# Java并发编程：Thread类的使用

http://www.cnblogs.com/dolphin0520/p/3920357.html

## 线程的状态

线程从创建到最终的消亡，要经历若干个状态。一般来说，线程包括以下这几个状态：创建(new)、就绪(runnable)、运行(running)、阻塞(blocked)、time waiting、waiting、消亡（dead）。

当需要新起一个线程来执行某个子任务时，就创建了一个线程。但是线程创建之后，不会立即进入就绪状态，因为线程的运行需要一些条件（比如内存资源，在前面的JVM内存区域划分，程序计数器、Java栈、本地方法栈都是线程私有的，所以需要为线程分配一定的内存空间），只有线程运行需要的所有条件满足了，才进入就绪状态。

当线程进入就绪状态后，不代表立刻就能获取CPU执行时间，也许此时CPU正在执行其他的事情，因此它要等待。当得到CPU执行时间之后，线程便真正进入运行状态。

线程在运行状态过程中，可能有多个原因导致当前线程不继续运行下去，比如用户主动让线程睡眠（睡眠一定的时间之后再重新执行）、用户主动让线程等待，或者被同步块给阻塞，此时就对应着多个状态：time waiting（睡眠或等待一定的事件）、waiting（等待被唤醒）、blocked（阻塞）。

当由于突然中断或者子任务执行完毕，线程就会被消亡。

## 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。举个简单的例子：比如一个线程A正在读取一个文件的内容，正读到文件的一半，此时需要暂停线程A，转去执行线程B，当再次切换回来执行线程A的时候，我们不希望线程A又从文件的开头来读取。

因此需要记录线程A的运行状态，那么会记录哪些数据呢？因为下次恢复时需要知道在这之前当前线程已经执行到哪条指令了，所以需要记录程序计数器的值，另外比如说线程正在进行某个计算的时候被挂起了，那么下次继续执行的时候需要知道之前挂起时变量的值时多少，因此需要记录CPU寄存器的状态。所以一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

说简单点的：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。

## Thread类中的方法

Thread类实现了Runnable接口，在Thread类中，有一些比较关键的属性，比如name是表示Thread的名字，可以通过Thread类的构造器中的参数来指定线程名字，priority表示线程的优先级（最大值为10，最小值为1，默认值为5），daemon表示线程是否是守护线程，target表示要执行的任务。

下面是Thread类中常用的方法：

以下是关系到线程运行状态的几个方法：

### 1）start方法

start()用来启动一个线程，当调用start方法后，系统才会开启一个新的线程来执行用户定义的子任务，在这个过程中，会为相应的线程分配需要的资源。

### 2）run方法

　　run()方法是不需要用户来调用的，当通过start方法启动一个线程之后，当线程获得了CPU执行时间，便进入run方法体去执行具体的任务。注意，继承Thread类必须重写run方法，在run方法中定义具体要执行的任务。

### 3）sleep方法

　　sleep方法有两个重载版本：

sleep(long millis)     //参数为毫秒

sleep(long millis,int nanoseconds)    //第一参数为毫秒，第二个参数为纳秒

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。

注意，如果调用了sleep方法，必须捕获InterruptedException异常或者将该异常向上层抛出。当线程睡眠时间满后，不一定会立即得到执行，因为此时可能CPU正在执行其他的任务。所以说调用sleep方法相当于让线程进入阻塞状态。

### 4）yield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。

### 5）join方法

　　join方法有三个重载版本：

join()

join(long millis)     //参数为毫秒

join(long millis,int nanoseconds)    //第一参数为毫秒，第二个参数为纳秒

假如在main线程中，调用thread.join方法，则main方法会等待thread线程执行完毕或者等待一定的时间。如果调用的是无参join方法，则等待thread执行完毕，如果调用的是指定了时间参数的join方法，则等待一定的时间。

可以看出，当调用thread1.join()方法后，main线程会进入等待，然后等待thread1执行完之后再继续执行。

　　实际上调用join方法是调用了Object的wait方法。wait方法会让线程进入阻塞状态，并且会释放线程占有的锁，并交出CPU执行权限。

　　由于wait方法会让线程释放对象锁，所以join方法同样会让线程释放对一个对象持有的锁。具体的wait方法使用在后面文章中给出。

### 6）interrupt方法

interrupt，顾名思义，即中断的意思。单独调用interrupt方法可以使得处于阻塞状态的线程抛出一个异常，也就说，它可以用来中断一个正处于阻塞状态的线程；另外，通过interrupt方法和isInterrupted()方法来停止正在运行的线程。

通过interrupt方法可以中断处于阻塞状态的线程，直接调用interrupt方法不能中断正在运行中的线程（非阻塞线程）。但是如果配合isInterrupted()能够中断正在运行的线程，因为调用interrupt方法相当于将中断标志位置为true，那么可以通过调用isInterrupted()判断中断标志是否被置位来中断线程的执行。但是一般情况下不建议通过这种方式来中断线程，一般会在MyThread类中增加一个属性 isStop来标志是否结束while循环，然后再在while循环中判断isStop的值。

### 7）stop方法(弃用)

stop方法已经是一个废弃的方法，它是一个不安全的方法。因为调用stop方法会直接终止run方法的调用，并且会抛出一个ThreadDeath错误，如果线程持有某个对象锁的话，会完全释放锁，导致对象状态不一致。所以stop方法基本是不会被用到的。

### 8）destroy方法(弃用)

destroy方法也是废弃的方法。基本不会被使用到。

以下是关系到线程属性的几个方法：

### 1）getId

　　用来得到线程ID

### 2）getName和setName

　　用来得到或者设置线程名称。

### 3）getPriority和setPriority

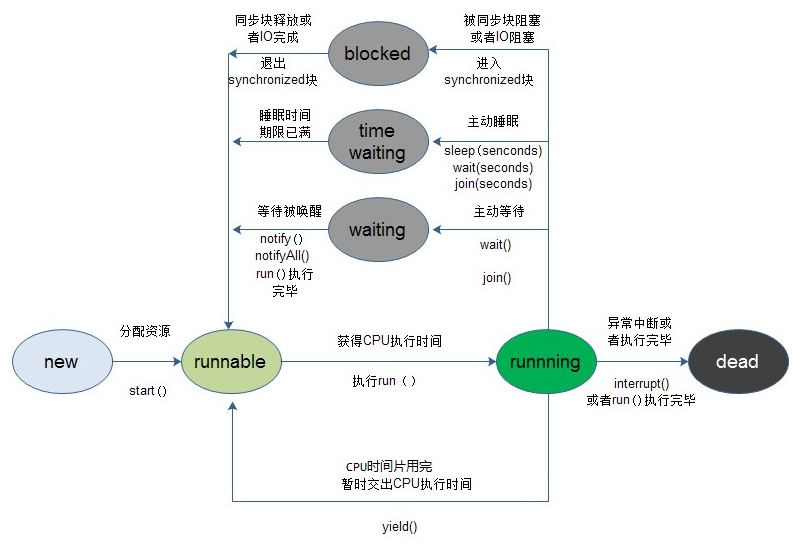
用来获取和设置线程优先级。

### 4）setDaemon和isDaemon

用来设置线程是否成为守护线程和判断线程是否是守护线程。

守护线程和用户线程的区别在于：守护线程依赖于创建它的线程，而用户线程则不依赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。

