网络编程懒人入门(八)：手把手教你写基于TCP的Socket长连接-网络编程/专项技术区 - 即时通讯开发者社区!

本文原作者：“水晶虾饺”，原文由“玉刚说”写作平台提供写作赞助，原文版权归“玉刚说”微信公众号所有，即时通讯网收录时有改动。

1、引言

好多小白初次接触即时通讯（比如：IM或者消息推送应用）时，总是不能理解Web短连接（就是最常见的HTTP通信了）跟长连接（主要指TCP、UDP协议实现的socket通信，当然HTML5里的Websocket协议也是长连接）的区别，导致写即时通讯这类系统代码时往往找不到最佳实践，搞的一脸蒙逼。

本篇我们先简单了解一下 TCP/IP，然后通过实现一个

echo 服务器来学习 Java 的 Socket API。最后我们聊聊偏高级一点点的 socket 长连接和协议设计。

另外，本系列文章的前2篇《网络编程懒人入门(一)：快速理解网络通信协议（上篇）》、《网络编程懒人入门(二)：快速理解网络通信协议（下篇）》快速介绍了网络基本通信协议及理论基础，如果您对网络基础毫无概念，则请务必首先阅读完这2篇文章。本系列的第3篇文章《网络编程懒人入门(三)：快速理解TCP协议一篇就够》有助于您快速理解TCP协议理论的方方面面，建议也可以读一读。

TCP 是互联网的核心协议之一，鉴于它的重要性，希望通过阅读上面介绍的几篇理论文章，再针对本文的动手实践，能真正加深您对TCP协议的理解。

如果您正打算系统地学习即时通讯开发，在读完本文后，建议您可以详细阅读《新手入门一篇就够：从零开发移动端IM》。

（**提示**：本文完整源码可以从**文末附件**打包下载）

2、系列文章

本文是系列文章中的第8篇，本系列文章的大纲如下：

- 《网络编程懒人入门(一)：快速理解网络通信协议（上篇）》

- 《网络编程懒人入门(二)：快速理解网络通信协议（下篇）》
- 《网络编程懒人入门(三)：快速理解TCP协议一篇就够》
- 《网络编程懒人入门(四)：快速理解TCP和UDP的差异》
- 《网络编程懒人入门(五)：快速理解为什么说UDP有时比TCP更有优势》
- 《网络编程懒人入门(六)：史上最通俗的集线器、交换机、路由器功能原理入门》
- 《网络编程懒人入门(七)：深入浅出，全面理解HTTP协议》
- 《网络编程懒人入门(八)：手把手教你写基于TCP的Socket长连接》（本文）
- 《网络编程懒人入门(九)：通俗讲解，有了IP地址，为何还要用MAC地址？》

本站的《脑残式网络编程入门》也适合入门学习，本系列大纲如下：

- 《脑残式网络编程入门(一)：跟着动画来学TCP三次握手和四次挥手》
- 《脑残式网络编程入门(二)：我们在读写Socket时，究竟在读写什么？》
- 《脑残式网络编程入门(三)：HTTP协议必知必会的一些知识》

如果您觉得本系列文章过于基础，您可直接阅读《不为人知的网络编程》系列文章，该系列目录如下：

- 《不为人知的网络编程(一)：浅析TCP协议中的疑难杂症(上篇)》
- 《不为人知的网络编程(二)：浅析TCP协议中的疑难杂症(下篇)》
- 《不为人知的网络编程(三)：关闭TCP连接时为什么会TIME_WAIT、CLOSE_WAIT》
- 《不为人知的网络编程(四)：深入研究分析TCP的异常关闭》
- 《不为人知的网络编程(五)：UDP的连接性和负载均衡》
- 《不为人知的网络编程(六)：深入地理解UDP协议并用好它》
- 《不为人知的网络编程(七)：如何让不可靠的UDP变的可靠？》
- 《不为人知的网络编程(八)：从数据传输层深度解密HTTP》
- 《不为人知的网络编程(九)：理论联系实际，全方位深入理解DNS》

如果您对服务端高性能网络编程感兴趣，可以阅读以下系列文章：

- 《高性能网络编程(一)：单台服务器并发TCP连接数到底可以有多少》

- 《高性能网络编程(二)：上一个10年，著名的C10K并发连接问题》
- 《高性能网络编程(三)：下一个10年，是时候考虑C10M并发问题了》
- 《高性能网络编程(四)：从C10K到C10M高性能网络应用的理论探索》

关于移动端网络特性及优化手段的总结性文章请见：

- 《现代移动端网络短连接的优化手段总结：请求速度、弱网适应、安全保障》
- 《移动端IM开发者必读(一)：通俗易懂，理解移动网络的“弱”和“慢”》
- 《移动端IM开发者必读(二)：史上最全移动弱网络优化方法总结》

3、参考资料

《TCP/IP详解 - 第11章·UDP：用户数据报协议》
《TCP/IP详解 - 第17章·TCP：传输控制协议》
《TCP/IP详解 - 第18章·TCP连接的建立与终止》
《TCP/IP详解 - 第21章·TCP的超时与重传》
《通俗易懂-深入理解TCP协议（上）：理论基础》
《通俗易懂-深入理解TCP协议（下）：RTT、滑动窗口、拥塞处理》

《理论经典：TCP协议的3次握手与4次挥手过程详解》
《理论联系实际：Wireshark抓包分析TCP 3次握手、4次挥手过程》
《计算机网络通讯协议关系图（中文珍藏版）》
《高性能网络编程(一)：单台服务器并发TCP连接数到底可以有多少》
《高性能网络编程(二)：上一个10年，著名的C10K并发连接问题》
《高性能网络编程(三)：下一个10年，是时候考虑C10M并发问题了》
《高性能网络编程(四)：从C10K到C10M高性能网络应用的理论探索》
《简述传输层协议TCP和UDP的区别》
《为什么QQ用的是UDP协议而不是TCP协议？》
《移动端即时通讯协议选择：UDP还是TCP？》

