# Java 并发编程之美：线程相关的基础知识

### 前言

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了；相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的开篇，首先通过通俗易懂的方式先来和大家聊聊多线程并发编程线程有关基础知识（本文结合示例进行讲解，定会让你耳目一新），具体内容如下：

* 什么是线程？线程和进程的关系。
* 线程创建与运行。创建一个线程有那几种方式？有何区别？
* 线程通知与等待，多线程同步的基础设施。
* 线程的虚假唤醒，以及如何避免。
* 等待线程执行终止的 方法。想让主线程在子线程执行完毕后在做一点事情？
* 让线程睡眠的 sleep 方法，sleep 的线程会释放持有的锁？
* 线程中断。中断一个线程，被中断的线程会自己终止？
* 理解线程上下文切换。线程多了一定好？
* 线程死锁，以及如何避免。
* 守护线程与用户线程。当 main 函数执行完毕，但是还有用户线程存在的时候，JVM 进程会退出？

### 什么是线程

在讨论什么是线程前有必要先说下什么是进程，因为线程是进程中的一个实体，线程本身是不会独立存在的。进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，线程则是进程的一个执行路径，一个进程至少有一个线程，进程中的多个线程是共享进程的资源的。

操作系统在分配资源时候是把资源分配给进程的，但是 CPU 资源就比较特殊，它是分派到线程的，因为真正要占用 CPU 运行的是线程，所以也说线程是 CPU 分配的基本单位。

Java 中当我们启动 main 函数时候其实就启动了一个 JVM 的进程，而 main 函数所在线程就是这个进程中的一个线程，也叫做主线程。



如图一个进程中有多个线程，多个线程共享进程的堆和方法区资源，但是每个线程有自己的程序计数器，栈区域。

其中程序计数器是一块内存区域，用来记录线程当前要执行的指令地址，那么程序计数器为何要设计为线程私有的呢？前面说了线程是占用 CPU 执行的基本单位，而 CPU 一般是使用时间片轮转方式让线程轮询占用的，所以当前线程 CPU 时间片用完后，要让出 CPU，等下次轮到自己时候在执行，那么如何知道之前程序执行到哪里了？其实程序计数器就是为了记录该线程让出 CPU 时候的执行地址，待再次分配到时间片时候就可以从自己私有的计数器指定地址继续执行了。

另外每个线程有自己的栈资源，用于存储该线程的局部变量，这些局部变量是该线程私有的，其它线程是访问不了的，另外栈还用来存放线程的调用栈帧。

堆是一个进程中最大的一块内存，堆是被进程中的所有线程共享的，是进程创建时候分配的，堆里面主要存放使用 new 操作创建的对象实例。

方法区则是用来存放进程中的代码片段的，是线程共享的。

### 线程创建与运行

Java 中有三种线程创建方法，分别为实现 Runnable 接口的run方法、继承 Thread 类并重写 run 方法、使用 FutureTask 方式。

首先看下继承 Thread 方法的实现：

**public** **class** **ThreadTest** {

//继承Thread类并重写run方法

**public** **static** **class** **MyThread** **extends** **Thread** {

@Override

**public** **void** **run**() {

System.out.println("I am a child thread");

}

}

**public** **static** **void** **main**(String[] args) {

// 创建线程

MyThread thread = **new** MyThread();

// 启动线程

thread.start();

}

}

如上代码 MyThread 类继承了 Thread 类，并重写了 run 方法，然后调用了线程的 start 方法启动了线程，当创建完 thread 对象后该线程并没有被启动执行.

当调用了 start 方法后才是真正启动了线程。其实当调用了 start 方法后线程并没有马上执行而是处于就绪状态，这个就绪状态是指该线程已经获取了除 CPU 资源外的其它资源，等获取 CPU 资源后才会真正处于运行状态。

当 run 方法执行完毕，该线程就处于终止状态了。使用继承方式好处是 run 方法内获取当前线程直接使用 this 就可以，无须使用 Thread.currentThread() 方法，不好的地方是 Java 不支持多继承，如果继承了 Thread 类那么就不能再继承其它类，另外任务与代码没有分离，当多个线程执行一样的任务时候需要多份任务代码，而 Runable 则没有这个限制，下面看下实现 Runnable 接口的 run 方法方式：

**public** **static** **class** **RunableTask** **implements** **Runnable**{

@Override

**public** **void** **run**() {

System.out.println("I am a child thread");

}

}

**public** **static** **void** **main**(String[] args) **throws** InterruptedException{

RunableTask task = **new** RunableTask();

**new** Thread(task).start();

**new** Thread(task).start();

}

如上面代码，两个线程公用一个 task 代码逻辑，需要的话 RunableTask 可以添加参数进行任务区分，另外 RunableTask 可以继承其他类，但是上面两种方法都有一个缺点就是任务没有返回值，下面看最后一种是使用 FutureTask：

//创任务类，类似Runable**public** **static** **class** **CallerTask** **implements** **Callable**<**String**>{

@Override

**public** String **call**() **throws** Exception {

**return** "hello";

}

}

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

// 创建异步任务

FutureTask<String> futureTask = **new** FutureTask<>(**new** CallerTask());

//启动线程

**new** Thread(futureTask).start();

**try** {

//等待任务执行完毕，并返回结果

String result = futureTask.get();

System.out.println(result);

} **catch** (ExecutionException e) {

e.printStackTrace();

}

}

**注：**每种方式都有自己的优缺点，应该根据实际场景进行选择。

### 线程通知与等待

Java 中 Object 类是所有类的父类，鉴于继承机制，Java 把所有类都需要的方法放到了 Object 类里面，其中就包含本节要讲的通知等待系列函数,这些通知等待函数是组成并发包中线程同步组件的基础。

下面讲解下 Object 中关于线程同步的通知等待函数。

#### void wait() 方法

首先谈下什么是共享资源，所谓共享资源是说该资源被多个线程共享，多个线程都可以去访问或者修改的资源。另外本文当讲到的共享对象就是共享资源。

当一个线程调用一个共享对象的 wait() 方法时候，调用线程会被阻塞挂起，直到下面几个事情之一发生才返回：

1. 其它线程调用了该共享对象的 notify() 或者 notifyAll() 方法；
2. 其它线程调用了该线程的 interrupt() 方法设置了该线程的中断标志，该线程会抛出 InterruptedException 异常返回。

另外需要注意的是如果调用 wait() 方法的线程没有事先获取到该对象的监视器锁，则调用 wait() 方法时候调用线程会抛出 IllegalMonitorStateException 异常。

那么一个线程如何获取到一个共享变量的监视器那？

（1）执行使用 synchronized 同步代码块时候，使用该共享变量作为参数：

**synchronized**（共享变量）{

//doSomething

}

（2）调用该共享变量的方法，并且该方法使用了 synchronized 修饰：

**synchronized** **void** **add**(**int** a,**int** b){

//doSomething

}

另外需要注意的是一个线程可以从挂起状态变为可以运行状态（也就是被唤醒）即使该线程没有被其它线程调用 notify()，notifyAll() 进行通知，或者被中断，或者等待超时，这就是所谓的虚假唤醒。

虽然虚假唤醒在应用实践中很少发生，但是还是需要防范于未然的，做法就是不停的去测试该线程被唤醒的条件是否满足，不满足则继续等待，也就是说在一个循环中去调用 wait() 方法进行防范，退出循环的条件是条件满足了唤醒该线程。

**synchronized** (obj) {

**while** (条件不满足){

obj.wait();

}

}

如上代码是经典的调用共享变量 wait() 方法的实例，首先通过同步块获取 obj 上面的监视器锁，然后通过 while 循环内调用 obj 的 wait() 方法。

下面从生产者消费者例子来加深理解，如下面代码是一个生产者的例子，其中 queue 为共享变量，生产者线程在调用 queue 的 wait 方法前，通过使用 synchronized 关键字拿到了该共享变量 queue 的监视器，所以调用 wait() 方法才不会抛出 IllegalMonitorStateException 异常，如果当前队列没有空闲容量则会调用 queued 的 wait() 挂起当前线程，这里使用循环就是为了避免上面说的虚假唤醒问题，这里假如当前线程虚假唤醒了，但是队列还是没有空余容量的话，当前线程还是会调用 wait() 把自己挂起。

//生产线程**synchronized** (queue) {

//消费队列满，则等待队列空闲

**while** (queue.size() == MAX\_SIZE) {

**try** {

//挂起当前线程，并释放通过同步块获取的queue上面的锁，让消费线程可以获取该锁，然后获取队列里面元素

queue.wait();

} **catch** (Exception ex) {

ex.printStackTrace();

}

}

//空闲则生成元素，并通知消费线程

queue.add(ele);

queue.notifyAll();

}

}

//消费线程**synchronized** (queue) {

//消费队列为空

**while** (queue.size() == 0) {

**try**

//挂起当前线程，并释放通过同步块获取的queue上面的锁，让生产线程可以获取该锁，生产元素放入队列

queue.wait();

} **catch** (Exception ex) {

ex.printStackTrace();

}

}

//消费元素，并通知唤醒生产线程

queue.take();

queue.notifyAll();

}

}

另外当一个线程调用了共享变量的 wait() 方法后该线程会被挂起，同时该线程会暂时释放对该共享变量监视器的持有，直到另外一个线程调用了共享变量的 notify() 或者 notifyAll() 方法才有可能会重新获取到该共享变量的监视器的持有权（这里说有可能，是因为考虑到多个线程第一次都调用了 wait() 方法，所以多个线程会竞争持有该共享变量的监视器）。

借用上面这个例子来讲解下调用共享变量 wait() 方法后当前线程会释放持有的共享变量的锁的理解。

如上代码假如生产线程 A 首先通过 synchronized 获取到了 queue 上的锁，那么其它生产线程和所有消费线程都会被阻塞，线程 A 获取锁后发现当前队列已满会调用 queue.wait() 方法阻塞自己，然后会释放获取的 queue 上面的锁，这里考虑下为何要释放该锁？如果不释放，由于其它生产线程和所有消费线程已经被阻塞挂起，而线程 A 也被挂起，这就处于了死锁状态。这里线程 A 挂起自己后释放共享变量上面的锁就是为了打破死锁必要条件之一的持有并等待原则。关于死锁下面章节会有讲到，线程 A 释放锁后其它生产线程和所有消费线程中会有一个线程获取 queue 上的锁进而进入同步块，这就打破了死锁。

最后再举一个例子说明当一个线程调用共享对象的 wait() 方法被阻塞挂起后，如果其它线程中断了该线程，则该线程会抛出 InterruptedException 异常后返回：

**public** **class** **WaitNotifyInterupt** {

**static** Object obj = **new** Object();

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

//创建线程

Thread threadA = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.out.println("---begin---");

//阻塞当前线程

**synchronized** (obj) {

obj.wait();

}

System.out.println("---end---");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

threadA.start();

Thread.sleep(1000);

System.out.println("---begin interrupt threadA---");

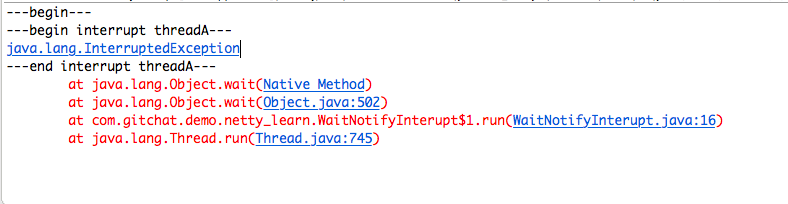
threadA.interrupt();

System.out.println("---end interrupt threadA---");

}

}

运行上面代码输出为：



如上代码 threadA 调用了共享对 obj 的 wait（） 方法后阻塞挂起了自己，然后主线程在休眠1s后中断了 threadA 线程，可知中断后 threadA 在 obj.wait() 处抛出了 java.lang.IllegalMonitorStateException 异常后返回后终止。

#### void wait(long timeout) 方法

该方法相比 wait() 方法多一个超时参数，不同在于如果一个线程调用了共享对象的该方法挂起后，如果没有在指定的 timeout ms 时间内被其它线程调用该共享变量的 notify() 或者 notifyAll() 方法唤醒，那么该函数还是会因为超时而返回。

需要注意的是如果在调用该函数时候 timeout 传递了负数会抛出 IllegalArgumentException 异常。

#### void wait(long timeout, int nanos) 方法

内部是调用 wait(long timeout)，如下代码：只是当 nanos>0 时候让参数一递增1。

**public** **final** **void** **wait**(**long** timeout, **int** nanos) **throws** InterruptedException {

**if** (timeout < 0) {

**throw** **new** IllegalArgumentException("timeout value is negative");

}

**if** (nanos < 0 || nanos > 999999) {

**throw** **new** IllegalArgumentException(

"nanosecond timeout value out of range");

}

**if** (nanos > 0) {

timeout++;

}

wait(timeout);

}

#### void notify() 方法

一个线程调用共享对象的 notify() 方法后，会唤醒一个在该共享变量上调用 wait 系列方法后被挂起的线程，一个共享变量上可能会有多个线程在等待，具体唤醒哪一个等待的线程是随机的。

另外被唤醒的线程不能马上从 wait 返回继续执行，它必须获取了共享对象的监视器后才可以返回,也就是唤醒它的线程释放了共享变量上面的监视器锁后，被唤醒它的线程也不一定会获取到共享对象的监视器，这是因为该线程还需要和其它线程一块竞争该锁，只有该线程竞争到了该共享变量的监视器后才可以继续执行。

类似 wait 系列方法，只有当前线程已经获取到了该共享变量的监视器锁后，才可以调用该共享变量的 notify() 方法，否者会抛出 IllegalMonitorStateException 异常。

#### void notifyAll() 方法

不同于 nofity() 方法在共享变量上调用一次就会唤醒在该共享变量上调用 wait 系列方法被挂起的一个线程，notifyAll() 则会唤醒所有在该共享变量上由于调用 wait 系列方法而被挂起的线程。

最后本小节最后讲一个例子来说明 notify() 和 notifyAll() 的具体含义和一些需要注意的地方，代码实例如下：

**private** **static** **volatile** Object resourceA = **new** Object();

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

// 创建线程

Thread threadA = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

// 获取resourceA共享资源的监视器锁

**synchronized** (resourceA) {

System.out.println("threadA get resourceA lock");

**try** {

System.out.println("threadA begin wait");

resourceA.wait();

System.out.println("threadA end wait");

} **catch** (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

});

// 创建线程

Thread threadB = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**synchronized** (resourceA) {

System.out.println("threadB get resourceA lock");

**try** {

System.out.println("threadB begin wait");

resourceA.wait();

System.out.println("threadB end wait");

} **catch** (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

});

// 创建线程

Thread threadC = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**synchronized** (resourceA) {

System.out.println("threadC begin notify");

resourceA.notifyAll();

}

}

});

// 启动线程

threadA.start();

threadB.start();

Thread.sleep(1000);

threadC.start();

// 等待线程结束

threadA.();

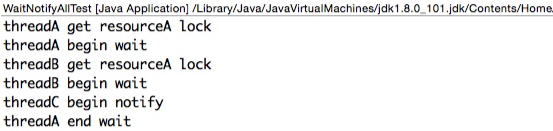
threadB.();

threadC.();

System.out.println("main over");

}

输出结果：



如上代码开启了三个线程，其中线程 A 和 B 分别调用了共享资源 resourceA 的 wait() 方法，线程 C 则调用了 nofity() 方法。

这里启动线程 C 前首先调用 sleep 方法让主线程休眠 1s，目的是让线程 A 和 B 全部执行到调用 wait 方法后在调用线程 C 的 notify 方法。

这个例子企图希望在线程 A 和线程 B 都因调用共享资源 resourceA 的 wait() 方法而被阻塞后，线程 C 在调用 resourceA 的 notify() 方法，希望可以唤醒线程 A 和线程 B，但是从执行结果看只有一个线程 A 被唤醒了，线程 B 没有被唤醒，

从结果看线程调度器这次先调度了线程 A 占用 CPU 来运行，线程 A 首先获取 resourceA 上面的锁，然后调用 resourceA 的 wait() 方法挂起当前线程并释放获取到的锁，然后线程 B 获取到 resourceA 上面的锁并调用了 resourceA 的 wait()，此时线程 B 也被阻塞挂起并释放了 resourceA 上的锁。

线程 C 休眠结束后在共享资源 resourceA 上调用了 notify() 方法，则会激活 resourceA 的阻塞集合里面的一个线程，这里激活了线程 A，所以线程 A 调用的 wait() 方法返回了，线程 A 执行完毕。而线程 B 还处于阻塞状态。

如果把线程 C 里面调用的 notify() 改为调用 notifyAll() 而执行结果如下：



可知线程 A 和线程 B 被挂起后，线程 C 调用 notifyAll() 函数会唤醒在 resourceA 等待的所有线程，这里线程 A 和线程 B 都会被唤醒，只是线程 B 先获取到 resourceA 上面的锁然后从 wait() 方法返回，等线程 B 执行完毕后，线程 A 又获取了 resourceA 上面的锁，然后从 wait() 方返回，当线程 A 执行完毕，主线程就返回后，然后打印输出。

**注：**在调用具体共享对象的 wait 或者 notify 系列函数前要先获取共享对象的锁；另外通知和等待是实现线程同步的原生方法，理解它们的协作功能很有必要；最后由于线程虚假唤醒的存在，一定要使用循环检查的方式。

### 等待线程执行终止的 join 方法

在项目实践时候经常会遇到一个场景，就是需要等待某几件事情完成后才能继续往下执行，比如多个线程去加载资源，当多个线程全部加载完毕后在汇总处理，Thread 类中有个 join 方法就可以做这个事情，前面介绍的等待通知方法是属于 Object 类的，而 join 方法则是直接在 Thread 类里面提供的，join 是无参，返回值为 void 的方法。下面看一个简单的例子来介绍 join 的使用：

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

Thread threadOne = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println("child threadOne over!");

}

});

Thread threadTwo = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println("child threadTwo over!");

}

});

//启动子线程

threadOne.start();

threadTwo.start();

System.out.println("wait all child thread over!");

//等待子线程执行完毕，返回

threadOne.join();

threadTwo.join();

System.out.println("all child thread over!");

}

如代码主线程里面启动了两个子线程，然后在分别调用了它们的 join() 方法，那么主线程首先会阻塞到 threadOne.join() 方法，等 threadOne 执行完毕后返回，threadOne 执行完毕后 threadOne.join() 就会返回，然后主线程调用 threadTwo.join() 后再次被阻塞，等 threadTwo 执行完毕后主线程也就返回了。这里只是为了演示 join 的作用，对应这类需求后面会讲的 CountDownLatch 是不错选择。

另外线程 A 调用线程 B 的 join 方法后会被阻塞，当其它线程调用了线程 B 的 interrupt() 方法中断了线程 B 时候，线程 B 会抛出 InterruptedException 异常而返回，下面通过一个例子来加深理解：

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

//线程one

Thread threadOne = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

System.out.println("threadOne begin run!");

**for** (;;) {

}

}

});

//获取主线程

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

//线程two

Thread threadTwo = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

//休眠1s

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

//中断主线程

mainThread.interrupt();

}

});

// 启动子线程

threadOne.start();

//延迟1s启动线程

threadTwo.start();

**try**{//等待线程one执行结束

threadOne.join();

}**catch**(InterruptedException e){

System.out.println("main thread:" + e);

}

}

输出结果：



如上代码 threadOne 线程里面执行死循环，主线程调用 threadOne 的 join 方法阻塞自己等待线程 threadOne 执行完毕，待 threadTwo 休眠 1s 后会调用主线程的 interrupt() 方法设置主线程的中断标志。

从结果看主线程中 threadOne.join() 处会抛出 InterruptedException 异常而返回。这里需要注意的是 threadTwo 里面调用的是主线程的 interrupt()，而不是线程 threadOne 的。

**注：**由于 CountDownLatch 功能比 join 更丰富，所以项目实践中一般使用 CountDownLatch，关于 CountDownLatch，后面 Chat《Java 并发编程之美高级篇》会有具体讲解。

### 让线程睡眠的 sleep 方法

Thread 类中有一个静态的 sleep 方法，当一个执行中的线程调用了 Thread 的 sleep 方法后，调用线程会暂时让出指定时间的执行权，也就是这期间不参与 CPU 的调度，但是该线程所拥有的监视器资源，比如锁还是持有不让出的。当指定的睡眠时间到了该函数会正常返回，线程就处于就绪状态，然后参与 CPU 的调度，当获取到了 CPU 资源就可以继续运行了。如果在睡眠期间其它线程调用了该线程的 interrupt() 方法中断了该线程,该线程会在调用 sleep 的地方抛出 InterruptedException 异常返回。

首先看一个例子来说明线程在睡眠时候拥有的监视器资源不会被释放是什么意思：

**public** **class** **SleepTest2** {

// 创建一个独占锁

**private** **static** **final** Lock lock = **new** ReentrantLock();

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

// 创建线程A

Thread threadA = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

// 获取独占锁

lock.lock();

**try** {

System.out.println("child threadA is in sleep");

Thread.sleep(10000);

System.out.println("child threadA is in awaked");

} **catch** (InterruptedException e) {

e.printStackTrace();

} **finally** {

// 释放锁

lock.unlock();

}

}

});

// 创建线程B

Thread threadB = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

// 获取独占锁

lock.lock();

**try** {

System.out.println("child threadB is in sleep");

Thread.sleep(10000);

System.out.println("child threadB is in awaked");

} **catch** (InterruptedException e) {

e.printStackTrace();

} **finally** {

// 释放锁

lock.unlock();

}

}

});

// 启动线程

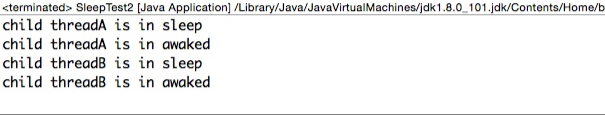
threadA.start();

threadB.start();

}

}

执行结果：



如上代码首先创建了一个独占锁，然后创建了两个线程，每个线程内部先获取锁，然后睡眠，睡眠结束后会释放锁。

首先无论你执行多少遍上面的代码都是先输出线程 A 的打印或者先输出线程 B 的打印，不会存在线程 A 和线程 B 交叉打印的情况。

从执行结果看线程 A 先获取了锁，那么线程 A 会先打印一行，然后调用 sleep 让自己沉睡 10s，在线程 A 沉睡的这 10s 内那个独占锁 lock 还是线程 A 自己持有的，线程 B 会一直阻塞直到线程 A 醒过来后执行 unlock 释放锁。

下面在来看下当一个线程处于睡眠时候如果另外一个线程中断了它，会不会在调用 sleep 处抛出异常。

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

//创建线程

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.out.println("child thread is in sleep");

Thread.sleep(10000);

System.out.println("child thread is in awaked");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

//启动线程

thread.start();

//主线程休眠2s

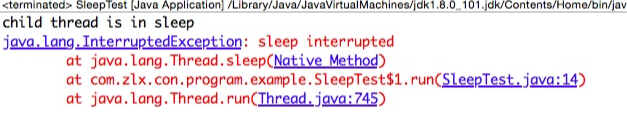
Thread.sleep(2000);

//主线程中断子线程

thread.interrupt();

}

执行结果：



如上代码在子线程睡眠期间主线程中断了它，所以子线程在调用 sleep 处抛出了 InterruptedException 异常。

**注：** sleep 方法只是会让调用线程暂时让出指定时间的 CPU 执行权，但是该线程所拥有的监视器资源，比如锁还是持有不让出的。

### 线程中断

Java 中线程中断是一种线程间协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是需要被中断的线程根据中断状态自行处理。

* void interrupt() 方法

中断线程，例如当线程 A 运行时，线程 B 可以调用线程 A 的 interrupt() 方法来设置线程 A 的中断标志为 true 并立即返回。设置标志仅仅是设置标志，线程 A 并没有实际被中断，会继续往下执行的。如果线程 A 因为调用了 wait 系列函数或者 join 方法或者 sleep 函数而被阻塞挂起，这时候线程 B 调用了线程 A 的 interrupt() 方法，线程 A 会在调用这些方法的地方抛出 InterruptedException 异常而返回。

* boolean isInterrupted()

检测当前线程是否被中断，如果是返回 true，否者返回 false。

**public** **boolean** **isInterrupted**() {

//传递false，说明不清除中断标志

**return** isInterrupted(**false**);

}

* boolean interrupted()

检测当前线程是否被中断，如果是返回 true，否者返回 false，与 isInterrupted 不同的是该方法如果发现当前线程被中断后会清除中断标志，并且该函数是 static 方法，可以通过 Thread 类直接调用。另外从下面代码可以知道 interrupted() 内部是获取当前调用线程的中断标志而不是调用 interrupted() 方法的实例对象的中断标志。

**public** **static** **boolean** **interrupted**() {

//清除中断标志

**return** currentThread().isInterrupted(**true**);

}

下面看一个线程使用 Interrupted 优雅退出的经典使用例子，代码如下：

**public** **void** **run**(){

**try**{

....

//线程退出条件

**while**(!Thread.currentThread().isInterrupted()&& more work to **do**){

// do more work;

}

}**catch**(InterruptedException e){

// thread was interrupted during sleep or wait

}

**finally**{

// cleanup, if required

}

}

下面看一个根据中断标志判断线程是否终止的例子：

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

//如果当前线程被中断则退出循环

**while** (!Thread.currentThread().isInterrupted())

System.out.println(Thread.currentThread() + " hello");

}

});

//启动子线程

thread.start();

//主线程休眠1s，以便中断前让子线程输出点东西

Thread.sleep(1);

//中断子线程

System.out.println("main thread interrupt thread");

thread.interrupt();

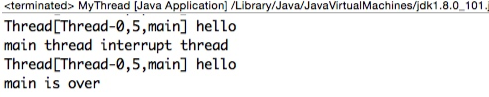
//等待子线程执行完毕

thread.join();

System.out.println("main is over");

}

输出结果：



如上代码子线程 thread 通过检查当前线程中断标志来控制是否退出循环，主线程在休眠 1s 后调用 thread 的 interrupt() 方法设置了中断标志，所以线程 thread 退出了循环。

**注：**中断一个线程仅仅是设置了该线程的中断标志，也就是设置了线程里面的一个变量的值，本身是不能终止当前线程运行的，一般程序里面是检查这个标志的状态来判断是否需要终止当前线程。

### 理解线程上下文切换

在多线程编程中，线程个数一般都大于 CPU 个数，而每个 CPU 同一时刻只能被一个线程使用，为了让用户感觉多个线程是在同时执行，CPU 资源的分配采用了时间片轮转的策略，也就是给每个线程分配一个时间片，在时间片内占用 CPU 执行任务。当前线程的时间片使用完毕后当前就会处于就绪状态并让出 CPU 让其它线程占用，这就是上下文切换，从当前线程的上下文切换到了其它线程。

那么就有一个问题让出 CPU 的线程等下次轮到自己占有 CPU 时候如何知道之前运行到哪里了？所以在切换线程上下文时候需要保存当前线程的执行现场，当再次执行时候根据保存的执行现场信息恢复执行现场。

线程上下文切换时机：

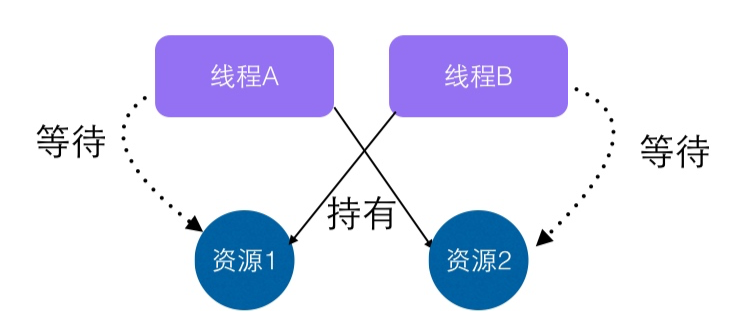
* 当前线程的 CPU 时间片使用完毕处于就绪状态时候；
* 当前线程被其它线程中断时候。

**注：**由于线程切换是有开销的，所以并不是开的线程越多越好，比如如果机器是4核心的，你开启了100个线程，那么同时执行的只有4个线程，这100个线程会来回切换线程上下文来共享这四个 CPU。

### 线程死锁

#### 什么是线程死锁

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去。



如上图，线程 A 已经持有了资源1的同时还想要资源2，线程 B 在持有资源2的时候还想要资源1，所以线程1和线程2就相互等待对方已经持有的资源，就进入了死锁状态。

那么产生死锁的原因都有哪些，学过操作系统的应该都知道死锁的产生必须具备以下四个必要条件。

* 互斥条件：指线程对已经获取到的资源进行排它性使用，即该资源同时只由一个线程占用。如果此时还有其它进行请求获取该资源，则请求者只能等待，直至占有资源的线程用毕释放。
* 请求并持有条件：指一个线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其其它线程占有，所以当前线程会被阻塞，但阻塞的同时并不释放自己已经获取的资源。
* 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后由自己释放。
* 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合{T0，T1，T2，···，Tn}中的 T0 正在等待一个 T1 占用的资源；T1 正在等待 T2 占用的资源，……Tn正在等待已被 T0 占用的资源。

下面通过一个案例来说明线程死锁：

**public** **class** **DeadLockTest2** {

// 创建资源

**private** **static** Object resourceA = **new** Object();

**private** **static** Object resourceB = **new** Object();

**public** **static** **void** **main**(String[] args) {

// 创建线程A

Thread threadA = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**synchronized** (resourceA) {

System.out.println(Thread.currentThread() + " get ResourceA");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread() + "waiting get ResourceB");

**synchronized** (resourceB) {

System.out.println(Thread.currentThread() + "get ResourceB");

}

}

}

});

// 创建线程B

Thread threadB = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**synchronized** (resourceB) {

System.out.println(Thread.currentThread() + " get ResourceB");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread() + "waiting get ResourceA");

**synchronized** (resourceA) {

System.out.println(Thread.currentThread() + "get ResourceA");

}

};

}

});

// 启动线程

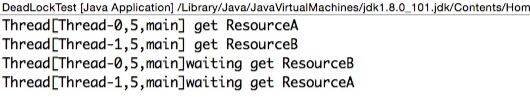
threadA.start();

threadB.start();

}

}

输出结果：



下面分析下代码和结果，其中 Thread-0 是线程 A，Thread-1 是线程 B，代码首先创建了两个资源，并创建了两个线程。

从输出结果可以知道线程调度器先调度了线程 A，也就是把 CPU 资源让给了线程 A，线程 A 调用了 getResourceA() 方法，方法里面使用 synchronized(resourceA) 方法获取到了 resourceA 的监视器锁，然后调用 sleep 函数休眠 1s，休眠 1s 是为了保证线程 A 在执行 getResourceB 方法前让线程 B 抢占到 CPU 执行 getResourceB 方法。

线程 A 调用了 sleep 期间，线程 B 会执行 getResourceB 方法里面的 synchronized(resourceB)，代表线程 B 获取到了 objectB 对象的监视器锁资源，然后调用 sleep 函数休眠 1S。

好了，到了这里线程 A 获取到了 objectA 的资源，线程 B 获取到了 objectB 的资源。线程 A 休眠结束后会调用 getResouceB 方法企图获取到 ojbectB 的资源，而 ObjectB 资源被线程 B 所持有，所以线程 A 会被阻塞而等待。而同时线程 B 休眠结束后会调用 getResourceA 方法企图获取到 objectA 上的资源，而资源 objectA 已经被线程 A 持有，所以线程 A 和 B 就陷入了相互等待的状态也就产生了死锁。

下面从产生死锁的四个条件来谈谈本案例如何满足了四个条件。

首先资源 resourceA 和 resourceB 都是互斥资源，当线程 A 调用 synchronized(resourceA) 获取到 resourceA 上的监视器锁后释放前，线程 B 在调用 synchronized(resourceA) 尝试获取该资源会被阻塞，只有线程 A 主动释放该锁，线程 B 才能获得，这满足了资源互斥条件。

线程 A 首先通过 synchronized(resourceA) 获取到 resourceA 上的监视器锁资源，然后通过 synchronized(resourceB) 等待获取到 resourceB 上的监视器锁资源，这就构造了持有并等待。

线程 A 在获取 resourceA 上的监视器锁资源后，不会被线程 B 掠夺走，只有线程 A 自己主动释放 resourceA 的资源时候，才会放弃对该资源的持有权，这构造了资源的不可剥夺条件。

线程 A 持有 objectA 资源并等待获取 objectB 资源，而线程 B 持有 objectB 资源并等待 objectA 资源，这构成了循环等待条件。

所以线程 A 和 B 就形成了死锁状态。

#### 如何避免线程死锁

要想避免死锁，需要破坏构造死锁必要条件的至少一个即可，但是学过操作系统童鞋应该都知道目前只有持有并等待和循环等待是可以被破坏的。

造成死锁的原因其实和申请资源的顺序有很大关系，使用资源申请的有序性原则就可以避免死锁，那么什么是资源的有序性呢，先看一下对上面代码的修改：

// 创建线程B

Thread threadB = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**synchronized** (resourceA) {

System.out.println(Thread.currentThread() + " get ResourceB");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread() + "waiting get ResourceA");

**synchronized** (resourceB) {

System.out.println(Thread.currentThread() + "get ResourceA");

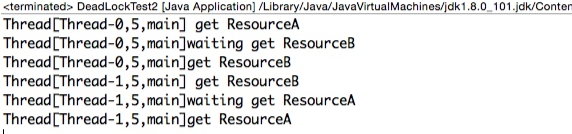
}

};

}

});

输出结果：



如上代码可知修改了线程 B 中获取资源的顺序和线程 A 中获取资源顺序一致，其实资源分配有序性就是指假如线程 A 和 B 都需要资源1，2，3……n 时候，对资源进行排序，线程 A 和 B 只有在获取到资源 n-1 时候才能去获取资源 n。

**注：**编写并发程序，多个线程进行共享多个资源时候要注意采用资源有序分配法避免死锁的产生。

### 守护线程与用户线程

Java 中线程分为两类，分别为 Daemon 线程（守护线程）和 User 线程（用户线程），在 JVM 启动时候会调用 main 函数，main 函数所在的线程是一个用户线程，这个是我们可以看到的线程，其实 JVM 内部同时还启动了好多守护线程，比如垃圾回收线程（严格说属于 JVM 线程）。

那么守护线程和用户线程有什么区别那？区别之一是当最后一个非守护线程结束时候，JVM 会正常退出，而不管当前是否有守护线程；也就是说守护线程是否结束并不影响 JVM 的退出。言外之意是只要有一个用户线程还没结束正常情况下 JVM 就不会退出。

那么 Java 中如何创建一个守护线程呢？代码如下：

**public** **static** **void** **main**(String[] args) {

Thread daemonThread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

}

});

//设置为守护线程

daemonThread.setDaemon(**true**);

daemonThread.start();

}

可知只需要设置线程的 daemon 参数为 true 即可。

下面通过例子来加深用户线程与守护线程的区别的理解，首先看下面代码：

**public** **static** **void** **main**(String[] args) {

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**for**(;;){}

}

});

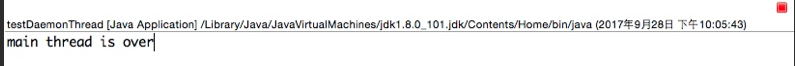
//启动子线

thread.start();

System.out.print("main thread is over");

}

结果输出为：



如上代码在 main 线程中创建了一个 thread 线程，thread 线程里面是无限循环，运行代码从结果看 main 线程已经运行结束了，那么 JVM 进程已经退出了？从 IDE 的输出结果右侧上的红色方块说明 JVM 进程并没有退出，另外 Mac 上执行 ps -eaf | grep java 会输出结果，也可以证明这个结论。

这个结果说明了当父线程结束后，子线程还是可以继续存在的，也就是子线程的生命周期并不受父线程的影响。也说明了当用户线程还存在的情况下 JVM 进程并不会终止。

那么我们把上面的 thread 线程设置为守护线程后在运行看看会有什么效果：

//设置为守护线程

thread.setDaemon(**true**);

//启动子线

thread.start();

执行结果为：



如上在启动线程前设置线程为守护线程，从输出结果可知 JVM 进程已经终止了，执行 ps -eaf |grep java 也看不到 JVM 进程了。这个例子里面 main 函数是唯一的用户线程，thread 线程是守护线程，当 main 线程运行结束后，JVM 发现当前已经没有用户线程了，就会终止 JVM 进程。

Java 中在 main 线程运行结束后，JVM 会自动启动一个叫做 DestroyJavaVM 线程，该线程会等待所有用户线程结束后终止 JVM 进程，下面通过简单的 JVM 代码来证明这个结论：

翻开 JVM 的代码，最终会调用到 JavaMain 这个函数：

**int** JNICALL**JavaMain**(**void** \* \_args){

...

//执行Java中的main函数

(\*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);

//main函数返回值

ret = (\*env)->ExceptionOccurred(env) == NULL ? 0 : 1;

//等待所有非守护线程结束，然后销毁JVM进程

LEAVE();

}

LEAVE 是 C 语言里面的一个宏定义，定义如下：

#define **LEAVE**() \

**do** { \

**if** ((\*vm)->DetachCurrentThread(vm) != JNI\_OK) { \

JLI\_ReportErrorMessage(JVM\_ERROR2); \

ret = 1; \

} \

**if** (JNI\_TRUE) { \

(\*vm)->DestroyJavaVM(vm); \

**return** ret; \

} \

} **while** (JNI\_FALSE)

上面宏的作用实际是创建了一个名字叫做 DestroyJavaVM 的线程来等待所有用户线程结束。

在 Tomcat 的 NIO 实现 NioEndpoint 中会开启一组接受线程用来接受用户的链接请求和一组处理线程负责具体处理用户请求，那么这些线程是用户线程还是守护线程呢？下面我们看下 NioEndpoint 的 startInternal 方法：

**public** **void** **startInternal**() **throws** Exception {

**if** (!running) {

running = **true**;

paused = **false**;

...

//创建处理线程

pollers = **new** Poller[getPollerThreadCount()];

**for** (**int** i=0; i<pollers.length; i++) {

pollers[i] = **new** Poller();

Thread pollerThread = **new** Thread(pollers[i], getName() + "-ClientPoller-"+i);

pollerThread.setPriority(threadPriority);

pollerThread.setDaemon(**true**);//声明为守护线程

pollerThread.start();

}

//启动接受线程

startAcceptorThreads();

}

**protected** **final** **void** **startAcceptorThreads**() {

**int** count = getAcceptorThreadCount();

acceptors = **new** Acceptor[count];

**for** (**int** i = 0; i < count; i++) {

acceptors[i] = createAcceptor();

String threadName = getName() + "-Acceptor-" + i;

acceptors[i].setThreadName(threadName);

Thread t = **new** Thread(acceptors[i], threadName);

t.setPriority(getAcceptorThreadPriority());

t.setDaemon(getDaemon());//设置是否为守护线程，默认为守护线程

t.start();

}

}

**private** **boolean** daemon = **true**;

**public** **void** **setDaemon**(**boolean** b) { daemon = b; }

**public** **boolean** **getDaemon**() { **return** daemon; }

如上代码也就是说默认情况下接受线程和处理线程都是守护线程，这意味着当 Tomact 收到 shutdown 命令后 Tomact 进程会马上消亡，而不会等处理线程处理完当前的请求。

**注：**如果你想在主线程结束后 JVM 进程马上结束，那么创建线程的时候可以设置线程为守护线程，否则如果希望主线程结束后子线程继续工作，等子线程结束后在让 JVM 进程结束那么就设置子线程为用户线程。

### 总结

本 Chat 作为 Java 并发编程之美系列的开篇，讲解了多线程并发编程线程有关基础知识，有了这些基础，为后面研究并发编程的高级知识打下基础。

# Java 并发编程之美：并发编程基础晋级篇

借用 Java 并发编程实践中的话；编写正确的程序并不容易，而编写正常的并发程序就更难了;相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的并发编程必备基础晋级篇，通过通俗易懂的方式来和大家聊聊多线程并发编程中涉及到的高级基础知识（建议先阅读[《Java 并发编程之美：线程相关的基础知识》](http://gitbook.cn/gitchat/activity/5aa4d205c2ff6f2e120891dd)），具体内容如下：

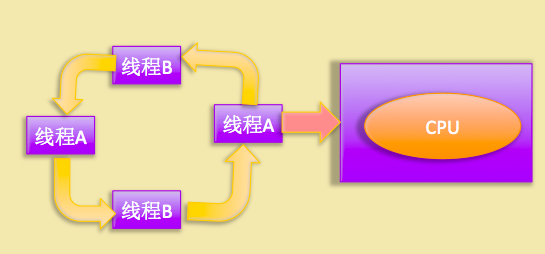
* 什么是多线程并发和并行。
* 什么是线程安全问题。
* 什么是共享变量的内存可见性问题。
* 什么是 Java 中原子性操作。
* 什么是 Java 中的 CAS 操作，AtomicLong 实现原理
* 什么是 Java 指令重排序。
* Java 中 Synchronized 关键字的内存语义是什么。
* Java 中 Volatile 关键字的内存语义是什么。
* 什么是伪共享，为何会出现，以及如何避免。
* 什么是可重入锁、乐观锁、悲观锁、公平锁、非公平锁、独占锁、共享锁。

### 多线程并发与并行

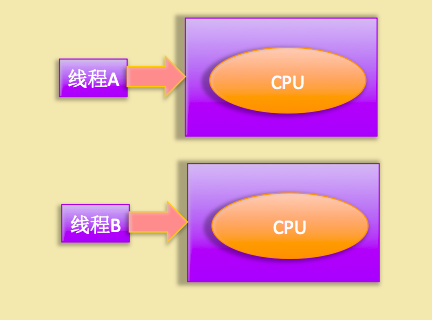
首先要澄清并发和并行的概念，并发是指同一个时间段内多个任务同时都在执行，并且都没有执行结束；而并行是说在单位时间内多个任务同时在执行；并发任务强调在一个时间段内同时执行，而一个时间段有多个单位时间累积而成，所以说并发的多个任务在单位时间内不一定同时在执行。

在单个 CPU 的时代多个任务同时运行都是并发，这是因为 CPU 同时只能执行一个任务，单个 CPU 时代多任务是共享一个 CPU 的，当一个任务占用 CPU 运行时候，其它任务就会被挂起，当占用 CPU 的任务时间片用完后，会把 CPU 让给其它任务来使用，所以在单 CPU 时代多线程编程的意义不大，并且线程间频繁的上下文切换还会带来开销。

如下图单个 CPU 上运行两个线程，可知线程 A 和 B 是轮流使用 CPU 进行任务处理的，也就是同时 CPU 只在执行一个线程上面的任务，当前线程 A 的时间片用完后会进行线程上下文切换，也就是保存当前线程的执行线程，然后切换线程 B 占用 CPU 运行任务。



如下图双 CPU 时候，线程 A 和线程 B 在自己的 CPU 上执行任务，实现了真正的并行运行。

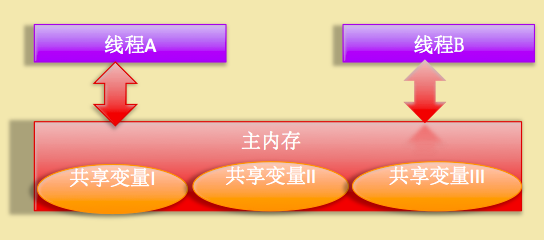


而在多线程编程实践中线程的个数往往多于 CPU 的个数，所以平时都是称多线程并发编程而不是多线程并行编程。

### 线程安全问题

谈到线程安全问题不得不先说说什么是共享资源，所谓共享资源是说多个线程都可以去访问的资源。

线程安全问题是指当多个线程同时读写一个共享资源并且没有任何同步措施的时候，导致脏数据或者其它不可预见的结果的问题。



如上图，线程 A 和线程 B 可以同时去操作主内存中的共享变量，是不是说多个线程共享了资源，都会产生线程安全问题呢？答案是否定的，如果多个线程都是只读取共享资源，而不去修改，那么就不会存在线程安全问题。

只有当至少一个线程修改共享资源时候才会存在线程安全问题。最典型的就是计数器类的实现，计数 count 本身是一个共享变量，多个线程可以对其进行增加一，如果不使用同步的话，由于递增操作是获取 -> 加1 -> 保存三步操作，所以可能导致导致计数不准确，如下表：

IMG_259

假如当前 count=0，t1 时刻线程 A 读取了 count 值到本地变量 countA。

然后 t2 时刻递增 countA 值为1，同时线程 B 读取 count 的值0放到本地变量 countB 值为0（因为 countA 还没有写入主内存）。

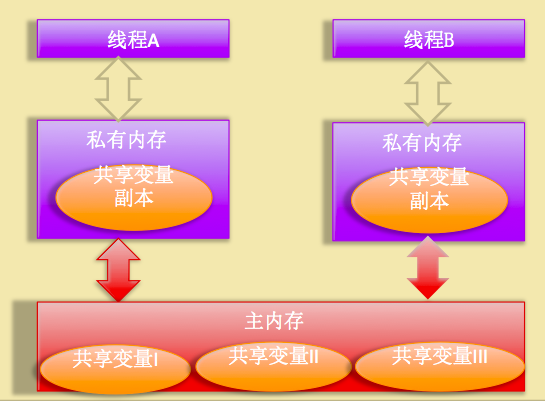
t3 时刻线程 A 才把 countA 为1的值写入主内存，至此线程 A 一次计数完毕，同时线程 B 递增 CountB 值为1。

t4 时刻线程 B 把 countB 值1写入内存，至此线程 B 一次计数完毕。

先不考虑内存可见性问题，明明是两次计数哇，为啥最后结果还是1而不是2呢？其实这就是共享变量的线程安全问题。那么如何解决？这就需要在线程访问共享变量时候进行适当的同步，Java 中首屈一指的是使用关键字 Synchronized 进行同步，这个下面会有具体介绍。

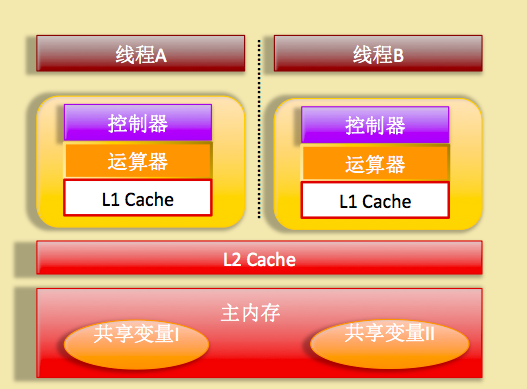
### 共享变量的内存可见性问题

要谈内存可见性首先需要介绍下 Java 中多线程下处理共享变量时候的内存模型。



如上图，Jav a内存模型规定了所有的变量都存放在主内存中，当线程使用变量时候都是把主内存里面的变量拷贝到了自己的工作空间或者叫做工作内存。

Java 内存模型是个抽象的概念，那么在实际实现中什么是线程的工作内存呢？



如上图是双核 CPU 系统架构，每核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辑运算，并且有自己的一级缓存，并且有些架构里面双核还有个共享的二级缓存。

那么 对应 Java 内存模型里面的工作内存，在实现上这里是指 L1 或者 L2 缓存或者 CPU 的寄存器。

假如线程 A 和 B 同时去处理一个共享变量，会出现什么情况呢？

使用上图 CPU 架构，假设线程 A和 B 使用不同 CPU 进行去修改共享变量 X，假设 X 的初始化为0，并且当前两级 Cache 都为空的情况，具体看下面分析:

* 假设线程 A 首先获取共享变量 X 的值，由于两级 Cache 都没有命中，所以到主内存加载了 X=0，然后会把 X=0 的值缓存到两级缓存，假设线程 A 修改 X 的值为1，然后写入到两级 Cache，并且刷新到主内存（注：如果没刷新会主内存也会存在内存不可见问题）。这时候线程 A 所在的 CPU 的两级 Cache 内和主内存里面 X 的值都是1；
* 然后假设线程 B 这时候获取 X 的值，首先一级缓存没有命中，然后看二级缓存，二级缓存命中了，所以返回 X=1；然后线程 B 修改 X 的值为2；然后存放到线程2所在的一级 Cache 和共享二级 Cache，最后更新主内存值为2；
* 然后假设线程 A 这次又需要修改 X 的值，获取时候一级缓存命中获取 X=1，到这里问题就出现了，明明线程 B 已经把 X 的值修改为了2，为啥线程 A 获取的还是1呢？这就是共享变量的内存不可见问题，也就是线程 B 写入的值对线程 A 不可见。

那么对于共享变量内存不可见问题如何解决呢？Java 中首屈一指的 Synchronized 和 Volatile 关键字就可以解决这个问题，下面会有讲解。

### Java 中 Synchronized 关键字

Synchronized 块是 Java 提供的一种原子性内置锁，Java 中每个对象都可以当做一个同步锁的功能来使用，这些 Java 内置的使用者看不到的锁被称为内部锁，也叫做监视器锁。

线程在进入 Synchronized 代码块前会自动尝试获取内部锁，如果这时候内部锁没有被其他线程占有，则当前线程就获取到了内部锁，这时候其它企图访问该代码块的线程会被阻塞挂起。

拿到内部锁的线程会在正常退出同步代码块或者异常抛出后或者同步块内调用了该内置锁资源的 wait 系列方法时候释放该内置锁；内置锁是排它锁，也就是当一个线程获取这个锁后，其它线程必须等待该线程释放锁才能获取该锁。

上一节讲了多线程并发修改共享变量时候会存在内存不可见的问题，究其原因是因为 Java 内存模型中线程操作共享变量时候会从自己的工作内存中获取而不是从主内存获取或者线程写入到本地内存的变量没有被刷新会主内存。

下面讲解下 Synchronized 的一个内存语义，这个内存语义就可以解决共享变量内存不可见性问题。

线程进入 Synchronized 块的语义是会把在 Synchronized 块内使用到的变量从线程的工作内存中清除，在 Synchronized 块内使用该变量时候就不会从线程的工作内存中获取了，而是直接从主内存中获取；退出 Synchronized 块的内存语义是会把 Synchronized 块内对共享变量的修改刷新到主内存。对应上面一节讲解的假如线程在 Synchronized 块内获取变量 X 的值，那么线程首先会清空所在的 CPU 的缓存，然后从主内存获取变量 X 的值；当线程修改了变量的值后会把修改的值刷新回主内存。

