# 线程与进程

一个进程对应一个应用程序，都有独立的内存空间

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位

同一个进程中的线程共享其进程中的内存和资源（共享的内存是堆内存和方法区内存，栈内存不共享，每个线程有自己的。）

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。更多详细信息请点击这里。

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

# 多线程优势

1、发挥多处理器的强大能力

现在，多处理器系统正日益盛行，并且价格不断降低，即时在低端服务器和中断桌面系统中，通常也会采用多个处理器，这种趋势还在进一步加快，因为通过提高时钟频率来提升性能已变得越来越困难，处理器生产厂商都开始转而在单个芯片上放置多个处理器核。试想，如果只有单个线程，双核处理器系统上程序只能使用一半的CPU资源，拥有100个处理器的系统上将有99%的资源无法使用。多线程程序则可以同时在多个处理器上执行，如果设计正确，**多线程程序可以通过提高处理器资源的利用率来提升系统吞吐率**。

2、在单处理器系统上获得更高的吞吐率

如果程序是单线程的，那么当程序等待某个同步I/O操作完成时，处理器将处于空闲状态。而在多线程程序中，如果一个线程在等待I/O操作完成，另一个线程可以继续运行，使得程序能在I/O阻塞期间继续运行。

3、建模的简单性

通过使用线程，可以将复杂并且异步的工作流进一步分解为一组简单并且同步的工作流，每个工作流在一个单独的线程中运行，并在特定的同步位置进行交互。我们可以通过一些现有框架来实现上述目标，例如Servlet和RMI，框架负责解决一些细节问题，例如请求管理、线程创建、负载平衡，并在正确的时候将请求分发给正确的应用程序组件。编写Servlet的开发人员不需要了解多少请求在同一时刻要被处理，也不需要了解套接字的输入流或输出流是否被阻塞，当调用Servlet的service方法来响应Web请求时，可以以同步的方式来处理这个请求，就好像它是一个单线程程序。

4、异步事件的简化处理

服务器应用程序在接受多个来自远程客户端的套接字连接请求时，如果为每个连接都分配其各自的线程并且使用同步I/O，那么就会降低这类程序的开发难度。如果某个应用程序对套接字执行读操作而此时还没有数据到来，那么这个读操作将一直阻塞，直到有数据到达。在单线程应用程序中，这不仅意味着在处理请求的过程中将停顿，而且还意味着在这个线程被阻塞期间，对所有请求的处理都将停顿。为了避免这个问题，单线程服务器应用程序必须使用非阻塞I/O，但是这种I/O的复杂性要远远高于同步I/O，并且很容易出错。然而，如果每个请求都拥有自己的处理线程，那么在处理某个请求时发生的阻塞将不会影响其他请求的处理。

# 多线程的实现

## 继承Thread类

## 实现Runnable接口

## 使用Callable和Future实现有返回值的多线程



# 上下文切换

对于单核CPU来说（对于多核CPU，此处就理解为一个核），CPU在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

由于可能当前线程的任务并没有执行完毕，所以在切换时需要保存线程的运行状态，以便下次重新切换回来时能够继续切换之前的状态运行。

对于线程的上下文切换实际上就是 **存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行**。

# java中fail-fast 和 fail-safe的区别

## fail-fast

### 什么是fail-fast

fail-fast机制在遍历一个集合时，当集合结构被修改，会抛出Concurrent Modification Exception。

fail-fast会在以下两种情况下抛出ConcurrentModificationException

（1）单线程环境

集合被创建后，在遍历它的过程中修改了结构。

注意 remove()方法会让expectModcount和modcount 相等，所以是不会抛出这个异常。

（2）多线程环境

当一个线程在遍历这个集合，而另一个线程对这个集合的结构进行了修改。

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误的做法：迭代器的快速失败行为应该仅用于检测 bug。

### fail-fast机制是如何检测的

迭代器在遍历过程中是直接访问内部数据的，因此内部的数据在遍历的过程中无法被修改。为了保证不被修改，迭代器内部维护了一个标记 “mode” ，当集合结构改变（添加删除或者修改），标记"mode"会被修改，而迭代器每次的hasNext()和next()方法都会检查该"mode"是否被改变，当检测到被修改时，抛出Concurrent Modification Exception。

## fail-safe机制

fail-safe任何对集合结构的修改都会在一个复制的集合上进行修改，因此不会抛出ConcurrentModificationException

fail-safe机制有两个问题

（1）需要复制集合，产生大量的无效对象，开销大

（2）无法保证读取的数据是目前原始数据结构中的数据。

# 竞态条件

竞态条件是指多个线程同时访问或者操作同一块数据，运行的结果依赖于不同线程访问数据的顺序。通俗一点讲就是线程A 需要判断一个变量的状态，然后根据这个变量的状态来执行某个操作。在执行这个操作之前，这个变量的状态可能会被其他线程修改。

