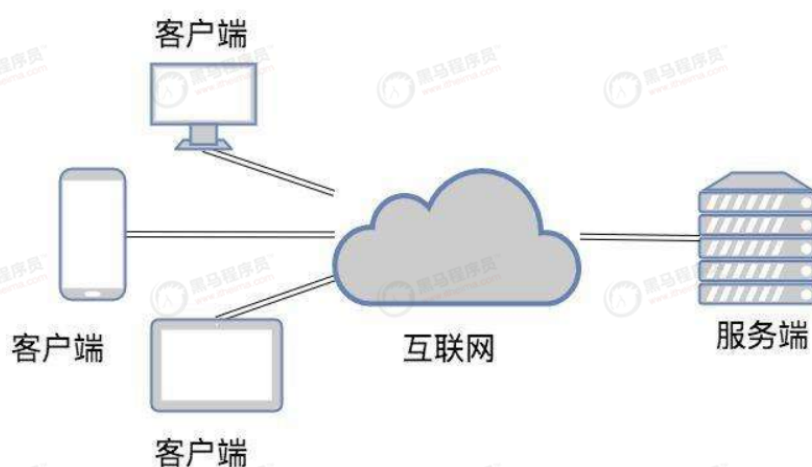


1 网络编程概述

1.1 什么是网络编程

网络编程就是直接或间接地通过**网络协议**与其他计算机进行通信，网络编程有时我们也将之称之为**套接字**(Socket)编程。



1.2 网络编程三要素

1、ip地址

2、端口号

3、网络协议

1.2.1 IP地址

IP地址是分配给网络中每台机器的数字标识符，它指出了设备在网络中的具体位置。

ip地址可以分为两个版本：ipv4以及ipv6



1) ipv4地址

IPv4地址（二进制）

```
01100100.00000100.00000101.00000110
```

IP地址（十进制）

```
100.4.5.6
```

2) ipv6地址

IPv6具有比IPv4大得多的地址空间。

这是因为IPv6采用128位的地址，而IPv4使用的是32位（4倍）。

因此

ipv6版的ip地址支持 $2^{128}(3.4 * 10^{38})$ 个ip地址

IPv6二进制制下为128位长度，以16位为一组，每组以冒号":"隔开，可以分为8组，每组以4位十六进制方式表示。例如：

```
2001:0db8:85a3:08d3:1319:8a2e:0370:7344
```

以上示例是一个合法的IPv6地址。

类似于IPv4的点分十进制，同样也存在点分十六进制的写法，将8组4位十六进制地址的冒号去除后，每位以点号"."分组，例如：

```
2001:0db8:85a3:08d3:1319:8a2e:0370:7344
```

可以标记为

```
2.0.0.1.0.d.b.8.8.5.a.3.0.8.d.3.1.3.1.9.8.a.2.e.0.3.7.0.7.3.4.4
```

同时IPv6在某些条件下可以省略

1. 每项数字前导的0可以省略，省略后前导数字仍是0则继续，例如下组IPv6是等价的。

```
2001:0DB8:02de:0000:0000:0000:0000:0e13
```

```
2001:DB8:2de:000:000:000:000:e13
```

```
2001:DB8:2de:00:00:00:00:e13
```

```
2001:DB8:2de:0:0:0:0:e13
```

- 2、可以用双冒号 "::" 表示一组0或多组连续的0，但只能出现一次：

如果四组数字都是零，可以被省略。遵照以上省略规则，下面这两组IPv6都是相等的(使用 "::" 表示多组连续的0)。

2001:DB8:2de:0:0:0:e13

2001:DB8:2de::e13

下面这两组IPv6都是相等的(使用 "::" 表示一组的0)

2001:0DB8:0000:0000:0000:0000:1428:57ab

2001:0DB8:0000:0000:0000::1428:57ab

2001::25de::cade 是非法的, 因为双冒号出现了两次。它有可能是下种情形之一, 造成无法推断。

2001:0000:0000:0000:0000:25de:0000:cade

2001:0000:0000:0000:25de:0000:0000:cade

2001:0000:0000:25de:0000:0000:0000:cade

2001:0000:25de:0000:0000:0000:0000:cade

查看ipv6的ip地址

通过ipconfig查看本机的ipv6地址

```
C:\Users\Administrator>ipconfig
```

Windows IP 配置

以太网适配器 以太网:

连接特定的 DNS 后缀 :

本地链接 IPv6 地址. : fe80::24cf:c587:c97d:83de%20 ----->>>

ipv6的ip地址

IPv4 地址 : 172.16.43.178

子网掩码 : 255.255.192.0

默认网关. : 172.16.0.1

无线局域网适配器 WLAN:

媒体状态 : 媒体已断开连接

连接特定的 DNS 后缀 : lan

```
C:\Users\Administrator>
```

通过ipconfig我们可以查看到ipv6对应的ip地址, 在该地址后面存在一个%20, %20表示的意思该地址仅限于标号为20的网卡接口。而在其他网卡接口这个地址是无效的。

总结:

长度 2^{128} , 约等于 $(3.4 * 10^{38})$ 个ip地址

ipv6分为8组, 每组4位16进制表示, 以冒号":"隔开, 总长度128位

点分十六进制的写法, 全部以 "." 隔开

前导为0可以省略; 依次类推, 为0继续省略

用双冒号 "::" 表示一组0或多组连续的0, 但只能出现一次

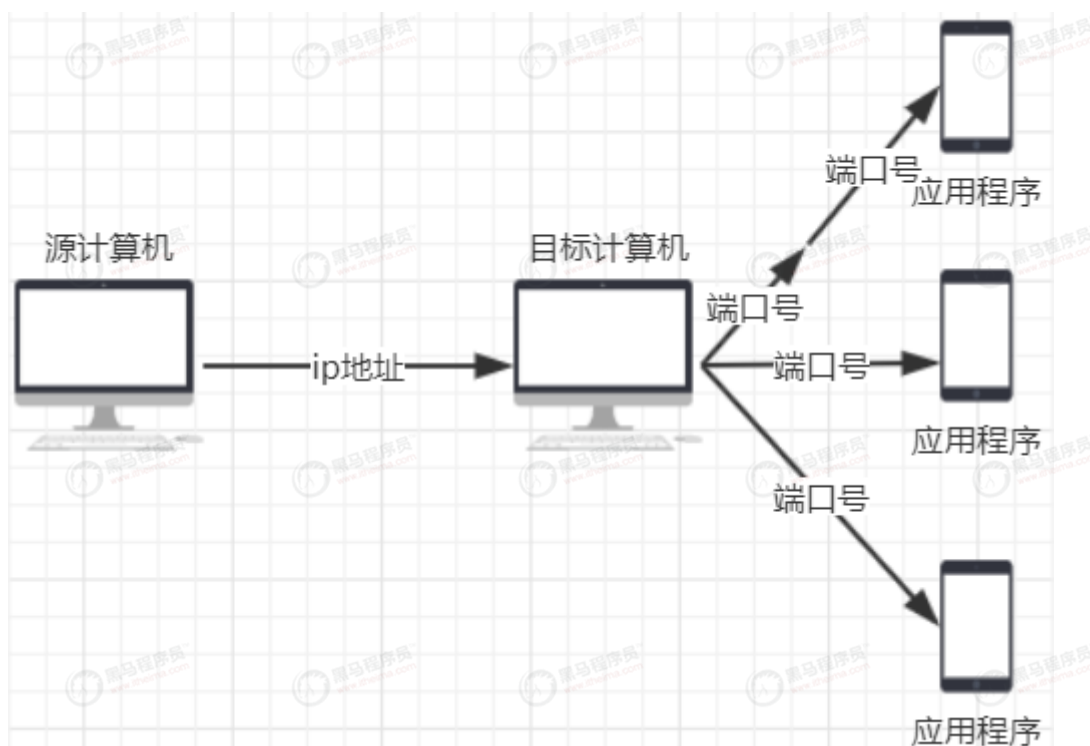
目前我们使用的较多的还是ipv4版的ip地址。

1.2.2 端口号

通过IP地址可以在网络中找到指定计算机，但如果想访问目标计算机中的某个应用程序，还需要指定端口号。

在计算机中，不同的应用程序是通过端口号区分的。端口号是用两个字节（16位的二进制数）表示的，它的取值范围是0~65535，其中，0~1023之间的端口号用于一些知名的网络服务和应用(一般都是系统服务和应用)，用户的普通应用程序需要使用1024以上的端口号，

从而避免端口号被另外一个应用或服务所占用。



总结

取值范围是0~65535

0~1023之间基本为系统端口

1024~65535 用户的普通应用程序需要使用1024以上的端口号

1.2.3 网络协议

网络协议 🔒 锁定

📖 本词条由“科普中国”科学百科词条编写与应用工作项目 审核。

网络协议为**计算机网络**中进行数据交换而建立的规则、标准或约定的集合。

例如，网络中一个微机用户和一个大型**主机**的操作员进行通信，由于这两个数据终端所用**字符集**不同，因此操作员所输入的命令彼此不认识。为了能进行通信，规定每个终端都要将各自字符集中的字符先变换为标准字符集的字符后，才进入网络传送，到达目的终端之后，再变换为该终端字符集的字符。当然，对于不相容终端，除了需变换字符集字符外还需转换其他特性，如显示格式、行长、行数、**屏幕滚动方式**等也需作相应的变换。 [1]