其实这也是加锁和释放锁的语义，当获取锁后会清空本地内存中后面将会用到的共享变量，在使用这些共享变量的时候会从主内存进行加载；在释放锁时候会刷新本地内存中修改的共享变量到主内存。

除了可以解决共享变量内存可见性问题外，Synchronized 经常被用来实现原子性操作，另外注意，Synchronized 关键字会引起线程上下文切换和线程调度的开销。

### Java 中 Volatile 关键字

上面介绍了使用锁的方式可以解决共享变量内存可见性问题，但是使用锁太重，因为它会引起线程上下文的切换开销，对于解决内存可见性问题，Java 还提供了一种弱形式的同步，也就是使用了 volatile 关键字。

一旦一个变量被 volatile 修饰了，当线程获取这个变量值的时候会首先清空线程工作内存中该变量的值，然后从主内存获取该变量的值；当线程写入被 volatile 修饰的变量的值的时候，首先会把修改后的值写入工作内存，然后会刷新到主内存。这就保证了对一个变量的更新对其它线程马上可见。

下面看一个使用 volatile 关键字解决内存不可见性的一个例子，如下代码的共享变量 value 是线程不安全的，因为它没有进行适当同步措施。

**public** **class** **ThreadNotSafeInteger** {

**private** **int** **value**;

**public** **int** **get**() {

**return** **value**;

}

**public** **void** **set**(**int** **value**) {

**this**.**value** = **value**;

}

}

首先看下使用 synchronized 关键字进行同步方式如下：

**public** **class** **ThreadSafeInteger** {

**private** **int** **value**;

**public** synchronized **int** **get**() {

**return** **value**;

}

**public** synchronized **void** **set**(**int** **value**) {

**this**.**value** = **value**;

}

}

然后看下使用 volatile 进行同步如下：

**public** **class** **ThreadSafeInteger** {

**private** **volatile** **int** **value**;

**public** **int** **get**() {

**return** **value**;

}

**public** **void** **set**(**int** **value**) {

**this**.**value** = **value**;

}

}

这里使用 synchronized 和使用 volatile 是等价的，都解决了共享变量 value 的内存不可见性问题；但是前者是独占锁，同时只能有一个线程调用 get() 方法，其它调用线程会被阻塞；并且会存在线程上下文切换和线程重新调度的开销；而后者是非阻塞算法，不会造成线程上下文切换的开销。

这里使用 synchronized 和使用 volatile 是等价的，但是并不是所有情况下都是等价的，这是因为 volatile 虽然提供了可见性保证，但是并没有保证操作的原子性。

那么一般什么时候才使用 volatile 关键字修饰变量呢？

* 当写入变量值时候不依赖变量的当前值。因为如果依赖当前值则是获取 -> 计算 -> 写入操作，而这三步操作不是原子性的，而 volatile 不保证原子性。
* 读写变量值时候没有进行加锁。因为加锁本身已经保证了内存可见性，这时候不需要把变量声明为 volatile。

另外变量被声明为 volatile 还可以避免重排序的发生，这个后面会讲到。

### Java 中原子性操作

所谓原子性操作是指当执行一系列操作时候，这些操作那么全部被执行，那么全部不被执行，不存在只执行其中一部分的情况。

在设计计数器时候一般都是先读取当前值，然后+1，然后更新，这个过程是读 -> 改 -> 写的过程，如果不能保证这个过程是原子性，那么就会出现线程安全问题。如下代码是线程不安全的，因为不能保证 ++value 是原子性操作。

**public** **class** **ThreadNotSafeCount** {

**private** Long **value**;

**public** Long **getCount**() {

**return** **value**;

}

**public** **void** **inc**() {

++**value**;

}

}

通过使用 Javap -c 查看汇编代码如下：

**public** **void** **inc**();

Code:

0: aload\_0

1: dup

2: getfield #2 // Field value:J

5: lconst\_1

6: ladd

7: putfield #2 // Field value:J

10: **return**

可知简单的 ++value有 2，5，6，7 组成，其中2是获取当前 value 的值并放入栈顶，5是把常量1放入栈顶，6是把当前栈顶中2个值相加并把结果放入栈顶，7则是把栈顶结果赋值会 value 变量，可知 Java 中简单的一句 ++value 转换为汇编后就不具有原子性了。

那么如何才能保证多个操作完成原子性呢，最简单的是使用 Synchronized 进行同步，修改代码如下：

**public** **class** **ThreadSafeCount** {

**private** Long **value**;

**public** synchronized Long **getCount**() {

**return** **value**;

}

**public** synchronized **void** **inc**() {

++**value**;

}

}

使用 Synchronized 的确可以实现线程安全，即实现内存可见性和同步，但是 Synchronized 是独占锁，同时只有一个线程可以调用 getCount 方法，其他没有获取内部锁的线程会被阻塞掉；而这里 getCount 方法只是读操作，多个线程同时调用不会存在线程安全问题，但是加了关键字 Synchronized 后同时就只能有一个线程可以调用了，这显然大大降低了并发性。

也许你会问既然是只读操作那么为何不去掉 getCount 方法上的 Synchronized 关键字呢？其实是不能去掉的，别忘了这里要靠 Synchronized 的内存语义来实现 value 的内存可见性。

那么有没有更好的实现呢？答案是肯定的，下面会讲到的内部使用非阻塞 CAS 算法实现的原子性操作类 AtomicLong 就是不错选择。

### Java 中的 CAS 操作和 AtomicLong 实现原理

#### CAS 来源

在 Java 中锁在并发处理中占据了一席之地，但是使用锁不好的地方是当一个线程没有获取到锁后会被阻塞挂起，这会导致线程上下文的切换和重新调度的开销。

Java 中提供了非阻塞的 volatile 关键字来解决共享变量的可见性问题，这在一定程度上弥补了锁所在带来的开销，但是 volatile 只能保证共享变量的可见性问题，但是还是不能解决例如读 -> 改 -> 写等的原子性问题。

CAS 即 Compare And Swap，是 JDK 提供的非阻塞原子性操作，它通过硬件保证了比较-更新操作的原子性，JDK 里面的 Unsafe 类提供了一些列的 compareAndSwap\* 方法，下面以 compareAndSwapLong 为例进行简单介绍。

* boolean compareAndSwapLong(Object obj,long valueOffset,long expect, long update)方法。

compareAndSwap 的意思也就是比较并交换，CAS 有四个操作数分别为：对象内存位置，对象中的变量的偏移量，变量预期值 expect，新的值 update。

操作含义是如果对象 obj 中内存偏移量为 valueOffset 位置的变量值为 expect 则使用新的值 update 替换旧的值 expect。这个是处理器提供的一个原子性指令。

#### AtomicLong 的原理

并发包中原子性操作类都有 AtomicInteger，AtomicLong，AtomicBoolean，原理类似，本节讲解下 AtomicLong 类。AtomicLong 是原子性递增或者递减类，其内部使用 Unsafe 来实现，下面看下代码：

**public** **class** **AtomicLong** **extends** **Number** **implements** **java**.**io**.**Serializable** {

**private** **static** **final** **long** serialVersionUID = 1927816293512124184L;

// （1）获取Unsafe实例

**private** **static** **final** Unsafe unsafe = Unsafe.getUnsafe();

//（2）存放变量value的偏移量

**private** **static** **final** **long** valueOffset;

//（3）判断JVM是否支持Long类型无锁CAS

**static** **final** **boolean** VM\_SUPPORTS\_LONG\_CAS = VMSupportsCS8();

**private** **static** **native** **boolean** **VMSupportsCS8**();

**static** {

**try** {

//（4）获取value在AtomicLong中偏移量

valueOffset = unsafe.objectFieldOffset

(AtomicLong.class.getDeclaredField("value"));

} **catch** (Exception ex) { **throw** **new** Error(ex); }

}

//（5）实际变量值

**private** **volatile** **long** value;

**public** **AtomicLong**(**long** initialValue) {

value = initialValue;

}

....

}

* 代码（1）创建了通过 Unsafe.getUnsafe（）方式获取到 Unsafe 类实例，这里你可能会疑问为何这里能通过 Unsafe.getUnsafe() 方式获取到 Unsafe 类实例？其实这是因为 AtomicLong 类也是在 rt.jar 包里面，AtomicLong 类的加载就是通过 BootStarp 类加载器进行加载的（关于 Unsafe 后面高级篇会具体讲解，这里先了解）
* 代码（5）中 value 声明为 volatile 是为了多线程下保证内存可见性，value 是具体存放计数的变量。
* 代码（2）（4）获取 value 变量在 AtomicLong 类中偏移量。

下面重点看下 AtomicLong 中主要函数：

* 递增和递减操作代码。

//（6）调用unsafe方法，原子性设置value值为原始值+1，返回值为递增后的值**public** final **long** **incrementAndGet**() {

**return** **unsafe**.getAndAddLong(**this**, valueOffset, 1L) + 1L;

}

//（7)调用unsafe方法，原子性设置value值为原始值-1，返回值为递减之后的值**public** final **long** **decrementAndGet**() {

**return** **unsafe**.getAndAddLong(**this**, valueOffset, -1L) - 1L;

}

//(8)调用unsafe方法，原子性设置value值为原始值+1,返回值为原始值**public** final **long** **getAndIncrement**() {

**return** **unsafe**.getAndAddLong(**this**, valueOffset, 1L);

}

//(9)调用unsafe方法，原子性设置value值为原始值-1，返回值为原始值**public** final **long** **getAndDecrement**() {

**return** **unsafe**.getAndAddLong(**this**, valueOffset, -1L);

}

如上代码内部都是调用 Unsafe 的 getAndAddLong 方法实现，这个函数是个原子性操作，这里第一个参数是 AtomicLong 实例的引用，第二个参数是 value 变量在 AtomicLong 中的偏移值，第三个参数是要设置第二个变量的值。

其中 getAndIncrement 方法在 JDK 7 的实现逻辑为：

**public** final **long** **getAndIncrement**() {

**while** (true) {

**long** current = **get**();

**long** next = current + 1;

**if** (compareAndSet(current, next))

**return** current;

}

}

如上代码可知每个线程是先拿到变量的当前值（由于是 value 是 volatile 变量所以这里拿到的是最新的值），然后在工作内存对其进行增一操作，然后使用 CAS 修改变量的值，如果设置失败，则循环继续尝试，直到设置成功。

而 JDK 8 逻辑为：

**public** final **long** **getAndIncrement**() {

**return** **unsafe**.getAndAddLong(**this**, valueOffset, 1L);

}

其中JDK8中unsafe.getAndAddLong代码为：

**public** **final** **long** **getAndAddLong**(Object paramObject, **long** paramLong1, **long** paramLong2)

{

**long** l;

**do**

{

l = getLongVolatile(paramObject, paramLong1);

} **while** (!compareAndSwapLong(paramObject, paramLong1, l, l + paramLong2));

**return** l;

}

可知 JDK 7 的 AtomicLong 中的循环逻辑已经被 JDK 8 的原子操作类 UNsafe 内置了，之所以内置应该是考虑到这种函数会在其它地方也会用到，内置可以提高复用性。

* boolean compareAndSet(long expect, long update)方法

**public** **final** **boolean** **compareAndSet**(**long** expect, **long** update) {

**return** unsafe.compareAndSwapLong(**this**, valueOffset, expect, update);

}

如上代码可知道内部还是调用了 unsafe.compareAndSwapLong 方法。如果原子变量中 value 的值等于 expect 则使用 update 值更新该值并返回 true，否者返回 false。

下面通过一个多线程使用 AtomicLong 统计0的个数的例子来加深对 AtomicLong 的理解：

/\*\*

统计0的个数

\*/**public** **class** **Atomic**

{

//(10)创建Long型原子计数器

**private** **static** AtomicLong atomicLong = **new** AtomicLong();

//(11)创建数据源

**private** **static** Integer[] arrayOne = **new** Integer[]{0,1,2,3,0,5,6,0,56,0};

**private** **static** Integer[] arrayTwo = **new** Integer[]{10,1,2,3,0,5,6,0,56,0};

**public** **static** **void** **main**( String[] args ) throws InterruptedException

{

//（12）线程one统计数组arrayOne中0的个数

Thread threadOne = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**int** size = arrayOne.length;

**for**(**int** i=0;i<size;++i){

**if**(arrayOne[i].intValue() == 0){

atomicLong.incrementAndGet();

}

}

}

});

//（13）线程two统计数组arrayTwo中0的个数

Thread threadTwo = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**int** size = arrayTwo.length;

**for**(**int** i=0;i<size;++i){

**if**(arrayTwo[i].intValue() == 0){

atomicLong.incrementAndGet();

}

}

}

});

//(14)启动子线程

threadOne.start();

threadTwo.start();

//(15)等待线程执行完毕

threadOne.**join**();

threadTwo.**join**();

System.**out**.println("count 0:" + atomicLong.**get**());

}

}

输出结果：count 0:7。

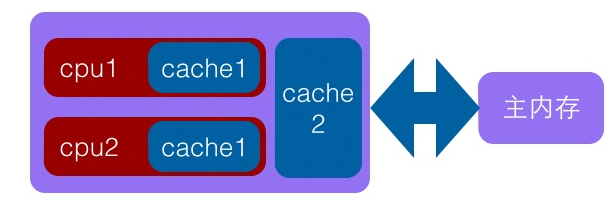
如上代码两个线程各自统计自己所在数据中0的个数，每当找到一个0就会调用 AtomicLong 的原子性递增方法。

**注：**在没有原子类的情况下例如最开始一节中自己做计数器的话，需要使用一定的同步措施，比如使用 Synchronized 关键字等，但是这些都是阻塞算法，对性能有一定损耗，而本节介绍的这些原子操作类都是使用 CAS 非阻塞算法，性能会更好。但是在高并发情况下 AtomicLong 还是会存在性能问题，后期高级篇会讲到 JDK 8 中提供了一个在高并发下性能更好的 LongAdder 类。

### 伪共享

#### 什么是伪共享

计算机系统中为了解决主内存与 CPU 运行速度的差距，在 CPU 与主内存之间添加了一级或者多级高速缓冲存储器（Cache），这个 Cache 一般是集成到 CPU 内部的，所以也叫 CPU Cache，如下图是两级 Cache 结构：

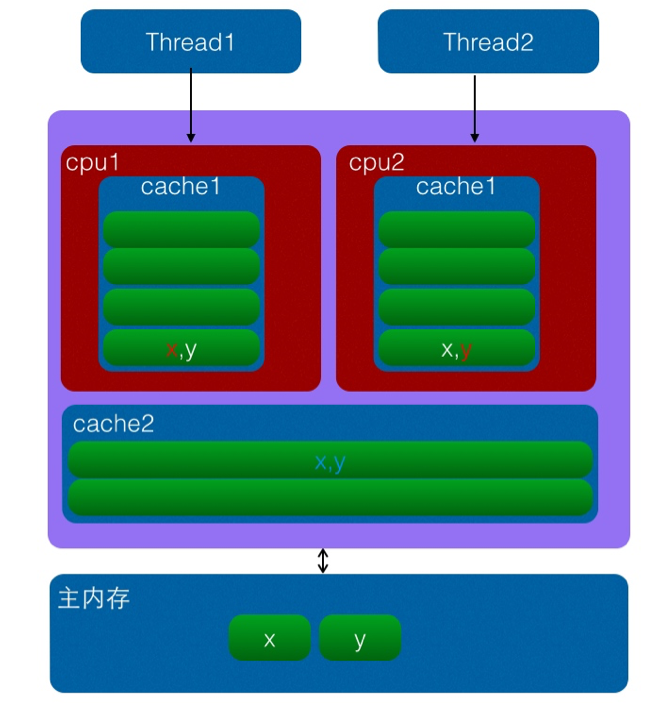


Cache 内部是按行存储的，其中每一行称为一个 Cache 行，Cache 行是 Cache 与主内存进行数据交换的单位，Cache 行的大小一般为2的幂次数字节。



当 CPU 访问某一个变量时候，首先会去看 CPU Cache 内是否有该变量，如果有则直接从中获取，否者就去主内存里面获取该变量，然后把该变量所在内存区域的一个 Cache 行大小的内存拷贝到 Cache（Cache 行是 Cache 与主内存进行数据交换的单位）。

由于存放到 Cache 行的的是内存块而不是单个变量，所以可能会把多个变量存放到了一个 Cache 行。当多个线程同时修改一个缓存行里面的多个变量时候，由于同时只能有一个线程操作缓存行，所以相比每个变量放到一个缓存行性能会有所下降，这就是伪共享。



如上图变量 x，y 同时被放到了 CPU 的一级和二级缓存，当线程1使用 CPU 1对变量 x 进行更新时候，首先会修改 CPU 1 的一级缓存变量 x 所在缓存行，这时候缓存一致性协议会导致 CPU 2 中变量 x 对应的缓存行失效。

那么线程2写入变量 x 的时候就只能去二级缓存去查找，这就破坏了一级缓存，而一级缓存比二级缓存更快，这里也说明了多个线程不可能同时去修改自己所使用的 CPU 中缓存行中相同缓存行里面的变量。更坏的情况下如果 CPU 只有一级缓存，那么会导致频繁的直接访问主内存。

#### 为何会出现伪共享

伪共享的产生是因为多个变量被放入了一个缓存行，并且多个线程同时去写入缓存行中不同变量。那么为何多个变量会被放入一个缓存行那。其实是因为 Cache 与内存交换数据的单位就是 Cache 行，当 CPU 要访问的变量没有在 Cache 命中时候，根据程序运行的局部性原理会把该变量在内存中大小为 Cache 行的内存放如缓存行。

**long** a;**long** b;**long** c;**long** d;

如上代码，声明了四个 long 变量，假设 Cache 行的大小为32个字节，那么当 CPU 访问变量 a 时候发现该变量没有在 Cache 命中，那么就会去主内存把变量 a 以及内存地址附近的 b、c、d 放入缓存行。

也就是地址连续的多个变量才有可能会被放到一个缓存行中，当创建数组时候，数组里面的多个元素就会被放入到同一个缓存行。那么单线程下多个变量放入缓存行对性能有影响？其实正常情况下单线程访问时候由于数组元素被放入到了一个或者多个 Cache 行对代码执行是有利的，因为数据都在缓存中，代码执行会更快，可以对比下面代码执行：

代码（1）：

**public** **class** **TestForContent** {

**static** final **int** LINE\_NUM = 1024;

**static** final **int** COLUM\_NUM = 1024;

**public** **static** **void** **main**(String[] args) {

**long** [][] **array** = **new** **long**[LINE\_NUM][COLUM\_NUM];

**long** startTime = System.currentTimeMillis();

**for**(**int** i =0;i<LINE\_NUM;++i){

**for**(**int** j=0;j<COLUM\_NUM;++j){

**array**[i][j] = i\*2+j;

}

}

**long** endTime = System.currentTimeMillis();

**long** cacheTime = endTime - startTime;

System.out.println("cache time:" + cacheTime);

}

}

代码（2）：

**public** **class** **TestForContent2** {

**static** final **int** LINE\_NUM = 1024;

**static** final **int** COLUM\_NUM = 1024;

**public** **static** **void** **main**(String[] args) {

**long** [][] **array** = **new** **long**[LINE\_NUM][COLUM\_NUM];

**long** startTime = System.currentTimeMillis();

**for**(**int** i =0;i<COLUM\_NUM;++i){

**for**(**int** j=0;j<LINE\_NUM;++j){

**array**[j][i] = i\*2+j;

}

}

**long** endTime = System.currentTimeMillis();

System.out.println("no cache time:" + (endTime - startTime));

}

}

我 Mac 电脑上执行代码（1）多次耗时均在10ms一下，执行代码（2）多次耗时均在10ms以上。

总的来说代码（1）比代码（2）执行的快，这是因为数组内数组元素之间内存地址是连续的，当访问数组第一个元素时候，会把第一个元素后续若干元素一块放入到 Cache 行，这样顺序访问数组元素时候会在 Cache 中直接命中，就不会去主内存读取，后续访问也是这样。

总结下也就是当顺序访问数组里面元素时候，如果当前元素在 Cache 没有命中，那么会从主内存一下子读取后续若干个元素到 Cache，也就是一次访问内存可以让后面多次直接在 Cache 命中。而代码（2）是跳跃式访问数组元素的，而不是顺序的，这破坏了程序访问的局部性原理，并且 Cache是有容量控制的，Cache 满了会根据一定淘汰算法替换 Cache 行，会导致从内存置换过来的 Cache 行的元素还没等到读取就被替换掉了。

所以单个线程下顺序修改一个 Cache 行中的多个变量，是充分利用了程序运行局部性原理，会加速程序的运行，而多线程下并发修改一个 Cache 行中的多个变量而就会进行竞争 Cache 行，降低程序运行性能。

#### 如何避免伪共享

JDK 8 之前一般都是通过字节填充的方式来避免，也就是创建一个变量的时候使用填充字段填充该变量所在的缓存行，这样就避免了多个变量存在同一个缓存行，如下代码：

**public** final **static** **class** **FilledLong** {

**public** **volatile** **long** **value** = 0L;

**public** **long** p1, p2, p3, p4, p5, p6;

}

假如 Cache 行为64个字节，那么我们在 FilledLong 类里面填充了6个 long 类型变量，每个 long 类型占用8个字节，加上 value 变量的8个字节总共56个字节，另外这里 FilledLong 是一个类对象，而类对象的字节码的对象头占用了8个字节，所以当 new 一个 FilledLong 对象时候实际会占用64个字节的内存，这个正好可以放入 Cache 的一个行。

在 JDK 8 中提供了一个 sun.misc.Contended 注解，用来解决伪共享问题，上面代码可以修改为如下：

@sun.misc.Contended

**public** **final** **static** **class** **FilledLong** {

**public** **volatile** **long** value = 0L;

}

上面是修饰类的，当然也可以修饰变量，比如 Thread 类中的使用：

/\*\* The current seed for a ThreadLocalRandom \*/

@**sun**.**misc**.**Contended**("**tlr**")

long threadLocalRandomSeed;

/\*\* Probe hash value; nonzero if threadLocalRandomSeed initialized \*/

@**sun**.**misc**.**Contended**("**tlr**")

int threadLocalRandomProbe;

/\*\* Secondary seed isolated from public ThreadLocalRandom sequence \*/

@**sun**.**misc**.**Contended**("**tlr**")

int threadLocalRandomSecondarySeed;

Thread 类里面这三个变量是在 ThreadLocalRandom（Chat：[《Java 并发编程之美：并发编程高级篇之一》](http://gitbook.cn/gitchat/activity/5ab99f3f2e38d007de69af9e)中对其进行了讲解）中为了实现高并发下高性能生成随机数时候使用的，这三个变量默认是初始化为0。

需要注意的是默认情况下 @Contended 注解只用到 Java 核心类，比如 rt 包下的类，如果需要在用户 classpath 下的类使用这个注解需要添加 JVM 参数：-XX:-RestrictContended，另外默认填充的宽度为128，如果你想要自定义宽度可以设置 -XX:ContendedPaddingWidth 参数。

**注：**本节讲述了伪共享如何产生，以及如何避免，并证明多线程下访问同一个 Cache 行的多个的变量时候才会出现伪共享，当单个线程访问一个 Cache 行里面的多个变量时候反而对程序运行起到加速作用。这里为后面高级篇讲解 LongAdder 的实现提供了基础。

### Java 中的指令重排序

Java 内存模型允许编译器和处理器对指令进行重排序以提高运行性能，并且重排序只会对不存在数据依赖性的指令进行重排序；在单线程下重排序可以保证最终执行的结果是与程序顺序执行的结果一致，但是在多线程下就会存在问题。

下面看一个例子

**int** a = 1;//(1)**int** b = 2;//(2)**int** c= a + b;//(3)

如上代码变量 c 的值依赖 a 和 b 的值，所以重排序后能够保证（3）的操作在（2）（1）之后，但是（1）（2）谁先执行就不一定了，这在单线程下不会存在问题，因为并不影响最终结果。

下面看一个多线程的例子：

**public** **static** **class** **ReadThread** **extends** **Thread** {

**public** **void** **run**() {

**while**(!Thread.currentThread().isInterrupted()){

**if**(ready){//(1)

System.out.println(num+num);//(2)

}

System.out.println("read thread....");

}

}

}

**public** **static** **class** **Writethread** **extends** **Thread** {

**public** **void** **run**() {

num = 2;//(3)

ready = **true**;//(4)

System.out.println("writeThread set over...");

}

}

**private** **static** **int** num =0;**private** **static** **boolean** ready = **false**;

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

ReadThread rt = **new** ReadThread();

rt.start();

Writethread wt = **new** Writethread();

wt.start();

Thread.sleep(10);

rt.interrupt();

System.out.println("main exit");

}

首先这段代码里面的变量没有声明为 volatile 也没有使用任何同步措施，所以多线程下存在共享变量内存可见性问题，这里先不谈内存可见性问题，因为通过把变量声明为 volatile 本身就可以避免指令重排序问题。

这里先看看指令重排序会造成什么影响，如上代码不考虑内存可见性问题的情况下 程序一定会输出4？答案是不一定，由于代码（1）（2）（3）（4）之间不存在依赖，所以写线程的代码（3）（4）可能被重排序为先执行（4）在执行（3），那么执行（4）后，读线程可能已经执行了（1）操作，并且在（3）执行前开始执行（2）操作，这时候打印结果为0而不是4。

这就是重排序在多线程下导致程序执行结果不是我们想要的了，这里使用 volatile 修饰 ready 可以避免重排序和内存可见性问题。

当写 volatile 变量时候，可以确保 volatile 写之前的操作不会被编译器重排序到 volatile 写之后。 当读 volatile 读变量时候，可以确保 volatile 读之后的操作不会被编译器重排序到 volatile 读之前。

### 锁的概述

#### 乐观锁与悲观锁

乐观锁和悲观锁是在数据库中使用的名词，本节这里也提下。

##### ****悲观锁****

悲观锁指对数据被外界修改持保守态度，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制，数据库中实现是对数据记录操作前给记录加排它锁。如果获取锁失败，则说明数据正在被其它线程修改，则等待或者抛出异常。如果加锁成功，则获取记录，对其修改，然后事务提交后释放排它锁。

使用悲观锁的一个常用的例子： select \* from 表 where .. for update;。

##### ****乐观锁****

乐观锁是相对悲观锁来说的，它认为数据一般情况下不会造成冲突，所以在访问记录前不会加排它锁，而是在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测。具体说是根据 update 返回的行数让用户决定如何去做。

例如： update 表 set comment='\*\*\*',status='operator',version=version+1 where version = 1 and id = 1;

乐观锁并不会使用数据库提供的锁机制，一般在表添加 version 字段或者使用业务状态来做。乐观锁直到提交的时候才去锁定，所以不会产生任何锁和死锁。

#### 公平锁与非公平锁

根据线程获取锁的抢占机制锁可以分为公平锁和非公平锁，公平锁表示线程获取锁的顺序是按照线程请求锁的时间长短来分决定的的，也就是最早获取锁的线程将最早获取到锁，也就是先来先得的 FIFO 顺序。而非公平锁则运行时候闯入，也就是先来不一定先得。

ReentrantLock 提供了公平和非公平锁的实现：

* 公平锁：ReentrantLock pairLock = new ReentrantLock(true);
* 非公平锁：ReentrantLock pairLock = new ReentrantLock(false);

如果构造函数不传递参数，则默认是非公平锁。

具体来说假设线程 A 已经持有了锁，这时候线程 B 请求该锁将会被挂起，当线程 A 释放锁后，假如当前有线程 C 也需要获取该锁，如果采用非公平锁方式，则根据线程调度策略线程 B 和 C 两者之一可能获取锁，这时候不需要任何其它干涉，如果使用公平锁则需要把 C 挂起，让 B 获取当前锁。

在没有公平性需求的前提下尽量使用非公平锁，因为公平锁会带来性能开销。

#### 独占锁与共享锁

根据锁只能被单个线程持有还是能被多个线程共同持有，锁分为独占锁和共享锁。

独占锁保证任何时候都只有一个线程能得到锁，ReentrantLock 就是以独占方式实现的。共享锁则同时有多个线程可以持有，例如 ReadWriteLock 读写锁，它允许一个资源可以被多线程同时进行读操作。

独占锁是一种悲观锁，每次访问资源都先加上互斥锁，这限制了并发性，因为读操作并不会影响数据一致性，而独占锁只允许同时一个线程读取数据，其它线程必须等待当前线程释放锁才能进行读取。

共享锁则是一种乐观锁，它放宽了加锁的条件，允许多个线程同时进行读操作。

#### 什么是可重入锁

当一个线程要获取一个被其它线程持有的独占锁时候，该线程会被阻塞，那么当一个线程再次获取它自己已经获取的锁时候是否会被阻塞那?如果不被阻塞，那么我们说该锁是可重入的，也就是只要该线程获取了该锁，那么可以无限制次数（高级篇我们会知道严格来说是有限次数）进入被该锁锁住的代码。

下面看一个例子看看什么情况下会用可重入锁。

**public** **class** **Hello**{

**public** Synchronized **void** **helloA**(){

System.**out**.println("hello");

}

**public** Synchronized **void** **helloB**(){

System.**out**.println("hello B");

helloA();

}

}

如上面代码当调用 helloB 函数前会先获取内置锁，然后打印输出，然后调用 helloA 方法，调用前会先去获取内置锁，如果内置锁不是可重入的那么该调用就会导致死锁了，因为线程持有并等待了锁导致调用 helloA 时候永远不会获取到锁。

实际上 synchronized 内部锁是可重入锁，可重入锁的原理是在锁内部维护了一个线程标示，用来标示该锁目前被那个线程占用，然后关联一个计数器。一开始计数器值为0，说明该锁没有被任何线程占用，当一个线程获取了该锁，计数器会变成1，其它线程在获取该锁时候发现锁的所有者不是自己就会被阻塞挂起。

但是当获取该锁的线程再次获取锁时候发现锁拥有者是自己，就会把计数器值+1， 当释放锁后计数器会-1，当计数器为0时候，锁里面的线程标示重置为 null，这时候阻塞的线程会获取被唤醒来竞争获取该锁。

### 总结

本章主要介绍了并发编程的基础知识，为后面高级篇讲解并发包源码提供了基础，通过图形结合讲述了为什么要使用多线程编程，多线程编程存在的线程安全问题，以及什么是内存可见性问题。然后讲解了 synchronized 和 volatile 关键字，并且强调了前者既保证了内存可见性同时也保证了原子性，而后者则主要做到了内存可见性，但是它们的内存语义还是很相似的，最后讲解的什么是 CAS 和线程间同步以及各种锁的概念，都为后面讲解 JUC 包源码奠定了基础。

# Java 并发编程之美：并发编程高级篇之一

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

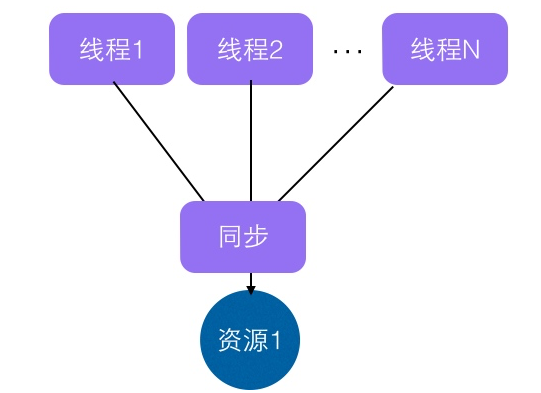
并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之一，主要讲解内容如下：（[建议先阅读《Java 并发编程之美：基础篇》](http://gitbook.cn/m/mazi/activity/5aafb17477918b6e8444b65f) ）

* ThreadLocal 的实现原理，ThreadLocal 作为变量的线程隔离方式，其内部是如何做的？
* InheritableThreadLocal 的实现原理，InheritableThreadLocal 是如何弥补 ThreadLocal 不支持继承的特性？
* JDK 并发包中 ThreadLocalRandom 类原理剖析，经常使用的随机数生成器 Random 类的原理是什么？及其局限性是什么？ThreadLocalRandom 是如何利用 ThreadLocal 的原理来解决 Random 的局限性？
* 最后 ThreadLocal 的一个使用场景，Spring 框架中 Scope 作用域 Bean 的实现原理。

### ThreadLocal 的实现原理

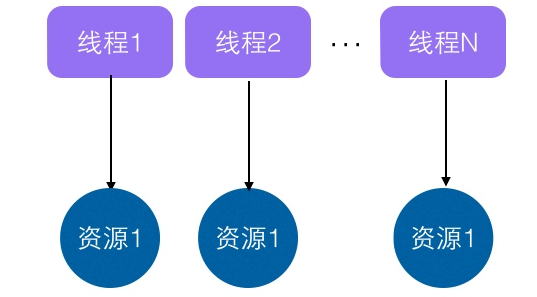
多线程访问同一个共享变量特别容易出现并发问题，特别是多个线程需要对一个共享变量进行写入时候，为了保证线程安全，一般需要使用者在访问共享变量的时候进行适当的同步，如下图：



同步的措施一般是加锁，这就需要使用者对锁也要有一定了解，这显然加重了使用者的负担。

那么有没有一个方式当创建一个变量时候，每个线程对其进行访问的时候访问的是自己线程的变量呢？其实 ThreadLocal 就可以做这个事情，虽然 ThreadLocal 的出现并不是为了解决上面的问题而出现的。

ThreadLocal 是在 JDK 包里面提供的，它提供了线程本地变量，也就是如果你创建了一个 ThreadLocal 变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作的自己本地内存里面的变量，从而避免了线程安全问题，创建一个 ThreadLocal 变量后每个线程会拷贝一个变量到自己本地内存，如下图：



#### ThreadLocal 简单使用

本节来看下 ThreadLocal 如何使用，从而加深理解，本例子开启了两个线程，每个线程内部设置了本地变量的值，然后调用 print 函数打印当前本地变量的值，如果打印后调用了本地变量的 remove 方法则会删除本地内存中的该变量，代码如下：

**public** **class** **ThreadLocalTest** {

//(1)打印函数

**static** **void** **print**(String str){

//1.1 打印当前线程本地内存中localVariable变量的值

System.**out**.println(str + ":" +localVariable.**get**());

//1.2 清除当前线程本地内存中localVariable变量

//localVariable.remove();

}

//(2) 创建ThreadLocal变量

**static** ThreadLocal<String> localVariable = **new** ThreadLocal<>();

**public** **static** **void** **main**(String[] args) {

//(3) 创建线程one

Thread threadOne = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

//3.1 设置线程one中本地变量localVariable的值

localVariable.**set**("threadOne local variable");

//3.2 调用打印函数

print("threadOne");

//3.3打印本地变量值

System.**out**.println("threadOne remove after" + ":" +localVariable.**get**());

}

});

//(4) 创建线程two

Thread threadTwo = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

//4.1 设置线程one中本地变量localVariable的值

localVariable.**set**("threadTwo local variable");

//4.2 调用打印函数

print("threadTwo");

//4.3打印本地变量值

System.**out**.println("threadTwo remove after" + ":" +localVariable.**get**());

}

});

//(5)启动线程

threadOne.start();

threadTwo.start();

}

运行结果：

threadOne:threadOne **local** variable

threadTwo:threadTwo **local** variable

threadOne remove after:threadOne **local** variable

threadTwo remove after:threadTwo **local** variable

* 代码（2）创建了一个 ThreadLocal 变量；
* 代码（3）、（4）分别创建了线程 one 和 two；
* 代码（5）启动了两个线程；
* 线程 one 中代码 3.1 通过 set 方法设置了 localVariable 的值，这个设置的其实是线程 one 本地内存中的一个拷贝，这个拷贝线程 two 是访问不了的。然后代码 3.2 调用了 print 函数，代码 1.1 通过 get 函数获取了当前线程（线程 one）本地内存中 localVariable 的值；
* 线程 two 执行类似线程 one。

解开代码 1.2 的注释后，再次运行，运行结果为：

threadOne:threadOne local variable

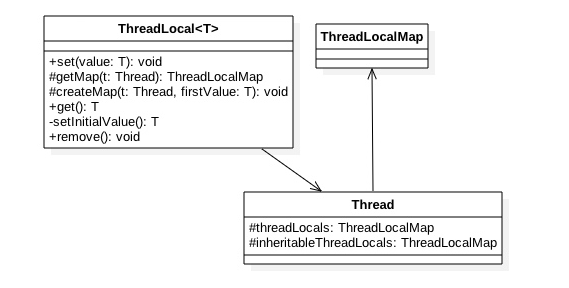
threadOne **remove** after:null

threadTwo:threadTwo local variable

threadTwo **remove** after:null

#### ThreadLocal 实现原理

首先看下 ThreadLocal 相关的类的类图结构。



如上类图可知 Thread 类中有一个 threadLocals 和 inheritableThreadLocals 都是 ThreadLocalMap 类型的变量，而 ThreadLocalMap 是一个定制化的 Hashmap，默认每个线程中这个两个变量都为 null，只有当前线程第一次调用了 ThreadLocal 的 set 或者 get 方法时候才会进行创建。

其实每个线程的本地变量不是存放到 ThreadLocal 实例里面的，而是存放到调用线程的 threadLocals 变量里面。也就是说 ThreadLocal 类型的本地变量是存放到具体的线程内存空间的。

ThreadLocal 就是一个工具壳，它通过 set 方法把 value 值放入调用线程的 threadLocals 里面存放起来，当调用线程调用它的 get 方法时候再从当前线程的 threadLocals变 量里面拿出来使用。

如果调用线程一直不终止，那么这个本地变量会一直存放到调用线程的 threadLocals 变量里面，所以当不需要使用本地变量时候可以通过调用 ThreadLocal 变量的 remove 方法，从当前线程的 threadLocals 里面删除该本地变量。另外 Thread 里面的 threadLocals 为何设计为 map 结构呢？很明显是因为每个线程里面可以关联多个 ThreadLocal 变量。

下面简单分析下 ThreadLocal 的 set，get，remove 方法的实现逻辑：

* void set(T value)

**public** **void** **set**(T **value**) {

//(1)获取当前线程

Thread t = Thread.currentThread();

//(2)当前线程作为key，去查找对应的线程变量，找到则设置

ThreadLocalMap map = getMap(t);

**if** (map != null)

map.**set**(**this**, **value**);

**else**

//(3)第一次调用则创建当前线程对应的HashMap

createMap(t, **value**);

}

如上代码（1）首先获取调用线程，然后使用当前线程作为参数调用了 getMap(t) 方法，getMap(Thread t) 代码如下：

ThreadLocalMap **getMap**(Thread t) {

**return** t.threadLocals;

}

可知 getMap(t) 所做的就是获取线程自己的变量 threadLocals，threadlocal 变量是绑定到了线程的成员变量里面。

如果 getMap(t) 返回不为空，则把 value 值设置进入到 threadLocals，也就是把当前变量值放入了当前线程的内存变量 threadLocals，threadLocals 是个 HashMap 结构，其中 key 就是当前 ThreadLocal 的实例对象引用，value 是通过 set 方法传递的值。

如果 getMap(t) 返回空那说明是第一次调用 set 方法，则创建当前线程的 threadLocals 变量，下面看 createMap(t, value) 里面做了啥呢？

**void** **createMap**(Thread t, T firstValue) {

t.threadLocals = **new** ThreadLocalMap(**this**, firstValue);

}

可知就是创建当前线程的 threadLocals 变量。

* T get()

**public** T **get**() {

//(4) 获取当前线程

Thread t = Thread.currentThread();

//(5)获取当前线程的threadLocals变量

ThreadLocalMap **map** = getMap(t);

//(6)如果threadLocals不为null，则返回对应本地变量值

**if** (**map** != null) {

ThreadLocalMap.Entry e = **map**.getEntry(**this**);

**if** (e != null) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

**return** result;

}

}

//(7)threadLocals为空则初始化当前线程的threadLocals成员变量

**return** setInitialValue();

}

如上代码（4）首先获取当前线程实例，如果当前线程的 threadLocals 变量不为 null 则直接返回当前线程绑定的本地变量。否者执行代码（7）进行初始化，setInitialValue() 的代码如下：

**private** T **setInitialValue**() {

//(8)初始化为null

T **value** = initialValue();

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

//(9)如果当前线程的threadLocals变量不为空

**if** (map != null)

map.**set**(**this**, **value**);

**else**

//(10)如果当前线程的threadLocals变量为空

createMap(t, **value**);

**return** **value**;

}

**protected** T **initialValue**() {

**return** null;

}

如上代码如果当前线程的 threadLocals 变量不为空，则设置当前线程的本地变量值为 null，否者调用 createMap 创建当前线程的 createMap 变量。

* void remove()

**public** **void** **remove**() {

ThreadLocalMap m = getMap(Thread.currentThread());

**if** (m != null)

m.**remove**(**this**);

}

如上代码，如果当前线程的 threadLocals 变量不为空，则删除当前线程中指定 ThreadLocal 实例的本地变量。

**注：**每个线程内部都有一个名字为 threadLocals 的成员变量，该变量类型为 HashMap，其中 key 为我们定义的 ThreadLocal 变量的 this 引用，value 则为我们 set 时候的值，每个线程的本地变量是存到线程自己的内存变量 threadLocals 里面的，如果当前线程一直不消失那么这些本地变量会一直存到，所以可能会造成内存泄露，所以使用完毕后要记得调用 ThreadLocal 的 remove 方法删除对应线程的 threadLocals 中的本地变量。

#### 子线程中获取不到父线程中设置的 ThreadLocal 变量的值

首先看个例子说明标题的意思：

**public** **class** **TestThreadLocal** {

//(1) 创建线程变量

**public** **static** ThreadLocal<String> threadLocal = **new** ThreadLocal<String>();

**public** **static** **void** **main**(String[] args) {

//(2) 设置线程变量

threadLocal.**set**("hello world");

//(3) 启动子线程

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

//(4)子线程输出线程变量的值

System.**out**.println("thread:" + threadLocal.**get**());

}

});

thread.start();

//(5)主线程输出线程变量值

System.**out**.println("main:" + threadLocal.**get**());

}

}

结果为：

**main**:hello **worldthread**:null

也就是说同一个 ThreadLocal 变量在父线程中设置值后，在子线程中是获取不到的。根据上节的介绍，这个应该是正常现象，因为子线程调用 get 方法时候当前线程为子线程，而调用 set 方法设置线程变量是 main 线程，两者是不同的线程，自然子线程访问时候返回 null，那么有办法让子线程访问到父线程中的值吗？答案是有。

### InheritableThreadLocal 原理

为了解决上节的问题 InheritableThreadLocal 应运而生，InheritableThreadLocal 继承自 ThreadLocal，提供了一个特性，就是子线程可以访问到父线程中设置的本地变量。

下面看下 InheritableThreadLocal 的代码：

**public** **class** **InheritableThreadLocal**<T> **extends** **ThreadLocal**<T> {

//(1)

**protected** T **childValue**(T parentValue) {

**return** parentValue;

}

//(2)

ThreadLocalMap **getMap**(Thread t) {

**return** t.inheritableThreadLocals;

}

//(3)

**void** **createMap**(Thread t, T firstValue) {

t.inheritableThreadLocals = **new** ThreadLocalMap(**this**, firstValue);

}

}

如上代码可知 InheritableThreadLocal 继承了 ThreadLocal，并重写了三个方法。

* 代码（3）可知 InheritableThreadLocal 重写了 createMap 方法，那么可知现在当第一次调用 set 方法时候创建的是当前线程的 inheritableThreadLocals 变量的实例而不再是 threadLocals。
* 代码（2）可知当调用 get 方法获取当前线程的内部 map 变量时候，获取的是 inheritableThreadLocals 而不再是 threadLocals。

综上可知在 InheritableThreadLocal 的世界里，线程中的变量 inheritableThreadLocals 替代了 threadLocals。

下面我们看下重写的代码（1）是何时被执行，以及如何实现的子线程可以访问父线程本地变量的。这个要从 Thread 创建的代码看起，Thread 的默认构造函数及 Thread.java 类的构造函数如下：

**public** **Thread**(Runnable target) {

init(null, target, "Thread-" + nextThreadNum(), 0);

}

**private** **void** **init**(ThreadGroup g, Runnable target, String name,

**long** stackSize, AccessControlContext acc) {

...

//(4)获取当前线程

Thread parent = currentThread();

...

//(5)如果父线程的inheritableThreadLocals变量不为null

**if** (parent.inheritableThreadLocals != null)

//(6)设置子线程中的inheritableThreadLocals变量

**this**.inheritableThreadLocals =

ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);

**this**.stackSize = stackSize;

tid = nextThreadID();

}

创建线程时候在构造函数里面会调用 init 方法，前面讲到了 inheritableThreadLocal 类 get，set 方法操作的是变量 inheritableThreadLocals，所以这里 inheritableThreadLocal 变量就不为 null，所以会执行代码（6），下面看下 createInheritedMap 代码：

**static** ThreadLocalMap **createInheritedMap**(ThreadLocalMap parentMap) {

**return** **new** ThreadLocalMap(parentMap);

}

可知 createInheritedMap 内部使用父线程的 inheritableThreadLocals 变量作为构造函数创建了一个新的 ThreadLocalMap 变量。然后赋值给了子线程的 inheritableThreadLocals 变量，那么下面看看 ThreadLocalMap 的构造函数里面做了什么：

**private** **ThreadLocalMap**(ThreadLocalMap parentMap) {

Entry[] parentTable = parentMap.table;

**int** len = parentTable.length;

setThreshold(len);

table = **new** Entry[len];

**for** (**int** j = 0; j < len; j++) {

Entry e = parentTable[j];

**if** (e != null) {

@SuppressWarnings("unchecked")

ThreadLocal<Object> key = (ThreadLocal<Object>) e.**get**();

**if** (key != null) {

//(7)调用重写的方法

Object **value** = key.childValue(e.**value**);//返回e.value

Entry c = **new** Entry(key, **value**);

**int** h = key.threadLocalHashCode & (len - 1);

**while** (table[h] != null)

h = nextIndex(h, len);

table[h] = c;

size++;

}

}

}

}

如上代码所做的事情就是把父线程的 inheritableThreadLocals 成员变量的值复制到新的 ThreadLocalMap 对象，其中代码（7）InheritableThreadLocal 类重写的代码（1）也映入眼帘了。

**总结：**InheritableThreadLocal 类通过重写代码（2）和（3）让本地变量保存到了具体线程的 inheritableThreadLocals 变量里面，线程通过 InheritableThreadLocal 类实例的 set 或者 get 方法设置变量时候就会创建当前线程的 inheritableThreadLocals 变量。当父线程创建子线程时候，构造函数里面会把父线程中 inheritableThreadLocals 变量里面的本地变量拷贝一份复制到子线程的 inheritableThreadLocals 变量里面。

把上节代码（1）修改为：

//(1) 创建线程变量

public **static** ThreadLocal<**String**> threadLocal = **new** InheritableThreadLocal<**String**>();

运行结果为：

**thread**:hello **worldmain**:hello **world**

可知现在可以从子线程中正常的获取到线程变量值了。

那么什么情况下需要子线程可以获取到父线程的 threadlocal 变量呢，情况还是蛮多的，比如存放用户登录信息的 threadlocal 变量，很有可能子线程中也需要使用用户登录信息，再比如一些中间件需要用统一的追踪 ID 把整个调用链路记录下来的情景。

### JDK 并发包中 ThreadLocalRandom 类原理剖析

ThreadLocalRandom 类是 JDK7 在 JUC 包下新增的随机数生成器，它解决了 Random 类在多线程下的不足。本节就来讲解下 JUC 下为何新增该类，以及该类的实现原理。

#### Random 类及其局限性

在 JDK7 之前包括现在，java.util.Random 应该是使用比较广泛的随机数生成工具类，另外 java.lang.Math 中的随机数生成也是使用的 java.util.Random 的实例。下面先看看 java.util.Random 的使用：

**public** **class** **RandomTest** {

**public** **static** **void** **main**(String[] args) {

//(1)创建一个默认种子的随机数生成器

Random random = **new** Random();

//(2)输出10个在0-5（包含0，不包含5）之间的随机数

**for** (**int** i = 0; i < 10; ++i) {

System.**out**.println(random.nextInt(5));

}

}

}

* 代码（1）创建一个默认随机数生成器，使用默认的种子。
* 代码（2）输出输出10个在0-5（包含0，不包含5）之间的随机数。

这里提下随机数的生成需要一个默认的种子，这个种子其实是一个 long 类型的数字，这个种子要么在 Random 的时候通过构造函数指定，那么默认构造函数内部会生成一个默认的值，有了默认的种子后，如何生成随机数呢？

**public** **int** **nextInt**(**int** bound) {

//(3)参数检查

**if** (bound <= 0)

**throw** **new** IllegalArgumentException(BadBound);

//(4)根据老的种子生成新的种子

**int** r = next(31);

//(5)根据新的种子计算随机数

...

