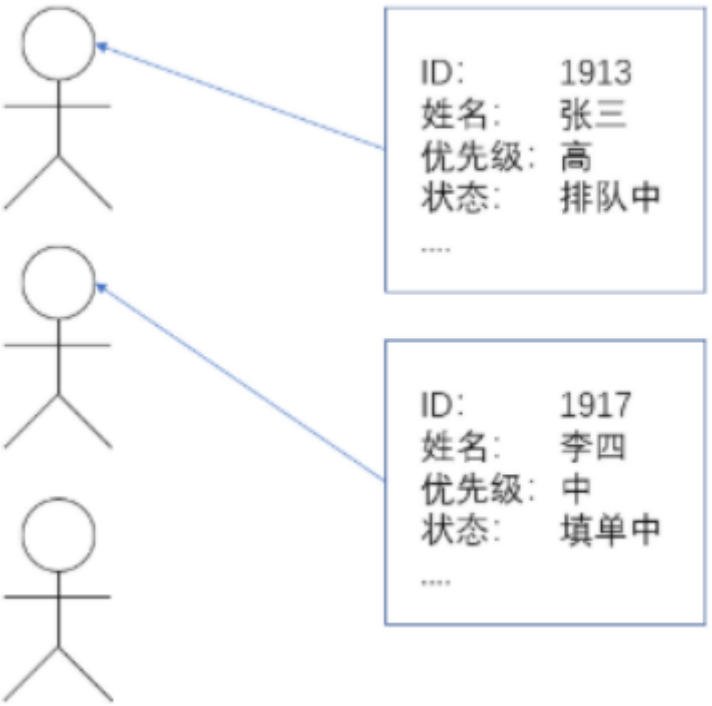
Thread 类是 JVM 用来管理线程的一个类，换句话说，每个线程都有一个唯一的 Thread 对象与之关 联。

用我们上面的例子来看，每个执行流，也需要有一个对象来描述，类似下图所示，而 Thread 类的对象 就是用来描述一个线程执行流的，JVM 会将这些 Thread 对象组织起来，用于线程调度，线程管理。



1、Thread 的常见构造方法

方法	说明
Thread()	创建线程对象
Thread(Runnable target)	使用 Runnable 对象创建线程对象
Thread(String name)	创建线程对象，并命名
Thread(Runnable target, String name)	使用 Runnable 对象创建线程对象，并命名
【了解】Thread(ThreadGroup group, Runnable target)	线程可以被用来分组管理，分好的组即为线程组，这个目前我们了解即可

```
Thread t1 = new Thread();
Thread t2 = new Thread(new MyRunnable());
Thread t3 = new Thread("这是我的名字");
Thread t4 = new Thread(new MyRunnable(), "这是我的名字");
```

1.1 方法一：

通过自己定义一个类继承 Thread 类,重写 Thread 类的 run 方法

```
class MyThread extends Thread{
 @Override
 public void run() {
 System.out.println("hello Thread!");
 }
}
```

```
public class ThreadDemo {
 public static void main(String[] args) {
 Thread thread = new MyThread();
  thread.start();
  }
}
```

1.2 方法二：

实现 Runnable 接口，重写 run

```
class MyRunnable implements Runnable{
 @Override
 public void run() {
 while(true){
 System.out.println("Hello Thread!");
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
  }
  }
  }
}

public class ThreadDemo2 {
  public static void main(String[] args) {
  Thread T = new Thread(new MyRunnable());
  T.start();
  }
}
```

1.3 方法三：

继承 Thread，重写 run，使用匿名内部类的方式

```
public class ThreadDemo3 {
 public static void main(String[] args) {
 // 这个语法是匿名内部类
 // 相当于创建了一个匿名的类，这个类是继承了 Thread
```

```
// 此处 new 的实例，其实是 new 了这个新的子类的实例，
Thread t = new Thread(){
@Override
public void run() {
while(true){
 System.out.println("hello Thread!");
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 };
 t.start();
 }
}
```

1.4 方法四：

实现Runnable，重写run，使用匿名内部类

```
public class ThreadDemo4 {
 public static void main(String[] args) {
 Thread t = new Thread(new Runnable() {
 @Override
 public void run() {
 while(true){
 System.out.println("hello Thread!");
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 });
 t.start();
```

```
 }
}
```

