尚硅谷：https://www.bilibili.com/video/BV16J411h7Rd?p=4

黑马程序员：https://www.bilibili.com/video/BV16J411h7Rd?p=221&spm\_id\_from=pageDriver

# 进程与线程

## 进程与线程

### 进程

* 程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备。进程就是用来加载指令、管理内存、管理 IO 的
* 当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。
* 进程就可以视为程序的一个实例。大部分程序可以同时运行多个实例进程（例如记事本、画图、浏览器等），也有的程序只能启动一个实例进程（例如网易云音乐、360 安全卫士等）

### 线程

* 一个进程之内可以分为一到多个线程。
* 一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给 CPU 执行
* Java 中，线程作为最小调度单位，进程作为资源分配的最小单位。 在 windows 中进程是不活动的，只是作为线程的容器

### 二者对比

* 进程基本上相互独立的，而线程存在于进程内，是进程的一个子集
* 进程拥有共享的资源，如内存空间等，供其内部的线程共享
* 进程间通信较为复杂：
* 同一台计算机的进程通信称为 IPC（Inter-process communication）
* 不同计算机之间的进程通信，需要通过网络，并遵守共同的协议，例如 HTTP
* 线程通信相对简单，因为它们共享进程内的内存，一个例子是多个线程可以访问同一个共享变量
* 线程更轻量，线程上下文切换成本一般上要比进程上下文切换低

## 井行与并发

### 单核cpu

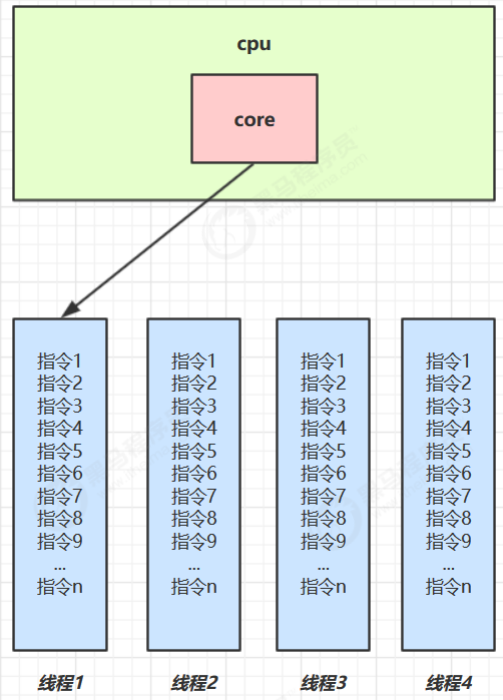
单核cpu下，线程实际还是串行执行的。操作系统中有一个组件叫做任务调度器，将cpu的时间片（windows

下时间片最小约为 15 毫秒）分给不同的程序使用，只是由于 cpu 在线程间（时间片很短）的切换非常快，人类感

觉是 同时运行的 。总结为一句话就是： **微观串行，宏观并行**

一般会将这种线程轮流使用CPU的做法称为并发，concurrent

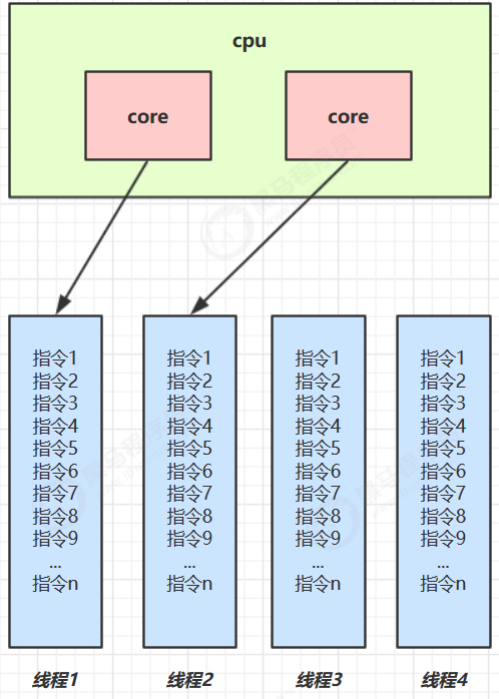




### 多核cpu

多核cp下，每个核（core）都可以调度运行线程，这时候线程可以是并行的。





### 井行与并发

引Rob Pik的一段描述：

* 并发（concurrent）是同一时间应对（dealing with）多件事情的能力
* 并行（parallel）是同一时间动手做（doing）多件事情的能力

例子：

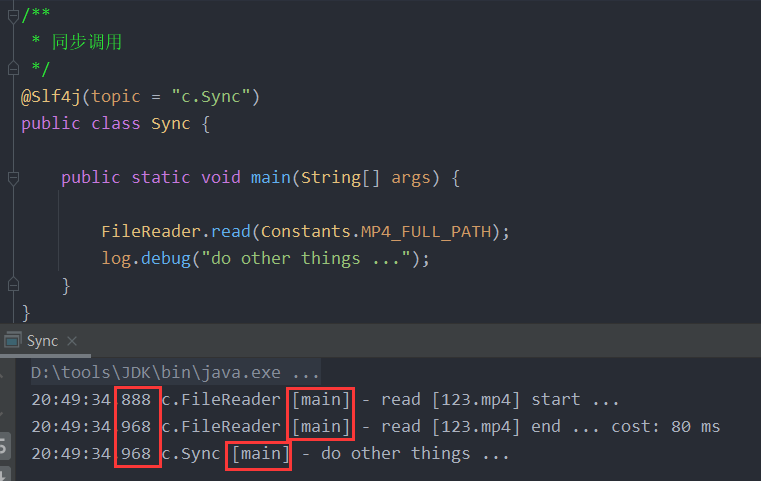
* 家庭主妇做饭、打扫卫生、给孩子喂奶，她一个人轮流交替做这多件事，这时就是并发；
* 家庭主妇雇了个保姆，她们一起这些事，这时既有并发，也有并行（这时会产生竞争，例如锅只有一口，一个人用锅时，另一个人就得等待）
* 雇了3个保姆，一个专做饭、一个专打扫卫生、一个专喂奶，互不干扰，这时是并行

## 应用

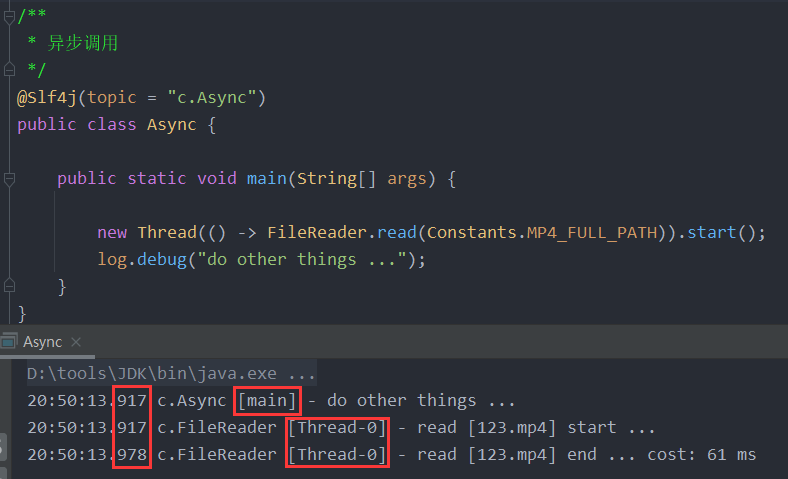
### 应用之异步调用

以调用方角度来讲，如果：

* 需要等待结果返回，才能继续运行就是同步



* 不需要等待结果返回，就能继续运行就是异步



注意：同步在多线程中还有另外一层意思，是让多个线程步调一致

#### 1）设计

多线程可以让方法执行变为异步的（即不要巴巴干等着）比如说读取磁盘文件时，假设读取操作花费了 5 秒钟，如

果没有线程调度机制，这 5 秒 cpu 什么都做不了，其它代码都得暂停...

#### 2）结论

* 比如在项目中，视频文件需要转换格式等操作比较费时，这时开一个新线程处理视频转换，避免阻塞主线程；
* tomcat的异步 servlet 也是类似的目的，让用户线程处理耗时较长的操作，避免阻塞 tomcat 的工作线程；
* ui程序中，开线程进行其他操作，避免阻塞ui线程。

### 应用之提高效率

充分利用多核 cpu 的优势，提高运行效率。想象下面的场景，执行 3 个计算，最后将计算结果汇总。



* 如果是串行执行，那么总共花费的时间是 10 + 11 + 9 + 1 = 31ms
* 但如果是四核cpu，各个核心分别使用线程 1 执行计算 1，线程 2 执行计算 2，线程 3 执行计算 3，那么 3 个线程是并行的，花费时间只取决于最长的那个线程运行的时间，即 11ms 最后加上汇总时间只会花费 12ms
* 注意：

**需要在多核cpu才能提高效率，单核仍然时是轮流执行**

#### 1）设计

>>>>> 代码见【应用之效率-案例1】<<<<<

#### 2）结论

* 单核 cpu 下，多线程不能实际提高程序运行效率，只是为了能够在不同的任务之间切换，不同线程轮流使用cpu ，不至于一个线程总占用 cpu，别的线程没法干活
* 多核 cpu 可以并行跑多个线程，但能否提高程序运行效率还是要分情况的
* 有些任务，经过精心设计，将任务拆分，并行执行，当然可以提高程序的运行效率。但不是所有计算任务都能拆分（参考后文的【阿姆达尔定律】）
* 也不是所有任务都需要拆分，任务的目的如果不同，谈拆分和效率没啥意义
* IO 操作不占用 cpu，只是我们一般拷贝文件使用的是【阻塞 IO】，这时相当于线程虽然不用 cpu，但需要一直等待 IO 结束，没能充分利用线程。所以才有后面的【非阻塞 IO】和【异步 IO】优化

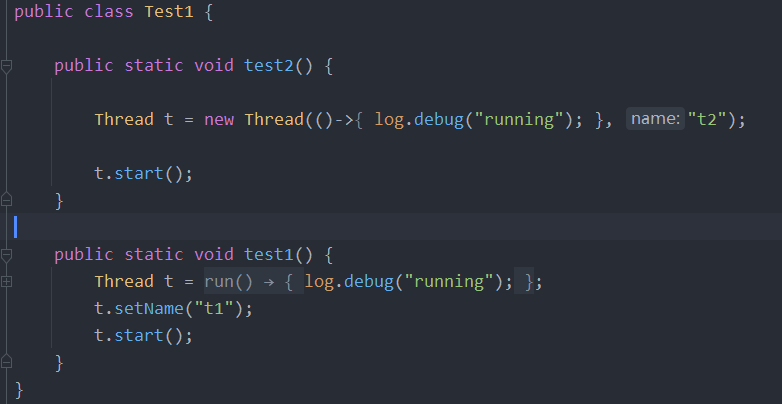
# Java线程

## 创建和运行线程

### 方法一：直接使用Thread







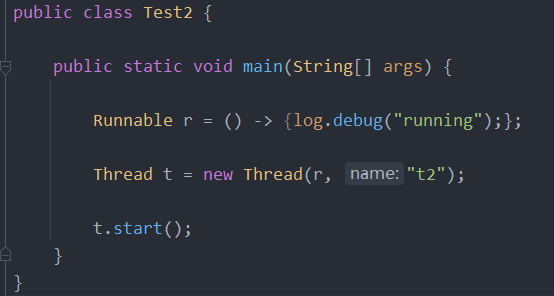
### 方法二：使用Runnable配合Thread

把【线程】和【任务】（要执行的代码）分开：

* Thread 代表线程
* Runnable 可运行的任务（线程要执行的代码）







### 原理之Thread与Runnable的关系

FutureTask能够接收Callable类型的参数，用来处理有返回结果的情况



### 方法三：FutureTask配合Thread

## 观察多个线程同时运行

## 查看进程线程的方法

## \* 原理之线程运行

## 常见方法

## start 与 run

## sleep 与 yield

## join方法详解

## interrupt方法详解

## 不推荐的方法

## 主线程与守护线程

## 五种状态

## 六种状态

## 习题

### 单核cpu

单核cpu下，线程实际还是串行执行的。操作系统中有一个组件叫做任务调度器，将cpu的时间片（windows

下时间片最小约为 15 毫秒）分给不同的程序使用，只是由于 cpu 在线程间（时间片很短）的切换非常快，人类感

觉是 同时运行的 。总结为一句话就是： **微观串行，宏观并行**

一般会将这种线程轮流使用CPU的做法称为并发，concurrent

### 多核cpu

多核cp下，每个核（core）都可以调度运行线程，这时候线程可以是并行的。

### 井行与并发

引Rob Pik的一段描述：

* 并发（concurrent）是同一时间应对（dealing with）多件事情的能力
* 并行（parallel）是同一时间动手做（doing）多件事情的能力

例子：

* 家庭主妇做饭、打扫卫生、给孩子喂奶，她一个人轮流交替做这多件事，这时就是并发；
* 家庭主妇雇了个保姆，她们一起这些事，这时既有并发，也有并行（这时会产生竞争，例如锅只有一口，一个人用锅时，另一个人就得等待）
* 雇了3个保姆，一个专做饭、一个专打扫卫生、一个专喂奶，互不干扰，这时是并行

