# Java并发编程原理与实战

## 第1节你真的了解并发吗

### 1.1聊聊并发

你真的了解并发吗

多线程和并发：多线程不一定是并发（也有可能是并行，例如多核处理器处理多线程）；并发也不一定是多线程（也有可能是多进程，多进程也是并发）。

多线程和多进程

线程一定快吗

多线程并不一定能够提高性能，反而在有些场景做线程切换耗费的性能要高于线程本身要处理的事情（java垃圾回收的serial收集效率要远高于其他多线程垃圾回收）；误区：迅雷的多线程下载，并不是多线程提高了下载速度，而是web服务器为每一个连接限制一定的带宽，比如一个连接进来，web服务器给一个响应，最高200k。那么如果把一个文件切分成多个，使用多个线程去下载，每个连接200k，等都下载完了进行合并，下载速度就得到了提高，并不是多线程提高了性能，而是web服务器对资源的限制导致了这一个问题，多线程下载的机制就突破的远程服务器的限制，也就使得下载速度得到提高。

学习并发的四个阶段

学习目标

适合人群

荐书

### 1.2学习并发的四个阶段

熟练掌握API，能够完成并发编程

熟读API源码，掌握其原理

理解Java虚拟机的内存模型

操作系统对并发的支持

### 1.3为什么要学习并发编程

1.发挥多处理的强大能力

2.建模的简单性

3.异步事件的简化处理

4.响应更加灵敏的用户界面

1.找工作，面试

2.了解并发编程的原理，提高代码的编写能力

3.解决工作中遇到的并发问题

4.Java进阶

### 1.4并发的缺点

安全性问题

活跃性问题（饥饿）

性能问题

### 1.5线程安全性问题

创建线程 多种方式

多线程运行 卖票

线程安全性问题引入

提出解决方案 synchronized

解决线程安全性问题

Synchronized 原理

…

### 1.6线程与进程

进程：运行中的程序

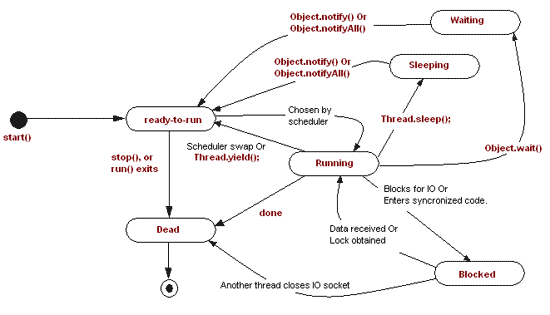
进程是资源分配的基本单位

进程中包含多个线程，线程共享进程的资源

线程是处理器调度的基本单位

## 第2节线程的状态

### 2.1线程转换状态图



### 2.2代码实现线程新建、线程sleep、线程等待

1. 线程新建及sleep

|  |
| --- |
| **public** **class** NewThread **implements** Runnable {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println("自定义线程...");  **try** {  // 线程sleeping状态，超时等待，倒计时启动唤醒，进入线程就绪状态  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **public** **static** **void** main(String[] args) {  // 创建线程，并指定线程任务--线程初始状态  Thread thread = **new** Thread(**new** NewThread());  // 启动线程  thread.start();  **while** (**true**) {  System.***out***.println("主线程....");  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  } |

执行结果：

主线程....

自定义线程...

自定义线程...

主线程....

自定义线程...

主线程....

1. 线程等待

|  |
| --- |
| **public** **class** ThreadWait **implements** Runnable {  @Override  **public** **synchronized** **void** run() {  **while** (**true**) {  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("自定义的线程执行了....");  }  }  **public** **static** **void** main(String[] args) {  ThreadWait n = **new** ThreadWait();  // 初始化状态  Thread thread = **new** Thread(n); // 创建线程,并指定线程任务  thread.start(); // 启动线程  **while** (**true**) {  **synchronized** (n) {  System.***out***.println("主线程执行了...");  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  n.notifyAll();  }  }  }  } |

执行结果：

主线程执行了...

主线程执行了...

主线程执行了...

主线程执行了...

主线程执行了...

自定义的线程执行了....

主线程执行了...

主线程执行了...

如果不使用**synchronized** (n)，则会报以下异常：

主线程执行了...

Exception in thread "main" java.lang.IllegalMonitorStateException

at java.lang.Object.notifyAll(Native Method)

at com.zte.thread.t1threadstatus.ThreadWait.main(ThreadWait.java:33)

## 第3节创建线程的多种方式

### 3.1线程的多种创建方式

* 继承Thread类
* 实现Runnable接口
* 匿名内部类的方式
* 带返回值的线程
* 定时器（quartz）
* 线程池的实现
* Lambda表达式实现
* Spring实现多线程

#### 继承Thread类

线程初始化，中断及源码

|  |
| --- |
| **public** **class** CreateThreadByExtend **extends** Thread {  **public** CreateThreadByExtend(String name) {  // 继承父类的Thread(String name)构造方法，自定义线程名  **super**(name);  }  @Override  **public** **void** run() {  // 判断线程状态，如果中断标志位是true，则不运行该段代码  **while** (!*interrupted*()) {  System.***out***.println(getName() + "线程执行了...");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **public** **static** **void** main(String[] args) {  CreateThreadByExtend o1 = **new** CreateThreadByExtend("线程1：");  CreateThreadByExtend o2 = **new** CreateThreadByExtend("线程2：");  Thread t1 = **new** Thread(o1);  Thread t2 = **new** Thread(o2);  // 线程后台执行，主线程退出后，该线程也退出  /\*  \* t1.setDaemon(true); t2.setDaemon(true);  \*/  t1.start();  t2.start();  // 使线程终止，资源和锁都没有释放，让线程永远的等待下去  /// t1.stop();  // 推荐使用interrupt方式中断线程,并在线程的run方法判断，如果线程未中断则执行，中断则不执行  t1.interrupt();  /\*  \* try { Thread.sleep(1); } catch (InterruptedException e) {  \* e.printStackTrace(); }  \*/  }  } |

中断：

**public** **void** interrupt()：中断线程

**public** **static** **boolean** interrupted()：判断线程中断状态

#### 2）实现Runnable接口

|  |
| --- |
| /\*\*  \* 作为线程任务存在  \* **@author** BYF  \*  \*/  **public** **class** CreateThreadByImplementsRunable **implements** Runnable{  @Override  **public** **void** run() {  **while**(**true**){  System.***out***.println("线程执行了...");  }  }    **public** **static** **void** main(String[] args) {  // 实现Runnable接口的对象，传递给Thread类的run方法，判断target是否等于null，不等于null则执行target的run方法，也就是实现Runnable接口的对象。  Thread t1 = **new** Thread(**new** CreateThreadByImplementsRunable());  t1.start();  }  } |

#### 3）匿名内部类的方式

|  |
| --- |
| **public** **class** CreateThreadByAnonymous {  **public** **static** **void** main(String[] args) {  // 1.使用匿名内部类，相当于集成Thread类，子类的方式来实现  /\*new Thread() {  @Override  public void run() {  System.out.println("线程执行了【匿名内部类，Thread子类】...");  };  }.start();\*/    // 2.通过Runnable接口实现  /\*new Thread(new Runnable(){  @Override  public void run() {  System.out.println("线程执行了【实现Runnable接口方式】...");  }  }).start();\*/    // 3.两个一起用，如果子类覆盖了父类的run方法，只有子类的run才可以执行，多态的特征  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  System.***out***.println("线程执行了【Runnable】...");  }  }){  @Override  **public** **void** run() {  //super.run();  System.***out***.println("线程执行了【子类】...");  }  }.start();  }  } |

#### 4）带返回值的线程

通过实现Callable<T>接口，实现call方法，创建线程需要执行任务。

FutureTask提前完成任务。

|  |
| --- |
| **public** **class** CreateThreadByCallable **implements** Callable<Integer>{  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  CreateThreadByCallable o1 = **new** CreateThreadByCallable();    FutureTask<Integer> task = **new** FutureTask(o1);    Thread t = **new** Thread(task);    t.start();    System.***out***.println("我先干点别的...");    Integer result = task.get();    System.***out***.println("线程执行结果为：" + result);    }  @Override  **public** Integer call() **throws** Exception {  System.***out***.println("正在进行紧张的计算...");  Thread.*sleep*(3000);  **return** 1;  }  } |

执行结果：

我先干点别的...

正在进行紧张的计算...

线程执行结果为：1

#### 定时器

新建Timer的执行计划，重写TimerTask的run方法，设置执行计划的执行开始时间，周期等参数。

|  |
| --- |
| **public** **class** CreateThreadByTimmer {    **public** **static** **void** main(String[] args) {  Timer timer = **new** Timer();  timer.schedule(**new** TimerTask() {    @Override  **public** **void** run() {  // 实现定时任务  System.***out***.println("timertask is run...");  }  }, 0, 1000);  }  } |

执行结果：

timertask is run...

timertask is run...

timertask is run...

timertask is run...

.....

#### 6）线程池的实现

使用jdk自带ExecutorService创建线程池，可以指定线程池中线程数量，也可以使用更为智能的*newCachedThreadPool*方式创建线程.

|  |
| --- |
| **public** **class** CreateThreadByThreadPool {  **public** **static** **void** main(String[] args) {  // 使用newFixedThreadPool，指定线程池中线程数量的方式  /\*ExecutorService threadPool = Executors.newFixedThreadPool(10);  for (int i = 0; i < 100; i++) {  threadPool.execute(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  });  }\*/    // 使用缓存线程池，线程不够用时，智能的增加线程池中线程数量  ExecutorService threadPool = Executors.*newCachedThreadPool*();  **for** (**int** i = 0; i < 1000; i++) {  threadPool.execute(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  }  });  }    threadPool.shutdown();  }  } |

执行结果：

pool-1-thread-2

pool-1-thread-4

pool-1-thread-3

pool-1-thread-1

pool-1-thread-6

pool-1-thread-1

pool-1-thread-5

pool-1-thread-7

pool-1-thread-7

pool-1-thread-4

pool-1-thread-1

pool-1-thread-4

pool-1-thread-2

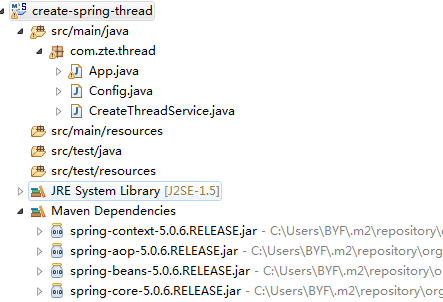
pool-1-thread-4

pool-1-thread-5

pool-1-thread-4

......

#### 7）Spring实现多线程



1.新建maven工程，pom.xml

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.zte</groupId>  <artifactId>create-spring-thread</artifactId>  <version>0.0.1-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>5.0.6.RELEASE</version>  </dependency>  </dependencies>  </project> |

2.使用java配置的风格，新建java配置类Config

|  |
| --- |
| @Configuration  @ComponentScan("com.zte.thread")  @EnableAsync  **public** **class** Config {  } |

3.创建Service类的bean对象

|  |
| --- |
| @Service  **public** **class** CreateThreadService {    @Async  **public** **void** a(){  **while**(**true**){  System.***out***.println("this is a...");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }    @Async  **public** **void** b(){  **while**(**true**){  System.***out***.println("this is b...");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

1. 主方法App

|  |
| --- |
| **public** **class** App {    **public** **static** **void** main(String[] args) {  AnnotationConfigApplicationContext ac = **new** AnnotationConfigApplicationContext(Config.**class**);    CreateThreadService service = ac.getBean(CreateThreadService.**class**);    service.a();    service.b();  }  } |

执行结果：

五月 28, 2018 11:47:10 下午 org.springframework.context.support.AbstractApplicationContext prepareRefresh

信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@14514713: startup date [Mon May 28 23:47:10 CST 2018]; root of context hierarchy

五月 28, 2018 11:47:10 下午 org.springframework.aop.interceptor.AsyncExecutionAspectSupport getDefaultExecutor

信息: No task executor bean found for async processing: no bean of type TaskExecutor and no bean named 'taskExecutor' either

this is b...

this is a...

this is a...

this is b...

this is b...

this is a...

