# 第15天【线程通信和线程池】

## 主要内容

1. 线程通信
2. 线程池ThreadPoolExecutor
3. ForkJoin

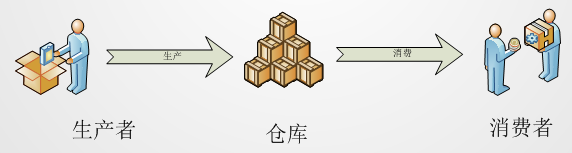
## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 线程通信引入 | 理解 |
| 使用同步代码块实现线程通信 | 掌握 |
| 使用同步方法实现线程通信 | 掌握 |
| 使用Lock锁实现线程通信 | 掌握 |
| Condition | 理解 |
| 线程池ThreadPoolExecutor引入 | 掌握 |
| 使用线程池执行大量的Runnable命令 | 掌握 |
| 使用线程池执行大量的Callable任务 | 掌握 |
| 线程池API | 理解 |
| ForkJoin框架 | 了解 |

## 一、线程通信

### 1.1 线程通信引入 线程同步

* **应用场景：生产者和消费者问题**
  + 假设仓库中只能存放一件产品，生产者将生产出来的产品放入仓库，消费者将仓库中产品取走消费
  + 如果仓库中没有产品，则生产者将产品放入仓库，否则停止生产并等待，直到仓库中的产品被消费者取走为止
  + 如果仓库中放有产品，则消费者可以将产品取走消费，否则停止消费并等待，直到仓库中再次放入产品为止



* **分析**
  + 这是一个线程同步问题，生产者和消费者共享同一个资源，并且生产者和消费者之间相互依赖，互为条件
  + 对于生产者，没有生产产品之前，要通知消费者等待。而生产了产品之后，又需要马上通知消费者消费
  + 对于消费者，在消费之后，要通知生产者已经消费结束，需要继续生产新产品以供消费
  + 在生产者消费者问题中，仅有线程同步是不够的
    - 线程同步可阻止并发更新同一个共享资源，实现了同步
    - 线程同步不能用来实现不同线程之间的消息传递（通信）
* **Java提供了3个方法解决线程之间的通信问题**

|  |  |
| --- | --- |
| 方法名 | 作 用 |
| final void ·() | 表示线程一直等待，直到其它线程通知 |
| void wait(long timeout) | 线程等待指定毫秒参数的时间 |
| final void wait(long timeout,int nanos) | 线程等待指定毫秒、微妙的时间 |
| final void notify() | 唤醒一个处于等待状态的线程 |
| final void notifyAll() | 唤醒同一个对象上所有调用wait()方法的线程，优先级别高的线程优先运行 |

|  |
| --- |
| **注意事项**   * 均是java.lang.Object类的方法 * 都只能在同步方法或者同步代码块中使用，否则会抛出异常 |

#### 【示例1】 准备生产者消费者问题

|  |
| --- |
| /\*\*  \* 商品类  \*/ **public class** Product {  **private** String **name**;//馒头、玉米饼  **private** String **color**;//白色 黄色  **public** Product() {  }  **public** Product(String name, String color) {  **this**.**name** = name;  **this**.**color** = color;  }  //…..省略getter和setter方法  **public** String toString() {  **return "Product{"** +  **"name='"** + **name** + **'\''** +  **", color='"** + **color** + **'\''** +  **'}'**;  } }  /\*\*  \* 生产者线程  \*/ **public class** ProduceRunnable **implements** Runnable {  //private Product product = new Product();  **private** Product **product**;  **public void** setProduct(Product product) {  **this**.**product** = product;  }  **public void** run() {  **int** i = 0;  **while**(**true**){  **if**(i%2==0){  **product**.setName(**"馒头"**);  **product**.setColor(**"白色"**);  }**else**{  **product**.setName(**"玉米饼"**);  **product**.setColor(**"黄色"**);  }  System.**out**.println(**"生产者生产商品"**+**product**.getName()  +**" "**+**product**.getColor());  i++;  }  } }  /\*\*  \* 消费者线程  \*/ **public class** ConsumeRunnable **implements** Runnable {  //private Product product = new Product();  **private** Product **product**;  **public** ConsumeRunnable() {  }  **public** ConsumeRunnable(Product product) {  **this**.**product** = product;  }  **public void** setProduct(Product product) {  **this**.**product** = product;  }  **public void** run() {  **while**(**true**){  System.**out**.println(**"消费者消费商品"**+**product**.getName()  +**" "**+**product**.getColor());  }  } }  **public class** Test {  **public static void** main(String[] args) {  Product product = **new** Product();  ProduceRunnable runnable1 = **new** ProduceRunnable();  runnable1.setProduct(product);  Thread thread1 = **new** Thread(runnable1);   Runnable runnable2 = **new** ConsumeRunnable(product);  Thread thread2 = **new** Thread(runnable2);  thread1.start();  thread2.start();  } } |

