第 3 章　ServerSocket 用法详解

在客户/服务器通信模式中，服务器端需要创建监听特定端口的 ServerSocket，ServerSocket 负责接收客户连接请求。本章首先介绍 ServerSocket 类的各个构造方法，以及成员方法的用法，接着介绍服务器如何用多线程来处理与多个客户的通信任务。

本章提供线程池的一种实现方式。线程池包括一个工作队列和若干工作线程。服务器程序向工作队列中加入与客户通信的任务，工作线程不断从工作队列中取出任务并执行它。本章还介绍了 java.util.concurrent 包中的线程池类的用法，在服务器程序中可以直接使用它们。

3.1　构造 ServerSocket

ServerSocket 的构造方法有以下几种重载形式：

- ServerSocket()throws IOException
- ServerSocket(int port) throws IOException
- ServerSocket(int port, int backlog) throws IOException
- ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

在以上构造方法中，参数 port 指定服务器要绑定的端口（服务器要监听的端口），参数 backlog 指定客户连接请求队列的长度，参数 bindAddr 指定服务器要绑定的 IP 地址。

3.1.1　绑定端口

除了第一个不带参数的构造方法以外，其他构造方法都会使服务器与特定端口绑定，该端口由参数 port 指定。例如，以下代码创建了一个与 80 端口绑定的服务器：

```
ServerSocket serverSocket=new ServerSocket(80);
```

如果运行时无法绑定到 80 端口，以上代码会抛出 IOException，更确切地说，是抛出 BindException，它是 IOException 的子类。BindException 一般是由以下原因造成的：

- 端口已经被其他服务器进程占用；
- 在某些操作系统中，如果没有以超级用户的身份来运行服务器程序，那么操作系统不允许服务器绑定到 1~1023 之间的端口。

如果把参数 port 设为 0，表示由操作系统来为服务器分配一个任意可用的端口。由操作系统分配的端口也称为匿名端口。对于多数服务器，会使用明确的端口，而不会使用匿名端口，因为客户程序需要事先知道服务器的端口，才能方便地访问服务器。在某些场合，匿名端口有着特殊的用途，本章 3.4 节会对此作介绍。

3.1.2 设定客户连接请求队列的长度

当服务器进程运行时，可能会同时监听到多个客户的连接请求。例如，每当一个客户进程执行以下代码：

```
Socket socket=new Socket(www.javathinker.org,80);
```

就意味着在远程 www.javathinker.org 主机的 80 端口上，监听到了一个客户的连接请求。管理客户连接请求的任务是由操作系统来完成的。操作系统把这些连接请求存储在一个先进先出的队列中。许多操作系统限定了队列的最大长度，一般为 50。当队列中的连接请求达到了队列的最大容量时，服务器进程所在的主机会拒绝新的连接请求。只有当服务器进程通过 ServerSocket 的 accept()方法从队列中取出连接请求，使队列腾出空位时，队列才能继续加入新的连接请求。

对于客户进程，如果它发出的连接请求被加入到服务器的队列中，就意味着客户与服务器的连接建立成功，客户进程从 Socket 构造方法中正常返回。如果客户进程发出的连接请求被服务器拒绝，Socket 构造方法就会抛出 ConnectionException。

ServerSocket 构造方法的 backlog 参数用来显式设置连接请求队列的长度，它将覆盖操作系统限定的队列的最大长度。值得注意的是，在以下几种情况中，仍然会采用操作系统限定的队列的最大长度：

- backlog 参数的值大于操作系统限定的队列的最大长度；
- backlog 参数的值小于或等于 0；
- 在 ServerSocket 构造方法中没有设置 backlog 参数。

以下例程 3-1 的 Client.java 和例程 3-2 的 Server.java 用来演示服务器的连接请求队列的特性。

例程 3-1　Client.java

```
import java.net.*;
public class Client {
   public static void main(String args[])throws Exception{
      final int length=100;
      String host="localhost";
      int port=8000;

      Socket[] sockets=new Socket[length];
      for(int i=0;i<length;i++){                    //试图建立 100 次连接
         sockets[i]=new Socket(host, port);
         System.out.println("第"+(i+1)+"次连接成功");
      }
      Thread.sleep(3000);
      for(int i=0;i<length;i++){
         sockets[i].close();                        //断开连接
      }
   }
}
```

例程 3-2　Server.java

```
import java.io.*;
import java.net.*;
public class Server {
   private int port=8000;
   private ServerSocket serverSocket;

   public Server() throws IOException {
      serverSocket = new ServerSocket(port,3);                    //连接请求队列的长度为 3
      System.out.println("服务器启动");
   }

   public void service() {
      while (true) {
         Socket socket=null;
         try {
            socket = serverSocket.accept();                       //从连接请求队列中取出一个连接
            System.out.println("New connection accepted " +
            socket.getInetAddress() + ":" +socket.getPort());
         }catch (IOException e) {
            e.printStackTrace();
         }finally {
            try{
               if(socket!=null)socket.close();
            }catch (IOException e) {e.printStackTrace();}
         }
      }
   }

   public static void main(String args[])throws Exception {
      Server server=new Server();
      Thread.sleep(60000*10);                                     //睡眠 10 分钟
      //server.service();
   }
}
```

Client 试图与 Server 进行 100 次连接。在 Server 类中，把连接请求队列的长度设为 3。这意味着当队列中有了 3 个连接请求时，如果 Client 再请求连接，就会被 Server 拒绝。下面按照以下步骤运行 Server 和 Client 程序。

（1）把 Server 类的 main()方法中的“server.service();”这行程序代码注释掉。这使得服务器与 8 000 端口绑定后，永远不会执行 serverSocket.accept()方法。这意味着队列中的连接请求永远不会被取出。先运行 Server 程序，然后再运行 Client 程序，Client 程序的打印结果如下：

```
第 1 次连接成功
第 2 次连接成功
第 3 次连接成功
Exception in thread "main" java.net.ConnectException: Connection refused: connect
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(Unknown Source)
        at java.net.PlainSocketImpl.connectToAddress(Unknown Source)
```

```
        at java.net.PlainSocketImpl.connect(Unknown Source)
        at java.net.SocksSocketImpl.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at Client.main(Client.java:10)
```

从以上打印结果可以看出，Client 与 Server 在成功地建立了 3 个连接后，就无法再创建其余的连接了，因为服务器的队列已经满了。

（2）把 Server 类的 main()方法按如下方式修改：

```
public static void main(String args[])throws Exception {
    Server server=new Server();
    //Thread.sleep(60000*10);   //睡眠 10 分钟
    server.service();
  }
```

作了以上修改，服务器与 8 000 端口绑定后，就会在一个 while 循环中不断执行 serverSocket.accept()方法，该方法从队列中取出连接请求，使得队列能及时腾出空位，以容纳新的连接请求。先运行 Server 程序，然后再运行 Client 程序，Client 程序的打印结果如下：

```
第 1 次连接成功
第 2 次连接成功
第 3 次连接成功
…
第 100 次连接成功
```

从以上打印结果可以看出，此时 Client 能顺利与 Server 建立 100 次连接。

3.1.3 设定绑定的 IP 地址

如果主机只有一个 IP 地址，那么默认情况下，服务器程序就与该 IP 地址绑定。ServerSocket 的第 4 个构造方法 ServerSocket(int port, int backlog, InetAddress bindAddr) 有一个 bindAddr 参数，它显式指定服务器要绑定的 IP 地址，该构造方法适用于具有多个 IP 地址的主机。假定一个主机有两个网卡，一个网卡用于连接到 Internet， IP 地址为 222.67.5.94，还有一个网卡用于连接到本地局域网，IP 地址为 192.168.3.4。如果服务器仅仅被本地局域网中的客户访问，那么可以按如下方式创建 ServerSocket：

```
ServerSocket serverSocket=new ServerSocket(8000,10,InetAddress.getByName ("192.168.3.4"));
```

