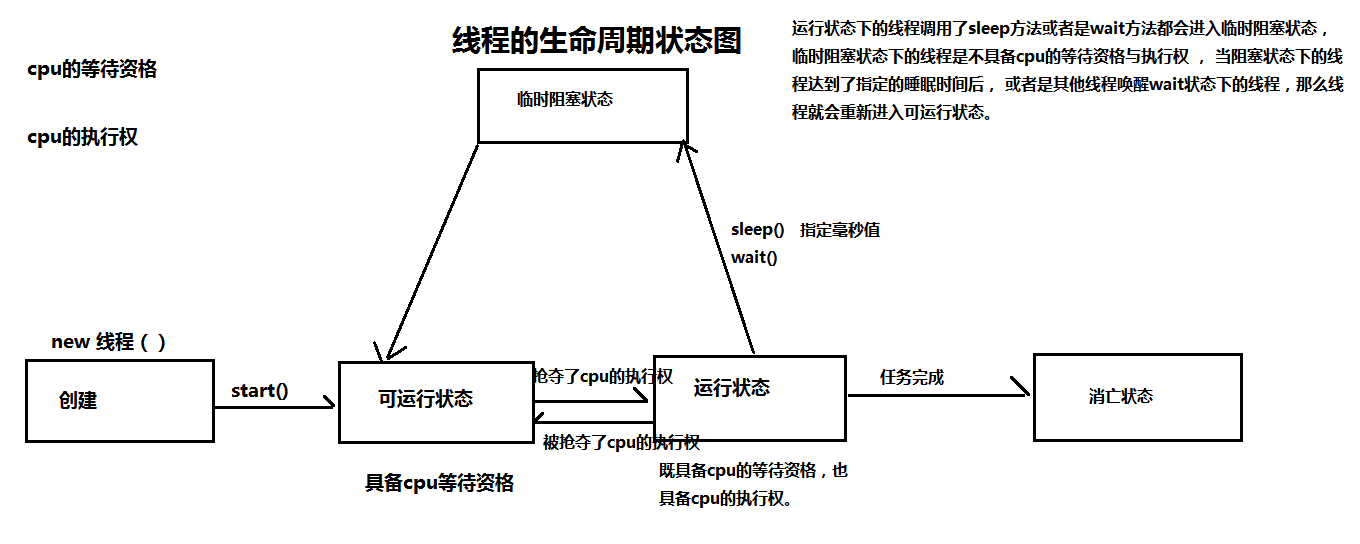
# 多线程

线程的概念:



创建多线程的两种方式:

1. 继承Thread类
2. 实现Runnable接口
3. 实现Callable接口

## 1.1继承Thread类(java.lang)

public class Thread

extends Object

implements Runnable

每个继承了Thread的子类必须覆写run()方法,这个run方法

没有返回值,这表明线程一旦开始就要一直执行,不能返回内容

启动线程要调用start方法:public void start()此方法会去执行run方法,

每个线程的start方法只能调用一次,否则抛出IllegalThreadStateException

public void run()

范例:

public class test{  
 public static void main(String args[]){  
 MyThread mt1 = new MyThread("线程A");  
 MyThread mt2 = new MyThread("线程B");  
 MyThread mt3 = new MyThread("线程C");  
 mt1.start();  
 mt2.start();  
 mt3.start();  
 }  
}  
class MyThread extends Thread{  
 private String name;  
 public MyThread(String name){  
 this.name = name;  
 }  
 @Override  
 public void run() {  
 for(int x=0;x<200;x++){  
 System.*out*.println(this.name + "-->" + x);  
 }  
 }  
}

## 1.2实现Runnable接口

虽然Thread类可以实现多线程的主题类定义,但是java具有单继承类局性问题,

所以应该使用实现Runnable接口

Runnable接口定义:

|  |
| --- |
| @FunctionalInterface  public interface Runnable{  public void run();  } |

实现Runnable接口,重写里面的run方法就可以了

Thread的构造方法:

|  |
| --- |
| **public Thread(Runnable target)** |

范例:

|  |
| --- |
| public class test{  public static void main(String args[]){  MyThread m1 = new MyThread("线程-A");  MyThread m2 = new MyThread("线程-B");  MyThread m3 = new MyThread("线程-C");  new Thread(m1).start();  new Thread(m2).start();  new Thread(m3).start();  }  }  class MyThread implements Runnable{  private String name;  public MyThread(String name){  this.name = name;  }  @Override  public void run() {  for(int x = 0;x<50;x++){  System.out.println(this.name + "-->" + x);  }  }  } |

