cloud.httpclientfactories.apache.enabled`或者`spring.cloud.httpclientfactories.ok.enabled`至`false` 。

3.9 Enabled Features译：3.9启用的功能

Spring Cloud Commons提供了一个`/features`执行器端点。 此端点返回类路径上可用的功能以及它们是否已启用。 返回的信息包括功能类型，名称，版本和供应商。

3.9.1 Feature types译：3.9.1特征类型

有两种类型的“功能”：抽象和命名。

抽象特征是其中的接口或抽象类定义，并且一个实施创建功能，如`DiscoveryClient`， `LoadBalancerClient` ，或`LockService` 。 抽象类或接口用于在上下文中查找该类型的bean。 显示的版本是`bean.getClass().getPackage().getImplementationVersion()` 。

已命名的功能是没有实现特定类的功能，例如“断路器”，“API网关”，“春季云总线”等。 这些功能需要一个名称和一个bean类型。

3.9.2 Declaring features译：3.9.2声明特征

任何模块都可以声明任意数量的 `HasFeature` bean，如以下示例中所示：

```
@Bean
public HasFeatures commonsFeatures() {
  return HasFeatures.abstractFeatures(DiscoveryClient.class, LoadBalancerClient.class);
}

@Bean
public HasFeatures consulFeatures() {
  return HasFeatures.namedFeatures(
    new NamedFeature("Spring Cloud Bus", ConsulBusAutoConfiguration.class),
    new NamedFeature("Circuit Breaker", HystrixCommandAspect.class));
}

@Bean
HasFeatures localFeatures() {
  return HasFeatures.builder()
      .abstractFeature(Foo.class)
      .namedFeature(new NamedFeature("Bar Feature", Bar.class))
      .abstractFeature(Baz.class)
      .build();
}
```

这些豆中的每一个都应该进入适当的守卫 `@Configuration` 。

Part II. Spring Cloud Config译：第二部分。 Spring Cloud配置

Finchley.RELEASE

Spring Cloud Config为分布式系统中的外部配置提供服务器端和客户端支持。 通过Config Server，您可以在所有环境中管理应用程序的外部属性。 客户端和服务器上的概念与Spring `Environment`和`PropertySource`抽象图完全相同，因此它们非常适合Spring应用程序，但可以与任何使用任何语言的应用程序一起使用。 随着应用程序从开发到测试转移到生产环境中，您可以管理这些环境之间的配置，并确保应用程序拥有迁移时所需的所有内容。 服务器存储后端的默认实现使用git，因此它很容易支持配置环境的标签版本，并可用于管理内容的各种工具。 使用Spring配置很容易添加替代实现并将其插入。

4. Quick Start译：4.快速入门

本快速入门介绍了如何使用Spring Cloud Config Server的服务器和客户端。

首先，启动服务器，如下所示：

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

该服务器是一个Spring Boot应用程序，所以如果您愿意，您可以从IDE运行它（主类为 `ConfigServerApplication` ）。

接下来尝试一个客户端，如下所示：

```
$ curl localhost:8888/foo/development
{"name":"foo","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}
]}
```

定位属性源的默认策略是克隆git仓库（在`spring.cloud.config.server.git.uri` ）并使用它来初始化一个迷你`SpringApplication` 。 迷你应用程序

的`Environment`用于枚举属性源并将它们发布到JSON端点。

HTTP服务具有以下形式的资源：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

其中`application`作为`spring.config.name`在`SpringApplication`（通常`application`在常规Spring Boot应用程序中）`profile`，`profile`是活动配置文件（或逗号分隔的属性列表），`label`是可选git标签（默认为`master`。）

Spring Cloud Config Server从git仓库（必须提供）中为远程客户端提取配置，如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

4.1 Client Side Usage译：4.1客户端使用情况

要在应用程序中使用这些功能，可以将其构建为依赖于spring-cloud-config-client的Spring Boot应用程序（例如，请参阅config-client或示例应用程序的测试用例）。 添加依赖项的最方便的方法是使用Spring Boot启动器`org.springframework.cloud:spring-cloud-starter-config`。 对于Maven用户还有一个父pom和BOM（`spring-cloud-starter-parent`），以及Gradle和Spring CLI用户的Spring IO版本管理属性文件。 以下示例显示了一个典型的Maven配置：

pom.xml中。

```
   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>{spring-boot-docs-version}</version>
       <relativePath /> <!-- lookup parent from repository -->
   </parent>

<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>{spring-cloud-version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>

<dependencies>
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
 </dependency>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
 </dependency>
</dependencies>

<build>
 <plugins>
          <plugin>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
 </plugins>
</build>

   <!-- repositories also needed for snapshots and milestones -->
```

现在您可以创建一个标准的Spring Boot应用程序，例如以下HTTP服务器：

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

当此HTTP服务器运行时，它会从端口8888上的默认本地配置服务器（如果它正在运行）中获取外部配置。要修改启动行为，可以使用`bootstrap.properties`（类似于`application.properties`更改配置服务器的位置但是用于应用程序上下文的引导阶段），如以下示例所示：

```
spring.cloud.config.uri: http://myconfigserver.com
```

引导属性在`/env`端点中显示为高优先级属性源，如以下示例所示。

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

名为 ``configService:<URL of remote repository>/<file name>`` 的财产来源包含值为 `bar` 的 `foo` 财产，并且是最高优先级。

属性源名称中的URL是git存储库，而不是配置服务器URL。

5. Spring Cloud Config Server 译：5. Spring Cloud Config Server

Spring Cloud Config Server为外部配置（名称 - 值对或同等YAML内容）提供基于HTTP资源的API。 通过使用 `@EnableConfigServer` 注释，服务器可嵌入到Spring Boot应用程序中。 因此，以下应用程序是一个配置服务器：

ConfigServer.java。

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

像所有的Spring Boot应用程序一样，默认情况下它在端口8080上运行，但您可以通过各种方式将其切换到更传统的端口8888。 最简单的，这也设置默认配置库，是通过启动它 `spring.config.name=configserver` （有一个 `configserver.yml` 在配置服务器JAR）。 另一种方法是使用自己的 `application.properties` ，如以下示例所示：

application.properties。

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

其中 `${user.home}/config-repo` 是一个包含YAML和属性文件的git存储库。

在Windows上，如果文件URL是绝对驱动器前缀（例如 `file:///${user.home}/config-repo` ），则需要在文件URL中添加一个额外的“/”。

以下列表显示了在上例中创建git存储库的配方：

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

为您的git存储库使用本地文件系统仅用于测试。 您应该使用服务器来托管生产中的配置存储库。

如果只保留文本文件，配置库的初始克隆可以快速有效。 如果您存储二进制文件，尤其是大型文件，则可能会在首次配置请求时遇到延迟，或者遇到服务器内存不足错误。

5.1 Environment Repository 译：5.1环境信息库

你应该在哪里存储配置服务器的配置数据？ 管理此行为的策略是 `EnvironmentRepository` ，服务 `Environment` 对象。 此 `Environment` 是Spring `Environment` （包括 `propertySources` 作为主要功能）的域的浅表副本。 `Environment` 资源通过三个变量进行参数化：

- `{application}` , which maps to `spring.application.name` on the client side.
- `{profile}` , which maps to `spring.profiles.active` on the client (comma-separated list).
- `{label}` , which is a server side feature labelling a "versioned" set of config files.

版本库实现通 `spring.config.name` Spring Boot应用程序，从 `spring.config.name` 加载等于 `{application}` 参数的配置文件， `spring.profiles.active` 等于 `{profiles}` 参数。 配置文件的优先级规则也与普通的Spring Boot应用程序相同：活动配置文件优先于默认配置文件，如果有多个配置文件，则最后一个配置文件会胜出（类似于将条目添加到 `Map` ）。

以下示例客户端应用程序具有此引导程序配置：

bootstrap.yml。

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

（像通常的Spring Boot应用程序一样，这些属性也可以通过环境变量或命令行参数来设置）。

如果存储库是基于文件的，服务器创建一个 `Environment` 从 `application.yml` （所有客户间共享）， `foo.yml` （与 `foo.yml` 采取优先次序）。 如果YAML文件内部有文档指向Spring配置文件，那么这些文件将以更高的优先级应用（按列出的配置文件的顺序）。 如果存在特定于配置文件的YAML（或属性）文件，则这些文件的优先级高于默认值。 较高的优先级转换为 `PropertySource` 较早列出的 `Environment` 。 （这些相同的规则适用于独立的Spring Boot应用程序。）

您可以将spring.cloud.config.server.accept-empty设置为false，以便在未找到应用程序时Server将返回HTTP 404状态。默认情况下，此标志设置为true。

5.1.1 Git Backend译：5.1.1 Git后端

默认实现`EnvironmentRepository`使用Git后端，这对管理升级和物理环境以及审核更改非常方便。 要更改存储库的位置，可以在配置服务器中设置`spring.cloud.config.server.git.uri`配置属性（例如，在`application.yml`）。 如果您使用`file:`前缀进行设置，它应该从本地存储库运行，以便无需服务器即可快速轻松地启动。 但是，在这种情况下，服务器直接在本地存储库上进行操作，而不克隆它（因为配置服务器从不对“远程”存储库进行更改，所以它不是裸露的并不重要）。 要扩展配置服务器并使其具有高可用性，您需要将服务器的所有实例指向同一个存储库，因此只有共享文件系统才能工作。 即使在这种情况下，最好对共享文件系统存储库使用`ssh:`协议，以便服务器可以克隆它并使用本地工作副本作为缓存。

此存储库实现将HTTP资源的`{label}`参数映射到git标签（提交标识，分支名称或标记）。 如果git分支或标签名称包含斜杠（`/`），则应该使用特殊字符串`(_)`（为了避免与其他URL路径`(_)`指定HTTP URL中的标签。 例如，如果标签为`foo/bar`，则替换斜杠将生成以下标签：`foo(_)bar`。 包含特殊字符串`(_)`也可以应用于`{application}`参数。 如果使用诸如curl这样的命令行客户端，请小心URL中的括号，否则应该用单引号（"）将它们从shell中转义出来。

Skipping SSL Certificate Validation译:跳过SSL证书验证

通过将`git.skipSslValidation`属性设置为`true`（默认值为`false`），可以禁用配置服务器验证Git服务器的SSL证书。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          skipSslValidation: true
```

Setting HTTP Connection Timeout译:设置HTTP连接超时

您可以配置配置服务器等待获取HTTP连接的时间（以秒为单位）。 使用`git.timeout`属性。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          timeout: 4
```

Placeholders in Git URI译：Git URI中的占位符

Spring Cloud Config Server支持包含`{application}`和`{profile}`（和`{label}`如果需要的话）的占位符的git存储库URL，但请记住该标签仍然作为git标签应用。 因此，您可以通过使用类似于以下内容的结构来支持“每个应用程序的一个存储库”策略：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

您也可以通过使用类似的模式支持“每个配置文件一个存储库”策略，但支持`{profile}`。

此外，在`{application}`参数中使用特殊字符串“（_）”可以支持多个组织，如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/{application}
```

其中`{application}`在请求时按以下格式提供：`organization(_)application`。

Pattern Matching and Multiple Repositories译:模式匹配和多个存储库

Spring Cloud Config还支持更复杂的需求，并在应用程序和配置文件名称上进行模式匹配。 模式格式是带有通配符的`{application}/{profile}`名称的逗号分隔列表（请注意，可能需要引用以通配符开头的模式），如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

如果`{application}/{profile}`与任何模式都不匹配，则使用`spring.cloud.config.server.git.uri`下定义的默认URI。 在上面的例子中，对于“简单”存储库，模式是`simple/*`（它只匹配所有配置文件中名为`simple`一个应用程序）。 “local”存储库匹配所有配置文件中以`local`开头的所有应用程序名称（`/*`后缀会自动添加到没有配置文件匹配器的任何模式）。

只有当要设置的唯一属性是URI时，才能使用“简单”示例中使用的“单线”快捷方式。 如果您需要设置其他任何内容（凭据，模式等），则需要使用完整的表单。

该`pattern`在回购属性实际上是一个数组，所以可以使用一个YAML阵列（或`[0]`，`[1]`等后缀在属性文件）绑定到多个图案。 如果要使用多个配置文件运行应用程

序，则可能需要这样做，如下例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

春云猜测包含的配置文件不结束模式`*`意味着你真的想匹配的开始这种模式的配置文件列表（所以`*/staging`是一条捷径`["*/staging", "*/staging,*"]`，依此类推）。 例如，您需要在本地“开发”配置文件中运行应用程序，而在远程运行“云”配置文件时，这也很常见。

每个存储库还可以选择将配置文件存储在子目录中，并且可以将指定为这些目录的模式指定为`searchPaths`。 以下示例显示了顶层的配置文件：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

在前面的示例中，服务器搜索顶层和`foo/`子目录中的配置文件以及名称以`bar`开头的任何子目录。

默认情况下，服务器首次请求配置时克隆远程存储库。 可以将服务器配置为在启动时克隆存储库，如以下顶层示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: http://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: http://git/team-b/config-repo.git
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

在前面的示例中，服务器在接受任何请求之前克隆组的一个启动时的配置回购。 除非请求存储库中的配置，否则所有其他存储库都不会被克隆。

在配置服务器启动时设置要克隆的存储库可以帮助在配置服务器启动时快速识别配置错误的配置源（例如无效的存储库URI）。 对于配置源未启用`cloneOnStart`，配置服务器可能会以错误配置或无效的配置源成功启动，并且在应用程序从该配置源请求配置之前不会检测到错误。

Authentication译:身份验证

要在远程存储库上使用HTTP基本身份验证，请单独添加`username`和`password`属性（不在URL中），如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

如果您不使用HTTPS和用户凭证，那么当您将密钥存储在默认目录（`~/.ssh`）并且URI指向SSH位置（例如`~/.ssh`）时，SSH也应该可以`git@github.com:configuration/cloud-configuration`。 `~/.ssh/known_hosts`文件中必须存在Git服务器的条目，并且格式为`ssh-rsa`。 其他格式（例如`ecdsa-sha2-nistp256`）不受支持。 为避免意外，您应该确保Git服务器的`known_hosts`文件中只有一个条目存在，并且它与您提供给配置服务器的URL相匹配。 如果您在URL中使用主机名，则您希望在`known_hosts`文件中具有完全相同的（不是IP）。 该存储库通过使用JGit进行访问，因此您找到的任何文档都应该适用。 HTTPS代理设置可以在系统属性（`-Dhttps.proxyHost`和`-Dhttps.proxyPort`）中设置为`~/.git/config`或（以与其他任何JVM进程相同的方式）。

如果您不知道`~/.git`目录的位置，请使用`git config --global`来操作设置（例如，`git config --global http.sslVerify false`）。

Authentication with AWS CodeCommit译:使用AWS CodeCommit进行身份验证

Spring Cloud Config Server还支持AWS CodeCommit身份验证。 从命令行使用Git时，AWS CodeCommit使用身份验证助手。 此帮助程序未与JGit库一起使用，因此如果Git URI与AWS CodeCommit模式匹配，则会创建用于AWS CodeCommit的JGit CredentialProvider。 AWS CodeCommit URI遵循以下模式：//git-codecommit.${AWS_REGION} .amazonaws.com / ${repopath}。

如果您使用AWS CodeCommit URI提供用户名和密码，则它们必须是可以访问存储库的AWS accessKeyId and secretAccessKey。 如果您未指定用户名和密码，则使用AWS Default Credential Provider Chain检索accessKeyId和secretAccessKey。

如果您的Git URI与CodeCommit URI模式相匹配（如前所示），则必须在用户名和密码中或缺省凭证提供程序链支持的其中一个位置提供有效的AWS凭证。 AWS EC2实例可能使用IAM Roles for EC2 Instances 。

`aws-java-sdk-core` jar是一个可选的依赖项。 如果`aws-java-sdk-core` jar不在您的类路径中，则不会创建AWS Code Commit凭据提供程序，无论git服务器URI如何。

Git SSH configuration using properties 译:使用属性的Git SSH配置

默认情况下，当通过使用SSH URI连接到Git存储库时，Spring Cloud Config Server使用的JGit库使用SSH配置文件，例如`~/.ssh/known_hosts`和`/etc/ssh/ssh_config` 。 在Cloud Foundry等云环境中，本地文件系统可能是短暂的或不易访问的。 对于这些情况，可以使用Java属性设置SSH配置。 为了激活基于属性的SSH配置，必须将`spring.cloud.config.server.git.ignoreLocalSshSettings`属性设置为`true` ，如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
                      -----BEGIN RSA PRIVATE KEY-----
                      MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
                      IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF
                      ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud
                      1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i
                      oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
                      DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
                      fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b
                      BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG
                      EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj
                      5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8
                      +AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe
                      pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
                      ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
                      xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW
                      dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi
                      PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX
                      VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z
                      FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R
                      gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
                      VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV
                      cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee
                      KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
                      CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
                      q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
                      69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
                      -----END RSA PRIVATE KEY-----
```

下表介绍了SSH配置属性。

表5.1。 SSH配置属性

Property Name	Remarks
ignoreLocalSshSettings	如果`true` ，使用基于属性而不是基于文件的SSH配置。 必须设置为`spring.cloud.config.server.git.ignoreLocalSshSettings` ， **而不是**在存储库定义中。
专用密钥	有效的SSH私钥。 如果`ignoreLocalSshSettings`为true且Git URI为SSH格式，则必须设置。
hostKey	有效的SSH主机密钥。 如果`hostKeyAlgorithm`也被设置，则必须设置。
hostKeyAlgorithm	`ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521` 。 如果还设置了`hostKey`则必须设置。
strictHostKeyChecking	`true`或`false` 。 如果为false，则忽略主机密钥的错误。
knownHostsFile	自定义`.known_hosts`文件的位置。
preferredAuthentications	覆盖服务器认证方法的顺序。 如果服务器在`publickey`方法之前进行了键盘交互式身份验证，则这应该允许您避免登录提示。

Placeholders in Git Search Paths 译: Git搜索路径中的占位符

Spring Cloud Config Server还支持带有`{application}`和`{profile}` （以及`{label}`如果需要）的占位符的搜索路径，如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

上面的列表导致搜索存储库中与目录同名的文件（以及顶层）。 通配符在带占位符的搜索路径中也是有效的（搜索中包含任何匹配的目录）。

Force pull in Git Repositories 译:强制拉入Git存储库

如前所述，如果本地副本变脏（例如，OS进程更改文件夹内容），Spring Cloud Config Server就会对远程git存储库进行克隆，从而使Spring Cloud Config Server无法从远程存储库更新本地副本。

要解决此问题，有一个`force-pull`属性，如果本地副本太脏，则会使Spring Cloud Config Server强制从远程存储`force-pull`取，如以下示例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

如果您有多个存储库配置，则可 `force-pull` 每个存储库配置 `force-pull` 属性，如下例所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
                pattern: team-a-*
                uri: http://git/team-a/config-repo.git
                force-pull: true
            team-b:
                pattern: team-b-*
                uri: http://git/team-b/config-repo.git
                force-pull: true
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

`force-pull` 属性的默认值为 `false` 。

Deleting untracked branches in Git Repositories 译:在Git存储库中删除未跟踪的分支

由于Spring Cloud Config Server在检出分支到本地回购后（例如通过标签获取属性）拥有远程git存储库的克隆，它将永久保留此分支或直到下一个服务器重新启动（这会创建新的本地回购）。 所以可能会出现这样的情况：远程分支被删除，但它的本地副本仍然可用于获取。 如果Spring Cloud Config Server客户端服务以 `--spring.cloud.config.label=deletedRemoteBranch,master` 开头，它将从 `deletedRemoteBranch` 本地分支获取属性，但不会从 `master` 获取属性。

为了使本地存储库分支保持清洁并可以远程访问 - `deleteUntrackedBranches` 属性可以设置。 它将使Spring Cloud Config Server **强制**从本地存储库中删除未跟踪的分支。 例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          deleteUntrackedBranches: true
```

`deleteUntrackedBranches` 属性的默认值是 `false` 。

5.1.2 Version Control Backend Filesystem Use 译：5.1.2版本控制后端文件系统使用

通过基于VCS的后端（git，svn），文件被检出或克隆到本地文件系统。 默认情况下，它们放在系统临时目录中，前缀为 `config-repo-` 。 例如，在Linux上，它可能是 `/tmp/config-repo-<randomid>` 。 一些操作系统routinely clean out临时目录。 这可能会导致意外的行为，如缺少属性。 为避免此问题，请通过将 `spring.cloud.config.server.git.basedir` 或 `spring.cloud.config.server.svn.basedir` 设置为不驻留在系统临时结构中的目录来更改Config Server使用的目录。

5.1.3 File System Backend 译：5.1.3文件系统后端

配置服务器中还有一个“本地”配置文件，它不使用Git，而是从本地类路径或文件系统（任何想要指向 `spring.cloud.config.server.native.searchLocations` 静态URL）加载配置文件。 要使用本机配置文件，请使用 `spring.profiles.active=native` 启动配置服务器。

请记住为文件资源使用 `file:` 前缀（没有前缀的缺省通常是类路径）。 与任何Spring Boot配置一样，您可以嵌入 `${}` 风格的环境占位符，但请记住，Windows中的绝对路径需要额外的 `/` （例如 `file:///${user.home}/config-repo` ）。

`searchLocations` 的默认值与本地Spring Boot应用程序（即 `[classpath:/, classpath:/config, file:./, file:./config]` ）相同。 这不会将 `application.properties` 从服务器公开到所有客户端，因为服务器中存在的任何属性源在发送到客户端之前都会被删除。

文件系统后端非常适合快速入门和测试。 要在生产环境中使用它，您需要确保文件系统在Config服务器的所有实例中都可靠并共享。

搜索位置可以包含占位符 `{application}` ， `{profile}` ，并 `{label}` 。 通过这种方式，您可以分隔路径中的目录并选择一种对您有意义的策略（例如，每个应用程序的子目录或每个配置文件的子目录）。

如果不在搜索位置中使用占位符，则此存储库还会将HTTP资源的 `{label}` 参数追加到搜索路径的后缀中，因此属性文件将从每个搜索位置**和**与标签名称相同的子目录中加载（在Spring环境中标记的属性优先）。 因此，没有占位符的默认行为与添加以 `/{label}/` 结尾的搜索位置 `/{label}/` 。 例如， `file:/tmp/config` 与 `file:/tmp/config,file:/tmp/config/{label}` 相同。 这种行为可以通过设置 `spring.cloud.config.server.native.addLabelLocations=false` 来禁用。

5.1.4 Vault Backend 译：5.1.4 Vault后端

Spring Cloud Config Server也支持 Vault作为后端。

> 保险柜是安全访问机密的工具。 秘密是指您想要严格控制访问权限的任何内容，例如API密钥，密码，证书和其他敏感信息。 保险柜为任何机密提供统一的接口，同时提供严格的访问控制并记录详细的审计日志。

有关Vault的更多信息，请参阅 Vault quick start guide 。

要使配置服务器能够使用Vault后端，您可以使用`vault`配置文件运行配置服务器。 例如，在您的配置服务器的`application.properties`，您可以添加`spring.profiles.active=vault` 。

默认情况下，配置服务器假定您的Vault服务器运行在`http://127.0.0.1:8200` 。 它还假定后端的名称是`secret`，密钥是`application` 。 所有这些默认值都可以在你的配置服务器的`application.properties`配置。 下表介绍了可配置的Vault属性：

Name	Default Value
主办	127.0.0.1
港口	8200
方案	HTTP
后端	秘密
defaultKey	应用
profileSeparator	,
kvVersion	1
skipSslValidation	假
时间到	五

Important

上表中的所有属性必须以`spring.cloud.config.server.vault`作为前缀。

所有可配置的属性都可以在`org.springframework.cloud.config.server.environment.VaultEnvironmentRepository`找到。

Vault 0.10.0引入了版本化的键值后端（k / v后端版本2），它公开了与早期版本不同的API，现在需要在装载路径和实际上下文路径之间放置一个`data/` ，并将秘密包装在`data`对象中。 设置`kvVersion=2`将考虑到这一点。

运行配置服务器后，您可以向服务器发出HTTP请求以从Vault后端检索值。 为此，您需要一个用于Vault服务器的令牌。

首先，将一些数据放入您的保险柜中，如以下示例所示：

```
$ vault write secret/application foo=bar baz=bam
$ vault write secret/myapp foo=myappsbar
```

其次，向您的配置服务器发出HTTP请求以检索值，如以下示例所示：

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

您应该看到类似于以下内容的回复：

```
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```

Multiple Properties Sources译:多个属性来源

使用保险柜时，您可以为您的应用程序提供多个属性来源。 例如，假设您已经将数据写入了Vault中的以下路径：

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

写入`secret/application`属性可用于all applications using the Config Server 。 名称为`myApp`的应用程序可以使用写入`secret/myApp`和`secret/application`任何属性。 当`myApp`启用了`dev`配置文件时，写入所有上述路径的属性将可用，列表中第一个路径的属性优先于其他路径。

5.1.5 Accessing Backends Through a Proxy译：5.1.5通过代理访问后端

配置服务器可以通过HTTP或HTTPS代理访问Git或Vault后端。 通过`proxy.http`和`proxy.https`下的设置可以为Git或Vault控制此行为。 这些设置是每个存储库的，因此如果您使用的是composite environment repository，则必须分别为每个后端中的每个后端配置代理设置。 如果使用需要使用单独代理服务器的HTTP和HTTPS URL的网络，则可以为单个后端配置HTTP和HTTPS代理设置。

下表介绍了HTTP和HTTPS代理的代理配置属性。 所有这些属性必须以`proxy.http`或`proxy.https`为前缀。

表5.2。 代理配置属性

Property Name	Remarks
主办	代理的主机。
港口	用于访问代理的端口。
nonProxyHosts	配置服务器应该在代理之外访问的任何主机。 如果为`proxy.http.nonProxyHosts`和`proxy.https.nonProxyHosts`提供了值，则将使用`proxy.http`值。
用户名	用于向代理进行身份验证的用户名。 如果为`proxy.http.username`和`proxy.https.username`提供了值，则将使用`proxy.http`值。
密码	用于向代理进行身份验证的密码。 如果为`proxy.http.password`和`proxy.https.password`提供了值，则将使用`proxy.http`值。

以下配置使用HTTPS代理来访问Git存储库。

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          proxy:
            https:
              host: my-proxy.host.io
              password: myproxypassword
              port: '3128'
              username: myproxyusername
              nonProxyHosts: example.com
```

5.1.6 Sharing Configuration With All Applications译：5.1.6与所有应用程序共享配置

在所有应用程序之间共享配置根据您采取的方法而有所不同，如以下主题中所述：

- the section called “File Based Repositories”
- the section called “Vault Server”

File Based Repositories译：基于文件的存储库

基于文件（GIT，SVN和本地）仓库，资源，文件名中`application*`（`application.properties`，`application.yml`，`application-*.properties`，等等）的所有客户端应用程序之间共享。 您可以使用具有这些文件名的资源来配置全局默认值，并根据需要让它们被应用程序特定的文件覆盖。

#_property_overrides [property overrides]功能也可用于设置全局默认值，并允许占位符应用程序在本地覆盖它们。

使用“本地”配置文件（本地文件系统后端），您应该使用不属于服务器自己配置的显式搜索位置。 否则，默认搜索位置中的`application*`资源将被删除，因为它们是服务器的一部分。

Vault Server译：Vault服务器

将Vault用作后端时，可以通过将配置置于`secret/application`与所有应用程序共享配置。 例如，如果运行以下Vault命令，则使用配置服务器的所有应用程序都将具有以下属性`foo`和`baz`：

```
$ vault write secret/application foo=bar baz=bam
```

5.1.7 JDBC Backend译：5.1.7 JDBC后端

Spring Cloud Config Server支持JDBC（关系数据库）作为配置属性的后端。 您可以通过将`spring-jdbc`添加到类路径并使用`jdbc`配置文件或添加类型`JdbcEnvironmentRepository`的bean来启用此功能。 如果您在类路径中包含正确的依赖关系（请参阅用户指南以获取更多详细信息），Spring Boot配置数据源。

数据库需要有一个表叫`PROPERTIES`一个名为列`APPLICATION`，`PROFILE`和`LABEL`（与通常`Environment`意思），加上`KEY`和`VALUE`在键和值对`Properties`风格。 所有字段都是Java中的String类型，因此您可以将它们`VARCHAR`为任意长度的`VARCHAR`。 属性值的行为方式与它们来自名为`{application}-{profile}.properties`Spring Boot属性文件（包括所有加密和解密）相同，它们将作为后处理步骤（即不在存储库实现中直接使用）应用。

5.1.8 Composite Environment Repositories译：5.1.8复合环境存储库

在某些情况下，您可能希望从多个环境存储库中提取配置数据。 为此，您可以在配置服务器的应用程序属性或YAML文件中启用`composite`配置文件。 例如，如果您想从Subversion存储库以及两个Git存储库中提取配置数据，则可以为您的配置服务器设置以下属性：

```
spring:
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
        -
          type: svn
          uri: file:///path/to/svn/repo
        -
          type: git
          uri: file:///path/to/rex/git/repo
        -
          type: git
          uri: file:///path/to/walter/git/repo
```

使用此配置时，优先级由`composite`密钥下存储库的列出顺序决定。 在上面的例子中，首先列出了Subversion版本库，因此在Subversion版本库中找到的值将覆盖为其中一个Git存储库中的相同属性找到的值。 在发现的值`rex` Git仓库会发现在同一个属性的值之前使用`walter` Git仓库。

如果只想从各个不同类型的存储库中提取配置数据，则可以在配置服务器的应用程序属性或YAML文件中启用相应的配置文件，而不是`composite`配置文件。 例如，如果您想要从单个Git存储库和单个HashiCorp Vault服务器提取配置数据，则可以为您的配置服务器设置以下属性：

```
spring:
  profiles:
    active: git, vault
  cloud:
    config:
      server:
        git:
          uri: file:///path/to/git/repo
          order: 2
        vault:
          host: 127.0.0.1
          port: 8200
          order: 1
```

使用此配置，可以通过`order`属性确定优先顺序。 您可以使用`order`属性来指定所有存储库的优先顺序。 `order`属性的数值越低，它的优先级就越高。 存储库的优先顺序有助于解决包含相同属性值的存储库之间的任何潜在冲突。

如果您的复合环境包含上一示例中的Vault服务器，则必须在对配置服务器进行的每个请求中都包含Vault令牌。 见Vault Backend 。

从环境存储库检索值时发生任何类型的故障都会导致整个组合环境出现故障。

在使用复合环境时，所有存储库都包含相同的标签很重要。 如果您的环境与前面示例中的环境类似，并且您使用`master`标签请求配置数据，但Subversion存储库不包含名为`master`的分支，则整个请求将失败。

Custom Composite Environment Repositories译:自定义复合环境存储库

除了使用Spring Cloud的环境存储库之外，您还可以提供自己的`EnvironmentRepository` bean作为组合环境的一部分。 为此，您的bean必须实现`EnvironmentRepository`接口。 如果要在组合环境中控制自定义`EnvironmentRepository`的优先级，则还应该实现`Ordered`接口并覆盖`getOrdered`方法。 如果您未实现`Ordered`接口，则`EnvironmentRepository`的优先级最低。

5.1.9 Property Overrides译: 5.1.9属性重写

配置服务器具有“覆盖”功能，可让操作员为所有应用程序提供配置属性。 被覆盖的属性不能被具有普通Spring Boot挂钩的应用程序意外更改。 要声明覆盖，请将名称 - 值对映射添加到`spring.cloud.config.server.overrides`，如下例所示：

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

前面的示例会使配置客户端的所有应用程序都读取`foo=bar`，而与其自己的配置无关。

配置系统不能强制应用程序以任何特定方式使用配置数据。 因此，覆盖不可执行。 但是，它们确实为Spring Cloud Config客户端提供了有用的默认行为。

通常情况下，使用Spring环境占位符`${}`可以用反斜杠（转义（和解决在客户端上）`\`），以逃避`$`或者`{`。 例如，`\${app.foo:bar}`解析为`bar`，除非应用程序提供自己的`app.foo`。

在YAML中，你不需要逃避反斜杠本身。 但是，在属性文件中，当您在服务器上配置覆盖时，您确实需要转义反斜杠。

通过在远程存储库中设置`spring.cloud.config.overrideNone=true`标志（缺省值为false），可以将客户端中所有覆盖的优先级更改为默认值，让应用程序在环境变量或系统属性中提供自己的值。

5.2 Health Indicator译: 5.2健康指标

配置服务器附带一个运行状况指示器，用于检查配置的`EnvironmentRepository`是否正常工作。 默认情况下，它会要求`EnvironmentRepository`查询名为`app`的应用程序，`default`配置文件以及`EnvironmentRepository`实施提供的默认标签。

您可以配置Health Indicator以检查更多应用程序以及自定义配置文件和自定义标签，如下例所示：

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

您可以通过设置 `spring.cloud.config.server.health.enabled=false` 来禁用健康指示器。

5.3 Security 译：5.3安全

您可以以任何对您有意义的方式保护您的配置服务器（从物理网络安全到OAuth2承载令牌），因为Spring Security和Spring Boot为许多安全配置提供支持。

要使用默认的Spring Boot配置的HTTP Basic安全性，请在类路径中包含Spring Security（例如，通过 `spring-boot-starter-security` ）。 默认值是 `user` 的用户名和随机生成的密码。 随机密码在实践中并不实用，所以我们建议您配置密码（通过设置 `spring.security.user.password` ）并加密（请参阅下面有关如何操作的说明）。

5.4 Encryption and Decryption 译：5.4加密和解密

Important

要使用加密和解密功能，您需要安装在JVM中的全功能JCE（默认情况下不包括此功能）。 您可以从Oracle下载“Java加密扩展（JCE）无限强化管辖权限策略文件”，并按照安装说明进行操作（实质上，您需要将JRE lib / security目录中的两个策略文件替换为您下载的那些文件）。

如果远程属性源包含加密内容（值以 `{cipher}` ），则在通过HTTP发送到客户端之前将它们解密。 这种设置的主要优点是，属性值在“休息”时不需要以纯文本格式（例如，在git存储库中）。 如果某个值无法解密，则将其从属性源中移除，并使用相同的键添加其他属性，但前缀为 `invalid` ，值为“不适用”（通常为 `<n/a>` ）。 这主要是为了防止密码文本被用作密码并意外泄漏。

如果您为配置客户端应用程序设置远程配置存储库，则它可能包含类似于以下内容的 `application.yml` ：

application.yml。

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

.properties文件中的加密值不得包含在引号中。 否则，该值不会被解密。 以下示例显示可以工作的值：

application.properties。

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

您可以安全地将此纯文本推送到共享的git存储库，并且秘密密码保持受保护状态。

该服务器还公开了 `/encrypt` 和 `/decrypt` 端点（假设这些端点是安全的并且只能由授权代理访问）。 如果编辑远程配置文件，则可以使用配置服务器通过 `/encrypt` 到 `/encrypt` 端点来加密值，如以下示例所示：

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

如果您加密的值中包含需要进行网址编码的字符，则应使用 `--data-urlencode` 选项至 `curl` 以确保它们已正确编码。

请务必不要在加密值中包含任何curl命令统计信息。 将该值输出到文件可以帮助避免此问题。

反向操作也可通过 `/decrypt` （前提是服务器配置了对称密钥或完整密钥对）提供，如以下示例所示：

```
$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

如果使用curl进行测试，则使用 `--data-urlencode` （而不是 `-d` ）或设置明确的 `Content-Type: text/plain` 以确保在有特殊字符（'+'特别棘手）时，curl正确编码数据。

取出加密的值，并在将其放入YAML或属性文件之前以及在提交之前将其添加到远程（可能不安全）的存储之前添加 `{cipher}` 前缀。

`/encrypt` 和 `/decrypt` 端点也都接受 `/*/{name}/{profiles}` 形式的路径，这些路径可用于在客户端调用主环境资源时根据每个应用程序（名称）和每个概要文件控制加密。

要以这种细化的方式控制加密，您还必须提供类型为 `@Bean` 的 `TextEncryptorLocator` ，该类型为每个名称和配置文件创建不同的加密器。 默认提供的不是这样（所有加密都使用相同的密钥）。

`spring` 命令行客户端（安装了Spring Cloud CLI扩展）也可用于加密和解密，如以下示例所示：

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

要在文件中使用密钥（例如用于加密的RSA公钥），请使用“@”预先输入密钥值并提供文件路径，如以下示例所示：

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

`--key`参数是强制性的（尽管前缀为`--`）。

5.5 Key Management译：5.5密钥管理

配置服务器可以使用对称（共享）密钥或不对称密钥（RSA密钥对）。 不对称选择在安全性方面优越，但使用对称密钥通常更方便，因为它是在`bootstrap.properties`配置的单个属性值。

要配置对称密钥，您需要将`encrypt.key`设置为密码字符串（或使用`ENCRYPT_KEY`环境变量将其保留为纯文本配置文件）。

要配置非对称密钥，您可以将密钥设置为PEM编码的文本值（在`encrypt.key`）或使用密钥库（例如由JDK附带的`keytool`实用程序创建的密钥库）。 下表介绍了密钥库属性：

Property	描述
`encrypt.keyStore.location`	包含`Resource`位置
`encrypt.keyStore.password`	保存解锁密钥库的密码
`encrypt.keyStore.alias`	标识要使用的商店中的哪个密钥

加密使用公钥完成，解密需要私钥。 因此，原则上，如果只想加密（并且准备用私钥在本地解密值），则只能在服务器中配置公钥。 实际上，您可能不想在本地进行解密，因为它将密钥管理过程分散到所有客户端，而不是集中在服务器中。 另一方面，如果您的配置服务器相对不安全，并且只有少数客户端需要加密属性，则它可能是一个有用的选项。

5.6 Creating a Key Store for Testing译：5.6创建用于测试的密钥存储区

要创建用于测试的密钥库，可以使用类似以下的命令：

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass letmein
```

将`server.jks`文件放入类路径中（例如），然后在`bootstrap.yml`中为Config Server创建以下设置：

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

5.7 Using Multiple Keys and Key Rotation译：5.7使用多个按键和按键旋转

除了加密属性值中的前缀`{cipher}`之外，配置服务器`{name:value}`在（Base64编码）密码文本开始之前查找零个或多个`{name:value}`前缀。 密钥被传递给`TextEncryptorLocator`，它可以做任何需要的逻辑来为密码找到`TextEncryptor`。 如果您已配置密钥库（`encrypt.keystore.location`），则默认定位器将使用`key`前缀提供的别名查找具有类似以下内容的密文的密钥：

```
foo:
  bar: `{cipher}{key:testkey}...`
```

定位器寻找名为“testkey”的密钥。 秘密也可以通过在前缀中使用`{secret:… }`值来提供。 但是，如果未提供密钥存储库，则默认使用密钥存储库密码（这是您构建密钥库时未获得的密码）。 如果你确实提供了一个秘密，你还应该使用自定义的`SecretLocator`加密秘密。

当密钥仅用于加密配置数据的几个字节时（也就是说，它们没有在其他地方使用），密钥旋转几乎不需要在密码的基础上。 但是，您可能偶尔需要更改密钥（例如，如果发生安全漏洞）。 在这种情况下，所有客户端都需要更改源配置文件（例如，在git中），并在所有密码中使用新的前缀`{key:… }`。 请注意，客户端需要首先检查密钥别名在配置服务器密钥库中是否可用。

如果您想让配置服务器处理所有加密以及解密，则也可以将`{name:value}`前缀添加为发布到`/encrypt`端点的纯文本。

5.8 Serving Encrypted Properties译：5.8提供加密属性

有时您希望客户端在本地解密配置，而不是在服务器中进行配置。 在这种情况下，如果您提供`encrypt.*`配置来查找密钥，您仍可以拥有`/encrypt`和`/decrypt`端点，但您需要通过将`spring.cloud.config.server.encrypt.enabled=false`放置在`bootstrap.[yml|properties]`来明确地关闭对传出属性的解密。 如果你不关心端点，它应该工作，如果你没有配置键或启用标志。

6. Serving Alternative Formats译：6提供替代格式

来自环境端点的默认JSON格式非常适合Spring应用程序使用，因为它直接映射到`Environment`抽象。 如果您愿意，可以通过向资源路径添加后缀（“.yml”，“.yaml”或“.properties”）来使用与YAML或Java属性相同的数据。 这对于不关心JSON端点结构或者它们提供的额外元数据的应用程序（例如，不使用Spring的应用程序可能从这种方法的简单性中受益）的消费很有用。

在可能的情况下，YAML和属性表示有一个额外的标志（作为布尔型查询参数`resolvePlaceholders`）用于表示源文档中的占位符（在标准Spring `${… }`表单中）应该在渲染之前在输出中解析。 对于不了解Spring占位符约定的消费者来说，这是一个有用的功能。

> 在使用YAML或属性格式方面存在限制，主要涉及元数据的丢失。 例如，JSON被构造为属性源的有序列表，其名称与源相关。 YAML和属性表单会合并到一个映射中，即使这些值的来源有多个来源，并且原始源文件的名称也会丢失。 此外，YAML表示不一定是备份库中YAML源的忠实代表。 它由一系列平面资产来源构成，并且必须假设关键的形式。

7. Serving Plain Text译：7提供纯文本

您的应用程序可能需要通用的纯文本配置文件，而不是使用`Environment`抽象（或YAML或其他属性格式中的其中一种替代表示形式），这些文件是针对其环境定制的。 在配置服务器通过额外的端点提供这些`/{name}/{profile}/{label}/{path}`，其中`name`， `profile`，并`label`的含义与常规环境终点相同，但`path`是一个文件名（如`log.xml`）。 此端点的源文件位置与环境端点的相同。 相同的搜索路径用于属性和YAML文件。 但是，不是汇总所有匹配的资源，而只返回匹配的第一个资源。

找到资源后，通过使用生效的`Environment`为所提供的应用程序名称，配置文件和标签解析正常格式（`${… }`）的占位符。 通过这种方式，资源端点与环境端点紧密集成。 考虑以下GIT或SVN存储库的示例：

```
application.yml
nginx.conf
```

`nginx.conf`看起来像这样：

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

和`application.yml`是这样的：

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

`/foo/default/master/nginx.conf`资源可能如下所示：

```
server {
    listen              80;
    server_name         example.com;
}
```

和`/foo/development/master/nginx.conf`是这样的：

```
server {
    listen              80;
    server_name         develop.com;
}
```

与用于环境配置的源文件一样，`profile`用于解析文件名。 因此，如果您需要特定于配置文件的文件，`/*/development/*/logback.xml`可以通过名为`logback-development.xml`（优先于`logback.xml`）的文件来解析。

如果您不想提供`label`并让服务器使用默认标签，则可以提供`useDefaultLabel`请求参数。 因此，上述`default`配置文件的示例可能为`/foo/default/nginx.conf?useDefaultLabel`。

8. Embedding the Config Server译：8嵌入配置服务器

Config Server作为独立应用程序运行得最好。 但是，如果需要，可以将其嵌入到其他应用程序中。 为此，请使用`@EnableConfigServer`注释。 在这种情况下，名为`spring.cloud.config.server.bootstrap`的可选属性可能很有用。 它是一个标志，用于指示服务器是否应该从其自己的远程存储库进行自我配置。 默认情况下，该标志是关闭的，因为它可以延迟启动。 但是，当嵌入到另一个应用程序中时，以与其他应用程序相同的方式初始化是有意义的。

如果您使用引导标志，则配置服务器需要在`bootstrap.yml`配置其名称和存储库URI。

要更改服务器端点的位置，可以（可选）设置`spring.cloud.config.server.prefix`（例如`/config`），以便在前缀下提供资源。 前缀应该以`/`开始但不以结束。 它应用于配置服务器中的`@RequestMappings`（即，在Spring Boot `server.servletPath`和`server.contextPath`前缀下）。

如果您想直接从后端存储库（而不是从配置服务器）读取应用程序的配置，那么您基本上就会看到一个没有端点的嵌入式配置服务器。 您可以完全通过不使用`@EnableConfigServer`注释（设置`spring.cloud.config.server.bootstrap=true`）来关闭端点。

9. Push Notifications and Spring Cloud Bus译：9推送通知和Spring Cloud Bus

许多源代码存储库提供者（例如Github，Gitlab，Gitee或Bitbucket）通过webhook向您通知存储库中的更改。 您可以通过提供者的用户界面将webhook配置为URL以及您感兴趣的一组事件。 例如， Github使用POST向webhook发送包含提交列表和标题（`X-Github-Event`）设置为`push`的JSON主体。 如果添加对`spring-cloud-config-monitor`库的依赖关系并在配置服务器中激活Spring Cloud Bus，则启用`/monitor`端点。

当webhook被激活时，Config服务器会发送一个`RefreshRemoteApplicationEvent`作为它认为可能发生更改的应用程序的目标。 变化检测可以制定策略。 但是，默认情况下，它会查找与应用程序名称匹配的文件中的更改（例如， `foo.properties`针对`foo`应用程序，而`application.properties`针对所有应用程序）。 当您要覆盖行为时使用的策略是`PropertyPathNotificationExtractor`，它接受请求标头和主体作为参数并返回已更改的文件路径的列表。

使用Github，Gitlab，Gitee或Bitbucket时，默认配置就可以使用。 除了从GitHub，Gitlab，Gitee，或到位桶的JSON通知，可以通过发布触发改变通知`/monitor`与的图案形式编码的身体参数`path={name}`。 这样做广播到匹配`{name}`模式（可以包含通配符）的应用程序。

只有在配置服务器和客户端应用程序中激活了`RefreshRemoteApplicationEvent`才会发送`spring-cloud-bus`。

默认配置还检测本地git存储库中的文件系统更改。 在这种情况下，不使用webhook。 但是，只要编辑配置文件，就会播放刷新。

10. Spring Cloud Config Client译：10. Spring Cloud Config客户端

Spring Boot应用程序可以立即利用Spring Config Server（或由应用程序开发人员提供的其他外部属性资源）。 它还提供了一些与`Environment`更改事件相关的其他有用功能。

10.1 Config First Bootstrap译：10.1配置第一引导程序

在类路径上具有Spring Cloud Config Client的任何应用程序的默认行为如下所示：配置客户端启动时，它将绑定到配置服务器（通过`spring.cloud.config.uri`引导配置属性）并使用远程属性源初始化Spring `Environment`。

这种行为的最终结果是，想要使用配置服务器的所有客户端应用程序都需要一个`bootstrap.yml`（或环境变量），其服务器地址设置为`spring.cloud.config.uri`（它默认为“http：// localhost：8888”）。

10.2 Discovery First Bootstrap译：10.2发现第一引导程序

如果您使用“DiscoveryClient”实施，例如Spring Cloud Netflix和Eureka服务发现或Spring Cloud Consul，则可以让Config服务器在发现服务中注册。 但是，在默认的“配置优先”模式下，客户无法利用注册。

如果您更喜欢使用`DiscoveryClient`来查找配置服务器，可以通过设置`spring.cloud.config.discovery.enabled=true`（默认值为`false`）来完成。 这样做的最终结果是客户端应用程序都需要具有适当发现配置的`bootstrap.yml`（或环境变量）。 例如，使用Spring Cloud Netflix，您需要定义Eureka服务器地址（例如，在`eureka.client.serviceUrl.defaultZone`）。 使用此选项的价格是启动时的额外网络往返，以查找服务注册。 好处是，只要发现服务是固定点，配置服务器就可以更改其坐标。 默认的服务ID是`configserver`，但您可以通过设置`spring.cloud.config.discovery.serviceId`（并在服务器上，以服务的常规方式（例如通过设置`spring.application.name`））来`spring.application.name`该服务。

发现客户端实现都支持某种类型的元数据映射（例如，我们有Eureka的`eureka.instance.metadataMap`）。 可能需要在其服务注册元数据中配置Config服务器的一些其他属性，以便客户端可以正确连接。 如果配置服务器使用HTTP Basic进行保护，则可以将凭据配置为`username`和`password`。 另外，如果配置服务器具有上下文路径，则可以设置`configPath`。 例如，以下YAML文件适用于作为Eureka客户端的Config服务器：

bootstrap.yml。

```
eureka:
  instance:
    ...
    metadataMap:
      user: osufhalskjrtl
      password: lviuhlszvaorhvlo5847
      configPath: /config
```

10.3 Config Client Fail Fast译：10.3配置客户端快速失败

在某些情况下，如果服务无法连接到配置服务器，您可能想要启动服务失败。 如果这是所需的行为，请设置引导程序配置属性`spring.cloud.config.fail-fast=true`以使客户端暂停并出现异常。

10.4 Config Client Retry译：10.4配置客户端重试

如果您希望应用程序启动时配置服务器可能偶尔不可用，那么可以让它在发生故障后继续尝试。 首先，你需要设置`spring.cloud.config.fail-fast=true`。 然后你需要将`spring-retry`和`spring-boot-starter-aop`添加到你的类路径中。 默认行为是重试六次，初始回退间隔为1000ms，后续回退为指数乘数1.1。 您可以通过设置`spring.cloud.config.retry.*`配置属性来配置这些属性（和其他属性）。

取重试行为的完全控制，加`@Bean`类型的`RetryOperationsInterceptor`用的ID `configServerRetryInterceptor`。 Spring Retry有一个支持创建一个的`RetryInterceptorBuilder`。

10.5 Locating Remote Configuration Resources译：10.5查找远程配置资源

配置服务提供来自`/{name}/{profile}/{label}`属性来源，其中客户端应用程序中的默认绑定如下所示：

- "name" = `${spring.application.name}`
- "profile" = `${spring.profiles.active}` (actually `Environment.getActiveProfiles()`)
- "label" = "master"

您可以通过设置覆盖所有这些`spring.cloud.config.*`（其中`*`为`name`， `profile`或者`label`）。 `label`对回滚到以前版本的配置很有用。 使用默认的Config Server实现，它可以是git标签，分支名称或提交ID。 标签也可以作为逗号分隔列表提供。 在这种情况下，列表中的项目将逐个尝试，直到成功。 在功能分支上工作时，此行为非常有用。 例如，您可能想要将配置标签与您的分支对齐，但将其设置为可选（在这种情况下，请使用`spring.cloud.config.label=myfeature,develop`）。

10.6 Specifying Multiple Urls for the Config Server译：10.6指定配置服务器的多个 Url

为了确保部署多个Config Server实例并预期一个或多个实例不时不可用时的高可用性，可以指定多个URL（作为`spring.cloud.config.uri`属性下的逗号分隔列表），或者让所有实例在Eureka等服务注册中注册（如果使用Discovery-First Bootstrap模式）。 请注意，这样做只有在Config Server未运行时（即应用程序退出时）或发生连接超时时才能确保高可用性。 例如，如果配置服务器返回500（内部服务器错误）响应或配置客户端从配置服务器收到401（由于凭据不正确或其他原因），则配置客户端不会尝试从其他URL获取属性。 这种错误表示用户问题，而不是可用性问题。

如果您在Config服务器上使用HTTP基本安全性，则仅当您在`spring.cloud.config.uri`属性下指定的每个URL中嵌入凭据时，才可以支持每个配置服务器授权凭证。 如果您使用任何其他类型的安全机制，则不能（当前）支持per-Config服务器身份验证和授权。

10.7 Configuring Read Timeouts译：10.7配置读取超时

如果要配置读取超时，可以使用属性`spring.cloud.config.request-read-timeout`完成此操作。

10.8 Security译：10.8安全

如果您在服务器上使用HTTP Basic安全性，客户端需要知道密码（如果不是默认密码，则需要输入用户名）。 您可以通过配置服务器URI或通过单独的用户名和密码属性指定用户名和密码，如以下示例所示：

bootstrap.yml。

```
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

以下示例显示了传递相同信息的备用方法：

bootstrap.yml。

```
spring:
  cloud:
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

`spring.cloud.config.password`和`spring.cloud.config.username`值覆盖URI中提供的任何内容。

如果您在Cloud Foundry上部署应用程序，提供密码的最佳方式是通过服务凭据（例如在URI中，因为它不需要位于配置文件中）。 以下示例适用于本地以及Cloud Foundry上用户提供的名为`configserver`：

bootstrap.yml。

```
spring:
  cloud:
    config:
     uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}
```

如果您使用其他形式的安全性，则可能需要 provide a `RestTemplate`至`ConfigServicePropertySourceLocator`（例如，通过在引导程序环境中抓取并注入它）。

10.8.1 Health Indicator译：10.8.1健康指标

Config Client提供一个Spring Boot Health Indicator，它试图从配置服务器加载配置。 通过设置`health.config.enabled=false`可以禁用健康指示器。 由于性能原因，响应也被缓存。 默认的缓存时间为5分钟。 要更改该值，请设置`health.config.time-to-live`属性（以毫秒为单位）。

10.8.2 Providing A Custom RestTemplate译：10.8.2提供自定义RestTemplate

在某些情况下，您可能需要自定义从客户端向配置服务器发出的请求。 通常，这样做需要传递特殊的`Authorization`标`Authorization`验证对服务器的请求。 提供自定义`RestTemplate`：

1. Create a new configuration bean with an implementation of `PropertySourceLocator`, as shown in the following example:

CustomConfigServiceBootstrapConfiguration.java。

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties = configClientProperties();
       ConfigServicePropertySourceLocator configServicePropertySourceLocator =  new ConfigServicePropertySourceLocator(clientProperties);
        configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));
        return configServicePropertySourceLocator;
    }
}
```

1. In `resources/META-INF`, create a file called `spring.factories` and specify your custom configuration, as shown in the following example:

spring.factories。

```
org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapConfiguration
```

10.8.3 Vault译：10.8.3 Vault

将Vault用作配置服务器的后端时，客户端需要为服务器提供令牌以从Vault检索值。 该令牌可以在客户端内设置提供`spring.cloud.config.token`在`bootstrap.yml`，如图以下示例：

bootstrap.yml。

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

10.9 Nested Keys In Vault译：10.9 Vault中的嵌套键

Vault支持将密钥嵌套到存储在Vault中的值中的功能，如以下示例所示：

`echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -`

该命令将JSON对象写入您的Vault。 要在Spring中访问这些值，可以使用传统的点（`.`）注释，如以下示例中所示

```
@Value("${appA.secret}")
String name = "World";
```

上面的代码会将`name`变量的值设置为`appAsecret`。

Part III. Spring Cloud Netflix 译:第三部分。 Spring Cloud Netflix

Finchley.RELEASE

该项目通过自动配置和绑定到Spring Environment和其他Spring编程模型成语，为Spring Boot应用程序提供了Netflix OSS集成。 通过一些简单的注释，您可以快速启用和配置应用程序内的通用模式，并使用经过战斗测试的Netflix组件构建大型分布式系统。 提供的模式包括服务发现（Eureka），断路器（Hystrix），智能路由（Zuul）和客户端负载平衡（Ribbon）。

11. Service Discovery: Eureka Clients 译: 11.服务发现：尤里卡客户

服务发现是基于微服务架构的关键原则之一。 尝试手动配置每个客户端或某种形式的约定可能很难做到，并且可能很脆弱。 Eureka是Netflix服务发现服务器和客户端。 服务器可以配置和部署为高可用性，每台服务器将注册服务的状态复制到其他服务器。

11.1 How to Include Eureka Client 译: 11.1如何包含Eureka客户端

要将尤里卡客户端包含在您的项目中，请使用组ID为`org.springframework.cloud`且工件ID为`spring-cloud-starter-netflix-eureka-client`的起动器。 有关使用当前的Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

11.2 Registering with Eureka 译: 11.2向Eureka注册

当客户注册Eureka时，它会提供有关自己的元数据，例如主机，端口，健康指示符URL，主页和其他详细信息。 尤里卡接收来自属于服务的每个实例的心跳消息。 如果心跳在可配置的时间表上失败，则通常从注册表中删除该实例。

以下示例显示了一个最小的Eureka客户端应用程序：

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

请注意，上例显示了一个正常的Spring Boot应用程序。 通过在类路径上拥有`spring-cloud-starter-netflix-eureka-client`，您的应用程序将自动注册到Eureka服务器。 找到Eureka服务器需要配置，如以下示例所示：

application.yml。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在前面的例子中，“defaultZone”是一个神奇的字符串回退值，它为任何不表示偏好的客户提供服务URL（换句话说，它是一个有用的默认值）。

默认应用程序名称（即服务ID），虚拟主机和非安全端口（取自`Environment`）分别`${spring.application.name}``${spring.application.name}`和`${server.port}`。

拥有`spring-cloud-starter-netflix-eureka-client`的类路径使得该应用变成了Eureka“实例”（即它自己注册）和一个“客户端”（它可以查询注册表以查找其他服务）。 实例行为由`eureka.instance.*`配置键驱动，但如果确保应用程序的值为`spring.application.name`（这是Eureka服务ID或VIP的默认值），则默认值就可以。

有关 可配置选项的更多详细信息，请参阅 EurekaInstanceConfigBean和 EurekaClientConfigBean 。

要禁用Eureka Discovery客户端，您可以将`eureka.client.enabled`设置为`false`。

11.3 Authenticating with the Eureka Server 译: 11.3使用Eureka服务器进行身份验证

如果其中一个`eureka.client.serviceUrl.defaultZone`URL包含凭证（卷曲样式，如下所示：`http://user:password@localhost:8761/eureka`），则HTTP基本身份验证会自动添加到您的尤里卡客户端。 对于更复杂的需求，您可以创建类型为`@Bean`的`DiscoveryClientOptionalArgs`并将`ClientFilter`实例注入到其中，所有这些实例都适用于从客户端到服务器的调用。

由于Eureka中的限制，无法支持每个服务器的基本身份验证凭据，因此只使用找到的第一个集。

11.4 Status Page and Health Indicator 译: 11.4状态页面和健康指示器

Eureka实例的状态页面和运行状况指示器默认分别默认为`/info`和`/health`，这是Spring Boot Actuator应用程序中有用端点的默认位置。 如果使用非默认上下文路径或servlet路径（如`server.servletPath=/custom`）或管理端点路径（如`management.contextPath=/admin`），则需要更改这些设置，即使对于Actuator应用程序也是`management.contextPath=/admin`。 以下示例显示了这两个设置的默认值：

application.yml。

```
eureka:
  instance:
    statusPageUrlPath: ${management.server.servlet.context-path}/info
    healthCheckUrlPath: ${management.server.servlet.context-path}/health
```

这些链接显示在客户端使用的元数据中，并在某些场景中用于决定是否将请求发送到您的应用程序，因此如果它们准确无误，这些信息会很有帮助。

11.5 Registering a Secure Application译：11.5注册安全应用程序

如果您的应用想通过HTTPS联系，则可以在`EurekaInstanceConfig`设置两个标志：

- `eureka.instance.[nonSecurePortEnabled]=[false]`
- `eureka.instance.[securePortEnabled]=[true]`

这样做使得Eureka发布实例信息，显示明确的安全通信偏好。 对于以此方式配置的服务，Spring Cloud `DiscoveryClient`始终会返回一个以`https`开头的URI。 同样，以这种方式配置服务时，Eureka（本机）实例信息具有安全的运行状况检查URL。

由于尤里卡在内部工作的方式，它仍然会为状态和主页发布不安全的URL，除非您也明确地覆盖这些URL。 您可以使用占位符来配置尤里卡实例URL，如下例所示：

application.yml。

```
eureka:
  instance:
    statusPageUrl: https://${eureka.hostname}/info
    healthCheckUrl: https://${eureka.hostname}/health
    homePageUrl: https://${eureka.hostname}/
```

（请注意， `${eureka.hostname}`是一个原生的占位符，仅在更高版本的Eureka中可用。您可以使用Spring占位符来实现同样的功能，例如，通过使用`${eureka.instance.hostName}` ）

如果您的应用程序在代理之后运行，并且SSL终端位于代理中（例如，如果您在Cloud Foundry或其他平台中作为服务运行），那么您需要确保截取代理“已转发”头并由应用程序处理。 如果嵌入在Spring Boot应用程序中的Tomcat容器具有“X-Forwarded - \ *”头的显式配置，则会自动发生。 您的应用程序向自身呈现的链接错误（错误的主机，端口或协议）表示您得到的配置错误。

11.6 Eureka’s Health Checks译：11.6尤里卡的健康检查

默认情况下，Eureka使用客户端心跳来确定客户端是否启动。 除非另有规定，否则Discovery客户端不会通过Spring Boot执行器传播应用程序的当前运行状况检查状态。 因此，在成功注册后，Eureka总是宣布申请处于“UP”状态。 可以通过启用尤里卡健康检查来更改此行为，这会导致向尤里卡传播应用程序状态。 因此，每个其他应用程序都不会将流量发送给“UP”以外的其他状态的应用程序。 以下示例显示如何为客户端启用运行状况检查：

application.yml。

```
eureka:
  client:
    healthcheck:
      enabled: true
```

`eureka.client.healthcheck.enabled=true`应该只设置在`application.yml` 。 设置`bootstrap.yml`的值会导致不良的副作用，例如在Eureka中以`UNKNOWN`状态注册。

如果您需要更多控制健康检查，请考虑实施您自己的 `com.netflix.appinfo.HealthCheckHandler` 。

11.7 Eureka Metadata for Instances and Clients译：11.7针对实例和客户的尤里卡元数据

值得花一些时间了解尤里卡元数据的工作原理，以便您可以在平台中使用它。 存在主机名，IP地址，端口号，状态页和运行状况检查等信息的标准元数据。 这些信息发布在服务注册中心，客户用它直接联系服务。 可以将额外的元数据添加到`eureka.instance.metadataMap`的实例注册中，并且可以在远程客户端访问此元数据。 通常，除非客户端知道元数据的含义，否则其他元数据不会更改客户端的行为。 本文稍后会介绍一些特殊情况，Spring Cloud已经为元数据映射赋予了含义。

11.7.1 Using Eureka on Cloud Foundry译：11.7.1在 Cloud Foundry上 使用 Eureka

Cloud Foundry具有全局路由器，因此同一应用程序的所有实例都具有相同的主机名（其他PaaS解决方案具有相似的体系结构具有相同的安排）。 这不一定是使用尤里卡的障碍。 但是，如果您使用路由器（推荐甚至是强制性的，取决于您的平台的设置方式），您需要明确设置主机名和端口号（安全或不安全），以便他们使用路由器。 您可能还想使用实例元数据，以便您可以区分客户端上的实例（例如，在自定义负载平衡器中）。 默认情况下，
`eureka.instance.instanceId`是`vcap.application.instance_id` ，如以下示例所示：

application.yml。

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

根据Cloud Foundry实例中安全规则的设置方式，您可以注册并使用主机VM的IP地址进行直接服务到服务调用。 此功能在Pivotal Web服务（ PWS ）上尚未提供。

11.7.2 Using Eureka on AWS译：11.7.2在 AWS上 使用 Eureka

如果计划将应用程序部署到AWS云，则必须将Eureka实例配置为可识别AWS。 您可以按如下方式自定义EurekaInstanceConfigBean ：

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {
  EurekaInstanceConfigBean b = new EurekaInstanceConfigBean(inetUtils);
  AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
  b.setDataCenterInfo(info);
  return b;
}
```

11.7.3 Changing the Eureka Instance ID译：11.7.3更改尤里卡实例ID

一个vanilla Netflix Eureka实例注册的ID与主机名相同（即每台主机只有一个服务）。 Spring Cloud Eureka提供了一个合理的默认设置，其定义如下：

`${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}}`

一个例子是 `myhost:myappname:8080` 。

通过使用Spring Cloud，您可以通过在 `eureka.instance.instanceId` 提供唯一标识符来覆盖此值，如以下示例所示：

application.yml。

```
eureka:
  instance:
    instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

通过前面的示例中显示的元数据以及部署在本地主机上的多个服务实例，将随机值插入此处以使实例具有唯一性。 在Cloud Foundry中，`vcap.application.instance_id` 自动填充到Spring Boot应用程序中，因此不需要随机值。

11.8 Using the EurekaClient 译：11.8使用 EurekaClient

一旦你有一个应用程序是一个发现客户端，你可以用它来发现Eureka Server的服务实例。 一种方法是使用本地 `com.netflix.discovery.EurekaClient` （与Spring Cloud `DiscoveryClient` ），如以下示例所示：

```
@Autowired
private EurekaClient discoveryClient;

public String serviceUrl() {
    InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);
    return instance.getHomePageUrl();
}
```

不要使用 `EurekaClient` 在 `@PostConstruct` 方法或在 `@Scheduled` 方法（或任何地方的 `ApplicationContext` 可能尚未启动）。 它初始化为 `SmartLifecycle` （含 `phase=0` ），所以最早可以依赖它的是另一个更高阶段的 `SmartLifecycle` 。

11.8.1 EurekaClient without Jersey 译：11.8.1没有泽西岛的 EurekaClient

默认情况下，EurekaClient使用Jersey进行HTTP通信。 如果你希望避免来自Jersey的依赖关系，你可以将它从你的依赖关系中排除。 Spring Cloud基于Spring `RestTemplate` 自动配置传输客户端。 以下示例显示Jersey被排除在外：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-client</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-core</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jersey.contribs</groupId>
            <artifactId>jersey-apache-client4</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

11.9 Alternatives to the Native Netflix EurekaClient 译：11.9本土 Netflix EurekaClient的替代方案

您无需使用原始Netflix `EurekaClient` 。 另外，在某种类型的包装器中使用它通常更方便。 Spring Cloud通过逻辑Eureka服务标识符（VIP）支持Feign（REST客户端构建器）和Spring `RestTemplate` ，而不是物理URL。 要使用固定的物理服务器列表配置功能区，可以将 `<client>.ribbon.listOfServers` 设置为以逗号分隔的物理地址（或主机名）列表，其中 `<client>` 是客户端的ID。

您还可以使用 `org.springframework.cloud.client.discovery.DiscoveryClient` ，它为发现客户端提供了一个简单的API（不是Netflix特有的），如以下示例所示：

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

11.10 Why Is It so Slow to Register a Service? 译：11.10为什么注册服务太慢？

作为一个实例还涉及到注册表的周期性心跳（通过客户 `serviceUrl` ），默认持续时间为30秒。 服务不可用于客户端发现，直到实例，服务器和客户端在其本地缓存中都具有相同的元数据（因此可能需要3次检测信号）。 您可以通过设置 `eureka.instance.leaseRenewalIntervalInSeconds` 来更改期间。 将其设置为小于30的值可加快客户端与其他服务的连接。 在生产中，由于服务器中的内部计算对租约续订期作出了假设，所以最好坚持使用默认值。

11.11 Zones 译：11.11区域

如果您已经将Eureka客户端部署到多个区域，则可能希望这些客户端在尝试其他区域中的服务之前使用同一区域中的服务。 要设置它，您需要正确配置您的Eureka客户端。

首先，您需要确保您已将Eureka服务器部署到每个区域，并且它们是彼此的同行。 有关更多信息，请参阅zones and regions上的章节 。

接下来，您需要告诉尤里卡您的服务在哪个区域。您可以使用 `metadataMap` 属性来完成此 `metadataMap` 。 例如，如果将 `service 1` 部署到 `zone 1` 和 `zone 2` ，则需要在 `service 1` 设置以下Eureka属性：

服务1在1区

```
eureka.instance.metadataMap.zone = zone1
eureka.client.preferSameZoneEureka = true
```

服务1在2区

```
eureka.instance.metadataMap.zone = zone2
eureka.client.preferSameZoneEureka = true
```

12. Service Discovery: Eureka Server译：12.服务发现：尤里卡服务器

本节介绍如何设置Eureka服务器。

12.1 How to Include Eureka Server译：12.1如何包含Eureka服务器

要在您的项目中包含Eureka服务器，请使用组`org.springframework.cloud`且工件ID为`spring-cloud-starter-netflix-eureka-server`的起动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

12.2 How to Run a Eureka Server译：12.2如何运行Eureka服务器

以下示例显示了一个最小的Eureka服务器：

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

该服务器的主页具有用于`/eureka/*`下的普通Eureka功能的UI和HTTP API端点。

以下链接有一些尤里卡背景阅读： flux capacitor和 google group discussion 。

由于Gradle的依赖关系解决规则以及缺少父`spring-cloud-starter-netflix-eureka-server`功能，根据`spring-cloud-starter-netflix-eureka-server`不同，可能会导致应用程序启动失败。 为了解决这个问题，添加Spring Boot Gradle插件并导入Spring云启动器父级bom，如下所示：

的build.gradle。

```
buildscript {
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:{spring-boot-docs-version}")
  }
}

apply plugin: "spring-boot"

dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-dependencies:{spring-cloud-version}"
  }
}
```

12.3 High Availability, Zones and Regions译：12.3高可用性，区域和区域

Eureka服务器没有后端存储，但注册表中的服务实例必须发送心跳信号以保持其注册是最新的（所以这可以在内存中完成）。 客户端还有一个Eureka注册的内存缓存（因此他们不必每次请求注册服务都去注册中心）。

默认情况下，每个Eureka服务器也是Eureka客户端，并且需要（至少一个）服务URL来定位对等端。 如果您没有提供该服务，该服务将运行并运行，但它会填满您的日志，并带来很多关于无法向对等端注册的噪音。

另请参阅客户端区域和区域的 below for details of Ribbon support 。

12.4 Standalone Mode译：12.4独立模式

只要存在某种监视器或弹性运行时（如Cloud Foundry）使其保持活动状态，两个缓存（客户端和服务器）和检测信号的组合使独立的Eureka服务器具有相当的故障恢复能力。 在独立模式下，您可能更愿意关闭客户端行为，以免它继续尝试并无法访问其对等端。 以下示例显示如何关闭客户端行为：

application.yml（独立的Eureka服务器）。

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

请注意， `serviceUrl`指向与本地实例相同的主机。

12.5 Peer Awareness译：12.5同伴意识

通过运行多个实例并要求它们相互注册，尤里卡可以变得更加灵活和可用。 实际上，这是默认行为，所以您只需将对象添加到有效的`serviceUrl`即可，如下例所示：

application.yml（两个Peer Aware Eureka服务器）。

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```

在前面的例子中，我们有一个YAML文件，它可以用来在两台主机上运行同一台服务器（ `peer1`和`peer2` ），方法是在不同的Spring配置文件中运行它。 您可以使用此配置通过操作`/etc/hosts`来解析主机名来测试单个主机上的对等认知（在生产中没有太多价值）。 事实上， `eureka.instance.hostname`如果你是知道自己的主机名的机器上运行时，不需要（默认情况下，它是通过使用抬头`java.net.InetAddress` ）。

您可以将多个对等点添加到系统，并且只要它们全部通过至少一个边相互连接，就可以在它们之间同步注册。 如果对等体在物理上是分开的（在数据中心内部或在多个数据中心之间），那么系统原则上可以存活“分裂 - 大脑”类型的故障。

12.6 When to Prefer IP Address译：12.6何时 优先选择 IP地址

在某些情况下，尤里卡最好公布服务的IP地址而不是主机名。 将`eureka.instance.preferIpAddress`设置为`true`并在应用程序向尤里卡注册时，它使用其IP地址而不是其主机名。

如果主机名不能由Java确定，那么IP地址将被发送给Eureka。 只有通过设置`eureka.instance.hostname`属性来显式设置主机名的方式。 您可以在运行时通过使用环境变量来设置主机名，例如`eureka.instance.hostname=${HOST_NAME}` 。

12.7 Securing The Eureka Server译：12.7确保Eureka服务器的安全

您可以通过添加通过Spring Security的到你的服务器A€™的类路径简单地保护您的尤里卡服务器`spring-boot-starter-security` 。 默认情况下，当Spring Security位于类路径中时，它将要求在向应用程序发送每个请求时发送一个有效的CSRF令牌。 尤里卡客户通常不会拥有有效的跨站请求伪造（CSRF）令牌，您需要禁用`/eureka/**`端点的此要求。 例如：

```
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().ignoringAntMatchers("/eureka/**");
        super.configure(http);
    }
}
```

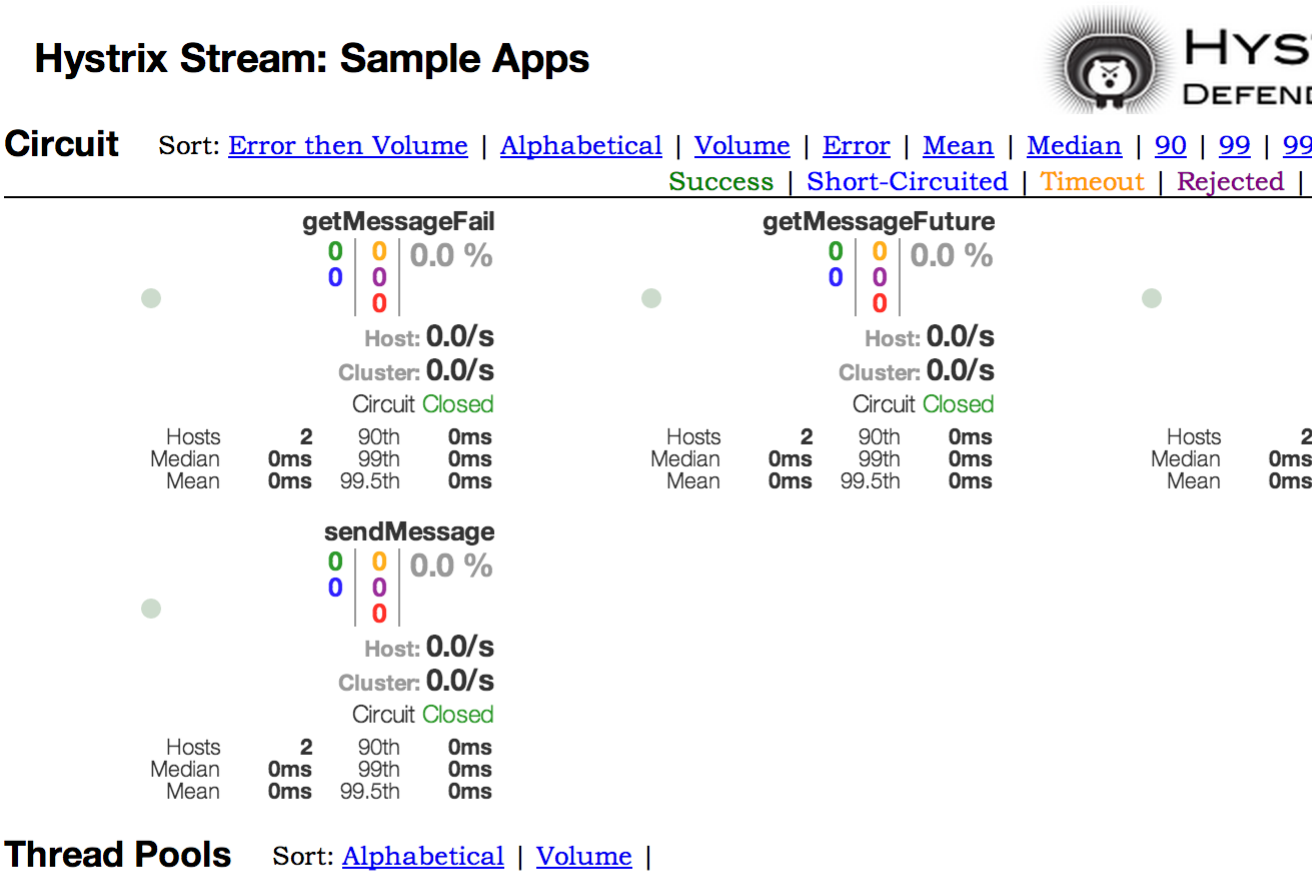
有关CSRF的更多信息，请参阅 Spring Security documentation 。

演示Eureka服务器可以在Spring Cloud Samples repo中找到 。

13. Circuit Breaker: Hystrix Clients译：13断路器： Hystrix客户

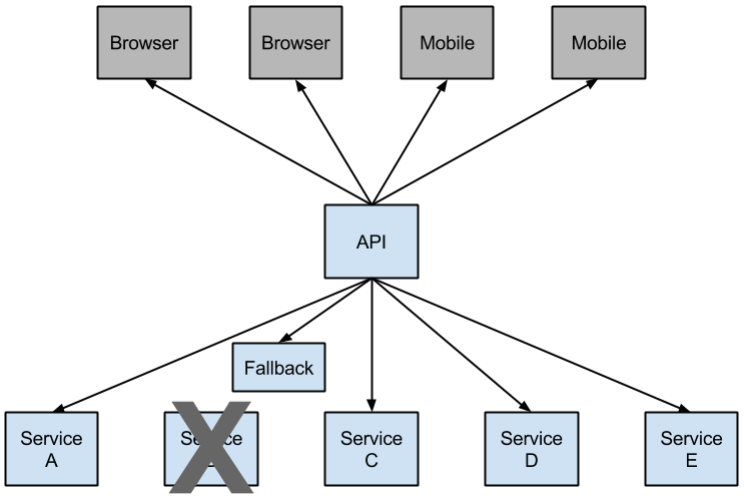
Netflix创建了一个名为Hystrix的库，实现了circuit breaker pattern 。 在微服务体系结构中，通常会有多个服务调用层，如以下示例所示：

图13.1。 微服务图



较低级别的服务中的服务故障可能导致级联故障直至用户。 当对特定服务的呼叫超过`circuitBreaker.requestVolumeThreshold`（默认值：20个请求）并且在由`metrics.rollingStats.timeInMilliseconds`（默认值：10秒）定义的滚动窗口中的故障百分比大于`circuitBreaker.errorThresholdPercentage`（默认值：> 50％）时，电路打开并且呼叫是没做。 在出现错误和开路的情况下，开发人员可以提供回退。

图13.2。 Hystrix回退防止级联失败



开路可以阻止级联故障，并允许服务时间不堪重负或无法恢复。 回退可以是另一个Hystrix受保护的调用，静态数据或合理的空值。 回退可能会链接在一起，以便第一次回退创建一些其他业务电话，然后又回退到静态数据。

13.1 How to Include Hystrix译：13.1如何包含Hystrix

要将Hystrix包含在您的项目中，请使用组`org.springframework.cloud`且工件ID为`spring-cloud-starter-netflix-hystrix`的起动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

以下示例显示了具有Hystrix断路器的最小Eureka服务器：

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}

@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object> parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object> parameters) {
        return /* something useful */;
    }
}
```

`@HystrixCommand`由一个名为"javanica"的Netflix contrib库提供 。 Spring Cloud会自动将带有该注释的Spring bean包装在连接到Hystrix断路器的代理中。 断路器计算何时打开和关闭电路，以及在发生故障时应采取的措施。

要配置`@HystrixCommand`您可以使用`commandProperties`属性和`@HystrixProperty`注释列表。 有关更多详细信息，请参阅here 。 有关可用属性的详细信息，请参阅Hystrix wiki 。

13.2 Propagating the Security Context or Using Spring Scopes译：13.2传播安全上下文或使用Spring作用域

如果你想要一些线程本地上下文传播到`@HystrixCommand`，默认声明不起作用，因为它在线程池中执行命令（在超时的情况下）。 您可以通过配置或直接在注释中使用与呼叫者相同的线程来切换Hystrix，方法是让它使用不同的"隔离策略"。 以下示例演示如何在注释中设置线程：

```
@HystrixCommand(fallbackMethod = "stubMyService", commandProperties = { @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE") } )
...
```

如果您使用`@SessionScope`或`@RequestScope`则同样适用。 如果遇到运行时异常，说明它找不到范围上下文，则需要使用相同的线程。

您还可以选择将`hystrix.shareSecurityContext`属性设置为`true` 。 这样做会自动配置Hystrix并发策略插件钩子，以将`SecurityContext`从主线程传输到Hystrix命令使用的线程。 Hystrix不允许注册多个Hystrix并发策略，因此可以通过将自己的`HystrixConcurrencyStrategy`声明为Spring bean来获得扩展机制。 Spring Cloud在Spring上下文中查找您的实现并将其包装在自己的插件中。

13.3 Health Indicator译：13.3健康指标

连接断路器的状态也暴露在调用应用程序的`/health`端点中，如以下示例所示：

```
{
    "hystrix": {
        "openCircuitBreakers": [
            "StoreIntegration::getStoresByLocationLink"
        ],
        "status": "CIRCUIT_OPEN"
    },
    "status": "UP"
}
```

13.4 Hystrix Metrics Stream译：13.4 Hystrix度量流

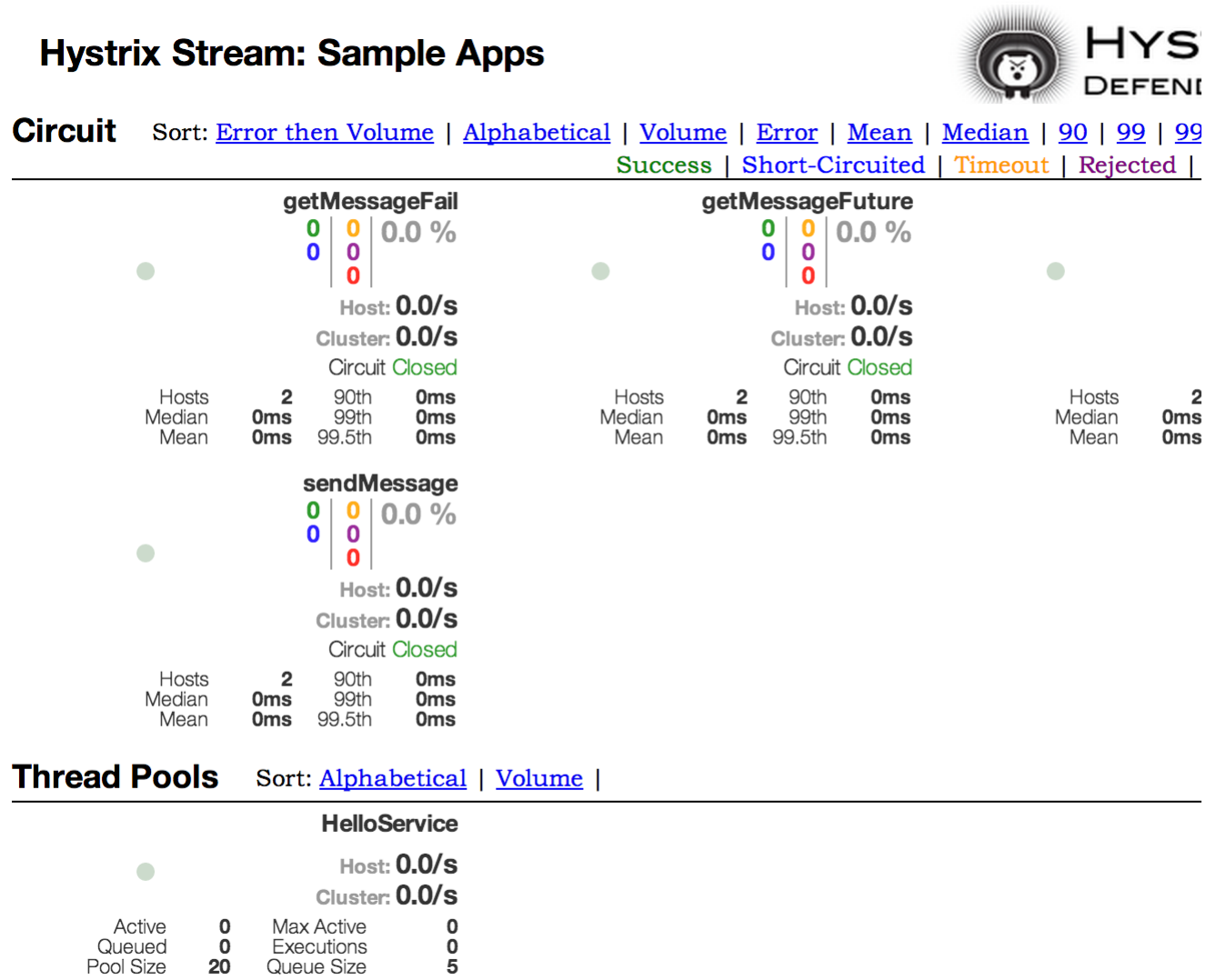
要启用Hystrix度量标准流，请包含对`spring-boot-starter-actuator`的依赖关系并设置`management.endpoints.web.exposure.include: hystrix.stream` 。 这样做`/actuator/hystrix.stream`作为管理端点公开，如以下示例所示：

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
```

14. Circuit Breaker: Hystrix Dashboard译：14.断路器：Hystrix仪表板

Hystrix的主要优势之一是它收集的每个HystrixCommand的度量集合。 Hystrix仪表板以高效的方式显示每个断路器的运行状况。

图14.1。 Hystrix仪表板



15. Hystrix Timeouts And Ribbon Clients 译：17）Hystrix Timeouts和Ribbon客户端

使用包装Ribbon客户端的Hystrix命令时，要确保将Hystrix超时配置为比配置的功能区超时更长，包括可能进行的任何潜在重试。 例如，如果您的功能区连接超时时间为一秒，并且功能区客户端可能会重试该请求三次，那么您的Hystrix超时应略超过三秒。

15.1 How to Include the Hystrix Dashboard 译：15.1如何包含Hystrix仪表板

要将Hystrix仪表板包含在您的项目中，请使用组号为`org.springframework.cloud`且工件ID为`spring-cloud-starter-netflix-hystrix-dashboard`的起动器。 请参阅Spring Cloud Project page以获取有关使用当前Spring Cloud Release Train设置构建系统的详细信息。

要运行Hystrix仪表板，请使用`@EnableHystrixDashboard`注释Spring Boot主类。 然后访问`/hystrix`并将仪表板指向Hystrix客户端应用程序中单个实例的端点`/hystrix.stream`。

连接到使用HTTPS的`/hystrix.stream`端点时，服务器使用的证书必须由JVM信任。 如果证书不可信，您必须将证书导入到JVM中，以便Hystrix仪表板成功连接到流终端。

15.2 Turbine 译：15.2涡轮机

从系统的整体健康角度看，单个实例的Hystrix数据并不是非常有用。 Turbine是一个应用程序，它将所有相关的`/hystrix.stream`端点汇总到用于Hystrix仪表板的组合`/turbine.stream`中。 个别实例通过Eureka定位。 运行涡轮需要使用`@EnableTurbine`注释（例如，使用spring-cloud-starter-netflix-turbine来设置类路径）来标注主类。 所有来自the Turbine 1 wiki的记录配置属性均适用。 唯一的区别是`turbine.instanceUrlSuffix`不需要预先添加端口，因为除非`turbine.instanceInsertPort=false`这是自动处理的。

默认情况下，Turbine通过在Eureka中查找其`hostName`和`port`条目，然后在其上追加`/hystrix.stream`来查找已注册实例上的`/hystrix.stream`端点。 如果实例的元数据包含`management.port`，则将其用于替代`/hystrix.stream`端点的`port`值。 默认情况下，名为`management.port`的元数据条目等于`management.port`配置属性。 可以通过以下配置覆盖它：

```
eureka:
  instance:
    metadata-map:
      management.port: ${management.port:8081}
```

`turbine.appConfig`配置密钥是涡轮机用于查找实例的Eureka服务`turbine.appConfig`列表。 涡轮流然后在Hystrix仪表板中使用，其URL类似于以下内容：

`http://my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME`

如果名称为`default`则可以省略群集参数。 该`cluster`参数必须的表项匹配`turbine.aggregator.clusterConfig`。 Eureka返回的值是大写。 因此，如果有一个名为`customers`的应用程序在Eureka注册，那么以下示例`customers`：

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

如果您需要自定义哪些群集名称应该由Turbine使用（因为您不想在`turbine.aggregator.clusterConfig`配置中存储群集名称），请提供类型为`TurbineClustersProvider`的bean。

可以通过`clusterName`中的SPEL表达式来定制`turbine.clusterNameExpression`，`turbine.clusterNameExpression`其作为`InstanceInfo`的实例。 默认值是`appName`，这意味着尤里卡`serviceId`成为群集键（即，`InstanceInfo`为客户具有`appName`的`CUSTOMERS`）。 另一个示例是`turbine.clusterNameExpression=aSGName`，它从AWS ASG名称获取集群名称。 以下列表显示了另一个示例：

```
turbine:
  aggregator:
    clusterConfig: SYSTEM,USER
  appConfig: customers,stores,ui,admin
  clusterNameExpression: metadata['cluster']
```

在前面的示例中，来自四个服务的集群名称将从其元数据映射中提取，预计其值包括`SYSTEM`和`USER`。

要为所有应用程序使用“默认”群集，您需要一个字符串文本表达式（如果它也位于YAML中，则需要单引号和双引号转义）：

```
turbine:
  appConfig: customers,stores
  clusterNameExpression: "'default'"
```

Spring Cloud提供了一个`spring-cloud-starter-netflix-turbine`，它拥有运行Turbine服务器所需的所有依赖关系。 要对Turnbine做广告，请创建一个Spring Boot应用程序并使用`@EnableTurbine`对其进行`@EnableTurbine`。

默认情况下，Spring Cloud允许Turbine使用主机和端口为每个主机，每个集群允许多个进程。 如果你想在本地Netflix的行为内置到涡轮*不允许*每个主机的多个进程，每簇（关键实例ID是主机名），设置`turbine.combineHostPort=false`。

15.2.1 Clusters Endpoint译：15.2.1集群端点

在某些情况下，其他应用程序可能会知道Turbine中配置了哪些custers。 为了支持这一点，您可以使用`/clusters`端点，该端点将返回所有配置群集的JSON数组。

GET /集群。

```
[
  {
    "name": "RACES",
    "link": "http://localhost:8383/turbine.stream?cluster=RACES"
  },
  {
    "name": "WEB",
    "link": "http://localhost:8383/turbine.stream?cluster=WEB"
  }
]
```

通过将`turbine.endpoints.clusters.enabled`设置为`false`可以禁用此端点。

15.3 Turbine Stream译：15.3涡轮流

在某些环境中（例如在PaaS设置中），从所有分布式Hystrix命令中提取度量标准的传统Turbine模型不起作用。 在这种情况下，您可能希望让您的Hystrix命令将指标推送到Turbine。 Spring Cloud支持通过消息传递。 要在客户端上执行此操作，请将依赖项添加到您选择的`spring-cloud-netflix-hystrix-stream`和`spring-cloud-starter-stream-*`。 有关代理的详细信息以及如何配置客户端凭证，请参阅Spring Cloud Stream documentation。 它应该为当地经纪人开箱即用。

在服务器端，创建一个Spring Boot应用程序并使用`@EnableTurbineStream`对其进行`@EnableTurbineStream`。 Turbine Stream服务器需要使用Spring Webflux，因此需要将`spring-boot-starter-webflux`包含在您的项目中。 默认情况下`spring-boot-starter-webflux`当添加包含`spring-cloud-starter-netflix-turbine-stream`到您的应用程序。

然后，您可以将Hystrix仪表板指向Turbine Stream Server，而不是单独的Hystrix流。 如果Turbine Stream在myhost上的端口8989上运行，则将`http://myhost:8989`放在Hystrix仪表板的流输入字段中。 电路前缀分别为`serviceId`，后跟一个点（`.`），然后为电路名称。

Spring Cloud提供了一个`spring-cloud-starter-netflix-turbine-stream`，它具有运行Turbine Stream服务器所需的所有依赖关系。 然后，您可以添加您所选择的流绑定器，例如`spring-cloud-starter-stream-rabbit`。

16. Client Side Load Balancer: Ribbon译：16客户端负载均衡器：功能区

功能区是一个客户端负载均衡器，可以让您对HTTP和TCP客户端的行为有很多控制权。 费恩已经使用功能区，所以，如果您使用`@FeignClient`，本节也适用。

Ribbon中的一个中心概念是指定客户的概念。 每个负载均衡器都是组合的一部分，它们一起工作以根据需要联系远程服务器，并且该集合具有作为应用程序开发人员提供的名称（例如，通过使用`@FeignClient`注释）。 根据需要，Spring Cloud通过使用`RibbonClientConfiguration`为每个指定客户端创建`ApplicationContext`新`RibbonClientConfiguration`。 这包含（除其他事项外）的`ILoadBalancer`，一个`RestClient`和`ServerListFilter`。

16.1 How to Include Ribbon译：16.1如何包含功能区

要在您的项目中包含功能区，请使用组ID为`org.springframework.cloud`且工件ID为`spring-cloud-starter-netflix-ribbon`的起动器。 有关使用当前的Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page。

16.2 Customizing the Ribbon Client译：16.2自定义功能区客户端

您可以使用`<client>.ribbon.*`外部属性来配置Ribbon客户端的某些位，这与使用Netflix API本地类似，不同之处在于您可以使用Spring Boot配置文件。 原生选项可以作为`CommonClientConfigKey`（ribbon-core的一部分）中的静态字段进行检查。

Spring Cloud还允许您通过使用`@RibbonClient`声明其他配置（在`RibbonClientConfiguration`顶部）来完全控制客户端，如以下示例所示：

```
@Configuration
@RibbonClient(name = "custom", configuration = CustomConfiguration.class)
public class TestConfiguration {
}
```

在这种情况下，客户`RibbonClientConfiguration`的组件和`RibbonClientConfiguration`任何`CustomConfiguration`（后者通常覆盖前者）组成。

该`CustomConfiguration` CLAS必须为`@Configuration`类，但要小心，它是不是在`@ComponentScan`主应用程序上下文。 否则，它会被所有的`@RibbonClients`共享。 如果使用`@ComponentScan`（或`@SpringBootApplication`），则需要采取措施以避免将其包含在内（例如，可以将其放在单独的，不重叠的包中，或指定要在`@ComponentScan`明确扫描的包）。

下表显示了Spring Cloud Netflix默认为功能区提供的bean：

Bean Type	Bean Name	Class Name
`IClientConfig`	`ribbonClientConfig`	`DefaultClientConfigImpl`
`IRule`	`ribbonRule`	`ZoneAvoidanceRule`
`IPing`	`ribbonPing`	`DummyPing`
`ServerList<Server>`	`ribbonServerList`	`ConfigurationBasedServerList`
`ServerListFilter<Server>`	`ribbonServerListFilter`	`ZonePreferenceServerListFilter`
`ILoadBalancer`	`ribbonLoadBalancer`	`ZoneAwareLoadBalancer`
`ServerListUpdater`	`ribbonServerListUpdater`	`PollingServerListUpdater`

创建这些类型的bean并将其放入`@RibbonClient`配置（例如上面的`FooConfiguration`）中，可以覆盖所描述的每个Bean，如以下示例所示：

```
@Configuration
protected static class FooConfiguration {
 @Bean
 public ZonePreferenceServerListFilter serverListFilter() {
  ZonePreferenceServerListFilter filter = new ZonePreferenceServerListFilter();
  filter.setZone("myTestZone");
  return filter;
 }

 @Bean
 public IPing ribbonPing() {
  return new PingUrl();
 }
}
```

所述包括在前面的示例语句来替换`NoOpPing`与`PingUrl`，并提供一个自定义`serverListFilter`。

16.3 Customizing the Default for All Ribbon Clients译：16.3自定义所有功能区客户端的默认设置

可以通过使用`@RibbonClients`注释并注册默认配置为所有功能区客户端提供默认配置，如以下示例所示：

```
@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)
public class RibbonClientDefaultConfigurationTestsConfig {

 public static class BazServiceList extends ConfigurationBasedServerList {
  public BazServiceList(IClientConfig config) {
   super.initWithNiwsConfig(config);
  }
 }
}

@Configuration
class DefaultRibbonConfig {

 @Bean
 public IRule ribbonRule() {
  return new BestAvailableRule();
 }

 @Bean
 public IPing ribbonPing() {
  return new PingUrl();
 }

 @Bean
 public ServerList<Server> ribbonServerList(IClientConfig config) {
  return new RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);
 }

 @Bean
 public ServerListSubsetFilter serverListFilter() {
  ServerListSubsetFilter filter = new ServerListSubsetFilter();
  return filter;
 }

}
```

16.4 Customizing the Ribbon Client by Setting Properties译：16.4通过设置属性自定义功能区客户端

从1.2.0版开始，Spring Cloud Netflix现在支持通过将属性设置为与 Ribbon documentation兼容来自定义功能区客户端。

这使您可以在不同的环境中启动时更改行为。

以下列表显示了支持的属性>：

- `<clientName>.ribbon.NFLoadBalancerClassName`: Should implement `ILoadBalancer`
- `<clientName>.ribbon.NFLoadBalancerRuleClassName`: Should implement `IRule`
- `<clientName>.ribbon.NFLoadBalancerPingClassName`: Should implement `IPing`
- `<clientName>.ribbon.NIWSServerListClassName`: Should implement `ServerList`
- `<clientName>.ribbon.NIWSServerListFilterClassName`: Should implement `ServerListFilter`

这些属性中定义的类优先于使用 `@RibbonClient(configuration=MyRibbonConfig.class)` 定义的 `@RibbonClient(configuration=MyRibbonConfig.class)` 和Spring Cloud Netflix提供的默认值。

要为 `IRule` 设置名为 `users` 的服务名称，可以设置以下属性：

application.yml。

```
users:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule
```

有关Ribbon提供的实现，请参阅 Ribbon documentation 。

16.5 Using Ribbon with Eureka译：16.5与 Eureka 一起使用 Ribbon

当Eureka与Ribbon一起使用时（即两者都在类路径中），`ribbonServerList` 将被覆盖，扩展名为 `DiscoveryEnabledNIWSServerList` ，它将填充Eureka的服务器列表。 它还将 `IPing` 接口替换为代表Eureka的 `NIWSDiscoveryPing` ，以确定服务器是否启动。 `ServerList` 默认安装的是 `DomainExtractingServerList` 。 其目的是在不使用AWS AMI元数据（这是Netflix依赖的）的情况下为负载平衡器提供元数据。 默认情况下，服务器列表是使用实例元数据中提供的“区域”信息构建的（因此，在远程客户端上，设置为 `eureka.instance.metadataMap.zone` ）。 如果缺少该设置并且 `approximateZoneFromHostname` 标志已设置，则可以使用服务器主机名中的域名作为该区域的代理。 一旦区域信息可用，它可以在 `ServerListFilter` 。 默认情况下，它用于定位与客户端位于同一区域的服务器，因为默认值为 `ZonePreferenceServerListFilter` 。 默认情况下，客户端区域的确定方式与远程实例相同（即，通过 `eureka.instance.metadataMap.zone` ）。

用于设置客户端区域的正统“archaius”方法是通过名为“@zone”的配置属性。 如果可用，Spring Cloud将优先于所有其他设置（请注意，密钥必须在YAML配置中引用）。

如果没有其他区域数据源，则根据客户端配置（与实例配置相反）进行猜测。 我们取 `eureka.client.availabilityZones` ，这是一个从区域名称到区域列表的地图，并将实例本身区域的第一个区域（即 `eureka.client.region` ，默认为“us-east-1”）与原生Netflix兼容）。

16.6 Example: How to Use Ribbon Without Eureka译：16.6示例：如何在没有尤里卡的情况下使用色带

Eureka是一种抽象发现远程服务器的便捷方式，因此您无需在客户端对其URL进行硬编码。 但是，如果您不想使用Eureka，Ribbon和Feign也可以使用。 假设你已经为“商店”声明了一个 `@RibbonClient` ，并且Eureka没有被使用（甚至不在类路径中）。 功能区客户端默认为已配置的服务器列表。 您可以按如下方式提供配置：

application.yml。

```
stores:
  ribbon:
    listOfServers: example.com,google.com
```

16.7 Example: Disable Eureka Use in Ribbon译：16.7示例：禁用功能区中的Eureka使用

将 `ribbon.eureka.enabled` 属性设置为 `false` 明确禁用在功能区中使用Eureka，如下例所示：

application.yml。

```
ribbon:
  eureka:
   enabled: false
```

16.8 Using the Ribbon API Directly译：16.8直接使用 Ribbon API

您也可以直接使用 `LoadBalancerClient` ，如以下示例所示：

```
public class MyClass {
    @Autowired
    private LoadBalancerClient loadBalancer;

    public void doStuff() {
        ServiceInstance instance = loadBalancer.choose("stores");
        URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), instance.getPort()));
        // ... do something with the URI
    }
}
```

16.9 Caching of Ribbon Configuration译：16.9缓存色带配置

每个名为客户端的Ribbon都有一个Spring Cloud维护的对应的子应用程序上下文。 这个应用程序上下文在第一次请求到客户端时被延迟加载。 通过指定Ribbon客户端的名称，可以将此延迟加载行为更改为在启动时急切加载这些子应用程序上下文，如下例所示：

application.yml。

```
ribbon:
  eager-load:
    enabled: true
    clients: client1, client2, client3
```

16.10 How to Configure Hystrix Thread Pools译：16.10如何配置Hystrix线程池

如果将 `zuul.ribbonIsolationStrategy` 更改为 `THREAD` ，则Hystrix的线程隔离策略将用于所有路由。 在这种情况下， `HystrixThreadPoolKey` 默认设置为 `RibbonCommand` 。 这意味着所有路由的HystrixCommands都在相同的Hystrix线程池中执行。 此行为可以通过以下配置进行更改：

application.yml。

```
zuul:
  threadPool:
    useSeparateThreadPools: true
```

前面的示例导致在每个路由的Hystrix线程池中执行HystrixCommands。

在这种情况下，默认 `HystrixThreadPoolKey` 与每个路由的服务ID相同。 要将前缀添加到 `HystrixThreadPoolKey` ，请将 `zuul.threadPool.threadPoolKeyPrefix` 设置为您要添加的值，如下例所示：

application.yml。

```
zuul:
  threadPool:
    useSeparateThreadPools: true
    threadPoolKeyPrefix: zuulgw
```

16.11 How to Provide a Key to Ribbon's `IRule`译：16.11如何为 Ribbon提供密钥 `IRule`

如果您需要提供自己的 `IRule` 实现来处理像â€œcanaryâ€测试一种特殊的路由需求，传递一些信息给 `choose` 的方法 `IRule` 。

com.netflix.loadbalancer.IRule.java。

```
public interface IRule{
    public Server choose(Object key);
         :
```

您可以提供一些 `IRule` 实现用来选择目标服务器的信息，如以下示例所示：

```
RequestContext.getCurrentContext()
              .set(FilterConstants.LOAD_BALANCER_KEY, "canary-test");
```

如果您使用 `RequestContext` 的密钥将任何对象放入 `FilterConstants.LOAD_BALANCER_KEY` ，它将传递给 `IRule` 实现的 `choose` 方法。 前面示例中显示的代码必须在执行 `RibbonRoutingFilter` 之前执行。 Zuul的预过滤器是最好的地方。 您可以通过预过滤器中的 `RequestContext` 访问HTTP标题和查询参数，以便可以使用它来确定传递给功能区的 `LOAD_BALANCER_KEY` 。 如果您在 `RequestContext` 未将 `LOAD_BALANCER_KEY` 与任何值 `LOAD_BALANCER_KEY` ， `RequestContext` null作为 `choose` 方法的参数传递。

17. External Configuration: Archaius译：17.外部配置：Archaius

Archaius是Netflix客户端配置库。 它是所有Netflix OSS组件用于配置的库。 Archaius是Apache Commons Configuration项目的延伸。 它允许更改配置，方法是轮询源更改

或者让源更改为客户端。 Archaius使用Dynamic <Type> Property类作为属性的句柄，如以下示例所示：

Archaius例子。

```
class ArchaiusTest {
    DynamicStringProperty myprop = DynamicPropertyFactory
            .getInstance()
            .getStringProperty("my.prop");

    void doSomething() {
        OtherClass.someMethod(myprop.get());
    }
}
```

Archaius有它自己的一套配置文件和加载优先级。 Spring应用程序通常不应该直接使用Archaius，但需要本地配置Netflix工具。 Spring Cloud有一个Spring环境桥，以便Archaius可以从Spring环境读取属性。 这个桥允许Spring Boot项目使用正常的配置工具链，同时让他们按照文档（大部分）配置Netflix工具。

18. Router and Filter: Zuul 译：18.路由器和过滤器：Zuul

路由是微服务体系结构的组成部分。 例如， `/` 可能会映射到您的Web应用程序， `/api/users` 映射到用户服务并且 `/api/shop` 映射到商店服务。 Zuul是来自Netflix的基于JVM的路由器和服务器端负载均衡器。

Netflix uses Zuul以下内容：

- Authentication
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing
- Service Migration
- Load Shedding
- Security
- Static Response handling
- Active/Active traffic management

Zuul的规则引擎允许规则和过滤器以基本上任何JVM语言编写，并具有对Java和Groovy的内置支持。

配置属性 `zuul.max.host.connections` 已被两个新属性 `zuul.host.maxTotalConnections` 和 `zuul.host.maxPerRouteConnections` ，默认值分别为200和20。

所有路线的默认Hystrix隔离模式（ `ExecutionIsolationStrategy` ）为 `SEMAPHORE` 。 如果首选隔离模式，则可以将 `zuul.ribbonIsolationStrategy` 更改为 `THREAD` 。

18.1 How to Include Zuul 译：18.1如何包含Zuul

要将Zuul加入到您的项目中，请使用组号为 `org.springframework.cloud` 的起动器和工件ID为 `spring-cloud-starter-netflix-zuul` 的起动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

18.2 Embedded Zuul Reverse Proxy 译：18.2嵌入式Zuul反向代理

Spring Cloud创建了一个嵌入式Zuul代理，以简化UI应用程序想要对一个或多个后端服务进行代理调用的常见用例的开发。 此功能对于用户界面代理其所需的后端服务非常有用，从而避免需要独立管理所有后端的CORS和身份验证问题。

要启用它，请使用 `@EnableZuulProxy` 注释Spring Boot主类。 这样做会导致本地电话转到相应的服务。 按照惯例，ID为 `users` 的服务接收来自位于 `/users` 的代理的请求（前缀已剥离）。 该代理使用功能区来查找要通过发现转发的实例。 所有请求都在hystrix command中执行，因此失败会显示在Hystrix度量标准中。 一旦电路打开，代理不会尝试联系服务。

Zuul初学者不包含发现客户端，因此，对于基于服务ID的路由，您还需要在类路径中提供其中的一个（Eureka是一种选择）。

要跳过自动添加服务，请将 `zuul.ignored-services` 设置为服务标识模式列表。 如果某个服务匹配的模式被忽略，但也包含在显式配置的路由映射中，则该模式将被忽略，如以下示例所示：

application.yml。

```
zuul:
 ignoredServices: '*'
 routes:
   users: /myusers/**
```

在上例中， **除** `users` **之外的**所有服务都被忽略。

要增加或更改代理路由，可以添加外部配置，如下所示：

application.yml。

```
zuul:
 routes:
   users: /myusers/**
```

前面的例子意味着 `/myusers` HTTP调用被转发到 `users` 服务（例如 `/myusers/101` 被转发到 `/101` ）。

为了更好地控制路由，可以单独指定路径和serviceId，如下所示：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
     serviceId: users_service
```

前面的例子意味着`/myusers` HTTP调用被转发到`users_service`服务。 该路线必须有`path` ，可以指定为蚂蚁样式，因此`/myusers/*`只匹配一个级别，但`/myusers/**`分层次匹配。

后端的位置可以指定为`serviceId` （用于来自发现的服务）或`url` （用于物理位置），如以下示例所示：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
     url: http://example.com/users_service
```

这些简单的url路由不会作为`HystrixCommand`执行，也不会使用功能区来平衡多个URL。 为了实现这些目标，您可以指定带有静态服务器列表的`serviceId` ，如下所示：

application.yml。

```
zuul:
  routes:
    echo:
      path: /myusers/**
      serviceId: myusers-service
      stripPrefix: true

hystrix:
  command:
    myusers-service:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: ...

myusers-service:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    listOfServers: http://example1.com,http://example2.com
    ConnectTimeout: 1000
    ReadTimeout: 3000
    MaxTotalHttpConnections: 500
    MaxConnectionsPerHost: 100
```

另一种方法是指定服务路由并为`serviceId`配置功能区客户端（这样做需要在功能区中禁用Eureka支持 - 请参阅 above for more information ），如以下示例所示：

application.yml。

```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users

ribbon:
  eureka:
    enabled: false

users:
  ribbon:
    listOfServers: example.com,google.com
```

您可以使用`regexmapper`在`serviceId`和路线之间提供约定。 它使用正则表达式命名的组来提取`serviceId`变量并将它们注入路由模式，如以下示例所示：

ApplicationConfiguration.java。

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
        "(?<name>^.+)-(?<version>v.+$)",
        "${version}/${name}");
}
```

前面的例子意味着`serviceId`的`myusers-v1`被映射到路由`/v1/myusers/**` 。 任何正则表达式都被接受，但所有已命名的组必须出现在`servicePattern`和`routePattern` 。 如果`servicePattern`不匹配`serviceId` ，则使用默认行为。 在前面的示例中，将`serviceId`的`myusers`映射到“/ myusers / **”路由（未检测到任何版本）。 此功能在默认情况下处于禁用状态，仅适用于发现的服务。

要为所有映射添加前缀，请将`zuul.prefix`设置为值，例如`/api` 。 默认情况下，代理前缀在请求被转发之前从请求中剥离（您可以使用`zuul.stripPrefix=false`关闭此行为）。 您还可以关闭从单个路由中删除特定于服务的前缀，如以下示例所示：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
     stripPrefix: false
```

`zuul.stripPrefix`仅适用于在`zuul.prefix`设置的前缀。 它不会对一个给定的routeâ€™的中定义的前缀任何影响`path` 。

在前面的例子中，请求`/myusers/101`转发到`/myusers/101`上`users`服务。

`zuul.routes`条目实际上绑定到类型为`ZuulProperties`的对象。 如果您查看该对象的属性，则可以看到它也有一个`retryable`标志。 将该标志设置为`true`以使Ribbon客户端自动重试失败的请求。 当需要修改使用功能区客户端配置的重试操作的参数时，您还可以将该标志设置为`true` 。

默认情况下， `X-Forwarded-Host`标题被添加到转发的请求中。 要关闭它，请设置`zuul.addProxyHeaders = false` 。 默认情况下，前缀路径被剥离，并且对后端的请求将拾取`X-Forwarded-Prefix`头（ `/myusers`示例中为`/myusers` ）。

如果您设置了默认路由（ `/` ），则具有`@EnableZuulProxy`的应用程序可以充当独立服务器。 例如， `zuul.route.home: /`会将所有流量（“/ **”）路由到“主页”服务。

如果需要更细粒度的忽略，则可以指定要忽略的特定模式。 这些模式在路由定位过程的开始时进行评估，这意味着模式中应包含前缀以保证匹配。 忽略的模式跨越所有服务并取代任何其他路由规范。 以下示例显示如何创建忽略模式：

application.yml。

```
zuul:
 ignoredPatterns: /**/admin/**
 routes:
   users: /myusers/**
```

前面的例子是指所有的呼叫（例如`/myusers/101` ）被转发到`/101`上`users`服务。 但是，包括`/admin/`在内的`/admin/`无法解析。

如果您需要您的路线保留其订单，则需要使用YAML文件，因为在使用属性文件时订单会丢失。 以下示例显示了这样一个YAML文件：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
   legacy:
     path: /**
```

如果要使用属性文件，则`legacy`路径可能会最终位于`users`路径的前面，从而`users`无法访问`users`路径。

18.3 Zuul Http Client译：18.3 Zuul Http客户端

Zuul使用的默认HTTP客户端现在由Apache HTTP Client支持，而不是已弃用的功能区`RestClient` 。 要使用`RestClient`或者`okhttp3.OkHttpClient` ，设置`ribbon.restclient.enabled=true`或者`ribbon.okhttp.enabled=true`分别。 如果您想定制Apache HTTP客户端或OK HTTP客户端，请提供类型为`ClosableHttpClient`或`OkHttpClient`的bean。

18.4 Cookies and Sensitive Headers译：18.4饼干和敏感头

您可以在同一系统中的服务之间共享标头，但您可能不希望敏感标头下游泄漏到外部服务器中。 您可以指定一个忽略的标题列表作为路由配置的一部分。 Cookie起着特殊的作用，因为它们在浏览器中有明确定义的语义，并且它们始终被视为敏感。 如果代理的用户是浏览器，那么下游服务的Cookie也会给用户带来问题，因为他们都混杂在一起（所有下游服务看起来都是来自同一个地方）。

如果您对服务的设计非常小心（例如，如果只有其中一个下游服务设置了cookie），则可能会让它们从后端一直流向调用方。 此外，如果您的代理设置了Cookie并且所有后端服务都是同一系统的一部分，那么简单地共享这些服务可能很自然（例如，使用Spring Session将它们链接到某个共享状态）。 除此之外，由下游服务设置的任何Cookie可能对呼叫者无用，因此建议您至少将`Set-Cookie`和`Cookie`为不属于您的域的路由的敏感标头。 即使对于属于您的域名的路由，也要尽量仔细考虑它们在让Cookie和代理之间流动Cookie之前的含义。

敏感标题可以配置为每个路由的逗号分隔列表，如以下示例所示：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders: Cookie,Set-Cookie,Authorization
     url: https://downstream
```

这是`sensitiveHeaders`的默认值，所以除非您希望它不同，否则不需要设置它。 这是Spring Cloud Netflix 1.1中的新功能（在1.0中，用户无法控制标题，并且所有Cookie均在双向流动）。

`sensitiveHeaders`是黑名单，默认不是空的。 因此，要使Zuul发送所有标题（ `ignored`除外），必须明确将其设置为空列表。 如果您想要将Cookie或授权标头传递给后端，则必须这样做。 以下示例显示如何使用`sensitiveHeaders` ：

application.yml。

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders:
     url: https://downstream
```

您还可以通过设置`zuul.sensitiveHeaders`来设置敏感标题。 如果在路由上设置了`sensitiveHeaders` ，则将覆盖全局`sensitiveHeaders`设置。

18.5 Ignored Headers译：18.5忽略标题

除了路由敏感报头之外，还可以为与下游服务交互期间应丢弃的值（请求和响应）设置一个名为`zuul.ignoredHeaders`的全局值。 默认情况下，如果Spring Security不在类路径中，它们是空的。 否则，它们将被初始化为Spring Security指定的一组众所周知的“安全性”头文件（例如，涉及缓存）。 在这种情况下，假设下游服务也可能会添加这些标头，但我们需要来自代理的值。 为了在Spring Security位于类路径中时不丢弃这些众所周知的安全性头文件，可以将`zuul.ignoreSecurityHeaders`设置为`false` 。 如果您在Spring Security中禁用了HTTP安全响应标头，并且需要下游服务提供的值，那么这样做会很有用。

18.6 Management Endpoints译：18.6管理端点

默认情况下，如果将 `@EnableZuulProxy` 与Spring Boot Actuator配合使用，则可以启用两个附加端点：

- Routes
- Filters

18.6.1 Routes Endpoint 译：18.6.1路由端点

GET路由端点 `/routes` 返回映射路由列表：

GET /路线。

```
{
  /stores/**: "http://localhost:8081"
}
```

其它路线详情可以通过添加请求 `?format=details` 查询字符串 `/routes` 。 这样做会产生以下输出：

GET / routes / details。

```
{
  "/stores/**": {
    "id": "stores",
    "fullPath": "/stores/**",
    "location": "http://localhost:8081",
    "path": "/**",
    "prefix": "/stores",
    "retryable": false,
    "customSensitiveHeaders": false,
    "prefixStripped": true
  }
}
```

`POST` 至 `/routes` 强制刷新现有路由（例如，服务目录中有更改时）。 您可以通过设置禁用此端点 `endpoints.routes.enabled` 至 `false` 。

路线应自动响应服务目录中的更改，但 `POST` 至 `/routes` 是强制变更立即发生的一种方式。

18.6.2 Filters Endpoint 译：18.6.2过滤器端点

甲 `GET` 到滤波器端点在 `/filters` 返回由类型地图上Zuul滤波器。 对于地图中的每个过滤器类型，您将获得该类型的所有过滤器的列表及其详细信息。

18.7 Strangulation Patterns and Local Forwards 译：18.7扼杀模式和本地前锋

迁移现有应用程序或API时的常见模式是“strangle”旧端点，用不同的实现慢慢替换它们。 Zuul代理是一个有用的工具，因为您可以使用它来处理来自旧端点的客户端的所有流量，但将一些请求重定向到新端点。

以下示例显示“strangle”场景的配置详细信息：

application.yml。

```
zuul:
 routes:
   first:
     path: /first/**
     url: http://first.example.com
   second:
     path: /second/**
     url: forward:/second
   third:
     path: /third/**
     url: forward:/3rd
   legacy:
     path: /**
     url: http://legacy.example.com
```

在前面的示例中，我们正在扼杀“legacy”应用程序，该应用程序映射到与其他模式不匹配的所有请求。 已使用外部URL将 `/first/**` 路径提取到新服务中。 `/second/**` 中的路径被转发，以便它们可以在本地处理（例如，使用普通的Spring `@RequestMapping` ）。 `/third/**` 中的路径也被转发，但具有不同的前缀（ `/third/foo` 转发给 `/3rd/foo` ）。

被忽略的模式没有被完全忽略，它们只是不被代理处理（所以它们也被有效地本地转发）。

18.8 Uploading Files through Zuul 译：18.8通过Zuul上传文件

如果使用 `@EnableZuulProxy` ，则可以使用代理路径上传文件，只要文件很小，文件就可以工作。 对于大文件，在“/ zuul / *”中有一条绕过Spring `DispatcherServlet` （以避免多部分处理）的替代路径。 换句话说，如果你有 `zuul.routes.customers=/customers/**` ，那么你可以把 `POST` 大文件改为 `/zuul/customers/*` 。 servlet路径通过 `zuul.servletPath` 外 `zuul.servletPath` 。 如果代理路由引导您通过功能区负载平衡器，则超大型文件还需要提升超时设置，如以下示例所示：

application.yml。

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

请注意，要使流式处理大型文件，您需要在请求中使用分块编码（某些浏览器默认不执行此操作），如以下示例所示：

```
$ curl -v -H "Transfer-Encoding: chunked" \
    -F "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

18.9 Query String Encoding译：18.9查询字符串编码

处理传入请求时，查询参数将被解码，以便它们可用于Zuul过滤器中的可能修改。 然后对它们进行重新编码，在路由过滤器中重建后端请求。 如果（例如）使用Javascript的`encodeURIComponent()`方法编码，结果可能与原始输入不同。 虽然这在大多数情况下不会造成任何问题，但某些Web服务器可能会对复杂的查询字符串进行编码。

要强制查询字符串的原始编码，可以将特殊标志传递给`ZuulProperties`以便查询字符串按照`HttpServletRequest::getQueryString`方法的`HttpServletRequest::getQueryString`，如以下示例所示：

application.yml。

```
zuul:
 forceOriginalQueryStringEncoding: true
```

此特殊标志仅适用于`SimpleHostRoutingFilter`。 此外，由于查询字符串现在直接在原始`HttpServletRequest`上获取，因此您无法轻松覆盖`RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)`查询参数。

18.10 Plain Embedded Zuul译：18.10平原嵌入Zuul

如果您使用`@EnableZuulServer`（而不是`@EnableZuulProxy`），则还可以运行Zuul服务器而无需代理或选择性地切换代理平台的某些部分。 任何添加到类型`ZuulFilter`应用程序的`ZuulFilter`自动安装（与`@EnableZuulProxy`），但不会自动添加任何代理筛选器。

在这种情况下，通过配置“zuul.routes。*”来指定进入Zuul服务器的路由，但是没有服务发现和代理。 因此，“serviceId”和“url”设置将被忽略。 以下示例将“/ api / **”中的所有路径映射到Zuul过滤器链：

application.yml。

```
zuul:
 routes:
   api: /api/**
```

18.11 Disable Zuul Filters译：18.11禁用Zuul过滤器

针对Spring Cloud的Zuul在代理和服务器模式下都配备了默认启用的`ZuulFilter`豆。 有关您可以启用的过滤器列表，请参阅the Zuul filters package 。 如果你想禁用一个，请设置`zuul.<SimpleClassName>.<filterType>.disable=true`。 按照惯例， `filters`之后的包是Zuul过滤器类型。 例如，要禁用`org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter`，请设置`zuul.SendResponseFilter.post.disable=true`。

18.12 Providing Hystrix Fallbacks For Routes译：17126487173038418.12提供Hystrix路由回退

当Zuul中给定路由的电路跳闸时，您可以通过创建`FallbackProvider`类型的bean来提供回退响应。 在此Bean中，您需要指定后备的路由ID，并提供`ClientHttpResponse`作为后备返回。 以下示例显示了一个相对简单的`FallbackProvider`实现：

```
class MyFallbackProvider implements FallbackProvider {

    @Override
    public String getRoute() {
        return "customers";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {
        if (cause instanceof HystrixTimeoutException) {
            return response(HttpStatus.GATEWAY_TIMEOUT);
        } else {
            return response(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    private ClientHttpResponse response(final HttpStatus status) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return status;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return status.value();
            }

            @Override
            public String getStatusText() throws IOException {
                return status.getReasonPhrase();
            }

            @Override
            public void close() {
            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

以下示例显示了如何显示前面示例的路由配置：

```
zuul:
  routes:
    customers: /customers/**
```

如果要为所有路由提供默认回退，可以创建类型为 `FallbackProvider` 的Bean，并让 `getRoute` 方法返回 `*` 或 `null` ，如以下示例中所示：

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable throwable) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

18.13 Zuul Timeouts 译：18.13 Zuul超时

如果您想配置套接字超时以及通过Zuul代理请求读取超时，则根据您的配置，您有两个选项：

- If Zuul uses service discovery, you need to configure these timeouts with the `ribbon.ReadTimeout` and `ribbon.SocketTimeout` Ribbon properties.

如果您通过指定URL来配置Zuul路由，则需要使用 `zuul.host.connect-timeout-millis` 和 `zuul.host.socket-timeout-millis` 。

18.14 Rewriting the `Location` header 译：18.14重写 `Location` 标题

如果Zuul面向Web应用程序，则当Web应用程序通过 `3XX` 的HTTP状态代码重定向时，可能需要重新编写 `Location` 标头。 否则，浏览器将重定向到Web应用程序的URL而不是Zuul URL。 您可以配置 `LocationRewriteFilter` Zuul过滤器，以将 `Location` 标头重新写入Zuul的URL。 它还添加了剥离的全局和特定于路由的前缀。 以下示例使用Spring配置文件添加过滤器：

```
import org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;
...