4、TCP/IP 协议简介

TCP/IP协议族是互联网最重要的基础设施之一，如有兴趣了解TCP/IP的贡献，可以读一读此文：《技术往事：改变世界的TCP/IP协议（珍贵多图、手机慎点）》，本文因篇幅原因仅作简要介绍。

4.1IP协议

首先我们看 IP（Internet Protocol）协议。IP 协议提供了主机和主机间的通信。

为了完成不同主机的通信，我们需要某种方式来唯一标识一台主机，这个标识，就是著名的IP地址。通过IP地址，IP 协议就能够帮我们把一个数据包发送给对方。

4.2TCP协议

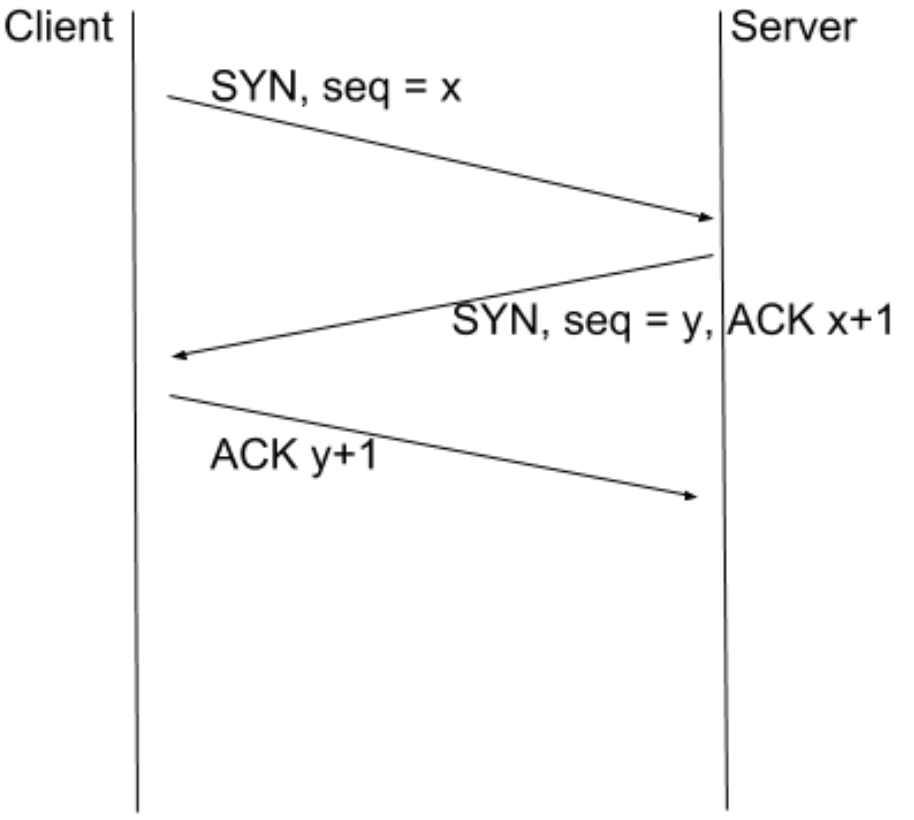
前面我们说过，IP 协议提供了主机和主机间的通信。TCP 协议在 IP 协议提供的主机间通信功能的基础上，完成这两个主机上进程对进程的通信。

有了 IP，不同主机就能够交换数据。但是，计算机收到数据后，并不知道这个数据属于哪个进程（简单讲，进程就是一个正在运行的应用程序）。TCP 的作用就在于，让我们能够知道这个数据属于哪个进程，从而完成进程间的通信。

为了标识数据属于哪个进程，我们给需要进行 TCP 通信的进程分配一个唯一的数字来标识它。这个数字，就是我们常说的端口号。

TCP 的全称是 Transmission Control Protocol，大家对它说得最多的，大概就是面向连接的特性了。之所以说它是有连接的，是说在进行通信前，通信双方需要先经过一个三次握手的过程。三次握手完成后，连接便建立了。这时候我们才可以开始发送/接收数据。（与之相对的是 UDP，不需要经过握手，就可以直接发送数据）。

下面我们简单了解一下三次握手的过程：



- 首先，客户向服务端发送一个 SYN，假设此时 sequence number 为 x。这个 x 是由操作系统根据一定的规则生成的，不妨认为它是一个随机数；
- 服务端收到 SYN 后，会向客户端再发送一个 SYN，此时服务器的 seq number = y。与此同时，会 ACK x+1，告诉客户端“已经收到了 SYN，可以发送数据了”；
- 客户端收到服务器的 SYN 后，回复一个 ACK y+1，这个 ACK 则是告诉服务器，SYN 已经收到，服务器