# 共享模型之管程

## 共享带来的问题

## synchronized 解决方案

## 方法上的 synchronized

## 变量的线程安全分析

## 习题

## Monitor概念

## wait notify

## wait notify的正确姿势

## Park& Unpark

## 重新理解线程状态转换

## 多把锁

## 活跃性

## ReentrantLock

# 共享模型内存

## Java内存模型

JMM 即 Java Memory Model，它定义了主存、工作内存抽象概念，底层对应着 CPU 寄存器、缓存、硬件内存、CPU 指令优化等。

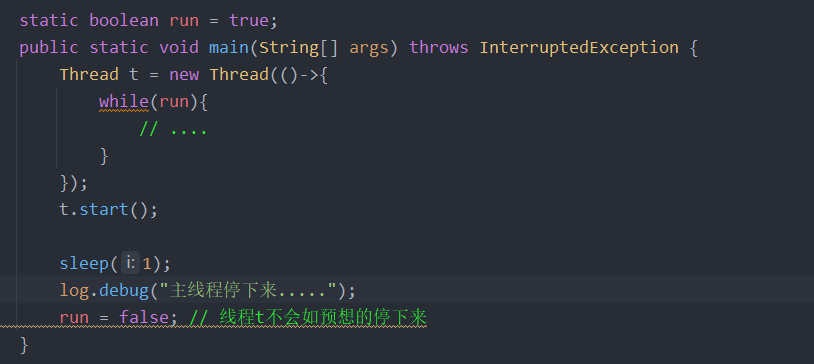
JMM 体现在以下几个方面：

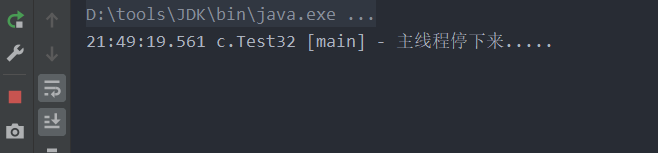
* 原子性 - 保证指令不会受到线程上下文切换的影响
* 可见性 - 保证指令不会受 cpu 缓存的影响
* 有序性 - 保证指令不会受 cpu 指令并行优化的影响

## 可见性

### 1、退不出的循环

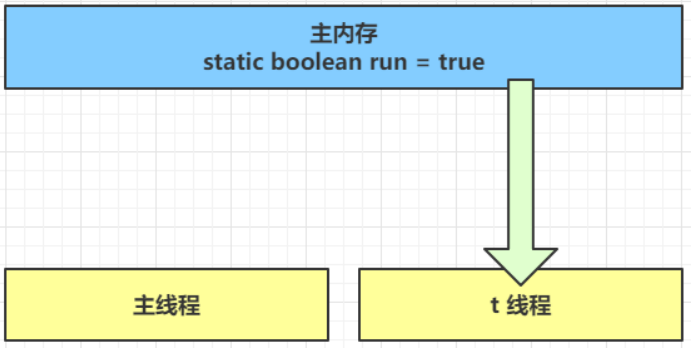
先来看一个现象，main 线程对 run 变量的修改对于 t 线程不可见，导致了 t 线程无法停止



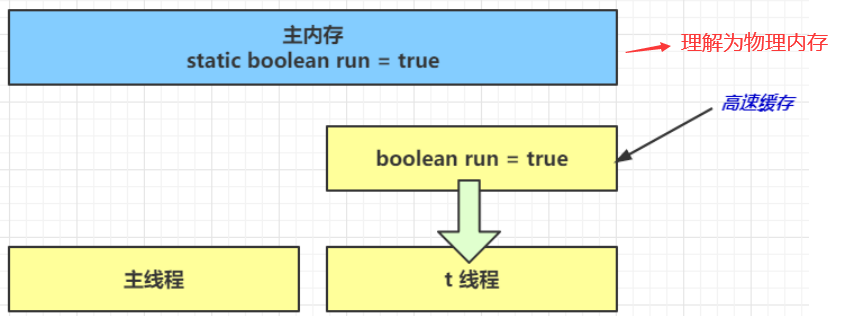


### 2、原因分析

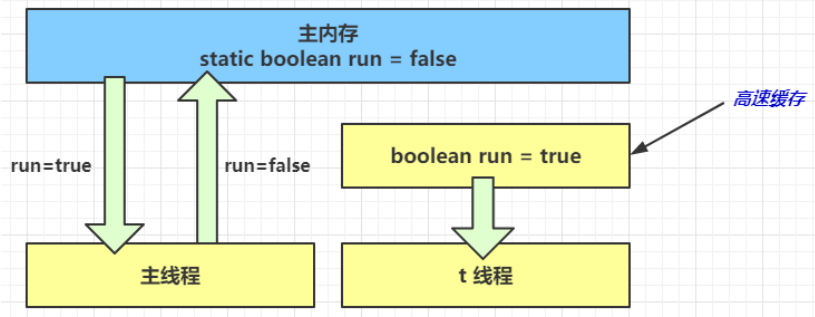
2.1 初始状态：t 线程刚开始从主内存读取了 run 的值到工作内存。



2.2 因为 t 线程要频繁从主内存中读取 run 的值，JIT 编译器会将 run 的值缓存至自己工作内存中的高速缓存中，减少对主存中 run 的访问，提高效率



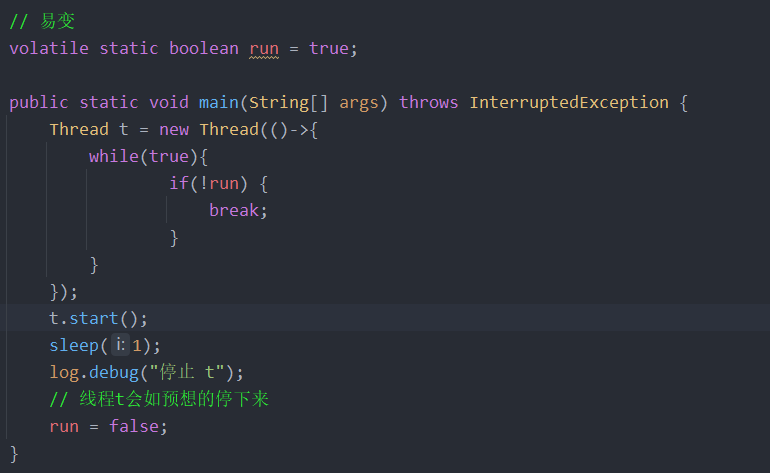
2.3 1秒之后，main 线程修改了 run 的值，并同步至主存，而 t 是从自己工作内存中的高速缓存中读取这个变量的值，结果永远是旧值

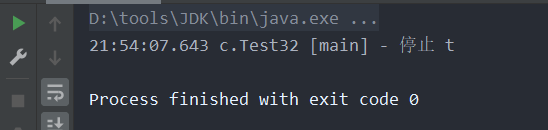


### 3、解决方法

使用volatile（易变关键字）：

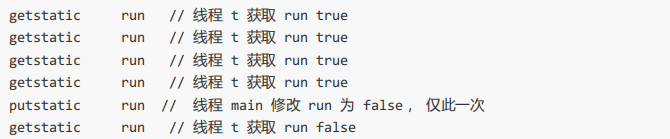
它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存



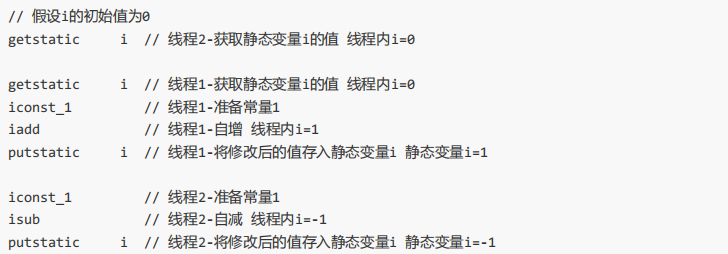


### 4、可见性 vs 原子性

前面例子体现的实际就是可见性，它保证的是在多个线程之间，一个线程对 volatile 变量的修改对另一个线程可见， 不能保证原子性，仅用在一个写线程，多个读线程的情况： 上例从字节码理解是这样的：

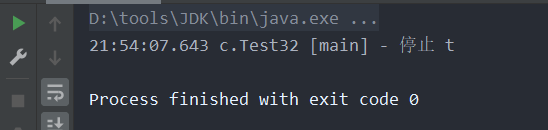


比较一下之前我们将线程安全时举的例子：两个线程一个 i++ 一个 i-- ，只能保证看到最新值，不能解决指令交错



注意：synchronized 语句块既可以保证代码块的原子性，也同时保证代码块内变量的可见性。但缺点是synchronized 是属于重量级操作，性能相对更低





## \* 原理之 CPU 缓存结构

## \* 模式之两阶段终止

## \* 模式之 Balking

## 有序性

JVM 会在不影响正确性的前提下，可以调整语句的执行顺序，思考下面一段代码



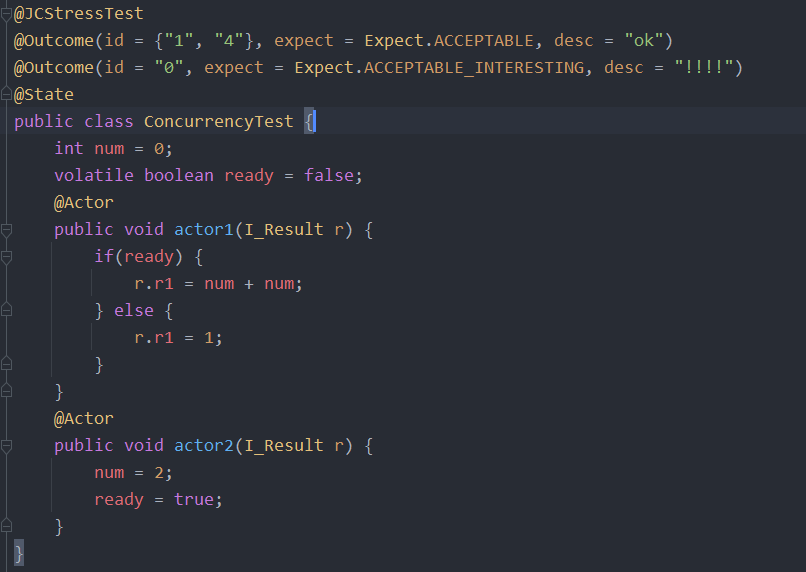
可以看到，至于是先执行 i 还是 先执行 j ，对最终的结果不会产生影响。所以，上面代码真正执行时，既可以是



这种特性称之为『指令重排』，多线程下『指令重排』会影响正确性。为什么要有重排指令这项优化呢？从 CPU执行指令的原理来理解一下吧

### \* 原理之指令级并行

### 诡异的结果



I\_Result是一个对象，有一个属性 r1 用来保存结果，问，可能的结果有几种？

有同学这么分析

* 情况1：线程1 先执行，这时 ready = false，所以进入 else 分支结果为 1
* 情况2：线程2 先执行 num = 2，但没来得及执行 ready = true，线程1 执行，还是进入 else 分支，结果为1
* 情况3：线程2 执行到 ready = true，线程1 执行，这回进入 if 分支，结果为 4（因为 num 已经执行过了）
* 但我告诉你，结果还有可能是 0，信不信吧！

这种情况下是：线程2 执行 ready = true，切换到线程1，进入 if 分支，相加为 0，再切回线程2 执行 num = 2相信很多人已经晕了

这种现象叫做指令重排，是 JIT 编译器在运行时的一些优化，这个现象需要通过大量测试才能复现

### 解决方法

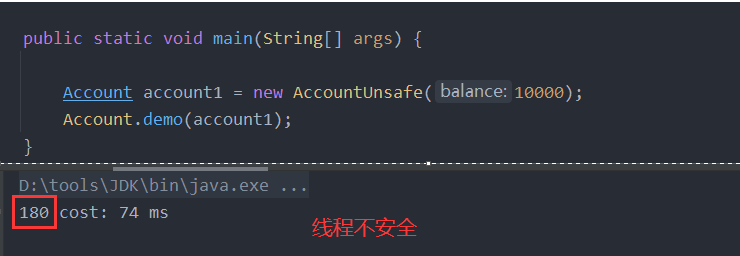
volatile修饰的变量，可以禁用指令重排，只需要在ready上加volatile即可，便可以防止ready=true之前的语句被执行重排

