Java
Programing

Java 并发编程学习笔记

高阶圣堂武士

人们一直认为并发编程技术是 Java 编程中的高级技术，没有必要掌握。由于一些技术框架提供了较好的多线程支持（比如 Servlet、RMI)，很多时候软件工程师可以不必过多考虑多线程问题。然而，如果你是一个大型数据分析/数据挖掘软件的架构师的话，你就需要使用并发技术来提供高吞吐量、高可伸缩性和快速响应能力，因为这些特性是一个设计良好的商业软件必须具备的，而技术框架提供的线程隔离已经不能满足这方面的需要。

通过使用并发技术可以开发出并行算法，充分利用多处理器系统的计算能力，提高任务执行速度。此外，很多软件中存在的时隐时现的 Bug 都是由于开发人员不懂并发造成的，这种 Bug 通常难以发现和修复，导致软件的维护成本很高。因此对于 Java 高级程序员来说，并发编程已成为一项必备的技能。

目前，在 Java 并发编程方面论述系统、内容详实的中文资料很少。本文是作者在实际工作中经验总结，有一部分内容来自《Java Concurrency In Practice》。如果英文较好的话，建议还是看《Java Concurrency In Practice》毕竟本文作者的水平有限。你需要具备 Java 核心编程基础知识才能够看懂本文中的大部分内容，因而并不鼓励编码量较少的初级 Java 程序员阅读。一般来说，读完前八章即可应付一般的 Java 多线程应用程序。

本文仅供编程爱好者学习交流之用，请下载后 24 小时内删除。请不要将本文用于任何商业目的，否则后果自负。

目 录

第一章 概述......7
1.1. 并发简史......7
1.2. 使用多线程的好处......8
1.2.1. 利用多处理器的处理能力......9
1.2.2. 建模的简单性......9
1.2.3. 简化异步事件的处理......10
1.2.4.更好的用户界面响应能力......10
1.3. 使用多线程的风险......11
1.3.1. 安全风险......11
1.3.2. 活跃性风险......13
1.3.3. 性能风险......13
1.4. 多线程无处不在......14
第二章 线程安全性......16
2.1. 什么是线程安全......17
2.1.1. 一个无状态的 Servlet......18
2.2. 原子性......19
2.2.1. 竞争条件......20
2.2.2. 延迟初始化......20
2.2.3. 复合操作......21
2.3. 锁......22
2.3.1. 内部锁......23
2.3.2. 重入......24
2.4. 使用锁确保对象状态一致性......25
2.5. 活跃性和性能......26
第三章 共享对象......30
3.1. 内存可见性......30
3.1.1. 陈旧数据......31
3.1.2. 非原子性 64 位操作......32
3.1.3. 锁和可见性......33
3.1.4. volatile 域......33
3.2. 发表与逃逸......34
3.2.1. 安全构造实践......36
3.3. 线程封闭......37
3.3.1. Ad-hoc 线程封闭......38
3.3.2. 堆栈线程封闭......38
3.3.3. ThreadLocal......39
3.4. 不可变对象......40
3.4.1. final 域......41
3.4.2. 使用 Volatile 来发表 Immutable 对象......42
3.5. 安全发表对象......43
3.5.1. 不合理的对象发表方式......44
3.5.2. Immutable 对象和初始化安全......44

3.5.3. 安全的对象发表方式......45
3.5.4. 等效 Immutable 对象......45
3.5.5. Mutable 对象......46
3.5.6. 安全地共享对象......46
第四章 组合对象......48
4.1. 设计一个线程安全类......48
4.1.1. 收集同步需求......49
4.1.2. 状态依赖方法......50
4.1.3. 对象所有权......51
4.2. 实例封闭......51
4.2.1. Java 监视器模式......53
4.2.2. 一个实例封闭的例子......53
4.3. 线程安全性代理......55
4.3.1. 将线程安全性代理给子组件......56
4.3.2. 相互独立的状态域......57
4.3.3. 代理失效怎么办......58
4.3.4. 发表状态域......59
4.3.5. 例子：发表其状态域的车辆追踪器类......59
4.4. 向线程安全类添加功能......61
4.4.1. 客户端锁......62
4.4.2. 组合......63
4.5. 在文档注释中记录使用的同步策略......67
4.5.1. 解释模糊的文档注释......68
第五章 并发构建块......70
5.1. 线程安全集合......70
5.1.1. 线程安全集合的问题......70
5.1.3. 隐藏的迭代器......72
5.2. 并发集合......73
5.2.1. 使用 ConcurrentHashMap......74
5.2.2. 额外的原子集合操作......75
5.2.3. 使用 CopyOnWriteArrayList......76
5.3. 阻塞式队列与生产者-消费者模式......77
5.3.1. 例子：磁盘文件索引......78
5.3.2. 串行线程封闭......80
5.3.3. 双头队列与工作窃取模式......80
5.4. 阻塞和可中断方法......81
5.5. 同步器......82
5.5.1. 锁存器......82
5.5.2. FutureTask......84
5.5.3. 信号量......86
5.5.4. 屏障......87
5.6. 构建一个有效的可伸缩的结果缓存类......93
第一部分总结......100
第六章 任务执行框架......101

6.1. 在线程中执行任务......101
6.1.1. 序列化执行任务......101
6.1.2. 为每个任务显式地创建线程......102
6.1.3. 无界线程创建的缺点......103
6.2. Executor 框架......104
6.2.1. 例子：使用 Executor 实现的 Web 服务器......105
6.2.2. 执行策略......106
6.2.3. 线程池......107
6.2.4. Executor 的生命周期......108
6.2.5. 延迟性任务和周期性任务......110
6.3. 寻找可利用的并行性......114
6.3.1. 示例：顺序性页面渲染器......114
6.3.2. Callable 接口和 Future 接口......115
6.3.3. 实例：使用 Future 实现页面渲染器......118
6.3.4. 并行异构任务的局限性......120
6.3.5. CompletionService......120
6.3.6. 实例：使用 CompletionService 实现页面渲染器......123
6.3.7. 为任务指定期限......124
6.3.8. 示例：一个旅游预约门户网站......125
6.4. 本章小结......127
第七章 任务的取消与关闭......129
7.1. 任务取消......129
7.1.1. 中断......132
7.1.2. 中断策略......135
7.1.3. 响应中断......135
7.1.4. 示例：限时任务......137
7.1.5. 使用 Future 来取消任务......137
7.1.6. 处理不可中断的阻塞式方法......139
7.1.7. 使用 newTaskFor 实现非标准化的任务取消......141
7.2. 关闭一个基于线程的服务......143
7.2.1. 实例：一个日志服务......143
7.2.2. ExecutorService 的 shutdown 方法......148
7.2.3. 毒药丸......149
7.2.4. 示例：一个一次性的 ExecutionService......152
7.2.5. shutdownNow 方法的局限性......154
7.3. 线程异常终止......159
7.3.1. UncaughtExceptionHandler......160
7.4. 关闭 JVM......163
7.4.1. 关闭钩子......164
7.4.2. daemon 线程......164
7.4.3. Finalizer......165
7.5. 本章小结......165
第八章 使用线程池......166
8.1. 任务与执行策略的耦合......166

8.1.1. 线程饥饿死锁......167
8.1.2. 耗时任务......167
8.2. 设置线程池大小......167
8.3. 配置 ThreadPoolExecutor......168
8.3.1. 线程的创建与销毁......169
8.3.2. 管理队列中的任务......169
8.3.3. 饱和策略......170
8.3.4. ThreadFactory 接口......172
8.3.5. 在创建之后配置 ThreadPoolExecutor......175
8.4. 扩展 ThreadPoolExecutor......176
8.4.1. 示例：为线程池添加统计信息......176
8.5. 并行递归算法......178
8.5.1. 示例：一个解谜框架......180
8.6. 本章小结......188
第九章 GUI 应用程序......189
9.1. 为什么 GUI 应用程序是单线程的？......189
9.1.1. 顺序化事件处理......189
9.1.2. Swing 中的线程封闭......190
9.2. 不耗时的 GUI 事件处理任务......193
9.3. 耗时的 GUI 事件处理任务......194
9.3.1. 取消耗时的事件处理任务......196
9.3.2. 进度和任务完成指示......197
9.3.3. SwingWorker......202
9.4. 共享数据模型......202
9.4.1. 线程安全的数据模型......202
9.4.2. 分裂的数据模型设计......203
9.5. 其它形式的单线程子系统......203
9.6. 本章小结......203
第十章 避免活跃性风险......204
10.1. 死锁......204
10.1.1. 锁序死锁......205
10.1.2. 动态锁序死锁......206
10.1.3. 合作对象之间的死锁......209
10.1.4. 开放调用......211
10.1.5. 资源死锁......212
10.2. 死锁的避免与诊断......213
10.2.1. 限时获取锁......213
10.2.2. 使用线程转储来分析死锁......214
10.3. 其他活跃性风险......215
10.3.1. 饥饿......215
10.3.2. 响应速度......216
10.3.3. 活锁......216
10.4. 本章小结......218
第十一章 性能和可伸缩性......219

11.1. 考虑性能......219
11.1.1. 性能和可伸缩性......219
11.1.2. 性能折中......220
11.2. 阿姆达尔定律......221
11.2.1. 示例：框架中隐藏的顺序化操作......223
11.2.2. 定性地应用阿姆达尔定律......224
11.3. 线程引起的开销......224
11.3.1. 上下文切换......224
11.3.2. 内存同步......225
11.3.3. 阻塞......226
11.4. 减少锁竞争......227
11.4.1. 减少锁的作用范围......227
11.4.2. 减少锁的粒度......229
11.4.3. 锁剥离技术......231
11.4.4. 避免过热的状态域......233
11.4.5. 排他锁的替代品......233
11.4.6. 监视 CPU 利用率......234
11.4.7. 不要使用对象池......235
11.5. 示例：比较不同 Map 实现的性能......235
11.6. 减少上下文切换造成的开销......236
11.7. 本章小结......237
第十二章 测试并发程序......238
12.1. 正确性测试......238
12.1.1. 基本测试单元......240
12.1.2. 测试可阻塞操作......241
12.1.3. 安全性测试......242
12.1.4. 测试资源管理......247
12.1.5. 使用回调......247
12.1.6. 产生更多的线程重叠......249
12.2. 性能测试......250
12.2.1. 扩展 PutTakeTest 类以支持性能测试......250
12.2.2. 比较多种算法......254
12.2.3. 测量响应速度......255
12.3. 避免性能测试中存在的陷阱......255
12.3.1. 垃圾回收......255
12.3.2. 动态编译......256
12.3.3. 不切实际的代码访问路径......257
12.3.4. 不切实际的竞争程度......257
12.3.5. 消除死代码......258
12.4. 其他测试方法......258
12.4.1. 代码审查......259
12.4.2. 静态分析工具......259
12.4.3. 面向切面的测试技术......260
12.4.4. 性能分析与监测工具......260

12.5. Object 对象中与同步相关的方法......260
12.6. 本章小结......265

第一章 概述

编写正确的程序很难，编写正确的并发程序更难。为什么要使用并发呢？在 Java 编程中你免不了要使用并发特性，在一些复杂系统的开发中使用并发使你能够将一些复杂的计算过程以最直接的方式用代码表现出来。此外，并发程序能够有效利用多核处理器的计算能力，随着计算机系统中处理器数量的增加，研究如何有效地编写 Java 并发程序变得非常重要。

1.1. 并发简史

在计算机发展的早期，操作系统还没有诞生。人们在计算机上执行单道程序。该程序在执行过程中独占计算机，可以访问计算机中的所有资源。编写可在裸机上执行的程序很困难，此外一次只执行一道程序效率是非常低下的，是对相对昂贵的计算机资源的一种浪费。

操作系统逐渐发展成可以允许多个程序并发执行的多任务模式。每个程序在一个相对独立的进程中执行，操作系统为各个进程分配资源，例如内存空间、文件句柄等。如果有需要，进程之间还可以使用一些粗粒度的机制相互通信，例如：套接字、信号句柄、共享内存、信号量和文件。

导致操作系统向多任务方向发展的因素有：

a） **资源利用**：当有的进程在等待外设完成某个操作的时候，可以让其他进程运行，这样可以避免时间上的浪费。

b） **公平**：多个用户或程序可能对计算机资源有着相等的需求，通过精细化的时间分片技术让他们共享计算机资源要比一个程序运行完之后再运行另一个要好得多。

c） **方便**：一般来说写几个程序，让他们各自负责执行不同的任务，并在有必要的时候相互沟通，要比写一个能够完成所有任务的程序要合适和容易。

在早期的分时系统中，每个进程都是一个虚拟的冯诺依曼计算机。它拥有自己的内存空间用来存储数据和指令，根据机器语言的语义顺序执行指令并且通过操作系统，使用一组输入输出原语与外设交互。对于每个被执行的指令来说都清楚地定义了下一条要执行的指令，用来根据指令集的规则控制程序的执行过程。

目前几乎所有广泛使用的编程语言都遵循这个序列化编程模型，语言规格说明明确地定义了在一个指令执行之后执行哪个指令。

序列化编程模型是最符合直觉的也是最自然的，因为这和人类的工作模式是一致的。例如，你打开碗橱，选择一种茶，如果茶壶中有足够多的水就将茶放入其中，如果茶壶中没有足够多的水就放入一些水，然后把茶壶放到炉子上，打开炉子，等待水沸。最后一步等待水沸的过程中可以引入一些异步操作。当水正在被加热的时候，你可以选择坐在那儿等，也可以利用那段时间做些其他的工作，例如打开烤箱或者看报纸。在做这些工作的时候，你还要保持警惕，当水开了的时候立即去处理。烤箱和茶壶的制造者知道他们的产品常常会在异步模式下被使用，因此它们在完成工作的时候都会发出音频信号。在序列化和异步化之间的平衡能力是一个人工作效率的重要特征，对于程序来说也是如此。

驱动进程发展的三个因素（资源利用、方便、公平）也同样驱动着线程的发展。线程允许在一个进程中存在多个执行控制流，它们共享进程提供的资源，例如内存和文件句柄，但是每个线程有自己的程序计数器、堆栈和局部变量。线程也提供了多处理器系统的硬件并行性的自然分解，同一个进程中的多个线程可以在不同的 CPU 上同时执行。

线程又被称为轻量级进程，并且大多数现代操作系统将线程而非进程作为最基本的调度单元。在没有明确的协调机制的时候，线程异步并发执行。由于同一个进程中的所有线程共享相同的内存空间，多个线程可以访问相同的变量，并在同一个堆中创建对象。相比于进程之间粗粒度的数据共享，线程之间存在精细化的数据共享。但是如果没有明确的同步机制来协调线程对这些共享数据的访问，一个线程可能修改另一个线程正在使用的变量，从而产生不可预测的结果。

1.2. 使用多线程的好处

如果能够合理地使用多线程，将能够缩减复杂应用程序的开发和维护成本，并能提供更好的性能。通过将异步工作流转换为多个序列化工作流，多线程可以更好地对人类的工作和交互方式建模。使用多线程，很多复杂的代码将变得更加直截了当，因此更容易编写、阅读和维护。

在图形用户界面程序中使用多线程可以提高界面响应速度，在服务器程序中使用多线程可以提高资源利用率和吞吐量。多线程还可以简化 JVM 的设计，垃圾

回收器一般在一个专用线程中工作。大多数卓越的 Java 应用程序都在某种程度上依赖于多线程。

1.2.1. 利用多处理器的处理能力

多处理器计算机系统曾经非常昂贵和稀有，只用在大型数据中心和科学计算基础设施中。今天它们更加便宜和丰富，即使是低端服务器和中档桌面计算机系统也往往拥有多个处理器。这种趋势只会增加，因为增加处理器的时钟频率越来越困难，处理器制造商将选择在一个处理器中包含更多的核心。所有的主流芯片制造商都已经开始了这种转变，并且我们已经目睹了一些拥有很多个处理器的计算机的出现。

由于最基本的调度单元是线程，只拥有一个线程的程序一次最多只能在一个处理器上运行。在一个拥有两个处理器的计算机系统中，单线程程序放弃了一半的处理器资源。在一个拥有 100 个处理器的计算机系统中，单线程程序放弃了99%的处理器资源。另一方面，拥有多个线程的程序，同时可以在多个处理器中执行。如果设计得比较合理，多线程程序可以更有效地利用处理器资源，从而增加程序的吞吐量。

即使在单处理器计算机系统中，使用多线程也可以帮助提高吞吐量。如果一个程序是单线程的，在等待一个异步输入输出操作完成的时候，处理器处于空闲状态。如果该程序是多线程的，另一个线程就可以利用这个时间段来执行（这就类似于一边读报纸一边等待水沸，而不是等水沸了之后再读报纸）。

1.2.2. 建模的简单性

当你只有一种类型工作要做的时候，你更容易管理时间。如果你只有一种工作要做，你可以从第一件开始逐个完成，直到完成最后一件。你不必耗费精力来确定下一件要做的工作是什么。另一方面，管理多个不同优先权和截止期限的工作，并在不同类型的工作之间切换，需要一些额外的开销。

对于软件来说也是如此，相比于同时管理多个不同类型任务的程序，一个管理一种类型任务的程序编写起来更简单，更不容易出错，更容易测试。一个复杂的异步工作流可以分解成多个简单的同步工作流，每个工作流在一个单独的线程中执行，只在特定的同步点进行线程间通信。

一些框架（例如 Servlet 和 RMI）利用了多线程的这种好处。由框架来处理

请求管理、线程创建、负载均衡等细节问题。Servlet 的编写者不需要担心同时有多少其他请求被处理或者输入输出流是否阻塞等问题。当 Servlet 的 Service 方法被调用的时候，它可以将对该请求的响应当做一个单线程程序。这简化了 Servlet 组件的开发并削减了该框架的学习难度。

1.2.3. 简化异步事件的处理

一个服务器应用程序可以从多个远程客户端接受 Socket 连接，如果为每个连接都分配一个单独的线程，并使用阻塞式 I/O，这样的程序更容易开发。

如果一个程序从 Socket 中读取数据，但是 Socket 中没有数据，那么这个 read 方法就会阻塞，直到 Socket 中有数据可用。如果在单线程程序中，这不仅意味着相应的请求的处理被拖延，其他请求的处理也会被拖延。为了避免这个问题，单线程服务器程序被迫使用非阻塞式 I/O，相比于阻塞式 I/O 它更复杂也更容易出错。然而，如果每个请求都有自己的线程，那么一个线程的阻塞不会影响到其他请求。

由于历史原因，操作系统一般只允许一个进程拥有几百个或者更少的线程。操作系统提供了有效的基础设施来多路复用 I/O，例如 Unix 系统中的 select 和 poll 系统调用，Java 类库中提供了 java.nio 包来访问这些设施。然而，操作系统开始逐渐支持更大数量的线程，使得为每个客户端分配一个线程的策略变为现实，即使是在拥有大量客户端的平台中。

1.2.4. 更好的用户界面响应能力

图形用户界面曾经是单线程的，这意味着你必须经常轮询，查看是否有输入事件到达，或者通过一个主事件循环间接地执行程序。如果主事件循环中调用的代码花了太多的时间来执行，那么在这块代码返回之前，用户界面会被冻结。因为在控制返回主事件循环之前无法处理界面事件。

现代图形用户界面，例如 AWT 和 Swing 工具箱，使用事件分派线程来替换主事件循环。当一个用户界面事件，例如一个单击按键事件发生了，应用程序特定的事件处理器会被事件分派线程调用（该事件处理器还是在事件分派线程中执行）。事件分派线程处于 GUI 工具箱的控制之下，而不是处于应用程序的控制之下。

如果事件处理器能够很快执行完毕，就不会影响事件分派线程对其他用户界

面事件的响应。然而，如果要在事件处理器中执行耗时的任务，比如大文档的拼写检查或者从网络上获取资源等，这样就会损害事件分派线程的响应能力。如果事件分派线程正在处理耗时任务，就无法对用户的界面事件进行响应，更糟糕的是，即使界面上有取消按钮，你也无法取消这个正在执行的耗时的任务，因为，此时事件分派线程正在忙碌，无法响应取消按钮的单击事件。如果新开一个线程，将该耗时任务放置在一个单独的线程中执行的话，就不会影响事件分派线程的响应能力。

1.3. 使用多线程的风险

Java 提供的对多线程的内建支持是一把双刃剑。尽管它为多线程提供了语言和库的支持以及一个跨平台的内存模型，简化了多线程应用程序的开发，它也对编程者提出了更高的要求，因为越来越多的程序将会使用多线程。多线程比较难懂，并发也是一个高级主题，现在，主流开发者必须注意到线程安全问题。

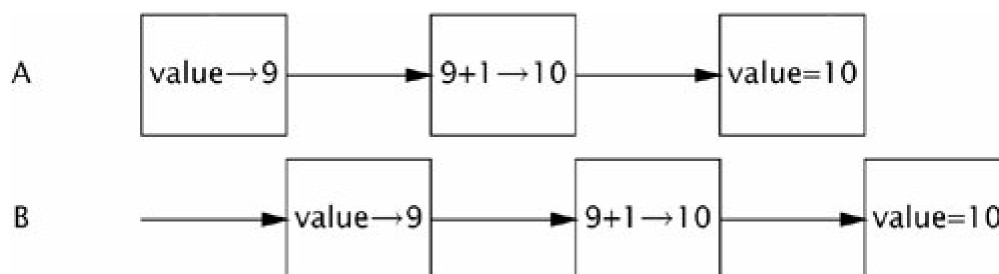
1.3.1. 安全风险

线程安全问题可能变得意想不到地微妙，因为在缺少合理的同步机制的情况下，多线程的执行顺序是不可预知的，有时甚至是令人惊讶的。下面的程序本来想产生一列无重复的整数值，在单线程环境中运行正常，在多线程环境中却运行失败。这个程序演示了多线程交错运行可能导致出乎意料的结果。

```
/**
 * 用于实现输出一个无重复的整数序列的类
 *
 * 该类不是线程安全的
 */
@NotThreadSafe
public class UnsafeSequence
{
    private int value;

    public int getNext()
    {
        return value++;
    }
}
```

上例在多线程环境中失败的原因是，在某些时候，两个线程调用 getNext()方法的时刻非常接近，以至于返回的是同一个数值。value++看似是一个操作，

实质上是三个操作：读取 value 值，value 值加 1，重设 value 值。由于多个线程中的操作可以任意地交错，可能出现两个线程获得的是相同数值的 value，然后各自都加 1 的情况。下图形象地说明了这一点：



这个例子演示了一种常见的并发风险，叫做竞争条件。nextValue()方法是否返回一列无重复的整数值取决于多线程在运行时的交错情况，这种不确定性不是我们希望看到的。

由于多线程共享同样的内存空间，并且并发执行，它们可以访问和修改其他线程正在使用的变量。这样非常方便，相比于其他线程间通信机制，共享变量更简单。但共享变量也有极大的风险。数据可能会以不希望的方式被修改掉，导致一些很难找出原因的 Bug。为了让多线程程序的行为具有可预测性，共享变量的访问必须被合理地协调，使得一个线程对该变量的访问不会干扰到另一个线程。幸运的是，Java 提供了同步机制来协调共享变量的访问。

为 getNext()方法加上 synchronized 修饰符，上述代码可以被修复为如下代码：

```
/**
 * 用于实现输出一个无重复的整数序列的类
 *
 * 该类是线程安全的
 */
@ThreadSafe
public class Sequence
{
    //nextValue这个域必须在持有本对象的内部锁的时候才能访问
    @GuardedBy("this")
    private int nextValue;

    public synchronized int getNext()
    {
        return nextValue++;
```

```
    }
}
```

如果没有 synchronized 修饰符，编译器、硬件以及运行时将拥有很大的自由来安排操作执行的顺序和时机，比如将变量缓存在寄存器中，让它不被其他线程可见。这些优化技巧是为了帮助提高运行速度，但是给开发者增加了负担，他们必须明确地指定多线程所共享的变量放置在哪里。你不需要为此担心，只要遵循第二和第三章中的规则就可以忽略这些低层的细节。

1.3.2. 活跃性风险

在开发并发程序的时候一定要注意线程安全问题，线程安全性是不能妥协的。不仅多线程程序需要注意安全性，单线程程序也需要注意安全性和正确性，但是多线程引入了额外的安全风险。相似地，多线程引入了活跃性故障，这在单线程程序中是不存在的。

当一个活动到达了无法前进的状态的时候，活跃性故障就发生了。在单线程程序中有一种活跃性故障是由无限循环造成的，导致循环之后的代码永远无法被执行。多线程的引入，增加了活跃性故障发生的可能性。例如线程 A 在等待一个资源，该资源被线程 B 排他性地占有了，并且线程 B 永远不释放该资源，那么线程 A 必须永远等待下去。这就是一种活跃性故障。本书在第十章将会描述各种形式的活跃性故障，以及如何避免它们，包括死锁、饥饿、活锁。就像大多数并发 Bug 一样，由活跃性故障引起的 Bug 往往难以锁定，因为它们依赖于不同线程中事件发生的相对时机，因此在开发和测试过程中这种 Bug 会时隐时现。

1.3.3. 性能风险

性能问题包含一大堆其他问题，包括响应速度、吞吐量、资源消耗量和可伸缩性。就像安全性和活跃性一样，多线程程序不仅要面对单线程程序中所有的性能风险，还要面对由多线程引入的性能风险。

在设计良好的并发程序中，多线程的使用可以提高性能，但是多线程会增加运行时开销。在多线程程序中调度器经常需要临时挂起一个线程，运行另一个线程，这被称为上下文切换。上下文切换会造成很大的开销，保存和恢复执行上下文、调度线程都需要时间。线程间的同步机制将会阻碍编译器对代码的优化、阻止内存缓冲区的清空和验证，这些因素都会降低多线程程序的性能。第十一章将

会介绍一些技术来分析并减少这些不利因素。

1.4. 多线程无处不在

即使你从没有显式地创建一个线程，框架也会代表你创建一些线程，并且这些线程调用的代码必须是线程安全的。这对开发人员造成了很大的设计和实现负担。因为开发线程安全类相比于非线程安全类更困难，需要付出更多的精力。

每一个 Java 程序都是多线程的，因为当 JVM 启动的时候它创建了一些线程用来负责一些工作，例如垃圾回收线程和 Main 线程。Main 线程用来执行 main 函数中的代码。AWT、Swing 等用户界面框架会创建一个线程用来管理用户界面事件。Timer 类会创建一个线程用来执行被延迟的任务。Servlet 或者 RMI 框架会创建线程池……。

如果你像很多其他开发者那样使用这些框架的话，你必须对并发和线程安全比较熟悉，因为这些框架创建了多线程，并在这些线程中调用你开发的组件。理想情况是将多线程看做是 Java 的高级特性，并不是必须掌握的，但是现实情况是，所有的 Java 程序都是多线程的，并且这些框架的使用并不能使你完全不用考虑多线程问题。

当你使用了一个带并发特性的框架（比如 Servlet）的时候，往往很难将并发限制于框架代码中，因为框架总是要使用回调来调用你编写的组件，从而来改变组件的内部状态。类似地，线程安全性考虑不仅仅对于这些组件是必要的，对于被这些组件调用的其他类也是必要的。因此， 线程安全性需求是会感染的。

下面介绍的所有设施都会导致你编写的程序中的一些代码在其他的线程中被调用。尽管线程安全性需求起源于这些设施，但是一般会导致应用程序中的线程安全性需求。

- **Timer 类**：Timer 类是一个方便的机制，用来在随后调用一个任务（一次性的或周期性的）。Timer 类的引入会使得原本序列化的程序变得复杂，因为一个 TimerTask 是在 Timer 管理的某个线程中执行的，而不是在应用程序的线程中执行的。如果某个 TimerTask 访问了一个被其他线程访问的数据，两者都必须以线程安全的形式进行访问。通常来说最简单的办法就是让被 TimerTask 访问的对象本身就是线程安全的，也就是说将线程安全性封装在共享对象中。

- **Servlet 和 JSP**：Servlet 框架用来部署 Web 应用程序并将远程 HTTP 请求分发到不同的线程中执行。一个请求将会被分发，然后通过一组过滤器，最终到达合适的 Servlet 或者 JSP。每个 Servlet 就是一个应用逻辑组件，在高访问量的服务器中很有可能出现多个客户端同时访问同一个 Servlet 的情况。Servlet 规格说明中要求 Servlet 必须是线程安全的。即使在构建 Web 应用程序的时候你可以确定某个 Servlet 同时只会在一个线程中运行，你还是要注意线程安全问题。Servlet 往往需要访问由多个 Servlet 共享的状态信息，例如存储在 ServletContext 中的对象以及存储在 HttpSession 中的对象。这些共享对象也需要考虑线程安全问题，因为它们可能在多线程中被同时访问。
- **RMI**：RMI 允许你调用另一个 JVM 中对象的方法，当你使用 RMI 调用远程方法的时候，方法参数会被打包成字节流，通过网络传送到远程 JVM 中，然后被还原为对象或原始数据类型，传送给远程方法。当远程对象的方法被调用的时候，是在声明线程中被调用吗？不是，是在一个由 RMI 管理的线程中被调用。同一个远程对象的同一个远程方法会被多个 RMI 线程同时调用吗？远程对象必须防范两个线程安全问题：被多个远程对象共享的对象应该是线程安全的，远程对象本身应该是线程安全的。
- **Swing 和 AWT**：GUI 应用程序与生俱来就是异步的。用户可能在任何时候选择一个菜单项或者按下一个按钮，用户希望应用程序在执行任何任务的时候都能够立即响应用户界面操作。Swing 和 AWT 通过创建一个独立的用于处理用户界面事件和更新图形界面的线程来解决这个问题。Swing 组件，例如 JTable 不是线程安全的，Swing 程序通过将所有对 GUI 的访问限制在事件响应线程中来达到线程安全的目的。如果一个程序想要在事件响应线程之外操作 GUI 的话，必须让操作 GUI 的代码在事件响应线程中执行。当用户在 GUI 上进行了某个操作，将会导致事件响应线程中某个事件处理函数被调用。如果这个事件处理函数需要访问被其他线程共享的对象，那么必须考虑线程安全问题。

第二章 线程安全性

在并发编程中不会过多地讨论多线程和锁，这一点可能令人惊讶。事实上编写多线程安全代码的核心问题是管理对象的状态，尤其是共享的 Mutable 对象。

粗略地说，对象的状态就是存储在实例域或者静态域中的数据。对象的状态可能包括引用其他对象的域。对象的状态包含了所有可以影响其对外行为的数据。

这里说的共享对象，是指能够被多线程访问的对象。Mutable 对象指的是其值可能在其生命周期中变化的对象。我们讨论线程安全其实是讨论如何保护数据使其不被非法地并发访问。

一个对象是否需要是线程安全的依赖于它是否会被多线程访问。这是关于对象将会如何被使用的问题，而不是对象的功能问题。将 Mutable 对象设计成线程安全的需要使用同步来协调多线程对该 Mutable 对象的访问。如果不能这么做将会导致数据破坏或者其他糟糕的结果。

当多个线程同时访问或者修改某个 Mutable 的对象的域的时候，必须使用同步机制。Java 中主要的同步机制是 synchronized 关键字，它提供了排他锁。同步机制还包括 volatile 变量、原子变量以及显式锁的使用。如果不使用同步机制，你的程序或许可以通过测试甚至能够正常运行好几年，但最终还是会出问题。

如果多个线程同时访问或修改某个 Mutable 对象，又没有合理地进行同步的话，你可以使用如下三种方法来修复这个问题：

- 避免多线程对该 Mutable 对象的共享
- 将该 Mutable 对象改成 immutable 对象
- 在访问或修改该共享 Mutable 对象的时候使用 synchronized 关键字

如果你不是在设计类的时候就考虑到了并发问题，上面三种方法中的有些方法做起来可能比较麻烦，最好的办法是在设计类的时候就将其设计成线程安全的。

在一个大型的程序中，确定某个对象是否可能被多个线程访问比较困难。幸运的是，那些能够帮助你创建结构良好、可维护性强的类的面向对象设计原则比如封装和数据隐藏可以帮助你创建线程安全类。要使得访问一个域的方法尽可能少，尽量用 private 域，然后为其定义 getter 和 setter 方法，这样可以便于同

步。

有时候好的设计方法会产生冲突，这时候就需要妥协。例如抽象和封装会与性能冲突，最好先保证程序正确性，然后再考虑性能。尽管这样，如果没有必要不用太在意性能。

如果你觉得你必须打破封装性原则，你还是可以确保你的程序是线程安全的，只是更加困难。此外，这样做将会使你的程序更加脆弱，在开发和维护期间将会须要更多的成本，并冒更大的风险。第四章将会讨论在哪些条件下放松封装性原则是安全的。

到目前为止，我们一直可替换地使用“线程安全类“和”线程安全程序“两个术语。一个线程安全程序就是全部使用线程安全类构建的程序吗？不是的，全部由线程安全类构成的程序未必是线程安全的。一个线程安全程序也可能包含非线程安全类。第四章将讨论线程安全类的组成。

2.1. 什么是线程安全

很难准确地定义线程安全这个概念，正式的定义不太好理解，来看看从Google 中搜到的几个非正式的定义：

…可以被多个线程调用，多线程之间不需要额外的通信。

…可以在同时被多个线程调用，调用者不需要额外的操作。

看看这些定义就会明白，难怪线程安全这么难以理解。像“如果一个类是线程安全的，那么它可以安全地被多线程访问”这样的话虽然找不出什么毛病，但是对于实际编程没有什么指导意义。我们如何区分线程安全类和非线程安全类呢？这里说的安全到底是什么意思？

所有合理的线程安全定义的核心部分都涉及到正确性的概念。线程安全的定义之所以模糊是因为我们无法准确地定义什么是正确性。

类的正确性意味着类必须遵守其规格说明，一个好的类规格说明必须明确定义对象状态的约束条件以及对象的各个操作的后置条件。鉴于我们常常对类的规格说明描述得比较简略的这个现状，我们如何知道一个类的实现是否满足正确性？我们不能，一旦我们确信代码能够工作，我们就会使用这些类。如果乐观地将正确性定义为一种我们能够识别的东西，我们可以直接将线程安全定义为：如果一个类在被多线程访问的时候能够持续地保持正确性，那么它就是一个线程安

全类。

更确切地说，如果一个类在被多线程访问的时候能够持续地保持正确性，经受住线程执行重叠的考验，并且在调用代码中不需要任何同步措施的话，那么这个类就是一个线程安全类。

由于单线程程序是多线程程序的特例，如果一个类在单线程调用的时候都不能正确工作，在多线程调用情况下肯定也不能。如果一个线程安全类实现正确的话，那么任何由 public 方法调用和 public 域读写组成的操作序列都不能够破坏该对象的约束条件和后置条件。对一个线程安全类的对象进行任何操作（序列的或者并发的）都不能导致该对象处于数据不一致状态。

线程安全类将所有必要的同步机制都封装到其内部，因此调用者不需要再提供额外的同步。

2.1.1. 一个无状态的 Servlet

在第一章中我们列举了一些框架，它们可以自动创建线程，并在多线程中调用你编写的逻辑组件。这样，你必须确保你的逻辑组件是线程安全的。线程安全性问题往往不是由于直接使用多线程机制引起的，而是由于使用了如 Servlet 这样的框架引起的。我们将开发一个简单的基于 Servlet 的因数分解服务作为一个例子，然后慢慢地扩展它，同时保持线程安全性，代码如下：

```
@ThreadSafe
public class StatelessFactorizer extends GenericServlet implements Servlet
{
    public void service(ServletRequest req, ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
    ……
}
```

像大多数 Servlet 一样，StatelessFactorizer 类是无状态的。它没有域，也没有引用其他类的域，计算结果仅仅依赖于执行线程中 service()方法的局部变量栈。一个访问 StatelessFactorizer 的线程无法影响另一个访问 StatelessFactorizer 的线程。这是因为两个线程不共享状态，就好像它们在访问不同的实例一样。

一个线程对某个无状态对象的访问不会影响其他线程对该对象的访问。无状态对象总是线程安全的。

大多数 servlet 都可以被实现为无状态类这个事实极大地减轻了 Servlet 的设计难度。只有在一个请求要影响其他请求行为的时候，才需要考虑多线程。

2.2. 原子性

为上一章的无状态的 Servlet 添加一个状态域如何？假设你想要添加一个域，用来表示请求的个数，最简单的办法是添加一个 long 类型的实例域，代码如下：

```
@NotThreadSafe
public class UnsafeCountingFactorizer extends GenericServlet implements
Servlet {
    private long count = 0;

    public long getCount() {
        return count;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
    ……
}
```

不幸的是 UnsafeCountingFactorizer 类并不是线程安全的，虽然在单线程环境下工作正常。当执行++count 的时候，虽然由于语法比较紧凑，看上去是一个操作，但是这个操作并不是原子操作，事实上它由三个操作组成：获取当前值，当前值加 1，回写新值。如果初始状态下 count 值为 9，在某个时刻多个线程都读取 count 的值，得到 9，然后加 1，回写，count 的值最终为 10。这并不是我们想要的结果。

你可能想就算漏加了几个 1 对于一个基于 Web 的服务程序来说也是一种可以接受的精度损失，有时候的确如此，但是如果 count 值被用来产生唯一标识符的话，就会造成严重的问题。在并发编程中这种产生错误计算结果的时间点很重要，它被称为竞争条件。

2.2.1. 竞争条件

UnsafeCountingFactorizer 类中存在几个竞争条件，导致计算结果不准确。当计算的正确性依赖于多线程执行的相对时间和重叠状态的话，竞争条件就产生了。大多数竞争条件都是由于在“检查再执行”类型的操作中，基于陈旧的数据计算最终结果造成的。

在日常生活中，我们常常碰到竞争条件。比如你和朋友约好了下午在学校大道的星巴克咖啡店见面。但是当你达到学校大道的时候，你发现有两家星巴克咖啡店，你不知道应该到哪一家咖啡店去见你的朋友。在 12 点 10 分的时候，你到星巴克 A 店去找，没发现你的朋友，你又跑到星巴克 B 店去找，也没找到。这样就有几种可能性：第一种可能性：你的朋友迟到了，不在任何一家咖啡店；第二种可能性：在你离开 A 店之后，你的朋友到达了 A 店；第三种可能性：你的朋友到达了 B 店没找到你，所以到 A 店去找你，当你到达 B 店的时候，你的朋友正在去 A 店的路上。来看看最糟糕的情况，比如在 12 点 15 分的时候，你们都到过了 A 店和 B 店，你们都怀疑自己被对方放了鸽子，那怎么办？你们可以再回到另一家咖啡店寻找，这样周而复始，花上一天时间也没找到对方。

这个例子演示了一种竞争条件，结果依赖于事件发生的时机。“对方不在 A 店”这个观察结果在你离开 A 店之后立刻就失效了。你使用了这个陈旧的信息作为你下一步操作的依据。这种竞争条件被称为“检查再执行”，你先检查一个条件为 true，然后依赖于这个条件进行下一步操作，但事实上，在你的检查与运行之间这段时间里，这个条件可能变为 false。

2.2.2. 延迟初始化

延迟初始化的目的是要在真正需要一个对象值的时候再对其进行初始化，并且保证它只被初始化一次。下面代码演示了延迟初始化，getInstance 方法中首先检查域 instance 是否已经被初始化，如果已经被初始化就将其返回，否则创建一个新的实例，赋值给 instance 域，然后将该对象返回。

```
@NotThreadSafe
public class LazyInitRace
{
    private ExpensiveObject instance = null;
    public ExpensiveObject getInstance()
    {
```

```
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

LazyInitRace 类中有竞争条件，将会破坏这个类的正确性。假设线程 A 和线程 B 同时执行 getInstance 方法，线程 A 看到 instance 域的值为 null，然后为其初始化一个新的 ExpensiveObject 对象，线程 B 也看到 instance 域的值为 null，也为其初始化一个新的 ExpensiveObject 对象。这样 instance 域就被初始化了两次。因此在多线程调用中，getInstance 方法有可能返回不同的对象。

就像大多数多线程错误一样，竞争条件并不会总是造成错误，但是它会引起严重的问题。如果上例 LazyInitRace 被用作单例，产生一个整个应用程序范围的注册表，由于 getInstance 方法返回的可能是不同的对象，将会造成注册信息的丢失或者数据不一致问题。

2.2.3. 复合操作

LazyInitRace 类和 UnsafeCountingRactorizer 类都包含了需要具有原子性的操作。为了避免竞争条件，必须找到一种方法，使得我们在对变量修改期间其他线程无法使用该对象。

如果 UnsafeSequence 中 return value++语句是原子的，竞争条件就不会出现。为了确保线程安全，“检查再执行”和“延迟初始化”类型的操作都必须是原子的，我们将它们称为复合操作。下一段中我们将讨论锁，它是 Java 中实现操作原子性的内建机制。现在我们来修复 UnsafeCountingRactorizer 的问题，见如下代码：

```
@ThreadSafe
public class CountingFactorizer extends GenericServlet implements Servlet
{
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
```

```
    }
    ……
}
```

我们用 AtomicLong 代替了 long，这样对 count 的所有操作都是原子的了。由于该 servlet 的状态就是 count 的状态，这样的话，该 servlet 又是线程安全的了。

这里是为一个无状态类添加了一个状态域，因此只要这个状态域是线程安全的，那么这个类就是线程安全的。但是如果为一个无状态类添加多个状态域，情况就没那么简单了。

2.3. 锁

如上一节所示，为一个无状态类添加一个状态域，只要这个状态域使用的是线程安全类，那么这个类仍是线程安全的。如果为一个无状态类添加两个状态域，只要他们都使用线程安全类就可以了吗？事实并非如此，例如你想优化你的因数分解服务，存储最近一次需要因数分解的数以及它的所有因数，你使用了线程安全类来表示这两个域，代码如下：

```
@NotThreadSafe
public class UnsafeCachingFactorizer extends GenericServlet implements Servlet
{
    private final AtomicReference<BigInteger> lastNumber
            = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
            = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
        {
            encodeIntoResponse(resp, lastFactors.get());
        }
        else
        {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
```

```
    }
    ……
}
```

即使 lastNumber 和 lastFactors 两个域都使用了线程安全类，但是 UnsafeCachingFactorizer 仍然不是线程安全的。线程安全性的定义需要在任何时候，多线程操作的任意重叠状态下都能保持约束条件。这个类的约束条件是，被分解的数必须等于所有因子的乘积。如果不能维持这个约束条件，我们的程序就无法保持正确性。

在某些不幸的时间点 UnsafeCachingFactorizer 类可能会违反这个约束条件。虽然使用了原子引用，我们还是无法同时更新 lastNumber 和 lastFactors 两个域的值。

2.3.1. 内部锁

Java 提供了内建的锁机制来实现多个操作的原子性：synchronized 块。Synchronized 块由两个部分组成，第一个部分是作为锁的对象，第二个部分是被锁保护的一个代码块。synchronized 方法是 synchronized 块的一个变种，相当于将 this 用作方法中所有语句的锁。

每个 Java 对象都可以作为锁，这些内建的锁被称为内部锁或监视锁。在线程进入 synchronized 块之前自动获取内部锁，在线程离开 synchronized 块之后自动释放内部锁（不管是正常离开 synchronized 块还是抛出异常导致离开 synchronized 块）。

内部锁是一种互斥锁，这意味着在任意时刻最多只能有一个线程可以获取这个锁。如果一个内部锁已经被线程 B 获取，线程 A 想要获取这个内部锁就必须等待，直到线程 B 释放该内部锁。如果线程 B 一直不释放，那么线程 A 就必须一直等待下去。

由于一次只允许有一个线程执行某个 synchronized 块中的代码，这就保证了 synchronized 块中的代码可作为一个原子操作执行。不可能出现在某个线程正在执行 synchronized 块中的某条代码的时候，另一个线程进入这个 synchronized 块的情况。

有了这种同步机制，我们的因数分解 Servlet 就容易实现线程安全了。下面的代码将 service 方法修饰为 synchronized 的，这样，同时最多只有一个线程

能够执行 service 方法中的代码。这样做有点过了，因为这会影响该 Servlet 的响应能力，但这是性能问题，而不是并发问题，对于这一点以后再做讨论。

```
@ThreadSafe
public class SynchronizedFactorizer extends GenericServlet implements Servlet
{
    @GuardedBy("this")
    private BigInteger lastNumber;
    @GuardedBy("this")
    private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
        {
            encodeIntoResponse(resp, lastFactors);
        }
        else
        {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
    ……
}
```

2.3.2. 重入

当一个线程请求一个被其他线程获取的锁的时候，这个线程会阻塞。但是由于 Java 中的内部锁是可重入的，因此当一个线程请求一个已经被其获取的内部锁的时候，这个请求会成功。

可重入性简化了面向对象并发代码的开发。下面代码中的子类覆盖了超类中的一个 synchronized 方法，并调用了超类中对应的方法，如果没有可重入锁，这样非常自然的代码将会造成死锁。由于 Widget 类和 LoggingWidget 类中的 doSomething 方法都是 synchronized 的，如果内部锁不是可重入的，那么 super.doSomething()语句将永远无法获取到锁，因为这个锁已经被其获取了。有了可重入锁就不会有这种问题。

```
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

2.4. 使用锁确保对象状态一致性

我们可以使用锁来确保多线程对保护块中代码的排他性访问，从而确保对象的状态一致性。共享对象上的复合操作，例如 count++或者延迟初始化操作都必须是原子操作。使用 synchronized 块就可以做到。然而仅仅将复合操作用 synchronized 块包起来还不够。需要在任何访问共享 Mutable 变量的地方都使用 synchronized 块。

人们常犯的一个错误是认为只有在对共享变量进行写操作的时候才需要使用 synchronized 块，事实并非如此，我们将在 3.1 节详细讨论这个问题。

对于任意一个可被多线程访问的 Mutable 状态变量，所有对该变量的访问（包括读和写）操作都必须由同一个锁来保护。

对象的内部锁和对象的状态之间并没有强制的关系。使用对象的内部锁来保护代码块只是为了方便，你也可以自己定义锁对象，然后用这个锁对象来保护你的代码块。

每一个共享 Mutable 变量都应该有一个统一的锁来保护，你应该在代码中明确指出这一点，这样维护者就不容易犯错误，例如你可以用@GuardedBy 标注来实现。

实现线程安全类的一个常用方法是将所有对 Mutable 状态域的访问操作都用该对象的内部锁进行保护。这样做不太好，因为当你添加新的方法或者修改代码的时候，你很容易忘记添加这样的保护措施。

只有 Mutable 状态域才需要用锁进行保护，immutable 状态域不需要。

对于一个牵涉到多个状态域的约束，这些状态域中所有的 Mutable 状态域

必须被同一个锁保护。

既然使用同步机制可以防止竞争条件，那么为什么不干脆将所有类中的方法都用 synchronized 关键字修饰呢？事实上，这样做要么会过度同步，要么会同步不足。例如 Vector 类中的 contains 方法和 add 方法都是用 synchronized 关键字修饰的，但如下代码并不是线程安全的：

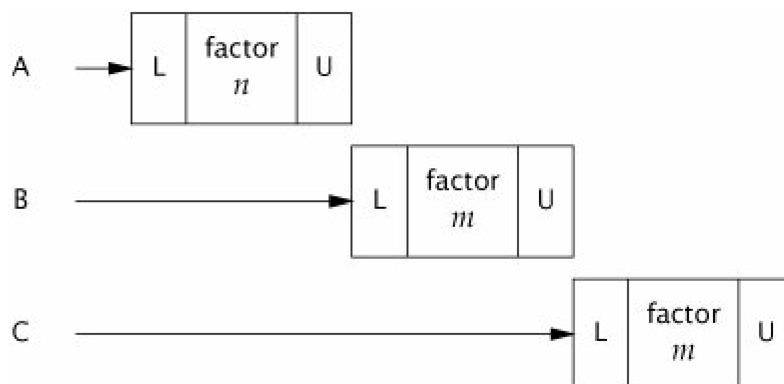
```
if (!vector.contains(element))
    vector.add(element);
```

当多个原子操作组合在一起形成复合操作的时候，还是需要额外的同步。此外，鲁莽地使用 synchronized 关键字来修饰方法可能会造成活跃性和性能问题。

2.5. 活跃性和性能

在 UnsafeCachingFactorizer 类中我们为了提高性能引入了因数分解缓冲机制，但是缓冲机制需要用共享状态域来实现，这就需要同步机制来维持对象状态的完整性。在 SynchronizedFactorizer 类中我们为整个 service 方法添加了 synchronized 修饰符，结果导致了性能低下，因为同时只能有一个线程能够执行 service 方法。这样的方法虽然能够实现多线程安全性，但是付出的性能代价太大。

SynchronizedFactorizer 类的做法违反了 Servlet 框架的基本原则：能够并发响应多个用户请求。这样的做法显然会造成很大的计算资源浪费。下图演示了当多用户请求到达的时候，SynchronizedFactorizer 是如何处理的：它将多个请求缓存起来，然后逐个处理。同时处理的请求数不是受计算资源的限制而是受到程序结构的限制。幸运的是，如果缩小 synchronized 块的范围，就可以在维持线程安全性的前提下大大提高程序的响应能力。但你也应该小心，不能将 synchronized 块缩减得太小，你不能将应该被视为一个原子操作的一段代码分散到多个 synchronized 块中。对于那些不影响共享状态的耗时操作从 synchronized 块中分离出来是一个好主意。



