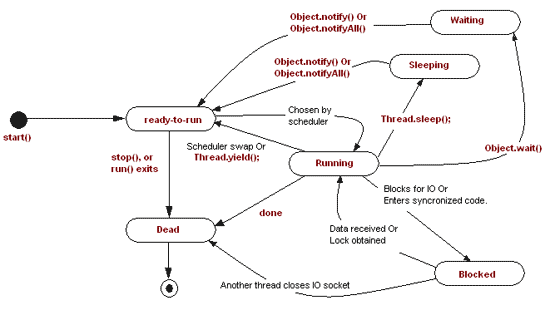
# 多线程

## 线程状态



public enum State {

//线程刚创建

NEW,

//在JVM中正在运行的线程

RUNNABLE,

//线程处于阻塞状态，等待监视锁，可以重新进行同步代码块中执行

BLOCKED,

//等待状态

WAITING,

//调用sleep() join() wait()方法可能导致线程处于等待状态

TIMED\_WAITING,

//线程执行完毕，已经退出

TERMINATED;

}

## 线程通讯的理解

线程之间如何通讯

共享内存

消息传递

线程之间如何同步

在共享内存的并发模型中，同步是显式的 synchonized

在消息传递的并发模型中，同步是隐式的

内存可见性

内存共享-堆内存

## Java线程中断理解（interrupte）

Java线程之中，一个线程的生命周期分为：初始、就绪、运行、阻塞以及结束。当然，其中也可以有四种状态，初始、就绪、运行以及结束。

一般而言，可能有三种原因引起阻塞：等待阻塞、同步阻塞以及其他阻塞（睡眠、jion或者IO阻塞）；对于Java而言，等待阻塞是调用wait方法产生的，同步阻塞则是由同步块（synchronized）产生的，睡眠阻塞是由sleep产生的，jion阻塞是由jion方法产生的。

言归正传，要中断一个Java线程，可调用线程类（Thread）对象的实例方法：interrupte()；然而interrupte()方法并不会立即执行中断操作；具体而言，这个方法只会给线程设置一个为true的中断标志（中断标志只是一个布尔类型的变量），而设置之后，则根据线程当前的状态进行不同的后续操作。如果，线程的当前状态处于非阻塞状态，那么仅仅是线程的中断标志被修改为true而已；如果线程的当前状态处于阻塞状态，那么在将中断标志设置为true后，还会有如下三种情况之一的操作：

如果是wait、sleep以及jion三个方法引起的阻塞，那么会将线程的中断标志重新设置为false，并抛出一个InterruptedException；

如果是java.nio.channels.InterruptibleChannel进行的io操作引起的阻塞，则会对线程抛出一个ClosedByInterruptedException；（待验证）

如果是轮询（java.nio.channels.Selectors）引起的线程阻塞，则立即返回，不会抛出异常。（待验证）

如果在中断时，线程正处于非阻塞状态，则将中断标志修改为true,而在此基础上，一旦进入阻塞状态，则按照阻塞状态的情况来进行处理；例如，一个线程在运行状态中，其中断标志被设置为true,则此后，一旦线程调用了wait、jion、sleep方法中的一种，立马抛出一个InterruptedException，且中断标志被清除，重新设置为false。

通过上面的分析，我们可以总结，调用线程类的interrupted方法，其本质只是设置该线程的中断标志，将中断标志设置为true，并根据线程状态决定是否抛出异常。因此，通过interrupted方法真正实现线程的中断原理是：开发人员根据中断标志的具体值，来决定如何退出线程。

一个简单的实现方式如下：



分别考虑了阻塞状态中进行中断线程和非阻塞状态中中断线程的处理方式。

最后，说明一下interrupte方法的调用，该方法可在需要中断的线程本身中调用，也可在其他线程中调用需要中断的线程对象的该方法。

## synchronized保证线程安全的原理

一段synchronized的代码被一个线程执行之前，他要先拿到执行这段代码的权限，在Java里边就是拿到某个同步对象的锁（一个对象只有一把锁）；如果这个时候同步对象的锁被其他线程拿走了，他（这个线程）就只能等了（线程阻塞在锁池等待队列中）。 取到锁后，他就开始执行同步代码(被synchronized修饰的代码）；线程执行完同步代码后马上就把锁还给同步对象，其他在锁池中等待的某个线程就可以拿到锁执行同步代码了。这样就保证了同步代码在统一时刻只有一个线程在执行。

Synchronized

修饰代码块时需要一个reference对象作为锁的对象；

修饰方法的时候默认是当前对象作为锁的对象；

修饰类的时候默认是当前类的class对象作为锁的对象；

对象监视器，编译成字节码后会出现monitorenter monitorexit，对于后续获取此对象锁的线程会放到等待同步队列中。

* + 1. synchronized 对象锁和类锁的区别

synchronized修饰非静态方法、同步代码块的synchronized (this)用法和synchronized (非this对象)的用法锁的是对象，线程想要执行对应同步代码，需要获得对象锁。

//形式1

public synchronized void Method1(){

……

}

//形式2

synchronized (this){

System.out.println("我是对象锁");

}

synchronized修饰静态方法以及同步代码块的synchronized (类.class)用法锁的是类，线程想要执行对应同步代码，需要获得类锁。

//形式1

public static synchronized void Method1() {

}

//形式2

synchronized (Test.class){

System.out.println(＂我是类锁二号＂);

}

1. 如果多线程同时访问同一类的类锁（synchronized 修饰的静态方法）以及对象锁（synchronized 修饰的非静态方法）这两个方法执行是异步的，原因：类锁和对象锁是2种不同的锁。

