Java 并发和多线程教程（2020 版）

作者：Jakob Jenkov，2020-03-29

翻译：GentlemanTsao，2020-6-15

（一），序章（多线程，多任务，并发的概念）

译者序：

我在读技术类的翻译书籍与文章时，常常遇到这样的困扰：一个段落看了几遍却怎么也不理解，虽然这样的段落在语法和语义上没有任何毛病。当我再去看英文原文时，发现原文是可以很自然的理解的。这让我意识到文章的质量不完全在于作者，也在于译者。

技术类的文章多数是英文，而英文和中文最显著的差异是思维方式和表达习惯。所以如果按照英文的思维方式和习惯来翻译，就会译出“看上去正确”的洋泾浜。博主的体会是，技术文章的翻译是一个转述的过程，这个过程的核心是完全表达出作者的技术思想，在这个基础上，以中国人的思维和习惯叙述出来。这需要译者热爱并理解技术，毕竟语言只是表达工具和载体，技术思想才是本质。

博主打算在空余时间里翻译出 java 并发的教程，方便更多的编程爱好者学习。在翻译时将遵循三个原则：简洁、清晰、灵活。当然，中心思想是“更容易理解”。

我会在这个专栏中持续更新最新的教程（内容较多，可以关注后慢慢再看）：

系列专栏：java 并发和多线程教程 2020 版

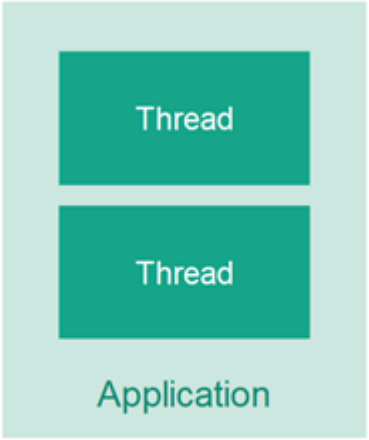
文章目录

前言

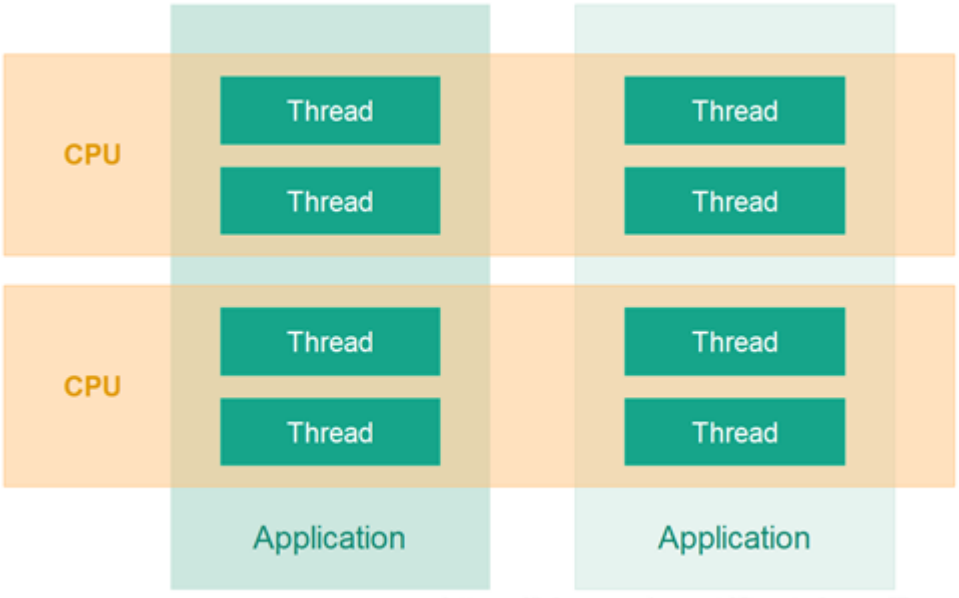
java 并发这一术语涵盖了 java 平台的多线程、并发和并行，它包括 java 并发工具，问题和解决办法。本教程涉及到 java 多线程相关的核心概念、并发结构、并发问题、多线程带来的好处和代价。

什么是多线程？

多线程是指在同一个程序里执行多个线程。线程就像是执行程序的一个单独的 CPU，所以多线程程序就像是有多个 CPU 同时在执行不同部分的代码一样。



不过一个线程并不等同于一个 CPU。通常情况下，多个线程共享一个 CPU 的执行时间。CPU 在每个线程之间以一定间隔切换着执行。当然，程序的各个线程也可以在不同的 CPU 上运行。



为什么需要多线程?

为什么要在程序中使用多线程呢？原因有很多，常见的原因是：

更好的使用单个 CPU；

更好的使用多个 CPU 或者多核 CPU；

从程序响应的角度给用户更好的体验；

从公平的角度给用户更好的体验。

我们会在下面的章节里更详细的解释这些原因。

更好的使用单个 CPU

使用多线程最常见的一个原因是为了能够更好的使用计算机资源。例如，如果某个线程通过网络发送了请求，在它等待网络返回结果的这段时间，另一个线程可以使用 CPU 来处理别的事情。此外，如果计算机配置了多个 CPU 或者多核 CPU，那程序通过多线程就可以使用这些额外的 CPU 核心。

更好的使用多个 CPU 或者多核 CPU

如果计算机配置了多个 CPU 或者多核 CPU，那就需要在你的程序中使用多线程，因为这样才能用到所有的 CPU 或者 CPU 核心。一个单独的线程最多只能使用一个 CPU。但是正如前文提到的那种情况，一个线程有时甚至没办法完全的利用好一个 CPU。

从程序响应的角度给用户更好的体验

使用多线程的另一个原因是为了提供更好的用户体验。例如，假如你在某个用户界面点击了一个按钮，结果是它要发送一个网络请求。问题在于，该让哪个线程执行该请求呢？如果让更新用户界面的线程来处理，那在等待请求响应的这段时

间，用户可能会察觉到界面“卡住”了。更好的做法是用一个后台线程来处理该请求，这样的话用户界面线程才有空闲来响应其他的用户请求。

从公平的角度给用户更好的体验

第四个原因是为了让用户更公平的分享计算机资源。例如，我们设想有一个服务器用于接收用户请求，但是这个服务器只有一个线程来处理这些请求。如果某个用户的请求需要花很长的时间来处理，那么所有其他用户的请求都要等这个请求处理完成才行。我们也可以让每个用户的请求都在自己的线程里执行，这样就避免了某个任务独占 CPU 的情况。

多线程 VS 多任务

在以前，计算机只有一个 CPU，而且同一时间只能运行一个程序。由于多数较小的计算机不具备同时运行多个程序的能力，也就不存在多任务。当然了，相对于个人计算机，很多大型计算机很早就已经具备同时运行多个程序的能力了。

多任务

后来出现了多任务，也就是计算机可以同时运行多个程序（也就是任务或进程）。但这实际上并不是“同一时刻”，而是程序之间共享一个 CPU。操作系统需要在程序之间切换，每个程序运行一小段时间，然后再切换到另一个。

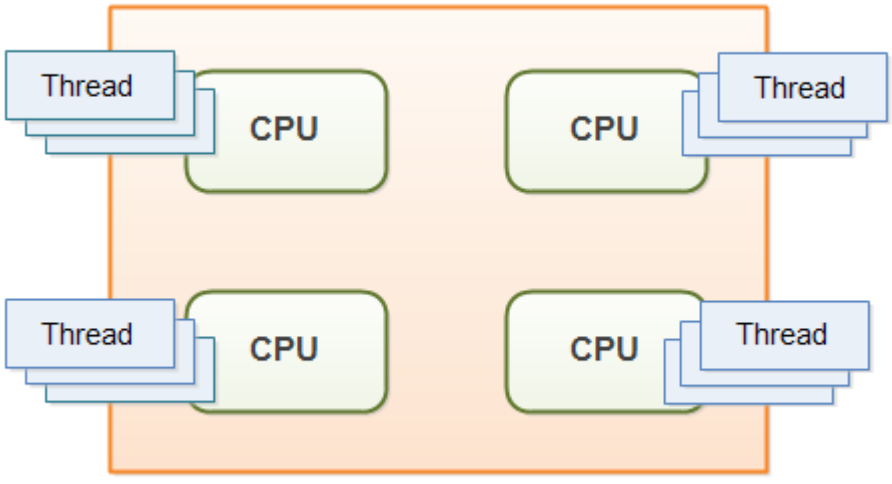
多任务的出现给软件开发者带来了新的挑战。程序不能再像以前那样，想当然的以为可以占用所有的 CPU 时间，内存以及其他计算机资源。一个良好的程序应该把不用的资源释放掉，这样其他程序才能使用。

多线程

后来又出现了多线程，也就是同一个程序内部可以执行多个线程。我们可以把执行的线程当作执行该程序的一个 CPU，所以同一个程序由多个线程执行就像是该程序由多个 CPU 执行一样。

多线程，有点难

对于某些类型的程序来说，用多线程来提升性能是非常好的方法。但是多线程比多任务的难度更大。由于多个线程在同一个程序里执行，因而它们会同时读取和写入同一块内存。这样的错误在单线程程序中是无法发现的。有些错误在只有单个 CPU 的机器上可能也发现不了，因为两个线程不可能真的“同时”执行。但是现代计算机出现了多核甚至多个 CPU 的配置。这就意味着这些单独的线程可以在各自的 CPU 或核心中同时执行。



如果一个线程读某一块内存的同时另一个线程在写这块内存，那第一个线程最后读到的值会是什么呢?是原来的值，还是第二个线程写入的值，还是两者混合之

后的值呢？还有，要是两个线程同时写同一块内存，那在写完之后，这块内存的值会是什么，是头一个线程写的值，还是第二个线程写的值，还是这两者混合的值呢？

这种冲突如果不能很好的预防，任何结果都有可能。这甚至会导致程序的行为无法预测，因为每次的执行结果会不一样。因此，对于开发者而言，知道怎样采取正确的预防手段十分重要。开发者需要学习控制线程访问共享资源的方法，比如如何访问内存，文件以及数据库等。

这是本 java 并发教程要讲解的主题之一。

java 的多线程和并发

java 是最早的让开发者的多线程编程变得更简单的编程语言之一。java 从一开始就已经具备了多线程的能力。因此，java 开发者经常会遇到我们在前面描述过的问题。这是我写这篇 java 并发教程的原因。它既可以作为自己的笔记，也能让做 java 开发的同伴们参考受益。

本教程将主要侧重于 java 多线程，但是多线程中出现的一些问题，和多任务以及分布式系统中出现的问题很相似。因而本教程中也会参考多任务以及分布式系统。所以我没有用“多线程”，而是使用了“并发”这个词。

并发模型

最初的 java 并发模型假设同一个程序内执行的多个线程会共享对象。这种类型的并发模型通常被称为“共享状态并发模型”。有许多并发编程语言的结构和工具包都是根据这种并发模型来设计的。

然而，自第一本 java 并发的书问世以来，甚至自 java5 并发工具包公布以来，并发的架构和设计都已经发生了翻天覆地的变化。共享状态并发模型带来了大量的并发问题，并且这些问题很难巧妙的处理。因此，另一种并发模型流行了起来，它被称为“无共享”或者叫“分离状态并发模型”。在这一模型中，线程不会共享任何对象和数据。这样就避免了很多在“共享状态并发模型”中存在的并发访问问题。

而今已出现新的异步“分离状态”平台和工具包，比如 Netty,Vert.x,Play / Akka 和 Qbit 。新的非阻塞式并发算法和非阻塞式工具也发布了，比如我们的工具包中新增了 LMax Disrupter。java7 的 Fork/Join 框架还推出了新的并行函数式编程，java8 又推出了 collection streams 调用接口。

随着这些新的发展，又到了我更新 java 并发教程的时机了。因此这本教程又一次开始不断扩充。我会在空闲时间撰写并发布新的教程。

（二）：多线程的好处（CPU 使用，程序设计，程序响应，资源分配）

多线程最明显的好处有：

更好的使用 CPU；

某些情况下让程序设计更简单；

让程序更好的响应；

让 CPU 资源在不同任务间分配的更均衡。

更好的使用 CPU

设想有这样一个程序，它从本地文件系统中读文件并处理。我们假定从硬盘中读一个文件需要 5s，然后处理这个文件需要 2s。所以处理两个文件就需要：

读文件 A，5s

处理文件 A，2s

读文件 B，5s

处理文件 B，2s

总共 14s。

在从硬盘读文件的时候，CPU 大部分时间都在等待硬盘读取数据，所以 CPU 多数时间处于空闲状态。我们可以改变操作顺序从而更好的使用 CPU，比如看下面的顺序：

读文件 A，5s

读文件 B，5s + 处理文件 A，2s

处理文件 B

总共 12s

CPU 先等待读取第一个文件，接着开始读第二个文件。当计算机 IO 设备在读第二个文件时，CPU 处理第一个文件。别忘了，CPU 等待从硬盘读文件的时候，几乎处于空闲状态。

总之，CPU 在等待 IO 时还可以处理其他事情。不一定是磁盘 IO，也可以是网络 IO 或者本机用户的输入。网络和磁盘 IO 通常比 CPU 和内存 IO 要慢的多。

让程序设计更简单

假如你要手工编写一个单线程的程序来实现上面的读取和处理文件的次序，你就需要跟踪每个文件的读取和处理状态。然而你也可以开启两个线程，每个线程仅负责读取和处理一个单独的文件。在等待磁盘读取文件的时候，这个线程会阻塞，而其他线程就可以让 CPU 处理已经读到的部分文件。于是，磁盘一直保持在工作状态，读取大量的文件到内存中。这样一来便更好的使用了磁盘和 CPU，也更易于编程，因为每个线程只需要跟踪一个单独的文件。

让程序更好的响应

把单线程程序改为多线程程序的另一个常见目的，是为了让程序更好的响应。设想一个监听端口请求的服务端程序。它接收到服务并处理该请求，然后继续监听。服务端大概是这样的循环：

```
whie（server is active）{

 监听请求

 处理请求

}
```

如果请求要花很长时间处理，那在这段时间里，其他用户则无法再发送请求。因为请求只有在服务端监听时才能被收到。

另一种设计是由监听线程把请求传递给工作线程，之后立即返回到监听状态。工作线程处理完请求后向用户发送返回消息。这种设计大概是这样：

```
while(server is active){

监听请求

把请求转给工作线程

}
```

这使得服务端线程可以很快的返回监听状态，于是更多的用户可以发请求给服务端，服务端的响应性更好了。

对于桌面应用也是一样的。如果你点击一个按钮开启一个长时间任务，并由更新界面的线程执行该任务。那么在该任务执行时，程序看上去是不响应的。也可以把任务交给工作线程，这样当工作线程忙于处理任务时，界面线程就有空闲来响应其他的用户请求。工作线程完成任务后通知界面线程，于是界面线程将任务结果更新到程序界面。采用工作线程设计的程序会显得有更好的响应性。

让 CPU 资源在不同任务间分配的更均衡

设想有一个接受客户端请求的服务端。假设其中一个客户端发送了一个请求，处理该请求耗时很长——比如要 10 秒。如果服务端只用一个线程来处理所有任务，那么直到该耗时请求被完全处理完，所有后面的请求都必须等待。

通过将 CPU 时间分配到多个线程，并且让 CPU 在线程之间切换的方式，多个线程便可以更公平的共享 CPU 执行时间。这样即使其中某个请求很耗时，其他的快速请求可以和这个耗时请求并行执行。当然，这会让耗时请求处理的更慢，因

为该请求无法再独占 CPU 来处理自己的任务了。即便如此，其他请求等待的时间会更短，因为它们不用再等耗时任务完成就可以处理了。而假如只有一个耗时请求待处理，那该任务仍然可以独占 CPU。

（三）：多线程的代价（程序设计，上下文切换，资源消耗）

文章目录

把程序从单线程转变到多线程并非仅仅带来好处，也需付出代价。我们不应该一味的让程序支持多线程，而只应在权衡了利大于弊的情况使用多线程。当我们不能确定时，可以先试着衡量下程序的性能和响应性，而不是凭空猜想。

多线程的设计更复杂

虽然多线程程序的某些部分比单线程更简单，但是其他部分更复杂。多线程代码在执行时需要特别注意共享数据的访问。线程之间的交互远不是一直这么简单的。因没有正确线程同步而引起的错误会很难发现，复现和解决。

多线程上下文切换带来开销

CPU 从一个线程切换到另一线程执行时，需要保存当前线程的局部数据和程序指针等，还要加载下一个线程待执行的局部数据和程序指针等。这个切换称为“上下文切换”。CPU 从某个线程的执行上下文切换到另一个线程的执行上下文。

上下文切换的开销可不低，在没必要时不应在线程之间切换。

在维基百科上可以读到更多关于上下文切换的内容：

http://en.wikipedia.org/wiki/Context_switch

多线程增加了资源消耗

线程需要一定的计算机资源才能运行，除了 CPU 时间，还需一定的内存来保存局部栈，可能还需要占用操作系统的一些资源来管理线程。我们可以试试在一个程序中创建 100 个线程，这些线程不做任何事只是等待，然后看看该程序在运行时占用了多少内存。

（四），并发模型（共享状态，分离状态，并行工作机模型，流水线模型，反应/事件驱动系统，函数式并行）

目录

并发系统可以由不同的并发模型来实现。并发模型规定了系统中线程的协作方式，从而能够共同完成指定任务。不同的并发模型会将任务按不同的方式分割，而线程之间也通过不同的方式来通信和协作。本并发教程将更深入一点的讨论时下（2015 - 2019）使用最普遍的并发模型。

并发模型和分布式系统的相似之处

本文所描述的并发模型和以不同架构实现的分布式系统很相似。在并发系统中不同线程之间彼此通信。在分布式系统中进程之间彼此通信（进程可能在不同的计算机上）。实际上线程和进程大同小异，这也正是为什么不同的并发模型和不同的分布式系统往往看上去很像。

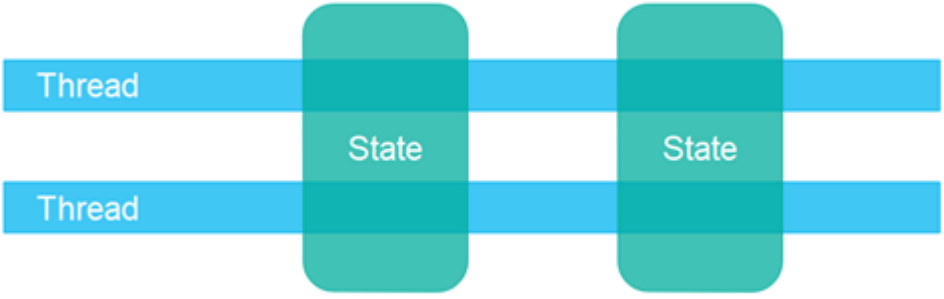
当然，分布式系统要面对更多的挑战，比如网络失败，远程计算机或进程故障等。而对于运行在大型服务器上的并发系统来说，存在类似的问题，比如 CPU 故障，网卡故障，硬盘故障等。故障的几率可能较低，但是理论上仍会发生故障。

由于并发模型和分布式系统架构很相似，所以它们之间常常可以互相借鉴。例如，工作线程之间分配工作的模型常常类似于分布式系统的负载均衡模型。类似的还有错误处理技术，比如日志记录，故障转移，任务的幂等性等。

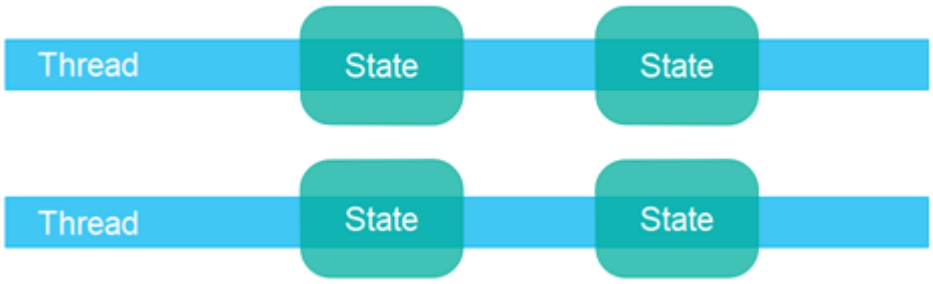
共享状态 VS 分离状态

并发模型的一个重要方面是，该让组件和线程在线程之间共享状态呢？还是各自拥有独立的状态，从不在线程之间共享呢？

共享状态表示系统的线程之间会共享一些状态。状态指的是某些数据，通常类似于一个或多个对象。线程间共享状态会导致一些问题，比如竞态条件，死锁等。当然这取决于线程是如何使用和访问共享对象的。



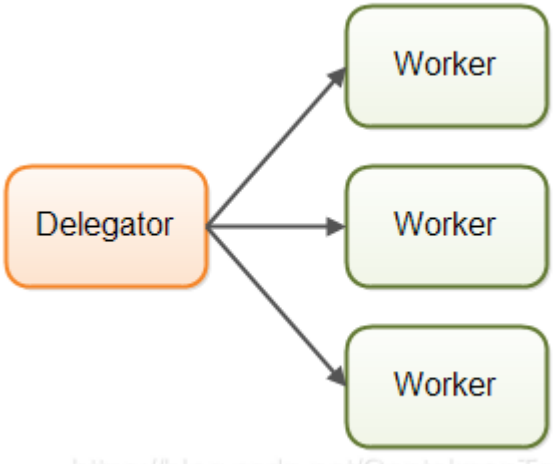
分离状态表示系统的不同线程之间不会共享任何状态。如果不同线程之间需要通信，它们可以通过交换不可变对象的方式实现，也可以通过发送对象（或数据）的拷贝的方式来实现。所以两个线程写入同一个对象（或数据、状态）的情况就不存在了，也就避免了大多数常见的并发问题。



采用分离状态的并发设计常常可以让一部分代码更容易实现和分析，因为你清楚的知道只有一个线程会写某个对象，不用担心该对象被同时访问。但是，使用分离状态并发来设计程序时，你可能还要统筹兼顾，考虑的更细致。我觉得这么做是值得的。我个人偏向于使用分离状态并发的设计。

并行工作机模型（Parallel Workers）

最初的并发模型我把它叫做并行工作机模型，它把接到的工作分配给不同的工作机。下面是并行工作机并发模型的示意图。



在并行工作机并发模型中，有个委派机负责把接到的任务分发给不同的工作机。每个工作机完成整个工作。工作机在不同的线程（可能在不同的 CPU）中并行的工作。

假如在汽车工厂中实现这个并行工作机模型，每辆车会由某个工人生产。工人会按照汽车的生产规范，从头到尾把这辆车制造出来。

并行工作机并发模型是 java 程序中最常用的并发模型，虽然这已经在变化了。在 java.util.concurrent 包中的很多并发工具类是以此模型而设计的。你也能在企业版 java 应用服务器中看到该模型的踪迹。

并行工作机模型的优点

并行工作机并发模型的优点是容易理解。你只需增加更多的工作机就可以增加应用的并行能力。

例如你要实现一个网络爬虫，你可以用不同数量的工作机来爬取一定量的页面，看用多少数量的工作机爬取的时间最短（也就意味着性能最好）。由于网络爬取是 IO 密集型作业，最终很可能每个 CPU 或核心只需开几个线程。一个 CPU 只开一个线程会太少，因为 CPU 大部分时间在等待下载数据，处于空闲状态。

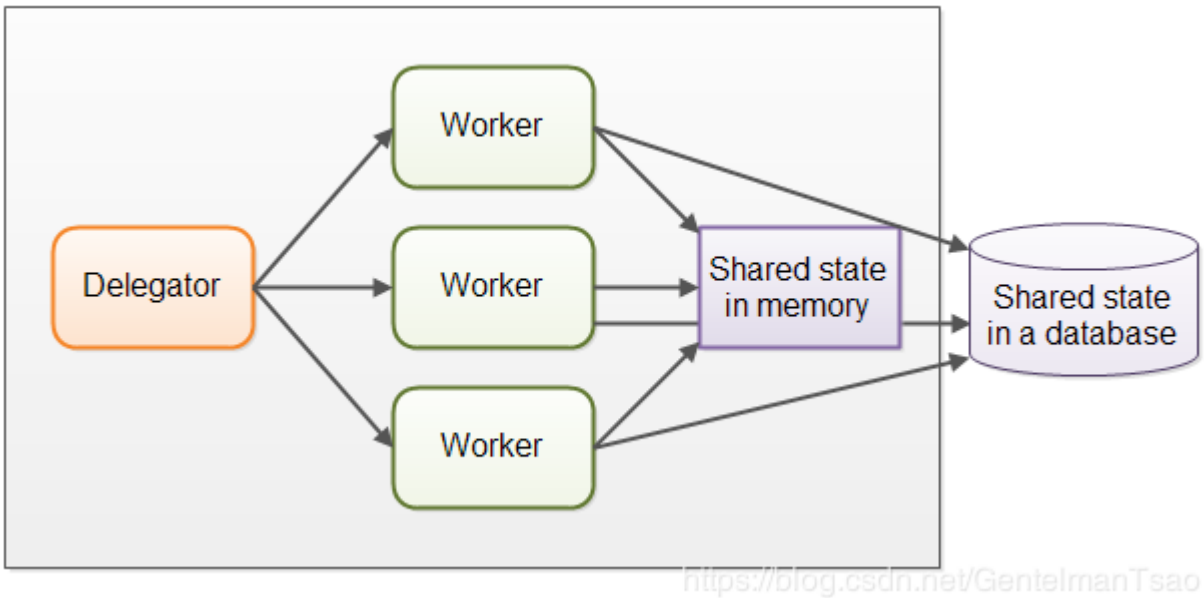
并行工作机模型的缺点

虽然表面上简单，但是并行工作机并发模型也存在一些缺点。我会在下面的章节中阐明最明显的缺点。

共享状态会变得更复杂

实际上的并行工作机并发模型要比上面示例中的更复杂一些。共享工作机经常需要访问某些类型的共享数据，这些数据处于内存或者某个共享数据库中。这种情

况让并行工作机并发模型变的复杂，如下图所示：



某些共享状态存在于诸如工作队列的通信机制中，而另一些共享状态则是业务数据，缓存数据，数据库连接池等。

一旦并行工作机并发模型中引入了共享状态，它就开始变得复杂。线程在访问共享数据时，需要确保一个线程所做的修改对其他线程可见（推送到主内存，而不是仅仅停留在执行线程的 CPU 缓存中）。线程需要避免竞态条件、死锁和许多其他共享状态并发问题。

此外，当线程因访问共享数据结构而彼此等待时，将会丢失部分并行能力。许多并发数据结构是阻塞的，这意味着一个或有限的线程集可以在任意时间访问它们。这可能导致对这些共享数据结构的争用。激烈的争用本质上将导致访问共享数据结构的部分代码要顺序执行。

现代非阻塞并发算法可以减少竞争，提高性能，但非阻塞算法很难实现。

另一种选择是持久化数据结构。持久化数据结构在修改时始终保留其以前的版本。因此，如果多个线程指向同一个持久性数据结构，并且某个线程对其进行了修改，则该线程将获得新结构的引用。所有其他线程都保留对旧结构的引用，该结构仍然保持不变，因此保持一致。Scala 编程包含了几个持久化数据结构。

虽然持久化数据结构很好的解决了并发地修改共享数据结构，但它往往不能很好地执行。

例如，持久化列表将所有新元素添加到列表的头部，并返回新元素的引用（该元素又指向列表的其余部分）。所有其他线程仍保留列表中前一个元素的引用，所以对于这些线程，列表看上去没有改变。它们看不到新添加的元素。

这种持久性列表是用链表实现的，但是链表在现代计算机上的性能并不好。列表中的每个元素都是一个单独的对象，这些对象可以分散在计算机内存的各个地方。当今的 CPU 在顺序访问数据方面要快得多，所以在当代计算机上，列表用数组实现会有更好的性能。数组是按顺序存储数据的，CPU 缓存可以一次将更大的数组块加载到缓存中，之后让 CPU 直接访问缓存中的数据。而对于元素分散在内存中的链表来说，这是不可能的。

无状态工作机（Stateless Workers）

共享状态可以由系统中的其他线程修改。因此，工作机每次都必须重新读取状态，才能确保在最新的状态上工作。无论共享状态保存在内存中还是外部数据库中，都需要这么做。内部不保留状态（但每次都要重新读取）的工作机称为无状态工作机。

.

