多线程

李聪

**修订历史**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版 本 | 章 节 | 类 型 | 日 期 | 作 者 | 说 明 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**说明：类型－创建（C）、修改（U）、删除（D）、增加（A）；**

目录

[文档名称4564 I](#_Toc415830246)

[1 标题1 1](#_Toc415830247)

[2 标题2 1](#_Toc415830248)

# 线程概述

进程指正在运行的程序，一个软件一旦开始运行就是一个进程。

线程是进程中的一个执行单元，一个进程中至少有一个线程，也是可以有多个线程的。

一个程序运行后至少有一个进程，一个进程中可以包含多个线程。

均时调度。

抢占式调度。根据优先级来分配cpu资源。

CPU(中央处理器)使用抢占式调度模式在多个线程间进行着高速的切换。

对于四核八线程的cpu来说，核是cpu的物理核心，线程是cpu的逻辑核心。

当一个cpu只有一个物理核心来说，同一时刻只能有一个线程运行。多个线程不停的切换。单核cpu在同一时间段内只能执行一个线程。

当使用的超线程技术，cpu的线程数要大于核心数。

# 多线程

jvm启动后，必然有一个执行路径(线程)从main方法开始的，一直执行到main方法结束，这个线程在java中称之为主线程（main线程）。

## 继承Thread类

### 构造方法

Thread()

Thread(String name): 给创建的线程命名

Thread(Runnable target): 把实现Runnable接口的线程任务作为参数传入线程对象

Thread(Runnable target, String name)

### 实现

**public** **class** ChildThread **extends** Thread {

**public** ChildThread(String name) {

**// 调用父类的String参数的构造方法，指定线程的名称**

**super**(name);

}

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

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

System.***out***.println(getName() + "：正在执行！" + i);

}

}

}

**public** **class** Demo01 {

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

**//创建自定义线程对象**

ChildThread mt = **new** ChildThread("子线程");

**//开启新线程**

mt.start();

**//在主方法中执行for循环**

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

System.***out***.println("main线程！"+i);

}

}

}

## 实现Runnable接口

### 实现

1、定义类实现Runnable接口。

2、覆盖接口中的run方法。。

3、创建Thread类的对象

4、将Runnable接口的子类对象作为参数传递给Thread类的构造函数。

5、调用Thread类的start方法开启线程。

public class Demo02 {

public static void main(String[] args) {

//创建线程执行目标类对象

Runnable runn = new MyRunnable();

//将Runnable接口的子类对象作为参数传递给Thread类的构造函数

Thread thread = new Thread(runn);

Thread thread2 = new Thread(runn);

//开启线程

thread.start();

thread2.start();

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

System.out.println("main线程：正在执行！"+i);

}

}

}

public class MyRunnable implements Runnable{

//定义线程要执行的run方法逻辑

@Override

public void run() {

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

System.out.println("我的线程：正在执行！"+i);

}

}

}

### 使用这种方式的好处

* 实现Runnable接口，避免了继承Thread类的单继承局限性，如果还要继承其它父类，就不能这样使用。
* Runnable接口对线程对象和线程任务进行解耦。实现Runnable接口，定义了线程任务，将Runnable接口的子类对象作为参数传入Thread的构造函数，创建了线程对象。更加的符合面向对象，线程分为两部分，一部分线程对象，一部分线程任务。

## 使用匿名内部类的形式

### Thread

**new** Thread(){

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

**for** (**int** i = 0; i < 3; i++) {

bussiness.subMethod();

}

}

}.start();

### Runnable

**new** Thread(**new** Runnable(){

@Override

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

**for** (**int** i = 0; i < 3; i++) {

bussiness.subMethod();

}

}

}).start();

Runnable r = **new** Runnable() {

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

**for** (**int** x = 0; x < 40; x++) {

System.*out*.println(Thread.*currentThread*().getName()

+ "...Y...." + x);

}

}

};

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

# 线程安全问题

如果有多个线程在同时运行，而这些线程可能会同时运行这段代码。程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

其实，线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

## 模拟卖票问题

**public** **class** Ticket **implements** Runnable{

**// 共100票**

**int** ticket = 100;

@Override

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

**// 模拟卖票**

**while** (**true**) {

**if** (ticket > 0) {

**// 模拟选坐的操作**

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(Thread.*currentThread*().getName() + "正在卖票:" + ticket--);

}

}

}

}

**public** **class** TestTicket {

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

**// 创建票对象**

Ticket ticket = **new** Ticket();

**// 创建3个窗口**

Thread t1 = **new** Thread(ticket, "窗口1");

Thread t2 = **new** Thread(ticket, "窗口2");

Thread t3 = **new** Thread(ticket, "窗口3");

t1.start();

t2.start();

t3.start();

}

}

会出现重复的，0，还是负数。正常是从100到1。

当一个线程正在对全局变量执行写操作的时候，这个线程停了，cpu资源被另一个线程抢占了，第一个线程要把ticke从1变成0，还没有变，另一个线程过来了。

## 线程同步

### 同步方法

public synchronized void method(){

可能会产生线程安全问题的代码

}

public class Ticket implements Runnable {

//共100票

int ticket = 100;

@Override

public void run() {

//模拟卖票

while(true){

//同步方法

method();

}

}

//同步方法,锁对象this

**public** **synchronized** **void** method(){

**if** (ticket > 0) {

//模拟选坐的操作

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() + "正在卖票:" + ticket--);

}

}

}

关键字synchronized可以保证在同一时刻，只有一个线程可以执行某个方法或某个代码块，同时synchronized可以保证一个线程的变化可见（可见性），即可以代替volatile。

可以确保线程互斥的访问同步代码。

Java中任何对象都可以作为锁，这是synchronized实现同步的基础：

* 普通同步方法（实例方法），锁是当前实例对象 ，进入同步代码前要获得当前实例的锁。
* 静态同步方法，锁是当前类的class字节码对象 ，进入同步代码前要获得当前字节码对象的锁。

字节码对象在内存中是唯一的，这个没有问题，但是实例对象不一定是唯一的。

**注意：**如果是在一个类中使用了多个同步方法，那么锁都是同一个当前类的实例对象。这个不对的，得看调用方法时，是不是同一个实例对象

public class Demo{

public synchronized void run() {

System.out.println(1);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(2);

}

}

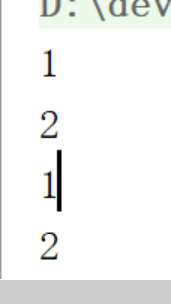
使用同一实例对象去访问

Demo demo = new Demo();

new Thread(() -> demo.run()).start();

new Thread(() -> demo.run()).start();

那么结果是同步的，因为用的是一个实例对象，是同一把锁。一个线程对同步方法没有执行完毕之前，另一个线程是执行不了的。

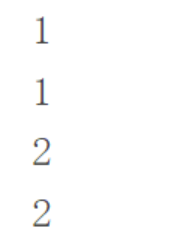


Demo demo = new Demo();

new Thread(() -> demo.run()).start();

Demo demo2 = new Demo();

new Thread(() -> demo2.run()).start();



结果可能不会同步。因为synchronized修饰普通方法时锁对象是this对象，而使用两个对象去访问，不是同一把锁。

线程访问同步方法的过程：进入同步代码前要获得当前实例的锁，哪个对象调用的，锁就是谁。当锁没有释放的时候，其它线程不会执行同步代码，当前同步代码执行完毕，释放锁，其它线程才可以获得锁，执行同步代码。并不是当前线程执行完毕之后才会释放锁。

#### 多个线程访问同一个对象的同一个方法

**public** **class** SynchronizedTest **implements** Runnable {

**// 共享资源**

**static** **int** *i* = 0;

/\*\*

\* synchronized 修饰实例方法

\*/

**public** **synchronized** **void** increase() {

*i*++;

}

@Override

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

**for** (**int** j = 0; j < 5; j++) {

System.***out***.println(Thread.*currentThread*().getName()+"===="+*i*);

increase();

}

}

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

SynchronizedTest test = **new** SynchronizedTest();

Thread t1 = **new** Thread(test);

Thread t2 = **new** Thread(test);

t1.start();

t1.join();//等待t1结束，这时候t2线程并未启动

t2.start();

**t2.join();**//等待t2结束，才执行主线程中的最后一行代码

System.***out***.println(*i*);

}

}

test就是实例对象

当两个线程同时对一个对象的一个方法进行操作，只有一个线程能够抢到锁。因为一个对象只有一把锁，一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，就不能访问该对象的其他synchronized实例方法，但是可以访问非synchronized修饰的方法。

join方法的作用是，让代码顺序执行，如果没有两个join方法，结果是0。因为线程只是启动了但是并没有执行完，就直接执行最后一行代码了，所以就是0.