@Configuration
@EnableZuulProxy
public class ZuulConfig {
    @Bean
    public LocationRewriteFilter locationRewriteFilter() {
        return new LocationRewriteFilter();
    }
}
```

Caution

仔细使用此过滤器。 该过滤器将作用于所有 `3XX` 响应代码的 `Location` 标头，这可能不适用于所有情况，例如将用户重定向到外部URL时。

18.15 Metrics 译：18.15度量

Zuul将在执行器度量标准端点下为路由请求时可能发生的任何故障提供度量标准。 这些指标可以通过点击 `/actuator/metrics` 查看。 指标的名称格式为 `ZUUL::EXCEPTION:errorCause:statusCode` 。

18.16 Zuul Developer Guide 译：18.16 Zuul开发人员指南

有关Zuul如何工作的一般概述，请参阅 the Zuul Wiki 。

18.16.1 The Zuul Servlet 译：18.16.1 Zuul Servlet

Zuul被实现为Servlet。 对于一般情况，Zuul嵌入到Spring Dispatch机制中。 这让Spring MVC可以控制路由。 在这种情况下，Zuul缓冲请求。 如果需要在没有缓冲请求的情况下通过Zuul（例如，对于大文件上传），Servlet也会安装在Spring Dispatcher之外。 默认情况下，该servlet的地址为 `/zuul` 。 该路径可以使用 `zuul.servlet-path` 属性进行更改。

18.16.2 Zuul RequestContext 译：18.16.2 Zuul RequestContext

为了在过滤器之间传递信息，Zuul使用`RequestContext`。 其数据保存在特定于每个请求的`ThreadLocal`。 有关在哪里路由请求，错误以及实际`HttpServletRequest`和`HttpServletResponse`的信息存储在那里。 `RequestContext`扩展了`ConcurrentHashMap`，所以任何东西都可以存储在上下文中。 `FilterConstants`包含由Spring Cloud Netflix安装的过滤器所使用的密钥（更多内容参见这些later）。

18.16.3 `@EnableZuulProxy` vs. `@EnableZuulServer` 译：18.16.3 `@EnableZuulProxy`与`@EnableZuulServer`

Spring Cloud Netflix安装了许多过滤器，具体取决于使用哪种注释来启用Zuul。 `@EnableZuulProxy`是的超集`@EnableZuulServer`。 换句话说，`@EnableZuulProxy`包含由`@EnableZuulServer`安装的所有过滤器。 “proxy”中的附加过滤器启用路由功能。 如果你想要一个“空白”Zuul，你应该使用`@EnableZuulServer`。

18.16.4 `@EnableZuulServer` Filters 译：18.16.4 `@EnableZuulServer`过滤器

`@EnableZuulServer`创建了一个`SimpleRouteLocator`，用于从Spring Boot配置文件加载路由定义。

安装了以下过滤器（与普通的Spring Beans一样）：

- 预过滤器：
  - `ServletDetectionFilter`: Detects whether the request is through the Spring Dispatcher. Sets a boolean with a key of `FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY`.
  - `FormBodyWrapperFilter`: Parses form data and re-encodes it for downstream requests.
  - `DebugFilter`: If the `debug` request parameter is set, sets `RequestContext.setDebugRouting()` and `RequestContext.setDebugRequest()` to `true`. *Route filters:
  - `SendForwardFilter`: Forwards requests by using the Servlet `RequestDispatcher`. The forwarding location is stored in the `RequestContext` attribute, `FilterConstants.FORWARD_TO_KEY`. This is useful for forwarding to endpoints in the current application.
- 后过滤器：
  - `SendResponseFilter`: Writes responses from proxied requests to the current response.
- 错误过滤器：
  - `SendErrorFilter`: Forwards to `/error` (by default) if `RequestContext.getThrowable()` is not null. You can change the default forwarding path (`/error`) by setting the `error.path` property.

18.16.5 `@EnableZuulProxy` Filters 译：18.16.5 `@EnableZuulProxy`过滤器

创建一个`DiscoveryClientRouteLocator`，用于从`DiscoveryClient`（例如Eureka）以及属性中加载路由定义。 从`DiscoveryClient`为每个`serviceId`创建一条路线。 随着新服务的添加，路由被刷新。

除了前面介绍的过滤器之外，还安装了以下过滤器（如普通的Spring Bean）：

- 预过滤器：
  - `PreDecorationFilter`: Determines where and how to route, depending on the supplied `RouteLocator`. It also sets various proxy-related headers for downstream requests.
- 路线过滤器：
  - `RibbonRoutingFilter`：使用Ribbon，Hystrix和可插入的HTTP客户端发送请求。 服务ID可在`RequestContext`属性中找到，`FilterConstants.SERVICE_ID_KEY`。 此过滤器可以使用不同的HTTP客户端：
    - Apache `HttpClient`: The default client.
    - Squareup `OkHttpClient` v3: Enabled by having the `com.squareup.okhttp3:okhttp` library on the classpath and setting `ribbon.okhttp.enabled=true`.
    - Netflix Ribbon HTTP client: Enabled by setting `ribbon.restclient.enabled=true`. This client has limitations, including that it does not support the PATCH method, but it also has built-in retry.
  - `SimpleHostRoutingFilter`: Sends requests to predetermined URLs through an Apache HttpClient. URLs are found in `RequestContext.getRouteHost()`.

18.16.6 Custom Zuul Filter Examples 译：18.16.6自定义Zuul过滤器示例

Sample Zuul Filters项目包括以下大部分“如何撰写”示例。 还有一些操作该存储库中的请求或响应主体的示例。

本节包含以下示例：

- the section called “How to Write a Pre Filter”
- the section called “How to Write a Route Filter”
- the section called “How to Write a Post Filter”

How to Write a Pre Filter 译：如何编写预过滤器

预过滤器在`RequestContext`设置数据用于下游过滤器。 主要的用例是设置路由过滤器所需的信息。 以下示例显示了Zuul预过滤器：

```
public class QueryParamPreFilter extends ZuulFilter {
 @Override
 public int filterOrder() {
  return PRE_DECORATION_FILTER_ORDER - 1; // run before PreDecoration
 }

 @Override
 public String filterType() {
  return PRE_TYPE;
 }

 @Override
 public boolean shouldFilter() {
  RequestContext ctx = RequestContext.getCurrentContext();
  return !ctx.containsKey(FORWARD_TO_KEY) // a filter has already forwarded
    && !ctx.containsKey(SERVICE_ID_KEY); // a filter has already determined serviceId
 }
    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
  HttpServletRequest request = ctx.getRequest();
  if (request.getParameter("sample") != null) {
      // put the serviceId in `RequestContext`
      ctx.put(SERVICE_ID_KEY, request.getParameter("foo"));
     }
        return null;
    }
}
```

前面的过滤器从`sample`请求参数填充`SERVICE_ID_KEY`。 在实践中，你不应该做那种直接映射。 相反，服务ID应该从`sample`的值中`sample`。

现在`SERVICE_ID_KEY`填充， `PreDecorationFilter`不会运行，并且`RibbonRoutingFilter`运行。

如果您想要路由到完整的网址，请拨打`ctx.setRouteHost(url)`。

要修改路由筛选器转发的路径，请设置`REQUEST_URI_KEY`。

How to Write a Route Filter译:如何编写路由过滤器

路由过滤器在预过滤器之后运行并向其他服务发出请求。 这里的大部分工作是将请求和响应数据转换为客户所需的模型。 以下示例显示了Zuul路由过滤器：

```
public class OkHttpRoutingFilter extends ZuulFilter {
 @Autowired
 private ProxyRequestHelper helper;

 @Override
 public String filterType() {
  return ROUTE_TYPE;
 }

 @Override
 public int filterOrder() {
  return SIMPLE_HOST_ROUTING_FILTER_ORDER - 1;
 }

 @Override
 public boolean shouldFilter() {
  return RequestContext.getCurrentContext().getRouteHost() != null
    && RequestContext.getCurrentContext().sendZuulResponse();
 }

    @Override
    public Object run() {
  OkHttpClient httpClient = new OkHttpClient.Builder()
    // customize
    .build();

  RequestContext context = RequestContext.getCurrentContext();
  HttpServletRequest request = context.getRequest();

  String method = request.getMethod();

  String uri = this.helper.buildZuulRequestURI(request);

  Headers.Builder headers = new Headers.Builder();
  Enumeration<String> headerNames = request.getHeaderNames();
  while (headerNames.hasMoreElements()) {
   String name = headerNames.nextElement();
   Enumeration<String> values = request.getHeaders(name);

   while (values.hasMoreElements()) {
    String value = values.nextElement();
    headers.add(name, value);
   }
  }

  InputStream inputStream = request.getInputStream();

  RequestBody requestBody = null;
  if (inputStream != null && HttpMethod.permitsRequestBody(method)) {
   MediaType mediaType = null;
   if (headers.get("Content-Type") != null) {
    mediaType = MediaType.parse(headers.get("Content-Type"));
   }
   requestBody = RequestBody.create(mediaType, StreamUtils.copyToByteArray(inputStream));
  }

  Request.Builder builder = new Request.Builder()
    .headers(headers.build())
    .url(uri)
    .method(method, requestBody);

  Response response = httpClient.newCall(builder.build()).execute();

  LinkedMultiValueMap<String, String> responseHeaders = new LinkedMultiValueMap<>();

  for (Map.Entry<String, List<String>> entry : response.headers().toMultimap().entrySet()) {
   responseHeaders.put(entry.getKey(), entry.getValue());
  }

  this.helper.setResponse(response.code(), response.body().byteStream(),
    responseHeaders);
  context.setRouteHost(null); // prevent SimpleHostRoutingFilter from running
  return null;
    }
}
```

前面的过滤器将Servlet请求信息转换为OkHttp3请求信息，执行HTTP请求，并将OkHttp3响应信息转换为Servlet响应。

How to Write a Post Filter译:如何写Post过滤器

后过滤器通常会操纵响应。 以下过滤器将随机`UUID`添加为`X-Sample`标头：

```
public class AddResponseHeaderFilter extends ZuulFilter {
 @Override
 public String filterType() {
  return POST_TYPE;
 }

 @Override
 public int filterOrder() {
  return SEND_RESPONSE_FILTER_ORDER - 1;
 }

 @Override
 public boolean shouldFilter() {
  return true;
 }

 @Override
 public Object run() {
  RequestContext context = RequestContext.getCurrentContext();
     HttpServletResponse servletResponse = context.getResponse();
  servletResponse.addHeader("X-Sample", UUID.randomUUID().toString());
  return null;
 }
}
```

其他操作（如转换响应主体）更复杂且计算密集得多。

18.16.7 How Zuul Errors Work译：18.16.7 Zuul错误如何工作

如果在Zuul过滤器生命周期的任何部分发生异常，则会执行错误过滤器。 `SendErrorFilter`仅在`RequestContext.getThrowable()`不是`null`时才运行。 然后它在请求中设置特定的`javax.servlet.error.*`属性，并将请求转发到Spring Boot错误页面。

18.16.8 Zuul Eager Application Context Loading译：18.16.8 Zuul Eager应用程序上下文加载

Zuul内部使用Ribbon来调用远程URL。 默认情况下，Spring客户端在第一次调用时会延迟加载Ribbon客户端。 通过使用以下配置可以为Zuul更改此行为，这会导致在应用程序启动时加载与子功能区相关的应用程序上下文。 以下示例显示如何启用预加载：

application.yml。

```
zuul:
  ribbon:
    eager-load:
      enabled: true
```

19. Polyglot support with Sidecar译：19.与边车的多边形支持

您是否使用了非JVM语言，并希望利用它来利用Eureka，功能区和配置服务器？ Spring Cloud Netflix Sidecar受到Netflix Prana的启发。 它包含一个HTTP API来获取给定服务的所有实例（通过主机和端口）。 您还可以通过嵌入式Zuul代理来代理服务调用，该代理从Eureka获取其路由条目。 Spring Cloud Config Server可以通过主机查找或通过Zuul代理直接访问。 非JVM应用程序应该执行健康检查，以便Sidecar可以向Eureka报告应用程序是启动还是关闭。

要在您的项目中包含Sidecar，请使用群组ID为`org.springframework.cloud`和工件ID或`spring-cloud-netflix-sidecar`。

要启用Sidecar，请使用`@EnableSidecar`创建一个Spring Boot应用程序。 这个注解包括`@EnableCircuitBreaker`， `@EnableDiscoveryClient`，并`@EnableZuulProxy`。 在与非JVM应用程序相同的主机上运行结果应用程序。

要配置侧面车辆，请将`sidecar.port`和`sidecar.health-uri`添加到`application.yml`。 `sidecar.port`属性是非JVM应用程序侦听的端口。 这是Sidecar可以正确地向Eureka注册应用程序。 `sidecar.health-uri`是一个可以在非JVM应用程序上模拟Spring Boot健康指示器的URI。 它应该返回一个类似于以下内容的JSON文档：

健康-URI文档。

```
{
  "status":"UP"
}
```

以下application.yml示例显示了Sidecar应用程序的示例配置：

application.yml。

```
server:
  port: 5678
spring:
  application:
    name: sidecar

sidecar:
  port: 8000
  health-uri: http://localhost:8000/health.json
```

`DiscoveryClient.getInstances()`方法的API是`/hosts/{serviceId}`。 以下示例响应`/hosts/customers`在不同主机上返回两个实例：

/主机/客户。

```
[
    {
        "host": "myhost",
        "port": 9000,
        "uri": "http://myhost:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    },
    {
        "host": "myhost2",
        "port": 9000,
        "uri": "http://myhost2:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    }
]
```

该非API应用程序（如果边车位于端口5678上）可访问此API，其位置为`http://localhost:5678/hosts/{serviceId}`。

Zuul代理自动将Eureka中已知的每项服务的路由添加到`/<serviceId>`，因此客户服务在`/customers`处`/customers`。 非JVM应用程序可以访问`http://localhost:5678/customers`处的客户服务（假设边车正在监听端口5678）。

如果配置服务器向Eureka注册，则非JVM应用程序可以通过Zuul代理访问它。 如果配置`serviceId`为`configserver`且Sidecar在端口5678上，则可以在http://localhost:5678/configserver处访问它。

非JVM应用程序可以利用配置服务器返回YAML文档的能力。 例如，致电http://sidecar.local.spring.io:5678/configserver/default-master.yml可能会导致YAML文档类似于以下内容：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  password: password
info:
  description: Spring Cloud Samples
  url: https://github.com/spring-cloud-samples
```

20. Retrying Failed Requests译：20.重试失败的请求

Spring Cloud Netflix提供了多种创建HTTP请求的方式。 您可以使用负载均衡`RestTemplate`，Ribbon或Feign。 无论您如何选择创建HTTP请求，始终有可能会导致请求失败。 当请求失败时，您可能希望自动重试请求。 要在使用Sping Cloud Netflix时执行此操作，您需要在应用程序的类路径中包含Spring Retry。 当存在Spring重试时，负载均衡`RestTemplates`，Feign和Zuul会自动重试任何失败的请求（假设您的配置允许这样做）。

20.1 BackOff Policies译：20.1 BackOff策略

默认情况下，重试请求时不使用退避策略。 如果您想配置退避策略，则需要创建类型为`LoadBalancedBackOffPolicyFactory`的bean，该bean用于为给定服务创建`BackOffPolicy`，如以下示例所示：

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedBackOffPolicyFactory backOffPolicyFactory() {
        return new LoadBalancedBackOffPolicyFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                return new ExponentialBackOffPolicy();
            }
        };
    }
}
```

20.2 Configuration译：20.2配置

当您通过Spring重试使用功能区时，可以通过配置某些功能区属性来控制重试功能。 要做到这一点，设置`client.ribbon.MaxAutoRetries`，`client.ribbon.MaxAutoRetriesNextServer`，并`client.ribbon.OkToRetryOnAllOperations`性能。 请参阅Ribbon documentation了解这些属性的用途。

启用`client.ribbon.OkToRetryOnAllOperations`包括重试POST请求，这可能会对服务器的资源产生影响，这是由于请求主体的缓冲。

另外，您可能希望在响应中返回某些状态码时重试请求。 您可以通过设置`clientName.ribbon.retryableStatusCodes`属性列出您希望Ribbon客户端重试的响应代码，如以下示例所示：

```
clientName:
  ribbon:
    retryableStatusCodes: 404,502
```

您还可以创建一个类型为`LoadBalancedRetryPolicy`的bean并实现`retryableStatusCode`方法，以根据状态代码重试请求。

20.2.1 Zuul译：20.2.1 Zuul

您可以通过设置关闭Zuulâ€™的重试功能`zuul.retryable`至`false`。 您还可以通过将`zuul.routes.routename.retryable`设置为`zuul.routes.routename.retryable`来`false`路由的重试功能。

21. HTTP Clients译：21. HTTP客户端

Spring Cloud Netflix自动为您创建Ribbon，Feign和Zuul使用的HTTP客户端。 但是，您也可以根据需要提供自定义的HTTP客户端。 要做到这一点，你可以创建类型的豆`ClosableHttpClient`，如果你正在使用的Apache HTTP Cient或`OkHttpClient`如果您使用OK HTTP。

当您创建自己的HTTP客户端时，您还负责为这些客户端实施正确的连接管理策略。 这样做不当可能会导致资源管理问题。

Part IV. Spring Cloud OpenFeign译:第四部分。 Spring Cloud OpenFeign

Finchley.RELEASE

该项目通过自动配置和绑定到Spring环境和其他Spring编程模型成语来为Spring Boot应用程序提供OpenFeign集成。

22. Declarative REST Client: Feign译: 22.声明REST客户：假装

Feign是声明式Web服务客户端。 它使编写Web服务客户端变得更容易。 使用Feign创建一个界面并对其进行注释。 它具有可插入的注释支持，包括Feign注释和JAX-RS注释。 Feign还支持可插拔编码器和解码器。 Spring Cloud添加了对Spring MVC注释的支持，并使用了Spring Web中默认使用的相同`HttpMessageConverters`。 Spring Cloud将Ribbon和Eureka集成在一起，在使用Feign时提供负载均衡的http客户端。

22.1 How to Include Feign译: 22.1如何包含假名

要在项目中包括Feign，请使用组`org.springframework.cloud`和工件编号`spring-cloud-starter-openfeign`的启动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

示例春季启动应用

```
@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

StoreClient.java。

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);
}
```

在`@FeignClient`注释中，字符串值（上面的“存储”）是任意客户端名称，用于创建功能区负载平衡器（请参阅below for details of Ribbon support ）。 您还可以使用`url`属性（绝对值或主机名）指定URL。 应用程序上下文中的bean的名称是接口的完全限定名称。 要指定自己的别名值，可以使用`@FeignClient`注释的`qualifier`值。

上面的功能区客户端需要发现“商店”服务的物理地址。 如果您的应用程序是尤里卡客户端，则它将解析尤里卡服务注册表中的服务。 如果您不想使用Eureka，则只需在外部配置中配置服务器列表即可（请参阅above for example ）。

22.2 Overriding Feign Defaults译: 22.2覆盖假名默认值

Spring Cloud的Feign支持中的一个中心概念是指定的客户端。 每个feign客户端都是组件的一部分，这些组件一起工作以根据需要联系远程服务器，并且该集合有一个名称，您可以使用`@FeignClient`注释作为应用程序开发人员提供。 春云创建一个新的集合为`ApplicationContext`的使用每个命名客户需求`FeignClientsConfiguration`。 这包含（除其他事项外）的`feign.Decoder`，一个`feign.Encoder`和`feign.Contract`。

通过使用`@FeignClient`声明其他配置（在`FeignClientsConfiguration`顶部），Spring Cloud允许您完全控制假客户端。 例：

```
@FeignClient(name = "stores", configuration = FooConfiguration.class)
public interface StoreClient {
    //..
}
```

在这种情况下，客户端从分量已经在由`FeignClientsConfiguration`连同任何在`FooConfiguration`（其中后者将覆盖前者）。

`FooConfiguration`不需要用`@Configuration`注释。 但是，如果是的话，小心从任何排除`@ComponentScan`，否则将包括此配置，它将成为默认源`feign.Decoder`，`feign.Encoder`，`feign.Contract`规定时，等。 这可以通过将它放在与任何`@ComponentScan`或`@SpringBootApplication`分开的，不重叠的包中来`@SpringBootApplication`，或者可以在`@ComponentScan`明确排除。

`serviceId`属性现在已被弃用，以支持`name`属性。

以前，使用`url`属性，不需要`name`属性。 现在需要使用`name`。

`name`和`url`属性支持占位符。

```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface StoreClient {
    //..
}
```

Spring Cloud Netflix默认提供以下bean for `BeanType`（`BeanType` beanName：`ClassName`）：

- `Decoder` feignDecoder: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` feignEncoder: `SpringEncoder`
- `Logger` feignLogger: `Slf4jLogger`
- `Contract` feignContract: `SpringMvcContract`
- `Feign.Builder` feignBuilder: `HystrixFeign.Builder`
- `Client` feignClient: if Ribbon is enabled it is a `LoadBalancerFeignClient`, otherwise the default feign client is used.

OkHttpClient和ApacheHttpClient假客户端可以分别设置`feign.okhttp.enabled`或`feign.httpclient.enabled`到`true`，并将它们放在类路径中使用。 您可以自定义提供的任何一个bean使用的HTTP客户端`ClosableHttpClient`使用Apache或当`OkHttpClient`使用OK HTTP时。

Spring Cloud Netflix默认情况下 *不*提供以下bean，但仍从应用程序上下文中查找这些类型的bean以创建假客户端：

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`
- `SetterFactory`

创建一个这种类型的bean并将其放置在`@FeignClient`配置中（例如上面的`FooConfiguration`），可以覆盖所描述的每个bean。 例：

```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}
```

这取代了`SpringMvcContract`与`feign.Contract.Default`，并增加了一个`RequestInterceptor`，以收集`RequestInterceptor`。

`@FeignClient`也可以使用配置属性进行配置。

application.yml

```
feign:
  client:
    config:
      feignName:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        errorDecoder: com.example.SimpleErrorDecoder
        retryer: com.example.SimpleRetryer
        requestInterceptors:
          - com.example.FooRequestInterceptor
          - com.example.BarRequestInterceptor
        decode404: false
        encoder: com.example.SimpleEncoder
        decoder: com.example.SimpleDecoder
        contract: com.example.SimpleContract
```

默认配置可以按照上述类似的方式在`@EnableFeignClients`属性`defaultConfiguration`中指定。 不同之处在于，此配置将适用于*所有*假客户。

如果您希望使用配置属性配置所有`@FeignClient`，则可以使用`default`名称创建配置属性。

application.yml

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
```

如果我们同时创建了`@Configuration` bean和配置属性，配置属性将会获胜。 它将覆盖`@Configuration`值。 但是，如果要将优先级更改为`@Configuration`，则可以将`feign.client.default-to-properties`更改为`false`。

如果您需要使用`ThreadLocal`绑定变量在`RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE`或假死禁用豪猪。

application.yml

```
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

22.3 Creating Feign Clients Manually译：22.3手动创建Feign客户端

在某些情况下，可能需要以不可能使用上述方法的方式自定义您的Feign客户端。 在这种情况下，您可以使用Feign Builder API创建客户端。 下面是一个例子，它创建两个具有相同接口的Feign客户端，但用一个单独的请求拦截器配置每个客户端。

```
@Import(FeignClientsConfiguration.class)
class FooController {

 private FooClient fooClient;

 private FooClient adminClient;

    @Autowired
 public FooController(Decoder decoder, Encoder encoder, Client client, Contract contract) {
  this.fooClient = Feign.builder().client(client)
    .encoder(encoder)
    .decoder(decoder)
    .contract(contract)
    .requestInterceptor(new BasicAuthRequestInterceptor("user", "user"))
    .target(FooClient.class, "http://PROD-SVC");

  this.adminClient = Feign.builder().client(client)
    .encoder(encoder)
    .decoder(decoder)
    .contract(contract)
    .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
    .target(FooClient.class, "http://PROD-SVC");
    }
}
```

在上例中， `FeignClientsConfiguration.class`是Spring Cloud Netflix提供的默认配置。

`PROD-SVC`是客户将要提供的服务的名称。

Feign `Contract`对象定义了什么注释和值在接口上是有效的。 autowired `Contract` bean为SpringMVC注释提供支持，而不是缺省的Feign本机注释。

22.4 Feign Hystrix Support译：22.4 Feign Hystrix支持

如果Hystrix在类路径上并且`feign.hystrix.enabled=true` ，Feign将用断路器包裹所有方法。 还可以返回`com.netflix.hystrix.HystrixCommand` 。 这可让您使用反应模式（呼叫`.toObservable()`或`.observe()`或异步使用（呼叫`.queue()` ））。

要以每个客户端为基础禁用Hystrix支持，请创建一个包含“原型”范围的vanilla `Feign.Builder` ，例如：

```
@Configuration
public class FooConfiguration {
     @Bean
 @Scope("prototype")
 public Feign.Builder feignBuilder() {
  return Feign.builder();
 }
}
```

在Spring Cloud Dalston发布之前，如果Hystrix在类路径上，Feign默认情况下会将所有方法都封装在断路器中。 Spring Cloud Dalston改变了这种默认行为，以支持选择加入方式。

22.5 Feign Hystrix Fallbacks译：22.5断路器 Hystrix后备

Hystrix支持回退的概念：当电路断开或出现错误时执行的默认代码路径。 要为给定的`@FeignClient`启用回退，请将`fallback`属性设置为实现回退的类名称。 您还需要将您的实现声明为Spring bean。

```
@FeignClient(name = "hello", fallback = HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```

如果一个人需要访问该做回退触发的原因之一，可以使用 `fallbackFactory`属性中 `@FeignClient` 。

```
@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)
protected interface HystrixClient {
 @RequestMapping(method = RequestMethod.GET, value = "/hello")
 Hello iFailSometimes();
}

@Component
static class HystrixClientFallbackFactory implements FallbackFactory<HystrixClient> {
 @Override
 public HystrixClient create(Throwable cause) {
  return new HystrixClient() {
   @Override
   public Hello iFailSometimes() {
    return new Hello("fallback; reason was: " + cause.getMessage());
   }
  };
 }
}
```

在Feign中实施回退以及Hystrix回退工作方式存在限制。 返回`com.netflix.hystrix.HystrixCommand`和`rx.Observable`方法目前不支持回退。

22.6 Feign and `@Primary` 译：22.6 Feign和 `@Primary`

当使用与伴猬回退，也有在多个豆`ApplicationContext`相同类型的。 这会导致`@Autowired`无法工作，因为不存在一个bean，或者标记为主要bean。 为了解决这个问题，Spring Cloud Netflix将所有Feign实例标记为`@Primary`，以便Spring Framework知道要注入哪个bean。 在某些情况下，这可能不合意。 要关闭此行为设置`primary`的属性`@FeignClient`为false。

```
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
 // methods here
}
```

22.7 Feign Inheritance Support 译：22.7 Feign继承支持

Feign通过单继承接口支持样板apis。 这允许将通用操作分组为方便的基础接口。

UserService.java。

```
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value ="/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java。

```
@RestController
public class UserResource implements UserService {

}
```

UserClient.java。

```
package project.user;

@FeignClient("users")
public interface UserClient extends UserService {

}
```

通常不建议在服务器和客户端之间共享接口。 它引入了紧耦合，并且实际上并不以当前的形式使用Spring MVC（方法参数映射不是继承的）。

22.8 Feign request/response compression 译：22.8 Feign请求/响应压缩

您可以考虑为您的Feign请求启用请求或响应GZIP压缩。 您可以通过启用其中一个属性来完成此操作：

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

Feign请求压缩为您提供了类似于您为Web服务器设置的设置：

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

这些属性使您可以选择压缩媒体类型和最小请求阈值长度。

22.9 Feign logging 译：22.9假日记录

为每个创建的Feign客户端创建一个记录器。 默认情况下，记录器的名称是用于创建Feign客户端的接口的完整类名称。 Feign日志记录仅响应`DEBUG`级别。

application.yml。

```
logging.level.project.user.UserClient: DEBUG
```

您可以为每个客户端配置的`Logger.Level`对象告诉Feign需要记录多少。 选择是：

- `NONE`, No logging (**DEFAULT**).
- `BASIC`, Log only the request method and URL and the response status code and execution time.
- `HEADERS`, Log the basic information along with request and response headers.
- `FULL`, Log the headers, body, and metadata for both requests and responses.

例如，以下操作将`Logger.Level`设置为`FULL`：

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

```
        OtherClass.someMethod(myprop.get());
    }
}
stripped). The proxy uses Ribbon to locate an instance to forward to
via discovery, and all requests are executed in a
<<hystrix-fallbacks-for-routes, hystrix command>>, so
failures will show up in Hystrix metrics, and once the circuit is open
the proxy will not try to contact the service.
```

Part V. Spring Cloud Stream 译:第五部分 春云流

23. Quick Start 译:23快速入门

即使在您按照这三步指南跳转到任何细节之前，您也可以在5分钟内尝试Spring Cloud Stream。

我们将向您展示如何创建一个Spring Cloud Stream应用程序，该应用程序接收来自您选择的消息中间件（稍后会详细介绍）的消息并将接收到的消息记录到控制台。 我们称之为`LoggingConsumer`。 虽然不太实用，但它提供了一些主要概念和抽象的良好介绍，使得更容易地消化本用户指南的其余部分。

三个步骤如下：

1. Section 23.1, “Creating a Sample Application by Using Spring Initializr”
2. Section 23.2, “Importing the Project into Your IDE”
3. Section 23.3, “Adding a Message Handler, Building, and Running”

23.1 Creating a Sample Application by Using Spring Initializr 译:23.1使用Spring Initializr创建一个示例应用程序

要开始，请访问Spring Initializr 。 从那里，你可以生成我们的`LoggingConsumer`应用程序。 要做到这一点：

1. In the **Dependencies** section, start typing `stream`. When the “Cloud Stream” option should appears, select it.
2. Start typing either 'kafka' or 'rabbit'.
3. 选择“Kafka”或“RabbitMQ”。
   基本上，你选择你的应用程序绑定的消息中间件。 我们建议使用您已经安装的或者感觉安装和运行更加舒适的。 另外，正如您从初始屏幕中看到的那样，您可以选择其他几个选项。 例如，您可以选择Gradle作为构建工具而不是Maven（默认）。
4. 在**Artifact**字段中，输入'logging-consumer'。
   Artifact字段的值将成为应用程序名称。 如果你选择RabbitMQ作为中间件，你的Spring Initializr现在应该如下所示：
5. 点击 **Generate Project**按钮。
   这样做会将生成的项目的压缩版本下载到您的硬盘。
6. Unzip the file into the folder you want to use as your project directory.

我们鼓励您探索Spring Initializr提供的许多可能性。 它可以让你创建许多不同类型的Spring应用程序。

23.2 Importing the Project into Your IDE 译:23.2将项目导入IDE

现在您可以将该项目导入到IDE中。 请记住，根据IDE，您可能需要遵循特定的导入程序。 例如，根据项目的生成方式（Maven或Gradle），您可能需要遵循特定的导入过程（例如，在Eclipse或STS中，您需要使用File→Import→Maven→Existing Maven项目）。

一旦导入，项目必须没有任何错误。 另外，`src/main/java`应该包含`com.example.loggingconsumer.LoggingConsumerApplication`。

从技术上讲，在这一点上，你可以运行应用程序的主类。 它已经是一个有效的Spring Boot应用程序。 但是，它什么都不做，所以我们想添加一些代码。

23.3 Adding a Message Handler, Building, and Running 译:23.3添加消息处理程序，构建和运行

修改`com.example.loggingconsumer.LoggingConsumerApplication`类看起来如下所示：

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class LoggingConsumerApplication {

 public static void main(String[] args) {
  SpringApplication.run(LoggingConsumerApplication.class, args);
 }

 @StreamListener(Sink.INPUT)
 public void handle(Person person) {
  System.out.println("Received: " + person);
 }

 public static class Person {
  private String name;
  public String getName() {
   return name;
  }
  public void setName(String name) {
   this.name = name;
  }
  public String toString() {
   return this.name;
  }
 }
}
```

正如你从前面的清单中看到的那样：

- We have enabled `Sink` binding (input-no-output) by using `@EnableBinding(Sink.class)`. Doing so signals to the framework to initiate binding to the messaging middleware, where it automatically creates the destination (that is, queue, topic, and others) that are bound to the `Sink.INPUT` channel.
- We have added a `handler` method to receive incoming messages of type `Person`. Doing so lets you see one of the core features of the framework: It tries to automatically convert incoming message payloads to type `Person`.

您现在拥有一个功能齐全的Spring Cloud Stream应用程序，它可以侦听消息。 从这里，为了简单起见，我们假定您选择了step one中的 RabbitMQ。 假设你已经安装并运行了RabbitMQ，你可以在你的IDE中运行它的`main`方法来启动应用程序。

您应该看到以下输出：

```
--- [ main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound: input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg, bound to: input
--- [ main] o.s.a.r.c.CachingConnectionFactory       : Attempting to connect to: [localhost:5672]
--- [ main] o.s.a.r.c.CachingConnectionFactory       : Created new connection: rabbitConnectionFactory#2a3a299:0/SimpleConnection@66c83fc8. . .
. . .
--- [ main] o.s.i.a.i.AmqpInboundChannelAdapter      : started inbound.input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg
. . .
--- [ main] c.e.l.LoggingConsumerApplication         : Started LoggingConsumerApplication in 2.531 seconds (JVM running for 2.897)
```

转到RabbitMQ管理控制台或任何其他RabbitMQ客户端并发送消息到`input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg`。 `anonymous.CbMIwdkJSBO1ZoPDOtHtCg`部分代表组名并生成，因此它在您的环境中必然会有所不同。 对于更可预测的事情，您可以通过设置`spring.cloud.stream.bindings.input.group=hello`（或任何您喜欢的名称）来使用明确的组名称。

消息的内容应该是`Person`类的JSON表示，如下所示：

```
{"name":"Sam Spade"}
```

然后，在您的控制台中，您应该看到：

```
Received: Sam Spade
```

您还可以构建应用程序并将其打包到引导jar（通过使用`./mvnw clean install`），并使用`java -jar`命令运行构建的JAR。

现在你有一个工作（尽管非常基本）的Spring Cloud Stream应用程序。

24. What's New in 2.0? 译：24什么是2.0的新功能？

Spring Cloud Stream引入了许多新功能，增强功能和更改。 以下几节概述了最值得注意的一些：

- Section 24.1, "New Features and Components"
- Section 24.2, "Notable Enhancements"

24.1 New Features and Components 译：24.1新功能和组件

- **Polling Consumers**: Introduction of polled consumers, which lets the application control message processing rates. See "Section 27.3.4, "Using Polled Consumers"" for more details. You can also read this blog post for more details.
- **Micrometer Support**: Metrics has been switched to use Micrometer. `MeterRegistry` is also provided as a bean so that custom applications can autowire it to capture custom metrics. See "Chapter 35, *Metrics Emitter*" for more details.
- **New Actuator Binding Controls**: New actuator binding controls let you both visualize and control the Bindings lifecycle. For more details, see Section 28.6, "Binding visualization and control".
- **Configurable RetryTemplate**: Aside from providing properties to configure `RetryTemplate`, we now let you provide your own template, effectively overriding the one provided by the framework. To use it, configure it as a `@Bean` in your application.

24.2 Notable Enhancements 译：24.2显著的增强功能

该版本包括以下显着增强功能：

- Section 24.2.1, "Both Actuator and Web Dependencies Are Now Optional"
- Section 24.2.2, "Content-type Negotiation Improvements"
- Section 24.3, "Notable Deprecations"

24.2.1 Both Actuator and Web Dependencies Are Now Optional 译：24.2.1执行器和Web依赖关系现在都是可选的

如果不需要执行程序和Web依赖关系，此更改将减少部署的应用程序的占用空间。 它还允许您通过手动添加以下某个依赖关系在反应式和常规Web范例之间切换。

以下清单显示了如何添加传统的Web框架：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

以下清单显示了如何添加反应性Web框架：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

以下列表显示了如何添加执行器相关性：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

24.2.2 Content-type Negotiation Improvements译：24.2.2内容类型协商改进

Verion 2.0的核心主题之一是围绕内容类型协商和消息转换进行改进（兼顾一致性和性能）。 以下摘要概述了该领域的显着变化和改进。 有关更多详细信息，请参阅“Chapter 30, *Content Type Negotiation*”部分。 另外this blog post包含更多的细节。

- All message conversion is now handled **only** by `MessageConverter` objects.
- We introduced the `@StreamMessageConverter` annotation to provide custom `MessageConverter` objects.
- We introduced the default `Content Type` as `application/json`, which needs to be taken into consideration when migrating 1.3 application or operating in the mixed mode (that is, 1.3 producer → 2.0 consumer).
- Messages with textual payloads and a `contentType` of `text/…` or `… /json` are no longer converted to `Message<String>` for cases where the argument type of the provided `MessageHandler` can not be determined (that is, `public void handle(Message<?> message)` or `public void handle(Object payload)`). Furthermore, a strong argument type may not be enough to properly convert messages, so the `contentType` header may be used as a supplement by some `MessageConverters`.

24.3 Notable Deprecations译：24.3值得注意的弃用

从版本2.0开始，以下项目已被弃用：

- Section 24.3.1, “Java Serialization (Java Native and Kryo)”
- Section 24.3.2, “Deprecated Classes and Methods”

24.3.1 Java Serialization (Java Native and Kryo)译：24.3.1 Java序列化（Java Native和Kryo）

`JavaSerializationMessageConverter`和`KryoMessageConverter`现在仍然存在。 但是，我们计划在未来将它们从核心软件包和支持中移出。 此弃用的主要原因是标记基于类型的，特定于语言的序列化可能导致分布式环境中的问题，其中生产者和使用者可能依赖于不同的JVM版本或具有不同版本的支持库（即Kryo）。 我们还想提请注意消费者和生产者甚至可能不是基于Java的事实，因此polyglot样式序列化（即JSON）更适合。

24.3.2 Deprecated Classes and Methods译：24.3.2不推荐的类和方法

以下是对重要弃用的简要总结。 有关更多详细信息，请参阅相应的{spring-cloud-stream-javadoc-current} [javadoc]。

- `SharedChannelRegistry`. Use `SharedBindingTargetRegistry`.
- `Bindings`. Beans qualified by it are already uniquely identified by their type — for example, provided `Source`, `Processor`, or custom bindings:

```
public interface Sample {
 String OUTPUT = "sampleOutput";

 @Output(Sample.OUTPUT)
 MessageChannel output();
}
```

- `HeaderMode.raw`. Use `none`, `headers` or `embeddedHeaders`
- `ProducerProperties.partitionKeyExtractorClass` in favor of `partitionKeyExtractorName` and `ProducerProperties.partitionSelectorClass` in favor of `partitionSelectorName`. This change ensures that both components are Spring configured and managed and are referenced in a Spring-friendly way.
- `BinderAwareRouterBeanPostProcessor`. While the component remains, it is no longer a `BeanPostProcessor` and will be renamed in the future.
- `BinderProperties.setEnvironment(Properties environment)`. Use `BinderProperties.setEnvironment(Map<String, Object> environment)`.

本节将更详细地介绍如何使用Spring Cloud Stream。 它涵盖了创建和运行流应用程序等主题。

25. Introducing Spring Cloud Stream译：25.介绍Spring Cloud Stream

Spring Cloud Stream是构建消息驱动的微服务应用程序的框架。 Spring Cloud Stream基于Spring Boot构建独立的生产级Spring应用程序，并使用Spring Integration为消息代理提供连接。 它提供了来自多个供应商的中间件的自定义配置，介绍了持久发布 - 订阅语义，消费者组和分区的概念。

您可以将`@EnableBinding`注释添加到您的应用程序中，以立即连接到消息代理，并且您可以将`@StreamListener`添加到某个方法以使其接收用于流处理的事件。 以下示例显示了接收外部消息的接收器应用程序：

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

  public static void main(String[] args) {
    SpringApplication.run(VoteRecordingSinkApplication.class, args);
  }

  @StreamListener(Sink.INPUT)
  public void processVote(Vote vote) {
      votingService.recordVote(vote);
  }
}
```

`@EnableBinding`注释将一个或多个接口用作参数（在这种情况下，参数是单个`Sink`接口）。 一个接口声明输入和输出通道。 春季云流提供`Source`， `Sink`，并`Processor`接口。 你也可以定义你自己的接口。

以下清单显示了`Sink`接口的定义：

```
public interface Sink {
  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

`@Input`注释标识一个输入通道，通过该通道接收的消息进入应用程序。 `@Output`注释标识输出通道，发布的消息通过该通道离开应用程序。 `@Input`和`@Output`注释可以将通道名称作为参数。 如果没有提供名称，则使用注释方法的名称。

Spring Cloud Stream为您创建界面的实现。 您可以通过自动装配在应用程序中使用它，如以下示例中所示（从测试用例中）：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)
@WebAppConfiguration
@DirtiesContext
public class StreamApplicationTests {

  @Autowired
  private Sink sink;

  @Test
  public void contextLoads() {
    assertNotNull(this.sink.input());
  }
}
```

26. Main Concepts译：26.主要概念

Spring Cloud Stream提供了许多抽象和原语，可以简化消息驱动的微服务应用程序的编写。 本节概述以下内容：

- Spring Cloud Stream’s application model
- Section 26.2, “The Binder Abstraction”
- Persistent publish-subscribe support
- Consumer group support
- Partitioning support
- A pluggable Binder SPI

26.1 Application Model译：26.1应用程序模型

Spring Cloud Stream应用程序由一个中间件中立核心组成。 该应用程序通过Spring Cloud Stream注入的输入和输出通道与外部世界进行通信。 通道通过中间件特定的Binder实现连接到外部代理。

图26.1。 Spring云流应用程序

26.1.1 Fat JAR译：26.1.1胖 JAR

Spring Cloud Stream应用程序可以在IDE中以独立模式运行以进行测试。 要在生产环境中运行Spring Cloud Stream应用程序，可以使用为Maven或Gradle提供的标准Spring Boot工具来创建可执行文件（或“fat”）JAR。 有关更多详细信息，请参阅Spring Boot Reference Guide 。

26.2 The Binder Abstraction译：26.2粘合剂抽象

Spring Cloud Stream为Kafka和Rabbit MQ提供了Binder实现。 Spring Cloud Stream还包含一个TestSupportBinder，它使未修改的频道保持不变，以便测试可以直接与频道进行交互，并可靠地断言所接收的内容。 您也可以使用可扩展API编写您自己的Binder。

Spring Cloud Stream使用Spring Boot进行配置，而Binder抽象使Spring Cloud Stream应用程序可以灵活地连接到中间件。 例如，部署者可以在运行时动态选择通道连接的目标（例如Kafka主题或RabbitMQ交换）。 可以通过外部配置属性和Spring Boot支持的任何形式（包括应用程序参数，环境变量和`application.yml`或`application.properties`文件）来提供此类配置。 在Chapter 25, *Introducing Spring Cloud Stream*部分的接收器示例中，将`spring.cloud.stream.bindings.input.destination`应用程序属性设置为`raw-sensor-data`会导致它从`raw-sensor-data` Kafka主题或绑定到`raw-sensor-data` RabbitMQ交换的队列`raw-sensor-data` 。

Spring Cloud Stream会自动检测并使用类路径中找到的联编程序。 您可以使用具有相同代码的不同类型的中间件。 为此，请在构建时包含不同的绑定程序。 对于更复杂的用例，您还可以将多个活页夹与应用程序打包在一起，并让它在运行时选择活页夹（甚至是否针对不同通道使用不同的活页夹）。

26.3 Persistent Publish-Subscribe Support译：26.3持久发布 - 订阅支持

应用程序之间的通信遵循发布 - 订阅模式，数据通过共享主题进行广播。 这可以在下图中看到，它显示了一组相互作用的Spring Cloud Stream应用程序的典型部署。

图26.2。 Spring云流发布 - 订阅

传感器向HTTP端点报告的数据被发送到名为`raw-sensor-data`的公共目的地。 从目的地，它是由微服务应用程序独立处理的，该应用程序计算时间窗平均值，并由另一个将原始数据摄入HDFS（Hadoop分布式文件系统）的微服务应用程序进行处理。 为了处理数据，这两个应用程序在运行时声明主题作为它们的输入。

发布 - 订阅通信模型降低了生产者和消费者的复杂性，并允许将新的应用程序添加到拓扑中而不会中断现有的流程。 例如，在平均计算应用程序的下游，您可以添加计算显示和监视的最高温度值的应用程序。 然后，您可以添加另一个应用程序来解释相同的平均流量以进行故障检测。 通过共享主题而不是点对点队列进行所有通信可以减少微服务之间的耦合。

尽管发布 - 订阅消息传递的概念并不新鲜，但Spring Cloud Stream采取了额外的步骤，使其成为其应用程序模型的斟酌选择。 通过使用本机中间件支持，Spring Cloud Stream还简化了跨不同平台的发布 - 订阅模型的使用。

26.4 Consumer Groups译：26.4消费群体

尽管发布 - 订阅模型可以通过共享主题轻松连接应用程序，但通过创建给定应用程序的多个实例来扩展的能力同样重要。 当这样做时，应用程序的不同实例被置于竞争的消费者关系中，其中只有一个实例需要处理给定的消息。

Spring Cloud Stream通过消费者组的概念来模拟此行为。 （Spring Cloud Stream消费者群体与Kafka消费者群体相似并受其启发）。每个消费者绑定可以使用`spring.cloud.stream.bindings.<channelName>.group`属性来指定群组名称。 对于下图中显示的消费者，此属性将设置为`spring.cloud.stream.bindings.<channelName>.group=hdfsWrite`或`spring.cloud.stream.bindings.<channelName>.group=average`。

图26.3。 春季云流消费群体

订阅给定目标的所有组都会收到已发布数据的副本，但每个组中只有一个成员收到来自该目标的给定消息。 默认情况下，当未指定组时，Spring Cloud Stream会将该应用程序分配给与所有其他使用者组处于发布 - 订阅关系的匿名且独立的单一成员使用者组。

26.5 Consumer Types译：26.5消费者类型

支持两种类型的消费者：

- Message-driven (sometimes referred to as Asynchronous)
- Polled (sometimes referred to as Synchronous)

在版本2.0之前，只支持异步使用者。 只要消息可用并且有线程可用来处理消息，就会立即发送消息。

如果您希望控制处理邮件的速率，则可能需要使用同步使用者。

26.5.1 Durability译：26.5.1耐久性

与Spring Cloud Stream的自以为是的应用程序模型一致，消费者组订阅是持久的。 也就是说，binder工具确保组预订是持久的，并且一旦创建了组的至少一个订阅，即使在组中的所有应用程序都停止时发送它们，组也会收到消息。

匿名订阅本质上不具有持久性。 对于某些binder实现（例如RabbitMQ），可能会有非持久组预订。

通常，将应用程序绑定到给定目标时总是指定一个使用者组是最好的。 在扩展Spring Cloud Stream应用程序时，您必须为每个输入绑定指定一个使用者组。 这样做可以防止应用程序的实例接收到重复的消息（除非需要这种行为，这是不常见的）。

26.6 Partitioning Support译：26.6分区支持

Spring Cloud Stream支持在给定应用程序的多个实例之间对数据进行分区。 在分区方案中，物理通信介质（如代理主题）被视为构建为多个分区。 一个或多个生产者应用程序实例将数据发送到多个消费者应用程序实例，并确保由共同特征标识的数据由同一个消费者实例处理。

Spring Cloud Stream为以统一的方式实现分区处理用例提供了一个通用抽象。 无论代理本身是否自然分区（例如，Kafka）或不是（例如RabbitMQ），都可以使用分区。

图26.4。 Spring云流分区

分区是有状态处理中的关键概念，对于确保所有相关数据一起处理而言，分区至关重要（无论是出于性能还是一致性原因）。 例如，在时间窗平均计算示例中，来自任何给定传感器的所有测量结果均由同一应用程序实例处理非常重要。

要设置分区处理方案，您必须配置数据生成和数据消耗两端。

27. Programming Model译：27.编程模型

要理解编程模型，您应该熟悉以下核心概念：

- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Destination Bindings:** Bridge between the external messaging systems and application provided *Producers* and *Consumers* of messages (created by the Destination Binders).
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

27.1 Destination Binders译：27.1目标绑定器

目标绑定器是Spring Cloud Stream的扩展组件，负责提供必要的配置和实现以促进与外部邮件系统的集成。 这种集成负责连接，委托以及向生产者和消费者发送消息的路由，数据类型转换，用户代码的调用等。

粘合剂处理许多锅炉板的责任，否则将落在你的肩上。 但是，要实现这一点，绑定器仍然需要一些帮助，以简单但需要的用户指令集的形式出现，这些指令通常以某种类型的配置形式出现。

尽管讨论所有可用的活页夹和绑定配置选项（本手册的其余部分广泛介绍了这些选项），但本节不在讨论范围内，但*目标绑定*确实需要特别注意。 下一节将详细讨论它。

27.2 Destination Bindings译：27.2目的地绑定

如前所述， *目的地绑定*为外部消息系统和应用程序提供的 *生产者*和 *消费者*之间提供了桥梁。

将@EnableBinding注释应用于其中一个应用程序的配置类定义了目标绑定。 `@EnableBinding`注释本身带有元注释`@Configuration`并触发Spring Cloud Stream基础结构的配置。

以下示例显示完全配置并运行的Spring Cloud Stream应用程序，该应用程序以 `String`类型（参见 Chapter 30, *Content Type Negotiation*部分）从 `INPUT`目标接收消息的有效内容，将其记录到控制台并将其转换为 `OUTPUT`目标，然后将其转换为大写。

```
@SpringBootApplication
@EnableBinding(Processor.class)
public class MyApplication {

 public static void main(String[] args) {
  SpringApplication.run(MyApplication.class, args);
 }

 @StreamListener(Processor.INPUT)
 @SendTo(Processor.OUTPUT)
 public String handle(String value) {
  System.out.println("Received: " + value);
  return value.toUpperCase();
 }
}
```

正如您所看到的， `@EnableBinding`注释可以将一个或多个接口类作为参数。 这些参数被称为*绑定*，并且它们包含表示 *可绑定组件的*方法。 这些组件通常是基于通道的活页夹 （例如Rabbit，Kafka和其他）的消息通道（请参阅Spring Messaging ）。 然而其他类型的绑定可以提供对相应技术的本地特征的支持。 例如Kafka Streams binder（以前称为KStream）允许直接绑定到Kafka Streams（请参阅Kafka Streams了解更多详细信息）。

Spring Cloud Stream已经为典型的消息交换契约提供了 *绑定*接口，其中包括：

- **Sink:** Identifies the contract for the message consumer by providing the destination from which the message is consumed.
- **Source:** Identifies the contract for the message producer by providing the destination to which the produced message is sent.
- **Processor:** Encapsulates both the sink and the source contracts by exposing two destinations that allow consumption and production of messages.

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();
}
```

```
public interface Processor extends Source, Sink {}
```

虽然前面的例子满足大多数情况下，你也可以定义自己的绑定接口定义自己的合同，并使用 `@Input`个 `@Output`注释，以确定实际 *可绑定组件* 。

例如：

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

使用前面的例子中示出的接口，以一个参数 `@EnableBinding`触发命名三个入境信道的创建 `orders`， `hotDrinks`，和 `coldDrinks`分别。

您可以根据需要提供尽可能多的绑定接口，作为 `@EnableBinding`批注的参数，如以下示例所示：

```
@EnableBinding(value={Orders.class, Payment.class}
```

在Spring Cloud Stream中，可绑定的 `MessageChannel`组件是Spring Messaging `MessageChannel` （用于出站）及其扩展 `SubscribableChannel` （用于入站）。

可Pollable目的地绑定

尽管先前描述的绑定支持基于事件的消息消费，但有时您需要更多的控制，例如消费率。

从版本2.0开始，您现在可以绑定可轮询的使用者：

以下示例显示如何绑定可轮询的使用者：

```
public interface PolledBarista {

    @Input
    PollableMessageSource orders();
 . . .
}
```

在这种情况下， `PollableMessageSource`的实现绑定到`orders` channel"。 有关更多详细信息，请参阅Section 27.3.4, "Using Polled Consumers" 。

自定义频道名称

通过使用 `@Input`和 `@Output`注释，您可以为通道指定自定义通道名称，如以下示例所示：

```
public interface Barista {
    @Input("inboundOrders")
    SubscribableChannel orders();
}
```

在上例中，创建的绑定通道名为 `inboundOrders` 。

通常，您不需要直接访问单个通道或绑定（ `@EnableBinding`通过`@EnableBinding`注释来配置它们）。 然而，当你这样做的时候，可能会有时间，例如测试或其他角落案例。

除了为每个绑定生成通道并将它们注册为Spring bean之外，对于每个绑定接口，Spring Cloud Stream都会生成一个实现该接口的bean。 这意味着您可以在应用程序中通过自动布线访问表示绑定或单个通道的接口，如以下两个示例所示：

Autowire绑定界面

```
@Autowire
private Source source

public void sayHello(String name) {
    source.output().send(MessageBuilder.withPayload(name).build());
}
```

Autowire个人频道

```
@Autowire
private MessageChannel output;

public void sayHello(String name) {
    output.send(MessageBuilder.withPayload(name).build());
}
```

您也可以使用标准Spring的 `@Qualifier`批注来查看频道名称是自定义的情况，还是需要特定命名频道的多频道场景。

以下示例显示如何以这种方式使用@Qualifier注释：

```
@Autowire
@Qualifier("myChannel")
private MessageChannel output;
```

27.3 Producing and Consuming Messages译：27.3生产和消费信息

您可以使用Spring Integration注释或Spring Cloud Stream本机注释编写Spring Cloud Stream应用程序。

27.3.1 Spring Integration Support译：27.3.1 Spring集成支持

春季云流是建立在定义的概念和模式 Enterprise Integration Patterns ，并依赖于它的内部实现对项目的春季投资组合中已经建立的和流行的实现企业集成模式的： Spring Integration框架。

所以它唯一的支持Spring集成已经建立的基础，语义和配置选项的技巧

例如，您可以将 `Source`的输出通道附加到 `MessageSource`并使用熟悉的 `@InboundChannelAdapter`批注，如下所示：

```
@EnableBinding(Source.class)
public class TimerSource {

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10", maxMessagesPerPoll = "1"))
  public MessageSource<String> timerMessageSource() {
    return () -> new GenericMessage<>("Hello Spring Cloud Stream");
  }
}
```

同样，您可以使用@Transformer或@ServiceActivator，同时为 *处理器*绑定合同提供消息处理程序方法的实现，如以下示例所示：

```
@EnableBinding(Processor.class)
public class TransformProcessor {
  @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
  public Object transform(String message) {
    return message.toUpperCase();
  }
}
```

虽然这可能会略微提前，但重要的是要明白，使用`@StreamListener`注释从相同绑定中使用时，将使用pub-sub模型。 用`@StreamListener`注解的每种方法`@StreamListener`收到其自己的消息副本，并且每个方法都有其自己的消费者组。 不过，如果你使用Spring Integration的注释（如一个来自相同的结合消耗`@Aggregator` ， `@Transformer` ，或`@ServiceActivator` ），这些消耗于竞争模型。 没有为每个订阅创建个人消费者组。

27.3.2 Using @StreamListener Annotation译：27.3.2使用 @StreamListener注释

为了补充其Spring Integration的支持下，春天的云流提供了自己`@StreamListener`注释，其他春天消息的注解（仿照`@MessageMapping`， `@JmsListener`，`@RabbitListener`，和其他人），并提供conviniences，如基于内容的路由等。

```
@EnableBinding(Sink.class)
public class VoteHandler {

  @Autowired
  VotingService votingService;

  @StreamListener(Sink.INPUT)
  public void handle(Vote vote) {
    votingService.record(vote);
  }
}
```

与其他春天消息的方法，方法的参数可以与注释`@Payload`， `@Headers`，并`@Header`。

对于返回数据的方法，必须使用`@SendTo`批注指定方法返回的数据的输出绑定目标，如以下示例所示：

```
@EnableBinding(Processor.class)
public class TransformProcessor {

  @Autowired
  VotingService votingService;

  @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
  public VoteResult handle(Vote vote) {
    return votingService.record(vote);
  }
}
```

27.3.3 Using @StreamListener for Content-based routing译：27.3.3使用@StreamListener进行基于内容的路由

Spring Cloud Stream支持根据条件将消息分派到多个处理程序方法，并使用`@StreamListener`注释。

为了有资格支持有条件的调度，一种方法必须满足以下条件：

- It must not return a value.
- It must be an individual message handling method (reactive API methods are not supported).

条件是由Spel位表达在指定`condition`注释的参数，并针对每个消息进行评估。 所有匹配条件的处理程序都在同一个线程中调用，并且不必假定调用的顺序。

在具有调度条件的`@StreamListener`的以下示例中，所有带有值为`bogey`的标头`type`的消息都将调度到`receiveBogey`方法，并且所有带有值为`bacall`的标头`type`的消息都将调度到`receiveBacall`方法。

```
@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class TestPojoWithAnnotatedArguments {

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bogey'")
    public void receiveBogey(@Payload BogeyPojo bogeyPojo) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bacall'")
    public void receiveBacall(@Payload BacallPojo bacallPojo) {
       // handle the message
    }
}
```

内容类型协商在`condition`的上下文中

了解`condition`参数`@StreamListener`的基于内容的路由背后的一些机制非常重要，特别是在整个消息类型的上下文中。 如果您在继续之前熟悉Chapter 30, *Content Type Negotiation*，也可能有所帮助。

考虑以下情况：

```
@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class CatsAndDogs {

    @StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Dog'")
    public void bark(Dog dog) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Cat'")
    public void purr(Cat cat) {
       // handle the message
    }
}
```

前面的代码是完全有效的。 它编译和部署没有任何问题，但它从来没有产生你期望的结果。

那是因为你正在测试一些你所期望的状态还不存在的东西。 这是因为消息的有效负载尚未从有线格式（`byte[]`）转换为所需的类型。 换句话说，它还没有经过Chapter 30, *Content Type Negotiation*中描述的类型转换过程。

因此，除非您使用评估原始数据的SPeL表达式（例如，字节数组中的第一个字节的值），请使用基于消息头的表达式（例如`condition = "headers['type']=='dog'"`）。

目前，通过`@StreamListener`条件进行调度仅支持基于通道的`@StreamListener`（不适用于响应式编程）支持。

27.3.4 Using Polled Consumers 译：27.3.4使用轮询的消费者

在使用民意调查的消费者时，您`PollableMessageSource`根据需要调查`PollableMessageSource`。 考虑以下被调查消费者的例子：

```
public interface PolledConsumer {

    @Input
    PollableMessageSource destIn();

    @Output
    MessageChannel destOut();

}
```

在前面的例子中给出了被轮询的消费者，你可以按如下方式使用它：

```
@Bean
public ApplicationRunner poller(PollableMessageSource destIn, MessageChannel destOut) {
    return args -> {
        while (someCondition()) {
            try {
                if (!destIn.poll(m -> {
                    String newPayload = ((String) m.getPayload()).toUpperCase();
                    destOut.send(new GenericMessage<>(newPayload));
                })) {
                    Thread.sleep(1000);
                }
            }
            catch (Exception e) {
                // handle failure (throw an exception to reject the message);
            }
        }
    };
}
```

`PollableMessageSource.poll()`方法需要`MessageHandler`参数（通常为lambda表达式，如此处所示）。 如果收到消息并成功处理，则返回`true`。

与消息驱动的消费者一样，如果`MessageHandler`引发异常，则消息将发布到错误通道，如“???”中所述。

通常，`poll()`方法在`MessageHandler`退出时确认该消息。 如果该方法异常退出，则该消息被拒绝（不重新排队）。 您可以通过承担确认责任来覆盖该行为，如以下示例所示：

```
@Bean
public ApplicationRunner poller(PollableMessageSource dest1In, MessageChannel dest2Out) {
    return args -> {
        while (someCondition()) {
            if (!dest1In.poll(m -> {
                StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).noAutoAck();
                // e.g. hand off to another thread which can perform the ack
                // or acknowledge(Status.REQUEUE)

            })) {
                Thread.sleep(1000);
            }
        }
    };
}
```

Important

您必须在某个时间点`ack`（或`nack`）该消息，以避免资源泄漏。

Important

一些消息系统（如Apache Kafka）在日志中保持一个简单的偏移量。 如果递送失败并且被重新排队`StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUEUE);`，则任何稍后成功的确认消息都会重新递送。

还有一个重载的`poll`方法，其定义如下：

```
poll(MessageHandler handler, ParameterizedTypeReference<?> type)
```

`type`是一个转换提示，允许传入的消息负载被转换，如以下示例所示：

```
boolean result = pollableSource.poll(received -> {
   Map<String, Foo> payload = (Map<String, Foo>) received.getPayload();
            ...

  }, new ParameterizedTypeReference<Map<String, Foo>>() {});
```

27.4 Error Handling 译：27.4错误处理

错误发生了，Spring Cloud Stream提供了几种灵活的机制来处理它们。 错误处理有两种形式：

- **application:** The error handling is done within the application (custom error handler).
- **system:** The error handling is delegated to the binder (re-queue, DL, and others). Note that the techniques are dependent on binder implementation and the capability of the underlying messaging middleware.

Spring Cloud Stream使用Spring Retry库来促进成功的消息处理。 有关更多详细信息，请参阅Section 27.4.3, “Retry Template”。 但是，如果全部失败，则由消息处理程序抛出的异常会传播回活页夹。 此时，活页夹调用自定义错误处理程序或将错误传回消息传递系统（重新排队，DLQ等）。

27.4.1 Application Error Handling译:27.4.1应用程序错误处理

有两种类型的应用程序级错误处理。 可以在每个绑定订阅处理错误，或者全局处理程序可以处理所有绑定订阅错误。 让我们回顾一下细节。

图27.1。 Spring云流应用程序与自定义和全局错误处理程序

对于每个输入绑定，Spring Cloud Stream会创建一个专用的错误通道，其语义为`<destinationName>.errors`。

`<destinationName>`由绑定的名称（如`input`）和组的名称（如`myGroup`）组成。

考虑以下：

```
spring.cloud.stream.bindings.input.group=myGroup
```

```
@StreamListener(Sink.INPUT) // destination name 'input.myGroup'
public void handle(Person value)
 throw new RuntimeException("BOOM!");
}

@ServiceActivator(inputChannel = Processor.INPUT + ".myGroup.errors") //channel name 'input.myGroup.errors'
public void error(Message<?> message) {
 System.out.println("Handling ERROR: " + message);
}
```

在前面的示例中，目标名称是`input.myGroup`，专用错误通道名称是`input.myGroup.errors`。

使用@StreamListener注解专门用于定义桥接内部通道和外部目标的绑定。 鉴于目标特定错误通道没有关联的外部目标，此类通道是Spring Integration（SI）的特权。 这意味着必须使用SI处理程序注释之一（即@ServiceActivator，@Transformer等）来定义此目标的处理程序。

如果没有指定`group`则使用匿名组（类似于`input.anonymous.2K37rb06Q6m2r51-SPIDDQ`），这不适合处理错误的错误，因为在创建目标之前您不知道它将会是什么。

另外，如果您不想访问现有目标，例如：

```
spring.cloud.stream.bindings.input.destination=myFooDestination
spring.cloud.stream.bindings.input.group=myGroup
```

完整的目标名称是`myFooDestination.myGroup`，然后专用错误通道名称是`myFooDestination.myGroup.errors`。

回到这个例子。。。

订购名为`input`的通道的`handle(..)`方法引发异常。 鉴于还有一个订户到错误通道`input.myGroup.errors`所有错误消息都由该订户处理。

如果你有多个绑定，你可能想要一个错误处理程序。 Spring Cloud Stream通过将每个*错误通道*桥接到名为`errorChannel`通道来自动为*全局错误通道*提供支持，从而允许单个订户处理所有错误，如以下示例所示：

```
@StreamListener("errorChannel")
public void error(Message<?> message) {
 System.out.println("Handling ERROR: " + message);
}
```

如果错误处理逻辑相同，无论哪个处理程序产生错误，这可能是一个方便的选项。

此外，通过为出站目标配置名为`error`的绑定，可将发送到`errorChannel`错误消息发布到代理的特定目标。 此选项提供了一种机制，可以将错误消息自动发送到绑定到该目标的另一个应用程序，或者用于以后检索（例如，审计）。 例如，要将错误消息发布到名为`myErrors`的代理目标，请设置以下属性：

```
spring.cloud.stream.bindings.error.destination=myErrors.
```

将全局错误通道桥接到代理目标的能力本质上提供了一种将 *应用程序级*错误处理与 *系统级*错误处理连接起来的机制。

27.4.2 System Error Handling译:27.4.2系统错误处理

系统级错误处理意味着将错误传回消息传递系统，并且由于并非每个消息传递系统都是相同的，所以这些功能可能会有所不同，从活页夹到活页夹。

也就是说，在本节中，我们将解释系统级错误处理背后的一般概念，并以兔子绑定器为例。 注：Kafka活页夹提供类似的支持，但某些配置属性确实不同。 另外，有关更多详细信息和配置选项，请参阅个别联编程序的文档。

如果未配置内部错误处理程序，则错误会传播到活页夹，然后活页夹会将这些错误传播回消息传递系统。 根据消息系统的功能，系统可能会*丢弃*消息，对消息*重新排队*以便重新处理或*将失败的消息发送给DLQ*。 Rabbit和Kafka都支持这些概念。 但是，其他活页夹可能不适用，因此请参阅您的单个活页夹的文档以获取有关支持的系统级错误处理选项的详细信息。

Drop Failed Messages译:投递失败的邮件

默认情况下，如果未提供其他系统级配置，则消息传递系统会丢弃失败的消息。 尽管在某些情况下可以接受，但在大多数情况下，这不是，我们需要一些恢复机制来避免信息丢失。

DLQ - Dead Letter Queue译:DLQ-死信队列

DLQ允许将失败的消息发送到特定目的地： - *死信队列*。

配置后，失败的消息将发送到此目的地，以便后续重新处理或审核和核对。

例如，继续前面的示例并使用Rabbit绑定器设置DLQ，您需要设置以下属性：

```
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
```

请记住，在上述属性中，`input`对应于输入目标绑定的名称。`consumer`表示它是消费者属性，`auto-bind-dlq`指示活页夹为`input`目标配置DLQ，这会导致另一个名为`input.myGroup.dlq` Rabbit队列。

配置完成后，所有失败的消息都将被路由到此队列，并显示与以下类似的错误消息：

```
delivery_mode: 1
headers:
x-death:
count: 1
reason: rejected
queue: input.hello
time: 1522328151
exchange:
routing-keys: input.myGroup
Payload {"name”:"Bob"}
```

正如你从上面看到的那样，你的原始信息被保留下来以便采取进一步行动。

但是，您可能已经注意到的一件事是，有关消息处理的原始问题的信息有限。 例如，您看不到与原始错误相对应的堆栈跟踪。 要获得有关原始错误的更多相关信息，您必须设置一个附加属性：

```
spring.cloud.stream.rabbit.bindings.input.consumer.republish-to-dlq=true
```

这样做会强制内部错误处理程序截获错误消息并在将其发布到DLQ之前向其添加其他信息。 配置完成后，您可以看到错误消息包含更多与原始错误相关的信息，如下所示：

```
delivery_mode: 2
headers:
x-original-exchange:
x-exception-message: has an error
x-original-routingKey: input.myGroup
x-exception-stacktrace: org.springframework.messaging.MessageHandlingException: nested exception is
      org.springframework.messaging.MessagingException: has an error, failedMessage=GenericMessage [payload=byte[15],
      headers={amqp_receivedDeliveryMode=NON_PERSISTENT, amqp_receivedRoutingKey=input.hello, amqp_deliveryTag=1,
      deliveryAttempt=3, amqp_consumerQueue=input.hello, amqp_redelivered=false, id=a15231e6-3f80-677b-5ad7-d4b1e61e486e,
      amqp_consumerTag=amq.ctag-skBFapilvtZhDsn0k3ZmQg, contentType=application/json, timestamp=1522327846136}]
      at org.spring...integ...han...MethodInvokingMessageProcessor.processMessage(MethodInvokingMessageProcessor.java:107)
      at. . . . .
Payload {"name”:"Bob"}
```

这有效地结合了应用程序级别和系统级别的错误处理，以进一步协助下游故障排除机制。

Re-queue Failed Messages译:重新排队失败的消息

如前所述，目前支持的绑定器（Rabbit和Kafka）依靠`RetryTemplate`来促进成功的消息处理。 详情请参阅Section 27.4.3, “Retry Template” 。 但是，对于`max-attempts`属性设置为1的情况，消息的内部重新处理被禁用。 此时，您可以通过指示消息系统重新排列失败的消息来促进消息重新处理（重试）。 一旦重新排队，失败的消息被发送回原来的处理程序，实质上创建一个重试循环。

如果错误的性质与一些资源的零星但短期的不可用性相关，那么这种选择可能是可行的。

要做到这一点，您必须设置以下属性：

```
spring.cloud.stream.bindings.input.consumer.max-attempts=1
spring.cloud.stream.rabbit.bindings.input.consumer.requeue-rejected=true
```

在前面的示例中，`max-attempts`设置为1，实质上禁用内部重`requeue-rejected`，`requeue-rejected`（重复*拒绝消息的*`true`）设置为`true`。 一旦设置，失败的消息被重新提交给相同的处理程序并持续循环，或者直到处理程序抛出`AmqpRejectAndDontRequeueException`本质上允许您在处理程序本身内构建自己的重试逻辑。

27.4.3 Retry Template译: 27.4.3重试模板

`RetryTemplate`是Spring Retry库的一部分。 虽然出了这dcument的范围，以涵盖所有的功能`RetryTemplate`，我们会提到，具体涉及以下消费性`RetryTemplate`：

maxAttempts

尝试处理邮件的次数。

默认值：3。

backOffInitialInterval

重试时的退避初始间隔。

默认1000毫秒。

backOffMaxInterval

最大回退间隔。

默认10000毫秒。

backOffMultiplier

退避乘数。

默认2.0。

虽然前面的设置对于大部分自定义要求都是足够的，但它们可能不满足某些复杂要求，您可能希望提供自己的实例`RetryTemplate`。 为此，请在应用程序配置`@Bean`其配置为`@Bean`。 应用程序提供的实例覆盖了框架提供的实例。

27.5 Reactive Programming Support译: 27.5反应式编程支持

Spring Cloud Stream还支持使用反应式API，将传入和传出数据作为连续数据流处理。 通过`spring-cloud-stream-reactive`支持反应式API，需要将其明确添加到项目

中。

具有反应式API的编程模型是声明式的。 您可以使用描述从入站到出站数据流的功能转换的运算符，而不是指定应如何处理每个单独的消息。

目前Spring Cloud Stream支持唯一的Reactor API 。 将来，我们打算支持基于反应流的更通用的模型。

反应式编程模型还使用`@StreamListener`注释来设置反应式处理程序。 不同之处在于：

- The `@StreamListener` annotation must not specify an input or output, as they are provided as arguments and return values from the method.
- The arguments of the method must be annotated with `@Input` and `@Output`, indicating which input or output the incoming and outgoing data flows connect to, respectively.
- The return value of the method, if any, is annotated with `@Output`, indicating the input where data should be sent.

反应式编程支持需要Java 1.8。

从Spring Cloud Stream 1.1.1及更高版本（从发行版Brooklyn.SR2开始），反应式编程支持需要使用Reactor 3.0.4.RELEASE及更高版本。 早期版本的Reactor（包括3.0.1.RELEASE，3.0.2.RELEASE和3.0.3.RELEASE）不受支持。 `spring-cloud-stream-reactive`传递地检索正确的版本，但项目结构有可能将`io.projectreactor:reactor-core`的版本管理为早期版本，特别是在使用Maven时。 对于使用Spring Initializr与Spring Boot 1.x生成的项目，这种情况会覆盖Reactor版本`2.0.8.RELEASE` 。 在这种情况下，您必须确保已发布工件的正确版本。 您可以通过将`io.projectreactor:reactor-core`与版本`3.0.4.RELEASE`或更高版本直接依赖关系添加到您的项目中。

当前使用的术语“反应性”是指被使用的反应性API，而不是反应性的执行模型（即，绑定端点仍然使用“推”而不是“拉”模型）。 尽管通过使用反应堆提供了一些背压支持，但我们打算在未来的版本中通过使用本地反应客户端为连接的中间件来支持完全反应性的管道。

27.5.1 Reactor-based Handlers译：27.5.1基于反应器的处理程序

基于Reactor的处理程序可以有以下参数类型：

- For arguments annotated with `@Input`, it supports the Reactor `Flux` type. The parameterization of the inbound Flux follows the same rules as in the case of individual message handling: It can be the entire `Message`, a POJO that can be the `Message` payload, or a POJO that is the result of a transformation based on the `Message` content-type header. Multiple inputs are provided.
- For arguments annotated with `Output`, it supports the `FluxSender` type, which connects a `Flux` produced by the method with an output. Generally speaking, specifying outputs as arguments is only recommended when the method can have multiple outputs.

基于Reactor的处理程序支持返回类型`Flux` 。 在这种情况下，它必须用`@Output`注释。 当单个输出`Flux`可用时，我们建议使用该方法的返回值。

以下示例显示了基于Reactor的 `Processor` ：

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Flux<String> receive(@Input(Processor.INPUT) Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```

使用输出参数的相同处理器如下例所示：

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  public void receive(@Input(Processor.INPUT) Flux<String> input,
     @Output(Processor.OUTPUT) FluxSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```

27.5.2 Reactive Sources译：27.5.2反应源

Spring Cloud Stream反应性支持还提供了通过`@StreamEmitter`注释创建反应性源的`@StreamEmitter` 。 通过使用`@StreamEmitter`注释，常规来源可能会转换为反应性来源。 `@StreamEmitter`是一种方法级别注释， `@StreamEmitter`方法标记为`@EnableBinding`声明的输出的发射器。 您不能将`@Input`注释与`@StreamEmitter`一起使用，因为使用此注释标记的方法不会侦听任何输入。 相反，标有`@StreamEmitter`方法`@StreamEmitter`生成输出。 遵循`@StreamListener`使用的相同编程模型， `@StreamEmitter`还允许灵活地使用`@Output`注释，具体取决于该方法是否有任何参数，返回类型和其他注意事项。

本节的其余部分包含以各种样式使用 `@StreamEmitter`注释的示例。

以下示例每毫秒发出 `Hello, World`消息并发布到Reactor `Flux` ：

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  public Flux<String> emit() {
    return Flux.intervalMillis(1)
            .map(l -> "Hello World");
  }
}
```

在前面的示例中，所得到的消息 `Flux`被发送到的输出信道 `Source` 。

下一个例子是`@StreamEmmitter`另一种味道，它发送一个反应堆`Flux` 。 以下方法使用`FluxSender`以编程`Flux`从源发送`Flux` ，而不是返回`Flux` ：

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  public void emit(FluxSender output) {
    output.send(Flux.intervalMillis(1)
            .map(l -> "Hello World"));
  }
}
```

下一个示例与上面的功能和样式片段完全相同。 但是，不是在方法上使用明确的`@Output`注释，而是在方法参数上使用注释。

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  public void emit(@Output(Source.OUTPUT) FluxSender output) {
    output.send(Flux.intervalMillis(1)
            .map(l -> "Hello World"));
  }
}
```

本节的最后一个例子是使用Reactive Streams Publisher API编写反应源的另一种风格，并利用Spring Integration Java DSL中的支持。 以下示例中的`Publisher`仍在使用Reactor `Flux`，但从应用程序角度来看，`Flux`用户是透明的，只需要Reactive Streams和Java DSL用于Spring集成：

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  @Bean
  public Publisher<Message<String>> emit() {
    return IntegrationFlows.from(() ->
                new GenericMessage<>("Hello World"),
        e -> e.poller(p -> p.fixedDelay(1)))
        .toReactivePublisher();
  }
}
```

28. Binders译：28.粘合剂

Spring Cloud Stream提供了一个Binder抽象，用于连接外部中间件的物理目标。 本节提供有关Binder SPI主要概念，主要组件和特定实现细节的信息。

28.1 Producers and Consumers译：28.1生产者和消费者

下图显示了生产者和消费者的一般关系：

图28.1。 生产者和消费者

制作人是将消息发送到频道的任何组件。 该通道可以绑定到外部消息代理，该代理具有`Binder`实现。 在调用`bindProducer()`方法时，第一个参数是代理中目标的名称，第二个参数是生产者发送消息的本地通道实例，第三个参数包含要生成的属性（如分区键表达式）在为该通道创建的适配器内使用。

消费者是接收来自频道的消息的任何组件。 与生产者一样，消费者的渠道可以绑定到外部消息代理。 调用`bindConsumer()`方法时，第一个参数是目标名称，第二个参数提供使用者逻辑组的名称。 由给定目标的消费者绑定表示的每个组都会收到生产者发送到该目标的每条消息的副本（即遵循正常的发布 - 订阅语义）。 如果有多个使用相同组名称绑定的使用者实例，则消息在这些使用者实例之间进行负载均衡，以便每个组中的消费者实例仅使用由生产者发送的每条消息（即，它遵循正常排队语义）。

28.2 Binder SPI译：28.2粘合剂SPI

Binder SPI包含许多接口，即用型实用程序类以及提供用于连接外部中间件的可插入机制的发现策略。

SPI的关键点是`Binder`接口，这是一种将输入和输出连接到外部中间件的策略。 以下清单显示了`Binder`接口的定义：

```
public interface Binder<T, C extends ConsumerProperties, P extends ProducerProperties> {
    Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

    Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);
}
```

界面被参数化，提供了许多扩展点：

- Input and output bind targets. As of version 1.0, only `MessageChannel` is supported, but this is intended to be used as an extension point in the future.
- Extended consumer and producer properties, allowing specific Binder implementations to add supplemental properties that can be supported in a type-safe manner.

一个典型的活页夹实现包括以下内容：

- A class that implements the `Binder` interface;
- A Spring `@Configuration` class that creates a bean of type `Binder` along with the middleware connection infrastructure.
- 在包含一个或多个联编程序定义的类路径中找到的`META-INF/spring.binders`文件，如下例所示：

```
kafka:\
org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```

28.3 Binder Detection译：28.3粘合剂检测

Spring Cloud Stream依赖于Binder SPI的实现来执行将通道连接到消息代理的任务。 每个Binder实现通常连接到一种类型的消息传递系统。

28.3.1 Classpath Detection译：28.3.1类路径检测

默认情况下，Spring Cloud Stream依靠Spring Boot的自动配置来配置绑定过程。 如果在类路径中找到单个Binder实现，Spring Cloud Stream会自动使用它。 例如，旨在仅绑定到RabbitMQ的Spring Cloud Stream项目可以添加以下依赖项：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

有关其他绑定器依赖项的特定Maven坐标，请参阅该绑定器实现的文档。

28.4 Multiple Binders on the Classpath译：28.4类路径中的多个绑定器

当类路径中存在多个联编程序时，应用程序必须指明哪个联编程序将用于每个通道绑定。 每个绑定器配置都包含一个`META-INF/spring.binders`文件，该文件是一个简单的属性文件，如以下示例所示：

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

其他提供的绑定器实现（例如Kafka）也存在类似的文件，并且自定义绑定器实现也有望提供它们。 键表示binder实现的标识名称，而值是逗号分隔的配置类列表，每个配置类都包含一个且只有一个类型为`org.springframework.cloud.stream.binder.Binder` bean定义。

通过在每个通道绑定上配置活页夹，可以使用`spring.cloud.stream.defaultBinder`属性（例如， `spring.cloud.stream.defaultBinder=rabbit` ）或单独执行活页夹选择。 例如，处理器应用程序（分别具有用于读取和写入的通道名称分别为`input`和`output` ）从Kafka读取并写入RabbitMQ的处理器应用程序可以指定以下配置：

```
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

28.5 Connecting to Multiple Systems译：28.5连接到多个系统

默认情况下，绑定器共享应用程序的Spring Boot自动配置，以便创建在类路径中找到的每个绑定器的一个实例。 如果您的应用程序应连接到同一类型的多个代理程序，则可以指定多个资料夹配置，每个配置程序具有不同的环境设置。

打开显式文件夹配置将完全禁用默认文件夹配置过程。 如果这样做，则所有正在使用的粘合剂必须包含在配置中。 打算透明地使用Spring Cloud Stream的框架可能会创建可以按名称引用的绑定器配置，但它们不会影响默认绑定器配置。 为此，绑定器配置可能会将其`defaultCandidate`标志设置为false（例如， `spring.cloud.stream.binders.<configurationName>.defaultCandidate=false` ）。 这表示独立于默认绑定器配置过程而存在的配置。

以下示例显示连接到两个RabbitMQ代理实例的处理器应用程序的典型配置：

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: thing1
          binder: rabbit1
        output:
          destination: thing2
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```

28.6 Binding visualization and control译：28.6绑定可视化和控制

自2.0版以来，Spring Cloud Stream支持通过执行器端点对绑定进行可视化和控制。

从版本2.0开始，执行器和Web是可选的，您必须首先添加一个Web依赖项，并手动添加执行器依赖项。 以下示例显示如何添加Web框架的依赖关系：

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

以下示例显示如何添加WebFlux框架的依赖关系：

```
<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

您可以按如下方式添加执行器依赖项：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

要在Cloud Foundry中运行Spring Cloud Stream 2.0应用程序，您必须将`spring-boot-starter-web`和`spring-boot-starter-actuator`添加到类路径中。 否则，由于运行状况检查失败，应用程序将无法启动。

您还必须通过设置以下属性来启用`bindings`执行器端点： `--management.endpoints.web.exposure.include=bindings` 。

一旦这些先决条件得到满足。 应用程序启动时，您应该在日志中看到以下内容：

```
: Mapped "{[/actuator/bindings/{name}],methods=[POST]. . .
: Mapped "{[/actuator/bindings],methods=[GET]. . .
: Mapped "{[/actuator/bindings/{name}],methods=[GET]. . .
```

要可视化当前绑定，请访问以下URL： `http://<host>:<port>/actuator/bindings`

或者，要查看单个绑定，请访问与以下内容类似的一个网址： `http://<host>:<port>/actuator/bindings/myBindingName`

您还可以通过发布到相同的URL来停止，启动，暂停和恢复单个绑定，同时提供`state`参数作为JSON，如以下示例所示：

curl -d'{"state"：“STOPPED”}'-H“Content-Type：application / json”-X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d'{"state"：“ STARTED ”}'-H“Content-Type：应用程序/ json“-X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d'{"state“：”PAUSED“}'-H”Content-Type：application / json“-X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d'{"state“：”RESUMED“} '-H“Content-Type：application / json”-X POST http://<host>:<port>/actuator/bindings/myBindingName

`PAUSED`和`RESUMED`仅在相应的绑定器及其基础技术支持它`RESUMED`起作用。 否则，您会在日志中看到警告消息。 目前，只有Kafka活页夹支持`PAUSED`和`RESUMED`状态。

28.7 Binder Configuration Properties译：28.7活页夹配置属性

定制资料夹配置时，以下属性可用。 这些属性通过`org.springframework.cloud.stream.config.BinderProperties`

它们必须以`spring.cloud.stream.binders.<configurationName>`为前缀。

type

活页夹类型。 它通常引用类路径中找到的绑定器之一 - 特别是`META-INF/spring.binders`文件中的一个键。

默认情况下，它具有与配置名称相同的值。

inheritEnvironment

配置是否继承应用程序本身的环境。

默认： `true` 。

environment

Root用于可用于自定义活页夹环境的一组属性。 当设置此属性时，创建联编程序的上下文不是应用程序上下文的子代。 该设置允许活页夹组件和应用程序组件之间完全分离。

默认： `empty` 。

defaultCandidate

绑定器配置是否被视为默认绑定的候选者，或者仅在明确引用时才能使用。 此设置允许添加活页夹配置而不干扰默认处理。

默认： `true` 。

29. Configuration Options译：29配置选项

Spring Cloud Stream支持常规配置选项以及绑定和绑定器的配置。 一些绑定器允许额外的绑定属性支持中间件特定的功能。

配置选项可以通过Spring Boot支持的任何机制提供给Spring Cloud Stream应用程序。 这包括应用程序参数，环境变量以及YAML或.properties文件。

29.1 Binding Service Properties译：29.1绑定服务属性

这些属性通过`org.springframework.cloud.stream.config.BindingServiceProperties`

spring.cloud.stream.instanceCount

应用程序的已部署实例的数量。 必须设置为生产者端的分区。 必须在使用RabbitMQ时与消费者一起设置，如果使用Kafka，则`autoRebalanceEnabled=false`在`autoRebalanceEnabled=false` 。

默认： `1` 。

spring.cloud.stream.instanceIndex

The instance index of the application: A number from `0` to `instanceCount - 1`. Used for partitioning with RabbitMQ and with Kafka if `autoRebalanceEnabled=false`. Automatically set in Cloud Foundry to match the application's instance index.

spring.cloud.stream.dynamicDestinations

可以动态绑定的目标列表（例如，在动态路由方案中）。 如果设置，则只能列出列出的目的地。

默认值：空（让任何目的地被绑定）。

spring.cloud.stream.defaultBinder

如果配置了多个绑定器，则使用默认绑定器。 见Multiple Binders on the Classpath 。

默认值：空。

spring.cloud.stream.overrideCloudConnectors

此属性仅适用于`cloud`配置文件处于活动状态且随应用程序提供了Spring Cloud连接器的情况。 如果该属性为`false`（默认值），则绑定程序会检测合适的绑定服务（例如RabbitMQ绑定在Cloud Foundry中的RabbitMQ服务），并将其用于创建连接（通常通过Spring Cloud连接器）。 当设置为`true`，此属性指示绑定器完全忽略绑定的服务并依赖Spring Boot属性（例如，依赖RabbitMQ绑定器环境中提供的`spring.rabbitmq.*`属性）。 此属性的典型用法是嵌套在自定义环境when connecting to multiple systems中 。

默认： `false` 。

spring.cloud.stream.bindingRetryInterval

例如，当活页夹不支持后期绑定以及代理（例如，Apache Kafka）关闭时，重试绑定创建之间的时间间隔（以秒为单位）。 将其设置为零可将致命的条件视为无效，从而阻止应用程序启动。

默认： `30`

29.2 Binding Properties译：29.2绑定属性

绑定属性通过使用格式`spring.cloud.stream.bindings.<channelName>.<property>=<value>` 。 所述`<channelName>`表示所配置的信道的名称（例如，`output`为`Source` ）。

为避免重复，Spring Cloud Stream支持所有通道的设置值，格式为 `spring.cloud.stream.default.<property>=<value>` 。

在下文中，我们指出我们已经省略了前缀 `spring.cloud.stream.bindings.<channelName>.`并只关注属性名称，并且理解前缀ise在运行时包含。

29.2.1 Common Binding Properties译：29.2.1通用绑定属性

这些属性通过 `org.springframework.cloud.stream.config.BindingProperties`

以下绑定属性可用于输入和输出绑定，并且必须以 `spring.cloud.stream.bindings.<channelName>.`为前缀（例如，`spring.cloud.stream.bindings.input.destination=ticktock` ）。

可以使用 `spring.cloud.stream.default`前缀（例如`spring.cloud.stream.default.contentType = application / json`）来设置默认值。

destination

The target destination of a channel on the bound middleware (for example, the RabbitMQ exchange or Kafka topic). If the channel is bound as a consumer, it could be bound to multiple destinations, and the destination names can be specified as comma-separated `String` values. If not set, the channel name is used instead. The default value of this property cannot be overridden.

group

频道的消费者群体。 仅适用于入站绑定。 见Consumer Groups 。

默认： `null` （表示匿名消费者）。

contentType

频道的内容类型。 请参阅“ Chapter 30, *Content Type Negotiation*” 。

默认值： `null` （不执行类型强制）。

binder

该绑定使用的联编程序。 有关详细信息，请参阅“ Section 28.4, “Multiple Binders on the Classpath”” 。

默认值： `null` （如果存在，则使用默认联编程序）。

29.2.2 Consumer Properties译：29.2.2消费者属性

这些属性通过 `org.springframework.cloud.stream.binder.ConsumerProperties`

以下绑定属性仅可用于输入绑定，并且必须以 `spring.cloud.stream.bindings.<channelName>.consumer.` （例如 `spring.cloud.stream.bindings.input.consumer.concurrency=3` ）作为前缀。

可以使用 `spring.cloud.stream.default.consumer`前缀（例如， `spring.cloud.stream.default.consumer.headerMode=none` ）来设置默认值。

concurrency

入境消费者的并发性。

默认： `1` 。

partitioned

消费者是否接收来自分区生产者的数据。

默认： `false` 。

headerMode

设置为`none` ，禁用输入上的标题解析。 仅对本机不支持消息标头且需要标头嵌入的消息传递中间件有效。 当不支持本机标头时，从非Spring Cloud Stream应用程序使用数据时，此选项非常有用。 设置为`headers` ，它使用中间件的本机标头机制。 当设置为`embeddedHeaders` ，它将头部嵌入到消息有效载荷中。

默认值：取决于活页夹的实施。

maxAttempts

如果处理失败，则处理消息的尝试次数（包括第一次）。 设置为`1`禁用重试。

默认： `3` 。

backOffInitialInterval

重试时的退避初始间隔。

默认： `1000` 。

backOffMaxInterval

最大回退间隔。

默认： `10000` 。

backOffMultiplier

退避乘数。

默认： `2.0` 。

instanceIndex

设置为大于等于零的值时，它允许自定义此使用者的实例索引（如果不同于`spring.cloud.stream.instanceIndex`）。 设置为负值时，默认为`spring.cloud.stream.instanceIndex`。 请参阅“ Section 32.2, “Instance Index and Instance Count”” 以了解更多信息。

默认： `-1` 。

instanceCount

当设置为大于等于零的值时，它允许自定义此使用者的实例数（如果与`spring.cloud.stream.instanceCount`不同）。 设置为负值时，默认为`spring.cloud.stream.instanceCount`。 有关更多信息，请参阅“ Section 32.2, “Instance Index and Instance Count”” 。

默认： `-1` 。

29.2.3 Producer Properties译：29.2.3生产者属性

这些属性通过`org.springframework.cloud.stream.binder.ProducerProperties`

以下绑定属性仅适用于输出绑定，并且必须以`spring.cloud.stream.bindings.<channelName>.producer.`（例如`spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id`）作为前缀。

可以使用前缀`spring.cloud.stream.default.producer`（例如，`spring.cloud.stream.default.producer.partitionKeyExpression=payload.id`）来设置默认值。

partitionKeyExpression

SpEL表达式，用于确定如何对出站数据进行分区。 如果已设置，或者设置了`partitionKeyExtractorClass`则将此通道上的出站数据分区。 `partitionCount`必须设置为大于1的值才能生效。 与`partitionKeyExtractorClass`。 请参阅“ Section 26.6, “Partitioning Support”” 。

默认值：null。

partitionKeyExtractorClass

一个`PartitionKeyExtractorStrategy`实现。 如果已设置，或者设置了`partitionKeyExpression`则会对此通道上的出站数据进行分区。 `partitionCount`必须设置为大于1的值才能生效。 与`partitionKeyExpression`。 请参阅“ Section 26.6, “Partitioning Support”” 。

默认： `null` 。

partitionSelectorClass

一个`PartitionSelectorStrategy`实现。 与`partitionSelectorExpression`。 如果两者均未设置，则将该分区选择为`hashCode(key) % partitionCount`，其中`key`通过`partitionKeyExpression`或`partitionKeyExtractorClass`计算`partitionKeyExtractorClass`。

默认： `null` 。

partitionSelectorExpression

用于定制分区选择的SpEL表达式。 与`partitionSelectorClass`。 如果两者均未设置，则将该分区选择为`hashCode(key) % partitionCount`，其中`key`通过`partitionKeyExpression`或`partitionKeyExtractorClass`计算`partitionKeyExtractorClass`。

默认： `null` 。

partitionCount

数据的目标分区数（如果启用分区）。 如果生产者被分区，则必须设置为大于1的值。 在卡夫卡，它被解释为一种暗示。 取而代之的是使用较大的值和目标主题的分区数。

默认： `1` 。

requiredGroups

A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (for example, by pre-creating durable queues in RabbitMQ).

headerMode

设置为`none`，将禁用输出中的标题嵌入。 它仅适用于本地不支持消息标头且需要标头嵌入的消息传递中间件。 在不支持本机标头时为非Spring Cloud Stream应用程序生成数据时，此选项非常有用。 设置为`headers`，它使用中间件的本机标头机制。 当设置为`embeddedHeaders`，它将标头嵌入到消息有效载荷中。

默认值：取决于资料夹的实施。

useNativeEncoding

设置为`true`，出站消息将直接由客户端库序列化，客户端库必须进行相应配置（例如，设置适当的Kafka生产者值序列化程序）。 使用此配置时，出站邮件编组不是基于绑定的`contentType`。 当使用本地编码时，消费者有责任使用合适的解码器（例如，Kafka消费者价值反序列化器）来反序列化入站消息。 此外，使用本地编码和解码时， `headerMode=embeddedHeaders`属性将被忽略，并且标头不会嵌入到消息中。

默认： `false` 。

errorChannelEnabled

当设置为`true`，如果资料夹支持异步发送结果，则将发送失败发送到目标的错误通道。 有关更多信息，请参阅“ ???” 。

默认： `false` 。

29.3 Using Dynamically Bound Destinations译：29.3使用动态绑定的目标

除了使用`@EnableBinding`定义的通道`@EnableBinding`，Spring Cloud Stream还允许应用程序将消息发送到动态绑定目标。 例如，当目标目标需要在运行时确定时，这很有用。 应用程序可以通过使用由`@EnableBinding`注释自动注册的`BinderAwareChannelResolver` bean来`@EnableBinding`。

'spring.cloud.stream.dynamicDestinations'属性可用于将动态目标名称限制为已知集（白名单）。 如果未设置此属性，则可以动态绑定任何目标。

可以直接使用`BinderAwareChannelResolver`，如以下使用路径变量决定目标通道的REST控制器示例所示：

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;

    @RequestMapping(path = "/{target}", method = POST, consumes = "*/*")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @PathVariable("target") target,
           @RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
        sendMessage(body, target, contentType);
    }

    private void sendMessage(String body, String target, Object contentType) {
        resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE, contentType))));
    }
}
```

现在考虑当我们在默认端口（8080）上启动应用程序时发生的情况，并使用CURL提出以下请求：

```
curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders
```

目的地'客户'和'订单'是在经纪人（用于交换兔子或在用于卡夫卡的主题中）中创建的，其名称为“客户”和“订单”，并且数据被发布到适当的目的地。

`BinderAwareChannelResolver`是一个通用的Spring集成`DestinationResolver`，可以注入其他组件，例如，在使用基于传入JSON消息的`target`字段的SpEL表达式的路由器中。 以下示例包含一个读取SpEL表达式的路由器：

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;


    @RequestMapping(path = "/", method = POST, consumes = "application/json")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
        sendMessage(body, contentType);
    }

    private void sendMessage(Object body, Object contentType) {
        routerChannel().send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE, contentType))));
    }

    @Bean(name = "routerChannel")
    public MessageChannel routerChannel() {
        return new DirectChannel();
    }

    @Bean
    @ServiceActivator(inputChannel = "routerChannel")
    public ExpressionEvaluatingRouter router() {
        ExpressionEvaluatingRouter router =
            new ExpressionEvaluatingRouter(new SpelExpressionParser().parseExpression("payload.target"));
        router.setDefaultOutputChannelName("default-output");
        router.setChannelResolver(resolver);
        return router;
    }
}
```

Router Sink Application使用此技术按需创建目的地。

如果频道名称事先已知，则可以像任何其他目的地一样配置制作者属性。 或者，如果您注册了`NewBindingCallback<>`bean，则在创建绑定之前调用它。 该回调采用绑定器使用的扩展生产者属性的泛型类型。 它有一个方法：

```
void configure(String channelName, MessageChannel channel, ProducerProperties producerProperties,
        T extendedProducerProperties);
```

以下示例显示如何使用RabbitMQ联编程序：

```
@Bean
public NewBindingCallback<RabbitProducerProperties> dynamicConfigurer() {
    return (name, channel, props, extended) -> {
        props.setRequiredGroups("bindThisQueue");
        extended.setQueueNameGroupOnly(true);
        extended.setAutoBindDlq(true);
        extended.setDeadLetterQueueName("myDLQ");
    };
}
```

如果需要使用多种联编程序类型支持动态目标，请使用`Object`作为通用类型，并根据需要`extended`参数。

30. Content Type Negotiation译：30.内容类型协商

数据转换是任何消息驱动的微服务架构的核心功能之一。 鉴于此，在Spring Cloud Stream中，此类数据表示为Spring `Message` ，因此在到达目的地之前可能必须将消息转换为所需的形状或大小。 这是必需的，原因有两个：

1. To convert the contents of the incoming message to match the signature of the application-provided handler.
2. To convert the contents of the outgoing message to the wire format.

有线格式通常为 `byte[]` （对于Kafka和Rabbit活页夹而言是这样的），但是它受到活页夹实施的控制。

在Spring Cloud Stream中，消息转换是通过 `org.springframework.messaging.converter.MessageConverter` 完成的。

作为要遵循的细节的补充，您可能还需要阅读以下 blog post 。

30.1 Mechanics译：30.1力学

为了更好地理解内容类型协商的机制和必要性，我们通过使用以下消息处理程序作为示例来看看一个非常简单的用例：

```
@StreamListener(Processor.INPUT)
@SendTo(Processor.OUTPUT)
public String handle(Person person) {..}
```

为了简单起见，我们假设这是应用程序中唯一的处理程序（我们假设没有内部管道）。

上例中显示的处理程序需要一个 `Person` 对象作为参数，并生成一个 `String` 类型作为输出。 为了使框架成功地将传入的 `Message` 作为参数传递给此处理函数，它必须以某种方式将 `Message` 类型的有效载荷从有线格式转换为 `Person` 类型。 换句话说，框架必须定位并应用适当的 `MessageConverter` 。 为了实现这个目标，框架需要用户的一些指导。 其中一条指令已由处理程序方法本身的签名提供（ `Person` 类型）。 因此，理论上，这应该是（并且在某些情况下）是足够的。 但是，对于大多数用例，为了选择合适的 `MessageConverter` ，框架需要额外的信息。 缺失的部分是 `contentType` 。

Spring Cloud Stream提供了三种机制来定义 `contentType` （按优先顺序排列）：

1. **HEADER**: The `contentType` can be communicated through the Message itself. By providing a `contentType` header, you declare the content type to use to locate and apply the appropriate `MessageConverter`.
2. **绑定** ： `contentType` 可每个目的地通过设置绑定设置 `spring.cloud.stream.bindings.input.content-type` 属性。

   属性名称中的 `input` 段与目标的实际名称（在我们的例子中为“输入”）相对应。 通过此方法，您可以在每个绑定的基础上声明用于查找和应用相应 `MessageConverter` 的内容类型。

3. **DEFAULT**: If `contentType` is not present in the `Message` header or the binding, the default `application/json` content type is used to locate and apply the appropriate `MessageConverter`.

如前所述，前面的列表还显示了并列情况下的优先顺序。 例如，标头提供的内容类型优先于任何其他内容类型。 这同样适用于基于每个绑定设置的内容类型，这基本上允许您覆盖默认内容类型。 但是，它也提供了一个合理的默认值（这是从社区反馈中确定的）。

使 `application/json` 成为分布式微服务体系结构驱动的互操作性需求的默认原因的另一个原因是，生产者和消费者不仅运行在不同的JVM中，而且还可以在不同的非JVM平台上运行。

当非空处理程序方法返回时，如果返回值已经是 `Message` ，则 `Message` 将成为有效负载。 但是，如果返回值不是 `Message` ，则新 `Message` 的返回值将作为有效负载构建，同时继承来自输入 `Message` 的标头减去由 `SpringIntegrationProperties.messageHandlerNotPropagatedHeaders` 定义或过滤的标头。 默认情况下，只有一个标题集： `contentType` 。 这意味着新的 `Message` 没有 `contentType` 标头集，从而确保 `contentType` 可以进化。 您可以始终选择退出处理程序方法的 `Message` ，您可以在其中插入任何标题。

如果存在内部管道，则通过经历相同的转换过程将 `Message` 发送到下一个处理程序。 但是，如果没有内部管道，或者已经到达管道末端，则 `Message` 被发送回输出目标。

30.1.1 Content Type versus Argument Type译：30.1.1内容类型与参数类型

如前所述，为了框架选择适当的 `MessageConverter` ，它需要参数类型和可选的内容类型信息。 用于选择适当的 `MessageConverter` 的逻辑驻留在参数解析器（ `HandlerMethodArgumentResolvers` ）中，该参数解析器 `HandlerMethodArgumentResolvers` 在调用用户定义的处理程序方法（实际参数类型为框架已知）之前触发。 如果参数类型与当前有效负载的类型不匹配，则框架将委托给预配置的 `MessageConverters` 的堆栈以查看它们中的任何一个是否可以转换有效负载。 如您所见，MessageConverter的 `Object fromMessage(Message<?> message, Class<?> targetClass);` 操作需要 `targetClass` 作为其参数之一。 该框架还确保提供的 `Message` 始终包含 `contentType` 标头。 如果没有contentType标头已经存在，它将注入每个绑定 `contentType` 标头或默认 `contentType` 标头。 参数类型 `contentType` 的组合是框架确定消息是否可以转换为目标类型的机制。 如果找不到合适的 `MessageConverter` ，则会引发异常，您可以通过添加自定义 `MessageConverter` （请参见“ Section 30.3, “User-defined Message Converters”” ）来处理该异常。

但是如果有效载荷类型与处理程序方法声明的目标类型匹配呢？ 在这种情况下，没有任何转换，并且有效负载未经修改就被传递。 虽然这听起来非常简单明了，但请记住以 `Message<?>` 或 `Object` 作为参数的处理程序方法。 通过将目标类型声明为 `Object` （这是 `instanceof` 所有内容），您基本上会放弃转换过程。

不要指望 `Message` 转换成其他类型的 `contentType` 。 请记住， `contentType` 与目标类型是互补的。 如果您愿意，您可以提供一个提示，其中 `MessageConverter` 可能会考虑也可能不考虑。

30.1.2 Message Converters译：30.1.2消息转换器

`MessageConverters` 定义了两种方法：

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);
```

了解这些方法的合同及其使用情况非常重要，特别是在Spring Cloud Stream环境中。

`fromMessage` 方法将传入的 `Message` 转换为参数类型。 `Message` 的有效载荷可以是任何类型，并且支持多种类型的 `MessageConverter` 的实际实施。 例如，某些转换器JSON可以支持有效载荷类型为 `byte[]` ， `String` ，以及其他。 当应用程序包含内部管道（即输入→处理程序→处理程序→...→输出）并且上游处理程序的输出导致 `Message` 可能不在最初的导线格式。

但是， `toMessage` 方法有一个更严格的合同，并且必须始终将 `Message` 转换为有线格式： `byte[]` 。

因此，对于所有意图和目的（特别是在实现自己的转换器时），您认为这两种方法具有以下签名：

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<byte[]> toMessage(Object payload, @Nullable MessageHeaders headers);
```

30.2 Provided MessageConverters 译：30.2提供 MessageConverters

如前所述，框架已经提供了一堆`MessageConverters`来处理最常见的用例。 以下列表按优先顺序（使用的第一个`MessageConverter`）描述了提供的`MessageConverters`：

1. `ApplicationJsonMessageMarshallingConverter`: Variation of the `org.springframework.messaging.converter.MappingJackson2MessageConverter`. Supports conversion of the payload of the `Message` to/from POJO for cases when `contentType` is `application/json` (DEFAULT).
2. `TupleJsonMessageConverter`: **DEPRECATED** Supports conversion of the payload of the `Message` to/from `org.springframework.tuple.Tuple`.
3. `ByteArrayMessageConverter`: Supports conversion of the payload of the `Message` from `byte[]` to `byte[]` for cases when `contentType` is `application/octet-stream`. It is essentially a pass through and exists primarily for backward compatibility.
4. `ObjectStringMessageConverter`: Supports conversion of any type to a `String` when `contentType` is `text/plain`. It invokes Object's `toString()` method or, if the payload is `byte[]`, a new `String(byte[])`.
5. `JavaSerializationMessageConverter`: **DEPRECATED** Supports conversion based on java serialization when `contentType` is `application/x-java-serialized-object`.
6. `KryoMessageConverter`: **DEPRECATED** Supports conversion based on Kryo serialization when `contentType` is `application/x-java-object`.
7. `JsonUnmarshallingConverter`: Similar to the `ApplicationJsonMessageMarshallingConverter`. It supports conversion of any type when `contentType` is `application/x-java-object`. It expects the actual type information to be embedded in the `contentType` as an attribute (for example, `application/x-java-object;type=foo.bar.Cat`).

当没有找到合适的转换器时，框架会引发异常。 发生这种情况时，应检查代码和配置，并确保不会错过任何内容（即，确保使用绑定或标题提供了`contentType`）。 但是，最有可能的是，您发现了一些不常见的情况（例如，也许是自定义`contentType`），并且当前提供的`MessageConverters`堆栈不知道如何转换。 如果是这种情况，您可以添加自定义`MessageConverter`。 见Section 30.3, “User-defined Message Converters”。

30.3 User-defined Message Converters 译：30.3用户定义的消息转换器

Spring Cloud Stream公开了定义和注册附加`MessageConverters`的机制。 要使用它，请执行`org.springframework.messaging.converter.MessageConverter`，将其配置为`@Bean`，并使用`@StreamMessageConverter`对其进行`@StreamMessageConverter`。 然后，它被添加到现有的`MessageConverter`堆栈中。

了解自定义`MessageConverter`实现添加到现有堆栈的头部很重要。 因此，自定义`MessageConverter`实现优先于现有的实现，这可以让您覆盖以及添加到现有的转换器。

以下示例显示如何创建一个消息转换器bean以支持名为`application/bar`的新内容类型：

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

    ...

    @Bean
    @StreamMessageConverter
    public MessageConverter customMessageConverter() {
        return new MyCustomMessageConverter();
    }
}

public class MyCustomMessageConverter extends AbstractMessageConverter {

    public MyCustomMessageConverter() {
        super(new MimeType("application", "bar"));
    }

    @Override
    protected boolean supports(Class<?> clazz) {
        return (Bar.class.equals(clazz));
    }

    @Override
    protected Object convertFromInternal(Message<?> message, Class<?> targetClass, Object conversionHint) {
        Object payload = message.getPayload();
        return (payload instanceof Bar ? payload : new Bar((byte[]) payload));
    }
}
```

Spring Cloud Stream还支持基于Avro的转换器和架构演变。 有关详细信息，请参阅“ Chapter 31, *Schema Evolution Support*”。

31. Schema Evolution Support 译：31.架构演变支持

Spring Cloud Stream为架构演进提供了支持，以便数据可以随时间演化，并且仍然适用于较旧或较新的生产者和消费者，反之亦然。 大多数序列化模型，特别是旨在跨不同平台和语言的可移植性的模型依赖于描述数据在二进制有效载荷中如何序列化的模式。 为了序列化数据然后解释它，发送方和接收方都必须能够访问描述二进制格式的模式。 在某些情况下，架构可以从序列化中的有效负载类型或反序列化中的目标类型推断出来。 但是，许多应用程序受益于访问描述二进制数据格式的显式模式。 模式注册表允许您以文本格式（通常为JSON）存储模式信息，并使该信息可供需要它以二进制格式接收和发送数据的各种应用程序访问。 模式可以引用为一个元组，其组成如下：

- A subject that is the logical name of the schema
- The schema version
- The schema format, which describes the binary format of the data

以下部分将详细介绍模式演变过程中涉及的各个组件。

31.1 Schema Registry Client 译：31.1架构注册表客户端

用于与模式注册服务器交互的客户端抽象是`SchemaRegistryClient`接口，它具有以下结构：

```
public interface SchemaRegistryClient {

    SchemaRegistrationResponse register(String subject, String format, String schema);

    String fetch(SchemaReference schemaReference);

    String fetch(Integer id);

}
```

Spring Cloud Stream提供了用于与自己的模式服务器进行交互以及与Confluent模式注册表交互的开箱即用的实现。

Spring云流模式注册中心的客户端可以使用 `@EnableSchemaRegistryClient` 进行配置，如下所示：

```
@EnableBinding(Sink.class)
@SpringBootApplication
@EnableSchemaRegistryClient
public static class AvroSinkApplication {
  ...
}
```

默认转换器经过优化，不仅可以缓存来自远程服务器的模式，还可以缓存 `parse()` 和 `toString()` 方法，这些方法相当昂贵。 因此，它使用不缓存响应的 `DefaultSchemaRegistryClient` 。 如果您打算更改默认行为，则可以直接在您的代码上使用客户端，并将其覆盖为期望的结果。 为此，您必须将属性 `spring.cloud.stream.schemaRegistryClient.cached=true` 添加到您的应用程序属性中。

31.1.1 Schema Registry Client Properties 译：31.1.1架构注册表客户端属性

Schema Registry Client支持以下属性：

`spring.cloud.stream.schemaRegistryClient.endpoint`
: The location of the schema-server. When setting this, use a full URL, including protocol (`http` or `https`) , port, and context path.