注意：必须保证生产者消费者操作的是一个商品对象，否则会出现消费者消费null的情况。

|  |
| --- |
| Product product = **new** Product();  ProduceRunnable runnable1 = **new** ProduceRunnable(); runnable1.setProduct(product); Runnable runnable2 = **new** ConsumeRunnable(product); |

### 1.2使用同步代码块实现线程同步

#### 【示例2】 使用同步代码块保证线程安全

|  |
| --- |
| **public class** ProduceRunnable **implements** Runnable {  **private** Product **product**;  **public void** setProduct(Product product) {  **this**.**product** = product;  }  **public void** run() {  **int** i = 0;  **while**(**true**){  **synchronized** (**product**){  **if**(i%2==0){  **product**.setName(**"馒头"**);  **try** {  Thread.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setColor(**"白色"**);  }**else**{  **product**.setName(**"玉米饼"**);  **try** {  Thread.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setColor(**"黄色"**);  }  System.**out**.println(**"生产者生产商品"**+**product**.getName()  +**" "**+**product**.getColor());  }  i++;  }  } }  **public class** ConsumeRunnable **implements** Runnable {  **private** Product **product**;  **public void** run() {  **while**(**true**){  **synchronized** (**product**){  System.**out**.println(**"消费者消费商品"**  +**product**.getName()+**" "**+**product**.getColor());  }  }  } } |

* 注意：不仅生产者要加锁，而且消费者也要加锁，并且必须是一把锁（不仅是一个引用变量，而且必须是指向同一个对象）

### 1.3 同步代码块下的线程通信

#### 【示例3】 实现线程通信—同步代码块方式

|  |
| --- |
| **public class** ProduceRunnable **implements** Runnable {  **public void** run() {  **int** i = 0;  **while**(**true**){  **synchronized** (**product**){  //如果已经有商品，就等待  **if**(**product**.**flag**){  **try** {  **product**.wait(); //必须调用同步监视器的通信方法  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //生产商品  **if**(i%2==0){  **product**.setName(**"馒头"**);  **try** {  Thread.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setColor(**"白色"**);  }**else**{  **product**.setName(**"玉米饼"**);  **try** {  Thread.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setColor(**"黄色"**);  }  //输出结果  System.**out**.println(**"生产者生产商品"**+**product**.getName()  +**" "**+**product**.getColor());  //修改一下商品的状态  **product**.**flag** = **true**;  //通知消费者来消费  **product**.notify();  }  i++;  }  } }  **public class** ConsumeRunnable **implements** Runnable {  **public void** run() {  **while**(**true**){  **synchronized** (**product**){  //如果没有商品，就等待  **if**(!**product**.**flag**){  **try** {  **product**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //消费商品  System.**out**.println(**"消费者消费商品"** +**product**.getName()+**" "**+**product**.getColor());  //修改商品的状态  **product**.**flag** = **false**;  //通知生产者进行生产  **product**.notifyAll();  }  }  } } |

**线程通信的细节  
细节1：进行线程通信的多个线程，要使用同一个同步监视器（product），还必须要调用该同步监视器的wait()、notify()、notifyAll();**

**细节2：线程通信的三个方法**

**wait() 等待**

在【其他线程】调用【此对象】的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。换句话说，此方法的行为就好像它仅执行 wait(0) 调用一样。当前线程必须拥有此对象监视器。

**wait(time) 等待**

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。 当前线程必须拥有此对象监视器。

**notify() 通知 唤醒**

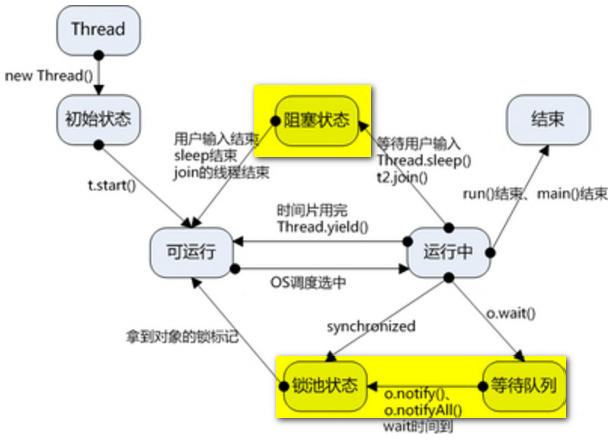
唤醒在【此对象监视器】上等待的【单个】线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。【选择是任意性的】，并在对实现做出决定时发生

**notifyAll() 通知所有 唤醒所有**

唤醒在【此对象监视器】上等待的【所有】线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程【进行竞争】；

**细节3：完整的线程生命周期** 阻塞状态有三种

* 普通的阻塞 sleep,join,Scanner input.next()
* 同步阻塞（锁池队列） 没有获取同步监视器的线程的队列
* 等待阻塞（阻塞队列） 被调用了wait()后释放锁，然后进行该队列