Thread类有一个比较常用的静态方法currentThread()用来获取当前线程。

在上面已经说到了Thread类中的大部分方法，那么Thread类中的方法调用到底会引起线程状态发生怎样的变化呢？下面一幅图就是在上面的图上进行改进而来的：

# [Java并发编程：synchronized](http://www.cnblogs.com/dolphin0520/p/3923737.html)

http://www.cnblogs.com/dolphin0520/p/3923737.html

## 什么时候会出现线程安全问题

多个线程同时访问一个资源时，会导致程序运行结果并不是想看到的结果。

也就是说，当多个线程同时访问临界资源（一个对象，对象中的属性，一个文件，一个数据库等）时，就可能会产生线程安全问题。

　　不过，当多个线程执行一个方法，方法内部的局部变量并不是临界资源，因为方法是在栈上执行的，而Java栈是线程私有的，因此不会产生线程安全问题。

## 2.如何解决线程安全问题

基本上所有的并发模式在解决线程安全问题时，都采用“序列化访问临界资源”的方案，即在同一时刻，只能有一个线程访问临界资源，也称作同步互斥访问。

　　通常来说，是在访问临界资源的代码前面加上一个锁，当访问完临界资源后释放锁，让其他线程继续访问。

在Java中，提供了两种方式来实现同步互斥访问：synchronized和Lock。

## 3.synchronized同步方法或同步块

在Java中，每一个对象都拥有一个锁标记（monitor），也称为监视器，同时访问某个对象时，线程只有获取了该对象的锁才能访问。

在Java中，可以使用synchronized关键字来标记一个方法或者代码块，当某个线程调用该对象的synchronized方法或者访问synchronized代码块时，这个线程便获得了该对象的锁，其他线程暂时无法访问这个方法，只有等待这个方法执行完毕或者代码块执行完毕，这个线程才会释放该对象的锁，其他线程才能执行这个方法或者代码块。

有几点需要注意：

1. 当一个线程正在访问一个对象的synchronized方法，那么其他线程不能访问该对象的其他synchronized方法。这个原因很简单，因为一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他synchronized方法。
2. 当一个线程正在访问一个对象的synchronized方法，那么其他线程能访问该对象的非synchronized方法。这个原因很简单，访问非synchronized方法不需要获得该对象的锁，假如一个方法没用synchronized关键字修饰，说明它不会使用到临界资源，那么其他线程是可以访问这个方法的
3. 如果一个线程A需要访问对象object1的synchronized方法fun1，另外一个线程B需要访问对象object2的synchronized方法fun1，即使object1和object2是同一类型），也不会产生线程安全问题，因为他们访问的是不同的对象，所以不存在互斥问题。

synchronized代码块使用起来比synchronized方法要灵活得多。因为也许一个方法中只有一部分代码只需要同步，如果此时对整个方法用synchronized进行同步，会影响程序执行效率。而使用synchronized代码块就可以避免这个问题，synchronized代码块可以实现只对需要同步的地方进行同步。

**所以要尽量使用synchronized代码块而不是synchronized方法。**

另外，每个类也会有一个锁，它可以用来控制对static数据成员的并发访问。

并且如果一个线程执行一个对象的非static synchronized方法，另外一个线程需要执行这个对象所属类的static synchronized方法，此时不会发生互斥现象，因为访问static synchronized方法占用的是类锁，而访问非static synchronized方法占用的是对象锁，所以不存在互斥现象。

对于synchronized方法，执行中的线程识别该方法的 method\_info 结构是否有 ACC\_SYNCHRONIZED 标记设置，然后它自动获取对象的锁，调用方法，最后释放锁。如果有异常发生，线程自动释放锁。

有一点要注意：对于synchronized方法或者synchronized代码块，当出现异常时，JVM会自动释放当前线程占用的锁，因此不会由于异常导致出现死锁现象。

# [Java并发编程：Lock](http://www.cnblogs.com/dolphin0520/p/3923167.html)

http://www.cnblogs.com/dolphin0520/p/3923167.html

## synchronized的缺陷

在上面一篇文章中，我们了解到如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时JVM会让线程自动释放锁。

那么如果这个获取锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过Lock就可以办到。

再举个例子：当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

总结一下，也就是说Lock提供了比synchronized更多的功能。但是要注意以下几点：

1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

2）Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。

## 2.java.util.concurrent.locks包下常用的类

### 1.Lock

Lock是一个接口：

public interface Lock {

void lock();//用来获取锁

void lockInterruptibly() throws InterruptedException; //用来获取锁

boolean tryLock();//用来获取锁

boolean tryLock(long time, TimeUnit unit) throws InterruptedException; //用来获取锁

void unlock();//用来释放锁

    Condition newCondition();

}

#### lock()方法

是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

由于在前面讲到如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用Lock必须在try{}catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。

#### tryLock()方法

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

#### lockInterruptibly()方法

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。因为本身在前面的文章中讲过单独调用interrupt()方法不能中断正在运行过程中的线程，只能中断阻塞过程中的线程。

因此当通过lockInterruptibly()方法获取某个锁时，如果不能获取到，只有进行等待的情况下，是可以响应中断的。而用synchronized修饰的话，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

### ReentrantLock

ReentrantLock，意思是“可重入锁”，关于可重入锁的概念在下一节讲述。ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

### ReadWriteLock

ReadWriteLock也是一个接口，在它里面只定义了两个方法：

public interface ReadWriteLock {

    //return the lock used for reading.

    Lock readLock();

    //Returns the lock used for writing.

    Lock writeLock();

}

一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。下面的ReentrantReadWriteLock实现了ReadWriteLock接口。

### ReentrantReadWriteLock

ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

### Lock和synchronized的选择

总结来说，Lock和synchronized有以下几点不同：

1. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；
2. synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
3. Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
4. 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
5. Lock可以提高多个线程进行读操作的效率。

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

## 3.锁的相关概念介绍

### 1.可重入锁

如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

EX:

class MyClass {

    public synchronized void method1() {

        method2();

    }

    public synchronized void method2() {

    }

}

上述代码中的两个方法method1和method2都用synchronized修饰了，假如某一时刻，线程A执行到了method1，此时线程A获取了这个对象的锁，而由于method2也是synchronized方法，假如synchronized不具备可重入性，此时线程A需要重新申请锁。但是这就会造成一个问题，因为线程A已经持有了该对象的锁，而又在申请获取该对象的锁，这样就会线程A一直等待永远不会获取到的锁。