**对上面代码的分析：**

Thread t1 = **new** Thread(test);

Thread t2 = **new** Thread(test);

t1.start();

t2.start();

t1.join();//等待t1结束，这时候t2线程并未启动

**//t2.join();**//等待t2结束，才执行主线程中的最后一行代码

System.***out***.println(*i*);

一个线程执行同步代码之前会获得锁，执行完毕之后释放锁。

Thread-0====0

Thread-0====1

Thread-0====2

Thread-0====3

Thread-0====4

Thread-1====0

5

Thread-1====6

Thread-1====7

Thread-1====8

Thread-1====9

锁是increase方法上面的，但是run方法内部的代码不是同步代码，没有锁，两个线程可以同时访问，两个线程谁先开始是随机的，可能是第二个线程先抢到cpu资源，这个时候i是0，但还没有进行increase操作，cpu就又把资源分配给了第一个线程，由于有join的存在，它会执行完毕之后，才轮到第二个线程，第二个线程又开始执行，由于刚才刚抢到的时候i还是0，就把0给打印出来，但是其实这个时候i已经变成了5，开始执行increase方法，i变成6，执行了一次cpu资源又被主线程抢到了，就先把最后一行代码打印了出来，第二个线程继续执行，打印6789.

#### 一个线程获取了该对象的锁之后，其他线程来访问其他synchronized实例方法

**public** **synchronized** **void** method1() {

System.***out***.println("Method 1 start");

**try** {

System.***out***.println("Method 1 execute");

Thread.*sleep*(3000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 1 end");

}

**public** **synchronized** **void** method2() {

System.***out***.println("Method 2 start");

**try** {

System.***out***.println("Method 2 execute");

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 2 end");

}

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

**final** syncTest test = **new** syncTest();

**new** Thread(**new** Runnable() {

@Override

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

test.method1();

}

}).start();

**new** Thread(**new** Runnable() {

@Override

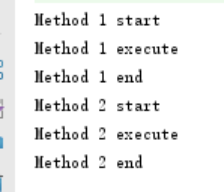
**public** **void** run() {

test.method2();

}

}).start();

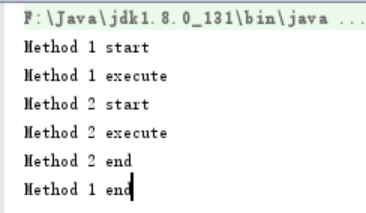
}



由于在一个类中的多个同步方法使用的锁都是同一个对象，当一个线程访问的同步方法没有结束之前，也就是没有释放锁，另外一个线程是访问不了其它同步方法的。

#### 一个线程获取了该对象的锁之后，其他线程来访问其他非synchronized实例方法

是可以的，把方法2中的synchronized去掉



#### 静态同步方法

public static synchronized void run() {

System.out.println(1);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(2);

}

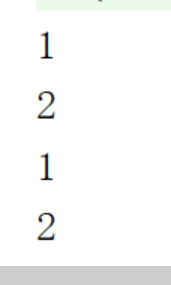
使用不同的对象访问

Demo demo = new Demo();

new Thread(() -> demo.run()).start();

Demo demo2 = new Demo();

new Thread(() -> demo2.run()).start();



可以看到结果是同步的，因为当修饰静态方法时，锁对象是class字节码文件对象，而两个对象是同一个class文件，所以使用的是一个锁。

### 同步代码块

将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次当线程进入synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待。

## Lock接口

Lock提供了一个更加面对对象的锁，在该锁中提供了更多的操作锁的功能。

public class Ticket implements Runnable {

//共100票

int ticket = 100;

//创建Lock锁对象

Lock ck = new ReentrantLock();

@Override

public void run() {

//模拟卖票

while(true){

//synchronized (lock){

ck.lock();

if (ticket > 0) {

//模拟选坐的操作

try {

Thread.*sleep*(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() + "正在卖票:" + ticket--);

}

ck.unlock();

//}

}

}

}

可以控制释放锁的时机。

## 线程间的制约关系

### 同步与互斥

1. 间接相互制约。一个系统中的多个线程必然要共享某种系统资源，如共享 CPU，共享 I/O 设备，所谓间接相互制约即源于这种资源共享，打印机就是最好的例子，线程 A 在使用打印机时，其它线程都要等待。

2. 直接相互制约。这种制约主要是因为线程之间的合作，如有线程 A 将计算结果提供给线程 B 作进一步处理，那么线程 B 在线程 A 将数据送达之前都将处于阻塞状态。

间接相互制约可以称为互斥，直接相互制约可以称为同步，对于互斥可以这样理解，线程 A 和线程 B 互斥访问某个资源则它们之间就会产个顺序问题——要么线程 A 等待线程 B 操作完毕，要么线程 B 等待线程操作完毕，这其实就是线程的同步了。因此同步包括互斥，互斥其实是一种特殊的同步。

### 代码实例

**public** **class** MainThread {

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

**final** Bussiness bussiness = **new** Bussiness();

**// 子线程**

**new** Thread(**new** Runnable(){

@Override

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

**for** (**int** i = 0; i < 3; i++) {

bussiness.subMethod();

}

}

}).start();

**// 主线程**

**for**(**int** i = 0;i<3;i++) {

bussiness.mainMethod();

}

}

}

**public** **class** Bussiness {

**private** **boolean** subFlag = **true**;

**//同步方法**

**public** **synchronized** **void** mainMethod() {

**while** (subFlag) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println(Thread.*currentThread*().getName() + " : main thread running loop count -- " + i);

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

subFlag = **true**;

notify();

}

**public** **synchronized** **void** subMethod() {

**while** (!subFlag) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

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

System.***err***.println(Thread.*currentThread*().getName() + " : sub thread running loop count -- " + i);

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

subFlag = **false**;

notify();

}

}

分析：

在main方法里面，首先创建了一个子线程，但是没有join方法的存在，主线程不管它继续往下走，首先走了mainMethod这个方法，此时subflag为true，有wait变成阻塞状态，释放同一个bussiness实例对象锁，这个时候cpu资源分配给了子线程，子线程获得bussiness实例对象锁，开始执行subMethod，此时subflag为true，不走wait，子线程打印了10遍，通过notify方法，唤醒主线程，notify唤醒的是**其所在锁所阻塞的线程。**

整体来说，就是一个阻塞，释放锁，另一个执行，执行完之后，唤醒另一个线程，这样交替执行。

## 其它总结

### synchronized释放锁的时机

任何线程进入同步代码块、同步方法之前，必须获得同步监视器的锁定，那么何时会释放这个锁定呢？在程序中，是无法显式释放对同步监视器的锁的，而会在**如下几个情况下释放锁：**

