线程或者说多线程，是我们处理多任务的强大工具。线程和进程是不同的，每个进程都是一个独立运行的程序，拥有自己的变量，且不同进程间的变量不能共享；而线程是运行在进程内部的，每个正在运行的进程至少有一个线程，而且不同的线程之间可以在进程范围内共享数据。也就是说进程有自己独立的存储空间，而线程是和它所属的进程内的其他线程共享一个存储空间。线程的使用可以使我们能够并行地处理一些事情。线程通过并行的处理给用户带来更好的使用体验，比如你使用的邮件系统（outlook、Thunderbird、foxmail 等），你当然不希望它们在收取新邮件的时候，导致你连已经收下来的邮件都无法阅读，而只能等待收取邮件操作执行完毕。这正是线程的意义所在。

实现线程的方式

实现线程的方式有两种：

1. 继承 java.lang.Thread，并重写它的 run()方法，将线程的执行主体放入其中。
2. 实现 java.lang.Runnable 接口，实现它的 run()方法，并将线程的执行主体放入其中。

这是继承 Thread 类实现线程的示例：

Java 代码

```
1. public class ThreadTest extends Thread {
2.     public void run() {
3.         // 在这里编写线程执行的主体
4.         // do something
5.     }
6. }
```

这是实现 Runnable 接口实现多线程的示例：

Java 代码

```
1. public class RunnableTest implements Runnable {
2.     public void run() {
3.         // 在这里编写线程执行的主体
4.         // do something
5.     }
6. }
```

这两种实现方式的区别并不大。继承 Thread 类的方式实现起来较为简单，但是继承它的类就不能再继承别的类了，因此也就不能继承别的类的有用的方法了。而使用是想 Runnable 接口的方式就不存在这个问题了，而且这种实现方式将线程主体和线程对象本身分离开来，逻辑上也较为清晰，所以推荐大家更多地采用这种方式。

如何启动线程

我们通过以上两种方式实现了一个线程之后，线程的实例并没有被创建，因此它们也并没有被运行。我们要启动一个线程，必须调用方法来启动它，这个方法就是 Thread 类的 start()方法，而不是 run()方法（既不是我们继承 Thread 类重写的 run()方法，也不是实现 Runnable 接口的 run()方法）。run()方法中包含的是线程的主体，也就是这个线程被启动后将要运行的代码，它跟线程的启动没有任何关系。上面两种实现线程的方式在启动时会有所不同。

继承 Thread 类的启动方式：

Java 代码

```
1. public class ThreadStartTest {
2.     public static void main(String[] args) {
3.         // 创建一个线程实例
4.         ThreadTest tt = new ThreadTest();
5.         // 启动线程
6.         tt.start();
7.     }
8. }
```

实现 Runnable 接口的启动方式：

Java 代码

```
1. public class RunnableStartTest {
2.     public static void main(String[] args) {
3.         // 创建一个线程实例
4.         Thread t = new Thread(new RunnableTest());
5.         // 启动线程
6.         t.start();
7.     }
8. }
```

实际上这两种启动线程的方式原理是一样的。首先都是调用本地方法启动一个线程，其次是在这个线程里执行目标对象的 run()方法。那么这个目标对象是什么呢？为了弄明白这个问题，我们来看看 Thread 类的 run()方法的实现：

Java 代码

```
1. public void run() {
2.     if (target != null) {
3.         target.run();
4.     }
5. }
```

当我们采用实现 Runnable 接口的方式来实现线程的情况下，在调用 new Thread(Runnable target)构造器时，将实现 Runnable 接口的类的实例设置成了线程要执行的主体所属的目标对象 target，当线程启动时，这个实例的 run()方法就被执行了。当我们采用继承 Thread 的方式实现线程时，线程的这个 run()方法被重写了，所以当线程启动时，执行的是这个对象自身的 run()方法。总结起来就一句话，线程类有一个 Runnable 类型的 target 属性，它是线程启动后要执行的 run()方法所属的主体，如果我们采用的是继承 Thread 类的方式，那么这个 target 就是线程对象自身，如果我们采用的是实现 Runnable 接口的方式，那么这个 target 就是实现了 Runnable 接口的类的实例。