Default
: `http://localhost:8990/`

`spring.cloud.stream.schemaRegistryClient.cached`
: Whether the client should cache schema server responses. Normally set to `false`, as the caching happens in the message converter. Clients using the schema registry client should set this to `true`.

Default
: `true`

31.2 Avro Schema Registry Client Message Converters 译：31.2 Avro模式注册客户端消息转换器

对于具有向应用程序上下文注册的SchemaRegistryClient bean的应用程序，Spring Cloud Stream会自动配置用于模式管理的Apache Avro消息转换器。 这简化了模式演变，因为接收消息的应用程序可以轻松访问可以与自己的读者模式进行协调的作者模式。

对于出站消息，如果通道的内容类型设置为 `application/*+avro` ，则激活 `MessageConverter` ，如以下示例所示：

```
spring.cloud.stream.bindings.output.contentType=application/*+avro
```

在出站转换期间，消息转换器会尝试推断每个出站消息的架构（根据其类型），并使用 `SchemaRegistryClient` 其注册到主题（基于有效负载类型）。 如果已经找到相同的模式，则检索对其的引用。 如果不是，架构已注册，并提供新的版本号。 该消息使用以下方案通过 `contentType` 标头发送： `application/[prefix].[subject].v[version]+avro` ，其中 `prefix` 可配置， `subject` 从有效内容类型中推导出来。

例如， `User` 类型的消息可能作为内容类型为 `application/vnd.user.v2+avro` 的二进制有效内容 `application/vnd.user.v2+avro` ，其中 `user` 是主题， `2` 是版本号。

当接收到消息时，转换器从传入消息的头部推断模式引用并尝试检索它。 模式在反序列化过程中用作写模式。

31.2.1 Avro Schema Registry Message Converter Properties 译：31.2.1 Avro模式注册表消息转换器属性

如果通过设置 `spring.cloud.stream.bindings.output.contentType=application/*+avro` 启用了基于Avro的架构注册表客户端，则可以通过设置以下属性来自定义注册的行为。

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled
: 如果您希望转换器使用反射来从POJO推断架构，请启用。

  默认： `false`

spring.cloud.stream.schema.avro.readerSchema
: Avro compares schema versions by looking at a writer schema (origin payload) and a reader schema (your application payload). See the Avro documentation for more information. If set, this overrides any lookups at the schema server and uses the local schema as the reader schema. Default: `null`

spring.cloud.stream.schema.avro.schemaLocations
: 使用架构服务器注册此属性中列出的所有 `.avsc` 文件。

  默认： `empty`

spring.cloud.stream.schema.avro.prefix
: 在Content-Type头部使用的前缀。

  默认： `vnd`

31.3 Apache Avro Message Converters 译：31.3 Apache Avro消息转换器

Spring Cloud Stream通过其 `spring-cloud-stream-schema` 模块为基于模式的消息转换器提供支持。 目前，基于模式的消息转换器开箱即用的唯一序列化格式是Apache Avro，未来版本中将添加更多格式。

`spring-cloud-stream-schema` 模块包含两种可用于Apache Avro序列化的消息转换器：

- Converters that use the class information of the serialized or deserialized objects or a schema with a location known at startup.
- Converters that use a schema registry. They locate the schemas at runtime and dynamically register new schemas as domain objects evolve.

31.4 Converters with Schema Support译：31.4带架构支持的转换器

`AvroSchemaMessageConverter`支持使用预定义模式或使用类中可用的模式信息（反射地或包含在`SpecificRecord`）来序列化和反序列化消息。 如果您提供自定义转换器，则不会创建默认的AvroSchemaMessageConverter bean。 以下示例显示了一个自定义转换器：

要使用自定义转换器，只需将其添加到应用程序上下文中，可以指定一个或多个`MimeTypes`关联的`MimeTypes`。 默认`MimeType`是`application/avro`。

如果目标类型的转换是`GenericRecord`，则必须设置模式。

以下示例显示如何通过在没有预定义架构的情况下注册Apache Avro `MessageConverter`来配置接收器应用程序中的转换器。 在此示例中，请注意，mime类型值是`avro/bytes`，而不是默认的`application/avro`。

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      return new AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));
  }
}
```

相反，以下应用程序使用预定义架构（在类路径中找到）注册转换器：

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      AvroSchemaMessageConverter converter = new AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));
      converter.setSchemaLocation(new ClassPathResource("schemas/User.avro"));
      return converter;
  }
}
```

31.5 Schema Registry Server译：31.5架构注册表服务器

Spring Cloud Stream提供了架构注册表服务器实现。 要使用它，可以将`spring-cloud-stream-schema-server`工件添加到项目中，并使用`@EnableSchemaRegistryServer`注释，该注释将模式注册表服务器REST控制器添加到您的应用程序中。 此注释旨在用于Spring Boot Web应用程序，并且服务器的侦听端口由`server.port`属性控制。 可以使用`spring.cloud.stream.schema.server.path`属性来控制架构服务器的根路径（尤其是将其嵌入到其他应用程序中时）。 `spring.cloud.stream.schema.server.allowSchemaDeletion`布尔属性可以删除模式。 默认情况下，这是禁用的。

模式注册中心服务器使用关系数据库来存储模式。 默认情况下，它使用嵌入式数据库。 您可以使用Spring Boot SQL database and JDBC configuration options自定义架构存储。

以下示例显示了启用架构注册表的Spring Boot应用程序：

```
@SpringBootApplication
@EnableSchemaRegistryServer
public class SchemaRegistryServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SchemaRegistryServerApplication.class, args);
    }
}
```

31.5.1 Schema Registry Server API译：31.5.1架构注册服务器API

架构注册服务器API包含以下操作：

- `POST /` — see “the section called “Registering a New Schema””
- 'GET /{subject}/{format}/{version}' — see “the section called “Retrieving an Existing Schema by Subject, Format, and Version””
- `GET /{subject}/{format}` — see “the section called “Retrieving an Existing Schema by Subject and Format””
- `GET /schemas/{id}` — see “the section called “Retrieving an Existing Schema by ID””
- `DELETE /{subject}/{format}/{version}` — see “the section called “Deleting a Schema by Subject, Format, and Version””
- `DELETE /schemas/{id}` — see “the section called “Deleting a Schema by ID””
- `DELETE /{subject}` — see “the section called “Deleting a Schema by Subject””

Registering a New Schema译：注册新的模式

要注册新架构，请将`POST`请求发送到`/`端点。

`/`接受带有以下字段的JSON有效内容：

- `subject`: The schema subject
- `format`: The schema format
- `definition`: The schema definition

它的响应是JSON中的模式对象，具有以下字段：

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version

- `definition`: The schema definition

Retrieving an Existing Schema by Subject, Format, and Version译:按主题，格式和版本检索现有架构

要按主题，格式和版本检索现有架构，请将`GET`请求发送到`/{subject}/{format}/{version}`端点。

它的响应是JSON中的模式对象，具有以下字段：

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Retrieving an Existing Schema by Subject and Format译:按主题和格式检索现有架构

要按主题和格式检索现有架构，请将`GET`请求发送到`/subject/format`端点。

它的响应是JSON中每个模式对象的模式列表，其中包含以下字段：

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Retrieving an Existing Schema by ID译:按ID检索现有模式

要通过ID检索架构，请将`GET`请求发送到`/schemas/{id}`端点。

它的响应是JSON中的模式对象，具有以下字段：

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Deleting a Schema by Subject, Format, and Version译:按主题，格式和版本删除架构

要删除由其主题，格式和版本标识的模式，请将`DELETE`请求发送到`/{subject}/{format}/{version}`端点。

Deleting a Schema by ID译:通过ID删除架构

要通过其ID删除架构，请将`DELETE`请求发送到`/schemas/{id}`端点。

Deleting a Schema by Subject译:按主题删除架构

`DELETE /{subject}`

按主题删除现有模式。

本说明仅适用于Spring Cloud Stream 1.1.0.RELEASE的用户。 Spring Cloud Stream 1.1.0.RELEASE使用表名`schema`来存储`Schema`对象。 `Schema`是许多数据库实现中的关键字。 为避免将来出现任何冲突，从1.1.1.RELEASE开始，我们选择存储表的名称为`SCHEMA_REPOSITORY`。 任何升级的Spring Cloud Stream 1.1.0.RELEASE用户都应在升级之前将其现有模式迁移到新表中。

31.5.2 Using Confluent's Schema Registry译：31.5.2使用Confluent的模式注册表

默认配置会创建一个`DefaultSchemaRegistryClient` bean。 如果你想使用Confluent模式注册表，你需要创建一个类型为`ConfluentSchemaRegistryClient`的bean，它取代了框架默认配置的bean。 以下示例显示如何创建这样一个bean：

```
@Bean
public SchemaRegistryClient schemaRegistryClient(@Value("${spring.cloud.stream.schemaRegistryClient.endpoint}") String endpoint){
  ConfluentSchemaRegistryClient client = new ConfluentSchemaRegistryClient();
  client.setEndpoint(endpoint);
  return client;
}
```

ConfluentSchemaRegistryClient已针对Confluent平台版本4.0.0进行了测试。

31.6 Schema Registration and Resolution译：31.6架构注册和解决方案

为了更好地理解Spring Cloud Stream如何注册和解析新模式以及如何使用Avro模式比较功能，我们提供了两个独立的小节：

- "Section 31.6.1, "Schema Registration Process (Serialization)""
- "Section 31.6.2, "Schema Resolution Process (Deserialization)""

31.6.1 Schema Registration Process (Serialization)译：31.6.1模式注册过程（序列化）

注册过程的第一部分是从通过通道发送的有效载荷中提取模式。 Avro类型（如`SpecificRecord`或`GenericRecord`已包含一个架构，可以立即从实例中检索该架构。 对于POJO，如果`spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled`属性设置为`true`（默认值），则会推断模式。

图31.1。 模式编写器解析过程

如果获得一个模式，转换器会从远程服务器加载其元数据（版本）。 首先，它查询本地缓存。 如果未找到任何结果，则会将数据提交给服务器，服务器会回复版本信息。 转换器总是缓存结果以避免查询架构服务器以查找需要序列化的每条新消息的开销。

图31.2。 模式注册过程

使用模式版本信息，转换器将设置消息的 `contentType` 标题以携带版本信息 - 例如： `application/vnd.user.v1+avro` 。

31.6.2 Schema Resolution Process (Deserialization)译：31.6.2模式解析过程（反序列化）

阅读包含版本信息的消息时（即 `contentType` 标题，其格式类似于“ `contentType` 所述的格式），转换器将查询模式服务器以获取消息的编写器模式。 一旦发现传入消息的正确模式，它就会检索阅读器模式，并通过使用Avro的模式解析支持，将其读入阅读器定义（设置默认值和缺少的属性）。

图31.3。 模式阅读解决过程

您应该了解作者模式（编写消息的应用程序）和读者模式（接收应用程序）之间的区别。 我们建议花点时间阅读the Avro terminology并了解过程。 Spring Cloud Stream总是提取作者模式以确定如何阅读消息。 如果你想让Avro的架构进化支持起作用，你需要确保为你的应用程序正确设置了 `readerSchema` 。

32. Inter-Application Communication译：32.应用程序间通信

Spring Cloud Stream支持应用程序之间的通信。 如下面的主题所述，应用程序间通信是一个涉及多个问题的复杂问题：

- “Section 32.1, “Connecting Multiple Application Instances””
- “Section 32.2, “Instance Index and Instance Count””
- “Section 32.3, “Partitioning””

32.1 Connecting Multiple Application Instances译：32.1连接多个应用程序实例

虽然Spring Cloud Stream使各个Spring Boot应用程序可以轻松连接到消息传递系统，但Spring Cloud Stream的典型场景是创建多应用程序管道，微服务应用程序将数据发送给对方。 您可以通过关联“相邻”应用程序的输入和输出目标来实现此场景。

假设一个设计需要Time Source应用程序将数据发送到Log Sink应用程序。 您可以在两个应用程序中使用名为 `ticktock` 的通用目标进行绑定。

时间源（具有频道名称 `output` ）将设置以下属性：

```
spring.cloud.stream.bindings.output.destination=ticktock
```

日志接收器（具有通道名称 `input` ）将设置以下属性：

```
spring.cloud.stream.bindings.input.destination=ticktock
```

32.2 Instance Index and Instance Count译：32.2实例索引和实例计数

在扩展Spring Cloud Stream应用程序时，每个实例都可以接收有关同一应用程序的多少个其他实例以及它自己的实例索引是什么的信息。 Spring Cloud Stream通过 `spring.cloud.stream.instanceCount` 和 `spring.cloud.stream.instanceIndex` 属性执行此 `spring.cloud.stream.instanceIndex` 。 例如，如果有一个HDFS接收器应用程序的三个实例中，所有三个实例具有 `spring.cloud.stream.instanceCount` 设定为 `3` ，和各个应用程序具有 `spring.cloud.stream.instanceIndex` 设定为 `0` ， `1` ，和 `2` ，分别。

当通过Spring Cloud Data Flow部署Spring Cloud Stream应用程序时，这些属性会自动配置; 当Spring Cloud Stream应用程序独立启动时，必须正确设置这些属性。 默认情况下， `spring.cloud.stream.instanceCount` 是 `1` ，而 `spring.cloud.stream.instanceIndex` 是 `0` 。

在放大的场景中，这两个属性的正确配置对于解决一般的分区行为（见下文）非常重要，并且这两个属性总是需要某些绑定器（例如，Kafka绑定器）以确保数据在多个消费者实例中正确分割。

32.3 Partitioning译：32.3分区

Spring Cloud中的分区流包含两项任务：

- “Section 32.3.1, “Configuring Output Bindings for Partitioning””
- “Section 32.3.2, “Configuring Input Bindings for Partitioning””

32.3.1 Configuring Output Bindings for Partitioning译：32.3.1配置输出绑定以进行分区

您可以配置输出绑定来发送分区数据，方法是设置 `partitionKeyExpression` 或 `partitionKeyExtractorName` 属性及其 `partitionCount` 属性中的一个（仅限其中一个）属性。

例如，以下是有效且典型的配置：

```
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id
spring.cloud.stream.bindings.output.producer.partitionCount=5
```

基于该示例配置，通过使用以下逻辑将数据发送到目标分区。

根据partitionKeyExpression计算每个发送到分区输出通道的消息的分区 `partitionKeyExpression` 。 `partitionKeyExpression` 是SpEL表达式，它针对用于提取分区键的出站邮件进行评估。

如果SpEL表达式不足以满足您的需求，则可以通过提供 `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` 的实现并将其配置为bean（使用 `@Bean` 注释）来计算分区键值。 如果在应用程序上下文中有多个可用的类型为 `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` bean，则可以通过使用 `partitionKeyExtractorName` 属性指定其名称来进一步对其进行过滤，如以下示例所示：

```
--spring.cloud.stream.bindings.output.producer.partitionKeyExtractorName=customPartitionKeyExtractor
--spring.cloud.stream.bindings.output.producer.partitionCount=5
. . .
@Bean
public CustomPartitionKeyExtractorClass customPartitionKeyExtractor() {
    return new CustomPartitionKeyExtractorClass();
}
```

在之前版本的Spring Cloud Stream中，您可以通过设置`spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass`属性来指定`org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy`的实现。 自2.0版以来，该属性已被弃用，并且在将来的版本中将会删除对它的支持。

计算完消息密钥后，分区选择过程将目标分区确定为介于`0`和`partitionCount - 1`之间的值。 默认计算适用于大多数情况下，基于以下公式：`key.hashCode() % partitionCount` 。 这可以对结合进行定制，通过设置Spel位表达式来针对“关键”（通过评估`partitionSelectorExpression`属性）或者通过配置的实施`org.springframework.cloud.stream.binder.PartitionSelectorStrategy`作为豆（通过使用@Bean注释）。 类似于`PartitionKeyExtractorStrategy` ，如果在应用程序上下文中有多个此类型的bean可用，则可以使用`spring.cloud.stream.bindings.output.producer.partitionSelectorName`属性进一步对其进行过滤，如以下示例所示：

```
--spring.cloud.stream.bindings.output.producer.partitionSelectorName=customPartitionSelector
. . .
@Bean
public CustomPartitionSelectorClass customPartitionSelector() {
    return new CustomPartitionSelectorClass();
}
```

在之前版本的Spring Cloud Stream中，您可以通过设置`spring.cloud.stream.bindings.output.producer.partitionSelectorClass`属性来指定`org.springframework.cloud.stream.binder.PartitionSelectorStrategy`的实现。 自2.0版以来，此属性已被弃用，并且将在未来的版本中删除对它的支持。

32.3.2 Configuring Input Bindings for Partitioning译：32.3.2配置输入绑定以进行分区

输入结合（与频道名称`input` ）被配置为通过设置其接收分区数据`partitioned`属性，以及所述`instanceIndex`点`instanceCount`于应用本身的特性，如显示在下面的例子：

```
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceIndex=3
spring.cloud.stream.instanceCount=5
```

`instanceCount`值表示数据应在其间分区的应用程序实例的总数。 `instanceIndex`必须是多个实例中的唯一值，其值在`0`和`instanceCount - 1`之间。 实例索引可帮助每个应用程序实例识别它从中接收数据的唯一分区。 这是使用不支持本地分区的技术的绑定器所必需的。 例如，对于RabbitMQ，每个分区都有一个队列，队列名称包含实例索引。 使用Kafka时，如果`autoRebalanceEnabled`为`true` （默认值），则Kafka负责跨实例分配分区，并且不需要这些属性。 如果`autoRebalanceEnabled`设置为false，则`instanceCount` `instanceIndex`将使用`instanceCount`和`instanceIndex`来确定实例订阅的分区（您必须至少具有与实例一样多的分区）。 活页夹分配分区而不是Kafka。 如果您希望特定分区的消息始终转到同一实例，这可能会很有用。 当联编程序配置需要它们时，正确设置这两个值很重要，以确保所有数据都被消耗，并且应用程序实例接收互斥数据集。

虽然使用多个实例进行分区数据处理的方案在独立案例中设置可能很复杂，但Spring Cloud Dataflow可以通过正确填充输入和输出值以及让您依赖运行时基础架构来显着简化流程提供有关实例索引和实例计数的信息。

33. Testing译：33.测试

Spring Cloud Stream支持在不连接消息传递系统的情况下测试您的微服务应用程序。 您可以通过使用`spring-cloud-stream-test-support`库提供的`spring-cloud-stream-test-support`来完成该`TestSupportBinder` ，该库可以作为测试依赖项添加到应用程序中，如以下示例所示：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

`TestSupportBinder`使用Spring Boot自动配置机制来取代类路径中找到的其他绑定器。 因此，添加活页夹作为依赖项时，必须确保正在使用`test`范围。

`TestSupportBinder`允许您与绑定通道进行交互，并检查应用程序发送和接收的任何消息。

对于出站消息通道， `TestSupportBinder`注册单个订户，并将应用程序发出的消息保留在`MessageCollector` 。 他们可以在测试过程中检索并对他们提出断言。

您还可以将消息发送到入站消息通道，以便消费者应用程序可以使用消息。 以下示例显示如何在处理器上测试输入和输出通道：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ExampleTest {

  @Autowired
  private Processor processor;

  @Autowired
  private MessageCollector messageCollector;

  @Test
  @SuppressWarnings("unchecked")
  public void testWiring() {
    Message<String> message = new GenericMessage<>("hello");
    processor.input().send(message);
    Message<String> received = (Message<String>) messageCollector.forChannel(processor.output()).poll();
    assertThat(received.getPayload(), equalTo("hello world"));
  }


  @SpringBootApplication
  @EnableBinding(Processor.class)
  public static class MyProcessor {

    @Autowired
    private Processor channels;

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public String transform(String in) {
      return in + " world";
    }
  }
}
```

在前面的例子中，我们创建了一个具有输入通道和输出通道的应用程序，它们都通过`Processor`接口绑定。 绑定的接口被注入到测试中，以便我们可以访问这两个通道。 我们在输入通道发送消息，以及我们使用`MessageCollector`由Spring云Streamâ€™的测试提供支持，以捕获该消息已发送到输出通道的结果。 收到消息后，我们可以验证组件是否正常工作。

33.1 Disabling the Test Binder Autoconfiguration译：33.1禁用测试文件夹自动配置

测试联编程序背后的意图是取代classpath中的所有其他联编程序，以便在不更改生产依赖关系的情况下轻松测试应用程序。 在某些情况下（例如集成测试），使用实际的生产绑定程序会很有用，而且这需要禁用测试绑定器自动配置。 为此，可以使用Spring Boot自动配置排除机制之一排除`org.springframework.cloud.stream.test.binder.TestSupportBinderAutoConfiguration`类，如以下示例所示：

```
    @SpringBootApplication(exclude = TestSupportBinderAutoConfiguration.class)
    @EnableBinding(Processor.class)
    public static class MyProcessor {

        @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
        public String transform(String in) {
            return in + " world";
        }
    }
```

禁用自动配置时，测试联编程序在类路径中可用，并且其`defaultCandidate`属性设置为`false`以便它不干扰常规用户配置。 它可以在名称`test`下引用，如以下示例所示：

```
spring.cloud.stream.defaultBinder=test
```

34. Health Indicator译：34健康指标

Spring Cloud Stream为绑定器提供健康指示器。 它以名称`binders`注册，可通过设置`management.health.binders.enabled`属性启用或禁用。

默认`management.health.binders.enabled`设置为`false` 。 将`management.health.binders.enabled`设置为`true`可启用运行状况指示器，从而允许您访问`/health`端点以检索活页夹运行状况指示器。

健康指示符是特定于活页夹的，某些活页夹实现可能不一定提供健康指示符。

35. Metrics Emitter译：35度量发射器

Spring Boot Actuator为 Micrometer提供依赖关系管理和自动配置， Micrometer是一个支持众多 monitoring systems的应用程序度量门面。

Spring Cloud Stream支持将任何可用的基于千分尺的指标发布到绑定目标，允许从流应用程序定期收集指标数据，而不依赖于轮询单个端点。

通过定义`spring.cloud.stream.bindings.applicationMetrics.destination`属性来激活度量标准发射器，该属性指定当前活页夹用于发布度量标准消息的绑定目标的名称。

例如：

```
spring.cloud.stream.bindings.applicationMetrics.destination=myMetricDestination
```

前面的示例指示绑定器绑定到`myMetricDestination` （即，兔子交换，Kafka主题等）。

以下属性可用于自定义指标的排放：

spring.cloud.stream.metrics.key

正在发布的度量标准的名称。 应该是每个应用程序的独特价值。

默认：`${spring.application.name:${vcap.application.name:${spring.config.name:application}}}`

spring.cloud.stream.metrics.properties

允许添加到指标有效负载的白名单应用程序属性

默认值：null。

spring.cloud.stream.metrics.meter-filter

模式来控制人们想要捕捉的'米'。 例如，指定`spring.integration.*`捕获名称以`spring.integration.`开头的仪表的度量信息

默认值：所有'米'被捕获。

spring.cloud.stream.metrics.schedule-interval

间隔来控制发布度量标准数据的速度。

默认值：1分钟

考虑以下：

```
java -jar time-source.jar \
    --spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \
    --spring.cloud.stream.metrics.properties=spring.application** \
    --spring.cloud.stream.metrics.meter-filter=spring.integration.*
```

以下示例显示了作为上述命令结果发布到绑定目标的数据：

```
{
 "name": "application",
 "createdTime": "2018-03-23T14:48:12.700Z",
 "properties": {
 },
 "metrics": [
  {
   "id": {
    "name": "spring.integration.send",
    "tags": [
     {
      "key": "exception",
      "value": "none"
     },
     {
      "key": "name",
      "value": "input"
     },
     {
      "key": "result",
      "value": "success"
     },
     {
      "key": "type",
      "value": "channel"
     }
    ],
    "type": "TIMER",
    "description": "Send processing time",
    "baseUnit": "milliseconds"
   },
   "timestamp": "2018-03-23T14:48:12.697Z",
   "sum": 130.340546,
   "count": 6,
   "mean": 333,
   "upper": 116.176299,
   "total": 130.340546
  }
 ]
}
```

36. Samples译：36.样品

对于Spring Cloud Stream示例，请参阅GitHub上的 spring-cloud-stream-samples存储库。

36.1 Deploying Stream Applications on CloudFoundry译：36.1在 CloudFoundry上 部署流应用程序

在CloudFoundry上，服务通常通过名为 VCAP_SERVICES的特殊环境变量 公开 。

在配置活页夹连接时，可以使用 dataflow Cloud Foundry Server文档中介绍的环境变量中的值。

Part VI. Binder Implementations译：第六部分。 活页夹实施

37. Apache Kafka Binder译：37. Apache Kafka粘合剂

37.1 Usage译：37.1用法

要使用Apache Kafka联编程序，您需要将 `spring-cloud-stream-binder-kafka`作为依赖项添加到Spring Cloud Stream应用程序中，如以下Maven示例所示：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

或者，您也可以使用Spring Cloud Stream Kafka Starter，如Maven的以下示例所示：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

37.2 Apache Kafka Binder Overview译：37.2 Apache Kafka粘合剂概述

下图显示了Apache Kafka联编程序如何运行的简图：

图37.1。 卡夫卡粘结剂

Apache Kafka Binder实现将每个目标映射到Apache Kafka主题。 消费者组直接映射到相同的Apache Kafka概念。 分区也直接映射到Apache Kafka分区。

该活页夹目前使用Apache Kafka `kafka-clients` 1.0.0 jar，旨在与至少该版本的代理一起使用。 该客户端可以与旧经纪商进行通信（请参阅Kafka文档），但某些功能可能无法使用。 例如，对于早于0.11.xx的版本，不支持本机标头。 另外，0.11.xx不支持`autoAddPartitions`属性。

37.3 Configuration Options译：37.3配置选项

本节包含Apache Kafka活页夹使用的配置选项。

有关活页夹的常见配置选项和属性，请参阅 core documentation 。

37.3.1 Kafka Binder Properties译：37.3.1卡夫卡粘结剂特性

spring.cloud.stream.kafka.binder.brokers

Kafka活页夹连接的经纪人列表。

默认： `localhost` 。

spring.cloud.stream.kafka.binder.defaultBrokerPort

`brokers`允许指定具有或不具有端口信息的主机（例如， `host1,host2:port2` ）。 当代理列表中没有配置端口时，这将设置默认端口。

默认： `9092` 。

spring.cloud.stream.kafka.binder.configuration

将客户端属性（生产者和消费者）的键/值映射传递给由活页夹创建的所有客户端。 由于生产者和消费者都使用这些属性，所以用法应该限制在通用属性 - 例如安全设置中。

默认：空地图。

spring.cloud.stream.kafka.binder.headers

由活页夹传输的自定义标题的列表。 只有在使用`kafka-clients`版本<0.11.0.0与旧应用程序（ `kafka-clients` 1.3.x）进行通信时才需要。 较新版本本身支持标题。

默认值：空。

spring.cloud.stream.kafka.binder.healthTimeout

等待获取分区信息的时间，以秒为单位。 如果此计时器到期，则运行状况报告为关闭。

默认值：10。

spring.cloud.stream.kafka.binder.requiredAcks

经纪人需要的确认数量。 请参阅生产者`acks`属性的Kafka文档。

默认： `1` 。

spring.cloud.stream.kafka.binder.minPartitionCount

仅在设置`autoCreateTopics`或`autoAddPartitions`有效。 资料夹在产生或消耗数据的主题上配置的全局最小分区数。 它可以被生产者的`partitionCount`设置或生产者的`instanceCount * concurrency`设置的值`instanceCount * concurrency` （如果其中任何一个更大）。

默认： `1` 。

spring.cloud.stream.kafka.binder.replicationFactor

如果`autoCreateTopics`处于活动状态，则自动创建主题的复制因子。 可以在每个绑定上重写。

默认： `1` 。

spring.cloud.stream.kafka.binder.autoCreateTopics

如果设置为`true` ，资料夹会自动创建新主题。 如果设置为`false` ，则活页夹依赖于已配置的主题。 在后一种情况下，如果主题不存在，则活页夹无法启动。

此设置独立于代理的`auto.topic.create.enable`设置，不影响它。 如果服务器设置为自动创建主题，则可以使用默认代理设置将它们创建为元数据检索请求的一部分。

默认： `true` 。

spring.cloud.stream.kafka.binder.autoAddPartitions

如果设置为`true` ，则活页夹会根据需要创建新的分区。 如果设置为`false` ，则活页夹依赖于已配置主题的分区大小。 如果目标主题的分区数小于预期值，则活页夹无法启动。

默认： `false` 。

spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix

启用资料夹中的交易。 见transaction.id卡夫卡文档和Transactions中spring-kafka文档。 启用交易时，将忽略单个producer属性，并且所有生产者都使用spring.cloud.stream.kafka.binder.transaction.producer.*属性。

默认 null （无交易）

spring.cloud.stream.kafka.binder.transaction.producer.*

交易活页夹中生产者的全球生产者属性。 请参阅spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix和Section 37.3.3, “Kafka Producer Properties”以及所有联编程序支持的常规生产者属性。

默认值：查看单个生产者属性。

spring.cloud.stream.kafka.binder.headerMapperBeanName

KafkaHeaderMapper的bean名称，用于将spring-messaging标题映射到Kafka标题和从中映射。 例如，如果您希望自定义DefaultKafkaHeaderMapper中使用JSON反序列化标头的可信软件包，请使用此选项。

默认值：无。

37.3.2 Kafka Consumer Properties译: 37.3.2卡夫卡消费属性

以下属性仅供卡夫卡消费者使用，且必须以 spring.cloud.stream.kafka.bindings.<channelName>.consumer.为前缀。

admin.configuration

一个 Map卡夫卡话题性的供应时topicsâ€‰€”â€‰一起使用，例如 spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0

默认值：无。

admin.replicas-assignment

一个Map <Integer，List <Integer>副本分配，键为分区，值为赋值。 在配置新主题时使用。 请参阅kafka-clients罐子中的NewTopic Javadocs。

默认值：无。

admin.replication-factor

配置主题时使用的复制因子。 覆盖绑定器范围的设置。 如果replicas-assignments存在，则忽略。

默认值：无（使用绑定器范围内的默认值1）。

autoRebalanceEnabled

当true，主题分区会在消费者组的成员之间自动重新平衡。 当false，每个消费者被分配一组基于spring.cloud.stream.instanceCount和spring.cloud.stream.instanceIndex的固定分区。 这需要在每个启动的实例上适当地设置spring.cloud.stream.instanceCount和spring.cloud.stream.instanceIndex属性。 在这种情况下， spring.cloud.stream.instanceCount属性的值通常必须大于1。

默认： true 。

ackEachRecord

autoCommitOffset为true，此设置决定是否在每个记录处理完成后提交偏移量。 默认情况下，在处理由consumer.poll()返回的批记录中的所有记录之后，将提交补偿。 通过投票返回的记录数量可以通过max.poll.records卡夫卡属性进行控制，该属性通过消费者configuration属性设置。 将其设置为true可能会导致性能下降，但这样做会降低发生故障时重新传输记录的可能性。 此外，请参阅活页夹requiredAcks属性，这也会影响提交偏移的性能。

默认： false 。

autoCommitOffset

是否在处理消息时自动提交偏移量。 如果设置为false，与密钥的报头kafka_acknowledgment的类型的org.springframework.kafka.support.Acknowledgment标头存在的入站消息中。 应用程序可以使用这个头来确认消息。 详细信息请参阅示例部分。 当此属性设置为false，卡夫卡粘结剂设置ACK模式为org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL和应用程序负责确认记录。 另请参阅ackEachRecord 。

默认： true 。

autoCommitOnError

仅在autoCommitOffset设置为true 。 如果设置为false，则会禁止导致错误并仅提交成功消息的消息的自动提交。 它允许流在上次成功处理的消息中自动重播，以防持续失败。 如果设置为true，它总是自动提交（如果启用了自动提交）。 如果没有设置（默认值），它实际上具有与enableDlq相同的值，如果将它们发送到DLQ而不提交它们，则自动提交错误消息。

默认值：未设置。

resetOffsets

是否将消费者偏移重置为startOffset提供的值。

默认： false 。

startOffset

新组的起始偏移量。 允许值： earliest和latest 。 如果消费者组被明确设置为消费者'绑定'（通过spring.cloud.stream.bindings.<channelName>.group），则'startOffset'被设置为earliest 。 否则， anonymous组的设置为latest 。 另请参阅resetOffsets （本列表前面的部分）。

默认值：null（相当于 earliest ）。

enableDlq

设置为true时，它将启用消费者的DLQ行为。 默认情况下，导致错误的消息被转发到名为error.<destination>.<group>的主题。 DLQ主题名称可以通过设置dlqName属性进行配置。 这为更常见的Kafka重播场景提供了一种替代选择，适用于错误数量相对较少并重放整个原始主题可能过于麻烦的情况。 有关更多信息，请参阅Section 37.6, “Dead-Letter Topic Processing”处理。 从2.0版开始，发送到DLQ主题的消息已得到增强，以下标题： x-original-topic ，x-exception-message ，并x-exception-stacktrace为byte[] 。

默认： false 。

configuration

映射包含通用Kafka使用者属性的键/值对。

默认：空地图。

dlqName

接收错误消息的DLQ主题的名称。

默认值：空（如果未指定，导致错误的消息将转发到名为 `error.<destination>.<group>` 的主题）。

dlqProducerProperties

使用此功能，可以设置DLQ特定的生产者属性。 通过kafka生产者属性可用的所有属性都可以通过此属性设置。

默认：默认Kafka生产者属性。

standardHeaders

指示哪些标准头由入站通道适配器填充。 允许值： `none` ， `id` ， `timestamp` ，或 `both` 。 如果使用本地反序列化并且第一个组件接收消息需要 `id` （例如配置为使用JDBC消息存储库的聚合器），则这很有用。

默认： `none`

converterBeanName

实现 `RecordMessageConverter` 的bean的名称。 在入站通道适配器中使用，以替换默认的 `MessagingMessageConverter` 。

默认： `null`

idleEventInterval

指示最近没有收到消息的事件之间的间隔（以毫秒为单位）。 使用 `ApplicationListener<ListenerContainerIdleEvent>` 来接收这些事件。 有关使用示例，请参见the section called “Example: Pausing and Resuming the Consumer” 。

默认： `30000`

37.3.3 Kafka Producer Properties 译：37.3.3卡夫卡生产者属性

以下属性仅供卡夫卡制作人使用，且必须以 `spring.cloud.stream.kafka.bindings.<channelName>.producer.` 为前缀。

admin.configuration

一个 `Map` 卡夫卡话题性的，例如提供新topicsâ€‰€"当€‰使用，
`spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0`

默认值：无。

admin.replicas-assignment

一个Map <Integer，List <Integer>副本分配，键为分区，值为赋值。 在配置新主题时使用。 见 `NewTopic` 中的javadoc `kafka-clients` 罐子。

默认值：无。

admin.replication-factor

设置新主题时使用的复制因子。 覆盖绑定器范围的设置。 如果存在 `replicas-assignments` 则忽略。

默认值：无（使用绑定器范围内的默认值1）。

bufferSize

以字节为单位的上限，即卡夫卡生产者在发送之前尝试批量处理的数据量。

默认： `16384` 。

sync

生产者是否同步。

默认： `false` 。

batchTimeout

在发送消息之前，生产者等待多长时间允许更多消息在同一批中累积。 （通常，生产者根本不会等待，只是发送前一次发送过程中累积的所有消息。）非零值可能会增加吞吐量，但会以延迟为代价。

默认： `0` 。

messageKeyExpression

SpEL表达式针对用于填充生成的Kafka消息的密钥（例如 `headers['myKey']` 的传出消息进行评估。 有效载荷无法使用，因为在评估此表达式时，有效载荷已经以 `byte[]` 的形式 `byte[]` 。

默认： `none` 。

headerPatterns

以逗号分隔的简单模式列表，用于匹配Spring邮件标题以映射到 `Headers` 中的Kafka `ProducerRecord` 。 模式可以以通配符（星号）开始或结束。 通过用 `!` 作为前缀，模式可以被否定。 匹配在第一场比赛后停止（正面或负面）。 例如 `!ask,as*` 将通过 `ash` 而不是 `ask` 。 `id` 和 `timestamp` 从不映射。

默认： `*` （所有标题 - `id` 和 `timestamp` ）

configuration

映射包含通用Kafka生产者属性的键/值对。

默认：空地图。

Kafka联编程序使用生产者的`partitionCount`设置作为提示来创建具有给定分区计数的主题（与`minPartitionCount`一起使用，其中两个值的最大值是正在使用的值）。 配置两个时务必小心`minPartitionCount`用于粘合剂和`partitionCount`为应用程序，作为更大的值被使用。 如果某个主题已经存在，并且分区数量较小，并且禁用了`autoAddPartitions`（默认设置），则活页夹无法启动。 如果某个主题已经存在，并且分区数量较小，并且启用了`autoAddPartitions`，则会添加新分区。 如果某个主题已经存在分区数大于（ `minPartitionCount`或`partitionCount` ）的最大分区数，则使用现有的分区数。

37.3.4 Usage examples译：37.3.4使用示例

在本节中，我们将展示如何在特定场景中使用上述属性。

Example: Setting `autoCommitOffset` to `false` and Relying on Manual Acking译：示例：将`autoCommitOffset`设置为`false`并依靠手动确认

此示例说明了如何在消费者应用程序中手动确认偏移量。

此示例要求将`spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset`设置为`false` 。 为您的示例使用相应的输入通道名称。

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class ManuallyAcknowdledgingConsumer {

 public static void main(String[] args) {
     SpringApplication.run(ManuallyAcknowdledgingConsumer.class, args);
 }

 @StreamListener(Sink.INPUT)
 public void process(Message<?> message) {
     Acknowledgment acknowledgment = message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT, Acknowledgment.class);
     if (acknowledgment != null) {
         System.out.println("Acknowledgment provided");
         acknowledgment.acknowledge();
     }
 }
}
```

Example: Security Configuration译：示例：安全配置

Apache Kafka 0.9支持客户端和代理之间的安全连接。 要利用此功能，请按照Apache Kafka Documentation以及Kafka 0.9 security guidelines from the Confluent documentation中的指导进行操作 。 使用`spring.cloud.stream.kafka.binder.configuration`选项为由活页夹创建的所有客户端设置安全属性。

例如，要将`security.protocol`设置为`SASL_SSL` ，请设置以下属性：

```
spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_SSL
```

所有其他安全属性可以用类似的方式设置。

在使用Kerberos时，请按照 reference documentation中的说明创建和引用JAAS配置。

Spring Cloud Stream支持使用JAAS配置文件和Spring Boot属性将JAAS配置信息传递给应用程序。

Using JAAS Configuration Files译：使用JAAS配置文件

可以使用系统属性为Spring Cloud Stream应用程序设置JAAS和（可选）krb5文件位置。 以下示例显示如何使用JAAS配置文件启动具有SASL和Kerberos的Spring Cloud Stream应用程序：

```
java -Djava.security.auth.login.config=/path.to/kafka_client_jaas.conf -jar log.jar \
  --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT
```

Using Spring Boot Properties译：使用Spring引导属性

作为拥有JAAS配置文件的替代方案，Spring Cloud Stream通过使用Spring Boot属性提供了一种为Spring Cloud Stream应用程序设置JAAS配置的机制。

以下属性可用于配置Kafka客户端的登录上下文：

spring.cloud.stream.kafka.binder.jaas.loginModule

登录模块名称。 在正常情况下不需要设置。

默认： `com.sun.security.auth.module.Krb5LoginModule` 。

spring.cloud.stream.kafka.binder.jaas.controlFlag

登录模块的控制标志。

默认： `required` 。

spring.cloud.stream.kafka.binder.jaas.options

使用包含登录模块选项的键/值对映射。

默认：空地图。

以下示例显示如何使用Spring Boot配置属性启动具有SASL和Kerberos的Spring Cloud Stream应用程序：

```
java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.autoCreateTopics=false \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT \
  --spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \
  --spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \
  --spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka_client.keytab \
  --spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client-1@EXAMPLE.COM
```

前面的例子代表了下面的JAAS文件的等价物：

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
};
```

如果所需的主题已存在于代理上或将由管理员创建，则可以关闭自动创建功能，并且只需发送客户端JAAS属性。

不要在同一应用程序中混合使用JAAS配置文件和Spring Boot属性。 如果`-Djava.security.auth.login.config`系统属性已存在，则Spring Cloud Stream将忽略Spring Boot属性。

在Kerberos中使用`autoCreateTopics`和`autoAddPartitions`时要小心。 通常，应用程序可能会使用在Kafka和Zookeeper中没有管理权限的主体。 因此，依靠Spring Cloud Stream来创建/修改主题可能会失败。 在安全环境中，我们强烈建议使用Kafka工具来管理性地创建主题并管理ACL。

Example: Pausing and Resuming the Consumer译:示例：暂停和恢复消费者

如果您希望暂停使用但不会导致分区重新平衡，则可以暂停并恢复使用者。 将`Consumer`作为参数添加到您的`@StreamListener`可以方便您进行`@StreamListener`。 要恢复，您需要一个`ApplicationListener``ListenerContainerIdleEvent`实例。 事件发布的频率由`idleEventInterval`属性控制。 由于消费者不是线程安全的，因此必须在调用线程上调用这些方法。

以下简单的应用程序显示了如何暂停和恢复：

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class Application {

 public static void main(String[] args) {
  SpringApplication.run(Application.class, args);
 }

 @StreamListener(Sink.INPUT)
 public void in(String in, @Header(KafkaHeaders.CONSUMER) Consumer<?, ?> consumer) {
  System.out.println(in);
  consumer.pause(Collections.singleton(new TopicPartition("myTopic", 0)));
 }

 @Bean
 public ApplicationListener<ListenerContainerIdleEvent> idleListener() {
  return event -> {
   System.out.println(event);
   if (event.getConsumer().paused().size() > 0) {
    event.getConsumer().resume(event.getConsumer().paused());
   }
  };
 }

}
```

37.4 Error Channels译：37.4错误通道

从版本1.3开始，活页夹无条件地将异常发送到每个使用者目标的错误通道，还可以配置为将异步生产者发送失败发送到错误通道。 有关更多信息，请参阅???。

发送失败的`ErrorMessage`的有效负载是`KafkaSendFailureException`具有以下属性：

- `failedMessage`: The Spring Messaging `Message<?>` that failed to be sent.
- `record`: The raw `ProducerRecord` that was created from the `failedMessage`

没有自动处理生产者异常（例如发送到Dead-Letter queue）。 您可以使用您自己的Spring Integration流程来使用这些异常。

37.5 Kafka Metrics译：37.5卡夫卡度量

Kafka活页夹模块公开了以下指标：

`spring.cloud.stream.binder.kafka.someGroup.someTopic.lag`：此度量指示给定使用者组从给定绑定器的主题尚未消耗多少个消息。 例如，如果度量`spring.cloud.stream.binder.kafka.myGroup.myTopic.lag`值为`1000`，则名为`myGroup`的使用者组将具有`1000`消息，该消息等待从主题`myTopic`消耗。 此度量标准对于向PaaS平台提供自动缩放反馈特别有用。

37.6 Dead-Letter Topic Processing译：37.6死信主题处理

由于您无法预测用户如何处理死信消息，因此该框架不提供任何标准机制来处理它们。 如果死书的原因是短暂的，您可能希望将消息路由回原始主题。 但是，如果问题是一个永久性问题，那可能会导致无限循环。 本主题中的示例Spring Boot应用程序是如何将这些消息路由回原始主题的示例，但它在三次尝试后将其移动到“停车场”主题。 该应用程序是另一个从死信主题中读取的spring-cloud-stream应用程序。 当5秒内没有收到消息时它终止。

这些示例假定原始目的地是`so8400out`，而消费者组是`so8400`。

有几个策略需要考虑：

- Consider running the rerouting only when the main application is not running. Otherwise, the retries for transient errors are used up very quickly.
- Alternatively, use a two-stage approach: Use this application to route to a third topic and another to route from there back to the main topic.

以下代码清单显示示例应用程序：

application.properties。

```
spring.cloud.stream.bindings.input.group=so8400replay
spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out
spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot
spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries
```

应用。

```
@SpringBootApplication
@EnableBinding(TwoOutputProcessor.class)
public class ReRouteDlqKApplication implements CommandLineRunner {

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) {
        SpringApplication.run(ReRouteDlqKApplication.class, args).close();
    }

    private final AtomicInteger processed = new AtomicInteger();

    @Autowired
    private MessageChannel parkingLot;

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Message<?> reRoute(Message<?> failed) {
        processed.incrementAndGet();
        Integer retries = failed.getHeaders().get(X_RETRIES_HEADER, Integer.class);
        if (retries == null) {
            System.out.println("First retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else if (retries.intValue() < 3) {
            System.out.println("Another retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(retries.intValue() + 1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else {
            System.out.println("Retries exhausted for " + failed);
            parkingLot.send(MessageBuilder.fromMessage(failed)
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build());
        }
        return null;
    }

    @Override
    public void run(String... args) throws Exception {
        while (true) {
            int count = this.processed.get();
            Thread.sleep(5000);
            if (count == this.processed.get()) {
                System.out.println("Idle, terminating");
                return;
            }
        }
    }

    public interface TwoOutputProcessor extends Processor {

        @Output("parkingLot")
        MessageChannel parkingLot();

    }

}
```

37.7 Partitioning with the Kafka Binder译：37.7使用Kafka结合测进行分区

Apache Kafka本身支持主题分区。

有时将数据发送到特定分区是有利的，例如，当您要严格订购消息处理时（特定客户的所有消息都应该到同一个分区）。

以下示例显示如何配置生产者和消费者方面：

```
@SpringBootApplication
@EnableBinding(Source.class)
public class KafkaPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
            "foo1", "bar1", "qux1",
            "foo2", "bar2", "qux2",
            "foo3", "bar3", "qux3",
            "foo4", "bar4", "qux4",
            };

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
                .setHeader("partitionKey", value)
                .build();
    }

}
```

application.yml。

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: partitioned.topic
          producer:
            partitioned: true
            partition-key-expression: headers['partitionKey']
            partition-count: 12
```

Important

该主题必须配置为拥有足够的分区才能为所有消费者组实现所需的并发性。 上述配置支持多达12个消费者实例（如果它们的`concurrency`是2,4，则它们是6，如果它们的并发性是3，则依此类推）。 一般来说，最好是“重新提供”分区，以便将来增加消费者或并发性。

上述配置使用默认分区（`key.hashCode() % partitionCount`）。 这可能会也可能不会提供适当平衡的算法，具体取决于关键值。 您可以使用`partitionSelectorExpression`或`partitionSelectorClass`属性覆盖此默认值。

由于分区本质上是由Kafka处理的，因此消费者不需要特殊的配置。 Kafka在实例中分配分区。

以下Spring Boot应用程序侦听Kafka流并向控制台输出每条消息的分区ID：

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class KafkaPartitionConsumerApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
        System.out.println(in + " received from partition " + partition);
    }

}
```

application.yml。

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: partitioned.topic
          group: myGroup
```

您可以根据需要添加实例。 卡夫卡重新平衡分区分配。 如果实例数（或`instance count * concurrency`）超过分区数，则某些消费者闲置。

38. Apache Kafka Streams Binder译：38. Apache Kafka Streams活页夹

38.1 Usage译：38.1用法

要使用Kafka Streams活页夹，只需使用以下Maven坐标将其添加到Spring Cloud Stream应用程序：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka-streams</artifactId>
</dependency>
```

38.2 Kafka Streams Binder Overview 译：38.2 Kafka Streams活页夹概述

Spring Cloud Stream的Apache Kafka支持还包含一个为Apache Kafka Streams绑定明确设计的绑定器实现。 通过这种本地集成，Spring云流“处理器”应用程序可以直接在核心业务逻辑中使用Apache Kafka Streams API。

Kafka Streams活页夹实施建立在 Kafka Streams in Spring Kafka项目提供的基础上。

作为此本地集成的一部分，Kafka Streams API提供的高级 Streams DSL也可用于业务逻辑。

Processor API支持的早期版本也可用。

如前所述，Spring Cloud Stream中的Kafka Streams支持严格地仅适用于处理器模型。 可以应用从入站主题读取消息的业务处理模型，并且可以将转换后的消息写入出站主题。 它也可用于带有无出站目的地的处理器应用程序。

38.2.1 Streams DSL 译：38.2.1流DSL

此应用程序消耗来自Kafka主题（例如， `words` ）的数据，计算5秒时间窗口中每个唯一字的字数，并将计算结果发送到下游主题（例如， `counts` ）以供进一步处理。

```
@SpringBootApplication
@EnableBinding(KStreamProcessor.class)
public class WordCountProcessorApplication {

 @StreamListener("input")
 @SendTo("output")
 public KStream<?, WordCount> process(KStream<?, String> input) {
  return input
                .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
                .groupBy((key, value) -> value)
                .windowedBy(TimeWindows.of(5000))
                .count(Materialized.as("WordCounts-multi"))
                .toStream()
                .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()
    }

 public static void main(String[] args) {
  SpringApplication.run(WordCountProcessorApplication.class, args);
 }
```

一旦构建为超级罐（例如， `wordcount-processor.jar` ），您可以运行上面的示例，如下所示。

```
java -jar wordcount-processor.jar  --spring.cloud.stream.bindings.input.destination=words --spring.cloud.stream.bindings.output.destination=counts
```

此应用程序将使用来自卡夫卡主题 `words` 消息，并将计算结果发布到输出主题 `counts` 。

Spring Cloud Stream将确保来自传入和传出主题的消息自动绑定为KStream对象。 作为开发人员，您可以专注于代码的业务方面，即编写处理器中所需的逻辑。 设置Streams Kafka Streams基础架构所需的DSL特定配置由框架自动处理。

38.3 Configuration Options 译：38.3配置选项

本节包含Kafka Streams活页夹使用的配置选项。

有关活页夹的常用配置选项和属性，请参阅 core documentation 。

38.3.1 Kafka Streams Properties 译：38.3.1 Kafka流属性

以下属性在资料夹级别可用，且必须以 `spring.cloud.stream.kafka.binder.` 文字作为前缀。

configuration

Map with a key/value pair containing properties pertaining to Apache Kafka Streams API. This property must be prefixed with `spring.cloud.stream.kafka.streams.binder.`. Following are some examples of using this property.

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde=org.apache.kafka.common.serialization.Serdes$StringSerde
spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde=org.apache.kafka.common.serialization.Serdes$StringSerde
spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms=1000
```

有关可能进入流配置的所有属性的更多信息，请参阅Apache Kafka Streams文档中的StreamsConfig JavaDocs。

brokers

经纪人网址

默认： `localhost`

zkNodes

Zookeeper URL

默认： `localhost`

serdeError

反序列化错误处理程序类型。 可能的值是- `logAndContinue` ， `logAndFail` 或者 `sendToDlq`

默认： `logAndFail`

applicationId

当前应用程序上下文中所有流配置的应用程序ID。 您可以使用绑定上的 `group` 属性覆盖单个 `StreamListener` 方法的应用程序ID。 在相同方法的多个输入的情况

下，您必须确保对所有输入绑定使用相同的组名。

默认： `default`

以下属性 *仅*适用于Kafka Streams生产者，并且必须以 `spring.cloud.stream.kafka.streams.bindings.<binding name>.producer.` 文字作为前缀。

keySerde

要使用的关键serde

默认： `none` 。

valueSerde

值serde使用

默认： `none` 。

useNativeEncoding

标志来启用本机编码

默认： `false` 。

以下属性 *仅*适用于Kafka Streams使用者，并且必须以 `spring.cloud.stream.kafka.streams.bindings.<binding name>.consumer.` 字面为前缀。

keySerde

要使用的关键serde

默认： `none` 。

valueSerde

值serde使用

默认： `none` 。

materializedAs

当使用传入的KTable类型时，状态存储将实现

默认： `none` 。

useNativeDecoding

标志以启用本地解码

默认： `false` 。

dlqName

DLQ主题名称。

默认： `none` 。

38.3.2 TimeWindow properties: 译：38.3.2 TimeWindow属性：

窗口化是流处理应用中的一个重要概念。 以下属性可用于配置时间窗计算。

spring.cloud.stream.kafka.streams.timeWindow.length

当给定此属性时，可以将 `TimeWindows` 自动装入应用程序中。 该值以毫秒表示。

默认： `none` 。

spring.cloud.stream.kafka.streams.timeWindow.advanceBy

值以毫秒为单位给出。

默认： `none` 。

38.4 Multiple Input Bindings 译：38.4多输入绑定

对于需要多个传入KStream对象或KStream和KTable对象组合的用例，Kafka Streams资料夹提供了多个绑定支持。

让我们看看它的行动。

38.4.1 Multiple Input Bindings as a Sink 译：38.4.1多输入绑定作为接收器

```
@EnableBinding(KStreamKTableBinding.class)
.....
.....
@StreamListener
public void process(@Input("inputStream") KStream<String, PlayEvent> playEvents,
                    @Input("inputTable") KTable<Long, Song> songTable) {
                    ....
                    ....
}

interface KStreamKTableBinding {

    @Input("inputStream")
    KStream<?, ?> inputStream();

    @Input("inputTable")
    KTable<?, ?> inputTable();
}
```

在上面的例子中，应用程序被编写为接收器，即没有输出绑定，应用程序必须决定是否需要下游处理。 当您以这种风格编写应用程序时，您可能需要向下游发送信息或将它们存储在状态存储中（请参阅下面的“可查询状态存储”）。

在传入KTable的情况下，如果要将计算实现到状态存储库，则必须通过以下属性表示它。

```
spring.cloud.stream.kafka.streams.bindings.inputTable.consumer.materializedAs: all-songs
```

38.4.2 Multiple Input Bindings as a Processor 译：38.4.2多输入绑定作为处理器

```
@EnableBinding(KStreamKTableBinding.class)
....
....

@StreamListener
@SendTo("output")
public KStream<String, Long> process(@Input("input") KStream<String, Long> userClicksStream,
                                     @Input("inputTable") KTable<String, String> userRegionsTable) {
....
....
}

interface KStreamKTableBinding extends KafkaStreamsProcessor {

    @Input("inputX")
    KTable<?, ?> inputTable();
}
```

38.5 Multiple Output Bindings (aka Branching) 译：38.5多输出绑定（又名分支）

Kafka Streams允许出站数据根据一些谓词分成多个主题。 Kafka Streams活页夹为此功能提供支持，而不会影响最终用户应用程序中通过`StreamListener`公开的编程模型。

您可以按照上面在单词计数示例中演示的通常方式编写应用程序。 但是，使用分支功能时，您需要做一些事情。 首先，您需要确保您的退货类型为`KStream[]`而不是普通的`KStream` 。 其次，您需要使用包含输出绑定顺序的`SendTo`注释（请参见下面的示例）。 对于每个输出绑定，您需要配置目标，内容类型等，以符合标准的Spring Cloud Stream期望。

这里是一个例子：

```
@EnableBinding(KStreamProcessorWithBranches.class)
@EnableAutoConfiguration
public static class WordCountProcessorApplication {

    @Autowired
    private TimeWindows timeWindows;

    @StreamListener("input")
    @SendTo({"output1","output2","output3"})
    public KStream<?, WordCount>[] process(KStream<Object, String> input) {

   Predicate<Object, WordCount> isEnglish = (k, v) -> v.word.equals("english");
   Predicate<Object, WordCount> isFrench =  (k, v) -> v.word.equals("french");
   Predicate<Object, WordCount> isSpanish = (k, v) -> v.word.equals("spanish");

   return input
     .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
     .groupBy((key, value) -> value)
     .windowedBy(timeWindows)
     .count(Materialized.as("WordCounts-1"))
     .toStream()
     .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()))))
     .branch(isEnglish, isFrench, isSpanish);
    }

    interface KStreamProcessorWithBranches {

      @Input("input")
      KStream<?, ?> input();

      @Output("output1")
      KStream<?, ?> output1();

      @Output("output2")
      KStream<?, ?> output2();

      @Output("output3")
      KStream<?, ?> output3();
     }
}
```

属性：

```
spring.cloud.stream.bindings.output1.contentType: application/json
spring.cloud.stream.bindings.output2.contentType: application/json
spring.cloud.stream.bindings.output3.contentType: application/json
spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms: 1000
spring.cloud.stream.kafka.streams.binder.configuration:
  default.key.serde: org.apache.kafka.common.serialization.Serdes$StringSerde
  default.value.serde: org.apache.kafka.common.serialization.Serdes$StringSerde
spring.cloud.stream.bindings.output1:
  destination: foo
  producer:
    headerMode: raw
spring.cloud.stream.bindings.output2:
  destination: bar
  producer:
    headerMode: raw
spring.cloud.stream.bindings.output3:
  destination: fox
  producer:
    headerMode: raw
spring.cloud.stream.bindings.input:
  destination: words
  consumer:
    headerMode: raw
```

38.6 Message Conversion译：38.6消息转换

与基于消息通道的活页夹应用程序类似，Kafka Streams活页夹可以适应开箱即用的内容类型转换，而不会有任何折衷。

Kafka Streams操作通常会知道SerDe用于正确转换密钥和值的类型。 因此，依靠Apache Kafka Streams库本身提供的入站和出站转换的SerDe工具而不是使用框架提供的内容类型转换可能更自然。 另一方面，您可能已经熟悉框架提供的内容类型转换模式，并且您希望继续使用入站和出站转换。

这两个选项在Kafka Streams活页夹实现中都受支持。

38.6.1 Outbound serialization译：38.6.1出站序列化

如果禁用了本地编码（这是默认设置），那么框架将使用用户设置的contentType转换消息（否则，将应用默认的`application/json`）。 在这种情况下，它将忽略出站序列中的任何SerDe集。

这是在出站设置contentType的属性。

```
spring.cloud.stream.bindings.output.contentType: application/json
```

这是启用本机编码的属性。

```
spring.cloud.stream.bindings.output.nativeEncoding: true
```

如果在输出绑定上启用了本机编码（用户必须如上所述启用它），那么框架将跳过任何形式的出站自动消息转换。 在这种情况下，它将切换到用户设置的Serde。 将使用在实际输出绑定上设置的`valueSerde`属性。 这是一个例子。

```
spring.cloud.stream.kafka.streams.bindings.output.producer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde
```

如果此属性未设置，则它将使用“默认”SerDe： `spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde`。

值得一提的是，Kafka Streams活页夹不会在出站时序列化密钥 - 它仅仅依赖于Kafka本身。 因此，您必须在绑定上指定`keySerde`属性，否则它将默认为应用程序范围的通用`keySerde`。

绑定级别键serde：

```
spring.cloud.stream.kafka.streams.bindings.output.producer.keySerde
```

公用钥匙serde：

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde
```

如果使用分支，那么您需要使用多个输出绑定。 例如，

```
interface KStreamProcessorWithBranches {

      @Input("input")
      KStream<?, ?> input();

      @Output("output1")
      KStream<?, ?> output1();

      @Output("output2")
      KStream<?, ?> output2();

      @Output("output3")
      KStream<?, ?> output3();
     }
```

如果设置了`nativeEncoding`，那么您可以在各个输出绑定上设置不同的SerDe，如下所示。

```
spring.cloud.stream.kafka.streams.bindings.output1.producer.valueSerde=IntegerSerde
spring.cloud.stream.kafka.streams.bindings.output2.producer.valueSerde=StringSerde
spring.cloud.stream.kafka.streams.bindings.output3.producer.valueSerde=JsonSerde
```

然后，如果你有`SendTo`这样，@SendTo（{“输出1”，“输出2”，“输出3”}），则`KStream[]`从树枝与适当SERDE对象施加如上所定义。 如果您未启用`nativeEncoding`，则可以在输出绑定上设置不同的contentType值，如下所示。 在这种情况下，框架将在发送给Kafka之前使用适当的消息转换器来转换消息。

```
spring.cloud.stream.bindings.output1.contentType: application/json
spring.cloud.stream.bindings.output2.contentType: application/java-serialzied-object
spring.cloud.stream.bindings.output3.contentType: application/octet-stream
```

38.6.2 Inbound Deserialization译：38.6.2入站反序列化

类似规则适用于入站数据反序列化。

如果禁用本地解码（这是默认设置），则框架将使用用户设置的contentType转换消息（否则，将应用默认的`application/json`）。 在这种情况下，它将忽略任何SerDe设置的入站反序列化。

这是在入站设置contentType的属性。

```
spring.cloud.stream.bindings.input.contentType: application/json
```

这是启用本机解码的属性。

```
spring.cloud.stream.bindings.input.nativeDecoding: true
```

如果在输入绑定上启用了本机解码（用户必须如上所述启用它），那么框架将跳过对入站的任何消息转换。 在这种情况下，它将切换到由用户设置的SerDe。 将使用实际输出绑定上设置的`valueSerde`属性。 这是一个例子。

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde
```

如果这个属性没有设置，它将使用默认的SerDe： `spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde`。

值得一提的是，Kafka Streams活页夹不会反序列化入站键 - 它仅仅依赖于Kafka本身。 因此，您必须在绑定上指定`keySerde`属性，否则它将默认为应用程序范围的通用`keySerde`。

绑定级别键serde：

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.keySerde
```

公用钥匙serde：

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde
```

就像KStream在出站分支的情况一样，每个绑定设置值SerDe的好处是，如果您有多个输入绑定（多个KStreams对象）并且它们都需要单独的值SerDe，那么您可以单独配置它们。 如果您使用通用配置方法，则此功能不适用。

38.7 Error Handling译：38.7错误处理

Apache Kafka Streams提供了从本地处理反序列化错误异常的功能。 有关此支持的详细信息，请参阅this Apache Kafka Streams提供了两种反序列化异常处理程序`logAndContinue`和`logAndFail`。 正如名称所示，前者将记录错误并继续处理下一个记录，后者将记录错误并失败。 `LogAndFail`是默认的反序列化异常处理程序。

38.7.1 Handling Deserialization Exceptions译：38.7.1处理反序列化异常

Kafka Streams活页夹通过以下属性支持一系列异常处理程序。

```
spring.cloud.stream.kafka.streams.binder.serdeError: logAndContinue
```

除了上述两个反序列化异常处理程序之外，活页夹还提供第三个用于将错误记录（毒药）发送到DLQ主题。 这是你如何启用这个DLQ异常处理程序。

```
spring.cloud.stream.kafka.streams.binder.serdeError: sendToDlq
```

当上述属性被设置时，所有的反序列化错误记录都会自动发送到DLQ主题。

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.dlqName: foo-dlq
```

如果已设置，则错误记录将发送到主题`foo-dlq`。 如果没有设置，那么它将创建名称为`error.<input-topic-name>.<group-name>`的DLQ主题。

在Kafka Streams活页夹中使用异常处理功能时需牢记几件事。

- The property `spring.cloud.stream.kafka.streams.binder.serdeError` is applicable for the entire application. This implies that if there are multiple `StreamListener` methods in the same application, this property is applied to all of them.
- The exception handling for deserialization works consistently with native deserialization and framework provided message conversion.

38.7.2 Handling Non-Deserialization Exceptions译：38.7.2处理非反序列化例外

对于Kafka Streams活页夹中的一般错误处理，最终用户应用程序可以处理应用程序级错误。 作为为反序列化异常处理程序提供DLQ的副作用，Kafka Streams binder提供了一种直接从应用程序访问DLQ发送Bean的方法。 一旦你获得了对这个bean的访问权限，你就可以通过编程的方式将你应用程序中的任何异常记录发送到DLQ。

使用高级DSL保持错误处理的能力仍然很难; Kafka Streams本身不支持错误处理。

但是，当您在应用程序中使用低级别的Processor API时，可以选择控制此行为。 见下文。

```
@Autowired
private SendToDlqAndContinue dlqHandler;

@StreamListener("input")
@SendTo("output")
public KStream<?, WordCount> process(KStream<Object, String> input) {

    input.process(() -> new Processor() {
       ProcessorContext context;

       @Override
       public void init(ProcessorContext context) {
        this.context = context;
       }

       @Override
       public void process(Object o, Object o2) {

           try {
               .....
               .....
           }
           catch(Exception e) {
               //explicitly provide the kafka topic corresponding to the input binding as the first argument.
                       //DLQ handler will correctly map to the dlq topic from the actual incoming destination.
                       dlqHandler.sendToDlq("topic-name", (byte[]) o1, (byte[]) o2, context.partition());
           }
       }

       .....
       .....
    });
}
```

38.8 Interactive Queries译: 38.8交互式查询

作为公共Kafka Streams活页夹API的一部分，我们公开了一个名为`QueryableStoreRegistry`的类。 您可以在应用程序中以Spring bean的身份访问它。 从应用程序访问此bean的简单方法是在应用程序中“自动”装入bean。

```
@Autowired
private QueryableStoreRegistry queryableStoreRegistry;
```

一旦你获得了这个bean的访问权限，那么你可以查询你感兴趣的特定状态存储。 见下文。

```
ReadOnlyKeyValueStore<Object, Object> keyValueStore =
      queryableStoreRegistry.getQueryableStoreType("my-store", QueryableStoreTypes.keyValueStore());
```

39. RabbitMQ Binder译: 39. RabbitMQ活页夹

39.1 Usage译: 39.1用法

要使用RabbitMQ联编程序，可以使用以下Maven坐标将它添加到Spring Cloud Stream应用程序中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

或者，您可以使用Spring Cloud Stream RabbitMQ Starter，如下所示：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

39.2 RabbitMQ Binder Overview译: 39.2 RabbitMQ活页夹概述

以下简图显示了RabbitMQ活页夹的工作方式：

图39.1。 RabbitMQ活页夹

默认情况下，RabbitMQ活页夹实现将每个目标映射到`TopicExchange`。 对于每个消费群体， `Queue`都绑定到`TopicExchange`。 每个消费者实例都有一个相应的RabbitMQ `Consumer`实例用于其组`Queue`。 对于分区生产者和消费者，队列以分区索引作为后缀，并使用分区索引作为路由键。 对于匿名消费者（没有`group`属性的消费者），使用自动删除队列（具有随机唯一名称）。

通过使用可选的`autoBindDlq`选项，可以配置活页夹以创建和配置死信队列（DLQ）（以及死信交换`DLX`以及路由基础结构）。 默认情况下，死信队列具有目的地的名称，并附有`.dlq`。 如果重试已启用（ `maxAttempts > 1` ），则在重试耗尽后，将失败的消息发送到DLQ。 如果禁用重试（ `maxAttempts = 1` ），则应将`requeueRejected`设置为`false` （默认值），以便将失败的消息路由到DLQ，而不是重新排队。 另外， `republishToDlq`会导致活页夹将失败的消息发布到DLQ（而不是拒绝它）。 通过此功能，可以将额外信息（如`x-exception-stacktrace`标题中的堆栈跟踪）添加到标题中的消息中。 该选项不需要重试启用。 只需一次尝试即可重新发布失败的消息。 从版本1.2开始，您可以配置重新发布的消息的传送模式。 请参阅`republishDeliveryMode` property。

Important

将`requeueRejected`设置为`true` （包含`republishToDlq=false` ）会导致邮件重新排队并不断重新递送，这可能不是您想要的，除非失败的原因是暂时的。 在一般情况下，你应该能够通过设置在粘合剂中的重试`maxAttempts`大于一个或通过设置`republishToDlq`至`true`。

有关这些属性的更多信息，请参阅 Section 39.3.1, “RabbitMQ Binder Properties” 。

该框架没有提供任何标准的机制来使用死信消息（或将它们重新路由回主要队列）。 有些选项在Section 39.6, “Dead-Letter Queue Processing”中描述。

当在Spring Cloud Stream应用程序中使用多个RabbitMQ绑定程序时，禁用“RabbitAutoConfiguration”非常重要，以避免将`RabbitAutoConfiguration`的相同配置应用于这两个绑定程序。 您可以使用`@SpringBootApplication`注释排除该类。

从2.0版开始，该`RabbitMessageChannelBinder`设置`RabbitTemplate.userPublisherConnection`属性为`true`，使非事务生产者避免对消费者死锁，如果缓存的连接被阻止，因为的它可以发生 memory alarm的经纪人。

39.3 Configuration Options译：39.3配置选项

本部分包含特定于RabbitMQ活页夹和绑定通道的设置。

有关一般绑定配置选项和属性，请参阅 Spring Cloud Stream core documentation 。

39.3.1 RabbitMQ Binder Properties译：39.3.1 RabbitMQ活页夹属性

默认情况下，RabbitMQ联编程序使用Spring Boot的`ConnectionFactory`。 相反，它支持RabbitMQ的所有Spring Boot配置选项。 （仅供参考，请参阅Spring Boot documentation ）。 RabbitMQ配置选项使用`spring.rabbitmq`前缀。

除了Spring Boot选项外，RabbitMQ活页夹还支持以下属性：

spring.cloud.stream.rabbit.binder.adminAddresses

RabbitMQ管理插件URL的逗号分隔列表。 仅当`nodes`包含多个条目时才使用。 此列表中的每个条目都必须在`spring.rabbitmq.addresses`具有相应的条目。 只有在使用RabbitMQ集群并希望从承载队列的节点使用时才需要。 有关更多信息，请参阅Queue Affinity and the LocalizedQueueConnectionFactory 。

默认值：空。

spring.cloud.stream.rabbit.binder.nodes

RabbitMQ节点名称的逗号分隔列表。 当有多个条目时，用于查找队列所在的服务器地址。 此列表中的每个条目都必须在`spring.rabbitmq.addresses`有相应的条目。 只有在使用RabbitMQ集群并希望从承载队列的节点使用时才需要。 有关更多信息，请参阅Queue Affinity and the LocalizedQueueConnectionFactory 。

默认值：空。

spring.cloud.stream.rabbit.binder.compressionLevel

压缩绑定的压缩级别。 见`java.util.zip.Deflater`。

默认：`1`（BEST_LEVEL）。

spring.cloud.stream.binder.connection-name-prefix

用于命名由此活页夹创建的连接的连接名称前缀。 名称是此前缀后跟`#n`，其中`n`每次打开新连接时`n`增加。

默认值：无（Spring AMQP默认）。

39.3.2 RabbitMQ Consumer Properties译：39.3.2 RabbitMQ用户属性

以下属性仅供Rabbit消费者使用，且必须以`spring.cloud.stream.rabbit.bindings.<channelName>.consumer.`为前缀。

acknowledgeMode

确认模式。

默认：`AUTO`。

autoBindDlq

是否自动声明DLQ并将其绑定到绑定器DLX。

默认：`false`。

bindingRoutingKey

用于将队列绑定到交换机的路由密钥（如果`bindQueue`为`true`）。 对于已分区的目标，将追加`-<instanceIndex>`。

默认：`#`。

bindQueue

是否将队列绑定到目标交换机。 如果您已经建立了自己的基础架构并且先前已经创建并绑定了队列，`false`其设置为`false`。

默认：`true`。

deadLetterQueueName

DLQ的名称

默认：`prefix+destination.dlq`

deadLetterExchange

要分配给队列的DLX。 仅在`autoBindDlq`为`true`。

默认值：'前缀+ DLX'

deadLetterRoutingKey

要分配给队列的死信路由密钥。 仅在`autoBindDlq`为`true`。

默认：`destination`

declareExchange

是否宣布交换目的地。

默认： `true` 。

delayedExchange

是否宣布交易所为`Delayed Message Exchange` 。 需要代理上延迟的消息交换插件。 `x-delayed-type`参数设置为`exchangeType` 。

默认： `false` 。

dlqDeadLetterExchange

如果声明了DLQ，则将DLX分配给该队列。

默认： `none`

dlqDeadLetterRoutingKey

如果声明了DLQ，则将分配给该队列的死信路由密钥。

默认： `none`

dlqExpires

在未使用的死信队列被删除之前多长时间（以毫秒为单位）。

默认： `no expiration`

dlqLazy

用`x-queue-mode=lazy`参数声明死信队列。 见“Lazy Queues” 。 考虑使用策略而不是此设置，因为使用策略可以在不删除队列的情况下更改设置。

默认： `false` 。

dlqMaxLength

死信队列中的最大消息数。

默认： `no limit`

dlqMaxLengthBytes

所有消息的死信队列中的最大字节总数。

默认： `no limit`

dlqMaxPriority

死信队列中消息的最大优先级（0-255）。

默认： `none`

dlqTtl

声明时（默认值为毫秒）的默认生存时间以应用于死信队列。

默认： `no limit`

durableSubscription

订阅是否应该持久。 只有在`group`也有效时才有效。

默认： `true` 。

exchangeAutoDelete

如果 `declareExchange`为真，交换是否应该被自动删除（即在删除最后一个队列后删除）。

默认： `true` 。

exchangeDurable

如果 `declareExchange`属实，交易所是否应该是持久的（也就是说，它在经纪商重新启动后仍然存在）。

默认： `true` 。

exchangeType

交换式： `direct` ， `fanout`或者 `topic`非分区目的地和 `direct`或者 `topic`的分区目的地。

默认： `topic` 。

exclusive

是否创建独家消费者。 如果这是`true`并发应为1。 通常在需要严格排序时使用，但在故障发生后启用热备份实例。 请参阅`recoveryInterval` ，它控制备用实例尝试使用的频率。

默认： `false` 。

expires

在未使用的队列被删除之前多长时间（以毫秒为单位）。

默认： `no expiration`

failedDeclarationRetryInterval

从队列中消失的尝试（如果缺失）之间的间隔（以毫秒为单位）。

默认值：5000

headerPatterns

从入站消息映射标题的模式。

默认：`['*']`（全部标题）。

lazy

用`x-queue-mode=lazy`参数声明队列。 见“Lazy Queues”。 考虑使用策略而不是此设置，因为使用策略可以在不删除队列的情况下更改设置。

默认：`false`。

maxConcurrency

消费者的最大数量。

默认：`1`。

maxLength

队列中消息的最大数量。

默认：`no limit`

maxLengthBytes

来自所有消息的队列中的最大字节总数。

默认：`no limit`

maxPriority

队列中消息的最大优先级（0-255）。

默认：`none`

missingQueuesFatal

当找不到队列时，是否将该条件视为致命并停止侦听器容器。 默认为`false`以便容器不断尝试从队列中消耗 - 例如，使用集群时，托管非HA队列的节点关闭。

默认：`false`

prefetch

预取计数。

默认：`1`。

prefix

要添加到`destination`和队列名称的前缀。

默认：“”。

queueDeclarationRetries

如果缺少队列重试消耗的次数。 只有当`missingQueuesFatal`是`true`。 否则，容器将不停地重试。

默认：`3`

queueNameGroupOnly

如果为true，则从队列中消耗名称等于`group`的队列。 否则，队列名称是`destination.group`。 例如，在使用Spring Cloud Stream从现有的RabbitMQ队列中使用时，这很有用。

默认值：false。

recoveryInterval

连接恢复尝试之间的间隔，以毫秒为单位。

默认：`5000`。

requeueRejected

重试禁用时应重新排队传递失败还是`republishToDlq`为`false`。

默认：`false`。

republishDeliveryMode

当`republishToDlq`为`true`，指定重新发布的消息的传递模式。

默认：`DeliveryMode.PERSISTENT`

republishToDlq

默认情况下，拒绝重试后失败的消息。 如果配置了死信队列（DLQ），则RabbitMQ将失败的消息（未更改）路由到DLQ。 如果设置为`true`，则绑定器会使用其他标头向DLQ重新发布失败消息，包括异常消息和最终失败原因的堆栈跟踪。

默认值：false

transacted

是否使用交易渠道。

默认：`false`。

ttl

声明时的默认活动时间（以毫秒为单位）。

默认：`no limit`

txSize

Ack之间交付的数量。

默认值： `1` 。

39.3.3 Rabbit Producer Properties译：39.3.3兔子生产者属性

以下属性仅供兔生产者使用，且必须以 `spring.cloud.stream.rabbit.bindings.<channelName>.producer.` 为前缀。

autoBindDlq

是否自动声明DLQ并将其绑定到绑定器DLX。

默认： `false` 。

batchingEnabled

是否启用生产者的消息批处理。 根据以下属性（在此列表中接下来的三个条目中描述）将消息批量化为一条消息：'batchSize'， `batchBufferLimit` 和 `batchTimeout` 。 有关更多信息，请参阅Batching 。

默认： `false` 。

batchSize

启用批处理时要缓冲的消息数量。

默认： `100` 。

batchBufferLimit

批处理启用时的最大缓冲区大小。

默认： `10000` 。

batchTimeout

批处理启用时的批处理超时。

默认： `5000` 。

bindingRoutingKey

用于将队列绑定到交换机的路由密钥（如果 `bindQueue` 是 `true` ）。 仅适用于未分区的目标。 仅在提供 `requiredGroups` 适用，然后仅适用于这些组。

默认： `#` 。

bindQueue

是否将队列绑定到目标交换机。 如果您已经建立了自己的基础架构并且之前创建并绑定了队列， `false` 其设置为 `false` 。 仅在提供 `requiredGroups` 适用， `requiredGroups` 仅适用于这些组。

默认： `true` 。

compress

数据是否应在发送时进行压缩。

默认： `false` 。

deadLetterQueueName

仅在提供 `requiredGroups` 且仅适用于这些组时，才应用DLQ的名称。

默认： `prefix+destination.dlq`

deadLetterExchange

要分配给队列的DLX。 只有当 `autoBindDlq` 是 `true` 。 仅在提供 `requiredGroups` 时适用，仅适用于这些组。

默认值：'前缀+ DLX'

deadLetterRoutingKey

要分配给队列的死信路由密钥。 只有当 `autoBindDlq` 是 `true` 。 仅在提供 `requiredGroups` 时适用，然后仅适用于这些组。

默认： `destination`

declareExchange

是否宣布交换目的地。

默认： `true` 。

delayExpression

SpEL表达式，用于评估应用于消息的延迟（ `x-delay` 标头）。 如果交换不是延迟的消息交换，则不起作用。

默认：否设置 `x-delay` 标题。

delayedExchange

是否宣布交易所为 `Delayed Message Exchange` 。 需要代理上延迟的消息交换插件。 `x-delayed-type` 参数设置为 `exchangeType` 。

默认： `false` 。

deliveryMode

交付模式。

默认： `PERSISTENT` 。

dlqDeadLetterExchange

声明DLQ时，将分配给该队列的DLX。 仅在提供 `requiredGroups` 且仅适用于这些组时适用。

默认： `none`

dlqDeadLetterRoutingKey

在声明DLQ时，将分配给该队列的死信路由密钥。 仅在提供`requiredGroups`时适用，仅适用于这些组。

默认： `none`

dlqExpires

在未使用的死信队列被删除之前多久（以毫秒为单位）。 仅在提供`requiredGroups`且仅适用于这些组时适用。

默认： `no expiration`

dlqLazy

Declare the dead letter queue with the `x-queue-mode=lazy` argument. See “Lazy Queues”. Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue. Applies only when `requiredGroups` are provided and then only to those groups.

dlqMaxLength

死信队列中的最大消息数。 仅在提供`requiredGroups`且仅适用于这些组时适用。

默认： `no limit`

dlqMaxLengthBytes

所有消息的死信队列中的最大字节总数。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认： `no limit`

dlqMaxPriority

死信队列中消息的最大优先级（0-255）仅在提供 `requiredGroups`时应用，然后仅应用于这些组。

默认： `none`

dlqTtl

默认时间（以毫秒为单位）在声明时适用于死信队列。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认： `no limit`

exchangeAutoDelete

如果`declareExchange`是`true`，那么交换机是否应该是自动删除（在最后一个队列被删除后它会被删除）。

默认： `true` 。

exchangeDurable

如果`declareExchange`是`true`，交易所是否应该持久（保持经纪人重启）。

默认： `true` 。

exchangeType

交换式： `direct`， `fanout`或者 `topic`非分区目的地和 `direct`或者 `topic`的分区目的地。

默认： `topic` 。

expires

在未使用的队列被删除之前多久（以毫秒为单位）。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认： `no expiration`

headerPatterns

标题的模式被映射到出站消息。

默认： `['*']` （全部标题）。

lazy

用`x-queue-mode=lazy`参数声明队列。 见“Lazy Queues”。 考虑使用策略而不是此设置，因为使用策略可以在不删除队列的情况下更改设置。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认： `false` 。

maxLength

队列中的最大消息数。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认： `no limit`

maxLengthBytes

所有消息的队列中总字节数的最大值。 仅在提供`requiredGroups`适用，然后仅适用于这些组。

默认： `no limit`

maxPriority

队列中消息的最大优先级（0-255）。 仅在提供`requiredGroups`适用，仅适用于这些组。

默认： `none`

prefix

要添加到 `destination`交换名称的前缀。

默认：“”。

queueNameGroupOnly

`true`，从名称等于`group`的队列中消耗。 否则，队列名称是`destination.group`。 例如，在使用Spring Cloud Stream从现有的RabbitMQ队列中使用时，这很有用。 仅在提供`requiredGroups`时适用，然后仅适用于这些组。

默认值：false。

routingKeyExpression

SpEL表达式，用于确定发布消息时使用的路由密钥。 对于一个固定的路由密钥，使用文字表达，例如`routingKeyExpression='my.routingKey'`在属性文件中或`routingKeyExpression: '''my.routingKey'''`在YAML文件。

默认值： `destination`或`destination-<partition>`分区目标。

transacted

是否使用交易渠道。

默认： `false`。

ttl

默认时间（以毫秒为单位）在申报时适用于队列。 仅在提供`requiredGroups`时适用，仅适用于这些组。

默认： `no limit`

对于RabbitMQ，内容类型标题可以由外部应用程序设置。 Spring Cloud Stream支持它们作为扩展内部协议的一部分，用于任何类型的传输，包括传输，如Kafka（0.11之前），本身不支持标头。

39.4 Retry With the RabbitMQ Binder译：39.4使用RabbitMQ活页夹重试

在活页夹中启用重试时，侦听器容器线程将暂停所有配置的回退期。 如果单个消费者需要严格的订购，这可能很重要。 但是，对于其他用例，它会阻止在该线程上处理其他消息。 使用活页夹重试的替代方法是在死信队列（DLQ）上设置带死时间的死文字以及DLQ本身的死信配置。 有关此处讨论的属性的更多信息，请参阅“ Section 39.3.1, “RabbitMQ Binder Properties”” 。 您可以使用以下示例配置来启用此功能：

- Set `autoBindDlq` to `true`. The binder create a DLQ. Optionally, you can specify a name in `deadLetterQueueName`.
- Set `dlqTtl` to the back off time you want to wait between redeliveries.
- Set the `dlqDeadLetterExchange` to the default exchange. Expired messages from the DLQ are routed to the original queue, because the default `deadLetterRoutingKey` is the queue name (`destination.group`). Setting to the default exchange is achieved by setting the property with no value, as shown in the next example.

要强制一条消息是死的， `AmqpRejectAndDontRequeueException`或`requeueRejected`设置为`true`（默认值）并抛出任何异常。

循环没有结束，这对于瞬态问题是很好的，但是你可能想在经过一些尝试后放弃。 幸运的是，RabbitMQ提供了`x-death`标题，它可以让您确定发生了多少次循环。

在放弃之后承认一条消息，抛出`ImmediateAcknowledgeAmqpException`。

39.4.1 Putting it All Together译：39.4.1把它放在一起

以下配置会创建一个交换`myDestination`，并使用通配符路由密钥`#`队列`myDestination.consumerGroup`绑定到主题交换：

```
---
spring.cloud.stream.bindings.input.destination=myDestination
spring.cloud.stream.bindings.input.group=consumerGroup
#disable binder retries
spring.cloud.stream.bindings.input.consumer.max-attempts=1
#dlx/dlq setup
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=
---
```

此配置会创建一个绑定到直接交换（`DLX`）的DLQ，路由密钥为`myDestination.consumerGroup`。 当邮件被拒绝时，它们被路由到DLQ。 5秒后，消息将过期并通过使用队列名称作为路由密钥而路由到原始队列，如下例所示：

Spring Boot应用程序。

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class XDeathApplication {

    public static void main(String[] args) {
        SpringApplication.run(XDeathApplication.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(String in, @Header(name = "x-death", required = false) Map<?,?> death) {
        if (death != null && death.get("count").equals(3L)) {
            // giving up - don't send to DLX
            throw new ImmediateAcknowledgeAmqpException("Failed after 4 attempts");
        }
        throw new AmqpRejectAndDontRequeueException("failed");
    }

}
```

请注意， `x-death`标题中的count属性为`Long`。

39.5 Error Channels译：39.5错误通道

从版本1.3开始，活页夹无条件地将异常发送到每个使用者目标的错误通道，还可以配置为将异步生产者发送失败发送到错误通道。 请参阅“ ???”了解更多信息。

RabbitMQ有两种类型的发送失败：

- Returned messages,

- Negatively acknowledged Publisher Confirms.

后者很少见。 根据RabbitMQ文档，“只有在负责队列的Erlang进程中发生内部错误时才会发送[A nack]”。

除了启用生产者错误通道（如“ ???”中所述），如果连接工厂已正确配置，则RabbitMQ绑定器仅向通道发送消息，如下所示：

- `ccf.setPublisherConfirms(true);`
- `ccf.setPublisherReturns(true);`

对连接工厂使用Spring Boot配置时，请设置以下属性：

- `spring.rabbitmq.publisher-confirms`
- `spring.rabbitmq.publisher-returns`

返回消息的 `ErrorMessage` 的有效负载是 `ReturnedAmqpMessageException` 具有以下属性：

- `failedMessage`: The spring-messaging `Message<?>` that failed to be sent.
- `amqpMessage`: The raw spring-amqp `Message`.
- `replyCode`: An integer value indicating the reason for the failure (for example, 312 - No route).
- `replyText`: A text value indicating the reason for the failure (for example, `NO_ROUTE`).
- `exchange`: The exchange to which the message was published.
- `routingKey`: The routing key used when the message was published.

对于否定确认，有效负载为 `NackedAmqpMessageException` 具有以下属性：

- `failedMessage`: The spring-messaging `Message<?>` that failed to be sent.
- `nackReason`: A reason (if available — you may need to examine the broker logs for more information).

没有自动处理这些异常（例如发送到dead-letter queue ）。 您可以使用您自己的Spring Integration流程来使用这些异常。

39.6 Dead-Letter Queue Processing译：39.6死信队列处理

由于您无法预测用户如何处理死信消息，因此该框架不提供任何标准机制来处理它们。 如果死书的原因是短暂的，你可能希望将消息路由回原始队列。 但是，如果问题是一个永久性问题，那可能会导致无限循环。 以下Spring Boot应用程序显示了如何将这些消息路由回原始队列的示例，但在三次尝试后将它们移至第三个“停车场”队列。 第二个示例使用RabbitMQ Delayed Message Exchange向重新排队的消息引入延迟。 在这个例子中，每次尝试的延迟都会增加。 这些示例使用 `@RabbitListener` 来接收来自DLQ的消息。 您也可以在批处理中使用 `RabbitTemplate.receive()` 。

这些示例假定原始目的地是 `so8400in` ，而消费者组是 `so8400` 。

39.6.1 Non-Partitioned Destinations译：39.6.1未分区的目标

前两个示例用于目标 **未**分区的情况：

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Integer retriesHeader = (Integer) failedMessage.getMessageProperties().getHeaders().get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            failedMessage.getMessageProperties().getHeaders().put(X_RETRIES_HEADER, retriesHeader + 1);
            this.rabbitTemplate.send(ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    private static final String DELAY_EXCHANGE = "dlqReRouter";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            headers.put("x-delay", 5000 * retriesHeader);
            this.rabbitTemplate.send(DELAY_EXCHANGE, ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public DirectExchange delayExchange() {
        DirectExchange exchange = new DirectExchange(DELAY_EXCHANGE);
        exchange.setDelayed(true);
        return exchange;
    }

    @Bean
    public Binding bindOriginalToDelay() {
        return BindingBuilder.bind(new Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_QUEUE);
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

39.6.2 Partitioned Destinations 译：39.6.2分区目标

对于分区目标，所有分区都有一个DLQ。 我们从头文件中确定原始队列。

`republishToDlq=false` 译：**`republishToDlq=false`**

当 `republishToDlq` 为 `false` ，RabbitMQ将消息发布到DLX / DLQ，其中 `x-death` 标题包含有关原始目标的信息，如以下示例所示：

```
@SpringBootApplication
public class ReRouteDlqApplication {

 private static final String ORIGINAL_QUEUE = "so8400in.so8400";

 private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

 private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

 private static final String X_DEATH_HEADER = "x-death";

 private static final String X_RETRIES_HEADER = "x-retries";

 public static void main(String[] args) throws Exception {
  ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
  System.out.println("Hit enter to terminate");
  System.in.read();
  context.close();
 }

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @SuppressWarnings("unchecked")
 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X_DEATH_HEADER);
   String exchange = (String) xDeath.get(0).get("exchange");
   List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");
   this.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

 @Bean
 public Queue parkingLot() {
  return new Queue(PARKING_LOT);
 }

}
```

`republishToDlq=true`#:**`republishToDlq=true`**

`republishToDlq`为`true`，重新发布恢复程序会将原始交换和路由键添加到标头，如以下示例所示：

```
@SpringBootApplication
public class ReRouteDlqApplication {

 private static final String ORIGINAL_QUEUE = "so8400in.so8400";

 private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

 private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

 private static final String X_RETRIES_HEADER = "x-retries";

 private static final String X_ORIGINAL_EXCHANGE_HEADER = RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

 private static final String X_ORIGINAL_ROUTING_KEY_HEADER = RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

 public static void main(String[] args) throws Exception {
  ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
  System.out.println("Hit enter to terminate");
  System.in.read();
  context.close();
 }

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   String exchange = (String) headers.get(X_ORIGINAL_EXCHANGE_HEADER);
   String originalRoutingKey = (String) headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
   this.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

 @Bean
 public Queue parkingLot() {
  return new Queue(PARKING_LOT);
 }

}
```

39.7 Partitioning with the RabbitMQ Binder译：39.7使用 RabbitMQ活 页 夹 进 行 分 区

RabbitMQ不支持本地分区。

有时，将数据发送到特定分区是有利的，例如，当您想严格订购消息处理时，特定客户的所有消息应该转到同一个分区。

`RabbitMessageChannelBinder`通过将每个分区的队列绑定到目标交换机来提供分区。

以下Java和YAML示例显示了如何配置生产者：

制片人。

```
@SpringBootApplication
@EnableBinding(Source.class)
public class RabbitPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
            "abc1", "def1", "qux1",
            "abc2", "def2", "qux2",
            "abc3", "def3", "qux3",
            "abc4", "def4", "qux4",
            };

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
                .setHeader("partitionKey", value)
                .build();
    }

}
```

application.yml。

```
    spring:
      cloud:
        stream:
          bindings:
            output:
              destination: partitioned.destination
              producer:
                partitioned: true
                partition-key-expression: headers['partitionKey']
                partition-count: 2
                required-groups:
                - myGroup
```

前面示例中的配置使用默认分区（`key.hashCode() % partitionCount`）。 这可能会也可能不会提供适当平衡的算法，具体取决于关键值。 您可以使用`partitionSelectorExpression`或`partitionSelectorClass`属性覆盖此默认值。
只有在部署生产者时需要调配消费者队列时，才需要`required-groups`属性。 否则，任何发送到分区的消息都将丢失，直到部署相应的使用者。

以下配置规定了主题交换：

以下队列绑定到该交换机上：

以下绑定将队列与交换机相关联：

以下Java和YAML示例继续前面的示例，并演示如何配置使用者：

消费者。

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class RabbitPartitionConsumerApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(AmqpHeaders.CONSUMER_QUEUE) String queue) {
        System.out.println(in + " received from queue " + queue);
    }

}
```

application.yml。

```
    spring:
      cloud:
        stream:
          bindings:
            input:
              destination: partitioned.destination
              group: myGroup
              consumer:
                partitioned: true
                instance-index: 0
```

Important

`RabbitMessageChannelBinder`不支持动态缩放。 每个分区必须至少有一个用户。 消费者的`instanceIndex`用于指示哪个分区被消耗。 像Cloud Foundry这样的平台只能有一个`instanceIndex`实例。

Part VII. Spring Cloud Bus译:第七部分. 春云总线

Spring Cloud Bus将分布式系统的节点与轻量级消息代理链接起来。 该代理随后可用于广播状态更改（如配置更改）或其他管理说明。 一个关键的想法是，总线就像一个分布式执行器，用于Spring Boot应用程序的扩展。 但是，它也可以用作应用程序之间的通信渠道。 该项目为AMQP经纪人或Kafka作为交通工具提供了启动者。

Spring Cloud是在非限制性的Apache 2.0许可下发布的。 如果您想为文档的这一部分做出贡献，或者如果发现错误，请在github的项目中找到源代码和问题跟踪器。

40. Quick Start译:40.快速入门

如果在类路径中检测到自身，Spring Cloud总线的工作方式是添加Spring Boot autconfiguration。 要启用总线，请将`spring-cloud-starter-bus-amqp`或`spring-cloud-starter-bus-kafka`添加到您的依赖管理中。 Spring Cloud负责其余部分。 确保经纪人（RabbitMQ或Kafka）可用并已配置。 在localhost上运行时，你不需要做任何事情。 如果远程运行，请使用Spring Cloud连接器或Spring Boot约定来定义代理凭据，如以下Rabbit示例所示：

application.yml。

```
spring:
  rabbitmq:
    host: mybroker.com
    port: 5672
    username: user
    password: secret
```

总线当前支持向所有节点发送消息，或者监听某个特定服务的所有节点（由Eureka定义）。 `/bus/*`执行器命名空间有一些HTTP端点。 目前有两个实施。 第一个，`/bus/env` ，发送键/值对来更新每个节点的Spring环境。 第二个， `/bus/refresh`重新加载每个应用程序的配置，就好像它们全部被`/refresh`在`/refresh`端点上一样。

Spring Cloud Bus启动器涵盖了Rabbit和Kafka，因为这是两种最常见的实现。 但是，Spring Cloud Stream非常灵活，并且该活页夹可以与`spring-cloud-bus`一起`spring-cloud-bus` 。

41. Bus Endpoints译：41.公共汽车的终点

Spring Cloud Bus提供了两个端点， `/actuator/bus-refresh`和 `/actuator/bus-env` ，分别对应于Spring Cloud Commons中的各个执行器端点，`/actuator/refresh`和 `/actuator/env` 。

41.1 Bus Refresh Endpoint译：41.1总线刷新端点

`/actuator/bus-refresh`端点清除`RefreshScope`缓存并重新`@ConfigurationProperties` 。 有关更多信息，请参阅Refresh Scope文档。

要公开 `/actuator/bus-refresh`端点，您需要将以下配置添加到您的应用程序中：

```
management.endpoints.web.exposure.include=bus-refresh
```

41.2 Bus Env Endpoint译：41.2公交环境终点

`/actuator/bus-env`端点使用跨多个实例的指定键/值对更新每个实例环境。

要公开 `/actuator/bus-env`端点，您需要将以下配置添加到您的应用程序中：

```
management.endpoints.web.exposure.include=bus-env
```

`/actuator/bus-env`端点接受具有以下形状的 `POST`请求：

```
{
 "name": "key1",
 "value": "value1"
}
```

42. Addressing an Instance译：42.寻址一个实例

应用程序的每个实例都有一个服务ID，其值可以设置为`spring.cloud.bus.id`并且其值应该是冒号分隔的标识符列表，按照从最不特定到最具体的顺序排列。 默认值由环境构建为`spring.application.name`和`server.port` （或`spring.application.index` ，如果设置）的组合。 ID的默认值以`app:index:id`的形式`app:index:id` ，其中：

- `app` is the `vcap.application.name`, if it exists, or `spring.application.name`
- `index` is the `vcap.application.instance_index`, if it exists, `spring.application.index`, `local.server.port`, `server.port`, or `0` (in that order).
- `id` is the `vcap.application.instance_id`, if it exists, or a random value.

