# 管程 进程 线程

进程：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是**操作系统**动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

线程：线程是进程中的实体，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程不拥有系统资源，只有运行必须的一些**数据结构**；它与父进程的其它线程共享该进程所拥有的全部资源。线程可以创建和撤消线程，从而实现程序的并发执行。一般，线程具有就绪、阻塞和运行三种基本状态。  
管程：管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。

一个进程至少有一个线程. 线程的划分尺度小于进程，使得多线程程序的并发性高。另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。   
   从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

   进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.   
   一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行。

信号量机制的引入解决了进程同步的描述问题，但信号量的大量同步操作分散在各个进程中不便于管理，还有可能导致系统死锁。如：生产者消费者问题中将P、V颠倒可能死锁。

      为此Dijkstra于1971年提出：把所有进程对某一种临界资源的同步操作都集中起来，构成一个所谓的秘书进程。凡要访问该临界资源的进程，都需先报告秘书，由秘书来实现诸进程对同一临界资源的互斥使用。

管程由四部分组成：

    1).管程内部的共享变量。

    2).管程内部的条件变量。

    3).管程内部并行执行的进程。

    4).对于局部与管程内部的共享数据设置初始值的语句**。**

由此可见，管程相当于围墙，它把共享变量和对它进行操作的若干个过程围了起来，所有的进程要访问临界资源时，都必须经过管程才能进入，而管程每次只允许一个进程进入管程，从而实现了进程的互斥。

# 哲学家就餐问题

http://blog.csdn.net/sunflowerfiona/article/details/52756819

# 信号量与互斥锁区别

信号量与普通整型变量的区别：

①信号量(semaphore)是非负整型变量，除了初始化之外，它只能通过两个标准原子操作：wait(semap) , signal(semap) ; 来进行访问；

②操作也被成为PV原语（P来源于Dutch proberen"测试"，V来源于Dutch verhogen"增加"），而普通整型变量则可以在任何语句块中被访问；

信号量与互斥锁之间的区别：

1. 互斥量用于线程的互斥，信号线用于线程的同步。    
  
这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别。    
  
互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。    
  
同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源    
  
2. 互斥量值只能为0/1，信号量值可以为非负整数。    
  
也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。信号量可以实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。    
  
3. 互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

信号量

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施, 它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源。

信号量可以分为几类：    
² 二进制信号量(binary semaphore)：只允许信号量取0或1值，其同时只能被一个线程获取。    
  
² 整型信号量（integer semaphore)：信号量取值是整数，它可以被多个线程同时获得，直到信号量的值变为0。    
  
² 记录型信号量（record semaphore)：每个信号量s除一个整数值value（计数）外，还有一个等待队列List，其中是阻塞在该信号量的各个线程的标识。当信号量被释放一个，值被加一后，系统自动从等待队列中唤醒一个等待中的线程，让其获得信号量，同时信号量再减一。    
  
信号量通过一个计数器控制对共享资源的访问，信号量的值是一个非负整数，所有通过它的线程都会将该整数减一。如果计数器大于0，则访问被允许，计数器减1；如果为0，则访问被禁止，所有试图通过它的线程都将处于等待状态。    
  
计数器计算的结果是允许访问共享资源的通行证。因此，为了访问共享资源，线程必须从信号量得到通行证， 如果该信号量的计数大于0，则此线程获得一个通行证，这将导致信号量的计数递减，否则，此线程将阻塞直到获得一个通行证为止。当此线程不再需要访问共享资源时，它释放该通行证，这导致信号量的计数递增，如果另一个线程等待通行证，则那个线程将在那时获得通行证。    
  
Semaphore可以被抽象为五个操作：    
- 创建 Create    
  
- 等待 Wait：    
  
线程等待信号量，如果值大于0，则获得，值减一；如果只等于0，则一直线程进入睡眠状态，知道信号量值大于0或者超时。    
  
-释放 Post    
  
执行释放信号量，则值加一；如果此时有正在等待的线程，则唤醒该线程。    
  
-试图等待 TryWait    
  
如果调用TryWait，线程并不真正的去获得信号量，还是检查信号量是否能够被获得，如果信号量值大于0，则TryWait返回成功；否则返回失败。    
  
-销毁 Destroy    
  
信号量，是可以用来保护两个或多个关键代码段，这些关键代码段不能并发调用。在进入一个关键代码段之前，线程必须获取一个信号量。如果关键代码段中没有任何线程，那么线程会立即进入该框图中的那个部分。一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量，然后将Acquire Semaphore VI以及Release Semaphore VI分别放置在每个关键代码段的首末端。确认这些信号量VI引用的是初始创建的信号量。 动作\系统  
 Win32  
 POSIX  
   
