# Day22

今日主要内容：

1. 并行与并发
2. 两种线程的实现方式
3. 线程中常的方法
4. 线程安全问题

## 程序、进程、线程三个概念

程序：是以文件形式存储在磁盘上。是一个静态方式

进程：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。

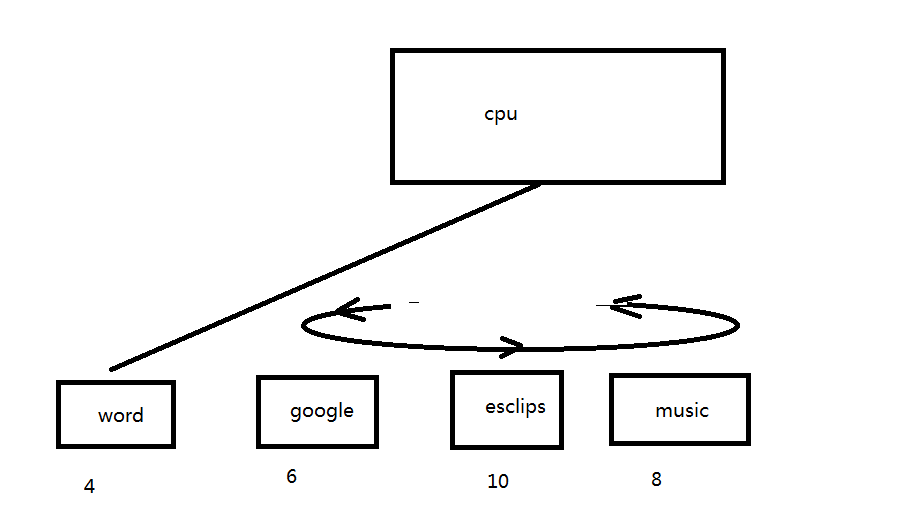
线程：是进程的一个单一控制流程，是进程执行的最小单位。一般情况下，一个进程可以分配好多任务，每个任务就是一个线程。

## 并行与并发

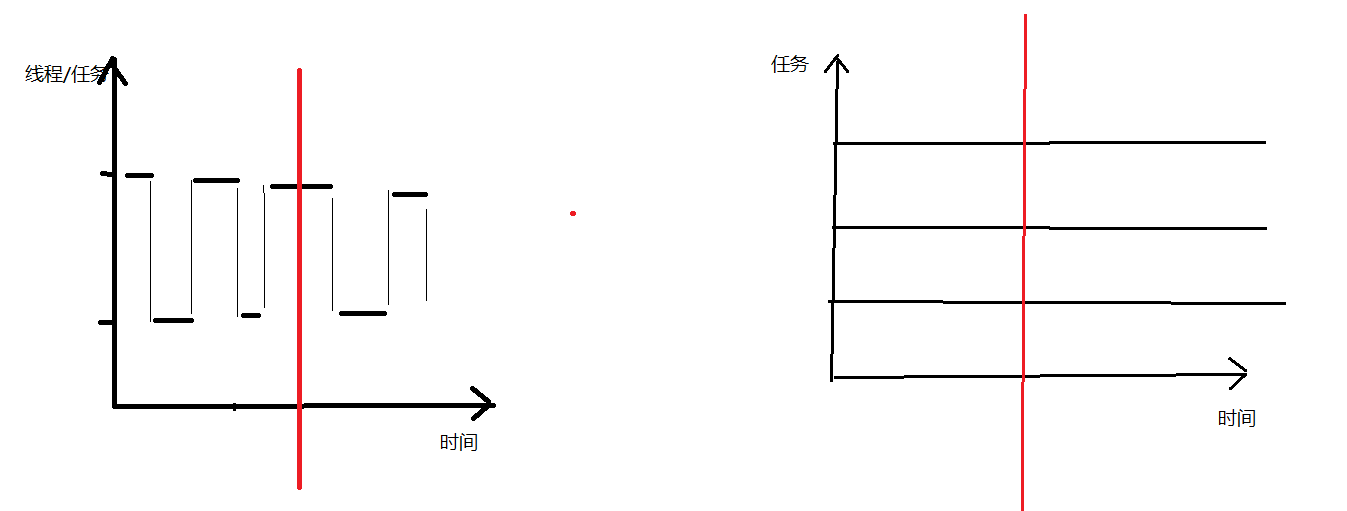
并行： 多个任务要同时运行（有多个CPU）,每个任务都运行在不同的Cpu上。并行编程。