CachedFactorizer 类将 service 中的同步块分为两个部分。第一个 synchronized 块用于检查你是否可以从缓存获得结果。第二个 synchronized 块用于更新缓存。Synchronized 块之外的代码只操作局部变量，这些局部变量只对当前线程有效，不是共享状态的一部分，因此不需要同步。

```
@ThreadSafe
public class CachedFactorizer extends GenericServlet implements Servlet
{
    //上一个需要因数分解的数
    @GuardedBy("this") private BigInteger lastNumber;
    //上一个因数分解结果
    @GuardedBy("this") private BigInteger[] lastFactors;
    //用户访问次数
    @GuardedBy("this") private long hits;
    //命中缓存次数
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits()
    {
        return hits;
    }

    public synchronized double getCacheHitRatio()
    {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp)
    {
        //获取需要因数分解的数字
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
```

```
        //看是否命中缓存，这里要访问4个状态变量，因此必须作为原子操作
        synchronized (this)
        {
            ++hits;
            if (i.equals(lastNumber))
            {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null)
        {
            factors = factor(i);
            //更新缓存，这里要访问2个状态变量，因此需要作为原子操作
            synchronized (this)
            {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
    ……
}
```

CachedFactorizer 类不再使用 AtomicLong 来声明 hits 变量，而是使用 long 类型。当然这里使用 AtomicLong 也是可以的，但是并不能够比使用 long 能提供更多的好处。既然我们已经使用 synchronized 块了，就没必要再使用 AtomicLong 了，同时使用两种同步机制容易让人感到混乱。

CachedFactorizer 类的结构既能够提供线程安全性，又能最大限度地提高响应速度。只有在需要访问 Mutable 状态域的时候才使用 synchronized 块，并且需要作为原子操作的代码都被放置在同一个 synchronized 块中。此外，每个 synchronized 块中的代码都尽量保持短小。像因数分解 factors = factor(i); 这样既耗时又不影响状态域的操作就不用放置在 synchronized 块中。

哪些语句需要放置在 synchronized 块中，synchronized 块的范围应该有多大？要做出这样的决定就必须在简单性和性能之间做平衡，不过线程安全性是必须满足的。大多数时候合理的平衡点是可以找到的。

在设计同步策略的时候尽量不要为了性能而牺牲简单性。

当你使用锁的时候，你应该清楚 synchronized 块中的代码是做什么的，有没有可能执行耗时操作（计算密集型操作和阻塞式操作是两种最常见的耗时操作）。如果 synchronized 块中存在耗时操作，将很有可能引起活跃性问题或者性能问题。

不要将 synchronized 块加在计算密集型操作、网络连接操作和控制台输入输出操作上，否则会引起活跃性和性能问题。

第三章 共享对象

我们在第二章中讲到，编写正确的并发程序的关键是管理好共享 Mutable 域的状态。上一章主要讲了如何用同步机制阻止多线程同时访问这些共享 Mutable 域。本章主要讨论如何编写可以被多线程安全使用的对象。我们也将讨论 java.util.concurrent 库中的设施，它是编写线程安全类和创建线程安全并发程序结构的基础。

我们已经知道如何使用 synchronized 块和 synchronized 方法来实现一组操作的原子性。但是同步的概念不仅包含原子性，也包含内存可见性。我们要确保当一个线程修改了某个共享对象的状态之后，其他线程能够看到该共享对象状态的变化。

3.1. 内存可见性

可见性问题是非常微妙的，因为可见性错误往往是违反直觉的。在单线程程序中，如果你向一个变量写入一个值，随后你可以从该变量中读取这个值。但是在多线程程序中情况就比较复杂了，你无法保证另一个线程向变量写入的值能够被本线程读取到。为了确保多线程间的内存可见性，你必须使用同步机制。

下面的代码演示了当多线程共享数据时可能出现的问题。Main 线程和一个 reader 线程共享变量 ready 和 number。Main 线程先启动 reader 线程，然后为 reader 和 number 赋值。乍一看，打印结果应该是 42，但实际上打印结果很有可能是 0，也有可能程序一直运行下去无法终止。由于同步不足，无法确保 main 线程对 ready 和 number 的赋值能够被 reader 线程看见。

```
public class NoVisibility
{
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread
    {
        public void run()
        {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
```

```
    }

    public static void main(String[] args)
    {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

NoVisibility 类可能无限循环下去，因为在 main 线程中为 ready 赋的值可能永远无法被 reader 线程看见。更奇怪的是，也可能打印 0，因为 reader 可能先看到 ready 为 true，但是在打印的时候还没看到 number 变为 42。这是完全可能的，虽然在 main 线程中先对 number 赋值，后对 ready 赋值，但是在 reader 线程中可能先看到对 ready 的赋值，后看到对 number 的赋值。

在没有同步机制的情况下，编译器和运行时可能对操作执行的顺序做稍微的调整。

NoVisibility 类算是最简单的并发程序了，只有两个线程，两个共享变量，但是仍然容易出错，甚至无法终止。有一种解决这个问题的简单方法：**不管变量什么时候被多线程共享，总是使用合适的同步机制**。

3.1.1. 陈旧数据

上例演示了同步不充分的并发程序可能造成出乎意料的计算结果。当 reader 线程读取 ready 变量的时候，它可能得到陈旧的数据，除非每次访问 ready 变量的时候都使用同步机制，否则这种可能性就会存在。

陈旧数据有时候可能变得非常危险，有可能造成严重的安全问题和活跃性问题。在 NoVisibility 类中，陈旧数据可能导致打印出错误的值，或者导致程序无法终止。一般来说，陈旧数据可能导致严重的，令人疑惑的程序异常、数据结构破坏、计算精度损失和无限循环。

下面的代码中定义的 MutableInteger 类不是线程安全的，因为 get 和 set 方法没有使用同步机制。如果一个线程在调用 set 方法，另一个线程正在调用 get 方法就可能获取到陈旧的数据。

```
@NotThreadSafe
public class MutableInteger
{
```

```
    private int value;

    public int get()
    {
        return value;
    }

    public void set(int value)
    {
        this.value = value;
    }
}
```

如果我们将 get 和 set 方法同步就可以将 MutableInteger 类改造成线程安全类，如下面的代码所示，仅仅同步 set 方法是不够的，get 方法还是可能获取到陈旧的数据，两者都需要同步。

```
@ThreadSafe
public class SynchronizedInteger
{
    @GuardedBy("this")
    private int value;

    public synchronized int get()
    {
        return value;
    }

    public synchronized void set(int value)
    {
        this.value = value;
    }
}
```

3.1.2. 非原子性 64 位操作

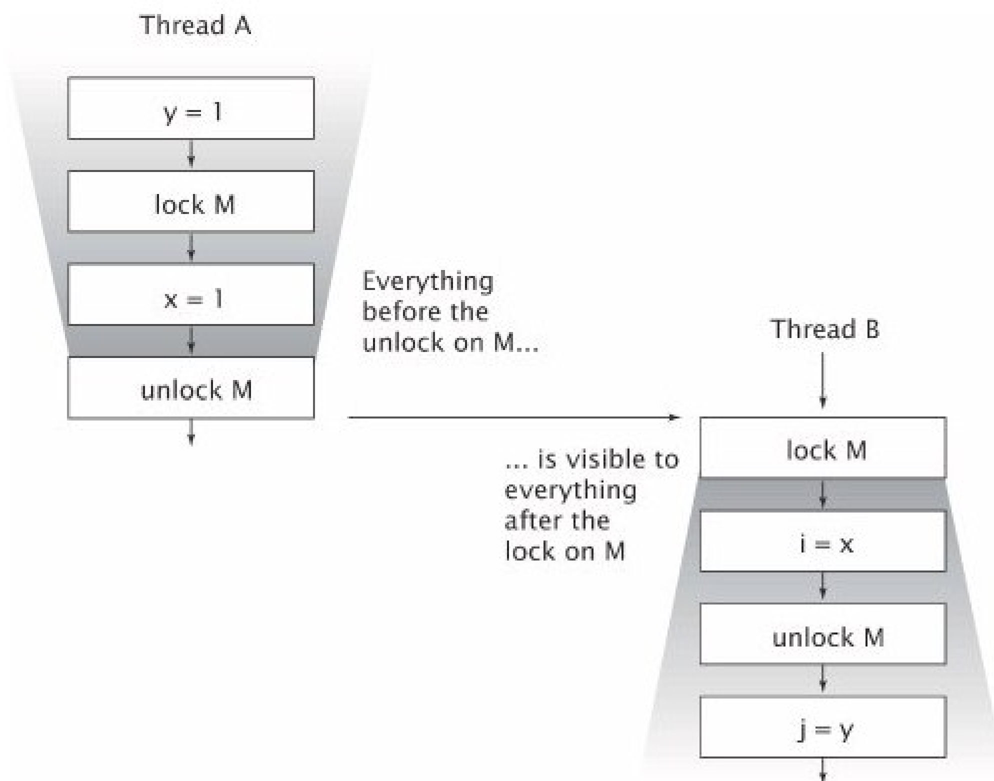
当一个线程读取一个未被同步的共享变量的时候，它可能获取到一个陈旧的数据，但是它应该至少能获取到一个之前被其他线程写入的数据，而不是获取到一个随机值，这被称为最低安全性保障。

64 位非 volatile 类型的变量（包括 double 和 long）比较特殊。JVM 可能将 64 位非 volatile 类型的变量分为两个 32 位数进行读写，从而导致当你读取一个 long 类型共享变量值的时候，高 32 为是一个线程写入的，低 32 位是另一

个线程写入的。因此，即使你不担心陈旧数据的问题，在多线程程序中使用非 volatile 的 long 或 double 类型的共享域都是不安全的。

3.1.3. 锁和可见性

如下图所示，内部锁可以被用来确保一个线程能够以我们期望的方式看到另一个线程对共享域的操作效果。当线程 A 进入被锁 M 保护的 synchronized 块之后，线程 B 也要进入被锁 M 保护的 synchronized 块，但要等到线程 A 从该 synchronized 块中退出之后线程 B 才能进入。线程 A 中对变量 x 的修改可以确保对线程 B 可见。



锁不仅能够提供互斥功能，也可以提供内存可见性。为了确保所有线程都能够看到 Mutable 共享域的最新值，所有对该 Mutable 共享域的读写操作都必须被同一个锁同步。

3.1.4. volatile 域

Java 语言也提供了一种较弱的机制来确保一个线程对共享域的修改能够被其他线程可见。当一个域被声明为 volatile 的时候，编译器和运行时就知道该域是共享域，与其相关的操作不能被改变执行顺序。Volatile 域不会被缓存到

CPU 寄存器中，因此，读取一个 volatile 共享域总是能够获取到其他线程对它的最新赋值。

你可以将 volatile 域的读写操作等价于 SynchronizedInteger 类中带 synchronized 的 get 和 set 方法。不过 Volatile 变量并没有使用锁，因此不会导致执行线程阻塞，因此相比于 synchronized 块，volatile 是一个轻量级的同步机制。

我们并不建议过多地依赖于 volatile 域，使用 volatile 域编写的代码更加脆弱，并且比使用锁实现内存可见性的代码更加难以理解。

下面的代码演示了 volatile 变量的一种典型用法：检查一个状态标志来决定是否退出循环。如果 asleep 域不是 volatile 的，while 所在的线程就无法意识到其他线程对 asleep 标志的修改。我们也可以使用锁来实现内存可见性，但是那样代码就没这么简洁了。

```
volatile boolean asleep;
...
    while (!asleep)
        countSomeSheep();
```

Volatile 域用起来比较方便，但是它也有局限性。Volatile 域最常见的用法是作为完成标志、中断标志和状态标志。比如上例中的 asleep。Volatile 域也可以被用于存储其他类型的标志信息，但是必须小心使用。例如，volatile 语义无法确保 count++操作是原子的，除非你可以确定只有一个线程可对域进行写操作，其他线程都是进行读操作。当然，Volatile 修饰的 Atomic 类型变量还是可以保证原子性的（例如 AtomicLong）。

锁可以保证原子性和内存可见性，而 volatile 域只能确保内存可见性。

只有在满足如下条件的情况下才能使用 volatile 域：

- 对域的写操作不依赖于它的当前值，或者你可以确保只有一个线程能够更新该 volatile 域的值。
- 该 volatile 域不和其他状态域一起组成对象的某个正确性约束。
- 对域的访问确实不需要使用锁。

3.2. 发表与逃逸

发表一个对象意味着使它能够被当前范围之外的代码使用，比如存储一个指

向该对象的引用，可供其他代码使用；从一个非私有方法中返回某个对象或者将某个对象作为参数传入其他类的方法中。如果一个对象被不恰当地发表了，就称为逃逸。

最明显的发表形式是将一个对象的引用存储在 public static 域中，这样任何类和线程都能够访问它。如下代码所示，initialize 方法创建了一个 HashSet 对象然后将引用赋值给 public 静态域 kownSecrets。

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

发表一个对象可能间接地发表其他对象，如果在 initialize 方法中为 knownSecrets 添加一个 Secret 元素，那么你也发表了这个 Secret 元素。因为任何代码都可以通过遍历 knownSecrets 来获取对该 Secret 元素的引用。相似地，从一个非 private 方法中返回一个对象引用也发表了这个对象，如下代码将私有的数组对象通过 return 语句发表出去：

```
class UnsafeStates
{
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates()
    {
        return states;
    }
}
```

发表一个对象也发表了它的非 private 域所引用的所有对象。更确切地说，所有在非 private 域和非 private 方法调用链上可达的对象都被一起发表了。

从类 C 的角度看，异型方法是那些行为不被 C 掌握的方法，包括所有其他类中的方法，和类 C 中的可覆盖方法（非 private 且非 final 的方法）。将一个对象作为参数传递给异型方法就发表了这个对象。这是因为你不知道传入的对象将被什么代码使用。

不管是否会有其他线程访问你发表的对象，风险都是存在的，一旦有对象逃逸，你应该设想其他类或者线程会恶意地或无意地错用这个对象。因此，我们有

足够的理由使用封装，因为它使得分析程序的正确性变得实际可行。关于封装，下一章会详述。

最后一个发表内部状态的方法是发表内部类的实例。如下代码所示，当 ThisEscape 发表内部匿名类实例的时候，把 this 对象也发布出去了，因为非静态内部类的实例隐式地包含了对外部类实例的引用。

```
public class ThisEscape
{
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }

    void doSomething(Event e)
    {
    }

    interface EventSource
    {
        void registerListener(EventListener e);
    }

    interface EventListener
    {
        void onEvent(Event e);
    }

    interface Event
    {
    }
}
```

3.2.1. 安全构造实践

上例中的 ThisEscape 类在其构造函数中发表非静态内部类的实例，不小心把 this 也发表出去了，但是由于构造函数还没有退出，发表出去的 this 是一个没有完全构造好的对象，这是非常危险的。

不允许在对象构造完成之前发表 this。

一个常见的导致在对象构造完成之前发表 this 的错误是在构造函数中创建一个线程并将 this 做为参数传入线程对象，然后启动该线程。这个类和线程类共享 this 对象。新创建的线程就可以在对象构造完成之前访问 this 对象。

你可以在构造函数中创建线程对象，但不要立即启动它，应该在构造函数之外，创建额外的方法来启动线程。在构造函数中调用一个可被覆盖的实例方法也可能造成 this 对象的逃逸。

如果你想在构造函数中注册一个事件监听器或者启动一个线程，你可以先创建一个 private 构造函数，然后创建一个 public 工厂方法，这样就可避免在对象构造完成之前发表 this，如下例所示：

```
public class SafeListener
{
    private final EventListener listener;

    private SafeListener()
    {
        listener = new EventListener()
        {
            public void onEvent(Event e)
            {
                doSomething(e);
            }
        };
    }
    public static SafeListener newInstance(EventSource source)
    {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

3.3. 线程封闭

访问共享 Mutable 对象需要使用同步机制。如果不共享该 Mutable 对象也就不需要使用同步机制了。线程封闭是达到线程安全性目的的最简单的方法。如果一个对象被封闭在一个线程中，那么它肯定是线程安全的。

Swing 框架中广泛地使用了线程封闭技术。可视化组件和数据模型不是线程安全的，如果只在事件分派线程中访问它们将会是线程安全的。在其他线程中运

行的代码不应该访问这些对象。Swing 中的很多并发错误都是由于在其他线程中对这些组件和数据模型的不恰当访问造成的。

另一个常见的使用线程封闭技术的例子是 JDBC 连接池。连接池中的 Connection 对象不是线程安全的。在典型的服务器端应用程序中，一个线程从连接池中获取一个 Connection 对象，将其用于处理单个客户端请求，然后将其返还给连接池。由于大多数客户端请求的处理过程都是单线程的，并且连接池也不会同时将一个连接发送给多个线程使用，这种 Connection 对象管理模式隐式地将一个 Connection 对象封闭在了客户端请求对应的单个线程之中了。

Java 语言没有提供给变量强制加锁的机制，同样也没有提供将对象限制在某个线程中的机制。线程封闭只是一种程序设计技巧，是编程者在编写程序的时候自己实现的。编程者必须确保被封闭的对象不被其他线程共享。

3.3.1. Ad-hoc 线程封闭

Ad-hoc 线程封闭指的是线程封闭的责任完全落到了编程者的肩上，编程者要确保被封闭的对象不被其他线程共享。这种线程封闭的方法比较脆弱，因为没有使用语言或核心库的支持。

使用线程封闭技术往往是由于要构造一个单线程子系统造成的。单线程子系统有很多好处。

线程封闭的一个特例是 volatile 域。如果你可确定只有一个线程可对 volatile 域执行写操作，那么你就阻止了竞争条件的出现，并且 volatile 域的可见性确保了其他线程能够看到该域的最新值。

由于 Ad-hoc 线程封闭比较脆弱，尽量不要使用。你可以使用更强大的线程封闭技术（堆栈封闭和 ThreadLocal）。

3.3.2. 堆栈线程封闭

堆栈线程封闭是一种特殊的线程封闭技术。局部变量使得将一个对象封闭到一个线程中更容易。局部变量与生俱来就属于其所在的执行线程，它们存在于执行线程的堆栈之中，其他线程是无法访问到的。

原始数据类型的局部变量总是满足堆栈线程封闭条件的，因为你无法获取一个原始数据类型变量的引用，语言的语义就确保了原始数据类型的局部变量总是线程封闭的。

```java
public int loadTheArk(Collection<Animal> candidates)
{
    Set<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals)
    {
        if (candidate == null || !candidate.isPotentialMate(a))
        {
            candidate = a;
        }
        else
        {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

在上例中我们实例化了一个 TreeSet 对象，然后将其引用存储在 animals 局部变量中，这时候只有一个指向该 TreeSet 对象的引用，并存放在局部变量中，因此该 TreeSet 对象是线程封闭的。不过，如果我们发表了这个引用，就会打破线程封闭。

如果一个对象是线程封闭的，即使这个对象不是线程安全的，也不会造成线程安全问题。但要注意，这个线程封闭约束只存在于实现者的头脑之中。如果没有以文档约束的方式记录下来，将来的维护者可能不知道某个对象必须是线程封闭的，从而造成问题。

3.3.3. ThreadLocal

实现线程封闭的最正式的方式是使用 ThreadLocal，它可以将只能被单线程访问的对象绑定到一个值保持对象。

例如由于 JDBC 中的 Connection 对象不是线程安全的，如果多线程使用一个共享的 Connection 对象将会造成问题。通过使用 ThreadLocal，每个线程将会

获得自己的 Connection 对象，各个线程之间互不干扰。代码如下：

```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue()
        {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection()
{
    return connectionHolder.get();
}
```

当一个线程第一次调用 connectionHolder.get 方法的时候，将会调用 initialValue 方法获取初始值，以后再调用 get 方法的话将会返回初始创建的那个对象。你可以将 ThreadLocal<T>想象为一个 Map<Thread,T>，用来存储线程到 T 对象的映射关系，虽然事实上并不是这样实现的，线程特定的对象是存储在 Thread 对象中的，在线程终止之后将会被垃圾回收器回收。

注意，不要滥用 ThreadLocal。

3.4. 不可变对象

几乎所有的原子性和可见性风险都是由于多线程同时访问共享 Mutable 域引起的。如果一个对象的状态不能被修改，那么这种风险也就不复存在了。

不可变（Immutable）对象是那种在创建完成之后其状态不能被改变的对象。

Immutable 对象与生俱来就是线程安全的。

Immutable 对象比较简单。程序设计的一个最困难的地方就是搞清楚复杂对象的所有可能的状态。对于 Immutable 对象来说这根本不是问题，因为它们只能有一种状态。

Immutable 对象比 Mutable 对象安全得多。将 Mutable 对象传递给其他代码是危险的，因为不可信的代码可能修改 Mutable 对象的状态。然而 Immutable 对象不担心恶意代码会对其进行修改，它们不需要进行保护性拷贝即可发表。

Java 语言语法和内存模型都没有正式地定义不可变性。但是不可变性并不是仅仅将引用某个对象的域声明为 final 这么简单。一个所有域都是 final 的对象可能不是 Immutable 的，因为 final 域可能引用 Mutable 对象。

当且仅当一个对象满足如下三个条件，它才是 Immutable 的：

- 在创建之后其状态不能被改变
- 所有域都是 final 的
- 在构造函数中，this 对象不能逃逸

Immutable 对象内部可以使用 Mutable 对象表示其状态。如下面的代码所示，尽管 stooges 域是 Mutable 的，但是 ThreeStooges 类的设计使得外部调用者无法修改 stooges，因此 ThreeStooges 对象创建之后其状态不能被修改，这满足了第一个条件。stooges 域是 final 的，这满足第二个条件。另外，构造函数中也没有允许 this 对象逃逸，满足了第三个条件。因此 ThreeStooges 类是 Immutable 的。

```
public final class ThreeStooges
{
    private final Set<String> stooges = new HashSet<String>();
    public ThreeStooges()
    {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name)
    {
        return stooges.contains(name);
    }
}
```

由于程序的状态一直在变化，你可能认为 Immutable 对象用处不大，事实上并非如此。对象不可变与对象引用不可变是两个概念。虽然你不能改变 Immutable 对象的状态，但你可以改变引用，让其指向具有新状态的 Immutable 对象。下一节会提供一个这方面的例子。

3.4.1. final 域

final 关键字支持 Immutable 对象的构建。final 域初始化之后不能被修改（如果它指向一个 Mutable 对象的话，这个 Mutable 对象的状态还是可以被修改的）。就是这个特性导致 Immutable 对象可以被多线程使用而不需要同步。

final 域的使用减少了对象可能的状态，对象的状态越少就越容易使用。就像如果没理由让一个域有更大的访问权限的话，应该将其声明为 private 的一样，如果没有理由让一个域为非 final 的，应该尽量将其声明为 final 的。

3.4.2. 使用 Volatile 来发表 Immutable 对象

在 2.3 节的 UnsafeCachingFactorizer 类中，我们使用了两个 AtomicReference 域来存储上一个要分解的数和上一个分解结果。但是这个 Servlet 不是线程安全的，因为我们无法原子地更新两个域。然而 Immutable 对象可以提供一种原子性。

UnsafeCachingFactorizer 类中有两种操作必须保持原子性：更新缓存和在匹配的情况下从缓存中取得分解结果。当一组相关的数据的操作必须满足原子性的时候，你应该考虑将其封装在一个 Immutable 类中，如下例所示：

```
@Immutable
public class OneValueCache
{
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,BigInteger[] factors)
    {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i)
    {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

如下代码使用了 OneValueCache 类来构建一个线程安全的因数分解 Servlet：

```
@ThreadSafe
public class VolatileCachedFactorizer extends GenericServlet implements
                                                                Servlet
```

```
{
    private volatile OneValueCache cache = new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        If(factors == null)
        {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
    ……
}
```

由于被声明为 volatile 的，当一个线程将 cache 域修改为指向一个新的 OneValueCache 对象之后，这个新的对象立刻就能够被其他线程看见和访问。

这种 Immutable 对象和 Volatile 域的结合不仅满足了原子性也满足了内存可见性，这样，虽然没有使用任何显式的锁，VolatileCachedFactorizer 类也是线程安全的。

3.5. 安全发表对象

迄今为止，我们专注于尽量不发表对象，将其封闭在某个线程中或将其封闭在某个对象中。但有时候我们不可避免地需要在线程间传递对象，这就涉及到安全性。不幸的是，即使是简单地将一个对象发表为某个 public 域都难保是安全的，代码如下：

```
// Unsafe publication
public Holder holder;

public void initialize()
{
    holder = new Holder(42);
}
```

你可能想象不到上面的代码有多糟糕，对于其他线程来说 holder 可能处于不一致状态，即使 Holder 类的构造函数能够保证其正确性约束。

3.5.1. 不合理的对象发表方式

你不能依赖于没有构建完成的对象的完整性。事实上如果 Holder 对象被以像上一节那样的方式发表的话，另一个线程调用 assertSanity 方法可能会抛出 AssertionError 异常，Holder 类代码如下：

```
public class Holder
{
    private int n;

    public Holder(int n)
    {
        this.n = n;
    }

    public void assertSanity()
    {
        if (n != n)
            throw new AssertionError("This statement is false.");
    }
}
```

这是因为发表 Holder 对象的时候没有使用同步，其他线程可能看不到 public holder 的改变。由此可能引起两种问题：第一，另一个线程可能看到陈旧的 holder 值，也就是 null。第二，另一个线程可能在执行 assertSanity 过程中看到 holder 指向了一个新的 Holder 对象，导致两次读取的 n 不相等，从而抛出 AssertionError 异常。

当在多线程之间共享数据，又没有足够的同步措施的话，将会发生非常诡异的事情。

3.5.2. Immutable 对象和初始化安全

由于 Immutable 对象及其重要，Java 内存模型为其提供了特别的保证和初始化安全。Immutable 对象在任何情况下都是线程安全的，即使是在缺乏同步措施的情况下发表 Immutable 对象也没有问题。如果上面的 Holder 类是 Immutable 的话，将 Holder 对象发布到 public 的 holder 域就不会有任何问题。

这种特别的保证可以扩展到所有 final 域。如果上面的 public 的 holder 域是 final 的话，这样的发表方式也是可以的。但是如果 final 域指向的是

Mutable 对象的话，使用者线程对该 Mutable 对象的操作需要额外的同步机制。

3.5.3. 安全的对象发表方式

Mutable 对象必须被安全地发表，发表者线程和使用者线程都需要使用同步机制。**本节我们主要确保使用者线程能够看到正确的对象状态。**如下四种方法可以安全地发表一个对象：

- 在静态初始化范围中初始化一个对象引用
- 将对象的引用存储在一个 volatile 域中或在一个 AtomicReference 域中
- 将对象的引用存储在一个构建良好的对象的 final 域中
- 将对象的引用存储在一个被锁保护的域中

线程安全集合（例如 Vector 和 SynchronizedList）的内部同步机制就是使用上述的第四种方式。如果线程 A 将对象 X 插入到一个线程安全集合中，线程 B 随后从这个线程安全集合中获取对象 X，那么获取到的对象 X 的状态就是线程 A 插入的对象的状态。即使调用者没有使用任何同步机制，线程安全类也能够保证线程安全性。

线程安全库中的一些类提供了如下安全发表功能：

- 将一个键或者值对象放置在 HashTable、SynchronizedMap 或者 ConcurrentMap 中都安全地将其发表给其他线程。
- 将一个对象放置在 Vector、CopyOnWriteArrayList、CopyOnWriteArraySet、SynchronizedList 或者 SynchronizedSet 中都安全地将其发表给其他线程。
- 将一个对象放置在 BlockingQueue 或者 ConcurrentLinkedQueue 中安全地将其发表给其他线程。

使用静态域初始化的方式是最简单也最安全的对象发表方式，例如：

```
public static Holder holder = new Holder(42);
```

静态域初始化是在 JVM 加载类的时候执行的。由于 JVM 的内部同步机制，这种发表对象的方式总是安全的。

3.5.4. 等效 Immutable 对象

如果对象发表之后，使用者不会修改对象的状态，那么上面介绍的方法就可

以保证对象发表的安全性了。对象发表之后其他线程能够立即看到新的对象。

如果一个对象不是 Immutable 的，但是发表之后，使用者不会修改其状态，那么这种对象被称为等效 Immutable 对象。使用等效 Immutable 对象可以简化开发并且提高性能。

安全发表的等效 Immutable 对象可以被其他线程安全地使用（不需要额外的同步措施）。

例如，Date 对象是 Mutable 的，但是你可以将其作为 Immutable 对象使用。假设你要维护一个 Map，用来存储用户名到用户登陆时间的映射。

```
public Map<String, Date> lastLogin =
    Collections.synchronizedMap(new HashMap<String, Date>());
```

如果这些 Date 对象在被放置到 Map 中之后不需要被修改的话，那么 SynchronizedMap 类就能提供足够的安全性，不需要额外的同步措施来存取它们。

3.5.5. Mutable 对象

如果一个对象在安全发表之后可能被使用者修改，那么使用者读写该对象的操作都必须使用同步机制。

现在，对于不同类型的对象的多线程共享问题总结如下：

- 对于 Immutable 对象，可以使用任何方式发表，不用担心线程安全问题。
- 对于等效 Immutable 对象，必须被安全地发表（3.5.3 节的四种方法）。
- 对于 Mutable 对象，不仅需要安全地发表，并且使用者对其读写操作还需要使用同步机制。

3.5.6. 安全地共享对象

每当你获取一个对象的引用的时候，你应该知道你被允许如何操作它。使用这个对象之前需要获得某个锁吗？你可以修改它的状态吗？当你发表一个对象的时候你应该在文档中写下该对象应该如何被使用。

在使用者线程中访问对象应该遵循如下准则：

线程封闭对象：一个线程封闭对象只属于一个线程，只能被该线程访问。

共享只读对象：一个共享只读对象可以被其他线程并发读取，但不能被他们修改。

线程安全对象：一个线程安全对象内部使用了同步机制，因此多线程可以通过 public 接口安全地使用它而不需要额外的同步措施。

锁保护对象：只有在使用者线程持有某个锁的时候才能访问。

第四章 组合对象

迄今为止，我们讨论了线程安全和同步的一些基础内容。但是我们不希望每次编写多线程程序的时候都要仔细分析每一个内存访问操作以确保我们的整个程序是线程安全的（这样做也不太现实）。我们希望能够将一些线程安全组件组合成更大的线程安全组件，从而构成整个程序。本章主要讨论如何构建类层次结构，使得它们的线程安全性不容易被破坏。

4.1. 设计一个线程安全类

虽然将程序中所有状态信息都保存在 public 静态域中也可以设计出线程安全的应用程序，但是这样做很难验证程序的线程安全性（因为整个程序一般都非常庞大），并且程序修改后很难保证它还是线程安全的。封装使得不需要检查整个程序的代码就能确定一个类是否是线程安全的。

线程安全类的设计过程应该包含如下三个基本元素：

- 识别组成对象状态的域
- 识别对象的约束
- 建立一个管理对象状态的并发存取的策略

如果对象的域全部是原始数据类型，那么这些域就构成了对象的状态。下面的 Counter 类只有一个 long 类型的域 value，因此 value 域就构成了 Counter 对象的状态。如果一个对象的域是指向其他对象的引用类型的话，那么该对象的状态还包含被引用对象的状态。例如一个 LinkedList 对象的状态包含所有列表中节点对象的状态。

同步策略指定了使用哪些方法（包括不变类、线程封闭、锁等）来实现类的线程安全性。为了确保你写的线程安全类能够被分析和维护，你应该在类注释中记下你使用了什么样的同步策略。

```
@ThreadSafe
public final class Counter
{
    @GuardedBy("this")
    private long value = 0;

    public synchronized long getValue()
    {
```

```
        return value;
    }

    public synchronized long increment()
    {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
```

4.1.1. 收集同步需求

要想将一个类设计成线程安全的，必须确保它在多线程存取条件下还能维持其约束条件。首先需要搞清楚对象的状态。对象的所有可能的状态组成其状态空间。对象的状态空间越小就越容易分析，因此你应该尽量将所有的域都声明为 final 的。

很多类都有约束条件，用以确定对象在指定的状态下是否是合法的。例如上面的Counter类中的value域是long类型的，它的状态空间为从Long.MIN_VALUE 到 Long.MAX_VALUE 的所有整数。但是 Counter 类的约束条件要求 value 值不能为负数。

相似地，操作一般都有后置条件，用来判断指定的状态转换是否合法。例如 Counter 对象的当前值是 17，下一个合法的值应该是 18。当下一个状态依赖于当前状态的时候，这个状态转换操作实际上是一个复合操作。也有些操作的下一个状态不依赖于当前状态，例如更新一个表示当前温度的变量，其新值不依赖于旧值。

类的约束条件和操作的后置条件提出了额外的同步需求。如果对象在某些状态下是非法的，那么该对象的状态域必须被封装，否则客户端代码可能修改状态域，把该对象设置成非法状态。对于那些可能引起非法状态转换的操作，必须将其设计成原子的。

一个约束条件有时可能约束多个状态变量，例如类 NumberRange 中维护了数据范围的下限和上限两个状态变量。约束条件是下限域必须小于或等于上限域。涉及多状态域的约束提出了原子性同步需求：对相关状态域的取值和更新必须放在一个原子操作中进行。

如果你不理解类的约束条件和后置条件就不能设计出线程安全类。类的约束条件和操作的后置条件可能提出原子性需求或封装需求。

```
public class NumberRange
{
    // INVARIANT: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public synchronized void setLower(int i)
    {
        // 涉及到两个状态域，setLower方法应该是原子的
        if (i > upper.get())
            throw new IllegalArgumentException("can't set lower to " + i
                    + " > upper");
        lower.set(i);
    }

    public synchronized void setUpper(int i)
    {
        //涉及到两个状态域，setUpper方法应该是原子的
        if (i < lower.get())
            throw new IllegalArgumentException("can't set upper to " + i
                    + " < lower");
        upper.set(i);
    }

    public synchronized boolean isInRange(int i)
    {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

4.1.2. 状态依赖方法

除了类约束条件和方法后置条件以外，有的时候还会涉及到方法前置条件。例如你不能删除一个空队列中的元素，在执行删除元素方法之前，队列对象必须处于 nonempty 状态。具有基于状态的前置条件的方法被称为状态依赖方法。

在单线程程序中，如果一个方法的前置条件不满足，则该方法将会运行失败，但是在并发程序中，可能需要等待其他线程的动作在稍后将前置条件变为 true，然后再继续执行本方法。

Object 类中的 wait 和 notify 方法是和对象的内部锁紧密绑定的，它们很难使用。要为状态依赖方法提供等待前置条件变为 true 然后再继续运行的机制，最好使用一些 Java 类库中的系统类，例如阻塞式队列或者信号量，不要使用 wait 和 notify 方法。后面章节还会讲到这个问题。

4.1.3. 对象所有权

组成对象的状态的一组域是对象所有域的子集。在定义类的时候我们需要清楚哪些域组成了对象的状态。例如你创建了一个 HashMap 对象，并向其中添加了一些键值对，事实上你创建了很多对象，一个 HashMap 对象，一组 Map.Entry 对象，可能还有其他一些内部对象。HashMap 对象的逻辑状态由这些 Map.Entry 对象和其他一些内部对象的状态组成。

不管怎样，垃圾回收器使我们不用过多地考虑对象所有权的问题。很多时候，对象所有权和封装紧密相连。所有权意味着控制权，但是一旦你将一个 mutable 对象的引用发表出去，其他类也拥有了控制权。一般来说一个类的方法和构造函数不拥有传入参数引用的对象的控制权（不要修改参数对象的状态），除非该方法是专门被设计用来传递控制权的。

集合类往往表现出所有权的分裂，集合拥有自己的状态，但是客户端代码拥有集合元素的状态。例如 ServletContext 类提供了类似于 Map 的功能，它提供了 setAttribute 和 getAttribute 方法，用于按名字注册和取回对象。这个类是线程安全的，调用 setAttribute 和 getAttribute 方法不需要使用同步机制，但是使用 ServletContext 类中存放的对象就需要使用同步机制了，因为这些对象是被多线程共享的。因此，这些对象必须是线程安全的或者是 Immutable 的，或者是被某个锁保护的（**注意，这一点常常被忽略**）。

4.2. 实例封闭

即使一个对象不是线程安全的，你也可以使用一些技术，使其可用在并发程序中。你可以使用线程封闭技术或者使用锁来保护所有对该对象的访问操作。

封装简化了线程安全类的设计。当一个对象被封装在另一个对象中的时候，所有访问该被封装对象的访问路径都是可控的。组合使用实例封闭和锁可以设计出线程安全类。

将一个对象封装入另一个对象可以将所有对被封装对象的访问都集中在外

部对象的方法中。这样就很容易确保被封装对象的存取被特定的锁保护。

被封闭的对象不能逃逸其应该在的范围。因此，设计者发表对象的时候应该小心。下面代码中的 PersonSet 类演示了如何使用实例封闭和锁来设计出线程安全类。PersonSet 对象的状态由一个 HashSet 对象 mySet 来表示。尽管状态变量 mySet 不是线程安全的，但是由于 mySet 是 private 的，因此这个 HashSet 对象被封闭在类 PersonSet 对象中。能够访问 mySet 的路径只有两条：addPerson 方法和 containsPerson 方法，由于这两个方法都有锁保护，所以 PersonSet 类是线程安全的。

```
@ThreadSafe
public class PersonSet
{
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p)
    {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p)
    {
        return mySet.contains(p);
    }

    interface Person
    {
    }
}
```

实例封闭是创建线程安全类的一个最简单的方式。它也允许灵活地选择加锁策略。本例是使用 PersonSet 对象的内部锁来实现线程安全的，但是使用其他的锁也是可以的。

Java 类库中的很多类都使用了实例封闭技术来实现线程安全性。例如 ArrayList 是非线程安全的，但是类库中提供了 Collections.synchronizedList 方法创建了一个线程安全的包装类，它将一个 ArrayList 对象封闭在其中，且包装类中所有的方法都加了锁，因此可以提供线程安全性（你不应该保留传入的 ArrayList 对象的引用）。

当然，被封闭的对象还是可能逃逸它应该在的范围的。如果发生逃逸，那么这一定是个 Bug。发表一个 iterator 或者内部类实例都可能导致被封闭的对象的间接逃逸。

4.2.1. Java 监视器模式

遵循实例封闭的原则，你可以使用 Java 监视器模式来设计线程安全类。你需要将所有 mutable 状态域都封装起来，然后用对象的内部锁来保护它们。Java 监视器模式的最大的优点是其简单性。

4.1 节中的 Counter 类就是使用 Java 监视器模式设计的类。Java 监视器模式只是个习惯用法，事实上不一定非要用对象的内部锁，其他锁也可以。如下代码演示了使用私有锁来保护对象状态域的情况。

```
public class PrivateLock
{
    private final Object myLock = new Object();
    @GuardedBy("myLock")
    Widget widget;

    void someMethod()
    {
        synchronized (myLock)
        {
            // Access or modify the state of widget
        }
    }
}
```

使用私有锁对象而不是内部锁有很大的好处。使用私有锁使得客户端代码无法获取该锁对象。如果 PrivateLock 使用内部锁来保护 widget 域，客户端代码也使用 PrivateLock 对象的内部锁来保护其他变量，这就可能造成活跃性问题。

4.2.2. 一个实例封闭的例子

4.1 节的 Counter 类太简单了，我们来看一个稍微复杂一点的例子。我们要创建一个车辆追踪器类。首先我们将使用 Java 监视器模式来创建它，然后看看如何放松一些封装需求，同时保持线程安全性。

每个车辆都由一个 String 字符串标识，并且每个车辆都由一个坐标（x,y）表示其位置。VehicleTracker 类封装了所有车辆的标识符和位置。下面的代码

使用 Java 监视器模式来实现线程安全的车辆追踪器类：

```
@ThreadSafe
public class MonitorVehicleTracker
{
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(Map<String, MutablePoint> locations)
    {
        // 使用了保护性拷贝技术，防止本类对传入对象的修改
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations()
    {
        // 使用了保护性拷贝技术，防止发表locations域
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id)
    {
        MutablePoint loc = locations.get(id);
        // 使用了保护性拷贝技术，防止发表locations域中的对象
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y)
    {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    /**
     * 静态方法，深度拷贝
     */
    private static Map<String, MutablePoint> deepCopy(Map<String,
                                                          MutablePoint> m)
    {
        Map<String, MutablePoint> result =
                                          new HashMap<String, MutablePoint>();
```

```
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));

        return Collections.unmodifiableMap(result);
    }
}
@NotThreadSafe
public class MutablePoint
{
    public int x, y;

    public MutablePoint()
    {
        x = 0;
        y = 0;
    }

    public MutablePoint(MutablePoint p)
    {
        this.x = p.x;
        this.y = p.y;
    }
}
```

虽然 MutablePoint 类不是线程安全的，但是 MonitorVehicleTracker 类是线程安全的。locations 域所引用的 Map 对象以及 Map 内部的对象都没有被发表。当我们需要返回一个车辆位置的时候，我们使用了保护性拷贝技术。

这里使用了保护性拷贝技术，在大多数情况下这都不会引起性能问题，但是当列表非常大的时候就不太好用了。保护性拷贝引起的另一个问题是，getLocations 方法返回的 Map 是不会随着 MonitorVehicleTracker 对象的状态变化而变化的。这种特性好于不好取决于你的需求。

4.3. 线程安全性代理

除了那些最简单的对象，大多数对象都是组合对象。Java 监视器模式适合用于从底层开始构建线程安全类或者封装一个或多个非线程安全子组件来构建线程安全类。但是，如果子组件就是线程安全的，我们还需要再加上一层线程安全性吗？这需要具体情况具体对待。

4.2.2 节中的表示车辆位置的类是非线程安全的，我们现在来创建一个

Immutable 的位置对象，表示车辆的位置。

```
@Immutable
public class Point
{
    public final int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

4.3.1. 将线程安全性代理给予组件

使用这个 Immutable 的位置对象，再将状态域改成线程安全的 Map，我们就可以给出线程安全的车辆追踪器类的另一种实现：

```
@ThreadSafe
public class DelegatingVehicleTracker
{
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points)
    {
        // 用points中的键值对作为初始值创建新Map，由于键和值都是Immutable
        // 对象，因此不会有问题（不需要保护性拷贝）
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Point getLocation(String id)
    {
        // 由于Point类是Immutable的，直接发表也不会有问题
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y)
    {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException("invalid vehicle name: " + id);
    }
```

```
    // 获取up-to-date车辆位置信息
    public Map<String, Point> getLocations()
    {
        return unmodifiableMap;
    }

    // 获取快照式车辆位置信息
    public Map<String, Point> getLocationsAsStatic()
    {
        return Collections.unmodifiableMap(new HashMap<String,
                                                Point>(locations));
    }
}
```

注意，这里提供了两种 getLocations 方法，一个是快照式的，另一个是联动的。getLocationsAsStatic 方法中直接将 locations 作为参数创建一个新的 HashMap 对象，两个 Map 共享了相同的内容，但是由于内容是 Immutable 的（String 和 Point 都是 Immutable 类）这不会造成问题。

DelegatingVehicleTracker 类是线程安全的，它把线程安全性代理给了 locations 域。

4.3.2. 相互独立的状态域

上一节的代理技术将线程安全责任代理给单个线程安全的状态域。我们也可以将线程安全责任代理给多个线程安全的状态域，只要这些状态域是相互独立的。独立的意思是说组合类不在多个状态域之上加任何约束条件。

如下代码中的 VisualComponent 类有两个状态域，分别是 keyListeners 和 mouseListeners，它们分别表示键盘事件监听器列表和鼠标事件监听器列表，两者没什么关系，因此它们是相互独立的。此外两者都是线程安全的，因此 VisualComponent 类可以将它的线程安全性责任代理给这两个状态域。

```
public class VisualComponent
{
    private final List<KeyListener> keyListeners
                            = new CopyOnWriteArrayList<KeyListener>();
    private final List<MouseListener> mouseListeners
                            = new CopyOnWriteArrayList<MouseListener>();

    public void addKeyListener(KeyListener listener)
    {
```

```
        keyListeners.add(listener);
    }

    public void addMouseListener(MouseListener listener) {
        mouseListeners.add(listener);
    }

    public void removeKeyListener(KeyListener listener) {
        keyListeners.remove(listener);
    }

    public void removeMouseListener(MouseListener listener) {
        mouseListeners.remove(listener);
    }
}
```

上例中每个列表都是 CopyOnWriteArrayList 类型的，这是个线程安全类，非常适合用于管理监听器列表。由于两个状态域都是线程安全的，并且没有约束条件加在两者之上，因此 VisualComponent 类可以将线程安全性责任代理给它们，换句话说，VisualComponent 虽然没有使用锁，但仍是线程安全类。

4.3.3. 代理失效怎么办

大多数组合类并不像上例所示的那么简单，它们在多个状态域之上加约束条件，例如如下的 NumberRange 类，它要求 lower 域的值总是小于或等于 upper 域的值。

```
public class NumberRange
{
    // INVARIANT: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i)
    {
        // Warning -- unsafe check-then-act
        if (i > upper.get())
            throw new IllegalArgumentException("can't set lower to " + i
                    + " > upper");
        lower.set(i);
    }

    public void setUpper(int i)
```

```
    {
        // Warning -- unsafe check-then-act
        if (i < lower.get())
            throw new IllegalArgumentException("can't set upper to " + i
                + " < lower");
        upper.set(i);
    }

    public boolean isInRange(int i)
    {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

显然，NumberRange 类不是线程安全的。它不能保证 lower 和 upper 之间的约束。setLower 和 setUpper 方法不是原子性的，可能导致 NumberRange 对象处于非法状态。尽管 AtomicInteger 类是线程安全的，NumberRange 类却不是线程安全的。因为 lower 和 upper 域不是相互独立的，NumberRange 不能简单地将线程安全性代理给它们。

在这种情况下，由于 setLower 和 setUpper 方法都是复合操作，因此光是用线程安全性代理是不够的，还要使用锁，用 synchronized 关键字修饰它们就可以了。

4.3.4. 发表状态域

如果你将组合对象的线程安全性代理给状态域，那么在什么情况下你可以发表这个状态域呢？这取决于组合对象是否在这些状态域上加约束条件。例如 Counter 类中的 value 域不能是负数，如果你发表了这个域，则客户端代码就有可能将其设置为负数，从而使 Counter 对象处于非法的状态。如果 Counter 类对 value 没有约束的话，发表 value 是可以的（如果 value 是 mutable 对象的话，最好还是使用保护性拷贝，不要直接发表）。

4.3.5. 例子：发表其状态域的车辆追踪器类

让我们来构建车辆追踪器类的另一个版本。这次使用的位置对象是线程安全的 Mutable 对象，代码如下：

```
@ThreadSafe
public final class SafePoint
{
```

```
    @GuardedBy("this")
    private int x, y;

    private SafePoint(int[] a)
    {
        this(a[0], a[1]);
    }

    public SafePoint(SafePoint p)
    {
        this(p.get());
    }

    public SafePoint(int x, int y)
    {
        this.set(x, y);
    }

    public synchronized int[] get()
    {
        return new int[]{x, y};
    }

    public synchronized void set(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

get 方法返回了一个包含两个元素的数组，如果将其拆分为 getX 和 getY 两个方法，就有可能存在两个方法调用之间坐标值被其他线程改变的情况，这样就会得到非法的位置信息，此处的 get 方法设计得比较合理。使用 SafePoint 类，我们就可以构建出一个可发表其状态域的车辆追踪器类，代码如下：

```
@ThreadSafe
public class PublishingVehicleTracker
{
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(Map<String, SafePoint> locations)
    {
        this.locations = new ConcurrentHashMap<String, SafePoint>(locations);
```

```
        this.unmodifiableMap = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations()
    {
        return unmodifiableMap;
    }

    public SafePoint getLocation(String id)
    {
        // 发表了状态域locations中的对象，由于该对象是线程安全的，且本类又没
        // 有加任何约束条件，因此不会引起问题
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y)
    {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException("invalid vehicle name: " + id);
        locations.get(id).set(x, y);
    }
}
```

这个类将线程安全性代理给 locations 域，这次 Map 中的值对象是 SafePoint 类型，这是个线程安全类但不是 Immutable 类。getLocations 方法返回了一个不能修改的 Map，调用者不能向这个 Map 中插入键值对，或者从该 Map 中删除键值对，但是可以改变值对象的状态。同样，调用者也可以改变 getLocation 方法返回的 SafePoint 对象的状态。

PublishingVehicleTracker 类是线程安全的。但是如果为 PublishingVehicleTracker 类加上约束，否决某些车辆位置改变或者当车辆位置改变的时候执行一些额外的操作的话，上面的实现方式就不行了，因为调用者可以任意修改返回的 SafePoint 对象。

4.4. 向线程安全类添加功能

编程的时候往往会发现 Java 类库中已经存在了一些能满足我们大多数需求的线程安全类，但还要添加一些功能，并维持其线程安全性。例如我们需要一个线程安全 List，让其拥有一个原子的操作，该操作只有在列表中不含有指定的对象的时候才向其添加该对象。Java 类库中的 Synchronized List 实现能够满

足大部分需求，它们提供了 contains 和 add 等方法，我们只需要添加一个 put-if-absent 方法就可以了。

最简单的办法是修改现有的 Synchronized List 实现，但是你没有源码，也不允许修改 Java 类库中的类。另一个办法是扩展这个类（假设它是可扩展的），下面的代码中 BetterVector 类扩展了 Java 类库中的 Vector 类，添加了 putIfAbsent 方法。扩展 Vector 类很简单，但是不是所有的类都会向子类暴露足够多的状态域，以支持这种扩展。

此外，扩展法比直接修改源码的方法更脆弱，因为现在你将同步策略放在两个类（父类和子类）中实现，且父类由系统库维护，子类由你维护。如果父类被修改了，使用了不同的同步策略，那么子类就会出问题。

```
@ThreadSafe
public class BetterVector<E> extends Vector<E>
{
    public synchronized boolean putIfAbsent(E x)
    {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

4.4.1. 客户端锁

第三种方法是使用包装类。下面的代码演示了这个方法，不过这个实现是有问题的，不能保证线程安全性。