HTTP端点接受一个“目标”路径参数，例如`/bus-refresh/customers:9000` ，其中`destination`是一个服务ID。 如果ID由总线上的实例拥有，则处理该消息，并且所有其他实例忽略该消息。

43. Addressing All Instances of a Service译：43.寻址服务的所有实例

“目标”参数用于`PathMatcher` （使用路径分隔符作为冒号“ `:` ）来确定实例是否处理消息。 使用前面的示例， `/bus-env/customers:**`定位“客户”服务的所有实例，而不考虑服务标识的其余部分。

44. Service ID Must Be Unique译：44.服务ID必须是唯一的

公交车尝试两次以消除处理来自原始`ApplicationEvent`一次`ApplicationEvent`和一次来自队列的事件。 为此，它会根据当前的服务ID检查发送的服务ID。 如果服务的多个实例具有相同的ID，则不处理事件。 在本地计算机上运行时，每个服务位于不同的端口上，并且该端口是ID的一部分。 Cloud Foundry提供了一个区分指标。 为确保该ID在Cloud Foundry外部是唯一的，请将`spring.application.index`设置为每个服务实例的唯一内容。

45. Customizing the Message Broker译：45.定制Message Broker

Spring Cloud Bus使用Spring Cloud Stream来广播消息。 所以，为了让消息流动，你只需要在classpath中包含你选择的binder实现。 有AMQP（RabbitMQ）和Kafka（`spring-cloud-starter-bus-[amqp|kafka]` ）的公交车有便利的启动器。 一般来说，Spring Cloud Stream依靠Spring Boot自动配置约定来配置中间件。 例如，可以使用`spring.rabbitmq.*`配置属性更改AMQP代理地址。 Spring云总线在`spring.cloud.bus.*`有几个本机配置属性（例如， `spring.cloud.bus.destination`是用作外部中间件的主题的名称）。 通常，默认值就足够了。

要了解有关如何自定义消息代理设置的更多信息，请参阅Spring Cloud Stream文档。

46. Tracing Bus Events译：46.追踪巴士事件

公交车事件（ `RemoteApplicationEvent`子类）可以通过设置`spring.cloud.bus.trace.enabled=true`来追踪。 如果你这样做，Spring Boot `TraceRepository`（如果存在的话）显示每个服务实例发送的每个事件和所有的`TraceRepository` 。 以下示例来自`/trace`端点：

```
{
  "timestamp": "2015-11-26T10:24:44.411+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "stores:8081",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.864+0000",
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.862+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
}
```

前面的跟踪显示 `RefreshRemoteApplicationEvent` 从 `customers:9000` 发送，广播到所有服务，并收到 `customers:9000` 和 `stores:8081` 。

自己处理的ACK信号，则可以添加 `@EventListener` 为 `AckRemoteApplicationEvent` 种 `SentApplicationEvent` 类型您的应用程序（并启用跟踪）。 或者，您可以点击 `TraceRepository` 并从那里挖掘数据。

任何总线应用程序都可以追踪ack。 但是，有时在中央服务中执行此操作非常有用，它可以对数据执行更复杂的查询或将其转发给专用的跟踪服务。

47. Broadcasting Your Own Events译：47.广播自己的事件

巴士可以携带 `RemoteApplicationEvent` 类型的任何事件。 默认传输是JSON，并且解串器需要知道哪些类型将提前使用。 要注册一个新类型，您必须将其放在 `org.springframework.cloud.bus.event` 的子包中。

要自定义事件名称，可以在自定义类上使用 `@JsonTypeName` ，或者使用默认策略，即使用类的简单名称。

生产者和消费者都需要访问类定义。

47.1 Registering events in custom packages译：47.1在自定义包中注册事件

如果您不能或不想为自定义事件使用 `org.springframework.cloud.bus.event` 的子包， `org.springframework.cloud.bus.event` 必须使用 `@RemoteApplicationEventScan` 注释指定要扫描类型为 `RemoteApplicationEvent` 的事件的包。 使用 `@RemoteApplicationEventScan` 指定的软件包包含子软件包。

例如，请考虑以下自定义事件，名为 `MyEvent` ：

```
package com.acme;

public class MyEvent extends RemoteApplicationEvent {
    ...
}
```

您可以通过以下方式向解串器注册该事件：

```
package com.acme;

@Configuration
@RemoteApplicationEventScan
public class BusConfiguration {
    ...
}
```

如果没有指定值，则使用 `@RemoteApplicationEventScan` 的类的包将被注册。 在本例中， `com.acme` 通过使用 `BusConfiguration` 的包进行 `BusConfiguration` 。

您也可以明确指定包通过使用扫描 `value` ， `basePackages` 或者 `basePackageClasses` 上性能 `@RemoteApplicationEventScan` ，如下面的例子：

```
package com.acme;

@Configuration
//@RemoteApplicationEventScan({"com.acme", "foo.bar"})
//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})
@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)
public class BusConfiguration {
    ...
}
```

所有前面的示例 `@RemoteApplicationEventScan` 都是等效的，因为 `com.acme` 包是通过在 `@RemoteApplicationEventScan` 上明确指定包来注册的。

您可以指定要扫描的多个基本包。

Part VIII. Spring Cloud Sleuth译:第八部分。春云侦探

阿德里安科尔，斯宾塞吉布，Marcin Grzejszczak，Dave Syer，杰伊布莱恩特

Finchley.RELEASE

48. Introduction译:48.介绍

Spring Cloud Sleuth为 Spring Cloud实施分布式追踪解决方案。

48.1 Terminology译:48.1术语

Spring Cloud Sleuth借用了 Dapper's的术语。

跨度：基本的工作单位。 例如，发送RPC是一个新的跨度，就像向RPC发送响应一样。 跨度由跨度唯一的64位标识和跨度所包含的跟踪的另一个64位标识标识。 Spans还具有其他数据，如描述，时间戳事件，键值注释（标记），导致它们的跨度ID以及进程ID（通常为IP地址）。

跨度可以被启动和停止，并且他们跟踪他们的时间信息。 一旦你创建了一个跨度，你必须在将来某个时候停止它。

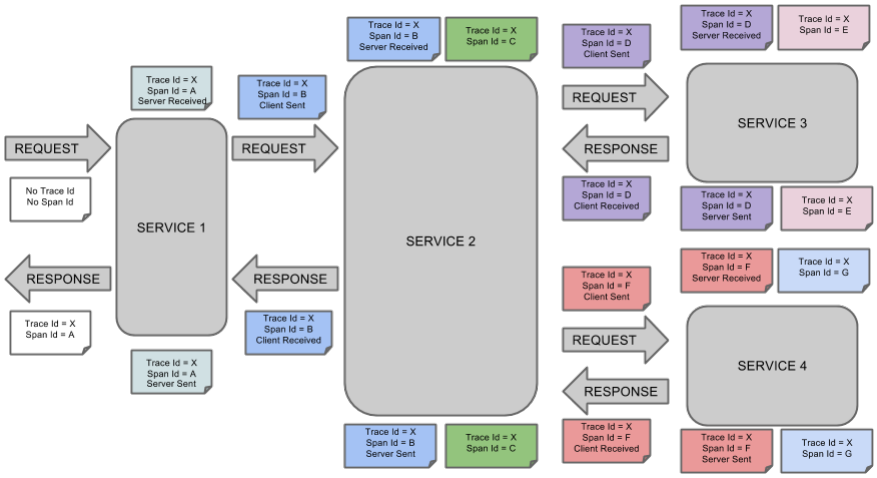
开始追踪的初始跨度称为`root span`。 该跨度的ID值等于跟踪ID。

痕迹：形成树状结构的一组跨度。 例如，如果您运行分布式大数据存储，则跟踪可能会由`PUT`请求组成。

注释：用于及时记录事件的存在。 使用Brave工具，我们不再需要为Zipkin设置特殊事件来了解客户端和服务器是谁，请求的起始位置以及结束位置。 然而，为了学习目的，我们标记这些事件以突出发生什么样的行为。

- **cs**: Client Sent. The client has made a request. This annotation indicates the start of the span.
- **sr**: Server Received: The server side got the request and started processing it. Subtracting the `cs` timestamp from this timestamp reveals the network latency.
- **ss**: Server Sent. Annotated upon completion of request processing (when the response got sent back to the client). Subtracting the `sr` timestamp from this timestamp reveals the time needed by the server side to process the request.
- **cr**: Client Received. Signifies the end of the span. The client has successfully received the response from the server side. Subtracting the `cs` timestamp from this timestamp reveals the whole time needed by the client to receive the response from the server.

下图显示了 **Span**和 **Trace**在系统中的外观以及Zipkin注释：

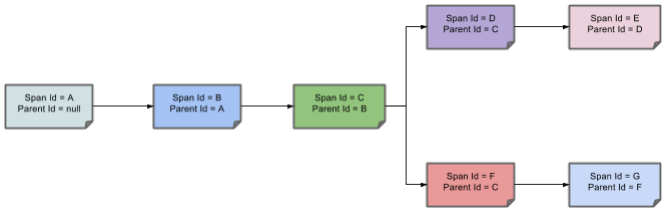


每个音符的颜色表示一个跨度（有七个跨度 - 从**A**到**G**）。 考虑以下说明：

```
Trace Id = X
Span Id = D
Client Sent
```

此注意表示当前跨度的**跟踪标识**设置为**X**， **跨度标识**设置为**D**. 此外， `Client Sent`事件发生。

下图显示了跨度的父子关系：

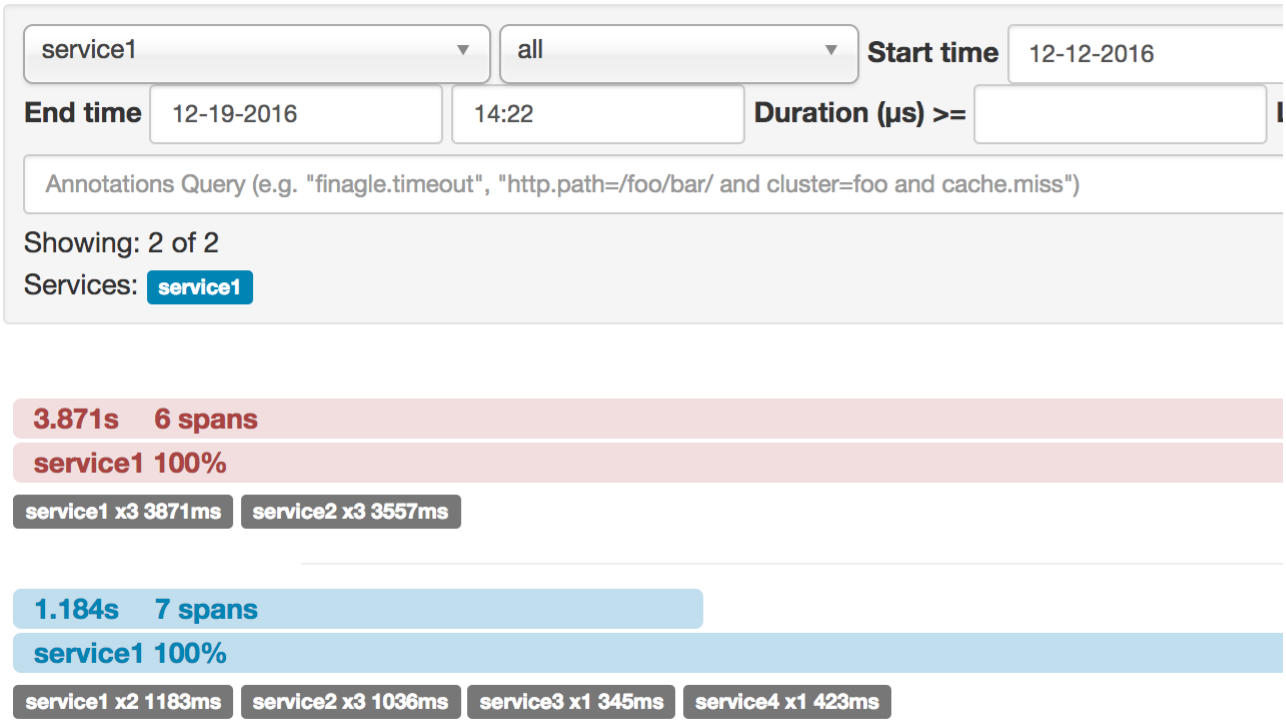


48.2 Purpose译:48.2目的

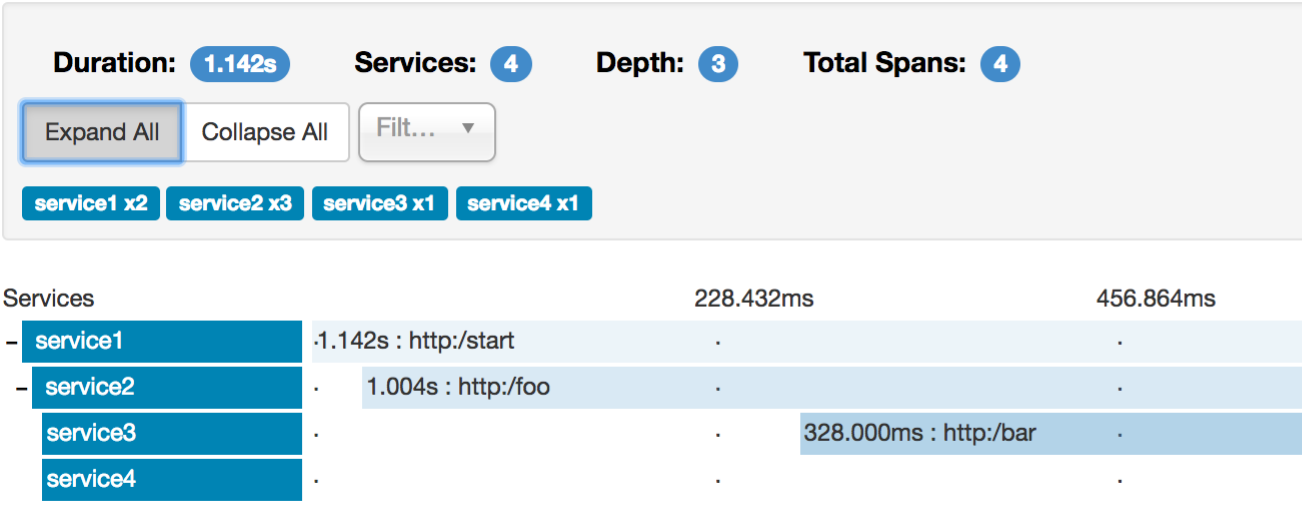
以下部分涉及上图中显示的示例。

48.2.1 Distributed Tracing with Zipkin译:48.2.1使用Zipkin进行分布式跟踪

这个例子有七个跨度。 如果您转到Zipkin中的轨迹，则可以在第二个轨迹中看到此数字，如下图所示：



但是，如果选择特定的轨迹，则可以看到四个跨度，如下图所示：



当您选择特定的轨迹时，您会看到合并的跨度。 这意味着，如果有两个发送给Zipkin的服务器接收和服务器发送，客户端接收和客户端发送的注释，它们将作为一个跨度显示。

为什么在这种情况下，七个和四个跨度之间存在差异？

- Two spans come from the `http:/start` span. It has the Server Received (`sr`) and Server Sent (`ss`) annotations.
- Two spans come from the RPC call from `service1` to `service2` to the `http:/foo` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service1` side. Server Received (`sr`) and Server Sent (`ss`) events took place on the `service2` side. These two spans form one logical span related to an RPC call.
- Two spans come from the RPC call from `service2` to `service3` to the `http:/bar` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service2` side. The Server Received (`sr`) and Server Sent (`ss`) events took place on the `service3` side. These two spans form one logical span related to an RPC call.
- Two spans come from the RPC call from `service2` to `service4` to the `http:/baz` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service2` side. Server Received (`sr`) and Server Sent (`ss`) events took place on the `service4` side. These two spans form one logical span related to an RPC call.

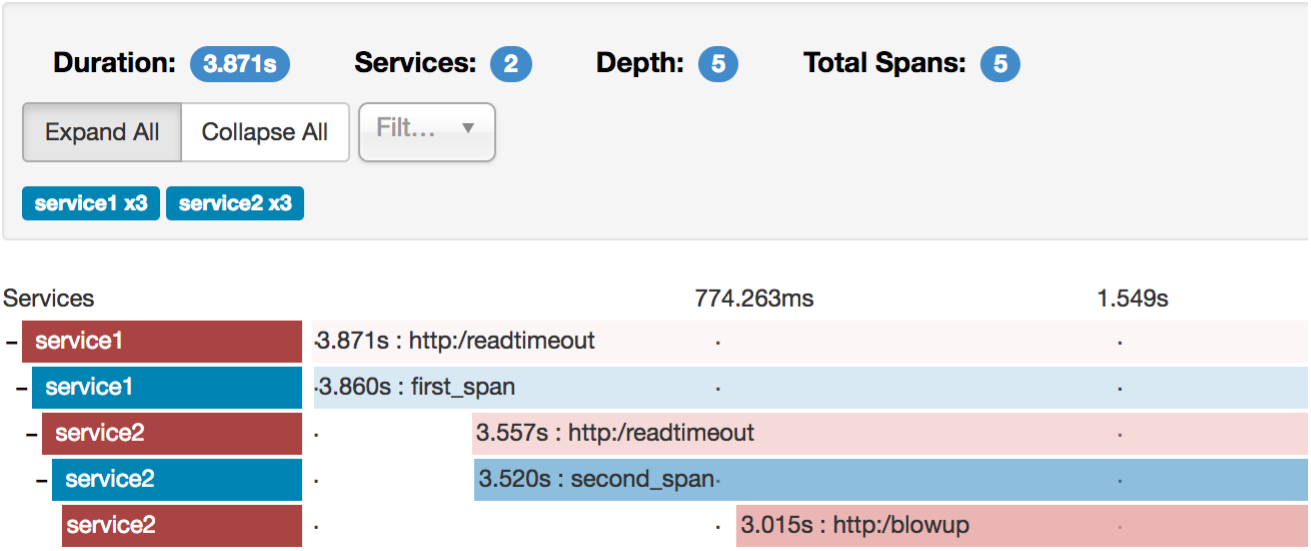
因此，如果我们计算物理跨度，我们有`http:/start`两个`service1` `service2`两个`service2` `service3`两个`service2` `service4` 。 总之，我们总共有七个跨度。

从逻辑上讲，我们可以看到总共四个Spans的信息，因为我们有一个与传入请求相关的跨度为`service1`并且有三个跨度与RPC调用相关。

48.2.2 Visualizing errors译：48.2.2可视化错误

Zipkin让你可以看到你的踪迹中的错误。 当抛出异常并且未被捕获时，我们在span上设置适当的标签，然后Zipkin可以正确着色。 您可以在轨迹列表中看到一条红色的轨迹。 这是因为抛出异常而出现的。

如果您单击该跟踪，则会看到类似的图片，如下所示：



如果您然后点击其中一个跨度，您会看到以下内容

service2.http:/readtimeout: 3.557s

AKA: service1,service2

Date Time	Relative Time	Annotation	Address
19/12/2016, 14:19:23	307.000ms	Client Send	127.0.0.1
19/12/2016, 14:19:23	310.000ms	Server Receive	127.0.0.1
19/12/2016, 14:19:26	3.836s	Server Send	127.0.0.1
19/12/2016, 14:19:27	3.864s	Client Receive	127.0.0.1

Key	Value
error	Request processing failed; nested exception is org.springframework.web.clien error on GET request for "http://localhost:8082/blowup": Read timed out; nest meoutException: Read timed out
http.host	localhost
http.method	GET
http.path	/readtimeout
http.status_code	500
http.url	http://localhost:8082/readtimeout
mvc.controller.class	BasicErrorController
mvc.controller.method	error

范围显示错误的原因以及与其相关的整个堆栈跟踪。

48.2.3 Distributed Tracing with Brave译：48.2.3勇敢的分布式跟踪

从版本`2.0.0`开始，Spring Cloud Sleuth使用Brave作为跟踪库。 因此，侦探不再需要保存上下文，而是将该工作委托给勇敢者。

由于Sleuth与Brave命名和标记规则不同，因此我们决定从现在开始遵循Brave的惯例。 但是，如果您要使用传统的Sleuth方法，则可以将`spring.sleuth.http.legacy.enabled`属性设置为`true` 。

48.2.4 Live examples译：48.2.4实例

图48.1。 点击Pivotal网站服务图标即可查看！

Click here to see it live!

Zipkin中的依赖关系图应该类似于以下图像：

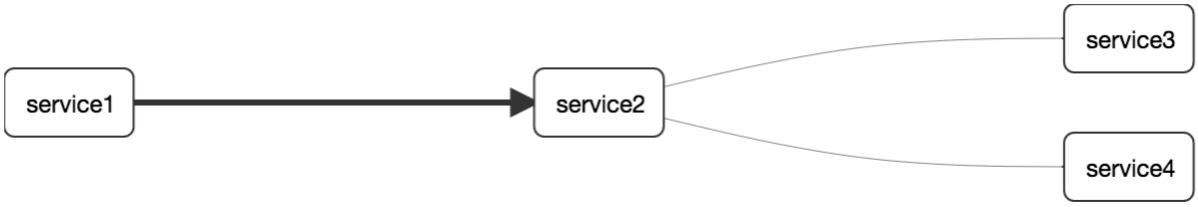


图48.2。 点击Pivotal网站服务图标即可查看！

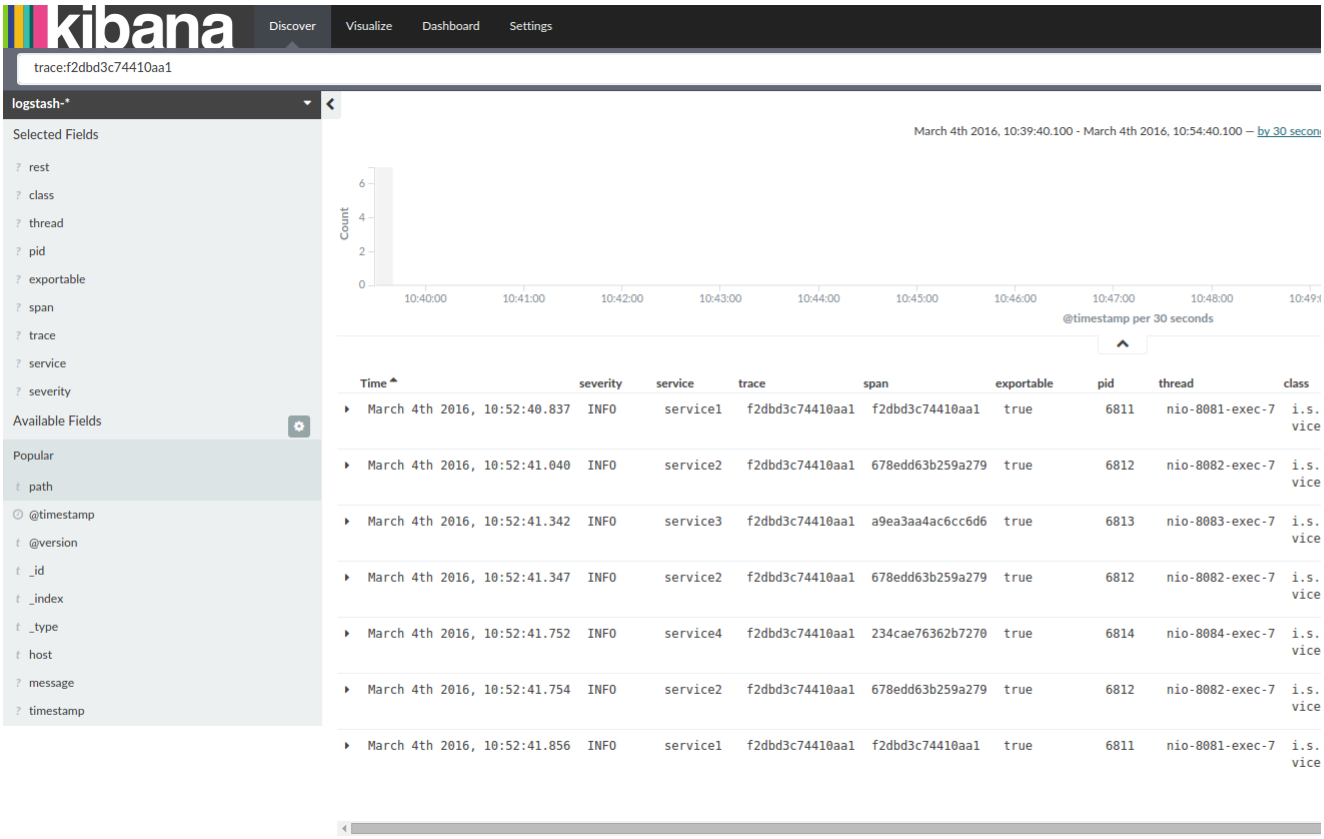
Click here to see it live!

48.2.5 Log correlation译：48.2.5日志关联

当使用grep通过扫描跟踪标识等于（例如） `2485ec27856c56f4`来读取这四个应用程序的日志时，您会得到类似于以下内容的输出：

```
service1.log:2016-02-26 11:15:47.561  INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.A
service2.log:2016-02-26 11:15:47.710  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.A
service3.log:2016-02-26 11:15:47.895  INFO [service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1] i.s.c.sleuth.docs.service3.Ap
service2.log:2016-02-26 11:15:47.924  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.A
service4.log:2016-02-26 11:15:48.134  INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1] i.s.c.sleuth.docs.service4.A
service2.log:2016-02-26 11:15:48.156  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.A
service1.log:2016-02-26 11:15:48.182  INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.A
```

如果您使用日志聚合工具（如Kibana， Splunk，和其他人），您可以订购所发生的事件。 Kibana的一个例子就像下面的图片：



如果您想使用 Logstash，以下列表显示Logstash的Grok模式：

```
filter {
       # pattern matching logback pattern
       grok {
              match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:ex
       }
}
```

如果您想将Grok与Cloud Foundry中的日志一起使用，则必须使用以下模式：

```
filter {
       # pattern matching logback pattern
       grok {
              match => { "message" => "(?m)OUT\s+%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span]
       }
}
```

JSON Logback with Logstash 译：JSON的 logback与 Logstash

通常，您不希望将日志存储在文本文件中，而是存储在Logstash可以立即选择的JSON文件中。 为此，您必须执行以下操作（为了便于阅读，我们在`groupId:artifactId:version`表示法中传递了依赖关系）。

依赖关系设置

1. Ensure that Logback is on the classpath (`ch.qos.logback:logback-core`).
2. Add Logstash Logback encode. For example, to use version `4.6`, add `net.logstash.logback:logstash-logback-encoder:4.6`.

Logback设置

考虑以下Logback配置文件的示例（名为 logback-spring.xml ）。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

 <springProperty scope="context" name="springAppName" source="spring.application.name"/>
 <!-- Example for logging into the build folder of your project -->
 <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>

 <!-- You can override this to have a custom pattern -->
 <property name="CONSOLE_LOG_PATTERN"
     value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint}

 <!-- Appender to log to console -->
 <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
  <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
   <!-- Minimum logging level to be presented in the console logs-->
   <level>DEBUG</level>
  </filter>
  <encoder>
   <pattern>${CONSOLE_LOG_PATTERN}</pattern>
   <charset>utf8</charset>
  </encoder>
 </appender>

 <!-- Appender to log to file -->
 <appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG_FILE}</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
   <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
   <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder>
   <pattern>${CONSOLE_LOG_PATTERN}</pattern>
   <charset>utf8</charset>
  </encoder>
 </appender>

 <!-- Appender to log to file in a JSON format -->
 <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG_FILE}.json</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
   <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
   <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
   <providers>
    <timestamp>
     <timeZone>UTC</timeZone>
    </timestamp>
    <pattern>
     <pattern>
      {
      "severity": "%level",
      "service": "${springAppName:-}",
      "trace": "%X{X-B3-TraceId:-}",
      "span": "%X{X-B3-SpanId:-}",
      "parent": "%X{X-B3-ParentSpanId:-}",
      "exportable": "%X{X-Span-Export:-}",
      "pid": "${PID:-}",
      "thread": "%thread",
      "class": "%logger{40}",
      "rest": "%message"
      }
     </pattern>
    </pattern>
   </providers>
  </encoder>
 </appender>

 <root level="INFO">
  <appender-ref ref="console"/>
  <!-- uncomment this to have also JSON logs -->
  <!--<appender-ref ref="logstash"/>-->
  <!--<appender-ref ref="flatfile"/>-->
 </root>
</configuration>
```

该Logback配置文件：

- Logs information from the application in a JSON format to a `build/${spring.application.name}.json` file.
- Has commented out two additional appenders: console and standard log file.
- Has the same logging pattern as the one presented in the previous section.

如果使用自定义`logback-spring.xml`，则必须通过`spring.application.name`中的`bootstrap`而不是`application`属性文件。 否则，您的自定义logback文件不能正确读取该属性。

48.2.6 Propagating Span Context 译：48.2.6传播跨度上下文

跨度上下文是必须传播到跨越进程边界的任何子跨度的状态。 跨度上下文的一部分是行李。 跟踪和跨度ID是跨度上下文的必需部分。 行李是可选部分。

行李是存储在跨度上下文中的一组关键值对。 行李与痕迹一起旅行，并附在每个跨度上。 Spring Cloud Sleuth明白，如果HTTP标题前缀为`baggage-`，则标头与行李相关，对于消息传递，它以`baggage_`。

Important

目前，行李物品的数量或尺寸没有限制。 但是，请记住，太多可能会降低系统吞吐量或增加RPC延迟。 在极端情况下，由于超出传输级别的消息或标头容量，太多的行李可能会使应用程序崩溃。

以下示例显示了在一个跨度上设置行李：

```
Span initialSpan = this.tracer.nextSpan().name("span").start();
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
 ExtraFieldPropagation.set("foo", "bar");
 ExtraFieldPropagation.set("UPPER_CASE", "someValue");
}
```

Baggage versus Span Tags译:行李与跨度标签

行李带着痕迹行驶（每个小孩都包含其父母的行李）。 Zipkin没有行李知识，也没有收到这些信息。

Important

从Sleuth 2.0.0开始，您必须在项目配置中明确传递行李键名称。 阅读有关该设置的更多信息here

标签附加到特定的跨度。 换句话说，它们仅针对特定的跨度呈现。 但是，如果存在具有搜索标签值的跨度，则可以通过标签搜索来查找跟踪。

如果您希望能够根据行李查找跨度，则应在根跨度中添加相应的条目作为标记。

Important

跨度必须在范围内。

以下列表显示使用行李的集成测试：

设置。

```
spring.sleuth:
  baggage-keys:
    - baz
    - bizarrecase
  propagation-keys:
    - foo
    - upper_case
```

代码。

```
initialSpan.tag("foo",
  ExtraFieldPropagation.get(initialSpan.context(), "foo"));
initialSpan.tag("UPPER_CASE",
  ExtraFieldPropagation.get(initialSpan.context(), "UPPER_CASE"));
```

48.3 Adding Sleuth to the Project译: 48.3为项目增加 测察力

本节介绍如何使用Maven或Gradle将Sleuth添加到项目中。

Important

为确保您的应用程序名称在Zipkin中正确显示，请将 `spring.application.name` 属性设置为 `bootstrap.yml` 。

48.3.1 Only Sleuth (log correlation)译: 48.3.1只有 Sleuth（对数相关）

如果您只想在没有Zipkin集成的情况下使用Spring Cloud Sleuth，请将 `spring-cloud-starter-sleuth` 模块添加到您的项目中。

以下示例显示如何使用Maven添加Sleuth：

Maven的。

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

1. 我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
2. 将依赖项添加到 `spring-cloud-starter-sleuth` 。

以下示例显示如何使用Gradle添加侦查器：

摇篮。

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-starter-sleuth"
}
```

我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
将依赖项添加到 `spring-cloud-starter-sleuth` 。

48.3.2 Sleuth with Zipkin via HTTP译：2 48.3.2通过 HTTP与 Zipkin侦察

如果你想要Sleuth和Zipkin，添加 `spring-cloud-starter-zipkin` 依赖项。

以下示例显示了如何为Maven执行此操作：

Maven的。

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
将依赖关系添加到 `spring-cloud-starter-zipkin` 。

以下示例显示了如何为Gradle执行此操作：

摇篮。

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-starter-zipkin"
}
```

我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
将依赖关系添加到 `spring-cloud-starter-zipkin` 。

48.3.3 Sleuth with Zipkin over RabbitMQ or Kafka译：48.3.3 Zipkin在 RabbitMQ或 Kafka上 的 侦 探

如果您想使用RabbitMQ或Kafka代替HTTP，请添加 `spring-rabbit` 或 `spring-kafka` 依赖项。 默认的目的地名称是 `zipkin` 。

> **Caution**
>
> `spring-cloud-sleuth-stream` 已被弃用且与这些目标不兼容。

如果你想通过RabbitMQ的Sleuth，添加 `spring-cloud-starter-zipkin` 和 `spring-rabbit` 依赖项。

以下示例显示了如何为Gradle执行此操作：

Maven的。

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```

我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
将依赖关系添加到 `spring-cloud-starter-zipkin` 。 这样，所有嵌套的依赖关系都被下载。
要自动配置RabbitMQ，请添加 `spring-rabbit` 依赖项。

摇篮。

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-starter-zipkin"
    compile "org.springframework.amqp:spring-rabbit"
}
```

我们建议您通过Spring BOM添加依赖关系管理，这样您就不需要自己管理版本。
将依赖项添加到 `spring-cloud-starter-zipkin` 。 这样，所有嵌套的依赖关系都被下载。
要自动配置RabbitMQ，请添加 `spring-rabbit` 依赖项。

49. Additional Resources 译: 49.其他资源

您可以观看 Reshmi Krishna和 Marcin Grzejszczak关于Spring Cloud Sleuth和Zipkin by clicking here的视频 。

您可以检查Sleuth和Brave in the openzipkin/sleuth-webmvc-example repository的不同设置。

50. Features 译:个 50.特点

- 将跟踪和跨度ID添加到Slf4J MDC，以便您可以从日志聚合器中的给定跟踪或跨度中提取所有日志，如以下示例日志所示：

  ```
  2016-02-02 15:30:57.902  INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
  2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
  2016-02-02 15:31:01.936  INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...
  ```

  请注意来自MDC的 `[appname,traceId,spanId,exportable]` 条目：
  - `spanId`: The ID of a specific operation that took place.
  - `appname`: The name of the application that logged the span.
  - `traceId`: The ID of the latency graph that contains the span.
  - `exportable`: Whether the log should be exported to Zipkin. When would you like the span not to be exportable? When you want to wrap some operation in a Span and have it written to the logs only.
- Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, and key-value annotations. Spring Cloud Sleuth is loosely based on HTrace but is compatible with Zipkin (Dapper).
- Sleuth records timing information to aid in latency analysis. By using sleuth, you can pinpoint causes of latency in your applications.
- Sleuth被写入不会记录太多并且不会导致您的生产应用程序崩溃。 为此，Sleuth：
  - Propagates structural data about your call graph in-band and the rest out-of-band.
  - Includes opinionated instrumentation of layers such as HTTP.
  - Includes a sampling policy to manage volume.
  - Can report to a Zipkin system for query and visualization.
- Instruments common ingress and egress points from Spring applications (servlet filter, async endpoints, rest template, scheduled actions, message channels, Zuul filters, and Feign client).
- Sleuth includes default logic to join a trace across HTTP or messaging boundaries. For example, HTTP propagation works over Zipkin-compatible request headers.
- Sleuth can propagate context (also known as baggage) between processes. Consequently, if you set a baggage element on a Span, it is sent downstream to other processes over either HTTP or messaging.
- Provides a way to create or continue spans and add tags and logs through annotations.
- 如果 `spring-cloud-sleuth-zipkin` 位于类路径中，该应用程序将生成并收集Zipkin兼容的跟踪。 默认情况下，它通过HTTP将它们发送到本地主机上的Zipkin服务器（端口9411）。 您可以通过设置 `spring.zipkin.baseUrl` 来配置服务的位置。
  - If you depend on `spring-rabbit` or `spring-kafka`, your app sends traces to a broker instead of HTTP. **

  > **Caution**
  >
  > `spring-cloud-sleuth-stream` 已弃用，因此不应再使用。

- Spring Cloud Sleuth is OpenTracing compatible.

> **Important**
>
> 如果使用Zipkin，请通过设置 `spring.sleuth.sampler.probability` （默认值：0.1，即10％）来配置导出跨度的概率。 否则，你可能会认为Sleuth不工作，因为它省略了一些跨度。

> SLF4J MDC始终处于设置状态，并且logback用户会立即在前面显示的示例中看到日志中的跟踪和跨度ID。 其他日志记录系统必须配置自己的格式化程序才能获得相同的结果。 默认值如下： `logging.pattern.level` 设置为 `%5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]` （这是用于登录用户的Spring Boot功能）。 如果您不使用SLF4J，则不会自动应用此模式。

50.1 Introduction to Brave 译: 50.1摘要的介绍

> **Important**
>
> 从版本 `2.0.0` 开始，Spring Cloud Sleuth使用Brave作为跟踪库。 为了您的方便，我们在此处嵌入了Brave的部分文档。

Important

在绝大多数情况下，您只需使用Sleuth提供的来自Brave的`Tracer`或`SpanCustomizer`豆。 下面的文档很好地概述了Brave是什么以及它是如何工作的。

Brave是一个库，用于捕获和向Zipkin报告有关分布式操作的延迟信息。 大多数用户不直接使用Brave。 他们使用图书馆或框架，而不是代表他们使用勇敢。

该模块包含一个创建和连接跨度的跟踪器，可以模拟潜在分布式工作的延迟。 它还包括通过网络边界传播跟踪上下文的库（例如，使用HTTP头）。

50.1.1 Tracing译:3 50.1.1追踪

最重要的是，您需要一个`brave.Tracer`，配置为 report to Zipkin 。

以下示例设置将跟踪数据（跨度）通过HTTP发送到Zipkin（与Kafka相对）：

```
class MyClass {

    private final Tracer tracer;

    // Tracer will be autowired
    MyClass(Tracer tracer) {
        this.tracer = tracer;
    }

    void doSth() {
        Span span = tracer.newTrace().name("encode").start();
        // ...
    }
}
```

Important

如果您的范围包含名称超过50个字符，则该名称将被截断为50个字符。 你的名字必须明确而具体。 大名称会导致延迟问题，有时甚至会引发异常。

跟踪器创建并加入跨度，这些跨度对潜在分布式工作的延迟进行建模。 它可以使用采样来减少处理过程中的开销，减少发送到Zipkin的数据量，或者两者兼而有之。

Spans在完成时由Tracker报告数据返回给Zipkin，或者如果未采样则不做任何事情。 开始跨度后，您可以注释感兴趣的事件或添加包含详细信息或查找键的标签。

跨度有一个上下文，其中包含跟踪标识符，将跨度放置在表示分布式操作的树中的正确位置。

50.1.2 Local Tracing译：50.1.2本地跟踪

跟踪本地代码时，可以在一个范围内运行它，如以下示例所示：

```
@Autowired Tracer tracer;

Span span = tracer.newTrace().name("encode").start();
try {
  doSomethingExpensive();
} finally {
  span.finish();
}
```

在前面的示例中，span是跟踪的根。 在很多情况下，跨度是现有轨迹的一部分。 在这种情况下，请拨`newChild`而不是`newTrace`，如以下示例所示：

```
@Autowired Tracer tracer;

Span span = tracer.newChild(root.context()).name("encode").start();
try {
  doSomethingExpensive();
} finally {
  span.finish();
}
```

50.1.3 Customizing Spans译：50.1.3定制跨度

一旦你有一个跨度，你可以添加标签。 标签可以用作查找键或细节。 例如，您可以使用运行时版本添加标签，如以下示例所示：

```
span.tag("clnt/finagle.version", "6.36.0");
```

在向第三方公开定制跨度的能力时，首选`brave.SpanCustomizer`而不是`brave.Span`。 前者更易于理解和测试，并且不会引发跨度生命周期挂钩的用户。

```
interface MyTraceCallback {
  void request(Request request, SpanCustomizer customizer);
}
```

由于`brave.Span`实现了`brave.SpanCustomizer`，因此可以将它传递给用户，如以下示例所示：

```
for (MyTraceCallback callback : userCallbacks) {
  callback.request(request, span);
}
```

50.1.4 Implicitly Looking up the Current Span译：50.1.4隐式查找当前量程

有时，您不知道跟踪是否正在进行，您不希望用户执行空检查。 `brave.CurrentSpanCustomizer`通过将数据添加到正在进行或正在删除的任何跨度来处理此问题，如以下示例所示：

防爆。

```
// The user code can then inject this without a chance of it being null.
@Autowired SpanCustomizer span;

void userCode() {
  span.annotate("tx.started");
  ...
}
```

50.1.5 RPC tracing译：50.1.5 RPC跟踪

在滚动您自己的RPC仪器之前检查 instrumentation written here和 Zipkin's list 。

RPC跟踪通常由拦截器自动完成。 在幕后，他们添加与他们在RPC操作中的角色相关的标签和事件。

以下示例显示如何添加客户端跨度：

```
@Autowired Tracer tracer;

// before you send a request, add metadata that describes the operation
span = tracer.newTrace().name("get").type(CLIENT);
span.tag("clnt/finagle.version", "6.36.0");
span.tag(TraceKeys.HTTP_PATH, "/api");
span.remoteEndpoint(Endpoint.builder()
    .serviceName("backend")
    .ipv4(127 << 24 | 1)
    .port(8080).build());

// when the request is scheduled, start the span
span.start();

// if you have callbacks for when data is on the wire, note those events
span.annotate(Constants.WIRE_SEND);
span.annotate(Constants.WIRE_RECV);

// when the response is complete, finish the span
span.finish();
```

One-Way tracing译:单向跟踪

有时候，您需要在有请求但没有响应的情况下为异步操作建模。 在正常的RPC跟踪中，您使用`span.finish()`来指示已收到响应。 在单向跟踪中，您可以使用`span.flush()`，因为您不期望得到响应。

以下示例显示了客户端如何建模单向操作：

```
@Autowired Tracer tracer;

// start a new span representing a client request
oneWaySend = tracer.newSpan(parent).kind(Span.Kind.CLIENT);

// Add the trace context to the request, so it can be propagated in-band
tracing.propagation().injector(Request::addHeader)
                     .inject(oneWaySend.context(), request);

// fire off the request asynchronously, totally dropping any response
request.execute();

// start the client side and flush instead of finish
oneWaySend.start().flush();
```

以下示例显示了服务器如何处理单向操作：

```
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// pull the context out of the incoming request
extractor = tracing.propagation().extractor(Request::getHeader);

// convert that context to a span which you can name and add tags to
oneWayReceive = nextSpan(tracer, extractor.extract(request))
    .name("process-request")
    .kind(SERVER)
    ... add tags etc.

// start the server side and flush instead of finish
oneWayReceive.start().flush();

// you should not modify this span anymore as it is complete. However,
// you can create children to represent follow-up work.
next = tracer.newSpan(oneWayReceive.context()).name("step2").start();
```

51. Sampling译：51.采样

采样可以用来减少收集和报告的过程数据。 当跨度不被采样时，它不会增加开销（一个noop）。

采样是预先决定的，这意味着报告数据的决定是在一个跟踪的第一个操作中做出的，并且决策是向下游传播的。

默认情况下，全局采样器将单个速率应用于所有跟踪的操作。 `Tracer.Builder.sampler`控制此设置，并且它默认跟踪每个请求。

51.1 Declarative sampling译：51.1声明式采样

某些应用程序需要根据java方法的类型或注释进行抽样。

大多数用户使用框架拦截器来实现这种策略的自动化。 以下示例显示了这可能如何在内部工作：

```
@Autowired Tracing tracing;

// derives a sample rate from an annotation on a java method
DeclarativeSampler<Traced> sampler = DeclarativeSampler.create(Traced::sampleRate);

@Around("@annotation(traced)")
public Object traceThing(ProceedingJoinPoint pjp, Traced traced) throws Throwable {
  Span span = tracing.tracer().newTrace(sampler.sample(traced))...
  try {
    return pjp.proceed();
  } finally {
    span.finish();
  }
}
```

51.2 Custom sampling 译：51.2自定义采样

根据操作的内容，您可能需要应用不同的策略。 例如，您可能不想跟踪对静态资源（如图像）的请求，或者您可能想要将所有请求都追踪到新的api。

大多数用户使用框架拦截器来实现这种策略的自动化。 以下示例显示了这可能如何在内部工作：

```
@Autowired Tracer tracer;

Span newTrace(Request input) {
  SamplingFlags flags = SamplingFlags.NONE;
  if (input.url().startsWith("/experimental")) {
    flags = SamplingFlags.SAMPLED;
  } else if (input.url().startsWith("/static")) {
    flags = SamplingFlags.NOT_SAMPLED;
  }
  return tracer.newTrace(flags);
}
```

51.3 Sampling in Spring Cloud Sleuth 译：51.3春季云 侦察采样

默认情况下，Spring Cloud Sleuth将所有跨度设置为不可导出。 这意味着痕迹出现在日志中，但不在任何远程存储中。 对于测试，默认通常就足够了，如果仅使用日志（例如，使用ELK聚合器），则可能只需要这些日志。 如果跨度的数据导出到基普金，还有一个`Sampler.ALWAYS_SAMPLE`设置，出口的一切，一个`ProbabilityBasedSampler`设置，样本跨度的固定分数。

如果您使用`spring-cloud-sleuth-zipkin`则`ProbabilityBasedSampler`是默认值。 您可以通过设置`spring.sleuth.sampler.probability`来配置导出。 传递的值需要从`0.0`到`1.0`。

可以通过创建一个bean定义来安装一个采样器，如下例所示：

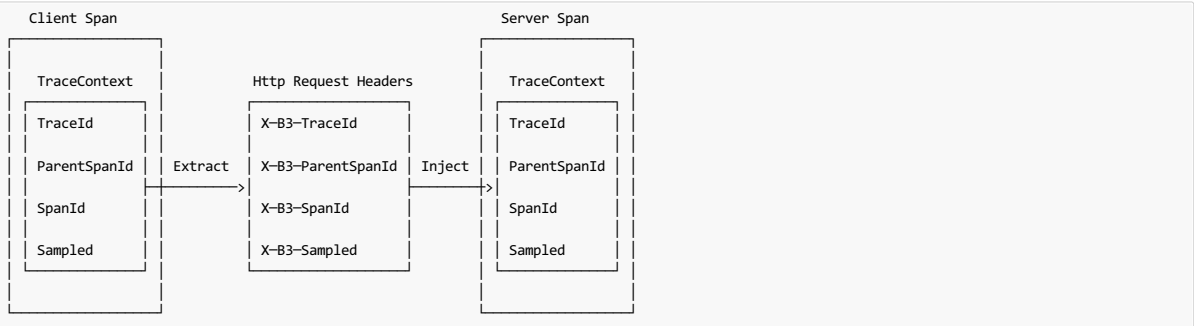
```
@Bean
public Sampler defaultSampler() {
 return Sampler.ALWAYS_SAMPLE;
}
```

您可以将HTTP标头`X-B3-Flags`设置为`1`，或者，在进行消息传递时，可以将`spanFlags`标头设置为`1`。 这样做会迫使当前的跨度可输出，而不管采样决定。

52. Propagation 译：52传播

需要传播以确保源自同一根的活动在相同的轨迹中收集在一起。 最常见的传播方法是通过向接收服务器的服务器发送RPC请求来复制客户端的跟踪上下文。

例如，在进行下游HTTP调用时，其跟踪上下文将被编码为请求标头并与其一起发送，如下图所示：



上面的名字来自 B3 Propagation，它是Brave内置的，并且在许多语言和框架中都有实现。

大多数用户使用框架拦截器来自动传播。 接下来的两个例子显示了这可能对客户端和服务器有效。

以下示例显示了客户端传播如何工作：

```
@Autowired Tracing tracing;

// configure a function that injects a trace context into a request
injector = tracing.propagation().injector(Request.Builder::addHeader);

// before a request is sent, add the current span's context to it
injector.inject(span.context(), request);
```

以下示例显示服务器端传播如何工作：

```
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// configure a function that extracts the trace context from a request
extractor = tracing.propagation().extractor(Request::getHeader);

// when a server receives a request, it joins or starts a new trace
span = tracer.nextSpan(extractor.extract(request));
```

52.1 Propagating extra fields译：52.1传播额外的领域

有时您需要传播额外的字段，例如请求ID或备用跟踪上下文。 例如，如果您位于Cloud Foundry环境中，则可能需要传递请求标识，如下例所示：

```
// when you initialize the builder, define the extra field you want to propagate
Tracing.newBuilder().propagationFactory(
  ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-vcap-request-id")
);

// later, you can tag that request ID or use it in log correlation
requestId = ExtraFieldPropagation.get("x-vcap-request-id");
```

您可能还需要传播您未使用的跟踪上下文。 例如，您可能位于Amazon Web Services环境中，但不会将数据报告给X-Ray。 为确保X射线可以正确共存，请传递其跟踪标头，如以下示例所示：

```
tracingBuilder.propagationFactory(
  ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-amzn-trace-id")
);
```

在Spring Cloud Sleuth中，跟踪构建器`Tracing.newBuilder()`所有元素都被定义为bean。 所以如果你想传递一个自定义的`PropagationFactory`，那么创建一个这种类型的bean就足够了，我们将在`Tracing` bean中设置它。

52.1.1 Prefixed fields译：52.1.1前缀字段

如果他们遵循一个通用模式，您也可以在字段前缀。 以下示例显示如何按`x-vcap-request-id`传播`x-vcap-request-id`字段，但分别`country-code` `user-id`发送`country-code`和`user-id`字段，分别为`x-baggage-country-code`和`x-baggage-user-id`：

```
Tracing.newBuilder().propagationFactory(
  ExtraFieldPropagation.newFactoryBuilder(B3Propagation.FACTORY)
                       .addField("x-vcap-request-id")
                       .addPrefixedFields("baggage-", Arrays.asList("country-code", "user-id"))
                       .build()
);
```

稍后，您可以调用以下代码来影响当前跟踪上下文的国家代码：

```
ExtraFieldPropagation.set("country-code", "FO");
String countryCode = ExtraFieldPropagation.get("country-code");
```

或者，如果您具有对跟踪上下文的引用，则可以明确使用它，如以下示例中所示：

```
ExtraFieldPropagation.set(span.context(), "country-code", "FO");
String countryCode = ExtraFieldPropagation.get(span.context(), "country-code");
```

Important

与以前版本的Sleuth的区别在于，使用Brave时，您必须通过行李钥匙列表。 有两个属性可以实现这一点。 通过`spring.sleuth.baggage-keys`，您可以设置以`baggage-`为前缀的密钥，以及`baggage_`的消息。 您还可以使用`spring.sleuth.propagation-keys`属性传递未加任何前缀列入白名单的前缀键列表。

52.1.2 Extracting a Propagated Context译：52.1.2提取传播的上下文

`TraceContext.Extractor<C>`从传入的请求或消息中读取跟踪标识符和采样状态。 载体通常是一个请求对象或标题。

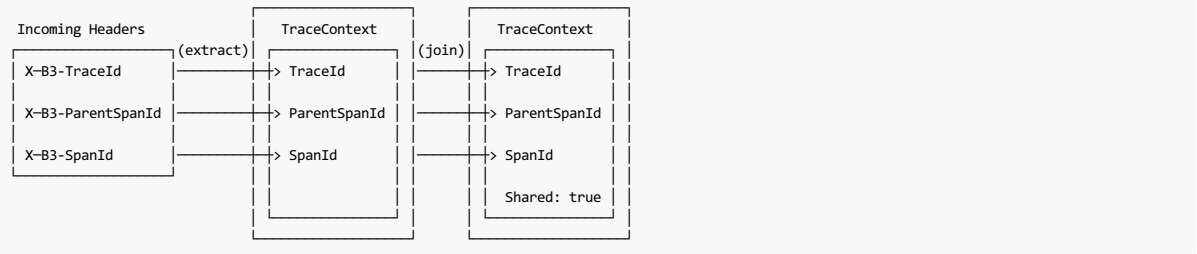
此实用程序用于标准仪器（如`HttpServerHandler``），但也可用于自定义RPC或消息传递代码。

`TraceContextOrSamplingFlags`通常只与`Tracer.nextSpan(extracted)`一起`Tracer.nextSpan(extracted)`，除非您在客户端和服务器之间共享`Tracer.nextSpan(extracted)`ID。

52.1.3 Sharing span IDs between Client and Server译：52.1.3在客户端和服务器之间共享跨度ID

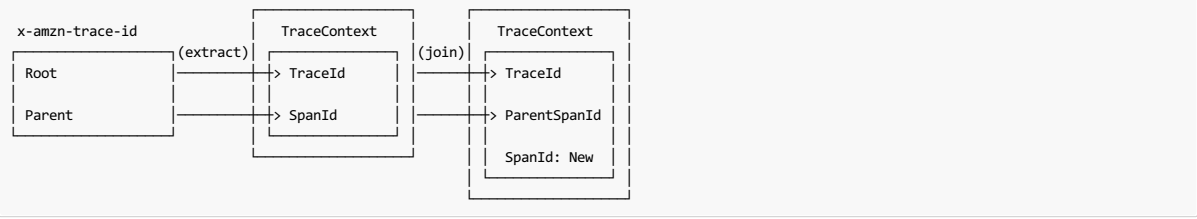
正常的检测模式是创建一个代表RPC服务器端的跨度。 `Extractor.extract`在应用于传入的客户端请求时可能会返回完整的跟踪上下文。 `Tracer.joinSpan`尝试继续此跟踪，如果支持，则使用相同的跨度ID，否则创建子跨度。 当span ID是共享时，报告的数据包含一个标志，如此。

下图显示了B3传播的一个示例：



一些传播系统只转发父跨度ID， `Propagation.Factory.supportsJoin() == false`时检测到。 在这种情况下，总是设置新的跨度ID，并且传入上下文确定父ID。

下图显示了AWS传播的一个示例：



注意：某些跨度报告器不支持共享跨度ID。 例如，如果您设置了`Tracing.Builder.spanReporter(amazonXrayOrGoogleStackdrive)`，则应该通过设置`Tracing.Builder.supportsJoin(false)`来禁用连接。 这样做强制`Tracer.joinSpan()`新的儿童跨度。

52.1.4 Implementing Propagation译：52.1.4实施传播

`TraceContext.Extractor<C>`由`Propagation.Factory`插件实现。 在内部，此代码使用以下其中一项创建联合类型`TraceContextOrSamplingFlags`：* `TraceContext`如果存在跟踪和跨度ID。 * `TraceIdContext`如果跟踪ID存在但跨度ID不存在。 * `SamplingFlags`如果没有标识符存在。

一些`Propagation`实现从提取点（例如，读入传入的头文件）向注入（例如，写出传出头文件）传送额外的数据。 例如，它可能带有一个请求ID。 当实现有额外的数据时，它们按如下方式处理它：*如果提取了`TraceContext`，则将额外的数据添加为`TraceContext.extra()`。 *否则，将其添加为`TraceContextOrSamplingFlags.extra()`，其中`Tracer.nextSpan`处理。

53. Current Tracing Component译：53.当前跟踪组件

勇敢的支持“当前追踪组件”的概念，只有当你没有其他方式获得引用时才应该使用它。 这是针对JDBC连接的，因为它们通常在跟踪组件之前初始化。

实例化的最新跟踪组件可通过`Tracing.current()`。 您也可以使用`Tracing.currentTracer()`来获取跟踪器。 如果您使用这些方法中的任何一种，请不要缓存结果。 相反，每次需要时都要查看它们。

54. Current Span译：54.当前跨度

勇敢支持代表飞行中操作的“当前跨度”概念。 您可以使用`Tracer.currentSpan()`将自定义标签添加到跨度，并使用`Tracer.nextSpan()`创建任何正在飞行的子项。

Important

在Sleuth中，您可以自动装配`Tracer` bean以通过`tracer.currentSpan()`方法检索当前范围。 要检索当前上下文，请致电`tracer.currentSpan().context()`。 要获取当前跟踪标识为String，可以使用`traceIdString()`方法：`tracer.currentSpan().context().traceIdString()`。

54.1 Setting a span in scope manually译：54.1手动设置范围

在编写新仪器时，重要的是将您在范围内创建的量程作为当前量程。 这样做不仅可以让用户通过`Tracer.currentSpan()`访问它，而且还允许自定义设置（如SLF4J MDC）查看当前跟踪ID。

`Tracer.withSpanInScope(Span)`提供了便利，并且通过使用try-with-resources成语最便于使用。 无论何时调用外部代码（例如继续执行拦截器或其他操作），请将范围放在范围内，如以下示例所示：

```
@Autowired Tracer tracer;

try (SpanInScope ws = tracer.withSpanInScope(span)) {
  return inboundRequest.invoke();
} finally { // note the scope is independent of the span
  span.finish();
}
```

在边缘情况下，您可能需要暂时清除当前范围（例如，启动不应与当前请求关联的任务）。 要做tso， `withSpanInScope` null传递给`withSpanInScope`，如以下示例所示：

```
@Autowired Tracer tracer;

try (SpanInScope cleared = tracer.withSpanInScope(null)) {
  startBackgroundThread();
}
```

55. Instrumentation译：55.仪器

Spring Cloud Sleuth自动处理所有的Spring应用程序，所以你不需要做任何事情来激活它。 根据可用的堆栈，使用各种技术添加检测工具。 例如，对于servlet Web应用程序，我们使用`Filter`，对于Spring Integration，我们使用`ChannelInterceptors`。

您可以自定义span标签中使用的键。 为了限制span数据量，默认情况下，HTTP请求仅使用少量元数据进行标记，例如状态码，主机和URL。 您可以通过配

置`spring.sleuth.keys.http.headers`（标题名称列表）来添加请求标题。

只有在`Sampler`允许的情况下，标签才会被收集和导出。 默认情况下，不存在这样的`Sampler`，以确保没有配置的情况下没有意外收集太多数据的危险）。

56. Span lifecycle译：56跨度生命周期

您可以通过`brave.Tracer`在Span上执行以下操作：

- start: When you start a span, its name is assigned and the start timestamp is recorded.
- close: The span gets finished (the end time of the span is recorded) and, if the span is sampled, it is eligible for collection (for example, to Zipkin).
- continue: A new instance of span is created. It is a copy of the one that it continues.
- detach: The span does not get stopped or closed. It only gets removed from the current thread.
- create with explicit parent: You can create a new span and set an explicit parent for it.

Spring Cloud Sleuth为您创建了一个`Tracer`的实例。 为了使用它，你可以自动装配它。

56.1 Creating and finishing spans译：56.1创建和修整跨度

您可以使用`Tracer`手动创建跨度，如以下示例所示：

```
// Start a span. If there was a span present in this thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(newSpan.start())) {
 // ...
 // You can tag a span
 newSpan.tag("taxValue", taxValue);
 // ...
 // You can log an event on a span
 newSpan.annotate("taxCalculated");
} finally {
 // Once done remember to finish the span. This will allow collecting
 // the span to send it to Zipkin
 newSpan.finish();
}
```

在前面的例子中，我们可以看到如何创建跨度的新实例。 如果此线程中已有跨度，则它将成为新跨度的父项。

Important

创建跨度后始终清洁。 此外，请始终完成要发送给Zipkin的任何跨度。

Important

如果您的跨度包含大于50个字符的名称，则该名称将被截断为50个字符。 你的名字必须明确而具体。 大名字导致延迟问题，有时甚至是例外。

56.2 Continuing Spans译：56.2持续跨度

有时候，你不想创建一个新的跨度，但你想继续。 这种情况的一个例子可能如下：

- **AOP**: If there was already a span created before an aspect was reached, you might not want to create a new span.
- **Hystrix**: Executing a Hystrix command is most likely a logical part of the current processing. It is in fact merely a technical implementation detail that you would not necessarily want to reflect in tracing as a separate being.

要继续跨度，可以使用`brave.Tracer`，如以下示例所示：

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X
Span continuedSpan = this.tracer.joinSpan(newSpan.context());
try {
 // ...
 // You can tag a span
 continuedSpan.tag("taxValue", taxValue);
 // ...
 // You can log an event on a span
 continuedSpan.annotate("taxCalculated");
} finally {
 // Once done remember to flush the span. That means that
 // it will get reported but the span itself is not yet finished
 continuedSpan.flush();
}
```

56.3 Creating a Span with an explicit Parent译：56.3使用明确的父项创建一个范围

您可能想要开始一个新的跨度并提供该跨度的明确父项。 假设跨度的父节点在一个线程中，并且您希望在另一个线程中开始一个新的跨度。 在“勇敢”中，无论何时调用`nextSpan()`，都会参照当前处于范围内的范围创建一个范围。 您可以将范围放在范围内，然后拨打`nextSpan()`，如以下示例所示：

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X. `initialSpan` will be the parent
// of the `newSpan`
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
 newSpan = this.tracer.nextSpan().name("calculateCommission");
 // ...
 // You can tag a span
 newSpan.tag("commissionValue", commissionValue);
 // ...
 // You can log an event on a span
 newSpan.annotate("commissionCalculated");
} finally {
 // Once done remember to finish the span. This will allow collecting
 // the span to send it to Zipkin. The tags and events set on the
 // newSpan will not be present on the parent
 if (newSpan != null) {
  newSpan.finish();
 }
}
```

Important

创建这样的跨度后，您必须完成它。 否则不会报告（例如，Zipkin）。

57. Naming spans译：57.命名跨度

选择一个跨度名称并不是一项简单的任务。 跨度名称应描述操作名称。 名字应该是低基数，所以它不应该包含标识符。

由于有很多仪器正在进行，有些跨度名称是人为的：

- `controller-method-name` when received by a Controller with a method name of `controllerMethodName`
- `async` for asynchronous operations done with wrapped `Callable` and `Runnable` interfaces.
- Methods annotated with `@Scheduled` return the simple name of the class.

幸运的是，对于异步处理，您可以提供明确的命名。

57.1 `@SpanName` Annotation译：57.1 `@SpanName`注释

您可以使用`@SpanName`注释明确命名该跨度，如下例所示：

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

 @Override public void run() {
  // perform logic
 }
}
```

在这种情况下，按以下方式处理时，跨度被命名为`calculateTax`：

```
Runnable runnable = new TraceRunnable(tracing, spanNamer,
  new TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

57.2 `toString()` method译：57.2 `toString()`方法

为`Runnable`或`Callable`创建单独的类是非常罕见的。 通常，创建这些类的匿名实例。 你不能注释这些类。 为了克服这个限制，如果没有`@SpanName`注释存在，我们检查该类是否具有`toString()`方法的自定义实现。

运行此类代码会导致创建一个名为`calculateTax`的跨度，如下例所示：

```
Runnable runnable = new TraceRunnable(tracing, spanNamer, new Runnable() {
 @Override public void run() {
  // perform logic
 }

 @Override public String toString() {
  return "calculateTax";
 }
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

58. Managing Spans with Annotations译：58.用批注管理 跨区

您可以使用各种注释来管理跨度。

58.1 Rationale译：58.1理由

管理带注释的跨度有很多很好的理由，其中包括：

- API-agnostic means to collaborate with a span. Use of annotations lets users add to a span with no library dependency on a span api. Doing so lets Sleuth change its core API to create less impact to user code.
- Reduced surface area for basic span operations. Without this feature, you must use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag, and log functionality, you can collaborate without accidentally breaking span lifecycle.

- Collaboration with runtime generated code. With libraries such as Spring Data and Feign, the implementations of interfaces are generated at runtime. Consequently, span wrapping of objects was tedious. Now you can provide annotations over interfaces and the arguments of those interfaces.

58.2 Creating New Spans译：58.2创建新跨度

如果您不想手动创建本地跨度，则可以使用`@NewSpan`注释。 另外，我们提供`@SpanTag`注释以自动方式添加标签。

现在我们可以考虑一些使用的例子。

```
@NewSpan
void testMethod();
```

在没有任何参数的情况下注释该方法会导致创建一个名称等于注释方法名称的新跨度。

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

如果您在注释中提供值（直接或通过设置`name`参数），则创建的跨度将提供的值作为名称。

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

您可以结合名称和标签。 让我们关注后者。 在这种情况下，带注释的方法的参数运行时值的值将成为标记的值。 在我们的示例中，标签密钥为`testTag`，标签值为`test`。

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

您可以将`@NewSpan`注释放在类和接口上。 如果覆盖该接口的方法并为`@NewSpan`注释提供不同的值，则最具体的一个将获胜（在此情况下`customNameOnTestMethod3`为`customNameOnTestMethod3`）。

58.3 Continuing Spans译：58.3持续跨度

如果要将标记和注释添加到现有跨度，则可以使用`@ContinueSpan`注释，如以下示例中所示：

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
this.testBean.testMethod13();
```

（请注意，与`@NewSpan`注释相比，您还可以使用`log`参数添加日志。）

这样，跨度得以延续，并且：

- Log entries named `testMethod11.before` and `testMethod11.after` are created.
- If an exception is thrown, a log entry named `testMethod11.afterFailure` is also created.
- A tag with a key of `testTag11` and a value of `test` is created.

58.4 Advanced Tag Setting译：58.4高级标签设置

有3种不同的方法可以为标签添加标签。 所有这些都由`SpanTag`注释控制。 优先顺序如下：

1. Try with a bean of `TagValueResolver` type and a provided name.
2. If the bean name has not been provided, try to evaluate an expression. We search for a `TagValueExpressionResolver` bean. The default implementation uses SPEL expression resolution. **IMPORTANT** You can only reference properties from the SPEL expression. Method execution is not allowed due to security constraints.
3. If we do not find any expression to evaluate, return the `toString()` value of the parameter.

58.4.1 Custom extractor译：58.4.1自定义提取器

以下方法的标签值由`TagValueResolver`接口的实现计算`TagValueResolver`。 它的类名必须作为`resolver`属性的值传递。

考虑下面的注释方法：

```
@NewSpan
public void getAnnotationForTagValueResolver(@SpanTag(key = "test", resolver = TagValueResolver.class) String test) {
}
```

现在进一步考虑下面的`TagValueResolver` bean的实现：

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
 return parameter -> "Value from myCustomTagValueResolver";
}
```

前面的两个示例导致设置标签值等于`Value from myCustomTagValueResolver`。

58.4.2 Resolving Expressions for a Value译：58.4.2解析值的表达式

考虑下面的注释方法：

```
@NewSpan
public void getAnnotationForTagValueExpression(@SpanTag(key = "test", expression = "'hello' + ' characters'") String test) {
}
```

`TagValueExpressionResolver`自定义实现`TagValueExpressionResolver`导致评估SPEL表达式，并且跨度上设置值为`4 characters`的标签。 如果你想使用其他表达式解析机制，你可以创建你自己的bean实现。

58.4.3 Using the `toString()` method译: 58.4.3使用`toString()`方法

考虑下面的注释方法：

```
@NewSpan
public void getAnnotationForArgumentToString(@SpanTag("test") Long param) {
}
```

运行上述方法的值为`15`导致设置字符串值为`"15"`。

59. Customizations译: 59.自定义

59.1 HTTP译: 59.1 HTTP

如果需要自定义客户端/服务器解析HTTP相关跨度，则只需注册类型为`brave.http.HttpClientParser`或`brave.http.HttpServerParser`的bean。 如果需要客户端/服务器采样，只需注册一个类型为`brave.http.HttpSampler`的bean， `sleuthClientSampler`为客户端采样器命名bean `sleuthClientSampler`，为服务器采样器`sleuthServerSampler`为`sleuthServerSampler`。 为方便起见，可以使用`@ClientSampler`和`@ServerSampler`注释来注入适当的bean或通过其静态String `NAME`字段来引用bean名称。

查看Brave的代码以查看如何制作基于路径的采样器的 示例https://github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy

如果您想完全重写`HttpTracing` bean，则可以使用`SkipPatternProvider`界面来检索应该不被抽样的跨度的URL `Pattern`。 下面你可以看到使用的例子`SkipPatternProvider`服务器端内， `HttpSampler`。

```
@Configuration
class Config {
  @Bean(name = ServerSampler.NAME)
  HttpSampler myHttpSampler(SkipPatternProvider provider) {
   Pattern pattern = provider.skipPattern();
   return new HttpSampler() {

    @Override public <Req> Boolean trySample(HttpAdapter<Req, ?> adapter, Req request) {
     String url = adapter.path(request);
     boolean shouldSkip = pattern.matcher(url).matches();
     if (shouldSkip) {
      return false;
     }
     return null;
    }
   };
  }
}
```

59.2 `TracingFilter`译: 59.2 `TracingFilter`

您还可以修改`TracingFilter`的行为，该组件负责处理输入的HTTP请求并添加基于HTTP响应的标记。 您可以通过注册您自己的`TracingFilter` bean实例来自定义标签或修改响应标头。

在以下示例中，我们注册了`TracingFilter` bean，添加了包含当前Span的跟踪标识的`ZIPKIN-TRACE-ID`响应标题，并向该范围添加了标记`custom`和值`tag`。

```
@Component
@Order(TraceWebServletAutoConfiguration.TRACING_FILTER_ORDER + 1)
class MyFilter extends GenericFilterBean {

 private final Tracer tracer;

 MyFilter(Tracer tracer) {
  this.tracer = tracer;
 }

 @Override public void doFilter(ServletRequest request, ServletResponse response,
   FilterChain chain) throws IOException, ServletException {
  Span currentSpan = this.tracer.currentSpan();
  if (currentSpan == null) {
   chain.doFilter(request, response);
   return;
  }
  // for readability we're returning trace id in a hex form
  ((HttpServletResponse) response)
    .addHeader("ZIPKIN-TRACE-ID",
      currentSpan.context().traceIdString());
  // we can also add some custom tags
  currentSpan.tag("custom", "tag");
  chain.doFilter(request, response);
 }
}
```

59.3 Custom service name译: 59.3自定义服务名称

默认情况下，Sleuth假设，当您向Spankin发送跨度时，您希望跨度服务名称等于`spring.application.name`属性的值。 但情况并非总是如此。 在某些情况下，您希望为来自应用程序的所有跨度显式提供不同的服务名称。 为此，您可以将以下属性传递给应用程序以覆盖该值（该示例适用于名为`myService`的服务）：

```
spring.zipkin.service.name: myService
```

59.4 Customization of Reported Spans 译：59.4定制报告的跨度

在报告跨度（例如，Zipkin）之前，您可能需要以某种方式修改该跨度。 您可以通过使用`SpanAdjuster`界面来完成`SpanAdjuster`操作。

在Sleuth中，我们生成一个固定名称的跨度。 一些用户想要根据标签的值修改名称。 您可以实现`SpanAdjuster`界面来更改该名称。

以下示例显示如何注册两个实现 `SpanAdjuster` bean：

```
@Bean SpanAdjuster adjusterOne() {
 return span -> span.toBuilder().name("foo").build();
}

@Bean SpanAdjuster adjusterTwo() {
 return span -> span.toBuilder().name(span.name() + " bar").build();
}
```

前面的示例导致在报告之前将报告的范围的名称更改为 `foo bar` （例如，Zipkin）。

59.5 Host Locator 译：59.5主机定位器

Important

本节介绍如何从服务发现中定义**主机** 。 这**不是**关于通过服务发现找到Zipkin。

要定义与特定跨度相对应的主机，我们需要解析主机名和端口。 默认的方法是从服务器属性中获取这些值。 如果没有设置，我们尝试从网络接口检索主机名。

如果启用了发现客户端并且希望从服务注册表中的已注册实例中检索主机地址，则必须设置 `spring.zipkin.locator.discovery.enabled`属性（它适用于基于HTTP和基于流的跨度报告），如下所示：

```
spring.zipkin.locator.discovery.enabled: true
```

60. Sending Spans to Zipkin 译：60.发送跨度到Zipkin

默认情况下，如果将`spring-cloud-starter-zipkin`作为依赖项添加到项目中，那么当范围关闭时，它会通过HTTP发送到Zipkin。 通信是异步的。 您可以通过设置`spring.zipkin.baseUrl`属性来配置URL，如下所示：

```
spring.zipkin.baseUrl: http://192.168.99.100:9411/
```

如果您想通过服务发现来查找Zipkin，则可以在URL内传递Zipkin的服务ID，如以下示例所示，用于 `zipkinserver`服务ID：

```
spring.zipkin.baseUrl: http://zipkinserver/
```

要禁用此功能，只需将 `spring.zipkin.discoveryClientEnabled`设置为false。

当启用发现客户端功能时，Sleuth使用`LoadBalancerClient`来查找Zipkin服务器的URL。 这意味着您可以通过功能区设置负载平衡配置。

```
zipkinserver:
  ribbon:
    ListOfServers: host1,host2
```

如果您在类路径中一起使用了web，rabbit或kafka，则可能需要选择想要将跨度发送到zipkin的方式。 要做到这一点，设置`web` ， `rabbit` ，或`kafka`至`spring.zipkin.sender.type`财产。 以下示例显示设置`web`的发件人类型：

```
spring.zipkin.sender.type: web
```

要定制 `RestTemplate`通过HTTP发送跨距基普金，你可以注册 `ZipkinRestTemplateCustomizer`豆。

```
@Configuration
class MyConfig {
 @Bean ZipkinRestTemplateCustomizer myCustomizer() {
  return new ZipkinRestTemplateCustomizer() {
   @Override
   void customize(RestTemplate restTemplate) {
    // customize the RestTemplate
   }
  };
 }
}
```

但是，如果您想要控制创建 `RestTemplate`对象的完整过程，则必须创建 `zipkin2.reporter.Sender`类型的bean。

```
@Bean Sender myRestTemplateSender(ZipkinProperties zipkin,
   ZipkinRestTemplateCustomizer zipkinRestTemplateCustomizer) {
  RestTemplate restTemplate = mySuperCustomRestTemplate();
  zipkinRestTemplateCustomizer.customize(restTemplate);
  return myCustomSender(zipkin, restTemplate);
 }
```

61. Zipkin Stream Span Consumer 译：61. Zipkin流跨度消费者

Important

我们建议使用Zipkin的本地支持来进行基于消息的跨度发送。 从Edgware版本开始，Zipkin Stream服务器已弃用。 在Finchley版本中，它被删除。

如果由于某种原因，您需要创建弃用的Stream Zipkin服务器，请参阅 Dalston Documentation 。

62. Integrations译：62.集成

62.1 OpenTracing译：62.1 OpenTracing

Spring Cloud Sleuth兼容OpenTracing 。 如果你在类路径上有OpenTracing，我们会自动注册OpenTracing `Tracer` bean。 如果您希望禁用此功能，请将`spring.sleuth.opentracing.enabled`设置为`false`

62.2 Runnable and Callable译：62.2可运行和可调用

如果将您的逻辑封装在 `Runnable`或 `Callable` ，则可以将这些类封装到他们的Sleuth代表中，如下面的示例 `Runnable` ：

```
Runnable runnable = new Runnable() {
 @Override
 public void run() {
  // do some work
 }

 @Override
 public String toString() {
  return "spanNameFromToStringMethod";
 }
};
// Manual `TraceRunnable` creation with explicit "calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(tracing, spanNamer, runnable,
  "calculateTax");
// Wrapping `Runnable` with `Tracing`. That way the current span will be available
// in the thread of `Runnable`
Runnable traceRunnableFromTracer = tracing.currentTraceContext().wrap(runnable);
```

以下示例显示了如何为 `Callable`执行此 `Callable` ：

```
Callable<String> callable = new Callable<String>() {
 @Override
 public String call() throws Exception {
  return someLogic();
 }

 @Override
 public String toString() {
  return "spanNameFromToStringMethod";
 }
};
// Manual `TraceCallable` creation with explicit "calculateTax" Span name
Callable<String> traceCallable = new TraceCallable<>(tracing, spanNamer, callable,
  "calculateTax");
// Wrapping `Callable` with `Tracing`. That way the current span will be available
// in the thread of `Callable`
Callable<String> traceCallableFromTracer = tracing.currentTraceContext().wrap(callable);
```

这样，您可以确保为每次执行创建并关闭新跨度。

62.3 Hystrix译：62.3 Hystrix

62.3.1 Custom Concurrency Strategy译：62.3.1自定义并发策略

我们注册了一个名为`TraceCallable`的自定义`HystrixConcurrencyStrategy`， `TraceCallable`所有`Callable`实例包装在他们的Sleuth代表中。 该策略要么开始要么继续一段时间，取决于在Hystrix命令被调用之前是否已经进行了跟踪。 要禁用自定义Hystrix并发策略，请将`spring.sleuth.hystrix.strategy.enabled`设置为`false` 。

62.3.2 Manual Command setting译：62.3.2手动命令设置

假设您拥有以下 `HystrixCommand` ：

```
HystrixCommand<String> hystrixCommand = new HystrixCommand<String>(setter) {
 @Override
 protected String run() throws Exception {
  return someLogic();
 }
};
```

要传递跟踪信息，必须在 `HystrixCommand`的Sleuth版本中包含相同的逻辑，该版本称为 `TraceCommand` ，如以下示例所示：

```
TraceCommand<String> traceCommand = new TraceCommand<String>(tracer, setter) {
 @Override
 public String doRun() throws Exception {
  return someLogic();
 }
};
```

62.4 RxJava译：62.4 RxJava

我们注册了一个自定义`RxJavaSchedulersHook` ，它将所有`Action0`实例包装在他们的Sleuth代表中，这个代表被称为`TraceAction` 。 挂钩要么开始要么继续一段时间，这取决于在Action被安排之前是否已经进行了跟踪。 要禁用自定义`RxJavaSchedulersHook` ，请将`spring.sleuth.rxjava.schedulers.hook.enabled`设置为`false` 。

您可以为不希望跨度创建的线程名称定义正则表达式列表。 为此，请在`spring.sleuth.rxjava.schedulers.ignoredthreads`属性中提供逗号分隔的正则表达式列表。

Important

反应式编程和Sleuth的建议方法是使用反应堆支持。

62.5 HTTP integration 译：62.5 HTTP集成

从该部分的功能可以通过设置被禁用 `spring.sleuth.web.enabled` 与值等于属性 `false` 。

62.5.1 HTTP Filter 译：62.5.1 HTTP过滤器

通过 `TracingFilter` ，所有采样的传入请求都会导致创建一个Span。 该Span的名称是 `http:` +发送请求的路径。 例如，如果请求发送到 `/this/that` 那么名称将为 `http:/this/that` 。 您可以通过设置 `spring.sleuth.web.skipPattern` 属性来配置您想跳过的URI。 如果类路径上有 `ManagementServerProperties` ，则其值 `contextPath` 会附加到所提供的跳转模式。 如果您想重用Sleuth的默认跳跃模式并仅追加自己的模式，请使用 `spring.sleuth.web.additionalSkipPattern` 来传递这些模式。

62.5.2 HandlerInterceptor 译：62.5.2 HandlerInterceptor

由于我们希望span名称是精确的，因此我们使用 `TraceHandlerInterceptor` ，其中包含现有 `HandlerInterceptor` 或直接添加到现有 `HandlerInterceptors` 的列表中。 该 `TraceHandlerInterceptor` 增加了一个特殊的请求属性给定 `HttpServletRequest` 。 如果 `TracingFilter` 没有看到该属性，它会创建一个“下降”跨度，这是在服务器端创建的一个额外跨度，以便跟踪在UI中正确呈现。 如果发生这种情况，可能会丢失仪器。 在这种情况下，请在Spring Cloud Sleuth中提出问题。

62.5.3 Async Servlet support 译：62.5.3异步Servlet支持

如果您的控制器返回 `Callable` 或 `WebAsyncTask` ，则Spring Cloud Sleuth将继续现有跨度，而不是创建新跨度。

62.5.4 WebFlux support 译：62.5.4 WebFlux支持

通过 `TraceWebFilter` ，所有采样传入请求都会导致创建一个Span。 该Span的名称是 `http:` +发送请求的路径。 例如，如果请求发送到 `/this/that` ，则名称是 `http:/this/that` 。 您可以使用 `spring.sleuth.web.skipPattern` 属性来配置要跳过的URI。 如果类路径上有 `ManagementServerProperties` ，则其值 `contextPath` 会附加到所提供的跳转模式。 如果您想重复使用Sleuth的默认跳跃模式并追加自己的模式，请使用 `spring.sleuth.web.additionalSkipPattern` 来传递这些模式。

62.5.5 Dubbo RPC support 译：62.5.5 Dubbo RPC支持

通过与勇敢的整合，Spring Cloud Sleuth支持Dubbo 。 它足以添加 `brave-instrumentation-dubbo-rpc` 依赖项：

```
<dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-instrumentation-dubbo-rpc</artifactId>
</dependency>
```

您还需要使用以下内容设置 `dubbo.properties` 文件：

```
dubbo.provider.filter=tracing
dubbo.consumer.filter=tracing
```

你可以阅读更多关于勇敢 - 达博集成here 。 Spring Cloud Sleuth和Dubbo的一个例子可以找到here 。

62.6 HTTP Client Integration 译：62.6 HTTP客户端集成

62.6.1 Synchronous Rest Template 译：62.6.1同步休息模板

我们注入一个 `RestTemplate` 拦截器，以确保所有跟踪信息都传递给请求。 每次打电话时，都会创建一个新的跨度。 收到答复后它会关闭。 要阻止同步 `RestTemplate` 功能，请将 `spring.sleuth.web.client.enabled` 设置为 `false` 。

Important

您必须将 `RestTemplate` 注册为一个bean，以便拦截器被注入。 如果使用 `new` 关键字创建 `RestTemplate` 实例，则检测不起作用。

62.6.2 Asynchronous Rest Template 译：62.6.2异步休息模板

Important

从Sleuth `2.0.0` 开始，我们不再注册 `AsyncRestTemplate` 类型的bean。 这取决于你创建这样一个bean。 然后我们来测试它。

要阻止 `AsyncRestTemplate` 功能，请将 `spring.sleuth.web.async.client.enabled` 设置为 `false` 。 要禁用创建默认 `TraceAsyncClientHttpRequestFactoryWrapper` ，请将 `spring.sleuth.web.async.client.factory.enabled` 设置为 `false` 。 如果您根本不想创建 `AsyncRestClient` ，请将 `spring.sleuth.web.async.client.template.enabled` 设置为 `false` 。

Multiple Asynchronous Rest Templates 译：多个异步休息模板

有时您需要使用异步休息模板的多个实现。 在以下代码片段中，您可以看到如何设置这种自定义 `AsyncRestTemplate` ：

```
@Configuration
@EnableAutoConfiguration
static class Config {

 @Bean(name = "customAsyncRestTemplate")
 public AsyncRestTemplate traceAsyncRestTemplate() {
  return new AsyncRestTemplate(asyncClientFactory(), clientHttpRequestFactory());
 }

 private ClientHttpRequestFactory clientHttpRequestFactory() {
  ClientHttpRequestFactory clientHttpRequestFactory = new CustomClientHttpRequestFactory();
  //CUSTOMIZE HERE
  return clientHttpRequestFactory;
 }

 private AsyncClientHttpRequestFactory asyncClientFactory() {
  AsyncClientHttpRequestFactory factory = new CustomAsyncClientHttpRequestFactory();
  //CUSTOMIZE HERE
  return factory;
 }
}
```

62.6.3 `WebClient` 译：62.6.3 `WebClient`

我们注入了一个创建跨度的 `ExchangeFilterFunction` 实现，并通过on-success和on-error回调来关闭客户端跨度。

要阻止此功能，请将 `spring.sleuth.web.client.enabled` 设置为 `false` 。

Important

您必须将 `WebClient` 注册为bean，以便跟踪检测得到应用。 如果使用 `new` 关键字创建 `WebClient` 实例，则检测不起作用。

62.6.4 Traverson 译：62.6.4 Traverson

如果使用Traverson库，则可以将 `RestTemplate` 作为bean注入到Traverson对象中。 由于 `RestTemplate` 已被拦截，因此您将全力支持您的客户端进行跟踪。 以下伪码显示了如何做到这一点：

```
@Autowired RestTemplate restTemplate;

Traverson traverson = new Traverson(URI.create("http://some/address"),
    MediaType.APPLICATION_JSON, MediaType.APPLICATION_JSON_UTF8).setRestOperations(restTemplate);
// use Traverson
```

62.6.5 Apache `HttpClientBuilder` and `HttpAsyncClientBuilder` 译：62.6.5 Apache `HttpClientBuilder` 和 `HttpAsyncClientBuilder`

我们测试 `HttpClientBuilder` 和 `HttpAsyncClientBuilder` 以便将跟踪上下文注入发送的请求。

要阻止这些功能，请将 `spring.sleuth.web.client.enabled` 设置为 `false` 。

62.6.6 Netty `HttpClient` 译：62.6.6 Netty `HttpClient`

我们测试Netty的 `HttpClient` 。

要阻止此功能，请将 `spring.sleuth.web.client.enabled` 设置为 `false` 。

Important

您必须将 `HttpClient` 注册为bean，以便进行检测。 如果使用 `new` 关键字创建 `HttpClient` 实例，则检测不起作用。

62.6.7 `UserInfoRestTemplateCustomizer` 译：62.6.7 `UserInfoRestTemplateCustomizer`

我们测试了Spring Security的 `UserInfoRestTemplateCustomizer` 。

要阻止此功能，请将 `spring.sleuth.web.client.enabled` 设置为 `false` 。

62.7 Feign 译：62.7假

默认情况下，Spring Cloud Sleuth通过 `TraceFeignClientAutoConfiguration` 提供与Feign的集成。 您可以完全通过设置禁用它 `spring.sleuth.feign.enabled` 至 `false` 。 如果这样做，则不会发生与Feign相关的仪器。

Feign仪器的一部分通过 `FeignBeanPostProcessor` 完成。 您可以通过设置禁用它 `spring.sleuth.feign.processor.enabled` 至 `false` 。 如果将其设置为 `false` ，则Spring Cloud Sleuth不会测试任何自定义Feign组件。 但是，所有默认仪器仍然存在。

62.8 Asynchronous Communication 译：62.8异步通信

62.8.1 `@Async` Annotated methods 译：62.8.1 `@Async` 注释方法

在Spring Cloud Sleuth中，我们处理与异步相关的组件，以便跟踪信息在线程之间传递。 您可以通过的值设置禁用此行为 `spring.sleuth.async.enabled` 至 `false` 。

如果您使用 `@Async` 注释您的方法，我们会自动创建一个具有以下特征的新Span：

- If the method is annotated with `@SpanName`, the value of the annotation is the Span's name.
- If the method is not annotated with `@SpanName`, the Span name is the annotated method name.
- The span is tagged with the method's class name and method name.

62.8.2 `@Scheduled` Annotated Methods译：62.8.2 `@Scheduled`注释的方法

在Spring Cloud Sleuth中，我们使用调度的方法执行，以便跟踪信息在线程之间传递。 您可以通过的值设置禁用此行为`spring.sleuth.scheduled.enabled`至`false`。

如果您使用`@Scheduled`注释您的方法，我们将自动创建具有以下特征的新跨度：

- The span name is the annotated method name.
- The span is tagged with the method's class name and method name.

如果你想跳过跨距创建一些`@Scheduled`注释类，你可以设置`spring.sleuth.scheduled.skipPattern`使用正则表达式匹配的的全名`@Scheduled`注解类。 如果一起使用`spring-cloud-sleuth-stream`和`spring-cloud-netflix-hystrix-stream`，则会为每个Hystrix度量标准创建一个跨度并发送到Zipkin。 这种行为可能很烦人。 这就是为什么，默认情况下， `spring.sleuth.scheduled.skipPattern=org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask`。

62.8.3 Executor, ExecutorService, and ScheduledExecutorService译：62.8.3执行程序，ExecutorService和ScheduledExecutorService

我们提供`LazyTraceExecutor`， `TraceableExecutorService`，并`TraceableScheduledExecutorService`。 每次提交，调用或调度新任务时，这些实现都会创建跨度。

以下示例显示了在使用`TraceableExecutorService`时如何传递跟踪信息`CompletableFuture`：

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
 // perform some logic
 return 1_000_000L;
}, new TraceableExecutorService(beanFactory, executorService,
  // 'calculateTax' explicitly names the span - this param is optional
  "calculateTax"));
```

Important

Sleuth不能与`parallelStream()`开箱即用。 如果您希望通过流传播跟踪信息，则必须使用`supplyAsync(… )`的方法，如前所述。

Customization of Executors译:执行者的自定义

有时，您需要设置`AsyncExecutor`的自定义实例。 以下示例显示如何设置此类自定义`Executor`：

```
@Configuration
@EnableAutoConfiguration
@EnableAsync
static class CustomExecutorConfig extends AsyncConfigurerSupport {

 @Autowired BeanFactory beanFactory;

 @Override public Executor getAsyncExecutor() {
  ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
  // CUSTOMIZE HERE
  executor.setCorePoolSize(7);
  executor.setMaxPoolSize(42);
  executor.setQueueCapacity(11);
  executor.setThreadNamePrefix("MyExecutor-");
  // DON'T FORGET TO INITIALIZE
  executor.initialize();
  return new LazyTraceExecutor(this.beanFactory, executor);
 }
}
```

62.9 Messaging译：62.9消息

从该部分的功能可以通过设置被禁用`spring.sleuth.messaging.enabled`与值等于属性`false`。

62.9.1 Spring Integration and Spring Cloud Stream译：62.9.1 Spring Integration和Spring Cloud Stream

Spring Cloud Sleuth与Spring Integration集成。 它为发布和订阅事件创建跨度。 要禁用弹簧集成工具，请将`spring.sleuth.integration.enabled`设置为`false`。

您可以提供`spring.sleuth.integration.patterns`模式以显式提供要包含的用于跟踪的通道的名称。 默认情况下，包括`hystrixStreamOutput`频道在内的所有频道。

Important

当使用`Executor`建立一个Spring集成`IntegrationFlow`，你必须使用的untraced版本`Executor`。 使用`TraceableExecutorService`装饰Spring集成执行程序通道会导致跨度不正确关闭。

62.9.2 Spring RabbitMq译：62.9.2 Spring RabbitMq

我们测试`RabbitTemplate`以便将跟踪标头注入消息。

要阻止此功能，请将`spring.sleuth.messaging.rabbit.enabled`设置为`false`。

62.9.3 Spring Kafka译：62.9.3春卡夫卡

我们测试Spring Kafka的`ProducerFactory`和`ConsumerFactory`以便跟踪头文件被注入创建的Spring Kafka的`Producer`和`Consumer`。

要阻止此功能，请将`spring.sleuth.messaging.kafka.enabled`设置为`false`。

我们不支持通过`@KafkaListener`注释的上下文传播。 检查this issue for more information。

62.10 Zuul 译：62.10 Zuul

我们通过丰富功能区请求和跟踪信息来衡量Zuul功能区集成。 要禁用Zuul支持，请将`spring.sleuth.zuul.enabled`属性设置为`false` 。

63. Running examples 译：63.运行示例

您可以看到部署在Pivotal Web Services中的运行示例。 请在以下链接查看它们：

- Zipkin for apps presented in the samples to the top. First make a request to Service 1 and then check out the trace in Zipkin.
- Zipkin for Brewery on PWS, its Github Code. Ensure that you've picked the lookback period of 7 days. If there are no traces, go to Presenting application and order some beers. Then check Zipkin for traces.

Part IX. Spring Cloud Consul 译：第九部分. 春云领事

Finchley.RELEASE

该项目通过自动配置和绑定到Spring Environment和其他Spring编程模型成语来为Spring Boot应用程序提供Consul集成。 通过一些简单的注释，您可以快速启用和配置应用程序内的通用模式，并使用基于Consul的组件构建大型分布式系统。 提供的模式包括服务发现，控制总线和配置。 智能路由（Zuul）和客户端负载平衡（Ribbon），断路器（Hystrix）通过与Spring Cloud Netflix集成提供。

64. Install Consul 译：64.安装领事

有关如何安装Consul的说明，请参阅 installation documentation 。

65. Consul Agent 译：65.领事代理

Consul Agent客户端必须对所有Spring云Consul应用程序都可用。 默认情况下，代理客户端预计为`localhost:8500` 。 有关如何启动代理客户端以及如何连接到Consul代理服务器集群的详细信息，请参阅Agent documentation 。 对于开发，安装consul之后，您可以使用以下命令启动Consul Agent：

```
./src/main/bash/local_run_consul.sh
```

这将在端口8500上以服务器模式启动代理，其中ui的可用位置为 http://localhost:8500

66. Service Discovery with Consul 译：66.领事服务发现

服务发现是基于微服务架构的关键原则之一。 试图手动配置每个客户端或某种形式的约定可能非常困难，并且可能非常脆弱。 领事通过HTTP API和DNS提供服务发现服务。 Spring Cloud Consul利用HTTP API进行服务注册和发现。 这并不妨碍非Spring云应用程序利用DNS接口。 领事代理服务器都在运行cluster经由通信gossip protocol并使用Raft consensus protocol 。

66.1 How to activate 译：66.1如何激活

要激活Consul Service Discovery，请使用组`org.springframework.cloud`和工件编号`spring-cloud-starter-consul-discovery`的起动器。 有关使用当前的Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

66.2 Registering with Consul 译：66.2向领事登记

当客户向Consul注册时，它提供有关其自身的元数据，例如主机和端口，ID，名称和标签。 默认情况下创建HTTP Check ，Consul每10秒`/health`端点。 如果运行状况检查失败，则将服务实例标记为关键。

示例领事客户端：

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

（即完全正常的Spring Boot应用程序）。 如果Consul客户端位于`localhost:8500`以外的其他`localhost:8500` ，则需要配置来查找客户端。 例：

application.yml。

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

> **Caution**
>
> 如果使用 Spring Cloud Consul Config ，则需要将上述值放在`bootstrap.yml`而不是`application.yml` 。

取自`Environment`的默认服务名称，实例ID和端口分别为`${spring.application.name}` ，Spring Context ID和`${server.port}` 。

要禁用Consul Discovery客户端，您可以将 `spring.cloud.consul.discovery.enabled` 设置为 `false` 。

要禁用服务注册，您可以将 `spring.cloud.consul.discovery.register` 设置为 `false` 。

66.3 HTTP Health Check译：66.3 HTTP健康检查

Consul实例的运行状况检查默认为“/ health”，这是Spring Boot Actuator应用程序中有用端点的默认位置。 如果使用非默认上下文路径或servlet路径（例如 `server.servletPath=/foo` ）或管理端点路径（例如 `management.server.servlet.context-path=/admin` ），则需要更改这些，即使是Actuator应用程序。 Consul用来检查健康端点的时间间隔也可以配置。 “10s”和“1m”分别代表10秒和1分钟。 例：

application.yml。

```
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.server.servlet.context-path}/health
        healthCheckInterval: 15s
```

您可以通过设置 `management.health.consul.enabled=false` 来禁用健康检查。

66.3.1 Metadata and Consul tags译：66.3.1元数据和Consul标签

领事还不支持服务的元数据。 Spring Cloud的 `ServiceInstance` 有 `Map<String, String> metadata` 字段。 Spring Consul使用Consul标签来近似元数据，直到Consul正式支持元数据。 表格 `key=value` 标签将被拆分并分别用作 `Map` 键和值。 没有等号 `=` 标签将被用作密钥和值。

application.yml。

```
spring:
  cloud:
    consul:
      discovery:
        tags: foo=bar, baz
```

以上配置将导致 `foo→bar` 和 `baz→baz` 的地图。

66.3.2 Making the Consul Instance ID Unique译：66.3.2使领事实例ID唯一

默认情况下，consul实例注册的ID等于其Spring应用程序上下文ID。 默认情况下，Spring应用程序上下文ID是 `${spring.application.name}:comma,separated,profiles:${server.port}` 。 对于大多数情况下，这将允许一个服务的多个实例在一台机器上运行。 如果需要进一步的唯一性，使用Spring Cloud，您可以通过在 `spring.cloud.consul.discovery.instanceId` 提供唯一标识符来覆盖此 `spring.cloud.consul.discovery.instanceId` 。 例如：

application.yml。

```
spring:
  cloud:
    consul:
      discovery:
        instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

有了这个元数据，并且在本地主机上部署了多个服务实例，随机值将在那里启动以使实例具有唯一性。 在Cloudfoundry中， `vcap.application.instance_id` 将自动填充到Spring Boot应用程序中，因此不需要随机值。

66.4 Looking up services译：66.4查找服务

66.4.1 Using Ribbon译：66.4.1使用色带

Spring Cloud支持Feign （REST客户端构建器），也支持Spring `RestTemplate` ，用于使用逻辑服务名称/ ids而不是物理URL查找服务。 Feign和发现感知RestTemplate都使用Ribbon来实现客户端负载平衡。

如果您想使用RestTemplate访问服务STORES，只需声明：

```
@LoadBalanced
@Bean
public RestTemplate loadbalancedRestTemplate() {
     new RestTemplate();
}
```

并像这样使用它（请注意我们如何使用Consul的STORES服务名称/ ID而不是完全限定的域名）：

```
@Autowired
RestTemplate restTemplate;

public String getFirstProduct() {
   return this.restTemplate.getForObject("https://STORES/products/1", String.class);
}
```

如果在多个数据中心中有Consul集群，并且您想访问另一个数据中心中的服务，则只用服务名称/ ID是不够的。 在这种情况下，您使用属性 `spring.cloud.consul.discovery.datacenters.STORES=dc-west` ，其中 `STORES` 是服务名称/ ID， `dc-west` 是STORES服务所在的数据中心。

66.4.2 Using the DiscoveryClient译：66.4.2使用DiscoveryClient

您也可以使用 `org.springframework.cloud.client.discovery.DiscoveryClient` ，它提供了一个简单的API用于发现Netflix不特定的客户端，例如

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

66.5 Consul Catalog Watch译：66.5领事目录观看

Consul Catalog Watch利用了领事能力watch services 。 Catalog Watch会阻止Consul HTTP API调用，以确定是否有任何服务发生了更改。 如果有新的服务数据发布心跳事件。

更改配置手表调用时的频率更改为`spring.cloud.consul.config.discovery.catalog-services-watch-delay` 。 默认值为1000，以毫秒为单位。 延迟是前一次调用结束后和下一次调用结束后的时间量。

禁用目录观察集 `spring.cloud.consul.discovery.catalogServicesWatch.enabled=false` 。

手表使用Spring `TaskScheduler` 安排与领事的电话。 默认情况下，它是一个 `ThreadPoolTaskScheduler` 具有 `poolSize` 的1.要改变 `TaskScheduler` ，创造型的豆 `TaskScheduler` 与命名 `ConsulDiscoveryClientConfiguration.CATALOG_WATCH_TASK_SCHEDULER_NAME` 不变。

67. Distributed Configuration with Consul译：67.与领事分布式配置

Consul提供一个用于存储配置和其他元数据的Key/Value Store 。 Spring Cloud Consul Config是Config Server and Client的替代品。 配置在特殊的“引导”阶段加载到Spring环境中。 配置默认存储在 `/config` 文件夹中。 多个 `PropertySource` 实例基于应用程序的名称和模仿Spring Cloud Config解析属性顺序的活动配置文件创建。 例如，名为“testApp”和“dev”配置文件的应用程序将创建以下属性源：

```
config/testApp,dev/
config/testApp/
config/application,dev/
config/application/
```

最具体的财产来源位于最上方，最不具体的位于底部。 `config/application` 文件夹中的属性适用于使用consul进行配置的所有应用程序。 `config/testApp` 文件夹中的属性仅适用于名为“testApp”的服务实例。

配置当前是在应用程序启动时读取的。 发送HTTP POST到 `/refresh` 将导致配置重新加载。 Section 67.3, “Config Watch”也将自动检测更改并重新加载应用程序上下文。

67.1 How to activate译：67.1如何激活

要开始使用Consul Configuration，请使用组 `org.springframework.cloud` 和工件编号 `spring-cloud-starter-consul-config` 的启动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

这将启用将设置Spring Cloud Consul配置的自动配置。

67.2 Customizing译：67.2自定义

Consul Config可以使用以下属性进行自定义：

bootstrap.yml。

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled` setting this value to "false" disables Consul Config
- `prefix` sets the base folder for configuration values
- `defaultContext` sets the folder name used by all applications
- `profileSeparator` sets the value of the separator used to separate the profile name in property sources with profiles

67.3 Config Watch译：67.3配置监视

Consul Config Watch利用领事watch a key prefix的优势 。 Config Watch会阻止Consul HTTP API调用，以确定当前应用程序是否有任何相关的配置数据已更改。 如果有新的配置数据发布刷新事件。 这相当于调用 `/refresh` 执行器端点。

更改配置手表调用时的频率更改为 `spring.cloud.consul.config.watch.delay` 。 默认值为1000，以毫秒为单位。 延迟是前一次调用结束后和下一次调用结束后的时间量。

要禁用配置观察集 `spring.cloud.consul.config.watch.enabled=false` 。

手表使用弹簧 `TaskScheduler` 安排到领事的电话。 默认情况下，它是一个 `ThreadPoolTaskScheduler` 具有 `poolSize` 的1.要改变 `TaskScheduler` ，创造型的豆 `TaskScheduler` 与命名 `ConsulConfigAutoConfiguration.CONFIG_WATCH_TASK_SCHEDULER_NAME` 不变。

67.4 YAML or Properties with Config译：67.4 YAML或带有配置的属性

使用YAML或Properties格式存储一组属性可能会更方便，而不是单独的键/值对。 将 `spring.cloud.consul.config.format` 属性设置为 `YAML` 或 `PROPERTIES` 。 例如要使用YAML：

bootstrap.yml。

```
spring:
  cloud:
    consul:
      config:
        format: YAML
```

YAML必须在领事的相应密钥中设置`data`。 使用上面的按键默认值将如下所示：

```
config/testApp,dev/data
config/testApp/data
config/application,dev/data
config/application/data
```

您可以在上面列出的任何键中存储YAML文档。

您可以使用`spring.cloud.consul.config.data-key`更改数据密钥。

67.5 git2consul with Config译：67.5 git2consul与配置

git2consul是一个Consul社区项目，可以将文件从git存储库加载到Consul的各个密钥中。 默认情况下，密钥的名称是文件的名称。 YAML和属性文件分别支持`.yml`和`.properties`文件扩展名。 将`spring.cloud.consul.config.format`属性设置为`FILES`。 例如：

bootstrap.yml。

```
spring:
  cloud:
    consul:
      config:
        format: FILES
```

鉴于以下键`/config`，该`development`轮廓和一个应用程序名称`foo`：

```
.gitignore
application.yml
bar.properties
foo-development.properties
foo-production.yml
foo.properties
master.ref
```

将创建以下属性来源：

```
config/foo-development.properties
config/foo.properties
config/application.yml
```

每个键的值需要是格式正确的YAML或属性文件。

67.6 Fail Fast译：67.6快速失败

在某些情况下（如本地开发或某些测试场景），如果consul不可用于配置，则可能不会失败。 在`bootstrap.yml`设置`spring.cloud.consul.config.failFast=false`将导致配置模块记录警告而不是抛出异常。 这将允许应用程序继续正常启动。

68. Consul Retry译：68.领事重试

如果您希望在您的应用程序启动时咨询代理偶尔可能不可用，您可以要求它在失败后继续尝试。 您需要将`spring-retry`和`spring-boot-starter-aop`添加到您的类路径中。 默认行为是重试6次，初始回退间隔为1000ms，后续回退的指数乘数为1.1。 您可以使用`spring.cloud.consul.retry.*`配置属性来配置这些属性（和其他属性）。 这适用于Spring Cloud Consul Config和Discovery注册。

要采取重试的完全控制添加`@Bean`型`RetryOperationsInterceptor` ID为“consulRetryInterceptor”。 Spring Retry有一个`RetryInterceptorBuilder`，可以很容易地创建一个。

69. Spring Cloud Bus with Consul译：与领事的 春云巴士

69.1 How to activate译：69.1如何激活

要开始使用Consul Bus，请使用组`org.springframework.cloud`和神器编号`spring-cloud-starter-consul-bus`的启动器。 有关使用当前的Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page。

请参阅 Spring Cloud Bus文档以了解可用的执行器端点以及如何发送定制消息。

70. Circuit Breaker with Hystrix译：70.断路器与老猪

应用程序可以使用Spring Cloud Netflix项目提供的Hystrix Circuit Breaker，方法是将此启动程序包含在项目pom.xml：`spring-cloud-starter-hystrix`。 Hystrix不依赖于Netflix Discovery Client。 `@EnableHystrix`注释应放置在配置类（通常是主类）上。 然后可以用`@HystrixCommand`注解方法以由断路器保护。 有关更多详细信息，请参阅the documentation。

71. Hystrix metrics aggregation with Turbine and Consul译：71.与Turbine和Consul进行Hystrix度量聚合

Turbine（由Spring Cloud Netflix项目提供）汇总了多个Hystrix度量流实例，因此仪表板可以显示聚合视图。 Turbine使用`DiscoveryClient`接口来查找相关实例。 要在Spring Cloud Consul中使用Turbine，请以类似于以下示例的方式配置Turbine应用程序：

pom.xml中。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

请注意，Turbine依赖不是一个启动器。 涡轮启动器包括对Netflix Eureka的支持。

application.yml。

```
spring.application.name: turbine
applications: consulhystrixclient
turbine:
  aggregator:
    clusterConfig: ${applications}
  appConfig: ${applications}
```

`clusterConfig`和 `appConfig`部分必须匹配，因此将逗号分隔的服务ID列表放入单独的配置属性中很有用。

Turbine.java。

```
@EnableTurbine
@SpringBootApplication
public class Turbine {
    public static void main(String[] args) {
        SpringApplication.run(DemoturbinecommonsApplication.class, args);
    }
}
```

Part X. Spring Cloud Zookeeper 译:第十部分 春云动物园管理员

该项目通过自动配置和绑定到Spring环境和其他Spring编程模型成语，为Spring Boot应用程序提供Zookeeper集成。 通过一些注释，您可以快速启用和配置应用程序内的通用模式，并使用基于Zookeeper的组件构建大型分布式系统。 提供的模式包括服务发现和配置。 与Spring Cloud Netflix集成提供智能路由（Zuul），客户端负载平衡（Ribbon）和断路器（Hystrix）。

72. Install Zookeeper 译：72.安装Zookeeper

有关如何安装Zookeeper的说明，请参阅 installation documentation 。

Spring云Zookeeper在幕后使用Apache Curator。 虽然Zookeeper 3.5.x仍然被Zookeeper开发团队视为“测试版”，但事实是，它被许多用户用于生产。 但是，Zookeeper 3.4.x也用于生产。 在Apache Curator 4.0之前，两个版本的Zookeeper都通过两个版本的Apache Curator来支持。 从Curator 4.0开始，Zookeeper的两个版本都通过相同的Curator库支持。

如果您正在与版本3.4集成，您需要更改`curator`附带的Zookeeper依赖`curator`，因此也需要更改`spring-cloud-zookeeper`。 为此，只需排除该依赖关系并添加如下所示的3.4.x版本即可。

行家。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-all</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.12</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

gradle这个。

```
compile('org.springframework.cloud:spring-cloud-starter-zookeeper-all') {
  exclude group: 'org.apache.zookeeper', module: 'zookeeper'
}
compile('org.apache.zookeeper:zookeeper:3.4.12') {
  exclude group: 'org.slf4j', module: 'slf4j-log4j12'
}
```

73. Service Discovery with Zookeeper 译：73.使用Zookeeper进行服务发现

服务发现是基于微服务架构的关键原则之一。 尝试手动配置每个客户端或某种形式的约定可能很难做到，并且可能很脆弱。 Curator （Zookeeper的Java库）通过Service Discovery Extension提供服务发现。 Spring云Zookeeper使用这个扩展来进行服务注册和发现。

73.1 Activating 译：73.1激活

包括对 `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` 的依赖可以启用设置Spring Cloud Zookeeper发现的自动配置。

对于网页功能，您仍然需要包含 `org.springframework.boot:spring-boot-starter-web` 。

Caution

在使用Zookeeper 3.4版本时，您需要按照 here所述更改包含依赖关系的方式。

73.2 Registering with Zookeeper 译：73.2注册 Zookeeper

当客户端注册到Zookeeper时，它会提供有关其自身的元数据（例如主机和端口，ID和名称）。

以下示例显示了Zookeeper客户端：

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

前面的例子是普通的Spring Boot应用程序。

如果Zookeeper位于 `localhost:2181` 以外的其他 `localhost:2181` ，则配置必须提供服务器的位置，如以下示例所示：

application.yml。

```
spring:
  cloud:
    zookeeper:
      connect-string: localhost:2181
```

Caution

如果使用 Spring Cloud Zookeeper Config ，则上例中显示的值需要位于 `bootstrap.yml` 而不是 `application.yml` 。

默认服务名称，实例ID和端口（取自 `Environment` ）分别为 `${spring.application.name}` ，Spring上下文ID和 `${server.port}` 。

在类路径上有 `spring-cloud-starter-zookeeper-discovery` 使得该应用程序进入一个Zookeeper“服务”（也就是它自己注册）和一个“客户端”（也就是说，它可以查询Zookeeper来查找其他服务）。

如果您想禁用Zookeeper发现客户端，可以将 `spring.cloud.zookeeper.discovery.enabled` 设置为 `false` 。

73.3 Using the DiscoveryClient 译：73.3使用 DiscoveryClient

Spring Cloud支持 Feign （REST客户端构建器）和 Spring `RestTemplate` ，它们使用逻辑服务名称而不是物理URL。

您还可以使用 `org.springframework.cloud.client.discovery.DiscoveryClient` ，它为Netflix不特定的发现客户端提供了一个简单的API，如以下示例所示：

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri().toString();
    }
    return null;
}
```

74. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components 译：74.使用 Spring Cloud Zookeeper和 Spring Cloud Netflix组件

Spring Cloud Netflix提供了有用的工具，无论您使用哪种 `DiscoveryClient` 实现，都可以工作。 Feign，Turbine，Ribbon和Zuul都与Spring Cloud Zookeeper合作。

74.1 Ribbon with Zookeeper 译：74.1带 Zookeeper的 Ribbon结

Spring Cloud Zookeeper提供了Ribbon的实现 `ServerList` 。 当您使用 `spring-cloud-starter-zookeeper-discovery` ，功能区默认情况下会自动配置为使用 `ZookeeperServerList` 。

75. Spring Cloud Zookeeper and Service Registry 译：75.春云动物园管理员和服务注册处

Spring云Zookeeper实现了 `ServiceRegistry` 接口，允许开发人员以编程方式注册任意服务。

所述 `ServiceInstanceRegistration` 类提供 `builder()` 方法创建 `Registration` 可以由使用对象 `ServiceRegistry` ，如图以下示例：

```
@Autowired
private ZookeeperServiceRegistry serviceRegistry;

public void registerThings() {
    ZookeeperRegistration registration = ServiceInstanceRegistration.builder()
            .defaultUriSpec()
            .address("anyUrl")
            .port(10)
            .name("/a/b/c/d/anotherservice")
            .build();
    this.serviceRegistry.register(registration);
}
```

75.1 Instance Status译：75.1实例状态

Netflix Eureka支持在服务器上注册`OUT_OF_SERVICE`实例。 这些实例不作为活动服务实例返回。 这对于诸如蓝/绿色部署之类的行为很有用。 （请注意，策展人服务发现配方不支持此行为。）利用灵活的有效负载，Spring Cloud Zookeeper通过更新某些特定元数据实现`OUT_OF_SERVICE`，然后过滤功能区`ZookeeperServerList`中的元数据。 `ZookeeperServerList`过滤掉所有不等于`UP`非空实例状态。 如果实例状态字段为空，则认为它是`UP`以实现向后兼容。 要更改实例的状态，`POST`和`OUT_OF_SERVICE`设置为`ServiceRegistry`实例状态执行程序端点，如下例所示：

```
$ http POST http://localhost:8081/service-registry status=OUT_OF_SERVICE
```

上例使用来自 https://httpie.org的`http`命令。

76. Zookeeper Dependencies译：76.动物园管理员依赖

以下主题介绍如何使用Spring云Zookeeper依赖关系：

- Section 76.1, “Using the Zookeeper Dependencies”
- Section 76.2, “Activating Zookeeper Dependencies”
- Section 76.3, “Setting up Zookeeper Dependencies”
- Section 76.4, “Configuring Spring Cloud Zookeeper Dependencies”

76.1 Using the Zookeeper Dependencies译：76.1使用Zookeeper的依赖关系

Spring Cloud Zookeeper为您提供了将应用程序的依赖关系作为属性的可能性。 作为依赖关系，您可以了解在Zookeeper中注册的其他应用程序以及您想要通过Feign（REST客户端构建器）和Spring `RestTemplate`调用的应用程序。

您还可以使用Zookeeper Dependency Watchers功能来控制和监视依赖关系的状态。

76.2 Activating Zookeeper Dependencies译：76.2激活Zookeeper依赖关系

包括对`org.springframework.cloud:spring-cloud-starter-zookeeper-discovery`的依赖可启用设置Spring Cloud Zookeeper相关性的自动配置。 即使您在属性中提供了依赖项，也可以关闭依赖项。 为此，请将`spring.cloud.zookeeper.dependency.enabled`属性设置为false（默认为`true`）。

76.3 Setting up Zookeeper Dependencies译：76.3设置Zookeeper依赖关系

考虑下面的依赖表示的例子：

application.yml。

```
spring.application.name: yourServiceName
spring.cloud.zookeeper:
  dependencies:
    newsletter:
      path: /path/where/newsletter/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.newsletter.$version+json
      version: v1
      headers:
        header1:
            - value1
        header2:
            - value2
      required: false
      stubs: org.springframework:foo:stubs
    mailing:
      path: /path/where/mailing/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.mailing.$version+json
      version: v1
      required: true
```

接下来的几节将依次介绍依赖关系的每个部分。 根属性名称是`spring.cloud.zookeeper.dependencies`。

76.3.1 Aliases译：76.3.1别名

在根属性下面，您必须将每个依赖项表示为别名。 这是由于功能区的限制，它要求将应用程序ID放在URL中。 因此，你不能通过任何复杂的路径，`/myApp/myRoute/name`）。 别名是你使用的不是名称`serviceId`为`DiscoveryClient`， `Feign`，或`RestTemplate`。

在前面的例子中，别名是`newsletter`和`mailing`。 以下示例显示了使用`newsletter`别名的`newsletter`：

```
@FeignClient("newsletter")
public interface NewsletterService {
        @RequestMapping(method = RequestMethod.GET, value = "/newsletter")
        String getNewsletters();
}
```

76.3.2 Path译：76.3.2路径

该路径由`path` YAML属性表示，并且是依赖项在Zookeeper下注册的路径。 如previous section所述，功能区在网址上运行。 因此，这条道路不符合其要求。 这就是Spring Cloud Zookeeper将别名映射到正确路径的原因。

76.3.3 Load Balancer Type译：76.3.3负载平衡器类型

负载平衡器类型由`loadBalancerType` YAML属性表示。

如果您知道调用此特定依赖关系时必须应用何种负载均衡策略，则可以在YAML文件中提供它，并自动应用它。 您可以选择以下负载平衡策略之一：

- STICKY: Once chosen, the instance is always called.
- RANDOM: Picks an instance randomly.
- ROUND_ROBIN: Iterates over instances over and over again.