1.5 方法五:

使用 lambda 表达式创建线程,(直接了当指定了任务)

lambda 本质上就是一个匿名函数，java语言是通过函数式接口的方式来实现的，为了追求时髦，破坏了java语法的一致性。

```
public class ThreadDemo5 {
 public static void main(String[] args) {
 Thread t = new Thread(()->{
 while(true){
 System.out.println("hello Thread!");
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
  }
  }
  }) ;
  t.start();
  }
}
```

注:以上这些创建线程的方式，本质都相同，都是借助 Thread 类，在内核中创建新的 PCB，加入到内核的双向链表中。只不过区别是，指定线程要执行任务的方式不一样。此处的区别，其实都只是 Java 语法层面的区别。

2、Thread 的几个常见属性

属性	获取方法
ID	getId()
名称	getName()
状态	getState()
优先级	getPriority()
是否后台线程	isDaemon()
是否存活	isAlive()
是否被中断	isInterrupted()

- ID是线程的唯一标识，不同线程不会重复
- 名称是各种调试工具用到
- 状态表示线程当前所处的一个情况，下面我们会进一步说明
- 优先级高的线程理论上来说更容易被调度到
- 关于后台线程，需要记住一点：JVM会在一个进程的所有非后台线程结束后，才会结束运行。
- 是否存活，即简单的理解，为 run 方法是否运行结束了
- 线程的中断问题，下面我们进一步说明

```
public class ThreadDemo6 {
 public static void main(String[] args) {
 Thread t = new Thread(new Runnable() {
 @Override
 public void run() {
 while(true){
 // 打印当前线程的名字
 // Thread.currentThread() 这个静态方法,来获取当前线程实例
 // 哪个线程调用的这个方法.就能获取到对应的实例
  System.out.println(Thread.currentThread().getName());
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 },"mythread");
```

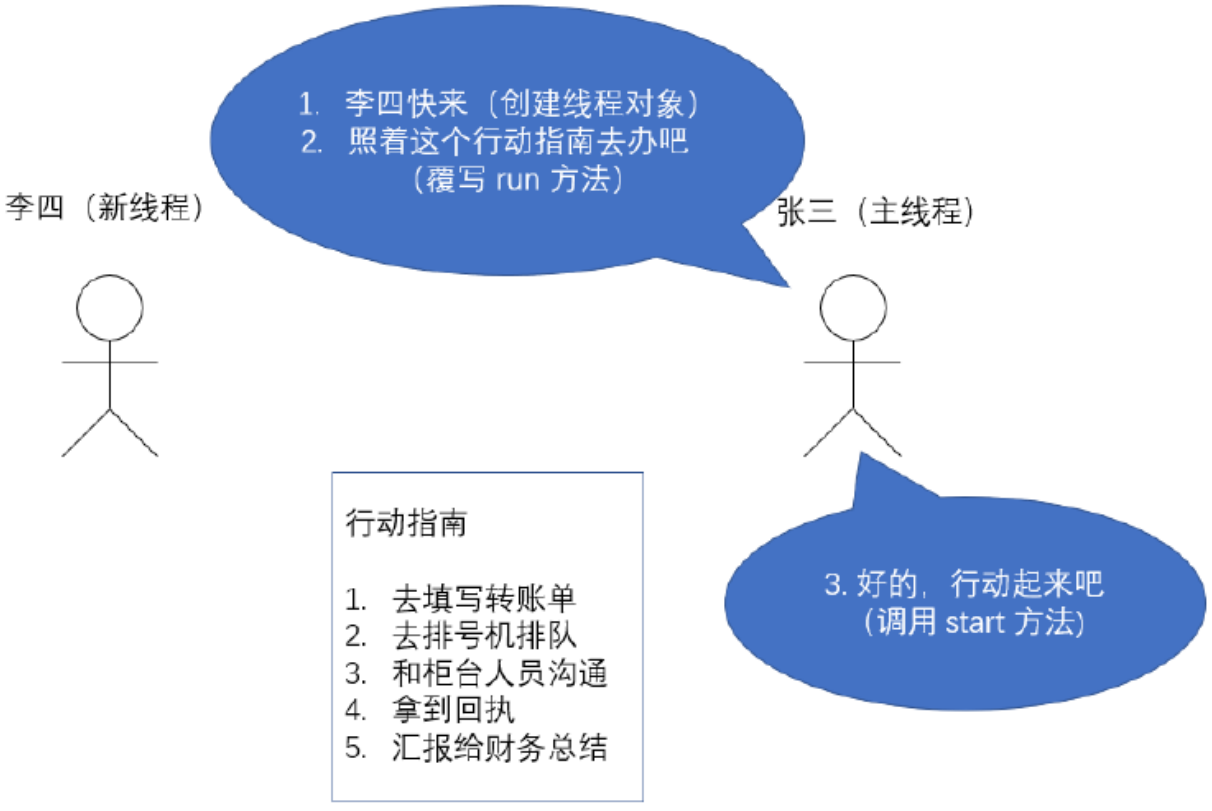
```
19    t.start();
20    // 打印线程的属性
21    System.out.println("id: " + t.getId());
22    System.out.println("name: " + t.getName());
23    System.out.println("state: " + t.getState());
24    System.out.println("priority: " + t.getPriority());
25    System.out.println("isDaemon: " + t.isDaemon() );
26    System.out.println("isInterupted: " + t.isInterrupted());
27    System.out.println("isLive: " + t.isAlive());
28    }
29  }
```