**return** r;

}

如上代码可知新的随机数的生成需要两个步骤：

* 首先需要根据老的种子生成新的种子。
* 然后根据新的种子来计算新的随机数。

其中步骤（4）我们可以抽象为 seed=f(seed)，其中 f 是一个固定的函数，比如 seed= f(seed)=a\*seed+b;，步骤（5）也可以抽象为 g(seed,bound)，其中 g 是一个固定的函数，比如 g(seed,bound)=(int)((bound \* (long)seed) >> 31);。在单线程情况下每次调用 nextInt 都是根据老的种子计算出来新的种子，这是可以保证随机数产生的随机性的。但是在多线程下多个线程可能都拿同一个老的种子去执行步骤（4）计算新的种子，这会导致多个线程产生的新种子是一样的，由于步骤（5）算法是固定的，所以会导致多个线程产生相同的随机值，这并不是我们想要的。

所以需要保证步骤（4）的原子性，也就是说多个线程在根据同一个老种子计算新种子时候，第一个线程的新种子计算出来后，第二个线程要丢弃自己老的种子，要使用第一个线程的新种子来计算自己的新种子，依次类推，只有保证了这个，才能保证多线程下产生的随机数是随机的。Random 函数使用一个原子变量达到了这个效果，在创建 Random 对象时候初始化的种子就保存到了种子原子变量里面，下面看下 next() 代码：

**protected** **int** **next**(**int** bits) {

**long** oldseed, nextseed;

AtomicLong seed = **this**.seed;

**do** {

//(6)

oldseed = seed.**get**();

//(7)

nextseed = (oldseed \* multiplier + addend) & mask;

//(8)

} **while** (!seed.compareAndSet(oldseed, nextseed));

//(9)

**return** (**int**)(nextseed >>> (48 - bits));

}

* 代码（6）获取当前原子变量种子的值；
* 代码（7）根据当前种子值计算新的种子；
* 代码（8）使用 CAS 操作，使用新的种子去更新老的种子，多线程下可能多个线程都同时执行到了代码（6），那么可能多个线程都拿到的当前种子的值是同一个，然后执行步骤（7）计算的新种子也都是一样的，但是步骤（8）的 CAS 操作会保证只有一个线程可以更新老的种子为新的，失败的线程会通过循环重新获取更新后的种子作为当前种子去计算老的种子，可见这里解决了上面提到的问题，也就保证了随机数的随机性。
* 代码（9）则使用固定算法根据新的种子计算随机数。

**总结：**每个 Random 实例里面有一个原子性的种子变量用来记录当前的种子的值，当要生成新的随机数时候要根据当前种子计算新的种子并更新回原子变量。多线程下使用单个 Random 实例生成随机数时候，多个线程同时计算新的种子时候会竞争同一个原子变量的更新操作，由于原子变量的更新是 CAS 操作，同时只有一个线程会成功，所以会造成大量线程进行自旋重试，这是会降低并发性能的，所以 ThreadLocalRandom 应运而生。

#### ThreadLocalRandom 类

为了解决多线程高并发下 Random 的缺陷，JUC 包下新增了 ThreadLocalRandom 类，下面首先看下它如何使用：

**public** **class** **RandomTest** {

**public** **static** **void** **main**(String[] args) {

//(10)获取一个随机数生成器

ThreadLocalRandom random = ThreadLocalRandom.current();

//(11)输出10个在0-5（包含0，不包含5）之间的随机数

**for** (**int** i = 0; i < 10; ++i) {

System.**out**.println(random.nextInt(5));

}

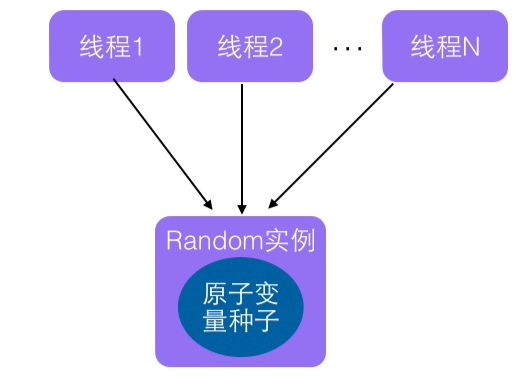
}

}

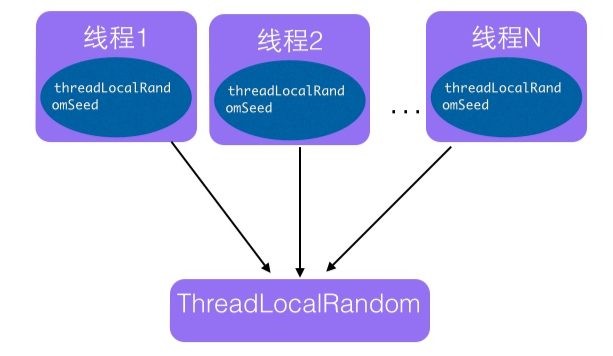
如上代码（10）调用 ThreadLocalRandom.current() 来获取当前线程的随机数生成器。

下面来分析下 ThreadLocalRandom 的实现原理。从名字看会让我们联想到[《Java 并发编程之美：基础篇》](http://gitbook.cn/m/mazi/activity/5aafb17477918b6e8444b65f) 中讲解的 ThreadLocal，ThreadLocal 的出现就是为了解决多线程下变量的隔离问题，让每一个线程拷贝一份变量，每个线程对变量进行操作时候实际是操作自己本地内存里面的拷贝。

实际上 ThreadLocalRandom 的实现也是这个原理，Random 的缺点是多个线程会使用原子性种子变量，会导致对原子变量更新的竞争，如下图：

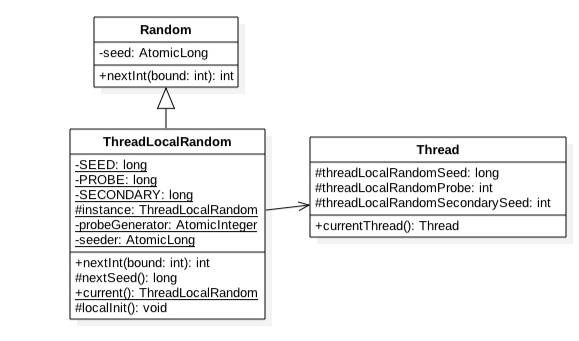


那么如果每个线程维护自己的一个种子变量，每个线程生成随机数时候根据自己老的种子计算新的种子，并使用新种子更新老的种子，然后根据新种子计算随机数，就不会存在竞争问题，这会大大提高并发性能，如下图 ThreadLocalRandom 原理：



##### ****源码分析****

首先看下 ThreadLocalRandom 的类图结构：



可知 ThreadLocalRandom 继承了 Random 并重写了 nextInt 方法，ThreadLocalRandom 中并没有使用继承自 Random 的原子性种子变量。

ThreadLocalRandom 中并没有具体存放种子，具体的种子是存放到具体的调用线程的 threadLocalRandomSeed 变量里面的，ThreadLocalRandom 类似于 ThreadLocal类 就是个工具类。当线程调用 ThreadLocalRandom 的 current 方法时候 ThreadLocalRandom 负责初始化调用线程的 threadLocalRandomSeed 变量，也就是初始化种子。

当调用 ThreadLocalRandom 的 nextInt 方法时候，实际上是获取当前线程的 threadLocalRandomSeed 变量作为当前种子来计算新的种子，然后更新新的种子到当前线程的 threadLocalRandomSeed 变量，然后在根据新种子和具体算法计算随机数。

这里需要注意的是 threadLocalRandomSeed 变量就是 Thread 类里面的一个普通 long 变量，并不是原子性变量，其实道理很简单，因为这个变量是线程级别的，根本不需要使用原子性变量，如果还是不理解可以思考下 ThreadLocal 的原理。

其中变量 seeder 和 probeGenerator 是两个原子性变量，在初始化调用线程的种子和探针变量时候用到，每个线程只会使用一次。

另外变量 instance 是个 ThreadLocalRandom 的一个实例，该变量是 static 的，当多线程通过 ThreadLocalRandom 的 current 方法获取 ThreadLocalRandom 的实例时候其实获取的是同一个，但是由于具体的种子是存放到线程里面的，所以 ThreadLocalRandom 的实例里面只是与线程无关的通用算法，所以是线程安全的。

下面看看 ThreadLocalRandom 的主要代码实现逻辑。

* Unsafe 机制的使用，具体的会在高级篇之二里面讲解。

**private** **static** **final** sun.misc.Unsafe UNSAFE;

**private** **static** **final** **long** SEED;

**private** **static** **final** **long** PROBE;

**private** **static** **final** **long** SECONDARY;

**static** {

**try** {

//获取unsafe实例

UNSAFE = sun.misc.Unsafe.getUnsafe();

Class<?> tk = Thread.class;

//获取Thread类里面threadLocalRandomSeed变量在Thread实例里面偏移量

SEED = UNSAFE.objectFieldOffset

(tk.getDeclaredField("threadLocalRandomSeed"));

//获取Thread类里面threadLocalRandomProbe变量在Thread实例里面偏移量

PROBE = UNSAFE.objectFieldOffset

(tk.getDeclaredField("threadLocalRandomProbe"));

//获取Thread类里面threadLocalRandomProbe变量在Thread实例里面偏移量，这个值在后面讲解的LongAdder里面会用到

SECONDARY = UNSAFE.objectFieldOffset

(tk.getDeclaredField("threadLocalRandomSecondarySeed"));

} **catch** (Exception e) {

**throw** **new** Error(e);

}

}

* ThreadLocalRandom current() 方法：该方法获取 ThreadLocalRandom 实例，并初始化调用线程中 threadLocalRandomSeed 和 threadLocalRandomProbe 变量。

**static** **final** ThreadLocalRandom instance = **new** ThreadLocalRandom();

**public** **static** ThreadLocalRandom **current**() {

//(12)

**if** (UNSAFE.getInt(Thread.currentThread(), PROBE) == 0)

//(13)

localInit();

//(14)

**return** instance;

}

**static** **final** **void** **localInit**() {

**int** p = probeGenerator.addAndGet(PROBE\_INCREMENT);

**int** probe = (p == 0) ? 1 : p; // skip 0

**long** seed = mix64(seeder.getAndAdd(SEEDER\_INCREMENT));

Thread t = Thread.currentThread();

UNSAFE.putLong(t, SEED, seed);

UNSAFE.putInt(t, PROBE, probe);

}

如上代码（12）如果当前线程中 threadLocalRandomProbe 变量值为0（默认情况下线程的这个变量为0），说明当前线程第一次调用 ThreadLocalRandom 的 current 方法，那么就需要调用 localInit 方法计算当前线程的初始化种子变量。这里设计为了延迟初始化，不需要使用随机数功能时候 Thread 类中的种子变量就不需要被初始化，这是一种优化。

代码（13）首先计算根据 probeGenerator 计算当前线程中 threadLocalRandomProbe 的初始化值，然后根据 seeder 计算当前线程的初始化种子，然后把这两个变量设置到当前线程。

代码（14）返回 ThreadLocalRandom 的实例，需要注意的是这个方法是静态方法，多个线程返回的是同一个 ThreadLocalRandom 实例。

* int nextInt(int bound) 方法：计算当前线程的下一个随机数。

**public** **int** **nextInt**(**int** bound) {

//(15)参数校验

**if** (bound <= 0)

**throw** **new** IllegalArgumentException(BadBound);

//(16) 根据当前线程中种子计算新种子

**int** r = mix32(nextSeed());

//(17)根据新种子和bound计算随机数

**int** m = bound - 1;

**if** ((bound & m) == 0) // power of two

r &= m;

**else** { // reject over-represented candidates

**for** (**int** u = r >>> 1;

u + m - (r = u % bound) < 0;

u = mix32(nextSeed()) >>> 1)

;

}

**return** r;

}

如上代码逻辑步骤与 Random 相似，我们重点看下 nextSeed() 方法：

**final** **long** **nextSeed**() {

Thread t; **long** r; //

UNSAFE.putLong(t = Thread.currentThread(), SEED,

r = UNSAFE.getLong(t, SEED) + GAMMA);

**return** r;

}

如上代码首先使用 r = UNSAFE.getLong(t, SEED) 获取当前线程中 threadLocalRandomSeed 变量的值，然后在种子的基础上累加 GAMMA 值作为新种子，然后使用 UNSAFE 的 putLong 方法把新种子放入当前线程的 threadLocalRandomSeed 变量。

**注：**本节首先讲解了 Random 的实现原理以及介绍了 Random 在多线程下存在竞争种子原子变量更新操作失败后自旋等待的缺点，从而引出 ThreadLocalRandom 类，ThreadLocalRandom 使用 ThreadLocal 的原理，让每个线程内持有一个本地的种子变量，该种子变量只有在使用随机数时候才会被初始化，多线程下计算新种子时候是根据自己线程内维护的种子变量进行更新，从而避免了竞争。

### Spring Request Scope 作用域 Bean 中 ThreadLocal 的使用

我们知道 Spring 中在 XML 里面配置 Bean 的时候可以指定 scope 属性来配置该 Bean 的作用域为 singleton、prototype、request、session 等，其中作用域为 request 的实现原理就是使用 ThreadLocal 实现的。

如果你想让你 Spring 容器里的某个 Bean 拥有 Web 的某种作用域，则除了需要 Bean 级上配置相应的 scope 属性，还必须在 web.xml 里面配置如下：

<listener>

<**listener-class**>org.springframework.web.context.request.RequestContextListener</**listener-class**></**listener**>

这里主要看 RequestContextListener 的两个方法：

**public** **void** **requestInitialized**(ServletRequestEvent

requestEvent)

和

**public** **void** **requestDestroyed**(ServletRequestEvent requestEvent)

当一个 Web 请求过来时候会执行 requestInitialized 方法：

**public** **void** **requestInitialized**(ServletRequestEvent requestEvent) {

.......

HttpServletRequest request = (HttpServletRequest) requestEvent.getServletRequest();

ServletRequestAttributes attributes = **new** ServletRequestAttributes(request);

request.setAttribute(REQUEST\_ATTRIBUTES\_ATTRIBUTE, attributes);

LocaleContextHolder.setLocale(request.getLocale());

//设置属性到threadlocal变量

RequestContextHolder.setRequestAttributes(attributes);

}

**public** **static** **void** **setRequestAttributes**(RequestAttributes attributes) {

setRequestAttributes(attributes, false);

}

**public** **static** **void** **setRequestAttributes**(RequestAttributes attributes, boolean inheritable) {

**if** (attributes == null) {

resetRequestAttributes();

}

**else** {

//默认inheritable=false

**if** (inheritable) {

inheritableRequestAttributesHolder.**set**(attributes);

requestAttributesHolder.**remove**();

}

**else** {

requestAttributesHolder.**set**(attributes);

inheritableRequestAttributesHolder.**remove**();

}

}

}

由于默认 inheritable 为 FALSE，我们的属性值都放到了 requestAttributesHolder 里面，而它的定义是：

**private** **static** **final** ThreadLocal<RequestAttributes> requestAttributesHolder =

**new** NamedThreadLocal<RequestAttributes>("Request attributes");

**private** **static** **final** ThreadLocal<RequestAttributes> inheritableRequestAttributesHolder =

**new** NamedInheritableThreadLocal<RequestAttributes>("Request context");

其中 NamedThreadLocal<T> extends ThreadLocal<T>，所以不具有继承性。

NamedInheritableThreadLocal<T> extends InheritableThreadLocal<T>，所以具有继承性，所以默认放入到 RequestContextHolder 里面的属性值在子线程中获取不到。

当请求结束时候调用 requestDestroyed 方法，代码如下：

**public** **void** **requestDestroyed**(ServletRequestEvent requestEvent) {

ServletRequestAttributes attributes =

(ServletRequestAttributes) requestEvent.getServletRequest().getAttribute(REQUEST\_ATTRIBUTES\_ATTRIBUTE);

ServletRequestAttributes threadAttributes =

(ServletRequestAttributes) RequestContextHolder.getRequestAttributes();

**if** (threadAttributes != null) {

// We're assumably within the original request thread...

**if** (attributes == null) {

attributes = threadAttributes;

}

//请求结束则清除当前线程的线程变量。

LocaleContextHolder.resetLocaleContext();

RequestContextHolder.resetRequestAttributes();

}

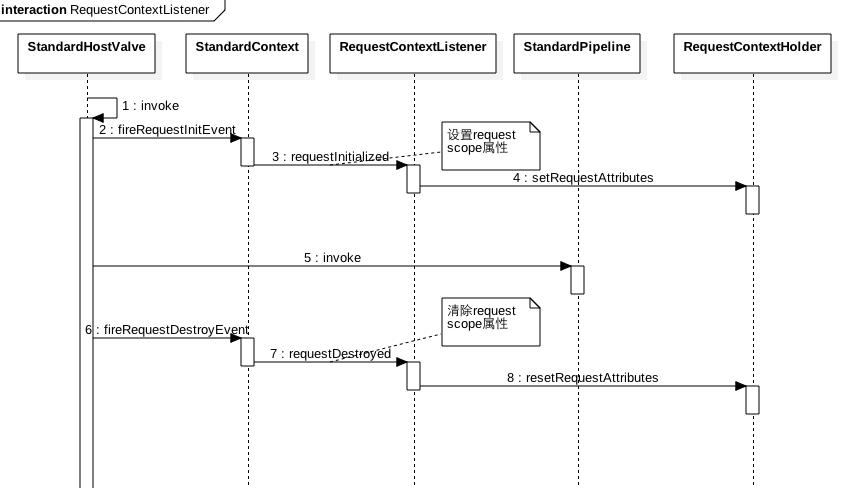
**if** (attributes != null) {

attributes.requestCompleted();

}

}

下面从时序图看下一次 Web 请求调用逻辑如何：



也就是说每次发起一个 Web 请求在 Tomcat 中 context（具体应用）处理前，host 匹配后都会去设置下 RequestContextHolder 属性，让 requestAttributesHolder 不为空，在请求结束时候会清除。

**总结：**默认情况下放入 RequestContextHolder 里面的属性子线程访问不到。Spring 的 request 作用域的 Bean 是使用 threadlocal 实现的。

#### 模拟请求的简单 Demo

* web.xml里面配置如下。

因为是 request 作用域，所以必须是 Web 项目，并且需要配置 RequestContextListener 到 web.xml。

<listener>

<**listener-class**>org.springframework.web.context.request.RequestContextListener</**listener-class**></**listener**>

* 注入一个 request 作用域 bean 到 IOC 容器。

<bean id="requestBean" **class**="com.zlx.test.RequestBean"

scope="request">

<**property** name="name" value="jiaduo" />

<**aop:scoped-proxy** />

</**bean**>

* 测试RPC。

@WebResource("/testService")**public** **class** **TestRpc** {

@Autowired

**private** RequestBean requestInfo;

@ResourceMapping("test")

**public** ActionResult **test**(ErrorContext context) {

ActionResult result = **new** ActionResult();

pvgInfo.setName("jiaduo");

String name = requestInfo.getName();

result.setValue(name);

**return** result;

}

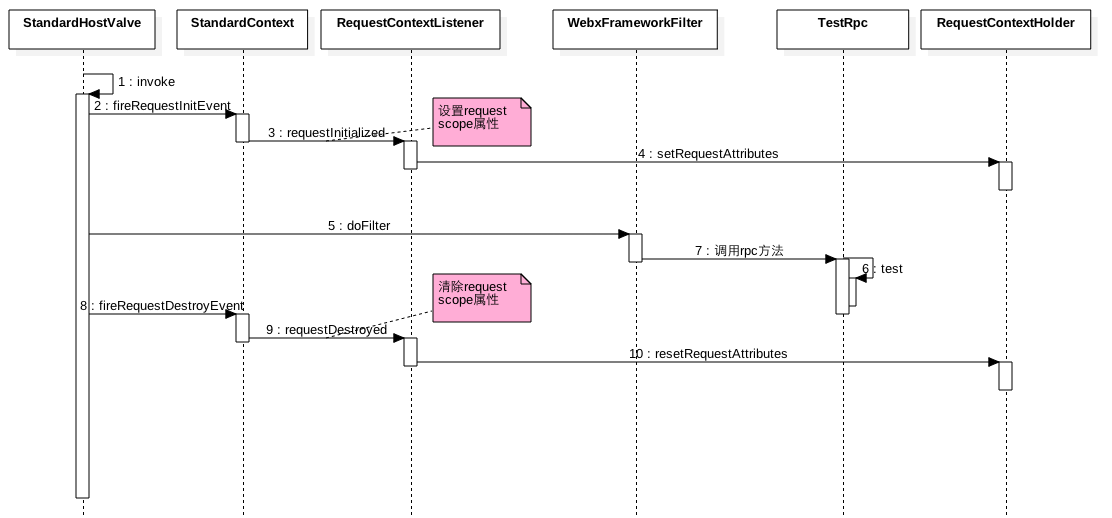
}

如上首先配置 RequestContextListener 到 web.xml 里面，然后注入了 Request 作用域的 RequestBean 的实例到 IOC 容器，最后 TestRpc 内注入了 RequestBean 的实例，方法 test 首先调用了 requestInfo 的方法 setName 设置 name 属性，然后获取 name 属性并返回。

这里如果 requestInfo 对象是单例的，那么多个线程同时调用 test 方法后，每个线程都是设置-获取的操作，这个操作不是原子性的，会导致线程安全问题。而这里声明的作用域为 request 级别，也是每个线程都有一个 requestInfo 的本地变量。

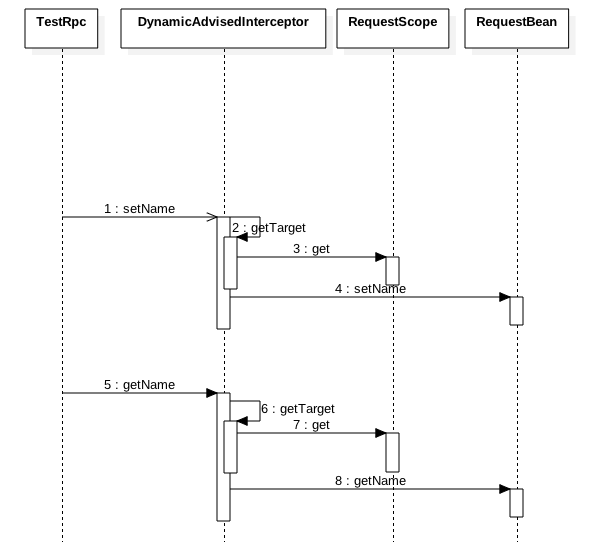
#### 原理剖析

下面分析下当调用这个 RPC 方法请求时候时序图为：



这个时序图与上节的类似，只是这里使用 WebX 搭建的 Web 项目，所以多了的 WebXFrameworkFilter，转发请求到我们定义的 TestRpc。

下面着重看下调用 test 时候发生了什么：



其实前面创建的 requestInfo 是被经过 CGliB 代理后的（感兴趣的童鞋可以研究下 ScopedProxyFactoryBean 这类），所以这里调用 setName 或者 getName 时候会被 DynamicAdvisedInterceptor 拦截的，拦击器里面最终会调用到 RequestScope 的 get 方法获取当前线程持有的本地变量。

RequestScope 的 get 方法代码如下：

public **Object** get(**String** name, ObjectFactory objectFactory) {

RequestAttributes attributes = RequestContextHolder.currentRequestAttributes();//（1）

**Object** scopedObject = attributes.getAttribute(name, getScope());

**if** (scopedObject == null) {

scopedObject = objectFactory.getObject();//（2）

attributes.setAttribute(name, scopedObject, getScope());//（3）

}

**return** scopedObject;

}

可知当发起一个请求时候，首先会通过 RequestContextListener.requestInitialized 里面调用 RequestContextHolder.setRequestAttributess 设置 requestAttributesHolder。

然后请求被路由到 TestRpc 的 test 方法后，test 方法内第一次调用 setName 方法时候，最终会调用 RequestScope.get()方法，get 方法内代码（1）获取通过 RequestContextListener.requestInitialized 设置的线程本地变量 requestAttributesHolder 保存的属性集的值。

然后看该属性集里面是否有名字为 requestInfo 的属性，由于是第一次调用，所以不存在，所以会执行代码（2）让 Spring 创建一个 RequestInfo 对象，然后设置到属性集 attributes，也就是保存到了当前请求线程的本地内存里面了。然后返回创建的对象，调用创建对象的 setName。

然后 test 方法内紧接着调用了 getName 方法，最终会调用 RequestScope.get() 方法，get 方法内代码（1）获取通过 RequestContextListener.requestInitialized 设置的线程本地变量 RequestAttributes，然后看该属性集里面是否有名字为 requestInfo 的属性，由于是第一次调用 setName 时候已经设置名字为 requestInfo 的 bean 到 ThreadLocal 变量里面了，并且调用 setName 和 getName 的是同一个线程，所以这里直接返回了调用 setName 时候创建的 RequestInfo 对象，然后调用它的 getName 方法。

### 总结

本文通过循序渐进的方式，先讲解了 ThreadLocal 的简单使用，然后讲解了 ThreadLocal 的实现原理，并指出 ThreadLocal 不支持继承性；然后紧接着讲解了 InheritableThreadLocal 是如何补偿了 ThreadLocal 不支持继承的特性；然后讲解了 ThreadLocalRandom 是如何借鉴 ThreadLocal 的思想补充了 Random 的不足；最后简单的介绍了 Spring 框架中如何使用 ThreadLocal 实现了 Reqeust Scope 的 Bean。在后面的并发编制之美——项目实践与常见问题解答中，我们会深入讲解使用 ThreadLocal 导致内存泄露的案例，敬请期待。

# Java 并发编程之美：并发编程高级篇之二

### 一、前言

Java 并发编程实践中的话：

编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却还都离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本场 Chat 作为 Java 并发编程之美系列的高级篇之二，主要讲解内容如下：（建议先阅读 [Java 并发编程之美：基础篇](http://gitbook.cn/m/mazi/activity/5aafb17477918b6e8444b65f) ）

* rt.jar 中 Unsafe 类主要函数讲解， Unsafe 类提供了硬件级别的原子操作，可以安全的直接操作内存变量，其在 JUC 源码中被广泛的使用，了解其原理为研究 JUC 源码奠定了基础。
* rt.jar 中 LockSupport 类主要函数讲解，LockSupport 是个工具类，主要作用是挂起和唤醒线程，是创建锁和其它同步类的基础，了解其原理为研究 JUC 中锁的实现奠定基础。
* 讲解 JDK8 新增原子操作类 LongAdder 实现原理，并讲解 AtomicLong 的缺点是什么，LongAdder 是如何解决 AtomicLong 的缺点的，LongAdder 和 LongAccumulator 是什么关系？
* JUC 并发包中并发组件 CopyOnWriteArrayList 的实现原理，CopyOnWriteArrayList 是如何通过写时拷贝实现并发安全的 List？

### 二、 Unsafe 类探究

JDK 的 rt.jar 包中的 Unsafe 类提供了硬件级别的原子操作，Unsafe 里面的方法都是 native 方法，通过使用 JNI 的方式来访问本地 C++ 实现库。下面我们看下 Unsafe 提供的几个主要方法以及编程时候如何使用 Unsafe 类做一些事情。

#### 2.1 主要方法介绍

* long objectFieldOffset(Field field) 方法

作用：返回指定的变量在所属类的内存偏移地址，偏移地址仅仅在该 Unsafe 函数中访问指定字段时候使用。如下代码使用 unsafe 获取AtomicLong 中变量 value 在 AtomicLong 对象中的内存偏移

**static** {

**try** {

valueOffset = unsafe.objectFieldOffset

(AtomicLong.class.getDeclaredField("value"));

} **catch** (Exception ex) { **throw** **new** Error(ex); }

}

int arrayBaseOffset(Class arrayClass) 方法 获取数组中第一个元素的地址

int arrayIndexScale(Class arrayClass) 方法 获取数组中单个元素占用的字节数

boolean compareAndSwapLong(Object obj, long offset, long expect, long update) 方法 比较对象 obj 中偏移量为 offset 的变量的值是不是和 expect 相等，相等则使用 update 值更新，然后返回 true，否者返回 false

public native long getLongVolatile(Object obj, long offset) 方法 获取对象 obj 中偏移量为 offset 的变量对应的 volatile 内存语义的值。

void putLongVolatile(Object obj, long offset, long value) 方法 设置 obj 对象中内存偏移为 offset 的 long 型变量的值为 value，支持 volatile 内存语义。

void putOrderedLong(Object obj, long offset, long value) 方法 设置 obj 对象中 offset 偏移地址对应的 long 型 field 的值为 value。这是有延迟的 putLongVolatile 方法，并不保证值修改对其它线程立刻可见。变量只有使用 volatile 修饰并且期望被意外修改的时候使用才有用。

void park(boolean isAbsolute, long time) 阻塞当前线程，其中参数 isAbsolute 等于 false 时候，time 等于 0 表示一直阻塞，time 大于 0 表示等待指定的 time 后阻塞线程会被唤醒，这个 time 是个相对值，是个增量值，也就是相对当前时间累加 time 后当前线程就会被唤醒。 如果 isAbsolute 等于 true，并且 time 大于 0 表示阻塞后到指定的时间点后会被唤醒，这里 time 是个绝对的时间，是某一个时间点换算为 ms 后的值。另外当其它线程调用了当前阻塞线程的 interrupt 方法中断了当前线程时候，当前线程也会返回，当其它线程调用了 unpark 方法并且把当前线程作为参数时候当前线程也会返回。

void unpark(Object thread) 唤醒调用 park 后阻塞的线程，参数为需要唤醒的线程。

下面是 Jdk8 新增的方法，这里简单的列出 Long 类型操作的方法

* long getAndSetLong(Object obj, long offset, long update) 方法 获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量 volatile 语义的值为 update。

**public** **final** **long** **getAndSetLong**(Object obj, **long** offset, **long** update)

{

**long** l;

**do**

{

l = getLongVolatile(obj, offset);//(1)

} **while** (!compareAndSwapLong(obj, offset, l, update));

**return** l;

}

从代码可知内部代码 (1) 处使用 getLongVolatile 获取当前变量的值，然后使用 CAS 原子操作进行设置新值，这里使用 while 循环是考虑到多个线程同时调用的情况 CAS 失败后需要自旋重试。

* long getAndAddLong(Object obj, long offset, long addValue) 方法 获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量值为原始值 +addValue。

**public** **final** **long** **getAndAddLong**(Object obj, **long** offset, **long** addValue)

{

**long** l;

**do**

{

l = getLongVolatile(obj, offset);

} **while** (!compareAndSwapLong(obj, offset, l, l + addValue));

**return** l;

}

类似 getAndSetLong 的实现，只是这里使用CAS的时候使用了原始值+传递的增量参数 addValue 的值。

#### 2.2 如何使用 Unsafe 类

看到 Unsafe 这个类如此牛叉，你肯定会忍不住撸下下面代码，期望能够使用 Unsafe 做点事情。

**public** **class** **TestUnSafe** {

//获取Unsafe的实例（2.2.1）

**static** **final** Unsafe unsafe = Unsafe.getUnsafe();

//记录变量state在类TestUnSafe中的偏移值（2.2.2）

**static** **final** **long** stateOffset;

//变量(2.2.3)

**private** **volatile** **long** state=0;

**static** {

**try** {

//获取state变量在类TestUnSafe中的偏移值(2.2.4)

stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state"));

} **catch** (Exception ex) {

System.out.println(ex.getLocalizedMessage());

**throw** **new** Error(ex);

}

}

**public** **static** **void** **main**(String[] args) {

//创建实例，并且设置state值为1(2.2.5)

TestUnSafe test = **new** TestUnSafe();

//(2.2.6)

Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);

System.out.println(sucess);

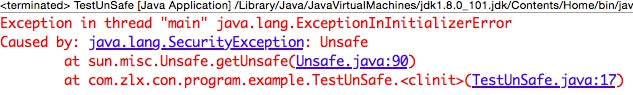
}

}

如上代码（2.2.1）获取了 Unsafe 的一个实例，代码（2.2.3）创建了一个变量 state 初始化为 0。

代码（2.2.4）使用 unsafe.objectFieldOffset 获取 TestUnSafe 类里面的 state 变量在 TestUnSafe 对象里面的内存偏移量地址并保存到 stateOffset 变量。

代码（2.2.6）调用创建的 unsafe 实例的 compareAndSwapInt 方法，设置 test 对象的 state 变量的值，具体意思是如果 test 对象内存偏移量为 stateOffset 的 state 的变量为 0，则更新该值为 1。

运行上面代码我们期望会输出 true，然而执行后会输出如下结果： 

为研究其原因，必然要翻看 getUnsafe 代码，看看里面做了啥：

**private** **static** **final** Unsafe theUnsafe = **new** Unsafe();

**public** **static** Unsafe **getUnsafe**(){

//（2.2.7）

Class localClass = Reflection.getCallerClass();

//（2.2.8)

**if** (!VM.isSystemDomainLoader(localClass.getClassLoader())) {

**throw** **new** SecurityException("Unsafe");

}

**return** theUnsafe;

}

//判断paramClassLoader是不是BootStrap类加载器(2.2.9)

**public** **static** **boolean** **isSystemDomainLoader**(ClassLoader paramClassLoader)

{

**return** paramClassLoader == **null**;

}

代码（2.2.7）获取调用 getUnsafe 这个方法的对象的 Class 对象，这里是 TestUnSafe.class。

代码（2.2.8）判断是不是 Bootstrap 类加载器加载的 localClass，这里是看是不是 Bootstrap 加载器加载了 TestUnSafe.class。很明显由于 TestUnSafe.class 是使用 AppClassLoader 加载的（可以参考 chat： [Java 类加载器揭秘](http://gitbook.cn/gitchat/activity/5a751b1391d6b7067048a213)），所以这里直接抛出了异常。

思考下，这里为何要有这个判断那？

我们知道 Unsafe 类是在 rt.jar 里面提供的，而 rt.jar 里面的类是使用 Bootstrap 类加载器加载的，而我们启动 main 函数所在的类是使用 AppClassLoader 加载的，所以在 main 函数里面加载 Unsafe 类时候鉴于委托机制会委托给 Bootstrap 去加载 Unsafe 类。

如果没有代码（2.2.8）这鉴权，那么我们应用程序就可以随意使用 Unsafe 做事情了，而 Unsafe 类可以直接操作内存，是不安全的，所以 JDK 开发组特意做了这个限制，不让开发人员在正规渠道下使用 Unsafe 类，而是在 rt.jar 里面的核心类里面使用 Unsafe 功能。

那么如果开发人员真的想要实例化 Unsafe 类，使用 Unsafe 的功能该如何做那？

方法有很多种，既然正规渠道访问不了，那么就玩点黑科技，使用万能的反射来获取 Unsafe 实例方法：

**public** **class** **TestUnSafe** {

**static** **final** Unsafe unsafe;

**static** **final** **long** stateOffset;

**private** **volatile** **long** state = 0;

**static** {

**try** {

// 反射获取 Unsafe 的成员变量 theUnsafe（2.2.10）

Field field = Unsafe.class.getDeclaredField("theUnsafe");

// 设置为可存取（2.2.11）

field.setAccessible(**true**);

// 获取该变量的值（2.2.12）

unsafe = (Unsafe) field.get(**null**);

//获取 state 在 TestUnSafe 中的偏移量 （2.2.13）

stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state"));

} **catch** (Exception ex) {

System.out.println(ex.getLocalizedMessage());

**throw** **new** Error(ex);

}

}

**public** **static** **void** **main**(String[] args) {

TestUnSafe test = **new** TestUnSafe();

Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);

System.out.println(sucess);

}

}

如上代码通过代码（2.2.10），（2.2.11），（2.2.12）反射获取 unsafe 的实例，然后运行结果输出： IMG_257

### 三、LockSupport类探究

JDK 中的 rt.jar 里面的 LockSupport 是个工具类，主要作用是挂起和唤醒线程，它是创建锁和其它同步类的基础。

LockSupport 类与每个使用它的线程都会关联一个许可证,默认调用LockSupport 类的方法的线程是不持有许可证的，LockSupport 内部使用 Unsafe 类实现，下面介绍下 LockSupport 内的几个主要函数：

* void park() 方法 如果调用 park() 的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 会马上返回，否者调用线程会被禁止参与线程的调度，也就是会被阻塞挂起。

如下代码，直接在 main 函数里面调用 park 方法，最终结果只会输出begin park!，然后当前线程会被挂起，这是因为默认下调用线程是不持有许可证的。

**public** **static** **void** **main**( String[] args )

{

System.out.println( "begin park!" );

LockSupport.park();

System.out.println( "end park!" );

}

在其它线程调用 unpark(Thread thread) 方法并且当前线程作为参数时候，调用park方法被阻塞的线程会返回，另外其它线程调用了阻塞线程的 interrupt() 方法，设置了中断标志时候或者由于线程的虚假唤醒原因后阻塞线程也会返回，所以调用 park() 最好也是用循环条件判断方式（关于虚假唤醒可以参考 chat ：[线程相关的基础知识](http://gitbook.cn/gitchat/activity/5aa4d205c2ff6f2e120891dd)）。

需要注意的是调用 park() 方法被阻塞的线程被其他线程中断后阻塞线程返回时候并不会抛出 InterruptedException 异常。

* void unpark(Thread thread) 方法 当一个线程调用了 unpark 时候，如果参数 thread 线程没有持有 thread 与 LockSupport 类关联的许可证，则让 thread 线程持有。如果 thread 之前调用了 park() 被挂起，则调用 unpark 后，该线程会被唤醒。如果 thread 之前没有调用 park，则调用 unPark 方法后，在调用 park() 方法，会立刻返回，上面代码修改如下：

**public** **static** **void** **main**( String[] args )

{

System.out.println( "begin park!" );

//使当前线程获取到许可证

LockSupport.unpark(Thread.currentThread());

//再次调用park

LockSupport.park();

System.out.println( "end park!" );

}

则会输出： begin park! end park!

下面再来看一个例子来加深对 park,unpark 的理解

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

System.out.println("child thread begin park!");

// 调用park方法，挂起自己

LockSupport.park();

System.out.println("child thread unpark!");

}

});

//启动子线程

thread.start();

//主线程休眠1S

Thread.sleep(1000);

System.out.println("main thread begin unpark!");

//调用unpark让thread线程持有许可证，然后park方法会返回

LockSupport.unpark(thread);

}

输出为：

child thread **begin** park!**main** **thread** **begin** unpark!**child** **thread** unpark!

上面代码首先创建了一个子线程 thread，启动后子线程调用 park 方法，由于默认子线程没有持有许可证，会把自己挂起。

主线程休眠 1s 为的是主线程在调用 unpark 方法前让子线程输出 child thread begin park! 并阻塞。

主线程然后执行 unpark 方法，参数为子线程，目的是让子线程持有许可证，然后子线程调用的 park 方法就返回了。

park 方法返回时候不会告诉你是因为何种原因返回，所以调用者需要根据之前是处于什么目前调用的 park 方法，再次检查条件是否满足，如果不满足的话还需要再次调用 park 方法。

例如，线程在返回时的中断状态，根据调用前后中断状态对比就可以判断是不是因为被中断才返回的。

为了说明调用 park 方法后的线程被中断后会返回，修改上面例子代码，删除 LockSupport.unpark(thread); 然后添加 thread.interrupt(); 代码如下：

**public** **static** **void** **main**(String[] args) **throws** InterruptedException {

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

System.out.println("child thread begin park!");

// 调用park方法，挂起自己,只有被中断才会退出循环

**while** (!Thread.currentThread().isInterrupted()) {

LockSupport.park();

}

System.out.println("child thread unpark!");

}

});

// 启动子线程

thread.start();

// 主线程休眠1S

Thread.sleep(1000);

System.out.println("main thread begin unpark!");

// 中断子线程线程

thread.interrupt();

}

输出为：

child thread **begin** park!**main** **thread** **begin** unpark!**child** **thread** unpark!

如上代码也就是只有当子线程被中断后子线程才会运行结束，如果子线程不被中断，即使你调用 unPark(thread) 子线程也不会结束。

* void parkNanos(long nanos)函数

和 park 类似，如果调用 park 的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 会马上返回，不同在于如果没有拿到许可调用线程会被挂起 nanos 时间后在返回。

park 还支持三个带有 blocker 参数的方法，当线程因为没有持有许可的情况下调用 park 被阻塞挂起时候，这个 blocker 对象会被记录到该线程内部。

使用诊断工具可以观察线程被阻塞的原因，诊断工具是通过调 getBlocker(Thread) 方法来获取该 blocker 对象的，所以 JDK 推荐我们使用带有 blocker 参数的 park 方法,并且 blocker 设置为 this，这样当内存 dump 排查问题时候就能知道是那个类被阻塞了。

例如下面代码：

**public** **class** **TestPark** {

**public** **void** **testPark**(){

LockSupport.park();//(1)

}

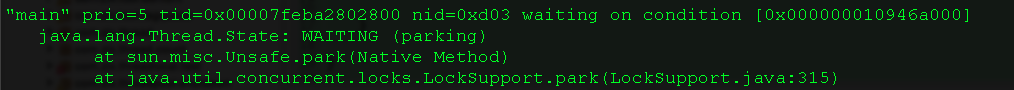
**public** **static** **void** **main**(String[] args) {

TestPark testPark = **new** TestPark();

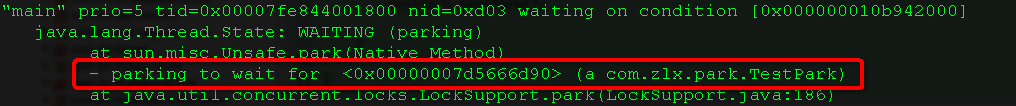
testPark.testPark();

}

}

运行后使用 jstack pid 查看线程堆栈时候可以看到如下： 

修改 代码（1）为 LockSupport.park(this) 后运行在 jstack pid 结果为：



可知使用带 blocker 的 park 方法后，线程堆栈可以提供更多有关阻塞对象的信息。

* park(Object blocker) 函数

**public** **static** **void** **park**(Object blocker) {

//获取调用线程

Thread t = Thread.currentThread();

//设置该线程的 blocker 变量

setBlocker(t, blocker);

//挂起线程

UNSAFE.park(**false**, 0L);

//线程被激活后清除 blocker 变量，因为一般都是线程阻塞时候才分析原因

setBlocker(t, **null**);

}

Thread 类里面有个变量 volatile Object parkBlocker 用来存放 park 传递的 blocker 对象，也就是把 blocker 变量存放到了调用 park 方法的线程的成员变量里面。

* void parkNanos(Object blocker, long nanos) 函数 相比 park(Object blocker) 多了个超时时间。
* void parkUntil(Object blocker, long deadline) parkUntil 的代码如下：

**public** **static** **void** **parkUntil**(Object blocker, **long** deadline) {

Thread t = Thread.currentThread();

setBlocker(t, blocker);

//isAbsolute=true,time=deadline;表示到 deadline 时间时候后返回

UNSAFE.park(**true**, deadline);

setBlocker(t, **null**);

}

可知是设置一个 deadline，时间单位为 milliseconds，是从 1970 到现在某一个时间点换算为毫秒后的值，这个和 parkNanos(Object blocker, long nanos) 区别是后者是从当前算等待 nanos 时间，而前者是指定一个时间点，比如我需要等待到 2017.12.11 日 12:00:00，则吧这个时间点转换为从 1970 年到这个时间点的总毫秒数。

最后在看一个例子

**class** **FIFOMutex** {

**private** **final** AtomicBoolean locked = **new** AtomicBoolean(**false**);

**private** **final** Queue<Thread> waiters = **new** ConcurrentLinkedQueue<Thread>();

**public** **void** **lock**() {

**boolean** wasInterrupted = **false**;

Thread current = Thread.currentThread();

waiters.add(current);

// 只有队首的线程可以获取锁（1）

**while** (waiters.peek() != current || !locked.compareAndSet(**false**, **true**)) {

LockSupport.park(**this**);

**if** (Thread.interrupted()) // （2）

wasInterrupted = **true**;

}

waiters.remove();

**if** (wasInterrupted) // （3）

current.interrupt();

}

**public** **void** **unlock**() {

locked.set(**false**);

LockSupport.unpark(waiters.peek());

}

}

这是一个先进先出的锁，也就是只有队列首元素可以获取锁，代码（1）处如果当前线程不是队首或者当前锁已经被其它线程获取，则调用park方法挂起自己。

然后代码（2）处判断，如果 park 方法是因为被中断而返回，则忽略中断，并且重置中断标志，只做个标记，然后再次判断当前线程是不是队首元素或者当前锁是否已经被其它线程获取，如果是则继续调用 park 方法挂起自己。

然后代码（3）中如果标记为 true 则中断该线程，这个怎么理解那？其实意思是其它线程中断了该线程，虽然我对中断信号不感兴趣，忽略它，但是不代表其它线程对该标志不感兴趣，所以要恢复下。

### 四、 LongAdder 和 LongAccumulator 原理探究

#### 4.1 LongAdder 原理

LongAdder 类是 JDK8 新增的一个原子性操作类。AtomicLong 通过 CAS 算法提供了非阻塞的原子性操作，相比使用阻塞算法的同步器来说性能已经很好了，但是 JDK 开发组并不满足于此，因为在非常高的并发请求下 AtomicLong 的性能不能让它们接受。（关于 CAS 操作可以参考 chat ：[并发编程基础晋级篇](http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f)）

如下 AtomicLong 的 incrementAndGet 的代码，虽然 AtomicLong 使用 CAS 算法，但是 CAS 失败后还是通过无限循环的自旋锁不断尝试的：

**public** **final** **long** **incrementAndGet**() {

**for** (;;) {

**long** current = get();

**long** next = current + 1;

**if** (compareAndSet(current, next))

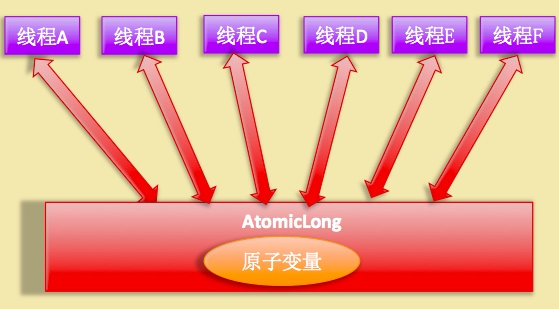
**return** next;

}

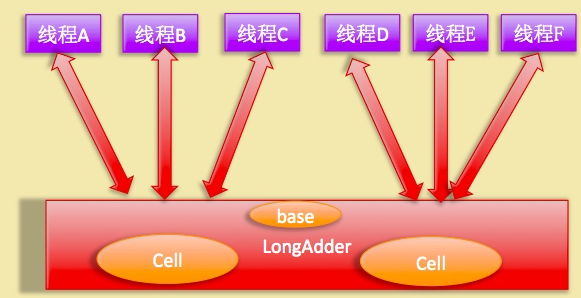
}

在高并发下 N 多线程同时去操作一个变量会造成大量线程 CAS 失败然后处于自旋状态，这大大浪费了 cpu 资源，降低了并发性。

那么既然 AtomicLong 性能问题是由于过多线程同时去竞争同一个变量的更新而降低的，那么如果把一个变量分解为多个变量，让同样多的线程去竞争多个资源那么性能问题不就解决了？是的，JDK 8 提供的 LongAdder 就是这个思路。下面通过图形来标示两者不同。



如上图 AtomicLong 是多个线程同时竞争同一个变量情景。



如上图 LongAdder 则是内部维护多个 Cell 变量，每个 Cell 里面有一个初始值为 0 的 long 型变量，在同等并发量的情况下，争夺单个变量的线程量会减少，这是变相的减少了争夺共享资源的并发量，另外多个线程在争夺同一个原子变量时候如果失败并不是自旋 CAS 重试，而是尝试获取其它原子变量的锁，最后获取当前值时候是把所有变量的值累加后在加上 base 返回的。

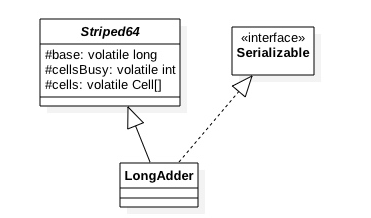
LongAdder 维护了一个延迟初始化的原子性更新数组和一个基值变量base. 数组的大小保持是 2 的 N 次方大小，数组表的下标使用每个线程的 hashcode 值的掩码表示，数组里面的变量实体是 Cell 类型。

Cell 类型是 AtomicLong 的一个改进，用来减少缓存的争用，对于大多数原子操作字节填充是浪费的，因为原子性操作都是无规律的分散在内存中进行的，多个原子性操作彼此之间是没有接触的，但是原子性数组元素彼此相邻存放将能经常共享缓存行也就是伪共享（关于伪共享，可以参考 Chat ：[并发编程基础晋级篇](http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f)），所以这在性能上是一个提升。

另外由于 Cells 占用内存是相对比较大的，所以一开始并不创建，而是在需要时候在创建，也就是惰性加载，当一开始没有空间时候，所有的更新都是操作base变量，

##### 4.1.1 LongAdder 代码简单分析

上节我们讲解了 LongAdder 的原理，这节简单介绍下代码实现（详细实现读者可以翻看代码研究），为了降低高并发下多线程对一个变量 CAS 争夺失败后大量线程会自旋而造成降低并发性能问题，LongAdder 内部通过根据并发请求量维护多个 Cell 元素（一个动态的 Cell 数组）来分担对单个变量进行争夺的并发量。

首先看下 LongAdder 的构造  如图可知 LongAdder 继承自 Striped64 类，Striped64 内部维护着三个变量，LongAdder 的真实值其实是 base 的值与 Cell 数组里面所有 Cell元素值的累加，base 是个基础值默认是 0，cellsBusy 用来实现自旋锁，当创建 Cell 元素或者扩容 Cell 数组时候用来进行线程间的同步。

下面看看 Cell 的构造：

@sun.misc.Contended

**static** **final** **class** **Cell** {

**volatile** **long** value;

Cell(**long** x) { value = x; }

**final** **boolean** **cas**(**long** cmp, **long** val) {

**return** UNSAFE.compareAndSwapLong(**this**, valueOffset, cmp, val);

}

// Unsafe mechanics

**private** **static** **final** sun.misc.Unsafe UNSAFE;

**private** **static** **final** **long** valueOffset;

**static** {

**try** {

UNSAFE = sun.misc.Unsafe.getUnsafe();

Class<?> ak = Cell.class;

valueOffset = UNSAFE.objectFieldOffset

(ak.getDeclaredField("value"));

} **catch** (Exception e) {

**throw** **new** Error(e);

}

}

}

如上代码可知 Cell 的构造很简单，内部维护一个声明为 volatile 的变量，这里声明为 volatile 是因为线程操作 value 变量时候没有使用锁，为了保证变量的内存可见性这里只有声明为 volatile（关于共享变量的内存可见性问题可以参考 chat ：[线程相关的基础知识](http://gitbook.cn/gitchat/activity/5aa4d205c2ff6f2e120891dd)）。另外还是老生常谈的使用 UNsafe 类的方法来设置 value 的值。