在每次需要时重新读取数据会变慢。如果状态存储在外部数据库中，会变得更慢。

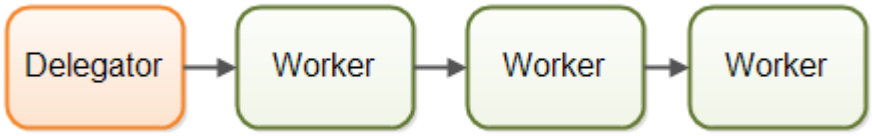
任务次序是不确定的

并行工作机模型的另一个缺点是任务执行顺序不确定。无法保证哪些任务是先执行还是最后执行。任务 A 可以在任务 B 之前交给工作机，但任务 B 可能在任务 A 之前执行。

并行工作机模型天然的不确定性导致很难随时预测系统状态，也更难（如果可行）保证一项任务先于另一项任务。

流水线模型（Assembly Line）

第二种并发模型我把它称为流水线并发模型。我选择这个名字只是为了配合先前的"并行工作机"的比喻。其他开发者根据平台或者社区使用其他名字（例如反应式系统或事件驱动系统）。下面是流水线并发模型的示意图：



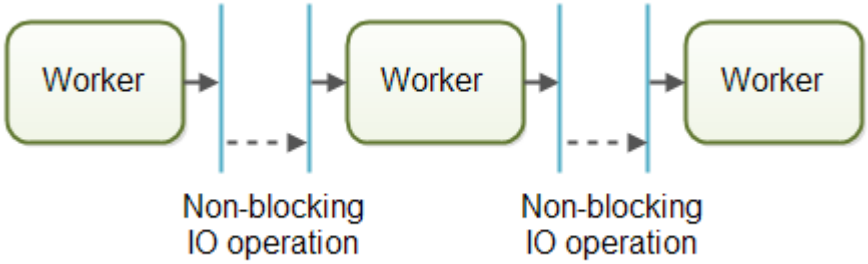
工作机像工厂装配线上的工人一样组织起来，每个工人只完成全部工作的一部分。当该部分完成时，工人将工作转发给下一个工人。

每个工作机都在自己的线程中运行，并且与其他工作机不共享任何状态。有时这也称为无共享并发模型。

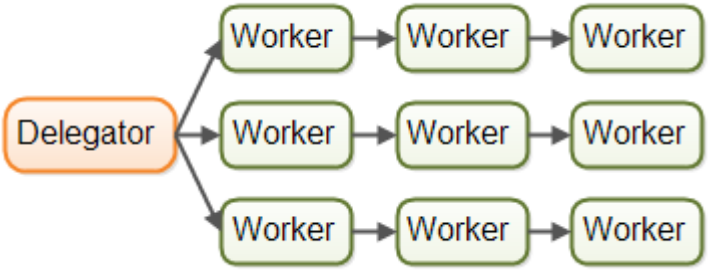
使用流水线并发模型的系统通常设计为使用非阻塞 IO。非阻塞 IO 意味着当工作进程启动 IO 操作（例如从网络连接读取文件或数据）时，工作进程不会等待 IO

调用完成。IO 操作很慢，因此等待 IO 操作完成是在浪费 CPU 时间。CPU 这时可以做别的事情。IO 操作完成后，IO 操作的结果（例如读到的数据或数据写入的状态）将传递给另一个工作机。

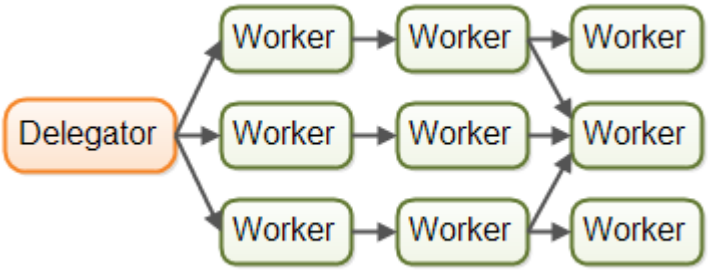
对于非阻塞 IO，IO 操作决定了工作机之间的边界。工作机尽其所能的工作，直到不得不启动 IO 操作，然后它就不再控制任务了。当 IO 操作完成时，流水线上的下一个工作机继续处理该任务，直到该任务也必须启动 IO 操作等。



实际上，这些任务可能不会沿着一条流水线进行。由于大多数系统可以执行多个任务，根据需要完成的任务，任务从一个工作机流向另一个工作机。实际上可能有很多不同的虚拟流水线在同时进行，这就是现实中流水线系统工作流程的样子。



任务甚至可以转发给多个工人进行并发处理。例如，任务可以转发给任务执行器和任务日志。此图说明了所有三个流水线将其任务转发给同一个工作机（中间装配线中的最后一个工作机）来完成任务：



流水线可能会变得更加复杂。

反应/事件驱动系统（Reactive, Event Driven Systems）

使用流水线并发模型的系统有时也称为反应系统，或事件驱动系统。系统的工作机对系统中发生的事件做出反应，这些事件来自外部世界，或者由其他工作机发出。事件可以是传入的 HTTP 请求，或者某个文件完成内存加载等。

在编写本文时，已经有许多有趣的反应/事件驱动平台，以后会有更多。下面这些似乎更受欢迎：

Vert.x

Akka

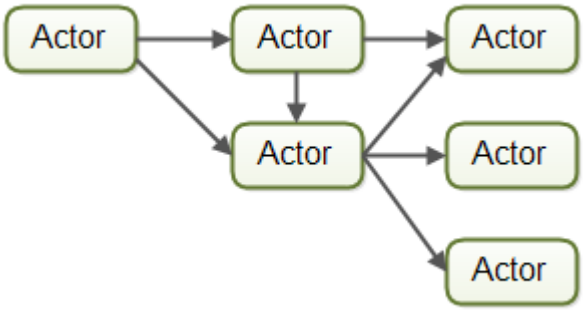
Node.JS (JavaScript)

就我个人而言，我觉得 Vert.x 非常有趣（特别是对于像我这样的 Java/JVM 老油条来说 ——**译者注：原文为 especially for a Java / JVM dinosaur like me**）。

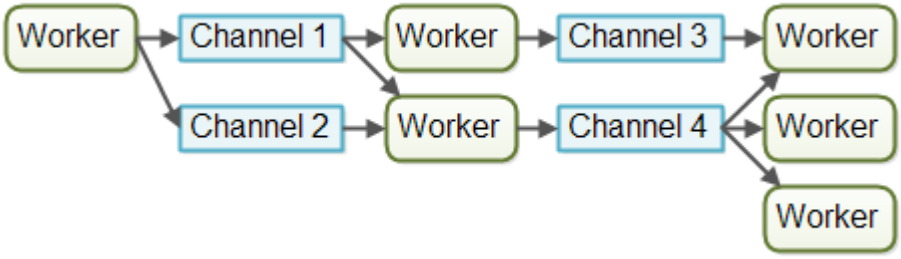
参与者与通道（Actors vs. Channels）

参与者和通道是流水线（或反应/事件驱动）模型的两个类似示例。在参与者模型中，每个工作机都称为参与者。参与者可以直接相互发送消息，消息是异步发送和处理的。如前所述，参与者可用于实现一个或多个工作流水线。下面是参与

者模型的示意图：



在通道模型中，工作机之间不直接沟通。相反，他们在不同的通道上发布消息（或事件）。其他工作机可以在这些通道上监听消息，而发送者不知道谁在监听。下面是通道模型的示意图：



在撰写本文时，通道模式对我来说似乎更加灵活。某个工作机不需要知道哪些工作机后续将在流水线上处理任务。它只需知道向哪个通道转发工作 （或给哪个通道发送消息，等等）。通道上的侦听器在注册和取消时不会影响到写通道的工作机。这使得工作机之间的耦合更加松散。

流水线模型的优点

与并行工作机模型相比，流水线并发模型有几个优点。我将在下面的章节中介绍最大的优点。

不共享状态

工作机与其他工作机不共享任何状态，这意味着在实现它们时，所有在并发访问共享状态上会出现的问题，我们都不用再考虑了。这使得工作机更容易实现。你在实现一个工作机时，这个工作机就好像是唯一一个执行该任务的线程—本质上是一个单线程实现。

有状态的工作机

由于工作机知道其他线程不会修改他们的数据，工作机可以是有状态的。我的意思是，他们可以将需要操作的数据保存在内存中，只需将改动写回最终的外部存储系统。因此，有状态的工作机通常比无状态的工作机更快。

更好的硬件整合

单线程代码的优点是，它通常更符合底层硬件的工作方式。首先，当你认为代码会以单线程模式执行时，通常可以创建优化更好的数据结构和算法。

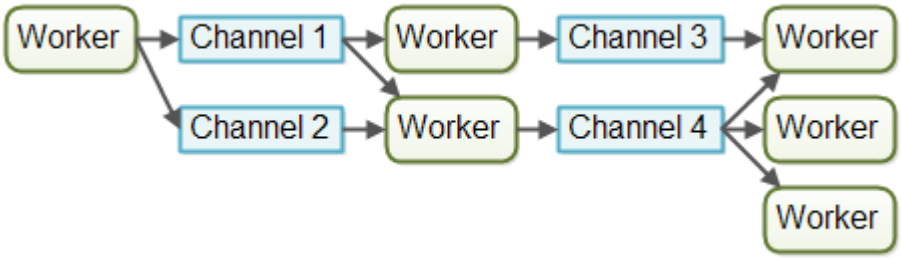
其次，单线程有状态工作机可以在内存中缓存数据，如上所述。当数据被缓存在内存中时，该数据也更有可能被缓存在执行线程的 CPU 缓存中。这使得访问缓存数据的速度更快了。

代码的编写方式天然的受益于底层硬件的工作方式，我称之为硬件一致性。一些开发者称之为机械同情（译者注：原文是 mechanical sympathy，直译为机械同情）。我更喜欢硬件一致性这个词，因为计算机很少有机械部件，在这种情况

下，“同情”一词隐喻为“更好地匹配”，而我相信“一致”一词传达得相当好。无论如何，这是吹毛求疵。你可以用你喜欢的任何术语。

可以进行任务排序

我们可以根据流水线并发模型，以保证任务有序的方式来实现一个并发系统。任务有序使我们更容易推测某个时间点的系统状态。不仅如此，还可以将所有传入的任务写入日志。进而可以使用此日志从头开始重建系统状态，以防系统的某部分出现故障。任务以特定顺序写入日志，此顺序会成为确定的任务顺序。下面是这样一个设计的样子：



（**译者注：上图看上去有错，应该是原文配图出错了，待原文修正后我再更新**）

虽然确定的任务顺序实现起来不一定容易，但往往是可行的。如果可行，它可以大大简化备份、恢复数据、复制数据等任务，因为这些都可以通过日志文件来完成。

流水线模型的缺点

流水线并发模型的主要缺点是，一个任务通常分布在多个工作机上执行，从而分布在项目中的多个类上。因此，很难确切地看到某个任务有哪些代码在执行。

编写代码也可能会更困难。工作机的代码有时编写成回调处理。于是代码中出现层层嵌套的回调处理，一些开发者把这称为回调地狱。回调地狱指的是很难在所有回调中跟踪代码真正在做什么，也很难确保每个回调都能访问所需的数据。

而使用并行工作机并发模型，这个问题就简单了。你可以打开工作机代码并读取从头到尾的执行代码。当然，并行工作机代码也可以分布在许多不同的类上，但是通常从代码中更容易读到执行顺序。

函数式并行（Functional Parallelism）

最近（2015）讨论较多的是第三种并发模型：函数式并行。

函数式并行的基本思想是使用函数调用来实现程序。函数可以看作是相互发送消息的“代理”或“参与者”，就像在流水线并发模型（也称为反应式或事件驱动系统）中的一样。一个函数调用另一个函数时类似于发送一个消息。

传递给函数的所有参数都会被复制，因此任何在接收函数之外的实体都不能操作数据。这种复制对于避免共享数据上的竞态条件至关重要。这使得函数的执行类似于原子操作。每个函数调用的执行都可以独立于任何其他函数。

当函数调用可以独立执行时，每个函数调用就可以在单独的一个 cpu 上执行。这意味着，一个用函数式实现的算法可以在多个 cpu 上并行执行。

在 Java7 中有了 Java.util.concurrent 包，其中包含了 ForkAndJoinPool，它可以帮助你实现类似于函数式并行的功能。在 Java 8 中有了并行流，它可以帮助你把大型集合的迭代并行化。要注意的是，有些开发人员对 ForkAndJoinPool 持批评态度（可以在我的 ForkAndJoinPool 教程中找到批评的链接）。

函数式并行的难点在于要弄清楚哪些函数调用要并行化。在 CPU 之间协调函数调用会带来开销。一个函数完成的工作单元需要达到一定的大小，才值得这样的开销。如果函数调用非常小，那么若把它们并行化实际上可能比单线程、单 CPU 执行的更慢。

根据我的理解（虽不完美），你可以使用一个反应式的、事件驱动的模型来实现一个算法，并将工作分解成类似于函数式并行所实现的那样。在我看来，使用事件驱动模型，你可以更准确地控制要并行化的内容和数量。

另外，只有当某个任务是程序当前执行的唯一任务时，将该任务拆分到多个 cpu 上所产生的协调开销才有意义。而如果系统同时执行多个其他任务（如 web 服务器、数据库服务器和许多其他系统），那么尝试并行一个任务是没有意义的。计算机中的其他 CPU 终将会忙于处理其他任务，因此没有理由试图用一个较慢的、函数式并行的任务来干扰它们。使用流水线（反应式）并发模型很可能更好，因为它具有更少的开销（以单线程模式顺序执行），并且更好地符合底层硬件的工作方式。

哪种并发模型最好?

那么，哪种并发模型更好呢?

通常情况下，答案取决于系统要做什么。如果你的任务本来就是并行的、独立的并且不需要共享状态，那么你可以使用并行工作机模型来实现你的系统。

然而，许多工作并不是自然而然并行和独立的。对于这些类型的系统，我相信流水线并发模型比并行工作机模型有更多的优点而不是缺点。

你甚至不必自己编写所有的流水线基础代码。像 Vert.x 这样的新平台已经为你实现了很多这样的功能。对于我来说，我将试着在 Vert.x 这样的平台上设计下一个项目。我觉得 Java EE 已经没有优势了。

（五）：类单线程（单线程扩展，无共享状态，负载分配，线程通信）

文章目录

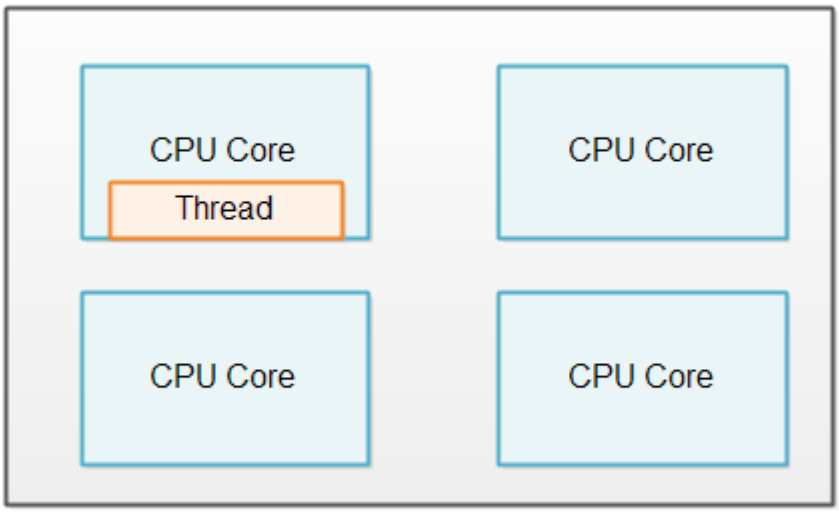
类单线程是这样一种并发模型，它把一个单线程系统扩展到 N 个单线程系统，结果是 N 个单线程系统并行运行。

类单线程系统不是纯粹的单线程系统，因为它包含多个线程。但是，每个线程都像单线程系统一样运行。因此称之为类单线程而不是单线程。

为什么要用单线程系统?

你可能好奇，为什么时至今日还有人会设计单线程系统。单线程系统之所以受欢迎，是因为其并发模型比多线程系统简单得多。单线程系统不与其他线程共享任何状态（对象/数据）。这使得单线程能够使用非并发数据结构，并更好地利用 CPU 和 CPU 缓存。

然而，单线程系统没有充分利用现代 CPU。一个现代的 CPU 通常会有 2，4，6，8 或更多个核心。每个核心都作为一个单独的 CPU 运行。单线程系统只能使用其中一个核心，如下所示：

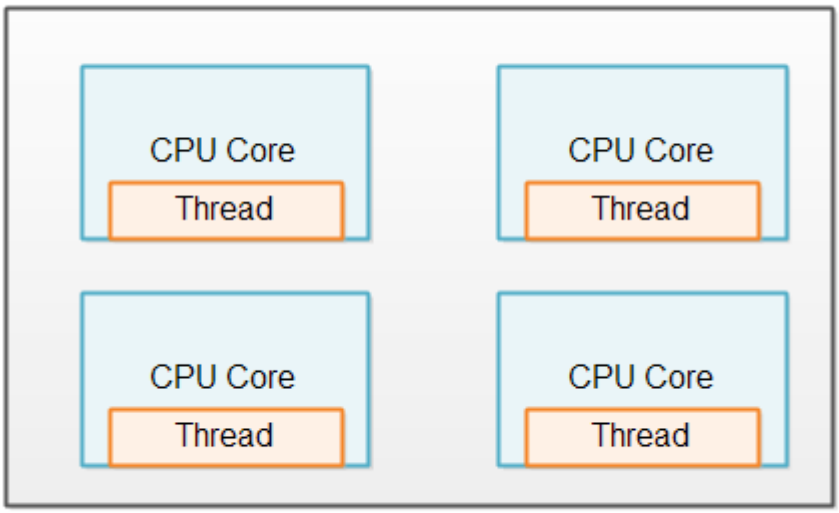


类单线程：单线程的扩展

为了充分利用 CPU 中的所有核心，可以扩展单线程系统来利用整个计算机。

每个 CPU 一个线程

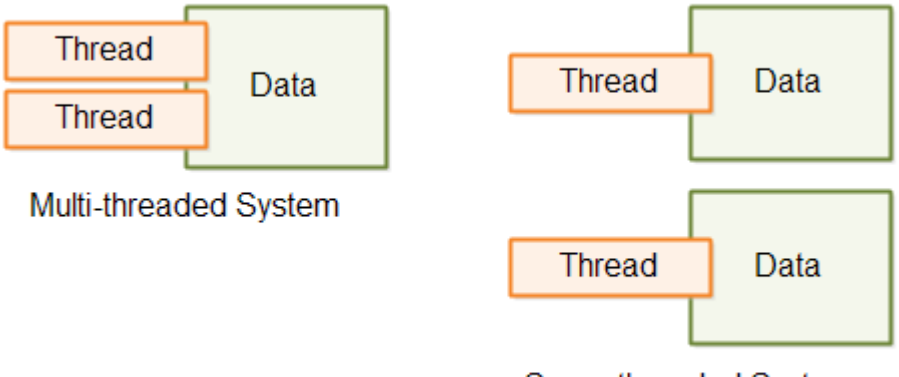
类单线程系统通常在计算机的每个 CPU 中运行一个线程。如果一台计算机包含 4 个 CPU，或者一个 CPU 有 4 个内核，那么正常情况下会运行类单线程系统的 4 个实例（4 个单线程系统）。下图显示了这一原理：



无共享状态

类单线程系统看起来类似于传统的多线程系统，因为类单线程系统中有多个线程在运行。但两者有一个微妙的区别。

类单线程系统与传统多线程系统的区别在于，类单线程系统中的线程不共享状态。既不存在用于线程间并发访问的共享内存，也不存在用于线程间共享数据的并发数据结构等。这种差异可用下图说明：



摒弃了共享状态就使得每个线程都表现的像是一个单线程系统。然而，由于类单线程系统可以包含多个单线程，因此它实际上不是“单线程系统”。由于没有更好的名称，我发现将这样的系统称为类单线程系统比称之为“具有单线程设计的多线程系统”更准确。类单线程这个名字更容易说，也更容易理解。

类单线程基本上意味着数据处理停留在同一个线程内，并且类单线程系统中没有线程同时共享数据。有时这也被称为无共享状态并发，或分离状态并发。

负载分配

显然，类单线程系统需要在运行的单线程实例之间共享工作负载。如果只有一个线程分配了工作，系统实际上是单线程的。

如何在线程之间分配负载取决于系统的设计。我将在下面的章节中介绍一些内容。

单线程微服务

如果系统由多个微服务组成，则每个微服务都可以在单线程模式下运行。当你在同一台计算机上部署多个单线程微服务时，每个微服务可以在单独的 CPU 上运行一个线程。

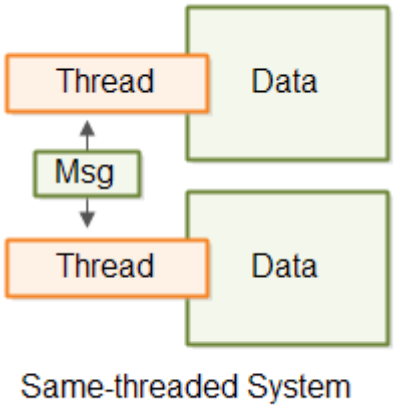
微服务本质上不共享任何数据，因此对于类单线程系统来说，微服务是一个很好的用例。

分片数据服务

如果你的系统确实需要共享数据，或者至少需要一个数据库，那么你可以对数据库进行分片。分片意味着数据被分成多个数据库。数据通常被分割，以便所有相互关联的数据都位于同一个数据库中。例如，所有属于某个“所有者”实体的数据都将插入到同一个数据库中。不过，切分数据超出了本教程的范围，因此你需要自己搜索有关该主题的教程。

线程通信

类单线程系统中的线程通过消息传递进行通信。如果线程 A 想向线程 B 发送消息，线程 A 可以通过生成一个消息（字节序列）来实现。然后线程 B 可以复制并读取该消息（字节序列）。线程 B 通过复制消息的方式，确保它在读取消息时线程 A 无法修改该消息。消息一旦复制，线程 A 就无法访问该消息的拷贝了。通过消息传递进行的线程通信如下所示：



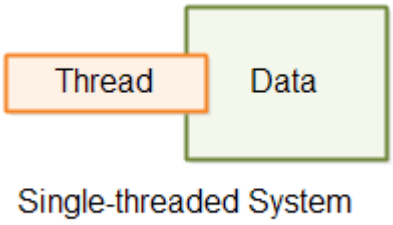
只要适合你的系统，线程可以通过队列、管道、unix 套接字、TCP 套接字等进行通信。

更简单的并发模型

在类单线程系统中，每个运行在其自身线程中的系统都可以像单线程一样实现。这意味着，与线程共享状态相比，内部并发模型变得简单得多。你不必再担心并发数据结构以及此类数据结构可能导致的所有并发问题。

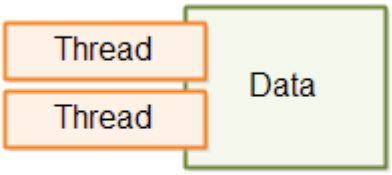
示意图

以下是单线程、多线程和类单线程系统的示意图，这样你可以更容易地了解它们之间的区别。 第一个图显示的是单线程系统。



Single-threaded System

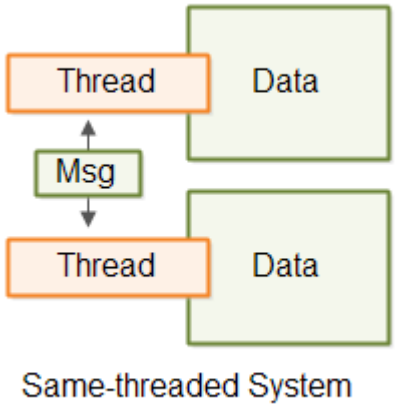
第二个图显示了多线程系统，其中线程共享了数据。



Multi-threaded System

第三个图显示了类单线程系统，其中有两个线程具有单独的数据，通过相互传递

消息进行通信。



Same-threaded System

Thread Ops for Java

Thread Ops for Java 是一个开源工具包，旨在帮助你更容易地实现分离状态的类单线程系统。它包含启动和停止单个线程的工具，以及使用单个线程实现某种程度的并发性。如果你对使用类单线程来设计应用程序感兴趣，那么你可能会对 Thread Ops 感兴趣。

（六）：并发和并行

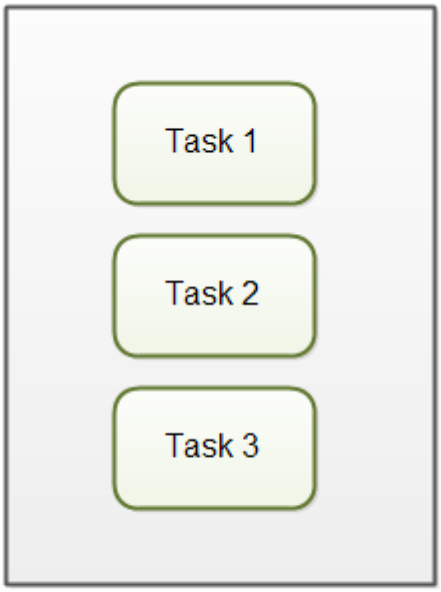
文章目录

并发和并行这两个术语通常用于多线程程序。但并发和并行到底是什么意思，这两个术语又有什么不同呢?

我花了一些时间才最终理解并发和并行之间的区别。因此，我打算在本 Java 并发教程中添加一篇关于并发与并行的文章。

并发

并发意味着一个应用程序同时（并发）进行多个任务。当然，如果计算机只有一个 CPU，那么应用程序可能不会真正同时进行多个任务，而是在一段时间内处理应用程序内部的多个任务。在开始下一个任务之前，它不会完成一整个任务。相反，CPU 在不同的任务之间切换，直到任务完成。

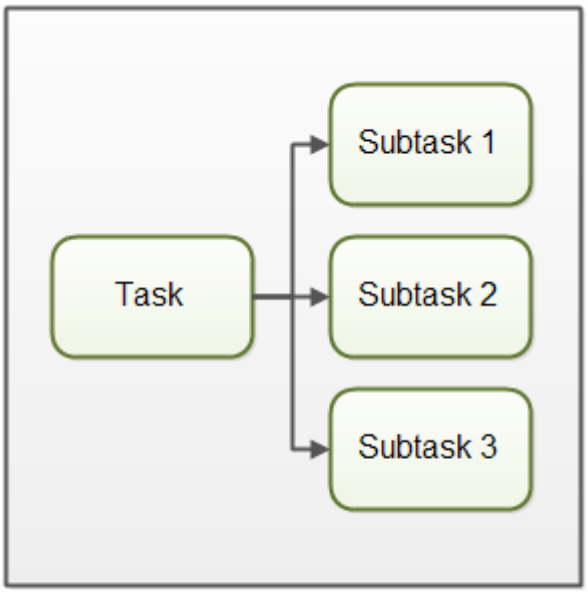


Concurrency:
Multiple tasks makes progress at the same time.

即使应用内部只有一个线程在运行，该应用也可以是并发的。顺便说一下，这是我们（Nanosai）Thread Ops 工具包的目标之一。

并行

并行的意思是应用程序将其任务拆分成更小的子任务，这些子任务可以并行处理，例如在多个 CPU 上真正的同时执行。



Parallelism:
Each task is broken into subtasks which can be processed in parallel.

要实现真正的并行，应用程序必须运行多个线程，或者至少能够把任务安排在其他线程、进程、CPU 或显卡等中执行。

并发与并行的详细对比

如你所见，并发与应用程序处理多个工作任务的方式有关。应用程序可以一次（顺序）处理一个任务，也可以同时（并发）处理多个任务。

而并行与应用程序处理每个单独任务的方式有关。应用程序可以从头到尾连续地处理任务，或者将任务拆分成可以并行完成的子任务。

如你所见，应用程序可以是并发且非并行的。这意味着它同时处理多个任务，但线程一次只执行一个任务。不存在并行线程或 CPU 中并行执行的任务。

应用程序也可以是并行且非并发的。这意味着应用程序一次只能处理一个任务，并且该任务被分解为可以并行处理的子任务。但是，每个任务（以及子任务）要先完成，才能再拆分和并行执行下一个任务。

此外，应用程序可以是非并发且非并行的。这意味着它一次只能处理一个任务，并且任务也不会分解为并行执行的子任务。

最后，应用程序也可以既是并发又是并行的，因为它既同时处理多个任务，又将每个任务分解为并行执行的子任务。但是，在这种情况下，并发和并行可能不会带来什么好处，因为计算机中的 CPU 光是处理并发或者并行就已经够忙的了。两者结合起来可能只会带来很小的性能提升甚至性能损失。在盲目采用并发并行模型之前，一定要先进行分析和度量。

（七）：创建并启动 java 线程（Thread，Runnable，线程名称 ， 暂停和停止线程）

文章目录

Java 线程就像一个虚拟 CPU，可以在 Java 应用程序中执行 Java 代码。当 Java 应用程序启动时，它的 main（）方法由主线程执行，主线程是 Java 虚拟机为运行应用程序而创建的一个特殊线程。你可在应用程序内部创建和启动更多的线程，这些线程可以与主线程并行执行应用程序的部分代码。

和其他 Java 对象一样，Java 线程也是对象。线程是 java.lang.Thread 类的实例，或者是该类的子类的实例。除了作为对象之外，java 线程还可以执行代码。在本 Java 线程教程中，我将讲解如何创建和启动线程。

创建和启动线程

在 Java 中创建线程的过程如下：

```
Thread thread = new Thread();
```

要启动 Java 线程，需要调用其 start（）方法，如下所示：

```
thread.start();
```

这个例子没有指定线程要执行的代码。因此，线程启动后会立即停止。

有两种方法可以指定线程应该执行的代码。第一种方法是创建线程的子类并重写 run（）方法。第二种方法是实现 Runnable（java.lang.Runnable）的对象，并将该对象传递给线程构造函数。下面将介绍这两种方法。

Thread 子类

指定线程该运行什么代码的第一种方法，是创建 Thread 的子类并重写 run（）方法。run（）方法是线程在调用 start（）之后执行的操作。下面是创建 Java Thread 子类的示例：

```
public class MyThread extends Thread {


    public void run(){
        System.out.println("MyThread running");
    }
}
```

要创建和启动上述线程，可以执行以下操作：

```
MyThread myThread = new MyThread();
myTread.start();
```

一旦线程启动，start（）调用就返回了，它不会等到 run（）方法完成。run（）方法就像是在另一个 CPU 中执行一样。当 run ( ) 方法执行时，它将输出文本“MyThread running”。

你还可以创建 Thread 的一个匿名子类，如下所示：

```
Thread thread = new Thread(){
    public void run(){
        System.out.println("Thread Running");
    }
}

thread.start();
```

这个例子将在新线程执行 run（）方法，并输出文本“Thread running”。

实现 Runnable 接口

指定线程该运行什么代码的第二种方法，是创建一个实现 java.lang.Runnable 接口的类。实现 Runnable 接口的 Java 对象可以由 Java Thread 执行。本教程之后会演示这种实现方法。

Runnable 接口是 Java 平台附带的标准 Java 接口。Runnable 接口只有一个 run ( ) 方法。以下是 Runnable 接口的基本代码：

```
public interface Runnable() {
```

```
    public void run();

}
```

无论线程在执行时应该做什么，都必须包含在 run（）方法的实现中。有三种方法可以实现 Runnable 接口：

1.创建一个实现 Runnable 接口的 Java 类。

2.创建一个实现 Runnable 接口的匿名类。

3.创建一个实现 Runnable 接口的 Java Lambda。

这三个选项将在下面的章节中进行说明。

Java 类实现 Runnable

实现 Java Runnable 接口的第一种方法是创建自己的 Java 类来实现 Runnable 接口。下面是实现 Runnable 接口的自定义 Java 类的示例：

```
public class MyRunnable implements Runnable {


    public void run(){
        System.out.println("MyRunnable running");
    }
}
```

这个 Runnable 的实现所做的是打印出文本“MyRunnable running”。打印该文本后，run（）方法退出，运行 run（）方法的线程也将停止。

Runnable 的匿名实现

你还可以创建 Runnable 的匿名实现。下面是实现 Runnable 接口的匿名 Java 类的示例：

```
Runnable myRunnable =
        new Runnable(){
            public void run(){
                System.out.println("Runnable running");
            }
        }
```

除了是一个匿名类之外，这个示例与使用自定义类实现 Runnable 接口的示例非常相似。

Runnable 的 Java Lambda 实现

实现 Runnable 接口的第三种方法是创建 Runnable 接口的 Java Lambda 实现。之所以能这么做是因为 Runnable 接口只有一个未实现的方法，因此它实际上（尽管可能并非特意）是一个功能性 Java 接口。

下面是实现 Runnable 接口的 Java lambda 表达式的示例：

```
Runnable runnable =
            () -> { System.out.println("Lambda Runnable running"); };
```

- 1

使用 Runnable 启动线程

要让 run（）方法由线程执行，可以实现 Runnable 接口的类、匿名类或 lambda 表达式，并将它们的实例传递给 Thread 的构造函数。下面是具体做法：

```
Runnable runnable = new MyRunnable(); // 或者是匿名类，Lambda...

Thread thread = new Thread(runnable);
thread.start();
```

当线程启动时，它将调用 MyRunnable 实例的 run（）方法，而不是执行它自己的 run（）方法。上面的例子会输出文本“MyRunnable running”。

用 Thread 子类还是用 Runnable?

