

# 15 网络编程

---

网络编程是Java最擅长的方向之一，使用Java进行网络编程时，由虚拟机实现了底层复杂的网络协议，Java程序只需要调用Java标准库提供的接口，就可以简单高效地编写网络程序。

本章我们详细介绍如何使用Java进行网络编程。



## 网络编程基础

---

在学习Java网络编程之前，我们先来了解什么是计算机网络。

计算机网络是指两台或更多的计算机组成的网络，在同一个网络中，任意两台计算机都可以直接通信，因为所有计算机都需要遵循同一种网络协议。

那什么是互联网呢？互联网是网络的网络（internet），即把很多计算机网络连接起来，形成一个全球统一的互联网。

对某个特定的计算机网络来说，它可能使用网络协议ABC，而另一个计算机网络可能使用网络协议XYZ。如果计算机网络各自的通讯协议不统一，就没法把不同的网络连接起来形成互联网。因此，为了把计算机网络接入互联网，就必须使用TCP/IP协议。

TCP/IP协议泛指互联网协议，其中最重要的两个协议是TCP协议和IP协议。只有使用TCP/IP协议的计算机才能够联入互联网，使用其他网络协议（例如NetBIOS、AppleTalk协议等）是无法联入互联网的。

## IP地址

在互联网中，一个IP地址用于唯一标识一个网络接口（Network Interface）。一台联入互联网的计算机肯定有一个IP地址，但也可能有多个IP地址。

IP地址分为IPv4和IPv6两种。IPv4采用32位地址，类似 101.202.99.12，而IPv6采用128位地址，类似 2001:0DA8:100A:0000:0000:1020:F2F3:1428。IPv4地址总共有232个（大约42亿），而IPv6地址则总共有2<sup>128</sup>个（大约340万亿亿亿亿），IPv4的地址目前已耗尽，而IPv6的地址是根本用不完的。

IP地址又分为公网IP地址和内网IP地址。公网IP地址可以直接被访问，内网IP地址只能在内网访问。内网IP地址类似于：

- 192.168.x.x
- 10.x.x.x

有一个特殊的IP地址，称之为本机地址，它总是 127.0.0.1。

IPv4地址实际上是一个32位整数。例如：

```
106717964 = 0x65ca630c
           = 65  ca  63  0c
           = 101.202.99.12
```

如果一台计算机只有一个网卡，并且接入了网络，那么，它有一个本机地址 127.0.0.1，还有一个IP地址，例如 101.202.99.12，可以通过这个IP地址接入网络。

如果一台计算机有两块网卡，那么除了本机地址，它可以有两个IP地址，可以分别接入两个网络。通常连接两个网络的设备是路由器或者交换机，它至少有两个IP地址，分别接入不同的网络，让网络之间连接起来。

如果两台计算机位于同一个网络，那么他们之间可以直接通信，因为他们的IP地址前段是相同的，也就是网络号是相同的。网络号是IP地址通过子网掩码过滤后得到的。例如：

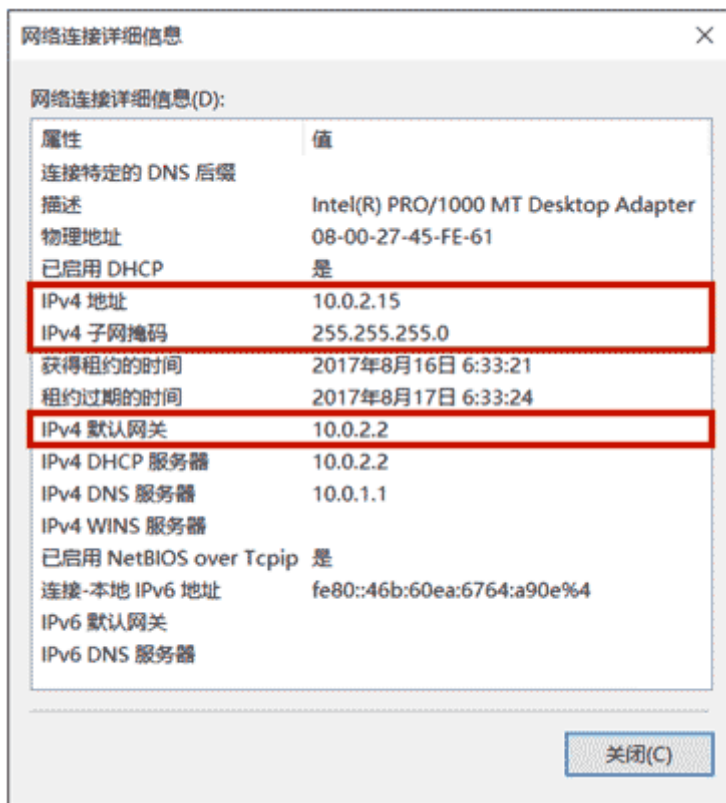
某台计算机的IP是 101.202.99.2，子网掩码是 255.255.255.0，那么计算该计算机的网络号是：

```
IP = 101.202.99.2
Mask = 255.255.255.0
Network = IP & Mask = 101.202.99.0
```

每台计算机都需要正确配置IP地址和子网掩码，根据这两个就可以计算网络号，如果两台计算机计算出的网络号相同，说明两台计算机在同一个网络，可以直接通信。如果两台计算机计算出的网络号不同，那么两台计算机不在同一个网络，不能直接通信，它们之间必须通过路由器或者交换机这样的网络设备间接通信，我们把这种设备称为网关。

网关的作用就是连接多个网络，负责把来自一个网络的数据包发到另一个网络，这个过程叫路由。

所以，一台计算机的一个网卡会有3个关键配置：



- IP地址，例如： 10.0.2.15
- 子网掩码，例如： 255.255.255.0
- 网关的IP地址，例如： 10.0.2.2

## 域名

因为直接记忆IP地址非常困难，所以我们通常使用域名访问某个特定的服务。域名解析服务器DNS负责把域名翻译成对应的IP，客户端再根据IP地址访问服务器。

用 `nslookup` 可以查看域名对应的IP地址：

```
$ nslookup www.liaoxuefeng.com
Server:   xxx.xxx.xxx.xxx
Address:  xxx.xxx.xxx.xxx#53

Non-authoritative answer:
Name:     www.liaoxuefeng.com
Address:  47.98.33.223
```

有一个特殊的本机域名 `localhost`，它对应的IP地址总是本机地址 `127.0.0.1`。

## 网络模型

由于计算机网络从底层的传输到高层的软件设计十分复杂，要合理地设计计算机网络模型，必须采用分层模型，每一层负责处理自己的操作。OSI（Open System Interconnect）网络模型是ISO组织定义的一个计算机互联的标准模型，注意它只是一个定义，目的是为了简化网络各层的操作，提供标准接口便于实现和维护。这个模型从上到下依次是：

- 应用层，提供应用程序之间的通信；
- 表示层：处理数据格式，加解密等等；
- 会话层：负责建立和维护会话；
- 传输层：负责提供端到端的可靠传输；
- 网络层：负责根据目标地址选择路由来传输数据；
- 链路层和物理层负责把数据进行分片并且真正通过物理网络传输，例如，无线网、光纤等。

互联网实际使用的TCP/IP模型并不是对应到OSI的7层模型，而是大致对应OSI的5层模型：

OSI	TCP/IP
应用层	应用层
表示层	
会话层	
传输层	传输层
网络层	IP层
链路层	网络接口层
物理层	

## 常用协议

IP协议是一个分组交换，它不保证可靠传输。而TCP协议是传输控制协议，它是面向连接的协议，支持可靠传输和双向通信。TCP协议是建立在IP协议之上的，简单地说，IP协议只负责发数据包，不保证顺序和正确性，而TCP协议负责控制数据包传输，它在传输数据之前需要先建立连接，建立连接后才能传输数据，传输完后还需要断开连接。TCP协议之所以能保证数据的可靠传输，是通过接收确认、超时重传这些机制实现的。并且，TCP协议允许双向通信，即通信双方可以同时发送和接收数据。