而由于synchronized和Lock都具备可重入性，所以不会发生上述现象。

### 可中断锁

可中断锁：顾名思义，就是可以相应中断的锁。

在Java中，synchronized就不是可中断锁，而Lock是可中断锁。

如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

在前面演示lockInterruptibly()的用法时已经体现了Lock的可中断性。

### 公平锁

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

我们可以在创建ReentrantLock对象时，通过以下方式来设置锁的公平性：

ReentrantLock lock = new ReentrantLock(true);

如果参数为true表示为公平锁，为fasle为非公平锁。默认情况下，如果使用无参构造器，则是非公平锁。

另外在ReentrantLock类中定义了很多方法，比如：

isFair()        //判断锁是否是公平锁

isLocked()    //判断锁是否被任何线程获取了

isHeldByCurrentThread()   //判断锁是否被当前线程获取了

hasQueuedThreads()   //判断是否有线程在等待该锁

在ReentrantReadWriteLock中也有类似的方法，同样也可以设置为公平锁和非公平锁。不过要记住，ReentrantReadWriteLock并未实现Lock接口，它实现的是ReadWriteLock接口。

### 读写锁

读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。

正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。

ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

可以通过readLock()获取读锁，通过writeLock()获取写锁。

# [Java并发编程：volatile关键字解析](http://www.cnblogs.com/dolphin0520/p/3920373.html)

<http://www.cnblogs.com/dolphin0520/p/3920373.html>

## 1.内存模型的相关概念

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题。

为了解决缓存不一致性问题，通常来说有以下2种解决方法：

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议

在早期的CPU当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为CPU和其他部件进行通信都是通过总线来进行的，如果对总线加LOCK#锁的话，也就是说阻塞了其他CPU对其他部件访问（如内存），从而使得只能有一个CPU能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他CPU才能从变量i所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。

但是上面的方式会有一个问题，由于在锁住总线期间，其他CPU无法访问内存，导致效率低下。

所以就出现了缓存一致性协议。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

## 2.并发编程中的三个概念

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。我们先看具体看一下这三个概念：

### 1.原子性

即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

### 2.可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

### 3.有序性

即程序执行的顺序按照代码的先后顺序执行。

什么是指令重排序：一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

如何保证？：处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。

指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

## 3.Java内存模型

Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。

那么Java语言 本身对 原子性、可见性以及有序性提供了哪些保证呢？

### 原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。

### 可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

### 有序性

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before原则（先行发生原则）。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

* 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
* 锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作
* volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
* 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
* 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作
* 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
* 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
* 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

## 深入剖析volatile关键字

### volatile关键字的两层语义

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* 1. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
  2. 禁止进行指令重排序。

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

### volatile保证原子性吗？

Volatile无法保证原子性。

**EX.**

public class Test {

    public volatile int inc = 0;

    public void increase() {

        inc++;

    }

    public static void main(String[] args) {

        final Test test = new Test();

        for(int i=0;i<10;i++){

            new Thread(){

                public void run() {

                    for(int j=0;j<1000;j++)

                        test.increase();

                };

            }.start();

        }

        while(Thread.activeCount()>1)  //保证前面的线程都执行完

            Thread.yield();

        System.out.println(test.inc);

    }

}

大家想一下这段程序的输出结果是多少？也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于10000的数字。

可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000次操作，那么最终inc的值应该是1000\*10=10000。

这里面就有一个误区了，volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量inc的值为10，

线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值是10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。

解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值。

根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

### volatile保证一致性吗?

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

volatile关键字禁止指令重排序有两层意思：

1. 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
2. 在进行指令优化时，不能将在对volatile变量访问前的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

**EX.**

//x、y为非volatile变量

//flag为volatile变量

x = 2;        //语句1

y = 0;        //语句2

flag = true;   //语句3

x = 4;        //语句4

y = -1;        //语句5

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会将语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

### Volatile的原理和实现机制

下面这段话摘自《深入理解Java虚拟机》：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”。

lock前缀指令实际上相当于一个内存屏障（也称内存栅栏），内存屏障会提供3个功能：

1. 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
2. 它会强制将对缓存的修改操作立即写入主存；
3. 如果是写操作，它会导致其他CPU中对应的缓存行无效。

### 使用volatile关键字的场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1. 对变量的写操作不依赖于当前值
2. 该变量没有包含在具有其他变量的不变式中

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。

# [Java并发编程：深入剖析ThreadLocal](http://www.cnblogs.com/dolphin0520/p/3920407.html)

http://www.cnblogs.com/dolphin0520/p/3920407.html

## 1.对ThreadLocal的理解

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

## 2.深入理解ThreadLocal类

ThreadLocal类提供的如下几个方法：

1. public T get() { }：用来获取ThreadLocal在当前线程中保存的变量副本
2. public void set(T value) { }：设置当前线程中变量的副本
3. public void remove() { }:移除当前线程中变量的副本
4. protected T initialValue() { }:用来在使用时进行重写, 是一个延迟加载方法

总结：

1. 实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；
2. 为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，就像上面代码中的longLocal和stringLocal；
3. 在进行get之前，必须先set，否则会报空指针异常；

如果想在get之前不需要调用set就能正常访问的话，必须重写initialValue()方法。

## 3.ThreadLocal的应用场景

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。案例详见原文。

# [Java并发编程：同步容器](http://www.cnblogs.com/dolphin0520/p/3933404.html)

<http://www.cnblogs.com/dolphin0520/p/3933404.html>

为了方便编写出线程安全的程序，Java里面提供了一些线程安全类和并发工具，比如：同步容器、并发容器、阻塞队列、Synchronizer（比如CountDownLatch）。今天我们就来讨论下同步容器。

## 为什么会出现同步容器？

在Java的集合容器框架中，主要有四大类别：List、Set、Queue、Map。

List、Set、Queue接口分别继承了Collection接口，Map本身是一个接口。

注意Collection和Map是一个顶层接口，而List、Set、Queue则继承了Collection接口，分别代表数组、集合和队列这三大类容器。

像ArrayList、LinkedList、HashMap这些容器都是非线程安全的。如果有多个线程并发地访问这些容器时，就会出现问题。因此，在编写程序时，必须要求程序员手动地在任何访问到这些容器的地方进行同步处理，这样导致在使用这些容器的时候非常地不方便。所以，Java提供了同步容器供用户使用。

## JAVA中的同步容器类

在Java中，同步容器主要包括2类：

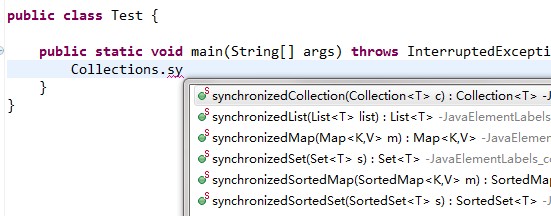
1. Vector、Stack、HashTable
2. Collections类中提供的静态工厂方法创建的类

Vector实现了List接口，Vector实际上就是一个数组，和ArrayList类似，但是Vector中的方法都是synchronized方法，即进行了同步措施。