* long sum() 返回当前的值，内部操作是累加所有 Cell 内部的 value 的值后累加 base，如下代码，由于计算总和时候没有对 Cell 数组进行加锁，所以在累加过程中可能有其它线程对 Cell 中的值进行了修改，也有可能数组进行了扩容，所以 sum 返回的值并不是非常精确的，返回值并不是一个调用 sum 方法时候的一个原子快照值。

**public** **long** **sum**() {

Cell[] as = cells; Cell a;

**long** sum = base;

**if** (as != **null**) {

**for** (**int** i = 0; i < as.length; ++i) {

**if** ((a = as[i]) != **null**)

sum += a.value;

}

}

**return** sum;

}

* void reset() 重置操作，如下代码把 base 置为 0，如果 Cell 数组有元素，则元素值重置为 0

**public** **void** **reset**() {

Cell[] as = cells; Cell a;

base = 0L;

**if** (as != **null**) {

**for** (**int** i = 0; i < as.length; ++i) {

**if** ((a = as[i]) != **null**)

a.value = 0L;

}

}

}

* long sumThenReset() sum 的改造版本，如下代码，在计算 sum 累加对应的 cell 值后，把当前 cell 的值重置为 0，base 重置为 0。 当多线程调用该方法时候会有问题，比如考虑第一个调用线程会清空 Cell 的值，后一个线程调用时候累加时候累加的都是 0 值。

**public** **long** **sumThenReset**() {

Cell[] as = cells; Cell a;

**long** sum = base;

base = 0L;

**if** (as != **null**) {

**for** (**int** i = 0; i < as.length; ++i) {

**if** ((a = as[i]) != **null**) {

sum += a.value;

a.value = 0L;

}

}

}

**return** sum;

}

long longValue() 等价于 sum()

void add(long x） 累加增量 x 到原子变量，这个过程是原子性的

**public** **void** **add**(**long** x) {

Cell[] as; **long** b, v; **int** m; Cell a;

**if** ((as = cells) != **null** || !casBase(b = base, b + x)) {//(1)

**boolean** uncontended = **true**;

**if** (as == **null** || (m = as.length - 1) < 0 ||//(2)

(a = as[getProbe() & m]) == **null** ||//(3)

!(uncontended = a.cas(v = a.value, v + x)))//(4)

longAccumulate(x, **null**, uncontended);//(5)

}

}

**final** **boolean** **casBase**(**long** cmp, **long** val) {

**return** UNSAFE.compareAndSwapLong(**this**, BASE, cmp, val);

}

如上代码当第一个线程 A 执行 add 时候，代码（1）会执行 casBase 方法通过 cas 设置 base 为 x，如果成功则直接返回，这时候base的值为 1。

假如多个线程同时执行 add 时候，同时执行到 casBase 则只有一个线程A成功返回其它线程由于cas失败执行代码（2），代码（2）是获取 cells 数组的长度，如果数组长度为0则执行（5），否者 cells 长度不为 0说明 cells 数组有元素则执行代码（3），代码（3）首先计算当前线程在数组中下标，然后获取当前线程对应的cell值，如果获取到则执行（4）进行 cas 操作，cas 失败则否者执行（5）。代码（5）里面是具体进行数组扩充和初始化的，这个代码比较杂，本文只讲解原理，不深入代码细节，感兴趣的读者可以去研究下。

#### 4.2 LongAccumulator 类原理探究

LongAdder 类是 LongAccumulator 的一个特例，LongAccumulator 提供了比 LongAdder 更强大的功能，如下构造函数其中 accumulatorFunction 一个双目运算器接口，根据输入的两个参数返回一个计算值，identity 则是 LongAccumulator 累加器的初始值。

**public** **LongAccumulator**(LongBinaryOperator accumulatorFunction,

**long** identity) {

**this**.function = accumulatorFunction;

base = **this**.identity = identity;

}

**public** **interface** **LongBinaryOperator** {

//根据两个参数计算返回一个值

**long** **applyAsLong**(**long** left, **long** right);

}

上面提到 LongAdder 其实是 LongAccumulator 的一个特例，调用LongAdder 相当使用下面的方式调用 LongAccumulator。

LongAdder adder = **new** LongAdder();

LongAccumulator accumulator = **new** LongAccumulator(**new** LongBinaryOperator() {

@Override

**public** **long** **applyAsLong**(**long** left, **long** right) {

**return** left + right;

}

}, 0);

LongAccumulator 相比于 LongAdder 可以提供累加器初始非 0 值，后者只能默认为0，另外前者还可以指定累加规则，比如不是累加而是相乘，只需要构造 LongAccumulator 时候传入自定义双面运算器就 OK，后者则内置累加的规则。

从下面代码知道 LongAccumulator 相比于 LongAdder 不同在于 casBase 时候后者传递的是 b+x，前者则是调用了 r = function.applyAsLong(b = base, x) 来计算。

**public** **void** **add**(**long** x) {

Cell[] as; **long** b, v; **int** m; Cell a;

**if** ((as = cells) != **null** || !casBase(b = base, b + x)) {

**boolean** uncontended = **true**;

**if** (as == **null** || (m = as.length - 1) < 0 ||

(a = as[getProbe() & m]) == **null** ||

!(uncontended = a.cas(v = a.value, v + x)))

longAccumulate(x, **null**, uncontended);

}

}

**public** **void** **accumulate**(**long** x) {

Cell[] as; **long** b, v, r; **int** m; Cell a;

**if** ((as = cells) != **null** ||

(r = function.applyAsLong(b = base, x)) != b && !casBase(b, r)) {

**boolean** uncontended = **true**;

**if** (as == **null** || (m = as.length - 1) < 0 ||

(a = as[getProbe() & m]) == **null** ||

!(uncontended =

(r = function.applyAsLong(v = a.value, x)) == v ||

a.cas(v, r)))

longAccumulate(x, function, uncontended);

}

}

另外前者调用 longAccumulate 时候传递到是 function，而后者是 null，从下面代码可知当 fn 为 null时候就是使用 v+x 加法运算这时候就等价于 LongAdder，fn 不为 null 时候则使用传递的 fn 函数计算，如果 fn 为加法则等价于 LongAdder；

**else** **if** (casBase(v = base, ((fn == **null**) ? v + x :

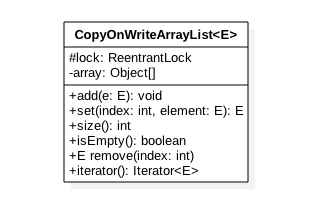
fn.applyAsLong(v, x))))

**break**; // Fall back on using base

### 四、JUC 并发包中 CopyOnWriteArrayList 原理探究

#### 4.1 介绍

并发包中并发 List 只有 CopyOnWriteArrayList 这一个，CopyOnWriteArrayList 是一个线程安全的 ArrayList，对其进行的修改操作和元素迭代操作都是在底层创建一个拷贝的数组（快照）上进行的，也就是写时拷贝策略。



如上 CopyOnWriteArrayList 的类图，每个 CopyOnWriteArrayList 对象里面有一个 array 数组对象用来存放具体元素，ReentrantLock 独占锁对象用来保证同时只有一个线程对 array 进行修改，这里只要记得 ReentrantLock 是独占锁，同时只有一个线程可以获取就可以了，后面的 chat 会专门对 JUC 中锁进行介绍的。

考虑如果让我们自己做一个写时拷贝的线程安全的list我们会怎么做，有哪些点需要考虑那？

* list 何时初始化，初始化 list 元素个数为多少，list 是有限大小？
* 如何保证线程安全，比如多个线程进行读写时候如何保证是线程安全的
* 如何保证使用迭代器遍历list时候的数据一致性

下面就看看 CopyOnWriteArrayList 的作者 Doug Lea 是如何设计的。

#### 4.2 主要函数源码讲解

##### 4.2.1 初始化

首先看下无参构造，如下代码内部创建了一个大小为 0 的 Object 数据作为 array 的初始值

**public** **CopyOnWriteArrayList**() {

setArray(**new** Object[0]);

}

然后看下有参构造函数

//创建一个list，其内部元素是入参toCopyIn的拷贝

**public** **CopyOnWriteArrayList**(E[] toCopyIn) {

setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, Object[].class));

}

//入参为集合，拷贝集合里面元素到本list

**public** **CopyOnWriteArrayList**(Collection<? extends E> c) {

Object[] elements;

**if** (c.getClass() == CopyOnWriteArrayList.class)

elements = ((CopyOnWriteArrayList<?>)c).getArray();

**else** {

elements = c.toArray();

// c.toArray might (incorrectly) not return Object[] (see 6260652)

**if** (elements.getClass() != Object[].class)

elements = Arrays.copyOf(elements, elements.length, Object[].class);

}

setArray(elements);

}

##### 4.2.2 添加元素

CopyOnWriteArrayList 中添加元素函数有 add(E e) ， add(int index , E element) ，addIfAbsent(E e) ， addAllAbsent(Collection<? extends E> c) 等操作，原理一致，所以本节以 add(E e) 为例来讲解。

**public** **boolean** **add**(E e) {

//加独占锁（1）

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

//获取array(2)

Object[] elements = getArray();

//拷贝array到新数组，添加元素到新数组(3)

**int** len = elements.length;

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

newElements[len] = e;

//使用新数组替换添加前的数组(4)

setArray(newElements);

**return** **true**;

} **finally** {

//释放独占锁(5)

lock.unlock();

}

}

如上代码，调用add方法的线程会首先执行（1）去获取独占锁，如果多个线程都调用add则只有一个线程会获取该锁，其它线程会被阻塞挂起直到锁被释放。

所以一个线程获取到锁后就保证了在该线程添加元素过程中其它线程不会对array进行修改。

线程获取锁后执行（2）获取array，然后执行（3）拷贝 array 到一个新数组（从这里可以知道新数组的大小是原来数组大小增加 1，所以 CopyOnWriteArrayList 是无界 list），并把要新增的元素添加到新数组。

然后执行到（4）把新数组替换元数组，然后返回前释放锁，由于加了锁，所以整个 add 过程是个原子性操作，需要注意的是添加元素时候首先是拷贝了一个快照，然后在快照上进行的添加，而不是直接在原来数组上进行。

##### 4.2.3 获取指定位置元素

* E get(int index) 获取下标为 index 的元素，如果元素不存在会抛出IndexOutOfBoundsException 异常

**public** E **get**(**int** index) {

**return** get(getArray(), index);

}

**final** Object[] getArray() {

**return** array;

}

**private** E **get**(Object[] a, **int** index) {

**return** (E) a[index];

}

如上代码获取指定位置的元素分为两步，首先获取到当前 list 里面的 array 数组这里称为步骤 A，然后通过随机访问的下标方式访问指定位置的元素这里称为步骤 B。

从代码可以看到整个过程中并没有加锁，这就可能会导致当执行完步骤A后执行步骤 B 前，另外一个线程 C 进行了修改操作比如 remove 操作，就会进行写时拷贝删除当前 get 方法要访问的元素，并且修改当前 list 的 array 为新数组。而这之后步骤 B 可能才开始执行，步骤 B 操作的是线程 C 删除元素前的一个快照数组（因为步骤 A 让 array 指向的是原来的数组），所以虽然线程 C 已经删除了 index 处的元素，但是步骤 B 还是返回 index 出的元素，这其实就是写时拷贝策略带来的弱一致性。

##### 4.2.4 修改指定元素

修改 list 中指定元素的值，如果指定位置的元素不存在则抛出 IndexOutOfBoundsException 异常，代码如下：

**public** E **set**(**int** index, E element) {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

Object[] elements = getArray();

E oldValue = get(elements, index);

**if** (oldValue != element) {

**int** len = elements.length;

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

newElements[index] = element;

setArray(newElements);

} **else** {

// Not quite a no-op; ensures volatile write semantics

setArray(elements);

}

**return** oldValue;

} **finally** {

lock.unlock();

}

}

如上代码首先获取了独占锁控制了其它线程对 array 数组的修改，然后获取当前数组，并调用 get 方法获取指定位置元素。

如果指定位置元素与新值不一致则创建新数组并拷贝元素，在新数组上修改指定位置元素值并设置新数组到 array。

如果指定位置元素与新值一样则为了保证 volatile 语义还是需要重新设置下 array，虽然 array 内容并没有改变（为了保证 volatile 语义是考虑到 set 方法本身应该提供 volatile 的语义）。

##### 4.2.5 删除元素

删除 list 里面指定的元素，主要方法如下：

* E remove(int index)
* boolean remove(Object o)
* boolean remove(Object o, Object[] snapshot, int index) 等方法，原理一致，这里讲解下 remove(int index) 方法

**public** E **remove**(**int** index) {

//获取独占锁

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

//获取数组

Object[] elements = getArray();

**int** len = elements.length;

//获取指定元素

E oldValue = get(elements, index);

**int** numMoved = len - index - 1;

//如果要删除的是最后一个元素

**if** (numMoved == 0)

setArray(Arrays.copyOf(elements, len - 1));

**else** {

//分两次拷贝除删除后的元素到新数组

Object[] newElements = **new** Object[len - 1];

System.arraycopy(elements, 0, newElements, 0, index);

System.arraycopy(elements, index + 1, newElements, index,

numMoved);

//使用新数组代替老的

setArray(newElements);

}

**return** oldValue;

} **finally** {

//释放锁

lock.unlock();

}

}

如上代码其实和新增元素时候类似，首先获取独占锁保证删除数据期间其它线程不能对 array 进行修改，然后获取数据中要给删除的元素，并把剩余的原始拷贝到新数组后把新数组替换原来的数组，最后在返回前释放锁。

##### 4.2.6 弱一致性的迭代器

遍历列表元素可以使用迭代器进行迭代操作，讲解什么是迭代器的弱一致性前先上一个例子说明下迭代器的使用。

**public** **static** **void** **main**( String[] args )

{

CopyOnWriteArrayList<String> arrayList = **new** CopyOnWriteArrayList<>();

arrayList.add("hello");

arrayList.add("alibaba");

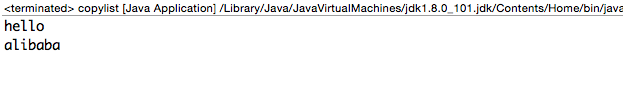
Iterator<String> itr = arrayList.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

输出： 

其中迭代器的 hasNext 方法用来判断是否还有元素，next 方法则是具体返回元素。好了，下面来看 CopyOnWriteArrayList 中迭代器是弱一致性，所谓弱一致性是指当返回迭代器后，其它线程对 list 的增删改对迭代器不可见，无感知，下面看看是如何做到的。

**public** Iterator<E> **iterator**() {

**return** **new** COWIterator<E>(getArray(), 0);

}

**static** **final** **class** **COWIterator**<**E**> **implements** **ListIterator**<**E**> {

//array的快照版本

**private** **final** Object[] snapshot;

//数组下标

**private** **int** cursor;

//构造函数

**private** **COWIterator**(Object[] elements, **int** initialCursor) {

cursor = initialCursor;

snapshot = elements;

}

//是否遍历结束

**public** **boolean** **hasNext**() {

**return** cursor < snapshot.length;

}

//获取元素

**public** E **next**() {

**if** (! hasNext())

**throw** **new** NoSuchElementException();

**return** (E) snapshot[cursor++];

}

如上代码当调用 iterator() 方法获取迭代器时候实际是返回一个 COWIterator 对象，COWIterator的snapshot 变量保存了当前list的内容，cursor 是遍历 list 数据的下标。

这里为什么说 snapshot 是 list 的快照那，明明是指针传递的引用哇，而不是拷贝。如果在该线程使用返回的迭代器遍历元素的过程中，其它线程没有对list进行增删改，那么 snapshot 本身就是 list 的 array，因为它们是引用关系。

但是如果在遍历期间其它线程对该 list 进行了增删改，那么snapshot 就是快照了，因为增删改后 list 里面的数组被新数组替换了，这时候老数组只有被 snapshot 所引用，所以这也说明获取迭代器后，使用该迭代器进行变量元素时候，其它线程对该list进行的增删改不可见，因为它们操作的是两个不同的数组，这也就是弱一致性的达成。

下面通过一个例子来演示多线程下迭代器的弱一致性的效果：

**public** **class** **copylist** {

**private** **static** **volatile** CopyOnWriteArrayList<String> arrayList = **new** CopyOnWriteArrayList<>();

**public** **static** **void** **main**( String[] args ) **throws** InterruptedException

{

arrayList.add("hello");

arrayList.add("alibaba");

arrayList.add("welcome");

arrayList.add("to");

arrayList.add("hangzhou");

Thread threadOne = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

//修改list中下标为1的元素为baba

arrayList.set(1, "baba");

//删除元素

arrayList.remove(2);

arrayList.remove(3);

}

});

//保证在修改线程启动前获取迭代器

Iterator<String> itr = arrayList.iterator();

//启动线程

threadOne.start();

//等在子线程执行完毕

threadOne.join();

//迭代元素

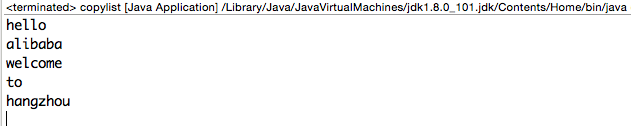
**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

输出结果： 

如上代码 main 函数首先初始化了 arrayList，然后在启动线程前获取到了 arrayList 迭代器，子线程 threadOne 启动后首先修改了 arrayList 的第一个元素的值，然后删除了 arrayList 中下标为2，3 的元素。

主线程等子线程执行完毕后使用获取的迭代器遍历数组元素，从打印结果知道在子线程里面进行的操作一个都没有生效，这就是迭代器弱一致性的效果，需要注意的是获取迭代器必须在子线程操作之前进行。

注：CopyOnWriteArrayList 使用写时拷贝的策略来保证list的一致性，而获取-拷贝-写入三步并不是原子性的，所以在修改增删改的过程中都使用了独占锁，保证了同时只有一个线程才能对list数组进行修改。另外 CopyOnWriteArrayList 提供了弱一致性的迭代器，保证在获取迭代器后，其它线程对list的修改不可见，迭代器遍历时候的数组是获取迭代器时候的一个快照，另外并发包中 CopyOnWriteArraySet 底层就是使用它进行实现，感兴趣的可以去翻翻看。

### 五、总结

本文首先讲解了rt.jar 中 Unsafe和 LockSupport 类主要函数，这两个类是 JUC 的基础，为研究 JUC 源码的实现奠定基础。然后讲解 JDK8 新增原子操作类 LongAdder 实现原理，并讲解了它如何解决 AtomicLong 的缺点的，LongAdder 和 LongAccumulator 是什么关系？最后讲解了JUC 并发包中并发组件 CopyOnWriteArrayList 是如何通过写时拷贝实现并发安全的 List，何为弱一致性。

# Java 并发编程之美：并发编程高级篇之三

### 一、前言

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之三，主要讲解锁，内容如下：（建议先阅读 [Java 并发编程之美：并发编程高级篇之二](http://gitbook.cn/gitchat/activity/5ac034eb67727920220cb904) ）

抽象同步队列 AQS (AbstractQueuedSynchronizer) 概述，AQS 是实现同步的基础组件，并发包中锁的实现底层就是使用 AQS 实现，虽然大多数开发者可能从来不会直接用到 AQS，但是知道其原理对于架构设计还是很有帮助的。

独占锁 ReentrantLock 原理探究，ReentrantLock 是可重入的独占锁或者叫做排它锁，同时只能有一个线程可以获取该锁，其实现分为公平与非公平的独占锁。

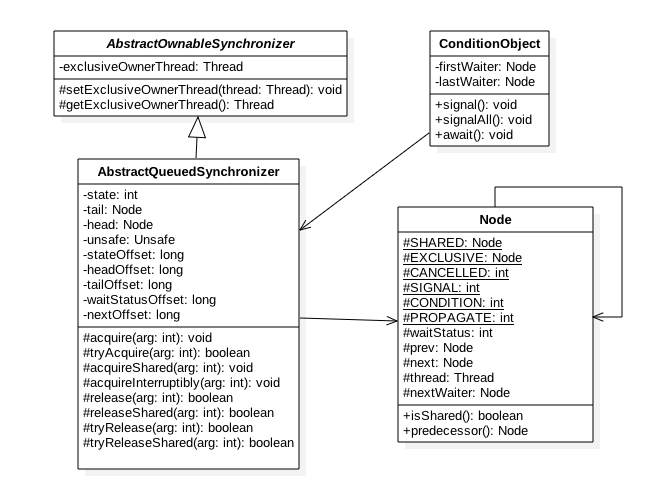
读写锁 ReentrantReadWriteLock 原理，ReentrantLock 是独占锁，同时只有一个线程可以获取该锁，而实际情况下会有写少读多的场景，显然 ReentrantLock 满足不了需求，所以 ReentrantReadWriteLock 应运而生，本文来介绍读写分离锁的实现。

StampedLock 锁原理探究，StampedLock 是并发包里面 jdk8 版本新增的一个锁，该锁提供了三种模式的读写控制。

### 二、抽象同步队列 AQS 概述

#### 2.1 AQS - 锁的底层支持

AbstractQueuedSynchronizer 抽象同步队列, 简称 AQS，是实现同步器的基础组件，并发包中锁的实现底层就是使用 AQS 实现，另外大多数开发者可能从来不会直接用到 AQS，但是知道其原理对于架构设计还是很有帮助的，下面看下 AQS 的类图结构：



AQS 是一个 FIFO 的双向队列，内部通过节点 head 和 tail 记录队首和队尾元素，队列元素类型为 Node。其中 Node 中 thread 变量用来存放进入 AQS 队列里面的线程；Node 节点内部 SHARED 用来标记该线程是获取共享资源时候被阻塞挂起后放入 AQS 队列，EXCLUSIVE 标示线程是获取独占资源时候被挂起后放入 AQS 队列；waitStatus 记录当前线程等待状态，分别为 CANCELLED（线程被取消了），SIGNAL（线程需要被唤醒），CONDITION（线程在条件队列里面等待），PROPAGATE（释放共享资源时候需要通知其它节点）；prev 记录当前节点的前驱节点，next 记录当前节点后继节点。

AQS 中维持了一个单一的状态信息 state, 可以通过 getState,setState,compareAndSetState 函数修改其值；对于 ReentrantLock 的实现来说，state 可以用来表示当前线程获取锁的可重入次数；对应读写锁 ReentrantReadWriteLock 来说 state 的高 16 位表示读状态也就是获取该读锁的次数，低 16 位表示获取到写锁的线程的可重入次数；对于 semaphore 来说 state 用来表示当前可用信号的个数；对于 FutuerTask 来说，state 用来表示任务状态（例如还没开始，运行，完成，取消）；对应 CountDownlatch 和 CyclicBarrie 来说 state 用来表示计数器当前的值。

AQS 有个内部类 ConditionObject 是用来结合锁实现线程同步，ConditionObject 可以直接访问 AQS 对象内部的变量，比如 state 状态值和 AQS 队列；ConditionObject 是条件变量，每个条件变量对应着一个条件队列 (单向链表队列)，用来存放调用条件变量的 await() 方法后被阻塞的线程，如类图，这个条件队列的头尾元素分别为 firstWaiter 和 lastWaiter。

对于 AQS 来说线程同步的关键是对状态值 state 进行操作，根据 state 是否属于一个线程，操作 state 的方式分为独占模式和共享模式。 独占方式下获取和释放资源使用方法为：

**void** **acquire**(**int** arg)**void** **acquireInterruptibly**(**int** arg)**boolean** **release**(**int** arg)

共享模式下获取和释放资源方法为：

**void** **acquireShared**(**int** arg)**void** **acquireSharedInterruptibly**(**int** arg)**boolean** **releaseShared**(**int** arg)

对于独占方式获取的资源是与具体线程绑定的，就是说如果一个线程获取到了资源，就会标记是那个线程获取到了，其它线程尝试操作 state 获取资源时候发现当前该资源不是自己持有的，就会获取失败后被阻塞；比如独占锁 ReentrantLock 的实现，当一个线程获取了 ReentrantLock 的锁后，AQS 内部会首先使用 CAS 操作把 state 状态值从 0 变为 1，然后设置当前锁的持有者为当前线程，当该线程再次获取锁时候发现当前线程就是锁的持有者则会把状态值从 1 变为 2 也就是设置可重入次数，当另外一个线程获取锁的时候发现自己并不是该锁的持有者就会被放入 AQS 阻塞队列后挂起。

对应共享操作方式资源是与具体线程不相关的，多个线程去请求资源时候是通过 CAS 方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次获取时候如果当前资源还能满足它的需要，则当前线程只需要使用 CAS 方式进行获取即可，共享模式下并不需要记录那个线程获取了资源；比如 Semaphore 信号量，当一个线程通过 acquire() 方法获取一个信号量时候，会首先看当前信号量个数是否满足需要，不满足则把当前线程放入阻塞队列，如果满足则通过自旋 CAS 获取信号量。

对应独占模式的获取与释放资源流程：

1）当一个线程调用 acquire(int arg) 方法获取独占资源时候，会首先使用 tryAcquire 尝试获取资源，具体是设置状态变量 state 的值，成功则直接返回。失败则将当前线程封装为类型为 Node.EXCLUSIVE 的 Node 节点后插入到 AQS 阻塞队列尾部，并调用 LockSupport.park(this) 挂起当前线程。

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

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

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

selfInterrupt();

}

2）当一个线程调用 release(int arg) 时候会尝试使用, tryRelease 操作释放资源，这里是设置状态变量 state 的值，然后调用 LockSupport.unpark(thread) 激活 AQS 队列里面最早被阻塞的线程 (thread)。被激活的线程则使用 tryAcquire 尝试看当前状态变量 state 的值是否能满足自己的需要，满足则该线程被激活然后继续向下运行，否者还是会被放入 AQS 队列并被挂起。

**public** **final** **boolean** **release**(**int** arg) {

**if** (tryRelease(arg)) {

Node h = head;

**if** (h != **null** && h.waitStatus != 0)

unparkSuccessor(h);

**return** **true**;

}

**return** **false**;

}

需要注意的 AQS 类并没有提供可用的 tryAcquire 和 tryRelease，正如 AQS 是锁阻塞和同步器的基础框架，tryAcquire 和 tryRelease 需要有具体的子类来实现。子类在实现 tryAcquire 和 tryRelease 时候要根据具体场景使用 CAS 算法尝试修改状态值 state, 成功则返回 true, 否者返回 false。子类还需要定义在调用 acquire 和 release 方法时候 state 状态值的增减代表什么含义。

比如继承自 AQS 实现的独占锁 ReentrantLock，定义当 status 为 0 的时候标示锁空闲，为 1 的时候标示锁已经被占用，在重写 tryAcquire 时候，内部需要使用 CAS 算法看当前 status 是否为 0，如果为 0 则使用 CAS 设置为 1，并设置当前线程的持有者为当前线程，并返回 true, 如果 CAS 失败则 返回 false。

比如继承自 AQS 实现的独占锁实现 tryRelease 时候，内部需要使用 CAS 算法把当前 status 值从 1 修改为 0，并设置当前锁的持有者为 null，然后返回 true, 如果 cas 失败则返回 false。

对应共享资模式的获取与释放流程：

1）当线程调用 acquireShared(int arg) 获取共享资源时候，会首先使用 tryAcquireShared 尝试获取资源，具体是设置状态变量 state 的值，成功则直接返回。失败则将当前线程封装为类型为 Node.SHARED 的 Node 节点后插入到 AQS 阻塞队列尾部，并使用 LockSupport.park(this) 挂起当前线程。

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

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

doAcquireShared(arg);

}

2）当一个线程调用 releaseShared(int arg) 时候会尝试使用, tryReleaseShared 操作释放资源，这里是设置状态变量 state 的值，然后使用 LockSupport.unpark（thread）激活 AQS 队列里面最早被阻塞的线程 (thread)。被激活的线程则使用 tryReleaseShared 尝试看当前状态变量 state 的值是否能满足自己的需要，满足则该线程被激活然后继续向下运行，否者还是会被放入 AQS 队列并被挂起。

**public** **final** **boolean** **releaseShared**(**int** arg) {

**if** (tryReleaseShared(arg)) {

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

同理需要注意的 AQS 类并没有提供可用的 tryAcquireShared 和 tryReleaseShared，正如 AQS 是锁阻塞和同步器的基础框架，tryAcquireShared 和 tryReleaseShared 需要有具体的子类来实现。子类在实现 tryAcquireShared 和 tryReleaseShared 时候要根据具体场景使用 CAS 算法尝试修改状态值 state, 成功则返回 true, 否者返回 false。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryAcquireShared 时候，首先看写锁是否被其它线程持有，如果是则直接返回 false，否者使用 cas 递增 status 的高 16 位，在 ReentrantReadWriteLock 中 status 的高 16 为获取读锁的次数。

比如继承自 AQS 实现的读写锁 ReentrantReadWriteLock 里面的读锁在重写 tryReleaseShared 时候，内部需要使用 CAS 算法把当前 status 值的高 16 位减一，然后返回 true, 如果 cas 失败则返回 false。

基于 AQS 实现的锁除了需要重写上面介绍的方法，还需要重写 isHeldExclusively 方法用来判断锁是被当前线程独占还是被共享。

另外也许你会好奇独占模式下的

**void** **acquire**(**int** arg)**void** **acquireInterruptibly**(**int** arg)

和共享模式下获取资源的：

**void** **acquireShared**(**int** arg) **void** **acquireSharedInterruptibly**(**int** arg)

这两套函数都有一个带有 Interruptibly 关键字的函数，那么带有这个关键字的和不带的有什么区别那？

其实不带 Interruptibly 关键字的方法是说不对中断进行响应，也就是线程在调用不带 Interruptibly 关键字的方法在获取资源的时候或者获取资源失败被挂起时候，其他线程中断了该线程，那么该线程不会因为被中断而抛出异常，还是继续获取资源或者被挂起，也就是不对中断进行响应，忽略中断。

带 Interruptibly 关键字的方法是说对中断进行响应，也就是也就是线程在调用带 Interruptibly 关键字的方法在获取资源的时候或者获取资源失败被挂起时候，其他线程中断了该线程，那么该线程会抛出 InterruptedException 异常而返回。

本节最后我们来看看 AQS 提供的队列是如何维护的，主要看入队操作

* 入队操作： 当一个线程获取锁失败后该线程会被转换为 Node 节点，然后就会使用 enq(final Node node) 方法插入该节点到 AQS 的阻塞队列，

**private** Node **enq**(**final** Node node) {

**for** (;;) {

Node t = tail;//(1)

**if** (t == **null**) { // Must initialize

**if** (compareAndSetHead(**new** Node()))//(2)

tail = head;

} **else** {

node.prev = t;//(3)

**if** (compareAndSetTail(t, node)) {//(4)

t.next = node;

**return** t;

}

}

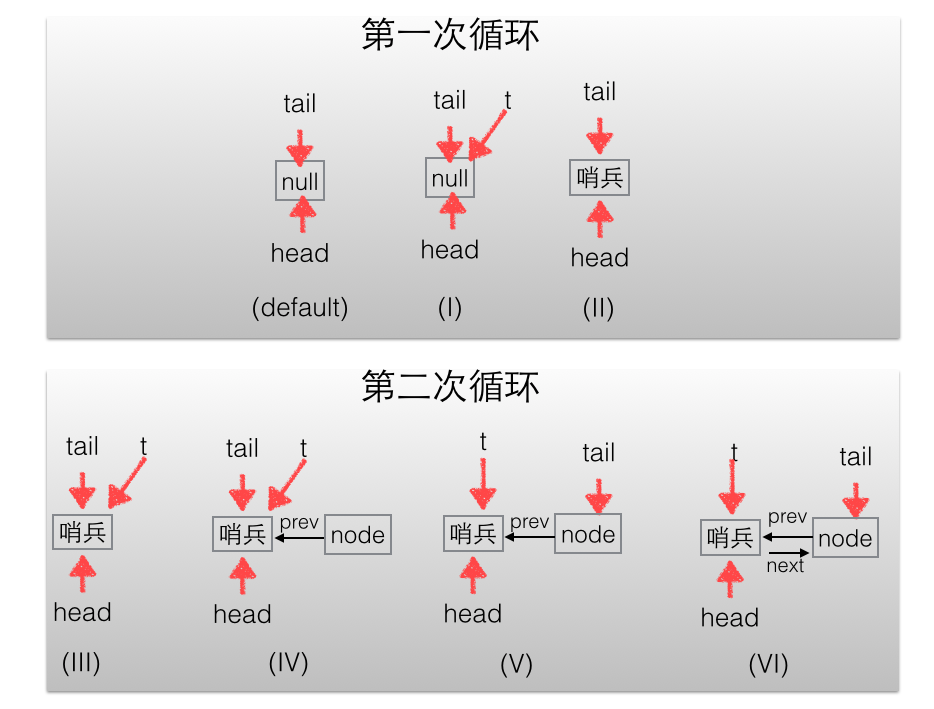
}

}

下面结合代码和下面的节点图来讲解下，如上代码第一次循环当要在 AQS 队列尾部插入元素时候，AQS 队列状态为图（default), 也就是队列头尾节点都指向 null；当执行代码（1）后节点 t 指向了尾部节点，这时候队列状态如图（I）。

可知这时候 t 为 null，则执行代码（2）使用 CAS 算法设置一个哨兵节点为头结点，如果 CAS 设置成功，然后让尾部节点也指向哨兵节点，这时候队列状态如图（II）。

到现在只是插入了一个哨兵节点，还需要插入的 node 节点，所以第二次循环后执行到步骤（1），这时候队列状态如图（III）；然后执行代码（3）设置 node 的前驱节点为尾部节点，这时候队列状态图如图（IV）；然后通过 CAS 算法设置 node 节点为尾部节点，CAS 成功后队列状态图为（V）；CAS 成功后在设置原来的尾部节点的后驱节点为 node, 这时候就完成了双向链表的插入了，这时候队列状态为图（VI）。



#### 2.2 AQS - 条件变量的支持

正如基础篇讲解的 notify 和 wait 是配合 synchronized 内置锁实现线程间同步基础设施，条件变量的 signal 和 await 方法是用来配合锁（使用 AQS 实现的锁）实现线程间同步的基础设施。

在基础篇讲解了在调用共享变量的 notify 和 wait 方法前必须先获取该共享变量的内置锁，同理在调用条件变量的 signal 和 await 方法前必须先获取条件变量对应的锁。

说了那么多，到底什么是条件变量那？如何使用那？不急，下面看一个例子：

ReentrantLock **lock** = **new** ReentrantLock();//(1)

Condition condition = **lock**.newCondition();//(2)

**lock**.**lock**();//(3)**try** {

System.**out**.println("begin wait");

condition.**await**();//(4)

System.**out**.println("end wait");

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**lock**.unlock();//(5)

}

**lock**.**lock**();//(6)**try** {

System.**out**.println("begin signal");

condition.signal();//(7)

System.**out**.println("end signal");

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**lock**.unlock();//(8)

}

如上代码（1）创建了一个独占锁 ReentrantLock 的对象，ReentrantLock 是基于 AQS 实现的锁。

代码（2）使用创建的 lock 对象的 newCondition（）方法创建了一个 ConditionObject 变量，这个变量就是 lock 锁对应的一个条件变量。需要注意的是一个 Lock 对象可以创建多个条件变量。

代码（3）首先获取了独占锁，代码（4) 则调用了条件变量的 await（）方法阻塞挂起了当前线程，当其它线程调用了条件变量的 signal 方法时候，被阻塞的线程才会从 await 处返回，需要注意的是和调用 Object 的 wait 方法一样，如果在没有获取到锁前调用了条件变量的 await 方法会抛出 java.lang.IllegalMonitorStateException 异常。

代码（5) 则释放了获取的锁。

其实这里的 lock 对象等价于 synchronized 加上共享变量，当调用 lock.lock（）方法就相当于进入了 synchronized 块（获取了共享变量的内置锁），当调用 lock.unLock() 方法时候就相当于退出了 synchronized 块。 当调用条件变量的 await() 方法时候就相当于调用了共享变量的 wait() 方法，当调用了条件变量的 signal 方法时候就相当于调用了共享变量的 notify() 方法。当调用了条件变量的 signalAll（）方法时候就相当于调用了共享变量的 notifyAll() 方法。

有了上面的解释相信大家对条件变量是什么，用来做什么用的有了一定的认识了。

上面通过 lock.newCondition() 作用其实是 new 了一个 AQS 内部声明的 ConditionObject 对象，ConditionObject 是 AQS 的内部类，可以访问到 AQS 内部的变量（例如状态变量 status 变量）和方法。对应每个条件变量内部维护了一个条件队列，用来存放当调用条件变量的 await() 方法被阻塞的线程。注意这个条件队列和 AQS 队列不是一回事情。

如下代码，当线程调用了条件变量的 await() 方法时候（事先必须先调用了锁的 lock() 方法获取锁），内部会构造一个类型为 Node.CONDITION 的 node 节点，然后插入该节点到条件队列末尾，然后当前线程会释放获取的锁（也就是会操作锁对应的 status 变量的值），并被阻塞挂起。这时候如果有其它线程调用了 lock.lock() 尝试获取锁时候，就会有一个线程获取到锁，如果获取到锁的线程有调用了条件变量的 await（）方法，则该线程也会被放入条件变量的阻塞队列，然后释放获取到的锁，阻塞到 await() 方法处。

**public** **final** **void** **await**() **throws** InterruptedException {

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

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

//创建新的node,并插入到条件队列末尾（9）

Node node = addConditionWaiter();

//释放当前线程获取的锁（10）

**int** savedState = fullyRelease(node);

**int** interruptMode = 0;

//调用park方法阻塞挂起当前线程（11）

**while** (!isOnSyncQueue(node)) {

LockSupport.park(**this**);

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

**break**;

}

...

}

如下代码，当另外一个线程调用了条件变量的 signal 方法时候（事先必须先调用了锁的 lock() 方法获取锁），内部会把条件队列里面队头的一个线程节点从条件队列里面移除后放入到 AQS 的阻塞队列里面，然后激活这个线程。

**public** **final** **void** **signal**() {

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

Node first = firstWaiter;

**if** (first != **null**)

//移动条件队列队头元素到AQS队列

doSignal(first);

}

需要注意的是 AQS 只提供了 ConditionObject 的实现，并没有提供 newCondition 函数来 new 一个 ConditionObject 对象，需要由 AQS 的子类来提供 newCondition 函数。

下面来看下当一个线程调用条件变量的 await() 方法被阻塞后，如何放入的条件队列。

**private** Node **addConditionWaiter**() {

Node t = lastWaiter;

...

//(1)

Node node = **new** Node(Thread.currentThread(), Node.CONDITION);

//(2)

**if** (t == null)

firstWaiter = node;

**else**

t.nextWaiter = node;//(3)

lastWaiter = node;//(4)

**return** node;

}

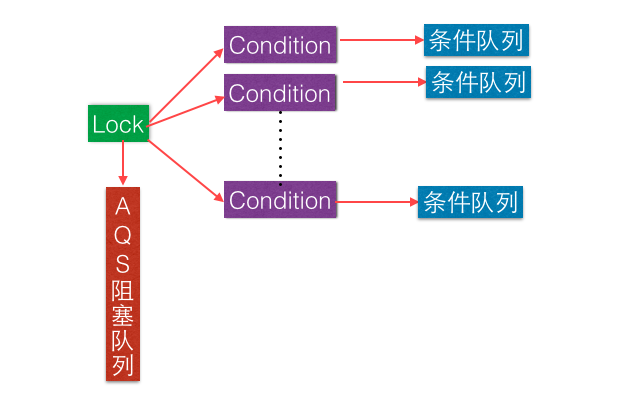
如上代码（1）首先根据当前线程创建一个类型为 Node.CONDITION 的节点，然后通过步骤（2）（3）（4）在单向条件队列尾部插入一个元素。

注：当多个线程同时调用 lock.lock() 获取锁的时候，同时只有一个线程获取到了该锁，其他线程会被转换为 Node 节点插入到 lock 锁对应的 AQS 阻塞队列里面，并做自旋 CAS 尝试获取锁；

如果获取到锁的线程又调用了对应的条件变量的 await() 方法，则该线程会释放获取到的锁，并被转换为 Node 节点插入到条件变量对应的条件队列里面；

这时候因为调用 lock.lock() 方法被阻塞到 AQS 队列里面的一个线程会获取到被释放的锁，如果该线程也调用了条件变量的 await（）方法则该线程也会被放入条件变量的条件队列。

当另外一个线程调用了条件变量的 signal() 或者 signalAll() 方法时候，会把条件队列里面的一个或者全部 Node 节点移动到 AQS 的阻塞队列里面，等待时机获取锁。

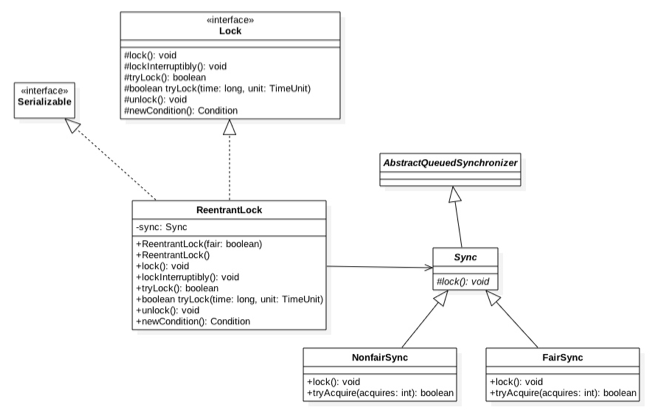


最后一个图总结下，一个锁对应有一个 AQS 阻塞队列，对应多个条件变量，每个条件变量有自己的一个条件队列。

### 三、独占锁 ReentrantLock 原理

#### 3.1 类图结构简介

ReentrantLock 是可重入的独占锁，同时只能有一个线程可以获取该锁，其它获取该锁的线程会被阻塞后放入该锁的 AQS 阻塞队列里面。首先一览 ReentrantLock 的类图以便对它的实现有个大致了解



从类图可知 ReentrantLock 最终还是使用 AQS 来实现，并且根据参数决定内部是公平还是非公平锁，默认是非公平锁：

**public** **ReentrantLock**() {

sync = **new** NonfairSync();

}

**public** **ReentrantLock**(**boolean** fair) {

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

}

其中类 Sync 直接继承自 AQS，它的子类 NonfairSync 和 FairSync 分别实现了获取锁的公平和非公平策略。

在这里 AQS 的状态值 state 代表线程获取该锁的可重入次数，默认情况下 state 的值为 0 标示当前锁没有被任何线程持有，当一个线程第一次获取该锁时候会使用尝试使用 CAS 设置 state 的值为 1，如果 CAS 成功则当前线程获取了该锁，然后记录该锁的持有者为当前线程，在该线程没有释放锁第二次获取改锁后状态值被为 2，这就是可重入次数，在该线程释放该锁的时候，会尝试使用 CAS 让状态值减一，如果减一后状态值为 0 则当前线程释放该锁。

#### 3.2 获取锁

* void lock() 当一个线程调用该方法，说明该线程希望获取该锁，如果锁当前没有被其它线程占用并且当前线程之前没有获取该锁，则当前线程会获取到该锁，然后设置当前锁的拥有者为当前线程，并设置 AQS 的状态值为 1 后直接返回。 如果当前线程之前已经获取过该锁，则这次只是简单的把 AQS 的状态值 status 加 1 后返回。 如果该锁已经被其它线程持有，则调用该方法的线程会被放入 AQS 队列后阻塞挂起。

**public** **void** **lock**() {

sync.**lock**();

}

如上代码 ReentrantLock 的 lock() 是委托给了 sync 类，根据创建 ReentrantLock 时候构造函数选择 sync 的实现是 NonfairSync 或者 FairSync，这里先看 sync 的子类 NonfairSync 的情况，也就是非公平锁的时候：

**final** **void** **lock**() {

//（1）CAS设置状态值

**if** (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

**else**

//（2）调用AQS的acquire方法

acquire(1);

}

如上代码（1）因为默认 AQS 的状态值为 0，所以第一个调用 Lock 的线程会通过 CAS 设置状态值为 1，CAS 成功则表示当前线程获取到了锁，然后 setExclusiveOwnerThread 设置了该锁持有者是当前线程。

如果这时候有其它线程调用 lock 方法企图获取该锁执行代码（1）CAS 会失败，然后会调用 AQS 的 acquire 方法，这里注意传递参数为 1，这里在贴下 AQS 的 acquire 骨干代码：

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

//(3)调用ReentrantLock重写的tryAcquire方法

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

// tryAcquiref返回false会把当前线程放入AQS阻塞队列

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

selfInterrupt();

}

之前说过 AQS 并没有提供可用的 tryAcquire 方法，tryAcquire 方法需要子类自己定制化，所以这里代码（3）会调用 ReentrantLock 重写的 tryAcquire 方法代码，这里先看下非公平锁的代码如下：

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

**return** nonfairTryAcquire(acquires);

}

**final** **boolean** **nonfairTryAcquire**(**int** acquires) {

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

**int** c = getState();

//（4）当前AQS状态值为0

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

**if** (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

**return** **true**;

}

}//(5)当前线程是该锁持有者

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

**int** nextc = c + acquires;

**if** (nextc < 0) // overflow

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

setState(nextc);

**return** **true**;

}//(6)

**return** **false**;

}

如上代码（4）会看当前锁的状态值是否为 0，为 0 则说明当前该锁空闲，那么就尝试 CAS 获取该锁（尝试将 AQS 的状态值从 0 设置为 1），并设置当前锁的持有者为当前线程返回返回 true。

如果当前状态值不为 0 则说明该锁已经被某个线程持有，所以代码（5）看当前线程是否是该锁的持有者，如果当前线程是该锁持有者，状态值增加 1 然后返回 true。

如果当前线程不是锁的持有者则返回 false, 然后会被放入 AQS 阻塞队列。

这里介绍完了非公平锁的实现代码，回过头来看看非公平在这里是怎么体现的，首先非公平是说先尝试获取锁的线程并不一定比后尝试获取锁的线程优先获取锁。

这里假设线程 A 调用 lock（）方法时候执行到了 nonfairTryAcquire 的代码（4）发现当前状态值不为 0，所以执行代码（5）发现当前线程不是线程持有者，则执行代码（6）返回 false，然后当前线程会被放入了 AQS 阻塞队列。

这时候线程 B 也调用了 lock() 方法执行到 nonfairTryAcquire 的代码（4）时候发现当前状态值为 0 了（假设占有该锁的其它线程释放了该锁）所以通过 CAS 设置获取到了该锁。而明明是线程 A 先请求获取的该锁那，这就是非公平锁的实现，这里线程 B 在获取锁前并没有看当前 AQS 队列里面是否有比自己请求该锁更早的线程，而是使用了抢夺策略。

那么下面看看公平锁是怎么实现公平的，公平锁的话只需要看 FairSync 重写的 tryAcquire 方法

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

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

**int** c = getState();

//（7）当前AQS状态值为0

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

//（8）公平性策略

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

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

**return** **true**;

}

}

//（9）当前线程是该锁持有者

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

**int** nextc = c + acquires;

**if** (nextc < 0)

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

setState(nextc);

**return** **true**;

}//(10)

**return** **false**;

}

}

如上代码公平性的 tryAcquire 策略与非公平的类似，不同在于代码（8）处在设置 CAS 前添加了 hasQueuedPredecessors 方法，该方法是实现公平性的核心代码，代码如下：

**public** **final** **boolean** **hasQueuedPredecessors**() {

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

**return** h != t &&

((s = h.next) == **null** || s.thread != Thread.currentThread());

}

如上代码如果当前线程节点有前驱节点则返回 true，否者如果当前 AQS 队列为空或者当前线程节点是 AQS 的第一个节点则返回 false.

其中如果 h==t 则说明当前队列为空则直接返回 false，如果 h!=t 并且 s==null 说明有一个元素将要作为 AQS 的第一个节点入队列，那么返回 true, 如果 h!=t 并且 s!=null 并且 s.thread != Thread.currentThread() 则说明队列里面的第一个元素不是当前线程则返回 true.