可以发送数据了。

经过这 3 步，TCP 连接就建立了，这里需要注意的有三点：

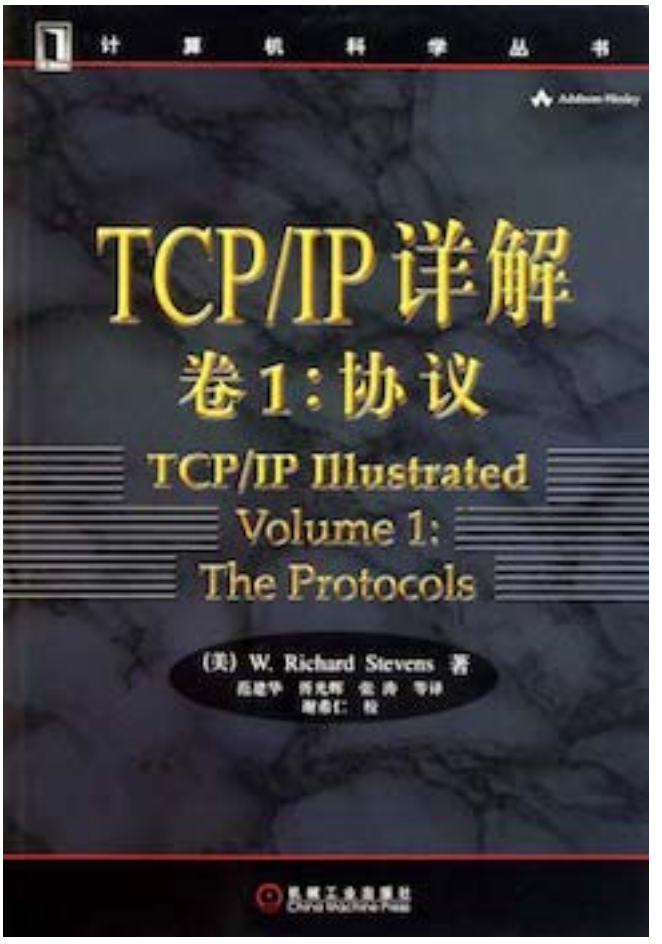
- 连接是由客户端主动发起的；
- 在第 3 步客户端向服务器回复 ACK 的时候，TCP 协议是允许我们携带数据的。之所以做不到，是 API 的限制导致的；
- TCP 协议还允许“四次握手”的发生，同样的，由于 API 的限制，这个极端的情况并不会发生。

TCP/IP 相关的理论知识我们就先了解到这里，如果对TCP的3次握手和4次挥手还不太理解，那就详细读读以下文章：

- 《通俗易懂-深入理解TCP协议（上）：理论基础》
- 《通俗易懂-深入理解TCP协议（下）：RTT、滑动窗口、拥塞处理》
- 《理论经典：TCP协议的3次握手与4次挥手过程详解》
- 《理论联系实际：Wireshark抓包分析TCP 3次握手、4次挥手过程》

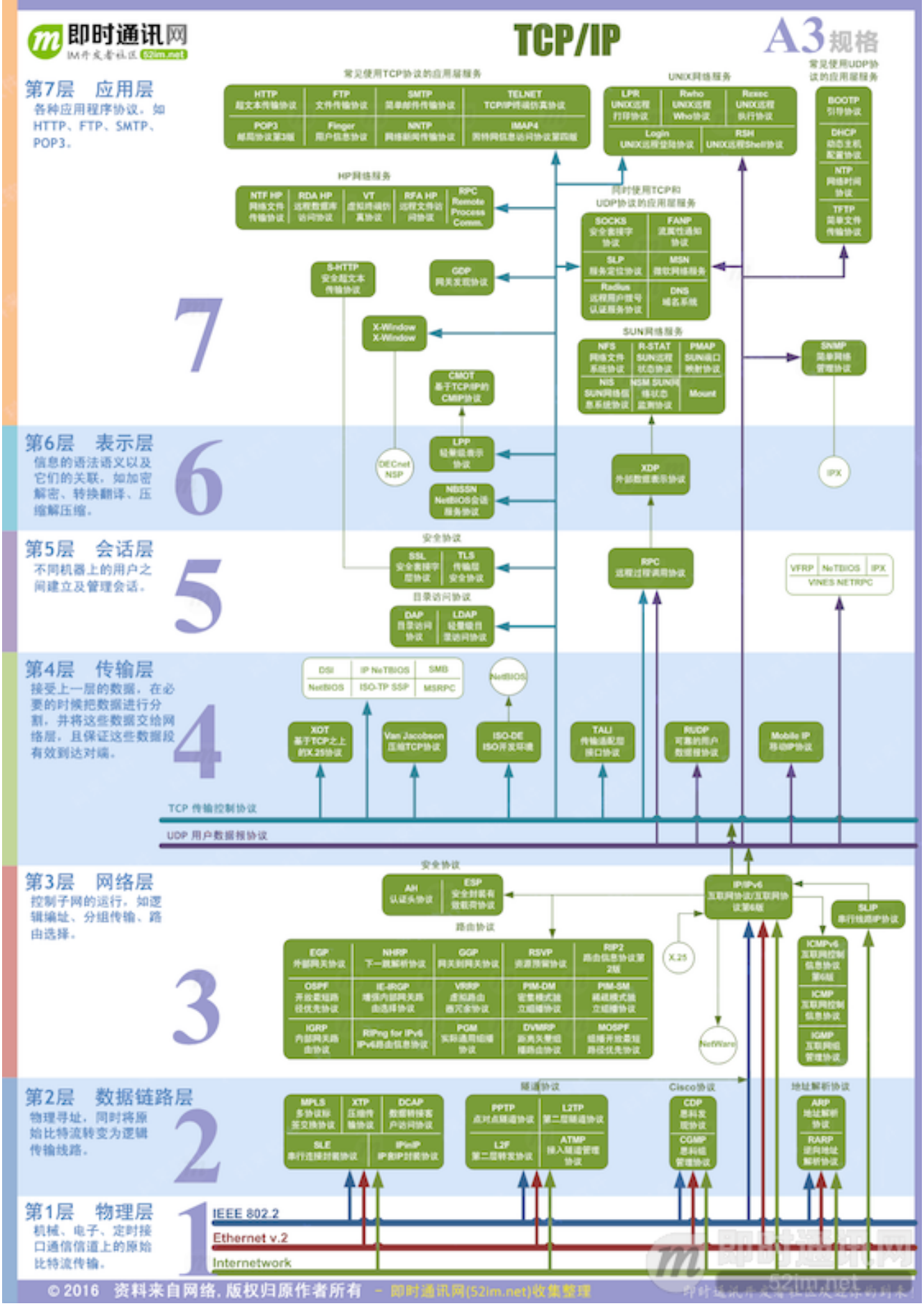
关于 TCP，还有诸如可靠性、流量控制、拥塞控制等非常有趣的特性。强烈推荐读者看一看 Richard 的名著