Stack也是一个同步容器，它的方法也用synchronized进行了同步，它实际上是继承于Vector类。

HashTable实现了Map接口，它和HashMap很相似，但是HashTable进行了同步处理，而HashMap没有。

Collections类是一个工具提供类，注意，它和Collection不同，Collection是一个顶层的接口。在Collections类中提供了大量的方法，比如对集合或者容器进行排序、查找等操作。最重要的是，在它里面提供了几个静态工厂方法来创建同步容器类，如图所示：



## 同步容器的缺陷

1. 首先是性能必定受到影响，正常情况下慢于非同步容器。由于Vector中的add方法和get方法都进行了同步，因此，在有多个线程进行访问时，如果多个线程都只是进行读取操作，那么每个时刻就只能有一个线程进行读取，其他线程便只能等待，这些线程必须竞争同一把锁。
2. 同步容器也不是完全安全的，案例见原文。
3. ConcurrentModificationException异常

# Java并发编程：ConcurrentModificationException异常原因和解决办法

http://www.cnblogs.com/dolphin0520/p/3933551.html

## ConcurrentModificationException异常出现的原因

案例见原文。

关键点就在于：调用list.remove()方法导致modCount和expectedModCount的值不一致。注意，像使用for-each进行迭代实际上也会出现这种问题。

## 在单线程环境下的解决办法

案例见原文。

在迭代器中如果要删除元素的话，需要调用Itr类的remove方法。

## 在多线程环境下的解决办法

案例见原文。

一般有2种解决办法：

1. 在使用iterator迭代的时候使用synchronized或者Lock进行同步；

2）使用并发容器CopyOnWriteArrayList代替ArrayList和Vector。

# [Java并发编程：并发容器](http://www.cnblogs.com/dolphin0520/p/3933404.html)之ConcurrentHashMap

<http://www.cnblogs.com/dolphin0520/p/3932905.html>

Java5.0开始针对多线程并发访问设计，提供了并发性能较好的并发容器，引入了java.util.concurrent包。与Vector和Hashtable、Collections.synchronizedXxx()同步容器等相比，util.concurrent中引入的并发容器主要解决了两个问题：   
1）根据具体场景进行设计，尽量避免synchronized，提供并发性。

2）定义了一些并发安全的复合操作，并且保证并发环境下的迭代操作不会出错。

util.concurrent中容器在迭代时，可以不封装在synchronized中，可以保证不抛异常，但是未必每次看到的都是"最新的、当前的"数据。

## 简介

ConcurrentHashMap代替同步的Map（Collections.synchronized（new HashMap()）），众所周知，HashMap是根据散列值分段存储的，同步Map在同步的时候锁住了所有的段，而ConcurrentHashMap加锁的时候根据散列值锁住了散列值锁对应的那段，因此提高了并发性能。

ConcurrentHashMap也增加了对常用复合操作的支持，比如"若没有则添加"：putIfAbsent()，替换：replace()。这2个操作都是原子操作。

CopyOnWriteArrayList和CopyOnWriteArraySet分别代替List和Set，主要是在遍历操作为主的情况下来代替同步的List和同步的Set，这也就是上面所述的思路：迭代过程要保证不出错，除了加锁，另外一种方法就是"克隆"容器对象。

ConcurrentLinkedQuerue是一个先进先出的队列。它是非阻塞队列。

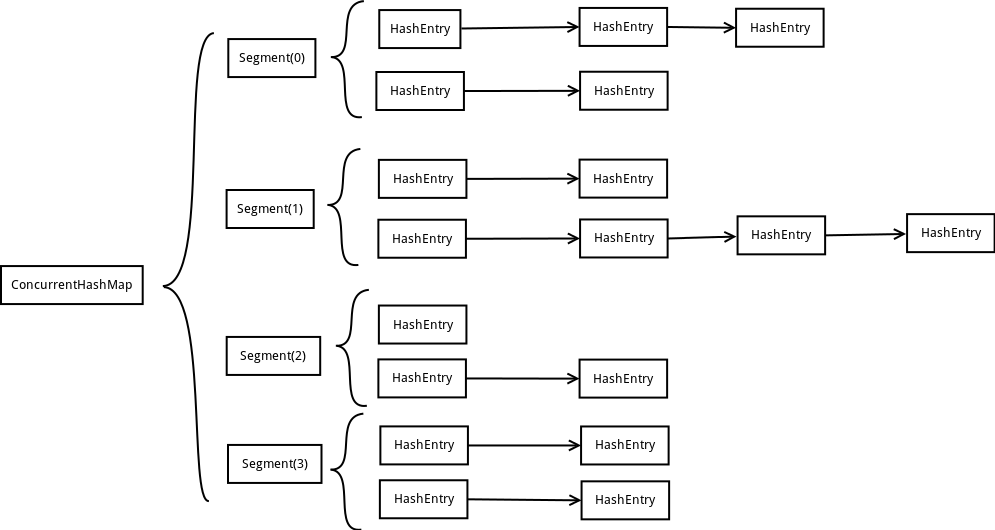
ConcurrentSkipListMap可以在高效并发中替代SoredMap（例如用Collections.synchronzedMap包装的TreeMap）。

ConcurrentSkipListSet可以在高效并发中替代SoredSet（例如用Collections.synchronzedSet包装的TreeMap）。

**ConcurrentHashMap可以做到读取数据不加锁，并且其内部的结构可以让其在进行写操作的时候能够将锁的粒度保持地尽量地小，不用对整个ConcurrentHashMap加锁。**

## ConcurrentHashMap的内部结构

ConcurrentHashMap为了提高本身的并发能力，在内部采用了一个叫做Segment的结构，一个Segment其实就是一个类Hash Table的结构，Segment内部维护了一个链表数组，我们用下面这一幅图来看下ConcurrentHashMap的内部结构：



从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是带来的好处是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上），所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

### Segment

Segment的数据结构：

static final class Segment<K,V> extends ReentrantLock implements Serializable {

    transient volatile int count; //Segment中元素的数量

    transient int modCount; //对table的大小造成影响的操作的数量（比如put或者remove操作）

    transient int threshold; //阈值，Segment里面元素的数量超过这个值依旧就会对Segment进行扩容

    transient volatile HashEntry<K,V>[] table; //链表数组，数组中的每一个元素代表了一个链表的头部

    final float loadFactor; //负载因子，用于确定threshold

}

transient :  
Java语言的关键字，用来表示一个域不是该对象串行化的一部分。当一个对象被串行化的时候，transient型变量的值不包括在串行化的表示中，然而非transient型的变量是被包括进去的

### HashEntry

Segment中的元素是以HashEntry的形式存放在链表数组中的，以下是HashEntry的结构：

static final class HashEntry<K,V> {

    final K key;

    final int hash;

    volatile V value;

    final HashEntry<K,V> next;

}

可以看到HashEntry的一个特点，除了value以外，其他的几个变量都是final的，这样做是为了防止链表结构被破坏，出现ConcurrentModification的情况。