## 1.3图解Thread和Runnable

主要实现多线程的核心功能

1.操作系统的资源分配

2调用run()

Runnable

Thread

MyThread

子类

真实

代理

子类

start()

客户端

整个结构看起来像是代理设计模式,如果是代理设计模式,客户端调用的是应该是接口里提供的方法,应该是run()才对

使用Runnable接口可以比Thread类能更好的描述数据共享这一概念,数据共享指的是多个线程访问同一个资源的操作

为了证明数据共享:

**范例:Thread**

|  |
| --- |
| public class test{  public static void main(String args[]){  //由于MyThread类有start(),所以每一个Thread就是一个线程对象  MyThread mt1 = new MyThread();  MyThread mt2 = new MyThread();  MyThread mt3 = new MyThread();  mt1.start();  mt2.start();  mt3.start();  }  }  class MyThread extends Thread{  private int ticket = 10;  @Override  public void run() {  for(int x = 0;x<100;x++){  if(this.ticket > 0){  System.out.println("买票,ticket=" + this.ticket--);  }  }  }  } |

**发现每个线程都在卖各自的票，没有数据共享这一概念**

**范例:Runnable**

|  |
| --- |
| public class test{  public static void main(String args[]){  MyThread mt = new MyThread();  //注意mt里的ticket  new Thread(mt).start();  new Thread(mt).start();  new Thread(mt).start();  }  }  class MyThread implements Runnable{  private int ticket = 10;  @Override  public void run() {  for(int x = 0;x<100;x++){  if(this.ticket > 0){  System.out.println("买票,ticket=" + this.ticket--);  }  }  }  } |

**三个线程都占用同一个MyThread对象引用,这个三个线程对象都访问同一个数据资源**

## 1.4实现Callable接口(java.util.concurrent)

### 1.4.1 Callable定义一个线程主体

能够返回线程的结果,很麻烦

@FunctionalInterface

public interface Callable<V>{

public V call() throws Exception

}

call()方法执行完线程主题功能后可以返回一个结果，而返回结果

的类型有Callable接口的泛型来决定

范例:定义一个线程主体类

|  |
| --- |
| class MyThread implements Callable<String>{ //线程返回一个String类型  private int ticket = 10;  @Override  public String call() {  for(int x = 0;x<100;x++){  if(this.ticket > 0){  System.out.println("买票,ticket=" + this.ticket);  }  }  return "票已经卖完了";  }  } |

### FutureTask<V>类

Java.util.concurrent包里

public class FutureTask<V>

extends Object

implements RunnableFuture<V>

作用:负责Callable接口对象操作的

有一个构造函数:

|  |
| --- |
| **public FutureTask(Callable<V> callable)** |

接受Callable的目的:取得call()方法的返回结果

### 1.4.3RunnableFuture<V>接口

Java.util.concurrent包里:

public interface RunnableFuture<V>

extends Runnable, Future<V>

### 1.4.4Future<V>接口

java.util.concurrent:

public interface Future<V>

两个方法:用来返回FutureTask存储的线程返回结果

|  |  |
| --- | --- |
|  | V get()  throws InterruptedException,  ExecutionException |
|  | V get(long timeout,  TimeUnit unit)  throws InterruptedException,  ExecutionException,  TimeoutException |

### 1.4.5. 完整demo

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  MyThread mt1 = new MyThread();  FutureTask<String> task1 = new FutureTask(mt); //获取call的返回结果  FutureTask<String> task2 = new FutureTask(mt1); //获取call的返回结果  //FutureTask是Runnable接口子类  new Thread(task1).start(); //启动多线程  new Thread(task2).start();  //多线程执行完毕后可以取得内容,依赖FutureTask的父  //接口Future中的get()方法  System.out.print("A线程中的返回结果:" + task1.get());  System.out.print("B线程中的返回结果:" + task2.get());  }  }  class MyThread implements Callable<String>{ //线程返回一个String类型  private int ticket = 10;  @Override  public String call() {  for(int x = 0;x<100;x++){  if(this.ticket > 0){  System.out.println("买票,ticket=" + this.ticket--);  }  }  return "票已经卖完了";  }  } |