```
@NotThreadSafe
class BadListHelper <E>
{
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public synchronized boolean putIfAbsent(E x)
    {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
```

```
}
```

PutIfAbsent 方法已经声明为 synchronized 了，为什么 BadListHelper 类不是线程安全的呢？原因是使用的锁不一样，PutIfAbsent 方法使用的是 BadListHelper 类的内部锁，而 list 域使用的锁肯定不是 BadListHelper 类的内部锁。因此，无法保证当一个线程在执行 putIfAbsent 方法的时候另一个线程不会执行 list 中的其它方法。

要解决这个问题，putIfAbsent 方法必须与 list 使用相同的锁。你必须知道 list 使用了什么锁。Vector 类和 Synchronized 包装类的规格说明中提到它们支持将它们的内部锁用作客户端锁。下面的代码演示了修改后的版本。

```
@ThreadSafe
class GoodListHelper <E>
{
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public boolean putIfAbsent(E x)
    {
        synchronized (list)
        {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

这种方式仍然很脆弱，它将类 C 的内部锁用在一个与类 C 关系不大的类中。这种方法破坏了同步策略的封装性。

4.4.2. 组合

组合是解决上述问题的一种比较好的方法。如下代码中的 ImprovedList 类实现了 List 接口，并把各个方法都代理给内部的 List 实例，此外还添加了一个原子的 putIfAbsent 方法。

```
@ThreadSafe
public class ImprovedList<T> implements List<T>
{
    // final保证了使用者的内存可见性
```

```
private final List<T> list;

/**
 * 构造函数，传入的list对象可以是线程安全的，也可以不是
 */
public ImprovedList(List<T> list)
{
    this.list = list;
}

public synchronized boolean putIfAbsent(T x)
{
    boolean contains = list.contains(x);
    if (contains)
        list.add(x);
    return !contains;
}

// Plain vanilla delegation for List methods.
// Mutative methods must be synchronized to ensure atomicity of putIfAbsent.
public int size()
{
    return list.size();
}

public boolean isEmpty()
{
    return list.isEmpty();
}

public boolean contains(Object o)
{
    return list.contains(o);
}

public Iterator<T> iterator()
{
    return list.iterator();
}

public Object[] toArray()
{
    return list.toArray();
}
```

```
public <T> T[] toArray(T[] a)
{
    return list.toArray(a);
}

public synchronized boolean add(T e)
{
    return list.add(e);
}

public synchronized boolean remove(Object o)
{
    return list.remove(o);
}

public boolean containsAll(Collection<?> c)
{
    return list.containsAll(c);
}

public synchronized boolean addAll(Collection<? extends T> c)
{
    return list.addAll(c);
}

public synchronized boolean addAll(int index, Collection<? extends T> c)
{
    return list.addAll(index, c);
}

public synchronized boolean removeAll(Collection<?> c)
{
    return list.removeAll(c);
}

public synchronized boolean retainAll(Collection<?> c)
{
    return list.retainAll(c);
}

public boolean equals(Object o)
{
    return list.equals(o);
```

```
}

public int hashCode()
{
    return list.hashCode();
}

public T get(int index)
{
    return list.get(index);
}

public synchronized T set(int index, T element)
{
    return list.set(index, element);
}

public synchronized void add(int index, T element)
{
    list.add(index, element);
}

public synchronized T remove(int index)
{
    return list.remove(index);
}

public int indexOf(Object o)
{
    return list.indexOf(o);
}

public int lastIndexOf(Object o)
{
    return list.lastIndexOf(o);
}

public ListIterator<T> listIterator()
{
    return list.listIterator();
}

public ListIterator<T> listIterator(int index)
{
```

```
        return list.listIterator(index);
    }

    public List<T> subList(int fromIndex, int toIndex)
    {
        return list.subList(fromIndex, toIndex);
    }

    public synchronized void clear()
    {
        list.clear();
    }
}
```

ImprovedList 类使用其内部锁添加了一层同步机制。它不关心内部的 List 是否是线程安全的，因为它提供了自己的线程安全策略，即使内部的 List 不是线程安全的，或更换了线程安全策略，都不会影响到 ImprovedList 类的线程安全性。虽然这样做对性能有稍微的损失，但是这样设计出的类非常可靠。事实上我们使用了 Java 监视器模式封装了一个 List 对象，并提供了自己的安全策略，只要不将这个 List 对象发表出去，就能保证 ImprovedList 的线程安全性。

4.5. 在文档注释中记录使用的同步策略

文档注释是管理线程安全性的最有力的工具之一。用户在使用类的时候会查看文档注释，以确定其是否是线程安全的。维护者修改代码之前也需要查看文档注释，以理解该类使用了什么样的同步策略。

你需要在文档注释中为类的使用者记录这个类的线程安全性，为维护者记录这个类使用的同步策略。

同步策略是程序的一部分，应该被记录在文档注释中。最好在设计类的时候将同步策略记录下来，不然过几周之后就就会忘掉当初的设计意图。

选择同步策略并不那么容易，你需要搞清楚哪些域用 volatile 修饰，哪些域用锁保护，用哪个锁，哪些对象应该是 immutable 的，哪些对象需要封闭在单线程中，哪些方法应该具有原子性等等。这些细节很重要，你必须将其记录下来，以便日后维护时使用。

你至少应该在文档注释中说明这个类是否是线程安全的。是不是必须先获取某个锁才能使用它？不要让用户猜，明确写下来。如果你创建的类不想支持客户

端锁就明确写出来。如果你使用了锁来保护状态域，你也应该记录下来，维护者需要这些信息。在这方面@GuardedBy 标注用起来比较顺手。

目前的现状是，即使是 Java 平台库中的类都没有很好的记录线程安全性方面的信息。很多官方的 Java 技术规格说明例如 Servlet 和 JDBC 也没有详细记录这方面的信息。

当一个类看上去该是线程安全的，但 Documentation 中没有提到的时候，我们应该将其假定为线程安全的吗？这是个很难回答的问题。

当我们进行这样的猜测的时候我们的直觉往往是错的。例如 java.text.SimpleDateFormat 类不是线程安全的，但是在 JDK1.4 之前 Javadoc 都没有明确说明这一点。当 JDK1.4 发布之后，很多开发者感到很惊讶，他们一直认为这个类是线程安全的，用这些非线程安全的类构建的对象被多线程共享在重载的情况下将会造成错误的计算结果。

一种解决这类问题的办法是，只要文档中没有说这个类是线程安全的，就默认其为非线程安全的。另一方面，如果不假定 HttpSession 类是线程安全的，就没法开发基于 Servlet 的应用程序。因此，你定义的类应该明确说明其是否是线程安全的，不要让你的客户或同事去猜。

4.5.1. 解释模糊的文档注释

很多 Java 技术规格说明都不愿说明它的一些接口是否是线程安全的。例如 ServletContext、HttpSession 或者 DataSource。这些接口一般是被容器提供者或者数据库供应商实现的，你看不到他们的源代码。此外你也不想依赖于某个提供者的特定实现，你想遵循标准规格说明，这样就可以适应不同的具体实现。但是在 JDBC 规格说明中根本就没提到并发或者线程，Servlet 规格说明中也很少提到，那该怎么办呢？

这样一来你只有猜了，一种比较好的猜测方法是，假定你是规格说明的实现者，你会怎么来实现。Servlet 一般是被一个容器管理的线程调用的，Servlet 容器必须使得 HttpSession 或 ServletContext 这样的对象能够为多个 Servlet 提供服务。因此他们应该是线程安全的，这样多线程以及各个 Servlet 才能够访问它们。

既然这些类一般是用在多线程环境中的，我们应该假定它们是线程安全的，

即使规格说明中没有明确说明这一点。那么是否要使用客户端锁来保护 HttpSession 和 ServletContext 对象呢？用哪个锁来保护？很难猜到，不过你可以看看 tutorial 中的例子。这样你就会发现，不需要使用任何客户端锁。

另外，对于那些通过 setAttribute 方法注册到 ServletContext 或 HttpSession 中的对象，它们是被多线程共享的，但是 Servlet 规格说明没有指定如何同步它们。因此这些对象必须是线程安全的或者是 Immutable 的。

对于 JDBC 中的 DataSource 类也可以做类似推断。一个 DataSource 对象表示一个数据库连接池。DataSource 对象一般不太可能用在单线程环境中，很多示例代码中也没有使用任何客户端锁来保护 DataSource 对象，因此尽管规格说明中没有明说 DataSource 是线程安全的，也没有要求实现者提供一个线程安全的实现，但是如果 DataSource 不是线程安全的，将会很荒谬，这样我们就可以推断 DataSource 是线程安全的，DataSource.getConnection 方法调用不需要任何客户端锁的保护。

另一方面我们无法将 JDBC 中的 Connection 推断为线程安全的，因为在绝大多数情况下，它们一旦被从 DataSource 中获取之后，就在单线程中被使用，然后再返还给 DataSource。所以如果一个 Connection 对象的确要被多线程共享的话，必须使用同步机制进行保护（这种情况很少出现）。

第五章 并发构建块

在上一章中我们展示了构建线程安全类的几种技术，包括将线程安全性代理给内部线程安全组件。如果可能的话尽量用这种方式，因为代理是最有效的构建线程安全类的方式。

平台库中包含了很多并发构建块，比如线程安全集合以及各种同步器（用以协调多线程的执行）。本章讲述这些最有用的并发构建块，尤其是那些在 Java5.0 和 Java6 中引入的。此外也讲解了使用这些并发构建块构建并发程序的模式。

5.1. 线程安全集合

线程安全集合类包括最原始的 JDK 中引入的 Vector 和 HashTable 类，以及在 JDK1.2 中新添加的一些线程安全集合包装类（这些线程安全集合包装类可通过 Collections.synchronizedXxx 方法调用）。这些线程安全集合类是通过封装它们的状态域，并用 synchronized 修饰每一个 public 方法来实现同步的。

5.1.1. 线程安全集合的问题

使用线程安全集合时经常遇到的问题是，你需要为其添加一些额外的复合操作，比如遍历集合、put-if-absent 等，你需要这些复合操作是原子的。例如如下代码：

```
public static Object getLast(Vector list)
{
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list)
{
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

这两个复合操作都先获取列表的长度，然后再获取或删除列表中最后一个元素。在多线程环境中，这样的复合操作必须是原子的。

由于这些线程安全集合支持客户端锁，要使上面两个新加的复合操作具有原子性并不难，可以使用 list 对象的内部锁来保护复合操作，这样就可以保证在 list.size 方法调用和 list.get 方法调用之间，list 不会被其他线程改变。

```
public static Object getLast(Vector list)
{
    synchronized (list)
    {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list)
{
    synchronized (list)
    {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

当我们在遍历 Vector 的时候，其长度可能被其他线程改变。如下代码说明了这一点：

```
for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));
```

这种遍历方式基于一种假设：在 size 调用到 get 调用之间不会有其他线程修改 vector。在单线程环境下这种假设总是成立的，但在多线程环境下就不一定了。如果在 size 调用到 get 调用之间其他线程修改了 vector，当本线程调用 get 的时候就有可能抛出 ArrayIndexOutOfBoundsException。

使用客户端锁可以解决这个问题，代码如下：

```
synchronized (vector)
{
    for(int i = 0; i < vector.size(); i++)
        doSomething(vector.get(i));
}
```

由于遍历过程被客户端锁保护，在遍历期间其他线程就不能修改 vector 了。不过这样做也降低了程序的并发程度。

为了简洁起见，我们在例子中一直用 Vector，其实这个类已经比较陈旧了，在新代码中尽量不要使用它。其他线程安全集合类也同样面临着添加原子复合操作的问题。

在 Java 5.0 中遍历集合的标准方法是使用迭代器或使用 for-each 循环（隐

式迭代器）。如果在遍历期间迭代器发觉集合被其他线程修改，将会抛出ConcurrentModificationException。其具体实现方式是将一个修改次数绑定到集合上，如果在迭代期间修改次数被改变，则 hasNext 和 next 方法将会抛出ConcurrentModificationException。但是由于检查修改次数操作没有使用同步，因此这种方法不是很可靠，可能会失效。

如下代码演示了一个 for-each 循环，Java 编译器会将其转换为迭代器遍历模式。因此运行过程中可能抛出 ConcurrentModificationException。为了消除这种可能性，你可以使用客户端锁来保护这个遍历过程。

```
List<Widget> widgetList
    = Collections.synchronizedList(new ArrayList<Widget>());
...
// May throw ConcurrentModificationException
for (Widget w : widgetList)
    doSomething(w);
```

在遍历集合过程中使用客户端锁将其保护起来也可能造成问题，如果集合比较大，遍历比较耗时，其他需要访问该集合的线程就必须长期等待。此外，如果使用外部锁的话，上例中的 doSomething 就必须在获取这个锁的情况下才能执行，这就容易造成死锁。

一个比较好的替代方法是，克隆这个集合，然后遍历副本。由于这个副本是线程封闭的，其他线程无法在遍历的时候访问它。这样就消除了抛出ConcurrentModificationException 的可能（克隆过程必须被客户端锁保护）。克隆过程也可能比较耗时，到底采用哪种方法，要看具体情况。

5.1.3. 隐藏的迭代器

虽然使用锁可以防止迭代的过程抛出 ConcurrentModificationException，但是每次迭代共享集合的时候都要使用锁，这比较麻烦。此外，很多时候迭代的过程是隐式的，例如 HiddenIterator 类中，虽然没有显式的迭代器，但是addTenThings 方法中隐含了对 set 域的迭代操作，这是因为标准集合的 toString 方法隐含了迭代操作。

```
public class HiddenIterator
{
    @GuardedBy("this")
    private final Set<Integer> set = new HashSet<Integer>();
```

```
    public synchronized void add(Integer i)
    {
        set.add(i);
    }

    public synchronized void remove(Integer i)
    {
        set.remove(i);
    }

    public void addTenThings()
    {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```

造成这种问题的原因是 HiddenIterator 类不是线程安全的，add 和 remove 方法是被内部锁保护的，但是 set.toString 方法没有被 HiddenIterator 类的内部锁保护，这就可能导致某个线程在打印 set 的时候，其他线程正在调用 remove 方法。解决办法就是用内部锁来保护 System.out.println 语句。

如果本例中 set 用的是线程安全集合的话就不会有问题。

hashCode 和 equals 方法都隐式地使用遍历，此外 containsAll、removeAll 和 retainAll 方法，以及以集合作为参数的构造函数，它们都隐含了集合遍历操作。它们都可能抛出 ConcurrentModificationException，使用的时候要小心。

5.2. 并发集合

Java 5.0 提供了几个并发集合类。Java 1.2 中引入的线程安全集合包装类通过封装和内部锁的方式达到线程安全性，并发程度较低。因为多线程需要竞争集合的内部锁，影响了吞吐量。

并发集合类是专门设计用来被多线程共享的。Java 5.0 中引入了 ConcurrentHashMap 类、CopyOnWriteArrayList 类和 ConcurrentMap 接口。其中 ConcurrentMap 接口在 Map 接口的基础上添加了 put-if-absent、replace 和条件删除操作。

用本节介绍的并发集合替换线程安全集合包装类可以极大地提升程序的伸缩性。

Java 5.0 中还引入了两个新的集合类型:Queue 接口和 BlockingQueue 接口。Queue 用于表示队列，有几个不同的实现，包括 ConcurrentLinkedQueue 类和 PriorityQueue 类。ConcurrentLinkedQueue 是线程安全的，PriorityQueue 不是线程安全的。Queue 操作不是阻塞的，如果队列为空则出队操作将返回 null。

BlockingQueue 接口扩展自 Queue 接口，添加了阻塞式入队和出队操作。如果队列为空，则出队操作会阻塞，直到队列不为空。如果队列满了，则入队操作会阻塞，直到队列中有空闲空间。阻塞式队列在生产者消费者模型中很有用，5.3 节会详述。

Java 6 添加了 ConcurrentSkipListMap 和 ConcurrentSkipListSet 两个类，分别用来代替 SortedMap 接口和 SortedSet 接口对应的线程安全集合包装类，例如 ConcurrentSkipListMap 类可代替 Collections.synchronizedMap(new TreeMap<String, String>())。

5.2.1. 使用 ConcurrentHashMap

线程安全集合包装类是通过封装和锁来实现的，每个 public 操作都被内部锁保护，比如 HashMap.get 操作中需要调用 Key 的 equals 方法，如果 Key 类的 hashCode 函数实现的不好，HashMap 内部的数据结构可能退化成一个列表，导致需要逐个调用各个 Key 对象的 equals 方法，这样 HashMap.get 方法就比较耗时，由于其同时又占有了内部锁，因此大大降低了并发性。

虽然 ConcurrentHashMap 类也是使用哈希表实现的，但是它使用了不同的同步策略，因此可以提供更大的并发度和可伸缩性。它没有用一个统一的锁来保护所有的 public 方法，而是使用一个精细的分离锁（11.4.3 节会详述）来达到更大程度的共享。任意数量的读线程和有限数量的写线程可以并发地访问共享 ConcurrentHashMap 对象。

ConcurrentHashMap 以及其他一些并发集合进一步增强了线程安全的集合包装类的功能，它们不会抛出 ConcurrentModificationException，因此在遍历的时候就不需要使用客户端锁了。ConcurrentHashMap 使用的迭代器是弱一致的，迭代过程中可以反映其他线程对集合的修改（但并不保证这一点）。

尽管 ConcurrentHashMap 做了很大的改进，但为了增强并发性，还是有所牺牲的，例如 size 方法返回的是集合的近似大小，而不是精确大小。isEmpty 方法也不是很准。虽然这看上去令人无法接受，但是在并发环境中这两个方法用得不是很多，所以为了提升其他重要操作（例如 get、put、containsKey 和 remove）的性能，牺牲了它们。

另一个需要注意的地方是，线程安全集合包装类支持客户端锁，而 ConcurrentHashMap 类不支持。你不能在客户端代码中使用 ConcurrentHashMap 对象作为客户端锁。

由于 ConcurrentHashMap 有很多的优点，缺点很少，因此你应该尽量用 ConcurrentHashMap 代替线程安全集合包装类。

5.2.2. 额外的原子集合操作

由于 ConcurrentHashMap 类不支持客户端锁，因此，你不能使用客户端锁来为其添加额外的原子操作，例如 put-if-absent 操作。事实上它已经提供了很多 Map 接口中没有的原子操作，例如 put-if-absent、remove-if-equal 和 replace-if-equal。这些操作都是在 ConcurrentMap 接口中定义的，代码如下：

```
public interface ConcurrentMap<K, V> extends Map<K, V>
{
    /**
     * 等价于
     * if (!map.containsKey(key))
     *         return map.put(key, value);
     *    else
     *         return map.get(key);
     */
     V putIfAbsent(K key, V value);

    /**
     * 等价于
     * if (map.containsKey(key) && map.get(key).equals(value))
     * {
     *      map.remove(key);
     *      return true;
     * }
     * else
     *      return false;
     */
```

```
    boolean remove(Object key, Object value);

    /**
     * 等价于
     * if (map.containsKey(key) && map.get(key).equals(oldValue))
     * {
     *       map.put(key, newValue);
     *       return true;
     * }
     * else
     *      return false;
     */
    boolean replace(K key, V oldValue, V newValue);

    /**
     * if (map.containsKey(key))
     * {
     *      return map.put(key, value);
     * }
     * else
     *      return null;
     */
    V replace(K key, V value);
}
```

5.2.3. 使用 CopyOnWriteArrayList

CopyOnWriteArrayList 可以代替 Collections.synchronizedList 方法返回的线程安全集合包装类。在遍历的 CopyOnWriteArrayList 时候不需要使用客户端锁（类似地 CopyOnWriteArraySet 可代替 Collections.synchronizedSet）。

这两个类基于这样一个事实来实现他们的线程安全性：当一个 Immutable 对象被合理地发表之后，对其访问就不需要使用额外的线程同步措施了。每次集合被修改的时候，他们就创建一个全新的集合对象。他们返回的 iterator 是基于 iterator 被创建的时候集合的快照的，即使其他线程随后修改了集合也不会影响该 iterator。遍历过程也不会抛出 ConcurrentModificationException。

显然，每次集合被修改都创建新的副本，这非常耗时，尤其是当集合很大的时候。这两个类最适合在遍历多、修改少的情况下使用（比如监听者列表）。

5.3. 阻塞式队列与生产者-消费者模式

BlockingQueue 类不仅提供了阻塞式入队出队方法 put 和 take，也提供了非阻塞式入队出队方法 offer 和 poll。

BlockingQueue 类支持生产者-消费者模式，生产者-消费者模式将一组工作放在一个待处理列表上。这种方式简化了开发过程，因为它消除了生产者类和消费者类之间的代码依赖，并简化了工作负荷管理。

如果使用阻塞式队列实现生产者-消费者模式，生产者不断将数据放置到队列中，消费者不断从队列中读取数据。生产者可以对消费者一无所知，也不必知道它自己是否是唯一的生产者，它只需要将数据添加到队列中就行了。同样，消费者也不需要知道有哪些生产者，它不关心数据从哪里来。BlockingQueue 接口简化了生产者-消费者模式的实现，最简单的实现方式是一个线程池配合一个工作队列，后面会详述。

两个人分工洗盘子是我们最熟悉的生产者-消费者模式的例子，一个人洗盘子，然后放到架子上，另一个人从架子上取盘子，然后烘干它。在这个场景中，架子就是阻塞队列，如果架子上没盘子，消费者就必须等，如果架子满了，生产者就必须等。这个例子也可扩展至多生产者和多消费者，他们每人都只和架子打交道。

阻塞式队列简化了生产者-消费者模式的实现，如果生产者产生的工作比较少，消费者就会常常处于空闲状态，比如如果访客（生产者）比较少，服务器（消费者）就比较空闲。不过对于有些情况来说，协调生产者和消费者，让它们都充分忙碌比较重要。

如果我们使用的是有界队列的话，如果生产者生产过快，就会在队列满的时候阻塞，给消费者充分的时间去处理队列中的任务，这样就能防止队列不断膨胀，导致内存耗尽。

阻塞队列也提供了非阻塞式入队方法 offer，如果队列已满则返回一个入队失败标识。这样做可以给你更大的灵活性，以设计一些应对策略，比如负载分流、减少生产者线程数量等。

有界队列是有力的资源管理工具，可用于构建高可信度、健壮的应用程序。

Java 类库中包含了几个 BlockingQueue 接口的实现。LinkedBlockingQueue

类和 ArrayBlockingQueue 类是先入先出队列，并且是线程安全的，此外它们也是有界的，不过 LinkedBlockingQueue 如果使用默认构造函数构建的话，其最大容量将是 Integer.MAX_VALUE。PriorityBlockingQueue 队列是优先权队列，可根据自然顺序（Comparable）或比较器（Comparator）比较元素的优先权。此外，PriorityBlockingQueue 是无界的，非线程安全的，且不允许插入 null 元素。

BlockingQueue 接口的最后一个实现类是 SynchronousQueue 类，不过它并不是一个队列，它维护了一队线程。拿洗盘子的例子做类比，这次没有架子了，生产者洗好盘子后直接将其递给空闲的消费者。虽然这样做看上去很奇怪，但是减少了延迟，此外生产者可以知道哪个消费者接了任务。SynchronousQueue 类的 put 和 take 方法将会阻塞，直到有消费者或生产者空闲。这个类一般适合于用在消费者很多，基本上总会有空闲消费者存在的场合。

5.3.1. 例子：磁盘文件索引

一个可以使用生产者-消费者模式实现的例子是，你想要建立磁盘上的文件索引，以加快文件搜索速度。DiskCrawler 类演示了一个生产者线程搜索文件层次结构，向队列中添加要建立索引的 File 对象。Indexer 类演示了一个消费者线程从队列中取出 File 对象，为其建立索引。

```
// 生产者线程
public class FileCrawler implements Runnable
{
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;
    ...
    public void run()
    {
        try
        {
            crawl(root);
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException
```

```
    {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null)
        {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
        }
    }
}

// 消费者线程
public class Indexer implements Runnable
{
    private final BlockingQueue<File> queue;

    public Indexer(BlockingQueue<File> queue)
    {
        this.queue = queue;
    }

    public void run()
    {
        Try
        {
            while (true)
                indexFile(queue.take());
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

使用生产者-消费者模式将遍历文件系统与建索引两个活动分离开来将会使代码更清晰，具有更高的可读性和可重用性。每个活动都只负责做一件事，阻塞队列处理所有的流程控制工作，因此代码变得即简单又清晰。

生产者-消费者模式还有性能上的优势，因为生产者和消费者可以并发执行。

下面的代码启动了几个文件索引生产者和消费者线程，不过在这个例子中消

费者无法终止，我们将在第七章中介绍解决办法。这里使用的是最简陋的多线程架构，你还可以使用 Executor 任务执行框架来实现文件索引功能。

```
public static void startIndexing(File[] roots)
{
    // 创建共享的阻塞式队列
    BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
    FileFilter filter = new FileFilter()
    {
        public boolean accept(File file)
        {
            return true;
        }
    };

    // 为每个文件树根启动一个生产者线程
    for (File root : roots)
        new Thread(new FileCrawler(queue, filter, root)).start();

    // 启动一定数量的消费者线程（生产者和消费者共享 queue 对象）
    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
```

5.3.2. 串行线程封闭

java.util.concurrent 包中实现的所有阻塞式队列都包含足够的内部同步机制，因此在生产者-消费者模式中可以放心使用。

如果任务是 mutable 对象，必须使用串行线程封闭技术将其从生产者传递给消费者。串行线程封闭是指，任务对象本来被封闭在生产者线程中，但是生产者线程将其传递给了唯一的一个消费者线程，且生产者线程随后不会再访问该任务对象，也就是说任务对象被封闭在了新的线程中了。消费者线程可以自由修改任务对象的状态。

5.3.3. 双头队列与工作窃取模式

Java 6 中又添加了两个集合类型，Deque 接口和 BlockingDeque 接口。它们分别继承自 Queue 和 BlockingQueue 接口。Deque 是一个双头队列，允许从两头插入和删除元素。这两个接口的具体实现类分别是 ArrayDeque 和 LinkedBlockingDeque。

双头队列一般用于实现工作窃取模式。在该模式中每个消费者都自己的双头队列，如果一个消费者的队列空了，它可以窃取其他消费者的队列尾部的任务对象。工作窃取模式一般比生产者-消费者模式并发程度更高，因为消费者们不会竞争一个共享队列。大多数时候它们只访问自己的队列，这减少了资源竞争。当一个消费者队列为空的时候，它从其他消费者队列末尾取任务对象，而不是开头，这进一步减少了资源竞争。

5.4. 阻塞和可中断方法

很多原因都可能引起线程的阻塞，比如等待 I/O 完成、等待获取某个锁、等待另一个线程完成等等。当一个线程阻塞的时候，它往往处于几种阻塞状态中的一种（BLOCKED、WAITING 或者 TIMED_WAITING）。阻塞式操作与一个耗时的非阻塞式操作的区别是，它必须等待一个不由它控制的事件来激活它，比如 I/O 完成、要获取的锁变得可用或者其他线程计算完成等。当这些事件发生，被阻塞的线程就会变为 RUNNABLE 状态，等待被调度。

像 Thread.sleep 方法一样，BlockingQueue 接口中定义的 put 和 take 方法都可能抛出检查性异常 InterruptedException。**如果一个方法可能抛出 InterruptedException，则说明该方法是一个阻塞式方法，如果被中断，它就可能提前恢复为 RUNNABLE 状态。**

Thread 类提供了 interrupt 方法，用于中断一个线程或者查询一个线程是否已经被中断了。每个线程都有一个布尔型属性，标识它的中断状态。

中断是多线程合作的机制。当线程 A 中断线程 B 的时候，线程 A 是请求 B 在到达某个合适的点的时候中断它正在做的事情。中断的最合理的用法是用于取消某个操作，第七章会详述。

有两种处理 InterruptedException 的方式：

- **抛出**：可以直接向调用者抛出或者捕获之后进行一些简单操作之后再抛出。
- **处理**：如果 Runnable 中调用某个可能抛出 InterruptedException 的方法，你必须捕获它，进行处理，代码如下：

```
public class TaskRunnable implements Runnable
{
    BlockingQueue<Task> queue;
```

```
    ...
    public void run()
    {
        try
        {
            processTask(queue.take());
        }
        catch (InterruptedException e)
        {
            // restore interrupted status
            Thread.currentThread().interrupt();
        }
    }
}
```

你不应该做的一件事是捕获 InterruptedException，然后什么都不做，这样的话，下次就不能再中断了。

5.5. 同步器

同步器是一些用其自身状态来调节线程控制流的对象。阻塞式队列可作为同步器，当它处于 full 状态的时候，生产者线程调用 put 方法就会阻塞。其他类型的同步器包括信号量、屏障、锁存器。Java 平台库中提供了很多同步器类，如果还不够用，你可以设计自己的同步器，第十四章会介绍如何自定义同步器。

所有的同步器都有相同的结构，他们封装了状态，该状态决定访问该同步器的线程是否应该继续执行或等待。此外，同步器还提供了方法来操纵它的状态，并且提供方法来等待其进入某个状态。

5.5.1. 锁存器

锁存器是一种同步器，它可以将线程延迟到锁存器达到终止状态的时候再执行。锁存器就像一个大门一样，在锁存器达到终止状态之前，门是关着的，所有线程都不能通过。当锁存器达到终止状态，门就开了，所有线程都能通过。锁存器一旦达到终止状态，其状态就不能改变了，因此门就永远打开了。锁存器可以用来确保当某个（或某些）活动完成的时候，另一个活动才能进行，比如：

- 确保在所需的资源没有初始化完成之前，计算操作不能进行。
- 确保一个服务在它所依赖的那个服务没有启动之前不会被启动。
- 等待活动中各个部件都准备好了再开始某项活动。

CountDownLatch 类是锁存器的一个具体实现，可用于上述三种情况。它允许一个或更多线程等待一组事件。其内部状态为一个数字，被设置为一个正整数，表示要等待的事件的个数。countDown 方法将这个数字减一，表示一个事件发生了。await 方法等待状态变为零，如果所有等待的事件都发生了调用 await 的线程就会继续运行，否则该线程就会阻塞。

```
public class TestHarness
{
    public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException
    {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++)
        {
            Thread t = new Thread()
            {
                @Override
                public void run()
                {
                    try
                    {
                        startGate.await();
                        try
                        {
                            task.run();
                        }
                        finally
                        {
                            endGate.countDown();
                        }
                    }
                    catch (InterruptedException ignored)
                    {
                    }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
```

```
        endGate.await();
        long end = System.nanoTime();
        return end - start;
    }
}
```

上述代码演示了 CountDownLatch 类的用法。timeTasks 方法创建了一些线程，并发地运行一些任务。它使用了两个锁存器，一个起始锁存器和一个结束锁存器。起始锁存器的状态被初始化为 1，结束锁存器的状态被初始化为工作线程的个数。每个工作线程在开始工作前都先等待通过起始锁存器，这就确保了在起始锁存器状态变为 0 之前这些工作线程都会阻塞。每个工作线程完成工作之后都会将结束锁存器状态减一，主线程会等待结束锁存器的状态变为 0，一旦其状态变为 0 就标识了所有的工作线程都完成了任务，因此就可以计算整个方法耗时了。

5.5.2. FutureTask

FutureTask 类实现了 Future 接口，使用 Callable 接口实现。FutureTask 对象可能出于三种状态：等待运行、正在运行、运行完毕。其中运行完毕包括正常运行完毕、取消和异常退出。一旦 FutureTask 对象处于运行完毕状态，它的状态就不可改变了。

Future.get 方法的行为取决于该对象当前的状态，如果处于运行完毕状态则立即返回运行结果，否则将会阻塞，直到该对象转换到运行完毕状态，如果是正常运行完毕则返回运行结果，如果是异常退出，则抛出异常。FutureTask 将执行线程的运算结果传递给调用 get 方法的线程。

FutureTask 一般被用在 Executor 框架中表示异步任务，也可以被用于表示那些在结果被需要之前可提前运行的耗时任务。如下代码中的 Preloader 类使用 FutureTask 来进行一个耗时的操作，该操作的运算结果随后将会被用到。由于提前开始了计算，在真正需要该计算结果的时候，就可以少等待一些时间了。

```
public class Preloader
{
    private final FutureTask<ProductInfo> future =
        new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
            public ProductInfo call() throws DataLoadException
            {
                return loadProductInfo();
```

```
            }
        });

    private final Thread thread = new Thread(future);

    public void start()
    {
        thread.start();
    }

    public ProductInfo get()
            throws DataLoadException, InterruptedException
    {
        try
        {
            return future.get();
        }
        catch (ExecutionException e)
        {
            Throwable cause = e.getCause();
            if (cause instanceof DataLoadException)
                throw (DataLoadException) cause;
            else
                throw LaunderThrowable.launderThrowable(cause);
        }
    }
}
```

上例中有一个 FutureTask 实例，它描述了一个从数据库中加载产品信息的耗时任务。此外，还有一个线程实例，FutureTask 将会在这个线程中运行。你可以在构造函数中或者在 static 块中调用 start 方法，提前执行 FutureTask，随后需要这些产品信息的时候你就可以调用 get 方法，如果已经加载完毕则该 get 方法返回产品信息对象，否则阻塞等待加载完毕，然后返回产品信息对象。

Callable 中的代码可能会抛出异常（包括检查性异常和非检查性异常）和错误。这些错误将会在调用 Future.get 的时候抛出，因此上例 get 方法中要捕获 ExecutionException，你可以使用 e.getCause 方法来获取真正的异常对象。

在上例的异常处理块中先看看异常类型是不是已知的，如果不是再调用 LaunderThrowable.launderThrowable 方法对其分类，LaunderThrowable 类的代码如下：

```
public class LaunderThrowable {
```

```
    public static RuntimeException launderThrowable(Throwable t)
    {
        //是运行时异常
        if (t instanceof RuntimeException)
            return (RuntimeException) t;
        //是Error
        else if (t instanceof Error)
            throw (Error) t;
        //逻辑错误
        else
            throw new IllegalStateException("Not unchecked", t);
    }
}
```

5.5.3. 信号量

计数信号量被用于控制可同时访问一个资源的线程数量。计数信号量可被用于实现资源池或者限制集合的大小。

信号量管理一组虚拟许可，初始许可数量从构造函数传入。线程可从信号量获取许可，运行结束后可向信号量返还许可。如果信号量中没有许可，则调用 acquire 的线程会阻塞，直到信号量中有许可获取。调用 release 方法可将许可返还到信号量中。一个退化的例子是一个初始值为 1 的信号量可实现一个不可重入的互斥锁。

信号量可用于实现资源池，比如数据库连接池。虽然你可以实现一个简单的固定大小的资源池，当池子为空的时候获取操作将会返回 null，但有时候你需要在池子为空的时候让调用获取操作的线程阻塞，当池子中有资源的时候再唤醒这个阻塞的线程。你需要将信号量中许可的数量设置为资源池的大小，并在获取资源之前先从信号量中获取一个许可，在使用完资源后将其还入池子中，再调用 release 方法将许可返还给信号量。

类似地，你可以使用信号量将所有集合都变成阻塞式有界集合。在如下代码中，信号量中的许可数量被初始化为集合的最大容量，add 方法在向集合中添加元素之前需要先从信号量中获取一个许可。一个成功的 remove 方法需要向信号量中返还一个许可。

```
public class BoundedHashSet <T>
{
    private final Set<T> set;
```

```
    private final Semaphore sem;

    public BoundedHashSet(int bound)
    {
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException
    {
       // 从信号量获取许可
        sem.acquire();
        boolean wasAdded = false;
        try
        {
            wasAdded = set.add(o);
            return wasAdded;
        }
        finally
        {
           // 添加失败，直接向信号量返还许可
            if (!wasAdded)
                sem.release();
        }
    }

    public boolean remove(Object o)
    {
        boolean wasRemoved = set.remove(o);

        // 删除成功，向信号量返还许可
        if (wasRemoved)
            sem.release();
        return wasRemoved;
    }
}
```

5.5.4. 屏障

我们已经看到锁存器可以实现同时启动一组相关线程或等待一组相关线程结束的功能（见 5.5.1 节）。锁存器是一次性的，一旦它转换到最终状态其状态就不能变化了。

屏障与锁存器类似，它阻塞一组线程，直到某个事件发生。

CyclicBarrier 类允许一组固定数量的线程在一个屏障点会和，然后开启屏障，然后再到这个屏障点会和，开启屏障……。这个类适合于实现并行迭代算法。每个线程到达屏障点的时候调用 await 方法，然后阻塞，当最后一个线程到达屏障点之后，屏障打开，所有线程开始变为 RUNNABLE 状态，等待调度，随后这个屏障对象会被自动重置，以便重复使用。

如果某个线程调用 await 超时或者在阻塞的过程中被中断，那么这个屏障就失效了，其他线程再调用 await 就会抛出 BrokenBarrierException。

CyclicBarrier 类也允许传递一个 Runnable 给构造函数，当屏障打开的时候这个 Runnable 会在所有被阻塞的线程唤醒之前被自动执行，其执行完毕之后再唤醒所有被该屏障阻塞的线程。

屏障常常用于实现并行仿真算法，例如下例中的 CellularAutomata 类。在并行仿真算法中，为每个元素（例如这里的元胞）都开一个线程是不现实的，因为这样的话，需要的线程数量太大。一个合理的解决方案是将问题分解为几个子部分，为每个子部分开一个线程，然后把各个线程的计算结果组合起来。

本例中将问题分解为 N 个部分（这里 N 是可用的 CPU 数量）。每个工作线程都计算属于自己的那部分元胞，计算完成之后进入阻塞状态，当所工作线程都完成任务之后，启动一个屏障线程将它们的结果组合起来。在这个屏障线程执行完之后，屏障打开，各个工作线程开始下一轮的计算。

```
public class CellularAutomata
{
    private final Board mainBoard;          //所有待处理的任务
    private final CyclicBarrier barrier;    //屏障
    private final Worker[] workers;         //所有工作线程

    // 构造函数
    public CellularAutomata(Board board)
    {
        this.mainBoard = board;
        int count = Runtime.getRuntime().availableProcessors();
        this.barrier = new CyclicBarrier(count, new Runnable()
        {
            // 屏障线程用于整合各个任务子部分的计算结果
            public void run()
            {
                mainBoard.commitNewValues();
```

```
        }
    });

    // 初始化所有工作线程，传入要处理的任务子部分
    this.workers = new Worker[count];
    for (int i = 0; i < count; i++)
        workers[i] = new Worker(mainBoard.getSubBoard(count, i));
}

//工作线程类
private class Worker implements Runnable
{
    private final Board board; //本工作线程要处理的任务

    public Worker(Board board)
    {
        this.board = board;
    }

    //工作线程迭代处理属于自己的那部分任务
    public void run()
    {
        while (!board.hasConverged())
        {
            for (int x = 0; x < board.getMaxX(); x++)
                for (int y = 0; y < board.getMaxY(); y++)
                    board.setNewValue(x, y, computeValue(x, y));
            try
            {
                //每处理完一步就阻塞，等待屏障打开再进行下一步
                barrier.await();
            }
            catch (InterruptedException ex)
            {
                return;
            }
            catch (BrokenBarrierException ex)
            {
                return;
            }
        }
    }

    private int computeValue(int x, int y)
```

```
        {
            // Compute the new value that goes in (x,y)
            return 0;
        }
    }

    // 启动所有工作线程
    public void start()
    {
        for (int i = 0; i < workers.length; i++)
            new Thread(workers[i]).start();
        mainBoard.waitForConvergence();
    }

    //任务接口，其实现类应该是线程安全的
    interface Board
    {
        int getMaxX();
        int getMaxY();
        int getValue(int x, int y);
        int setNewValue(int x, int y, int value);
        void commitNewValues();
        boolean hasConverged();
        void waitForConvergence();
        Board getSubBoard(int numPartitions, int index);
    }
}
```

Exchanger 类是屏障的另一种具体实现，它支持将任务分解成两个部分，在屏障点处交换两个部分的数据。Exchanger 类适合于非对称活动，例如一个线程向缓冲区中填入数据，另一个线程从缓冲区中取数据，两者需要不时地交换他们的缓冲区。使用 Exchanger 交换对象可保证两个线程中对象的安全发表。

交换的时机可以是当填充者线程的缓冲区满了或者消费者线程的缓冲区空了的时候启动交换。这样做可以尽量减少交换的次数，不过也可能造成对象长时间停留在缓冲区中不被处理。另一种交换方案是当填充者线程的缓冲区满了或者填充者缓冲区半满，但是距离上次交换已经过了一段特定的时间，下面看一个使用 Exchanger 的示例：

```
// 生产者类
public class ExchangerProducer implements Runnable
{
    // 要交换的对象
```

```
    private List<Integer> intList;
    // Exchanger
    private Exchanger<List<Integer>> listChange;

    public ExchangerProducer(List<Integer> list,
            Exchanger<List<Integer>> exchaner)
    {
        this.intList = list;
        this.listChange = exchaner;
    }

    public void run()
    {
        try
        {
            while (!Thread.interrupted())
            {
                //每过1秒产生一个数，缓冲区满了之后尝试交换数据
                if(intList.size()>2)
                {
                    //等待与listChange关联的另一个线程达到交换点然后交换对象
                    intList = listChange.exchange(intList);
                    System.out.println("生产者接收到消费者" + intList);
                    intList.clear();
                }
                else
                {
                    Thread.sleep(1000);
                    intList.add((new Random()).nextInt(100));
                }
            }
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
// 消费者类
public class ExchangerConsumer implements Runnable
{
    // 待交换的对象
    private List<Integer> intList;
    // Exchanger
```

```
    private Exchanger<List<Integer>> listChange;

    public ExchangerConsumer(List<Integer> list,
            Exchanger<List<Integer>> exchaner)
    {
        this.intList = list;
        this.listChange = exchaner;
    }

    public void run()
    {
        try
        {
            while (!Thread.interrupted())
            {
                //每过1秒消耗一个数，缓冲区空了之后尝试交换
                if(intList.isEmpty())
                {
                    //等待与listChange关联的另一个线程达到交换点然后交换对象
                    intList = listChange.exchange(intList);
                    System.out.println("消费者接收到生产者" + intList);
                }
                else
                {
                    intList.remove(intList.size()-1);
                    Thread.sleep(1000);
                }
            }
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
// 测试类
public class ExchangeTest
{
    public static void main(String[] args) throws InterruptedException
    {
        ExecutorService service = Executors.newCachedThreadPool();

        Exchanger<List<Integer>> exchanger = new Exchanger<List<Integer>>();
```

```
        //由于这两个对象是要被交换的，所以尽量使用线程安全类
        List<Integer> producerList = new CopyOnWriteArrayList<Integer>();
        List<Integer> consumerList = new CopyOnWriteArrayList<Integer>();

        service.execute(new ExchangerProducer(producerList, exchanger));
        service.execute(new ExchangerConsumer(consumerList, exchanger));
    }
}
```

上例的一个测试结果如下：

生产者接收到消费者[]
消费者接收到生产者[81, 74, 28]
生产者接收到消费者[]
消费者接收到生产者[67, 80, 26]
……

5.6. 构建一个有效的可伸缩的结果缓存类

几乎每个服务器端应用程序都要使用缓存，重用之前的计算结果可以减少延迟，增加吞吐量，但是需要消耗一些额外的内存空间。

在本节中我们将开发一个有效的可伸缩的结果缓存类，用于存储计算昂贵型操作的结果。最简单的方法是用 HashMap 做缓存，不过它存在诸多问题，我们将会讨论如何修复这些问题。

如下代码中的 Computable<A,V>接口描述了一个以 A 类型作为输入，V 类型作为输出的功能。ExpensiveFunction 类实现了这个接口，它要花很长的时间来完成 compute 方法的计算过程。我们想要创建一个 Computable 包装类，让它能够缓存之前的计算结果。

```
interface Computable <A, V>
{
    V compute(A arg) throws InterruptedException;
}

class ExpensiveFunction
        implements Computable<String, BigInteger>
{
    public BigInteger compute(String arg)
    {
        // after deep thought...
        return new BigInteger(arg);
    }
}
```

```java
public class Memoizer1 <A, V> implements Computable<A, V>
{
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c)
    {
        this.c = c;
    }

    public synchronized V compute(A arg) throws InterruptedException
    {
        V result = cache.get(arg);
        if (result == null)
        {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

上例中使用了 HashMap 来缓存之前的计算结果。compute 方法首先查看缓存中是否有了想要的结果，如果有则直接将其返回，否则进行计算，然后将结果存入缓存，再返回计算结果。

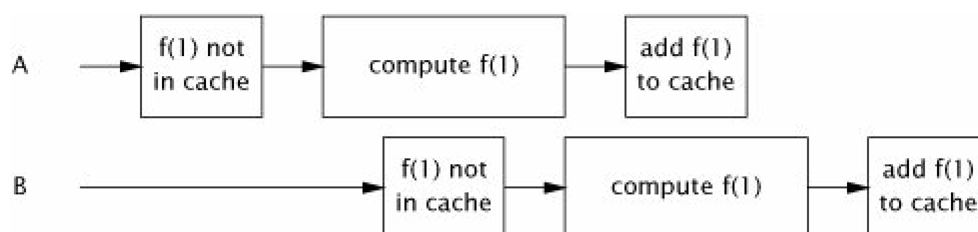
由于 HashMap 不是线程安全的，因此上例中使用内部锁来保护整个 compute 方法。这样做虽然能够确保线程安全性，但是牺牲了伸缩性，一次只允许一个线程执行 compute 方法。如果一个线程执行 compute 方法，没有在缓存中查到想要的结果，因而需要进行耗时操作，这时候其他调用 compute 方法的线程就必须长期等待。

为了解决上例中存在的问题，我们在如下代码中用 ConcurrentHashMap 替换 HashMap，由于 ConcurrentHashMap 是线程安全的，compute 方法就不需要使用锁来保护了。

```java
public class Memoizer2 <A, V> implements Computable<A, V>
{
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;
```

```
    public Memoizer2(Computable<A, V> c)
    {
        this.c = c;
    }

    public V compute(A arg) throws InterruptedException
    {
        V result = cache.get(arg);
        if (result == null)
        {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

上例中 Memoizer2 类的并发性比 Memoizer1 类提升了很多，但仍然有缺陷。如果两个线程同时调用 compute 方法，他们的输入相同，则可能导致相同输入的多次耗时计算，如下图所示：



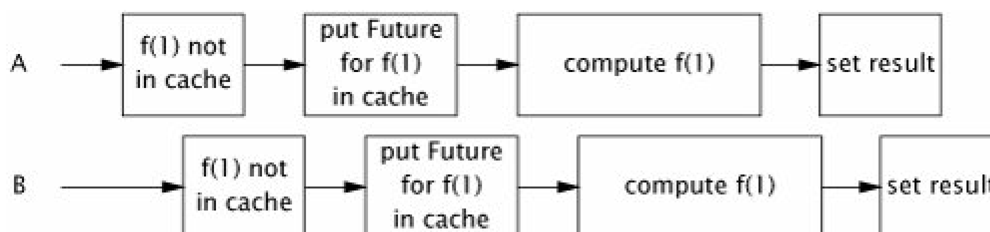
对于这个问题我们可以使用 FutureTask 来解决。在 Memoizer3 类中我们将支持 Map 由 ConcurrentHashMap<A,V>改为 ConcurrentHashMap<A,Future<V>>，当有线程调用 compute 方法时，先检查缓存中是否有指定的条目，如果有，则等待其计算结果，如果没有，则创建一个 FutureTask，注册到缓存中并开一个线程启动 FutureTask 中的计算。计算结果可能是立即返回的，也可能是经过耗时计算，延迟了一会再返回的，不过这个过程对 FutureTask.get 方法的调用线程来说是透明的。

```
// 带记忆特性的包装类
public class Memoizer3 <A, V> implements Computable<A, V>
{
    // 缓存
```

```
    private final Map<A, Future<V>> cache
            = new ConcurrentHashMap<A, Future<V>>();

    // 被包装的进行耗时计算的对象
    private final Computable<A, V> c;

    // 构造函数
    public Memoizer3(Computable<A, V> c)
    {
        this.c = c;
    }

    public V compute(final A arg) throws InterruptedException
    {
        // 获取指定输入对应的Future对象
        Future<V> f = cache.get(arg);

        // 该输入不在缓存中，注册一个FutureTask到缓存中
        if (f == null)
        {
            Callable<V> eval = new Callable<V>()
            {
                public V call() throws InterruptedException
                {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = ft;
            cache.put(arg, ft);
            ft.run(); // call to c.compute happens here
        }
        try
        {
            return f.get();
        }
        catch (ExecutionException e)
        {
            throw LaunderThrowable.launderThrowable(e.getCause());
        }
    }
}
```

Memoizer3 类的设计几乎完美了，它展现出很好的并发性。如果缓存中有合

适的结果则 compute 方法立即返回，如果其他同样输入的线程正在进行耗时计算则本线程耐心等待其返回结果。不过仍然不能排除两个具有相同输入的线程进行重复耗时计算的可能性，如下图所示：



Memoizer3 的缺陷是由于 cache 域的 put-if-absent 操作的非原子性造成的。如下代码中的 Memoizer 类使用了 ConcurrentHashMap 中的 putIfAbsent 原子方法来改进 Memoizer3：

```
public class Memoizer<A, V> implements Computable<A, V>
{
    private final ConcurrentMap<A, Future<V>> cache =
            new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer(Computable<A, V> c)
    {
        this.c = c;
    }

    public V compute(final A arg) throws InterruptedException
    {
        while (true)
        {
            Future<V> f = cache.get(arg);
            if (f == null)
            {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException
                    {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null)
                {
```

```
                    f = ft;
                    ft.run();
                }
            }
            try
            {
                return f.get();
            }
            catch (CancellationException e)
            {
                cache.remove(arg, f);
            }
            catch (ExecutionException e)
            {
                throw LaunderThrowable.launderThrowable(e.getCause());
            }
        }
    }
}
```