#### Lambda表达式实现

|  |
| --- |
| **public** **class** CreateThreadByLambda {  **public** **static** **void** main(String[] args) {  List<Integer> values = Arrays.*asList*(10,20,30,40);  **int** res = **new** CreateThreadByLambda().addParallel(values);  System.***out***.println("计算结果为：" + res);    /\*// 类型猜测，省去了ArrayList<>中的类型  List<Object> lists = new ArrayList<>();    // Lambda的foreach猜测，省去了Object的类型描述  for(Object i : lists){    }\*/    }    **public** **int** addParallel(List<Integer> values){  System.***out***.println("并行打印values中的元素");  // 并行打印values中的元素  values.parallelStream().forEach(System.***out*** :: println);    System.***out***.println("顺序打印forEachOrdered，parallel流中的元素");  // 顺序打印forEachOrdered，parallel流中的元素  values.parallelStream().forEachOrdered(System.***out*** :: println);  **return** values.parallelStream().mapToInt(i -> i \* 2).sum();  }    **public** **int** add(List<Integer> values){  **return** values.parallelStream().mapToInt(i -> i).sum();  }    **public** **int** addStringInt(List<String> values){  **return** values.parallelStream().mapToInt(i -> Integer.*parseInt*(i)).sum();  }  } |

执行结果：

并行打印values中的元素

30

40

20

10

顺序打印forEachOrdered，parallel流中的元素

10

20

30

40

计算结果为：100

## 第4节线程安全性问题

线程带来的风险：

* 线程安全性问题
* 活跃性问题
* 性能问题

### 4.1自增数字序列生成器

|  |
| --- |
| **public** **class** Sequence {  **private** **int** values;    **public** **int** getNext(){  **return** values++;  }    **public** **static** **void** main(String[] args) {  Sequence s = **new** Sequence();  // 只有一个线程任务在执行，顺序产生数字序列  /\*while(true){  System.out.println(s.getNext());  }\*/    // 定义多个线程，一起执行  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **while**(**true**){  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }}).start();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **while**(**true**){  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }}).start();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **while**(**true**){  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }}).start();  }  } |

执行结果：

Thread-0 0

Thread-1 1

Thread-2 2

Thread-1 4

Thread-0 3

Thread-2 5

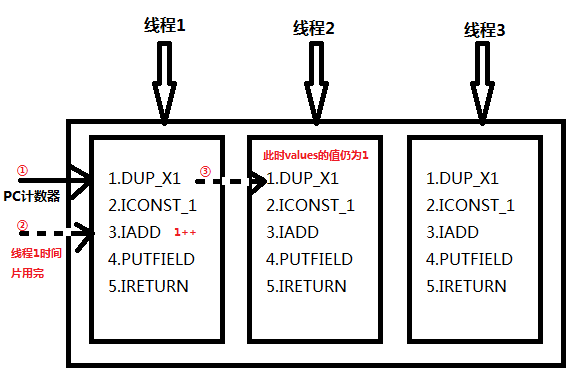
Thread-1 6

Thread-0 6

Thread-2 7

Thread-0 8

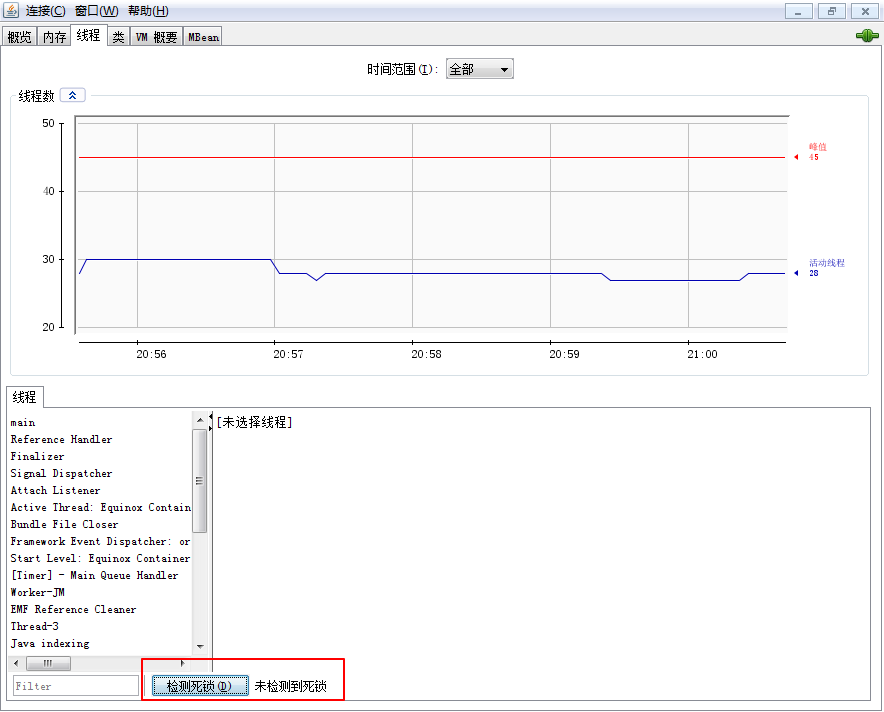
|  |
| --- |
| /\*  \* 增加synchronized可以使得线程安全，但同时也成了串行去自增序列，为了提高程序的性能，展开更多的问题探讨  \*/  **public** **synchronized** **int** getNext(){  **return** values++;  } |



### 4.2活跃性问题

* 死锁：五个哲学家使四双用刀子和叉子吃饭的问题。

Jconsole工具提供了一种检测死锁的方法：



* 饥饿：单窗口排队打饭，没素质的人插队，打完饭也不走，导致弱小的打不到饭吃而饿死。
* 活锁：两个人过两座桥，相互谦让，导致重重相遇而过不去河。

### 4.3饥饿与公平

1. 高优先级吞噬所有低优先级的CPU时间片

不同平台的线程优先等级设置可能会有所差异，如windows设置1-10，而linux设置可能是1-99，也有其他常亮定义的优先级。

线程被永久堵塞在一个等待进入同步块的状态

等待的线程永远不被唤醒

如何尽量避免饥饿问题

设置合理的优先级

使用锁来代替synchronized

### 4.4线程安全性问题的必要条件

1. 多线程环境下；
2. 多个线程共享一个资源；（struct2的action是共享变量的，解决办法就是指定action为多例的）
3. 对共享资源进行非原子性操作（读写操作）。

三个条件缺一不可。

错误实例：

|  |
| --- |
| **public** **class** MyServlet **extends** GenericServlet{  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  // 错误实例，多线程共享变量写操作有可能引起线程安全问题  **private** **int** i;  @Override  **public** **void** service(ServletRequest arg0, ServletResponse arg1) **throws** ServletException, IOException {  i++;  }  } |

## 第5节Synchronized原理与使用

### 5.1 synchronized使用

内置锁

互斥锁

修饰普通方法

修饰静态方法

修饰代码块

|  |
| --- |
| **public** **class** Sequence {  **private** **int** values;    /\*\*  \* 1、synchronized放在普通方法上，内置锁就是当前实例的对象  \* **@return**  \*/  **public** **synchronized** **int** getNext(){  **return** values++;  }    /\*\*  \* 2、synchronized 放在静态方法（类方法），内置锁是当前类的字节码对象  \* Sequence.class  \* **@return**  \*/  **private** **static** **int** getPrevious() {  **return** 0;  }    **public** **void** tasks(){    /\*// 1、this-->方法前加synchronized；  synchronized(this){    }\*/    /\*// 2、使用任意对象；  synchronized(Integer.valueOf(values)){    }\*/    // 3、使用类的字节码对象；（锁重入）  **synchronized**(Sequence.**class**){  **if**(values > 0){  System.***out***.println("values大于0...");  }**else**{  System.***out***.println("values小于0...");  }  }    }  } |

### 5.2 synchronized字节码指令

使用javap查看**synchronized**代码块开始和结束字节码：

|  |
| --- |
| public void tasks();  descriptor: ()V  flags: ACC\_PUBLIC  Code:  stack=2, locals=2, args\_size=1  0: ldc #1  2: dup  3: astore\_1  4: monitorenter  5: aload\_0  6: getfield #18  9: ifle 23  12: getstatic #22  15: ldc #28  17: invokevirtual #30  20: goto 31  23: getstatic #22  26: ldc #36  28: invokevirtual #30  31: aload\_1  32: monitorexit  33: goto 39  36: aload\_1  37: monitorexit  38: athrow  39: return  Exception table:  from to target type  5 33 36 any  36 38 36 any |

### 5.3锁原理及锁分类

任何对象都可以作为锁，那么锁信息又存在对象的什么地方呢？

存在对象头中

对象头中的信息

Mark Word

Class Metadata Address

Array Length

1. 偏向锁

每次获取锁和释放锁会浪费资源

很多情况下，竞争锁不是由多个线程，而是由一个线程在使用。

只有一个线程在访问同步代码块的场景

一个线程过来后，会看Mark Word中的信息：

* 线程id
* Epoch
* 对象的分代年龄信息
* 是否是偏向锁
* 锁标志位

1. 轻量级锁

自旋锁：第一个线程获取到锁之后，第二个线程不停的获取锁，直到第一个线程释放，

多个线程可同时

3）重量级锁

## 第6节单例模式与线程安全性问题

### 6.1饿汉式单例模式

* 饿汉式

没有线程安全性问题

|  |
| --- |
| **public** **class** SingletonHungry {  **private** SingletonHungry(){    }    // 饿汉式，直接创建实例  **private** **static** SingletonHungry *instance* = **new** SingletonHungry();  **public** **static** SingletonHungry getInstance(){  **return** *instance*;  }  /\*  \* 线程安全性问题的必要条件：  \* 1、多线程环境下；  \* 2、必须有共享变量；  \* 3、对资源进行原子性操作。  \*/  } |

### 6.2懒汉式单例模式

懒汉单例模式仍有可能引入线程不安全问题：

|  |
| --- |
| **public** **class** SingletonLazybones {    **private** SingletonLazybones(){    }    **private** **static** SingletonLazybones *instance* = **null**;    **public** **static** SingletonLazybones getInstance(){  **if**(*instance* == **null**){  // thread-0和thread-1同时进入if时，instance都为null，new了两个实例  *instance* = **new** SingletonLazybones();  }  **return** *instance*;  }  } |

执行结果：

……

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@25144c3e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

com.zte.thread.t5synchronized.SingletonLazybonesThreadSafe@104fa29e

……

出现了两个实例的情况。

### 6.3懒汉式单例模式改进

1. 使用synchronized关键字

|  |
| --- |
| **private** **static** SingletonLazybonesThreadSafe *instance* = **null**;  /\*\*  \* 1、方法添加synchronized关键字，可以避免线程安全性问题；  \* 虽然synchronized引入偏向锁、轻量级锁，性能已经提升了许多，偏向锁：单线程的情况使用偏向锁，显然单例模式不是单线程的；  \* 轻量级锁：第二个线程在if代码块外自旋，相当于while(true)，直到第一个线程释放锁，是不停的浪费CPU资源的。  \* **@return**  \*/  **public** **static** **synchronized** SingletonLazybonesThreadSafe getInstance(){  // 自旋 == while(true)  **if**(*instance* == **null**){  // thread-0和thread-1同时进入if时，instance都为null，new了两个实例  *instance* = **new** SingletonLazybonesThreadSafe();  }  **return** *instance*;  } |

1. 使用双重检测加锁

|  |
| --- |
| **private** **static** **volatile** SingletonLazybonesThreadSafe *instance2* = **null**;  /\*\*  \* 2、代码块添加synchronized，锁当前的字节码对象，并使用双重检测加锁;  \* 仍然会有指令重排序，在虚拟机中会有多个步骤：  \* 1）申请内存空间；  \* 2）在这块空间里实例化对象；  \* 3）instance的引用指向这块空间的地址。  \* 如果虚拟机先执行了3）后执行了2），这是instance虽然不为空，那么单例使用就会有问题。  \* 解决：在instance前加volatile，可以减少指令重排序问题；  \* **@return**  \*/  **public** **static** SingletonLazybonesThreadSafe getInstance2(){  **if**(*instance2* == **null**){  **synchronized** (SingletonLazybonesThreadSafe.**class**){  **if**(*instance2* == **null**){  *instance2* = **new** SingletonLazybonesThreadSafe();  }  }  }  **return** *instance*;  } |

## 第7节锁重入，自旋锁，以及死锁详解

### 7.1锁重入

单个线程，遇到同一个对象锁，可以重入锁执行

|  |
| --- |
| **public** **class** ReentrantLock {    **private** **synchronized** **void** a() {  System.***out***.println("a");  b();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    **private** **synchronized** **void** b() {  System.***out***.println("b");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **public** **static** **void** main(String[] args) {    /\*  \* 单个线程，遇到同一个对象锁，可以重入锁执行  \*/  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  ReentrantLock lock = **new** ReentrantLock();  lock.a();  }  }).start();  }  } |

执行结果：a、b同时打印，没有出现竞争。

多个线程，当线程0正在持有同一个对象锁时，其他线程必须等待其释放后才能持有

|  |
| --- |
| **public** **class** ReentrantLock {    **private** **synchronized** **void** a() {  System.***out***.println("a");  //b();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    **private** **synchronized** **void** b() {  System.***out***.println("b");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **public** **static** **void** main(String[] args) {    /\*  \* 单个线程，遇到同一个对象锁，可以重入锁执行  \*/  /\*new Thread(new Runnable() {    @Override  public void run() {  ReentrantLock lock = new ReentrantLock();  lock.a();  }  }).start();\*/    /\*  \* 多个线程，当线程0正在持有同一个对象锁时，其他线程必须等待其释放后才能持有  \*/  ReentrantLock lock = **new** ReentrantLock();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  lock.a();  }  }).start();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  lock.b();  }  }).start();  }  } |

执行结果：打印a后，等待1秒后打印b。

### 7.2自旋锁

场景：自旋锁示例，多个线程执行完毕后，打印一句话所有线程都执行完毕...

|  |
| --- |
| **public** **class** SpinLock {  **public** **static** **void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "线程执行");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行完毕了...");  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "线程执行");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行完毕了...");  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "线程执行");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行完毕了...");  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "线程执行");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行完毕了...");  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "线程执行");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行完毕了...");  }  }).start();    // 1.直接在代码块最后打印，并没有等到所有线程都执行完毕才打印  //System.out.println("所有线程都已经执行完毕了...");    // 2.使用if判断执行到以下代码时，是否只有一个线程（主线程，当然在实际开发中不能这样干），结果并没有打印任所有线程都已经执行完毕  /\*if(Thread.activeCount() == 1){  System.out.println("所有线程都已经执行完毕了...");  }\*/    // 3.使用while等待是否还剩下一个线程,否则空转CPU时间片，自旋  **while**(Thread.*activeCount*() != 1){  // 自旋  }  System.***out***.println("所有线程都已经执行完毕了...");  }  } |

1. 执行结果：

Thread-0线程执行

Thread-1线程执行

Thread-2线程执行

Thread-3线程执行

所有线程都已经执行完毕了...（提前打印）

Thread-4线程执行

Thread-0线程执行完毕了...

Thread-1线程执行完毕了...

Thread-2线程执行完毕了...

Thread-3线程执行完毕了...

Thread-4线程执行完毕了...

1. 执行结果：

Thread-0线程执行

Thread-1线程执行

Thread-2线程执行

Thread-3线程执行

Thread-4线程执行

Thread-0线程执行完毕了...

Thread-1线程执行完毕了...

Thread-2线程执行完毕了...

Thread-3线程执行完毕了...

Thread-4线程执行完毕了...

没有打印所有线程都已经执行完毕了...

1. 执行结果：

Thread-0线程执行

Thread-1线程执行

Thread-2线程执行

Thread-3线程执行

Thread-4线程执行

Thread-1线程执行完毕了...

Thread-0线程执行完毕了...

Thread-2线程执行完毕了...

Thread-3线程执行完毕了...

Thread-4线程执行完毕了...