**细节4：sleep()和wait()的区别**

* 区别1:sleep()：线程会让出CPU进入阻塞状态，但不会释放对象锁；wait()：线程会让出CPU进入阻塞状态，也会放弃对象锁，进入等待此对象的等待锁定池
* 区别2：wait只能在同步控制方法或同步控制块里面使用，而sleep可以在任何地方使用

### 本节作业

1. 使用同步代码块方式实现生产者-消费者问题
2. sleep()和wait()的区别
3. 线程的完整的生命周期

## 二、线程通信

### 2.1 同步方法下实现线程通信

#### 【示例4】实现线程通信-使用同步方法方式

|  |
| --- |
| **public class** Product {  **private** String **name**;//馒头、玉米饼  **private** String **color**;//白色 黄色  **boolean flag** = **false**;//默认没有商品  **public synchronized void** produce(String name, String color) {//this  //如果已经有商品，就等待  **if** (**flag**) {  **try** {  //让出了CPU，会同时释放锁  **this**.wait(); //必须调用同步监视器的通信方法  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //生产商品  **this**.**name** = name;  **try** {  Thread.sleep(1); //让出了CPU,不释放锁  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.**color** = color;  //输出结果  System.**out**.println(**"生产者生产商品"** + getName()+**" "**+getColor());  //修改一下商品的状态  **flag** = **true**;  //通知消费者来消费  **this**.notify();  }  **public synchronized void** consume() {//this  //如果没有商品，就等待  **if**(!**flag**){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //消费商品  System.**out**.println(**"消费者消费商品"** +**name**+**" "**+**color**);  //修改商品的状态  **flag** = **false**;  //通知生产者进行生产  **this**.notifyAll();  }  **public class** ProduceRunnable **implements** Runnable {  **public void** run() {  **int** i = 0;  **while**(**true**){  **if**(i%2==0){  **product**.produce(**"馒头"**,**"白色"**);  }**else**{  **product**.produce(**"玉米饼"**,**"黄色"**);  }  i++;  }  } }  **public class** ConsumeRunnable **implements** Runnable {  **public void** run() {  **while**(**true**){  **product**.consume();  }  } } |

* 同步方法的同步监视器都是this，所以需要将produce()和consume()放入一个类Product中，保证是同一把锁
* 必须调用this的wait()、notify()、notifyAll()方法，this可以省略，因为同步监视器是this。

### 2.2 Lock锁下实现线程通信

之前实现线程通信时，是生产者和消费者在一个等待队列中，会存在本来打算唤醒消费者，却唤醒一个生产者的问题，能否让生产者和消费者线程在不同的队列中等待呢？在新一代的Lock中提供这种实现方式。

#### 【示例5】实现线程通信-使用Lock锁

|  |
| --- |
| **public class** Product {  **private** String **name**;//馒头、玉米饼  **private** String **color**;//白色 黄色  **boolean flag** = **false**;//默认没有商品  Lock **lock** = **new** ReentrantLock();  Condition **produceCondtion** = **lock**.newCondition();  Condition **consumeCondition** = **lock**.newCondition();   **public void** produce(String name, String color) {//this  **lock**.lock();  **try**{  //如果已经有商品，就等待  **if** (**flag**) {  **try** {  **produceCondtion**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //生产商品  **this**.**name** = name;  **try** {  Thread.sleep(1); //让出了CPU,不释放锁  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.**color** = color;  //输出结果  System.**out**.println(**"生产者生产商品"** + getName()  + **" "** + getColor());  //修改一下商品的状态  **flag** = **true**;  //通知消费者来消费  **consumeCondition**.signal();  }**finally** {  **lock**.unlock();  }  }  **public void** consume() {//this  **lock**.lock();  **try**{  //如果没有商品，就等待  **if**(!**flag**){  **try** {  **consumeCondition**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  //消费商品  System.**out**.println(**"消费者消费商品"** +**name**+**" "**+**color**);  //修改商品的状态  **flag** = **false**;  //通知生产者进行生产  **produceCondtion**.signalAll();  }**finally** {  **lock**.unlock();  }  } } |

### 2.3 Condition

Condition是在[Java](http://lib.csdn.net/base/java" \t "_blank)1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。

它的更强大的地方在于：能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition。**一个Condition包含一个等待队列。一个Lock可以产生多个Condition，所以可以有多个等待队列**。

在Object的监视器模型上，一个对象拥有一个同步队列和等待队列，而Lock（同步器）**拥有一个同步队列和多个等待队列。**

Object中的wait(),notify(),notifyAll()方法是和"同步锁"(synchronized关键字)捆绑使用的；而Condition是需要与"互斥锁"/"共享锁"（Lock）捆绑使用的。