本例中允许计算代理 c 抛出 CancellationException，如 compute 方法中的第一个 catch 块所示，一旦用户决定取消该计算，则需要从缓存中移除对应的键值对。

本例的缓存没有提供过期删除功能，不过你可以设计一个 FutureTask 的子类，让其提供一个过期时间域，这样就可以周期性地遍历 cache 域，删除过期的存储条目。

有了这个高并发性的结果缓存类我们就可以为第二章中讨论的因数分解 Servlet 提供真正的缓存功能了。如下代码中的 Factorizer 类使用了 Memoizer 类来缓存之前计算过的因数分解结果：

```
@ThreadSafe
public class Factorizer extends GenericServlet implements Servlet
{
    // 进行耗时操作的计算对象
    private final Computable<BigInteger, BigInteger[]> c =
            new Computable<BigInteger, BigInteger[]>() {
        public BigInteger[] compute(BigInteger arg)
        {
            return factor(arg);
        }
    };
```

```
    // 缓存，提供整数到一组因数的映射（使用c来代理计算过程）
    private final Computable<BigInteger, BigInteger[]> cache =
            new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req, ServletResponse resp)
    {
        try
        {
            BigInteger i = extractFromRequest(req);
            encodeIntoResponse(resp, cache.compute(i));
        }
        catch (InterruptedException e)
        {
            encodeError(resp, "factorization interrupted");
        }
    }

    void encodeIntoResponse(ServletResponse resp, BigInteger[] factors)
    {
    }

    void encodeError(ServletResponse resp, String errorString)
    {
    }

    BigInteger extractFromRequest(ServletRequest req)
    {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i)
    {
        // Doesn't really factor
        return new BigInteger[] { i };
    }
}
```

第一部分总结

现在我们来总结一下第二章到第五章的内容，将一些主要的概念和规则罗列如下：

- 所有的并发问题最终都归结到如何协调多线程对 mutable 状态域的访问。一个类中的 mutable 状态域越少，就越容易实现线程安全。
- 尽量将类的域声明为 final。
- Immutable 对象总是线程安全的。Immutable 对象极大地简化了并发程序的开发，它们可以被多线程共享，并且不需要使用锁和保护性拷贝。
- 封装能够很好地管理类的复杂性。使用封装和锁可以设计出线程安全类。
- **每个 mutable 状态域都应该被锁保护。**
- 所有属于同一个约束条件的 mutable 状态域都应该被同一个锁保护。
- 复合操作一般都要维持原子性，可用锁来实现。
- 在没有同步机制的情况下，多线程共享 mutable 变量将设计出有 Bug 的程序。
- 在文档说明中明确记录你编写的类是否是线程安全的。
- 在文档说明中明确记录你使用的同步策略。

第六章 任务执行框架

大多数并发程序是围绕任务组织的，将应用程序的工作划分成任务简化了程序的组织，通过提供自然的事务边界来支持错误恢复并通过提供自然的并行结构来提高程序的并发性。

6.1. 在线程中执行任务

按照任务来组织程序的第一步是识别任务的边界。理想情况下，任务是一个独立的活动，不依赖于其他任务的状态、结果和副作用。独立性有利于提高并发性，因为如果计算资源充足的话，独立的任务可以被并行执行。为了提高调度和负载均衡的灵活性，每个任务应该表示应用程序中一小部分的工作。

服务器应用程序在正常负荷下应该表现出高吞吐量和高响应速度。应用程序提供者希望它能够支持尽量多的用户，用户希望系统的响应速度尽可能快。此外在重负荷情况下，应用程序应该能优雅地处理这种情况，而不是直接崩溃。选择合理的任务边界和任务执行策略有助于达到这个目的。

大多数服务器应用程序提供一个自然的任务边界：单用户请求。Web 应用服务器、邮件服务器、文件服务器、EJB 容器以及数据库服务器都通过网络连接接受远端客户的访问。一般来说，将每个客户端访问作为一个单独的任务，既提供了独立性，也使得各个任务具有合适的大小。例如向邮件服务器提交的一个消息不会受其他正在处理的消息的影响，此外处理一个消息提交请求只需要很少的计算资源。

6.1.1. 序列化执行任务

有很多种任务调度策略，最简单的是在单线程中顺序地执行任务。如下代码中的 SingleThreadWebServer 类顺序地处理从 80 端口到来的 Http 请求。请求的处理细节不重要，我们关心的是各种任务执行策略的并发性特征。

```
public class SingleThreadWebServer
{
    public static void main(String[] args) throws IOException
    {
        ServerSocket socket = new ServerSocket(80);
        while (true)
        {
            Socket connection = socket.accept();
```

```
            handleRequest(connection);
        }
    }

    private static void handleRequest(Socket connection)
    {
        // request-handling logic here
    }
}
```

SingleThreadWebServer 类很简单，理论上来说也是正确的，但是不太实用，因为它一次只能处理一个请求。主线程既等待用户请求，又处理相应的请求，在处理请求的时候如果有客户端请求到达，就必须等待。如果处理请求阶段没多少工作要做，这种方式效率可能会高一点，但总的来说，这种任务执行策略不适合用于实现商业服务器。

处理 Web 请求涉及到计算和 I/O。服务器必须通过 Socket 操作来获取请求和返回响应。这些 Socket 操作可能会由于网络拥塞和连接问题而阻塞。服务器也可能需要读取本地文件或和数据库交互，这些操作也可能会阻塞。在单线程程序中阻塞不仅延迟了当前正在处理的请求的完成，也使得正在排队的其他请求无法被处理。如果一个请求阻塞了很长时间，所有客户都会以为服务器停用了，因为发出去的请求长时间得不到回应。同时，资源利用率也很低，因为在单线程等待 I/O 的时候，服务器处于空闲状态。

在服务器应用程序中，线性化处理方式很难提供高吞吐量和高响应速度。

6.1.2. 为每个任务显式地创建线程

一个更好的任务处理方式是为每个任务创建一个新的线程，如下例所示：

```
public class ThreadPerTaskWebServer
{
    public static void main(String[] args) throws IOException
    {
        ServerSocket socket = new ServerSocket(80);
        while (true)
        {
            final Socket connection = socket.accept();
            Runnable task = new Runnable()
            {
                public void run()
                {
```

```
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }

    private static void handleRequest(Socket connection)
    {
        // request-handling logic here
    }
}
```

ThreadPerTaskWebServer 的结构与单线程版本的 SingleThreadWebServer 类类似。主线程不停地接收请求，并分派请求。区别是，对于每个请求，主线程创建一个新的线程来处理这个请求，而不是在主线程中处理这个请求。这将导致三个结果：

- 任务处理工作不在主线程中进行，这样主线程就能尽快准备好接收下一个请求。在上一个请求没处理完成的情况下就可以接收下一个请求，从而提高响应速度。
- 任务可以被并行处理，服务器可以同时服务多个请求。如果有多个处理器的话或者有些任务需要阻塞的话这种方式可大大提高吞吐量。
- 任务处理代码必须是线程安全的，因为它可被多任务并发调用。

在轻载或者中等负荷情况下这种每个请求一个线程的任务调度方式相比于线性化任务调度方式是一个很大的改进。只要请求到达速度不超过服务器的处理能力，这种方式提供了更好的响应速度和吞吐量。

6.1.3. 无界线程创建的缺点

在产品化代码中，上例所示的为每个请求开一个线程的方式是不现实的，尤其是可能创建大量线程的场合。下面看看这种任务执行方式有哪些缺点：

线程生命周期开销。线程的创建和销毁都需要消耗大量的系统资源。大多数服务器应用程序处理的都是大量的小规模请求，为这样的每个请求都开一个线程，将会浪费很多计算资源。

资源消耗。活动线程需要消耗系统资源，尤其是内存。当线程数量超过可用的处理器数量的时候，部分线程就会空闲。系统中如果拥有大量空闲线程就会占

用大量的内存空间，从而对垃圾回收器造成压力，甚至导致服务器崩溃。

稳定性。系统中能够存在的线程的数量是有限制的。这个上限在不同平台有不同的取值，同时也受 JVM 调用参数的影响。当线程的数量达到这个上限的时候，很有可能抛出 OutOfMemoryError，要从这个错误中恢复是很困难的，最简单的办法是避免线程数量达到这个上限。

在到达某个拐点之前，开的线程越多，就越能提升吞吐量。但是超过了这个拐点，创建更多的线程将会拖慢程序的运行速度。创建太多的线程有可能造成程序崩溃。最安全的方法是，限制你的应用程序可以创建的线程数量，并详细地测试程序，以确保即使达到了这个上限（但不超过），程序也不会崩溃。

上一节介绍的为每个任务创建一个线程的任务执行方式没有设置线程上限。就像其他并发风险一样，无界线程创建在开发阶段看不出问题，只有在程序部署完成后，在重负荷的情况下才会出问题。因此，如果系统访问量达到一个比较大的范围，一个普通的客户端请求就可能导致系统崩溃。对于那些必须提供高可用性和需要能够有效处理重负荷情况的商业化服务器应用程序来说，这是个严重的问题。

6.2. Executor 框架

任务是工作的逻辑单元，线程是异步执行任务的机制。我们已经讨论了使用线程执行任务的两个策略：单线程策略和为每个任务分配一个单独线程的策略。这两个策略都有很严重的问题，单线程策略响应能力和吞吐量都很差，为每个任务分配一个线程的策略的资源管理能力很差。

在第五章中，我们讨论了如何使用有界队列来阻止超载的应用程序耗尽内存。在线程管理方面，线程池也提供了同样的好处，作为 Executor 框架的一部分的 java.util.concurrent 包提供了一个灵活的线程池实现。在 Java 类库中主要的任务执行抽象不是线程，而是 Executor 接口，其代码如下：

```
public interface Executor
{
    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the <tt>Executor</tt> implementation.
     */
    void execute(Runnable command);
```

```
}
```

虽然 Executor 接口非常的简单，但是它形成了一个复杂、有力的异步任务执行框架的基础，该异步执行框架可支持各种不同的任务执行策略。它提供了将任务发布与任务执行解耦的一个标准方法。Executor 接口的具体实现也提供了生命周期支持和用于统计与监控的钩子。

Executor 基于生产者-消费者模式，提交任务的程序是生产者，执行任务的线程是消费者。

使用 Executor 是实现生产者-消费者模式的最简单方式。

6.2.1. 例子：使用 Executor 实现的 Web 服务器

使用 Executor 接口构建 Web 服务器很简单。如下代码中的 TaskExecutionWebServer 类使用 Executor 来实现。我们使用了 Executor 接口的一个标准实现：一个含有 100 个线程的固定大小线程池。

在 TaskExecutionWebServer 类中，客户端请求产生的任务的提交与执行被 Executor 解耦。通过更换不同的 Executor 接口的实现类就可以轻松修改整个应用程序的行为。Executor 的配置一般来说是一次性事件，而且可以从配置文件中读取参数来对其进行配置。

```java
public class TaskExecutionWebServer
{
    private static final int NTHREADS = 100;
    // 创建一个含有100个线程的固定大小线程池
    private static final Executor exec =
            Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException
    {
        ServerSocket socket = new ServerSocket(80);
        while (true)
        {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run()
                {
                    handleRequest(connection);
                }
            };
            //让线程池中某个线程来执行该任务
```

```
            exec.execute(task);
        }
    }

    private static void handleRequest(Socket connection)
    {
        // request-handling logic here
    }
}
```

我们可以很方便地将上例修改为为每个请求创建一个线程，方法很简单，提供一个 Executor 的具体实现：

```
public class ThreadPerTaskExecutor implements Executor
{
    public void execute(Runnable r)
    {
        new Thread(r).start();
    }
}
```

要想实现一任务一线程的执行策略，只要将 TaskExecutionWebServer 类中的 exec 域的值设置为一个 ThreadPerTaskExecutor 对象。

当然，要实现单线程执行策略，也简单，提供一个 Executor 接口的具体实现，在 execute 方法中不要开新的线程，直接调用 run 方法就好了，代码如下：

```
public class WithinThreadExecutor implements Executor
{
    public void execute(Runnable r)
    {
        r.run();
    };
}
```

6.2.2. 执行策略

任务提交与任务执行解耦的价值在于你可以方便地为一个任务指定任务执行策略，或者在随后更换任务执行策略。任务执行策略指定了如下几个方面：

- 任务在哪个线程中执行？
- 任务以什么样的顺序执行（先入先出、后入先出、优先权顺序）？
- 多少个任务可以被并发执行？
- 多少个任务可以被放在队列中等待执行？

- 如果系统过载，必须拒绝某个任务的执行，拒绝哪个？如何告知应用程序某个任务被拒绝了？
- 在任务执行前后应该做什么？

任务执行策略是资源管理工具，最优策略依赖于可用的计算机资源和服务质量方面的需求。将任务提交和任务执行分开使得我们在应用程序部署后可以方便地根据硬件情况配置具体的任务执行策略。

每次当你看到如下形式的代码的时候，你应该提醒自己，你有可能需要使用更灵活的任务执行策略，仔细考虑是否需要使用 Executor 来代替它。

```
new Thread(runnable).start()
```

6.2.3. 线程池

线程池与一个任务队列紧紧绑定在一起，该任务队列用于存放等待被执行的任务。工作线程的生命周期是：从任务队列中请求下一个要处理的任务，执行该任务，然后请求下一个要处理的任务。

使用线程池执行任务有很多优点。由于重用了线程，而不是每次都新建一个线程，因此摊销了线程创建和销毁的成本。此外，由于在请求到来时，工作线程已经存在了，不需要创建，这就降低了延迟，提高了响应速度。通过微调线程池的大小，你可以使线程池中有足够的线程，以保持处理器足够地忙碌，同时又不会造成内存耗尽。

Java 类库提供了很多方便的带预定义配置的线程池实现。你可以调用 Executors 类的如下几个静态方法之一创建一个线程池。

- newFixedThreadPool。创建一个线程池，重用固定数量的线程，这些线程共享一个无界任务队列。任何时候活动线程数不会超过上限。如果所有线程都处于活动状态，额外的任务被提交后将会在任务队列里等待，直到有线程可用。如果某个线程在执行期间由于抛出异常而终止，且任务队列中又有任务等待处理，将会自动创建一个新的线程。除了被显式的关闭，线程池中的线程一旦创建，就会一直存在于线程池中。
- newCachedThradPool。一个带缓冲的线程池，会尽量使用池中可用的线程。如果池中没有可用的线程，则创建一个新的线程，并添加到池子中去。连续 60 秒没有被使用的线程将被终止，并从池中删除。这个线程

池没有设上限。

- **newSingleThreadExecutor**。一个单线程 Executor，它创建一个单工作线程，来顺序处理各个任务，如果由于抛出异常导致该线程销毁，则会自动创建一个新的工作线程。任务执行顺序由任务队列（先入先出、后入先出、优先权）来指定。
- **newScheduledThreadPool**。一个固定大小的线程池，支持延迟执行和周期性执行，类似于 Timer。

NewFixedThreadPool 和 newCachedThreadPool 两个工厂方法返回了通用目的的 ThreadPoolExecutor 类的实例，这个类可被用来构建更加专业的执行器。我们将在第八章中深入讨论线程池的配置。

TaskExecutionWebServer 类中使用了一个带有界线程池的执行器。通过调用 execute 方法来提交一个任务到任务队列中，各个工作线程，不断从该队列中取出要执行的任务，然后执行它们。

从为每个任务提供一个线程的任务执行策略转换到基于线程池的任务执行策略，对应用程序的稳定性有很大的影响。Web 应用程序将不会在重负载情况下崩溃。它也不会在重负载情况下创建上千个线程来竞争有限的 CPU 和内存资源。此外，使用 Executor 提供了额外的功能，你可以微调、管理、监控、记录、报错，这在没有使用 Executor 的情况下是很难做到的。

6.2.4. Executor 的生命周期

我们已经看到了如何创建一个 Executor，现在来看看如何关闭一个 Executor。一个 Executor 实现往往要创建很多线程，由于 JVM 在所有线程终止之前不能关闭，因此如果不能关闭 Executor，JVM 就无法退出。

由于 Executor 异步处理任务，在任何时间点，之前提交的任务的状态都是未知的，有些任务被处理完成，有些正在被处理，其他的正在等待被执行。对于应用程序的关闭，根据执行情况可以有从优雅地关闭（完成所有任务的处理，并且不再接收新的任务)到直接关闭机器电源,以及介于两者之间所有的关闭方式。

由于 Executor 为应用程序提供服务，它应该可以被关闭，并且将关闭所影响的任务的状态返回给应用程序。

为了表示执行服务的生命周期，ExecutorService 接口扩展自 Executor 接

口，添加了一些生命周期管理方法（也包括一些任务提交方法）。ExecutorService 接口中的生命周期管理方法如下：

```
public interface ExecutorService extends Executor
{
    void shutdown();

    List<Runnable> shutdownNow();

    boolean isShutdown();

    boolean isTerminated();

    boolean awaitTermination(long timeout, TimeUnit unit)
                                        throws InterruptedException;
    ……
}
```

ExecutorService 对象有三种状态：正在运行状态、正在关闭状态、终止状态。刚创建的时候处于正在运行状态。Shutdown 方法启动一个优雅的关闭过程：不再接收新的任务，但是之前接收的任务要执行完。ShutdownNow 方法启动一个立即关闭过程，它试着取消正在执行的任务（通过中断机制实现，任务必须响应中断并将其解释为取消任务），并抛弃任务队列中所有正在等待被执行的任务。

在调用 shutdown 或者 shutdownNow 之后提交给 ExecutorService 的任务将会被拒绝执行处理器处理（见 8.3.3 节），这个任务可能被无声地丢弃，也可能导致 execute 方法抛出非检查性异常 RejectedExecutionException。一旦所有的任务都被完成，ExecutorService 转换到终止状态，你可以调用阻塞式方法 awaitTermination 来等待 ExecutorService 转换到终止状态。此外你也可以调用非阻塞式方法 isTerminated 方法来轮询 ExecutorService 是否达到终止状态。在调用 shutdown 方法之后立即调用 awaitTermination 方法，可造成同步关闭 ExecutorService 的假象（至于 Executor 的关闭和任务的取消将在第七章中详述）。

如下代码中的 LifecycleWebServer 类提供了生命周期支持。它可以被以两种方式关闭：以程序调用方式调用 stop 方法，向服务器发送特定格式的 Http 请求。

```
public class LifecycleWebServer
```

```
{
    private final ExecutorService exec = Executors.newCachedThreadPool();

    public void start() throws IOException
    {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown())
        {
            try
            {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run()
                    {
                        handleRequest(conn);
                    }
                });
            }
            catch (RejectedExecutionException e)
            {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() {
        exec.shutdown();
    }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

6.2.5. 延迟性任务和周期性任务

Timer 类是用于管理任务的延迟执行和周期性执行的，但是这个类有一些缺陷，你应该使用 ScheduledThreadPoolExecutor 类替代它。你可以使用构造函数

来构造 ScheduledThreadPoolExecutor 对象，也可以使用 Executors 类的静态工厂方法 newScheduledThreadPool。

Timer 为要执行的任务创建一个单线程。如果一个任务非常耗时，将影响其他任务的时间精度。如果一个任务每 10 毫秒调度一次，另一个任务只调度一次，但耗时 40 毫秒，这将导致有可能在等待 40 毫秒后，快速地调用第一个任务 4 次或者第一个任务错过 4 次调度。ScheduledThreadPoolExecutor 使用多线程来处理任务，因此可以克服这种问题。

另一个问题是，如果任务抛出非检查性异常，将终止 Timer 的运行。如下代码中的 OutOfTime 类演示了 Timer 的问题。你可能认为程序应该运行 6 秒，然后退出，事实是在运行 1 秒后它抛出 IllegalStateException，异常消息是“Timer 已经被取消”。ScheduledThreadPoolExecutor 类可以很好解决这个问题。在 Java 5.0 及以后版本中，不应再使用 Timer 类。

```
public class OutOfTime
{
    public static void main(String[] args) throws Exception
    {
        Timer timer = new Timer();
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(1);
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(5);
    }

    static class ThrowTask extends TimerTask
    {
        public void run()
        {
            throw new RuntimeException();
        }
    }
}
```

如果你想构建自己的调度服务，你可以使用 DelayQueue，这是一个阻塞式队列实现，为 ScheduledThreadPoolExecutor 提供了支持。

DelayQueue 实现 BlockingQueue 接口，管理了一组实现了 Delayed 接口的对象，每个对象都有一个延迟时间属性，只有在延迟时间到期之后才能从队列中

取出。DelayQueue 队列是有序的，最先到期的元素排在最前，注意不能将 null 元素放置在这种队列中。

Delayed 接口继承自 Comparable 接口。因此其实现者必须提供 getDelay 方法和 compareTo 方法。DelayQueue 使用 getDelay 方法来判断该 Delayed 元素是否过期。compareTo 方法用于实现将最先过期的 Delayed 元素排在队列最前面。

下面是一个使用 DelayQueue 的例子：

```
/**
 * 实现Delayed的类
 */
public class TestDelayed implements Delayed
{
    private String str;           //状态域
    private long   timeout;       //到期时间（纳秒表示）

    /**
     * 构造函数
     * @param str 状态参数
     * @param delayTime 延迟时间（纳秒表示）
     */
    TestDelayed(String str, long delayTime)
    {
        this.str = str;

        //到期时间为当前时间加上延迟时间
        this.timeout = delayTime + System.nanoTime();
    }

    /**
     * 用于根据到期时间对一组TestString对象排序
     */
    @Override
    public int compareTo(Delayed other)
    {
        if (other == this)
            return 0;

        TestDelayed t = (TestDelayed)other;
        long d = (getDelay(TimeUnit.NANOSECONDS) -
                      t.getDelay(TimeUnit.NANOSECONDS));
        return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
    }
```

```
/**
 * 返回到期时间与当前时间的差值，为正则说明未到期，为负说明已到期
 */
@Override
public long getDelay(TimeUnit unit)
{
    return unit.convert(timeout-System.nanoTime(),
                                TimeUnit.NANOSECONDS);
}

public void print()
{
    System.out.print(str);
}

//测试DelayQueue
public static void main(String[] args) throws InterruptedException
{
    ArrayList<String> list = new ArrayList<String>();
    list.add("string 1");
    list.add("string 2");
    list.add("string 3");
    list.add("string 4");
    list.add("string 5");
    list.add("string 6");
    list.add("string 7");
    list.add("string 8");
    list.add("string 9");
    list.add("string 10");

    DelayQueue<TestDelayed> queue = new DelayQueue<TestDelayed>();

    long start = System.currentTimeMillis();

    //入队，每个元素都指定了延迟时间
    for(int i = 0;i<list.size();i++)
    {
        queue.put(new TestDelayed(list.get(i),
                TimeUnit.NANOSECONDS.convert(2000-i*100,
                                TimeUnit.MILLISECONDS)));
    }

    //出队，不断地打印获取的队首元素
```

```
        for(int i = 0;i<list.size();i++)
        {
            try
            {
                //获取队首元素，并打印（如果没有元素过期take方法会阻塞）
                queue.take().print();
                System.out.println(" -- After " +
                    (System.currentTimeMillis()-start) + " MilliSeconds");
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

输出：

```
string 10 -- After 1124 MilliSeconds
string 9 -- After 1218 MilliSeconds
string 8 -- After 1312 MilliSeconds
string 7 -- After 1421 MilliSeconds
string 6 -- After 1515 MilliSeconds
string 5 -- After 1624 MilliSeconds
string 4 -- After 1717 MilliSeconds
string 3 -- After 1827 MilliSeconds
string 2 -- After 1920 MilliSeconds
string 1 -- After 2014 MilliSeconds
```

6.3. 寻找可利用的并行性

Executor 框架使得指定任务执行策略非常方便，但是在使用 Executor 之前，你必须先用 Runnable 来表示你的任务。在大多数服务器应用程序中任务有自然的边界：单客户端请求。但是，在桌面应用程序中任务的边界往往不会这么明显。此外，在服务器应用程序中，单客户端请求仍可能包含可分解的并行性。

在本节中，我们将开发几个并发程度各异的组件。我们的示例组件是浏览器应用程序中的页面渲染部分，将 HTML 页面渲染为图像。为了简单起见，我们假设 HTML 页面只包含标记文本和指定了尺寸与 URL 的图像。

6.3.1. 示例：顺序性页面渲染器

最简单的处理方法是顺序地处理 HTML 文档。遇到标记文本就将其渲染在图

像缓存中，遇到图像就通过网络获取该图像数据，然后绘制在图像缓存中。这种方式很容易实现，并且每个 HTML 元素只处理一遍。不过这可能让用户非常不爽，因为他们必须花很长时间等待页面渲染完成。

该方法的一个变种是，先渲染文本，遇到图像就绘制空白图像来占位。上述工作完成后，再下载各个图像，图像全部下载完成后再将它们绘制到图像缓存中去，替换之前的空白占位图像。代码如下：

```
public class SingleThreadRenderer
{
    void renderPage(CharSequence source)
    {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : scanForImageInfo(source))
            imageData.add(imageInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }
}
```

下载图像的过程一般需要等待 I/O 完成，在此过程中 CPU 比较空闲。因此顺序性渲染方法的 CPU 利用率较低，且渲染过程比较耗时。如果将任务分解为几个独立的任务并发执行，将会大大提高 CPU 利用率和程序的响应速度。

6.3.2. Callable 接口和 Future 接口

Executor 框架使用 Runnable 接口来表示要执行的任务。Runnable 接口有局限性，run 方法没有返回值，也不能抛出检查性异常。

很多任务都属于延迟计算，例如数据库查询、通过网络获取资源或者计算一个复杂的函数。Callable 接口比 Runnable 接口更适合表示这类任务。call 方法可以返回一个值，并且可抛出 Exception。

Runnable 接口和 Callable 接口用于描述抽象的计算任务。任务一般是有限的，有一个明确的起始点，并且最终会执行完成。Executor 所执行的一个任务的生命周期分为四个阶段：创建、提交、启动、完成。由于任务可能比较耗时，我们需要能够取消一个任务的执行。在 Executor 框架中，已提交但是还没有启动的任务总是可以被取消的，已经被启动的任务如果支持中断的话也是可以被取消的，已经完成的任务是无法取消的。第七章将会详述如何取消一个任务。

Future 接口用于表示任务的生命周期，并且提供方法来测试任务是否已完成，或者被取消，获取任务执行结果以及取消任务。任务的生命周期只能前进不能后退，一旦任务完成，就永远处于完成状态。

Callable 接口和 Future 接口定义如下：

```
public interface Callable<V>
{
    /**
     * 计算一个结果或抛出异常
     */
    V call() throws Exception;
}

public interface Future<V>
{
    /**
     * 试图取消一个任务，如果任务已完成、已取消或由于其它原因无法取消，
     * 返回false。如果调用该方法的时候任务尚未启动，该任务将不会被运行
     * 如果任务已经启动，mayInterruptIfRunning参数用于是否要中断任务执
     * 行线程。
     *
     * 如果取消成功，随后调用isDone和isCancelled方法总是返回true。
     *
     * @param mayInterruptIfRunning 如果正在执行该任务的线程应该被中断则指
     * 定为true，如果指定为false则无法终止正在执行的任务。
     * @return 如果成功取消则返回true，否则返回false
     */
    boolean cancel(boolean mayInterruptIfRunning);

    /**
     * 如果在任务完成（正常完成或异常完成）之前被取消则返回true
     */
    boolean isCancelled();

    /**
     * 如果任务完成（正常完成或异常完成）或者被成功取消则返回true
     */
    boolean isDone();

    /**
     * 等待任务执行完成，返回执行结果
     *
     * @throws CancellationException 如果任务被取消则抛出该异常
```

```
     * @throws ExecutionException 如果任务执行期间抛出异常则抛出该异常
     * @throws InterruptedException 如果在阻塞期间调用本方法的线程被中断抛
     * 出该异常
     */
    V get() throws InterruptedException, ExecutionException;

    /**
     * 等待任务执行完成，返回执行结果
     *
     * @param timeout 等待的最长时间
     * @param unit 时间单位
     * @return 返回任务执行结果
     * @throws CancellationException 如果任务被取消则抛出该异常
     * @throws ExecutionException 如果任务执行期间抛出异常则抛出该异常
     * @throws InterruptedException 如果在阻塞期间调用本方法的线程被中断抛
     * 出该异常
     * @throws TimeoutException 如果等待超时则抛出该异常
     */
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

get 方法的行为依赖于任务的状态。如果任务已经正常完成，get 方法将会立即返回任务执行结果，如果任务异常完成，任务抛出的异常对象将会被包装进 ExecutionException 然后从 get 方法调用处抛出。如果任务没有完成，则 get 方法会阻塞，直到任务完成。如果在阻塞期间任务抛出异常，则会将该异常对象包装进 ExecutionException，可以通过 getCause 获取实际的异常对象。如果任务被取消，则 get 方法会抛出 CancellationException。

有几种创建 Future 的方式，ExecutorService 的 submit 方法返回 Future 对象，因此你可以将一个 Runnable 或者 Callable 对象提交给一个执行器，然后获取返回的 Future 对象，用于取回任务执行结果或取消任务的执行。此外，你也可以直接创建一个 FutureTask 对象，将一个 Callable 对象或者 Runnable 对象作为参数传入。你可以将 FutureTask 对象提交给执行器执行，或者直接调用 run 方法在本线程中执行（FutureTask 间接实现了 Runnable 接口）。

ExecutorService 接口的具体实现类可以实现或覆盖 newTaskFor 方法，来指定对应于提交的 Callable 或者 Runnable 对象，返回什么 Future 实现。AbstractExecutorService 类实现了 ExecutorService 接口，它的 newTaskFor

方法实现代码如下：

```
/**
 * 指定对应于提交的Runnable对象，返回什么Future实现，这里直接创建一个
 * FutureTask
 * @param  runnable 要运行的任务
 * @param result 运行成功要返回的结果值对象（这里指定默认的返回结果值对象）
 */
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value)
{
    return new FutureTask<T>(runnable, value);
}

/**
 * 指定对应于提交的Callable对象，返回什么Future实现，这里直接创建一个
 * FutureTask
 * @param  callable 要运行的任务
 */
protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable)
{
    return new FutureTask<T>(callable);
}
```

将 Runnable 对象和 Callable 对象提交给 Executor，实际上是将他们安全地从提交线程发表给执行线程。为 Future 设置默认结果值，实际上是将该结果值对象从提交线程发表给随后调用 get 的线程。

6.3.3. 实例：使用 Future 实现页面渲染器

为了提高 HTML 页面渲染程序的并发性，我们将它分解为两个任务，一个任务渲染标记文本，另一个任务下载图像。第一个任务主要消耗 CPU 资源，第二个任务需要等待 I/O，两个任务并发执行，这样就可以提高系统资源利用率。

Callable 和 Future 接口可以帮助我们表达这些相互合作的任务之间的交互。在如下代码中我们先创建了一个 Callable 负责下载所有的图像，然后将其提交给 ExecutorService。提交函数返回一个 Future，用于获取任务的执行情况。当我们的主任务需要绘制某个图像的时候，它调用该 Future 对象的 get 方法来获取图像加载完成后得到的图像列表。如果幸运的话可以立即得到需要的图像，当然也可能需要等待，但至少提前开始了下载图像的工作。

```
public class FutureRenderer
{
```

```
    private final ExecutorService executor = ...;

    //渲染 HTML 页面
    void renderPage(CharSequence source)
    {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        //创建用于加载图像的任务
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call()
                    {
                        List<ImageData> result=new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };
        //提交加载图像任务，返回 future 对象
        Future<List<ImageData>> future =  executor.submit(task);

        //渲染标记文本
        renderText(source);

        //获取图像列表，逐个渲染图像
        try
        {
            List<ImageData> imageData =  future.get();
            for (ImageData data : imageData)
                renderImage(data);
        }
        catch (ExecutionException e)
        {
            throw launderThrowable(e.getCause());
        }
        catch (InterruptedException e)
        {
            //将当前线程的中断状态标记重置为 true 以通知本方法的调用者当前线
            //程在执行过程中被中断了
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        }
    }
}
```

get 方法的调用者不需要知道任务的执行状态，任务提交和结果获取的安全发布特性使得这样做是线程安全的。包围 Future.get 方法的异常处理代码处理两种可能的问题：call 中的代码抛出异常和 get 方法阻塞期间当前线程被中断。

FutureRenderer 使得在加载图像的同时可以渲染标记文本。在所有图像都加载完成且标记文本渲染完成后再将所有图像逐个绘制到图像缓存中。这是一种改进，用户可以尽快看到部分结果，比 6.3.1 节的顺序性页面渲染器并行性高。但是，这个程序还有很大的改进空间。用户不需要等待所有图像都加载完成，再绘制它们，他们更希望在单个图像加载完成后就立即绘制该图像。

6.3.4. 并行异构任务的局限性

在上例中我们将两个不同类型的任务并行执行：加载图像和渲染页面。但是试图将顺序执行的异构任务改成并行执行以获得较大的性能提升不是那么容易做到的。

对于洗盘子这项任务，如果有两个工人，则很容易分工，一个人负责洗，另一个人负责风干。不过这种为每个工人分配一种类型任务的任务划分方式伸缩性不好。想象一下如果有很多工人参与进来，但是任务类型只有两个，怎么进行劳动分工呢？

为多个工作者划分不同类型任务引起的另一个问题是不同类型的任务其工作量差异很大。如果你将任务 A 和任务 B 分别分配给工作者 1 和工作者 2，但是任务 A 需要消耗的时间是任务 B 的十倍。这样，你仅仅提升了 9%的运行效率。

FutureRenderer 中有两个任务，一个负责渲染标记文本，另一个负责加载图像，一般来说渲染标记文本比加载图像快得多，因此 FutureRenderer 的运行效率比顺序性页面渲染器的执行效率高不了多少。从理论上看，将一个任务分解为两个任务，最多可以将执行效率提高一倍。

根据任务类型进行任务分解对提高并发程度的贡献不大。

将一个程序分解为大量相互独立的，同类型的任务并行执行可极大地提高程序的并发程度。

6.3.5. CompletionService

如果你有一组任务要提交给一个 Executor，并且希望在他们执行完毕之后可以获取执行结果。你可以用 Future 表示各个任务，然后使用参数为 0 的 get

方法不断地询问各个 Future 是否完成。这样做是可能的，但是很枯燥。幸运的是，有一个更好的实现方式：CompletionService。

CompletionService 接口结合了 Executor 和 BlockingQueue 的功能。你可以提交一组 Callable 给它执行，然后使用 take 和 poll 方法来获取任务执行结果。ExecutorCompletionService 类实现了 CompletionService 接口，它将任务执行工作代理给一个 Executor。

ExecutorCompletionService 类的实现非常直接。在构造函数中创建了一个 BlockingQueue 对象来存放各个任务的计算结果。当一个任务被提交给 ExecutorCompletionService 的时候将其包装为 QueueingFuture（FutureTask 的子类），QueueingFuture 类覆盖了 done 方法（该方法在任务转换为 isDone 状态后自动被调用），将计算结果放置在 BlockingQueue 中。take 方法和 poll 方法将它们的工作代理给 BlockingQueue。ExecutorCompletionService 类的部分代码如下：

```java
/**
 * 本类将任务执行工作代理给executor，在各个任务完成之后将对应的Future对象
 * 入队。
 * 用户可以调用take和poll方法获取这些Future对象
 */
public class ExecutorCompletionService<V> implements CompletionService<V>
{
    private final Executor executor;                        //执行器
    private final BlockingQueue<Future<V>> completionQueue;//阻塞式队列
    ......

    /**
     * 扩展自FutureTask的内部类，实现在任务完成后自动将任务入队
     */
    private class QueueingFuture extends FutureTask<Void>
    {
        private final Future<V> task;

        QueueingFuture(RunnableFuture<V> task)
        {
            super(task, null);
            this.task = task;
        }

        protected void done()
```

```
        {
            completionQueue.add(task);
        }
    }

    /**
     * 提交一个任务
     */
    public Future<V> submit(Callable<V> task)
    {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task);
        executor.execute(new QueueingFuture(f));
        return f;
    }

    /**
     * 提交一个任务
     */
    public Future<V> submit(Runnable task, V result)
    {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task, result);
        executor.execute(new QueueingFuture(f));
        return f;
    }

    /**
     * 获取并删除下一个完成的任务对应的Future，如果队列为空则阻塞
     * 如果等待期间调用本方法的线程被中断，则抛出InterruptedException异常
     */
    public Future<V> take() throws InterruptedException
    {
        return completionQueue.take();
    }

    /**
     * 获取并删除下一个完成的任务对应的Future，如果队列为空则返回null
     */
    public Future<V> poll() {
        return completionQueue.poll();
    }

    /**
```

```
     * 获取并删除下一个完成的任务对应的Future，如果队列为空则等待，如果超时
     * 则返回null。
     * 如果等待期间调用本方法的线程被中断，则抛出InterruptedException异常
     */
    public Future<V> poll(long timeout, TimeUnit unit)
                                            throws InterruptedException
    {
        return completionQueue.poll(timeout, unit);
    }
}
```

6.3.6. 实例：使用 CompletionService 实现页面渲染器

使用 CompletionService，我们可以从两个方面提升页面渲染器的性能：更短的渲染时间和更快的响应速度。我们可以为每个图像的下载工作创建一个任务，然后在线程池中执行这些任务。这样缩减了下载图像所需要的时间。通过从 CompletionService 获取任务执行结果，并且在图像加载完成后立即绘制到图像缓存中，我们可以给客户提供一个更加动态和响应速度更快的用户界面，具体实现如下：

```
public class Renderer
{
    //执行器
    private final ExecutorService executor;

    Renderer(ExecutorService executor)
    {
        this.executor = executor;
    }

    //渲染 HTML 页面
    void renderPage(CharSequence source)
    {
        final List<ImageInfo> info = scanForImageInfo(source);
        //创建 CompletionService，传入执行器用于执行任务
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);

        //为每个图像创建一个任务，并提交给 completionService 执行
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                 public ImageData call()
                 {
```

```
                return imageInfo.downloadImage();
            }
        });

        //渲染标记文本
        renderText(source);

        try
        {
            //不断地从队列中取得执行完毕的任务，获取加载的图像，然后绘制
            for (int t = 0, n =  info.size(); t < n;  t++)
            {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
        catch (ExecutionException e)
        {
            throw launderThrowable(e.getCause());
        }
    }
}
```

多个 ExecutorCompletionService 可以共用一个 Executor。当使用这种方式划分任务的时候 Future 用于管理单个任务，CompletionService 用于管理一组任务。只需要记住你向 CompletionService 中提交了多少个任务，以及已经从 CompletionService 中提取了多少个已完成任务你就可以知道还有多少个任务没有完成。

6.3.7. 为任务指定期限

有时候，如果一个任务在指定时间内没有完成，那么该任务就没有了意义，执行结果也就不再需要了。例如一个 Web 应用程序从另一个服务器获取广告信息，如果两秒内没有获取到就会用一个默认的广告信息来代替，这样不可用的广告信息就不会影响该 Web 应用程序的响应速度。类似地，一个门户网站可能同时从多个数据源并行加载数据，但是在呈现页面之前只会等待有限的时间。

Future.get(long timeout, TimeUnit unit)方法支持这种需求，如果任务没有完成该方法会阻塞等待，一旦任务完成，该方法就会返回，如果达到指定的超时时间任务还没有完成该方法就会抛出 TimeoutException 异常。

另一个问题是需要在超时之后取消任务的执行。如果 Future.get(long timeout, TimeUnit unit)方法抛出 TimeoutException 异常，你应该在异常处理块中取消该任务的执行。如下例所示：

```
Page renderPageWithAd() throws InterruptedException
{
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());

    // Render the page while waiting for the ad
    Page page = renderPageBody();
    Ad ad;
    try
    {
        // Only wait for the remaining time budget
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    }
    catch (ExecutionException e)
    {
        ad = DEFAULT_AD;
    }
    catch (TimeoutException e)
    {
        ad = DEFAULT_AD;
        //超时，取消该任务的执行
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}
```

6.3.8. 示例：一个旅游预约门户网站

上例中的预算时间方式可以很容易扩展到多任务情况。考虑一个旅游预约门户网站：用户输入旅游日期、需求，门户获取各种航线、宾馆、租车公司的价格。获取价格需要调用一个 Web 服务或者查询数据库。为了防止页面响应时间被比较慢的远程调用影响，设定一个时间预算，只显示在该时间预算中完成的远程调用

信息是一个好办法。对于那些在时间预算内没有完成的远程调用可以丢弃或者先用占位符占位，等该远程调用完成，再将信息补上去。

本例中获取各个价格的远程调用是相互独立的，因此可作为合理的任务边界，并发执行。你可以创建 n 个任务，提交给一个线程池，获取 n 个 Future，然后使用 Future.get(long timeout, TimeUnit unit)方法获取各个任务处理结果。不过还有更方便的办法，使用 invokeAll 方法，示例代码如下：

```
/**
 * 该类用于表示一个远程查询的任务
 */
private class QuoteTask implements Callable<TravelQuote>
{
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception
    {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes(
        TravelInfo travelInfo, Set<TravelCompany> companies,
        Comparator<TravelQuote> ranking, long time, TimeUnit unit)
        throws InterruptedException
{
    //创建一组任务
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    //调用 ExecutorService 的 invokeAll 方法，提交所有任务，如果所有任务执行
    //完毕或者时间到期该则方法返回。该方法一旦返回，那些没有完成的任务将被
    //取消（是调用 Future.cancel(true)实现的）。执行期间不允许修改 tasks。
    //关于如何取消任务请参阅第七章
    List<Future<TravelQuote>> futures = exec.invokeAll(tasks, time, unit);

    //创建一个列表，用于存储一组远程查询结果
    List<TravelQuote> quotes = new ArrayList<TravelQuote>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();

    //futures 中各个 future 与 tasks 中各个 task 一一对应
```

```
    for (Future<TravelQuote> f : futures)
    {
        QuoteTask task = taskIter.next();
        try
        {
            quotes.add(f.get());
        }
        catch (ExecutionException e)
        {
            //如果远程查询任务执行失败，则用该任务对应的默认失败查询结果
            //替代
            quotes.add(task.getFailureQuote(e.getCause()));
        }
        catch (CancellationException e)
        {
            //如果远程查询任务执行超时，则用该任务对应的默认超时查询结果
            //替代
            quotes.add(task.getTimeoutQuote(e));
        }
    }

    Collections.sort(quotes, ranking);
    return quotes;
}
```

本例使用的 invokeAll 方法带时间期限，用于向 ExecutorService 提交一组任务，并返回一组 Future。返回的 Future 列表的顺序与提交的任务列表顺一一对应。所有任务完成、调用 invokeAll 方法的线程被中断、时间到期都能导致 invokeAll 方法返回。如果时间到期，没有完成的任务将会被取消。一旦 invokeAll 方法返回，所有任务要么执行完毕（正常执行完毕或异常执行完毕）要么被取消，可以用 Future.get 或 Future.isCancelled 方法来判断，如本例所示。

6.4. 本章小结

将应用程序分解为很多可执行的任务可以简化开发过程，提高并行性。Executor 框架允许你将任务提交与任务执行策略解耦，并且支持大量预定义的任务执行策略。每当你需要创建新线程来执行任务的时候，请考虑使用 Executor 框架来代替。为了最大化程序的执行效率，你应该识别任务的合理边界。在有些应用程序中自然的任务边界工作得很好，但是在其它应用程序中，你需要做一些

分析来寻找更细粒度的任务边界划分方案，以提高并行性。

第七章 任务的取消与关闭

启动任务和线程很容易。大多数情况下我们让任务执行线程执行到任务完成为止，但有时候我们需要提前停止任务的执行。原因可能是用户取消了某个操作或者应用程序需要快速关闭。

实现任务和任务执行线程的安全、快速地关闭并不总是那么容易。Java 不提供安全快速关闭任务执行线程的机制，相反，它提供了中断机制，中断使得一个线程可以要求另一个线程停止其正在进行的工作。

中断机制是很有必要的，我们很少要求一个线程立即停止工作，因为那样将会导致一些共享数据结构处于不一致状态。不过，可以将任务设计成在接到停止请求后先做一些收尾工作，然后再终止。

是否能优雅地处理出错、关闭和任务取消是设计良好的应用程序与勉强能运行的应用程序的一个重要区别。本章主要讲解了任务取消和中断机制，以及如何编写任务来响应任务取消请求。

7.1. 任务取消

如果外部代码可以在任务正常完成之前将其变为完成状态的话，这个任务就是可取消的。取消任务的需求可能是由如下几个原因引起的：

- **用户取消了某个操作**。例如用户在图形界面上点击了“取消”按钮。
- **时间上受限的活动**。程序在有限的时间内搜索问题空间，并返回在指定时间内搜索到的最优结果。时间期限到期后，所有未完成的搜索任务必须被取消。
- **应用程序事件**。应用程序分配多个任务来搜索问题空间，当一个任务找到解后，其他任务就不必再继续运行，应该被停止。
- **出错**。例如一组 Web 爬虫，搜索网页，将感兴趣的信息存储在磁盘上。当一个 Web 爬虫发现磁盘已满，出错退出，其他 Web 爬虫应该停止搜索。
- **关闭应用程序**。如果需要优雅地关闭应用程序，则应该将正在运行的任务执行结束，并丢弃那些还没有开始运行的任务。如果是立即关闭应用程序，则应该停止那些正在执行的任务，并丢弃那些还没有开始运行的任务。

Java 中提供了一些线程间合作机制来支持任务取消。一个合作机制是设置一个取消标记，任务周期性地检查这个取消标记，如果发现这个取消标记为 true，则任务提前终止。如下代码中的 PrimeGenerator 展示了这种技术。PrimeGenerator 一直产生素数，直到被取消。cancel 方法用于将 cancelled 标记设置为 true，主循环在搜索下一个素数之前总要查看 cancelled 是否为 true，如果为 true 则终止。

```
@ThreadSafe
public class PrimeGenerator implements Runnable
{
    //执行器
    private static ExecutorService exec = Executors.newCachedThreadPool();

    //用于存储一组素数的列表
    @GuardedBy("this")
    private final List<BigInteger> primes = new ArrayList<BigInteger>();

    //任务取消标记（注意，必须用volatile修饰）
    private volatile boolean cancelled;

    public void run()
    {
        BigInteger p = BigInteger.ONE;
        //每个循环开始都要检查取消标记是否为true，如果为true则停止循环
        while (!cancelled)
        {
            p = p.nextProbablePrime();
            synchronized (this)
            {
                primes.add(p);
            }
        }
    }

    /**
     * 请求取消任务的执行
     */
    public void cancel()
    {
        cancelled = true;
    }
```

```
    /**
     * 获取产生的素数列表
     */
    public synchronized List<BigInteger> get()
    {
        return new ArrayList<BigInteger>(primes);
    }

    /**
     * 静态方法，执行任务1秒钟，产生一组素数
     */
    static List<BigInteger> aSecondOfPrimes() throws InterruptedException
    {
        PrimeGenerator generator = new PrimeGenerator();
        //启动执行器
        exec.execute(generator);

        try
        {
            //主线程睡眠1秒，让任务执行线程运行
            SECONDS.sleep(1);
        }
        finally
        {
            //1秒后请求取消任务的执行（generator可能不会立即停止）
            generator.cancel();
        }

        return generator.get();
    }
}
```

aSecondOfPrimes 方法让任务执行器执行 1 秒钟，然后请求取消其执行。请求取消任务后任务执行线程不会立即终止，因为在调用 generator.cancel 方法到 run 方法中下一次检查取消标记之间还有一段时间。generator.cancel 方法被放置在了 finally 块中，以确保即使 sleep 方法被中断，任务也能够被取消。一个应该取消却没有被取消的任务将会消耗 CPU 和内存资源，并阻止 JVM 退出。

一个支持取消的任务必须有一个**取消策略**。取消策略指定其他代码如何请求取消本任务、本任务什么时候检查它是否被请求取消以及如何响应取消请求。

7.1.1. 中断

上例中使用的取消策略虽然能够通过调用 cancel 方法请求取消任务，并且任务最终能够被取消，但是从请求取消任务到任务执行线程退出需要一段时间。如果 run 方法中的循环里的代码调用了阻塞式方法，例如 BlockingQueue.put，这样可能长期不会检查取消标记，导致任务执行线程长期不能终止。如下代码演示了这种可能性：

```
class BrokenPrimeProducer extends Thread
{
    private final BlockingQueue<BigInteger> queue;
    private volatile boolean cancelled = false;

    BrokenPrimeProducer(BlockingQueue<BigInteger> queue)
    {
        this.queue = queue;
    }

    public void run()
    {
        try
        {
            BigInteger p = BigInteger.ONE;
            while (!cancelled)
                queue.put(p = p.nextProbablePrime());
        }
        catch (InterruptedException consumed)
        {
        }
    }

    public void cancel()
    {
        cancelled = true;
    }
}
......
void consumePrimes() throws InterruptedException
{
    //创建两个线程共享的阻塞式队列
    BlockingQueue<BigInteger> primes = ...;
    //创建并启动生产者线程
    BrokenPrimeProducer producer = new BrokenPrimeProducer(primes);
```