所有线程都已经执行完毕了...（正确打印）

### 7.3死锁

构造两个共享变量o1、o2，作为方法a()和b()的可持有锁对象，a()需要先持有o1，然后再持有o2，b()需要先持有o2，然后再持有o1，两个线程一个执行a()，另一个执行b()，相互竞争，便会出现死锁。

|  |
| --- |
| **public** **class** DeadLock {    **private** Object o1 = **new** Object();  **private** Object o2 = **new** Object();    **private** **void** a(){  **synchronized**(o1){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **synchronized**(o2){  System.***out***.println("a");  }  }  }    **private** **void** b(){  **synchronized**(o2){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **synchronized**(o1){  System.***out***.println("b");  }  }  }  **public** **static** **void** main(String[] args) {  DeadLock lock = **new** DeadLock();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  lock.a();  }  }).start();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  lock.b();  }  }).start();    }  } |

执行后程序卡住，使用jconsole检测：



## 第8节volatile原理与使用

### 8.1 volatile的线程可见性

Volatile称之为轻量级锁，被volatile修饰的变量，在线程之间是可见的。

可见：一个线程修改了这个变量的值，在另外一个线程中能够读到这个修改后的值。

Synchronized除了线程之间互斥意外，还有一个非常大的作用，就是保证可见性

1. synchronized保证共享变量的可见性

|  |
| --- |
| /\*\*  \* synchronized与可见性示例  \* 保证可见性前提：多个线程拿到同一把锁  \* **@author** BYF  \*  \*/  **public** **class** ThreadVisibilityBySynchronized {  **public** **int** i = 1;  **public** **synchronized** **int** getI() {  **return** i;  }  **public** **synchronized** **void** setI(**int** i) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  // 如果线程是同步的，i的值一旦发生变化，i的值对持有同一把锁的的线程是可见的  **this**.i = i;  System.***out***.println("i的值已经发生改变...");  }    **public** **static** **void** main(String[] args) {  ThreadVisibilityBySynchronized t1 = **new** ThreadVisibilityBySynchronized();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  t1.setI(10);  }  }).start();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **while**(t1.getI()!=10){    }  System.***out***.println("自旋直到i的值为" +t1.getI());  }  }).start();      **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("最终执行结果：" + t1.getI());  }  } |

执行结果：

i的值已经发生改变...

自旋直到i的值为10

最终执行结果：10

2）volatile只针对线程间的共享变量可见性，并不能保证非原子操作的线程安全。

|  |
| --- |
| /\*\*  \* volatile与可见性示例  \*  \* **@author** BYF  \*  \*/  **public** **class** ThreadVisibilityByVolatile {  **public** **volatile** **boolean** run = **false**;    **public** **static** **void** main(String[] args) {  ThreadVisibilityByVolatile t1 = **new** ThreadVisibilityByVolatile();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **for** (**int** i = 1; i <= 5; i++) {  System.***out***.println(Thread.*currentThread*().getName() + "执行了第 " + i + "次...");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  t1.run = **true**;  }  }).start();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **while**(!t1.run){  // 自旋  }  System.***out***.println(Thread.*currentThread*().getName() + "线程执行了");  }  }).start();  }  } |

执行结果：

Thread-0执行了第 1次...

Thread-0执行了第 2次...

Thread-0执行了第 3次...

Thread-0执行了第 4次...

Thread-0执行了第 5次...

Thread-1线程执行了

### 8.2Lock指令

Volatile底层实现：创建对象实例时，如果存在volatile关键字，在汇编中类的实例就多了一个Lock指令，也就是Lock指令生效。

Lock指令在多处理器的系统上

1）将当前处理器缓存行(CPU缓存最小单位)的内容写回到系统内存；

2）这个写回到内存的操作会使在其他CPU里缓存了该内存地址的数据失效。

硬盘 -- 内存 -- CPU的缓存

硬盘中的class、网络中的class，通过java虚拟机载入内存，程序在内存执行过程中，会将一些内容放到CPU的缓存中，便于提高程序性能。

没有volatile修饰的变量，CPU计算完成后，并不会立即把结果写回到内存，而是在缓存中保留一段时间。如果是多CPU处理器，其中一个CPU计算后结果发生变化，其他CPU是不可见的。

加volatile之后的变量，CPU计算完成后，将结果写回到内存，并要求其他CPU的缓存该内存的地址失效，CPU遵循数据一致性协议，每个处理器通过嗅探在总线上传播的数据，来检测自己的缓存值是否过期，如果下次使用则从内存中重新读取。

如果在一个类中使用了大量的volatile关键字，那么CPU的缓存就无法得到很好的利用，程序的性能也就有所降低。同时，volatile修饰的变量会减少虚拟机的重排序，虚拟机的优化也不能充分的利用。

### 8.3volatile与synchronized比较

1. volatile仅保证多线程之间共享变量的可见性，并不能保证线程的安全性；

例如，在调用getI()时返回的是i++，并不是一个原子性操作，其他线程在访问时，无法保证其访问到的数据是一致的，这时使用volatile关键字修饰是没有意义的。

1. synchronized不仅可以保证线程之间共享变量的可见性，而且可以保证线程的安全性；
2. Synchronized可以完全取代volatile，但volatile不能替代synchronized，那为什么还要使用volatile？Volatile对于仅仅是get、set等原子操作，能够保证线程之间读的一致性，效率更高。

## 第9节JDK提供的原子类原理及使用

### 9.1原子类操作分类

* 原子更新基本类型
* 原子更新数组
* 原子更新抽象类型
* 原子更新字段

### 9.2原子类操作测试

|  |
| --- |
| **public** **class** Sequence {  **private** **int** values;  **public** AtomicInteger atomicInteger = **new** AtomicInteger();  **public** **int**[] intArray = { 2, 0, 1, 8 };  **public** AtomicIntegerArray atomicIntArray = **new** AtomicIntegerArray(intArray);    **public** AtomicReference<User> user = **new** AtomicReference<User>();    **public** AtomicIntegerFieldUpdater<User> age = AtomicIntegerFieldUpdater.*newUpdater*(User.**class**, "age");    /\*  \* 增加synchronized可以使得线程安全，但同时也成了串行去自增序列，为了提高程序的性能，展开更多的问题探讨  \*/  **public** **int** getNext() {  // return values++;    // 1、基本数据类型的原子类操作  // return atomicInteger.getAndIncrement();  // return atomicInteger.getAndAdd(10);    // 2、数组类型的原子类操作  //return atomicIntArray.getAndAdd(2, 10);  //return atomicIntArray.getAndIncrement(2);    // 3、对抽象数据类型User类实例的get和set原子类操作  // User a = new User();  // a.setAge(atomicInteger.getAndIncrement());  // a.setName("a");  // user.set(a);  // return user.get().getAge();    // 3、对抽象数据类型User类实例字段的get和set原子类操作  User a = **new** User();  a.setAge(atomicInteger.getAndIncrement());  a.setName("a");  **return** age.getAndIncrement(a);  }  **public** **int** getAtomicIntegerNext() {  **return** atomicInteger.getAndIncrement();  }  **public** **static** **void** main(String[] args) {  Sequence s = **new** Sequence();  // 只有一个线程任务在执行，顺序产生数字序列  /\*  \* while(true){ System.out.println(s.getNext()); }  \*/  // 定义多个线程，一起执行  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + s.getNext());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  }).start();  }  } |

1. 基本数据类型的原子类操作执行结果：

1）atomicInteger.getAndIncrement();

Thread-0 0

Thread-1 1

Thread-2 2

Thread-0 3

Thread-1 4

Thread-2 5

Thread-0 6

Thread-1 7

Thread-2 8

Thread-0 9

......

2）atomicInteger.getAndAdd(10);

Thread-0 0

Thread-1 10

Thread-2 20

Thread-0 30

Thread-1 40

Thread-2 50

Thread-0 60

Thread-1 70

Thread-2 80

Thread-0 90

Thread-2 100

......

1. 数组类型的原子类操作执行结果

1）atomicIntArray.getAndAdd(2, 10);

Thread-0 1

Thread-1 11

Thread-2 21

Thread-0 41

Thread-1 31

Thread-2 51

Thread-0 61

Thread-1 71

Thread-2 81

Thread-0 91

.....

2）atomicIntArray.getAndIncrement(2);

Thread-0 1

Thread-1 2

Thread-2 3

Thread-0 4

Thread-2 5

Thread-1 6

Thread-0 7

Thread-2 8

Thread-1 9

Thread-0 10

......

1. 对抽象数据类型User类实例的get和set原子类操作执行结果
2. User a = **new** User();

a.setAge(atomicInteger.getAndIncrement());

a.setName("a");

user.set(a);

**return** user.get().getAge();

Thread-0 0

Thread-1 1

Thread-2 2

Thread-0 3

Thread-1 3

Thread-2 5

Thread-0 6

Thread-1 7

Thread-2 8

Thread-0 10

......

1. 对抽象数据类型User类实例字段的get和set原子类操作执行结果：
2. User a = **new** User();

a.setAge(atomicInteger.getAndIncrement());

a.setName("a");

**return** age.getAndIncrement(a);

Thread-0 0

Thread-1 1

Thread-2 2

Thread-0 3

Thread-1 4

Thread-2 5

Thread-0 6

Thread-1 7

Thread-2 8

Thread-0 9

Thread-1 10

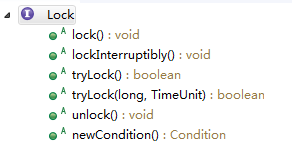
### 9.3Atomic原子类操作原理

|  |
| --- |
| **public** **int** getAndIncrement(T obj) {  **int** prev, next;  **do** {  prev = get(obj);  next = prev + 1;  } **while** (!compareAndSet(obj, prev, next));  **return** prev;  } |

compareAndSet检测prev在获取get(obj)时，是否有其他线程修改了get出来的值，如果有compareAndSet返回true，while循环继续自增的操作，直到只有线程自己完成了自增操作，没有其他线程修改get出来的值，那么compareAndSet返回false。

## 第10节Lock锁认识与使用

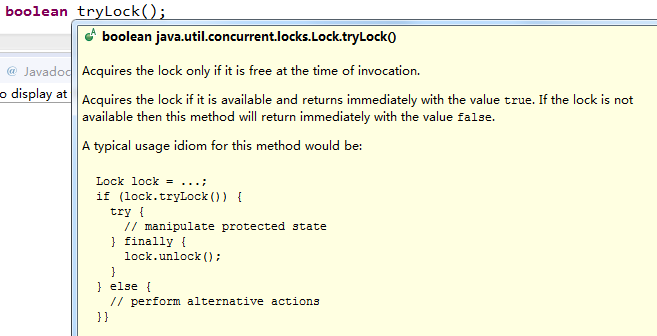
### 10.1Lock锁的认识

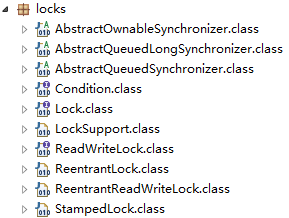


lock()：获取锁；

unlock()：释放锁；

tryLock()：尝试获取锁；





1）Lock需要显示地获取和释放锁，繁琐能让代码更灵活，锁的创建和获取是可以随时实现；

Synchronized不需要显示地获取和释放锁，简单；

2）使用Lock可以方便的实现公平性

3）非阻塞的获取锁；

能被中断的获取锁；

超时获取锁；

Synchronized：既能保证多线程变量可见性，又能保证非原子性的操作；

Volatile：仅能保证多线程变量的可见性；

AtomicInteger ：仅能保证多线程原子性操作；

### 10.2Lock锁的使用

使用lock、unlock

|  |
| --- |
| Lock lock = **new** ReentrantLock();  Lock lock1 = **new** ReentrantLock();  /\*  \* 增加synchronized可以使得线程安全，但同时也成了串行去自增序列，为了提高程序的性能，展开更多的问题探讨  \*/  **public** **int** getNext() {  // 每次进来都创建锁，是不能锁住这段代码块的，需要提到类的属性变量，公用一把锁。  //Lock lock = new ReentrantLock();  lock.lock();  lock1.unlock();  values++;  lock.unlock();  **return** values;  } |

使用tryLock

|  |
| --- |
| **private** **int** values;  Lock lock = **new** ReentrantLock();  Lock lock1 = **new** ReentrantLock();  /\*  \* 增加synchronized可以使得线程安全，但同时也成了串行去自增序列，为了提高程序的性能，展开更多的问题探讨  \*/  **public** **int** getNext() {  // 每次进来都创建锁，是不能锁住这段代码块的，需要提到类的属性变量，公用一把锁。  //Lock lock = new ReentrantLock();  **if**(lock.tryLock()){  **try** {  values++;  } **finally** {  lock.unlock();  }  } **else** {  System.***out***.println("获取锁失败....");  }  **return** values;  } |

### 10.3实现一个不可重入锁

1. 实现Lock接口实现一个简单自定义：
2. 定义类变量isLocked判断当前锁的状态；
3. 重写lock方法，自旋直到锁的状态是可持有时，继续往下运行；
4. 重写unlock方法，将锁的状态更改为可持有，唤醒出于阻塞状态的其他线程；
5. 实现方法lock和unlock都要用synchronized关键字修饰，保证是持有同一把锁的线程更改同一个isLocked状态；

|  |
| --- |
| **public** **class** MyLockNotReentered **implements** Lock{  **private** **boolean** isLocked = **false**;    @Override  **public** **synchronized** **void** lock() {  **while**(isLocked){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  isLocked = **true**;  }    @Override  **public** **synchronized** **void** unlock() {  isLocked = **false**;  notify();  }    @Override  **public** **void** lockInterruptibly() **throws** InterruptedException {  // **TODO** Auto-generated method stub    }  @Override  **public** **boolean** tryLock() {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** **boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** Condition newCondition() {  // **TODO** Auto-generated method stub  **return** **null**;  }  } |

2）实用上述锁，使用数字序列生成器进行加锁

|  |
| --- |
| **private** MyLockNotReentered lock = **new** MyLockNotReentered();  **private** **int** values = 0;  **public** **int** getNext() {  lock.lock();  values++;  lock.unlock();  **return** values;  } |

三个线程同时执行，执行结果：

Thread-0: 1

Thread-1: 2

Thread-2: 3

Thread-0: 4

Thread-1: 5

Thread-2: 6

Thread-0: 7

Thread-1: 8

Thread-2: 9

Thread-0: 10

.....