《TCP/IP 详解 - 卷1》（注意，是第1版，不是第2版）。



▲ 网络编程理论经典《TCP/IP 详解 - 卷1》（在线阅读版点此进入）

另外，TCP/IP协议其实是一个庞大的协议族，《计算机网络通讯协议关系图（中文珍藏版）》一文中为您清晰展现了这个协议族之间的关系，很有收藏价值，建议务必读一读。



▲ TCP/IP协议族图（高清原图点此进入）

下面我们看一些偏实战的东西。

5、Socket 基本用法

Socket 是 TCP 层的封装，通过 socket，我们就能进行 TCP 通信。

在 Java 的 SDK 中，socket 的共有两个接口：用于监听客户连接的 ServerSocket 和用于通信的 Socket。

使用 socket 的步骤如下：

- 1）创建 ServerSocket 并监听客户连接；
- 2）使用 Socket 连接服务端；
- 3）通过 Socket.getInputStream()/getOutputStream() 获取输入输出流进行通信。

下面，我们通过实现一个简单的 echo 服务来学习 socket 的使用。所谓的 echo 服务，就是客户端向服务端写入任意数据，服务器都将数据原封不动地写回给客户端。

5.1第一步：创建 ServerSocket 并监听客户连接

```
01

02
```

```
03
04
05  public class EchoServer {
06      private final ServerSocket mServerSocket;
07      public EchoServer(int port) throws IOException {
08
09          mServerSocket = new ServerSocket(port);
10      }
11      public void run() throws IOException {
12
13          Socket client = mServerSocket.accept();
14          handleClient(client);
15      }
16      private void handleClient(Socket socket) {
17
18      }
19      public static void main(String[] argv) {
20          try {
21              EchoServer server = new
    EchoServer(9877);
22              server.run();
23          } catch (IOException e) {
24              e.printStackTrace();
          }
```

```
25      }
26  }
27
28
29
```

5.2第二步：使用 Socket 连接服务端

```
01
02
03
04  public class EchoClient {
05
06      private final Socket mSocket;
07
08      public EchoClient(String host, int port) throws
    IOException {
09
10          mSocket = new Socket(host, port);
11
12      }
13
14      public void run() {
15
16      }
17
18      public static void main(String[] argv) {
19
20          try {
21
22              EchoClient client = new EchoClient("127.0.0.1",
    8000);
23              client.run();
24          } catch (IOException e) {
25              e.printStackTrace();
26          }
27      }
28  }
```

```
14
15              EchoClient client = new
EchoClient("localhost", 9877);
16              client.run();
17          } catch (IOException e) {
18              e.printStackTrace();
19          }
20      }
21  }
22
23
24
```

5.3第三步：通过socket.getInputStream()/getOutputStream()获取输入/输出流进行通信

首先，我们来实现服务端：

```
01
02  public class EchoServer {
03
04      private void handleClient(Socket socket) throws
IOException {
05          InputStream in = socket.getInputStream();
          OutputStream out =
```

```
06 socket.getOutputStream();
07         byte[] buffer = new byte[1024];
08         int n;
09         while ((n = in.read(buffer)) > 0) {
10             out.write(buffer, 0, n);
11         }
12     }
13 }
```

可以看到，服务端的实现其实很简单，我们不停地读取输入数据，然后写回给客户端。

下面我们看看客户端：

```
01
02
03 public class EchoClient {
04
05     public void run() throws IOException {
06         Thread readerThread = new
Thread(this::readResponse);
07         readerThread.start();
08         OutputStream out =
mSocket.getOutputStream();
09         byte[] buffer = new byte[1024];
10         int n;
```

```
11          while ((n = System.in.read(buffer)) > 0) {
12              out.write(buffer, 0, n);
13          }
14      }
15
16      private void readResponse() {
17          try {
18              InputStream in =
mSocket.getInputStream();
19              byte[] buffer = new byte[1024];
20              int n;
21              while ((n = in.read(buffer)) > 0) {
22                  System.out.write(buffer, 0, n);
23              }
24          } catch (IOException e) {
25              e.printStackTrace();
26          }
27      }
28  }
```

客户端会稍微复杂一点点，在读取用户输入的同时，我们又想读取服务器的响应。所以，这里创建了一个线程来读服务器的响应。

不熟悉 lambda 的读者，可以把Thread readerThread = new Thread(this::readResponse) 换成下面这个代码：

```
Thread readerThread = new Thread(new Runnable() {
    @Override
    public void run() {
        readResponse();
    }
});
```

打开两个 terminal 分别执行如下命令：

```

$ javac EchoServer.java
$ java EchoServer
$ javac EchoClient.java
$ java EchoClient
hello Server
hello Server
foo
foo

```

在客户端，我们会看到，输入的所有字符都打印了出来。

5.4最后需要注意的有几点

- 1）在上面的代码中，我们所有的异常都没有处理。实际应用中，在发生异常时，需要关闭 socket，并根据实际业务做一些错误处理工作；
- 2）在客户端，我们没有停止 readThread。实际应用中，我们可以通过关闭 socket 来让线程从阻塞读中返回。推荐读者阅读《Java并发编程实战》；
- 3）我们的服务端只处理了一个客户连接。如果需要同时处理多个客户端，可以创建线程来处理请求。这个作为练习留给读者来完全。

6、Socket、ServerSocket 傻傻分不清楚

在进入这一节的主题前，读者不妨先考虑一个问题：在上一节的实例中，我们运行 echo 服务后，在客户端连接成功时，一个有多少个 socket 存在?