没人规定这两种方法中哪一种最好。两种方法都有效。不过，就我个人而言，我更喜欢实现一个 Runnable，并将该实现的实例传给 Thread 实例。当线程池执行 Runnable 时，很容易将 Runnable 实例排队，直到池中的线程空闲。这对于 Thread 子类来说有点困难。

有时你可能既需要实现 Runnable 又需要实现 Thread 子类。例如，假如要创建可以执行多个 Runnable 的 Thread 子类。在实现线程池时通常会出现这种情况。

常见陷阱：调用 run（）而不是 start（）

创建和启动线程时，常见的错误是调用线程的 run（）方法而不是 start（），如下所示：

```
Thread newThread = new Thread(MyRunnable());
```

```
newThread.run();  //应该是 start();
```

- 1

起初，你可能不会察觉到有什么不对，因为 Runnable 的 run（）方法执行的跟你预想的一样。但是，它不是由你刚刚创建的新线程执行的。相反，run（）方法由创建线程的线程执行，也就是执行上述两行代码的线程。要让新创建的线程 newThread 调用 MyRunnable 实例的 run（）方法，必须调用 newThread.start（）方法。

线程名称

当你创建一个 Java 线程时，你可以给它起个名字，用来帮助你区分不同的线程。例如，如果多个线程写入 System.out，则可以方便地查看是哪个线程写入了文本。下面是一个例子：

```
Thread thread = new Thread("New Thread") {
    public void run(){
       System.out.println("run by: " + getName());
    }
};



thread.start();
System.out.println(thread.getName());
```

注意字符串“New Thread”作为参数传递给 Thread 构造函数。此字符串是线程的名称。该名称可以通过线程的 getName（）方法获得。用 Runnable 实现时，也可以将名称传递给线程。像这样：

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");

thread.start();
System.out.println(thread.getName());
```

但是请注意，由于 MyRunnable 类不是线程的子类，因此它不能访问执行它的线程的 getName（）方法。

Thread.currentThread（）

Thread.currentThread（）方法返回一个引用，该引用指向在执行 currentThread（）的线程实例。这样你就可以访问在执行特定代码块的 Java 线程对象。下面是使用 Thread.currentThread（）的示例：

```
Thread thread = Thread.currentThread();
```

一旦有了对线程对象的引用，就可以对其调用方法。例如，可以获取当前执行代码的线程的名称，如下所示：

```
String threadName = Thread.currentThread().getName();
```

Java 线程示例

这里有一个小例子。首先，它打印出执行 main（）方法的线程的名称。该线程由 JVM 分配。然后启动 10 个线程，并给它们一个数字作为名称("" + i)。之后每个线程都会打印出其名称，然后停止执行。

```
public class ThreadExample {


   public static void main(String[] args){
      System.out.println(Thread.currentThread().getName());
      for(int i=0; i<10; i++){
         new Thread("" + i){
            public void run(){
               System.out.println("Thread: " + getName() + " running");
            }
         }.start();
      }
   }
}
```

要注意的是，即使线程是按顺序（1、2、3 等）启动的，它们也未必依次执行，也就是线程 1 未必是第一个将其名称写入 System.out 的线程。这是因为线程原则上是并行执行的，而不是顺序执行的。JVM 和操作系统分别或共同决定线程的执行顺序。此顺序不必与它们启动时的顺序相同。

暂停线程

线程可以通过调用静态方法 thread.sleep（）暂停自身。sleep（）以毫秒为参数。sleep（）方法将尝试休眠指定毫秒后恢复执行。Thread 的 sleep（）虽不是 100%精确，但已经很好了。下面的示例通过调用 Thread 的 sleep（）方法暂停 Java 线程 3 秒（3.000 millliseconds）：

（译者注：原文错误，实际应是暂停 10 秒，即 10，000 毫秒）

```
try {
    Thread.sleep(10L * 1000L);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

执行上述 Java 代码的线程将休眠大约 10 秒（10.000 毫秒）。

停止线程

想要停止 Java 线程，需要在线程的实现代码中做些准备工作。Java Thread 类包含 stop（）方法，但已弃用。原始的 stop（）方法无法保证线程被停止后会处于何种状态。也就是说，线程在执行期间访问的所有 Java 对象都将处于未知状态。如果应用程序中的其他线程也访问了同样的对象，则应用程序可能会出现意料之外和不可预知的错误。

不要调用 stop（）方法，你必须自己实现停止线程的代码。下面是一个实现 Runnable 的类的示例，该类包含一个额外方法 doStop()，用于向 Runnable 发出停止的信号。Runnable 将检查此信号，并在准备好时停止。

```
public class MyRunnable implements Runnable {

    private boolean doStop = false;

    public synchronized void doStop() {
        this.doStop = true;
    }

    private synchronized boolean keepRunning() {
        return this.doStop == false;
    }

    @Override
    public void run() {
        while(keepRunning()) {
            // 持续做线程该做的工作
            System.out.println("Running");

            try {
```

```
                Thread.sleep(3L * 1000L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
}
```

注意 doStop（）和 keepRunning（）方法。doStop（）是用于给别的线程调用的，而不是由执行 MyRunnable 的 run（）方法的线程调用。keepRunning（）方法由执行 MyRunnable 的 run () 方法的线程在内部调用。只要没有调用 doStop ()，keep running（）方法就会返回 true，这意味着执行 run（）方法的线程将继续运行。

下面的示例启动了一个 Java 线程，该线程执行上述 MyRunnable 类的实例，并在延迟后停止线程：

```
public class MyRunnableMain {

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread = new Thread(myRunnable);
```

```
        thread.start();


        try {
            Thread.sleep(10L * 1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }


        myRunnable.doStop();
    }
}
```

本例首先创建一个 MyRunnable 实例，然后将该实例传递给一个线程并启动该线程。之后执行 main（）方法的线程（也就是主线程）休眠 10 秒，然后调用 MyRunnable 实例的 doStop（）方法。这会让执行 MyRunnable 方法的线程停止，因为调用 doStop（）后，keepRunning（）将返回 true。

请记住，如果你的 Runnable 要实现的不仅仅是 run () 方法 (例如，还有 stop () 或 pause（）方法），那么你就不能再使用 Java lambda 表达式创建你的 Runnable 实现。Java lambda 只能实现一个方法。相应的，你必须使用自定义类，或自定义一个继承 Runnable 的接口，该接口需要有额外的方法，并且由匿名类实现。

（八）：竞态条件和临界区

文章目录

竞态条件是可能发生在临界区内的特殊条件。临界区是由多个线程执行的一段代码，它的并发执行结果会因线程的执行顺序而有差别。

多个线程执行一个临界区，可能因线程执行的顺序不同而带来不同的结果，在这种情况下，该临界区称为含有竞态条件。术语竞态条件源于这样一个隐喻，即线程在争抢临界区，而争抢的结果会影响临界区的执行结果。

这听起来可能有点复杂，所以我将在下面的章节中详细介绍竞态条件和临界区。

临界区

在同一个应用程序中运行多个线程本身不会导致问题。问题出现在当多个线程访问同一资源时。例如，访问相同的内存（变量、数组或对象）、系统（数据库、web 服务等）或文件。

事实上，只有当一个或多个线程写入这些资源时，才会出现问题。只要资源不变，允许多个线程读取同样的资源是安全的。

下面是一个临界区 Java 代码示例，如果由多个线程同时执行，则可能会出错：

```
public class Counter {

    protected long count = 0;

    public void add(long value){
        this.count = this.count + value;
    }
}
```

假设两个线程 A 和 B 在 Counter 类的同一个实例上执行 add 方法。我们无法知道操作系统何时在两个线程之间切换。Java 虚拟机不会将 add（）方法中的代码作为单个原子指令执行，而是作为一组较小的指令执行的，类似于：

把 this.count 从内存读入寄存器中。

寄存器加上 value。

将寄存器写入内存。

观察以下线程 A 和 B 的混合执行情况：

```
this.count = 0;
```

A: 读 this.count 到寄存器 (0)

B: 读 this.count 到寄存器 (0)

B: 寄存器加上 value 2

B: 寄存器 (2) 写回内存. this.count 现在等于 2

A: 寄存器加上 value 3

A: 寄存器 (3) 写回内存. this.count 现在等于 3

两个线程希望将值 2 和 3 添加到计数器中。因此，这两个线程执行完之后，该值应该是 5。但是，由于两个线程是交错执行的，最后的结果却不一样了。

在上面列出的执行序列示例中，两个线程都从内存中读取值 0。然后，他们加上各自的值 2 和 3，并将结果写回内存。可最终 this.count 的值不是 5，而是最后一个线程写入的值。在上述情况下，它是线程 A，但它也可能是线程 B。

临界区的竞态条件

在前面示例中，add（）方法中的代码包含一个临界区。当多个线程执行此临界区时，就出现了竞态条件。

更正式地说，当两个线程竞争同一资源，且访问该资源的顺序又十分重要时，这种情况称为竞态条件。导致竞态条件的代码段称为临界区。

避免竞态条件

为了防止竞态条件发生，必须确保临界区作为原子指令执行。也就是说一旦一个线程执行它，在第一个线程离开临界区之前，其他线程都不能执行它。

通过在临界区进行适当的线程同步，可以避免竞态条件。线程同步可以使用同步的 Java 代码块来实现。线程同步也可以使用其他同步结构（例如锁）或原子变量（例如 java.util.concurrent.atomic.AtomicInteger）来实现。

临界区吞吐量

对于较小的临界区，把整个临界区作为同步块是可行的。但是，对于较大的临界区，好的做法可能是将临界区分成较小的临界区，以便允许多个线程执行每个较小的临界区。这可以减少对共享资源的争用，从而增加整个临界区的吞吐量。

这里有一个非常简单的 Java 代码示例来说明：

```
public class TwoSums {


    private int sum1 = 0;
    private int sum2 = 0;


    public void add(int val1, int val2){
        synchronized(this){
            this.sum1 += val1;
            this.sum2 += val2;
        }
    }
}
```

上例中有两个不同的 sum 成员变量，注意 add（）方法是如何增加它们的值的。为了避免竞态条件，求和被放在 Java 同步块内执行。这样实现的结果是，同一时间只有一个线程可以执行求和。

但是，由于两个求和变量彼此独立，因此可以将它们的求和拆分为两个单独的同步块，如下所示：

```
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;

    private Integer sum1Lock = new Integer(1);
    private Integer sum2Lock = new Integer(2);

    public void add(int val1, int val2){
        synchronized(this.sum1Lock){
            this.sum1 += val1;
        }
        synchronized(this.sum2Lock){
            this.sum2 += val2;
        }
    }
}
```

现在两个线程可以同时执行 add（）方法了。一个线程在第一个同步块中执行，另一个线程在第二个同步块中执行。两个同步块在不同的对象上同步，因此两个不同的线程可以独立地执行这两个块。这样就减少了等对方线程执行 add（）方法的时间。

当然，这个例子很简单。在现实中的共享资源中，临界区的分解可能要复杂得多，需要对可能的执行顺序进行更多的分析。

（九）：线程安全和共享资源（局部变量，局部对象引用，对象成员变量，线程控制逸出规则）

文章目录

可以由多个线程同时安全调用的代码称为线程安全代码。线程安全的代码不包含竞态条件。只有当多个线程更新共享资源时，才会出现竞态条件。因此，了解 Java 线程在执行时共享了哪些资源非常重要。

局部变量

局部变量存储在线程自己的堆栈中。这意味着局部变量永远不会在线程之间共享。这也意味着所有的原始局部变量都是线程安全的。以下是线程安全的原始局部变量的示例：

```
public void someMethod(){

    long threadSafeInt = 0;

    threadSafeInt++;
}
```

局部对象引用

对象的局部引用有点不同，引用本身是不共享的。但是，引用的对象并不是存储在每个线程的局部栈中，而是所有对象都存储在共享堆中。

如果局部创建的对象从不在创建它的方法之外使用，则它是线程安全的。实际上，你也可以把它传递给其他方法和对象，只要这些方法或对象不会再把它给其他线程使用。

以下是线程安全局部对象的示例：

```
public void someMethod(){

    LocalObject localObject = new LocalObject();

    localObject.callMethod();
```

```
    method2(localObject);
}

public void method2(LocalObject localObject){
    localObject.setValue("value");
}
```

本例中的 LocalObject 实例不会从该方法返回，也不会传递给 someMethod（）方法外部可访问的其他对象。每个执行 someMethod（）方法的线程都将创建自己的 LocalObject 实例并将其分配给 localObject 引用。因此，这里使用的 LocalObject 是线程安全的。

实际上，整个方法 someMethod（）都是线程安全的。即使 LocalObject 实例作为参数传递给同一个类的其他方法，或其他类中的方法，它的使用也是线程安全的。

当然，唯一的例外是，如果其中某个方法使用 LocalObject 作为参数调用，又存储了 LocalObject 实例，而存储的实例允许其他线程访问。

对象成员变量

对象成员变量（字段）与对象一起存储在堆中。因此，如果两个线程调用同一个对象实例的方法，并且此方法更新对象成员变量，则该方法不是线程安全的。下面是一个非线程安全的方法示例：

```
public class NotThreadSafe{
      StringBuilder builder = new StringBuilder();
```

```
    public add(String text){
        this.builder.append(text);
    }
}
```

- 6

如果两个线程同时调用同一个 NotThreadSafe 实例的 add（）方法，则会导致竞态条件。

例如：

```
NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{
   NotThreadSafe instance = null;

   public MyRunnable(NotThreadSafe instance){
      this.instance = instance;
   }

   public void run(){
```

```
        this.instance.add("some text");
    }
}
```

注意下两个 MyRunnable 实例是如何共享同一个 NotThreadSafe 实例的。因此，当它们调用 NotThreadSafe 实例的 add（）方法时，会导致竞态条件。

但是，如果两个线程同时调用不同实例的 add（）方法，则不会导致竞态条件。下面是在之前的示例上稍作修改：

```
new Thread(new MyRunnable(new NotThreadSafe())).start();
new Thread(new MyRunnable(new NotThreadSafe())).start();
```

现在这两个线程都有自己的 NotThreadSafe 实例，因此它们对 add 方法的调用不会相互干扰。代码不存在竞态条件了。所以，即使一个对象不是线程安全的，它仍然有避免竞态条件的使用方式。

线程控制逸出规则

当试图确定代码对某个资源的访问是否是线程安全时，可以使用线程控制逸出规则：

如果资源的创建、使用和释放（译者注：原文为 disposed，这里的意思是丢弃）

是在同一个线程的控制下，

并且永远不会逃离出该线程的控制，

则该资源的使用是线程安全的。

资源可以是任何共享资源，如对象、数组、文件、数据库连接、套接字等。在 Java 中，并不总是显式地释放对象，因此“释放”意味着丢弃或将对象的引用置空。

即使对象的使用是线程安全的，但如果该对象指向共享资源（如文件或数据库），则整个应用程序可能不是线程安全的。例如，如果线程 1 和线程 2 各自创建自己的数据库连接：连接 1 和连接 2，则每个连接本身的使用是线程安全的。但是连接指向的数据库的使用可能不是线程安全的。例如，如果两个线程都执行这样的代码：

检查是否存在记录 X

如果不存在，插入记录 X

- 1

如果两个线程同时执行此操作，并且它们正在检查的记录 X 恰好是同一个记录，则有可能两个线程最终都会插入该记录。就像这样：

线程 1 检查是否存在记录 X. Result = no

线程 2 检查是否存在记录 X. Result = no

线程 1 插入记录 X

线程 2 插入记录 X

线程在操作文件或其他共享资源时也可能发生这种情况。因此，有必要认清线程控制的对象究竟是资源，或者仅仅是资源的引用（就像数据库连接那样）。

（十）：线程安全和不变性

只有当多个线程访问同一资源，并且一个或多个线程写入该资源时，才会出现竞态条件。如果多个线程读取同一资源，则不会出现竞态条件。

我们可以使共享对象不可变，以此来确保线程之间共享的对象不会被任何线程更新，从而保证线程安全。下面是一个例子：

```
public class ImmutableValue{


  private int value = 0;


  public ImmutableValue(int value){
    this.value = value;
  }


  public int getValue(){
    return this.value;
  }
}
```

注意下 ImmutableValue 实例的构造函数是如何传递 value 的。还要注意这里没有赋值方法。一旦创建了 ImmutableValue 实例，就不能更改其值。它是不可变的。但是你可以使用 getValue（）方法读取它的值。

如果需要对 ImmutableValue 实例执行操作，可以用操作的结果值返回一个新的实例来实现。下面是一个 add 操作的示例：

```
public class ImmutableValue{


  private int value = 0;


  public ImmutableValue(int value){
    this.value = value;
  }


  public int getValue(){
    return this.value;
  }



      public ImmutableValue add(int valueToAdd){
      return new ImmutableValue(this.value + valueToAdd);
      }


}
```

注意 add（）方法用加法操作的结果返回了一个新的 ImmutableValue 实例，而不是自己加上 value。

引用不是线程安全的!

必须记住，即使一个对象是不可变的从而是线程安全的，该对象的引用也不一定是线程安全的。看看这个例子：

```
public class Calculator{

  private ImmutableValue currentValue = null;


  public ImmutableValue getValue(){

    return currentValue;

  }


  public void setValue(ImmutableValue newValue){

    this.currentValue = newValue;

  }


  public void add(int newValue){

    this.currentValue = this.currentValue.add(newValue);

  }

}
```

Calculator 类保存了一个 ImmutableValue 实例的引用。请注意，它可以通过 setValue（）和 add（）方法更改该引用。因此，即使 Calculator 类在内部使用了不可变对象，它本身却不是不可变的，因此也不是线程安全的。换言之：

ImmutableValue 类是线程安全的，但使用它（的类或方法）却不是线程安全的。当试图通过不变性实现线程安全时，需要记住这一点。

为了使 Calculator 类线程安全，可以声明 getValue（）、setValue（）和 add（）方法为 synchronized，从而解决该问题。

（十一）：Java 内存模型（内存模型，硬件内存架构，共享对象的可见性，竞态条件）

文章目录

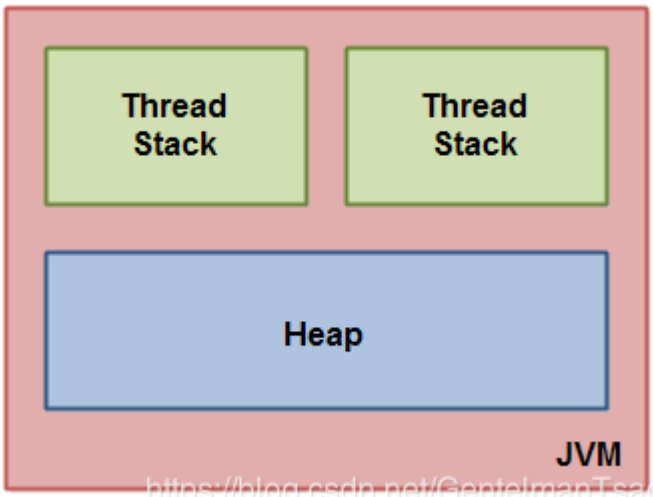
Java 内存模型规定了 Java 虚拟机使用计算机内存（RAM）的方式。Java 虚拟机是整个计算机的一种模型，所以这个模型自然包含一个内存模型，也就是 Java 内存模型。

要想设计出正确的并发程序，理解 Java 内存模型非常重要。Java 内存模型规定了不同线程用何种方式、以及何时可以看到其他线程写入共享变量的值，以及在必要时如何同步对共享变量的访问。

原始的 Java 内存模型有很多不足，所以在 Java 1.5 中 Java 内存模型得到了完善。而 Java8 仍沿用了此版本的 Java 内存模型。

Java 内部内存模型

JVM 内部使用的 java 内存模型将内存划分为线程栈和堆。此图从逻辑角度说明了 Java 内存模型：



Java 虚拟机中运行的每个线程都有自己的线程栈。线程栈的信息包含了线程调用了哪些方法以到达当前执行点。我将此称为“调用栈”。调用栈随着线程执行其代码而改变。

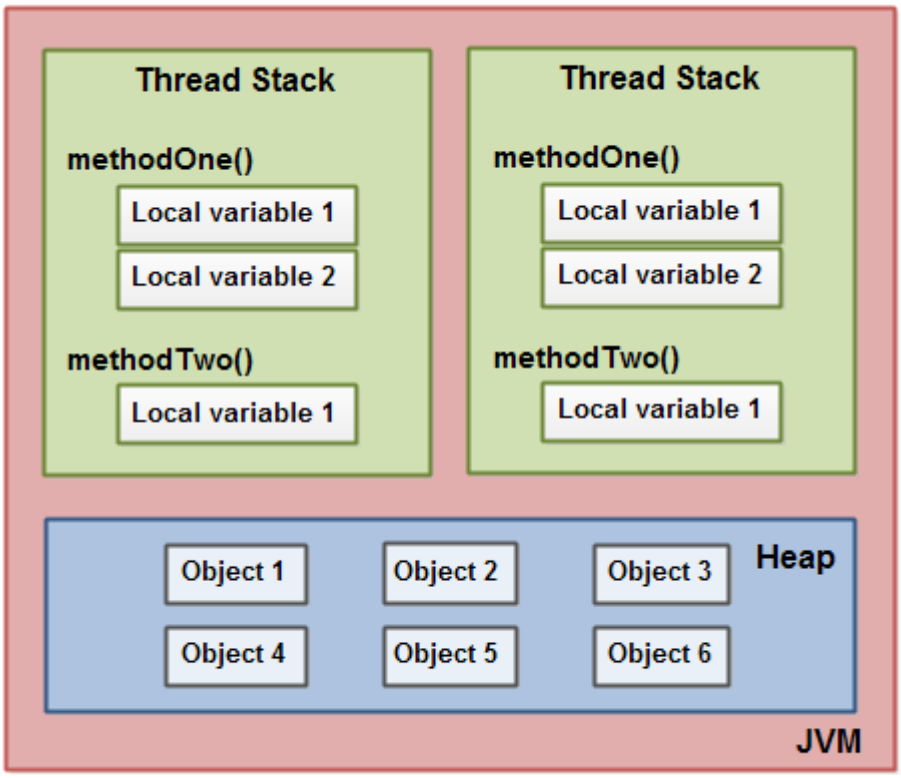
线程栈还包含执行中的每个方法（调用栈上的所有方法）的所有局部变量。一个线程只能访问它自己的线程栈。由线程创建的局部变量只对创建它的线程可见，而对所有其他线程都不

可见。即使两个线程正在执行完全相同的代码，两个线程仍将在各自的线程栈中创建该代码的局部变量。因此，每个线程都有每个局部变量自己的版本。

所有原始类型（boolean、byte、short、char、int、long、float、double）的局部变量都完全存储在线程堆栈中，因此对其他线程不可见。一个线程可以将原始变量的副本传递给另一个线程，但它不能共享原始局部变量本身。

堆包含了 Java 应用程序中创建的所有对象，而不管是哪个线程创建的对象。这包括原始类型（例如 Byte、Integer、Long 等）的对象版本。不管对象是创建并分配给本地变量，还是作为另一个对象的成员变量，对象都是存储在堆中。

下图说明了存储在线程栈上的调用栈和局部变量，以及存储在堆上的对象。



局部变量可以是基本类型，在这种情况下，它完全保留在线程栈中。

局部变量也可以是对象的引用。在这种情况下，引用（局部变量）存储在线程栈上，而对象本身则存储在堆上。

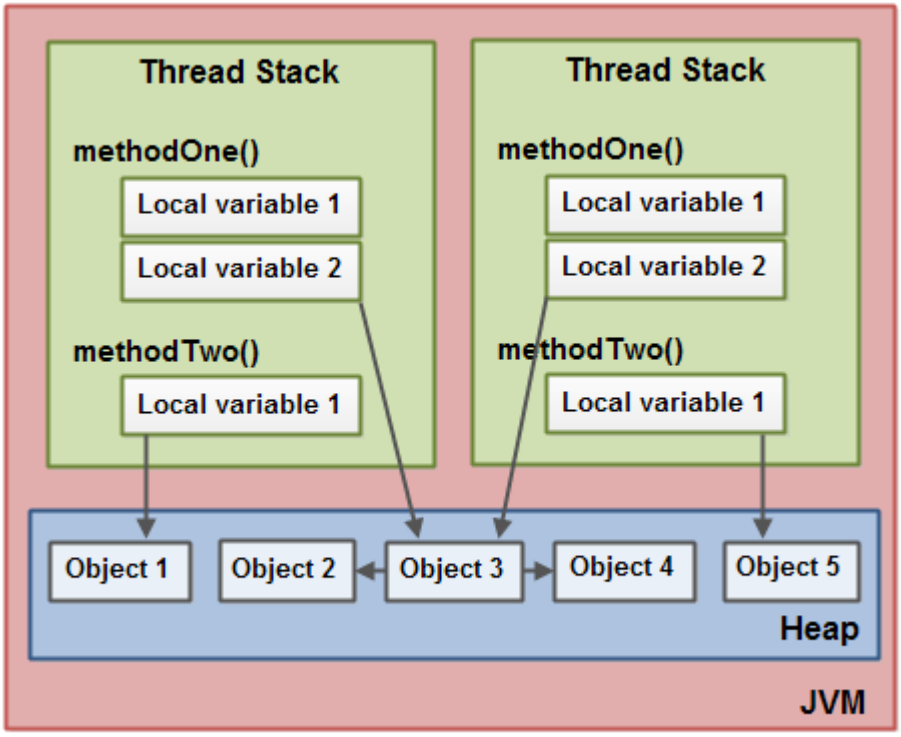
对象可以包含方法，这些方法可以包含局部变量。这些局部变量也存储在线程栈中，即使方法所属的对象存储在堆中。

对象的成员变量与对象本身一起存储在堆中。不管成员变量是基本类型还是对象的引用，都是如此。

静态类变量也与类定义一起存储在堆中。

堆上的对象可以被所有引用该对象的线程访问。当线程有权访问某个对象时，它也可以访问该对象的成员变量。如果两个线程同时调用同一个对象上的方法，它们都可以访问该对象的成员变量，但每个线程都有自己的本地变量副本。

下面的示意图说明了以上几点：



两个线程有一组局部变量。其中一个局部变量（局部变量 2）指向堆上的共享对象（对象 3）。这两个线程对同一个对象分别有不同的引用。它们的引用是局部变量，因此存储在每个线程的线程栈中（在每个线程上）。不过，这两个不同的引用指向的是堆中的同一个对象。

请注意，共享对象（对象 3）将对象 2 和对象 4 的引用作为成员变量（如从对象 3 到对象 2 和对象 4 的箭头所示）。通过对象 3 中的这些成员变量引用，两个线程可以访问对象 2 和对象 4。

该图还显示了一个局部变量指向堆上两个不同的对象。在这种情况下，引用指向两个不同的对象（对象 1 和对象 5），而不是同一个对象。理论上，如果两个线程都引用了对象 1 和对

象 5，那么两个线程都可以访问这两个对象。但是在上面的图中，每个线程只有其中一个对象的引用。

那么，什么样的 Java 代码可以生成上面的内存图呢？代码如下所示：

```
public class MyRunnable implements Runnable() {

    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
```

```
        Integer localVariable1 = new Integer(99);

        //... do more with local variable.
    }
}
public class MySharedObject {

    //static variable pointing to instance of MySharedObject

    public static final MySharedObject sharedInstance =
        new MySharedObject();


    //member variables pointing to two objects on the heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member1 = 67890;
}
```

如果有两个线程正在执行 run（）方法，那么结果就如同前面的示意图所示。run（）方法调用 methodOne（），methodOne（）调用 methodTwo（）。

methodOne（）声明一个原始局部变量（类型为 int 的 localVariable1）和一个作为对象引用的局部变量（localVariable2）。

每个执行 methodOne（）的线程都将在各自的线程栈上创建自己的 localVariable1 和 localVariable2 副本。localVariable1 变量彼此是完全分离的，只存在于每个线程的线程栈中。一个线程看不到另一个线程对其 localVariable1 副本所做的更改。

每个执行 methodOne（）的线程还将创建自己的 localVariable2 副本。但是，localVariable2 的两个不同副本都指向堆上的同一个对象。代码将 localVariable2 设置为指向一个由静态变量引用的对象。静态变量只有一个副本，此副本存储在堆中。因此，localVariable2 的两个副本都指向静态变量指向的 MySharedObject 的同一个实例。MySharedObject 实例也存储在堆中。它对应于上图中的对象 3。

注意 MySharedObject 类也包含两个成员变量。成员变量本身与对象一起存储在堆中。这两个成员变量指向另外两个整数对象。这些整数对象对应于上图中的对象 2 和对象 4。

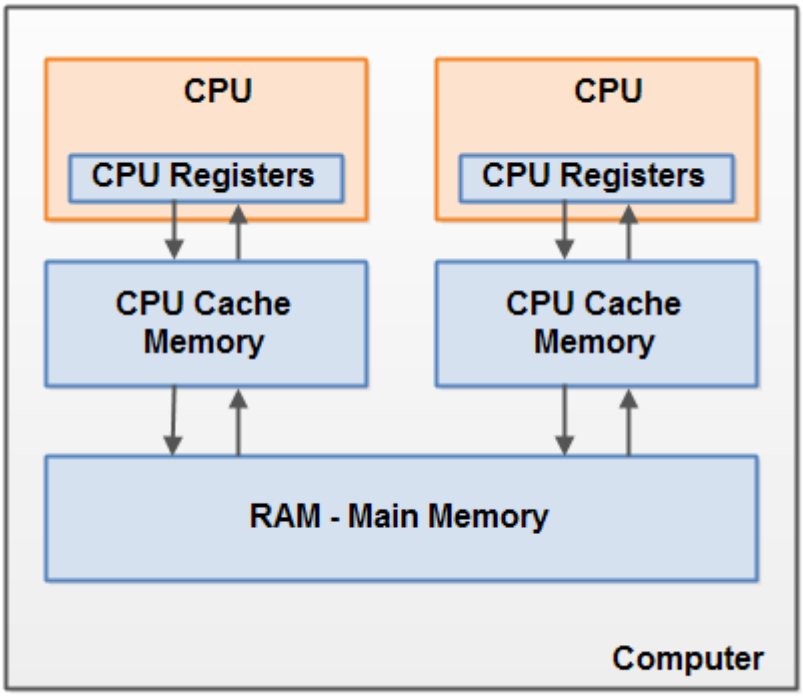
还要注意 methodTwo（）创建了名为 localVariable1 的局部变量。此局部变量是整数对象的引用。该方法将 localVariable1 引用设置为指向新的整数实例。localVariable1 引用存储在每个执行 methodTwo()的线程的一个副本中。实例化的两个整数对象存储在堆中，但是由于该方法每次执行时都会创建一个新的整数对象，因此执行该方法的两个线程将创建单独的整数实例。methodTwo（）中创建的整数对象对应于上图中的对象 1 和对象 5。

