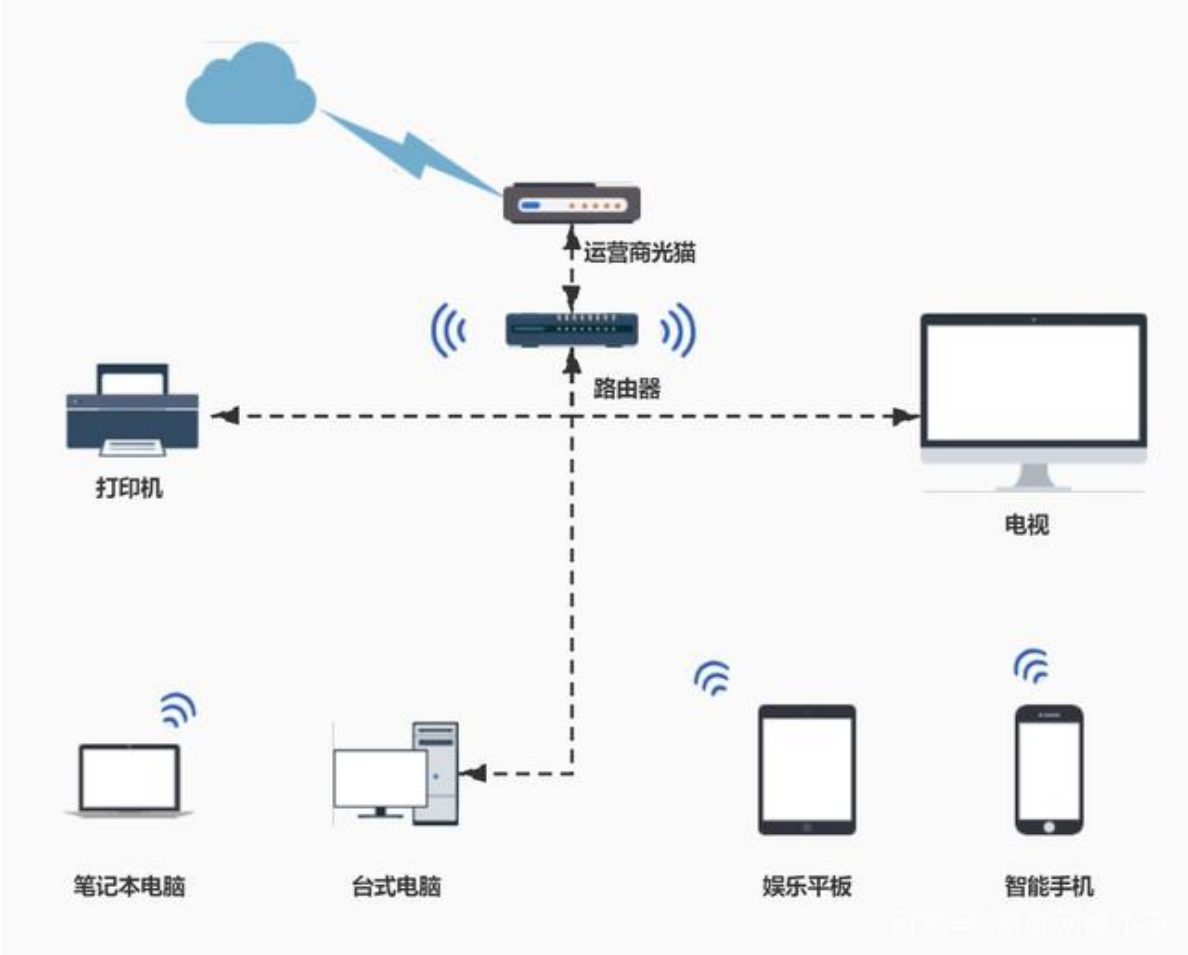
Java网络编程

在JavaSE阶段，我们学习了I/O流，既然I/O流如此强大，那么能否跨越不同的主机进行I/O操作呢？这就要提到Java的网络编程了。

注意：本章会涉及到`计算机网络`相关内容（只会讲解大致内容，不会完整的讲解计算机网络知识）

计算机网络基础

利用通信线路和通信设备，将地理位置不同的、功能独立的多台计算机互连起来，以功能完善的网络软件来实现资源共享和信息传递，就构成了计算机网络系统。

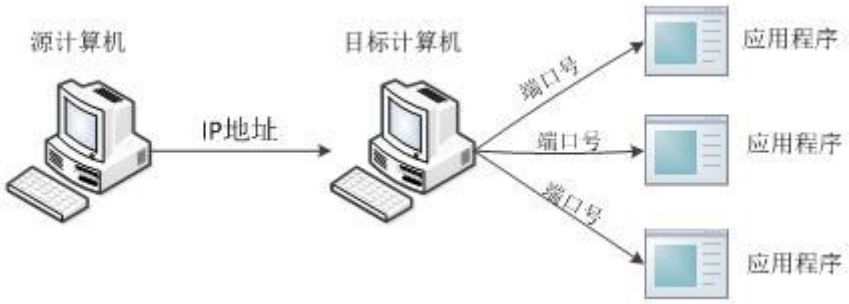


比如我们家里的路由器，通过将我们的设备（手机、平板、电脑、电视剧）连接到路由器，来实现对互联网的访问。实际上，我们的路由器连接在互联网上，而我们的设备又连接了路由器，这样我们的设备就可以通过路由器访问到互联网了。通过网络，我们可以直接访问互联网上的另一台主机，比如我们要把QQ的消息发送给我们的朋友，或是通过远程桌面管理来操作另一台电脑，也可以是连接本地网络上的打印机。

既然我们可以通过网络访问其他计算机，那么如何区别不同的计算机呢？通过IP地址，我们就可以区分不同的计算机了：