1、当前线程的同步方法、代码块执行结束的时候释放

2、当前线程在同步方法、同步代码块中遇到break 、 return 终于该代码块或者方法的时候释放。

3、出现未处理的error或者exception导致异常结束的时候释放

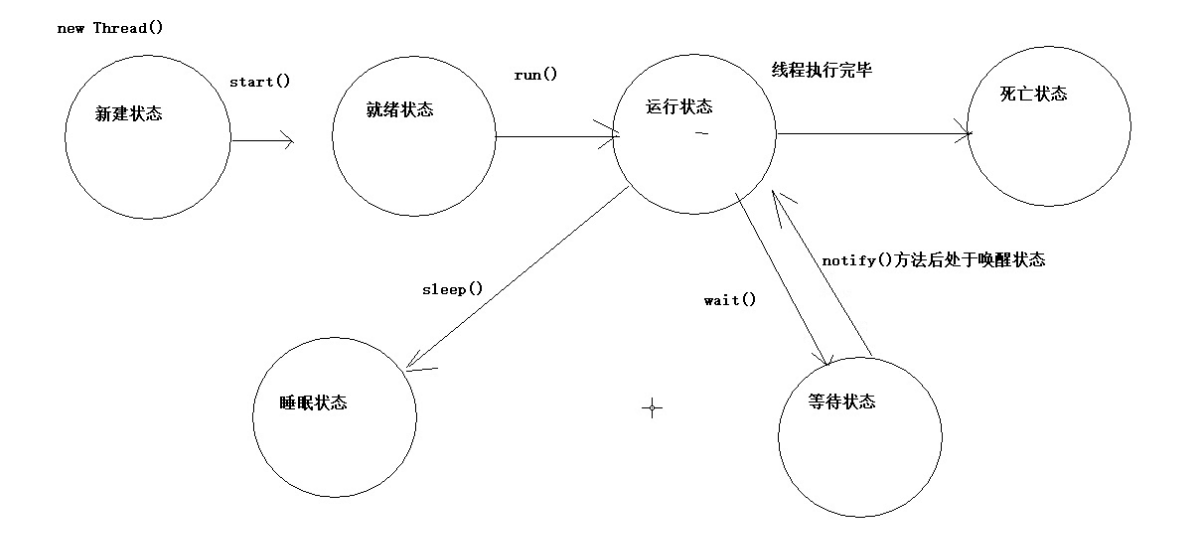
4、程序执行了同步对象wait方法，当前线程暂停，释放锁

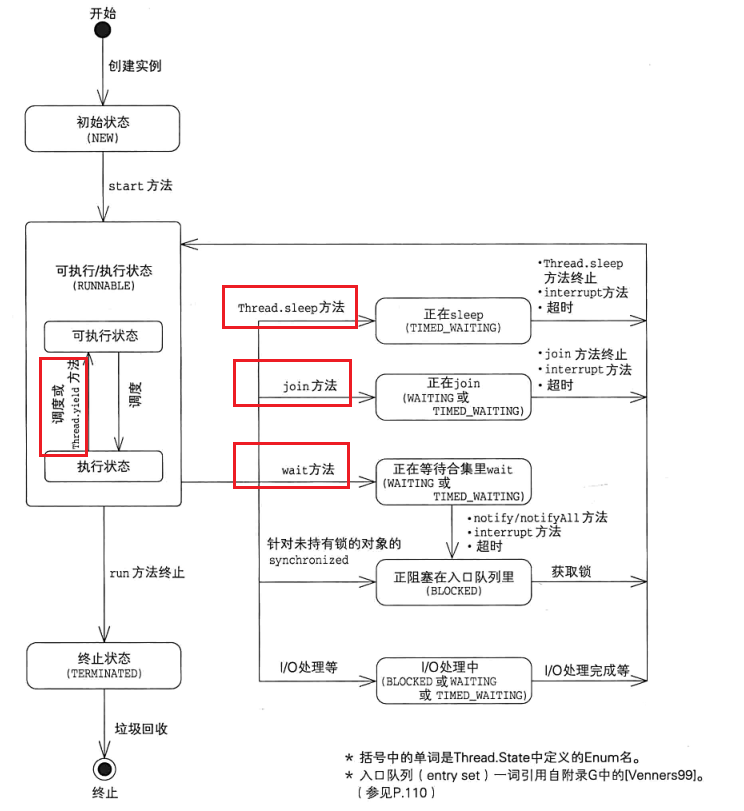
**如下情况不会释放锁**：

1、程序调用 Thread.sleep() Thread.yield() 这些方法暂停线程的执行，不会释放。

2、线程执行同步代码块时，其他线程调用 suspend 方法将该线程挂起，该线程不会释放锁 ，所以我们应该避免使用 suspend 和 resume 来控制线程

### 线程的生命周期





### notify与join方法

Java多线程开发中，我们常用到wait()和notify()方法来实现线程间的协作，简单的说步骤如下：

1. A线程取得锁，执行wait()，释放锁;

2. B线程取得锁，完成业务后执行notify()，再释放锁;

3. B线程释放锁之后，A线程取得锁，继续执行wait()之后的代码；

一个线程start了之前，处于就绪状态，使用join方法，后面的代码就会先等着，这个线程执行完毕之后，才会往下继续执行。

线程自闭，**join()**方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

分析：

在主线程中调用创建的子线程的join方法，则主线程转入阻塞状态，直到子线程运行结束，主线程再由阻塞状态转为就绪状态，继续运行。

### Java中实现线程阻塞的方法

1）线程睡眠：Thread.sleep (long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。sleep不释放锁。

（2）线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait() 一样。wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

（3）线程礼让，Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程.

（4）线程自闭，join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

（5）suspend() 和 resume() 方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

# 线程局部变量ThreadLocal

## 作用跟目的

用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。

## 使用方法

每个线程调用全局ThreadLocal对象的set方法，在set方法中，首先根据当前线程获取当前线程的ThreadLocalMap对象，然后往这个map中插入一条记录， key 其实是 ThreadLocal 对象， value 是各自的 set方法传进去的值。也就是每个线程其实都有一份自己独享的ThreadLocalMap对象，该对象的Key是ThreadLocal

对象，值是用户设置的具体值。在线程结束时可以调用 ThreadLocal.remove()方法，这样会更快释放内存，不调用也可以，因为线程结束后也可以自动释放相关的ThreadLocal变量。

这个map只能存放一对key-value。

应用场景：

* 订单处理包含一系列操作：减少库存量、增加一条流水台账、修改总账，这几个操作要在同一个事务中完成，通常也即同一个线程中进行处理，如果累加公司应收款的操作失败了，则应该把前面的操作回滚，否则，提交所有操作，这要求这些操作使用相同的数据库连接对象，而这些操作的代码分别位于不同的模块类中。
* 银行转账包含一系列操作： 把转出帐户的余额减少，把转入帐户的余额增加，这两个操作要在同一个事务中完成，它们必须使用相同的数据库连接对象，转入和转出操作的代码分别是两个不同的帐户对象的方法。

## 使用ThreadLocal操作数据库

分析背景: 银行转账

当操作数据库的时候，事务只能在一个连接里面。事务的开启和提交（回滚）必须在一个连接中进行，不能跨连接。

当线程1执行out方法的时候，cpu资源被线程2抢走了，连接对象也被抢走了，当线程1再执行in方法的时候，又从连接池里面获取了一个连接，线程1在执行out 和 in两个方法时，用了两个连接对象，这样的话我们没有办法保证事务是一致的。

一个连接对象同时只能被一个线程使用。

ThreadLocal可以保证一个线程在执行过程（三部分操作web，service，dao）TreadLocal存储的连接对象，在整个线程都可以使用，线程结束这个连接对象该怎么样就怎么样，其它线程是取不出来的。

//连接池

private static ComboPooledDataSource dataSource = new ComboPooledDataSource("itcast");

//给当前线程绑定 连接

private static ThreadLocal<Connection> local = new ThreadLocal<Connection>();

/\*\*

\* 获得连接

\* @return

\*/

public static Connection getConnection(){

try {

//#1从当前线程中， 获得已经绑定的连接

Connection conn = local.get();

if(conn == null){

//#2 第一次获得，绑定内容 – 从连接池获得

conn = dataSource.getConnection();

//#3 将连接存 ThreadLocal

local.set(conn);

}

return conn; //获得连接

} catch (Exception e) {

//将编译时异常 转换 运行时 ， 以后开发中 运行时异常使用比较多的。

// \* 此处可以编写自定义异常。

throw new RuntimeException(e);

// \* 类与类之间 进行数据交换时，可以使用return返回值。也可以自定义异常返回值，调用者try{} catch(e){ e.getMessage() 获得需要的数据}

//throw new MyConnectionException(e);

}

}

# 多线程并发库java.util.concurrent

java.util.concurrent.atomic 包 (多线程的原子性操作提供的工具类)

java.util.concurrent.locks包 (多线程的锁机制)

## concurrent的组件

* 执行程序（线程池）
* 并发队列
* 同步器
* 并发Collocation

### 执行程序

#### Executors线程池工厂类

##### 线程池的作用

线程池作用就是限制系统中执行线程的数量。

根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。

用线程池控制线程数量，其他线程排队等候。

一个任务执行完毕，再从队列（应该就是线程池中的多个线程）中取最前面的任务开始执行。若队列中没有等待线程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为因为消耗过多的内存，而使服务器死机。(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)

