# 什么是线程安全？

线程安全经常会被各种行业大佬或者面试官大佬挂在嘴边，如何找到一个通俗易懂一点的方式解释线程安全呢，伟大的砖家给出了答案：**如果一个对象可以安全的被多个对象使用，那它就是线程安全的**。是的，这种回答非常的巧妙精彩…巧妙到你无法怼它是错误的…但是也无法从中找到任何有用信息…

众多定义中，《Java Concurrency In Practice》的作者Brian Goetz对线程安全有一个比较恰当的定义：

**当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那么这个对象就是线程安全哒~~~**

是不是还有点云里雾里？没关系，我们来具体的分一下，看看Java语言中操作共享数据的5中情况，然后结合这些情况来看Java中的线程安全。

**1. 不可变**

这种方式是最安全最简单粗暴的。但是很不幸也是适用范围最少的。成也不可变，败也不可变。

如何实现不可变，很简单用final关键字修饰的对象就是不可变的，比如String，我们调用它的substring，trim等各种方法但是它原始的值还是不变的，以至于我们不得不用一个新对象去接收返回结果。

**2. 绝对线程安全**

绝对线程安全也就是达到上面那个Brian大佬讲的，在任何情况下随意调用都是线程安全的，他讲的还是很严格的，Java中口口声称自己是线程安全的类，大多数不是绝对线程安全的，比如Vector，具体不做解释，可自行百度，反正sun已经不建议使用Vector了，效率比ArrayList低就不说了，还是一样得做额外的同步。所以，要想绝对线程安全，我们还是得求助synchronized大哥，或者用ReentrantLock。

**3. 相对线程安全**

相对线程安全就是我们通常讲的线程安全，比如Vector，Hashtable, 里面的方法，用了synchronized关键字修饰过，是相对想成安全的。

**4. 线程兼容**

我们一般说一个类或方法不是线程安全的，基本八九不离十也就是这种情况了。比如HashMap，ArrayList等，我们接触的最多，简单粗暴高效，但是只是相对线程安全，也就是无人争抢的时候不会出错，一旦发生争抢还是得做额外的同步措施。

**5. 线程对立**

这是最可怕的一种情况，就是说无论怎么抢救都要往阎王殿跑的八头马拉不回的，根本无法在多线程环境中并发使用的代码。这种排斥多线的代码Java中很少见，但是还是有，比如Thread类的suspend和resume方法，如果连个线程同时持有一个对象，一个尝试终止，另一个尝试去恢复，那么很容易就死锁了，所以伟大的伞公司也废弃了这两个方法。

# 线程安全的实现方法

**1. 互斥同步**

通常用的比较多的就是加入synchronized关键字来修饰需要同步的方法或代码块，以及需要上锁的对象。当然用ReentrantLock也可以达到同样的效果，或者更高级的体验。

**Synchronized关键字和ReentrantLock类：**

在Java多线程同步机制中，ReentrantLock可以达到和synchronized关键字同样的效果，不仅如此，它在扩展功能上也更加强大，比如具有嗅探锁定，多路分支通知等功能，而且在使用上也比synchronized更加灵活。

ReentrantLock支持查询等待队列的状态，支持手动上锁解锁，支持等待/通知模式，支持多对象监视器，另外它的派生类ReentrantReadWriteLock还支持读写锁（即共享锁/排它锁）。

**ReentrantLock有以下三项优势：**

1）等待可中断：当持有的线程长期占着茅坑不拉屎（不释放锁）的时候，正在等待的线程可以放弃等待，改为处理其他事情，可中断特性对处理时间较长的同步块尤为重要。

2）公平锁：指多个线程在等待同一个锁的时候，必须按照申请的时间顺序来依次获得锁（先来先得），而非公平锁则不能保证这一点，synchronized是非公平的，ReentrantLock在默认情况下也是非公平的，需要通过bool类型的构造参数设置。

3）绑定多个条件：指一个ReentrantLock对象可以同时绑定多个Condition对象，而在synchronized中，如果要做到多个对象监视器，必须多加几个synchronized关键字，而ReentrantLock中不需要这样傻，只需要每次都newCondition()方法优雅地解决。

性能优势：以前ReentrantLock在多线程环境下是比synchronized要稳定高效许多的，但是随着JDK版本的不断改进，synchronized也能保持一个很高效稳定的状态，JDK的意思肯定还是想多推广使用synchronized，毕竟如此简单粗暴，因此，如果不需要考虑以上三点ReentrantLock所具备的优势，则优先使用synchronized。

**2. 非阻塞同步**

互斥同步的主要问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也称为：阻塞同步（Blocking Synchronization）。 从处理问题的方式上来说，**互斥同步属于一种悲观的并发策略（悲观锁），即：总是认为只要不进行正确的同步措施（比如加锁）就肯定会出现问题，无论共享数据是否真的会出现竞争，它都要进行加锁**（这里讨论的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁）、**用户态和心态的转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作**。随着指令集的发展，我们有了另一个选择：**基于冲突检测的乐观并发策略（乐观锁），即：新进行操作，如果没有其他线程争用共享数据，那操作就成功了，如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施**（最常见的补偿措施就是不断的重试，知道成功为止），这种乐观的并发策略的很多实现都不需要把线程挂起，因此这种同步操作称为非阻塞同步（Non-Blocking Synchronization）。