每一台电脑在同一个网络上都有一个自己的IP地址，用于区别于其他的电脑，我们可以通过对方主机的IP地址对其进行访问。那么我手机连接的移动流量，能访问到连接家里路由器的电脑吗？（不能，因为他们不属于同一个网络）

而我们的电脑上可能运行着大量的程序，每一个程序可能都需要通过网络来访问其他计算机，那这时该如何区分呢？我们可以通过端口号来区分：

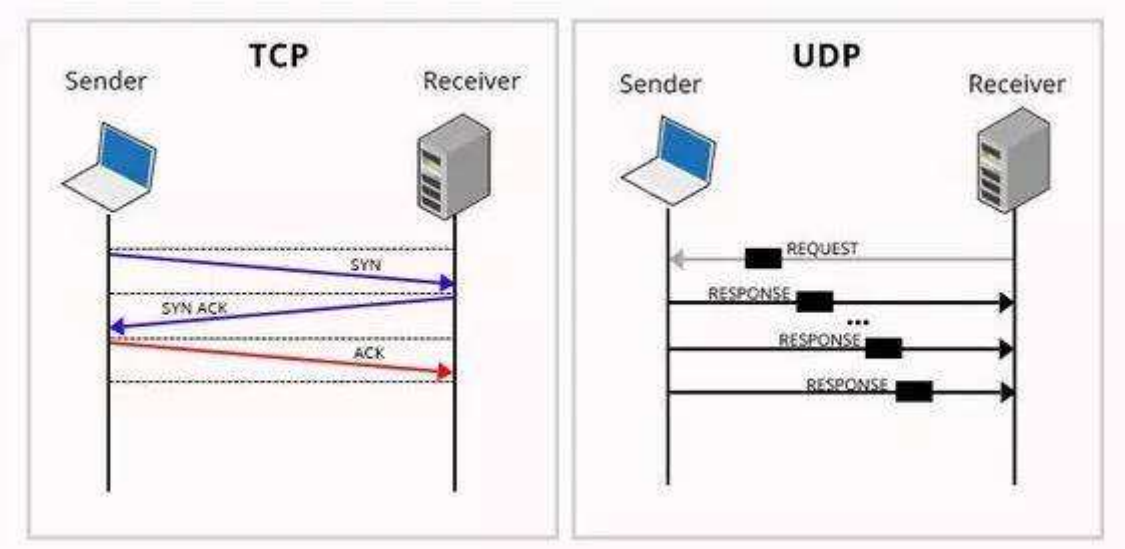


因此，我们一般看到的是这样的：`192.168.0.11:8080`，通过`IP:端口`的形式来访问目标主机上的一个应用程序服务。注意端口号只能是0-65535之间的值！

IP地址分为IPv4和IPv6，IPv4类似于`192.168.0.11`，我们上面提到的例子都是使用的IPv4，它一共有四组数字，每组数字占8个bit位，IPv4地址`0.0.0.0`表示为2进制就是：00000000.00000000.00000000.00000000，共32个bit，最大为`255.255.255.255`，实际上，IPv4能够表示的所有地址，早就已经被用完了。IPv6能够保存128个bit位，因此它也可以表示更多的IP地址，一个IPv6地址看起来像这样：`1030::C9B4:FF12:48AA:1A2B`，目前也正在向IPv6的阶段过度。