##### Executor

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。 ThreadPoolExecutor 是 Executors 类的底层实现。

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多（未死亡）的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

##### 线程池的种类

//创建固定大小的线程池  
ExecutorService fPool = Executors.newFixedThreadPool(3);  
//创建缓存大小的线程池  
ExecutorService cPool = Executors.newCachedThreadPool();  
//创建单一的线程池  
ExecutorService sPool = Executors.newSingleThreadExecutor();

* 固定大小线程池

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.ExecutorService;

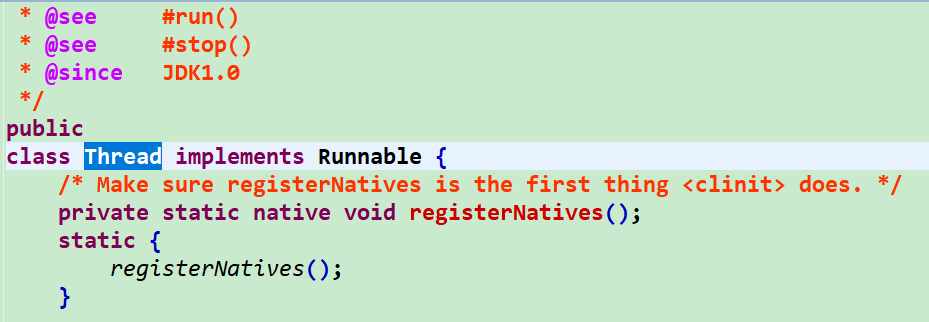
**public** **class** Test {

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

**// 创建一个可重用固定线程数的线程池**

ExecutorService pool = Executors.*newFixedThreadPool*(2);

**// 创建实现了 Runnable 接口对象， Thread 对象当然也实现了 Runnable 接口**

 Thread t1 = **new** MyThread();

Thread t2 = **new** MyThread();

Thread t3 = **new** MyThread();

Thread t4 = **new** MyThread();

Thread t5 = **new** MyThread();

**// 将线程放入池中进行执行**

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

pool.execute(t5);

**// 关闭线程池**

pool.shutdown();

}

}

**class** MyThread **extends** Thread {

@Override

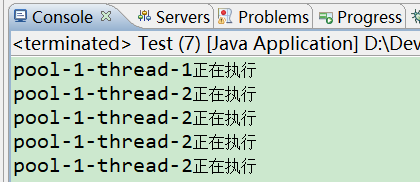
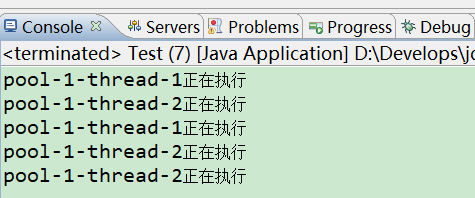
**public** **void** run() {

System.***out***.println(Thread.*currentThread*().getName() + "正在执行");

}

}

执行结果：

从上面的运行来看，我们 Thread 类都是在线程池中运行的，线程池在执行 execute方法来执行 Thread 类中的 run 方法。不管 execute 执行几次，线程池始终都会使用 2 个线程来处理。不会再去创建出其他线程来处理run 方法执行。这就是固定大小线程池。

* 单线程线程池

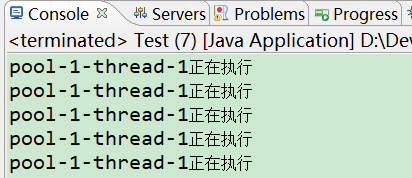
**// // 创建一个可重用固定线程数的线程池**

**// ExecutorService pool = Executors.newFixedThreadPool(2);**

**//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。**

ExecutorService pool = Executors.*newSingleThreadExecutor*();

执行结果：



单任务线程池在使用 execute 方法来执行 Thread 类中的 run 方法。不管 execute 执行几次，线程池始终都会使用单个线程来处理。

**在 java 的多线程中，一但线程关闭，就会成为死线程。关闭后死线程就没有办法在启动了。再次启动就会出现异常信息： Exception in thread "main" java.lang.IllegalThreadStateException。那么如何解决这个问题呢？  
我们这里就可以使用 Executors.newSingleThreadExecutor()来再次启动一个线程。（面试）**

* 可变大小的线程池

**// // 创建一个可重用固定线程数的线程池**

**// ExecutorService pool = Executors.newFixedThreadPool(2);**

**// //创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。**

**// ExecutorService pool = Executors.newSingleThreadExecutor();**

**//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。**

ExecutorService pool = Executors.*newCachedThreadPool*();

**// 创建实现了 Runnable 接口对象， Thread 对象当然也实现了 Runnable 接口**

Thread t1 = **new** MyThread();

Thread t2 = **new** MyThread();

Thread t3 = **new** MyThread();

Thread t4 = **new** MyThread();

Thread t5 = **new** MyThread();

Thread t6 = **new** MyThread();

Thread t7 = **new** MyThread();

Thread t8 = **new** MyThread();

**// 将线程放入池中进行执行**

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

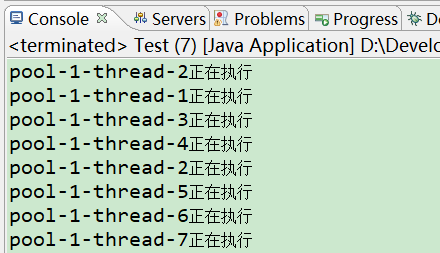
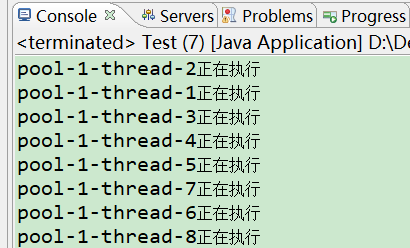
pool.execute(t5);

pool.execute(t6);

pool.execute(t7);

pool.execute(t8);

执行结果：

如果有8个任务需要执行的话，可能线程池分别去创建8个线程去执行，也有可能，前面的线程执行完任务之后，再去执行其它任务。

* 延迟线程池

**public** **class** Test {

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

**// // 创建一个可重用固定线程数的线程池**

**// ExecutorService pool = Executors.newFixedThreadPool(2);**

**// //创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。**

**// ExecutorService pool = Executors.newSingleThreadExecutor();**

**// //创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。**

**// ExecutorService pool = Executors.newCachedThreadPool();**

**//创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。**

ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(2);

**// 创建实现了 Runnable 接口对象， Thread 对象当然也实现了 Runnable 接口**

Thread t1 = **new** MyThread();

Thread t2 = **new** MyThread();

Thread t3 = **new** MyThread();

Thread t4 = **new** MyThread();

Thread t5 = **new** MyThread();

**// 将线程放入池中进行执行**

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

**// pool.execute(t4);**

**// pool.execute(t5);**

**//使用定时执行风格的方法**

pool.schedule(t4, 10, TimeUnit.***MILLISECONDS***); **//t4 和 t5 在 10 秒后执行**

pool.schedule(t5, 10, TimeUnit.***MILLISECONDS***);

**// 关闭线程池**

pool.shutdown();

}

}

**class** MyThread **extends** Thread {

@Override

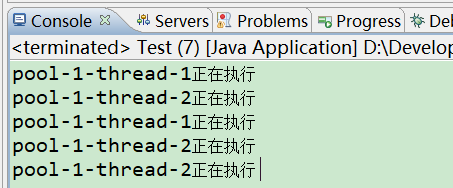
**public** **void** run() {

System.***out***.println(Thread.*currentThread*().getName() + "正在执行 ");