## ConcurrentHashMap的初始化

结合源代码来具体分析一下ConcurrentHashMap的实现，先看下初始化方法：

public ConcurrentHashMap(int initialCapacity,

                         float loadFactor, int concurrencyLevel) {

    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

        throw new IllegalArgumentException();

    if (concurrencyLevel > MAX\_SEGMENTS)

        concurrencyLevel = MAX\_SEGMENTS;

    // Find power-of-two sizes best matching arguments

    int sshift = 0;

    int ssize = 1;

    while (ssize < concurrencyLevel) {

        ++sshift;

        ssize <<= 1;

    }

    segmentShift = 32 - sshift;

    segmentMask = ssize - 1;

    this.segments = Segment.newArray(ssize);

    if (initialCapacity > MAXIMUM\_CAPACITY)

        initialCapacity = MAXIMUM\_CAPACITY;

    int c = initialCapacity / ssize;

    if (c \* ssize < initialCapacity)

        ++c;

    int cap = 1;

    while (cap < c)

        cap <<= 1;

    for (int i = 0; i < this.segments.length; ++i)

        this.segments[i] = new Segment<K,V>(cap, loadFactor);

}

CurrentHashMap的初始化一共有三个参数，一个initialCapacity，表示初始的容量，一个loadFactor，表示负载参数，最后一个是concurrentLevel，代表ConcurrentHashMap内部的Segment的数量，ConcurrentLevel一经指定，不可改变，后续如果ConcurrentHashMap的元素数量增加导致ConrruentHashMap需要扩容，ConcurrentHashMap不会增加Segment的数量，而只会增加Segment中链表数组的容量大小，这样的好处是扩容过程不需要对整个ConcurrentHashMap做rehash，而只需要对Segment里面的元素做一次rehash就可以了。

整个ConcurrentHashMap的初始化方法还是非常简单的，先是根据concurrentLevel来new出Segment，这里Segment的数量是不大于concurrentLevel的最大的2的指数，就是说Segment的数量永远是2的指数个，这样的好处是方便采用移位操作来进行hash，加快hash的过程。接下来就是根据intialCapacity确定Segment的容量的大小，每一个Segment的容量大小也是2的指数，同样使为了加快hash的过程。

这边需要特别注意一下两个变量，分别是segmentShift和segmentMask，这两个变量在后面将会起到很大的作用，假设构造函数确定了Segment的数量是2的n次方，那么segmentShift就等于32减去n，而segmentMask就等于2的n次方减一。

## ConcurrentHashMap的get,put,remove,size操作

1. 首先remove操作也是确定需要删除的元素的位置，不过这里删除元素的方法不是简单地把待删除元素的前面的一个元素的next指向后面一个就完事了，我们之前已经说过HashEntry中的next是final的，一经赋值以后就不可修改，在定位到待删除元素的位置以后，程序就将待删除元素前面的那一些元素全部复制一遍，然后再一个一个重新接到链表上去
2. 一个Segment中有一个modCount变量，代表的是对Segment中元素的数量造成影响的操作的次数，这个值只增不减，size操作就是遍历了两次Segment，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，就将原先遍历的结果返回，如果不相同，则把这个过程再重复做一次，如果再不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了

# [Java并发编程：并发容器之CopyOnWriteArrayList](http://www.cnblogs.com/dolphin0520/p/3938914.html)

<http://www.cnblogs.com/dolphin0520/p/3938914.html>

Copy-On-Write简称COW，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。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

## 何为CopyOnWrite容器

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的实现原理

在使用CopyOnWriteArrayList之前，我们先阅读其源码了解下它是如何实现的。以下代码是向CopyOnWriteArrayList中add方法的实现（向CopyOnWriteArrayList里添加元素），可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

 public boolean add(E e) {

    final ReentrantLock lock = this.lock;

    lock.lock();

    try {

        Object[] elements = getArray();

        int len = elements.length;

        Object[] newElements = Arrays.copyOf(elements, len + 1);

        newElements[len] = e;

        setArray(newElements);

        return true;

    } finally {

        lock.unlock();

    }

}

读的时候不需要加锁，如果读的时候有多个线程正在向CopyOnWriteArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的CopyOnWriteArrayList。

## CopyOnWrite的应用场景

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。

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。

## CopyOnWrite的缺点

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，所以占用内存较大。

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/)。

**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。

# Java并发编程：阻塞队列

<http://www.cnblogs.com/dolphin0520/p/3932906.html>

在前面我们接触的队列都是非阻塞队列，比如PriorityQueue、LinkedList（LinkedList是双向链表，它实现了Dequeue接口）。

使用非阻塞队列的时候有一个很大问题就是：它不会对当前线程产生阻塞，那么在面对类似消费者-生产者的模型时，就必须额外地实现同步策略以及线程间唤醒策略，这个实现起来就非常麻烦。但是有了阻塞队列就不一样了，它会对当前线程产生阻塞，比如一个线程从一个空的阻塞队列中取元素，此时线程会被阻塞直到阻塞队列中有了元素。当队列中有元素后，被阻塞的线程会自动被唤醒（不需要我们编写代码去唤醒）。这样提供了极大的方便性。

## 几种主要的阻塞队列

### 1)ArrayBlockingQueue

基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。

### 2) LinkedBlockingQueue

基于链表实现的一个阻塞队列，在创建LinkedBlockingQueue对象时如果不指定容量大小，则默认大小为Integer.MAX\_VALUE。

### PriorityBlockingQueue

以上2种队列都是先进先出队列，而PriorityBlockingQueue却不是，它会按照元素的优先级对元素进行排序，按照优先级顺序出队，每次出队的元素都是优先级最高的元素。注意，此阻塞队列为无界阻塞队列，即容量没有上限（通过源码就可以知道，它没有容器满的信号标志），前面2种都是有界队列。

### DelayQueue

基于PriorityQueue，一种延时阻塞队列，DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue也是一个无界队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

## 阻塞队列中的方法 VS 非阻塞队列中的方法

### 非阻塞队列中的几个主要方法：

add(E e):将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常；

remove()：移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常；

offer(E e)：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false；

poll()：移除并获取队首元素，若成功，则返回队首元素；否则返回null；

peek()：获取队首元素，若成功，则返回队首元素；否则返回null

对于非阻塞队列，一般情况下建议使用offer、poll和peek三个方法，不建议使用add和remove方法。因为使用offer、poll和peek三个方法可以通过返回值判断操作成功与否，而使用add和remove方法却不能达到这样的效果。注意，非阻塞队列中的方法都没有进行同步措施。

### 阻塞队列中的几个主要方法：

阻塞队列包括了非阻塞队列中的大部分方法，上面列举的5个方法在阻塞队列中都存在，但是要注意这5个方法在阻塞队列中都进行了同步措施。除此之外，阻塞队列提供了另外4个非常有用的方法：

put(E e): 向队尾存入元素，如果队列满，则等待；

take():从队首取元素，如果队列为空，则等待；

offer(E e,long timeout, TimeUnit unit): 向队尾存入元素，如果队列满，则等待一定的时间，当时间期限达到时，如果还没有插入成功，则返回false；否则返回true；

poll(long timeout, TimeUnit unit): 从队首取元素，如果队列空，则等待一定的时间，当时间期限达到时，如果取到，则返回null；否则返回取得的元素；

