# 1、线程基础、线程之间的共享和协作

## 基础概念

### 什么是进程和线程

**进程是程序运行资源分配的最小单位**

进程是操作系统进行资源分配的最小单位,其中资源包括:CPU、内存空间、磁盘IO等,同一进程中的多条线程共享该进程中的全部系统资源,而进程和进程之间是相互独立的。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。

进程是程序在计算机上的一次执行活动。当你运行一个程序,你就启动了一个进程。显然,程序是死的、静态的,进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程,它们就是处于运行状态下的操作系统本身,用户进程就是所有由你启动的进程。

**线程是CPU调度的最小单位,必须依赖于进程而存在**

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

**线程无处不在**

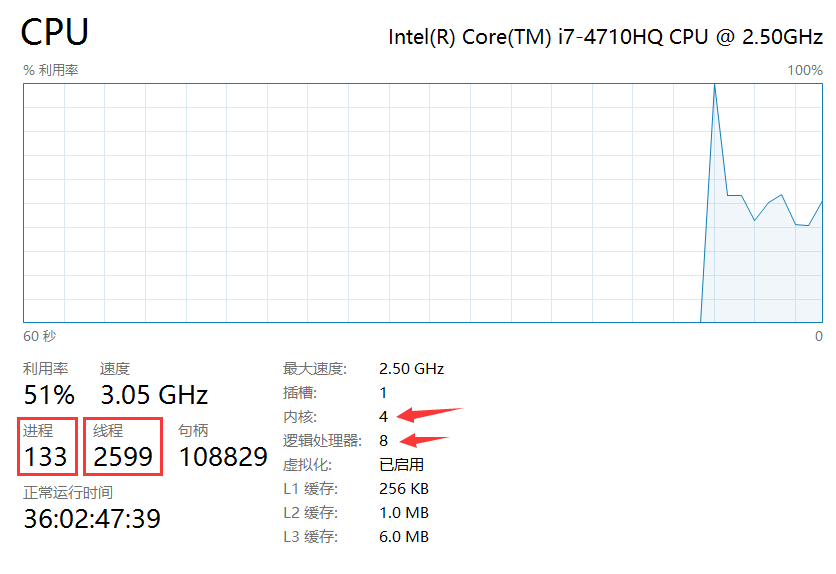
任何一个程序都必须要创建线程,特别是Java不管任何程序都必须启动一个main函数的主线程; Java Web开发里面的定时任务、定时器、JSP和 Servlet、异步消息处理机制,远程访问接口RM等,任何一个监听事件, onclick的触发事件等都离不开线程和并发的知识。

### CPU核心数和线程数的关系

多核心:也指单芯片多处理器( Chip Multiprocessors,简称CMP),CMP是由美国斯坦福大学提出的,其思想是将大规模并行处理器中的SMP(对称多处理器)集成到同一芯片内,各个处理器并行执行不同的进程。这种依靠多个CPU同时并行地运行程序是实现超高速计算的一个重要方向,称为并行处理

多线程: Simultaneous Multithreading.简称SMT.让同一个处理器上的多个线程同步执行并共享处理器的执行资源。

核心数、线程数:目前主流CPU都是多核的。增加核心数目就是为了增加线程数,因为操作系统是通过线程来执行任务的,一般情况下它们是1:1对应关系,也就是说四核CPU一般拥有四个线程。但 Intel引入超线程技术后,使核心数与线程数形成1:2的关系



### CPU时间片轮转机制

我们平时在开发的时候，感觉并没有受cpu核心数的限制，想启动线程就启动线程，哪怕是在单核CPU上，为什么？这是因为操作系统提供了一种CPU时间片轮转机制。

时间片轮转调度是一种最古老、最简单、最公平且使用最广的算法,又称RR调度。每个进程被分配一个时间段,称作它的时间片,即该进程允许运行的时间。

百度百科对CPU时间片轮转机制原理解释如下:

如果在时间片结束时进程还在运行,则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结来,则CPU当即进行切换。调度程序所要做的就是维护一张就绪进程列表,当进程用完它的时间片后,它被移到队列的末尾

时间片轮转调度中唯一有趣的一点是时间片的长度。从一个进程切换到另一个进程是需要定时间的,包括保存和装入寄存器值及内存映像,更新各种表格和队列等。假如进程切( processwitch),有时称为上下文切换( context switch),需要5ms,再假设时间片设为20ms,则在做完20ms有用的工作之后,CPU将花费5ms来进行进程切换。CPU时间的20%被浪费在了管理开销上了。

为了提高CPU效率,我们可以将时间片设为5000ms。这时浪费的时间只有0.1%。但考虑到在一个分时系统中,如果有10个交互用户几乎同时按下回车键,将发生什么情况?假设所有其他进程都用足它们的时间片的话,最后一个不幸的进程不得不等待5s才获得运行机会。多数用户无法忍受一条简短命令要5才能做出响应,同样的问题在一台支持多道程序的个人计算机上也会发

结论可以归结如下:时间片设得太短会导致过多的进程切换,降低了CPU效率:而设得太长又可能引起对短的交互请求的响应变差。将时间片设为100ms通常是一个比较合理的折衷。

在CPU死机的情况下,其实大家不难发现当运行一个程序的时候把CPU给弄到了100%再不重启电脑的情况下,其实我们还是有机会把它KⅢ掉的,我想也正是因为这种机制的缘故。

### 澄清并行和并发

我们举个例子,如果有条高速公路A上面并排有8条车道,那么最大的**并行**车辆就是8辆此条高速公路A同时并排行走的车辆小于等于8辆的时候,车辆就可以并行运行。CPU也是这个原理,一个CPU相当于一个高速公路A,核心数或者线程数就相当于并排可以通行的车道;而多个CPU就相当于并排有多条高速公路,而每个高速公路并排有多个车道。

当谈论**并发**的时候一定要加个单位时间,也就是说单位时间内并发量是多少?离开了单位时间其实是没有意义的。

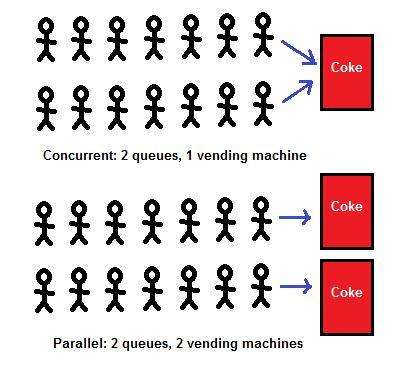
俗话说,一心不能二用,这对计算机也一样,原则上一个CPU只能分配给一个进程,以便运行这个进程。我们通常使用的计算机中只有一个CPU,也就是说只有一颗心,要让它一心多用同时运行多个进程,就必须使用并发技术。实现并发技术相当复杂,最容易理解的是“时间片轮转进程调度算法”。

综合来说：

并发:指应用能够交替执行不同的任务,比如单CPU核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已.

并行:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行

两者区别:一个是交替执行,一个是同时执行.



### 高并发编程的意义、好处和注意事项

由于多核多线程的CPU的诞生,多线程、高并发的编程越来越受重视和关注。多线程可以给程序带来如下好处。

(1)充分利用CPU的资源

从上面的CPU的介绍,可以看的出来,现在市面上没有CPU的内核不使用多线程并发机制的,特别是服务器还不止一个CPU,如果还是使用单线程的技术做思路,明显就out了。因为程序的基本调度单元是线程,并且一个线程也只能在一个CPU的一个核的一个线程跑,如果你是个i3的CPU的话,最差也是双核心4线程的运算能力:如果是一个线程的程序的话,那是要浪费3/4的CPU性能:如果设计一个多线程的程序的话,那它就可以同时在多个CPU的多个核的多个线程上跑,可以充分地利用CPU,减少CPU的空闲时间,发挥它的运算能力,提高并发量。

就像我们平时坐地铁一样,很多人坐长线地铁的时候都在认真看书,而不是为了坐地铁而坐地铁,到家了再去看书,这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕,而有些人老是说没时间的一个原因,工作也是这样,有的时候可以并发地去做几件事情,充分利用我们的时间,CPU也是一样,也要充分利用。

(2)加快响应用户的时间

比如我们经常用的迅雷下载,都喜欢多开几个线程去下载,谁都不愿意用一个线程去下载,为什么呢?答案很简单,就是多个线程下载快啊。

我们在做程序开发的时候更应该如此,特别是我们做互联网项目,网页的响应时间若提升1s,如果流量大的话,就能增加不少转换量。做过高性能web前端调优的都知道,要将静态资源地址用两三个子域名去加载,为什么?因为每多一个子域名,浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源,提升网站的响应速度。多线程,高并发真的是无处不在。

(3)可以使你的代码模块化,异步化,简单化

例如我们实现电商系统，下订单和给用户发送短信、邮件就可以进行拆分，将给用户发送短信、邮件这两个步骤独立为单独的模块，并交给其他线程去执行。这样既增加了异步的操作，提升了系统性能，又使程序模块化,清晰化和简单化。

多线程应用开发的好处还有很多,大家在日后的代码编写过程中可以慢慢体会它的魅力。

### 多线程程序需要注意事项

(1)线程之间的安全性

从前面的章节中我们都知道,在同一个进程里面的多线程是资源共享的,也就是都可以访问同一个内存地址当中的一个变量。例如:若每个线程中对全局变量、静态变量只有读操作,而无写操作,一般来说,这个全局变量是线程安全的:若有多个线程同时执行写操作,一般都需要考虑线程同步,否则就可能影响线程安全。

(2)线程之间的死锁

为了解决线程之间的安全性引入了Java的锁机制,而一不小心就会产生Java线程死锁的多线程问题,因为不同的线程都在等待那些根本不可能被释放的锁,从而导致所有的工作都无法完成。假设有两个线程,分别代表两个饥饿的人,他们必须共享刀叉并轮流吃饭。他们都需要获得两个锁:共享刀和共享叉的锁。

假如线程A获得了刀,而线程B获得了叉。线程A就会进入阻塞状态来等待获得叉,而线程B则阻塞来等待线程A所拥有的刀。这只是人为设计的例子,但尽管在运行时很难探测到,这类情况却时常发生

(3)线程太多了会将服务器资源耗尽形成死机当机

线程数太多有可能造成系统创建大量线程而导致消耗完系统内存以及CPU的“过渡切换”,造成系统的死机,那么我们该如何解决这类问题呢?

某些系统资源是有限的,如文件描述符。多线程程序可能耗尽资源,因为每个线程都可能希望有一个这样的资源。如果线程数相当大,或者某个资源的侯选线程数远远超过了可用的资源数则最好使用资源池。一个最好的示例是数据库连接池。只要线程需要使用一个数据库连接,它就从池中取出一个,使用以后再将它返回池中。资源池也称为资源库。

多线程应用开发的注意事项很多,希望大家在日后的工作中可以慢慢体会它的危险所在。

## 认识Java里的线程

### Java程序天生就是多线程的

一个Java程序从main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。

[6] Monitor Ctrl-Break //监控Ctrl-Break中断信号的

[5] Attach Listener //内存dump，线程dump，类信息统计，获取系统属性等

[4] Signal Dispatcher // 分发处理发送给JVM信号的线程

[3] Finalizer // 调用对象finalize方法的线程

[2] Reference Handler//清除Reference的线程

[1] main //main线程，用户程序入口

### 线程的启动与中止

#### 启动

启动线程的方式有：

1、X extends Thread;，然后X.start

2、X implements Runnable；然后交给Thread运行

参见代码：cn.enjoyedu.ch1.base.NewThread

##### Thread和Runnable的区别

Thread才是Java里对线程的唯一抽象，Runnable只是对任务（业务逻辑）的抽象。Thread可以接受任意一个Runnable的实例并执行。

#### 中止

##### 线程自然终止

要么是run执行完成了，要么是抛出了一个未处理的异常导致线程提前结束。

##### stop

暂停、恢复和停止操作对应在线程Thread的API就是**suspend()、resume()和stop()**。但是这些API是过期的，也就是不建议使用的。不建议使用的原因主要有：以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。正因为suspend()、resume()和stop()方法带来的副作用，这些方法才被标注为不建议使用的过期方法。

##### 中断

安全的中止则是其他线程通过调用某个线程A的**interrupt()**方法对其进行中断操作, 中断好比其他线程对该线程打了个招呼，“A，你要中断了”，不代表线程A会立即停止自己的工作，同样的A线程完全可以不理会这种中断请求。因为java里的线程是协作式的，不是抢占式的。线程通过检查自身的中断标志位是否被置为true来进行响应，

线程通过方法**isInterrupted()**来进行判断是否被中断，也可以调用静态方法**Thread.interrupted()**来进行判断当前线程是否被中断，不过Thread.interrupted()会同时将中断标识位改写为false。

如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、thread.wait等），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为false。

不建议自定义一个取消标志位来中止线程的运行。因为run方法里有阻塞调用时会无法很快检测到取消标志，线程必须从阻塞调用返回后，才会检查这个取消标志。这种情况下，使用中断会更好，因为，

一、一般的阻塞方法，如sleep等本身就支持中断的检查，

二、检查中断位的状态和检查取消标志位没什么区别，用中断位的状态还可以避免声明取消标志位，减少资源的消耗。

**注意：处于死锁状态的线程无法被中断**

## 对Java里的线程再多一点点认识

### 深入理解run()和start()

Thread类是Java里对线程概念的抽象，可以这样理解：我们通过new Thread()其实只是new出一个Thread的实例，还没有操作系统中真正的线程挂起钩来。只有执行了start()方法后，才实现了真正意义上的启动线程。

start()方法让一个线程进入就绪队列等待分配cpu，分到cpu后才调用实现的run()方法，start()方法不能重复调用，如果重复调用会抛出异常。

而run方法是业务逻辑实现的地方，本质上和任意一个类的任意一个成员方法并没有任何区别，可以重复执行，也可以被单独调用。

### 其他的线程相关方法

yield()方法：使当前线程让出CPU占有权，但让出的时间是不可设定的。也不会释放锁资源。注意：并不是每个线程都需要这个锁的，而且执行yield( )的线程不一定就会持有锁，我们完全可以在释放锁后再调用yield方法。

所有执行yield()的线程有可能在进入到就绪状态后会被操作系统再次选中马上又被执行。

wait()/notify()/notifyAll()：后面会单独讲述

### join方法

把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。(此处为常见面试考点)

### 线程的优先级

在Java线程中，通过一个整型成员变量priority来控制优先级，优先级的范围从1~10，在线程构建的时候可以通过setPriority(int)方法来修改优先级，默认优先级是5，优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者I/O操作）的线程需要设置较高优先级，而偏重计算（需要较多CPU时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。在不同的JVM以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

### 守护线程

Daemon（守护）线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着，当一个Java虚拟机中不存在**非**Daemon线程的时候，Java虚拟机将会退出。可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程。我们一般用不上，比如垃圾回收线程就是Daemon线程。

Daemon线程被用作完成支持性工作，但是在Java虚拟机退出时Daemon线程中的finally块并不一定会执行。在构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

## 线程间的共享和协作

### 线程间的共享

#### synchronized内置锁

线程开始运行，拥有自己的栈空间，就如同一个脚本一样，按照既定的代码一步一步地执行，直到终止。但是，每个运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，包括数据之间的共享，协同处理事情。这将会带来巨大的价值。

Java支持多个线程同时访问一个对象或者对象的成员变量，关键字synchronized可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性，又称为内置锁机制。

##### 对象锁和类锁：

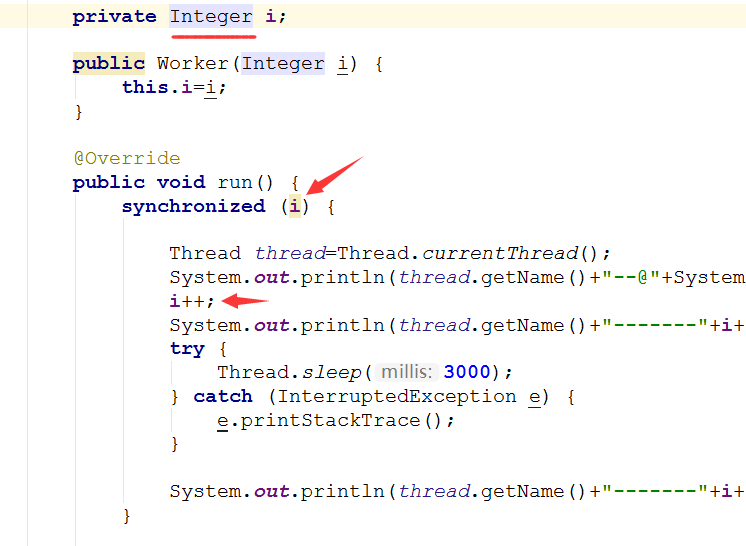
对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的class对象上的。我们知道，类的对象实例可以有很多个，但是每个类只有一个class对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。

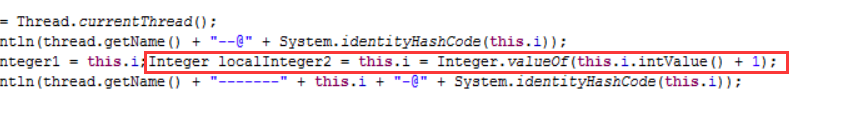
但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，类锁其实锁的是每个类的对应的class对象。类锁和对象锁之间也是互不干扰的。

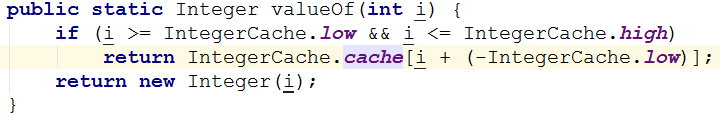
对象锁和类锁，以及锁static变量之间的运行情况，请参考包cn.enjoyedu.ch1.syn下的代码。

#### 错误的加锁和原因分析

参见代码cn.enjoyedu.ch1. syn.TestIntegerSyn

原因：虽然我们对i进行了加锁，但是

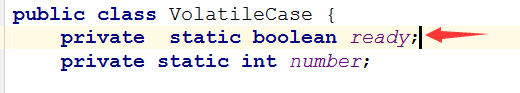
但是当我们反编译这个类的class文件后，可以看到i++实际是，



本质上是返回了一个新的Integer对象。也就是每个线程实际加锁的是不同的Integer对象。

#### volatile，最轻量的同步机制

volatile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。参见代码：cn.enjoyedu.ch1.vola. VolatileCase



不加volatile时，子线程无法感知主线程修改了ready的值，从而不会退出循环，而加了volatile后，子线程可以感知主线程修改了ready的值，迅速退出循环。

但是volatile不能保证数据在多个线程下同时写时的线程安全，参见代码：

cn.enjoyedu.ch1.vola. NotSafe

volatile最适用的场景：一个线程写，多个线程读。

### ThreadLocal辨析

#### 与Synchonized的比较

ThreadLocal和Synchonized都用于解决多线程并发訪问。可是ThreadLocal与synchronized有本质的差别。synchronized是利用锁的机制，使变量或代码块在某一时该仅仅能被一个线程訪问。而ThreadLocal为每个线程都提供了变量的副本，使得每个线程在某一时间訪问到的并非同一个对象，这样就隔离了多个线程对数据的数据共享。

Spring的事务就借助了ThreadLocal类。Spring会从数据库连接池中获得一个connection，然会把connection放进ThreadLocal中，也就和线程绑定了，事务需要提交或者回滚，只要从ThreadLocal中拿到connection进行操作。为何Spring的事务要借助ThreadLocal类？

以JDBC为例，正常的事务代码可能如下：

dbc = new DataBaseConnection();//第1行

Connection con = dbc.getConnection();//第2行

con.setAutoCommit(false);// //第3行

con.executeUpdate(...);//第4行

con.executeUpdate(...);//第5行

con.executeUpdate(...);//第6行

con.commit();////第7行

上述代码，可以分成三个部分:

事务准备阶段：第1～3行

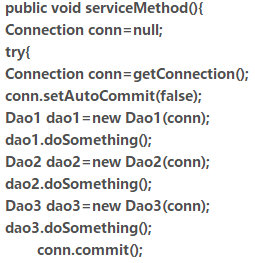
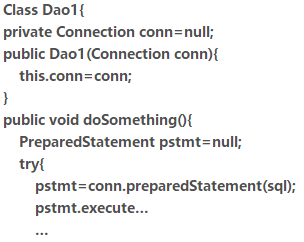
业务处理阶段：第4～6行

事务提交阶段：第7行

可以很明显的看到，不管我们开启事务还是执行具体的sql都需要一个具体的数据库连接。

现在我们开发应用一般都采用三层结构，如果我们控制事务的代码都放在DAO(DataAccessObject)对象中，在DAO对象的每个方法当中去打开事务和关闭事务，当Service对象在调用DAO时，如果只调用一个DAO，那我们这样实现则效果不错，但往往我们的Service会调用一系列的DAO对数据库进行多次操作，那么，这个时候我们就无法控制事务的边界了，因为实际应用当中，我们的Service调用的DAO的个数是不确定的，可根据需求而变化，而且还可能出现Service调用Service的情况。

如果不使用ThreadLocal，代码大概就会是这个样子：

但是需要注意一个问题，如何让三个DAO使用同一个数据源连接呢？我们就必须为每个DAO传递同一个数据库连接，要么就是在DAO实例化的时候作为构造方法的参数传递，要么在每个DAO的实例方法中作为方法的参数传递。这两种方式无疑对我们的Spring框架或者开发人员来说都不合适。为了让这个数据库连接可以跨阶段传递，又不显示的进行参数传递，就必须使用别的办法。

Web容器中，每个完整的请求周期会由一个线程来处理。因此，如果我们能将一些参数绑定到线程的话，就可以实现在软件架构中跨层次的参数共享（是隐式的共享）。而JAVA中恰好提供了绑定的方法--使用ThreadLocal。

结合使用Spring里的IOC和AOP，就可以很好的解决这一点。

只要将一个数据库连接放入ThreadLocal中，当前线程执行时只要有使用数据库连接的地方就从ThreadLocal获得就行了。

#### ThreadLocal的使用

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

• void set(Object value)

设置当前线程的线程局部变量的值。

• public Object get()

该方法返回当前线程所对应的线程局部变量。

• public void remove()

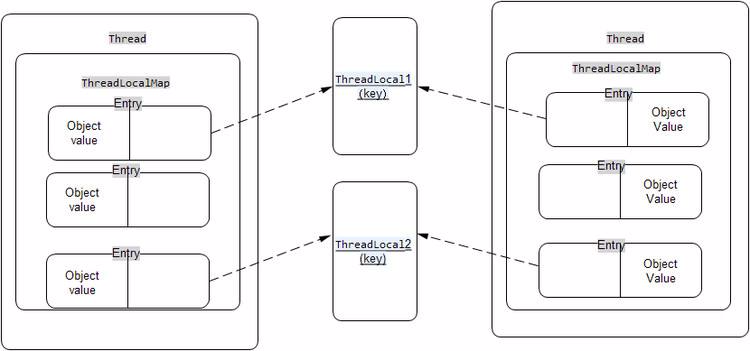
将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

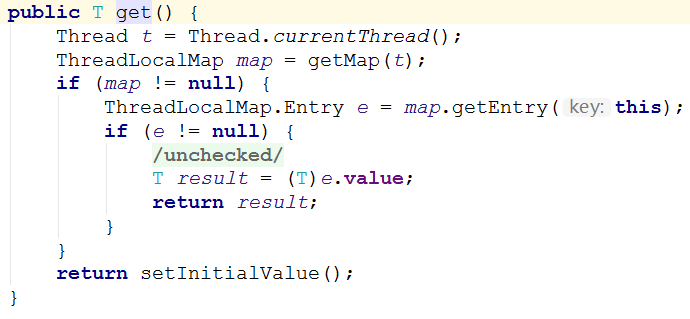
• protected Object initialValue()

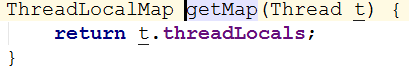
返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

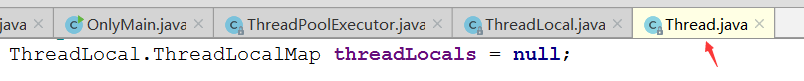
public final static ThreadLocal<String> RESOURCE = new ThreadLocal<String>();RESOURCE代表一个能够存放String类型的ThreadLocal对象。此时不论什么一个线程能够并发访问这个变量，对它进行写入、读取操作，都是线程安全的。

#### 实现解析



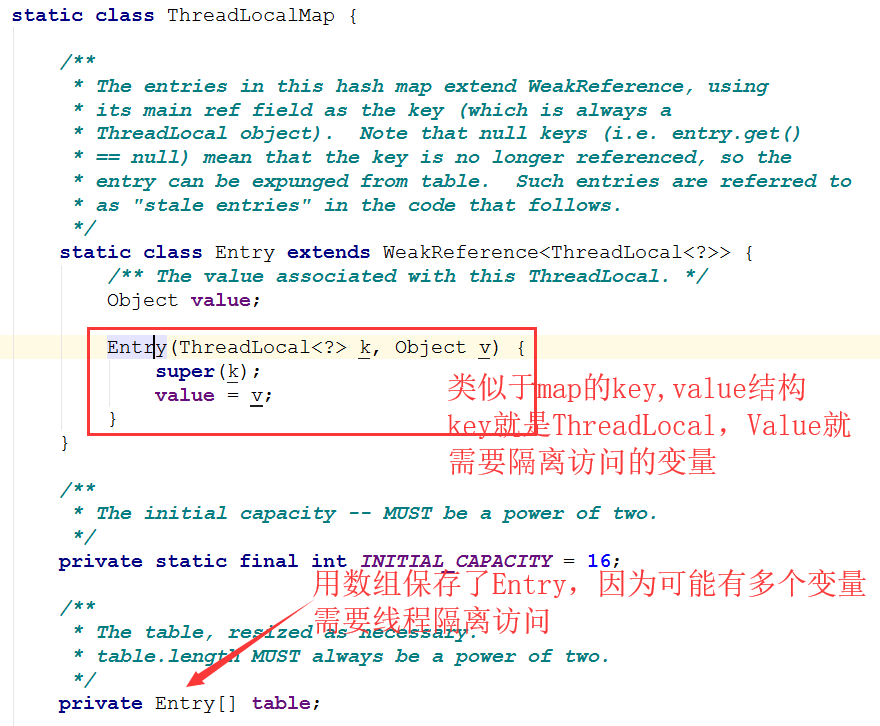






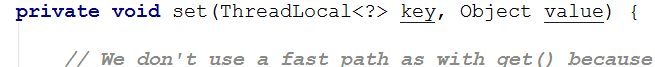
上面先取到当前线程，然后调用getMap方法获取对应的ThreadLocalMap，ThreadLocalMap是ThreadLocal的静态内部类，然后Thread类中有一个这样类型成员，所以getMap是直接返回Thread的成员。