调用Condition的await()、signal()、signalAll()方法，都必须在lock保护之内，**就是说必须在lock.lock()和lock.unlock之间才可以使用**

* Conditon中的await()对应Object的wait()；
* Condition中的signal()对应Object的notify()；
* Condition中的signalAll()对应Object的notifyAll()。

void **await**() throws [InterruptedException](../../../../java/lang/InterruptedException.html" \o "java.lang 中的类)

造成当前线程在接到信号或被[中断](../../../../java/lang/Thread.html" \l "interrupt())之前一直处于等待状态。

与此 Condition 相关的锁以原子方式释放，并且出于线程调度的目的，将禁用当前线程，且在发生以下四种情况*之一* 以前，当前线程将一直处于休眠状态：

* 其他某个线程调用此 Condition 的 [signal()](../../../../java/util/concurrent/locks/Condition.html" \l "signal()) 方法，并且碰巧将当前线程选为被唤醒的线程；或者
* 其他某个线程调用此 Condition 的 [signalAll()](../../../../java/util/concurrent/locks/Condition.html" \l "signalAll()) 方法；或者
* 其他某个线程[中断](../../../../java/lang/Thread.html" \l "interrupt())当前线程，且支持中断线程的挂起；或者
* 发生“虚假唤醒”

在所有情况下，在此方法可以返回当前线程之前，都必须重新获取与此条件有关的锁。在线程返回时，可以保证它保持此锁。

void **signal**()

唤醒一个等待线程。

如果所有的线程都在等待此条件，则选择其中的一个唤醒。在从 await 返回之前，该线程必须重新获取锁。

void **signalAll**()

唤醒所有等待线程。

如果所有的线程都在等待此条件，则唤醒所有线程。在从 await 返回之前，每个线程都必须重新获取锁。

### 2.4 LockSupport锁下线程通信

 LockSupport优势，优势1：不需要再锁中进行；优势二：park方法可以不先于unpark方法。



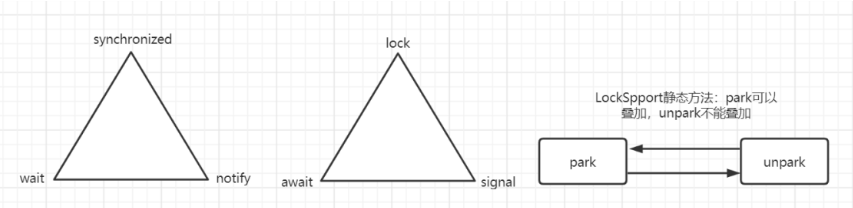
|  |
| --- |
| **public class** Product {   **private** String **color**;*// 白色 黄色* **private** String **name**;*// 馒头 玉米饼* **private boolean flag**=**false**;*//库存信号 默认的是没有库存* **private** ProduceRunnable **produceRunnable**;  **private** ConsumeRunnable **consumeRunnable**;  **public** ProduceRunnable getProduceRunnable() {  **return produceRunnable**;  }   **public void** setProduceRunnable(ProduceRunnable produceRunnable) {  **this**.**produceRunnable** = produceRunnable;  }   **public** ConsumeRunnable getConsumeRunnable() {  **return consumeRunnable**;  }   **public void** setConsumeRunnable(ConsumeRunnable consumeRunnable) {  **this**.**consumeRunnable** = consumeRunnable;  }  **public** Product(){}   **public** Product(String color, String name) {  **this**.**color** = color;  **this**.**name** = name;  }  **public boolean** isFlag() {  **return flag**;  }   **public void** setFlag(**boolean** flag) {  **this**.**flag** = flag;  }  **public** String getColor() {  **return color**;  }   **public void** setColor(String color) {  **this**.**color** = color;  }  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }   @Override  **public** String toString() {  **return "Product{"** +  **"color='"** + **color** + **'\''** +  **", name='"** + **name** + **'\''** +  **'}'**;  } } |
| */\*\*  \** ***@author*** *: 尚学堂  \** ***@date*** *: 14:28 2021/4/26  \*  \* 基于LockSupport锁下实现线程通信：  \*  \* 不需要加锁  \* park() 等待  \* unpark(Thread th) 唤醒  \*  \*  \*/* **public class** Test01 {   **public static void** main(String[] args) {  Product product=**new** Product();   ProduceRunnable produceRunnable = **new** ProduceRunnable();  ConsumeRunnable consumeRunnable = **new** ConsumeRunnable();  product.setProduceRunnable(produceRunnable);  product.setConsumeRunnable(consumeRunnable);  produceRunnable.setProduct(product);  consumeRunnable.setProduct(product);  *//开启生产者线程* produceRunnable.start();  *//开启消费者线程* consumeRunnable.start();    } }  *//生产者类* **class** ProduceRunnable **extends** Thread {  **private** Product **product**;   **public void** setProduct(Product product) {  **this**.**product** = product;  }  @Override  **public void** run() {   **int** num=0;   **while** (**true**){   **if**(**product**.isFlag()){  *//证明当前有库存，使当前线程处于等待状态* LockSupport.*park*();  }  **if**(num%2==0){  **product**.setColor(**"白色"**);  **try** {  Thread.*sleep*(5);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setName(**"馒头"**);  }**else**{  **product**.setColor(**"黄色"**);  **try** {  Thread.*sleep*(5);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **product**.setName(**"玉米饼"**);  }   System.***out***.println(**"生产者生产了:"**+**product**.getColor()+**" "**+**product**.getName());  num++;   *//生产结束* **product**.setFlag(**true**);  *//唤醒消费者线程* LockSupport.*unpark*(**product**.getConsumeRunnable());  }  } }  *//消费者线程* **class** ConsumeRunnable **extends** Thread{  *//这样操作的目的就是保证生产者和消费者使用的同一件商品* **private** Product **product**;   **public void** setProduct(Product product) {  **this**.**product** = product;  }   @Override  **public void** run() {   **while** (**true**){  **if**(!**product**.isFlag()){  LockSupport.*park*();  }  System.***out***.println(**"消费者消费了:"**+**product**.getColor()+**" "**+**product**.getName());   **product**.setFlag(**false**);  *//唤醒生产者生产* LockSupport.*unpark*(**product**.getProduceRunnable());  }  } } |