创建  
 CreateSemaphore  
 sem\_init  
   
等待  
 WaitForSingleObject  
 sem \_wait  
   
释放  
 ReleaseMutex  
 sem \_post  
   
试图等待  
 WaitForSingleObject  
 sem \_trywait  
   
销毁  
 CloseHandle  
 sem\_destroy

互斥量(Mutex)    
  
互斥量表现互斥现象的数据结构，也被当作二元信号灯。一个互斥基本上是一个多任务敏感的二元信号，它能用作同步多任务的行为，它常用作保护从中断来的临界段代码并且在共享同步使用的资源。    
Mutex本质上说就是一把锁，提供对资源的独占访问，所以Mutex主要的作用是用于互斥。Mutex对象的值，只有0和1两个值。这两个值也分别代表了Mutex的两种状态。值为0, 表示锁定状态，当前对象被锁定，用户进程/线程如果试图Lock临界资源，则进入排队等待；值为1，表示空闲状态，当前对象为空闲，用户进程/线程可以Lock临界资源，之后Mutex值减1变为0。    
  
Mutex可以被抽象为四个操作：    
  
- 创建 Create    
  
- 加锁 Lock    
  
- 解锁 Unlock    
  
- 销毁 Destroy    
  
Mutex被创建时可以有初始值，表示Mutex被创建后，是锁定状态还是空闲状态。在同一个线程中，为了防止死锁，系统不允许连续两次对Mutex加锁(系统一般会在第二次调用立刻返回)。也就是说，加锁和解锁这两个对应的操作，需要在同一个线程中完成。    
  
不同操作系统中提供的Mutex函数: 动作\系统  
 Win32  
 Linyx  
 Solaris  
   
创建  
 CreateMutex  
 pthread\_mutex\_init  
 mutex\_init  
   
加锁  
 WaitForSingleObject  
 pthread\_mutex\_lock  
 mutex\_lock  
   
解锁  
 ReleaseMutex  
 pthread\_mutex\_unlock  
 mutex\_unlock  
   
销毁  
 CloseHandle  
 pthread\_mutex\_destroy  
 mutex\_destroy

# 乐观锁 悲观锁

为什么需要锁（并发控制）？

　　在**多用户环境**中，在同一时间可能会有多个用户更新相同的记录，这会产生冲突。这就是著名的并发性问题。

**典型的冲突**有：

* **丢失更新**：一个事务的更新覆盖了其它事务的更新结果，就是所谓的更新丢失。例如：用户A把值从6改为2，用户B把值从2改为6，则用户A丢失了他的更新。
* **脏读**：当一个事务读取其它完成一半事务的记录时，就会发生脏读取。例如：用户A,B看到的值都是6，用户B把值改为2，用户A读到的值仍为6。

为了解决这些并发带来的问题。 我们需要引入并发控制机制。

# 并发控制机制

**悲观锁：假定会发生并发冲突，**屏蔽一切可能违反数据完整性的操作。[1]

**乐观锁：假设不会发生并发冲突，**只在提交操作时检查是否违反数据完整性。[1] 乐观锁不能解决脏读的问题。

**乐观锁应用**

**乐观锁介绍：**

　　乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。那么我们如何实现乐观锁呢，一般来说有以下2种方式：

　　1.使用**数据版本**（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

2.乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型**使用时间戳（timestamp）**, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

乐观锁并发性能比较好。

## 悲观锁应用

需要使用数据库的锁机制，比如SQL SERVER 的TABLOCKX（排它表锁） 此选项被选中时，SQL  Server  将在整个表上置排它锁直至该命令或事务结束。这将防止其他进程读取或修改表中的数据。

传统的关系型[数据库](http://lib.csdn.net/base/mysql)里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。传统的关系型[数据库](http://lib.csdn.net/base/mysql)里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

悲观锁具有排他性。

# 结论

　　在实际生产环境里边,如果并发量不大且不允许脏读，可以使用悲观锁解决并发问题；但如果系统的并发非常大的话,悲观锁定会带来非常大的性能问题,所以我们就要选择乐观锁定的方法.