看下ThreadLocal的内部类ThreadLocalMap源码：



可以看到有个Entry内部静态类，它继承了WeakReference，总之它记录了两个信息，一个是ThreadLocal<?>类型，一个是Object类型的值。getEntry方法则是获取某个ThreadLocal对应的值，set方法就是更新或赋值相应的ThreadLocal对应的值。





回顾我们的get方法，其实就是拿到**每个线程独有的ThreadLocalMap**

然后再用ThreadLocal的当前实例，拿到Map中的相应的Entry，然后就可以拿到相应的值返回出去。当然，如果Map为空，还会先进行map的创建，初始化等工作。

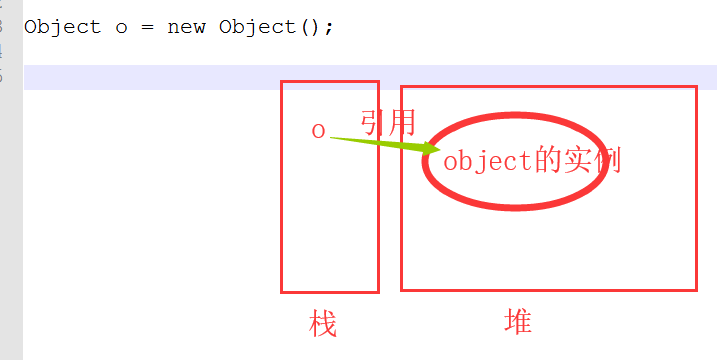
#### 引发的内存泄漏分析

##### 预备知识

引用

Object o = new Object();

这个o，我们可以称之为对象引用，而new Object()我们可以称之为在内存中产生了一个对象实例。

****

当写下 **o=null**时，只是表示o不再指向堆中object的对象实例，不代表这个对象实例不存在了。

**强引用**就是指在程序代码之中普遍存在的，类似“Object obj=new Object（）”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象实例。

**软引用**是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象实例列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK 1.2之后，提供了SoftReference类来实现软引用。

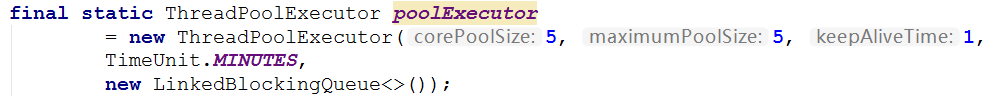
**弱引用**也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象实例只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象实例。在JDK 1.2之后，提供了WeakReference类来实现弱引用。

**虚引用**也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象实例是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象实例被收集器回收时收到一个系统通知。在JDK 1.2之后，提供了PhantomReference类来实现虚引用。

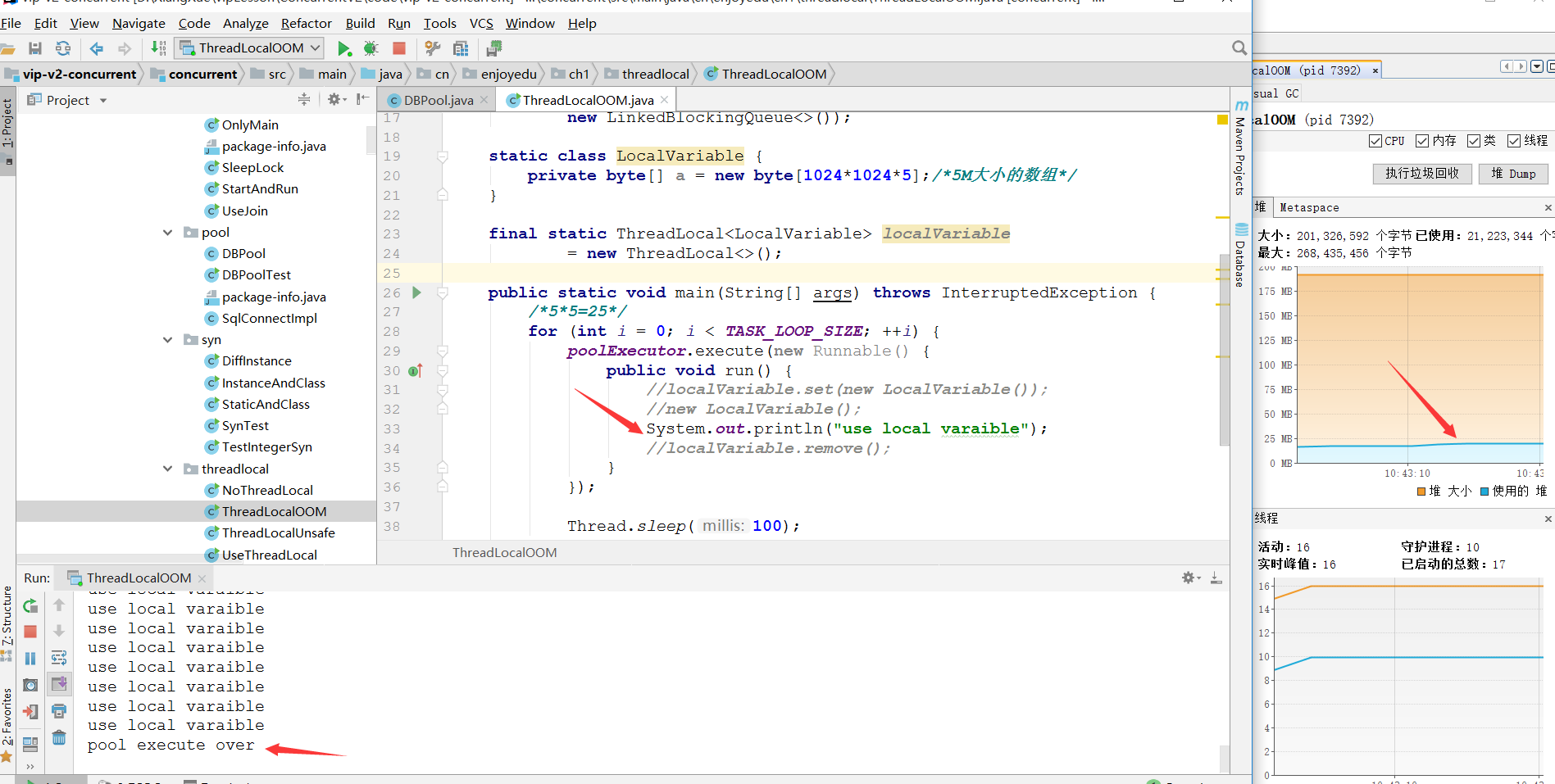
##### 内存泄漏的现象

执行cn.enjoyedu.ch1.threadlocal下的ThreadLocalOOM，并将堆内存大小设置为-Xmx256m，

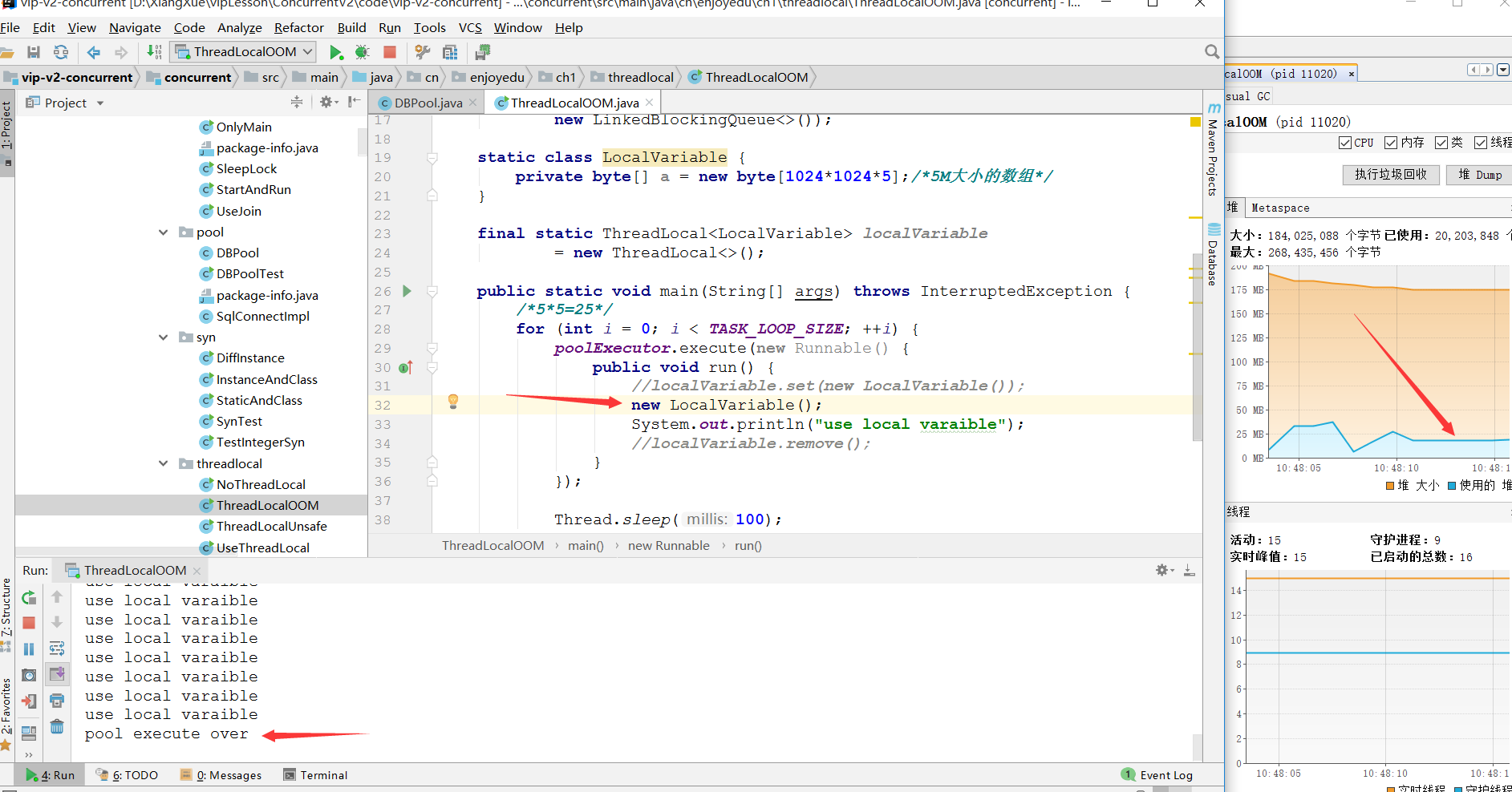
我们启用一个线程池，大小固定为5个线程



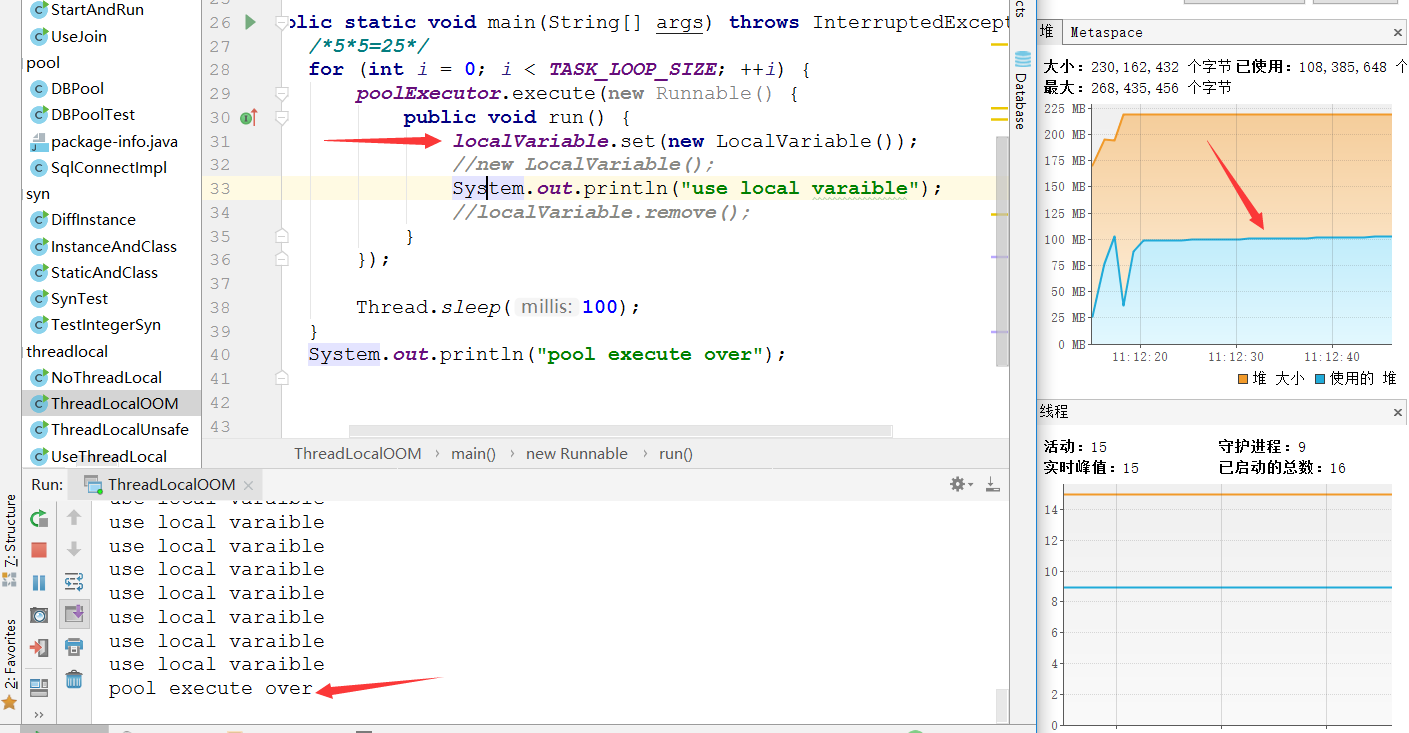
场景1，首先任务中不执行任何有意义的代码，当所有的任务提交执行完成后，可以看见，我们这个应用的内存占用基本上为25M左右



场景2，然后我们只简单的在每个任务中new出一个数组，执行完成后我们可以看见，内存占用基本和场景1同

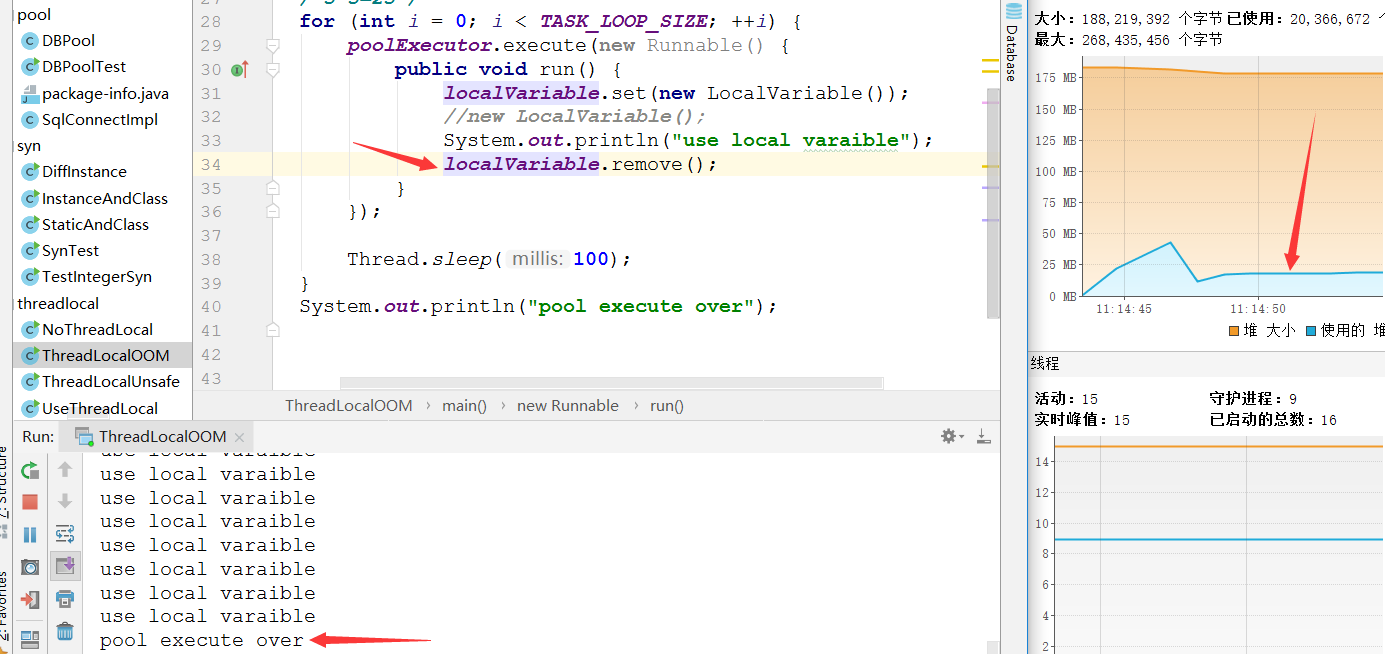


场景3，当我们启用了ThreadLocal以后：



执行完成后我们可以看见，内存占用变为了100M左右

场景4，于是，我们加入一行代码，再执行，看看内存情况:



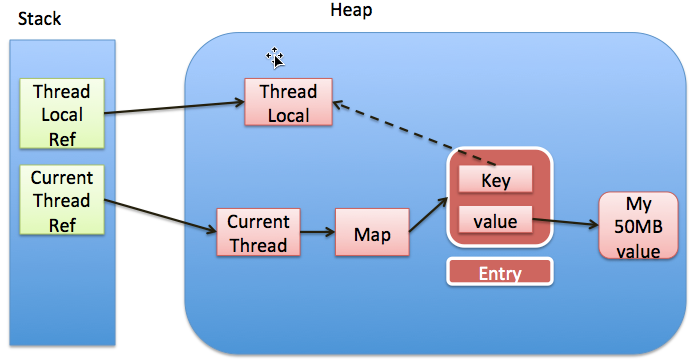
可以看见，内存占用基本和场景1同。

这就充分说明，场景3，当我们启用了ThreadLocal以后确实发生了内存泄漏。

##### 分析

根据我们前面对ThreadLocal的分析，我们可以知道每个Thread 维护一个 ThreadLocalMap，这个映射表的 key 是 ThreadLocal实例本身，value 是真正需要存储的 Object，也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。仔细观察ThreadLocalMap，这个map是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。

因此使用了ThreadLocal后，引用链如图所示



图中的虚线表示弱引用。

这样，当把threadlocal变量置为null以后，没有任何强引用指向threadlocal实例，所以threadlocal将会被gc回收。这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value，而这块value永远不会被访问到了，所以存在着内存泄露。

只有当前thread结束以后，current thread就不会存在栈中，强引用断开，Current Thread、Map value将全部被GC回收。最好的做法是不在需要使用ThreadLocal变量后，都调用它的remove()方法，清除数据。

所以回到我们前面的实验场景，场景3中，虽然线程池里面的任务执行完毕了，但是线程池里面的5个线程会一直存在直到JVM退出，我们set了线程的localVariable变量后没有调用localVariable.remove()方法，导致线程池里面的5个线程的threadLocals变量里面的new LocalVariable()实例没有被释放。

其实考察ThreadLocal的实现，我们可以看见，无论是get()、set()在某些时候，调用了expungeStaleEntry方法用来清除Entry中Key为null的Value，但是这是不及时的，也不是每次都会执行的，所以一些情况下还是会发生内存泄露。只有remove()方法中显式调用了expungeStaleEntry方法。

从表面上看内存泄漏的根源在于使用了弱引用，但是另一个问题也同样值得思考：为什么使用弱引用而不是强引用？

下面我们分两种情况讨论：

key 使用强引用：对ThreadLocal对象实例的引用被置为null了，但是ThreadLocalMap还持有这个ThreadLocal对象实例的强引用，如果没有手动删除，ThreadLocal的对象实例不会被回收，导致Entry内存泄漏。

key 使用弱引用：对ThreadLocal对象实例的引用被被置为null了，由于ThreadLocalMap持有ThreadLocal的弱引用，即使没有手动删除，ThreadLocal的对象实例也会被回收。value在下一次ThreadLocalMap调用set，get，remove都有机会被回收。

比较两种情况，我们可以发现：由于ThreadLocalMap的生命周期跟Thread一样长，如果都没有手动删除对应key，都会导致内存泄漏，但是使用弱引用可以多一层保障。

因此，ThreadLocal内存泄漏的根源是：由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏，而不是因为弱引用。

##### 总结

JVM利用设置ThreadLocalMap的Key为弱引用，来避免内存泄露。

JVM利用调用remove、get、set方法的时候，回收弱引用。

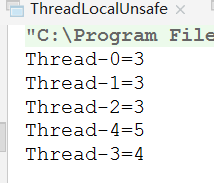
当ThreadLocal存储很多Key为null的Entry的时候，而不再去调用remove、get、set方法，那么将导致内存泄漏。

使用**线程池+** ThreadLocal时要小心，因为这种情况下，线程是一直在不断的重复运行的，从而也就造成了value可能造成累积的情况。

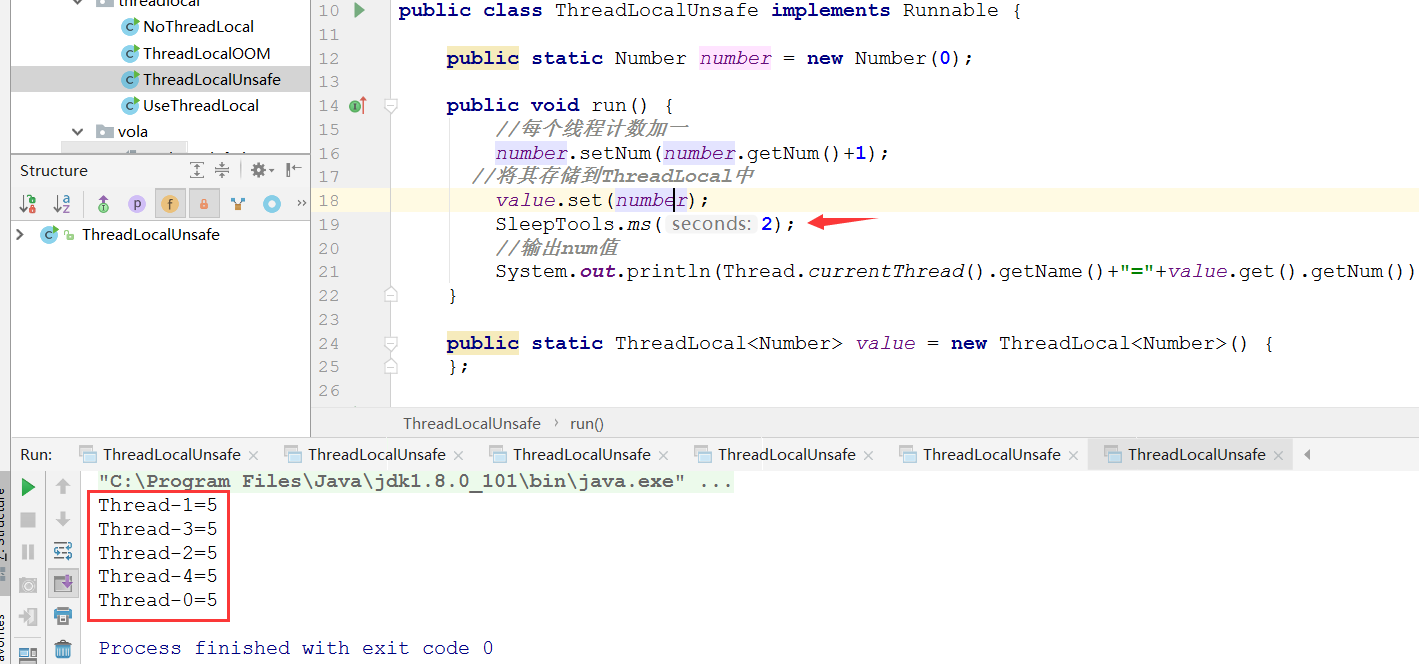
#### 错误使用ThreadLocal导致线程不安全

参见代码cn.enjoyedu.ch1.threadlocal. ThreadLocalUnsafe

运行后的结果为



如果我们加入SleepTools.ms(2);会看的更明显



为什么每个线程都输出5？难道他们没有独自保存自己的Number副本吗？为什么其他线程还是能够修改这个值？仔细考察ThreadLocal和Thead的代码，我们发现ThreadLocalMap中保存的其实是对象的一个引用，这样的话，当有其他线程对这个引用指向的对象实例做修改时，其实也同时影响了所有的线程持有的对象引用所指向的同一个对象实例。这也就是为什么上面的程序为什么会输出一样的结果：5个线程中保存的是同一Number对象的引用，在线程睡眠的时候，其他线程将num变量进行了修改，而修改的对象Number的实例是同一份，因此它们最终输出的结果是相同的。

而上面的程序要正常的工作，应该的用法是让每个线程中的ThreadLocal都应该持有一个新的Number对象。

### 线程间的协作

线程之间相互配合，完成某项工作，比如：一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（what）和“怎么做”（How），简单的办法是让消费者线程不断地循环检查变量是否符合预期在while循环中设置不满足的条件，如果条件满足则退出while循环，从而完成消费者的工作。却存在如下问题：

1） 难以确保及时性。

2）难以降低开销。如果降低睡眠的时间，比如休眠1毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

#### 等待/通知机制

是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

notify()：

通知一个在对象上等待的线程,使其从wait方法返回,而返回的前提是该线程获取到了对象的锁，没有获得锁的线程重新进入WAITING状态。

notifyAll()：

通知所有等待在该对象上的线程

wait()

调用该方法的线程进入 WAITING状态,只有等待另外线程的通知或被中断才会返回.需要注意,调用wait()方法后,会释放对象的锁

wait(long)

超时等待一段时间,这里的参数时间是毫秒,也就是等待长达n毫秒,如果没有通知就超时返回

wait (long,int)

对于超时时间更细粒度的控制,可以达到纳秒

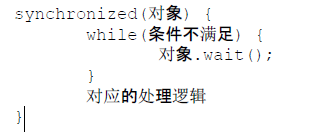
##### 等待和通知的标准范式

等待方遵循如下原则。

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

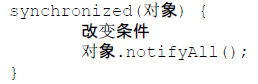


通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。



**在调用wait（）、notify()系列方法之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用wait（）方法、notify()系列方法**，进入wait（）方法后，当前线程释放锁，在从wait（）返回前，线程与其他线程竞争重新获得锁， 执行notify()系列方法的线程退出调用了notifyAll的synchronized代码块的时候后，他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出synchronized代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