比如：我们在构造懒加载模式单例时，需要先判断要构造的单例对象是否已经创建，如果没有创建则进行创建。如果线程A和线程B同时执行这一块代码，此时就会出现竞态条件；当线程A判断单例对象为空，此时线程B也执行这块代码并判断单例对象为空从而构造一个单例对象；此时A再继续执行后面代码从而又创建了一次，从而重复创建单例对象从而可能会引发一些问题；

临界区实现方法有两种，一种是用synchronized，一种是用Lock显式锁实现。

# 线程状态



**说明**：  
线程共包括以下5种状态。  
1. **新建状态(New)**: 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。  
2. **就绪状态(Runnable)**: 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。  
3. **运行状态(Running)** : 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。  
4. **阻塞状态(Blocked)** : 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：  
    (01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。  
    (02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。  
    (03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。  
5. **死亡状态(Dead)**    : 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

这5种状态涉及到的内容包括Object类, Thread和synchronized关键字。这些内容我们会在后面的章节中逐个进行学习。

# 线程几个重要方法

## Join

join() 的作用：让“主线程”等待“子线程”结束之后才能继续运行。

## Sleep

静态方法。sleep()作用是在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行）。sleep相当于让线程睡眠，交出CPU，让CPU去执行其他的任务。但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。

## yield

静态方法。调用yield方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。

注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。也就是说当前线程执行yield方法后，可能继续执行任务。

## isAlive

方法isAlive()的作用是测试线程是否偶处于活动状态。什么是活动状态呢？活动状态就是线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是“存活”的。有个需要注意的地方

## join

非静态方法。在很多情况下，主线程创建并启动了线程，如果子线程中药进行大量耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到join()方法了。方法join()的作用是等待线程对象销毁。

## 总结

### ****sleep()方法和对象的wait()方法区别****

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

### ****sleep()方法和yield()方法的区别****

1、sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；  
 2、线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；  
 3、sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；  
 4、sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

### ****与线程同步以及线程调度相关的方法****

1、wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

2、sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；

3、notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；

4、notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

# 线程等待和唤醒

## 方法介绍

Object类中关于等待/唤醒的API详细信息如下：  
**notify()**-- 唤醒在此对象监视器上等待的单个线程。  
**notifyAll()**  -- 唤醒在此对象监视器上等待的所有线程。  
**wait()** -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout)** -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout, int nanos)**-- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量”，当前线程被唤醒(进入“就绪状态”)。

## 为什么notify(), wait()等函数定义在Object中

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“当前线程”等待，因为线程进入等待状态，所以线程应该释放它锁持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
OK，线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“唤醒线程”)，它只有在获取“该对象的同步锁”(这里的同步锁必须和等待线程的同步锁是同一个)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

# 线程优先级

在操作系统中，线程可以划分优先级，优先级较高的线程得到的CPU资源较多，也就是CPU优先执行优先级较高的线程对象中的任务。设置线程优先级有助于帮“线程规划器”确定在下一次选择哪一个线程来优先执行。设置线程的优先级使用setPriority()方法。在Java中，线程的优先级分为1~10这10个等级，默认是5，如果小于1或大于10，则JDK抛出异常throw new IllegalArgumentException()。

## 优先级的特性

继承性  
比如A线程启动B线程，则B线程的优先级与A是一样的。

规则性  
高优先级的线程总是大部分先执行完，但不代表高优先级线程全部先执行完。

随机性  
优先级较高的线程不一定每一次都先执行完。

# 守护线程

在Java线程中有两种线程，一种是User Thread（用户线程），另一种是Daemon Thread(守护线程)。

Daemon的作用是为其他线程的运行提供服务，比如说GC线程。其实User Thread线程和Daemon Thread守护线程本质上来说去没啥区别的，唯一的区别之处就在虚拟机的离开：如果User Thread全部撤离，那么Daemon Thread也就没啥线程好服务的了，所以虚拟机也就退出了。

守护线程并非虚拟机内部可以提供，用户也可以自行的设定守护线程，方法：public final void setDaemon(boolean on) ；但是有几点需要注意：

thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。 （备注：这点与守护进程有着明显的区别，守护进程是创建后，让进程摆脱原会话的控制+让进程摆脱原进程组的控制+让进程摆脱原控制终端的控制；所以说寄托于虚拟机的语言机制跟系统级语言有着本质上面的区别）

在Daemon线程中产生的新线程也是Daemon的。 （这一点又是有着本质的区别了：守护进程fork()出来的子进程不再是守护进程，尽管它把父进程的进程相关信息复制过去了，但是子进程的进程的父进程不是init进程，所谓的守护进程本质上说就是“父进程挂掉，init收养，然后文件0,1,2都是/dev/null，当前目录到/”）