3、启动一个线程-start()

```
1  通过覆写 run 方法创建一个线程对象，但线程对象被创建出来并不意味着线程
```

就开始运行了。

- 覆写 run 方法是提供给线程要做的事情的指令清单
- 线程对象可以认为是把 李四、王五叫过来了
- 而调用 start() 方法，就是喊一声：”行动起来！ “，线程才真正独立去执行了。



调用 start 方法，才真的在操作系统的底层创建出一个线程.

4、中断一个线程

让线程结束，就是让内核里的 PCB 被销毁。

让线程结束的关键：就是让线程对应的入口方法（继承 Thread 重写的 run 方法、实现Runnable 重写的 run、lambda表达式），执行完毕。

```
public void run（）{
 System.out.println("hello Thread!");
}
```

像这种情况，只要 run 执行完毕，线程就随之结束了！

更多的情况，线程不一定这么快就执行完 run 方法，比如：

```
public void run() {
 while(true){
 // 打印当前线程的名字
 // Thread.currentThread() 这个静态方法,来获取当前线程实例
 // 哪个线程调用的这个方法.就能获取到对应的实例
 System.out.println(Thread.currentThread().getName());
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
  e.printStackTrace();
  }
  }
}
```

如果 run 方法里是一个死循环，此时的线程就会一直持续运行，直到进程结束。实际开发中，并不希望线程的 run 就是一个死循环，更希望能够控制这个线程，按照实际需求随时结束。