三种线程通信方式对比

① 对于Object中wait和notify：必须在锁中使用，这里指synchronized，并且wait方法要先于notify，否则wait方法会一直阻塞该该线程。

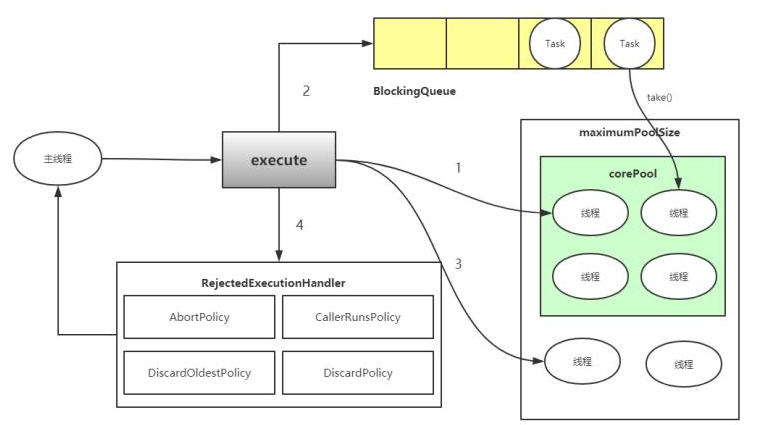
② Lock中lock.newCondition:condition.await()和condition.signal()：必须在锁中使用，这里指lock.lock()之后，并且await方法要先于singnal，否则await方法会一直阻塞该该线程。

③ LockSupport优势，优势1：不需要再锁中进行；优势二：park方法可以不先于unpark方法。



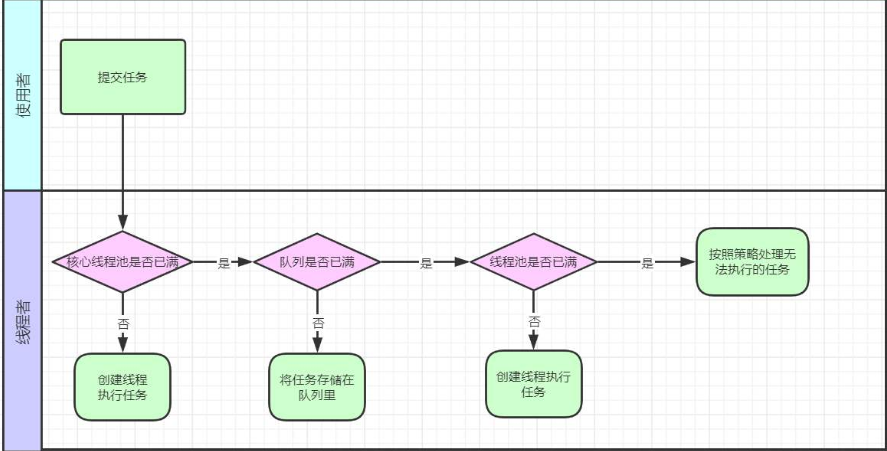
## 三、线程池

### 3.1线程池ThreadPoolExecutor引入



**什么是线程池**

* + 创建和销毁对象是非常耗费时间的
  + 创建对象：需要分配内存等资源
  + 销毁对象：虽然不需要程序员操心，但是垃圾回收器会在后台一直跟踪并销毁
  + 对于经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
  + 思路：创建好多个线程，放入线程池中，使用时直接获取引用，不使用时放回池中。可以避免频繁创建销毁、实现重复利用
  + 生活案例：在尚学堂借用和归还电脑，共享单车
  + 技术案例：线程池、数据库连接池
  + JDK1.5起，提供了内置线程池