中文名	网络协议	性 质	网络术语
外文名	Network Protocol	目 的	使两个进程相互通信
		解 释	描述进程之间 信息交换 数据

1.3 网络编程应用场景



应用场景

(IM,即时通讯)：QQ、微信、企业微信、钉钉
通信框架、文件操作、消息推送等等

2 网络分层模型通信解读与实战

2.1 大促背后的网络分层原理

客户端访问



A-B站点进行数据传输过程



2.2 网络分层模型与协议族

OSI（Open System Interconnect），即[开放式系统](#)互联。一般都叫OSI参考模型，是[ISO](#)国际标准化组织在1985年研究的[网络互联](#)模型。该[体系结构](#)标准定义了网络互联的七层框架（[物理层](#)、[数据链路层](#)、[网络层](#)、[传输层](#)、[会话层](#)、[表示层](#)和[应用层](#)），即OSI[开放系统互连参考模型](#)。

TCP/IP协议族里的协议最早发源于美国国防部（缩写为DoD）的ARPA网项目，因此也被称作DoD模型（DoD Model）。

OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	HTTP、TFTP, FTP, NFS, WAIS
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

设备：

应用层----->网关

传输层----->四层交换机

网络层----->路由器/三层交换机

数据链路层----->网桥、集线器（物理层）中继器（物理层），光纤、双绞线

TCP/IP四层模型名称：

网络层：协议族也可以叫做【网际互联层】

物理链路层：协议族也可以叫做【网络接口层】

OSI七层模型是理论上的分层方式，而四层模型是实践过程中的分层模型

ip协议——>网络互连协议

用途：将多个包在网络中联系起来，传输数据包（不可靠传输），最基本功能就是寻址和分段功能，不提供端到端，路由到路由的确认，不提供重发和流量控制。是计算机网络能够互相通信的基本规则。出错则像ICMP报告，ICMP在IP模块中实现。

ICMP协议——面向无连接协议

用途：用户传输错误报告控制信息（控制信息是指网络不通畅，主机是否到达，路由是否可用的这些网络本身的消息，不涉及用户传输的数据）。

ARP协议——地址解析协议

用途：根据IP地址获取物理地址的协议（即MAC地址）。在同一子网内通过ARP协议可以实现数据包的互相传递。不在一个子网内则无法获得MAC地址，只有通过网关去处理。