##### notify和notifyAll应该用谁

尽可能用notifyall()，谨慎使用notify()，因为notify()只会唤醒一个线程，我们无法确保被唤醒的这个线程一定就是我们需要唤醒的线程，具体表现参见代码:包cn.enjoyedu.ch1.wn 下

##### 等待超时模式实现一个连接池

调用场景：调用一个方法时等待一段时间（一般来说是给定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。

假设等待时间段是T，那么可以推断出在当前时间now+T之后就会超时

等待持续时间：REMAINING=T。

•超时时间：FUTURE=now+T。

// 对当前对象加锁

public synchronized Object get(long mills) throws InterruptedException {

long future = System.currentTimeMillis() + mills;

long remaining = mills;

// 当超时大于0并且result返回值不满足要求

while ((result == null) && remaining > 0) {

wait(remaining);

remaining = future - System.currentTimeMillis();

}

return result;

}

具体实现参见：包下cn.enjoyedu.ch1.pool的代码

客户端获取连接的过程被设定为等待超时的模式，也就是在1000毫秒内如果无法获取到可用连接，将会返回给客户端一个null。设定连接池的大小为10个，然后通过调节客户端的线程数来模拟无法获取连接的场景。

它通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用fetchConnection(long)方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用releaseConnection(Connection)方法将连接放回线程池

## 面试题

调用yield() 、sleep()、wait()、notify()等方法对锁有何影响？

yield() 、sleep()被调用后，都不会释放当前线程所持有的锁。

调用wait()方法后，会释放当前线程持有的锁，而且当前被唤醒后，会重新去竞争锁，锁竞争到后才会执行wait方法后面的代码。

调用notify()系列方法后，对锁无影响，线程只有在syn同步代码执行完后才会自然而然的释放锁，所以notify()系列方法一般都是syn同步代码的最后一行。

# 2、线程的并发工具类

## Fork-Join

java下多线程的开发可以我们自己启用多线程，线程池，还可以使用forkjoin，forkjoin可以让我们不去了解诸如Thread,Runnable等相关的知识，只要遵循forkjoin的开发模式，就可以写出很好的多线程并发程序，

### 分而治之

同时forkjoin在处理某一类问题时非常的有用，哪一类问题？分而治之的问题。十大计算机经典算法：快速排序、堆排序、归并排序、二分查找、线性查找、

深度优先、广度优先、Dijkstra、动态规划、朴素贝叶斯分类，有几个属于分而治之？3个，快速排序、归并排序、二分查找，还有大数据中M/R都是。

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，**这些子问题互相独立且与原问题形式相同(子问题相互之间有联系就会变为动态规范算法)**，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

#### 归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

若将两个有序表合并成一个有序表，称为2-路归并，与之对应的还有多路归并。

对于给定的一组数据，利用递归与分治技术将数据序列划分成为越来越小的半子表，在对半子表排序后，再用递归方法将排好序的半子表合并成为越来越大的有序序列。

为了提升性能，有时我们在半子表的个数小于某个数（比如15）的情况下，对半子表的排序采用其他排序算法，比如插入排序。

#### 归并排序（降序）示例



先讲数组划分为左右两个子表：

，

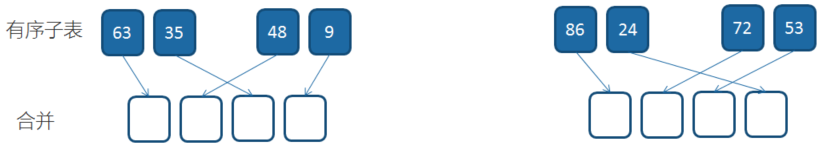
然后继续左右两个子表拆分：



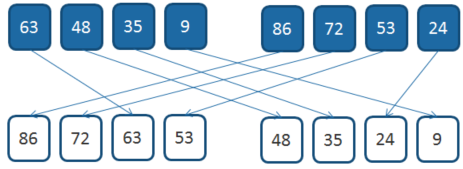
对最后的拆分的子表，两两进行排序



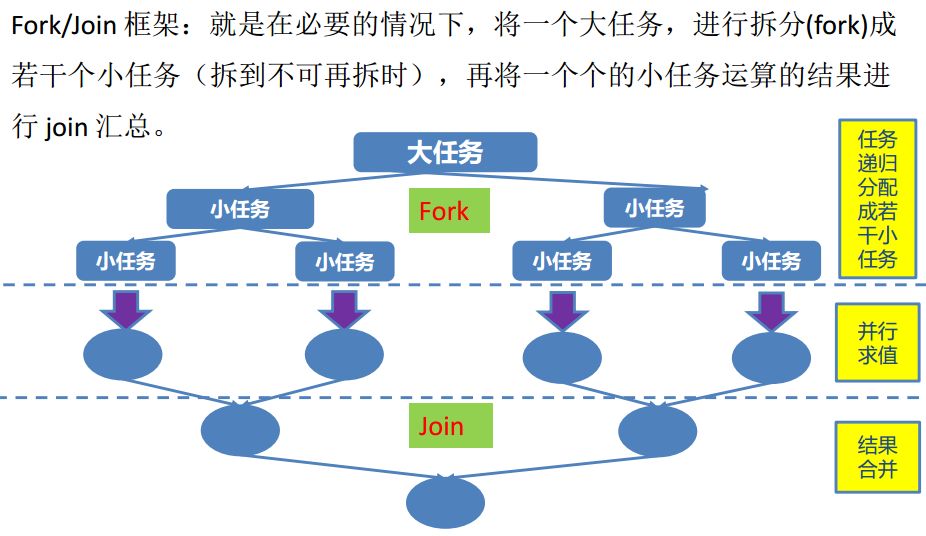
对有序的子表进行排序和比较合并



对合并后的子表继续比较合并



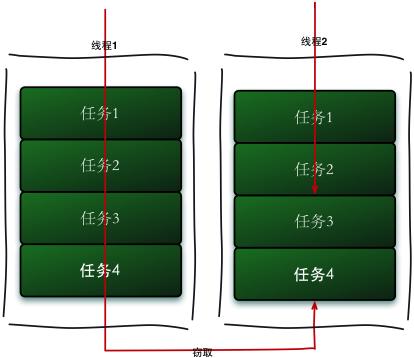
### Fork-Join原理



#### 工作密取

即当前线程的Task已经全被执行完毕，则自动取到其他线程的Task池中取出Task继续执行。

ForkJoinPool中维护着多个线程（一般为CPU核数）在不断地执行Task，每个线程除了执行自己职务内的Task之外，还会根据自己工作线程的闲置情况去获取其他繁忙的工作线程的Task，如此一来就能能够减少线程阻塞或是闲置的时间，提高CPU利用率。



### Fork/Join实战

#### Fork/Join使用的标准范式

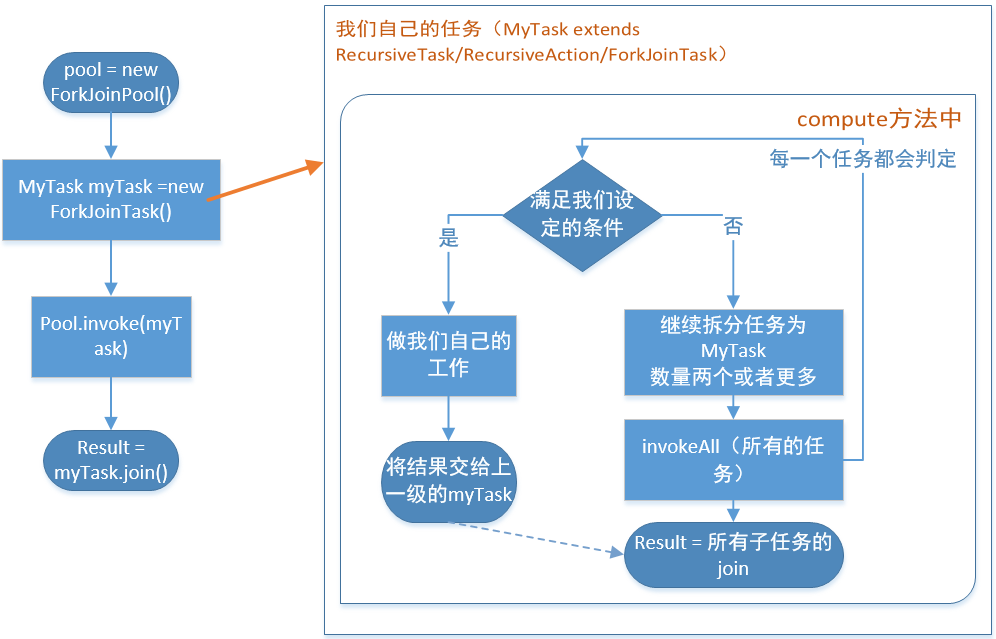
我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork和join的操作机制，通常我们不直接继承ForkjoinTask类，只需要直接继承其子类。

1. RecursiveAction，用于没有返回结果的任务

2. RecursiveTask，用于有返回值的任务

task要通过ForkJoinPool来执行，使用submit 或 invoke 提交，两者的区别是：invoke是同步执行，调用之后需要等待任务完成，才能执行后面的代码；submit是异步执行。

join()和get方法当任务完成的时候返回计算结果。



在我们自己实现的compute方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用invokeAll方法时，又会进入compute方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用join方法会等待子任务执行完并得到其结果。

#### Fork/Join的同步用法和异步用法

参见代码包cn.enjoyedu.ch2.forkjoin下

## CountDownLatch

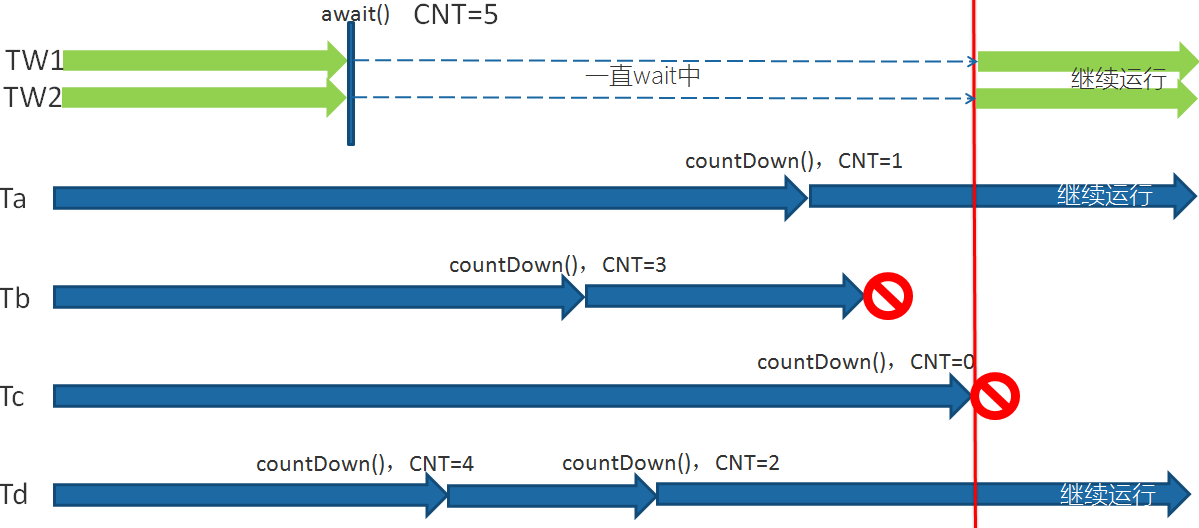
闭锁，CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行。

CountDownLatch是通过一个计数器来实现的，计数器的初始值为初始任务的数量。每当完成了一个任务后，计数器的值就会减1（CountDownLatch.countDown()方法）。当计数器值到达0时，它表示所有的已经完成了任务，然后在闭锁上等待CountDownLatch.await()方法的线程就可以恢复执行任务。

应用场景：

实现最大的并行性：有时我们想同时启动多个线程，实现最大程度的并行性。例如，我们想测试一个单例类。如果我们创建一个初始计数为1的CountDownLatch，并让所有线程都在这个锁上等待，那么我们可以很轻松地完成测试。我们只需调用 一次countDown()方法就可以让所有的等待线程同时恢复执行。

开始执行前等待n个线程完成各自任务：例如应用程序启动类要确保在处理用户请求前，所有N个外部系统已经启动和运行了，例如处理excel中多个表单。



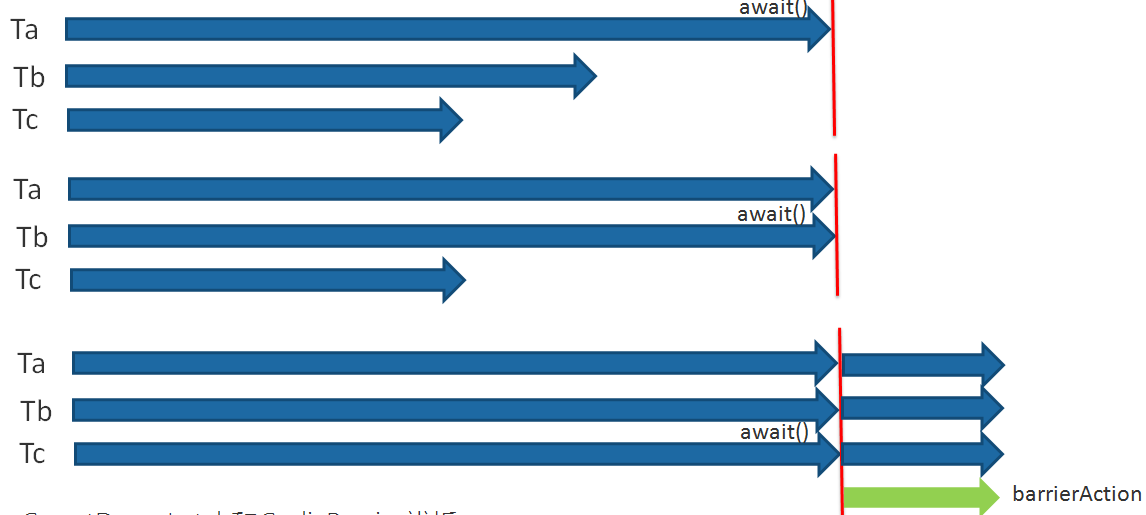
参见代码包cn.enjoyedu.ch2.tools下

## CyclicBarrier

CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。CyclicBarrier默认的构造方法是CyclicBarrier（int parties），其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier（int parties，Runnable barrierAction），用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的场景。



参见代码包cn.enjoyedu.ch2.tools下

## CountDownLatch和CyclicBarrier辨析

CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以反复使用。

CountDownLatch.await一般阻塞工作线程，所有的进行预备工作的线程执行countDown，而CyclicBarrier通过工作线程调用await从而自行阻塞，直到所有工作线程达到指定屏障，再大家一起往下走。

在控制多个线程同时运行上，CountDownLatch可以不限线程数量，而CyclicBarrier是固定线程数。

同时，CyclicBarrier还可以提供一个barrierAction，合并多线程计算结果。

## Semaphore

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。应用场景Semaphore可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接。假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发地读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有10个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，就可以使用Semaphore来做流量控制。。Semaphore的构造方法Semaphore（int permits）接受一个整型的数字，表示可用的许可证数量。Semaphore的用法也很简单，首先线程使用Semaphore的acquire()方法获取一个许可证，使用完之后调用release()方法归还许可证。还可以用tryAcquire()方法尝试获取许可证。

Semaphore还提供一些其他方法，具体如下。

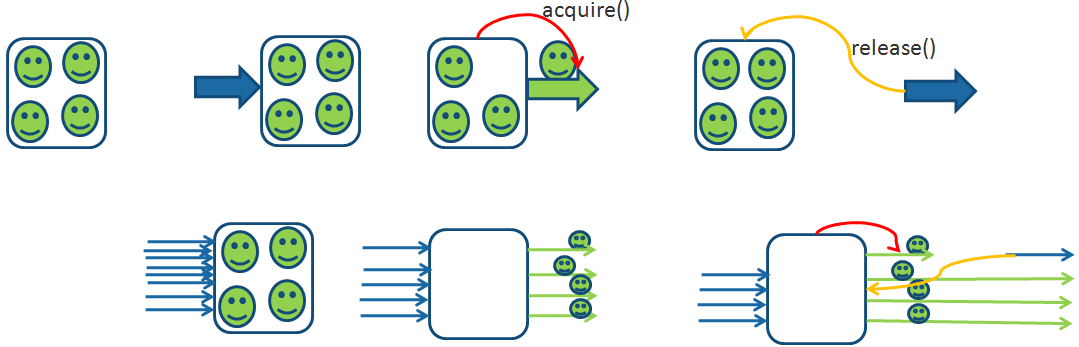
•intavailablePermits()：返回此信号量中当前可用的许可证数。

•intgetQueueLength()：返回正在等待获取许可证的线程数。

•booleanhasQueuedThreads()：是否有线程正在等待获取许可证。

•void reducePermits（int reduction）：减少reduction个许可证，是个protected方法。

•Collection getQueuedThreads()：返回所有等待获取许可证的线程集合，是个protected方法。



### 用Semaphore实现数据库连接池

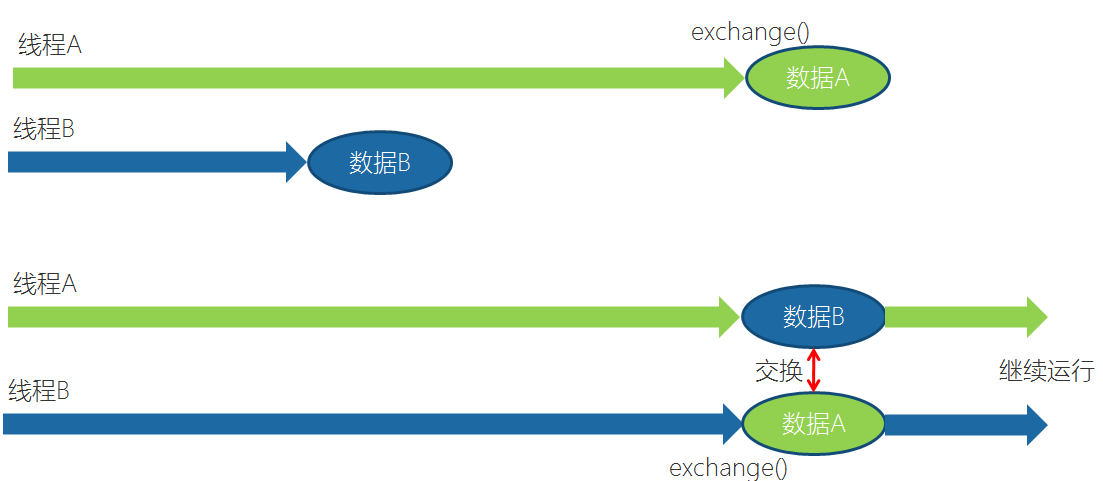
参见代码,包cn.enjoyedu.ch2.tools.semaphore下

### Semaphore注意事项

参见代码类cn.enjoyedu.ch2.tools.semaphore. DBPoolNoUseless下

## Exchange

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。



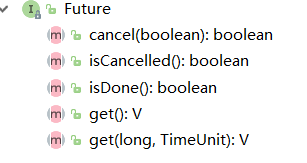
参见代码包cn.enjoyedu.ch2.tools下

## Callable、Future和FutureTask

Runnable是一个接口，在它里面只声明了一个run()方法，由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

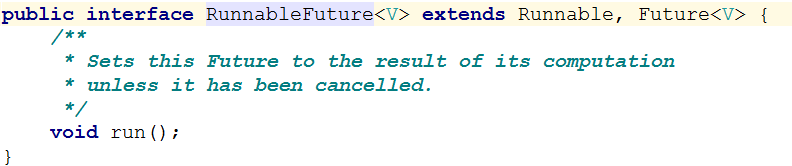
Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

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。

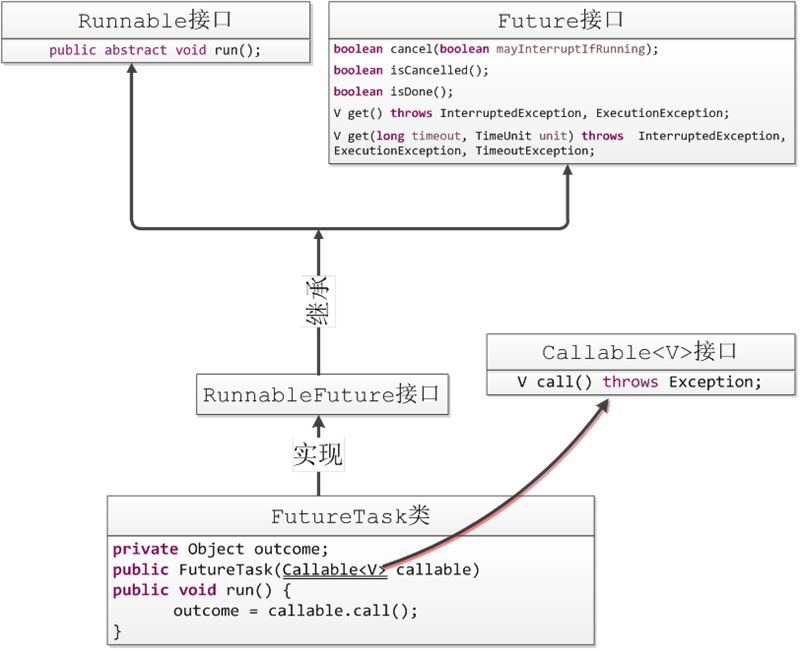


因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。



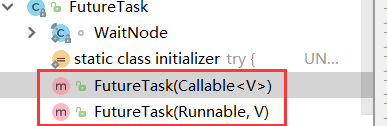


FutureTask类实现了RunnableFuture接口，RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。



因此我们通过一个线程运行Callable，但是Thread不支持构造方法中传递Callable的实例，所以我们需要通过FutureTask把一个Callable包装成Runnable，然后再通过这个FutureTask拿到Callable运行后的返回值。

要new一个FutureTask的实例，有两种方法



参见代码包cn.enjoyedu.ch2.tools下

# 3、原子操作CAS

## 什么是原子操作？如何实现原子操作？

假定有两个操作A和B，如果从执行A的线程来看，当另一个线程执行B时，要么将B全部执行完，要么完全不执行B，那么A和B对彼此来说是原子的。

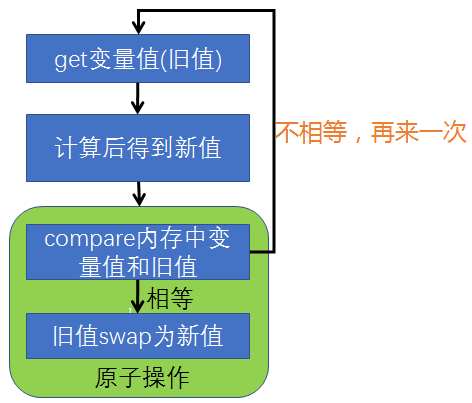
实现原子操作可以使用锁，锁机制，满足基本的需求是没有问题的了，但是有的时候我们的需求并非这么简单，我们需要更有效，更加灵活的机制，synchronized关键字是基于阻塞的锁机制，也就是说当一个线程拥有锁的时候，访问同一资源的其它线程需要等待，直到该线程释放锁，

这里会有些问题：首先，如果被阻塞的线程优先级很高很重要怎么办？其次，如果获得锁的线程一直不释放锁怎么办？（这种情况是非常糟糕的）。还有一种情况，如果有大量的线程来竞争资源，那CPU将会花费大量的时间和资源来处理这些竞争，同时，还有可能出现一些例如死锁之类的情况，最后，其实锁机制是一种比较粗糙，粒度比较大的机制，相对于像计数器这样的需求有点儿过于笨重。

实现原子操作还可以使用当前的处理器基本都支持CAS()的指令，只不过每个厂家所实现的算法并不一样，每一个CAS操作过程都包含三个运算符：一个内存地址V，一个期望的值A和一个新值B，操作的时候如果这个地址上存放的值等于这个期望的值A，则将地址上的值赋为新值B，否则不做任何操作。

CAS的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。循环CAS就是在一个循环里不断的做cas操作，直到成功为止。

CAS是怎么实现线程的安全呢？语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。



## CAS实现原子操作的三大问题

### ABA问题。

因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。

ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A→B→A就会变成1A→2B→3A。举个通俗点的例子，你倒了一杯水放桌子上，干了点别的事，然后同事把你水喝了又给你重新倒了一杯水，你回来看水还在，拿起来就喝，如果你不管水中间被人喝过，只关心水还在，这就是ABA问题。

如果你是一个讲卫生讲文明的小伙子，不但关心水在不在，还要在你离开的时候水被人动过没有，因为你是程序员，所以就想起了放了张纸在旁边，写上初始值0，别人喝水前麻烦先做个累加才能喝水。

### 循环时间长开销大。

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

### 只能保证一个共享变量的原子操作。

当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。

还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

## Jdk中相关原子操作类的使用

参见代码，包cn.enjoyedu.ch3下

### AtomicInteger

•int addAndGet（int delta）：以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。

•boolean compareAndSet（int expect，int update）：如果输入的数值等于预期值，则以原子方式将该值设置为输入的值。

•int getAndIncrement()：以原子方式将当前值加1，注意，这里返回的是自增前的值。

•int getAndSet（int newValue）：以原子方式设置为newValue的值，并返回旧值。

### AtomicIntegerArray

主要是提供原子的方式更新数组里的整型，其常用方法如下。

•int addAndGet（int i，int delta）：以原子方式将输入值与数组中索引i的元素相加。

•boolean compareAndSet（int i，int expect，int update）：如果当前值等于预期值，则以原子方式将数组位置i的元素设置成update值。

需要注意的是，数组value通过构造方法传递进去，然后AtomicIntegerArray会将当前数组复制一份，所以当AtomicIntegerArray对内部的数组元素进行修改时，不会影响传入的数组。

### 更新引用类型

原子更新基本类型的AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3个类。

#### AtomicReference

原子更新引用类型。

#### AtomicStampedReference

利用版本戳的形式记录了每次改变以后的版本号，这样的话就不会存在ABA问题了。这就是AtomicStampedReference的解决方案。AtomicMarkableReference跟AtomicStampedReference差不多， AtomicStampedReference是使用pair的int stamp作为计数器使用，AtomicMarkableReference的pair使用的是boolean mark。 还是那个水的例子，AtomicStampedReference可能关心的是动过几次，AtomicMarkableReference关心的是有没有被人动过，方法都比较简单。

#### AtomicMarkableReference：

原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference（V initialRef，booleaninitialMark）。