不是所有的应用都可以分配给Daemon线程来进行服务，比如读写操作或者计算逻辑。因为在Daemon Thread还没来的及进行操作时，虚拟机可能已经退出了。

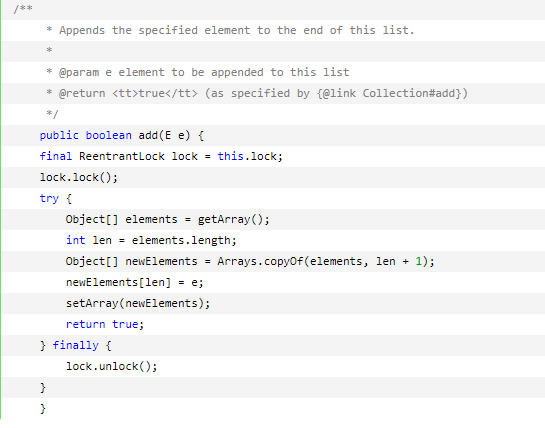
# synchronized关键字

## 作用

# 线程安全集合

## CopyOnWriteArrayList

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。



### 应用场景

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。



### 最优实践

代码很简单，但是使用CopyOnWriteMap需要注意两件事情：

1. 减少扩容开销。根据实际需要，初始化CopyOnWriteMap的大小，避免写时CopyOnWriteMap扩容的开销。

2. 使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的addBlackList方法。

### 缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

**内存占用问题**。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如[ConcurrentHashMap](http://ifeve.com/concurrenthashmap/" \t "_blank)。

**数据一致性问题**。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

# 原子类

## 基本类型原子类

### 方法

// 构造函数

AtomicLong()

// 创建值为initialValue的AtomicLong对象

AtomicLong(long initialValue)

// 以原子方式设置当前值为newValue。

final void set(long newValue)

// 获取当前值

final long get()

// 以原子方式将当前值减 1，并返回减1后的值。等价于“--num”

final long decrementAndGet()

// 以原子方式将当前值减 1，并返回减1前的值。等价于“num--”

final long getAndDecrement()

// 以原子方式将当前值加 1，并返回加1后的值。等价于“++num”

final long incrementAndGet()

// 以原子方式将当前值加 1，并返回加1前的值。等价于“num++”

final long getAndIncrement()

// 以原子方式将delta与当前值相加，并返回相加后的值。

final long addAndGet(long delta)

// 以原子方式将delta添加到当前值，并返回相加前的值。

final long getAndAdd(long delta)

// 如果当前值 == expect，则以原子方式将该值设置为update。成功返回true，否则返回false，并且不修改原值。

final boolean compareAndSet(long expect, long update)

// 以原子方式设置当前值为newValue，并返回旧值。

final long getAndSet(long newValue)

// 返回当前值对应的int值

int intValue()

// 获取当前值对应的long值

long longValue()

// 以 float 形式返回当前值

float floatValue()

// 以 double 形式返回当前值

double doubleValue()

// 最后设置为给定值。延时设置变量值，这个等价于set()方法，但是由于字段是volatile类型的，因此次字段的修改会比普通字段（非volatile字段）有稍微的性能延时（尽管可以忽略），所以如果不是想立即读取设置的新值，允许在“后台”修改值，那么此方法就很有用。如果还是难以理解，这里就类似于启动一个后台线程如执行修改新值的任务，原线程就不等待修改结果立即返回（这种解释其实是不正确的，但是可以这么理解）。

final void lazySet(long newValue)

// 如果当前值 == 预期值，则以原子方式将该设置为给定的更新值。JSR规范中说：以原子方式读取和有条件地写入变量但不 创建任何 happen-before 排序，因此不提供与除 weakCompareAndSet 目标外任何变量以前或后续读取或写入操作有关的任何保证。大意就是说调用weakCompareAndSet时并不能保证不存在happen-before的发生（也就是可能存在指令重排序导致此操作失败）。但是从Java源码来看，其实此方法并没有实现JSR规范的要求，最后效果和compareAndSet是等效的，都调用了unsafe.compareAndSwapInt()完成操作。

final boolean weakCompareAndSet(long expect, long update)

### 原理

基本类型原子类在进行自增和add方法时，最终都是借助CAS（Compare and Swap）实现原子操作。Java代码中通过compareAndSet方法调用到Unsafe中的compareAndSwapInt方法（循环调用compareAndSet方法直到返回成功）；最终通过jni调用c++代码。而在c++代码中，是采用硬件（处理器）级别实现原子操作。