3.1.4 默认构造方法的作用

ServerSocket 有一个不带参数的默认构造方法。通过该方法创建的 ServerSocket 不与任何端口绑定，接下来还需要通过 bind()方法与特定端口绑定。

这个默认构造方法的用途是，允许服务器在绑定到特定端口之前，先设置 ServerSocket 的一些选项。因为一旦服务器与特定端口绑定，有些选项就不能再改变了。

在以下代码中，先把 ServerSocket 的 SO_REUSEADDR 选项设为 true，然后再把

它与 8000 端口绑定：

```
ServerSocket serverSocket=new ServerSocket();
serverSocket.setReuseAddress(true);                    //设置 ServerSocket 的选项
serverSocket.bind(new InetSocketAddress(8000));        //与 8000 端口绑定
```

如果把以上程序代码改为：

```
ServerSocket serverSocket=new ServerSocket(8000);
serverSocket.setReuseAddress(true);                    //设置 ServerSocket 的选项
```

那么 serverSocket.setReuseAddress(true)方法就不起任何作用了，因为 SO_REUSEADDR 选项必须在服务器绑定端口之前设置才有效。

3.2 接收和关闭与客户的连接

ServerSocket 的 accept()方法从连接请求队列中取出一个客户的连接请求，然后创建与客户连接的 Socket 对象，并将它返回。如果队列中没有连接请求，accept()方法就会一直等待，直到接收到了连接请求才返回。

接下来，服务器从 Socket 对象中获得输入流和输出流，就能与客户交换数据。当服务器正在进行发送数据的操作时，如果客户端断开了连接，那么服务器端会抛出一个 IOException 的子类 SocketException 异常：

```
java.net.SocketException: Connection reset by peer
```

这只是服务器与单个客户通信中出现的异常，这种异常应该被捕获，使得服务器能继续与其他客户通信。

以下程序显示了单线程服务器采用的通信流程：

```
public void service() {
   while (true) {
      Socket socket=null;
      try {
         socket = serverSocket.accept();                    //从连接请求队列中取出一个连接
         System.out.println("New connection accepted " +
         socket.getInetAddress() + ":" +socket.getPort());
         //接收和发送数据
         …
      }catch (IOException e) {
         //这只是与单个客户通信时遇到的异常，可能是由于客户端过早断开连接引起的
         //这种异常不应该中断整个 while 循环
          e.printStackTrace();
      }finally {
          try{
             if(socket!=null)socket.close();               //与一个客户通信结束后，要关闭 Socket
          }catch (IOException e) {e.printStackTrace();}
      }
   }
}
```

与单个客户通信的代码放在一个 try 代码块中，如果遇到异常，该异常被 catch 代

码块捕获。try 代码块后面还有一个 finally 代码块，它保证不管与客户通信正常结束还是异常结束，最后都会关闭 Socket，断开与这个客户的连接。

3.3 关闭 ServerSocket

ServerSocket 的 close()方法使服务器释放占用的端口，并且断开与所有客户的连接。当一个服务器程序运行结束时，即使没有执行 ServerSocket 的 close()方法，操作系统也会释放这个服务器占用的端口。因此，服务器程序并不一定要在结束之前执行 ServerSocket 的 close()方法。

在某些情况下，如果希望及时释放服务器的端口，以便让其他程序能占用该端口，则可以显式调用 ServerSocket 的 close()方法。例如，以下代码用于扫描 1~65535 之间的端口号。如果 ServerSocket 成功创建，意味着该端口未被其他服务器进程绑定，否者说明该端口已经被其他进程占用：

```
for(int port=1;port<=65535;port++){
   try{
   ServerSocket serverSocket=new ServerSocket(port);
   serverSocket.close();          //及时关闭 ServerSocket
   }catch(IOException e){
      System.out.println("端口"+port+" 已经被其他服务器进程占用");
}
}
```

以上程序代码创建了一个 ServerSocket 对象后，就马上关闭它，以便及时释放它占用的端口，从而避免程序临时占用系统的大多数端口。

ServerSocket 的 isClosed()方法判断 ServerSocket 是否关闭，只有执行了 ServerSocket 的 close()方法，isClosed()方法才返回 true；否则，即使 ServerSocket 还没有和特定端口绑定，isClosed()方法也会返回 false。

ServerSocket 的 isBound()方法判断 ServerSocket 是否已经与一个端口绑定，只要 ServerSocket 已经与一个端口绑定，即使它已经被关闭，isBound()方法也会返回 true。

如果需要确定一个 ServerSocket 已经与特定端口绑定，并且还没有被关闭，则可以采用以下方式：

```
boolean isOpen=serverSocket.isBound() && !serverSocket.isClosed();
```

3.4 获取 ServerSocket 的信息

ServerSocket 的以下两个 get 方法可分别获得服务器绑定的 IP 地址，以及绑定的端口：

- public InetAddress getInetAddress()
- public int getLocalPort()

前面已经讲到，在构造 ServerSocket 时，如果把端口设为 0，那么将由操作系统为

服务器分配一个端口（称为匿名端口），程序只要调用 getLocalPort()方法就能获知这个端口号。如例程 3-3 所示的 RandomPort 创建了一个 ServerSocket，它使用的就是匿名端口。

例程 3-3　RandomPort.java

```
import java.io.*;
import java.net.*;

public class RandomPort{
   public static void main(String args[])throws IOException{
      ServerSocket serverSocket=new ServerSocket(0);
      System.out.println("监听的端口为:"+serverSocket.getLocalPort());
   }
}
```

多次运行 RandomPort 程序，可能会得到如下运行结果：

```
C:\chapter03\classes>java RandomPort
监听的端口为:3000
C:\chapter03\classes>java RandomPort
监听的端口为:3004
C:\chapter03\classes>java RandomPort
监听的端口为:3005
```

多数服务器会监听固定的端口，这样才便于客户程序访问服务器。匿名端口一般适用于服务器与客户之间的临时通信，通信结束，就断开连接，并且 ServerSocket 占用的临时端口也被释放。

FTP（文件传输）协议就使用了匿名端口。如图 3-1 所示，FTP 协议用于在本地文件系统与远程文件系统之间传送文件。

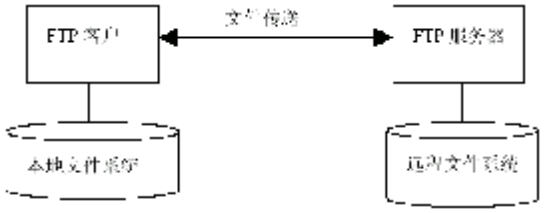

图 3-1　FTP 协议用于在本地文件系统与远程文件系统之间传送文件

FTP 使用两个并行的 TCP 连接：一个是控制连接，一个是数据连接。控制连接用于在客户和服务器之间发送控制信息，如用户名和口令、改变远程目录的命令或上传和下载文件的命令。数据连接用于传送文件。TCP 服务器在 21 端口上监听控制连接，如果有客户要求上传或下载文件，就另外建立一个数据连接，通过它来传送文件。数据连接的建立有两种方式。

（1）如图 3-2 所示，TCP 服务器在 20 端口上监听数据连接，TCP 客户主动请求建立与该端口的连接。

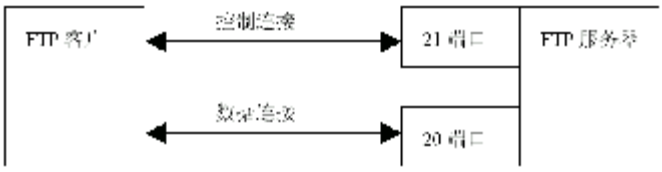

图 3-2　TCP 服务器在 20 端口上监听数据连接

（2）如图 3-3 所示，首先由 TCP 客户创建一个监听匿名端口的 ServerSocket，再把这个 ServerSocket 监听的端口号（调用 ServerSocket 的 getLocalPort()方法就能得到端口号）发送给 TCP 服务器，然后由 TCP 服务器主动请求建立与客户端的连接。