## 阻塞队列的实现原理

前面谈到了非阻塞队列和阻塞队列中常用的方法，下面来探讨阻塞队列的实现原理，本文以ArrayBlockingQueue为例，其他阻塞队列实现原理可能和ArrayBlockingQueue有一些差别，但是大体思路应该类似。

### 1)首先看一下ArrayBlockingQueue类中的几个成员变量：

public class ArrayBlockingQueue<E> extends AbstractQueue<E>

implements BlockingQueue<E>, java.io.Serializable {

private static final long serialVersionUID = -817911632652898426L;

private final E[] items;

private int takeIndex; //队首元素的下标

private int putIndex; //队尾元素的下标

private int count; //队列中元素的个数

private final ReentrantLock lock; //可重入锁

private final Condition notEmpty; //notEmpty和notFull是等待条件

private final Condition notFull;

}

### 2)下面看一下ArrayBlockingQueue的构造器，构造器有三个重载版本：

public ArrayBlockingQueue(int capacity) {//capacity容量}

public ArrayBlockingQueue(int capacity, boolean fair) {

//fair公平性

}

public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c) {//c用另外一个集合进行初始化}

### 3)然后看它的两个关键方法的实现：put()和take()：

public void put(E e) throws InterruptedException {

    if (e == null) throw new NullPointerException();

    final E[] items = this.items;

    final ReentrantLock lock = this.lock;

    lock.lockInterruptibly();

    try {

        try {

            while (count == items.length)

                notFull.await();

        } catch (InterruptedException ie) {

            notFull.signal(); // propagate to non-interrupted thread

            throw ie;

        }

        insert(e);

    } finally {

        lock.unlock();

    }

}

从put方法的实现可以看出，它先获取了锁，并且获取的是可中断锁，然后判断当前元素个数是否等于数组的长度，如果相等，则调用notFull.await()进行等待，如果捕获到中断异常，则唤醒线程并抛出异常。

当被其他线程唤醒时，通过insert(e)方法插入元素，最后解锁。

看一下insert方法的实现：

private void insert(E x) {

    items[putIndex] = x;

    putIndex = inc(putIndex);

    ++count;

    notEmpty.signal();

}

它是一个private方法，插入成功后，通过notEmpty唤醒正在等待取元素的线程。

### 4)下面是take()方法的实现：

public E take() throws InterruptedException {

    final ReentrantLock lock = this.lock;

    lock.lockInterruptibly();

    try {

        try {

            while (count == 0)

                notEmpty.await();

        } catch (InterruptedException ie) {

            notEmpty.signal(); // propagate to non-interrupted thread

            throw ie;

        }

        E x = extract();

        return x;

    } finally {

        lock.unlock();

    }

}

跟put方法实现很类似，只不过put方法等待的是notFull信号，而take方法等待的是notEmpty信号。在take方法中，如果可以取元素，则通过extract方法取得元素，下面是extract方法的实现：

private E extract() {

    final E[] items = this.items;

    E x = items[takeIndex];

    items[takeIndex] = null;

    takeIndex = inc(takeIndex);

    --count;

    notFull.signal();

    return x;

}

跟insert方法也很类似。

其实从这里大家应该明白了阻塞队列的实现原理，事实它和我们用Object.wait()、Object.notify()和非阻塞队列实现生产者-消费者的思路类似，只不过它把这些工作一起集成到了阻塞队列中实现。

## 示例和使用场景

在并发编程中，一般推荐使用阻塞队列，这样实现可以尽量地避免程序出现意外的错误。

阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。还有其他类似的场景，只要符合生产者-消费者模型的都可以使用阻塞队列。

# Java并发编程：线程池的使用

<http://www.cnblogs.com/dolphin0520/p/3932921.html>

## Java中的ThreadPoolExecutor类

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。下面我们来看一下ThreadPoolExecutor类的具体实现源码。

在ThreadPoolExecutor类中提供了四个构造方法：

public class ThreadPoolExecutor extends AbstractExecutorService {

public ThreadPoolExecutor(

int corePoolSize, //核心池的大小

int maximumPoolSize, //线程池最大线程数

long keepAliveTime, //表示线程没有任务执行时最多保持多久时间会终止

TimeUnit unit, //参数keepAliveTime的时间单位

BlockingQueue<Runnable> workQueue //一个阻塞队列，用来存储等待执行的任务

);

  public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

            BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory //线程工厂，主要用来创建线程；

);

  public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

            BlockingQueue<Runnable> workQueue,

RejectedExecutionHandler handler //当拒绝处理任务时的策略

);

  public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

        BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

    ...

}

maximumPoolSize：这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

* maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
* keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
* unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：
* TimeUnit.DAYS; //天
* TimeUnit.HOURS; //小时
* TimeUnit.MINUTES; //分钟
* TimeUnit.SECONDS; //秒
* TimeUnit.MILLISECONDS; //毫秒
* TimeUnit.MICROSECONDS; //微妙
* TimeUnit.NANOSECONDS; //纳秒
* workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：
* ArrayBlockingQueue;
* LinkedBlockingQueue;
* SynchronousQueue;

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关

* threadFactory：线程工厂，主要用来创建线程；
* handler：表示当拒绝处理任务时的策略，有以下四种取值：
* ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
* ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
* ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
* ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

在ThreadPoolExecutor类中有几个非常重要的方法：

execute() //通过这个方法可以向线程池提交一个任务，交由线程池去执行。

submit() //这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果

shutdown() //用来关闭线程池

shutdownNow() //用来关闭线程池

## 深入剖析线程池实现原理

### 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

volatile int runState;

static final int RUNNING = 0;

static final int SHUTDOWN = 1;

static final int STOP = 2;

static final int TERMINATED = 3;

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

### 任务的执行

在了解将任务提交给线程池到任务执行完毕整个过程之前，我们先来看一下ThreadPoolExecutor类中其他的一些比较重要成员变量：

private final BlockingQueue<Runnable> workQueue;//任务缓存队列，用来存放等待执行的任务

private final ReentrantLock mainLock = new ReentrantLock();//线程池的主要状态锁，对线程池状态（比如线程池大小,runState等）的改变都要使用这个锁

private final HashSet<Worker> workers = new HashSet<Worker>();//用来存放工作集

private volatile long  keepAliveTime;//线程存货时间

private volatile boolean allowCoreThreadTimeOut;//是否允许为核心线程设置存活时间

private volatile int   corePoolSize;//核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）

private volatile int   maximumPoolSize;//线程池最大能容忍的线程数

private volatile int   poolSize;//线程池中当前的线程数

private volatile RejectedExecutionHandler handler;//任务拒绝策略

private volatile ThreadFactory threadFactory;//线程工厂，用来创建线程

private int largestPoolSize;//用来记录线程池中曾经出现过的最大线程数

private long completedTaskCount;//用来记录已经执行完毕的任务个数

总结一下任务提交给线程池之后到被执行的整个过程：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

### 线程池中的线程初始化

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

* 1. prestartCoreThread()：初始化一个核心线程；
* 2. prestartAllCoreThreads()：初始化所有核心线程

### 任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

1. ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；
2. LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；
3. synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