### 原子更新字段类

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类，Atomic包提供了以下3个类进行原子字段更新。

要想原子地更新字段类需要两步。第一步，因为原子更新字段类都是抽象类，每次使用的时候必须使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性。第二步，更新类的字段（属性）必须使用public volatile修饰符。

#### AtomicIntegerFieldUpdater：

原子更新整型的字段的更新器。

#### AtomicLongFieldUpdater：

原子更新长整型字段的更新器。

#### AtomicReferenceFieldUpdater：

原子更新引用类型里的字段。

# 4、显式锁和AQS

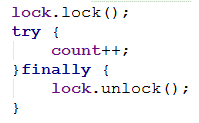
## 显式锁

有了synchronized为什么还要Lock？

Java程序是靠synchronized关键字实现锁功能的，使用synchronized关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放。



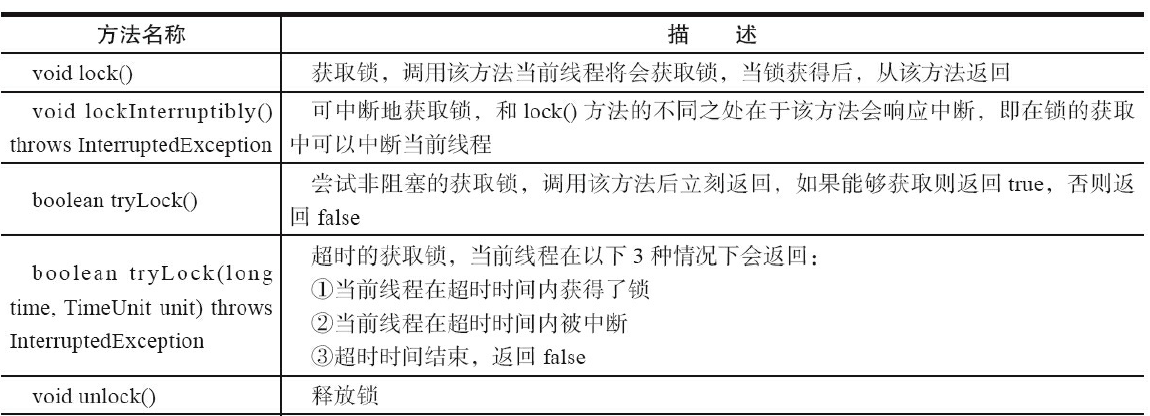
### Lock的标准用法



在finally块中释放锁，目的是保证在获取到锁之后，最终能够被释放。

不要将获取锁的过程写在try块中，因为如果在获取锁（自定义锁的实现）时发生了异常，异常抛出的同时，也会导致锁无故释放。

### Lock的常用API



### ReentrantLock

#### 锁的可重入

**简单地讲**就是：“同一个线程对于已经获得到的锁，可以多次继续申请到该锁的使用权”。而synchronized关键字隐式的支持重进入，比如一个synchronized修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获得该锁。ReentrantLock在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞。

#### 公平和非公平锁

如果在时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。 ReentrantLock提供了一个构造函数，能够控制锁是否是公平的。事实上，公平的锁机制往往没有非公平的效率高。

在激烈竞争的情况下,非公平锁的性能高于公平锁的性能的一个原因是:在恢复一个被挂起的线程与该线程真正开始运行之间存在着严重的延迟。假设线程A持有一个锁,并且线程B请求这个锁。由于这个锁已被线程A持有,因此B将被挂起。当A释放锁时,B将被唤醒,因此会再次尝试获取锁。与此同时,如果C也请求这个锁,那么C很可能会在B被完全唤醒之前获得、使用以及释放这个锁。这样的情况是一种“双赢”的局面:B获得锁的时刻并没有推迟,C更早地获得了锁,并且吞吐量也获得了提高。

### 读写锁ReentrantReadWriteLock

之前提到锁（如Mutex和ReentrantLock）基本都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

除了保证写操作对读操作的可见性以及并发性的提升之外，读写锁能够简化读写交互场景的编程方式。假设在程序中定义一个共享的用作缓存数据结构，它大部分时间提供读服务（例如查询和搜索），而写操作占有的时间很少，但是写操作完成之后的更新需要对后续的读服务可见。

在没有读写锁支持的（Java 5之前）时候，如果需要完成上述工作就要使用Java的等待通知机制，就是当写操作开始时，所有晚于写操作的读操作均会进入等待状态，只有写操作完成并进行通知之后，所有等待的读操作才能继续执行（写操作之间依靠synchronized关键进行同步），这样做的目的是使读操作能读取到正确的数据，不会出现脏读。改用读写锁实现上述功能，只需要在读操作时获取读锁，写操作时获取写锁即可。当写锁被获取到时，后续（非当前写操作线程）的读写操作都会被阻塞，写锁释放之后，所有操作继续执行，编程方式相对于使用等待通知机制的实现方式而言，变得简单明了。

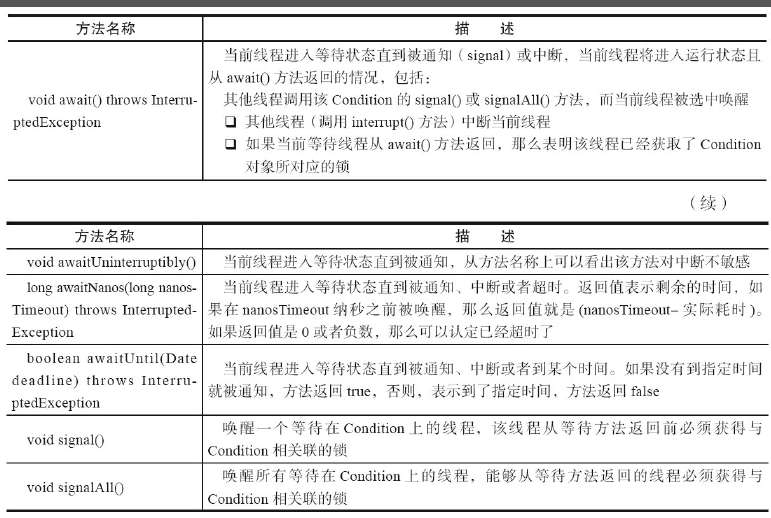
一般情况下，读写锁的性能都会比排它锁好，因为大多数场景读是多于写的。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量

ReentrantReadWriteLock其实实现的是ReadWriteLock接口

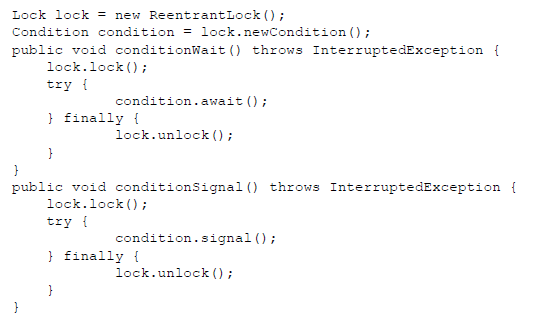
### Condition接口

任意一个Java对象，都拥有一组监视器方法（定义在java.lang.Object上），主要包括wait()、wait(long timeout)、notify()以及notifyAll()方法，这些方法与synchronized同步关键字配合，可以实现等待/通知模式。Condition接口也提供了类似Object的监视器方法，与Lock配合可以实现等待/通知模式。

#### Condition常用方法



#### Condition使用范式



#### Condition使用

参见cn.enjoyedu.ch4.condition. ExpressCond

## 了解LockSupport

LockSupport定义了一组的公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而LockSupport也成为构建同步组件的基础工具。

LockSupport定义了一组以park开头的方法用来阻塞当前线程，以及unpark(Thread thread)方法来唤醒一个被阻塞的线程。LockSupport增加了park(Object blocker)、parkNanos(Object blocker,long nanos)和parkUntil(Object blocker,long deadline)3个方法，用于实现阻塞当前线程的功能，其中参数blocker是用来标识当前线程在等待的对象（以下称为阻塞对象），该对象主要用于问题排查和系统监控。

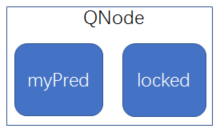
## CLH队列锁

CLH队列锁即Craig, Landin, and Hagersten (CLH) locks。

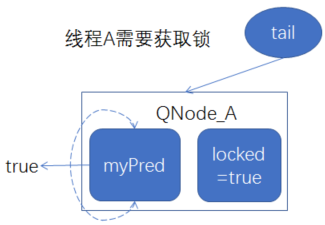
CLH队列锁也是一种基于链表的可扩展、高性能、公平的自旋锁，申请线程仅仅在本地变量上自旋，它不断轮询前驱的状态，假设发现前驱释放了锁就结束自旋。

当一个线程需要获取锁时：

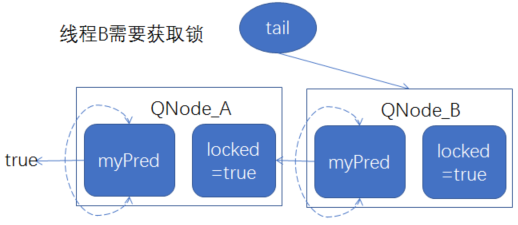
1. 创建一个的QNode，将其中的locked设置为true表示需要获取锁，myPred表示对其前驱结点的引用



1. 线程A对tail域调用getAndSet方法，使自己成为队列的尾部，同时获取一个指向其前驱结点的引用myPred



线程B需要获得锁，同样的流程再来一遍



3.线程就在前驱结点的locked字段上旋转，直到前驱结点释放锁(前驱节点的锁值 locked == false)

4.当一个线程需要释放锁时，将当前结点的locked域设置为false，同时回收前驱结点



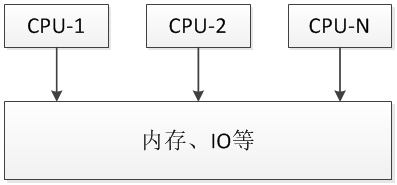
如上图所示，前驱结点释放锁，线程A的myPred所指向的前驱结点的locked字段变为false，线程A就可以获取到锁。

CLH队列锁的优点是空间复杂度低（如果有n个线程，L个锁，每个线程每次只获取一个锁，那么需要的存储空间是O（L+n），n个线程有n个myNode，L个锁有L个tail）。CLH队列锁常用在SMP体系结构下。

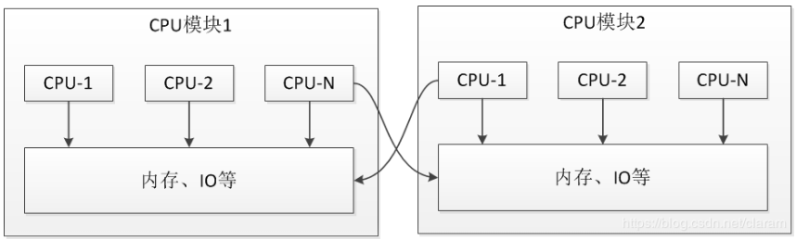
Java中的AQS是CLH队列锁的一种变体实现。

#### 扩展知识点

*SMP(Symmetric Multi-Processor)。即对称多处理器结构，指server中多个CPU对称工作，每一个CPU訪问内存地址所需时间同样。其主要特征是共享，包括对CPU，内存，I/O等进行共享。SMP的长处是可以保证内存一致性。缺点是这些共享的资源非常可能成为性能瓶颈。随着CPU数量的添加，每一个CPU都要訪问同样的内存资源，可能导致内存訪问冲突，可能会导致CPU资源的浪费。经常使用的PC机就属于这样的。*

**

*非一致存储访问，将CPU分为CPU模块，每个CPU模块由多个CPU组成，并且具有独立的本地内存、I/O槽口等，模块之间可以通过互联模块相互访问，访问本地内存（本CPU模块的内存）的速度将远远高于访问远地内存(其他CPU模块的内存)的速度，这也是非一致存储访问的由来。NUMA较好地解决SMP的扩展问题，当CPU数量增加时，因为访问远地内存的延时远远超过本地内存，系统性能无法线性增加。*

**

*CLH唯一的缺点是在NUMA系统结构下性能很差，但是在SMP系统结构下该法还是非常有效的。解决NUMA系统结构的思路是MCS队列锁。*

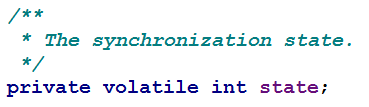
## AbstractQueuedSynchronizer

### 学习AQS的必要性

队列同步器AbstractQueuedSynchronizer（以下简称同步器或AQS），是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。并发包的大师（Doug Lea）期望它能够成为实现大部分同步需求的基础。

### AQS使用方式和其中的设计模式

AQS的主要使用方式是继承，子类通过继承AQS并实现它的抽象方法来管理同步状态，在AQS里由一个int型的state来代表这个状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法（getState()、setState(int newState)和compareAndSetState(int expect,int update)）来进行操作，因为它们能够保证状态的改变是安全的。



在实现上，子类推荐被定义为自定义同步组件的静态内部类，AQS自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型的同步组件（ReentrantLock、ReentrantReadWriteLock和CountDownLatch等）。

同步器是实现锁（也可以是任意同步组件）的关键，在锁的实现中聚合同步器。可以这样理解二者之间的关系：

锁是面向使用者的，它定义了使用者与锁交互的接口（比如可以允许两个线程并行访问），隐藏了实现细节；

同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。锁和同步器很好地隔离了使用者和实现者所需关注的领域。

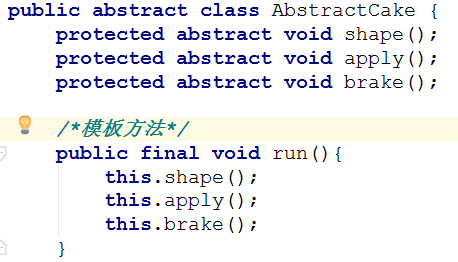
实现者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。

#### 模板方法模式

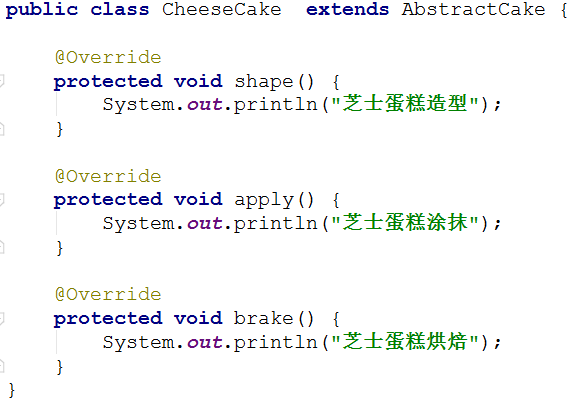
同步器的设计基于模板方法模式。模板方法模式的意图是，定义一个操作中的算法的骨架，而将一些步骤的实现延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。我们最常见的就是Spring框架里的各种Template。

##### 实际例子

我们开了个蛋糕店，蛋糕店不能只卖一种蛋糕呀，于是我们决定先卖奶油蛋糕，芝士蛋糕和慕斯蛋糕。三种蛋糕在制作方式上一样，都包括造型，烘焙和涂抹蛋糕上的东西。所以可以定义一个抽象蛋糕模型

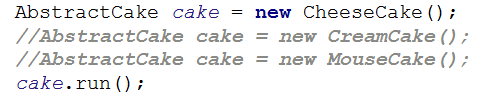


然后就可以批量生产三种蛋糕

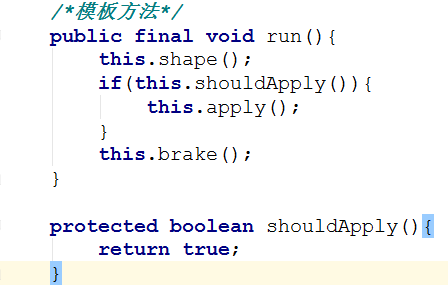




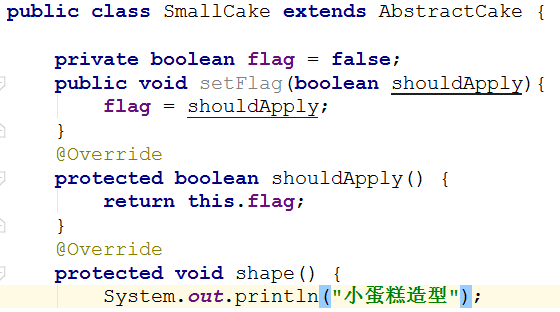




这样一来，不但可以批量生产三种蛋糕，而且如果日后有扩展，只需要继承抽象蛋糕方法就可以了，十分方便，我们天天生意做得越来越赚钱。突然有一天，我们发现市面有一种最简单的小蛋糕销量很好，这种蛋糕就是简单烘烤成型就可以卖，并不需要涂抹什么食材，由于制作简单销售量大，这个品种也很赚钱，于是我们也想要生产这种蛋糕。但是我们发现了一个问题，抽象蛋糕是定义了抽象的涂抹方法的，也就是说扩展的这种蛋糕是必须要实现涂抹方法，这就很鸡儿蛋疼了。怎么办？我们可以将原来的模板修改为带钩子的模板。



做小蛋糕的时候通过flag来控制是否涂抹，其余已有的蛋糕制作不需要任何修改可以照常进行。



### AQS中的方法

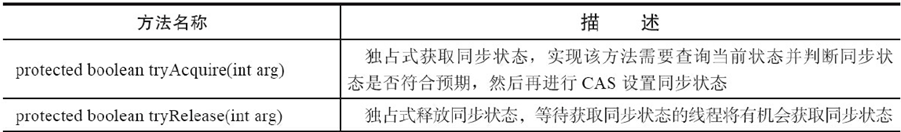
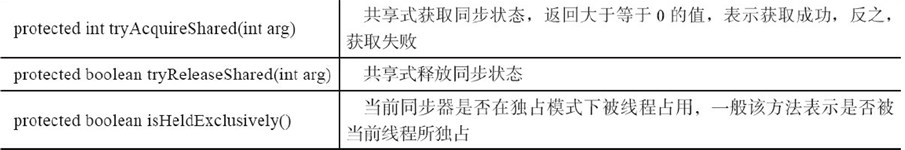
#### 模板方法

实现自定义同步组件时，将会调用同步器提供的模板方法，



这些模板方法同步器提供的模板方法基本上分为3类：独占式获取与释放同步状态、共享式获取与释放、同步状态和查询同步队列中的等待线程情况。

#### 可重写的方法

#### 访问或修改同步状态的方法

重写同步器指定的方法时，需要使用同步器提供的如下3个方法来访问或修改同步状态。

•getState()：获取当前同步状态。

•setState(int newState)：设置当前同步状态。

•compareAndSetState(int expect,int update)：使用CAS设置当前状态，该方法能够保证状态设置的原子性。

### 实现一个自己的独占锁

参见代码cn.enjoyedu.ch4.aqs.SelfLock

### 深入源码

#### AQS中的数据结构-节点和同步队列

##### 节点Node

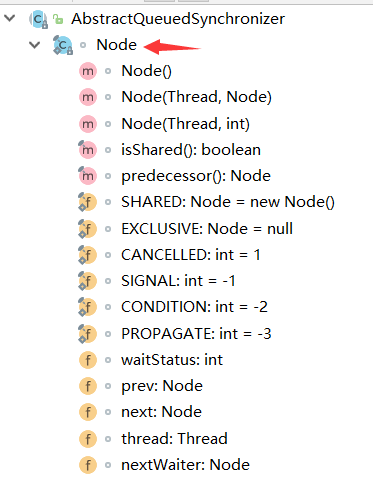
既然说Java中的AQS是CLH队列锁的一种变体实现，毫无疑问，作为队列来说，必然要有一个节点的数据结构来保存我们前面所说的各种域，比如前驱节点，节点的状态等，这个数据结构就是AQS中的内部类Node。作为这个数据结构应该关心些什么信息？

1、线程信息，肯定要知道我是哪个线程；

2、队列中线程状态，既然知道是哪一个线程，肯定还要知道线程当前处在什么状态，是已经取消了“获锁”请求，还是在“”等待中”，或者说“即将得到锁”

3、前驱和后继线程，因为是一个等待队列，那么也就需要知道当前线程前面的是哪个线程，当前线程后面的是哪个线程（因为当前线程释放锁以后，理当立马通知后继线程去获取锁）。

所以这个Node类是这么设计的：



其中包括了：

**线程的2种等待模式：**

SHARED：表示线程以共享的模式等待锁（如ReadLock）

EXCLUSIVE：表示线程以互斥的模式等待锁（如ReetrantLock），互斥就是一把锁只能由一个线程持有，不能同时存在多个线程使用同一个锁

**线程在队列中的状态枚举：**

CANCELLED：值为1，表示线程的获锁请求已经“取消”

SIGNAL：值为-1，表示该线程一切都准备好了,就等待锁空闲出来给我

CONDITION：值为-2，表示线程等待某一个条件（Condition）被满足

PROPAGATE：值为-3，当线程处在“SHARED”模式时，该字段才会被使用上

初始化Node对象时，默认为0

**成员变量：**

waitStatus：该int变量表示线程在队列中的状态，其值就是上述提到的CANCELLED、SIGNAL、CONDITION、PROPAGATE

prev：该变量类型为Node对象，表示该节点的前一个Node节点（前驱）

next：该变量类型为Node对象，表示该节点的后一个Node节点（后继）

thread：该变量类型为Thread对象，表示该节点的代表的线程

nextWaiter：该变量类型为Node对象，表示等待condition条件的Node节点

当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点（Node）并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。同步队列中的节点（Node）用来保存获取同步状态失败的线程引用、等待状态以及前驱和后继节点。

##### head和tail

AQS还拥有**首节点**（head）和**尾节点**（tail）两个引用，一个指向**队列头节点**，而另一个指向**队列尾节点**。

*注意：因为首节点head是不保存线程信息的节点，仅仅是因为数据结构设计上的需要，在数据结构上，这种做法往往叫做“空头节点链表”。对应的就有“非空头结点链表”*

### 节点在同步队列中的增加和移出

#### 节点加入到同步队列

当一个线程成功地获取了同步状态（或者锁），其他线程将无法获取到同步状态，也就是获取同步状态失败，AQS会将这个线程以及等待状态等信息构造成为一个节点（Node）并将其加入同步队列的尾部。而这个加入队列的过程必须要保证线程安全，因此同步器提供了一个基于CAS的设置尾节点的方法：compareAndSetTail(Node expect,Nodeupdate)，它需要传递当前线程“认为”的尾节点和当前节点，只有设置成功后，当前节点才正式与之前的尾节点建立关联。

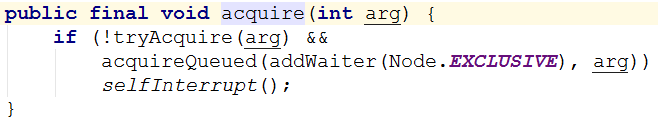
#### 首节点的变化

首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，而后继节点将会在获取同步状态成功时将自己设置为首节点。设置首节点是通过获取同步状态成功的线程来完成的，由于只有一个线程能够成功获取到同步状态，因此设置头节点的方法并不需要使用CAS来保证，它只需要将首节点设置成为原首节点的后继节点并断开原首节点的next引用即可。

#### 独占式同步状态获取与释放

##### 获取

通过调用同步器的acquire(int arg)方法可以获取同步状态，主要完成了同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的相关工作，其主要逻辑是：

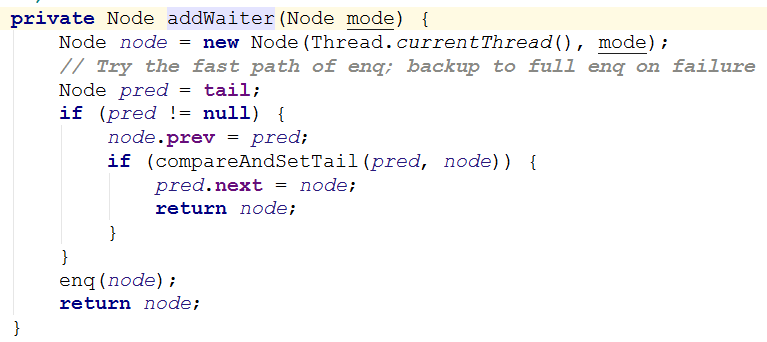


首先调用自定义同步器实现的tryAcquire(int arg)方法，该方法需要保证线程安全的获取同步状态。

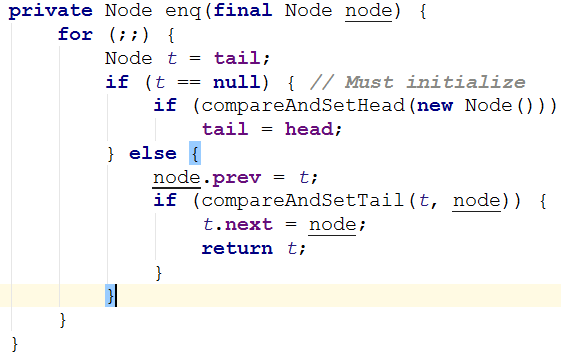
如果同步状态获取失败（tryAcquire返回false），则构造同步节点（独占式Node.EXCLUSIVE，同一时刻只能有一个线程成功获取同步状态）并通过addWaiter(Node node)方法将该节点加入到同步队列的尾部，

最后调用acquireQueued(Node node,int arg)方法，使得该节点以“死循环”的方式获取同步状态。如果获取不到则阻塞节点中的线程，而被阻塞线程的唤醒主要依靠前驱节点的出队或阻塞线程被中断来实现。

addWaiter(Node node)方法中



将当前线程包装成Node后，队列不为空的情况下，先尝试把当前节点加入队列并成为尾节点，如果不成功或者队列为空进入enq(final Node node)方法。



在enq(final Node node)方法中，同步器通过“死循环”来保证节点的正确添加，这个死循环中，做了两件事，第一件，如果队列为空，初始化队列，new出一个空节点，并让**首节点**（head）和**尾节点**（tail）两个引用都指向这个空节点；第二件事，把当前节点加入队列。

在“死循环”中只有通过CAS将节点设置成为尾节点之后，当前线程才能从该方法返回，否则，当前线程不断地尝试设置。

节点进入同步队列之后，观察acquireQueued(Node node,int arg)方法



其实就是一个自旋的过程，每个节点（或者说每个线程）都在自省地观察，当条件满足，获取到了同步状态，就可以从这个自旋过程中退出，否则依旧留在这个自旋过程中（并会阻塞节点的线程）。

在acquireQueued(final Node node,int arg)方法中，当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头节点才能够尝试获取同步状态，这是为什么？原因有两个。