TCP协议也是应用最广泛的协议，许多高级协议都是建立在TCP协议之上的，例如HTTP、SMTP等。

UDP协议（User Datagram Protocol）是一种数据报文协议，它是无连接协议，不保证可靠传输。因为UDP协议在通信前不需要建立连接，因此它的传输效率比TCP高，而且UDP协议比TCP协议要简单得多。

选择UDP协议时，传输的数据通常是能容忍丢失的，例如，一些语音视频通信的应用会选择UDP协议。

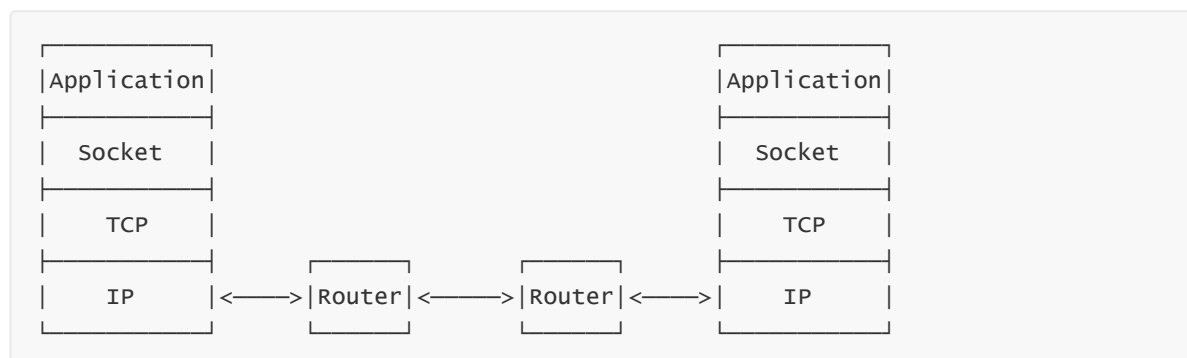
## 小结

计算机网络的基本概念主要有：

- 计算机网络：由两台或更多计算机组成的网络；
- 互联网：连接网络的网络；
- IP地址：计算机的网络接口（通常是网卡）在网络中的唯一标识；
- 网关：负责连接多个网络，并在多个网络之间转发数据的计算机，通常是路由器或交换机；
- 网络协议：互联网使用TCP/IP协议，它泛指互联网协议簇；
- IP协议：一种分组交换传输协议；
- TCP协议：一种面向连接，可靠传输的协议；
- UDP协议：一种无连接，不可靠传输的协议。

## TCP编程

在开发网络应用程序的时候，我们又会遇到Socket这个概念。Socket是一个抽象概念，一个应用程序通过一个Socket来建立一个远程连接，而Socket内部通过TCP/IP协议把数据传输到网络：



Socket、TCP和部分IP的功能都是由操作系统提供的，不同的编程语言只是提供了对操作系统调用的简单的封装。例如，Java提供的几个Socket相关的类就封装了操作系统提供的接口。

为什么需要Socket进行网络通信？因为仅仅通过IP地址进行通信是不够的，同一台计算机同一时间会运行多个网络应用程序，例如浏览器、QQ、邮件客户端等。当操作系统接收到一个数据包的时候，如果只有IP地址，它没法判断应该发给哪个应用程序，所以，操作系统抽象出Socket接口，每个应用程序需要各自对应到不同的Socket，数据包才能根据Socket正确地发到对应的应用程序。

一个Socket就是由IP地址和端口号（范围是0~65535）组成，可以把Socket简单理解为IP地址加端口号。端口号总是由操作系统分配，它是一个0~65535之间的数字，其中，小于1024的端口属于**特权端口**，需要管理员权限，大于1024的端口可以由任意用户的应用程序打开。

- 101.202.99.2:1201
- 101.202.99.2:1304
- 101.202.99.2:15000

使用Socket进行网络编程时，本质上就是两个进程之间的网络通信。其中一个进程必须充当服务器端，它会主动监听某个指定的端口，另一个进程必须充当客户端，它必须主动连接服务器的IP地址和指定端口，如果连接成功，服务器端和客户端就成功地建立了一个TCP连接，双方后续就可以随时发送和接收数据。

因此，当Socket连接成功地在服务器端和客户端之间建立后：

- 对服务器端来说，它的Socket是指定的IP地址和指定的端口号；
- 对客户端来说，它的Socket是它所在计算机的IP地址和一个由操作系统分配的随机端口号。

## 服务器端

要使用Socket编程，我们首先要编写服务器端程序。Java标准库提供了 `ServerSocket` 来实现对指定IP和指定端口的监听。`ServerSocket` 的典型实现代码如下：

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(6666); // 监听指定端口
        System.out.println("server is running...");
        for (;;) {
            Socket sock = ss.accept();
            System.out.println("connected from " +
sock.getRemoteSocketAddress());
            Thread t = new Handler(sock);
            t.start();
        }
    }
}

class Handler extends Thread {
    Socket sock;

    public Handler(Socket sock) {
        this.sock = sock;
    }

    @Override
    public void run() {
        try (InputStream input = this.sock.getInputStream()) {
            try (OutputStream output = this.sock.getOutputStream()) {
                handle(input, output);
            }
        } catch (Exception e) {
            try {
                this.sock.close();
            } catch (IOException ioe) {}
        }
        System.out.println("client disconnected.");
    }
}

    private void handle(InputStream input, OutputStream output) throws
IOException {
        var writer = new BufferedWriter(new OutputStreamWriter(output,
StandardCharsets.UTF_8));
        var reader = new BufferedReader(new InputStreamReader(input,
StandardCharsets.UTF_8));
        writer.write("hello\n");
        writer.flush();
        for (;;) {
            String s = reader.readLine();
            if (s.equals("bye")) {
```

```

        writer.write("bye\n");
        writer.flush();
        break;
    }
    writer.write("ok: " + s + "\n");
    writer.flush();
}
}
}

```

服务器端通过代码：

```

ServerSocket ss = new ServerSocket(6666);

```

在指定端口 6666 监听。这里我们没有指定IP地址，表示在计算机的所有网络接口上进行监听。

如果 `ServerSocket` 监听成功，我们就使用一个无限循环来处理客户端的连接：

```

for (;;) {
    Socket sock = ss.accept();
    Thread t = new Handler(sock);
    t.start();
}

```

注意到代码 `ss.accept()` 表示每当有新的客户端连接进来后，就返回一个 `Socket` 实例，这个 `Socket` 实例就是用来和刚连接的客户端进行通信的。由于客户端很多，要实现并发处理，我们就必须为每个新的 `Socket` 创建一个新线程来处理，这样，主线程的作用就是接收新的连接，每当收到新连接后，就创建一个新线程进行处理。

我们在多线程编程的章节中介绍过线程池，这里也完全可以利用线程池来处理客户端连接，能大大提高运行效率。

如果没有客户端连接进来，`accept()` 方法会阻塞并一直等待。如果有多个客户端同时连接进来，`ServerSocket` 会把连接扔到队列里，然后一个一个处理。对于Java程序而言，只需要通过循环不断调用 `accept()` 就可以获取新的连接。

## 客户端

相比服务器端，客户端程序就要简单很多。一个典型的客户端程序如下：

```

public class Client {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("localhost", 6666); // 连接指定服务器和端口
        try (InputStream input = sock.getInputStream()) {
            try (OutputStream output = sock.getOutputStream()) {
                handle(input, output);
            }
        }
        sock.close();
        System.out.println("disconnected.");
    }

    private static void handle(InputStream input, OutputStream output) throws
    IOException {

```

```

        var writer = new BufferedWriter(new OutputStreamWriter(output,
StandardCharsets.UTF_8));
        var reader = new BufferedReader(new InputStreamReader(input,
StandardCharsets.UTF_8));
        Scanner scanner = new Scanner(System.in);
        System.out.println("[server] " + reader.readLine());
        for (;;) {
            System.out.print(">>> "); // 打印提示
            String s = scanner.nextLine(); // 读取一行输入
            writer.write(s);
            writer.newLine();
            writer.flush();
            String resp = reader.readLine();
            System.out.println("<<< " + resp);
            if (resp.equals("bye")) {
                break;
            }
        }
    }
}

```

客户端程序通过：