答案是 3 个 socket：客户端一个，服务端有两个。跟这个问题的答案直接关联的是本节的主题——Socket 和 ServerSocket 的区别是什么。

眼尖的读者，可能会注意到在上一节我是这样描述他们的：

在 Java 的 SDK 中，socket 的共有两个接口：用于监听客

户连接的 ServerSocket 和用于通信的 Socket。

注意：我只说 ServerSocket 是用于监听客户连接，而没有说它也可以用来通信。下面我们来详细了解一下他们的区别。

注：以下描述使用的是 UNIX/Linux 系统的 API。

首先，我们创建 ServerSocket 后，内核会创建一个 socket。这个 socket 既可以拿来监听客户连接，也可以连接远端的服务。由于 ServerSocket 是用来监听客户连接的，紧接着它就会对内核创建的这个 socket 调用 listen 函数。这样一来，这个 socket 就成了所谓的 listening socket，它开始监听客户的连接。

接下来，我们的客户端创建一个 Socket，同样的，内核也创建一个 socket 实例。内核创建的这个 socket 跟 ServerSocket 一开始创建的那个没有什么区别。不同的是，接下来 Socket 会对它执行 connect，发起对服务端的连接。前面我们说过，socket API 其实是 TCP 层的封装，所以 connect 后，内核会发送一个 SYN 给服务端。

现在，我们切换角色到服务端。服务端的主机在收到这个 SYN 后，会创建一个新的 socket，这个新创建的 socket 跟客户端继续执行三次握手过程。

三次握手完成后，我们执行的 serverSocket.accept() 会返回一个 Socket 实例，这个 socket 就是上一步内核自动帮我们创建的。

所以说：在一个客户端连接的情况下，其实有 3 个 socket。

关于内核自动创建的这个 socket，还有一个很有意思的地方。它的端口号跟 ServerSocket 是一毛一样的。咦！！不是说，一个端口只能绑定一个 socket 吗？其实这个说法并不够准确。

前面我说的TCP 通过端口号来区分数据属于哪个进程的说法，在 socket 的实现里需要改一改。Socket 并不仅仅使用端口号来区别不同的 socket 实例，而是使用 <peer addr:peer port, local addr:local port> 这个四元组。

在上面的例子中，我们的 ServerSocket 长这样：<*:*, *:9877>。意思是，可以接受任何的客户端，和本地任何 IP。

accept 返回的 Socket 则是这样：<127.0.0.1:xxxx, 127.0.0.1:9877>。其中，xxxx 是客户端的端口号。

如果数据是发送给一个已连接的 socket，内核会找到一个完全匹配的实例，所以数据准确发送给了对端。

如果是客户端要发起连接，这时候只有 <*:*, *:9877> 会匹配成功，所以 SYN 也准确发送给了监听套接字。

Socket/ServerSocket 的区别我们就讲到这里。如果读者觉得不过瘾，可以参考《TCP/IP 详解》卷1、卷2。

7、Socket“长”连接的实现

7.1背景知识

Socket 长连接，指的是在客户和服务端之间保持一个 socket 连接长时间不断开。

比较熟悉 Socket 的读者，可能知道有这样一个 API：

```
socket.setKeepAlive(true);
```

嗯……keep alive，“保持活着”，这个应该就是让 TCP 不断开的意思。那么，我们要实现一个 socket 的长连接，只需要这一个调用即可。

遗憾的是，生活并不总是那么美好。对于 4.4BSD 的实现来说，Socket 的这个 keep alive 选项如果打开并且两个小时内没有通信，那么底层会发一个心跳，看看对方是不是还活着。

注意：两个小时才会发一次。也就是说，在没有实际数据通信的时候，我把网线拔了，你的应用程序要经过两个小时才会知道。

这个话题，对于即时通讯的老手来说，也就是经常讨论的“网络连接心跳保活”这个话题了，感兴趣的话可以读一读《聊聊iOS中网络编程长连接的那些事》、《为何基于

TCP协议的移动端IM仍然需要心跳保活机制？》、《微信团队原创分享：Android版微信后台保活实战分享(网络保活篇)》、《Android端消息推送总结：实现原理、心跳保活、遇到的问题等》。

在说明如果实现长连接前，我们先来理一理我们面临的问题。

假定现在有一对已经连接的 socket，在以下情况发生时候，socket 将不再可用：

- 1）某一端关闭是 socket（这不是废话吗）：主动关闭的一方会发送 FIN，通知对方要关闭 TCP 连接。在这种情况下，另一端如果去读 socket，将会读到 EoF（End of File）。于是我们知道对方关闭了 socket；
- 2）应用程序奔溃：此时 socket 会由内核关闭，结果跟情况1一样；
- 3）系统奔溃：这时候系统是来不及发送 FIN 的，因为它已经跪了。此时对方无法得知这一情况。对方在尝试读取数据时，最后会返回 read time out。如果写数据，则是 host unreachable 之类的错误。
- 4）电缆被挖断、网线被拔：跟情况3差不多，如果没有对 socket 进行读写，两边都不知道发生了事故。跟情况3不同的是，如果我们把网线接回去，socket 依旧可以正常使用。

在上面的几种情形中，有一个共同点就是，只要去读、写

socket，只要 socket 连接不正常，我们就能够知道。基于这一点，要实现一个 socket 长连接，我们需要做的就是不断地给对方写数据，然后读取对方的数据，也就是所谓的**心跳**。只要心还在跳，socket 就是活的。写数据的间隔，需要根据实际的应用需求来决定。

心跳包不是实际的业务数据，根据通信协议的不同，需要做不同的处理。

比方说，我们使用 JSON 进行通信，那么，可以为协议包加一个 type 字段，表面这个 JSON 是心跳还是业务数据：

```
1
2
3
4
5
{
    "type": 0,


}
```

使用二进制协议的情况类似。要求就是，我们能够区别一个数据包是心跳还是真实数据。这样，我们便实现了一个 socket 长连接。

7.2实现示例

这一小节我们一起来实现一个带长连接的 Android echo 客户端。完整的代码可以在本文末尾的附件找到。

首先了接口部分：

```
01
02
03
04
05
06  public final class LongLiveSocket {
07      /**
08       * 错误回调
09       */
10      public interface ErrorCallback {
11          /**
12           * 如果需要重连，返回 true
13           */
14          boolean onError();
15      }
16      /**
17       * 读数据回调
18       */
19      public interface DataCallback {
20          void onData(byte[] data, int offset, int
    len);
    }
```

```
21      /**
22       * 写数据回调
23       */
24      public interface WritingCallback {
25          void onSuccess();
26          void onFail(byte[] data, int offset, int
len);
27      }
28      public LongLiveSocket(String host, int port,
29                            DataCallback
dataCallback, ErrorCallback errorCallback) {
30      }
31      public void write(byte[] data, WritingCallback
callback) {
32      }
33      public void write(byte[] data, int offset, int
34  len, WritingCallback callback) {
35      }
36      public void close() {
37      }
38  }
39
40
41
42
```

```
43
```