对于线程安全问题来说，上述实现是可以保证多线程安全，但是从可重入锁的角度来看，该锁的实现方式显然是不可重入的，只要线程第一次持有了锁，那么再想调用其他synchronized方法，或者需要同步代码块，调用lock.lock()就会出现永远的自旋下去。

### 10.4实现一个可重入锁

1）那么对同一个线程来说，要想使得锁是可重入的，自定义锁就需要以下条件：

1. 需要记录当前锁被哪个线程持有，来判断是否是同一个线程的重入；
2. 需要记录同一个线程重入的次数，在加锁的时候自增，在释放锁时自减；

|  |
| --- |
| **public** **class** MyLockReentered **implements** Lock{  **private** **boolean** isLocked = **false**;    **private** Thread lockBy = **null**;    **private** **int** countLocks = 0;    @Override  **public** **synchronized** **void** lock() {  Thread currentLock = Thread.*currentThread*();  **while**(isLocked && currentLock != lockBy){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  isLocked = **true**;  lockBy = currentLock;  countLocks++;  }  @Override  **public** **synchronized** **void** unlock() {  **if**(lockBy == Thread.*currentThread*()){  countLocks--;  **if**(countLocks == 0){  isLocked = **false**;  }  }  }    @Override  **public** **void** lockInterruptibly() **throws** InterruptedException {  // **TODO** Auto-generated method stub    }  @Override  **public** **boolean** tryLock() {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** **boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** Condition newCondition() {  // **TODO** Auto-generated method stub  **return** **null**;  }  } |

1. 使用相互调用的重入锁方法，测试可重入锁。

|  |
| --- |
| **public** **class** ReenteredMethod {    **private** MyLockReentered lock = **new** MyLockReentered();  //private MyLockNotReentered lock = new MyLockNotReentered();    **public** **void** a(){  lock.lock();  System.***out***.println("a()");  b();  lock.unlock();  }    **public** **void** b(){  lock.lock();  System.***out***.println("a-->b()");  c();  lock.unlock();  }    **public** **void** c(){  lock.lock();  System.***out***.println("b--c()");  lock.unlock();  }  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  ReenteredMethod obj = **new** ReenteredMethod();    **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  obj.a();  }  }).start();  }  } |

执行结果

a()

a-->b()

b--c()

## 第11节AbstractQueuedSynchronizer(AQS)详解

### 11.1AQS的API文档

<http://tool.oschina.net/apidocs/apidoc?api=jdk-zh>

为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量、事件，等等）提供一个框架。此类的设计目标是成为依靠单个原子 int 值来表示状态的大多数同步器的一个有用基础。子类必须定义更改此状态的受保护方法，并定义哪种状态对于此对象意味着被获取或被释放。假定这些条件之后，此类中的其他方法就可以实现所有排队和阻塞机制。子类可以维护其他状态字段，但只是为了获得同步而只追踪使用 getState()、setState(int) 和 compareAndSetState(int, int) 方法来操作以原子方式更新的 int 值。

应该将子类定义为非公共内部帮助器类，可用它们来实现其封闭类的同步属性。类 AbstractQueuedSynchronizer 没有实现任何同步接口。而是定义了诸如 acquireInterruptibly(int) 之类的一些方法，在适当的时候可以通过具体的锁和相关同步器来调用它们，以实现其公共方法。

### 11.2使用AQS重写自定义锁

1. 实现一个内部类继承AQS，重写tryAcquire和tryRelease方法；并继承Lock接口，实现Lock的几个接口方法。

|  |
| --- |
| **public** **class** MyLockByAQS **implements** Lock{    **private** Helper helper = **new** Helper();  **private** **class** Helper **extends** AbstractQueuedSynchronizer{    @Override  **protected** **boolean** tryAcquire(**int** arg)  {  // 1.如果第一个线程进来，可以拿到锁返回true    // 2.如果是第二个线程进来，拿不到锁，返回false    // 3.如何判断第一个线程进来还是其他线程进来    **int** state = getState();    // 改造为可重入的锁增加**else** **if**  Thread currentThread = Thread.*currentThread*();  **if**(state == 0){  **if**(compareAndSetState(0, arg)){  setExclusiveOwnerThread(Thread.*currentThread*());  **return** **true**;  }  } **else** **if**(currentThread == getExclusiveOwnerThread()){  setState(state + 1);  **return** **true**;  }  **return** **false**;  }    @Override  **protected** **boolean** tryRelease(**int** arg) {    //锁的获取和释放肯定是一一对应的，那么调用此方法的线程一定是当前线程  **if**(Thread.*currentThread*() != getExclusiveOwnerThread()){  **throw** **new** RuntimeException();  }  **int** state = getState() - arg;  **boolean** flag = **false**;  **if**(state == 0){  flag = **true**;  setExclusiveOwnerThread(**null**);  }  setState(state);    **return** flag;  }    Condition newCondition(){  **return** **new** ConditionObject();  }  }  @Override  **public** **void** lock() {  helper.acquire(1);  }  @Override  **public** **void** lockInterruptibly() **throws** InterruptedException {  helper.acquireInterruptibly(1);  }  @Override  **public** **boolean** tryLock() {  **return** helper.tryAcquire(1);  }  @Override  **public** **boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException {  **return** helper.tryAcquireNanos(1, unit.toNanos(time));  }  @Override  **public** **void** unlock() {  helper.release(1);  }  @Override  **public** Condition newCondition() {  **return** helper.newCondition();  }  } |

2）实现一个数字序列生成器测试线程安全

|  |
| --- |
| **public** **class** Sequence {  **private** **int** values;    **private** MyLockByAQS lock = **new** MyLockByAQS();  **private** **int** getNext() {  lock.lock();  **try** {  Thread.*sleep*(100);  **return** values++;  } **catch** (InterruptedException e) {  **throw** **new** RuntimeException();  } **finally**{  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {  Sequence sequence = **new** Sequence();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + "-->" + sequence.getNext());  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + "-->" + sequence.getNext());  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while**(**true**){  System.***out***.println(Thread.*currentThread*().getName() + "-->" + sequence.getNext());  }  }  }).start();    }  } |

执行结果：

Thread-0-->0

Thread-1-->1

Thread-2-->2

Thread-0-->3

Thread-1-->4

Thread-2-->5

Thread-0-->6

Thread-1-->7

Thread-2-->8

Thread-0-->9

Thread-1-->10

Thread-2-->11

Thread-0-->12

Thread-1-->13

Thread-2-->14

Thread-0-->15

Thread-1-->16

Thread-2-->17

......

3）测试自定义锁的可重入性

|  |
| --- |
| **public** **class** ReenteredMethod {    **private** MyLockByAQS lock = **new** MyLockByAQS();    **public** **void** a(){  lock.lock();  System.***out***.println("a");  b();  lock.unlock();  }  **public** **void** b() {  lock.lock();  System.***out***.println("b");  lock.unlock();  }    **public** **static** **void** main(String[] args) {  ReenteredMethod reenteredMethod = **new** ReenteredMethod();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  reenteredMethod.a();  }  }).start();  }    } |

执行结果：

a

b

### 11.3公平锁与非公平锁

1. 自定义一个公平锁

|  |
| --- |
| **public** **class** FairLock {  **private** **boolean** isLocked = **false**;  **private** Thread lockingThread = **null**;  **private** List<QueueObject> waitingThreads = **new** ArrayList<QueueObject>();  **public** **void** lock() **throws** InterruptedException {  QueueObject queueObject = **new** QueueObject();  **boolean** isLockedForThisThread = **true**;  **synchronized** (**this**) {  waitingThreads.add(queueObject);  }  // 实现可重入  **while**(isLockedForThisThread){  **synchronized**(**this**){  isLockedForThisThread = isLocked || waitingThreads.get(0) != queueObject;  **if**(!isLockedForThisThread){  isLocked = **true**;  waitingThreads.remove(queueObject);  lockingThread = Thread.*currentThread*();  **return**;  }  }  **try** {  queueObject.doWait();  } **catch** (InterruptedException e) {  **synchronized** (**this**) {  waitingThreads.remove(queueObject);  }  **throw** e;  }  }    /\*// 不可重入直接使用  try {  queueObject.doWait();  } catch (InterruptedException e) {  synchronized (this) {  waitingThreads.remove(queueObject);  }  throw e;  }\*/  }  **public** **synchronized** **void** unlock() {  **if** (**this**.lockingThread != Thread.*currentThread*()) {  **throw** **new** IllegalMonitorStateException("Calling thread has not locked this lock");  }  isLocked = **false**;  lockingThread = **null**;  **if** (waitingThreads.size() > 0) {  waitingThreads.get(0).doNotify();  }  }  } |

1. 每个锁都维护一个线程的队列，线程的唤醒由队列中对象的wait和notify方法唤醒各自进入锁的线程。锁的释放顺序从队列的首部对象唤醒当前线程。

|  |
| --- |
| **public** **class** QueueObject {  **private** **boolean** isNotified = **false**;  **public** **synchronized** **void** doWait() **throws** InterruptedException {  **while** (!isNotified) {  **this**.wait();  }  **this**.isNotified = **false**;  }  **public** **synchronized** **void** doNotify() {  **this**.isNotified = **true**;  **this**.notify();  }  **public** **boolean** equals(Object o) {  **return** **this** == o;  }  } |

11.4同步器的tryAcquire公平和非公平锁



1）公平锁的tryAcquire

|  |
| --- |
| /\*\*  \* Fair version of tryAcquire. Don't grant access unless  \* recursive call or no waiters or is first.  \*/  **protected** **final** **boolean** tryAcquire(**int** acquires) {  **final** Thread current = Thread.*currentThread*();  **int** c = getState();  **if** (c == 0) {  **if** (!hasQueuedPredecessors() &&  compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  **return** **true**;  }  }  **else** **if** (current == getExclusiveOwnerThread()) {  **int** nextc = c + acquires;  **if** (nextc < 0)  **throw** **new** Error("Maximum lock count exceeded");  setState(nextc);  **return** **true**;  }  **return** **false**;  } |

2）非公平锁的tryAcquire

|  |
| --- |
| **protected** **final** **boolean** tryAcquire(**int** acquires) {  **return** nonfairTryAcquire(acquires);  }  /\*\*  \* Performs non-fair tryLock. tryAcquire is implemented in  \* subclasses, but both need nonfair try for trylock method.  \*/  **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**;  } |

3）判断队列链表中的当前Node前是否有其他Node

|  |
| --- |
| **public** **final** **boolean** hasQueuedPredecessors() {  // The correctness of this depends on head being initialized  // before tail and on head.next being accurate if the current  // thread is first in queue.  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*());  } |

## 第12节读写锁

### 12.1读写锁概述

排他锁与共享锁

Synchronized、自定义Lock、ReenteredLock都是排他锁；

读写锁既是排他锁，也是共享锁。读锁是共享锁，写锁是排他锁；

排他锁：在同一时刻只允许一个线程访问，例如只要加上synchronized之后，那么在同一时刻只有一个线程能够进来，即使是轻量级锁能够进入到synchronized块内，但是它依然是出于自旋状态，它并不能执行synchronized代码块中的内容，所以它也是一种排他锁。

共享锁：在同一时刻可以由多个线程访问，例如读写锁的读锁，允许多个线程同时访问，也就是说共享锁与共享锁之间是可以同时访问的。读与读之间不是互斥的，读和写之间、写和写之间是互斥的。

数据库遇到高并发时，会设置读写分离，目的就是为了提高读的性能，其实一般在一个应用中大部分时间是读操作，而写操作是比较少的，如果使用排他锁那么对于所有的读写线程操作都是互斥的

### 12.2读写锁使用

1）实现一个具有读写锁的Map

|  |
| --- |
| **public** **class** ReadAndWriteLockMap {    **private** Map<String, Object> map = **new** HashMap<>();    **private** ReadWriteLock rwlock = **new** ReentrantReadWriteLock();    **private** Lock rlock = rwlock.readLock();    **private** Lock wlock = rwlock.writeLock();    **public** Object get(String key) {  rlock.lock();  System.***out***.println(Thread.*currentThread*().getName() + " 读操作正在执行...");  **try** {  **try** {  System.***out***.println("---->处理3秒钟...");  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** map.get(key);  } **finally** {  rlock.unlock();  System.***out***.println(Thread.*currentThread*().getName() + " 读操作执行完毕...");  }  }    **public** **void** put(String key, Object value) {  wlock.lock();  System.***out***.println(Thread.*currentThread*().getName() + " 写操作正在执行...");  **try** {  **try** {  System.***out***.println("---->处理3秒钟...");  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  map.put(key, value);  } **finally** {  wlock.unlock();  System.***out***.println(Thread.*currentThread*().getName() + " 写操作执行完毕...");  }  }  } |

2）测试写和写操作同时进行

|  |
| --- |
| **public** **class** WriteAndWriteOperate {  **public** **static** **void** main(String[] args) {  ReadAndWriteLockMap map = **new** ReadAndWriteLockMap();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  map.put("key1", "value1");  }  }).start();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  map.put("key2", "value2");  }  }).start();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  map.put("key3", "value3");  }  }).start();  }  } |

执行结果：

Thread-0 写操作正在执行...

---->处理3秒钟...

Thread-0 写操作执行完毕...

Thread-1 写操作正在执行...

---->处理3秒钟...

Thread-1 写操作执行完毕...

Thread-2 写操作正在执行...

---->处理3秒钟...

Thread-2 写操作执行完毕...

3）测试读写操作同时进行

|  |
| --- |
| **public** **class** WriteAndReadOperate {  **public** **static** **void** main(String[] args) {  ReadAndWriteLockMap map = **new** ReadAndWriteLockMap();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  map.put("key1", "value1");  }  }).start();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println(map.get("key1"));;  }  }).start();  }  } |

执行结果：

Thread-0 写操作正在执行...

---->处理3秒钟...

Thread-0 写操作执行完毕...

Thread-1 读操作正在执行...

---->处理3秒钟...

Thread-1 读操作执行完毕...

value1

4）测试读与读操作同时进行

|  |
| --- |
| **public** **class** ReadAndReadOperate {  **public** **static** **void** main(String[] args) {  ReadAndWriteLockMap map = **new** ReadAndWriteLockMap();  map.put("key1", "value1");  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println(map.get("key1"));;  }  }).start();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println(map.get("key1"));;  }  }).start();  }  } |

执行结果：

main 写操作正在执行...

---->处理3秒钟...

main 写操作执行完毕...

Thread-0 读操作正在执行...

---->处理3秒钟...