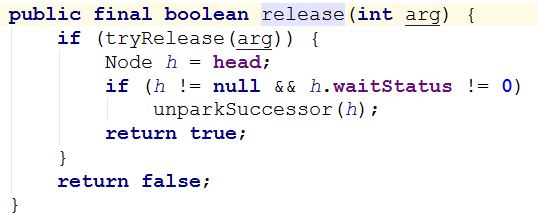
第一，头节点是成功获取到同步状态的节点，而头节点的线程释放了同步状态之后，将会唤醒其后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点。

第二，维护同步队列的FIFO原则。

当前线程获取到同步状态后，让**首节点**（head）这个引用指向自己所在节点。当同步状态获取成功后，当前线程就从acquire方法返回了。如果同步器实现的是锁，那就代表当前线程获得了锁。

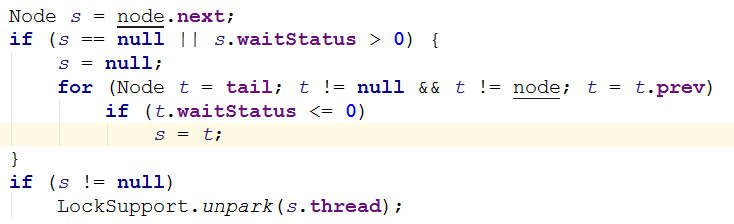
##### 释放

当前线程获取同步状态并执行了相应逻辑之后，就需要释放同步状态，使得后续节点能够继续获取同步状态。通过调用同步器的release(int arg)方法可以释放同步状态，该方法在释放了同步状态之后，会唤醒其后继节点（进而使后继节点重新尝试获取同步状态）。



该方法执行时，会唤醒**首节点**（head）所指向节点的后继节点线程，unparkSuccessor(Node node)方法使用LockSupport来唤醒处于等待状态的线程。

而在unparkSuccessor中，



这段代码的意思，一般情况下，被唤醒的是head指向节点的后继节点线程，如果这个后继节点处于被cancel状态，（我推测开发者的思路这样的：后继节点处于被cancel状态，意味着当锁竞争激烈时，队列的第一个节点等了很久（一直被还未加入队列的节点抢走锁），包括后续的节点cancel的几率都比较大，所以）先从尾开始遍历，找到最前面且没有被cancel的节点。

##### 总结

在获取同步状态时，同步器维护一个同步队列，获取状态失败的线程都会被加入到队列中并在队列中进行自旋；移出队列（或停止自旋）的条件是前驱节点为头节点且成功获取了同步状态。在释放同步状态时，同步器调用tryRelease(int arg)方法释放同步状态，然后唤醒head指向节点的后继节点。

#### 共享式同步状态获取与释放

共享式获取与独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。以读写为例，如果一个程序在进行读操作，那么这一时刻写操作均被阻塞，而读操作能够同时进行。写操作要求对资源的独占式访问，而读操作可以是共享式访问。

在acquireShared(int arg)方法中，同步器调用tryAcquireShared(int arg)方法尝试获取同步状态，tryAcquireShared(int arg)方法返回值为int类型，当返回值大于等于0时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。可以看到，在doAcquireShared(int arg)方法的自旋过程中，如果当前节点的前驱为头节点时，尝试获取同步状态，如果返回值大于等于0，表示该次获取同步状态成功并从自旋过程中退出。

该方法在释放同步状态之后，将会唤醒后续处于等待状态的节点。对于能够支持多个线程同时访问的并发组件（比如Semaphore），它和独占式主要区别在于tryReleaseShared(int arg)方法必须确保同步状态（或者资源数）线程安全释放，一般是通过循环和CAS来保证的，因为释放同步状态的操作会同时来自多个线程。

### 共享式的同步工具类

设计一个同步工具：该工具在同一时刻，只允许至多3个线程同时访问，超过3个线程的访问将被阻塞。

首先，确定访问模式。TrinityLock能够在同一时刻支持多个线程的访问，这显然是共享式访问，因此，需要使用同步器提供的acquireShared(int args)方法等和Shared相关的方法，这就要求TwinsLock必须重写tryAcquireShared(int args)方法和tryReleaseShared(int args)方法，这样才能保证同步器的共享式同步状态的获取与释放方法得以执行。

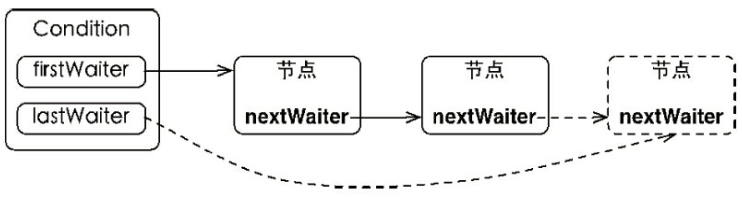
其次，定义资源数。TrinityLock在同一时刻允许至多三个线程的同时访问，表明同步资源数为3，这样可以设置初始状态status为3，当一个线程进行获取，status减1，该线程释放，则status加1，状态的合法范围为0、1和2,3，其中0表示当前已经有3个线程获取了同步资源，此时再有其他线程对同步状态进行获取，该线程只能被阻塞。在同步状态变更时，需要使用compareAndSet(int expect,int update)方法做原子性保障。

最后，组合自定义同步器。前面的章节提到，自定义同步组件通过组合自定义同步器来完成同步功能，一般情况下自定义同步器会被定义为自定义同步组件的内部类。

### 了解Condition的实现

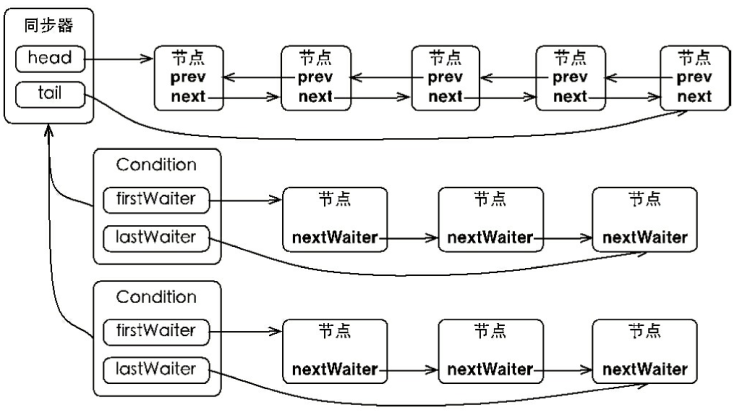
#### Condition的数据结构

等待队列是一个FIFO的队列，在队列中的每个节点都包含了一个线程引用，该线程就是在Condition对象上等待的线程，如果一个线程调用了Condition.await()方法，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态。事实上，节点的定义复用了同步器中节点的定义，也就是说，同步队列和等待队列中节点类型都是同步器的静态内部类。



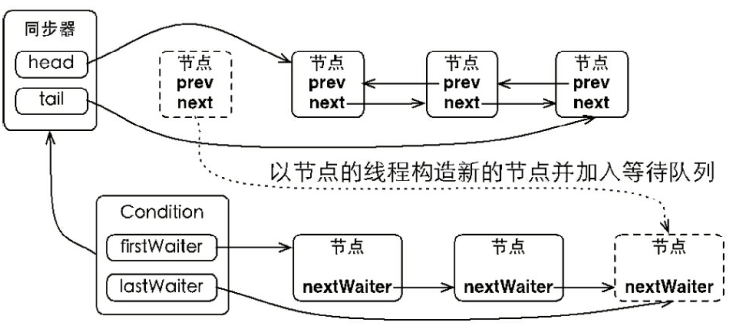
一个Condition包含一个等待队列，Condition拥有首节点（firstWaiter）和尾节点（lastWaiter）。当前线程调用Condition.await()方法，将会以当前线程构造节点，并将节点从尾部加入等待队列。Condition拥有首尾节点的引用，而新增节点只需要将原有的尾节点nextWaiter指向它，并且更新尾节点即可。上述节点引用更新的过程并没有使用CAS保证，原因在于调用await()方法的线程必定是获取了锁的线程，也就是说该过程是由锁来保证线程安全的。

Lock（更确切地说是同步器）拥有一个同步队列和多个等待队列。

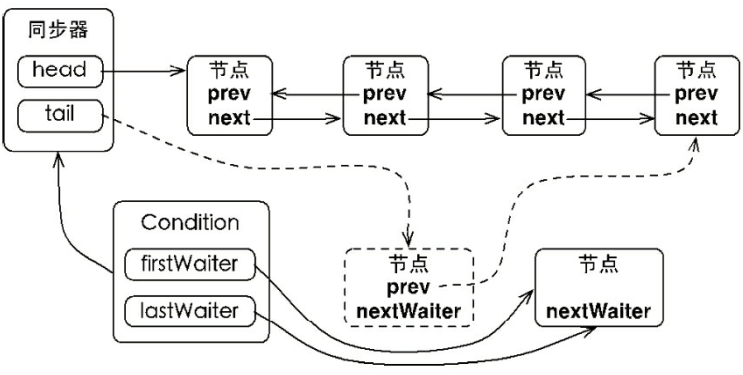


调用Condition的await()方法（或者以await开头的方法），会使当前线程进入等待队列并释放锁，同时线程状态变为等待状态。当从await()方法返回时，当前线程一定获取了Condition相关联的锁。

如果从队列（同步队列和等待队列）的角度看await()方法，当调用await()方法时，相当于同步队列的首节点（获取了锁的节点）移动到Condition的等待队列中。调用该方法的线程成功获取了锁的线程，也就是同步队列中的首节点，该方法会将当前线程构造成节点并加入等待队列中，然后释放同步状态，唤醒同步队列中的后继节点，然后当前线程会进入等待状态。当等待队列中的节点被唤醒，则唤醒节点的线程开始尝试获取同步状态。如果不是通过其他线程调用Condition.signal()方法唤醒，而是对等待线程进行中断，则会抛出InterruptedException。



如图所示，同步队列的首节点并不会直接加入等待队列，而是通过addConditionWaiter()方法把当前线程构造成一个新的节点并将其加入等待队列中。



调用Condition的signal()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移到同步队列中。

调用该方法的前置条件是当前线程必须获取了锁，可以看到signal()方法进行了isHeldExclusively()检查，也就是当前线程必须是获取了锁的线程。接着获取等待队列的首节点，将其移动到同步队列并使用LockSupport唤醒节点中的线程。

通过调用同步器的enq(Node node)方法，等待队列中的头节点线程安全地移动到同步队列。当节点移动到同步队列后，当前线程再使用LockSupport唤醒该节点的线程。

被唤醒后的线程，将从await()方法中的while循环中退出（isOnSyncQueue(Node node)方法返回true，节点已经在同步队列中），进而调用同步器的acquireQueued()方法加入到获取同步状态的竞争中。

成功获取同步状态（或者说锁）之后，被唤醒的线程将从先前调用的await()方法返回，此时该线程已经成功地获取了锁。

Condition的signalAll()方法，相当于对等待队列中的每个节点均执行一次signal()方法，效果就是将等待队列中所有节点全部移动到同步队列中，并唤醒每个节点的线程。

## 回头看Lock的实现

### ReentrantLock的实现

#### 锁的可重入

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决以下两个问题。

1）线程再次获取锁。锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。

2）锁的最终释放。线程重复n次获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已经成功释放。

nonfairTryAcquire方法增加了再次获取同步状态的处理逻辑：通过判断当前线程是否为获取锁的线程来决定获取操作是否成功，如果是获取锁的线程再次请求，则将同步状态值进行增加并返回true，表示获取同步状态成功。同步状态表示锁被一个线程重复获取的次数。

如果该锁被获取了n次，那么前(n-1)次tryRelease(int releases)方法必须返回false，而只有同步状态完全释放了，才能返回true。可以看到，该方法将同步状态是否为0作为最终释放的条件，当同步状态为0时，将占有线程设置为null，并返回true，表示释放成功。

#### 公平和非公平锁

ReentrantLock的构造函数中，默认的无参构造函数将会把Sync对象创建为NonfairSync对象，这是一个“非公平锁”；而另一个构造函数ReentrantLock(boolean fair)传入参数为true时将会把Sync对象创建为“公平锁”FairSync。

nonfairTryAcquire(int acquires)方法，对于非公平锁，只要CAS设置同步状态成功，则表示当前线程获取了锁，而公平锁则不同。tryAcquire方法，该方法与nonfairTryAcquire(int acquires)比较，唯一不同的位置为判断条件多了hasQueuedPredecessors()方法，即加入了同步队列中当前节点是否有前驱节点的判断，如果该方法返回true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

#### 改造我们的独占锁为可重入

参见代码cn.enjoyedu.ch4.aqs.ReenterSelfLock

### ReentrantReadWriteLock的实现

#### 读写状态的设计

读写锁同样依赖自定义同步器来实现同步功能，而读写状态就是其同步器的同步状态。

回想ReentrantLock中自定义同步器的实现，同步状态表示锁被一个线程重复获取的次数，而读写锁的自定义同步器需要在同步状态（一个整型变量）上维护多个读线程和一个写线程的状态，使得该状态的设计成为读写锁实现的关键。

如果在一个整型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写，读写锁是如何迅速确定读和写各自的状态呢？

答案是通过位运算。假设当前同步状态值为S，写状态等于S&0x0000FFFF（将高16位全部抹去），读状态等于S>>>16（无符号补0右移16位）。当写状态增加1时，等于S+1，当读状态增加1时，等于S+(1<<16)，也就是S+0x00010000。根据状态的划分能得出一个推论：S不等于0时，当写状态（S&0x0000FFFF）等于0时，则读状态（S>>>16）大于0，即读锁已被获取。

#### 写锁的获取与释放

写锁是一个支持重进入的排它锁。如果当前线程已经获取了写锁，则增加写状态。如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程，则当前线程进入等待状态。

该方法除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取，原因在于：读写锁要确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。

写锁的释放与ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续读写线程可见。

#### 读锁的获取与释放

读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有其他写线程访问（或者写状态为0）时，读锁总会被成功地获取，而所做的也只是（线程安全的）增加读状态。如果当前线程已经获取了读锁，则增加读状态。

如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态。读状态是所有线程获取读锁次数的总和，而每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，由线程自身维护。在tryAcquireShared(int unused)方法中，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全，依靠CAS保证）增加读状态，成功获取读锁。读锁的每次释放（线程安全的，可能有多个读线程同时释放读锁）均减少读状态。

#### 锁的升降级

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。

RentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。目的是保证数据可见性，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。

# 5、并发容器

## 预备知识

### hash

就是把任意长度的[输入](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5/5481954)（又叫做预映射， pre-image），通过散列算法，变换成固定长度的[输出](https://baike.baidu.com/item/%E8%BE%93%E5%87%BA)，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。常用HASH函数：直接取余法、乘法取整法、平方取中法。

处理冲突方法:

1．[开放寻址法](https://baike.baidu.com/item/%E5%BC%80%E6%94%BE%E5%AF%BB%E5%9D%80%E6%B3%95)； 2． 再[散列法](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E6%B3%95)： 3． 链地址法(拉链法)

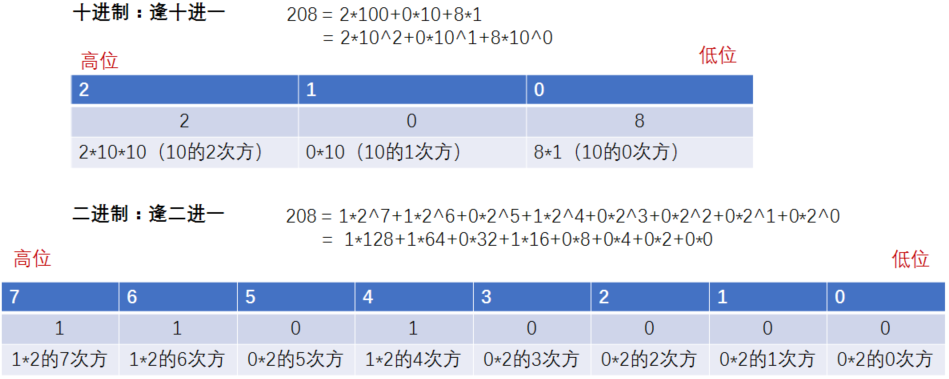
常用hash算法的介绍：**（1)**MD4**（2)**[MD5](https://baike.baidu.com/item/MD5)它对输入仍以512位分组，其输出是4个32位字的级联**（3)**[SHA-1](https://baike.baidu.com/item/SHA-1)及其他。

### 位运算

#### 二进制

从小学一年级开始，我们就开始学加减乘除，特别是学加法的时候，刚开始算超过十的时候，如果不熟练，我们会掰着手指头数，有时候恨不得把脚趾头也加上。正是因为我们一般人有十个手指头，所以，从我们老老。。。祖先开始就用十进制。那计算机怎么办？计算机又没十个手指头。自然界要找出有十种状态的东西很难，但是只有两种状态的东西或者现象还是很容易的，所以计算机中要用二进制。

逢十进一，意味着至少两点，1，每个位上的数字不能超过10，最大只能到9，第二，如果某个位上的数字因为运算超过了10，怎么办？往高位进位。那么二进制也是一样的，不能说你二进制比较二，规则就不一样了，基本法还是要遵守的。所以二进制里同样要遵循：1，每个位上的数字不能超过2，最大只能到1，第二，如果某个位上的数字因为运算超过了2，怎么办？往高位进位。



#### 常用位运算

位与 & (1&1=1 0&0=0 1&0=0)

位或 | (1|1=1 0|0=0 1|0=1)

位非 ~(~1=0 ~0=1)

位异或 ^ (1^1=0 1^0=1 0^0=0)

有符号右移>>(若正数,高位补0,负数,高位补1)

有符号左移<<

无符号右移>>>(不论正负,高位均补0)

有趣的取模性质：取模a % (2^n) 等价于 a & (2^n - 1)，所以在map里的数组个数一定是2的乘方数，计算key值在哪个元素中的时候，就用位运算来快速定位。

具体位运算的运行结果参见代码cn.enjoyedu.ch5.bitwise. IntToBinary

#### 位运算运用场景

Java中的类修饰符、成员变量修饰符、方法修饰符，比如Class类中

Java容器中的HashMap和ConcurrentHashMap的实现

权限控制或者商品属性

简单可逆加密，比如异或运算(1^1=0 ; 0^1=1 )

##### 实战：权限控制

参见代码cn.enjoyedu.ch5.bitwise. Permission

##### 使用位运算的优劣势：

节省很多代码量、效率高、属性变动影响小、不直观

## 为什么要使用ConcurrentHashMap

### 1.7中HashMap死循环分析

在多线程环境下，使用HashMap进行put操作会引起死循环，导致CPU利用率接近100%，HashMap在并发执行put操作时会引起死循环，是因为多线程会导致HashMap的Entry链表

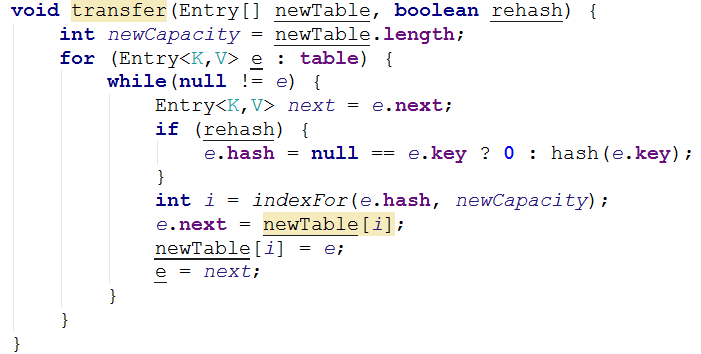
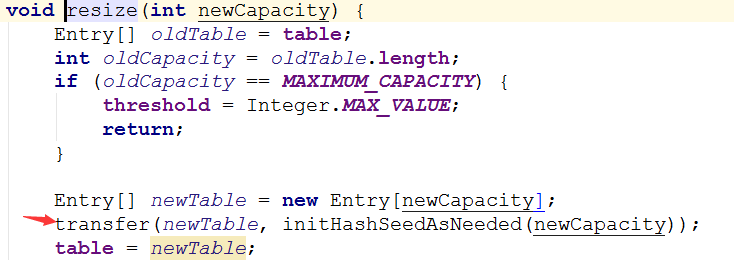
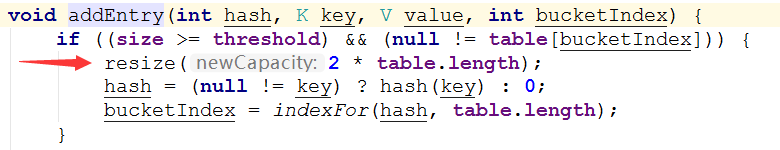
形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获取Entry。那么这个死循环是如何生成的呢？我们来仔细分析下。

#### HashMap扩容流程

##### 原理

引发死循环，是在HashMap的扩容操作中。

正常的扩容操作是这个流程。HashMap的扩容在put操作中会触发扩容，主要是三个方法：



综合来说，HashMap一次扩容的过程：

1、取当前table的2倍作为新table的大小

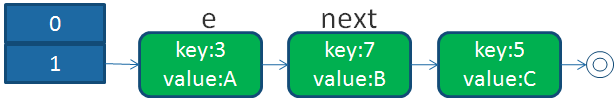
2、根据算出的新table的大小new出一个新的Entry数组来，名为newTable

3、轮询原table的每一个位置，将每个位置上连接的Entry，算出在新table上的位置，并以链表形式连接

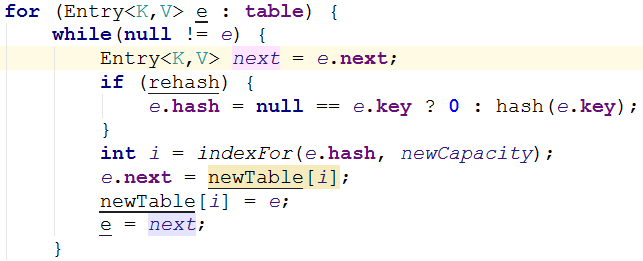
4、原table上的所有Entry全部轮询完毕之后，意味着原table上面的所有Entry已经移到了新的table上，HashMap中的table指向newTable

##### 实例

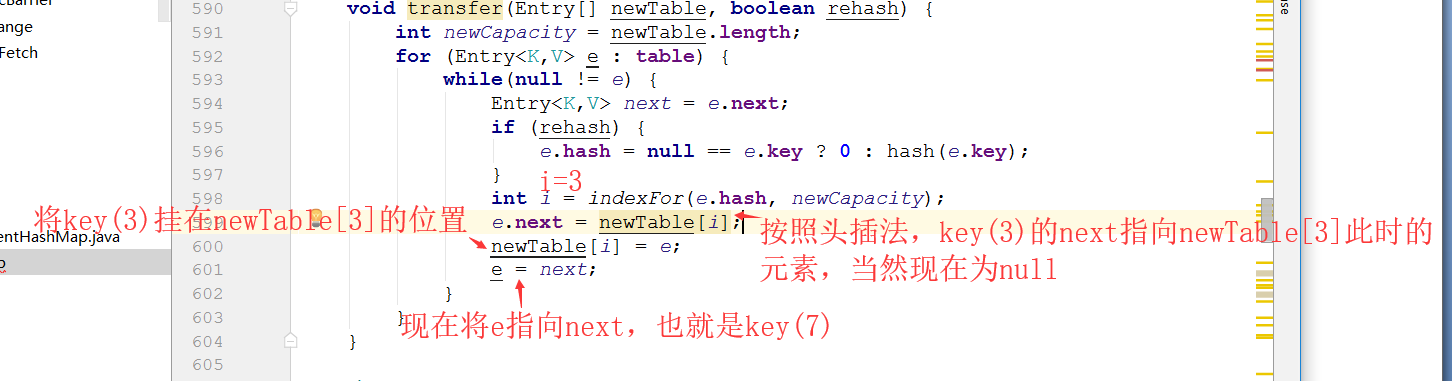
现在hashmap中有三个元素，Hash表的size=2, 所以key = 3, 7, 5，在mod 2以后都冲突在table[1]这里了。

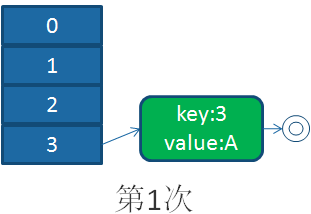


按照方法中的代码

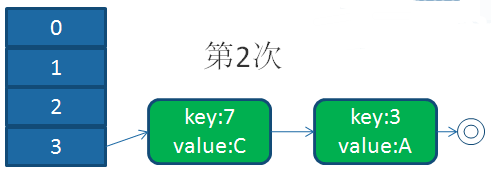


对table[1]中的链表来说，进入while循环，此时e=key(3)，那么next=key(7)，经过计算重新定位e=key(3)在新表中的位置，并把e=key(3)挂在newTable[3]的位置





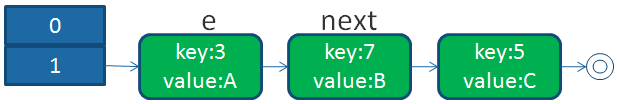
这样循环下去，将table[1]中的链表循环完成后，于是HashMap就完成了扩容