线程的状态

在 Java 1.4 及以下的版本中，每个线程都具有新建、可运行、阻塞、死亡四种状态，但是在 Java 5.0 及以上版本中，线程的状态被扩充为新建、可运行、阻塞、等待、定时等待、死亡六种。线程的状态完全包含了一个线程从新建到运行，最后到结束的整个生命周期。线程状态的具体信息如下：

1. **NEW（新建状态、初始化状态）：** 线程对象已经被创建，但是还没有被启动时的状态。这段时间就是在我们调用 new 命令之后，调用 start()方法之前。
2. **RUNNABLE（可运行状态、就绪状态）：** 在我们调用了线程的 start()方法之后线程所处的状态。处于 RUNNABLE 状态的线程在 JAVA 虚拟机（JVM）上是运行着的，但是它可能还正在等待操作系统分配给它相应的运行资源以得以运行。
3. **BLOCKED（阻塞状态、被中断运行）：** 线程正在等待其它的线程释放同步锁，以进入一个同步块或者同步方法继续运行；或者它已经进入了某个同步块或同步方法，在运行的过程中它调用了某个对象继承自 java.lang.Object 的 wait()方法，正在等待重新返回这个同步块或同步方法。

4. **WAITING（等待状态）：**当前线程调用了 java.lang.Object.wait()、java.lang.Thread.join()或者 java.util.concurrent.locks.LockSupport.park()三个中的任意一个方法，正在等待另外一个线程执行某个操作。比如一个线程调用了某个对象的 wait()方法，正在等待其它线程调用这个对象的 notify()或者 notifyAll()（这两个方法同样是继承自 Object 类）方法来唤醒它；或者一个线程调用了另一个线程的 join()（这个方法属于 Thread 类）方法，正在等待这个方法运行结束。
5. **TIMED_WAITING（定时等待状态）：**当前线程调用了 java.lang.Object.wait(long timeout)、java.lang.Thread.join(long millis)、java.util.concurrent.locks.LockSupport.packNanos(long nanos)、java.util.concurrent.locks.LockSupport.packUntil(long deadline)四个方法中的任意一个，进入等待状态，但是与 WAITING 状态不同的是，它有一个最大等待时间，即使等待的条件仍然没有满足，只要到了这个时间它就会自动醒来。
6. **TERMINATED（死亡状态、终止状态）：**线程完成执行后的状态。线程执行完 run()方法中的全部代码，从该方法中退出，进入 TERMINATED 状态。还有一种情况是 run()在运行过程中抛出了一个异常，而这个异常没有被程序捕获，导致这个线程异常终止进入 TERMINATED 状态。

在 Java5.0 及以上版本中，线程的全部六种状态都以枚举类型的形式定义在 java.lang.Thread 类中了，代码如下：

Java 代码

```
1. public enum State {
2.      NEW,
3.      RUNNABLE,
4.      BLOCKED,
5.      WAITING,
6.      TIMED_WAITING,
7.      TERMINATED;
8. }
```

sleep()和 wait()的区别

sleep()方法和 wait()方法都成产生让当前运行的线程停止运行的效果，这是它们的共同点。下面我们来详细说说它们的不同之处。

sleep()方法是本地方法，属于 Thread 类，它有两种定义：

Java 代码

```
1. public static native void sleep(long millis) throws Interrupted
   Exception;
2.
3. public static void sleep(long millis, int nanos) throws Interru
   ptedException {
4.     //other code
5. }
```