# Ｈappen-Before规则

我们一般说一个操作happen-before另一个操作，这到底是什么意思呢？当说操作A happen-before操作B时，我们其实是在说在发生操作B之前，操作A对内存施加的影响能够被观测到。所谓“对内存施加的影响”就是指对变量的写入，“被观测到”指当读取这个变量时能够得到刚才写入的值（如果中间没有发生其它的写入）。听起来很绕口？这就对了，请你保持耐心，举个例子来说明一下。线程Ⅰ执行了操作A：x=3，线程Ⅱ执行了操作Ｂ：y=x。如果操作Ａhappen-before操作B，线程Ⅱ在执行操作B之前就确定操作"x=3"被执行了，它能够确定，是因为如果这两个操作之间没有任何对x的写入的话，它读取x的值将得到3，这意味着线程Ⅱ执行操作B会写入y的值为3。如果两个操作之间还有对x的写入会怎样呢？假设线程Ⅲ在操作A和B之间执行了操作C: x=5，并且操作C和操作B之前并没有happen-before关系(后面我会说明时间上的先后并不一定导致happen-before关系)。这时线程Ⅱ执行操作B会讲到x的什么值呢？3还是5?答案是两者皆有可能，这是因为happen-before关系**保证一定** 能够观测到前一个操作施加的内存影响，只有时间上的先后关系而并没有happen-before关系**可能但并不保证** 能观测前一个操作施加的内存影响。如果读到了值3，我们就说读到了“**陈旧** ”的数据。正是多种可能性导致了多线程的不确定性和复杂性，但是要分析多线程的安全性，我们只能分析确定性部分，这就要求找出happen-before关系，这又得利用happen-before规则。

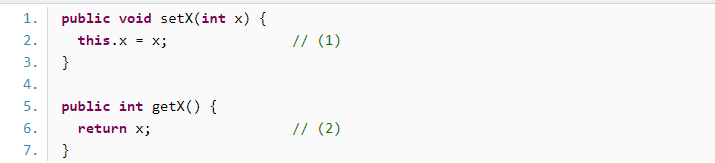
下面是我列出的三条非常重要的happen-before规则，利用它们可以确定两个操作之间是否存在happen-before关系。

1、同一个线程中，书写在前面的操作happen-before书写在后面的操作。这条规则是说，在单线程中操作间happen-before关系完全是由源代码的顺序决定的，这里的前提“在同一个线程中”是很重要的，这条规则也称为单线程规则 。这个规则多少说得有些简单了，考虑到控制结构和循环结构，书写在后面的操作可能happen-before书写在前面的操作，不过我想读者应该明白我的意思。

2、对锁的unlock操作happen-before后续的对同一个锁的lock操作。这里的“后续”指的是时间上的先后关系，unlock操作发生在退出同步块之后，lock操作发生在进入同步块之前。这是条最关键性的规则，线程安全性主要依赖于这条规则。但是仅仅是这条规则仍然不起任何作用，它必须和下面这条规则联合起来使用才显得意义重大。这里关键条件是必须对“同一个锁”的lock和unlock。

3、如果操作A happen-before操作B，操作B happen-before操作C，那么操作A happen-before操作C。这条规则也称为传递规则。

现在暂时放下happen-before规则，先探讨一下“一个操作在时间上先于另一个操作发生”和“一个操作happen-before另一个操作之间”的关系。两者有关联却并不相同。关联部分在第2条happen-before规则中已经谈到了，通常我们得假定一个时间上的先后顺序然后据此得出happen-before关系。不同部分体现在，首先，**一个操作在时间上先于另一个操作发生，并不意味着一个操作happen-before另一个操作** 。看下面的例子：



假设线程Ⅰ先执行setX方法，接着线程Ⅱ执行getX方法，在时间上线程Ⅰ的操作A：this.x = x先于线程Ⅱ的操作B：return x。但是操作A却并不happen-before操作B，让我们逐条检查三条happen-before规则。第１条规则在这里不适用，因为这时两个不同的线程。第２条规则也不适用，因为这里没有任何同步块，也就没有任何lock和unlock操作。第３条规则必须基于已经存在的happen-before关系，现在没有得出任何happen-before关系，因此第三条规则对我们也任何帮助。通过检查这三条规则，我们就可以得出，操作A和操作B之间没有happen-before关系。这意味着如果线程Ⅰ调用了setX(3)，接着线程Ⅱ调用了getX()，其返回值可能不是3，尽管两个操作之间没有任何其它操作对x进行写入，它可能返回任何一个曾经存在的值或者默认值0。“任何曾经存在的值”需要做点解释，假设在线程Ⅰ调用setX(3)之前，还有别的线程或者就是线程Ⅰ还调用过setX(5), setX(8)，那么x的曾经可能值为0, 5和8(这里假设setX是唯一能够改变x的方法)，其中0是整型的默认值，用在这个例子中，线程Ⅱ调用getX()的返回值可能为0, 3, 5和8，至于到底是哪个值是不确定的。

# 锁框架

## 同步锁synchronized

同步锁的原理是，对于每一个对象，有且仅有一个同步锁；不同的线程能共同访问该同步锁。但是，在同一个时间点，该同步锁能且只能被一个线程获取到。这样，获取到同步锁的线程就能进行CPU调度，从而在CPU上执行；而没有获取到同步锁的线程，必须进行等待，直到获取到同步锁之后才能继续运行。这就是，多线程通过同步锁进行同步的原理！

public void produce() {

synchronized (this) {

while (mBuf.isFull()) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

mBuf.add();

notifyAll();