#### 并发下的扩容

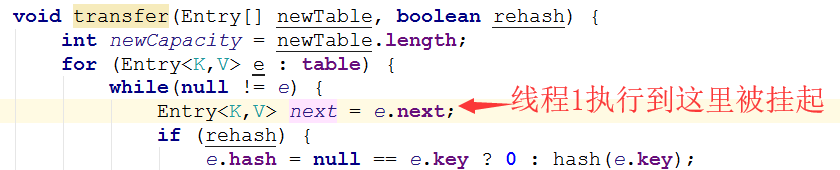
上面都是单线程下的扩容，当多线程进行扩容时，会是什么样子呢？

初始的HashM还是：

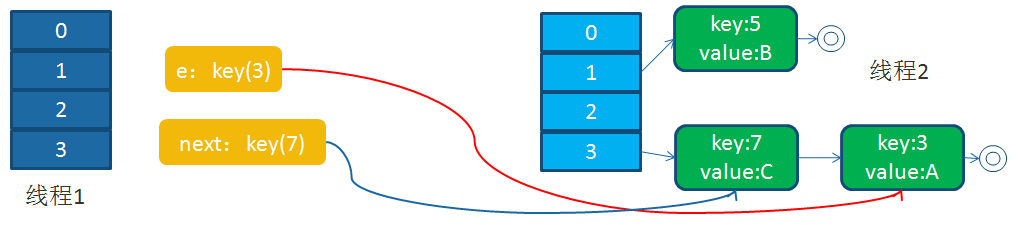


我们现在假设有两个线程并发操作，都进入了扩容操作，我们以颜色进行区分两个线程。

回顾我们的扩容代码，我们假设，线程1执行到Entry<K,V> next = e.next;时被操作系统调度挂起了，而线程2执行完成了扩容操作

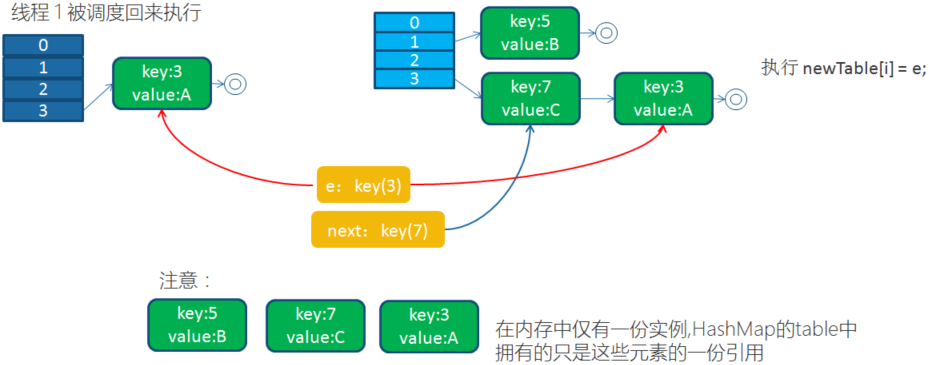


于是，在线程1,2看来，就应该是这个样子

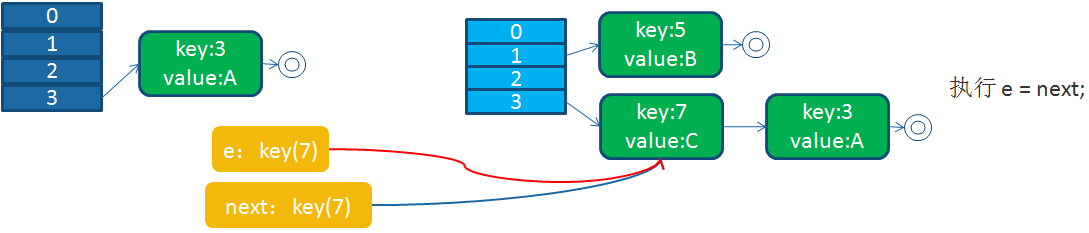


接下来，线程1被调度回来执行：

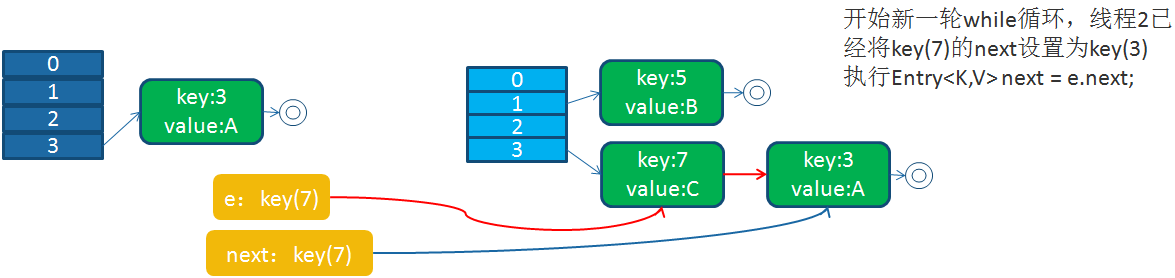
1）



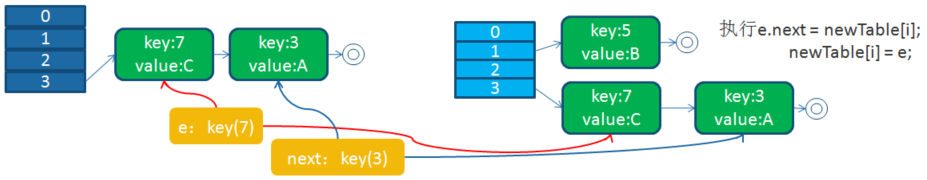
2）



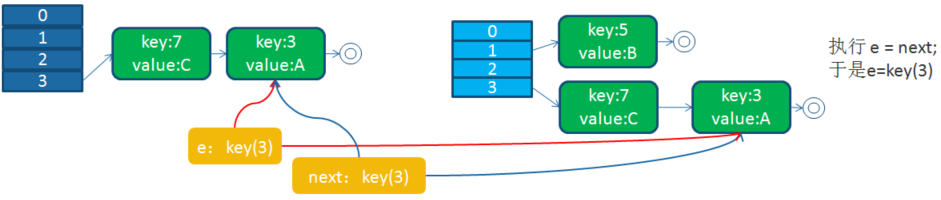
3）



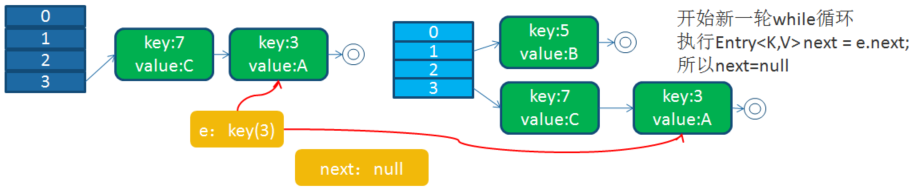
4）



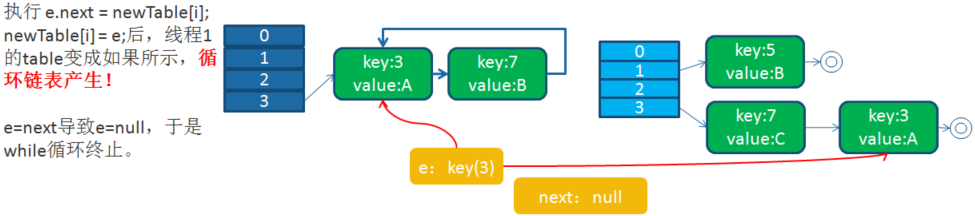
5）



6）



7）



循环列表产生后，一旦线程1调用get（11,15之类的元素）时，就会进入一个死循环的情况，将CPU的消耗到100%。

#### 总结

HashMap之所以在并发下的扩容造成死循环，是因为，多个线程并发进行时，因为一个线程先期完成了扩容，将原Map的链表重新散列到自己的表中，并且链表变成了倒序，后一个线程再扩容时，又进行自己的散列，再次将倒序链表变为正序链表。于是形成了一个环形链表，当get表中不存在的元素时，造成死循环。

## ConcurrentHashMap

### 使用

除了Map系列应该有的线程安全的get，put等方法外，ConcurrentHashMap还提供了一个在并发下比较有用的方法 putIfAbsent，如果传入key对应的value已经存在，就返回存在的value，不进行替换。如果不存在，就添加key和value，返回null。在代码层面它的作用类似于：

synchronized(map){

if (map.get(key) == null){

return map.put(key, value);

} else{

return map.get(key);

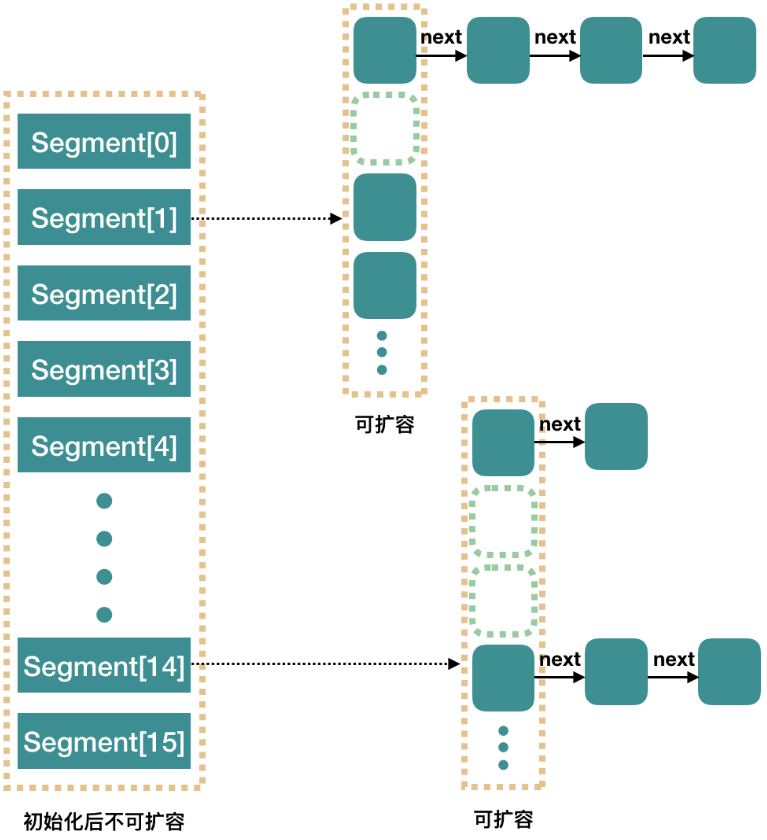
}

}

它让上述代码的整个操作是线程安全的。

### ConcurrentHashMap实现分析

#### 在1.7下的实现





ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组。Segment的结构和HashMap类似，是一种数组和链表结构。一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁。

##### 构造方法和初始化



ConcurrentHashMap初始化方法是通过initialCapacity、loadFactor和concurrencyLevel(参数concurrencyLevel是用户估计的并发级别，就是说你觉得最多有多少线程共同修改这个map，根据这个来确定Segment数组的大小concurrencyLevel默认是DEFAULT\_CONCURRENCY\_LEVEL = 16;)等几个参数来初始化segment数组、段偏移量segmentShift、段掩码segmentMask和每个segment里的HashEntry数组来实现的。

并发级别可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap默认的并发度为16，但用户也可以在构造函数中设置并发度。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。

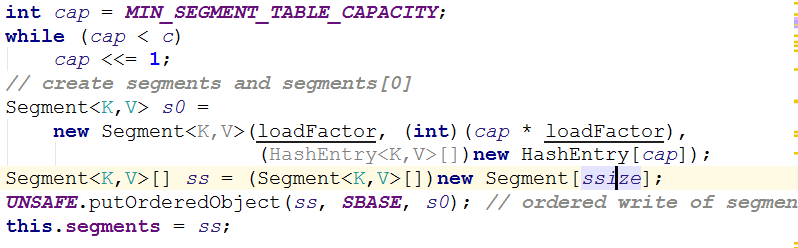
如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。（文档的说法是根据你并发的线程数量决定，太多会导性能降低）

segments数组的长度ssize是通过concurrencyLevel计算得出的。为了能通过按位与的散列算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14、15或16，ssize都会等于16，即容器里锁的个数也是16。

以下知识了解即可，无需深究

*初始化segmentShift和segmentMask*

*这两个全局变量需要在定位segment时的散列算法里使用，sshift等于ssize从1向左移位的次数，在默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。segmentShift用于定位参与散列运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的。segmentMask是散列运算的掩码，等于ssize减1，即15，掩码的二进制各个位的值都是1。因为ssize的最大长度是65536，所以segmentShift最大值是16，segmentMask最大值是65535，对应的二进制是16位，每个位都是1。*

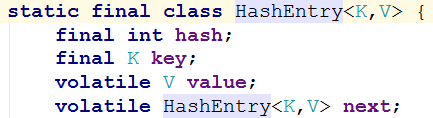


输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个segment的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个segment。上面代码中的变量cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以segment里HashEntry数组的长度不是1，就是2的N次方。

在默认情况下， ssize = 16，initialCapacity = 16，loadFactor = 0.75f，那么cap = 1，threshold = (int) cap \* loadFactor = 0。

既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须先通过散列算法定位到Segment。ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再散列。

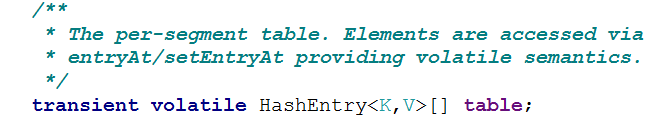
ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的以及volatile关键字。



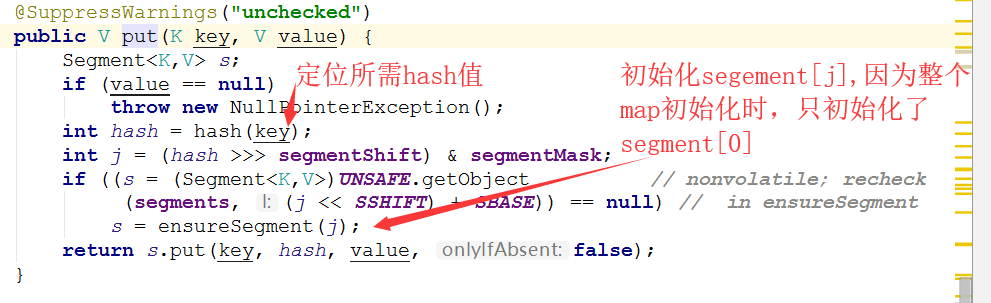
##### get操作



get操作先经过一次再散列，然后使用这个散列值通过散列运算定位到Segment(使用了散列值的高位部分)，再通过散列算法定位到table(使用了散列值的全部)。整个get过程，没有加锁，而是通过volatile保证get总是可以拿到最新值。



##### put操作

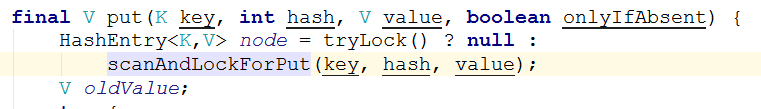


ConcurrentHashMap 初始化的时候会初始化第一个槽 segment[0]，对于其他槽，在插入第一个值的时候再进行初始化。

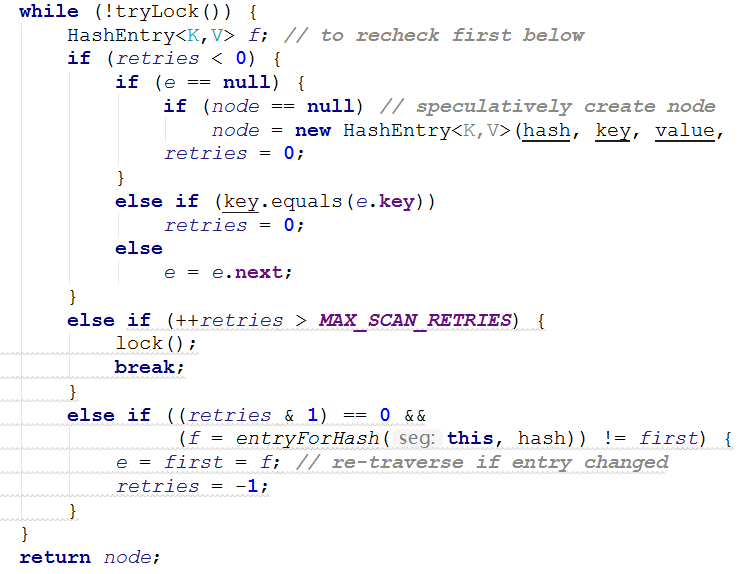
ensureSegment方法考虑了并发情况，多个线程同时进入初始化同一个槽 segment[k]，但只要有一个成功就可以了。

具体实现是

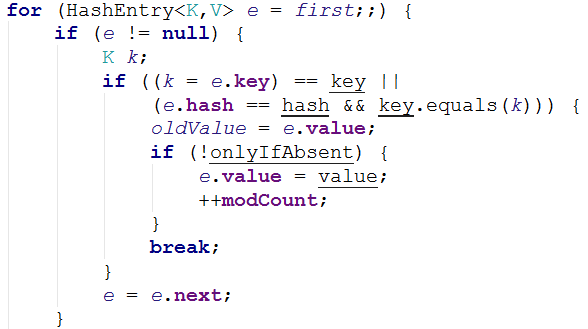




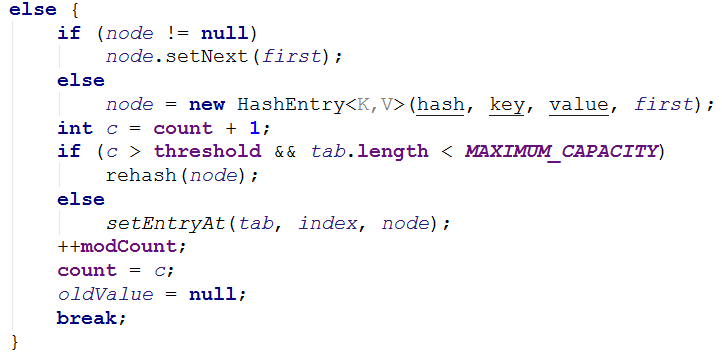
put方法会通过tryLock()方法尝试获得锁，获得了锁，node为null进入try语句块，没有获得锁，调用scanAndLockForPut方法自旋等待获得锁。



scanAndLockForPut方法里在尝试获得锁的过程中会对对应hashcode的链表进行遍历，如果遍历完毕仍然找不到与key相同的HashEntry节点，则为后续的put操作提前创建一个HashEntry。当tryLock一定次数后仍无法获得锁，则通过lock申请锁。



在获得锁之后，Segment对链表进行遍历，如果某个HashEntry节点具有相同的key，则更新该HashEntry的value值，



否则新建一个HashEntry节点，采用头插法，将它设置为链表的新head节点并将原头节点设为新head的下一个节点。新建过程中如果节点总数（含新建的HashEntry）超过threshold，则调用rehash()方法对Segment进行扩容，最后将新建HashEntry写入到数组中。

##### rehash操作

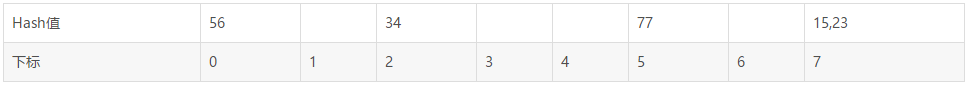
扩容是新创建了数组，然后进行迁移数据，最后再将 newTable 设置给属性 table。

为了避免让所有的节点都进行复制操作：由于扩容是基于2的幂指来操作，假设扩容前某HashEntry对应到Segment中数组的index为i，数组的容量为capacity，那么扩容后该HashEntry对应到新数组中的index只可能为i或者i+capacity，因此很多HashEntry节点在扩容前后index可以保持不变。

假设原来table长度为4，那么元素在table中的分布是这样的



扩容后table长度变为8，那么元素在table中的分布变成：



可以看见 hash值为34和56的下标保持不变，而15,23,77的下标都是在原来下标的基础上+4即可，可以快速定位和减少重排次数。

该方法没有考虑并发，因为执行该方法之前已经获取了锁。

##### remove操作

与put方法类似，都是在操作前需要拿到锁，以保证操作的线程安全性。

##### ConcurrentHashMap的弱一致性

然后对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。如果要求强一致性，那么必须使用Collections.synchronizedMap()方法。

##### size、containsValue

这些方法都是基于整个ConcurrentHashMap来进行操作的，他们的原理也基本类似：首先不加锁循环执行以下操作：循环所有的Segment，获得对应的值以及所有Segment的modcount之和。在 put、remove 和 clean 方法里操作元素前都会将变量 modCount 进行变动，如果连续两次所有Segment的modcount和相等，则过程中没有发生其他线程修改ConcurrentHashMap的情况，返回获得的值。

当循环次数超过预定义的值时，这时需要对所有的Segment依次进行加锁，获取返回值后再依次解锁。所以一般来说，应该避免在多线程环境下使用size和containsValue方法。

#### 在1.8下的实现

##### 改进

改进一：取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对缩小锁的粒度，进一步减少并发冲突的概率，并大量使用了采用了 CAS + synchronized 来保证并发安全性。

改进二：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。

使用 Node（1.7 为 Entry） 作为链表的数据结点，仍然包含 key，value，hash 和 next 四个属性。 红黑树的情况使用的是 TreeNode（**extends** Node）。

根据数组元素中，第一个结点数据类型是 Node 还是 TreeNode 可以判断该位置下是链表还是红黑树。

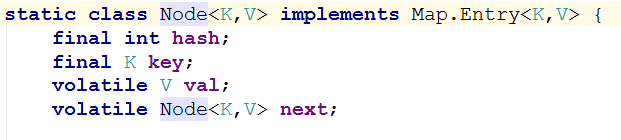
用于判断是否需要将链表转换为红黑树的阈值

用于判断是否需要将红黑树转换为链表的阈值

##### 核心数据结构和属性

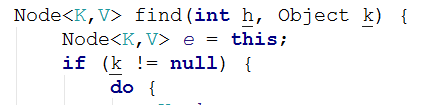
###### Node

Node是最核心的内部类，它包装了key-value键值对。



定义基本和1.7中的HashEntry相同。而这个map本身所持有的也是一个Node型的数组

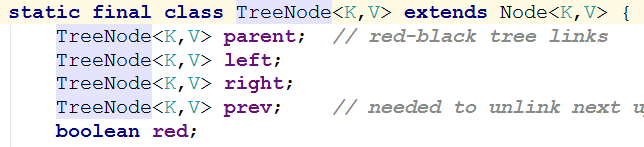
增加了一个find方法来用以辅助map.get()方法。其实就是遍历链表，子类中会覆盖这个方法。



在map中还定义了Segment这个类，不过只是为了向前兼容而已，不做过多考虑。

###### TreeNode

树节点类，另外一个核心的数据结构。当链表长度过长的时候，会转换为TreeNode。



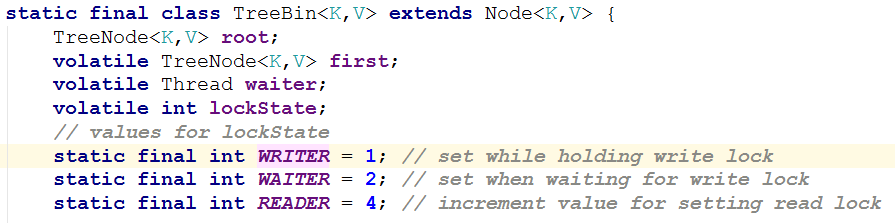
与1.8中HashMap不同点：

1、它并不是直接转换为红黑树，而是把这些结点放在TreeBin对象中，由TreeBin完成对红黑树的包装。

2、TreeNode在ConcurrentHashMap扩展自Node类，而并非HashMap中的扩展自LinkedHashMap.Entry<K,V>类，也就是说TreeNode带有next指针。

###### TreeBin

负责TreeNode节点。它代替了TreeNode的根节点，也就是说在实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，而不是TreeNode对象。另外这个类还带有了读写锁机制。



###### 特殊的ForwardingNode

一个特殊的 Node 结点，hash 值为 -1，其中存储 nextTable 的引用。有 table 发生扩容的时候，ForwardingNode 发挥作用，作为一个占位符放在 table 中表示当前结点为 null 或者已经被移动。

###### sizeCtl

用来控制 table 的初始化和扩容操作。

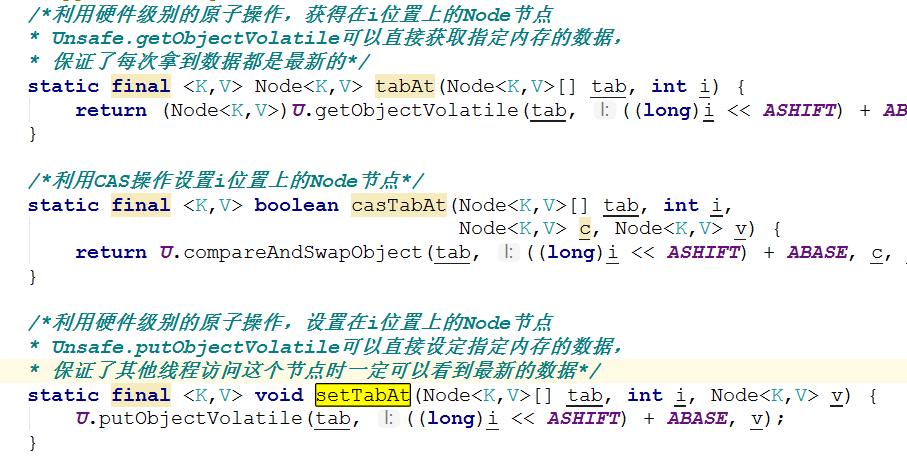
负数代表正在进行初始化或扩容操作

* -1代表正在初始化
* -N 表示有N-1个线程正在进行扩容操作

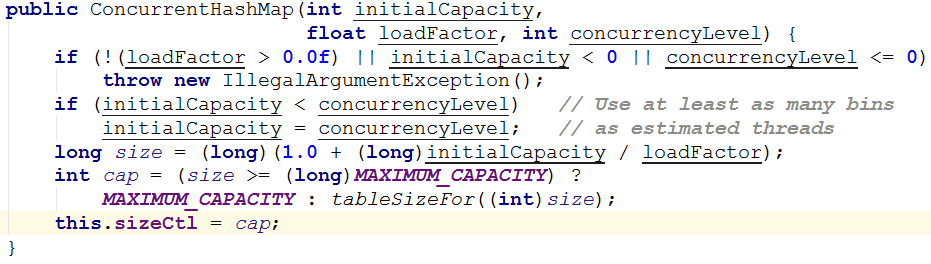
0为默认值，代表当时的table还没有被初始化

正数表示初始化大小或Map中的元素达到这个数量时，需要进行扩容了。

##### 核心方法



##### 构造方法

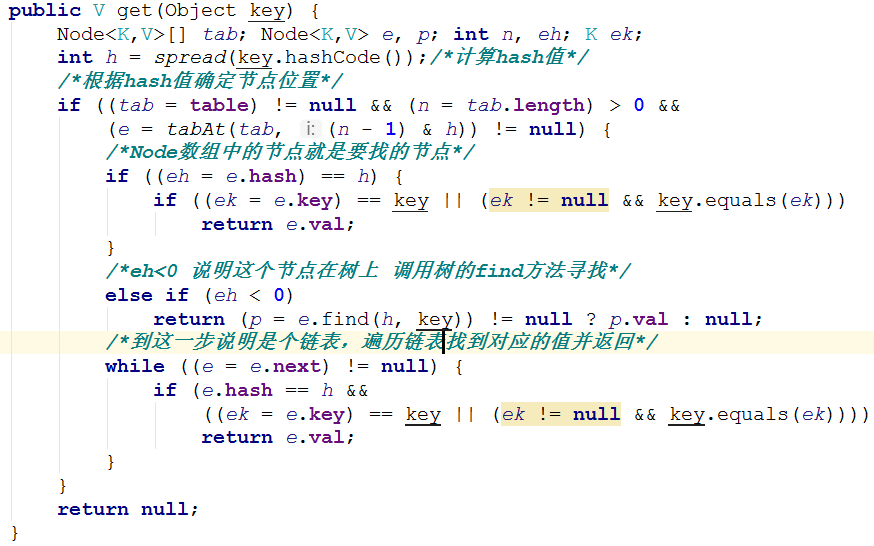


可以发现，在new出一个map的实例时，并不会创建其中的数组等等相关的部件，只是进行简单的属性设置而已，同样的，table的大小也被规定为必须是2的乘方数。

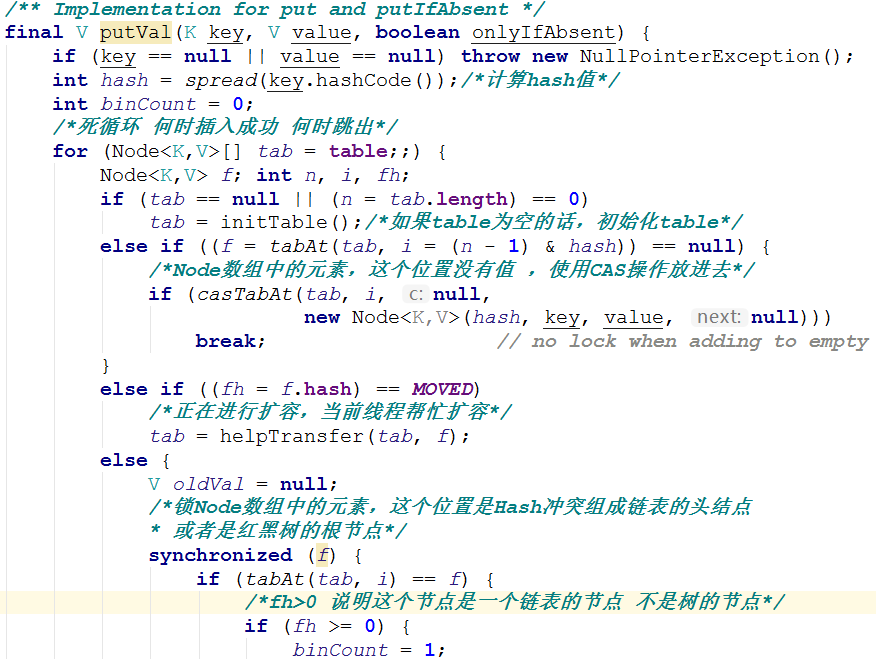
真正的初始化在放在了是在向ConcurrentHashMap中插入元素的时候发生的。如调用put、computeIfAbsent、compute、merge等方法的时候，调用时机是检查table==null。