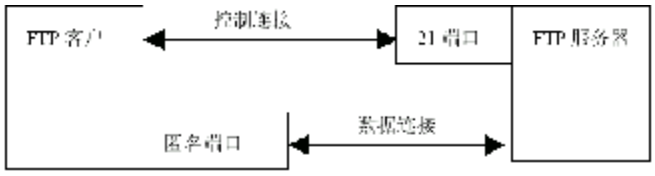

图 3-3　TCP 客户在匿名端口上监听数据连接

以上第二种方式就使用了匿名端口，并且是在客户端使用的，用于和服务器建立临时的数据连接。在实际应用中，在服务器端也可以使用匿名端口。

3.5　ServerSocket 选项

ServerSocket 有以下 3 个选项。

- SO_TIMEOUT：表示等待客户连接的超时时间。
- SO_REUSEADDR：表示是否允许重用服务器所绑定的地址。
- SO_RCVBUF：表示接收数据的缓冲区的大小。

3.5.1　SO_TIMEOUT 选项

- 设置该选项：public void setSoTimeout(int timeout) throws SocketException
- 读取该选项：public int getSoTimeout () throws IOException

SO_TIMEOUT 表示 ServerSocket 的 accept()方法等待客户连接的超时时间，以毫秒为单位。如果 SO_TIMEOUT 的值为 0，表示永远不会超时，这是 SO_TIMEOUT 的默认值。

当服务器执行 ServerSocket 的 accept()方法时，如果连接请求队列为空，服务器就会一直等待，直到接收到了客户连接才从 accept()方法返回。如果设定了超时时间，那么当服务器等待的时间超过了超时时间，就会抛出 SocketTimeoutException，它是 InterruptedException 的子类。

如例程 3-4 所示的 TimeoutTester 把超时时间设为 6 秒钟。

例程 3-4　TimeoutTester.java

```
import java.io.*;
import java.net.*;

public class TimeoutTester{
   public static void main(String args[])throws IOException{
      ServerSocket serverSocket=new ServerSocket(8000);
      serverSocket.setSoTimeout(6000); //等待客户连接的时间不超过 6 秒
      Socket socket=serverSocket.accept();
      socket.close();
      System.out.println("服务器关闭");
   }
}
```

运行以上程序，过 6 秒钟后，程序会从 serverSocket.accept()方法中抛出 SocketTimeoutException：

```
C:\chapter03\classes>java TimeoutTester
Exception in thread "main" java.net.SocketTimeoutException: Accept timed out
        at java.net.PlainSocketImpl.socketAccept(Native Method)
        at java.net.PlainSocketImpl.accept(Unknown Source)
        at java.net.ServerSocket.implAccept(Unknown Source)
        at java.net.ServerSocket.accept(Unknown Source)
        at TimeoutTester.main(TimeoutTester.java:8)
```

如果把程序中的“serverSocket.setSoTimeout(6000)”注释掉，那么 serverSocket.accept()方法永远不会超时，它会一直等待下去，直到接收到了客户的连接，才会从 accept()方法返回。

Tips

服务器执行 serverSocket.accept()方法时，等待客户连接的过程也称为阻塞。本书第 4 章的 4.1 节（线程阻塞的概念）详细介绍了阻塞的概念。

3.5.2　SO_REUSEADDR 选项

- 设置该选项：public void setResuseAddress(boolean on) throws SocketException
- 读取该选项：public boolean getResuseAddress() throws SocketException

这个选项与 Socket 的 SO_REUSEADDR 选项相同，用于决定如果网络上仍然有数据向旧的 ServerSocket 传输数据，是否允许新的 ServerSocket 绑定到与旧的 ServerSocket 同样的端口上。SO_REUSEADDR 选项的默认值与操作系统有关，在某些操作系统中，允许重用端口，而在某些操作系统中不允许重用端口。

当 ServerSocket 关闭时，如果网络上还有发送到这个 ServerSocket 的数据，这个 ServerSocket 不会立刻释放本地端口，而是会等待一段时间，确保接收到了网络上发送过来的延迟数据，然后再释放端口。

许多服务器程序都使用固定的端口。当服务器程序关闭后，有可能它的端口还会被占用一段时间，如果此时立刻在同一个主机上重启服务器程序，由于端口已经被占

用，使得服务器程序无法绑定到该端口，服务器启动失败，并抛出 BindException：

```
Exception in thread "main" java.net.BindException: Address already in use: JVM_Bind
```

为了确保一个进程关闭了 ServerSocket 后，即使操作系统还没释放端口，同一个主机上的其他进程还可以立刻重用该端口，可以调用 ServerSocket 的 setResuse-Address(true)方法：

```
if(!serverSocket.getResuseAddress())serverSocket.setResuseAddress(true);
```

值得注意的是，serverSocket.setResuseAddress(true)方法必须在 ServerSocket 还没有绑定到一个本地端口之前调用，否则执行 serverSocket.setResuseAddress(true)方法无效。此外，两个共用同一个端口的进程必须都调用 serverSocket.setResuseAddress(true)方法，才能使得一个进程关闭 ServerSocket 后，另一个进程的 ServerSocket 还能够立刻重用相同端口。

3.5.3 SO_RCVBUF 选项

- 设置该选项：public void setReceiveBufferSize(int size) throws SocketException
- 读取该选项：public int getReceiveBufferSize() throws SocketException

SO_RCVBUF 表示服务器端的用于接收数据的缓冲区的大小，以字节为单位。一般说来，传输大的连续的数据块（基于 HTTP 或 FTP 协议的数据传输）可以使用较大的缓冲区，这可以减少传输数据的次数，从而提高传输数据的效率。而对于交互式的通信（Telnet 和网络游戏），则应该采用小的缓冲区，确保能及时把小批量的数据发送给对方。

SO_RCVBUF 的默认值与操作系统有关。例如，在 Windows 2000 中运行以下代码时，显示 SO_RCVBUF 的默认值为 8192:

```
ServerSocket serverSocket=new ServerSocket(8000);
System.out.println(serverSocket.getReceiveBufferSize());          //打印 8192
```

无论在 ServerSocket 绑定到特定端口之前或之后，调用 setReceiveBufferSize()方法都有效。例外情况下是如果要设置大于 64K 的缓冲区，则必须在 ServerSocket 绑定到特定端口之前进行设置才有效。例如，以下代码把缓冲区设为 128K：

```
ServerSocket serverSocket=new ServerSocket();
int size=serverSocket.getReceiveBufferSize();
if(size<131072) serverSocket.setReceiveBufferSize(131072);          //把缓冲区的大小设为 128K
serverSocket.bind(new InetSocketAddress(8000));                     //与 8 000 端口绑定
```

执行 serverSocket.setReceiveBufferSize()方法，相当于对所有由 serverSocket.accept()方法返回的 Socket 设置接收数据的缓冲区的大小。

3.5.4 设定连接时间、延迟和带宽的相对重要性

- public void setPerformancePreferences(int connectionTime,int latency,int bandwidth)

该方法的作用与 Socket 的 setPerformancePreferences()方法的作用相同，用于设定连接时间、延迟和带宽的相对重要性，参见本书第 2 章的 2.5.10 节（设定连接时间、延迟和带宽的相对重要性）。

3.6 创建多线程的服务器

在本书第 1 章的 1.5.1 节的例程 1-2 的 EchoServer 中，其 service()方法负责接收客户连接，以及与客户通信。service()方法的处理流程如下：

```
while (true) {
   Socket socket=null;
   try {
      socket = serverSocket.accept();                    //接收客户连接
      //从 Socket 中获得输入流与输出流，与客户通信
      …

   }catch (IOException e) {
       e.printStackTrace();
   }finally {
       try{
           if(socket!=null)socket.close();               //断开连接
       }catch (IOException e) {e.printStackTrace();}
   }
}
```