}

}

执行结果：

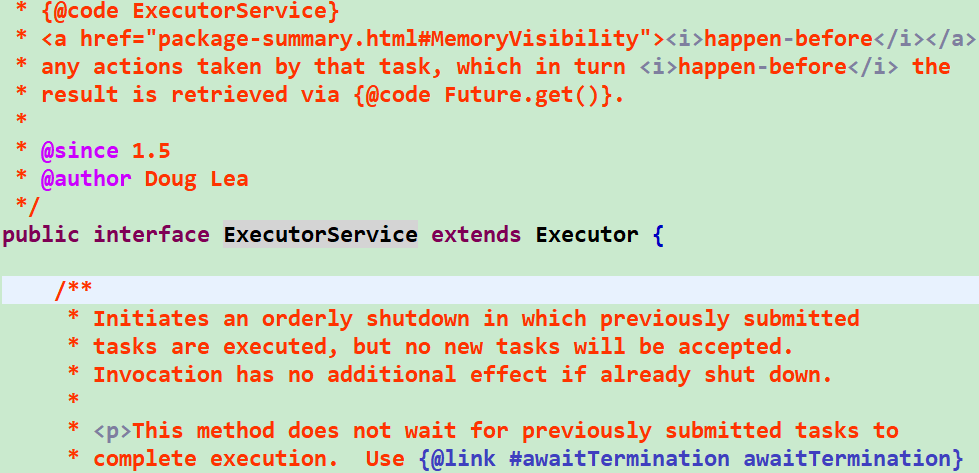


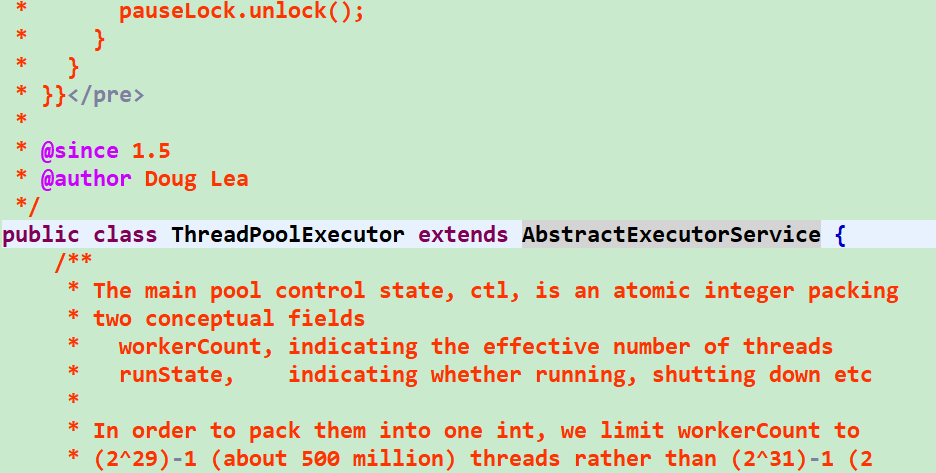
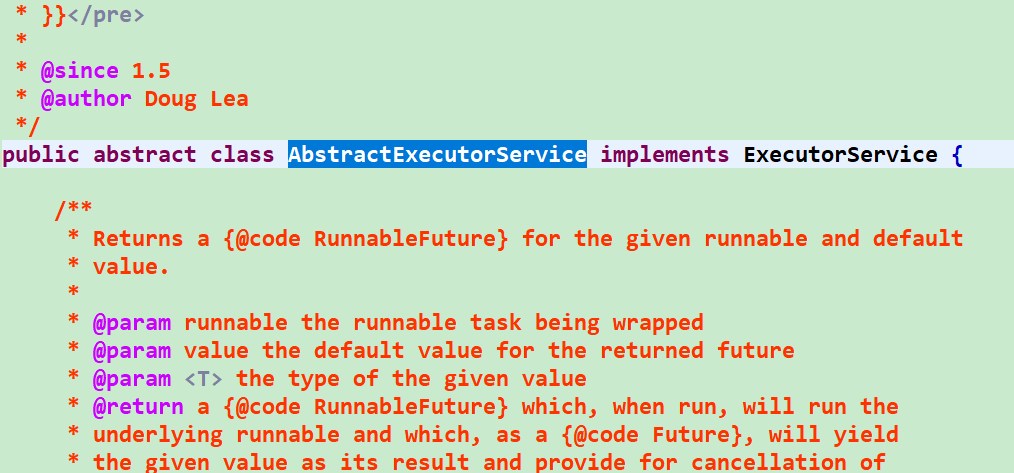
#### ExecutorService执行器服务

##### 分析

Executors不是真正的线程池顶层接口，ExecutorService才是。

java.util.concurrent.ExecutorService接口表示一个**异步执行**机制。ExecutorService类似于一个线程池，ExecutorService的实现就是一个线程池实现。





**// 线程工厂类创建出线程池**

ExecutorService executorService = Executors.*newFixedThreadPool*(10);

**// 执行一个线程任务**

executorService.execute(**new** Runnable() {

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

System.***out***.println("Asynchronous task");

}

});

**// 线程池关闭**

executorService.shutdown();

System.***out***.println("主线程执行完毕");

**分析：**

上面代码首先使用*newFixedThreadPool*(10)工厂方法创建一个 ExecutorService。这里创建了一个十个线程执行任务的线程池。然后，将一个 Runnable 接口的匿名实现类传递给 execute ()方法。这将导致 ExecutorService中的某个线程执行该 Runnable。

这里可以看成一个**任务分派**，示例代码中的任务分派我们可以理解为：

* 一个线程将一个任务委派给一个 ExecutorService去异步执行。
* 一旦该线程将任务委派给 ExecutorService，该线程将继续它自己的执行，独立于该任务的执行。

意思就是，主线程把一个线程任务分配给了ExecutorService，不用等ExecutorService执行完毕，主线程继续执行自己的代码。

同步：按顺序来，像avticeMQ一样，一个人发消息，另一个接收到，才算做完

异步：我发了消息就不管了，可以继续发。

##### ExecutorService实现

java.util.concurrent 包提供了 ExecutorService 接口的以下实现类：

* ThreadPoolExecutor
* ScheduledThreadPoolExecutor

##### ExecutorService创建

可以使用Executor工厂类来创建ExecutorService实例：

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

也可以使用：

中科软pcm系统中对ThreadPoolExecutor使用：

**public** **static** ThreadPoolExecutor *threadPool*=

**new** ThreadPoolExecutor(

PbocConsts.*QUERYTHREADNUM*, //5

PbocConsts.*QUERYTHREADNUM*, //5

60000,

TimeUnit.***SECONDS***,

**new** ArrayBlockingQueue<Runnable>(2000),

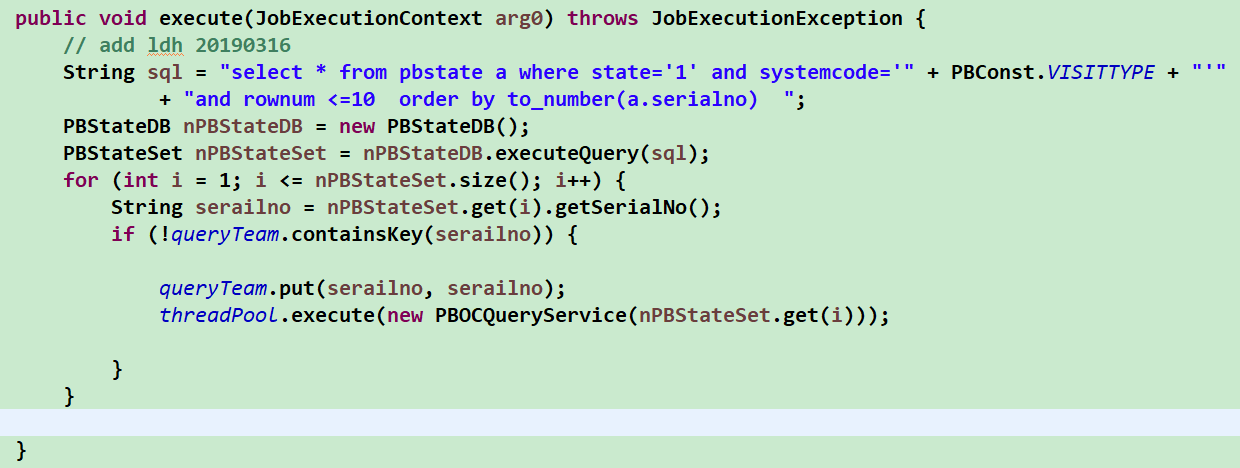
**new** ThreadPoolExecutor.CallerRunsPolicy()

);