```
    producer.start();

    try
    {
        //在循环中不断消费素数（本线程是消费者线程）
        while (needMorePrimes())
            consume(primes.take());
    }
    finally
    {
        producer.cancel();
    }
}
```

生产者线程不断产生素数，然后放置到阻塞式队列中去。如果生产者线程运行速度比消费者线程快，队列会满，put 方法调用会导致生产者线程阻塞，如果这时候消费者线程请求取消生产者线程，它调用了 producer.cancel()方法，将 producer 的 cancelled 标记设置为 true，但是生产者线程没有机会检查 cancelled 标记，因为它无法从阻塞状态被唤醒。

我们在第五章中提到，有些阻塞式库方法支持中断。中断是一种线程间合作机制，它允许一个线程请求另一个线程在它方便的时候停止正常工作，做一些额外的处理工作。

中断机制一般用于取消任务的执行。

每一个线程都有一个中断状态标记，中断一个线程就将其中断状态标记设置为 true。Thread 类中包含请求中断线程和查询中断状态的方法。interrupt 方法用于中断目标线程，isInterrupted 方法返回目标线程的中断状态。静态方法 interrupted 用于清除当前线程的中断状态（将当前线程的中断状态标记设置为 false），并返回清除前的中断状态。该方法是清除线程中断状态的唯一方法。Thread 类的部分代码如下：

```
class Thread implements Runnable
{
    public void interrupt(){...}
    public boolean isInterrupted() {...}
    public static boolean interrupted() {...}
}
```

阻塞式库方法，比如 Thread.sleep 和 Object.wait 都可以响应中断，他们被中断时都会将它们所在的线程的中断状态标记设置为 true 然后抛出

InterruptedException。

虽然JVM不能保证从请求中断到抛出InterruptedException要花多长时间，但从实际运行效果来看这是非常快的。

如果被中断的线程当前并没有处于阻塞状态，那么 interrupt 方法只将其中断状态标记设置为 true，不会抛出 InterruptedException。该线程需要自己不断轮询其中断状态标记是否被设置为 true 来确定其是否已被中断。

调用一个线程的 interrupt 方法不是立即停止该线程的运行，而是告诉该线程其他线程向其提出了中断请求。

Object.wait、Object.join、Thread.sleep 等方法都是阻塞式方法，如果接收到中断请求，或者刚进入的时候就发现当前线程中断标记为 true，它们会抛出 InterruptedException。

静态方法 Thread.interrupted 用起来要小心，因为它将清除当前线程的中断状态标记。

BrokenPrimeProducer 类演示了用任务取消标记实现的任务取消机制在任务中包含阻塞式方法调用的时候可能无法正常工作的情况。你可以使用中断机制来实现任务取消功能。

中断机制往往是实现任务取消的最合理的方式。

如下代码演示了如何使用中断机制解决 BrokenPrimeProducer 类中存在的问题：

```
class PrimeProducer extends Thread
{
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue)
    {
        this.queue = queue;
    }

    public void run()
    {
        try
        {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
```

```
        }
        catch (InterruptedException consumed)
        {
        }
    }

    public void cancel()
    {
        interrupt();
    }
}
```

每次循环中有两个中断检查点：如果 put 方法在阻塞状态下的时候本线程被中断将会抛出 InterruptedException，如果在其他时机本线程被中断，可通过 isInterrupted 方法检测到。另外在调用 put 方法的时候如果发现本线程的中断状态标记已经为 true 则会抛出 InterruptedException。

虽然这里面!Thread.currentThread().isInterrupted()不是非常必要，但是它提高了中断响应速度。如果不是每次循环都调用 put 方法，那么这种显式测试中断状态标识的方法就很有必要了。

7.1.2. 中断策略

就像任务应该有取消策略一样，线程应该有中断策略。中断策略指定一个线程如何解释中断请求，如果检测到一个中断请求如何响应，哪些操作相对于中断应该是原子的，对中断的响应速度应该有多快。

搞清楚任务和线程如何响应中断非常重要。一个任务在接收到中断请求后可以继续完成其当前的操作，随后再通过测试中断状态标记或者抛出 InterruptedException 的方式响应中断。这种技术可以防止中断破坏已经被修改了一半的数据结构的完整性。

由于每个线程都有自己的中断策略，因此除非你知道中断对目标线程意味着什么，否则不要中断其他线程。

7.1.3. 响应中断

如果你调用一个可中断方法例如 Thread.sleep 或者 BlockingQueue.put 方法，有两种处理 InterruptedException 的方法：

- 如果你的方法调用了这些可能抛出 InterruptedException 的阻塞式方

法，并选择继续向外抛出，那么你的方法也变成了一个可中断的阻塞式方法。

- 如果你的方法调用了这些可能抛出 InterruptedException 的阻塞式方法，并选择提供自己的 InterruptedException 处理块，需要在处理块中将当前线程的中断状态标记重置为 true，这样高层调用者就可以看到中断请求，处理中断。

第一种方法比较简单，你只需向方法声明后面添加 throws InterruptedException 就可以了，例如：

```
BlockingQueue<Task> queue;
...
public Task getNextTask() throws InterruptedException
{
    return queue.take();
}
```

如果你不想继续抛出 InterruptedException，你必须找到其他方法来保留中断请求信息。标准方法是调用 Thread.currentThread().interrupt()方法将当前线程的中断状态标记重置为 true。一般来说你不应该用空异常处理块来处理中断异常，除非在 Runnable 的 run 方法中或者 Callable 的 call 方法中。

7.1.1 节的 PrimeProducer 类中的 run 方法中就提供了一个空的中断异常处理块，这样做的原因是此时线程即将运行完毕，没有其他代码需要知道当前线程是否被中断。

只有实现线程中断策略的代码可以包含空中断异常处理块（如 7.1.1 节讨论的 PrimeProducer 类中的 run 方法），其他代码中不能直接吞掉中断请求。

对于那些调用了可中断阻塞式方法且不支持取消的活动，应该将该活动放置在一个循环中，当该活动被中断后应该重新执行，并且需要记录中断事件，在方法退出前将当前线程的中断状态标记设置为 true，例如：

```
/**
 * 该方法从queue中取得一个元素返回，如果在取元素过程中当前线程被中断，则将
 * 中断信息记录在局部变量中，然后重新试图从queue取出一个元素。
 * 如果局部中断信息为true，在方法返回前将当前线程的中断状态标记设置为true
 */
public Task getNextTask(BlockingQueue<Task> queue)
{
    boolean interrupted = false;  //局部中断标记
```

```
    try
    {
        while (true)
        {
            try
            {
                return queue.take();
            }
            catch (InterruptedException e)
            {
                interrupted = true;
            }
        }
    }
    finally
    {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

注意，一般来说阻塞式方法在阻塞或做任何实际工作之前总是先检查当前线程的中断状态标记，如果为 true，则抛出 InterruptedException 并将当前线程的中断状态标记设置为 false。

如果你的代码没有调用任何可中断的阻塞式方法，这段代码还是可以响应中断请求的，方法是查询当前线程的中断状态标记。

如果用中断实现任务取消，中断只是给任务执行线程提个醒，引起其注意，被引起中断的线程存储在其他位置的信息可以提供进一步指示（注意，存取这些指示信息需要使用同步）。

7.1.4. 示例：限时任务

很多问题可能需要无限运行下去，例如枚举所有素数。也有些问题在有限时间内能找到足够合理的解，但找到最优解需要的运行时间是无穷大。对于这类问题，我们可以给一个限定时间，寻找在这段时间内所能找到的最优解作为整个问题的次优解。

7.1.5. 使用 Future 来取消任务

现在来考虑一个问题，如果任务执行过程中抛出检查性异常怎么办？

ExecutorService.submit 方法返回一个 Future 对象用于管理任务的声明周期。Future.cancel(boolean mayInterruptIfRunning)方法返回值表示是否成功取消了任务的执行。如果任务支持中断、mayInterruptedIfRunning 为 true 并且任务正在某个线程中执行，则这个线程将被中断，cancel 方法返回 true。如果该任务不支持中断，mayInterruptedIfRunning 应该指定为 false，不支持中断的任务如果已经运行则无法被终止。

如果你试图取消一个任务，你不应该直接中断线程池，因为你不知道线程池中的各个线程运行什么任务。你应该通过 Future 来取消任务。调用 Future.cancel(true)方法将导致任务执行线程被中断，在该线程中响应中断，取消任务。

如下代码实现了限时运行，在 timedRun 方法中将任务 r 提交给一个 ExecutorService，获取返回的 Future 对象 task，然后调用限时版本的 Future.get 方法获取任务计算结果（这里 get 方法返回值自然是 null，不过任务的计算结果可以保存在 r 的私有域中）。如果抛出 TimeoutException，在 TimeoutException 处理块中取消任务的运行，如果 r 中的代码在时间到期之前抛出一个异常，get 方法将会抛出 ExecutionException，在 ExecutionException 处理块中将异常重新抛出。

```
//任务执行器
private static final ScheduledExecutorService taskExec =
                        Executors.newScheduledThreadPool(10);

/**
 * 在指定时间内执行任务，任务可能在到期前完成，也可能运行超时，如果
 * 运行超时将会调用Future.cancel(true)方法取消任务，任务执行线程将
 * 会被中断，任务执行线程负责定义自己的中断响应策略以便优雅地结束任
 * 务的执行。
 *
 * 注意由于调用了Future.get方法，本方法可能抛出InterruptedException
 * 因此，本方法是可中断的阻塞式方法。
 *
 * @param r 要执行的任务
 * @param timeout 超时期限
 * @param unit 时间单位
 * @throws Exception 任务运行可能抛出非检查性异常。
 */
public static void timedRun(Runnable r,
```

```
                              long timeout, TimeUnit unit)
                              throws Exception
{
    //将任务提交给执行器执行
    Future<?> task = taskExec.submit(r);
    try
    {
        task.get(timeout, unit);
    }
    catch (TimeoutException e)
    {
        //任务执行超时，取消任务（取消任务的代码放到finally块中了）
    }
    catch (ExecutionException e)
    {
        //获取任务中抛出的异常，重新抛出
        throw new Exception(e.getCause());
    }
    finally
    {
        //如果任务仍在运行，取消任务，如果任务已经完成，本调用没有效果
        task.cancel(true);
    }
}
```

7.1.6. 处理不可中断的阻塞式方法

很多阻塞式库方法通过提前返回并抛出 InterruptedException 来响应中断，这为使用中断来实现任务取消提供了方便。然而，不是所有的阻塞式方法或阻塞机制都支持中断，如果一个线程由于执行同步 I/O 而阻塞或者由于等待获取某个对象的内部锁而阻塞，中断请求不会导致某个 InterruptedException 被抛出，只会将线程的中断状态标记设置为 true。

```
public class ReaderThread extends Thread
{
    private static final int BUFSZ = 512;
    private final Socket socket;          //Socket
    private final InputStream in;         //和Socket对应的输入流

    public ReaderThread(Socket socket) throws IOException
    {
        this.socket = socket;
        this.in = socket.getInputStream();
```

```
    }

    /**
     * 覆盖interrupt方法，实现在提交中断请求之前先关闭Socket
     */
    public void interrupt()
    {
        try
        {
            socket.close();
        }
        catch (IOException ignored)
        {
        }
        finally
        {
            super.interrupt();
        }
    }

    public void run()
    {
        try
        {
            byte[] buf = new byte[BUFSZ];
            while (true)
            {
                int count = in.read(buf);
                if (count < 0)
                    break;
                else if (count > 0)
                    processBuffer(buf, count);
            }
        }
        catch (IOException e)
        {
            /* Allow thread to exit */
        }
    }

    public void processBuffer(byte[] buf, int count)
    {
    }
}
```

上述代码中 run 方法内部的代码不支持中断，因为 in.read 方法虽然是阻塞式的，但不会抛出 InterruptedException。

上例中覆盖了 interrupt 方法，在提交中断请求之前先关闭 Socket，如果在调用 socket.close 的时候，run 方法中的 in.read 方法正在阻塞，read 方法将抛出 SocketException（IOException 的子类）。此外，如果 socket 已经被关闭，调用 in.read 将会抛出 IOException。这样中断一个 ReaderThread 将会导致线程退出。本例中，如果 run 方法内包含可响应中断的阻塞式方法也是可以的，只需加个 InterruptedException 处理块就可以了。

7.1.7. 使用 newTaskFor 实现非标准化的任务取消

在 Java6 中上例 ReaderThread 中使用的技术可以改为使用 newTaskFor 钩子实现。当调用 submit 方法将一个 Callable 提交给 ExecutorService 的时候，将会返回一个 Future 对象用于取回任务执行结果或取消任务。在 submit 方法内部将会调用 newTaskFor 方法返回一个 RunnableFuture 对象来创建这个 Future 对象。RunnableFuture 接口扩展自 Runnable 接口和 Future 接口，FutureTask 类是该接口的实现类。

如果自定义 Future 类，你就可以覆盖 Future.cancel 方法，在取消任务的时候做些日志记录或收集一些统计信息。这种方式可以在任务不支持中断的情况下实现任务取消。

```
/**
 * 本接口扩展自Callable接口，除了提供call方法之外还提供了cancel方法和
 * newTask方法，用于取消该任务和创建新Future封装该任务
 */
public interface CancellableTask<T> extends Callable<T>
{
    void cancel();                    //取消该任务
    RunnableFuture<T> newTask();   //创建新Future封装该任务
}

/**
 * 该类继承自ThreadPoolExecutor类，覆盖了newTaskFor方法
 */
@ThreadSafe
public class CancellingExecutor extends ThreadPoolExecutor
{
    ...
```

```
    Protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable)
    {
        //如果任务是 CancellableTask 类型则将新建 Future 的任务代理给 callable
        if (callable instanceof CancellableTask)
            return ((CancellableTask<T>) callable).newTask();
        //如果任务是不是 CancellableTask 类型则使用父类的 newTaskFor 方法
        else
            return super.newTaskFor(callable);
    }
}

/**
 * 该抽象类实现了CancellableTask接口，实现了cancel和newTask两个方法，
 * 没有实现call方法，该方法留给具体子类实现
 */
public abstract class SocketUsingTask<T> implements CancellableTask<T>
{
    @GuardedBy("this")
    private Socket socket;        //Socket

    protected synchronized void setSocket(Socket s)
    {
        socket = s;
    }

    //实现 cancel 方法，在取消该任务之前先关闭 Socket
    public synchronized void cancel()
    {
        try
        {
            if (socket != null)
                socket.close();
        }
        catch (IOException ignored)
        { }
    }

    //用于创建新的 Future 对象，表示该任务
    public RunnableFuture<T> newTask()
    {
        //返回修改了 cancel 函数功能的 FutureTask 对象
        return new FutureTask<T>(this) {
            public boolean cancel(boolean mayInterruptIfRunning)
            {
```

```
                try
                {
                    SocketUsingTask.this.cancel();
                }
                finally
                {
                    return super.cancel(mayInterruptIfRunning);
                }
            }
        };
    }
}
```

本例中的 CancellableTask 接口扩展自 Callable 并添加了 cancel 方法和 newTask 方法用于创建一个 RunnableFuture。CancellingExecutor 类扩展自 ThreadPoolExecutor 类，覆盖了 newTaskFor 方法，如果传入的任务是 CancellableTask 类的话将创建 Future 对象的任务代理给它。

SocketUsingTask 类实现了 CancellableTask 接口，实现了 cancel 方法用于在取消之前先关闭 Socket。如果向 CancellingExecutor 提交一个 SocketUsingTask 任务，通过返回的 Future 对象来取消这个任务，将会导致先关闭 Socket，然后再中断任务执行线程。

7.2. 关闭一个基于线程的服务

应用程序一般会创建一些服务，每个服务拥有一组线程，比如线程池。这些服务的生命周期往往比创建这些服务的方法的生命周期长得多。如果想要优雅地关闭应用程序，需要先终止这些服务中的线程。

Java 中使用 Thread 对象来表示线程，一般来说，线程对象属于某个服务，服务是工作线程的所有者。应用程序不应该直接关闭这些工作线程，相反，拥有这些工作线程的服务应该提供生命周期方法来关闭这些线程以及其自身。ExecutorService 提供了 shutdown 方法和 shutdownNow 方法，其他拥有线程的服务类也应该提供类似的关闭机制。

7.2.1. 实例：一个日志服务

大多数服务器端应用程序都使用日志功能，比如使用 println 语句。最常见的方式是提供一个 log 方法，将日志消息存入一个队列，由另一个线程来处理这个队列。

如下代码所示是一个简单的日志服务，记录日志的工作被移到一个单独的线程中去。log 方法没有亲自将日志消息写入输出流，它将日志消息入队，由日志写出线程将日志消息写出到输出流中。这是一个多生产者单消费者的例子，如果入队太快将会导致 BlockingQueue 的阻塞。

```
public class LogWriter
{
    private static final int CAPACITY = 1000;

    //用于存储日志信息的队列
    private final BlockingQueue<String> queue;
    private final LoggerThread logger;

    public LogWriter(Writer writer)
    {
        this.queue = new LinkedBlockingQueue<String>(CAPACITY);
        this.logger = new LoggerThread(writer);
    }

    //启动日志写出线程
    public void start()
    {
        logger.start();
    }

    //记录日志，将日志信息入队
    public void log(String msg) throws InterruptedException
    {
        queue.put(msg);
    }

    /**
     * 用于不断地从队列中取日志信息，写到输出流的线程
     */
    private class LoggerThread extends Thread
    {
        private final PrintWriter writer;

        public LoggerThread(Writer writer)
        {
            //将writer包进PrintWriter中，因为PrintWriter是线程安全的
            this.writer = new PrintWriter(writer, true);
        }
```

```
        public void run()
        {
            try
            {
                while (true)
                    writer.println(queue.take());
            }
            catch (InterruptedException ignored)
            {
            }
            finally
            {
                writer.close();
            }
        }
    }
}
```

为了让 LogWriter 这样的服务能有实用价值，我们必须为其添加一个关闭功能。本例中 queue.take 是可中断的阻塞式方法，且 InterruptedException 中断处理块为空，因此中断消费者线程就可将其关闭。

然而，简单地将消费者线程关闭不是一个好的关闭机制。这样一个唐突的中断将会导致队列中剩余的日志信息被丢弃，更重要的是，调用 log 方法由于队列满而被阻塞的线程将一直处于阻塞状态。

另一个关闭 LogWriter 的方式是提供一个关闭请求标记，如果该标记为 true，将不再响应新的入队请求，代码如下：

```
public void log(String msg) throws InterruptedException
{
    if (!shutdownRequested)
        queue.put(msg);
    else
        throw new IllegalStateException("Logger is shut down");
}
```

这种方法也不太实用，因为这里存在竞争条件。有可能 if 条件为 true，但当调用 put 的时候 if 条件变为 false。

为了给 LogWriter 提供可靠的关闭功能，必须修复这个竞争条件。如果用锁当然可以，不过这里 put 方法可能阻塞，因此如果用锁则容易造成死锁。这里提

供了一个比较好的方法：

```
public class LogService
{
    private final BlockingQueue<String> queue;    //用于存放日志消息的队列
    private final LoggerThread loggerThread;      //消费者线程
    private final PrintWriter writer;             //输出流

    @GuardedBy("this")
    private boolean isShutdown;                   //关闭请求标识
    @GuardedBy("this")
    private int reservations;                     //队列长度标识

    public LogService(Writer writer)
    {
        this.queue = new LinkedBlockingQueue<String>();
        this.loggerThread = new LoggerThread();
        this.writer = new PrintWriter(writer);
    }

    /**
     * 启动消费者线程
     */
    public void start()
    {
        loggerThread.start();
    }

    /**
     * 请求关闭消费者线程
     */
    public void stop()
    {
        synchronized (this)
        {
            isShutdown = true;
        }
        loggerThread.interrupt();
    }

    /**
     * 记录日志消息
     * @param msg 日志消息
     * @throws InterruptedException 由于调用了queue.put方法，因此
     * 可能抛出InterruptedException，本方法也是一个可中断的阻塞式方法
```

```
 */
public void log(String msg) throws InterruptedException
{
    synchronized (this)
    {
        if (isShutdown)
            throw new IllegalStateException("Logger is shut down! ");

        //每次有日志消息入队reservations加一
        ++reservations;
    }
    queue.put(msg);
}

//消费者线程，用于不断从队列中取出日志消息，打印到输出流
private class LoggerThread extends Thread
{
    public void run()
    {
        try
        {
            while (true)
            {
                try
                {
                    //如果已请求关闭，并且队列长度为零，退出
                    synchronized (LogService.this)
                    {
                        if (isShutdown && reservations == 0)
                            break;
                    }

                    String msg = queue.take();
                    //每从队列中取走一条日志消息，reservations减一
                    synchronized (LogService.this)
                    {
                        --reservations;
                    }
                    writer.println(msg);
                }
                catch (InterruptedException e)
                {
                  /* 如果take方法被中断，重试 */
                }
```

```
                }
            }
            finally
            {
                //消费者线程退出前先关闭输出流
                writer.close();
            }
        }
    }
}
```

7.2.2. ExecutorService 的 shutdown 方法

在 6.2.4 节我们讨论过，ExecutorService 类提供了 shutdown 方法和 shutdownNow 方法，前者用于优雅地关闭 ExecutorService。后者用于立即关闭。shutdownNow 方法返回一列任务，这些任务是提交后未被执行的任务，提交后执行一半的任务将被中断，由任务自己负责响应中断，退出执行。

这两个关闭方法分别提供了高安全性和高响应速度。shutdownNow 方法关闭任务较快，但也更容易出错，因为任务可能在执行过程中被中断，如果不能合理地响应中断，优雅地终止那些正在执行的任务，将有可能导致数据结构的破坏。shutdown 方法关闭任务比较慢，因为它需要先将已经提交的任务执行完毕。其他涉及到多线程的服务也应该考虑提供两种不同的关闭模式。

简单的应用程序可能会直接维护一个 ExecutorService 对象，在 main 函数中启动和关闭这个 ExecutorService。复杂的应用程序可能会在服务类中封装一个 ExecutorService 对象，这个高层的服务类需要向外提供自己的生命周期方法，例如如下代码中日志服务类将关闭任务代理给内部的 exec。

```
public class LogService
{
    private final ExecutorService exec = new SingleThreadExecutor();
    ...

    public void start()
    {
    }

    //关闭执行器，并等待所有已运行的任务执行完毕，本方法是可中断的阻塞式方法
    public void stop() throws InterruptedException
    {
```

```
        try
        {
            exec.shutdown();
            exec.awaitTermination(TIMEOUT, UNIT);
        }
        finally
        {
            writer.close();
        }
    }

    public void log(String msg)
    {
        try
        {
            exec.execute(new WriteTask(msg));
        }
        catch (RejectedExecutionException ignored)
        {
        }
    }
}
```

7.2.3. 毒药丸

另一个关闭生产者-消费者服务的方法是使用一个毒药丸。毒药丸是用于放入队列中的一种特殊的对象，消费者线程获得这个对象的时候应该停止运行。如果是先入先出队列，可以确保在毒药丸入队之前入队的所有任务都会被处理。在毒药丸入队之后，生产者就不应该再向队列中插入任何对象了。

为了演示如何使用毒药丸关闭生产者-消费者服务，重新考虑第五章讨论过的文件索引服务。生产者线程如下：

```
class CrawlerThread extends Thread
{
    public void run()
    {
        try
        {
            crawl(root);
        }
        catch (InterruptedException e)
        {
            /*  被中断，退出  */
```

```
        }
        finally
        {
            //确保在退出前向队列中插入一个毒药丸
            while (true)
            {
                try
                {
                    queue.put(POISON);
                    break;
                }
                catch (InterruptedException e1)
                {
                    /*  retry  */
                }
            }
        }
    }

    //爬行器，将指定目录下的所有文件入队
    private void crawl(File root) throws InterruptedException
    {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null)
        {
            for (File entry : entries)
            {
                if (entry.isDirectory())
                    crawl(entry);
                else
                    queue.put(entry);
            }
        }
    }
}
```

消费者线程如下：

```
class IndexerThread extends Thread
{
    public void run()
    {
        try
        {
            while (true)
            {
```

```
                File file = queue.take();
                if (file == POISON)
                    break;
                else
                    indexFile(file);
            }
        }
        catch (InterruptedException consumed)
        {
            /*  被中断，退出，事实上本线程不应该被中断  */
        }
    }
}
```

文件索引服务类代码如下：

```
public class IndexingService
{
    //毒药丸对象
    private static final File POISON = new File("");
    //消费者
    private final IndexerThread consumer = new IndexerThread();
    //生产者
    private final CrawlerThread producer = new CrawlerThread();
    //共享队列
    private final BlockingQueue<File> queue;

    private final FileFilter fileFilter;
    private final File root;

    //定义两个内部类
    class CrawlerThread extends Thread { … }
    class IndexerThread extends Thread { … }

    //启动文件索引服务
    public void start()
    {
        producer.start();
        consumer.start();
    }

    //关闭文件索引服务
    public void stop()
    {
        //中断生产者线程，优雅地关闭该文件索引服务
        producer.interrupt();
```

```
    }

    //阻塞等待消费者线程运行结束
    public void awaitTermination() throws InterruptedException
    {
        consumer.join();
    }
}
```

毒药丸模式只有在生产者线程和消费者线程数量已知的情况下才能工作。上例可以扩展到有 N 个生产者线程一个消费者线程，每个生产者线程关闭的时候都会向队列中插入一个毒药丸，消费者线程在获取 N 个毒药丸之后才停止运行。

上例也可以扩展到 N 个生产者线程，M 个消费者线程，每个生产者线程关闭的时候都向队列中插入 M 个毒药丸，每个消费者线程在获取 N 个毒药丸之后停止运行。

毒药丸模式中使用的共享队列必须是无界队列。

7.2.4. 示例：一个一次性的 ExecutionService

如果一个方法需要处理一批任务，直到所有任务完成才返回。使用一个与该方法生命周期绑定的 Executor 可以简化这个方法的设计。当然，也可以使用 invokeAll 和 invokeAny 方法实现。

如下代码中的 test 方法并行执行 10 个耗时的任务，它创建了一个私有的 ExecutorService 对象。提交任务之后试图优雅地关闭这个 ExecutorService，然后等待 ExecutorService 关闭完成。

```
public class TestShutDown
{
    /**
     * 并行执行10个任务然后关闭执行器，本方法是一个可中断的阻塞式方法
     */
    public static void test(long timeout, TimeUnit unit) throws
                                                    InterruptedException
    {
        // 私有ExecutorService
        ExecutorService exec = Executors.newCachedThreadPool();

        try
        {
            for (int i=0;i<10;i++)
            {
```

```
                exec.execute(new Runnable()
                {
                    public void run()
                    {
                        //先延时一段时间，模拟一个耗时的任务
                        while(true)
                        {
                            try
                            {
                                Thread.sleep((new Random()).nextInt(100));
                                break;
                            }
                            catch (InterruptedException e)
                            {
                            }
                        }
                        System.out.println(Thread.currentThread().
                                                    toString());
                    }
                });
            }
        }
        finally
        {
            // 试着优雅地关闭ExecutorService（需要先完成所有已提交的任务）
            exec.shutdown();
            // 阻塞，直到所有任务都完成或者时间到期
            // 时间到期后未完成的任务继续执行
            exec.awaitTermination(timeout, unit);
        }
    }

    public static void main(String[] args) throws InterruptedException
    {
        //最多等待50毫秒
        test(50, TimeUnit.MILLISECONDS);
        System.err.println("Time Out!");
    }
}
```

输出结果：

```
Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-7,5,main]
Thread[pool-1-thread-5,5,main]
Thread[pool-1-thread-1,5,main]
```

```
Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-2,5,main]
Thread[pool-1-thread-6,5,main]
Time Out!
Thread[pool-1-thread-9,5,main]
Thread[pool-1-thread-10,5,main]
Thread[pool-1-thread-8,5,main]
```

可见，超时退出后未完成的任务还会继续执行，直到完成为止。

7.2.5. shutdownNow 方法的局限性

当调用 ExecutorService 的 shutdownNow 方法的时候，它试图取消所有已经开始运行但没有完成的任务（使用 Future.cancel(true)实现），并返回一列已经提交但还没开始运行的任务。

然而，没有一个通用的方法来确定在调用 shutdownNow 的时刻哪些任务已经开始运行但没有完成。唯一的办法是在每个任务中加检查点

如下代码中的 TrackingExecutor 类展示了一种在关闭时能够确定哪些任务正在运行的技术。

```
/**
 * 该类扩展自AbstractExecutorService，实现了在关闭后能够获取那些由于在关闭
 * 时正在执行而被取消（使用Future.cancel(true)实现的）的任务
 */
public class TrackingExecutor extends AbstractExecutorService
{
    //内部封装的ExecutorService，代理
    private final ExecutorService exec;

    //用于存储在关闭时正在执行而被取消的一组任务
    private final Set<Runnable> tasksCancelledAtShutdown = Collections
            .synchronizedSet(new HashSet<Runnable>());

    public TrackingExecutor(ExecutorService exec)
    {
        this.exec = exec;
    }

    public void shutdown()
    {
        exec.shutdown();
    }
```

```
public List<Runnable> shutdownNow() {
    return exec.shutdownNow();
}

public boolean isShutdown()
{
    return exec.isShutdown();
}

public boolean isTerminated()
{
    return exec.isTerminated();
}

public boolean awaitTermination(long timeout, TimeUnit unit)
                                            throws InterruptedException
{
    return exec.awaitTermination(timeout, unit);
}

/**
 * 获取一组由于该执行器被关闭而被取消的任务
 */
public List<Runnable> getCancelledTasks()
{
    if (!exec.isTerminated())
        throw new IllegalStateException("执行器尚未关闭！");
    return new ArrayList<Runnable>(tasksCancelledAtShutdown);
}

/**
 * 实现execute方法，将runnable包装之后发给exec执行，保证任务执行完毕或
 * 被中断后进行检查如果发现已经请求终止且当前线程被中断的话，将该任务
 * 存入tasksCancelledAtShutdown
 */
public void execute(final Runnable runnable)
{
    exec.execute(new Runnable()
    {
        public void run()
        {
            try
            {
                runnable.run();
```

```
                }
                finally
                {
                    if (isShutdown() &&
                                    Thread.currentThread().isInterrupted())
                        tasksCancelledAtShutdown.add(runnable);
                }
            }
        });
    }
}
```

在调用 shutdownNow 之后，等待该执行器终止（由 isTerminated 方法确定），任务执行器终止后可以调用 getCancelledTasks 方法返回由于立即关闭而被取消执行的一组任务。

为了让 TrackingExecutor 能够正常运行，每个任务在被中断之后，应该确保其执行线程的中断状态标记为 true。

如下代码中的 WebCrawler 类演示了 TrackingExecutor 的使用。如果需要关闭网络爬虫，我们往往希望保存其当前状态，以便在下次启动的时候接着执行。CrawlTask 类中提供了一个 getPage 方法，用于返回本任务当前正在处理的网址。当爬虫被关闭，所有已提交但未启动的任务和已运行但未结束的任务对应的网址都会被记录在内部状态域 urlsToCrawl 中，下次再调用 start 方法启动的时候，这些网址将会被重新处理。

```
/**
 * 网络爬虫，实现了优雅的关闭功能
 * 由于关闭而未被执行的任务以及由于关闭而被取消的任务将在下次启动时重新处理
 */
public abstract class WebCrawler
{
    private static final long TIMEOUT = 500;
    private static final TimeUnit UNIT = MILLISECONDS;

    //执行器
    private volatile TrackingExecutor exec;

    //存储一组需要搜索的网址
    @GuardedBy("this")
    private final Set<URL> urlsToCrawl = new HashSet<URL>();
```

```
public WebCrawler(URL startUrl)
{
    urlsToCrawl.add(startUrl);
}

/**
 * 启动网络爬虫
 */
public synchronized void start()
{
    //创建任务执行器
    exec = new TrackingExecutor(Executors.newCachedThreadPool());

    //对于每个网址启动一个搜索任务进行搜索
    for (URL url : urlsToCrawl)
        submitCrawlTask(url);

    //清空待搜索的网址集合
    urlsToCrawl.clear();
}

/**
 * 关闭网络爬虫，该方法是一个可中断的阻塞式方法
 */
public synchronized void stop() throws InterruptedException
{
    try
    {
        //保存已提交但尚未启动的搜索任务
        saveUncrawled(exec.shutdownNow());

       //如果在指定时间内成功取消了已提交但未结束的搜索任务，将这些任务
       //保存起来
        if (exec.awaitTermination(TIMEOUT, UNIT))
            saveUncrawled(exec.getCancelledTasks());
    }
    finally
    {
        exec = null;
    }
}

/**
 * 抽象方法，处理一个网页并返回所有外部链接
```

```
     */
    protected abstract List<URL> processPage(URL url);

    /**
     * 将一组未完成的任务对应的网址存储在urlsToCrawl中
     */
    private void saveUncrawled(List<Runnable> uncrawled)
    {
        for (Runnable task : uncrawled)
            urlsToCrawl.add(((CrawlTask) task).getPage());
    }

    /**
     * 创建一个网页搜索任务，提交给执行器执行
     */
    private void submitCrawlTask(URL u)
    {
        exec.execute(new CrawlTask(u));
    }

    /**
     * 内部类，表示一个搜索任务
     */
    private class CrawlTask implements Runnable
    {
        private final URL url; //当前正在处理的网页地址

        CrawlTask(URL url)
        {
            this.url = url;
        }

        public void run()
        {
            //处理本网页，此外对于每个外部链接再提交一个搜索任务
            for (URL link : processPage(url))
            {
                //如果线程被中断，提前返回
                if (Thread.currentThread().isInterrupted())
                    return;

                //提交一个搜索外部链接link的任务
                submitCrawlTask(link);
            }
```

```
        }

        /**
         * 返回本任务所处理的网页
         */
        public URL getPage()
        {
            return url;
        }
    }
}
```

本例有一个问题，在某个任务的最后一个指令完毕到任务被标记为执行完毕状态之间有个时间差。如果在这段时间内调用 shutdownNow 的话，将会导致该任务被识别为已运行未结束任务（实际为已完成任务）。对于网络爬虫来说，一个网页被搜索两次没问题，对于其他应用背景可能需要特别处理这个问题。

7.3. 线程异常终止

对于一个单线程控制台应用程序来说，一个未处理的异常将导致应用程序终止运行，并打印堆栈信息。并发程序中的线程异常往往表现得比较复杂，可能会将堆栈信息打印在控制台上，但是没人会去看控制台上的信息。此外，当一个线程运行异常，整个程序看上去还是可以继续运行，因此可能不会注意到一个线程的异常。

线程异常终止一般是由未捕获的非检查性异常造成的。

这类异常一般表示编程错误或者其他不可恢复的问题。运行时异常是不需要被捕获的，一般会一直被传递到顶层，在控制台上打印堆栈信息，然后终止其所在的线程的执行。

线程异常终止的后果可能是良性的，也可能是灾难性的，具体取决于线程在整个应用程序中的角色。

几乎每行代码都可能抛出运行时异常（非检查性异常）。此外，你调用的代码可能正常返回也可能抛出其方法签名所声明的检查性异常。

任务处理线程，比如线程池中的工作线程或者 Swing 程序中的事件分派线程一直都在调用未知代码，这些线程应该对这些未知代码是否能正常运行保持高度的警惕。如果事件处理代码抛出空指针异常导致整个事件分派线程终止，这将是

非常糟糕的。因此，这些任务应该被包在 try-catch-finally 块中，以确保即使任务抛出异常也不会导致执行线程异常终止。

ThreadPoolExecutor 和 Swing 中的事件分派线程使用了一些技术来确保一些可能抛出异常的任务不会导致随后的任务无法运行。它们都使用了如下类似的结构：

```
public void run()
{
    Throwable thrown = null;
    try
    {
        while (!isInterrupted())
            runTask(getTaskFromWorkQueue());
    }
    catch (Throwable e)
    {
        thrown = e;
    }
    finally
    {
        //通知框架，某个线程在执行过程中由于任务抛出异常而终止
        threadExited(this, thrown);
    }
}
```

如果你在编写一个多线程服务框架，用于执行各种提交的任务或者在线程中调用不可信的外部代码的话，可以考虑采用这种方式防止不可信的外部代码抛出异常导致工作线程非正常终止。

7.3.1. UncaughtExceptionHandler

Thread 的 API 中提供了 UncaughtExceptionHandler 机制，可以检测出由于抛出异常导致的线程终止。如果有某个线程由于未捕获的异常而终止，JVM 将这个事件报告给该线程的 UncaughtExceptionHandler，如果该对象没有设置，默认的行为就是将堆栈信息打印在 System.err 上。

```
/**
 * Thread类的内部接口
 */
public interface UncaughtExceptionHandler
{
    /**
```

```
     * 本方法中抛出的任何异常都将被虚拟机忽略
     */
    void uncaughtException(Thread t, Throwable e);
}
```

uncaughtException 方法应该做些什么呢？最常见的用法是在该方法的实现中记录出错消息，例如：

```
public class UEHLogger implements Thread.UncaughtExceptionHandler
{
    public void uncaughtException(Thread t, Throwable e)
    {
        Logger logger = Logger.getAnonymousLogger();
        logger.log(Level.SEVERE,"Thread terminated with exception: " +
                                                          t.getName(),e);
    }
}
```

除了在日志中记录出错信息，也可以在 uncaughtException 方法中重启线程或者关闭应用程序。

对于长期运行的应用程序，比如服务器应用程序，最好提供一个自定义的 UncaughtExceptionHandler，在有线程异常退出的情况下至少能够在日志中记录下来。

如何为线程池设置自定义的 UncaughtExceptionHandler 呢？可以为 ThreadPoolExecutor 类的构造函数提供一个 ThreadFactory 对象作为参数。ThreadFactory 是一个接口，其定义如下：

```
public interface ThreadFactory
{
    Thread newThread(Runnable r);
}
```

其中 newThread 方法用于为任务创建具体的线程对象。该接口的最简单实现如下：

```
class SimpleThreadFactory implements ThreadFactory
{
    public Thread newThread(Runnable r)
    {
        return new Thread(r);
    }
}
```

为了给 ThreadPoolExecutor 指定特定的 UncaughtExceptionHandler，可以

提供一个如下 ThreadFactory 实现：

```
public class UEHThreadFactory implements ThreadFactory
{
    private final static Thread.UncaughtExceptionHandler UEHLOGGER
                                                    = new UEHLogger();

    @Override
    public Thread newThread(Runnable r)
    {
        Thread t= new Thread(r);
        //设置线程的UncaughtExceptionHandler
        t.setUncaughtExceptionHandler(UEHLOGGER);

        return t;
    }
}
```

如下代码测试了上述设置 UncaughtExceptionHandler 方法是否可行：

```
/**
 * 任务类
 */
public class Task implements Runnable
{
    /**
     * 该任务将会抛出异常，导致任务执行线程异常退出
     */
    @Override
    public void run()
    {
        if(Math.random()<0.5)
            throw new NullPointerException();
        else
            throw new IllegalArgumentException();
    }
}

/**
 * 测试类
 */
public class Test
{
    public static void main(String[] args)
    {
```

```
        //创建ThreadFactory对象
        ThreadFactory factory=new UEHThreadFactory();

        //创建线程池对象，传入定制的ThreadFactory对象作为参数
        ExecutorService exec=new ThreadPoolExecutor(10,
                100, 100, TimeUnit.MILLISECONDS,
                new LinkedBlockingDeque<Runnable>(), factory);

        //执行三个任务
        for(int i=0;i<3;i++)
        {
            exec.execute(new Task());
        }
    }
}
```

输出结果：

```
Thread Thread-1 is abnormally terminated caused by exception
java.lang.NullPointerException
Thread Thread-2 is abnormally terminated caused by exception
java.lang.IllegalArgumentException
Thread Thread-0 is abnormally terminated caused by exception
java.lang.IllegalArgumentException
```

线程池的标准行为是当有任务抛出异常的时候，将该任务的执行线程终止，并通知线程池以便线程池可以创建新的线程来替换它。如果没有提供 UncaughtExceptionHandler，工作线程就会无声地终止，导致程序出现一些令人费解的现象。本例可以保证在工作线程由于任务抛出异常而终止的时候可以在日志中记录下来。

需要注意的是，如果将本例中的 Test 类中的 exec.execute(new Task()); 改为 exec.submit(new Task());程序的运行将不会抛出异常。因为这样一来，run 方法中抛出的检查性和非检查性异常将在随后调用 Future.get 方法的时候被包装进 ExecutionException 抛出。

7.4. 关闭 JVM

JVM 可以优雅地关闭，也可以唐突地关闭。当最后一个非 daemon 线程终止会后将会触发 JVM 优雅地关闭。调用 System.exit 方法也会触发 JVM 优雅地关闭。调用 Runtime.halt 或者直接关闭相应的操作系统进程将导致 JVM 唐突地关闭。

7.4.1. 关闭钩子

如果是优雅地关闭 JVM，JVM 在关闭之前将会调用一组注册的关闭钩子（关闭钩子是线程）。可以调用 Runtime.addShutdownHook 向系统注册钩子，JVM 不保证各个钩子执行的先后顺序。

在钩子运行的时候，那些还没有运行完成的应用程序线程还会继续运行下去，如果所有钩子运行完毕，且 runFinalizersOnExit 被设置为 true 的话将会调用 finalizer。随后关闭 JVM，掐断所有没有结束的应用程序线程。

如果钩子一直不能运行完成，或者 finalizer 无法终止，应用程序将只能通过唐突关闭的方式关闭。唐突关闭方式直接终止 JVM 的运行，不会调用钩子，直接掐断所有应用程序线程的运行。

关闭钩子应该是线程安全的，它们在存取共享数据结构的时候必须使用同步机制。此外关闭钩子也必须当心死锁问题。编写关闭钩子的时候要小心各种问题，不要将钩子的正确运行建立在应用程序的任何假设之上。此外，钩子必须能够快速运行完毕。

关闭钩子可以做一些清理工作，例如删除临时文件。如下代码是一个注册关闭钩子的例子：

```
Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run()
        {
            try
            {
                //在 JVM 关闭之前先停止 LogService 服务
                LogService.this.stop();
            }
            catch (InterruptedException ignored)
            {}
        }
    });
```

最常见的用法是注册一个关闭钩子，用于在 JVM 关闭之前逐个关闭应用程序中的服务。

7.4.2. daemon 线程

有时候你想创建一些提供帮助功能的线程，但是你不希望这些线程的存在会阻止 JVM 关闭。这样的线程就是 daemon 线程。

线程被分为两种类型：普通线程和 daemon 线程。JVM 启动后它自动创建的所有线程都是 daemon 线程（除了 main 线程），例如垃圾回收线程。线程创建的时候，从创建者继承 daemon 状态，因此，main 线程创建的线程都是普通线程。

如果某个线程终止，剩下的所有线程都是 daemon 线程，将导致虚拟机优雅地关闭。当虚拟机停止的时候，所有剩下的 daemon 线程将会被掐断。

Daemon 线程应该尽量少使用，且只能用在那些随时被掐断都没问题的情况下。特别地，不能在 daemon 线程中进行 I/O 操作。Daemon 线程最适合处理内务工作，例如有一个背景线程周期性地从内存缓冲区中删除过期条目。

7.4.3. Finalizer

垃圾回收器可以自动回收内存资源，但是文件句柄和 Socket 句柄在使用完毕之后必须显式地归还给操作系统。对于 finalize 方法非空的对象，垃圾回收器会特别对待：在对象被回收的时候，该对象的 finalize 方法会被调用，以释放持久性资源。

由于 finalize 方法在 JVM 管理的一个线程中运行，finalize 中访问的对象都将被多线程共享，因此需要使用同步机制。finalize 方法不担保一定会被运行，对运行的时间也没有任何承诺，此外 finalize 方法会造成性能损失。

在大多数情况下提供一个 finally 块，在其中调用 close 方法就好了，不要使用 finalize 方法。一个例外是，系统库中的一些方法，需要释放通过 native 方法获取的资源，这就需要使用 finalize 方法了。

7.5. 本章小结

Java 没有提供内建的机制来支持任务取消，它提供了中断机制。中断机制一般用来实现任务取消，不过你在编写任务的时候需要响应中断，做特定的处理。使用 Executor 框架简化了可取消任务和服务的构建。

第八章 使用线程池

第六章中介绍了任务执行框架，该框架提供了一个简单灵活的方式将任务提交与任务执行解耦。第七章讨论了在使用任务执行框架过程中遇到的一些生命周期方面的问题。本章主要讨论线程池配置与微调方面的一些高级选项、在使用任务执行框架过程中应该注意的风险，此外，还提供了一些高级示例。

8.1. 任务与执行策略的耦合

我们之前提到，任务执行框架将任务提交与任务执行解耦。不过这么说有点言过其实了。虽然任务执行框架在指定与修改执行策略方面提供了很大的灵活性，但是并不是所有的任务与所有的执行策略都兼容。不同的任务往往需要指定不同的执行策略：

- **相互依赖的任务**。多数情况下任务是相互独立的，一个任务的执行不依赖于其他任务的执行。如果在线程池中执行相互独立的并行任务，你可以自由改变线程池的大小，这样做只会影响性能，不会影响功能。如果提交给一个线程池的任务是相互依赖的，这时候就要小心设计执行策略了，不然会出现活跃性问题。
- **在单线程中执行的任务**。单线程执行器可以确保任务只在这个线程中执行，这样就可以放松对任务代码的线程安全性要求。
- **响应时间敏感的任务**。GUI 应用程序一般对响应时间比较敏感。当任务比较耗时的时候，不要在事件分派线程中执行，可开一个单线程执行，或者开个线程池来执行。
- **使用 ThreadLocal 的任务**。ThreadLocal 可以确保每个线程有一个独立版本的变量。然而任务执行器可以在适当的时候重用线程，在任务少的时候可以回收一些线程，在任务多的时候可以创建一些线程，添加到线程池中。此外如果某个任务抛出非检查性异常导致工作线程终止，任务执行器还会创建一个新的工作线程来替换。因此，如果任务中使用了 ThreadLocal 一定要小心。

线程池最适合用于执行一组相互独立的同类型任务。向线程池中提交一些依赖于其他任务的任务将会增加死锁的可能性。

8.1.1. 线程饥饿死锁

如果线程池中一个任务依赖于其他任务，将可能造成死锁。如果使用单线程执行策略，一个任务向执行器提交另一个任务，等待其执行完毕，然后再继续运行，这将肯定会造成死锁。这是因为第二个任务被放置在调度队列中一直得不到执行，这种现象被称为线程饥饿死锁。

每当你向任务执行器提交一些非独立的任务的时候，你需要考虑到线程饥饿死锁的问题。

你需要将线程池大小或配置方面的约束记录下来。如果你的应用程序有一个拥有 10 个连接的数据库连接池，并且每个任务都需要一个数据库连接，那么你的线程池中的线程数不能超过 10 个，因为如果超过 10 个，有部分工作线程在获取数据库连接的时候将会阻塞等待。

8.1.2. 耗时任务

如果线程池中所有线程都忙于执行耗时任务，将会导致对新提交的任务长期无法响应。如果线程池中线程的数量少于稳定状态下必须执行的耗时任务数量将会导致这种情况。

解决这个问题的办法是，总是调用带时间界限的阻塞式方法。Thread.join，BlockingQueue.put，CountDownLatch.await 方法都有带时间界限和不带时间界限两个版本。如果阻塞式方法超时你可以将任务标记为失败，然后重新提交或者丢弃。这样就可以确保目前能够运行的任务尽量先执行完成。另外，如果线程池中的多数线程常常阻塞，这往往预示着线程池太小了。

8.2. 设置线程池大小

线程池的理想大小一般取决于要执行的任务的类型和数量。一般来说，线程池的大小很少是硬编码的，可以从配置文件中读取，也可以根据当前可用的处理器数量来确定。

一般来说线程池不能太大，也不能太小。如果线程池太大，就会有大量线程竞争有限的处理器和内存资源，有可能造成资源耗尽。如果线程池太小就会浪费系统资源，影响吞吐量。

为了确定线程池的大小，你需要搞明白系统的计算环境、任务的特点等问题。

系统中有多少个处理器？有多大内存？任务是计算密集型的还是 I/O 密集型的，或者是混合式的？任务需要使用稀缺资源吗（比如数据库连接）？如果你有多种不同特性的任务，考虑使用多个线程池，这样每个线程池就可以根据任务特性进行优化。

对于计算密集型任务，如果系统中有 N 个 CPU，线程池的大小应该设置为 N+1。如果任务包含 I/O 或其他阻塞操作，你就需要更大的线程池。为了设置合理的线程池大小，你需要估计任务等待时间与计算时间的比值。

定义：

N_{cpu} 表示系统中 CPU 数量

U_{cpu} 表示 CPU 资源利用率，范围在[0,1]之间

W/C 表示任务等待时间与计算时间比值

可以根据如下公式计算达到指定的 CPU 资源利用率的线程数量：

$N_{threads}=N_{cpu}*U_{cpu}*(1+W/C)$

你可以调用如下代码获取系统中 CPU 数量：

```
int N_CPUS=Runtime.getRuntime().avaiableProcessors();
```

当然，CPU 不是线程池需要消耗的唯一资源。其他资源，比如内存、文件句柄、Socket 句柄、数据库连接对线程池大小的合理值也有影响。使用这些资源计算线程池大小上限比较简单。比如每个任务需要消耗 10M 的内存，总共有 50M 的内存空间，这样最多可以支持 5 个线程，因此线程池最大不能超过 5。

当任务需要某个资源池中的资源（比如需要获取数据库连接池中的连接）的时候，线程池的大小和资源池的大小相互影响。如果每个任务都需要一个数据库连接，那么线程池的大小不能超过连接池的大小。相反，如果只有线程池中的任务使用数据库连接池中的连接，那么线程池的大小将会影响数据库连接池的大小。