76.3.4 `Content-Type` Template and Version译：76.3.4`Content-Type`模板和版本

`Content-Type`模板和版本由`contentTypeTemplate`和`version` YAML属性表示。

如果您在`Content-Type`标头中版本化API，则不希望将此标头添加到每个请求中。 另外，如果你想调用一个新版本的API，你不想漫游你的代码来搞砸API版本。 这就是为什么你可以提供`contentTypeTemplate`一个特殊的`$version`占位符。 该占位符将由`version` YAML属性的值填充。 考虑以下`contentTypeTemplate`示例：

```
application/vnd.newsletter.$version+json
```

进一步考虑以下`version`：

```
v1
```

`contentTypeTemplate`和版本的组合导致为每个请求创建`Content-Type`标头，如下所示：

```
application/vnd.newsletter.v1+json
```

76.3.5 Default Headers译：76.3.5默认标题

默认标题由YAML中的`headers`地图表示。

有时候，每次调用依赖项都需要设置一些默认头文件。 要在代码中不这样做，可以将它们设置在YAML文件中，如以下示例`headers`部分所示：

```
headers:
    Accept:
        - text/html
        - application/xhtml+xml
    Cache-Control:
        - no-cache
```

`headers`部分会导致在您的HTTP请求中添加`Accept`和`Cache-Control`标头以及适当的值列表。

76.3.6 Required Dependencies译：76.3.6必需的依赖关系

所需的依赖关系在YAML中由`required`属性表示。

如果您的某个依赖项需要在应用程序引导时启动，则可以在YAML文件中设置`required: true`属性。

如果您的应用程序在引导期间无法本地化所需的依赖项，则会引发异常，并且Spring Context无法设置。 换句话说，如果所需的依赖关系未在Zookeeper中注册，则您的应用程序无法启动。

您可以阅读更多关于Spring Cloud Zookeeper Presence Checker later in this document的信息。

76.3.7 Stubs译：76.3.7存根

您可以提供冒号分隔的路径给包含依赖项的存根的JAR，如以下示例所示：

`stubs: org.springframework:myApp:stubs`

哪里：

- `org.springframework` is the `groupId`.
- `myApp` is the `artifactId`.
- `stubs` is the classifier. (Note that `stubs` is the default value.)

因为`stubs`是默认分类器，所以前面的示例等于以下示例：

`stubs: org.springframework:myApp`

76.4 Configuring Spring Cloud Zookeeper Dependencies译：76.4配置Spring云Zookeeper依赖关系

您可以设置以下属性来启用或禁用Zookeeper相关性功能的一部分：

- `spring.cloud.zookeeper.dependencies`: If you do not set this property, you cannot use Zookeeper Dependencies.
- `spring.cloud.zookeeper.dependency.ribbon.enabled` (enabled by default): Ribbon requires either explicit global configuration or a particular one for a dependency. By turning on this property, runtime load balancing strategy resolution is possible, and you can use the `loadBalancerType` section of the Zookeeper Dependencies. The configuration that needs this property has an implementation of `LoadBalancerClient` that delegates to the `ILoadBalancer` presented in the next

bullet.

- `spring.cloud.zookeeper.dependency.ribbon.loadbalancer` (enabled by default): Thanks to this property, the custom `ILoadBalancer` knows that the part of the URI passed to Ribbon might actually be the alias that has to be resolved to a proper path in Zookeeper. Without this property, you cannot register applications under nested paths.
- `spring.cloud.zookeeper.dependency.headers.enabled` (enabled by default): This property registers a `RibbonClient` that automatically appends appropriate headers and content types with their versions, as presented in the Dependency configuration. Without this setting, those two parameters do not work.
- `spring.cloud.zookeeper.dependency.resttemplate.enabled` (enabled by default): When enabled, this property modifies the request headers of a `@LoadBalanced`-annotated `RestTemplate` such that it passes headers and content type with the version set in dependency configuration. Without this setting, those two parameters do not work.

77. Spring Cloud Zookeeper Dependency Watcher 译：77. 春云动物园管理员依赖观察员

依赖关系观察器机制可让您将侦听器注册到您的依赖关系。 实际上，该功能实现了`Observator`模式。 当依赖性发生变化时，其状态（向上或向下）可以应用一些自定义逻辑。

77.1 Activating 译：77.1激活

需要启用Spring Cloud Zookeeper依赖关系功能才能使用Dependency Watcher机制。

77.2 Registering a Listener 译：77.2注册监听器

要注册一个监听器，您必须实现一个名为`org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener`的接口并将其注册为一个bean。 界面给你一种方法：

```
void stateChanged(String dependencyName, DependencyState newState);
```

如果你想为一个特定的依赖项注册一个监听器，那么`dependencyName`就是你具体实现的鉴别器。 `newState`为您提供有关您的依赖关系是否已更改为`CONNECTED`或`DISCONNECTED` 。

77.3 Using the Presence Checker 译：77.3使用 Presence Checker

与Dependency Watcher绑定的是称为Presence Checker的功能。 它允许您在应用程序引导时提供自定义行为，根据您的依赖关系的状态做出反应。

摘要`org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier`类的默认实现是`org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier` ，其工作方式如下。

1. If the dependency is marked us `required` and is not in Zookeeper, when your application boots, it throws an exception and shuts down.
2. If the dependency is not `required`, the `org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker` logs that the dependency is missing at the `WARN` level.

因为`DefaultDependencyPresenceOnStartupVerifier`仅在没有类型为`DependencyPresenceOnStartupVerifier` bean时注册，所以可以覆盖此功能。

78. Distributed Configuration with Zookeeper 译：78.使用 Zookeeper进行分布式配置

Zookeeper提供了一个hierarchical namespace ，允许客户端存储任意数据，例如配置数据。 Spring Cloud Zookeeper Config是Config Server and Client的替代品。 配置在特殊的“启动”阶段加载到Spring环境中。 配置默认存储在`/config`名称空间中。 根据应用程序的名称和活动配置文件创建多个`PropertySource`实例，以模仿Spring Cloud Config解析属性的顺序。 例如，名称为`testApp`和`dev`配置文件的应用程序具有`testApp`创建的以下属性源：

- `config/testApp,dev`
- `config/testApp`
- `config/application,dev`
- `config/application`

最具体的财产来源位于最上方，最不具体的位于底部。 `config/application`名称空间中的属性适用于使用zookeeper进行配置的所有应用程序。 `config/testApp`名称空间中的属性仅可用于名为`testApp`的服务实例。

配置当前是在应用程序启动时读取的。 发送HTTP `POST`请求至`/refresh`会导致重新加载配置。 监视配置名称空间（Zookeeper支持的）目前尚未实现。

78.1 Activating 译：78.1激活

包括对`org.springframework.cloud:spring-cloud-starter-zookeeper-config`的依赖关系启用设置Spring Cloud Zookeeper配置的自动配置。

Caution

在使用Zookeeper 3.4版本时，需要按照 here所述更改包含依赖关系的方式。

78.2 Customizing 译：78.2定制

Zookeeper配置可以通过设置以下属性进行自定义：

bootstrap.yml。

```
spring:
  cloud:
    zookeeper:
      config:
        enabled: true
        root: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled`: Setting this value to `false` disables Zookeeper Config.
- `root`: Sets the base namespace for configuration values.
- `defaultContext`: Sets the name used by all applications.

- `profileSeparator`: Sets the value of the separator used to separate the profile name in property sources with profiles.

78.3 Access Control Lists (ACLs) 译：78.3访问控制列表（ACL）

您可以通过调用`CuratorFramework` bean的`addAuthInfo`方法来添加Zookeeper ACL的认证信息。 完成此操作的一种方法是提供您自己的`CuratorFramework` bean，如以下示例所示：

```
@BoostrapConfiguration
public class CustomCuratorFrameworkConfig {

  @Bean
  public CuratorFramework curatorFramework() {
    CuratorFramework curator = new CuratorFramework();
    curator.addAuthInfo("digest", "user:password".getBytes());
    return curator;
  }

}
```

请参阅 the ZookeeperAutoConfiguration class以查看 `CuratorFramework` bean的默认配置。

或者，您可以从取决于现有 `CuratorFramework` bean的类中添加凭据，如以下示例所示：

```
@BoostrapConfiguration
public class DefaultCuratorFrameworkConfig {

  public ZookeeperConfig(CuratorFramework curator) {
    curator.addAuthInfo("digest", "user:password".getBytes());
  }

}
```

这个bean的创建必须发生在助推阶段。 您可以注册要在此阶段运行的配置类，方法是使用`@BootstrapConfiguration`它们进行注释，并将它们包含在以`resources/META-INF/spring.factories`文件中的`org.springframework.cloud.bootstrap.BootstrapConfiguration`属性值设置的逗号分隔列表中，如以下示例所示：

资源/ META-INF / spring.factories。

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
my.project.CustomCuratorFrameworkConfig,\
my.project.DefaultCuratorFrameworkConfig
```

spring-cloud.adoc中未解决的指令 - include :: ../../../../ cli / docs / src / main / asciidoc / spring-cloud-cli.adoc []

Part XI. Spring Cloud Security 译：第十一部分。 Spring云安全

Spring Cloud Security提供了一组用于构建安全应用程序和服务的基本组件，并以最小的麻烦。 一个可以在外部（或集中）大量配置的声明模型适用于大型协作式远程组件系统的实现，通常使用中央管理服务。 在Cloud Foundry等服务平台中使用它也非常容易。 基于Spring Boot和Spring Security OAuth2，我们可以快速创建实现常见模式的系统，如单点登录，令牌中继和令牌交换。

Spring Cloud是在非限制性的Apache 2.0许可下发布的。 如果您想为本文档的这一部分做出贡献，或者如果您发现错误，请在项目github中找到源代码和问题跟踪器。

79. Quickstart 译：79快速入门

79.1 OAuth2 Single Sign On 译：79.1 OAuth2单一登录

这里是一个带有HTTP基本认证和一个用户帐户的Spring Cloud“Hello World”应用程序：

app.groovy。

```
@Grab('spring-boot-starter-security')
@Controller
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

您可以使用`spring run app.groovy`运行它并观察密码的日志（用户名是“user”）。 到目前为止，这只是Spring Boot应用程序的默认值。

这是一款带有OAuth2 SSO的Spring Cloud应用程序：

app.groovy。

```
@Controller
@EnableOAuth2Sso
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

指出不同？ 这个应用程序实际上与以前的应用程序的行为完全相同，因为它还不知道它的OAuth2鞋钉。

您可以很容易地在github上注册一个应用程序，所以如果您想在自己的域中使用生产应用程序，请尝试一下。 如果您很高兴在localhost：8080上进行测试，请在您的应用程序配置中设置这些属性：

application.yml。

```
security:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

运行上面的应用程序，它会重定向到github进行授权。 如果您已经登录github，您甚至不会注意到它已经通过身份验证。 这些凭证只有在您的应用程序在端口8080上运行时才有效。

要限制客户端在获取访问令牌时要求的范围，可以设置`security.oauth2.client.scope`（逗号分隔或YAML中的数组）。 默认情况下，作用域为空，授权服务器决定默认设置应该是什么，通常取决于它所保存的客户端注册中的设置。

上面的例子都是Groovy脚本。 如果您想在Java（或Groovy）中编写相同的代码，您需要将Spring Security OAuth2添加到类路径中（例如，请参阅sample here）。

79.2 OAuth2 Protected Resource译：79.2 OAuth2受保护的资源

您想使用OAuth2令牌保护API资源？ 这是一个简单的例子（与上面的客户端配对）：

app.groovy。

```
@Grab('spring-cloud-starter-security')
@RestController
@EnableResourceServer
class Application {

  @RequestMapping('/')
  def home() {
    [message: 'Hello World']
  }

}
```

和

application.yml。

```
security:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

80. More Detail译：80.更多细节

80.1 Single Sign On译：80.1单一登录

所有的OAuth2 SSO和资源服务器功能都在1.3版本中转移到了Spring Boot。 您可以在Spring Boot user guide中找到文档。

80.2 Token Relay译：80.2令牌中继

令牌中继是OAuth2使用者充当客户端并将传入令牌转发给传出资源请求的地方。 消费者可以是纯客户端（如SSO应用程序）或资源服务器。

80.2.1 Client Token Relay译：80.2.1客户端令牌中继

如果您的应用程序是面向OAuth2客户端的用户（即声明了`@EnableOAuth2Sso`或`@EnableOAuth2Client`），那么它在Spring Boot的请求范围内有`OAuth2ClientContext`。 您可以从此上下文创建自己的`OAuth2RestTemplate`，并自动`OAuth2ProtectedResourceDetails`，然后上下文将始终向下游转发访问令牌，并在访问令牌到期时自动刷新访问令牌。 （这些是Spring Security和Spring Boot的特性。）

春季启动（1.4.1）不会创建一个`OAuth2ProtectedResourceDetails`如果您正在使用自动`client_credentials`令牌。 在这种情况下，您需要创建自己的`ClientCredentialsResourceDetails`并使用`@ConfigurationProperties("security.oauth2.client")`配置。

80.2.2 Client Token Relay in Zuul Proxy译：80.2.2 Zuul Proxy中的客户端令牌中继

如果您的应用还包含Spring Cloud Zuul嵌入式反向代理（使用`@EnableZuulProxy`），那么您可以要求它将OAuth2访问令牌向下游转发给它所代理的服务。 因此，上面的SSO应用程序可以像这样增强：

app.groovy。

```
@Controller
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}
```

它将（除了登录用户和获取令牌之外）将认证令牌下游传递给`/proxy/*`服务。 如果这些服务使用`@EnableResourceServer`实现，那么他们将在正确的头部中获得有效的令牌。

它是如何工作的？ `@EnableOAuth2Sso`注释会在`spring-cloud-starter-security`（您可以在传统应用程序中手动执行）引发`ZuulFilter`，然后触发某个`ZuulFilter`自动配置，因为Zuul位于类路径上（通过`@EnableZuulProxy`），因此本身已激活。 filter只是从当前通过身份验证的用户提取一个访问令牌，并将其放入下游请求的请求标头中。

80.2.3 Resource Server Token Relay译：80.2.3资源服务器令牌中继

如果您的应用有`@EnableResourceServer`，则可能需要将传入令牌下游中继到其他服务。 如果您使用`RestTemplate`联系下游服务，那么这只是如何使用正确的上下文创建模板的问题。

如果您的服务使用`UserInfoTokenServices`对传入令牌进行身份验证（即使用`security.oauth2.user-info-uri`配置），则可以使用自动装配的`OAuth2ClientContext`（在身份验证过程碰到后端代码之前填充它）简单地创建`OAuth2RestTemplate`。 等效地（使用Spring Boot 1.4），您可以注入`UserInfoRestTemplateFactory`并在配置中获取其`OAuth2RestTemplate`。 例如：

MyConfiguration.java。

```
@Bean
public OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {
    return factory.getUserInfoRestTemplate();
}
```

这个剩余模板将具有与认证过滤器所使用的相同的`OAuth2ClientContext`（请求范围），因此您可以使用它来发送具有相同访问令牌的请求。

如果您的应用程序没有使用`UserInfoTokenServices`，但仍然是一个客户端（即它宣称`@EnableOAuth2Client`或者`@EnableOAuth2Sso`），然后使用Spring Security云中的任何`OAuth2RestOperations`用户从创建`@Autowired` `@OAuth2Context`也将向前令牌。 此功能默认实现为MVC处理程序拦截器，因此它只能在Spring MVC中使用。 如果您不使用MVC，则可以使用自定义过滤器或AOP拦截器来包装`AccessTokenContextRelay`以提供相同的功能。

这是一个基本的例子，显示了在其他地方创建的自动布线休息模板的使用（“foo.com”是一个资源服务器接受与周围应用程序相同的代币）：

MyController.java。

```
@Autowired
private OAuth2RestOperations restTemplate;

@RequestMapping("/relay")
public String relay() {
    ResponseEntity<String> response =
      restTemplate.getForEntity("https://foo.com/bar", String.class);
    return "Success! (" + response.getBody() + ")";
}
```

如果你不想转发令牌（这是一个有效的选择，因为你可能想扮演自己的角色，而不是发送给你令牌的客户端），那么你只需要创建自己的`OAuth2Context`而不是自动装配默认的一个。

Feign客户也会选择一个使用`OAuth2ClientContext`的拦截器（如果可用），所以他们也应该在`RestTemplate`所在的任何位置执行令牌中继。

81. Configuring Authentication Downstream of a Zuul Proxy译：81.配置Zuul代理的下游认证

您可以通过`proxy.auth.*`设置来控制`@EnableZuulProxy`下游的授权行为。 例：

application.yml。

```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

在这个例子中，“customers”服务获得一个OAuth2令牌中继，“stores”服务获得一个passthrough（授权头只是传递给下游），而“recommendations”服务的授权头被删除。 如果有令牌可用，默认行为是执行令牌中继，否则为passthru。

有关完整的详细信息，请参阅 ProxyAuthenticationProperties 。

Part XII. Spring Cloud for Cloud Foundry译：第十二部分。 Cloud Foundry的Spring Cloud

Spring Cloud for Cloudfoundry使Cloud Foundry（平台即服务）中运行Spring Cloud应用程序变得非常简单。 Cloud Foundry具有“服务”的概念，即“绑定”到应用程序的中间件，本质上为它提供包含凭据的环境变量（例如，用于服务的位置和用户名）。

`spring-cloud-cloudfoundry-commons`模块配置基于反应堆的Cloud Foundry Java客户端v 3.0，并且可以单独使用。

`spring-cloud-cloudfoundry-web`项目为Cloud Foundry中的webapps的一些增强功能提供基本支持：自动绑定到单点登录服务，并可选择启用粘性路由以进行发现。

该`spring-cloud-cloudfoundry-discovery`项目提供春季云共享的实现`DiscoveryClient`这样你就可以`@EnableDiscoveryClient`，并提供您的凭据为`spring.cloud.cloudfoundry.discovery.[username,password]`（也`*.url`如果你没有连接到 Pivotal Web Services ），然后你可以使用`DiscoveryClient`直接或通过`LoadBalancerClient`。

第一次使用它时，发现客户端可能会很慢，因为它必须从Cloud Foundry获取访问令牌。

82. Discovery译: 82.发现

这里是Cloud Foundry发现的Spring Cloud应用程序：

app.groovy。

```
@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')
@RestController
@EnableDiscoveryClient
class Application {

  @Autowired
  DiscoveryClient client

  @RequestMapping('/')
  String home() {
    'Hello from ' + client.getLocalServiceInstance()
  }

}
```

如果你运行它没有任何服务绑定：

```
$ spring jar app.jar app.groovy
$ cf push -p app.jar
```

它将在主页中显示其应用程序名称。

`DiscoveryClient`可以根据其身份验证凭据列出空间中的所有应用程序，其中空间默认为客户`DiscoveryClient`在运行的应用程序（如果有）。 如果没有配置组织或空间，则默认情况下会根据用户在Cloud Foundry中的配置文件进行配置。

83. Single Sign On译: 83.单点登录

所有的OAuth2 SSO和资源服务器功能都在1.3版本中转移到了Spring Boot。 您可以在Spring Boot user guide中找到文档。

该项目提供了从CloudFoundry服务凭证到Spring Boot功能的自动绑定。 例如，如果您拥有名为“sso”的CloudFoundry服务，并且包含“client_id”，“client_secret”和“auth_domain”的凭据，它将自动绑定到通过`@EnableOAuth2Sso`（从Spring Boot启用）启用的Spring OAuth2客户端。 该服务的名称可以使用`spring.oauth2.sso.serviceId`进行参数`spring.oauth2.sso.serviceId`。

Part XIII. Spring Cloud Contract译:第十三部分. 春天的云合同

文档作者：Adam Dudczak，MathiasDüsterhöft，Marcin Grzejszczak，Dennis Kieselhorst，JakubKubryÅ“ś，Karol Lassak，Olga Maciaszek-Sharma，MariuszSmykuÅ，a，Dave Syer，Jay Bryant

Finchley.RELEASE

84. Spring Cloud Contract译: 84.春云合同

将新功能推送到分布式系统中的新应用程序或服务时，您需要有信心。 该项目为Spring应用程序中的消费者驱动契约和服务模式提供支持（用于HTTP和基于消息的交互），涵盖了编写测试，将它们作为资产发布以及声明合同由生产者和消费者保存的一系列选项。

85. Spring Cloud Contract Verifier Introduction译: 85. Spring Cloud Contract Verifier简介

Accurest项目最初由Marcin Grzejszczak和Jakub Kubrynski（ codearte.io ）创建，

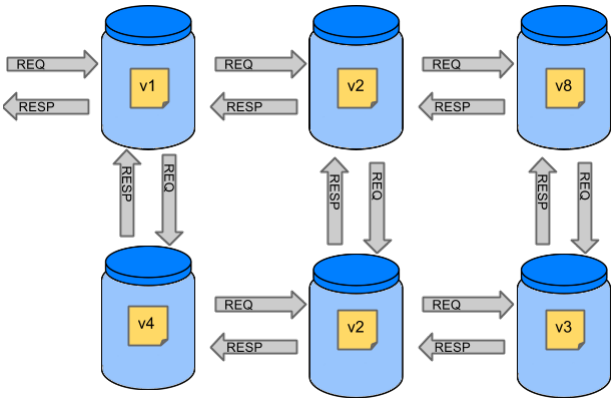
Spring Cloud Contract Verifier支持基于JVM的应用程序的消费者驱动契约（CDC）开发。 它将TDD提升到软件架构的水平。

Spring Cloud Contract Verifier附带*合同定义语言*（CDL）。 合同定义用于产生以下资源：

- JSON stub definitions to be used by WireMock when doing integration testing on the client code (*client tests*). Test code must still be written by hand, and test data is produced by Spring Cloud Contract Verifier.
- Messaging routes, if you're using a messaging service. We integrate with Spring Integration, Spring Cloud Stream, Spring AMQP, and Apache Camel. You can also set your own integrations.
- Acceptance tests (in JUnit or Spock) are used to verify if server-side implementation of the API is compliant with the contract (*server tests*). A full test is generated by Spring Cloud Contract Verifier.

85.1 Why a Contract Verifier?译: 85.1为什么选择合同验证者？

假设我们有一个由多个微服务组成的系统：



85.1.1 Testing issues 译：85.1.1测试问题

如果我们想测试左上角的应用程序以确定它是否可以与其他服务通信，我们可以做以下两件事之一：

- Deploy all microservices and perform end-to-end tests.
- Mock other microservices in unit/integration tests.

两者都有其优点，但也有很多缺点。

部署所有微服务并执行端到端测试

优点：

- Simulates production.
- Tests real communication between services.

缺点：

- To test one microservice, we have to deploy 6 microservices, a couple of databases, etc.
- The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
- They take a long time to run.
- The feedback comes very late in the process.
- They are extremely hard to debug.

在单元/集成测试中模拟其他微服务

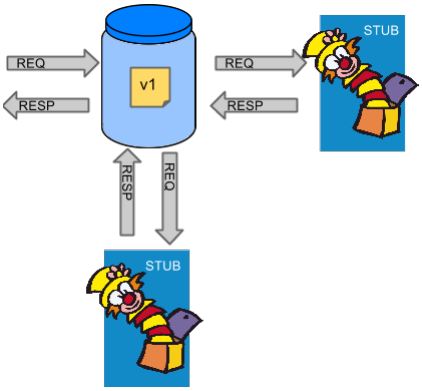
优点：

- They provide very fast feedback.
- They have no infrastructure requirements.

缺点：

- The implementor of the service creates stubs that might have nothing to do with reality.
- You can go to production with passing tests and failing production.

为了解决上述问题，创建了带有存根运行器的Spring Cloud Contract Verifier。 主要想法是给你非常快速的反馈，而不需要建立微服务的整个世界。 如果你在存根上工作，那么你需要的应用程序就是你的应用程序直接使用的应用程序。



Spring Cloud Contract Verifier让您确信您使用的存根是由您正在调用的服务创建的。 另外，如果你可以使用它们，这意味着它们是针对生产者的一方进行测试的。 总之，你可以信任那些存根。

85.2 Purposes 译：85.2目的

Spring Cloud Contract Verifier与Stub Runner的主要目的是：

- To ensure that WireMock/Messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
- To promote ATDD method and Microservices architectural style.
- To provide a way to publish changes in contracts that are immediately visible on both sides.
- To generate boilerplate test code to be used on the server side.

Important

> Spring Cloud Contract Verifier的目的不是开始在合同中写入业务功能。 假设我们有一个欺诈检查的商业用例。 如果一个用户可能因为100个不同的原因而出现欺诈行为，那么我们会假设您会创建2个合同，一个用于正面案例，另一个用于负面案例。 合同测试用于测试应用程序之间的合同，而不是模拟完整的行为。

85.3 How It Works译：85.3它是如何工作的

本节探讨Spring Cloud Contract Verifier与Stub Runner的工作原理。

85.3.1 A Three-second Tour译：85.3.1三秒游

这个非常简短的行程将介绍使用Spring Cloud合同：

- the section called “On the Producer Side”
- the section called “On the Consumer Side”

你可以找到一个更长的游览 here 。

On the Producer Side译:在制片方

首先春云合同工作，与添加文件`REST/`在任何Groovy的DSL或YAML到合同目录，这是由设置表示消息合约`contractsDslDir`财产。 默认情况下，它是`$rootDir/src/test/resources/contracts` 。

然后将Spring Cloud Contract Verifier依赖项和插件添加到您的构建文件中，如以下示例所示：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-verifier</artifactId>
 <scope>test</scope>
</dependency>
```

下面的清单显示了如何添加插件，该插件应该放在文件的build / plugins部分：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
</plugin>
```

运行`./mvnw clean install`自动生成测试，以验证应用程序是否符合所添加的合同。 默认情况下，测试生成在`org.springframework.cloud.contract.verifier.tests.` 。

由于合同描述的功能的实现尚不存在，测试失败。

要使它们通过，您必须添加处理HTTP请求或消息的正确实现。 此外，您必须为项目添加一个正确的自动生成测试的基础测试类。 该类由所有自动生成的测试扩展，并且应该包含运行它们所需的所有设置（例如`RestAssuredMockMvc`控制器设置或消息传递测试设置）。

一旦实现和测试基类就绪后，测试就会通过，应用程序和存根构件都会构建并安装在本地Maven存储库中。 现在可以合并这些更改，并且应用程序和存根工件都可以在联机存储库中发布。

On the Consumer Side译:在消费者方面

`Spring Cloud Contract Stub Runner`可以在集成测试可以用来获取模拟实际服务运行WireMock实例或通讯线路。

为此，请将依赖项添加到`Spring Cloud Contract Stub Runner` ，如以下示例所示：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
 <scope>test</scope>
</dependency>
```

您可以通过以下两种方式之一获取安装在Maven存储库中的Producer端存根：

- 通过检出Producer端存储库并添加合同并通过运行以下命令生成存根：

  ```
  $ cd local-http-server-repo
  $ ./mvnw clean install -DskipTests
  ```

  
  由于生产者方合同实施尚未到位，所以测试正在被跳过，所以自动生成的合同测试失败。

- 通过从远程存储库获取已存在的生产者服务存根。 为此，请将存根工件标识和工件存储库URL作为`Spring Cloud Contract Stub Runner`属性`Spring Cloud Contract Stub Runner` ，如下例所示：

  ```
  stubrunner:
    ids: 'com.example:http-server-dsl:+:stubs:8080'
    repositoryRoot: http://repo.spring.io/libs-snapshot
  ```

现在，您可以使用`@AutoConfigureStubRunner`注释您的测试课程。 在注释中，提供`group-id`和`artifact-id`为值`Spring Cloud Contract Stub Runner`运行协作者的存根你，如图以下示例：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, stubsMode = StubRunnerProperties.StubsMode.LOCAL)
@DirtiesContext
public class LoanApplicationServiceTests {
```


使用`REMOTE` `stubsMode`从在线存储库和下载时存根`LOCAL`为脱机工作。

现在，在您的集成测试中，您可以接收预计由协作者服务发出的HTTP响应或消息的存根版本。

85.3.2 A Three-minute Tour 译：85.3.2三分钟的游览

这个简短的介绍使用Spring Cloud Contract：

- the section called “On the Producer Side”
- the section called “On the Consumer Side”

你可以找到一个更简单的游览 here 。

On the Producer Side 译:在制片人方面

要开始使用`Spring Cloud Contract`，请将带有`REST/`消息传递合同的文件（以Groovy DSL或YAML表示）添加到由`contractsDslDir`属性设置的合同目录中。 默认情况下，它是`$rootDir/src/test/resources/contracts`。

对于HTTP存根，合约定义了针对给定请求应该返回哪种响应（考虑HTTP方法，URL，标题，状态码等）。 以下示例显示了Groovy DSL中的HTTP存根协定：

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'PUT'
  url '/fraudcheck'
  body([
      "client.id": $(regex('[0-9]{10}')),
      loanAmount: 99999
  ])
  headers {
   contentType('application/json')
  }
 }
 response {
  status OK()
  body([
      fraudCheckStatus: "FRAUD",
      "rejection.reason": "Amount too high"
  ])
  headers {
   contentType('application/json')
  }
 }
}
```

在YAML中表达的同样的合同看起来像下面的例子：

```
request:
  method: PUT
  url: /fraudcheck
  body:
    "client.id": 1234567890
    loanAmount: 99999
  headers:
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id']
        type: by_regex
        value: "[0-9]{10}"
response:
  status: 200
  body:
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers:
    Content-Type: application/json;charset=UTF-8
```

在消息传递的情况下，您可以定义：

- The input and the output messages can be defined (taking into account from and where it was sent, the message body, and the header).
- The methods that should be called after the message is received.
- The methods that, when called, should trigger a message.

以下示例显示了使用Groovy DSL表达的骆驼消息传递协定：

```
def contractDsl = Contract.make {
 label 'some_label'
 input {
  messageFrom('jms:delete')
  messageBody([
    bookName: 'foo'
  ])
  messageHeaders {
   header('sample', 'header')
  }
  assertThat('bookWasDeleted()')
 }
}
```

以下示例显示了在YAML中表达的同一合同：

```
label: some_label
input:
  messageFrom: jms:delete
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
  assertThat: bookWasDeleted()
```

然后，您可以将Spring Cloud Contract Verifier依赖项和插件添加到您的构建文件中，如以下示例所示：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-verifier</artifactId>
 <scope>test</scope>
</dependency>
```

下面的清单显示了如何添加插件，该插件应该放在文件的build / plugins部分：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
</plugin>
```

运行`./mvnw clean install`自动生成验证应用程序是否符合所添加合同的测试。 默认情况下，生成的测试在`org.springframework.cloud.contract.verifier.tests.`。

以下示例显示了HTTP协议的自动生成测试示例：

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");
        assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");
}
```

上例使用Spring的`MockMvc`来运行测试。 这是HTTP合约的默认测试模式。 但是，也可以使用JAX-RX客户端和显式HTTP调用。 （为此，请将插件的`testMode`属性分别更改为`JAX-RS`或`EXPLICIT`）。

除了默认的JUnit之外，你可以使用Spock测试，将插件`testFramework`属性设置为`Spock`。

现在，您还可以基于合同生成WireMock场景，方法是在合同文件名的开头包含一个订单号，后跟一个下划线。

以下示例显示Spock中针对邮件存根合约的自动生成的测试：

```
[source,groovy,indent=0]
```

```
given:
  ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
  \'\'\'{"bookName":"foo"}\'\'\',
  ['sample': 'header']
 )

when:
  contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:
  noExceptionThrown()
  bookWasDeleted()
```

由于合同描述的功能的实现尚不存在，测试失败。

为了让它们通过，你必须添加处理HTTP请求或消息的正确实现。 此外，您必须为项目添加一个正确的自动生成测试的基础测试类。 此类由所有自动生成的测试扩展，并应包含运行它们所需的所有设置（例如，`RestAssuredMockMvc`控制器设置或消息传递测试设置）。

一旦实现和测试基类就绪后，测试就会通过，应用程序和存根构件都会构建并安装在本地Maven存储库中。 有关将存根jar安装到本地存储库的信息将显示在日志中，如以下示例所示：

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT
[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pc
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SN
```

您现在可以合并更改并将应用程序和存根工件发布到联机存储库中。

Docker项目

为了在非JVM技术中创建应用程序时启用合同，已经创建了`springcloud/spring-cloud-contract` Docker镜像。 它包含一个自动生成HTTP协议测试的项目，并以`EXPLICIT`测试模式执行它们。 然后，如果测试通过，它会生成Wiremock存根，并可选择将它们发布到工件管理器。 为了使用该映像，您可以将合同安装到`/contracts`目录并设置一些环境变量。

On the Consumer Side 译:在消费者方面

`Spring Cloud Contract Stub Runner`可以在集成测试可以用来获取模拟实际服务运行WireMock实例或通讯线路。

要开始，请将依赖关系添加到`Spring Cloud Contract Stub Runner`：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
 <scope>test</scope>
</dependency>
```

您可以通过以下两种方式之一获取安装在Maven存储库中的Producer端存根：

- 通过检出Producer端存储库并添加合同并通过运行以下命令生成存根：

  ```
  $ cd local-http-server-repo
  $ ./mvnw clean install -DskipTests
  ```

  
  由于生产者方的合同实施还没有到位，所以测试会被跳过，所以自动生成的合同测试失败。

- 从远程存储库获取现有的生产者服务存根。 为此，请将存根工件标识和工件存储库URI作为`Spring Cloud Contract Stub Runner`属性`Spring Cloud Contract Stub Runner`，如下例所示：

  ```
  stubrunner:
    ids: 'com.example:http-server-dsl:+:stubs:8080'
    repositoryRoot: http://repo.spring.io/libs-snapshot
  ```

现在，您可以使用`@AutoConfigureStubRunner`注释您的测试课程。 在注释中，请提供`group-id`和`artifact-id`以供`Spring Cloud Contract Stub Runner`为您运行协作者的存根，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, stubsMode = StubRunnerProperties.StubsMode.LOCAL)
@DirtiesContext
public class LoanApplicationServiceTests {
```


使用`REMOTE` `stubsMode`从在线存储库和下载时存根`LOCAL`为脱机工作。

在您的集成测试中，您可以接收预计由协作者服务发出的HTTP响应的消息版本或消息。 您可以在构建日志中看到与以下内容类似的条目：

```
2016-07-19 14:22:25.403  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Desired version is + - will try to resolve the l
2016-07-19 14:22:25.438  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolving artifact com.example:http-server:jar:s
2016-07-19 14:22:25.451  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved artifact com.example:http-server:jar:st
2016-07-19 14:22:25.465  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacking stub from JAR [URI: file:/path/to/your
2016-07-19 14:22:25.475  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacked file to [/var/folders/0p/xwq47sq106x1_g
2016-07-19 14:22:27.737  INFO 41050 --- [           main] o.s.c.c.stubrunner.StubRunnerExecutor    : All stubs are now running RunningStubs [namesAnd
```

85.3.3 Defining the Contract 译:85.3.3定义合同

作为服务的消费者，我们需要确定我们想要达到的目标。 我们需要制定我们的期望。 这就是我们写合同的原因。

假设你想发送一个包含客户公司的ID和它想从我们借的金额的请求。 您还希望通过PUT方法将其发送到/ fraudcheck网址。

Groovy DSL。

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
 request { // (1)
  method 'PUT' // (2)
  url '/fraudcheck' // (3)
  body([ // (4)
      "client.id": $(regex('[0-9]{10}')),
      loanAmount: 99999
  ])
  headers { // (5)
   contentType('application/json')
  }
 }
 response { // (6)
  status OK() // (7)
  body([ // (8)
      fraudCheckStatus: "FRAUD",
      "rejection.reason": "Amount too high"
  ])
  headers { // (9)
   contentType('application/json')
  }
 }
}

/*
From the Consumer perspective, when shooting a request in the integration test:

(1) - If the consumer sends a request
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `client.id` that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the response will be sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

(1) - A request will be sent to the producer
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the test will assert if the response has been sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` matching `application/json.*`
 */
```

YAML。

```
request: # (1)
  method: PUT # (2)
  url: /fraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body:  # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json;charset=UTF-8


#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/fraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/fraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`
```

85.3.4 Client Side译：85.3.4客户端

Spring Cloud Contract会生成存根，您可以在客户端测试期间使用它。 你会得到一个模拟服务的正在运行的WireMock实例/消息传递路由。 您希望为该实例提供适当的存根定义。

在某个时间点，您需要向欺诈检测服务发送请求。

```
ResponseEntity<FraudServiceResponse> response =
  restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,
    new HttpEntity<>(request, httpHeaders),
    FraudServiceResponse.class);
```

使用`@AutoConfigureStubRunner`注释您的测试课程。 在注释中，为Stub Runner提供组标识和工件标识以下载协作者的存根。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, stubsMode = StubRunnerProperties.StubsMode.LOCAL)
@DirtiesContext
public class LoanApplicationServiceTests {
```

之后，在测试期间，Spring Cloud Contract会自动在Maven存储库中找到存根（模拟真实服务）并将其公开在配置的（或随机）端口上。

85.3.5 Server Side译：85.3.5服务器端

既然你正在开发你的存根，你需要确定它实际上类似于你的具体实现。 你不能有一种情况，即你的存根以一种方式行为，你的应用程序以不同的方式行为，特别是在生产中。

为了确保您的应用程序按照您在存根中定义的方式运行，将从您提供的存根中生成测试。

自动生成的测试或多或少看起来像这样：

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");
        assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");
}
```

85.4 Step-by-step Guide to Consumer Driven Contracts (CDC)译：85.4消费者驱动合同（CDC）分步指南

考虑一个欺诈检测和贷款发放流程的例子。 业务场景是这样的，我们想向人们发放贷款，但不希望他们从我们这里偷走。 我们系统的当前实施向每个人提供贷款。

假定`Loan Issuance`是`Fraud Detection`服务器的客户端。 在目前的冲刺中，我们必须发展一项新功能：如果客户想要借钱太多，那么我们将客户标记为欺诈。

技术评论 - 欺诈检测有`artifact-id`的`http-server`，而贷款发行有一个神器id为`http-client`，并且都有`group-id`的`com.example`。

社交评论 - 客户和服务器开发团队需要直接沟通，并在整个过程中讨论变化。 CDC是关于沟通的。

server side code is available here和 the client code here 。

在这种情况下，生产者拥有合同。 在物理上，所有的合同都在生产者的仓库中。

85.4.1 Technical note译：85.4.1技术说明

如果使用 **SNAPSHOT** / **Milestone** / **Release Candidate**版本，请将以下部分添加到您的版本中：

Maven的。

```
<repositories>
 <repository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
</pluginRepositories>
```

摇篮。

```
repositories {
 mavenCentral()
 mavenLocal()
 maven { url "http://repo.spring.io/snapshot" }
 maven { url "http://repo.spring.io/milestone" }
 maven { url "http://repo.spring.io/release" }
}
```

85.4.2 Consumer side (Loan Issuance) 译：85.4.2消费者方（贷款发行）

作为贷款发行服务（欺诈检测服务器的消费者）的开发者，您可以执行以下步骤：

1. Start doing TDD by writing a test for your feature.
2. Write the missing implementation.
3. Clone the Fraud Detection service repository locally.
4. Define the contract locally in the repo of Fraud Detection service.
5. Add the Spring Cloud Contract Verifier plugin.
6. Run the integration tests.
7. File a pull request.
8. Create an initial implementation.
9. Take over the pull request.
10. Write the missing implementation.
11. Deploy your app.
12. Work online.

通过为您的功能编写测试来开始进行TDD。

```
@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
 // given:
 LoanApplication application = new LoanApplication(new Client("1234567890"),
   99999);
 // when:
 LoanApplicationResult loanApplication = service.loanApplication(application);
 // then:
 assertThat(loanApplication.getLoanApplicationStatus())
   .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_REJECTED);
 assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");
}
```

假设你已经写了一个新功能的测试。 如果收到大额贷款申请，系统应该用一些描述拒绝该贷款申请。

编写缺少的实现。

在某个时间点，您需要向欺诈检测服务发送请求。 假设您需要发送包含客户ID和客户想要借入的金额的请求。 您想通过`PUT`方法将其发送到`/fraudcheck`网址。

```
ResponseEntity<FraudServiceResponse> response =
  restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,
    new HttpEntity<>(request, httpHeaders),
    FraudServiceResponse.class);
```

为简单起见，欺诈检测服务的端口设置为`8080`，应用程序在`8090`上`8090`。

如果此时开始测试，则会中断测试，因为当前没有服务在端口`8080`上运行。

在本地克隆欺诈检测服务库。

您可以先玩弄服务器端合同。 为此，您必须先克隆它。

```
$ git clone https://your-git-server.com/server-side.git local-http-server-repo
```

在欺诈检测服务的回购中本地定义合同。

作为消费者，您需要确定您想要达到的目标。 你需要制定你的期望。 为此，请写下以下合同：

Important

将合同放在`src/test/resources/contracts/fraud`文件夹下。 `fraud`文件夹非常重要，因为生产者的测试基类名称引用该文件夹。

Groovy DSL。

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
 request { // (1)
  method 'PUT' // (2)
  url '/fraudcheck' // (3)
  body([ // (4)
      "client.id": $(regex('[0-9]{10}')),
      loanAmount: 99999
  ])
  headers { // (5)
   contentType('application/json')
  }
 }
 response { // (6)
  status OK() // (7)
  body([ // (8)
      fraudCheckStatus: "FRAUD",
      "rejection.reason": "Amount too high"
  ])
  headers { // (9)
   contentType('application/json')
  }
 }
}

/*
From the Consumer perspective, when shooting a request in the integration test:

(1) - If the consumer sends a request
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `client.id` that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the response will be sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

(1) - A request will be sent to the producer
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the test will assert if the response has been sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` matching `application/json.*`
 */
```

YAML。

```
request: # (1)
  method: PUT # (2)
  url: /fraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body:  # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json;charset=UTF-8


#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/fraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/fraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`
```

YML合同非常简单。 但是，当您查看使用静态类型的Groovy DSL编写的合同时 - 您可能想知道`value(client(… ), server(… ))`部件是什么。 通过使用这种表示法，Spring Cloud Contract允许您定义JSON块，URL等动态的部分。 如果是标识符或时间戳，则不需要硬编码值。 你想允许一些不同的值范围。 要启用值范围，可以为消费者端设置与这些值匹配的正则表达式。 您可以通过地图符号或带插值的字符串提供主体。 Consult the docs for more information.我们强烈建议使用地图符号！

您必须了解地图符号才能设置合同。 请阅读Groovy docs regarding JSON 。

以前显示的合同是双方达成的协议：

- 如果一个HTTP请求被全部发送
  - a `PUT` method on the `/fraudcheck` endpoint,
  - a JSON body with a `client.id` that matches the regular expression `[0-9]{10}` and `loanAmount` equal to `99999`,
  - and a `Content-Type` header with a value of `application/vnd.fraud.v1+json`,
- 然后HTTP消息发送给消费者
  - has status `200`,
  - contains a JSON body with the `fraudCheckStatus` field containing a value `FRAUD` and the `rejectionReason` field having value `Amount too high`,
  - and a `Content-Type` header with a value of `application/vnd.fraud.v1+json`.

一旦准备在集成测试中检查API，您需要在本地安装存根。

添加Spring Cloud Contract Verifier插件。

我们可以添加Maven或Gradle插件。 在这个例子中，你会看到如何添加Maven。 首先，添加`Spring Cloud Contract` BOM。

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>${spring-cloud-dependencies.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

接下来，添加`Spring Cloud Contract Verifier` Maven插件

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
 <configuration>
  <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
 </configuration>
</plugin>
```

自插件添加完成后，您将获得`Spring Cloud Contract Verifier`功能，这些功能来自所提供的合同：

- generate and run tests
- produce and install stubs

你不想生成测试，因为你作为消费者只想玩这些存根。 您需要跳过测试生成和执行。 当你执行：

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```

在日志中，你会看到类似这样的内容：

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT
[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT
```

以下行非常重要：

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT
```

它确认`http-server`的存根已安装在本地存储库中。

运行集成测试。

为了从自动存根下载的Spring Cloud Contract Stub Runner功能中获益，您必须在消费者方项目（`Loan Application service`）中执行以下`Loan Application service`：

添加`Spring Cloud Contract` BOM：

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>${spring-cloud-dependencies.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

将依赖关系添加到`Spring Cloud Contract Stub Runner`：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
 <scope>test</scope>
</dependency>
```

使用`@AutoConfigureStubRunner`注释您的测试课程。 在注释中，为Stub Runner提供`group-id`和`artifact-id`以下载协作者的存根。（可选步骤）因为您正在与协作者离线玩，您还可以提供离线工作开关（`StubRunnerProperties.StubsMode.LOCAL`）。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, stubsMode = StubRunnerProperties.StubsMode.LOCAL)
@DirtiesContext
public class LoanApplicationServiceTests {
```

现在，当你运行你的测试时，你会看到类似这样的东西：

```
2016-07-19 14:22:25.403  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Desired version is + - will try to resolve the l
2016-07-19 14:22:25.438  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolving artifact com.example:http-server:jar:s
2016-07-19 14:22:25.451  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved artifact com.example:http-server:jar:st
2016-07-19 14:22:25.465  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacking stub from JAR [URI: file:/path/to/your
2016-07-19 14:22:25.475  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacked file to [/var/folders/0p/xwq47sq106x1_g
2016-07-19 14:22:27.737  INFO 41050 --- [           main] o.s.c.c.stubrunner.StubRunnerExecutor    : All stubs are now running RunningStubs [namesAnd
```

此输出表明存根亚军已发现您的存根和启动服务器与组ID您的应用程序`com.example`，工件ID `http-server`配版`0.0.1-SNAPSHOT`存根，并与`stubs`分类端口`8080`。

提出拉取请求。

到目前为止你所做的是一个迭代过程。 您可以玩合同，在当地安装，并在消费方工作，直到合同按照您的意愿工作。

一旦您对结果和测试通过感到满意，请向服务器端发布拉取请求。 目前，消费者方面的工作已经完成。

85.4.3 Producer side (Fraud Detection server) 译：85.4.3生产者端（欺诈检测服务器）

作为欺诈检测服务器（贷款发行服务的服务器）的开发者：

创建一个初始实现。

提醒一下，你可以在这里看到最初的实现：

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

接管拉请求。

```
$ git checkout -b contract-change-pr master
$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr
```

您必须添加自动生成的测试所需的依赖关系：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-verifier</artifactId>
 <scope>test</scope>
</dependency>
```

在Maven插件的配置中，传递`packageWithBaseClasses`属性

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
 <configuration>
  <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
 </configuration>
</plugin>
```

Important

此示例通过设置`packageWithBaseClasses`属性使用“基于约定”的命名。 这样做意味着最后两个软件包合并为基础测试类的名称。 在我们的案例中，合同被置于`src/test/resources/contracts/fraud`。 由于您没有从`contracts`文件夹开始的两个软件包，请仅选择一个，该文件夹应为`fraud`。 添加`Base`后缀并大写`fraud`。 这给你`FraudBase`测试类名称。

所有生成的测试都会扩展该类。 在那里，你可以设置你的Spring Context或其他必要的东西。 在这种情况下，请使用Rest Assured MVC启动服务器端`FraudDetectionController`。

```
package com.example.fraud;

import org.junit.Before;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

public class FraudBase {
 @Before
 public void setup() {
  RestAssuredMockMvc.standaloneSetup(new FraudDetectionController(),
    new FraudStatsController(stubbedStatsProvider()));
 }

 private StatsProvider stubbedStatsProvider() {
  return fraudType -> {
   switch (fraudType) {
   case DRUNKS:
    return 100;
   case ALL:
    return 200;
   }
   return 0;
  };
 }

 public void assertThatRejectionReasonIsNull(Object rejectionReason) {
  assert rejectionReason == null;
 }
}
```

现在，如果你运行`./mvnw clean install`，你会得到这样的东西：

```
Results :

Tests in error:
  ContractVerifierTest.validate_shouldMarkClientAsFraud:32 » IllegalState Parsed...
```

出现此错误是因为您有一个新的合同从中生成测试，并且由于您尚未实现此功能而失败。 自动生成的测试将如下所示：

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");
        assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");
}
```

如果您使用Groovy DSL，您可以看到， `value(consumer(… ), producer(… ))` 块中存在的所有合同 `producer()` 部分 `value(consumer(… ), producer(… ))` 注入测试。 在使用YAML的情况下， `matchers` 部分也 `response` 。

请注意，在制作方，你也在做TDD。 期望以测试的形式表达。 此测试使用合同中定义的URL，标题和正文向我们自己的应用程序发送请求。 它也期待在响应中精确定义的值。 换句话说，你有 `red` 的一部分 `red` ， `green` ，并 `refactor` 。 现在是将 `red` 转换为 `red` 的 `green` 。

编写缺少的实现。

因为您知道预期的输入和预期输出，所以您可以编写缺少的实现：

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
if (amountGreaterThanThreshold(fraudCheck)) {
 return new FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT_TOO_HIGH);
}
return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

当你再次执行 `./mvnw clean install` 时，测试通过。 由于 `Spring Cloud Contract Verifier` 插件将测试添加到 `generated-test-sources` ，因此您可以从IDE运行这些测试。

部署您的应用程序。

完成工作后，您可以部署更改。 首先，合并分支：

```
$ git checkout master
$ git merge --no-ff contract-change-pr
$ git push origin master
```

您的CI可能运行类似 `./mvnw clean deploy` 的应用程序和存根工件。

85.4.4 Consumer Side (Loan Issuance) Final Step译：85.4.4消费方（贷款发行）最后一步

作为贷款发行服务（欺诈检测服务器的消费者）的开发者：

将分支合并到主。

```
$ git checkout master
$ git merge --no-ff contract-change-pr
```

在线工作。

现在，您可以禁用Spring Cloud Contract Stub Runner的脱机工作，并指出存根与存根的位置。 此时服务器端的存根将自动从Nexus / Artifactory下载。 您可以将值 `stubsMode` 设置为 `REMOTE` 。 以下代码显示了通过更改属性来实现同一事物的示例。

```
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: http://repo.spring.io/libs-snapshot
```

就是这样！

85.5 Dependencies译：85.5相关性

添加依赖关系的最佳方法是使用适当的 `starter` 依赖项。

对于 `stub-runner` ，请使用 `spring-cloud-starter-stub-runner` 。 当您使用插件时，请添加 `spring-cloud-starter-contract-verifier` 。

85.6 Additional Links译：85.6其他链接

以下是与Spring Cloud Contract Verifier和Stub Runner相关的一些资源。 请注意，有些可能已经过时，因为Spring Cloud Contract Verifier项目正在不断发展。

85.6.1 Spring Cloud Contract video译：85.6.1 Spring Cloud合约视频

您可以查看华沙JUG关于Spring Cloud合同的视频：

85.6.2 Readings译：85.6.2读数

- Slides from Marcin Grzejszczak's talk about Accurest
- Accurest related articles from Marcin Grzejszczak's blog
- Spring Cloud Contract related articles from Marcin Grzejszczak's blog
- Groovy docs regarding JSON

85.7 Samples译：85.7样品

你可以在 samples找到一些样品。

86. Spring Cloud Contract FAQ译：86. Spring Cloud合同FAQ

86.1 Why use Spring Cloud Contract Verifier and not X ?译：86.1为什么使用Spring Cloud Contract Verifier而不是X?

目前Spring Cloud Contract是基于JVM的工具。 所以当你已经为JVM创建软件时，这可能是你的第一个选择。 这个项目有很多非常有趣的功能，但其中不少确实让Spring Cloud Contract Verifier在消费者驱动契约（CDC）工具的“市场”中脱颖而出。 其中最有趣的是：

- Possibility to do CDC with messaging
- Clear and easy to use, statically typed DSL
- Possibility to copy paste your current JSON file to the contract and only edit its elements
- Automatic generation of tests from the defined Contract
- Stub Runner functionality - the stubs are automatically downloaded at runtime from Nexus / Artifactory
- Spring Cloud integration - no discovery service is needed for integration tests
- Spring Cloud Contract integrates with Pact out of the box and provides easy hooks to extend its functionality
- Via Docker adds support for any language & framework used

86.2 I don’t want to write a contract in Groovy!译：86.2我不想在Groovy中编写合同！

没问题。 你可以在YAML写一份合约！

86.3 What is this value(consumer(), producer()) ?译：86.3这个值是什么（consumer（），producer（））？

与存根相关的最大挑战之一是其可重用性。 只有当它们能被广泛使用时，它们才能达到其目的。 通常很难做到的是请求/响应元素的硬编码值。 例如日期或ID。 想象一下下面的JSON请求

```
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

和JSON响应

```
{
    "time" : "2016-10-10 21:10:15",
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

想象一下，通过改变系统中的时钟或提供数据提供者的存根实现，需要设置`time`字段的适当值（让我们假设该内容由数据库生成）所需的痛苦。 这与`id` 。 你会创建UUID生成器的存根实现吗？ 很有意义.....“

因此，作为消费者，您希望发送与任何时间或任何UUID相匹配的请求。 这样你的系统将照常工作 - 将生成数据，而且你不必将任何东西都删除。 让我们假设在上述JSON的情况下，最重要的部分是`body`字段。 您可以专注于此并为其他字段提供匹配。 换句话说，你希望存根工作是这样的：

```
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "foo"
}
```

就消费者的回应而言，您需要一个可以操作的具体价值。 所以这样的JSON是有效的

```
{
    "time" : "2016-10-10 21:10:15",
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

正如你在前面的章节中看到的那样，我们从合约生成测试。 所以从生产者的角度来看，情况看起来很不一样。 我们正在解析所提供的合同，并且在测试中我们希望向您的终端发送真正的请求。 因此，对于生产者请求的情况，我们无法进行任何匹配。 我们需要生产者的后端可以工作的具体价值。 这样的JSON将是一个有效的：

```
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

另一方面，从合同有效性的角度来看，答复不一定必须包含`time`或`id`具体值。 让我们说你在生产者方面产生了这些 - 再一次，你必须做大量的存根以确保你总是返回相同的值。 这就是为什么从制片人的角度来看，您可能需要的是以下回应：

```
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "bar"
}
```

那么你如何为消费者提供一次匹配器，并为生产者提供一个具体价值，反之亦然？ 在Spring Cloud Contract中，我们允许您提供**动态值** 。 这意味着它可以在通信双方中有所不同。 您可以传递这些值：

通过`value`方法

```
value(consumer(...), producer(...))
value(stub(...), test(...))
value(client(...), server(...))
```

或者使用 `$()` 方法

```
$(consumer(...), producer(...))
$(stub(...), test(...))
$(client(...), server(...))
```

您可以在 Contract DSL section中阅读更多关于此的 信息 。

致电 `value()` 或 `$()` 告知Spring Cloud Contract您将传递一个动态值。 在 `consumer()` 方法中，您传递应在消费者端使用的值（在生成的存根中）。 在 `producer()` 方法中，您传递应在生产者端使用的值（在生成的测试中）。

如果一方面你已经通过正则表达式，而你没有通过另一方，那么另一方会自动生成。

大多数情况下，您将与 `regex` 辅助方法一起使用该方法。 例如 `consumer(regex('[0-9]{10}'))` 。

综上所述，上述场景的合同看起来或多或少像这样（时间和UUID的正则表达式被简化并且很可能是无效的，但我们希望在这个例子中保持简单）：

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
     method 'GET'
     url '/someUrl'
     body([
         time : value(consumer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
         id: value(consumer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
         body: "foo"
     ])
    }
   response {
    status OK()
    body([
         time : value(producer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
         id: value([producer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
         body: "bar"
     ])
   }
}
```

Important

请阅读 Groovy docs related to JSON以了解如何正确构建请求/响应机构。

86.4 How to do Stubs versioning? 译：86.4如何做存根版本控制？

86.4.1 API Versioning 译：86.4.1 API版本控制

让我们试着回答一个问题，即版本化的真正含义。 如果你指的是API版本，那么有不同的方法。

- use Hypermedia, links and do not version your API by any means
- pass versions through headers / urls

我不会试图回答哪个方法更好的问题。 无论适合您的需求，并允许您创造商业价值，都应该挑选出来。

让我们假设你的版本是你的API。 在这种情况下，您应该提供您支持的许多版本的合同。 您可以为每个版本创建一个子文件夹，或将其附加到合约名称 - 无论您更适合您。

86.4.2 JAR versioning 译：86.4.2 JAR版本控制

如果通过版本控制来指代包含存根的JAR版本，那么基本上有两种主要方法。

假设您正在进行持续交付/部署，这意味着您每次穿过管道时都会生成一个新版本的jar，并且该jar可以随时投入生产。 例如，你的jar版本看起来像这样（它建立在20.10.2016 20:15:21）：

```
1.0.0.20161020-201521-RELEASE
```

在这种情况下，您生成的存根jar将如下所示。

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```

在这种情况下，当引用存根提供最新版本的存根时，应该在 `application.yml` 或 `@AutoConfigureStubRunner` 内。 您可以通过传递 `+` 标志来完成此 `+` 。 例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

但是，如果版本控制是固定的（例如 `1.0.4.RELEASE` 或 `2.1.1` ），则必须设置jar版本的具体值。 2.1.1的例子。

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```

86.4.3 Dev or prod stubs 译：86.4.3 Dev或prod存根

您可以操作分类器以针对其他服务的存根的当前开发版本或已部署到生产的存根运行测试。 如果您在生产部署时使用 `prod-stubs` 分类器更改构建以部署存根，那么您可以使用dev存根和带有存根的存根运行测试。

使用开发版存根的测试示例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

使用生产版本存根的测试示例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})
```

您也可以通过部署管道中的属性传递这些值。

86.5 Common repo with contracts译：86.5与合同共同回购

除了与制片人合作以外，存储合同的另一种方式是将他们保存在一个共同的地方。 它可能与消费者无法克隆生产者代码的安全问题有关。 此外，如果您将合同保存在一个地方，那么作为制片人，您将知道您拥有多少消费者以及哪些消费者会因您的本地更改而中断。

86.5.1 Repo structure译：86.5.1回购结构

莱塔€™的假设我们有一个具有坐标制片`com.example:server`和3名消费者： `client1` ， `client2` ， `client3` 。 然后在具有常见合同的仓库中，您将拥有以下设置（您可以结算here ）：

```
├── com
│   └── example
│       └── server
│           ├── client1
│           │   └── expectation.groovy
│           ├── client2
│           │   └── expectation.groovy
│           ├── client3
│           │   └── expectation.groovy
│           └── pom.xml
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── assembly
        └── contracts.xml
```

正如你所看到的斜杠分隔GROUPID下`/`工件ID文件夹（ `com/example/server` ），你有3名消费者（预期`client1` ， `client2`和`client3` ）。 期望是本文档中描述的标准Groovy DSL合同文件。 该存储库必须生成一个映射到repo内容的JAR文件。

`server`文件夹中的 `pom.xml`示例。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example</groupId>
 <artifactId>server</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <name>Server Stubs</name>
 <description>POM used to install locally stubs for consumer side</description>

 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.3.RELEASE</version>
  <relativePath />
 </parent>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.8</java.version>
  <spring-cloud-contract.version>2.0.1.BUILD-SNAPSHOT</spring-cloud-contract.version>
  <spring-cloud-dependencies.version>Finchley.BUILD-SNAPSHOT</spring-cloud-dependencies.version>
  <excludeBuildFolders>true</excludeBuildFolders>
 </properties>

 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>${spring-cloud-dependencies.version}</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
 </dependencyManagement>

 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
     <!-- By default it would search under src/test/resources/ -->
     <contractsDirectory>${project.basedir}</contractsDirectory>
    </configuration>
   </plugin>
  </plugins>
 </build>

 <repositories>
  <repository>
   <id>spring-snapshots</id>
   <name>Spring Snapshots</name>
```

```
   <url>https://repo.spring.io/snapshot</url>
   <snapshots>
    <enabled>true</enabled>
   </snapshots>
  </repository>
  <repository>
   <id>spring-milestones</id>
   <name>Spring Milestones</name>
   <url>https://repo.spring.io/milestone</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </repository>
  <repository>
   <id>spring-releases</id>
   <name>Spring Releases</name>
   <url>https://repo.spring.io/release</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </repository>
 </repositories>
 <pluginRepositories>
  <pluginRepository>
   <id>spring-snapshots</id>
   <name>Spring Snapshots</name>
   <url>https://repo.spring.io/snapshot</url>
   <snapshots>
    <enabled>true</enabled>
   </snapshots>
  </pluginRepository>
  <pluginRepository>
   <id>spring-milestones</id>
   <name>Spring Milestones</name>
   <url>https://repo.spring.io/milestone</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </pluginRepository>
  <pluginRepository>
   <id>spring-releases</id>
   <name>Spring Releases</name>
   <url>https://repo.spring.io/release</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </pluginRepository>
 </pluginRepositories>

</project>
```

正如你所看到的，除了Spring Cloud Contract Maven插件之外，没有任何依赖关系。 这些poms是消费者运行`mvn clean install -DskipTests`本地安装生产者项目的存根所必需的。

根文件夹中的 `pom.xml`可能如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example.standalone</groupId>
 <artifactId>contracts</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <name>Contracts</name>
 <description>Contains all the Spring Cloud Contracts, well, contracts. JAR used by the producers to generate tests and stubs</description>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

 <build>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
     <execution>
      <id>contracts</id>
      <phase>prepare-package</phase>
      <goals>
       <goal>single</goal>
      </goals>
      <configuration>
       <attach>true</attach>
       <descriptor>${basedir}/src/assembly/contracts.xml</descriptor>
       <!-- If you want an explicit classifier remove the following line -->
       <appendAssemblyId>false</appendAssemblyId>
      </configuration>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>

</project>
```

它使用汇编插件来构建带有所有合同的JAR。 这种设置的例子在这里：

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
 <id>project</id>
 <formats>
  <format>jar</format>
 </formats>
 <includeBaseDirectory>false</includeBaseDirectory>
 <fileSets>
  <fileSet>
   <directory>${project.basedir}</directory>
   <outputDirectory>/</outputDirectory>
   <useDefaultExcludes>true</useDefaultExcludes>
   <excludes>
    <exclude>**/${project.build.directory}/**</exclude>
    <exclude>mvnw</exclude>
    <exclude>mvnw.cmd</exclude>
    <exclude>.mvn/**</exclude>
    <exclude>src/**</exclude>
   </excludes>
  </fileSet>
 </fileSets>
</assembly>
```

86.5.2 Workflow译：86.5.2工作流程

工作流程与`Step by step guide to CDC`的工作流程类似。 唯一的区别是生产者不再拥有合同。 因此，消费者和生产者必须在公共仓库中处理共同合同。

86.5.3 Consumer译：86.5.3消费者

当 **消费者**希望离线处理合同时，消费者团队克隆公共存储库，转到所需的生产者文件夹（例如 `com/example/server` ），并运行 `mvn clean install -DskipTests`以在本地安装从该转换器转换而来的存根合同。

你需要有 Maven installed locally

86.5.4 Producer译：86.5.4生产者

作为一个 **生产者**，它足以改变Spring Cloud Contract Verifier来提供包含合同的JAR的URL和依赖关系：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <configuration>
  <contractsMode>REMOTE</contractsMode>
  <contractsRepositoryUrl>http://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>
  <contractDependency>
   <groupId>com.example.standalone</groupId>
   <artifactId>contracts</artifactId>
  </contractDependency>
 </configuration>
</plugin>
```

有了这个设置与GROUPID的JAR `com.example.standalone`和artifactId的`contracts`将从下载`http://link/to/your/nexus/or/artifactory/or/sth` 。 然后它将被解压缩到一个本地临时文件夹中， `com/example/server`下的`com/example/server`将被挑选为用于生成测试和存根的合同。 由于这一惯例，生产者团队将知道当一些不兼容的变更完成后哪些消费者团队将被打破。

剩下的流程看起来是一样的。

86.5.5 How can I define messaging contracts per topic not per producer?译：86.5.5如何根据主题定义消息合约而非每个生产者？

为了避免通用回购中的消息传递合同重复，当很少生产者将消息写入一个主题时，我们可以创建该结构，其余合同将放置在每个生产者的文件夹和每个主题的文件夹中的消息传递合约中。

For Maven Project译：Maven项目

为了能够在生产者端工作，我们可以做以下事情（全部通过Maven插件）：

- Add common repo dependency to your classpath:

```
<dependency>
   <groupId>com.example</groupId>
   <artifactId>common-repo</artifactId>
   <version>${common-repo.version}</version>
</dependency>
```

- Download the JAR with the contracts and unpack the JAR to target:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-dependency-plugin</artifactId>
   <version>3.0.0</version>
   <executions>
      <execution>
         <id>unpack-dependencies</id>
         <phase>process-resources</phase>
         <goals>
            <goal>unpack</goal>
         </goals>
         <configuration>
            <artifactItems>
               <artifactItem>
                  <groupId>com.example</groupId>
                  <artifactId>common-repo</artifactId>
                  <type>jar</type>
                  <overWrite>false</overWrite>
                  <outputDirectory>${project.build.directory}/contracts</outputDirectory>
               </artifactItem>
            </artifactItems>
         </configuration>
      </execution>
   </executions>
</plugin>
```

- Rip out all the folders we're not interested in:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-antrun-plugin</artifactId>
   <version>1.8</version>
   <executions>
      <execution>
         <phase>process-resources</phase>
         <goals>
            <goal>run</goal>
         </goals>
         <configuration>
            <tasks>
               <delete includeemptydirs="true">
                  <fileset dir="${project.build.directory}/contracts">
                     <include name="**/*" />
                     <!--Producer artifactId-->
                     <exclude name="**/${project.artifactId}/**" />
                     <!--List of the supported topics-->
                     <exclude name="**/${first-topic}/**" />
                     <exclude name="**/${second-topic}/**" />
                  </fileset>
               </delete>
            </tasks>
         </configuration>
      </execution>
   </executions>
</plugin>
```

- Run the contract plugin by pointing to the contracts to the folder under target:

```
<plugin>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-contract-maven-plugin</artifactId>
   <version>${spring-cloud-contract.version}</version>
   <extensions>true</extensions>
   <configuration>
      <packageWithBaseClasses>com.example</packageWithBaseClasses>
      <baseClassMappings>
         <baseClassMapping>
            <contractPackageRegex>.*intoxication.*</contractPackageRegex>
            <baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
         </baseClassMapping>
      </baseClassMappings>
      <contractsDirectory>${project.build.directory}/contracts</contractsDirectory>
   </configuration>
</plugin>
```

For Gradle Project译：Gradle项目

- Add a custom configuration for the common-repo dependency:

```
ext {
    conractsGroupId = "com.example"
    contractsArtifactId = "common-repo"
    contractsVersion = "1.2.3"
}

configurations {
    contracts {
        transitive = false
    }
}
```

- Add the common-repo dependency to your classpath:

```
dependencies {
    contracts "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"
    testCompile "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"
}
```

- Download the dependency to an appropriate folder:

```
task getContracts(type: Copy) {
    from configurations.contracts
    into new File(project.buildDir, "downloadedContracts")
}
```

- Unzip JAR:

```
task unzipContracts(type: Copy) {
    def zipFile = new File(project.buildDir, "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")
    def outputDir = file("${buildDir}/unpackedContracts")

    from zipTree(zipFile)
    into outputDir
}
```

- Cleanup unused contracts:

```
task deleteUnwantedContracts(type: Delete) {
    delete fileTree(dir: "${buildDir}/unpackedContracts",
        include: "**/*",
        excludes: [
            "**/${project.name}/**"",
            "**/${first-topic}/**",
            "**/${second-topic}/**"])
}
```

- Create task dependencies:

```
unzipContracts.dependsOn("getContracts")
deleteUnwantedContracts.dependsOn("unzipContracts")
build.dependsOn("deleteUnwantedContracts")
```

- Configure plugin by specifying the directory containing contracts using `contractsDslDir` property

```
contracts {
    contractsDslDir = new File("${buildDir}/unpackedContracts")
}
```

86.6 Do I need a Binary Storage? Can't I use Git? 译：86.6我需要二进制存储吗？ 我能否使用Git？

在多语世界中，有些语言不使用像Artifactory或Nexus这样的二进制存储。 从Spring Cloud Contract 2.0.0开始，我们提供了一些机制来将合约和存根存储在SCM存储库中。 目前唯一支持的SCM是Git。

存储库将需要以下设置（您可以结帐 here ）：

```
.
└── META-INF
    └── com.example
        └── beer-api-producer-git
            └── 0.0.1-SNAPSHOT
                ├── contracts
                │   └── beer-api-consumer
                │       ├── messaging
                │       │   ├── shouldSendAcceptedVerification.groovy
                │       │   └── shouldSendRejectedVerification.groovy
                │       └── rest
                │           ├── shouldGrantABeerIfOldEnough.groovy
                │           └── shouldRejectABeerIfTooYoung.groovy
                └── mappings
                    └── beer-api-consumer
                        └── rest
                            ├── shouldGrantABeerIfOldEnough.json
                            └── shouldRejectABeerIfTooYoung.json
```

在 `META-INF` 文件夹下：

- we group applications via `groupId` (e.g. `com.example`)
- then each application is represented via the `artifactId` (e.g. `beer-api-producer-git`)
- next, the version of the application. The version is mandatory! (e.g. `0.0.1-SNAPSHOT`)
- 最后，有两个文件夹：
  - `contracts` - the good practice is to store the contracts required by each consumer in the folder with the consumer name (e.g. `beer-api-consumer`). That way you can use the `stubs-per-consumer` feature. Further directory structure is arbitrary.
  - `mappings` - in this folder the Maven / Gradle Spring Cloud Contract plugins will push the stub server mappings. On the consumer side, Stub Runner will scan this folder to start stub servers with stub definitions. The folder structure will be a copy of the one created in the `contracts` subfolder.

86.6.1 Protocol convention 译：86.6.1议定书约定

为了控制合同来源的类型和位置（无论是二进制存储还是SCM存储），您可以在存储库的URL中使用该协议。 Spring Cloud Contract迭代注册的协议解析器并尝试获取合约（通过插件）或存根（通过Stub Runner）。

对于SCM功能，目前我们支持Git存储库。 要使用它，在需要放置存储库URL的属性中，只需在连接URL前加上`git://` 。 在这里你可以找到几个例子：

```
git://file:///foo/bar
git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git
git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git
```

86.6.2 Producer译：86.6.2生产者

对于生产者来说，要使用SCM方法，我们可以重用我们用于外部合同的相同机制。 我们路由Spring Cloud Contract以通过包含`git://`协议的URL来使用SCM实现。

Important

您必须在Maven中手动添加`pushStubsToScm`目标，或者在Gradle中执行（绑定） `pushStubsToScm`任务。 我们不会将存根推`origin`您的git存储库的`origin`开箱即用。

Maven的。

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <!-- Base class mappings etc. -->

        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

        <!-- We reuse the contract dependency section to set up the path
        to the folder that contains the contract definitions. In our case the
        path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>${project.artifactId}</artifactId>
            <version>${project.version}</version>
        </contractDependency>

        <!-- The contracts mode can't be classpath -->
        <contractsMode>REMOTE</contractsMode>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <!-- By default we will not push the stubs back to SCM,
                you have to explicitly add it as a goal -->
                <goal>pushStubsToScm</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

摇篮。

```
contracts {
 // We want to pick contracts from a Git repository
 contractDependency {
  stringNotation = "${project.group}:${project.name}:${project.version}"
 }
 /*
 We reuse the contract dependency section to set up the path
 to the folder that contains the contract definitions. In our case the
 path will be /groupId/artifactId/version/contracts
  */
 contractRepository {
  repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git"
 }
 // The mode can't be classpath
 contractsMode = "REMOTE"
 // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is executed
*/
publish.dependsOn("publishStubsToScm")
```

有了这样的设置：

- Git project will be cloned to a temporary directory
- The SCM stub downloader will go to `META-INF/groupId/artifactId/version/contracts` folder to find contracts. E.g. for `com.example:foo:1.0.0` the path would be `META-INF/com.example/foo/1.0.0/contracts`
- Tests will be generated from the contracts
- Stubs will be created from the contracts
- Once the tests pass, the stubs will be committed in the cloned repository
- Finally, a push will be done to that repo's `origin`

86.6.3 Consumer译：86.6.3消费者

在消费者方面，从`@AutoConfigureStubRunner`注释，JUnit规则或属性传递`repositoryRoot`参数时，它足以传递SCM存储库的URL（以协议为前缀）。 例如

```
@AutoConfigureStubRunner(
    stubsMode="REMOTE",
    repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git",
    ids="com.example:bookstore:0.0.1.RELEASE"
)
```

有了这样的设置：

- Git project will be cloned to a temporary directory
- The SCM stub downloader will go to `META-INF/groupId/artifactId/version/` folder to find stub definitions and contracts. E.g. for `com.example:foo:1.0.0` the path would be `META-INF/com.example/foo/1.0.0/`
- Stub servers will be started and fed with mappings
- Messaging definitions will be read and used in the messaging tests

86.7 Can I use the Pact Broker? 译：86.7我可以使用Pact Broker吗？

使用Pact时，可以使用Pact Broker来存储和共享Pact定义。 从Spring Cloud Contract 2.0.0开始，可以从Pact Broker获取Pact文件以生成测试和存根。

作为先决条件，Pact Converter和Pact Stub Downloader是必需的。 您必须通过`spring-cloud-contract-pact`依赖项添加它。 您可以在Section 93.1.1, “Pact Converter”部分阅读更多内容。

Important

契约遵循消费者合同公约。 这意味着消费者首先创建Pact定义，然后与Producer共享文件。 这些期望是从消费者的代码中产生的，如果期望没有得到满足，可以打破生产者。

86.7.1 Pact Consumer 译：86.7.1协议消费者

消费者使用Pact框架生成Pact文件。 Pact文件被发送到Pact Broker。 这种设置的一个例子可以找到here 。

86.7.2 Producer 译：86.7.2生产者

对于生产者来说，要使用Pact Broker中的Pact文件，我们可以重复使用我们用于外部合同的相同机制。 我们路由Spring Cloud Contract以通过包含`pact://`协议的URL使用Pact实现。 它足以将URL传递给Pact Broker。 这种设置的一个例子可以找到here 。

Maven的。

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <!-- Base class mappings etc. -->

        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>pact://http://localhost:8085</contractsRepositoryUrl>

        <!-- We reuse the contract dependency section to set up the path
        to the folder that contains the contract definitions. In our case the
        path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>${project.artifactId}</artifactId>
            <!-- When + is passed, a latest tag will be applied when fetching pacts -->
            <version>+</version>
        </contractDependency>

        <!-- The contracts mode can't be classpath -->
        <contractsMode>REMOTE</contractsMode>
    </configuration>
    <!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-pact</artifactId>
            <version>${spring-cloud-contract.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

摇篮。

```
buildscript {
 repositories {
  //...
 }

 dependencies {
  // ...
  // Don't forget to add spring-cloud-contract-pact to the classpath!
  classpath "org.springframework.cloud:spring-cloud-contract-pact:${contractVersion}"
 }
}

contracts {
 // When + is passed, a latest tag will be applied when fetching pacts
 contractDependency {
  stringNotation = "${project.group}:${project.name}:+"
 }
 contractRepository {
  repositoryUrl = "pact://http://localhost:8085"
 }
 // The mode can't be classpath
 contractsMode = "REMOTE"
 // Base class mappings etc.
}
```

有了这样的设置：

- Pact files will be downloaded from the Pact Broker

- Spring Cloud Contract will convert the Pact files into tests and stubs
- The JAR with the stubs gets automatically created as usual

86.7.3 Pact Consumer (Producer Contract approach)译: 86.7.3协定消费者（生产者合同法）

在你不想做消费者合同方法（每个消费者定义期望值）但你更愿意做生产者合同（生产者提供合同并发布存根）的情况下，它已经足够了与Stub Runner选项一起使用Spring Cloud Contract。 这种设置的一个例子可以找到here 。

首先，请记住将Stub Runner和Spring Cloud Contract Pact模块添加为测试依赖项。

Maven的。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-contract-pact</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

摇篮。

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

dependencies {
    //...
    testCompile("org.springframework.cloud:spring-cloud-starter-contract-stub-runner")
    // Don't forget to add spring-cloud-contract-pact to the classpath!
    testCompile("org.springframework.cloud:spring-cloud-contract-pact")
}
```

接下来，只需将Pact Broker的URL传递至`repositoryRoot` ，前缀为`pact://`协议。 例如`pact://http://localhost:8085`

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.REMOTE, ids = "com.example:beer-api-producer-pact", repositoryRoot = "pact://http
public class BeerControllerTest {
    //Inject the port of the running stub
    @StubRunnerPort("beer-api-producer-pact") int producerPort;
    //...
}
```

有了这样的设置：

- Pact files will be downloaded from the Pact Broker
- Spring Cloud Contract will convert the Pact files into stub definitions
- The stub servers will be started and fed with stubs

有关Pact支持的更多信息，请参阅 Section 93.7, “Using the Pact Stub Downloader”部分。

86.8 How can I debug the request/response being sent by the generated tests client?译: 86.8如何调试生成的测试客户端发送的请求/响应？

生成的测试全部归结为以某种形式或方式依赖于Apache HttpClient的RestAssured 。 HttpClient有一个名为wire logging的工具， 它将整个请求和响应记录到HttpClient。 Spring Boot有一个日志记录common application property用于做这种事情，只需将它添加到您的应用程序属性

```
logging.level.org.apache.http.wire=DEBUG
```

86.8.1 How can I debug the mapping/request/response being sent by WireMock?译: 86.8.1如何调试由WireMock发送的映射/请求/响应？

从版本`1.2.0`开始，我们打开WireMock日志记录到信息，并且WireMock通知器变为冗长。 现在您将完全知道WireMock服务器收到了什么请求，以及选择了哪个匹配响应定义。

要关闭此功能，只需将WireMock日志记录到 `ERROR`

```
logging.level.com.github.tomakehurst.wiremock=ERROR
```

86.8.2 How can I see what got registered in the HTTP server stub?译: 86.8.2如何查看HTTP服务器存根中注册的内容？

您可以使用`@AutoConfigureStubRunner`或`StubRunnerRule`上的`mappingsOutputFolder`属性来转储每个工件标识的所有映射。 此外，连接了给定存根服务器的端口。

86.8.3 Can I reference text from file? 译：86.8.3我可以从文件中引用文本吗？

是! 在版本1.2.0中，我们增加了这种可能性。 它足以在DSL中调用`file(… )`方法，并提供相对于契约位置的路径。 如果您使用YAML只需使用`bodyFromFile`属性。

87. Spring Cloud Contract Verifier Setup 译：87. Spring Cloud Contract Verifier设置

您可以通过以下方式设置Spring Cloud Contract Verifier：

- As a Gradle project
- As a Maven project
- As a Docker project

87.1 Gradle Project 译：87.1 Gradle项目

要了解如何为Spring Cloud Contract Verifier设置Gradle项目，请阅读以下部分：

87.1.1 Prerequisites 译：87.1.1先决条件

为了在WireMock中使用Spring Cloud Contract Verifier，您可以使用Gradle或Maven插件。

如果您想在项目中使用Spock，则必须单独添加`spock-core`和`spock-spring`模块。 检查Spock docs for more information

87.1.2 Add Gradle Plugin with Dependencies 译：87.1.2添加具有依赖关系的Gradle插件

要添加一个具有依赖关系的Gradle插件，请使用与此类似的代码：

```
buildscript {
 repositories {
  mavenCentral()
 }
 dependencies {
     classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot_version}"
  classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier_version}"
 }
}

apply plugin: 'groovy'
apply plugin: 'spring-cloud-contract'

dependencyManagement {
 imports {
  mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifier_version}"
 }
}

dependencies {
 testCompile 'org.codehaus.groovy:groovy-all:2.4.6'
 // example with adding Spock core and Spock Spring
 testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'
 testCompile 'org.spockframework:spock-spring:1.0-groovy-2.4'
 testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```

87.1.3 Gradle and Rest Assured 2.0 译：87.1.3 Gradle和Rest保证2.0

默认情况下，Rest Assured 3.x将添加到类路径中。 但是，要使用Rest Assured 2.x，您可以将其添加到插件类路径中，如下所示：

```
buildscript {
 repositories {
  mavenCentral()
 }
 dependencies {
     classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot_version}"
  classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier_version}"
  classpath "com.jayway.restassured:rest-assured:2.5.0"
  classpath "com.jayway.restassured:spring-mock-mvc:2.5.0"
 }
}

depenendencies {
    // all dependencies
    // you can exclude rest-assured from spring-cloud-contract-verifier
    testCompile "com.jayway.restassured:rest-assured:2.5.0"
    testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"
}
```

这样，插件会自动看到RestAssured 2.x存在于类路径中，并相应地修改了导入。

87.1.4 Snapshot Versions for Gradle译：87.1.4 Gradle的快照版本

将其他快照存储库添加到您的build.gradle中以使用快照版本，这些快照版本在每次成功构建后自动上传，如下所示：

```
buildscript {
 repositories {
  mavenCentral()
  mavenLocal()
  maven { url "http://repo.spring.io/snapshot" }
  maven { url "http://repo.spring.io/milestone" }
  maven { url "http://repo.spring.io/release" }
 }
}
```

87.1.5 Add stubs译：87.1.5添加存根

默认情况下，Spring Cloud Contract Verifier正在 `src/test/resources/contracts` 目录中查找存根。

包含存根定义的目录被视为一个类名，并且每个存根定义被视为单个测试。 Spring Cloud Contract Verifier假定它至少包含一个要用作测试类名称的目录级别。 如果存在多个嵌套目录级别，则除最后一个以外的所有目录均用作包名称。 例如，具有以下结构：

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract Verifier使用两种方法创建一个名为 `defaultBasePackage.MyService` 的测试类：

- `shouldCreateUser()`
- `shouldReturnUser()`

87.1.6 Run the Plugin译：87.1.6运行插件

该插件注册自己以在 `check` 任务之前被调用。 如果你希望它成为你的构建过程的一部分，你就不需要再做任何事情。 如果您只想生成测试，请调用 `generateContractTests` 任务。

87.1.7 Default Setup译：87.1.7默认设置

默认的Gradle Plugin安装程序会创建以下的构建Gradle部分（伪代码）：

```
contracts {
    targetFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestSourcesDir = project.file("${project.buildDir}/generated-test-sources/contracts")
    contractsDslDir = "${project.rootDir}/src/test/resources/contracts"
    basePackageForTests = 'org.springframework.cloud.verifier.tests'
    stubsOutputDir = project.file("${project.buildDir}/stubs")

    // the following properties are used when you want to provide where the JAR with contract lays
    contractDependency {
        stringNotation = ''
    }
    contractsPath = ''
    contractsWorkOffline = false
    contractRepository {
        cacheDownloadedContracts(true)
    }
}

tasks.create(type: Jar, name: 'verifierStubsJar', dependsOn: 'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

project.artifacts {
    archives task
}

tasks.create(type: Copy, name: 'copyContracts') {
    from contracts.contractsDslDir
    into contracts.stubsOutputDir
}

verifierStubsJar.dependsOn 'copyContracts'

publishing {
    publications {
        stubs(MavenPublication) {
            artifactId project.name
            artifact verifierStubsJar
        }
    }
}
```

87.1.8 Configure Plugin译：87.1.8配置插件

要更改默认配置，请将`contracts`片段添加到您的Gradle配置中，如下所示：

```
contracts {
 testMode = 'MockMvc'
 baseClassForTests = 'org.mycompany.tests'
 generatedTestSourcesDir = project.file('src/generatedContract')
}
```

87.1.9 Configuration Options译：87.1.9配置选项

- **testMode**: Defines the mode for acceptance tests. By default, the mode is MockMvc, which is based on Spring's MockMvc. It can also be changed to **JaxRsClient** or to **Explicit** for real HTTP calls.
- **imports**: Creates an array with imports that should be included in generated tests (for example ['org.myorg.Matchers']). By default, it creates an empty array.
- **staticImports**: Creates an array with static imports that should be included in generated tests(for example ['org.myorg.Matchers.*']). By default, it creates an empty array.
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from `baseClassForTests's package and from `packageWithBaseClasses`. If neither of these values are set, then the value is set to `org.springframework.cloud.contract.verifier.tests`.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is `spock.lang.Specification`.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over **baseClassForTests**.
- **baseClassMappings**: Explicitly maps a contract package to a FQN of a base class. This setting takes precedence over **packageWithBaseClasses** and **baseClassForTests**.
- **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
- **ignoredFiles**: Uses an `Antmatcher` to allow defining stub files for which processing should be skipped. By default, it is an empty array.
- **contractsDslDir**: Specifies the directory containing contracts written using the GroovyDSL. By default, its value is `$rootDir/src/test/resources/contracts`.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default its value is `$buildDir/generated-test-sources/contractVerifier`.
- **stubsOutputDir**: Specifies the directory where the generated WireMock stubs from the Groovy DSL should be placed.
- **targetFramework**: Specifies the target test framework to be used. Currently, Spock and JUnit are supported with JUnit being the default framework.
- **contractsProperties**: a map containing properties to be passed to Spring Cloud Contract components. Those properties might be used by e.g. inbuilt or custom Stub Downloaders.

当您要指定包含合同的JAR的位置时，将使用以下属性：* **contractDependency**：指定提供`groupid:artifactid:version:classifier`坐标的依赖`groupid:artifactid:version:classifier`。您可以使用`contractDependency`闭包进行设置。* **contractsPath**：指定jar的路径。如果下载合同依赖关系，则路径默认为`groupid/artifactid`，其中`groupid`以斜杠分隔。否则，它会扫描提供的目录下的合同。* **contractsMode**：指定下载合同的模式（JAR是否可以离线或远程使用）* **contractsSnapshotCheckSkip**：如果设置为`true`则不会断言下载的存根/合同JAR是从远程位置还是从本地位置下载（仅限于适用于Maven回购协议，不适用于Git或Pact）。* **deleteStubsAfterTest**：如果设置为`false`则不会从临时目录中删除任何已下载的合约

87.1.10 Single Base Class for All Tests译：87.1.10所有测试的单个基类

在默认MockMvc中使用Spring Cloud Contract Verifier时，您需要为所有生成的验收测试创建一个基本规范。在这个类中，你需要指向一个端点，该端点应该被验证。

```
abstract class BaseMockMvcSpec extends Specification {

 def setup() {
  RestAssuredMockMvc.standaloneSetup(new PairIdController())
 }

 void isProperCorrelationId(Integer correlationId) {
  assert correlationId == 123456
 }

 void isEmpty(String value) {
  assert value == null
 }

}
```

如果您使用`Explicit`模式，则可以使用基类来初始化整个测试的应用程序，就像您在常规集成测试中看到的那样。 如果您使用`JAXRSCLIENT`模式，则此基类还应包含`protected WebTarget webTarget`字段。 目前，测试JAX-RS API的唯一选择是启动一个Web服务器。

87.1.11 Different Base Classes for Contracts 译：87.1.11合同的不同基类

如果您的基类在合约之间有所不同，那么您可以通过自动生成的测试告诉Spring Cloud合约插件应该扩展哪个类。 你有两个选择：

- Follow a convention by providing the `packageWithBaseClasses`
- Provide explicit mapping via `baseClassMappings`

按照惯例

约定是这样的，如果你有一个合同（例如） `src/test/resources/contract/foo/bar/baz/`并将`packageWithBaseClasses`属性的值设置为`com.example.base`，那么Spring Cloud Contract Verifier假定`com.example.base`包下有一个`BarBazBase`类。 换句话说，系统将包裹的最后两部分（如果它们存在），并形成带有`Base`后缀的类别。 此规则优先于**baseClassForTests** 。 以下是关于`contracts`关闭如何工作的`contracts`：

```
packageWithBaseClasses = 'com.example.base'
```

通过映射

您可以手动将合同包的正则表达式映射到匹配合同的基类的标准名称。 您必须提供名为`baseClassMappings`的列表，该列表由`baseClassMapping`对象组成，其中包含`contractPackageRegex`到`baseClassFQN`映射。 考虑下面的例子：

```
baseClassForTests = "com.example.FooBase"
baseClassMappings {
 baseClassMapping('.*/com/.*', 'com.example.ComBase')
 baseClassMapping('.*/bar/.*':'com.example.BarBase')
}
```

让我们假设你有合同 - `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

通过提供`baseClassForTests`，我们在案例映射没有成功的情况下有一个回退。（您也可以提供`packageWithBaseClasses`作为后备）。通过这种方式，`src/test/resources/contract/com/`合同生成的测试扩展了`com.example.ComBase`，而其余测试扩展了`com.example.FooBase`。

87.1.12 Invoking Generated Tests 译：87.1.12调用生成的测试

为了确保提供者方符合定义的合约，您需要调用：

```
./gradlew generateContractTests test
```

87.1.13 Pushing stubs to SCM 译：87.1.13推送存根到SCM

如果您正在使用SCM存储库来保留合同和存根，那么您可能需要自动执行将存根推送到存储库的步骤。 为此，调用`pushStubsToScm`任务就足够了。 例：

```
$ ./gradlew pushStubsToScm
```

在 Section 93.6, “Using the SCM Stub Downloader”下，您可以找到所有可能的配置选项，您可以通过`contractsProperties`字段（例如`contracts { contractsProperties = [foo:"bar"] }`，通过`contractsProperties`方法（例如`contracts { contractsProperties([foo:"bar"]) }`系统属性或环境变量。

87.1.14 Spring Cloud Contract Verifier on the Consumer Side 译：87.1.14消费者端的Spring Cloud合约验证程序

在使用服务中，您需要像提供程序一样配置Spring Cloud Contract Verifier插件。 如果您不想使用Stub Runner，那么您需要复制存储在`src/test/resources/contracts`合同并使用以下`src/test/resources/contracts`生成WireMock JSON存根：

```
./gradlew generateClientStubs
```

必须设置`stubsOutputDir`选项以使存根生成才能工作。

如果存在，JSON存根可用于使用服务的自动测试。

```
@ContextConfiguration(loader == SpringApplicationContextLoader, classes == Application)
class LoanApplicationServiceSpec extends Specification {

 @ClassRule
 @Shared
 WireMockClassRule wireMockRule == new WireMockClassRule()

 @Autowired
 LoanApplicationService sut

 def 'should successfully apply for loan'() {
   given:
  LoanApplication application =
   new LoanApplication(client: new Client(clientPesel: '12345678901'), amount: 123.123)
   when:
 LoanApplicationResult loanApplication == sut.loanApplication(application)
   then:
 loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN_APPLIED
 loanApplication.rejectionReason == null
 }
}
```

`LoanApplication`拨打`FraudDetection`服务。 此请求由配置有Spring Cloud Contract Verifier生成的存根的WireMock服务器处理。

87.2 Maven Project译：87.2 Maven项目

要了解如何为Spring Cloud Contract Verifier设置Maven项目，请阅读以下部分：

87.2.1 Add maven plugin译：87.2.1添加maven插件

以类似于以下的方式添加Spring Cloud合同BOM：

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>${spring-cloud-dependencies.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

接下来，添加`Spring Cloud Contract Verifier`Maven插件：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
 <configuration>
  <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
 </configuration>
</plugin>
```

您可以在 Spring Cloud Contract Maven Plugin Documentation中阅读更多 内容 。

87.2.2 Maven and Rest Assured 2.0译：87.2.2 Maven和其他保证2.0

默认情况下，Rest Assured 3.x将添加到类路径中。 但是，您可以将Rest Assured 2.x添加到插件类路径中，如下所示：

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example</packageWithBaseClasses>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-verifier</artifactId>
            <version>${spring-cloud-contract.version}</version>
        </dependency>
        <dependency>
           <groupId>com.jayway.restassured</groupId>
           <artifactId>rest-assured</artifactId>
           <version>2.5.0</version>
           <scope>compile</scope>
        </dependency>
        <dependency>
           <groupId>com.jayway.restassured</groupId>
           <artifactId>spring-mock-mvc</artifactId>
           <version>2.5.0</version>
           <scope>compile</scope>
        </dependency>
    </dependencies>
</plugin>

<dependencies>
    <!-- all dependencies -->
    <!-- you can exclude rest-assured from spring-cloud-contract-verifier -->
    <dependency>
       <groupId>com.jayway.restassured</groupId>
       <artifactId>rest-assured</artifactId>
       <version>2.5.0</version>
       <scope>test</scope>
    </dependency>
    <dependency>
       <groupId>com.jayway.restassured</groupId>
       <artifactId>spring-mock-mvc</artifactId>
       <version>2.5.0</version>
       <scope>test</scope>
    </dependency>
</dependencies>
```

这样，插件会自动看到类别路径中存在Rest Assured 3.x，并相应地修改导入。

87.2.3 Snapshot versions for Maven译：87.2.3 Maven的快照版本

对于快照和Milestone版本，您必须将以下部分添加到 `pom.xml` ，如下所示：

```
<repositories>
 <repository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
</pluginRepositories>
```

87.2.4 Add stubs译：87.2.4添加存根

默认情况下，Spring Cloud Contract Verifier正在查找`src/test/resources/contracts`目录中的存根。 包含存根定义的目录被视为一个类名，并且每个存根定义被视为单个测试。 我们假设它至少包含一个用作测试类名称的目录。 如果嵌套目录的级别不止一个，则除最后一个以外的所有嵌套目录均用作包名称。 例如，具有以下结构：

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract Verifier使用两种方法创建名为`defaultBasePackage.MyService`的测试类

- `shouldCreateUser()`
- `shouldReturnUser()`

87.2.5 Run plugin译：87.2.5运行插件

插件目标`generateTests`被分配为在名为`generate-test-sources`的阶段被调用。 如果你希望它成为你的构建过程的一部分，你不需要做任何事情。 如果您只想生成测试，请调用`generateTests`目标。

87.2.6 Configure plugin译：87.2.6配置插件

要更改默认配置，只需将`configuration`部分添加到插件定义或`execution`定义中，如下所示：

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>convert</goal>
                <goal>generateStubs</goal>
                <goal>generateTests</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackageForTests>
        <baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpec</baseClassForTests>
    </configuration>
</plugin>
```

87.2.7 Configuration Options译：87.2.7配置选项

- **testMode**: Defines the mode for acceptance tests. By default, the mode is MockMvc, which is based on Spring's MockMvc. It can also be changed to **JaxRsClient** or to **Explicit** for real HTTP calls.
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from `baseClassForTests's package and from `packageWithBaseClasses`. If neither of these values are set, then the value is set to `org.springframework.cloud.contract.verifier.tests`.
- **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is `spock.lang.Specification`.
- **contractsDirectory**: Specifies a directory containing contracts written with the GroovyDSL. The default directory is `/src/test/resources/contracts`.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock and JUnit are supported with JUnit being the default framework
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over **baseClassForTests**. The convention is such that, if you have a contract under (for example) `src/test/resources/contract/foo/bar/baz/` and set the value of the `packageWithBaseClasses` property to `com.example.base`, then Spring Cloud Contract Verifier assumes that there is a `BarBazBase` class under the `com.example.base` package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a `Base` suffix.
- **baseClassMappings**: Specifies a list of base class mappings that provide `contractPackageRegex`, which is checked against the package where the contract is located, and `baseClassFQN`, which maps to the fully qualified name of the base class for the matched contract. For example, if you have a contract under `src/test/resources/contract/foo/bar/baz/` and map the property `.* → com.example.base.BaseClass`, then the test class generated from these contracts extends `com.example.base.BaseClass`. This setting takes precedence over **packageWithBaseClasses** and **baseClassForTests**.
- **contractsProperties**: a map containing properties to be passed to Spring Cloud Contract components. Those properties might be used by e.g. inbuilt or custom Stub Downloaders.

如果你想从Maven仓库下载合约定义，你可以使用下列选项：

- **contractDependency**: The contract dependency that contains all the packaged contracts.
- **contractsPath**: The path to the concrete contracts in the JAR with packaged contracts. Defaults to `groupid/artifactid` where `gropuid` is slash separated.
- **contractsMode**: Picks the mode in which stubs will be found and registered
- **contractsSnapshotCheckSkip**: If `true` then will not assert whether a stub / contract JAR was downloaded from local or remote location
- **deleteStubsAfterTest**: If set to `false` will not remove any downloaded contracts from temporary directories
- **contractsRepositoryUrl**: URL to a repo with the artifacts that have contracts. If it is not provided, use the current Maven ones.
- **contractsRepositoryUsername**: The user name to be used to connect to the repo with contracts.
- **contractsRepositoryPassword**: The password to be used to connect to the repo with contracts.
- **contractsRepositoryProxyHost**: The proxy host to be used to connect to the repo with contracts.
- **contractsRepositoryProxyPort**: The proxy port to be used to connect to the repo with contracts.