**线程池的好处**

* + 提高响应速度（减少了创建新线程的时间）
  + 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  + 提高线程的可管理性：避免线程无限制创建、从而销耗系统资源，降低系统稳定性，甚至内存溢出或者CPU耗尽

**线程池的应用场合**

* + 需要大量线程，并且完成任务的时间短
  + 对性能要求苛刻
  + 接受突发性的大量请求

### 3.2使用线程池执行大量的Runnable命令

#### 【示例6】使用线程池执行大量的Runnable命令

|  |
| --- |
| **public class** TestThreadPool1 {  **public static void** main(String[] args) {  //创建一个线程池  //创建一个线程池，池中只有1个线程，保证线程一直存在  //ExecutorService pool = Executors.newSingleThreadExecutor();  //创建一个线程池，池中有固定数量的线程  //ExecutorService pool = Executors.newFixedThreadPool(10);  //创建一个线程池，池中线程的数量可以动态变化  ExecutorService pool = Executors.newCachedThreadPool();  //使用线程池执行大量的Runnable命令  **for**(**int** i=0;i<20;i++){  **final int** n = i;  //指定Runnable命令  Runnable command = **new** Runnable(){  **public void** run() {  System.**out**.println(**"开始执行"**+n);  **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.**out**.println(**"执行结束"**+n);  }   };  //不需要new Thread(command);  //使用线程池执行命令  pool.execute(command);  }  //关闭线程池  pool.shutdown();  } } //class MyRunnable implements Runnable{ // public void run() { // System.out.println("开始执行"); // System.out.println("执行结束"); // } //} |

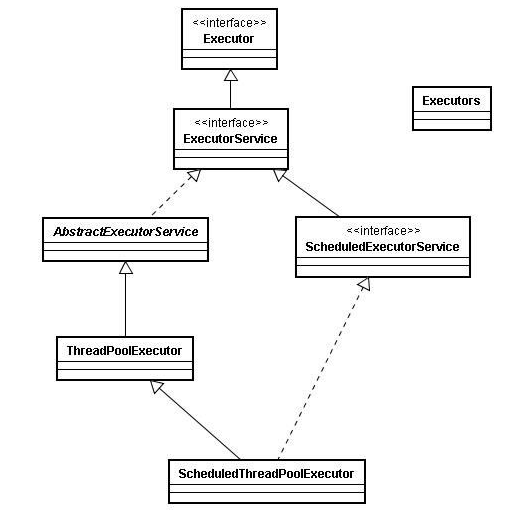
### 3.3使用线程池执行大量的Callable任务

#### 【示例7】使用线程池执行大量的Callable任务

|  |
| --- |
| **public class** TestThreadPool2 {  **public static void** main(String[] args)   **throws** InterruptedException, ExecutionException {  //创建线程池  ExecutorService pool = Executors.newFixedThreadPool(10);  //使用线程池  List<Future> list = **new** ArrayList<Future>();  **for**(**int** i=0;i<20000;i++){  //指定一个Callable任务  Callable<Integer> task = **new** MyCallable();  //将任务交给线程池  Future<Integer> future = pool.submit(task);   //将Future加入到List  list.add(future);  }  System.**out**.println(**"ok?"**);  //遍历并输出多个结果  **for**(Future f:list){  System.**out**.println(f.get());  }   System.**out**.println(**"OK!"**);  //关闭线程池  pool.shutdown();  } } **class** MyCallable **implements** Callable<Integer>{  **public** Integer call() **throws** Exception {  Thread.sleep(2000);   **return new** Random().nextInt(10);  }  } |

### 3.4线程池API 总结

* Executor：线程池顶级接口，只有一个方法
* ExecutorService：真正的线程池接口
  + void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable
  + Futuresubmit(Callable task)：执行任务，有返回值，一般又来执行Callable
  + void shutdown() ：关闭线程池
* AbstractExecutorService：基本实现了ExecutorService的所有方法



* **ThreadPoolExecutor：默认的线程池实现类**
* ScheduledThreadPoolExecutor：实现周期性任务调度的线程池
* Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池
  + Executors.newCachedThreadPool()：创建可根据需要创建新线程的线程池
  + Executors.newFixedThreadPool(n); 创建一个可重用固定线程数的线程池
  + Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池
  + Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。
* 线程池ThreadPoolExecutor参数

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

* + corePoolSize：核心池的大小

默认情况下，创建了线程池后，线程数为0，当有任务来之后，就会创建一个线程去执行任务。

但是当线程池中线程数量达到corePoolSize，就会把到达的任务放到队列中等待。

* + maximumPoolSize：最大线程数。

corePoolSize和maximumPoolSize之间的线程数会自动释放，小于等于corePoolSize的不会释放。当大于了这个值就会将任务由一个丢弃处理机制来处理。