再简单总结下悲观锁和乐观锁：

悲观锁（阻塞式同步）：总是任务只要不做同步措施，就肯定会出现问题，无论是否发生争抢共享数据情况。（*总是认为只要不戴tt就一定会怀孕，无论对方是否在安全期*）

乐观锁（非阻塞式同步）：先执行操作，如果没有其他线程争抢共享数据，操作就成功；如果有，采取一定的不就措施，最常见的就是不断重试。（*直接开干，没怀孕就算是爽到了，如果怀孕再去医院补救*）

乐观并发策略的前景是令人心动的，但是它依赖指令集的发展（老的指令集不支持），因为我们需要操作和冲突检测这两个步骤具有原子性，考什么来保证原子性呢，如果此时再使用互斥同步来保证原子性就失去意义了，所以我们只能依赖硬件的发展来完成这件事情。硬件需要保证一个从语义上来讲需要多次操作的行为只需要通过一条指令就能完成，这类指令常用的有：

测试并设置

获取并添加

交换

比较并交换

加载链接/条件存储

有了这些指令的支持，我们才可以做到非阻塞式同步，也就是必要同步的那一块是硬件代替了，硬件的执行速度和软件程序的执行速度谁高谁低非常的明显。

**3. 无同步方案**

保证线程安全不一定是需要同步的。

**可重入代码：**

可以在执行代码的任何时刻中断它，转而去执行另一段代码，在控制权返回后原来的程序不会发生任何错误，因此所以可重入的代码都是线程安全的，但并非所有线程安全的代码都是可重入的。

可重入代码有这些特征：不依赖堆上的数据，用到的状态量都由参数传入，不调用非可重入方法等。

是不是有点懵，举个具体的例子，我们常用的方法，只要它不依赖某些成员变量，或者依赖的这些全局变量是不发生改变的（单例或final），无论任何时候，我们只需要传入相应的参数并且都能得到一致的返回结果，那么这个方法就是可重入的。这也就是为什么我们要谨慎使用全局变量，并且尽量用方法的参数去代替全局变量，因为后者更加的线程安全。

**线程本地存储：**

ThreadLocal在前面的《线程间的通信》博客里面也讲到了，它就是维护线程内的全局变量的一个对象，这样根本不会发生多个线程争抢的情况，因为它只属于某一个线程。

线程本地存储并不止这一个案例，现在很多流行的Web框架也屡见不鲜，比如Spring MVC中，会将每一个请求都创建一个线程，将请求的数据放在这个线程中。

# 锁优化

为了在线程之间更高效地共享数据，JDK大佬们尽心竭力的进行了很多优化，高效并发是JDK1.5 到JDK1.6的一个很重要的改进。

**1. 自旋锁与自适应自旋锁**

前面我们提到了，互斥同步对性能的最大影响是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性能带来了很大的压力。虚拟机的大佬们注意到，许多共享数据的锁定状态只会持续很短的时间，在这么短的时间去挂起和恢复线程有点大炮打蚊子的赶脚，于是乎，他们想到一个办法 – 让后面的线程稍等一下，状态依旧是Runable态，不用发生改变，等这个线程执行完，也不用恢复了直接就可以执行。读到这你肯定会说，这么小儿科我也能想到，看吧，你离大神不远矣！是的，就是这么随意，接下来，JDK大佬要解决的就是怎么让后面的线程喝喝茶“稍等一会”的问题了？于是，**自旋锁就出来了，从字面也很好理解---线程自己进入一个循环自己空转**。当然也不是瞎转悠，JDK大佬给他们设置了一个默认的次数：10次（你可以修改JVM参数调整这个数字），也就是10次over了，它就又跳出来争抢共享数据，如果发现那个占着茅坑的线程走了，它就蹲过来了，如果发现别人还没走，就只能乖乖挂起来了（进入等待队列）。

JDK1.6中又引入了**自适应自旋锁，自适应就意味着自旋的时间/次数不在固定，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来动态决定的**。比方说，如果之前碰到有个人占了茅坑一小会，那么它就自己在旁边晃悠一下，比如10次；如果之前碰到别人占了好一会，那么它可能晃悠100圈；如果更变态的之前碰到别人占着茅坑不拉屎，在里面玩游戏，那么它干脆就懒得在旁边晃悠了，乖乖跑到厕所外等待。

自适应自旋锁肯定比普通的自旋锁更“聪明”，相信，随着虚拟机的发展，以后它会越来越聪明。

**2. 锁消除**

锁消除是指虚拟机即时编译器在运行时，对一些在代码上要求同步，但被检测到不可能出现共享数据竞争的锁进行消除。

**3. 锁粗化**

前面两种都是伟大的虚拟机大侠自动帮我们这些小白解决的，但虚拟机也不是救世主，有些情况需要我们自己在设计代码的时候考虑的，具体场景应具体对待。

通常映像中，我们应该是认为锁的范围越小越好，这样可以使多个线程异步执行的范围就越大，效率应该就越高。但是如果一系列的操作都对同一个对象进行反复的加锁/解锁，甚至更变态的加锁的操作是在循环体中进行的，那即时没有线程竞争，频繁的进行互斥同步也会带来很多不必要的性能消耗。

所以，反而言之，如果对同一个对象的同步操作比较多，那么我们是否可以考虑将这些操作放到一个同步块或同步方法里面去呢？

**4. 锁细化**

和上面那个家伙不同，如果对同一个对象操作比较少，只需要很小的一块地方需要同步，那么久尽量去细化，也就是被同步的代码块或方法尽量的简短。