还要注意类 MySharedObject 中的两个 long 类型的成员变量，long 是一个基本类型。因为这些变量是成员变量，所以它们仍然与对象一起存储在堆中。只有局部变量存储在线程栈中。

硬件内存架构

现代的硬件内存架构与 Java 内部内存模型有些不同。要想理解 Java 内存模型是如何与硬件一起工作的，那么理解硬件内存架构也很重要。本节描述了常见的硬件内存架构，下一节将描述 Java 内存模型如何与之一起工作。

下面是现代计算机硬件架构的简化图：



现代计算机通常有两个或更多的 CPU。其中一些 CPU 可能也有多个内核。关键是，在一台拥有两个或更多 CPU 的现代计算机上，有可能同时运行多个线程。每个 CPU 都能在任何

特定的时间运行一个线程。这意味着，如果 Java 应用程序是多线程的，那么每个 CPU 可能同时地（并发地）在 Java 应用程序中运行一个线程。

每个 CPU 包含一组寄存器，这些寄存器本质上位于 CPU 内存中。CPU 在这些寄存器上执行操作比在主存中的变量上执行操作要快得多。这是因为 CPU 访问这些寄存器的速度比访问主存的速度快得多。

每个 CPU 还可以具有 CPU 高速缓冲（cache）存储器层。事实上，大多数现代 CPU 都有一定大小的高速缓存层。CPU 访问高速缓存比主存快得多，但通常不如内部寄存器速度快。因此，CPU 高速缓存的速度介于内部寄存器和主内存之间。有些 CPU 可能有多个缓存层（级别 1 和级别 2），但了解 Java 内存模型如何与内存交互并不重要。重要的是要知道 CPU 可以有某种类型的缓存层。

计算机还包含一个主存储器区（RAM）。所有 CPU 都可以访问主存。主内存区域通常比 CPU 的高速缓存大得多。

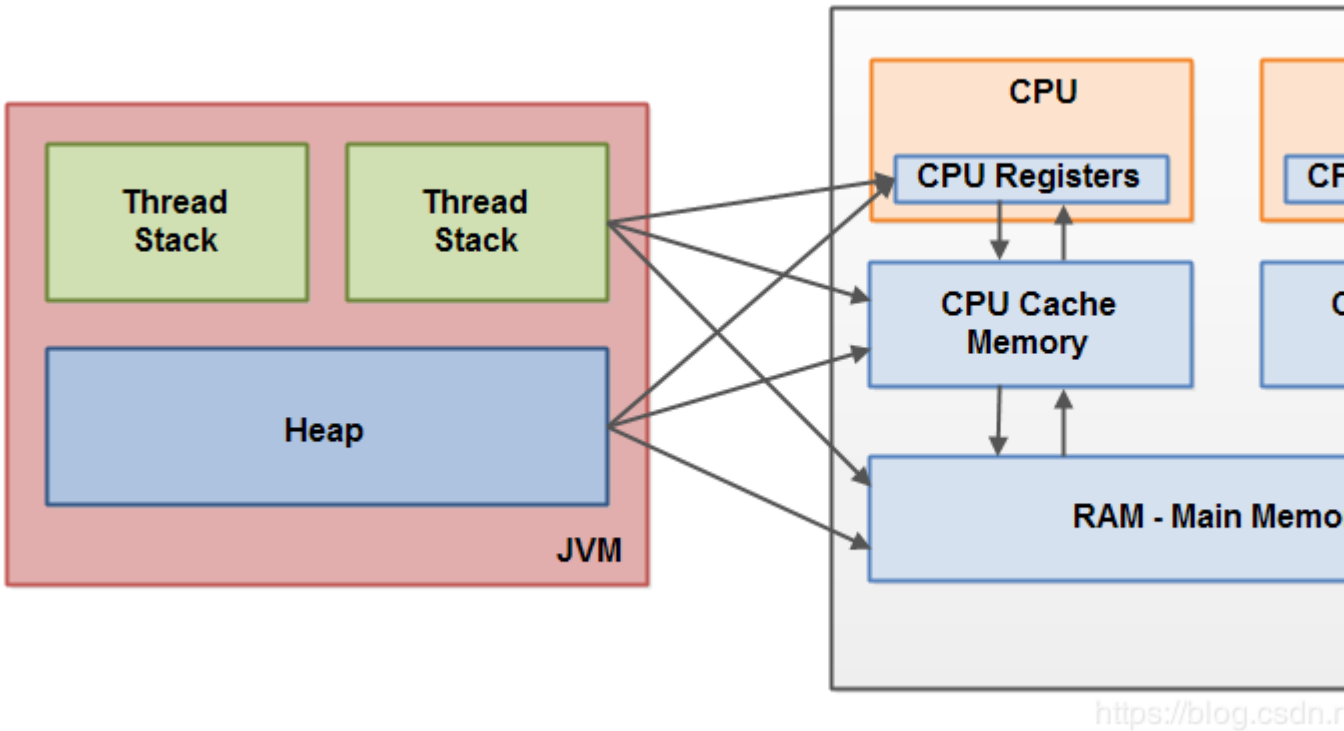
通常，当 CPU 需要访问内存时，它会将一部分内存读入 CPU 缓存。它甚至可以将一部分缓存读入其内部寄存器，然后对其执行操作。当 CPU 需要将结果写回主存时，它会将值从其内部寄存器刷新到高速缓冲存储器，并在某个时刻将值刷新回主存。

当 CPU 需要在缓存中存储其他内容时，存储在缓存中的值通常会被刷新回主内存。CPU 缓存可以一次将数据写入其部分内存，并一次刷新其部分内存。它不必每次更新时都读/写完整的缓存。通常，缓存在称为“缓存线”的较小内存块中更新。一条或多条高速缓存线可能被读入高速缓存，一条或多条高速缓存线可能被再次刷新回主内存。

跨越 Java 内存模型和硬件内存架构之间的鸿沟

如前所述，Java 内存模型和硬件内存架构是不同的。硬件内存架构不会区分线程栈和堆。

在硬件上，线程栈和堆都位于主内存中。部分线程堆栈和堆有时可能存在于 CPU 缓存和 CPU 内部寄存器中。如图所示：



当对象和变量可以存储在计算机的多个不同存储区域时，可能会出现某些问题。两个主要问题是：

线程更新（写入）到共享变量的可见性。

读取、检查和写入共享变量时的竞态条件。

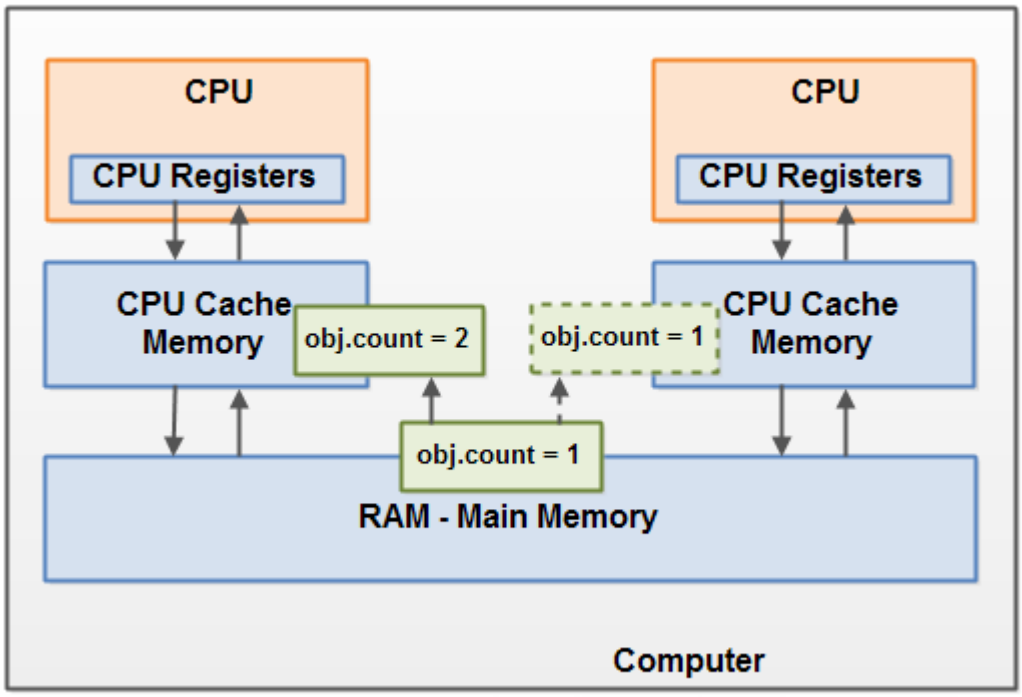
在下面的章节中将解释这两个问题。

共享对象的可见性

如果两个或多个线程共享一个对象，而没有正确使用 volatile 声明或同步，则其他线程可能看不到一个线程对共享对象所做的更新。

假设共享对象最初存储在主内存中。然后，在 CPU 1 上运行的线程将共享对象读入其 CPU 缓存。它在缓存上对共享对象进行了更改。只要 CPU 缓存没有被刷新回主内存，其他 CPU 上运行的线程就看不到共享对象的更改版本。这样，每个线程最终都可能拥有自己的共享对象副本，每个副本都位于不同的 CPU 缓存中。

下图说明了大致情况。在左侧 CPU 上运行的一个线程将共享对象复制到其 CPU 缓存中，并将其 count 变量更改为 2。此更改对在右侧 CPU 上运行的其他线程不可见，因为 count 的更新尚未刷新回主内存。



要解决这个问题，可以使用 Java 的 volatile 关键字。volatile 关键字可以确保直接从内存读取修饰的变量，并且在更新时总是写回内存。

竞态条件

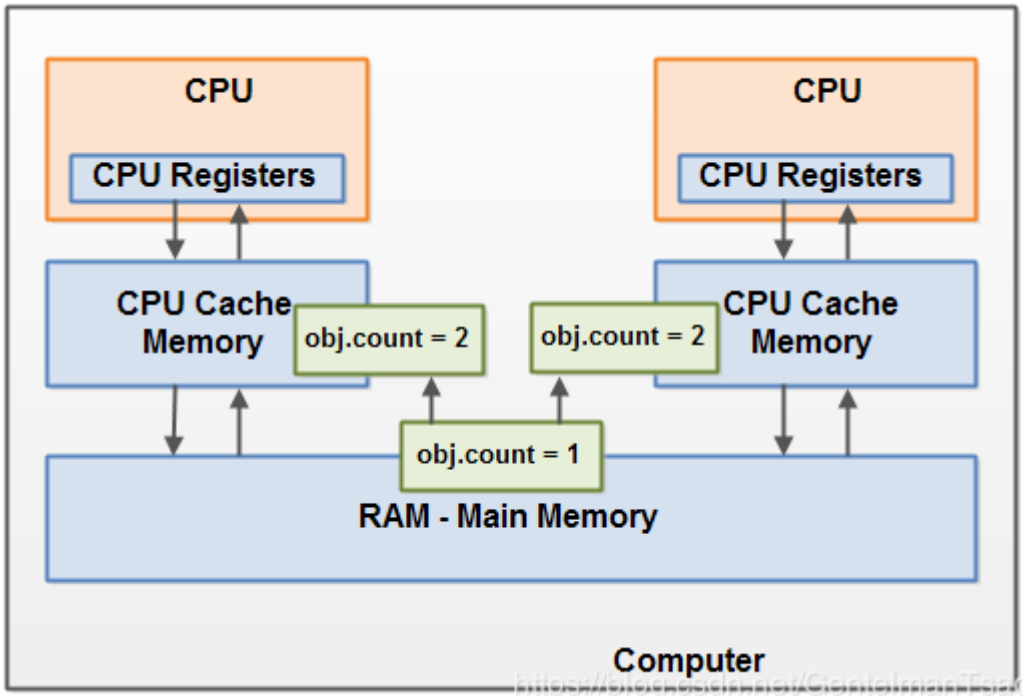
如果两个或多个线程共享一个对象，并且多个线程更新该共享对象中的变量，则可能出现竞态。

假设线程 A 将共享对象的变量 count 读入其 CPU 缓存。再假设，线程 B 也做了同样的事情，但是 count 读到了不同的 CPU 缓存。现在线程 A 给 count 加 1，线程 B 也这样做。现在 var1 增加了两次，每个 CPU 缓存一次。(**译者注：原文即为 var1，但实际应指 count**)

如果这些增量操作是按顺序执行的，那么变量 count 将增加两次，并将原始值加 2 写回内存。

然而，这两个增量操作是在没有适当同步的情况下并发执行的。不管是线程 A 还是线程 B 将更新后的 count 版本写回内存，更新后的值仅比原始值高 1，尽管增加了 2 次。

下图说明了上述出现的竞态条件问题：



要解决这个问题，可以使用 Java 同步块。同步块确保了在任何时间只有一个线程可以进入代码的临界区。同步块还确保了同步块中访问的所有变量都将从主内存中读取，当线程退出同步块时，所有更新的变量都将再次刷新回主内存，无论变量是否声明为 volatile。

（十二）：Java 同步块（synchronized 关键字，四种同步块，数据可见性，指令重排，性能开销，可重入性）

文章目录

Java 同步块将方法或代码块标记为 synchronized。Java 中的同步块一次只能在一个线程中执行（取决于你如何使用它）。因此，可以使用 Java 同步块来避免竞态条件。本 Java synchronized 教程将更详细地解释 Java synchronized 关键字是如何工作的。

Java 并发工具包

synchronized 机制是 Java 的第一个同步机制，用于访问多个线程共享的对象。不过，synchronized 机制并不怎么先进。这就是为什么 Java5 有一整套并发工具类，来帮助开发人员实现比使用 synchronized 更细粒度的并发控制。

Java synchronized 关键字

Java 中的同步块用 Synchronized 关键字标记。Java 中的同步块是在某个对象上同步的。同一时刻，在同一对象上同步的所有同步块只能有一个线程在其中执行。所有其他线程将被阻止进入同步块，直到同步块中的线程退出。

synchronized 关键字可用于标记四种不同类型的块：
实例方法
静态方法
实例方法中的代码块
静态方法中的代码块
这些块在不同的对象上同步。用哪种类型的同步块要看具体情况。下面将更详细地解释每一个同步块。

同步实例方法

下面是一个同步实例方法：

```
public class MyCounter {

   private int count = 0;

   public synchronized void add(int value){
        this.count += value;
   }
}
```

注意 add（）方法声明中 synchronized 关键字的使用。这告诉 Java 该方法是同步的。

Java 中的同步实例方法在拥有该方法的实例（对象）上同步。因此，每个实例都在不同的对象（所属实例）上同步其同步方法。

在同步实例方法中，每个实例只能执行一个线程。如果存在多个实例，则在每个实例的同步实例方法内，一次可以执行一个线程。每个实例一个线程。

对于同一对象（实例）的所有同步实例方法都是如此。因此，在下面的示例中，只有一个线程可以在两个同步方法中的任何一个方法内执行。每个实例就一个线程：

```
public class MyCounter {

   private int count = 0;

   public synchronized void add(int value){
        this.count += value;
```

```
    }

    public synchronized void subtract(int value){

        this.count -= value;

    }


}
```

同步静态方法

把静态方法标记为同步的就像实例方法使用 synchronized 关键字一样。下面是一个 Java 同步静态方法示例：

```
public static MyStaticCounter{


    private static int count = 0;


    public static synchronized void add(int value){

        count += value;

    }




}
```

同样的，这里 synchronized 关键字告诉 Java， add（）方法是同步的。

同步静态方法在它所属类的类对象上同步。由于每个类在 Java VM 中只存在一个类对象，

因此在同一个类中的静态同步方法中只能执行一个线程。

如果一个类包含多个静态同步方法，那么在这些方法中，同一时间只有一个线程可以执行。

看看这个静态同步方法示例：

```
public static MyStaticCounter{

   private static int count = 0;

   public static synchronized void add(int value){
      count += value;
   }

   public static synchronized void subtract(int value){
      count -= value;
   }
}
```

在任何给定时间，只有一个线程可以在 add（）或 subtract（）其中任何一个方法内执行。

如果线程 A 正在执行 add（），则在线程 A 退出 add（）之前，线程 B 不能执行 add（）或 subtract（）。

如果静态同步方法位于不同的类中，则可以在每个类的静态同步方法中执行一个线程。每个类一个线程，不管它调用哪个静态同步方法。

实例方法中的同步块

你不必同步整个方法，有时最好只同步方法的一部分。这可以用方法中的 Java 同步块。

下面是未同步的 Java 方法中的代码同步块：

```
public void add(int value){


    synchronized(this){
        this.count += value;
    }
}
```

本例使用 Java 同步块构造将代码块标记为 synchronized。这段代码现在将像同步方法一样执行。

注意 Java 同步块构造在括号中接受了一个对象。在示例中使用“this”，这是 add 方法的实例。同步构造在括号中获取的对象称为监视（monitor）对象。这段代码称为在监视对象上同步。同步实例方法将其所属的对象用作监视对象。

在同一个监视器对象上同步的 java 代码块内，只有一个线程可以执行。

以下两个示例都是在调用它们的实例上同步的。因此，它们在同步效果上是等价的：

```
public class MyClass {


    public synchronized void log1(String msg1, String msg2){
        log.writeln(msg1);
        log.writeln(msg2);
    }
```

```
    public void log2(String msg1, String msg2){
        synchronized(this){
            log.writeln(msg1);
            log.writeln(msg2);
        }
    }
}
```

同一时刻，只有一个线程可以在这两个方法其中一个内执行。

如果第二个同步块被同步到别的对象而不是 MyClass.class 上，则可以在每个方法内同时执行一个线程。

静态方法中的同步块

同步块也可以在静态方法内部使用。下面是上一节中与静态方法相同的两个示例。这些方法在方法所属类的类对象上同步：

```
public class MyClass {

    public static synchronized void log1(String msg1, String msg2){
        log.writeln(msg1);
        log.writeln(msg2);
    }
```

```
    public static void log2(String msg1, String msg2){
        synchronized(MyClass.class){
            log.writeln(msg1);
            log.writeln(msg2);
        }
    }
}
```

因此，在本例中，只有一个线程可以在两个同步块中的其中一个内执行。

如果第二个同步块被同步到与此不同的对象上，则在同一时间一个线程可以在两个方法中执行。

同步块的 Lambda 表达式

你甚至可以在 Java Lambda 表达式和匿名类中使用同步块。

下面是一个 Java lambda 表达式的示例，其中包含一个同步块。请注意，同步块是在包含 lambda 表达式的类的类对象上同步的。它也可以在另一个对象上同步，如果这更有意义的话（如果给一个特定的用例），但是在这个例子中使用类对象是可以的。

```
import java.util.function.Consumer;

public class SynchronizedExample {
```

```
public static void main(String[] args) {

    Consumer<String> func = (String param) -> {

        synchronized(SynchronizedExample.class) {

            System.out.println(
                    Thread.currentThread().getName() +
                            " step 1: " + param);

            try {
                Thread.sleep( (long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(
                    Thread.currentThread().getName() +
                            " step 2: " + param);
        }
```

```
        };


        Thread thread1 = new Thread(() -> {
            func.accept("Parameter");
        }, "Thread 1");

        Thread thread2 = new Thread(() -> {
            func.accept("Parameter");
        }, "Thread 2");

        thread1.start();
        thread2.start();
    }
}
```

Java 同步示例

下面是一个例子，它启动两个线程，让它们在同一个 Counter 实例上调用 add 方法。在同一个实例上，一次只能有一个线程调用 add 方法，因为该方法在它所属的实例上是同步的。

```
public class Example {

    public static void main(String[] args){
```

```
        Counter counter = new Counter();
        Thread    threadA = new CounterThread(counter);
        Thread    threadB = new CounterThread(counter);

        threadA.start();
        threadB.start();
    }
}
```

下面是上例中使用的两个类：Counter 和 CounterThread。

```
public class Counter{

    long count = 0;

    public synchronized void add(long value){
       this.count += value;
    }
}
public class CounterThread extends Thread{

    protected Counter counter = null;

    public CounterThread(Counter counter){
```

```
        this.counter = counter;
    }

    public void run() {
    for(int i=0; i<10; i++){
            counter.add(i);
        }
    }
}
```

这个例子创建了两个线程，并将同一个计数器实例传递给它们的构造函数。Counter.add()方法在实例上是同步的，因为 add 方法是实例方法，并且标记为 synchronized。因此，同一时间只能有一个线程调用 add（）方法。另一个线程将等待，直到第一个线程离开 add（）方法，然后才能执行该方法。

如果这两个线程引用了两个单独的计数器实例，那么同时调用 add（）方法就不会有问题。这些调用在不同对象上，因此被调用的方法也会在不同的对象（拥有该方法的对象）上同步。因此，调用不会阻塞。像下面这样：

```
public class Example {

  public static void main(String[] args){
    Counter counterA = new Counter();
    Counter counterB = new Counter();
    Thread  threadA = new CounterThread(counterA);
```

```
        Thread    threadB = new CounterThread(counterB);


        threadA.start();
        threadB.start();
    }
}
```

注意两个线程 threadA 和 threadB 不再引用同一个 counter 实例。counterA 和 counterB 的 add 方法在其所属的两个实例上同步。因此，对 counterA 调用 add（）不会阻止对 counterB 调用 add（）。

同步和数据可见性

如果不使用 synchronized 关键字（或 Java volatile 关键字），则无法保证当一个线程更改与其他线程共享的变量值（例如，通过所有线程都有权访问的对象）时，其他线程可以看到更改后的值。既无法保证一个线程保存在 CPU 寄存器中的变量何时"提交"到主内存；也无法保证其他线程何时从主内存"刷新"保存在 CPU 寄存器中的变量。

有 synchronized 关键字就不一样了。当线程进入同步块时，它将刷新线程所有可见变量的值。当线程退出同步块时，线程可见变量的所有更改都将提交到主内存。这与 volatile 关键字的工作原理类似。

同步和指令重新排序

Java 编译器和 Java 虚拟机可以对代码中的指令重新排序，以使它们执行得更快，通常是使重新排序的指令能够由 CPU 并行执行。

指令重新排序可能会导致多个线程同时执行的代码出现问题。例如，如果发生在同步块内部的变量的写入被重新排序为发生在同步块外部。

为了解决这个问题，Java synchronized 关键字对同步块之前、内部和之后的指令重新排序设置了一些限制。这类似于 volatile 关键字设置的限制。

最终目标是，你可以确保代码工作正常——没有发生任何指令重新排序，导致代码的最终行为不同于代码的预期行为。

在哪个对象上同步

正如本 Java 同步教程中多次提到的，同步块必须在某个对象上同步。实际上，你可以选择任何对象来同步，但建议不要用字符串对象或任何基本类型包装器对象进行同步，因为编译器可能会对这些对象进行优化，从而导致你以为用的是不同实例，而实际上用的是同一个实例。看看这个例子：

```
synchronized("Hey") {
    //do something in here.
}
```

1

2

如果有多个 synchronized 块在文本字符串值“Hey”上同步，那么编译器实际上可能在幕后使用相同的字符串对象。结果是，这两个同步块随后在同一对象上同步。这可能不是你想要的行为。

对于使用基本类型包装器对象也是如此。看看这个例子：

```
synchronized(Integer.valueOf(1)) {
```

```
    //do something in here.
}
1
2
```

如果你调用 Integer.valueOf(1)） 多次，它可能会根据相同的输入参数值返回相同的包装器对象实例。也就是说，如果在同一个基本包装器对象上同步多个块(例如.多次使用 Integer.valueOf(1) 作为监视对象)，则有可能使这些同步块都在同一对象上同步。这也可能不是你想要的行为。

为了安全起见，请在 this 或 new Object() 上进行同步。Java 编译器、Java 虚拟机或 Java 库不会在内部缓存或重复使用它们。

同步块的局限和替代方案

Java 中的同步块有几个局限。例如，Java 中的同步块一次只允许一个线程进入。但是，如果两个线程只想读取一个共享值，而不更新它呢？这可能是安全的。作为同步块的替代，你可以使用读/写锁来保护代码，这是比同步块更高级的锁定语义。Java 实际上附带了一个内置的 ReadWriteLock 类，你可以使用它。

如果你想让 N 个线程进入一个同步块，而不仅仅是一个呢？你可以使用信号量来实现这种行为。Java 实际上附带了一个可以使用的内置 Java 信号量类。

同步块不保证线程按照等待进入的顺序访问同步块。如果您需要保证线程按照它们请求访问同步块的确切顺序访问该块，该怎么办？你需要自己实现公平性。

如果只有一个线程写共享变量，而其他线程只读取该变量，会怎么样？你只需使用 volatile 变量就可以了，而不需要进行任何同步。

同步块性能开销

在 Java 中，进入和退出同步块相关的性能开销很小。随着 java 的发展，这种性能开销已经下降，但仍要付出一点小代价。

如果在一个紧密的循环中多次进入和退出同步块，那么进入和退出同步块的性能开销就是最需要担心的。

另外，尽量不要让同步块过大。换句话说，只同步真正需要同步的操作，以避免阻止其他线程执行不需要同步的操作。同步块中只包含绝对必要的指令。这会增加代码的并行性。

同步块可重入性

一旦一个线程进入一个同步块，线程就被称为"保持锁定"在该块所同步的监视对象上。如果线程调用了另一个方法，该方法调用了第一个内部有同步块的方法，则持有锁的线程可以重新进入同步块。之所以不会阻塞是因为线程（本身）持有锁。只有当一个不同的线程持有锁（才会阻塞）。看看这个例子：

```
public class MyClass {


   List<String> elements = new ArrayList<String>();


   public void count() {
      if(elements.size() == 0) {
            return 0;
      }
      synchronized(this) {
```

```
            elements.remove();
            return 1 + count();
        }
    }

}
```

暂且不管上面这种计算列表元素的方法，该方法毫无意义。只需关注 count（）方法内的 synchronized 块内如何递归调用 count（）方法。因此，调用 count（）的线程可能最终多次进入同一个同步块。这是允许的，也是可能的。

不过，请记住，如果你不仔细的设计代码，线程进入多个同步块的设计可能会导致嵌套管程锁死（nested monitor lockout）。

集群设置中的同步块

请记住，同步块只阻止同一个 Java 虚拟机中的线程进入该代码块。如果同一个 Java 应用程序在集群中的多个 Java 虚拟机上运行，那么同一时刻，每个 Java 虚拟机可能有一个线程进入同步块。

如果需要在集群中的所有 Java 虚拟机之间进行同步，则需要使用其他同步机制，而不仅仅是同步块。

（十三）：Java volatile 关键字（变量可见性，可见性规则，指令重排序，Happens-Before 规则）

文章目录

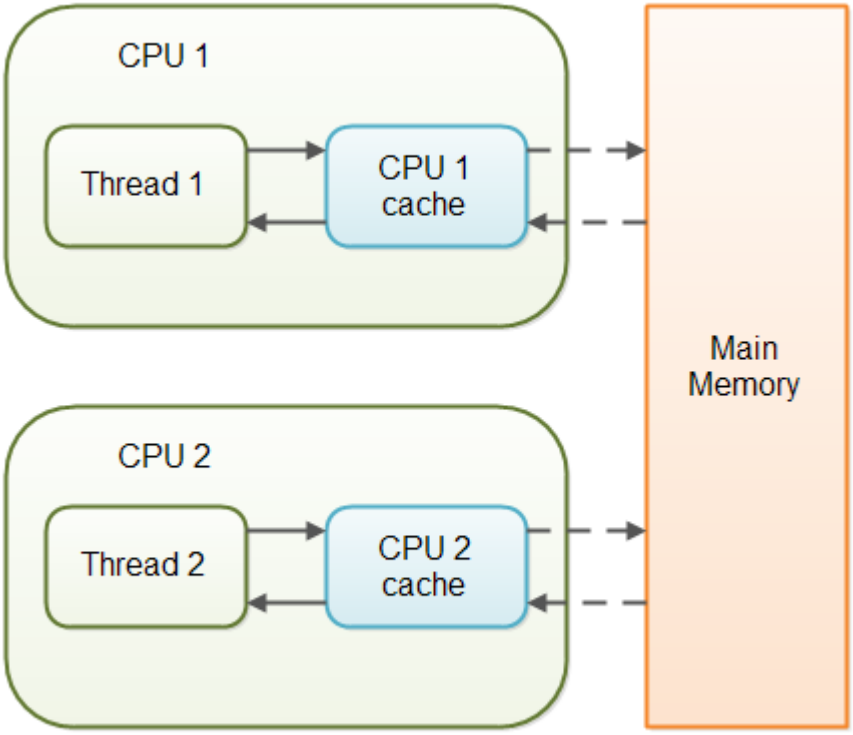
Java volatile 关键字用于将 Java 变量标记为“存储在主内存中”。更确切地说，这意味着：volatile 变量每次都将从计算机的主存储器读取，而不是 CPU 缓存中；volatile 变量每次都将写入主存储器，而不仅仅是 CPU 缓存。

实际上，从 Java5 开始，volatile 关键字保证的不仅仅是 volatile 变量从主内存读取和写入。我将在下面的章节中解释这一点。

变量可见性问题

Java volatile 关键字保证了跨线程变量更改的可见性。这听起来可能有点抽象，所以让我详细说明一下。

在多线程应用程序中，线程对非易失性变量进行操作，出于性能原因，每个线程都可以在处理变量时将变量从主内存复制到 CPU 缓存中。如果计算机包含多个 CPU，则每个线程可能在不同的 CPU 上运行。这意味着，每个线程可以将变量复制到不同 CPU 的 CPU 缓存中。下面举例说明：