其中的参数 millis 代表毫秒数（千分之一秒），nanos 代表纳秒数（十亿分之一秒）。这两个方法都可以让调用它的线程沉睡（停止运行）指定的时间，到了这个时间，线程就会自动醒来，变为可运行状态（RUNNABLE），但这并不表示它马上就会被运行，因为线程调度机制恢复线程的运行也需要时间。调用 sleep()方法并不会让线程释放它所持有的同步锁；而且在这期间它也不会阻碍其它线程的运行。上面的连个方法都声明抛出一个 InterruptedException 类型的异常，这是因为线程在 sleep()期间，有可能被持有它的引用的其它线程调用它的 interrupt()方法而中断。中断一个线程会导致一个 InterruptedException 异常的产生，如果你的程序不捕获这个异常，线程就会异常终止，进入 TERMINATED 状态，如果你的程序捕获了这个异常，那么程序就会继续执行 catch 语句块（可能还有 finally 语句块）以及以后的代码。

为了更好地理解 interrupt()效果，我们来看一下下面这个例子：

Java 代码

```
1. public class InterruptTest {
2.     public static void main(String[] args) {
3.         Thread t = new Thread() {
4.             public void run() {
5.                 try {
6.                     System.out.println("我被执行了-在 sleep()方
   法前");
7.                     // 停止运行 10 分钟
8.                     Thread.sleep(1000 * 60 * 60 * 10);
9.                     System.out.println("我被执行了-在 sleep()方
   法后");
10.                } catch (InterruptedException e) {
11.                    System.out.println("我被执行了-在 catch 语句
   块中");
12.                }
13.                System.out.println("我被执行了-在 try{}语句块后
   ");
14.            }
15.        };
```

```
16.             // 启动线程
17.             t.start();
18.             // 在 sleep()结束前中断它
19.             t.interrupt();
20.     }
21.}
```

运行结果：

```
1. 我被执行了-在 sleep()方法前
2. 我被执行了-在 catch 语句块中
3. 我被执行了-在 try{}语句块后
```

wait()方法也是本地方法，属于 Object 类，有三个定义：

Java 代码

```
1. public final void wait() throws InterruptedException {
2.     //do something
3. }
4.
5. public final native void wait(long timeout) throws InterruptedE
   xception;
6.
7. public final void wait(long timeout, int nanos) throws Interrup
   tedException {
8.     //do something
9. }
```

wari()和 wait(long timeout,int nanos)方法都是基于 wait(long timeout)方法实现的。同样地，timeout 代表毫秒数，nanos 代表纳秒数。当调用了某个对象的 wait()方法时，当前运行的线程就会转入等待状态（WAITING），等待别的线程再次调用这个对象的 notify()或者 notifyAll()方法(这两个方法也是本地方法）唤醒它，或者到了指定的最大等待时间，线程自动醒来。如果线程拥有某个或某些对象的同步锁，那么在调用了 wait()后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了 wait()方法的对象。wait()方法同样会被 Thread 类的 interrupt()方法中断,并产生一个 InterruptedException 异常，效果同 sleep()方法被中断一样。

实现同步的方式

同步是多线程中的重要概念。同步的使用可以保证在多线程运行的环境中，程序不会产生设计之外的错误结果。同步的实现方式有两种，同步方法和同步块，这两种方式都要用到 synchronized 关键字。

给一个方法增加 synchronized 修饰符之后就可以使它成为同步方法，这个方法可以是静态方法和非静态方法，但是不能是抽象类的抽象方法，也不能是接口中的接口方法。下面代码是一个同步方法的示例：

Java 代码

```
1. public synchronized void aMethod() {
2.     // do something
3. }
4.
5. public static synchronized void anotherMethod() {
6.     // do something
7. }
```

线程在执行同步方法时是具有排它性的。当任意一个线程进入到一个对象的任意一个同步方法时，这个对象的所有同步方法都被锁定了，在此期间，其他任何线程都不能访问这个对象的任意一个同步方法，直到这个线程执行完它所调用的同步方法并从中退出，从而导致它释放了该对象的同步锁之后。在一个对象被某个线程锁定之后，其他线程是可以访问这个对象的所有非同步方法的。