# 多线程的代价

## 设计更加复杂

虽然一些多线程应用程序比单线程应用程序简单，但是其他一般更加复杂。共享数据 同步的处理。

### 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。

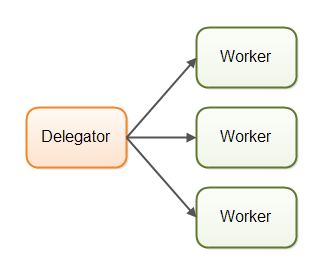
### 增加资源消耗

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。

## 并发编程模型

### 并行工作者模型

传入的作业会被分配到不同的工作者上。



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。

优点：比较简单，只需要添加更多的工作者来提高系统的并行度。

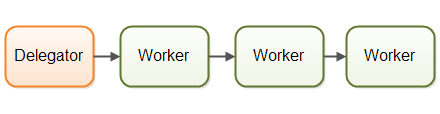
比如网络爬虫，是一个IO密集型工作，在等待数据下载的过程中CPU将会空闲大量时间。

缺点：

共享状态可能比较复杂。

任务执行顺序是不确定的

### 流水线模型

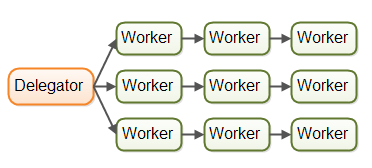


每个工作者只负责作业中的部分工作。当完成了自己的这部分工作时，工作者会将作业转发给下一个工作者。每个工作者在自己的线程中运行，并且不会和其他工作者共享状态。也称为无共享并行模型。

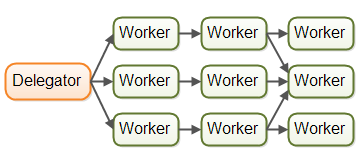
通常使用非阻塞的IO来设计使用流水线并发模型的系统。

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），这个工作者不会一直等待IO操作结束。IO操作速度很慢，等待IO操作结束会很浪费CPU时间，此时CPU可以做其他的事情。当IO操作完成后，IO操作的结果被传递给下一个工作者。

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：

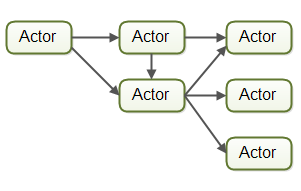


采用流水线并发模型的系统有时候也称为反应器系统或事件驱动系统。

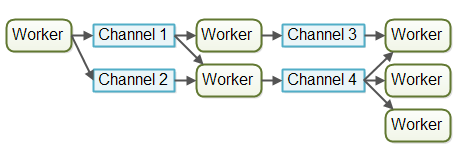
系统内的工作者对系统内出现的事件做出反应，这些事件也有可能来自于外部世界或者发自其他工作者。事件可以是传入的HTTP请求，也可以是某个文件成功加载到内存中等。

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：



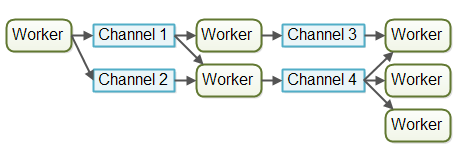
优点：

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。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。

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：



缺点：

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。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。

### 函数式并行

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。

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。

# 创建并运行线程

1 创建一个类继承Thread类，并重写run方法

public class MyThread extends Thread {

public void run(){

System.out.println("MyThread running");

}

}

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。

Thread thread = new Thread(){

public void run(){

System.out.println("Thread Running");

}

};

thread.start();

2 实现Runnable接口，重写run方法

public class MyRunnable implements Runnable {

public void run(){

System.out.println("MyRunnable running");

}

}

Thread thread = new Thread(new MyRunnable());

thread.start();