并发： 多个任务同时发起请求，（cpu） 在某一个时刻，只一个任务（线程）在cpu中执行。

Cpu并发执行线程图



并发与并行图：



在某一个时刻只有一个交点，就是并发，如果有多个交点就是并行。

并发提升了Cpu的利用率。对于线程来讲，效率降低了。

所谓并发其实解决不同硬件之间速率不同的问题。

Cpu去执行了一个线程，该线程去硬盘上读取一个数据。此时cpu不会等待该线程读取数据，而是切换到其它线程上去执行。Cpu是不能空闲的。不能出出现高昂的设备去等待低廉的设备。

## 两种线程的实现方式

### 第一种继承的方式

1. 写一个类去继承Thread
2. 去实现run方法，是线程启动后要运行任务。要运行的代码
3. 创建一个线程对象
4. 调用它的start方法进行启动

第二种是实现Runnable接口

1. 写一个类去实现Runnable接口
2. 实现Runnable接口中的run方法
3. 创建一个线程对象，把我们的要执行的任务，绑定到线程中
4. 启动线程。

|  |
| --- |
| public class DemoThreadOne {    public static void main(String[] args) {    //新建了一个线程对象而已  //Thread01 t1=new Thread01();  //t1.start(); //这时该线程才有可能被cpu执行    //创建一个任务对象  MyTask mt=new MyTask();  //把任务对象绑定到一个线程上  Thread t2=new Thread(mt);  //启动线程  t2.start();      }  }  class Thread01 extends Thread{    public void run() {  //写该线程要执行的代码  for(int i=1;i<=200;i++)  System.out.println("运行了"+i);  }    }  class MyTask implements Runnable{  @Override  public void run() {    //任务所要执行的内容  System.out.println("执行了Runnable中的run方法");  }    } |

线程的二种实现方式有什么区别？

1. 继承方式：

如果一个类去继承Thread，你这个类就不能去再继承其它类。我们的任务和线程对象高度耦合，扩展性差。

1. 实现Runnable接口的方式

如果一个类去实现Runnable接口，该类还可以去继承其它类，我们的任务与线程对象进行了耦合度比较低，扩展性较强。这种方式可以被多个线程去执行同一个任务。还可以为多个线程共享资源，方便的实现线程之间的数据交换。

### 还可以通过匿名内部类的方式创建线程

|  |
| --- |
| public class DeThreadTwo {    public static void main(String[] args) {    new Thread() {    public void run() {  System.out.println("aaaa");  }  }.start();    new Thread(new Runnable() {    public void run() {  System.out.println("bbbb");  }    }).start();      }  } |

### 如何获取线程的名子

1. getName()方法，可以获取线程的名子
2. 默认情况下线程的名子叫 thread-xx xx表示整型数 如0,1,2
3. 实现了Runnable接口的作任中不能直接使用getName

可以使用Thread类中的静态方法currentThread()，获取当前线程对象。

然后就可以使用该线程对象去调用getName方法来获取线程名子

|  |
| --- |
| public class DemoThreadThree {    public static void main(String[] args) {  //创建线程对象  MyThread01 mt01=new MyThread01();  //启动线程，只有启动了线程，该线程才能在cpu的队列中排队  mt01.start();  String threadName=mt01.getName();    //由main所有的线程打印的mt01线程的名子  System.out.println(threadName);    new Thread() {    public void run() {  System.out.println("bbb");  //线程对象本身调用了getName方法，获取了自己线程名子  System.out.println(getName());  }  }.start();      }  }  class MyThread01 extends Thread{    public void run() {  System.out.println("aaa");  }  } |

### 设置线程的名子

在thread中有一个setName方法可以设置线程的名子,

可以在起动线程之前或之后都可以设置线程的名子

|  |
| --- |
| public class DemoThreadFour {    public static void main(String[] args) {    Thread td=new Thread() {    public void run() {  System.out.println("-----aaa");  System.out.println(getName());  }  };    td.setName("线程10");  //启动线程  td.start();  //main线程设置td线程的名子  //td.setName("线程100");    }  } |

练习:

获取执行main方法线程的名子

获取垃圾回收对象的线程的名子

|  |
| --- |
| public class DemoThreadFive {    public static void main(String[] args) {  //获取执行main方法线程的名子  String mainName=Thread.currentThread().getName();  System.out.println(mainName);    for(int i=1;i<999999;i++) {  new Demo();  }    }  }  class Demo{    @Override  protected void finalize() throws Throwable {  String threadName=Thread.currentThread().getName();  System.out.println(threadName);  }    } |