同步块的形式虽然与同步方法不同，但是原理和效果是一致的。同步块是通过锁定一个指定的对象，来对同步块中包含的代码进行同步；而同步方法是对这个方法块里的代码进行同步，而这种情况下锁定的对象就是同步方法所属的主体对象自身。如果这个方法是静态同步方法呢？那么线程锁定的就不是这个类的对象了，也不是这个类自身，而是这个类对应的 java.lang.Class 类型的对象。同步方法和同步块之间的相互制约只限于同一个对象之间，所以静态同步方法只受它所属类的其它静态同步方法的制约，而跟这个类的实例（对象）没有关系。

下面这段代码演示了同步块的实现方式：

Java 代码

```
1. public void test() {
2.     // 同步锁
3.     String lock = "LOCK";
4.
5.     // 同步块
6.     synchronized (lock) {
7.         // do something
8.     }
```

```
9.
10.     int i = 0;
11.     // ...
12.}
```

对于作为同步锁的对象并没有什么特别要求，任意一个对象都可以。如果一个对象既有同步方法，又有同步块，那么当其中任意一个同步方法或者同步块被某个线程执行时，这个对象就被锁定了，其他线程无法在此时访问这个对象的同步方法，也不能执行同步块。

synchronized 和 Lock

Lock 是一个接口，它位于 Java 5.0 新增的 java.utils.concurrent 包的子包 locks 中。concurrent 包及其子包中的类都是用来处理多线程编程的。实现 Lock 接口的类具有与 synchronized 关键字同样的功能，但是它更加强大一些。java.utils.concurrent.locks.ReentrantLock 是较常用的实现了 Lock 接口的类。下面是 ReentrantLock 类的一个应用实例：

Java 代码

```
1. private Lock lock = new ReentrantLock();
2.
3. public void testLock() {
4.     // 锁定对象
5.     lock.lock();
6.     try {
7.         // do something
8.     } finally {
9.         // 释放对对象的锁定
10.        lock.unlock();
11.    }
12.}
```

lock()方法用于锁定对象，unlock()方法用于释放对对象的锁定，他们都是在 Lock 接口中定义的方法。位于这两个方法之间的代码在被执行时，效果等同于被放在 synchronized 同步块中。一般用法是将需要在 lock()和 unlock()方法之间执行的代码放在 try{}块中，并且在 finally{}块中调用 unlock()方法，这样就可以保证即使在执行代码抛出异常的情况下，对象的锁也总是会被释放，否则的话就会为死锁的产生增加可能。

使用 synchronized 关键字实现的同步，会把一个对象的所有同步方法和同步块看做一个整体，只要有一个被某个线程调用了，其他的就无法被别的线程执行，

即使这些方法或同步块与被调用的代码之间没有任何逻辑关系，这显然降低了程序的运行效率。而使用 Lock 就能够很好地解决这个问题。我们可以把一个对象中按照逻辑关系把需要同步的方法或代码进行分组，为每个组创建一个 Lock 类型的对象，对实现同步。那么，当一个同步块被执行时，这个线程只会锁定与当前运行代码相关的其他代码最小集合，而并不影响其他线程对其余同步代码的调用执行。

关于死锁

死锁就是一个进程中的每个线程都在等待这个进程中的其他线程释放所占用的资源，从而导致所有线程都无法继续执行的情况。死锁是多线程编程中一个隐藏的陷阱，它经常发生在多个线程共用资源的时候。在实际开发中，死锁一般隐藏的较深，不容易被发现，一旦死锁现象发生，就必然会导致程序的瘫痪。因此必须避免它的发生。

程序中必须同时满足以下四个条件才会引发死锁：

1. **互斥（Mutual exclusion）：**线程所使用的资源中至少有一个是不能共享的，它在同一时刻只能由一个线程使用。
2. **持有与等待（Hold and wait）：**至少有一个线程已经持有了资源，并且正在等待获取其他的线程所持有的资源。
3. **非抢占式（No pre-emption）：**如果一个线程已经持有了某个资源，那么在这个线程释放这个资源之前，别的线程不能把它抢夺过去使用。
4. **循环等待（Circular wait）：**假设有 N 个线程在运行，第一个线程持有了一个资源，并且正在等待获取第二个线程持有的资源，而第二个线程正在等待获取第三个线程持有的资源，依此类推……第 N 个线程正在等待获取第一个线程持有的资源，由此形成一个循环等待。