## \* 原理之 volatile

# 共享模型之无锁

## 一、问题提出





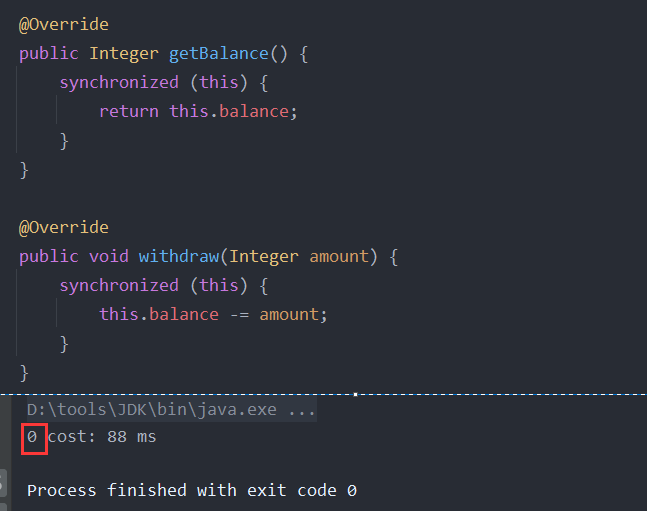
### 为什么不安全

* 单核的指令交错
* 多核的指令交错

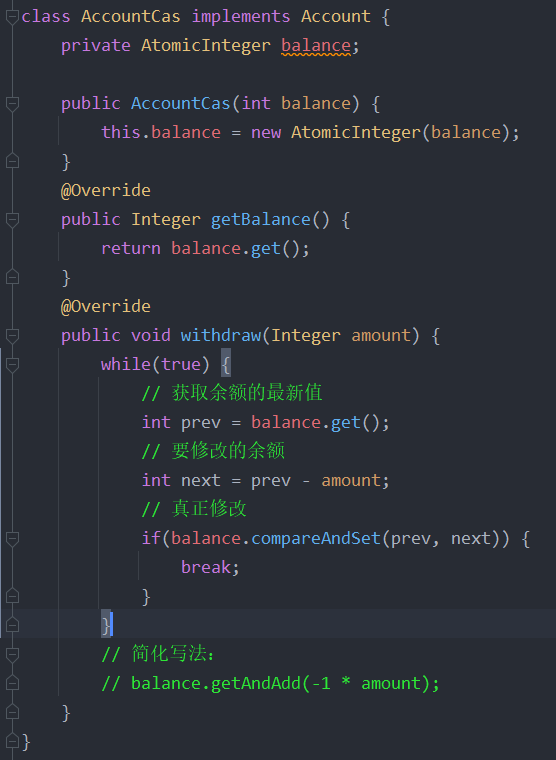


### 解决思路-锁

首先想到的是给 Account 对象加锁



### 解决思路-无锁

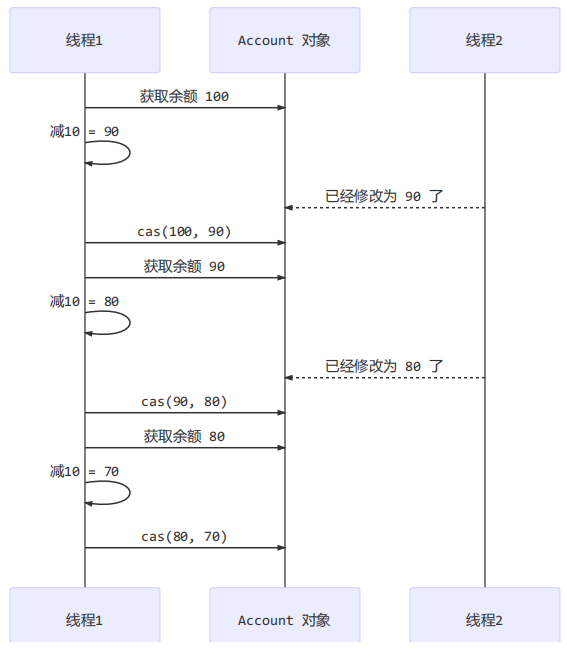


## 二、CAS与volatile

前面看到的 AtomicInteger 的解决方法，内部并没有用锁来保护共享变量的线程安全。那么它是如何实现的呢？



其中的关键是 **compareAndSet**，它的**简称就是 CAS** （也有 Compare And Swap 的说法），它必须是原子操作



注意：

其实CAS的底层是lock cmpxchg指令（X86 架构），在单核CPU和多核CPU下都能够保证【比较-交换】的原子性。

在多核状态下，某个核执行到带lock的指令时，CPU 会让总线锁住，当这个核把此指令执行完毕，再开启总线。这个过程中不会被线程的调度机制所打断，保证了多个线程对内存操作的准确性，是原子的。

### 慢动作分析

### Volatile

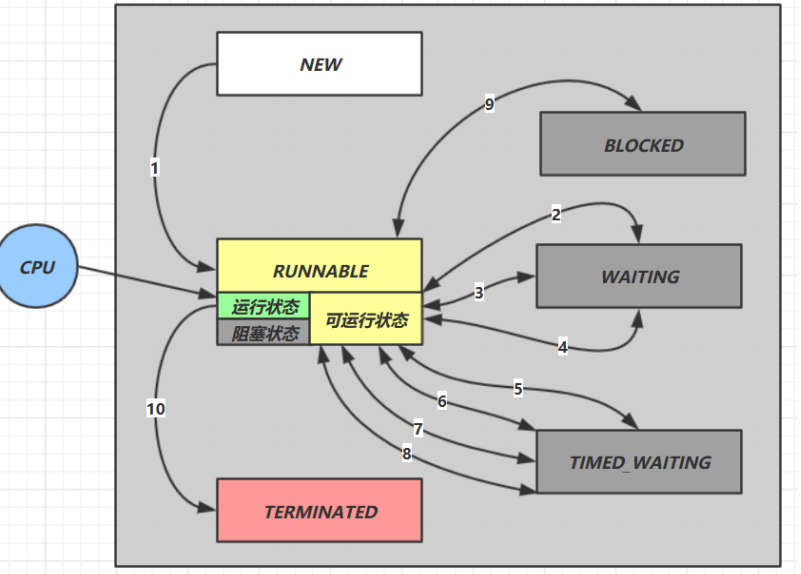
* 获取共享变量时，为了保证该变量的可见性，需要使用 volatile 修饰。
* 它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。即一个线程对 volatile 变量的修改，对另一个线程可见。
* 注意：

volatile仅仅保证了共享变量的可见性，让其它线程能够看到最新值，但不能解决指令交错问题（不能保证原子性）

* CAS 必须借助 volatile 才能读取到共享变量的最新值来实现【比较并交换】的效果

### 为什么无锁效率高

* 无锁情况下，即使重试失败，线程始终在高速运行，没有停歇，而 synchronized 会让线程在没有获得锁的时候，发生上下文切换，进入阻塞。打个比喻
* 线程就好像高速跑道上的赛车，高速运行时，速度超快，一旦发生上下文切换，就好比赛车要减速、熄火，等被唤醒又得重新打火、启动、加速... 恢复到高速运行，代价比较大
* 但无锁情况下，因为线程要保持运行，需要额外 CPU 的支持，CPU 在这里就好比高速跑道，没有额外的跑道，线程想高速运行也无从谈起，虽然不会进入阻塞，但由于没有分到时间片，仍然会进入可运行状态，还是会导致上下文切换。



### CAS的特点

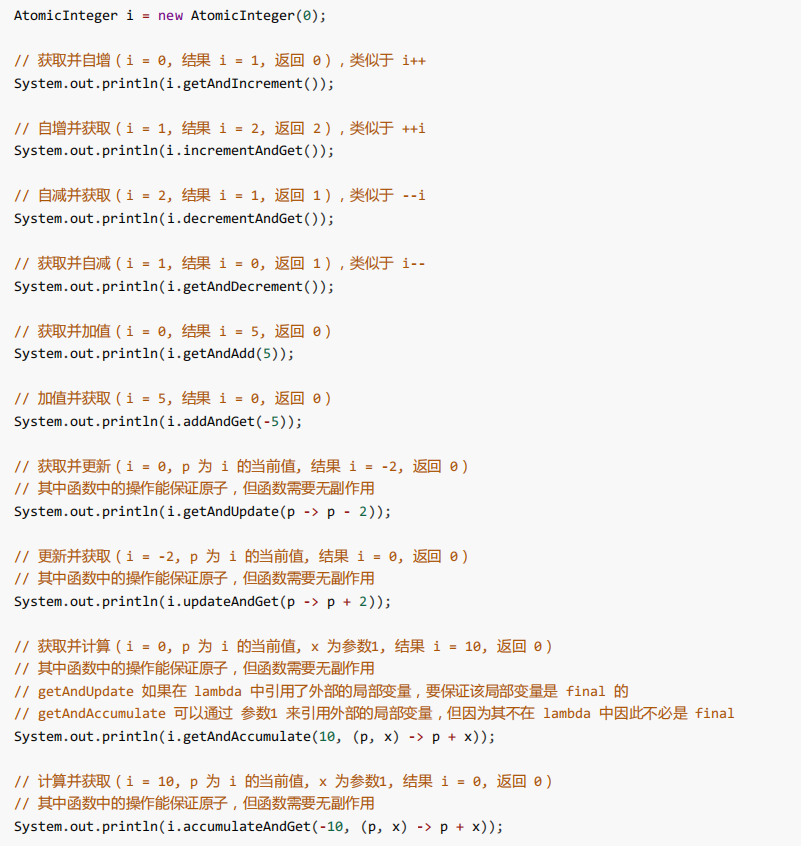
结合 CAS 和 volatile 可以实现无锁并发，适用于线程数少、多核 CPU 的场景下。

* CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
* synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。
* CAS 体现的是无锁并发、无阻塞并发，请仔细体会这两句话的意思
* 因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一
* 但如果竞争激烈，可以想到重试必然频繁发生，反而效率会受影响

## 三、原子整数

J.U.C 并发包提供了：

* AtomicBoolean
* AtomicInteger
* AtomicLong



**package** cn.itcast.test;  
  
**import** java.util.concurrent.atomic.AtomicInteger;  
**import** java.util.function.IntUnaryOperator;  
  
**public class** Test34 {  
 **public static void** main(String[] args) {  
  
 AtomicInteger i = **new** AtomicInteger(5);  
  
*// System.out.println(i.incrementAndGet()); // ++i 1  
// System.out.println(i.getAndIncrement()); // i++ 2  
  
 // 当前打印的是i的值，要想获得计算后的值，需要用i.get()获取。2 , 7  
// System.out.println(i.getAndAdd(5));  
// System.out.println(i.addAndGet(5)); // 12, 12  
  
 // 读取到 设置值  
// i.updateAndGet(value -> value \* 10);  
// i.getAndUpdate(value -> value \* 10);* System.***out***.println(*updateAndGet*(i, p -> p / 2));  
   
 System.***out***.println(i.get());  
 }  
  
 */\*\*  
 \* 自定义方法  
 \** ***@param i*** *\** ***@param operator*** *操作  
 \** ***@return*** *\*/* **public static int** updateAndGet(AtomicInteger i, IntUnaryOperator operator) {  
  
 **while** (**true**) {  
 *// 获取当前值* **int** prev = i.get();  
 *// 根据当前值计算，根据prev得到next* **int** next = operator.applyAsInt(prev);  
 *// 如果prev与当前线程中的值（共享变量）一致，即共享变量没有被修改，则共享变量更新为为next* **if** (i.compareAndSet(prev, next)) {  
 **return** next;  
 }  
 }  
 }  
}

## 四、原子引用

AtomicReference

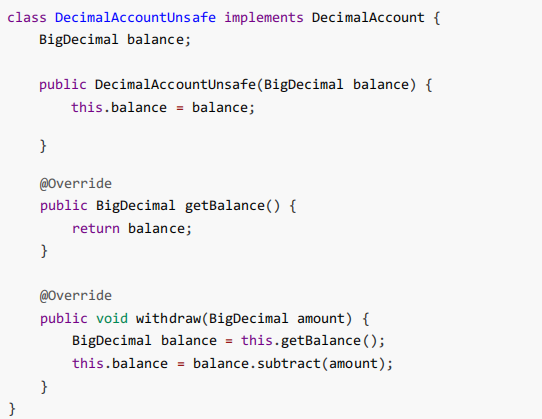
AtomicMarkableReference

AtomicStampedReference

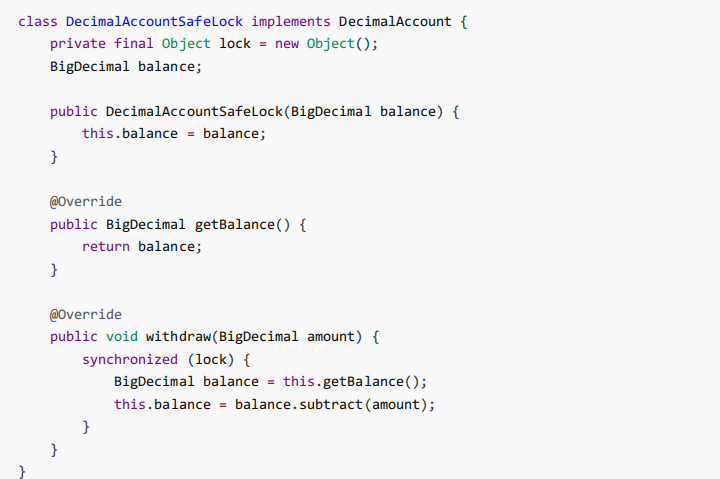
需求：试着提供不同的 DecimalAccount 实现，实现安全的取款操作



### 不安全实现

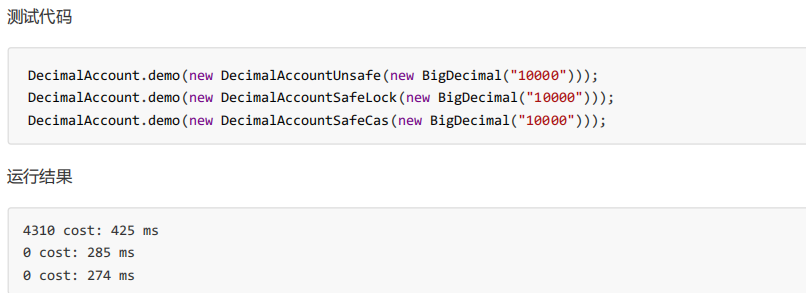


### 安全实现-使用锁



### 安全实现-使用CAS

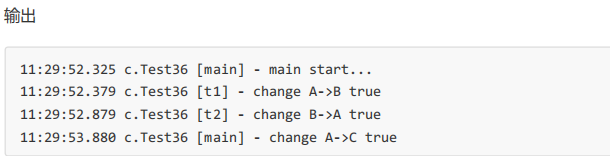




### ABA问题及解决

#### 4.1 ABA 问题





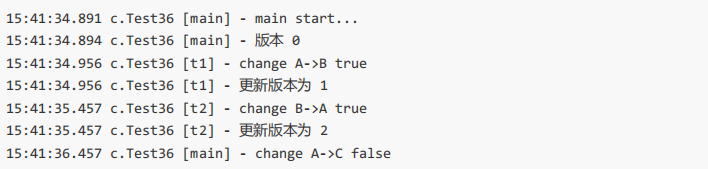
主线程仅能判断出共享变量的值与最初值 A 是否相同，不能感知到这种从 A 改为 B 又 改回 A 的情况，