```

Socket sock = new Socket("localhost", 6666);

```

连接到服务器端，注意上述代码的服务器地址是 "localhost"，表示本机地址，端口号是 6666。如果连接成功，将返回一个 Socket 实例，用于后续通信。

## Socket流

当Socket连接创建成功后，无论是服务器端，还是客户端，我们都使用 Socket 实例进行网络通信。因为TCP是一种基于流的协议，因此，Java标准库使用 InputStream 和 OutputStream 来封装Socket的数据流，这样我们使用Socket的流，和普通IO流类似：

```

// 用于读取网络数据：
InputStream in = sock.getInputStream();
// 用于写入网络数据：
OutputStream out = sock.getOutputStream();

```

最后我们重点来看看，为什么写入网络数据时，要调用 flush() 方法。

如果不调用 flush()，我们很可能会发现，客户端和服务器都收不到数据，这并不是Java标准库的设计问题，而是我们以流的形式写入数据的时候，并不是一写入就立刻发送到网络，而是先写入内存缓冲区，直到缓冲区满了以后，才会一次性真正发送到网络，这样设计的目的是为了提<sub>高</sub>传输效率。如果缓冲区的数据很少，而我们又想强制把这些数据发送到网络，就必须调用 flush() 强制把缓冲区数据发送出去。

## 练习

下载练习：[使用Socket实现服务器和客户端通信](#)（推荐使用[IDE练习插件](#)快速下载）



## 小结

使用Java进行TCP编程时，需要使用Socket模型：

- 服务器端用 `ServerSocket` 监听指定端口；
- 客户端使用 `Socket(InetAddress, port)` 连接服务器；
- 服务器端用 `accept()` 接收连接并返回 `Socket`；
- 双方通过 `Socket` 打开 `InputStream/OutputStream` 读写数据；
- 服务器端通常使用多线程同时处理多个客户端连接，利用线程池可大幅提升效率；
- `flush()` 用于强制输出缓冲区到网络。

## UDP编程

和TCP编程相比，UDP编程就简单得多，因为UDP没有创建连接，数据包也是一次收发一个，所以没有流的概念。

在Java中使用UDP编程，仍然需要使用Socket，因为应用程序在使用UDP时必须指定网络接口（IP）和端口号。注意：UDP端口和TCP端口虽然都使用0~65535，但他们是两套独立的端口，即一个应用程序用TCP占用了端口1234，不影响另一个应用程序用UDP占用端口1234。

## 服务器端

在服务器端，使用UDP也需要监听指定的端口。Java提供了 `DatagramSocket` 来实现这个功能，代码如下：

```
DatagramSocket ds = new DatagramSocket(6666); // 监听指定端口
for (;;) { // 无限循环
    // 数据缓冲区：
    byte[] buffer = new byte[1024];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    ds.receive(packet); // 收取一个UDP数据包
    // 收取到的数据存储在buffer中，由packet.getOffset(), packet.getLength()指定起始位置
    和长度
    // 将其按UTF-8编码转换为String:
    String s = new String(packet.getData(), packet.getOffset(),
        packet.getLength(), StandardCharsets.UTF_8);
    // 发送数据：
    byte[] data = "ACK".getBytes(StandardCharsets.UTF_8);
    packet.setData(data);
    ds.send(packet);
}
```

服务器端首先使用如下语句在指定的端口监听UDP数据包：

```
DatagramSocket ds = new DatagramSocket(6666);
```

如果没有其他应用程序占据这个端口，那么监听成功，我们就使用一个无限循环来处理收到的UDP数据包：

```
for (;;) {
    // ...
}
```

要接收一个UDP数据包，需要准备一个 `byte[]` 缓冲区，并通过 `DatagramPacket` 实现接收：



```
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
ds.receive(packet);
```

假设我们收取到的是一个 `String`，那么，通过 `DatagramPacket` 返回的 `packet.getOffset()` 和 `packet.getLength()` 确定数据在缓冲区的起止位置：

```
String s = new String(packet.getData(), packet.getOffset(), packet.getLength(),
    StandardCharsets.UTF_8);
```

当服务器收到一个 `DatagramPacket` 后，通常必须立刻回复一个或多个 UDP 包，因为客户端地址在 `DatagramPacket` 中，每次收到的 `DatagramPacket` 可能是不同的客户端，如果不回复，客户端就收不到任何 UDP 包。

发送 UDP 包也是通过 `DatagramPacket` 实现的，发送代码非常简单：

```
byte[] data = ...
packet.setData(data);
ds.send(packet);
```

## 客户端

和服务器端相比，客户端使用 UDP 时，只需要直接向服务器端发送 UDP 包，然后接收返回的 UDP 包：

```
DatagramSocket ds = new DatagramSocket();
ds.setSoTimeout(1000);
ds.connect(InetAddress.getByName("localhost"), 6666); // 连接指定服务器和端口
// 发送：
byte[] data = "Hello".getBytes();
DatagramPacket packet = new DatagramPacket(data, data.length);
ds.send(packet);
// 接收：
byte[] buffer = new byte[1024];
packet = new DatagramPacket(buffer, buffer.length);
ds.receive(packet);
String resp = new String(packet.getData(), packet.getOffset(),
    packet.getLength());
ds.disconnect();
```

客户端打开一个 `DatagramSocket` 使用以下代码：

```
DatagramSocket ds = new DatagramSocket();
ds.setSoTimeout(1000);
ds.connect(InetAddress.getByName("localhost"), 6666);
```

客户端创建 `DatagramSocket` 实例时并不需要指定端口，而是由操作系统自动指定一个当前未使用的端口。紧接着，调用 `setSoTimeout(1000)` 设定超时 1 秒，意思是后续接收 UDP 包时，等待时间最多不会超过 1 秒，否则在没有收到 UDP 包时，客户端会无限等待下去。这一点和服务端不一样，服务端可以无限等待，因为它本来就被设计成长时间运行。

注意到客户端的 `DatagramSocket` 还调用了 `connect()` 方法“连接”到指定的服务器端。不是说 UDP 是无连接的协议吗？为啥这里需要 `connect()`？

这个 `connect()` 方法不是真连接，它是为了在客户端的 `DatagramSocket` 实例中保存服务器端的IP和端口号，确保这个 `DatagramSocket` 实例只能往指定的地址和端口发送UDP包，不能往其他地址和端口发送。这么做不是UDP的限制，而是Java内置了安全检查。

如果客户端希望向两个不同的服务器发送UDP包，那么它必须创建两个 `DatagramSocket` 实例。

后续的收发数据和服务器端是一致的。通常来说，客户端必须先发UDP包，因为客户端不发UDP包，服务器端就根本不知道客户端的地址和端口号。

如果客户端认为通信结束，就可以调用 `disconnect()` 断开连接：