为了实现这需求，有两种方法：

1.通过共享的标记来进行沟通（使用 boolean 变量作为循环结束的标记）

2.调用 interrupt() 方法来通知

获取线程内置的标记位：线程的 isInterrupted() 判定当前的线程是不是应该要结束循环。

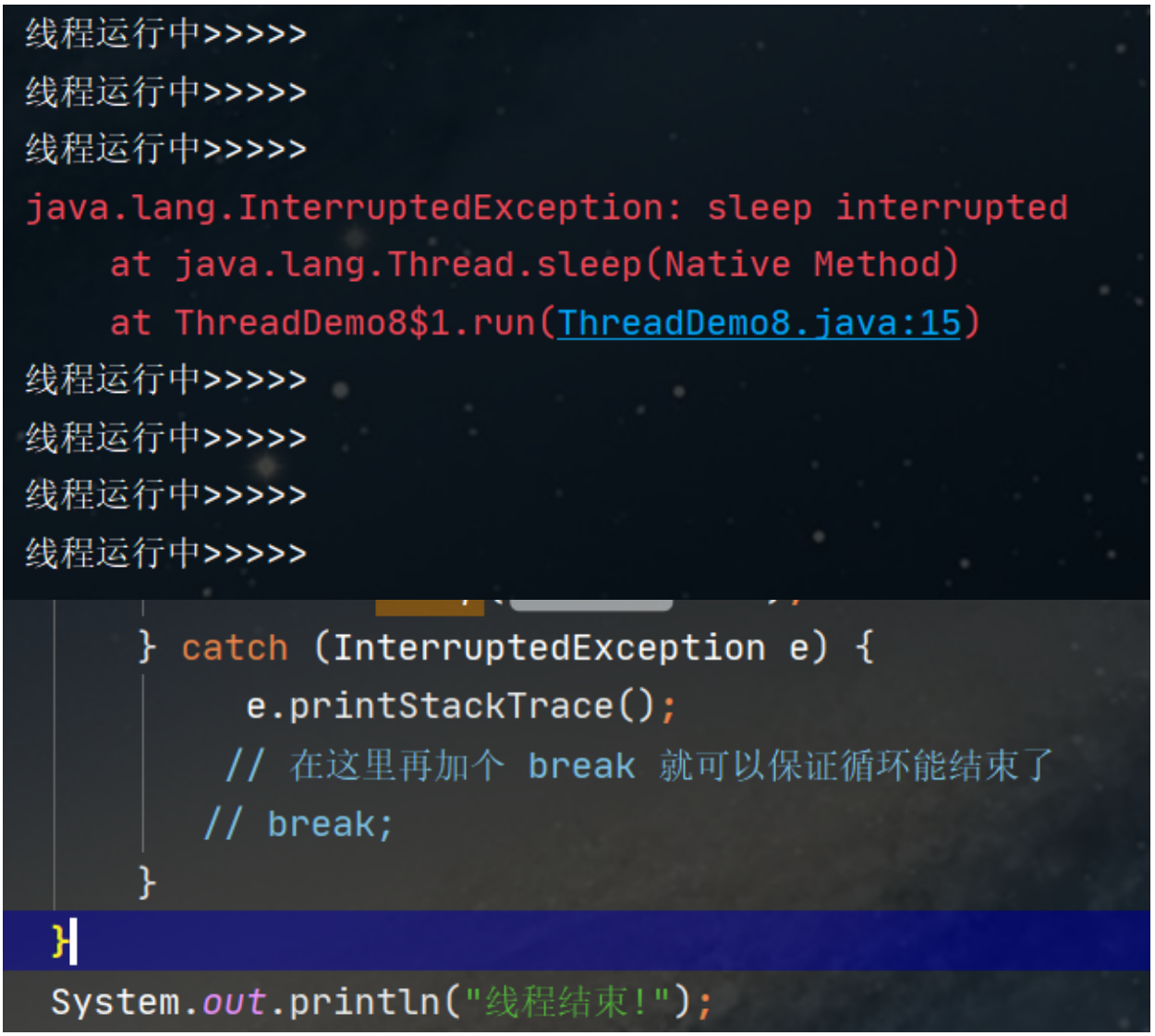
修改线程内置的标记位：Thread.interrupt() 来修改这个标记位。

```
public class ThreadDemo7 {
 private static boolean flag = true;
 public static void main(String[] args) {
 Thread t = new Thread(){
 @Override
 public void run() {
 while(flag){
 System.out.println("线程运行中>>>>>");
 try {
  Thread.sleep(1000);
  } catch (InterruptedException e) {
  e.printStackTrace();
  }
  }
  System.out.println("线程结束!");
  }
  };
  t.start();

  // 主循环中也等待三秒
  try {
  Thread.sleep(3000);
  } catch (InterruptedException e) {
  e.printStackTrace();
  }

  // 三秒钟之后,就把flag 改成 flag
  flag = false;
  }
}
```

```
public class ThreadDemo8 {
  public static void main(String[] args) throws InterruptedExcept
ion {
```

```
3   Thread t = new Thread(){
4   @Override
5   public void run() {
6   // 默认情况 isInterrupted 值为 false
7   while( !Thread.currentThread().isInterrupted()){
8   System.out.println("线程运行中>>>>>");
9   try {
10   Thread.sleep(1000);
11   } catch (InterruptedException e) {
12   // e.printStackTrace();
13   // 在这里再加个 break 就可以保证循环能结束了
14   break;
15   }
16   }
17   System.out.println("线程结束!");
18   }
19   };
20   t.start();
21   // 在主线程中,通过 t.interrupt() 方法来设置这个标记位
22   Thread.sleep(3000);
23   // 这个操作就是把 Thread.currentThread().isInterrupted() 给设置
 成 true
24   t.interrupt();
25   }
26  }
```