Runnable myRunnable = new Runnable(){

public void run(){

System.out.println("Runnable running");

}

}

Thread thread = new Thread(myRunnable);

thread.start();

3 线程池

## 竞态条件与临界区

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。在临界区中使用适当的同步就可以避免竞态条件。

# 线程安全与共享资源

线程安全：允许被多个线程同时执行的代码称作线程安全的代码。它不会产生不确定的结果。

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。

## 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。

## 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。

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。

## 线程控制逃逸规则

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。

# 线程安全性与不可变性

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。

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

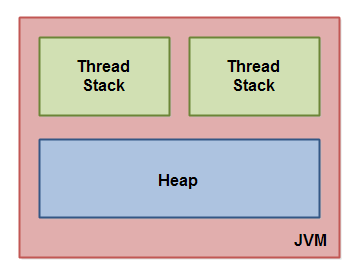
}

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。

# 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内存模型把Java虚拟机内部划分为线程栈和堆。



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。

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。

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。

硬件内存架构



每个CPU都包含一系列的寄存器，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。

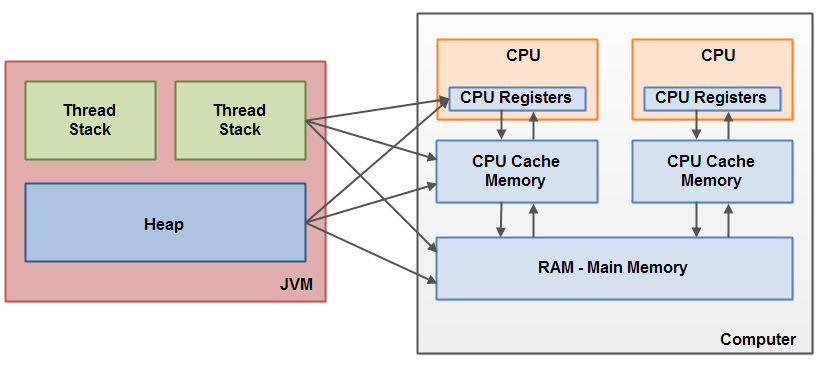
每个CPU可能还有一个CPU缓存层。实际上，绝大多数的现代CPU都有一定大小的缓存层。CPU访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。

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

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。



## 共享对象的可见性

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。

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。

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。

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。

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。

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。

# Java同步块

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

## 实例方法同步

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。

## 实例方法同步块

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。

## 静态方法同步块

不允许同时被线程访问，除非第二个方法不是同步在同一个类对象上的。

# 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：

## 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。

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

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()。

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。

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

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

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。

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()。

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

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。

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。

被唤醒之后的线程会再次获取锁对象，如果获得锁对象，则直接从wait方法之后继续执行。这也就是为什么使用if会出现假唤醒的原因。

## 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()

MyWaitNotify例子里使用字符串常量（””）作为管程对象。

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;

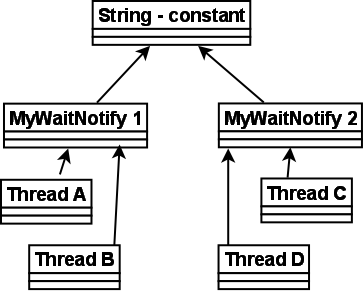
myMonitorObject.notify();

}

}

}

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。

**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**。

# Java ThreadLocal

Java中的ThreadLocal类可以让你创建的变量只被同一个线程进行读和写操作。因此，尽管有两个线程同时执行一段相同的代码，而且这段代码又有一个指向同一个ThreadLocal变量的引用，但是这两个线程依然不能看到彼此的ThreadLocal变量域。

## 创建一个ThreadLocal对象

private ThreadLocal myThreadLocal = new ThreadLocal();

你实例化了一个ThreadLocal对象。每个线程仅需要实例化一次即可。虽然不同的线程执行同一段代码时，访问同一个ThreadLocal变量，但是每个线程只能看到私有的ThreadLocal实例。所以不同的线程在给ThreadLocal对象设置不同的值时，他们也不能看到彼此的修改。

## 访问ThreadLocal对象

一旦创建了一个ThreadLocal对象，你就可以通过以下方式来存储此对象的值：

myThreadLocal.set("A thread local value");