直接创建了一个ThreadPoolExecutor线程池，规定了线程池的大小为5，

**public** **static** ConcurrentHashMap<String,String> *queryTeam*=**new** ConcurrentHashMap<String,String>();

还创建了一个任务队列



我之前在工作中对多线程的使用，没有太复杂，仅仅是为了提高系统对征信报告的查询效率，而且是在我们那个lis框架中，没有涉及到线程安全什么的。

我们的那个查询job中使用了多线程的方式，首先是提交的查询申请记录保存在pbstate表里面，状态为待查询，使用定时任务的方式，从数据库中查询出待查询状态的数据，直接创建了一个线程池ThreadPoolExecutor，分配了线程的数量，还有其它一些参数，同时还创建了一个任务队列，把需要查询的任务放入到任务队列里面，使用线程池execute，查询完毕之后，从任务队列中移除。

##### ExecutorService使用

有有几种不同的方式来将任务委托给 ExecutorService 去执行：

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(…)
* invokeAll(…)
* execute(Runnable)

execute(Runnable) 方法要求一个 java.lang.Runnable 对象，然后对它进行异步执行。

**// 从 Executors 中获得 ExecutorService**

ExecutorService executorService = Executors.*newSingleThreadExecutor*();

**// 执行 ExecutorService 中的方法**

executorService.execute(**new** Runnable() {

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

System.***out***.println("Asynchronous task");

}

});

**// 线程池关闭**

executorService.shutdown();

特点：没有办法得知Runnable的执行结果

* submit(Runnable)

submit(Runnable) 方法也要求一个Runnable实现类，但它返回一个Future对象。这个Future对象可以用来检查 Runnable 是否已经执行完毕。

**// 从 Executors 中获得 ExecutorService**

ExecutorService executorService = Executors.*newSingleThreadExecutor*();

Future future = executorService.submit(**new** Runnable() {

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

System.***out***.println("Asynchronous task");

}

});

future.get(); **// 获得执行完 run 方法后的返回值，这里使用的 Runnable，所以这里没有返回值 返回的是 null。**

executorService.shutdown();

* submit(Callable)

**// 从 Executors 中获得 ExecutorService**

ExecutorService executorService = Executors.*newSingleThreadExecutor*();

Future **future** = executorService.submit(**new** Callable() {

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

System.***out***.println("Asynchronous Callable");

**return** "Callable Result";

}

});

System.***out***.println("future.get() = " + **future.get()**);

executorService.shutdown();

call方法能够返回一个结果

##### Exectuors关闭

shutdown 只是将空闲的线程 interrupt() 了， shutdown（）之前提交的任务可以继续执行直到结束。

shutdownNow 是 interrupt 所有线程， 因此大部分线程将立刻被中断。之所以是大部分，而不是全部 ，是因为 interrupt()方法能力有限。

#### ThreadPoolExecutor线程池执行者

java.util.concurrent.ThreadPoolExecutor是ExecutorService 接口的一个实现。 ThreadPoolExecutor 使用其内部池中的线程执行给定线程任务(Callable 或者 Runnable)。

**int** corePoolSize = 5;

**int** maxPoolSize = 10;

**long** keepAliveTime = 5000;

ExecutorService threadPoolExecutor = **new** ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.***MILLISECONDS***,

**new** LinkedBlockingQueue<Runnable>());

**// 构造方法参数列表解释：**

**// corePoolSize - 池中所保存的线程数，包括空闲线程。**

**// maximumPoolSize - 池中允许的最大线程数。**

**// keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。**

**// unit - keepAliveTime 参数的时间单位。**

**// workQueue - 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。**

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空 闲 线 程 。 如 果 内 部 任 务 队 列 已 满 ， 而 且 有 至 少 corePoolSize 正 在 运 行 ， 但 是 运 行 线 程 的 数 量 低 于maximumPoolSize，一个新的线程将被创建去执行该任务。

**使用下面的代码实例深度分析：**

**public** **class** Test {

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

**//核心线程数为2**

**//最大线程数为5**

**//workQueue任务队列大小为1,也就是说只能放进去一个任务,等待线程池调度**

ExecutorService executorService = **new** ThreadPoolExecutor(2, 5, 0, TimeUnit.***DAYS***, **new** ArrayBlockingQueue<>(1),

**new** ThreadFactory() {

@Override

**public** Thread newThread(Runnable r) {

Thread thread = **new** Thread(r);

**// thread.setDaemon(true);**

**return** thread;

}

});

**// ThreadPoolExecutor对象创建后，线程实际还没开始创建**

**// 执行execute时，检查当前池中线程数大小是否小于core number, 如果小于，则创建新线程，任务1立即执行**

executorService.execute(() -> {

System.***out***.println("任务1@" + Thread.*currentThread*().getName());

*sleepTime*();

System.***out***.println(1);

});

**// 检查当前池中线程数大小是否小于core number, 如果小于，则创建新线程，然后任务2就立即执行了**

executorService.execute(() -> {

System.***out***.println("任务2@" + Thread.*currentThread*().getName());

*sleepTime*();

System.***out***.println(2);

});

**// 现在线程池中已经创建两个线程了,已经不小于core number**

**// 检查当前池中线程数大小是否小于core number, 如果不小于，则尝试放入队列**

**// 这个任务是加到队列去的, 注意队列大小只有1，只能放进去1个任务**

**// TODO 队列中的任务是什么时候取出来的？ 任务1或者2结束后，其所占用的线程，变成空闲线程，会执行队列中的任务，这个任务是在最后才运行，比4运行的还晚**

executorService.execute(() -> {

System.***out***.println("任务3@" + Thread.*currentThread*().getName());

*sleepTime*();

System.***out***.println(3);

});

**// 检查当前池中线程数大小是否小于core number, 如果不是,则偿试放入队列,放入队列也失败，则增加新的worker线程**

**// 这个任务是加到core以外的新线程去的**

executorService.execute(() -> {

System.***out***.println("任务4@" + Thread.*currentThread*().getName());

*sleepTime*();

System.***out***.println(4);

});

}

**public** **static** **void** sleepTime() {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

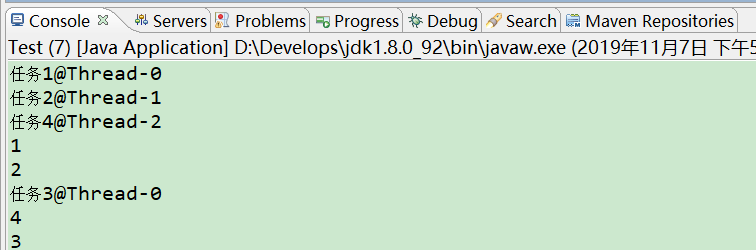
e.printStackTrace();