* void lockInterruptibly() 与 lock() 方法类似，不同在于该方法对中断响应，就是当前线程在调用该方式时候，如果其它线程调用了当前线程线程的 interrupt（）方法，当前线程会抛出 InterruptedException 异常然后返回

**public** **void** **lockInterruptibly**() **throws** InterruptedException {

sync.acquireInterruptibly(1);

}

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

**throws** InterruptedException {

//当前线程被中断则直接抛出异常

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

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

//尝试获取资源

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

//调用AQS可被状态的方法

doAcquireInterruptibly(arg);

}

* boolean tryLock() 尝试获取锁，如果当前该锁没有被其它线程持有则当前线程获取该锁并返回 true, 否者返回 false，注意该方法不会引起当前线程阻塞。

**public** **boolean** **tryLock**() {

**return** sync.nonfairTryAcquire(1);

}

**final** **boolean** **nonfairTryAcquire**(**int** acquires) {

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

**int** c = getState();

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

**if** (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

**return** **true**;

}

}

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

**int** nextc = c + acquires;

**if** (nextc < 0) // overflow

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

setState(nextc);

**return** **true**;

}

**return** **false**;

}

如上代码与非公平锁的 tryAcquire() 方法类似，所以 tryLock() 使用的是非公平策略。

* boolean tryLock(long timeout, TimeUnit unit) 尝试获取锁与 tryLock（）不同在于设置了超时时间，如果超时没有获取该锁则返回 false。

**public** **boolean** **tryLock**(**long** timeout, TimeUnit unit)

**throws** InterruptedException {

//调用AQS的tryAcquireNanos方法。

**return** sync.tryAcquireNanos(1, unit.toNanos(timeout));

}

#### 3.3 释放锁

* void unlock() 尝试释放锁，如果当前线程持有该锁，调用该方法会让该线程对该线程持有的 AQS 状态值减一，如果减去 1 后当前状态值为 0 则当前线程会释放对该锁的持有，否者仅仅减一而已。如果当前线程没有持有该锁调用了该方法则会抛出 IllegalMonitorStateException 异常 ，代码如下：

**public** **void** **unlock**() {

sync.release(1);

}

**protected** **final** **boolean** **tryRelease**(**int** releases) {

//(11)如果不是锁持有者调用UNlock则抛出异常。

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

**if** (Thread.currentThread() != getExclusiveOwnerThread())

**throw** **new** IllegalMonitorStateException();

**boolean** free = **false**;

//(12)如果当前可重入次数为0，则清空锁持有线程

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

free = **true**;

setExclusiveOwnerThread(**null**);

}

//(13)设置可重入次数为原始值-1

setState(c);

**return** free;

}

如上代码（11）如果当前线程不是该锁持有者则直接抛出异常，否者看状态值剩余值是否为 0，为 0 则说明当前线程要释放对该锁的持有权，则执行（12）把当前锁持有者设置为 null。如果剩余值不为 0，则仅仅让当前线程对该锁的可重入次数减一。

#### 3.4 案例介绍

下面使用 ReentrantLock 来实现一个简单的线程安全的 list：

**public** **static** **class** **ReentrantLockList** {

//线程不安全的list

**private** ArrayList<String> array = **new** ArrayList<String>();

//独占锁

**private** **volatile** ReentrantLock **lock** = **new** ReentrantLock();

//添加元素

**public** **void** **add**(String e) {

**lock**.**lock**();

**try** {

array.**add**(e);

} **finally** {

**lock**.unlock();

}

}

//删元素

**public** **void** **remove**(String e) {

**lock**.**lock**();

**try** {

array.**remove**(e);

} **finally** {

**lock**.unlock();

}

}

//获取数据

**public** String **get**(**int** index) {

**lock**.**lock**();

**try** {

**return** array.**get**(index);

} **finally** {

**lock**.unlock();

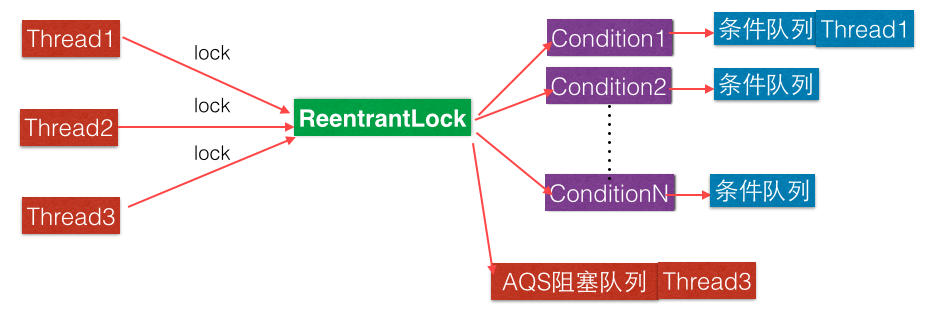
}

}

如上代码通过在操作 array 元素前进行加锁保证同时只有一个线程可以对 array 数组进行修改，但是同时也只能有一个线程对 array 元素进行访问。

同理最后使用几个图来加深理解：

如上图，假如线程 Thread1,Thread2,Thread3 同时尝试获取独占锁 ReentrantLock，假设 Thread1 获取到了，则 Thread2 和 Thread3 就会被转换为 Node 节点后放入 ReentrantLock 对应的 AQS 阻塞队列后阻塞挂起。



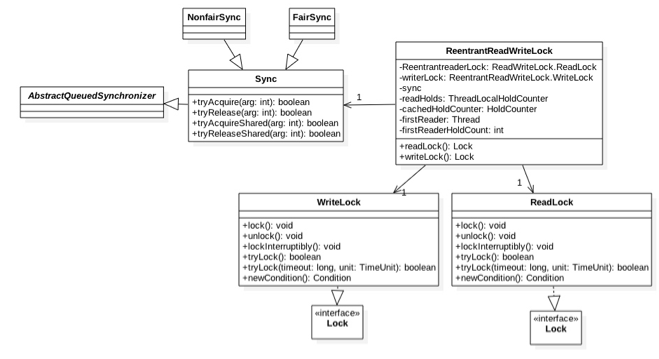
如上图，假设 Thread1 获取锁后调用了对应的锁创建的条件变量 1，那么 Thread1 就会释放获取到的锁，然后当前线程就会被转换为 Node 节点后插入到条件变量 1 的条件队列，由于 Thread1 释放了锁，所以阻塞到 AQS 队列里面 Thread2 和 Thread3 就有机会获取到该锁，假如使用的公平策略，那么这时候 Thread2 会获取到该锁，会从 AQS 队列里面移除 Thread2 对应的 Node 节点。

### 四、读写锁 ReentrantReadWriteLock 原理

在解决线程安全问题上使用 ReentrantLock 就可以，但是 ReentrantLock 是独占锁，同时只有一个线程可以获取该锁，而实际情况下会有写少读多的场景，显然 ReentrantLock 满足不了需求，所以 ReentrantReadWriteLock 应运而生，ReentrantReadWriteLock 采用读写分离，多个线程可以同时获取读锁。

#### 4.1 类图结构介绍

为了一览 ReentrantReadWriteLock 内部构造先看下它的类图结构：



如图读写锁内部维护了一个 ReadLock 和 WriteLock，并且也提供了公平和非公平的实现，下面只介绍下非公平的读写锁实现。我们知道 AQS 里面只维护了一个 state 状态，而 ReentrantReadWriteLock 则需要维护读状态和写状态，一个 state 是无法表示写和读状态的。ReentrantReadWriteLock 巧妙的使用 state 的高 16 位表示读状态也就是获取该读锁的线程个数，低 16 位表示获取到写锁的线程的可重入次数。

**static** **final** **int** SHARED\_SHIFT = 16;

//共享锁（读锁）状态单位值65536 **static** **final** **int** SHARED\_UNIT = (1 << SHARED\_SHIFT);//共享锁线程最大个数65535**static** **final** **int** MAX\_COUNT = (1 << SHARED\_SHIFT) - 1;

//排它锁(写锁)掩码 二进制 15个1**static** **final** **int** EXCLUSIVE\_MASK = (1 << SHARED\_SHIFT) - 1;

/\*\* 返回读锁线程数 \*/**static** **int** **sharedCount**(**int** c) { **return** c >>> SHARED\_SHIFT; }/\*\* 返回写锁可重入个数 \*/**static** **int** **exclusiveCount**(**int** c) { **return** c & EXCLUSIVE\_MASK; }

类图中 firstReader 用来记录第一个获取到读锁的线程，firstReaderHoldCount 则记录第一个获取到读锁的线程获取读锁的可重入数。cachedHoldCounter 用来记录最后一个获取读锁的线程获取读锁的可重入次数：

**static** **final** **class** **HoldCounter** {

**int** count = 0;

//线程id

**final** **long** tid = getThreadId(Thread.currentThread());

}

readHolds 是 ThreadLocal 变量，用来存放除去第一个获取读锁线程外的其它线程获取读锁的可重入数, 可知 ThreadLocalHoldCounter 继承了 ThreadLocal，里面 initialValue 方法返回一个 HoldCounter 对象。

**static** **final** **class** **ThreadLocalHoldCounter**

**extends** **ThreadLocal**<**HoldCounter**> {

**public** HoldCounter **initialValue**() {

**return** **new** HoldCounter();

}

}

##### ****4.1.1 写锁的获取与释放****

ReentrantReadWriteLock 中写锁是使用的 WriteLock 来实现的。

* void lock() 写锁是个独占锁，同时只有一个线程可以获取该锁。 如果当前没有线程获取到读锁和写锁则当前线程可以获取到写锁然后返回。 如果当前已经有线程取到读锁和写锁则当前线程则当前请求写锁线程会被阻塞挂起。 另外写锁是可重入锁，如果当前线程已经获取了该锁，再次获取的只是简单的把可重入次数加一后直接返回。

**public** **void** **lock**() {

sync.acquire(1);

}

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

// sync重写的tryAcquire方法

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

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

selfInterrupt();

}

如上代码 lock() 内部调用了 AQS 的 acquire 方法，其中的 tryAcquire 是 ReentrantReadWriteLock 内部 sync 类重写的，代码如下：

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

Thread current = Thread.currentThread();

**int** c = getState();

**int** w = exclusiveCount(c);

//（1） c!=0说明读锁或者写锁已经被某线程获取

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

（2）//w=0说明已经有线程获取了读锁或者w!=0并且当前线程不是写锁拥有者，则返回false

**if** (w == 0 || current != getExclusiveOwnerThread())

**return** **false**;

（3）//说明某线程获取了写锁，判断可重入个数

**if** (w + exclusiveCount(acquires) > MAX\_COUNT)

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

（4）// 设置可重入数量(1)

setState(c + acquires);

**return** **true**;

}

（5）//第一个写线程获取写锁

**if** (writerShouldBlock() ||

!compareAndSetState(c, c + acquires))

**return** **false**;

setExclusiveOwnerThread(current);

**return** **true**;

}

如上代码（1）如果当前 AQS 状态值不为 0 则说明当前已经有线程获取到了读锁或者写锁，代码（2）如果 w==0 说明状态值的低 16 位为 0，而状态值不为 0，则说明高 16 位不为 0，这暗示已经有线程获取了读锁，所以直接返回 false。

如果 w!=0 说明当前已经有线程获取了该写锁，则看当前线程是不是该锁的持有者，如果不是则返回 false。

执行到代码（3）说明当前线程之前已经获取到了该锁，则判断该线程的可重入此时是不是超过了最大值，是则抛出异常，否者执行点（4）增加当前线程的可重入次数然后返回 true.

如果 AQS 的状态值等于 0 则说明目前没有线程获取到读锁和写锁，则执行代码（5），其中对于 writerShouldBlock 方法非公平锁的实现为：

**final** **boolean** **writerShouldBlock**() {

**return** **false**; // writers can always barge

}

如代码对于非公平锁来说固定返回 false，则说明代码（5）抢占式执行 CAS 尝试获取写锁，获取成功则设置当前锁的持有者为当前线程返回 true, 否者返回 false。

对于公平锁的实现为：

**final** **boolean** **writerShouldBlock**() {

**return** hasQueuedPredecessors();

}

可知还是使用 hasQueuedPredecessors 来判断当前线程节点是否有前驱节点，如果有则当前线程放弃获取写锁的权限直接返回 false。

* void lockInterruptibly() 类似 lock() 方法，不同在于该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

**public** **void** **lockInterruptibly**() **throws** InterruptedException {

sync.acquireInterruptibly(1);

}

* boolean tryLock() 尝试获取写锁，如果当前没有其它线程持有写锁或者读锁，则当前线程获取写锁会成功，然后返回 true 如果当前已经其它线程持有写锁或者读锁则该方法直接返回 false，当前线程并不会被阻塞。 如果当前线程已经持有了该写锁则简单增加 AQS 的状态值后直接返回 true

**public** **boolean** **tryLock**( ) {

**return** sync.tryWriteLock();

}

**final** **boolean** **tryWriteLock**() {

Thread current = Thread.currentThread();

**int** c = getState();

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

**int** w = exclusiveCount(c);

**if** (w == 0 || current != getExclusiveOwnerThread())

**return** **false**;

**if** (w == MAX\_COUNT)

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

}

**if** (!compareAndSetState(c, c + 1))

**return** **false**;

setExclusiveOwnerThread(current);

**return** **true**;

}

如上代码与 tryAcquire 方法类似这里不再讲述，不同在于这里使用的非公平策略。

* boolean tryLock(long timeout, TimeUnit unit) 与 tryAcquire（）不同在于多了超时时间的参数，如果尝试获取写锁失败则会把当前线程挂起指定时间，待超时时间到后当前线程被激活，如果还是没有获取到写锁则返回 false。另外该方法对中断响应, 也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

**public** **boolean** **tryLock**(**long** timeout, TimeUnit unit)

**throws** InterruptedException {

**return** sync.tryAcquireNanos(1, unit.toNanos(timeout));

* void unlock() 尝试释放锁，如果当前线程持有该锁，调用该方法会让该线程对该线程持有的 AQS 状态值减一，如果减去 1 后当前状态值为 0 则当前线程会释放对该锁的持有，否者仅仅减一而已。如果当前线程没有持有该锁调用了该方法则会抛出 IllegalMonitorStateException 异常 ，代码如下：

**public** **void** **unlock**() {

sync.release(1);

}

**public** **final** **boolean** **release**(**int** arg) {

//调用ReentrantReadWriteLock中sync实现的tryRelease方法

**if** (tryRelease(arg)) {

//激活阻塞队列里面的一个线程

Node h = head;

**if** (h != **null** && h.waitStatus != 0)

unparkSuccessor(h);

**return** **true**;

}

**return** **false**;

}

**protected** **final** **boolean** **tryRelease**(**int** releases) {

//（6） 看是否是写锁拥有者调用的unlock

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

//（7）获取可重入值，这里没有考虑高16位，因为写锁时候读锁状态值肯定为0

**int** nextc = getState() - releases;

**boolean** free = exclusiveCount(nextc) == 0;

//（8）如果写锁可重入值为0则释放锁，否者只是简单更新状态值。

**if** (free)

setExclusiveOwnerThread(**null**);

setState(nextc);

**return** free;

}

如上代码 tryRelease 首先通过 isHeldExclusively 判断是否当前线程是该写锁的持有者，如果不是则抛出异常，否者执行代码（7）说明当前线程持有写锁，持有写锁说明状态值的高 16 位为 0，所以这里 nextc 值就是当前线程写锁的剩余可重入次数。

代码（8）判断当前可重入次数是否为 0，如果 free 为 true 说明可重入次数为 0，则当前线程会释放对写锁的持有，当前锁的持有者设置为 null。如果 free 为 false 则简单更新可重入次数。

##### ****4.1.2 读锁的获取与释放****

ReentrantReadWriteLock 中写锁是使用的 ReadLock 来实现的。

* void lock() 获取读锁，如果当前没有其它线程持有写锁，则当前线程可以获取读锁，AQS 的高 16 位的值会增加 1，然后方法返回。否者如果其它有一个线程持有写锁，则当前线程会被阻塞。

**public** **void** **lock**() {

sync.acquireShared(1);

}

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

//调用ReentrantReadWriteLock中的sync的tryAcquireShared方法

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

//调用AQS的doAcquireShared方法

doAcquireShared(arg);

}

如上代码读锁的 lock 方法调用了 AQS 的 acquireShared 方法，内部调用了 ReentrantReadWriteLock 中的 sync 重写的 tryAcquireShared 方法代码如下：

**protected** final **int** **tryAcquireShared**(**int** unused) {

//(1)获取当前状态值

Thread current = Thread.currentThread();

**int** c = getState();

//(2)判断是否写锁被占用

**if** (exclusiveCount(c) != 0 &&

getExclusiveOwnerThread() != current)

**return** -1;

//（3）获取读锁计数

**int** r = sharedCount(c);

//（4）尝试获取锁，多个读线程只有一个会成功，不成功的进入下面fullTryAcquireShared进行重试

**if** (!readerShouldBlock() &&

r < MAX\_COUNT &&

compareAndSetState(c, c + SHARED\_UNIT)) {

//(5)第一个线程获取读锁

**if** (r == 0) {

firstReader = current;

firstReaderHoldCount = 1;

//(6)如果当前线程是第一个获取读锁的线程

} **else** **if** (firstReader == current) {

firstReaderHoldCount++;

} **else** {

//（7）记录最后一个获取读锁的线程或记录其它线程读锁的可重入数

HoldCounter rh = cachedHoldCounter;

**if** (rh == null || rh.tid != current.getId())

cachedHoldCounter = rh = readHolds.**get**();

**else** **if** (rh.count == 0)

readHolds.**set**(rh);

rh.count++;

}

**return** 1;

}

//(8)类似tryAcquireShared，但是是自旋获取

**return** fullTryAcquireShared(current);

}

如上代码首先获取了当前 AQS 的状态值，然后代码（2）看是否有其它线程获取到了写锁，如果是则直接返回了 - 1，然后调用 AQS 的 doAcquireShared 方法把当前线程放入阻塞队列。

否者执行到代码（3）得到获取到读锁的线程个数，到这里说明目前没有线程获取到写锁，但是还是可能有线程持有读锁，然后执行代码（4），非公平锁的 readerShouldBlock 实现代码如下：

**final** **boolean** **readerShouldBlock**() {

**return** apparentlyFirstQueuedIsExclusive();

}

**final** **boolean** **apparentlyFirstQueuedIsExclusive**() {

Node h, s;

**return** (h = head) != **null** &&

(s = h.next) != **null** &&

!s.isShared() &&

s.thread != **null**;

}

如上代码作用是如果队列里面存在一个元素，则判断第一个元素是不是正在尝试获取写锁，如果不是的话，则当前线程使用判断当前获取读锁线程是否达到了最大值，最后执行 CAS 操作设置 AQS 状态值的高 16 位值增加 1。

代码（5）（6）记录第一个获取读锁的线程并统计该线程获取读锁的可重入数，代码（7）使用 cachedHoldCounter 记录最后一个获取到读锁的线程并同时该线程获取读锁的可重入数，另外 readHolds 记录了当前线程获取读锁的可重入数。

如果 readerShouldBlock 返回 true 则说明有线程正在获取写锁，则执行代码（8）fullTryAcquireShared 代码与 tryAcquireShared 类似，不同在于前者是通过循环自旋获取。

final **int** **fullTryAcquireShared**(Thread current) {

HoldCounter rh = null;

**for** (;;) {

**int** c = getState();

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

**if** (getExclusiveOwnerThread() != current)

**return** -1;

// else we hold the exclusive lock; blocking here

// would cause deadlock.

} **else** **if** (readerShouldBlock()) {

// Make sure we're not acquiring read lock reentrantly

**if** (firstReader == current) {

// assert firstReaderHoldCount > 0;

} **else** {

**if** (rh == null) {

rh = cachedHoldCounter;

**if** (rh == null || rh.tid != getThreadId(current)) {

rh = readHolds.**get**();

**if** (rh.count == 0)

readHolds.**remove**();

}

}

**if** (rh.count == 0)

**return** -1;

}

}

**if** (sharedCount(c) == MAX\_COUNT)

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

**if** (compareAndSetState(c, c + SHARED\_UNIT)) {

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

firstReader = current;

firstReaderHoldCount = 1;

} **else** **if** (firstReader == current) {

firstReaderHoldCount++;

} **else** {

**if** (rh == null)

rh = cachedHoldCounter;

**if** (rh == null || rh.tid != getThreadId(current))

rh = readHolds.**get**();

**else** **if** (rh.count == 0)

readHolds.**set**(rh);

rh.count++;

cachedHoldCounter = rh; // cache for release

}

**return** 1;

}

}

}

void lockInterruptibly() 类似 lock() 方法，不同在于该方法对中断响应，也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

boolean tryLock() 尝试获取读锁，如果当前没有其它线程持有写锁，则当前线程获取写锁会成功，然后返回 true

如果当前已经其它线程持有写锁则该方法直接返回 false，当前线程并不会被阻塞。

如果其它获取当前线程已经持有了该读锁则简单增加 AQS 的状态值高 16 位后直接返回 true。代码类似 tryLock 这里不再讲述。

boolean tryLock(long timeout, TimeUnit unit) 与 tryLock（）不同在于多了超时时间的参数，如果尝试获取读锁失败则会把当前线程挂起指定时间，待超时时间到后当前线程被激活，如果还是没有获取到读锁则返回 false。另外该方法对中断响应, 也就是当其它线程调用了该线程的 interrupt() 方法中断了当前线程，当前线程会抛出异常 InterruptedException

void unlock()

**public** **void** **unlock**() {

sync.releaseShared(1);

}

**public** **final** **boolean** **releaseShared**(**int** arg) {

**if** (tryReleaseShared(arg)) {

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

**protected** final boolean **tryReleaseShared**(**int** unused) {

Thread current = Thread.currentThread();

//如果当前线程是第一个获取读锁线程

**if** (firstReader == current) {

//如果可重入次数为1

**if** (firstReaderHoldCount == 1)

firstReader = null;

**else**//否者可重入次数减去1

firstReaderHoldCount--;

} **else** {

//如果当前线程不是最后一个获取读锁线程，则从threadlocal里面获取

HoldCounter rh = cachedHoldCounter;

**if** (rh == null || rh.tid != current.getId())

rh = readHolds.**get**();

//如果可重入次数<=1则清除threadlocal

**int** count = rh.count;

**if** (count <= 1) {

readHolds.**remove**();

**if** (count <= 0)

**throw** unmatchedUnlockException();

}

//可重入次数减去一

--rh.count;

}

//循环直到自己的读计数-1 cas更新成功

**for** (;;) {

**int** c = getState();

**int** nextc = c - SHARED\_UNIT;

**if** (compareAndSetState(c, nextc))

**return** nextc == 0;

}

}

##### ****4.1.3 案例介绍****

上节介绍了使用 ReentrantLock 实现的线程安全的 list, 但是由于 ReentrantLock 是独占锁所以在读多写少的情况下性能很差，下面使用 ReentrantReadWriteLock 来改造为如下代码：

**public** **static** **class** **ReentrantLockList** {

//线程不安全的list

**private** ArrayList<String> array = **new** ArrayList<String>();

//独占锁

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

**private** final Lock readLock = **lock**.readLock();

**private** final Lock writeLock = **lock**.writeLock();

//添加元素

**public** **void** **add**(String e) {

writeLock.**lock**();

**try** {

array.**add**(e);

} **finally** {

writeLock.unlock();

}

}

//删元素

**public** **void** **remove**(String e) {

writeLock.**lock**();

**try** {

array.**remove**(e);

} **finally** {

writeLock.unlock();

}

}

//获取数据

**public** String **get**(**int** index) {

readLock.**lock**();

**try** {

**return** array.**get**(index);

} **finally** {

readLock.unlock();

}

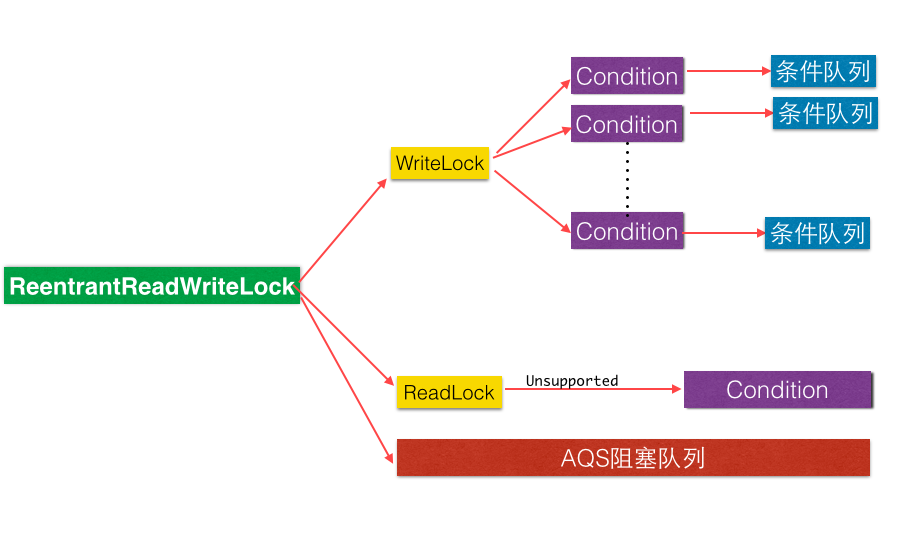
}

}

如代码调用 get 方法适合使用的是读锁，这样运行多个读线程同时访问 list 的元素，在读多写少的情况下性能相比 ReentrantLock 会很好。

注：本节介绍了读写锁 ReentrantReadWriteLock 的原理，可知底层还是使用 AQS 实现的，ReentrantReadWriteLock 巧妙的使用 AQS 的状态值的高 16 位表示获取读锁的线程个数，低 16 位表示获取写锁的线程的可重入次数，并通过 CAS 对其进行操作实现了读写分离，在读多写少的场景下比较适用。

最后一张图加深对 ReentrantReadWriteLock 的理解：



#### 4.2 JDK8 新增的 StampedLock 锁探究

StampedLock 是并发包里面 jdk8 版本新增的一个锁，该锁提供了三种模式的读写控制，当调用获取锁的系列函数时候，会返回一个 long 型的变量，我们称之为戳记（stamp），这个戳记代码了锁的状态。其中 try 系列获取锁的函数，当获取锁失败后会返回为 0 的 stamp 值。当调用释放锁和转换锁的方法时候需要传入获取锁时候返回的 stamp 值。

StampedLock 提供的三种读写模式的锁分别如下：

写锁 writeLock，是个排它锁或者叫独占锁，同时只有一个线程可以获取该锁，当一个线程获取该锁后，其它请求读锁和写锁的线程必须等待，类似于 ReentrantReadWriteLock 的写锁（不同在于这里的写锁是不可重入锁）；当目前没有线程持有读锁或者写锁的时候才可以获取到该锁，请求该锁成功后会返回一个 stamp 票据变量用来表示该锁的版本，当释放该锁时候需要调用 unlockWrite 方法并传递获取锁时候的 stamp 参数。并且提供了非阻塞的 tryWriteLock 方法。

悲观读锁 readLock，是个共享锁，在没有线程获取独占写锁的情况下，同时多个线程可以获取该锁; 如果已经有线程持有写锁，其它线程请求获取该读锁会被阻塞，这类似 ReentrantReadWriteLock 的读锁（不同在于这里的读锁是不可重入锁）。这里说的悲观是指在具体操作数据前悲观的认为其它线程可能要对自己操作的数据进行修改，所以需要先对数据加锁，这是在读少写多的情况下的一种考虑, 请求该锁成功后会返回一个 stamp 票据变量用来表示该锁的版本，当释放该锁时候需要 unlockRead 并传递参数 stamp。并且提供了非阻塞的 tryReadLock。

乐观读锁 tryOptimisticRead，是相对于悲观锁来说的，在操作数据前并没有通过 CAS 设置锁的状态，仅仅是通过位运算测试；如果当前没有线程持有写锁，则简单的返回一个非 0 的 stamp 版本信息，获取该 stamp 后在具体操作数据前还需要调用 validate 验证下该 stamp 是否已经不可用，也就是看当调用 tryOptimisticRead 返回 stamp 后，到当前时间间是否有其它线程持有了写锁，如果是那么 validate 会返回 0，否者就可以使用该 stamp 版本的锁对数据进行操作。由于 tryOptimisticRead 并没有使用 CAS 设置锁状态，所以不需要显示的释放该锁。该锁的一个特点是适用于读多写少的场景，因为获取读锁只是使用位操作进行检验，不涉及 CAS 操作，所以效率会高很多，但是同时由于没有使用真正的锁，在保证数据一致性上需要拷贝一份要操作的变量到方法栈，并且在操作数据时候可能其它写线程已经修改了数据，而我们操作的是方法栈里面的数据，也就是一个快照，所以最多返回的不是最新的数据，但是一致性还是得到保障的。

StampedLock 还支持这三种锁在一定条件下进行相互转换： 例如long tryConvertToWriteLock(long stamp)期望把 stamp 标示的锁升级为写锁，这个函数会在下面几种情况下返回一个有效的 stamp（也就是晋升写锁成功）：

1）当前锁已经是写锁模式了。

2）当前锁处于读锁模式，并且没有其他线程是读锁模式

3）当前处于乐观读模式，并且当前写锁可用。

另外 StampedLock 的读写锁都是不可重入锁，所以当获取锁后释放锁前不应该在调用会获取锁的操作，以避免产生死锁。当多个线程同时尝试获取读锁和写锁时候，谁写获取锁没有一定的规则，完全都是尽力而为，是随机的。并且该锁不是直接实现 Lock 或 ReadWriteLock 接口，而是内部自己维护了一个双向阻塞队列。

下面通过 JDK8 里面提供的一个管理二维点的例子讲解来加深对上面讲解的理解。

**class** **Point** {

// 成员变量

**private** **double** x, y;

// 锁实例

**private** **final** StampedLock sl = **new** StampedLock();

// 排它锁-写锁（writeLock）

**void** **move**(**double** deltaX, **double** deltaY) {

**long** stamp = sl.writeLock();

**try** {

x += deltaX;

y += deltaY;

} **finally** {

sl.unlockWrite(stamp);

}

}

// 乐观读锁（tryOptimisticRead）

**double** **distanceFromOrigin**() {

// 尝试获取乐观读锁（1）

**long** stamp = sl.tryOptimisticRead();

// 将全部变量拷贝到方法体栈内（2）

**double** currentX = x, currentY = y;

// 检查在（1）获取到读锁票据后，锁有没被其它写线程排它性抢占（3）

**if** (!sl.validate(stamp)) {

// 如果被抢占则获取一个共享读锁（悲观获取）（4）

stamp = sl.readLock();

**try** {

// 将全部变量拷贝到方法体栈内（5）

currentX = x;

currentY = y;

} **finally** {

// 释放共享读锁（6）

sl.unlockRead(stamp);

}

}

// 返回计算结果（7）

**return** Math.sqrt(currentX \* currentX + currentY \* currentY);

}

// 使用悲观锁获取读锁，并尝试转换为写锁

**void** **moveIfAtOrigin**(**double** newX, **double** newY) {

// 这里可以使用乐观读锁替换（1）

**long** stamp = sl.readLock();

**try** {

// 如果当前点在原点则移动（2）

**while** (x == 0.0 && y == 0.0) {

// 尝试将获取的读锁升级为写锁（3）

**long** ws = sl.tryConvertToWriteLock(stamp);

// 升级成功，则更新票据，并设置坐标值，然后退出循环（4）

**if** (ws != 0L) {

stamp = ws;

x = newX;

y = newY;

**break**;

} **else** {

// 读锁升级写锁失败则释放读锁，显示获取独占写锁，然后循环重试（5）

sl.unlockRead(stamp);

stamp = sl.writeLock();

}

}

} **finally** {

// 释放锁（6）

sl.unlock(stamp);

}

}

}

如上代码 Point 类里面有两个成员变量（x,y) 来标示一个点的二维坐标，和三个操作坐标变量的方法，另外实例化了一个 StampedLock 对象用来保证操作的原子性。

首先分析下 move 方法，该函数作用是使用参数的增量值，改变当前 point 坐标的位置；代码先获取到了写锁，然后对 point 坐标进行修改，然后释放锁。该锁是排它锁，这保证了其它线程调用 move 函数时候会被阻塞，也保证了其它线程不能获取读锁，读取坐标的值，直到当前线程显示释放了写锁，也就是保证了对变量 x,y 操作的原子性和数据一致性。

然后看下 distanceFromOrigin 方法，该方法作用是计算当前位置到原点（坐标为 0,0）的距离，代码（1）首先尝试获取乐观读锁，如果当前没有其它线程获取到了写锁，那么（1）会返回一个非 0 的 stamp 用来表示版本信息，代码（2）拷贝坐标变量到本地方法栈里面。

代码（3）检查在（1）获取到的 stamp 值是否还有效，之所以还要在此校验是因为代码（1）获取读锁时候并没有通过 CAS 操作修改锁的状态，而是简单的通过与或操作返回了一个版本信息，这里校验是看在在获取版本信息到现在的时间段里面是否有其它线程持有了写锁，如果有则之前获取的版本信息就无效了。

这里如果校验成功则执行（7）使用本地方法栈里面的值进行计算然后返回。需要注意的是在代码（3) 校验成功后，代码（7）计算期间，其它线程可能获取到了写锁并且修改了 x,y 的值，而当前线程执行代码（7）进行计算时候采用的还是修改前值的拷贝，也就是操作的值是对之前值的一个拷贝，一个快照，并不是最新的值。

另外还有个问题，代码（2) 和（3）能否互换，答案是不能。假设位置换了，那么首先执行 validate，假如验证通过了，要拷贝 x,y 值到本地方法栈，而在拷贝的过程中很有可能其它线程已经修改了 x,y 中的一个，这就造成了数据的不一致性了。那么你可能会问，即使不交换代码 (2) 和（3），在拷贝 x,y 值到本地方法栈里面时，也会存在其它线程修改了 x,y 中的一个值那，这不也会存在问题？这个确实会存在，但是，别忘了拷贝后还有一道 validate, 如果这时候有线程修改了 x,y 中的值，那么肯定是有线程在调用 validate 前，调用 sl.tryOptimisticRead 后获取了写锁，那么进行 validate 时候就会失败。

现在应该明白了吧，这也是乐观读设计的精妙之处也是使用时候容易出问题的地方。下面继续分析 validate 失败后会执行代码（4）获取悲观读锁，如果这时候其他线程持有写锁则代码（4）会导致的当前线程阻塞直到其它线程释放了写锁。如果这时候没有其他线程获取到写锁，那么当前线程就可以获取到读锁，然后执行代码（5）重新拷贝新的坐标值到本地方法栈，然后就是代码（6）释放了锁，拷贝的时候由于加了读锁，所以拷贝期间其它线程获取写锁时候会被阻塞，这保证了数据的一致性，另外这里 x,y 没有被声明为 volatie，会不会存在内存不可见性问题那？答案是不会，因为加锁的语义保证了内存可见性，关系内存可见性问题可以参考 chat(<http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f>)。

最后代码（7）使用方法栈里面数据计算返回，同理这里在计算时候使用的数据也可能不是最新的，其它写线程可能已经修改过原来的 x,y 值了。

最后一个方法 moveIfAtOrigin 作用是如果当前坐标为原点则移动到指定的位置。代码（1）获取悲观读锁，保证其它线程不能获取写锁修改 x,y 值，然后代码（2）判断如果当前点在原点则更新坐标，代码（3) 尝试升级读锁为写锁，这里升级不一定成功，因为多个线程都可以同时获取悲观读锁，当多个线程都执行到（3）时候只有一个可以升级成功，升级成功则返回非 0 的 stamp，否非返回 0。这里假设当前线程升级成功，然后执行步骤（4）更新 stamp 值和坐标值，然后退出循环。如果升级失败则执行步骤（5）首先释放读锁然后申请写锁，获取到写锁后在循环重新设置坐标值。最后步骤（6) 释放锁。

使用乐观读锁还是很容易犯错误的，必须要小心，必须要保证如下的使用顺序：

**long** stamp = **lock**.tryOptimisticRead(); //非阻塞获取版本信息

copyVaraibale2ThreadMemory();//拷贝变量到线程本地堆栈**if**(!**lock**.validate(stamp)){ // 校验

**long** stamp = **lock**.readLock();//获取读锁

**try** {

copyVaraibale2ThreadMemory();//拷贝变量到线程本地堆栈

} **finally** {

**lock**.unlock(stamp);//释放悲观锁

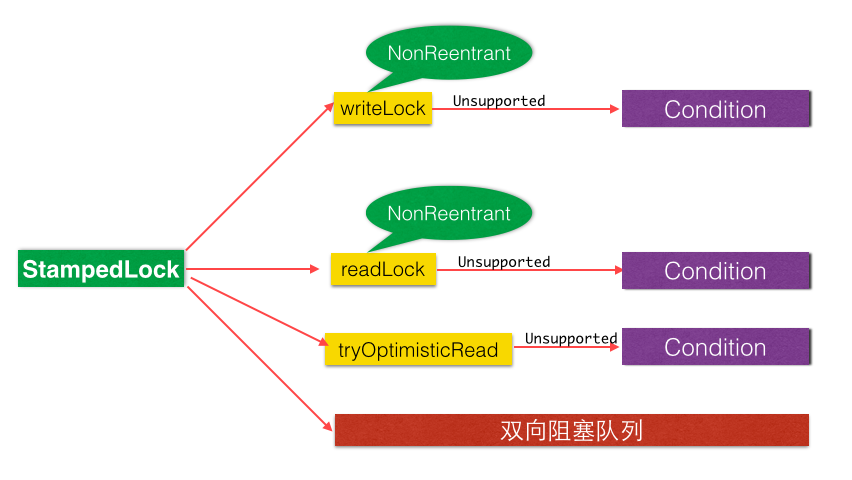
}

}

useThreadMemoryVarables();//使用线程本地堆栈里面的数据进行操作

总结： StampedLock 提供的读写锁与 ReentrantReadWriteLock 类似，只是前者的都是不可重入锁。但是前者通过提供乐观读锁在多线程多读的情况下提供更好的性能，这是因为获取乐观读锁时候不需要进行 CAS 操作设置锁的状态，而只是简单的测试状态。

最后通过一个图来一览 StampedLock 的组成：



### 六、总结

本文首先对 AQS 进行概述，讲述了 AQS 做为同步器的基础设施，为子类抽象了哪些能力。然后讲解了独占锁 ReentrantLock 原理探究，讲解了如何基于 AQS 的能力实现了公平与非公平的独占锁，然后讲解了读写锁 ReentrantReadWriteLock 原理，ReentrantReadWriteLock 使用读写分离锁，在读多写少的情景下的比较适用。最后讲解了 JDK8 新增的 StampedLock 锁原理探究，该锁提供了三种模式的读写控制。锁是 JUC 包中最难理解的一块，希望读者能够在本文指导下结合 JUC 包源码进行深入思考，以便加深理解。

# Java 并发编程之美：并发编程高级篇之四

### 一、前言

Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之四，图形结合讲解 JDK 中线程安全的并发队列实现原理，内容如下：（建议先阅读 [并发编程高级篇之三 - 锁](http://gitbook.cn/gitchat/activity/5ac85e1b2a04fd6c837137a2) ）

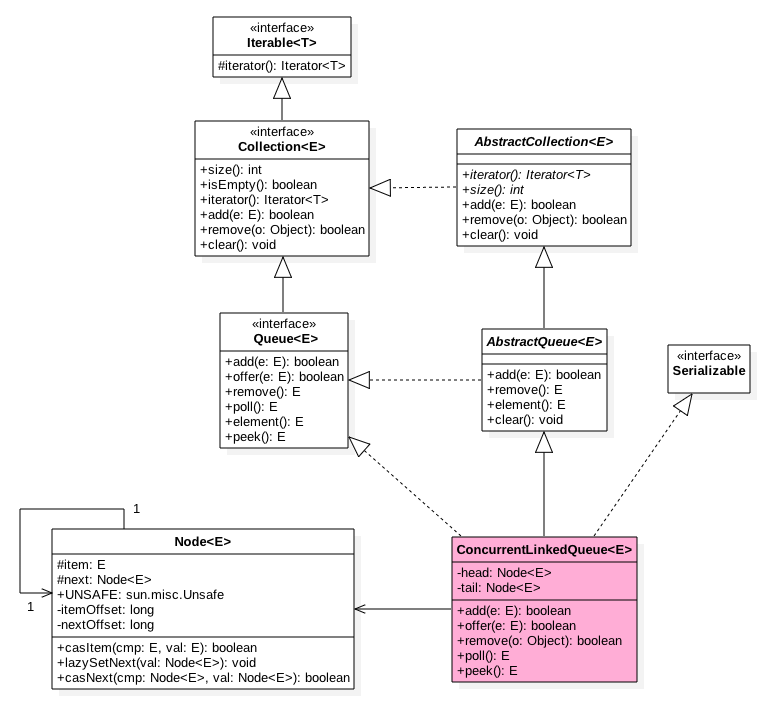
* JDK 中基于链表的非阻塞无界队列 ConcurrentLinkedQueue 原理剖析，ConcurrentLinkedQueue 内部是如何使用 CAS 非阻塞算法来保证多线程下入队出队操作的线程安全？
* JDK 中基于链表的阻塞队列 LinkedBlockingQueue 原理剖析，LinkedBlockingQueue 内部是如何使用两个独占锁 ReentrantLock 以及对应的条件变量保证多线程先入队出队操作的线程安全？为什么不使用一把锁，使用两把为何能提高并发度？
* JDK 中基于数组的阻塞队列 ArrayBlockingQueue 原理剖析，ArrayBlockingQueue 内部如何基于一把独占锁以及对应的两个条件变量实现出入队操作的线程安全？
* JDK 中无界优先级队列 PriorityBlockingQueue 原理剖析，PriorityBlockingQueue 内部使用堆算法保证每次出队都是优先级最高的元素，元素入队时候是如何建堆的，元素出队后如何调整堆的平衡的？
* 浅谈上面各种队列的比较，以及给出部分队列在开源框架中使用样例。

### 二、ConcurrentLinkedQueue 原理探究

ConcurrentLinkedQueue 是线程安全的无界非阻塞队列，底层数据结构使用单向链表实现，入队和出队操作使用 CAS 来实现线程安全。

#### 2.1 ConcurrentLinkedQueue 类图结构

先简单介绍下 ConcurrentLinkedQueue 的类图结构如下图：



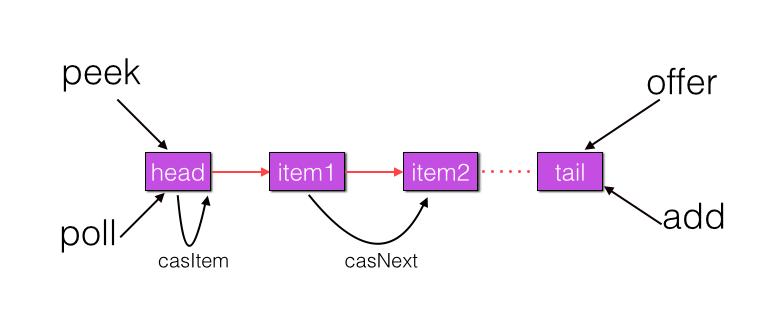
* 如上类图 ConcurrentLinkedQueue 内部的队列是使用单向链表方式实现，其中两个 volatile 类型的 Node 节点分别用来存放队列的首尾节点。从下面无参构造函数可知默认头尾节点都是指向 item 为 null 的哨兵节点。

**public** **ConcurrentLinkedQueue**() {

head = tail = **new** Node<E>(null);

}

* Node 节点内部则维护一个 volatile 修饰的变量 item 用来存放节点的值，next 用来存放链表的下一个节点，从而链接为一个单向无界链表。

首先一个图来概况该队列构成，读者可以读完本节后在回头体会这个图： 

#### 2.2 ConcurrentLinkedQueue 原理介绍

本节主要介绍 ConcurrentLinkedQueue 的几个主要的方法的实现原理

##### 2.2.1 offer 操作

offer 操作是在队列末尾添加一个元素，如果传递的参数是 null 则抛出 NPE 异常，否者由于 ConcurrentLinkedQueue 是无界队列该方法一直会返回 true。另外由于使用 CAS 无阻塞算法，该方法不会阻塞调用线程，下面具体看看实现原理。

**public** **boolean** **offer**(E e) {

//（1）e为null则抛出空指针异常

checkNotNull(e);

//（2）构造Node节点

**final** Node<E> newNode = **new** Node<E>(e);

//（3）从尾节点进行插入

**for** (Node<E> t = tail, p = t;;) {

Node<E> q = p.next;

//（4）如果q==null说明p是尾节点，则执行插入

**if** (q == **null**) {

//（5）使用CAS设置p节点的next节点

**if** (p.casNext(**null**, newNode)) {

//（6）cas成功，则说明新增节点已经被放入链表，然后设置当前尾节点

**if** (p != t)

casTail(t, newNode); // Failure is OK.

**return** **true**;

}

}

**else** **if** (p == q)//(7）

//多线程操作时候，由于poll操作移除元素后有可能会把head变为自引用，然后head的next变为新head，所以这里需要

//重新找新的head，因为新的head后面的节点才是正常的节点。

p = (t != (t = tail)) ? t : head;

**else**

//（8） 寻找尾节点

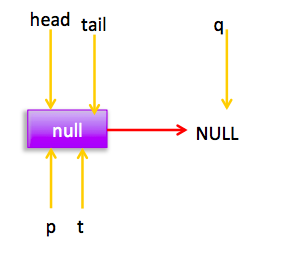
p = (p != t && t != (t = tail)) ? t : q;

}

}

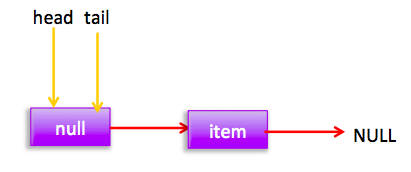
上节类图结构时候谈到构造队列时候参构造函数创建了一个 item 为 null 的哨兵节点，并且 head 和 tail 都是指向这个节点，下面通过图形结合来讲解下 offer 操作的代码实现。

* 首先看下当一个线程调用 offer（item）时候情况：首先代码（1）对传参判断空检查，如果为 null 则抛出 NPE 异常，然后代码（2）则使用 item 作为构造函数参数创建了一个新的节点，代码（3）从队列尾部节点开始循环，意图是从队列尾部添加元素。

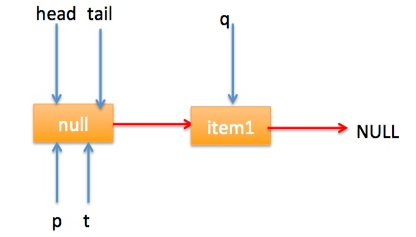


上图是执行代码（4）时候队列的情况，这时候节点 p,t,head,tail 同时指向了 item 为 null 的哨兵节点，由于哨兵节点的 next 节点为 null, 所以这里 q 指向也是 null。

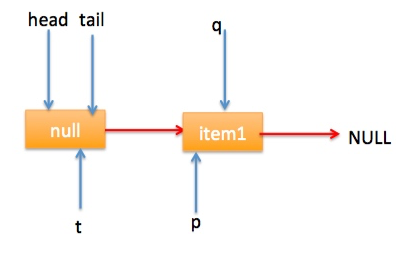
代码（4）发现q==null则执行代码（5）通过 CAS 原子操作判断 p 节点的 next 节点是否为 null，如果为 null 则使用节点 newNode 替换 p 的 next 节点，然后执行代码（6）由于p==t所以没有设置尾部节点，然后退出 offer 方法，这时候队列的状态图如下：



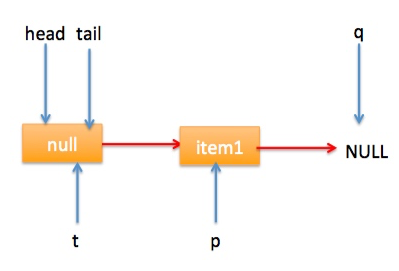
* 上面讲解的是一个线程调用 offer 方法的情况，如果多个线程同时调用，就会存在多个线程同时执行到代码（5），假设线程 A 调用 offer（item1), 线程 B 调用 offer(item2), 线程 A 和 B 同时执行到 p.casNext(null, newNode)。而 CAS 的比较并设置操作是原子性的，假设线程 A 先执行了比较设置操作则发现当前 p 的 next 节点确实是 null 则会原子性更新 next 节点为 newNode，这时候线程 B 也会判断 p 的 next 节点是否为 null，结果发现不是 null（因为线程 A 已经设置了 p 的 next 为 newNode）则会跳到步骤（3），然后执行到步骤（4）时候队列分布图为：



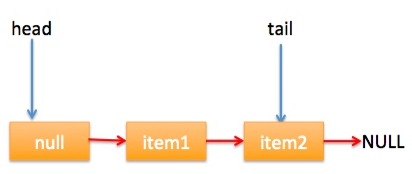
根据这个状态图可知线程 B 会去执行代码（8），然后 q 赋值给了 p，这时候队列状态图为：



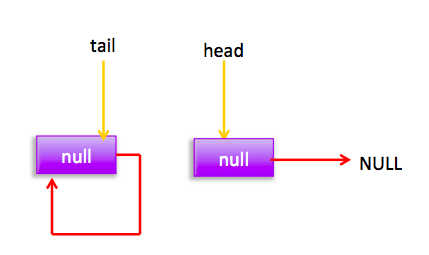
然后线程 B 再次跳转到代码（3）执行，当执行到代码（4）时候队列状态图为：



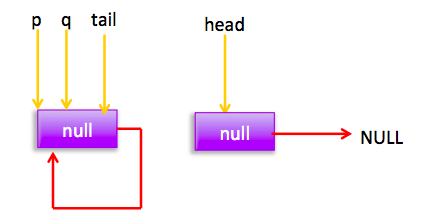
由于这时候 q==null, 所以线程 B 会执行步骤（5），通过 CAS 操作判断当前 p 的 next 节点是否是 null，不是则再次循环后尝试，是则使用 newNode 替换，假设 CAS 成功了，那么执行步骤（6）由于 p!=t 所以设置 tail 节点为 newNode，然后退出 offer 方法。这时候队列分布图为：



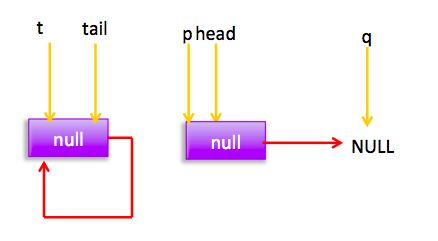
* 分析到现在，offer 代码的执行路径现在就差步骤（7）还没走过，其实这个要在执行 poll 操作后才会出现，这里先看下执行 poll 操作后可能会存在的的一种情况如下图：



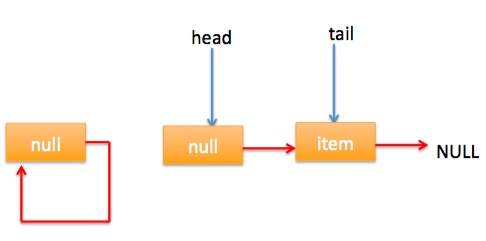
下面分析下当队列处于这种状态时候调用 offer 添加元素代码执行到步骤（4）时候的状态图：



由于 q 节点不为空并且p==q所以执行步骤（7），由于t==tail所以 p 被赋值为了 head，然后进入循环，循环后执行到代码（4）时候队列状态为：



由于q==null, 所以执行步骤（5）进行 CAS 操作，如果当前没有其他线程执行 offer 操作，则 CAS 操作会成功，p 的 next 节点被设置为新增节点，然后执行步骤（6），由于p!=t所以设置新节点为队列为节点，现在队列状态如下：



这里自引用的节点会被垃圾回收掉。

总结：可见 offer 操作里面关键步骤是代码（5）通过原子 CAS 操作来进行控制同时只有一个线程可以追加元素到队列末尾，进行 cas 竞争失败的线程则会通过循环一次次尝试进行 cas 操作，直到 cas 成功才会返回，也就是通过使用无限循环里面不断进行 CAS 尝试方式来替代阻塞算法挂起调用线程，相比阻塞算法这是使用 CPU 资源换取阻塞所带来的开销。

##### 2.2.2 poll 操作

poll 操作是在队列头部获取并且移除一个元素，如果队列为空则返回 null，下面看看实现原理。

**public** E **poll**() {

//(1) goto标记

restartFromHead:

//（2）无限循环

**for** (;;) {

**for** (Node<E> h = head, p = h, q;;) {

//（3）保存当前节点值

E item = p.item;

//（4）当前节点有值则cas变为null

**if** (item != **null** && p.casItem(item, **null**)) {

//（5）cas成功标志当前节点以及从链表中移除

**if** (p != h)

updateHead(h, ((q = p.next) != **null**) ? q : p);

**return** item;

}

//（6）当前队列为空则返回null

**else** **if** ((q = p.next) == **null**) {

updateHead(h, p);

**return** **null**;

}

//（7）自引用了，则重新找新的队列头节点

**else** **if** (p == q)

**continue** restartFromHead;

**else**//(8）

p = q;

}

}

}

**final** **void** **updateHead**(Node<E> h, Node<E> p) {

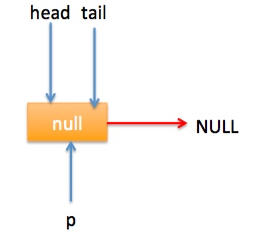
**if** (h != p && casHead(h, p))

h.lazySetNext(h);

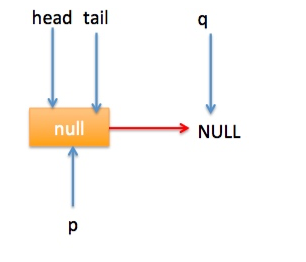
}

同理本节也通过图形结合的方式来讲解代码执行逻辑：

* poll 操作是从队头获取元素，所以代码（2）内层循环是从 head 节点开始迭代，代码（3）获取当前队头的节点，当队列一开始为空时候队列状态为：

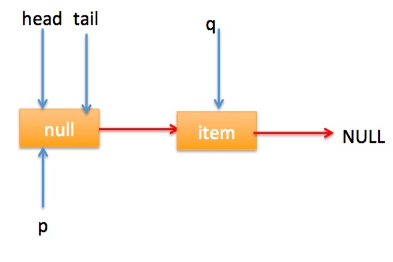


由于 head 节点指向的为 item 为 null 的哨兵节点，所以会执行到代码（6），假设这个过程中没有线程调用 offer 方法，则此时 q 等于 null 如下图：

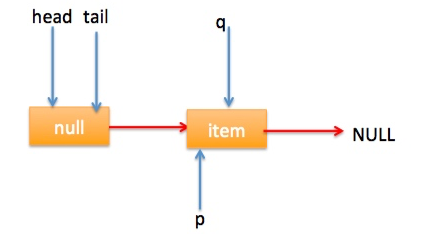


所以执行 updateHead 方法，由于 h 等于 p 所以没有设置头结点，poll 方法直接返回 null。

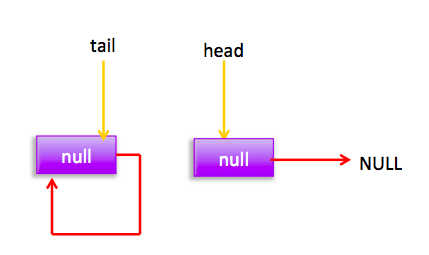
* 假设执行到代码（6）时候已经有其它线程调用了 offer 方法成功添加一个元素到队列，这时候 q 指向的是新增元素的节点，这时候队列状态为：



所以代码（6）判断结果为 false，然后会转向代码（7）执行，而此时 p 不等于 q，所以转向代码（8）执行，执行结果是 p 指向了节点 q，此时队列状态为：

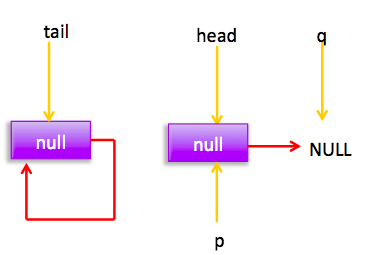


然后程序转向代码（3）执行，p 现在指向的元素值不为 null，则执行p.casItem(item, null) 通过 CAS 操作尝试设置 p 的 item 值为 null，如果此时没有其它线程进行 poll 操作，CAS 成功则执行代码（5）由于此时 p!=h 所以设置头结点为 p，poll 然后返回被从队列移除的节点值 item。此时队列状态为:



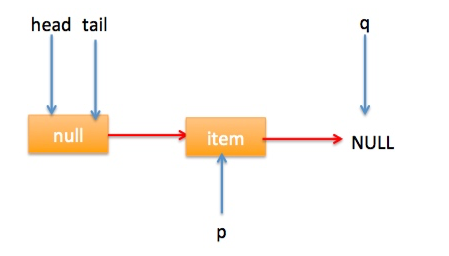
这个状态就是讲解 offer 操作时候，offer 代码的执行路径（7）执行的前提状态。

* 假如现在一个线程调用了 poll 操作，则在执行代码（4) 时候队列状态为：



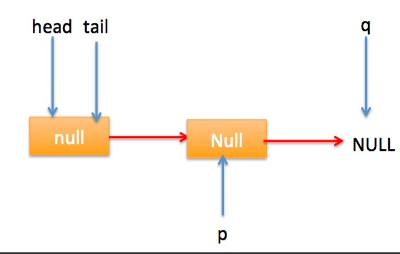
可知这时候执行代码（6）返回 null.