RAPP协议——反转地址协议

用途：将主机的物理地址转换成IP地址。

BOOTP协议——引导程序协议

用途：用于无盘工作站的局域网中，可以无盘工作站从一个中心服务器上获得IP地址。

传输层：提供两台主机间端到端的通信

TCP协议——传输控制协议

用途：主要用于网间传输的协议，分割处理报文并把结果包传到IP层，并接受处理IP曾传到的数据包。

UDP——用户数据协议

用途：主要用于需要在计算器之间传输数据的应用，将网络数据流量压缩成数据包。

应用层：用于不同的应用程序

NET协议——网络地址转换协议

用途：实现内网IP地址和公司地址之间的相互转换。将大量的内网IP转换成一个或者少量的公网IP。

FTP协议——文件传输协议

用途：通过FTP协议在FTP客户端访问FTP服务端，默认使用20和21端口，20用于传输数据，21用于传输控制信息。

HTTP协议——超文本协议

用途：是用于从WWW服务端传输超文本到本地浏览器的传输协议。是客户端浏览器或其他程序与WEB服务器之间的应用层通信协议。

TELNET协议

用途：是Internet远程登录服务的标准协议和主要方式，为用户提供了在本地计算机上完成远程主机工作的能力。

SMTP——简单邮件传输协议

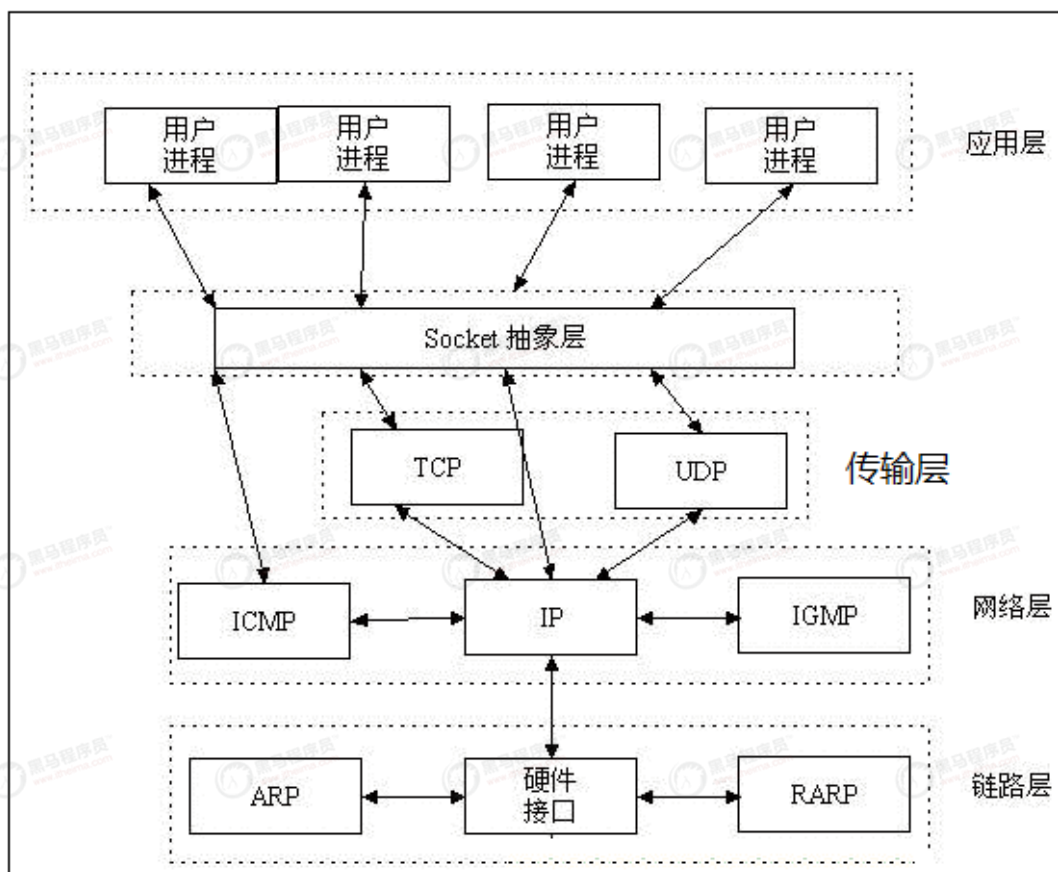
用途：控制邮件传输的规则，以及邮件的中转方式。

DNS协议

用途：定义域名规则，将域名和IP相互映射。

2.3 大促背后的网络分层解析

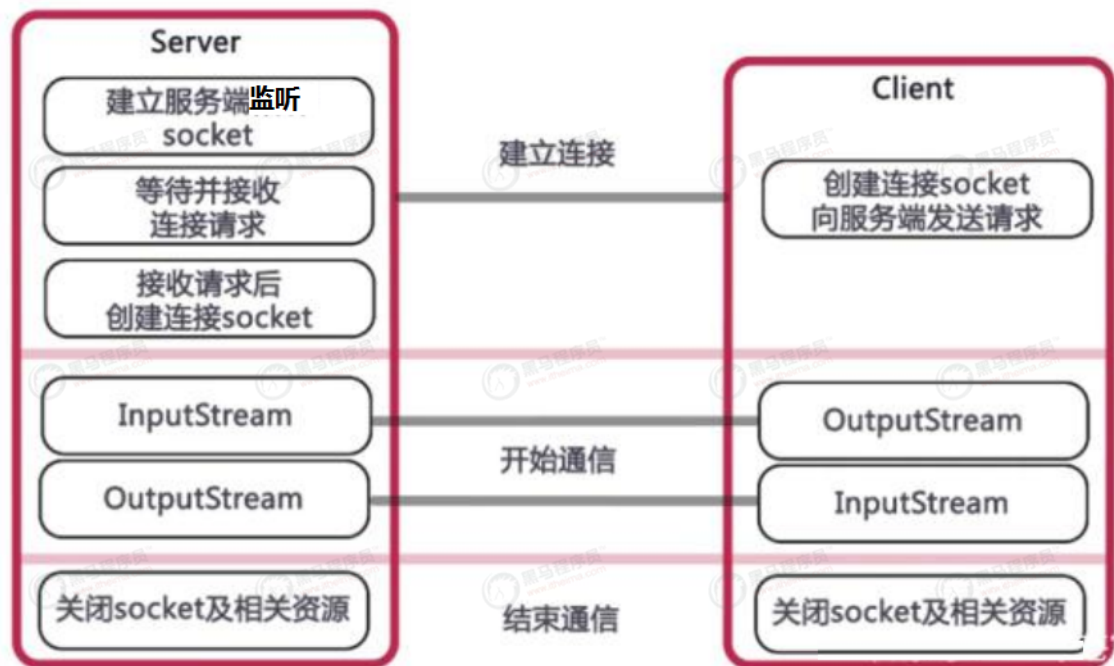




Socket其实并不是一个协议，而是为了方便使用TCP或UDP而抽象出来的一层，是位于应用层和传输控制层之间的一组接口
提供一套调用TCP/IP协议的API。

在设计模式中，**Socket**其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在**Socket**接口后面，对用户来说，一组简单的接口就是全部，让**Socket**去组织数据，以符合指定的协议。
当两台主机通信时，必须通过**Socket**连接，**Socket**则利用TCP/IP协议建立TCP连接。TCP连接则更依赖于底层的IP协议，IP协议的连接则依赖于链路层等更低层次