我们只缓存非快照，明确提供的版本（例如， `+`或`1.0.0.BUILD-SNAPSHOT`不会被缓存）。 默认情况下，此功能处于打开状态。

87.2.8 Single Base Class for All Tests译：87.2.8所有测试的单个基类

在默认MockMvc中使用Spring Cloud Contract Verifier时，您需要为所有生成的验收测试创建一个基本规范。 在这个类中，你需要指向一个端点，该端点应该被验证。

```
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class MvcSpec extends Specification {
  def setup() {
   RestAssuredMockMvc.standaloneSetup(new ExampleSpringController())
  }
}
```

如有必要，您还可以设置整个上下文。

```
import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes = SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

 @Autowired
 WebApplicationContext context;

 @Before
 public void setup() {
  RestAssuredMockMvc.webAppContextSetup(this.context);
 }
}
```

如果您使用 `EXPLICIT`模式，则可以使用基类以类似方式初始化整个测试应用程序，就像您在常规集成测试中发现的那样。

```
import io.restassured.RestAssured;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes = SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

 @LocalServerPort
 int port;

 @Before
 public void setup() {
  RestAssured.baseURI = "http://localhost:" + this.port;
 }
}
```

如果使用`JAXRSCLIENT`模式，则此基类还应包含`protected WebTarget webTarget`字段。 目前，测试JAX-RS API的唯一选择是启动一个Web服务器。

87.2.9 Different base classes for contracts译：87.2.9给同的不同基类

如果您的基类在合约之间有所不同，那么您可以通过自动生成的测试告诉Spring Cloud合约插件应该扩展哪个类。 你有两个选择：

- Follow a convention by providing the `packageWithBaseClasses`
- provide explicit mapping via `baseClassMappings`

按照惯例

约定是这样的，如果你有下（例如）合同`src/test/resources/contract/foo/bar/baz/`和的值设置`packageWithBaseClasses`属性为`com.example.base`，然后春云合同验证假设有一个`BarBazBase`个类`com.example.base`包。 换句话说，系统将包的最后两部分（如果存在的话）包含`Base`，并且形成带有`Base`后缀的类。 此规则优先于**baseClassForTests**。 以下是关于如何在`contracts`关闭中工作的`contracts`：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <configuration>
  <packageWithBaseClasses>hello</packageWithBaseClasses>
 </configuration>
</plugin>
```

通过映射

您可以手动将合同包的正则表达式映射到匹配合同的基类的标准名称。 您必须提供一个名为`baseClassMappings`的列表，其中包含`baseClassMapping`对象，它们需要`contractPackageRegex`到`baseClassFQN`映射。 考虑下面的例子：

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <configuration>
  <baseClassForTests>com.example.FooBase</baseClassForTests>
  <baseClassMappings>
   <baseClassMapping>
    <contractPackageRegex>.*com.*</contractPackageRegex>
    <baseClassFQN>com.example.TestBase</baseClassFQN>
   </baseClassMapping>
  </baseClassMappings>
 </configuration>
</plugin>
```

假设你有这两个地点的合同：* `src/test/resources/contract/com/` * `src/test/resources/contract/foo/`

通过提供`baseClassForTests`，我们在案例映射没有成功的情况下有一个回退。（您也可以提供`packageWithBaseClasses`作为后备。）这样，从`src/test/resources/contract/com/`合同生成的测试扩展了`com.example.ComBase`，而其余测试扩展了`com.example.FooBase`。

87.2.10 Invoking generated tests译：87.2.10调用生成的测试

Spring Cloud Contract Maven插件在一个名为`/generated-test-sources/contractVerifier`的目录中生成验证码，并将此目录附加到`testCompile`目标。

对于Groovy Spock代码，请使用以下代码：

```
<plugin>
 <groupId>org.codehaus.gmavenplus</groupId>
 <artifactId>gmavenplus-plugin</artifactId>
 <version>1.5</version>
 <executions>
  <execution>
   <goals>
    <goal>testCompile</goal>
   </goals>
  </execution>
 </executions>
 <configuration>
  <testSources>
   <testSource>
    <directory>${project.basedir}/src/test/groovy</directory>
    <includes>
     <include>**/*.groovy</include>
    </includes>
   </testSource>
   <testSource>
    <directory>${project.build.directory}/generated-test-sources/contractVerifier</directory>
    <includes>
     <include>**/*.groovy</include>
    </includes>
   </testSource>
  </testSources>
 </configuration>
</plugin>
```

为确保提供者方符合定义的合约，您需要调用 `mvn generateTest test` 。

87.2.11 Pushing stubs to SCM 译：87.2.11将存根推送到SCM

如果您正在使用SCM存储库来保留合同和存根，那么您可能需要自动执行将存根推送到存储库的步骤。 要做到这一点，就足以添加`pushStubsToScm`目标。 例：

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <!-- Base class mappings etc. -->

        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

        <!-- We reuse the contract dependency section to set up the path
        to the folder that contains the contract definitions. In our case the
        path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>${project.artifactId}</artifactId>
            <version>${project.version}</version>
        </contractDependency>

        <!-- The contracts mode can't be classpath -->
        <contractsMode>REMOTE</contractsMode>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <!-- By default we will not push the stubs back to SCM,
                you have to explicitly add it as a goal -->
                <goal>pushStubsToScm</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

在 Section 93.6, “Using the SCM Stub Downloader”下，您可以找到所有可能的配置选项，您可以通过 `<configuration><contractProperties>` 映射，系统属性或环境变量传递。

87.2.12 Maven Plugin and STS 译：87.2.12 Maven插件和STS

如果在使用STS时看到以下异常：

当你点击错误标记时，你会看到如下所示的内容：

```
 plugin:1.1.0.M1:convert:default-convert:process-test-resources) org.apache.maven.plugin.PluginExecutionException: Execution default-convert of goal
 cloud-contract-maven-plugin:1.1.0.M1:convert failed. at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginManager.java
 org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.j
...
 org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by: java.lang.NullPointerException at
 org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildContext.hasDelta(EclipseIncrementalBuildContext.java:53) at
 org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildContext.java:59) at
```

为了解决此问题，请在您的 `pom.xml` 提供以下部分：

```
<build>
    <pluginManagement>
        <plugins>
            <!--This plugin's configuration is used to store Eclipse m2e settings
                only. It has no influence on the Maven build itself. -->
            <plugin>
                <groupId>org.eclipse.m2e</groupId>
                <artifactId>lifecycle-mapping</artifactId>
                <version>1.0.0</version>
                <configuration>
                    <lifecycleMappingMetadata>
                        <pluginExecutions>
                             <pluginExecution>
                                <pluginExecutionFilter>
                                    <groupId>org.springframework.cloud</groupId>
                                    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
                                    <versionRange>[1.0,)</versionRange>
                                    <goals>
                                        <goal>convert</goal>
                                    </goals>
                                </pluginExecutionFilter>
                                <action>
                                    <execute />
                                </action>
                             </pluginExecution>
                        </pluginExecutions>
                    </lifecycleMappingMetadata>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

87.3 Stubs and Transitive Dependencies译：87.3存根和传递依赖

Maven和Gradle插件添加了为您创建存根jar的任务。 出现的一个问题是，在重新使用存根时，可能会错误地导入所有存根的依赖关系。 当构建一个Maven神器时，即使你有几个不同的罐子，它们都共享一个罐子：

```
├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-SNAPSHOT.jar
├── github-webhook-0.0.1.BUILD-SNAPSHOT.pom
├── github-webhook-0.0.1.BUILD-SNAPSHOT-stubs.jar
├── ...
└── ...
```

有三种可能性来处理这些依赖关系，以免在传递依赖关系中存在任何问题：

- Mark all application dependencies as optional
- Create a separate artifactid for the stubs
- Exclude dependencies on the consumer side

将所有应用程序依赖项标记为可选

如果在`github-webhook`应用程序`github-webhook`所有依赖项标记为可选，那么当您在另一个应用程序中包含`github-webhook`存根时（或者由Stub Runner下载该依赖关系时），因为所有依赖项都是可选的，所以它们不会获得下载。

为存根创建一个单独的`artifactid`

如果你创建一个单独的`artifactid`，那么你可以以任何你想要的方式进行设置。 例如，你可能决定根本没有依赖关系。

排除消费者方面的依赖性

作为使用者，如果将存根依赖关系添加到类路径中，则可以显式排除不需要的依赖关系。

87.4 CI Server setup译：87.4CI服务器设置

当在CI，共享环境中获取存根/契约时，可能发生的情况是生产者和消费者重用相同的本地Maven存储库。 由于这个原因，负责从远程位置下载存根JAR的框架无法决定应该选择哪个JAR，本地还是远程JAR。 这导致了`"The artifact was found in the local repository but you have explicitly stated that it should be downloaded from a remote one"`异常并且构建失败。

对于这种情况，我们将引入属性和插件设置机制：

- via `stubrunner.snapshot-check-skip` system property
- via `STUBRUNNER_SNAPSHOT_CHECK_SKIP` environment variable

如果其中任一值设置为`true`，则存根下载器将不会验证下载的JAR的来源。

对于插件，您需要将`contractsSnapshotSkipCheck`属性设置为`true`。

87.5 Scenarios译：87.5情景

您可以使用Spring Cloud Contract Verifier处理场景。 您只需在创建合同时遵守适当的命名惯例。 该公约要求包括一个后跟下划线的订单号。 这将用于您是否正在使用YAML或Groovy。 例：

```
my_contracts_dir\
  scenario1\
    1_login.groovy
    2_showCart.groovy
    3_logout.groovy
```

这样的树导致Spring Cloud Contract Verifier生成WireMock的场景，其名称为 `scenario1` 并执行以下三个步骤：

1. login marked as `Started` pointing to…
2. showCart marked as `Step1` pointing to…
3. logout marked as `Step2` which will close the scenario.

有关WireMock场景的更多详细信息，请参阅 http://wiremock.org/stateful-behaviour.html

Spring Cloud Contract Verifier还可以保证执行顺序生成测试。

87.6 Docker Project译: 87.6码头工程

我们将发布一个包含项目的 `springcloud/spring-cloud-contract` Docker镜像，该项目将生成测试并以 `EXPLICIT` 模式针对正在运行的应用程序执行测试。

`EXPLICIT` 模式意味着从合同生成的测试将发送真正的请求而不是嘲笑的测试。

87.6.1 Short intro to Maven, JARs and Binary storage译: 87.6.1简短介绍Maven，JAR和二进制存储

由于Docker镜像可以被非JVM项目使用，所以很好解释Spring Cloud合同包装默认的基本术语。

以下部分定义取自 Maven Glossary

- `Project`: Maven thinks in terms of projects. Everything that you will build are projects. Those projects follow a well defined “Project Object Model”. Projects can depend on other projects, in which case the latter are called “dependencies”. A project may consistent of several subprojects, however these subprojects are still treated equally as projects.
- `Artifact`: An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions. Each artifact is uniquely identified by a group id and an artifact ID which is unique within a group.
- `JAR`: JAR stands for Java ARchive. It’s a format based on the ZIP file format. Spring Cloud Contract packages the contracts and generated stubs in a JAR file.
- `GroupId`: A group ID is a universally unique identifier for a project. While this is often just the project name (eg. commons-collections), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. org.apache.maven). Typically, when published to the Artifact Manager, the `GroupId` will get slash separated and form part of the URL. E.g. for group id `com.example` and artifact id `application` would be `/com/example/application/`.
- `Classifier`: The Maven dependency notation looks as follows: `groupId:artifactId:version:classifier`. The classifier is additional suffix passed to the dependency. E.g. `stubs`, `sources`. The same dependency e.g. `com.example:application` can produce multiple artifacts that differ from each other with the classifier.
- `Artifact manager`: When you generate binaries / sources / packages, you would like them to be available for others to download / reference or reuse. In case of the JVM world those artifacts would be JARs, for Ruby these are gems and for Docker those would be Docker images. You can store those artifacts in a manager. Examples of such managers can be Artifactory or Nexus.

87.6.2 How it works译: 87.6.2它是如何工作的

该图像搜索 `/contracts` 文件夹下的合同。 运行测试的输出将在 `/spring-cloud-contract/build` 文件夹下 `/spring-cloud-contract/build` （这对于调试目的很有用）。

它足以让你安装你的合同，传递环境变量和图像：

- generate the contract tests
- execute the tests against the provided URL
- generate the WireMock stubs
- (optional - turned on by default) publish the stubs to a Artifact Manager

Environment Variables译:环境变量

Docker镜像需要一些环境变量指向正在运行的应用程序，以及Artifact管理器实例等。

- `PROJECT_GROUP` - your project’s group id. Defaults to `com.example`
- `PROJECT_VERSION` - your project’s version. Defaults to `0.0.1-SNAPSHOT`
- `PROJECT_NAME` - artifact id. Defaults to `example`
- `REPO_WITH_BINARIES_URL` - URL of your Artifact Manager. Defaults to `http://localhost:8081/artifactory/libs-release-local` which is the default URL of Artifactory running locally
- `REPO_WITH_BINARIES_USERNAME` - (optional) username when the Artifact Manager is secured
- `REPO_WITH_BINARIES_PASSWORD` - (optional) password when the Artifact Manager is secured
- `PUBLISH_ARTIFACTS` - if set to `true` then will publish artifact to binary storage. Defaults to `true`.

这些环境变量在合同放置在外部存储库中时使用。 要启用此功能，您必须设置 `EXTERNAL_CONTRACTS_ARTIFACT_ID` 环境变量。

- `EXTERNAL_CONTRACTS_GROUP_ID` - group id of the project with contracts. Defaults to `com.example`
- `EXTERNAL_CONTRACTS_ARTIFACT_ID` - artifact id of the project with contracts.
- `EXTERNAL_CONTRACTS_CLASSIFIER` - classifier of the project with contracts. Empty by default
- `EXTERNAL_CONTRACTS_VERSION` - version of the project with contracts. Defaults to `+`, equivalent to picking the latest
- `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL` - URL of your Artifact Manager. Defaults to value of `REPO_WITH_BINARIES_URL` env var. If that’s not set, defaults to `http://localhost:8081/artifactory/libs-release-local` which is the default URL of Artifactory running locally
- `EXTERNAL_CONTRACTS_PATH` - path to contracts for the given project, inside the project with contracts. Defaults to slash separated `EXTERNAL_CONTRACTS_GROUP_ID` concatenated with `/` and `EXTERNAL_CONTRACTS_ARTIFACT_ID`. E.g. for group id `foo.bar` and artifact id `baz`, would result in `foo/bar/baz` contracts path.
- `EXTERNAL_CONTRACTS_WORK_OFFLINE` - if set to `true` then will retrieve artifact with contracts from the container’s `.m2`. Mount your local `.m2` as a volume available at the container’s `/root/.m2` path. You must not set both `EXTERNAL_CONTRACTS_WORK_OFFLINE` and `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL`.

这些环境变量在执行测试时使用：

- `APPLICATION_BASE_URL` - url against which tests should be executed. Remember that it has to be accessible from the Docker container (e.g. `localhost` will not work)
- `APPLICATION_USERNAME` - (optional) username for basic authentication to your application
- `APPLICATION_PASSWORD` - (optional) password for basic authentication to your application

87.6.3 Example of usage译: 87.6.3使用示例

让我们看看一个简单的MVC应用程序

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

合同可在 `/contracts` 文件夹下 `/contracts` 。

87.6.4 Server side (nodejs) 译：87.6.4服务器端（nodejs）

由于我们想运行测试，我们可以执行：

```
$ npm test
```

然而，出于学习目的，我们将其分成几部分：

```
# Stop docker infra (nodejs, artifactory)
$ ./stop_infra.sh
# Start docker infra (nodejs, artifactory)
$ ./setup_infra.sh

# Kill & Run app
$ pkill -f "node app"
$ nohup node app &

# Prepare environment variables
$ SC_CONTRACT_DOCKER_VERSION="..."
$ APP_IP="192.168.0.100"
$ APP_PORT="3000"
$ ARTIFACTORY_PORT="8081"
$ APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
$ ARTIFACTORY_URL="http://${APP_IP}:${ARTIFACTORY_PORT}/artifactory/libs-release-local"
$ CURRENT_DIR="$( pwd )"
$ CURRENT_FOLDER_NAME=${PWD##*/}
$ PROJECT_VERSION="0.0.1.RELEASE"

# Execute contract tests
$ docker run  --rm -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" -e "PUBLISH_ARTIFACTS=true" -e "PROJECT_NAME=${CURRENT_FOLDER_NAME}" -e "REPO_WI

# Kill app
$ pkill -f "node app"
```

会发生什么是通过bash脚本：

- infrastructure will be set up (MongoDb, Artifactory). In real life scenario you would just run the NodeJS application with mocked database. In this example we want to show how we can benefit from Spring Cloud Contract in no time.
- 由于这些限制，合同也代表了有状态的情况
  - first request is a `POST` that causes data to get inserted to the database
  - second request is a `GET` that returns a list of data with 1 previously inserted element
- the NodeJS application will be started (on port `3000`)
- 合约测试将通过Docker生成，测试将针对正在运行的应用程序执行
  - the contracts will be taken from `/contracts` folder.
  - the output of the test execution is available under `node_modules/spring-cloud-contract/output`.
- the stubs will be uploaded to Artifactory. You can check them out under http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/ . The stubs will be here http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar.

要了解客户端的样子，请查看 Section 89.9, “Stub Runner Docker”部分。

88. Spring Cloud Contract Verifier Messaging 译：88. Spring Cloud Contract Verifier消息传递

Spring Cloud Contract Verifier允许您验证使用消息传递作为通信手段的应用程序。 本文档中显示的所有集成均可与Spring一起使用，但您也可以创建自己的一个并使用它。

88.1 Integrations 译：88.1集成

您可以使用以下四种集成配置之一：

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring AMQP

由于我们使用Spring Boot，因此如果您已将其中一个库添加到类路径中，则会自动设置所有消息传递配置。

Important

请记住将`@AutoConfigureMessageVerifier`放在生成的测试的基类上。 否则，Spring Cloud Contract Verifier的消息传递部分不起作用。

Important

如果您想使用Spring Cloud Stream，请记住添加对 `org.springframework.cloud:spring-cloud-stream-test-support`的依赖关系，如下所示：

Maven的。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

摇篮。

```
testCompile "org.springframework.cloud:spring-cloud-stream-test-support"
```

88.2 Manual Integration Testing译: 88.2手动集成测试

测试使用的主界面是`org.springframework.cloud.contract.verifier.messaging.MessageVerifier` 。 它定义了如何发送和接收消息。 您可以创建自己的实现来实现相同的目标。

在测试中，您可以注入`ContractVerifierMessageExchange`以发送和接收遵循合同的消息。 然后将`@AutoConfigureMessageVerifier`添加到您的测试中。 这是一个例子：

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

  @Autowired
  private MessageVerifier verifier;
  ...
}
```

如果您的测试也需要存根，那么`@AutoConfigureStubRunner`包含消息传递配置，因此您只需要一个注释。

88.3 Publisher-Side Test Generation译: 88.3发行方测试生成

在您的DSL中拥有`input`或`outputMessage`部分会导致在发布者端创建测试。 默认情况下，创建JUnit测试。 但是，也有可能创建Spock测试。

我们应该考虑三种主要场景：

- Scenario 1: There is no input message that produces an output message. The output message is triggered by a component inside the application (for example, scheduler).
- Scenario 2: The input message triggers an output message.
- Scenario 3: The input message is consumed and there is no output message.

Important

传递给`messageFrom`或`sentTo`的目标对于不同的消息传递实现可以具有不同的含义。 对于**Stream**和**Integration**，它首先被解析为一个频道的`destination` 。 然后，如果没有这样的`destination`它将被解析为频道名称。 对于**骆驼来说** ，这是一个特定的组件（例如， `jms` ）。

88.3.1 Scenario 1: No Input Message译: 88.3.1情况1：没有输入消息

对于给定的合同：

Groovy DSL。

```
def contractDsl = Contract.make {
 label 'some_label'
 input {
  triggeredBy('bookReturnedTriggered()')
 }
 outputMessage {
  sentTo('activemq:output')
  body('''{ "bookName" : "foo" }''')
  headers {
   header('BOOK-NAME', 'foo')
   messagingContentType(applicationJson())
  }
 }
}
```

YAML。

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
    contentType: application/json
```

以下JUnit测试已创建：

```
'''
 // when:
  bookReturnedTriggered();

 // then:
  ContractVerifierMessage response = contractVerifierMessaging.receive("activemq:output");
  assertThat(response).isNotNull();
  assertThat(response.getHeader("BOOK-NAME")).isNotNull();
  assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");
  assertThat(response.getHeader("contentType")).isNotNull();
  assertThat(response.getHeader("contentType").toString()).isEqualTo("application/json");
 // and:
  DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
  assertThatJson(parsedJson).field("bookName").isEqualTo("foo");
'''
```

并且将创建以下Spock测试：

```
'''
 when:
  bookReturnedTriggered()

 then:
  ContractVerifierMessage response = contractVerifierMessaging.receive('activemq:output')
  assert response != null
  response.getHeader('BOOK-NAME')?.toString()  == 'foo'
  response.getHeader('contentType')?.toString()  == 'application/json'
 and:
  DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))
  assertThatJson(parsedJson).field("bookName").isEqualTo("foo")

'''
```

88.3.2 Scenario 2: Output Triggered by Input 译：88.3.2场景2：由输入触发的输出

对于给定的合同：

Groovy DSL。

```
def contractDsl = Contract.make {
 label 'some_label'
 input {
  messageFrom('jms:input')
  messageBody([
    bookName: 'foo'
  ])
  messageHeaders {
   header('sample', 'header')
  }
 }
 outputMessage {
  sentTo('jms:output')
  body([
    bookName: 'foo'
  ])
  headers {
   header('BOOK-NAME', 'foo')
  }
 }
}
```

YAML。

```
label: some_label
input:
  messageFrom: jms:input
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
outputMessage:
  sentTo: jms:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
```

以下JUnit测试已创建：

```
'''
// given:
 ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
  "{\\"bookName\\":\\"foo\\"}"
, headers()
  .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage, "jms:input");

// then:
 ContractVerifierMessage response = contractVerifierMessaging.receive("jms:output");
 assertThat(response).isNotNull();
 assertThat(response.getHeader("BOOK-NAME")).isNotNull();
 assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");
// and:
 DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
 assertThatJson(parsedJson).field("bookName").isEqualTo("foo");
'''
```

并且将创建以下Spock测试：

```
"""\
given:
   ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
    '''{"bookName":"foo"}''',
    ['sample': 'header']
  )

when:
   contractVerifierMessaging.send(inputMessage, 'jms:input')

then:
   ContractVerifierMessage response = contractVerifierMessaging.receive('jms:output')
   assert response != null
   response.getHeader('BOOK-NAME')?.toString()  == 'foo'
and:
   DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))
   assertThatJson(parsedJson).field("bookName").isEqualTo("foo")
"""
```

88.3.3 Scenario 3: No Output Message译：88.3.3情况3：无输出消息

对于给定的合同：

Groovy DSL。

```
def contractDsl = Contract.make {
 label 'some_label'
 input {
  messageFrom('jms:delete')
  messageBody([
    bookName: 'foo'
  ])
  messageHeaders {
   header('sample', 'header')
  }
  assertThat('bookWasDeleted()')
 }
}
```

YAML。

```
label: some_label
input:
  messageFrom: jms:delete
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
  assertThat: bookWasDeleted()
```

以下JUnit测试已创建：

```
'''
// given:
 ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
 "{\\"bookName\\":\\"foo\\"}"
, headers()
 .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage, "jms:delete");

// then:
 bookWasDeleted();
'''
```

并且将创建以下Spock测试：

```
'''
given:
  ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
  \'\'\'{"bookName":"foo"}\'\'\',
  ['sample': 'header']
 )

when:
  contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:
  noExceptionThrown()
  bookWasDeleted()
'''
```

88.4 Consumer Stub Generation译：88.4消费者存根生成

与HTTP部分不同，在消息传递中，我们需要使用存根发布JAR中的Groovy DSL。 然后在消费者端进行解析并创建适当的存根路线。

有关更多信息，请参阅 the Stub Runner Messaging sections 。

Maven的。

```
<dependencies>
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
 </dependency>

 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
 </dependency>
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-test-support</artifactId>
  <scope>test</scope>
 </dependency>
</dependencies>

<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>Finchley.BUILD-SNAPSHOT</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

摇篮。

```
ext {
 contractsDir = file("mappings")
 stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin but you can modify it as you wish

publishing {
 publications {
  stubs(MavenPublication) {
   artifactId "${project.name}-stubs"
   artifact verifierStubsJar
  }
 }
}
```

89. Spring Cloud Contract Stub Runner译：89.春云合同存根更军

使用Spring Cloud Contract Verifier时可能遇到的问题之一是将生成的WireMock JSON存根从服务器端传递到客户端（或各种客户端）。 用于消息传送的客户端生成方式也是如此。

复制JSON文件并手动设置客户端进行消息传递是不可能的。 这就是为什么我们推出Spring Cloud Contract Stub Runner。 它可以自动下载并运行存根。

89.1 Snapshot versions译：89.1快照版本

将其他快照存储库添加到`build.gradle`文件以使用快照版本，每次成功构建后都会自动上传快照版本：

Maven的。

```
<repositories>
 <repository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
  <id>spring-snapshots</id>
  <name>Spring Snapshots</name>
  <url>https://repo.spring.io/snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-milestones</id>
  <name>Spring Milestones</name>
  <url>https://repo.spring.io/milestone</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
 <pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </pluginRepository>
</pluginRepositories>
```

摇篮。

```
buildscript {
 repositories {
  mavenCentral()
  mavenLocal()
  maven { url "http://repo.spring.io/snapshot" }
  maven { url "http://repo.spring.io/milestone" }
  maven { url "http://repo.spring.io/release" }
 }
```

89.2 Publishing Stubs as JARs译：89.2将存根发布为JAR

最简单的方法是集中存根的方式。 例如，您可以将它们保存为Maven存储库中的jar。

对于Maven和Gradle来说，安装程序已准备就绪。 但是，如果您愿意，可以自定义它。

Maven的。

```
<!-- First disable the default jar setup in the properties section -->
<!-- we don't want the verifier to do a jar for us -->
<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>

<!-- Next add the assembly plugin to your build -->
<!-- we want the assembly plugin to generate the JAR -->
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-assembly-plugin</artifactId>
 <executions>
  <execution>
   <id>stub</id>
   <phase>prepare-package</phase>
   <goals>
    <goal>single</goal>
   </goals>
   <inherited>false</inherited>
   <configuration>
    <attach>true</attach>
    <descriptors>
     $../../../../src/assembly/stub.xml
    </descriptors>
   </configuration>
  </execution>
 </executions>
</plugin>

<!-- Finally setup your assembly. Below you can find the contents of src/main/assembly/stub.xml -->
<assembly
 xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
 <id>stubs</id>
 <formats>
  <format>jar</format>
 </formats>
 <includeBaseDirectory>false</includeBaseDirectory>
 <fileSets>
  <fileSet>
   <directory>src/main/java</directory>
   <outputDirectory>/</outputDirectory>
   <includes>
    <include>**com/example/model/*.*</include>
   </includes>
  </fileSet>
  <fileSet>
   <directory>${project.build.directory}/classes</directory>
   <outputDirectory>/</outputDirectory>
   <includes>
    <include>**com/example/model/*.*</include>
   </includes>
  </fileSet>
  <fileSet>
   <directory>${project.build.directory}/snippets/stubs</directory>
   <outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDirectory>
   <includes>
    <include>**/*</include>
   </includes>
  </fileSet>
  <fileSet>
   <directory>$../../../../src/test/resources/contracts</directory>
   <outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputDirectory>
   <includes>
    <include>**/*.groovy</include>
   </includes>
  </fileSet>
 </fileSets>
</assembly>
```

摇篮。

```
ext {
 contractsDir = file("mappings")
 stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin but you can modify it as you wish

publishing {
 publications {
  stubs(MavenPublication) {
   artifactId "${project.name}-stubs"
   artifact verifierStubsJar
  }
 }
}
```

89.3 Stub Runner Core译：89.3 Stub Runner核心

运行服务协作者的存根。 作为服务合同处理存根允许使用存根运行者作为Consumer Driven Contracts的实现。

Stub Runner允许您自动下载提供的依赖关系的存根（或从类路径中选择这些存根），为他们启动WireMock服务器并为它们提供适当的存根定义。 对于消息传递，定义了特殊的存根路线。

89.3.1 Retrieving stubs译:89.3.1检索存根

您可以选择以下选项来获取存根

- Aether based solution that downloads JARs with stubs from Artifactory / Nexus
- Classpath scanning solution that searches classpath via pattern to retrieve stubs
- Write your own implementation of the `org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder` for full customization

后面的例子在 Custom Stub Runner部分进行了描述。

Stub downloading译:存根下载

您可以通过`stubsMode`交换机控制存根下载。 它从`StubRunnerProperties.StubsMode`枚举`StubRunnerProperties.StubsMode` 。 您可以使用以下选项

- `StubRunnerProperties.StubsMode.CLASSPATH` (default value) - will pick stubs from the classpath
- `StubRunnerProperties.StubsMode.LOCAL` - will pick stubs from a local storage (e.g. `.m2`)
- `StubRunnerProperties.StubsMode.REMOTE` - will pick stubs from a remote location

例：

```
@AutoConfigureStubRunner(repositoryRoot="http://foo.bar", ids = "com.example:beer-api-producer:+:stubs:8095", stubsMode = StubRunnerProperties.StubsM
```

Classpath scanning译:类路径扫描

如果将`stubsMode`属性设置为`StubRunnerProperties.StubsMode.CLASSPATH` （或者由于`CLASSPATH`是默认值，`StubRunnerProperties.StubsMode.CLASSPATH`设置任何内容），则将扫描classpath。 让我们看看下面的例子：

```
@AutoConfigureStubRunner(ids = {
    "com.example:beer-api-producer:+:stubs:8095",
    "com.example.foo:bar:1.0.0:superstubs:8096"
})
```

如果你已经添加了依赖到你的类路径

Maven的。

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>beer-api-producer-restdocs</artifactId>
    <classifier>stubs</classifier>
    <version>0.0.1-SNAPSHOT</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.example.foo</groupId>
    <artifactId>bar</artifactId>
    <classifier>superstubs</classifier>
    <version>1.0.0</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

摇篮。

```
testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {
    transitive = false
}
testCompile("com.example.foo:bar:1.0.0:superstubs") {
    transitive = false
}
```

然后，您的类路径中的以下位置将被扫描。 为`com.example:beer-api-producer-restdocs`

- /META-INF/com.example/beer-api-producer-restdocs/**/*.*
- /contracts/com.example/beer-api-producer-restdocs/**/*.*
- /mappings/com.example/beer-api-producer-restdocs/**/*.*

和 `com.example.foo:bar`

- /META-INF/com.example.foo/bar/**/*.*
- /contracts/com.example.foo/bar/**/*.*
- /mappings/com.example.foo/bar/**/*.*

正如你所看到的，当打包生产者存根时，你必须明确地提供组和工件ID。

生产者会设置这样的合同：

```
└── src
    └── test
        └── resources
            └── contracts
                └── com.example
                    └── beer-api-producer-restdocs
                        └── nested
                            └── contract3.groovy
```

实现适当的存根包装。

或者使用 Maven `assembly` plugin或 Gradle Jar任务，您必须在存根jar中创建以下结构。

```
└── META-INF
    └── com.example
        └── beer-api-producer-restdocs
            └── 2.0.0
                ├── contracts
                │   └── nested
                │       └── contract2.groovy
                └── mappings
                    └── mapping.json
```

通过维护此结构，可以扫描类路径，并且您可以从消息传递/ HTTP存根中获益，而无需下载工件。

89.3.2 Running stubs译：89.3.2运行存根

Limitations译:限制

Important

StubRunner可能会在测试之间关闭端口时出现问题。 您可能会遇到导致端口冲突的情况。 只要你在测试中使用相同的上下文，一切正常。 但是当上下文不同时（例如不同的存根或不同的配置文件），则必须使用`@DirtiesContext`关闭存根服务器，或者在每个测试的不同端口上运行它们。

Running using main app译:使用主应用程序运行

您可以将以下选项设置为主类：

```
-c, --classifier               Suffix for the jar containing stubs (e.
                                 g. 'stubs' if the stub jar would
                                 have a 'stubs' classifier for stubs:
                                 foobar-stubs ). Defaults to 'stubs'
                                 (default: stubs)
--maxPort, --maxp <Integer>    Maximum port value to be assigned to
                                 the WireMock instance. Defaults to
                                 15000 (default: 15000)
--minPort, --minp <Integer>    Minimum port value to be assigned to
                                 the WireMock instance. Defaults to
                                 10000 (default: 10000)
-p, --password                 Password to user when connecting to
                                 repository
--phost, --proxyHost           Proxy host to use for repository
                                 requests
--pport, --proxyPort [Integer] Proxy port to use for repository
                                 requests
-r, --root                     Location of a Jar containing server
                                 where you keep your stubs (e.g. http:
                                 //nexus.
                                 net/content/repositories/repository)
-s, --stubs                    Comma separated list of Ivy
                                 representation of jars with stubs.
                                 Eg. groupid:artifactid1,groupid2:
                                 artifactid2:classifier
--sm, --stubsMode              Stubs mode to be used. Acceptable values
                                 [CLASSPATH, LOCAL, REMOTE]
-u, --username                 Username to user when connecting to
                                 repository
```

HTTP Stubs译：HTTP存根

存根在JSON文档中定义，其语法在 WireMock documentation中定义

例：

```
{
    "request": {
        "method": "GET",
        "url": "/ping"
    },
    "response": {
        "status": 200,
        "body": "pong",
        "headers": {
            "Content-Type": "text/plain"
        }
    }
}
```

Viewing registered mappings译:查看注册的映射

每个存根协作者都暴露`__/admin/`端点下的已定义映射列表。

您还可以使用`mappingsOutputFolder`属性将映射转储到文件。 对于基于注解的方法，它看起来像这样

```
@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",
mappingsOutputFolder = "target/outputmappings/")
```

对于像这样的JUnit方法：

```
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
   .repoRoot("http://some_url")
   .downloadStub("a.b.c", "loanIssuance")
   .downloadStub("a.b.c:fraudDetectionServer")
   .withMappingsOutputFolder("target/outputmappings")
```

然后，如果您检出文件夹`target/outputmappings`您会看到以下结构

```
.
├── fraudDetectionServer_13705
└── loanIssuance_12255
```

这意味着有两个存根登记。`fraudDetectionServer`已在端口`13705`和`loanIssuance`在端口`12255`。如果我们看一下我们将看到的一个文件（对于WireMock）给定服务器可用的映射：

```
[{
  "id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7",
  "request" : {
    "url" : "/name",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "fraudDetectionServer"
  },
  "uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"
},
...
]
```

Messaging Stubs译:消息存根

根据提供的Stub Runner依赖关系和DSL，消息路由会自动设置。

89.4 Stub Runner JUnit Rule译: 89.4 Stub Runner JUnit规则

Stub Runner附带了JUnit规则，因此您可以轻松下载并运行给定组和工件标识的存根：

```
@ClassRule public static StubRunnerRule rule = new StubRunnerRule()
  .repoRoot(repoRoot())
  .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
  .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer");
```

在执行该规则之后，Stub Runner连接到您的Maven存储库，并且给定的依赖关系列表尝试：

- download them
- cache them locally
- unzip them to a temporary folder
- start a WireMock server for each Maven dependency on a random port from the provided range of ports / provided port
- feed the WireMock server with all JSON files that are valid WireMock definitions
- can also send messages (remember to pass an implementation of `MessageVerifier` interface)

Stub Runner使用Eclipse Aether机制下载Maven依赖关系。检查他们的docs获取更多信息。

由于`StubRunnerRule`实现了`StubFinder`它允许您查找已启动的存根：

```
package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

public interface StubFinder extends StubTrigger {
 /**
  * For the given groupId and artifactId tries to find the matching
  * URL of the running stub.
  *
  * @param groupId - might be null. In that case a search only via artifactId takes place
  * @return URL of a running stub or throws exception if not found
  */
 URL findStubUrl(String groupId, String artifactId) throws StubNotFoundException;

 /**
  * For the given Ivy notation {@code [groupId]:artifactId:[version]:[classifier]} tries to
  * find the matching URL of the running stub. You can also pass only {@code artifactId}.
  *
  * @param ivyNotation - Ivy representation of the Maven artifact
  * @return URL of a running stub or throws exception if not found
  */
 URL findStubUrl(String ivyNotation) throws StubNotFoundException;

 /**
  * Returns all running stubs
  */
 RunningStubs findAllRunningStubs();

 /**
  * Returns the list of Contracts
  */
 Map<StubConfiguration, Collection<Contract>> getContracts();
}
```

Spock测试中的使用示例：

```
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
  .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
  .repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())
  .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
  .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
  .withMappingsOutputFolder("target/outputmappingsforrule")


def 'should start WireMock servers'() {
 expect: 'WireMocks are running'
  rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null
  rule.findStubUrl('loanIssuance') != null
  rule.findStubUrl('loanIssuance') == rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')
  rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
 and:
  rule.findAllRunningStubs().isPresent('loanIssuance')
  rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')
  rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
 and: 'Stubs were registered'
  "${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
  "${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
}

def 'should output mappings to output folder'() {
 when:
  def url = rule.findStubUrl('fraudDetectionServer')
 then:
  new File("target/outputmappingsforrule", "fraudDetectionServer_${url.port}").exists()
}
```

JUnit测试中的使用示例：

```
@Test
public void should_start_wiremock_servers() throws Exception {
 // expect: 'WireMocks are running'
  then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")).isNotNull();
  then(rule.findStubUrl("loanIssuance")).isNotNull();
  then(rule.findStubUrl("loanIssuance")).isEqualTo(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs", "loanIssuance"));
  then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isNotNull();
 // and:
  then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();
  then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs", "fraudDetectionServer")).isTrue();
  then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isTrue();
 // and: 'Stubs were registered'
  then(httpGet(rule.findStubUrl("loanIssuance").toString() + "/name")).isEqualTo("loanIssuance");
  then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() + "/name")).isEqualTo("fraudDetectionServer");
}
```

有关如何应用Stub Runner的全局配置的更多信息，请查看 **JUnit和Spring的 Common properties** 。

Important

要将JUnit规则与消息一起使用，必须提供规则构建器的`MessageVerifier`接口的实现（例如`rule.messageVerifier(new MyMessageVerifier())`）。 如果你不这样做，那么每当你尝试发送消息时都会抛出异常。

89.4.1 Maven settings

存根下载器授予其他本地存储库文件夹的Maven设置。 目前没有考虑存储库和配置文件的身份验证详细信息，因此您需要使用上述属性指定它。

89.4.2 Providing fixed ports 译：89.4.2提供固定端口

您也可以在固定端口上运行存根。 你可以用两种不同的方式来做到这一点。 一种是通过属性传递它，另一种是通过JUnit规则的流畅API。

89.4.3 Fluent API 译：89.4.3流利的API

使用 `StubRunnerRule` ，可以添加一个存根以下载，然后传递最后下载的存根的端口。

```
@ClassRule public static StubRunnerRule rule = new StubRunnerRule()
  .repoRoot(repoRoot())
  .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
  .withPort(12345)
  .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:12346");
```

您可以看到，对于此示例，以下测试是有效的：

```
then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.create("http://localhost:12345").toURL());
then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(URI.create("http://localhost:12346").toURL());
```

89.4.4 Stub Runner with Spring 译：89.4.4带弹簧的Stub Runner

设置Stub Runner项目的Spring配置。

通过在配置文件中提供存根的列表，存根运行程序会自动下载并在WireMock中注册选定的存根。

如果你想找到你的 `StubFinder` 依赖关系的URL，你可以自动装配 `StubFinder` 接口并使用它的方法，如下所示：

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest(properties = [" stubrunner.cloud.enabled=false", 'foo=${stubrunner.runningstubs.fraudDetectionServer.port}', 'fooWithGroup=${stubrunn
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/")
@DirtiesContext
@ActiveProfiles("test")
class StubRunnerConfigurationSpec extends Specification {

 @Autowired StubFinder stubFinder
 @Autowired Environment environment
 @StubRunnerPort("fraudDetectionServer") int fraudDetectionServerPort
 @StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer") int fraudDetectionServerPortWithGroupId
 @Value('${foo}') Integer foo

 @BeforeClass
 @AfterClass
 void setupProps() {
  System.clearProperty("stubrunner.repository.root")
  System.clearProperty("stubrunner.classifier")
 }

 def 'should start WireMock servers'() {
  expect: 'WireMocks are running'
   stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null
   stubFinder.findStubUrl('loanIssuance') != null
   stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')
   stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')
   stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') == stubFinder.findStubUrl('org.springframe
   stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
  and:
   stubFinder.findAllRunningStubs().isPresent('loanIssuance')
   stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')
   stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
  and: 'Stubs were registered'
   "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
   "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
 }

 def 'should throw an exception when stub is not found'() {
  when:
   stubFinder.findStubUrl('nonExistingService')
  then:
   thrown(StubNotFoundException)
  when:
   stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId')
  then:
   thrown(StubNotFoundException)
 }

 def 'should register started servers as environment variables'() {
  expect:
   environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null
   stubFinder.findAllRunningStubs().getPort("loanIssuance") == (environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)
  and:
   environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null
   stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port")
  and:
   environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null
   stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.co
 }

 def 'should be able to interpolate a running stub in the passed test property'() {
  given:
   int fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
  expect:
   fraudPort > 0
   environment.getProperty("foo", Integer) == fraudPort
   environment.getProperty("fooWithGroup", Integer) == fraudPort
   foo == fraudPort
 }

 @Issue("#573")
 def 'should be able to retrieve the port of a running stub via an annotation'() {
  given:
   int fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
  expect:
   fraudPort > 0
   fraudDetectionServerPort == fraudPort
   fraudDetectionServerPortWithGroupId == fraudPort
 }

 def 'should dump all mappings to a file'() {
  when:
   def url = stubFinder.findStubUrl("fraudDetectionServer")
  then:
   new File("target/outputmappings/", "fraudDetectionServer_${url.port}").exists()
 }

 @Configuration
 @EnableAutoConfiguration
 static class Config {}
}
```

对于以下配置文件：

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    - org.springframework.cloud.contract.verifier.stubs:loanIssuance
    - org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
    - org.springframework.cloud.contract.verifier.stubs:bootService
  stubs-mode: remote
```

您也可以使用`@AutoConfigureStubRunner`内的属性来代替使用属性。 您可以在下面找到一个通过设置注释值获得相同结果的示例。

```
@AutoConfigureStubRunner(
  ids = ["org.springframework.cloud.contract.verifier.stubs:loanIssuance",
  "org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",
  "org.springframework.cloud.contract.verifier.stubs:bootService"],
  stubsMode = StubRunnerProperties.StubsMode.REMOTE,
  repositoryRoot = "classpath:m2repo/repository/")
```

Stub Runner Spring为每个注册的WireMock服务器按以下方式注册环境变量。 Stub Runner ID `com.example:foo` `com.example:bar` 。

- `stubrunner.runningstubs.foo.port`
- `stubrunner.runningstubs.com.example.foo.port`
- `stubrunner.runningstubs.bar.port`
- `stubrunner.runningstubs.com.example.bar.port`

你可以在你的代码中引用它。

您还可以使用`@StubRunnerPort`注释来注入运行存根的端口。 注释的值可以是`groupid:artifactid`也可以是`artifactid` 。 Stub Runner ID `com.example:foo` `com.example:bar` 。

```
@StubRunnerPort("foo")
int fooPort;
@StubRunnerPort("com.example:bar")
int barPort;
```

89.5 Stub Runner Spring Cloud 译: 89.5存根运行 春云

Stub Runner可以与Spring Cloud集成。

对于真实生活的例子，你可以检查

- producer app sample
- consumer app sample

89.5.1 Stubbing Service Discovery 译: 89.5.1存根服务发现

`Stub Runner Spring Cloud`最重要的特征是它的`Stub Runner Spring Cloud`

- `DiscoveryClient`
- `Ribbon` `ServerList`

这意味着无论您是使用Zookeeper，Consul，Eureka还是其他任何软件，您都不需要在测试中使用它。 我们重新启动您的依赖关系的WireMock实例，并且无论您何时使用`Feign` （负载均衡`RestTemplate`或`DiscoveryClient`直接调用那些存根服务器，而不是调用真正的服务发现工具，我们都会告诉您的应用程序。

例如，这个测试会通过

```
def 'should make service discovery work'() {
 expect: 'WireMocks are running'
  "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
  "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
 and: 'Stubs can be reached via load service discovery'
  restTemplate.getForObject('http://loanIssuance/name', String) == 'loanIssuance'
  restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name', String) == 'fraudDetectionServer'
}
```

为以下配置文件

```
stubrunner:
  idsToServiceIds:
    ivyNotation: someValueInsideYourCode
    fraudDetectionServer: someNameThatShouldMapFraudDetectionServer
```

Test profiles and service discovery 译:测试配置文件和服务发现

在您的集成测试中，您通常不希望既不呼叫发现服务（例如Eureka），也不想调用配置服务器。 这就是为什么你要创建一个你想禁用这些功能的额外测试配置。

由于`spring-cloud-commons`的某些限制来实现这一点，您可以通过静态块禁用这些属性，如下所示（Eureka示例）

```
    //Hack to work around https://github.com/spring-cloud/spring-cloud-commons/issues/156
    static {
        System.setProperty("eureka.client.enabled", "false");
        System.setProperty("spring.cloud.config.failFast", "false");
    }
```

89.5.2 Additional Configuration 译: 89.5.2其他配置

您可以使用`stubrunner.idsToServiceIds:`地图将存根的artifactId与应用的名称进行`stubrunner.idsToServiceIds:` 。 您可以通过提供：`stubrunner.cloud.ribbon.enabled`等于`false`来禁用Stub Runner Ribbon支持您可以通过提供以下`stubrunner.cloud.ribbon.enabled`来禁用Stub Runner支持： `stubrunner.cloud.enabled`等于`false`

> 默认情况下，所有服务发现都将被存根。 这意味着无论事实如果你有一个现有的`DiscoveryClient`其结果将被忽略。 但是，如果要重新使用它，只需将`stubrunner.cloud.delegate.enabled`设置为`true`，然后将现有的`DiscoveryClient`结果与`DiscoveryClient`合并。

Stub Runner使用的默认Maven配置可以通过以下系统属性或环境变量进行调整

- `maven.repo.local` - path to the custom maven local repository location
- `org.apache.maven.user-settings` - path to custom maven user settings location
- `org.apache.maven.global-settings` - path to maven global settings location

89.6 Stub Runner Boot Application 译：89.6 Stub Runner启动应用程序

Spring Cloud Contract Stub Runner Boot是Spring Boot应用程序，它公开REST端点以触发消息传递标签并访问启动的WireMock服务器。

其中一个用例是在部署的应用程序上运行一些烟雾（端对端）测试。 您可以查看Spring Cloud Pipelines项目获取更多信息。

89.6.1 How to use it? 译：89.6.1如何使用它？

Stub Runner Server 译：存根运行服务器

只需添加

```
compile "org.springframework.cloud:spring-cloud-starter-stub-runner"
```

使用`@EnableStubRunnerServer`注释一个类，创建一个胖罐，然后您就可以开始了！

对于属性检查 **Stub Runner Spring**部分。

Stub Runner Server Fat Jar 译：存根（stub）跑步者服务器脂肪罐

您可以从Maven下载独立JAR（例如，版本1.2.3.RELEASE），如下所示：

```
$ wget -O stub-runner.jar 'https://search.maven.org/remote_content?g=org.springframework.cloud&a=spring-cloud-contract-stub-runner-boot&v=1.2.3.RELEA
$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...
```

Spring Cloud CLI 译：Spring Cloud CLI

从 Spring Cloud CLI项目的`1.4.0.RELEASE`版本开始，您可以通过执行`spring cloud stubrunner`来启动Stub Runner Boot。

为了通过配置，只需在当前工作目录或`config`或`~/.spring-cloud`的子目录中创建一个`stubrunner.yml`文件。 该文件可能看起来像这样（本地安装的运行存根的示例）

stubrunner.yml。

```
stubrunner:
  stubsMode: LOCAL
  ids:
    - com.example:beer-api-producer:+:9876
```

然后在终端窗口中调用`spring cloud stubrunner`以启动Stub Runner服务器。 它将在`8750`港口`8750`。

89.6.2 Endpoints 译：89.6.2端点

HTTP 译：HTTP

- GET `/stubs` - returns a list of all running stubs in `ivy:integer` notation
- GET `/stubs/{ivy}` - returns a port for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

Messaging 译：消息

用于消息传递

- GET `/triggers` - returns a list of all running labels in `ivy : [ label1, label2 … ]` notation
- POST `/triggers/{label}` - executes a trigger with `label`
- POST `/triggers/{ivy}/{label}` - executes a trigger with `label` for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

89.6.3 Example 译：89.6.3示例

```
@ContextConfiguration(classes = StubRunnerBoot, loader = SpringBootContextLoader)
@SpringBootTest(properties = "spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec extends Specification {

 @Autowired StubRunning stubRunning

 def setup() {
  RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
    new TriggerController(stubRunning))
 }

 def 'should return a list of running stub servers in "full ivy:port" notation'() {
  when:
   String response = RestAssuredMockMvc.get('/stubs').body.asString()
  then:
   def root = new JsonSlurper().parseText(response)
   root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs' instanceof Integer
 }

 def 'should return a port on which a [#stubId] stub is running'() {
  when:
   def response = RestAssuredMockMvc.get("/stubs/${stubId}")
  then:
   response.statusCode == 200
   Integer.valueOf(response.body.asString()) > 0
  where:
   stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:+:stubs',
        'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',
        'org.springframework.cloud.contract.verifier.stubs:bootService:+',
        'org.springframework.cloud.contract.verifier.stubs:bootService',
        'bootService']
 }

 def 'should return 404 when missing stub was called'() {
  when:
   def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")
  then:
   response.statusCode == 404
 }

 def 'should return a list of messaging labels that can be triggered when version and classifier are passed'() {
  when:
   String response = RestAssuredMockMvc.get('/triggers').body.asString()
  then:
   def root = new JsonSlurper().parseText(response)
   root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs'?.containsAll(["delete_book","return_book_1","return_book
 }

 def 'should trigger a messaging label'() {
  given:
   StubRunning stubRunning = Mock()
   RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning), new TriggerController(stubRunning))
  when:
   def response = RestAssuredMockMvc.post("/triggers/delete_book")
  then:
   response.statusCode == 200
  and:
   1 * stubRunning.trigger('delete_book')
 }

 def 'should trigger a messaging label for a stub with [#stubId] ivy notation'() {
  given:
   StubRunning stubRunning = Mock()
   RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning), new TriggerController(stubRunning))
  when:
   def response = RestAssuredMockMvc.post("/triggers/$stubId/delete_book")
  then:
   response.statusCode == 200
  and:
   1 * stubRunning.trigger(stubId, 'delete_book')
  where:
   stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:stubs', 'org.springframework.cloud.contract.verifier.stubs:bootService',
 }

 def 'should throw exception when trigger is missing'() {
  when:
   RestAssuredMockMvc.post("/triggers/missing_label")
  then:
   Exception e = thrown(Exception)
   e.message.contains("Exception occurred while trying to return [missing_label] label.")
   e.message.contains("Available labels are")
   e.message.contains("org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs=[]")
   e.message.contains("org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs=")
 }

}
```

89.6.4 Stub Runner Boot with Service Discovery译：89.6.4带有服务发现的Stub Runner引导

使用Stub Runner Boot的一个可能性是将其用作“烟雾测试”的存根的馈送。 这是什么意思？ 假设您不想将50个微服务部署到测试环境中，以检查您的应用程序是否正常工作。 您在构建过程中已经执行了一系列测试，但您也希望确保应用程序的打包状态良好。 你可以做的是将你的应用程序部署到一个环境中，启动它并运行一些测试来看它是否工作正常。 我们可以称这些测试为烟雾测试，因为他们的想法是只检查少数测试场景。

这种方法的问题在于，如果您正在使用微服务，则很可能您正在使用服务发现工具。 Stub Runner Boot允许您通过启动所需的存根并将其注册到服务发现工具中来解决此问题。 让我们来看看Eureka这样一个设置的例子。 让我们假设尤里卡已经在跑步了。

```
@SpringBootApplication
@EnableStubRunnerServer
@EnableEurekaClient
@AutoConfigureStubRunner
public class StubRunnerBootEurekaExample {

 public static void main(String[] args) {
  SpringApplication.run(StubRunnerBootEurekaExample.class, args);
 }

}
```

正如您所看到的，我们希望启动存根运行引导服务器`@EnableStubRunnerServer`，启用Eureka客户端`@EnableEurekaClient`并且我们希望启用存根运行功能`@AutoConfigureStubRunner`。

现在让我们假设我们要启动这个应用程序，以便存根被自动注册。 我们可以通过运行应用程序`java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar`，其中`${SYSTEM_PROPS}`将包含以下属性列表

```
-Dstubrunner.repositoryRoot=http://repo.spring.io/snapshots (1)
-Dstubrunner.cloud.stubbed.discovery.enabled=false (2)
-Dstubrunner.ids=org.springframework.cloud.contract.verifier.stubs:loanIssuance,org.springframework.cloud.contract.verifier.stubs:fraudDetectionServe
-Dstubrunner.idsToServiceIds.fraudDetectionServer=someNameThatShouldMapFraudDetectionServer (4)

(1) - we tell Stub Runner where all the stubs reside
(2) - we don't want the default behaviour where the discovery service is stubbed. That's why the stub registration will be picked
(3) - we provide a list of stubs to download
(4) - we provide a list of artifactId to serviceId mapping
```

这样，您部署的应用程序就可以通过服务发现向启动的WireMock服务器发送请求。 在`application.yml`最有可能的点1-3可以被默认设置，因为它们不可能改变。 这样，只要您启动Stub Runner引导，您就只能提供要下载的存根的列表。

89.7 Stubs Per Consumer 译：89.7每个消费者的存根

有些情况下，同一端点的两位消费者想要有两个不同的响应。

这种方法还允许您立即知道哪个消费者正在使用您的API的哪一部分。 您可以删除API生成的部分响应，并可以看到哪些自动生成的测试失败。 如果没有失败，那么您可以安全地删除没有人使用它的那部分响应。

让我们看一下下面的示例，为生产者定义的合同`producer`。 有2个消费者：`foo-consumer`和`bar-consumer`。

消费者`foo-service`

```
request {
   url '/foo'
   method GET()
}
response {
    status OK()
    body(
       foo: "foo"
    )
}
```

消费者`bar-service`

```
request {
   url '/foo'
   method GET()
}
response {
    status OK()
    body(
       bar: "bar"
    )
}
```

您无法为相同的请求生成2个不同的响应。 这就是为什么您可以妥善打包合同，然后从`stubsPerConsumer`功能中获利。

在生产者方面，消费者可以拥有一个包含仅与它们相关的合同的文件夹。 通过设定`stubrunner.stubs-per-consumer`标志`true`我们不再注册的所有stub，但只有那些符合消费者applicationâ€™的名称。 换句话说，我们将扫描每个存根的路径，并且如果它包含具有路径中的使用者名称的子文件夹，那么它将被注册。

在`foo`生产者方面，合同看起来像这样

```
.
└── contracts
    ├── bar-consumer
    │   ├── bookReturnedForBar.groovy
    │   └── shouldCallBar.groovy
    └── foo-consumer
        ├── bookReturnedForFoo.groovy
        └── shouldCallFoo.groovy
```

身为`bar-consumer`消费者，你可以设定`spring.application.name`或者`stubrunner.consumer-name`至`bar-consumer`或者设置测试如下：

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest(properties = ["spring.application.name=bar-consumer"])
@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers", repositoryRoot = "classpath:m2repo/
@DirtiesContext
class StubRunnerStubsPerConsumerSpec extends Specification {
...
}
```

然后只 `bar-consumer` 在其名称中包含 `bar-consumer` 的路径下注册的存根（即来自 `src/test/resources/contracts/bar-consumer/some/contracts/…` 文件夹的那些 `src/test/resources/contracts/bar-consumer/some/contracts/…` ）被引用。

或者显式设置消费者名称

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest
@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers", repositoryRoot = "classpath:m2repo/
@DirtiesContext
class StubRunnerStubsPerConsumerWithConsumerNameSpec extends Specification {
...
}
```

然后只 `foo-consumer` 在其名称中包含 `foo-consumer` 的路径下注册的存根（即来自 `src/test/resources/contracts/foo-consumer/some/contracts/…` 文件夹的那些 `src/test/resources/contracts/foo-consumer/some/contracts/…` ）被引用。

您可以查看 issue 224 了解更改背后原因的更多信息。

89.8 Common 译：89.8常见

本节简要介绍常见属性，包括：

- Section 89.8.1, “Common Properties for JUnit and Spring”
- Section 89.8.2, “Stub Runner Stubs IDs”

89.8.1 Common Properties for JUnit and Spring 译：89.8.1 JUnit和Spring的通用属性

您可以使用系统属性或Spring配置属性来设置重复属性。 以下是他们的名字及其默认值：

Property name	Default value	描述
stubrunner.minPort	10000	带存根的已启动WireMock的端口的最小值。
stubrunner.maxPort	15000	带存根的已启动WireMock的端口的最大值。
stubrunner.repositoryRoot		Maven回购网址。 如果空白，请致电当地的Maven回购。
stubrunner.classifier	存根	存根工件的默认分类器。
stubrunner.stubsMode	CLASSPATH	您想要获取并注册存根的方式
stubrunner.ids		下载一组常春藤符号存根。
stubrunner.username		可选用户名，用于访问存储带存根的JAR的工具。
stubrunner.password		用于访问存储带存根的JAR的工具的可选密码。
stubrunner.stubsPerConsumer	假	如果您想为每个消费者使用不同的存根，而不是为每个消费者注册所有存根，请设置为 `true` 。
stubrunner.consumerName		如果您想为每个消费者使用存根，并且想要覆盖消费者名称，只需更改此值即可。

89.8.2 Stub Runner Stubs IDs 译：89.8.2 Stub Runner存根ID

您可以通过 `stubrunner.ids` 系统属性提供存根以供下载。 他们遵循这种模式：

```
groupId:artifactId:version:classifier:port
```

需要注意的是 `version` ， `classifier` 和 `port` 是可选的。

- If you do not provide the `port`, a random one will be picked.
- If you do not provide the `classifier`, the default is used. (Note that you can pass an empty classifier this way: `groupId:artifactId:version:`).
- If you do not provide the `version`, then the `+` will be passed and the latest one is downloaded.

`port` 表示WireMock服务器的端口。

Important

从版本1.0.4开始，您可以提供一系列您希望Stub Runner考虑的版本。 你可以阅读更多关于Aether versioning ranges here 。

89.9 Stub Runner Docker 译：89.9 Stub Runner Docker

我们将发布一个 `spring-cloud/spring-cloud-contract-stub-runner` Docker镜像，它将启动Stub Runner的独立版本。

如果您想了解更多关于Maven的基础知识，工件ID，组ID，分类器和工件管理器，请点击这里 Section 87.6, “Docker Project” 。

89.9.1 How to use it 译：89.9.1如何使用它

只需执行泊坞窗图像。 您可以将任何Section 89.8.1, “Common Properties for JUnit and Spring”作为环境变量。 公约是所有的字母应该是大写。 驼峰符号应该和点（ `.` ）应该通过下划线（ `_` ）分开。 例如 `stubrunner.repositoryRoot` 属性应该表示为 `STUBRUNNER_REPOSITORY_ROOT` 环境变量。

89.9.2 Example of client side usage in a non JVM project 译：89.9.2非JVM项目中客户端使用的示例

WEA€™倒要使用此创建的存根Section 87.6.4, “Server side (nodejs)”一步。 假设我们想要在端口 `9876` 上运行存根。 NodeJS代码在这里可用：

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

让我们使用存根运行Stub Runner Boot应用程序。

```
# Provide the Spring Cloud Contract Docker version
$ SC_CONTRACT_DOCKER_VERSION="..."
# The IP at which the app is running and Docker container can reach it
$ APP_IP="192.168.0.100"
# Spring Cloud Contract Stub Runner properties
$ STUBRUNNER_PORT="8083"
# Stub coordinates 'groupId:artifactId:version:classifier:port'
$ STUBRUNNER_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"
$ STUBRUNNER_REPOSITORY_ROOT="http://${APP_IP}:8081/artifactory/libs-release-local"
# Run the docker with Stub Runner Boot
$ docker run  --rm -e "STUBRUNNER_IDS=${STUBRUNNER_IDS}" -e "STUBRUNNER_REPOSITORY_ROOT=${STUBRUNNER_REPOSITORY_ROOT}" -e "STUBRUNNER_STUBS_MODE=REMO
```

发生的是这样的

- a standalone Stub Runner application got started
- it downloaded the stub with coordinates `com.example:bookstore:0.0.1.RELEASE:stubs` on port `9876`
- it got downloaded from Artifactory running at `http://192.168.0.100:8081/artifactory/libs-release-local`
- after a while Stub Runner will be running on port `8083`
- and the stubs will be running at port `9876`

在服务器端，我们构建了一个有状态的存根。 让我们使用curl来断言存根已正确设置。

```
# let's execute the first request (no response is returned)
$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title", "genre" : "Genre", "description" : "Description", "author" : "Author",
# Now time for the second request
$ curl -X GET http://localhost:9876/api/books
# You will receive contents of the JSON
```

Important

如果要使用本地创建的存根，请在主机上使用环境变量 `-e STUBRUNNER_STUBS_MODE=LOCAL` 并安装当地m2的卷 `-v "${HOME}/.m2/:/root/.m2:ro"`

90. Stub Runner for Messaging 译：90.存根赛跑者的消息传递

Stub Runner可以在内存中运行发布的存根。 它可以与以下框架集成：

- Spring Integration
- Spring Cloud Stream
- Spring AMQP

它还提供了与市场上任何其他解决方案集成的切入点。

Important

如果在类路径中有多个框架，Stub Runner将需要定义应该使用哪一个。 假设您在类路径中同时拥有AMQP，Spring Cloud Stream和Spring Integration。 那么你需要设置 `stubrunner.stream.enabled=false` 和 `stubrunner.integration.enabled=false` 。 这样，剩下的唯一框架就是Spring AMQP。

90.1 Stub triggering 译：90.1存根触发

要触发消息，请使用 `StubTrigger` 接口：

```
package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

public interface StubTrigger {

 /**
  * Triggers an event by a given label for a given {@code groupid:artifactid} notation. You can use only {@code artifactId} too.
  *
  * Feature related to messaging.
  *
  * @return true - if managed to run a trigger
  */
 boolean trigger(String ivyNotation, String labelName);

 /**
  * Triggers an event by a given label.
  *
  * Feature related to messaging.
  *
  * @return true - if managed to run a trigger
  */
 boolean trigger(String labelName);

 /**
  * Triggers all possible events.
  *
  * Feature related to messaging.
  *
  * @return true - if managed to run a trigger
  */
 boolean trigger();

 /**
  * Returns a mapping of ivy notation of a dependency to all the labels it has.
  *
  * Feature related to messaging.
  */
 Map<String, Collection<String>> labels();
}
```

为了方便起见， `StubFinder`接口扩展了 `StubTrigger` ，所以在测试中您只需要一个或另一个。

`StubTrigger`为您提供以下选项来触发消息：

- Section 90.1.1, “Trigger by Label”
- Section 90.1.2, “Trigger by Group and Artifact Ids”
- Section 90.1.3, “Trigger by Artifact Ids”
- Section 90.1.4, “Trigger All Messages”

90.1.1 Trigger by Label 译：90.1.1通过标签触发

```
stubFinder.trigger('return_book_1')
```

90.1.2 Trigger by Group and Artifact Ids 译：90.1.2按组和工件ID进行触发

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService', 'return_book_1')
```

90.1.3 Trigger by Artifact Ids 译：90.1.3由工件标识触发

```
stubFinder.trigger('streamService', 'return_book_1')
```

90.1.4 Trigger All Messages 译：90.1.4触发所有消息

```
stubFinder.trigger()
```

90.2 Stub Runner Integration 译：90.2 Stub Runner集成

Spring Cloud Contract Verifier Stub Runner的消息传递模块为您提供了一种与Spring Integration集成的简单方法。 对于提供的工件，它自动下载存根并注册所需的路线。

90.2.1 Adding the Runner to the Project 译：90.2.1将 Runner添加到项目中

您可以在类路径中同时拥有Spring Integration和Spring Cloud Contract Stub Runner。 请记住用`@AutoConfigureStubRunner`注释您的测试课程。

90.2.2 Disabling the functionality 译：90.2.2禁用功能

如果您需要禁用此功能，请设置 `stubrunner.integration.enabled=false`属性。

假设您有 `integrationService`应用程序的部署存根的以下Maven存储库：

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── integrationService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── integrationService-0.0.1-SNAPSHOT.pom
                            │   ├── integrationService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

进一步假设存根包含以下结构：

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

考虑以下合同（编号 **1** ）：

```
Contract.make {
 label 'return_book_1'
 input {
  triggeredBy('bookReturnedTriggered()')
 }
 outputMessage {
  sentTo('output')
  body('''{ "bookName" : "foo" }''')
  headers {
   header('BOOK-NAME', 'foo')
  }
 }
}
```

现在考虑 **2** ：

```
Contract.make {
 label 'return_book_2'
 input {
  messageFrom('input')
  messageBody([
    bookName: 'foo'
  ])
  messageHeaders {
   header('sample', 'header')
  }
 }
 outputMessage {
  sentTo('output')
  body([
    bookName: 'foo'
  ])
  headers {
   header('BOOK-NAME', 'foo')
  }
 }
}
```

和下面的Spring Integration路线：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/integration
   http://www.springframework.org/schema/integration/spring-integration.xsd">


 <!-- REQUIRED FOR TESTING -->
 <bridge input-channel="output"
   output-channel="outputTest"/>

 <channel id="outputTest">
  <queue/>
 </channel>

</beans:beans>
```

这些例子适用于三种情况：

- the section called “Scenario 1 (no input message)”
- ???
- the section called “Scenario 3 (input with no output)”

Scenario 1 (no input message)译:情景1（无输入消息）

要通过 `return_book_1` 标签触发邮件，请使用 `StubTigger` 界面，如下所示：

```
stubFinder.trigger('return_book_1')
```

收听发送至 `output` 的消息的输出：

```
Message<?> receivedMessage = messaging.receive('outputTest')
```

收到的消息将通过以下声明：

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (output triggered by input) 译:情景2（输入触发输出）

由于路线是为您设置的，因此您可以将消息发送至 `output` 目的地：

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'input')
```

收听发送至 `output` 的消息的输出：

```
Message<?> receivedMessage = messaging.receive('outputTest')
```

收到的消息传递以下声明：

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (input with no output) 译:情景3（输入无输出）

由于路由是为您设置的，因此您可以将消息发送到 `input` 目标：

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```

90.3 Stub Runner Stream 译：90.3 Stub Runner Stream

Spring Cloud Contract Verifier Stub Runner的消息模块为您提供了一种与Spring Stream集成的简单方法。 对于提供的工件，它自动下载存根并注册所需的路线。

如果Stub Runner与Stream `messageFrom` 或 `sentTo` 字符串的集成首先作为通道的 `destination` 解析并且不存在此类 `destination` ，则目标将解析为通道名称。

Important

如果您想使用Spring Cloud Stream，请在 `org.springframework.cloud:spring-cloud-stream-test-support` 上添加依赖关系。

Maven的。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

摇篮。

```
testCompile "org.springframework.cloud:spring-cloud-stream-test-support"
```

90.3.1 Adding the Runner to the Project 译：90.3.1将 Runner添加到项目中

您可以在类路径上同时拥有Spring Cloud Stream和Spring Cloud Contract Stub Runner。 请记住用 `@AutoConfigureStubRunner` 注释您的测试课程。

90.3.2 Disabling the functionality 译：90.3.2禁用功能

如果您需要禁用此功能，请设置 `stubrunner.stream.enabled=false` 属性。

假设您拥有以下Maven存储库，并为 `streamService` 应用程序部署了存根：

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── streamService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── streamService-0.0.1-SNAPSHOT.pom
                            │   ├── streamService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

进一步假设存根包含以下结构：

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

考虑以下合同（编号 **1** ）：

```
Contract.make {
 label 'return_book_1'
 input { triggeredBy('bookReturnedTriggered()') }
 outputMessage {
  sentTo('returnBook')
  body('''{ "bookName" : "foo" }''')
  headers { header('BOOK-NAME', 'foo') }
 }
}
```

现在考虑 **2** ：

```
Contract.make {
 label 'return_book_2'
 input {
  messageFrom('bookStorage')
  messageBody([
   bookName: 'foo'
  ])
  messageHeaders { header('sample', 'header') }
 }
 outputMessage {
  sentTo('returnBook')
  body([
   bookName: 'foo'
  ])
  headers { header('BOOK-NAME', 'foo') }
 }
}
```

现在考虑下面的Spring配置：

```
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids: org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-SNAPSHOT:stubs
stubrunner.stubs-mode: remote
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: returnBook
        input:
          destination: bookStorage

server:
  port: 0

debug: true
```

这些例子适用于三种情况：

- the section called “Scenario 1 (no input message)”
- the section called “Scenario 2 (output triggered by input)”
- the section called “Scenario 3 (input with no output)”

Scenario 1 (no input message)译:情景1（无输入消息）

要通过 `return_book_1` 标签触发消息，请按以下方式使用 `StubTrigger` 界面：

```
stubFinder.trigger('return_book_1')
```

要收听发送到 `destination` 为 `returnBook` 为 `returnBook` 的频道的消息的输出：

```
Message<?> receivedMessage = messaging.receive('returnBook')
```

收到的消息传递以下声明：

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (output triggered by input)译:情景2（输入触发输出）

由于路线是为您设置的，因此您可以发送消息给 `bookStorage` `destination` ：

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'bookStorage')
```

收听发送至 `returnBook` 的消息的输出：

```
Message<?> receivedMessage = messaging.receive('returnBook')
```

收到的消息传递以下声明：

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (input with no output) 译:情景3（输入无输出）

由于路线是为您设置的，因此您可以将消息发送到 `output` 目的地：

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```

90.4 Stub Runner Spring AMQP 译：90.4 Stub Runner Spring AMQP

Spring Cloud Contract Verifier Stub Runner的消息传递模块提供了一种与Spring AMQP的Rabbit模板集成的简单方法。 对于提供的工件，它自动下载存根并注册所需的路线。

集成尝试独立工作（即，不与正在运行的RabbitMQ消息代理进行交互）。 它期望在应用程序上下文中使用 `RabbitTemplate` ，并将其用作名为 `@SpyBean` 的弹簧引导测试。 因此，它可以使用mockito间谍功能来验证和检查应用程序发送的消息。

在消息使用者方面，存根运行程序会考虑应用程序上下文中所有 `@RabbitListener` 注释的端点和所有 `SimpleMessageListenerContainer` 对象。

由于消息通常发送给AMQP中的交易所，消息合约包含交易所名称作为目的地。 另一端的消息侦听器绑定到队列。 绑定将交换机连接到队列。 如果消息协定被触发，Spring AMQP stub runner集成将查找与此交换匹配的应用程序上下文的绑定。 然后它收集来自Spring交换机的队列，并尝试查找绑定到这些队列的消息监听器。 所有匹配的消息侦听器都会触发该消息。

90.4.1 Adding the Runner to the Project 译：90.4.1将 Runner添加到项目中

您可以在类路径上同时包含Spring AMQP和Spring Cloud Contract Stub Runner，并设置属性 `stubrunner.amqp.enabled=true` 。 请记住用 `@AutoConfigureStubRunner` 注释您的测试课程。

Important

如果在类路径中已经有Stream和Integration，则需要通过设置 `stubrunner.stream.enabled=false` 和 `stubrunner.integration.enabled=false` 属性来明确禁用它们。

假设您拥有以下Maven存储库，并为 `spring-cloud-contract-amqp-test` 应用程序部署了存根。

```
└── .m2
    └── repository
        └── com
            └── example
                └── spring-cloud-contract-amqp-test
                    ├── 0.4.0-SNAPSHOT
                    │   ├── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT.pom
                    │   ├── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT-stubs.jar
                    │   └── maven-metadata-local.xml
                    └── maven-metadata-local.xml
```

进一步假设存根包含以下结构：

```
├── META-INF
│   └── MANIFEST.MF
└── contracts
    └── shouldProduceValidPersonData.groovy
```

考虑以下合同：

```
Contract.make {
    // Human readable description
    description 'Should produce valid person data'
    // Label by means of which the output message can be triggered
    label 'contract-test.person.created.event'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('createPerson()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo 'contract-test.exchange'
        headers {
            header('contentType': 'application/json')
            header('__TypeId__': 'org.springframework.cloud.contract.stubrunner.messaging.amqp.Person')
        }
        // the body of the output message
        body ([
                id: $(consumer(9), producer(regex("[0-9]+"))),
                name: "me"
        ])
    }
}
```

现在考虑下面的Spring配置：

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids: org.springframework.cloud.contract.verifier.stubs.amqp:spring-cloud-contract-amqp-test:0.4.0-SNAPSHOT:stubs
  stubs-mode: remote
  amqp:
    enabled: true
server:
  port: 0
```

Triggering the message 译:触发消息

要使用上述合同触发邮件，请按以下方式使用 `StubTrigger` 界面：

```
stubTrigger.trigger("contract-test.person.created.event")
```

该消息的目标地址为 `contract-test.exchange` ，因此Spring AMQP stub runner集成查找与此交换相关的绑定。

```
@Bean
public Binding binding() {
 return BindingBuilder.bind(new Queue("test.queue")).to(new DirectExchange("contract-test.exchange")).with("#");
}
```

绑定定义绑定队列 `test.queue` 。 结果，以下侦听器定义与合同消息匹配并被调用。

```
@Bean
public SimpleMessageListenerContainer simpleMessageListenerContainer(ConnectionFactory connectionFactory,
                  MessageListenerAdapter listenerAdapter) {
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
 container.setConnectionFactory(connectionFactory);
 container.setQueueNames("test.queue");
 container.setMessageListener(listenerAdapter);

 return container;
}
```

另外，以下带注释的侦听器匹配并被调用：

```
@RabbitListener(bindings = @QueueBinding( value = @Queue(value = "test.queue"), exchange = @Exchange(value = "contract-test.exchange", ignoreDeclarat
public void handlePerson(Person person) {
 this.person = person;
}
```

该消息被直接交给 `onMessage` 所述的方法 `MessageListener` 与匹配相关联 `SimpleMessageListenerContainer` 。

Spring AMQP Test Configuration 译：Spring AMQP测试配置

为了避免Spring AMQP在我们的测试中尝试连接到正在运行的代理，请配置一个模拟 `ConnectionFactory` 。

要禁用 `stubrunner.amqp.mockConnection=false` ConnectionFactory，请设置以下属性： `stubrunner.amqp.mockConnection=false`

```
stubrunner:
  amqp:
    mockConnection: false
```

91. Contract DSL 译：91.合同DSL

Spring Cloud Contract支持开箱即用的两种类型的DSL。 一本是 `Groovy` ，一本是 `YAML` 。

如果您决定在Groovy中编写合同，如果您以前没有使用过Groovy，请不要惊慌。 因为DSL只使用它的一小部分（只有文字，方法调用和关闭），所以不需要对语言的了解。 而且，DSL是静态类型的，为了让程序员可读，而不需要任何DSL本身的知识。

Important

请记住，在Groovy合同文件中，必须为 `Contract` 类和 `make` 静态导入提供完全限定名称，例如 `org.springframework.cloud.spec.Contract.make { … }` 。 您还可以提供 `Contract` 类的导入： `import org.springframework.cloud.spec.Contract` ，然后致电 `Contract.make { … }` 。

Spring Cloud Contract支持在单个文件中定义多个合约。

以下是Groovy合同定义的完整示例：

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'PUT'
  url '/api/12'
  headers {
   header 'Content-Type': 'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-analyzer.v1+json'
  }
  body '''\
  [{
   "created_at": "Sat Jul 26 09:38:57 +0000 2014",
   "id": 492967299297845248,
   "id_str": "492967299297845248",
   "text": "Gonna see you at Warsaw",
   "place":
   {
    "attributes":{},
    "bounding_box":
    {
     "coordinates":
      [[
       [-77.119759,38.791645],
       [-76.909393,38.791645],
       [-76.909393,38.995548],
       [-77.119759,38.995548]
      ]],
     "type":"Polygon"
    },
    "country":"United States",
    "country_code":"US",
    "full_name":"Washington, DC",
    "id":"01fbe706f872cb32",
    "name":"Washington",
    "place_type":"city",
    "url": "http://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
   }
  }]
 '''
 }
 response {
  status OK()
 }
}
```

以下是YAML合同定义的完整示例：

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

您可以使用独立maven命令将合约编译为存根映射： `mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert`

91.1 Limitations译：91.1限制

Spring Cloud Contract Verifier不能正确支持XML。 请使用JSON或帮助我们实现此功能。

验证JSON数组大小的支持是实验性的。 如果要打开它，请将以下系统属性的值设置为`true` ： `spring.cloud.contract.verifier.assert.size` 。 默认情况下，此功能设置为`false` 。 您还可以在插件配置中提供`assertJsonSize`属性。

因为JSON结构可以有任何形式，所以在`GString`使用Groovy DSL和`value(consumer(… ), producer(… ))`表示法时可能无法正确解析它。 这就是为什么你应该使用Groovy Map符号。

91.2 Common Top-Level elements译：91.2通用顶层元素

以下各节介绍最常见的顶级元素：

91.2.1 Description译：91.2.1说明

您可以将`description`添加到您的合同中。 描述是任意文本。 以下代码显示了一个示例：

Groovy DSL。

```
  org.springframework.cloud.contract.spec.Contract.make {
   description('''
given:
 An input
when:
 Sth happens
then:
 Output
''')
  }
```

YAML。

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

91.2.2 Name译：91.2.2名称

您可以为合同提供一个名称。 假定您提供了以下名称： `should register a user` 。 如果你这样做，自动生成测试的名称是`validate_should_register_a_user` 。 此外，WireMock存根中存根的名称是`should_register_a_user.json` 。

Important

您必须确保该名称不包含任何使生成的测试不能编译的字符。 另外，请记住，如果您为多个合同提供相同的名称，则自动生成的测试无法编译，并且生成的存根会互相覆盖。

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 name("some_special_name")
}
```

YAML。

```
name: some name
```

91.2.3 Ignoring Contracts译：91.2.3忽略合同

如果您想忽略合同，则可以在插件配置中设置忽略合同的值，也可以在合同本身上设置 `ignored` 属性：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 ignored()
}
```

YAML。

```
ignored: true
```

91.2.4 Passing Values from Files译：91.2.4从文件传递值

从版本`1.2.0`开始，您可以传递文件中的值。 假设您在我们的项目中拥有以下资源。

```
└── src
    └── test
        └── resources
            └── contracts
                ├── readFromFile.groovy
                ├── request.json
                └── response.json
```

进一步假设你的合同如下：

Groovy DSL。

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
 request {
  method('PUT')
  headers {
   contentType(applicationJson())
  }
  body(file("request.json"))
  url("/1")
 }
 response {
  status OK()
  body(file("response.json"))
  headers {
   contentType(textPlain())
  }
 }
}
```

YAML。

```
request:
  method: GET
  url: /foo
  bodyFromFile: request.json
response:
  status: 200
  bodyFromFile: response.json
```

进一步假设JSON文件如下所示：

request.json

```
{ "status" : "REQUEST" }
```

response.json

```
{ "status" : "RESPONSE" }
```

当测试或存根生成发生时，文件的内容被传递给请求或响应的主体。 文件的名称需要是一个文件，其位置与合约所在的文件夹相关。

91.2.5 HTTP Top-Level Elements 译：91.2.5 HTTP顶级元素

以下方法可以在合同定义的顶层关闭中调用。 `request`和`response`是强制性的。 `priority`是可选的。

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 // Definition of HTTP request part of the contract
 // (this can be a valid request or invalid depending
 // on type of contract being specified).
 request {
  //...
 }

 // Definition of HTTP response part of the contract
 // (a service implementing this contract should respond
 // with following response after receiving request
 // specified in "request" part above).
 response {
  //...
 }

 // Contract priority, which can be used for overriding
 // contracts (1 is highest). Priority is optional.
 priority 1
}
```

YAML。

```
priority: 8
request:
...
response:
...
```

Important

如果您想让您的合同具有**较高**的优先级，您需要将**较低的**编号传递给`priority`标签/方法。 例如， `priority`与价值`5`比**更高的**优先级`priority`与价值`10` 。

91.3 Request 译：91.3请求

HTTP协议只需要在请求中指定**方法和url** 。 在合同的请求定义中，同样的信息是强制性的。

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  // HTTP request method (GET/POST/PUT/DELETE).
  method 'GET'

  // Path component of request URL is specified as follows.
  urlPath('/users')
 }

 response {
  //...
 }
}
```

YAML。

```
method: PUT
url: /foo
```

可以指定绝对值而不是相对值`url` ，但推荐使用`urlPath` ，因为这样做会使测试 **独立**于 **主机** 。

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'GET'

  // Specifying `url` and `urlPath` in one contract is illegal.
  url('http://localhost:8888/users')
 }

 response {
  //...
 }
}
```

YAML。

```
request:
  method: PUT
  urlPath: /foo
```

`request`可能包含 **查询参数** 。

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  //...

  urlPath('/users') {

   // Each parameter is specified in form
   // `'paramName' : paramValue` where parameter value
   // may be a simple literal or one of matcher functions,
   // all of which are used in this example.
   queryParameters {

    // If a simple literal is used as value
    // default matcher function is used (equalTo)
    parameter 'limit': 100

    // `equalTo` function simply compares passed value
    // using identity operator (==).
    parameter 'filter': equalTo("email")

    // `containing` function matches strings
    // that contains passed substring.
    parameter 'gender': value(consumer(containing("[mf]")), producer('mf'))

    // `matching` function tests parameter
    // against passed regular expression.
    parameter 'offset': value(consumer(matching("[0-9]+")), producer(123))

    // `notMatching` functions tests if parameter
    // does not match passed regular expression.
    parameter 'loginStartsWith': value(consumer(notMatching(".{0,2}")), producer(3))
   }
  }

  //...
 }

 response {
  //...
 }
}
```

YAML。

```
request:
...
  queryParameters:
    a: b
    b: c
  headers:
    foo: bar
    fooReq: baz
  cookies:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
    cookies:
      - key: foo2
        regex: bar
      - key: foo3
        predefined:
```

`request`可能包含额外的 **请求标头** ， 如以下示例所示：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  //...

  // Each header is added in form `'Header-Name' : 'Header-Value'`.
  // there are also some helper methods
  headers {
   header 'key': 'value'
   contentType(applicationJson())
  }

  //...
 }

 response {
  //...
 }
}
```

YAML。

```
request:
...
headers:
  foo: bar
  fooReq: baz
```

`request`可能包含其他 **请求Cookie** ，如以下示例所示：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  //...

  // Each Cookies is added in form `'Cookie-Key' : 'Cookie-Value'`.
  // there are also some helper methods
  cookies {
   cookie 'key': 'value'
   cookie('another_key', 'another_value')
  }

  //...
 }

 response {
  //...
 }
}
```

YAML。

```
request:
...
cookies:
  foo: bar
  fooReq: baz
```

`request`可能包含 **请求主体** ：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  //...

  // Currently only JSON format of request body is supported.
  // Format will be determined from a header or body's content.
  body '''{ "login" : "john", "name": "John The Contract" }'''
 }

 response {
  //...
 }
}
```

YAML。

```
request:
...
body:
  foo: bar
```

`request`可能包含**多部分**元素。 要包含多部分元素，请使用`multipart`方法/部分，如以下示例所示

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract contractDsl = org.springframework.cloud.contract.spec.Contract.make {
 request {
  method "PUT"
  url "/multipart"
  headers {
   contentType('multipart/form-data;boundary=AaB03x')
  }
  multipart(
    // key (parameter name), value (parameter value) pair
    formParameter: $(c(regex('".+"')), p('"formParameterValue"')),
    someBooleanParameter: $(c(regex(anyBoolean())), p('true')),
    // a named parameter (e.g. with `file` name) that represents file with
    // `name` and `content`. You can also call `named("fileName", "fileContent")`
    file: named(
      // name of the file
      name: $(c(regex(nonEmpty())), p('filename.csv')),
      // content of the file
      content: $(c(regex(nonEmpty())), p('file content')),
      // content type for the part
      contentType: $(c(regex(nonEmpty())), p('application/json')))
  )
 }
 response {
  status OK()
 }
}
org.springframework.cloud.contract.spec.Contract contractDsl = org.springframework.cloud.contract.spec.Contract.make {
 request {
  method "PUT"
  url "/multipart"
  headers {
   contentType('multipart/form-data;boundary=AaB03x')
  }
  multipart(
    file: named(
      name: value(stub(regex('.+')), test('file')),
      content: value(stub(regex('.+')), test([100, 117, 100, 97] as byte[]))
    )
  )
 }
 response {
  status 200
 }
}
```

YAML。

```
request:
  method: PUT
  url: /multipart
  headers:
    Content-Type: multipart/form-data;boundary=AaB03x
  multipart:
    params:
    # key (parameter name), value (parameter value) pair
      formParameter: '"formParameterValue"'
      someBooleanParameter: true
    named:
      - paramName: file
        fileName: filename.csv
        fileContent: file content
  matchers:
    multipart:
      params:
        - key: formParameter
          regex: ".+"
        - key: someBooleanParameter
          predefined: any_boolean
      named:
        - paramName: file
          fileName:
            predefined: non_empty
          fileContent:
            predefined: non_empty
response:
  status: 200
```

在前面的例子中，我们用两种方法中的任何一种来定义参数：

Groovy DSL

- Directly, by using the map notation, where the value can be a dynamic property (such as `formParameter: $(consumer(… ), producer(… ))`).
- By using the `named(… )` method that lets you set a named parameter. A named parameter can set a `name` and `content`. You can call it either via a method with two arguments, such as `named("fileName", "fileContent")`, or via a map notation, such as `named(name: "fileName", content: "fileContent")`.

YAML

- The multipart parameters are set via `multipart.params` section
- The named parameters (the `fileName` and `fileContent` for a given parameter name) can be set via the `multipart.named` section. That section contains the `paramName` (name of the parameter), `fileName` (name of the file), `fileContent` (content of the file) fields
- 动态位可以通过 `matchers.multipart` 部分进行设置
  - for parameters use the `params` section that can accept `regex` or a `predefined` regular expression
  - for named params use the `named` section where first you define the parameter name via `paramName` and then you can pass the parametrization of either `fileName` or `fileContent` via `regex` or a `predefined` regular expression

从这份合同中，生成的测试如下：

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Content-Type", "multipart/form-data;boundary=AaB03x")
   .param("formParameter", "\"formParameterValue\"")
   .param("someBooleanParameter", "true")
   .multiPart("file", "filename.csv", "file content".getBytes());

// when:
 ResponseOptions response = given().spec(request)
   .put("/multipart");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
```

WireMock存根如下所示：

```
   '''
{
  "request" : {
 "url" : "/multipart",
 "method" : "PUT",
 "headers" : {
   "Content-Type" : {
  "matches" : "multipart/form-data;boundary=AaB03x.*"
   }
 },
 "bodyPatterns" : [ {
  "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"formParameter\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Transfer-Encoding: .*\\r
    }, {
       "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"someBooleanParameter\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Transfer-Enc
    }, {
   "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"file\\"; filename=\\"[\\\\S\\\\s]+\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Tr
 } ]
  },
  "response" : {
 "status" : 200,
 "transformers" : [ "response-template", "foo-transformer" ]
  }
}
 '''
```

91.4 Response译：91.4响应

响应必须包含一个**HTTP状态码**并可能包含其他信息。 以下代码显示了一个示例：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  //...
 }
 response {
  // Status code sent by the server
  // in response to request specified above.
  status OK()
 }
}
```

YAML。

```
response:
...
status: 200
```

除了状态之外，响应可能包含 **标头** ， **cookie**和 **主体** ，这两者的指定方式与请求中的相同（请参见上一段）。

通过Groovy DSL，您可以引用`org.springframework.cloud.contract.spec.internal.HttpStatus`方法来提供有意义的状态，而不是数字。 例如，对于`OK()`，您可以拨打`OK()`或`BAD_REQUEST()` `200`或`400` 。

91.5 Dynamic properties译：91.5动态属性

该合约可以包含一些动态属性：时间戳，ID等等。 您不希望强制消费者将其时钟存根以始终返回相同的时间值，以便与存根匹配。

对于Groovy DSL，您可以通过两种方式在合同中提供动态部分：直接在主体中传递它们，或者将它们设置在名为`bodyMatchers`的单独部分中。

在2.0.0之前，这些设置是使用`testMatchers`和`stubMatchers`设置的，请参阅 migration guide以获取更多信息。

对于YAML，您只能使用`matchers`部分。

91.5.1 Dynamic properties inside the body译：91.5.1体内动态属性

Important

本节仅适用于Groovy DSL。 查看Section 91.5.7, “Dynamic Properties in the Matchers Sections”部分了解相似功能的YAML示例。

您可以使用`value`方法或使用Groovy映射表示法与`$()`来设置主体内的属性。 以下示例显示如何使用value方法设置动态属性：

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```

以下示例显示如何使用 `$()` 设置动态属性：

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
$(client(...), server(...))
```

两种方法的效果都很好。 `stub` 个 `client` 方法是在别名 `consumer` 方法。 后续部分将仔细研究您可以对这些值做些什么。

91.5.2 Regular expressions 译：91.5.2正则表达式

Important

本节仅适用于Groovy DSL。 查看Section 91.5.7, “Dynamic Properties in the Matchers Sections”部分，了解类似功能的YAML示例。

您可以使用正则表达式在Contract DSL中编写您的请求。 这样做特别有用，当您要指示给定的响应应该为遵循给定模式的请求提供时。 另外，当您需要使用模式而不是准确的值同时用于测试和服务器端测试时，您可以使用正则表达式。

以下示例显示如何使用正则表达式来编写请求：

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method('GET')
  url $(consumer(~/\/[0-9]{2}/), producer('/12'))
 }
 response {
  status OK()
  body(
    id: $(anyNumber()),
    surname: $(
      consumer('Kowalsky'),
      producer(regex('[a-zA-Z]+'))
    ),
    name: 'Jan',
    created: $(consumer('2014-02-02 12:23:43'), producer(execute('currentDate(it)'))),
    correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-72d2220dd827'),
      producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))
    )
  )
  headers {
   header 'Content-Type': 'text/plain'
  }
 }
}
```

您也可以仅使用正则表达式提供通信的一方。 如果这样做，那么契约引擎会自动提供与提供的正则表达式匹配的生成的字符串。 以下代码显示了一个示例：

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'PUT'
  url value(consumer(regex('/foo/[0-9]{5}')))
  body([
   requestElement: $(consumer(regex('[0-9]{5}')))
  ])
  headers {
   header('header', $(consumer(regex('application\\/vnd\\.fraud\\.v1\\+json;.*'))))
  }
 }
 response {
  status OK()
  body([
   responseElement: $(producer(regex('[0-9]{7}')))
  ])
  headers {
   contentType("application/vnd.fraud.v1+json")
  }
 }
}
```

在前面的例子中，通信的另一面有为请求和响应生成的相应数据。

Spring Cloud Contract附带一系列可用于合同中的预定义正则表达式，如以下示例所示：

```
protected static final Pattern TRUE_OR_FALSE = Pattern.compile(/(true|false)/)
protected static final Pattern ALPHA_NUMERIC = Pattern.compile('[a-zA-Z0-9]+')
protected static final Pattern ONLY_ALPHA_UNICODE = Pattern.compile(/[\p{L}]*/)
protected static final Pattern NUMBER = Pattern.compile('-?(\\d*\\.\\d+|\\d+)')
protected static final Pattern INTEGER = Pattern.compile('-?(\\d+)')
protected static final Pattern POSITIVE_INT = Pattern.compile('([1-9]\\d*)')
protected static final Pattern DOUBLE = Pattern.compile('-?(\\d*\\.\\d+)')
protected static final Pattern HEX = Pattern.compile('[a-fA-F0-9]+')
protected static final Pattern IP_ADDRESS = Pattern.compile('([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-
protected static final Pattern HOSTNAME_PATTERN = Pattern.compile('((http[s]?|ftp):/)/?([^:/\\s]+)(:[0-9]{1,5})?')
protected static final Pattern EMAIL = Pattern.compile('[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,6}')
protected static final Pattern URL = UrlHelper.URL
protected static final Pattern HTTPS_URL = UrlHelper.HTTPS_URL
protected static final Pattern UUID = Pattern.compile('[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}')
protected static final Pattern ANY_DATE = Pattern.compile('(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])')
protected static final Pattern ANY_DATE_TIME = Pattern.compile('([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):(
protected static final Pattern ANY_TIME = Pattern.compile('(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')
```

```
protected static final Pattern NON_EMPTY = Pattern.compile(/[\S\s]+/)
protected static final Pattern NON_BLANK = Pattern.compile(/^\s*\S[\S\s]*/)
protected static final Pattern ISO8601_WITH_OFFSET = Pattern.compile(/([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0

protected static Pattern anyOf(String... values){
 return Pattern.compile(values.collect({"^$it\$"}).join("|"))
}

Pattern onlyAlphaUnicode() {
 return ONLY_ALPHA_UNICODE
}

Pattern alphaNumeric() {
 return ALPHA_NUMERIC
}

Pattern number() {
 return NUMBER
}

Pattern positiveInt() {
 return POSITIVE_INT
}

Pattern anyBoolean() {
 return TRUE_OR_FALSE
}

Pattern anInteger() {
 return INTEGER
}

Pattern aDouble() {
 return DOUBLE
}

Pattern ipAddress() {
 return IP_ADDRESS
}

Pattern hostname() {
 return HOSTNAME_PATTERN
}

Pattern email() {
 return EMAIL
}

Pattern url() {
 return URL
}

Pattern httpsUrl() {
 return HTTPS_URL
}

Pattern uuid(){
 return UUID
}

Pattern isoDate() {
 return ANY_DATE
}

Pattern isoDateTime() {
 return ANY_DATE_TIME
}

Pattern isoTime() {
 return ANY_TIME
}

Pattern iso8601WithOffset() {
 return ISO8601_WITH_OFFSET
}

Pattern nonEmpty() {
 return NON_EMPTY
}

Pattern nonBlank() {
 return NON_BLANK
}
```

在您的合同中，您可以使用它，如以下示例所示：

```
Contract dslWithOptionalsInString = Contract.make {
 priority 1
 request {
  method POST()
  url '/users/password'
  headers {
   contentType(applicationJson())
  }
  body(
    email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
    callback_url: $(consumer(regex(hostname())), producer('http://partners.com'))
  )
 }
 response {
  status 404
  headers {
   contentType(applicationJson())
  }
  body(
    code: value(consumer("123123"), producer(optional("123123"))),
    message: "User not found by email = [${value(producer(regex(email())), consumer('not.existing@user.com'))}]"
  )
 }
}
```

91.5.3 Passing Optional Parameters 译：91.5.3传递可选参数

Important

本节仅适用于Groovy DSL。 查看Section 91.5.7, “Dynamic Properties in the Matchers Sections”部分，了解类似功能的YAML示例。

可以在合同中提供可选参数。 但是，只能为以下内容提供可选参数：

- *STUB* side of the Request
- *TEST* side of the Response

以下示例显示如何提供可选参数：

```
org.springframework.cloud.contract.spec.Contract.make {
 priority 1
 request {
  method 'POST'
  url '/users/password'
  headers {
   contentType(applicationJson())
  }
  body(
    email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
    callback_url: $(consumer(regex(hostname())), producer('http://partners.com'))
  )
 }
 response {
  status 404
  headers {
   header 'Content-Type': 'application/json'
  }
  body(
    code: value(consumer("123123"), producer(optional("123123")))
  )
 }
}
```

通过用 `optional()` 方法包装身体的一部分，可以创建一个正则表达式，该表达式必须存在0次或更多次。

如果你使用Spock for，下面的测试将从前面的例子中产生：

```
"""
 given:
  def request = given()
    .header("Content-Type", "application/json")
    .body('''{"email":"abc@abc.com","callback_url":"http://partners.com"}''')

 when:
  def response = given().spec(request)
    .post("/users/password")

 then:
  response.statusCode == 404
  response.header('Content-Type')  == 'application/json'
 and:
  DocumentContext parsedJson = JsonPath.parse(response.body.asString())
  assertThatJson(parsedJson).field("['code']").matches("(123123)?")
"""
```

还会生成以下存根：

```
'''
{
  "request" : {
 "url" : "/users/password",
 "method" : "POST",
 "bodyPatterns" : [ {
   "matchesJsonPath" : "$[?(@.['email'] =~ /([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,6})?/)]"
 }, {
   "matchesJsonPath" : "$[?(@.['callback_url'] =~ /((http[s]?|ftp):\\\\/)\\\\/?([^:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"
 } ],
 "headers" : {
   "Content-Type" : {
  "equalTo" : "application/json"
   }
 }
  },
  "response" : {
 "status" : 404,
 "body" : "{\\"code\\":\\"123123\\",\\"message\\":\\"User not found by email == [not.existing@user.com]\\"}",
 "headers" : {
   "Content-Type" : "application/json"
 }
  },
  "priority" : 1
}
'''
```

91.5.4 Executing Custom Methods on the Server Side译：91.5.4在服务器端执行自定义方法

Important

本节仅适用于Groovy DSL。 查看Section 91.5.7, “Dynamic Properties in the Matchers Sections”部分了解相似功能的YAML示例。

您可以定义在测试期间在服务器端执行的方法调用。 这种方法可以添加到配置中定义为“baseClassForTests”的类中。 以下代码显示了测试用例的合同部分示例：

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'PUT'
  url $(consumer(regex('^/api/[0-9]{2}$')), producer('/api/12'))
  headers {
   header 'Content-Type': 'application/json'
  }
  body '''\
    [{
     "text": "Gonna see you at Warsaw"
    }]
   '''
 }
 response {
  body (
    path: $(consumer('/api/12'), producer(regex('^/api/[0-9]{2}$'))),
    correlationId: $(consumer('1223456'), producer(execute('isProperCorrelationId($it)')))
  )
  status OK()
 }
}
```

以下代码显示了测试用例的基类部分：

```
abstract class BaseMockMvcSpec extends Specification {

 def setup() {
  RestAssuredMockMvc.standaloneSetup(new PairIdController())
 }

 void isProperCorrelationId(Integer correlationId) {
  assert correlationId == 123456
 }

 void isEmpty(String value) {
  assert value == null
 }

}
```

Important

您不能同时使用字符串和`execute`来执行串联。 例如，致电`header('Authorization', 'Bearer ' + execute('authToken()'))`会导致不正确的结果。 相反，请致电`header('Authorization', execute('authToken()'))`并确保`authToken()`方法返回您需要的一切。

根据JSON路径，从JSON中读取的对象的类型可以是下列之一：

- `String`: If you point to a `String` value in the JSON.
- `JSONArray`: If you point to a `List` in the JSON.
- `Map`: If you point to a `Map` in the JSON.
- `Number`: If you point to `Integer`, `Double` etc. in the JSON.
- `Boolean`: If you point to a `Boolean` in the JSON.

在合同的请求部分中，您可以指定`body`应该来自某个方法。

Important

您必须同时提供消费者和生产者方面。 `execute`部分适用于全身 - 不适用于部分。

以下示例显示如何从JSON中读取对象：

```
Contract contractDsl = Contract.make {
 request {
  method 'GET'
  url '/something'
  body(
    $(c("foo"), p(execute("hashCode()")))
  )
 }
 response {
  status OK()
 }
}
```

前面的示例导致在请求正文中调用`hashCode()`方法。 它应该类似于下面的代码：

```
// given:
 MockMvcRequestSpecification request = given()
   .body(hashCode());

// when:
 ResponseOptions response = given().spec(request)
   .get("/something");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
```

91.5.5 Referencing the Request from the Response译：91.5.5引用来自响应的请求

最好的情况是提供固定值，但有时您需要在回复中引用请求。

如果您使用Groovy DSL编写合同，则可以使用`fromRequest()`方法，该方法可以引用HTTP请求中的一组元素。 您可以使用以下选项：

- `fromRequest().url()`: Returns the request URL and query parameters.
- `fromRequest().query(String key)`: Returns the first query parameter with a given name.
- `fromRequest().query(String key, int index)`: Returns the nth query parameter with a given name.
- `fromRequest().path()`: Returns the full path.
- `fromRequest().path(int index)`: Returns the nth path element.
- `fromRequest().header(String key)`: Returns the first header with a given name.
- `fromRequest().header(String key, int index)`: Returns the nth header with a given name.
- `fromRequest().body()`: Returns the full request body.
- `fromRequest().body(String jsonPath)`: Returns the element from the request that matches the JSON Path.

如果您使用YAML合同定义，则必须使用 Handlebars `{{{ }}}`表示法以及定制的Spring Cloud Contract函数来实现此目的。

- `{{{ request.url }}}`: Returns the request URL and query parameters.
- `{{{ request.query.key.[index] }}}`: Returns the nth query parameter with a given name. E.g. for key `foo`, first entry `{{{ request.query.foo.[0] }}}`
- `{{{ request.path }}}`: Returns the full path.
- `{{{ request.path.[index] }}}`: Returns the nth path element. E.g. for first entry `{{{ request.path.[0] }}}
- `{{{ request.headers.key }}}`: Returns the first header with a given name.
- `{{{ request.headers.key.[index] }}}`: Returns the nth header with a given name.
- `{{{ request.body }}}`: Returns the full request body.
- `{{{ jsonpath this 'your.json.path' }}}`: Returns the element from the request that matches the JSON Path. E.g. for json path `$.foo` - `{{{ jsonpath this '$.foo' }}}`

考虑以下合同：

Groovy DSL。

```
Contract contractDsl = Contract.make {
 request {
  method 'GET'
  url('/api/v1/xxxx') {
   queryParameters {
    parameter("foo", "bar")
    parameter("foo", "bar2")
   }
  }
  headers {
   header(authorization(), "secret")
   header(authorization(), "secret2")
  }
  body(foo: "bar", baz: 5)
 }
 response {
  status OK()
  headers {
   header(authorization(), "foo ${fromRequest().header(authorization())} bar")
  }
  body(
    url: fromRequest().url(),
    path: fromRequest().path(),
    pathIndex: fromRequest().path(1),
    param: fromRequest().query("foo"),
    paramIndex: fromRequest().query("foo", 1),
    authorization: fromRequest().header("Authorization"),
    authorization2: fromRequest().header("Authorization", 1),
    fullBody: fromRequest().body(),
    responseFoo: fromRequest().body('$.foo'),
    responseBaz: fromRequest().body('$.baz'),
    responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla"
  )
 }
}
```

YAML。

```
request:
  method: GET
  url: /api/v1/xxxx
  queryParameters:
    foo:
      - bar
      - bar2
  headers:
    Authorization:
      - secret
      - secret2
  body:
    foo: bar
    baz: 5
response:
  status: 200
  headers:
    Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"
  body:
    url: "{{{ request.url }}}"
    path: "{{{ request.path }}}"
    pathIndex: "{{{ request.path.1 }}}"
    param: "{{{ request.query.foo }}}"
    paramIndex: "{{{ request.query.foo.1 }}}"
    authorization: "{{{ request.headers.Authorization.0 }}}"
    authorization2: "{{{ request.headers.Authorization.1 }}"
    fullBody: "{{{ request.body }}}"
    responseFoo: "{{{ jsonpath this '$.foo' }}}"
    responseBaz: "{{{ jsonpath this '$.baz' }}}"
    responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"
```

运行JUnit测试生成将导致类似以下示例的测试：

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Authorization", "secret")
   .header("Authorization", "secret2")
   .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
 ResponseOptions response = given().spec(request)
   .queryParam("foo","bar")
   .queryParam("foo","bar2")
   .get("/api/v1/xxxx");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
 assertThat(response.header("Authorization")).isEqualTo("foo secret bar");
// and:
 DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
 assertThatJson(parsedJson).field("['fullBody']").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");
 assertThatJson(parsedJson).field("['authorization']").isEqualTo("secret");
 assertThatJson(parsedJson).field("['authorization2']").isEqualTo("secret2");
 assertThatJson(parsedJson).field("['path']").isEqualTo("/api/v1/xxxx");
 assertThatJson(parsedJson).field("['param']").isEqualTo("bar");
 assertThatJson(parsedJson).field("['paramIndex']").isEqualTo("bar2");
 assertThatJson(parsedJson).field("['pathIndex']").isEqualTo("v1");
 assertThatJson(parsedJson).field("['responseBaz']").isEqualTo(5);
 assertThatJson(parsedJson).field("['responseFoo']").isEqualTo("bar");
 assertThatJson(parsedJson).field("['url']").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2");
 assertThatJson(parsedJson).field("['responseBaz2']").isEqualTo("Bla bla bar bla bla");
```

如您所见，请求中的元素已在响应中正确引用。

生成的WireMock存根应类似于以下示例：

```
{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['baz'] == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\"authorization\":\"{{{request.headers.Authorization.[0]}}}\",\"path\":\"{{{request.path}}}\",\"responseBaz\":{{{jsonpath this '$.baz'
    "headers" : {
      "Authorization" : "{{{request.headers.Authorization.[0]}}};foo"
    },
    "transformers" : [ "response-template" ]
  }
}
```

发送诸如合同 `request` 部分中提出的请求后，将发送以下响应主体：

```
{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "path" : "/api/v1/xxxx",
  "pathIndex" : "v1",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\":\"bar\",\"baz\":5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}
```

Important

此功能只适用于版本大于或等于2.5.1的WireMock。 Spring Cloud Contract Verifier使用WireMock的 `response-template` 响应变换器。 它使用 `{{{ }}}` 将胡须 `{{{ }}}` 模板转换为适当的值。 此外，它还注册了两个辅助功能：

- `escapejsonbody`: Escapes the request body in a format that can be embedded in a JSON.
- `jsonpath`: For a given parameter, find an object in the request body.

91.5.6 Registering Your Own WireMock Extension 译：91.5.6注册自己的WireMock扩展

WireMock让你注册自定义扩展。 默认情况下，Spring Cloud Contract会注册转换器，它允许您引用来自响应的请求。 如果你想提供你自己的扩展，你可以注册一个 `org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions` 接口的实现。 由于我们使用了spring.factories扩展方法，因此您可以在 `META-INF/spring.factories` 文件中创建类似于以下内容的条目：

```
org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\
org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtensions
```

以下是自定义扩展的示例：

TestWireMockExtensions.groovy。

```
package org.springframework.cloud.contract.verifier.dsl.wiremock

import com.github.tomakehurst.wiremock.extension.Extension

/**
 * Extension that registers the default transformer and the custom one
 */
class TestWireMockExtensions implements WireMockExtensions {
 @Override
 List<Extension> extensions() {
  return [
    new DefaultResponseTransformer(),
    new CustomExtension()
  ]
 }
}

class CustomExtension implements Extension {

 @Override
 String getName() {
  return "foo-transformer"
 }
}
```

Important

请记住，要覆盖 `applyGlobally()` 方法并将其设置为 `false` 如果您希望将转换仅应用于明确需要它的映射。

91.5.7 Dynamic Properties in the Matchers Sections 来源:91.5.7匹配器段中的动态属性

如果您使用Pact，下面的讨论可能看起来很熟悉。 不少用户习惯于将身体分开，并设置合同的动态部分。

您可以使用 `bodyMatchers` 部分有两个原因：

- Define the dynamic values that should end up in a stub. You can set it in the `request` or `inputMessage` part of your contract.
- Verify the result of your test. This section is present in the `response` or `outputMessage` side of the contract.