对于非易失性变量，无法保证 Java 虚拟机（JVM）何时将数据从主内存读取到 CPU 缓存，何时将数据从 CPU 缓存写入主内存。这可能会导致一些问题，我将在下面的章节中解释。

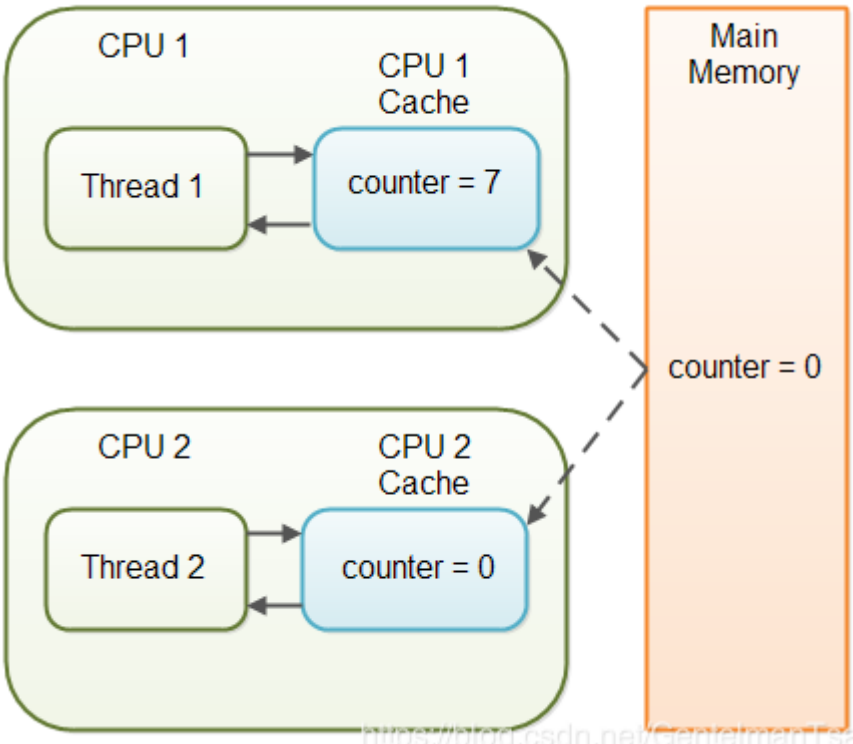
假设两个或多个线程访问一个共享对象，该对象包含了一个 counter 变量声明如下：

```
public class SharedObject {
```

```
    public int counter = 0;


}
```

想象一下，只有线程 1 递增 counter 变量，但线程 1 和线程 2 都可能不时地读取 counter 变量。

如果计数器变量未声明为 volatile，则无法保证 counter 变量的值何时从 CPU 缓存写入主内存。这意味着，CPU 缓存中的 counter 变量值可能与主内存中的不同。这种情况如下所示：



线程没有看到一个变量的最新值，因为它还没有被另一个线程写回主内存，这个问题被称为“可见性”问题。一个线程的更新对其他线程不可见。

Java volatile 可见性规则

Java volatile 关键字旨在解决变量可见性问题。通过声明 counter 变量 volatile，所有对 counter 变量的写入都将立即写回主内存。此外，counter 变量的所有读取都将直接从主内存中读取。

volatile 声明的 counter 变量像下面这样：

```
public class SharedObject {

    public volatile int counter = 0;

}
```

因此，声明变量 volatile 可以保证该变量的写入对其他线程的可见性。

在上面给出的场景中，一个线程（T1）修改 counter，另一个线程（T2）读取 counter（但从不修改），声明 counter 变量 volatile 足以保证 counter 变量的写入对 T2 可见。

但是，如果 T1 和 T2 都在递增 counter 变量，那么声明 counter 变量为 volatile 是不够的。接下来再讨论。

volatile 可见性完整规则

实际上，Java volatile 的可见性规则超出了 volatile 变量本身。可见性规则如下：

- 如果线程 A 写入 volatile 变量，而线程 B 随后读取相同的 volatile 变量，那么线程 A 在写入 volatile 变量之前可见的所有变量，在线程 B 读取 volatile 变量之后也将可见。
- 如果线程 A 读取 volatile 变量，那么线程 A 在读取 volatile 变量时可见的所有变量也将从主内存中重新读取。

  让我用一个代码示例来说明这一点：

  ```
  public class MyClass {
      private int years;
      private int months
      private volatile int days;


      public void update(int years, int months, int days){
          this.years  = years;
          this.months = months;
          this.days   = days;
      }
  }
  ```

-

  udpate（）方法写入三个变量，其中只有 days 是 volatile。

volatile 可见性完整规则意味着，当 days 写入一个值时，线程可见的所有变量也将写入主内存。这意味着，当 days 写入一个值时，years 和 months 的值也被写入主内存。

你可以这样做来读取 years，months，days 的值:

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;

    public int totalDays() {
        int total = this.days;
        total += months * 30;
        total += years * 365;
        return total;
    }

    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

注意 totalDays（）方法首先将 days 的值读入 total 变量。当读取 days 的值时，months 和 years 的值也会读入主存储器。因此，按上述读取顺序，可以确保读到 days、months 和 years 的最新值。

指令重排序带来的难题

出于性能原因，只要指令的语义保持不变，Java 虚拟机和 CPU 就可以对程序中的指令重新排序。例如，看看以下指令：

```
int a = 1;
int b = 2;

a++;
b++;
```

这些指令可以重新排序为以下顺序，而不会丧失程序的语义：

```
int a = 1;
a++;

int b = 2;
b++;
```

然而，当其中一个变量是 volatile 变量时，指令重新排序是一个难题。让我们看看本篇前面示例中的 MyClass 类：

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;


    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

update（）方法将值写入 days 后，新写入的 years 和 months 值也将写入主内存。但是，如果 Java 虚拟机重新排列了指令呢？比如像这样：

```
public void update(int years, int months, int days){
    this.days   = days;
    this.months = months;
    this.years  = years;
}
```

当 days 变量被修改时，months 和 years 的值仍会写入主内存，但这次是在新值写入 months 和 years 之前发生的。因此，其他线程无法正确地看到新值。重新排序的指令的语义发生了变化。

我们将在下一节中看到，Java 有一个方案可以解决这个问题。

Java volatile 的 Happens-Before 规则

为了解决指令重新排序的难题，除了可见性规则之外，Java volatile 关键字还提供“happens before”规则。“happens before”规则确保了：

- 如果读/写最初发生在对 volatile 变量的写入之前，则不能将对其他变量的读/写重新排序为在对 volatile 变量的写入之后发生。
- 在写入 volatile 变量之前的读/写操作保证在写入 volatile 变量之前发生。请注意，仍然有可能出现这种情况，例如，volatile 的写操作之后的其他变量的读/写，重排序为在 volatile 的写操作之前。反之则不行。允许从后到前，但不允许从前到后。
- 如果对其他变量的读取/写入最初发生在读取 volatile 变量之后，则不能将该读取和写入重新排序为在读取 volatile 变量之前发生。注意，在 volatile 变量的读取之前发生的其他变量的读取可能会被重新排序为在 volatile 变量的读取之后发生。反之则不行。允许从前到后，但不允许从后到前。

（

译者注：

作者对 H-B 规则描述的虽然正确，但不够清晰。H-B 规则是为了解决指令重排序与可见性规则的冲突。所以，将 H-B 规则和可见性规则对照起来，就很容易理解了。

首先，可见性规则解决了缓存一致性问题。解决方法可理解为“**两个刷新**”：

写 volatile 时将所有可见变量从缓存刷新到内存，简记为“**写刷新**”；

读 volatile 时将所有可见变量从内存刷新到缓存，简记为“**读刷新**”。

通过“两个刷新”，保证了在读/写 volatile 时，变量在内存和缓存中是一致。

但是，由于指令重排序的存在，刷新的动作会导致语义变化。例如对于写刷新，代码预期的是变量修改后刷新到内存，结果由于指令重排序变成了刷新到内存后再修改变量。错的很离谱。

于是，H-B 规则对指令重排作了限制，本质上可以理解为，指令重排不能影响刷新的结果。

建议不必记具体规则，但应理解为什么要这么做。

)

上述的 happens-before 规则确保了 volatile 关键字的可见性规则能够生效。

volatile 不是万金油

即使 volatile 关键字保证所有 volatile 变量的读取都直接从内存中读取，并且所有 volatile 变量的写入都直接写入内存，但在某些情况下，仅声明变量为 volatile 仍然是不够的。

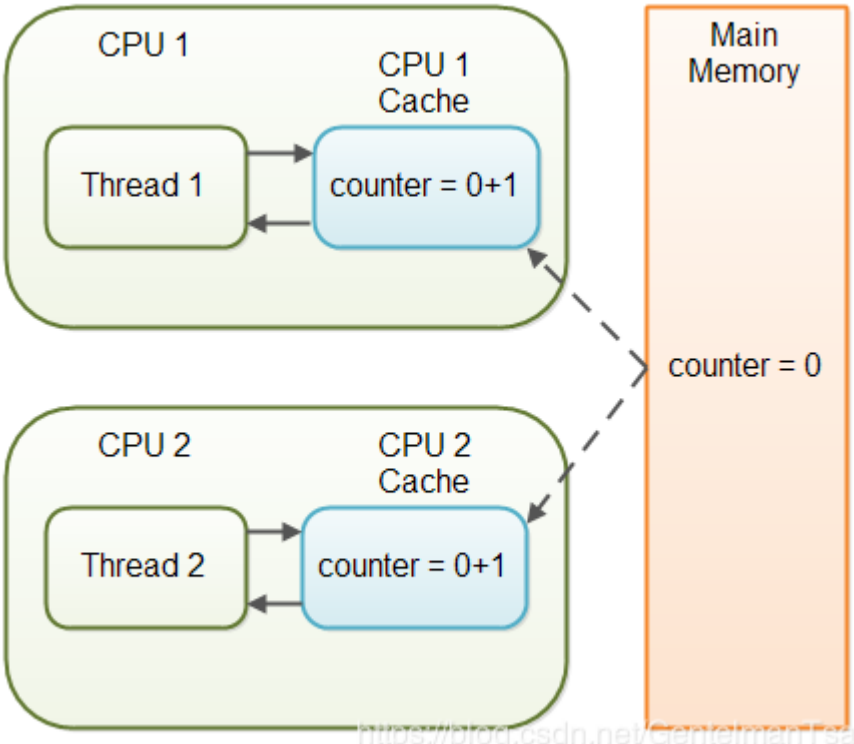
在前面阐述的只有线程 1 写入共享 counter 变量的情况下，声明 counter 变量为 volatile 足以确保线程 2 始终看到最新的写入值。

事实上，如果写入变量的新值不依赖于它的前值，甚至可以有多个线程写入共享的 volatile 变量，并且主内存中存储的值仍然是正确的。换句话说，如果一个线程向共享的 volatile 变量写入一个值，它不需要首先读取它的值来计算它的下一个值。

只要线程需要首先读取 volatile 变量的值，并基于该值生成新值赋给共享的 volatile 变量，volatile 变量就不再足以保证正确的可见性。读取 volatile 变量和写入新值之间的短暂时间间隔，造成了一个竞态条件，多个线程可能读取 volatile 变量的同一个值，为该变量生成一个新值，并且在将该值写入主内存时覆盖彼此的值。

多个线程递增同一个计数器的情况正是这样一种情况，即 volatile 变量不够用了。以下各节将更详细地解释这个案例。

假设线程 1 将一个值为 0 的共享 counter 变量读入其 CPU 缓存，将其递增为 1，但没有将更改后的值写回主内存。然后，线程 2 可以将相同的 counter 变量从主内存（变量值仍然为 0）读取到自己的 CPU 缓存中。然后线程 2 也可以将 counter 增加到 1，并且也没有将其写回主内存。这种情况如下图所示：



线程 1 和线程 2 现在几乎不同步。共享 counter 变量的实际值应该是 2，但每个线程的 CPU

缓存中该变量的值是 1，在主内存中该值仍然为 0。真是一团糟！即使线程最终将共享 counter 变量的值写回主内存，该值也会出错。

volatile 在什么时候有用

如前所述，如果两个线程都在读写一个共享变量，那么使用 volatile 关键字是不够的。在这种情况下，需要使用 synchronized 来保证变量的读写是原子的。读取或写入 volatile 变量不会阻塞线程的读取或写入。为此，必须在临界区周围使用 synchronized 关键字。

作为同步块的替代，还可以使用 java.util.concurrent 包中的原子数据类型。例如，AtomicLong 或 AtomicReference 或其他类型。

如果只有一个线程读取和写入 volatile 变量的值，而其他线程只读取该变量，那么可以保证读取线程看到 volatile 变量写入的最新值。如果不标记变量为 volatile，就无法保证这一点。

volatile 关键字可以保证在 32 位和 64 个变量上有效。

volatile 的性能斟酌

读写 volatile 变量会导致变量从内存读写。从内存读写比访问 CPU 缓存的开销更大。访问 volatile 变量还会阻止指令重新排序，这是一种正常的性能增强技术。因此，只有在真正需要保证变量可见性时，才应该使用 volatile 变量。

（十四）：Java ThreadLocal（泛型类型，supplier 接口，延后设置，InheritableThreadLocal）

文章目录

Java ThreadLocal 类可以创建只能由同一个线程读写的变量。因此，即使两个线程正在执行相同的代码，并且代码引用了相同的 ThreadLocal 变量，两个线程也无法看到彼此的 ThreadLocal 变量。因此，Java ThreadLocal 类提供了一种使代码线程安全的简单方法，

而没有这个方法则线程不是安全的。

创建 ThreadLocal

创建一个 ThreadLocal 实例就像创建其他 Java 对象一样，通过 new 操作符。下面的示例演示了如何创建 ThreadLocal 变量：

```
private ThreadLocal threadLocal = new ThreadLocal();
```

这段代码每个线程只需要执行一次。多个线程现在可以在这个 ThreadLocal 中获取和设置值，并且每个线程只能看到它自己设置的值。

设置 ThreadLocal 值

创建 ThreadLocal 后，可以使用其 set（）方法设置要存储在其中的值。

```
threadLocal.set("A thread local value");
```

获取 ThreadLocal 值

可以使用其 get（）方法读取存储在 ThreadLocal 中的值。下面是获取存储在 Java ThreadLocal 中的值的示例：

```
String threadLocalValue = (String) threadLocal.get();
```

删除 ThreadLocal 值

可以删除 ThreadLocal 变量中的值。通过调用 ThreadLocal remove（）方法删除值。下面是删除 Java ThreadLocal 上的值的示例：

threadLocal.remove();

泛型类型 ThreadLocal

可以使用泛型类型创建 ThreadLocal。使用泛型类型时，只能将泛型类型的对象设置为 ThreadLocal 上的值。此外，不必对 get（）返回的值进行类型转换。下面是一个泛型 ThreadLocal 示例：

```
private ThreadLocal<String> myThreadLocal = new
ThreadLocal<String>();
```

现在只能在 ThreadLocal 实例中存储字符串。此外，不需要对从 ThreadLocal 获得的值进行类型转换：

```
myThreadLocal.set("Hello ThreadLocal");


String threadLocalValue = myThreadLocal.get();
```

- 1
- 2

初始化 ThreadLocal 值

在调用 set（）赋值之前，可以为 Java ThreadLocal 设置一个初始值，该值将在第一次调用 get（）时使用。指定 ThreadLocal 的初始值有两种方式：

- 创建一个 ThreadLocal 子类来重写 initialValue（）方法。
- 使用 Supplier 接口的实现来创建 ThreadLocal。

一下章节将演示这两种方式。

重写 initialValue（）

为 Java ThreadLocal 变量指定初始值的第一种方法是创建 ThreadLocal 的子类，该子类重写其 initialValue（）方法。创建 ThreadLocal 的子类的最简单方法就是创建一个匿名子类，就在这里创建 ThreadLocal 变量。下面是创建 ThreadLocal 的匿名子类的示例，该子类重写 initialValue（）方法：

```
private ThreadLocal myThreadLocal = new ThreadLocal<String>() {
    @Override protected String initialValue() {
        return String.valueOf(System.currentTimeMillis());
    }
};
```

注意，不同的线程仍然会看到不同的初始值。每个线程都将创建自己的初始值。只有从 initialValue（）方法返回完全相同的对象，所有线程才会看到相同的对象。但是，使用 ThreadLocal 的首要目的是为了避免不同的线程看到相同的实例。

实现一个 Supplier 接口

为 Java ThreadLocal 变量指定初始值的第二种方法是使用其静态 factory 方法 withInitial(Supplier)，传递 Supplier 接口的实现作为参数。Supplier 接口的实现提供了 ThreadLocal 的初始值。下面是一个使用其 withInitial（）静态工厂方法创建 ThreadLocal 的示例，传递了一个简单的 Supplier 实现作为参数：

```
ThreadLocal<String> threadLocal = ThreadLocal.withInitial(new
Supplier<String>() {
    @Override
    public String get() {
        return String.valueOf(System.currentTimeMillis());
    }
});
```

由于 Supplier 是一个功能接口，因此它可以使用 Java Lambda 表达式实现。下面是将 Supplier 实现作为 lambda 表达式提供给 withInitial（）：

```
ThreadLocal threadLocal = ThreadLocal.withInitial(
        () -> { return String.valueOf(System.currentTimeMillis()); } );
```

- 1

如你所见，这比前面的示例略短。但是，如果使用 lambda 表达式最密集的语法，语句还可以更短：

```
ThreadLocal threadLocal3 = ThreadLocal.withInitial(
        () -> String.valueOf(System.currentTimeMillis()) );
```

- 1

ThreadLocal 值的延后设置

在某些情况下，不能使用标准方法设置初始值。例如，你可能需要一些在创建 ThreadLocal 变量时不可用的配置信息。在这种情况下，可以延后设置初始值。下面是如何在 Java ThreadLocal 上延后设置初始值的示例：

```
public class MyDateFormatter {

    private ThreadLocal<SimpleDateFormat>
simpleDateFormatThreadLocal = new ThreadLocal<>();

    public String format(Date date) {
        SimpleDateFormat simpleDateFormat =
getThreadLocalSimpleDateFormat();
        return simpleDateFormat.format(date);
    }


    private SimpleDateFormat getThreadLocalSimpleDateFormat() {
        SimpleDateFormat simpleDateFormat =
simpleDateFormatThreadLocal.get();
        if(simpleDateFormat == null) {
            simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
            simpleDateFormatThreadLocal.set(simpleDateFormat);
```

```
            }
            return simpleDateFormat;
        }
}
```

注意 format（）方法调用 getThreadLocalSimpleDateFormat（）方法来获取 Java SimpleDatFormat 实例。如果尚未在 ThreadLocal 中设置 simpledataformat 实例，则会创建一个新的 simpledataformat 设给 ThreadLocal 变量。一旦一个线程在 ThreadLocal 变量中设置了自己的 SimpleDateFormat，该线程将使用同一个 SimpleDateFormat 对象。但只限于该线程。每个线程都创建自己的 SimpleDateFormat 实例，因为它们看不到在 ThreadLocal 变量上设置的其他实例。

SimpleDateFormat 类不是线程安全的，因此多个线程不能同时使用它。为了解决这个问题，上面的 MyDateFormatter 类为每个线程创建一个 SimpleDateFormat，因此调用 format（）方法的每个线程都将使用自己的 SimpleDateFormat 实例。

ThreadLocal 与线程池或 ExecutorService 一起使用

如果计划在任务内部使用 Java ThreadLocal，并将任务传递给 Java 线程池或 Java ExecutorService，请记住，你无法确定哪个线程执行任务。但是，如果只需要确保每个线程使用其自己的某个对象实例，则不会有问题。所以可以将 Java ThreadLocal 与线程池或 ExecutorService 配合使用。

完整的 ThreadLocal 示例

下面是一个完全可运行的 Java ThreadLocal 示例：

```
public class ThreadLocalExample {

    public static void main(String[] args) {
        MyRunnable sharedRunnableInstance = new MyRunnable();

        Thread thread1 = new Thread(sharedRunnableInstance);
        Thread thread2 = new Thread(sharedRunnableInstance);

        thread1.start();
        thread2.start();

        thread1.join(); //wait for thread 1 to terminate
        thread2.join(); //wait for thread 2 to terminate
    }

}
public class MyRunnable implements Runnable {

    private ThreadLocal<Integer> threadLocal = new
ThreadLocal<Integer>();
```

```
        @Override
        public void run() {
            threadLocal.set( (int) (Math.random() * 100D) );

            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
            }

            System.out.println(threadLocal.get());
        }
}
```

这个例子创建了一个 MyRunnable 实例，并传递给两个不同的线程。两个线程都执行 run（）方法，因此在 ThreadLocal 实例上设置不同的值。如果 set（）调用是同步的，并且不是 ThreadLocal 对象，则第二个线程将重写第一个线程设置的值。

但是，由于它是一个 ThreadLocal 对象，因此两个线程无法看到彼此的值。因此，它们设置和读取的是不同的值。

InheritableThreadLocal

InheritableThreadLocal 类是 ThreadLocal 的子类。InheritableThreadLocal 将对值的访问权授予线程和该线程创建的所有子线程，而不是每个线程在 ThreadLocal 中都有自己的值。

（十五）：线程信号传递（忙等待，wait、notify、notifyall，信号丢失，虚假唤醒）

文章目录

线程信号传递的目的是使线程能够相互发送信号。另外，线程信号传递使线程能够等待来自其他线程的信号。例如，线程 B 可能会等待线程 a 发出的的信号，指示数据已准备好待处理。

通过共享对象发送信号

线程相互发送信号的一种简单方法是在某个共享对象变量中设置信号值。线程 A 可以在同步块内将布尔成员变量 hasDataToProcess 设置为 true，线程 B 也可以在同步块内读取 hasDataToProcess 成员变量。下面是一个简单的对象示例，它可以保存这样的信号，并提供设置和检查它的方法：

```
public class MySignal{

  protected boolean hasDataToProcess = false;

  public synchronized boolean hasDataToProcess(){
    return this.hasDataToProcess;
  }

  public synchronized void setHasDataToProcess(boolean hasData){
    this.hasDataToProcess = hasData;
  }

}
```

线程 A 和 B 必须引用共享 MySignal 实例，才能使信令工作。如果线程 A 和线程 B 引用不同的 MySignal 实例，它们将不会互相检测信号。要处理的数据可以位于共享缓冲区中，与 MySignal 实例分离。

忙等待

处理数据的线程 B 正在等待数据可供处理。换言之，它正在等待来自线程 a 的信号，该信号会令 hasDataToProcess（）返回 true。这是线程 B 在等待此信号时运行的循环：

```
protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){
   //do nothing... busy waiting
}
```

注意 while 循环一直执行，直到 hasDataToProcess（）返回 true。这叫忙等待。线程等待时处于忙碌状态。

wait(), notify() 和 notifyAll()

在运行等待线程的计算机中，忙等待没有很有效地利用 CPU。除非平均等待时间很短，否则更有效的做法是，等待的线程能够以某种方式休眠或变为非活动状态，直到接收到正在等待的信号。

Java 有一个内置的等待机制，使线程在等待信号时变为非活动状态。java.lang.Object 类定义了三个方法，wait（）、notify（）和 notifyAll（），以便实现。

线程对某个对象调用 wait（）使自己变为非活动线程，直到另一个线程对该对象调用 notify（）为止。要调用 wait（）或 notify（），调用线程必须首先获取该对象的锁。换句话说，

调用线程必须从同步块内部调用 wait（）或 notify（）。下面是 MySignal 的一个修改版本，名为 MyWaitNotify，它使用 wait（）和 notify（）。

```
public class MonitorObject{
}

public class MyWaitNotify{

  MonitorObject myMonitorObject = new MonitorObject();

  public void doWait(){
    synchronized(myMonitorObject){
      try{
        myMonitorObject.wait();
      } catch(InterruptedException e){...}
    }
  }

  public void doNotify(){
    synchronized(myMonitorObject){
      myMonitorObject.notify();
    }
  }
```

}

等待线程将调用 doWait（），通知线程将调用 doNotify（）。当线程调用对象的 notify（）时，等待该对象的其中一个线程将被唤醒并允许执行。还有一个 notifyAll（）方法，它将唤醒等待该对象的所有线程。

等待和通知线程都从同步块中调用 wait（）和 notify（）。这是必须的！如果线程不持有调用该方法的对象的锁，则不能调用 wait（）、notify（）或 notifyAll（）。否则，会引发 IllegalMonitorStateException。

但是，这怎么可能呢？只要在同步块内执行，等待线程不就会保留 monitor 对象（myMonitorObject）上的锁吗？等待线程不会阻止通知线程进入 doNotify（）中的 synchronized 块吗？不会阻止。一旦一个线程调用 wait（），它就会释放在 monitor 对象上的锁。这允许其他线程也调用 wait（）或 notify（），因为这些方法必须在同步块内部调用。

线程被唤醒时，它不能直接退出 wait（），而需等调用 notify（）的线程离开其同步块。换句话说：唤醒的线程必须重新获取 monitor 对象上的锁，然后才能退出 wait（），因为 wait（）嵌套在同步块中。如果使用 notifyAll（）唤醒多个线程，则一次只能有一个唤醒的线程可以退出 wait（）方法，因为每个线程在退出 wait（）之前必须依次获取 monitor 对象上的锁。

信号丢失

如果方法 notify（）和 notifyAll（）在调用时没有线程在等待，也不会保存该方法调用。因而通知信号就丢失了。因此，如果线程在等待线程调用 wait（）之前调用 notify（），则等待线程将错过该信号。这可能是问题，也可能不是问题，但在某些情况下，这可能会导致等待线程永远等待，永远不会唤醒，因为唤醒信号丢失了。

为了避免丢失信号，信号应该存储在发信号的类中。在 MyWaitNotify 示例中，通知信号应存储在 MyWaitNotify 实例内的成员变量中。这在下面的 MyWaitNotify 的修改版本中实现了：

```
public class MyWaitNotify2{

  MonitorObject myMonitorObject = new MonitorObject();
  boolean wasSignalled = false;

  public void doWait(){
    synchronized(myMonitorObject){
      if(!wasSignalled){
        try{
          myMonitorObject.wait();
        } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
```

```
        }
    }

    public void doNotify(){
        synchronized(myMonitorObject){
            wasSignalled = true;
            myMonitorObject.notify();
        }
    }
}
```

注意，在调用 notify（）之前，doNotify（）方法将 wasSignaled 变量设置为 true。另外，注意 doWait（）方法在调用 wait（）之前检查 wasSignaled 变量。事实上，只有前一个 doWait（）调用和此调用之间没有收到信号时，才调用 wait（）。

虚假唤醒

使没有调用 notify（）和 notifyAll（），线程也有可能被莫名其妙地唤醒。这就是所谓的虚假唤醒，无缘无故地醒来。

如果 MyWaitNofity2 类的 doWait（）方法中出现虚假唤醒，则等待线程没有接收正确的信号就可以继续处理！这可能会导致应用程序出现严重问题。

为了防止虚假唤醒，要在 while 循环中而不是在 if 语句中检查信号成员变量。这种 while 循环也称为自旋锁。线程唤醒后会旋转，直到自旋锁（while 循环）中的条件变为 false。下面是 MyWaitNotify2 的修改版本，展示了自旋锁的用法：

```
public class MyWaitNotify3{

  MonitorObject myMonitorObject = new MonitorObject();
  boolean wasSignalled = false;

  public void doWait(){
    synchronized(myMonitorObject){
      while(!wasSignalled){
        try{
          myMonitorObject.wait();
        } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
    }
  }

  public void doNotify(){
    synchronized(myMonitorObject){
```

```
            wasSignalled = true;
            myMonitorObject.notify();
        }
    }
}
```

注意 wait（）调用现在是嵌套在 while 循环中而不是 if 语句中。如果等待的线程在没有收到信号的情况下唤醒，则成员变量 wassignaled 仍然为 false，while 循环将再次执行，从而导致唤醒的线程返回到等待状态。

多个线程等待同一个信号

如果有多个线程在等待，while 循环也是一个不错的解决方案，这些线程都是使用 notifyAll（）唤醒的，但是应该只允许其中一个线程继续。一次只能有一个线程能够获得 monitor 对象上的锁，这意味着只有一个线程可以退出 wait（）并清除 wassignaled 标志。一旦这个线程在 doWait（）方法中退出 synchronized 块，其他线程就可以退出 wait（）并检查 while 循环中的 wassignaled 成员变量。但是，第一个线程唤醒时清除了该标志，因此其余唤醒的线程返回等待状态，直到下一个信号到达。

不要对常量字符串或全局对象调用 wait（）

本文在以前有一个版本的 MyWaitNotify 示例类使用常量字符串（“”）作为监视对象，如下所示：

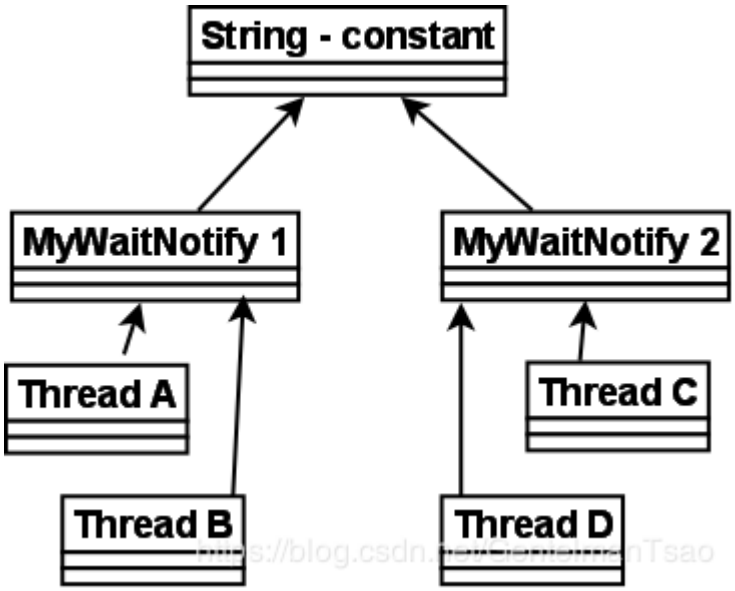
```
public class MyWaitNotify{
```

```
String myMonitorObject = "";
boolean wasSignalled = false;

public void doWait(){
   synchronized(myMonitorObject){
      while(!wasSignalled){
         try{
            myMonitorObject.wait();
          } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
   }
}

public void doNotify(){
   synchronized(myMonitorObject){
      wasSignalled = true;
      myMonitorObject.notify();
   }
}
```