3秒之后，interrupt 方法并没有修改这个标记位，循环还在继续执行同时，还会出现一个异常：



这里的 interrupt（） 方法可能有两种行为：

1.如果当前线程正在进行中，此时就会修改 Thread.currentThread().isInterrupted() 标记位为 true

2.如果当前线程正在sleep、wait、等待锁……，此时就会触发 InterruptedException 异常，此时要不要结束线程取决于catch代码中写法，可以选择忽略这个异常，也可以跳出循环结束线程！

方法	说明
public void interrupt()	中断对象关联的线程，如果线程正在阻塞，则以异常方式通知，否则设置标志位
public static boolean interrupted()	判断当前线程的中断标志位是否设置，调用后清除标志位
public boolean isInterrupted()	判断对象关联的线程的标志位是否设置，调用后不清除标志位

注：isInterrupted() 和 interrupted() 的区别：

1.isInterrupted() 是 Thread 的实例方法，interrupted() 是Thread 的类方法。

2.当调用静态的 interrupted() 方法来判定标记位时，就会返回 true，同时把把标记位再改回成 false，下次再调用 interrupted() 就返回 false；当调用非静态的 isInterrupted() 来判断标记位，也会返回 true，但是不会对标记位进行修改，然后再调用 isInterrupted() 的时候就仍然返回的时 true。

使用 Thread.interrupted()，线程中断会清除标志位.

```
1  public class ThreadDemo3 {
2   private static class MyRunnable implements Runnable {
3   @Override
4   public void run() {
5   for (int i = 0; i < 10; i++) {
6   System.out.println(Thread.interrupted());
7   }
8   }
9   }
10   public static void main(String[] args) throws InterruptedExcep
 tion {
11    MyRunnable target = new MyRunnable();
12    Thread thread = new Thread(target, "李四");
13    thread.start();
14    thread.interrupt();
15    }
16  }
17
18  true // 只有一开始是 true，后边都是 false，因为标志位被清
19  false
20  false
21  false
22  false
23  false
24  false
25  false
```

```
false
false
```

使用 Thread.currentThread().isInterrupted()，线程中断标记位不会清除.

```
public class ThreadDemo4 {
 private static class MyRunnable implements Runnable {
 @Override
 public void run() {
 for (int i = 0; i < 10; i++) {
 System.out.println(Thread.currentThread().isInterrupted());
 }
 }
 }
 public static void main(String[] args) throws InterruptedExcep
tion {
 MyRunnable target = new MyRunnable();
 Thread thread = new Thread(target, "李四");
 thread.start();
 thread.interrupt();
 }
}

true // 全部是 true，因为标志位没有被清
true
true
true
true
true
true
true
true
true
```

5、等待一个线程-join（）

线程和线程之间，调度顺序是完全不确定的（取决于操作系统调度器自身的实现）。但是实际中，有时候我们希望是可控的，此时线程等待就是一种方法。所谓的线程等待，主要指的是控制线程结束的先后顺序。

常见的逻辑：t1线程，创t2、t3、t4，让三个新的线程来分别执行一些任务，然后t1线程最后汇总结果，这样的情况需要t1结束时机就必须比t2,t3,t4都迟。

```
Thread t = new Thread();
t.join(); // 执行到这个代码线程就会阻塞等待
```

所谓的阻塞等待就是操作系统短时间不会把这个线程调度到 CPU 上了

```
t1.start();
try{
 // 此处 join 就会阻塞等待
 t1.join();
}
```

执行 start 方法的时候,就会立即创建一个新的线程来,同时 main 这个线程也立刻往下执行,就执行到t1.join(),执行到t1.join()的时候就发现,当前t1线程还在运行中,join就会一直阻塞等待,一直等到t1线程执行结束