}

}

public void consume() {

synchronized (this) {

while (mBuf.isEmpty()) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

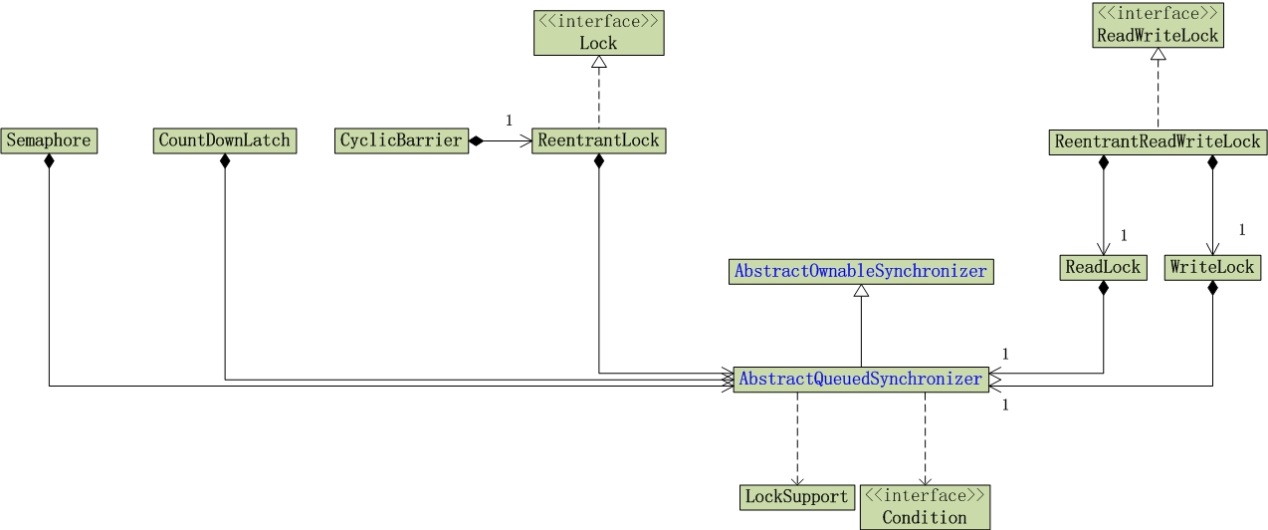
mBuf.remove();

notifyAll();

}

}

## JUC包中的锁



### ****AbstractQueuedSynchronizer****

AbstractQueueSynchronizer简称AQS，是java.util.concurrent的核心，CountDownLatch、FutureTask、Semaphore、ReentrantLock等都有一个内部类是这个抽象类的子类。先用两张表格介绍一下AQS。第一个讲的是Node，由于**AQS是基于FIFO队列的实现**，因此必然存在一个个节点，Node就是一个节点，Node里面有：

|  |  |
| --- | --- |
| **属    性** | **定    义** |
| Node SHARED = new Node() | 表示Node处于共享模式 |
| Node EXCLUSIVE = null | 表示Node处于独占模式 |
| int CANCELLED = 1 | 因为超时或者中断，Node被设置为取消状态，被取消的Node不应该去竞争锁，只能保持取消状态不变，不能转换为其他状态，处于这种状态的Node会被踢出队列，被GC回收 |
| int SIGNAL = -1 | 表示这个Node的继任Node被阻塞了，到时需要通知它 |
| int CONDITION = -2 | 表示这个Node在条件队列中，因为等待某个条件而被阻塞 |
| int PROPAGATE = -3 | 使用在共享模式头Node有可能处于这种状态， 表示锁的下一次获取可以无条件传播 |
| int waitStatus | 0，新Node会处于这种状态 |
| Node prev | 队列中某个Node的前驱Node |
| Node next | 队列中某个Node的后继Node |
| Thread thread | 这个Node持有的线程，表示等待锁的线程 |
| Node nextWaiter | 表示下一个等待condition的Node |