}

对空字符串或常量字符串调用 wait（）和 notify（）的问题是，JVM/编译器在内部将常量字符串转换为同一个对象。这意味着，即使有两个不同的 MyWaitNotify 实例，它们都引用同一个空字符串实例。这也意味着在第一个 MyWaitNotify 实例上调用 doWait（）的线程有可能错误地被第二个 MyWaitNotify 实例上的 doNotify（）调用唤醒。

这种情况如下图所示：



请记住，即使 4 个线程对同一个共享字符串实例调用 wait（）和 notify（），来自 doWait（）和 doNotify（）的信号也分别存储在两个 MyWaitNotify 实例中。对 MyWaitNotify 1 的 doNotify（）调用可能会唤醒在 MyWaitNotify 2 中等待的线程，但该信号将仅存储在 MyWaitNotify 1 中。

起初，这似乎不是什么大问题。毕竟，如果在第二个 MyWaitNotify 实例上调用 doNotify（），那么真正可能发生的是线程 A 和 B 被错误唤醒。此唤醒线程（A 或 B）将在 while 循环中检查其信号，并返回到 waiting，因为在其等待的第一个 MyWaitNotify 实例上并未

调用 doNotify（）。这种情况相当于引起一次虚假觉醒。线程 A 或线程 B 在没有信号通知的情况下被唤醒。但是代码可以处理这个问题，所以线程返回到等待状态。

问题是，由于 doNotify（）只调用了 notify（）而不调用 notifyAll（），即使 4 个线程在同一个字符串实例（空字符串）上等待，也只有一个线程被唤醒。因此，如果一个线程 A 或 B 被唤醒，而信号真正要给的是 C 或 D，那么被唤醒的线程（A 或 B）将检查其信号，发现没有收到信号，然后返回等待。C 或 D 都没有唤醒来检查它们实际应接收的信号，因此信号丢失。这种情况等同于前面描述的丢失信号问题。C 和 D 收到了一个信号，但没有回应。

如果 doNotify（）方法调用了 notifyAll（）而不是 notify（），则所有等待的线程都会依次被唤醒并检查信号。线程 A 和 B 会回到等待状态，但是 C 或 D 中的一个会注意到这个信号，并离开 doWait（）方法。C 和 D 中的另一个将返回等待，因为检测到信号的线程在离开 doWait（）时已将信号清除。

然后，你可能很想总是调用 notifyAll（）而不是 notify（），但这对性能不利。当只有一个线程可以响应信号时，没有理由唤醒所有等待的线程。

所以：不要对 wait（）/notify（）机制使用全局对象、字符串常量等。对于使用对象的构造方法来说，对象应该是唯一的。例如，每个 MyWaitNotify3（前面章节的示例）实例都有自己的 MonitorObject 实例，而不是使用空字符串进行 wait（）/notify（）调用。

（十六）：死锁（DeadLock，线程死锁，数据库死锁）

文章目录

线程死锁

死锁是指两个或多个线程被阻塞，等待获得死锁中其他一些线程所持有的锁。当多个线程同时需要相同的锁，但以不同的顺序获取它们时，可能会发生死锁。

例如，如果线程 1 锁定 A 并试图锁定 B，而线程 2 已经锁定 B 并试图锁定 A，则会出现死锁。线程 1 永远得不到 B，线程 2 永远得不到 A。此外，他们都不会知道。他们将永远被阻塞在他们各自的对象上，A 和 B。这种情况就是死锁。

这种情况如下所示：

```
Thread 1    locks A, waits for B

Thread 2    locks B, waits for A
```

1

下面的示例是在不同实例中调用同步方法的 TreeNode 类：

```
public class TreeNode {


   TreeNode parent     = null;
```

```
List        children = new ArrayList();

public synchronized void addChild(TreeNode child){
   if(!this.children.contains(child)) {
      this.children.add(child);
      child.setParentOnly(this);
   }
}

public synchronized void addChildOnly(TreeNode child){
   if(!this.children.contains(child){
      this.children.add(child);
   }
}

public synchronized void setParent(TreeNode parent){
   this.parent = parent;
   parent.addChildOnly(this);
}

public synchronized void setParentOnly(TreeNode parent){
   this.parent = parent;
```

```
    }
}
```

如果线程（1）调用 parent.addChild(child)方法，另一个线程（2）同时调用 child.setParent(parent)方法，在同一 parent 实例和 child 实例上，可能会发生死锁。可以用下面的伪代码说明这一点：

```
Thread 1: parent.addChild(child); //locks parent
                --> child.setParentOnly(parent);

Thread 2: child.setParent(parent); //locks child
                --> parent.addChildOnly()
```

首先，线程 1 调用 parent.addChild(child)。由于 addChild（）是同步的，因此线程 1 实际上锁定了 parent 对象，使其他线程无法访问。

然后，线程 2 调用 child.setParent(parent)。由于 setParent（）是同步的，因此线程 2 实际上锁定了 child 对象，使其他线程无法访问。

现在 child 对象和 parent 对象都被两个不同的线程锁定了。接下来，线程 1 试图调用 child.setParentOnly() 方法，但 child 对象被线程 2 锁定，因此该方法就阻塞了。线程 2 也尝试调用 parent.addChildOnly()，但 parent 对象被线程 1 锁定，导致线程 2 在该方法上阻塞。现在，两个线程都被阻塞，等待获得另一个线程持有的锁。

注意：两个线程必须同时调用 parent.addChild(child)和 child.setParent(parent)，就像上面描述的那样，在相同的 parent 实例和 child 实例上，才发生死锁。上面的代码可能会在很长一段时间内执行正常，直到突然死锁。

线程真的需要“在同一时刻”锁定。例如，如果线程 1 比线程 2 提前一点，从而锁定了 A 和 B，那么线程 2 在尝试锁定 B 时将被阻塞。也就不会发生死锁。由于线程调度通常是不可预测的，因此无法预测何时发生死锁。只能知道死锁“会”发生。

更复杂的死锁

死锁还可以包含两个以上的线程。这使得它更难被发现。下面是四个线程死锁的示例：

线程 1　锁定 A, 等待 B

线程 2　锁定 B, 等待 C

线程 3　锁定 C, 等待 D

线程 4　锁定 D, 等待 A

线程 1 等待线程 2，线程 2 等待线程 3，线程 3 等待线程 4，线程 4 等待线程 1.

数据库死锁

发生死锁的一种更复杂的情况是数据库业务。一笔数据库业务可能包含许多 SQL 更新请求。当一笔数据库业务更新记录时，该记录将被锁定，使其他数据库业务无法更新，直到第一笔业务完成。因此，同一业务中的各个更新请求可能会锁定数据库中的某些记录。

例如：

Transaction 1, request 1, locks record 1 for update

Transaction 2, request 1, locks record 2 for update

Transaction 1, request 2, tries to lock record 2 for update.

Transaction 2, request 2, tries to lock record 1 for update.

由于锁是在不同的请求中获取的，并且无法全部知道业务所需的所有锁，因此很难检测或防止数据库业务中出现死锁。

（十七）：防范死锁（锁排序，锁超时，死锁检测）

文章目录

在某些情况下，可以防止死锁。我将在本文中描述三种技术：

锁排序

当多个线程需要相同的锁但以不同的顺序获取它们时，就会发生死锁。

如果确保任何线程始终以相同的顺序获取所有锁，则不会出现死锁。看看这个例子：

```
Thread 1:


  lock A

  lock B



Thread 2:


    wait for A

    lock C (when A locked)



Thread 3:


    wait for A

    wait for B

    wait for C
```

如果一个线程，例如线程 3，需要多个锁，它必须按照确定的顺序获取它们。它不能获取序列后面的锁，除非它获得了前面的锁。

例如，线程 2 或线程 3 在锁定 A 之前都不能锁定 C。由于线程 1 持有锁 A，因此线程 2 和 3 必须首先等待锁 A 解锁。然后他们必须成功锁定 A，才能尝试锁定 B 或 C。

锁排序是一种简单而有效的死锁预防机制。但是，只有知道获取某个锁之前所需的所有锁，才能使用它。但并非总是能够知道。

锁超时

另一种防止死锁的机制是对锁请求设置超时，这意味着试图获取锁的线程只会尝试一定时间，然后放弃。如果线程在给定的时间内未能成功获取所有必需的锁，则它将备份、释放所有已获取的锁，等待随机一段时间，然后重试。等待的随机时间量可以让其他试图获取相同锁的线程有机会获取所有锁，从而让应用程序继续运行，而不会被阻塞。

下面的示例是两个线程尝试以不同顺序获取相同的两个锁，线程将备份并重试：

```
Thread 1 locks A

Thread 2 locks B


Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked


Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.


Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well
```

```
Thread 2 waits randomly (e.g. 43 millis) before retrying.
```

在上面的示例中，线程 2 将在线程 1 之前大约 200 毫秒重试获取锁，因此很可能成功获取这两个锁。然后，线程 1 将等待尝试获取锁 A。当线程 2 完成时，线程 1 将能够同时获取这两个锁（除非线程 2 或另一个线程在这期间又获取了锁）。

需要记住的一个问题是，仅仅因为锁超时并不一定意味着线程已经死锁。这也可能是由于持有锁的线程（导致另一个线程超时）需要很长时间才能完成其任务。

此外，如果有大量的线程争夺相同的资源，仍然有可能不断发生线程同时占用该资源的情况，即使超时和备份也无法避免。对于两个线程，等待 0 到 500 毫秒

然后重试，可能不会有问题；但对于 10 或 20 个线程，情况就不同了。此时两个线程等待同样的时间（或时间足够接近以导致问题）然后重试的可能性要高得多。

锁超时机制的一个问题是，无法为 Java 中的同步块设置超时设置。您必须创建一个自定义锁类或使用 java.util.concurrency 包中的某个 Java 5 并发构造。编写自定义锁并不困难，但这不在本文讨论范围之内。 Java 并发教程中的后续文本将涵盖自定义锁。

死锁检测

死锁检测是一种重度的防止死锁的机制，用于无法进行锁排序和锁超时不可行的情况。

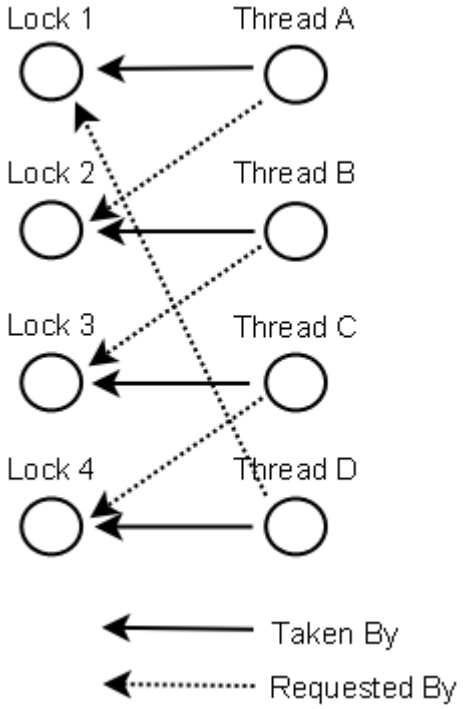
线程每获得一个锁，就会在线程和锁的数据结构（映射，图形等）中进行记录。另外，每当线程请求锁定时，此数据结构中也会对此进行记录。

当线程请求锁定但请求被拒绝时，线程可以遍历锁图以检查死锁。例如，如果线程 A 请求锁 7，但锁 7 由线程 B 持有，则线程 A 可以检查线程 B 是否请求了线程 A 持有的锁（如果有）。如果线程 B 请求了，则表示发生死锁（线程 A 获取了锁 1，请求锁 7，线程 B 获取了锁 7，请求锁 1）。

当然，死锁的情形可能远比两个互相持有锁的线程更复杂。线程 A 可能等待线程 B，线程 B 等待线程 C，线程 C 等待线程 D，线程 D 等待线程 A。为了使线程 A 检测到死锁，它必须可传递地检查线程 B 所请求的所有锁。从线程 B 请求

的锁中，线程 A 将到达线程 C，然后到达线程 D，从线程 D 中找到线程 A 本身持有的一个锁。然后就知道发生了死锁。

以下是 4 个线程（A，B，C 和 D）获取和请求的锁的图表。这样的数据结构可用于检测死锁。



那么，如果检测到死锁，线程将如何处理?

一种可能的操作是释放所有锁，备份，等待随机的一段时间，然后重试。 这类似于简单一点的锁超时机制，只不过线程仅在实际发生死锁时才进行备份，而非仅仅是因为锁定请求超时。 但是，如果许多线程在争用相同的锁，那么即使它们备份并等待，它们也可能反复陷入死锁。

更好的选择是确定或分配线程的优先级，以便仅备份一个（或几个）线程。 其余线程继续获取所需的锁，就好像没有发生死锁一样。 如果分配给线程的优先

级是固定的，则相同的线程将始终被赋予更高的优先级。 为避免这种情况，可以在检测到死锁时随机分配优先级。

翻译花絮

原文：

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up.

说明：

我很佩服中国古人，因为他们一段文章没有一个标点符合，足见肺活量很大。没想到，本文作者的肺活量也很了得。

译文：

另一种防止死锁的机制是对锁请求设置超时，这意味着试图获取锁的线程只会尝试一定时间，然后放弃。

（十八）：饥饿与公平性（线程饥饿，锁，公平锁）

文章目录

如果某个线程因为其他线程抢占了所有时间而没有获得 CPU 时间，则称为“饥饿”。 该线程“因饥饿而死亡”，因为其他线程占用了 CPU 时间而它却得不到。 饥饿的解决方案称为“公平性”——所有线程都被公平地给予执行的机会。

java 中饥饿的原因

以下三个常见原因可能导致 Java 中的线程的饥饿：

1. 高优先级的线程吞噬低优先级的线程的所有 CPU 时间。
2. 线程无限期地等待着进入同步块，因为在它之前始终有其他线程被允许访问同步块。
3. 在某个对象上等待的线程（在该对象上调用 wait（））一直无限期地等待，因为其他线程会不断被唤醒，而轮不到它。

高优先级的线程吞噬低优先级的线程的所有 CPU 时间

你可以分别设置每个线程的线程优先级。 优先级越高，授予线程的 CPU 时间就越多。 你可以设置线程的优先级在 1 到 10 之间。对优先级的具体解释取决于应用程序所运行的操作系统。 对于大多数应用程序，最好不要更改优先级。

线程无限期地等待着进入同步块

Java 的同步代码块可能是造成饥饿的另一个原因。 Java 的同步代码块不能保证允许等待进入同步块的线程进入的顺序。 这意味着在理论上存在一种风险，那就是线程试图进入该块而始终处于阻塞状态，因为其他线程在此之前一直被授予访问权限。 这个问题称为“饥饿”，一个线程被“饿死”是因为其他线程占用了 CPU 时间而不是它。

在某个对象上等待的线程（在该对象上调用 wait（））一直无限期地等待

如果多个线程在某对象上调用了 wait（），则 notify（）方法不能保证哪个线程被唤醒。 可能是任何正在等待的线程。 因此，存在这样的风险，即等待在某个对象上的线程永远不会被唤醒，因为被唤醒的总是其他等待线程而不是它。

java 中实现公平性

尽管不可能在 Java 中实现 100%的公平性，但我们仍然可以实现同步结构来增加线程之间的公平性。

让我们先研究一个简单的同步代码块：

```
public class Synchronizer{

  public synchronized void doSynchronized(){
    //do a lot of work which takes a long time
  }
```

}

如果有多个线程调用 doSynchronized（）方法，则其中的一些线程将被阻塞，直到第一个被授予访问权限的线程离开该方法为止。 如果有多个线程被阻塞等待访问，则不能保证接下来被授予访问权限的是哪个线程。

使用锁替代同步块

为了增加等待线程的公平性，我们首先将代码块更改为由锁保护，而不是由同步块保护：

```
public class Synchronizer{
   Lock lock = new Lock();


   public void doSynchronized() throws InterruptedException{
      this.lock.lock();
         //critical section, do a lot of work which takes a long time
      this.lock.unlock();
   }


}
```

请注意，doSynchronized（）方法不再声明为 synchronized。 相反，临界区由 lock.lock（）和 lock.unlock（）调用保护。

Lock 类的简单实现如下所示：

```
public class Lock{
  private boolean isLocked       = false;
  private Thread  lockingThread = null;

  public synchronized void lock() throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked       = true;
    lockingThread = Thread.currentThread();
  }

  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked       = false;
    lockingThread = null;
    notify();
  }
```

}

查看上面的 Synchronizer 类并研究此 Lock 实现，你会注意到，如果多个线程同时调用 lock（），则线程现在试图访问 lock（）方法时被阻塞。 其次，如果锁被锁定，则线程在 lock（）方法的 while（isLocked）循环内的 wait（）调用中被阻塞。 请记住，调用 wait（）的线程会释放 Lock 实例上的同步锁，因此等待进入 lock（）的线程现在可以这样做。 结果是多个线程最终可能在 lock（）内部调用了 wait（）。

如果回头看 doSynchronized（）方法，您会注意到 lock（）和 unlock（）状态之间的注释，这两个调用之间的代码需要花费很长的时间才能执行。 让我们进一步假设，与进入 lock（）方法和调用 wait（）相比，此代码需要花费较长的时间执行，因为该锁已被锁定。 这意味着在等待锁定锁并进入临界区时，大部分时间都花在了 lock（）方法内部的 wait（）调用中，而不是阻塞在尝试进入 lock（）方法时。

如前所述，如果有多个线程在等待进入同步块，则同步块不能保证授予哪个线程访问权限。 wait（）也不保证在调用 notify（）时唤醒了哪个线程。 因此，当前的 Lock 类版本与 doSynchronized（）的 synchronized 版本在公平性方面没有什么不同。 但是我们可以修改。

当前版本的 Lock 类调用自己的 wait（）方法。 然而，如果每个线程在一个单独的对象上调用 wait（），从而只有一个线程在各自对象上调用了 wait（），则 Lock 类可以决定在哪个对象上调用 notify（），从而实际上准确的选择要唤醒的线程 。

公平锁

下面展示了将原来的 Lock 类变成了一个公平锁，即 FairLock。 你会发现，与前面展示的 Lock 类相比，在同步和 wait（）/ notify（）上的实现有所变化。

确切地说，达成这个设计是从上一个 Lock 类开始的。说来话长，这涉及几个增量设计步骤，每个步骤都解决了上一步的问题：嵌套管程锁死，滑丝条件和信号丢失。 为了使文本简短，该讨论被省略了，但是每个步骤在该主题的相应文章中讨论（请参见上面的链接）。 重要的是，现在每个调用 lock（）的线程都已排队，并且只有队列中的第一个线程才可以锁定 FairLock 实例（如果已解锁）。 所有其他线程将被暂存，直到它们到达队列的头部。

```
public class FairLock {
    private boolean       isLocked       = false;
    private Thread        lockingThread  = null;
    private List<QueueObject> waitingThreads =
            new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject        = new QueueObject();
    boolean     isLockedForThisThread = true;
    synchronized(this){
      waitingThreads.add(queueObject);
    }

    while(isLockedForThisThread){
```

```
        synchronized(this){
          isLockedForThisThread =
                isLocked || waitingThreads.get(0) != queueObject;
          if(!isLockedForThisThread){
            isLocked = true;
              waitingThreads.remove(queueObject);
              lockingThread = Thread.currentThread();
              return;
            }
        }
        try{
          queueObject.doWait();
        }catch(InterruptedException e){
          synchronized(this) { waitingThreads.remove(queueObject); }
          throw e;
        }
    }
}

public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
        throw new IllegalMonitorStateException(
```

```
            "Calling thread has not locked this lock");
        }
        isLocked          = false;
        lockingThread = null;
        if(waitingThreads.size() > 0){
            waitingThreads.get(0).doNotify();
        }
    }
}
public class QueueObject {

    private boolean isNotified = false;

    public synchronized void doWait() throws InterruptedException {
        while(!isNotified){
            this.wait();
        }
        this.isNotified = false;
    }

    public synchronized void doNotify() {
        this.isNotified = true;
```

```
        this.notify();
    }

    public boolean equals(Object o) {
        return this == o;
    }
}
```

首先，你可能会注意到 lock（）方法不再声明为 synchronized， 而是仅将需要同步的块嵌套在 synchronized 块内。

FairLock 创建一个新的 QueueObject 实例，并为每个调用 lock（）的线程排队。 调用 unlock（）的线程将获取队列头部的 QueueObject 并对其调用 doNotify（），以唤醒在该对象上等待的线程。 这样，一次仅唤醒一个等待线程，而不唤醒所有等待线程。 这个部分决定着 FairLock 的公平性。

请注意，同一同步块内仍要测试并设置锁的状态，以免发生滑丝条件。

还要注意，QueueObject 实际上是一个信号量。 doWait（）和 doNotify（）方法将信号存储在 QueueObject 中。 这样做是为了避免导致信号丢失——由于线程在调用 queueObject.doWait（）之前被另一个线程所抢占，而另一个线程又调用 unlock（），从而调用 queueObject.doNotify()。调用 queueObject.doWait()放置在 synchronized（this）块之外，以避免嵌套管程锁死，因此，当在 lock（）方法的 synchronized（this）块内没有线程正在执行时，另一个线程实际上可以调用 unlock（）。

最后，请注意在 try-catch 块内调用了 queueObject.doWait（）。 如果抛出 InterruptedException，线程将离开 lock（）方法，我们需要将其出队。

性能上的说明

如果比较 Lock 和 FairLock 类，你会发现 FairLock 类的 lock（）和 unlock（）内部还有更多操作。 此额外的代码使 FairLock 成为比 Lock 稍慢的同步机制。 它对应用程序的影响取决于 FairLock 保护的临界区中的代码执行的时间。 执行时间越长，同步器增加的开销就越小。 当然，这也取决于此代码被调用的频率。

翻译花絮

原文：

Exactly how I arrived at this design beginning from the previous Lock class is a longer story involving several incremental design steps, each fixing the problem of the previous step: Nested Monitor Lockout,

Slipped Conditions, and

Missed Signals.

说明：

Slipped Conditions，表示没有真正锁住，本文译为滑丝条件，比喻锁打滑了。

译文：

确切地说，达成这个设计是从上一个 Lock 类开始的。说来话长，这涉及几个增量设计步骤，每个步骤都解决了上一步的问题：嵌套管程锁死，**滑丝条件**和信号丢失。

（十九）：嵌套管程锁死

文章目录

嵌套管程锁死是如何发生的

嵌套管程锁死的问题类似于死锁。 嵌套管程锁死是这样发生的：

Thread 1 synchronizes on A

Thread 1 synchronizes on B (while synchronized on A)

Thread 1 decides to wait for a signal from another thread before continuing

Thread 1 calls B.wait() thereby releasing the lock on B, but not A.

Thread 2 needs to lock both A and B (in that sequence)

to send Thread 1 the signal.

Thread 2 cannot lock A, since Thread 1 still holds the lock on A.

Thread 2 remain blocked indefinately waiting for Thread1

to release the lock on A

Thread 1 remain blocked indefinately waiting for the signal from
Thread 2, thereby
never releasing the lock on A, that must be released to make
it possible for Thread 2 to send the signal to Thread 1, etc.

这听起来像是一种理论上的情况，但是请看下面 Lock 的直接实现：

//带有嵌套管程锁死的 lock 实现

```
public class Lock{
   protected MonitorObject monitorObject = new MonitorObject();
   protected boolean isLocked = false;

   public void lock() throws InterruptedException{
      synchronized(this){
         while(isLocked){
            synchronized(this.monitorObject){
                  this.monitorObject.wait();
            }
         }
         isLocked = true;
      }
```

```
    }

    public void unlock(){
        synchronized(this){
            this.isLocked = false;
            synchronized(this.monitorObject){
                this.monitorObject.notify();
            }
        }
    }
}
```

请注意 lock（）方法首先在“ this”上同步，然后在 monitorObject 成员变量上同步。 如果 isLocked 为 false，则没有问题。 该线程不调用 monitorObject.wait（）。 但是，如果 isLocked 为 true，则调用 lock（）的线程将停在 monitorObject.wait（）调用中。

这样做的问题在于，调用 monitorObject.wait（）仅释放了 monitorObject 成员上的同步管程，而不释放与“ this”关联的同步管程。换句话说，刚刚停在等待状态的线程仍持有“ this”上的同步锁。

当首个将 Lock 锁定的线程尝试调用 unlock（）对其进行解锁时，将在尝试进入 unlock（）方法中的 synchronized(this)代码块时被阻塞。 它将一直保持阻塞状态，直到在 lock（）中等待的线程离开了 synchronized（this）为止。 但是，在 lock（）方法中等待的线程不

会离开该块，直到 isLocked 设置为 false，并执行 monitorObject.notify（），而这正是 unlock（）要做的。

简而言之，等待 lock（）的线程需要成功执行一次 unlock（），以使其退出 lock（）和其中的同步块。 但是，在 lock()中等待的线程离开外部同步块之前，没有任何线程可以实际执行 unlock（）。

结果是，任何调用 lock（）或 unlock（）的线程都将永远阻塞。 这称为嵌套管程锁死。

一个更现实的例子

你可能会说，你永远不会照上面的方式实现一个锁。 因为你不会在内部管程对象上调用 wait（）和 notify（），而是在 this 上调用。很可能是这样。 但是在某些情况下，可能会出现上述设计。 例如，如果要在 Lock 中实现公平性。 这样做时，你希望每个线程在各自的队列对象上调用 wait（），以便可以一次通知一个线程。

看一下这种公平锁的简单实现：

```
//Fair Lock implementation with nested monitor lockout problem


public class FairLock {
   private boolean       isLocked       = false;
   private Thread        lockingThread  = null;
   private List<QueueObject> waitingThreads =
                 new ArrayList<QueueObject>();
```

```
public void lock() throws InterruptedException{
  QueueObject queueObject = new QueueObject();

  synchronized(this){
    waitingThreads.add(queueObject);

    while(isLocked || waitingThreads.get(0) != queueObject){

      synchronized(queueObject){
        try{
          queueObject.wait();
        }catch(InterruptedException e){
          waitingThreads.remove(queueObject);
          throw e;
        }
      }
    }
    waitingThreads.remove(queueObject);
    isLocked = true;
    lockingThread = Thread.currentThread();
  }
```

```
    }

    public synchronized void unlock(){
        if(this.lockingThread != Thread.currentThread()){
            throw new IllegalMonitorStateException(
                "Calling thread has not locked this lock");
        }
        isLocked      = false;
        lockingThread = null;
        if(waitingThreads.size() > 0){
            QueueObject queueObject = waitingThreads.get(0);
            synchronized(queueObject){
                queueObject.notify();
            }
        }
    }
}
public class QueueObject {}
```

乍看起来，此实现看起来不错，但请注意 lock（）方法调用了 queueObject.wait（），该调用在两个同步块内部。 一个在“ this”上同步，另一个嵌套在其中，在 queueObject 局部变量上同步。 当线程调用 queueObject.wait（）时，它释放 QueueObject 实例上的锁，但不释放与“ this”关联的锁。

还要注意，unlock（）方法声明为 synchronized，它相当于 synchronized（this）块。 这意味着，如果线程在 lock（）中等待，则与“ this”关联的管程对象将被该等待的线程锁定。所有调用 unlock（）的线程将无限期地阻塞，等待正在等待的线程释放对“ this”的锁定。 但这永远不会发生，因为只有在线程成功将信号发送到等待的线程时才会发生，而发送信号只能通过执行 unlock（）方法。

因此，上面的 FairLock 实现可能导致嵌套管程锁死。在《饥饿和公平性》篇中讲解了如何更好的实现公平锁。

嵌套管程锁死 vs 死锁

嵌套管程锁死和死锁的结果几乎相同：所涉及的线程最终被阻塞，永远等待对方。

但这两种情况并不等价。 如关于死锁篇中所述，当两个线程以不同顺序获得锁时，就会发生死锁。 线程 1 锁定 A，等待 B。线程 2 锁定 B，然后等待 A。如防死锁篇所述，可以通过始终以相同顺序锁定锁（锁定顺序）来避免死锁。 但是，嵌套管程锁死正是由两个线程以相同顺序进行锁定而发生。 线程 1 锁定 A 和 B，然后释放 B 并等待来自线程 2 的信号。线程 2 同时需要 A 和 B 才能向线程 1 发送信号。 结果就是，一个线程正在等待信号，另一个线程正在等待释放锁。

区别总结如下：

- 在死锁中，两个线程等待彼此释放锁。

- 嵌套管程锁死中，线程 1 持有锁 A，然后等待

  来自线程 2 的信号。线程 2 需要锁 A 来发送

  信号给线程 1。

翻译花絮

原文:

At first glance this implementation may look fine, but **notice how** the lock()

method calls

queueObject.wait(); from inside two synchronized blocks.

说明:

“notice how”很容易译成“注意 xxx 是如何”，其实并非如此。这里的“how”不是“如何”，而是引导词，后接宾语从句。所以不需要直译出来。

译文:

乍看起来，此实现看起来不错，但**请注意** lock（）方法调用了 queueObject.wait（），该调用在两个同步块内部。

（二十）：滑移条件（Slipped Conditions）

文章目录

什么是滑移条件?