8.3. 配置 ThreadPoolExecutor

ThreadPoolExecutor 是一个灵活、健壮的线程池实现，提供了很多自定义配置功能。Executors 类中的 newCachedThreadPool、newFixedThreadPool、newScheduledThreadExecutor 等方法返回的线程池都是用 ThreadPoolExecutor 实现的，不过做了不同的配置。

8.3.1. 线程的创建与销毁

线程池核心池大小、最多允许的线程数、活跃时间等参数影响着工作线程的创建和销毁。如果核心池大小够用则线程池中会一直维持这么多的活动线程。如果核心池不够用则会创建额外的线程，但池中活动线程数量不能超过最多允许的线程数。如果一个活动线程的空闲时间超过活跃时间且当前线程池中活动线程数量超过核心池大小的话将有可能回收这个线程。

ThreadPoolExecutor 类有很多构造函数，通用性最强的构造函数声明如下：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) { ... }
```

通过微调核心池的大小与活跃时间两个参数，你可以鼓励线程池尽快回收空闲线程的资源。不过这是一个平衡问题，如果回收得过快，随后又要创建工作线程的话就不划算了。

Executors.newFixedThreadPool 方法创建的线程池可以指定核心池大小和最多允许线程数（两者相等），没有指定活跃时间，活跃时间默认为无穷大。如果某个线程由于抛出异常而终止，且队列中还有任务等待执行的话，就会新创建一个线程来代替它。

Executors.newCachedThreadPool 方法创建的线程池将最多允许的线程数指定为 Integer.MAX_VALUE，将核心池的大小指定为零，活跃时间指定为一分钟。这样返回的线程池在任务多的时候可以无限扩张，在任务少的时候可以逐渐收缩。

8.3.2. 管理队列中的任务

有界线程池限制了可以并发执行的任务数量。单线程执行器是一个特例，它确保没有任务可以并发执行。

在 6.1.2 节中我们了解到无界线程创建会导致应用程序在重载情况下不稳定。为了解决这个问题，我们使用一个固定大小的线程池代替了为每个任务新创建一个新线程的任务执行策略。然而，这样仍不能完全解决问题，如果任务到达

的速度超过任务处理的速度，就会有大量的任务被放在队列里等待执行。

有时候虽然平均任务提交速度比较平稳，但短期内可能提交大量的任务。虽然队列可以对这种情况进行平滑，但是如果短期内提交的任务过多，还是有可能造成内存耗尽。一般来说在内存耗尽之前，随着任务队列的增长，响应速度就会变得越来越差了。

ThreadPoolExecutor 允许你为其提供自定义的 BlockingQueue，用来作为任务队列。newFixedThreadPool 方法和 newSingleThreadExecutor 方法返回的线程池默认使用无界队列 LinkedBlockingQueue。

更稳定的方案是使用有界队列，比如 ArrayBlockingQueue。如果使用有界队列，队列的大小与线程池的大小需要综合调整。

对于大型的或者无界的线程池来说，你可以使用 SynchronousQueue 代替任务队列。SynchronousQueue 不是队列，它是一种线程间传递对象的机制。要向 SynchronousQueue 中添加一个元素，必须有另一个线程正在等待接受这个元素。如果没有其他线程在等待从 SynchronousQueue 中接收任务，并且线程池中当前活动线程数少于上限，那么 ThreadPoolExecutor 就会创建一个工作线程来接收这个任务。如果没有其他线程在等待从 SynchronousQueue 中接收任务，并且当前线程池中活动线程数已经到达上限，该任务将会被拒绝（拒绝策略见下节）。

由于可以直接将任务交给工作线程，因此使用 SynchronousQueue 效率更高。如果线程池是无界的或者拒绝一个任务是可以接受的话可以考虑使用它。newCachedThreadPool 方法返回的线程池默认使用 SynchronousQueue。

如果使用 FIFO 队列比如 LinkedBlockingQueue 或者 ArrayBlockingQueue，先提交的任务将会被先执行。如果使用 PriorityBlockingQueue，优先权高的任务将被先执行。优先权可使用 Comparable 定义也可以使用 Comparator 定义。

有界线程池和有界队列只适合于任务相互独立的情况。如果任务相互依赖，有界线程池或有界队列将会造成线程饥饿死锁，这时候应该使用无界线程池，比如 Executors.newCachedThreadPool。

8.3.3. 饱和策略

当线程池中的有界任务队列满了的时候，就要考虑饱和策略了。可以使用 setRejectedExecutionHandler 方法设置 ThreadPoolExecutor 的饱和策略。如

果向已关闭的执行器提交任务，这个任务也会被交给饱和策略来处理。有几种不同的饱和策略：AbortPolicy、CallerRunsPolicy、DiscardPolicy 和 DiscardOldestPolicy。

默认的饱和策略是 AbortPolicy。这将导致 execute 方法抛出非检查性异常 RejectedExecutionException。

如果新提交的任务无法被加入任务执行队列，DiscardPolicy 会将该任务无声地丢弃。

如果新提交的任务无法被加入任务执行队列，DiscardOldestPolicy 会将队首的任务丢弃，然后将该任务插入队列。

CallerRunsPolicy 不会在线程池中执行任务，而是让任务在调用 execute 的线程中执行。这样可以给线程池中的工作线程一些时间，让他们去完成任务，下次再调用 execute 的时候说不定就能将任务入队了。

如下代码演示了如何为线程池设置饱和策略：

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(N_THREADS, N_THREADS,
                                 0L, TimeUnit.MILLISECONDS,
                                 new LinkedBlockingQueue<Runnable>(CAPACITY));
executor.setRejectedExecutionHandler(
                                 new ThreadPoolExecutor.CallerRunsPolicy());
```

execute 方法是非阻塞的，如何实现在任务队列满的时候将 execute 方法阻塞呢？可以使用信号量来实现，例如如下代码所示：

```
/**
 * 阻塞式任务执行器
 * 信号量的大小应该设置为线程池大小加上允许在队列中等待的任务数量
 * 当任务过多的时候将会导致submitTask方法阻塞
 * exec的任务队列应该是无界的
 */
@ThreadSafe
public class BoundedExecutor
{
    private final Executor exec;          //内部代理执行器
    private final Semaphore semaphore;    //信号量

    public BoundedExecutor(Executor exec, int bound)
    {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
```

```
    }

    /**
     * 提交任务，本方法是可中断的阻塞式方法
     */
    public void submitTask(final Runnable command)
            throws InterruptedException
    {
        //提交任务之前必须先获取信号量标记
        semaphore.acquire();
        try
        {
            exec.execute(new Runnable()
            {
                public void run()
                {
                    try
                    {
                        command.run();
                    }
                    finally
                    {
                        //任务执行完毕之后返回信号量标记
                        semaphore.release();
                    }
                }
            });
        }
        catch (RejectedExecutionException e)
        {
            //如果任务被拒绝返回信号量标记
            semaphore.release();
        }
    }
}
```

在这种情况下任务队列应该使用无界队列，信号量的大小设置为线程池大小加上你想允许的正在等待执行的任务的数量。如果提交的任务过多，来不及执行就会导致 submitTask 方法阻塞。

8.3.4. ThreadFactory 接口

当线程池需要创建新的线程的时候，它通过 ThreadFactory 来创建。该接口

有一个名为 newThread 的方法，默认行为是创建一个新的，非 daemon 线程。

有很多原因使得我们需要在创建线程池的时候为其指定自定义的 ThreadFactory，例如需要为新创建的 Thread 设置 UncaughtExceptionHandler 或者创建一个自定义的 Thread 对象作为工作线程。你也可能需要修改新创建的工作线程的优先权或者 daemon 状态。

如下代码中的 MyThreadFactory 类实现了 ThreadFactory，其 newThread 方法返回一个 MyAppThread 对象。MyAppThread 类继承自 Thread 类。

```
public class MyThreadFactory implements ThreadFactory
{
    //线程池名
    private final String poolName;

    /**
     * 构造函数
     */
    public MyThreadFactory(String poolName)
    {
        this.poolName = poolName;
    }

    /**
     * 实现newThread方法，返回MyAppThread对象
     */
    public Thread newThread(Runnable runnable)
    {
        return new MyAppThread(runnable, poolName);
    }
}
```

如下代码定义了 MyAppThread 类，你可以为其设置线程名、设置自定义的 UncaughtExceptionHandler、维护统计信息（已经创建了多少个 MyAppThread 对象？现在活动的 MyAppThread 有多少？）。此外，如果处于 debug 状态的话，工作线程在启动和终止的时候会向日志中记录下该相应的事件。

```
/**
 * MyAppThread类扩展自Thread
 * 专用于作为线程池中的工作线程，比Thread增加了一些日志记录功能
 */
```

```
public class MyAppThread extends Thread
{
    //是否在Debug
    private static volatile boolean debugLifecycle = false;
    //默认线程名
    public static final String DEFAULT_NAME = "MyAppThread";
    //到目前为止创建的MyAppThread线程数
    private static final AtomicInteger created = new AtomicInteger();
    //目前活动的MyAppThread线程数
    private static final AtomicInteger alive = new AtomicInteger();
    //日志记录器
    private static final Logger log = Logger.getAnonymousLogger();

    /**
     * 构造函数
     */
    public MyAppThread(Runnable r)
    {
        //将线程名设置为"MyAppThread"
        this(r, DEFAULT_NAME);
    }

    /**
     * 构造函数
     */
    public MyAppThread(Runnable runnable, String name)
    {
        //将线程名设置为name+"-"+序号
        super(runnable, name + "-" + created.incrementAndGet());
        //为线程设置UncaughtExceptionHandler
        setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()
        {
            //如果线程因任务执行过程中抛出运行时异常而终止，该方法将被调用
            public void uncaughtException(Thread t,Throwable e)
            {
                log.log(Level.SEVERE,"UNCAUGHT in thread " + t.getName(), e);
            }
        });
    }

    public void run()
    {
        // Copy debug flag to ensure consistent value throughout.
        boolean debug = debugLifecycle;
```

```
        if (debug)
            log.log(Level.FINE, "Running " + getName());
        try
        {
            alive.incrementAndGet();
            //真正开始执行任务
            super.run();
        }
        finally
        {
            alive.decrementAndGet();
            if (debug)
                log.log(Level.FINE, "Exiting " + getName());
        }
    }

    public static int getThreadsCreated()
    {
        return created.get();
    }

    public static int getThreadsAlive()
    {
        return alive.get();
    }

    public static boolean getDebug()
    {
        return debugLifecycle;
    }

    public static void setDebug(boolean b)
    {
        debugLifecycle = b;
    }
}
```

8.3.5. 在创建之后配置 ThreadPoolExecutor

ThreadPoolExecutor 在创建之后还可以通过 setter 方法进行配置。如果 ThreadPoolExecutor 对象是通过 Executors 的 newXXX 方法创建的，你可以先将其强制类型转换为 ThreadPoolExecutor 然后再调用 setter 方法对其进行配置，例如：

```
ExecutorService exec = Executors.newCachedThreadPool();
if (exec instanceof ThreadPoolExecutor)
    ((ThreadPoolExecutor) exec).setCorePoolSize(10);
else
    throw new AssertionError("Oops, bad assumption");
```

Executors 类中有一个工厂方法 unconfigurableExecutorService，该方法将一个 ThreadPoolExecutor 包装起来并只向外暴露 ExecutorService 定义的方法，这样，返回的线程池对象就不能重新配置了。

单线程执行器实际上是只有一个线程的线程池，一次只能执行一个任务。如果某段代码要增加单线程执行器中线程的数量，将会破坏其执行语义。因此，你可以用 unconfigurableExecutorService 方法将其包装起来，这样就避免了单线程执行器被重新配置的可能了。

如果你要将执行器暴露给不可信的代码，最好使用这种方法将其包装起来以防止任务执行器被重新配置。

8.4. 扩展 ThreadPoolExecutor

ThreadPoolExecutor 类可以被自定义类扩展，其中几个方法可以被覆盖，比如：beforeExecute、afterExecute 和 terminated。

beforeExecute 方法和 afterExecute 方法在任务执行线程中被调用，分别在任务执行线程执行任务前和执行任务后调用，可用于记录日志或收集统计信息。不管是任务正常执行完成或者是由于抛出异常而执行完成，afterExecute 方法都会被调用（如果 run 方法抛出 Error，afterExecute 方法不会被调用）。如果 beforeExecute 抛出一个运行时异常，任务将不会被执行，afterExecute 也不会被调用。

Terminated 方法在线程池完成关闭过程（所有任务执行完毕、所有线程关闭）之后被调用。可用于释放 Executor 在其生命周期中申请的资源、记录日志或者收集统计信息。

8.4.1. 示例：为线程池添加统计信息

如下代码中的 TimingThreadPool 类是一个自定义线程池，扩展自 ThreadPoolExecutor，覆盖了 beforeExecute、afterExecute 和 terminated 三个方法。

```java
/**
 * 自定义线程池
 */
public class TimingThreadPool extends ThreadPoolExecutor
{

    /**
     * 构造函数，采取单线程执行策略
     */
    public TimingThreadPool()
    {
        super(1, 1, 0L, TimeUnit.SECONDS, null);
    }

    //用于表示任务起始时间，每个工作线程能获得自己的版本
    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    //日志记录器
    private final Logger log = Logger.getLogger("TimingThreadPool");
    //已经执行完成的任务数量
    private final AtomicLong numTasks = new AtomicLong();
    //所有任务运行时间之和
    private final AtomicLong totalTime = new AtomicLong();

    /**
     * 在每个任务执行之前被调用的钩子方法（该方法在工作线程中执行）
     */
    protected void beforeExecute(Thread t, Runnable r)
    {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    /**
     * 在每个任务执行完成后被调用的钩子方法（该方法在工作线程中执行）
     */
    protected void afterExecute(Runnable r, Throwable t)
    {
        try
        {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
```

```
            log.fine(String.format("Thread %s: end %s, time=%dns",
                                                    t, r,taskTime));
        }
        finally
        {
            super.afterExecute(r, t);
        }
    }

    /**
     * 在线程池被关闭后被调用的钩子方法
     */
    protected void terminated()
    {
        try
        {
            //打印统计信息
            log.info(String.format("Terminated: avg time=%dns",
                                       totalTime.get()/numTasks.get()));
        }
        finally
        {
            super.terminated();
        }
    }
}
```

8.5. 并行递归算法

一个包含大量计算或阻塞式 I/O 的循环体一般可并行处理，例如：

```
void processSequentially(List<Element> elements)
{
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements)
{
    for (final Element e : elements)
    {
        exec.execute(new Runnable()
        {
            public void run()
            {
```

```
                process(e);
            }
        });
    }
}
```

上例将一个线性化循环体转换成了并行化的循环体，之所以可以这么做是因为我们假设对不同元素的处理操作是相互独立的。

processInParallel 方法比 processSequentially 方法快得多，因为它只需要将各个任务提交给执行器，然后就可以返回，而不需要等待所有元素都处理完毕才返回。

当循环处理的各个任务相互独立的时候可以将线性化循环转换为并行化循环。

在某些递归算法设计中也可使用并行化。如果每次递归都不需要使用子递归的处理结果的话，可以将顺序递归算法转换为并行递归算法，例如：

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
                                               Collection<T> results)
{
    for (Node<T> n : nodes)
    {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}

public<T> void parallelRecursive(final Executor exec,
                                 List<Node<T>> nodes,
                                 final Collection<T> results)
{
    for (final Node<T> n : nodes)
    {
        exec.execute(new Runnable()
        {
            public void run()
            {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

当 parallelRecursive 返回的时候，每个树节点都被访问过了（树节点遍历是顺序进行的，只有 compute 方法调用是并行的）并且对于每个节点的处理操作也提交给了执行器。parallelRecursive 方法的调用者可以等待任务执行器执行完毕，然后获取处理结果，具体方法是调用任务执行器的 shutdown 方法，然后调用 awaitTermination 方法等待并行处理完成，例如：

```
public<T> Collection<T> getParallelResults(List<Node<T>> nodes)
                                            throws InterruptedException
{
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive(exec, nodes, resultQueue);

    //将已提交的任务运行完成再关闭任务执行器
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}
```

8.5.1. 示例：一个解谜框架

并行递归算法的一个典型应用是用于找出如何从一个初始状态经过若干个中间状态转换到最终的目标状态，比如大家都熟悉的滑块类游戏、纸牌游戏。

我们将谜题定义为一个初始位置、一个目标位置和所有合法移动规则的组合。如下接口抽象地描述了一个谜题：

```
public interface Puzzle <P, M>
{
    //获取初始位置
    P initialPosition();

    //判断指定的位置是否是目标位置
    boolean isGoal(P position);

    //获取从指定位置出发的所有合法移动
    Set<M> legalMoves(P position);

    //获取从指定位置出发向指定方向移动得到的新位置
    P move(P position, M move);
}
```

其中类型参数 P 表示位置类，类型参数 M 表示移动类。以该接口为基础，我

们可以写一个简单的求解器来搜索推理空间直到找到谜题的解或者推理空间耗尽，谜题无解。

如下代码中的 PuzzleNode 类表示一个由某个位置移动得到的新位置。循着 prev 域不断向上找，可以重构出一个移动序列。

```
@Immutable
public class PuzzleNode <P, M>
{
    //表示当前位置
    final P pos;
    //表示当前位置的前一个位置
    final PuzzleNode<P, M> prev;
    //表示前一个位置如何移动到本位置
    final M move;

    /**
     * 构造函数
     */
    public PuzzleNode(P pos, M move, PuzzleNode<P, M> prev)
    {
        this.pos = pos;
        this.move = move;
        this.prev = prev;
    }

    /**
     * 获取从初始位置移动到当前位置的移动序列
     */
    List<M> asMoveList()
    {
        List<M> solution = new LinkedList<M>();
        for (PuzzleNode<P, M> n=this; n!=null && n.move!=null; n=n.prev)
        {
            solution.add(0, n.move);
        }
        return solution;
    }
}
```

如下代码中的 SequentialPuzzleSolver 类演示了一个顺序性的谜题求解器框架，对推理空间进行深度优先搜索。当发现解的时候就会终止运行，如果推理空间耗尽也没有找到解，则返回 null。

```
public class SequentialPuzzleSolver<P, M>
{
    //要求解的谜题
    private final Puzzle<P, M> puzzle;

    //已经搜索过的位置
    private final Set<P> seen = new HashSet<P>();

    /**
     * 构造函数
     */
    public SequentialPuzzleSolver(Puzzle<P, M> puzzle)
    {
        this.puzzle = puzzle;
    }

    /**
     * 开始求解，如果找到解则返回移动序列，否则返回null
     */
    public List<M> solve()
    {
        P pos = puzzle.initialPosition();
        return search(new PuzzleNode<P, M>(pos, null, null));
    }

    /**
     * 搜索推理空间，如果找到解则返回从初始位置到目标位置的移动序列
     * 否则返回null
     */
    private List<M> search(PuzzleNode<P, M> node)
    {
        //如果该位置没有搜索过
        if (!seen.contains(node.pos))
        {
            seen.add(node.pos);

            //已找到目标位置，返回移动序列
            if (puzzle.isGoal(node.pos))
                return node.asMoveList();

            //本位置不是目标位置，搜索本位置的所有一下个可能的位置
            for (M move : puzzle.legalMoves(node.pos))
            {
                P pos = puzzle.move(node.pos, move);
```

```
                PuzzleNode<P, M> child = new PuzzleNode<P, M>(pos, move, node);
                List<M> result = search(child);
                if (result != null)
                    return result;
            }
        }
        return null;
    }
}
```

如下代码中的 ConcurrentPuzzleSolver 类使用了一个内部类 SolverTask 表示搜索任务。大多数工作都在 run 方法中执行：判断是否已经找到问题的解、评估当前位置是否是目标位置、递归搜索所有可达的下一步位置。

```
public class ConcurrentPuzzleSolver <P, M>
{
    //要求解的谜题
    private final Puzzle<P, M> puzzle;
    //任务执行器
    private final ExecutorService exec;
    //用于存储所有已搜索过的位置
    private final ConcurrentMap<P, Boolean> seen;
    //用于保存问题的解（如果哪个搜索任务发现了问题的解就存储在solution的内部
    //值中）
    protected final ValueLatch<PuzzleNode<P, M>> solution = new
                                              ValueLatch<PuzzleNode<P, M>>();

    /**
     * 构造函数
     */
    public ConcurrentPuzzleSolver(Puzzle<P, M> puzzle)
    {
        this.puzzle = puzzle;
        this.exec = initThreadPool();
        this.seen = new ConcurrentHashMap<P, Boolean>();
    }

    /**
     * 初始化线程池
     */
    private ExecutorService initThreadPool()
    {
        ExecutorService e= Executors.newCachedThreadPool();
        if (e instanceof ThreadPoolExecutor)
```

```
        {
            ThreadPoolExecutor tpe = (ThreadPoolExecutor) e;
            //设置饱和策略为无声丢弃
            tpe.setRejectedExecutionHandler(new ThreadPoolExecutor.
                                                      DiscardPolicy());
        }
        return e;
    }

    /**
     * 开始求解，如果找到解则返回移动序列，否则返回null
     */
    public List<M> solve() throws InterruptedException
    {
        try
        {
            P p = puzzle.initialPosition();
            exec.execute(newTask(p, null, null));

            // 该方法阻塞，直到solution内部的值被设置了为止
            PuzzleNode<P, M> solnPuzzleNode = solution.getValue();
            return (solnPuzzleNode == null) ? null :
                                         solnPuzzleNode.asMoveList();
        }
        finally
        {
            exec.shutdown();
        }
    }

    /**
     * 获取新的解搜索任务
     *
     * @param p 当前位置
     * @param m 移动
     * @param n 上一个位置
     */
    protected Runnable newTask(P p, M m, PuzzleNode<P, M> n)
    {
        return new SolverTask(p, m, n);
    }

    /**
     * 解搜索任务，用于搜索指定的位置
```

```
     */
    protected class SolverTask extends PuzzleNode<P, M> implements Runnable
    {
        /**
         * 构造函数
         */
        SolverTask(P pos, M move, PuzzleNode<P, M> prev)
        {
            super(pos, move, prev);
        }

        public void run()
        {
            //如果已经找到解或者本位置已经被搜索过，任务结束
            if (solution.isSet() || seen.putIfAbsent(pos, true) != null)
                return;

            //如果找到问题的解
            if (puzzle.isGoal(pos))
            {
                solution.setValue(this);
            }
            //当前位置不是问题的解，搜索其下一个位置
            else
            {
                for (M m : puzzle.legalMoves(pos))
                    exec.execute(newTask(puzzle.move(pos, m), m, this));
            }
        }
    }
}
```

为了避免无限循环，本例使用了一个 ConcurrentHashMap 对象来表示所有已经搜索过的位置。使用 ConcurrentHashMap 可以提供多线程安全性，此外 putifAbsent 方法具有原子性，避免了先判断再插入在多线程情况下造成的竞争条件。

顺序版本的谜题求解器进行深度优先搜索，因此所能搜索的深度受到堆栈的限制。并行版本的谜题求解器进行广度优先搜索，因此不会造成堆栈溢出。不过如果推理空间过大，seen 占用的内存空间可能过大，创建的线程数量也可能过多，从而造成内存耗尽。

为了在某个搜索任务找到解的时候停止搜索，我们需要一个机制来判断是否

有某个工作线程找到了问题的解。如果我们只返回第一个找到的解，我们还需要实现在解已经找到的时候再找到其他的解也不会覆盖掉之前找到的解。为了实现这样的功能，我们可以使用锁存器创建一个结果包装类，代码如下：

```
/**
 * 使用锁存器实现的值保持类
 */
@ThreadSafe
public class ValueLatch <T>
{
    //内部值，默认为null
    @GuardedBy("this")
    private T value = null;

    //锁存器
    private final CountDownLatch done = new CountDownLatch(1);

    /**
     * 判断内部值是否已被设置
     */
    public boolean isSet()
    {
        return (done.getCount() == 0);
    }

    /**
     * 设置内部值，如果已经被设置，该方法无效
     */
    public synchronized void setValue(T newValue)
    {
        if (!isSet())
        {
            value = newValue;
            done.countDown();
        }
    }

    /**
     * 获取内部值，该方法在内部值被设置之前会阻塞
     */
    public T getValue() throws InterruptedException
    {
        done.await();
        synchronized (this)
```

```
        {
            return value;
        }
    }
}
```

每个搜索任务首先看 solution 内部值是否已经被设置了，如果已经被设置说明其他搜索任务已经找到了问题的解，本任务就可以结束了。主线程中的 getValue 方法在 solution 内部值被设置之前会一直阻塞，直到某个工作线程发现了问题的解并调用 solution.setValue 方法将其设置到 solution 内部值中。

如果某个工作线程找到了问题的解，主线程会调用线程池的 shutdown 方法，这样已经提交的任务将会被运行结束，新提交的任务将会被拒绝执行。由于已经将饱和策略（拒绝策略）设置为 DiscardPolicy，在 shutdown 方法被调用后新提交的任务都会被无声地丢弃。

ConcurrentPuzzleSolver 没有处理无解的情况，如果推理空间完全搜索之后也没有找到问题的解，solution.getValue 方法就会一直阻塞下去。如何解决这个问题？一种办法是当活动任务数量为零的时候将 solution 的内部值设置为 null，例如：

```
public class PuzzleSolver <P,M> extends ConcurrentPuzzleSolver<P, M>
{
    PuzzleSolver(Puzzle<P, M> puzzle)
    {
        super(puzzle);
    }

    //用于存储当前活动任务数
    private final AtomicInteger taskCount = new AtomicInteger(0);

    protected Runnable newTask(P p, M m, PuzzleNode<P, M> n)
    {
        return new CountingSolverTask(p, m, n);
    }

    class CountingSolverTask extends SolverTask
    {
        CountingSolverTask(P pos, M move, PuzzleNode<P, M> prev)
        {
            super(pos, move, prev);
            //每创建一个搜索任务，taskCount加一
```

```
            taskCount.incrementAndGet();
        }

        public void run()
        {
            try
            {
                super.run();
            }
            finally
            {
                //每个任务完成之后检查当前活动任务数是否为零
                //如果为零则将solution内部值设置为null，表示没找到问题的解
                if (taskCount.decrementAndGet() == 0)
                    solution.setValue(null);
            }
        }
    }
}
```

还有一种情况，只要时间足够长，总是能够求出问题的解，但是用户等不及了，我们可以采取好几种办法来终止求解器。一种办法是为 ValueLatch 类提供一个定时版本的 getValue 方法（内部调用定时版的 await 方法），如果时间超时则将 solution 的内部值设置为 null，并关闭任务执行器。另一种方法是如果搜索过指定数量的位置之后还没找到问题的解就将 solution 的内部值设置为 null。第三种办法是提供一个取消机制，让用户自己决定什么时候停止搜索。

8.6. 本章小结

Executor 框架是一个强大的、灵活的并发任务执行框架。它提供了一些微调选项，比如创建和销毁线程的策略、任务队列、饱和策略。此外还提供了一些钩子来扩展其功能。不过有些任务需要特定的执行策略，如何微调，自己根据具体需要判断吧。

第九章 GUI 应用程序

如果你尝试使用 Swing 创建一个简单的 GUI 应用程序，你会意识到 GUI 应用程序有独特的线程问题。为了保持安全性，有些任务必须在事件分派线程中执行，但是你无法在事件分派线程中执行耗时任务，这样会导致用户界面假死。此外，Swing 程序中涉及到的数据结构往往不是线程安全的，如果封闭在事件分派线程中使用没有问题，如果在多线程中使用，必须小心。

如果有一个活动一半在应用程序线程中运行，另一半在事件分派线程中运行就很容易出现多线程 Bug，而且不一定立即发作，只会在一些意想不到的时刻导致系统运行出错。虽然大多数操作都被封闭在事件分派线程中运行，但是在写 GUI 程序的时候你仍然需要小心多线程问题。

9.1. 为什么 GUI 应用程序是单线程的？

在以前 GUI 应用程序是单线程的，GUI 事件在主事件循环中被处理。现代 GUI 框架稍微有点不同：创建了一个专门用于处理 GUI 事件的事件分派线程。

单线程 GUI 框架不是 Java 独有的，Qt、NextStep、MacOS Cocoa、X Windows 等都使用单线程 GUI 框架。很多人尝试过多线程 GUI 框架，但是由于竞争条件和死锁问题最终还是使用一个专用线程从事件队列中取事件，然后调用相应的事件处理程序来处理这个事件。

单线程 GUI 框架使用线程封闭来达到线程安全性。所有的 GUI 对象，包括可视化组件和数据模型都在事件分派线程中被访问。不过编程者需要确保这些对象被封闭在事件分派线程中。

9.1.1. 顺序化事件处理

GUI 应用程序可以处理细粒度的事件，例如点击鼠标、键盘按键、时钟到期等。事件就像任务，事件处理机制就像任务处理器。

由于只有一个事件分派线程，因此，GUI 事件是被顺序响应的。不可能出现对两个事件的处理操作重叠的情况。了解这一点对写代码有很大帮助，你不用担心事件之间的相互干扰问题。

顺序化事件处理方式的问题是如果某个事件处理任务比较耗时的话，其他事件就必须在事件队列中长期等待。这就容易出现程序假死的现象。为了解决这个

问题，对于耗时的事件处理任务，你必须另开一个线程，以便控制权可以立即返回给事件分派线程。如果耗时事件处理任务在另一个线程中运行的过程中需要更新进度条的进度或者在任务完成后需要提供一个可视化的反馈的话，又需要在事件分派线程中执行代码，这样的话情况就比较复杂了，不过不用担心，后面还会讨论这个问题。

9.1.2. Swing 中的线程封闭

Swing 中的所有可视化组件（例如 JButton 和 JTable）和数据模型对象（比如 TableModel 和 TreeModel）都封闭到事件分派线程中，因此所有访问这些对象的代码都必须在事件分派线程中执行。这样做的优点是事件分派线程中的事件处理代码在访问这些对象的时候不需要担心同步问题，缺点是不能在其他线程中访问这些对象。

Swing 单线程规则：Swing 可视化组件和数据模型对象只能在事件分派线程中创建、修改和查询。

不过也有例外，对于在 JavaDoc 中明确标明为线程安全的方法可以在其他线程中调用。其他例外包括：

- SwingUtilities.isEventDispatchThread，该方法用于判断当前线程是否是事件分派线程。
- SwingUtilities.invokeLater，该方法用于将一个 Runnable 发送到事件分派线程中执行（该方法可以在任意线程中调用）。
- SwingUtilities.invokeAndWait，该方法将一个 Runnable 发送到事件分派线程中执行，然后阻塞，直到其执行完成（只能在非事件分派线程中调用）。
- 添加或删除事件监听器的方法（可以在任何线程中被调用，但是事件处理代码还是在事件分派线程中运行）。
- 入队一个 repaint 请求或 revalidation 请求（可在任意线程中调用）。

使用 Executor 框架很容易实现 SwingUtilities 类，例如：

```
public class SwingUtilities
{
    private static final ExecutorService exec = Executors
            .newSingleThreadExecutor(new SwingThreadFactory());
```

```
    private static volatile Thread swingThread;

    private static class SwingThreadFactory implements ThreadFactory
    {
        public Thread newThread(Runnable r)
        {
            swingThread = new Thread(r);
            return swingThread;
        }
    }

    public static boolean isEventDispatchThread()
    {
        return Thread.currentThread() == swingThread;
    }

    public static void invokeLater(Runnable task)
    {
        exec.execute(task);
    }

    public static void invokeAndWait(Runnable task)
            throws InterruptedException, InvocationTargetException
    {
        Future<?> f = exec.submit(task);
        try
        {
            f.get();
        }
        catch (ExecutionException e)
        {
            throw new InvocationTargetException(e.getCause());
        }
    }
}
```

实际上 SwingUtilities 类不是这样实现的，因为 Executor 框架出现得比 SwingUtilities 晚。不过如果 Executor 出现得比 SwingUtilities 早的话，这段代码将是更好的实现方式。

Swing 任务分派线程可以被理解为一个单线程任务执行器，不断从任务队列中获取事件，然后处理。如果某个事件处理代码导致这个单线程异常终止，将会

无声地创建一个新的线程替换它。当事件处理任务不怎么耗时或者事件处理代码的可预测性不是很重要或者事件处理任务必须是顺序执行的时候，这个使用单事件分派线程顺序执行 GUI 事件处理任务的策略是合理的。

如下代码中的 GuiExecutor 类是一个任务执行器，它将任务代理给 SwingUtilities 执行。在其他 GUI 框架中也可以实现类似的功能，比如 SWT 中的 Display.asyncExec 方法等价于 Swing 中的 SwingUtilities.invokeLater。

```
public class GuiExecutor extends AbstractExecutorService
{
    // Singletons have a private constructor and a public factory
    private static final GuiExecutor instance = new GuiExecutor();

    private GuiExecutor()
    {
    }

    public static GuiExecutor instance()
    {
        return instance;
    }

    public void execute(Runnable r)
    {
        if (SwingUtilities.isEventDispatchThread())
            r.run();
        else
            SwingUtilities.invokeLater(r);
    }

    public void shutdown()
    {
        throw new UnsupportedOperationException();
    }

    public List<Runnable> shutdownNow()
    {
        throw new UnsupportedOperationException();
    }

    public boolean awaitTermination(long timeout, TimeUnit unit)
            throws InterruptedException
    {
```

```
        throw new UnsupportedOperationException();
    }

    public boolean isShutdown()
    {
        return false;
    }

    public boolean isTerminated()
    {
        return false;
    }
}
```

9.2. 不耗时的 GUI 事件处理任务

在 GUI 应用程序中事件源自于事件分派线程，然后冒泡给应用程序中的事件处理器，事件处理器中的代码对事件进行处理，有可能会影响可视化组件的状态，对于简单的，不耗时的事件处理任务，整个过程都在事件分派线程中进行。对于耗时任务，可能涉及到其他线程。

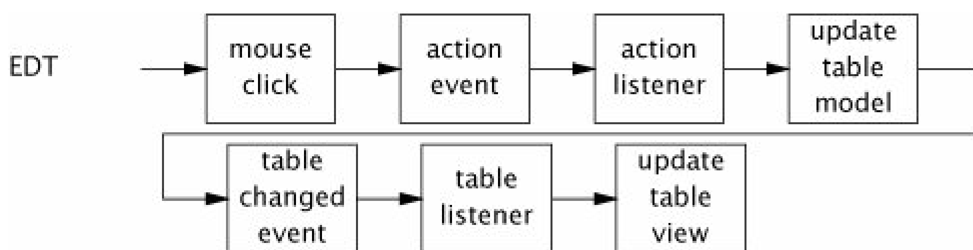
如下代码创建了一个按钮，当被点击之后，该按钮随机改变颜色。

```
final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        button.setBackground(new Color(random.nextInt()));
    }
});
```

当用户点击这个按钮的时候，Java 工具箱发布一个 ActionEvent 给事件监听器。事件监听器选择一个新的颜色，然后修改 Button 对象的背景色。因此事件来自于 GUI 工具箱，然后发送给事件监听器，事件监听器响应事件，修改用户界面，整个控制流程如下：

如果事件处理任务不耗时并且事件监听器中只存取 GUI 对象(或者其他封闭在事件分派线程中的对象或线程安全对象)，你可以不用考虑线程安全问题，随意编写事件处理代码。

下图展示了一种稍微复杂一点的 GUI 事件场景。这涉及到使用数据模型，比如 TableModel 或者 TreeModel。Swing 将大多数可视化组件分割成两个部分：模型和视图。要显示的数据放在模型中，决定如何显示数据的规则放在视图中。模型对象可以触发事件，表示这个模型已经被修改过了，视图可以监听这些事件。



当视图接收到一个事件表示对应的模型对象已经被修改过了，它从对应的模型中获取新的数据，然后更新显示。所以，上图表示的意思是，当用户单击按钮，在按钮单击事件处理器中修改 Table 的数据模型，然后调用这个数据模型的某个 fireXxx 方法，这样就会触发表格视图的模型修改事件监听器，在此事件处理器中表格视图更新其显示。

控制流一直在事件分派线程中。由于 fireXxx 方法是直接调用所有监听者的事件处理方法，而不是提交一个新的事件到事件队列中去，因此 fireXxx 方法只能在事件分派线程中被调用。

9.3. 耗时的 GUI 事件处理任务

如果所有的事件处理任务都是不耗时的，且应用程序没有至关重要的非 GUI 部分，那么整个应用程序可以在事件分派线程中执行，你不必关系多线程问题。然而，有的 GUI 应用程序需要执行耗时任务，比如拼写检查、后台编译或者获取网络资源等。这些任务必须在单独的线程中运行，以便当这些任务在运行的时候，GUI 应用程序还能响应用户事件。

对于耗时的事件处理任务，我们可以创建一个任务执行器。一个缓冲线程池或许是个好的选择，一般来说 GUI 应用程序中耗时任务不会很多，因此使用缓冲线程池一般不会造成资源耗尽。

我们可以从一个简单的耗时事件处理任务开始，该任务不支持取消或进度显示，且任务完成后不更新用户界面。随后我们会逐渐添加这些特征。如下代码显示了这个耗时的事件处理任务：

```
ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        backgroundExec.execute(new Runnable()
        {
            public void run()
            {
                doBigComputation();
            }
        });
    }
});
```

actionPerformed 方法在事件分派线程中执行，run 方法在线程池中执行。这个例子提交了一个耗时任务然后就不管了，这样做是有问题的，因为大多数耗时任务执行完毕之后都需要给出一个可视化的反馈。但是你不能在工作线程中访问可视化组件，因此任务完成后必须提交一个任务给事件分派线程，以便更新用户界面。

如下代码展示了一种直截了当的方法，用于在耗时任务完成之后将按钮设为可用，将 Label 的文本设为“idle”。

```
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable()
        {
            public void run()
            {
                try
                {
                    doBigComputation();
                }
```

```
                finally
                {
                    //提交一个任务，让其在事件分派线程中运行
                    SwingUtilities.invokeLater(new Runnable()
                    {
                        public void run()
                        {
                            button.setEnabled(true);
                            label.setText("idle");
                        }
                    });
                }
            }
        });
    }
});
```

9.3.1. 取消耗时的事件处理任务

有时候耗时任务运行的时间太长了，用户需要取消它。你可以直接使用中断来实现耗时任务的取消，不过使用 Future 更方便，因为 Future 是专门设计来管理可取消任务的。

当你调用 Future.cancel(true)的时候，将会中断工作线程，如果工作线程中运行的任务被设计成可响应中断的，就可以实现提前终止任务。如下代码展示了一个任务，不断地查询工作线程的中断状态，如果发现工作线程被中断了就提前终止。

```
Future<?>  runningTask = null;    //封闭在事件分派线程中使用
...
startButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        if (runningTask == null)
        {
            runningTask = backgroundExec.submit(new Runnable()
            {
                public void run()
                {
                    while (moreWork())
                    {
                        if (Thread.currentThread().isInterrupted())
```

```
                        {
                            cleanUpPartialWork();
                            break;
                        }
                        doSomeWork();
                    }
                    runningTask = null;
                }
            });
        }
    }
});

cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        if (runningTask != null)
        {
            runningTask.cancel(true);
            runningTask = null;
        }
    }
});
```

由于 runningTask 被封闭在事件分派线程中使用，因此设置和检查它都不需要同步，startButton 按钮可以确保一次只有一个背景线程在运行。但是我们往往需要知道背景线程执行的进度，并在执行完成后接到一个可视化的反馈。我们将在下一段中解决这个问题。

9.3.2. 进度和任务完成指示

使用 Future 来表示一个长期运行的任务极大地简化了任务取消的实现。FutureTask 有一个 done 方法，在任务完成后会被自动调用。我们设计了一个背景任务类 BackgroundTask，它提供了任务进度通知、取消、任务完成通知三大功能，代码如下：

```
/**
 * BackgroundTask
 * 支持取消、完成通知、进度通知的背景任务类
 */
public abstract class BackgroundTask<V> implements Runnable, Future<V>
{
```

```
private final FutureTask<V> computation = new Computation();

/**
 * 内部类Computation
 */
private class Computation extends FutureTask<V>
{
    public Computation()
    {
        super(new Callable<V>() {
            public V call() throws Exception
            {
                return BackgroundTask.this.compute();
            }
        });
    }

    /**
     * 覆盖done方法，提供一个钩子，在事件分派线程中获取任务
     * 结果，然后调用onCompletion方法
     */
    protected final void done()
    {
        GuiExecutor.instance().execute(new Runnable()
        {
            public void run()
            {
                V value = null;
                Throwable thrown = null;
                boolean cancelled = false;
                try
                {
                    value = get();
                }
                catch (ExecutionException e)
                {
                    thrown = e.getCause();
                }
                catch (CancellationException e)
                {
                    cancelled = true;
                }
                catch (InterruptedException consumed)
                {
```

```
                }
                finally
                {
                    onCompletion(value, thrown, cancelled);
                }
            };
        });
    }
}

/**
 * 设置任务执行进度，该方法在工作线程中执行
 *
 * @param current 当前进度值
 * @param max 最大进度值
 */
protected void setProgress(final int current, final int max)
{
    GuiExecutor.instance().execute(new Runnable()
    {
        public void run()
        {
            onProgress(current, max);
        }
    });
}

/**
 * 任务执行方法，该方法在工作线程中执行
 */
protected abstract V compute() throws Exception;

/**
 * 任务完成通知方法，在事件分派线程中执行
 *
 * @param result 任务执行结果
 * @param exception 任务执行过程中抛出的异常
 * @param cancelled 任务是否被取消
 */
protected void onCompletion(V result, Throwable exception,
                            boolean cancelled)
{
}
```

```
    /**
     * 进度通知方法，在事件分派线程中执行
     */
    protected void onProgress(int current, int max)
    {
    }

    // Future中其他方法代理给computation
    public boolean cancel(boolean mayInterruptIfRunning)
    {
        return computation.cancel(mayInterruptIfRunning);
    }

    // Future中其他方法代理给computation
    public V get() throws InterruptedException, ExecutionException
    {
        return computation.get();
    }

    // Future中其他方法代理给computation
    public V get(long timeout, TimeUnit unit) throws
            InterruptedException, ExecutionException,TimeoutException
    {
        return computation.get(timeout, unit);
    }

    // Future中其他方法代理给computation
    public boolean isCancelled()
    {
        return computation.isCancelled();
    }

    // Future中其他方法代理给computation
    public boolean isDone()
    {
        return computation.isDone();
    }

    // 实现Runnable中的run方法，代理给computation
    public void run()
    {
        computation.run();
    }
}
```

在 compute 方法中可以调用 setProgress 方法，这样将导致 onProgress 方法在事件分派线程中被调用，在 onProgress 方法中可以更新可视化组件，以反映任务执行进度的变化。

你可以为 BackgroundTask 类创建子类，只需要提供 compute 方法，该方法将在工作线程中执行。你也可以覆盖 onCompletion 方法和 onProgress 方法，它们在事件分派线程中执行。

BackgroundTask 类也支持任务取消，例如：

```
//取消按钮的监听器类
class CancelListener implements ActionListener
{
    BackgroundTask<?> task;
    public void actionPerformed(ActionEvent event)
    {
        if (task != null)
            task.cancel(true);
    }
}

final CancelListener listener = new CancelListener();
listener.task = new BackgroundTask<Void>()
{
    //实现 compute 方法，该方法在工作线程中执行
    public void compute()
    {
        while (moreWork() && !isCancelled())
            doSomeWork();
            return null;
    }

    //覆盖 onCompletion 方法，该方法在事件分派线程中执行
    public void onCompletion(String s, Throwable exception, boolean cancelled)
    {
        cancelButton.removeActionListener(listener);
        label.setText("done");
    }
};

//为取消按钮设置事件监听器
cancelButton.addActionListener(listener);

//在任务执行器中执行该任务
```

```
backgroundExec.execute(task);
```

在 compute 方法中可以通过不断地查询 isCancelled 方法是否返回 true 来判断任务是否已经被取消。

9.3.3. SwingWorker

在本节中我们构建了一个简单的框架来支持 GUI 程序中耗时任务的提交、取消、进度通知和完成通知。这种技术也可以应用于其他单线程 GUI 框架，比如 SWT。

在 Swing 中 SwingWorker 类已经提供了任务取消、任务完成通知和任务进度通知三个功能，JDK 中提供了很多个版本的 SwingWorker，最新更新在 JDK6 中。

9.4. 共享数据模型

Swing 中的可视化组件和数据模型（比如 TableModel 或者 TreeModel）被封闭在事件分派线程中。对于简单的 GUI 应用程序来说，除了事件分派线程之外只有 main 线程。在这样的 GUI 应用程序中遵守单线程规则比较容易：不要在 main 线程中访问可视化组件和数据模型。

更复杂的 GUI 应用程序可能需要在多个线程之间共享数据模型，例如你可能需要用一个树形组件来显示远程文件系统的目录结构。你不希望在显示树形组件之前遍历整个文件系统，这样太耗时了。你可以使用延迟加载的方法，先显示最顶层的目录，当用户打开特定的目录的时候再显示该目录的所有子目录。

查询远程文件系统中某个目录的所有子目录操作比较耗时，需要在后台任务中进行处理，而不能在事件分派线程中直接处理。当后台加载任务完成之后需要将数据传递给树形控件。这也容易实现，使用 invokeLater 方法就能做到，或者将任务执行结果存起来，让事件分派线程轮询。

9.4.1. 线程安全的数据模型

如果数据模型是线程安全的，事件分派线程和背景线程就可以共享它了。线程安全的数据模型在模型被修改之后必须能产生一个事件通知视图更新其显示。

有时候你可以尝试使用 CopyOnWriteArrayList，当你使用迭代器来遍历一个 CopyOnWriteArrayList 的时候，该遍历器所看到的列表内容是该遍历器被创建时看到的视图。当查询操作较多、修改操作较少的时候使用 Copy-On-Write

集合可以得到性能更高的程序。

9.4.2. 分裂的数据模型设计

TableModel 和 TreeModel 等是用于存储要展示的数据的官方数据模型。然而，这些数据模型可能是应用程序中其他数据结构的视图。如果一个 GUI 应用程序既有应用程序域的数据结构又有展示域的数据结构（前者为后者提供数据），我们说这个应用程序采用的是分裂的数据模型设计。

展示域数据模型被封闭在事件分派线程中，其他数据模型应该是线程安全的，可以在事件分派线程中访问也可以在应用程序线程中访问。展示域数据模型监听应用程序域数据模型，当应用程序域数据模型更新之后，它会通知展示域数据模型，展示域数据模型就可以更新其内部数据（可以考虑使用观察者模式实现）。

当一个数据模型必须被多线程共享，但是如果使用线程安全的数据模型的话将会有诸多缺陷（基于响应速度、复杂性、数据一致性考虑）的时候，考虑使用分裂的数据模型。

9.5. 其它形式的单线程子系统

线程封闭技术不只用在 GUI 程序中，当某个功能必须使用单线程子系统实现的时候，都可以使用线程封闭。有时候，由于没有更好的办法来避免同步和死锁问题，只能使用线程封闭。例如，

模仿 GUI 应用程序的实现方式，你可以轻松地创建一个专用线程或者单线程执行器，然后创建任务（任务执行过程中需要访问线程封闭对象），将该任务提交给这个专用线程。将 9.3.2 节的 BackgroundTask 类稍微修改一下就可做出一个单线程子系统。

9.6. 本章小结

GUI 框架几乎总是使用单线程子系统实现，所有与展示相关的代码都作为任务在事件分派线程中执行。由于只有一个事件分派线程，耗时任务将会影响界面的响应速度，因此必须在背景线程中运行。9.3.2 节提供的 BackgroundTask 类以及 JDK 中提供的 SwingWorker 类都支持任务的提交、取消、进度通知、完成通知，使用它们可以简化耗时任务的处理。

第十章 避免活跃性风险

线程安全性与活跃性之间往往有矛盾。我们使用锁来提供线程安全性，但是不分青红皂白地乱用锁可能导致**锁序死锁**问题。相似地，我们使用线程池和信号量来限制资源消耗，但是使用不当将会造成**资源死锁**问题。Java 应用程序无法从死锁中恢复，因此我们必须防止死锁的发生。本章将讨论一些活跃性问题发生的原因以及如何进行规避。

10.1. 死锁

死锁可以用哲学家进餐问题来说明（尽管这个例子可能不太卫生）。五个哲学家围着圆桌吃中餐。有五只筷子，每人左边有一只，右边有一只，哲学家们要么思考要么吃东西。他们每个人吃东西前需要获得两只筷子，吃完之后把筷子放回原处，然后进入思考状态。有这样一种筷子管理策略：一个饥饿的哲学家试图拿起与他相邻的两只筷子，但是如果有一只拿不到，就放回已经拿到手的那只筷子，然后等一会再试。这种方法比较合理，此外也有另一种筷子管理策略：一个饥饿的哲学家试图拿起他左边的筷子，如果成功再试图拿起他右侧的筷子，如果拿不到右侧的筷子则持有其左侧的筷子等待右侧的筷子可用，这种筷子管理策略将有可能导致哲学家们陷入一种僵局：所有人都无法吃东西。每个人都持有其他人需要的资源，同时每个人都需要其他人持有的资源，并且在获取到需要的资源之前不会释放已经到手的资源。

当一个线程永久地持有某个锁的时候，其他要获取这个锁的线程将会永远阻塞。比方说，线程 A 持有锁 L 并试图获取锁 M，同时线程 B 持有锁 M 并试图获取锁 L，这两个线程都将永远等待。这是最简单的一种死锁，也可能出现多个线程相互等待获取对方的资源，形成一个环的死锁情形。