EchoServer 接收到一个客户连接，就与客户进行通信，通信完毕后断开连接，然后再接收下一个客户连接。假如同时有多个客户请求连接，这些客户就必须排队等候 EchoServer 的响应。EchoServer 无法同时与多个客户通信。

许多实际应用要求服务器具有同时为多个客户提供服务的能力。HTTP 服务器就是最明显的例子。任何时刻，HTTP 服务器都可能接收到大量的客户请求，每个客户都希望能快速得到 HTTP 服务器的响应。如果长时间让客户等待，会使网站失去信誉，从而降低访问量。

可以用并发性能来衡量一个服务器同时响应多个客户的能力。一个具有好的并发性能的服务器，必须符合两个条件：

- 能同时接收并处理多个客户连接；
- 对于每个客户，都会迅速给予响应。

服务器同时处理的客户连接数目越多，并且对每个客户作出响应的速度越快，就表明并发性能越高。

用多个线程来同时为多个客户提供服务，这是提高服务器的并发性能的最常用的手段。本节将按照 3 种方式来重新实现 EchoServer，它们都使用了多线程。

- 为每个客户分配一个工作线程。
- 创建一个线程池，由其中的工作线程来为客户服务。
- 利用 JDK 的 Java 类库中现成的线程池，由它的工作线程来为客户服务。

3.6.1 为每个客户分配一个线程

服务器的主线程负责接收客户的连接，每次接收到一个客户连接，就会创建一个工作线程，由它负责与客户的通信。以下是 EchoServer 的 service()方法的代码：

```
    public void service() {
        while (true) {
            Socket socket=null;
            try {
                socket = serverSocket.accept();                    //接收客户连接
                Thread workThread=new Thread(new Handler(socket));  //创建一个工作线程
                workThread.start();                                //启动工作线程
            }catch (IOException e) {
                 e.printStackTrace();
            }
        }
    }
```

以上工作线程 workThread 执行 Handler 的 run()方法。Handler 类实现了 Runnable 接口，它的 run()方法负责与单个客户通信，与客户通信结束后，就会断开连接，执行 Handler 的 run()方法的工作线程也会自然终止。如例程 3-5 所示是 EchoServer 类及 Handler 类的源程序。

例程 3-5　EchoServer.java（为每个任务分配一个线程）

```
package multithread1;
import java.io.*;
import java.net.*;
public class EchoServer {
    private int port=8000;
    private ServerSocket serverSocket;

    public EchoServer() throws IOException {
        serverSocket = new ServerSocket(port);
        System.out.println("服务器启动");
    }

    public void service() {
        while (true) {
            Socket socket=null;
            try {
                socket = serverSocket.accept();                    //接收客户连接
                Thread workThread=new Thread(new Handler(socket));  //创建一个工作线程
                workThread.start();                                //启动工作线程
            }catch (IOException e) {
                 e.printStackTrace();
            }
        }
    }

    public static void main(String args[])throws IOException {
        new EchoServer().service();
    }
}

class Handler implements Runnable{                                 //负责与单个客户的通信
    private Socket socket;
    public Handler(Socket socket){
        this.socket=socket;
```

```
    }
    private PrintWriter getWriter(Socket socket)throws IOException{…}
    private BufferedReader getReader(Socket socket)throws IOException{…}
    public String echo(String msg) {…}
    public void run(){
      try {
        System.out.println("New connection accepted " +
        socket.getInetAddress() + ":" +socket.getPort());
        BufferedReader br =getReader(socket);
        PrintWriter pw = getWriter(socket);

      String msg = null;
        while ((msg = br.readLine()) != null) {                    //接收和发送数据，直到通信结束
          System.out.println(msg);
          pw.println(echo(msg));
          if (msg.equals("bye"))
            break;
        }
      }catch (IOException e) {
          e.printStackTrace();
      }finally {
          try{
            if(socket!=null)socket.close();                        //断开连接
          }catch (IOException e) {e.printStackTrace();}
      }
    }
  }
```

3.6.2 创建线程池

在 3.6.1 节介绍的实现方式中，对每个客户都分配一个新的工作线程。当工作线程与客户通信结束，这个线程就被销毁。这种实现方式有以下不足之处。

- 服务器创建和销毁工作线程的开销（包括所花费的时间和系统资源）很大。如果服务器需要与许多客户通信，并且与每个客户的通信时间都很短，那么有可能服务器为客户创建新线程的开销比实际与客户通信的开销还要大。
- 除了创建和销毁线程的开销之外，活动的线程也消耗系统资源。每个线程本身都会占用一定的内存（每个线程需要大约 1M 内存），如果同时有大量客户连接服务器，就必须创建大量工作线程，它们消耗了大量内存，可能会导致系统的内存空间不足。
- 如果线程数目固定，并且每个线程都有很长的生命周期，那么线程切换也是相对固定的。不同操作系统有不同的切换周期，一般在 20 毫秒左右。这里所说的线程切换是指在 Java 虚拟机，以及底层操作系统的调度下，线程之间转让 CPU 的使用权。如果频繁创建和销毁线程，那么将导致频繁地切换线程，因为一个线程被销毁后，必然要把 CPU 转让给另一个已经就绪的线程，使该线程获得运行机会。在这种情况下，线程之间的切换不再遵循系统的固定切换周期，切换线程的开销甚至比创建及销毁线程的开销还大。

线程池为线程生命周期开销问题和系统资源不足问题提供了解决方案。线程池中

预先创建了一些工作线程，它们不断从工作队列中取出任务，然后执行该任务。当工作线程执行完一个任务时，就会继续执行工作队列中的下一个任务。线程池具有以下优点：

- 减少了创建和销毁线程的次数，每个工作线程都可以一直被重用，能执行多个任务。
- 可以根据系统的承载能力，方便地调整线程池中线程的数目，防止因为消耗过量系统资源而导致系统崩溃。

如例程 3-6 所示，ThreadPool 类提供了线程池的一种实现方案。

例程 3-6　ThreadPool.java

```
package multithread2;
import java.util.LinkedList;
public class ThreadPool extends ThreadGroup {
   private boolean isClosed=false;                    //线程池是否关闭
   private LinkedList<Runnable> workQueue;            //表示工作队列
   private static int threadPoolID;                   //表示线程池 ID
   private int threadID;                              //表示工作线程 ID

   public ThreadPool(int poolSize) {                  //poolSize 指定线程池中的工作线程数目
      super("ThreadPool-" + (threadPoolID++));
      setDaemon(true);
      workQueue = new LinkedList<Runnable>();         //创建工作队列
      for (int i=0; i<poolSize; i++)
         new WorkThread().start();                    //创建并启动工作线程
   }

  /** 向工作队列中加入一个新任务，由工作线程去执行该任务 */
   public synchronized void execute(Runnable task) {
      if (isClosed) {                                 //线程池被关则抛出 IllegalStateException 异常
         throw new IllegalStateException();
      }
      if (task != null) {
         workQueue.add(task);
         notify();                                    //唤醒正在 getTask()方法中等待任务的工作线程
      }
   }

   /** 从工作队列中取出一个任务，工作线程会调用此方法 */
   protected synchronized Runnable getTask()throws InterruptedException{
      while (workQueue.size() == 0) {
         if (isClosed) return null;
         wait();                                      //如果工作队列中没有任务，就等待任务
      }
      return workQueue.removeFirst();
   }

   /** 关闭线程池 */
   public synchronized void close() {
      if (!isClosed) {
         isClosed = true;
         workQueue.clear();                           //清空工作队列
```

```
            interrupt();                                    //中断所有的工作线程，该方法继承自 ThreadGroup 类
        }
    }

    /** 等待工作线程把所有任务执行完 */
    public void join() {
        synchronized (this) {
            isClosed = true;
            notifyAll();                                    //唤醒还在 getTask()方法中等待任务的工作线程
        }

        Thread[] threads = new Thread[activeCount()];
        //enumerate()方法继承自 ThreadGroup 类，获得线程组中当前所有活着的工作线程
        int count = enumerate(threads);
        for (int i=0; i<count; i++) {                       //等待所有工作线程运行结束
            try {
                threads[i].join();                          //等待工作线程运行结束
            }catch(InterruptedException ex) { }
        }
    }

    /**  内部类：工作线程  */
    private class WorkThread extends Thread {
        public WorkThread() {
            //加入到当前 ThreadPool 线程组中
            super(ThreadPool.this,"WorkThread-" + (threadID++));
        }

        public void run() {
            while (!isInterrupted()) {   //isInterrupted()方法继承自 Thread 类，判断线程是否被中断
                Runnable task = null;
                try {                                       //取出任务
                    task = getTask();
                }catch (InterruptedException ex){}

                // 如果 getTask()返回 null 或者线程执行 getTask()时被中断，则结束此线程
                if (task == null) return;

                try { //运行任务，异常在 catch 代码块中捕获
                    task.run();
                } catch (Throwable t) {
                    t.printStackTrace();
                }
            }                                               //#while
        }                                                   //#run()
    }                                                       //#WorkThread 类
}
```