也可以直接读取一个ThreadLocal对象的值：

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

get()方法会返回一个Object对象，而set()方法则依赖一个Object对象参数。

## ThreadLocal泛型

为了使get()方法返回值不用做强制类型转换，通常可以创建一个泛型化的ThreadLocal对象。以下就是一个泛型化的ThreadLocal示例：

private ThreadLocal myThreadLocal1 = new ThreadLocal<String>();

现在你可以存储一个字符串到ThreadLocal实例里，此外，当你从此ThreadLocal实例中获取值的时候，就不必要做强制类型转换

myThreadLocal1.set("Hello ThreadLocal");

String threadLocalValues = myThreadLocal.get();

## 初始化ThreadLocal

由于ThreadLocal对象的set()方法设置的值只对当前线程可见，那有什么方法可以为ThreadLocal对象设置的值对所有线程都可见。

为此，我们可以通过ThreadLocal子类的实现，并覆写initialValue()方法，就可以为ThreadLocal对象指定一个初始化值。如下所示:

private ThreadLocal myThreadLocal = new ThreadLocal<String>() {

@Override protected String initialValue() {

return "This is the initial value";

}

};

此时，在set()方法调用前，当调用get()方法的时候，所有线程都可以看到同一个初始化值。

public class ThreadLocalExample {

public static class MyRunnable implements Runnable {

private ThreadLocal<Integer> threadLocal =

new ThreadLocal<Integer>();

@Override

public void run() {

threadLocal.set( (int) (Math.random() \* 100D) );

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

}

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

}

}

public static void main(String[] args) {

MyRunnable sharedRunnableInstance = new MyRunnable();

Thread thread1 = new Thread(sharedRunnableInstance);

Thread thread2 = new Thread(sharedRunnableInstance);

thread1.start();

thread2.start();

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

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

}

}

上面创建了两个线程共享一个MyRunnable实例。每个线程执行run()方法的时候，会给同一个ThreadLocal实例设置不同的值。如果调用set()方法的时候用synchronized关键字同步，而且不是一个ThreadLocal对象实例，那么第二个线程将会覆盖第一个线程所设置的值。

然而，由于是ThreadLocal对象，所以两个线程无法看到彼此的值。因此，可以设置或者获取不同的值。

## InheritableThreadLocal

InheritableThreadLocal类是ThreadLocal的子类。为了解决ThreadLocal实例内部每个线程都只能看到自己的私有值，所以InheritableThreadLocal允许一个线程创建的所有子线程访问其父线程的值。

# 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。

#### 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。

# 避免死锁

## 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。

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

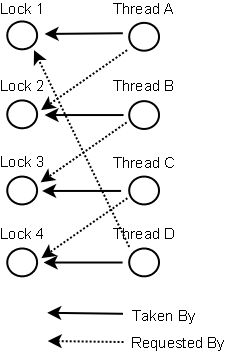
死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

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。

# 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。

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)。

2. 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。

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。

后面讲的不好 自行百度公平锁的原理

ReentrantLock有两个构造方法，默认是无参构造。带参构造是传入一个boolean值，决定是否创建公平锁。其内部有三个内部类，Sync NonfairSync FairSync,后两个是继承了Sync, Sync继承了AbstractQueuedSynchronizer，采用的是模板方法模式。内部维护了一个等待队列，先进先出