看完了Node，下面再看一下AQS中有哪些变量和方法：

|  |  |
| --- | --- |
| **属性/方法** | **含    义** |
| Thread exclusiveOwnerThread | 这个是AQS父类AbstractOwnableSynchronizer的属性，表示独占模式同步器的当前拥有者 |
| Node | 上面已经介绍过了，FIFO队列的基本单位 |
| Node head | FIFO队列中的头Node |
| Node tail | FIFO队列中的尾Node |
| int state | 同步状态，0表示未锁 |
| int getState() | 获取同步状态 |
| setState(int newState) | 设置同步状态 |
| boolean compareAndSetState(int expect, int update) | 利用CAS进行State的设置 |
| long spinForTimeoutThreshold = 1000L | 线程自旋等待的时间 |
| Node enq(final Node node) | 插入一个Node到FIFO队列中 |
| Node addWaiter(Node mode) | 为当前线程和指定模式创建并扩充一个等待队列 |
| void setHead(Node node) | 设置队列的头Node |
| void unparkSuccessor(Node node) | 如果存在的话，唤起Node持有的线程 |
| void doReleaseShared() | 共享模式下做释放锁的动作 |
| void cancelAcquire(Node node) | 取消正在进行的Node获取锁的尝试 |
| boolean shouldParkAfterFailedAcquire(Node pred, Node node) | 在尝试获取锁失败后是否应该禁用当前线程并等待 |
| void selfInterrupt() | 中断当前线程本身 |
| boolean parkAndCheckInterrupt() | 禁用当前线程进入等待状态并中断线程本身 |
| boolean acquireQueued(final Node node, int arg) | 队列中的线程获取锁 |
| tryAcquire(int arg) | 尝试获得锁（**由AQS的子类实现它**） |
| tryRelease(int arg) | 尝试释放锁（**由AQS的子类实现它**） |
| isHeldExclusively() | 是否独自持有锁 |
| acquire(int arg) | 获取锁 |
| release(int arg) | 释放锁 |
| compareAndSetHead(Node update) | 利用CAS设置头Node |
| compareAndSetTail(Node expect, Node update) | 利用CAS设置尾Node |
| compareAndSetWaitStatus(Node node, int expect, int update) | 利用CAS设置某个Node中的等待状态 |

上面列出了AQS中最主要的一些方法和属性。**整个AQS是典型的模板模式的应用**，设计得十分精巧，对于FIFO队列的各种操作在AQS中已经实现了，**AQS的子类一般只需要重写tryAcquire(int arg)和tryRelease(int arg)两个方法即可**。

### Lock与synchronized区别

1、synchronized是基于JVM层面实现的，而Lock是基于JDK层面实现的。

2、对于使用者的直观体验上Lock比synchronized复杂，需要lock后再finally释放锁，否则会导致死锁。

3、从锁的粒度控制上，lock比synchronized更灵活，更细

4、synchronized不支持锁获取超时和获取锁响应中断，而lock支持

### ****ReentrantLock****

#### 公平锁和非公平锁

公平锁指的是线程获取锁的顺序是按照加锁顺序来的，而非公平锁指的是抢锁机制，先lock的线程不一定先获得锁。

#### 使用

在并发控制API中， Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()，传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

Condition的强大之处在于它可以为多个线程间建立不同的Condition，下面引入API中的一段代码，加以说明。

class BoundedBuffer {

   final Lock lock = new ReentrantLock();//锁对象

   final Condition notFull  = lock.newCondition();//写线程条件

   final Condition notEmpty = lock.newCondition();//读线程条件

   final Object[] items = new Object[100];//缓存队列

   int putptr/\*写索引\*/, takeptr/\*读索引\*/, count/\*队列中存在的数据个数\*/;

   public void put(Object x) throws InterruptedException {

     lock.lock();

     try {

       while (count == items.length)//如果队列满了

         notFull.await();//阻塞写线程

       items[putptr] = x;//赋值

       if (++putptr == items.length) putptr = 0;//如果写索引写到队列的最后一个位置了，那么置为0

       ++count;//个数++

       notEmpty.signal();//唤醒读线程

     } finally {

       lock.unlock();

     }

   }

   public Object take() throws InterruptedException {

     lock.lock();

     try {

       while (count == 0)//如果队列为空

         notEmpty.await();//阻塞读线程

       Object x = items[takeptr];//取值

       if (++takeptr == items.length) takeptr = 0;//如果读索引读到队列的最后一个位置了，那么置为0

       --count;//个数--

       notFull.signal();//唤醒写线程

       return x;

     } finally {

       lock.unlock();

     }

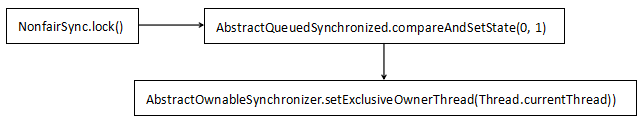
   }

 }

上述代码是是ArrayBlockingQueue的内部实现。体现了Condition的强大之处。假设缓存队列中已经存满，那么阻塞的肯定是写线程，唤醒的肯定是读线程，相反，阻塞的肯定是读线程，唤醒的肯定是写线程，那么假设只有一个Condition会有什么效果呢，缓存队列中已经存满，这个Lock不知道唤醒的是读线程还是写线程了，如果唤醒的是读线程，皆大欢喜，如果唤醒的是写线程，那么线程刚被唤醒，又被阻塞了，这时又去唤醒，这样就浪费了很多时间。