在 ThreadPool 类中定义了一个 LinkedList 类型的 workQueue 成员变量，它表示工作队列，用来存放线程池要执行的任务，每个任务都是 Runnable 实例。ThreadPool 类的客户程序（利用 ThreadPool 来执行任务的程序）只要调用 ThreadPool 类的 execute (Runnable task)方法，就能向线程池提交任务。在 ThreadPool 类的 execute()方法中，先判断线程池是否已经关闭。如果线程池已经关闭，就不再接收任务，否则就把任务加

入到工作队列中，并且唤醒正在等待任务的工作线程。

在 ThreadPool 类的构造方法中，会创建并启动若干工作线程，工作线程的数目由构造方法的参数 poolSize 决定。WorkThread 类表示工作线程，它是 ThreadPool 类的内部类。工作线程从工作队列中取出一个任务，接着执行该任务，然后再从工作队列中取出下一个任务并执行它，如此反复。

工作线程从工作队列中取任务的操作是由 ThreadPool 类的 getTask()方法实现的，它的处理逻辑如下：

- 如果队列为空并且线程池已关闭，那就返回 null，表示已经没有任务可以执行了；
- 如果队列为空并且线程池没有关闭，那就在此等待，直到其他线程将其唤醒或者中断；
- 如果队列中有任务，就取出第一个任务并将其返回。

线程池的 join()和 close()方法都可用来关闭线程池。join()方法确保在关闭线程池之前，工作线程把队列中的所有任务都执行完。而 close()方法则立即清空队列，并且中断所有的工作线程。

ThreadPool 类是 ThreadGroup 类的子类。ThreadGroup 类表示线程组，它提供了一些管理线程组中线程的方法。例如，interrupt()方法相当于调用线程组中所有活着的线程的 interrupt()方法。线程池中的所有工作线程都加入到当前 ThreadPool 对象表示的线程组中。ThreadPool 类在 close()方法中调用了 interrupt()方法：

```
    /** 关闭线程池 */
    public synchronized void close() {
        if (!isClosed) {
            isClosed = true;
            workQueue.clear();              //清空工作队列
            interrupt();                    //中断所有的工作线程，该方法继承自 ThreadGroup 类
        }
    }
```

以上 interrupt()方法用于中断所有的工作线程。interrupt()方法会对工作线程造成以下影响：

- 如果此时一个工作线程正在 ThreadPool 的 getTask()方法中因为执行 wait()方法而阻塞，则会抛出 InterruptedException；
- 如果此时一个工作线程正在执行一个任务，并且这个任务不会被阻塞，那么这个工作线程会正常执行完任务，但是在执行下一轮 while (!isInterrupted()){…}循环时，由于 isInterrupted()方法返回 true，因此退出 while 循环。

如例程 3-7 所示，ThreadPoolTester 类用于测试 ThreadPool 的用法。

例程 3-7　ThreadPoolTester.java

```
package multithread2;
public class ThreadPoolTester {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println(
```

```
            "用法: java ThreadPoolTest numTasks poolSize");
            System.out.println(
            "    numTasks - integer:  任务的数目");
            System.out.println(
            "    numThreads - integer:  线程池中的线程数目");
            return;
        }
        int numTasks = Integer.parseInt(args[0]);
        int poolSize = Integer.parseInt(args[1]);

        ThreadPool threadPool = new ThreadPool(poolSize);          //创建线程池

        // 运行任务
        for (int i=0; i<numTasks; i++)
            threadPool.execute(createTask(i));

        threadPool.join();                                         //等待工作线程完成所有的任务
        // threadPool.close();                                     //关闭线程池
    }//#main()

    /**   定义了一个简单的任务(打印 ID)    */
    private static Runnable createTask(final int taskID) {
        return new Runnable() {
            public void run() {
                System.out.println("Task " + taskID + ": start");
                try {
                    Thread.sleep(500);                             //增加执行一个任务的时间
                } catch (InterruptedException ex) { }
                System.out.println("Task " + taskID + ": end");
            }
        };
    }
}
```

ThreadPoolTester 类的 createTask()方法负责创建一个简单的任务。ThreadPoolTester 类的 main()方法读取用户从命令行输入的两个参数，它们分别表示任务的数目和工作线程的数目。main()方法接着创建线程池和任务，并且由线程池来执行这些任务，最后调用线程池的 join()方法，等待线程池把所有的任务执行完毕。

运行命令“java multithread2.ThreadPoolTester 5 3”，线程池将创建 3 个工作线程，由它们执行 5 个任务。程序的打印结果如下：

```
Task 0: start
Task 1: start
Task 2: start
Task 0: end
Task 3: start
Task 1: end
Task 4: start
Task 2: end
Task 3: end
Task 4: end
```

从打印结果看出，主线程等到工作线程执行完所有任务后，才结束程序。如果把

main()方法中的“threadPool.join()”改为“threadPool.close()”，再运行程序，则会看到，尽管有一些任务还没有执行，程序就运行结束了。

如例程 3-8 所示，EchoServer 利用线程池 ThreadPool 来完成与客户的通信任务。

例程 3-8　EchoServer.java（使用线程池 ThreadPool 类）

```
package multithread2;
import java.io.*;
import java.net.*;
public class EchoServer {
   private int port=8000;
   private ServerSocket serverSocket;
   private ThreadPool threadPool;                          //线程池
   private final int POOL_SIZE=4;                          //单个 CPU 时线程池中工作线程的数目

   public EchoServer() throws IOException {
      serverSocket = new ServerSocket(port);
      //创建线程池
      //Runtime 的 availableProcessors()方法返回当前系统的 CPU 的数目
      //系统的 CPU 越多，线程池中工作线程的数目也越多
      threadPool= new ThreadPool(
                Runtime.getRuntime().availableProcessors() * POOL_SIZE);

      System.out.println("服务器启动");
   }

   public void service() {
      while (true) {
         Socket socket=null;
         try {
            socket = serverSocket.accept();
            threadPool.execute(new Handler(socket));          //把与客户通信的任务交给线程池
         }catch (IOException e) {
             e.printStackTrace();
         }
      }
   }

   public static void main(String args[])throws IOException {
      new EchoServer().service();
   }
}

/** 负责与单个客户通信的任务，代码与 3.6.1 节的例程 3-5 的 Handler 类相同 */
class Handler implements Runnable{…}
```

在以上 EchoServer 的 service()方法中，每接收到一个客户连接，就向线程池 ThreadPool 提交一个与客户通信的任务。ThreadPool 把任务加入到工作队列中，工作线程会在适当的时候从队列中取出这个任务并执行它。