线程池

线程池就像数据库连接池一样，是一个对象池。所有的对象池都有一个共同的目的，那就是为了提高对象的使用率，从而达到提高程序效率的目的。比如对于 Servlet，它被设计为多线程的（如果它是单线程的，你就可以想象，当 1000 个人同时请求一个网页时，在第一个人获得请求结果之前，其它 999 个人都在郁闷地等待），如果为每个用户的每一次请求都创建一个新的线程对象来运行的话，系统就会在创建线程和销毁线程上耗费很大的开销，大大降低系统的效率。因此，Servlet 多线程机制背后有一个线程池在支持，线程池在初始化初期就创建了一定数量的线程对象，通过提高对这些对象的利用率，避免高频率地创建对象，从而达到提高程序的效率的目的。

下面实现一个最简单的线程池，从中理解它的实现原理。为此我们定义了四个类，它们的用途及具体实现如下：

1. **Task（任务）：**这是个代表任务的抽象类，其中定义了一个 deal()方法，继承 Task 抽象类的子类需要实现这个方法，并把这个任务需要完成的具体工作在 deal()方法编码实现。线程池中的线程之所以被创建，就是为了执行各种各样数量繁多的任务的，为了方便线程对任务的处理，我们需要用 Task 抽象类来保证任务的具体工作统一放在 deal()方法里来完成，这样也使代码更加规范。

   Task 的定义如下：

   Java 代码

```
1. public abstract class Task {
2.     public enum State {
3.         /* 新建 */NEW, /* 执行中 */RUNNING, /* 已完成 */FINISHED
4.     }
5.
6.     // 任务状态
7.     private State state = State.NEW;
8.
9.     public void setState(State state) {
10.        this.state = state;
11.    }
12.
13.    public State getState() {
14.        return state;
15.    }
16.
17.    public abstract void deal();
18.}
```

2. **TaskQueue（任务队列）：**在同一时刻，可能有很多任务需要执行，而程序在同一时刻只能执行一定数量的任务，当需要执行的任务数超过了程序所能承受的任务数时怎么办呢？这就有了先执行哪些任务，后执行哪些任务的规则。TaskQueue 类就定义了这些规则中的一种，它采用的是 FIFO（先进先出，英文名是 First In First Out）的方式，也就是按照任务到达的先后顺序执行。

   TaskQueue 类的定义如下：

   Java 代码

```
1. import java.util.Iterator;
```

```
2. import java.util.LinkedList;
3. import java.util.List;
4.
5. public class TaskQueue {
6.     private List<Task> queue = new LinkedList<Task>();
7.
8.     // 添加一项任务
9.     public synchronized void addTask(Task task) {
10.         if (task != null) {
11.             queue.add(task);
12.         }
13.     }
14.
15.     // 完成任务后将它从任务队列中删除
16.     public synchronized void finishTask(Task task) {
17.         if (task != null) {
18.             task.setState(Task.State.FINISHED);
19.             queue.remove(task);
20.         }
21.     }
22.
23.     // 取得一项待执行任务
24.     public synchronized Task getTask() {
25.         Iterator<Task> it = queue.iterator();
26.         Task task;
27.         while (it.hasNext()) {
28.             task = it.next();
29.             // 寻找一个新建的任务
30.             if (Task.State.NEW.equals(task.getState())) {

31.                 // 把任务状态置为运行中
32.                 task.setState(Task.State.RUNNING);
33.                 return task;
34.             }
35.         }
36.         return null;
37.     }
38.}
```

addTask(Task task)方法用于当一个新的任务到达时，将它添加到任务队列中。这里使用了 LinkedList 类来保存任务到达的先后顺序。
finishTask(Task task)方法用于任务被执行完毕时，将它从任务队列中清除出去。getTask()方法用于取得当前要执行的任务。