数据库系统一般被设计成能够检测死锁并从死锁中恢复。一个事务的执行可能需要先获取很多锁，事务执行完毕之后再释放这些锁。因此，很可能出现两个事务相互等待获取对方持有的锁，而陷入死锁的情况。不过数据库管理系统不会允许这种情况的发生，当它发现一组事务陷入死锁之后，将会选一个牺牲者，终止该事务的执行。这将释放被牺牲的事务所持有的锁，以便其他事务能够运行下去，随后可以试着重新执行被取消的事务，这次这个事务可能就可以顺利执行了。

JVM 没有提供像数据库管理系统那样的死锁检查与恢复机制。当一组 Java 线程陷入死锁，那就 GameOver 了。唯一能够恢复的办法是终止程序，然后重新启动，然后希望这次运气好一点，死锁不要再发生了。

就像很多其他并发引起的风险一样，死锁可能不会立即表现出来。一个具有死锁的可能性的程序不是每次运行都会死锁。死锁问题一般在重载情况下才会发作。

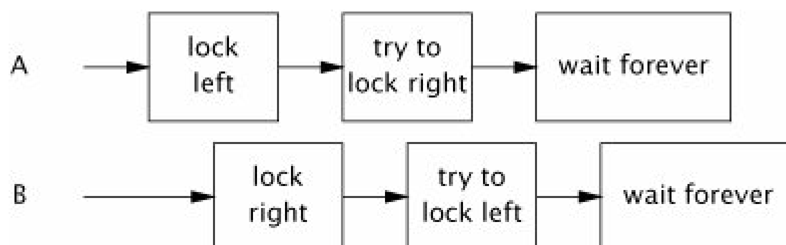
10.1.1. 锁序死锁

如下代码中的 LeftRightDeadLock 类可能会引起死锁问题：

```
public class LeftRightDeadlock
{
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight()
    {
        synchronized (left)
        {
            synchronized (right)
            {
                doSomething();
            }
        }
    }

    public void rightLeft()
    {
        synchronized (right)
        {
            synchronized (left)
            {
                doSomethingElse();
            }
        }
    }

    void doSomething()
    {
    }
```

```
    void doSomethingElse()
    {
    }
}
```

leftRight 方法和 rightLeft 方法都需要获取 left 锁和 right 锁，如果一个线程调用 leftRight 方法，另一个线程调用 rightLeft 方法，并且他们的操作产生了重叠，将会导致死锁，如下图所示：



本例中死锁的原因来自于两个线程试图以不同的顺序获取一组锁。如果它们以相同的顺序获取 left 锁和 right 锁，就不会出现环路，因此就不会有死锁的可能性。

如果所有线程都从全局的高度，以相同的顺序获取一组需要的锁就可避免锁序死锁的发生。

要验证锁序的一致性需要从全局来分析程序中所有的锁。如本例所示，只检查 leftRight 方法或只检查 rightLeft 方法是不够的，需要综合判断。

10.1.2. 动态锁序死锁

有时候你无法控制锁序，如下代码实现了在两个账户间转账的功能。在执行转账之前需要获取两个账户的锁，这样才能保证转账操作是原子的。

```
public void transferMoney(Account fromAccount,
                          Account toAccount, DollarAmount amount)
                          throws InsufficientFundsException
{
    synchronized (fromAccount)
    {
        synchronized (toAccount)
        {
            if (fromAccount.getBalance().compareTo(amount) < 0)
            {
                throw new InsufficientFundsException();
            }
```

```
            else
            {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```

在本例中锁序取决于传入的参数的次序，如果有两个线程同时以不同的账户顺序调用 transferMoney 的话将会出现死锁，例如：

A: transferMoney(myAccount, yourAccount, 10);

B: transferMoney(yourAccount, myAccount, 20);

像上例一样，寻找嵌套的锁获取结构就可以找到这类问题。但是如何解决这个问题呢？我们可以使用 System.indentityHashCode 方法获取两个账户对象的哈希码，总是先获取哈希码小的账户对象的锁，后获取哈希码大的账户对象的锁，如下代码使用了这种技术：

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct, final DollarAmount amount)
                          throws InsufficientFundsException
{
    //局部类
    class Helper
    {
        public void transfer() throws InsufficientFundsException
        {
            if (fromAcct.getBalance().compareTo(amount) < 0)
            {
                throw new InsufficientFundsException();
            }
            else
            {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }

    //获取两个账户对象的哈希码
```

```
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);
    //先获取哈希码较小的账户对象的锁
    if (fromHash < toHash)
    {
        synchronized (fromAcct)
        {
            synchronized (toAcct)
            {
                new Helper().transfer();
            }
        }
    }
    else if (fromHash > toHash)
    {
        synchronized (toAcct)
        {
            synchronized (fromAcct)
            {
                new Helper().transfer();
            }
        }
    }
    else
    {
        synchronized (tieLock)
        {
            synchronized (fromAcct)
            {
                synchronized (toAcct)
                {
                   new Helper().transfer();
                }
            }
        }
    }
}
```

如果两个账户对象的哈希码相同(虽然发生的可能性比较小),如何区分呢?这样的话死锁问题又回来了。为了避免这种问题，我们使用了第三方锁，在获取两个账户对象锁之前先获取第三方锁,这样就确保了一次只有一个线程尝试获取这两个哈希码相等的账户对象的锁。如果哈希冲突较多这种技术有可能造成瓶颈，不过一般情况下只要设计好 hashCode 方法就可避免这个问题。

如果每个账户对象由一个唯一的，immutable 的，可相互比较的键来标记，比如账户号，避免锁序死锁问题就更容易了，连第三方锁都不需要了，因为不会出现某个账户自己给自己转账的情况。

你可能认为我们夸大了死锁的危害，因为程序只是短暂地持有锁，很快就释放了，但是死锁问题是实际系统中很严重的问题。一个大型商业应用程序一天之内可能需要进行数十亿次的锁获取和锁释放操作。只要有一次死锁就完了，此外，再彻底的测试也无法发现所有隐藏着的死锁可能性。

10.1.3. 合作对象之间的死锁

多个锁获取引起的死锁问题往往并不像 10.1.1 节中的 LeftRightDeadLock 和 10.1.2 节中的 TransferMoney 展示的那么简单，获取锁的操作可能分布在多个方法之中。如下代码展示了这种情况：

```
/**
 * 出租车类
 */
public class Taxi
{
    @GuardedBy("this")
    private Point location;                    //当前位置

    @GuardedBy("this")
    private Point destination;             //目的地

    private final Dispatcher dispatcher;   //车辆分派器

    /**
     * 构造函数
     */
    public Taxi(Dispatcher dispatcher)
    {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation()
    {
        return location;
    }

    public synchronized void setLocation(Point location)
```

```
    {
        this.location = location;

        //到达目的地，通知分派器
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}

/**
 * 车辆分派器
 */
public class Dispatcher
{
    @GuardedBy("this")
    private final Set<Taxi> taxis;            //所有出租车

    @GuardedBy("this")
    private final Set<Taxi> availableTaxis;   //所有空闲的出租车

    /**
     * 构造函数
     */
    public Dispatcher()
    {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    /**
     * 某辆车到达目的地，将其加入到空闲出租车集合中
     */
    public synchronized void notifyAvailable(Taxi taxi)
    {
        availableTaxis.add(taxi);
    }

    /**
     * 获取一个图像，显示所有出租车的位置
     */
    public synchronized Image getImage()
    {
        Image image = new Image();
        for (Taxi t : taxis)
```

```
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

尽管没有哪个方法显式地获取两个锁，但是仍然存在死锁的可能性。如果一个线程调用 setLocation，它先获取 Taxi 对象的锁，然后试图获取 Dispatcher 对象的锁。如果另一个线程调用 getImage 方法，它先获取 Dispatcher 对象的锁然后试图获取每个 Taxi 对象的锁（一次一个，用完即释放）。这样就存在锁序死锁的可能性了。

这种锁序死锁的可能性比较难以发现，线索是在持有锁的情况下调用异形方法。

在持有锁的情况下调用异形方法很容易引起活跃性问题。一方面，异形方法可能需要获取其他锁，因而有可能形成死锁。此外异形方法也可能长期阻塞，导致该方法长期不能释放其持有的锁。

10.1.4. 开放调用

在持有锁的同时调用异形方法会导致程序很难分析，因此是非常危险的，因为方法调用本身就是一个抽象屏障，使得你不用去关心被调用的方法的细节。

在没有持有锁的情况下调用异形方法被称为开放调用。依赖于开放调用的类比在持有锁的情况下调用异形方法的类更可靠。使用开放调用来避免死锁可类比于使用封装来提供线程安全。虽然不用封装也可以构建线程安全的应用程序，但是使用封装来实现线程安全可使得安全性更加容易分析。

相似地，如果程序只依赖于开放调用的话，程序的活跃性分析也会更加容易。只使用开放调用使得很容易识别那些获取多个锁的代码路径，因此可以确保以全局一致性的顺序获取锁。

使用开放调用，我们可以很容易修改上一节中的 Taxi 和 Dispatcher 类来排除死锁风险。只要缩减 synchronized 块，只保护共享状态域就可以了。修改后的代码如下：

```
//in Taxi
public void setLocation(Point location)
{
    boolean reachedDestination;
```

```
    synchronized  (this)
    {
        this.location = location;
        reachedDestination = location.equals(destination);
    }

    if (reachedDestination)
    {
        dispatcher.notifyAvailable(this);
    }
}

//in Dispatcher
public Image getImage()
{
    Set<Taxi> copy;

    synchronized  (this)
    {
        copy = new HashSet<Taxi>(taxis);
    }

    Image image = new Image();
    for (Taxi t : copy)
        image.drawMarker(t.getLocation());
    return image;
}
```

编程的时候尽量使用开放调用。使用开放调用的程序其活跃性更容易分析，更容易避免死锁。

将synchronized方法缩减为使用synchronized块的方法有时会导致一些不好的后果。因为它将原本是原子的操作变成了非原子操作。在大多数情况下原子性的丢失是可以接受的。在本例中修改车辆位置和通知车辆分派器本出租车已经到达目的地不需要是原子的。

有时候原子性的丢失将会造成大问题。你必须使用其他技术来保持原子性。

10.1.5. 资源死锁

多个线程各自持有锁，并相互等待获取对方持有的锁将导致死锁问题。同样，多线程各自持有资源，并相互等待获取对方持有的资源也会造成死锁。

比方说你有两个不同数据库的数据库连接池。资源池一般使用信号量实现，

以便在池空的时候，获取资源操作会阻塞。假设线程 A 已经持有一个数据库 D1 的连接，然后试图获取一个数据库 D2 的连接。与此同时线程 B 已经持有一个数据库 D2 的连接，然后试图获取一个数据库 D1 的连接。这样就可能出现两个线程都阻塞等待获取对方持有的资源的情况。

解决办法是两个线程都以相同的顺序访问资源，比如它们都先试图获取一个数据库 D1 的连接，然后再获取一个数据库 D2 的连接。

另一种资源引起的死锁是线程饥饿死锁。比如，一个在单线程执行器中运行的任务向该任务执行器提交了另一个任务，并等待该任务返回执行结果（比如调用 Future.get 方法）。这样的话，第一个任务将会永远等待，任务执行器陷入停顿。

一个任务等待其他任务执行结果然后再继续执行是造成线程饥饿死锁的主要原因。

有界线程池不能支持非独立任务。

10.2. 死锁的避免与诊断

一个从没有一次获取超过一个锁的应用程序不用担心存在锁序死锁的问题。如果你必须一次获取多个锁，那么在设计阶段就应该考虑到锁序死锁的问题，试图最小化涉及到的锁的个数，并将你所采用的锁序协议记录下来。

如果一个应用程序使用了精细化的锁，请使用如下两阶段策略来检查代码是否存在死锁的可能性：首先找到那些一次获取多个锁的代码段，然后确保这些代码段以全局一致的顺序使用锁。此外，尽可能使用开放调用。

10.2.1. 限时获取锁

另一个死锁检测与恢复技术是使用显式锁 Lock 类的带时间期限的 tryLock 方法。当一个线程获取内部锁的时候，如果其他线程占有了这个内部锁一直不释放的话，本线程就会一直等待。显式锁 Lock 允许你指定一个时间期限，如果时间到期还没有获取到锁，tryLock 就会返回失败标识。

限时锁获取失败之后你不需要知道原因。可能是由于出现了死锁，也可能是某个已经获取了这个锁的线程进入了死循环，导致锁一直不能被释放出来，当然也可能是某个已经获取了这个锁的线程运行时间过长，超过了锁获取限时。不过

你仍有机会将锁获取失败事件记录下来，然后重新试图获取锁。

使用限时锁获取技术获取多个锁可以有效地预防死锁，即使这些锁的获取顺序不是全局一致的。如果一个锁获取操作超时，它可以释放所有已经获取的锁，稍等一下再次尝试获取这些锁，这样就有可能消除死锁条件，恢复程序的正常运行。关于显式锁的使用请参考第十三章。

10.2.2. 使用线程转储来分析死锁

虽然预防死锁是程序员的工作，但是当死锁已经发生的时候 JVM 可以使用线程转储技术帮助定位死锁。一个线程转储包括线程的堆栈路径和锁信息（线程持有哪些锁、这些锁都是在哪些栈帧中获取的、如果线程处于阻塞状态那么它在等待获取哪个锁？）。在产生线程转储之前，JVM 会搜索锁等待获取图，检查是否有环路，如果有环路说明有死锁发生，记录与死锁有关的线程和锁并在代码中定位它们。

你需要向 JVM 进程发送一个 SIGQUIT 信号来触发线程转储。在 Unix 平台上按下 Ctrl+\键或者在 Windows 平台上按下 Ctrl+Break 键都可以。很多 IDE 也支持线程转储。

如果你使用的是显式锁 Lock 而不是对象的内部锁的话，Java5 的线程转储技术不支持显式锁信息。Java6 的线程转储技术支持显式锁，不过定位信息没有内部锁准确。内部锁是与获取它的栈帧绑定的，显式锁只与获取它的线程绑定。

如下代码显示了一个商业 J2EE 应用程序的线程转储信息。死锁问题由三个部分引起：一个 J2EE 应用程序、一个 J2EE 容器和一个 JDBC 驱动器。这三个部分都是商业产品，都经过了详细的测试，他们都包含一些无害的小 Bug，但是相互作用起来就会产生致命的问题。

```
Found one Java-level deadlock:
=============================
"ApplicationServerThread":
  waiting to lock monitor 0x080f0cdc (a MumbleDBConnection),
  which is held by "ApplicationServerThread"
"ApplicationServerThread":
  waiting to lock monitor 0x080f0ed4 (a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread"

Java stack information for the threads listed above:
```

```
"ApplicationServerThread":
        at MumbleDBConnection.remove_statement
        - waiting to lock <0x650f7f30> (a MumbleDBConnection)
        at MumbleDBStatement.close
        - locked <0x6024ffb0> (a MumbleDBCallableStatement)
     ...

"ApplicationServerThread":
        at MumbleDBCallableStatement.sendBatch
        - waiting to lock <0x6024ffb0> (a MumbleDBCallableStatement)
        at MumbleDBConnection.commit
        - locked <0x650f7f30> (a MumbleDBConnection)
     ...
```

这里只显示了线程转储信息中与死锁相关的一部分。JVM 在死锁诊断方面为我们做了很多工作：哪些锁造成了问题、牵涉到哪些线程、这些线程持有了哪些锁等。从上述代码中可以看到一个线程持有了 MutableDBConnection 对象的锁，等待获取 MutableDbCallableStatement 对象的锁，另一个线程恰巧相反。

这里使用的 JDBC 驱动程序明显有一个锁序死锁 Bug：不同的调用链以不同的顺序获取多个锁。这里还有另外一个 Bug：一个数据库连接被多个线程并发共享。JDBC 规格说明中没有说数据库连接是线程安全的，一般来说应该将其封闭在单线程中使用。本例本来想使用锁来将这些非线程安全对象转换为线程安全的，但是由于没有考虑锁序的问题，因此造成了死锁。

10.3. 其他活跃性风险

除了最常见的死锁问题，在并发编程中还有其他几种活跃性风险：饥饿、丢失信号和活锁（丢失信号问题在 14.2.3 节再讲）。

10.3.1. 饥饿

如果一个线程需要获取某个资源，然后才能继续执行下去，但是一直获取不到这个资源，这就引发了饥饿问题。在 Java 应用程序中，线程优先权设置得不合理可能引起饥饿问题（优先权低的线程长期得不到执行）。一个持有锁的线程陷入一个无法终止的结构（比如死循环）长期无法释放这个锁也会造成其他想要获取这个锁的线程处于饥饿状态。

Java 应用程序中线程优先权分十级，在适当的时候 JVM 可将这十级优先权映射为操作系统提供的调度优先权。在不同的平台上这种映射是不一样的。两个

不同的 Java 线程优先权可能映射到操作系统上同一个调度优先权，也可能映射到操作系统上两个不同的调度优先权。

操作系统调度器尽全力提供公平的和活跃的调度功能。大多数 Java 应用程序将所有线程都声明为相同的优先权：Thread.NORM_PRIORITY。线程优先权机制比较迟钝，有时候改变优先权没多大效果。

尽量不要修改线程的优先权，一旦你开始修改线程的优先权，你的程序就不是跨平台的了，且引入了饥饿的风险。你可以在一些程序中看到 Thread.sleep 和 Thread.yield 等调用，它们是为了给低优先权的线程一些运行的机会。

不要调整线程的优先权，使用默认的优先权就很好了。

10.3.2. 响应速度

我们在第九章中开发了一个框架来向背景线程提交耗时任务，同时避免冻结用户界面。尽管如此，CPU 密集型的背景任务仍然可能影响程序的响应速度，因为它们会影响事件分派线程的执行。这个时候你需要降低 CPU 密集型背景线程的优先级（我知道这违反了上一节中提出的规则，但是没有办法）。

程序的响应能力差也可能是由糟糕的锁管理机制造成的。如果一个线程长期持有一个锁（可能在持有锁的情况下遍历大型集合，为每个集合元素做大量工作），其他需要使用这个锁的线程就必须等很长的时间，从而导致程序响应速度慢。

10.3.3. 活锁

活锁也是一种活跃性问题。一个未阻塞的线程反复尝试一个无法成功的操作，无法继续进行下去，将会导致活锁。例如一个线程处理一个任务，该任务执行失败，在异常处理器中将该任务重新提交到任务队列的队首，这样的话将陷入一个死循环。

对于活锁有个比喻，两个过度有礼貌的人在走廊里面对面行走，他们都给对方让道，结果又都挡了对方的道，然后再让道……。如下代码说明了这个过程：

```
public class Person implements Runnable
{
    private String name;            //Person名称
    private Person friend;          //该Person的鞠躬对象
    private boolean bow = true;     //是否在鞠躬的状态标识，默认为true
```

```
/**
 * 构造函数
 */
public Person(String n)
{
    this.name=n;
}

/**
 * 鞠躬
 */
public void bow()
{
    bow = true;
    System.out.println(name+" bowed");
}

/**
 * 起身
 */
public void up()
{
    bow = false;
    System.out.println(name+" uped");
}

/**
 * 设置本Person的鞠躬对象
 */
public void setFriend(Person friend)
{
    this.friend = friend;
}

/**
 * 任务
 */
public void run()
{
    while (friend.bow)
    {
        this.bow();
        try
```

```
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            this.up();
        }
    }

    public static void main(String[] args) throws InterruptedException
    {
        Person p1 = new Person("P1");
        Person p2 = new Person("P2");
        p1.friend = p2;
        p2.friend = p1;

        new Thread(p1).start();
        Thread.sleep(100);
        new Thread(p2).start();
    }
}
```

10.4. 本章小结

活跃性风险是很严重的问题。最常见的活跃性风险是锁序死锁。为了避免锁序死锁，需要以全局一致的顺序获取多个锁。尽量使用开放调用，这样可以极大地减少一次获取多个锁的情况，并且方便找到那些一次获取多个锁的代码块的位置。

第十一章 性能和可伸缩性

使用多线程的一个主要目的是提升性能。使用多线程可以使应用程序更有效地利用处理器资源并通过并发运行多个任务提高程序的响应能力。

本章讨论了用于分析、监控和提升并发程序性能的一些技术。不幸的是，很多能够提升性能的技术同时也增加了复杂性，因此增加了活跃性风险和安全性问题。更糟糕的是，有些技术虽然能够提升性能但会降低开发效率和系统可维护性，或者以牺牲一方面性能为代价，提升另一方面的性能。

需要记住的是安全性总是比性能更重要。首先你必须保证程序是正确的，然后再试着使它运行得更快，况且实际开发中一般没有必要将性能提升到极致。

11.1. 考虑性能

提升性能意味着用更少的资源做更多的事情。对于特定的任务最紧缺的资源可以是 CPU 周期、内存空间、网络带宽、I/O 带宽、数据库请求、磁盘空间或者其他一些资源。

尽管使用多线程的目的是为了提升应用程序的性能，但是多线程的使用总是需要一些额外的开销。线程的一些附带信息、线程的创建与销毁、上下文切换等都需要消耗系统资源。

如果程序是计算密集的，我们增加更多的 CPU 有助于提升程序的运行速度。如果程序不是计算密集的，CPU 没有充分忙碌，则添加更多的 CPU 没有什么效果。多线程可以将应用程序拆解成多个任务，分配给多个处理器，让它们都有活干，从而提升应用程序的运行速度。

11.1.1. 性能和可伸缩性

应用程序的性能可以用很多方法进行衡量，比如服务时间、延迟、吞吐量、可伸缩性。这些指标中有些（服务时间、延迟）是用来标识一个任务处理速度的，有些（吞吐量、可伸缩性）是用来标识系统在现有的资源情况下能处理多少任务的。

可伸缩性用来描述当为系统添加额外的资源（CPU、内存条、硬盘、I/O 带宽）之后系统吞吐量的提升能力。

设计和调优并发程序以提供更高的可伸缩性与传统的程序性能优化有所不

同。传统的程序优化可能使用缓存来优化响应速度或者使用一个 n*log(n)级别的算法来替换一个 n*n 级别的算法。可伸缩性的调优工作一般是寻找可分解的并行性。

为了提升整个应用程序的可伸缩性或者为了增加硬件利用率，我们往往需要增加每个任务的工作量。需要注意的是，很多在单线程环境下用于提升程序性能的小技巧对提升可伸缩性是不利的。

我们熟悉的三层架构：将应用程序分成展示层、商业逻辑层和数据持久层的做法告诉我们可伸缩性的增加往往以性能的牺牲为代价。如果将程序设计成铁板一块，将能提供更好的性能（响应速度更快），因为这样不会牵涉到网络延迟，也不需要在多层之间传递任务。

然而，当铁板一块的系统的处理能力饱和之后我们就会遇到严重的问题：很难大幅度地提升系统的处理能力。因此我们采用三层架构，尽管每个任务消耗了更多的计算资源，但是我们可以通过扩展系统大幅提升处理能力。

对于服务器应用程序来说吞吐量比响应速度更重要，对于桌面交互式应用程序来说响应速度更重要。本章主要讨论可伸缩性。

11.1.2. 性能折中

优化的过程中往往需要用一种代价换另一种代价，比如用额外的内存资源换更短的服务时间。很多性能优化是以牺牲可读性和可维护性为代价的。有时候还要牺牲良好的面向对象设计原则，比如打破封装。

在认定一种设计比另一种设计更快之前先考虑如下几个问题：

- “更快”到底是什么意思？
- 在什么情况下这种设计才表现出更快的速度？轻载还是重载？大数据集还是小数据集？
- 这些情况会经常出现吗？
- 这个设计会以牺牲什么为代价？这么做值得吗？

对于优化，我们为什么要给出这么保守的建议呢？原因是优化往往是并发程序中 Bug 的最大来源。经常有人认为同步导致程序运行速度慢，然后就设计一些看似聪明但十分危险的习惯用法来削减同步，比如使用双检查锁。由于并发 Bug 很难定位和排除，因此对于所有可能引起并发 Bug 的因素都应该慎重考虑。

更糟的是，如果你以牺牲安全性为代价来提升性能，结果将是两者都得不到。在并发编程中，开发者的直觉（性能瓶颈的位置、提升运行速度或可伸缩性的方法）往往是错误的。在调优的前后你需要比较程序的运行效果，看是否达真的到了调优的目的。

尽量不要以牺牲安全性和可维护性为性能优化的代价。

市面上有很多性能分析工具来测量和追踪性能瓶颈。例如免费软件 PerfBar 可以告诉你各个 CPU 有多繁忙，这样你就知道是否需要调优或者调优有没有效果了。

11.2. 阿姆达尔定律

有些问题如果有更多的资源就可以解决得更快，比如收割玉米，工人越多，收割得越快。但有些任务是线性的，增加资源也没用。

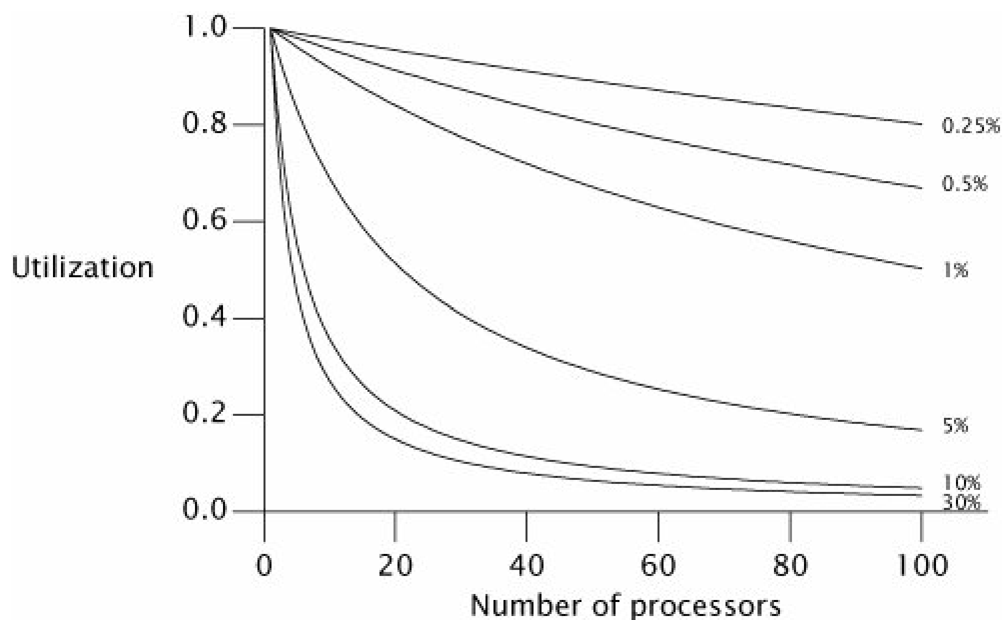
一般来说并发程序既包含很多可并行化处理的任务也包含一些必须线性化处理的任务。阿姆达尔定律描述了一个并发应用程序在理论上最大可加速到什么程度。如果 F 表示必须顺序化处理的任务量占整个任务量的比例，N 表示机器中 CPU 的数量，我们最大可以将程序速度提升到单线程的 S 倍。有如下公式：

$$S \le \frac{1}{F + \frac{(1-F)}{N}}$$

当 N 趋近于无穷大，S 收敛于 1/F。如果一个程序中有一半工作量是必须顺序化处理的，那么使用并发的话最大可加速两倍。如果一个程序中有十分之一的工作量必须是线性化处理的，那么使用并发的话最大可加速十倍。

如果程序中有十分之一工作量必须是顺序化处理的，且 N=10，可算出 S<=5.3（CPU 利用率为 53%）。如果 N=100，可算出 S<=9.2（CPU 利用率为 9%）。

下图显示了最大 CPU 利用率随 CPU 数目变化曲线图（每条曲线代表不同的 F 取值）。



为了确定最大可加速到什么程度，你需要那些找到必须顺序化处理的任务。对于如下程序：

```
public class WorkerThread extends Thread
{
    private final BlockingQueue<Runnable> queue;

    public WorkerThread(BlockingQueue<Runnable> queue)
    {
        this.queue = queue;
    }

    public void run()
    {
        while (true)
        {
            try
            {
                Runnable task = queue.take();
                task.run();
            }
            catch (InterruptedException e)
            {
                break; /* Allow thread to exit */
            }
        }
    }
}
```

想象一下有 N 个线程，不断从共享队列中取任务然后执行。假设各个任务是相互独立的。如果我们增加 CPU，这个程序能加速到什么程度呢？乍一看，程序是完全可并行的，CPU 越多就会有越多的任务被并发处理。然而，这里有个必须顺序化处理的任务：从共享队列中取任务。共享队列被多个线程访问，take 方法必须采用同步机制（比如锁）来维护数据完整性。

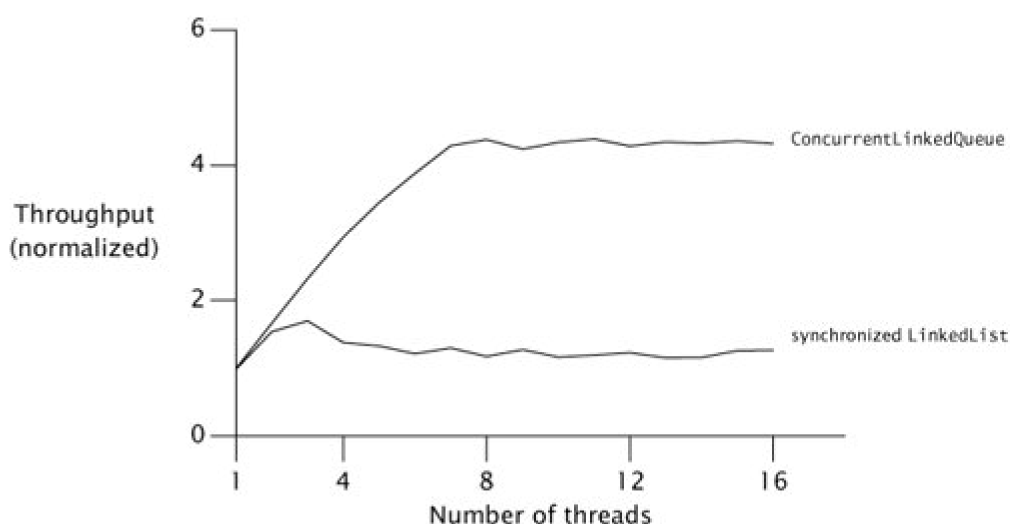
单个任务的处理时间不仅包括执行 task.run 所消耗的时间，也包括执行 queue.take 所消耗的时间。访问共享数据结构不可避免地会向程序中引入顺序化操作。

这里我们还忽略了一个顺序化操作来源：结果处理。task.run 不可避免地会产生一些副作用，比如将计算结果写入日志文件或者将计算结果放在某个数据结构（结果容器）中。日志文件和结果容器往往是多线程共享的，因此也必须使用同步，这就引入了顺序化操作。

所有的并发程序都包含一些必须顺序化处理的操作，没有例外。

11.2.1. 示例：框架中隐藏的顺序化操作

为了展示必须线性化处理的操作如何隐藏在应用程序中，我们可以观察随着线程数量的增加，吞吐量的变化情况。下图显示了一个简单的应用程序，多线程不断地从一个共享队列中取元素然后处理它，各个线程是相互独立的。如果有个线程发现队列为空，它会将一批元素入队。访问共享队列的操作引入了顺序化，但处理各个元素的操作完全是可并行化的，没有牵涉到共享数据。



上图测试了两个线程安全队列实现：一个用 synchronizedList 方法包装的

LinkedList 和一个 ConcurrentLinkedQueue。我们看到，仅仅是更换了共享队列的具体实现就可以对程序的可伸缩性产生很大的影响。

使用 ConcurrentLinkedQueue 的程序的吞吐量不断增加直到达到 CPU 数量，然后基本保持不变。LinkedList 当线程数量达到三之后不但吞吐量没有上升反而有所下降。这是同步的额外开销造成的。

不同的吞吐量变化曲线来自于两者不同的顺序化操作量。同步的 LinkedList 使用一个锁来保护整个队列，ConcurrentLinkedQueue 使用一个复杂的非阻塞队列算法。对于前者来说所有插入或删除操作都必须顺序化处理，对于后者只有更新队列内部的指针的操作才必须顺序化处理。

11.2.2. 定性地应用阿姆达尔定律

阿姆达尔定律定量地描述了如果我们能够精确地估计必须顺序化处理的操作占所有操作比例的话，增加更多计算资源能使程序运行速度增加多少倍。尽管直接测量这个比例比较困难，但是阿姆达尔定律还是非常有用的。

虽然目前很多机器只有 2 个或 4 个 CPU，但是随着多核技术逐渐成为主流，很快，很多服务器将拥有成百上千个处理器。在四个 CPU 上伸缩性良好的算法如果移植到拥有几百个 CPU 的机器上的话，其中隐藏的瓶颈就会显露出来。

当你评估一个并行算法的时候，考虑如果在具有成百上千个 CPU 的系统上运行的话将会发生什么，这样的话你就可能知道可伸缩性瓶颈在哪儿。我们在 11.4.2 节和 11.4.3 节将会讨论两种技术来减少锁的粒度从而增加系统的可伸缩性。不过，性能优化必须基于实际的性能需求，如果没有必要，就不要优化了。

11.3. 线程引起的开销

单线程程序不会导致线程调度和同步方面的开销，也不需要使用锁来保护共享数据结构的一致性。线程调度和线程间通信会导致性能方面的开销，如果采用多线程的话，并行化所带来的性能提升应该超过所带来的开销。

11.3.1. 上下文切换

如果线程数超过 CPU 数，操作系统最终一定会挂起某个线程，让其他线程运行。这就是上下文切换，需要保存当前正在运行的线程的上下文信息并恢复新调度的线程的上下文信息。

上下文切换是需要代价的，线程调度需要访问操作系统和 JVM 中的共享数据结构。OS 和 JVM 与你的应用程序使用同一组 CPU，如果 OS 和 JVM 消耗了更多的 CPU 周期的话，留给你的应用程序的 CPU 周期就少了。不过 OS 和 JVM 活动不是上下文切换引起的唯一开销。当一个新的线程被调度时，它需要的数据不太可能存在于处理器缓存中。因此调度器会给每个线程一个最小运行时间，即使此时还有很多线程等待被执行。这样做可以增加整个程序的吞吐量，线程切换得太频繁会增大开销。

当一个线程因为等待获取某个锁而阻塞的时候，JVM 将会暂停这个线程的执行，允许其被切换出去。如果线程经常阻塞，它们将不能完全使用最小运行时间。经常阻塞的线程将会增加调度开销并降低吞吐量。

根据程序运行的平台不同，上下文切换的实际开销也不同。在大多数平台上上下文切换一般要花费 5000 到 10000 个 CPU 时钟周期，或者几微秒时间。

Unix 系统中的 vmstat 命令和 Windows 系统中的 perfmon 工具可以报告上下文切换的数量和内核的 CPU 消耗率。高内核消耗百分比（超过 10%）表示线程调度比较频繁，这可能是由于 I/O 阻塞或者获取锁阻塞造成的。

11.3.2. 内存同步

同步引起的性能开销可能来自于几个方面。synchronized 和 volatile 可能导致代码编译后包含一些特殊的指令，有些信息不存储在 CPU 缓存中，此外，也可能阻止编译器优化，从而影响程序的性能。

当考察同步引起的性能下降问题时，区分竞争性同步和非竞争性同步很重要。JVM 可以将那些不被竞争的代码块的锁除去，以便减小同步带来的开销。如果某个锁只在一个线程中可能被获取，JVM 可以将这个锁除去，因为不可能有其他线程使用这个锁来同步。例如对于如下代码中的同步机制，JVM 总是会将获取锁的部分除去。

```
synchronized (new Object())
{
    // do something
}
```

更加复杂的 JVM 可以使用逃逸分析来识别那些生存周期只存在于栈中的局部对象，这些对象是线程封闭的。如下代码中的 stooges 对象只存在于栈中，因

而是线程封闭的。执行一次 getStoogeNames 方法需要获取然后释放 Vector 上的锁四次（三次 add，一次 toString）。然而，一个聪明的运行时编译器可以识别出 stooges 是线程封闭的，对其加锁没有必要，因而会除去这些加锁和释放锁的步骤。

```
public String getStoogeNames()
{
    List<String> stooges = new Vector<String>();
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly ");
    return stooges.toString();
}
```

即使不使用逃逸分析，编译器也可以将上述四次获取然后释放锁的步骤合并为一次获取然后释放锁的操作。这样即较少了同步开销也让优化器有了一个更大的代码块可做优化。

不用太担心非竞争性同步带来的开销。基本的机制已经够快的了，此外 JVM 还会做一些额外的优化，进一步缩减开销。你应该把精力集中在竞争性同步所造成的开销上面。

11.3.3. 阻塞

非竞争性同步可以完全由 JVM 来处理。竞争性同步则必须得到 OS 的支持，因而增加了开销。如果多线程竞争一个锁，当前没有获取到锁的线程将会阻塞。JVM 可以使用轮询（不断尝试获取锁，直到成功为止）来实现阻塞也可以通过操作系统来挂起被阻塞的线程。对于短期等待来说轮询法比较好，对于长期等待来说挂起法比较好。有些 JVM 通过对历史数据的统计分析来决定使用哪一种方法，但大多数 JVM 直接选择后者。

获取锁导致的线程阻塞、条件等待导致的线程阻塞或者 I/O 导致的线程阻塞都会导致线程被挂起。挂起会导致两个线程上下文切换和一些缓存清空操作：阻塞的线程会在时间配额到期之前被切换出去，当锁可用或者条件满足时该线程还会被切换回来。当线程释放锁的时候它会通知 OS，让其恢复正在等待这个锁的其他线程。

11.4. 减少锁竞争

我们知道必须顺序化处理的操作会损害程序的可伸缩性，上下文切换会损害程序的性能。竞争性的锁会同时引起这两个不利因素，因此减少锁竞争可以同时提升可伸缩性与性能。

访问被排他性锁保护的资源是一种顺序化操作，同时最多这能有一个线程能够访问这个资源。当然，我们使用锁是有原因的，比如防止多线程破坏数据一致性。不过这么做是要付出代价的，锁竞争会限制程序的可伸缩性。

在并发编程中排他性资源锁会影响程序的可伸缩性。

有两个因素能够影响锁的竞争：该锁被获取的频率和获取保持时间。如果这两个因素的乘积比较小，竞争就不会太激烈，就不会造成太大的可伸缩性问题。然而，如果乘积比较大，竞争就会比较激烈，甚至造成只有少数线程在运行，其他线程都在等待获取锁的情况，这样就损害了可伸缩性。

有三种方式来减少锁的竞争：

- 减少锁的获取保持时间
- 减少获取锁的频率
- 使用协调机制来替换排他锁，这样可以提供更高的并发性

11.4.1. 减少锁的作用范围

一个减少锁竞争的有效方式是尽量减少锁的作用范围。可以将不需要锁保护的代码移到 synchronized 块外面，尤其是对于比较耗时的操作和可能阻塞的操作（比如 I/O 操作）。

如下代码中的 AttributeStore 类演示了锁的作用范围过大的情况。userLocationMatches 方法在 Map 中查找用户的位置，然后使用正则表达式来判断位置是否匹配。整个 userLocationMatches 方法都用 synchronized 关键字修饰，但是只有 Map.get 操作才真正需要锁保护。

```
@ThreadSafe
public class AttributeStore
{
    @GuardedBy("this")
    private final Map<String, String> attributes =
                                        new HashMap<String, String>();
```

```
    public synchronized boolean userLocationMatches(String name,
                                                    String regexp)
    {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

如下代码中的BetterAttributeStore类重写了userLocationMatches方法，缩减了锁保护块的大小。由于构造键值字符串和正则匹配操作都不需要访问共享状态域，因此它们不需要锁的保护。

```
@ThreadSafe
public class BetterAttributeStore
{
    @GuardedBy("this")
    private final Map<String, String> attributes =
                                        new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp)
    {
        String key = "users." + name + ".location";
        String location;

        synchronized (this)
        {
            location = attributes.get(key);
        }

        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

由于AttributeStore类只有一个状态域：attribute，你一可以使用线程安全性代理技术，用一个线程安全的Map（比如HashTable、synchronizedMap包装集合或ConcurrentHashMap）来声明attribute。这样的话，synchronized关

键字就彻底不需要了。

虽然缩减 synchronized 的作用范围可以提高程序的可伸缩性，但是有些 synchronized 块中的操作要求必须是原子的，比如更新一组必须符合某个一致性约束的变量。这是需要考虑的问题。

一般来说，如果 synchronized 块中包含不需要锁保护的耗时操作或可阻塞操作的话，可以将它们移出 synchronized 块。

11.4.2. 减少锁的粒度

另一个减少锁竞争的方法是尽量减少线程获取这个锁的频率。可以使用锁分离和锁剥离技术来实现。可以将之前一个锁保护的代码块分解为由多个锁分别保护的不同共享状态域。

考虑一下，如果整个应用程序只有一个锁会怎么样，这样的话将导致很多线程竞争一个共享的全局锁。这样的话，两个线程同时尝试获取共享锁的可能性就大大增加了，从而造成更多的锁竞争。如果每个对象使用不同锁的话，锁竞争就会大大减少，应用程序可伸缩性就大大增强了。

如果一个锁保护多个相互独立的状态域的话，你可以将其分离成多个锁，分别保护不同的状态域，这样的话，每个锁被请求的几率就会减少。

如下代码中的 ServerStatus 类是数据库服务器的监听器的一部分，它维护了一组当前已登录的用户和一组正在执行的查询。这两个状态域是相互独立的，因而没有必要被同一个锁保护。

```
@ThreadSafe
public class ServerStatusBeforeSplit
{
    @GuardedBy("this")
    public final Set<String> users;
    @GuardedBy("this")
    public final Set<String> queries;

    public ServerStatusBeforeSplit()
    {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }

    public synchronized void addUser(String u)
```

```
    {
        users.add(u);
    }

    public synchronized void addQuery(String q)
    {
        queries.add(q);
    }

    public synchronized void removeUser(String u)
    {
        users.remove(u);
    }

    public synchronized void removeQuery(String q)
    {
        queries.remove(q);
    }
}
```

我们可以使用两个不同的锁来分别保护两个状态域，代码如下：

```
@ThreadSafe
public class ServerStatusAfterSplit
{
    @GuardedBy("users")
    public final Set<String> users;
    @GuardedBy("queries")
    public final Set<String> queries;

    public ServerStatusAfterSplit()
    {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }

    public void addUser(String u)
    {
        synchronized (users)
        {
            users.add(u);
        }
    }

    public void addQuery(String q)
```

```
    {
        synchronized (queries)
        {
            queries.add(q);
        }
    }

    public void removeUser(String u)
    {
        synchronized (users)
        {
            users.remove(u);
        }
    }

    public void removeQuery(String q)
    {
        synchronized (queries)
        {
            queries.remove(q);
        }
    }
}
```

通过分离锁，每个状态域都有专门的锁来保护，因此减少了锁竞争。

11.4.3. 锁剥离技术

如果出现了中等程度的锁竞争，上一节所讨论的锁分离技术可以起到较好的效果。但是如果出现了较严重的锁竞争，锁分离可能将一个竞争严重的锁分离成两个竞争严重的锁。这样做仍然不能大幅度地提升应用程序的可伸缩性，特别是当系统中有很多 CPU 的时候。

锁分离技术可以扩展为锁剥离技术。例如 ConcurrentHashMap 类的实现就使用了锁剥离技术，它使用了 16 个锁，分别保护 16 个哈希桶，这样基本上将锁竞争削减为原来的 16 分之一。

如下代码中的 StripedMap 类演示了一个基于锁剥离技术的哈希 Map 的实现。一共有 N_LOCKS 个锁，每个锁保护一组桶。大多数方法，比如 get，只需要获取单个桶的锁。有些方法需要获取所有的锁，比如 clear 方法(这里可以异步获取)。

```
@ThreadSafe
public class StripedMap
```

```
{
    // Synchronization policy: buckets[n] guarded by locks[n%N_LOCKS]
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    private static class Node
    {
        Node next;
        Object key;
        Object value;
    }

    public StripedMap(int numBuckets)
    {
        buckets = new Node[numBuckets];

        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++)
            locks[i] = new Object();
    }

    private final int hash(Object key)
    {
        return Math.abs(key.hashCode() % buckets.length);
    }

    public Object get(Object key)
    {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS])
        {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }

    public void clear()
    {
        for (int i = 0; i < buckets.length; i++)
        {
            synchronized (locks[i % N_LOCKS])
```

