# 并发基础

## 什么场景下使用多线程

多线程解决的是等待问题。

1、cpu密集型的计算速度慢的数据，通过并行来提高计算速度。

2、等待网络I/O比较耗时的操作，通过异步减少网络阻塞。

## 实现多线程的方式

1. 继承Thread 类实现

|  |
| --- |
| public class ThreadDemo extends Thread{   public static void main(String[] args) {  ThreadDemo t1 = new ThreadDemo();  ThreadDemo t2 = new ThreadDemo();  ThreadDemo t3 = new ThreadDemo();  t1.start();  t2.start();  t3.start();  }  @Override  public void run() {  System.*out*.println("ThreadDemo run....");  } } |

1. 实现Runnable接口

|  |
| --- |
| public class RunnableDemo implements Runnable {  public static void main(String[] args) {  for(int i=0;i<3;i++){  Thread t = new Thread(new RunnableDemo(),"线程"+i);  t.start();  }  }  @Override  public void run() {  System.*out*.println("线程名称:"+Thread.*currentThread*().getName()+" RunnableDemo is run....");  } } |

1. 实现Callable接口通过futureTask包装器来创建线程

依赖子线程的返回结果

|  |
| --- |
| public class CallableDemo implements Callable<String> {  public static void main(String[] args) throws ExecutionException, InterruptedException {  ExecutorService executorService= Executors.*newFixedThreadPool*(1);  CallableDemo callableDemo = new CallableDemo();  Future<String> futures= executorService.submit(callableDemo);  System.*out*.println(futures.get());  executorService.shutdown();  }  @Override  public String call() throws Exception {  int a =1;  int b=2;  return "执行结果："+(a+b);  } } |

## 线程的状态

线程是java程序中执行的最小单元，有6种状态 NEW 、RUNNABLE、 BLOCKED 、WAITTING 、TIME\_WAITING 、TERMINATED

NEW：初始状态，线程被构建，但是还没有调用start方法

RUNNABLED：运行状态，JAVA线程把操作系统中的就绪和运行两种状态统一称为“运行中”

BLOCKED：阻塞状态，表示线程进入等待状态,也就是线程因为某种原因放弃了CPU使用权，阻塞也分为几种情况

* 等待阻塞：运行的线程执行wait方法，jvm会把当前线程放入到等待队列
* 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被其他线程锁占用了，那么jvm会把当前的线程放入到锁池中
* 其他阻塞：运行的线程执行Thread.sleep或者t.join方法，或者发出了I/O请求时，JVM会把当前线程设置为阻塞状态，当sleep结束、join线程终止、io处理完毕则线程恢复

TIME\_WAITING：超时等待状态，超时以后自动返回

TERMINATED：终止状态，表示当前线程执行完毕



## 线程的停止

虽然jdk 提供了stop方法，但是是一个过期的方法，调用后并不保证线程能停止，资源能够释放。如果要优雅的去中断一个线程，使用interrupt方法。

### interrupt方法

当其他线程通过调用当前线程的interrupt方法，表示向当前线程打个招呼，告诉他可以中断线程的执行了，至于什么时候中断，取决于当前线程自己。

线程通过检查资深是否被中断来进行相应，可以通过isInterrupted()来判断是否被中断。

|  |
| --- |
| public class InterruptDemo {  private static int *i*;  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (!Thread.*currentThread*().isInterrupted()){  *i*++;  }  System.*out*.println("i的值："+*i*);  }  },"interruptDemo");   t.start();  Thread.*sleep*(1000);  t.interrupt();  }  } |

### Thread.interrupted 线程复位

* 当前线程复位

|  |
| --- |
| public class InterruptedDemo {  private static int *i*;  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (true){  boolean ii=Thread.*currentThread*().isInterrupted();  if(ii){  System.*out*.println("before:"+ii);  //对线程进行复位  Thread.*interrupted*();  System.*out*.println("after:"+Thread.*currentThread*().isInterrupted());  }  }  //System.out.println("i的值："+i);  }  },"InterruptedDemo");   t.start();  Thread.*sleep*(1000);  t.interrupt();  }  } |

* 其他线程复位

除了通过Thread.interrupted方法对线程中断标识进行复位以外，还有一种被动复位的场景，就是对抛出InterruptedException异常的方法，在InterruptedException抛出之前，JVM会先把线程的中断标识位清除，然后才会抛出InterruptedException，这个时候如果调用isInterrupted方法，将会返回false

|  |
| --- |
| public class InterruptExceptionDemo {  public static void main(String[] args) throws InterruptedException {   Thread t = new Thread(new Runnable() {  @Override  public void run() {  while (true){   try {  Thread.*sleep*(100000);  } catch (InterruptedException e) {  //抛出该异常，会将复位标识设置为false  e.printStackTrace();  }  }  }  },"InterruptedDemo");  t.start();  TimeUnit.*SECONDS*.sleep(1);  t.interrupt();//设置复位标识为true  TimeUnit.*SECONDS*.sleep(1);  System.*out*.println(t.isInterrupted());//false  }  } |

## 其他的线程停止方法

|  |
| --- |
| public class VolatileDemo {  private static volatile boolean *stop*=false;   public static void main(String[] args) throws InterruptedException {  Thread thread = new Thread(new Runnable() {  int i=0;  @Override  public void run() {  while(!*stop*){  i++;  }  System.*out*.println("i="+i);  }  });  thread.start();  System.*out*.println("开始执行 Thread");  Thread.*sleep*(1000);  *stop*=true;   } } |