滑移条件是指，一个线程从检查了某个条件到对其执行操作的时间内，该条件被另一个线程更改，从而使第一个线程操作出错。 下面是一个简单的示例:

```
public class Lock {


    private boolean isLocked = true;


    public void lock(){
      synchronized(this){
        while(isLocked){
          try{
            this.wait();
          } catch(InterruptedException e){
            //do nothing, keep waiting
          }
```

```
            }
        }

        synchronized(this){
            isLocked = true;
        }
    }

    public synchronized void unlock(){
        isLocked = false;
        this.notify();
    }

}
```

请注意 lock（）方法包含两个同步块。 第一个块等待，直到 isLocked 为 false。 第二个块将 isLocked 设置为 true，以锁定 Lock 实例，不让其他线程访问。

假设 isLocked 为 false，并且两个线程同时调用 lock（）。 如果第一个线程刚好抢占并进入第一个同步块，则该线程将检查 isLocked 并发现它为 false。 如果现在允许第二个线程执行，从而进入第一个同步块，则该线程也将看到 isLocked 是 false。 现在，两个线程读到的条件都是 false。 然后，两个线程都将进入第二个同步块，将 isLocked 设置为 true 并继续执行。

这种情况是滑移条件的一个例子。 两个线程都检测条件，然后退出同步块，从而允许其他线程检测该条件，之后两个线程中的一个才为后续线程更改条件。 换句话说，从检查条件开始到该线程为后续线程更改条件之前，条件一直在打滑。

为了避免滑移条件，条件的测试和设置必须由执行该操作的线程自动进行，这意味着，在第一个线程测试和设置条件的时间内，其他线程不可以检测该条件。

上面示例中的解决方案很简单。 只需将 isLocked = true；这行移到第一个同步块中，在 while 循环之后。 像下面这样：

```
public class Lock {


    private boolean isLocked = true;


    public void lock(){
       synchronized(this){
          while(isLocked){
             try{
                this.wait();
             } catch(InterruptedException e){
                //do nothing, keep waiting
             }
          }
          isLocked = true;
```

```
        }
    }

    public synchronized void unlock(){
        isLocked = false;
        this.notify();
    }

}
```

现在，isLocked 条件的测试和设置是在同一个同步块内部自动完成的。

更现实的示例

你可能会理直气壮地说，你永远不会像本文中所示的第一个实现那样实现 Lock，因此断言滑移条件只不过是理论上存在的问题。 但是第一个示例之所以设计得这么简单是为了更好地传达滑移条件的概念。

一个更现实的例子是在公平锁的实现中，如《饥饿和公平篇》中所述。 如果我们看看《嵌套管程锁死篇》中的那个稚拙的实现，并尝试消除嵌套管程锁死问题，那么很容易得出带有滑移条件的实现。 首先，我们看一下《嵌套管程锁死》篇中的示例：

```
//公平锁的实现，存在嵌套管程锁定问题

public class FairLock {
```

```
private boolean    isLocked  = false;
private Thread     lockingThread = null;
private List<QueueObject> waitingThreads =
          new ArrayList<QueueObject>();

public void lock() throws InterruptedException{
  QueueObject queueObject = new QueueObject();

  synchronized(this){
    waitingThreads.add(queueObject);

    while(isLocked || waitingThreads.get(0) != queueObject){

      synchronized(queueObject){
        try{
          queueObject.wait();
        }catch(InterruptedException e){
          waitingThreads.remove(queueObject);
          throw e;
        }
      }
    }
```

```
            waitingThreads.remove(queueObject);
            isLocked = true;
            lockingThread = Thread.currentThread();
        }
    }

    public synchronized void unlock(){
        if(this.lockingThread != Thread.currentThread()){
            throw new IllegalMonitorStateException(
                "Calling thread has not locked this lock");
        }
        isLocked      = false;
        lockingThread = null;
        if(waitingThreads.size() > 0){
            QueueObject queueObject = waitingThread.get(0);
            synchronized(queueObject){
                queueObject.notify();
            }
        }
    }
}
public class QueueObject {}
```

请注意，synchronized（queueObject）连同其 queueObject.wait（）调用嵌套在 synchronized（this）块内，从而导致嵌套管程锁死问题。 为避免此问题，必须将 synchronized（queueObject）块移到 Synchronized（this）块之外。 像下面这样：

```
//公平锁的实现，存在滑移条件问题

public class FairLock {
   private boolean                isLocked          = false;
   private Thread                 lockingThread    = null;
   private List<QueueObject> waitingThreads =
                   new ArrayList<QueueObject>();

   public void lock() throws InterruptedException{
      QueueObject queueObject = new QueueObject();

      synchronized(this){
         waitingThreads.add(queueObject);
      }

      boolean mustWait = true;
      while(mustWait){

         synchronized(this){
```

```
            mustWait = isLocked || waitingThreads.get(0) != queueObject;
        }

        synchronized(queueObject){
            if(mustWait){
                try{
                    queueObject.wait();
                }catch(InterruptedException e){
                    waitingThreads.remove(queueObject);
                    throw e;
                }
            }
        }

        synchronized(this){
            waitingThreads.remove(queueObject);
            isLocked = true;
            lockingThread = Thread.currentThread();
        }
    }
}
```

注意：仅展示 lock（）方法，因为我只修改了这个方法。

请注意，lock（）方法现在包含 3 个同步块。

第一个 sync（this）块通过设置 mustWait = isLocked || waitingThreads.get(0) != queueObject 来检查条件。

第二个 synchronized（queueObject）块检查线程是否要等待。 此时，另一个线程可能已经解除了锁定，但是我们暂且不管。 假设锁已解锁，那么线程立即退出了 synchronized（queueObject）块。

第三个 synchronized（this）块仅当 mustWait = false 时才执行。 这会将条件 isLocked 设置回 true，并离开 lock（）方法。

设想，如果两个线程在锁解锁时同时调用 lock（）将会发生什么。首先，线程 1 将检查 isLocked 条件，并发现它是 false。 然后线程 2 将执行相同的操作。 然后它们都不等待，并且都将状态 isLocked 设置为 true。 这是滑移条件的典型例子。

消除滑移条件问题

要从上面的示例中消除滑移条件问题，必须将最后一个 synchronized（this）块的内容上移到第一个块中。 当然，为了适应这一变化，代码自然也必须稍作更改。 像下面这样：

```
//公平锁的实现，不存在嵌套管程锁死问题
//但存在信号丢失问题
```

```
public class FairLock {
   private boolean                isLocked       = false;
   private Thread                 lockingThread  = null;
   private List<QueueObject> waitingThreads =
                   new ArrayList<QueueObject>();

   public void lock() throws InterruptedException{
      QueueObject queueObject = new QueueObject();

      synchronized(this){
         waitingThreads.add(queueObject);
      }

      boolean mustWait = true;
      while(mustWait){


            synchronized(this){
                  mustWait = isLocked || waitingThreads.get(0) !=
queueObject;
                  if(!mustWait){
                        waitingThreads.remove(queueObject);
```

```
                isLocked = true;
                lockingThread = Thread.currentThread();
                return;
            }
        }

        synchronized(queueObject){
          if(mustWait){
            try{
              queueObject.wait();
            }catch(InterruptedException e){
              waitingThreads.remove(queueObject);
              throw e;
            }
          }
        }
      }
    }
}
```

注意，现在在同一个同步代码块中测试和设置局部变量 mustWait。 还要注意，即使在 synchronized(this)代码块之外也对 mustWait 局部变量进行了检查，在 while(mustWait)子句中，mustWait 变量的值也不会在 synchronized(this)之外被改变。一个将 mustWait

判断为 false 的线程也会自动设置内部条件（isLocked），所以其他线程检查该条件都回判断为 true。

synchronized(this)块中的 return 语句不是必须的，而只是一个小的优化。如果线程不必等待（mustWait == false），则没有理由进入 synchronized（queueObject）块并执行 if（mustWait）子句。

细心的读者会注意到，上述公平锁的实现仍然存在信号丢失的问题。 设想，当线程调用 lock（）时，FairLock 实例被锁定。 在第一个 synchronized(this)块之后 mustWait 为 true。再设想一下，调用 lock（）的线程被抢占了，而锁定了锁的线程则调用了 unlock（）。 如果查看前面所示的 unlock（）实现，会注意到它调用 queueObject.notify（）。 但是，由于在 lock（）中等待的线程尚未调用 queueObject.wait（），因此 queueObject.notify（）无法被接收到。 信号丢失了。 当线程在调用 queueObject.wait（）之后立即调用 lock（）时，它将保持阻塞状态，直到其他线程调用 unlock（），这可能永远也等不到。

信号丢失问题的原因在"饥饿和公平篇"中的 FairLock 实现有提到。在该篇中将 QueueObject 类改为一个信号量，有两个方法：doWait（）和 doNotify（）。 这些方法在 QueueObject 内部存储并响应信号。 这样，即使在 doWait（）之前调用了 doNotify（），也不会丢失信号。

翻译花絮：

原文:

Both threads test the condition, then exit the synchronized block, **thereby** allowing

other threads to test the condition, **before** any of the two first threads change the conditions for subsequent threads.

解析：

“before”直译为“在…之前”。在原文中，thereby 这句表达了递进的意思，before 后接的是状语。但对于中文来说，这段话无论状语前置还是后置，都无法自然的表达出递进的意思。本译文将“before”译为“之后…才”，这样一来也消除了状语，使语句衔接更自然。

译文：

两个线程都检测条件，然后退出同步块，从而允许其他线程检测该条件，**之后**两个线程中的一个**才**为后续线程更改条件。

（二十一）：Java 中的锁（普通锁，可重入锁，公平锁）

文章目录

锁是一种类似于同步块的线程同步机制，但是锁比 Java 的同步块更复杂。 锁（以及其他更高级的同步机制）是使用同步块创建的，因此我们无法完全抛弃 synchronized 关键字。

从 Java 5 开始，包 java.util.concurrent.locks 包含多个锁实现，因此不需要再实现自己的锁。 但是你仍然需要知道如何使用它们，并且了解其实现背后的理论仍然很有用。 有关更多详细信息，请参见我在 java.util.concurrent.locks.Lock 接口上的教程。

简单的锁

让我们从 Java 代码的同步块开始看：

```
public class Counter{

   private int count = 0;

   public int inc(){
      synchronized(this){
         return ++count;
      }
   }
}
```

注意 inc（）方法中的 synchronized（this）块。 此块可确保一次只有一个线程可以执行 return ++ count。 同步块中的代码本来可以更复杂，但是用简单的++ count 足以说明要点。

Counter 类还可以这样写，使用 Lock 而不是同步块：

```
public class Counter{

    private Lock lock = new Lock();
    private int count = 0;

    public int inc(){
        lock.lock();
        int newCount = ++count;
        lock.unlock();
        return newCount;
    }
}
```

lock（）方法锁定 Lock 实例，因而所有调用 lock（）的线程都被阻塞，直到执行 unlock（）为止。

这是一个简单的 Lock 实现：

```
public class Lock{

    private boolean isLocked = false;

    public synchronized void lock()
    throws InterruptedException{
```

```
        while(isLocked){
            wait();
        }
        isLocked = true;
    }

    public synchronized void unlock(){
        isLocked = false;
        notify();
    }
}
```

注意 while（isLocked）循环，也称为“自旋锁”。 自旋锁以及方法 wait（）和 notify（）在“**线程信号**”一文中有详细介绍。 当 isLocked 为 true 时，调用 lock（）的线程将停在 wait（）中等待。 万一线程在没有收到 notify（）调用的情况下从 wait（）中意外返回（也称为“**虚假唤醒**”），则该线程会重新检查 isLocked 条件以查看是否可以安全进行，而不是仅仅假定被唤醒就表示可以安全进行。如果 isLocked 为 false，则线程退出 while（isLocked）循环，并将 isLocked 设置为 true，以锁定 Lock 实例而不给其他调用 lock（）的线程使用。

当线程执行完**临界区**中的代码（lock（）和 unlock（）之间的代码），线程将调用 unlock（）。 执行 unlock（）会将 isLocked 设置为 false，并通知（唤醒）在 lock（）方法中的 wait（）中等待的某个线程（如果有的话）。

锁的可重入性

Java 中的同步块是可重入的。 这意味着，如果 Java 线程进入了同步的代码块，从而锁定了同步该块的管程对象，则该线程可以进入在同一管程对象上同步的其他 Java 代码块。 见下面的例子：

```
public class Reentrant{

   public synchronized outer(){
      inner();
   }

   public synchronized inner(){
      //do something
   }
}
```

请注意，outer ( ) 和 inner ( ) 都被声明为 synchronized，这在 Java 中等效于 synchronized（this）块。 如果线程调用 outer()，则从 outer()内部调用 inner（）没问题，因为两个方法（或块）都在同一个管程对象（“ this”）上同步。 如果线程已经拥有管程对象上的锁，则它可以访问在同一管程对象上同步的所有的块。 这称为重入性。 线程可以重新进入已经为其持有锁的任何代码块。

前面所示的锁实现不是可重入的。 如果我们像下面那样重写 Reentrant 类，则调用 outer() 的线程将阻塞在 inner（）方法的 lock.lock（）内部。

```
public class Reentrant2{

   Lock lock = new Lock();

   public outer(){
      lock.lock();
      inner();
      lock.unlock();
   }

   public synchronized inner(){
      lock.lock();
      //do something
      lock.unlock();
   }
}
```

调用 outer()的线程将首先锁定 Lock 实例。 然后它将调用 inner（）。 在 inner（）方法内部，线程将再次尝试锁定 Lock 实例。 这将失败（这意味着线程将被阻塞），因为 Lock 实例已在 outside（）方法中被锁定了。

线程第二次调用 lock（）时，因为没有先调用 unlock（）而被阻塞，这个原因查看 lock（）实现就很明显了。

```
public class Lock{

   boolean isLocked = false;

   public synchronized void lock()
   throws InterruptedException{
      while(isLocked){
         wait();
      }
      isLocked = true;
   }

   ...
}
```

是否允许线程退出 lock（）方法是由 while 循环（自旋锁）中的条件决定的。 当前的条件是 isLocked 必须为 false 才能允许此操作，而不管由哪个线程锁定。

要使 Lock 类可重入，我们需要做一些小改动：

```
public class Lock{
```

```
boolean isLocked = false;
Thread    lockedBy = null;
int        lockedCount = 0;

public synchronized void lock()
throws InterruptedException{
   Thread callingThread = Thread.currentThread();
   while(isLocked && lockedBy != callingThread){
      wait();
   }
   isLocked = true;
   lockedCount++;
   lockedBy = callingThread;
}


public synchronized void unlock(){
   if(Thread.curentThread() == this.lockedBy){
      lockedCount--;

      if(lockedCount == 0){
```

```
            isLocked = false;
            notify();
          }
        }
    }

    ...
}
```

请注意，while 循环（自旋锁）现在将锁定 Lock 实例的线程也考虑在内。 如果锁被解锁（isLocked = false）或调用线程是锁定 Lock 实例的线程，则 while 循环将不会执行，因而调用 lock（）的线程能够退出该方法。

另外，我们需要计算锁被同一线程锁定的次数。 否则，即使该锁已被多次锁定，一次调用 unlock（）也会解锁该锁。 除非锁定该锁的线程调用 unlock（）的次数 lock（）一样，否则我们不希望锁被解锁。

lock 类现在是可重入的了。

锁的公平性

Java 的同步块无法保证尝试进入同步块的线程的访问顺序。 因此，如果许多线程一直在争夺对同一同步块的访问权，一个或多个线程有可能永远得不到访问权——该访问权始终会授予其他线程。 这称为饥饿。 为了避免这种情况，锁应该实现公平性。 由于本文中示例

的 Lock 实现在内部使用同步块，因此它们不能保证公平性。 关于饥饿和公平性，在《**饥饿和公平性**》篇中有更详细的讨论。

从 finally 子句中调用 unlock（）

当用 Lock 保护临界区时，临界区可能会引发异常，因此一定要从 finally 子句内部调用 unlock（）方法。 这样做可以确保锁可以被解锁，以便其他线程可以锁定它。 这是一个例子：

```
lock.lock();
try{
   //do critical section code, which may throw exception
} finally {
   lock.unlock();
}
```

这个小结构确保万一临界区中的代码引发异常时，锁可以解锁。 如果不从 finally 子句内部调用 unlock（），并且从临界区抛出了异常，则 Lock 将永远保持锁定状态，从而导致在该 Lock 实例上调用 lock（）的所有线程无限期地暂停。

（二十二）：Java 中的读/写锁（可重入锁，完全可重入的 ReadWriteLock）

文章目录

Java 读/写锁的实现

读/写锁的可重入性

读锁重入

写锁重入

读锁到写锁重入

写锁到读锁重入

完全可重入的 ReadWriteLock

从 finally 子句中调用 unlock（）

翻译花絮

Java 中的读/写锁比《java 中的锁》一文中示例的 Lock 实现更复杂。设想，你有一个应用程序可以读写一些资源，但是写资源操作没有读取资源那样多。 两个线程读取同一资源不会彼此造成问题，因此，要读取资源的多个线程可以同时被授予访问权限，这可以重叠。 但是，如果某个线程想要写入资源，则在同一时刻不可以进行其他读取或写入。 要解决允许多个读线程但只有一个写线程的问题，你需要一个读/写锁。

Java 5 在 java.util.concurrent 包中附带了读/写锁定实现。 即使这样，了解其实现背后的理论仍然是有用的。

Java 读/写锁的实现

首先，让我们总结一下获得资源的读写访问权的条件：

读取访问权

没有线程在写，并且没有线程请求写访问。

写入访问权

没有线程正在读取或写入。

如果线程想要读取资源，只要没有线程正在写入资源，并且没有线程请求对资源进行写访问，则可以。 我们只需优先处理写访问请求，即假设写请求比读请求更重要。 此外，如果读取操作最频繁，而我们没有提升写入优先级，则可能会发生饥饿。 请求写访问权限的线程会被阻塞，直到所有读线程都解锁了 ReadWriteLock 为止，。 如果新线程不断地获取到读取访问权限，则等待写入访问权限的线程将始终保持阻塞状态，从而导致饥饿。 因此，仅当没有线程正锁定 ReadWriteLock 进行写入，或请求锁定 ReadWriteLock 进行写入时，才可以授予线程读访问权限。

当没有线程在读或写资源时，想要获得写资源权限的线程可以被授予相应的权限。 不用考虑有多少个线程以及以哪种顺序请求了写访问权限，除非您想保证请求写访问的线程之间的公平性。

牢记这些简单的规则，我们可以实现一个如下的 ReadWriteLock：

```
public class ReadWriteLock{


  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;


  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
```

```
    readers++;
}

public synchronized void unlockRead(){
    readers--;
    notifyAll();
}

public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
        wait();
    }
    writeRequests--;
    writers++;
}

public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
}
```

}

ReadWriteLock 有两个锁定方法和两个解锁方法。 一种锁定和解锁方法用于读访问，一种锁定和解锁方法用于写访问。

读取访问的规则在 lockRead（）方法中实现。 除非存在具有写访问权限的线程，或者一个或多个线程请求写访问权限，否则所有线程都具有读访问权限。

用于写访问的规则在 lockWrite（）方法中实现。 想要写访问权限的线程通过请求写访问权限（writeRequests ++）开始。 然后它将检查它是否真的可以获取写访问权限。 如果没有线程具有对资源的读访问权，也没有线程具有对资源的写访问权，则线程可以获得写访问权限。 有多少个线程请求写访问权限无关紧要。

值得注意的是，unlockRead（）和 unlockWrite（）都调用 notifyAll（）而不是 notify（）。 为什么要这么做？请设想以下情形：

在 ReadWriteLock 内部，有等待读取的线程和等待写入的线程。 如果 notify（）唤醒的线程是读线程，则它会返回等待状态，因为有线程在等待写访问权限。 但是，等待写入的线程没有一个被唤醒，因此其他什么也没发生。 没有任何线程获得读或写访问权限。 通过调用 noftifyAll（），所有正在等待的线程都将被唤醒，并检查它们是否可以获得所需的访问权限。

调用 notifyAll（）还有一个优点。 如果有多个线程正在等待读取权限，而没有线程在等待写入权限，并且已调用了 unlockWrite（），则所有等待读取的线程都会被立即授予读取访问权限，而不是一个接一个。

读/写锁的可重入性

前面示例的 ReadWriteLock 类不是可重入的。 如果具有写访问权的线程再次请求它，则它将阻塞，因为已经有一个写线程了——它自己。 此外，请考虑这种情况：

线程 1 获得读取访问权限。

线程 2 请求写访问权限，但由于有一个读线程而被阻塞。

线程 1 重新请求读取访问权限（重新进入该锁），但由于存在写请求而被阻塞。

在这种情况下，上面的 ReadWriteLock 将被锁定——这种情况类似于死锁。 不管是请求读取还是写入，线程都无法获得权限。

要使 ReadWriteLock 可重入，必须进行一些更改。 分别处理读线程和写线程的可重入性。

读锁重入

为了使 ReadWriteLock 对读线程可重入，我们首先要建立读锁重入的规则：

如果线程可以获取读取访问权限（无写入线程或写入请求），或者它已经拥有读取访问权限（不管有没有写入请求），则授予该线程读锁重入权限。

为了确定某个线程是否已经具有读取访问权限，将每个已授予读取访问权限的线程的引用保留在 Map 中，还包含其已获得读取锁定的次数。 在确定是否可以授予读取访问权限时，将检查此 Map 以获取调用线程的引用。 下面是修改后的 lockRead () 和 unlockRead () 方法：

```
public class ReadWriteLock{


  private Map<Thread, Integer> readingThreads =
      new HashMap<Thread, Integer>();


  private int writers            = 0;
  private int writeRequests    = 0;
```

```
public synchronized void lockRead() throws InterruptedException{
   Thread callingThread = Thread.currentThread();
   while(! canGrantReadAccess(callingThread)){
      wait();
   }

   readingThreads.put(callingThread,
       (getAccessCount(callingThread) + 1));
}



public synchronized void unlockRead(){
   Thread callingThread = Thread.currentThread();
   int accessCount = getAccessCount(callingThread);
   if(accessCount == 1){ readingThreads.remove(callingThread); }
   else { readingThreads.put(callingThread, (accessCount -1)); }
   notifyAll();
}



private boolean canGrantReadAccess(Thread callingThread){
   if(writers > 0)                  return false;
```

```
        if(isReader(callingThread) return true;
        if(writeRequests > 0)          return false;
        return true;
    }

    private int getReadAccessCount(Thread callingThread){
        Integer accessCount = readingThreads.get(callingThread);
        if(accessCount == null) return 0;
        return accessCount.intValue();
    }

    private boolean isReader(Thread callingThread){
        return readingThreads.get(callingThread) != null;
    }

}
```

如你所见，读锁重入仅在当前没有线程写入资源时才被授予。 此外，如果调用线程已经具有读取访问权限，则此优先级高于所有写入请求。

写锁重入

仅当线程已经拥有写访问权限时，才授予写锁重入权限。 以下是修改后 lockWrite（）和 unlockWrite（）方法：

```
public class ReadWriteLock{
```

```
    private Map<Thread, Integer> readingThreads =
            new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writeAccesses--;
    if(writeAccesses == 0){
```

```
            writingThread = null;
        }
        notifyAll();
    }

    private boolean canGrantWriteAccess(Thread callingThread){
        if(hasReaders())                return false;
        if(writingThread == null)       return true;
        if(!isWriter(callingThread)) return false;
        return true;
    }

    private boolean hasReaders(){
        return readingThreads.size() > 0;
    }

    private boolean isWriter(Thread callingThread){
        return writingThread == callingThread;
    }
}
```

注意，在确定调用线程是否可以得到写访问权限时，现在要考虑当前持有写锁的线程。

读锁到写锁重入

有时，拥有读取访问权限的线程也需要获得写入访问权限。 为此，线程必须是唯一的读线程。 为了实现这一点，要稍微更改 writeLock（）方法。 像下面这样：

```
public class ReadWriteLock{


    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();


    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;


  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }
```

```
public synchronized void unlockWrite() throws InterruptedException{
   writeAccesses--;
   if(writeAccesses == 0){
      writingThread = null;
   }
   notifyAll();
}

private boolean canGrantWriteAccess(Thread callingThread){
   if(isOnlyReader(callingThread))       return true;
   if(hasReaders())                      return false;
   if(writingThread == null)             return true;
   if(!isWriter(callingThread))          return false;
   return true;
}

private boolean hasReaders(){
   return readingThreads.size() > 0;
}

private boolean isWriter(Thread callingThread){
   return writingThread == callingThread;
```

```
    }

    private boolean isOnlyReader(Thread thread){
        return readers == 1 && readingThreads.get(callingThread) != null;
        }

}
```

现在，ReadWriteLock 类是读锁到写锁访问可重入的。

写锁到读锁重入

有时具有写访问权的线程也需要读访问权。 如果写线程请求读访问权，则应始终授予。 如果一个线程拥有写访问权限，则其他线程都不能有读或写访问权限，因此这样做并不危险。

更改后的 canGrantReadAccess（）方法如下所示：

```
public class ReadWriteLock{

    private boolean canGrantReadAccess(Thread callingThread){
        if(isWriter(callingThread)) return true;
        if(writingThread != null)     return false;
        if(isReader(callingThread)    return true;
        if(writeRequests > 0)           return false;
        return true;
    }
```

}

完全可重入的 ReadWriteLock

下面是完全可重入的 ReadWriteLock 实现。 我对访问条件进行了一些重构，以使它们更易于阅读，从而更容易使自己确信它们是正确的。

```
public class ReadWriteLock{


  private Map<Thread, Integer> readingThreads =
          new HashMap<Thread, Integer>();


   private int writeAccesses    = 0;
   private int writeRequests    = 0;
   private Thread writingThread = null;



  public synchronized void lockRead() throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(! canGrantReadAccess(callingThread)){
      wait();
    }

    readingThreads.put(callingThread,
      (getReadAccessCount(callingThread) + 1));
```

```
}


private boolean canGrantReadAccess(Thread callingThread){
   if( isWriter(callingThread) ) return true;
   if( hasWriter()              ) return false;
   if( isReader(callingThread) ) return true;
   if( hasWriteRequests()       ) return false;
   return true;
}



public synchronized void unlockRead(){
   Thread callingThread = Thread.currentThread();
   if(!isReader(callingThread)){
      throw new IllegalMonitorStateException("Calling Thread does not" +
          " hold a read lock on this ReadWriteLock");
   }
   int accessCount = getReadAccessCount(callingThread);
   if(accessCount == 1){ readingThreads.remove(callingThread); }
   else { readingThreads.put(callingThread, (accessCount -1)); }
   notifyAll();
}
```

```
public synchronized void lockWrite() throws InterruptedException{
   writeRequests++;
   Thread callingThread = Thread.currentThread();
   while(! canGrantWriteAccess(callingThread)){
      wait();
   }
   writeRequests--;
   writeAccesses++;
   writingThread = callingThread;
}


public synchronized void unlockWrite() throws InterruptedException{
   if(!isWriter(Thread.currentThread()){
      throw new IllegalMonitorStateException("Calling Thread does not" +
         " hold the write lock on this ReadWriteLock");
   }
   writeAccesses--;
   if(writeAccesses == 0){
      writingThread = null;
   }
   notifyAll();
```

```
}

private boolean canGrantWriteAccess(Thread callingThread){
   if(isOnlyReader(callingThread))       return true;
   if(hasReaders())                      return false;
   if(writingThread == null)             return true;
   if(!isWriter(callingThread))          return false;
   return true;
}


private int getReadAccessCount(Thread callingThread){
   Integer accessCount = readingThreads.get(callingThread);
   if(accessCount == null) return 0;
   return accessCount.intValue();
}


private boolean hasReaders(){
   return readingThreads.size() > 0;
}
```

```
    private boolean isReader(Thread callingThread){
        return readingThreads.get(callingThread) != null;
    }

    private boolean isOnlyReader(Thread callingThread){
        return readingThreads.size() == 1 &&
                readingThreads.get(callingThread) != null;
    }

    private boolean hasWriter(){
        return writingThread != null;
    }

    private boolean isWriter(Thread callingThread){
        return writingThread == callingThread;
    }

    private boolean hasWriteRequests(){
        return this.writeRequests > 0;
    }

}
```

从 finally 子句中调用 unlock（）

当使用 ReadWriteLock 保护临界区时，临界区可能会抛出异常，所以应该从从 finally 子句内部调用 readUnlock（）和 writeUnlock（）方法。 这样做可以确保 ReadWriteLock 可以解锁，以便其他线程可以锁定它。 下面是一个例子：

```
lock.lockWrite();
try{
   //do critical section code, which may throw exception
} finally {
   lock.unlockWrite();
}
```

这个小结构可以确保在临界区的代码中抛出异常时，ReadWriteLock 可以解锁。 如果未从 finally 子句中调用 unlockWrite（），并且从临界区抛出了异常，则 ReadWriteLock 将永远保持写锁定，从而导致该 ReadWriteLock 实例上调用 lockRead（）或 lockWrite（）的所有线程一直暂停 。 再次解锁 ReadWriteLock 的唯一方法是在 ReadWriteLock 可重入的前提下，引发异常时锁定该锁的线程随后又成功锁定它，执行临界区并随后再次调用 unlockWrite（）。 那样才能再次解锁 ReadWriteLock。 但是，为什么要等这种情况发生呢？ 如果发生异常怎么办？从 finally 子句中调用 unlockWrite（）是一个更可靠的解决方案。

翻译花絮

原文：

By up-prioritizing write-access requests we assume that write requests are more important than read-requests

解析：

“by”，以...方式，本句直译为“通过”。

“assume”，直译为“假定，认为”。

但这样翻译有两个问题：

1.不符合中文习惯。

2.没有准确表达出本句与上一句的自然衔接关系，也就是“提出问题”——“给出方案”。

考虑到上下文的衔接，在译文中采用了另一种表达方式，像下面这样。

译文：

我们只需优先处理写访问请求，即假设写请求比读请求更重要。

(二十三)：重入锁死

重入锁死是一种类似于**死锁**和**嵌套管程锁死**的情况。 重入锁死在**“锁”**和**“读/写锁”**一文中也有涉及。

如果线程重新进入 Lock，ReadWriteLock 或其他不可重入的同步器，则可能会发生重入锁死。 可重入是指已持有锁的线程可以重新获取该锁。 Java 的同步块是可重入的。 因此，以下代码可以正常工作：

```
public class Reentrant{


  public synchronized outer(){
    inner();
  }
```

```
  public synchronized inner(){
    //do something
  }
}
```

请注意，outer（）和 inner（）都被声明为 synchronized，这在 Java 中等效于 synchronized（this）块。 如果线程调用 external（），则从 external（）内部调用 inner()不会有问题，因为两个方法(或块)都在同一个管程对象(“ this”)上同步。 如果线程已经拥有管程对象上的锁，则它可以访问在同一管程对象上同步的所有块。 这称为重入。 线程可以重新进入已持有锁的任何代码块。

以下 Lock 实现是不可重入的：

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
```

```
  }


  public synchronized void unlock(){
     isLocked = false;
     notify();
  }
}
```

如果线程两次调用 lock（）而不在其间调用 unlock（），则第二次调用 lock（）将阻塞。 发生了重入锁死。

有两种做法可以避免重入锁死：

- 避免编写出重新进入锁的代码
- 使用可重入锁

这两种做法哪种最适合你的项目，取决于具体情况。 可重入锁的性能通常不如不可重入锁，并且很难实现，但对于你的情况来说可能算不上是问题。代码用重入锁是否更易于实现必须视情况而定。

翻译花絮：

原文：

Whether or not your code is easier to implement with or without lock reentrance must be determined case by case.

解析：

whether or not, with or without：表示是或不是，用或不用，语义有重叠，所

以翻译可以合并；

case by case：逐个情况；

译文：

代码用重入锁是否更易于实现必须视情况而定。

（二十四）：信号量（Semaphores，计数信号量，有界信号量，信号量用作锁）

文章目录

信号量是一种线程同步结构，可用于在线程之间发送信号以避免信号丢失，或像使用锁一样保护临界区。 Java 5 在 java.util.concurrent 包中附带了信号量实现，因此你不必实现自己的信号量。 尽管如此，了解其实现和使用背后的理论还是很有用的。

简单的信号量

如下是一个简单的信号量实现：