2.4.2 Socket通信流程

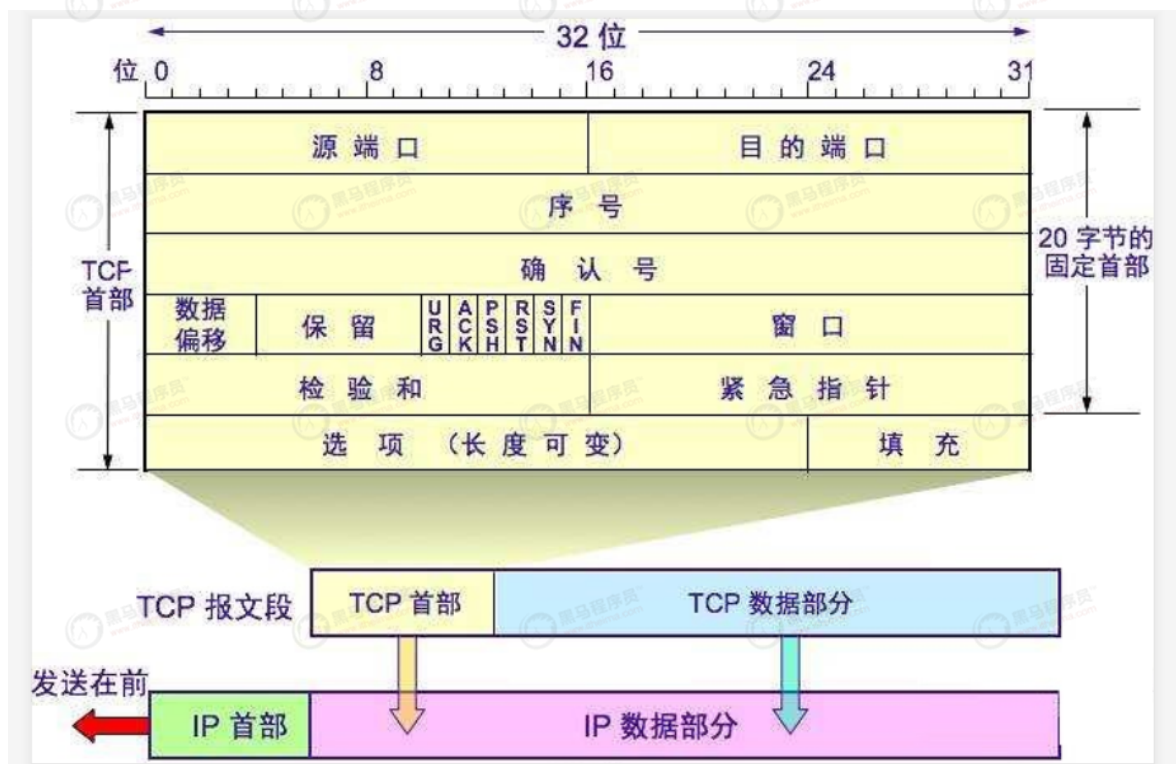


2.4.3 网络传输报头格式

TCP是Transmission Control Protocol的简称,中文名称为传输控制协议。它具有以下几个特点:

1. TCP协议是一种面向连接的协议。
2. TCP协议是一种可靠的协议。
3. TCP协议是基于IO流进行数据传输,传输数据无大小限制。
4. TCP协议的它的通信效率比较低。

- 1- 数据传输之前必须先建立连接,数据传输完成之后,必须释放连接
- 2- 传送的数据无差错,不丢失,不重复,且顺序与与源数据一致
- 3- 将数据拆分成不同大小的的段,也就是segment进行传输
- 4- TCP是面向连接的协议,传输数据之前需要建立连接,因此它的通信效率比较低。



我们来分析分析每部分的含义和作用

- 源端口号/目的端口号：表示数据从哪个进程来, 到哪个进程去。
- 序号和确认号：是TCP可靠传输的关键部分。**序号**是本报文段发送的数据组的第一个字节的序号。在TCP传送的流中，每一个字节一个序号：