### 任务拒绝策略

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

### 线程池关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

### 线程池容量的动态调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()，

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

## 使用示例

在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池：

1）Executors.newCachedThreadPool();//创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

2）Executors.newSingleThreadExecutor();//创建容量为1的缓冲池

3）Executors.newFixedThreadPool(int);//创建固定容量大小的缓冲池

newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

实际中，如果Executors提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置ThreadPoolExecutor的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

# [Java并发编程：CountDownLatch、CyclicBarrier和Semaphore](http://www.cnblogs.com/dolphin0520/p/3920397.html)

<http://www.cnblogs.com/dolphin0520/p/3920397.html>

## 1.CountDownLatch用法

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

CountDownLatch类只提供了一个构造器：

public CountDownLatch(int count) {  };  //参数count为计数值

然后下面这3个方法是CountDownLatch类中最重要的方法：

public void await() throws InterruptedException { };//调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };//和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public void countDown() { };//将count值减1

## 2.CyclicBarrier用法

字面意思回环栅栏，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。

CyclicBarrier类位于java.util.concurrent包下，CyclicBarrier提供2个构造器：

public CyclicBarrier(int parties, Runnable barrierAction)// parties:让多少个线程或者任务等待至barrier状态；barrierAction:当这些线程都达到barrier状态时会执行的内容.

public CyclicBarrier(int parties)

然后CyclicBarrier中最重要的方法就是await方法，它有2个重载版本：

public int await() throws InterruptedException, BrokenBarrierException //挂起当前线程，直至所有线程都到达barrier状态再同时执行后续任务

public int await(long timeout, TimeUnit unit)throws InterruptedException,BrokenBarrierException,TimeoutException

//让这些线程等待至一定的时间，如果还有线程没有到达barrier状态就直接让到达barrier的线程执行后续任务

CyclicBarrier可以重复使用，而CountDownLatch不行

## 3.Semaphore用法

Semaphore翻译成字面意思为 信号量，Semaphore可以控制同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

Semaphore类位于java.util.concurrent包下，它提供了2个构造器：

public Semaphore(int permits) {//参数permits表示许可数目，即同时可以允许多少线程进行访问

    sync = new NonfairSync(permits);

}

public Semaphore(int permits, boolean fair) {//这个多了一个参数fair表示是否是公平的，即等待时间越久的越先获取许可

    sync = (fair)? new FairSync(permits) : new NonfairSync(permits);

}

Semaphore类中比较重要的几个方法:

public void acquire() throws InterruptedException //获取一个许可, 若无许可能够获得，则会一直等待，直到获得许可

public void acquire(int permits) throws InterruptedException//获取permits个许可

public void release()//释放一个许可, 注意，在释放许可之前，必须先获获得许可

public void release(int permits) //释放permits个许可

这4个方法都会被阻塞，如果想立即得到执行结果，可以使用下面几个方法：

public boolean tryAcquire()//尝试获取一个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

public boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException//尝试获取一个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

public boolean tryAcquire(int permits) //尝试获取permits个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

public boolean tryAcquire(int permits, long timeout,TimeUnit unit) throws InterruptedException //尝试获取permits个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

另外还可以通过availablePermits()方法得到可用的许可数目。

## 4.总结：

1)CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；

另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

2)Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。

# [Java并发编程：Callable、Future和FutureTask](http://www.cnblogs.com/dolphin0520/p/3949310.html)

<http://www.cnblogs.com/dolphin0520/p/3949310.html>

在前面的文章中我们讲述了创建线程的2种方式，一种是直接继承Thread，另外一种就是实现Runnable接口。这2种方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，这样使用起来就比较麻烦。而自从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。

## 1. Callable

Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()：

public interface Callable<V> {

    V call() throws Exception;

}

可以看到，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

那么怎么使用Callable呢？一般情况下是配合ExecutorService来使用的，在ExecutorService接口中声明了若干个submit方法的重载版本：

<T> Future<T> submit(Callable<T> task);//该方法里面的参数类型就是Callable

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

## 2. Future

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

Future类位于java.util.concurrent包下，它是一个接口：

**public interface Future<V>** {

**boolean cancel(boolean mayInterruptIfRunning);**// 用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是false，此方法肯定返回false，即如果取消已经完成的任务会返回false；如果任务正在执行，若mayInterruptIfRunning设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false；如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true

**boolean isCancelled();**//表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。

**boolean isDone();**//表示任务是否已经完成，若任务完成，则返回true

**V get()** throws InterruptedException, ExecutionException;// 用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回

**V get(long timeout, TimeUnit unit)** throws InterruptedException, ExecutionException, TimeoutException;// 用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null

}

也就是说Future提供了三种功能：

1)判断任务是否完成；

2)能够中断任务；

3)能够获取任务执行结果

因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask

## FutureTask

先来看一下FutureTask的实现：

public class FutureTask<V> implements RunnableFuture<V>

FutureTask类实现了RunnableFuture接口，我们看一下RunnableFuture接口的实现：

public interface RunnableFuture<V> extends Runnable, Future<V> {

    void run();

}

可以看出RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

FutureTask提供了2个构造器：

public FutureTask(Callable<V> callable) {

}

public FutureTask(Runnable runnable, V result) {

}

事实上，FutureTask是Future接口的一个唯一实现类。

# Java并发编程：Timer和TimerTask

<http://www.cnblogs.com/dolphin0520/p/3938991.html>

其实就Timer来讲就是一个调度器,而TimerTask呢只是一个实现了run方法的一个类,而具体的TimerTask需要由你自己来实现,例如这样:

Timer timer = new Timer();

timer.schedule(new TimerTask() {

        public void run() {

            System.out.println("abc");

        }

}, 200000 , 1000);

这里直接实现一个TimerTask(当然，你可以实现多个TimerTask，多个TimerTask可以被一个Timer会被分配到多个Timer中被调度，后面会说到Timer的实现机制就是说内部的调度机制)，然后编写run方法，20s后开始执行，每秒执行一次，当然你通过一个timer对象来操作多个timerTask，其实timerTask本身没什么意义，只是和timer集合操作的一个对象，实现它就必然有对应的run方法，以被调用，他甚至于根本不需要实现Runnable，因为这样往往混淆视听了，为什么呢？也是本文要说的重点。