Thread-1 读操作正在执行...

---->处理3秒钟...

Thread-0 读操作执行完毕...

value1

Thread-1 读操作执行完毕...

value1

## 第13节线程之间的通信

### 13.1wait和notify

|  |
| --- |
| **public** **class** Demo2 {    **private** **volatile** **int** signal;    **public** **void** set (**int** value) {  **this**.signal = value;  }    **public** **int** get () {  **return** signal;  }    **public** **static** **void** main(String[] args) {  Demo2 d = **new** Demo2();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **synchronized** (d) {  System.***out***.println("修改状态的线程执行...");  **try** {  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  d.set(1);  System.***out***.println("状态值修改成功。。。");  d.notify();  }  }  }).start();      **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  **synchronized** (d) {  // 等待signal为1开始执行，否则不能执行  **while**(d.get() != 1) {  **try** {  d.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  // 当信号为1 的时候，执行代码  System.***out***.println("模拟代码的执行...");  }  }  }).start();  }  } |

执行结果：

修改状态的线程执行...

状态值修改成功。。。

模拟代码的执行...

(同时执行，唤醒后立刻执行)

### 13.2wait和notifyAll()

|  |
| --- |
| **public** **class** Demo3 {  **private** **volatile** **int** signal;    **public** **synchronized** **void** set () {  signal = 1;  notifyAll(); // notify方法会随机叫醒一个处于wait状态的线程  // notifyAll叫醒所有的处于wait线程，争夺到时间片的线程只有一个  System.***out***.println("叫醒线程叫醒之后休眠开始...");  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    **public** **synchronized** **int** get () {  System.***out***.println(Thread.*currentThread*().getName() + " 方法执行了...");  **if**(signal != 1) {  **try** {  wait();  System.***out***.println("叫醒之后");  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println(Thread.*currentThread*().getName() + " 方法执行完毕...");  **return** signal;  }    **public** **static** **void** main(String[] args) {    Demo3 d = **new** Demo3();  Target1 t1 = **new** Target1(d);  Target2 t2 = **new** Target2(d);    **new** Thread(t2).start();  **new** Thread(t2).start();  **new** Thread(t2).start();  **new** Thread(t2).start();    **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    **new** Thread(t1).start();    }  } |

执行结果：

Thread-0 方法执行了...

Thread-3 方法执行了...

Thread-2 方法执行了...

Thread-1 方法执行了...

叫醒线程叫醒之后休眠开始...

叫醒之后

Thread-1 方法执行完毕...

叫醒之后

Thread-2 方法执行完毕...

叫醒之后

Thread-3 方法执行完毕...

叫醒之后

Thread-0 方法执行完毕...

## 第14节 生产者消费者模式

### 14.1生产消费模型

1）天猫商城Tmall

|  |
| --- |
| **public** **class** Tmall {  // 库存  **private** **int** count = 0;    **private** **final** **int** MAX\_COUNT = 10;  **public** **synchronized** **void** push() {  **while**(count >= MAX\_COUNT){  System.***out***.println(Thread.*currentThread*().getName() + "生成达到上限，停止生产！");  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    count++;  System.***out***.println("生产者生成，当前库存为：" + count);  notifyAll();  }  **public** **synchronized** **void** get() {  **while**(count <=0 ){  System.***out***.println(Thread.*currentThread*() + "没有商品可以销售，消费者等待！");  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    count--;  System.***out***.println("消费者消费，当前库存为：" + count);  notifyAll();  }      } |

2）生产者PushTarget

|  |
| --- |
| **public** **class** PushTarget **implements** Runnable{    **private** Tmall tmall;    **public** PushTarget (Tmall tmall){  **this**.tmall = tmall;  }  @Override  **public** **void** run() {  **while**(**true**){  tmall.push();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    }  } |

3）消费者TakeTarget

|  |
| --- |
| **public** **class** TakeTarget **implements** Runnable{    **private** Tmall tmall;    **public** TakeTarget(Tmall tmall) {  **this**.tmall = tmall;  }  @Override  **public** **void** run() {  **while**(**true**){  tmall.get();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }      } |

4）主方法Main

|  |
| --- |
| **public** **class** Mail {  **public** **static** **void** main(String[] args) {  Tmall tmall = **new** Tmall();  PushTarget p = **new** PushTarget(tmall);  TakeTarget t = **new** TakeTarget(tmall);    **new** Thread(p).start();  **new** Thread(p).start();  **new** Thread(p).start();  **new** Thread(p).start();  **new** Thread(p).start();    **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();  **new** Thread(t).start();    }  } |

执行结果：

生产者生成，当前库存为：1

生产者生成，当前库存为：2

生产者生成，当前库存为：3

生产者生成，当前库存为：4

生产者生成，当前库存为：5

消费者消费，当前库存为：4

消费者消费，当前库存为：3

消费者消费，当前库存为：2

消费者消费，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-10,5,main]没有商品可以销售，消费者等待！

Thread[Thread-11,5,main]没有商品可以销售，消费者等待！

Thread[Thread-12,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-11,5,main]没有商品可以销售，消费者等待！

Thread[Thread-10,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

生产者生成，当前库存为：2

生产者生成，当前库存为：3

消费者消费，当前库存为：2

消费者消费，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-5,5,main]没有商品可以销售，消费者等待！

Thread[Thread-6,5,main]没有商品可以销售，消费者等待！

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

Thread[Thread-8,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

Thread[Thread-6,5,main]没有商品可以销售，消费者等待！

Thread[Thread-5,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-6,5,main]没有商品可以销售，消费者等待！

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

生产者生成，当前库存为：1

生产者生成，当前库存为：2

消费者消费，当前库存为：1

消费者消费，当前库存为：0

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-7,5,main]没有商品可以销售，消费者等待！

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

Thread[Thread-6,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

Thread[Thread-7,5,main]没有商品可以销售，消费者等待！

Thread[Thread-5,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

Thread[Thread-8,5,main]没有商品可以销售，消费者等待！

Thread[Thread-10,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-11,5,main]没有商品可以销售，消费者等待！

生产者生成，当前库存为：1

消费者消费，当前库存为：0

Thread[Thread-10,5,main]没有商品可以销售，消费者等待！

Thread[Thread-8,5,main]没有商品可以销售，消费者等待！

Thread[Thread-9,5,main]没有商品可以销售，消费者等待！

## 线程顺序被唤醒执行之Condition

### 15.1 使用synchronized+signal实现

关键点：

1.三个方法等待被线程调用，并且顺序执行；

2.三个线程任务，分别不停调用1中的三个方法，各自调用一个；

3.主线程实现三个线程，分别start，此时并非顺序调用；

4.使用signal分别标识每个方法执行的值，执行完改变signal，唤醒所有线程；

5.方法增加synchronized和wait，执行完后，notifyAll，此时顺序执行。

|  |
| --- |
| **public** **class** AThenBThenC {    **private** **int** signal;  **public** **synchronized** **void** a() {  **while**(signal != 0){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("->a");  signal++;  notifyAll();  }  **public** **synchronized** **void** b() {  **while**(signal != 1){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("-->b");  signal++;  notifyAll();  }  **public** **synchronized** **void** c() {  **while**(signal != 2){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("--->c");  signal = 0;  notifyAll();  }    **public** **static** **void** main(String[] args) {  AThenBThenC d = **new** AThenBThenC();    A a = **new** A(d);  B b = **new** B(d);  C c = **new** C(d);    **new** Thread(a).start();  **new** Thread(b).start();  **new** Thread(c).start();  }    }  **class** A **implements** Runnable{  **private** AThenBThenC aThenBThenC;    **public** A (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**) {  aThenBThenC.a();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }    }  **class** B **implements** Runnable{    **private** AThenBThenC aThenBThenC;    **public** B (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**) {  aThenBThenC.b();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    }    }  **class** C **implements** Runnable{    **private** AThenBThenC aThenBThenC;    **public** C (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**){  aThenBThenC.c();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }    } |

### 15.2 使用Condition锁，唤醒指定线程

关键点：

1. 删除所有15.1中synchronized，实现一个ReenteredLock，并通过lock.newCondition()的方式，实现三个Condition对象；
2. 三个方法增加lock.lock()和lock.unlock()，wait改成对于Condition的a.await()，唤醒时指定唤醒锁，b.signal()；

|  |
| --- |
| **public** **class** AThenBThenC {    **private** **int** signal;    **private** Lock lock = **new** ReentrantLock();  **private** Condition a = lock.newCondition();  **private** Condition b = lock.newCondition();  **private** Condition c = lock.newCondition();    **public** **void** a() {  lock.lock();  **while**(signal != 0){  **try** {  a.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("->a");  signal++;  b.signal();  lock.unlock();  }  **public** **void** b() {  lock.lock();  **while**(signal != 1){  **try** {  b.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("-->b");  signal++;  c.signal();  lock.unlock();  }  **public** **void** c() {  lock.lock();  **while**(signal != 2){  **try** {  c.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("--->c");  signal = 0;  a.signal();  lock.unlock();  }    **public** **static** **void** main(String[] args) {  AThenBThenC d = **new** AThenBThenC();    A a = **new** A(d);  B b = **new** B(d);  C c = **new** C(d);    **new** Thread(a).start();  **new** Thread(b).start();  **new** Thread(c).start();  }    }  **class** A **implements** Runnable{  **private** AThenBThenC aThenBThenC;    **public** A (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**) {  aThenBThenC.a();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }    }  **class** B **implements** Runnable{    **private** AThenBThenC aThenBThenC;    **public** B (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**) {  aThenBThenC.b();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    }    }  **class** C **implements** Runnable{    **private** AThenBThenC aThenBThenC;    **public** C (AThenBThenC aThenBThenC) {  **this**.aThenBThenC = aThenBThenC;  }  @Override  **public** **void** run() {  **while**(**true**){  aThenBThenC.c();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }    } |

## 第16节使用Condition重写waitnotify案例并实现一个有界队列

### 16.1改造TMall使用Condition实现消费者生产者

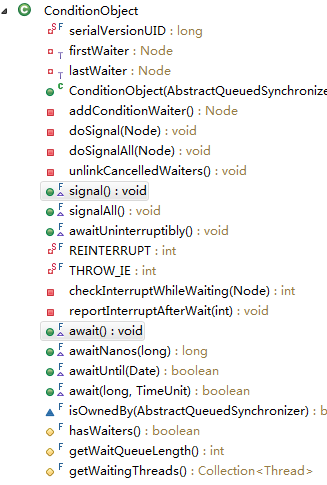
|  |
| --- |
| **public** **class** TmallCondition {  // 库存  **private** **int** count = 0;    **private** Lock lock = **new** ReentrantLock();  **private** Condition pushCondition = lock.newCondition();  **private** Condition getCondition = lock.newCondition();    **private** **final** **int** MAX\_COUNT = 10;  **public** **void** push() {  lock.lock();  **while**(count >= MAX\_COUNT){  System.***out***.println(Thread.*currentThread*().getName() + "生成达到上限，停止生产！");  **try** {  pushCondition.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    count++;  System.***out***.println("生产者生成，当前库存为：" + count);  getCondition.signal();  lock.unlock();  }  **public** **void** get() {  lock.lock();  **while**(count <=0 ){  System.***out***.println(Thread.*currentThread*() + "没有商品可以销售，消费者等待！");  **try** {  getCondition.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    count--;  System.***out***.println("消费者消费，当前库存为：" + count);  pushCondition.signal();  lock.unlock();  }      } |

### 16.2使用Condition实现有界线程安全队列

|  |
| --- |
| **public** **class** MyQueueLimitList<E> {    **private** Object [] obj;    **private** **int** addIndex;  **private** **int** removeIndex;  **private** **int** queueSize;    **private** Lock lock = **new** ReentrantLock();  Condition addCondition = lock.newCondition();  Condition removeCondition = lock.newCondition();    **public** MyQueueLimitList(**int** count) {  obj = **new** Object[count];  }  **public** **void** add(E e) {  lock.lock();  **while**(queueSize == obj.length) {  **try** {  addCondition.await();  } **catch** (InterruptedException e1) {  e1.printStackTrace();  }  }  obj[addIndex] = e;    **if**(++ addIndex == obj.length){  addIndex = 0;  }  queueSize ++;  removeCondition.signal();  lock.unlock();  }    **private** **void** remove() {  lock.lock();  **while**(queueSize == 0){  **try** {  // System.out.println(Thread.currentThread().getName() + "队列为空，不进行溢出");  removeCondition.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  obj[removeIndex] = **null**;  **if**(++ removeIndex == obj.length){  removeIndex = 0;  }  queueSize --;  addCondition.signal();  lock.unlock();  }    } |

## 第17节 深入解析Condition源码

ReenteredLock-->newCondition-->(await、signal)



### 17.1同步队列与等待队列

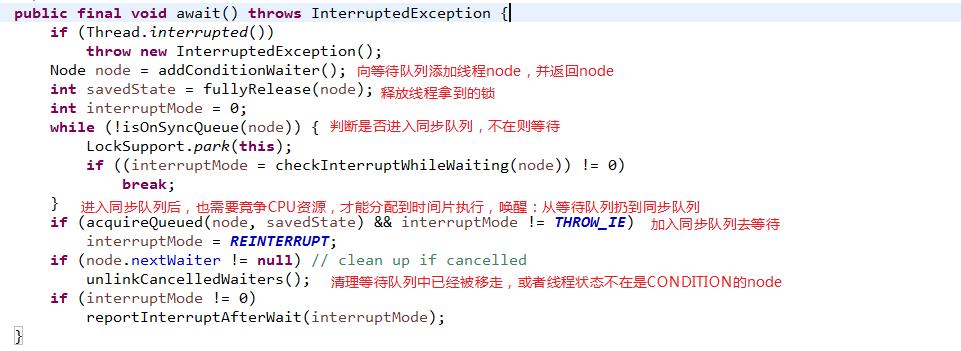
同步队列：

线程没有竞争到锁，会被扔到同步队列；

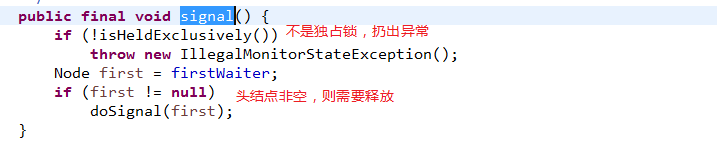
等待队列：

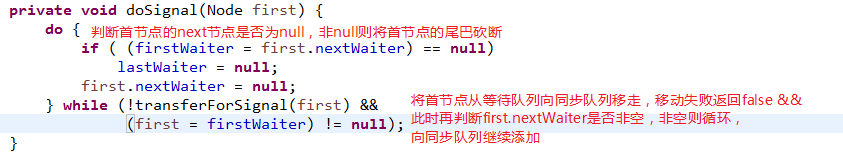
调用wait方法，会被扔到等待队列；

### 17.2 await方法



### 17.3 signal方法





## 第18节线程通信之join应用与实现原理剖析

### 18.1线程加塞join Demo

|  |
| --- |
| **public** **class** JoinThread {  **public** **void** a(Thread joinThread) {  System.***out***.println("方法a执行了...");  joinThread.start();  **try** {  joinThread.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("方法a执行完毕了...");  }  **public** **void** b() {  System.***out***.println("加塞线程开始执行....");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("加塞线程执行完毕....");  }  **public** **static** **void** main(String[] args) {  JoinThread joinThreadDemo = **new** JoinThread();  Thread joinThread = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  joinThreadDemo.b();  }  });  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  joinThreadDemo.a(joinThread);  }  }).start();  }  } |