```
public class Semaphore {
   private boolean signal = false;

   public synchronized void take() {
      this.signal = true;
      this.notify();
   }

   public synchronized void release() throws InterruptedException{
      while(!this.signal) wait();
      this.signal = false;
   }

}
```

take（）方法发送一个信号，该信号存储在信号量内部。 release（）方法等待信号。 收到信号标志后，将再次将其清除，并退出 release（）方法。

使用这样的信号量可以避免信号丢失。 用 take（）代替 notify（），以及用 release（）代替 wait（）。 如果对 take（）的调用发生在对 release（）的调用之前，则调用 release（）的线程仍会知道 take（）被调用了，因为信号存储在内部的 signal 变量中。 使用 wait（）和 notify（）不会存储信号。

当使用信号量进行信号传递时，方法名 take（）和 release（）似乎有点奇怪。 该名称源于信号量作为锁来使用，如本文后面所述。 在这种情况下，该方法名更有意义。

使用信号量传递信号

如下是一个简化示例，两个线程使用信号量互相发信号：

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();
sender.start();
```

- 7

```
public class SendingThread {
  Semaphore semaphore = null;

  public SendingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
```

```
        while(true){
            //do something, then signal
            this.semaphore.take();

        }
    }
}
public class RecevingThread {
    Semaphore semaphore = null;

    public ReceivingThread(Semaphore semaphore){
        this.semaphore = semaphore;
    }

    public void run(){
        while(true){
            this.semaphore.release();
            //receive signal, then do something...
        }
    }
}
```

计数信号量

上一节中的 Semaphore 实现不计算 take（）方法发送信号的次数。 我们可以更改 Semaphore 来做到这一点。 这称为计数信号量。 如下是计数信号量的简单实现：

```
public class CountingSemaphore {
   private int signals = 0;

   public synchronized void take() {
      this.signals++;
      this.notify();
   }

   public synchronized void release() throws InterruptedException{
      while(this.signals == 0) wait();
      this.signals--;
   }

}
```

有界信号量

CoutingSemaphore 没有存储信号数量的上限。我们可以将信号量的实现更改为有上限的，如下所示：

```
public class BoundedSemaphore {
   private int signals = 0;
   private int bound     = 0;

   public BoundedSemaphore(int upperBound){
      this.bound = upperBound;
   }

   public synchronized void take() throws InterruptedException{
      while(this.signals == bound) wait();
      this.signals++;
      this.notify();
   }

   public synchronized void release() throws InterruptedException{
      while(this.signals == 0) wait();
      this.signals--;
      this.notify();
   }
}
```

请注意，如果信号数量等于上限，则 take () 方法现在会阻塞。 如果 BoundedSemaphore 已达到其信号上限，则调用 take () 的线程不再允许传递其信号，直到某个线程调用 release （）。

将信号量用作锁

可以将有界信号量用作锁。 为此，请将上限设置为 1，并调用 take（）和 release（）来保护临界区。 如下是一个例子：

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{
   //critical section
} finally {
   semaphore.release();
}
```

与使用信号相比，方法 take（）和 release（）现在由同一线程调用。 由于只允许一个线程使用信号量，因此所有调用 take () 的其他线程都将被阻塞，直到调用 release () 为止。由于总是先调用 take（），因此 release（）的调用永远不会阻塞。

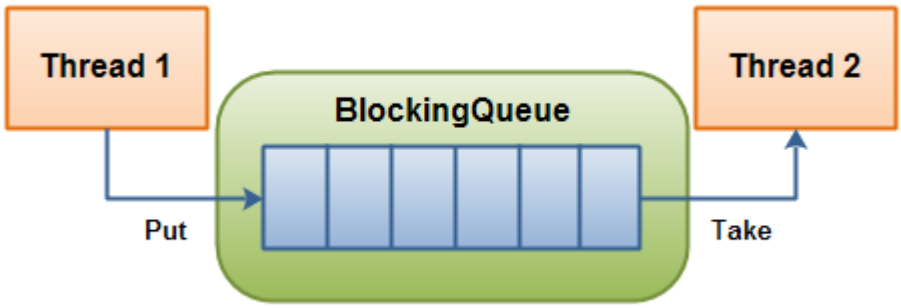
你还可以使用有界信号量来限制代码段中允许的线程数。 例如，在上面的示例中，如果将 BoundedSemaphore 的限制设置为 5，情况会怎么样呢？ 那样的话，一次允许 5 个线程进入临界区。 但是，你必须确保这 5 个线程的线程操作不会冲突，否则应用程序将出错。

从 finally 块内部调用 release（）方法以确保即使从临界区抛出异常也能被调用到。

(二十五)：阻塞队列

阻塞队列是这样一个队列，当尝试在队列为空时出队，或者尝试在队列已满时入队，它将阻塞。 尝试从空队列中出队的线程将被阻塞，直到其他线程插入一项到队列中为止。 尝试使一个项目进入满队列的线程将被阻塞，直到某个其他线程在队列中腾出空间为止，方法是使一个或多个项目出队或完全清除队列。

下面的示意图显示两个线程通过阻塞队列进行协作：



Java 5 在 java.util.concurrent 包中附带了阻塞队列实现。 可以在我的 java.util.concurrent.BlockingQueue 教程中了解该类。 即使 Java 5 附带了阻塞队列实现，了解它们实现背后的理论也会很有用。

阻塞队列实现

阻塞队列的实现看起来类似于有界信号量。 下面是阻塞队列的简单实现：

```
public class BlockingQueue {


  private List queue = new LinkedList();

  private int   limit = 10;


  public BlockingQueue(int limit){

    this.limit = limit;

  }




  public synchronized void enqueue(Object item)

  throws InterruptedException   {

    while(this.queue.size() == this.limit) {

      wait();

    }

    this.queue.add(item);

    if(this.queue.size() == 1) {

      notifyAll();

    }

  }
```

```
  public synchronized Object dequeue()

  throws InterruptedException{

     while(this.queue.size() == 0){

        wait();

     }

     if(this.queue.size() == this.limit){

        notifyAll();

     }


     return this.queue.remove(0);

  }


}
```

请注意，只有在队列大小等于大小界限（0 或上限），则才会从 enqueue（）和 dequeue（）调用 notifyAll（）。 如果在调用 enqueue（）或 dequeue（）时队列大小不等于大小界限，则可能没有线程在等待入队或出队。

(二十六)：线程池（Thread Pool)

当需要限制应用程序中同时运行的线程数时，线程池很有用。启动新线程会带来性能开销，并且每个线程还为其栈等分配了一些内存。

与其为每个并发执行的任务启动新线程，不如将任务传递给线程池。 一旦池中有任何空闲线程，就会将任务分配给其中一个并执行。 内部实现为将任务插入到阻塞队列中，线程池中的线程从该队列中取出。 当一个新任务插入到队列后，其中一个空闲线程将它从队列中出队并执行它。 线程池中的其余空闲线程将被阻塞，等待任务出队。

线程池通常用于多线程服务器中。 通过网络到达服务器的每个连接都被包装为一个任务，并传递给线程池。 线程池中的线程将同时处理连接请求。 后续的教程将详细介绍有关在 Java 中实现多线程服务器的信息。

Java 5 在 java.util.concurrent 包中带有内置的线程池，因此不必实现自己的线程池。 你可以在 java.util.concurrent.ExecutorService 的文章中阅读有关此内容的更多信息。 不管怎样，了解线程池的实现还是很有用的。

下面是一个简单的线程池实现。 请注意，此实现使用了我自己的 BlockingQueue 类，如 Blocking Queues 教程中所述。 在实际实现时，你可能会改用 Java 的某个内置阻塞队列。

```
public class ThreadPool {

    private BlockingQueue taskQueue = null;
    private List<PoolThread> threads = new ArrayList<PoolThread>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new BlockingQueue(maxNoOfTasks);
```

```
        for(int i=0; i<noOfThreads; i++){
            threads.add(new PoolThread(taskQueue));
        }
        for(PoolThread thread : threads){
            thread.start();
        }
    }

    public synchronized void    execute(Runnable task) throws
Exception{
        if(this.isStopped) throw
            new IllegalStateException("ThreadPool is stopped");

        this.taskQueue.enqueue(task);
    }

    public synchronized void stop(){
        this.isStopped = true;
        for(PoolThread thread : threads){
            thread.doStop();
        }
```

```
    }

}
public class PoolThread extends Thread {

    private BlockingQueue taskQueue = null;
    private boolean         isStopped = false;

    public PoolThread(BlockingQueue queue){
        taskQueue = queue;
    }

    public void run(){
        while(!isStopped()){
            try{
                Runnable runnable = (Runnable) taskQueue.dequeue();
                runnable.run();
            } catch(Exception e){
                //log or otherwise report exception,
                //but keep pool thread alive.
            }
        }
```

```
    }

    public synchronized void doStop(){
        isStopped = true;
        this.interrupt(); //break pool thread out of dequeue() call.
    }

    public synchronized boolean isStopped(){
        return isStopped;
    }
}
```

线程池实现由两部分组成。 ThreadPool 类是线程池的公共接口，而 PoolThread 类是实现执行任务的线程的类。

要执行任务，需调用 ThreadPool.execute（Runnable r）方法并以一个 Runnable 实现作为参数。 Runnable 在内部进入阻塞队列，等待被出队。

Runnable 将由空闲的 PoolThread 出队并执行。 可以在 PoolThread.run（）方法中看到这一点。 执行后，PoolThread 循环并尝试再次使任务出队，直到停止。

要停止 ThreadPool，需调用 ThreadPool.stop（）方法。stop 的调用被内部记录在 isStopped 成员变量中。 然后，在每个线程上调用 doStop（）来停止线程池中的每个线

程。 注意，如果在调用 stop（）之后调用 execute（），则 execute（）方法将抛出 IllegalStateException。

完成当前执行的任务后，线程将停止。注意 PoolThread.doStop()中调用了 this.interrupt（）。 这样可以确保在 taskQueue.dequeue（）内阻塞在 wait（）中的线程跳出 wait（），并在 dequeue（）方法中抛出 InterruptedException 并退出。 此异常将在 PoolThread.run（）方法中被捕获并上报，然后检查 isStopped 变量。 由于 isStopped 现在为 true，因此 PoolThread.run（）将退出并且线程死亡。

（二十七）：比较交换（CAS）

文章目录

比较交换是设计并发算法时使用的一种技术。 基本上，比较交换将期望值与变量的实际值进行比较，如果变量的实际值等于期望值，则将变量的值替换为新的值。 比较交换听起来可能有点复杂，但是一旦你理解了它，实际上就相当简单了，所以让我对这个话题做进一步的阐述。

哪些情况需要用到比较交换

在程序和并发算法中，一种常会出现的模式是“先检查后行动”模式。代码首先检查变量的值，然后根据该值进行操作，如此便出现了“先检查后行动”模式。 下面是一个简单的示例：

```
class MyLock {


    private boolean locked = false;


    public boolean lock() {
        if(!locked) {
            locked = true;
            return true;
        }
        return false;
    }
}
```

此代码在多线程应用程序中使用会有很多错误，但是现在请忽略。

如你所见，lock（）方法首先检查 locked 成员变量是否等于 false（检查），如果等于则将 locked 设为 true（然后行动）。

如果多个线程访问同一 MyLock 实例，则不能保证上述 lock（）函数可以正常工作。 如果线程 A 检查 locked 的值并发现它为 false，则线程 B 也可以在同一时间检查 locked 的值并发现它为 false。 或者，实际上，在线程 A 检查 locked 并看到它为 false 与线程 A 设置 locked 为 true 之间的任何时间，线程 B 都可以检查 locked。 因此，线程 A 和线程 B 都可能发现 locked 为 false，然后都将基于该信息进行操作。

为了在多线程应用程序中正常工作，“先检查后行动”操作必须是原子操作。 “原子”是指“检查”和“行动”动作均作为原子（不可分割）代码块执行。 任何开始执行该块的线程都将完成该块的执行，而不会受到其他线程的干扰。 没有其他线程可以同时执行原子块。

这是前面的代码示例，其中使用 synchronized 关键字将 lock（）方法转换为原子代码块：

```
class MyLock {


    private boolean locked = false;


    public synchronized boolean lock() {
        if(!locked) {
            locked = true;
            return true;
        }
        return false;
    }
}
```

现在，lock（）方法是同步的，因此在同一 MyLock 实例上，一次只能执行一个线程。 lock（）方法实际上是原子的。

原子 lock（）方法实际上是“比较交换”的一个示例。 lock（）方法将 locked 变量与期望值 false 进行比较，如果 locked 等于该期望值，则将变量的值变为 true。

比较交换作为原子操作

现代 CPU 内置了对原子比较交换操作的支持。 从 Java 5 中，可以通过 java.util.concurrent.atomic 包中的一些新的原子类访问 CPU 中的这些功能。

如下的示例展示了如何使用 AtomicBoolean 类实现前面所示的 lock（）方法：

```
public static class MyLock {

    private AtomicBoolean locked = new AtomicBoolean(false);


    public boolean lock() {

        return locked.compareAndSet(false, true);

    }


}
```

请注意，变量 locked 不再是 boolean 而是 AtomicBoolean。 该类有 compareAndSet（）函数，该函数会将 AtomicBoolean 实例的值与期望值进行比较，如果符合期望值，它将用新值替换原值。 在本例中，它将 locked 的值与 false 进行比较，如果为 false，则将 locked 的新值设置为 true。（译注：原文是将 AtomicBoolean 的新值设置为 true，存在错误，译文做了修正。）

如果替换了新值，则 compareAndSet（）方法返回 true，否则返回 false。

使用 Java 5 以上版本附带的比较交换功能而不是自己实现的优点是，Java 5 以上版本内置的比较交换功能可以利用应用程序所运行的 CPU 的底层比较交换功能。 这使你的比较交换代码更快。

（二十八）：同步器的结构

很多同步器（锁，信号量，阻塞队列等）虽然在功能上有所不同，但它们的内部设计通常没有太大不同。 换句话说，它们在内部由相同（或相似）的基本部分组成。 在设计同步器时，了解这些基本部分会很有帮助。 本文将重点讨论这些部分。

大多数（如果不是全部）同步器的目的是保护代码的某些区域（临界区）免受线程的并发访问。 为此，同步器中通常需要以下部分：

- 状态
- 访问条件
- 状态改变
- 通知策略
- 检查并设置方法
- 设置方法

并非所有同步器都具有这些部分，而那些有这些部分的同步器也未必与此处所述完全相同。不过，通常可以找到其中一个或多个部分。

状态

同步器的状态用于访问条件决定是否可以授予线程访问权限。 在 Lock 中，状态保存在一个 boolean 变量中，说明 Lock 是否被锁定。 在有界信号量中，内部状态保存在一个计数器（int）和一个上限（int）中，分别描述当前的“ takes”数量和“ takes”的最大数量。 在阻塞队列中，状态保留在队列中的元素列表以及最大队列大小（int）成员（如果有）中。

这是 Lock 和 BoundedSemaphore 的两个代码段。 状态代码以粗体标记。

```
public class Lock{

//state is kept here
private boolean isLocked = false;

public synchronized void lock()
throws InterruptedException{
while(isLocked){
wait();
}
isLocked = true;
}

...
}

public class BoundedSemaphore {
```

```
//state is kept here
private int signals = 0;
private int bound = 0;

public BoundedSemaphore(int upperBound){
this.bound = upperBound;
}

public synchronized void take() throws InterruptedException{
while(this.signals == bound) wait();
this.signal++;
this.notify();
}
...
}
```

访问条件

如果线程调用了“测试并设置状态”方法，访问条件决定了是否允许它设置状态。 访问条件通常基于同步器的状态，在 while 循环中检查，以防止虚假唤醒。 访问条件的检查结果是 true 或 false。

在 Lock 中，访问条件只是检查 isLocked 成员变量的值。 在有界信号量中，实际上有两种访问条件，具体取决于是要“获取”还是“释放”信号量。 如果线程尝试获取信号量，则检查 signals 变量是否超出上限。 如果线程尝试释放信号量，则检查 signals 变量是否为 0。

如下是 Lock 和 BoundedSemaphore 的两个代码段，访问条件用粗体标记。 注意始终在 while 循环中检查条件。

```
public class Lock{

private boolean isLocked = false;

public synchronized void lock()
throws InterruptedException{
//access condition
while(isLocked){
wait();
}
isLocked = true;
}

...
}

public class BoundedSemaphore {
private int signals = 0;
private int bound = 0;

public BoundedSemaphore(int upperBound){
this.bound = upperBound;
}
```

```
public synchronized void take() throws InterruptedException{
//access condition
while(this.signals == bound) wait();
this.signals++;
this.notify();
}

public synchronized void release() throws InterruptedException{
//access condition
while(this.signals == 0) wait();
this.signals–;
this.notify();
}
}
```

状态改变

一旦线程获得对临界区的访问权，它就必须更改同步器的状态，以（可能）阻止其他线程进入。 换句话说，状态需要反映一个事实，即有线程正在临界区内部执行。对于其他尝试获得访问权限的线程， 这将影响它们的访问条件。

在 Lock 中，状态更改是代码设置 isLocked = true。 在信号量中，是 signals–或者 signals++。

如下是两个代码片段，其中状态更改代码以粗体显示：

```
public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

//state change

isLocked = true;

}

public synchronized void unlock(){

//state change

isLocked = false;

notify();

}

}

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;
```

```
public BoundedSemaphore(int upperBound){
this.bound = upperBound;
}

public synchronized void take() throws InterruptedException{
while(this.signals == bound) wait();
//state change
this.signals++;
this.notify();
}

public synchronized void release() throws InterruptedException{
while(this.signals == 0) wait();
//state change
this.signals--;
this.notify();
}
}
```

通知策略

一旦线程更改了同步器的状态，有时可能需要将状态更改通知其他正在等待的线程。 也许此状态更改会使其他线程的访问条件变为 true。

通知策略通常分为三类。

1. 通知所有等待的线程。
2. 通知 N 个等待线程中随机的 1 个。
3. 通知 N 个等待线程中特定的 1 个。

通知所有正在等待的线程非常容易。 所有等待线程都在同一对象上调用 wait（）。 每当线程想要通知等待的线程，它将某个对象上调用 notifyAll（），该对象是等待的线程调用 wait()的对象。

通知一个随机的等待线程也很容易。 只需让通知线程在某个对象上调用 notify（），该对象是等待线程调用 wait()的对象。 调用 notify 不能保证将通知哪个等待线程。 因此，这里用术语“随机等待线程”来表示。

有时可能需要通知特定的线程而不是随机的等待线程。 例如，如果需要保证以特定的顺序通知正在等待的线程，以它们调用同步器的顺序，或者以某些优先顺序。 为了实现这一点，每个等待线程必须在自己的单独对象上调用 wait（）。 当通知线程想要通知特定的等待线程时，它将在该特定线程已调用 wait（）的对象上调用 notify（）。 在“饥饿与公平”一文中有这样一个例子。

下面是通知策略（通知 1 个随机等待线程，用粗体标记）的代码片段：

```
public class Lock{

private boolean isLocked = false;

public synchronized void lock()
throws InterruptedException{
```

```
while(isLocked){
//wait strategy - related to notification strategy
wait();
}
isLocked = true;
}

public synchronized void unlock(){
isLocked = false;
notify(); //notification strategy
}
}
```

检查并设置方法

同步器通常有两种类型的方法，其中第一种类型是检查并设置（另一种是设置）。 检查并设置是指调用此方法的线程**检查**同步器的内部状态是否满足访问条件。 如果满足条件，线程将**设置**同步器的内部状态以反映该线程已获得访问权限。

状态转换通常会导致其他尝试获取访问权限的线程的访问条件变为假，但可能并非总是如此。例如，在“读写锁”中，获得读取访问权限的线程将更新读写锁的状态以反映该状态，但是只要没有线程请求写入访问权限，其他请求读取访问权限的线程也将被授予访问权限。

必须以原子方式执行“检查并设置”操作，这意味着在“检查并设置”方法的检查和状态设置之间不允许执行其他线程。

“检查并设置”方法的程序流程通常类似于以下内容：

1. 必要时在检查前设置状态
2. 检查状态是否满足访问条件
3. 如果不满足访问条件，继续等待
4. 如果满足访问条件，设置状态，并在必要时通知等待线程

下面显示的 ReadWriteLock 类的 lockWrite（）方法是“检查并设置”方法的示例。 调用 lockWrite（）的线程首先在检查之前设置状态（writeRequests ++）。 然后，它在 canGrantWriteAccess（）方法中检查内部状态是否符合访问条件。 如果检查成功，则退出该方法之前，将再次设置内部状态。 请注意，此方法不会通知等待线程。

```
public class ReadWriteLock{
private Map<Thread, Integer> readingThreads =
new HashMap<Thread, Integer>();

private int writeAccesses    = 0;
private int writeRequests    = 0;
private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{
writeRequests++;
Thread callingThread = Thread.currentThread();
while(! canGrantWriteAccess(callingThread)){
wait();
```

}

writeRequests–;

writeAccesses++;

writingThread = callingThread;

}

}

下面显示的 BoundedSemaphore 类具有两个“检查并设置”方法：take ( ) 和 release()。 两种方法都检查并设置内部状态。

```
public class BoundedSemaphore {
   private int signals = 0;
   private int bound     = 0;


   public BoundedSemaphore(int upperBound){
      this.bound = upperBound;
   }



         public synchronized void take() throws InterruptedException{
         while(this.signals == bound) wait();
         this.signals++;
         this.notify();
```

```
    }

    public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
    }

}
```

设置方法

设置方法是同步器通常包含的第二种方法。 设置方法仅设置同步器的内部状态，而无需先对其检查。 设置方法的一个典型示例是 Lock 类的 unlock（）方法。 持有锁的线程可以总是直接解锁，而不必检查 Lock 是否已解锁。

设置方法的程序流通常遵循以下原则：

1. 设置内部状态
2. 通知等待线程

下面的示例是一个 unlock()方法：

```
public class Lock{
```

```
    private boolean isLocked = false;


        public synchronized void unlock(){
        isLocked = false;
        notify();
        }


}
```

(三十):终章:阿姆达尔定律(Amdahl‘s Law)

文章目录

阿姆达尔定律可用于推测计算量通过部分并行运行可以加速多少。 阿姆达尔定律以吉恩·阿姆达尔（Gene Amdahl）的名字命名，他在 1967 年提出了该定律。即使不知道阿姆达尔定律，大多数使用并行或并发系统工作的开发人员都对潜在的加速有着直觉的感觉。 无论如何，了解阿姆达尔定律可能仍然有用。

我将先以数学方式解释阿姆达尔定律，然后使用图表说明阿姆达尔定律。

阿姆达尔定律定义

可并行化的程序（或算法）可以分为两部分：

- 无法并行化的部分
- 可以并行化的部分

设想一个处理磁盘文件的程序。 该程序的一小部分可能会扫描目录并在内存内部创建文件列表。 之后，每个文件都传递到单独的线程进行处理。 扫描目录并创建文件列表的部分无法并行化，但处理文件可以并行化。

串行（非并行）执行程序所花费的总时间称为 T。时间 T 包括不可并行部分和可并行部分的时间。 不可并行化的部分称为 B。可并行化的部分称为 T-B。以下列表总结了这些定义：

- T = 串行执行总时间
- B = 不可并行化部分的总时间
- T - B = 可并行化部分的总时间（串行执行时，而非并行时）

由此可见：

T = B + (T-B)

乍一看，程序的可并行化部分在方程式中没有自己的符号，可能看起来有些奇怪。 但是，由于可以使用总时间 T 和 B（不可并行化的部分）表示方程的可并行化部分，因此实际上从概念上简化了方程，这意味着该形式包含的变量较少。

可以通过并行执行而加速的部分是可并行化部分 T-B。 可以加速多少取决于申请了多少个线程或 CPU 来执行。 线程或 CPU 的数量称为 N。因此，可并行化部分可以执行的最快速度是：

(T - B) / N

另一种写法是：

(1/N) * (T - B)

根据阿姆达尔定律，使用 N 个线程或 CPU 执行可并行化部分时，程序的总执行时间为：

T(N) = B + (T - B) / N

T（N）表示并行度为 N 的全部执行时间。因此，T 可以写为 T（1），并行度为 1 的总执行时间。使用 T（1）代替 T，阿姆达尔定律看起来像 像这样：

T(N) = B + (T(1) - B) / N

其含义是一样的。

一个推算示例

为了更好地理解阿姆达尔定律，我们来看一个推算示例。 执行程序的总时间设置为 1。程序的不可并行化部分为 40%，所以 1 的 40%等于 0.4。 因此，可并行化部分等于 1-0.4 = 0.6。

并行度为 2 (可并行化部分由 2 个线程或 CPU 执行, 因此 N 为 2) 的程序的执行时间将为：

T(2) = 0.4 + (1 - 0.4) / 2

= 0.4 + 0.6 / 2

= 0.4 + 0.3

= 0.7

若并行化系数使用 5 而不是 2，进行相同的计算将如下所示：

T(5) = 0.4 + (1 - 0.4) / 5

= 0.4 + 0.6 / 5

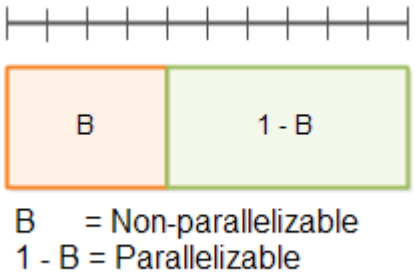
= 0.4 + 0.12

= 0.52

图解阿姆达尔定律

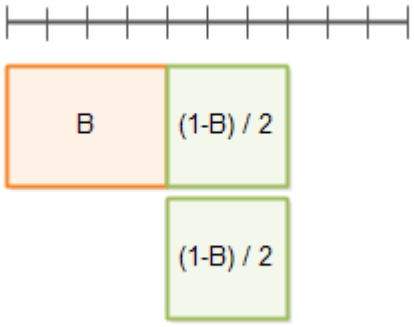
为了更好地理解阿姆达尔定律，我将尝试说明该定律是如何得出的。

首先，程序可分解为不可并行化的部分 B 和可并行化的部分 1-B，如下图所示：

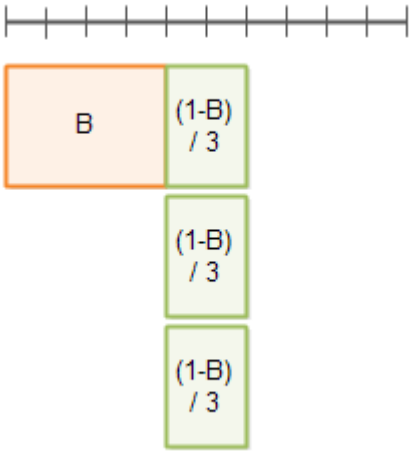


顶部带有分隔符的行是总时间 T（1）。

下面是并行度为 2 的执行时间：



下面是并行度为 3 的执行时间：



优化算法

根据阿姆达尔定律，可以很自然的推论出，可并行化部分可以通过堆硬件来加速执行。 即使用更多线程或 CPU。 但是，不可并行化的部分只能通过优化代码来加速执行。 因此，可以通过优化不可并行化的部分来提高程序的速度和并行性。 通常，你甚至可以通过将一些工作移到可并行化部分中（如果可行），来将算法更改为具有较小的不可并行化部分。

优化串行部分

如果要优化程序的串行部分，还可以使用阿姆达尔定律来计算优化后程序的执行时间。 如果将不可并行部分 B 以因数 O 来优化，则阿姆达尔定律表示为：

T(O,N) = B / O + (1 - B / O) / N

请记住，程序的不可并行化部分现在需要 B / O 时间，因此并行化部分需要 1-B / O 时间。

如果 B 为 0.4，O 为 2，N 为 5，则计算如下：

T(2,5) = 0.4 / 2 + (1 - 0.4 / 2) / 5
= 0.2 + (1 - 0.4 / 2) / 5
= 0.2 + (1 - 0.2) / 5
= 0.2 + 0.8 / 5
= 0.2 + 0.16
= 0.36

执行时间与加速

到目前为止，我们仅使用阿姆达尔定律来计算优化或并行化后程序或算法的执行时间。 我们还可以使用阿姆达尔定律计算加速比，也就是新算法或程序比旧版本快多少。

如果程序或算法的旧版本时间为 T，则加速比为

Speedup = T / T(O,N)

我们通常将 T 设置为 1，只是为了计算执行时间和加速时间是原来的几分之一。 等式如下所示：

Speedup = 1 / T(O,N)

如果我们用阿姆达尔定律计算公式替换 T（O，N），则会得到以下公式：

Speedup = 1 / (B / O + (1 - B / O) / N)

在 B = 0.4，O = 2 和 N = 5 的情况下，计算公式为：

Speedup = 1 / (0.4 / 2 + (1 - 0.4 / 2) / 5)

= 1 / (0.2 + (1 - 0.4 / 2) / 5)

= 1 / (0.2 + (1 - 0.2) / 5)

= 1 / (0.2 + 0.8 / 5)

= 1 / (0.2 + 0.16)

= 1 / 0.36

= 2.77777 ...

这就是说，如果以因数 2 来优化不可并行化（串行）部分，并以因数 5 并行执行可并行化部分，则该程序或算法优化后的新版本的运行速度最多比旧版本快 2.77777 倍 。

要测量，别仅是计算

尽管阿姆达尔定律使你能够计算算法并行化的理论速度，但不要过分依赖此类计算。 实际上，在优化或并行化算法时，也可能受许多其他因素影响。

内存，CPU 高速缓存，磁盘，网卡等（如果使用磁盘或网络）的速度也可能是限制因素。 如果新版本算法在并行化时，导致更多的 CPU 高速缓存未命中，你可能无法获得想要的使用 N 个 CPU 带来 xN 的加速。如果新版本算法最终使内存总线，磁盘，网卡或网络连接饱和，情况也是如此。

我的建议是使用阿姆达尔定律来了解在何处进行优化，但要使用度量方法来确定优化的实际加速。 请记住，有时，高度序列化的串行（单 CPU）算法可能会优于并行算法，这仅仅是因为串行算法没有协调开销（分解工作并重新聚合），而且因为单个 CPU 算法可能更好地符合底层硬件的工作方式（CPU 管道，CPU 缓存等）。

翻译后记

至此，《java 并发和多线程教程》已经全部翻译完结了。全教程共 30 篇，让我们再回顾一下吧：

并发系列专栏：

Java 并发和多线程教程 2020 版