#### 实现原理

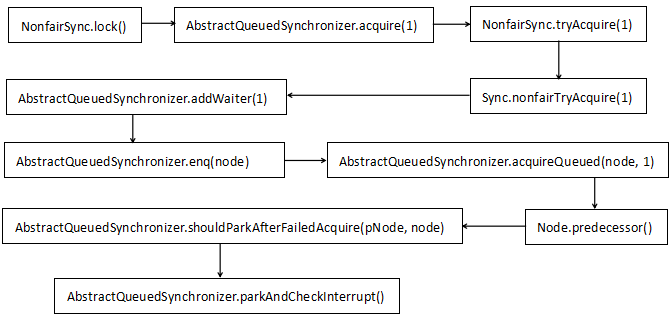
ReentrantLock根据传入构造方法的布尔型参数实例化出Sync的实现类FairSync和NonfairSync，分别表示公平的Sync和非公平的Sync。由于ReentrantLock我们用的比较多的是非公平锁，所以看下非公平锁是如何实现的。假设线程1调用了ReentrantLock的lock()方法，那么线程1将会独占锁，整个调用链十分简单：



第一个获取锁的线程就做了两件事情：

**1、设置AbstractQueuedSynchronizer的state为1**

**2、设置AbstractOwnableSynchronizer的thread为当前线程**

这两步做完之后就表示线程1独占了锁。然后线程2也要尝试获取同一个锁，在线程1没有释放锁的情况下必然是行不通的，所以线程2就要阻塞。那么，线程2如何被阻塞？看下线程2的方法调用链，这就比较复杂了：

#### 总结

Condition与Object中的wati,notify,notifyAll区别：

1.Condition中的await()方法相当于Object的wait()方法，Condition中的signal()方法相当于Object的notify()方法，Condition中的signalAll()相当于Object的notifyAll()方法。  
不同的是，Object中的这些方法是和同步锁捆绑使用的；而Condition是需要与互斥锁/共享锁捆绑使用的。

2.Condition它更强大的地方在于：能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition。  
例如，假如多线程读/写同一个缓冲区：当向缓冲区中写入数据之后，唤醒"读线程"；当从缓冲区读出数据之后，唤醒"写线程"；并且当缓冲区满的时候，"写线程"需要等待；当缓冲区为空时，"读线程"需要等待。

如果采用Object类中的wait(),notify(),notifyAll()实现该缓冲区，当向缓冲区写入数据之后需要唤醒"读线程"时，不可能通过notify()或notifyAll()明确的指定唤醒"读线程"，而只能通过notifyAll唤醒所有线程(但是notifyAll无法区分唤醒的线程是读线程，还是写线程)。 但是，通过Condition，就能明确的指定唤醒读线程。

#### 锁选择

synchronized：   
 在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好，不管用没用过5.0多线程包的程序员都能理解。   
  
 ReentrantLock:   
 ReentrantLock提供了多样化的同步，比如有时间限制的同步，可以被Interrupt的同步（synchronized的同步是不能Interrupt的）等。在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

只要线程可以在30到50次自旋里拿到锁,那么Synchronized就不会升级为重量级锁,而等待的线程也就不用被挂起,我们也就少了挂起和唤醒这个上下文切换的过程开销.

但如果是ReentrantLock呢?不会自旋,而是直接被挂起,这样一来,我们就很容易会多出线程上下文开销的代价.当然,你也可以使用tryLock(),但是这样又出现了一个问题,你怎么知道tryLock的时间呢?在时间范围里还好,假如超过了呢?

所以,在锁被细化到如此程度上,使用Synchronized是最好的选择了.这里再补充一句,Synchronized和ReentrantLock他们的开销差距是在释放锁时唤醒线程的数量,Synchronized是唤醒锁池里所有的线程+刚好来访问的线程,而ReentrantLock则是当前线程后进来的第一个线程+刚好来访问的线程.

如果是线程并发量不大的情况下,那么Synchronized因为自旋锁,偏向锁,轻量级锁的原因,不用将等待线程挂起,偏向锁甚至不用自旋,所以在这种情况下要比ReentrantLock高效。

### ReentrantReadWriteLock

#### 简介

ReadWriteLock，顾名思义，是读写锁。它维护了一对相关的锁 — — “读取锁”和“写入锁”，一个用于读取操作，另一个用于写入操作。“读取锁”用于只读操作，它是“共享锁”，能同时被多个线程获取。“写入锁”用于写入操作，它是“独占锁”，写入锁只能被一个线程锁获取。注意：不能同时存在读取锁和写入锁！ReadWriteLock是一个接口。ReentrantReadWriteLock是它的实现类，ReentrantReadWriteLock包括子类ReadLock和WriteLock。

