Java并发性和多线程

极客学院出版

前言

Java 是最先支持多线程开发的语言之一， Java 多线程和并发也是 Java 学习的重点加难点。本教程根据作者多年 Java 开发经验总结而成，旨在帮助读者明白并发的原理。

适用人群

本教程是 Java 初级工程师的进阶教程。

学习前提

学习本教程前，你需要对操作系统和 Java 语言有一定的了解。

鸣谢：并发编程网 – ifeve.com

目录

1

Java 并发性和多线程介绍

在过去单 CPU 时代，单任务在一个时间点只能执行单一程序。之后发展到多任务阶段，计算机能在同一时间点并行执行多任务或多进程。虽然并不是真正意义上的“同一时间点”，而是多个任务或进程共享一个 CPU，并交由操作系统来完成多任务间对 CPU 的运行切换，以使得每个任务都有机会获得一定的时间片运行。

随着多任务对软件开发者带来的新挑战，程序不在能假设独占所有的 CPU 时间、所有的内存和其他计算机资源。一个好的程序榜样是在其不再使用这些资源时对其进行释放，以使得其他程序能有机会使用这些资源。

再后来发展到多线程技术，使得在一个程序内部能拥有多个线程并行执行。一个线程的执行可以被认为是一个 CPU 在执行该程序。当一个程序运行在多线程下，就好像有多个 CPU 在同时执行该程序。

多线程比多任务更加有挑战。多线程是在同一个程序内部并行执行，因此会对相同的内存空间进行并发读写操作。这可能是在单线程程序中从来不会遇到的问题。其中的一些错误也未必会在单 CPU 机器上出现，因为两个线程从来不会得到真正的并行执行。然而，更现代的计算机伴随着多核 CPU 的出现，也就意味着不同的线程能被不同的 CPU 核得到真正意义的并行执行。

如果一个线程在读一个内存时，另一个线程正向该内存进行写操作，那进行读操作的那个线程将获得什么结果呢？是写操作之前旧的值？还是写操作成功之后的新值？或是一半新一半旧的值？或者，如果是两个线程同时写同一个内存，在操作完成后将会是什么结果呢？是第一个线程写入的值？还是第二个线程写入的值？还是两个线程写入的一个混合值？因此如没有合适的预防措施，任何结果都是可能的。而且这种行为的发生甚至不能预测，所以结果也是不确定性的。

Java 的多线程和并发性

Java 是最先支持多线程的开发的语言之一，Java 从一开始就支持了多线程能力，因此 Java 开发者能常遇到上面描述的问题场景。这也是我想为 Java 并发技术而写这篇系列的原因。作为对自己的笔记，和对其他 Java 开发的追随者都可获益的。

该系列主要关注 Java 多线程，但有些在多线程中出现的问题会和多任务以及分布式系统中出现的存在类似，因此该系列会将多任务和分布式系统方面作为参考，所以叫法上称为“并发性”，而不是“多线程”。

2

多线程的优点

尽管面临很多挑战，多线程有一些优点使得它一直被使用。这些优点是：

- 资源利用率更好
- 程序设计在某些情况下更简单
- 程序响应更快

资源利用率更好

想象一下，一个应用程序需要从本地文件系统中读取和处理文件的情景。比方说，从磁盘读取一个文件需要 5 秒，处理一个文件需要 2 秒。处理两个文件则需要：

```
5秒读取文件A
2秒处理文件A
5秒读取文件B
## 2秒处理文件B总共需要14秒
```

从磁盘中读取文件的时候，大部分的 CPU 时间用于等待磁盘去读取数据。在这段时间里，CPU 非常的空闲。它可以做一些别的事情。通过改变操作的顺序，就能够更好的使用 CPU 资源。看下面的顺序：

```
5秒读取文件A
5秒读取文件B + 2秒处理文件A
## 2秒处理文件B总共需要12秒
```

CPU 等待第一个文件被读取完。然后开始读取第二个文件。当第二文件在被读取的时候，CPU 会去处理第一个文件。记住，在等待磁盘读取文件的时候，CPU大 部分时间是空闲的。

总的说来，CPU 能够在等待 IO 的时候做一些其他的事情。这个不一定就是磁盘 IO。它也可以是网络的 IO，或者用户输入。通常情况下，网络和磁盘的 IO 比 CPU 和内存的 IO 慢的多。

程序设计更简单

在单线程应用程序中，如果你想编写程序手动处理上面所提到的读取和处理的顺序，你必须记录每个文件读取和处理的状态。相反，你可以启动两个线程，每个线程处理一个文件的读取和操作。线程会在等待磁盘读取文件的过程中被阻塞。在等待的时候，其他的线程能够使用 CPU 去处理已经读取完的文件。其结果就是，磁盘总是在繁忙地读取不同的文件到内存中。这会带来磁盘和 CPU 利用率的提升。而且每个线程只需要记录一个文件，因此这种方式也很容易编程实现。

程序响应更快

将一个单线程应用程序变成多线程应用程序的另一个常见的目的是实现一个响应更快的应用程序。设想一个服务器应用，它在某一个端口监听进来的请求。当一个请求到来时，它去处理这个请求，然后再返回去监听。

服务器的流程如下所述：

```
while(server is active){
   listen for request
   process request
}
```

如果一个请求需要占用大量的时间来处理，在这段时间内新的客户端就无法发送请求给服务端。只有服务器在监听的时候，请求才能被接收。另一种设计是，监听线程把请求传递给工作者线程(worker thread)，然后立刻返回去监听。而工作者线程则能够处理这个请求并发送一个回复给客户端。这种设计如下所述：

```
while(server is active){
   listen for request
   hand request to worker thread
}
```

这种方式，服务端线程迅速地返回去监听。因此，更多的客户端能够发送请求给服务端。这个服务也变得响应更快。

桌面应用也是同样如此。如果你点击一个按钮开始运行一个耗时的任务，这个线程既要执行任务又要更新窗口和按钮，那么在任务执行的过程中，这个应用程序看起来好像没有反应一样。相反，任务可以传递给工作者线程（word thread)。当工作者线程在繁忙地处理任务的时候，窗口线程可以自由地响应其他用户的请求。当工作者线程完成任务的时候，它发送信号给窗口线程。窗口线程便可以更新应用程序窗口，并显示任务的结果。对用户而言，这种具有工作者线程设计的程序显得响应速度更快。

3

多线程的代价

从一个单线程的应用到一个多线程的应用并不仅仅带来好处，它也会有一些代价。不要仅仅为了使用多线程而使用多线程。而应该明确在使用多线程时能多来的好处比所付出的代价大的时候，才使用多线程。如果存在疑问，应该尝试测量一下应用程序的性能和响应能力，而不只是猜测。

设计更复杂

虽然有一些多线程应用程序比单线程的应用程序要简单，但其他的一般都更复杂。在多线程访问共享数据的时候，这部分代码需要特别的注意。线程之间的交互往往非常复杂。不正确的线程同步产生的错误非常难以被发现，并且重现以修复。

上下文切换的开销

当 CPU 从执行一个线程切换到执行另外一个线程的时候，它需要先存储当前线程的本地的数据，程序指针等，然后载入另一个线程的本地数据，程序指针等，最后才开始执行。这种切换称为“上下文切换”（“context switch”）。CPU 会在一个上下文中执行一个线程，然后切换到另外一个上下文中执行另外一个线程。

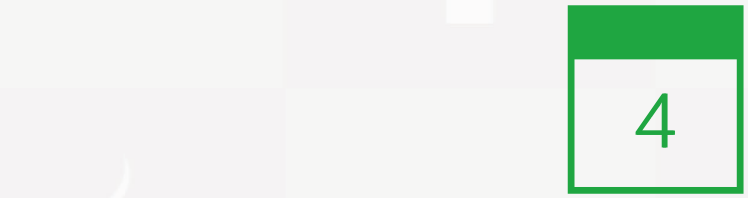
上下文切换并不廉价。如果没有必要，应该减少上下文切换的发生。

你可以通过维基百科阅读更多的关于上下文切换相关的内容：

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

增加资源消耗

线程在运行的时候需要从计算机里面得到一些资源。除了CPU，线程还需要一些内存来维持它本地的堆栈。它也需要占用操作系统中一些资源来管理线程。我们可以尝试编写一个程序，让它创建 100 个线程，这些线程什么事情都不做，只是在等待，然后看看这个程序在运行的时候占用了多少内存。



4

并发编程模型

并发系统可以采用多种并发编程模型来实现。并发模型指定了系统中的线程如何通过协作来完成分配给它们的作业。不同的并发模型采用不同的方式拆分作业，同时线程间的协作和交互方式也不相同。这篇并发模型教程将会较深入地介绍目前（2015 年，本文撰写时间）比较流行的几种并发模型。

并发模型与分布式系统之间的相似性

本文所描述的并发模型类似于分布式系统中使用的很多体系结构。在并发系统中线程之间可以相互通信。在分布式系统中进程之间也可以相互通信（进程有可能在不同的机器中）。线程和进程之间具有很多相似的特性。这也就是为什么很多并发模型通常类似于各种分布式系统架构。

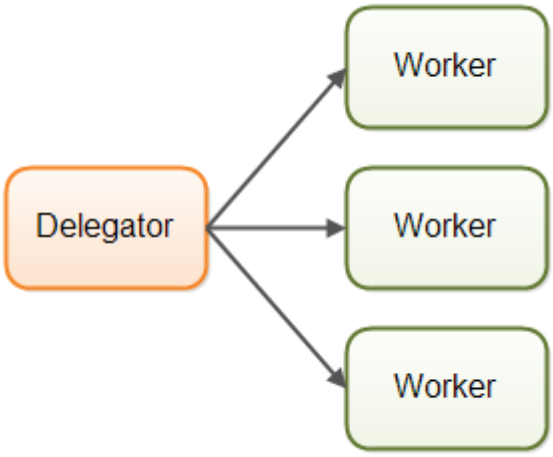
当然，分布式系统在处理网络失效、远程主机或进程宕掉等方面也面临着额外的挑战。但是运行在巨型服务器上的并发系统也可能遇到类似的问题，比如一块 CPU 失效、一块网卡失效或一个磁盘损坏等情况。虽然出现失效的概率可能很低，但是在理论上仍然有可能发生。

由于并发模型类似于分布式系统架构，因此它们通常可以互相借鉴思想。例如，为工作者们（线程）分配作业的模型一般与分布式系统中的负载均衡系统比较相似。同样，它们在日志记录、失效转移、幂等性等错误处理技术上也具有相似性。

【注：幂等性，一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同】

并行工作者

第一种并发模型就是我所说的并行工作者模型。传入的作业会被分配到不同的工作者上。下图展示了并行工作者模型：



在并行工作者模型中，委派者（Delegator）将传入的作业分配给不同的工作者。每个工作者完成整个任务。工作者们并行运作在不同的线程上，甚至可能在不同的 CPU 上。

如果在某个汽车厂里实现了并行工作者模型，每台车都会由一个工人来生产。工人们将拿到汽车的生产规格，并且从头到尾负责所有工作。

在 Java 应用系统中，并行工作者模型是最常见的并发模型（即使正在转变）。java.util.concurrent包中的许多并发实用工具都是设计用于这个模型的。你也可以在 Java 企业级（J2EE）应用服务器的设计中看到这个模型的踪迹。

并行工作者模型的优点

并行工作者模式的优点是，它很容易理解。你只需添加更多的工作者来提高系统的并行度。

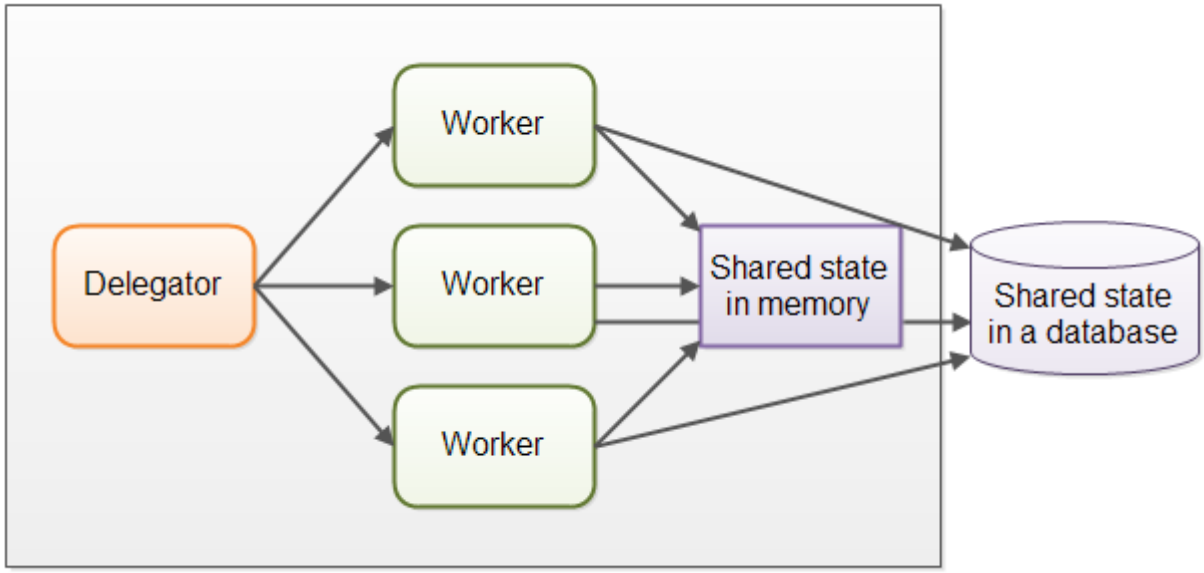
例如，如果你正在做一个网络爬虫，可以试试使用不同数量的工作者抓取到一定数量的页面，然后看看多少数量的工作者消耗的时间最短（意味着性能最高）。由于网络爬虫是一个 IO 密集型工作，最终结果很有可能是你电脑中的每个 CPU 或核心分配了几个线程。每个 CPU 若只分配一个线程可能有点少，因为在等待数据下载的过程中 CPU 将会空闲大量时间。

并行工作者模型的缺点

并行工作者模型虽然看起来简单，却隐藏着一些缺点。接下来的章节中我会分析一些最明显的弱点。

共享状态可能会很复杂

在实际应用中，并行工作者模型可能比前面所描述的情况要复杂得多。共享的工作者经常需要访问一些共享数据，无论是内存中的或者共享的数据库中的。下图展示了并行工作者模型是如何变得复杂的：



有些共享状态是在像作业队列这样的通信机制下。但也有一些共享状态是业务数据，数据缓存，数据库连接池等。

一旦共享状态潜入到并行工作者模型中，将会使情况变得复杂起来。线程需要以某种方式存取共享数据，以确保某个线程的修改能够对其他线程可见（数据修改需要同步到主存中，不仅仅将数据保存在执行这个线程的CPU的缓存中）。线程需要避免竞态，死锁以及很多其他共享状态的并发性问题。

此外，在等待访问共享数据结构时，线程之间的互相等待将会丢失部分并行性。许多并发数据结构是阻塞的，意味着在任何一个时间只有一个或者很少的线程能够访问。这样会导致在这些共享数据结构上出现竞争状态。在执行需要访问共享数据结构部分的代码时，高竞争基本上会导致执行时出现一定程度的串行化。

现在的非阻塞并发算法也许可以降低竞争并提升性能，但是非阻塞算法的实现比较困难。

可持久化的数据结构是另一种选择。在修改的时候，可持久化的数据结构总是保护它的前一个版本不受影响。因此，如果多个线程指向同一个可持久化的数据结构，并且其中一个线程进行了修改，进行修改的线程会获得一个指向新结构的引用。所有其他线程保持对旧结构的引用，旧结构没有被修改并且因此保证一致性。Scala 编程包含几个持久化数据结构。

【注：这里的可持久化数据结构不是指持久化存储，而是一种数据结构，比如 Java 中的 String 类，以及 Copy OnWriteArrayList 类，具体可参考】

虽然可持久化的数据结构在解决共享数据结构的并发修改时显得很优雅，但是可持久化的数据结构的表现往往不尽人意。

比如说，一个可持久化的链表需要在头部插入一个新的节点，并且返回指向这个新加入的节点的一个引用（这个节点指向了链表的剩余部分）。所有其他现场仍然保留了这个链表之前的第一个节点，对于这些线程来说链表仍然是为改变的。它们无法看到新加入的元素。

这种可持久化的列表采用链表来实现。不幸的是链表在现代硬件上表现的不太好。链表中得每个元素都是一个独立的对象，这些对象可以遍布在整个计算机内存中。现代 CPU 能够更快的进行顺序访问，所以你可以在现代的硬件上用数组实现的列表，以获得更高的性能。数组可以顺序的保存数据。CPU 缓存能够一次加载数组的一大块进行缓存，一旦加载完成 CPU 就可以直接访问缓存中的数据。这对于元素散落在 RAM 中的链表来说，不太可能做得到。

无状态的工作者

共享状态能够被系统中得其他线程修改。所以工作者在每次需要的时候必须重读状态，以确保每次都能访问到最新的副本，不管共享状态是保存在内存中的还是在外部数据库中。工作者无法在内部保存这个状态（但是每次需要的时候可以重读）称为无状态的。

每次都重读需要的数据，将会导致速度变慢，特别是状态保存在外部数据库中的时候。

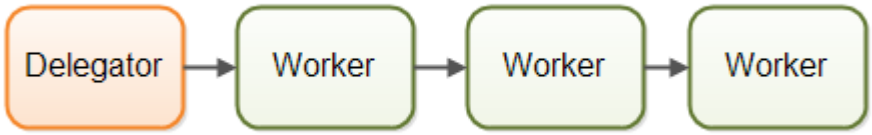
任务顺序是不确定的

并行工作者模式的另一个缺点是，作业执行顺序是不确定的。无法保证哪个作业最先或者最后被执行。作业 A 可能在作业 B 之前就被分配工作者了，但是作业 B 反而有可能在作业A之前执行。

并行工作者模式的这种非确定性的特性，使得很难在任何特定的时间点推断系统的状态。这也使得它也更难（如果不是不可能的话）保证一个作业在其他作业之前被执行。

流水线模式

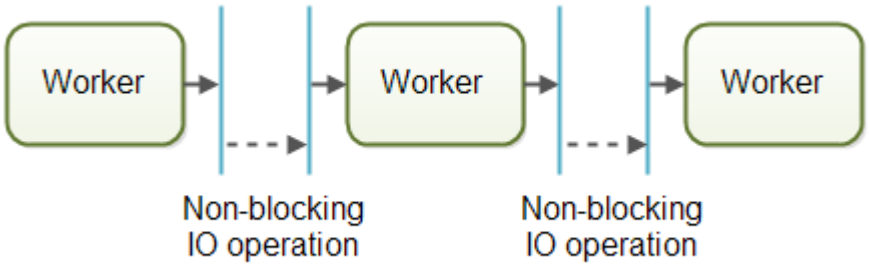
第二种并发模型我们称之为流水线并发模型。我之所以选用这个名字，只是为了配合“并行工作者”的隐喻。其他开发者可能会根据平台或社区选择其他称呼（比如说反应器系统，或事件驱动系统）。下图表示一个流水线并发模型：



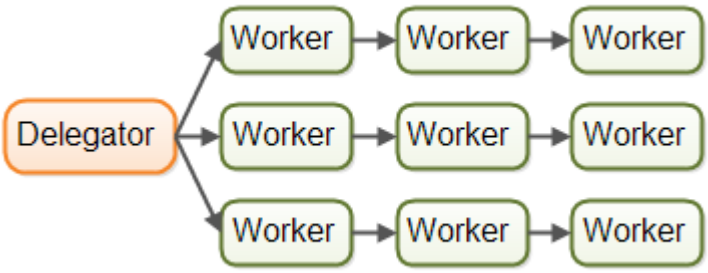
类似于工厂中生产线上的工人们那样组织工作者。每个工作者只负责作业中的部分工作。当完成了自己的这部分工作时工作者会将作业转发给下一个工作者。每个工作者在自己的线程中运行，并且不会和其他工作者共享状态。有时也被成为无共享并行模型。

通常使用非阻塞的 IO 来设计使用流水线并发模型的系统。非阻塞 IO 意味着，一旦某个工作者开始一个 IO 操作的时候（比如读取文件或从网络连接中读取数据），这个工作者不会一直等待 IO 操作的结束。IO 操作速度很慢，所以等待 IO 操作结束很浪费 CPU 时间。此时 CPU 可以做一些其他事情。当 IO 操作完成的时候，IO 操作的结果（比如读出的数据或者数据写完的状态）被传递给下一个工作者。

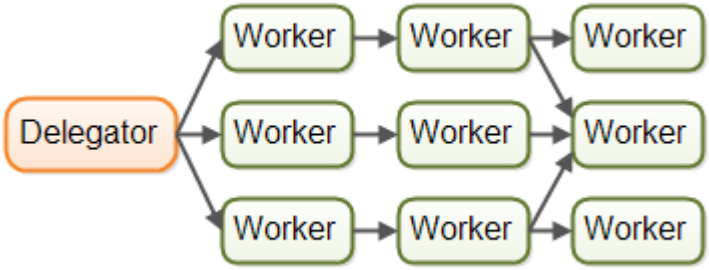
有了非阻塞 IO，就可以使用 IO 操作确定工作者之间的边界。工作者会尽可能多运行直到遇到并启动一个 IO 操作。然后交出作业的控制权。当 IO 操作完成的时候，在流水线上的下一个工作者继续进行操作，直到它也遇到并启动一个 IO 操作。



在实际应用中，作业有可能不会沿着单一流水线进行。由于大多数系统可以执行多个作业，作业从一个工作者流向另一个工作者取决于作业需要做的工作。在实际中可能会有多个不同的虚拟流水线同时运行。这是现实当中作业在流水线系统中可能的移动情况：



作业甚至也有可能被转发到超过一个工作者上并发处理。比如说，作业有可能被同时转发到作业执行器和作业日志器。下图说明了三条流水线是如何通过将作业转发给同一个工作者（中间流水线的最后一个工作者）来完成作业:



流水线有时候比这个情况更加复杂。

反应器，事件驱动系统

采用流水线并发模型的系统有时候也称为反应器系统或事件驱动系统。系统内的工作者对系统内出现的事件做出反应，这些事件也有可能来自于外部世界或者发自其他工作者。事件可以是传入的 HTTP 请求，也可以是某个文件成功加载到内存中等。在写这篇文章的时候，已经有很多有趣的反应器/事件驱动平台可以使用了，并且不久的将来会有更多。比较流行的似乎是这几个:

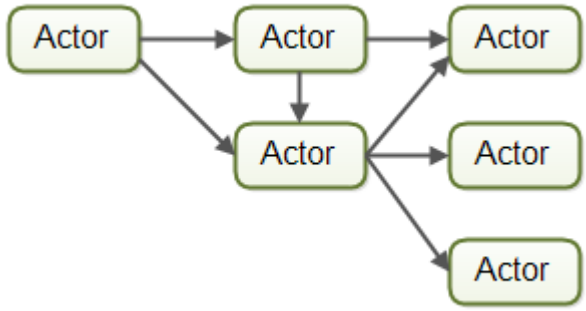
- Vert.x
- AKKa
- Node.JS(JavaScript)