我们这个支持长连接的类就叫 LongLiveSocket 好了。如果在 socket 断开后需要重连，只需要在对应的接口里面返回 true 即可（在真实场景里，我们还需要让客户设置重连的等待时间，还有读写、连接的 timeout等。为了简单，这里就直接不支持了。

另外需要注意的一点是，如果要做一个完整的库，需要同时提供阻塞式和回调式API。同样由于篇幅原因，这里直接省掉了。

下面我们直接看实现：

```
001
002
003
004
005
006
007
008
009
010
011
012
```

```
013
014
015
016
017
018
019
020
021
022
023
024
025 public final class LongLiveSocket {
026     private static final String TAG = "LongLiveSocket
027     private static final long RETRY_INTERVAL_MILLIS =
028     private static final long HEART_BEAT_INTERVAL_MILI
029     private static final long HEART_BEAT_TIMEOUT_MILLI
030     /**
031      * 错误回调
032      */
033     public interface ErrorCallback {
034         /**
```

```
035         * 如果需要重连，返回 true
036         */
037        boolean onError();
038    }
039    /**
040     * 读数据回调
041     */
042    public interface DataCallback {
043        void onData(byte[] data, int offset, int len)
044    }
045    /**
046     * 写数据回调
047     */
048    public interface WritingCallback {
049        void onSuccess();
050        void onFail(byte[] data, int offset, int len)
051    }
052    private final String mHost;
053    private final int mPort;
054    private final DataCallback mDataCallback;
055    private final ErrorCallback mErrorCallback;
056    private final HandlerThread mWriterThread;
057    private final Handler mWriterHandler;
```

```
057     private final Handler mUIHandler = new
Handler(Looper.getMainLooper());
058
059     private final Object mLock = new Object();
060
061     private Socket mSocket;
062
063     private boolean mClosed;
064
065     private final Runnable mHeartBeatTask = new Runnal
066
067         private byte[] mHeartBeat = new byte[0];
068
069         @Override
070
071         public void run() {
072
073
074             write(mHeartBeat, new WritingCallback() {
075
076                 @Override
077
078                 public void onSuccess() {
079
080
081                     mWriterHandler.postDelayed(mHear
HEART_BEAT_INTERVAL_MILLIS);
082                     mUIHandler.postDelayed(mHeartBea
HEART_BEAT_TIMEOUT_MILLIS);
083                 }
084
085                 @Override
086
087                 public void onFail(byte[] data, int o
088
089
090
091                 }
092
093             });
094
```

```
079         }
080     };
081     private final Runnable mHeartBeatTimeoutTask = ()
082         Log.e(TAG, "mHeartBeatTimeoutTask#run: heart
083         closeSocket();
084     };
085     public LongLiveSocket(String host, int port,
086                           DataCallback dataCallback,
errorCallback) {
087         mHost = host;
088         mPort = port;
089         mDataCallback = dataCallback;
090         mErrorCallback = errorCallback;
091         mWriterThread = new HandlerThread("socket-wr
092         mWriterThread.start();
093         mWriterHandler = new Handler(mWriterThread.g
094         mWriterHandler.post(this::initSocket);
095     }
096     private void initSocket() {
097         while (true) {
098             if (closed()) return;
099             try {
100                 Socket socket = new Socket(mHost, mP
101                 synchronized (mLock) {
```

```
101
102                     if (mClosed) {
103                         silentlyClose(socket);
104                         return;
105                     }
106                     mSocket = socket;
107
108                     Thread reader = new Thread(new Re
"socket-reader");
109                     reader.start();
110                     mWriterHandler.post(mHeartBeatTa
111                 }
112                 break;
113             } catch (IOException e) {
114                 Log.e(TAG, "initSocket: ", e);
115                 if (closed() || !mErrorCallback.onEr
116                     break;
117                 }
118                 try {
119                     TimeUnit.MILLISECONDS.sleep(RETR
120                 } catch (InterruptedException e1) {
121
122                     break;
123                 }
```

```
123             }
124         }
125     }
126     public void write(byte[] data, WritingCallback ca
127         write(data, 0, data.length, callback);
128     }
129     public void write(byte[] data, int offset, int len
callback) {
130         mWriterHandler.post(() -> {
131             Socket socket = getSocket();
132             if (socket == null) {
133                 throw new IllegalStateException("Sock
134 initialized");
135             }
136             try {
137                 OutputStream outputStream = socket.g
138                 DataOutputStream out = new
DataOutputStream(outputStream);
139                 out.writeInt(len);
140                 out.write(data, offset, len);
141                 callback.onSuccess();
142             } catch (IOException e) {
143                 Log.e(TAG, "write: ", e);
144                 closeSocket();
                callback.onFail(data, offset, len);
```

```
145                 if (!closed() && mErrorCallback.onEr
146                     initSocket();
147                 }
148             }
149         });
150     }
151     private boolean closed() {
152         synchronized (mLock) {
153             return mClosed;
154         }
155     }
156     private Socket getSocket() {
157         synchronized (mLock) {
158             return mSocket;
159         }
160     }
161     private void closeSocket() {
162         synchronized (mLock) {
163             closeSocketLocked();
164         }
165     }
166     private void closeSocketLocked() {
167         if (mSocket == null) return;
168
```

```
167         silentlyClose(mSocket);
168         mSocket = null;
169         mWriterHandler.removeCallbacks(mHeartBeatTas
170     }
171     public void close() {
172         if (Looper.getMainLooper() == Looper.myLoope
173             new Thread() {
174                 @Override
175                 public void run() {
176                     doClose();
177                 }
178             }.start();
179         } else {
180             doClose();
181         }
182     }
183     private void doClose() {
184         synchronized (mLock) {
185             mClosed = true;
186 
187             closeSocketLocked();
188         }
189         mWriterThread.quit();
190 
```

```
189         mWriterThread.interrupt();
190     }
191     private static void silentlyClose(Closeable close
192         if (closeable != null) {
193             try {
194                 closeable.close();
195             } catch (IOException e) {
196                 Log.e(TAG, "silentlyClose: ", e);
197 
198             }
199         }
200     }
201     private class ReaderTask  implements Runnable {
202         private final Socket mSocket;
203         public ReaderTask(Socket socket) {
204             mSocket = socket;
205         }
206         @Override
207         public void run() {
208             try {
209                 readResponse();
210             } catch (IOException e) {
                    Log.e(TAG, "ReaderTask#run: ", e);
```

```
211             }
212         }
213         private void readResponse() throws IOExceptio
214
215             byte[] buffer = new byte[1024];
216             InputStream inputStream = mSocket.getInp
217             DataInputStream in = new DataInputStream
218             while (true) {
219                 int nbyte = in.readInt();
220                 if (nbyte == 0) {
221                     Log.i(TAG, "readResponse: heart
222                     mUIHandler.removeCallbacks(mHear
223                     continue;
224                 }
225                 if (nbyte > buffer.length) {
226                     throw new IllegalStateException("
with len " + nbyte +
227                                         " which exceeds
buffer.length);
228                 }
229                 if (readn(in, buffer, nbyte) != 0) {
230
231                     silentlyClose(mSocket);
232
```

```
233                     break;
234                 }
235                 mDataCallback.onData(buffer, 0, nbyt
236             }
237         }
238         private int readn(InputStream in, byte[] buff
IOException {
239             int offset = 0;
240             while (n > 0) {
241                 int readBytes = in.read(buffer, offs
242                 if (readBytes < 0) {
243
244                     break;
245                 }
246                 n -= readBytes;
247                 offset += readBytes;
248             }
249             return n;
250         }
    }
251 }
252
253
254
```