线程进入读锁的前提条件：

     1. 没有其他线程的写锁

　　2. 没有写请求，或者有写请求但调用线程和持有锁的线程是同一个线程

进入写锁的前提条件：

　　1. 没有其他线程的读锁

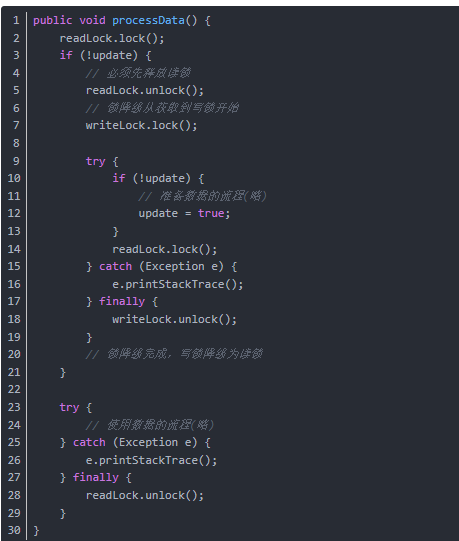
　　2. 没有其他线程的写锁

#### 锁降级

锁降级是指写锁降级成为读锁，但是需要遵循先获取写锁、获取读锁在释放写锁的次序，注意如果当前线程先获取写锁，然后释放写锁，再获取读锁这个过程不能称之为锁降级，锁降级一定要遵循那个次序。我们看一个锁降级的示例。因为数据不常变化，所以多个线程可以并发地进行数据处理，当数据变更后，如果当前线程感知到数据变化，则进行数据的准备工作，同时其他处理线程则被阻塞，直到当前线程完成数据的准备工作。

锁降级中读锁的获取是否是必要的呢？答案是必要的。主要原因是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取了读锁，既遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

ReentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。目的也是为了保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。



### CountDownLatch

#### [CountDownLatch简介](http://www.cnblogs.com/skywang12345/p/3533887.html" \l "a1)

CountDownLatch是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

**CountDownLatch和CyclicBarrier的区别**  
(01) CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。  
(02) CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。  
关于CyclicBarrier的原理，后面一章再来学习。



#### CountDownLatch使用场景

CountDownLatch典型用法1：某一线程在开始运行前等待n个线程执行完毕。将CountDownLatch的计数器初始化为n new CountDownLatch(n) ，每当一个任务线程执行完毕，就将计数器减1 countdownlatch.countDown()，当计数器的值变为0时，在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

CountDownLatch典型用法2：实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的CountDownLatch(1)，将其计数器初始化为1，多个线程在开始执行任务前首先 coundownlatch.await()，当主线程调用 countDown() 时，计数器变为0，多个线程同时被唤醒。

### CyclicBarrier

通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了。

对于失败的同步尝试，CyclicBarrier 使用了一种要么全部要么全不 (all-or-none) 的破坏模式：如果因为中断、失败或者超时等原因，导致线程过早地离开了屏障点，那么在该屏障点等待的其他所有线程也将通过 BrokenBarrierException（如果它们几乎同时被中断，则用 InterruptedException）以反常的方式离开。



#### CyclicBarrier与CountDownLatch比较

　　1）CountDownLatch:一个线程(或者多个)，等待另外N个线程完成某个事情之后才能执行；CyclicBarrier:N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

　　2）CountDownLatch:一次性的；CyclicBarrier:可以重复使用。

　　3）CountDownLatch基于AQS；CyclicBarrier基于锁和Condition。本质上都是依赖于volatile和CAS实现的。

### ****Semaphore****

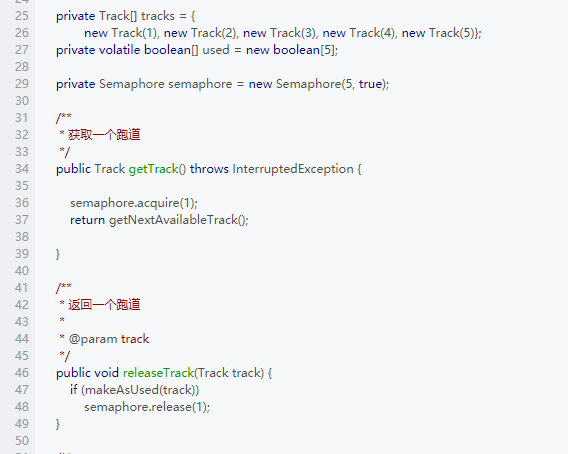
#### 简介

Semaphore是一个计数信号量，它的本质是一个"[共享锁](http://www.cnblogs.com/skywang12345/p/3505809.html)"。

信号量维护了一个信号量许可集。线程可以通过调用acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可。

#### 使用

Semaphore经常用于限制获取某种资源的线程数量。下面举个例子，比如说操场上有5个跑道，一个跑道一次只能有一个学生在上面跑步，一旦所有跑道在使用，那么后面的学生就需要等待，直到有一个学生不跑了，下面是这个例子：



# ThreadLocal