我个人觉得 Vert.x 是相当有趣的（特别是对于我这样使用 Java/JVM 的人来说）

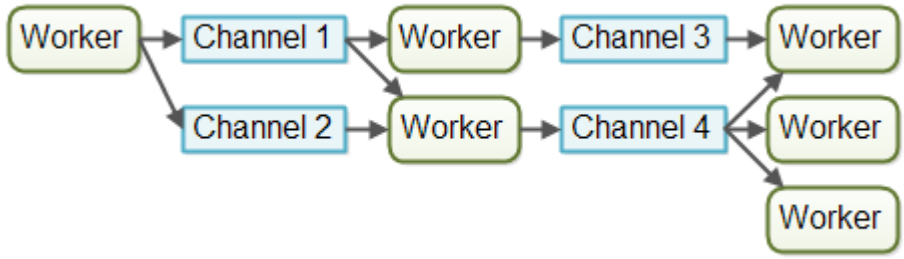
Actors 和 Channels

Actors 和 channels 是两种比较类似的流水线（或反应器/事件驱动）模型。

在 Actor 模型中每个工作者被称为 actor。Actor 之间可以直接异步地发送和处理消息。Actor 可以被用来实现一个或多个像前文描述的那样的作业处理流水线。下图给出了 Actor 模型：



而在 Channel 模型中，工作者之间不直接进行通信。相反，它们在不同的通道中发布自己的消息（事件）。其他工作者们可以在这些通道上监听消息，发送者无需知道谁在监听。下图给出了 Channel 模型：



在写这篇文章的时候，channel 模型对于我来说似乎更加灵活。一个工作者无需知道谁在后面的流水线上处理作业。只需知道作业（或消息等）需要转发给哪个通道。通道上的监听者可以随意订阅或者取消订阅，并不会影响向这个通道发送消息的工作者。这使得工作者之间具有松散的耦合。

流水线模型的优点

相比并行工作者模型，流水线并发模型具有几个优点，在接下来的章节中我会介绍几个最大的优点。

无需共享的状态

工作者之间无需共享状态，意味着实现的时候无需考虑所有因并发访问共享对象而产生的并发性问题。这使得在实现工作者的时候变得非常容易。在实现工作者的时候就好像是单个线程在处理工作–基本上是一个单线程的实现。

有状态的工作者

当工作者知道了没有其他线程可以修改它们的数据，工作者可以变成有状态的。对于有状态，我是指，它们可以在内存中保存它们需要操作的数据，只需在最后将更改写回到外部存储系统。因此，有状态的工作者通常比无状态的工作者具有更高的性能。

较好的硬件整合（Hardware Conformity）

单线程代码在整合底层硬件的时候往往具有更好的优势。首先，当能确定代码只在单线程模式下执行的时候，通常能够创建更优化的数据结构和算法。

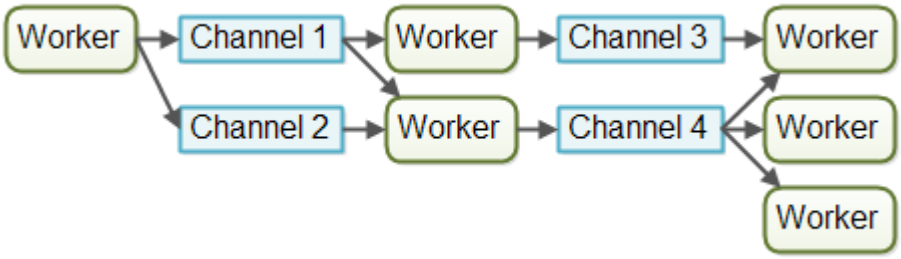
其次，像前文描述的那样，单线程有状态的工作者能够在内存中缓存数据。在内存中缓存数据的同时，也意味着数据很有可能也缓存在执行这个线程的 CPU 的缓存中。这使得访问缓存的数据变得更快。

我说的硬件整合是指，以某种方式编写的代码，使得能够自然地受益于底层硬件的工作原理。有些开发者称之为 mechanical sympathy。我更倾向于硬件整合这个术语，因为计算机只有很少的机械部件，并且能够隐喻“更好的匹配（match better）”，相比“同情（sympathy）”这个词在上下文中的意思，我觉得“conform”这个词表达的非常好。当然了，这里有点吹毛求疵了，用自己喜欢的术语就行。

合理的作业顺序

基于流水线并发模型实现的并发系统，在某种程度上是有可能保证作业的顺序的。作业的有序性使得它更容易地推出系统在某个特定时间点的状态。更进一步，你可以将所有到达的作业写入到日志中去。一旦这个系统的某一

部分挂掉了，该日志就可以用来重头开始重建系统当时的状态。按照特定的顺序将作业写入日志，并按这个顺序作为有保障的作业顺序。下图展示了一种可能的设计：



实现一个有保障的作业顺序是不容易的，但往往是可行的。如果可以，它将大大简化一些任务，例如备份、数据恢复、数据复制等，这些都可以通过日志文件来完成。

流水线模型的缺点

流水线并发模型最大的缺点是作业的执行往往分布到多个工作者上，并因此分布到项目中的多个类上。这样导致在追踪某个作业到底被什么代码执行时变得困难。

同样，这也加大了代码编写的难度。有时会将工作者的代码写成回调处理的形式。若在代码中嵌入过多的回调处理，往往会出现所谓的回调地狱（callback hell）现象。所谓回调地狱，就是意味着在追踪代码在回调过程中到底做了什么，以及确保每个回调只访问它需要的数据的时候，变得非常困难

使用并行工作者模型可以简化这个问题。你可以打开工作者的代码，从头到尾优美的阅读被执行的代码。当然并行工作者模式的代码也可能同样分布在不同的类中，但往往也能够很容易的从代码中分析执行的顺序。

函数式并行（Functional Parallelism）

第三种并发模型是函数式并行模型，这是也最近（2015）讨论的比较多的一种模型。函数式并行的基本思想是采用函数调用实现程序。函数可以看作是”代理人（agents）“或者”actor“，函数之间可以像流水线模型（AKA 反应器或者事件驱动系统）那样互相发送消息。某个函数调用另一个函数，这个过程类似于消息发送。

函数都是通过拷贝来传递参数的，所以除了接收函数外没有实体可以操作数据。这对于避免共享数据的竞态来说是很有必要的。同样也使得函数的执行类似于原子操作。每个函数调用的执行独立于任何其他函数的调用。

一旦每个函数调用都可以独立的执行，它们就可以分散在不同的 CPU 上执行了。这也就意味着能够在多处理器上并行的执行使用函数式实现的算法。

Java7 中的 java.util.concurrent 包里包含的 ForkAndJoinPool 能够帮助我们实现类似于函数式并行的一些东西。而 Java8 中并行 streams 能够用来帮助我们并行的迭代大型集合。记住有些开发者对 ForkAndJoinPool 进行了批判（你可以在我的 ForkAndJoinPool 教程里面看到批评的链接）。

函数式并行里面最难的是确定需要并行的那个函数调用。跨 CPU 协调函数调用需要一定的开销。某个函数完成的工作单元需要达到某个大小以弥补这个开销。如果函数调用作用非常小，将它并行化可能比单线程、单 CPU 执行还慢。

我个人认为（可能不太正确），你可以使用反应器或者事件驱动模型实现一个算法，像函数式并行那样的方法实现工作的分解。使用事件驱动模型可以更精确的控制如何实现并行化（我的观点）。

此外，将任务拆分给多个 CPU 时协调造成的开销，仅仅在该任务是程序当前执行的唯一任务时才有意义。但是，如果当前系统正在执行多个其他的任务时（比如 web 服务器，数据库服务器或者很多其他类似的系统），将单个任务进行并行化是没有意义的。不管怎样计算机中的其他 CPU 们都在忙于处理其他任务，没有理由用一个慢的、函数式并行的任务去扰乱它们。使用流水线（反应器）并发模型可能会更好一点，因为它开销更小（在单线程模式下顺序执行）同时能更好的与底层硬件整合。

使用那种并发模型最好?

所以，用哪种并发模型更好呢?

通常情况下，这个答案取决于你的系统打算做什么。如果你的作业本身就是并行的、独立的并且没有必要共享状态，你可能会使用并行工作者模型去实现你的系统。虽然许多作业都不是自然并行和独立的。对于这种类型的系统，我相信使用流水线并发模型能够更好的发挥它的优势，而且比并行工作者模型更有优势。

你甚至不用亲自编写所有流水线模型的基础结构。像 Vert.x 这种现代化的平台已经为你实现了很多。我也会去为探索如何设计我的下一个项目，使它运行在像 Vert.x 这样的优秀平台上。我感觉 Java EE 已经没有任何优势了。

5

如何创建并运行 java 线程

Java 线程类也是一个 object 类，它的实例都继承自 java.lang.Thread 或其子类。 可以用如下方式用 java 中创建一个线程：

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

执行该线程可以调用该线程的 start()方法:

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

在上面的例子中，我们并没有为线程编写运行代码，因此调用该方法后线程就终止了。

编写线程运行时执行的代码有两种方式：一种是创建 Thread 子类的一个实例并重写 run 方法，第二种是创建类的时候实现 Runnable 接口。接下来我们会具体讲解这两种方法：

创建 Thread 的子类

创建 Thread 子类的一个实例并重写 run 方法，run 方法会在调用 start()方法之后被执行。例子如下：

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

可以用如下方式创建并运行上述 Thread 子类

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

一旦线程启动后 start 方法就会立即返回，而不会等待到 run 方法执行完毕才返回。就好像 run 方法是在另外一个 cpu 上执行一样。当 run 方法执行后，将会打印出字符串 MyThread running。

你也可以如下创建一个 Thread 的匿名子类：

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

当新的线程的 run 方法执行以后，计算机将会打印出字符串”Thread Running”。

实现 Runnable 接口

第二种编写线程执行代码的方式是新建一个实现了 java.lang.Runnable 接口的类的实例，实例中的方法可以被线程调用。下面给出例子：

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

为了使线程能够执行 run()方法，需要在 Thread 类的构造函数中传入 MyRunnable 的实例对象。示例如下：

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

当线程运行时，它将会调用实现了 Runnable 接口的 run 方法。上例中将会打印出”MyRunnable running”。

同样，也可以创建一个实现了 Runnable 接口的匿名类，如下所示：

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

创建子类还是实现 Runnable 接口?

对于这两种方式哪种好并没有一个确定的答案，它们都能满足要求。就我个人意见，我更倾向于实现 Runnable 接口这种方法。因为线程池可以有效的管理实现了 Runnable 接口的线程，如果线程池满了，新的线程就会排队等候执行，直到线程池空闲出来为止。而如果线程是通过实现 Thread 子类实现的，这将会复杂一些。

有时我们要同时融合实现 Runnable 接口和 Thread 子类两种方式。例如，实现了 Thread 子类的实例可以执行多个实现了 Runnable 接口的线程。一个典型的应用就是线程池。

常见错误：调用 run()方法而非 start()方法

创建并运行一个线程所犯的常见错误是调用线程的 run()方法而非 start()方法，如下所示：

```
Thread newThread = new Thread(MyRunnable());
newThread.run();  //should be start();
```

起初你并不会感觉到有什么不妥，因为 run()方法的确如你所愿的被调用了。但是，事实上,run()方法并非是由刚创建的新线程所执行的，而是被创建新线程的当前线程所执行了。也就是被执行上面两行代码的线程所执行的。想要让创建的新线程执行 run()方法，必须调用新线程的 start 方法。

线程名

当创建一个线程的时候，可以给线程起一个名字。它有助于我们区分不同的线程。例如：如果有多个线程写入 System.out，我们就能够通过线程名容易的找出是哪个线程正在输出。例子如下：

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");
thread.start();
System.out.println(thread.getName());
```

需要注意的是，因为 MyRunnable 并非 Thread 的子类，所以 MyRunnable 类并没有 getName()方法。可以通过以下方式得到当前线程的引用：

```
Thread.currentThread();
```

因此，通过如下代码可以得到当前线程的名字：

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

线程代码举例：

这里是一个小小的例子。首先输出执行main()方法线程名字。这个线程 JVM 分配的。然后开启 10 个线程，命名为 1~10。每个线程输出自己的名字后就退出。

```
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();
     }
  }
}
```

需要注意的是，尽管启动线程的顺序是有序的，但是执行的顺序并非是有序的。也就是说，1 号线程并不一定是第一个将自己名字输出到控制台的线程。这是因为线程是并行执行而非顺序的。Jvm 和操作系统一起决定了线程的执行顺序，他和线程的启动顺序并非一定是一致的。

6

竞态条件与临界区

在同一程序中运行多个线程本身不会导致问题，问题在于多个线程访问了相同的资源。如，同一内存区（变量，数组，或对象）、系统（数据库，web services 等）或文件。实际上，这些问题只有在一或多个线程向这些资源做了写操作时才有可能发生，只要资源没有发生变化,多个线程读取相同的资源就是安全的。

多线程同时执行下面的代码可能会出错：

```
public class Counter {
   protected long count = 0;
   public void add(long value){
      this.count = this.count + value;
   }
}
```

想象下线程 A 和 B 同时执行同一个 Counter 对象的 add()方法，我们无法知道操作系统何时会在两个线程之间切换。JVM 并不是将这段代码视为单条指令来执行的，而是按照下面的顺序：

```
从内存获取 this.count 的值放到寄存器
将寄存器中的值增加 value
将寄存器中的值写回内存
```

观察线程 A 和 B 交错执行会发生什么：

```
this.count = 0;
A: 读取 this.count 到一个寄存器 (0)
B: 读取 this.count 到一个寄存器 (0)
B: 将寄存器的值加 2
B: 回写寄存器值(2)到内存. this.count 现在等于 2
A: 将寄存器的值加 3
A: 回写寄存器值(3)到内存. this.count 现在等于 3
```

两个线程分别加了 2 和 3 到 count 变量上，两个线程执行结束后 count 变量的值应该等于 5。然而由于两个线程是交叉执行的，两个线程从内存中读出的初始值都是 0。然后各自加了 2 和 3，并分别写回内存。最终的值并不是期望的 5，而是最后写回内存的那个线程的值，上面例子中最后写回内存的是线程 A，但实际中也可能是线程 B。如果没有采用合适的同步机制，线程间的交叉执行情况就无法预料。

竞态条件 & 临界区

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。导致竞态条件发生的代码区称作临界区。上例中 add()方法就是一个临界区,它会产生竞态条件。在临界区中使用适当的同步就可以避免竞态条件。

7

线程安全与共享资源

允许被多个线程同时执行的代码称作线程安全的代码。线程安全的代码不包含竞态条件。当多个线程同时更新共享资源时会引发竞态条件。因此，了解 Java 线程执行时共享了什么资源很重要。

局部变量

局部变量存储在线程自己的栈中。也就是说，局部变量永远也不会被多个线程共享。所以，基础类型的局部变量是线程安全的。下面是基础类型的局部变量的一个例子：

```
public void someMethod(){

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

局部的对象引用

对象的局部引用和基础类型的局部变量不太一样。尽管引用本身没有被共享，但引用所指的对象并没有存储在线程的栈内。所有的对象都存在共享堆中。如果在某个方法中创建的对象不会逃逸出（*译者注：即该对象不会被其它方法获得，也不会被非局部变量引用到*）该方法，那么它就是线程安全的。实际上，哪怕将这个对象作为参数传给其它方法，只要别的线程获取不到这个对象，那它仍是线程安全的。下面是一个线程安全的局部引用样例：

```
public void someMethod(){

  LocalObject localObject = new LocalObject();

  localObject.callMethod();
  method2(localObject);
}

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

样例中 LocalObject 对象没有被方法返回，也没有被传递给 someMethod()方法外的对象。每个执行 someMethod()的线程都会创建自己的 LocalObject 对象，并赋值给 localObject 引用。因此，这里的 LocalObject 是线程安全的。事实上，整个 someMethod()都是线程安全的。即使将 LocalObject 作为参数传给同一个类的其它方法或其它类的方法时，它仍然是线程安全的。当然，如果 LocalObject 通过某些方法被传给了别的线程，那它就不再是线程安全的了。

对象成员

对象成员存储在堆上。如果两个线程同时更新同一个对象的同一个成员，那这个代码就不是线程安全的。下面是一个样例：

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

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

如果两个线程同时调用同一个 NotThreadSafe 实例上的 add()方法，就会有竞态条件问题。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");
  }
}
```

注意两个 MyRunnable 共享了同一个 NotThreadSafe 对象。因此，当它们调用 add()方法时会造成竞态条件。

当然，如果这两个线程在不同的 NotThreadSafe 实例上调用 call()方法，就不会导致竞态条件。下面是稍微修改后的例子：

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

现在两个线程都有自己单独的 NotThreadSafe 对象，调用 add()方法时就会互不干扰，再也不会有竞态条件问题了。所以非线程安全的对象仍可以通过某种方式来消除竞态条件。

线程控制逃逸规则

线程控制逃逸规则可以帮助你判断代码中对某些资源的访问是否是线程安全的。

> 如果一个资源的创建，使用，销毁都在同一个线程内完成，
> 且永远不会脱离该线程的控制，则该资源的使用就是线程安全的。

资源可以是对象，数组，文件，数据库连接，套接字等等。Java 中你无需主动销毁对象，所以“销毁”指不再有引用指向对象。

即使对象本身线程安全，但如果该对象中包含其他资源（文件，数据库连接），整个应用也许就不再是线程安全的了。比如 2 个线程都创建了各自的数据库连接，每个连接自身是线程安全的，但它们所连接到的同一个数据库也许不是线程安全的。比如，2 个线程执行如下代码：

> 检查记录 X 是否存在，如果不存在，插入 X

如果两个线程同时执行，而且碰巧检查的是同一个记录，那么两个线程最终可能都插入了记录：

> 线程 1 检查记录 X 是否存在。检查结果：不存在
> 线程 2 检查记录 X 是否存在。检查结果：不存在
> 线程 1 插入记录 X
> 线程 2 插入记录 X

同样的问题也会发生在文件或其他共享资源上。因此，区分某个线程控制的对象是资源本身，还是仅仅到某个资源的引用很重要。

8

线程安全及不可变性

当多个线程同时访问同一个资源，并且其中的一个或者多个线程对这个资源进行了写操作，才会产生竞态条件。多个线程同时读同一个资源不会产生竞态条件。

我们可以通过创建不可变的共享对象来保证对象在线程间共享时不会被修改，从而实现线程安全。如下示例：

```
public class ImmutableValue{
    private int value = 0;

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

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

请注意 ImmutableValue 类的成员变量 value 是通过构造函数赋值的，并且在类中没有 set 方法。这意味着一旦 ImmutableValue 实例被创建，value 变量就不能再被修改，这就是不可变性。但你可以通过 getValue()方法读取这个变量的值。

（译者注：注意，“不变”（Immutable）和“只读”（Read Only）是不同的。当一个变量是“只读”时，变量的值不能直接改变，但是可以在其它变量发生改变的时候发生改变。比如，一个人的出生年月日是“不变”属性，而一个人的年龄便是“只读”属性，但是不是“不变”属性。随着时间的变化，一个人的年龄会随之发生变化，而一个人的出生年月日则不会变化。这就是“不变”和“只读”的区别。（摘自《Java 与模式》第 34 章））

如果你需要对 ImmutableValue 类的实例进行操作，可以通过得到 value 变量后创建一个新的实例来实现，下面是一个对 value 变量进行加法操作的示例：

```
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);
```

```
  }
}
```

请注意 add()方法以加法操作的结果作为一个新的 ImmutableValue 类实例返回，而不是直接对它自己的 value 变量进行操作。

引用不是线程安全的!

重要的是要记住，即使一个对象是线程安全的不可变对象，指向这个对象的引用也可能不是线程安全的。看这个例子：

```
public void 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 类持有一个指向 ImmutableValue 实例的引用。注意，通过 setValue()方法和 add()方法可能会改变这个引用。因此，即使 Calculator 类内部使用了一个不可变对象，但 Calculator 类本身还是可变的，因此 Calculator 类不是线程安全的。换句话说：ImmutableValue 类是线程安全的，但使用它的类不是。当尝试通过不可变性去获得线程安全时，这点是需要牢记的。

要使 Calculator 类实现线程安全，将 getValue()、setValue()和 add()方法都声明为同步方法即可。

9

Java 内存模型

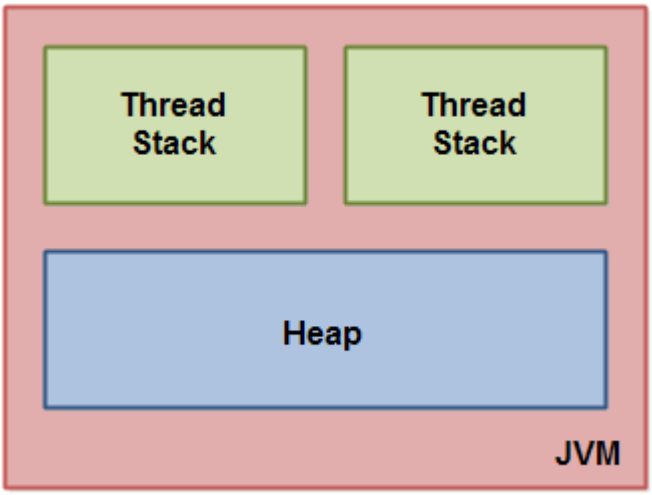
Java 内存模型规范了 Java 虚拟机与计算机内存是如何协同工作的。Java 虚拟机是一个完整的计算机的一个模型，因此这个模型自然也包含一个内存模型——又称为 Java 内存模型。

如果你想设计表现良好的并发程序，理解 Java 内存模型是非常重要的。Java 内存模型规定了如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。