```
ds.disconnect();
```

注意到 `disconnect()` 也不是真正地断开连接，它只是清除了客户端 `DatagramSocket` 实例记录的远程服务器地址和端口号，这样，`DatagramSocket` 实例就可以连接另一个服务器端。

## 练习

下载练习：[使用UDP实现服务器和客户端通信](#)（推荐使用[IDE练习插件](#)快速下载）

## 小结

使用UDP协议通信时，服务器和客户端双方无需建立连接：

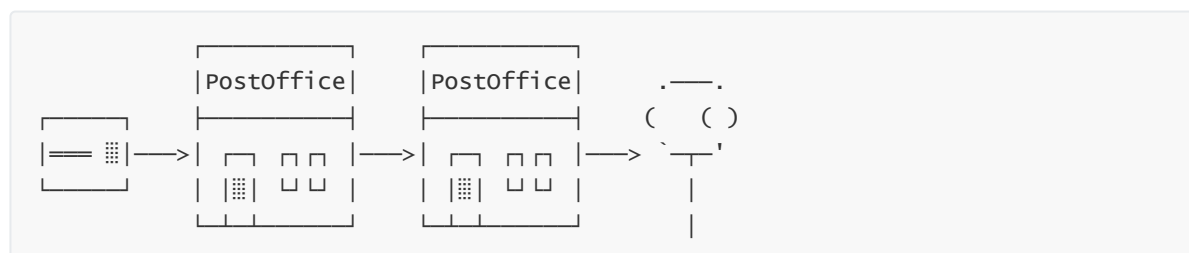
- 服务器端用 `DatagramSocket(port)` 监听端口；
- 客户端使用 `DatagramSocket.connect()` 指定远程地址和端口；
- 双方通过 `receive()` 和 `send()` 读写数据；
- `DatagramSocket` 没有IO流接口，数据被直接写入 `byte[]` 缓冲区。

## 发送Email

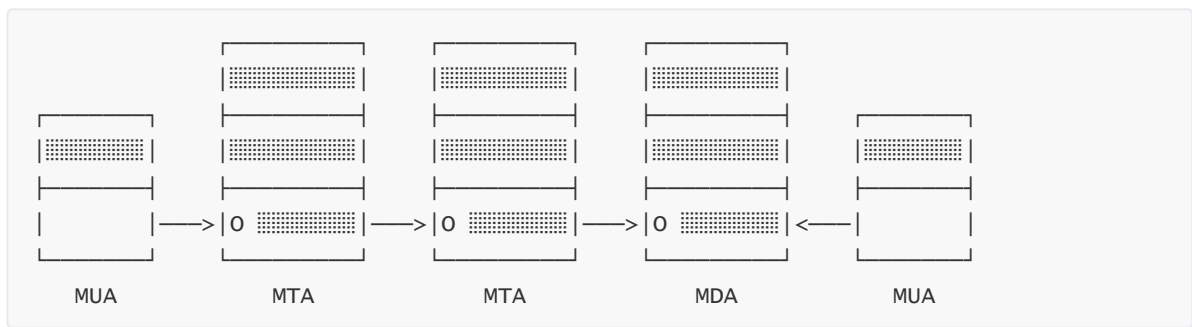
Email就是电子邮件。电子邮件的应用已经有几十年的历史了，我们熟悉的邮箱地址比如 `abc@example.com`，邮件软件比如Outlook都是用来收发邮件的。

使用Java程序也可以收发电子邮件。我们先来看一下传统的邮件是如何发送的。

传统的邮件是通过邮局投递，然后从一个邮局到另一个邮局，最终到达用户的邮箱：



电子邮件的发送过程也是类似的，只不过是电子邮件是从用户电脑的邮件软件，例如Outlook，发送到邮件服务器上，可能经过若干个邮件服务器的中转，最终到达对方邮件服务器上，收件方就可以用软件接收邮件：



我们把类似Outlook这样的邮件软件称为MUA：Mail User Agent，意思是给用户服务的邮件代理；邮件服务器则称为MTA：Mail Transfer Agent，意思是邮件中转的代理；最终到达的邮件服务器称为MDA：Mail Delivery Agent，意思是邮件到达的代理。电子邮件一旦到达MDA，就不再动了。实际上，电子邮件通常就存储在MDA服务器的硬盘上，然后等收件人通过软件或者登陆浏览器查看邮件。

MTA和MDA这样的服务器软件通常是现成的，我们不关心这些服务器内部是如何运行的。要发送邮件，我们关心的是如何编写一个MUA的软件，把邮件发送到MTA上。

MUA到MTA发送邮件的协议就是SMTP协议，它是Simple Mail Transport Protocol的缩写，使用标准端口25，也可以使用加密端口465或587。

SMTP协议是一个建立在TCP之上的协议，任何程序发送邮件都必须遵守SMTP协议。使用Java程序发送邮件时，我们无需关心SMTP协议的底层原理，只需要使用JavaMail这个标准API就可以直接发送邮件。

## 准备SMTP登录信息

假设我们准备使用自己的邮件地址 `me@example.com` 给小明发送邮件，已知小明的邮件地址是 `xiaoming@somewhere.com`，发送邮件前，我们首先要确定作为MTA的邮件服务器地址和端口号。邮件服务器地址通常是 `smtp.example.com`，端口号由邮件服务商确定使用25、465还是587。以下是一些常用邮件服务商的SMTP信息：

- QQ邮箱：SMTP服务器是smtp.qq.com，端口是465/587；
- 163邮箱：SMTP服务器是smtp.163.com，端口是465；
- Gmail邮箱：SMTP服务器是smtp.gmail.com，端口是465/587。

有了SMTP服务器的域名和端口号，我们还需要SMTP服务器的登录信息，通常是使用自己的邮件地址作为用户名，登录口令是用户口令或者一个独立设置的SMTP口令。

我们来看看如何使用JavaMail发送邮件。

首先，我们需要创建一个Maven工程，并把JavaMail相关的两个依赖加入进来：

```
<dependencies>
  <dependency>
    <groupId>javax.mail</groupId>
    <artifactId>javax.mail-api</artifactId>
    <version>1.6.2</version>
  </dependency>
  <dependency>
    <groupId>com.sun.mail</groupId>
    <artifactId>javax.mail</artifactId>
    <version>1.6.2</version>
  </dependency>
  ...
</dependencies>
```

然后，我们通过JavaMail API连接到SMTP服务器上：

```
// 服务器地址:
String smtp = "smtp.office365.com";
// 登录用户名:
String username = "jxsmtp101@outlook.com";
// 登录口令:
String password = "*****";
// 连接到SMTP服务器587端口:
Properties props = new Properties();
props.put("mail.smtp.host", smtp); // SMTP主机名
props.put("mail.smtp.port", "587"); // 主机端口号
props.put("mail.smtp.auth", "true"); // 是否需要用户认证
props.put("mail.smtp.starttls.enable", "true"); // 启用TLS加密
// 获取Session实例:
Session session = Session.getInstance(props, new Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(username, password);
    }
});
// 设置debug模式便于调试:
session.setDebug(true);
```

以587端口为例，连接SMTP服务器时，需要准备一个Properties对象，填入相关信息。最后获取Session实例时，如果服务器需要认证，还需要传入一个Authenticator对象，并返回指定的用户名和口令。

当我们获取到Session实例后，打开调试模式可以看到SMTP通信的详细内容，便于调试。

## 发送邮件

发送邮件时，我们需要构造一个Message对象，然后调用Transport.send(Message)即可完成发送：

```
MimeMessage message = new MimeMessage(session);
// 设置发送方地址:
message.setFrom(new InternetAddress("me@example.com"));
// 设置接收方地址:
message.setRecipient(Message.RecipientType.TO, new
InternetAddress("xiaoming@somewhere.com"));
// 设置邮件主题:
message.setSubject("Hello", "UTF-8");
// 设置邮件正文:
message.setText("Hi Xiaoming...", "UTF-8");
// 发送:
Transport.send(message);
```

绝大多数邮件服务器要求发送方地址和登录用户名必须一致，否则发送将失败。

填入真实的地址，运行上述代码，我们可以在控制台看到JavaMail打印的调试信息：

```
这是JavaMail打印的调试信息:
DEBUG: setDebug: JavaMail version 1.6.2
DEBUG: getProvider() returning
javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Oracle]
DEBUG SMTP: need username and password for authentication
DEBUG SMTP: protocolConnect returning false, host=smtp.office365.com, ...
DEBUG SMTP: useEhlo true, useAuth true
开始尝试连接smtp.office365.com:
DEBUG SMTP: trying to connect to host "smtp.office365.com", port 587, ...
```