##### get操作

get方法比较简单，给定一个key来确定value的时候，必须满足两个条件 key相同 hash值相同，对于节点可能在链表或树上的情况，需要分别去查找。



##### put操作







总结来说，put方法就是，沿用HashMap的put方法的思想，根据hash值计算这个新插入的点在table中的位置i，如果i位置是空的，直接放进去，否则进行判断，如果i位置是树节点，按照树的方式插入新的节点，否则把i插入到链表的末尾。

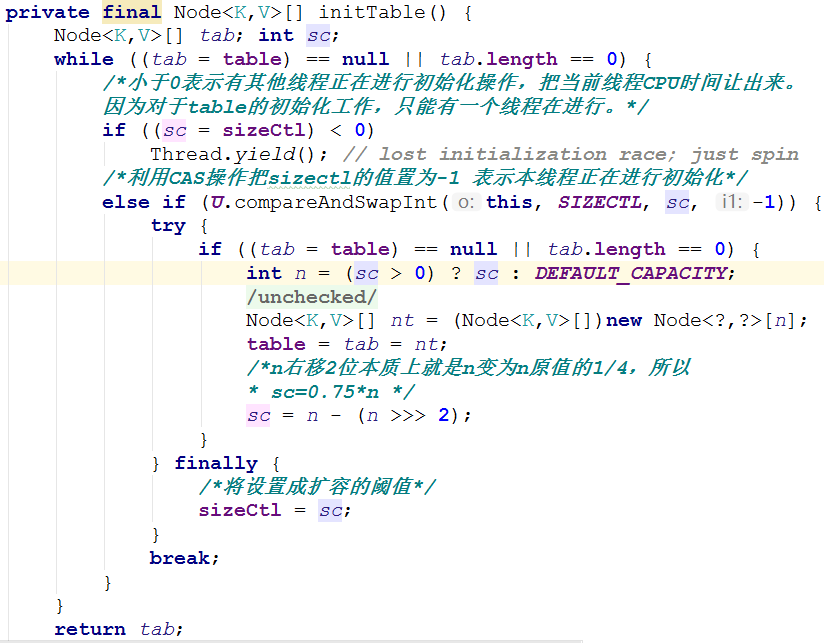
整体流程上，就是首先定义不允许key或value为null的情况放入  对于每一个放入的值，首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置。

如果这个位置是空的，那么直接放入，而且不需要加锁操作。

 如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果是链表节点,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到hash值与key值都与新加入节点是一致的情况，则只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点。如果加入这个节点以后链表长度大于8，就把这个链表转换成红黑树。如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值。

##### 初始化

前面说过，构造方法中并没有真正初始化，真正的初始化在放在了是在向ConcurrentHashMap中插入元素的时候发生的。具体实现的方法就是initTable



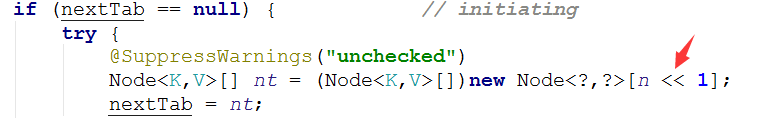
##### transfer

当ConcurrentHashMap容量不足的时候，需要对table进行扩容。这个方法的基本思想跟HashMap是很像的，但是由于它是支持并发扩容的，所以要复杂的多。我们不深入源码去讲述，只讲述其大概原理。

为何要并发扩容？因为在扩容的时候，总是会涉及到从一个“数组”到另一个“数组”拷贝的操作，如果这个操作能够并发进行，就能利用并发处理去减少扩容带来的时间影响。

整个扩容操作分为两个部分：

第一部分是构建一个nextTable,它的容量是原来的2倍。



第二个部分就是将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

整个扩容流程就是遍历和复制：

为null或者已经处理过的节点，会被设置为forwardNode节点，当线程准备扩容时，发现节点是forwardNode节点，跳过这个节点，继续寻找未处理的节点，找到了，对节点上锁，

如果这个位置是Node节点（fh>=0），说明它是一个链表，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上

如果这个位置是TreeBin节点（fh<0），也做一个反序处理，并且判断是否需要红黑树转链表，把处理的结果分别放在nextTable的i和i+n的位置上

遍历过所有的节点以后就完成了复制工作，这时让nextTable作为新的table，并且更新sizeCtl为新容量的0.75倍 ，完成扩容。

并发扩容其实就是将数据迁移任务拆分成多个小迁移任务，在实现上使用了一个变量stride作为步长控制，每个线程每次负责迁移其中的一部分。

##### remove

移除方法的基本流程和put方法很类似，只不过操作由插入数据变为移除数据而已，而且如果存在红黑树的情况下，会检查是否需要将红黑树转为链表的步骤。不再重复讲述。

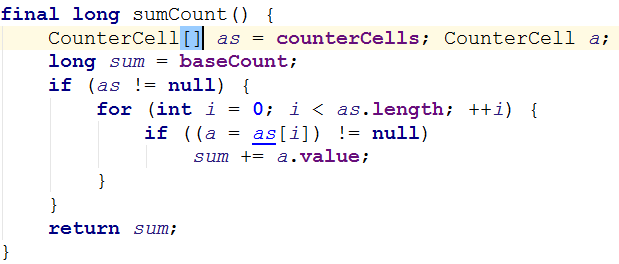
##### treeifyBin

用于将过长的链表转换为TreeBin对象。但是他并不是直接转换，而是进行一次容量判断，如果容量没有达到转换的要求，直接进行扩容操作并返回；如果满足条件才将链表的结构转换为TreeBin ，这与HashMap不同的是，它并没有把TreeNode直接放入红黑树，而是利用了TreeBin这个小容器来封装所有的TreeNode。

##### size

在JDK1.8版本中，对于size的计算，在扩容和addCount()方法就已经有处理了，可以注意一下Put函数，里面就有addCount()函数，早就计算好的，然后你size的时候直接给你。JDK1.7是在调用size()方法才去计算，其实在并发集合中去计算size是没有多大的意义的，因为size是实时在变的。

在具体实现上，计算大小的核心方法都是 sumCount()



可以看见，统计数量时使用了 baseCount、和CounterCell 类型的变量counterCells 。其实baseCount就是记录容器数量的，而counterCells则是记录CAS更新baseCounter值时，由于高并发而导致失败的值。这两个变量的变化在addCount() 方法中有体现，大致的流程就是：

1、对 baseCount 做 CAS 自增操作。

2、如果并发导致 baseCount CAS 失败了，则使用 counterCells。

3、如果counterCells CAS 失败了，在 fullAddCount 方法中，会继续死循环操作，直到成功。

### HashTable

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

### 并发下的Map常见面试题汇总

**Q：HashMap 和 HashTable 有什么区别？**

A：①、HashMap 是线程不安全的，HashTable 是线程安全的；

②、由于线程安全，所以 HashTable 的效率比不上 HashMap；

③、HashMap最多只允许一条记录的键为null，允许多条记录的值为null，而 HashTable 不允许；

④、HashMap 默认初始化数组的大小为16，HashTable 为 11，前者扩容时，扩大两倍，后者扩大两倍+1；

⑤、HashMap 需要重新计算 hash 值，而 HashTable 直接使用对象的 hashCode

**Q：Java 中的另一个线程安全的与 HashMap 极其类似的类是什么？同样是线程安全，它与 HashTable 在线程同步上有什么不同？**

A：ConcurrentHashMap 类（是 Java并发包 java.util.concurrent 中提供的一个线程安全且高效的 HashMap 实现）。

HashTable 是使用 synchronize 关键字加锁的原理（就是对对象加锁）；

而针对 ConcurrentHashMap，在 JDK 1.7 中采用分段锁的方式；JDK 1.8 中直接采用了CAS（无锁算法）+ synchronized，也采用分段锁的方式并大大缩小了锁的粒度。

**HashMap & ConcurrentHashMap 的区别？**

A：除了加锁，原理上无太大区别。

另外，HashMap 的键值对允许有null，但是ConCurrentHashMap 都不允许。

在数据结构上，红黑树相关的节点类

**Q：为什么 ConcurrentHashMap 比 HashTable 效率要高？**

A：HashTable 使用一把锁（锁住整个链表结构）处理并发问题，多个线程竞争一把锁，容易阻塞；

ConcurrentHashMap

JDK 1.7 中使用分段锁（ReentrantLock + Segment + HashEntry），相当于把一个 HashMap 分成多个段，每段分配一把锁，这样支持多线程访问。锁粒度：基于 Segment，包含多个 HashEntry。

JDK 1.8 中使用 CAS + synchronized + Node + 红黑树。锁粒度：Node（首结点）（实现 Map.Entry<K,V>）。锁粒度降低了。

**Q：针对 ConcurrentHashMap 锁机制具体分析（JDK 1.7 VS JDK 1.8）？**

JDK 1.7 中，采用分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构，包括两个核心静态内部类 Segment 和 HashEntry。

①、Segment 继承 ReentrantLock（重入锁） 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；

②、HashEntry 用来封装映射表的键-值对；

③、每个桶是由若干个 HashEntry 对象链接起来的链表。

JDK 1.8 中，采用Node + CAS + Synchronized来保证并发安全。取消类 Segment，直接用 table 数组存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY\_THRESHOLD 时，链表转换为红黑树，提升性能。底层变更为数组 + 链表 + 红黑树。

**Q：ConcurrentHashMap 在 JDK 1.8 中，为什么要使用内置锁 synchronized 来代替重入锁 ReentrantLock？**

A：

1、JVM 开发团队在1.8中对 synchronized做了大量性能上的优化，而且基于 JVM 的 synchronized 优化空间更大，更加自然。

2、在大量的数据操作下，对于 JVM 的内存压力，基于 API 的 ReentrantLock 会开销更多的内存。

**Q：ConcurrentHashMap 简单介绍？**

A：

①、重要的常量：

private transient volatile int sizeCtl;

当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；

当为 0 时，表示 table 还没有初始化；

当为其他正数时，表示初始化或者下一次进行扩容的大小。

②、数据结构：

Node 是存储结构的基本单元，继承 HashMap 中的 Entry，用于存储数据；

TreeNode 继承 Node，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于红黑树中存储数据；

TreeBin 是封装 TreeNode 的容器，提供转换红黑树的一些条件和锁的控制。

③、存储对象时（put() 方法）：

1.如果没有初始化，就调用 initTable() 方法来进行初始化；

2.如果没有 hash 冲突就直接 CAS 无锁插入；

3.如果需要扩容，就先进行扩容；

4.如果存在 hash 冲突，就加锁来保证线程安全，两种情况：一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入；

5.如果该链表的数量大于阀值 8，就要先转换成红黑树的结构，break 再一次进入循环

6.如果添加成功就调用 addCount() 方法统计 size，并且检查是否需要扩容。

④、扩容方法 transfer()：默认容量为 16，扩容时，容量变为原来的两倍。

helpTransfer()：调用多个工作线程一起帮助进行扩容，这样的效率就会更高。

⑤、获取对象时（get()方法）：

1.计算 hash 值，定位到该 table 索引位置，如果是首结点符合就返回；

2.如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find()方法，查找该结点，匹配就返回；

3.以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

**Q：ConcurrentHashMap 的并发度是什么？**

A：1.7中程序运行时能够同时更新 ConccurentHashMap 且不产生锁竞争的最大线程数。默认为 16，且可以在构造函数中设置。当用户设置并发度时，ConcurrentHashMap 会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。

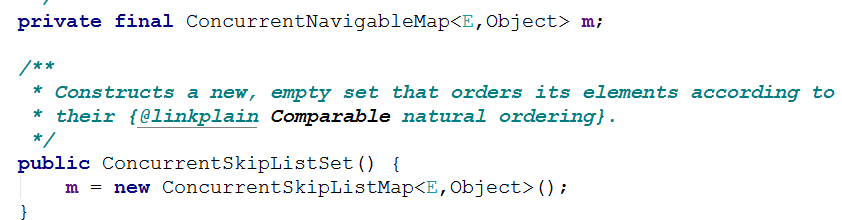
1.8中并发度则无太大的实际意义了，主要用处就是当设置的初始容量小于并发度，将初始容量提升至并发度大小。

## ConcurrentSkipList系列

ConcurrentSkipListMap 有序Map

ConcurrentSkipListSet 有序Set

TreeMap和TreeSet使用红黑树按照key的顺序（自然顺序、自定义顺序）来使得键值对有序存储*，*但是只能在单线程下安全使用；多线程下想要使键值对按照key的顺序来存储，则需要使用ConcurrentSkipListMap和ConcurrentSkipListSet，分别用以代替TreeMap和TreeSet，存入的数据按key排序。在实现上，ConcurrentSkipListSet 本质上就是ConcurrentSkipListMap。



### 了解什么是SkipList？

#### 二分查找和AVL树查找

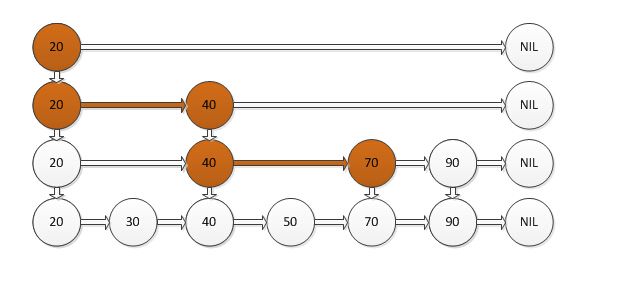
二分查找要求元素可以随机访问，所以决定了需要把元素存储在连续内存。这样查找确实很快，但是插入和删除元素的时候，为了保证元素的有序性，就需要大量的移动元素了。

如果需要的是一个能够进行二分查找，又能快速添加和删除元素的数据结构，首先就是二叉查找树，二叉查找树在最坏情况下可能变成一个链表。

于是，就出现了平衡二叉树，根据平衡算法的不同有AVL树，B-Tree，B+Tree，红黑树等，但是AVL树实现起来比较复杂，平衡操作较难理解，这时候就可以用SkipList跳跃表结构。

#### 什么是跳表

传统意义的单链表是一个线性结构，向有序的链表中插入一个节点需要O(n)的时间，查找操作需要O(n)的时间。



如果我们使用上图所示的跳跃表，就可以减少查找所需时间为O(n/2)，因为我们可以先通过每个节点的最上面的指针先进行查找，这样子就能跳过一半的节点。

比如我们想查找50，首先和20比较，大于20之后，在和40进行比较，然后在和70进行比较，发现70大于50，说明查找的点在40和50之间，从这个过程中，我们可以看出，查找的时候跳过了30。

跳跃表其实也是一种通过“空间来换取时间”的一个算法，令链表的每个结点不仅记录next结点位置，还可以按照level层级分别记录后继第level个结点。此法使用的就是“**先大步查找确定范围，再逐渐缩小迫近**”的思想进行的查找。跳跃表在算法效率上很接近红黑树。

跳跃表又被称为概率，或者说是随机化的数据结构，目前开源软件 Redis 和 lucence都有用到它。

都是线程安全的Map实现，ConcurrentHashMap的性能和存储空间要优于ConcurrentSkipListMap，但是ConcurrentSkipListMap有一个功能： 它会按照键的顺序进行排序。

## ConcurrentLinkedQueue

**无界非阻塞队列，它是一个基于链表的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先加入的，尾是最近加入的。插入元素是追加到尾上。提取一个元素是从头提取。**

大家可以看成是LinkedList的并发版本，常用方法：

concurrentLinkedQueue.add("c");

 concurrentLinkedQueue.offer("d"); // 将指定元素插入到此队列的尾部。

concurrentLinkedQueue.peek(); // 检索并不移除此队列的头，如果此队列为空，则返回 null。

concurrentLinkedQueue.poll(); // 检索并移除此队列的头，如果此队列为空，则返回 null。

## 写时复制容器

### 什么是写时复制容器

CopyOnWriteArrayList和CopyOnWriteArraySet

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。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并发容器用于对于绝大部分访问都是读，且**只是偶尔写**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。

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。

### 写时复制容器的问题

#### 性能问题

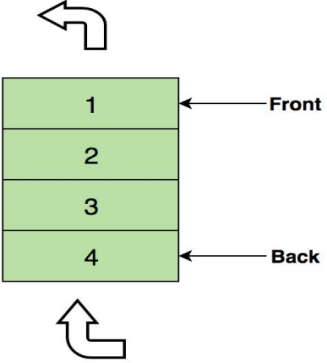
每次修改都创建一个新数组，然后复制所有内容，如果数组比较大，修改操作又比较频繁，可以想象，性能是很低的，而且内存开销会很大。

#### 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，不要使用CopyOnWrite容器。

## 阻塞队列BlockingQueue

### 队列



队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。

在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

### 什么是阻塞队列

1）支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2）支持阻塞的移除方法：意思是在队列为空时，获取元素的线程会等待队列变为非空。

在并发编程中使用**生产者和消费者模式**能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序整体处理数据的速度。

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。

为了解决这种生产消费能力不均衡的问题，便有了生产者和消费者模式。生产者和消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通信，而是通过阻塞队列来进行通信，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

阻塞队列常用于生产者和消费者的场景，生产者是向队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器。



·抛出异常：当队列满时，如果再往队列里插入元素，会抛出IllegalStateException（"Queuefull"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。

·返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。

·一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。

·超时退出：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

### 常用阻塞队列

·ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

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

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

·DelayQueue：一个使用优先级队列实现的无界阻塞队列。

·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。

以上的阻塞队列都实现了BlockingQueue接口，也都是线程安全的。

#### 有界无界？

有限队列就是长度有限，满了以后生产者会阻塞，无界队列就是里面能放无数的东西而不会因为队列长度限制被阻塞，当然空间限制来源于系统资源的限制，如果处理不及时，导致队列越来越大越来越大，超出一定的限制致使内存超限，操作系统或者JVM帮你解决烦恼，直接把你 OOM kill 省事了。

无界也会阻塞，为何？因为阻塞不仅仅体现在生产者放入元素时会阻塞，消费者拿取元素时，如果没有元素，同样也会阻塞。

#### ArrayBlockingQueue

是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证线程公平的访问队列，所谓公平访问队列是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列。非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可以争夺访问队列的资格，有可能先阻塞的线程最后才访问队列。初始化时有参数可以设置

#### LinkedBlockingQueue

是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序。

#### Array实现和Linked实现的区别

1. 队列中锁的实现不同

ArrayBlockingQueue实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁；

LinkedBlockingQueue实现的队列中的锁是分离的，即生产用的是putLock，消费是takeLock

2. 在生产或消费时操作不同

ArrayBlockingQueue实现的队列中在生产和消费的时候，是直接将枚举对象插入或移除的；

LinkedBlockingQueue实现的队列中在生产和消费的时候，需要把枚举对象转换为Node<E>进行插入或移除，会影响性能

3. 队列大小初始化方式不同

ArrayBlockingQueue实现的队列中必须指定队列的大小；

LinkedBlockingQueue实现的队列中可以不指定队列的大小，但是默认是Integer.MAX\_VALUE

#### PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

#### DelayQueue

是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

DelayQueue非常有用，可以将DelayQueue运用在以下应用场景。

**缓存系统的设计：**可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。**还有订单到期，限时支付等等**

#### SynchronousQueue

是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

#### LinkedTransferQueue

多了tryTransfer和transfer方法，

（1）transfer方法

如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。

（2）tryTransfer方法

tryTransfer方法是用来试探生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回，而transfer方法是必须等到消费者消费了才返回。

#### LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队列的两端插入和移出元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。

多了addFirst、addLast、offerFirst、offerLast、peekFirst和peekLast等方法，以First单词结尾的方法，表示插入、获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入、获取或移除双端队列的最后一个元素。另外，插入方法add等同于addLast，移除方法remove等效于removeFirst。但是take方法却等同于takeFirst，不知道是不是JDK的bug，使用时还是用带有First和Last后缀的方法更清楚。在初始化LinkedBlockingDeque时可以设置容量防止其过度膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中。

### 了解阻塞队列的实现原理

使用了等待通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。通过查看JDK源码发现ArrayBlockingQueue使用了Condition来实现。其余队列的实现，大家可以自行查看，队列的实现的代码总体来说，并不复杂。

# 6、线程池

## 为什么要用线程池？

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来3个好处。

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。假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。 如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

假设一个服务器一天要处理50000个请求，并且每个请求需要一个单独的线程完成。在线程池中，线程数一般是固定的，所以产生线程总数不会超过线程池中线程的数目，而如果服务器不利用线程池来处理这些请求则线程总数为50000。一般线程池大小是远小于50000。所以利用线程池的服务器程序不会为了创建50000而在处理请求时浪费时间，从而提高效率。

## ThreadPoolExecutor 的类关系

Executor是一个接口，它是Executor框架的基础，它将任务的提交与任务的执行分离开来。

ExecutorService接口继承了Executor，在其上做了一些shutdown()、submit()的扩展，可以说是真正的线程池接口；

AbstractExecutorService抽象类实现了ExecutorService接口中的大部分方法；

ThreadPoolExecutor是线程池的核心实现类，用来执行被提交的任务。

ScheduledExecutorService接口继承了ExecutorService接口，提供了带"周期执行"功能ExecutorService；

ScheduledThreadPoolExecutor是一个实现类，可以在给定的延迟后运行命令，或者定期执行命令。ScheduledThreadPoolExecutor比Timer更灵活，功能更强大。

## 线程池的创建各个参数含义

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)

### corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

### maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

### keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用

TimeUnit

keepAliveTime的时间单位

workQueue

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能

### workQueue

用于保存等待执行的任务的阻塞队列，一般来说，我们应该尽量使用有界队列，因为使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize。

2）由于1，使用无界队列时maximumPoolSize将是一个无效参数。

3）由于1和2，使用无界队列时keepAliveTime将是一个无效参数。

4）更重要的，使用无界queue可能会耗尽系统资源，有界队列则有助于防止资源耗尽，同时即使使用有界队列，也要尽量控制队列的大小在一个合适的范围。

所以我们一般会使用，ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue。

### threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名，当然还可以更加自由的对线程做更多的设置，比如设置所有的线程为守护线程。

参见代码cn.enjoyedu.ch6. ThreadPoolAdv

Executors静态工厂里默认的threadFactory，线程的命名规则是“pool-数字-thread-数字”。

### RejectedExecutionHandler

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

## 扩展线程池

能扩展线程池的功能吗？比如在任务执行的前后做一点我们自己的业务工作？实际上，JDK 的线程池已经为我们预留的接口，在线程池核心方法中，有2 个方法是空的，就是给我们预留的。还有一个线程池退出时会调用的方法。参见代码cn.enjoyedu.ch6. ThreadPoolExt

可以看到，每个任务执行前后都会调用 beforeExecute和 afterExecute方法。相当于执行了一个切面。而在调用 shutdown 方法后则会调用 terminated 方法。

## 线程池的工作机制

1）如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3）如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务。

4）如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

## 提交任务

execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。

submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

## 关闭线程池

可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断**所有没有正在执行任务的线程**。

只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都已关闭后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。

## 合理地配置线程池

要想合理地配置线程池，就必须首先分析任务特性

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

•任务的性质：CPU密集型任务、IO密集型任务和混合型任务。

•任务的优先级：高、中和低。

•任务的执行时间：长、中和短。

•任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。

CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。

混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

对于IO型的任务的最佳线程数，有个公式可以计算

Nthreads = NCPU \* UCPU \* (1 + W/C)

其中：

❑NCPU是处理器的核的数目

❑UCPU是期望的CPU利用率（该值应该介于0和1之间）

❑W/C是等待时间与计算时间的比率

等待时间与计算时间我们在Linux下使用相关的vmstat命令或者top命令查看。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，等待的时间越长，则CPU空闲时间就越长，那么线程数应该设置得越大，这样才能更好地利用CPU。

建议使用有界队列。有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点儿，比如几千。

假设，我们现在有一个Web系统，里面使用了线程池来处理业务，在某些情况下，系统里后台任务线程池的队列和线程池全满了，不断抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞，任务积压在线程池里。

如果当时我们设置成无界队列，那么线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。

## 预定义线程池

### FixedThreadPool详解

创建使用固定线程数的FixedThreadPool的API。适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，它适用于负载比较重的服务器。FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。

当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的

最长时间，超过这个时间后多余的线程将被终止。这里把keepAliveTime设置为0L，意味着多余的空闲线程会被立即终止。

FixedThreadPool使用有界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。

### SingleThreadExecutor

创建使用单个线程的SingleThread-Executor的API，于需要保证顺序地执行各个任务；并且在任意时间点，不会有多个线程是活动的应用场景。

corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。SingleThreadExecutor使用有界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。

### CachedThreadPool

创建一个会根据需要创建新线程的CachedThreadPool的API。大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。

corePoolSize被设置为0，即corePool为空；maximumPoolSize被设置为Integer.MAX\_VALUE。这里把keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。

FixedThreadPool和SingleThreadExecutor使用有界队列LinkedBlockingQueue作为线程池的工作队列。CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。

### WorkStealingPool

利用所有运行的处理器数目来创建一个工作窃取的线程池，使用forkjoin实现

### ScheduledThreadPoolExecutor

使用工厂类Executors来创建。Executors可以创建2种类型的ScheduledThreadPoolExecutor，如下。

•ScheduledThreadPoolExecutor。包含若干个线程的ScheduledThreadPoolExecutor。

•SingleThreadScheduledExecutor。只包含一个线程的ScheduledThreadPoolExecutor。

ScheduledThreadPoolExecutor适用于需要多个后台线程执行周期任务，同时为了满足资源管理的需求而需要限制后台线程的数量的应用场景。

SingleThreadScheduledExecutor适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。

#### 提交定时任务

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)

//向定时任务线程池提交一个延时Runnable任务（仅执行一次）

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

//向定时任务线程池提交一个延时的Callable任务（仅执行一次）

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

//向定时任务线程池提交一个固定时间间隔执行的任务

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

//向定时任务线程池提交一个固定延时间隔执行的任务

固定时间间隔的任务不论每次任务花费多少时间，下次任务开始执行时间从理论上讲是确定的，当然执行任务的时间不能超过执行周期。

固定延时间隔的任务是指每次执行完任务以后都延时一个固定的时间。由于操作系统调度以及每次任务执行的语句可能不同，所以每次任务执行所花费的时间是不确定的，也就导致了每次任务的执行周期存在一定的波动。

#### 定时任务超时问题

scheduleAtFixedRate中，若任务处理时长超出设置的定时频率时长，本次任务执行完才开始下次任务，下次任务已经处于超时状态，会马上开始执行。

若任务处理时长小于定时频率时长，任务执行完后，定时器等待，下次任务会在定时器等待频率时长后执行。

如下例子：

设置定时任务每60s执行一次，那么从理论上应该第一次任务在第0s开始,第二次任务在第60s开始，第三次任务在120s开始，但实际运行时第一次任务时长80s，第二次任务时长30s，第三次任务时长50s，则实际运行结果为：

第一次任务第0s开始,第80s结束；

第二次任务第80s开始,第110s结束(上次任务已超时,本次不会再等待60s,会马上开始)；

第三次任务第120s开始,第170s结束.