TCP和UDP是两种不同的传输层协议：

- TCP：当一台计算机想要与另一台计算机通讯时，两台计算机之间的通信需要畅通且可靠（会进行三次握手，断开也会进行四次挥手），这样才能保证正确收发数据，因此TCP更适合一些可靠的数据传输场景。
- UDP：它是一种无连接协议，数据想发就发，而且不会建立可靠传输，也就是说传输过程中有可能会导致部分数据丢失，但是它比TCP传输更加简单高效，适合视频直播之类的。



了解Socket技术

通过Socket技术（它是计算机之间进行**通信**的**一种约定**或一种方式），我们就可以实现两台计算机之间的通信，Socket也被翻译为`套接字`，是操作系统底层提供的一项通信技术，它支持TCP和UDP。而Java就对socket底层支持进行了一套完整的封装，我们可以通过Java来实现Socket通信。

要实现Socket通信，我们必须创建一个数据发送者和一个数据接收者，也就是客户端和服务端，我们需要提前启动服务端，来等待客户端的连接，而客户端只需要随时启动去连接服务端即可！

```
//服务端
public static void main(String[] args) {
    try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口8080上
        System.out.println("正在等待客户端连接...");
        Socket socket = server.accept();  //当没有客户端连接时，线程会阻塞，直到有客户端连接为止
        System.out.println("客户端已连接，IP地址为："+socket.getInetAddress().getHostAddress());
    }catch (IOException e){
        e.printStackTrace();
    }
}
```

```
//客户端
public static void main(String[] args) {
    try (Socket socket = new Socket("localhost", 8080)){
        System.out.println("已连接到服务端！");
    }catch (IOException e){
        System.out.println("服务端连接失败！");
        e.printStackTrace();
    }
}
```

实际上它就是一个TCP连接的建立过程：

一旦TCP连接建立，服务端和客户端之间就可以相互发送数据，直到客户端主动关闭连接。当然，服务端不仅仅只可以让一个客户端进行连接，我们可以尝试让服务端一直运行来不断接受客户端的连接：

```
public static void main(String[] args) {
    try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口8080上
        System.out.println("正在等待客户端连接...");
        while (true){   //无限循环等待客户端连接
            Socket socket = server.accept();
            System.out.println("客户端已连接，IP地址为："+socket.getInetAddress().getHostAddress());
        }
    }catch (IOException e){
        e.printStackTrace();
    }
}
```

现在我们就可以多次去连接此服务端了。

使用Socket进行数据传输

通过Socket对象，我们就可以获取到对应的I/O流进行网络数据传输：

```
public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 8080);
             Scanner scanner = new Scanner(System.in)){
            System.out.println("已连接到服务端！");
            OutputStream stream = socket.getOutputStream();
```

```
            OutputStreamWriter writer = new OutputStreamWriter(stream);  //通过转
换流来帮助我们快速写入内容
            System.out.println("请输入要发送给服务端的内容：");
            String text = scanner.nextLine();
            writer.write(text+'\n');   //因为对方是readLine()这里加个换行符
            writer.flush();
            System.out.println("数据已发送："+text);
        }catch (IOException e){
            System.out.println("服务端连接失败！");
            e.printStackTrace();
        }finally {
            System.out.println("客户端断开连接！");
        }
    }
}
```

```
public static void main(String[] args) {
    try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口8080上
        System.out.println("正在等待客户端连接...");
        Socket socket = server.accept();
        System.out.println("客户端已连接，IP地址
为："+socket.getInetAddress().getHostAddress());
        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));  //通过
        System.out.print("接收到客户端数据：");
        System.out.println(reader.readLine());
        socket.close();   //和服务端TCP连接完成之后，记得关闭socket
    }catch (IOException e){
        e.printStackTrace();
    }
}
```