下面是我们新实现的 **EchoClient**：

```
01
02
public class EchoClient {
03
    private static final String TAG = "EchoClient";
04
    private final LongLiveSocket mLongLiveSocket;
05
    public EchoClient(String host, int port) {
06
        mLongLiveSocket = new LongLiveSocket(
07
                host, port,
08
                (data, offset, len) -> Log.i(TAG,
09
"EchoClient: received: " + new String(data, offset,
len)),
10
11
                () -> true);
12
    }
13
    public void send(String msg) {
14
        mLongLiveSocket.write(msg.getBytes(), new
LongLiveSocket.WritingCallback() {
15
            @Override
16
            public void onSuccess() {
17
                Log.d(TAG, "onSuccess: ");
18
            }
19
            @Override
20
            public void onFail(byte[] data, int
offset, int len) {
21
                Log.w(TAG, "onFail: fail to write:
```

```
" + new String(data, offset, len));


                mLongLiveSocket.write(data, offset,
len, this);
            }
        });
    }
}

```

就这样，一个带 socket 长连接的客户端就完成了。剩余代码跟我们这里的主题没有太大关系，感兴趣的读者可以看看文末附件里的源码或者自己完成这个例子。

下面是一些输出示例：

```
03:54:55.583 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:00.588 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:05.594 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:09.638 12691-12710/com.example.echo
D/EchoClient: onSuccess:

03:55:09.639 12691-12713/com.example.echo
I/EchoClient: EchoClient: received: hello

03:55:10.595 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:14.652 12691-12710/com.example.echo
D/EchoClient: onSuccess:
```

```
03:55:14.654 12691-12713/com.example.echo
I/EchoClient: EchoClient: received: echo

03:55:15.596 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:20.597 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received

03:55:25.602 12691-12713/com.example.echo
I/LongLiveSocket: readResponse: heart beat received
```

最后需要说明的是，如果想节省资源，在有客户发送数据的时候可以省略 heart beat。

我们对读出错时候的处理，可能也存在一些争议。读出错后，我们只是关闭了 socket。socket 需要等到下一次写动作发生时，才会重新连接。实际应用中，如果这是一个问题，在读出错后可以直接开始重连。这种情况下，还需要一些额外的同步，避免重复创建 socket。heart beat timeout 的情况类似。

8、跟 TCP/IP 学协议设计

如果仅仅是为了使用是 socket，我们大可以不去理会协议的细节。之所以推荐大家去看一看《TCP/IP 详解》，是因为它们有太多值得学习的地方。很多我们工作中遇到的问题，都可以在这里找到答案。

以下每一个小节的标题都是一个小问题，建议读者独立思考一下，再继续往下看。

8.1协议版本如何升级?

有这么一句流行的话：这个世界唯一不变的，就是变化。当我们对协议版本进行升级的时候，正确识别不同版本的协议对软件的兼容非常重要。那么，我们如何设计协议，才能够为将来的版本升级做准备呢?

答案可以在 IP 协议找到。

IP 协议的第一个字段叫 version，目前使用的是 4 或 6，分别表示 IPv4 和 IPv6。由于这个字段在协议的开头，接收端收到数据后，只要根据第一个字段的值就能够判断这个数据包是 IPv4 还是 IPv6。

再强调一下，这个字段在两个版本的IP协议都位于第一个字段，为了做兼容处理，对应的这个字段必须位于同一位置。文本协议（如，JSON、HTML）的情况类似。

8.2如何发送不定长数据的数据包?

举个例子，我们用微信发送一条消息。这条消息的长度是不确定的，并且每条消息都有它的边界。我们如何来处理这个边界呢?