原始的 Java 内存模型存在一些不足，因此 Java 内存模型在 Java1.5 时被重新修订。这个版本的 Java 内存模型在 Java8 中人在使用。

Java 内存模型内部原理

Java 内存模型把 Java 虚拟机内部划分为线程栈和堆。这张图演示了 Java 内存模型的逻辑视图。

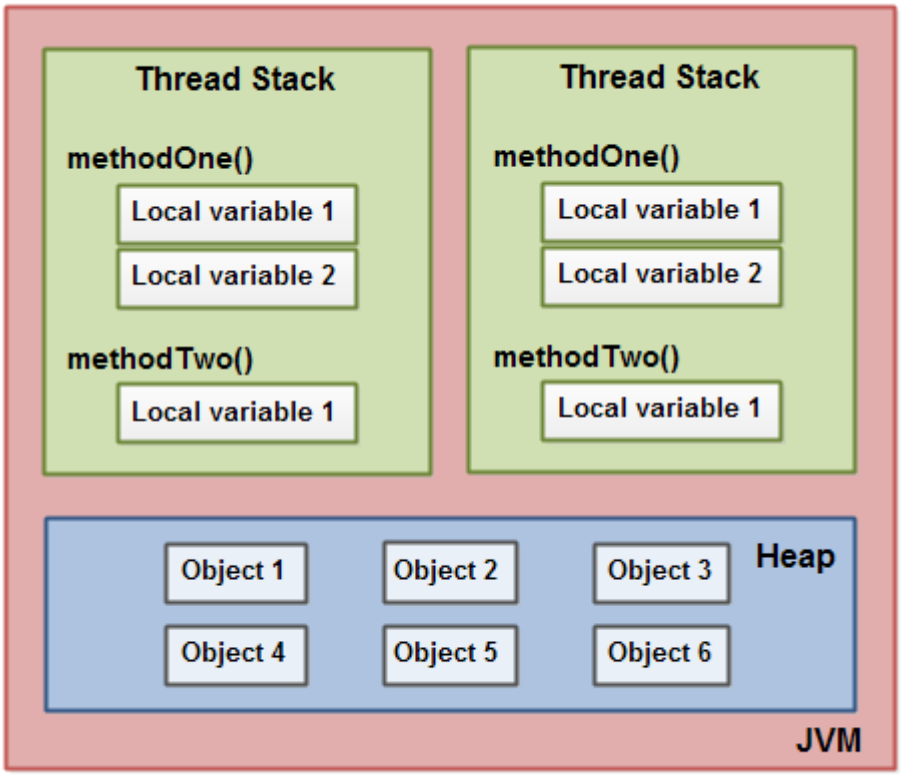


每一个运行在 Java 虚拟机里的线程都拥有自己的线程栈。这个线程栈包含了这个线程调用的方法当前执行点相关的信息。一个线程仅能访问自己的线程栈。一个线程创建的本地变量对其它线程不可见，仅自己可见。即使两个线程执行同样的代码，这两个线程任然在在自己的线程栈中的代码来创建本地变量。因此，每个线程拥有每个本地变量的独有版本。

所有原始类型的本地变量都存放在线程栈上，因此对其它线程不可见。一个线程可能向另一个线程传递一个原始类型变量的拷贝，但是它不能共享这个原始类型变量自身。

堆上包含在 Java 程序中创建的所有对象，无论是哪一个对象创建的。这包括原始类型的对象版本。如果一个对象被创建然后赋值给一个局部变量，或者用来作为另一个对象的成员变量，这个对象任然是存放在堆上。

下面这张图演示了调用栈和本地变量存放在线程栈上，对象存放在堆上。



一个本地变量可能是原始类型，在这种情况下，它总是“呆在”线程栈上。

一个本地变量也可能是指向一个对象的一个引用。在这种情况下，引用（这个本地变量）存放在线程栈上，但是对象本身存放在堆上。

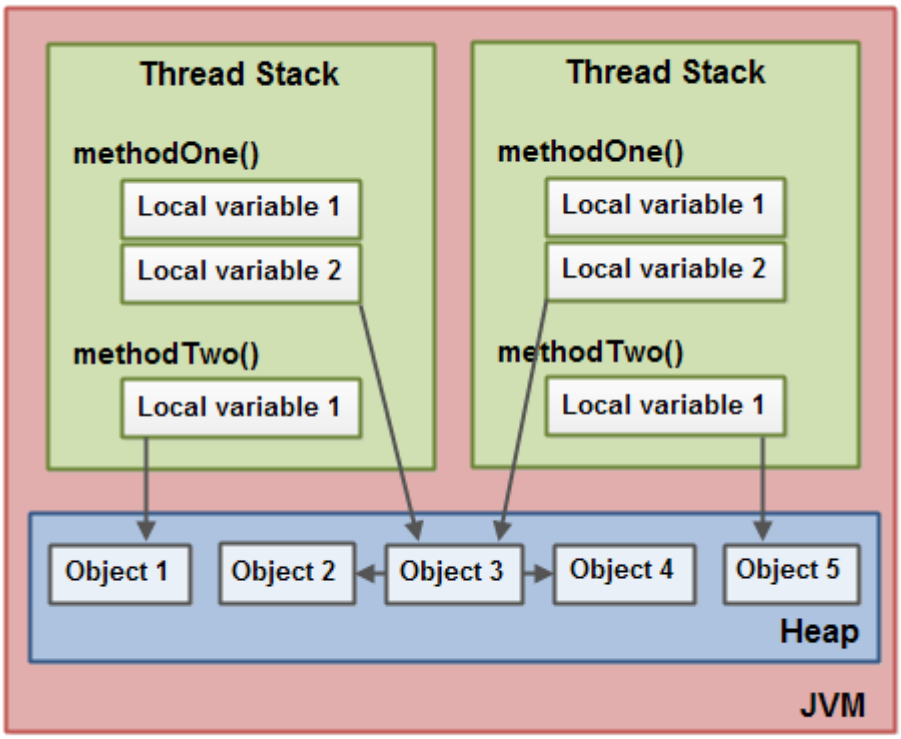
一个对象可能包含方法，这些方法可能包含本地变量。这些本地变量任然存放在线程栈上，即使这些方法所属的对象存放在堆上。

一个对象的成员变量可能随着这个对象自身存放在堆上。不管这个成员变量是原始类型还是引用类型。

静态成员变量跟随着类定义一起也存放在堆上。

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。

下图演示了上面提到的点：



两个线程拥有一些列的本地变量。其中一个本地变量（Local Variable 2）执行堆上的一个共享对象（Object 3）。这两个线程分别拥有同一个对象的不同引用。这些引用都是本地变量，因此存放在各自线程的线程栈上。这两个不同的引用指向堆上同一个对象。

注意，这个共享对象（Object 3）持有 Object2 和 Object4 一个引用作为其成员变量（如图中 Object3 指向 Object2 和 Object4 的箭头）。通过在 Object3 中这些成员变量引用，这两个线程就可以访问 Object2 和 Object4。

这张图也展示了指向堆上两个不同对象的一个本地变量。在这种情况下，指向两个不同对象的引用不是同一个对象。理论上，两个线程都可以访问 Object1 和 Object5，如果两个线程都拥有两个对象的引用。但是在上图中，每一个线程仅有一个引用指向两个对象其中之一。

因此，什么类型的 Java 代码会导致上面的内存图呢？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;
}
```

如果两个线程同时执行 run()方法，就会出现上图所示的情景。run()方法调用 methodOne()方法，methodOne()调用 methodTwo()方法。

methodOne()声明了一个原始类型的本地变量和一个引用类型的本地变量。

每个线程执行 methodOne()都会在它们对应的线程栈上创建 localVariable1 和 localVariable2 的私有拷贝。localVariable1 变量彼此完全独立，仅“生活”在每个线程的线程栈上。一个线程看不到另一个线程对它的 localVariable1 私有拷贝做出的修改。

每个线程执行 methodOne()时也将会创建它们各自的 localVariable2 拷贝。然而，两个 localVariable2 的不同拷贝都指向堆上的同一个对象。代码中通过一个静态变量设置 localVariable2 指向一个对象引用。仅存在一个静态变量的一份拷贝，这份拷贝存放在堆上。因此，localVariable2 的两份拷贝都指向由 MySharedObject 指向的静态变量的同一个实例。MySharedObject 实例也存放在堆上。它对应于上图中的 Object3。

注意，MySharedObject 类也包含两个成员变量。这些成员变量随着这个对象存放在堆上。这两个成员变量指向另外两个 Integer 对象。这些 Integer 对象对应于上图中的 Object2 和 Object4.

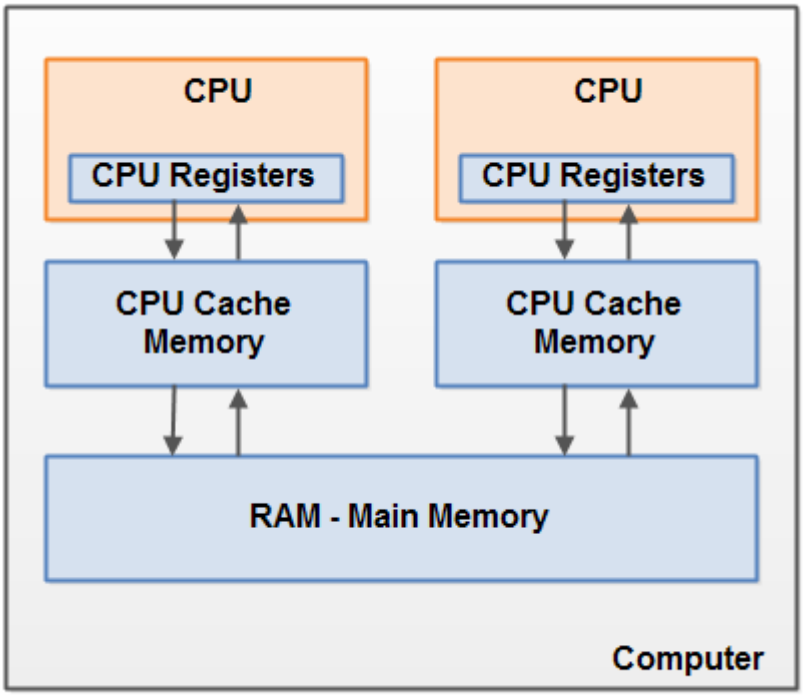
注意，methodTwo()创建一个名为 localVariable 的本地变量。这个成员变量是一个指向一个 Integer 对象的对象引用。这个方法设置 localVariable1 引用指向一个新的 Integer 实例。在执行 methodTwo 方法时，localVariable1 引用将会在每个线程中存放一份拷贝。这两个 Integer 对象实例化将会被存储堆上，但是每次执行这个方法时，这个方法都会创建一个新的 Integer 对象，两个线程执行这个方法将会创建两个不同的 Integer 实例。methodTwo 方法创建的 Integer 对象对应于上图中的 Object1 和 Object5。

还有一点，MySharedObject 类中的两个 long 类型的成员变量是原始类型的。因为，这些变量是成员变量，所以它们任然随着该对象存放在堆上，仅有本地变量存放在线程栈上。

硬件内存架构

现代硬件内存模型与 Java 内存模型有一些不同。理解内存模型架构以及 Java 内存模型如何与它协同工作也是非常重要的。这部分描述了通用的硬件内存架构，下面的部分将会描述 Java 内存是如何与它“联手”工作的。

下面是现代计算机硬件架构的简单图示：



一个现代计算机通常由两个或者多个 CPU。其中一些 CPU 还有多核。从这一点可以看出，在一个有两个或者多个 CPU 的现代计算机上同时运行多个线程是可能的。每个 CPU 在某一时刻运行一个线程是没有问题的。这意味着，如果你的 Java 程序是多线程的，在你的 Java 程序中每个 CPU 上一个线程可能同时（并发）执行。

每个 CPU 都包含一系列的寄存器，它们是 CPU 内内存的基础。CPU 在寄存器上执行操作的速度远大于在主存上执行的速度。这是因为 CPU 访问寄存器的速度远大于主存。

每个 CPU 可能还有一个 CPU 缓存层。实际上，绝大多数的现代 CPU 都有一定大小的缓存层。CPU 访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。一些 CPU 还有多层缓存，但这些对理解 Java 内存模型如何和内存交互不是那么重要。只要知道 CPU 中可以有一个缓存层就可以了。

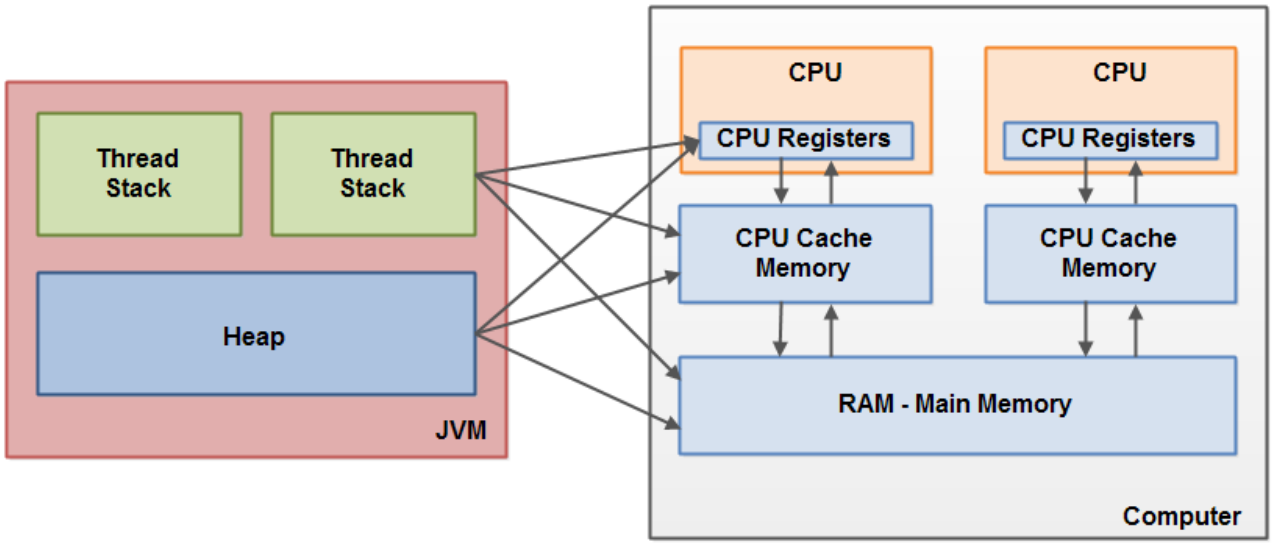
一个计算机还包含一个主存。所有的 CPU 都可以访问主存。主存通常比 CPU 中的缓存大得多。

通常情况下，当一个 CPU 需要读取主存时，它会将主存的部分读到 CPU 缓存中。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当 CPU 需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在某个时间点将值刷新回主存。

当 CPU 需要在缓存层存放一些东西的时候，存放在缓存中的内容通常会被刷新回主存。CPU 缓存可以在某一时刻将数据局部写到它的内存中，和在某一时刻局部刷新它的内存。它不会再某一时刻读/写整个缓存。通常，在一个被称作“cache lines”的更小的内存块中缓存被更新。一个或者多个缓存行可能被读到缓存，一个或者多个缓存行可能再被刷新回主存。

Java 内存模型和硬件内存架构之间的桥接

上面已经提到，Java 内存模型与硬件内存架构之间存在差异。硬件内存架构没有区分线程栈和堆。对于硬件，所有的线程栈和堆都分布在主内中。部分线程栈和堆可能有时候会出现在 CPU 缓存中和 CPU 内部的寄存器中。如下图所示：



当对象和变量被存放在计算机中各种不同的内存区域中时，就可能会出现一些具体的问题。主要包括如下两个方面：

- 线程对共享变量修改的可见性
- 当读，写和检查共享变量时出现 race conditions

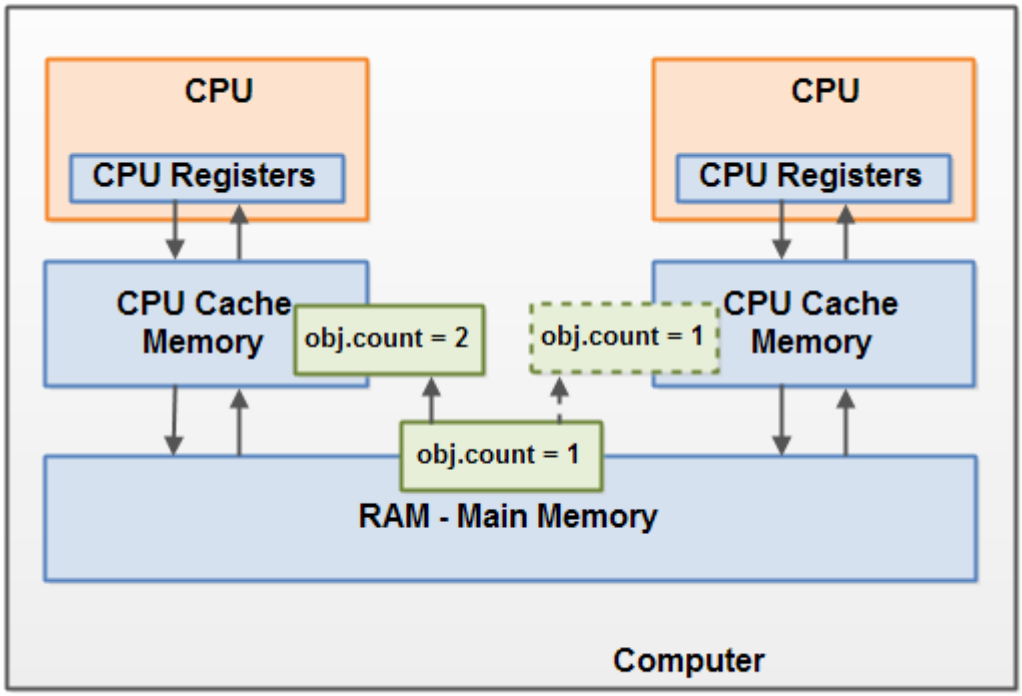
下面我们专门来解释以下这两个问题。

共享对象可见性

如果两个或者更多的线程在没有正确的使用 volatile 声明或者同步的情况下共享一个对象，一个线程更新这个共享对象可能对其它线程来说是不接见的。

想象一下，共享对象被初始化在主存中。跑在 CPU 上的一个线程将这个共享对象读到 CPU 缓存中。然后修改了这个对象。只要 CPU 缓存没有被刷新会主存，对象修改后的版本对跑在其它 CPU 上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的 CPU 缓存中。

下图示意了这种情形。跑在左边 CPU 的线程拷贝这个共享对象到它的 CPU 缓存中，然后将 count 变量的值修改为 2。这个修改对跑在右边 CPU 上的其它线程是不可见的，因为修改后的 count 的值还没有被刷新回主存中去。



解决这个问题你可以使用 Java 中的 volatile 关键字。volatile 关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。

Race Conditions

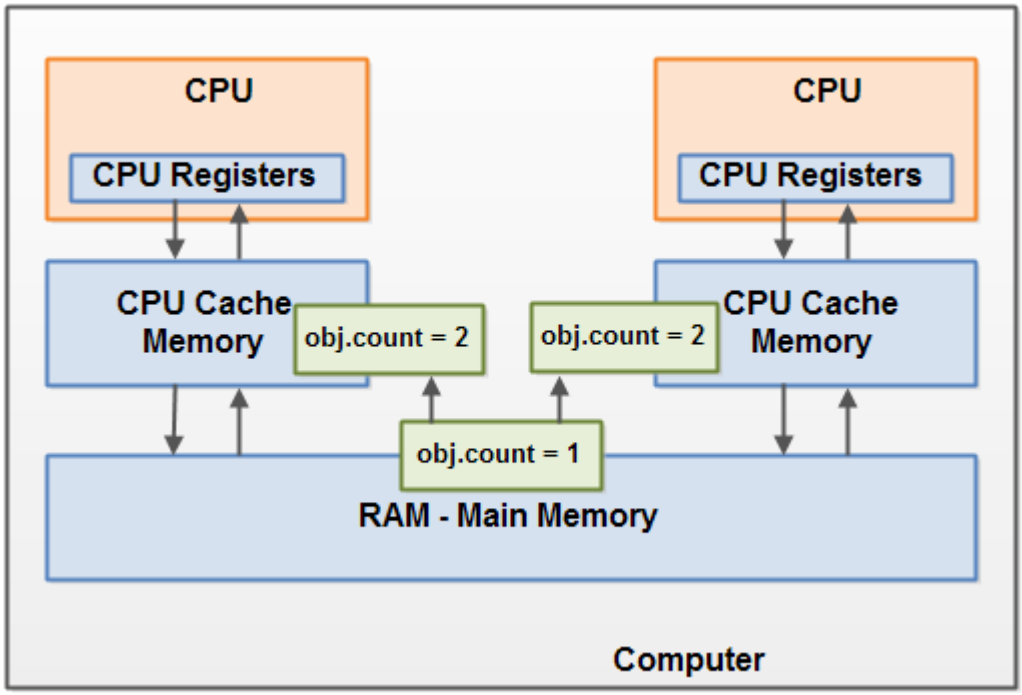
如果两个或者更多的线程共享一个对象，多个线程在这个共享对象上更新变量，就有可能发生 race conditions。

想象一下，如果线程 A 读一个共享对象的变量 count 到它的 CPU 缓存中。再想象一下，线程 B 也做了同样的事情，但是往一个不同的 CPU 缓存中。现在线程 A 将 count 加 1，线程 B 也做了同样的事情。现在 count 已经被增在了两个，每个 CPU 缓存中一次。

如果这些增加操作被顺序的执行，变量 count 应该被增加两次，然后原值+2 被写回到主存中去。

然而，两次增加都是在没有适当的同步下并发执行的。无论是线程 A 还是线程 B 将 count 修改后的版本写回到主存中取，修改后的值仅会被原值大 1，尽管增加了两次。

下图演示了上面描述的情况：



解决这个问题可以使用 Java 同步块。一个同步块可以保证在同一时刻仅有一个线程可以进入代码的临界区。同步块还可以保证代码块中所有被访问的变量将会从主存中读入，当线程退出同步代码块时，所有被更新的变量都会被刷新回主存中去，不管这个变量是否被声明为 volatile。

10

Java 同步块

Java 同步块（synchronized block）用来标记方法或者代码块是同步的。Java 同步块用来避免竞争。本文介绍以下内容：

- Java 同步关键字（synchronzied）
- 实例方法同步
- 静态方法同步
- 实例方法中同步块
- 静态方法中同步块
- Java 同步示例

Java 同步关键字（synchronized）

Java 中的同步块用 synchronized 标记。同步块在 Java 中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

有四种不同的同步块：

1. 实例方法
2. 静态方法
3. 实例方法中的同步块
4. 静态方法中的同步块

上述同步块都同步在不同对象上。实际需要那种同步块视具体情况而定。

实例方法同步

下面是一个同步的实例方法：

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

注意在方法声明中同步（ synchronized ）关键字。这告诉 Java 该方法是同步的。

Java 实例方法同步是同步在拥有该方法的对象上。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么一个线程一次可以在一个实例同步块中执行操作。一个实例一个线程。

静态方法同步

静态方法同步和实例方法同步方法一样，也使用 synchronized 关键字。Java 静态方法同步如下示例：

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

同样，这里 synchronized 关键字告诉 Java 这个方法是同步的。

静态方法的同步是指同步在该方法所在的类对象上。因为在 Java 虚拟机中一个类只能对应一个类对象，所以同时只允许一个线程执行同一个类中的静态同步方法。

对于不同类中的静态同步方法，一个线程可以执行每个类中的静态同步方法而无需等待。不管类中的那个静态同步方法被调用，一个类只能由一个线程同时执行。

实例方法中的同步块

有时你不需要同步整个方法，而是同步方法中的一部分。Java 可以对方法的一部分进行同步。

在非同步的 Java 方法中的同步块的例子如下所示：