同理，既然服务端可以读取客户端的内容，客户端也可以在发送后等待服务端给予响应：

```
public static void main(String[] args) {
    try (Socket socket = new Socket("localhost", 8080);
         Scanner scanner = new Scanner(System.in)){
        System.out.println("已连接到服务端！");
        OutputStream stream = socket.getOutputStream();
        OutputStreamWriter writer = new OutputStreamWriter(stream);  //通过转换流来
帮助我们快速写入内容
        System.out.println("请输入要发送给服务端的内容：");
        String text = scanner.nextLine();
        writer.write(text+'\n');   //因为对方是readLine()这里加个换行符
        writer.flush();
        System.out.println("数据已发送："+text);
        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        System.out.println("收到服务器返回："+reader.readLine());
    }catch (IOException e){
        System.out.println("服务端连接失败！");
        e.printStackTrace();
    }finally {
        System.out.println("客户端断开连接！");
```

```
    }
}
```

```
public static void main(String[] args) {
    try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口8080上
        System.out.println("正在等待客户端连接...");
        Socket socket = server.accept();
        System.out.println("客户端已连接，IP地址
为："+socket.getInetAddress().getHostAddress());
        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));  //通过
        System.out.print("接收到客户端数据：");
        System.out.println(reader.readLine());
        OutputStreamWriter writer = new
OutputStreamWriter(socket.getOutputStream());
        writer.write("已收到！");
        writer.flush();
    }catch (IOException e){
        e.printStackTrace();
    }
}
```

我们可以手动关闭单向的流：

```
socket.shutdownOutput();  //关闭输出方向的流
socket.shutdownInput();  //关闭输入方向的流
```

如果我们不希望服务端等待太长的时间，我们可以通过调用`setSoTimeout()`方法来设定IO超时时间：

```
socket.setSoTimeout(3000);
```

当超过设定时间都依然没有收到客户端或是服务端的数据时，会抛出异常：

```
java.net.SocketTimeoutException: Read timed out
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
    at java.net.SocketInputStream.read(SocketInputStream.java:171)
    at java.net.SocketInputStream.read(SocketInputStream.java:141)
    at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
    at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
    at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    at java.io.InputStreamReader.read(InputStreamReader.java:184)
    at java.io.BufferedReader.fill(BufferedReader.java:161)
    at java.io.BufferedReader.readLine(BufferedReader.java:324)
    at java.io.BufferedReader.readLine(BufferedReader.java:389)
    at com.test.Main.main(Main.java:41)
```

我们之前使用的都是通过构造方法直接连接服务端，那么是否可以等到我们想要的时候再去连接呢？

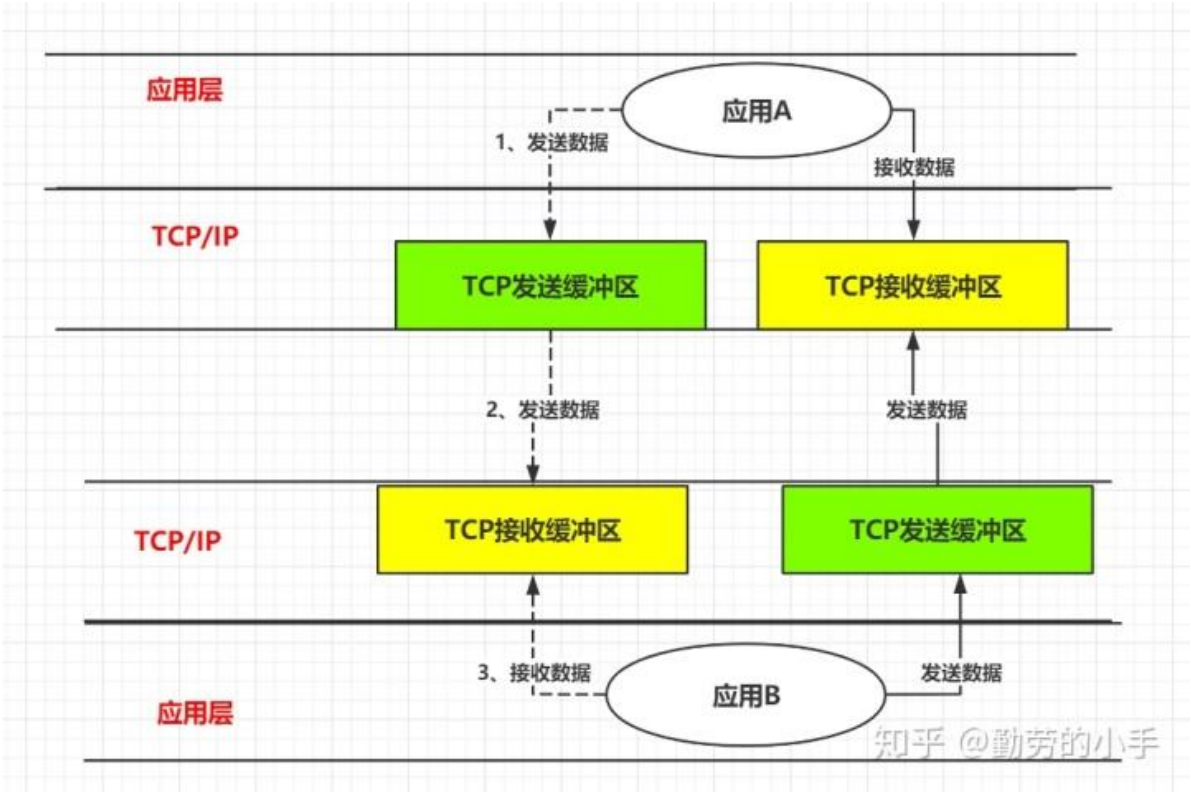
```
try (Socket socket = new Socket(); //调用无参构造不会自动连接
     Scanner scanner = new Scanner(System.in)){
    socket.connect(new InetSocketAddress("localhost", 8080), 1000);  //手动调用
connect方法进行连接
```