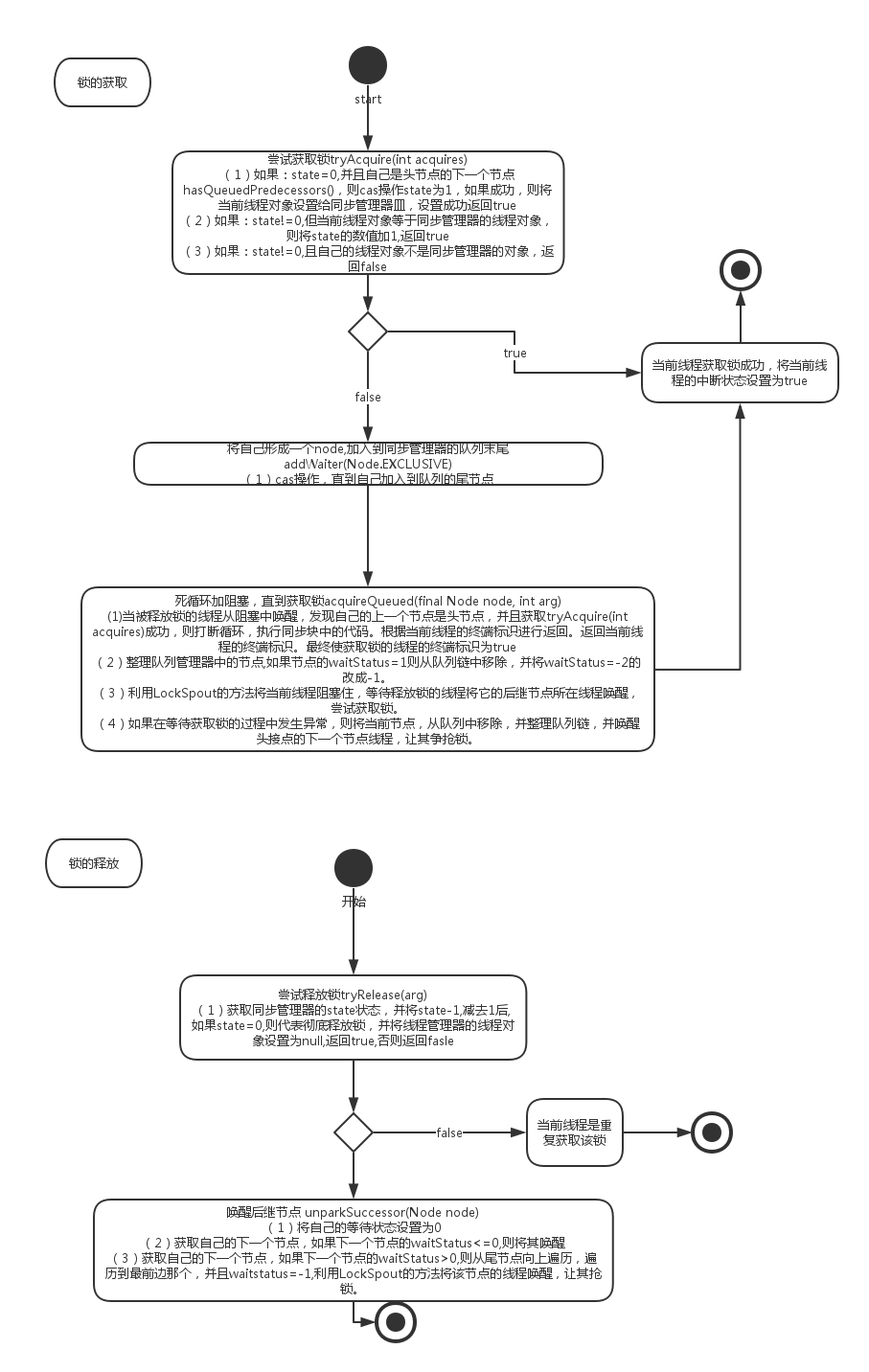
ReentrantLock在采用非公平锁构造时，首先检查锁的状态，如果锁可用，则直接通过CAS（比较并交换）设置为持有状态（1），且把当前线程设置为锁的拥有者。如果当前锁已经被持有，则接下来进行可重入检查，如果可重入，则需要为锁状态加上一个请求数。如果不属于上面两种情况，说明锁是被其他线程持有的，当前线程会放入一个等待队列中。