```
public void add(int value){

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

示例使用 Java 同步块构造器来标记一块代码是同步的。该代码在执行时和同步方法一样。

注意 Java 同步块构造器用括号将对象括起来。在上例中，使用了“this”，即为调用 add 方法的实例本身。在同步构造器中用括号括起来的对象叫做监视器对象。上述代码使用监视器对象同步，同步实例方法使用调用方法本身的实例作为监视器对象。

一次只有一个线程能够在同步于同一个监视器对象的 Java 方法内执行。

下面两个例子都同步他们所调用的实例对象上，因此他们在同步的执行效果上是等效的。

```
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);
    }
  }
}
```

在上例中，每次只有一个线程能够在两个同步块中任意一个方法内执行。

如果第二个同步块不是同步在 this 实例对象上，那么两个方法可以被线程同时执行。

静态方法中的同步块

和上面类似，下面是两个静态方法同步的例子。这些方法同步在该方法所属的类对象上。

```
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);
      }
    }
  }
```

这两个方法不允许同时被线程访问。

如果第二个同步块不是同步在 MyClass.class 这个对象上。那么这两个方法可以同时被线程访问。

Java 同步实例

在下面例子中，启动了两个线程，都调用 Counter 类同一个实例的 add 方法。因为同步在该方法所属的实例上，所以同时只能有一个线程访问该方法。

```
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);
       }
    }
 }
 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();
   }
 }
```

创建了两个线程。他们的构造器引用同一个 Counter 实例。Counter.add 方法是同步在实例上，是因为 add 方法是实例方法并且被标记上 synchronized 关键字。因此每次只允许一个线程调用该方法。另外一个线程必须要等到第一个线程退出 add()方法时，才能继续执行方法。

如果两个线程引用了两个不同的 Counter 实例，那么他们可以同时调用 add()方法。这些方法调用了不同的对象，因此这些方法也就同步在不同的对象上。这些方法调用将不会被阻塞。如下面这个例子所示：

```
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();
  }
}
```

注意这两个线程，threadA 和 threadB，不再引用同一个 counter 实例。CounterA 和 counterB 的 add 方法同步在他们所属的对象上。调用 counterA 的 add 方法将不会阻塞调用 counterB 的 add 方法。

线程通信

线程通信的目标是使线程间能够互相发送信号。另一方面，线程通信使线程能够等待其他线程的信号。

例如，线程 B 可以等待线程 A 的一个信号，这个信号会通知线程 B 数据已经准备好了。本文将讲解以下几个 JAVA 线程间通信的主题：

1. 通过共享对象通信
2. 忙等待
3. wait()，notify()和 notifyAll()
4. 丢失的信号
5. 假唤醒
6. 多线程等待相同信号
7. 不要对常量字符串或全局对象调用 wait()

通过共享对象通信

线程间发送信号的一个简单方式是在共享对象的变量里设置信号值。线程 A 在一个同步块里设置 boolean 型成员变量 hasDataToProcess 为 true，线程 B 也在同步块里读取 hasDataToProcess 这个成员变量。这个简单的例子使用了一个持有信号的对象，并提供了 set 和 check 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 必须获得指向一个 MySignal 共享实例的引用，以便进行通信。如果它们持有的引用指向不同的 MySingal 实例，那么彼此将不能检测到对方的信号。需要处理的数据可以存放在一个共享缓存区里，它和 MySignal 实例是分开存放的。

忙等待(Busy Wait)

准备处理数据的线程 B 正在等待数据变为可用。换句话说，它在等待线程 A 的一个信号，这个信号使 hasDataToProcess()返回 true。线程 B 运行在一个循环里，以等待这个信号：

```
protected MySignal sharedSignal = ...

...

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

wait()，notify()和 notifyAll()

忙等待没有对运行等待线程的 CPU 进行有效的利用，除非平均等待时间非常短。否则，让等待线程进入睡眠或者非运行状态更为明智，直到它接收到它等待的信号。

Java 有一个内建的等待机制来允许线程在等待信号的时候变为非运行状态。java.lang.Object 类定义了三个方法，wait()、notify()和 notifyAll()来实现这个等待机制。

一个线程一旦调用了任意对象的 wait()方法，就会变为非运行状态，直到另一个线程调用了同一个对象的 notify()方法。为了调用 wait()或者 notify()，线程必须先获得那个对象的锁。也就是说，线程必须在同步块里调用 wait()或者 notify()。以下是 MySingal 的修改版本——使用了 wait()和 notify()的 MyWaitNotify：

```
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()，而唤醒线程将调用 doNotify()。当一个线程调用一个对象的 notify()方法，正在等待该对象的所有线程中将有一个线程被唤醒并允许执行（校注：这个将被唤醒的线程是随机的，不可以指定唤醒哪个线程）。同时也提供了一个 notifyAll()方法来唤醒正在等待一个给定对象的所有线程。

如你所见，不管是等待线程还是唤醒线程都在同步块里调用 wait()和 notify()。这是强制性的！一个线程如果没有持有对象锁，将不能调用 wait()，notify()或者 notifyAll()。否则，会抛出 IllegalMonitorStateException 异常。

（*校注：JVM 是这么实现的，当你调用 wait 时候它首先要检查下当前线程是否是锁的拥有者，不是则抛出 IllegalMonitorStateExcept。*）

但是，这怎么可能？等待线程在同步块里面执行的时候，不是一直持有监视器对象（myMonitor 对象）的锁吗？等待线程不能阻塞唤醒线程进入 doNotify()的同步块吗？答案是：的确不能。一旦线程调用了 wait()方法，它就释放了所持有的监视器对象上的锁。这将允许其他线程也可以调用 wait()或者 notify()。

一旦一个线程被唤醒，不能立刻就退出 wait()的方法调用，直到调用 notify()的线程退出了它自己的同步块。换句话说：被唤醒的线程必须重新获得监视器对象的锁，才可以退出 wait()的方法调用，因为 wait 方法调用运行在同步块里面。如果多个线程被 notifyAll()唤醒，那么在同一时刻将只有一个线程可以退出 wait()方法，因为每个线程在退出 wait()前必须获得监视器对象的锁。

丢失的信号（Missed Signals）

notify()和 notifyAll()方法不会保存调用它们的方法，因为当这两个方法被调用时，有可能没有线程处于等待状态。通知信号过后便丢弃了。因此，如果一个线程先于被通知线程调用 wait()前调用了 notify()，等待的线程将错过这个信号。这可能是也可能不是个问题。不过，在某些情况下，这可能使等待线程永远在等待，不再醒来，因为线程错过了唤醒信号。

为了避免丢失信号，必须把它们保存在信号类里。在 MyWaitNotify 的例子中，通知信号应被存储在 MyWaitNotify 实例的一个成员变量里。以下是 MyWaitNotify 的修改版本：

```
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();
    }
  }
}
```

留意 doNotify()方法在调用 notify()前把 wasSignalled 变量设为 true。同时，留意 doWait()方法在调用 wait()前会检查 wasSignalled 变量。事实上，如果没有信号在前一次 doWait()调用和这次 doWait()调用之间的时间段里被接收到，它将只调用 wait()。

（校注：为了避免信号丢失， 用一个变量来保存是否被通知过。在 notify 前，设置自己已经被通知过。在 wait 后，设置自己没有被通知过，需要等待通知。）

假唤醒

由于莫名其妙的原因，线程有可能在没有调用过 notify()和 notifyAll()的情况下醒来。这就是所谓的假唤醒（ spurious wakeups ）。无端端地醒过来了。

如果在 MyWaitNotify2 的 doWait()方法里发生了假唤醒，等待线程即使没有收到正确的信号，也能够执行后续的操作。这可能导致你的应用程序出现严重问题。

为了防止假唤醒，保存信号的成员变量将在一个 while 循环里接受检查，而不是在 if 表达式里。这样的一个 while 循环叫做自旋锁（ 校注：这种做法要慎重，目前的 JVM 实现自旋会消耗 CPU，如果长时间不调用 doNotify 方法，doWait 方法会一直自旋，CPU 会消耗太大 ）。被唤醒的线程会自旋直到自旋锁(while 循环)里的条件变为 false。以下 MyWaitNotify2 的修改版本展示了这点：

```
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();
    }
  }
}
```

留意 wait()方法是在 while 循环里，而不在 if 表达式里。如果等待线程没有收到信号就唤醒，wasSignalled 变量将变为 false，while 循环会再执行一次，促使醒来的线程回到等待状态。

多个线程等待相同信号

如果你有多个线程在等待，被 notifyAll()唤醒，但只有一个被允许继续执行，使用 while 循环也是个好方法。每次只有一个线程可以获得监视器对象锁，意味着只有一个线程可以退出 wait()调用并清除 wasSignalled 标志（设为 false）。一旦这个线程退出 doWait()的同步块，其他线程退出 wait()调用，并在 while 循环里检查 wasSignalled 变量值。但是，这个标志已经被第一个唤醒的线程清除了，所以其余醒来的线程将回到等待状态，直到下次信号到来。

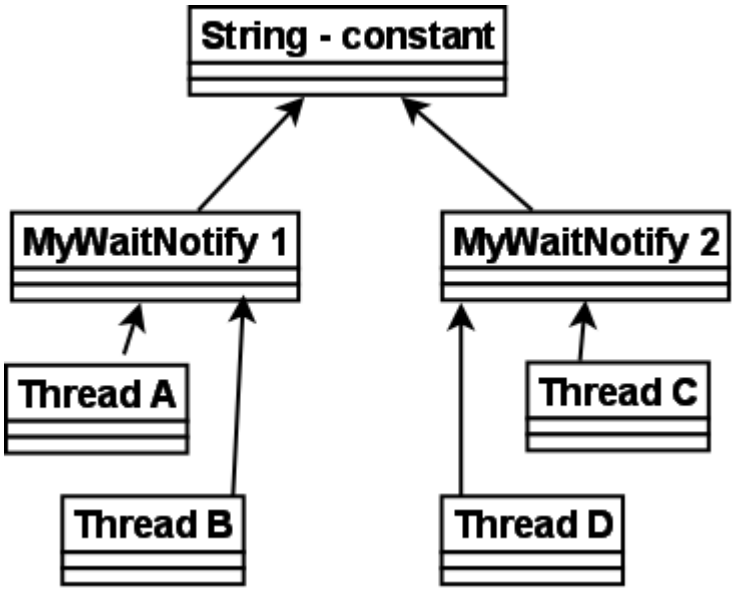
不要在字符串常量或全局对象中调用 wait()

（校注：本章说的字符串常量指的是值为常量的变量）

本文早期的一个版本在 MyWaitNotify 例子里使用字符串常量（””）作为管程对象。以下是那个例子：

```
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();
    }
  }
}
```

在空字符串作为锁的同步块(或者其他常量字符串)里调用 wait()和 notify()产生的问题是，JVM/编译器内部会把常量字符串转换成同一个对象。这意味着，即使你有 2 个不同的 MyWaitNotify 实例，它们都引用了相同的空字符串实例。同时也意味着存在这样的风险：在第一个 MyWaitNotify 实例上调用 doWait()的线程会被在第二个 MyWaitNotify 实例上调用 doNotify()的线程唤醒。这种情况可以画成以下这张图：



起初这可能不像个大问题。毕竟，如果 doNotify()在第二个 MyWaitNotify 实例上被调用，真正发生的事不外乎线程 A 和 B 被错误的唤醒了 。这个被唤醒的线程（A 或者 B）将在 while 循环里检查信号值，然后回到等待状态，因为 doNotify()并没有在第一个 MyWaitNotify 实例上调用，而这个正是它要等待的实例。这种情况相当于引发了一次假唤醒。线程 A 或者 B 在信号值没有更新的情况下唤醒。但是代码处理了这种情况，所以线程回到了等待状态。记住，即使 4 个线程在相同的共享字符串实例上调用 wait()和 notify()，doWait()和 doNotify()里的信号还会被 2 个 MyWaitNotify 实例分别保存。在 MyWaitNotify1 上的一次 doNotify()调用可能唤醒 MyWaitNotify2 的线程，但是信号值只会保存在 MyWaitNotify1 里。

问题在于，由于 doNotify()仅调用了 notify()而不是 notifyAll()，即使有 4 个线程在相同的字符串（空字符串）实例上等待，只能有一个线程被唤醒。所以，如果线程 A 或 B 被发给 C 或 D 的信号唤醒，它会检查自己的信号值，看看有没有信号被接收到，然后回到等待状态。而 C 和 D 都没被唤醒来检查它们实际上接收到的信号值，这样信号便丢失了。这种情况相当于前面所说的丢失信号的问题。C 和 D 被发送过信号，只是都不能对信号作出回应。

如果 doNotify()方法调用 notifyAll()，而非 notify()，所有等待线程都会被唤醒并依次检查信号值。线程 A 和 B 将回到等待状态，但是 C 或 D 只有一个线程注意到信号，并退出 doWait()方法调用。C 或 D 中的另一个将回到等待状态，因为获得信号的线程在退出 doWait()的过程中清除了信号值(置为 false)。

看过上面这段后，你可能会设法使用 notifyAll()来代替 notify()，但是这在性能上是个坏主意。在只有一个线程能对信号进行响应的情况下，没有理由每次都去唤醒所有线程。

所以：**在 wait()/notify()机制中，不要使用全局对象，字符串常量等。应该使用对应唯一的对象**。例如，每一个 MyWaitNotify3 的实例（前一节的 例子 ）拥有一个属于自己的监视器对象，而不是在空字符串上调用 wait()/notify()。

校注：

管程 (英语：Monitors，也称为监视器) 是对多个工作线程实现互斥访问共享资源的对象或模块。这些共享资源一般是硬件设备或一群变量。管程实现了在一个时间点，最多只有一个线程在执行它的某个子程序。与那些通过修改数据结构实现互斥访问的并发程序设计相比，管程很大程度上简化了程序设计。

12

死锁

死锁是两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程 1 锁住了 A，然后尝试对 B 进行加锁，同时线程 2 已经锁住了 B，接着尝试对 A 进行加锁，这时死锁就发生了。线程 1 永远得不到 B，线程 2 也永远得不到 A，并且它们永远也不会知道发生了这样的事情。为了得到彼此的对象（A 和 B），它们将永远阻塞下去。这种情况就是一个死锁。

该情况如下：

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for A
```

这里有一个 TreeNode 类的例子，它调用了不同实例的 synchronized 方法：

```
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)方法的同时有另外一个线程 2 调用 child.setParent(parent)方法，两个线程中的 parent 表示的是同一个对象，child 亦然，此时就会发生死锁。下面的伪代码说明了这个过程：

```
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)。因为 addChild()是同步的，所以线程 1 会对 parent 对象加锁以不让其它线程访问该对象。

然后线程 2 调用 child.setParent(parent)。因为 setParent()是同步的，所以线程 2 会对 child 对象加锁以不让其它线程访问该对象。

现在 child 和 parent 对象被两个不同的线程锁住了。接下来线程 1 尝试调用 child.setParentOnly()方法，但是由于 child 对象现在被线程 2 锁住的，所以该调用会被阻塞。线程 2 也尝试调用 parent.addChildOnly()，但是由于 parent 对象现在被线程 1 锁住，导致线程 2 也阻塞在该方法处。现在两个线程都被阻塞并等待着获取另外一个线程所持有的锁。

注意：像上文描述的，这两个线程需要同时调用 parent.addChild(child)和 child.setParent(parent)方法，并且是同一个 parent 对象和同一个 child 对象，才有可能发生死锁。上面的代码可能运行一段时间才会出现死锁。

这些线程需要**同时**获得锁。举个例子，如果线程 1 稍微领先线程 2，然后成功地锁住了 A 和 B 两个对象，那么线程 2 就会在尝试对 B 加锁的时候被阻塞，这样死锁就不会发生。因为线程调度通常是不可预测的，因此没有一个办法可以准确预测**什么时候**死锁会发生，仅仅是 **可能会**发生。

更复杂的死锁

死锁可能不止包含 2 个线程，这让检测死锁变得更加困难。下面是 4 个线程发生死锁的例子：

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for C
Thread 3  locks C, waits for D
Thread 4  locks D, waits for 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。

数据库的死锁

更加复杂的死锁场景发生在数据库事务中。一个数据库事务可能由多条 SQL 更新请求组成。当在一个事务中更新一条记录，这条记录就会被锁住避免其他事务的更新请求，直到第一个事务结束。同一个事务中每一个更新请求都可能会锁住一些记录。

当多个事务同时需要对一些相同的记录做更新操作时，就很有可能发生死锁，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.
```

因为锁发生在不同的请求中，并且对于一个事务来说不可能提前知道所有它需要的锁，因此很难检测和避免数据库事务中的死锁。

13

避免死锁

在有些情况下死锁是可以避免的。本文将展示三种用于避免死锁的技术：

1. 加锁顺序
2. 加锁时限
3. 死锁检测

加锁顺序

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。看下面这个例子：

```
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（比如线程 3）需要一些锁，那么它必须按照确定的顺序获取锁。它只有获得了从顺序上排在前面的锁之后，才能获取后面的锁。

例如，线程 2 和线程 3 只有在获取了锁 A 之后才能尝试获取锁 C(*译者注：获取锁 A 是获取锁 C 的必要条件*)。因为线程 1 已经拥有了锁 A，所以线程 2 和 3 需要一直等到锁 A 被释放。然后在它们尝试对 B 或 C 加锁之前，必须成功地对 A 加了锁。

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(*译者注：并对这些锁做适当的排序*)，但总有些时候是无法预知的。

加锁时限

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(*译者注：加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑*)。

以下是一个例子，展示了两个线程以不同的顺序尝试获取相同的两个锁，在发生超时后回退并重试的场景：

```
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.
```

在上面的例子中，线程 2 比线程 1 早 200 毫秒进行重试加锁，因此它可以先成功地获取到两个锁。这时，线程 1 尝试获取锁 A 并且处于等待状态。当线程 2 结束时，线程 1 也可以顺利的获得这两个锁（除非线程 2 或者其它线程在线程 1 成功获得两个锁之前又获得其中的一些锁）。

需要注意的是，由于存在锁的超时，所以我们不能认为这种场景就一定是出现了死锁。也可能是因为获得了锁的线程（导致其它线程超时）需要很长的时间去完成它的任务。

此外，如果有非常多的线程同一时间去竞争同一批资源，就算有超时和回退机制，还是可能会导致这些线程重复地尝试但却始终得不到锁。如果只有两个线程，并且重试的超时时间设定为 0 到 500 毫秒之间，这种现象可能不会发生，但是如果是 10 个或 20 个线程情况就不同了。因为这些线程等待相等的重试时间的概率就高的多（或者非常接近以至于会出现问题）。

(译者注：超时和重试机制是为了避免在同一时间出现的竞争，但是当线程很多时，其中两个或多个线程的超时时间一样或者接近的可能性就会很大，因此就算出现竞争而导致超时后，由于超时时间一样，它们又会同时开始重试，导致新一轮的竞争，带来了新的问题。)

这种机制存在一个问题，在 Java 中不能对 synchronized 同步块设置超时时间。你需要创建一个自定义锁，或使用 Java5 中 java.util.concurrent 包下的工具。写一个自定义锁类不复杂，但超出了本文的内容。后续的 Java 并发系列会涵盖自定义锁的内容。

死锁检测

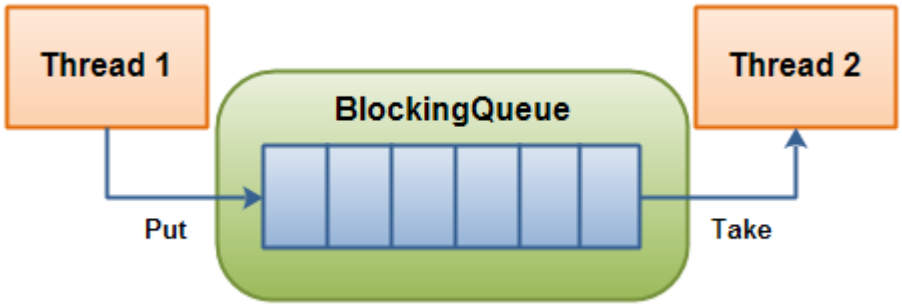
死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph 等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程 A 请求锁 7，但是锁 7 这个时候被线程 B 持有，这时线程 A 就可以检查一下线程 B 是否已经请求了线程 A 当前所持有的锁。如果线程 B 确实有这样的请求，那么就是发生了死锁（线程 A 拥有锁 1，请求锁 7；线程 B 拥有锁 7，请求锁 1）。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程 A 等待线程 B，线程 B 等待线程 C，线程 C 等待线程 D，线程 D 又在等待线程 A。线程 A 为了检测死锁，它需要递进地检测所有被 B 请求的锁。从线程 B 所请求的锁开始，线程 A 找到了线程 C，然后又找到了线程 D，发现线程 D 请求的锁被线程 A 自己持有着。这是它就知道发生了死锁。

下面是一幅关于四个线程（A,B,C 和 D）之间锁占有和请求的关系图。像这样的数据结构就可以被用来检测死锁。



那么当检测出死锁时，这些线程该做些什么呢?

一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。这个和简单的加锁超时类似，不一样的是只有死锁已经发生了才回退，而不会是因为加锁的请求超时了。虽然有回退和等待，但是如果有大量的线程竞争同一批锁，它们还是会重复地死锁（*编者注：原因同超时类似，不能从根本上减轻竞争*）。

一个更好的方案是给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁。如果赋予这些线程的优先级是固定不变的，同一批线程总是会拥有更高的优先级。为避免这个问题，可以在死锁发生的时候设置随机的优先级。

14

饥饿和公平

如果一个线程因为 CPU 时间全部被其他线程抢走而得不到 CPU 运行时间，这种状态被称之为“饥饿”。而该线程被“饥饿致死”正是因为它得不到 CPU 运行时间的机会。解决饥饿的方案被称之为“公平性” - 即所有线程均能公平地获得运行机会。

下面是本文讨论的主题：

1. Java 中导致饥饿的原因：

- 高优先级线程吞噬所有的低优先级线程的 CPU 时间。
- 线程被永久堵塞在一个等待进入同步块的状态。
- 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)。

    1. 在 Java 中实现公平性方案，需要:

- 使用锁，而不是同步块。
- 公平锁。
- 注意性能方面。

Java 中导致饥饿的原因

在 Java 中，下面三个常见的原因会导致线程饥饿：

1. 高优先级线程吞噬所有的低优先级线程的 CPU 时间。
2. 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
3. 线程在等待一个本身(在其上调用 wait())也处于永久等待完成的对象，因为其他线程总是被持续地获得唤醒。

高优先级线程吞噬所有的低优先级线程的 CPU 时间

你能为每个线程设置独自的线程优先级，优先级越高的线程获得的 CPU 时间越多，线程优先级值设置在 1 到 10 之间，而这些优先级值所表示行为的准确解释则依赖于你的应用运行平台。对大多数应用来说，你最好是不要改变其优先级值。

线程被永久堵塞在一个等待进入同步块的状态

Java 的同步代码区也是一个导致饥饿的因素。Java 的同步代码区对哪个线程允许进入的次序没有任何保障。这就意味着理论上存在一个试图进入该同步区的线程处于被永久堵塞的风险，因为其他线程总是能持续地先于它获得访问，这即是“饥饿”问题，而一个线程被“饥饿致死”正是因为它得不到 CPU 运行时间的机会。

线程在等待一个本身(在其上调用 wait())也处于永久等待完成的对象

如果多个线程处在 wait()方法执行上，而对其调用 notify()不会保证哪一个线程会获得唤醒，任何线程都有可能处于继续等待的状态。因此存在这样一个风险：一个等待线程从来得不到唤醒，因为其他等待线程总是能被获得唤醒。

在 Java 中实现公平性

虽 Java 不可能实现 100% 的公平性，我们依然可以通过同步结构在线程间实现公平性的提高。

首先来学习一段简单的同步态代码：