### 线程的阻塞

1. Thread类中提供了一个sleep(整型值)方法，让线程休息或阻塞一定的时间。整型值的设置，单位ms sleep(10) sleep(3000),通过设置休息时间，可以在调试代码时，显示指定的效果。只要线程阻塞cpu就会立刻切换到其它线程。

异常处理：InterruptedException 编译时异常

如果该异常在普通方法中，可以捕获try{} catch{} 或对外进行声明

如果在线程的run方法中只能捕获

|  |
| --- |
| public class DemoThreadSix {    public static void main(String[] args) throws InterruptedException {    Thread td=new Thread() {    public void run() {  for(int i=1;i<1000;i++) {  System.out.println("aaa===="+i);    }    }  };  td.start();    for(int i=1;i<10;i++) {  System.out.println("bbbbb======"+i);  Thread.sleep(50);  }      }  }  class MyThread05 extends Thread{    public void run() {    System.out.println("ccc");  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

### 守护线程

守护线程（用户线程、后台线程）就守护正线程的运行，为正常线程（核心线程）提交良好运行服务。如果正常线程都运行完毕了，守护线程也就没有存在的必要了，过一段时间之后，程序将退出。

如果想把一个线程设置成一个守护线程：**[setDaemon](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "setDaemon(boolean))**(boolean on)如果为 true，则将该线程标记为守护线程。

判断一个线程是否为守护线程：**[isDaemon](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "isDaemon())**() 返回true表示守护线程

|  |
| --- |
| public class DemoThreadSeven {    public static void main(String[] args) {    Thread td=new Thread() {  public void run() {  while(true) {  System.out.println("我一直为大家提供服务...");  }  }  };  //设置td为守护线程  td.setDaemon(true);  //启动线程  td.start();    for(int i=1;i<1000;i++) {  System.out.println(Thread.currentThread().getName()+"======"+i);  }    }  } |

练习: 测试执行垃圾回收的线程是否为守护线程。

|  |
| --- |
| public class DemoThreadEnghit {  public static void main(String[] args) {  new Gar();  System.gc();//手动调用垃圾回收器}  }  class Gar{    @Override  protected void finalize() throws Throwable {  //获取当前执行的线程对象  Thread currentThread=Thread.currentThread();  System.out.println(currentThread.isDaemon());    }  } |

### 线程的优先级

通过**[setPriority](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "setPriority(int))**(int newPriority) 可以设置线程的优先级，一共有10级别

最小的级别为1 默认级别为5 最大级别为10

**[MAX\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "MAX_PRIORITY)** 线程可以具有的最高优先级。

**[MIN\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "MIN_PRIORITY)**线程可以具有的最低优先级

**[NORM\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "NORM_PRIORITY)**分配给线程的默认优先级

高优先级的线程运行的机会可能会多一点。（在前半段时间内，先运行高优先级的线），低优先级的线程会少一点（在后半段时间它会运行的多一点）。

|  |
| --- |
| public class DemoThreadNine {    public static void main(String[] args) {    Thread t1=new Thread() {    public void run() {    for(int i=1;i<=200;i++) {  System.out.println(getName()+"......."+i);  }  }  };  //设置线程的优先级  t1.setPriority(Thread.MAX\_PRIORITY);  //设置线程名子  t1.setName("高优先级的线程");    Thread t2=new Thread() {    public void run() {    for(int i=1;i<=200;i++) {  System.out.println(getName()+"======="+i);  }  }  };  //设置线程的优先级  t2.setPriority(Thread.MIN\_PRIORITY);  //设置线程名子  t2.setName("低优先级的线程");    t2.start();    t1.start();    }  } |

## 线程的安全问题

某段代码在没有执行完的情况下，cpu就可以被其它线程抢走了，导入我们的代码没执行完，会数据错误发生。

前提条件：是多个线程共享同一个资源，这个资源具有完整性和原子性。

|  |
| --- |
| public class DemoThreadTen {    public static void main(String[] args) {    MyTask100 mt100=new MyTask100();  Thread t1=new Thread(mt100,"线程A");  Thread t2=new Thread(mt100,"线程B");  t1.start();  t2.start();    }  }  class MyTask100 implements Runnable{  int money=0;  public void run() {  for(int i=1;i<=10000;i++) {  //线程A  money=money+1;  }  System.out.println(Thread.currentThread().getName()+"==="+money);  }    } |