}

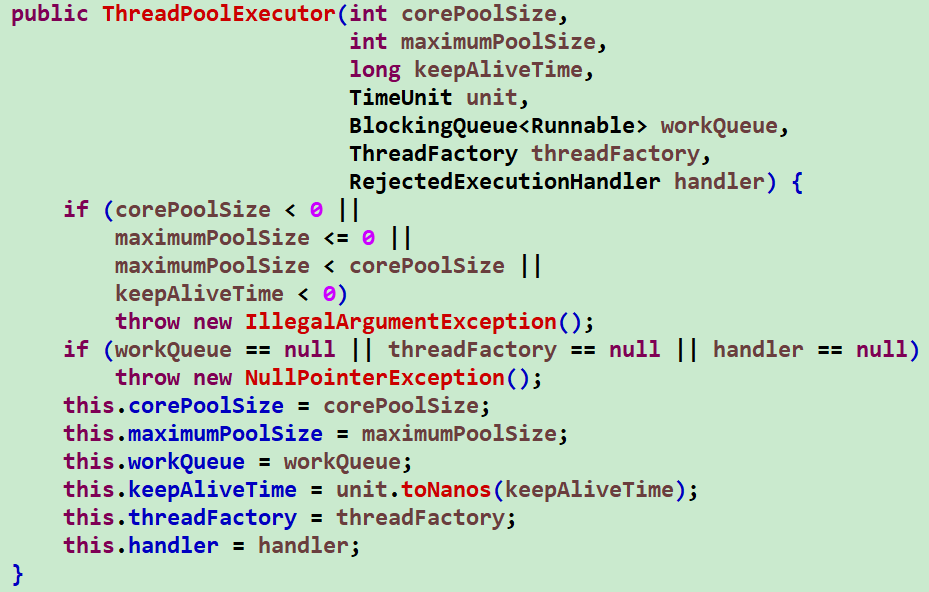
}

}

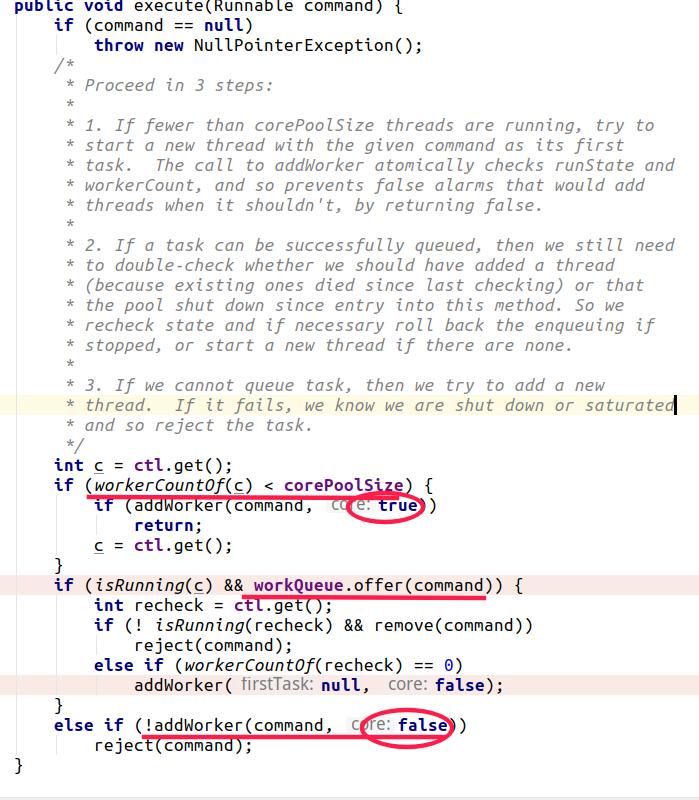
**执行结果：**



在new ThreadPoolExecutor()时，Thread线程对象并没有初始化. 这里仅仅指定了几个初始参数，也就是说线程池里面并没有线程。



执行第一个execute时，

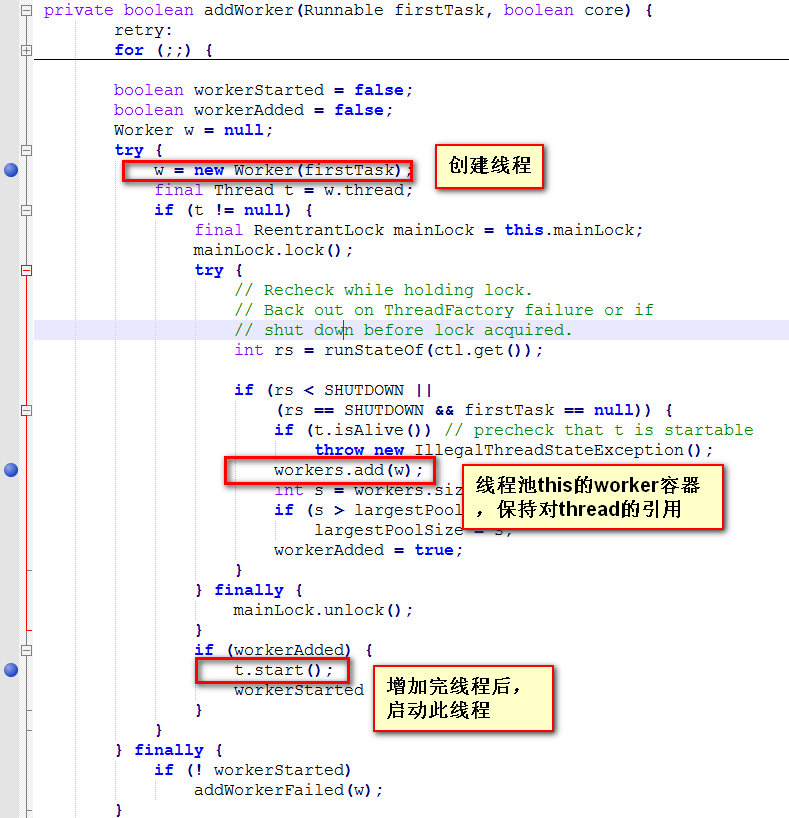


第一个if，判断如果当前线程数小于corePoolSize, 则创建新的核心worker对象（Worker指向Thread对象，就是线程对象，保持引用，保证不会被GC回收）我们的示例代码中，第1和第2个任务都是这样创建出线程的，创建完之后就return。代码不再往下执行。

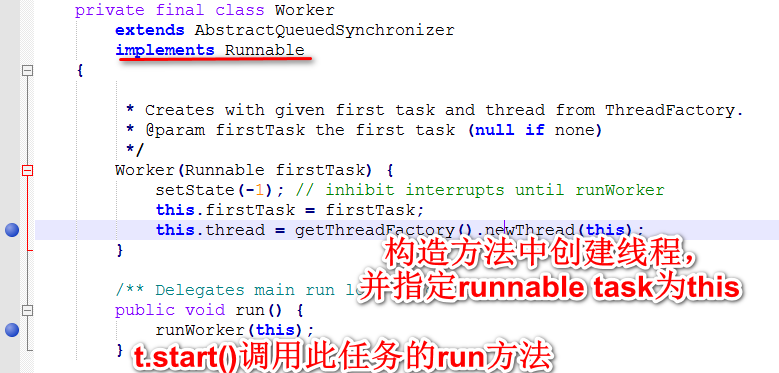
第二个if，判断如果当前线程数大于corePoolSize，并尝试放入队列 workQueue.offer(command) , 放入**成功**后等待线程池调度【见后面的getTask()】，并不是立即就会调度。

第三个if，尝试放入队列 workQueue.offer(command)失败，增加一个非core线程。示例代码中，第4个任务是这样开始的。

创建线程的过程，



worker其实是一个线程任务



线程启动后，



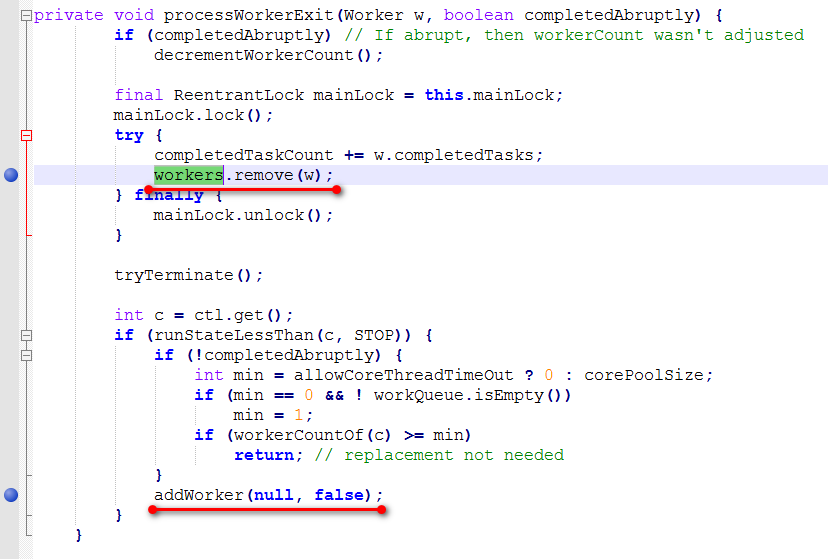
小于core number的时候，



task是不为空的，就直接走task.run()了。

只要是新创建的线程都是走第一个判断条件，从任务队列里面拿的话走的是第二个判断条件，因为它的task是空的。

没抛异常时，此线程会一直在while（task !=null || (task = getTask())!=null）中跑，那么有异常时，再看一下processWorkerExit