```
public class Synchronizer{

   public synchronized void doSynchronized(){

   //do a lot of work which takes a long time

   }
}
```

如果有一个以上的线程调用 doSynchronized()方法，在第一个获得访问的线程未完成前，其他线程将一直处于阻塞状态，而且在这种多线程被阻塞的场景下，接下来将是哪个线程获得访问是没有保障的。

使用锁方式替代同步块

为了提高等待线程的公平性，我们使用锁方式来替代同步块。

```
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();
   }
}
```

注意到 doSynchronized()不再声明为 synchronized，而是用 lock.lock()和 lock.unlock()来替代。

下面是用 Lock 类做的一个实现：

```
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();

    }
}
```

注意到上面对 Lock 的实现，如果存在多线程并发访问 lock()，这些线程将阻塞在对 lock()方法的访问上。另外，如果锁已经锁上（校对注：这里指的是 isLocked 等于 true 时），这些线程将阻塞在 while(isLocked)循环的 wait()调用里面。要记住的是，当线程正在等待进入 lock() 时，可以调用 wait()释放其锁实例对应的同步锁，使得其他多个线程可以进入 lock()方法，并调用 wait()方法。

这回看下 doSynchronized()，你会注意到在 lock()和 unlock()之间的注释：在这两个调用之间的代码将运行很长一段时间。进一步设想，这段代码将长时间运行，和进入 lock()并调用 wait()来比较的话。这意味着大部分时间用在等待进入锁和进入临界区的过程是用在 wait()的等待中，而不是被阻塞在试图进入 lock()方法中。

在早些时候提到过，同步块不会对等待进入的多个线程谁能获得访问做任何保障，同样当调用 notify()时，wait()也不会做保障一定能唤醒线程（至于为什么，请看线程通信）。因此这个版本的 Lock 类和 doSynchronized()那个版本就保障公平性而言，没有任何区别。

但我们能改变这种情况。当前的 Lock 类版本调用自己的 wait()方法，如果每个线程在不同的对象上调用 wait()，那么只有一个线程会在该对象上调用 wait()，Lock 类可以决定哪个对象能对其调用 notify()，因此能做到有效的选择唤醒哪个线程。

公平锁

下面来讲述将上面 Lock 类转变为公平锁 FairLock。你会注意到新的实现和之前的 Lock 类中的同步和 wait()/notify()稍有不同。

准确地说如何从之前的 Lock 类做到公平锁的设计是一个渐进设计的过程，每一步都是在解决上一步的问题而前进的：Nested Monitor Lockout, Slipped Conditions 和 Missed Signals。这些本身的讨论虽已超出本文的范围，但其中每一步的内容都将会专题进行讨论。重要的是，每一个调用 lock()的线程都会进入一个队列，当解锁后，只有队列里的第一个线程被允许锁住 Farlock 实例，所有其它的线程都将处于等待状态，直到他们处于队列头部。

```
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;
}

}
```

首先注意到 lock()方法不在声明为 synchronized，取而代之的是对必需同步的代码，在 synchronized 中进行嵌套。

FairLock 新创建了一个 QueueObject 的实例，并对每个调用 lock()的线程进行入队列。调用 unlock()的线程将从队列头部获取 QueueObject，并对其调用 doNotify()，以唤醒在该对象上等待的线程。通过这种方式，在同一时间仅有一个等待线程获得唤醒，而不是所有的等待线程。这也是实现 FairLock 公平性的核心所在。

请注意，在同一个同步块中，锁状态依然被检查和设置，以避免出现滑漏条件。

还需注意到，QueueObject 实际是一个 semaphore。doWait()和 doNotify()方法在 QueueObject 中保存着信号。这样做以避免一个线程在调用 queueObject.doWait()之前被另一个调用 unlock()并随之调用 queueObject.doNotify()的线程重入，从而导致信号丢失。queueObject.doWait()调用放置在 synchronized(this)块之外，以避免被 monitor 嵌套锁死，所以另外的线程可以解锁，只要当没有线程在 lock 方法的 synchronized(this)块中执行即可。

最后，注意到 queueObject.doWait()在 try – catch 块中是怎样调用的。在 InterruptedException 抛出的情况下，线程得以离开 lock()，并需让它从队列中移除。

性能考虑

如果比较 Lock 和 FairLock 类，你会注意到在 FairLock 类中 lock()和 unlock()还有更多需要深入的地方。这些额外的代码会导致 FairLock 的同步机制实现比 Lock 要稍微慢些。究竟存在多少影响，还依赖于应用在 FairLock 临界区执行的时长。执行时长越大，FairLock 带来的负担影响就越小，当然这也和代码执行的频繁度相关。

15

嵌套管程锁死

嵌套管程锁死类似于死锁， 下面是一个嵌套管程锁死的场景：

> 线程 1 获得 A 对象的锁。
> 线程 1 获得对象 B 的锁（同时持有对象 A 的锁）。
> 线程 1 决定等待另一个线程的信号再继续。
> 线程 1 调用 B.wait()，从而释放了 B 对象上的锁，但仍然持有对象 A 的锁。
>
> 线程 2 需要同时持有对象 A 和对象 B 的锁，才能向线程 1 发信号。
> 线程 2 无法获得对象 A 上的锁，因为对象 A 上的锁当前正被线程 1 持有。
> 线程 2 一直被阻塞，等待线程 1 释放对象 A 上的锁。
>
> 线程 1 一直阻塞，等待线程 2 的信号，因此，不会释放对象 A 上的锁，
> 而线程 2 需要对象 A 上的锁才能给线程 1 发信号……

你可以能会说，这是个空想的场景，好吧，让我们来看看下面这个比较挫的 Lock 实现：

```
//lock implementation with nested monitor lockout problem
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();
      }
    }
  }
}
```

可以看到，lock()方法首先在”this”上同步，然后在 monitorObject 上同步。如果 isLocked 等于 false，因为线程不会继续调用 monitorObject.wait()，那么一切都没有问题 。但是如果 isLocked 等于 true，调用 lock()方法的线程会在 monitorObject.wait()上阻塞。

这里的问题在于，调用 monitorObject.wait()方法只释放了 monitorObject 上的管程对象，而与”this“关联的管程对象并没有释放。换句话说，这个刚被阻塞的线程仍然持有”this”上的锁。

*（校对注：如果一个线程持有这种 Lock 的时候另一个线程执行了 lock 操作）*当一个已经持有这种 Lock 的线程想调用 unlock()，就会在 unlock()方法进入 synchronized(this)块时阻塞。这会一直阻塞到在 lock()方法中等待的线程离开 synchronized(this)块。但是，在 unlock 中 isLocked 变为 false，monitorObject.notify()被执行之后，lock()中等待的线程才会离开 synchronized(this)块。

简而言之，在 lock 方法中等待的线程需要其它线程成功调用 unlock 方法来退出 lock 方法，但是，在 lock()方法离开外层同步块之前，没有线程能成功执行 unlock()。

结果就是，任何调用 lock 方法或 unlock 方法的线程都会一直阻塞。这就是嵌套管程锁死。

一个更现实的例子

你可能会说，这么挫的实现方式我怎么可能会做呢？你或许不会在里层的管程对象上调用 wait 或 notify 方法，但完全有可能会在外层的 this 上调。 有很多类似上面例子的情况。例如，如果你准备实现一个公平锁。你可能希望每个线程在它们各自的 QueueObject 上调用 wait()，这样就可以每次唤醒一个线程。

下面是一个比较挫的公平锁实现方式：

```
//Fair Lock implementation with nested monitor lockout problem
public class FairLock {
   private boolean isLocked = false;
   private Thread lockingThread = null;
   private List waitingThreads =
      new ArrayList();

   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 {}
```

乍看之下，嗯，很好，但是请注意 lock 方法是怎么调用 queueObject.wait()的，在方法内部有两个 synchronized 块，一个锁定 this，一个嵌在上一个 synchronized 块内部，它锁定的是局部变量 queueObject。

当一个线程调用 queueObject.wait()方法的时候，它仅仅释放的是在 queueObject 对象实例的锁，并没有释放”this”上面的锁。

现在我们还有一个地方需要特别注意， unlock 方法被声明成了 synchronized，这就相当于一个 synchronized（this）块。这就意味着，如果一个线程在 lock()中等待，该线程将持有与 this 关联的管程对象。所有调用 unlock()的线程将会一直保持阻塞，等待着前面那个已经获得 this 锁的线程释放 this 锁，但这永远也发生不了，因为只有某个线程成功地给 lock()中等待的线程发送了信号，this 上的锁才会释放，但只有执行 unlock()方法才会发送这个信号。

因此，上面的公平锁的实现会导致嵌套管程锁死。更好的公平锁实现方式可以参考 Starvation and Fairness。

嵌套管程锁死 VS 死锁

嵌套管程锁死与死锁很像：都是线程最后被一直阻塞着互相等待。

但是两者又不完全相同。在死锁中所说的，死锁可以通过总是以相同的顺序获取锁来避免。

但是发生嵌套管程锁死时锁获取的顺序是一致的。线程 1 获得 A 和 B，然后释放 B，等待线程 2 的信号。线程 2 需要同时获得 A 和 B，才能向线程 1 发送信号。所以，一个线程在等待唤醒，另一个线程在等待想要的锁被释放。

不同点归纳如下：

> 死锁中，二个线程都在等待对方释放锁。
>
> 嵌套管程锁死中，线程 1 持有锁 A，同时等待从线程 2 发来的信号，线程 2 需要锁 A 来发信号给线程 1。

16

Slipped Conditions

所谓 Slipped conditions，就是说， 从一个线程检查某一特定条件到该线程操作此条件期间，这个条件已经被其它线程改变，导致第一个线程在该条件上执行了错误的操作。这里有一个简单的例子：

```
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();
  }
}
```

我们可以看到，lock()方法包含了两个同步块。第一个同步块执行 wait 操作直到 isLocked 变为 false 才退出，第二个同步块将 isLocked 置为 true，以此来锁住这个 Lock 实例避免其它线程通过 lock()方法。

我们可以设想一下，假如在某个时刻 isLocked 为 false， 这个时候，有两个线程同时访问 lock 方法。如果第一个线程先进入第一个同步块，这个时候它会发现 isLocked 为 false，若此时允许第二个线程执行，它也进入第一个同步块，同样发现 isLocked 是 false。现在两个线程都检查了这个条件为 false，然后它们都会继续进入第二个同步块中并设置 isLocked 为 true。

这个场景就是 slipped conditions 的例子，两个线程检查同一个条件， 然后退出同步块，因此在这两个线程改变条件之前，就允许其它线程来检查这个条件。换句话说，条件被某个线程检查到该条件被此线程改变期间，这个条件已经被其它线程改变过了。

为避免 slipped conditions，条件的检查与设置必须是原子的，也就是说，在第一个线程检查和设置条件期间，不会有其它线程检查这个条件。

解决上面问题的方法很简单，只是简单的把 isLocked = true 这行代码移到第一个同步块中，放在 while 循环后面即可：

```
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();
  }
}
```

现在检查和设置 isLocked 条件是在同一个同步块中原子地执行了。

一个更现实的例子

也许你会说，我才不可能写这么挫的代码，还觉得 slipped conditions 是个相当理论的问题。但是第一个简单的例子只是用来更好的展示 slipped conditions。

饥饿和公平中的例子：

```
//Fair Lock implementation with nested monitor lockout problem
public class FairLock {
  private boolean isLocked = false;
  private Thread lockingThread = null;
  private List waitingThreads =
          new ArrayList();

  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 {}
```

我们可以看到 synchronized(queueObject)及其中的 queueObject.wait()调用是嵌在 synchronized(this)块里面的，这会导致嵌套管程锁死问题。为避免这个问题，我们必须将 synchronized(queueObject)块移出 synchronized(this)块。移出来之后的代码可能是这样的：

```
//Fair Lock implementation with slipped conditions problem
public class FairLock {
  private boolean isLocked = false;
  private Thread lockingThread  = null;
  private List waitingThreads =
          new ArrayList();

  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：因为我只改动了 lock()方法，这里只展现了 lock 方法。

现在 lock()方法包含了 3 个同步块。

第一个，synchronized(this)块通过 mustWait = isLocked || waitingThreads.get(0) != queueObject 检查内部变量的值。

第二个，synchronized(queueObject)块检查线程是否需要等待。也有可能其它线程在这个时候已经解锁了，但我们暂时不考虑这个问题。我们就假设这个锁处在解锁状态，所以线程会立马退出 synchronized(queueObject)块。

第三个，synchronized(this)块只会在 mustWait 为 false 的时候执行。它将 isLocked 重新设回 true，然后离开 lock()方法。

设想一下，在锁处于解锁状态时，如果有两个线程同时调用 lock()方法会发生什么。首先，线程 1 会检查到 isLocked 为 false，然后线程 2 同样检查到 isLocked 为 false。接着，它们都不会等待，都会去设置 isLocked 为 true。这就是 slipped conditions 的一个最好的例子。

解决 Slipped Conditions 问题

要解决上面例子中的 slipped conditions 问题，最后一个 synchronized(this)块中的代码必须向上移到第一个同步块中。为适应这种变动，代码需要做点小改动。下面是改动过的代码：

```
//Fair Lock implementation without nested monitor lockout problem,
//but with missed signals problem.
public class FairLock {
  private boolean isLocked = false;
  private Thread lockingThread  = null;
  private List waitingThreads =
          new ArrayList();

  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;
          }
        }
      }
    }
```

```
  }
}
```

我们可以看到对局部变量 mustWait 的检查与赋值是在同一个同步块中完成的。还可以看到，即使在 synchronized(this)块外面检查了 mustWait，在 while(mustWait)子句中，mustWait 变量从来没有在 synchronized(this)同步块外被赋值。当一个线程检查到 mustWait 是 false 的时候，它将自动设置内部的条件（isLocked），所以其它线程再来检查这个条件的时候，它们就会发现这个条件的值现在为 true 了。

synchronized(this)块中的 return;语句不是必须的。这只是个小小的优化。如果一个线程肯定不会等待（即 mustWait 为 false），那么就没必要让它进入到 synchronized(queueObject)同步块中和执行 if(mustWait)子句了。

细心的读者可能会注意到上面的公平锁实现仍然有可能丢失信号。设想一下，当该 FairLock 实例处于锁定状态时，有个线程来调用 lock()方法。执行完第一个 synchronized(this)块后，mustWait 变量的值为 true。再设想一下调用 lock()的线程是通过抢占式的，拥有锁的那个线程那个线程此时调用了 unlock()方法，但是看下之前的 unlock()的实现你会发现，它调用了 queueObject.notify()。但是，因为 lock()中的线程还没有来得及调用 queueObject.wait()，所以 queueObject.notify()调用也就没有作用了，信号就丢失掉了。如果调用 lock()的线程在另一个线程调用 queueObject.notify()之后调用 queueObject.wait()，这个线程会一直阻塞到其它线程调用 unlock 方法为止，但这永远也不会发生。

公平锁实现的信号丢失问题在饥饿和公平一文中我们已有过讨论，把 QueueObject 转变成一个信号量，并提供两个方法：doWait()和 doNotify()。这些方法会在 QueueObject 内部对信号进行存储和响应。用这种方式，即使 doNotify()在 doWait()之前调用，信号也不会丢失。

17

Java 中的锁

锁像 synchronized 同步块一样，是一种线程同步机制，但比 Java 中的 synchronized 同步块更复杂。因为锁（以及其它更高级的线程同步机制）是由 synchronized 同步块的方式实现的，所以我们还不能完全摆脱 synchronized 关键字（*译者注：这说的是 Java 5 之前的情况*）。

自 Java 5 开始，java.util.concurrent.locks 包中包含了一些锁的实现，因此你不用去实现自己的锁了。但是你仍然需要去了解怎样使用这些锁，且了解这些实现背后的理论也是很有用处的。可以参考我对 java.util.concurrent.locks.Lock 的介绍，以了解更多关于锁的信息。

以下是本文所涵盖的主题：

1. 一个简单的锁
2. 锁的可重入性
3. 锁的公平性
4. 在 finally 语句中调用 unlock()

一个简单的锁

让我们从 java 中的一个同步块开始：

```
public class Counter{
   private int count = 0;

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

可以看到在 inc()方法中有一个 synchronized(this)代码块。该代码块可以保证在同一时间只有一个线程可以执行 return ++count。虽然在 synchronized 的同步块中的代码可以更加复杂，但是++count 这种简单的操作已经足以表达出线程同步的意思。

以下的 Counter 类用 Lock 代替 synchronized 达到了同样的目的：

```
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()方法会对 Lock 实例对象进行加锁，因此所有对该对象调用 lock()方法的线程都会被阻塞，直到该 Lock 对象的 unlock()方法被调用。

这里有一个 Lock 类的简单实现：

```
public class Counter{
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();
    }
}
```

注意其中的 while(isLocked)循环，它又被叫做“自旋锁”。自旋锁以及 wait()和 notify()方法在线程通信），这个线程会重新去检查 isLocked 条件以决定当前是否可以安全地继续执行还是需要重新保持等待，而不是认为线程被唤醒了就可以安全地继续执行了。如果 isLocked 为 false，当前线程会退出 while(isLocked)循环，并将 isLocked 设回 true，让其它正在调用 lock()方法的线程能够在 Lock 实例上加锁。

当线程完成了临界区（位于 lock()和 unlock()之间）中的代码，就会调用 unlock()。执行 unlock()会重新将 isLocked 设置为 false，并且通知（唤醒）其中一个（若有的话）在 lock()方法中调用了 wait()函数而处于等待状态的线程。

锁的可重入性

Java 中的 synchronized 同步块是可重入的。这意味着如果一个 java 线程进入了代码中的 synchronized 同步块，并因此获得了该同步块使用的同步对象对应的管程上的锁，那么这个线程可以进入由同一个管程对象所同步的另一个 java 代码块。下面是一个例子：

```
public class Reentrant{
   public synchronized outer(){
      inner();
   }

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

注意 outer()和 inner()都被声明为 synchronized，这在 Java 中和 synchronized(this)块等效。如果一个线程调用了 outer()，在 outer()里调用 inner()就没有什么问题，因为这两个方法（代码块）都由同一个管程对象（”this”)所同步。如果一个线程已经拥有了一个管程对象上的锁，那么它就有权访问被这个管程对象同步的所有代码块。这就是可重入。线程可以进入任何一个它已经拥有的锁所同步着的代码块。

前面给出的锁实现不是可重入的。如果我们像下面这样重写 Reentrant 类，当线程调用 outer()时，会在 inner()方法的 lock.lock()处阻塞住。

```
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()的线程首先会锁住 Lock 实例，然后继续调用 inner()。inner()方法中该线程将再一次尝试锁住 Lock 实例，结果该动作会失败（也就是说该线程会被阻塞），因为这个 Lock 实例已经在 outer()方法中被锁住了。

两次 lock()之间没有调用 unlock()，第二次调用 lock 就会阻塞，看过 lock()实现后，会发现原因很明显：

```
public class Lock{
   boolean isLocked = false;

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

   ...
}
```

一个线程是否被允许退出 lock()方法是由 while 循环（自旋锁）中的条件决定的。当前的判断条件是只有当 isLocked 为 false 时 lock 操作才被允许，而没有考虑是哪个线程锁住了它。

为了让这个 Lock 类具有可重入性，我们需要对它做一点小的改动：

```
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();
        }
      }
   }

   ...
}
```

注意到现在的 while 循环（自旋锁）也考虑到了已锁住该 Lock 实例的线程。如果当前的锁对象没有被加锁(isLocked = false)，或者当前调用线程已经对该 Lock 实例加了锁，那么 while 循环就不会被执行，调用 lock()的线程就可以退出该方法（译者注：“被允许退出该方法”在当前语义下就是指不会调用 wait()而导致阻塞）。

除此之外，我们需要记录同一个线程重复对一个锁对象加锁的次数。否则，一次 unblock()调用就会解除整个锁，即使当前锁已经被加锁过多次。在 unlock()调用没有达到对应 lock()调用的次数之前，我们不希望锁被解除。

现在这个 Lock 类就是可重入的了。

锁的公平性

Java 的 synchronized 块并不保证尝试进入它们的线程的顺序。因此，如果多个线程不断竞争访问相同的 synchronized 同步块，就存在一种风险，其中一个或多个线程永远也得不到访问权 —— 也就是说访问权总是分配给了其它线程。这种情况被称作线程饥饿。为了避免这种问题，锁需要实现公平性。本文所展现的锁在内部是用 synchronized 同步块实现的，因此它们也不保证公平性。饥饿和公平中有更多关于该内容的讨论。

在 finally 语句中调用 unlock()

如果用 Lock 来保护临界区，并且临界区有可能会抛出异常，那么在 finally 语句中调用 unlock()就显得非常重要了。这样可以保证这个锁对象可以被解锁以便其它线程能继续对其加锁。以下是一个示例：

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

这个简单的结构可以保证当临界区抛出异常时 Lock 对象可以被解锁。如果不是在 finally 语句中调用的 unlock()，当临界区抛出异常时，Lock 对象将永远停留在被锁住的状态，这会导致其它所有在该 Lock 对象上调用 lock()的线程一直阻塞。

18

Java 中的读/写锁

相比 Java 中的锁(Locks in Java)里 Lock 实现，读写锁更复杂一些。假设你的程序中涉及到对一些共享资源的读和写操作，且写操作没有读操作那么频繁。在没有写操作的时候，两个线程同时读一个资源没有任何问题，所以应该允许多个线程能在同时读取共享资源。但是如果有一个线程想去写这些共享资源，就不应该再有其它线程对该资源进行读或写（*译者注：也就是说：读-读能共存，读-写不能共存，写-写不能共存*）。这就需要一个读/写锁来解决这个问题。

Java5 在 java.util.concurrent 包中已经包含了读写锁。尽管如此，我们还是应该了解其实现背后的原理。

以下是本文的主题

1. 读/写锁的 Java 实现(Read / Write Lock Java Implementation)
2. 读/写锁的重入(Read / Write Lock Reentrance)
3. 读锁重入(Read Reentrance)
4. 写锁重入(Write Reentrance)
5. 读锁升级到写锁(Read to Write Reentrance)
6. 写锁降级到读锁(Write to Read Reentrance)
7. 可重入的 ReadWriteLock 的完整实现(Fully Reentrant ReadWriteLock)
8. 在 finally 中调用 unlock() (Calling unlock() from a finally-clause)