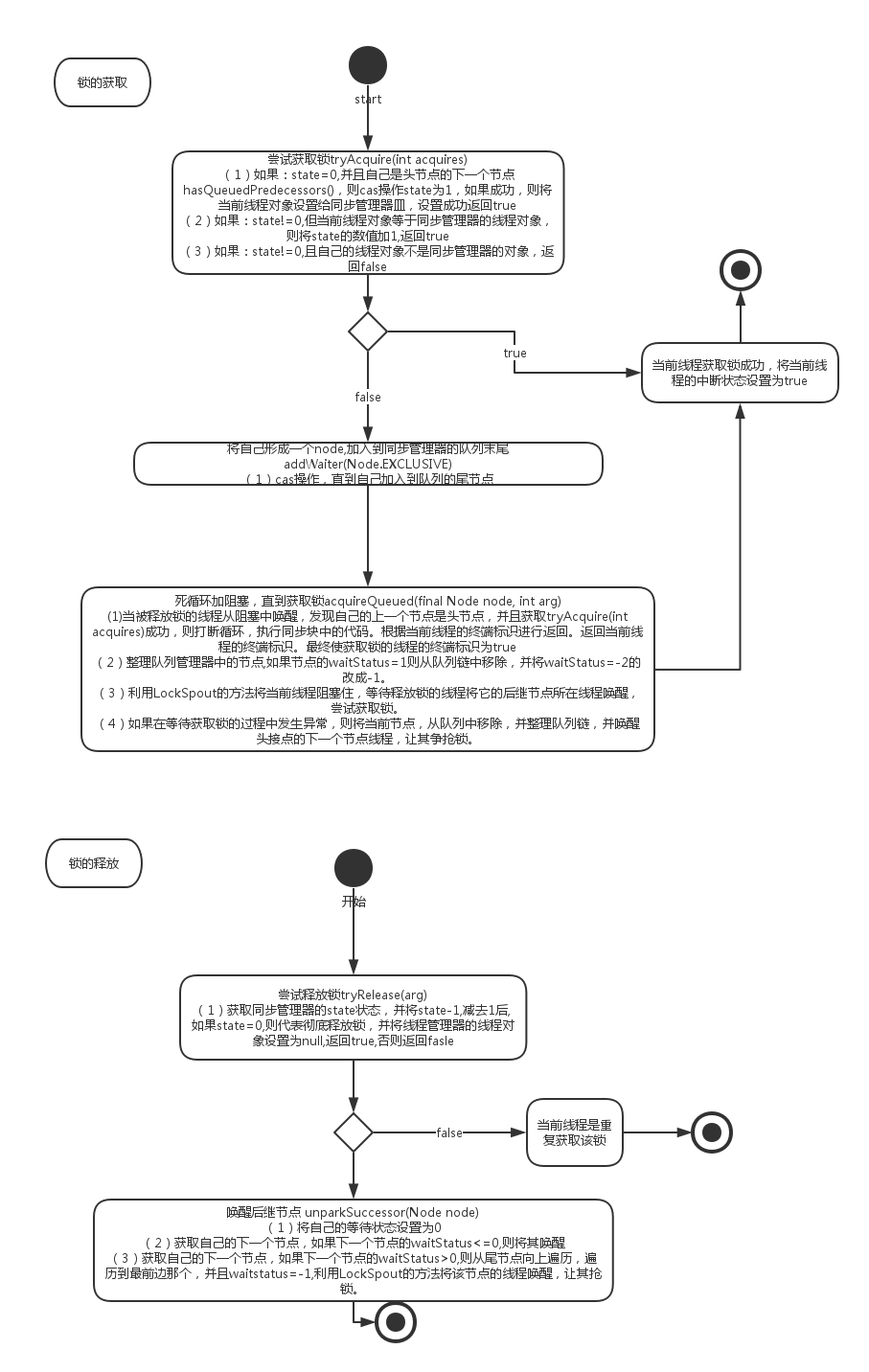
在放入等待队列过程中，首先要检查队列是否为空，如果为空，则需要创建虚拟的头节点，然后把对当前线程封装的节点加入到队列尾部。由于设置尾部节点采用了CAS，为了保证尾节点能够设置成功，这里采用了无限循环方式，直至设置成功。

在完成放入等待队列任务后，则需要维护节点的状态，以及及时清除处于Cancel状态的节点，以帮助垃圾收集器及时回收。如果当前节点之前的节点的等待状态小于1，说明当前节点之前的线程处于等待状态(挂起)，那么当前节点的线程也应处于等待状态(挂起)。挂起的工作是由LockSupport类支持的，LockSupport通过JNI调用本地操作系统来完成挂起的任务(java中除了废弃的suspend等方法，没有其他的挂起操作)。  
    在当前等待的线程，被唤起后，检查中断状态，如果处于中断状态，那么需要中断当前线程。