```
DEBUG SMTP: connected to host "smtp.office365.com", port: 587
发送命令EHLO:
EHLO localhost
SMTP服务器响应250:
250-SG3P274CA0024.outlook.office365.com Hello
250-SIZE 157286400
...
DEBUG SMTP: Found extension "SIZE", arg "157286400"
发送命令STARTTLS:
STARTTLS
SMTP服务器响应220:
220 2.0.0 SMTP server ready
EHLO localhost
250-SG3P274CA0024.outlook.office365.com Hello [111.196.164.63]
250-SIZE 157286400
250-PIPELINING
250-...
DEBUG SMTP: Found extension "SIZE", arg "157286400"
...
尝试登录:
DEBUG SMTP: protocolConnect login, host=smtp.office365.com, user=*****,
password=*****
DEBUG SMTP: Attempt to authenticate using mechanisms: LOGIN PLAIN DIGEST-MD5 NTLM
XOAUTH2
DEBUG SMTP: Using mechanism LOGIN
DEBUG SMTP: AUTH LOGIN command trace suppressed
登录成功:
DEBUG SMTP: AUTH LOGIN succeeded
DEBUG SMTP: use8bit false
开发发送邮件, 设置FROM:
MAIL FROM:<*****@outlook.com>
250 2.1.0 Sender OK
设置TO:
RCPT TO:<*****@sina.com>
250 2.1.5 Recipient OK
发送邮件数据:
DATA
服务器响应354:
354 start mail input; end with <CRLF>.<CRLF>
真正的邮件数据:
Date: Mon, 2 Dec 2019 09:37:52 +0800 (CST)
From: *****@outlook.com
To: *****001@sina.com
Message-ID: <1617791695.0.1575250672483@localhost>
邮件主题是编码后的文本:
Subject: =?UTF-8?Q?JavaMail=E9=82=AE=E4=BB=B6?=
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: base64

邮件正文是Base64编码的文本:
SGVsbG8sI0i/meaYr+S4gOwwgeadpeiHqmphdmFtYWls55qE6YKu5Lu277yB
.
邮件数据发送完成后, 以\r\n.\r\n结束, 服务器响应250表示发送成功:
250 2.0.0 OK <HK0PR03MB4961.apcprd03.prod.outlook.com>
[Hostname=HK0PR03MB4961.apcprd03.prod.outlook.com]
DEBUG SMTP: message successfully delivered to mail server
发送QUIT命令:
```

```
QUIT
服务器响应221结束TCP连接：
221 2.0.0 Service closing transmission channel
```

从上面的调试信息可以看出，SMTP协议是一个请求-响应协议，客户端总是发送命令，然后等待服务器响应。服务器响应总是以数字开头，后面的信息才是用于调试的文本。这些响应码已经被定义在[SMTP协议](#)中了，查看具体的响应码就可以知道出错原因。

如果一切顺利，对方将收到一封文本格式的电子邮件：

**JavaMail邮件** ☆ 📧 📧

jxsmtp101 于2019年12月2日 星期一 上午09:37 发送给 javacourse001...

Hello, 这是一封来自javamail的邮件！

## 发送HTML邮件

发送HTML邮件和文本邮件是类似的，只需要把：

```
message.setText(body, "UTF-8");
```

改为：

```
message.setText(body, "UTF-8", "html");
```

传入的 body 是类似 `Hello! , xxx` 这样的HTML字符串即可。

HTML邮件可以在邮件客户端直接显示为网页格式：

**Java HTML邮件** ☆ 📧 📧

jxsmtp101 于2019年12月2日 星期一 上午09:41 发送给 javacourse001...

Hello

这是一封[javamail](#)HTML邮件！

## 发送附件

要在电子邮件中携带附件，我们就不能直接调用 `message.setText()` 方法，而是要构造一个 `Multipart` 对象：

```

Multipart multipart = new MimeMultipart();
// 添加text:
BodyPart textpart = new MimeBodyPart();
textpart.setContent(body, "text/html;charset=utf-8");
multipart.addBodyPart(textpart);
// 添加image:
BodyPart imagepart = new MimeBodyPart();
imagepart.setFileName(fileName);
imagepart.setDataHandler(new DataHandler(new ByteArrayDataSource(input,
"application/octet-stream")));
multipart.addBodyPart(imagepart);
// 设置邮件内容为multipart:
message.setContent(multipart);

```

一个 `Multipart` 对象可以添加若干个 `BodyPart`，其中第一个 `BodyPart` 是文本，即邮件正文，后面的 `BodyPart` 是附件。`BodyPart` 依靠 `setContent()` 决定添加的内容，如果添加文本，用 `setContent("...", "text/plain;charset=utf-8")` 添加纯文本，或者用 `setContent("...", "text/html;charset=utf-8")` 添加HTML文本。如果添加附件，需要设置文件名（不一定和真实文件名一致），并且添加一个 `DataHandler()`，传入文件的MIME类型。二进制文件可以用 `application/octet-stream`，Word文档则是 `application/msword`。

最后，通过 `setContent()` 把 `Multipart` 添加到 `Message` 中，即可发送。

带附件的邮件在客户端会被提示下载：



## 发送内嵌图片的HTML邮件

有些童鞋可能注意到，HTML邮件中可以内嵌图片，这是怎么做到的？

如果给一个`，这样的外部图片链接通常会被邮件客户端过滤，并提示用户显示图片并不安全。只有内嵌的图片才能正常在邮件中显示。

内嵌图片实际上也是一个附件，即邮件本身也是 `Multipart`，但需要做一点额外的处理：



```

Multipart multipart = new MimeMultipart();
// 添加text:
BodyPart textpart = new MimeBodyPart();
textpart.setContent("<h1>Hello</h1><p><img src=\"cid:img01\"></p>",
"text/html; charset=utf-8");
multipart.addBodyPart(textpart);
// 添加image:
BodyPart imagepart = new MimeBodyPart();
imagepart.setFileName(fileName);
imagepart.setDataHandler(new DataHandler(new ByteArrayDataSource(input,
"image/jpeg")));
// 与HTML的关联:
imagepart.setHeader("Content-ID", "<img01>");
multipart.addBodyPart(imagepart);

```

在HTML邮件中引用图片时，需要设定一个ID，用类似`引用，然后，在添加图片作为BodyPart时，除了要正确设置MIME类型（根据图片类型使用 image/jpeg 或 image/png），还需要设置一个Header：

```
imagepart.setHeader("Content-ID", "<img01>");
```

这个ID和HTML中引用的ID对应起来，邮件客户端就可以正常显示内嵌图片：

**Hello Java HTML邮件内嵌图片** ★ 📧 📧

jxsmtp101 于2019年12月2日 星期一 上午09:42 发送给 javacourse001...

Hello



这是一封内嵌图片的javamail邮件！

📎 附件 (1个)

全部下载

## 常见问题

如果用户名或口令错误，会导致 535 登录失败：

```
DEBUG SMTP: AUTH LOGIN failed
Exception in thread "main" javax.mail.AuthenticationFailedException: 535 5.7.3
Authentication unsuccessful [HK0PR03CA0105.apcprd03.prod.outlook.com]
```

如果登录用户和发件人不一致，会导致 554 拒绝发送错误：

```
DEBUG SMTP: MessagingException while sending, THROW:
com.sun.mail.smtp.SMTPSendFailedException: 554 5.2.0
STOREDRV.Submission.Exception:SendAsDeniedException.MapiExceptionSendAsDenied;
```

有些时候，如果邮件主题和正文过于简单，会导致 554 被识别为垃圾邮件的错误：

```
DEBUG SMTP: MessagingException while sending, THROW:
com.sun.mail.smtp.SMTPSendFailedException: 554 DT:SPM
```

## 练习

下载练习：[使用SMTP发送邮件](#)（推荐使用[IDE练习插件](#)快速下载）

## 小结

- 使用JavaMail API发送邮件本质上是一个MUA软件通过SMTP协议发送邮件至MTA服务器；
- 打开调试模式可以看到详细的SMTP交互信息；
- 某些邮件服务商需要开启SMTP，并需要独立的SMTP登录密码。

## 接收Email

发送Email的过程我们在上一节已经讲过了，客户端总是通过SMTP协议把邮件发送给MTA。

接收Email则相反，因为邮件最终到达收件人的MDA服务器，所以，接收邮件是收件人用自己的客户端把邮件从MDA服务器上抓取到本地的过程。

接收邮件使用最广泛的协议是POP3：Post Office Protocol version 3，它也是一个建立在TCP连接之上的协议。POP3服务器的标准端口是110，如果整个会话需要加密，那么使用加密端口995。

另一种接收邮件的协议是IMAP：Internet Mail Access Protocol，它使用标准端口143和加密端口993。IMAP和POP3的主要区别是，IMAP协议在本地的所有操作都会自动同步到服务器上，并且，IMAP可以允许用户在邮件服务器的收件箱中创建文件夹。

JavaMail也提供了IMAP协议的支持。因为POP3和IMAP的使用方式非常类似，因此我们只介绍POP3的用法。

使用POP3收取Email时，我们无需关心POP3协议底层，因为JavaMail提供了高层接口。首先需要连接到Store对象：