读/写锁的 Java 实现

先让我们对读写访问资源的条件做个概述：

读取 没有线程正在做写操作，且没有线程在请求写操作。

写入 没有线程正在做读写操作。

如果某个线程想要读取资源，只要没有线程正在对该资源进行写操作且没有线程请求对该资源的写操作即可。我们假设对写操作的请求比对读操作的请求更重要，就要提升写请求的优先级。此外，如果读操作发生的比较频繁，我们又没有提升写操作的优先级，那么就会产生“饥饿”现象。请求写操作的线程会一直阻塞，直到所有的读线程都从 ReadWriteLock 上解锁了。如果一直保证新线程的读操作权限，那么等待写操作的线程就会一直阻塞下去，结果就是发生“饥饿”。因此，只有当没有线程正在锁住 ReadWriteLock 进行写操作，且没有线程请求该锁准备执行写操作时，才能保证读操作继续。

当其它线程没有对共享资源进行读操作或者写操作时，某个线程就有可能获得该共享资源的写锁，进而对共享资源进行写操作。有多少线程请求了写锁以及以何种顺序请求写锁并不重要，除非你想保证写锁请求的公平性。

按照上面的叙述，简单的实现出一个读/写锁，代码如下

```
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 类中，读锁和写锁各有一个获取锁和释放锁的方法。

读锁的实现在 lockRead()中,只要没有线程拥有写锁（writers==0），且没有线程在请求写锁（writeRequests==0），所有想获得读锁的线程都能成功获取。

写锁的实现在 lockWrite()中,当一个线程想获得写锁的时候，首先会把写锁请求数加 1（writeRequests++），然后再去判断是否能够真能获得写锁，当没有线程持有读锁（readers==0 ）,且没有线程持有写锁（writers==0）时就能获得写锁。有多少线程在请求写锁并无关系。

需要注意的是，在两个释放锁的方法（unlockRead，unlockWrite）中，都调用了 notifyAll 方法，而不是 notify。要解释这个原因，我们可以想象下面一种情形：

如果有线程在等待获取读锁，同时又有线程在等待获取写锁。如果这时其中一个等待读锁的线程被 notify 方法唤醒，但因为此时仍有请求写锁的线程存在（writeRequests>0），所以被唤醒的线程会再次进入阻塞状态。然而，等待写锁的线程一个也没被唤醒，就像什么也没发生过一样（译者注：信号丢失现象）。如果用的是 notifyAll 方法，所有的线程都会被唤醒，然后判断能否获得其请求的锁。

用 notifyAll 还有一个好处。如果有多个读线程在等待读锁且没有线程在等待写锁时，调用 unlockWrite()后，所有等待读锁的线程都能立马成功获取读锁 —— 而不是一次只允许一个。

读/写锁的重入

上面实现的读/写锁(ReadWriteLock) 是不可重入的，当一个已经持有写锁的线程再次请求写锁时，就会被阻塞。原因是已经有一个写线程了——就是它自己。此外，考虑下面的例子：

1. Thread 1 获得了读锁。
2. Thread 2 请求写锁，但因为 Thread 1 持有了读锁，所以写锁请求被阻塞。
3. Thread 1 再想请求一次读锁，但因为 Thread 2 处于请求写锁的状态，所以想再次获取读锁也会被阻塞。

   上面这种情形使用前面的 ReadWriteLock 就会被锁定——一种类似于死锁的情形。不会再有线程能够成功获取读锁或写锁了。

为了让 ReadWriteLock 可重入，需要对它做一些改进。下面会分别处理读锁的重入和写锁的重入。

读锁重入

为了让 ReadWriteLock 的读锁可重入，我们要先为读锁重入建立规则：

要保证某个线程中的读锁可重入，要么满足获取读锁的条件（没有写或写请求），要么已经持有读锁（不管是否有写请求）。要确定一个线程是否已经持有读锁，可以用一个 map 来存储已经持有读锁的线程以及对应线程获取读锁的次数，当需要判断某个线程能否获得读锁时，就利用 map 中存储的数据进行判断。下面是方法 lockRead 和 unlockRead 修改后的的代码：

```
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;
    }
}
```

代码中我们可以看到，只有在没有线程拥有写锁的情况下才允许读锁的重入。此外，重入的读锁比写锁优先级高。

写锁重入

仅当一个线程已经持有写锁，才允许写锁重入（再次获得写锁）。下面是方法 lockWrite 和 unlockWrite 修改后的的代码。

```
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;
    }
}
```

注意在确定当前线程是否能够获取写锁的时候，是如何处理的。

读锁升级到写锁

有时，我们希望一个拥有读锁的线程，也能获得写锁。想要允许这样的操作，要求这个线程是唯一一个拥有读锁的线程。writeLock()需要做点改动来达到这个目的：

```
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 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 类就可以从读锁升级到写锁了。

写锁降级到读锁

有时拥有写锁的线程也希望得到读锁。如果一个线程拥有了写锁，那么自然其它线程是不可能拥有读锁或写锁了。所以对于一个拥有写锁的线程，再获得读锁，是不会有什么危险的。我们仅仅需要对上面 canGrantReadAccess 方法进行简单地修改：

```
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;
   }
}
```

可重入的 ReadWriteLock 的完整实现

下面是完整的 ReadWriteLock 实现。为了便于代码的阅读与理解，简单对上面的代码做了重构。重构后的代码如下。

```
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;
    }
}
```

在 finally 中调用 unlock()

在利用 ReadWriteLock 来保护临界区时，如果临界区可能抛出异常，在 finally 块中调用 readUnlock()和 writeUnlock()就显得很重要了。这样做是为了保证 ReadWriteLock 能被成功解锁，然后其它线程可以请求到该锁。这里有个例子：

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

上面这样的代码结构能够保证临界区中抛出异常时 ReadWriteLock 也会被释放。如果 unlockWrite 方法不是在 finally 块中调用的，当临界区抛出了异常时，ReadWriteLock 会一直保持在写锁定状态，就会导致所有调用 lockRead()或 lockWrite()的线程一直阻塞。唯一能够重新解锁 ReadWriteLock 的因素可能就是 ReadWriteLock 是可重入的，当抛出异常时，这个线程后续还可以成功获取这把锁，然后执行临界区以及再次调用 unlockWrite()，这就会再次释放 ReadWriteLock。但是如果该线程后续不再获取这把锁了呢？所以，在 finally 中调用 unlockWrite 对写出健壮代码是很重要的。

19 重入锁死

重入锁死与死锁两篇文章中都有涉及到重入锁死的问题。

当一个线程重新获取锁，读写锁或其他不可重入的同步器时，就可能发生重入锁死。可重入的意思是线程可以重复获得它已经持有的锁。Java 的 synchronized 块是可重入的。因此下面的代码是没问题的：

（译者注：这里提到的锁都是指的不可重入的锁实现，并不是 Java 类库中的 Lock 与 ReadWriteLock 类）

```
public class Reentrant{
  public synchronized outer(){
    inner();
  }

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

注意 outer()和 inner()都声明为 synchronized，这在 Java 中这相当于 synchronized(this)块（*译者注：这里两个方法是实例方法，synchronized 的实例方法相当于在 this 上加锁，如果是 static 方法，则不然，更多阅读：哪个对象才是锁?*）。如果某个线程调用了 outer()，outer()中的 inner()调用是没问题的，因为两个方法都是在同一个管程对象(即 this)上同步的。如果一个线程持有某个管程对象上的锁，那么它就有权访问所有在该管程对象上同步的块。这就叫可重入。若线程已经持有锁，那么它就可以重复访问所有使用该锁的代码块。

下面这个锁的实现是不可重入的：

```
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();
  }
}
```

如果一个线程在两次调用 lock()间没有调用 unlock()方法，那么第二次调用 lock()就会被阻塞，这就出现了重入锁死。

避免重入锁死有两个选择：

1. 编写代码时避免再次获取已经持有的锁
2. 使用可重入锁

至于哪个选择最适合你的项目，得视具体情况而定。可重入锁通常没有不可重入锁那么好的表现，而且实现起来复杂，但这些情况在你的项目中也许算不上什么问题。无论你的项目用锁来实现方便还是不用锁方便，可重入特性都需要根据具体问题具体分析。

20

信号量

Semaphore（信号量）是一个线程同步结构，用于在线程间传递信号，以避免出现信号丢失（译者注：下文会具体介绍），或者像锁一样用于保护一个关键区域。自从 5.0 开始，jdk 在 java.util.concurrent 包里提供了 Semaphore 的官方实现，因此大家不需要自己去实现 Semaphore。但是还是很有必要去熟悉如何使用 Semaphore 及其背后的原理

本文的涉及的主题如下：

1. 简单的 Semaphore 实现
2. 使用 Semaphore 来发出信号
3. 可计数的 Semaphore
4. 有上限的 Semaphore
5. 把 Semaphore 当锁来使用

简单的 Semaphore 实现

下面是一个信号量的简单实现：

```
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 方法发出一个被存放在 Semaphore 内部的信号，而 Release 方法则等待一个信号，当其接收到信号后，标记位 signal 被清空，然后该方法终止。

使用这个 semaphore 可以避免错失某些信号通知。用 take 方法来代替 notify，release 方法来代替 wait。如果某线程在调用 release 等待之前调用 take 方法，那么调用 release 方法的线程仍然知道 take 方法已经被某个线程调用过了，因为该 Semaphore 内部保存了 take 方法发出的信号。而 wait 和 notify 方法就没有这样的功能。

当用 semaphore 来产生信号时，take 和 release 这两个方法名看起来有点奇怪。这两个名字来源于后面把 semaphore 当做锁的例子，后面会详细介绍这个例子，在该例子中，take 和 release 这两个名字会变得很合理。

使用 Semaphore 来产生信号

下面的例子中，两个线程通过 Semaphore 发出的信号来通知对方

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();

sender.start();

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...

}

}

}
```

可计数的 Semaphore

上面提到的 Semaphore 的简单实现并没有计算通过调用 take 方法所产生信号的数量。可以把它改造成具有计数功能的 Semaphore。下面是一个可计数的 Semaphore 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--;

}

}
```

有上限的 Semaphore

上面的 CountingSemaphore 并没有限制信号的数量。下面的代码将 CountingSemaphore 改造成一个信号数量有上限的 BoundedSemaphore。

```
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();

}

}
```

在 BoundedSemaphore 中，当已经产生的信号数量达到了上限，take 方法将阻塞新的信号产生请求，直到某个线程调用 release 方法后，被阻塞于 take 方法的线程才能传递自己的信号。

把 Semaphore 当锁来使用

当信号量的数量上限是 1 时，Semaphore 可以被当做锁来使用。通过 take 和 release 方法来保护关键区域。请看下面的例子：

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{

//critical section

} finally {

semaphore.release();

}
```

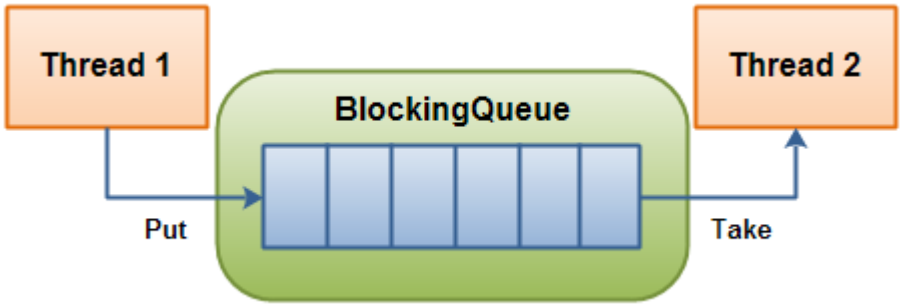
在前面的例子中，Semaphore 被用来在多个线程之间传递信号，这种情况下，take 和 release 分别被不同的线程调用。但是在锁这个例子中，take 和 release 方法将被同一线程调用，因为只允许一个线程来获取信号（允许进入关键区域的信号），其它调用 take 方法获取信号的线程将被阻塞，知道第一个调用 take 方法的线程调用 release 方法来释放信号。对 release 方法的调用永远不会被阻塞，这是因为任何一个线程都是先调用 take 方法，然后再调用 release。

通过有上限的 Semaphore 可以限制进入某代码块的线程数量。设想一下，在上面的例子中，如果 BoundedSemaphore 上限设为 5 将会发生什么？意味着允许 5 个线程同时访问关键区域，但是你必须保证，这个 5 个线程不会互相冲突。否则你的应用程序将不能正常运行。

必须注意，release 方法应当在 finally 块中被执行。这样可以保在关键区域的代码抛出异常的情况下，信号也一定会被释放。

阻塞队列

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列，下图展示了如何通过阻塞队列来合作：



线程 1 往阻塞队列中添加元素，而线程 2 从阻塞队列中移除元素

从 5.0 开始，JDK 在 java.util.concurrent 包里提供了阻塞队列的官方实现。尽管 JDK 中已经包含了阻塞队列的官方实现，但是熟悉其背后的原理还是很有帮助的。

阻塞队列的实现

阻塞队列的实现类似于带上限的 Semaphore 的实现。下面是阻塞队列的一个简单实现

```
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();

}

if(this.queue.size() == 0) {

notifyAll();

}

this.queue.add(item);

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();
```

```
}
if(this.queue.size() == this.limit){
notifyAll();
}
return this.queue.remove(0);
}
}
```

必须注意到，在 enqueue 和 dequeue 方法内部，只有队列的大小等于上限（limit）或者下限（0）时，才调用 notifyAll 方法。如果队列的大小既不等于上限，也不等于下限，任何线程调用 enqueue 或者 dequeue 方法时，都不会阻塞，都能够正常的往队列中添加或者移除元素。

22

线程池

线程池（Thread Pool）对于限制应用程序中同一时刻运行的线程数很有用。因为每启动一个新线程都会有相应的性能开销，每个线程都需要给栈分配一些内存等等。

我们可以把并发执行的任务传递给一个线程池，来替代为每个并发执行的任务都启动一个新的线程。只要池里有空闲的线程，任务就会分配给一个线程执行。在线程池的内部，任务被插入一个阻塞队列（Blocking Queue），线程池里的线程会去取这个队列里的任务。当一个新任务插入队列时，一个空闲线程就会成功的从队列中取出任务并且执行它。

线程池经常应用在多线程服务器上。每个通过网络到达服务器的连接都被包装成一个任务并且传递给线程池。线程池的线程会并发的处理连接上的请求。以后会再深入有关 Java 实现多线程服务器的细节。

Java 5 在 java.util.concurrent 包中自带了内置的线程池，所以你不用非得实现自己的线程池。你可以阅读我写的 java.util.concurrent.ExecutorService 的文章以了解更多有关内置线程池的知识。不过无论如何，知道一点关于线程池实现的知识总是有用的。

这里有一个简单的线程池实现：

```
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 void synchronized execute(Runnable task) {
    if(this.isStopped) throw
      new IllegalStateException("ThreadPool is stopped");

    this.taskQueue.enqueue(task);
  }

  public synchronized boolean stop() {
    this.isStopped = true;
    for (PoolThread thread : threads) {
```

```
      thread.stop();
    }
  }

}
```

(校注：原文有编译错误，我修改了下)

```
public class PoolThread extends Thread {

  private BlockingQueue<Runnable> taskQueue = null;
  private boolean       isStopped = false;

  public PoolThread(BlockingQueue<Runnable> queue) {
    taskQueue = queue;
  }

  public void run() {
    while (!isStopped()) {
      try {
        Runnable runnable =taskQueue.take();
        runnable.run();
      } catch(Exception e) {
        // 写日志或者报告异常,
        // 但保持线程池运行.
      }
    }
  }

  public synchronized void toStop() {
    isStopped = true;
    this.interrupt(); // 打断池中线程的 dequeue() 调用.
  }

  public synchronized boolean isStopped() {
    return isStopped;
  }
}
```

线程池的实现由两部分组成。类 ThreadPool 是线程池的公开接口，而类 PoolThread 用来实现执行任务的子线程。

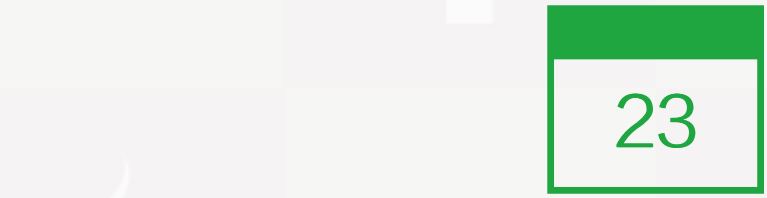
为了执行一个任务，方法 ThreadPool.execute(Runnable r)用 Runnable 的实现作为调用参数。在内部，Runnable 对象被放入阻塞队列 (Blocking Queue) ，等待着被子线程取出队列。

一个空闲的 PoolThread 线程会把 Runnable 对象从队列中取出并执行。你可以在 PoolThread.run()方法里看到这些代码。执行完毕后，PoolThread 进入循环并且尝试从队列中再取出一个任务，直到线程终止。

调用 ThreadPool.stop()方法可以停止 ThreadPool。在内部，调用 stop 先会标记 isStopped 成员变量（为 true）。然后，线程池的每一个子线程都调用 PoolThread.stop()方法停止运行。注意，如果线程池的 execute()在 stop()之后调用，execute()方法会抛出 IllegalStateException 异常。

子线程会在完成当前执行的任务后停止。注意 PoolThread.stop() 方法中调用了 this.interrupt()。它确保阻塞在 taskQueue.dequeue() 里的 wait()调用的线程能够跳出 wait()调用（校对注：因为执行了中断 interrupt，它能够打断这个调用），并且抛出一个 InterruptedException 异常离开 dequeue()方法。这个异常在 PoolThread.run()方法中被截获、报告，然后再检查 isStopped 变量。由于 isStopped 的值是 true, 因此 PoolThread.run()方法退出，子线程终止。

（校对注：看完觉得不过瘾？更详细的线程池文章参见 Java 线程池的分析和使用）



23

CAS

CAS（Compare and swap）比较和替换是设计并发算法时用到的一种技术。简单来说，比较和替换是使用一个期望值和一个变量的当前值进行比较，如果当前变量的值与我们期望的值相等，就使用一个新值替换当前变量的值。这听起来可能有一点复杂但是实际上你理解之后发现很简单，接下来，让我们跟深入的了解一下这项技术。

CAS 的使用场景

在程序和算法中一个经常出现的模式就是“check and act”模式。先检查后操作模式发生在代码中首先检查一个变量的值，然后再基于这个值做一些操作。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;
   }
}
```

上面这段代码，如果用在多线程的程序会出现很多错误，不过现在请忘掉它。

如你所见，lock()方法首先检查 locked>成员变量是否等于 false，如果等于，就将 locked 设为 true。

如果同个线程访问同一个 MyLock 实例，上面的 lock()将不能保证正常工作。如果一个线程检查 locked 的值，然后将其设置为 false，与此同时，一个线程 B 也在检查 locked 的值，又或者，在线程 A 将 locked 的值设为 false 之前。因此，线程 A 和线程 B 可能都看到 locked 的值为 false，然后两者都基于这个信息做一些操作。

为了在一个多线程程序中良好的工作，”check then act” 操作必须是原子的。原子就是说”check“操作和”act“被当做一个原子代码块执行。不存在多个线程同时执行原子块。

下面是一个代码示例，把之前的 lock()方法用 synchronized 关键字重构成一个原子块。

```
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，所以，在某一时刻只能有一个线程在同一个 MyLock 实例上执行它。

原子的 lock 方法实际上是一个”compare and swap“的例子。

CAS 用作原子操作

现在 CPU 内部已经执行原子的 CAS 操作。Java5 以来，你可以使用 java.util.concurrent.atomic 包中的一些原子类来使用 CPU 中的这些功能。

下面是一个使用 AtomicBoolean 类实现 lock()方法的例子：

```
public static class MyLock {
   private AtomicBoolean locked = new AtomicBoolean(false);

   public boolean lock() {
      return locked.compareAndSet(false, true);
   }

}
```

locked 变量不再是 boolean 类型而是 AtomicBoolean。这个类中有一个 compareAndSet()方法，它使用一个期望值和 AtomicBoolean 实例的值比较，和两者相等，则使用一个新值替换原来的值。在这个例子中，它比较 locked 的值和 false，如果 locked 的值为 false，则把修改为 true。 如果值被替换了，compareAndSet()返回 true，否则，返回 false。

使用 Java5+提供的 CAS 特性而不是使用自己实现的的好处是 Java5+中内置的 CAS 特性可以让你利用底层的你的程序所运行机器的 CPU 的 CAS 特性。这会使还有 CAS 的代码运行更快。

24

剖析同步器

虽然许多同步器（如锁，信号量，阻塞队列等）功能上各不相同，但它们的内部设计上却差别不大。换句话说，它们内部的的基础部分是相同（或相似）的。了解这些基础部件能在设计同步器的时候给我们大大的帮助。这就是本文要细说的内容。

注：本文的内容是哥本哈根信息技术大学一个由 Jakob Jenkov，Toke Johansen 和 Lars Bjørn 参与的 M.Sc.学生项目的部分成果。在此项目期间我们咨询 Doug Lea 是否知道类似的研究。有趣的是在开发 Java 5 并发工具包期间他已经提出了类似的结论。Doug Lea 的研究，我相信，在《Java Concurrency in Practice》一书中有描述。这本书有一章“剖析同步器”就类似于本文，但不尽相同。

大部分同步器都是用来保护某个区域（临界区）的代码，这些代码可能会被多线程并发访问。要实现这个目标，同步器一般要支持下列功能：

1. 状态
2. 访问条件
3. 状态变化
4. 通知策略
5. Test-and-Set 方法
6. Set 方法

并不是所有同步器都包含上述部分，也有些并不完全遵照上面的内容。但通常你能从中发现这些部分的一或多个。

状态

同步器中的状态是用来确定某个线程是否有访问权限。在 Lock 的状态是该队列中元素列表以及队列的最大容量。

下面是 Lock 和 BoundedSemaphore 中的两个代码片段。

```
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

访问条件决定调用 test-and-set-state 方法的线程是否可以对状态进行设置。访问条件一般是基于同步器状态的。通常是放在一个 while 循环里，以避免虚假唤醒问题。访问条件的计算结果要么是 true 要么是 false。

Lock 实际上有两个访问条件。如果某个线程想“获取”许可，将检查 signals 变量是否达到上限；如果某个线程想“释放”许可，将检查 signals 变量是否为 0。

这里有两个来自 Lock 和 BoundedSemaphore 的代码片段，它们都有访问条件。注意观察条件是怎样在 while 循环中检查的。

```
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();
```

```
  }
}
```