3. **TaskThread（执行任务的线程）：**它继承自 Thread 类，专门用于执行任务队列中的待执行任务。

Java 代码

```
1. public class TaskThread extends Thread {
2.     // 该线程所属的线程池
3.     private ThreadPoolService service;
4.
5.     public TaskThread(ThreadPoolService tps) {
6.         service = tps;
7.     }
8.
9.     public void run() {
10.        // 在线程池运行的状态下执行任务队列中的任务
11.        while (service.isRunning()) {
12.            TaskQueue queue = service.getTaskQueue();
13.            Task task = queue.getTask();
14.            if (task != null) {
15.                task.deal();
16.            }
17.            queue.finishTask(task);
18.        }
19.    }
20.}
```

4. **ThreadPoolService（线程池服务类）：**这是线程池最核心的一个类。它在被创建了时候就创建了几个线程对象，但是这些线程并没有启动运行，但调用了 start()方法启动线程池服务时，它们才真正运行。stop()方法可以停止线程池服务，同时停止池中所有线程的运行。而 runTask(Task task)方法是将一个新的待执行任务交与线程池来运行。
ThreadPoolService 类的定义如下：

Java 代码

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class ThreadPoolService {
5.     // 线程数
6.     public static final int THREAD_COUNT = 5;
7.
8.     // 线程池状态
9.     private Status status = Status.NEW;
```

```
10.
11.     private TaskQueue queue = new TaskQueue();
12.
13.     public enum Status {
14.         /* 新建 */NEW, /* 提供服务中 */RUNNING, /* 停止服
    务 */TERMINATED,
15.     }
16.
17.     private List<Thread> threads = new ArrayList<Thread>(
    );
18.
19.     public ThreadPoolService() {
20.         for (int i = 0; i < THREAD_COUNT; i++) {
21.             Thread t = new TaskThread(this);
22.             threads.add(t);
23.         }
24.     }
25.
26.     // 启动服务
27.     public void start() {
28.         this.status = Status.RUNNING;
29.         for (int i = 0; i < THREAD_COUNT; i++) {
30.             threads.get(i).start();
31.         }
32.     }
33.
34.     // 停止服务
35.     public void stop() {
36.         this.status = Status.TERMINATED;
37.     }
38.
39.     // 是否正在运行
40.     public boolean isRunning() {
41.         return status == Status.RUNNING;
42.     }
43.
44.     // 执行任务
45.     public void runTask(Task task) {
46.         queue.addTask(task);
47.     }
48.
49.     protected TaskQueue getTaskQueue() {
50.         return queue;
51.     }
```

```
52.}
```

完成了上面四个类，我们就实现了一个简单的线程池。现在我们就可以使用它了，下面的代码做了一个简单的示例：

Java 代码

```
1. public class SimpleTaskTest extends Task {
2.     @Override
3.     public void deal() {
4.         // do something
5.     }
6.
7.     public static void main(String[] args) throws InterruptedEx
   ception {
8.         ThreadPoolService service = new ThreadPoolService();
9.         service.start();
10.        // 执行十次任务
11.        for (int i = 0; i < 10; i++) {
12.            service.runTask(new SimpleTaskTest());
13.        }
14.        // 睡眠 1 秒钟，等待所有任务执行完毕
15.        Thread.sleep(1000);
16.        service.stop();
17.    }
18.}
```

当然，我们实现的是最简单的，这里只是为了演示线程池的实现原理。在实际应用中，根据情况的不同，可以做很多优化。比如：

- 调整任务队列的规则，给任务设置优先级，级别高的任务优先执行。
- 动态维护线程池，当待执行任务数量较多时，增加线程的数量，加快任务的执行速度；当任务较少时，回收一部分长期闲置的线程，减少对系统资源的消耗。

事实上 Java5.0 及以上版本已经为我们提供了线程池功能，无需再重新实现。这些类位于 java.util.concurrent 包中。

Executors 类提供了一组创建线程池对象的方法，常用的有一下几个：

Java 代码