使用同步代码块来解决此问题 ，同步代码块的语法:

synchronized (锁对象) {

//受保护的代码

//具有完整性的代码

}

当Cpu想去执行同步代码块中的代码时，首先要获取锁对象，获取锁对象之后就可以执行同步代码块中的代码了，此时cpu不会切换到跟我使用相锁的对同步块中去执行代码。

解决线程安全使用同步代码块案例：

|  |
| --- |
| public class DemoThreadPrint {    public static void main(String[] args) {    Printor p=new Printor();  Thread t1=new Thread() {    public void run() {  while(true) {  p.print1();  }  }  };    Thread t2=new Thread() {    public void run() {  while(true) {  p.print2();  }  }  };    t1.start();  t2.start();    }  }  class Printor{    //Object obj=new Object();  public void print1() {    //这个锁对象必须是同一个对象  synchronized (this) {  System.out.print("中");  System.out.print("公");  System.out.print("教");  System.out.println("育");  }  }    public void print2() {    synchronized (this) {  System.out.print("优");  System.out.print("就");  System.out.println("业");  }  }  } |

同步方法:我们可以设置一个方法为同步方法。就是给这个方法加上一个锁对象

其实，同步方法是同步块的一种简化。

同步方法的格式:

访问控制符 修改符 同步关键字 返回值 方法名(参数){

}

对于同步方法，如果该方法是一个非静态的，使用的锁就是this,

如果方法是一个静态的方法，如果要加锁应使用该的Class类的对象 XXX.class

**对于静态方法的同步锁案例**

|  |
| --- |
| public class DemoThreadPrint {    public static void main(String[] args) {    Printor p=new Printor();  Thread t1=new Thread() {    public void run() {  while(true) {  p.print1();  }  }  };    Thread t2=new Thread() {    public void run() {  while(true) {  p.print2();  }  }  };    t1.start();  t2.start();    }  }  class Printor{    //Object obj=new Object();  //Printor.class  public static synchronized void print1() {  /\*  //这个锁对象必须是同一个对象  synchronized (this) {  System.out.print("中");  System.out.print("公");  System.out.print("教");  System.out.println("育");  }\*/    System.out.print("中");  System.out.print("公");  System.out.print("教");  System.out.println("育");  }    public void print2() {    synchronized (Printor.class) {  System.out.print("优");  System.out.print("就");  System.out.println("业");  }  }  } |

### 死锁

A线程需要甲资源，拥有乙资源，B线程需要乙资源，拥有甲资源，两条线程都不肯释放自己的资源，同时也需要其它资源，程序就无法运行了。

|  |
| --- |
| public class DemoThreadLock {    public static void main(String[] args) {    Thread t1=new Thread("A哲学家") {    public void run() {  while(true) {    synchronized ("A筷子") {  System.out.println(getName()+" 拥用A筷子，等待B筷子");  synchronized ("B筷子") {  System.out.println(getName()+" 可以疯狂的吃饭");  }    }  }    }  };    Thread t2=new Thread("B哲学家") {    public void run() {  while(true) {    synchronized ("B筷子") {  System.out.println(getName()+" 拥用B筷子，等待A筷子");  synchronized ("A筷子") {  System.out.println(getName()+" 可以疯狂的吃饭");  }    }  }    }  };    t1.start();  t2.start();  }  } |

解决死锁的原因是因为使用同步块嵌套，在我们编程时尽量不要使用同步块嵌套，来避免死锁的发生。

经典案例：销售火车票示例

假设有三个窗口来销售，同一个列车中的火车票。

这个三个窗口用三个线程来模拟。

票数字来表示，假设这班列车有100票

|  |
| --- |
| public class TicketDemo {    public static void main(String[] args) {      Ticket t1=new Ticket("A窗口");  Ticket t2=new Ticket("B窗口");  Ticket t3=new Ticket("C窗口");  t1.start();  t2.start();  t3.start();    }      }  class Ticket extends Thread{    static int index=100;    public Ticket(String name) {  super(name);    }  public void run() {    while(true) {  synchronized (Ticket.class) {    if(index<1) {  break;  }    try {  Thread.sleep(30);  } catch (InterruptedException e) {  e.printStackTrace();  }    index--;  System.out.println(getName()+"卖了1张票,还剩下"+index);  }    }    }    } |