可以 看出，有异常时 旧的worker会被删除（GC回收），再创建新的Worker， 即有异常时 旧worker不可能再执行新的任务。

Q. 线程池是什么时候创建线程的？

*A.任务提交的时候*

Q.任务runnable task是先放到core到maxThread之间的线程，还是先放到队列？

*A.先放队列!!!*

Q. 队列中的任务是什么时候取出来的？

A. 新创建的线程把对应的任务执行完，开始从队列中取任务执行，worker中 runWorker()，一个任务完成后，会取下一个任务。

Q. 什么时候会触发reject策略？

A.队列满并且maxthread也满了， 还有新任务，默认策略是reject

Q. core到maxThread之间的线程什么时候会die?

A.  没有任务时，或者抛异常时。core线程也会die的，core到maxThread之间的线程有可能会晋升到core线程区间，core max只是个数值，线程并不是创建后就固定在一个区间了。

Q. task抛出异常，线程池中这个work thread还能运行其他任务吗?

A. 不能。但是会创建新的线程, 新线程可以运行其他task。

对于 schedulerThreadPoolExecutor? 虽然有新线程，但是旧的循环任务不会再继续执行了， 开发实践推荐任务中捕获所有Exception

### 并发队列-阻塞队列。

#### 分析

常用的并发队列分为阻塞队列和非阻塞队列。

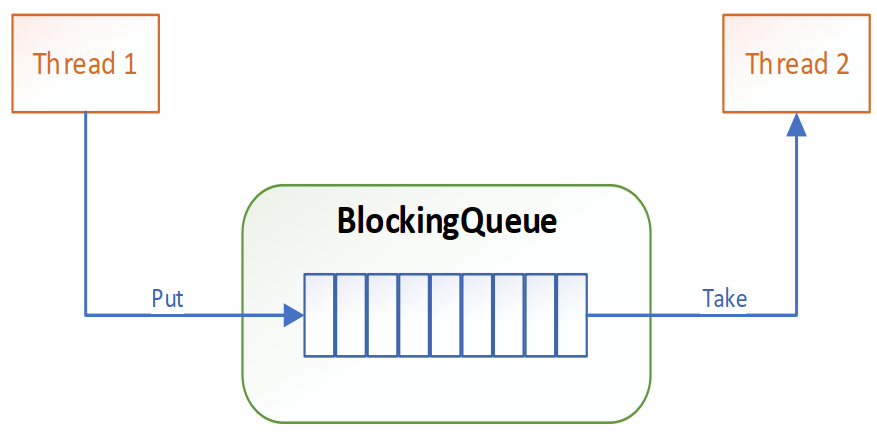
阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：

1、从阻塞队列取数据时，如果队列为空，获取元素的线程会阻塞等待直到队列变为非空。

2、向阻塞队列插入数据时，如果队列为满，存储元素的线程会阻塞等待直到队列为非满。

阻塞队列是**线程安全**的。 并发包下的很多高级同步类的实现都是基于BlockingQueue实现的。

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。



这个阻塞队列里面存储的是什么？

存储的是线程任务，应该是Runnable的实现。

一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

#### BlockingQueue的方法

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。



四组不同的行为方式解释：

抛异常： 如果试图的操作无法立即执行，抛一个异常。

特定值： 如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

阻塞： 如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

超时： 如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

**无法向一个 BlockingQueue 中插入 null。如果你试图插入 null， BlockingQueue 将会抛出一个NullPointerException.**

#### BlockingQueue 的实现类

BlockingQueue只是一个接口，它有如下实现类：

* ArrayBlockingQueue

ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个

数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

* LinkedBlockingQueue

LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如

果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。

* DelayQueue

DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现java.util.concurrent.Delayed 接口。

* PriorityBlockingQueue

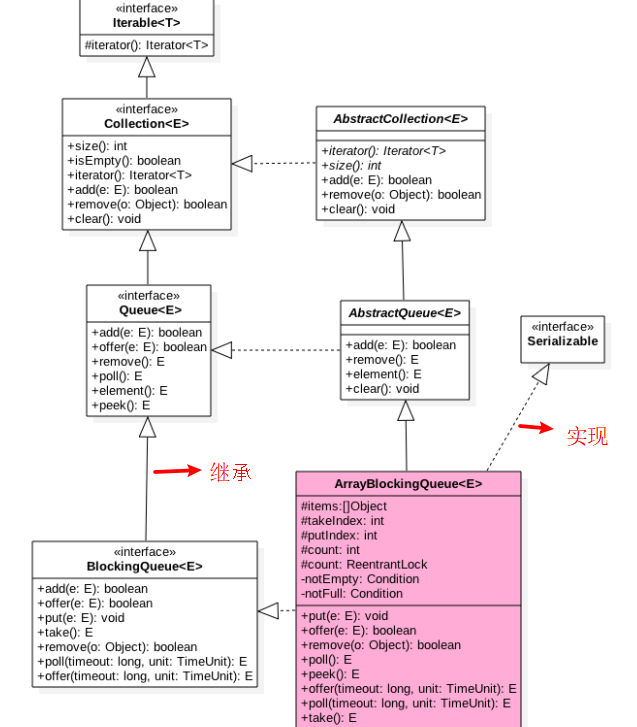
PriorityBlockingQueue 是 一 个 无 界 的 并 发 队 列 。 它 使 用 了 和 类java.util.PriorityQueue 一 样 的 排 序 规 则 。 你 无 法 向 这 个 队 列 中 插 入 null 值 。 所 有 插 入 到PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。

* SynchronousQueue

SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

#### ArrayBlockingQueue详细介绍

##### 继承与实现关系图



如上图， ArrayBlockingQueue 内部有个数组 items 用来存放队列元素， putindex 下标标示入队元素下标，takeIndex 是出队下标， count 统计队列元素个数，从定义可知道并没有使用 volatile 修饰，这是因为访问这些变量使用都是在锁块内，并不存在可见性问题。另外有个独占锁 lock 用来对出入队操作加锁，这导致同时只有一个线程可以访问入队出队，另外 notEmpty， notFull 条件变量用来进行出入队的同步。

##### ArrayBlockingQueue方法

* 构造方法

**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();

}

另外构造函数必须传入队列大小参数，所以为有界队列，默认lock为非公平锁。

公平锁：

就是在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照 FIFO 的规则从队列中取到自己。

非公平锁：

比较粗鲁，上来就直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式。

##### ArrayBlockingQueue 示例

需求：在多线程操作下，一个数组中最多只能存入 3 个元素。多放入不可以存入数组，或等待某线程对数组中某个元素取走才能放入，要求使用 java 的多线程来实现。

**public** **class** TestArrayBlockingQueue {

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

**final** BlockingQueue queue = **new** ArrayBlockingQueue(3);

**//创建两个线程去放数据**

**for** (**int** i = 0; i < 2; i++) {

**new** Thread() {

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

**while** (**true**) {

**try** {

Thread.*sleep*((**long**) (Math.*random*() \* 1000));

System.***out***.println(Thread.*currentThread*().getName() + "准备放数据!");

queue.put(1);

System.***out***.println(

Thread.*currentThread*().getName() + "已经放了数据， " + "队列目前有" + queue.size() + "个数据");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

**// 创建一个线程去取数据**

**new** Thread() {

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

**while** (**true**) {

**try** {

**// 将此处的睡眠时间分别改为 100 和 1000，观察运行结果**

Thread.*sleep*(1000);

System.***out***.println(Thread.*currentThread*().getName() + "准备取数据!");

System.***err***.println(queue.take());

System.***out***.println(

Thread.*currentThread*().getName() + "已经取走数据， " + "队列目前有" + queue.size() + "个数据");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

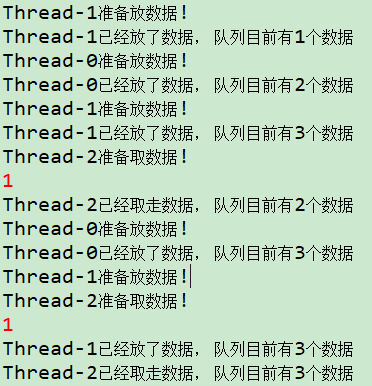
}

}.start();

}

}

执行结果：



分析执行结果：

此处while(true)的作用就是两个线程死循环的不停的去放数据，一个线程死循环的不停的去取数据。

Thread-0先创建，但是还没有来的及执行，cpu就被第二个线程Thread-1抢了，Thread-1向队列里面放入了一个数据，Thread-0继续执行，再次向队列里面放入一个数据。Thread-1又放入一个数据。此时cpu分配给第三个线程Thread-2，去取数据。