TCP首部长度：由于TCP首部包含一个长度可变的选项部分，所以需要这么一个值来指定这个TCP报文段到底有多长。

URG：表示本报文段中发送的数据是否包含紧急数据。URG=1，表示有紧急数据。后面的紧急指针字段只有当URG=1时才有效。

ACK：表示是否前面的确认号字段是否有效。ACK=1，表示有效。只有当ACK=1时，前面的确认号字段才有效。TCP规定，连接建立后，ACK必须为1。

PSH：告诉对方收到该报文段后是否应该立即把数据推送给上层。如果为1，则表示对方应当立即把数据提交给上层，而不是缓存起来。

RST：只有当RST=1时才有用。如果你收到一个RST=1的报文，说明你与主机的连接出现了严重错误（如主机崩溃），必须释放连接，然后再重新建立连接。或者说明你上次发送给主机的数据有问题，主机拒绝响应。

SYN：在建立连接时使用，用来同步序号。当SYN=1，ACK=0时，表示这是一个请求建立连接的报文段；当SYN=1，ACK=1时，表示对方同意建立连接。SYN=1，说明这是一个请求建立连接或同意建立连接的报文。只有在前两次握手中SYN才置为1。

FIN：标记数据是否发送完毕。如果FIN=1，就相当于告诉对方：“我的数据已经发送完毕，你可以释放连接了”

- 窗口：滑动窗口大小，用来告知发送端接受端的缓存大小，以此控制发送端发送数据的速率，从而达到流量控制：

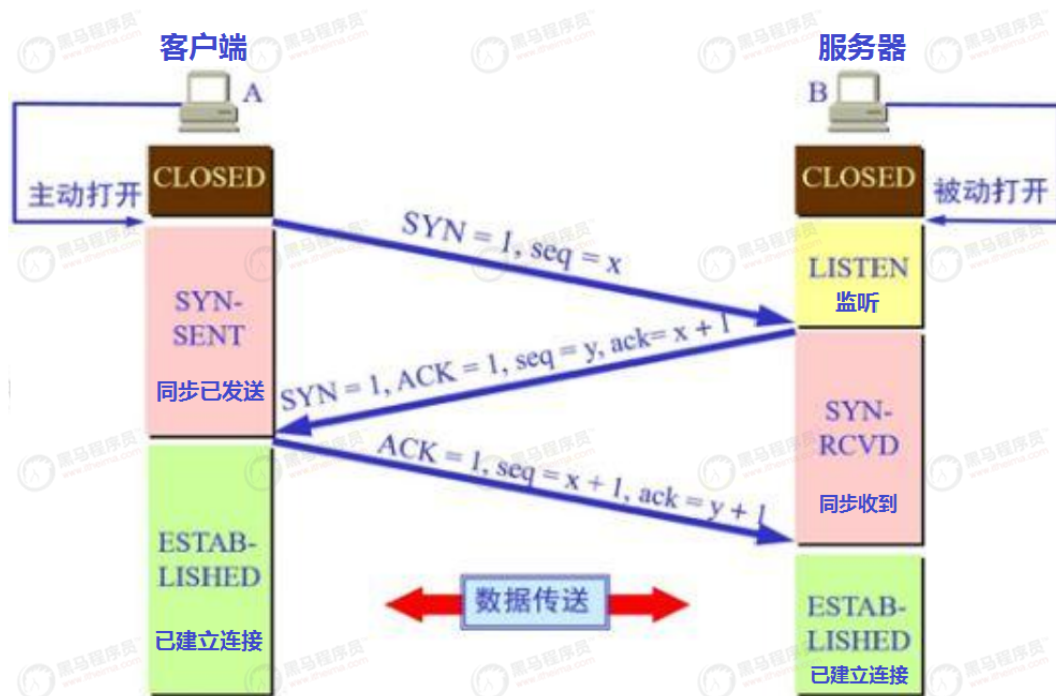
选项和填充：最常见的可选字段是最长报文大小，又称为MSS（Maximum Segment Size），每个连接方通常都在通信的第一个报文段（为建立连接而设置SYN标志为1的那个段）中指明这个选项，它表示本端所能接受的最大报文段的长度。选项长度不一定是32位的整数倍，所以要加填充位，即在这个字段中加入额外的零，以保证TCP头是32的整数倍

数据部分： *TCP* 报文段中的数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有 *TCP* 首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

2.4.4 TCP连接断开原理剖析

(1) TCP连接三次握手

TCP 协议在双方建立连接的时候需要三次握手，所谓的三次握手即 *TCP* 连接的建立。这个连接必须是一方主动打开，另一方被动打开的。以下为客户端主动发起连接的图解：