2. 类锁对该类的所有对象都能起作用，而对象锁不能。

## 深入理解volatile原理与使用

对声明了volatile的变量进行写操作的时候，jvm会向处理器发送一条lock前缀的命令。会把这个变量所在缓存的数据写到主内存；

在多处理器的情况下，保证各个处理器缓存一致性的 特点，就会实现缓存一致性的协议；

Volatile可以做的变量的原子性、可见性，但是做不到多复合操作的原子性->Synchonized；

## 锁

* + 1. 公平锁和非公平锁

**区别：**

所谓公平锁指的是哪个线程先运行，那就可以先得到锁。非公平锁是不管线程是否是先运行，都是随机获得锁的。

公平和非公平锁的队列都基于锁内部维护的一个双向链表，表结点Node的值就是每一个请求当前锁的线程。公平锁则在于每次都是依次从队首取值。

非公平锁在等待锁的过程中， 如果有任意新的线程妄图获取锁，都是有很大的几率直接获取到锁的。

**非公平锁：**

简化版的步骤：（非公平锁的核心）

基于CAS尝试将state（锁数量）从0设置为1

A、如果设置成功，设置当前线程为独占锁的线程；

B、如果设置失败，还会再获取一次锁数量，

B1、如果锁数量为0，再基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

**公平锁：**

获取公平锁的步骤：

B1、如果锁数量为0，如果当前线程是等待队列中的头节点，基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者当前线程不是等待队列中的头节点或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

公平锁的实现机理在于每次有线程来抢占锁的时候，都会检查一遍有没有等待队列，如果有， 当前线程会执行如下步骤：

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

其中hasQueuedPredecessors是用于检查是否有等待队列的。

* + 1. 可重入锁

又称递归锁，同一线程外层函数获得锁后，内层递归函数仍有获取该锁的代码但是不受影响。

一个可重入的互斥锁 Lock，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。

ReentrantLock 将由最近成功获得锁，并且还没有释放该锁的线程所拥有。当锁没有被另一个线程所拥有时，调用 lock 的线程将成功获取该锁并返回。如果当前线程已经拥有该锁，此方法将立即返回。可以使用 isHeldByCurrentThread() 和 getHoldCount() 方法来检查此情况是否发生。

此类的构造方法接受一个可选的公平 参数。当设置为 true 时，在多个线程的争用下，这些锁倾向于将访问权授予等待时间最长的线程。否则此锁将无法保证任何特定访问顺序。与采用默认设置（使用不公平锁）相比，使用公平锁的程序在许多线程访问时表现为很低的总体吞吐量（即速度很慢，常常极其慢），但是在获得锁和保证锁分配的均衡性时差异较小。不过要注意的是，公平锁不能保证线程调度的公平性。因此，使用公平锁的众多线程中的一员可能获得多倍的成功机会，这种情况发生在其他活动线程没有被处理并且目前并未持有锁时。还要注意的是，未定时的 tryLock 方法并没有使用公平设置。因为即使其他线程正在等待，只要该锁是可用的，此方法就可以获得成功。

/\*\*

\* Fair version of tryAcquire. Don't grant access unless

\* recursive call or no waiters or is first.

\*/

**protected** **final** **boolean** tryAcquire(**int** acquires) {

**final** Thread current = Thread.*currentThread*();

**int** c = *getState*();

**if** (c == 0) {

**if** (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

**return** **true**;

}

}

**else** **if** (current == getExclusiveOwnerThread()) {

**int** nextc = c + acquires;

**if** (nextc < 0)

**throw** **new** Error("Maximum lock count exceeded");

setState(nextc);

**return** **true**;

}

**return** **false**;

}

* + 1. 共享锁
    2. 读写锁

此类具有以下属性：

* 获取顺序

此类不会将读取者优先或写入者优先强加给锁访问的排序。但是，它确实支持可选的公平 策略。

**>**非公平模式（默认）

当非公平地（默认）构造时，未指定进入读写锁的顺序，受到 reentrancy 约束的限制。连续竞争的非公平锁可能无限期地推迟一个或多个 reader 或 writer 线程，但吞吐量通常要高于公平锁。

**>**公平模式

当公平地构造线程时，线程利用一个近似到达顺序的策略来争夺进入。当释放当前保持的锁时，可以为等待时间最长的单个 writer 线程分配写入锁，如果有一组等待时间大于所有正在等待的 writer 线程 的 reader 线程，将为该组分配写入锁。

如果保持写入锁，或者有一个等待的 writer 线程，则试图获得公平读取锁（非重入地）的线程将会阻塞。直到当前最旧的等待 writer 线程已获得并释放了写入锁之后，该线程才会获得读取锁。当然，如果等待 writer 放弃其等待，而保留一个或更多 reader 线程为队列中带有写入锁自由的时间最长的 waiter，则将为那些 reader 分配读取锁。

试图获得公平写入锁的（非重入地）的线程将会阻塞，除非读取锁和写入锁都自由（这意味着没有等待线程）。（注意，非阻塞 ReentrantReadWriteLock.ReadLock.tryLock() 和 ReentrantReadWriteLock.WriteLock.tryLock() 方法不会遵守此公平设置，并将获得锁（如果可能），不考虑等待线程）。

重入

此锁允许 reader 和 writer 按照 ReentrantLock 的样式重新获取读取锁或写入锁。在写入线程保持的所有写入锁都已经释放后，才允许重入 reader 使用它们。

此外，writer 可以获取读取锁，但反过来则不成立。在其他应用程序中，当在调用或回调那些在读取锁状态下执行读取操作的方法期间保持写入锁时，重入很有用。如果 reader 试图获取写入锁，那么将永远不会获得成功。

锁降级

重入还允许从写入锁降级为读取锁，其实现方式是：先获取写入锁，然后获取读取锁，最后释放写入锁。但是，从读取锁升级到写入锁是不可能的。

锁获取的中断

读取锁和写入锁都支持锁获取期间的中断。

Condition 支持

写入锁提供了一个 Condition 实现，对于写入锁来说，该实现的行为与 ReentrantLock.newCondition() 提供的 Condition 实现对 ReentrantLock 所做的行为相同。当然，此 Condition 只能用于写入锁。

读取锁不支持 Condition，readLock().newCondition() 会抛出 UnsupportedOperationException。

监测

此类支持一些确定是保持锁还是争用锁的方法。这些方法设计用于监视系统状态，而不是同步控制。

此类行为的序列化方式与内置锁的相同：反序列化的锁处于解除锁状态，无论序列化该锁时其状态如何。

**public** **class** CachedData {

**volatile** **boolean** cacheValid;

ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();