* 现在 poll 的代码还有个分支（7）没有执行过，那么什么时候会执行那？下面来看看，假设线程 A 执行 poll 操作时候当前队列状态为：

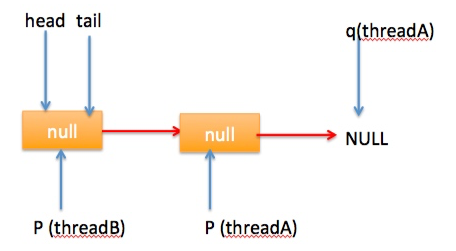


那么执行p.casItem(item, null) 通过 CAS 操作尝试设置 p 的 item 值为 null。

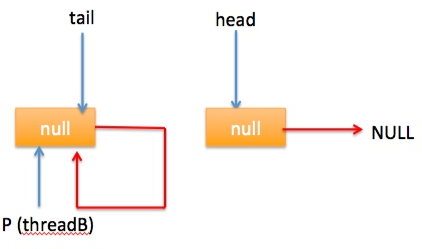
假设 CAS 设置成功则标示该节点从队列中移除了，此时队列状态为：



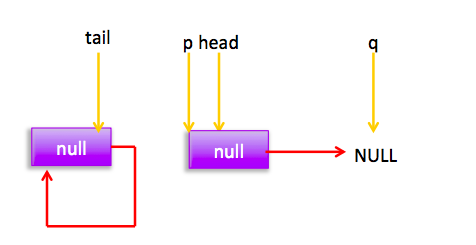
然后由于 p!=h, 所以会执行 updateHead 方法，假如线程 A 执行 updateHead 前另外一个线程 B 开始 poll 操作这时候线程 B 的 p 指向 head 节点，但是还没有执行到代码（6）这时候队列状态为：



然后线程 A 执行 updateHead 操作，执行完毕后线程 A 退出，这时候队列状态为：



然后线程 B 继续执行代码（6）q=p.next由于该节点是自引用节点所以p==q所以会执行代码（7）跳到外层循环 restartFromHead，重新获取当前队列队头 head, 现在状态为：



总结：poll 方法移除一个元素时候只是简单的使用 CAS 操作把当前节点的 item 值设置 null，然后通过重新设置头结点让该元素从队列里面摘除，被摘除的节点就成了孤立节点，这个节点会被在垃圾回收的时候会回收掉。另外执行分支中如果发现头节点被修改了要跳到外层循环重新获取新的头节点。

##### 2.2.3 peek 操作

peek 操作是获取队列头部一个元素（只获取不移除），如果队列为空则返回 null，下面看看实现原理。

**public** E **peek**() {

//(1)

restartFromHead:

**for** (;;) {

**for** (Node<E> h = head, p = h, q;;) {

//(2)

E item = p.item;

//(3)

**if** (item != null || (q = p.next) == null) {

updateHead(h, p);

**return** item;

}

//(4)

**else** **if** (p == q)

**continue** restartFromHead;

**else**

//(5)

p = q;

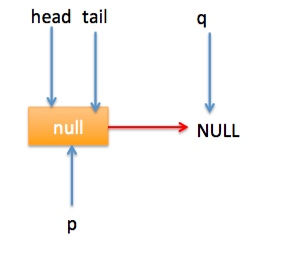
}

}

}

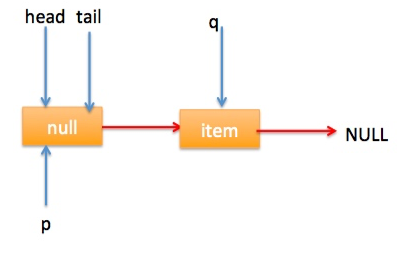
代码结构与 poll 操作类似，不同在于步骤（3）的使用只是少了 castItem 操作，其实这很正常，因为 peek 只是获取队列头元素值并不清空其值，根据前面我们知道第一次执行 offer 后 head 指向的是哨兵节点（也就是 item 为 null 的节点），那么第一次 peek 时候代码（3）中会发现 item==null, 然后会执行 q = p.next, 这时候 q 节点指向的才是队列里面第一个真正的元素或者如果队列为 null 则 q 指向 null。

* 当队列为空时候这时候队列状态为：

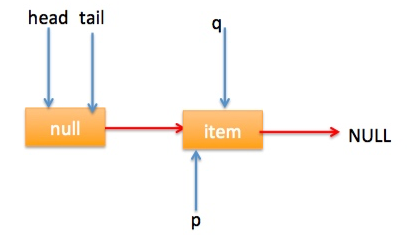


这时候执行 updateHead 由于 h 节点等于 p 节点所以不进行任何操作，然后 peek 操作会返回 null。

* 当队列至少有一个元素时候（这里假设只有一个）这时候队列状态为：



这时候执行代码（5）这时候 p 指向了 q 节点，然后执行代码（3）这时候队列状态为：



执行代码（3）发现 item 不为 null，则执行 updateHead 方法，由于 h!=p, 所以设置头结点，设置后队列状态为：



也就是剔除了哨兵节点。

总结：peek 操作代码与 poll 操作类似只是前者只获取队列头元素但是并不从队列里面删除，而后者获取后需要从队列里面删除，另外在第一次调用 peek 操作时候，会删除哨兵节点，并让队列的 head 节点指向队列里面第一个元素或者 null。

##### 2.2.4 size 操作

获取当前队列元素个数，在并发环境下不是很有用，因为 CAS 没有加锁所以从调用 size 函数到返回结果期间有可能增删元素，导致统计的元素个数不精确。

**public** **int** **size**() {

**int** count = 0;

**for** (Node<E> p = first(); p != **null**; p = succ(p))

**if** (p.item != **null**)

// 最大返回Integer.MAX\_VALUE

**if** (++count == Integer.MAX\_VALUE)

**break**;

**return** count;

}

//获取第一个队列元素（哨兵元素不算），没有则为nullNode<E> **first**() {

restartFromHead:

**for** (;;) {

**for** (Node<E> h = head, p = h, q;;) {

**boolean** hasItem = (p.item != **null**);

**if** (hasItem || (q = p.next) == **null**) {

updateHead(h, p);

**return** hasItem ? p : **null**;

}

**else** **if** (p == q)

**continue** restartFromHead;

**else**

p = q;

}

}

}

//获取当前节点的next元素，如果是自引入节点则返回真正头节点**final** Node<E> **succ**(Node<E> p) {

Node<E> next = p.next;

**return** (p == next) ? head : next;

}

##### 2.2.5 remove 操作

如果队列里面存在该元素则删除给元素，如果存在多个则删除第一个，并返回 true，否者返回 false

**public** boolean **remove**(Object o) {

//查找元素为空，直接返回false

**if** (o == null) **return** false;

Node<E> pred = null;

**for** (Node<E> p = first(); p != null; p = succ(p)) {

E item = p.item;

//相等则使用cas值null,同时一个线程成功，失败的线程循环查找队列中其它元素是否有匹配的。

**if** (item != null &&

o.**equals**(item) &&

p.casItem(item, null)) {

//获取next元素

Node<E> next = succ(p);

//如果有前驱节点，并且next不为空则链接前驱节点到next,

**if** (pred != null && next != null)

pred.casNext(p, next);

**return** true;

}

pred = p;

}

**return** false;

}

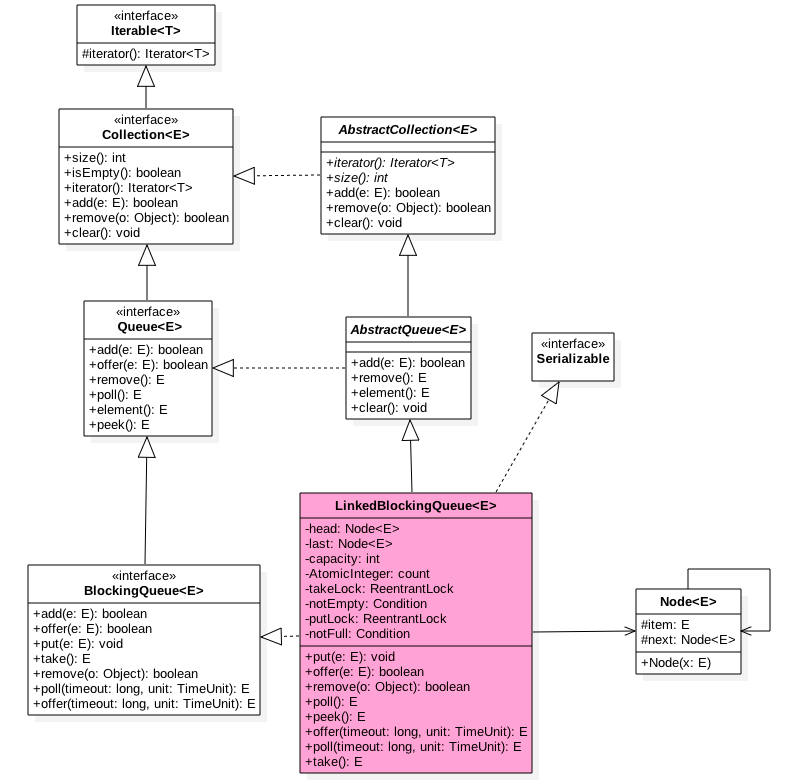
注：ConcurrentLinkedQueue 底层使用单向链表数据结构来保存队列元素，每个元素被包装为了一个 Node 节点，队列是靠头尾节点来维护的，创建队列时候头尾节点指向一个 item 为 null 的哨兵节点，第一次 peek 或者 first 时候会把 head 指向第一个真正的队列元素。由于使用非阻塞 CAS 算法，没有加锁，所以获取 size 的时候有可能进行了 offer，poll 或者 remove 操作，导致获取的元素个数不精确，所以在并发情况下 size 函数不是很有用。

### 三、LinkedBlockingQueue 原理探究

前面介绍了使用 CAS 算法实现的非阻塞队列 ConcurrentLinkedQueue，下面就来介绍下使用独占锁实现的阻塞队列 LinkedBlockingQueue 的实现

#### 3.1 LinkedBlockingQueue 类图结构

同理首先看下 LinkedBlockingQueue 的类图结构



如上类图可知 LinkedBlockingQueue 也是使用单向链表实现，也有两个 Node 分别用来存放首尾节点，并且里面有个初始值为 0 的原子变量 count 用来记录队列元素个数。另外里面有两个 ReentrantLock 的实例，分别用来控制元素入队和出队的原子性，其中 takeLock 用来控制同时只有一个线程可以从队列获取元素，其它线程必须等待，putLock 控制同时只能有一个线程可以获取锁去添加元素，其它线程必须等待。另外 notEmpty 和 notFull 是信号量，内部分别有一个条件队列用来存放进队和出队时候被阻塞的线程，其实这个是个生产者 - 消费者模型。如下是独占锁创建代码：

/\*\* 执行take, poll等操作时候需要获取该锁 \*/

**private** **final** ReentrantLock takeLock = **new** ReentrantLock();

/\*\* 当队列为空时候执行出队操作（比如take）的线程会被放入这个条件队列进行等待 \*/

**private** **final** Condition notEmpty = takeLock.newCondition();

/\*\* 执行put, offer等操作时候需要获取该锁\*/

**private** **final** ReentrantLock putLock = **new** ReentrantLock();

/\*\*当队列满时候执行进队操作（比如put)的线程会被放入这个条件队列进行等待 \*/

**private** **final** Condition notFull = putLock.newCondition();

/\*\* 当前队列元素个数 \*/

**private** **final** AtomicInteger count = **new** AtomicInteger(0);

如下是 LinkedBlockingQueue 无参构造函数代码：

**public** **static** **final** **int** MAX\_VALUE = 0x7fffffff;

**public** **LinkedBlockingQueue**() {

**this**(Integer.MAX\_VALUE);

}

**public** **LinkedBlockingQueue**(**int** capacity) {

**if** (capacity <= 0) **throw** **new** IllegalArgumentException();

**this**.capacity = capacity;

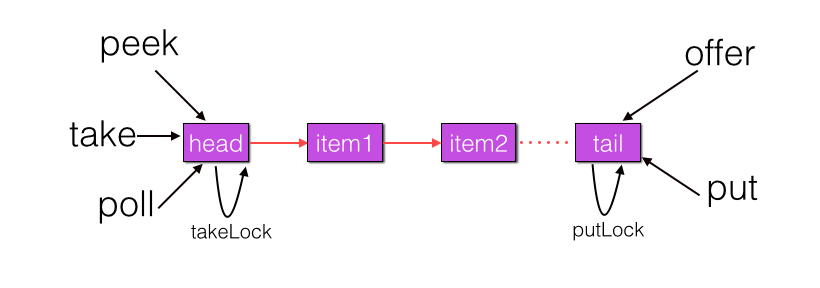
//初始化首尾节点,指向哨兵节点

last = head = **new** Node<E>(**null**);

}

从代码可知默认队列容量为 0x7fffffff; 用户也可以自己指定容量，所以一定程度上 LinkedBlockingQueue 可以说是有界阻塞队列。

首先使用一个图来概况该队列，读者在读完本节后在回头体会下:



#### 3.2 LinkedBlockingQueue 原理介绍

##### 3.2.1 offer 操作

向队列尾部插入一个元素，如果队列有空闲容量则插入成功后返回 true，如果队列已满则丢弃当前元素然后返回 false，如果 e 元素为 null 则抛出 NullPointerException 异常，另外该方法是非阻塞的。

**public** boolean **offer**(E e) {

//（1）空元素抛空指针异常

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

//(2) 如果当前队列满了则丢弃将要放入的元素，然后返回false

final AtomicInteger count = **this**.count;

**if** (count.**get**() == capacity)

**return** false;

//(3) 构造新节点，获取putLock独占锁

**int** c = -1;

Node<E> node = **new** Node<E>(e);

final ReentrantLock putLock = **this**.putLock;

putLock.**lock**();

**try** {

//(4)如果队列不满则进队列，并递增元素计数

**if** (count.**get**() < capacity) {

enqueue(node);

c = count.getAndIncrement();

//(5)

**if** (c + 1 < capacity)

notFull.signal();

}

} **finally** {

//(6)释放锁

putLock.unlock();

}

//(7)

**if** (c == 0)

signalNotEmpty();

//(8)

**return** c >= 0;

}

**private** **void** **enqueue**(Node<E> node) {

last = last.next = node;

}

* 步骤（2）判断如果当前队列已满则丢弃当前元素并返回 false
* 步骤（3）获取到 putLock 锁，当前线程获取到该锁后，则其它调用 put 和 offer 的线程将会被阻塞（阻塞的线程被放到 putLock 锁的 AQS 阻塞队列）。
* 步骤（4）这里有重新判断了下当前队列是否满了，这是因为在执行代码（2）和获取到 putLock 锁期间可能其它线程通过 put 或者 offer 方法向队列里面添加了新元素。重新判断队列确实不满则新元素入队，并递增计数器。
* 步骤（5）判断如果新元素入队后队列还有空闲空间，则唤醒 notFull 的条件队列里面因为调用了 notFull 的 await 操作（比如执行 put 方法而队列满了的时候）而被阻塞的一个线程，因为队列现在有空闲所以这里可以提前唤醒一个入队线程。
* 代码（6) 则释放获取的 putLock 锁，这里要注意锁的释放一定要在 finally 里面做，因为即使 try 块抛异常了，finally 也是会被执行到的。另外释放锁后其它因为调用 put 和 offer 而被阻塞的线程将会有一个获取到该锁。
* 代码（7）c==0 说明在执行代码（6）释放锁时候队列里面至少有一个元素，队列里面有元素则执行 signalNotEmpty，下面看看 signalNotEmpty 的代码：

**private** **void** **signalNotEmpty**() {

final ReentrantLock takeLock = **this**.takeLock;

takeLock.**lock**();

**try** {

notEmpty.signal();

} **finally** {

takeLock.unlock();

}

}

可知作用是激活 notEmpty 的条件队列中因为调用 notEmpty 的 await 方法（比如调用 take 方法并且队列为空的时候）而被阻塞的一个线程，这里也说明了调用条件变量的方法前要首先获取对应的锁。

综上可知 offer 方法中通过使用 putLock 锁保证了在队尾新增元素的原子性和队列元素个数的比较和递增操作的原子性。

##### 3.2.2 put 操作

向队列尾部插入一个元素，如果队列有空闲则插入后直接返回 true，如果队列已满则阻塞当前线程直到队列有空闲插入成功后返回 true，如果在阻塞的时候被其它线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回，另外如果 e 元素为 null 则抛出 NullPointerException 异常。

put 操作的代码结构与 offer 操作类似，代码如下：

**public** **void** **put**(E e) **throws** InterruptedException {

//（1）空元素抛空指针异常

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

//(2) 构建新节点，并获取独占锁putLock

**int** c = -1;

Node<E> node = **new** Node<E>(e);

**final** ReentrantLock putLock = **this**.putLock;

**final** AtomicInteger count = **this**.count;

putLock.lockInterruptibly();

**try** {

//(3)如果队列满则等待

**while** (count.get() == capacity) {

notFull.await();

}

//（4）进队列并递增计数

enqueue(node);

c = count.getAndIncrement();

//(5)

**if** (c + 1 < capacity)

notFull.signal();

} **finally** {

//(6)

putLock.unlock();

}

//(7)

**if** (c == 0)

signalNotEmpty();

}

* 代码（2）中使用 putLock.lockInterruptibly() 获取独占锁，相比 offer 方法中这个获取独占锁方法意味着可以被中断，具体说是当前线程在获取锁的过程中，如果被其它线程设置了中断标志则当前线程会抛出 InterruptedException 异常，所以 put 操作在获取锁过程中是可被中断的。
* 代码（3）如果当前队列已满，则调用 notFull 的 await() 把当前线程放入 notFull 的条件队列，当前线程被阻塞挂起并释放获取到的 putLock 锁，由于 putLock 锁被释放了，所以现在其它线程就有机会获取到 putLock 锁了。
* 另外考虑下代码（3）判断队列是否为空为何使用 while 循环而不是 if 语句那？其实是考虑到当前线程被虚假唤醒的问题，也就是其它线程没有调用 notFull 的 singal 方法时候 notFull.await() 在某种情况下会自动返回。如果使用 if 语句那么虚假唤醒后会执行代码（4）元素入队，并且递增计数器，而这时候队列已经是满了的，导致队列元素个数大于了队列设置的容量，导致程序出错。而使用 while 循环假如 notFull.await() 被虚假唤醒了，那么循环在检查一下当前队列是否是满的，如果是则再次进行等待。

##### 3.2.3 poll 操作

从队列头部获取并移除一个元素，如果队列为空则返回 null，该方法是不阻塞的。

**public** E **poll**() {

//(1)队列为空则返回null

final AtomicInteger count = **this**.count;

**if** (count.**get**() == 0)

**return** null;

//(2)获取独占锁

E x = null;

**int** c = -1;

final ReentrantLock takeLock = **this**.takeLock;

takeLock.**lock**();

**try** {

//(3)队列不空则出队并递减计数

**if** (count.**get**() > 0) {//3.1

x = dequeue();//3.2

c = count.getAndDecrement();//3.3

//(4)

**if** (c > 1)

notEmpty.signal();

}

} **finally** {

//(5)

takeLock.unlock();

}

//(6)

**if** (c == capacity)

signalNotFull();

//(7)返回

**return** x;

}

**private** E **dequeue**() {

Node<E> h = head;

Node<E> first = h.next;

h.next = h; // help GC

head = first;

E x = first.item;

first.item = null;

**return** x;

}

* 代码 (1) 如果当前队列为空，则直接返回 null
* 代码（2）获取独占锁 takeLock，当前线程获取该锁后，其它线程在调用 poll 或者 take 方法会被阻塞挂起
* 代码 (3) 如果当前队列不为空则进行出队操作，然后递减计数器。
* 代码（4）如果 c>1 则说明当前线程移除掉队列里面的一个元素后队列不为空（c 是删除元素前队列元素个数），那么这时候就可以激活因为调用 poll 或者 take 方法而被阻塞到 notEmpty 的条件队列里面的一个线程。
* 代码（6）说明当前线程移除队头元素前当前队列是满的，移除队头元素后队列当前至少有一个空闲位置，那么这时候就可以调用 signalNotFull 激活因为调用 put 或者 offer 而被阻塞放到 notFull 的条件队列里的一个线程，signalNotFull 的代码如下：

**private** **void** **signalNotFull**() {

final ReentrantLock putLock = **this**.putLock;

putLock.**lock**();

**try** {

notFull.signal();

} **finally** {

putLock.unlock();

}

}

poll 代码逻辑比较简单，值得注意的是获取元素时候只操作了队列的头节点。

##### 3.2.4 peek 操作

获取队列头部元素但是不从队列里面移除，如果队列为空则返回 null，该方法是不阻塞的。

**public** E **peek**() {

//(1)

**if** (count.**get**() == 0)

**return** null;

//(2)

final ReentrantLock takeLock = **this**.takeLock;

takeLock.**lock**();

**try** {

Node<E> first = head.next;

//(3)

**if** (first == null)

**return** null;

**else**

//(4)

**return** first.item;

} **finally** {

//(5)

takeLock.unlock();

}

}

peek 操作代码也比较简单，这里需要注意的是代码（3）这里还是需要判断下 first 是否为 null 的，不能直接执行代码（4）。正常情况下执行到代码（2）说明队列不为空，但是代码（1）和（2）不是原子性操作，也就是在执行点（1）判断队列不空后，在代码（2）获取到锁前有可能其它线程执行了 poll 或者 take 操作导致队列变为了空，然后当前线程获取锁后，直接执行 first.item 会抛出空指针异常。

##### 3.2.5 take 操作

获取当前队列头部元素并从队列里面移除，如果队列为空则阻塞调用线程。如果队列为空则阻塞当前线程直到队列不为空然后返回元素，如果在阻塞的时候被其它线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。

**public** E **take**() **throws** InterruptedException {

E x;

**int** c = -1;

**final** AtomicInteger count = **this**.count;

//(1)获取锁

**final** ReentrantLock takeLock = **this**.takeLock;

takeLock.lockInterruptibly();

**try** {

//(2)当前队列为空则阻塞挂起

**while** (count.get() == 0) {

notEmpty.await();

}

//(3)出队并递减计数

x = dequeue();

c = count.getAndDecrement();

//(4)

**if** (c > 1)

notEmpty.signal();

} **finally** {

//(5)

takeLock.unlock();

}

//(6)

**if** (c == capacity)

signalNotFull();

//(7)

**return** x;

}

* 代码（1）当前线程获取到独占锁，其它调用 take 或者 poll 的线程将会被阻塞挂起。
* 代码（2）如果队列为空则阻塞挂起当前线程，并把当前线程放入 notEmpty 的条件队列。
* 代码（3）进行出队操作并递减计数。
* 代码（4）如果 c>1 说明当前队列不为空，则唤醒 notEmpty 的条件队列的条件队列里面的一个因为调用 take 或者 poll 而被阻塞的线程。
* 代码（5）释放锁。
* 代码（6）如果 c == capacity 则说明当前队列至少有一个空闲位置，则激活条件变量 notFull 的条件队列里面的一个因为调用 put 或者 offer 而被阻塞的线程。

##### 3.2.6 remove 操作

删除队列里面指定元素，有则删除返回 true，没有则返回 false

**public** boolean **remove**(Object o) {

**if** (o == null) **return** false;

//（1）双重加锁

fullyLock();

**try** {

//（2)遍历队列找则删除返回true

**for** (Node<E> trail = head, p = trail.next;

p != null;

trail = p, p = p.next) {

//(3)

**if** (o.**equals**(p.item)) {

unlink(p, trail);

**return** true;

}

}

//(4)找不到返回false

**return** false;

} **finally** {

//(5)解锁

fullyUnlock();

}

}

* 代码（1）通过 fullyLock 获取双重锁，当前线程获取后，其它线程进行入队或者出队的操作时候就会被阻塞挂起。

**void** **fullyLock**() {

putLock.**lock**();

takeLock.**lock**();

}

* 代码（2）遍历队列寻找要删除的元素，找不到则直接返回 false，找到则执行 unlink 操作，unlik 操作代码如下：

**void** **unlink**(Node<E> p, Node<E> trail) {

p.item = null;

trail.next = p.next;

**if** (last == p)

last = trail;

如果当前队列满，删除后，也不忘记唤醒等待的线程

**if** (count.getAndDecrement() == capacity)

notFull.signal();

}

可知删除元素后，如果发现当前队列有空闲空间，则唤醒 notFull 的条件队列中一个因为调 用 put 或者 offer 方法而被阻塞的线程。

* 代码（5）调用 fullyUnlock 方法使用与加锁顺序相反的顺序释放双重锁

**void** **fullyUnlock**() {

takeLock.unlock();

putLock.unlock();

}

总结下，由于 remove 方法在删除指定元素前加了两把锁，所以在遍历队列查找指定元素过程中是线程安全的，并且此时其它调用入队出队操作的线程全部会被阻塞，另外获取多个资源锁与释放的顺序是相反的。

##### 3.2.7 size 操作

* int size() : 获取当前队列元素个数。

**public** **int** **size**() {

**return** count.**get**();

}

由于在操作出队入队时候操作 Count 的时候是加了锁的，所以相比 ConcurrentLinkedQueue 的 size 方法比较准确。这里考虑下为何 ConcurrentLinkedQueue 中需要遍历链表来获取 size 而不适用一个原子变量那？这是因为使用原子变量保存队列元素个数需要保证入队出队操作和操作原子变量是原子性操作，而 ConcurrentLinkedQueue 是使用 CAS 无锁算法的，所以无法做到这个。

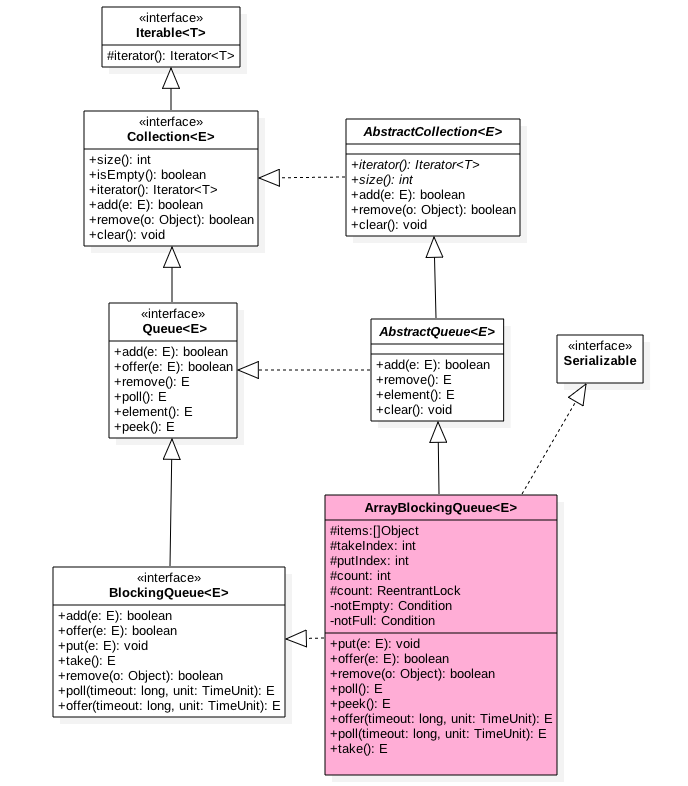
注：LinkedBlockingQueue 内部是通过单向链表实现，使用头尾节点来进行入队和出队操作，也就是入队操作都是对尾节点进行操作，出队操作都是对头节点进行操作，而头尾节点的操作分别使用了单独的独占锁保证了原子性，所以出队和入队操作是可以同时进行的。另外头尾节点的独占锁都配备了一个条件队列，用来存放被阻塞的线程，并结合入队出队操作实现了一个生产消费模型。

### 四、ArrayBlockingQueue 原理探究

上节介绍了有界链表方式的阻塞队列 LinkedBlockingQueue，本节来研究下有界使用数组方式实现的阻塞队列 ArrayBlockingQueue 的原理

#### 4.1 ArrayBlockingQueue 类图结构

同理为了能从全局一览 ArrayBlockingQueue 的内部构造，先看下类图：



如图 ArrayBlockingQueue 内部有个数组 items 用来存放队列元素，putindex 变量标示入队元素下标，takeIndex 是出队下标，count 统计队列元素个数，从定义可知并没有使用 volatile 修饰，这是因为访问这些变量使用都是在锁块内，而加锁已经保证了锁块内变量的内存可见性了。

另外有个独占锁 lock 用来保证出入队操作原子性，这保证了同时只有一个线程可以进行入队出队操作，另外 notEmpty，notFull 条件变量用来进行出入队的同步。

另外由于 ArrayBlockingQueue 是有界队列，所以构造函数必须传入队列大小参数，构造函数代码如下：

**public** **ArrayBlockingQueue**(**int** capacity) {

**this**(capacity, false);

}

**public** **ArrayBlockingQueue**(**int** capacity, boolean fair) {

**if** (capacity <= 0)

**throw** **new** IllegalArgumentException();

**this**.items = **new** Object[capacity];

**lock** = **new** ReentrantLock(fair);

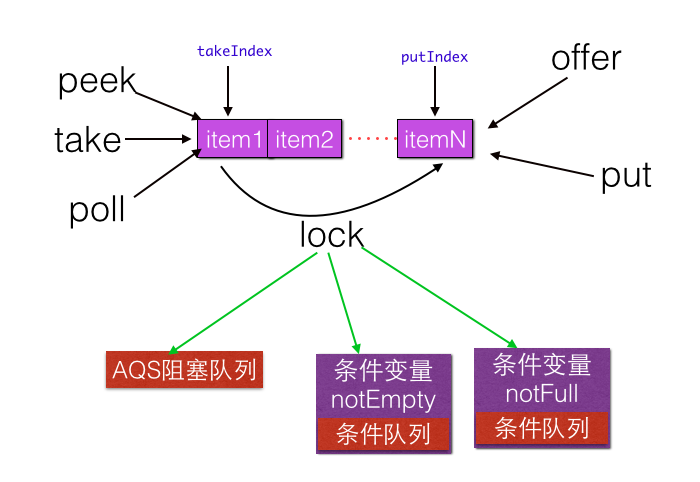
notEmpty = **lock**.newCondition();

notFull = **lock**.newCondition();

}

可知默认情况下使用的是 ReentrantLock 提供的非非公平独占锁进行出入队操作的加锁。

首先一个图概况该队列，读者可以读完本节后在回头体会下：



#### 4.2 ArrayBlockingQueue 原理介绍

本节主要讲解下面几个主要函数的原理。

##### 4.2.1 offer 操作

向队列尾部插入一个元素，如果队列有空闲容量则插入成功后返回 true，如果队列已满则丢弃当前元素然后返回 false，如果 e 元素为 null 则抛出 NullPointerException 异常，另外该方法是不阻塞的。

**public** boolean **offer**(E e) {

//（1）e为null，则抛出NullPointerException异常

checkNotNull(e);

//（2）获取独占锁

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

//（3）如果队列满则返回false

**if** (count == items.length)

**return** false;

**else** {

//（4）否者插入元素

enqueue(e);

**return** true;

}

} **finally** {

**lock**.unlock();

}

}

* 代码（2）获取独占锁，当前线程获取该锁后，其它入队和出队操作的线程都会被阻塞挂起后放入 lock 锁的 AQS 阻塞队列。
* 代码（3）如果队列满则直接返回 false，否者调用 enqueue 方法后返回 true，enqueue 的代码如下：

**private** **void** **enqueue**(E x) {

//（6）元素入队

**final** Object[] items = **this**.items;

items[putIndex] = x;

//（7）计算下一个元素应该存放的下标

**if** (++putIndex == items.length)

putIndex = 0;

count++;

//(8)

notEmpty.signal();

}

如上代码首先把当前元素放入 items 数组，然后计算下一个元素应该存放的下标，然后递增元素个数计数器，最后激活 notEmpty 的条件队列中因为调用 poll 或者 take 操作而被阻塞的的一个线程。这里由于在操作共享变量比如 count 前加了锁，所以不存在内存不可见问题，加过锁后获取的共享变量都是从主内存获取的，而不是在 CPU 缓存或者寄存器里面的值。

* 代码（5）释放锁，释放锁后会把修改的共享变量值比如 Count 的值刷新回主内存中，这样其它线程通过加锁在次读取这些共享变量后就可以看到最新的值。

##### 4.2.2 put 操作

向队列尾部插入一个元素，如果队列有空闲则插入后直接返回 true，如果队列已满则阻塞当前线程直到队列有空闲插入成功后返回 true，如果在阻塞的时候被其它线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回，另外如果 e 元素为 null 则抛出 NullPointerException 异常。

**public** **void** **put**(E e) throws InterruptedException {

//(1)

checkNotNull(e);

final ReentrantLock **lock** = **this**.**lock**;

//(2)获取锁（可被中断）

**lock**.lockInterruptibly();

**try** {

//(3)如果队列满，则把当前线程放入notFull管理的条件队列

**while** (count == items.length)

notFull.**await**();

//(4)插入元素

enqueue(e);

} **finally** {

//(5)

**lock**.unlock();

}

}

* 代码（2）在获取锁的过程中当前线程被其它线程中断了，则当前线程会抛出 InterruptedException 异常而退出。
* 代码（3）判断如果当前队列满了，则把当前线程阻塞挂起后放入到 notFull 的条件队列，注意这里也是使用了 while 而不是 if。
* 代码（4）如果队列不满则插入当前元素，此处不再累述。

##### 4.2.3 poll 操作

从队列头部获取并移除一个元素，如果队列为空则返回 null，该方法是不阻塞的。

**public** E **poll**() {

//(1)获取锁

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

//（2）当前队列为空则返回null,否者调用dequeue（）获取

**return** (count == 0) ? null : dequeue();

} **finally** {

//(3)释放锁

**lock**.unlock();

}

}

* 代码（1）获取独占锁
* 代码（2）如果队列为空则返回 null，否者调用 dequeue() 方法，dequeue 代码如下：

**private** E **dequeue**() {

**final** Object[] items = **this**.items;

//（4）获取元素值

@SuppressWarnings("unchecked")

E x = (E) items[takeIndex];

//（5）数组中值值为null;

items[takeIndex] = **null**;

//（6）队头指针计算，队列元素个数减一

**if** (++takeIndex == items.length)

takeIndex = 0;

count--;

//（7）发送信号激活notFull条件队列里面的一个线程

notFull.signal();

**return** x;

}

可知首先获取当前队头元素保存到局部变量，然后重置队头元素为 null，并重新设置队头下标，元素计数器递减，最后发送信号激活 notFull 的条件队列里面一个因为调用 put 或者 offer 而被阻塞的线程。

##### 4.2.4 take 操作

获取当前队列头部元素并从队列里面移除，如果队列为空则阻塞调用线程。如果队列为空则阻塞当前线程直到队列不为空然后返回元素，如果在阻塞的时候被其它线程设置了中断标志，则被阻塞线程会抛出 InterruptedException 异常而返回。

**public** E **take**() throws InterruptedException {

//(1)获取锁

final ReentrantLock **lock** = **this**.**lock**;

**lock**.lockInterruptibly();

**try** {

//（2）队列为空，则等待，直到队列有元素

**while** (count == 0)

notEmpty.**await**();

//（3）获取队头元素

**return** dequeue();

} **finally** {

//(4) 释放锁

**lock**.unlock();

}

}

take 操作的代码也比较简单与 poll 相比只是步骤（2）如果队列为空则把当前线程挂起后放入到 notEmpty 的条件队列，等其它线程调用 notEmpty.signal() 方法后在返回，需要注意的是这里也是使用 while 循环进行检测并等待而不是使用 if。

##### 4.2.5 peek 操作

获取队列头部元素但是不从队列里面移除，如果队列为空则返回 null，该方法是不阻塞的。

**public** E **peek**() {

//(1)获取锁

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

//（2）

**return** itemAt(takeIndex);

} **finally** {

//(3)

**lock**.unlock();

}

}

@SuppressWarnings("unchecked")final E **itemAt**(**int** i) {

**return** (E) items[i];

}

peek 的实现更简单，首先获取独占锁，然后从数组 items 中获取当前队头下标的值并返回，在返回前释放了获取的锁。

##### 4.2.6 size 操作

获取当前队列元素个数。

**public** **int** **size**() {

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

**return** count;

} **finally** {

**lock**.unlock();

}

}

size 操作是简单的，获取锁后直接返回 count，并在返回前释放锁。也许你会疑问这里有没有修改 Count 的值，只是简单的获取下，为何要加锁那？其实如果 count 声明为 volatile 这里就不需要加锁了，因为 volatile 类型变量保证了内存的可见性，而 ArrayBlockingQueue 的设计中 count 并没有声明为 volatile，是因为 count 的操作都是在获取锁后进行的，而获取锁的语义之一是获取锁后访问的变量都是从主内存获取的，这保证了变量的内存可见性。

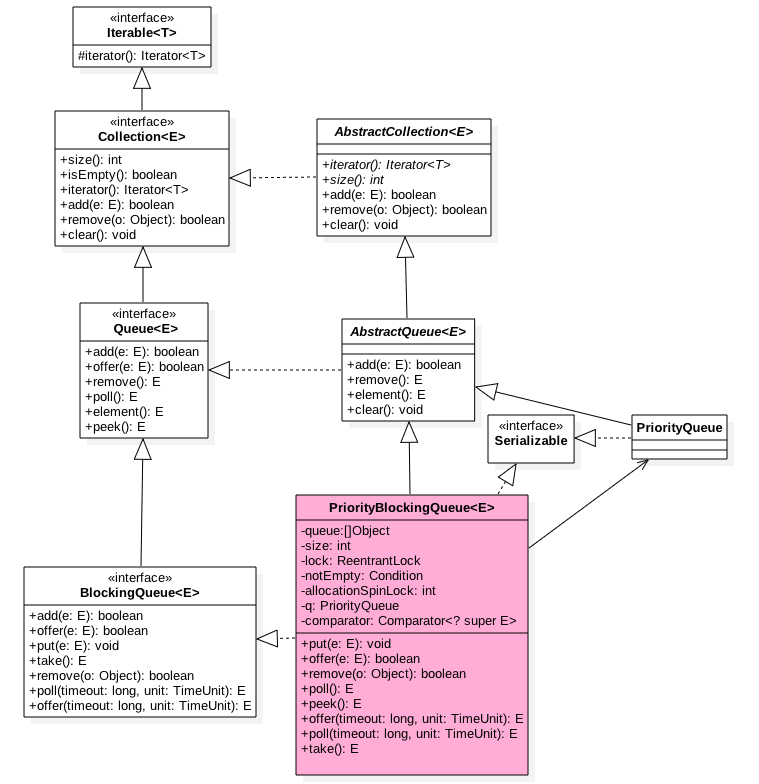
注：ArrayBlockingQueue 通过使用全局独占锁实现同时只能有一个线程进行入队或者出队操作，这个锁的粒度比较大，有点类似在方法上添加 synchronized 的意味。ArrayBlockingQueue 的 size 操作的结果是精确的，因为计算前加了全局锁。

### 五、PriorityBlockingQueue 原理探究

PriorityBlockingQueue 是带优先级的无界阻塞队列，每次出队都返回优先级最高或者最低的元素，内部是平衡二叉树堆的实现。

#### 5.1 PriorityBlockingQueue 类图结构

下面首先通过类图来从全局了解下 PriorityBlockingQueue 的结构



如图 PriorityBlockingQueue 内部有个数组 queue 用来存放队列元素，size 用来存放队列元素个数，allocationSpinLock 是个自旋锁，用 CAS 操作来保证同时只有一个线程可以扩容队列，状态为 0 或者 1，其中 0 表示当前没有在进行扩容，1 标示当前正在扩容。

如下构造函数，默认队列容量为 11，默认比较器为 null，也就是使用元素的 compareTo 方法进行比较来确定元素的优先级，这意味着队列元素必须实现了 Comparable 接口;

**private** **static** final **int** DEFAULT\_INITIAL\_CAPACITY = 11;

**public** **PriorityBlockingQueue**() {

**this**(DEFAULT\_INITIAL\_CAPACITY, null);

}

**public** **PriorityBlockingQueue**(**int** initialCapacity) {

**this**(initialCapacity, null);

}

**public** **PriorityBlockingQueue**(**int** initialCapacity,

Comparator<? super E> comparator) {

**if** (initialCapacity < 1)

**throw** **new** IllegalArgumentException();

**this**.**lock** = **new** ReentrantLock();

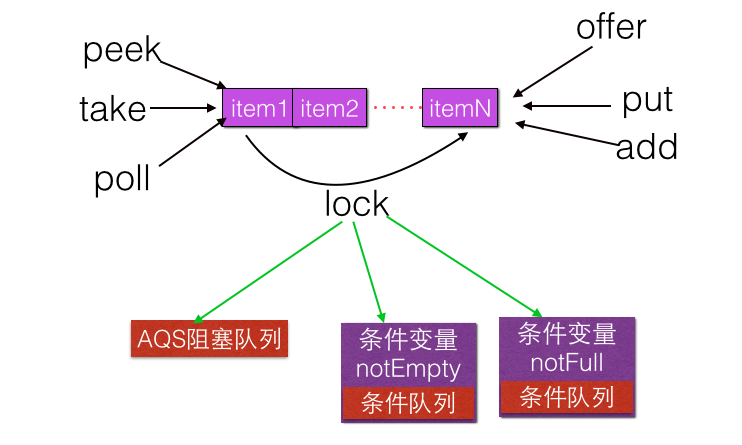
**this**.notEmpty = **lock**.newCondition();

**this**.comparator = comparator;

**this**.queue = **new** Object[initialCapacity];

}

首先通过一个图来对该队列进行概况，读者读完本机后，可以回头在体会下：



#### 5.2 原理介绍

##### 5.2.1 offer 操作

offer 操作作用是在队列插入一个元素，由于是无界队列，所以一直返回 true，如下是 offer 函数的代码：

**public** boolean **offer**(E e) {

**if** (e == null)

**throw** **new** NullPointerException();

//获取独占锁

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**int** n, cap;

Object[] array;

//如果当前元素个数>=队列容量，则扩容(1)

**while** ((n = size) >= (cap = (array = queue).length))

tryGrow(array, cap);

**try** {

Comparator<? super E> cmp = comparator;

//默认比较器为null (2)

**if** (cmp == null)

siftUpComparable(n, e, array);

**else**

//自定义比较器 (3)

siftUpUsingComparator(n, e, array, cmp);

//队列元素增加1，并且激活notEmpty的条件队列里面的一个阻塞线程（9）

size = n + 1;

notEmpty.signal();//激活调用take（）方法被阻塞的线程

} **finally** {

//释放独占锁

**lock**.unlock();

}

**return** true;

}

如上代码，主流程比较简单，下面主要看看如何进行扩容的和内部如何建堆的，首先看下扩容逻辑：

**private** **void** **tryGrow**(Object[] array, **int** oldCap) {

**lock**.unlock(); //释放获取的锁

Object[] newArray = null;

//cas成功则扩容(4)

**if** (allocationSpinLock == 0 &&

UNSAFE.compareAndSwapInt(**this**, allocationSpinLockOffset,

0, 1)) {

**try** {

//oldGap<64则扩容新增oldcap+2,否者扩容50%，并且最大为MAX\_ARRAY\_SIZE

**int** newCap = oldCap + ((oldCap < 64) ?

(oldCap + 2) : // grow faster if small

(oldCap >> 1));

**if** (newCap - MAX\_ARRAY\_SIZE > 0) { // possible overflow

**int** minCap = oldCap + 1;

**if** (minCap < 0 || minCap > MAX\_ARRAY\_SIZE)

**throw** **new** OutOfMemoryError();

newCap = MAX\_ARRAY\_SIZE;

}

**if** (newCap > oldCap && queue == array)

newArray = **new** Object[newCap];

} **finally** {

allocationSpinLock = 0;

}

}

//第一个线程cas成功后，第二个线程会进入这个地方，然后第二个线程让出cpu，尽量让第一个线程执行下面点获取锁，但是这得不到肯定的保证。(5)

**if** (newArray == null) // back off if another thread is allocating

Thread.**yield**();

**lock**.**lock**();//(6)

**if** (newArray != null && queue == array) {

queue = newArray;

System.arraycopy(array, 0, newArray, 0, oldCap);