3.6.3　使用 JDK 类库提供的线程池

java.util.concurrent 包提供了现成的线程池的实现，它比 3.6.2 节介绍的线程池更加

健壮，而且功能也更强大。如图 3-4 所示是线程池的类框图。

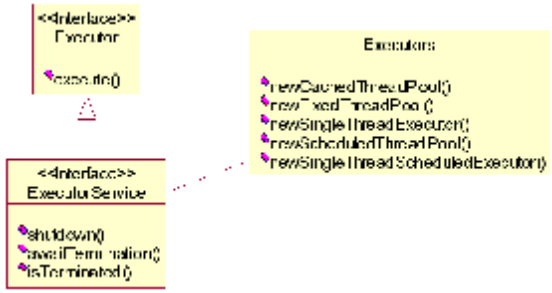

图 3-4　JDK 类库中的线程池的类框图

Executor 接口表示线程池，它的 execute(Runnable task)方法用来执行 Runnable 类型的任务。Executor 的子接口 ExecutorService 中声明了管理线程池的一些方法，比如用于关闭线程池的 shutdown()方法等。Executors 类中包含一些静态方法，它们负责生成各种类型的线程池 ExecutorService 实例，如表 3-1 所示。

表 3-1　Executors 类生成的 ExecutorService 实例的静态方法

Executors 类的静态方法	创建的 ExecutorService 线程池的类型
newCachedThreadPool()	在有任务时才创建新线程，空闲线程被保留 60 秒
newFixedThreadPool(int nThreads)	线程池中包含固定数目的线程，空闲线程会一直保留。参数 nThreads 设定线程池中线程的数目
newSingleThreadExecutor()	线程池中只有一个工作线程，它依次执行每个任务
newScheduledThreadPool(int corePoolSize)	线程池能按时间计划来执行任务，允许用户设定计划执行任务的时间。参数 corePoolSize 设定线程池中线程的最小数目。当任务较多时，线程池可能会创建更多的工作线程来执行任务
newSingleThreadScheduledExecutor()	线程池中只有一个工作线程，它能按时间计划来执行任务

如例程 3-9 所示，EchoServer 就利用上述线程池来负责与客户通信的任务。

例程 3-9　EchoServer.java（使用 java.util.concurrent 包中的线程池类）

```
package multithread3;
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

public class EchoServer {
   private int port=8000;
   private ServerSocket serverSocket;
   private ExecutorService executorService;                    //线程池
   private final int POOL_SIZE=4;                              //单个 CPU 时线程池中工作线程的数目

   public EchoServer() throws IOException {
      serverSocket = new ServerSocket(port);
      //创建线程池
      //Runtime 的 availableProcessors()方法返回当前系统的 CPU 的数目
      //系统的 CPU 越多，线程池中工作线程的数目也越多
```

```
            executorService= Executors.newFixedThreadPool(
             Runtime.getRuntime().availableProcessors() * POOL_SIZE);

            System.out.println("服务器启动");
        }

        public void service() {
            while (true) {
                Socket socket=null;
                try {
                    socket = serverSocket.accept();
                    executorService.execute(new Handler(socket));
                }catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        public static void main(String args[])throws IOException {
            new EchoServer().service();
        }
    }

    /** 负责与单个客户通信的任务，代码与 3.6.1 节的例程 3-5 的 Handler 类相同 */
    class Handler implements Runnable{…}
```

在 EchoServer 的构造方法中，调用 Executors.newFixedThreadPool()创建了具有固定工作线程数目的线程池。在 EchoServer 的 service()方法中，通过调用 executorService.execute()方法，把与客户通信的任务交给了 ExecutorService 线程池来执行。

3.6.4 使用线程池的注意事项

虽然线程池能大大提高服务器的并发性能，但使用它也会存在一定风险。与所有多线程应用程序一样，用线程池构建的应用程序容易产生各种并发问题，如对共享资源的竞争和死锁。此外，如果线程池本身的实现不健壮，或者没有合理地使用线程池，还容易导致与线程池有关的死锁、系统资源不足和线程泄漏等问题。

1．死锁

任何多线程应用程序都有死锁风险。造成死锁的最简单的情形是，线程 A 持有对象 X 的锁，并且在等待对象 Y 的锁，而线程 B 持有对象 Y 的锁，并且在等待对象 X 的锁。线程 A 与线程 B 都不释放自己持有的锁，并且等待对方的锁，这就导致两个线程永远等待下去，死锁就这样产生了。

虽然任何多线程程序都有死锁的风险，但线程池还会导致另外一种死锁。在这种情形下，假定线程池中的所有工作线程都在执行各自任务时被阻塞，它们都在等待某个任务 A 的执行结果。而任务 A 依然在工作队列中，由于没有空闲线程，使得任务 A 一直不能被执行。这使得线程池中的所有工作线程都永远阻塞下去，死锁就这样产生了。

2．系统资源不足

如果线程池中的线程数目非常多，这些线程会消耗包括内存和其他系统资源在内

的大量资源，从而严重影响系统性能。

3．并发错误

线程池的工作队列依靠 wait()和 notify()方法来使工作线程及时取得任务，但这两个方法都难于使用。如果编码不正确，可能会丢失通知，导致工作线程一直保持空闲状态，无视工作队列中需要处理的任务。因此使用这些方法时，必须格外小心，即便是专家也可能在这方面出错。最好使用现有的、比较成熟的线程池。例如，直接使用 java.util.concurrent 包中的线程池类。

4．线程泄漏

使用线程池的一个严重风险是线程泄漏。对于工作线程数目固定的线程池，如果工作线程在执行任务时抛出 RuntimeException 或 Error，并且这些异常或错误没有被捕获，那么这个工作线程就会异常终止，使得线程池永久失去了一个工作线程。如果所有的工作线程都异常终止，线程池就最终变为空，没有任何可用的工作线程来处理任务。

导致线程泄漏的另一种情形是，工作线程在执行一个任务时被阻塞，如等待用户的输入数据，但是由于用户一直不输入数据（可能是因为用户走开了），导致这个工作线程一直被阻塞。这样的工作线程名存实亡，它实际上不执行任何任务了。假如线程池中所有的工作线程都处于这样的阻塞状态，那么线程池就无法处理新加入的任务了。

5．任务过载

当工作队列中有大量排队等候执行的任务时，这些任务本身可能会消耗太多的系统资源而引起系统资源缺乏。

综上所述，线程池可能会带来种种风险，为了尽可能避免它们，使用线程池时需要遵循以下原则。

（1）如果任务 A 在执行过程中需要同步等待任务 B 的执行结果，那么任务 A 不适合加入到线程池的工作队列中。如果把像任务 A 一样的需要等待其他任务执行结果的任务加入到工作队列中，可能会导致线程池的死锁。

（2）如果执行某个任务时可能会阻塞，并且是长时间的阻塞，则应该设定超时时间，避免工作线程永久的阻塞下去而导致线程泄漏。在服务器程序中，当线程等待客户连接，或者等待客户发送的数据时，都可能会阻塞。可以通过以下方式设定超时时间：

- 调用 ServerSocket 的 setSoTimeout(int timeout)方法，设定等待客户连接的超时时间，参见本章 3.5.1 节（SO_TIMEOUT 选项）；
- 对于每个与客户连接的 Socket，调用该 Socket 的 setSoTimeout(int timeout)方法，设定等待客户发送数据的超时时间，参见本书第 2 章的 2.5.3 节（SO_TIMEOUT 选项）。

（3）了解任务的特点，分析任务是执行经常会阻塞的 I/O 操作，还是执行一直不会阻塞的运算操作。前者时断时续地占用 CPU，而后者对 CPU 具有更高的利用率。预计完成任务大概需要多长时间？是短时间任务还是长时间任务？

根据任务的特点，对任务进行分类，然后把不同类型的任务分别加入到不同线程池的工作队列中，这样可以根据任务的特点，分别调整每个线程池。

（4）调整线程池的大小。线程池的最佳大小主要取决于系统的可用 CPU 的数目，以及工作队列中任务的特点。假如在一个具有 N 个 CPU 的系统上只有一个工作队列，并且其中全部是运算性质（不会阻塞）的任务，那么当线程池具有 N 或 N+1 个工作线程时，一般会获得最大的 CPU 利用率。