```
// 准备登录信息：
String host = "pop3.example.com";
int port = 995;
String username = "bob@example.com";
String password = "password";
```

```

Properties props = new Properties();
props.setProperty("mail.store.protocol", "pop3"); // 协议名称
props.setProperty("mail.pop3.host", host); // POP3主机名
props.setProperty("mail.pop3.port", String.valueOf(port)); // 端口号
// 启动SSL:
props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
props.put("mail.smtp.socketFactory.port", String.valueOf(port));

// 连接到Store:
URLName url = new URLName("pop3", host, port, "", username, password);
Session session = Session.getInstance(props, null);
session.setDebug(true); // 显示调试信息
Store store = new POP3SSLStore(session, url);
store.connect();

```

一个 `Store` 对象表示整个邮箱的存储，要收取邮件，我们需要通过 `Store` 访问指定的 `Folder`（文件夹），通常是 `INBOX` 表示收件箱：

```

// 获取收件箱:
Folder folder = store.getFolder("INBOX");
// 以读写方式打开:
folder.open(Folder.READ_WRITE);
// 打印邮件总数/新邮件数量/未读数量/已删除数量:
System.out.println("Total messages: " + folder.getMessageCount());
System.out.println("New messages: " + folder.getNewMessageCount());
System.out.println("Unread messages: " + folder.getUnreadMessageCount());
System.out.println("Deleted messages: " + folder.getDeletedMessageCount());
// 获取每一封邮件:
Message[] messages = folder.getMessages();
for (Message message : messages) {
    // 打印每一封邮件:
    printMessage((MimeMessage) message);
}

```

当我们获取到一个 `Message` 对象时，可以强制转型为 `MimeMessage`，然后打印出邮件主题、发件人、收件人等信息：

```

void printMessage(MimeMessage msg) throws IOException, MessagingException {
    // 邮件主题:
    System.out.println("Subject: " + MimeUtility.decodeText(msg.getSubject()));
    // 发件人:
    Address[] froms = msg.getFrom();
    InternetAddress address = (InternetAddress) froms[0];
    String personal = address.getPersonal();
    String from = personal == null ? address.getAddress() :
(MimeUtility.decodeText(personal) + " <" + address.getAddress() + ">");
    System.out.println("From: " + from);
    // 继续打印收件人:
    ...
}

```

比较麻烦的是获取邮件的正文。一个 `MimeMessage` 对象也是一个 `Part` 对象，它可能只包含一个文本，也可能是一个 `Multipart` 对象，即由几个 `Part` 构成，因此，需要递归地解析出完整的正文：

```

String getBody(Part part) throws MessagingException, IOException {

```

```

    if (part.isMimeType("text/*")) {
        // Part是文本:
        return part.getContent().toString();
    }
    if (part.isMimeType("multipart/*")) {
        // Part是一个Multipart对象:
        Multipart multipart = (Multipart) part.getContent();
        // 循环解析每个子Part:
        for (int i = 0; i < multipart.getCount(); i++) {
            BodyPart bodyPart = multipart.getBodyPart(i);
            String body = getBody(bodyPart);
            if (!body.isEmpty()) {
                return body;
            }
        }
    }
    return "";
}

```

最后记得关闭 Folder 和 Store :

```

folder.close(true); // 传入true表示删除操作会同步到服务器上（即删除服务器收件箱的邮件）
store.close();

```

## 练习

下载练习: [使用POP3接收邮件](#) (推荐使用[IDE练习插件](#)快速下载)

## 小结

- 使用Java接收Email时, 可以用POP3协议或IMAP协议。
- 使用POP3协议时, 需要用Maven引入JavaMail依赖, 并确定POP3服务器的域名 / 端口 / 是否使用SSL等, 然后, 调用相关API接收Email。
- 设置debug模式可以查看通信详细内容, 便于排查错误。

## HTTP编程

什么是HTTP? HTTP就是目前使用最广泛的Web应用程序使用的基础协议, 例如, 浏览器访问网站, 手机App访问后台服务器, 都是通过HTTP协议实现的。

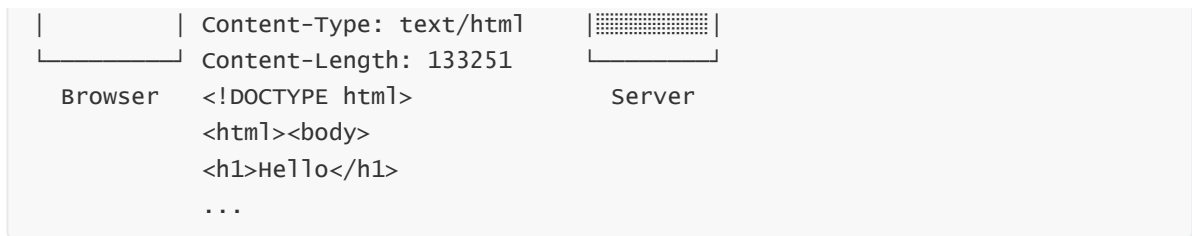
HTTP是HyperText Transfer Protocol的缩写, 翻译为超文本传输协议, 它是基于TCP协议之上的一种请求-响应协议。

我们来看一下浏览器请求访问某个网站时发送的HTTP请求-响应。当浏览器希望访问某个网站时, 浏览器和网站服务器之间首先建立TCP连接, 且服务器总是使用 80 端口和加密端口 443, 然后, 浏览器向服务器发送一个HTTP请求, 服务器收到后, 返回一个HTTP响应, 并且在响应中包含了HTML的网页内容, 这样, 浏览器解析HTML后就可以给用户显示网页了。一个完整的HTTP请求-响应如下:

```

GET / HTTP/1.1
Host: www.sina.com.cn
User-Agent: Mozilla/5 MSIE
Accept: */*
Accept-Language: zh-CN,en
|----->
|<-----|
| HTTP/1.1 200 OK

```



HTTP请求的格式是固定的，它由HTTP Header和HTTP Body两部分构成。第一行总是 请求方法 路径 HTTP版本，例如，GET / HTTP/1.1 表示使用 GET 请求，路径是 /，版本是 HTTP/1.1。

后续的每一行都是固定的 Header: value 格式，我们称为HTTP Header，服务器依靠某些特定的Header来识别客户端请求，例如：

- Host：表示请求的域名，因为一台服务器上可能有多个网站，因此有必要依靠Host来识别用于请求；
- User-Agent：表示客户端自身标识信息，不同的浏览器有不同的标识，服务器依靠User-Agent判断客户端类型；
- Accept：表示客户端能处理的HTTP响应格式，\*/ \* 表示任意格式，text/\* 表示任意文本，image/png 表示PNG格式的图片；
- Accept-Language：表示客户端接收的语言，多种语言按优先级排序，服务器依靠该字段给用户返回特定语言的网页版本。

如果是 GET 请求，那么该HTTP请求只有HTTP Header，没有HTTP Body。如果是 POST 请求，那么该HTTP请求带有Body，以一个空行分隔。一个典型的带Body的HTTP请求如下：

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

username=hello&password=123456
```

POST 请求通常要设置 Content-Type 表示Body的类型，Content-Length 表示Body的长度，这样服务器就可以根据请求的Header和Body做出正确的响应。

此外，GET 请求的参数必须附加在URL上，并以URLEncode方式编码，例如：

http://www.example.com/?a=1&b=k%26R，参数分别是 a=1 和 b=k&R。因为URL的长度限制，GET 请求的参数不能太多，而 POST 请求的参数就没有长度限制，因为 POST 请求的参数必须放到Body中。并且，POST 请求的参数不一定是URL编码，可以按任意格式编码，只需要在 Content-Type 中正确设置即可。常见的发送JSON的 POST 请求如下：

```
POST /login HTTP/1.1
Content-Type: application/json
Content-Length: 38

{"username":"bob","password":"123456"}
```