**void** processCachedData(){

lock.readLock().lock();

**if**(!cacheValid){

//获取写锁之前必须释放锁

lock.readLock().unlock();

lock.writeLock().lock();

//重新检查状态，因为另一个线程可能获得了写锁并且已经改变了状态

**if** (!cacheValid) {

data = ...

cacheValid = **true**;

}

//释放写锁之前 通过获取读锁的方式降级

//获取读锁。在写锁持有期间获取读锁

//此处获取读锁，是为了防止，当释放写锁后，又有一个线程T获取锁，对数据进行改变，

//而当前线程下面对改变的数据无法感知。

//如果获取了读锁，则线程T则被阻塞，直到当前线程释放了读锁，那个T线程才有可能获取写锁。

lock.readLock().lock();

lock.writeLock().unlock(); //释放写锁，仍然持有读锁

}

use(data);

lock.readLock().unlock();

}

}

**锁降级：**

锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

锁降级的必要性：

锁降级中读锁的获取是否必要呢？答案是必要的。主要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。如果当前线程获取读锁，即遵循锁降级的步骤，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。

这里要着重讲一讲“无法感知”是什么意思：

也就是说，在另一个线程（假设叫线程1）修改数据的那一个瞬间，当前线程（线程2）是不知道数据此时已经变化了，但是并不意味着之后线程2使用的数据就是旧的数据，相反线程2使用还是被线程1更新之后的数据。也就是说，就算我不使用锁降级，程序的运行结果也是正确的（这是因为锁的机制和volatile关键字相似）。

那么为什么还要锁降级呢，其实目的是为了减少线程的阻塞唤醒。明显当不使用锁降级，线程2修改数据时，线程1自然要被阻塞，而使用锁降级时则不会。“感知”其实是想强调读的实时连续性，但是却容易让人误导为强调数据操作。

## AbstractQueuedSynchronizer(AQS)详解

0、同步器的基本功能

一个同步器至少需要包含两个功能：

1. 获取同步状态

如果允许，则获取锁，如果不允许就阻塞线程，直到同步状态允许获取。

2. 释放同步状态

修改同步状态，并且唤醒等待线程。

根据作者论文， aqs 同步机制同时考虑了如下需求：

1. 独占锁和共享锁两种机制。

2. 线程阻塞后，如果需要取消，需要支持中断。

3. 线程阻塞后，如果有超时要求，应该支持超时后中断的机制。

1、同步状态的获取与释放

AQS实现了一个同步器的基本结构，下面以独占锁与共享锁分开讨论，来说明AQS怎样实现获取、释放同步状态。

**1.1、独占模式**

独占获取： tryAcquire 本身不会阻塞线程，如果返回 true 成功就继续，如果返回 false 那么就阻塞线程并加入阻塞队列。

/\*\*

\*以独占模式获取对象，忽略中断。通过至少调用一次 tryAcquire(int) 来实现此方法，并在成功时返回。否则在成功之前，一直调用 tryAcquire(int) 将线程加入队列，线程可能重复被阻塞或不被阻塞。可以使用此方法来实现 Lock.lock() 方法。

\*\*/

**public** **final** **void** acquire(**int** arg) {

**if** (!**tryAcquire**(arg) &&

**acquireQueued**(addWaiter(Node.***EXCLUSIVE***), arg))

*selfInterrupt*();

}

独占且可中断模式获取：支持中断取消

/\*\*

以独占模式获取对象，如果被中断则中止。通过先检查中断状态，然后至少调用一次 tryAcquire(int) 来实现此方法，并在成功时返回。否则在成功之前，或者线程被中断之前，一直调用 tryAcquire(int) 将线程加入队列，线程可能重复被阻塞或不被阻塞。可以使用此方法来实现 Lock.lockInterruptibly() 方法。

\*\*/

**public** **final** **void** acquireInterruptibly(**int** arg)

**throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**if** (!tryAcquire(arg))

doAcquireInterruptibly(arg);

}

独占且支持超时模式获取： 带有超时时间，如果经过超时时间则会退出。

/\*\*

试图以独占模式获取对象，如果被中断则中止，如果到了给定超时时间，则会失败。通过先检查中断状态，然后至少调用一次 tryAcquire(int) 来实现此方法，并在成功时返回。否则，在成功之前、线程被中断之前或者到达超时时间之前，一直调用 tryAcquire(int) 将线程加入队列，线程可能重复被阻塞或不被阻塞。可以用此方法来实现 Lock.tryLock(long, TimeUnit) 方法。

\*\*/

独占模式释放：释放成功会唤醒后续节点

/\*\*

\*\*以独占模式释放对象。如果 tryRelease(int) 返回 true，则通过消除一个或多个线程的阻塞来实现此方法。可以使用此方法来实现 Lock.unlock() 方法。

\*\*/

1.2、共享模式

共享模式获取

/\*\*

以共享模式获取对象，忽略中断。通过至少先调用一次 tryAcquireShared(int) 来实现此方法，并在成功时返回。否则在成功之前，一直调用 tryAcquireShared(int) 将线程加入队列，线程可能重复被阻塞或不被阻塞。

\*\*/

**public** **final** **void** acquireShared(**int** arg) {

**if** (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

3、状态获取、释放成功或失败的后续行为：线程的阻塞、唤醒机制

有别于wait和notiry。这里利用 jdk1.5 开始提供的 LockSupport.park() 和 LockSupport.unpark() 的本地方法实现，实现线程的阻塞和唤醒。

得到锁的线程禁用(park)和唤醒(unpark)，也是直接native实现（这几个native方法的实现代码在hotspot\src\share\vm\prims\unsafe.cpp文件中，但是关键代码park的最终实现是和操作系统相关的，比如windows下实现是在os\_windows.cpp中，有兴趣的同学可以下载jdk源码查看）。唤醒一个被park()线程主要手段包括以下几种

1. 其他线程调用以被park()线程为参数的unpark(Thread thread).