## 1.5线程的命名

Thread的一些方法

|  |  |
| --- | --- |
| 构造函数 | public Thread(Runnable target,String name) |
| 设置线程的名字 | public final void setName(String name) |
| 取得线程的名字 | public final String getName() |
| 当前程序所在的进程 | public static Thread currentThread() |

如果没有给线程命名,则线程的名字会被自动命名

范例:

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  System.out.println(Thread.currentThread().getName());  new Thread(mt,"自己的线程A").start();  new Thread(mt).start();  new Thread(mt,"自己的线程B").start();  new Thread(mt).start();  new Thread(mt).start();  }  }  class MyThread implements Runnable{ //线程返回一个String类型  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  } |

## 1.6线程的休眠

指的是让线程的执行速度稍微变慢一点,休眠的方法:

Thread类里:

|  |
| --- |
| public static void sleep(long millis)  throws InterruptedException |

范例:

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  new Thread(mt,"自己的线程A").start();  new Thread(mt,"自己的线程B").start();  new Thread(mt,"自己的线程C").start();  new Thread(mt,"自己的线程D").start();  new Thread(mt,"自己的线程E").start();  }  }  class MyThread implements Runnable{ //线程返回一个String类型  @Override  public void run() {  for(int x = 0;x<1000;x++) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ",x=" + x);  }  }  } |

## 1.7线程优先级

优先级:指的是越高的优先级,越有可能先执行

Thread类方法:

|  |  |
| --- | --- |
| 设置优先级 | public final void setPriority(int newPriority) |
| 取得优先级 | public final int getPriority() |

Thread类常量:

|  |  |
| --- | --- |
| 最高优先级 10 | public static final int MAX\_PRIORITY |
| 中等优先级 5 主线程为5 | public static final int NORM\_PRIORITY |
| 最低优先级 1 | public static final int MIN\_PRIORITY |

范例:

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  Thread t1 = new Thread(mt,"自己的线程A");  Thread t2 = new Thread(mt,"自己的线程B");  Thread t3 = new Thread(mt,"自己的线程C");  t3.setPriority(Thread.MAX\_PRIORITY);  t2.setPriority(Thread.NORM\_PRIORITY);  t1.setPriority(Thread.MIN\_PRIORITY);  t1.start();  t2.start();  t3.start();  System.out.print(Thread.currentThread().getPriority());//主线程优先级 5  }  }  class MyThread implements Runnable{ //线程返回一个String类型  @Override  public void run() {  for(int x = 0;x<20;x++) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ",x=" + x);  }  }} |

## 1.8线程同步问题

### 1.8.1同步问题引出

范例:

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  new Thread(mt,"A").start();  new Thread(mt,"B").start();  new Thread(mt,"C").start();  }  }  class MyThread implements Runnable{ //线程返回一个String类型  private int ticket = 5;  @Override  public void run() {  for(int x = 0;x<20;x++) {  if(this.ticket>0) {  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()+",ticket"+this.ticket--);  }  }  }  } |

B,ticket4

A,ticket3

C,ticket5

B,ticket2

A,ticket1

C,ticket0

B,ticket-1

分析:如果ticket=1时,A线程进入到for循环里判断ticket>0,此时睡眠100ms,这时B线程因为A线程再睡眠并没有减1,此时B里的ticket还是1,依然>0,C线程也一样,最后出去的时候会都会减1,然后就有了负的数据

### 1.8.1synchronized关键字

可以通过两种方法使用:

1. 同步代码块 //有四种代码块:普通代码块,构造块,静态块,同步块
2. 同步方法

范例:同步块(不常用)

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  new Thread(mt,"A").start();  new Thread(mt,"B").start();  new Thread(mt,"C").start();  }  }  class MyThread implements Runnable{ //线程返回一个String类型  private int ticket = 5;  @Override  public void run() {  for(int x = 0;x<20;x++) {  synchronized (this) { //当前操作每次只允许一个对象进入  if (this.ticket > 0) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ",ticket" + this.ticket--);  }  }  }  }  } |

范例:同步方法解决(常用)

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  MyThread mt = new MyThread();  new Thread(mt,"A").start();  new Thread(mt,"B").start();  new Thread(mt,"C").start();  }  }  class MyThread implements Runnable{ //线程返回一个String类型  private int ticket = 5;  @Override  public void run() {  for(int x=0;x<200;x++){  this.sale(); //调用同步方法  }  }  public synchronized void sale() { //每次只能有一个线程访问sale方法  if (this.ticket > 0) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ",ticket" + this.ticket--);  }  }  } |