如果工作队列中包含会执行 I/O 操作并常常阻塞的任务，则要让线程池的大小超过可用 CPU 的数目，因为并不是所有工作线程都一直在工作。选择一个典型的任务，然后估计在执行这个任务的过程中，等待时间（WT）与实际占用 CPU 进行运算的时间（ST）之间的比例 WT/ST。对于一个具有 N 个 CPU 的系统，需要设置大约 N×(1+WT/ST)个线程来保证 CPU 得到充分利用。

当然，CPU 利用率不是调整线程池大小过程中唯一要考虑的事项。随着线程池中工作线程数目的增长，还会碰到内存或者其他系统资源的限制，如套接字、打开的文件句柄或数据库连接数目等。要保证多线程消耗的系统资源在系统的承载范围之内。

（5）避免任务过载。服务器应根据系统的承载能力，限制客户并发连接的数目。当客户并发连接的数目超过了限制值，服务器可以拒绝连接请求，并友好地告知客户：服务器正忙，请稍后再试。

3.7 关闭服务器

前面介绍的 EchoServer 服务器都无法关闭自身，只有依靠操作系统来强行终止服务器程序。这种强行终止服务器程序的方式尽管简单方便，但是会导致服务器中正在执行的任务被突然中断。如果服务器处理的任务不是非常重要，允许随时中断，则可以依靠操作系统来强行终止服务器程序；如果服务器处理的任务非常重要，不允许被突然中断，则应该由服务器自身在恰当的时刻关闭自己。

本节介绍的 EchoServer 服务器就具有关闭自己的功能。它除了在 8000 端口监听普通客户程序 EchoClient 的连接外，还会在 8001 端口监听管理程序 AdminClient 的连接。当 EchoServer 服务器在 8001 端口接收到了 AdminClient 发送的“shutdown”命令时，EchoServer 就会开始关闭服务器，它不会再接收任何新的 EchoClient 进程的连接请求，对于那些已经接收但是还没有处理的客户连接，则会丢弃与该客户的通信任务，而不会把通信任务加入到线程池的工作队列中。另外，EchoServer 会等到线程池把当前工作队列中的所有任务执行完，才结束程序。

如例程 3-10 所示是 EchoServer 的源程序，其中关闭服务器的任务是由 shutdownThread 线程来负责的。

例程 3-10　EchoServer.java(具有关闭服务器的功能)

```
package multithread4;
import java.io.*;
import java.net.*;
```

```
import java.util.concurrent.*;

public class EchoServer {
   private int port=8000;
   private ServerSocket serverSocket;
   private ExecutorService executorService;                    //线程池
   private final int POOL_SIZE=4;                              //单个 CPU 时线程池中工作线程的数目

   private int portForShutdown=8001;                           //用于监听关闭服务器命令的端口
   private ServerSocket serverSocketForShutdown;
   private boolean isShutdown=false;                           //服务器是否已经关闭

   private Thread shutdownThread=new Thread(){                 //负责关闭服务器的线程
      public void start(){
         this.setDaemon(true);                                 //设置为守护线程（也称为后台线程）
         super.start();
      }
      public void run(){
         while (!isShutdown) {
            Socket socketForShutdown=null;
            try {
               socketForShutdown= serverSocketForShutdown.accept();
               BufferedReader br = new BufferedReader(
                                    new InputStreamReader(socketForShutdown.getInputStream()));
               String command=br.readLine();
             if(command.equals("shutdown")){
                  long beginTime=System.currentTimeMillis();
                  socketForShutdown.getOutputStream().write("服务器正在关闭\r\n".getBytes());
                  isShutdown=true;
                  //请求关闭线程池
                  //线程池不再接收新的任务，但是会继续执行完工作队列中现有的任务
                  executorService.shutdown();

                  //等待关闭线程池，每次等待的超时时间为 30 秒
                  while(!executorService.isTerminated())
                     executorService.awaitTermination(30,TimeUnit.SECONDS);

                  serverSocket.close(); //关闭与 EchoClient 客户通信的 ServerSocket
                  long endTime=System.currentTimeMillis();
                  socketForShutdown.getOutputStream().write(("服务器已经关闭，"+
                        "关闭服务器用了"+(endTime-beginTime)+"毫秒\r\n").getBytes());
                  socketForShutdown.close();
                  serverSocketForShutdown.close();

               }else{
                  socketForShutdown.getOutputStream().write("错误的命令\r\n".getBytes());
                  socketForShutdown.close();
               }
            }catch (Exception e) {
                e.printStackTrace();
            }
         }
      }
   };
```

```
    public EchoServer() throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(60000);                    //设定等待客户连接的超过时间为 60 秒
        serverSocketForShutdown = new ServerSocket(portForShutdown);

        //创建线程池
        executorService= Executors.newFixedThreadPool(
         Runtime.getRuntime().availableProcessors() * POOL_SIZE);

        shutdownThread.start();                              //启动负责关闭服务器的线程
        System.out.println("服务器启动");
    }

    public void service() {
        while (!isShutdown) {
            Socket socket=null;
            try {
                socket = serverSocket.accept();
                //可能会抛出 SocketTimeoutException 和 SocketException
                socket.setSoTimeout(60000);                  //把等待客户发送数据的超时时间设为 60 秒
                executorService.execute(new Handler(socket));
                //可能会抛出 RejectedExecutionException
            }catch(SocketTimeoutException e){
                 //不必处理等待客户连接时出现的超时异常
            }catch(RejectedExecutionException e){
                 try{
                    if(socket!=null)socket.close();
                 }catch(IOException x){}
                 return;
            }catch(SocketException e) {
                 //如果是由于在执行 serverSocket.accept()方法时，
                 //ServerSocket 被 ShutdownThread 线程关闭而导致的异常，就退出 service()方法
                 if(e.getMessage().indexOf("socket closed")!=-1)return;
              }catch(IOException e) {
                 e.printStackTrace();
            }
        }
    }

    public static void main(String args[])throws IOException {
        new EchoServer().service();
    }
}
/** 负责与单个客户通信的任务，代码与 3.6.1 节的例程 3-5 的 Handler 类相同 */
class Handler implements Runnable{…}
```

shutdownThread 线程负责关闭服务器。它一直监听 8001 端口，如果接收到了 AdminClient 发送的“shutdown”命令，就把 isShutdown 变量设为 true。shutdownThread 线程接着执行 executorService.shutdown()方法，该方法请求关闭线程池，线程池将不再接收新的任务，但是会继续执行完工作队列中现有的任务。shutdownThread 线程接着等待线程池关闭：

```
    while(!executorService.isTerminated())
        executorService.awaitTermination(30,TimeUnit.SECONDS);   //等待 30 秒
```

当线程池的工作队列中的所有任务执行完毕，executorService.isTerminated()方法就会返回 true。

shutdownThread 线程接着关闭监听 8000 端口的 ServerSocket，最后再关闭监听 8001 端口的 ServerSocket。

shutdownThread 线程在执行上述代码时，主线程正在执行 EchoServer 的 service()方法。shutdownThread 线程一系列操作会对主线程造成以下影响。