2. 其他线程中断被park()线程,如waiters.peek().interrupt();waiters为存储线程对象的队列.

3. 不知原因的返回。

**4、线程阻塞队列的维护**

阻塞线程节点队列 CHL Node queue 。

根据论文里描述， AQS 里将阻塞线程封装到一个内部类 Node 里。并维护一个 CHL Node FIFO 队列。 CHL队列是一个非阻塞的 FIFO 队列，也就是说往里面插入或移除一个节点的时候，在并发条件下不会阻塞，而是通过自旋锁和 CAS 保证节点插入和移除的原子性。实现无锁且快速的插入。关于非阻塞算法可以参考 Java 理论与实践: 非阻塞算法简介 。CHL队列对应代码如下：

**5、小结**

从源码可以看出AQS实现基本的功能：

1.同步器基本范式、结构

2.线程的阻塞、唤醒机制

3.线程阻塞队列的维护

AQS虽然实现了acquire，和release方法，但是里面调用的tryAcquire和tryRelease是由子类来定制的。可以认为同步状态的维护、获取、释放动作是由子类实现的功能，而动作成功与否的后续行为时有AQS框架来实现

**2、不同实现类的Sync与State：**

基于AQS构建的Synchronizer包括ReentrantLock,Semaphore,CountDownLatch, ReetrantRead WriteLock,FutureTask等，这些Synchronizer实际上最基本的东西就是原子状态的获取和释放，只是条件不一样而已。

## 并发等待条件（Condition）

前言

前面介绍了排它锁，共享锁的实现机制，本篇继续学习AQS中的另外一个内容-Condition。想必学过java的都知道Object.wait和Object.notify，同时也应该知晓这两个方法的使用离不开synchronized关键字。synchronized是jvm级别提供的同步原语，它的实现机制隐藏在jvm实现中。作为Lock系列功能中的Condition，就是用来实现类似 Object.wait和Object.notify 对应功能的。

为了更好的理解Lock和Condition的使用场景，下面我们先来实现这样一个功能：有多个生产者，多个消费者，一个产品容器，我们假设容器最多可以放3个产品，如果满了，生产者需要等待产品被消费，如果没有产品了，消费者需要等待。我们的目标是一共生产10个产品，最终消费10个产品，如何在多线程环境下完成这一挑战呢？

深入理解Condition的实现原理

上面的示例只是为了展示 Lock结合Condition可以实现的一种经典场景，在有了感性的认识之后，我们将一步一步来观察Lock和Condition是如何协作完成这一任务的，这也是本篇的核心内容。

为了更好的理解和演示这一个过程，我们使用到的锁是使用公平策略模式的，我们会使用上面例子运作的流程。我们会使用到3个生产线程，3个消费线程，分别表示 p1、p2、p3和c1、c2、c3。

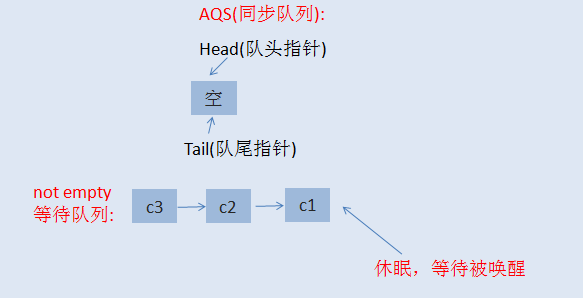
Condition的内部实现是使用节点链来实现的，每个条件实例对应一个节点链，我们有notEmpty 和 notFull 两个条件实例，所以会有两个等待节点链。

一切准备就绪 ，开始我们的探索之旅。

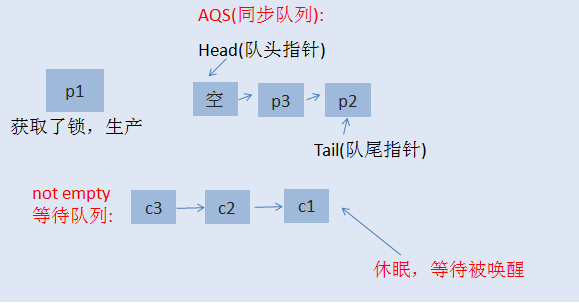
1、线程c3执行，然后发现没有产品可以消费，执行 notEmpty.await，进入等待队列中等候。



2、线程c2和线程c1执行，然后发现没有产品可以消费，执行 notEmpty.await，进入等待队列中等候。

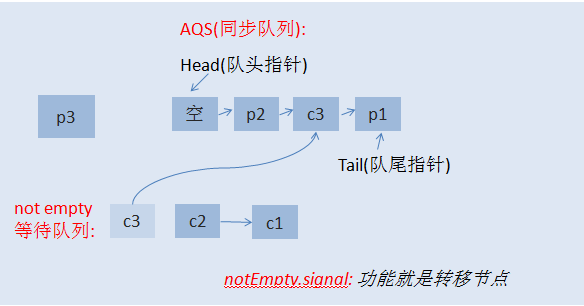


3、 线程 p1 启动，得到了锁，p1开始生产产品，这时候p3抢在p2之前，执行了lock操作，结果p2和p3都处于等待状态，入同步队列等待。



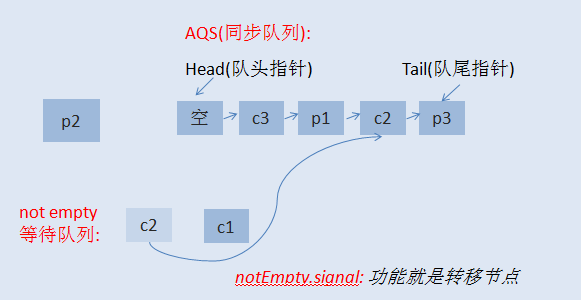
注意，本例中我们使用的是公平策略模式下的排它锁，由于p3抢先执行取锁操作，所以虽然p2和p3都被阻塞了，但是p3会优先被唤醒 。

4、这会，p1生产完毕，通知 not empty等待队列，可以唤醒一个等待线程节点了，然后释放了锁，释放锁会导致p3被唤醒，然后p1进入下一个循环，进入同步队列。