在说到timer的原理时，我们先看看Timer里面的一些常见方法：

**public void schedule(TimerTask task, long delay)// 这个方法是调度一个task，经过delay(ms)后开始进行调度，仅仅调度一次。**

**public void schedule(TimerTask task, Date time)// 在指定的时间点time上调度一次**

**public void schedule(TimerTask task, long delay, long period)// 调度一个task，在delay（ms）后开始调度，每次调度完后，最少等待period（ms）后才开始调度**

**public void schedule(TimerTask task, Date firstTime, long period)// 和上一个方法类似，唯一的区别就是传入的第二个参数为第一次调度的时间**

**public void scheduleAtFixedRate(TimerTask task, long delay, long period)// 调度一个task，在delay(ms)后开始调度，然后每经过period(ms)再次调度**

**public void scheduleAtFixedRate(TimerTask task, Date firstTime,long period)// 方法同上，唯一的区别就是第一次调度时间设置为一个Date时间，而不是当前时间的一个时间片**

接下来看源码, 首先看Timer的**构造方法**:

**1)**无参构造方法，简单通过Tiemer为前缀构造一个线程名称, 创建的线程不为主线程，则主线程结束后，timer自动结束，而无需使用cancel来完成对timer的结束

public Timer() {

    this("Timer-" + serialNumber());

}

**2)**传入了是否为守护线程，守护线程当且仅当进程结束时，自动注销掉

public Timer(boolean isDaemon) {

    this("Timer-" + serialNumber(), isDaemon);

}

1. 另外两个构造方法负责传入名称和将timer启动：

public Timer(String name, boolean isDaemon) {

      thread.setName(name);

      thread.setDaemon(isDaemon);

      thread.start();

}

这里有一个thread，这个thread很明显是一个线程，被包装在了Timer类中，我们看下这个thread的定义是：

private TimerThread thread = new TimerThread(queue);

而定义TimerThread部分的是：

class TimerThread extends Thread {

看到这里知道了，Timer内部包装了一个线程，用来做独立于外部线程的调度，而TimerThread是一个default类型的，默认情况下是引用不到的，是被Timer自己所使用的。

**接下来看下有那些属性**

除了上面提到的thread，还有一个很重要的属性是：

private TaskQueue queue = new TaskQueue();

**来看sched方法的实现体：**

private void sched(TimerTask task, long time, long period) {

        if (time < 0)

            throw new IllegalArgumentException("Illegal execution time.");

        synchronized(queue) {

            if (!thread.newTasksMayBeScheduled)

                throw new IllegalStateException("Timer already cancelled.");

            synchronized(task.lock) {

                if (task.state != TimerTask.VIRGIN)

                    throw new IllegalStateException(

                        "Task already scheduled or cancelled");

                task.nextExecutionTime = time;

                task.period = period;

                task.state = TimerTask.SCHEDULED;

            }

            queue.add(task);

            if (queue.getMin() == task)

                queue.notify();

        }

    }

简言之，这里就是讲task放入队列queue的过程，此时，你可能对queue的结构有些兴趣，那么我们先来看看queue属性的结构TaskQueue：

class TaskQueue {

    private TimerTask[] queue = new TimerTask[128];

    private int size = 0;

可见，TaskQueue的结构很简单，为一个数组，加一个size，有点像ArrayList，是不是长度就128呢，当然不是，ArrayList可以扩容，它可以，只是会造成内存拷贝而已，所以一个Timer来讲，只要内部的task个数不超过128是不会造成扩容的；内部提供了add(TimerTask)、size()、getMin()、get(int)、removeMin()、quickRemove(int)、rescheduleMin(long newTime)、isEmpty()、clear()、fixUp()、fixDown()、heapify()；

这里面的方法大概意思是：

**add(TimerTaskt)**为增加一个任务

**size()**任务队列的长度

**getMin()**获取当前排序后最近需要执行的一个任务，下标为1，队列头部0是不做任何操作的

**get(inti)**获取指定下标的数据，当然包括下标0

**removeMin()**为删除当前最近执行的任务，也就是第一个元素，通常只调度一次的任务，在执行完后，调用此方法，就可以将TimerTask从队列中移除

**quickRmove(inti)**删除指定的元素，一般来说是不会调用这个方法的，这个方法只有在Timer发生purge的时候，并且当对应的**TimerTask**调用了**cancel**方法的时候，才会被调用这个方法，也就是取消某个**TimerTask**，然后就会从队列中移除(注意如果任务正在执行中，还是仍然会执行的，虽然在队列中被移除了)，还有就是这个cancel方法并不是Timer的cancel方法而是TimerTask，一个是调度器的，一个是单个任务的，最后注意，这个quickRmove完成后，是将队列最后一个元素补充到这个位置，所以此时会造成顺序不一致的问题，后面会有方法进行回补。

**rescheduleMin**(long newTime)是重新设置当前执行的任务的下一次执行时间，并在队列中将其从新排序到合适的位置，而调用的是后面说的**fixDown**方法。

对于**fixUp**和**fixDown**方法来讲，前者是当新增一个task的时候，首先将元素放在队列的尾部，然后向前找是否有比自己还要晚执行的任务，如果有，就将两个任务的顺序进行交换一下。而fixDown正好相反，执行完第一个任务后，需要加上一个时间片得到下一次执行时间，从而需要将其顺序与后面的任务进行对比下。

最后一个方法是**heapify**，其实就是将队列的后半截，全部做一次**fixeDown**的操作，这个操作主要是为了回补**quickRemove**方法，当大量的quickRmove后，顺序被打乱后，此时将一半的区域做一次非常简单的排序即可。

我们继续回到Timer，里面还有两个方法是：cancel()和方法purge()方法，其实就cancel方法来讲，一个取消操作，在测试中你会发现，如果一旦执行了这个方法timer就会结束掉

**回过头来再看看：**

Timer和TimerTask的简单组合是多线程的嘛？不是，一个Timer内部包装了“一个Thread”和“一个Task”队列，这个队列按照一定的方式将任务排队处理，包含的线程在Timer的构造方法调用时被启动，这个Thread的run方法无限循环这个Task队列，若队列为空且没发生cancel操作，此时会一直等待，如果等待完成后，队列还是为空，则认为发生了cancel从而跳出死循环，结束任务；循环中如果发现任务需要执行的时间小于系统时间，则需要执行，那么根据任务的时间片从新计算下次执行时间，若时间片为0代表只执行一次，则直接移除队列即可。

但是是否能实现多线程呢？可以，任何东西是否是多线程完全看个人意愿，多个Timer自然就是多线程的，每个Timer都有自己的线程处理逻辑，当然Timer从这里来看并不是很适合很多任务在短时间内的快速调度，至少不是很适合同一个timer上挂很多任务，在多线程的领域中我们更多是使用多线程中的：

Executors.newScheduledThreadPool

来完成对调度队列中的线程池的处理，内部通过**newScheduledThreadPoolExecutor**来创建线程池的**Executor**的创建，当然也可以调用：

Executors.unconfigurableScheduledExecutorService

方法来创建一个**DelegatedScheduledExecutorService**其实这个类就是包装了下下**scheduleExecutor**，也就是这只是一个壳，英文理解就是被委派的意思，被托管的意思。