目前，Spring Cloud Contract Verifier仅支持基于JSON路径的匹配器，并具有以下匹配可能性：

Groovy DSL

- 对于存根（在消费者方面的测试中）：
  - `byEquality()`: The value taken from the consumer's request via the provided JSON Path must be equal to the value provided in the contract.
  - `byRegex(… )`: The value taken from the consumer's request via the provided JSON Path must match the regex.
  - `byDate()`: The value taken from the consumer's request via the provided JSON Path must match the regex for an ISO Date value.
  - `byTimestamp()`: The value taken from the consumer's request via the provided JSON Path must match the regex for an ISO DateTime value.
  - `byTime()`: The value taken from the consumer's request via the provided JSON Path must match the regex for an ISO Time value.
- 对于验证（在生产者端生成的测试中）：
  - `byEquality()`: The value taken from the producer's response via the provided JSON Path must be equal to the provided value in the contract.
  - `byRegex(… )`: The value taken from the producer's response via the provided JSON Path must match the regex.
  - `byDate()`: The value taken from the producer's response via the provided JSON Path must match the regex for an ISO Date value.
  - `byTimestamp()`: The value taken from the producer's response via the provided JSON Path must match the regex for an ISO DateTime value.
  - `byTime()`: The value taken from the producer's response via the provided JSON Path must match the regex for an ISO Time value.
  - `byType()`: The value taken from the producer's response via the provided JSON Path needs to be of the same type as the type defined in the body of the response in the contract. `byType` can take a closure, in which you can set `minOccurrence` and `maxOccurrence`. That way, you can assert the size of the flattened collection. To check the size of an unflattened collection, use a custom method with the `byCommand(… )` testMatcher.
  - `byCommand(… )`：通过提供的JSON路径从生产者响应中获取的值作为输入传递给您提供的自定义方法。 例如， `byCommand('foo($it)')` 导致调用 `foo` 方法，其中与JSON路径匹配的值将通过该方法。 根据JSON路径，从JSON中读取的对象的类型可以是下列之一：
    - `String`: If you point to a `String` value.
    - `JSONArray`: If you point to a `List`.
    - `Map`: If you point to a `Map`.
    - `Number`: If you point to `Integer`, `Double`, or other kind of number.
    - `Boolean`: If you point to a `Boolean`.
  - `byNull()`: The value taken from the response via the provided JSON Path must be null

YAML。 *请阅读Groovy章节，详细解释类型的含义*

对于YAML来说，匹配器的结构看起来像这样

```
- path: $.foo
  type: by_regex
  value: bar
```

或者，如果您想使用其中一个预定义的正则表达式
`[only_alpha_unicode, number, any_boolean, ip_address, hostname, email, url, uuid, iso_date, iso_date_time, iso_time, iso_8601_with_offset, non_emp`
：

```
- path: $.foo
  type: by_regex
  predefined: only_alpha_unicode
```

下面你可以找到允许的`类型`列表。

- 对于 `stubMatchers`：
  - `by_equality`
  - `by_regex`
  - `by_date`
  - `by_timestamp`
  - `by_time`
- 对于 `testMatchers`：
  - `by_equality`
  - `by_regex`
  - `by_date`
  - `by_timestamp`
  - `by_time`
  - `by_type`
    - there are 2 additional fields accepted: `minOccurrence` and `maxOccurrence`.
  - `by_command`
  - `by_null`

考虑下面的例子：

Groovy DSL。

```
Contract contractDsl = Contract.make {
 request {
  method 'GET'
  urlPath '/get'
  body([
    duck: 123,
    alpha: "abc",
    number: 123,
    aBoolean: true,
    date: "2017-01-01",
    dateTime: "2017-01-01T01:23:45",
    time: "01:02:34",
    valueWithoutAMatcher: "foo",
    valueWithTypeMatch: "string",
    key: [
      'complex.key' : 'foo'
    ]
  ])
  bodyMatchers {
   jsonPath('$.duck', byRegex("[0-9]{3}"))
   jsonPath('$.duck', byEquality())
   jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))
   jsonPath('$.alpha', byEquality())
   jsonPath('$.number', byRegex(number()))
   jsonPath('$.aBoolean', byRegex(anyBoolean()))
   jsonPath('$.date', byDate())
   jsonPath('$.dateTime', byTimestamp())
   jsonPath('$.time', byTime())
   jsonPath("\$.['key'].['complex.key']", byEquality())
  }
  headers {
   contentType(applicationJson())
  }
 }
 response {
  status OK()
  body([
    duck: 123,
    alpha: "abc",
    number: 123,
    positiveInteger: 1234567890,
    negativeInteger: -1234567890,
    positiveDecimalNumber: 123.4567890,
    negativeDecimalNumber: -123.4567890,
    aBoolean: true,
    date: "2017-01-01",
    dateTime: "2017-01-01T01:23:45",
    time: "01:02:34",
    valueWithoutAMatcher: "foo",
    valueWithTypeMatch: "string",
    valueWithMin: [
     1,2,3
    ],
    valueWithMax: [
     1,2,3
    ],
    valueWithMinMax: [
     1,2,3
    ],
    valueWithMinEmpty: [],
    valueWithMaxEmpty: [],
    key: [
            'complex.key' : 'foo'
    ],
    nullValue: null
  ])
  bodyMatchers {
   // asserts the jsonpath value against manual regex
   jsonPath('$.duck', byRegex("[0-9]{3}"))
   // asserts the jsonpath value against the provided value
   jsonPath('$.duck', byEquality())
   // asserts the jsonpath value against some default regex
   jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))
   jsonPath('$.alpha', byEquality())
   jsonPath('$.number', byRegex(number()))
   jsonPath('$.positiveInteger', byRegex(anInteger()))
```

```
   jsonPath('$.negativeInteger', byRegex(anInteger()))
   jsonPath('$.positiveDecimalNumber', byRegex(aDouble()))
   jsonPath('$.negativeDecimalNumber', byRegex(aDouble()))
   jsonPath('$.aBoolean', byRegex(anyBoolean()))
   // asserts vs inbuilt time related regex
   jsonPath('$.date', byDate())
   jsonPath('$.dateTime', byTimestamp())
   jsonPath('$.time', byTime())
   // asserts that the resulting type is the same as in response body
   jsonPath('$.valueWithTypeMatch', byType())
   jsonPath('$.valueWithMin', byType {
    // results in verification of size of array (min 1)
    minOccurrence(1)
   })
   jsonPath('$.valueWithMax', byType {
    // results in verification of size of array (max 3)
    maxOccurrence(3)
   })
   jsonPath('$.valueWithMinMax', byType {
    // results in verification of size of array (min 1 & max 3)
    minOccurrence(1)
    maxOccurrence(3)
   })
   jsonPath('$.valueWithMinEmpty', byType {
    // results in verification of size of array (min 0)
    minOccurrence(0)
   })
   jsonPath('$.valueWithMaxEmpty', byType {
    // results in verification of size of array (max 0)
    maxOccurrence(0)
   })
   // will execute a method `assertThatValueIsANumber`
   jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))
   jsonPath("\$.['key'].['complex.key']", byEquality())
   jsonPath('$.nullValue', byNull())
  }
  headers {
   contentType(applicationJson())
   header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))
  }
 }
}
```

YAML。

```
request:
  method: GET
  urlPath: /get
  body:
    duck: 123
    alpha: "abc"
    number: 123
    aBoolean: true
    date: "2017-01-01"
    dateTime: "2017-01-01T01:23:45"
    time: "01:02:34"
    valueWithoutAMatcher: "foo"
    valueWithTypeMatch: "string"
    key:
      "complex.key": 'foo'
    nullValue: null
  matchers:
    headers:
      - key: Content-Type
        regex: "application/json.*"
    body:
      - path: $.duck
        type: by_regex
        value: "[0-9]{3}"
      - path: $.duck
        type: by_equality
      - path: $.alpha
        type: by_regex
        predefined: only_alpha_unicode
      - path: $.alpha
        type: by_equality
      - path: $.number
        type: by_regex
        predefined: number
      - path: $.aBoolean
        type: by_regex
        predefined: any_boolean
      - path: $.date
        type: by_date
      - path: $.dateTime
        type: by_timestamp
      - path: $.time
        type: by_time
      - path: "$.['key'].['complex.key']"
        type: by_equality
      - path: $.nullvalue
        type: by_null
  headers:
    Content-Type: application/json
response:
  status: 200
  body:
    duck: 123
    alpha: "abc"
```

```
    number: 123
    aBoolean: true
    date: "2017-01-01"
    dateTime: "2017-01-01T01:23:45"
    time: "01:02:34"
    valueWithoutAMatcher: "foo"
    valueWithTypeMatch: "string"
    valueWithMin:
      - 1
      - 2
      - 3
    valueWithMax:
      - 1
      - 2
      - 3
    valueWithMinMax:
      - 1
      - 2
      - 3
    valueWithMinEmpty: []
    valueWithMaxEmpty: []
    key:
      'complex.key' : 'foo'
    nulValue: null
  matchers:
    headers:
      - key: Content-Type
        regex: "application/json.*"
    body:
      - path: $.duck
        type: by_regex
        value: "[0-9]{3}"
      - path: $.duck
        type: by_equality
      - path: $.alpha
        type: by_regex
        predefined: only_alpha_unicode
      - path: $.alpha
        type: by_equality
      - path: $.number
        type: by_regex
        predefined: number
      - path: $.aBoolean
        type: by_regex
        predefined: any_boolean
      - path: $.date
        type: by_date
      - path: $.dateTime
        type: by_timestamp
      - path: $.time
        type: by_time
      - path: $.valueWithTypeMatch
        type: by_type
      - path: $.valueWithMin
        type: by_type
        minOccurrence: 1
      - path: $.valueWithMax
        type: by_type
        maxOccurrence: 3
      - path: $.valueWithMinMax
        type: by_type
        minOccurrence: 1
        maxOccurrence: 3
      - path: $.valueWithMinEmpty
        type: by_type
        minOccurrence: 0
      - path: $.valueWithMaxEmpty
        type: by_type
        maxOccurrence: 0
      - path: $.duck
        type: by_command
        value: assertThatValueIsANumber($it)
      - path: $.nullValue
        type: by_null
        value: null
  headers:
    Content-Type: application/json
```

在上例中，您可以在`matchers`部分查看合约的动态部分。 对于请求部分，您可以看到除`valueWithoutAMatcher`所有字段都显式设置存根应包含的正则表达式的值。 对于`valueWithoutAMatcher` ，验证以与不使用匹配器相同的方式进行。 在这种情况下，测试执行相等性检查。

对于`bodyMatchers`部分中的响应方面，我们以类似的方式定义动态部分。 唯一的区别是`byType`匹配器也存在。 验证器引擎检查四个字段以验证来自测试的响应是否具有JSON路径与给定字段匹配的值，与响应正文中定义的类型相同，并通过以下检查（基于方法被调用）：

- For `$.valueWithTypeMatch`, the engine checks whether the type is the same.
- For `$.valueWithMin`, the engine check the type and asserts whether the size is greater than or equal to the minimum occurrence.
- For `$.valueWithMax`, the engine checks the type and asserts whether the size is smaller than or equal to the maximum occurrence.
- For `$.valueWithMinMax`, the engine checks the type and asserts whether the size is between the min and maximum occurrence.

结果测试将类似于以下示例（请注意， `and`节将自动生成的断言与来自匹配器的断言分隔开来）：

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Content-Type", "application/json")
   .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01

// when:
 ResponseOptions response = given().spec(request)
   .get("/get");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
 assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
 DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
 assertThatJson(parsedJson).field("['valueWithoutAMatcher']").isEqualTo("foo");
// and:
 assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
 assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
 assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
 assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
 assertThat(parsedJson.read("$.number", String.class)).matches("-?(\\d*\\.\\d+|\\d+)");
 assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
 assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
 assertThat(parsedJson.read("$.dateTime", String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9])
 assertThat(parsedJson.read("$.time", String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
 assertThat((Object) parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.class);
 assertThat((Object) parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.class);
 assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin", java.util.Collection.class)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);
 assertThat((Object) parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.class);
 assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax", java.util.Collection.class)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);
 assertThat((Object) parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.class);
 assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax", java.util.Collection.class)).as("$.valueWithMinMax").hasSizeBetween(1, 3);
 assertThat((Object) parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.class);
 assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty", java.util.Collection.class)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEq
 assertThat((Object) parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java.util.List.class);
 assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty", java.util.Collection.class)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqual
 assertThatValueIsANumber(parsedJson.read("$.duck"));
 assertThat(parsedJson.read("$.['key'].['complex.key']", String.class)).isEqualTo("foo");
```

Important

请注意，对于`byCommand`方法，该示例调用`assertThatValueIsANumber`。 此方法必须在测试基类中定义或静态导入到测试中。 请注意，`byCommand`电话已转换为`assertThatValueIsANumber(parsedJson.read("$.duck"));`。 这意味着引擎采用了方法名称并将适当的JSON路径作为参数传递给它。

生成的WireMock存根位于以下示例中：

```
    '''
{
  "request" : {
 "urlPath" : "/get",
 "method" : "POST",
 "headers" : {
   "Content-Type" : {
  "matches" : "application/json.*"
   }
 },
 "bodyPatterns" : [ {
   "matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"
 }, {
   "matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"
 }, {
   "matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"
 }, {
   "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] == 4)]"
 }, {
   "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with value')]"
 }, {
   "matchesJsonPath" : "$[?(@.duck =~ /([0-9]{3})/)]"
 }, {
   "matchesJsonPath" : "$[?(@.duck == 123)]"
 }, {
   "matchesJsonPath" : "$[?(@.alpha =~ /([\\\\p{L}]*)/)]"
 }, {
   "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
 }, {
   "matchesJsonPath" : "$[?(@.number =~ /(-?(\\\\d*\\\\.\\\\d+|\\\\d+))/)]"
 }, {
   "matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"
 }, {
   "matchesJsonPath" : "$[?(@.date =~ /((\\\\d\\\\d\\\\d\\\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01]))/)]"
 }, {
   "matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
 }, {
   "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
 }, {
   "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
 } ]
  },
  "response" : {
 "status" : 200,
 "body" : "{\\"date\\":\\"2017-01-01\\",\\"dateTime\\":\\"2017-01-01T01:23:45\\",\\"number\\":123,\\"aBoolean\\":true,\\"duck\\":123,\\"alpha\\":\\"a
 "headers" : {
   "Content-Type" : "application/json"
 }
  }
}
'''
```

Important

如果您使用`matcher`，那么`matcher`地址与JSON路径的请求和响应部分将从断言中删除。 在验证集合的情况下，您必须为集合的**所有**元素创建匹配器。

考虑下面的例子：

```
Contract.make {
    request {
        method 'GET'
        url("/foo")
    }
    response {
        status OK()
        body(events: [[
                                operation          : 'EXPORT',
                                eventId            : '16f1ed75-0bcc-4f0d-a04d-3121798faf99',
                                status             : 'OK'
                        ], [
                                operation          : 'INPUT_PROCESSING',
                                eventId            : '3bb4ac82-6652-462f-b6d1-75e424a0024a',
                                status             : 'OK'
                        ]
                ]
        )
        bodyMatchers {
            jsonPath('$.events[0].operation', byRegex('.+'))
            jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))
            jsonPath('$.events[0].status', byRegex('.+'))
        }
    }
}
```

上述代码导致创建以下测试（代码块仅显示断言部分）：

```
and:
 DocumentContext parsedJson = JsonPath.parse(response.body.asString())
 assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("16f1ed75-0bcc-4f0d-a04d-3121798faf99")
 assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("EXPORT")
 assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("INPUT_PROCESSING")
 assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("3bb4ac82-6652-462f-b6d1-75e424a0024a")
 assertThatJson(parsedJson).array("['events']").contains("['status']").isEqualTo("OK")
and:
 assertThat(parsedJson.read("\$.events[0].operation", String.class)).matches(".+")
 assertThat(parsedJson.read("\$.events[0].eventId", String.class)).matches("^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]
 assertThat(parsedJson.read("\$.events[0].status", String.class)).matches(".+")
```

正如你所看到的，断言是畸形的。 只有数组的第一个元素被置位。 为了解决这个问题，你应该把这个断言应用到整个`$.events`集合中，并用`byCommand(… )`方法断言它。

91.6 JAX-RS Support译：91.6 JAX-RS支持

Spring Cloud Contract Verifier支持JAX-RS 2客户端API。 基类需要定义`protected WebTarget webTarget`和服务器初始化。 测试JAX-RS API的唯一选择是启动Web服务器。 此外，具有主体的请求需要设置内容类型。 否则，将使用默认的`application/octet-stream` 。

为了使用JAX-RS模式，请使用以下设置：

```
testMode == 'JAXRSCLIENT'
```

以下示例显示了生成的测试API：

```
'''
 // when:
  Response response = webTarget
    .path("/users")
    .queryParam("limit", "10")
    .queryParam("offset", "20")
    .queryParam("filter", "email")
    .queryParam("sort", "name")
    .queryParam("search", "55")
    .queryParam("age", "99")
    .queryParam("name", "Denis.Stepanov")
    .queryParam("email", "bob@email.com")
    .request()
    .method("GET");

  String responseAsString = response.readEntity(String.class);

 // then:
  assertThat(response.getStatus()).isEqualTo(200);
 // and:
  DocumentContext parsedJson = JsonPath.parse(responseAsString);
  assertThatJson(parsedJson).field("['property1']").isEqualTo("a");
'''
```

91.7 Async Support译：91.7异步支持

如果有啊€™重新使用在服务器端异步通信（你的控制器返回`Callable`， `DeferredResult`，等等），那么，你的合同里面，你必须提供一个`async()`的方法`response`部分。 以下代码显示了一个示例：

Groovy DSL。

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status OK()
        body 'Passed'
        async()
    }
}
```

YAML。

```
response:
    async: true
```

91.8 Working with Context Paths译：91.8使用上下文路径

Spring Cloud Contract支持上下文路径。

Important

完全支持上下文路径所需的唯一更改是**PRODUCER**方面的开关。 另外，自动生成的测试必须使用**EXPLICIT**模式。 消费者方面保持不变。 为了使生成的测试通过，您必须使用**EXPLICIT**模式。

Maven的。

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <testMode>EXPLICIT</testMode>
    </configuration>
</plugin>
```

摇篮。

```
contracts {
  testMode = 'EXPLICIT'
}
```

这样，你生成了一个**不**使用MockMvc的测试。 这意味着你产生了真正的请求，你需要设置你生成的测试的基类来处理一个真正的套接字。

考虑以下合同：

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'GET'
  url '/my-context-path/url'
 }
 response {
  status OK()
 }
}
```

以下示例显示如何设置基类和Rest Assured：

```
import io.restassured.RestAssured;
import org.junit.Before;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

 @LocalServerPort int port;

 @Before
 public void setup() {
  RestAssured.baseURI = "http://localhost";
  RestAssured.port = this.port;
 }
}
```

如果你这样做：

- All of your requests in the autogenerated tests are sent to the real endpoint with your context path included (for example, `/my-context-path/url`).
- Your contracts reflect that you have a context path. Your generated stubs also have that information (for example, in the stubs, you have to call `/my-context-path/url`).

91.9 Messaging Top-Level Elements译：91.9消息顶级元素

用于消息传递的DSL看起来与专注于HTTP的DSL稍有不同。 以下几节解释了这些差异：

- Section 91.9.1, “Output Triggered by a Method”
- Section 91.9.2, “Output Triggered by a Message”
- Section 91.9.3, “Consumer/Producer”
- Section 91.9.4, “Common”

91.9.1 Output Triggered by a Method译：91.9.1由方法触发的输出

输出消息可以通过调用一个方法来触发（例如 `Scheduler`当a启动并发送消息时），如以下示例所示：

Groovy DSL。

```
def dsl = Contract.make {
 // Human readable description
 description 'Some description'
 // Label by means of which the output message can be triggered
 label 'some_label'
 // input to the contract
 input {
  // the contract will be triggered by a method
  triggeredBy('bookReturnedTriggered()')
 }
 // output message of the contract
 outputMessage {
  // destination to which the output message will be sent
  sentTo('output')
  // the body of the output message
  body('''{ "bookName" : "foo" }''')
  // the headers of the output message
  headers {
   header('BOOK-NAME', 'foo')
  }
 }
}
```

YAML。

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
input:
  # the contract will be triggered by a method
  triggeredBy: bookReturnedTriggered()
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

在前面的示例中，如果执行名为`bookReturnedTriggered`的方法，则输出消息将发送到`output`。 在消息**发布者**方面，我们生成一个调用该方法来触发消息的测试。 在**消费者**方面，您可以使用`some_label`来触发该消息。

91.9.2 Output Triggered by a Message译：91.9.2由消息触发的输出

输出消息可以通过接收消息来触发，如以下示例所示：

Groovy DSL。

```
def dsl = Contract.make {
 description 'Some Description'
 label 'some_label'
 // input is a message
 input {
  // the message was received from this destination
  messageFrom('input')
  // has the following body
  messageBody([
          bookName: 'foo'
  ])
  // and the following headers
  messageHeaders {
   header('sample', 'header')
  }
 }
 outputMessage {
  sentTo('output')
  body([
          bookName: 'foo'
  ])
  headers {
   header('BOOK-NAME', 'foo')
  }
 }
}
```

YAML。

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
# input is a message
input:
  messageFrom: input
  # has the following body
  messageBody:
    bookName: 'foo'
  # and the following headers
  messageHeaders:
    sample: 'header'
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

在前面的示例中，如果在`input`目标上接收到适当的消息， `output`将输出消息发送到`output`。 在消息**发布者**方面，引擎生成一个测试，将输入消息发送到定义的目标。 在**消费者**方面，您可以向输入目标发送消息或使用标签（ `some_label`中为`some_label` ）来触发消息。

91.9.3 Consumer/Producer译：91.9.3消费者/生产者

Important

本节仅适用于Groovy DSL。

在HTTP中，你有一个概念`client` / `stub and `server` / `test`符号。 您也可以在消息传递中使用这些范例。 另外，Spring Cloud Contract Verifier还提供了`consumer`和`producer`方法，如以下示例所示（请注意，您可以使用`$`或`value`方法提供`consumer`和`producer`部件）：

```
Contract.make {
 label 'some_label'
 input {
  messageFrom value(consumer('jms:output'), producer('jms:input'))
  messageBody([
    bookName: 'foo'
  ])
  messageHeaders {
   header('sample', 'header')
  }
 }
 outputMessage {
  sentTo $(consumer('jms:input'), producer('jms:output'))
  body([
    bookName: 'foo'
  ])
 }
}
```

91.9.4 Common 译：91.9.4常见

在`input`或`outputMessage`部分，您可以调用`assertThat`，其名称为`method`（例如`assertThatMessageIsOnTheQueue()`），您已在基类或静态导入中定义该名称。 Spring Cloud Contract将在生成的测试中执行该方法。

91.10 Multiple Contracts in One File 译：91.10一个文件中的多个合同

您可以在一个文件中定义多个合约。 这样的合同可能类似于下面的例子：

Groovy DSL。

```
import org.springframework.cloud.contract.spec.Contract

[
        Contract.make {
            name("should post a user")
            request {
                method 'POST'
                url('/users/1')
            }
            response {
                status OK()
            }
        },
        Contract.make {
            request {
                method 'POST'
                url('/users/2')
            }
            response {
                status OK()
            }
        }
]
```

YAML。

```
---
name: should post a user
request:
  method: POST
  url: /users/1
response:
  status: 200

---
request:
  method: POST
  url: /users/2
response:
  status: 200
```

在前面的例子中，一个合约有`name`字段，另一个没有。 这导致产生两个或多或少看起来像这样的测试：

```
package org.springframework.cloud.contract.verifier.tests.com.hello;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;

import static com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

 @Test
 public void validate_should_post_a_user() throws Exception {
  // given:
   MockMvcRequestSpecification request = given();

  // when:
   ResponseOptions response = given().spec(request)
     .post("/users/1");

  // then:
   assertThat(response.statusCode()).isEqualTo(200);
 }

 @Test
 public void validate_withList_1() throws Exception {
  // given:
   MockMvcRequestSpecification request = given();

  // when:
   ResponseOptions response = given().spec(request)
     .post("/users/2");

  // then:
   assertThat(response.statusCode()).isEqualTo(200);
 }

}
```

请注意，对于具有`name`字段的合同，生成的测试方法名为`validate_should_post_a_user`。对于没有名字的那个，它被称为`validate_withList_1`。它对应于文件名称`WithList.groovy`和列表中合同的索引。

以下示例显示了生成的存根：

```
should post a user.json
1_WithList.json
```

如您所见，第一个文件从合同中获得了`name`参数。第二个获得以索引为前缀的合同文件名称（`WithList.groovy`）（在这种情况下，合同文件中的合同列表中的索引为`1`）。

正如你所看到的，如果你为合同命名，因为这样做会让你的测试更有意义。

91.11 Generating Spring REST Docs snippets from the contracts译：91.11从合同生成Spring REST Docs片段

如果您想使用Spring REST Docs来包含API的请求和响应，则只需使用MockMvc和RestAssuredMockMvc对安装进行一些较小的更改即可。如果你还没有，只需包含以下依赖关系。

Maven的。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-contract-verifier</artifactId>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>org.springframework.restdocs</groupId>
 <artifactId>spring-restdocs-mockmvc</artifactId>
 <optional>true</optional>
</dependency>
```

摇篮。

```
testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
testCompile 'org.springframework.restdocs:spring-restdocs-mockmvc'
```

接下来，您需要对基类进行一些更改，如下例所示。

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

import org.junit.Before;
import org.junit.Rule;
import org.junit.rules.TestName;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.restdocs.JUnitRestDocumentation;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class)
public abstract class FraudBaseWithWebAppSetup {

 private static final String OUTPUT = "target/generated-snippets";

 @Rule
 public JUnitRestDocumentation restDocumentation = new JUnitRestDocumentation(OUTPUT);

 @Rule public TestName testName = new TestName();

 @Autowired
 private WebApplicationContext context;

 @Before
 public void setup() {
 RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(this.context)
   .apply(documentationConfiguration(this.restDocumentation))
   .alwaysDo(document(getClass().getSimpleName() + "_" + testName.getMethodName()))
   .build());
 }

 protected void assertThatRejectionReasonIsNull(Object rejectionReason) {
  assert rejectionReason == null;
 }
}
```

如果您正在使用独立设置，则可以像这样设置RestAssuredMockMvc：

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.Rule;
import org.junit.rules.TestName;
import org.springframework.restdocs.JUnitRestDocumentation;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

public abstract class FraudBaseWithStandaloneSetup {

 private static final String OUTPUT = "target/generated-snippets";

 @Rule
 public JUnitRestDocumentation restDocumentation = new JUnitRestDocumentation(OUTPUT);

 @Rule public TestName testName = new TestName();

 @Before
 public void setup() {
  RestAssuredMockMvc.standaloneSetup(MockMvcBuilders.standaloneSetup(new FraudDetectionController())
    .apply(documentationConfiguration(this.restDocumentation))
    .alwaysDo(document(getClass().getSimpleName() + "_" + testName.getMethodName())));
 }

}
```

您不需要为自Spring REST Docs 1.2.0.RELEASE以来生成的片段指定输出目录。

92. Customization译：92.定制

Important

本节仅适用于Groovy DSL

您可以通过扩展DSL来自定义Spring Cloud Contract Verifier，如本节的其余部分所示。

92.1 Extending the DSL译：92.1扩展DSL

您可以将自己的功能提供给DSL。 此功能的关键要求是保持静态兼容性。 在本文档的后面，您可以看到以下示例：

- Creating a JAR with reusable classes.
- Referencing of these classes in the DSLs.

你可以找到完整的例子 here 。

92.1.1 Common JAR译：92.1.1通用JAR

以下示例显示了可以在DSL中重用的三个类。

PatternUtils包含 **消费者**和 **生产者**使用的函数。

```
package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your tests
 * then you can create a class resembling this one. It can
 * contain all the {@link Pattern} you want to use in the DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class PatternUtils {

 public static String tooYoung() {
  //remove::start[]
  return "[0-1][0-9]";
  //remove::end[return]
 }

 public static Pattern oldEnough() {
  //remove::start[]
  return Pattern.compile("[2-9][0-9]");
  //remove::end[return]
 }

 /**
  * Makes little sense but it's just an example ;)
  */
 public static Pattern ok() {
  //remove::start[]
  return Pattern.compile("OK");
  //remove::end[return]
 }
}
//end::impl[]
```

ConsumerUtils包含由 **消费者**使用的功能。

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ClientDslProperty;

/**
 * DSL Properties passed to the DSL from the consumer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you can have a regular expression.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ConsumerUtils {
 /**
  * Consumer side property. By using the {@link ClientDslProperty}
  * you can omit most of boilerplate code from the perspective
  * of dynamic values. Example
  *
  * <pre>
  * {@code
  * request {
  *     body(
  *         [ age: $(ConsumerUtils.oldEnough())]
  *     )
  * }
  * </pre>
  *
  * That way it's in the implementation that we decide what value we will pass to the consumer
  * and which one to the producer.
  *
  * @author Marcin Grzejszczak
  */
 public static ClientDslProperty oldEnough() {
  //remove::start[]
  // this example is not the best one and
  // theoretically you could just pass the regex instead of `ServerDslProperty` but
  // it's just to show some new tricks :)
  return new ClientDslProperty(PatternUtils.oldEnough(), 40);
  //remove::end[return]
 }

}
//end::impl[]
```

ProducerUtils包含由 **生产者** 使用的功能。

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ServerDslProperty;

/**
 * DSL Properties passed to the DSL from the producer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you have to have a concrete value.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ProducerUtils {

 /**
  * Producer side property. By using the {@link ProducerUtils}
  * you can omit most of boilerplate code from the perspective
  * of dynamic values. Example
  *
  * <pre>
  * {@code
  * response {
  *     body(
  *         [ status: $(ProducerUtils.ok())]
  *     )
  * }
  * </pre>
  *
  * That way it's in the implementation that we decide what value we will pass to the consumer
  * and which one to the producer.
  */
 public static ServerDslProperty ok() {
  // this example is not the best one and
  // theoretically you could just pass the regex instead of `ServerDslProperty` but
  // it's just to show some new tricks :)
  return new ServerDslProperty( PatternUtils.ok(), "OK");
 }
}
//end::impl[]
```

92.1.2 Adding the Dependency to the Project 译：92.1.2向项目添加依赖项

为了使插件和IDE能够引用常见的JAR类，您需要将依赖项传递给您的项目。

92.1.3 Test the Dependency in the Project's Dependencies 译：92.1.3测试项目依赖项中的依赖项

首先，添加通用jar依赖项作为测试依赖项。 由于您的合同文件在测试资源路径中可用，常见的jar类会自动在您的Groovy文件中可见。 以下示例显示如何测试依赖关系：

Maven的。

```
<dependency>
 <groupId>com.example</groupId>
 <artifactId>beer-common</artifactId>
 <version>${project.version}</version>
 <scope>test</scope>
</dependency>
```

摇篮。

```
testCompile("com.example:beer-common:0.0.1-SNAPSHOT")
```

92.1.4 Test a Dependency in the Plugin's Dependencies译：92.1.4在插件的依赖项中测试依赖项

现在，您必须添加插件的依赖项以在运行时重用，如以下示例所示：

Maven的。

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
 <configuration>
  <packageWithBaseClasses>com.example</packageWithBaseClasses>
  <baseClassMappings>
   <baseClassMapping>
    <contractPackageRegex>.*intoxication.*</contractPackageRegex>
    <baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
   </baseClassMapping>
  </baseClassMappings>
 </configuration>
 <dependencies>
  <dependency>
   <groupId>com.example</groupId>
   <artifactId>beer-common</artifactId>
   <version>${project.version}</version>
   <scope>compile</scope>
  </dependency>
 </dependencies>
</plugin>
```

摇篮。

```
classpath "com.example:beer-common:0.0.1-SNAPSHOT"
```

92.1.5 Referencing classes in DSLs译：92.1.5引用DSL中的类

您现在可以在DSL中引用您的类，如以下示例所示：

```
package contracts.beer.rest

import com.example.ConsumerUtils
import com.example.ProducerUtils
import org.springframework.cloud.contract.spec.Contract

Contract.make {
 description("""
Represents a successful scenario of getting a beer

```
given:
 client is old enough
when:
 he applies for a beer
then:
 we'll grant him the beer
```

""")
 request {
  method 'POST'
  url '/check'
  body(
    age: $(ConsumerUtils.oldEnough())
)
  headers {
   contentType(applicationJson())
  }
 }
 response {
  status 200
  body("""
   {
    "status": "${value(ProducerUtils.ok())}"
   }
   """)
  headers {
   contentType(applicationJson())
  }
 }
}
```

93. Using the Pluggable Architecture译：93.使用可插入的体系结构

您可能会遇到您已经以其他格式定义合同的情况，例如YAML，RAML或PACT。 在这些情况下，您仍然想从测试和存根的自动生成中受益。 您可以添加自己的实现来生成测试和存根。 此外，您可以自定义生成测试的方式（例如，可以为其他语言生成测试）以及生成方式存根（例如，可以为其他HTTP服务器实现生成存根）。

93.1 Custom Contract Converter译：93.1自定义合同转换器

`ContractConverter`界面允许您注册自己的合同结构转换器的实现。 以下代码清单显示了`ContractConverter`接口：

```
package org.springframework.cloud.contract.spec

/**
 * Converter to be used to convert FROM {@link File} TO {@link Contract}
 * and from {@link Contract} to {@code T}
 *
 * @param <T> - type to which we want to convert the contract
 *
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
interface ContractConverter<T> {

 /**
  * Should this file be accepted by the converter. Can use the file extension
  * to check if the conversion is possible.
  *
  * @param file - file to be considered for conversion
  * @return - {@code true} if the given implementation can convert the file
  */
 boolean isAccepted(File file)

 /**
  * Converts the given {@link File} to its {@link Contract} representation
  *
  * @param file - file to convert
  * @return - {@link Contract} representation of the file
  */
 Collection<Contract> convertFrom(File file)

 /**
  * Converts the given {@link Contract} to a {@link T} representation
  *
  * @param contract - the parsed contract
  * @return - {@link T} the type to which we do the conversion
  */
 T convertTo(Collection<Contract> contract)
}
```

你的实现必须定义它应该开始转换的条件。 另外，您必须定义如何在两个方向上执行该转换。

Important

创建实现后，您必须创建一个`/META-INF/spring.factories`文件，其中提供了您的实现的完全限定名称。

以下示例显示了一个典型的`spring.factories`文件：

```
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.verifier.converter.YamlContractConverter
```

93.1.1 Pact Converter译：93.1.1 Pact Converter

春季云合同包括对第4 版合约的Pact表示支持。 您可以使用Pact文件来代替使用Groovy DSL。 在本节中，我们将介绍如何为您的项目添加Pact支持。 但请注意，并非所有功能都受支持。 从v3开始，您可以为同一元素组合多个匹配器; 你可以使用匹配器的正文，标题，请求和路径; 你可以使用值生成器。 Spring Cloud Contract目前仅支持使用AND规则逻辑组合的多个匹配器。 在该转换旁边，请求和路径匹配器被跳过。 当使用给定格式的日期，时间或日期时间值生成器时，给定格式将被跳过，并使用ISO格式。

93.1.2 Pact Contract译：93.1.2合约

请考虑以下Pact合同示例，该合同是`src/test/resources/contracts`文件夹下的`src/test/resources/contracts`文件。

```
{
  "provider": {
    "name": "Provider"
  },
  "consumer": {
    "name": "Consumer"
  },
  "interactions": [
    {
      "description": "",
      "request": {
        "method": "PUT",
        "path": "/fraudcheck",
        "headers": {
          "Content-Type": "application/vnd.fraud.v1+json"
        },
        "body": {
          "clientId": "1234567890",
          "loanAmount": 99999
        },
        "generators": {
          "body": {
            "$.clientId": {
              "type": "Regex",
              "regex": "[0-9]{10}"
            }
          }
        },
        "matchingRules": {
          "header": {
            "Content-Type": {
              "matchers": [
                {
                  "match": "regex",
                  "regex": "application/vnd\\.fraud\\.v1\\+json.*"
                }
              ],
              "combine": "AND"
            }
          },
          "body" : {
            "$.clientId": {
              "matchers": [
                {
                  "match": "regex",
                  "regex": "[0-9]{10}"
                }
              ],
              "combine": "AND"
            }
          }
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/vnd.fraud.v1+json;charset=UTF-8"
        },
        "body": {
          "fraudCheckStatus": "FRAUD",
          "rejectionReason": "Amount too high"
        },
        "matchingRules": {
          "header": {
            "Content-Type": {
              "matchers": [
                {
                  "match": "regex",
                  "regex": "application/vnd\\.fraud\\.v1\\+json.*"
                }
              ],
              "combine": "AND"
            }
          },
          "body": {
            "$.fraudCheckStatus": {
              "matchers": [
                {
                  "match": "regex",
                  "regex": "FRAUD"
                }
              ],
              "combine": "AND"
            }
          }
        }
      }
    }
  ],
  "metadata": {
    "pact-specification": {
      "version": "3.0.0"
    },
    "pact-jvm": {
      "version": "3.5.13"
    }
  }
}
```

本部分关于使用Pact的其余部分引用前面的文件。

93.1.3 Pact for Producers译：93.1.3生产者协议

在生产者方面，你必须为你的插件配置添加两个额外的依赖关系。 一个是Spring Cloud Contract Pact支持，另一个代表您使用的当前Pact版本。

Maven的。

```
<plugin>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-maven-plugin</artifactId>
 <version>${spring-cloud-contract.version}</version>
 <extensions>true</extensions>
 <configuration>
  <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
 </configuration>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-contract-pact</artifactId>
   <version>${spring-cloud-contract.version}</version>
  </dependency>
 </dependencies>
</plugin>
```

摇篮。

```
classpath "org.springframework.cloud:spring-cloud-contract-pact:${findProperty('verifierVersion') ?: verifierVersion}"
```

当您执行应用程序的构建时，将会生成一个测试。 生成的测试可能如下所示：

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
 // given:
  MockMvcRequestSpecification request = given()
    .header("Content-Type", "application/vnd.fraud.v1+json")
    .body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

 // when:
  ResponseOptions response = given().spec(request)
    .put("/fraudcheck");

 // then:
  assertThat(response.statusCode()).isEqualTo(200);
  assertThat(response.header("Content-Type")).matches("application/vnd\\.fraud\\.v1\\+json.*");
 // and:
  DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
  assertThatJson(parsedJson).field("['rejectionReason']").isEqualTo("Amount too high");
 // and:
  assertThat(parsedJson.read("$.fraudCheckStatus", String.class)).matches("FRAUD");
}
```

相应生成的存根可能如下所示：

```
{
  "id" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "request" : {
    "url" : "/fraudcheck",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/vnd\\.fraud\\.v1\\+json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['loanAmount'] == 99999)]"
    }, {
      "matchesJsonPath" : "$[?(@.clientId =~ /([0-9]{10})/)]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Amount too high\"}",
    "headers" : {
      "Content-Type" : "application/vnd.fraud.v1+json;charset=UTF-8"
    },
    "transformers" : [ "response-template" ]
  },
}
```

93.1.4 Pact for Consumers译：93.1.4消费者协议

在生产者方面，您必须为您的项目依赖项添加两个附加依赖项。 一个是Spring Cloud Contract Pact支持，另一个代表您使用的当前Pact版本。

Maven的。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-contract-pact</artifactId>
 <scope>test</scope>
</dependency>
```

摇篮。

```
testCompile "org.springframework.cloud:spring-cloud-contract-pact"
```

93.2 Using the Custom Test Generator 译：93.2使用自定义测试生成器

如果您想为Java以外的语言生成测试，或者您对验证器构建Java测试的方式不满意，则可以注册您自己的实现。

`SingleTestGenerator`界面允许您注册自己的实施。 以下代码清单显示了`SingleTestGenerator`接口：

```
package org.springframework.cloud.contract.verifier.builder

import org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties
import org.springframework.cloud.contract.verifier.file.ContractMetadata
/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
interface SingleTestGenerator {

 /**
  * Creates contents of a single test class in which all test scenarios from
  * the contract metadata should be placed.
  *
  * @param properties - properties passed to the plugin
  * @param listOfFiles - list of parsed contracts with additional metadata
  * @param className - the name of the generated test class
  * @param classPackage - the name of the package in which the test class should be stored
  * @param includedDirectoryRelativePath - relative path to the included directory
  * @return contents of a single test class
  */
 String buildClass(ContractVerifierConfigProperties properties, Collection<ContractMetadata> listOfFiles,
       String className, String classPackage, String includedDirectoryRelativePath)

 /**
  * Extension that should be appended to the generated test class. E.g. {@code .java} or {@code .php}
  *
  * @param properties - properties passed to the plugin
  */
 String fileExtension(ContractVerifierConfigProperties properties)
}
```

同样，您必须提供`spring.factories`文件，如以下示例中所示：

```
org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/
com.example.MyGenerator
```

93.3 Using the Custom Stub Generator 译：93.3使用自定义存根发生器

如果您想为WireMock以外的存根服务器生成存根，则可以插入自己的`StubGenerator`接口实现。 以下代码清单显示了`StubGenerator`界面：

```
package org.springframework.cloud.contract.verifier.converter

import groovy.transform.CompileStatic
import org.springframework.cloud.contract.spec.Contract
import org.springframework.cloud.contract.verifier.file.ContractMetadata

/**
 * Converts contracts into their stub representation.
 *
 * @since 1.1.0
 */
@CompileStatic
interface StubGenerator {

 /**
  * Returns {@code true} if the converter can handle the file to convert it into a stub.
  */
 boolean canHandleFileName(String fileName)

 /**
  * Returns the collection of converted contracts into stubs. One contract can
  * result in multiple stubs.
  */
 Map<Contract, String> convertContents(String rootName, ContractMetadata content)

 /**
  * Returns the name of the converted stub file. If you have multiple contracts
  * in a single file then a prefix will be added to the generated file. If you
  * provide the {@link Contract#name} field then that field will override the
  * generated file name.
  *
  * Example: name of file with 2 contracts is {@code foo.groovy}, it will be
  * converted by the implementation to {@code foo.json}. The recursive file
  * converter will create two files {@code 0_foo.json} and {@code 1_foo.json}
  */
 String generateOutputFileNameForInput(String inputFileName)
}
```

同样，您必须提供`spring.factories`文件，如以下示例中所示：

```
# Stub converters
org.springframework.cloud.contract.verifier.converter.StubGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter
```

默认的实现是WireMock存根生成。

您可以提供多个存根生成器实现。 例如，从单个DSL，您可以生成WireMock存根和Pact文件。

93.4 Using the Custom Stub Runner译：93.4使用自定义存根转轮

如果您决定使用自定义存根生成，则还需要使用不同存根提供程序运行存根的自定义方式。

假设您使用 Moco构建存根，并且已经编写了存根生成器并将存根放入JAR文件中。

为了使Stub Runner知道如何运行存根，您必须定义一个自定义的HTTP存根服务器实现，该实现可能类似于以下示例：

```
package org.springframework.cloud.contract.stubrunner.provider.moco

import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import com.github.dreamhead.moco.runner.RunnerSetting
import groovy.util.logging.Slf4j
import org.springframework.cloud.contract.stubrunner.HttpServerStub
import org.springframework.util.SocketUtils

@Slf4j
class MocoHttpServerStub implements HttpServerStub {

 private boolean started
 private JsonRunner runner
 private int port

 @Override
 int port() {
  if (!isRunning()) {
   return -1
  }
  return port
 }

 @Override
 boolean isRunning() {
  return started
 }

 @Override
 HttpServerStub start() {
  return start(SocketUtils.findAvailableTcpPort())
 }

 @Override
 HttpServerStub start(int port) {
  this.port = port
  return this
 }

 @Override
 HttpServerStub stop() {
  if (!isRunning()) {
   return this
  }
  this.runner.stop()
  return this
 }

 @Override
 HttpServerStub registerMappings(Collection<File> stubFiles) {
  List<RunnerSetting> settings = stubFiles.findAll { it.name.endsWith("json") }
    .collect {
   log.info("Trying to parse [{}]", it.name)
   try {
    return RunnerSetting.aRunnerSetting().withStream(it.newInputStream()).build()
   } catch (Exception e) {
    log.warn("Exception occurred while trying to parse file [{}]", it.name, e)
    return null
   }
  }.findAll { it }
  this.runner = JsonRunner.newJsonRunnerWithSetting(settings,
    HttpArgs.httpArgs().withPort(this.port).build())
  this.runner.run()
  this.started = true
  return this
 }

 @Override
 String registeredMappings() {
  return ""
 }

 @Override
 boolean isAccepted(File file) {
  return file.name.endsWith(".json")
 }
}
```

然后，您可以将其注册到 `spring.factories` 文件中，如以下示例所示：

```
org.springframework.cloud.contract.stubrunner.HttpServerStub=\
org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub
```

现在你可以用Moco运行存根。

Important

如果你没有提供任何实现，那么使用默认的（WireMock）实现。 如果您提供多个，则使用列表中的第一个。

93.5 Using the Custom Stub Downloader 译：93.5使用自定义存根下载器

您可以通过创建 `StubDownloaderBuilder` 接口的实现来自定义存根的下载方式，如下例所示：

```
package com.example;

class CustomStubDownloaderBuilder implements StubDownloaderBuilder {

 @Override
 public StubDownloader build(final StubRunnerOptions stubRunnerOptions) {
  return new StubDownloader() {
   @Override
   public Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(
     StubConfiguration config) {
    File unpackedStubs = retrieveStubs();
    return new AbstractMap.SimpleEntry<>(
      new StubConfiguration(config.getGroupId(), config.getArtifactId(), version,
        config.getClassifier()), unpackedStubs);
   }

   File retrieveStubs() {
       // here goes your custom logic to provide a folder where all the stubs reside
   }
}
```

然后，您可以将其注册到 `spring.factories` 文件中，如以下示例所示：

```
# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\
com.example.CustomStubDownloaderBuilder
```

现在你可以选择一个文件夹来存放你的存根。

Important

如果您没有提供任何实现，则使用默认值（扫描类路径）。 如果您提供`stubsMode = StubRunnerProperties.StubsMode.LOCAL`或`, stubsMode = StubRunnerProperties.StubsMode.REMOTE`则将使用Aether实现。 如果您提供多个，则使用列表中的第一个。

93.6 Using the SCM Stub Downloader 译：93.6使用SCM存根下载器

每当 `repositoryRoot` 以SCM协议开始（目前我们仅支持 `git://` ），存根下载器将尝试克隆存储库并将其用作生成测试或存根的合同来源。

通过环境变量，系统属性，插件内部设置的属性或合约存储库配置，您可以调整下载器的行为。 您可以在下面找到属性列表

表93.1。 SCM Stub Downloader属性

属性的类型	财产的名称	描述
* `git.branch` （插件道具） * `stubrunner.properties.git.branch` （系统道具） * `STUBRUNNER_PROPERTIES_GIT_BRANCH` （env道具）	主	哪个分支要结帐
* `git.username` （插件道具） * `stubrunner.properties.git.username` （系统道具） * `STUBRUNNER_PROPERTIES_GIT_USERNAME` （env道具）		Git克隆用户名
* `git.password` （插件道具） * `stubrunner.properties.git.password` （系统道具） * `STUBRUNNER_PROPERTIES_GIT_PASSWORD` （env道具）		Git克隆密码
* `git.no-of-attempts` （插件道具） * `stubrunner.properties.git.no-of-attempts` （系统道具） * `STUBRUNNER_PROPERTIES_GIT_NO_OF_ATTEMPTS` （env道具）	10	尝试将提交压入 `origin`
* `git.wait-between-attempts` （插件道具） * `stubrunner.properties.git.wait-between-attempts` （系统道具） * `STUBRUNNER_PROPERTIES_GIT_WAIT_BETWEEN_ATTEMPTS` （env道具）	1000	在尝试将提交推送到 `origin` 之间等待的 `origin`

93.7 Using the Pact Stub Downloader 译：93.7使用 Pact Stub Downloader

每当`repositoryRoot`以Pact协议开始（以`pact://`）时，存根下载器将尝试从Pact Broker获取Pact合约定义。 无论在`pact://`之后设置什么，都将被解析为Pact Broker URL。

通过环境变量，系统属性，插件内部设置的属性或合约存储库配置，您可以调整下载器的行为。 您可以在下面找到属性列表

表93.2。 SCM Stub Downloader属性

物业的名称	默认	描述

* `pactbroker.host` （插件道具） * `stubrunner.properties.pactbroker.host` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_HOST` （env道具）	来自URL的主机传递给 `repositoryRoot`	Pact Broker的URL是什么？
* `pactbroker.port` （插件道具） * `stubrunner.properties.pactbroker.port` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_PORT` （env道具）	端口从URL传递到 `repositoryRoot`	Pact Broker的端口是什么？
* `pactbroker.protocol` （插件道具） * `stubrunner.properties.pactbroker.protocol` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_PROTOCOL` （env道具）	来自URL的协议传递给 `repositoryRoot`	Pact Broker的协议是什么？
* `pactbroker.tags` （插件道具） * `stubrunner.properties.pactbroker.tags` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_TAGS` （env道具）	存根的版本，或者 `latest` 如果版本为 `+`	应该使用哪些标签来提取存根
* `pactbroker.auth.scheme` （插件道具） * `stubrunner.properties.pactbroker.auth.scheme` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_SCHEME` （env道具）	`Basic`	应使用什么样的身份验证来连接到Pact Broker
* `pactbroker.auth.username` （插件道具） * `stubrunner.properties.pactbroker.auth.username` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_USERNAME` （env道具）	用户名传递给 `contractsRepositoryUsername` （maven）或 `contractRepository.username` （gradle）	用于连接到Pact Broker的用户名
* `pactbroker.auth.password` （插件道具） * `stubrunner.properties.pactbroker.auth.password` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_PASSWORD` （env道具）	密码传递给 `contractsRepositoryPassword` （maven）或 `contractRepository.password` （gradle）	用于连接到Pact Broker的密码
* `pactbroker.provider-name-with-group-id` （插件道具） * `stubrunner.properties.pactbroker.provider-name-with-group-id` （系统道具） * `STUBRUNNER_PROPERTIES_PACTBROKER_PROVIDER_NAME_WITH_GROUP_ID` （env道具）	假	当 `true` ，提供者名称将是 `groupId:artifactId` 的组合。如果使用 `false` ，则只需 `artifactId`

94. Spring Cloud Contract WireMock译：94. Spring Cloud Contract WireMock

Spring Cloud Contract WireMock模块允许您在Spring Boot应用程序中使用WireMock 。 查看samples了解更多详情。

如果您有一个使用Tomcat作为嵌入式服务器的Spring Boot应用程序（默认为 `spring-boot-starter-web` ），则可以将 `spring-cloud-starter-contract-stub-runner` 添加到类路径并添加 `@AutoConfigureWireMock` ，以便能够在测试中使用Wiremock。 Wiremock作为存根服务器运行，您可以使用Java API或通过静态JSON声明注册存根行为，作为测试的一部分。 以下代码显示了一个示例：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {
 // A service that calls out over HTTP
 @Autowired private Service service;

 // Using the WireMock APIs in the normal way:
 @Test
 public void contextLoads() throws Exception {
  // Stubbing WireMock
  stubFor(get(urlEqualTo("/resource"))
    .willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));
  // We're asserting if WireMock responded properly
  assertThat(this.service.go()).isEqualTo("Hello World!");
 }

}
```

要在不同的端口上启动存根服务器，请使用（例如） `@AutoConfigureWireMock(port=9999)` 。 对于随机端口，请使用值 `0` 。 末端服务器端口可以使用“wiremock.server.port”属性绑定到测试应用程序上下文中。 使用 `@AutoConfigureWireMock` 添加类型的豆 `WiremockConfiguration` 到您的测试应用程序上下文，它会具有同样的背景下，同样作为Spring集成测试方法和类之间进行缓存英寸

94.1 Registering Stubs Automatically译：94.1自动注册存根

如果使用 `@AutoConfigureWireMock` ，它会从文件系统或类路径中注册WireMock JSON存根（默认情况下，从 `file:src/test/resources/mappings` ）。 您可以使用注释中的 `stubs` 属性自定义位置，该属性可以是Ant样式的资源模式或目录。 在目录的情况下，附加 `*/.json` 。 以下代码显示了一个示例：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

 @Autowired
 private Service service;

 @Test
 public void contextLoads() throws Exception {
  assertThat(this.service.go()).isEqualTo("Hello World!");
 }

}
```

实际上，WireMock总是加载来自`src/test/resources/mappings`映射以及存根属性中的自定义位置。 要改变这种行为，你也可以指定一个文件根，如本文下一节所述。

94.2 Using Files to Specify the Stub Bodies译：94.2使用文件指定存根 主体

WireMock可以从类路径或文件系统上的文件中读取响应实体。 在这种情况下，您可以在JSON DSL中看到响应具有`bodyFileName`而不是（文字） `body` 。 这些文件是相对于根目录解析的（默认为`src/test/resources/__files`）。 要自定义此位置，可以将`@AutoConfigureWireMock`注释中的`files`属性设置为父目录的位置（换句话说， `__files`是子目录）。 您可以使用Spring资源记法来引用`file:…`或`classpath:…`位置。 通用网址不受支持。 可以给出一个值列表，在这种情况下，WireMock解析当需要查找响应主体时存在的第一个文件。

配置`files`根时，它也会影响存根的自动加载，因为它们来自称为“映射”的子目录中的根位置。 `files`的值对从`stubs`属性显式加载的存根没有影响。

94.3 Alternative: Using JUnit Rules译：94.3替代方法：使用JUnit规则

对于更传统的WireMock体验，您可以使用JUnit `@Rules`来启动和停止服务器。 为此，请使用`WireMockSpring`便利类获取`Options`实例，如以下示例所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

 // Start WireMock on some dynamic port
 // for some reason `dynamicPort()` is not working properly
 @ClassRule
 public static WireMockClassRule wiremock = new WireMockClassRule(
   WireMockSpring.options().dynamicPort());
 // A service that calls out over HTTP to localhost:${wiremock.port}
 @Autowired
 private Service service;

 // Using the WireMock APIs in the normal way:
 @Test
 public void contextLoads() throws Exception {
  // Stubbing WireMock
  wiremock.stubFor(get(urlEqualTo("/resource"))
    .willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));
  // We're asserting if WireMock responded properly
  assertThat(this.service.go()).isEqualTo("Hello World!");
 }

}
```

`@ClassRule`意味着在此类中的所有方法都已运行后服务器关闭。

94.4 Relaxed SSL Validation for Rest Template译：94.4休息模板的轻松SSL验证

WireMock允许您使用“https”URL协议来存储“安全”服务器。 如果您的应用程序想在集成测试中联系该存根服务器，它会发现SSL证书无效（自我安装证书的常见问题）。 最好的选择是经常重新配置客户端使用“http”。 如果这不是一个选项，你可以要求Spring配置一个忽略SSL验证错误的HTTP客户端（当然，这样做只适用于测试）。

要使这项工作最小化，您需要在您的应用中使用Spring Boot `RestTemplateBuilder` ，如以下示例所示：

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
 return builder.build();
}
```

您需要`RestTemplateBuilder`因为构建器会通过回调来初始化它，所以SSL验证可以在客户端进行设置。 如果您使用`@AutoConfigureWireMock`注释或存根运行程序，则会在您的测试中自动执行此操作。 如果您使用JUnit `@Rule`方法，则还需要添加`@AutoConfigureHttpClient`注释，如以下示例中所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest("app.baseUrl=https://localhost:6443")
@AutoConfigureHttpClient
public class WiremockHttpsServerApplicationTests {

 @ClassRule
 public static WireMockClassRule wiremock = new WireMockClassRule(
   WireMockSpring.options().httpsPort(6443));
...
}
```

如果您使用的是`spring-boot-starter-test` ，则在类路径中具有Apache HTTP客户端，并且它由`RestTemplateBuilder`选择并配置为忽略SSL错误。 如果您使用默认的`java.net`客户端，则不需要注释（但它不会造成任何伤害）。 目前没有其他客户支持，但可能会在未来的版本中添加。

要禁用自定义 `RestTemplateBuilder` ，请将 `wiremock.rest-template-ssl-enabled`属性设置为 `false` 。

94.5 WireMock and Spring MVC Mocks译：94.5 WireMock和 Spring MVC Mocks

Spring Cloud Contract提供了一个便捷类，可以将JSON WireMock存根加载到Spring `MockRestServiceServer` 。 以下代码显示了一个示例：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

 @Autowired
 private RestTemplate restTemplate;

 @Autowired
 private Service service;

 @Test
 public void contextLoads() throws Exception {
  // will read stubs classpath
  MockRestServiceServer server = WireMockRestServiceServer.with(this.restTemplate)
    .baseUrl("http://example.org").stubs("classpath:/stubs/resource.json")
    .build();
  // We're asserting if WireMock responded properly
  assertThat(this.service.go()).isEqualTo("Hello World");
  server.verify();
 }
}
```

`baseUrl` 值被预置为所有模拟调用，而 `stubs()` 方法将存根路径资源模式作为参数。 在前面的示例中，定义在 `/stubs/resource.json` 的存根被加载到模拟服务器中。 如果 `RestTemplate` 被要求访问 `http://example.org/` ，它会得到在该URL处声明的响应。 可以指定多个存根模式，每个存根可以是一个目录（用于所有“.json”的递归列表），固定文件名（如上例）或Ant样式模式。 JSON格式是普通的WireMock格式，你可以在WireMock website中阅读。

目前，Spring Cloud Contract Verifier支持Tomcat，Jetty和Undertow作为Spring Boot嵌入式服务器，Wiremock本身对特定版本的Jetty（当前为9.2）具有“原生”支持。 要使用本机Jetty，您需要添加本机Wiremock依赖项并排除Spring Boot容器（如果有）。

94.6 Customization of WireMock configuration译：94.6定制 WireMock配置

您可以注册一个 `org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer` 类型的bean，以便自定义WireMock配置（例如，添加自定义转换器）。 例：

```
  @Bean WireMockConfigurationCustomizer optionsCustomizer() {
   return new WireMockConfigurationCustomizer() {
    @Override public void customize(WireMockConfiguration options) {
// perform your customization here
    }
   };
  }
```

94.7 Generating Stubs using REST Docs译：94.7使用 REST文档生成存根

Spring REST Docs可用于为Spring MockMvc或 `WebTestClient` 或Rest Assured生成HTTP API的文档（例如Asciidoctor格式）。 在为您的API生成文档的同时，您还可以使用Spring Cloud Contract WireMock生成WireMock存根。 为此，请编写常规REST Docs测试用例，并使用 `@AutoConfigureRestDocs` 在REST Docs输出目录中自动生成存根。 以下代码显示了使用 `MockMvc` 的示例：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

 @Autowired
 private MockMvc mockMvc;

 @Test
 public void contextLoads() throws Exception {
  mockMvc.perform(get("/resource"))
    .andExpect(content().string("Hello World"))
    .andDo(document("resource"));
 }
}
```

此测试会在“target / snippets / stubs / resource.json”中生成一个WireMock存根。 它将所有GET请求匹配到“/ resource”路径。 与 `WebTestClient` （用于测试Spring WebFlux应用程序）相同的示例如下所示：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureWebTestClient
public class ApplicationTests {

 @Autowired
 private WebTestClient client;

 @Test
 public void contextLoads() throws Exception {
  client.get().uri("/resource").exchange()
    .expectBody(String.class).isEqualTo("Hello World")
     .consumeWith(document("resource"));
 }
}
```

如果没有任何额外的配置，这些测试将为HTTP方法和除“主机”和“内容长度”以外的所有标头创建一个带有请求匹配器的存根。 为了更准确地匹配请求（例如，匹配POST或PUT的主体），我们需要明确地创建一个请求匹配器。 这样做有两个影响：

- Creating a stub that matches only in the way you specify.

- Asserting that the request in the test case also matches the same conditions.

此功能的主要入口点是 `WireMockRestDocs.verify()` ，可用作 `document()` 便捷方法的替代方法，如以下示例所示：

```
import static org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

 @Autowired
 private MockMvc mockMvc;

 @Test
 public void contextLoads() throws Exception {
  mockMvc.perform(post("/resource")
              .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
    .andExpect(status().isOk())
    .andDo(verify().jsonPath("$.id")
                      .stub("resource"));
 }
}
```

该合同规定任何有“id”字段的有效POST都会收到本次测试中定义的响应。 您可以将呼叫链接到 `.jsonPath()` 以添加其他匹配器。 如果JSON Path不熟悉， JayWay documentation可以帮助您加快速度。 此测试的 `WebTestClient` 版本有一个类似的 `verify()` 静态帮助程序，您可以在同一位置插入。

而不是 `jsonPath` 和 `contentType` 便利方法，您也可以使用WireMock API来验证请求是否与创建的存根匹配，如下例所示：

```
@Test
public void contextLoads() throws Exception {
 mockMvc.perform(post("/resource")
             .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
   .andExpect(status().isOk())
   .andDo(verify()
     .wiremock(WireMock.post(
      urlPathEquals("/resource"))
      .withRequestBody(matchingJsonPath("$.id"))
                    .stub("post-resource"));
}
```

WireMock API非常丰富。 您可以通过正则表达式以及JSON路径匹配标题，查询参数和请求正文。 这些功能可用于创建具有更广泛参数的存根。 上面的例子生成一个存根类似于下面的例子：

-resource.json后。

```
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }]
  },
  "response" : {
    "status" : 200,
    "body" : "Hello World",
    "headers" : {
      "X-Application-Context" : "application:-1",
      "Content-Type" : "text/plain"
    }
  }
}
```

您可以使用 `wiremock()` 方法或 `jsonPath()` 和 `contentType()` 方法创建请求匹配器，但不能同时使用这两种方法。

在消费者方面，可以使本节前面生成的 `resource.json` 在类路径中可用（例如， publishing stubs as JARs ）。 之后，您可以使用WireMock以多种不同方式创建存根，包括使用 `@AutoConfigureWireMock(stubs="classpath:resource.json")` ，如本文档前面所述。

94.8 Generating Contracts by Using REST Docs译：94.8使用REST文档生成合同

您还可以使用Spring REST Docs生成Spring Cloud Contract DSL文件和文档。 如果您与Spring Cloud WireMock结合使用，您将获得合同和存根。

你为什么要使用这个功能？ 社区中的一些人询问了他们想转移到基于DSL的合同定义的情况，但他们已经有很多Spring MVC测试。 使用此功能可以生成您稍后可以修改并移动到文件夹（在您的配置中定义）的合约文件，以便插件找到它们。

你可能想知道为什么这个功能在WireMock模块中。 功能在那里，因为生成合同和存根都是有意义的。

考虑以下测试：

```
  this.mockMvc.perform(post("/foo")
     .accept(MediaType.APPLICATION_PDF)
     .accept(MediaType.APPLICATION_JSON)
     .contentType(MediaType.APPLICATION_JSON)
     .content("{\"foo\": 23, \"bar\" : \"baz\" }"))
    .andExpect(status().isOk())
    .andExpect(content().string("bar"))
    // first WireMock
    .andDo(WireMockRestDocs.verify()
      .jsonPath("$[?(@.foo >= 20)]")
      .jsonPath("$[?(@.bar in ['baz','bazz','bazzz'])]")
      .contentType(MediaType.valueOf("application/json"))
      .stub("shouldGrantABeerIfOldEnough"))
    // then Contract DSL documentation
    .andDo(document("index", SpringCloudContractRestDocs.dslContract()));
```

前面的测试会创建前一节中提到的存根，生成合同和文档文件。

该合同被称为`index.groovy`，可能看起来像下面的例子：

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url '/foo'
        body('''
            {"foo": 23 }
        ''')
        headers {
            header('''Accept''', '''application/json''')
            header('''Content-Type''', '''application/json''')
        }
    }
    response {
        status OK()
        body('''
        bar
        ''')
        headers {
            header('''Content-Type''', '''application/json;charset=UTF-8''')
            header('''Content-Length''', '''3''')
        }
        testMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}
```

生成的文档（在这种情况下在Asciidoc中格式化）包含格式化的合同。 这个文件的位置是`index/dsl-contract.adoc`。

95. Migrations译：95迁移

有关最新的迁移指南，请访问该项目的 wiki page 。

本部分涵盖从Spring Cloud合同验证工具的一个版本迁移到下一个版本。 它涵盖了以下版本升级途径：

95.1 1.0.x → 1.1.x译：95.1 1.0.x→1.1.x

本节介绍从版本1.0升级到版本1.1。

95.1.1 New structure of generated stubs译：95.1.1生成的存根的新结构

在`1.1.x`我们引入了对生成的存根结构的更改。 如果您一直使用`@AutoConfigureWireMock`表示法来使用类路径中的存根，则不再有效。 以下示例显示`@AutoConfigureWireMock`表示法如何用于工作：

```
@AutoConfigureWireMock(stubs = "classpath:/customer-stubs/mappings", port = 8084)
```

您必须将存根的位置更改为：`classpath:… /META-INF/groupId/artifactId/version/mappings`或使用新的基于类路径的`@AutoConfigureStubRunner`，如以下示例所示：

```
@AutoConfigureWireMock(stubs = "classpath:customer-stubs/META-INF/travel.components/customer-contract/1.0.2-SNAPSHOT/mappings/", port = 8084)
```

如果您不想使用`@AutoConfigureStubRunner`并且希望保留旧结构，请相应地设置插件任务。 以下示例适用于前面代码片段中介绍的结构。

Maven的。

```
<!-- start of pom.xml -->

<properties>
    <!-- we don't want the verifier to do a jar for us -->
    <spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>
</properties>

<!-- ... -->

<!-- You need to set up the assembly plugin -->
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <id>stub</id>
                    <phase>prepare-package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <inherited>false</inherited>
                    <configuration>
                        <attach>true</attach>
                        <descriptor>$../../../../src/assembly/stub.xml</descriptor>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
<!-- end of pom.xml -->

<!-- start of stub.xml-->

<assembly
 xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
 <id>stubs</id>
 <formats>
  <format>jar</format>
 </formats>
 <includeBaseDirectory>false</includeBaseDirectory>
 <fileSets>
  <fileSet>
   <directory>${project.build.directory}/snippets/stubs</directory>
   <outputDirectory>customer-stubs/mappings</outputDirectory>
   <includes>
    <include>**/*</include>
   </includes>
  </fileSet>
  <fileSet>
   <directory>$../../../../src/test/resources/contracts</directory>
   <outputDirectory>customer-stubs/contracts</outputDirectory>
   <includes>
    <include>**/*.groovy</include>
   </includes>
  </fileSet>
 </fileSets>
</assembly>

<!-- end of stub.xml-->
```

摇篮。

```
task copyStubs(type: Copy, dependsOn: 'generateWireMockClientStubs') {
//    Preserve directory structure from 1.0.X of spring-cloud-contract
    from "${project.buildDir}/resources/main/customer-stubs/META-INF/${project.group}/${project.name}/${project.version}"
    into "${project.buildDir}/resources/main/customer-stubs"
}
```

95.2 1.1.x → 1.2.x 译：95.2 1.1.x→1.2.x

本节介绍从版本1.1升级到版本1.2。

95.2.1 Custom `HttpServerStub` 译：95.2.1自定义 `HttpServerStub`

`HttpServerStub`包含一个不在版本1.1中的方法。 方法是`String registeredMappings()`如果您的类实现了`HttpServerStub`，则您现在必须实现`registeredMappings()`方法。 它应该返回`String`代表单个`HttpServerStub`可用的所有映射。

有关更多详细信息，请参阅 issue 355 。

95.2.2 New packages for generated tests 译：95.2.2生成测试的新软件包

设置生成的测试包名称的流程如下所示：

- Set `basePackageForTests`
- If `basePackageForTests` was not set, pick the package from `baseClassForTests`
- If `baseClassForTests` was not set, pick `packageWithBaseClasses`
- If nothing got set, pick the default value: `org.springframework.cloud.contract.verifier.tests`

详情请参阅 issue 260 。

95.2.3 New Methods in TemplateProcessor译：95.2.3 TemplateProcessor中的新方法

为了添加对 `fromRequest.path` 支持， `fromRequest.path` 将以下方法添加到 `TemplateProcessor` 接口中：

- `path()`
- `path(int index)`

详情请参阅 issue 388 。

95.2.4 RestAssured 3.0译：95.2.4 RestAssured 3.0

在所生成的测试课程中使用的Rest Assured碰到了 `3.0` 。 如果您手动设置Spring Cloud Contract和发行版的版本，则可能会看到以下异常：

```
Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile (default-testCompile) on project some-project: Compilation fail
[ERROR] /some/path/SomeClass.java:[4,39] package com.jayway.restassured.response does not exist
```

由于测试是使用旧版本的插件生成的，并且在测试执行期间发布了不兼容版本的版本（反之亦然），因此会发生此异常。

通过 issue 267完成

95.3 1.2.x → 2.0.x译：95.3 1.2.x–2.0.x

95.3.1 No Camel support译：95.3.1否骆驼支持

我们将在修复 issue之后才会添加Apache Camel支持

96. Links译：96.链接

使用Spring Cloud Contract Verifier时，以下链接可能会有所帮助：

- Spring Cloud Contract Github Repository
- Spring Cloud Contract Samples
- Spring Cloud Contract Documentation
- Accurest Legacy Documentation
- Spring Cloud Contract Stub Runner Documentation
- Spring Cloud Contract Stub Runner Messaging Documentation
- Spring Cloud Contract Gitter
- Spring Cloud Contract Maven Plugin
- Spring Cloud Contract WJUG Presentation by Marcin Grzejszczak

Part XIV. Spring Cloud Vault译：第十四部分。 Spring Cloud Vault

Spring Cloud Vault Config为分布式系统中的外部配置提供客户端支持。 通过HashiCorp's Vault，您可以在所有环境中管理应用程序的外部秘密属性。 保险柜可以管理静态和动态秘密，例如远程应用程序/资源的用户名/密码，并为外部服务提供凭据，例如MySQL，PostgreSQL，Apache Cassandra，MongoDB，Consul，AWS等。

97. Quick Start译：97.快速入门

先决条件

要开始使用Vault和本指南，您需要一个* NIX类操作系统，该操作系统提供：

- `wget`, `openssl` and `unzip`
- at least Java 7 and a properly configured `JAVA_HOME` environment variable

安装保险柜

```
$ src/test/bash/install_vault.sh
```

为Vault创建SSL证书

```
$ src/test/bash/create_certificates.sh
```

`create_certificates.sh`在`work/ca`和JKS `work/keystore.jks` `create_certificates.sh`创建证书。 如果要使用此快速入门指南运行Spring Cloud Vault，则需要将信任`spring.cloud.vault.ssl.trust-store`属性配置为`file:work/keystore.jks` 。

启动Vault服务器

```
$ src/test/bash/local_run_vault.sh
```

Vault开始使用`inmem`存储和`https`在`0.0.0.0:8200`进行`https` 。 启动时，保险库被密封并且未初始化。

如果您想运行测试，请保持Vault未初始化。 测试将初始化Vault并创建一个根令牌`00000000-0000-0000-0000-000000000000` 。

如果您想为您的应用程序使用Vault，或者尝试一下，那么您需要首先对其进行初始化。

```
$ export VAULT_ADDR="https://localhost:8200"
$ export VAULT_SKIP_VERIFY=true # Don't do this for production
$ vault init
```

你应该看到像这样的东西：

```
Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701
Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02
Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03
Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04
Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805
Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

Vault initialized with 5 keys and a key threshold of 3. Please
securely distribute the above keys. When the Vault is re-sealed,
restarted, or stopped, you must provide at least 3 of these keys
to unseal it again.

Vault does not store the master key. Without at least 3 keys,
your Vault will remain permanently sealed.
```

保险柜将初始化并返回一组未解除密钥和根令牌。 选择3把钥匙并开启保险库。 将Vault令牌存储在`VAULT_TOKEN`环境变量中。

```
$ vault unseal (Key 1)
$ vault unseal (Key 2)
$ vault unseal (Key 3)
$ export VAULT_TOKEN=(Root token)
# Required to run Spring Cloud Vault tests after manual initialization
$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"
```

Spring Cloud Vault访问不同的资源。 默认情况下，启用秘密后端，通过JSON端点访问秘密配置设置。

HTTP服务具有以下形式的资源：

```
/secret/{application}/{profile}
/secret/{application}
/secret/{defaultContext}/{profile}
/secret/{defaultContext}
```

“应用程序”在`SpringApplication`（即常规Spring Boot应用程序中通常是“应用程序”）中注入`spring.application.name`，“profile”是活动配置文件（或逗号分隔的属性列表）。 从保险柜中检索的属性将被“按原样”使用，而无需对属性名称做进一步的前缀。

98. Client Side Usage译：98.客户端使用

要在应用程序中使用这些功能，只需将其构建为取决于`spring-cloud-vault-config`的Spring Boot应用程序（例如，参见测试用例）。 示例Maven配置：

实例98.1。 的pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-vault-config</artifactId>
        <version>Finchley.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

<!-- repositories also needed for snapshots and milestones -->
```

然后你可以创建一个标准的Spring Boot应用程序，就像这个简单的HTTP服务器一样

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

运行时，它将从端口`8200`上的默认本地Vault服务器（如果它正在运行）中选取外部配置。 要修改启动行为，可以使用`bootstrap.properties`（如`application.properties`但是应用程序上下文的引导阶段）更改Vault服务器的位置，例如

实例98.2。 bootstrap.yml

```
spring.cloud.vault:
    host: localhost
    port: 8200
    scheme: https
    uri: https://localhost:8200
    connection-timeout: 5000
    read-timeout: 15000
    config:
        order: -10
```

- `host` sets the hostname of the Vault host. The host name will be used for SSL certificate validation
- `port` sets the Vault port
- `scheme` setting the scheme to `http` will use plain HTTP. Supported schemes are `http` and `https`.
- `uri` configure the Vault endpoint with an URI. Takes precedence over host/port/scheme configuration
- `connection-timeout` sets the connection timeout in milliseconds
- `read-timeout` sets the read timeout in milliseconds
- `config.order` sets the order for the property source

启用进一步的集成需要额外的依赖性和配置。 根据您设置Vault的方式，您可能需要其他配置，如SSL和authentication 。

如果应用程序导入 `spring-boot-starter-actuator`项目，则会通过 `/health`端点提供Vault服务器的状态。

可以通过属性 `health.vault.enabled` （默认为 `true` ）启用或禁用保险库健康指示器。

98.1 Authentication译：98.1认证

保险柜需要 authentication mechanism至 authorize client requests 。

Spring Cloud Vault支持多个 authentication mechanisms以使用Vault对应用程序进行身份验证。

为了快速入门，请使用由 Vault initialization打印的根令牌。

实施例98.3。 bootstrap.yml

```
spring.cloud.vault:
    token: 19aefa97-cccc-bbbb-aaaa-225940e63d76
```

仔细考虑你的安全要求。 如果您想快速开始使用Vault，静态令牌身份验证就没有问题，但静态令牌不再受到任何保护。 对意外方的任何披露都允许Vault使用相关的标记角色。

99. Authentication methods译：99.认证方法

不同的组织对安全和认证有不同的要求。 保险库通过运送多种身份验证方法来反映这种需求 Spring Cloud Vault支持令牌和AppId身份验证。

99.1 Token authentication译：99.1令牌认证

令牌是Vault中验证的核心方法。 令牌认证需要使用Bootstrap Application Context提供静态令牌。

令牌认证是默认的认证方法。 如果发现令牌，则意外方可以访问保险柜并可以访问预期客户的秘密。

例99.1。 bootstrap.yml

```
spring.cloud.vault:
    authentication: TOKEN
    token: 00000000-0000-0000-0000-000000000000
```

- `authentication` setting this value to `TOKEN` selects the Token authentication method
- `token` sets the static token to use

另见： Vault Documentation: Tokens

99.2 AppId authentication译：99.2AppId身份验证

保险柜支持AppId身份验证，其中包含两个难以猜测的令牌。 AppId默认为`spring.application.name`，它是静态配置的。 第二个标记是UserId，它是由应用程序确定的部分，通常与运行时环境相关。 IP地址，Mac地址或Docker容器名称都是很好的例子。 Spring Cloud Vault配置支持IP地址，Mac地址和静态UserId（例如通过系统属性提供）。 IP和Mac地址用十六进制编码的SHA256散列表示。

基于IP地址的UserId使用本地主机的IP地址。

例99.2。 bootstrap.yml使用SHA256 IP地址UserId's

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: IP_ADDRESS
```

- `authentication` setting this value to `APPID` selects the AppId authentication method
- `app-id-path` sets the path of the AppId mount to use
- `user-id` sets the UserId method. Possible values are `IP_ADDRESS`, `MAC_ADDRESS` or a class name implementing a custom `AppIdUserIdMechanism`

从命令行生成IP地址UserId的相应命令是：

```
$ echo -n 192.168.99.1 | sha256sum
```

包括`echo`会导致不同的散列值，因此请确保包含`-n`标志。

基于Mac地址的UserId从本地主机绑定设备获取其网络设备。 该配置还允许指定`network-interface`提示选择正确的设备。 `network-interface`的值是可选的，可以是接口名称或接口索引（基于0）。

例99.3。 bootstrap.yml使用SHA256 Mac-Address UserId's

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: MAC_ADDRESS
        network-interface: eth0
```

- `network-interface` sets network interface to obtain the physical address

从命令行生成IP地址UserId的相应命令是：

```
$ echo -n 0AFEDE1234AC | sha256sum
```

Mac地址被指定为大写且不含冒号。 包括`echo`会导致不同的散列值，因此请确保包含`-n`标志。

99.2.1 Custom UserId 译：99.2.1自定义 用户标识

UserId一代是一个开放的机制。 您可以将`spring.cloud.vault.app-id.user-id`设置为任何字符串，并将配置的值用作静态UserId。

更高级的方法可让您将`spring.cloud.vault.app-id.user-id`设置为类名。 该类必须位于类路径中，并且必须实现`org.springframework.cloud.vault.AppIdUserIdMechanism`接口和`createUserId`方法。 Spring Cloud Vault将通过每次使用AppId进行身份验证以获取令牌时调用`createUserId`来获取UserId。

例99.4。 bootstrap.yml

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: com.examlple.MyUserIdMechanism
```

例99.5。 MyUserIdMechanism.java

```
public class MyUserIdMechanism implements AppIdUserIdMechanism {

  @Override
  public String createUserId() {
    String userId = ...
    return userId;
  }
}
```

另见： Vault Documentation: Using the App ID auth backend

99.3 AppRole authentication 译：99.3 AppRole认证

AppRole用于机器验证，如已弃用（自Vault 0.6.1） Section 99.2, “AppId authentication” 。 AppRole身份验证由两个难以猜测（秘密）令牌组成：RoleId和SecretId。

Spring Vault支持各种AppRole场景（推/拉模式和包装）。

RoleId和可选的SecretId必须由配置提供，Spring Vault不会查找这些或创建自定义SecretId。

例99.6。 具有AppRole认证属性的bootstrap.yml

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
```

以下方案支持所需的配置细节：

表99.1。 组态

方法	角色ID	的SecretID	ROLENAME	代币
提供了RoleId / SecretId	提供	提供		
提供没有SecretId的RoleId	提供			
提供了RoleId，Pull SecretId	提供	提供	提供	提供
提供RoleId，提供SecretId		提供	提供	提供
全拉模式			提供	提供
包裹				提供
包装的RoleId提供了SecretId	提供			提供
提供了RoleId，包装SecretId		提供		提供

表99.2。 拉/推/包矩阵

角色ID	的SecretID	支持的
提供	提供	AOE ...
提供	拉	AOE ...
提供	包裹	AOE ...
提供	缺席	AOE ...
拉	提供	AOE ...
拉	拉	AOE ...
拉	包裹	â
拉	缺席	â
包裹	提供	AOE ...
包裹	拉	â
包裹	包裹	AOE ...
包裹	缺席	â

通过在boostrap上下文中提供配置的`AppRoleAuthentication` bean，您仍然可以使用推/拉/包模式的所有组合。 Spring Cloud Vault无法从配置属性中派生出所有可能的AppRole组合。

例99.7。 具有所有AppRole认证属性的bootstrap.yml

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
        secret-id: 1696536f-1976-73b1-b241-0b4213908d39
        role: my-role
        app-role-path: approle
```

- `role-id` sets the RoleId.
- `secret-id` sets the SecretId. SecretId can be omitted if AppRole is configured without requiring SecretId (See `bind_secret_id`).
- `role`: sets the AppRole name for pull mode.
- `app-role-path` sets the path of the approle authentication mount to use.

另见： Vault Documentation: Using the AppRole auth backend

99.4 AWS-EC2 authentication 译：99.4 AWS-EC2认证

aws-ec2 auth后端为AWS EC2实例提供安全的引入机制，允许自动检索Vault令牌。 与大多数Vault身份验证后端不同，此后端不需要首先部署或设置安全敏感凭据（令牌，用户名/密码，客户端证书等）。 相反，它将AWS视为受信任的第三方，并使用唯一代表每个EC2实例的加密签名的动态元数据信息。

例子99.8。 bootstrap.yml使用AWS-EC2认证

```
spring.cloud.vault:
    authentication: AWS_EC2
```

AWS-EC2认证默认启用nonce，遵循首先使用信任（TOFU）原则。 任何获得对PKCS＃7身份元数据访问权限的非预期方都可以针对Vault进行身份验证。

在第一次登录时，Spring Cloud Vault会生成一个随机数，存储在auth后端旁边的实例Id中。 重新认证需要发送相同的随机数。 任何其他方都没有现成现象，并且可以在保险库中发出警报以进一步调查。

随机数保存在内存中，在应用程序重新启动时丢失。 您可以使用`spring.cloud.vault.aws-ec2.nonce`配置静态随机数。

AWS-EC2认证角色是可选的，并且是AMI的默认设置。 您可以通过设置`spring.cloud.vault.aws-ec2.role`属性来配置身份验证角色。

例99.9。 具有配置角色的bootstrap.yml

```
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
```

例99.10。 具有所有AWS EC2认证属性的bootstrap.yml

```
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
        aws-ec2-path: aws-ec2
        identity-document: http://...
        nonce: my-static-nonce
```

- `authentication` setting this value to `AWS_EC2` selects the AWS EC2 authentication method
- `role` sets the name of the role against which the login is being attempted.
- `aws-ec2-path` sets the path of the AWS EC2 mount to use
- `identity-document` sets URL of the PKCS#7 AWS EC2 identity document
- `nonce` used for AWS-EC2 authentication. An empty nonce defaults to nonce generation

另见： Vault Documentation: Using the aws auth backend

99.5 AWS-IAM authentication译：99.5 AWS-IAM认证

aws后端为AWS IAM角色提供安全的身份验证机制，允许基于正在运行的应用程序的当前IAM角色对Vault进行自动身份验证。 与大多数Vault身份验证后端不同，此后端不需要首先部署或设置安全敏感凭据（令牌，用户名/密码，客户端证书等）。 相反，它会将AWS视为受信任的第三方，并使用由调用方签名的4条信息与他们的IAM凭证来验证调用方确实正在使用该IAM角色。

自动计算应用程序运行的当前IAM角色。 如果您在AWS ECS上运行应用程序，则应用程序将使用分配给正在运行的容器的ECS任务的IAM角色。 如果您在EC2实例的顶部裸体运行应用程序，则使用的IAM角色将是分配给EC2实例的角色。

使用AWS-IAM身份验证时，您必须在Vault中创建一个角色并将其分配给您的IAM角色。 一个空的`role`默认为当前IAM角色的友好名称。

例99.11。 bootstrap.yml和所需的AWS-IAM身份验证属性

```
spring.cloud.vault:
    authentication: AWS_IAM
```

例99.12。 bootstrap.yml与所有AWS-IAM身份验证属性

```
spring.cloud.vault:
    authentication: AWS_IAM
    aws-iam:
        role: my-dev-role
        aws-path: aws
        server-id: some.server.name
```

- `role` sets the name of the role against which the login is being attempted. This should be bound to your IAM role. If one is not supplied then the friendly name of the current IAM user will be used as the vault role.
- `aws-path` sets the path of the AWS mount to use
- `server-id` sets the value to use for the `X-Vault-AWS-IAM-Server-ID` header preventing certain types of replay attacks.

AWS-IAM需要AWS Java SDK依赖项（`com.amazonaws:aws-java-sdk-core`），因为身份验证实施将AWS SDK类型用于凭据和请求签名。

另见： Vault Documentation: Using the aws auth backend

99.6 TLS certificate authentication译：99.6 TLS证书认证

`cert` auth后端允许使用由CA签名或自签名的SSL / TLS客户端证书进行身份验证。

要启用 `cert`身份验证，您需要：

1. Use SSL, see Chapter 105, *Vault Client SSL configuration*
2. Configure a Java `Keystore` that contains the client certificate and the private key
3. Set the `spring.cloud.vault.authentication` to `CERT`

例99.13。 bootstrap.yml

```
spring.cloud.vault:
    authentication: CERT
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: changeit
        cert-auth-path: cert
```

另见： Vault Documentation: Using the Cert auth backend

99.7 Cubbyhole authentication译：99.7 Cubbyhole认证

Cubbyhole身份验证使用Vault基元来提供安全的身份验证工作流程。 Cubbyhole身份验证使用令牌作为主要登录方法。 短暂标记用于从Vault的Cubbyhole秘密后端获取第二个登录VaultToken。 登录令牌通常较长寿命并用于与Vault进行交互。 登录令牌将从存储在`/cubbyhole/response`的包装响应中`/cubbyhole/response`。

创建一个包装的标记

响应使用标记创建的包装需要Vault 0.6.0或更高版本。

例99.14。 创建和存储令牌

```
$ vault token-create -wrap-ttl="10m"
Key                           Value
---                           -----
wrapping_token:               397ccb93-ff6c-b17b-9389-380b01ca2645
wrapping_token_ttl:           0h10m0s
wrapping_token_creation_time: 2016-09-18 20:29:48.652957077 +0200 CEST
wrapped_accessor:             46b6aebb-187f-932a-26d7-4f3d86a68319
```

例99.15。 bootstrap.yml

```
spring.cloud.vault:
    authentication: CUBBYHOLE
    token: 397ccb93-ff6c-b17b-9389-380b01ca2645
```

也可以看看：

- Vault Documentation: Tokens
- Vault Documentation: Cubbyhole Secret Backend
- Vault Documentation: Response Wrapping

99.8 Kubernetes authentication译：99.8 Kubernetes身份验证

Kubernetes身份验证机制（自Vault 0.8.3起）允许使用Kubernetes服务帐户令牌对Vault进行身份验证。 身份验证是基于角色的，角色绑定到服务帐户名称和名称空间。

包含适用于pod服务帐户的JWT令牌的文件将自动安装在`/var/run/secrets/kubernetes.io/serviceaccount/token`。

例99.16。 具有所有Kubernetes身份验证属性的bootstrap.yml

```
spring.cloud.vault:
    authentication: KUBERNETES
    kubernetes:
        role: my-dev-role
        service-account-token-file: /var/run/secrets/kubernetes.io/serviceaccount/token
```

- `role` sets the Role.
- `service-account-token-file` sets the location of the file containing the Kubernetes Service Account Token. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/token`.

也可以看看：

- Vault Documentation: Kubernetes
- Kubernetes Documentation: Configure Service Accounts for Pods

100. Secret Backends译：100.秘密后端

100.1 Generic Backend译：100.1通用后端

Spring Cloud Vault在基本级别支持通用的秘密后端。 通用的秘密后端允许存储任意值作为键值存储。 单个上下文可以存储一个或多个键值元组。 上下文可以分层次组织。 Spring Cloud Vault允许将应用程序名称和默认上下文名称（`application`）与活动配置文件结合使用。

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

应用程序名称由属性决定：

- `spring.cloud.vault.generic.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

可以通过将其路径添加到应用程序名称中，并使用逗号分隔，从通用后端的其他上下文中获取秘密。 例如，给定应用程序名称`usefulapp,mysql1,projectx/aws`，将使用以下每个文件夹：

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault将所有活动配置文件添加到可能的上下文路径列表中。 没有活动的配置文件将跳过访问配置文件名称的上下文。

属性会像存储一样暴露（即没有额外的前缀）。

```
spring.cloud.vault:
    generic:
        enabled: true
        backend: secret
        profile-separator: '/'
        default-context: application
        application-name: my-app
```

- `enabled` setting this value to `false` disables the secret backend config usage
- `backend` sets the path of the secret mount to use
- `default-context` sets the context name used by all applications
- `application-name` overrides the application name for use in the generic backend
- `profile-separator` separates the profile name from the context in property sources with profiles

键值秘密后端可以以版本化（v2）和非版本化（v1）模式运行。 根据操作模式，访问秘密需要使用不同的API。 请务必使`generic`非版本键值后端秘密后端使用和`kv`的版本键值后端秘密后端使用。

另见： Vault Documentation: Using the KV Secrets Engine - Version 1 (generic secret backend)

100.2 Versioned Key-Value Backend 译：100.2版本化键值后端

Spring Cloud Vault支持版本化的Key-Value秘密后端。 键值后端允许存储任意值作为键值存储。 单个上下文可以存储一个或多个键值元组。 上下文可以分层次组织。 Spring Cloud Vault允许将应用程序名称和默认上下文名称（ `application` ）与活动配置文件结合使用。

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

应用程序名称由属性决定：

- `spring.cloud.vault.kv.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

可以通过将键盘路径添加到应用程序名称中，并使用逗号分隔，从键值后端内的其他上下文中获取秘密。 例如，给定应用程序名称`usefulapp,mysql1,projectx/aws` ，将使用以下每个文件夹：

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault将所有活动配置文件添加到可能的上下文路径列表中。 没有活动的配置文件将跳过访问配置文件名称的上下文。

属性会像存储一样暴露（即没有额外的前缀）。

Spring Cloud Vault在安装路径和实际上下文路径之间添加 `data/` 上下文。

```
spring.cloud.vault:
    kv:
        enabled: true
        backend: secret
        profile-separator: '/'
        default-context: application
        application-name: my-app
```

- `enabled` setting this value to `false` disables the secret backend config usage
- `backend` sets the path of the secret mount to use
- `default-context` sets the context name used by all applications
- `application-name` overrides the application name for use in the generic backend
- `profile-separator` separates the profile name from the context in property sources with profiles

键值秘密后端可以以版本化（v2）和非版本化（v1）模式运行。 根据操作模式，访问秘密需要使用不同的API。 请务必使`generic`非版本键值后端秘密后端使用和`kv`的版本键值后端秘密后端使用。

另见： Vault Documentation: Using the KV Secrets Engine - Version 2 (versioned key-value backend)

100.3 Consul 译：100.3领事

Spring Cloud Vault可以获得HashiCorp领事的凭证。 Consul集成需要`spring-cloud-vault-config-consul`依赖项。

例100.1。 的pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-consul</artifactId>
        <version>Finchley.RELEASE</version>
    </dependency>
</dependencies>
```

可以通过设置 `spring.cloud.vault.consul.enabled=true` （默认 `false` ）并提供角色名称 `spring.cloud.vault.consul.role=…` 来启用集成。

获取的令牌存储在 `spring.cloud.consul.token` 因此使用Spring Cloud Consul可以在没有进一步配置的情况下获取生成的凭证。 您可以通过设置 `spring.cloud.vault.consul.token-property` 来配置属性名称。

```
spring.cloud.vault:
    consul:
        enabled: true
        role: readonly
        backend: consul
        token-property: spring.cloud.consul.token
```

- `enabled` setting this value to `true` enables the Consul backend config usage
- `role` sets the role name of the Consul role definition
- `backend` sets the path of the Consul mount to use
- `token-property` sets the property name in which the Consul ACL token is stored

另见： Vault Documentation: Setting up Consul with Vault

100.4 RabbitMQ 译：100.4 RabbitMQ

Spring Cloud Vault可以获取RabbitMQ的凭据。

RabbitMQ集成需要 `spring-cloud-vault-config-rabbitmq` 依赖。

例100.2。 的pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-rabbitmq</artifactId>
        <version>Finchley.RELEASE</version>
    </dependency>
</dependencies>
```

可以通过设置 `spring.cloud.vault.rabbitmq.enabled=true` （默认 `false` ）并提供角色名称 `spring.cloud.vault.rabbitmq.role=…` 来启用集成。

用户名和密码存储在 `spring.rabbitmq.username` 和 `spring.rabbitmq.password` 因此使用Spring Boot将无需进一步配置即可获取生成的凭证。 您可以通过设置 `spring.cloud.vault.rabbitmq.username-property` 和 `spring.cloud.vault.rabbitmq.password-property` 来配置属性名称。

```
spring.cloud.vault:
    rabbitmq:
        enabled: true
        role: readonly
        backend: rabbitmq
        username-property: spring.rabbitmq.username
        password-property: spring.rabbitmq.password
```

- `enabled` setting this value to `true` enables the RabbitMQ backend config usage
- `role` sets the role name of the RabbitMQ role definition
- `backend` sets the path of the RabbitMQ mount to use
- `username-property` sets the property name in which the RabbitMQ username is stored
- `password-property` sets the property name in which the RabbitMQ password is stored

另见： Vault Documentation: Setting up RabbitMQ with Vault

100.5 AWS 译：100.5 AWS

Spring Cloud Vault可以获取AWS的凭据。

AWS集成需要 `spring-cloud-vault-config-aws` 依赖项。

例100.3。 的pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-aws</artifactId>
        <version>Finchley.RELEASE</version>
    </dependency>
</dependencies>
```

可以通过设置 `spring.cloud.vault.aws=true` （默认 `false` ）并提供角色名称 `spring.cloud.vault.aws.role=…` 来启用集成。

访问密钥和密钥存储在 `cloud.aws.credentials.accessKey` 和 `cloud.aws.credentials.secretKey` 因此使用Spring Cloud AWS将在没有进一步配置的情况下提取生成的凭证。 您可以通过设置 `spring.cloud.vault.aws.access-key-property` 和 `spring.cloud.vault.aws.secret-key-property` 来配置属性名称。

```
spring.cloud.vault:
    aws:
        enabled: true
        role: readonly
        backend: aws
        access-key-property: cloud.aws.credentials.accessKey
        secret-key-property: cloud.aws.credentials.secretKey
```

- `enabled` setting this value to `true` enables the AWS backend config usage
- `role` sets the role name of the AWS role definition
- `backend` sets the path of the AWS mount to use
- `access-key-property` sets the property name in which the AWS access key is stored

- `secret-key-property` sets the property name in which the AWS secret key is stored

另见： Vault Documentation: Setting up AWS with Vault

101. Database backends译：101.数据库后端

保险柜支持多个数据库秘密后端，以根据配置的角色动态生成数据库凭证。 这意味着需要访问数据库的服务不再需要配置凭据：他们可以从保险柜请求它们，并使用Vault的租赁机制更容易地滚动密钥。

Spring Cloud Vault与这些后端集成在一起：

- Section 101.1, “Database”
- Section 101.2, “Apache Cassandra”
- Section 101.3, “MongoDB”
- Section 101.4, “MySQL”
- Section 101.5, “PostgreSQL”

使用数据库秘密后端需要在配置和 `spring-cloud-vault-config-databases` 依赖项中启用后端。

Vault自0.7.1开始发布，并有专用的 `database` 秘密后端，允许通过插件进行数据库集成。 您可以通过使用通用数据库后端来使用该特定后端。 确保指定适当的后端路径，例如 `spring.cloud.vault.mysql.role.backend=database` 。

例101.1。 的pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-databases</artifactId>
        <version>Finchley.RELEASE</version>
    </dependency>
</dependencies>
```

启用多个JDBC兼容数据库将生成凭证并将其默认存储在相同的属性键中，因此JDBC秘密的属性名称需要单独配置。

101.1 Database译：101.数据库

Spring Cloud Vault可以获取https://www.vaultproject.io/api/secret/databases/index.html上列出的任何数据库的凭据。 可以通过设置 `spring.cloud.vault.database.enabled=true` （默认 `false` ）并提供角色名称 `spring.cloud.vault.database.role=…` 来启用集成。

虽然数据库后端是通用数据库，但 `spring.cloud.vault.database` 专门针对JDBC数据库。 用户名和密码存储在 `spring.datasource.username` 和 `spring.datasource.password` 因此使用Spring Boot将无需进一步配置即可为您的 `DataSource` 选取生成的凭证。 您可以通过设置 `spring.cloud.vault.database.username-property` 和 `spring.cloud.vault.database.password-property` 来配置属性名称。

```
spring.cloud.vault:
    database:
        enabled: true
        role: readonly
        backend: database
        username-property: spring.datasource.username
        password-property: spring.datasource.username
```

- `enabled` setting this value to `true` enables the Database backend config usage
- `role` sets the role name of the Database role definition
- `backend` sets the path of the Database mount to use
- `username-property` sets the property name in which the Database username is stored
- `password-property` sets the property name in which the Database password is stored

另见： Vault Documentation: Database Secrets backend

101.2 Apache Cassandra译：101.2 Apache Cassandra

该 `cassandra` 后端已被弃用，在跳马0.7.1，建议使用 `database` 后端和安装作为 `cassandra` 。

Spring Cloud Vault可以获得Apache Cassandra的凭据。 可以通过设置 `spring.cloud.vault.cassandra.enabled=true` （默认 `false` ）并提供角色名称 `spring.cloud.vault.cassandra.role=…` 来启用集成。

用户名和密码存储在 `spring.data.cassandra.username` 和 `spring.data.cassandra.password` 因此使用Spring Boot将在没有进一步配置的情况下选择生成的凭证。 您可以通过设置 `spring.cloud.vault.cassandra.username-property` 和 `spring.cloud.vault.cassandra.password-property` 来配置属性名称。

```
spring.cloud.vault:
    cassandra:
        enabled: true
        role: readonly
        backend: cassandra
        username-property: spring.data.cassandra.username
        password-property: spring.data.cassandra.username
```

- `enabled` setting this value to `true` enables the Cassandra backend config usage
- `role` sets the role name of the Cassandra role definition
- `backend` sets the path of the Cassandra mount to use
- `username-property` sets the property name in which the Cassandra username is stored
- `password-property` sets the property name in which the Cassandra password is stored

另见： Vault Documentation: Setting up Apache Cassandra with Vault

101.3 MongoDB译：101.3 MongoDB

> `mongodb`后端已在Vault 0.7.1中弃用，建议使用`database`后端并将其挂载为`mongodb`。

Spring Cloud Vault可以获取MongoDB的凭据。 可以通过设置`spring.cloud.vault.mongodb.enabled=true`（默认`false`）并提供角色名称`spring.cloud.vault.mongodb.role=…`来启用集成。

用户名和密码存储在`spring.data.mongodb.username`和`spring.data.mongodb.password`因此使用Spring Boot将无需进一步配置即可获取生成的凭据。 您可以通过设置`spring.cloud.vault.mongodb.username-property`和`spring.cloud.vault.mongodb.password-property`来配置属性名称。

```
spring.cloud.vault:
    mongodb:
        enabled: true
        role: readonly
        backend: mongodb
        username-property: spring.data.mongodb.username
        password-property: spring.data.mongodb.password
```

- `enabled` setting this value to `true` enables the MongodB backend config usage
- `role` sets the role name of the MongoDB role definition
- `backend` sets the path of the MongoDB mount to use
- `username-property` sets the property name in which the MongoDB username is stored
- `password-property` sets the property name in which the MongoDB password is stored

另见： Vault Documentation: Setting up MongoDB with Vault

101.4 MySQL译：101.4 MySQL

> `mysql`后端已在Vault 0.7.1中弃用，建议使用`database`后端并将其挂载为`mysql`。 未来版本中将删除`spring.cloud.vault.mysql`配置。

Spring Cloud Vault可以获取MySQL的凭据。 可以通过设置`spring.cloud.vault.mysql.enabled=true`（默认`false`）并提供角色名称`spring.cloud.vault.mysql.role=…`来启用集成。

用户名和密码存储在`spring.datasource.username`和`spring.datasource.password`因此使用Spring Boot将无需进一步配置即可获取生成的凭据。 您可以通过设置`spring.cloud.vault.mysql.username-property`和`spring.cloud.vault.mysql.password-property`来配置属性名称。

```
spring.cloud.vault:
    mysql:
        enabled: true
        role: readonly
        backend: mysql
        username-property: spring.datasource.username
        password-property: spring.datasource.username
```

- `enabled` setting this value to `true` enables the MySQL backend config usage
- `role` sets the role name of the MySQL role definition
- `backend` sets the path of the MySQL mount to use
- `username-property` sets the property name in which the MySQL username is stored
- `password-property` sets the property name in which the MySQL password is stored

另见： Vault Documentation: Setting up MySQL with Vault

101.5 PostgreSQL译：101.5 PostgreSQL

> 该`postgresql`后端已被弃用，在跳马0.7.1，建议使用`database`后端和安装作为`postgresql`。 未来版本中将删除`spring.cloud.vault.postgresql`配置。

Spring Cloud Vault可以获取PostgreSQL的凭证。 可以通过设置`spring.cloud.vault.postgresql.enabled=true`（默认`false`）并提供角色名称`spring.cloud.vault.postgresql.role=…`来启用集成。

用户名和密码存储在`spring.datasource.username`和`spring.datasource.password`因此使用Spring Boot将无需进一步配置即可获取生成的凭据。 您可以通过设置`spring.cloud.vault.postgresql.username-property`和`spring.cloud.vault.postgresql.password-property`来配置属性名称。

```
spring.cloud.vault:
    postgresql:
        enabled: true
        role: readonly
        backend: postgresql
        username-property: spring.datasource.username
        password-property: spring.datasource.username
```

- `enabled` setting this value to `true` enables the PostgreSQL backend config usage
- `role` sets the role name of the PostgreSQL role definition
- `backend` sets the path of the PostgreSQL mount to use
- `username-property` sets the property name in which the PostgreSQL username is stored
- `password-property` sets the property name in which the PostgreSQL password is stored

另见： Vault Documentation: Setting up PostgreSQL with Vault

102. Configure `PropertySourceLocator` behavior译：102配置`PropertySourceLocator`行为

Spring Cloud Vault使用基于属性的配置为通用和发现的秘密后端创建`PropertySource`。

发现的后端提供`VaultSecretBackendDescriptor` bean来描述使用秘密后端的配置状态`PropertySource`。 需要`SecretBackendMetadataFactory`才能创建包含路

径，名称和属性转换配置的`SecretBackendMetadata`对象。

`SecretBackendMetadata`用于支持特定的`PropertySource`。

您可以注册任意数量的实现`VaultConfigurer`定制的bean。 如果Spring Cloud Vault发现至少一个`VaultConfigurer` bean，则默认的通用和发现的后端注册将被禁用。 但是，您可以使用`SecretBackendConfigurer.registerDefaultGenericSecretBackends()`和`SecretBackendConfigurer.registerDefaultDiscoveredSecretBackends()`启用默认注册。

```
public class CustomizationBean implements VaultConfigurer {

    @Override
    public void addSecretBackends(SecretBackendConfigurer configurer) {

        configurer.add("secret/my-application");

        configurer.registerDefaultGenericSecretBackends(false);
        configurer.registerDefaultDiscoveredSecretBackends(true);
    }
}
```


所有定制都需要在引导程序环境中进行。 在您的应用程序`META-INF/spring.factories`您的配置类添加到`META-INF/spring.factories`
`org.springframework.cloud.bootstrap.BootstrapConfiguration`。

103. Service Registry Configuration 译：103.服务注册表配置

通过设置spring.cloud.vault.discovery.enabled = true（默认为`false`），您可以使用`DiscoveryClient`（例如Spring Cloud Consul）来查找Vault服务器。 最终的结果是，您的应用程序需要具有适当发现配置的bootstrap.yml（或环境变量）。 好处是只要发现服务是固定点，Vault就可以改变其坐标。 默认服务ID是`vault`但您可以使用`spring.cloud.vault.discovery.serviceId`更改客户端上的服务ID。

发现客户端实现都支持某种类型的元数据映射（例如，对于Eureka，我们有eureka.instance.metadataMap）。 可能需要在其服务注册元数据中配置服务的一些其他属性，以便客户端可以正确连接。 不提供有关传输层安全性的服务注册表需要提供`scheme`元数据条目，以设置为`https`或`http`。 如果没有配置方案并且该服务未作为安全服务公开，则配置默认为`spring.cloud.vault.scheme`，即`https`如果未设置。

```
spring.cloud.vault.discovery:
    enabled: true
    service-id: my-vault-service
```

104. Vault Client Fail Fast 译：104.保管库客户端快速失败

在某些情况下，如果服务无法连接到Vault服务器，则可能需要启动服务时失败。 如果这是所需的行为，请设置引导程序配置属性`spring.cloud.vault.fail-fast=true`，客户端将暂停并出现异常。

```
spring.cloud.vault:
    fail-fast: true
```

105. Vault Client SSL configuration 译：105. Vault客户端SSL配置

可以通过设置各种属性以声明方式配置SSL。 您可以设置`javax.net.ssl.trustStore`以配置JVM范围的SSL设置，或者设置`spring.cloud.vault.ssl.trust-store`以仅为Spring Cloud Vault配置设置SSL设置。

```
spring.cloud.vault:
    ssl:
        trust-store: classpath:keystore.jks
        trust-store-password: changeit
```

- `trust-store` sets the resource for the trust-store. SSL-secured Vault communication will validate the Vault SSL certificate with the specified trust-store.
- `trust-store-password` sets the trust-store password

请注意，配置`spring.cloud.vault.ssl.*`只能在Apache Http组件或OkHttp客户端位于类路径上时应用。

106. Lease lifecycle management (renewal and revocation) 译：106.租赁生命周期管理（更新和撤销）

凭借每一个秘密，Vault都会创建一个租约：包含持续时间，可更新性等信息的元数据。

保险柜承诺数据在给定的时间内有效，或生存时间（TTL）。 一旦租约到期，保险柜可以撤销数据，而秘密的使用者不能再确定它是否有效。

Spring Cloud Vault在创建登录令牌和秘密之后维护租赁生命周期。 也就是说，与租约相关的登录令牌和秘密将在租约到期之前计划续约，直到终端到期。 应用程序关闭会取消获得的登录令牌和可续租租赁。

秘密服务和数据库后端（例如MongoDB或MySQL）通常会生成可再生租约，因此在应用程序关闭时将禁用生成的凭据。

静态令牌不会被更新或撤销。

租赁更新和撤销默认启用，可通过将`spring.cloud.vault.config.lifecycle.enabled`设置为`spring.cloud.vault.config.lifecycle.enabled`来`false`。 这是不推荐的，因为租约可能会过期并且Spring Cloud Vault无法再使用生成的凭据访问Vault或服务，并且在应用程序关闭后有效凭证保持活动状态。

```
spring.cloud.vault:
    config.lifecycle.enabled: true
```

另见： Vault Documentation: Lease, Renew, and Revoke

Part XV. Spring Cloud Gateway 译:第十五部分。 Spring云网关

Finchley.RELEASE

该项目提供了一个构建在Spring Ecosystem之上的API网关，包括：Spring 5，Spring Boot 2和Project Reactor。 Spring Cloud Gateway旨在提供一种简单而有效的途径来发送API，并为他们提供横切关注点，例如：安全性，监控/指标和弹性。

107. How to Include Spring Cloud Gateway 译: 107.如何包含Spring云网关

要在项目中加入Spring Cloud Gateway，请使用组`org.springframework.cloud`和工件ID `spring-cloud-starter-gateway`的启动器。 有关使用当前Spring Cloud Release Train设置构建系统的详细信息，请参阅Spring Cloud Project page 。

如果您包含启动器，但出于某种原因，您不希望网关启用，请设置 `spring.cloud.gateway.enabled=false` 。

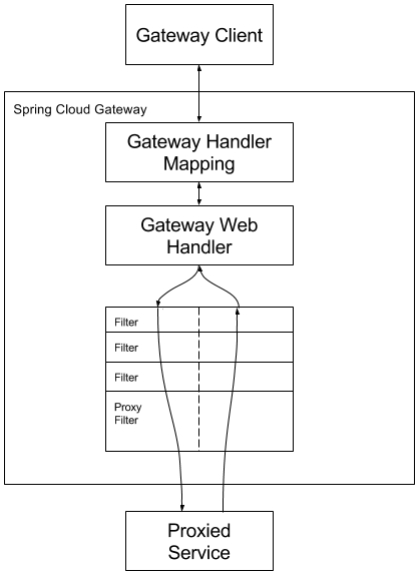
Important

Spring Cloud Gateway需要Spring Boot和Spring Webflux提供的Netty运行时。 它不适用于传统的Servlet容器或构建为WAR。

108. Glossary 译: 108.术语表

- **Route**: Route the basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates and a collection of filters. A route is matched if aggregate predicate is true.
- **Predicate**: This is a Java 8 Function Predicate. The input type is a Spring Framework `ServerWebExchange`. This allows developers to match on anything from the HTTP request, such as headers or parameters.
- **Filter**: These are instances Spring Framework `GatewayFilter` constructed in with a specific factory. Here, requests and responses can be modified before or after sending the downstream request.

109. How It Works 译: 109.它是如何工作的



客户向Spring Cloud Gateway发出请求。 如果网关处理程序映射确定请求与路由匹配，则将其发送到网关Web处理程序。 此处理程序运行通过特定于请求的筛选器链发送请求。 过滤器之间用虚线分开的原因是过滤器可能会在发送代理请求之前或之后执行逻辑。 所有“预”过滤器逻辑被执行，然后代理请求被做出。 代理请求完成后，执行“后”过滤器逻辑。

在没有端口的路由中定义的URI将分别为HTTP和HTTPS URI获取默认端口80和443。

110. Route Predicate Factories 译: 110.路线谓词工厂

Spring Cloud Gateway将路由作为Spring WebFlux `HandlerMapping`基础架构的一部分进行匹配。 Spring Cloud Gateway包含许多内置的Route Predicate Factory。 所有这些谓词匹配HTTP请求的不同属性。 多重路线谓词工厂可以合并并通过逻辑`and`合并。

110.1 After Route Predicate Factory 译: 110.1路由谓词工厂后

After Route Predicate Factory采用一个参数 - 日期时间。 此谓词匹配在当前日期时间之后发生的请求。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: http://example.org
        predicates:
        - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

此路线适用于2017年1月20日17:42山地时间（丹佛）之后的任何请求。

110.2 Before Route Predicate Factory 译：110.2在路由谓词工厂之前

Before Route Predicate Factory需要一个参数，一个日期时间。 此谓词匹配在当前日期时间之前发生的请求。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: before_route
        uri: http://example.org
        predicates:
        - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
```

此路线符合2017年1月20日17:42山地时间（丹佛）之前的任何请求。

110.3 Between Route Predicate Factory 译：110.3路由谓词工厂

Between Route Predicate Factory有两个参数，datetime1和datetime2。 此谓词匹配在datetime1之后和datetime2之前发生的请求。 datetime2参数必须在datetime1之后。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: http://example.org
        predicates:
        - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]
```

此路线适用于2017年1月20日17:42山地时间（丹佛）和2017年1月21日17:42山地时间（丹佛）之后的任何请求。 这对于维护窗口可能很有用。

110.4 Cookie Route Predicate Factory 译：110.4 Cookie路由谓词工厂

Cookie路由谓词工厂接受两个参数，即cookie名称和正则表达式。 此谓词匹配具有给定名称且值与正则表达式匹配的cookie。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: http://example.org
        predicates:
        - Cookie=chocolate, ch.p
```

此路线匹配请求具有名为 `chocolate` 的cookie，其值与 `ch.p` 正则表达式匹配。

110.5 Header Route Predicate Factory 译：110.5标题 路径谓词工厂

标题路由谓词工厂接受两个参数，标题名称和正则表达式。 此谓词与具有给定名称并且值与正则表达式匹配的标头匹配。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: http://example.org
        predicates:
        - Header=X-Request-Id, \d+
```

如果请求有一个名为 `X-Request-Id` 值，该值与 `\d+` 正则表达式（具有一个或多个数字的值）匹配，则此路由匹配。

110.6 Host Route Predicate Factory 译：110.6主机路由谓词工厂

主机路由谓词工厂接受一个参数：主机名称模式。 该模式是以 `.` 作为分隔符的Ant样式模式。 此谓词匹配与模式匹配的 `Host` 标题。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: http://example.org
        predicates:
        - Host=**.somehost.org
```

如果请求的 `Host` 标头的值为 `www.somehost.org` 或 `beta.somehost.org` 则此路线将匹配。

110.7 Method Route Predicate Factory 译：110.7方法路由谓词工厂

方法路线谓词工厂接受一个参数：匹配的HTTP方法。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: http://example.org
        predicates:
        - Method=GET
```

如果请求方法是`GET`此路线将匹配。

110.8 Path Route Predicate Factory译：110.8路径路由谓词工厂

路径路径谓词工厂采用一个参数：春季`PathMatcher`模式。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: http://example.org
        predicates:
        - Path=/foo/{segment}
```

如果请求路径是，则此路线将匹配，例如：`/foo/1`或`/foo/bar`。

此谓词将URI模板变量（`segment`定义的`segment`为名称和值的映射，并将其放置在`ServerWebExchange.getAttributes()`并使用在`PathRoutePredicate.URL_PREDICATE_VARS_ATTR`定义的键。 这些值可供GatewayFilter Factories使用

110.9 Query Route Predicate Factory译：110.9查询路由谓词工厂

查询路线谓词工厂有两个参数：必需的`param`和可选的`regexp`。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: http://example.org
        predicates:
        - Query=baz
```

如果请求包含`baz`查询参数，则此路线将匹配。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: http://example.org
        predicates:
        - Query=foo, ba.
```

如果请求包含`foo`查询参数，其值与`ba.`表达式匹配，则该路线将匹配，因此`bar`和`baz`将匹配。

110.10 RemoteAddr Route Predicate Factory译：110.10 RemoteAddr路由谓词工厂

RemoteAddr Route Predicate Factory采用CIDR表示法（IPv4或IPv6）字符串的列表（最小大小1），例如`192.168.0.1/16`（其中`192.168.0.1`是IP地址，`16`是子网掩码）。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: http://example.org
        predicates:
        - RemoteAddr=192.168.1.1/24
```

如果请求的远程地址是，例如，`192.168.1.10`，此路线将匹配。

110.10.1 Modifying the way remote addresses are resolved译：110.10.1修改远程地址解析的方式

默认情况下，RemoteAddr Route Predicate Factory使用传入请求中的远程地址。 如果Spring云网关位于代理层后面，则这可能与实际的客户端IP地址不匹配。

您可以通过设置自定义`RemoteAddressResolver`来自定义远程地址的解析方式。 春季云网关来与基于关闭的一个非默认的远程地址解析器X-Forwarded-For header，`XForwardedRemoteAddressResolver`。

`XForwardedRemoteAddressResolver`有两种静态构造方法，它们采用不同的安全方法：

`XForwardedRemoteAddressResolver::trustAll`返回一个`RemoteAddressResolver`，它始终采用`X-Forwarded-For`标头中的第一个IP地址。 这种方法容易受到欺骗，因为恶意客户可以为解析器所接受的`X-Forwarded-For`设置初始值。

`XForwardedRemoteAddressResolver::maxTrustedIndex`的索引与Spring Cloud Gateway前面运行的可信基础架构的数量相关。 例如，如果Spring Cloud Gateway仅可通过HAProxy访问，则应使用值1。 如果在可访问Spring Cloud Gateway之前需要两跳可信基础架构，则应使用值2。

给定以下标题值：

```
X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3
```

下面的 `maxTrustedIndex`值将产生以下远程地址。

`maxTrustedIndex`	result
[`Integer.MIN_VALUE`]	（无效，初始化期间为 `IllegalArgumentException` ）
1	0.0.0.3
2	0.0.0.2
3	0.0.0.1
[`Integer.MAX_VALUE`]	0.0.0.1

使用Java配置：

GatewayConfig.java

```
RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...