其中比较重要的字段有：

SYN(synchronous建立连接) ACK(acknowledgement 确认) PSH(push传送) FIN(finish结束)
RST(reset重置) URG(urgent紧急)

字段	含义
URG	紧急指针是否有效。为1，表示某一位需要被优先处理
ACK	确认号是否有效，一般置为1。
PSH	提示接收端应用程序立即从TCP缓冲区把数据读走。
RST	对方要求重新建立连接，复位。
SYN	请求建立连接，并在其序列号的字段进行序列号的初始值设定。建立连接，设置为1
FIN	希望断开连接。

tips

不要将确认序号Ack与标志位中的ACK搞混了。确认方Ack=发起方Seq+1，两端配对

为什么要进行第三次握手？

为了防止服务器端开启一些无用的连接增加服务器开销以及防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

(2) TCP的四次挥手

所谓的四次挥手即TCP连接的释放(解除)。连接的释放必须是一方主动释放，另一方被动释放。以下为客户端主动发起释放连接的图解：



其中比较重要的字段有：

SYN(synchronous建立连接) ACK(acknowledgement 确认) PSH(push传送) FIN(finish结束)
RST(reset重置) URG(urgent紧急)

字段	含义
URG	紧急指针是否有效。为1，表示某一位需要被优先处理
ACK	确认号是否有效，一般置为1。
PSH	提示接收端应用程序立即从TCP缓冲区把数据读走。
RST	对方要求重新建立连接，复位。
SYN	请求建立连接，并在其序列号的字段进行序列号的初始值设定。建立连接，设置为1
FIN	希望断开连接。

为什么客户端在TIME-WAIT阶段要等2MSL

MSL指的是Maximum Segment Lifetime：一段TCP报文在传输过程中的最大生命周期。2MSL即是服务器端发出为FIN报文和客户端发出的ACK确认报文所能保持有效的最大时长

2.5 Socket编程与实战

2.5.1 从传统网络编程看BIO并发问题

1、传统网络编程正常消息发送

JDK提供的ServerSocket类表示我们的服务端，使用该类可以接收客户端的连接请求进而实现两端的通信。

ServerSocket常用方法

```
public ServerSocket() // 无参构造方,系统自动分配一个可用端口
(匿名端口),服务端一般不使用
public ServerSocket(int port) // 构造方法,用于创建服务端对象接收客户
端连接(默认连接数为50); port指定服务端程序所占用的端口
public ServerSocket(int port, int backlog) // 构造方法,用于创建服务端对象; port
指定服务端程序所占用的端口,backlog指定可接收的客户端数量
public Socket accept() // 接收客户端连接请求(监听客户端),如
果没有客户端连接该方法阻塞。
```

服务端代码实现步骤:

1. 创建ServerSocket对象,用于客户端连接
2. 调用accept方法(监听客户端),该方法是一个阻塞方法
3. 调用Socket对象的方法获取输入流对象
4. 使用输入流对象读取数据
5. 释放资源

```
public static void main(String[] args) throws Exception {
    // 创建ServerSocket对象,用于客户端连接
    ServerSocket serverSocket = new ServerSocket(8080);
    // 使用输入流对象读取数据
    byte[] bytes = new byte[1024];
    try {
        while (true) {
            // 调用accept方法(监听客户端),该方法是一个阻塞方法
            Socket accept = serverSocket.accept();
            // 调用Socket对象的方法获取输入流对象
            InputStream inputStream = accept.getInputStream();
            int read = inputStream.read(bytes);
            System.out.println(new String(bytes, 0, read));
            //关闭资源
            accept.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (serverSocket != null && !serverSocket.isClosed()) {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }

}

}

```

客户端

JDK为了TCP通信提供了两个核心类：Socket(客户端)，ServerSocket(服务端)。使用Socket可以与服务端建立连接，实现数据的发送与接收。

Socket中的常用方法

<code>public Socket(String host, int port)</code>	// 构造方法，用于和服务端建立连接
接;address: 表示要连接的主机地址，port表示端口号	
<code>public OutputStream getOutputStream()</code>	// 获取输出流对象，写数据
<code>public InputStream getInputStream()</code>	// 获取输入流对象，读数据
<code>public void close()</code>	// 释放资源

客户端代码实现步骤：

1. 创建Socket对象，与服务端建立连接
2. 获取输出流对象
3. 使用输出流对象写数据
4. 释放资源

代码实现

```

public class SocketClient {
    public static void main(String[] args) throws Exception {
        // 创建Socket对象，与服务端建立连接
        Socket socket = new Socket("127.0.0.1", 8080);
        // 获取输出流对象
        OutputStream outputStream = socket.getOutputStream();
        // 使用输出流对象写数据
        outputStream.write("itheima-TCP".getBytes());
        // 释放资源
        socket.close();
    }
}