## 1.9生产者消费者模型

### 1.9.1问题引出

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  Info info = new Info();  new Thread(new Productor(info)).start();  new Thread(new Customer(info)).start();  }  }  class Info{  private String title;  private String content;  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public String getContent() {  return content;  }  public void setContent(String content) {  this.content = content;  }  }  class Productor implements Runnable{  private Info info;  public Productor(Info info){  this.info = info;  }  @Override  public void run() {  for(int x = 0;x<100;x++){  if(x%2 == 0){  this.info.setTitle("A");  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.info.setContent("好学生");  }else{  this.info.setTitle("B");  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.info.setContent("坏学生");  }  }  }  }  class Customer implements Runnable{  private Info info;  public Customer(Info info){  this.info = info;  }  @Override  public void run() {  for(int x=0;x<100;x++){  try {  Thread.sleep(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(this.info.getTitle()+ "-" + this.info.getContent());  }  }  } |

两个问题:

1. 数据错位
2. 数据重复取出,数据重复设置

### 1.9.2解决数据错位的问题

数据的错位完全是因为非同步的操作造成了,即设置数据的时候有两个数据要设置title,content,应该使用同步处理，将两个set函数合并为一个,将两个get函数合并为一个

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  Info info = new Info();  new Thread(new Productor(info)).start();  new Thread(new Customer(info)).start();  }  }  class Info{  private String title;  private String content;  public synchronized void set(String title,String content){  this.title = title;  try {  Thread.sleep(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.content = content;  }  public synchronized void get(){  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(this.title+ "-" + this.content);  }  }  class Productor implements Runnable{  private Info info;  public Productor(Info info){  this.info = info;  }  @Override  public void run() {  for(int x = 0;x<100;x++){  if(x%2 == 0){  this.info.set("A","好学生");  }else{  this.info.set("B","坏学生");  }  }  }  }  class Customer implements Runnable{  private Info info;  public Customer(Info info){  this.info = info;  }  @Override  public void run() {  for(int x=0;x<100;x++){  this.info.get();  }  }  } |

同步代码解决了错位的问题

### 1.9.3解决重复的问题wait notify notifyAll

要解决重复问题,必须加入等待和唤醒机制,在Object里

|  |  |
| --- | --- |
| 等待 | public final void wait()  throws InterruptedException |
| 唤醒第一个等待线程 | public final void notify() |
| 唤醒所有等待线程 | public final void notifyAll() |

注意:wait和notify所在的函数一定要是synchronized或者同步代码块,不然报错

范例:解决重复

|  |
| --- |
| public class test{  public static void main(String args[]) throws ExecutionException, InterruptedException {  Info info = new Info();  new Thread(new Productor(info)).start();  new Thread(new Customer(info)).start();  }  }  class Info{  private String title;  private String content;  private boolean flag = true;  //flag = true,表示可以生产,但是不可以取走  //flag = false,表示可以取走,但是不可以生产  public synchronized void set(String title,String content){  if(this.flag == false){  try {  super.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  this.title = title;  try {  Thread.sleep(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.content = content;  this.flag = false;  super.notify();//唤醒其他等待线程  }  public synchronized void get(){  if(this.flag == true){  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(this.title+ "-" + this.content);  this.flag = true;  notify();  }  }  class Productor implements Runnable{  private Info info;  public Productor(Info info){  this.info = info;  }  @Override  public void run() {  for(int x = 0;x<100;x++){  if(x%2 == 0){  this.info.set("A","好学生");  }else{  this.info.set("B","坏学生");  }  }  }  }  class Customer implements Runnable{  private Info info;  public Customer(Info info){  this.info = info;  }  @Override  public void run() {  for(int x=0;x<100;x++){  this.info.get();  }  }  } |

## 2.0死锁

|  |
| --- |
| public class test {  public static void main(String args[]) throws Exception {  DeadLockThread mt1 = new DeadLockThread("张三");  DeadLockThread mt2 = new DeadLockThread("李四");  Thread t1 = new Thread(mt1,mt1.getName());  Thread t2 = new Thread(mt2,mt2.getName());  t1.start();  t2.start();  }  }  class DeadLockThread implements Runnable{  private String name;  public String getName() {  return name;  }  public DeadLockThread(String name) {  this.name = name;  }  @Override  public void run() {  if("张三".equals(this.getName())){  synchronized ("遥控器"){  System.out.println(this.getName() + "取走了遥控器,准备取电池");  synchronized ("电池"){  System.out.println(this.getName()+ "取到了电池,开着空调爽歪歪");  }  }  }else if("李四".equals(this.getName())){  synchronized ("电池"){  System.out.println(this.getName() + "取走了电池,准备取遥控器");  synchronized ("遥控器"){  System.out.print(this.getName()+"取走了遥控器,开着空调爽歪歪");;  }  }  }  }  } |