公平锁的锁获取，严格按照等待顺序进行锁获取，在获取锁的时候有一个判断hasQueuedPredeccssors(),同步队列中是否有前驱节点在等待获取锁，如果有，则放弃获取锁，而是添加到队尾，排队获取锁

在释放锁时，如果是公平锁，则当前节点会检查前驱是否在等待锁，如果在等待，则前驱释放锁，自己继续等待。





非公平锁性能高于公平锁性能的原因：

在恢复一个被挂起的线程与该线程真正运行之间存在着严重的延迟。

假设线程A持有一个锁，并且线程B请求这个锁。由于锁被A持有，因此B将被挂起。当A释放锁时，B将被唤醒，因此B会再次尝试获取这个锁。与此同时，如果线程C也请求这个锁，那么C很可能会在B被完全唤醒之前获得、使用以及释放这个锁。这样就是一种双赢的局面：B获得锁的时刻并没有推迟，C更早的获得了锁，并且吞吐量也提高了。

当持有锁的时间相对较长或者请求锁的平均时间间隔较长，应该使用公平锁。在这些情况下，插队带来的吞吐量提升（当锁处于可用状态时，线程却还处于被唤醒的过程中）可能不会出现。

ReentrantLock具有公平和非公平两种模式，也各有优缺点：  
公平锁是严格的以FIFO的方式进行锁的竞争，但是非公平锁是无序的锁竞争，刚释放锁的线程很大程度上能比较快的获取到锁，队列中的线程只能等待，所以非公平锁可能会有“饥饿”的问题。但是重复的锁获取能减小线程之间的切换，而公平锁则是严格的线程切换，这样对操作系统的影响是比较大的，所以非公平锁的吞吐量是大于公平锁的，这也是为什么JDK将非公平锁作为默认的实现。

# 嵌套管程锁死

嵌套管程锁死类似于死锁

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……

//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。

#### 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](http://ifeve.com/deadlock/)中我们已经对死锁有了个大概的解释，死锁通常是因为两个线程获取锁的顺序不一致造成的，线程1锁住A，等待获取B，线程2已经获取了B，再等待获取A。如[死锁避免](http://ifeve.com/deadlock-prevention/)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。

# 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。

# Java中的锁

## 同步块

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。

用Lock代替synchronized

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。

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();

}

}

其中的while(isLocked)循环，它又被叫做“自旋锁”。

当isLocked为true时，调用lock()的线程在wait()调用上阻塞等待。为防止该线程没有收到notify()调用也从wait()中返回（也称作虚假唤醒），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：

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

ava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。

## ****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。

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。

# Java中读写锁

<http://www.codeceo.com/article/java-rd-lock.html>

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。

## 读/写锁的Java实现

**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。

简单的读写锁实现

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();

}

}

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

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;

}

}

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。

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。

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

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;

}

}

**重入性**

* 读写锁允许读线程和写线程按照请求锁的顺序重新获取读取锁或者写入锁。当然了只有写线程释放了锁，读线程才能获取重入锁。
* 写线程获取写入锁后可以再次获取读取锁，但是读线程获取读取锁后却不能获取写入锁。
* 另外读写锁最多支持65535个递归写入锁和65535个递归读取锁。

# 重入锁死

重入锁死与死锁和嵌套管程锁死非常相似。

重入锁死与死锁和嵌套管程锁死非常相似。锁和读写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：

public class Reentrant{

public synchronized outer(){

inner();

}

public synchronized inner(){

//do something

}

}

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

# 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。或者像锁一样用于保护一个关键区域。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。

<http://ifeve.com/semaphore/>

信号丢失: 线程 A 必须在 B 发送信号之前开始等待。如果 B 在 A 执行等待操作之前发送了信号，这个信号就丢失了，同时 A会一直阻塞直到其它线程再次发送信号到这个条件变量。

# 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。

# 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。

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();

}

}

}

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。

# 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。

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。

# 剖析同步器

http://ifeve.com/anatomy-of-a-synchronizer/

# 死锁 嵌套管程锁死 重入锁死