第四次任务第180s开始.....

具体代码，参见cn.enjoyedu.ch6.schd. ScheduleWorkerTime

## CompletionService

CompletionService实际上可以看做是Executor和BlockingQueue的结合体。CompletionService在接收到要执行的任务时，通过类似BlockingQueue的put和take获得任务执行的结果。

CompletionService的一个实现是ExecutorCompletionService，ExecutorCompletionService把具体的计算任务交给Executor完成。

在实现上，ExecutorCompletionService在构造函数中会创建一个BlockingQueue（使用的基于链表的LinkedBlockingQueue），该BlockingQueue的作用是保存Executor执行的结果。

当提交一个任务到ExecutorCompletionService时，首先将任务包装成QueueingFuture，它是FutureTask的一个子类，然后改写FutureTask的done方法，之后把Executor执行的计算结果放入BlockingQueue中。

与ExecutorService最主要的区别在于submit的task不一定是按照加入时的顺序完成的。CompletionService对ExecutorService进行了包装，内部维护一个保存Future对象的BlockingQueue。只有当这个Future对象状态是结束的时候，才会加入到这个Queue中，take()方法其实就是Producer-Consumer中的Consumer。它会从Queue中取出Future对象，如果Queue是空的，就会阻塞在那里，直到有完成的Future对象加入到Queue中。所以，先完成的必定先被取出。这样就减少了不必要的等待时间。

参见代码cn.enjoyedu.ch6.comps. CompletionCase，我们可以得出结论：

使用方法一，自己创建一个集合来保存Future存根并循环调用其返回结果的时候，主线程并不能保证首先获得的是最先完成任务的线程返回值。它只是按加入线程池的顺序返回。因为take方法是阻塞方法，后面的任务完成了，前面的任务却没有完成，主程序就那样等待在那儿，只到前面的完成了，它才知道原来后面的也完成了。

使用方法二，使用CompletionService来维护处理线程不的返回结果时，主线程总是能够拿到最先完成的任务的返回值，而不管它们加入线程池的顺序。

# 7、并发安全

## 什么是线程安全性

在《Java并发编程实战》中，定义如下：

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在调用代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

### 线程封闭

实现好的并发是一件困难的事情，所以很多时候我们都想躲避并发。避免并发最简单的方法就是线程封闭。什么是线程封闭呢？

就是把对象封装到一个线程里，只有这一个线程能看到此对象。那么这个对象就算不是线程安全的也不会出现任何安全问题。实现线程封闭有哪些方法呢？

##### ad-hoc线程封闭

这是完全靠实现者控制的线程封闭，他的线程封闭完全靠实现者实现。Ad-hoc线程封闭非常脆弱，应该尽量避免使用。

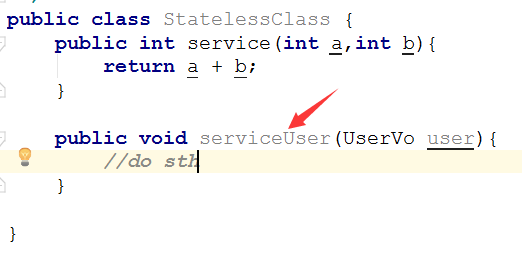
##### 栈封闭

栈封闭是我们编程当中遇到的最多的线程封闭。什么是栈封闭呢？简单的说就是局部变量。多个线程访问一个方法，此方法中的局部变量都会被拷贝一份到线程栈中。所以局部变量是不被多个线程所共享的，也就不会出现并发问题。所以能用局部变量就别用全局的变量，全局变量容易引起并发问题。

### 无状态的类

没有任何成员变量的类，就叫无状态的类，这种类一定是线程安全的。参见代码：cn.enjoyedu.ch7.safeclass. StatelessClass。

如果这个类的方法参数中使用了对象，也是线程安全的吗？比如：



当然也是，为何？因为多线程下的使用，固然user这个对象的实例会不正常，但是对于StatelessClass这个类的对象实例来说，它并不持有UserVo的对象实例，它自己并不会有问题，有问题的是UserVo这个类，而非StatelessClass本身。

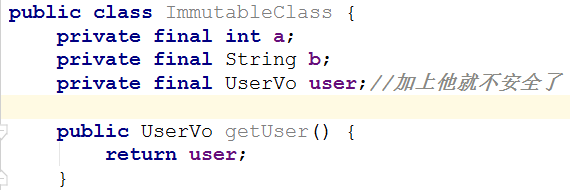
### 让类不可变

让状态不可变，两种方式：

1，加final关键字，对于一个类，所有的成员变量应该是私有的，同样的只要有可能，所有的成员变量应该加上final关键字，但是加上final，要注意如果成员变量又是一个对象时，这个对象所对应的类也要是不可变，才能保证整个类是不可变的。参见代码cn.enjoyedu.ch7.safeclass.ImmutableClass

2、根本就不提供任何可供修改成员变量的地方，同时成员变量也不作为方法的返回值。参见代码cn.enjoyedu.ch7.safeclass. ImmutableClassToo

但是要注意，一旦类的成员变量中有对象，上述的final关键字保证不可变并不能保证类的安全性，为何？因为在多线程下，虽然对象的引用不可变，但是对象在堆上的实例是有可能被多个线程同时修改的，没有正确处理的情况下，对象实例在堆中的数据是不可预知的。这就牵涉到了如何安全的发布对象这个问题。



### volatile

并不能保证类的线程安全性，只能保证类的可见性，最适合一个线程写，多个线程读的情景。

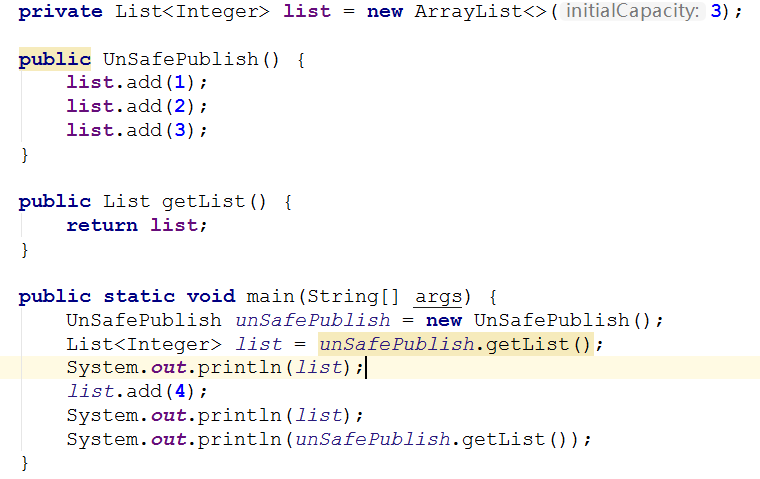
### 加锁和CAS

我们最常使用的保证线程安全的手段，使用**synchronized**关键字，使用显式锁，使用各种原子变量，修改数据时使用CAS机制等等。

### 安全的发布

类中持有的成员变量，如果是基本类型，发布出去，并没有关系，因为发布出去的其实是这个变量的一个副本，参见代码cn.enjoyedu.ch7.safeclass. SafePublish。

但是如果类中持有的成员变量是对象的引用，如果这个成员对象不是线程安全的，通过get等方法发布出去，会造成这个成员对象本身持有的数据在多线程下不正确的修改，从而造成整个类线程不安全的问题。参见代码cn.enjoyedu.ch7.safeclass. UnSafePublish，可以看见，



这个list发布出去后，是可以被外部线程之间修改，那么在多个线程同时修改的情况下不安全问题是肯定存在的，怎么修正这个问题呢？我们在发布这对象出去的时候，就应该用线程安全的方式包装这个对象。参见代码cn.enjoyedu.ch7.safeclass. SafePublishToo。我们将list用Collections.*synchronizedList*进行包装以后，无论多少线程使用这个list，就都是线程安全的了。



对于我们自己使用或者声明的类，JDK自然没有提供这种包装类的办法，但是我们可以仿造这种模式或者委托给线程安全的类，当然，对这种通过get等方法发布出去的对象，最根本的解决办法还是应该在实现上就考虑到线程安全问题，参见代码包cn.enjoyedu.ch7.safeclass.safepublish下的代码

### TheadLocal

ThreadLocal是实现线程封闭的最好方法。ThreadLocal内部维护了一个Map，Map的key是每个线程的名称，而Map的值就是我们要封闭的对象。每个线程中的对象都对应着Map中一个值，也就是ThreadLocal利用Map实现了对象的线程封闭。

### Servlet辨析

不是线程安全的类，为什么我们平时没感觉到：

1、在需求上，很少有共享的需求，2、接收到了请求，返回应答的时候，一般都是由一个线程来负责的。

但是只要Servlet中有成员变量，一旦有多线程下的写，就很容易产生线程安全问题。

## 死锁

### 概念

是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。

举个例子：A和B去按摩洗脚，都想在洗脚的时候，同时顺便做个头部按摩，13技师擅长足底按摩，14擅长头部按摩。

这个时候A先抢到14，B先抢到13，两个人都想同时洗脚和头部按摩，于是就互不相让，扬言我死也不让你，这样的话，A抢到14，想要13，B抢到13，想要14，在这个想同时洗脚和头部按摩的事情上A和B就产生了死锁。怎么解决这个问题呢？

第一种，假如这个时候，来了个15，刚好也是擅长头部按摩的，A又没有两个脑袋，自然就归了B，于是B就美滋滋的洗脚和做头部按摩，剩下A在旁边气鼓鼓的，这个时候死锁这种情况就被打破了，不存在了。

第二种，C出场了，用武力强迫A和B，必须先做洗脚，再头部按摩，这种情况下，A和B谁先抢到13，谁就可以进行下去，另外一个没抢到的，就等着，这种情况下，也不会产生死锁。

所以总结一下：

死锁是必然发生在多操作者（M>=2个）情况下，争夺多个资源（N>=2个，且N<=M）才会发生这种情况。很明显，单线程自然不会有死锁，只有B一个去，不要2个，打十个都没问题；单资源呢？只有13，A和B也只会产生激烈竞争，打得不可开交，谁抢到就是谁的，但不会产生死锁。同时，死锁还有一个重要的要求，争夺资源的顺序不对，如果争夺资源的顺序是一样的，也不会产生死锁。

#### 学术化的定义

死锁的发生必须具备以下四个必要条件。

1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2）请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3）不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

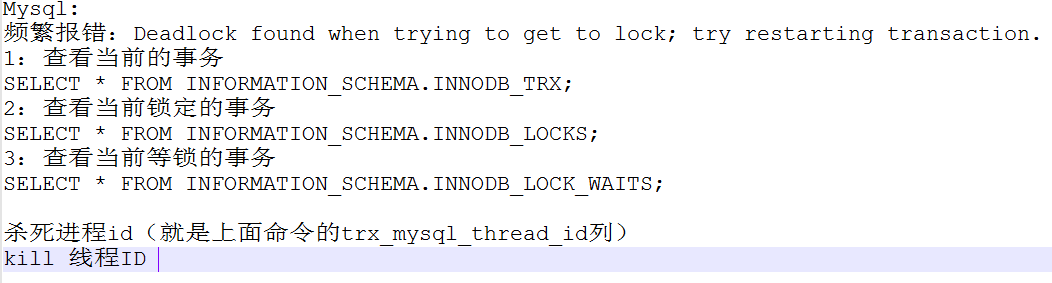
4）环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

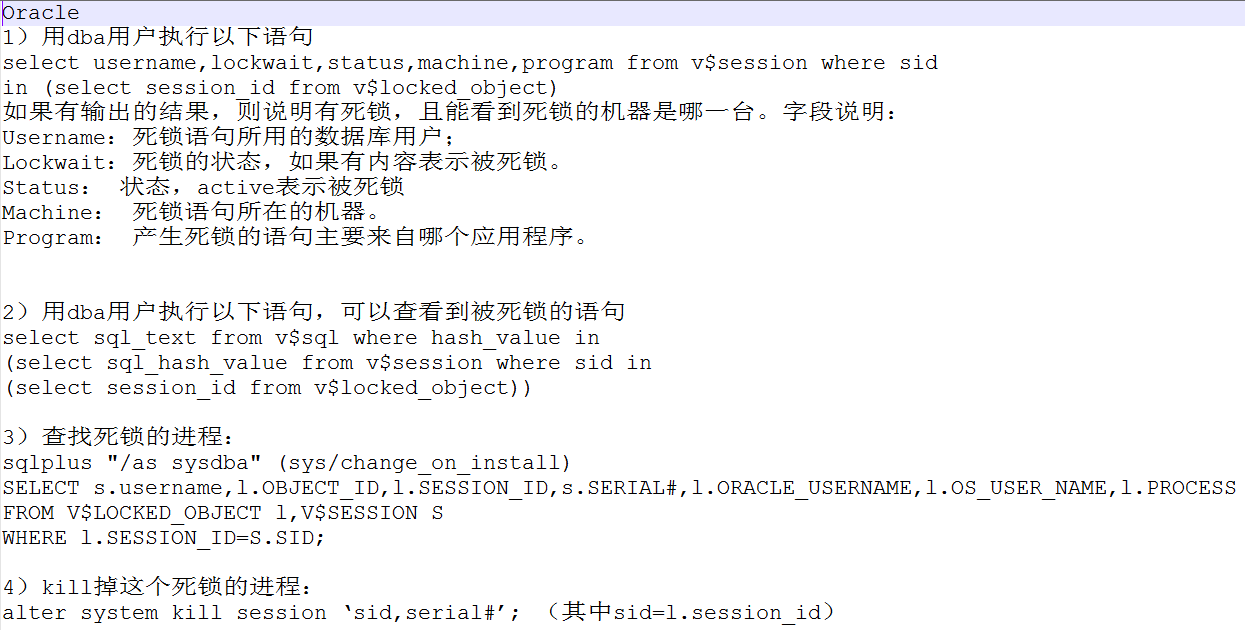
理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。只要打破四个必要条件之一就能有效预防死锁的发生：打破互斥条件：改造独占性资源为虚拟资源，大部分资源已无法改造。打破不可抢占条件：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。打破占有且申请条件：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。打破循环等待条件：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

避免死锁常见的算法有有序资源分配法、银行家算法。

### 现象、危害和解决

在我们IT世界有没有存在死锁的情况，有：数据库里多事务而且要同时操作多个表的情况下。所以数据库设计的时候就考虑到了检测死锁和从死锁中恢复的机制。比如oracle提供了检测和处理死锁的语句，而mysql也提供了“循环依赖检测的机制”





在Java世界里存在着多线程争夺多个资源，不可避免的存在着死锁。那么我们在编写代码的时候什么情况下会发生呢？

#### 现象

##### 简单顺序死锁

参见代码cn.enjoyedu.ch7. NormalDeadLock

##### 动态顺序死锁

顾名思义也是和获取锁的顺序有关，但是比较隐蔽，不像简单顺序死锁，往往从代码一眼就看出获取锁的顺序不对。

参见代码cn.enjoyedu.ch7.tranfer.service. UserAccount

#### 危害

1、线程不工作了，但是整个程序还是活着的2、没有任何的异常信息可以供我们检查。3、一旦程序发生了发生了死锁，是没有任何的办法恢复的，只能重启程序，对生产平台的程序来说，这是个很严重的问题。

##### 实际工作中的死锁

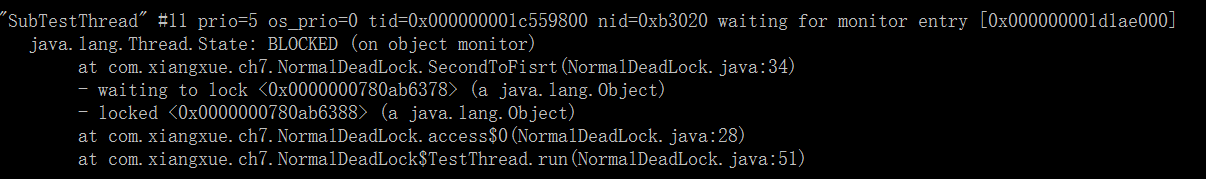
时间不定，不是每次必现；一旦出现没有任何异常信息，只知道这个应用的所有业务越来越慢，最后停止服务，无法确定是哪个具体业务导致的问题；测试部门也无法复现，并发量不够。

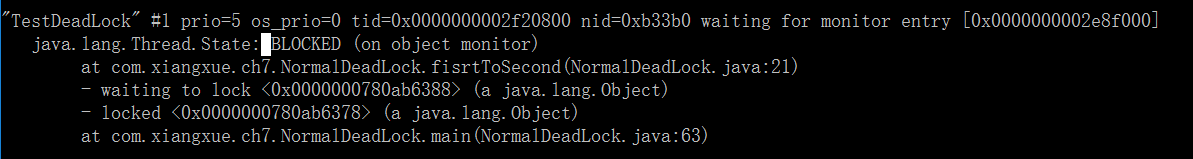
#### 解决

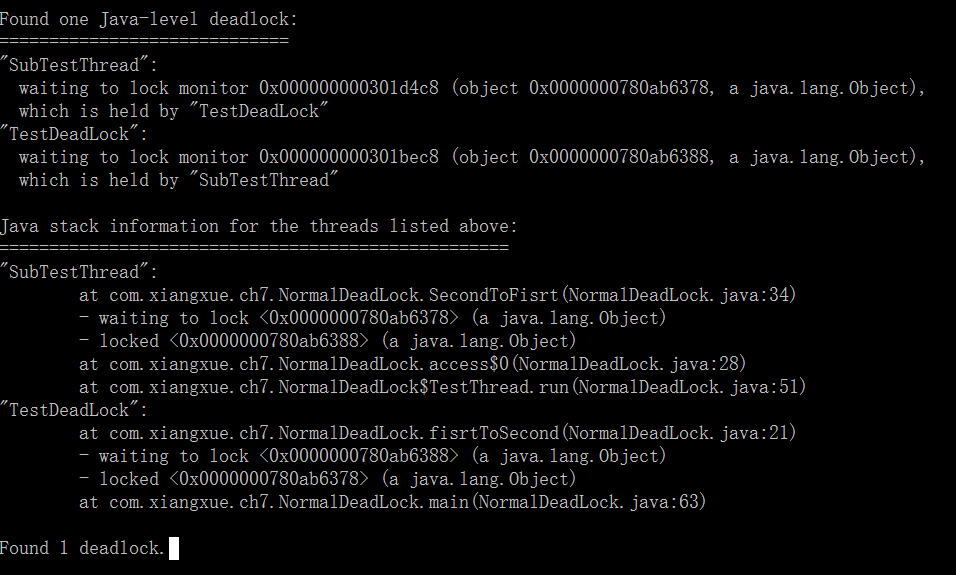
##### 定位

要解决死锁，当然要先找到死锁，怎么找？

通过**jps** 查询应用的 id，再通过**jstack id** 查看应用的锁的持有情况







##### 修正

关键是保证拿锁的顺序一致

两种解决方式

* 1. 内部通过顺序比较，确定拿锁的顺序；
  2. 采用尝试拿锁的机制。

参见代码cn.enjoyedu.ch7.tranfer.service. SafeOperate和SafeOperateToo

## 其他安全问题

### 活锁

两个线程在尝试拿锁的机制中，发生多个线程之间互相谦让，不断发生同一个线程总是拿到同一把锁，在尝试拿另一把锁时因为拿不到，而将本来已经持有的锁释放的过程。

解决办法：每个线程休眠随机数，错开拿锁的时间。

### 线程饥饿

低优先级的线程，总是拿不到执行时间

## 并发下的性能

使用并发的目标是为了提高性能，引入多线程后，其实会引入额外的开销，如线程之间的协调、增加的上下文切换，线程的创建和销毁，线程的调度等等。过度的使用和不恰当的使用，会导致多线程程序甚至比单线程还要低。

衡量应用的程序的性能：服务时间，延迟时间，吞吐量，可伸缩性等等，其中服务时间，延迟时间（多快），吞吐量（处理能力的指标，完成工作的多少）。多快和多少，完全独立，甚至是相互矛盾的。

对服务器应用来说：多少（可伸缩性，吞吐量）这个方面比多快更受重视。

我们做应用的时候：

**1、先保证程序正确，确实达不到要求的时候，再提高速度。（黄金原则）**

**2、一定要以测试为基准。**

### 线程引入的开销

#### 上下文切换

如果主线程是唯一的线程,那么它基本上不会被调度出去。另一方面,如果可运行的线程数大于CPU的数量,那么操作系统最终会将某个正在运行的线程调度出来,从而使其他线程能够使用CPU。这将导致一次上下文切换,在这个过程中将保存当前运行线程的执行上下文,并将新调度进来的线程的执行上下文设置为当前上下文。上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。

切换上下文需要一定的开销,而在线程调度过程中需要访问由操作系统和JVM共享的数据结构。应用程序、操作系统以及JVM都使用一组相同的CPU。在JVM和操作系统的代码中消耗越多的CPU时钟周期,应用程序的可用CPU时钟周期就越少。但上下文切换的开销并不只是包含JVM和操作系统的开销。当一个新的线程被切换进来时,它所需要的数据可能不在当前处理器的本地缓存中,因此上下文切换将导致一些缓存缺失,因而线程在首次调度运行时会更加缓慢。

当线程由于等待某个发生竞争的锁而被阻塞时,JVM通常会将这个线程挂起,并允许它被交换出去。如果线程频繁地发生阻塞,那么它们将无法使用完整的调度时间片。在程序中发生越多的阻塞(包括阻塞IO,等待获取发生竞争的锁,或者在条件变量上等待),与CPU密集型的程序就会发生越多的上下文切换,从而增加调度开销,并因此而降低吞吐量。

上下文切换是计算密集型操作。也就是说，它需要相当可观的处理器时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。上下文切换的实际开销会随着平台的不同而变化,然而按照经验来看:在大多数通用的处理器中,上下文切换的开销相当于50~10000个时钟周期,也就是几微秒。

UNIX系统的 vmstat命令能报告上下文切换次数以及在内核中执行时间所占比例等信息。如果内核占用率较高(超过10%),那么通常表示调度活动发生得很频繁,这很可能是由IO或竞争锁导致的阻塞引起的。

#### 内存同步

同步操作的性能开销包括多个方面。在 synchronized和 volatile提供的可见性保证中可能会使用一些特殊指令,即内存栅栏( Memory Barrier)。

内存栅栏可以刷新缓存,使缓存无效刷新硬件的写缓冲,以及停止执行管道。

内存栅栏可能同样会对性能带来间接的影响,因为它们将抑制一些编译器优化操作。在内存栅栏中,大多数操作都是不能被重排序的。

#### 阻塞

引起阻塞的原因：包括阻塞IO,等待获取发生竞争的锁,或者在条件变量上等待等等。

阻塞会导致线程挂起【挂起：挂起进程在操作系统中可以定义为暂时被淘汰出内存的进程，机器的资源是有限的，在资源不足的情况下，操作系统对在内存中的程序进行合理的安排，其中有的进程被暂时调离出内存，当条件允许的时候，会被操作系统再次调回内存，重新进入等待被执行的状态即就绪态，系统在超过一定的时间没有任何动作】。

很明显这个操作至少包括两次额外的上下文切换，还有相关的操作系统级的操作等等。

### 如何减少锁的竞争

#### 减少锁的粒度

使用锁的时候，锁所保护的对象是多个，当这些多个对象其实是独立变化的时候，不如用多个锁来一一保护这些对象。但是如果有同时要持有多个锁的业务方法，要注意避免发生死锁

#### 缩小锁的范围

对锁的持有实现快进快出，尽量缩短持由锁的的时间。将一些**与锁无关的代码**移出锁的范围，特别是一些耗时，可能阻塞的操作

#### 避免多余的锁

两次加锁之间的语句非常简单，导致加锁的时间比执行这些语句还长，这个时候应该进行锁粗化—扩大锁的范围。

#### 锁分段

ConcurrrentHashMap就是典型的锁分段。

#### 替换独占锁

在业务允许的情况下：

1. 使用读写锁，
2. 用自旋CAS

3、使用系统的并发容器