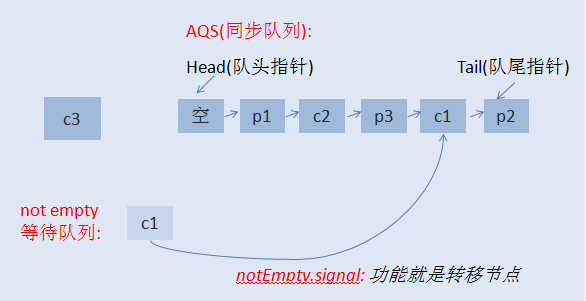


事情开始变得有趣了，p1执行一次生产后，执行了 notEmpty.signal，其效果就是把 not empty等待列表中的头节点，即c3节点移到同步等待列队中，重新参与抢占锁。

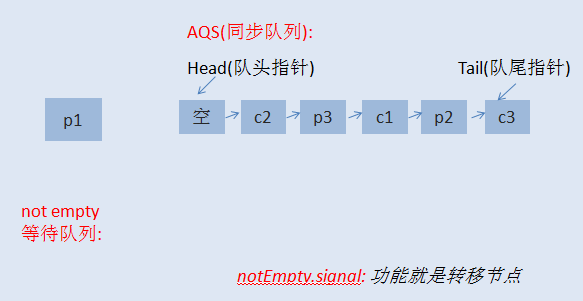
5、p3生产完了产品后，继续notEmpty.signal，同时释放锁，释放锁后会唤醒p2线程，然后p3在下一轮尝试获取锁的时候，再次入队。



6、接着，p2继续生产，生产后执行 notEmpty.signal，同时释放锁，释放锁后唤醒c3线程，然后p2在下一轮尝试取锁的时候，入列。



7、c3进行消费，你可以看到，现在 not empty等待列队中已经没有等待节点了，由于我们使用的是公平策略排它锁，这就会导致同步队列中的节点一个接着一个执行，而目前同步队列中的节点排列为一生产，一消费，这不难可以知道，接下来代码已经不会进入 wait条件了，所以一个一个轮流执行就是，比如c3，执行完了，继续notFull.signal(); 然后释放锁，入队，这里要明白，notFull.signal();这句代码其实没有作用了，因为 not full等待队列中没有任何等待线程节点。 c3执行后，状态如下图所示：



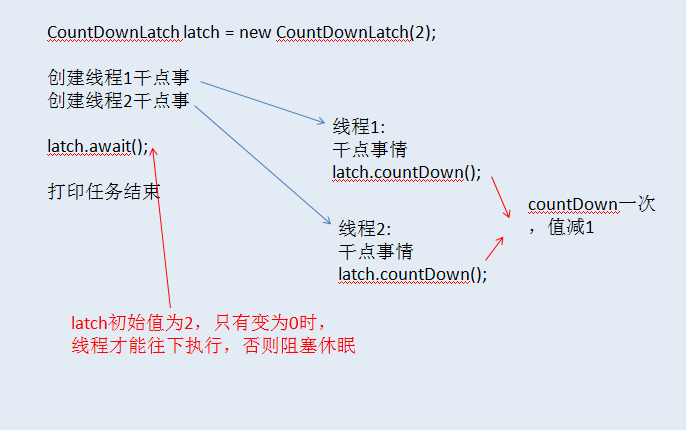
8、后面的事情我想大家都可以想得出来是怎样一步一步交替执行的了。

总结

本篇基于一个实例来演示结合Lock和Condition如何实现生产-消费模式，而且只讨论一种可能执行的流程，是想更简单的表述AQS底层是如何实现的。基于上面这个演示过程，针对其它的执行流程，其原来也是一样的。Condition内部使用一个节点链来保存所有 wait状态的线程，当对应条件被signal的时候，就会把等待节点转移到同步队列中，继续竞争锁。原理其实并不复杂，有兴趣的朋友可以翻阅源码。

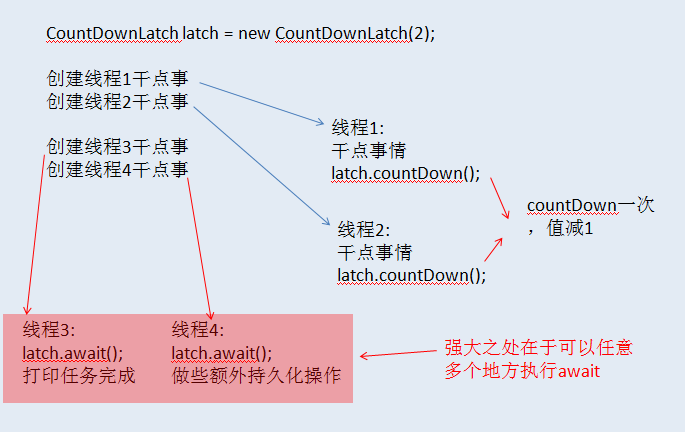
## 等待多线程完成CountDownLatch

CountDownLatch是java5中新增的一个并发工具类，其使用非常简单，下面通过伪代码简单看一下使用方式：



这是一个使用CountDownLatch非常简单的例子，创建的时候，需要指定一个初始状态值，本例为2，主线程调用 latch.await时，除非latch状态值为0，否则会一直阻塞休眠。当所有任务执行完后，主线程唤醒，最终执行打印动作。

以上只是一个最简单的例子，接着咱们再来看一个，这回，咱们想要在任务执行完后做更多的事情，如下图所示：



这一次，在线程3和线程4中，分别调用了latch.await()，当latch状态值为0时，这两个线程将会继续执行任务，但是顺序性是无法保证的。

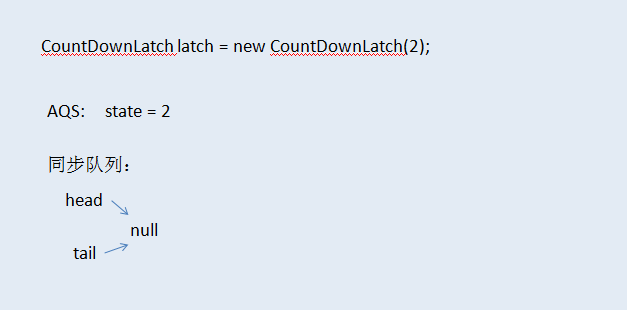
CountDownLatch的方便之处在于，你可以在一个线程中使用，也可以在多个线程上使用，一切只依据状态值，这样便不会受限于任何的场景。