```
1. public static ExecutorService newCachedThreadPool() {
2.     // other code
3. }
4.
5. public static ExecutorService newFixedThreadPool(int nThreads)
   {
6.     // other code
7. }
8.
9. public static ExecutorService newSingleThreadExecutor() {
10.    // other code
11.}
```

newCachedThreadPool()方法创建一个动态的线程池，其中线程的数量会根据实际需要来创建和回收，适合于执行大量短期任务的情况；
newFixedThreadPool(int nThreads)方法创建一个包含固定数量线程对象的线程池，nThreads 代表要创建的线程数，如果某个线程在运行的过程中因为异常而终止了，那么一个新的线程会被创建和启动来代替它；而
newSingleThreadExecutor()方法则只在线程池中创建一个线程，来执行所有的任务。

这三个方法都返回了一个 ExecutorService 类型的对象。实际上，
ExecutorService 是一个接口，它的 submit()方法负责接收任务并交与线程池中的线程去运行。submit()方法能够接受 Callable 和 Runnable 两种类型的对象。它们的用法和区别如下：

1. **Runnable 接口：**继承 Runnable 接口的类要实现它的 run()方法，并将执行任务的代码放入其中，run()方法没有返回值。适合于只做某种操作，不关心运行结果的情况。
2. **Callable 接口：**继承 Callable 接口的类要实现它的 call()方法，并将执行任务的代码放入其中，call()将任务的执行结果作为返回值。适合于执行某种操作后，需要知道执行结果的情况。

无论是接收 Runnable 型参数，还是接收 Callable 型参数的 submit()方法，都会返回一个 Future（也是一个接口）类型的对象。该对象中包含了任务的执行情况以及结果。调用 Future 的 boolean isDone()方法可以获知任务是否执行完毕；调用 Object get()方法可以获得任务执行后的返回结果，如果此时任务还没有执行完，get()方法会保持等待，直到相应的任务执行完毕后，才会将结果返回。

我们用下面的一个例子来演示 Java5.0 中线程池的使用：

Java 代码

```
1. import java.util.concurrent.*;
2.
3. public class ExecutorTest {
4.     public static void main(String[] args) throws InterruptedEx
   ception,
5.             ExecutionException {
6.         ExecutorService es = Executors.newSingleThreadExecutor(
   );
7.         Future fr = es.submit(new RunnableTest());// 提交任
   务
8.
9.         Future fc = es.submit(new CallableTest());// 提交任
   务
10.        // 取得返回值并输出
11.        System.out.println((String) fc.get());
12.
13.        // 检查任务是否执行完毕
14.        if (fr.isDone()) {
15.            System.out.println("执行完毕
   -RunnableTest.run()");
16.        } else {
17.            System.out.println("未执行完
   -RunnableTest.run()");
18.        }
19.
20.        // 检查任务是否执行完毕
21.        if (fc.isDone()) {
22.            System.out.println("执行完毕
   -CallableTest.run()");
23.        } else {
24.            System.out.println("未执行完
   -CallableTest.run()");
25.        }
26.
27.        // 停止线程池服务
28.        es.shutdown();
29.    }
30.}
31.
```

```
32.class RunnableTest implements Runnable {
33.    public void run() {
34.        System.out.println("已经执行-RunnableTest.run()");
35.    }
36.}
37.
38.class CallableTest implements Callable {
39.    public Object call() {
40.        System.out.println("已经执行-CallableTest.call()");
41.        return "返回值-CallableTest.call()";
42.    }
43.}
```

运行结果：

1. 已经执行-RunnableTest.run()
2. 已经执行-CallableTest.call()
3. 返回值-CallableTest.call()
4. 执行完毕-RunnableTest.run()
5. 执行完毕-CallableTest.run()

使用完线程池之后，需要调用它的 shutdown()方法停止服务，否则其中的所有线程都会保持运行，程序不会退出。