HTTP响应也是由Header和Body两部分组成，一个典型的HTTP响应如下：

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 133251

<!DOCTYPE html>
<html><body>
<h1>Hello</h1>
...
```

响应的第一行总是 HTTP 版本 响应代码 响应说明，例如，HTTP/1.1 200 OK 表示版本是 HTTP/1.1，响应代码是 200，响应说明是 OK。客户端只依赖响应代码判断 HTTP 响应是否成功。HTTP 有固定的响应代码：

- 1xx：表示一个提示性响应，例如 101 表示将切换协议，常见于 WebSocket 连接；
- 2xx：表示一个成功的响应，例如 200 表示成功，206 表示只发送了部分内容；
- 3xx：表示一个重定向的响应，例如 301 表示永久重定向，303 表示客户端应该按指定路径重新发送请求；
- 4xx：表示一个因为客户端问题导致的错误响应，例如 400 表示因为 Content-Type 等各种原因导致的无效请求，404 表示指定的路径不存在；
- 5xx：表示一个因为服务器问题导致的错误响应，例如 500 表示服务器内部故障，503 表示服务器暂时无法响应。

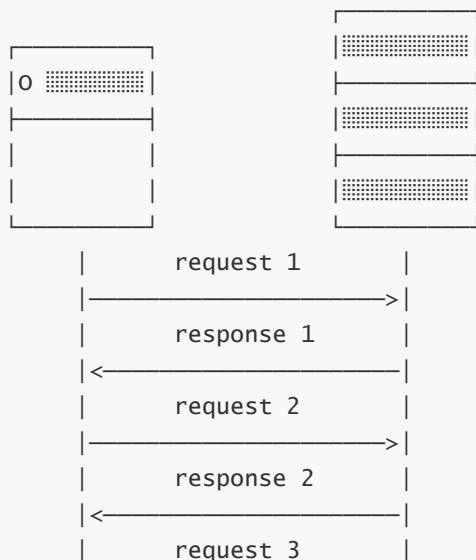
当浏览器收到第一个 HTTP 响应后，它解析 HTML 后，又会发送一系列 HTTP 请求，例如，GET /logo.jpg HTTP/1.1 请求一个图片，服务器响应图片请求后，会直接把二进制内容的图片发送给浏览器：

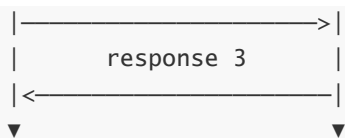
```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 18391

????JFIFH??XExifMM?i&??X?... (二进制的 JPEG 图片)
```

因此，服务器总是被动地接收客户端的一个 HTTP 请求，然后响应它。客户端则根据需要发送若干个 HTTP 请求。

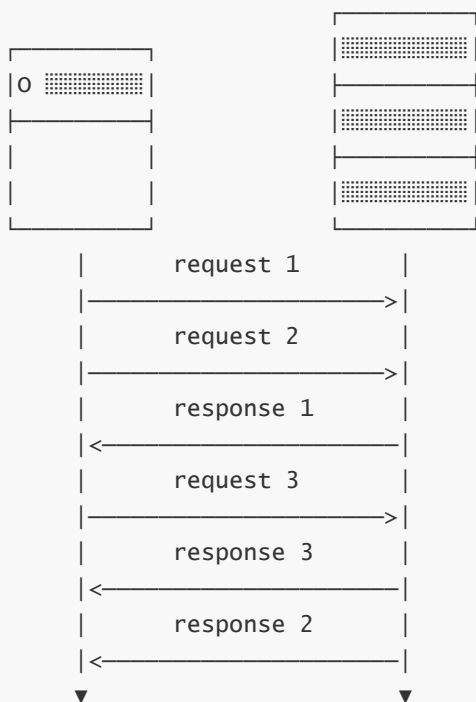
对于最早期的 HTTP/1.0 协议，每次发送一个 HTTP 请求，客户端都需要先创建一个新的 TCP 连接，然后，收到服务器响应后，关闭这个 TCP 连接。由于建立 TCP 连接就比较耗时，因此，为了提高效率，HTTP/1.1 协议允许在一个 TCP 连接中反复发送-响应，这样就能大大提高效率：





因为HTTP协议是一个请求-响应协议，客户端在发送了一个HTTP请求后，必须等待服务器响应后，才能发送下一个请求，这样一来，如果某个响应太慢，它就会堵住后面的请求。

所以，为了进一步提速，HTTP/2.0允许客户端在没有收到响应的时候，发送多个HTTP请求，服务器返回响应的时候，不一定按顺序返回，只要双方能识别出哪个响应对应哪个请求，就可以做到并行发送和接收：



可见，HTTP/2.0进一步提高了效率。

## HTTP编程

既然HTTP涉及到客户端和服务端，和TCP类似，我们也需要针对客户端编程和针对服务器端编程。

本节我们不讨论服务器端的HTTP编程，因为服务器端的HTTP编程本质上就是编写Web服务器，这是一个非常复杂的体系，也是JavaEE开发的核心内容，我们在后面的章节再仔细研究。

本节我们只讨论作为客户端的HTTP编程。

因为浏览器也是一种HTTP客户端，所以，客户端的HTTP编程，它的行为本质上和浏览器是一样的，即发送一个HTTP请求，接收服务器响应后，获得响应内容。只不过浏览器进一步把响应内容解析后渲染并展示给了用户，而我们使用Java进行HTTP客户端编程仅限于获得响应内容。

我们来看一下Java如果使用HTTP客户端编程。

Java标准库提供了基于HTTP的包，但是要注意，早期的JDK版本是通过 `HttpURLConnection` 访问HTTP，典型代码如下：

```
URL url = new URL("http://www.example.com/path/to/target?a=1&b=2");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setUseCaches(false);
```



```

conn.setConnectTimeout(5000); // 请求超时5秒
// 设置HTTP头:
conn.setRequestProperty("Accept", "/*/*");
conn.setRequestProperty("User-Agent", "Mozilla/5.0 (compatible; MSIE 11; windows NT 5.1)");
// 连接并发送HTTP请求:
conn.connect();
// 判断HTTP响应是否200:
if (conn.getResponseCode() != 200) {
    throw new RuntimeException("bad response");
}
// 获取所有响应Header:
Map<String, List<String>> map = conn.getHeaderFields();
for (String key : map.keySet()) {
    System.out.println(key + ": " + map.get(key));
}
// 获取响应内容:
InputStream input = conn.getInputStream();
// ...

```

上述代码编写比较繁琐，并且需要手动处理 `InputStream`，所以用起来很麻烦。

从Java 11开始，引入了新的 `HttpClient`，它使用链式调用的API，能大大简化HTTP的处理。

我们来看一下如何使用新版的 `HttpClient`。首先需要创建一个全局 `HttpClient` 实例，因为 `HttpClient` 内部使用线程池优化多个HTTP连接，可以复用：

```

static HttpClient httpClient = HttpClient.newBuilder().build();

```

使用 GET 请求获取文本内容代码如下：

```

import java.net.URI;
import java.net.http.*;
import java.net.http.HttpClient.Version;
import java.time.Duration;
import java.util.*;
public class Main {
    // 全局HttpClient:
    static HttpClient httpClient = HttpClient.newBuilder().build();

    public static void main(String[] args) throws Exception {
        String url = "https://www.sina.com.cn/";
        HttpRequest request = HttpRequest.newBuilder(new URI(url))
            // 设置Header:
            .header("User-Agent", "Java HttpClient").header("Accept", "/*/*")
            // 设置超时:
            .timeout(Duration.ofSeconds(5))
            // 设置版本:
            .version(Version.HTTP_2).build();
        HttpResponse<String> response = httpClient.send(request,
            HttpResponse.BodyHandlers.ofString());
        // HTTP允许重复的Header，因此一个Header可对应多个value:
        Map<String, List<String>> headers = response.headers().map();
        for (String header : headers.keySet()) {
            System.out.println(header + ": " + headers.get(header).get(0));
        }
        System.out.println(response.body().substring(0, 1024) + "...");
    }
}

```