**java共享锁模型**

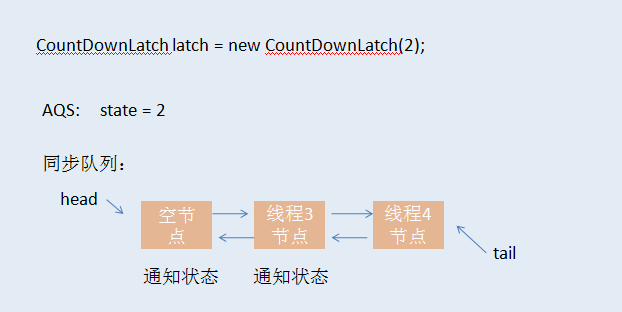
在java5提供的并发包下，有一个AbstractQueuedSynchronizer抽象类，也叫AQS，此类根据大部分并发共性作了一些抽象，便于开发者实现如排他锁，共享锁，条件等待等更高级的业务功能。它通过使用CAS和队列模型，出色的完成了抽象任务，在此向Doug Lea致敬。

AQS比较抽象，并且是优化精简的代码，如果一头扎进去，可能会比较容易迷失。本篇只解说CountDownLatch中使用到的共享锁模型。

我们以CountDownLatch第二个例子作为案例来分析一下，一开始，我们创建了一个CountDownLatch实例，



此时，AQS中，状态值state=2，对于 CountDownLatch 来说，state=2表示所有调用await方法的线程都应该阻塞，等到同一个latch被调用两次countDown后才能唤醒沉睡的线程。接着线程3和线程4执行了 await方法，这会的状态图如下:



注意，上面的通知状态是节点的属性，表示该节点出队后，必须唤醒其后续的节点线程。当线程1和线程2分别执行完latch.countDown方法后，会把state值置为0，此时，通过CAS成功置为0的那个线程将会同时承担起唤醒队列中第一个节点线程的任务，从上图可以看出，第一个节点即为线程3，当线程3恢复执行之后，其发现状态值为通知状态，所以会唤醒后续节点，即线程4节点，然后线程3继续做自己的事情，到这里，线程3和线程4都已经被唤醒，CountDownLatch功成身退。

上面的流程，如果落实到代码，把 state置为0的那个线程，会判断head指向节点的状态，如果为通知状态，则唤醒后续节点，即线程3节点，然后head指向线程3节点，head指向的旧节点会被删除掉。当线程3恢复执行后，发现自身为通知状态，又会把head指向线程4节点，然后删除自身节点，并唤醒

线程4。

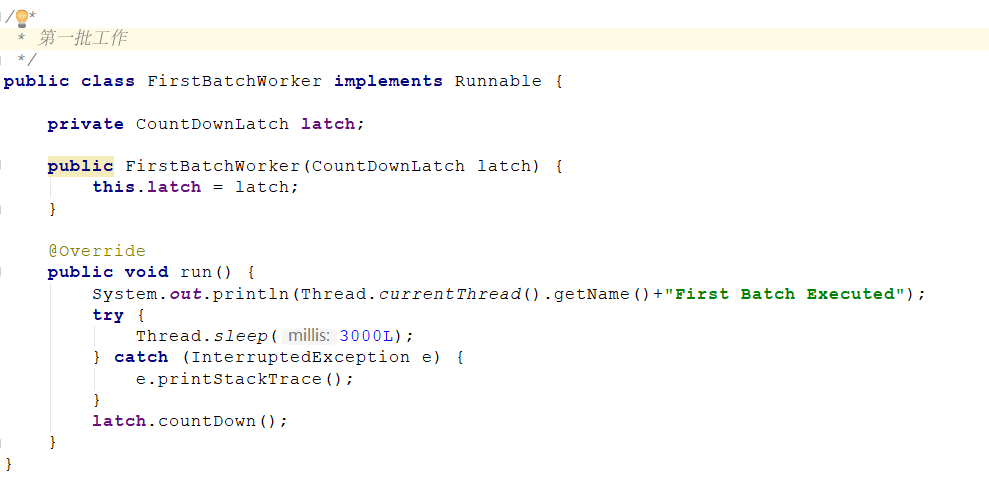
这里可能读者会有个疑问，线程节点的状态是什么时候设置上去的。其实，一个线程在阻塞之前，就会把它前面的节点设置为通知状态，这样便可以实现链式唤醒机制了。