如果主线程希望：

只要有其它线程【动过了】共享变量，那么自己的 cas 就算失败，这时，仅比较值是不够的，需要再加一个版本号

#### 4.2 解决方案一：AtomicStampedReference

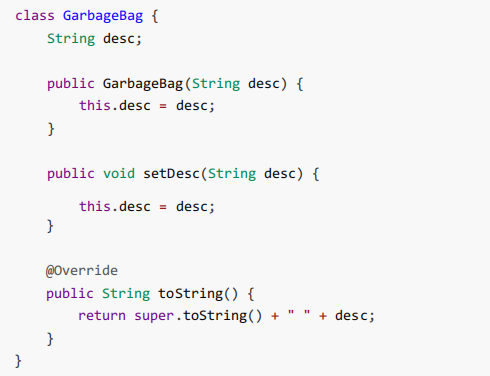
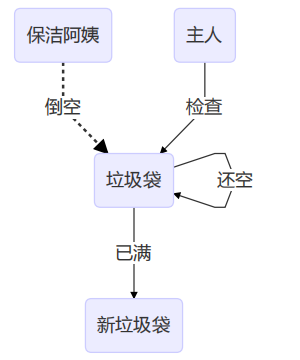




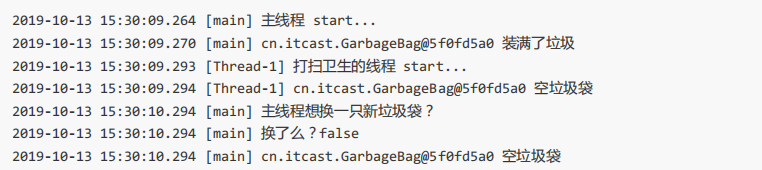
AtomicStampedReference 可以给原子引用加上版本号，追踪原子引用整个的变化过程，如： A -> B -> A -> C ，通过AtomicStampedReference，我们可以知道，引用变量中途被更改了几次。

#### 4.3 解决方案二：AtomicMarkableReference

有时候并不关心引用变量更改了几次，只是单纯的关心是否更改过，所以就有了AtomicMarkableReference







## 五、原子数组

### 不安全的数组

### 安全的数组

## 六、字段更新器

## 七、原子累加器

### 累加器性能比较

### \* 源码之LongAdder

### cas锁

### \* 原理之伪共享

## 八、**Unsafe**

# 共享模型之不可变

## 一、日期转换的问题

### 问题提出

### 思路-同步锁

### 思路-不可变

## 二、不可变设计

### final的使用

### 保护性拷贝

### \* 模式之享元

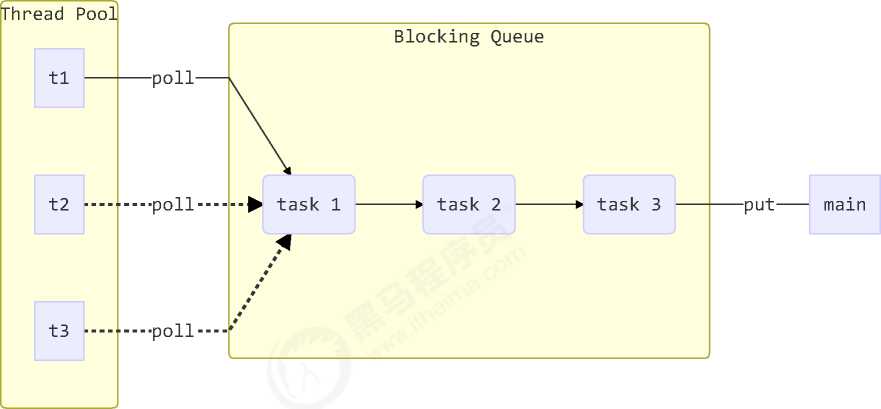
### \* 原理之final

## 三、无状态

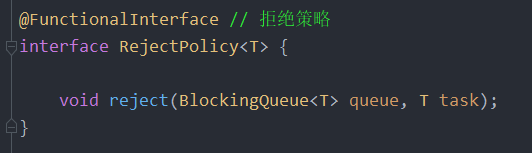
# 共享模型之工具

## 一、线程池

### 自定义线程池



步骤1：自定义拒绝策略接口



步骤2：自定义任务队列

@Slf4j(topic = **"c.BlockingQueue"**)  
**class** BlockingQueue<T> {  
  
 *// 1. 任务队列* **private** Deque<T> **queue** = **new** ArrayDeque<>();  
  
 *// 2. 锁* **private** ReentrantLock **lock** = **new** ReentrantLock();  
  
 *// 3. 生产者条件变量* **private** Condition **fullWaitSet** = **lock**.newCondition();  
  
 *// 4. 消费者条件变量* **private** Condition **emptyWaitSet** = **lock**.newCondition();  
  
 *// 5. 容量* **private int capcity**;  
  
 **public** BlockingQueue(**int** capcity) {  
 **this**.**capcity** = capcity;  
 }  
  
 *// 带超时阻塞获取* **public** T poll(**long** timeout, TimeUnit unit) {  
 **lock**.lock();  
 **try** {  
 *// 将 timeout 统一转换为 纳秒* **long** nanos = unit.toNanos(timeout);  
 **while** (**queue**.isEmpty()) {  
 **try** {  
 **if** (nanos <= 0) {  
 **return null**;  
 }  
 *// 返回值是剩余时间* nanos = **emptyWaitSet**.awaitNanos(nanos);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 T t = **queue**.removeFirst();  
 **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *// 阻塞获取，死等* **public** T take() {  
 **lock**.lock();  
 **try** {  
 **while** (**queue**.isEmpty()) {  
 **try** {  
 **emptyWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *// 拿到并移除第一个元素* T t = **queue**.removeFirst();  
 *// 唤醒阻塞添加* **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *// 阻塞添加* **public void** put(T task) {  
 **lock**.lock();  
 **try** {  
 **while** (**queue**.size() == **capcity**) {  
 **try** {  
 ***log***.debug(**"等待加入任务队列 {} ..."**, task);  
 **fullWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"加入任务队列 {}"**, task);  
 **queue**.addLast(task);  
 *// 唤醒阻塞获取* **emptyWaitSet**.signal();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *// 带超时时间阻塞添加* **public boolean** offer(T task, **long** timeout, TimeUnit timeUnit) {  
 **lock**.lock();  
 **try** {  
 **long** nanos = timeUnit.toNanos(timeout);  
 **while** (**queue**.size() == **capcity**) {  
 **try** {  
 **if**(nanos <= 0) {  
 **return false**;  
 }  
 ***log***.debug(**"等待加入任务队列 {} ..."**, task);  
 nanos = **fullWaitSet**.awaitNanos(nanos);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"加入任务队列 {}"**, task);  
 **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 **return true**;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *// 获取大小* **public int** size() {  
 **lock**.lock();  
 **try** {  
 **return queue**.size();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 **public void** tryPut(RejectPolicy<T> rejectPolicy, T task) {  
 **lock**.lock();  
 **try** {  
 *// 判断队列是否满* **if**(**queue**.size() == **capcity**) {  
 rejectPolicy.reject(**this**, task);  
 } **else** { *// 有空闲* ***log***.debug(**"加入任务队列 {}"**, task);  
 **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 }  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
}

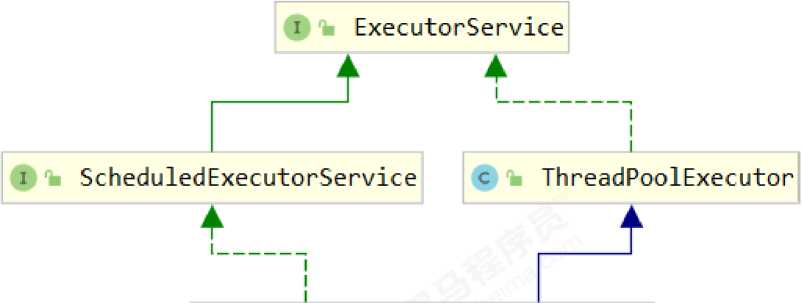
步骤3：自定义线程池

@Slf4j(topic = **"c.ThreadPool"**)  
**class** ThreadPool {  
  
 *// 任务队列* **private** BlockingQueue<Runnable> **taskQueue**;  
  
 *// 线程集合* **private** HashSet<Worker> **workers** = **new** HashSet<>();  
  
 *// 核心线程数* **private int coreSize**;  
  
 *// 获取任务时的超时时间* **private long timeout**;  
  
 **private** TimeUnit **timeUnit**;  
  
 **private** RejectPolicy<Runnable> **rejectPolicy**;  
  
 **public** ThreadPool(**int** coreSize, **long** timeout, TimeUnit timeUnit, **int** queueCapcity, RejectPolicy<Runnable> rejectPolicy) {  
 **this**.**coreSize** = coreSize;  
 **this**.**timeout** = timeout;  
 **this**.**timeUnit** = timeUnit;  
 **this**.**taskQueue** = **new** BlockingQueue<>(queueCapcity);  
 **this**.**rejectPolicy** = rejectPolicy;  
 }  
  
 *// 执行任务* **public void** execute(Runnable task) {  
 *// 当任务数没有超过 coreSize 时，直接交给 worker 对象执行  
 // 如果任务数超过 coreSize 时，加入任务队列暂存* **synchronized** (**workers**) {  
 **if**(**workers**.size() < **coreSize**) {  
 Worker worker = **new** Worker(task);  
 ***log***.debug(**"新增 worker{}, {}"**, worker, task);  
 **workers**.add(worker);  
 worker.start();  
 } **else** {  
*// taskQueue.put(task);  
 // 1) 死等  
 // 2) 带超时等待  
 // 3) 让调用者放弃任务执行  
 // 4) 让调用者抛出异常  
 // 5) 让调用者自己执行任务* **taskQueue**.tryPut(**rejectPolicy**, task);  
 }  
 }  
 }  
  
 *// 线程对象* **class** Worker **extends** Thread{  
 **private** Runnable **task**;  
  
 **public** Worker(Runnable task) {  
 **this**.**task** = task;  
 }  
  
 @Override  
 **public void** run() {  
 *// 执行任务  
 // 1) 当 task 不为空，执行任务  
 // 2) 当 task 执行完毕，再接着从任务队列获取任务并执行  
// while(task != null || (task = taskQueue.take()) != null) {* **while**(**task** != **null** || (**task** = **taskQueue**.poll(**timeout**, **timeUnit**)) != **null**) {  
 **try** {  
 ***log***.debug(**"正在执行...{}"**, **task**);  
 **task**.run();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **task** = **null**;  
 }  
 }  
 **synchronized** (**workers**) {  
 ***log***.debug(**"worker 被移除{}"**, **this**);  
 **workers**.remove(**this**);  
 }  
 }  
 }  
}

步骤4：测试

@Slf4j(topic = **"c.TestPool"**)  
**public class** TestPool {  
 **public static void** main(String[] args) {  
 ThreadPool threadPool = **new** ThreadPool(1,  
 1000, TimeUnit.***MILLISECONDS***, 1, (queue, task)->{  
 *// 1. 死等  
// queue.put(task);  
 // 2) 带超时等待  
// queue.offer(task, 1500, TimeUnit.MILLISECONDS);  
 // 3) 让调用者放弃任务执行  
// log.debug("放弃{}", task);  
 // 4) 让调用者抛出异常  
// throw new RuntimeException("任务执行失败 " + task);  
 // 5) 让调用者自己执行任务* task.run();  
 });  
 **for** (**int** i = 0; i < 4; i++) {  
 **int** j = i;  
 threadPool.execute(() -> {  
 **try** {  
 Thread.*sleep*(1000L);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"{}"**, j);  
 });  
 }  
 }  
}

### Thread Pool Executor



#### 线程池状态

ThreadPoolExecutor使用int的高3位来表示线程池状态，低29位表示线程^量

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态名 | 3位 | 接收新任务 | 处理阻塞队列任务 | 说明 |
| RUNNING | 111 | Y | Y |  |
| SHUTDOWN | 000 | N | Y | 不会接收新偿,但会岫且塞队列剩余 |
| STOP | 001 | N | N | 会中断正国丸行的偿,并抛弃阻塞队列 |
| TIDYING | 010 | - | - | 任务全执行完毕，活动爨为0即将进入 终结 |
| TERMINATED | 011 | - | - | 终结状态 |