* + keepAliveTime：线程没有任务时最多保持多长时间后会终止

默认只限于corePoolSize和maximumPoolSize间的线程

* + TimeUnit：

keepAliveTime的时间单位

* + BlockingQueue：

存储等待执行的任务的阻塞队列，有多种选择，可以是顺序队列、链式队列等。

* + ThreadFactory

线程工厂，默认是DefaultThreadFactory，Executors的静态内部类

* + RejectedExecutionHandler：

拒绝处理任务时的策略。如果线程池的线程已经饱和，并且任务队列也已满，对新的任务应该采取什么策略。

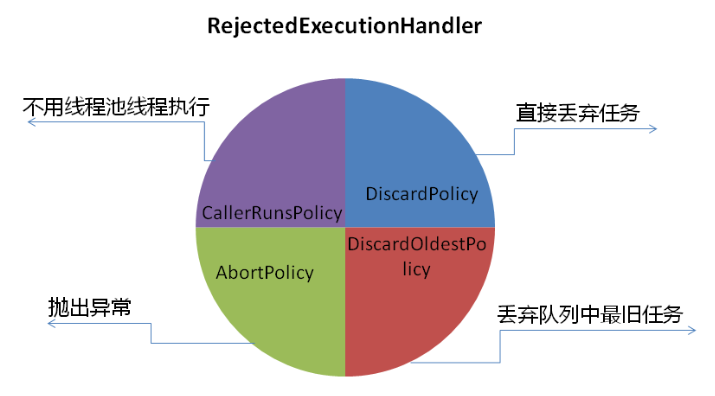
比如抛出异常、直接舍弃、丢弃队列中最旧任务等，默认是直接抛出异常。

1、CallerRunsPolicy：如果发现线程池还在运行，就直接运行这个线程

2、DiscardOldestPolicy：在线程池的等待队列中，将头取出一个抛弃，然后将当前线程放进去。

3、DiscardPolicy：什么也不做

4、AbortPolicy：java默认，抛出一个异常

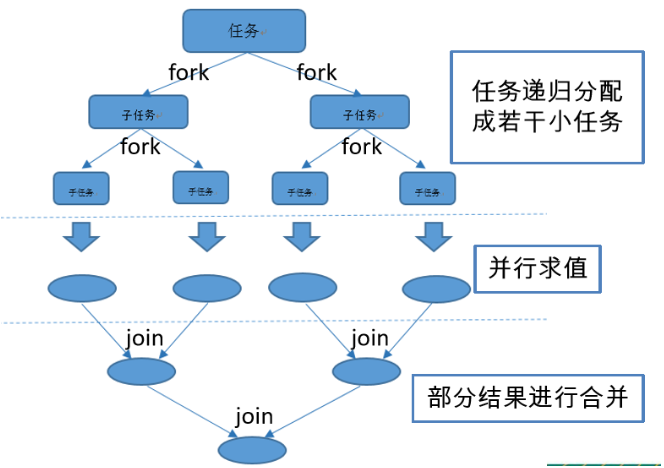


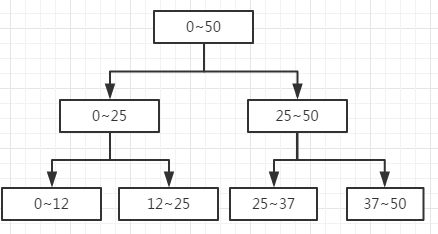
### 3.5 ForkJoin框架

**1. 什么是ForkJoin框架 适用场景**

虽然目前处理器核心数已经发展到很大数目，但是按任务并发处理并不能完全充分的利用处理器资源，因为一般的应用程序没有那么多的并发处理任务。基于这种现状，考虑把一个任务拆分成多个单元，每个单元分别得到执行，最后合并每个单元的结果。

Fork/Join框架是JAVA7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干小任务，最终汇总每个小任务结果得到大任务结果的框架。



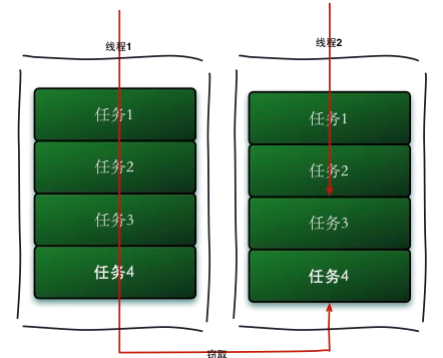


**2.工作窃取算法（work-stealing）**

一个大任务拆分成多个小任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列中，并且每个队列都有单独的线程来执行队列里的任务，线程和队列一一对应。

但是会出现这样一种情况：A线程处理完了自己队列的任务，B线程的队列里还有很多任务要处理。

A是一个很热情的线程，想过去帮忙，但是如果两个线程访问同一个队列，会产生竞争，所以A想了一个办法，从双端队列的尾部拿任务执行。而B线程永远是从双端队列的头部拿任务执行。