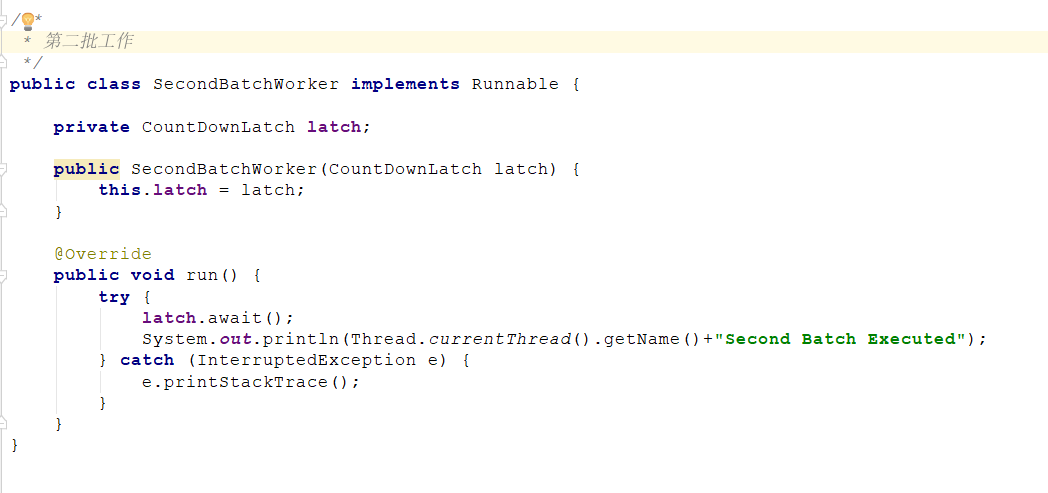
结束语

本篇从CountDownLatch入手讲解AQS中的共享锁模式，主要是由CountDownLatch的实现相对简单，但却实现了共享锁模型，如果在理解了模型的基础上，从CountDownLatch入手来看AQS关于共享锁的代码还比较好看懂，在看的时候，建议以看懂大致内容为主，学习其设计的思路，不要陷入所有条件处理细节中，多线程环境中，对与错有时候不是那么容易看出来的。

例子：

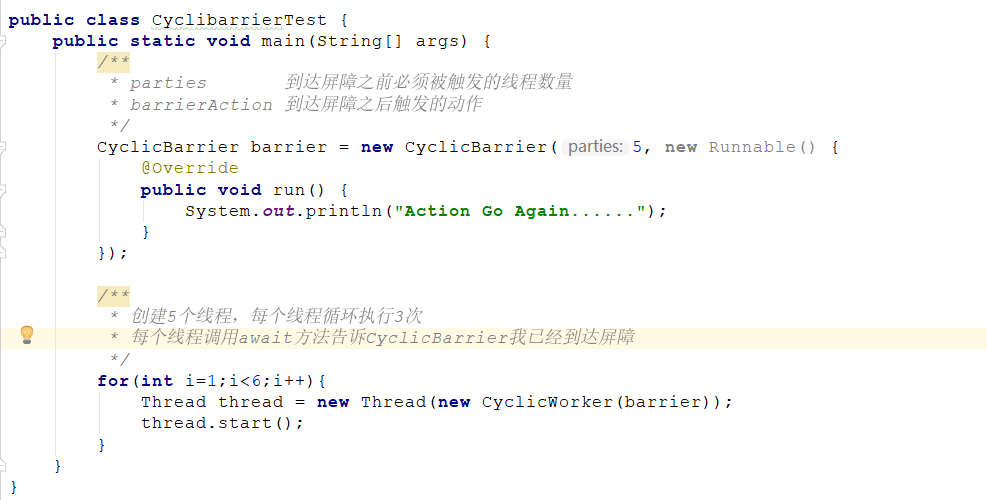


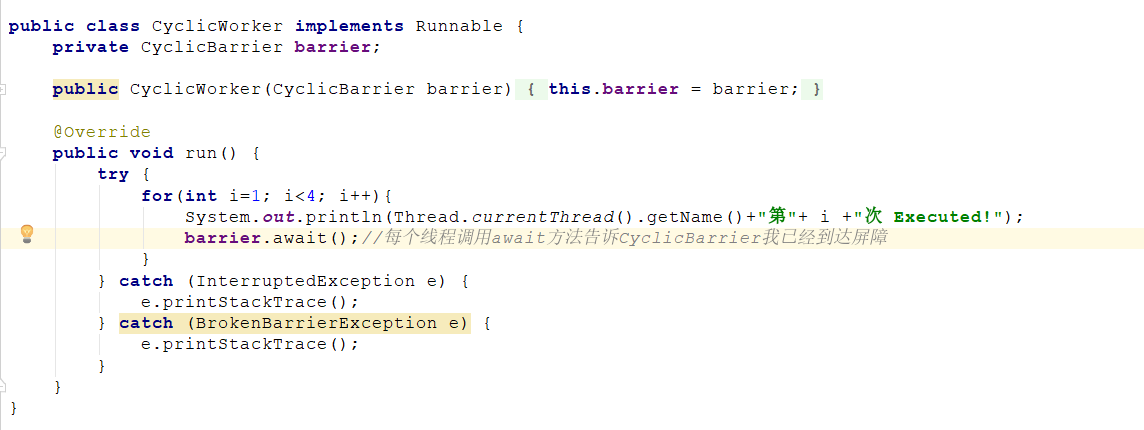




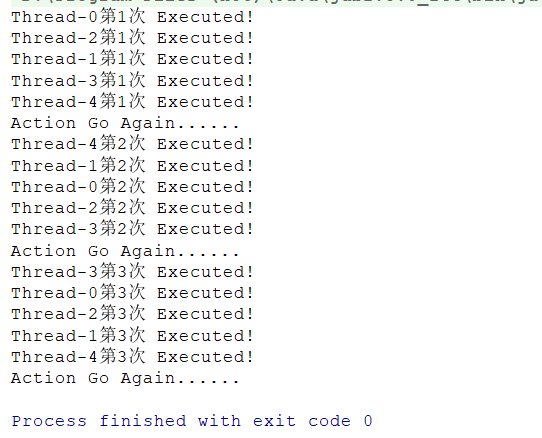
## 同步屏障CyclicBarrier

让一组线程达到一个屏障时被阻塞，直到最后一个线程到达屏障，所有被屏障拦截的线程才会继续运行。





输出结果：



使用场景：

多线程计算数据，最后合并计算结果；

排队；

## CountDownLatch与CyclicBarrier区别

CountDownLatch的计数器只能使用1次，而CyclicBarrier的计数器可以使用reset方法重置；

CountDownLatch的基本操作组合是countDown/await；

CyclicBarrier的基本操作组合是await()，当所有伙伴都调用了await()，才会继续调用任务，并进行自动重置；

## Java阻塞队列

* ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

*先进先出（FIFO），默认情况下不保证线程公平的访问队列，但是可以创建时指定*

* LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

*此队列默认和最大长度为Integer.MAX\_VALUE，此队列按照先进先出的原则对元素进行排序*

* PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

*默认采用自然顺序升序排列，也可以自定义实现compareTo()来指定元素的排序规则，或者初始化PriorityBlockingQueue时指定构造参数Comparator来对元素进行排序。*