- 如果 shutdownThread 线程已经把 isShutdown 变量设为 true，而主线程正准备执行 service()方法的下一轮 while(!isShutdown){…}循环时，由于 isShutdown 变量为 true，就会退出循环。
- 如果 shutdownThread 线程已经执行了监听 8 000 端口的 ServerSocket 的 close()方法，而主线程正在执行该 ServerSocket 的 accept()方法，那么该方法会抛出 SocketException。EchoServer 的 service()方法捕获了该异常，在异常处理代码块中退出 service()方法。
- 如果 shutdownThread 线程已经执行了 executorService.shutdown()方法，而主线程正在执行 executorService.execute(…)方法，那么该方法会抛出 RejectedExecutionException。EchoServer 的 service()方法捕获了该异常，在异常处理代码块中退出 service()方法。
- 如果 shutdownThread 线程已经把 isShutdown 变量设为 true，但还没有调用监听 8 000 端口的 ServerSocket 的 close()方法，而主线程正在执行 ServerSocket 的 accept()方法，主线程阻塞 60 秒后会抛出 SocketTimeoutException。在准备执行 service()方法的下一轮 while(!isShutdown){…}循环时，由于 isShutdown 变量为 true，就会退出循环。
- 由此可见，当 shutdownThread 线程开始执行关闭服务器的操作时，主线程尽管不会立即终止，但是迟早会结束运行。

如例程 3-11 所示是 AdminClient 的源程序，它负责向 EchoServer 发送“shutdown”命令，从而关闭 EchoServer。

例程 3-11　AdminClient.java

```
package multithread4;
import java.net.*;
import java.io.*;
public class AdminClient{
   public static void main(String args[]){
      Socket socket=null;
      try{
         socket=new Socket("localhost",8001);
         //发送关闭命令
         OutputStream socketOut=socket.getOutputStream();
         socketOut.write("shutdown\r\n".getBytes());

         //接收服务器的反馈
         BufferedReader br = new BufferedReader(
                                        new InputStreamReader(socket.getInputStream()));
         String msg=null;
```

```
                while((msg=br.readLine())!=null)
                    System.out.println(msg);
                }catch(IOException e){
                    e.printStackTrace();
                }finally{
                    try{
                        if(socket!=null)socket.close();
                    }catch(IOException e){e.printStackTrace();}
                }
            }
        }
```

下面按照以下方式运行 EchoServer、EchoClient 和 AdminClient，以观察 EchoServer 服务器的关闭过程。EchoClient 类的源程序参见本书第 1 章的 1.5.2 节的例程 1-3。

（1）先运行 EchoServer，然后运行 AdminClient。EchoServer 与 AdminClient 进程都结束运行，并且在 AdminClient 的控制台打印如下结果：

```
服务器正在关闭
服务器已经关闭，关闭服务器用了 60 毫秒
```

（2）先运行 EchoServer，再运行 EchoClient，然后再运行 AdminClient。EchoServer 程序不会立即结束，因为它与 EchoClient 的通信任务还没有结束。在 EchoClient 的控制台中输入“bye”， EchoServer、EchoClient 和 AdminClient 进程都会结束运行。

（3）先运行 EchoServer，再运行 EchoClient，然后再运行 AdminClient。EchoServer 程序不会立即结束，因为它与 EchoClient 的通信任务还没有结束。不要在 EchoClient 的控制台中输入任何字符串，过 60 秒后，EchoServer 等待 EchoClient 的发送数据超时，结束与 EchoClient 的通信任务，EchoServer 和 AdminClient 进程结束运行。如果在 EchoClient 的控制台再输入字符串，则会抛出“连接已断开”的 SocketException。

3.8 小结

在 EchoServer 的构造方法中可以设定 3 个参数。

- 参数 port：指定服务器要绑定的端口。
- 参数 backlog：指定客户连接请求队列的长度。
- 参数 bindAddr：指定服务器要绑定的 IP 地址。

ServerSocket 的 accept()方法从连接请求队列中取出一个客户的连接请求，然后创建与客户连接的 Socket 对象，并将它返回。如果队列中没有连接请求，accept()方法就会一直等待，直到接收到了连接请求才返回。SO_TIMEOUT 选项表示 ServerSocket 的 accept()方法等待客户连接请求的超时时间，以毫秒为单位。如果 SO_TIMEOUT 的值为 0，表示永远不会超时，这是 SO_TIMEOUT 的默认值。可以通过 ServerSocket 的 setSoTimeout()方法来设置等待连接请求的超时时间。如果设定了超时时间，那么当服务器等待的时间超过了超时时间后，就会抛出 SocketTimeoutException，它是 InterruptedException 的子类。

许多实际应用要求服务器具有同时为多个客户提供服务的能力。用多个线程来同

时为多个客户提供服务，这是提高服务器的并发性能的最常用的手段。本章采用 3 种方式来重新实现 EchoServer，它们都使用了多线程：

（1）为每个客户分配一个工作线程；

（2）创建一个线程池，由其中的工作线程来为客户服务；

（3）利用 java.util.concurrent 包中现成的线程池，由它的工作线程来为客户服务。

第一种方式需要频繁地创建和销毁线程，如果线程执行的任务本身很简短，那么有可能服务器在创建和销毁线程方面的开销比在实际执行任务上的开销还要大。线程池能很好地避免这一问题。线程池先创建了若干工作线程，每个工作线程执行完一个任务后就会继续执行下一个任务，线程池减少了创建和销毁线程的次数，从而提高了服务器的运行性能。

3.9 练习题

1．关于 ServerSocket 构造方法的 backlog 参数，以下哪些说法正确？（多选）

A．backlog 参数用来显式设置操作系统中的连接请求队列的长度

B．如果没有设置 backlog 参数，那么连接请求队列的长度由操作系统决定

C．当一个客户的连接请求被加入到服务器端的连接请求队列中时，就意味着客户端建立了与服务器的连接

D．如果 backlog 参数的值大于操作系统限定的队列的最大长度，那么 backlog 参数无效

E．连接请求队列直接由 ServerSocket 创建并管理

F．ServerSocket 的 accept()方法从连接请求队列中取出连接请求

2．对于以下程序代码：

```
ServerSocket serverSocket=new ServerSocket(8000);
serverSocket.setReuseAddress(true);
```

以下哪个说法正确？（单选）

A．以上代码运行时出错

B．以上代码编译时出错

C．以上代码尽管编译和运行时都不会出错，但对 SO_REUSEADDR 选项的设置无效

D．以上说法都不正确

3．如何判断一个 ServerSocket 已经与特定端口绑定，并且还没有被关闭？（单选）

A．boolean isOpen=serverSocket.isBound();

B．boolean isOpen=serverSocket.isBound() && !serverSocket.isClosed();

C．boolean isOpen=serverSocket.isBound() && serverSocket.isConnected();

D．boolean isOpen=!serverSocket.isClosed();

4．ServerSocket 与 Socket 都有一个 SO_TIMEOUT 选项，它们的作用是否相同？（单选）

A．相同　　B．不同

5. 服务器端对每个客户都分配一个新的工作线程。当工作线程与客户通信结束时，这个线程就被销毁。这种实现方式有哪些不足？

6．服务器端采用线程池来保证并发响应多个客户的请求，线程池有哪些优缺点？

7．用 Java 实现一个线程池，线程池在初始状态下没有任何工作线程。当工作队列中有未执行的任务时，分以下两种情况处理：

（1）如果线程池中工作线程数目为 MAX_SIZE，那就什么也不做；

（2）如果线程池中工作线程数目小于 MAX_SIZE，那么创建一个工作线程，使它执行新任务。

对于线程池中空闲的工作线程，如果其闲置时间超过 TIMEOUT 秒，就销毁该线程。

8．用 Java 实现一个采用用户自定义协议的文件传输服务器 FileServer 和客户 FileClient。当 FileClient 发送请求“GET xxx.xxx”时，FileServer 就把 xxx.xxx 文件发送给 FileClient，FileClient 把该文件保存到客户端的本地文件系统中；当 FileClient 发送请求“PUT xxx.xxx”时，FileServer 就做好接收 xxx.xxx 文件的准备，FileClient 接着发送 xxx.xxx 文件的内容，FileServer 把接收到的文件内容保存到服务器端的本地文件系统中。GET 或 PUT 命令中的文件允许采用相对路径，其根路径由用户自定义的 FILE_PATH 系统属性指定。

答案：1．ABDF　　2．C　　3．B　　4．B