}

}

tryGrow 目的是扩容，这里要思考下为啥在扩容前要先释放锁，然后使用 cas 控制只有一个线程可以扩容成功。其实这里不先释放锁，也是可行的，也就是在整个扩容期间一直持有锁，但是扩容是需要花时间的，如果扩容时候还占用锁那么其它线程在这个时候是不能进行出队和入队操作的，这大大降低了并发性。所以为了提高性能，使用 CAS 控制只有一个线程可以进行扩容，并且在扩容前释放了锁，让其它线程可以进行入队出队操作。

spinlock 锁使用 CAS 控制只有一个线程可以进行扩容，CAS 失败的线程会调用 Thread.yield() 让出 cpu，目的意在让扩容线程扩容后优先调用 lock.lock 重新获取锁，但是这得不到一定的保证。有可能 yield 的线程在扩容线程扩容完成前已经退出，并执行代码（6）获取到了锁，这时候获取到的锁的线程发现 newArray 为 null 就会执行代码（1）。如果当前数组扩容还没完毕，当前线程会再次调用 tryGrow 方法，然后释放锁，这又给扩容线程获取锁提供了机会，如果这时候扩容线程还没扩容完毕，则当前线程释放锁后有调用 yield 方法出让 CPU。可知当扩容线程进行扩容期间，其他线程是原地自旋通过代码（1）检查当前扩容是否完毕，等扩容完毕后才退出代码（1）的循环。

当扩容线程扩容完毕后会重置自旋锁变量 allocationSpinLock 为 0，这里并没有使用 UNSAFE 方法的 CAS 进行设置是因为同时只可能有一个线程获取了该锁，并且 allocationSpinLock 被修饰为了 volatile。

当扩容线程扩容完毕后会执行代码 (6) 获取锁，获取锁后复制当前 queue 里面的元素到新数组。

然后看下具体建堆算法：

**private** **static** <T> void siftUpComparable(int k, T x, Object[] **array**) {

Comparable<? super T> key = (Comparable<? super T>) x;

//队列元素个数>0则判断插入位置，否者直接入队(7)

**while** (k > 0) {

int **parent** = (k - 1) >>> 1;

Object e = **array**[**parent**];

**if** (key.compareTo((T) e) >= 0)

**break**;

**array**[k] = e;

k = **parent**;

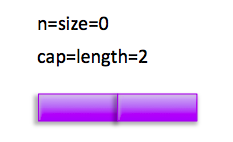
}

**array**[k] = key;(8)

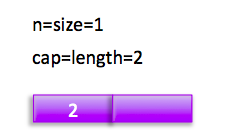
}

下面用图来解释上面算法过程，假设队列初始化容量为 2, 创建的优先级队列的泛型参数为 Integer。

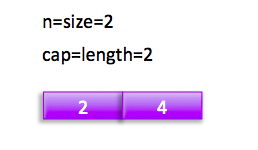
* 首先调用队列的 offer(2) 方法，希望插入元素 2 到队列，插入前队列状态如下图：



首先执行代码（1)，从上图变量值可知判断值为 false，所以紧接着执行代码（2），由于 k=n=size=0 所以代码（7）判断结果为 false，所以会执行代码（8）直接把元素 2 入队，最后执行代码（9）设置 size 的值加 1，这时候队列的状态如下图：



* 然后调用队列的 offer(4) 时候，首先执行代码（1)，从上图变量值可知判断为 false，所以执行代码（2），由于 k=1, 所以进入 while 循环，由于 parent=0;e=2;key=4; 默认元素比较器是使用元素的 compareTo 方法，可知 key>e 所以执行 break 退出 siftUpComparable 中的循环; 然后把元素存到数组下标为 1 的地方，最后执行代码（9）设置 size 的值加 1，这时候队列状态为：



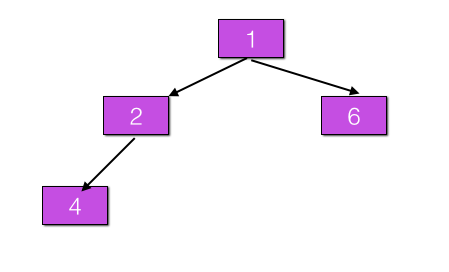
* 然后调用队列的 offer(6) 时候，首先执行代码（1)，从上图变量值知道这时候判断值为 true, 所以调用 tryGrow 进行数组扩容, 由于 2<64 所以 newCap=2 + (2+2)=6; 然后创建新数组并拷贝，然后调用 siftUpComparable 方法，由于 k=2>0 进入 while 循环，由于 parent=0;e=2;key=6;key>e 所以 break 后退出 while 循环; 并把元素 6 放入数组下标为 2 的地方，最后设置 size 的值加 1，现在队列状态：



* 然后调用队列的 offer(1) 时候，首先执行代码（1)，从上图变量值知道这次判断值为 false，所以执行代码（2），由于k=3, 所以进入 while 循环，由于parent=0;e=4;key=1; key<e，所以把元素 4 复制到数组下标为 3 的地方，然后 k=0 退出 while 循环；然后把元素 1 存放到下标为 0 地方，现在状态：



这时候二叉树堆的树形图如下：



可知堆的根元素是 1，也就是这是一个最小堆，那么当调用这个优先级队列的 poll 方法时候，会一次返回堆里面值最小的元素。

##### 5.2.2 poll 操作

poll 操作作用是获取队列内部堆树的根节点元素，如果队列为空，则返回 null。poll 函数代码如下：

**public** E **poll**() {

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();//获取独占锁

**try** {

**return** dequeue();

} **finally** {

**lock**.unlock();//释放独占锁

}

}

如上代码可知在进行出队操作过程中要先加锁，这意味着，当当前线程进行出队操作时候，其它线程不能再进行入队和出队操作，但是从前面介绍 offer 函数时候知道这时候可以有其它线程进行扩容，下面主要看下具体执行出队操作的 dequeue 方法的代码：

**private** E dequeue() {

//队列为空，则返回null

int n = size - 1;

**if** (n < 0)

**return** **null**;

**else** {

//获取队头元素(1)

Object[] **array** = queue;

E result = (E) **array**[0];

//获取队尾元素，并值null(2)

E x = (E) **array**[n];

**array**[n] = **null**;

Comparator<? super E> cmp = comparator;

**if** (cmp == **null**)//(3)

siftDownComparable(0, x, **array**, n);

**else**

siftDownUsingComparator(0, x, **array**, n, cmp);

size = n;//（4）

**return** result;

}

}

如上代码，如果队列为空则直接返回 null，否者执行代码（1）获取数组第一个元素作为返回值存放到变量 Result，这里需要注意下数组里面第一个元素是优先级最小或者最大的元素，出队操作就是返回这个元素。 然后代码（2）获取队列尾部元素存放到变量 x, 并且置空尾部节点，然后执行代码（3）插入变量 x 到数组下标为 0 的位置后，重新调成堆为最大或者最小堆，然后返回。这里重要的是看如何去掉堆的根节点后，使用剩下的节点重新调整为一个最大或者最小堆，下面我们看下 siftDownComparable 的代码实现：

**private** **static** <T> **void** **siftDownComparable**(**int** k, T x, Object[] **array**,

**int** n) {

**if** (n > 0) {

Comparable<? super T> key = (Comparable<? super T>)x;

**int** half = n >>> 1; // loop while a non-leaf

**while** (k < half) {

**int** child = (k << 1) + 1; // assume left child is least

Object c = **array**[child];（5）

**int** right = child + 1;（6)

**if** (right < n &&

((Comparable<? super T>) c).compareTo((T) **array**[right]) > 0)(7)

c = **array**[child = right];

**if** (key.compareTo((T) c) <= 0)(8)

**break**;

**array**[k] = c;

k = child;

}

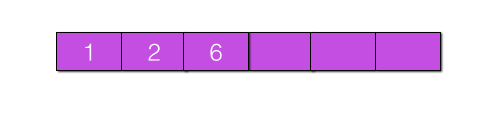
**array**[k] = key;(9)

}

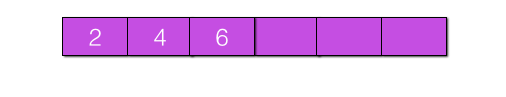
}

同理下面我们结合图来模拟上面调整堆的算法过程，接着上节队列的状态继续讲解，上节队列元素序列为 1，2，6，4：

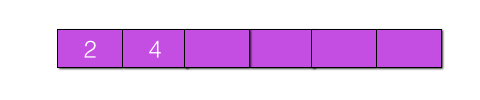
* 第一次调用队列的 poll() 方法时候，首先执行代码（1）（2），这时候变量 size =4;n=3;result=1；x=4; 这时候队列状态



然后执行代码（3）调整堆后队列状态为：



* 第二次调用队列的 poll() 方法时候，首先执行代码（1）（2），这时候变量 size =3;n=2;result=2；x=6; 这时候队列状态：



然后执行代码（3）调整堆后队列状态为：



* 第三次调用队列的 poll() 方法时候，首先执行代码（1）（2），这时候变量 size =2;n=1;result=4；x=6; 这时候队列状态：

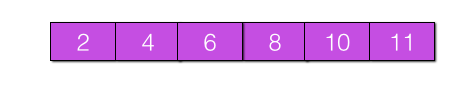


然后执行代码（3）调整堆后队列状态为：

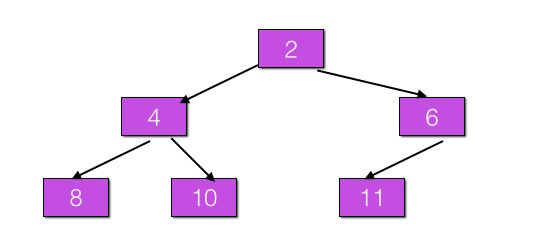


* 第四次直接返回元素 6.

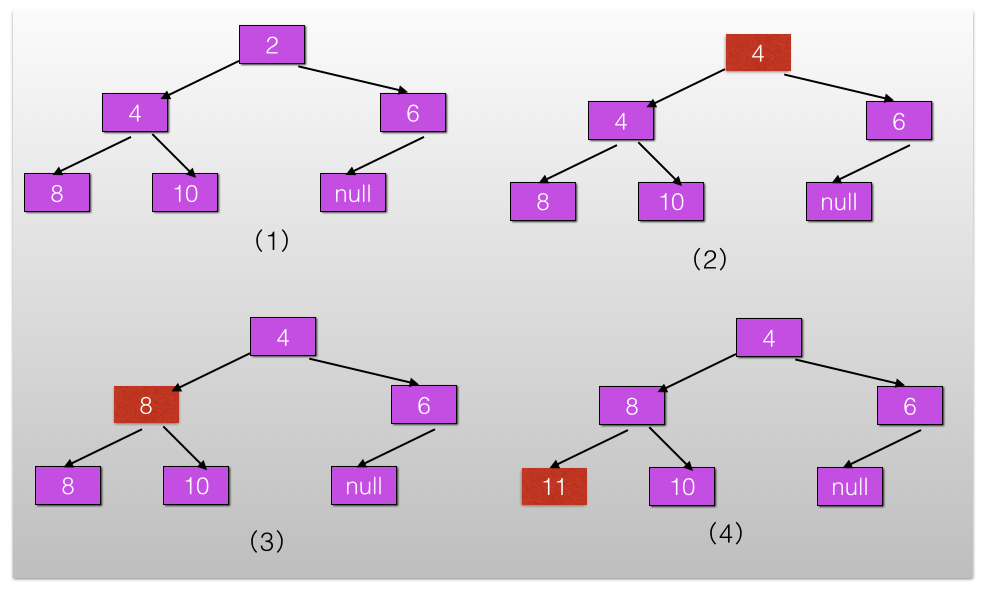
下面重点说说 siftDownComparable 这个调整堆的算法： 首先说下堆调整的思路，由于队列数组第 0 个元素为树根，出队时候要被移除，这时候数组就不在是最小堆了，所以需要调整堆，具体是要从被移除的树根的左右子树中找一个最小的值来当树根，左右子树又会看自己作为根节点的树的左右子树里面那个是最小值，这是一个递归，直到树叶节点结束递归，如果还不明白，没关系，下面结合图来说明下，假如当前队列内容如下：



其对应的二叉堆树为：



这时候如果调用了 poll(); 那么 result=2;x=11；队列末尾的元素设置为 null 后，剩下的元素调整堆的步骤如下图：



如上图（1）树根的 leftChildVal = 4;rightChildVal = 6; 4<6; 所以 c=4; 然后 11>4 也就是 key>c；所以使用元素 4 覆盖树根节点的值，现在堆对应的树如图（2）。

然后树根的左子树树根的左右孩子节点中 leftChildVal = 8;rightChildVal = 10; 8<10; 所以 c=8; 然后发现 11>8 也就是 key>c；所以元素 8 作为树根左子树的根节点，现在树的形状如图（3）, 这时候判断 k<half 为 false 就会退出循环，然后把 x=11 设置到数组下标为 3 的地方，这时候堆树如图（4），至此调整堆完毕，siftDownComparable 返回 result=2，poll 方法也返回了。

##### 5.2.3 put 操作

put 操作内部调用的 offer, 由于是无界队列，所以不需要阻塞

**public** **void** **put**(E e) {

offer(e); // never need to block

}

##### 5.2.4 take 操作

take 操作作用是获取队列内部堆树的根节点元素，如果队列为空则阻塞，如下代码：

**public** E **take**() throws InterruptedException {

//获取锁，可被中断

final ReentrantLock **lock** = **this**.**lock**;

**lock**.lockInterruptibly();

E result;

**try** {

//如果队列为空，则阻塞，把当前线程放入notEmpty的条件队列

**while** ( (result = dequeue()) == null)

notEmpty.**await**();//阻塞当前线程

} **finally** {

**lock**.unlock();//释放锁

}

**return** result;

}

如上代码，首先通过 lock.lockInterruptibly() 获取独占锁，这个方式获取的锁是对中断进行响应的。然后调用 dequeue 方法返回堆树根节点元素，如果队列为空，则返回 false，然后当前线程调用 notEmpty.await() 阻塞挂起当前线程，直到有线程调用了 offer（）方法（offer 方法内在添加元素成功后调用了 notEmpty.signal 方法会激活一个阻塞在 notEmpty 的条件队列里面的一个线程）。另外这里使用 while 而不是 if 是为了避免虚假唤醒。

##### 5.2.5 size 操作

获取队列元个数，如下代码，在返回 size 前加了锁，保证在调用 size() 方法时候不会有其它线程进行入队和出队操作，另外由于 size 变量没有被修饰为 volatie，这里加锁也保证了多线程下 size 变量的内存可见性。

**public** **int** **size**() {

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

**return** size;

} **finally** {

**lock**.unlock();

}

}

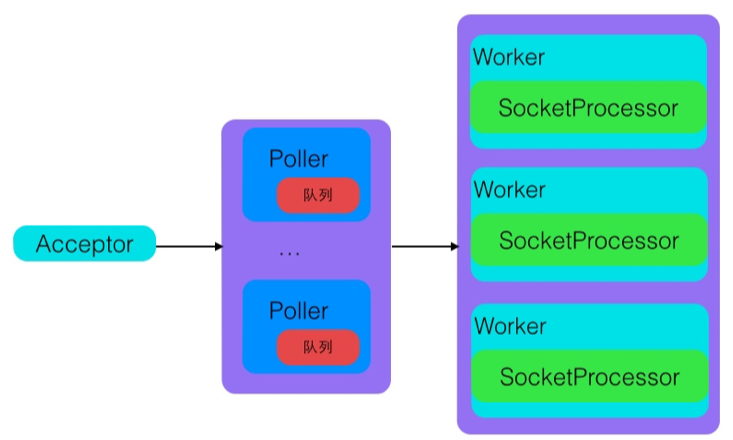
注：PriorityBlockingQueue 队列内部使用二叉树堆维护元素优先级，内部使用数组作为元素存储的数据结构，这个数组是可扩容的，当当前元素个数 >= 最大容量时候会通过算法扩容，出队时候始终保证出队的元素是堆树的根节点，而不是在队列里面停留时间最长的元素，默认元素优先级比较规则是使用元素的 compareTo 方法来做，用户可以自定义优先级的比较规则。

### 六、浅谈各种队列的对比

上面介绍的各种队列中只有 ConcurrentLinkedQueue 是使用 UNSAFE 类提供的 CAS 非阻塞算法实现的，其他几个队列内部都是使用锁来保证线程安全的。使用 CAS 算法的效率较好，那么是不是所有场景都用 ConcurrentLinkedQueue 那？

其实不然，因为 ConcurrentLinkedQueue 还是无界队列，无界队列使用不当可能造成 OOM。所以当使用 ConcurrentLinkedQueue 的时候在添加元素前应该先判断当前队列元素个数是否已经达到了设定的阈值，如果达到就做一定的处理措施，比如直接丢弃等。这里需要注意判断当前队列元素个数与阈值这个操作不是原子性的，最终会导致队列元素个数比设置的阈值大。

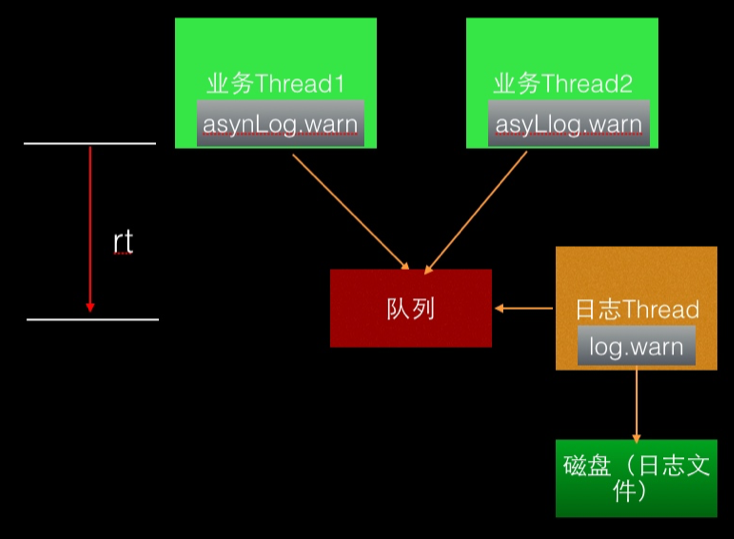
ConcurrentLinkedQueue 在 Tomcat 的的 NioEndPoint 中得到了应用，通过使用 ConcurrentLinkedQueue 将同步转换为异步，可以让 tomcat 同时接受更多请求，模型如下图：



tomcat 的 NioEndPoint 模式中 acceptor 线程负责接受用户请求，接受后把请求放入到 poll 线程对应的队列，poll 线程从队列里面获取任务后委托给 worker 线程具体处理。

LinkedBlockingQueue 和 ArrayBlockingQueue 都是有界阻塞队列，不同在于一个底层数据结构是链表，一个是数组；另外前者入队出队使用单独的锁，而后者出入队使用同一个锁，所以前者的并发度比后者高。另外创建前者时候可以不指定队列大小，默认队列元素个数为 Integer.MAX\_VALUE，而后者必须要指定数组大小。所以使用 LinkedBlockingQueue 时候要记得指定队列大小。

比如比较有名的 LogBack 日志系统的异步日志打印实现中就是用了 ArrayBlockingQueue 作为缓冲队列，如下图，业务检查调用异步 log 进行写入日志时候，实际是把日志放入了 ArrayBlockingQueue 队列就返回了，而具体真正写入日志到磁盘是一个日志线程从队列里面获取任务来做的，这其实是一个多生产单消费模型：



PriorityBlockingQueue 是无界阻塞队列，是一个队列元素有优先级的队列，前面的队列模式都是 FIFO 先进先出，而 PriorityBlockingQueue 而是优先级最高的元素先出队，而不管谁先进入队列的，所以 PriorityBlockingQueue 经常会用在一些任务具有优先级的场景。还比如上面说的 logback 异步日志模型，如果把日志等级分了优先级，比如 error>warn>info，那么上述模型中队列就可以使用 PriorityBlockingQueue，日志线程会先从队列里面首先获取 error 级别的日志，但是需要注意的是如果业务线程一直向队列里面写入 error 级别日志，那么可能先写入到队列的 warn 和 info 级别的日志将很久甚至永远没机会写入到磁盘。还有一点要注意 PriorityBlockingQueue 是无界限队列，要注意判断队列元素个数不要超过设置的阈值。

# Java 并发编程之美：并发编程高级篇之五

### 一、前言

Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之五，讲解 JUC 包中提供的三种线程同步器的使用与原理分析内容如下：（建议先阅读 [并发编程高级篇之三 - 锁](http://gitbook.cn/gitchat/activity/5ac85e1b2a04fd6c837137a2) ）

* JUC 中倒数计数器 CountDownLatch 的使用与原理分析，当需要等待多个线程执行完毕后在做一件事情时候 CountDownLatch 是比调用线程的 join 方法更好的选择，CountDownLatch 与 线程的 join 方法区别是什么？
* JUC 中 回环屏障 CyclicBarrier 的使用与分析，它也可以实现像 CountDownLatch 一样让一组线程全部到达一个状态后再全部同时执行，但是 CyclicBarrier 可以被复用。那么 CyclicBarrier 内部的实现与 CountDownLatch 有何不同那？
* JUC 中 Semaphore 的使用与原理分析，Semaphore 也是 Java 中的一个同步器，与 CountDownLatch 和 CycleBarrier 不同在于它内部的计数器是递增的，那么，Semaphore 的内部实现是怎样的那？
* 最后对上面三种同步器实现进行简单对比。

### 二、CountDownLatch 原理分析

#### 2.1 案例介绍

日常开发中经常会遇到需要在主线程中开启多线程去并行执行任务，并且主线程需要等待所有子线程执行完毕后在进行汇总的场景，在 CountDownLatch 出现之前一般都是使用线程的 join() 方法来实现，但是 join 不够灵活，不能够满足不同场景的需要，下面看一个使用 CountDownLatch 的例子:

**public** **class** **JoinCountDownLatch** {

// 创建一个CountDownLatch实例，管理计数为ThreadNum

**private** **static** **volatile** CountDownLatch countDownLatch = **new** CountDownLatch(2);

**public** **static** **void** **main**(String[] args) throws InterruptedException {

Thread threadOne = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.**out**.println("child threadOne over!");

countDownLatch.countDown();

}

});

Thread threadTwo = **new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.**out**.println("child threadTwo over!");

countDownLatch.countDown();

}

});

// 启动子线程

threadOne.start();

threadTwo.start();

System.**out**.println("wait all child thread over!");

// 等待子线程执行完毕，返回

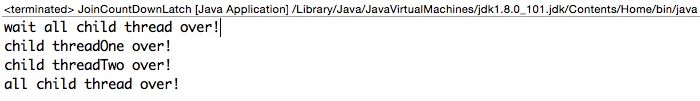
countDownLatch.**await**();

System.**out**.println("all child thread over!");

}

}

输出结果：

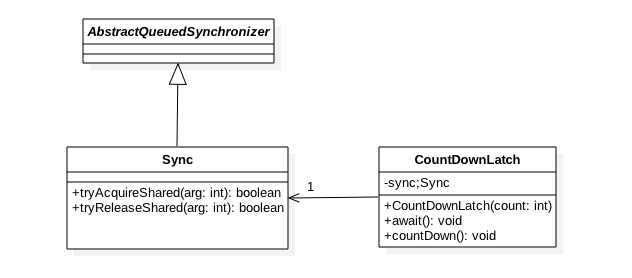


如上代码，创建了一个 CountDownLatch 实例，因为有两个子线程所以构造函数参数传递为 2，主线程调用 countDownLatch.await（）方法后会被阻塞。子线程执行完毕后调用 countDownLatch.countDown() 方法让 countDownLatch 内部的计数器减一，等所有子线程执行完毕调用 countDown（）后计数器会变为 0，这时候主线程的 await（）才会返回。

注：CountDownLatch 与 join 方法的区别，一个区别是调用一个子线程的 join（）方法后，该线程会一直被阻塞直到该线程运行完毕，而 CountDownLatch 则使用计数器允许子线程运行完毕或者运行中时候递减计数，也就是 CountDownLatch 可以在子线程运行任何时候让 await 方法返回而不一定必须等到线程结束；另外使用线程池来管理线程时候一般都是直接添加 Runable 到线程池这时候就没有办法在调用线程的 join 方法了，countDownLatch 相比 Join 方法让我们对线程同步有更灵活的控制。

#### 2.2 实现原理探究

从 CountDownLatch 的名字可以猜测内部应该有个计数器，并且这个计数器是递减的，下面就通过源码看看 JDK 开发组是何时初始化计数器，何时递减的，计数器变为 0 时候做了什么操作，多个线程是如何通过计时器值实现同步的，为了一览 CountDownLatch 内部结构，先看下类图：



从类图可知 CountDownLatch 内部还是使用 AQS 实现的，通过下面构造函数初始化了计数器的值，可知实际上是把计数器的值赋值给了 AQS 的状态值 state，也就是这里 AQS 的状态值来表示计数器值。

**public** **CountDownLatch**(**int** count) {

**if** (count < 0) **throw** **new** IllegalArgumentException("count < 0");

**this**.sync = **new** Sync(count);

}

Sync(**int** count) {

setState(count);

}

下面主要看下 CountDownLatch 中几个重要的方法内部是如何调用 AQS 来实现功能的

* void await() 方法 当线程调用了 CountDownLatch 对象的 await 方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回： （1）当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计时器值为 0 的时候； （2）其它线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程会抛出 InterruptedException 异常后返回.

下面看下 await() 方法内部是如何调用 AQS 的方法的：

//CountDownLatch的await（）方法**public** **void** **await**() **throws** InterruptedException {

sync.acquireSharedInterruptibly(1);

}

//AQS的获取共享资源时候可被中断的方法

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

**throws** InterruptedException {

//如果线程被中断则抛异常

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

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

//尝试看当前是否计数值为0，为0则直接返回，否者进入AQS的队列等待

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

doAcquireSharedInterruptibly(arg);

}

//sync类实现的AQS的接口

**protected** **int** **tryAcquireShared**(**int** acquires) {

**return** (getState() == 0) ? 1 : -1;

}

从代码可知 await() 方法委托 sync 调用了 AQS 的 acquireSharedInterruptibly 方法，该方法的特点是线程获取资源的时候可以被中断，并且获取的资源是共享资源，这里为何要用 AQS 的这个方法，而不是调用独占资源的 acquireInterruptibly 方法那？是因为这里状态值需要的并不是非 0 即 1 的效果，而是和初始化时候指定的计数器值有关系，比如你初始化时候计数器值为 8，那么 state 的值应该就有 0 到 8 的状态，而不是只有 0 和 1 的独占效果。

这里 await() 方法调用 acquireSharedInterruptibly 时候传递的是 1 就是说明要获取一个资源，而这里计数器值是资源总个数，也就意味着是让总的资源数减去 1，acquireSharedInterruptibly 内部首先判断如果当前线程被中断了则抛出异常，否者调用 sync 实现的 tryAcquireShared 方法看当前状态值（计数器值）是否为 0，是则当前线程的 await() 方法直接返回，否者调用 AQS 的 doAcquireSharedInterruptibly 让当前线程阻塞。

另外调用 tryAcquireShared 的方法仅仅是检查当前状态值是不是为 0 并没有调用 CAS 让当前状态值减去 1。

* boolean await(long timeout, TimeUnit unit)

当线程调用了 CountDownLatch 对象的该方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回： （1）当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计时器值为 0 的时候，这时候返回 true； (2) 设置的 timeout 时间到了，因为超时而返回 false； （3）其它线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程会抛出 InterruptedException 异常后返回。

**public** **boolean** **await**(**long** timeout, TimeUnit unit)

**throws** InterruptedException {

**return** sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));

}

* void countDown() 当线程调用了该方法后，会递减计数器的值，递减后如果计数器为 0 则会唤醒所有调用 await 方法而被阻塞的线程，否者什么都不做，下面看下 countDown() 方法内部是如何调用 AQS 的方法的：

//CountDownLatch的countDown（）方法

**public** **void** **countDown**() {

//委托sync调用AQS的方法

sync.releaseShared(1);

}

//AQS的方法

**public** **final** **boolean** **releaseShared**(**int** arg) {

//调用sync实现的tryReleaseShared

**if** (tryReleaseShared(arg)) {

//AQS的释放资源方法

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

如上代码可知 CountDownLatch 的 countDown（）方法是委托 sync 调用了 AQS 的 releaseShared 方法，后者调用了 sync 实现的 AQS 的 tryReleaseShared 代码如下：

//syn的方法**protected** **boolean** **tryReleaseShared**(**int** releases) {

//循环进行cas，直到当前线程成功完成cas使计数值（状态值state）减一并更新到state

**for** (;;) {

**int** c = getState();

//如果当前状态值为0则直接返回（1）

**if** (c == 0)

**return** **false**;

//CAS设置计数值减一（2）

**int** nextc = c-1;

**if** (compareAndSetState(c, nextc))

**return** nextc == 0;

}

}

如上代码可知首先获取当前状态值（计数器值），代码（1）如果当前状态值为 0 则直接返回 false，则 countDown（）方法直接返回；否者执行代码（2）使用 CAS 设置计数器减一，CAS 失败则循环重试，否者如果当前计数器为 0 则返回 true。

返回 true 后说明当前线程是最后一个调用的 countdown 方法的线程，那么该线程除了让计数器值减一外，还需要唤醒调用 CountDownLatch 的 await 方法而被阻塞的线程。

这里代码（1）貌似是多余的，其实不然，之所以添加代码（1）是为了防止当计数器值为 0 后，其它线程又调用了 countDown 方法，如果没有代码（1），状态值就会变成负数了。

* long getCount() 获取当前计数器的值，也就是 AQS 的 state 的值，一般在 debug 测试时候使用，下面看下代码：

**public** **long** **getCount**() {

**return** sync.getCount();

}

**int** **getCount**() {

**return** getState();

}

如上代码可知内部还是调用了 AQS 的 getState 方法来获取 state 的值（计数器当前值）。

### 三 、CyclicBarrier 原理分析

上面介绍的 CountDownLatch 在解决多个线程同步方面相对于调用线程的 join 已经提供了不少改进，但是 CountDownLatch 的计数器是一次性的，也就是等到计数器变为 0 后，在调用 CountDownLatch 的 await 和 countdown 方法都会立刻返回，这就起不到线程同步的效果了。CyclicBarrier 类的功能不限于 CountDownLatch 所提供的功能，从字面意思理解 CyclicBarrier 是回环屏障的意思，它可以实现让一组线程全部到达一个状态后再全部同时执行。这里之所以叫做回环是因为当所有等待线程执行完毕之后，重置 CyclicBarrier 的状态后可以被重用。

#### 3.1 案例介绍

在介绍原理前先介绍几个使用实例，下面例子我们要实现的是使用两个线程去执行一个被分解的任务 A，当两个线程把自己的任务都执行完毕后在对它们的结果进行汇总处理。

**public** **class** **CycleBarrierTest1** {

// 创建一个CyclicBarrier实例,添加一个所有子线程全部到达屏障后执行的一个任务

**private** **static** **volatile** CyclicBarrier cyclicBarrier = **new** CyclicBarrier(2, **new** Runnable() {

**public** **void** **run**() {

System.**out**.println(Thread.currentThread() + " task1 merge result");

}

});

**public** **static** **void** **main**(String[] args) throws InterruptedException {

//创建一个线程个数固定为2的线程池

ExecutorService executorService = Executors.newFixedThreadPool(2);

// 加入线程A到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " task1-1");

System.**out**.println(Thread.currentThread() + " enter in barrier");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " enter out barrier");

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

// 加入线程B到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " task1-2");

System.**out**.println(Thread.currentThread() + " enter in barrier");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " enter out barrier");

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

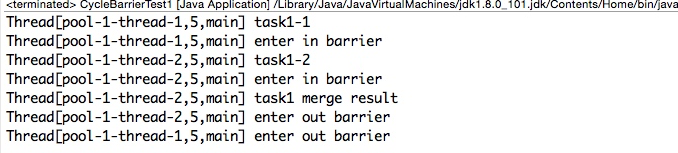
// 关闭线程池

executorService.shutdown();

}

}

输出结果：



如上代码创建了一个 CyclicBarrier 对象，第一个参数为计数器初始值，第二个参数 Runable 是指当计数器为 0 时候需要执行的任务。main 函数里面首先创建了固定大小为 2 的线程池，然后添加两个子任务到线程池，每个子任务在执行完自己的逻辑后会调用 await 方法。

一开始计数器为 2，当第一个线程调用 await 方法时候，计数器会递减为 1，由于计数器不为 0，所以当前线程就到了屏障点会被阻塞，然后第二个线程调用 await 时候，会进入屏障，计数器也会递减现在计数器为 0，就会去执行在 CyclicBarrier 构造时候的任务，执行完毕后就会退出屏障点，并且会唤醒被阻塞的第一个线程，这时候第一个线程也会退出屏障点继续向下运行。

上面的例子说明了多个线程之间是相互等待的，假如计数器为 N，那么调用 await 方法的前面 N-1 的线程都会因为到达屏障点被阻塞，当第 N 个线程调用 await 后，计数器为 0 了，这时候第 N 个线程才会发出通知唤醒前面的 N-1 个线程。也就是全部线程达到屏障点时候才能一块继续向下执行，对与这个例子来说使用 CountDownLatch 也可以达到类似输出结果，下面在放个例子来说明 CyclicBarrier 的可复用性。

假设一个任务由阶段 1、阶段 2、阶段 3 组成，每个线程要串行的执行阶段 1 和 2 和 3，多个线程执行该任务时候，必须要保证所有线程的阶段 1 全部完成后才能进行阶段 2 执行，所有线程的阶段 2 全部完成后才能进行阶段 3 执行，下面使用 CyclicBarrier 来完成这个需求。

**public** **class** **CycleBarrierTest2** {

// 创建一个CyclicBarrier实例

**private** **static** **volatile** CyclicBarrier cyclicBarrier = **new** CyclicBarrier(2);

**public** **static** **void** **main**(String[] args) throws InterruptedException {

ExecutorService executorService = Executors.newFixedThreadPool(2);

// 加入线程A到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " step1");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " step2");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " step3");

} **catch** (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

});

// 加入线程B到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " step1");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " step2");

cyclicBarrier.**await**();

System.**out**.println(Thread.currentThread() + " step3");

} **catch** (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

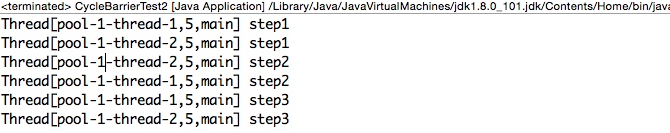
});

//关闭线程池

executorService.shutdown();

}

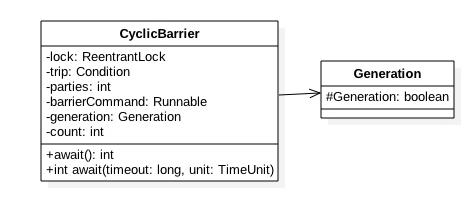
}



如上代码，在每个子线程执行完 step1 后都调用了 await 方法，所有线程都到达屏障点后才会一块往下执行，这就保证了所有线程完成了 step1 后才会开始执行 step2，然后在 step2 后面调用了 await 方法，这保证了所有线程的 step2 完成后，线程才能开始 step3 的执行，这个功能使用单个 CountDownLatch 是无法完成的。

#### 3.2 实现原理探究

为了能够一览 CyclicBarrier 的架构设计，下面先看下 CyclicBarrier 的类图结构



如上类图可知 CyclicBarrier 内部并不是直接使用 AQS 实现，而是使用了独占锁 ReentrantLock 来实现的同步；parties 用来记录线程个数，用来表示需要多少线程先调用 await 后，所有线程才会冲破屏障继续往下运行；

而 count 一开始等于 parties，每当线程调用 await 方法后就递减 1，当为 0 时候就表示所有线程都到了屏障点，另外你可能疑惑为何维护 parties 和 count 这两个变量，只有 count 不就可以了？

别忘了 cycleBarier 是可以被复用的，使用两个变量原因是用 parties 始终来记录总的线程个数，当 count 计数器变为 0 后，会使用 parties 赋值给 count，已达到复用的作用。这两个变量是在构造 CyclicBarrier 对象时候传递的，如下：

**public** **CyclicBarrier**(**int** parties, Runnable barrierAction) {

**if** (parties <= 0) **throw** **new** IllegalArgumentException();

**this**.parties = parties;

**this**.count = parties;

**this**.barrierCommand = barrierAction;

}

这里还有一个变量 barrierCommand 也通过构造函数传递而来，这是一个任务，这个任务的执行时机是当所有线程都都到达屏障点后。另外 CyclicBarrier 内部使用独占锁 Lock 来保证同时只有一个线程调用 await 方法时候才可以返回，使用 lock 首先保证了更新计数器 count 的原子性，另外使用 lock 的条件变量 trip 支持了线程间使用 notify，wait 操作进行同步。

最后变量 generation 内部就一个变量 broken 用来记录当前屏障是否被打破，另外注意这里 broken 并没有被声明为 volatile，是因为锁内使用变量不需要。

**private** **static** **class** **Generation** {

**boolean** broken = **false**;

}

下面来看下中 CyclicBarrier 几个重要的函数：

* int await() 当前线程调用 CyclicBarrier 的该方法时候当前线程会被阻塞，直到满足下面条件之一才会返回： （1） parties 个线程都调用了 await() 函数，也就是线程都到了屏障点 (2) 其它线程调用了当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常返回 （3） 当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时候，会抛出 BrokenBarrierException 异常

如下代码可知内部调用了 dowait 方法，第一个参数 false 说明不设置超时时间，这时候第二个参数没有意义

**public** **int** **await**() **throws** InterruptedException, BrokenBarrierException {

**try** {

**return** dowait(**false**, 0L);

} **catch** (TimeoutException toe) {

**throw** **new** Error(toe); // cannot happen

}

}

* boolean await(long timeout, TimeUnit unit) 当前线程调用 CyclicBarrier 的该方法时候当前线程会被阻塞，直到满足下面条件之一才会返回： （1） parties 个线程都调用了 await() 函数，也就是线程都到了屏障点，这时候返回 true (2) 当设置的超时时间到了后返回 false (3) 其它线程调用了当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常返回 （4） 当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时候，会抛出 BrokenBarrierException 异常

如下代码可知内部调用了 dowait 方法，第一个参数 true 说明设置了超时时间，这时候第二个参数是超时时间

**public** **int** **await**(**long** timeout, TimeUnit unit)

**throws** InterruptedException,

BrokenBarrierException,

TimeoutException {

**return** dowait(**true**, unit.toNanos(timeout));

}

* int dowait(boolean timed, long nanos) 该方法是实现 CyclicBarrier 的核心功能，代码如下：

**private** **int** **dowait**(boolean timed, **long** nanos)

throws InterruptedException, BrokenBarrierException,

TimeoutException {

final ReentrantLock **lock** = **this**.**lock**;

**lock**.**lock**();

**try** {

...

//(1)如果index==0说明所有线程都到到了屏障点，则执行初始化时候传递的任务

**int** index = --count;

**if** (index == 0) { // tripped

boolean ranAction = false;

**try** {

//(2)执行任务

**if** (command != null)

command.run();

ranAction = true;

//（3）激活其它调用await而被阻塞的线程，并重置CyclicBarrier

nextGeneration();

//返回

**return** 0;

} **finally** {

**if** (!ranAction)

breakBarrier();

}

}

// (4)如果index!=0

**for** (;;) {

**try** {

//（5）没有设置超时时间，

**if** (!timed)

trip.**await**();

//（6）设置了超时时间

**else** **if** (nanos > 0L)

nanos = trip.awaitNanos(nanos);

} **catch** (InterruptedException ie) {

...

}

...

}

} **finally** {

**lock**.unlock();

}

}

**private** **void** **nextGeneration**() {

//（7）唤醒条件队列里面阻塞线程

trip.signalAll();

//（8） 重置CyclicBarrier

count = parties;

generation = **new** Generation();

}

上如是 dowait 方法的主干代码，当一个线程调用了 dowait 方法后首先会获取独占锁 lock，如果创建 CycleBarrier 时候传递的参数为 10，那么后面 9 个调用线程会被阻塞；

然后当前获取线程对计数器 count 进行递减操作，递减后的 Count=index=9，因为 index!=0 所以当前线程会执行（4）处代码。如果是无参数的当前线程调用的是无参数的 await 方法，则这里 timed=false，所以当前线程会被放入条件变量 trip 的阻塞队列，当前线程会被挂起并释放获取的 Lock 锁；

如果是调用的有参数的 await 方法则 timed=true，则当前线程线程也会被放入条件变量阻塞队列并释放锁资源，但是不同的是当前线程会在指定时间超时后自动被激活。

当第一个获取锁的线程由于被阻塞释放锁后，被阻塞的 9 个线程中有一个会竞争到 lock 锁，然后执行第一个线程同样的操作，直到最后一个线程获取到 lock 时候已经有 9 个线程被放入了 Lock 的条件队列里面，最后这一个线程 count 递减后 Count=index 等于 0，所以执行（2）处代码，如果创建 CyclicBarrier 时候传递了任务，则在其它线程被唤醒前先执行任务，任务执行完毕后在执行（3）处代码，唤醒其它 9 个线程，并重置 CyclicBarrier，然后这 10 个线程就可以继续向下运行了。

### 四、 信号量 Semaphore 原理探究

Semaphore 信号量也是 Java 中一个同步器，与 CountDownLatch 和 CycleBarrier 不同在于它内部的计数器是递增的。

#### 4.1 案例介绍

类似于 CountDownLatch，下面我们的例子也是在主线程中开启两个子线程进行执行，等所有子线程执行完毕后主线程在继续向下运行。

**public** **class** **SemaphoreTest** {

// 创建一个Semaphore实例

**private** **static** **volatile** Semaphore semaphore = **new** Semaphore(0);

**public** **static** **void** **main**(String[] args) throws InterruptedException {

ExecutorService executorService = Executors.newFixedThreadPool(2);

// 加入线程A到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " over");

semaphore.release();

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

// 加入线程B到线程池

executorService.submit(**new** Runnable() {

**public** **void** **run**() {

**try** {

System.**out**.println(Thread.currentThread() + " over");

semaphore.release();

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

// 等待子线程执行完毕，返回

semaphore.acquire(2);

System.**out**.println("all child thread over!");

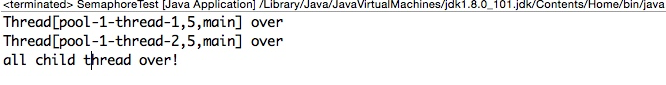
//关闭线程池

executorService.shutdown();

}

}

输出结果:

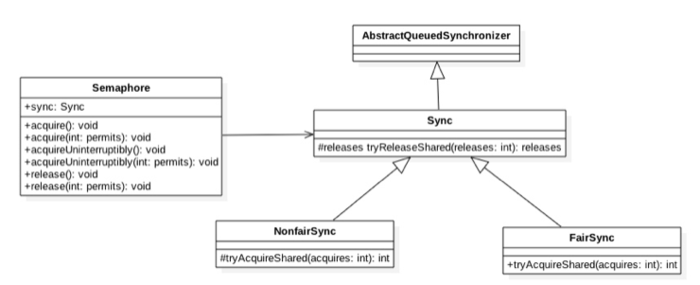


如上代码首先首先创建了一个信号量实例，构造函数的入参为 0，说明当前信号量计数器为 0，然后 main 函数添加两个线程任务到线程池，每个线程内部调用了信号量的 release 方法，相当于计数值递增一，最后在 main 线程里面调用信号量的 acquire 方法，参数传递为 2 说明调用 acquire 方法的线程会一直阻塞，直到信号量的计数变为 2 时才会返回。

看到这里也就明白了，如果构造 Semaphore 时候传递的参数为 N，在 M 个线程中调用了该信号量的 release 方法，那么在调用 acquire 对 M 个线程进行同步时候传递的参数应该是 M+N;

#### 4.2 实现原理探究

为了能够一览 Semaphore 的内部结构，首先看下 Semaphore 的类图：



如上类图可知 Semaphore 内部还是使用 AQS 来实现，Sync 只是对 AQS 的一个修饰，并且 sync 有俩个实现类分别代表获取信号量时候是否采用公平策略。如下代码创建 Semaphore 时候会有一个变量标示是否使用公平策略：

**public** **Semaphore**(**int** permits) {

sync = **new** NonfairSync(permits);

}

**public** **Semaphore**(**int** permits, **boolean** fair) {

sync = fair ? **new** FairSync(permits) : **new**

NonfairSync(permits);

}

Sync(**int** permits) {

setState(permits);

}

如上代码 Semaphore 默认采用的非公平策略，如果你需要公平策略则可以使用带两个参数的构造函数来构造 Semaphore 对象，另外和 CountDownLatch 一构造函数里面传递的初始化信号量个数 permits 被赋值给了 AQS 的 state 状态变量，也就是这里 AQS 的 state 值表示当前持有的信号量个数。

下面来看下 Semaphore 实现的主要函数：

* void acquire() 当前线程调用该方法时候目的是希望获取一个信号量资源，如果当前信号量计数个数大于 0，并且当前线程获取到了一个信号量则该方法直接返回，当前信号量的计数会减少 1. 否者会被放入 AQS 的阻塞队列，当前线程被挂起，直到其它线程调用了 release 方法释放了信号量，并且当前线程通过竞争获取到了该信号量。当当前线程被其它线程调用了 interrupt（）方法中断后，当前线程会抛出 InterruptedException 异常然后返回。下面看下代码实现：

**public** **void** **acquire**() **throws** InterruptedException {

//传递参数为1，说明要获取1个信号量资源

sync.acquireSharedInterruptibly(1);

}

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

**throws** InterruptedException {

//（1）如果线程被中断，则抛出中断异常

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

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

//（2）否者调用sync子类方法尝试获取,这里根据构造函数确定使用公平策略

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

//如果获取失败则放入阻塞队列,然后再次尝试如果失败则调用park方法挂起当前线程

doAcquireSharedInterruptibly(arg);

}

如上代码可知 acquire() 内部调用了 sync 的 acquireSharedInterruptibly 方法，后者是对中断响应的（如果当前线程被中断，则抛出中断异常），尝试获取信号量资源的 AQS 的方法 tryAcquireShared 是由 sync 的子类实现，所以这里就要分公平性了，这里讨论非公平策略 NonfairSync 类的 tryAcquireShared 方法，代码如下：

**protected** **int** **tryAcquireShared**(**int** acquires) {

**return** nonfairTryAcquireShared(acquires);

}

**final** **int** **nonfairTryAcquireShared**(**int** acquires) {

**for** (;;) {

//获取当前信号量值

**int** available = getState();

//计算当前剩余值

**int** remaining = available - acquires;

//如果当前剩余小于0或者CAS设置成功则返回

**if** (remaining < 0 ||

compareAndSetState(available, remaining))

**return** remaining;

}

}

如上代码先计算当前信号量值（available）减去需要获取的值（acquires）得到剩余的信号量个数（remaining），如果剩余值小于 0 说明当前信号量个数满足不了需求则直接返回负数，然后当前线程会被放入 AQS 的阻塞队列，当前线程被挂起。

如果剩余值大于 0 则使用 CAS 操作设置当前信号量值为剩余值，然后返回剩余值。另外可知 NonFairSync 是非公平性获取的，是说先调用 aquire 方法获取信号量的线程不一定比后来者先获取到。

下面看下公平性的 FairSync 类是如何保证公平性的：

**protected** **int** **tryAcquireShared**(**int** acquires) {

**for** (;;) {

**if** (hasQueuedPredecessors())

**return** -1;

**int** available = getState();

**int** remaining = available - acquires;

**if** (remaining < 0 ||

compareAndSetState(available, remaining))

**return** remaining;

}

}

可知公平新还是靠 hasQueuedPredecessors 这个函数来做的，前面锁章节已经讲过公平性是看当前线程节点的是否有前驱节点也在等待获取该资源，如果是则自己放弃获取的权利，然后当前线程会被放入 AQS 阻塞队列，否者就去获取。

**public** **final** **boolean** **hasQueuedPredecessors**() {

Node t = tail;

Node h = head;

Node s;

**return** h != t &&

((s = h.next) == **null** || s.thread != Thread.currentThread());

}

如上代码如果当前线程节点有前驱节点则返回 true，否者如果当前 AQS 队列为空或者当前线程节点是 AQS 的第一个节点则返回 false. 其中如果 h==t 则说明当前队列为空则直接返回 false，如果 h!=t 并且 s==null 说明有一个元素将要作为 AQS 的第一个节点入队列 (回顾下 enq 函数第一个元素入队列是两步操作，首先创建一个哨兵头节点，然后第一个元素插入到哨兵节点后面)，那么返回 true，如果 h!=t 并且 s!=null 并且 s.thread != Thread.currentThread() 则说明队列里面的第一个元素不是当前线程则返回 true.

* void acquire(int permits) 该方法与 acquire() 不同在与后者只需要获取一个信号量值，而前者则获取指定 permits 个。

**public** **void** **acquire**(**int** permits) **throws** InterruptedException {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.acquireSharedInterruptibly(permits);

}

* void acquireUninterruptibly() 该方法与 acquire() 类似，不同之处在于该方法对中断不响应，也就是当当前线程调用了 acquireUninterruptibly 获取资源过程中（包含被阻塞后）其它线程调用了当前线程的 interrupt（）方法设置了当前线程的中断标志当前线程并不会抛出 InterruptedException 异常而返回。

**public** **void** **acquireUninterruptibly**() {

sync.acquireShared(1);

}

* void acquireUninterruptibly(int permits) 该方法与 acquire(int permits) 不同在于该方法对中断不响应。

**public** **void** **acquireUninterruptibly**(**int** permits) {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.acquireShared(permits);

}

* void release() 该方法作用的把当前 semaphore 对象的信号量值增加 1，如果当前有线程因为调用 aquire 方法被阻塞放入了 AQS 的阻塞队列，则会根据公平策略选择一个线程进行激活，激活的线程会尝试获取刚增加的信号量，下面看下代码实现：

**public** **void** **release**() {

//(1)arg=1

sync.releaseShared(1);

}

**public** **final** **boolean** **releaseShared**(**int** arg) {

//(2)尝试释放资源

**if** (tryReleaseShared(arg)) {

//(3)资源释放成功则调用park唤醒AQS队列里面最先挂起的线程

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

**protected** **final** **boolean** **tryReleaseShared**(**int** releases) {

**for** (;;) {

//(4)获取当前信号量值

**int** current = getState();

//(5)当前信号量值增加releases，这里为增加1

**int** next = current + releases;

**if** (next < current) // 移除处理

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

//(6)使用cas保证更新信号量值的原子性

**if** (compareAndSetState(current, next))

**return** **true**;

}

}

如上代码 release()->sync.releaseShared(1)，可知 release 方法每次只会对信号量值增加 1，tryReleaseShared 方法是无限循环，使用 CAS 保证了 release 函数对信号量递增 1 的原子性操作。当 tryReleaseShared 函数增加信号量成功后会执行（3）处的代码，调用 AQS 的方法来激活因为调用 aquire 方法而被阻塞的线程。

* void release(int permits) 该方法与不带参数的不同之处在于前者每次调用会在信号量值原来基础上增加 permits，而后者每次增加 1.