状态变化

一旦一个线程获得了临界区的访问权限，它得改变同步器的状态，让其它线程阻塞，防止它们进入临界区。换而言之，这个状态表示正有一个线程在执行临界区的代码。其它线程想要访问临界区的时候，该状态应该影响到访问条件的结果。

在 Lock 中，通过代码设置 isLocked = true 来改变状态，在信号量中，改变状态的是 signals - 或 signals++;

这里有两个状态变化的代码片段：

```
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

一旦某个线程改变了同步器的状态，可能需要通知其它等待的线程状态已经变了。因为也许这个状态的变化会让其它线程的访问条件变为 true。

通知策略通常分为三种：

1. 通知所有等待的线程
2. 通知 N 个等待线程中的任意一个
3. 通知 N 个等待线程中的某个指定的线程

通知所有等待的线程非常简单。所有等待的线程都调用的同一个对象上的 wait()方法，某个线程想要通知它们只需在这个对象上调用 notifyAll()方法。

通知等待线程中的任意一个也很简单，只需将 notifyAll()调用换成 notify()即可。调用 notify 方法没办法确定唤醒的是哪一个线程，也就是“等待线程中的任意一个”。

有时候可能需要通知指定的线程而非任意一个等待的线程。例如，如果你想保证线程被通知的顺序与它们进入同步块的顺序一致，或按某种优先级的顺序来通知。想要实现这种需求，每个等待的线程必须在其自有的对象上调用 wait()。当通知线程想要通知某个特定的等待线程时，调用该线程自有对象的 notify()方法即可。饥饿和公平中有这样的例子。

下面是通知策略的一个例子（通知任意一个等待线程）：

```
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
```

```
  }
}
```

Test-and-Set 方法

同步器中最常见的有两种类型的方法，test-and-set 是第一种（set 是另一种）。Test-and-set 的意思是，调用这个方法的线程检查访问条件，如若满足，该线程设置同步器的内部状态来表示它已经获得了访问权限。

状态的改变通常使其它试图获取访问权限的线程计算条件状态时得到 false 的结果，但并不一定总是如此。例如，在读写锁中，获取读锁的线程会更新读写锁的状态来表示它获取到了读锁，但是，只要没有线程请求写锁，其它请求读锁的线程也能成功。

test-and-set 很有必要是原子的，也就是说在某个线程检查和设置状态期间，不允许有其它线程在 test-and-set 方法中执行。

test-and-set 方法的程序流通常遵照下面的顺序：

1. 如有必要，在检查前先设置状态
2. 检查访问条件
3. 如果访问条件不满足，则等待
4. 如果访问条件满足，设置状态，如有必要还要通知等待线程

下面的 ReadWriteLock 类的 lockWrite()方法展示了 test-and-set 方法。调用 lockWrite()的线程在检查之前先设置状态(writeRequests++)。然后检查 canGrantWriteAccess()中的访问条件，如果检查通过，在退出方法之前再次设置内部状态。这个方法中没有去通知等待线程。

```
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;
    }

    ...
}
```

下面的 BoundedSemaphore 类有两个 test-and-set 方法：take()和 release()。两个方法都有检查和设置内部状态。

```
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();
  }
}
```

set 方法

set 方法是同步器中常见的第二种方法。set 方法仅是设置同步器的内部状态，而不先做检查。set 方法的一个典型例子是 Lock 类中的 unlock()方法。持有锁的某个线程总是能够成功解锁，而不需要检查该锁是否处于解锁状态。

set 方法的程序流通常如下：

1. 设置内部状态
2. 通知等待的线程

这里是 unlock()方法的一个例子：

```
public class Lock{
  private boolean isLocked = false;

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

25

无阻塞算法

在并发上下文中，非阻塞算法是一种允许线程在阻塞其他线程的情况下访问共享状态的算法。在绝大多数项目中，在算法中如果一个线程的挂起没有导致其它的线程挂起，我们就说这个算法是非阻塞的。

为了更好的理解阻塞算法和非阻塞算法之间的区别，我会先讲解阻塞算法然后再讲解非阻塞算法。

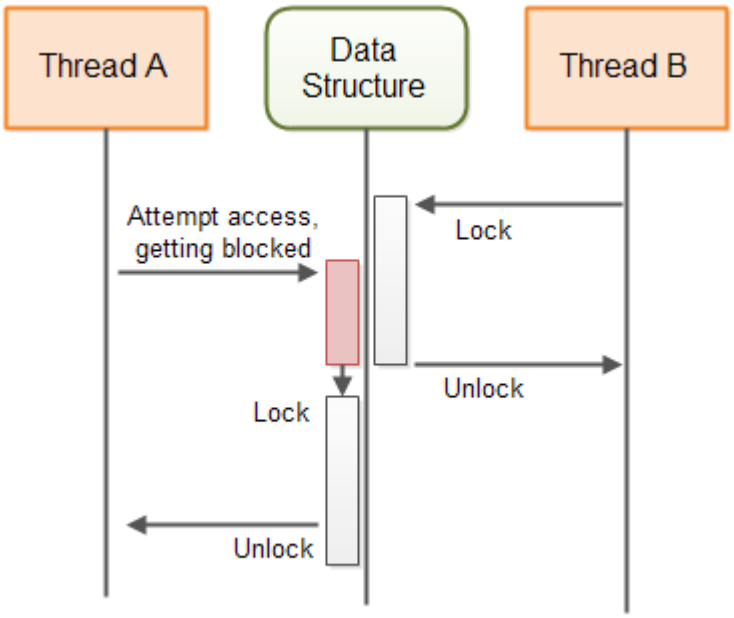
阻塞并发算法

一个阻塞并发算法一般分下面两步：

- **执行线程请求的操作**
- **阻塞线程直到可以安全地执行操作**

很多算法和并发数据结构都是阻塞的。例如，java.util.concurrent.BlockingQueue 的不同实现都是阻塞数据结构。如果一个线程要往一个阻塞队列中插入一个元素，队列中没有足够的空间，执行插入操作的线程就会阻塞直到队列中有了可以存放插入元素的空间。

下图演示了一个阻塞算法保证一个共享数据结构的行为：



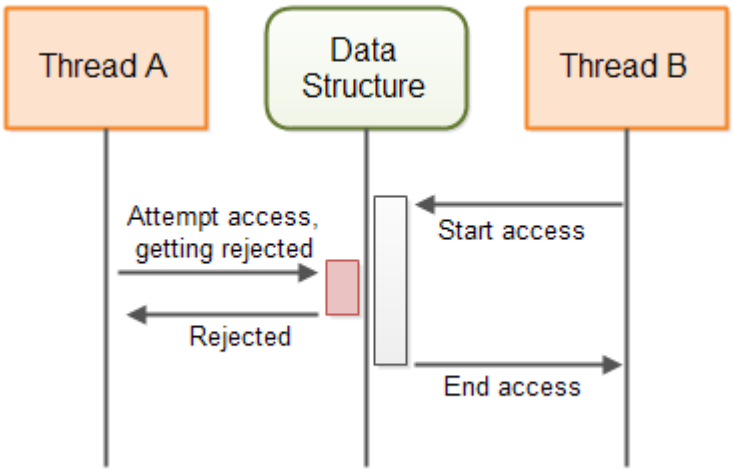
非阻塞并发算法

一个非阻塞并发算法一般包含下面两步：

- **执行线程请求的操作**
- **通知请求线程操作不能被执行**

Java 也包含几个非阻塞数据结构。AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference 都是非阻塞数据结构的例子。

下图演示了一个非阻塞算法保证一个共享数据结构的行为：



非阻塞算法 vs 阻塞算法

阻塞算法和非阻塞算法的主要不同在于上面两部分描述的它们的行为的第二步。换句话说，它们之间的不同在于当请求操作不能够执行时阻塞算法和非阻塞算法会怎么做。

阻塞算法会阻塞线程知道请求操作可以被执行。非阻塞算法会通知请求线程操作不能够被执行，并返回。

一个使用了阻塞算法的线程可能会阻塞直到有可能去处理请求。通常，其它线程的动作使第一个线程执行请求的动作成为了可能。 如果，由于某些原因线程被阻塞在程序某处，因此不能让第一个线程的请求动作被执行，第一个线程会阻塞——可能一直阻塞或者直到其他线程执行完必要的动作。

例如，如果一个线程产生往一个已经满了的阻塞队列里插入一个元素，这个线程就会阻塞，直到其他线程从这个阻塞队列中取走了一些元素。如果由于某些原因，从阻塞队列中取元素的线程假定被阻塞在了程序的某处，那么，尝试往阻塞队列中添加新元素的线程就会阻塞，要么一直阻塞下去，要么知道从阻塞队列中取元素的线程最终从阻塞队列中取走了一个元素。

非阻塞并发数据结构

在一个多线程系统中，线程间通常通过一些数据结构”交流“。例如可以是任何的数据结构，从变量到更高级的俄数据结构（队列，栈等）。为了确保正确，并发线程在访问这些数据结构的时候，这些数据结构必须由一些并发算法来保证。这些并发算法让这些数据结构成为**并发数据结构**。

如果某个算法确保一个并发数据结构是阻塞的，它就被称为是一个**阻塞算法**。这个数据结构也被称为是一个**阻塞，并发数据结构**。

如果某个算法确保一个并发数据结构是非阻塞的，它就被称为是一个**非阻塞算法**。这个数据结构也被称为是一个**非阻塞，并发数据结构**。

每个并发数据结构被设计用来支持一个特定的通信方法。使用哪种并发数据结构取决于你的通信需要。在接下里的部分，我会引入一些非阻塞并发数据结构，并讲解它们各自的适用场景。通过这些并发数据结构工作原理的讲解应该能在非阻塞数据结构的设计和实现上一些启发。

Volatile 变量

Java 中的 volatile 变量是直接从主存中读取值的变量。当一个新的值赋给一个 volatile 变量时，这个值总是会被立即写回到主存中去。这样就确保了，一个 volatile 变量最新的值总是对跑在其他 CPU 上的线程可见。其他线程每次会从主存中读取变量的值，而不是比如线程所运行 CPU 的 CPU 缓存中。

colatile 变量是非阻塞的。修改一个 volatile 变量的值是一耳光原子操作。它不能够被中断。不过，在一个 volatile 变量上的一个 read-update-write 顺序的操作不是原子的。因此，下面的代码如果由多个线程执行可能导致竞态条件。

```
volatile myVar = 0;
...
int temp = myVar;
temp++;
myVar = temp;
```

首先，myVar 这个 volatile 变量的值被从主存中读出来赋给了 temp 变量。然后，temp 变量自增 1。然后，temp 变量的值又赋给了 myVar 这个 volatile 变量这意味着它会被写回到主存中。

如果两个线程执行这段代码，然后它们都读取 myVar 的值，加 1 后，把它的值写回到主存。这样就存在 myVar 仅被加 1，而没有被加 2 的风险。

你可能认为你不会写像上面这样的代码，但是在实践中上面的代码等同于如下的代码：

```
myVar++;
```

执行上面的代码时，myVar 的值读到一个 CPU 寄存器或者一个本地 CPU 缓存中，myVar 加 1，然后这个 CPU 寄存器或者 CPU 缓存中的值被写回到主存中。

单个写线程的情景

在一些场景下，你仅有一个线程在向一个共享变量写，多个线程在读这个变量。当仅有一个线程在更新一个变量，不管有多少个线程在读这个变量，都不会发生竞态条件。因此，无论时候当仅有一个线程在写一个共享变量时，你可以把这个变量声明为 volatile。

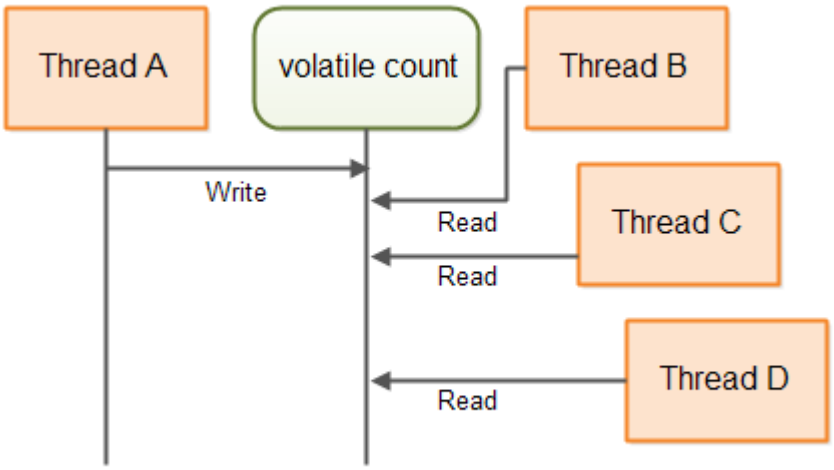
当多个线程在一个共享变量上执行一个 read-update-write 的顺序操作时才会发生竞态条件。如果你只有一个线程在执行一个 raed-update-write 的顺序操作，其他线程都在执行读操作，将不会发生竞态条件。

下面是一个单个写线程的例子，它没有采取同步手段但任然是并发的。

```
public class SingleWriterCounter{
   private volatile long count = 0;

   /**
    *Only one thread may ever call this method
    *or it will lead to race conditions
    */
    public void inc(){
       this.count++;
    }

    /**
     *Many reading threads may call this method
     *@return
     */
     public long count(){
        return this.count;
     }
}
```

多个线程访问同一个 Counter 实例，只要仅有一个线程调用 inc()方法，这里，我不是说在某一时刻一个线程，我的意思是，仅有相同的，单个的线程被允许去调用 inc()>方法。多个线程可以调用 count()方法。这样的场景将不会发生任何竞态条件。

下图，说明了线程是如何访问 count 这个 volatile 变量的。


Thread A
volatile count
Thread B
Write
Read
Thread C
Read
Thread D
Read

基于 volatile 变量更高级的数据结构

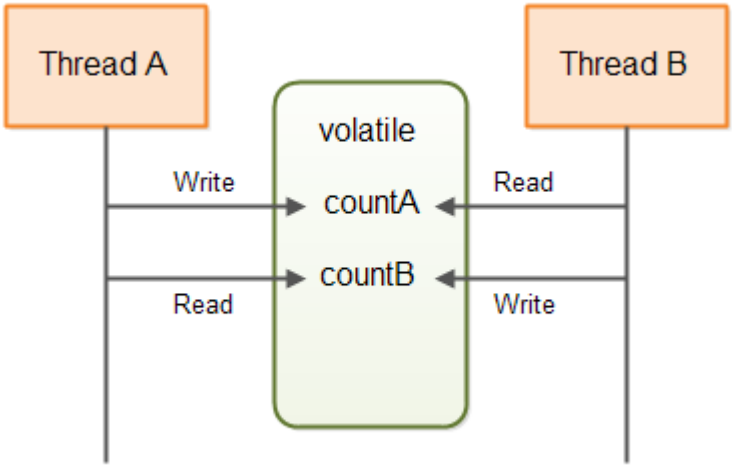
使用多个 volatile 变量去创建数据结构是可以的，构建出的数据结构中每一个 volatile 变量仅被一个单个的线程写，被多个线程读。每个 volatile 变量可能被一个不同的线程写（但仅有一个）。使用像这样的数据结构多个线程可以使用这些 volatile 变量以一个非阻塞的方法彼此发送信息。

下面是一个简单的例子：

```
public class DoubleWriterCounter{
    private volatile long countA = 0;
    private volatile long countB = 0;

    /**
     *Only one (and the same from thereon) thread may ever call this method,
     *or it will lead to race conditions.
     */
     public void incA(){
        this.countA++;
     }

     /**
      *Only one (and the same from thereon) thread may ever call this method,
      *or it will  lead to race conditions.
      */
      public void incB(){
         this.countB++;
      }

      /**
       *Many reading threads may call this method
       */
      public long countA(){
         return this.countA;
      }

     /**
      *Many reading threads may call this method
      */
      public long countB(){
         return this.countB;
      }
}
```

如你所见，DoubleWriterCoounter 现在包含两个 volatile 变量以及两对自增和读方法。在某一时刻，仅有一个单个的线程可以调用 inc()，仅有一个单个的线程可以访问 incB()。不过不同的线程可以同时调用 incA()和 incB()。countA()和 countB()可以被多个线程调用。这将不会引发竞态条件。

DoubleWriterCoounter 可以被用来比如线程间通信。countA 和 countB 可以分别用来存储生产的任务数和消费的任务数。下图，展示了两个线程通过类似于上面的一个数据结构进行通信的。



聪明的读者应该已经意识到使用两个 SingleWriterCounter 可以达到使用 DoubleWriterCoounter 的效果。如果需要，你甚至可以使用多个线程和 SingleWriterCounter 实例。

使用 CAS 的乐观锁

> 如果你确实需要多个线程区写同一个共享变量，volatile 变量是不合适的。你将会需要一些类型的排它锁（悲观锁）访问这个变量。下面代码演示了使用 Java 中的同步块进行排他访问的。

```
public class SynchronizedCounter{
   long count = 0;

   public void inc(){
      synchronized(this){
         count++;
      }
   }

   public long count(){
      synchronized(this){
         return this.count;
      }
   }
}
```

注意，inc()和 count()方法都包含一个同步块。这也是我们像避免的东西——同步块和 wait()-notify 调用等。

我们可以使用一种 Java 的原子变量来代替这两个同步块。在这个例子是 AtomicLong。下面是 Synchronized Counter 类的 AtomicLong 实现版本。

```
import java.util.concurrent.atomic.AtomicLong;

public class AtomicLong{
   private AtomicLong count = new AtomicLong(0);

   public void inc(){
      boolean updated = false;
      while(!updated){
         long prevCount = this.count.get();
         updated = this.count.compareAndSet(prevCount, prevCount + 1);
      }
   }

   public long count(){
      return this.count.get();
   }
}
```

> 这个版本仅仅是上一个版本的线程安全版本。这一版我们感兴趣的是 inc()方法的实现。inc()方法中不再含有一个同步块。而是被下面这些代码替代：

```
boolean updated = false;
while(!updated){
    long prevCount = this.count.get();
    updated = this.count.compareAndSet(prevCount, prevCount + 1);
}
```

上面这些代码并不是一个原子操作。也就是说，对于两个不同的线程去调用 inc()方法，然后执行 long prevCount = this.count.get()语句，因此获得了这个计数器的上一个 count。但是，上面的代码并没有包含任何的竞态条件。

秘密就在于 while 循环里的第二行代码。compareAndSet()方法调用是一个原子操作。它用一个期望值和 AtomicLong 内部的值去比较，如果这两个值相等，就把 AtomicLong 内部值替换为一个新值。compareAndSet()通常被 CPU 中的 compare-and-swap 指令直接支持。因此，不需要去同步，也不需要去挂起线程。

假设，这个 AtomicLong 的内部值是 20。然后，两个线程去读这个值，都尝试调用 compareAndSet(20, 20 + 1)。尽管 compareAndSet()是一个原子操作，这个方法也会被这两个线程相继执行（某一个时刻只有一个）。

第一个线程会使用期望值 20（这个计数器的上一个值）与 AtomicLong 的内部值进行比较。由于两个值是相等的，AtomicLong 会更新它的内部值至 21（20 + 1 ）。变量 updated 被修改为 true，while 循环结束。

现在，第二个线程调用 compareAndSet(20, 20 + 1)。由于 AtomicLong 的内部值不再是 20，这个调用将不会成功。AtomicLong 的值不会再被修改为 21。变量，updated 被修改为 false，线程将会再次在 while 循环外自旋。这段时间，它会读到值 21 并企图把值更新为 22。如果在此期间没有其它线程调用 inc()。第二次迭代将会成功更新 AtomicLong 的内部值到 22。

为什么称它为乐观锁

上一部分展现的代码被称为**乐观锁**（optimistic locking）。乐观锁区别于传统的锁，有时也被称为**悲观锁**。传统的锁会使用同步块或其他类型的锁阻塞对临界区域的访问。一个同步块或锁可能会导致线程挂起。

乐观锁允许所有的线程在不发生阻塞的情况下创建一份共享内存的拷贝。这些线程接下来可能会对它们的拷贝进行修改，并企图把它们修改后的版本写回到共享内存中。如果没有其它线程对共享内存做任何修改， CAS 操作就允许线程将它的变化写回到共享内存中去。如果，另一个线程已经修改了共享内存，这个线程将不得不再次获得一个新的拷贝，在新的拷贝上做出修改，并尝试再次把它们写回到共享内存中去。

称之为“乐观锁”的原因就是，线程获得它们想修改的数据的拷贝并做出修改，在乐观的假在此期间没有线程对共享内存做出修改的情况下。当这个乐观假设成立时，这个线程仅仅在无锁的情况下完成共享内存的更新。当这个假设不成立时，线程所做的工作就会被丢弃，但任然不使用锁。

乐观锁使用于共享内存竞用不是非常高的情况。如果共享内存上的内容非常多，仅仅因为更新共享内存失败，就用浪费大量的 CPU 周期用在拷贝和修改上。但是，如果砸共享内存上有大量的内容，无论如何，你都要把你的代码设计的产生的争用更低。

乐观锁是非阻塞的

我们这里提到的乐观锁机制是非阻塞的。如果一个线程获得了一份共享内存的拷贝，当尝试修改时，发生了阻塞，其它线程去访问这块内存区域不会发生阻塞。

对于一个传统的加锁/解锁模式，当一个线程持有一个锁时，其它所有的线程都会一直阻塞直到持有锁的线程再次释放掉这个锁。如果持有锁的这个线程被阻塞在某处，这个锁将很长一段时间不能被释放，甚至可能一直不能被释放。

不可替换的数据结构

简单的 CAS 乐观锁可以用于共享数据结果，这样一来，整个数据结构都可以通过一个单个的 CAS 操作被替换成为一个新的数据结构。尽管，使用一个修改后的拷贝来替换真个数据结构并不总是可行的。

假设，这个共享数据结构是队列。每当线程尝试从向队列中插入或从队列中取出元素时，都必须拷贝这个队列然后在拷贝上做出期望的修改。我们可以通过使用一个 AtomicReference 来达到同样的目的。拷贝引用，拷贝和修改队列，尝试替换在 AtomicReference 中的引用让它指向新创建的队列。

然而，一个大的数据结构可能会需要大量的内存和 CPU 周期来复制。这会使你的程序占用大量的内存和浪费大量的时间再拷贝操作上。这将会降低你的程序的性能，特别是这个数据结构的竞用非常高情况下。更进一步说，一个线程花费在拷贝和修改这个数据结构上的时间越长，其它线程在此期间修改这个数据结构的可能性就越大。如你所知，如果另一个线程修改了这个数据结构在它被拷贝后，其它所有的线程都不等不再次执行 拷贝–修改 操作。这将会增大性能影响和内存浪费，甚至更多。