执行结果：

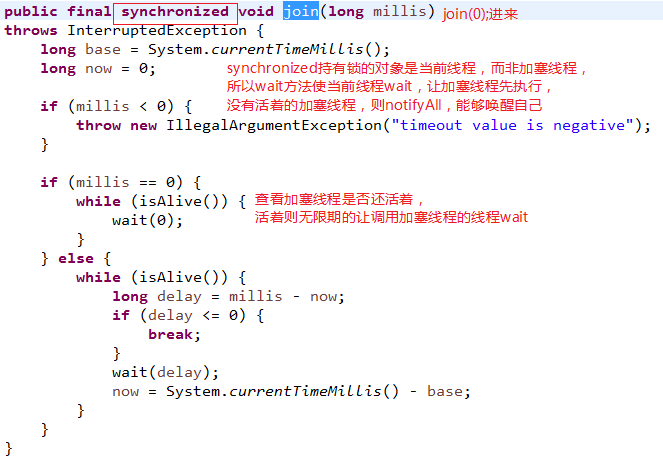
方法a执行了...

加塞线程开始执行....

加塞线程执行完毕....

方法a执行完毕了...

### 18.2 join原理



## ThreadLocal原理与使用

ThreadLocal表示线程的局部变量，之前出现的线程安全性问题，都是由于共享资源导致，有了ThreadLocal也可以解决线程安全性问题。

每个线程有一个局部变量，比如，数据库连接池，常见的session都可以使用ThreadLocal管理，例如数据库连接池，每一个线程有一个连接，这个连接对于这个线程来说是独立的。

### 19.1 ThreadLocal使用

1）当前线程是由哪个用户登录，请求的对象是什么

|  |
| --- |
| **public** **class** RequestHolder {  **private** **static** **final** ThreadLocal<SysUser> ***userHolder*** = **new** ThreadLocal<SysUser>();  **private** **static** **final** ThreadLocal<HttpServletRequest> ***requestHolder*** = **new** ThreadLocal<HttpServletRequest>();  **public** **static** **void** add(SysUser sysUser) {  ***userHolder***.set(sysUser);  }  **public** **static** **void** add(HttpServletRequest request) {  ***requestHolder***.set(request);  }  **public** **static** SysUser getCurrentUser() {  **return** ***userHolder***.get();  }  **public** **static** HttpServletRequest getCurrentRequest() {  **return** ***requestHolder***.get();  }  **public** **static** **void** remove() {  ***userHolder***.remove();  ***requestHolder***.remove();  }  } |

2）每个线程一个自增序列

|  |
| --- |
| **public** **class** ThreadLocalDemo {  **public** ThreadLocal<Integer> count = **new** ThreadLocal<Integer>() {  @Override  **protected** Integer initialValue() {  **return** **new** Integer(0);  }  };  **public** **int** getNext() {  Integer value = count.get();  value++;  count.set(value);  **return** value;  }  **public** **static** **void** main(String[] args) {  ThreadLocalDemo d = **new** ThreadLocalDemo();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + d.getNext());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + d.getNext());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName() + " " + d.getNext());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();  }  } |

运行结果：

Thread-0 1

Thread-1 1

Thread-2 1

Thread-0 2

Thread-1 2

Thread-2 2

Thread-0 3

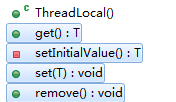
Thread-1 3

Thread-2 3

### 19.2ThreadLocal原理

使用Thread类线程实例的ThreadLocalMap局部变量，将ThreadLocal对象作为ThreadLocalMap的key，它的值作为ThreadLocalMap的value，从而实现每个线程独享一份局部变量。

关键方法：



注意：setInitialValue默认返回null，如果没有赋予初值，会抛出空指针异常。

## 并发工具类详解

### 20.1 CountDownLatch

对一个多行数字文件求和，每行一个线程求当前行的和

Data.txt

1,2,3,4,5

6,7,8,9,10

11,12,13,14,15

16,17,18,19,20

1. 使用查看当前存活线程数等于1，只有主线程执行时，才进行对所有行的和求总和；

|  |
| --- |
| **public** **class** ReadFileAndSum {  **private** **int** [] nums;    **public** ReadFileAndSum(**int** line) {  nums = **new** **int**[line];  }    **public** **void** calc(String line, **int** index) {  String [] numStrs = line.split(",");  **int** total = 0;  **for**(String num : numStrs){  total += Integer.*parseInt*(num);  }  nums[index] = total;  System.***out***.println("当前行求和结果为：" + total);  }    **public** **void** sum() {  System.***out***.println("汇总线程开始执行...");  **int** total = 0;  **for**(**int** i=0;i<nums.length;i++){  total += nums[i];  }  System.***out***.println("最終求和結果為：" + total);  }    **private** **static** List<String> readFile(){  List<String> contents = **new** ArrayList<>();  String line = **null**;  BufferedReader br = **null**;  **try** {  br = **new** BufferedReader(**new** FileReader("C:/Users/BYF/workspace/concurrent-program/src/com/zte/thread/ta19CountDownLatch/data.txt"));  **while**((line = br.readLine()) != **null**){  contents.add(line);  }  } **catch** (Exception e) {  // **TODO**: handle exception  }  **return** contents;  }    **public** **static** **void** main(String[] args) {  List<String> contents = *readFile*();  **int** lineCount = contents.size();    ReadFileAndSum rdf = **new** ReadFileAndSum(lineCount);    **for** (**int** i = 0; i < lineCount; i++) {  **final** **int** j = i;  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  rdf.calc(contents.get(j), j);  }    }).start();  }    **while**(Thread.*activeCount*() > 1){    }  rdf.sum();  }  } |

1. 使用CountDownLatch让主线程等待其他计算线程执行完毕后，汇总行的和求总和；

|  |
| --- |
| **public** **class** ReadFileAndSum {  **private** **int** [] nums;    **public** ReadFileAndSum(**int** line) {  nums = **new** **int**[line];  }    **public** **void** calc(String line, **int** index, CountDownLatch latch) {  String [] numStrs = line.split(",");  **int** total = 0;  **for**(String num : numStrs){  total += Integer.*parseInt*(num);  }  nums[index] = total;  System.***out***.println("当前行求和结果为：" + total);  latch.countDown();  }    **public** **void** sum() {  System.***out***.println("汇总线程开始执行...");  **int** total = 0;  **for**(**int** i=0;i<nums.length;i++){  total += nums[i];  }  System.***out***.println("最終求和結果為：" + total);  }    **private** **static** List<String> readFile(){  List<String> contents = **new** ArrayList<>();  String line = **null**;  BufferedReader br = **null**;  **try** {  br = **new** BufferedReader(**new** FileReader("C:/Users/BYF/workspace/concurrent-program/src/com/zte/thread/ta19CountDownLatch/data.txt"));  **while**((line = br.readLine()) != **null**){  contents.add(line);  }  } **catch** (Exception e) {  // **TODO**: handle exception  }  **return** contents;  }    **public** **static** **void** main(String[] args) {  List<String> contents = *readFile*();  **int** lineCount = contents.size();    CountDownLatch latch = **new** CountDownLatch(lineCount);  ReadFileAndSum rdf = **new** ReadFileAndSum(lineCount);    **for** (**int** i = 0; i < lineCount; i++) {  **final** **int** j = i;  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  rdf.calc(contents.get(j), j, latch);  }    }).start();  }    **try** {  latch.await();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  /\*while(Thread.activeCount() > 1){    }\*/  rdf.sum();  }  } |

执行结果：

当前行求和结果为：15

当前行求和结果为：90

当前行求和结果为：65

当前行求和结果为：40

汇总线程开始执行...

最終求和結果為：210

1. 源码解析



### 20.2 CyclicBarrier

1）下午三点开始大会，当所有人到齐后，开始开会，开始后每个人都要发言。

|  |
| --- |
| **public** **class** Meeting {  Random random = **new** Random();  **public** **void** meeting(CyclicBarrier barrier) {  **try** {  Thread.*sleep*(random.nextInt(4000));  } **catch** (InterruptedException e1) {  // **TODO** Auto-generated catch block  e1.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + " 到达会议室，等待开会");  // 线程1进来后挂了，没有执行barrier.await()，那么所有人都得等到够10个人才能开始  **if**(Thread.*currentThread*().getName().equals("Thread-1")){  // **throw** **new** RuntimeException();  }  **try** {  barrier.await();  } **catch** (InterruptedException | BrokenBarrierException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.***out***.println(Thread.*currentThread*().getName() + " 发言");  }  **public** **static** **void** main(String[] args) {  Meeting meet = **new** Meeting();    CyclicBarrier barrier = **new** CyclicBarrier(10, **new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("人到齐了，开始开会...");  }  });    **for**(**int** i = 0 ; i< 10; i++){  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  meet.meeting(barrier);  }  }).start();  }  }  } |

执行结果：

Thread-3 到达会议室，等待开会

Thread-9 到达会议室，等待开会

Thread-7 到达会议室，等待开会

Thread-4 到达会议室，等待开会

Thread-0 到达会议室，等待开会

Thread-5 到达会议室，等待开会

Thread-8 到达会议室，等待开会

Thread-2 到达会议室，等待开会

Thread-6 到达会议室，等待开会

Thread-1 到达会议室，等待开会

人到齐了，开始开会...

Thread-1 发言

Thread-3 发言

Thread-9 发言

Thread-7 发言

Thread-0 发言

Thread-8 发言

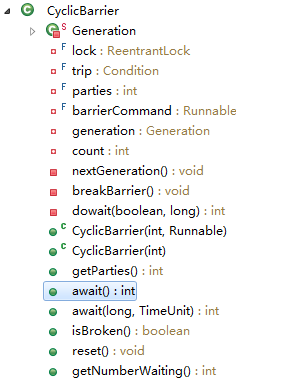
Thread-4 发言

Thread-6 发言

Thread-2 发言

Thread-5 发言

1. 源码解析



等待所有各方都在这个障碍上等待。

如果当前线程不是最后到达的线程，那么它将被禁用以进行线程调度，并且在发生以下任何一种情况之前处于休眠状态：

最后一个帖子到了;要么

其他一些线程会中断当前线程;要么

其他一些线程会中断其他一个等待线程;要么

其他一些线程在等待障碍时超时;要么

其他一些线程调用此屏障上的重置。

如果当前线程：

在进入此方法时设置其中断状态;要么

在等待时被打断

然后抛出InterruptedException并清除当前线程的中断状态。

如果在任何线程正在等待时重置屏障，或者在调用await时或在任何线程正在等待时屏障被破坏，则抛出BrokenBarrierException。

如果任何线程在等待时被中断，那么所有其他等待线程将抛出BrokenBarrierException并且屏障处于断开状态。

如果当前线程是要到达的最后一个线程，并且在构造函数中提供了非null屏障操作，则当前线程在允许其他线程继续之前运行该操作。如果在屏障操作期间发生异常，则该异常将在当前线程中传播，并且屏障处于断开状态。

### 20.3 Semaphore

1）线程不停的创建，但正在执行的只能有10个，其他线程不断的创建，但都处于等待状态

|  |
| --- |
| **public** **class** ThreadThoughMethod {  **public** **void** method (Semaphore semaphore) {  **try** {  semaphore.acquire();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.***out***.println(Thread.*currentThread*().getName() + " is running...");    **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    semaphore.release();  }    **public** **static** **void** main(String[] args) {  ThreadThoughMethod t = **new** ThreadThoughMethod();  Semaphore semaphore = **new** Semaphore(10);    **while**(**true**){  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  t.method(semaphore);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }).start();  }    // 与线程池ExcutorService，通过Executors.newFixedThreadPool(10)创建  // 同时只能有十个线程执行，但要想创建新的线程，必须要等到一个线程归还线程池才可以    /\*ExecutorService threadPool = Executors.newFixedThreadPool(10);  while(true){  threadPool.execute(new Runnable() {    @Override  public void run() {  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + " 执行完毕");  }  });  }\*/  }  } |

2）与线程池，只有10个线程在执行的区别

|  |
| --- |
| **public** **class** ThreadThoughMethod {  **public** **void** method (Semaphore semaphore) {  **try** {  semaphore.acquire();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.***out***.println(Thread.*currentThread*().getName() + " is running...");  }    **public** **static** **void** main(String[] args) {  ThreadThoughMethod t = **new** ThreadThoughMethod();  Semaphore semaphore = **new** Semaphore(10);    /\*while(true){  new Thread(new Runnable() {    @Override  public void run() {  t.method(semaphore);  }  }).start();  }\*/    // 与线程池ExcutorService，通过Executors.newFixedThreadPool(10)创建  // 同时只能有十个线程执行，但要想创建新的线程，必须要等到一个线程归还线程池才可以    ExecutorService threadPool = Executors.*newFixedThreadPool*(10);  **while**(**true**){  threadPool.execute(**new** Runnable() {    @Override  **public** **void** run() {  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + " 执行完毕");  }  });  }  }  } |

执行结果

Thread-1 is running...

Thread-2 is running...