.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1")
.route("proxied-route",
    r -> r.remoteAddr(resolver,  "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2")
)
```

111. GatewayFilter Factories译：111. GatewayFilter工厂

路由过滤器允许以某种方式修改传入的HTTP请求或传出的HTTP响应。 路由过滤器的作用域是一个特定的路由。 Spring Cloud Gateway包含许多内置的GatewayFilter工厂。

注意有关如何使用以下任何滤镜的更多详细示例，请查看 unit tests 。

111.1 AddRequestHeader GatewayFilter Factory译：111.1 AddRequestHeader GatewayFilter Factory

AddRequestHeader GatewayFilter Factory采用名称和值参数。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: http://example.org
        filters:
        - AddRequestHeader=X-Request-Foo, Bar
```

这将为所有匹配请求添加 `X-Request-Foo:Bar`头部到下游请求的头部。

111.2 AddRequestParameter GatewayFilter Factory译：111.2 AddRequestParameter GatewayFilter Factory

AddRequestParameter GatewayFilter Factory接受名称和值参数。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: http://example.org
        filters:
        - AddRequestParameter=foo, bar
```

这将为下游请求的所有匹配请求的查询字符串添加 `foo=bar` 。

111.3 AddResponseHeader GatewayFilter Factory译：111.3 AddResponseHeader GatewayFilter Factory

AddResponseHeader GatewayFilter Factory接受名称和值参数。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: http://example.org
        filters:
        - AddResponseHeader=X-Response-Foo, Bar
```

这将为所有匹配请求添加`X-Response-Foo:Bar`头部到下游响应的头部。

111.4 Hystrix GatewayFilter Factory译：111.4 Hystrix GatewayFilter工厂

Hystrix是来自Netflix的库，实现了circuit breaker pattern 。 Hystrix GatewayFilter允许您将断路器引入网关路由，保护您的服务免受级联故障的影响，并允许您在发生下行故障时提供回退响应。

要在您的项目中启用Hystrix GatewayFilters，请从 Spring Cloud Netflix添加对`spring-cloud-starter-netflix-hystrix`的依赖关系。

Hystrix GatewayFilter Factory需要一个`name`参数，它是`HystrixCommand`的名称。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: hystrix_route
        uri: http://example.org
        filters:
        - Hystrix=myCommandName
```

这将`HystrixCommand`的其余过滤器包含在命令名称`myCommandName`。

Hystrix过滤器也可以接受可选的`fallbackUri`参数。 目前，仅支持`forward:`策划的URI。 如果回调被调用，请求将被转发到与URI匹配的控制器。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: hystrix_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingserviceendpoint
        filters:
        - name: Hystrix
          args:
            name: fallbackcmd
            fallbackUri: forward:/incaseoffailureusethis
        - RewritePath=/consumingserviceendpoint, /backingserviceendpoint
```

当Hystrix回退被调用时，这将转发到`/incaseoffailureusethis` URI。 请注意，此示例还通过目标URI上的`lb`前缀演示（可选）Spring Cloud Netflix功能区负载平衡。

Hystrix设置（例如超时）可以使用全局默认值进行配置，也可以使用 Hystrix wiki中介绍的应用程序属性按路线进行 配置 。

要为上面的示例路由设置5秒超时，将使用以下配置：

application.yml。

```
hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseconds: 5000
```

111.5 PrefixPath GatewayFilter Factory译：111.5 PrefixPath GatewayFilter Factory

PrefixPath GatewayFilter Factory采用一个`prefix`参数。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: http://example.org
        filters:
        - PrefixPath=/mypath
```

这会将前缀`/mypath`到所有匹配请求的路径中。 所以请求`/hello`，将被发送到`/mypath/hello`。

111.6 PreserveHostHeader GatewayFilter Factory译：111.6 PreserveHostHeader GatewayFilter Factory

PreserveHostHeader GatewayFilter Factory没有参数。 此过滤器设置路由过滤器将检查的请求属性，以确定是否应发送原始主机标头，而不是由http客户端确定的主机标头。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: preserve_host_route
        uri: http://example.org
        filters:
        - PreserveHostHeader
```

这会将`/mypath`前缀`/mypath`到所有匹配请求的路径中。 所以请求`/hello`，将被发送到`/mypath/hello`。

111.7 RequestRateLimiter GatewayFilter Factory译：111.7 RequestRateLimiter GatewayFilter Factory

RequestRateLimiter GatewayFilter Factory使用`RateLimiter`实现来确定是否允许当前请求继续。 如果不是，则返回状态`HTTP 429 - Too Many Requests`（默认情况下）。

此过滤器采用可选的`keyResolver`参数和特定于速率限制器的参数（请参见下文）。

`keyResolver`是实现`KeyResolver`接口的bean。 在配置中，使用SpEL按名称引用bean。 `#{@myKeyResolver}`是引用名称为`myKeyResolver`的bean的SpEL表达式。

KeyResolver.java。

```
public interface KeyResolver {
 Mono<String> resolve(ServerWebExchange exchange);
}
```

`KeyResolver`接口允许可插拔策略派生出限制请求的关键。 在未来的里程碑中，将会有一些`KeyResolver`实现。

的默认实现`KeyResolver`为`PrincipalNameKeyResolver`以检索`Principal`从`ServerWebExchange`并调用`Principal.getName()`。

RequestRateLimiter不能通过“快捷方式”符号进行配置。 下面的例子是*无效的*

application.properties。

```
# INVALID SHORTCUT CONFIGURATION
spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2, #{@userkeyresolver}
```

111.7.1 Redis RateLimiter译：111.7.1 Redis RateLimiter

redis的实现基于Stripe完成的工作。 它需要使用`spring-boot-starter-data-redis-reactive` Spring Boot启动器。

使用的算法是 Token Bucket Algorithm 。

`redis-rate-limiter.replenishRate`是您希望允许用户每秒处理多少个请求，而不会丢失请求。 这是令牌桶被填充的速率。

`redis-rate-limiter.burstCapacity`是允许用户在一秒钟内完成的最大请求数。 这是令牌桶可以容纳的令牌的数量。 将此值设置为零将阻止所有请求。

通过在`replenishRate`和`burstCapacity`设置相同的值来实现稳定的速率。 设置`burstCapacity`高于`replenishRate`可以允许临时爆发。 在这种情况下，速率限制器需要在脉冲之间允许一段时间（根据`replenishRate`），因为2个连续的脉冲将导致请求丢失（`HTTP 429 - Too Many Requests`）。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: http://example.org
        filters:
        - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
```

Config.java。

```
@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
}
```

这定义了每个用户10个请求速率限制。 允许突发20次，但下一秒只有10次请求可用。 `KeyResolver`是一个简单的获取`user`请求参数（注意：这不建议用于生产）。

速率限制器也可以定义为实现`RateLimiter`接口的bean。 在配置中，使用SpEL按名称引用bean。 `#{@myRateLimiter}`是引用名称为`myRateLimiter`的bean的SpEL表达式。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: http://example.org
        filters:
        - name: RequestRateLimiter
          args:
            rate-limiter: "#{@myRateLimiter}"
            key-resolver: "#{@userKeyResolver}"
```

111.8 RedirectTo GatewayFilter Factory译：111.8 RedirectTo GatewayFilter Factory

RedirectTo GatewayFilter Factory采用`status`和`url`参数。 该状态应该是300系列重定向http代码，例如301.该网址应该是有效的网址。 这将是`Location`标题的值。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: http://example.org
        filters:
        - RedirectTo=302, http://acme.org
```

这将发送一个状态302和一个 `Location:http://acme.org`标题来执行重定向。

111.9 RemoveNonProxyHeaders GatewayFilter Factory译：111.9 RemoveNonProxyHeaders GatewayFilter Factory

RemoveNonProxyHeaders GatewayFilter Factory从转发的请求中删除标题。 被删除的标题的默认列表来自IETF 。

默认删除标题是：

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer
- Transfer-Encoding
- Upgrade

要更改此设置，请将 `spring.cloud.gateway.filter.remove-non-proxy-headers.headers`属性设置为要删除的标题名称列表。

111.10 RemoveRequestHeader GatewayFilter Factory译：111.10 RemoveRequestHeader GatewayFilter Factory

RemoveRequestHeader GatewayFilter Factory需要`name`参数。 这是要删除标题的名称。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestheader_route
        uri: http://example.org
        filters:
        - RemoveRequestHeader=X-Request-Foo
```

这将在下游发送之前删除 `X-Request-Foo`标头。

111.11 RemoveResponseHeader GatewayFilter Factory译：111.11 RemoveResponseHeader GatewayFilter Factory

RemoveResponseHeader GatewayFilter Factory需要`name`参数。 这是要删除标题的名称。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: removeresponseheader_route
        uri: http://example.org
        filters:
        - RemoveResponseHeader=X-Response-Foo
```

这会在返回到网关客户端之前从响应中删除 `X-Response-Foo`头。

111.12 RewritePath GatewayFilter Factory译：111.12 RewritePath GatewayFilter Factory

RewritePath GatewayFilter Factory采用路径`regexp`参数和`replacement`参数。 这使用Java正则表达式来灵活地重写请求路径。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritepath_route
        uri: http://example.org
        predicates:
        - Path=/foo/**
        filters:
        - RewritePath=/foo/(?<segment>.*), /$\{segment}
```

对于请求路径`/foo/bar`，这将在进行下游请求之前将路径设置为`/bar` 。 由于YAML规范，请注意`$\`被替换为`$` 。

111.13 SaveSession GatewayFilter Factory译：111.13 SaveSession GatewayFilter Factory

SaveSession GatewayFilter Factory *在*向下游转发呼叫*之前*强制执行`WebSession::save`操作。 当使用类似Spring Session的缓存数据存储并且需要确保在进行转发呼叫

之前保存会话状态时，这是特别有用的。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: save_session
        uri: http://example.org
        predicates:
        - Path=/foo/**
        filters:
        - SaveSession
```

如果您将 Spring Security与Spring Session集成，并且希望确保将安全细节转发给远程进程，这至关重要。

111.14 SecureHeaders GatewayFilter Factory 译：111.14 SecureHeaders GatewayFilter Factory

所述SecureHeaders GatewayFilter厂增加了许多标题来在从reccomendation响应 this blog post 。

添加以下标题（使用默认值的allong）：

- `X-Xss-Protection:1; mode=block`
- `Strict-Transport-Security:max-age=631138519`
- `X-Frame-Options:DENY`
- `X-Content-Type-Options:nosniff`
- `Referrer-Policy:no-referrer`
- `Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src htt`
- `X-Download-Options:noopen`
- `X-Permitted-Cross-Domain-Policies:none`

要更改默认值，请在 `spring.cloud.gateway.filter.secure-headers` 命名空间中设置适当的属性：

要更改的属性：

- `xss-protection-header`
- `strict-transport-security`
- `frame-options`
- `content-type-options`
- `referrer-policy`
- `content-security-policy`
- `download-options`
- `permitted-cross-domain-policies`

111.15 SetPath GatewayFilter Factory 译：111.15 SetPath GatewayFilter Factory

SetPath GatewayFilter Factory采用路径 `template` 参数。 它提供了一种通过允许路径的模板化段来操纵请求路径的简单方法。 这使用了Spring Framework的uri模板。 允许多个匹配段。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: setpath_route
        uri: http://example.org
        predicates:
        - Path=/foo/{segment}
        filters:
        - SetPath=/{segment}
```

对于请求路径 `/foo/bar` ，这将在进行下游请求之前将路径设置为 `/bar` 。

111.16 SetResponseHeader GatewayFilter Factory 译：111.16 SetResponseHeader GatewayFilter Factory

SetResponseHeader GatewayFilter Factory需要 `name` 和 `value` 参数。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: http://example.org
        filters:
        - SetResponseHeader=X-Response-Foo, Bar
```

这个GatewayFilter用所给的名字替换所有的头，而不是添加。 因此，如果下游服务器使用 `X-Response-Foo:1234` 作出响应，则将替换为网关客户端将收到的 `X-Response-Foo:Bar` 。

111.17 SetStatus GatewayFilter Factory 译：111.17 SetStatus GatewayFilter Factory

SetStatus GatewayFilter Factory采用一个 `status` 参数。 它必须是有效的春季 `HttpStatus` 。 它可能是整数值 `404` 或枚举 `NOT_FOUND` 的字符串表示 `NOT_FOUND` 。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatusstring_route
        uri: http://example.org
        filters:
        - SetStatus=BAD_REQUEST
      - id: setstatusint_route
        uri: http://example.org
        filters:
        - SetStatus=401
```

无论哪种情况，响应的HTTP状态都将设置为401。

111.18 StripPrefix GatewayFilter Factory译：111.18 StripPrefix GatewayFilter Factory

StripPrefix GatewayFilter Factory需要一个参数`parts`。 `parts`参数指示在向下游发送请求之前从请求中剥离的路径中的部件数量。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: nameRoot
        uri: http://nameservice
        predicates:
        - Path=/name/**
        filters:
        - StripPrefix=2
```

当一个请求通过网关做出`/name/bar/foo`给发出请求`nameservice`看起来像`http://nameservice/foo`。

111.19 Retry GatewayFilter Factory译：111.19重试GatewayFilter工厂

重试GatewayFilter厂需要`retries`， `statuses`， `methods`，并`series`作为参数。

- `retries`: the number of retries that should be attempted
- `statuses`: the HTTP status codes that should be retried, represented using `org.springframework.http.HttpStatus`
- `methods`: the HTTP methods that should be retried, represented using `org.springframework.http.HttpMethod`
- `series`: the series of status codes to be retried, represented using `org.springframework.http.HttpStatus.Series`

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: retry_test
        uri: http://localhost:8080/flakey
        predicates:
        - Host=*.retry.com
        filters:
        - name: Retry
          args:
            retries: 3
            statuses: BAD_GATEWAY
```

此时使用`forward`协议的URI不支持使用重`forward`滤器。

112. Global Filters译：112.全局过滤器

该`GlobalFilter`接口具有相同的签名`GatewayFilter`。 这些是有条件应用于所有路线的特殊过滤器。（这个界面和用法在未来的里程碑中可能会发生变化）。

112.1 Combined Global Filter and GatewayFilter Ordering译：112.1组合全局过滤器和GatewayFilter排序

TODO：文件订购

112.2 Forward Routing Filter译：112.2前向路由过滤器

`ForwardRoutingFilter`在交换属性`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`查找URI。 如果url有`forward`方案（即`forward:///localendpoint`），它将使用Spring `DispatcherHandler`来处理请求。 请求URL的路径部分将被转发URL中的路径覆盖。 未修改的原始URL将追加到`ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR`属性的列表中。

112.3 LoadBalancerClient Filter译：112.3 LoadBalancerClient过滤器

`LoadBalancerClientFilter`在交换属性`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`查找URI。 如果URL有`lb`方案（即`lb://myservice`），它会使用Spring云`LoadBalancerClient`解析名称（ `myservice`前面例子中）和实际主机和端口，并在相同的属性替换URI。 未修改的原始URL将附加到`ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR`属性的列表中。 该过滤器也会查看`ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR`属性，看它是否等于`lb`，然后应用相同的规则。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
        predicates:
        - Path=/service/**
```

112.4 Netty Routing Filter译：112.4 Netty路由过滤器

如果位于`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`交换属性中的url具有`http`或`https`方案，`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`运行Netty路由筛选器。 它使用Netty `HttpClient`进行下游代理请求。 该响应放在`ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR`交换属性中，以用于以后的过滤器。（有一个实验`WebClientHttpRoutingFilter`执行相同的功能，但不需要netty）

112.5 Netty Write Response Filter译：112.5 Netty写入响应过滤器

该`NettyWriteResponseFilter`运行，如果有一个的Netty `HttpClientResponse`在`ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR`交换属性。 它在所有其他过滤器完成后运行，并将代理响应写回网关客户端响应。（有一个实验`WebClientWriteResponseFilter`执行相同的功能，但不需要netty）

112.6 RouteToRequestUrl Filter译：112.6 RouteToRequestUrl过滤器

该`RouteToRequestUrlFilter`运行，如果有一个`Route`对象在`ServerWebExchangeUtils.GATEWAY_ROUTE_ATTR`交换属性。 它根据请求URI创建一个新的URI，但使用`Route`对象的URI属性进行更新。 新的URI位于`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`交换属性中。

如果URI具有方案前缀（如`lb:ws://serviceid`，则`lb`方案将从URI中剥离并放置在`ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR`以供以后在过滤器链中使用。

112.7 Websocket Routing Filter译：112.7 Websocket路由过滤器

如果位于`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`交换属性中的url具有`ws`或`wss`方案，则运行Websocket路由筛选器。 它使用Spring Web Socket基础结构向下游转发WebSocket请求。

可以通过在`lb`加上URI来加载Websockets，例如`lb:ws://serviceid`。

如果您使用 SockJS作为普通http的回退，则应该配置正常的HTTP路由以及WebSocket路由。

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      # SockJS route
      - id: websocket_sockjs_route
        uri: http://localhost:3001
        predicates:
        - Path=/websocket/info/**
      # Normwal Websocket route
      - id: websocket_route
        uri: ws://localhost:3001
        predicates:
        - Path=/websocket/**
```

112.8 Making An Exchange As Routed译：112.8交换路由

在网关路由`ServerWebExchange`，它将通过将`gatewayAlreadyRouted`添加到交换属性来将该交换标记为“路由”。 一旦请求被标记为路由，其他路由过滤器将不会再次路由请求，实际上会跳过该过滤器。 您可以使用便利方法将交易所标记为路由或检查交易所是否已经路由。

- `ServerWebExchangeUtils.isAlreadyRouted` takes a `ServerWebExchange` object and checks if it has been "routed"
- `ServerWebExchangeUtils.setAlreadyRouted` takes a `ServerWebExchange` object and marks it as "routed"

113. Configuration译：113配置

Spring Cloud Gateway的配置由`RouteDefinitionLocator`集合驱动。

RouteDefinitionLocator.java。

```
public interface RouteDefinitionLocator {
 Flux<RouteDefinition> getRouteDefinitions();
}
```

默认情况下，`PropertiesRouteDefinitionLocator`使用Spring Boot的`@ConfigurationProperties`机制加载属性。

上面的配置示例都使用使用位置参数而不是命名的快捷方式表示法。 下面的两个例子是等价的：

application.yml。

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatus_route
        uri: http://example.org
        filters:
        - name: SetStatus
          args:
            status: 401
      - id: setstatusshortcut_route
        uri: http://example.org
        filters:
        - SetStatus=401
```

对于网关的某些用法，属性将足够，但某些生产用例将从加载来自外部源（例如数据库）的配置中受益。 未来的里程碑版本将基于Spring数据存储库（如Redis，MongoDB和Cassandra）实现`RouteDefinitionLocator`实现。

113.1 Fluent Java Routes API译：113.1流利Java路由API

为了在Java中进行简单的配置，在`RouteLocatorBuilder`bean中定义了流畅的API。

GatewaySampleApplication.java。

```
// static imports from GatewayFilters and RoutePredicates
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder, ThrottleGatewayFilterFactory throttle) {
    return builder.routes()
            .route(r -> r.host("**.abc.org").and().path("/image/png")
                .filters(f ->
                        f.addResponseHeader("X-TestHeader", "foobar"))
                .uri("http://httpbin.org:80")
            )
            .route(r -> r.path("/image/webp")
                .filters(f ->
                        f.addResponseHeader("X-AnotherHeader", "baz"))
                .uri("http://httpbin.org:80")
            )
            .route(r -> r.order(-1)
                .host("**.throttle.org").and().path("/get")
                .filters(f -> f.filter(throttle.apply(1,
                        1,
                        10,
                        TimeUnit.SECONDS)))
                .uri("http://httpbin.org:80")
            )
            .build();
}
```

这种风格还允许更多的自定义谓词断言。 由`RouteDefinitionLocator`bean定义的谓词使用逻辑`and`进行组合。 通过使用流利的Java API，你可以使用`and()`，`or()`和`negate()`对运营`Predicate`类。

113.2 DiscoveryClient Route Definition Locator译：113.2 DiscoveryClient路由定义定位器

网关可以被配置为创建基于与注册服务路线`DiscoveryClient`兼容的服务注册表。

要启用此功能，请设置`spring.cloud.gateway.discovery.locator.enabled=true`并确保`DiscoveryClient`实现位于类路径中并且已启用（例如Netflix Eureka，Consul或Zookeeper）。

114. Actuator API译：114执行器API

TODO：记录`/gateway`执行器端点

115. Developer Guide译：115.开发者指南

TODO：编写定制集成的概述

115.1 Writing Custom Route Predicate Factories译：115.1编写自定义路由谓词工厂

TODO：编写自定义路由谓词工厂的文档

115.2 Writing Custom GatewayFilter Factories译：115.2编写自定义GatewayFilter工厂

为了写一个GatewayFilter，你需要实现`GatewayFilterFactory`。 有一个可以扩展的抽象类`AbstractGatewayFilterFactory`。

PreGatewayFilterFactory.java。

```
---
public class PreGatewayFilterFactory extends AbstractGatewayFilterFactory<PreGatewayFilterFactory.Config> {

public PreGatewayFilterFactory() {
 super(Config.class);
}

@Override
public GatewayFilter apply(Config config) {
 // grab configuration from Config object
 return (exchange, chain) -> {
           //If you want to build a "pre" filter you need to manipulate the
           //request before calling change.filter
           ServerHttpRequest.Builder builder = exchange.getRequest().mutate();
           //use builder to manipulate the request
           return chain.filter(exchange.mutate().request(request).build());
 };
}
```

```
public static class Config {
        //Put the configuration properties for your filter here
}

}
```

PostGatewayFilterFactory.java。

```
---
public class PostGatewayFilterFactory extends AbstractGatewayFilterFactory<PostGatewayFilterFactory.Config> {

public PostGatewayFilterFactory() {
 super(Config.class);
}

@Override
public GatewayFilter apply(Config config) {
 // grab configuration from Config object
 return (exchange, chain) -> {
  return chain.filter(exchange).then(Mono.fromRunnable(() -> {
   ServerHttpReponse response = exchange.getResponse();
   //Manipulate the response in some way
  }));
 };
}

public static class Config {
        //Put the configuration properties for your filter here
}

}
```

115.3 Writing Custom Global Filters译：115.3编写自定义全局过滤器

TODO：编写自定义全局过滤器的文档

115.4 Writing Custom Route Locators and Writers译：115.4编写自定义路线定位器和作家

TODO：撰写自定义路线定位器和作家的文档

116. Building a Simple Gateway Using Spring MVC or Webflux译：116.使用Spring MVC或Webflux构建一个简单的网关

Spring Cloud Gateway提供了一个名为`ProxyExchange`的实用程序对象，您可以在常规的Spring Web处理程序中将其用作方法参数。 它通过镜像HTTP动词的方法支持基本的下游HTTP交换。 通过MVC，它还支持通过`forward()`方法转发到本地处理程序。 要使用`ProxyExchange`只需在类路径中包含正确的模块（`spring-cloud-gateway-mvc`或`spring-cloud-gateway-webflux`）。

MVC示例（代理请求以“/ test”向下游到远程服务器）：

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

 @Value("${remote.home}")
 private URI home;

 @GetMapping("/test")
 public ResponseEntity<?> proxy(ProxyExchange<byte[]> proxy) throws Exception {
  return proxy.uri(home.toString() + "/image/png").get();
 }

}
```

Webflux也是如此：

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

 @Value("${remote.home}")
 private URI home;

 @GetMapping("/test")
 public Mono<ResponseEntity<?>> proxy(ProxyExchange<byte[]> proxy) throws Exception {
  return proxy.uri(home.toString() + "/image/png").get();
 }

}
```

`ProxyExchange`上有便利的方法来使处理程序方法能够发现和增强传入请求的URI路径。 例如，您可能想要提取路径的尾部元素以向下游传递它们：

```
@GetMapping("/proxy/path/**")
public ResponseEntity<?> proxyPath(ProxyExchange<byte[]> proxy) throws Exception {
  String path = proxy.path("/proxy/path/");
  return proxy.uri(home.toString() + "/foos/" + path).get();
}
```

Spring MVC或Webflux的所有功能都可用于网关处理程序方法。 例如，您可以注入请求标头和查询参数，并且可以使用映射注释中的声明约束传入的请求。 有关这些功能的更多详细信息，请参阅Spring MVC中`@RequestMapping`的文档。

头可以被添加到使用下游响应`header()`上的方法`ProxyExchange`。

您还可以通过将映射器添加到`get()`等方法来操纵响应头（以及任何您喜欢的响应）。 该映射器是一个`Function`，它接收传入的`ResponseEntity`并将其转换为传出。

为“敏感”标题（默认为“cookie”和“授权”）提供一流的支持，这些标题不会传递到下游，也可用于“代理”标题（`x-forwarded-*`）。

Part XVI. Spring Cloud Function译：第十六部分. 春天云功能

Mark Fisher，Dave Syer

117. Introduction译：117.介绍

Spring Cloud Function是一个具有以下高层目标的项目：

- Promote the implementation of business logic via functions.
- Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
- Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
- Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

它将所有的传输细节和基础设施抽象出来，允许开发人员保留所有熟悉的工具和流程，并将精力集中在业务逻辑上。

这里有一个完整的，可执行的，可测试的Spring Boot应用程序（实现简单的字符串操作）：

```
@SpringBootApplication
public class Application {

  @Bean
  public Function<Flux<String>, Flux<String>> uppercase() {
    return flux -> flux.map(value -> value.toUpperCase());
  }

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

它只是一个Spring Boot应用程序，所以它可以在本地和CI构建中以与其他Spring Boot应用程序相同的方式构建，运行和测试。 `Function`是从`java.util`和`Flux`是Reactive Streams `Publisher`从Project Reactor 。 该功能可以通过HTTP或消息传递进行访问。

春云功能有四个主要特点：

1. Wrappers for `@Beans` of type `Function`, `Consumer` and `Supplier`, exposing them to the outside world as either HTTP endpoints and/or message stream listeners/publishers with RabbitMQ, Kafka etc.
2. Compiling strings which are Java function bodies into bytecode, and then turning them into `@Beans` that can be wrapped as above.
3. Deploying a JAR file containing such an application context with an isolated classloader, so that you can pack them together in a single JVM.
4. Adapters for AWS Lambda, Azure, Apache OpenWhisk and possibly other "serverless" service providers.

Spring Cloud是在非限制性的Apache 2.0许可下发布的。 如果您想为本文档的这一部分做出贡献，或者如果您发现错误，请在github的项目中找到源代码和问题跟踪器。

118. Getting Started译：118.入门

从命令行构建（和“安装”样本）：

```
$ ./mvnw clean install
```

（如果你喜欢YOLO添加 `-DskipTests` ）

运行其中一个样本，例如

```
$ java -jar spring-cloud-function-samples/function-sample/target/*.jar
```

这运行应用程序并通过HTTP公开其功能，因此您可以将字符串转换为大写，如下所示：

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d Hello
HELLO
```

您可以通过用新行分隔多个字符串（一个 `Flux<String>` ）

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'Hello
> World'
HELLOWORLD
```

（您可以在终端中使用 `^J` ，以便像这样在文字串中插入新行。）

119. Building and Running a Function译：119.建立和运行一个功能

上面的示例 `@SpringBootApplication`有一个函数，可以在运行时由Spring Cloud Function修饰为HTTP端点或Stream处理器，例如RabbitMQ，Apache Kafka或JMS。

该`@Beans`可`Function` ， `Consumer`或者`Supplier` （均来自`java.util` ），以及它们的参数类型可以是字符串或POJO。 如果`spring-cloud-function-stream`位于类路径上， `Function`将作为Spring Cloud Stream `Processor` `spring-cloud-function-stream` 。 `Consumer`也暴露为流`Sink`和`Supplier`转换为流`Source` 。 如果Stream联编程序为`spring-cloud-stream-binder-servlet`则会显示HTTP端点。

函数可以是`Flux<String>`或`Flux<Pojo>`并且Spring Cloud Function负责将数据转换为所需类型并将其转换为所需类型，只要它以纯文本形式或（在POJO的情况下）JSON。 TBD：支持`Flux<Message<Pojo>>`和`Pojo`类型（框架暗示和实现的`Pojo` ）。

函数可以组合在一个应用程序中，也可以一次一个jar地部署。 这取决于开发人员的选择。 具有多种功能的应用程序可以多次部署在不同的“个性”中，在不同的物理传输中暴露不同的功能。

120. Function Catalog and Flexible Function Signatures译：120.功能目录和灵活的功能签名

Spring Cloud Function的主要功能之一是适应和支持用户定义函数的一系列类型签名。 因此，用户可以提供`Function<String,String>`类型的bean，`FunctionCatalog`将包装成`Function<Flux<String>,Flux<String>>` 。 用户通常不必关心`FunctionCatalog` ，但了解用户代码支持哪些类型的功能是非常有用的。

一般来说，用户可以预期，如果他们为简单的旧Java类型（或原始包装器）编写函数，那么函数目录会将其包装为相同类型的`Flux` 。 如果用户使用`Message` （来自

spring-messaging）编写函数，它将从支持键值元数据的任何适配器（例如HTTP标头）接收和传输标题。 这里是细节。

User Function	Catalog Registration
`Function<S,T>`	`Function<Flux<S>, Flux<T>>`
`Function<Message<S>,Message<T>>`	`Function<Flux<Message<S>>, Flux<Message<T>>>`
`Function<Flux<S>, Flux<T>>`	`Function<Flux<S>, Flux<T>>` （通过）
`Supplier<T>`	`Supplier<Flux<T>>`
`Supplier<Flux<T>>`	`Supplier<Flux<T>>`
`Consumer<T>`	`Function<Flux<T>, Mono<Void>>`
`Consumer<Message<T>>`	`Function<Flux<Message<T>>, Mono<Void>>`
`Consumer<Flux<T>>`	`Consumer<Flux<T>>`

消费者有点特殊，因为它有一个`void`返回类型，这意味着阻塞，至少潜在的。 很可能你不需要编写`Consumer<Flux<?>>`，但如果你需要这样做，请记住订阅输入流量。 如果您声明非发布者类型的`Consumer`（这是正常的），它将被转换为返回发布者的函数，以便可以以受控方式订阅它。

函数目录可以包含具有相同名称的`Supplier`和`Function`（或`Consumer`）（如同一个资源的GET和POST）。 它甚至可以包含`Consumer<Flux<>>`具有相同名称为`Function`，但它不能包含`Consumer<T>`和`Function<T,S>`具有相同的名称时`T`不是`Publisher`，因为消费者将被转换为`Function`，只有其中一个可以被注册。

121. Standalone Web Applications译：121.独立的Web应用程序

`spring-cloud-function-web`模块具有自动配置功能，当它被包含在Spring Boot Web应用程序中时（具有MVC支持），该模块就会激活。 还有一个`spring-cloud-starter-function-web`收集所有可选依赖项，以防您只需要一个简单的入门体验。

在Web配置激活的情况下，您的应用程序将拥有一个MVC端点（默认为“/”，但可配置为`spring.cloud.function.web.path`），可用于访问应用程序上下文中的功能。 支持的内容类型是纯文本和JSON。

方法	Path	Request	Response	Status
得到	/ {}供应商	-	来自指定供应商的物品	200 OK
POST	/{消费者}	JSON对象或文本	镜像输入并将请求主体推送到消费者	202接受
POST	/{消费者}	JSON数组或带有新行的文本	镜子输入并将身体逐一推入消费者	202接受
POST	/{功能}	JSON对象或文本	应用指定函数的结果	200 OK
POST	/{功能}	JSON数组或带有新行的文本	应用指定函数的结果	200 OK
得到	/ {函数} / {}项	-	将项目转换为对象并返回应用该函数的结果	200 OK

如上表所示，端点的行为取决于方法以及传入请求数据的类型。 当传入数据是单值时，目标函数声明为明显单值（即不返回集合或`Flux`），那么响应也将包含单个值。 对于多值响应，客户端可以通过发送“Accept：text / event-stream”来请求服务器发送的事件流。如果只有一个函数（消费者等），则路径中的名称是可选的。可以使用管道或逗号来分隔函数名称（管道在URL路径中是合法的，但在命令行上键入有点尴尬）。

在`Message<?>`使用输入和输出声明的函数和使用者将在输入消息中看到请求标头，并且输出消息标头将被转换为HTTP标头。

在发布文本时，响应格式可能与Spring Boot 2.0和旧版本不同，具体取决于内容协商（提供内容类型和获取最佳结果的头文件）。

122. Standalone Streaming Applications译：122.独立流媒体应用程序

要发送或接收来自代理（如RabbitMQ或Kafka）的消息，您可以使用`spring-cloud-function-stream`适配器。 将适配器与来自Spring Cloud Stream的适当绑定器一起添加到您的类路径中。 该适配器将绑定到消息代理`Processor`（输入和输出流），除非用户使用`spring.cloud.function.stream.{source,sink}.enabled=false`明确禁用其中一个。

传入的消息被路由到功能（或消费者）。 如果只有一个，那么选择是显而易见的。 如果有多个函数可以接受传入消息，则检查该消息以查看是否存在包含函数名称的`stream_routekey`头。 路由标题或函数名称可以使用逗号或管道分隔的名称进行组合。 标题也被添加到来自供应商的传出消息中。 没有路由密钥的消息可以通过指定`spring.cloud.function.stream.{processor,sink}.name`专门路由到功能或消费者。 如果无法识别单个功能来处理传入消息，则会出现错误，除非您设置了`spring.cloud.function.stream.shared=true`，在这种情况下，这些消息将被发送到所有兼容功能。 可以使用`spring.cloud.function.stream.source.name`为供应商的输出消息（如果有多个可用）选择单一供应商。

如果消息代理不可用并且功能目录包含在访问时立即生成消息的供应商，则某些活页夹将在启动时失败。 您可以使用`spring.cloud.function.strean.supplier.enabled=false`标志在启动时关闭供应商的自动发布。

123. Deploying a Packaged Function译：123.部署打包函数

Spring Cloud Function提供了一个“部署者”库，允许您使用独立的类加载器启动一个jar文件（或展开的归档文件或一组jar文件），并展示其中定义的函数。 这是一个非常强大的工具，它允许您在不更改目标jar文件的情况下，将函数调整到一系列不同的输入输出适配器。 无服务器平台往往有这种内置的功能，所以你可以把它看作一个构建块的函数调用在这样一个平台（的确Riff Java函数调用使用这个库）。

API的标准入口点是Spring配置注释`@EnableFunctionDeployer`。 如果在Spring Boot应用程序中使用了该功能，则部署者将启动并查找一些配置，以告诉它在哪里查找功能jar。 用户必须至少提供一个`function.location`，它是包含这些功能的归档的URL或资源位置。 它可以选择使用`maven:`前缀通过依赖关系查找来查找工件（有关完整的详细信息，请参见`FunctionProperties`）。 Spring Boot应用程序从jar文件中引导，使用`MANIFEST.MF`找到一个启动类，以便标准的Spring Boot fat jar运行良好。 如果目标罐子可以成功启动，那么结果就是在主应用程序`FunctionCatalog`注册的一个函数。 已注册的函数可以通过主应用程序中的代码应用，即使它是在隔离的类加载器（通过deault）中创建的。

124. Dynamic Compilation译：124.动态编译

有一个示例应用程序使用函数编译器从配置属性创建函数。 香草“功能样本”也具有该功能。 还有一些脚本可以运行以查看运行时发生的编译。 要运行这些示例，请转到`scripts`目录：

```
cd scripts
```

另外，在本地启动RabbitMQ服务器（例如执行`rabbitmq-server`）。

启动功能注册表服务：

```
./function-registry.sh
```

注册一个功能：

```
./registerFunction.sh -n uppercase -f "f->f.map(s->s.toString().toUpperCase())"
```

使用该功能运行REST Microservice：

```
./web.sh -f uppercase -p 9000
curl -H "Content-Type: text/plain" -H "Accept: text/plain" localhost:9000/uppercase -d foo
```

注册供应商：

```
./registerSupplier.sh -n words -f "()->Flux.just(\"foo\",\"bar\")"
```

使用该供应商运行REST Microservice：

```
./web.sh -s words -p 9001
curl -H "Accept: application/json" localhost:9001/words
```

注册消费者：

```
./registerConsumer.sh -n print -t String -f "System.out::println"
```

使用该消费者运行REST Microservice：

```
./web.sh -c print -p 9002
curl -X POST -H "Content-Type: text/plain" -d foo localhost:9002/print
```

运行流处理微服务：

首先注册流文字供应商：

```
./registerSupplier.sh -n wordstream -f "()->Flux.interval(Duration.ofMillis(1000)).map(i->\"message-\"+i)"
```

然后启动源（供应商），处理器（功能）和接收器（消费者）应用程序（以相反顺序）：

```
./stream.sh -p 9103 -i uppercaseWords -c print
./stream.sh -p 9102 -i words -f uppercase -o uppercaseWords
./stream.sh -p 9101 -s wordstream -o words
```

输出将显示在接收器应用程序的控制台中（每秒一条消息，转换为大写）：

```
MESSAGE-0
MESSAGE-1
MESSAGE-2
MESSAGE-3
MESSAGE-4
MESSAGE-5
MESSAGE-6
MESSAGE-7
MESSAGE-8
MESSAGE-9
...
```

125. Serverless Platform Adapters译：125.无服务器平台适配器

除了能够作为独立进程运行，Spring云功能应用程序可以适用于运行其中一个现有的无服务器平台。 在该项目中有对适配器AWS Lambda， Azure，并Apache OpenWhisk 。 Oracle Fn platform有它自己的Spring云功能适配器。 而Riff支持Java函数，其本身的Java Function Invoker行为是Spring Cloud Function jars的适配器。

125.1 AWS Lambda译：125.1 AWS Lambda

AWS适配器采用Spring Cloud Function应用程序，并将其转换为可在AWS Lambda中运行的表单。

125.1.1 Introduction译：125.1.1简介

适配器有几个可以使用的通用请求处理程序。 最通用的是`SpringBootStreamHandler`，它使用Spring Boot提供的Jackson `ObjectMapper`对函数中的对象进行序列化和反序列化。 还有一个`SpringBootRequestHandler`，您可以扩展它，并将输入和输出类型作为类型参数提供（使AWS可以检查类并执行JSON转换本身）。

如果您的应用有多个`@Bean`类型`Function`等，则可以通过配置`function.name`（例如，AWS中的`FUNCTION_NAME`环境变量）选择要使用的`function.name`。 这些函数从Spring Cloud `FunctionCatalog`中提取（首先搜索`Function`然后搜索`Consumer`，最后搜索`Supplier`）。

125.1.2 Notes on JAR Layout译：125.1.2关于JAR布局的注意事项

您在运行时不需要Lambda中的Spring Cloud Function Web或Stream适配器，因此您可能需要在创建发送给AWS的JAR之前排除这些适配器。 Lambda应用程序必须加阴影，但Spring Boot独立应用程序不能，因此您可以使用2个独立的jar（根据示例）运行相同的应用程序。 示例应用程序创建了2个jar文件，其中一个具有用于在Lambda中部署的`aws`分类器，以及一个包含`spring-cloud-function-web`可执行（精简）jar在运行时。 Spring Cloud Function将尝试使用`Start-Class`属性（如果您使用初始父项，将通过Spring Boot工具为您添加）从JAR文件清单中为您找到“主类”。 如果清单中没有`Start-Class`，则可以在将功能部署到AWS时使用环境变量`MAIN_CLASS`。

125.1.3 Upload译：125.1.3上传

在`spring-cloud-function-samples/function-sample-aws`下构建示例并将`-aws` jar文件上传到Lambda。 处理程序可以是`example.Handler`或`org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler`（该类的FQN， *不是*方法引用，尽管Lambda确实接受方法引用）。

```
./mvnw -U clean package
```

使用AWS命令行工具，它看起来像这样：

```
aws lambda create-function --function-name Uppercase --role arn:aws:iam::[USERID]:role/service-role/[ROLE] --zip-file fileb://function-sample-aws/tar
```

AWS示例中函数的输入类型是具有名为“value”的单个属性的Foo。 所以你需要这个来测试它：

```
{
  "value": "test"
}
```

125.1.4 Platfom Specific Features译：125.1.4 Platfom特定功能

HTTP and API Gateway译：HTTP和API网关

AWS有一些平台特定的数据类型，包括批量处理消息，这比单独处理每个消息更有效。 要使用这些类型，可以编写一个依赖于这些类型的函数。 或者您可以依靠Spring从AWS类型中提取数据并将其转换为Spring `Message`。 要执行此操作，请告诉AWS此函数是特定通用处理程序类型（取决于AWS服务），并提供类型为`Function<Message<S>,Message<T>>`的bean，其中`S`和`T`是您的业务数据类型。 如果有多个类型为`Function` bean， `Function`可能还需要将Spring Boot属性`function.name`配置为目标Bean的名称（例如，使用`FUNCTION_NAME`作为环境变量）。

下面列出了受支持的AWS服务和通用处理程序类型：

Service	AWS Types	Generic Handler
API网关	`APIGatewayProxyRequestEvent`，`APIGatewayProxyResponseEvent`	`org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler`
室壁运动	KinesisEvent	org.springframework.cloud.function.adapter.aws.SpringBootKinesisEventHandler

例如，要部署在API网关后面， `--handler org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler`在AWS命令行中使用`--handler org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler`（通过UI），并定义类型为`@Bean`的`Function<Message<Foo>,Message<Bar>>`，其中`Foo`和`Bar`是POJO类型（数据将由AWS使用Jackson进行编组`Bar`）。

125.2 Azure Functions译：125.2 Azure函数

Azure适配器引导Spring Cloud Function上下文，并在必要时使用Spring Boot配置将来自Azure框架的函数调用传递到用户函数。 Azure函数具有相当独特的入侵式编程模型，涉及用户代码中特定于平台的注释。 Spring Cloud函数Azure适配器交换了这些注释的便利性，以实现函数实现的可移植性。 用手工编写一些JSON（至少现在）来引导平台在适配器中调用正确的方法，而不是使用注释。

此项目为Spring云功能应用程序提供适配器层到Azure。 您可以使用`@Bean`类型的应用程序编写一个`@Bean`类型的应用程序，并且可以在Azure中进行部署， `Function`是您将JAR文件布置得恰到好处。

该适配器具有可供选择使用的通用HTTP请求处理程序。 有一个必须扩展的`AzureSpringBootRequestHandler`，并提供输入和输出类型作为类型参数（使Azure能够检查类并执行JSON转换本身）。

如果您的应用有`@Bean`类型`Function`等，则可以通过配置`function.name`来选择要使用的`function.name`。 这些功能是从Spring Cloud `FunctionCatalog`中提取的。

125.2.1 Notes on JAR Layout译：125.2.1关于JAR布局的注意事项

您在Azure运行时不需要Spring Cloud Function Web，因此您需要在创建部署到Azure的JAR之前将其排除。 Azure上的函数应用程序必须加阴影，但Spring Boot独立应用程序不会，因此您可以使用2个独立的jar（根据此示例）运行相同的应用程序。 示例应用程序创建阴影的jar文件，并使用`azure`分类器在Azure中进行部署。

125.2.2 JSON Configuration译：125.2.2 JSON配置

Azure工具需要找到一些JSON配置文件来告诉它如何部署和集成该功能（例如，哪个Java类用作入口点，以及使用哪个触发器）。 这些文件可以使用Maven插件为非Spring函数创建，但该工具对于当前形式的适配器还不起作用。 示例中有一个示例`function.json`，它将功能挂钩为HTTP端点：

```
{
  "scriptFile" : "../function-sample-azure-1.0.0.RELEASE-azure.jar",
  "entryPoint" : "example.FooHandler.execute",
  "bindings" : [ {
    "type" : "httpTrigger",
    "name" : "foo",
    "direction" : "in",
    "authLevel" : "anonymous",
    "methods" : [ "get", "post" ]
  }, {
    "type" : "http",
    "name" : "$return",
    "direction" : "out"
  } ],
  "disabled" : false
}
```

125.2.3 Build译：125.2.3构建

```
./mvnw -U clean package
```

125.2.4 Running the sample译：125.2.4运行样品

您可以在本地运行示例，就像其他Spring Cloud Function示例一样：

和 `curl -H "Content-Type: text/plain" localhost:8080/function -d '{"value": "hello foobar"}'` 。

您将需要 `az` CLI应用程序和一些node.js fu（有关更多详细信息，请参阅https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven ）。 在Azure运行时上部署该功能：

```
$ az login
$ mvn azure-functions:deploy
```

在另一个终端上试试这个： `curl https://<azure-function-url-from-the-log>/api/uppercase -d '{"value": "hello foobar!"}'` 。 请确保您使用上述功能的正确网址。 或者，您可以在Azure仪表板UI中测试函数（单击函数名称，右侧并单击“测试”，然后单击右下角的“运行”）。

Azure示例中函数的输入类型是一个具有名为“value”的单个属性的Foo。 所以你需要用下面的东西来测试它：

```
{
  "value": "foobar"
}
```

125.3 Apache Openwhisk译：125.3 Apache Openwhisk

OpenWhisk适配器采用可执行jar形式，可用于将Docker映像部署到Openwhisk。 该平台以请求响应模式工作，在特定端点上监听8080端口，因此该适配器是一个简单的Spring MVC应用程序。

125.3.1 Quick Start译：125.3.1快速入门

实施POF（请务必使用 `functions` 包装）：

```
package functions;

import java.util.function.Function;

public class Uppercase implements Function<String, String> {

 public String apply(String input) {
  return input.toUpperCase();
 }
}
```

将其安装到本地Maven存储库中：

```
./mvnw clean install
```

创建一个提供其Maven坐标的 `function.properties` 文件。 例如：

```
dependencies.function: com.example:pof:0.0.1-SNAPSHOT
```

将openwhisk runner JAR复制到工作目录（与属性文件相同的目录）：

```
cp spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk/target/spring-cloud-function-adapter-openwhisk-1.0.0.RELEASE.jar runner.jar
```

使用以上属性文件从跑步者JAR的 `--thin.dryrun` 中生成一个m2回购：

```
java -jar -Dthin.root=m2 runner.jar --thin.name=function --thin.dryrun
```

使用以下Dockerfile：

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
COPY m2 /m2
ADD runner.jar .
ADD function.properties .
ENV JAVA_OPTS=""
ENTRYPOINT [ "java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "runner.jar", "--thin.root=/m2", "--thin.name=function", "--function.name=upp
EXPOSE 8080
```

您可以使用Spring Cloud Function应用程序，而不是仅使用带有POF的jar，在这种情况下，您必须更改应用程序在容器中运行的方式，以便它将主类作为源文件提取。 例如，您可以更改上面的 `ENTRYPOINT` 并添加 `--spring.main.sources=com.example.SampleApplication` 。

构建Docker镜像：

```
docker build -t [username/appname] .
```

推送Docker镜像：

```
docker push [username/appname]
```

使用OpenWhisk CLI（例如，在 `vagrant ssh` 之后）创建操作：

```
wsk action create example --docker [username/appname]
```

调用该操作：

```
wsk action invoke example --result --param payload foo
{
    "result": "FOO"
}
```

Part XVII. Appendix: Compendium of Configuration Properties译:第十七部分。附录：配置属性大纲

Name	Default	描述
encrypt.fail-ON误差	真正	标志说如果存在加密或解密错误，则进程应该失败。
encrypt.key		对称密钥。 作为一个更强有力的选择，考虑使用密钥库。
encrypt.key-store.alias		商店中的密钥的别名。
encrypt.key-store.location		密钥存储文件的位置，例如classpath：/keystore.jks。
encrypt.key-store.password		锁定密钥库的密码。
encrypt.key-store.secret		保护密钥的秘密（默认与密码相同）。
encrypt.rsa.algorithm		使用RSA算法（DEFAULT或OEAP）。 一旦设置，不要更改它有的密码将不可解密）。
encrypt.rsa.salt	DEADBEEF	Salt用于加密密文的随机密码。 一旦设置，不要更改它（或现有码将不可解密）。
encrypt.rsa.strong	假	表明应在内部使用“强大的”AES加密标志。 如果为true，则将G法应用于AES加密字节。 默认值为false（在这种情况下使用“标准”CBC）。 一旦设置，不要更改它（或现有的密码将不可解密
encrypt.salt	DEADBEEF	对称密钥的盐以十六进制编码的字节数组的形式表示。 作为一有力的选择，考虑使用密钥库。
endpoints.zookeeper.enabled	真正	启用/ zookeeper端点来检查动物园管理员的状态。
eureka.client.allow-重定向	假	指示服务器是否可以将客户端请求重定向到备份服务器/群集。置为false，服务器将直接处理请求，如果设置为true，则可能会HTTP重定向发送到客户端，并带有新的服务器位置。
eureka.client.availability-区		获取此实例所在区域的可用区域列表（在AWS数据中心中使用 这些更改在运行时在registryFetchIntervalSeconds指定的下一个获取周期中生效。
eureka.client.backup-注册表IMPL		仅当第一次启动eureka客户端时，获取实现BackupRegistry以获册表信息作为回退选项的实现的名称。 对于注册表信息需要额外弹性的应用程序来说，这可能是需要的则它无法运行。
eureka.client.cache刷新执行人指数，回退绑定	10	缓存刷新执行程序指数退避相关属性。 对于发生超时序列的情况是重试延迟的最大乘数值。
eureka.client.cache刷新执行人线程池大小	2	用于初始化的cacheRefreshExecutor的线程池大小
eureka.client.client数据，接受		Eureka接受客户数据的接受名称
eureka.client.decoder名		这是一个暂时的配置，一旦最新的编解码器稳定，可以删除（已有一个）
eureka.client.disable-Δ	假	指示尤里卡客户端是否应禁用获取增量，并应该求助于获取完整册表信息。 请注意，增量提取可以极大地减少流量，因为尤里卡服务器的变通常远低于提取率。 这些更改在运行时在registryFetchIntervalSeconds指定的下一个获取周期中生效
eureka.client.dollar更换	_-	在eureka服务器中序列化/反序列化信息期间，获取美元符号<co</ code>的替换字符串。
eureka.client.enabled	真正	表示Eureka客户端已启用的标志。
eureka.client.encoder名		这是一个暂时的配置，一旦最新的编解码器稳定，可以删除（已有一个）

Name	Default	描述
eureka.client.escape-字符替换	__	在序列化/反序列化尤里卡服务器中的信息期间获取下划线符号< _ </ code>的替换字符串。
eureka.client.eureka连接空闲超时秒	三十	表示到尤里卡服务器的HTTP连接在可以关闭之前可保持空闲的（以秒为单位）。 在AWS环境中，建议值为30秒或更少，因为防火墙在几分钟后接信息，使连接处于闲置状态
eureka.client.eureka - 服务器连接超时秒	五	指示到尤里卡服务器的连接需要超时之前等待的时间（以秒为单位）。 请注意，客户端中的连接由org.apache.http.client.HttpCl并，并且此设置会影响实际的连接创建以及从池中获取连接的等间。
eureka.client.eureka服务器的DNS名称		获取要查询的DNS名称以获取尤里卡服务器的列表。如果合同现serviceUrls返回服务URL，则不需要此信息。 当useDnsForFetchingServiceUrls设置为true并且eureka客户端DNS以某种方式进行配置以便它可以动态获取更改的eureka服时，将使用DNS机制。 这些更改在运行时有效。
eureka.client.eureka服务器端口		当尤里卡服务器列表来自DNS时，获取用于构建服务URL以联卡服务器的端口。如果该合同返回服务URL eurekaServerServiceUrls（String），则不需要此信息。 当useDnsForFetchingServiceUrls设置为true并且eureka客户端DNS以某种方式进行配置以便它可以动态获取更改的eureka服时，将使用DNS机制。 这些更改在运行时有效。
eureka.client.eureka服务器读取超时秒	8	表示从尤里卡服务器读取超时之前需要等待多长时间（以秒为单位）。
eureka.client.eureka - 服务器 - 总的连接	200	获取从尤里卡客户端到所有尤里卡服务器允许的连接总数。
eureka.client.eureka - 服务器 - 总的连接每主机	50	获取从尤里卡客户端到尤里卡服务器主机所允许的连接总数。
eureka.client.eureka服务器的URL上下文		当尤里卡服务器列表来自DNS时，获取用于构建服务URL以联卡服务器的URL上下文。 如果合同从eurekaServerServiceUrls务URL，则不需要此信息。 当useDnsForFetchingServiceUrls设置为true并且eureka客户端DNS以某种方式进行配置以便它可以动态获取更改的eureka服时，将使用DNS机制。 这些更改在运行时有效。
eureka.client.eureka服务-URL轮询间隔秒	0	指示多久（以秒为单位）轮询尤里卡服务器信息的更改。 可以删除Eureka服务器，并且此设置控制尤里卡客户应该多久知道它
eureka.client.fetch-注册表	真正	指示此客户端是否应从eureka服务器获取eureka注册表信息。
eureka.client.fetch远程区域的注册表		以逗号分隔的尤里卡注册表信息将被提取的区域列表。 必须为域返回的每个区域定义可用区域。 如果不这样做，将导致发现启动失败。
eureka.client.filter只-UP-实例	真正	指示在仅针对InstanceStatus UP状态的实例过滤应用程序后是否应用程序。
eureka.client.g拉链内容	真正	指示服务器是否支持从eureka服务器获取的内容是否需要进行压来自尤里卡服务器的注册表信息被压缩以获得最佳网络流量。
eureka.client.heartbeat执行人指数，回退绑定	10	心跳执行者指数退还相关财产。 对于发生超时序列的情况，这延迟的最大乘数值。
eureka.client.heartbeat执行人线程池大小	2	heartbeatExecutor用于初始化的线程池大小
eureka.client.initial实例-信息复制间隔秒	40	指示最初（以秒为单位）将实例信息复制到尤里卡服务器的时间
eureka.client.instance-信息复制间隔秒	三十	指示复制实例更改以复制到尤里卡服务器的频率（以秒为单位）

Name	Default	描述
eureka.client.log-Δ-DIFF	假	指示是否根据注册表信息记录尤里卡服务器和尤里卡客户端之间 异。 尤里卡客户端试图只从尤里卡服务器检索增量变化以最小化网络 量。 收到增量后，eureka客户端协调来自服务器的信息，以验证 有遗漏一些信息。 当客户端发生网络问题与服务器通信时，可能 生协调失败。如果协调失败，则eureka客户端将获取完整的注册 息。 在获取完整注册表信息的同时，尤里卡客户端可以记录客户端和 器之间的差异，并且此设置控制着这一点。 这些更改在运行时在registryFetchIntervalSecondsr指定的下一个 获取周期中生效
eureka.client.on按需更新状态变化	真正	如果设置为true，则通过ApplicationInfoManager进行本地状态更 触发对远程eureka服务器的按需（但速率有限）注册/更新
eureka.client.prefer-同一区域 - 尤里卡	真正	指示此实例是否尝试在同一区域中使用尤里卡服务器出于延迟和 他原因。 理想情况下，尤里卡客户端被配置为与同一区域中的服务器通讯 这些更改在运行时在registryFetchIntervalSeconds指定的下一个 获取周期中生效
eureka.client.property - 解析器		
eureka.client.proxy主机		获取eureka服务器的代理主机（如果有的话）。
eureka.client.proxy密码		获取代理密码（如果有）。
eureka.client.proxy端口		获取尤里卡服务器的代理端口（如果有的话）。
eureka.client.proxy用户名		获取代理用户名（如果有）。
eureka.client.region	美国 - 东 - 1	获取此实例所在的区域（在AWS数据中心中使用）。
eureka.client.register与 - 尤里卡	真正	指示此实例是否应将其信息注册到尤里卡服务器以供其他人发现 在某些情况下，您不希望发现您的实例，而只想发现其他实例。
eureka.client.registry取间隔秒	三十	指示从eureka服务器获取注册表信息的频率（以秒为单位）。
eureka.client.registry刷新单VIP地址		指示客户端是否只对单个VIP的注册表信息感兴趣。
eureka.client.service-URL		将可用区映射到完全限定URL列表以与尤里卡服务器进行通信。 值可以是单个URL或以逗号分隔的其他位置列表。 通常情况下，尤里卡服务器URL包含协议，主机，端口，上下文 本信息（如果有的话）。 例如： http://ec2-256-156-243-129.co 1.amazonaws.com:7001/eureka/ 这些更改在运行时在eurekaServiceUrlPollIntervalSeconds指定的 个服务URL刷新周期中生效。
eureka.client.should-执行登记-AT-INIT	假	指示客户端是否应在初始化期间执行注册。 默认为false。
eureka.client.should - 注销 - 在关机	真正	指示在客户端关闭时客户端是否应显式从远程服务器注销自己。
eureka.client.use-DNS-用于取服务的URL	假	指示尤里卡客户端是否应使用DNS机制来获取要与之通话的尤里 务器列表。 当DNS名称更新为具有其他服务器时，在eureka客户 询eurekaServiceUrlPollIntervalSeconds中指定的信息后立即使用 息。 或者，可以将服务URL返回serviceUrls，但用户应该实现自己的 在发生更改时返回更新的列表。 这些更改在运行时有效。
eureka.dashboard.enabled	真正	启用Eureka仪表板的标志。 默认为true。
eureka.dashboard.path	/	Eureka仪表板的路径（相对于servlet路径）。 默认为“/”。
eureka.instance.asg名		获取与此实例关联的AWS自动调整组名称。 此信息专门用于AW 境，以便在实例启动后自动将实例置于服务状态，并且已针对消 用该实例。
eureka.instance.app组名		获取要使用eureka注册的应用程序组的名称。
eureka.instance.appname	未知	获取要使用eureka注册的应用程序的名称。

Name	Default	描述
eureka.instance.data中心，信息		返回此实例部署的数据中心。 如果实例部署在AWS中，则使用获取一些AWS特定的实例信息。
eureka.instance.default地址解析顺序	[]	
eureka.instance.environment		
eureka.instance.health - 检查 - 网址		获取此实例的绝对健康检查页面URL。 如果运行状况检查页驻留eureka交谈的同一实例中，则用户可以提供healthCheckUrlPath在实例是某个其他服务器的代理的情况下，用户可以提供完整的URL。 如果提供完整的URL，则优先。<p>它通常用于根据实例的健康状况作出明智的决定 - 例如，它确定是继续部署到整个农场还是停止部署而不会造成进一步的完整的URL应遵循格式http://${eureka.hostname}:7001/，其中{eureka.hostname}在运行时被替换。
eureka.instance.health检查的URL路径		获取此实例的相对健康检查URL路径。 运行状况检查页面URL名和通信类型组成 - 按securePort和nonSecurePort中指定的安全安全。 它通常用于根据实例的健康状况作出合理的决定 - 例如，它可用是继续部署到整个农场还是停止部署而不会造成进一步的损害。
eureka.instance.home页的URL		获取此实例的绝对主页URL。 如果主页驻留在与eureka交谈的例中，则用户可以提供homePageUrlPath，否则在实例是某个务器的代理的情况下，用户可以提供完整的URL。 如果提供完URL，则优先。 它通常用于其他服务的信息目的，以将其用作登录页面。 完整应遵循格式http://${eureka.hostname}:7001/，其中${eureka.hostname}的值在运行时被替换。
eureka.instance.home页的URL路径	/	获取此实例的相对主页URL路径。 然后，主页URL由hostName类型组成 - 安全或不安全。 它通常用于其他服务的信息目的，以将其用作登录页面。
eureka.instance.hostname		主机名如果可以在配置时确定（否则会从OS原语中猜出）。
eureka.instance.initial状态		注册rmeote Eureka服务器的初始状态。
启用eureka.instance.instance-，在它身上	假	指示是否应该启用实例以便在通过eureka注册后立即获取流量。应用程序可能需要做一些预处理，然后才准备好接收流量。
eureka.instance.instance-ID		获取此实例的独特Id（在appName的范围内），以便使用eurek注册。
eureka.instance.ip地址		获取实例的IP地址。 此信息仅用于学术目的，因为来自其他实例信主要通过{@link #getHostName（boolean）}中提供的信息进
eureka.instance.lease过期持续时间 - 在秒	90	指示尤里卡服务器从它的视图中删除此实例之前接收到最后一次的等待时间，以秒为单位，并且不允许此实例的流量。 将此值设置得太长可能意味着即使实例未处于活动状态，流量也会路由到实例。 将此值设置得太小可能意味着，由于临时网络实例可能会被取消流量。此值将设置为至少高于leaseRenewalIntervalInSeconds中指定的值。
eureka.instance.lease更新间隔-在秒	三十	指示尤里卡客户端需要多长时间（以秒为单位）向eureka服务器检测信号以表明它仍然存在。 如果在leaseExpirationDurationInSeconds中指定的时间段内没有收到检号，则尤里卡服务器将从其视图中删除该实例，从而禁止该实例量。 请注意，如果实例实施HealthCheckCallback，然后决定使其本用，则该实例仍可能无法访问流量。
eureka.instance.metadata地图		获取与此实例关联的元数据名称/值对。 这些信息被发送到eurek器，并且可以被其他实例使用。
eureka.instance.namespace	尤里卡	获取用于查找属性的名称空间。 在春云中忽略。
eureka.instance.non安全港	80	获取实例应该接收流量的非安全端口。
启用eureka.instance.non安全端口，	真正	指示是否应为流量启用非安全端口。
eureka.instance.prefer-IP地址	假	标记说，当猜测主机名时，服务器的IP地址应该用于操作系统报主机名。
eureka.instance.registry.default开换流数	1	确定何时取消租赁时使用的值，对于独立，默认值为1。 对于同的eurekas应该设置为0

Name	Default	描述
eureka.instance.registry.expected-数的-更新每分钟	1	
eureka.instance.secure健康检查，网址		获取此实例的绝对安全的运行状况检查页面URL。 如果运行状况页驻留在与尤里卡交谈的同一实例中，则用户可以提供secureHealthCheckUrl，否则在实例是某个其他服务器的代理的情况下，用户可以提供完整的URL。 如果提供完整的URL，则优先。 <p>它通常用于根据实例的健康状况作出明智的决定 - 例如，它确定是继续部署到整个农场还是停止部署而不会造成进一步的损完整的URL应遵循格式http://${eureka.hostname}:7001/，其中${eureka.hostname}的值在运行时被替换。
eureka.instance.secure端口	443	获取实例应该接收流量的安全端口。
启用eureka.instance.secure端口，	假	指示是否应为流量启用安全端口。
eureka.instance.secure虚拟主机名	未知	获取为此实例定义的安全虚拟主机名。 这通常是其他实例通过使用安全虚拟主机名称来查找此实例的方认为这与完全限定的域名类似，您的服务的用户将需要找到此实
eureka.instance.status页的URL		获取此实例的绝对状态页面URL路径。 如果状态页驻留在与eur谈的同一实例中，则用户可以提供statusPageUrlPath，否则在某个其他服务器的代理的情况下，用户可以提供完整的URL。 如供完整的URL，则优先。 它通常用于其他服务的信息目的，以查找此实例的状态。 用户可供一个简单的HTML来指示实例的当前状态。
eureka.instance.status页的URL路径		获取此实例的相对状态页面URL路径。 然后状态页面URL由hos和通信类型组成 - 根据securePort和nonSecurePort中指定的安全安全。 它通常用于其他服务的信息目的，以查找此实例的状态。 用户可供一个简单的HTML来指示实例的当前状态。
eureka.instance.virtual主机名	未知	获取为此实例定义的虚拟主机名。 这通常是其他实例通过使用虚拟主机名称来查找此实例的方式。这与完全限定的域名类似，您的服务的用户将需要找到此实例。
eureka.server.asg-缓存到期超时毫秒	0	
eureka.server.asg查询超时毫秒	300	
eureka.server.asg更新间隔毫秒	0	
eureka.server.aws访问-ID		
eureka.server.aws秘密密钥		
eureka.server.batch复制	假	
eureka.server.binding策略		
eureka.server.delta保留定时器间隔-在毫秒	0	
eureka.server.disable-Δ	假	
eureka.server.disable-Δ换远程区域	假	
eureka.server.disable透明-后备到其他区域	假	
eureka.server.eip绑定-重新绑定-重试	3	
eureka.server.eip结合重试间隔毫秒	0	
eureka.server.eip结合重试间隔-MS-当非结合	0	
eureka.server.enable复制的请求压缩	假	
eureka.server.enable-自保	真正	
eureka.server.eviction间隔定时器-在毫秒	0	
eureka.server.g-拉链内容从远程区域	真正	
eureka.server.json编解码器名称		

Name	Default	描述
eureka.server.list-自动缩放 - 组 - 角色名	ListAutoScalingGroups	
eureka.server.log身份报头	真正	
eureka.server.max元素-在等复制池	10000	
eureka.server.max元素，在状态复制池	10000	
eureka.server.max空闲线程年龄在分钟换同行复制	15	
eureka.server.max空闲线程的 - 分 - 年龄的状态复制	10	
eureka.server.max线程换同行复制	20	
eureka.server.max线程换状态复制	1	
eureka.server.max时间换复制	30000	
eureka.server.min可用-情况下，对等复制	-1	
eureka.server.min线程换同行复制	五	
eureka.server.min线程换状态复制	1	
eureka.server.number-的复制重试次数	五	
eureka.server.peer-尤里卡节点更新间隔毫秒	0	
eureka.server.peer - 尤里卡状态刷新时间间隔毫秒	0	
eureka.server.peer节点，连接超时毫秒	200	
eureka.server.peer节点连接空闲超时秒	三十	
eureka.server.peer节点读取超时毫秒	200	
eureka.server.peer节点与总连接	1000	
eureka.server.peer节点与总连接 - 每个主机	500	
eureka.server.prime-AWS-复制品的连接	真正	
eureka.server.property - 解析器		
eureka.server.rate限制器突发大小	10	
启用eureka.server.rate限制器，	假	
eureka.server.rate限制器全取平均利率	100	
eureka.server.rate限制器特权的客户端		
eureka.server.rate限制器的注册表取平均利率	500	
eureka.server.rate限制器油门标准的客户端	假	
eureka.server.registry个同步重试	0	
eureka.server.registry同步重试等待毫秒	0	
eureka.server.remote区域-APP-白名单		
eureka.server.remote区域，连接超时毫秒	1000	
eureka.server.remote区域连接空闲超时秒	三十	
eureka.server.remote区域取线程池大小	20	
eureka.server.remote区域读取超时毫秒	1000	
eureka.server.remote区域的注册表取入间隔	三十	
eureka.server.remote区域与总连接	1000	
eureka.server.remote区域与总连接 - 每个主机	500	

Name	Default	描述
eureka.server.remote区域信任店		
eureka.server.remote区域垄断店内密码	更改	
eureka.server.remote区域的URL		
eureka.server.remote区域的URL与 - 名		
eureka.server.renewal％的阈值	0.85	
eureka.server.renewal阈值更新间隔毫秒	0	
eureka.server.response缓存，自动失效，以秒	180	
eureka.server.response缓存更新间隔毫秒	0	
eureka.server.retention - 时间 - 在-MS-在-Δ-队列	0	
eureka.server.route53绑定-重新绑定-重试	3	
eureka.server.route53结合重试间隔毫秒	0	
eureka.server.route53域-TTL	三十	
eureka.server.sync-当时间戳-不同	真正	
eureka.server.use-只读响应缓存	真正	
eureka.server.wait时，在MS-时同步空	0	
eureka.server.xml编解码器名称		
health.config.enabled	假	标记以指示应该安装配置服务器运行状况指示器。
health.config.time生存	0	生存缓存结果的时间，以毫秒为单位。 默认300000（5分钟）。
hystrix.metrics.enabled	真正	启用Hystrix度量轮询。 默认为true。
hystrix.metrics.polling间隔毫秒	2000	后续轮询指标之间的时间间隔。 默认为2000毫秒。
management.endpoint.bindings.cache.time生存	0毫秒	可以缓存响应的最长时间。
management.endpoint.bindings.enabled	真正	是否启用绑定端点。
management.endpoint.bus-env.enabled	真正	是否启用总线环境端点。
management.endpoint.bus-refresh.enabled	真正	是否启用总线刷新端点。
management.endpoint.channels.cache.time生存	0毫秒	可以缓存响应的最长时间。
management.endpoint.channels.enabled	真正	是否启用通道端点。
management.endpoint.consul.cache.time生存	0毫秒	可以缓存响应的最长时间。
management.endpoint.consul.enabled	真正	是否启用领事终点。
management.endpoint.env.post.enabled	真正	启用通过POST更改环境到/ env。
management.endpoint.features.cache.time生存	0毫秒	可以缓存响应的最长时间。
management.endpoint.features.enabled	真正	是否启用功能端点。
management.endpoint.gateway.enabled	真正	是否启用网关端点。
management.endpoint.hystrix.config		Hystrix设置。 这些传统上使用servlet参数进行设置。 有关更多i息，请参阅Hystrix的文档。
management.endpoint.hystrix.stream.enabled	真正	是否启用hystrix.stream端点。
management.endpoint.pause.enabled	真正	启用/暂停端点（发送Lifecycle.stop（））。
management.endpoint.refresh.enabled	真正	启用/ refresh端点来刷新配置并重新初始化刷新范围的bean。
management.endpoint.restart.enabled	真正	启用/重新启动端点以重新启动应用程序上下文。
management.endpoint.resume.enabled	真正	启用/恢复端点（发送Lifecycle.start（））。

Name	Default	描述
management.endpoint.service-registry.cache.time生存	0毫秒	可以缓存响应的最长时间。
management.endpoint.service-registry.enabled	真正	是否启用服务注册表端点。
management.health.refresh.enabled	真正	为刷新范围启用健康端点。
management.health.zookeeper.enabled	真正	为zookeeper启用运行状况端点。
proxy.auth.load平衡	假	
proxy.auth.routes		每个路由的认证策略。
ribbon.eager-load.clients		
ribbon.eager-load.enabled	假	
ribbon.secure端口		
spring.cloud.bus.ack.destination服务		希望听到acks的服务。 默认情况下为null（表示所有服务）。
spring.cloud.bus.ack.enabled	真正	标志关闭ack（默认打开）。
spring.cloud.bus.destination	springCloudBus	消息的Spring云流目标的名称。
spring.cloud.bus.enabled	真正	标记以指示总线已启用。
spring.cloud.bus.env.enabled	真正	标志关闭环境变化事件（默认开启）。
spring.cloud.bus.id	应用	此应用程序实例的标识符。
spring.cloud.bus.refresh.enabled	真正	标志关闭刷新事件（默认开启）。
spring.cloud.bus.trace.enabled	假	标记打开acks的追踪（默认关闭）。
spring.cloud.cloudfoundry.discovery.default服务器端口	80	没有端口由功能区定义时使用的端口。
spring.cloud.cloudfoundry.discovery.enabled	真正	标记以指示已启用发现。
spring.cloud.cloudfoundry.discovery.heartbeat频	5000	心跳频率以毫秒为单位。 客户端将在此频率上进行轮询并广播 列表。
spring.cloud.cloudfoundry.org		最初定位的组织名称。
spring.cloud.cloudfoundry.password		用户验证和获取令牌的密码。
spring.cloud.cloudfoundry.skip的SSL验证	假	
spring.cloud.cloudfoundry.space		空间名称最初的目标。
spring.cloud.cloudfoundry.url		Cloud Foundry API（云控制器）的URL。
spring.cloud.cloudfoundry.username		用户名来验证（通常是一个电子邮件地址）。
spring.cloud.config.allow，覆盖	真正	表示可以使用{@link #isOverrideSystemProperties（）systemPropertiesOverride}的 设置为false可防止用户意外更改默认设置。 默认为true。
spring.cloud.config.discovery.enabled	假	标记以指示配置服务器发现已启用（配置服务器URL将通过发现 找）。
spring.cloud.config.discovery.service-ID	configserver	服务ID找到配置服务器。
spring.cloud.config.enabled	真正	标志说远程配置已启用。 默认为true;
spring.cloud.config.fail快	假	表示连接到服务器失败的标志是致命的（默认为false）。
spring.cloud.config.headers		用于创建客户端请求的其他标头。
spring.cloud.config.label		用于提取远程配置属性的标签名称。 默认值是在服务器上设置 常是基于git的服务器的“master”）。
spring.cloud.config.name		用于提取远程属性的应用程序的名称。

Name	Default	描述
spring.cloud.config.override，无	假	表示当{@link #setAllowOverride（boolean）allowOverride}为tru 外部属性应该具有最低优先级，并且不会覆盖任何现有属性源 本地配置文件）。 默认为false。
spring.cloud.config.override系统的属性	真正	标志以指示外部属性应该覆盖系统属性。 默认为true。
spring.cloud.config.password		联系远程服务器时使用的密码（HTTP Basic）。
spring.cloud.config.profile	默认	获取远程配置时使用的默认配置文件（以逗号分隔）。 默认是“ 认”。
spring.cloud.config.request读取超时	0	等待从配置服务器读取数据超时。
spring.cloud.config.retry.initial间隔	1000	初始重试间隔，以毫秒为单位。
spring.cloud.config.retry.max-尝试	6	最大尝试次数。
spring.cloud.config.retry.max间隔	2000	最大退避时间间隔。
spring.cloud.config.retry.multiplier	1.1	下一个时间间隔的乘数。
spring.cloud.config.send状态	真正	标记以指示是否发送状态。 默认为true。
spring.cloud.config.server.accept空	真正	标记以指示如果未找到应用程序，则需要发送HTTP 404
spring.cloud.config.server.bootstrap	假	指示配置服务器应该使用来自远程存储库的属性初始化其自己的 的标志。 默认情况下关闭，因为它会延迟启动，但在将服务器 另一个应用程序时可能会有用。
spring.cloud.config.server.default应用程序名称	应用	传入请求没有特定的应用程序名称时的默认应用程序名称。
spring.cloud.config.server.default标签		传入请求没有特定标签时的默认存储库标签。
spring.cloud.config.server.default瞩目	默认	传入请求中没有特定的应用程序配置文件
spring.cloud.config.server.encrypt.enabled	真正	在发送给客户端之前启用环境属性的解密。
spring.cloud.config.server.git.basedir		存储库本地工作副本的基本目录。
spring.cloud.config.server.git.clone上启动	假	标记以指示存储库应在启动时克隆（不按需）。 通常会导致启 较慢但第一次查询速度较快
spring.cloud.config.server.git.default标签		将与remore存储库一起使用的默认标签
spring.cloud.config.server.git.delete-未跟踪分支	假	标志表示如果分支的原始追踪分支已被移除，应将分支本地删除
spring.cloud.config.server.git.force拉	假	标记以指示存储库应该强制拉取。 如果true放弃任何本地更改并 程存储库获取。
spring.cloud.config.server.git.host键		有效的SSH主机密钥。 如果还设置了hostKeyAlgorithm，则必须 置。
spring.cloud.config.server.git.host密钥算法		ssh-dss，ssh-rsa，ecdsa-sha2-nistp256，ecdsa-sha2-nistp384 ecdsa-sha2-nistp521之一。 如果hostKey也被设置，必须设置。
spring.cloud.config.server.git.ignore本地-SSH的设置	假	如果为true，则使用基于属性而不是基于文件的SSH配置。
spring.cloud.config.server.git.known的主机文件		自定义.known_hosts文件的位置。
spring.cloud.config.server.git.order		环境存储库的顺序。
spring.cloud.config.server.git.passphrase		用于解锁您的ssh私钥的密码。
spring.cloud.config.server.git.password		用于远程存储库认证的密码。
spring.cloud.config.server.git.preferred-认证		覆盖服务器认证方法的顺序。 如果服务器在publickey方法之前 键盘交互式身份验证，这应该允许避免登录提示。
spring.cloud.config.server.git.private键		有效的SSH私钥。 如果ignoreLocalSshSettings为true且Git URI 格式，则必须设置。
spring.cloud.config.server.git.proxy		HTTP代理配置。
spring.cloud.config.server.git.refresh率	0	刷新git存储库之间的时间（以秒为单位）
spring.cloud.config.server.git.repos		存储库标识符到位置和其他属性的映射。

Name	Default	描述
spring.cloud.config.server.git.search路径		搜索在本地工作副本中使用的路径。 默认情况下只搜索根。
spring.cloud.config.server.git.skip的SSL验证	假	标记以指示在与通过HTTPS连接提供服务的存储库进行通信时，绕过SSL证书验证。
spring.cloud.config.server.git.strict主机密钥检查	真正	如果为false，则忽略主机密钥的错误
spring.cloud.config.server.git.timeout	五	用于获取HTTP或SSH连接的超时（以秒为单位）（如果适用）为5秒。
spring.cloud.config.server.git.uri		远程存储库的URI。
spring.cloud.config.server.git.username		用于远程存储库认证的用户名。
spring.cloud.config.server.health.repositories		
spring.cloud.config.server.jdbc.order	0	
spring.cloud.config.server.jdbc.sql	选择键，值来自PROPERTIES其中APPLICATION =？ 和PROFILE =？ 和LABEL =?	SQL用于查询数据库中的键和值
spring.cloud.config.server.native.add标签，位置	真正	标记以确定是否应添加标签位置。
spring.cloud.config.server.native.default标签	主	
spring.cloud.config.server.native.fail-ON误差	假	标记以确定在解密期间如何处理异常（默认为false）。
spring.cloud.config.server.native.order		
spring.cloud.config.server.native.search-位置	[]	搜索配置文件的位置。 默认为与Spring Boot应用相同，所以[classpath：/，classpath：/ config /，file：./，file：./ config /]。
spring.cloud.config.server.native.version		要为本地存储库报告的版本字符串
spring.cloud.config.server.overrides		额外的地图资源来无条件发送给所有客户。
spring.cloud.config.server.prefix		配置资源路径的前缀（默认为空）。 当您不想更改上下文路径或servlet路径时，在嵌入另一个应用程序时非常有用。
spring.cloud.config.server.strip文档从 - YAML	真正	表示YAML文档是文本或集合（不是地图）的标志应以“本机”形式回。
spring.cloud.config.server.svn.basedir		存储库本地工作副本的基本目录。
spring.cloud.config.server.svn.default标签		将与remore存储库一起使用的默认标签
spring.cloud.config.server.svn.order		环境存储库的顺序。
spring.cloud.config.server.svn.passphrase		用于解锁您的ssh私钥的密码。
spring.cloud.config.server.svn.password		用于远程存储库认证的密码。
spring.cloud.config.server.svn.search路径		搜索在本地工作副本中使用的路径。 默认情况下只搜索根。
spring.cloud.config.server.svn.strict主机密钥检查	真正	拒绝来自不在已知主机列表中的远程服务器的传入SSH主机密钥
spring.cloud.config.server.svn.uri		远程存储库的URI。
spring.cloud.config.server.svn.username		用于远程存储库认证的用户名。
spring.cloud.config.server.vault.backend	秘密	保险库后端。 默认为秘密。
spring.cloud.config.server.vault.default键	应用	所有应用程序共享的保管库中的密钥。 默认为应用程序。 设置禁用。
spring.cloud.config.server.vault.host	127.0.0.1	电子仓库主机。 默认为127.0.0.1。
spring.cloud.config.server.vault.kv版本	1	指示使用哪个版本的Vault kv后端的值。 默认为1。
spring.cloud.config.server.vault.order		
spring.cloud.config.server.vault.port	8200	Vault端口。 默认为8200。
spring.cloud.config.server.vault.profile分离器	,	Vault配置文件分隔符。 默认为逗号。
spring.cloud.config.server.vault.proxy		HTTP代理配置。
spring.cloud.config.server.vault.scheme	HTTP	保险计划。 默认为http。

Name	Default	描述
spring.cloud.config.server.vault.skip的SSL验证	假	标记以指示在与通过HTTPS连接提供服务的存储库进行通信时，绕过SSL证书验证。
spring.cloud.config.server.vault.timeout	五	用于获取HTTP连接的超时（以秒为单位），默认为5秒。
spring.cloud.config.token		安全令牌传递给底层环境存储库。
spring.cloud.config.uri	[http://localhost:8888]	远程服务器的URI（默认为 http://localhost:8888 ）。
spring.cloud.config.username		联系远程服务器时使用的用户名（HTTP Basic）。
spring.cloud.consul.config.acl令牌		
spring.cloud.consul.config.data键	数据	如果格式为Format.PROPERTIES或Format.YAML，则以下字段找consul进行配置的关键字。
spring.cloud.consul.config.default上下文	应用	
spring.cloud.consul.config.enabled	真正	
spring.cloud.consul.config.fail快	真正	如果为真，则在配置查找期间抛出异常，否则会记录警告。
spring.cloud.consul.config.format		
spring.cloud.consul.config.name		替代spring.application.name用于查找领事KV中的值。
spring.cloud.consul.config.prefix	配置	
spring.cloud.consul.config.profile分离器	,	
spring.cloud.consul.config.watch.delay	1000	毫表中的固定延迟值。 默认为1000。
spring.cloud.consul.config.watch.enabled	真正	如果手表已启用。 默认为true。
spring.cloud.consul.config.watch.wait时间	55	查询等待（或阻止）的秒数默认为55.需要小于默认的ConsulClient（默认为60）。 要增加ConsulClient超时，请使用定义HttpClient的自定义ConsulRawClient创建一个ConsulClient
spring.cloud.consul.discovery.acl令牌		
spring.cloud.consul.discovery.catalog服务手表延迟	1000	在millis中查看领事目录的电话之间的延迟，默认为1000。
spring.cloud.consul.discovery.catalog服务手表超时	2	观看领事目录时屏蔽的秒数，默认为2。
spring.cloud.consul.discovery.datacenters		在服务器列表中查询serviceId的s数据中心的地图。 这允许在另据中心查找服务。
spring.cloud.consul.discovery.default查询标签		如果没有在serverListQueryTags中列出，则在服务列表中查询标
spring.cloud.consul.discovery.default区的元数据名称	区	服务实例区域来自元数据。 这允许更改元数据标签名称。
spring.cloud.consul.discovery.deregister	真正	禁用领事自动注销服务。
spring.cloud.consul.discovery.enabled	真正	是否启用服务发现？
spring.cloud.consul.discovery.fail快	真正	如果服务注册期间为真，则抛出异常，否则，记录警告（默认为true）。
spring.cloud.consul.discovery.health检查临界超时		注销超时的关键服务超时（例如30米）超时。 要求领事版本7.x高。
spring.cloud.consul.discovery.health检查间隔	10S	执行健康检查的频率（例如10s），默认为10s。
spring.cloud.consul.discovery.health检查路径	/执行器/健康	要调用健康检查的替代服务器路径
spring.cloud.consul.discovery.health检查超时		健康检查超时（例如10秒）。
spring.cloud.consul.discovery.health检查-TLS-跳过验证		在服务检查期间跳过证书验证是否为真，否则运行证书验证。
spring.cloud.consul.discovery.health - 检查 - 网址		自定义健康检查网址可覆盖默认值
spring.cloud.consul.discovery.heartbeat.enabled	假	
spring.cloud.consul.discovery.heartbeat.interval比		

Name	Default	描述
spring.cloud.consul.discovery.heartbeat.ttl单元	小号	
spring.cloud.consul.discovery.heartbeat.ttl价值	三十	
spring.cloud.consul.discovery.hostname		访问服务器时使用的主机名
spring.cloud.consul.discovery.instance组		服务实例组
spring.cloud.consul.discovery.instance-ID		唯一的服务实例ID
spring.cloud.consul.discovery.instance区		服务实例区域
spring.cloud.consul.discovery.ip地址		访问服务时使用的IP地址（还必须设置preferIpAddress才能使用
spring.cloud.consul.discovery.lifecycle.enabled	真正	
spring.cloud.consul.discovery.management端口		下注册管理服务的端口（默认为管理端口）
spring.cloud.consul.discovery.management后缀	管理	注册管理服务时使用后缀
spring.cloud.consul.discovery.management标签		注册管理服务时使用的标签
spring.cloud.consul.discovery.port		端口下注册服务（默认为监听端口）
spring.cloud.consul.discovery.prefer剂地址	假	我们将如何确定要使用的地址
spring.cloud.consul.discovery.prefer-IP地址	假	注册期间使用ip地址而不是主机名
spring.cloud.consul.discovery.query扯皮	假	将'传递'参数添加到/ v1 / health / service / serviceName。 这将使 传递到服务器。
spring.cloud.consul.discovery.register	真正	在领事注册为服务。
spring.cloud.consul.discovery.register健康检查	真正	在领事注册健康检查。 在开发服务时有用。
spring.cloud.consul.discovery.scheme	HTTP	是否注册http或https服务
spring.cloud.consul.discovery.server列表查询标签		在服务器列表中查询serviceId的s标签的地图。 这允许通过单个 过滤服务。
spring.cloud.consul.discovery.service名		服务名称
spring.cloud.consul.discovery.tags		注册服务时使用的标签
spring.cloud.consul.enabled	真正	是春天的云领事启用
spring.cloud.consul.host	本地主机	Consul代理主机名。 默认为'localhost'。
spring.cloud.consul.port	8500	Consul代理端口。 默认为'8500'。
spring.cloud.consul.retry.initial间隔	1000	初始重试间隔，以毫秒为单位。
spring.cloud.consul.retry.max-尝试	6	最大尝试次数。
spring.cloud.consul.retry.max间隔	2000	最大退避时间间隔。
spring.cloud.consul.retry.multiplier	1.1	下一个时间间隔的乘数。
spring.cloud.consul.scheme		Consul代理方案（HTTP / HTTPS）。 如果地址中没有方案 - 客 使用HTTP。
spring.cloud.discovery.client.health-indicator.enabled	真正	
spring.cloud.discovery.client.health-indicator.include-描述	假	
spring.cloud.discovery.client.simple.instances		
spring.cloud.discovery.client.simple.local.metadata		服务实例的元数据。 发现客户端可以使用它来修改每个实例的行 例如负载平衡时。
spring.cloud.discovery.client.simple.local.service-ID		服务的标识符或名称。 多个实例可能共享相同的服务ID。
spring.cloud.discovery.client.simple.local.uri		服务实例的URI。 将解析提取该计划，主机和端口。
spring.cloud.gateway.default过滤器		应用于每条路线的过滤器定义列表。

Name	Default	描述
spring.cloud.gateway.discovery.locator.enabled	假	启用DiscoveryClient网关集成的标志
spring.cloud.gateway.discovery.locator.filters		
spring.cloud.gateway.discovery.locator.include表达	真正	SpEL表达式将评估是否在网关集成中包含服务，默认为：true
spring.cloud.gateway.discovery.locator.lower情况下，服务ID	假	谓词和过滤器中的小写serviceId选项，默认为false。 自动大写serviceId时适用于尤里卡。 所以MYSERIVCE会匹配/ myservice
spring.cloud.gateway.discovery.locator.predicates		
spring.cloud.gateway.discovery.locator.route-ID前缀		routeId的前缀默认为discoveryClient.getClass（）。getSimpleName（）+“_”。 服务ID将被追加以创建routeId。
spring.cloud.gateway.discovery.locator.url表达	'LB：//' +服务Id	创建每个路由的uri的SpEL表达式默认为：'lb：//'+ serviceId
spring.cloud.gateway.filter.remove-逐hop.headers		
spring.cloud.gateway.filter.remove-逐hop.order		
spring.cloud.gateway.filter.secure-headers.content安全政策	default-src'self'https :; font-src'self'https：data :; img-src'self'https：data :; object-src'none'; script-src https :; style-src'self'https：'unsafe-inline'	
spring.cloud.gateway.filter.secure-headers.content型选项	nosniff	
spring.cloud.gateway.filter.secure-headers.download选项	noopen	
spring.cloud.gateway.filter.secure-headers.frame选项	拒绝	
spring.cloud.gateway.filter.secure-headers.permitted交叉域的策略	没有	
spring.cloud.gateway.filter.secure-headers.referrer政策	没有引用	
spring.cloud.gateway.filter.secure-headers.strict运输安全	最大年龄= 631138519	
spring.cloud.gateway.filter.secure-headers.xss保护头	1; 模式=块	
spring.cloud.gateway.httpclient.connect超时		毫秒中的连接超时，默认为45秒。
spring.cloud.gateway.httpclient.pool.acquire超时		仅适用于FIXED类型，以millis等待获取的最长时间。
spring.cloud.gateway.httpclient.pool.max的连接		仅适用于FIXED类型，即在启动现有数据的暂挂获取之前的最大数。
spring.cloud.gateway.httpclient.pool.name	代理	通道池映射名称默认为代理。
spring.cloud.gateway.httpclient.pool.type		HttpClient使用的池的类型，默认为ELASTIC。
spring.cloud.gateway.httpclient.proxy.host		Netty HttpClient的代理配置主机名。
spring.cloud.gateway.httpclient.proxy.non代理的主机模式		正则表达式（Java），用于绕过代理直接到达的已配置主机列表
spring.cloud.gateway.httpclient.proxy.password		Netty HttpClient的代理配置密码。
spring.cloud.gateway.httpclient.proxy.port		Netty HttpClient的代理配置端口。
spring.cloud.gateway.httpclient.proxy.username		Netty HttpClient的代理配置的用户名。
spring.cloud.gateway.httpclient.ssl.use不安全信托经理	假	安装netty InsecureTrustManagerFactory。 这是不安全的，不适产。
spring.cloud.gateway.proxy.headers		修复了将被添加到所有下游请求的标头值。
spring.cloud.gateway.proxy.sensitive		默认情况下不会向下游发送的一组敏感标头名称。
spring.cloud.gateway.redis速率-limiter.burst容量头	X-RateLimit连拍容量	返回突发容量配置的标头的名称。
spring.cloud.gateway.redis率，limiter.config		

Name	Default	描述
spring.cloud.gateway.redis率-limiter.include报头	真正	是否包含包含速率限制器信息的标题，默认为true。
spring.cloud.gateway.redis速率-limiter.remaining头	X-RateLimit残留	在当前秒期间返回剩余请求数量的标题名称。
spring.cloud.gateway.redis速率-limiter.replenish率报头	X-RateLimit-补货速率	返回补充率配置的标题的名称。
spring.cloud.gateway.routes		路线列表
spring.cloud.gateway.streaming媒体类型		
spring.cloud.gateway.x-forwarded.enabled	真正	如果启用了XForwardedHeadersFilter。
spring.cloud.gateway.x-forwarded.for，追加	真正	如果附加X-Forwarded-For作为列表已启用。
启用spring.cloud.gateway.x-forwarded.for -	真正	如果启用了X-Forwarded-For。
spring.cloud.gateway.x-forwarded.host，追加	真正	如果将X-Forwarded-Host附加为列表已启用。
启用spring.cloud.gateway.x-forwarded.host -	真正	如果启用了X-Forwarded-Host。
spring.cloud.gateway.x-forwarded.order	0	XForwardedHeadersFilter的顺序。
spring.cloud.gateway.x-forwarded.port，追加	真正	如果将X-Forwarded-Port附加为列表已启用。
启用spring.cloud.gateway.x-forwarded.port -	真正	如果X-Forwarded-Port已启用。
spring.cloud.gateway.x-forwarded.proto，追加	真正	如果将X-Forwarded-Proto附加为列表已启用。
启用spring.cloud.gateway.x-forwarded.proto -	真正	如果X-Forwarded-Proto已启用。
spring.cloud.hypermedia.refresh.fixed延迟	5000	
spring.cloud.hypermedia.refresh.initial延迟	10000	
spring.cloud.inetutils.default主机名	本地主机	默认主机名。 在出现错误的情况下使用。
spring.cloud.inetutils.default-IP地址	127.0.0.1	默认的ipaddress。 在出现错误的情况下使用。
spring.cloud.inetutils.ignored接口		将被忽略的网络接口的Java正则表达式列表。
spring.cloud.inetutils.preferred的网络		将首选的网络地址的Java正则表达式列表。
spring.cloud.inetutils.timeout秒	1	用于计算主机名的超时时间。
spring.cloud.inetutils.use只站点本地接口	假	仅使用具有站点本地地址的接口。 有关更多详细信息，请参阅{ InetAddress＃isSiteLocalAddress（）}。
spring.cloud.loadbalancer.retry.enabled	真正	
spring.cloud.refresh.extra，刷新	真正	bean将额外的类名称发布到刷新范围中。
spring.cloud.service-registry.auto-registration.enabled	真正	如果启用自动服务注册，则默认为true。
spring.cloud.service-registry.auto-registration.fail快	假	如果没有AutoServiceRegistration，启动将失败，默认为false。
spring.cloud.service-registry.auto-registration.register管理	真正	是否将管理注册为服务，默认为true
spring.cloud.stream.binders		如果使用多于一个相同类型的活页夹（即，连接到RabbitMq的多 例），则每个活页夹的其他属性（请参阅{@link BinderPropertie 在这里您可以指定多个活页夹配置，每个活页夹配置不同的环境 spring.cloud.stream.binders.rabbit1.environment。 。 。 ，spring.cloud.stream.binders.rabbit2.environment。 。 。
spring.cloud.stream.binding重试间隔	三十	重试间隔（以秒为单位）用于计划绑定尝试。 默认：30秒。
spring.cloud.stream.bindings		其他绑定属性（请参阅{@link BinderProperties}）每个绑定名称 如'输入'）。 例如; 这为Sink应用程序的'输入'绑定设置内容类 型：'spring.cloud.stream.bindings.input.contentType = text / plain
spring.cloud.stream.consul.binder.event超时	五	

Name	Default	描述
spring.cloud.stream.default的粘合剂		活动中可用的多个活页夹（例如“兔子”）的所有绑定使用活页夹称;
spring.cloud.stream.dynamic-目的地	[]	可以动态绑定的目标列表。 如果设置，则只能列出列出的目的地
spring.cloud.stream.instance数	1	应用程序的已部署实例的数量。 默认值：1.注意：也可以按单个进行管理“spring.cloud.stream.bindings.foo.consumer.instance-c其中'foo'是绑定的名称。
spring.cloud.stream.instance指数	0	应用程序的实例ID：从0到instanceCount-1的数字。 用于分区并Kafka。 注意：也可以按单个绑定进行管理“spring.cloud.stream.bindings.foo.consumer.instance-index”，中'foo'是绑定的名称。
spring.cloud.stream.integration.message处理程序 - 不传播，头		不会从入站消息复制的消息头名称。
spring.cloud.stream.metrics.export的属性		将要附加到每条消息的属性列表。 一旦上下文刷新以避免每个消基础开销，这将由onApplicationEvent填充。
spring.cloud.stream.metrics.key		正在发布的度量标准的名称。 应该是每个应用程序的唯一值。 为：$ {spring.application.name：$ {vcap.application.name:${spring.config.name:application}}}
spring.cloud.stream.metrics.meter过滤器		模式来控制人们想要捕捉的'米'。 默认情况下，所有'米'将被捕获如，'spring.integration。*'将仅捕获名称以'spring.integration'开头的度量信息。
spring.cloud.stream.metrics.properties		应该添加到指标有效负载的应用程序属性例如：`spring.application**`
spring.cloud.stream.metrics.schedule间隔	60年代	间隔表示为调度指标快照发布的持续时间。 默认为60秒
spring.cloud.stream.rabbit.binder.admin-地址	[]	管理插件的网址; 只需要队列关联。
spring.cloud.stream.rabbit.binder.admin，不会忽略		
spring.cloud.stream.rabbit.binder.compression级	0	压缩绑定的压缩级别; 请参阅'java.util.zip.Deflator'。
spring.cloud.stream.rabbit.binder.connection名前缀		来自此资料夹的连接名称的前缀。
spring.cloud.stream.rabbit.binder.nodes	[]	集群成员节点名称; 只需要队列关联。
spring.cloud.stream.rabbit.bindings		
spring.cloud.zookeeper.base睡眠时间质谱	50	在重试之间等待的最初时间
spring.cloud.zookeeper.block-直到连接单元		与阻止连接到Zookeeper相关的时间单位
spring.cloud.zookeeper.block，直到连接等待	10	等待时间阻止连接到Zookeeper
spring.cloud.zookeeper.connect串	本地主机：2181	到Zookeeper集群的连接字符串
spring.cloud.zookeeper.default健康端点		默认健康端点将被检查以验证依赖关系是否存在
spring.cloud.zookeeper.dependencies		将别名映射到ZookeeperDependency。 从功能区透视图来看，际上是serviceID，因为功能区无法接受serviceID中的嵌套结构
spring.cloud.zookeeper.dependency的配置		
spring.cloud.zookeeper.dependency-名		
spring.cloud.zookeeper.discovery.enabled	真正	
spring.cloud.zookeeper.discovery.initial状态		此实例的初始状态（默认为{@link StatusConstants＃STATUS_UP}）。
spring.cloud.zookeeper.discovery.instance主机		预定义的主机，服务可以在Zookeeper中注册自己。 对应于来自范的{code address}。
spring.cloud.zookeeper.discovery.instance-ID		Id用于注册zookeeper。 默认为随机UUID。
spring.cloud.zookeeper.discovery.instance端口		端口下注册服务（默认为监听端口）
spring.cloud.zookeeper.discovery.instance的SSL端口		注册服务的Ssl端口。

Name	Default	描述
spring.cloud.zookeeper.discovery.metadata		获取与此实例关联的元数据名称/值对。 这些信息被发送到zook并且可以被其他实例使用。
spring.cloud.zookeeper.discovery.register	真正	在zookeeper中注册为服务。
spring.cloud.zookeeper.discovery.root	/服务	所有实例注册的Root Zookeeper文件夹
spring.cloud.zookeeper.discovery.uri规格	{方案}：// {地址}：{端口}	在Zookeeper中服务注册期间要解析的URI规范
spring.cloud.zookeeper.enabled	真正	Zookeeper是否启用
spring.cloud.zookeeper.max重试次数	10	最大重试次数
spring.cloud.zookeeper.max睡眠-MS	500	最大时间（以毫秒为单位）在每次重试时睡眠
spring.cloud.zookeeper.prefix		将应用于所有Zookeeper依赖性路径的通用前缀
spring.integration.poller.fixed延迟	1000	修正了默认轮询器的延迟。
spring.integration.poller.max的消息每次轮询	1	默认轮询器的每次轮询的最大消息数。
spring.sleuth.annotation.enabled	真正	
spring.sleuth.async.configurer.enabled	真正	启用默认的AsyncConfigurer。
spring.sleuth.async.enabled	真正	启用检测异步相关组件，以便跟踪信息在线程之间传递。
spring.sleuth.baggage密钥		应该在流程外传播的行李钥匙名称列表。 这些密钥将在实际密钥以`baggage`作为前缀。 该属性的设置是为了与之前的Sleuth版兼容。 @see brave.propagation.ExtraFieldPropagation.FactoryBuilder addPrefixedFields（String，java.util.Collection）
spring.sleuth.enabled	真正	
spring.sleuth.feign.enabled	真正	使用Feign时启用跨度信息传播。
spring.sleuth.feign.processor.enabled	真正	启用后置处理器，将Feign上下文包装在其跟踪表示中。
spring.sleuth.http.enabled	真正	
spring.sleuth.http.legacy.enabled	假	
spring.sleuth.hystrix.strategy.enabled	真正	启用自定义HystrixConcurrencyStrategy，将所有Callable实例包们的侦查代表 - TraceCallable中。
spring.sleuth.integration.enabled	真正	启用Spring Integration侦测工具。
spring.sleuth.integration.patterns	[！hystrixStreamOutput *，*]	通道名称将与之匹配的一组模式。 @see org.springframework.integration.config.GlobalChannelIntercepto patterns（）。 默认为任何不符合Hystrix Stream通道名称的通道称。
spring.sleuth.integration.websockets.enabled	真正	启用WebSockets的跟踪。
spring.sleuth.keys.http.headers		额外的标题，如果它们存在，应该添加为标记。 如果标题值是的，则标记值将是逗号分隔的单引号列表。
spring.sleuth.keys.http.prefix	HTTP。	标题名称的前缀，如果它们添加为标记。
spring.sleuth.log.slf4j.enabled	真正	启用在日志中打印跟踪信息的{@link Slf4jCurrentTraceContext}。
spring.sleuth.messaging.enabled	假	
spring.sleuth.messaging.kafka.enabled	假	
spring.sleuth.messaging.kafka.remote服务名称	卡夫卡	
spring.sleuth.messaging.rabbit.enabled	假	
spring.sleuth.messaging.rabbit.remote服务名称	的RabbitMQ	
spring.sleuth.opentracing.enabled	真正	

Name	Default	描述
spring.sleuth.propagation密钥		与线上相同的进程中引用的字段列表。 例如，名称“x-vcap-requ id”将被设置为包含前缀。 注意：{@code fieldName}将隐式小写。 @see brave.propagation.ExtraFieldPropagation.FactoryBuilder addField（String）
spring.sleuth.rxjava.schedulers.hook.enabled	真正	通过RxJavaSchedulersHook启用对RxJava的支持。
spring.sleuth.rxjava.schedulers.ignoredthreads	[HystrixMetricPoller，^ RxComputation。* $]	线程名称的跨度不会被抽样。
spring.sleuth.sampler.probability	0.1	应该抽样的请求的概率。 例如，应该对1.0 - 100％的请求进行 精度仅为整数（即不支持0.1％的迹线）。
spring.sleuth.scheduled.enabled	真正	为{@link org.springframework.scheduling.annotation.Scheduled 踪。
spring.sleuth.scheduled.skip模式	org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask	应该跳过的类的完全限定名称的模式。
spring.sleuth.supports联接	真正	True表示跟踪系统支持在客户端和服务器之间共享一个span ID。
spring.sleuth.trace-id128	假	如果为true，则生成128位跟踪ID而不是64位跟踪ID。
spring.sleuth.web.additional-跳过模式		跟踪时应跳过的网址的其他模式。 这将被追加到{@link SleuthWebProperties＃skipPattern}
spring.sleuth.web.client.enabled	真正	将拦截器注入到{@link org.springframework.web.client.RestTem
spring.sleuth.web.enabled	真正	如果为true，则启用Web应用程序的检测
spring.sleuth.web.skip模式	/api-docs.*	/自动配置
/ configprops	/倾倒	/健康
/信息	/metrics.*	/映射
/跟踪	/昂首阔步。*	。* \。PNG
。* \。CSS	。* \。JS	。* \。HTML
/favicon.ico	/hystrix.stream	/应用/。*
/actuator.*	/ cloudfoundryapplication	跟踪中应跳过的网址模式
spring.sleuth.zuul.enabled	真正	使用Zuul时启用跨度信息传播。
stubrunner.amqp.enabled	假	是否启用对Stub Runner和AMQP的支持。
stubrunner.amqp.mockCOnnection	真正	是否启用对Stub Runner和AMQP模拟连接工厂的支持。
stubrunner.classifier	存根	默认情况下，在常青藤协调中使用的分类器用于存根。
stubrunner.cloud.consul.enabled	真正	是否在领事中启用存根注册。
stubrunner.cloud.delegate.enabled	真正	是否启用DiscoveryClient的Stub Runner实现。
stubrunner.cloud.enabled	真正	是否为Stub Runner启用Spring Cloud支持。
stubrunner.cloud.eureka.enabled	真正	是否在Eureka中启用存根注册。
stubrunner.cloud.ribbon.enabled	真正	是否启用Stub Runner的功能区集成。
stubrunner.cloud.stubbed.discovery.enabled	真正	服务发现是否应该存根存根亚军。 如果设置为false，存根将在 发现中注册。
stubrunner.cloud.zookeeper.enabled	真正	是否在Zookeeper中启用存根注册。
stubrunner.consumer名		您可以通过为此参数设置一个值来覆盖此字段的默认{@code spring.application.name}。
stubrunner.delete存根，之后测试	真正	如果设置为{false}，则不会在运行测试后从临时文件夹中删除存
stubrunner.ids	[]	以“ivy”符号（[groupId]：artifactId：[version]：[classifier] [：port] 的存根的ID。 {@code groupId}，{code classifier}，{code code} code}可以是可选的。

Name	Default	描述
stubrunner.ids到服务组标识符		将基于Ivy的Ivy符号映射到应用程序内的serviceIds 例 “a：b”→“myService”“artifactId”→“myOtherService”
stubrunner.integration.enabled	真正	是否启用Stub Runner与Spring Integration的集成。
stubrunner.mappings输出文件夹		转储每个HTTP服务器到选定文件夹的映射
stubrunner.max端口	15000	自动启动的WireMock服务器的最大端口值
stubrunner.min端口	10000	自动启动的WireMock服务器的最小端口值
stubrunner.password		存储库密码
stubrunner.properties		可以传递给自定义的属性的映射{@link org.springframework.cloud.contract.stubrunner.StubDownloader
stubrunner.proxy主机		存储库代理主机
stubrunner.proxy端口		存储库代理端口
stubrunner.snapshot - 检查 - 跳过	假	如果设置为{true}，则不会断言下载的存根/合同JAR是从远程位置 从本地位置下载（仅适用于Maven仓库，而不适用于Git或Pact）
stubrunner.stream.enabled	真正	是否启用Stub Runner与Spring Cloud Stream的集成。
stubrunner.stubs模式		选择存根应该从哪里来
stubrunner.stubs每消费	假	应该只有这个特定的使用者的存根才能在HTTP服务器存根中注
stubrunner.username		存储库用户名