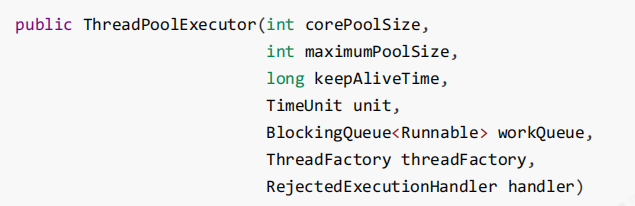
从数字上比较，TERMINATED > TIDYING > STOP > SHUTDOWN > RUNNING

这些信息存储在一个原子变量 ctl 中，目的是将线程池状态与线程个数合二为一，这样就可以用一次 cas 原子操作

进行赋值

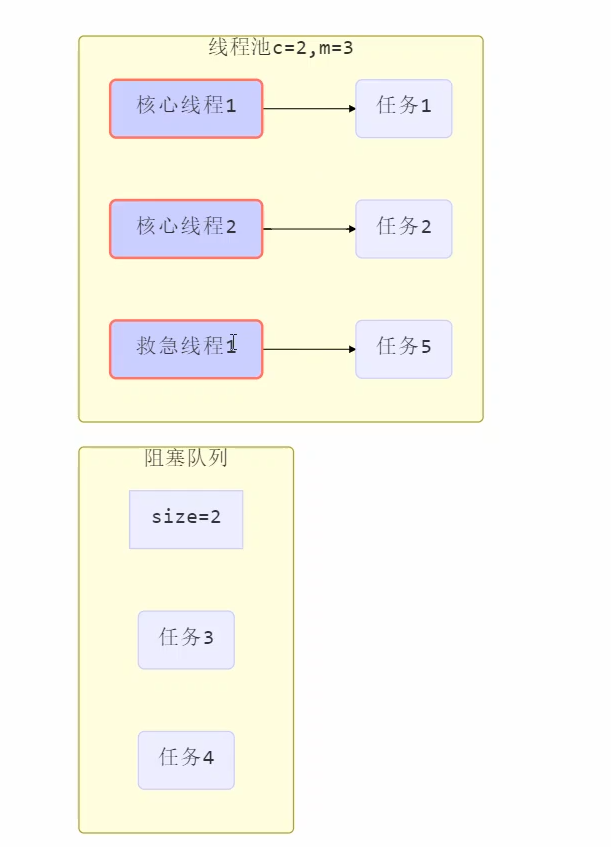
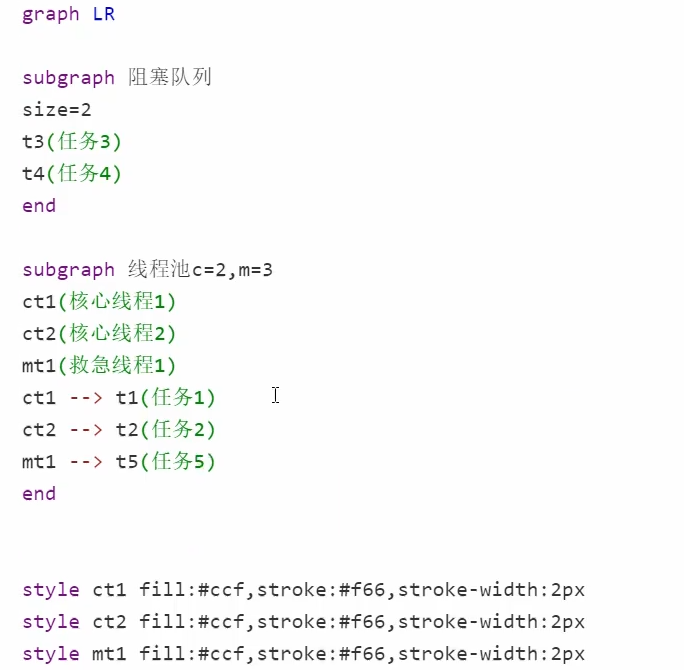


#### 构造方法

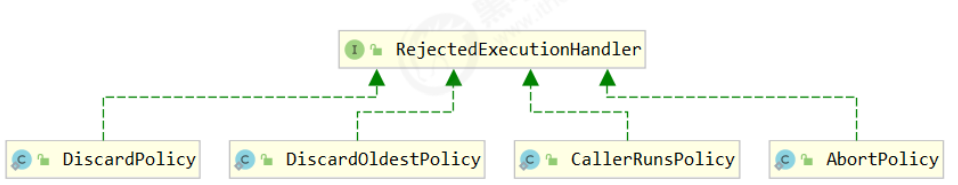


* corePoolSize 核心线程数目 (最多保留的线程数)
* maximumPoolSize 最大线程数目
* keepAliveTime 生存时间 - 针对救急线程
* unit 时间单位 - 针对救急线程
* workQueue 阻塞队列
* threadFactory 线程工厂 - 可以为线程创建时起个好名字
* handler 拒绝策略

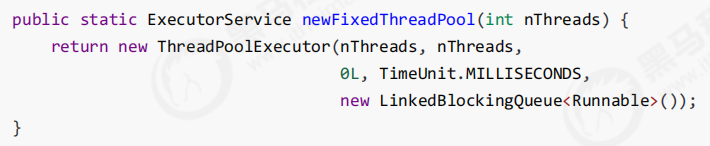
#### 工作方式



* 线程池中刚开始没有线程，当一个任务提交给线程池后，线程池会创建一个新线程来执行任务。
* 当线程数达到 corePoolSize 并没有线程空闲，这时再加入任务，新加的任务会被加入workQueue 队列排队，直到有空闲的线程。
* 如果队列选择了有界队列，那么任务超过了队列大小时，会创建 maximumPoolSize - corePoolSize 数目的线程来救急。
* 如果线程到达 maximumPoolSize 仍然有新任务这时会执行拒绝策略。拒绝策略jdk提供了4种实现，其它著名框架也提供了实现
* AbortPolicy 让调用者抛出 RejectedExecutionException 异常，这是默认策略
* CallerRunsPolicy 让调用者运行任务
* DiscardPolicy 放弃本次任务
* DiscardOldestPolicy 放弃队列中最早的任务，本任务取而代之
* Dubbo 的实现，在抛出 RejectedExecutionException 异常之前会记录日志，并 dump 线程栈信息，方便定位问题
* Netty 的实现，是创建一个新线程来执行任务
* ActiveMQ 的实现，带超时等待（60s）尝试放入队列，类似我们之前自定义的拒绝策略
* PinPoint 的实现，它使用了一个拒绝策略链，会逐一尝试策略链中每种拒绝策略
* 当高峰过去后，超过corePoolSize 的救急线程如果一段时间没有任务做，需要结束节省资源，这个时间由keepAliveTime 和 unit 来控制。



#### newFixedThreadPool



特点：

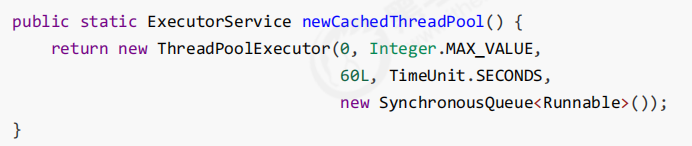
核心线程数 == 最大线程数（没有救急线程被创建），因此也无需超时时间

阻塞队列是无界的，可以放任意数量的任务

评价：

适用于任务量已知，相对耗时的任务

#### newCachedThreadPool

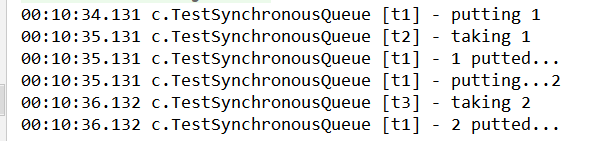


特点：

* 核心线程数是 0，最大线程数是Integer.MAX\_VALUE，救急线程的空闲生存时间是 60s，意味着：
* 全部都是救急线程（60s后可以回收）
* 救急线程可以无限创建
* 队列采用了 SynchronousQueue 实现特点是，它没有容量，没有线程来取是放不进去的（一手交钱、一手交货）

@Slf4j(topic = **"c.TestSynchronousQueue"**)  
**public class** TestSynchronousQueue {  
  
 **public static void** main(String[] args) {  
 SynchronousQueue<Integer> integers = **new** SynchronousQueue<>();  
 **new** Thread(() -> {  
 **try** {  
 ***log***.debug(**"putting {} "**, 1);  
 integers.put(1);  
 ***log***.debug(**"{} putted..."**, 1);  
  
 ***log***.debug(**"putting...{} "**, 2);  
 integers.put(2);  
 ***log***.debug(**"{} putted..."**, 2);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"t1"**).start();  
  
 *sleep*(1);  
  
 **new** Thread(() -> {  
 **try** {  
 ***log***.debug(**"taking {}"**, 1);  
 integers.take();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"t2"**).start();  
  
 *sleep*(1);  
  
 **new** Thread(() -> {  
 **try** {  
 ***log***.debug(**"taking {}"**, 2);  
 integers.take();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"t3"**).start();  
 }  
}

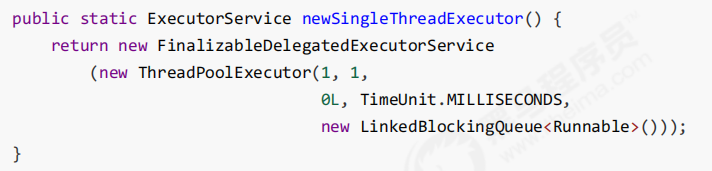
输出：



评价：

整个线程池表现为线程数会根据任务量不断增长，没有上限，当任务执行完毕，空闲分钟后释放线程。适合任务数比较密集，但每个任务执行时间较短的情况。

#### newSingleThreadExecutor



**使用场景：**

希望多个任务排队执行。线程数固定为 1，任务数多于 1 时，会放入无界队列排队。任务执行完毕，这唯一的线程也不会被释放。

**区别：**

* 自己创建一个单线程串行执行任务，如果任务执行失败而终止那么没有任何补救措施，而线程池还会新建一个线程，保证池的正常工作
* Executors.newSingleThreadExecutor() 线程个数始终为1，不能修改
* FinalizableDelegatedExecutorService 应用的是装饰器模式，只对外暴露了 ExecutorService 接口，因此不能调用 ThreadPoolExecutor 中特有的方法
* Executors.newFixedThreadPool(1) 初始时为1，以后还可以修改
* 对外暴露的是 ThreadPoolExecutor 对象，可以强转后调用 setCorePoolSize 等方法进行修改

@Slf4j(topic = **"c.TestExecutors"**)  
**public class** TestExecutors {  
 **public static void** main(String[] args) **throws** InterruptedException {  
 *test2*();  
 }  
  
 **public static void** test2() {  
 ExecutorService pool = Executors.*newSingleThreadExecutor*();  
 pool.execute(() -> {  
 ***log***.debug(**"1"**);  
 **int** i = 1 / 0;  
 });  
  
 pool.execute(() -> {  
 ***log***.debug(**"2"**);  
 });  
  
 pool.execute(() -> {  
 ***log***.debug(**"3"**);  
 });  
 }  
  
 **private static void** test1() {  
  
 ExecutorService pool = Executors.*newFixedThreadPool*(2, **new** ThreadFactory() {  
 **private** AtomicInteger **t** = **new** AtomicInteger(1);  
  
 @Override  
 **public** Thread newThread(Runnable r) {  
 **return new** Thread(r, **"mypool\_t"** + **t**.getAndIncrement());  
 }  
 });  
  
 pool.execute(() -> {  
 ***log***.debug(**"1"**);  
 });  
  
 pool.execute(() -> {  
 ***log***.debug(**"2"**);  
 });  
  
 pool.execute(() -> {  
 ***log***.debug(**"3"**);  
 });  
 }  
}

#### 提交任务



@Slf4j(topic = **"c.TestSubmit"**)  
**public class** TestSubmit {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 ExecutorService pool = Executors.*newFixedThreadPool*(1);  
  
 *method3*(pool);  
 }  
  
 **private static void** method3(ExecutorService pool) **throws** InterruptedException, ExecutionException {  
  
 *// invokeAny：提交tasks中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消* String result = pool.invokeAny(Arrays.*asList*(  
 () -> {  
 ***log***.debug(**"begin 1"**);  
 Thread.*sleep*(1000);  
 ***log***.debug(**"end 1"**);  
 **return "1"**;  
 },  
 () -> {  
 ***log***.debug(**"begin 2"**);  
 Thread.*sleep*(500);  
 ***log***.debug(**"end 2"**);  
 **return "2"**;  
 },  
 () -> {  
 ***log***.debug(**"begin 3"**);  
 Thread.*sleep*(2000);  
 ***log***.debug(**"end 3"**);  
 **return "3"**;  
 }  
 ));  
 ***log***.debug(**"{}"**, result);  
 }  
  
 **private static void** method2(ExecutorService pool) **throws** InterruptedException {  
  
 *// invokeAll：提交 tasks 中所有任务* List<Future<String>> futures = pool.invokeAll(Arrays.*asList*(  
 () -> {  
 ***log***.debug(**"begin"**);  
 Thread.*sleep*(1000);  
 **return "1"**;  
 },  
 () -> {  
 ***log***.debug(**"begin"**);  
 Thread.*sleep*(500);  
 **return "2"**;  
 },  
 () -> {  
 ***log***.debug(**"begin"**);  
 Thread.*sleep*(2000);  
 **return "3"**;  
 }  
 ));  
 futures.forEach( f -> {  
 **try** {  
 ***log***.debug(**"{}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
  
 **private static void** method1(ExecutorService pool) **throws** InterruptedException, ExecutionException {  
  
 *// submit：提交任务task，用返回值Future获得任务执行结果* Future<String> future = pool.submit(() -> {  
  