```
}  
}
```

如果我们要获取图片这样的二进制内容，只需要把 `HttpResponse.BodyHandlers.ofString()` 换成 `HttpResponse.BodyHandlers.ofByteArray()`，就可以获得一个 `HttpResponse` 对象。如果响应的内容很大，不希望一次性全部加载到内存，可以使用 `HttpResponse.BodyHandlers.ofInputStream()` 获取一个 `InputStream` 流。

要使用 `POST` 请求，我们要准备好发送的 `Body` 数据并正确设置 `Content-Type`：

```
String url = "http://www.example.com/login";  
String body = "username=bob&password=123456";  
HttpRequest request = HttpRequest.newBuilder(new URI(url))  
    // 设置Header:  
    .header("Accept", "*/.*")  
    .header("Content-Type", "application/x-www-form-urlencoded")  
    // 设置超时:  
    .timeout(Duration.ofSeconds(5))  
    // 设置版本:  
    .version(Version.HTTP_2)  
    // 使用POST并设置Body:  
    .POST(BodyPublishers.ofString(body, StandardCharsets.UTF_8)).build();  
HttpResponse<String> response = httpClient.send(request,  
    HttpResponse.BodyHandlers.ofString());  
String s = response.body();
```

可见发送 `POST` 数据也十分简单。

## 练习

下载练习：[使用HttpClient](#)（推荐使用[IDE练习插件](#)快速下载）

## 小结

- Java提供了 `HttpClient` 作为新的HTTP客户端编程接口用于取代老的 `HttpURLConnection` 接口；
- `HttpClient` 使用链式调用并通过内置的 `BodyPublishers` 和 `BodyHandlers` 来更方便地处理数据。

## RMI远程调用

Java的RMI远程调用是指，一个JVM中的代码可以通过网络实现远程调用另一个JVM的某个方法。RMI是 Remote Method Invocation的缩写。

提供服务的一方我们称之为服务器，而实现远程调用的一方我们称之为客户端。

我们先来实现一个最简单的RMI：服务器会提供一个 `worldClock` 服务，允许客户端获取指定时区的时间，即允许客户端调用下面的方法：

```
LocalDateTime getLocalDateTime(String zoneId);
```

要实现RMI，服务器和客户端必须共享同一个接口。我们定义一个 `worldClock` 接口，代码如下：

```
public interface worldClock extends Remote {  
    LocalDateTime getLocalDateTime(String zoneId) throws RemoteException;  
}
```

Java的RMI规定此接口必须派生自 `java.rmi.Remote`，并在每个方法声明抛出 `RemoteException`。

下一步是编写服务器的实现类，因为客户端请求的调用方法 `getLocalDateTime()` 最终会通过这个实现类返回结果。实现类 `WorldClockService` 代码如下：

```
public class WorldClockService implements WorldClock {
    @Override
    public LocalDateTime getLocalDateTime(String zoneId) throws RemoteException
    {
        return LocalDateTime.now(ZoneId.of(zoneId)).withNano(0);
    }
}
```

现在，服务器端的服务相关代码就编写完毕。我们需要通过Java RMI提供的一系列底层支持接口，把上面编写的服务以RMI的形式暴露在网络上，客户端才能调用：

```
public class Server {
    public static void main(String[] args) throws RemoteException {
        System.out.println("create world clock remote service...");
        // 实例化一个WorldClock:
        WorldClock worldClock = new WorldClockService();
        // 将此服务转换为远程服务接口:
        WorldClock skeleton = (WorldClock)
        UnicastRemoteObject.exportObject(worldClock, 0);
        // 将RMI服务注册到1099端口:
        Registry registry = LocateRegistry.createRegistry(1099);
        // 注册此服务，服务名为"WorldClock":
        registry.rebind("WorldClock", skeleton);
    }
}
```

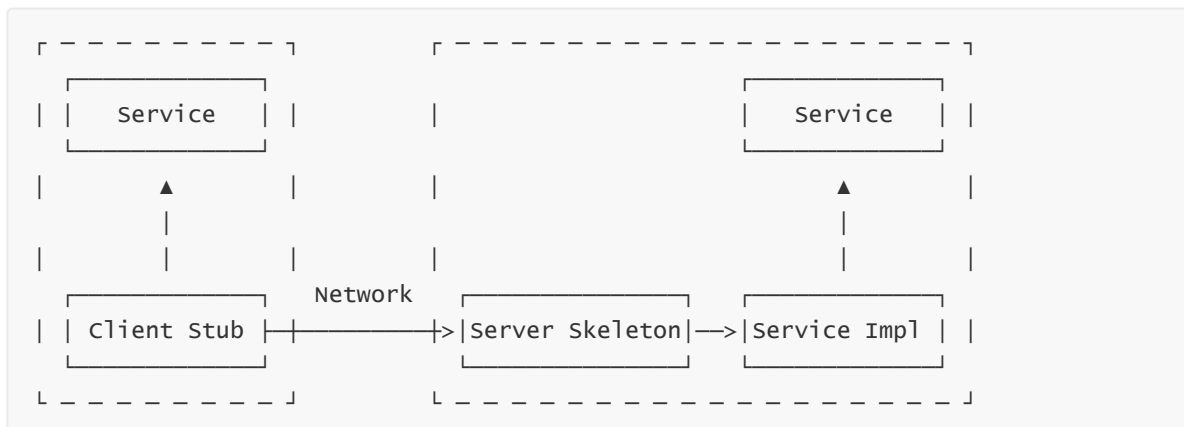
上述代码主要目的是通过RMI提供的相关类，将我们自己的 `WorldClock` 实例注册到RMI服务上。RMI的默认端口是 `1099`，最后一步注册服务时通过 `rebind()` 指定服务名称为 `"WorldClock"`。

下一步我们就可以编写客户端代码。RMI要求服务器和客户端共享同一个接口，因此我们要把 `WorldClock.java` 这个接口文件复制到客户端，然后在客户端实现RMI调用：

```
public class Client {
    public static void main(String[] args) throws RemoteException,
    NotBoundException {
        // 连接到服务器localhost，端口1099:
        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        // 查找名称为"WorldClock"的服务并强制转型为WorldClock接口:
        WorldClock worldClock = (WorldClock) registry.lookup("WorldClock");
        // 正常调用接口方法:
        LocalDateTime now = worldClock.getLocalDateTime("Asia/Shanghai");
        // 打印调用结果:
        System.out.println(now);
    }
}
```

先运行服务器，再运行客户端。从运行结果可知，因为客户端只有接口，并没有实现类，因此，客户端获得的接口方法返回值实际上是通过网络从服务器端获取的。整个过程实际上非常简单，对客户端来说，客户端持有的 `WorldClock` 接口实际上对应了一个“实现类”，它是由 `Registry` 内部动态生成的，并负责把方法调用通过网络传递到服务器端。而服务器端接收网络调用的服务并不是我们自己编写的

`WorldClockService`，而是 Registry 自动生成的代码。我们把客户端的“实现类”称为 `stub`，而服务器端的网络服务类称为 `skeleton`，它会真正调用服务器端的 `WorldClockService`，获取结果，然后把结果通过网络传递给客户端。整个过程由 RMI 底层负责实现序列化和反序列化：



Java的RMI严重依赖序列化和反序列化，而这种情况下可能会造成严重的安全漏洞，因为Java的序列化和反序列化不但涉及到数据，还涉及到二进制的字节码，即使使用白名单机制也很难保证100%排除恶意构造的字节码。因此，使用RMI时，双方必须是内网互相信任的机器，不要把1099端口暴露在公网上作为对外服务。

此外，Java的RMI调用机制决定了双方必须是Java程序，其他语言很难调用Java的RMI。如果要使用不同语言进行RPC调用，可以选择更通用的协议，例如[gRPC](#)。

## 练习

下载练习：[使用RMI远程调用](#)（推荐使用[IDE练习插件](#)快速下载）

## 小结

- Java提供了RMI实现远程方法调用：
- RMI通过自动生成`stub`和`skeleton`实现网络调用，客户端只需要查找服务并获得接口实例，服务器端只需要编写实现类并注册为服务；
- RMI的序列化和反序列化可能会造成安全漏洞，因此调用双方必须是内网互相信任的机器，不要把1099端口暴露在公网上作为对外服务。