Thread-3 is running...

Thread-0 is running...

Thread-4 is running...

Thread-6 is running...

Thread-5 is running...

Thread-7 is running...

Thread-8 is running...

Thread-9 is running...

Thread-22156 is running...

Thread-22157 is running...

Thread-10 is running...

Thread-22158 is running...

Thread-11 is running...

Thread-22159 is running...

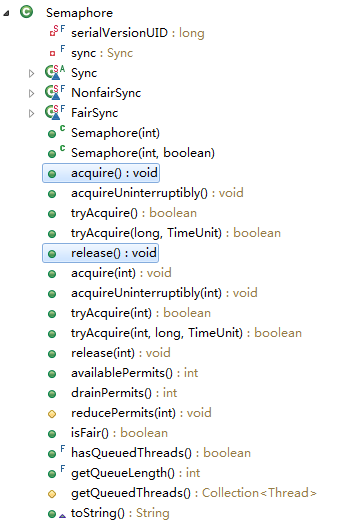
Thread-12 is running...

Thread-22160 is running...

Thread-13 is running...

Thread-14 is running...

3）源码解析



Acquire：

从此信号量获取许可，阻塞直到一个可用，或者线程被中断。

获得许可证（如果有）并立即返回，将可用许可证数量减少一个。

如果没有可用的许可证，那么当前线程将被禁用以进行线程调度，并且在发生以下两种情况之一之前处于休眠状态：

其他一些线程调用此信号量的释放方法，然后为当前线程分配一个许可证;要么

其他一些线程会中断当前线程。

如果当前线程：

在进入此方法时设置其中断状态;要么

在等待许可证时被打断，

然后抛出InterruptedException并清除当前线程的中断状态。

Release

发布许可证，将其返回到信号量。

发布许可证，将可用许可证数量增加一个。如果任何线程试图获得许可证，则选择一个并获得刚刚发布的许可证。该线程被（重新）启用以进行线程调度。

没有要求释放许可证的线程必须通过调用获取来获得该许可证。通过应用程序中的编程约定来建立信号量的正确使用。

### 20.4 Exchanger

1）A线程需要依赖b线程产生的数据，进行比较结果

|  |
| --- |
| **public** **class** TwoThreadCaptureData {  **public** **void** a (Exchanger<String> exch) {  System.***out***.println("a 开始抓取数据...");  **try** {  System.***out***.println("a 正在抓取数据...");  Thread.*sleep*(2000);  System.***out***.println("a 抓取完毕...");  } **catch** (InterruptedException e) {  e.printStackTrace();  }    String res = "12345";  **try** {  System.***out***.println("等待对比结果...");  exch.exchange(res);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** **void** b (Exchanger<String> exch) {  System.***out***.println("b 开始抓取数据...");  **try** {  System.***out***.println("b 正在抓取数据...");  Thread.*sleep*(4000);  System.***out***.println("b 抓取完毕...");  } **catch** (InterruptedException e) {  e.printStackTrace();  }    String res = "12345";    **try** {  String value = exch.exchange(res);  System.***out***.println("开始进行对比...");  System.***out***.println("比对结果为：" + value.equals(res));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **public** **static** **void** main(String[] args) {  TwoThreadCaptureData t = **new** TwoThreadCaptureData();    Exchanger<String> exch = **new** Exchanger<>();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  t.b(exch);  }  }).start();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  t.a(exch);  }  }).start();  }  } |

执行结果：

b 开始抓取数据...

b 正在抓取数据...

a 开始抓取数据...

a 正在抓取数据...

a 抓取完毕...

等待对比结果...

b 抓取完毕...

开始进行对比...

比对结果为：true

2）jdk文档中交换缓冲区的例子也是比较形象，一个线程写缓冲区，写满后交给读缓冲区，读完清空缓冲区交给写缓冲区。

|  |
| --- |
| **class** FillAndEmpty {  Exchanger<StringBuffer> exchanger = **new** Exchanger<StringBuffer>();  StringBuffer initialEmptyBuffer = **new** StringBuffer();  StringBuffer initialFullBuffer = **new** StringBuffer();  **public** **static** **int** *i* = 0;    **public** **static** **void** printBuffer(StringBuffer sb) {  **for**(**int** i = 0; i< sb.length(); i++){  System.***out***.print(sb.charAt(i));  System.***out***.print("|");  }  }  **class** FillingLoop **implements** Runnable {  **public** **void** run() {  StringBuffer currentBuffer = initialEmptyBuffer;  **try** {  **while** (currentBuffer != **null**) {  addToBuffer(currentBuffer);  System.***out***.println(Thread.*currentThread*().getName() + "当前缓冲区长度：" + currentBuffer.length());  **if** (currentBuffer.length() == 5){  System.***out***.println(Thread.*currentThread*().getName() + "缓冲区满，等待换取空的缓冲区");  currentBuffer = exchanger.exchange(currentBuffer);  }  }  } **catch** (InterruptedException ex) {  }  }  **private** **void** addToBuffer(StringBuffer currentBuffer) {    **try** {  Thread.*sleep*(100);  **if**(*i* < 5) {  currentBuffer.append(++*i*);  System.***out***.println(Thread.*currentThread*().getName() + "write->填充缓冲区 ： ");  } **else** {  System.***out***.println(Thread.*currentThread*().getName() + "缓冲区满，等待交换...");  }  *printBuffer*(currentBuffer);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **class** EmptyingLoop **implements** Runnable {  **public** **void** run() {  StringBuffer currentBuffer = initialFullBuffer;  **try** {  **while** (currentBuffer != **null**) {  takeFromBuffer(currentBuffer);  **if** (currentBuffer.length() == 0){  System.***out***.println(Thread.*currentThread*().getName() + "缓冲区空，等待换取满的缓冲区");  currentBuffer = exchanger.exchange(currentBuffer);  }    }  } **catch** (InterruptedException ex) {    }  }  **private** **void** takeFromBuffer(StringBuffer currentBuffer) {    **try** {  **if**(currentBuffer.length() == 5) {  System.***out***.println(Thread.*currentThread*().getName() + "read<-开始从缓冲区取数据: ");  Thread.*sleep*(100);  currentBuffer.setLength(0);  *i*= 0;  System.***out***.println(Thread.*currentThread*().getName() + "read<-读完缓冲区，清空...");  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **void** start() {  **new** Thread(**new** FillingLoop()).start();  **new** Thread(**new** EmptyingLoop()).start();  }    **public** **static** **void** main(String[] args) {  FillAndEmpty fae = **new** FillAndEmpty();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  fae.start();  }  }).start();  }  } |

执行结果：

Thread-2缓冲区空，等待换取满的缓冲区

Thread-1write->填充缓冲区 ：

1|Thread-1当前缓冲区长度：1

Thread-1write->填充缓冲区 ：

1|2|Thread-1当前缓冲区长度：2

Thread-1write->填充缓冲区 ：

1|2|3|Thread-1当前缓冲区长度：3

Thread-1write->填充缓冲区 ：

1|2|3|4|Thread-1当前缓冲区长度：4

Thread-1write->填充缓冲区 ：

1|2|3|4|5|Thread-1当前缓冲区长度：5

Thread-1缓冲区满，等待换取空的缓冲区

Thread-2read<-开始从缓冲区取数据:

Thread-1缓冲区满，等待交换...

Thread-1当前缓冲区长度：0

Thread-2read<-读完缓冲区，清空...

Thread-2缓冲区空，等待换取满的缓冲区

Thread-1write->填充缓冲区 ：

1|Thread-1当前缓冲区长度：1

Thread-1write->填充缓冲区 ：

1|2|Thread-1当前缓冲区长度：2

Thread-1write->填充缓冲区 ：

1|2|3|Thread-1当前缓冲区长度：3

Thread-1write->填充缓冲区 ：

1|2|3|4|Thread-1当前缓冲区长度：4

Thread-1write->填充缓冲区 ：

1|2|3|4|5|Thread-1当前缓冲区长度：5

Thread-1缓冲区满，等待换取空的缓冲区

## 提前完成任务之FutureTask使用

### 21.1 示例

|  |
| --- |
| **public** **class** FutureTaskDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Callable<Integer> callable = **new** Callable<Integer>() {  @Override  **public** Integer call() **throws** Exception {  System.***out***.println("正在进行计算...");  Thread.*sleep*(3000);  **return** 1;  }  };    FutureTask<Integer> task = **new** FutureTask<>(callable);  Thread thread = **new** Thread(task);  thread.start();    System.***out***.println("干点别的...");    Integer result = task.get();  System.***out***.println("计算结果为：" + result);  }  } |

2）使用蛋糕工厂生产蛋糕

1.蛋糕类

|  |
| --- |
| **public** **class** Product {  **private** **int** id;  **private** String name;  **public** Product(**int** id, String name) {  System.***out***.println("开始生产蛋糕...");  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.id = id;  **this**.name = name;  System.***out***.println("蛋糕生产完毕");  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Product [id=" + id + ", name=" + name + "]";  }  } |

2.订单类，通过订单取蛋糕

|  |
| --- |
| **public** **class** Future {  **private** Product product;  **private** **boolean** done;    **public** **synchronized** Product getProduct() {  **while**(!done){  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **return** product;  }  **public** **synchronized** **void** setProduct(Product product) {  **if**(done){  **return**;  }    **this**.product = product;  done = **true**;  notifyAll();  }    } |

3.蛋糕工厂

|  |
| --- |
| **public** **class** ProductFactory {  **public** Future createProduct(){  System.***out***.println("下单成功，你可以去上班了...");  Future f = **new** Future();  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  Product p = **new** Product((**new** Random()).nextInt(), "蛋糕");  f.setProduct(p);  }  }).start();      **return** f;  }  } |

4.主线程

|  |
| --- |
| **public** **class** Main {  **public** **static** **void** main(String[] args) {  ProductFactory pf = **new** ProductFactory();    Future f = pf.createProduct();    System.***out***.println("我去上班了，下班过来拿蛋糕...");    System.***out***.println("下班了，我来取蛋糕" + f.getProduct());    }  } |

执行结果：

下单成功，你可以去上班了...

我去上班了，下班过来拿蛋糕...

开始生产蛋糕...

蛋糕生产完毕

下班了，我来取蛋糕Product [id=-1686904718, name=蛋糕]

## ForkJoin框架详解

### 22.1 循环求和与二分求和

1）循环求和

|  |
| --- |
| **public** **class** SumDemo **extends** RecursiveTask<Integer>{  **private** **int** begin;  **private** **int** end;    **public** SumDemo(**int** begin, **int** end) {  **this**.begin = begin;  **this**.end = end;  }  @Override  **protected** Integer compute() {  //System.out.println(Thread.currentThread().getName() + "...");  **int** sum = 0;  // 拆分任务  **if**(end - begin <= 2){  **for** (**int** i = begin; i <= end; i++) {  sum += i;  }  } **else** {  // 拆分  SumDemo sd1 = **new** SumDemo(begin, (begin+end)/2);  SumDemo sd2 = **new** SumDemo((begin + end)/2+1, end);    // 执行任务  sd1.fork();  sd2.fork();    // 合并  Integer a = sd1.join();  Integer b= sd2.join();    sum = a + b;  }  **return** sum;  }  **public** **static** **void** main(String[] args) **throws** Exception{  **long** startTime = System.*currentTimeMillis*();  /\*ForkJoinPool pool = new ForkJoinPool();    Future<Integer> future = pool.submit(new SumDemo(1,1000000000));    System.out.println("....");    System.out.println("计算结果为：" + future.get());\*/  **int** sum = 0 ;  **for** (**int** i = 0; i < 1000000000; i++) {  sum += i;  }  System.***out***.println("计算结果为：" + sum);  **long** endTime = System.*currentTimeMillis*();  System.***out***.println("程序运行时间：" + (endTime - startTime) + "ms");  }  } |

执行结果：

计算结果为：-1243309312

程序运行时间：381ms

2）二分求和

|  |
| --- |
| **public** **class** SumDemo **extends** RecursiveTask<Integer>{  **private** **int** begin;  **private** **int** end;    **public** SumDemo(**int** begin, **int** end) {  **this**.begin = begin;  **this**.end = end;  }  @Override  **protected** Integer compute() {  //System.out.println(Thread.currentThread().getName() + "...");  **int** sum = 0;  // 拆分任务  **if**(end - begin <= 2){  **for** (**int** i = begin; i <= end; i++) {  sum += i;  }  } **else** {  // 拆分  SumDemo sd1 = **new** SumDemo(begin, (begin+end)/2);  SumDemo sd2 = **new** SumDemo((begin + end)/2+1, end);    // 执行任务  sd1.fork();  sd2.fork();    // 合并  Integer a = sd1.join();  Integer b= sd2.join();    sum = a + b;  }  **return** sum;  }  **public** **static** **void** main(String[] args) **throws** Exception{  **long** startTime = System.*currentTimeMillis*();  ForkJoinPool pool = **new** ForkJoinPool();    Future<Integer> future = pool.submit(**new** SumDemo(1,1000000000));    System.***out***.println("....");    System.***out***.println("计算结果为：" + future.get());  /\*int sum = 0 ;  for (int i = 0; i < 1000000000; i++) {  sum += i;  }  System.out.println("计算结果为：" + sum);\*/  **long** endTime = System.*currentTimeMillis*();  System.***out***.println("程序运行时间：" + (endTime - startTime) + "ms");  }  } |

执行结果：

....