## 2.1守护线程

判断是否为守护线程的方法:

Thread类中里public final boolean isDaemon()

设置线程为守护线程:

Thread类中里: public final void setDaemon(boolean on)

守护线程(后台线程):**当前一个java应用只剩下守护线程的时候,**

**守护线程会马上结束**

**注意点:**

1)所有的线程默认都不是守护线程

范例:守护进程

|  |
| --- |
| public class test {  public static void main(String args[]) throws Exception {  Demo d = new Demo("守护线程");  Thread t1 = new Thread(d,d.getName());  t1.setDaemon(true); //设置为守护线程  System.out.println("是守护线程吗?" + t1.isDaemon());  t1.start();  Thread.sleep(100);  for(int i=0;i<100;i++){  System.out.println(Thread.currentThread().getName()+ ":" + i + "%");  }  }  }  class Demo implements Runnable {  private String name;  public Demo(String name) {  this.name = name;  }  public String getName() {  return name;  }  @Override  public void run() {  for(int i=0;i<100;i++){  System.out.println(this.getName() + "已经下载了:" + i + "%");  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.out.print("下载完毕,正在安装更新包");  }  } |

## 2.2join等待某个线程

如果在A线程中创建了B线程,则默认状态下,A,B会同时执行,

如果调用B.join()方法后则知道B线程结束后才会继续执行A线程

Thread类中:

public final void join() throws InterruptedException

还有两个重载的join()函数

|  |
| --- |
| public class test {  public static void main(String args[]) throws Exception {  Monther m = new Monther();  Thread t = new Thread(m);  t.start();  Thread t1 = new Thread(new Son());  t1.start();  t1.join();  }  }  class Monther implements Runnable{  @Override  public void run() {  System.out.println("妈妈洗菜...");  System.out.println("妈妈切菜...");  System.out.println("妈妈发现没有酱油了...");  //通知儿子去打酱油  // Son s = new Son();  // Thread t1 = new Thread(s);  // t1.start();  // try {  // t1.join(); //当前线程等待t1线程执行完后,再执行当前线程  // } catch (InterruptedException e) {  // e.printStackTrace();  // }  System.out.println("妈妈炒菜...");  System.out.println("全家一起吃饭...");  }  }  class Son implements Runnable{  @Override  public void run() {  try {  System.out.println("儿子下楼梯");  Thread.sleep(1000);  System.out.println("一直往前走...");  System.out.println("买到了酱油");  System.out.println("跑回来...");  Thread.sleep(1000);  System.out.println("儿子把酱油给老妈");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

## 2.3线程间通讯

当一个线程完成了一个任务后,要通知另外的一个线程去处理其他事情,主要是通过notify和wait方法来完成的,请看消费者和生产者模型

## 2.4线程的停止

Thread类里:

public void interrupt()//当线程调用wait或者sleep后,会进入阻塞状态,如果再调用Interrupt()则会清楚阻塞状态,并且给线程发送一个InterruptedException异常

注意:和notify的区别是interrupt能指定清楚哪个线程的阻塞状态,notify是不能的

范例:

|  |
| --- |
| public class test {  public static void main(String args[]) throws Exception {  Demo d = new Demo();  Thread t = new Thread(d,"线程demo");  t.start();  //当主线程i为80时停止,demo线程  for(int i=0;i<100;i++){  if(i==80){  d.flag = false;  t.interrupt();//无法停止线程  }  System.out.println(Thread.currentThread().getName()+":" + i);  }  }  }  class Demo implements Runnable{  public boolean flag = true;  @Override  public synchronized void run() {  int i = 0;  while(flag){  try {  this.wait();//这个wait要在同步块或者同步函数中出现不然报错  } catch (InterruptedException e) {  System.out.println("接受到了一个interrupted");  }  System.out.println(Thread.currentThread().getName()+":" + i);  ++i;  }  }  } |