注意：线程池中的每个线程都有自己的工作队列（PS，这一点和ThreadPoolExecutor不同，ThreadPoolExecutor是所有线程公用一个工作队列，所有线程都从这个工作队列中取任务），当自己队列中的任务都完成以后，会从其它线程的工作队列中偷一个任务执行，这样可以充分利用资源。

**工作窃取算法的优点：**

利用了线程进行并行计算，减少了线程间的竞争。

**工作窃取算法的缺点：**

     1、如果双端队列中只有一个任务时，线程间会存在竞争。

     2、窃取算法消耗了更多的系统资源，如会创建多个线程和多个双端队列。

**3.主要类**

* **ForkJoinTask**：

使用该框架，需要创建一个ForkJoin任务，它提供在任务中执行fork和join操作的机制。一般情况下，我们并不需要直接继承ForkJoinTask类，只需要继承它的子类，它的子类有两个：

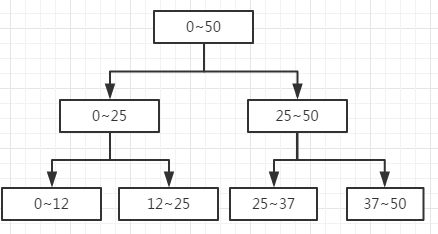
* RecursiveAction:用于没有返回结果的任务。
* RecursiveTask:用于有返回结果的任务。
* fork()方法：将任务放入队列并安排异步执行
* join()方法：等待计算完成并返回计算结果。
* **ForkJoinPool：**

任务ForkJoinTask需要通过ForkJoinPool来执行。

* **ForkJoinWorkerThread：**

ForkJoinPool线程池中的一个执行任务的线程。

**4. ForkJoin框架示例**

****

#### 【示例8】使用ForkJoin框架计算 1+2+3......+n = ?

|  |
| --- |
| **public class** SumTask **extends** RecursiveTask<Long> {  **private int start**;  **private int end**;  **private final int step** = 2000000;*//最小拆分成几个数相加* **public** SumTask(**int** start, **int** end) {  **this**.**start** = start;  **this**.**end** = end;  }  @Override  **protected** Long compute() {  **long** sum = 0;  **if**(**end** - **start** <= **step** ){  *//小于5个数，直接求和* **for** (**int** i = **start**; i <=**end**; i++) {  sum+=i;  }  }**else**{  *//大于5个数，分解任务* **int** mid = (**end** + **start**)/2;  SumTask leftTask = **new** SumTask(**start**,mid);  SumTask rightTask = **new** SumTask(mid+1,**end**);  *//执行子任务* leftTask.fork();  rightTask.fork();  *//子任务，执行完，得到执行结果* **long** leftSum = leftTask.join();  **long** rightSum = rightTask.join();  sum = leftSum+rightSum;  }  **return** sum;  }  **public static void** main(String[] args)  **throws** ExecutionException, InterruptedException {  *//如果多核CPU，其实是一个一直使用，其他闲置；怎么办，多线程解决；  //但是涉及到任务的拆分与合并等众多细节，不要紧，*  *//现在使用ForkJoin框架，可以较轻松解决；* **long** start = System.*currentTimeMillis*();  **long** sum = 0;  **for**(**int** i=0;i<=1000000000;i++){  sum +=i;  }  System.***out***.println(sum);  **long** end = System.*currentTimeMillis*();  System.***out***.println(**"for:"**+(end - start));  *//使用ForkJoin框架解决  //创建一个线程池* ForkJoinPool pool = **new** ForkJoinPool();  *//定义一个任务* SumTask sumTask = **new** SumTask(1,1000000000);  *//将任务交给线程池* start = System.*currentTimeMillis*();  Future<Long> future = pool.submit(sumTask);  *//得到结果并输出* Long result = future.get();  System.***out***.println(result);  end = System.*currentTimeMillis*();  System.***out***.println(**"pool:"**+(end - start));  } } |

可以看出，使用了 ForkJoinPool 的实现逻辑全部集中在了 compute() 这个函数里，仅用了很少行就实现了完整的计算过程。特别是，在这段代码里没有显式地“把任务分配给线程”，只是分解了任务，而把具体的任务到线程的映射交给了 ForkJoinPool 来完成。

### 本节作业

1. 线程池的特点、优点和适用场合
2. 实现使用线程池执行大量Runnable命令
3. 实现使用线程池执行大量Callable任务
4. 线程池ThreadPoolExecutor的默认参数有哪些，含义是什么
5. 线程池拒绝处理任务时的策略有哪些
6. 说明ForkJoin所采用的工作窃取算法（双端队列 偷偷做好事）
7. 使用ForKJoin计算1+2+3……..+n的和