计算结果为：-243309312

程序运行时间：17073ms

分任务求和并不一定能提高程序得执行速度，反倒在拆分任务时会消耗大量的时间，一般适用于任务计算较慢，计算量比较大的情况，多核CPU的处理性能也比较好，可以尝试采用ForkJoin框架。单核CPU没有适用ForkJoin的必要。

## 同步容器与并发容器

### 23.1容器的类型

ArryList（线程安全）-->Vector（线程不安全）-->CopyOnWriteList（并发容器）

HashMap-->HashTable-->CopyOnWriteMap（并发容器，降低锁的粒度）

ArrayList是线程不安全的，Vector是线程安全的，但并发访问时不会用Vector，使用Collections工具类可以封装ArrayList为线程安全的容器，Collections.synchronizedList（arraylist）；同样HashMap线程不安全，HashTable是线程安全的，也可以使用Collections.synchronizedMap(hashmap)为线程安全的容器。

|  |
| --- |
| **public** **class** CollectionsDemo {  **public** **static** **void** main(String[] args) {  ArrayList<String> list = **new** ArrayList<>();  // 使用Collections封装一个代理，使得ArrayList线程安全；  Collections.*synchronizedList*(list);    HashMap<String, String> map = **new** HashMap<>();  // 使用Collections封装一个代理，使得HashMap线程安全；  Collections.*synchronizedMap*(map);  }  } |

并发写操作较多时，并发容器会因为拷贝消耗较大的内存，读较多时CopyOnWrite容器就比较适合了。

### 23.2并发容器非阻塞队列CopyOnWriteArrayList原理与使用

|  |
| --- |
| **public** **class** COWListDemo {  **public** **static** **void** main(String[] args) {  CopyOnWriteArrayList<String> list = **new** CopyOnWriteArrayList<>();  list.add("apple");  list.set(0, "banana");  list.get(0);  list.remove("banana");  System.***out***.println("list.size():" + list.size());  }  } |

增删改会使用ReenteredLock对代码块加锁，操作时复制一份原数组，在新数组进行增删改，操作完，将原数组的引用指向新数组的地址。

### 23.3并发容器ConcurrentLinkedQueue原理与使用

1. 单向链表的增加操作，head并非一定指向头结点，tail并非一定指向尾节点，添加队列操作。

|  |
| --- |
| **public** **boolean** offer(E e) {  *checkNotNull*(e);  **final** Node<E> newNode = **new** Node<E>(e);  **for** (Node<E> t = tail, p = t;;) {  Node<E> q = p.next;  **if** (q == **null**) {  // p是最后一个节点  **if** (p.casNext(**null**, newNode)) {  // Successful CAS is the linearization point  // for e to become an element of this queue,  // and for newNode to become "live".  // 成功的CAS是电子成为这个队列元素的线性化点，并为newNode成为“活”。  **if** (p != t) // hop two nodes at a time|一次跳两个节点，所以tail仅可能指向1,3,5...节点  casTail(t, newNode); // Failure is OK.  **return** **true**;  }  // Lost CAS race to another thread; re-read next  // 其他线程cas操作失败，重新读取下一个  }  **else** **if** (p == q)  // We have fallen off list. If tail is unchanged, it  // will also be off-list, in which case we need to  // jump to head, from which all live nodes are always  // reachable. Else the new tail is a better bet.  // 我们已经脱离了list。 如果tail没有改变，它也将是off-list，在这种情况下我们需要跳转到头部，从中可以始终到达所有活动节点。 否则新尾巴是更好的选择。  p = (t != (t = tail)) ? t : head;  **else**  // Check for tail updates after two hops.  // 两跳后检查尾部更新。  p = (p != t && t != (t = tail)) ? t : q;  }  } |

1. 移除队列操作

|  |
| --- |
| **public** E poll() {  restartFromHead:  **for** (;;) {  **for** (Node<E> h = head, p = h, q;;) {  E item = p.item;  **if** (item != **null** && p.casItem(item, **null**)) {  // Successful CAS is the linearization point  // for item to be removed from this queue.  // 从queue移除线性节点item表示cas操作成功  **if** (p != h) // hop two nodes at a time  updateHead(h, ((q = p.next) != **null**) ? q : p);  **return** item;  }  **else** **if** ((q = p.next) == **null**) {  updateHead(h, p);  **return** **null**;  }  **else** **if** (p == q)  **continue** restartFromHead;  **else**  p = q;  }  }  } |

## Java中的阻塞队列原理与使用

### 24.1阻塞队列BlockingQueue

阻塞：

Put

Take

抛出异常：

Add

Remove

有返回值：

Offer

Poll

### 24.2阻塞的Tmall商城队列

|  |
| --- |
| **public** **class** TmallBlockingQueue **implements** Shop{  // 库存  **private** **final** **int** MAX\_COUNT = 10;    **private** BlockingQueue<Integer> queue = **new** ArrayBlockingQueue<>(MAX\_COUNT);  **public** **void** push() {  **try** {  queue.put(1);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **public** **void** get() {  **try** {  queue.take();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  @Override  **public** **void** size() {  **while**(**true**) {  System.***out***.println("当前队列的长度：" + queue.size());  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }      } |

执行结果：

当前队列的长度：1

当前队列的长度：3

当前队列的长度：5

当前队列的长度：7

当前队列的长度：9

当前队列的长度：10

当前队列的长度：10

当前队列的长度：10

当前队列的长度：10

当前队列的长度：10

### 24.3阻塞队列源码

1. put(e)

|  |
| --- |
| **public** **void** put(E e) **throws** InterruptedException {  *checkNotNull*(e);  **final** ReentrantLock lock = **this**.lock;  lock.lockInterruptibly();  **try** {  **while** (count == items.length)  notFull.await();  enqueue(e);  } **finally** {  lock.unlock();  }  } |

2）

|  |
| --- |
| **public** E take() **throws** InterruptedException {  **final** ReentrantLock lock = **this**.lock;  lock.lockInterruptibly();  **try** {  **while** (count == 0)  notEmpty.await();  **return** dequeue();  } **finally** {  lock.unlock();  }  } |

## 线程池的原理与使用

### 25.1线程池使用

1. 线程池饱和策略：DiscardOldestPolicy

|  |
| --- |
| **public** **class** ThreadPoolDemo {  **public** **static** **void** main(String[] args) {  ThreadPoolExecutor threadPool = **new** ThreadPoolExecutor(10, 20, 10, TimeUnit.***DAYS***, **new** ArrayBlockingQueue<>(10), **new** DiscardOldestPolicy());  ThreadPoolExecutor threadPool1 = **new** ThreadPoolExecutor(10, 20, 10, TimeUnit.***DAYS***, **new** ArrayBlockingQueue<>(10), **new** ThreadPoolExecutor.CallerRunsPolicy());  AtomicInteger count = **new** AtomicInteger();  **for** (**int** i = 0; i < 100; i++) {  threadPool.execute(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  count.incrementAndGet();  }  });  }    threadPool.shutdown();    **while**(Thread.*activeCount*() > 1) {    }  System.***out***.println(count.get());  }  } |

执行结果：

....

pool-1-thread-6

pool-1-thread-6

pool-1-thread-6

pool-1-thread-8

pool-1-thread-11

pool-1-thread-2

pool-1-thread-3

pool-1-thread-9

pool-1-thread-10

pool-1-thread-7

pool-1-thread-5

pool-1-thread-18

pool-1-thread-19

pool-1-thread-20

pool-1-thread-17

pool-1-thread-12

pool-1-thread-1

pool-1-thread-4

pool-1-thread-13

pool-1-thread-15

pool-1-thread-16

pool-1-thread-14

64

线程并没有完全执行完，由于线程池的饱和策略进行处理。

1. 线程饱和策略：ThreadPoolExecutor.CallerRunsPolicy

|  |
| --- |
| **public** **class** ThreadPoolDemo {  **public** **static** **void** main(String[] args) {  ThreadPoolExecutor threadPool = **new** ThreadPoolExecutor(10, 20, 10, TimeUnit.***DAYS***, **new** ArrayBlockingQueue<>(10), **new** DiscardOldestPolicy());  ThreadPoolExecutor threadPool1 = **new** ThreadPoolExecutor(10, 20, 10, TimeUnit.***DAYS***, **new** ArrayBlockingQueue<>(10), **new** ThreadPoolExecutor.CallerRunsPolicy());  AtomicInteger count = **new** AtomicInteger();  **for** (**int** i = 0; i < 100; i++) {  threadPool1.execute(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  count.incrementAndGet();  }  });  }    threadPool1.shutdown();    **while**(Thread.*activeCount*() > 1) {    }  System.***out***.println(count.get());  }  } |

执行结果：

...

pool-2-thread-7

pool-2-thread-17

pool-2-thread-5

pool-2-thread-6

pool-2-thread-8

pool-2-thread-1

pool-2-thread-12

pool-2-thread-10

pool-2-thread-4

main

pool-2-thread-13

pool-2-thread-15

100

线程完全执行完毕。

## 第26节Executors创建线程池

### 26.1使用new ThreadPoolExecutor执行线程池参数创建

使用给定的初始参数和默认线程工厂以及拒绝策略创建一个新的ThreadPoolExecutor。使用一种Executors工厂方法来代替通用构造函数可能更方便。

参数：

corePoolSize要保留在池中的线程数，即使它们处于空闲状态，除非设置了allowCoreThreadTimeOut

maximumPoolSize池中允许的最大线程数

keepAliveTime当线程数大于核心时，这是多余空闲线程在终止之前等待新任务的最长时间。

单位keepAliveTime参数的时间单位

workQueue队列，用于在执行任务之前保存任务。此队列将仅保存execute方法提交的Runnable任务。

|  |
| --- |
| **public** **class** ExecutorDemo {  **public** **static** **void** main(String[] args) {  // 10个线程来处理大量任务，使用new ThreadPoolExecutor方式，指定线程池的参数  ThreadPoolExecutor pool = **new** ThreadPoolExecutor(10, 10, 0, TimeUnit.***MILLISECONDS***, **new** LinkedBlockingQueue<>());    **while** (**true**) {  pool.execute(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  });  }  }  } |

### 26.2 使用*newFixedThreadPool*创建线程池

创建一个线程池，该池重用在共享的无界队列中运行的固定数量的线程。 在任何时候，最多nThreads线程将是活动的处理任务。 如果在所有线程都处于活动状态时提交了其他任务，则它们将在队列中等待，直到线程可用。 如果任何线程由于在关闭之前执行期间的故障而终止，则如果需要执行后续任务，则新线程将取代它。 池中的线程将一直存在，直到它被明确关闭。

参数：

n更新池中的线程数

|  |
| --- |
| **public** **class** NewFixedThreadPoolDemo {  **public** **static** **void** main(String[] args) {  // 10个线程来处理大量任务，使用  // 使用Executors创建一个ExecutorService线程池，  ExecutorService pool = Executors.*newFixedThreadPool*(10);  **while** (**true**) {  pool.execute(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  });  }  }  } |

26.3使用*newCachedThreadPool*创建线程池

## 创建简单的web服务器

### 27.1单次请求到达

|  |
| --- |
| **public** **class** HttpServer {  **public** **static** **void** main(String[] args) **throws** IOException {  // 不停接收客户端的请求  ServerSocket server = **new** ServerSocket(8888);  System.***out***.println("服务端启动，监听" + server.getLocalPort() + "端口");  **while**(!Thread.*interrupted*()) {  // 不停的接收客户端的请求  Socket client = server.accept();    // 获取输入输出流  InputStream ins = client.getInputStream();  OutputStream ous = client.getOutputStream();    // 读取请求内容  /\*int len = 0;  byte[] b = new byte[1024];  len = ins.read(b);  System.out.println(new String(b,0,len));\*/    BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(ins));  String line = bf.readLine();  System.***out***.println(line);    // 给用户响应  PrintWriter pw = **new** PrintWriter(ous);  InputStream in = **new** FileInputStream("E:\\webroot\\index.html");    BufferedReader bf1 = **new** BufferedReader(**new** InputStreamReader(in));    pw.println("HTTP/1.1 200 OK");  pw.println("Content-Type: text/html;chartset=utf-8");  pw.println("Content-length: " + in.available());  pw.println("Server: hello");  pw.println();  pw.flush();    String c = **null**;  **while**((c = bf1.readLine()) != **null**) {  pw.print(c);  }  pw.flush();    bf1.close();  pw.close();  bf.close();  client.close();    }    server.close();    }  } |

### 27.2 多次请求到达

|  |
| --- |
| **public** **class** HttpServer2 {  **public** **static** **void** main(String[] args) **throws** IOException {  // 不停接收客户端的请求  ServerSocket server = **new** ServerSocket(8888);  System.***out***.println("服务端启动，监听" + server.getLocalPort() + "端口");  **while**(!Thread.*interrupted*()) {  // 不停的接收客户端的请求  Socket client = server.accept();    **new** Thread(**new** ServerThread(client)).start();  }    server.close();    }  } |

线程任务

|  |
| --- |
| **public** **class** ServerThread **implements** Runnable {  **private** Socket client;  **private** InputStream ins;  **private** OutputStream ous;  **private** PrintWriter pw;  **private** BufferedReader br;  **public** ServerThread(Socket client) {  **this**.client = client;  init();  }  **private** **void** init() {  // 获取输入输出流  **try** {  InputStream ins = client.getInputStream();  OutputStream ous = client.getOutputStream();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  @Override  **public** **void** run() {  **try** {  go();  } **catch** (Exception e) {  e.printStackTrace();  }  }  **private** **void** go() **throws** Exception {  // 读取请求内容  /\*  \* int len = 0; byte[] b = new byte[1024]; len = ins.read(b);  \* System.out.println(new String(b,0,len));  \*/  BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(ins));  String line = bf.readLine();  System.***out***.println(line);  // 给用户响应  PrintWriter pw = **new** PrintWriter(ous);  InputStream in = **new** FileInputStream("E:\\webroot\\index.html");  BufferedReader bf1 = **new** BufferedReader(**new** InputStreamReader(in));  pw.println("HTTP/1.1 200 OK");  pw.println("Content-Type: text/html;chartset=utf-8");  pw.println("Content-length: " + in.available());  pw.println("Server: hello");  pw.println();  pw.flush();  String c = **null**;  **while** ((c = bf1.readLine()) != **null**) {  pw.print(c);  }  pw.flush();  bf1.close();  pw.close();  bf.close();  client.close();  }  } |

## 第27节JDK8的新增原子操作类LongAddr原理与使用



## 第28节JDK8新增锁StampedLock锁

·ReadWriteLock写锁是互斥的

·读-写

·写-写

并发高的情况，可能读线程较多，有可能导致写线程饥饿；

·StampedLock

·读锁并不会阻塞写锁；（读写不互斥，写的时候可以读）

## 第29节happens-before

1. happens-before

·用来指定两个操作之间的执行顺序。提供跨线程的内存可见性。

·在Java内存模型中，如果一个操作执行的结果对另一个操作可见，那么这两个操作之间必然存在happens-before关系。

·happens-before规则如下

·程序顺序规则

·监视器锁规则

·volatile变量规则

·

· （2）程序顺序规则