* DelayQueue：一个使用优先级队列实现的无界阻塞队列。

*队列使用PriorityQueue实现，队列中的元素必须实现Delayed接口，用于缓存系统设计、定时任务调度等*

* SynchronousQueue：一个不存储元素的阻塞队列。
* LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
* LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

实现原理：使用通知模式实现

## 简单阻塞队列实现

**package** cn.gov.pbc.java.concurrency.queue;  
**import** java.util.Arrays;  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class** SimpleBlockingQueue<E> {  
 **final** Object **items**[];  
 **int count**;*//队列中元素的个数* **int putIndex**;  
 **int takeIndex**;  
 **private final** ReentrantLock **lock** = **new** ReentrantLock();  
 *//等待放入元素的条件* **private final** Condition **notFull** = **lock**.newCondition();  
 *//等待去元素的条件* **private final** Condition **notEmpty** = **lock**.newCondition();  
  
 **public** SimpleBlockingQueue(**int** capacity) {  
 **this**.**items**=**new** Object[capacity];  
 }  
  
 **public void** put(E e){  
 **final** ReentrantLock lock = **this**.**lock**;  
 **try** {  
 lock.lockInterruptibly();  
 **while** (**count**==**items**.**length**){  
 System.***out***.println(**"放入线程阻塞....."**);  
 **notFull**.await();  
 }  
 insert(e);  
 } **catch** (InterruptedException e1) {  
 e1.printStackTrace();  
 }**finally** {  
 lock.unlock();  
 }  
 }  
  
 **public void** take(){  
 **final** ReentrantLock lock = **this**.**lock**;  
 **try** {  
 lock.lockInterruptibly();  
 **while**(**count**==0){  
 System.***out***.println(**"取出线程阻塞....."**);  
 **notEmpty**.await();  
 }  
 extract();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }**finally** {  
 lock.unlock();  
 }  
  
 }  
  
 */\*\*  
 \* 放入元素  
 \** ***@param x*** *\*/* **private void** insert(E x){  
 **final** Object items[] = **this**.**items**;  
 items[**putIndex**] = x;  
 **if**(++**putIndex**==items.**length**)  
 **putIndex**=0;  
 **count**++;  
 **notEmpty**.signal();  
 }  
  
 */\*\*  
 \* 取出元素  
 \*/* **private void** extract(){  
 **final** Object items[] = **this**.**items**;  
 E x = (E) items[**takeIndex**];  
 items[**takeIndex**] = **null**;  
 **if**(++**takeIndex**==items.**length**)  
 **takeIndex** = 0;  
 **count**--;  
 **notFull**.signal();  
 System.***out***.println(**"取出一个元素完成....."**);  
 }  
  
 **public int** size() {  
 **final** ReentrantLock lock = **this**.**lock**;  
 lock.lock();  
 **try** {  
 **return count**;  
 } **finally** {  
 lock.unlock();  
 }  
 }  
  
 @Override  
 **public** String toString() {  
 **return "SimpleBlockingQueue{"** +  
 **"items="** + Arrays.*toString*(**items**) +  
 **'}'**;  
 }  
}

测试程序

**public class** SimpleBlockingQueueTest {  
 **public static void** main(String[] args) {  
 SimpleBlockingQueue queue = **new** SimpleBlockingQueue(5);  
  
 queue.put(**"1111"**);  
 queue.put(**"2222"**);  
 queue.put(**"3333"**);  
 queue.put(**"4444"**);  
 queue.put(**"5555"**);  
 System.***out***.println(queue +**"==="**+ queue.size());  
  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(**"新启动线程，休息一会，取出一个元素....."**);  
 **try** {  
 Thread.*sleep*(5000L);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"取出元素之前......"**+queue+**"===="**+queue.size());  
 queue.take();  
 System.***out***.println(**"取出元素之后......"**+queue+**"===="**+queue.size());  
 }  
 }, **"mythread"**).start();  
  
 queue.put(**"6666"**);  
 System.***out***.println(queue +**"==="**+ queue.size());  
 }  
}

**1.** 多个线程同时读写，读线程的数量远远⼤于写线程，你认为应该如何解决 并发的问题？你会选择加什么样的锁？

**2.** JAVA的AQS是否了解，它是⼲嘛的？

**3.**除了synchronized关键字之外，你是怎么来保障线程安全的？

**4.** 什么时候需要加volatile关键字？它能保证线程安全吗？

**5.** 线程池内的线程如果全部忙，提交⼀个新的任务，会发⽣什么？队列全部 塞满了之后，还是忙，再提交会发⽣什么？

**6.** Tomcat本身的参数你⼀般会怎么调整？

**7.** synchronized关键字锁住的是什么东⻄？在字节码中是怎么表示的？在内存中的对象上表现为什么？

**8.** wait/notify/notifyAll⽅法需不需要被包含在synchronized块中？这是为什 么？

**9.** ExecutorService你⼀般是怎么⽤的？是每个service放⼀个还是⼀个项⽬ ⾥⾯放⼀个？有什么好处？

## Lock接口

## ThreadLocal 使用及实现原理

首先明确一个概念，那就是ThreadLocal并不是用来并发控制访问一个共同对象，而是为了给每个线程分配一个只属于该线程的对象（这么粗暴的解释可能还不太准确），更准确的说是为了实现线程间的数据隔离。而ThreadLocal应用场景更多是想共享一个变量，但是该变量又不是线程安全的，那么可以用ThreadLocal维护一个线程一个实例。有时候ThreadLocal也可以用来避免一些参数传递，通过ThreadLocal来访问对象。

## CopyOnWrite容器

Copy-On-Write是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。

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。

CopyOnWrite并发容器用于读多写少的并发场景。

缺点：

内存占用问题

数据一致性问题

## 线程池的作用

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。   
第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。   
第三：提高线程的可管理性。   
常用线程池：ExecutorService 是主要的实现类，其中常用的有 Executors.newSingleThreadPool(),newFixedThreadPool(),newcachedTheadPool(),newScheduledThreadPool()。