还是一样，看看 IP。IP 的头部有个 header length 和 data length 两个字段。通过添加一个 len 域，我们就能够把数

据根据应用逻辑分开。

跟这个相对的，还有另一个方案，那就是在数据的末尾放置终止符。比方说，想 C 语言的字符串那样，我们在每个数据的末尾放一个 \0 作为终止符，用以标识一条消息的尾部。这个方法带来的问题是，用户的数据也可能存在 \0。此时，我们就需要对用户的数据进行转义。比方说，把用户数据的所有 \0 都变成 \0\0。读消息的过程总，如果遇到 \0\0，那它就代表 \0，如果只有一个 \0，那就是消息尾部。

使用 len 字段的好处是，我们不需要对数据进行转义。读取数据的时候，只要根据 len 字段，一次性把数据都读进来就好，效率会更高一些。

终止符的方案虽然要求我们对数据进行扫描，但是如果我们可能从任意地方开始读取数据，就需要这个终止符来确定哪里才是消息的开头了。

当然，这两个方法不是互斥的，可以一起使用。

8.3上传多个文件，只有所有文件都上传成功时才算成功

现在我们有一个需求，需要一次上传多个文件到服务器，只有在所有文件都上传成功的情况下，才算成功。我们该如何来实现呢?

IP 在数据报过大的时候，会把一个数据报拆分成多个，并设置一个 MF （more fragments）位，表示这个包只是被拆分后的数据的一部分。

好，我们也学一学 IP。这里，我们可以给每个文件从 0 开始编号。上传文件的同时，也携带这个编号，并额外附带一个 MF 标志。除了编号最大的文件，所有文件的 MF 标志都置位。因为 MF 没有置位的是最后一个文件，服务器就可以根据这个得出总共有多少个文件。

另一种不使用 MF 标志的方法是，我们在上传文件前，就告诉服务器总共有多少个文件。

如果读者对数据库比较熟悉，学数据库用事务来处理，也是可以的。这里就不展开讨论了。

8.4如何保证数据的有序性?

这里讲一个我曾经遇到过的面试题。现在有一个任务队列，多个工作线程从中取出任务并执行，执行结果放到一个结果队列中。先要求，放入结果队列的时候，顺序顺序需要跟从工作队列取出时的一样（也就是说，先取出的任务，执行结果需要先放入结果队列）。

我们看看 TCP/IP 是怎么处理的。IP 在发送数据的时候，不同数据报到达对端的时间是不确定的，后面发送的数据

有可能较先到达。TCP 为了解决这个问题，给所发送数据的每个字节都赋了一个序列号，通过这个序列号，TCP 就能够把数据按原顺序重新组装。

一样，我们也给每个任务赋一个值，根据进入工作队列的顺序依次递增。工作线程完成任务后，在将结果放入结果队列前，先检查要放入对象的写一个序列号是不是跟自己的任务相同，如果不同，这个结果就不能放进去。此时，最简单的做法是等待，知道下一个可以放入队列的结果是自己所执行的那一个。但是，这个线程就没办法继续处理任务了。

更好的方法是，我们维护多一个结果队列的缓冲，这个缓冲里面的数据按序列号从小到大排序。

工作线程要将结果放入，有两种可能：

- 1）刚刚完成的任务刚好是下一个，将这个结果放入队列。然后从缓冲的头部开始，将所有可以放入结果队列的数据都放进去；
- 2）所完成的任务不能放入结果队列，这个时候就插入结果队列。然后，跟上一种情况一样，需要检查缓冲。

如果测试表明，这个结果缓冲的数据不多，那么使用普通的链表就可以。如果数据比较多，可以使用一个最小堆。

8.5如何保证对方收到了消息?

我们说，TCP 提供了可靠的传输。这样不就能够保证对方收到消息了吗?

很遗憾，其实不能。在我们往 socket 写入的数据，只要对端的内核收到后，就会返回 ACK，此时，socket 就认为数据已经写入成功。然而要注意的是，这里只是对方所运行的系统的内核成功收到了数据，并不表示应用程序已经成功处理了数据。

解决办法还是一样，我们学 TCP，添加一个应用层的 APP ACK。应用接收到消息并处理成功后，发送一个 APP ACK 给对方。

有了 APP ACK，我们需要处理的另一个问题是，如果对方真的没有收到，需要怎么做?

TCP 发送数据的时候，消息一样可能丢失。TCP 发送数据后，如果长时间没有收到对方的 ACK，就假设数据已经丢失，并重新发送。

我们也一样，如果长时间没有收到 APP ACK，就假设数据丢失，重新发送一个。

关于数据送达保证和应应答机制，以下文章进行了详细讨论:

- 《IM消息送达保证机制实现(一)：保证在线实时消息的可靠投递》
- 《IM消息送达保证机制实现(二)：保证离线消息的可靠投递》
- 《IM群聊消息如此复杂，如何保证不丢不重？》
- 《从客户端的角度来谈谈移动端IM的消息可靠性和送达机制》

9、源码附件下载

手把手教你写基于TCP的Socket长连接-源码(52im.net).zip *(142.48 KB，下载次数: 237，售价: 1 金币)*

（原文链接：https://jekton.github.io/2018/06/23/socket-intro/，有改动）

附录：更多网络编程资料

《技术往事：改变世界的TCP/IP协议（珍贵多图、手机慎点）》
《UDP中一个包的大小最大能多大？》
《Java新一代网络编程模型AIO原理及Linux系统AIO介绍》
《NIO框架入门(一)：服务端基于Netty4的UDP双向通信

Demo演示》

《NIO框架入门(二)：服务端基于MINA2的UDP双向通信Demo演示》

《NIO框架入门(三)：iOS与MINA2、Netty4的跨平台UDP双向通信实战》

《NIO框架入门(四)：Android与MINA2、Netty4的跨平台UDP双向通信实战》

《P2P技术详解(一)：NAT详解——详细原理、P2P简介》

《P2P技术详解(二)：P2P中的NAT穿越(打洞)方案详解》

《P2P技术详解(三)：P2P技术之STUN、TURN、ICE详解》

《通俗易懂：快速理解P2P技术中的NAT穿透原理》

>> 更多同类文章 ……