```
                {
                    buckets[i] = null;
                }
            }
        }
}
```

11.4.4. 避免过热的状态域

锁分离和锁剥离技术可以提升应用程序的可伸缩性，因为他们将不同的状态域（或者同一个状态域的不同部分）分开保护。

当你实现 HashMap 的时候，你如何实现 size 方法呢？最简单的办法是逐个数 Map 中键值对的个数。这样太过耗时，一个优化的方法是额外维护一个状态域，用于记录键值对的数目，每次调用 put 方法和 remove 方法的时候都会更新这个状态域。这种优化带来的问题是每个修改 Map 的操作都必须更新这个记录键值对数目的状态域。由于这个状态域也需要锁的保护，这就造成了锁竞争，使得这个状态域成为过热的状态域。

ConcurrentHashMap 的实现方法是为每个剥离锁提供一个键值对数量状态域（该状态域由对应的剥离锁保护），计算 size 的时候逐个获取各个剥离锁，然后将它们的键值对数目加起来。

11.4.5. 排他锁的替代品

消除锁竞争的第三个技术是放弃使用排他锁，使用一些有利于并发的共享状态域管理机制。这包括使用并发集合、读写锁、immutable 对象和原子变量。

ReadWriteLock 允许多个读线程，一个写线程。多个读线程可以并发地访问共享资源，只要他们不需要修改共享资源就可以了。写线程要访问共享资源必须先获取锁。对于那些读操作比较多，写操作比较少的数据结构，相比于排他锁，ReadWriteLock 能提供更好的并发性。关于读写锁后面章节还会详述。

对于 immutable 对象，由于不能对其进行修改，可以完全不使用锁。

原子变量提供了一种更新过热状态域的方法。原子变量类提供了非常精细化的、支持可伸缩性的原子操作，它们使用低层并发机制实现。如果你的类中有为数不多的过热状态域，且他们没有与其他状态域组成某个一致性约束的话，使用原子变量来声明他们可以提升应用程序的可伸缩性。

11.4.6. 监视 CPU 利用率

在测试应用程序的可伸缩性的时候，我们的目标往往是使处理器能够被充分利用。Unix 系统中的 vmstat 和 mpstat 以及 Windows 系统中的 perfmon 等工具软件都可以告诉你各个 CPU 的运行情况。

如果 CPU 被非对称地利用的话，你的首要目标应该是寻找应用程序中更多可分解的并行性。CPU 非对称使用暗示着程序中大多数任务集中在少数几个线程中运行，你的应用程序不能有效地利用额外的处理器。

如果不能完全利用所有的 CPU，你需要找出原因，一般有如下几个原因：

- **负载不足。**可能你的应用程序处于轻负载状态，没有多少任务要处理。你可以增加负载，然后测试响应速度、CPU 利用率等指标。
- **I/O 受限。**你可以使用 iostat 或者 perfmon 测试下你的应用程序是否需要经常访问磁盘。或者使用网络监视工具测试你的应用程序是否需要消耗很多网络带宽。
- **外部服务受限。**如果你的应用程序依赖于外部服务，比如数据库或者 Web Service，那么瓶颈也许不在你的代码中，而存在于外部服务中。你可以使用性能分析工具或者数据库管理器来查看外部服务返回结果的平均耗时。
- **锁竞争。**性能分析工具可以告诉你，你的应用程序中有多少个锁竞争，热点在哪儿？你也可以使用线程转储功能来查看锁竞争情况，如果一个线程由于等待某个锁而阻塞，线程转储信息中会显示“waiting to lock monitor ...”。竞争严重的锁常常会出现在线程转储信息中。

如果你的应用程序能够让 CPU 足够地忙碌，你可以使用监视工具来推断它是否能够从提供更多的 CPU 中获益。一个拥有四个线程的应用程序能够使得拥有四个 CPU 的系统充分忙碌，但是如果增加到八个 CPU，就无法得到较大的性能提升。vmstat 提供的信息中有一列显示的是当前可以运行但由于 CPU 数量不够而没有运行的线程的数量。如果 CPU 利用率很高，并且总是有可运行的线程在等待获取 CPU 调度的话，增加 CPU 将会提供应用程序的性能。

11.4.7. 不要使用对象池

在早期的 JVM 版本中，对象分配和垃圾回收都很慢，不过现在这些情况都有了很大的转变。事实上，目前 Java 中对象分配速度比 C 语言中 malloc 函数更快。

有的开发者喜欢使用对象池，对象是被循环利用的，而不是使用完之后自动被垃圾回收，下次再用则重新创建。然而，对象池会造成性能损失。

当线程创建一个新的对象的时候，很少需要进行线程间的协调操作，但是如果要利用对象池中的对象的话就必须使用同步机制。这样，获取对象池中对象的操作就有可能阻塞，由于锁竞争造成的线程阻塞远比对象创建昂贵，因此将会严重影响程序的运行速度和可伸缩性。对象池有它的作用，但是用于优化性能的话一般来说是不行的。

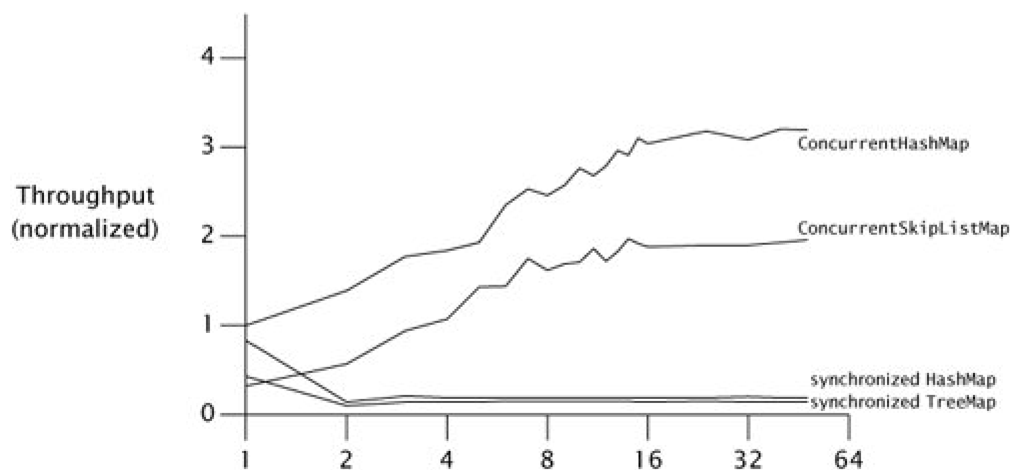
创建新对象总比使用同步机制获取对象池中的对象便宜。

11.5. 示例：比较不同 Map 实现的性能

在单线程应用中ConcurrentHashMap的性能只比synchronizedMap方法包装的 Map 好一点，但是在并发编程环境中 ConcurrentHashMap 比 synchronizedMap 方法包装的 Map 的性能好得多。ConcurrentHashMap 适合于用在那些大多数操作都是从一个 Map 中获取已存在的键值对的环境中。

SynchronizedMap 方法包装的 Map 的主要伸缩性障碍是整个对象只有一个锁，因此一次只能有一个线程访问它。另一方面 ConcurrentHashMap 的大多数读操作都不使用锁，对于某些需要使用锁的读操作和写操作，ConcurrentHashMap 使用了锁剥离技术。因此，可以同时有多个线程并发访问 ConcurrentHashMap。

下图演示了几个不同 Map 实现（ConcurrentHashMap、被 SynchronizedMap 方法包装的 HashMap 和 TreeMap、ConcurrentSkipListMap）在可伸缩性方面的差异。ConcurrentHashMap 和 ConcurrentSkipListMap 本身就是线程安全的，HashMap 和 TreeMap 不是线程安全的，但是可以被包装为线程全的。现在做如下测试：有 N 个线程并发执行一个循环，在循环中每个线程随机选择一个键，然后试着从 Map 中获取与该键对应的值，如果找不到该键则以 60%的可能性将该键值对插入到 Map 中。如果能找到该键则以 2%的可能性删除相应的键值对。



从图中可以看出，随着线程数量的增长，ConcurrentHashMap 和 ConcurrentSkipListMap 的吞吐量持续增长。尽管图中线程数量不是很大，但是该测试程序比真实程序竞争性要强，因为在测试程序中每个线程除了访问 Map 之外几乎没做什么其他工作。真实程序中每个线程除了访问 Map 之外还要做很多额外的工作。

SynchronizedMap 方法包装的 HashMap 和 TreeMap 的测试数据令人沮丧。当线程数大于或等于二的时候，吞吐量下降很厉害，且随着线程数的增加，吞吐量一直保持在较低水平。这是因为随着线程数量增多，锁竞争将会加剧，线程上下文切换和调度延迟将会消耗大量的系统资源。

11.6. 减少上下文切换造成的开销

很多任务包含可阻塞操作。从运行状态转换到阻塞状态将引起上下文切换。为了演示上下文切换如何影响吞吐量，我们分析两种不同的日志记录方式。

一种日志记录方式是调用 println 方法，另一种是像 7.2.1 节的 LogWriter 一样，使用一个专用的背景线程来记录日志消息。从开发者的角度看，这两种方式区别不大，但是对性能的影响是不同的。这取决于日志消息量、同时记录日志的线程数量以及一些其它因素，比如上下文切换。

如果 I/O 操作阻塞，操作系统将会阻塞日志记录线程直到 I/O 操作完成。如果有多个线程同时记录日志信息，它们将会竞争输出流的锁，第一种日志记录方式会导致多个线程竞争输出流的锁，这将引起额外的上下文切换，因此需要消耗更多的时间，造成服务器程序服务时间的增加。

服务时间的增加对系统的性能是不利的。更长的服务时间意味着客户必须等待更长的时间才能得到服务器的响应。将 I/O 操作移出客户端请求响应线程可以大大缩减服务时间。调用 log 方法的线程不会阻塞等待获取输出流的锁，或者等待 I/O 操作完成。只需要将日志消息入队，log 方法就可以返回。虽然入队操作 put 也会引起多锁竞争，但是这个操作是轻量的，很快就返回，因此 put 方法调用在队列未满的情况下一般不会阻塞。由于客户端请求处理线程不大可能阻塞，因此上下文切换也不太可能发生。因此，第二种日志记录方式性能更为优越。

从某种程度上讲，我们仅仅是将所有日志记录工作移到了一个专用的线程中。我们消除了多线程对输出流的竞争，并且客户端请求响应线程不会阻塞，这样就增强了整个程序的吞吐量。

11.7. 本章小结

由于使用多线程的一个最主要目的是有效利用多处理的处理能力。在讨论并发程序的性能时，我们往往更关心吞吐量和可伸缩性而不是服务时间。阿姆达尔定律告诉我们应用程序的可伸缩性受到必须顺序执行的操作的限制。由于 Java 应用程序中必须顺序执行的操作的主要来源是排他资源锁，因此尽量减少持有锁的时间或者减少锁的粒度可以提升系统的可伸缩性。

第十二章 测试并发程序

并发程序与单线程程序的区别在于，并发程序包含不确定性，增加了潜在的出错可能性。用于测试单线程程序正确性和性能的技术也可用于测试并发程序，不过并发程序的错误空间要大得多，出错的可能性也多得多。

测试并发程序所面临的一个最大的挑战是有些故障是以一定概率出现的，而不是每次测试一定会发生。对于这类 Bug 的测试要比在单线程程序中更加严格才行。

安全性测试的任务是保证每个类的行为符合相应的规格说明。假设有个链表，维护了一个 size 变量用于表示链表的长度，每次修改链表的时候都要更新这个变量。一个安全性测试可能就是检测这个变量的值是否等于链表中元素的个数。在单线程程序中这种测试很容易，因为在你读取 size 变量值的时候，链表对象的状态不可能改变。然而，在并发程序中，获取 size 变量值的操作和获取链表中元素数目的操作必须被包装进某个原子操作中去。

活跃性测试也不容易，你怎么区分一个线程是阻塞了还是运行速度慢（但仍在运行）呢？相似地，你如何通过测试来确定一个算法不会死锁呢？

性能测试与活跃性测试关系密切。性能可以用如下几个指标来衡量：

- **吞吐量：**并发任务的执行速度
- **响应速度：**请求与响应之间的时间差
- **可伸缩性：**当有更多计算资源可用时，吞吐量的提升能力

12.1. 正确性测试

为了演示并发类的测试，我们先来创建一个有界缓冲器 BoundedBuffer，它使用信号量来实现定界和阻塞。

BoundedBuffer 使用一个固定长度的数组，并提供了可阻塞的 put 和 take 方法。这两个方法被两个信号量控制。availableItems 信号量表示缓冲区中现有的元素数目，初始值为 0。availableSpaces 信号量表示缓冲区中还可以插入多少个元素，初始值为缓冲区的大小。

```
@ThreadSafe
public class BoundedBuffer<E>
{
    private final Semaphore availableItems, availableSpaces;
```

```
@GuardedBy("this")
private final E[] items;

@GuardedBy("this")
private int putPosition = 0, takePosition = 0;

@SuppressWarnings("unchecked")
public BoundedBuffer(int capacity)
{
    availableItems = new Semaphore(0);
    availableSpaces = new Semaphore(capacity);
    items = (E[]) new Object[capacity];
}

public boolean isEmpty()
{
    return availableItems.availablePermits() == 0;
}

public boolean isFull()
{
    return availableSpaces.availablePermits() == 0;
}

public void put(E x) throws InterruptedException
{
    availableSpaces.acquire();
    doInsert(x);
    availableItems.release();
}

public E take() throws InterruptedException
{
    availableItems.acquire();
    E item = doExtract();
    availableSpaces.release();
    return item;
}

private synchronized void doInsert(E x)
{
    int i = putPosition;
    items[i] = x;
```

```
        putPosition = (++i == items.length) ? 0 : i;
    }

    private synchronized E doExtract()
    {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length) ? 0 : i;
        return x;
    }
}
```

take 方法首先需要从 availableItems 信号量获取一个许可，如果缓冲区非空，这个许可立马就可以获取到，否则该操作会导致当前线程阻塞，直到缓冲区非空为止。put 方法与 take 方法相反，因此，put 方法或者 take 方法结束后两个信号量中许可的总和总是等于缓冲器的容量。

在实际项目中，如果你需要使用有界缓冲器的话你不用自己实现有界缓冲器类，你可以使用 ArrayBlockingQueue 或者 LinkedBlockingQueue。

12.1.1. 基本测试单元

对于 BoundedBuffer 类的最基本测试是这样的：首先创建一个 BoundedBuffer 对象，然后调用它的一些方法，最后再判断是否满足后置条件和一致性约束。你可以想到的是，一个新建的 BoundedBuffer 对象应该是空的，isEmpty 方法应该返回 true，isFull 方法应该返回 false。另一个复杂一点的测试用例是，假设 BoundedBuffer 的最大容量为 N，插入 N 个元素之后测试 isFull 方法是否返回 true。我们可以提供如下测试用例：

```
public class BoundedBufferTest extends TestCase
{
    void testIsEmptyWhenConstructed()
    {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);

        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException
```

```
    {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);

        for (int i = 0; i < 10; i++)
            bb.put(i);

        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
}
```

这些简单的测试方法虽然没有涉及到多线程，但也是十分必要的，因为你必须先确定这个类在单线程环境下不会有问题，然后再测试其在多线程环境下是否能正常工作。

12.1.2. 测试可阻塞操作

测试某个类的并发特性需要引入多线程，不幸的是大多数单元测试框架（比如 JUnit）都不是并发友好的，它们缺乏创建多线程然后检测他们是否运行正常的机制。

测试一个方法是否阻塞会引入额外的复杂性：如果方法真的阻塞了，你怎么知道呢？最简单的办法是使用中断机制，在一个单独的线程中调用该方法，等待其阻塞，然后中断这个线程，然后声明阻塞操作完成。当然这需要你的阻塞式方法能够响应中断，在被中断之后能够提前返回或者抛出 InterruptedException。

如下代码演示了如何测试阻塞式方法：

```
/**
 * 新建一个BoundedBuffer对象，然后立即从中取元素，take方法应该阻塞才对
 */
void testTakeBlocksWhenEmpty()
{
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread()
    {
        public void run()
        {
            try
            {
                int unused = bb.take();
                fail();  // if we get here, it's an error
            }
```

```
                catch (InterruptedException success)
                {
                }
            }
        };

        try
        {
            taker.start();            //启动测试线程
            Thread.sleep(1000);       //等待一秒钟
            taker.interrupt();        //中断测试线程
            taker.join(1000);         //等待测试线程完成善后工作

            //如果测试线程仍然活跃说明测试失败
            assertFalse(taker.isAlive());
        }
        catch (Exception unexpected)
        {
            fail();//测试过程中抛出任何异常，此次测试失败
        }
    }
```

这个测试用例创建了一个测试线程 taker，其主要工作是尝试从一个空 BoundedBuffer 对象中取元素。如果 take 方法不阻塞，测试用例不通过。如果 take 方法阻塞，在调用 taker.interrupt 方法之后，take 方法应该抛出 InterruptedException，随后 taker 线程应该终止。主线程使用 join 方法等待 taker 线程终止，然后调用 taker.isAlive 方法判断 taker.join 方法是成功返回还是超时返回，如果是成功返回 taker.join 方法应该很快就返回。

你也可以用同样方式来测试当队列中有元素时 take 方法不应该阻塞，如果阻塞则测试用例不通过。

不要用 Thread.getState 来测试线程的状态，因为这种方式不可靠。该方法返回的结果不应该用于并发控制，对于测试没多大用。

12.1.3. 安全性测试

上面两节的测试方法无法检测出多线程竞争条件，有些 Bug 只有在某些多线程重叠状态下才会发作。为了测试一个并发类在不可预测的多线程访问环境下能正常工作，我们需要设置多个线程，不断地执行 put 和 take 方法，然后测试看

是否有错误发生。

用于发现并发类中安全性问题的并发性测试用例不太好写，因为测试用例本身就是并发程序。有时候开发一个好的并发性测试用例比开发一个并发类更困难。

先看如下一个随机数产生函数：

```
static int xorShift(int y)
{
    y ^= (y << 6);
    y ^= (y >>> 21);
    y ^= (y << 7);
    return y;
}
```

如下是一个用于测试 BoundedBuffer 中是否存在竞争条件的测试用例：

```
public class PutTakeTest extends TestCase
{
    //任务执行器
    protected static final ExecutorService pool =
                                Executors.newCachedThreadPool();

    //用于协调线程运行的屏障
    protected CyclicBarrier barrier;

    //被测试的BoundedBuffer
    protected final BoundedBuffer<Integer> bb;

    //生产者数量（等于消费者数量）
    protected final int nPairs;

    //每个生产者线程向BoundedBuffer中插入几个元素
    protected final int nTrials;

    //用于记录所有曾经被插入到BoundedBuffer中的元素的和
    protected final AtomicInteger putSum = new AtomicInteger(0);

    //用于记录所有曾经从BoundedBuffer中取出的元素的和
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception
    {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
```

```
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials)
    {
        this.bb = new BoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test()
    {
        try
        {
            for (int i = 0; i < nPairs; i++)
            {
                pool.execute(new Producer());
                pool.execute(new Consumer());
            }

            // 该语句将本线程阻塞，直到所有生产者和消费者线程都调用了
            // 第一个barrier.await()方法，然后所有线程都继续运行
            barrier.await();

            // 该语句将本线程阻塞，直到所有生产者和消费者线程
            // 都调用了第二个barrier.await()方法
            barrier.await();

            // 两者若不等，说明BoundedBuffer有安全性问题
            assertEquals(putSum.get(), takeSum.get());
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }

    static int xorShift(int y)
    {
        y ^= (y << 6);
        y ^= (y >>> 21);
        y ^= (y << 7);
        return y;
```

```
}

class Producer implements Runnable
{
    public void run()
    {
        try
        {
            int seed = (this.hashCode() ^ (int) System.nanoTime());
            int sum = 0;
            barrier.await();

            //不断地向BoundedBuffer中添加元素
            for (int i = nTrials; i > 0; --i)
            {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }

            //将本线程向BoundedBuffer中添加的所有元素之和加到putSum之上
            putSum.getAndAdd(sum);
            barrier.await();
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }
}

class Consumer implements Runnable
{
    public void run()
    {
        try
        {
            barrier.await();
            int sum = 0;

            //不断从BoundedBuffer中取出元素
            for (int i = nTrials; i > 0; --i)
            {
                sum += bb.take();
```

```
                }

                //将本线程从BoundedBuffer中取出的所有元素之和加到takeSum上
                takeSum.getAndAdd(sum);
                barrier.await();
            }
            catch (Exception e)
            {
                throw new RuntimeException(e);
            }
        }
    }
}
```

需要注意的是，测试程序中不应该引入同步，因此对于多线程共享状态域 takeSum 和 putSum，这里使用 AtomicInteger 来表示。

PutTakeTest 方法启动了 N 个生产者线程用于生产元素，然后入队。同时也启动了 N 个消费者线程，不断从队列中取出元素。元素入队和出队的时候都要更新相应的求和校验域。

在不同的平台上，创建和启动线程都是中量级操作。如果线程较短，你在一个循环里启动了很多这样的线程的话，效率会很低，而且线程重叠也不易发生。我们这里使用了屏障 CyclicBarrier 保证了多线程一起启动，增加了线程重叠的可能性。另外，我们保证了每个线程都执行足够长的一段时间，进一步增加了线程重叠的可能性。

test 方法中启动了相同数量的生产者和消费者线程，而且所有生产者线程入队的元素最终都会被消费者线程从队列中取出，因此最终 putSum 和 takeSum 的值应该相等。

像 PutTakeTest 这样的测试趋向于能够发现安全性漏洞。例如，BoundedBuffer 类中的 doInsert 方法和 doExtract 方法应该是排他的，因此应该加 synchronized 关键字，如果缺失 synchronized 关键字，PutTakeTest 测试很快就能发现这里面存在的问题。

PutTakeTest 测试应该在多处理器系统中运行，这样做是为了增加潜在的线程重叠多样性。为了最大化对时机敏感的竞争条件的测试能力，线程数量应该超过 CPU 数量，以确保在任何时候都有线程在执行，有线程被挂起，减少线程交互

的可预测性。

在并发测试中，有时候测试用例由于 Bug 的存在一直不能运行结束。处理方法是让测试框架来判断，如果超过一段时间测试用例没有执行结束的话就将其终止。超时时间怎么设置主要靠经验判断。其实这不是并发程序测试独有的，单线程程序中也需要区分耗时任务和永远无法完成的任务。

12.1.4. 测试资源管理

前面几节主要关注于测试一个类，判断它是否符合规格说明。另一个需要测试的方面是，类不应该做没让它做的事，例如资源泄露。任何持有其他对象的对象不应该在不必要的时候仍然持有这个对象。内存泄漏会阻止垃圾回收器回收不需要的内存空间（也可能是文件句柄、Socket 或者数据库连接）。这将导致资源耗尽和程序崩溃。

对于 BoundedBuffer 这样的类来说资源管理问题是极其重要的。使用有界缓冲器的目的就是为了防止生产者线程跑得太快导致程序内存耗尽。

可以使用堆检查工具来检查应用程序的内存使用情况从而发现不需要的内存保留。有很多商业的和开源的堆分析工具可以做这样的事。

12.1.5. 使用回调

在如下代码中 TestingThreadFactory 类主要用于测试 Executors 中默认的线程工厂。它维护了一个状态域 numCreated，用于记录已经创建的线程数量。测试用例可以验证创建的线程数量是否正确。

```
public class TestingThreadFactory implements ThreadFactory
{
    public final AtomicInteger numCreated = new AtomicInteger();
    private final ThreadFactory factory = Executors.defaultThreadFactory();

    public Thread newThread(Runnable r) {
        numCreated.incrementAndGet();
        return factory.newThread(r);
    }
}
```

如果核心池大小小于最大允许的线程数量，当核心池中线程不够用时，线程池会创建额外的线程。你可以提交几个耗时任务给线程池，这样在一段时间内线

程池中就会有固定数量的线程，这时候你可以做断言，测试线程池的扩展情况是否和预期的相同。例如：

```
public class TestingThreadFactory implements ThreadFactory
{
    public final AtomicInteger numCreated = new AtomicInteger();
    private final ThreadFactory factory = Executors.defaultThreadFactory();

    public Thread newThread(Runnable r)
    {
        numCreated.incrementAndGet();
        return factory.newThread(r);
    }

    public static void testPoolExpansion() throws InterruptedException
    {
        int MAX_SIZE = 10;

        //固定大小的线程池
        ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE);

        //为线程池设置线程工厂
        TestingThreadFactory threadFactory=new TestingThreadFactory();
        ((ThreadPoolExecutor)exec).setThreadFactory(threadFactory);

        //提交很多耗时任务给线程池
        for (int i = 0; i < 10*  MAX_SIZE; i++)
        {
            exec.execute(new Runnable()
            {
                public void run()
                {
                    try
                    {
                        Thread.sleep(Long.MAX_VALUE);
                    }
                    catch (InterruptedException e)
                    {
                        Thread.currentThread().interrupt();
                    }
                }
            });
        }
```

```
        //等待创建的线程数大于或等于MAX_SIZE，最多等待2秒
        for (int i = 0;i < 20 && threadFactory.numCreated.get() < MAX_SIZE;i++)
            Thread.sleep(100);
        //判断实际创建的线程数量是否符合预期
        if(threadFactory.numCreated.get()!=MAX_SIZE)
            System.out.println("Error Happend!");

        System.out.println(threadFactory.numCreated.get()+" Threads has been
created!");

        exec.shutdownNow();
    }

    public static void main(String[] args) throws InterruptedException
    {
        TestingThreadFactory.testPoolExpansion();
    }
}
```

12.1.6. 产生更多的线程重叠

由于很多潜在的并发 Bug 都是随机发作的，且概率较低，因此并发 Bug 的检测是一种数字游戏。不过你可以采取一些措施来提高并发 Bug 发作的概率。我们前面提到了，应该在多处理器系统中进行测试，且 CPU 数量应该少于活跃的线程数量，这样可以产生更多的线程重叠。事实上，使用大量的不同处理器数量、操作系统和处理器体系结构的系统来进行测试可以发现更多 Bug，特别是那种只在某些特定运行环境中才会发作的 Bug。

一个有用的技巧是使用 Thread.yield 方法来鼓励更多的线程上下文切换。需要注意的是在有些平台上 JVM 将 Thread.yield 解释为 no-op，因此这种方法并不总是有效。使用一个短时睡眠或许是更好的选择，例如 Thread.sleep(10)。例如在如下代码中，转账方法从一个账户转一定数量的钱到另一个账户。在两个更新操作之间插入 Thread.yield 方法可以有助于发现一些时间敏感的并发 Bug。不过在产品化的时候应该将 Thread.yield();这一句去掉。

```
public synchronized void transferCredits(Account from, Account to, int amount)
{
    from.setBalance(from.getBalance() - amount);
    Thread.yield();
    to.setBalance(to.getBalance() + amount);
```

```
}
```

12.2. 性能测试

虽然性能测试往往与功能测试有所重叠，但是它们的目标是不同的。一般来说，性能测试的主要目标是测试响应速度。性能测试应该反映实际运行场景中某个组件的使用情况。

大多数情况下实际应用场景很容易找到。例如 BoundedBuffer 类一般只用于生产者-消费者模式，因此我们很可以将 12.1.3 节的 PutTakeTest 类扩展成为一个性能测试用例类。

性能测试的另一个目标是测试在不同的参数配置下程序的响应速度，比如线程数量、缓冲区大小等。由于这些参数的最优值受到硬件环境的影响（比如处理器类型、处理器数量、内存空间等），因此实际运行中这些参数的值应该动态配置，不过一般来说需要找到一个在多个系统上都能运行良好的默认值。

12.2.1. 扩展 PutTakeTest 类以支持性能测试

我们对 PutTakeTest 类的扩展工作主要是为其加上时间测量机制。我们可以将一个操作重复做多次，然后用总耗时除以次数，得到这个操作的平均耗时。我们先来创建一个计时组件：

```
public class BarrierTimer implements Runnable
{
    private boolean started;
    private long startTime, endTime;

    public synchronized void run()
    {
        long t = System.nanoTime();
        if (!started)
        {
            started = true;
            startTime = t;
        }
        else
        {
            endTime = t;
        }
    }
```

```
    public synchronized void clear()
    {
        started = false;
    }

    public synchronized long getTime()
    {
        return endTime - startTime;
    }
}
```

现在我们来扩展 PutTakeTest 类，为其加上计时设施：

```
public class TimedPutTakeTest extends PutTakeTest
{
    //计时器
    private BarrierTimer timer = new BarrierTimer();

    public TimedPutTakeTest(int cap, int pairs, int trials)
    {
        super(cap, pairs, trials);
        barrier = new CyclicBarrier(nPairs * 2 + 1, timer);
    }

    /**
     * 覆盖test方法
     */
    @Override
    public void test()
    {
        try
        {
            //清空计时器
            timer.clear();
            for (int i = 0; i < nPairs; i++)
            {
                pool.execute(new PutTakeTest.Producer());
                pool.execute(new PutTakeTest.Consumer());
            }

            //第一次开启屏障之后将会运行timer.run方法，设置开始时间
            barrier.await();

            //第二次开启屏障之后将会运行timer.run方法，设置结束时间
            barrier.await();
```

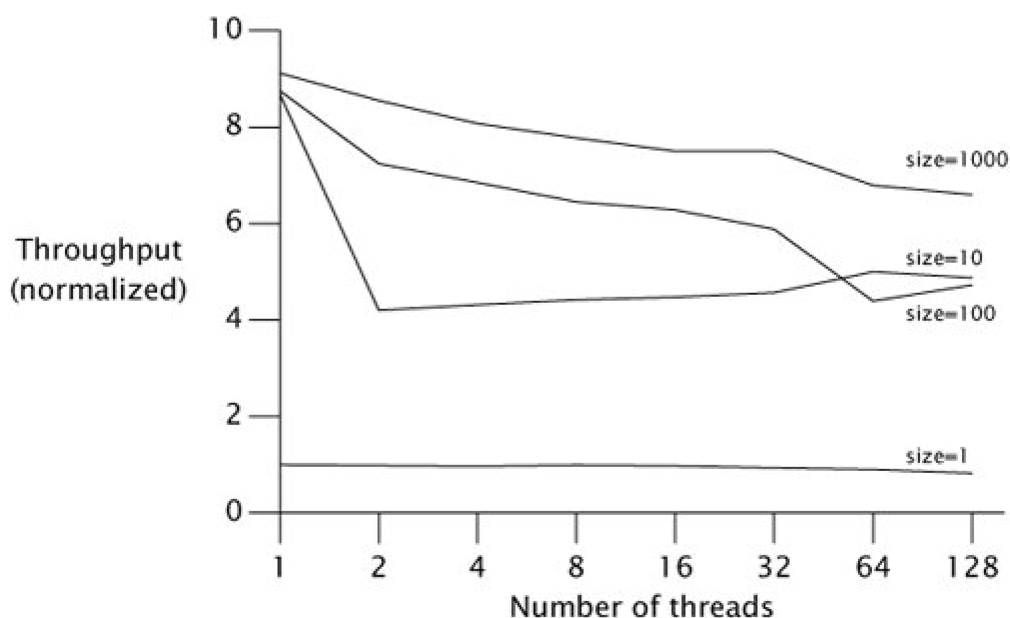
```
            //计算并打印平均耗时
            long nsPerItem = timer.getTime() / (nPairs * (long) nTrials);
            System.out.println("Throughput: " + nsPerItem + " ns/item");

            //正确性测试，两者若不等，说明BoundedBuffer有安全性问题
            assertEquals(putSum.get(), takeSum.get());
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }

    /**
     * 主程序
     */
    public static void main(String[] args) throws Exception
    {
        //每个生产者线程向BoundedBuffer中插入10000个数
        int tpt = 10000;

        //测试不同的缓冲器容量与线程数量组合对应运行耗时
        for (int cap = 1; cap <= 1000; cap *= 10)
        {
            for (int pairs = 1; pairs <= 128; pairs *= 2)
            {
                System.out.print("Capacity: " + cap + "\t");
                System.out.print("Pairs: " + pairs + "\t");
                TimedPutTakeTest t = new TimedPutTakeTest(cap, pairs, tpt);
                t.test();
            }
            Thread.sleep(100);
        }
        PutTakeTest.pool.shutdown();
    }
}
```

输出结果：

```
Capacity: 1Pairs: 1     Throughput: 5883 ns/item
Capacity: 1Pairs: 2     Throughput: 7388 ns/item
Capacity: 1Pairs: 4     Throughput: 11270 ns/item
Capacity: 1Pairs: 8     Throughput: 11108 ns/item
Capacity: 1Pairs: 16    Throughput: 10652 ns/item
```

```
Capacity: 1 Pairs: 32   Throughput: 11156 ns/item
Capacity: 1 Pairs: 64   Throughput: 9438 ns/item
Capacity: 1 Pairs: 128  Throughput: 10356 ns/item
Capacity: 10    Pairs: 1     Throughput: 645 ns/item
Capacity: 10    Pairs: 2     Throughput: 671 ns/item
Capacity: 10    Pairs: 4     Throughput: 563 ns/item
Capacity: 10    Pairs: 8     Throughput: 520 ns/item
Capacity: 10    Pairs: 16    Throughput: 504 ns/item
Capacity: 10    Pairs: 32    Throughput: 493 ns/item
Capacity: 10    Pairs: 64    Throughput: 493 ns/item
Capacity: 10    Pairs: 128   Throughput: 503 ns/item
Capacity: 100   Pairs: 1     Throughput: 732 ns/item
Capacity: 100   Pairs: 2     Throughput: 702 ns/item
Capacity: 100   Pairs: 4     Throughput: 633 ns/item
Capacity: 100   Pairs: 8     Throughput: 618 ns/item
Capacity: 100   Pairs: 16    Throughput: 528 ns/item
Capacity: 100   Pairs: 32    Throughput: 471 ns/item
Capacity: 100   Pairs: 64    Throughput: 472 ns/item
Capacity: 100   Pairs: 128   Throughput: 476 ns/item
Capacity: 1000  Pairs: 1     Throughput: 724 ns/item
Capacity: 1000  Pairs: 2     Throughput: 741 ns/item
Capacity: 1000  Pairs: 4     Throughput: 589 ns/item
Capacity: 1000  Pairs: 8     Throughput: 585 ns/item
Capacity: 1000  Pairs: 16    Throughput: 515 ns/item
Capacity: 1000  Pairs: 32    Throughput: 475 ns/item
Capacity: 1000  Pairs: 64    Throughput: 466 ns/item
Capacity: 1000  Pairs: 128   Throughput: 466 ns/item
```

将吞吐量与缓冲器容量和线程数量之间的关系画成曲线图如下：



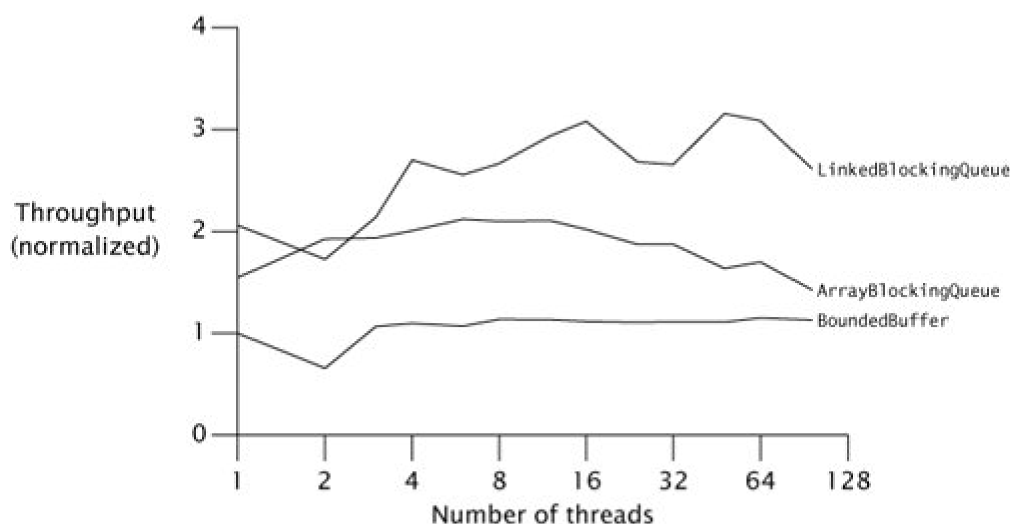
可以看出当缓冲器容量为 1 的时候，吞吐量很低，但是增加到 10 之后吞吐量有很大提升。缓冲器容量继续增大对吞吐量提升的效果不明显。

从图上可以看出，创建大量的生产者和消费者线程并没有大幅降低程序的性能。然而不要因此就认为你总是可以在使用有界队列的生产者-消费者程序中使用更多的线程。这个测试程序是人造的，能在多大程度上模仿真实使用场景还很难说。如果在真实应用场景中，生产者除了向有界队列中添加元素之外还做很多其他工作，消费者除了从有界队列中取元素之外还做很多其他工作，这样的话，如果存在大量的生产者和消费者线程将会对吞吐量造成影响。

12.2.2. 比较多种算法

虽然 BoundedBuffer 作为一个可阻塞的有界队列来说，是一个不错的实现，但是它比不过 ArrayBlockingQueue 和 LinkedBlockingQueue。系统库中的类都是经过严格筛选和调优的（使用类似于上一节的测试方法）。BoundedBuffer 之所以性能较差的原因是在不同线程中运行的多个 put 和 take 方法之间会争用信号量与锁。

使用一个与 TimedPutTakeTest 类似的测试程序对 ArrayBlockingQueue、BoundedBuffer、LinkedBlockingQueue 进行测试，不过这次他们的容量都设置为 256。测试结果如下图所示：



从图中可以看出 LinkedBlockingQueue 的可伸缩性比 ArrayBlockingQueue 的可伸缩性要好。这是因为 LinkedBlockingQueue 允许更多的线程并发访问 put

和 take 方法。（这是另一个单线程中程序优化直觉在多线程环境下失灵的例子）

12.2.3. 测量响应速度

迄今为止，在本章中我们一直在讨论吞吐量，这是并发程序中最重要的性能指标。但是，有时候我们需要测量响应速度，它是服务时间的变种。有时候为了获得相对稳定的服务时间，增加平均服务时间是值得的。此外，可预测性也是另一个重要的性能指标，有时候我们需要知道在 100 毫秒内能完成百分之多少的任务。

任务执行耗时数据的直方图是衡量服务时间稳定性的一个比较好的可视化手段（显示执行时间在各个范围内的任务数量）。

12.3. 避免性能测试中存在的陷阱

从理论上来讲，开发一个性能测试用例是比较容易的。首先，寻找典型应用场景，然后写一个程序让其在这个场景下执行很多次，然后计算耗时。

从实践上来讲，你还必须警惕一些代码上的陷阱，这些陷阱可能导致性能测试得到无意义的结果。

12.3.1. 垃圾回收

垃圾回收器运行的时机是无法预测的，因此在测试阶段是否会进行垃圾回收谁也说不准。一种可能的情况是，如果你运行 N 次迭代不会引起垃圾回收，但是如果你运行 N+1 次迭代就会引起垃圾回收。一个小的变动就有可能导致测试出的每个迭代平均耗时出现很大不同。

有两种方法可以消除垃圾回收对你的性能测试结果的干扰。一种方法是确保在测试阶段不会进行垃圾回收（你可以使用-verbose:gc 选项来启动 JVM，这样如果在测试过程中进行垃圾回收，则会在控制台上打印出来）。另一种方法是确保垃圾回收器在测试期间运行一定次数，这样就能使你的性能测试结果反映出内存分配与垃圾回收造成的影响，从而更贴近实际场景中的运行状况。

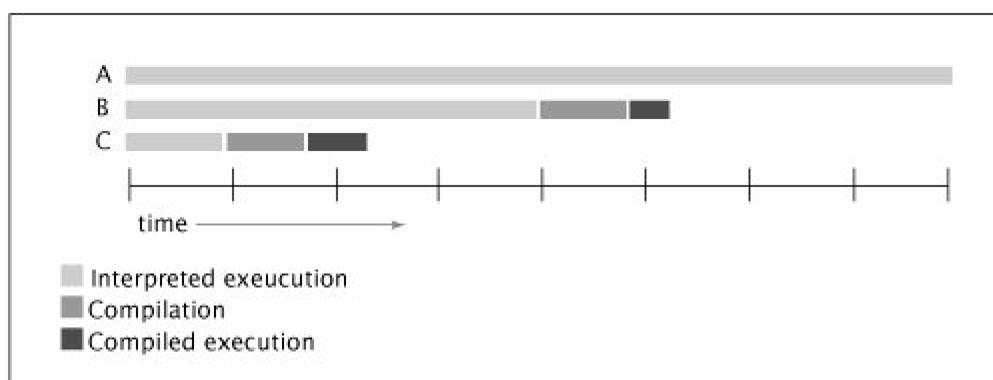
大多数生产者-消费者模式都会造成大量的内存分配和垃圾回收，生产者线程不断创建对象，然后将它们入队，消费者线程不断从队列中取出对象，使用完之后将其丢弃。如果测试用例能够运行足够长的时间（几分钟），则在测试期间必定会有多次垃圾回收，这样得出的测试结果将会更加准确。

12.3.2. 动态编译

确定动态编译语言（比如 Java）的性能指标很困难。HotSpot JVM 使用字节码解释与动态编译相结合的执行方式。当一个类第一次被加载的时候，JVM 通过执行相应的字节码来执行它。如果一个方法经常运行，JVM 有可能使用动态编译技术将其编译成机器码，这将对其直接执行而不是解释执行。

动态编译介入的时机是不可预测的。你应该在所有可被动态编译的代码已经被动态编译之后再进行性能测试。由于在实际场景中，经常运行的方法都会被动态编译，因此对这些代码的解释执行性能进行测试没有意义。

如果在测试的过程中动态编译突然介入，那么得出的性能测试结果将是不准确的。这是因为动态编译也要消耗 CPU 资源。此外，测试过程中一半数据来自解释执行，另一半数据来自直接执行，这样得出的测试结果是没有意义的。下图显示了三种情况，线 A 表示在多次迭代过程中迭代体都是解释执行的，直到测试结束。线 B 表示在解释执行完成一半迭代之后动态编译开始介入，编译完成后，直接执行。线 C 表示动态编译在早期就开始介入。



动态编译后直接执行的代码在某个时机可能被反转为解释执行，在某个恰当的时机有可能又被重新编译然后直接执行。这些时机很难确定。

一种能够防止动态编译干扰性能测试结果的方法是让你的测试程序运行较长的时间（至少几分钟）。这样的话就能将解释执行阶段的耗时与动态编译耗时这两个干扰因素忽略不计。另一种方法是先进行热身运行，在热身运行阶段不进行计时测量，这样在真正开始计时的时候，所有应该被动态编译的代码段已经被动态编译完成了。在 HotSpot JVM 中，使用-XX:+PrintCompilation 选项启动 Java 虚拟机可以在动态编译开始的时候打印一个消息到控制台。这样你就可以

知道在进行实际计时测量之前是否完成了动态编译。

另外，你也可以在同一个虚拟机上进行好几组测试，第一组测试结果将会被丢弃，因为它会受到动态编译的影响。后面几组测试的结果应该差别不大，如果有较大差别，需要寻找到底是什么原因导致测试结果的不可重复。

JVM 使用了很多背景线程来处理一些内务工作。如果在一个测试线程中进行多个不相关的计算密集型测试的话，应该在这些测试之间适当停顿，让 JVM 有机会执行这些后台线程，这样就能减少后台线程对性能测试结果的影响。

12.3.3. 不切实际的代码访问路径

JVM 可以使用执行中收集到的信息来优化动态编译产生的代码，也就是说同样一个方法在不同运行过程中会被编译产生不同的代码。

因此，你的测试程序不仅要与实际使用场景近似，而且要与实际使用场景中的代码访问路径近似。不然的话动态编译器就可能对代码进行不同的优化，这样的话得到的性能测试结果就不准了。

12.3.4. 不切实际的竞争程度

并发应用程序倾向于将不同类型的操作重叠执行，例如一个线程正在从共享队列中取任务，另一个线程正在执行任务。依赖于这两种操作的比例，应用程序的竞争程度也不同，因此会展现出不同的性能和可伸缩性。

假设有 N 个线程从共享任务队列中取任务，然后执行任务，如果任务是计算密集型的，且是耗时的，这样的话竞争就很少。如果任务很短那么将会有很多竞争。

为了能够测试出合乎实际的结果，性能测试程序应该尽量与真实应用场景一致。如果性能测试程序中的任务特征与真实场景不同，将会得出错误的性能瓶颈。从 11.5 节中可以看出，不同 Map 实现的竞争程度不同可以导致吞吐量的巨大差异。在那个测试程序中每个线程只是不断访问 Map，然而，在实际应用场景中每次访问 Map 之后一般会有大量的计算，因此在实际场景中竞争是较低的。

从这方面来说，TimedPutTakeTest 可能并不是一个好的性能测试用例。这是因为，工作线程除了访问 BoundedBuffer 之外没有做多少其他工作，在大多数使用 BoundedBuffer 的生产者-消费者程序中，这都不是典型的应用场景。

12.3.5. 消除死代码

JVM 可以消除一些死代码（对整个应用程序运算结果没有影响的代码块），这将导致性能测试的结果不准确。很多测试程序使用-server 选项运行将比使用-client 选项运行速度更快。这不仅是因为-server 选项能产生更加优化的代码，更重要的是-server 选项使得 JVM 能够消除死代码。你应该让你写的代码不受死代码消除的影响。

在 PutTakeTest 测试用例中，我们在每个线程中计算了所有入队的元素的和并且跨线程地将这些和加起来，但是如果你不使用 putSum 和 takeSum 的话，这些求和操作可能会被消除掉。在 PutTakeTest 中我们恰巧需要使用它们来做校验和验证，不过你也可以使用打印语句将其打印到控制台，这样也可保证某个变量被使用，从而不会被死代码消除功能消除掉。

不过在进行测试的时候使用 I/O 功能可能导致测试结果不准确。一个好的方法是，计算对象的某个域的哈希码，然后和 System.nanoTime 比较，如果恰巧相等的话就打印一个空格，像这样：

```
if (foo.x.hashCode() == System.nanoTime())
    System.out.print(" ");
```

不仅每个计算结果应该被使用，而且计算结果的值应该是不可猜的，不然的话有些聪明的优化器会将计算过程替换为某个预先计算出来的数值。那些输入是静态数据的测试程序最应该注意这一点。

12.4. 其他测试方法

虽然我们一般都会认为一个好的测试程序应该能够发现被测试程序中的所有 Bug，但这是不现实的。NASA 在测试方面下了很大的力气，他们为每个开发人员配备了大约 20 个测试人员，即使这样，他们开发出的代码产品还是会有缺陷。对于复杂的应用程序是无法测试出所有的 Bug 的。

测试的目的不是找出错误，而是增加对代码能够正常运行的信心。由于发现所有的 Bug 是不可能的，QA 的目标应该是在有限的测试资源下尽可能提高对代码能正常运行的信心。测试对于提高并发类能正常运行的信心是至关重要的。不过测试只是 QA 使用的所有方法中的一种。

通过使用其他 QA 方法，比如代码审查和静态分析，你可以进一步增强代码

能正常运行的信心。

12.4.1. 代码审查

虽然单元测试和压力测试很重要，但是多人代码审查也是不可替代的。审查人员应该有多个，但不包括代码的作者。并发专家比测试程序更能发现并发程序中隐藏的一些微妙的竞争条件。此外，代码审查还可以提高注释质量和实现细节的质量，从而减少将来的维护成本。

12.4.2. 静态分析工具

静态分析工具是对形式化测试与代码审查的一个很好的补充。静态代码分析是一种在不执行代码的情况下对其进行分析的过程。代码审计工具可以通过分析类来寻找一些经常发生的错误模式。FindBug（一种开源的静态分析工具）包含的 Bug 模式检测器可以发现很多常见的错误，这些错误中有很多是无法通过形式化测试和代码审查发现的。

静态分析工具可以产生一列警告，需要人工审查这些警告来确定它们是否真的代表某些错误。虽然静态分析工具还比较原始，但是已经逐渐成为形式化测试之外，用于发现 Bug 的一种有效补充。

目前为止 FindBugs 可以检测出如下几种并发 Bug 模式，以后会添加更多：

- **不一致的同步。**很多对象都使用其内部锁来保护它的所有状态域。如果某个域在被访问过程中并不总是被 this 锁保护，这可能意味着发现了一个并发 Bug。
- **调用 Thread.run 方法。**虽然调用 Thread.run 方法没问题，但是这个方法不是被设计用来直接调用的，如果代码中直接调用 Thread.run 方法，可能意味着这是一个 Bug。
- **未释放的锁。**与内部锁不同，显式锁不是自动释放的。标准用法是在 finally 块中释放显式锁，如果忘记释放显式锁，这将是个 Bug。
- **空的同步块。**
- **双重检查锁。**
- **在构造函数中启动一个线程。**在构造函数中启动线程可能导致子类出问题，也会导致 this 逃逸。

- **条件等待错误。**
- **通知错误。**
- **在持有锁的时候睡眠或者调用 Object.wait 方法。**在持有锁的情况下调用 Thread.sleep 方法会导致活跃性风险。在持有锁的情况下调用 Object.wait 可能导致当前线程再也无法被唤醒。
- **旋转循环。**如果一个循环频繁地检查一个域的值是否满足要求，这将浪费 CPU 周期，此外如果这个域不是 volatile 的，可能导致这个循环永远无法终止。

12.4.3. 面向切面的测试技术

面向切面编程技术可以用来判定约束是否满足。

12.4.4. 性能分析与监测工具

大多数商业性能分析工具都支持多线程。使用它们可以了解到程序的内部运行情况。大多数性能分析工具都会显示一个时间线，用不同的颜色标出各个线程的状态（可运行、阻塞等待锁、阻塞等待 I/O 等）。这样就可以了解程序是如何使用 CPU 资源的，如果 CPU 利用率很低，可以找到瓶颈所在。很多性能分析工具可以找出哪些锁引起的竞争最多。

12.5. Object 对象中与同步相关的方法

wait、notify 和 notifyAll 方法是 Object 类的 final native 方法。所以这些方法不能被子类重写，Object 类是所有类的超类。

- void notifyAll()：解除所有那些在该对象上调用 wait 方法的线程的阻塞状态，让它们有机会获取该对象的锁。该方法只能在同步方法或同步块内部调用。如果当前线程不是锁的持有者，该方法抛出一个 IllegalMonitorStateException 异常。
- void notify()：随机选择一个在该对象上调用 wait 方法的线程，解除其阻塞状态，让它有机会获取该对象的锁。该方法只能在同步方法或同步块内部调用。如果当前线程不是锁的持有者，该方法抛出一个 IllegalMonitorStateException 异常。
- void wait()：导致当前线程进入阻塞状态，直到它被其他线程通过

notify()或者notifyAll唤醒。唤醒后当前线程将有机会竞争对象的锁。该方法只能在同步方法中调用。如果当前线程不是锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

- **void wait(long millis)和void wait(long millis, int nanos)：**导致当前线程进入阻塞状态直到它被通知或者经过指定的时间。这些方法只能在同步方法中调用。如果当前线程不是锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

Object.wait() 和 Object.notify() 和 Object.notifyAll() 必须写在synchronized方法内部或者synchronized块内部，这是因为这几个方法要求当前正在运行object.wait()方法的线程拥有object的对象锁。

一个使用 Object.wait()和 Object.notify()方法实现生产者-消费者模式的例子如下：

```
/**
 * 被多线程共享的信息类
 */
class Info
{
    private String name = "name";
    private String content = "content";
    private boolean flag = true;

    /**
     * 修改对象状态，该方法被生产者调用
     */
    public synchronized void set(String name, String content)
    {
        // 标志位为false，进入阻塞状态，直到被通知为止
        if (!flag)
        {
            try
            {
                this.wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
```

```
        // 设置状态域
        this.name=name;
        try
        {
            Thread.sleep(10);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        this.content=content;

        // 修改标志位为false，表示生产者已经完成资源，消费者可以消费。
        flag = false;

        // 唤醒消费者进程
        this.notify();
    }

    /**
     * 获取对象状态，该方法被消费者调用
     */
    public synchronized void get()
    {
        // 标志位为true，进入阻塞状态，直到被通知为止
        if (flag)
        {
            try
            {
                this.wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        // 打印状态信息
        try
        {
            Thread.sleep(100);
        }
        catch (InterruptedException e)
```

```
            {
                e.printStackTrace();
            }
            System.out.println(this.name + ":-->" + this.content);

            // 修改标志位为true，表示消费者拿走资源，生产者可以生产。
            flag = true;

            // 唤醒生产者进程。
            this.notify();
        }
    }

/**
 * 生产者
 */
class Producer implements Runnable
{
    private Info info = null;

    public Producer(Info info)
    {
        this.info = info;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            this.info.set("name" + i, "content" + i);
        }
    }
}

/**
 * 消费者
 */
class Consumer implements Runnable
{
    private Info info = null;

    public Consumer(Info info)
    {
```

```
        this.info = info;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            this.info.get();
        }
    }
}

/**
 * 测试类
 */
public class WaitNotifyTest
{
    public static void main(String args[])
    {
        Info info = new Info();
        Producer pro = new Producer(info);
        Consumer con = new Consumer(info);
        new Thread(pro).start();
        new Thread(con).start();
    }
}
```

输出结果：

name0:-->content0
name1:-->content1
name2:-->content2
name3:-->content3
name4:-->content4
name5:-->content5
name6:-->content6
name7:-->content7
name8:-->content8
name9:-->content9

可以看出，虽然生产者运行速度要比消费者快，但是由于使用了 wait 和 notify 机制，两者能够协调工作。如果去掉 set 和 get 方法中与 wait、notify 相关的代码输出结果如下：

name0:-->content0

```
name2:-->content2
name9:-->content9
name9:-->content9
name9:-->content9
name9:-->content9
name9:-->content9
name9:-->content9
name9:-->content9
name9:-->content9
```

可以看出，生产者线程与消费者线程失步了，中间漏掉了很多设置信息。

12.6. 本章小结

测试并发程序的正确性比较困难，因为大多数并发程序的出错模式都是不确定的，且发作概率较低，这也错误的发作往往是由于线程重叠时机等难以重现的原因引起的。此外，测试用例也可能引入额外的同步需求，从而掩盖被测类中的并发 Bug。

测试并发程序的性能也比较困难。Java 程序是动态编译语言，比静态编译的 C 语言更难以测试，因为动态编译、垃圾回收和优化操作会影响性能测试结果的有效性。

为了能够在代码产品化之前尽可能多地发现潜在的 Bug，你需要使用传统测试技术同时结合代码审查和静态分析工具。这些方法可以发现其他方法无法找到的 Bug。