如果连接的双方发生意外而通知不到对方，导致一方还持有连接，这样就会占用资源，因此我们可以使用`setKeepAlive()`方法来防止此类情况发生：

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

当客户端连接后，如果设置了keeplive为 true，当对方没有发送任何数据过来，超过一个时间(看系统内核参数配置)，那么我们这边会发送一个ack探测包发到对方，探测双方的TCP/IP连接是否有效。

TCP在传输过程中，实际上会有一个缓冲区用于数据的发送和接收：



此缓冲区大小为：8192，我们可以手动调整其大小来优化传输效率：

```
socket.setReceiveBufferSize(25565);    //TCP接收缓冲区
socket.setSendBufferSize(25565);     //TCP发送缓冲区
```

使用Socket传输文件

既然Socket为我们提供了IO流便于数据传输，那么我们就可以轻松地实现文件传输了。

使用浏览器访问Socket服务器

在了解了如何使用Socket传输文件后，我们来看看，浏览器是如何向服务器发起请求的：

```
public static void main(String[] args) {
        try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口
8080上
            System.out.println("正在等待客户端连接...");
```

```
            Socket socket = server.accept();
            System.out.println("客户端已连接，IP地址
为："+socket.getInetAddress().getHostAddress());
            InputStream in = socket.getInputStream();  //通过
            System.out.println("接收到客户端数据：");
            while (true){
                int i = in.read();
                if(i == -1) break;
                System.out.print((char) i);
            }
        }catch (Exception e){
            e.printStackTrace();
        }
    }
```

我们现在打开浏览器，输入http://localhost:8080或是http://127.0.0.1:8080/，来连接我们本地开放的服务器。

我们发现浏览器是无法打开这个链接的，但是我们服务端却收到了不少的信息：

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="94", "Google Chrome";v="94", ";Not A Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,imag
e/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,und;q=0.8,en;q=0.7
```

实际上这些内容都是Http协议规定的请求头内容。HTTP是一种应用层协议，全称为超文本传输协议，它本质也是基于TCP协议进行数据传输，因此我们的服务端能够读取HTTP请求。但是Http协议并不会保持长连接，在得到我们响应的数据后会立即关闭TCP连接。

既然使用的是Http连接，如果我们的服务器要支持响应HTTP请求，那么就需要按照HTTP协议的规则，返回一个规范的响应文本，首先是响应头，它至少要包含一个响应码：

```
HTTP/1.1 200 Accpeted
```

然后就是响应内容（注意一定要换行再写），我们尝试来编写一下支持HTTP协议的响应内容：

```
public static void main(String[] args) {
    try(ServerSocket server = new ServerSocket(8080)){    //将服务端创建在端口8080上
        System.out.println("正在等待客户端连接...");
```

```
        Socket socket = server.accept();
        System.out.println("客户端已连接，IP地址
为："+socket.getInetAddress().getHostAddress());
        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));  //通过
        System.out.println("接收到客户端数据：");
        while (reader.ready()) System.out.println(reader.readLine());   //ready是
判断当前流中是否还有可读内容
        OutputStreamWriter writer = new
OutputStreamWriter(socket.getOutputStream());
        writer.write("HTTP/1.1 200 Accepted\r\n");   //200是响应码，Http协议规定200
为接受请求，400为错误的请求，404为找不到此资源（不止这些，还有很多）
        writer.write("\r\n");   //在请求头写完之后还要进行一次换行，然后写入我们的响应实体
（会在浏览器上展示的内容）
        writer.write("lbwnb!");
        writer.flush();
    }catch (Exception e){
        e.printStackTrace();
    }
}
```

我们可以打开浏览器的开发者模式（这里推荐使用Chrome/Edge浏览器，按下F12即可打开），我们来观察一下浏览器的实际请求过程。