 ***log***.debug(**"running...."**);  
 Thread.*sleep*(1000);  
 **return "ok"**;  
 });  
 *// get()：唤醒主线程* ***log***.debug(future.get());  
 }  
}

#### 关闭线程池

**Shutdown：**

/\*

线程池状态变为 SHUTDOWN

- 不会接收新任务

- 但已提交任务会执行完

- 此方法不会阻塞调用线程的执行

\*/

void shutdown();

public void shutdown() {

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

checkShutdownAccess();

// 修改线程池状态

advanceRunState(SHUTDOWN);

// 仅会打断空闲线程

interruptIdleWorkers();

onShutdown(); // 扩展点 ScheduledThreadPoolExecutor

} finally {

mainLock.unlock();

}

// 尝试终结(没有运行的线程可以立刻终结，如果还有运行的线程也不会等)

tryTerminate();

}

**shutdownNow：**

/\*

线程池状态变为 STOP

- 不会接收新任务

- 会将队列中的任务返回

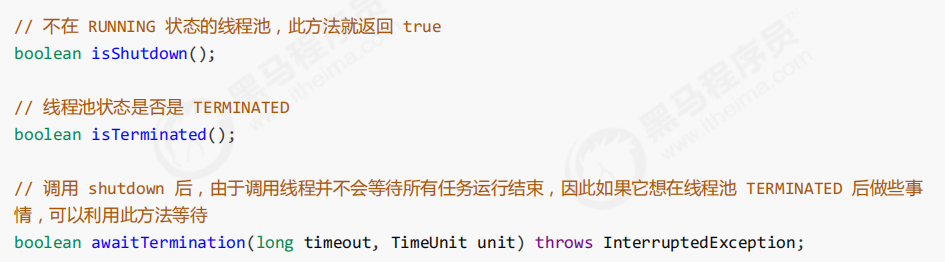
- 并用 interrupt 的方式中断正在执行的任务

\*/

List<Runnable> shutdownNow();



**其它方法**



@Slf4j(topic = **"c.TestShutDown"**)  
**public class** TestShutDown {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 ExecutorService pool = Executors.*newFixedThreadPool*(2);  
  
 Future<Integer> result1 = pool.submit(() -> {  
 ***log***.debug(**"task 1 running..."**);  
 Thread.*sleep*(1000);  
 ***log***.debug(**"task 1 finish..."**);  
 **return** 1;  
 });  
  
 Future<Integer> result2 = pool.submit(() -> {  
 ***log***.debug(**"task 2 running..."**);  
 Thread.*sleep*(1000);  
 ***log***.debug(**"task 2 finish..."**);  
 **return** 2;  
 });  
  
 Future<Integer> result3 = pool.submit(() -> {  
 ***log***.debug(**"task 3 running..."**);  
 Thread.*sleep*(1000);  
 ***log***.debug(**"task 3 finish..."**);  
 **return** 3;  
 });  
  
 ***log***.debug(**"shutdown"**);  
 *// 1、不会接收新任务result4 2、已提交任务会执行完 3、此方法不会阻塞调用线程后续的执行  
// pool.shutdown();  
// pool.awaitTermination(3, TimeUnit.SECONDS);  
  
 // 1、不会接收新任务result4 2、会将队列中的任务result3返回 3、并用interrupt的方式中断正在执行的任务*result1、result2List<Runnable> runnables = pool.shutdownNow();  
 ***log***.debug(**"other.... {}"** , runnables);  
  
 Future<Integer> result4 = pool.submit(() -> {  
  
 ***log***.debug(**"task 4 running..."**);  
 Thread.*sleep*(1000);  
 ***log***.debug(**"task 4 finish..."**);  
 **return** 4;  
 });  
 }  
}

#### \* 模式之Worker Thread

##### 1. 定义

让有限的工作线程（Worker Thread）来轮流异步处理无限多的任务。也可以将其归类为分工模式，它的典型实现就是线程池，也体现了经典设计模式中的享元模式。

例如，海底捞的服务员（线程），轮流处理每位客人的点餐（任务），如果为每位客人都配一名专属的服务员，那么成本就太高了（对比另一种多线程设计模式：Thread-Per-Message）

注意，不同任务类型应该使用不同的线程池，这样能够避免饥饿，并能提升效率

例如，如果一个餐馆的工人既要招呼客人（任务类型A），又要到后厨做菜（任务类型B）显然效率不咋地，分成服务员（线程池A）与厨师（线程池B）更为合理，当然你能想到更细致的分工

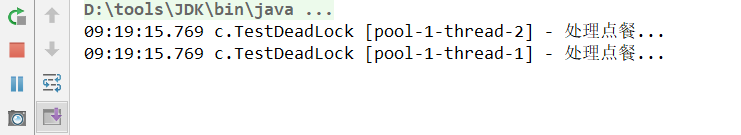
##### 2. 饥饿现象

固定大小线程池会有饥饿现象：

* 两个工人是同一个线程池中的两个线程
* 他们要做的事情是：为客人点餐和到后厨做菜，这是两个阶段的工作
* 客人点餐：必须先点完餐，等菜做好，上菜，在此期间处理点餐的工人必须等待
* 后厨做菜：没啥说的，做就是了
* 比如工人A 处理了点餐任务，接下来它要等着 工人B 把菜做好，然后上菜，他俩也配合的蛮好
* 但现在同时来了两个客人，这个时候工人A 和工人B 都去处理点餐了，这时没人做饭了，饥饿

问题描述：

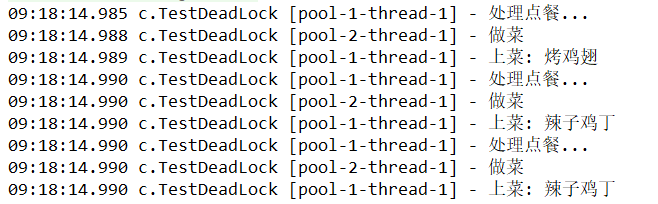
@Slf4j(topic = **"c.TestDeadLock"**)  
**public class** TestStarvation {  
  
 **static final** List<String> ***MENU*** = Arrays.*asList*(**"地三鲜"**, **"宫保鸡丁"**, **"辣子鸡丁"**, **"烤鸡翅"**);  
 **static** Random *RANDOM* = **new** Random();  
 **static** String cooking() {  
 **return *MENU***.get(*RANDOM*.nextInt(***MENU***.size()));  
 }  
 **public static void** main(String[] args) {  
  
 *method1*();  
 }  
 *// 出现饥饿现象，两个点餐线程就把线程占满* **private static void** method1() {  
 ExecutorService pool = Executors.*newFixedThreadPool*(2);  
  
 *// 两个任务：一个点餐、一个做菜* pool.execute(() -> {  
 ***log***.debug(**"处理点餐..."**);  
 Future<String> f = pool.submit(() -> {  
 ***log***.debug(**"做菜"**);  
 **return** *cooking*();  
 });  
 **try** {  
 ***log***.debug(**"上菜: {}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 pool.execute(() -> {  
 ***log***.debug(**"处理点餐..."**);  
 Future<String> f = pool.submit(() -> {  
 ***log***.debug(**"做菜"**);  
 **return** *cooking*();  
 });  
 **try** {  
 ***log***.debug(**"上菜: {}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
}



**解决方法可以增加线程池的大小**，不过不是根本解决方案，还是前面提到的，不同的任务类型，采用不同的线程池，例如：

@Slf4j(topic = **"c.TestDeadLock"**)  
**public class** TestStarvation {  
  
 **static final** List<String> ***MENU*** = Arrays.*asList*(**"地三鲜"**, **"宫保鸡丁"**, **"辣子鸡丁"**, **"烤鸡翅"**);  
 **static** Random *RANDOM* = **new** Random();  
 **static** String cooking() {  
 **return *MENU***.get(*RANDOM*.nextInt(***MENU***.size()));  
 }  
 **public static void** main(String[] args) {  
  
 *method2*();  
 }  
 **private static void** method2() {  
  
 *// 分别定义两个线程池，分别处理不同的线程* ExecutorService waiterPool = Executors.*newFixedThreadPool*(1);  
 ExecutorService cookPool = Executors.*newFixedThreadPool*(1);

*// 定义三个线程*  
 waiterPool.execute(() -> {  
 ***log***.debug(**"处理点餐..."**);  
 Future<String> f = cookPool.submit(() -> {  
 ***log***.debug(**"做菜"**);  
 **return** *cooking*();  
 });  
 **try** {  
 ***log***.debug(**"上菜: {}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 waiterPool.execute(() -> {  
 ***log***.debug(**"处理点餐..."**);  
 Future<String> f = cookPool.submit(() -> {  
 ***log***.debug(**"做菜"**);  
 **return** *cooking*();  
 });  
 **try** {  
 ***log***.debug(**"上菜: {}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 waiterPool.execute(() -> {  
 ***log***.debug(**"处理点餐..."**);  
 Future<String> f = cookPool.submit(() -> {  
 ***log***.debug(**"做菜"**);  
 **return** *cooking*();  
 });  
 **try** {  
 ***log***.debug(**"上菜: {}"**, f.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
}



##### 3. 创建多少线程池合适

* 过小会导致程序不能充分地利用系统资源、容易导致饥饿
* 过大会导致更多的线程上下文切换，占用更多内存

###### 3.1 CPU密集型运算

通常采用 **cpu 核数 + 1** 能够实现最优的 CPU 利用率，+1 是保证当线程由于页缺失故障（操作系统）或其它原因导致暂停时，额外的这个线程就能顶上去，保证 CPU 时钟周期不被浪费

###### 3.2 I/O密集型运算

CPU 不总是处于繁忙状态，例如，当你执行业务计算时，这时候会使用 CPU 资源，但当你执行 I/O 操作时、远程RPC 调用时，包括进行数据库操作时，这时候 CPU 就闲下来了，你可以利用多线程提高它的利用率。

经验公式如下：

**线程数 = 核数 \* 期望 CPU 利用率 \* 总时间(CPU计算时间+等待时间) / CPU 计算时间**

例如 4 核 CPU 计算时间是 50% ，其它等待时间是 50%，期望 cpu 被 100% 利用，套用公式

4 \* 100% \* 100% / 50% = 8

例如 4 核 CPU 计算时间是 10% ，其它等待时间是 90%，期望 cpu 被 100% 利用，套用公式

4 \* 100% \* 100% / 10% = 40

##### 4. 自定义线程池

#### 任务调度线程池

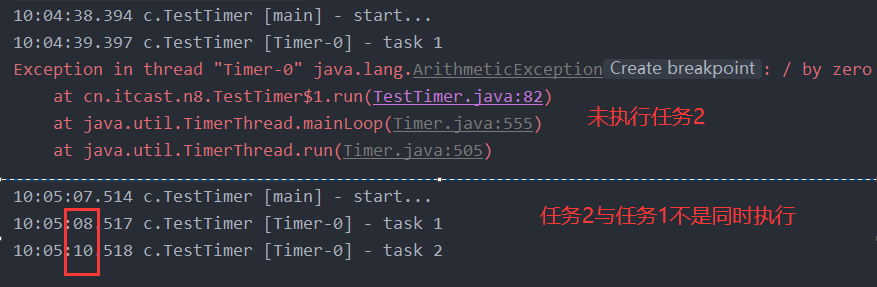
在『任务调度线程池』功能加入之前，可以使用 java.util.Timer 来实现定时功能，Timer 的优点在于简单易用，但由于所有任务都是由同一个线程来调度，因此所有任务都是串行执行的，同一时间只能有一个任务在执行，前一个任务的延迟或异常都将会影响到之后的任务。

##### 使用Timer存在的问题

@Slf4j(topic = **"c.TestTimer"**)  
**public class** TestTimer {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 Timer timer = **new** Timer();  
 TimerTask task1 = **new** TimerTask() {  
 @Override  
 **public void** run() {  
 ***log***.debug(**"task 1"**);  
*// int i = 1/0;  
 sleep*(2);  
 }  
 };  
 TimerTask task2 = **new** TimerTask() {  
 @Override  
 **public void** run() {  
 ***log***.debug(**"task 2"**);  
 }  
 };  
 // 使用 timer 添加两个任务，希望它们都在 1s 后执行

// 但由于 timer 内只有一个线程来顺序执行队列中的任务，因此『任务1』的延时，影响了『任务2』的执行

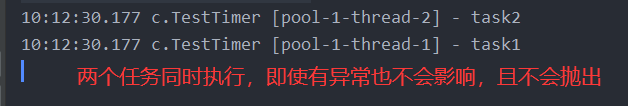
// 同时若任务1存在异常，线程2也不会执行  
 ***log***.debug(**"start..."**);  
 timer.schedule(task1, 1000);  
 timer.schedule(task2, 1000);  
}



##### 使用ScheduledExecutorService

@Slf4j(topic = **"c.TestTimer"**)  
**public class** TestTimer {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 *// 延时执行任务* ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(2);