接下来的部分将会讲解一种实现非阻塞数据结构的方法，这种数据结构可以被并发修改，而不仅仅是拷贝和修改。

共享预期的修改

用来替换拷贝和修改整个数据结构，一个线程可以共享它们对共享数据结构预期的修改。一个线程向对修改某个数据结构的过程变成了下面这样：

- **检查是否另一个线程已经提交了对这个数据结构提交了修改**
- **如果没有其他线程提交了一个预期的修改，创建一个预期的修改，然后向这个数据结构提交预期的修**
- **执行对共享数据结构的修改**
- **移除对这个预期的修改的引用，向其它线程发送信号，告诉它们这个预期的修改已经被执行**

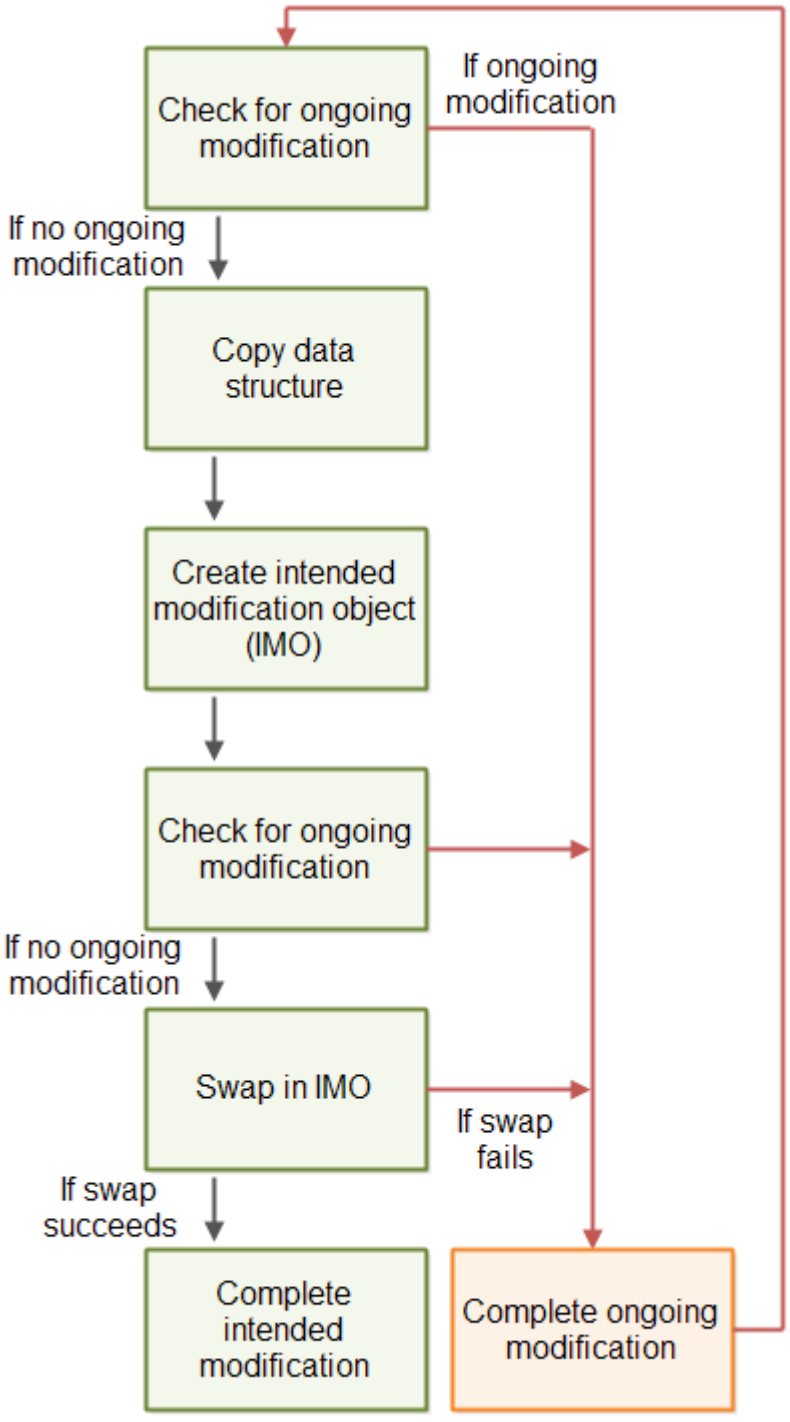
如你所见，第二步可以阻塞其他线程提交一个预期的修改。因此，第二步实际的工作是作为这个数据结构的一个锁。如果一个线程已经成功提交了一个预期的修改，其他线程就不可以再提交一个预期的修改直到第一个预期的修改执行完毕。

如果一个线程提交了一个预期的修改，然后做一些其它的工作时发生阻塞，这时候，这个共享数据结构实际上是被锁住的。其它线程可以检测到它们不能够提交一个预期的修改，然后回去做一些其它的事情。很明显，我们需要解决这个问题。

可完成的预期修改

为了避免一个已经提交的预期修改可以锁住共享数据结构，一个已经提交的预期修改必须包含足够的信息让其他线程来完成这次修改。因此，如果一个提交了预期修改的线程从未完成这次修改，其他线程可以在它的支持下完成这次修改，保证这个共享数据结构对其他线程可用。

下图说明了上面描述的非阻塞算法的蓝图：



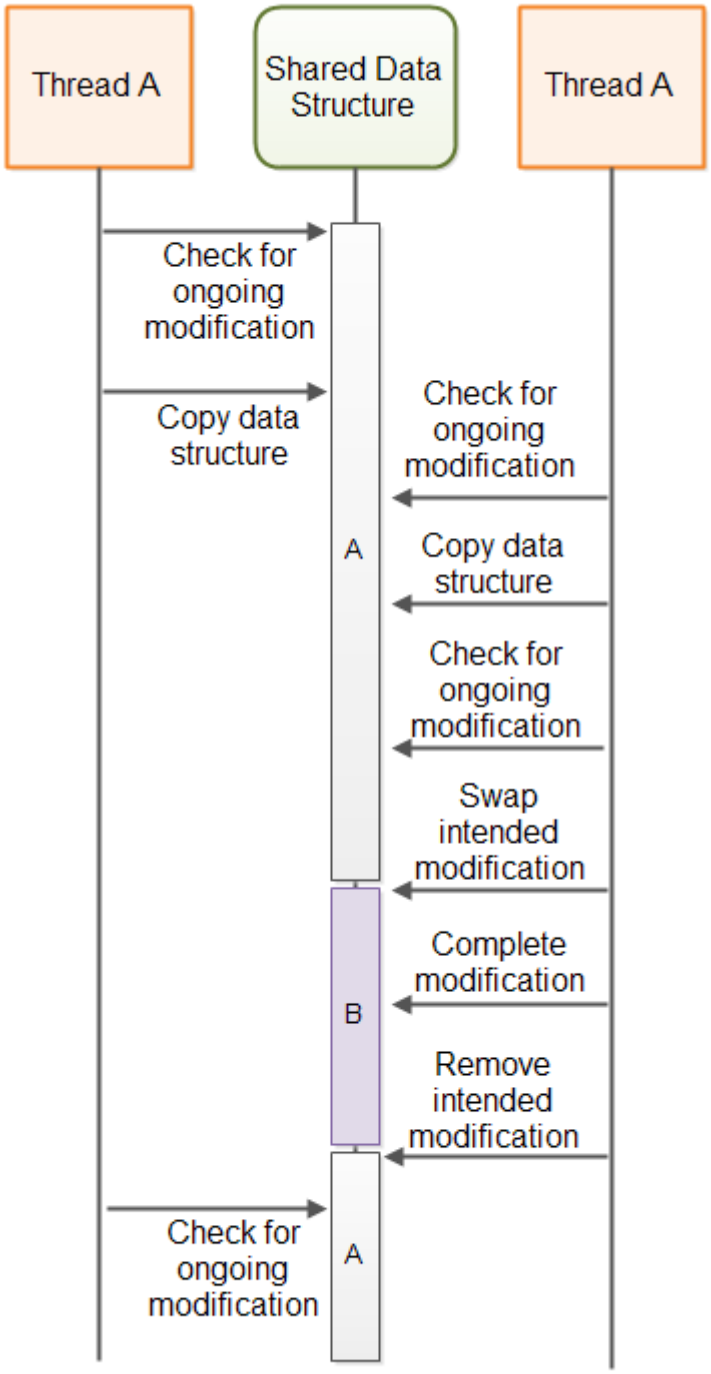
修改必须被当做一个或多个 CAS 操作来执行。因此，如果两个线程尝试去完成同一个预期修改，仅有一个线程可以所有的 CAS 操作。一旦一条 CAS 操作完成后，再次企图完成这个 CAS 操作都不会“得逞”。

A–B–A 问题

上面演示的算法可以称之为 **A–B–A 问题**。A–B–A 问题指的是一个变量被从 A 修改到了 B，然后又被修改回 A 的一种情景。其他线程对于这种情况却一无所知。

如果线程 A 检查正在进行的数据更新，拷贝，被线程调度器挂起，一个线程 B 在此期可能可以访问这个共享数据结构。如果线程对这个数据结构执行了全部的更新，移除了它的预期修改，这样看起来，好像线程 A 自从拷贝了这个数据结构以来没有对它做任何的修改。然而，一个修改确实已经发生了。当线程 A 继续基于现在已经过期的数据拷贝执行它的更新时，这个数据修改已经被线程 B 的修改破坏。

下图说明了上面提到的 A–B–A 问题：


Thread A
Shared Data Structure
Thread A
Check for ongoing modification
Copy data structure
Check for ongoing modification
A
Copy data structure
Check for ongoing modification
Swap intended modification
B
Complete modification
Remove intended modification
Check for ongoing modification
A

A-B-A 问题的解决方案

A-B-A 通常的解决方法就是不再仅仅替换指向一个预期修改对象的指针，而是指针结合一个计数器，然后使用一个单个的 CAS 操作来替换指针 + 计数器。这在支持指针的语言像 C 和 C++中是可行的。因此，尽管当前修改指针被设置回指向 “不是正在进行的修改”（no ongoing modification），指针 + 计数器的计数器部分将会被自增，使修改对其它线程是可见的。

在 Java 中，你不能将一个引用和一个计数器归并在一起形成一个单个的变量。不过 Java 提供了 AtomicStampedReference 类，利用这个类可以使用一个 CAS 操作自动的替换一个引用和一个标记（stamp）。

一个非阻塞算法模板

下面的代码意在在如何实现非阻塞算法上一些启发。这个模板基于这篇教程所讲的东西。

注意：在非阻塞算法方面，我并不是一位专家，所以，下面的模板可能错误。不要基于我提供的模板实现自己的非阻塞算法。这个模板意在给你一个关于非阻塞算法大致是什么样子的一个 idea。如果，你想实现自己的非阻塞算法，首先学习一些实际的工业水平的非阻塞算法的时间，在实践中学习更多关于非阻塞算法实现的知识。

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicStampedReference;

public class NonblockingTemplate{
    public static class IntendedModification{
        public AtomicBoolean completed = new AtomicBoolean(false);
    }

    private AtomicStampedReference<IntendedModification> ongoinMod = new AtomicStampedReference<IntendedModif
    //declare the state of the data structure here.

    public void modify(){
        while(!attemptModifyASR());
    }


    public boolean attemptModifyASR(){
        boolean modified = false;

        IntendedMOdification currentlyOngoingMod = ongoingMod.getReference();
        int stamp = ongoingMod.getStamp();

        if(currentlyOngoingMod == null){
            //copy data structure - for use
            //in intended modification

            //prepare intended modification
            IntendedModification newMod = new IntendModification();

            boolean modSubmitted = ongoingMod.compareAndSet(null, newMod, stamp, stamp + 1);

            if(modSubmitted){
                //complete modification via a series of compare-and-swap operations.
                //note: other threads may assist in completing the compare-and-swap
```

```
            // operations, so some CAS may fail
            modified = true;
          }
      }else{
          //attempt to complete ongoing modification, so the data structure is freed up
          //to allow access from this thread.
          modified = false;
      }

      return modified;
   }
}
```

非阻塞算法是不容易实现的

正确的设计和实现非阻塞算法是不容易的。在尝试设计你的非阻塞算法之前，看一看是否已经有人设计了一种非阻塞算法正满足你的需求。

Java 已经提供了一些非阻塞实现（比如 ConcurrentLinkedQueue），相信在 Java 未来的版本中会带来更多的非阻塞算法的实现。

除了 Java 内置非阻塞数据结构还有很多开源的非阻塞数据结构可以使用。例如，LAMX Disrupter 和 Cliff Click 实现的非阻塞 HashMap。查看我的 Java concurrency references page 查看更多的资源。

使用非阻塞算法的好处

非阻塞算法和阻塞算法相比有几个好处。下面让我们分别看一下：

选择

非阻塞算法的第一个好处是，给了线程一个选择当它们请求的动作不能够被执行时做些什么。不再是被阻塞在那，请求线程关于做什么有了一个选择。有时候，一个线程什么也不能做。在这种情况下，它可以选择阻塞或自我等待，像这样把 CPU 的使用权让给其它的任务。不过至少给了请求线程一个选择的机会。

在一个单个的 CPU 系统可能会挂起一个不能执行请求动作的线程，这样可以让其它线程获得 CPU 的使用权。不过即使在一个单个的 CPU 系统阻塞可能导致死锁，线程饥饿等并发问题。

没有死锁

非阻塞算法的第二个好处是，一个线程的挂起不能导致其它线程挂起。这也意味着不会发生死锁。两个线程不能互相彼此等待来获得被对方持有的锁。因为线程不会阻塞当它们不能执行它们的请求动作时，它们不能阻塞互相等待。非阻塞算法任然可能产生活锁（live lock），两个线程一直请求一些动作，但一直被告知不能够被执行（因为其他线程的动作）。

没有线程挂起

挂起和恢复一个线程的代价是昂贵的。没错，随着时间的推移，操作系统和线程库已经越来越高效，线程挂起和恢复的成本也不断降低。不过，线程的挂起和户对任然需要付出很高的代价。

无论什么时候，一个线程阻塞，就会被挂起。因此，引起了线程挂起和恢复过载。由于使用非阻塞算法线程不会被挂起，这种过载就不会发生。这就意味着 CPU 有可能花更多时间在执行实际的业务逻辑上而不是上下文切换。

在一个多个 CPU 的系统上，阻塞算法会对阻塞算法产生重要的影响。运行在 CPUA 上的一个线程阻塞等待运行在 CPU B 上的一个线程。这就降低了程序天生就具备的并行水平。当然，CPU A 可以调度其他线程去运行，但是挂起和激活线程（上下文切换）的代价是昂贵的。需要挂起的线程越少越好。

降低线程延迟

在这里我们提到的延迟指的是一个请求产生到线程实际的执行它之间的时间。因为在非阻塞算法中线程不会被挂起，它们就不需要付昂贵的，缓慢的线程激活成本。这就意味着当一个请求执行时可以得到更快的响应，减少它们的响应延迟。

非阻塞算法通常忙等待直到请求动作可以被执行来降低延迟。当然，在一个非阻塞数据数据结构有着很高的线程争用的系统中，CPU 可能在它们忙等待期间停止消耗大量的 CPU 周期。这一点需要牢牢记住。非阻塞算法可能不是最好的选择如果你的数据结构哦有着很高的线程争用。不过，也常常存在通过重构你的程序来达到更低的线程争用。

26 阿姆达尔定律

阿姆达尔定律可以用来计算处理器平行运算之后效率提升的能力。阿姆达尔定律因 Gene Amdal 在 1967 年提出这个定律而得名。绝大多数使用并行或并发系统的开发者有一种并发或并行可能会带来提速的感觉，甚至不知道阿姆达尔定律。不管怎样，了解阿姆达尔定律还是有用的。

我会首先以算术的方式介绍阿姆达尔定律定律，然后再用图表演示一下。

阿姆达尔定律定义

一个程序（或者一个算法）可以按照是否可以被并行化分为下面两个部分：

- **可以被并行化的部分**
- **不可以被并行化的部分**

假设一个程序处理磁盘上的文件。这个程序的一小部分用来扫描路径和在内存中创建文件目录。做完这些后，每个文件交个一个单独的线程去处理。扫描路径和创建文件目录的部分不可以被并行化，不过处理文件的过程可以。

程序串行（非并行）执行的总时间我们记为 T。时间 T 包括不可以被并行和可以被并行部分的时间。不可以被并行的部分我们记为 B。那么可以被并行的部分就是 T−B。下面的列表总结了这些定义：

- **T = 串行执行的总时间**
- **B = 不可以并行的总时间**
- **T− B = 并行部分的总时间**

从上面可以得出：

T = B + (T − B)

首先，这个看起来可能有一点奇怪，程序的可并行部分在上面这个公式中并没有自己的标识。然而，由于这个公式中可并行可以用总时间 T 和 B（不可并行部分）表示出来，这个公式实际上已经从概念上得到了简化，也即是指以这种方式减少了变量的个数。

T− B 是可并行化的部分，以并行的方式执行可以提高程序的运行速度。可以提速多少取决于有多少线程或者多少个 CPU 来执行。线程或者 CPU 的个数我们记为 N。可并行化部分被执行的最快时间可以通过下面的公式计算出来：

(T − B) / N

或者通过这种方式

(1 / N) * (T − B)

维基中使用的是第二种方式。

根据阿姆达尔定律，当一个程序的可并行部分使用 N 个线程或 CPU 执行时，执行的总时间为：

$$T(N) = B + (T - B) / N$$

T(N)指的是在并行因子为 N 时的总执行时间。因此，T(1)就执行在并行因子为 1 时程序的总执行时间。使用 T(1)代替 T，阿姆达尔定律定律看起来像这样：

$$T(N) = B + (T(1) - B) / N$$

表达的意思都是是一样的。

一个计算例子

为了更好的理解阿姆达尔定律，让我们来看一个计算的例子。执行一个程序的总时间设为 1。程序的不可并行化占 40%，按总时间 1 计算，就是 0.4，可并行部分就是 1 - 0.4=0.6.

在并行因子为 2 的情况下，程序的执行时间将会是：

```
T(2) = 0.4 + ( 1 − 0.4 ) / 2
 = 0.4 + 0.6 / 2
 = 0.4 + 0.3
 = 0.7
```

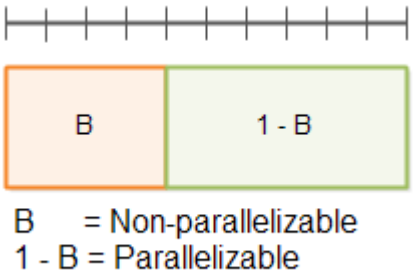
在并行因子为 5 的情况下，程序的执行时间将会是：

```
T(5) = 0.4 + ( 1 − 0.4 ) / 5
 = 0.4 + 0.6 / 6
 = 0.4 + 0.12
 = 0.52
```

阿姆达尔定律图示

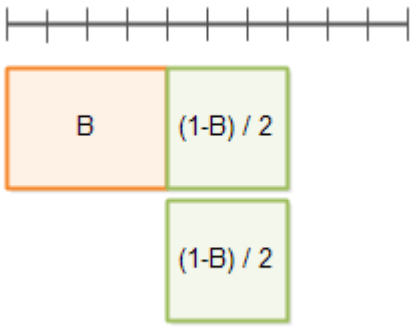
为了更好地理解阿姆达尔定律，我会尝试演示这个定定律是如何诞生的。

首先，一个程序可以被分割为两部分，一部分为不可并行部分 B，一部分为可并行部分 1 - B。如下图：

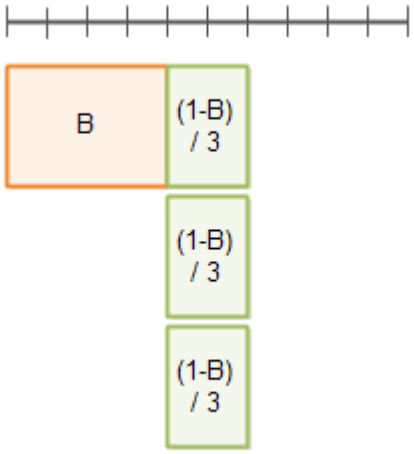


在顶部被带有分割线的那条直线代表总时间 T(1)。

下面你可以看到在并行因子为 2 的情况下的执行时间：



并行因子为 3 的情况：



优化算法

从阿姆达尔定律可以看出，程序的可并行化部分可以通过使用更多的硬件（更多的线程或 CPU）运行更快。对于不可并行化的部分，只能通过优化代码来达到提速的目的。因此，你可以通过优化不可并行化部分来提高你的程序的运行速度和并行能力。你可以对不可并行化在算法上做一点改动，如果有可能，你也可以把一些移到可并行化放的部分。

优化串行分量

如果你优化一个程序的串行化部分，你也可以使用阿姆达尔定律来计算程序优化后的执行时间。如果不可并行部分通过一个因子 O 来优化，那么阿姆达尔定律看起来就像这样：

```
T(O, N) = B / O + (1 − B / O) / N
```

记住，现在程序的不可并行化部分占了 B / O 的时间，所以，可并行化部分就占了 1 − B / O 的时间。

如果 B 为 0.1，O 为 2，N 为 5，计算看起来就像这样：

```
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
```

运行时间 vs. 加速

到目前为止，我们只用阿姆达尔定律计算了一个程序或算法在优化后或者并行化后的执行时间。我们也可以使用阿姆达尔定律计算加速比（speedup），也就是经过优化后或者串行化后的程序或算法比原来快了多少。

如果旧版本的程序或算法的执行时间为 T，那么增速比就是：

```
Speedup = T / T(O , N);
```

为了计算执行时间，我们常常把 T 设为 1，加速比为原来时间的一个分数。公式大致像下面这样：

```
Speedup = 1 / T ( O,N)
```

如果我们使用阿姆达尔定律来代替 T(O,N)，我们可以得到下面的公式：

```
Speedup = 1 / ( B / O + (1 − B / O) / N)
```

如果 B = 0.4， O = 2， N = 5， 计算变成下面这样：

```
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 ...
```

上面的计算结果可以看出，如果你通过一个因子 2 来优化不可并行化部分，一个因子 5 来并行化可并行化部分，这个程序或算法的最新优化版本最多可以比原来的版本快 2.77777 倍。

测量，不要仅是计算

虽然阿姆达尔定律允许你并行化一个算法的理论加速比，但是不要过度依赖这样的计算。在实际场景中，当你优化或并行化一个算法时，可以有很多的因子可以被考虑进来。

内存的速度，CPU 缓存，磁盘，网卡等可能都是一个限制因子。如果一个算法的最新版本是并行化的，但是导致了大量的 CPU 缓存浪费，你可能不会再使用 xN 个 CPU 来获得 xN 的期望加速。如果你的内存总线（memory bus），磁盘，网卡或者网络连接都处于高负载状态，也是一样的情况。

我们的建议是，使用阿姆达尔定律定律来指导我们优化程序，而不是用来测量优化带来的实际加速比。记住，有时候一个高度串行化的算法胜过一个并行化的算法，因为串行化版本不需要进行协调管理（上下文切换），而且一个单个的 CPU 在底层硬件工作（CPU 管道、CPU 缓存等）上的一致性可能更好。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java-concurrent/