附录

方法	说明
public void join()	等待线程结束
public void join(long millis)	等待线程结束，最多等 millis 毫秒
public void join(long millis, int nanos)	同理，但可以更高精度

join 无参数版本:相当于死等

join 有参数版本,参数就是最大等待时间

6、获取当前的线程应用

currentThread () 能够获取到当前线程对应的 Thread 实例的应用

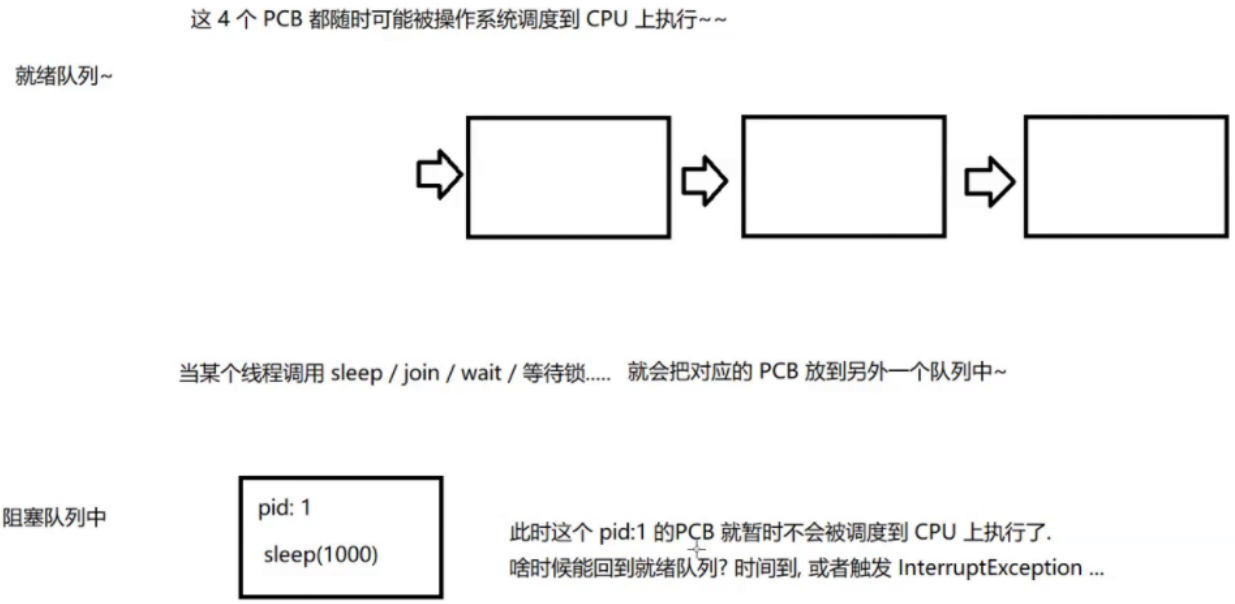
```
public class ThreadDemo6 {
 public static void main(String[] args) {
 Thread thread = new Thread(){
 @Override
```

```
5   public void run() {
6   System.out.println(Thread.currentThread().getId());
7   // System.out.println(this.getName());
8   }
9   };
10   }
11  }
```

在这个代码中,看起来就好像 this 和 Thread.currentThread() 区别不大,实际中没区别的前提,是使用继承 Thread ,重写 run 的方式创建线程。如果是通过 Runnable 或者 lambda 的方式，就是不行的

7、休眠当前的线程

sleep 这个方法，本质上是把线程 PCB 从就绪队列移动到阻塞队列。



因为线程的调度是不可控的，所以，这个方法只能保证实际休眠时间是大于等于参数设置的休眠时间的。

方法	说明
public static void sleep(long millis) throws InterruptedException	休眠当前线程 millis 毫秒
public static void sleep(long millis, int nanos) throws InterruptedException	可以更高精度的休眠

```
1  public class ThreadDemo7 {
2   public static void main(String[] args) throws InterruptedExcept
 ion {
```

```
3   System.out.println(System.currentTimeMillis());
4   Thread.sleep(3 * 1000);
5   System.out.println(System.currentTimeMillis());
6   }
7 }
```