**public** **void** **release**(**int** permits) {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.releaseShared(permits);

}

另外注意到这里调用的是 sync.releaseShared 是共享方法，这说明该信号量是线程共享的，信号量没有和固定线程绑定，多个线程可以同时使用 CAS 去更新信号量的值而不会被阻塞。

### 五、 总结

本节介绍了并发包中线程协作的一些重要类，首先 CountDownLatch 通过计数器提供了更灵活的控制，只要检测到计数器为 0，而不管当前线程是否结束调用 await 的线程就可以往下执行，相比使用 jion 必须等待线程执行完毕后主线程才会继续向下运行更灵活。

CyclicBarrier 也可以达到 CountDownLatch 的效果，但是后者当计数器变为 0 后，就不能在被复用，而前者则使用 reset 方法可以重置后复用，前者对同一个算法但是输入参数不同的类似场景下比较适用。

而 semaphore 采用了信号量递增的策略，一开始并不需要关心需要同步的线程个数，等调用 aquire 时候在指定需要同步个数，并且提供了获取信号量的公平性策略。

掌握本章的知识后，会大大减少你在 Java 中使用 wait,notify 等复杂的并发控制代码，在日常开发中当需要进行线程同步时候使用这些同步类会节省很多代码量并且可以保证正确性。

# Java 并发编程之美：并发编程实践

### 一、前言

Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

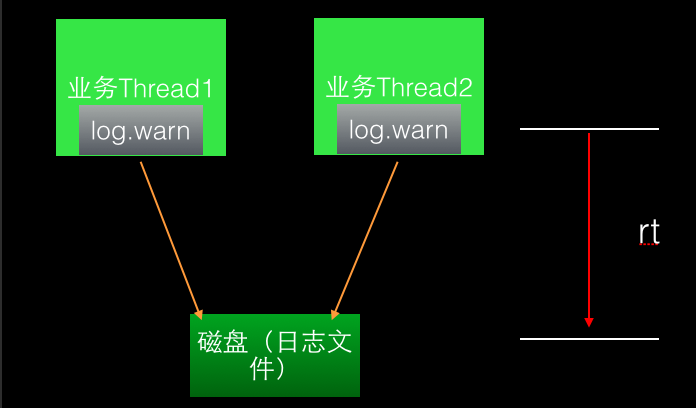
本 Chat 作为 Java 并发编程之美系列的终章，我们来讲解并发编程中的一些实践与经常会遇到的问题，内容如下：（建议先阅读 [并发编程高级篇之三 - 锁](http://gitbook.cn/gitchat/activity/5ac85e1b2a04fd6c837137a2) ）

* Logback 日志框架中异步日志打印中 ArrayBlockingQueue 的使用，Logback 是如何借助队列将同步转换为异步，节省调用线程 RT 响应时间的？
* 并发组件 ConcurrentHashMap 使用注意项，虽然 ConcurrentHashMap 是并发安全的组件，但是使用不当还是会造成程序错误，这里列出一些常见的出错点，并讲解如何避免。
* 使用定时器 Timer 的时候需要注意的一些问题，结合源码讲解出现问题的原因，以及如何避免。
* SimpleDateFormat 是线程不安全？为啥？应该如何正确使用？
* 线程池使用 FutureTask 时候需要注意的一点事，FutureTask 使用不当可能会造成调用线程一直阻塞，如何避免？
* 使用 ThreadLocal 不当可能会导致内存泄露，本节讲解为何会出现内存泄露，以及如何避免。

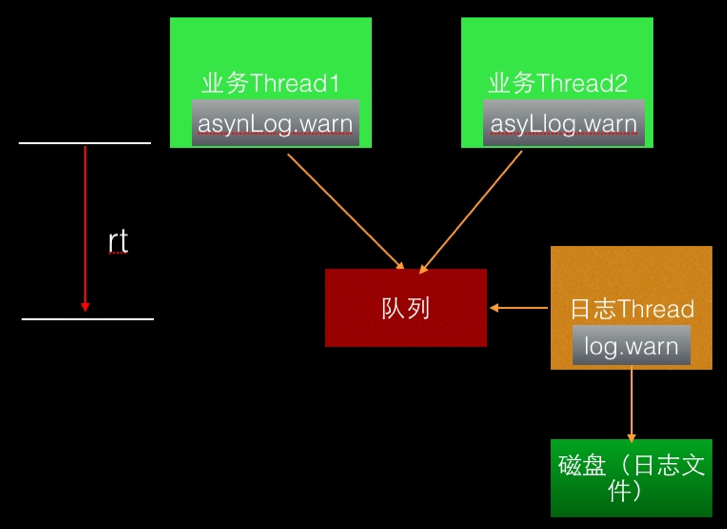
### 二、Logback 框架中异步日志打印中 ArrayBlockingQueue 的使用

#### 2.1 异步日志打印模型概述

在高并发并且响应时间要求比较小的系统中同步打日志已经满足不了需求了，这是因为打日志本身是需要同步写磁盘的，会造成 rt 增加，如下图同步日志打印模型为：



异步模型是业务线程把要打印的日志任务写入一个队列后直接返回，然后使用一个线程专门负责从队列中获取日志任务写入磁盘，其模型具体如下图：



如图可知其实 logback 的异步日志模型是一个多生产者单消费者模型，通过使用队列把同步日志打印转换为了异步，业务线程调用异步 appender 只需要把日志任务放入日志队列，日志线程则负责使用同步的 appender 进行具体的日志打印到磁盘。

#### 2.2 异步日志打印具体实现

要把同步日志打印改为异步需要修改 logback 的 xml 配置文件如下：

<**appender** name="PROJECT" class="ch.qos.logback.core.FileAppender">

<**file**>project.log</**file**>

<**encoding**>UTF-8</**encoding**>

<**append**>true</**append**>

<**rollingPolicy** class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!-- daily rollover -->

<**fileNamePattern**>project.log.%d{yyyy-MM-dd}</**fileNamePattern**>

<!-- keep 7 days' worth of history -->

<**maxHistory**>7</**maxHistory**>

</**rollingPolicy**>

<**layout** class="ch.qos.logback.classic.PatternLayout">

<**pattern**><![CDATA[

%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method} %X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n %-5level %logger{35} - %m%n

]]></**pattern**>

</**layout**>

</**appender**>

<**appender** name="asyncProject" class="ch.qos.logback.classic.AsyncAppender">

<**discardingThreshold**>0</**discardingThreshold**>

<**queueSize**>1024</**queueSize**>

<**neverBlock**>true</**neverBlock**>

<**appender-ref** ref="PROJECT" />

</**appender**>

<**logger** name="PROJECT\_LOGGER" additivity="false">

<**level** value="WARN" />

<**appender-ref** ref="asyncProject" />

</**logger**>

可知 AsyncAppender 是实现异步日志的关键，下节主要讲这个的内部实现。

## 2.3 异步日志实现原理

本文使用 logback-classic 的版本为 1.0.13。

首先从 AsyncAppender 的类图结构来从全局了解下 AsyncAppender 中组件构成：



* 如上图可知 AsyncAppender 继承自 AsyncAppenderBase，其中后者具体实现了异步日志模型的主要功能，前者只是重写了其中的一些方法。另外从上类图可知 logback 中的异步日志队列是一个阻塞队列, 后面会知道其实是一个有界阻塞队列 ArrayBlockingQueue, 其中 queueSize 是有界队列的元素个数默认为 256。
* worker 是个线程，也就是异步日志打印模型中的单消费者线程，aai 是一个 appender 的装饰器里面存放同步日志的 appender, 其中 appenderCount 记录 aai 里面附加的同步 appender 的个数；neverBlock 是当日志队列满了的时候是否阻塞打日志的线程的一个开关；discardingThreshold 是一个阈值，当日志队列里面空闲个数小于该值时候新来的某些级别的日志会被直接丢弃，下面会具体讲到。

首先我们来看下何时创建的日志队列以及何时启动的消费线程，这需要看下 AsyncAppenderBase 的 start 方法，该方法是在解析完毕配置 AsyncAppenderBase 的 xml 的节点元素后被调用：

**public** **void** **start**() {

...

//（1）日志队列为有界阻塞队列

blockingQueue = **new** ArrayBlockingQueue<E>(queueSize);

//(2)如果没设置discardingThreshold则设置为队列大小的1/5

**if** (discardingThreshold == UNDEFINED)

discardingThreshold = queueSize / 5;

//(3)设置消费线程为守护线程，并设置日志名称

worker.setDaemon(**true**);

worker.setName("AsyncAppender-Worker-" + worker.getName());

//(4)设置启动消费线程

**super**.start();

worker.start();

}

从上代码可知 logback 使用的队列是有界队列 ArrayBlockingQueue，之所以使用有界队列是考虑到内存溢出问题，在高并发下写日志的 qps 会很高如果设置为无界队列队列本身会占用很大内存，很可能会造成 OOM。

这里消费日志队列的 worker 线程被设置为了守护线程，意味着当主线程运行结束并且当前没有用户线程时候该 worker 线程会随着 JVM 的退出而终止，而不管日志队列里面是否还有日志任务未被处理。另外这里设置了线程的名称是个很好的习惯，因为这在查找问题的时候很有帮助，根据线程名字就可以定位到是哪个线程。

既然是有界队列那么肯定需要考虑如果队列满了，该如何处置，是丢弃老的日志任务，还是阻塞日志打印线程直到队列有空余元素那? 要回答这个问题，我们需要看看具体进行日志打印的 AsyncAppenderBase 的 append 方法：

**protected** **void** **append**(E eventObject) {

//（5）调用AsyncAppender重写的isDiscardable

**if** (isQueueBelowDiscardingThreshold() && isDiscardable(eventObject)) {

**return**;

}

...

//（6）放入日志任务到队列

put(eventObject);

}

**private** **boolean** **isQueueBelowDiscardingThreshold**() {

**return** (blockingQueue.remainingCapacity() < discardingThreshold);

}

其中 (5) 调用了调用 AsyncAppender 重写的 isDiscardable 具体内容为：

//(7)

**protected** boolean **isDiscardable**(ILoggingEvent **event**) {

Level level = **event**.getLevel();

**return** level.toInt() <= Level.INFO\_INT;

}

结合 (5)(7) 可知如果当前日志的级别小于 INFO\_INT 级别并且当前队列的剩余容量小于 discardingThreshold 时候会直接丢弃这些日志任务。

下面看具体步骤 (6) 的 put 方法：

**private** **void** **put**(E eventObject) {

//(8)

**if** (neverBlock) {

blockingQueue.offer(eventObject);

} **else** {

**try** {//(9)

blockingQueue.put(eventObject);

} **catch** (InterruptedException e) {

// Interruption of current thread when in doAppend method should not be consumed

// by AsyncAppender

Thread.currentThread().interrupt();

}

}

}

可知如果 neverBlock 设置为了 false（默认为 false）则会调用阻塞队列的 put 方法，而 put 是阻塞的，也就是说如果当前队列满了，如果在企图调用 put 方法向队列放入一个元素则调用线程会被阻塞直到队列有空余空间。

这里有必要提下其中第 (9) 步当日志队列满了的时候 put 方法会调用 await() 方法阻塞当前线程，如果其它线程中断了该线程，那么该线程会抛出 InterruptedException 异常，那么当前的日志任务就会被丢弃了。

如果 neverBlock 设置为了 true 则会调用阻塞队列的 offer 方法，而该方法是非阻塞的，如果当前队列满了，则会直接返回，也就是丢弃当前日志任务。

最后看下 addAppender 方法内做了啥：

**public** **void** **addAppender**(Appender<E> newAppender) {

**if** (appenderCount == 0) {

appenderCount++;

...

aai.addAppender(newAppender);

} **else** {

addWarn("One and only one appender may be attached to AsyncAppender.");

addWarn("Ignoring additional appender named [" + newAppender.getName() + "]");

}

}

如上代码可知一个异步 appender 只能绑定一个同步 appender, 这个 appender 会被放到 AppenderAttachableImpl 的 appenderList 列表里面。

到这里我们已经分析完了日志生产线程放入日志任务到日志队列的实现，下面一起来看下消费线程是如何从队列里面消费日志任务并写入磁盘的，由于消费线程是一个线程，那就从 worker 的 run 方法看起：

**class** **Worker** **extends** **Thread** {

**public** void run() {

AsyncAppenderBase<E> **parent** = AsyncAppenderBase.this;

AppenderAttachableImpl<E> aai = **parent**.aai;

//（10）一直循环直到该线程被中断

**while** (**parent**.isStarted()) {

**try** {//（11）从阻塞队列获取元素

E e = **parent**.blockingQueue.take();

aai.appendLoopOnAppenders(e);

} **catch** (InterruptedException ie) {

**break**;

}

}

//（12）到这里说明该线程被中断，则吧队列里面的剩余日志任务

//刷新到磁盘

**for** (E e : **parent**.blockingQueue) {

aai.appendLoopOnAppenders(e);

**parent**.blockingQueue.remove(e);

}

...

.. }

}

其中（11）从日志队列使用 take 方法获取一个日志任务，如果当前队列为空则当前线程会阻塞到 take 方法直到队列不为空才返回，获取到日志任务后会调用 AppenderAttachableImpl 的 aai.appendLoopOnAppenders 方法，该方法会循环调用通过 addAppender 注入的同步日志 appener 具体实现日志打印到磁盘的任务。

### 三、ConcurrentHashMap 的使用注意项

ConcurrentHashMap 虽然为并发安全的组件，但是使用不当还是会导致程序错误，本节通过使用简单的案例来复现这些问题并给出开发时候如何进行避免的策略

这里借用直播的一个场景，直播业务中，每个直播间对应一个 topic，每个用户进入直播间时候会把自己设备 id 绑定到这个 topic 上，也就是一个 topic 对应一堆用户设备，可知可以使用 map 来维护这些信息，key 为 topic，value 为设备的 list。下面通过代码模拟多用户同时进入直播间时候 map 信息的维护：

**public** **class** **TestMap** {

//(1)创建map,key为topic,value为设备列表

**static** ConcurrentHashMap<String, List<String>> map = **new** ConcurrentHashMap<>();

**public** **static** **void** **main**(String[] args) {

//(2)进入直播间topic1 线程one

Thread threadOne = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device1");

list1.**add**("device2");

map.put("topic1", list1);

System.**out**.println(JSON.toJSONString(map));

}

});

//(3)进入直播间topic1 线程two

Thread threadTwo = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device11");

list1.**add**("device22");

map.put("topic1", list1);

System.**out**.println(JSON.toJSONString(map));

}

});

//(4)进入直播间topic2 线程three

Thread threadThree = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device111");

list1.**add**("device222");

map.put("topic2", list1);

System.**out**.println(JSON.toJSONString(map));

}

});

//(5)启动线程

threadOne.start();

threadTwo.start();

threadThree.start();

}

}

* 代码（1）创建了一个并发 map 用来存放 topic 与其对应的设备列表
* 代码（2）（3）模拟用户进入直播间 topic1，代码（4）模拟用户进入直播间 topic2
* 代码（5）启动线程

运行代码输出：

{"topic1":["device11","device22"],"topic2":["device111","device222"]}

{"topic1":["device11","device22"],"topic2":["device111","device222"]}

{"topic1":["device11","device22"],"topic2":["device111","device222"]}

或者

{"topic1":["device1","device2"],"topic2":["device111","device222"]}

{"topic1":["device1","device2"],"topic2":["device111","device222"]}

{"topic1":["device1","device2"],"topic2":["device111","device222"]}

可知 topic1 房间中的用户会丢失一部分，这是因为 put 方法如果发现 map 里面存在这个 key, 则使用 value 覆盖该 key 对应的老的 value 值，而 putIfAbsent 方法则如果已经存在该 key 则返回该 key 对应的 value 并不进行覆盖，如果不存在则会新增该 key，并且判断和写入是原子性操作。使用 putIfAbsent 替代 put 方法后代码如下：

**public** **class** **TestMap2** {

//(1)创建map,key为topic,value为设备列表

**static** ConcurrentHashMap<String, List<String>> map = **new** ConcurrentHashMap<>();

**public** **static** **void** **main**(String[] args) {

//(2)进入直播间topic1 线程one

Thread threadOne = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device1");

list1.**add**("device2");

//(2.1)

List<String> oldList = map.putIfAbsent("topic1", list1);

**if**(null != oldList){

oldList.addAll(list1);

}

System.**out**.println(JSON.toJSONString(map));

}

});

//(3)进入直播间topic1 线程two

Thread threadTwo = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device11");

list1.**add**("device22");

List<String> oldList = map.putIfAbsent("topic1", list1);

**if**(null != oldList){

oldList.addAll(list1);

}

System.**out**.println(JSON.toJSONString(map));

}

});

//(4)进入直播间topic2 线程three

Thread threadThree = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

List<String> list1 = **new** ArrayList<>();

list1.**add**("device111");

list1.**add**("device222");

List<String> oldList = map.putIfAbsent("topic2", list1);

**if**(null != oldList){

oldList.addAll(list1);

}

System.**out**.println(JSON.toJSONString(map));

}

});

//(5)启动线程

threadOne.start();

threadTwo.start();

threadThree.start();

}

}

如上代码（2.1）使用 map.putIfAbsent 方法添加新设备列表，如果 topic1 在 map 中不存在则放入 topic1 和对应设备列表到 map，要注意的是这个判断不存在和放入是原子性操作，这时候放入后会返回 null。如果 topic1 已经在 map 里面存在，则调用 putIfAbsent 会返回 topic1 对应的设备里面，代码发现返回的设备列表不为 null 则把新的设备列表添加到返回的设备列表里面，从而问题得到解决。

运行结果为：

{"topic1":["device1","device2","device11","device22"],"topic2":["device111","device222"]}

{"topic1":["device1","device2","device11","device22"],"topic2":["device111","device222"]}

{"topic1":["device1","device2","device11","device22"],"topic2":["device111","device222"]}

总结：put(K key, V value) 方法如果 key 已经存在则使用 value 覆盖原来的值并返回原来的值，如果不存在则把 value 放入并返回 null。而 putIfAbsent(K key, V value) 方法如果 key 已经存在则直接返回原来对应的值并不使用 value 覆盖，如果 key 不存在则存入 value 并返回 null，另外要注意判断 key 不存在和存入是原子操作。

### 四、使用定时器 Timer 的时候需要注意的一些问题

#### 4.1 问题产生

这里做了一个小的 demo 来复现问题，代码如下：

**public** **class** **TestTimer** {

//创建定时器对象

**static** Timer timer = **new** Timer();

**public** **static** **void** **main**(String[] args) {

//添加任务1,延迟500ms执行

timer.schedule(**new** TimerTask() {

@Override

**public** **void** **run**() {

System.**out**.println("---one Task---");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**throw** **new** RuntimeException("error ");

}

}, 500);

//添加任务2，延迟1000ms执行

timer.schedule(**new** TimerTask() {

@Override

**public** **void** **run**() {

**for** (;;) {

System.**out**.println("---two Task---");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}, 1000);

}

}

如上代码先添加了一个任务在 500ms 后执行，然后添加了第二个任务在 1s 后执行，我们期望的是当第一个任务输出 ---one Task--- 后等待 1s 后第二个任务会输出 ---two Task---，但是执行完毕代码后输出结果为：

---one Task---

Exception **in** thread "Timer-0" java.lang.RuntimeException: error

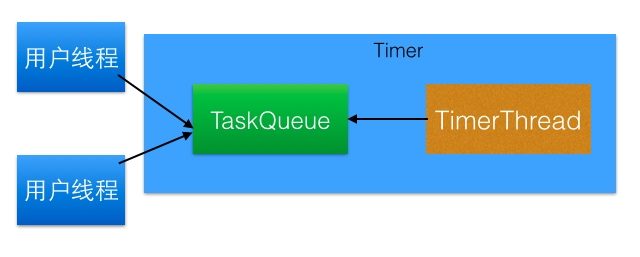
at com.zlx.Timer.TestTimer**$1**.run(TestTimer.java:22)

at java.util.TimerThread.mainLoop(Timer.java:555)

at java.util.TimerThread.run(Timer.java:505)

#### 4.2 Timer 实现原理分析

下面简单介绍下 Timer 的原理，如下图是 Timer 的原理模型介绍：



* 其中 TaskQueue 是一个平衡二叉树堆实现的优先级队列，每个 Timer 对象内部有唯一一个 TaskQueue 队列。用户线程调用 timer 的 schedule 方法就是把 TimerTask 任务添加到 TaskQueue 队列，在调用 schedule 的方法时候 long delay 参数用来说明该任务延迟多少时间执行。
* TimerThread 是具体执行任务的线程，它从 TaskQueue 队列里面获取优先级最小的任务进行执行，需要注意的是只有执行完了当前的任务才会从队列里面获取下一个任务而不管队列里面是否有已经到了设置的 delay 时间，一个 Timer 只有一个 TimerThread 线程，所以可知 Timer 的内部实现是一个多生产者单消费者模型。

从实现模型可以知道要探究上面的问题只需看 TimerThread 的实现就可以了，TimerThread 的 run 方法主要逻辑代码如下：

**public** **void** **run**() {

**try** {

mainLoop();

} finally {

// Someone killed this Thread, behave as if Timer cancelled

synchronized(**queue**) {

newTasksMayBeScheduled = false;

**queue**.clear(); // Eliminate obsolete references

}

}

}

**private** **void** **mainLoop**() {

**while** (**true**) {

**try** {

TimerTask task;

**boolean** taskFired;

//从队列里面获取任务时候要加锁

**synchronized**(queue) {

...

}

**if** (taskFired)

task.run();//执行任务

} **catch**(InterruptedException e) {

}

}

}

可知当任务执行过程中抛出了除 InterruptedException 之外的异常后，唯一的消费线程就会因为抛出异常而终止，那么队列里面的其他待执行的任务就会被清除。所以 TimerTask 的 run 方法内最好使用 try-catch 结构 catch 主可能的异常，不要把异常抛出到 run 方法外。其实要实现类似 Timer 的功能使用 ScheduledThreadPoolExecutor 的 schedule 是比较好的选择。ScheduledThreadPoolExecutor 中的一个任务抛出了异常，其他任务不受影响的。

**public** **class** **TestScheduledThreadPoolExecutor** {

**static** ScheduledThreadPoolExecutor scheduledThreadPoolExecutor = **new** ScheduledThreadPoolExecutor(1);

**public** **static** **void** **main**(String[] args) {

scheduledThreadPoolExecutor.schedule(**new** Runnable() {

@Override

**public** **void** **run**() {

System.**out**.println("---one Task---");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**throw** **new** RuntimeException("error ");

}

}, 500, TimeUnit.MICROSECONDS);

scheduledThreadPoolExecutor.schedule(**new** Runnable() {

@Override

**public** **void** **run**() {

**for** (**int** i =0;i<2;++i) {

System.**out**.println("---two Task---");

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

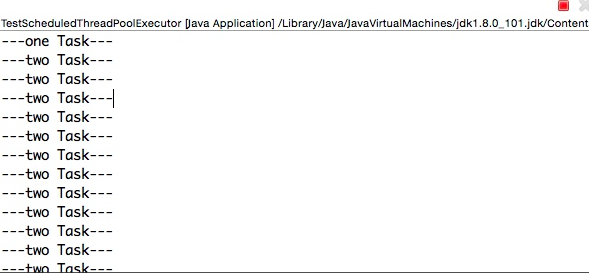
}, 1000, TimeUnit.MICROSECONDS);

scheduledThreadPoolExecutor.shutdown();

}

}

运行结果：

 之所以 ScheduledThreadPoolExecutor 的其他任务不受抛出异常的任务的影响是因为 ScheduledThreadPoolExecutor 中的 ScheduledFutureTask 任务中 catch 掉了异常，但是在线程池任务的 run 方法内使用 catch 捕获异常并打印日志是最佳实践。

### 五、SimpleDateFormat 是线程不安全的

SimpleDateFormat 是 Java 提供的一个格式化和解析日期的工具类，日常开发中应该经常会用到，但是由于它是线程不安全的，多线程公用一个 SimpleDateFormat 实例对日期进行解析或者格式化会导致程序出错，本节就讨论下它为何是线程不安全的，以及如何避免。

#### 5.1 问题复现

为了复现该问题，编写如下代码：

**public** **class** **TestSimpleDateFormat** {

//(1)创建单例实例

**static** SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

**public** **static** **void** **main**(String[] args) {

//(2)创建多个线程，并启动

**for** (**int** i = 0; i <10 ; ++i) {

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {//(3)使用单例日期实例解析文本

System.**out**.println(sdf.parse("2017-12-13 15:17:27"));

} **catch** (ParseException e) {

e.printStackTrace();

}

}

});

thread.start();//(4)启动线程

}

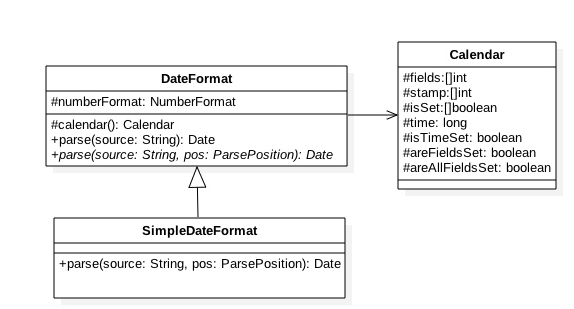
}

}

代码（1）创建了 SimpleDateFormat 的一个实例，代码（2）创建 10 个线程，每个线程都公用同一个 sdf 对象对文本日期进行解析，多运行几次就会抛出 java.lang.NumberFormatException 异常，加大线程的个数有利于该问题复现。

#### 5.2 问题分析

为了便于分析首先奉上 SimpleDateFormat 的类图结构：

 可知每个 SimpleDateFormat 实例里面有一个 Calendar 对象，从后面会知道其实 SimpleDateFormat 之所以是线程不安全的就是因为 Calendar 是线程不安全的，后者之所以是线程不安全的是因为其中存放日期数据的变量都是线程不安全的，比如里面的 fields，time 等。

下面从代码层面看下 parse 方法做了什么事情：

**public** Date **parse**(String text, ParsePosition pos)

{

//(1)解析日期字符串放入CalendarBuilder的实例calb中

.....

Date parsedDate;

**try** {//（2）使用calb中解析好的日期数据设置calendar

parsedDate = calb.establish(calendar).getTime();

...

}

**catch** (IllegalArgumentException e) {

...

**return** null;

}

**return** parsedDate;

}

Calendar **establish**(Calendar cal) {

...

//（3）重置日期对象cal的属性值

cal.clear();

//(4) 使用calb中中属性设置cal

...

//(5)返回设置好的cal对象

**return** cal;

}

* 代码（1）主要的作用是解析字符串日期并把解析好的数据放入了 CalendarBuilder 的实例 calb 中，CalendarBuilder 是一个建造者模式，用来存放后面需要的数据。
* 代码（3）重置 Calendar 对象里面的属性值，如下代码：

**public** **final** **void** **clear**()

{

**for** (**int** i = 0; i < fields.length; ) {

stamp[i] = fields[i] = 0; // UNSET == 0

isSet[i++] = **false**;

}

areAllFieldsSet = areFieldsSet = **false**;

isTimeSet = **false**;

}

* 代码（4）使用 calb 中解析好的日期数据设置 cal 对象
* 代码（5） 返回设置好的 cal 对象

从上面步骤可知步骤（3）（4）（5）操作不是原子性操作，当多个线程调用 parse 方法时候比如线程 A 执行了步骤（3）（4）也就是设置好了 cal 对象，在执行步骤（5）前线程 B 执行了步骤（3）清空了 cal 对象，由于多个线程使用的是一个 cal 对象，所以线程 A 执行步骤（5）返回的就可能是被线程 B 清空后的对象，当然也有可能线程 B 执行了步骤（4）被线程 B 修改后的 cal 对象。从而导致程序错误。

那么怎么解决那？

* 第一种方式：每次使用时候 new 一个 SimpleDateFormat 的实例，这样可以保证每个实例使用自己的 Calendar 实例, 但是每次使用都需要 new 一个对象，并且使用后由于没有其它引用，就会需要被回收，开销会很大。
* 第二种方式：究其原因是因为多线程下步骤（3）（4）（5）三个步骤不是一个原子性操作，那么容易想到的是对其进行同步，让（3）（4）（5）成为原子操作，可以使用 synchronized 进行同步，具体如下：

**public** **class** **TestSimpleDateFormat** {

// (1)创建单例实例

**static** SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

**public** **static** **void** **main**(String[] args) {

// (2)创建多个线程，并启动

**for** (**int** i = 0; i < 10; ++i) {

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {// (3)使用单例日期实例解析文本

synchronized (sdf) {

System.**out**.println(sdf.parse("2017-12-13 15:17:27"));

}

} **catch** (ParseException e) {

e.printStackTrace();

}

}

});

thread.start();// (4)启动线程

}

}

}

使用同步意味着多个线程要竞争锁，在高并发场景下会导致系统响应性能下降。

* 第三种方式：使用 ThreadLocal，这样每个线程只需要使用一个 SimpleDateFormat 实例相比第一种方式大大节省了对象的创建销毁开销，并且不需要对多个线程直接进行同步，使用 ThreadLocal 方式代码如下：

**public** **class** **TestSimpleDateFormat2** {

// (1)创建threadlocal实例

**static** ThreadLocal<DateFormat> safeSdf = **new** ThreadLocal<DateFormat>(){

@Override

**protected** SimpleDateFormat **initialValue**(){

**return** **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

}

};

**public** **static** **void** **main**(String[] args) {

// (2)创建多个线程，并启动

**for** (**int** i = 0; i < 10; ++i) {

Thread thread = **new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {// (3)使用单例日期实例解析文本

System.**out**.println(safeSdf.**get**().parse("2017-12-13 15:17:27"));

} **catch** (ParseException e) {

e.printStackTrace();

}**finally** {

//(4)使用完毕记得清除，避免内存泄露

safeSdf.**remove**();

}

}

});

thread.start();// (4)启动线程

}

}

}

代码（1）创建了一个线程安全的 SimpleDateFormat 实例，步骤（3）在使用的时候首先使用 get() 方法获取当前线程下 SimpleDateFormat 的实例，在第一次调用 ThreadLocal 的 get（）方法适合会触发其 initialValue 方法用来创建当前线程所需要的 SimpleDateFormat 对象。另外需要注意的是代码（4）使用完毕线程变量后要记得进行清理，以避免内存泄露。

### 六、线程池使用 FutureTask 时候需要注意的一点事

线程池使用 FutureTask 的时候如果拒绝策略设置为了 DiscardPolicy和DiscardOldestPolicy并且在被拒绝的任务的 Future 对象上调用无参 get 方法那么调用线程会一直被阻塞。

#### 6.1 问题复现

下面先通过一个简单的例子来复现问题：

**public** **class** **FutureTest** {

//(1)线程池单个线程，线程池队列元素个数为1

**private** final **static** ThreadPoolExecutor executorService = **new** ThreadPoolExecutor(1, 1, 1L, TimeUnit.MINUTES,

**new** ArrayBlockingQueue<Runnable>(1),**new** ThreadPoolExecutor.DiscardPolicy());

**public** **static** **void** **main**(String[] args) throws Exception {

//(2)添加任务one

Future futureOne = executorService.submit(**new** Runnable() {

@Override

**public** **void** **run**() {

System.**out**.println("start runable one");

**try** {

Thread.sleep(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

//(3)添加任务two

Future futureTwo = executorService.submit(**new** Runnable() {

@Override

**public** **void** **run**() {

System.**out**.println("start runable two");

}

});

//(4)添加任务three

Future futureThree=null;

**try** {

futureThree = executorService.submit(**new** Runnable() {

@Override

**public** **void** **run**() {

System.**out**.println("start runable three");

}

});

} **catch** (Exception e) {

System.**out**.println(e.getLocalizedMessage());

}

System.**out**.println("task one " + futureOne.**get**());//(5)等待任务one执行完毕

System.**out**.println("task two " + futureTwo.**get**());//(6)等待任务two执行完毕

System.**out**.println("task three " + (futureThree==null?null:futureThree.**get**()));// (7)等待任务three执行完毕

executorService.shutdown();//(8)关闭线程池，阻塞直到所有任务执行完毕

}

运行代码结果为：



代码 (1) 创建了一个单线程并且队列元素个数为 1 的线程池，并且拒绝策略设置为了DiscardPolicy

代码（2）向线程池提交了一个任务 one，那么这个任务会使用唯一的一个线程进行执行，任务在打印 start runable one后会阻塞该线程 5s.

代码（3）向线程池提交了一个任务 two，这时候会把任务 two 放入到阻塞队列

代码（4）向线程池提交任务 three，由于队列已经满了则会触发拒绝策略丢弃任务 three, 从执行结果看在任务 one 阻塞的 5s 内，主线程执行到了代码 (5) 等待任务 one 执行完毕，当任务 one 执行完毕后代码（5）返回，主线程打印出 task one null。任务 one 执行完成后线程池的唯一线程会去队列里面取出任务 two 并执行所以输出 start runable two 然后代码（6）会返回，这时候主线程输出 task two null，然后执行代码（7）等待任务 three 执行完毕，从执行结果看代码（7）会一直阻塞不会返回，至此问题产生，如果把拒绝策略修改为 DiscardOldestPolicy 也会存在有一个任务的 get 方法一直阻塞只是现在是任务 two 被阻塞。但是如果拒绝策略设置为默认的 AbortPolicy 则会正常返回，并且会输出如下结果：

**start** runable one

Task java.util.concurrent.FutureTask@135fbaa4 rejected **from** java.util.concurrent.ThreadPoolExecutor@45ee12a7[Running, pool **size** = 1, active threads = 1, queued tasks = 1, completed tasks = 0]

task one null**start** runable two

task two null

task three null

#### 6.2 问题分析

要分析这个问题需要看下线程池的 submit 方法里面做了什么，submit 方法代码如下：

**public** Future<?> submit(Runnable task) {

...

//（1）装饰Runnable为Future对象

RunnableFuture<Void> ftask = newTaskFor(task, null);

execute(ftask);

//(6)返回future对象

**return** ftask;

}

**protected** <T> RunnableFuture<T> **newTaskFor**(Runnable runnable, T **value**) {

**return** **new** FutureTask<T>(runnable, **value**);

}

**public** **void** **execute**(Runnable command) {

...

//(2) 如果线程个数小于核心线程数则新增处理线程处理

**int** c = ctl.**get**();

**if** (workerCountOf(c) < corePoolSize) {

**if** (addWorker(command, true))

**return**;

c = ctl.**get**();

}

//（3）如果当前线程个数已经达到核心线程数则任务放入队列

**if** (isRunning(c) && workQueue.offer(command)) {

**int** recheck = ctl.**get**();

**if** (! isRunning(recheck) && **remove**(command))

reject(command);

**else** **if** (workerCountOf(recheck) == 0)

addWorker(null, false);

}

//（4）尝试新增处理线程进行处理

**else** **if** (!addWorker(command, false))

reject(command);//(5)新增失败则调用拒绝策略

}

根据代码可以总结如下：

* 代码（1）装饰 Runnable 为 FutureTask 对象，然后调用线程池的 execute 方法
* 代码 (2) 如果线程个数小于核心线程数则新增处理线程处理
* 代码（3）如果当前线程个数已经达到核心线程数则任务放入队列
* 代码（4）尝试新增处理线程进行处理，失败则进行代码（5），否者直接使用新线程处理
* 代码（5）执行具体拒绝策略，从这里也可以看出拒绝策略执行是使用的业务线程。

所以要分析上面例子中问题所在只需要看步骤（5）对被拒绝任务的影响，这里先看下拒绝策略 DiscardPolicy 的代码：

**public** **static** **class** **DiscardPolicy** **implements** **RejectedExecutionHandler** {

**public** **DiscardPolicy**() { }

**public** **void** **rejectedExecution**(Runnable r, ThreadPoolExecutor e) {

}

}

可知拒绝策略 rejectedExecution 方法里面什么都没做，所以代码（4）调用 submit 后会返回一个 future 对象，这里有必要在重新说 future 是有状态的，future 的状态枚举值如下：

**private** **static** **final** **int** NEW = 0;

**private** **static** **final** **int** COMPLETING = 1;

**private** **static** **final** **int** NORMAL = 2;

**private** **static** **final** **int** EXCEPTIONAL = 3;

**private** **static** **final** **int** CANCELLED = 4;

**private** **static** **final** **int** INTERRUPTING = 5;

**private** **static** **final** **int** INTERRUPTED = 6;

在步骤（1）的时候使用 newTaskFor 方法转换 Runnable 任务为 FutureTask，而 FutureTask 的构造函数里面设置的状态就是 New。

**public** **FutureTask**(Runnable runnable, V result) {

**this**.callable = Executors.callable(runnable, result);

**this**.state = NEW; // ensure visibility of callable

}

所以使用 DiscardPolicy 策略提交后返回了一个状态为 NEW 的 future 对象。

那么我们下面就需要看下当调用 future 的无参 get 方法时候当 future 变为什么状态时候才会返回那, 那就需看下 FutureTask 的 get（）方法代码：

**public** V **get**() **throws** InterruptedException, ExecutionException {

**int** s = state;

//当状态值<=COMPLETING时候需要等待，否者调用report返回

**if** (s <= COMPLETING)

s = awaitDone(**false**, 0L);

**return** report(s);

}

**private** V **report**(**int** s) **throws** ExecutionException {

Object x = outcome;

//状态值为NORMAL正常返回

**if** (s == NORMAL)

**return** (V)x;

//状态值大于等于CANCELLED则抛异常

**if** (s >= CANCELLED)

**throw** **new** CancellationException();

**throw** **new** ExecutionException((Throwable)x);

}

也就是说当 future 的状态 > COMPLETING 时候调用 get 方法才会返回，而明显 DiscardPolicy 策略在拒绝元素的时候并没有设置该 future 的状态，后面也没有其他机会可以设置该 future 的状态，所以 future 的状态一直是 NEW，所以一直不会返回，同理 DiscardOldestPolicy 策略也是这样的问题，最老的任务被淘汰时候没有设置被淘汰任务对于 future 的状态。

那么默认的 AbortPolicy 策略为啥没问题那？其实 AbortPolicy 策略时候步骤（5）直接会抛出 RejectedExecutionException 异常，也就是 submit 方法并没有返回 future 对象，这时候 futureThree 是 null。

所以当使用 Future 的时候，尽量使用带超时时间的 get 方法，这样即使使用了 DiscardPolicy 拒绝策略也不至于一直等待，等待超时时间到了会自动返回的，如果非要使用不带参数的 get 方法则可以重写 DiscardPolicy 的拒绝策略在执行策略时候设置该 Future 的状态大于 COMPLETING 即可，但是查看 FutureTask 提供的方法发现只有 cancel 方法是 public 的并且可以设置 FutureTask 的状态大于 COMPLETING，重写拒绝策略具体代码可以如下：

**public** **class** **MyRejectedExecutionHandler** **implements** **RejectedExecutionHandler**{

@Override

**public** **void** **rejectedExecution**(Runnable runable, ThreadPoolExecutor e) {

**if** (!e.isShutdown()) {

**if**(**null** != runable && runable **instanceof** FutureTask){

((FutureTask) runable).cancel(**true**);

}

}

}

}

使用这个策略时候由于从 report 方法知道在 cancel 的任务上调用 get() 方法会抛出异常所以代码（7）需要使用 try-catch 捕获异常代码（7）修改为如下：

**try**{

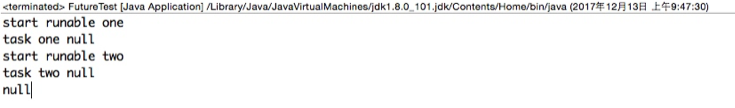
System.**out**.println("task three " + (futureThree==null?null:futureThree.**get**()));// (6)等待任务three

}**catch**(Exception e){

System.**out**.println(e.getLocalizedMessage());

}

执行结果为：



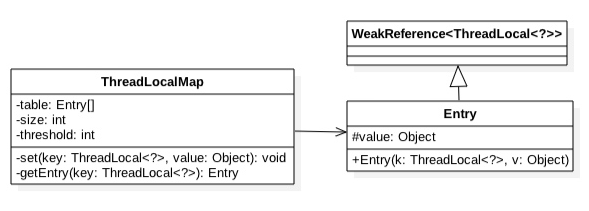
当然这相比正常情况下多了一个异常捕获，其实最好的情况是重写拒绝策略时候设置 FutureTask 的状态为 NORMAL，但是这需要重写 FutureTask 方法了，因为 FutureTask 并没有提供接口进行设置。

### 七、使用 ThreadLocal 不当可能会导致内存泄露

基础篇已经讲解了 ThreadLocal 的原理，本节着重来讲解下使用 ThreadLocal 会导致内存泄露的原因，并讲解使用 ThreadLocal 导致内存泄露的案例。

#### 7.1 为何会出现内存泄露

基础篇我们讲到了 ThreadLocal 只是一个工具类，具体存放变量的是在线程的 threadLocals 变量里面，threadLocals 是一个 ThreadLocalMap 类型的，



如上图 ThreadLocalMap 内部是一个 Entry 数组, Entry 继承自 WeakReference，Entry 内部的 value 用来存放通过 ThreadLocal 的 set 方法传递的值，那么 ThreadLocal 对象本身存放到哪里了吗? 下面看看 Entry 的构造函数：

Entry(ThreadLocal<?> k, Object v) {

super(k);

value = v;

}

**public** **WeakReference**(T referent) {

super(referent);

}

Reference(T referent) {

**this**(referent, null);

}

Reference(T referent, ReferenceQueue<? super T> **queue**) {

**this**.referent = referent;

**this**.**queue** = (**queue** == null) ? ReferenceQueue.NULL : **queue**;

}

可知 k 被传递到了 WeakReference 的构造函数里面，也就是说 ThreadLocalMap 里面的 key 为 ThreadLocal 对象的弱引用，具体是 referent 变量引用了 ThreadLocal 对象，value 为具体调用 ThreadLocal 的 set 方法传递的值。

当一个线程调用 ThreadLocal 的 set 方法设置变量时候，当前线程的 ThreadLocalMap 里面就会存放一个记录，这个记录的 key 为 ThreadLocal 的引用，value 则为设置的值。

但是考虑如果这个 ThreadLocal 变量没有了其他强依赖，而当前线程还存在的情况下，由于线程的 ThreadLocalMap 里面的 key 是弱依赖，则当前线程的 ThreadLocalMap 里面的 ThreadLocal 变量的弱引用会被在 gc 的时候回收，但是对应 value 还是会造成内存泄露，这时候 ThreadLocalMap 里面就会存在 key 为 null 但是 value 不为 null 的 entry 项。

其实在 ThreadLocal 的 set 和 get 和 remove 方法里面有一些时机是会对这些 key 为 null 的 entry 进行清理的，但是这些清理不是必须发生的，下面简单说下 ThreadLocalMap 的 remove 方法的清理过程：

**private** **void** **remove**(ThreadLocal<?> key) {

//(1)计算当前ThreadLocal变量所在table数组位置，尝试使用快速定位方法

Entry[] tab = table;

**int** len = tab.length;

**int** i = key.threadLocalHashCode & (len-1);

//(2)这里使用循环是防止快速定位失效后，变量table数组

**for** (Entry e = tab[i];

e != null;

e = tab[i = nextIndex(i, len)]) {

//(3)找到

**if** (e.**get**() == key) {

//(4)找到则调用WeakReference的clear方法清除对ThreadLocal的弱引用

e.clear();

//(5)清理key为null的元素

expungeStaleEntry(i);

**return**;

}

}

}

**private** **int** **expungeStaleEntry**(**int** staleSlot) {

Entry[] tab = table;

**int** len = tab.length;

//（6）去掉去value的引用

tab[staleSlot].**value** = null;

tab[staleSlot] = null;

size--;

Entry e;

**int** i;

**for** (i = nextIndex(staleSlot, len);

(e = tab[i]) != null;

i = nextIndex(i, len)) {

ThreadLocal<?> k = e.**get**();

//(7)如果key为null,则去掉对value的引用。

**if** (k == null) {

e.**value** = null;

tab[i] = null;

size--;

} **else** {

**int** h = k.threadLocalHashCode & (len - 1);

**if** (h != i) {

tab[i] = null;

**while** (tab[h] != null)

h = nextIndex(h, len);

tab[h] = e;

}

}

}

**return** i;

}

* 步骤（4）调用了 Entry 的 clear 方法，实际调用的是父类 WeakReference 的 clear 方法，作用是去掉对 ThreadLocal 的弱引用。
* 步骤（6）是去掉对 value 的引用，到这里当前线程里面的当前 ThreadLocal 对象的信息被清理完毕了。
* 代码（7）从当前元素的下标开始看 table 数组里面的其他元素是否有 key 为 null 的，有则清理。循环退出的条件是遇到 table 里面有 null 的元素。所以这里知道 null 元素后面的 Entry 里面 key 为 null 的元素不会被清理。

注：ThreadLocalMap 内部 Entry 中 key 使用的是对 ThreadLocal 对象的弱引用，这为避免内存泄露是一个进步，因为如果是强引用，那么即使其他地方没有对 ThreadLocal 对象的引用，ThreadLocalMap 中的 ThreadLocal 对象还是不会被回收，而如果是弱引用则这时候 ThreadLocal 引用是会被回收掉的。

但是对于的 value 还是不能被回收，这时候 ThreadLocalMap 里面就会存在 key 为 null 但是 value 不为 null 的 entry 项，虽然 ThreadLocalMap 提供了 set,get,remove 方法在一些时机下会对这些 Entry 项进行清理，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，所以在使用完毕后即使调用 remove 方法才是解决内存泄露的王道。

#### 7.2 线程池中使用 ThreadLocal 导致的内存泄露

下面先看线程池中使用 ThreadLocal 的例子：

**public** **class** **ThreadPoolTest** {

**static** **class** **LocalVariable** {

**private** Long[] a = **new** Long[1024\*1024];

}

// (1)

final **static** ThreadPoolExecutor poolExecutor = **new** ThreadPoolExecutor(5, 5, 1, TimeUnit.MINUTES,

**new** LinkedBlockingQueue<>());

// (2)

final **static** ThreadLocal<LocalVariable> localVariable = **new** ThreadLocal<LocalVariable>();

**public** **static** **void** **main**(String[] args) throws InterruptedException {

// (3)

**for** (**int** i = 0; i < 50; ++i) {

poolExecutor.execute(**new** Runnable() {

**public** **void** **run**() {

// (4)

localVariable.**set**(**new** LocalVariable());

// (5)

System.**out**.println("use local varaible");

//localVariable.remove();

}

});

Thread.sleep(1000);

}

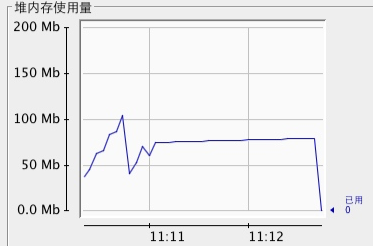
// (6)

System.**out**.println("pool execute over");

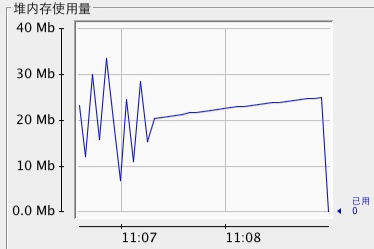
}

* 代码（1）创建了一个核心线程数和最大线程数为 5 的线程池，这个保证了线程池里面随时都有 5 个线程在运行。
* 代码（2）创建了一个 ThreadLocal 的变量，泛型参数为 LocalVariable，LocalVariable 内部是一个 Long 数组。
* 代码（3）向线程池里面放入 50 个任务
* 代码（4）设置当前线程的 localVariable 变量，也就是把 new 的 LocalVariable 变量放入当前线程的 threadLocals 变量。
* 由于没有调用线程池的 shutdown 或者 shutdownNow 方法所以线程池里面的用户线程不会退出，进而 JVM 进程也不会退出。

运行当前代码，使用 jconsole 监控堆内存变化如下图：



然后解开 localVariable.remove() 注释，然后在运行，观察堆内存变化如下:



从运行结果一可知，当主线程处于休眠时候进程占用了大概 77M 内存，运行结果二则占用了大概 25M 内存，可知运行代码一时候内存发生了泄露，下面分析下泄露的原因。

运行结果一的代码，在设置线程的 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，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。这里需要注意的是由于 localVariable 被声明了 static，虽然线程的 ThreadLocalMap 里面是对 localVariable 的弱引用，localVariable 也不会被回收。运行结果二的代码由于线程在设置 localVariable 变量后即使调用了localVariable.remove()方法进行了清理，所以不会存在内存泄露。

总结：线程池里面设置了 ThreadLocal 变量一定要记得及时清理，因为线程池里面的核心线程是一直存在的，如果不清理，那么线程池的核心线程的 threadLocals 变量一直会持有 ThreadLocal 变量。

### 八、总结

本文针对并发编程中经常遇到的一些问题进行讲解，希望读者能慢慢体会，以免在生产环境遇到坑，实际在并发编程中还会遇到其他的问题，限于篇幅本文没有提起，更多实战问题敬请期待《并发编程之美》一书的出版，这本书最后一章专门讲并发编程实战。