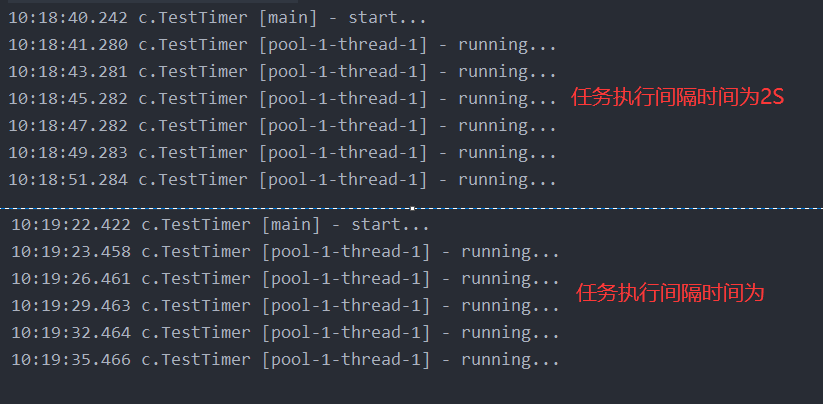
*// 添加两个任务，希望它们都在 1s 后执行*  
 pool.schedule(() -> {  
 ***log***.debug(**"task1"**);  
*// int i = 1 / 0;  
 sleep*(2);  
 }, 1, TimeUnit.***SECONDS***);  
  
 pool.schedule(() -> {  
 ***log***.debug(**"task2"**);  
 }, 1, TimeUnit.***SECONDS***);  
}



##### scheduleAtFixedRate和scheduleWithFixedDelay

@Slf4j(topic = **"c.TestTimer"**)  
**public class** TestTimer {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(1);  
 ***log***.debug(**"start..."**);  
 *// scheduleAtFixedRate：以固定的速率执行任务 参数：初始时间（主线程启动后间隔时间开始执行）、时间间隔、单位  
 // delay是以上一次任务耗时计算，此处的间隔为2S* pool.scheduleAtFixedRate(() -> {  
 ***log***.debug(**"running..."**);

*// 任务执行时间超过了间隔时间*  
 *sleep*(2);  
 }, 1, 1, TimeUnit.***SECONDS***);  
  
 *// delay是从上一次任务结束后开始计算，此处的间隔为3S* pool.scheduleWithFixedDelay(() -> {  
 ***log***.debug(**"running..."**);  
 *sleep*(2);  
 }, 1, 1, TimeUnit.***SECONDS***);  
}



scheduleAtFixedRate输出分析：一开始，延时 1s，接下来，由于任务执行时间 > 间隔时间，间隔被『撑』到了 2s

scheduleWithFixedDelay输出分析：一开始，延时 1s，scheduleWithFixedDelay 的间隔是 上一个任务结束 <-> 延时 <-> 下一个任务开始 所以间隔都是 3s

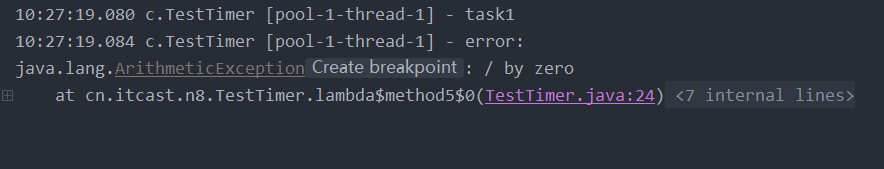
**评价：**

整个线程池表现为：线程数固定，任务数多于线程数时，会放入无界队列排队。任务执行完毕，这些线程也不会被释放。用来执行延迟或反复执行的任务

#### 正确处理执行任务异常

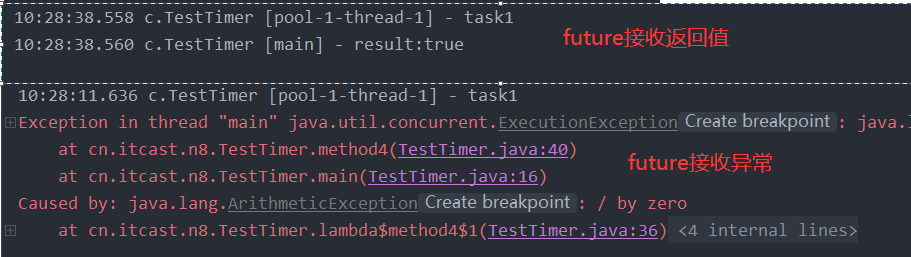
##### 方法1：主动捉异常

@Slf4j(topic = **"c.TestTimer"**)  
**public class** TestTimer {  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(1);  
 pool.schedule(() -> {  
 **try** {  
 ***log***.debug(**"task1"**);  
 **int** i = 1 / 0;  
 } **catch** (Exception e) {  
 ***log***.error(**"error:"**, e);  
 }  
 }, 1, TimeUnit.***SECONDS***);  
}



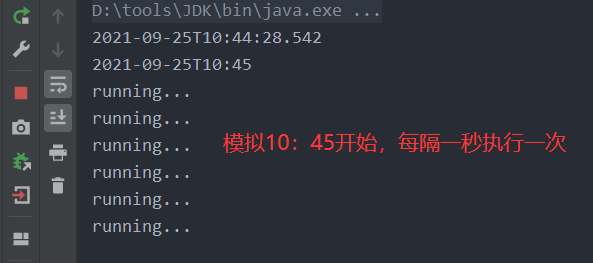
##### 方法2：使用Future

@Slf4j(topic = **"c.TestTimer"**)  
**public class** TestTimer {  
  
 **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
  
 ExecutorService pool1 = Executors.*newFixedThreadPool*(1);  
 Future<Boolean> future = pool1.submit(() -> {  
 ***log***.debug(**"task1"**);  
 **int** i = 1 / 0;  
 **return true**;  
 });  
 ***log***.debug(**"result:{}"**,future.get());  
}



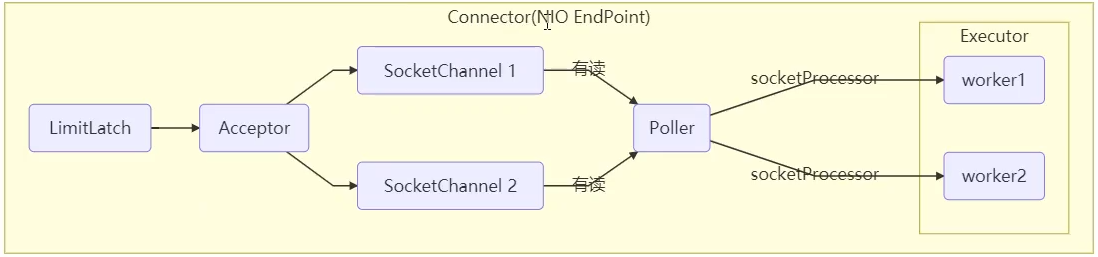
#### \* 应用之定时任务





#### Tomcat线程池

Tomcat用到的线程池：

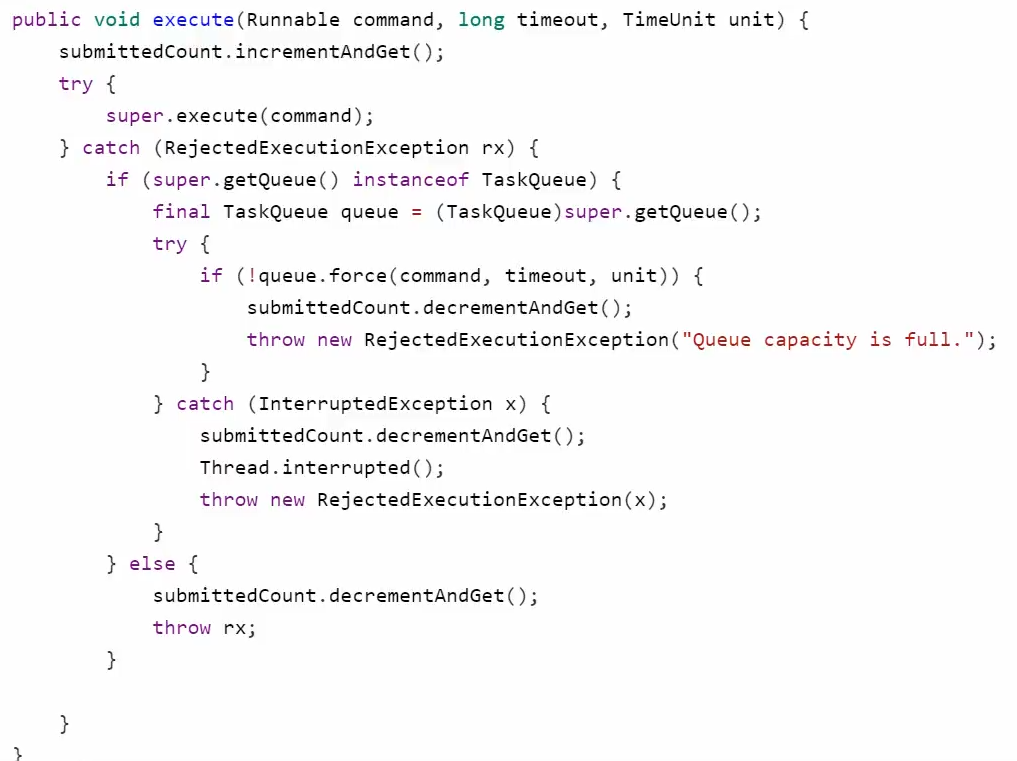


* LimitLatch 用来限流，可以控制最大连接个数，类似 J.U.C 中的 Semaphore 后面再讲
* Acceptor 只负责【接收新的 socket 连接】
* Poller 只负责监听 socket channel 是否有【可读的 I/O 事件】
* 一旦可读，封装一个任务对象（socketProcessor），提交给 Executor 线程池处理
* Executor 线程池中的工作线程最终负责【处理请求】

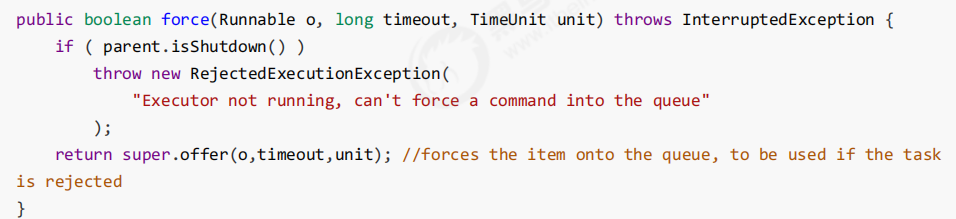
Tomcat 线程池扩展了 ThreadPoolExecutor，行为稍有不同

* 如果总线程数达到 maximumPoolSize
* 这时不会立刻抛 RejectedExecutionException 异常
* 而是再次尝试将任务放入队列，如果还失败，才抛出 RejectedExecutionException 异常

源码 tomcat-7.0.42



TaskQueue.java

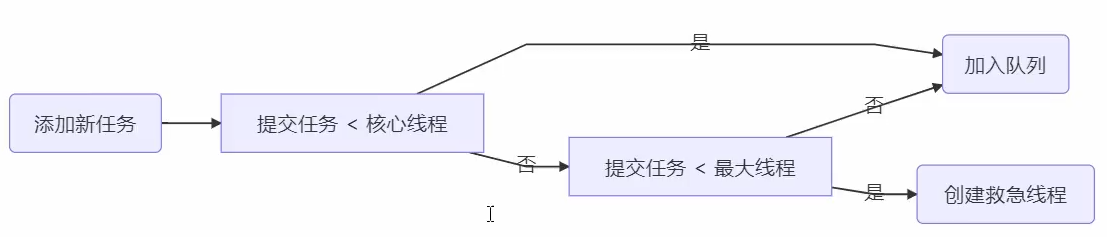


Connector配置：



Executor线程配置：





### Fork/Join线程池

#### 概念

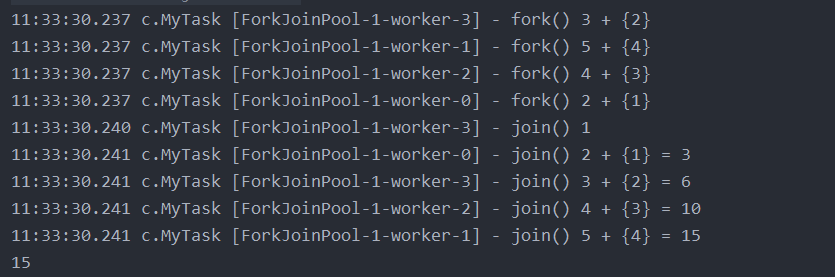
* Fork/Join是 JDK 1.7 加入的新的线程池实现，它体现的是一种分治思想，适用于能够进行任务拆分的 cpu 密集型运算
* 所谓的任务拆分，是将一个大任务拆分为算法上相同的小任务，直至不能拆分可以直接求解。跟递归相关的一些计算，如归并排序、斐波那契数列、都可以用分治思想进行求解
* Fork/Join 在分治的基础上加入了多线程，可以把每个任务的分解和合并交给不同的线程来完成，进一步提升了运算效率
* Fork/Join 默认会创建与 cpu 核心数大小相同的线程池

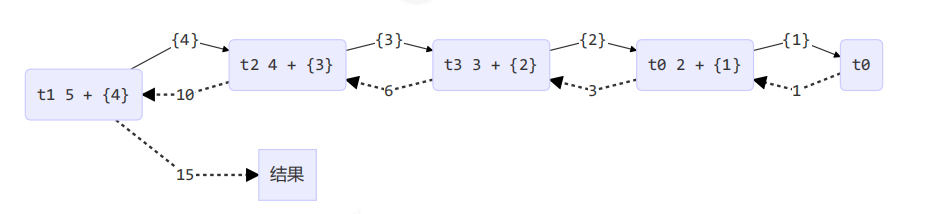
#### 使用

提交给 Fork/Join 线程池的任务需要继承 RecursiveTask（有返回值）或 RecursiveAction（没有返回值），

例如下面定义了一个对 1~n 之间的整数求和的任务:

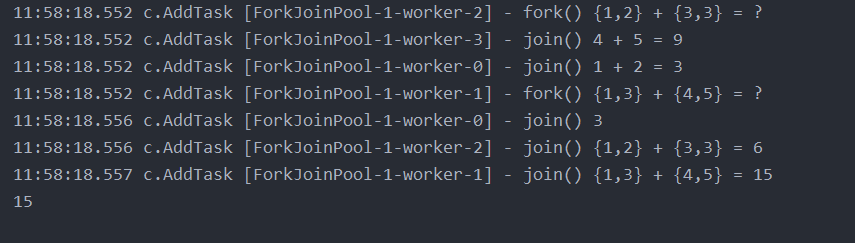
@Slf4j(topic = **"c.TestForkJoin2"**)  
**public class** TestForkJoin2 {  
  
 **public static void** main(String[] args) {  
  
 *// 4个线程，若不传，则以CPU核心为线程数* ForkJoinPool pool = **new** ForkJoinPool(4);  
 System.***out***.println(pool.invoke(**new** MyTask(5)));  
  
 *// 任务拆分：  
 // new MyTask(5)：5 + new MyTask(4)  
 // new MyTask(4)：4 + new MyTask(3)  
 // new MyTask(3)：3 + new MyTask(2)  
 // new MyTask(2)：2 + new MyTask(1)* }  
}  
  
*// 1~n 之间整数的和*@Slf4j(topic = **"c.MyTask"**)  
**class** MyTask **extends** RecursiveTask<Integer> {  
  
 **private int n**;  
  
 **public** MyTask(**int** n) {  
 **this**.**n** = n;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "{"** + **n** + **'}'**;  
 }  
  
 @Override  
 **protected** Integer compute() {  
  
 *// 如果n为 1，可以求得结果了* **if** (**n** == 1) {  
 ***log***.debug(**"join() {}"**, **n**);  
 **return n**;  
 }  
  
 *// 将任务进行拆分(fork)  
// AddTask1 t1 = new AddTask1(n - 1);* MyTask t1 = **new** MyTask(**n** - 1);  
 *// 让一个线程去执行此任务* t1.fork();  
 ***log***.debug(**"fork() {} + {}"**, **n**, t1);  
  
 *// 合并(join)结果* **int** result = **n** + t1.join();  
 ***log***.debug(**"join() {} + {} = {}"**, **n**, t1, result);  
 **return** result;  
 }  
}

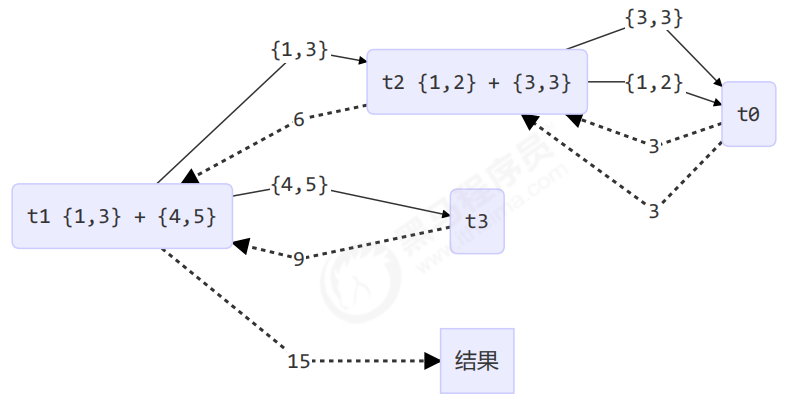




#### 改进

**package** cn.itcast.n8;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
**import** java.util.concurrent.ForkJoinPool;  
**import** java.util.concurrent.RecursiveTask;  
  
**public class** TestForkJoin {  
  
 **public static void** main(String[] args) {  
 ForkJoinPool pool = **new** ForkJoinPool(4);  
*// System.out.println(pool.invoke(new AddTask1(5)));* System.***out***.println(pool.invoke(**new** AddTask3(1, 5)));  
 }  
}  
  
@Slf4j(topic = **"c.AddTask"**)  
**class** AddTask1 **extends** RecursiveTask<Integer> {  
  
 **int n**;  
  
 **public** AddTask1(**int** n) {  
 **this**.**n** = n;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "{"** + **n** + **'}'**;  
 }  
  
 @Override  
 **protected** Integer compute() {  
 **if** (**n** == 1) {  
 ***log***.debug(**"join() {}"**, **n**);  
 **return n**;  
 }  
 AddTask1 t1 = **new** AddTask1(**n** - 1);  
  
 t1.fork();  
 ***log***.debug(**"fork() {} + {}"**, **n**, t1);  
 **int** result = **n** + t1.join();  
 ***log***.debug(**"join() {} + {} = {}"**, **n**, t1, result);  
 **return** result;  
 }  
}  
  
@Slf4j(topic = **"c.AddTask"**)  
**class** AddTask2 **extends** RecursiveTask<Integer> {  
  
 **int begin**;  
 **int end**;  
  
 **public** AddTask2(**int** begin, **int** end) {  
 **this**.**begin** = begin;  
 **this**.**end** = end;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "{"** + **begin** + **","** + **end** + **'}'**;  
 }  
  
 @Override  
 **protected** Integer compute() {  
 **if** (**begin** == **end**) {  
 ***log***.debug(**"join() {}"**, **begin**);  
 **return begin**;  
 }  
 **if** (**end** - **begin** == 1) {  
 ***log***.debug(**"join() {} + {} = {}"**, **begin**, **end**, **end** + **begin**);  
 **return end** + **begin**;  
 }  
 **int** mid = (**end** + **begin**) / 2;  
  
 AddTask2 t1 = **new** AddTask2(**begin**, mid - 1);  
 t1.fork();  
 AddTask2 t2 = **new** AddTask2(mid + 1, **end**);  
 t2.fork();  
 ***log***.debug(**"fork() {} + {} + {} = ?"**, mid, t1, t2);  
  
 **int** result = mid + t1.join() + t2.join();  
 ***log***.debug(**"join() {} + {} + {} = {}"**, mid, t1, t2, result);  
 **return** result;  
 }  
}  
  
@Slf4j(topic = **"c.AddTask"**)  
**class** AddTask3 **extends** RecursiveTask<Integer> {  
  
 **int begin**;  
 **int end**;  
  
 **public** AddTask3(**int** begin, **int** end) {  
 **this**.**begin** = begin;  
 **this**.**end** = end;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "{"** + **begin** + **","** + **end** + **'}'**;  
 }  
  
 @Override  
 **protected** Integer compute() {  
 **if** (**begin** == **end**) {  
 ***log***.debug(**"join() {}"**, **begin**);  
 **return begin**;  
 }  
 **if** (**end** - **begin** == 1) {  
 ***log***.debug(**"join() {} + {} = {}"**, **begin**, **end**, **end** + **begin**);  
 **return end** + **begin**;  
 }  
 **int** mid = (**end** + **begin**) / 2;  
  
 AddTask3 t1 = **new** AddTask3(**begin**, mid);  
 t1.fork();  
 AddTask3 t2 = **new** AddTask3(mid + 1, **end**);  
 t2.fork();  
 ***log***.debug(**"fork() {} + {} = ?"**, t1, t2);  
  
 **int** result = t1.join() + t2.join();  
 ***log***.debug(**"join() {} + {} = {}"**, t1, t2, result);  
 **return** result;  
 }  
}





## 二、J.U.C

### AQS原理

#### 1.1 概述

全称是AbstractQueuedSynchronizer，是阻塞式锁和相关的同步器工具的框架

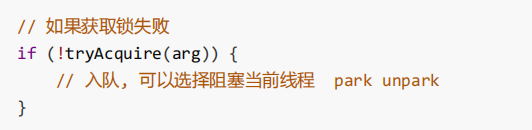
**特点：**

* 用 state 属性来表示资源的状态（分独占模式和共享模式），子类需要定义如何维护这个状态，控制如何获取锁和释放锁
* getState - 获取 state 状态
* setState - 设置 state 状态
* compareAndSetState - cas 机制设置 state 状态
* 独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源
* 提供了基于 FIFO 的等待队列，类似于 Monitor 的 EntryList
* 条件变量来实现等待、唤醒机制，支持多个条件变量，类似于 Monitor 的 WaitSet

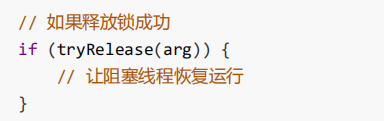
子类主要实现这样一些方法（默认抛出 UnsupportedOperationException）

* tryAcquire
* tryRelease
* tryAcquireShared
* tryReleaseShared
* isHeldExclusively

获取锁的姿势

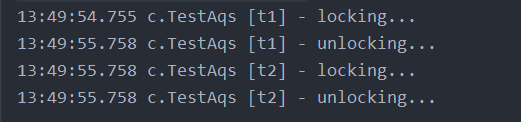


释放锁的姿势



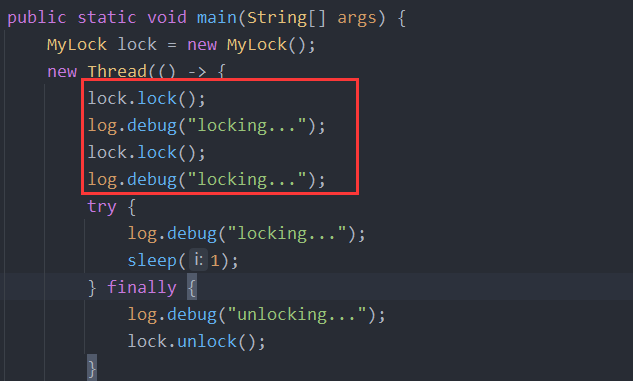
#### 1.2 实现不可重入锁

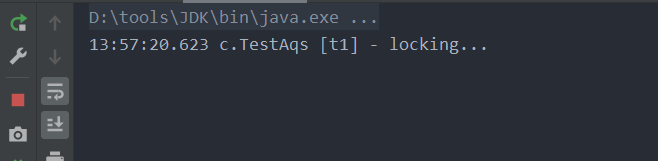
@Slf4j(topic = **"c.TestAqs"**)  
**public class** TestAqs {  
  
 **public static void** main(String[] args) {  
 MyLock lock = **new** MyLock();  
 **new** Thread(() -> {  
 lock.lock();  
 **try** {  
 ***log***.debug(**"locking..."**);  
 *sleep*(1);  
 } **finally** {  
 ***log***.debug(**"unlocking..."**);  
 lock.unlock();  
 }  
 },**"t1"**).start();  
  
 **new** Thread(() -> {  
 lock.lock();  
 **try** {  
 ***log***.debug(**"locking..."**);  
 } **finally** {  
 ***log***.debug(**"unlocking..."**);  
 lock.unlock();  
 }  
 },**"t2"**).start();  
 }  
}  
  
*// 自定义锁（不可重入锁，可以挡住自己）***class** MyLock **implements** Lock {  
  
 *// 独占锁 同步器类* **class** MySync **extends** AbstractQueuedSynchronizer {  
  
 @Override *// 尝试获取锁* **protected boolean** tryAcquire(**int** arg) {  
 **if**(compareAndSetState(0, 1)) {  
 *// 加上了锁，并设置 owner 为当前线程* setExclusiveOwnerThread(Thread.*currentThread*());  
 **return true**;  
 }  
 **return false**;  
 }  
   
 @Override *// 尝试释放锁* **protected boolean** tryRelease(**int** arg) {  
 setExclusiveOwnerThread(**null**);  
 setState(0); *// 解锁* **return true**;  
 }  
  
 @Override *// 是否持有独占锁* **protected boolean** isHeldExclusively() {  
 **return** getState() == 1;  
 }  
  
 *// 创建条件变量* **public** Condition newCondition() {  
 **return new** ConditionObject();  
 }  
 }  
  
 **private** MySync **sync** = **new** MySync();  
  
 @Override *// 加锁（不成功会进入等待队列）* **public void** lock() {  
 **sync**.acquire(1);  
 }  
  
 @Override *// 加锁，可打断* **public void** lockInterruptibly() **throws** InterruptedException {  
 **sync**.acquireInterruptibly(1);  
 }  
  
 @Override *// 尝试加锁（一次）* **public boolean** tryLock() {  
 **return sync**.tryAcquire(1);  
 }  
  
 @Override *// 尝试加锁，带超时* **public boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException {  
 **return sync**.tryAcquireNanos(1, unit.toNanos(time));  
 }  
  
 @Override *// 解锁* **public void** unlock() {  
 **sync**.release(1);  
 }  
  
 @Override *// 创建条件变量* **public** Condition newCondition() {  
 **return sync**.newCondition();  
 }  
}



#### 1.3 不可重入测试

如果改为下面代码，会发现自己也会被挡住（只会打印一次 locking）





### ReentrantLock 原理

### 读写锁

#### ReentrantReadWriteLock

#### StampedLock

### Semaphore

### CountdownLatch

### CyclicBarrier

### 线程安全集合类概述

### ConcurrentHashMap

### BlockingQueue

### ConcurrentLinkedQueue

### CopyOnWriteArrayList