```

案例抓包剖析：

1、三次握手

TCP协议是面向连接的通信协议，即传输数据之前，在客户端和服务端需要建立逻辑连接，然后再传输数据。

客户端与服务端在进行连接建立的时

候，需要经过三个步骤,这个3个步骤相当于3次握手。

1. 客户端到服务端: 我要连接

2. 服务端到客户端: 好的, 已经连接上了
3. 客户端到服务端: 收到, 确认已连接上了

我们可以借助于一个工具wireshark来抓取TCP客户端与服务端建立连接时数据传输的过程。(wireshark工具的安装以及使用参考<<wireshark安装与使用文档>>)

客户端代码

```
// 创建Socket对象
Socket socket = new Socket("127.0.0.1", 8080);

// 让线程休眠60s
TimeUnit.SECONDS.sleep(60);
```

服务端代码

```
// 创建ServerSocket对象
ServerSocket serverSocket = new ServerSocket(9999) ;

// 让线程休眠60s
TimeUnit.SECONDS.sleep(60);
```

首先运行服务端, 然后在运行客户端, 在wireshark工具中捕获完整的通信过程, 结果如下图所示:

```
ip.src == 127.0.0.1 and tcp.port==8080
```

No.	Time	Source	Destination	Protocol	Length	Info
406	27.739830	127.0.0.1	127.0.0.1	TCP	56	4723 → 8080 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
407	27.739886	127.0.0.1	127.0.0.1	TCP	56	8080 → 4723 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
408	27.740147	127.0.0.1	127.0.0.1	TCP	44	4723 → 8080 [ACK] Seq=1 Ack=1 Win=8192 Len=0

第一次"握手"如下图所示

406	27.739830	127.0.0.1	127.0.0.1	TCP	56	4723 → 8080 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
-----	-----------	-----------	-----------	-----	----	---

在第一次"握手"时, 客户端向服务端发送SYN标志位, 目的是与服务端建立连接。Seq代表sequence number(发送数据流序号), 例如: Seq的值是5, 说明在数据流中曾经一共发送了 1, 2, 3, 4 这4次数据。而在本次"握手"中, Seq的值是0, 代表客户端曾经没有给服务端发送数据。另外Len=0也可以看出来是没有数据可供发送的, 客户端仅仅发送一个SYN标志位到服务端代表要进行连接。

第二次"握手"如下图所示

407	27.739886	127.0.0.1	127.0.0.1	TCP	56	8080 → 4723 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
-----	-----------	-----------	-----------	-----	----	--

第二次"握手"时, 服务端向客户端发送 SYN ACK 标志位, 其中ACK标志位表示是对收到的数据包的确认, 说明服务端接收到了客户端的连接。ACK的值是1, 表示服务端期待下一次从客户端发送数据流的序列号是1, 而Seq=0代表服务端曾经并没有给客户端发送数据, 而本次也没有发送数据, 因为Len=0也证明了这一点。

第三次"握手"如下图所示

408 27.740147 127.0.0.1 127.0.0.1 TCP 44 4723 → 8080 [ACK] Seq=1 Ack=1 Win=8192 Len=0

第三次“握手”时，客户端向服务端发送的ACK标志位为1，Seq的值是1。Seq=1代表这正是服务端所期望的Ack=1。Len=0说明客户端这次还是没有向服务端传递数据，而客户端向服务端发送ACK标志位为1的信息，说明客户端期待服务端下一次传送的Seq的值是1。

2、四次挥手

客户端与服务端在断开连接的时候需要进行4次"挥手"，4次"挥手"的过程如下：

1. 客户端到服务端：我关了
2. 服务端到客户端：好的，收到
3. 服务端到客户端：我也关了
4. 客户端到服务端：好的，收到

我们也可以借助于wireshark这个抓包工具来抓取连接断开的整个数据传输的过程。

```
ip.src == 127.0.0.1 and tcp.port==8080
```

客户端代码

```
// 创建Socket对象
Socket socket = new Socket("127.0.0.1", 8080);
// 线程休眠
TimeUnit.SECONDS.sleep(10);

// 关闭连接
socket.close();
```

服务端代码

```
// 创建ServerSocket对象
ServerSocket serverSocket = new ServerSocket(8080);

// 获取连接对象
Socket socket = serverSocket.accept();

// 线程休眠
TimeUnit.SECONDS.sleep(10);

// 释放资源
socket.close();
serverSocket.close();
```

首先运行服务端，然后在运行客户端，在wireshark工具中捕获完整的通信过程，结果如下图所示：

135	31.532680	169.254.170.6	169.254.170.6	TCP	44 4825 → 9999 [FIN, ACK] Seq=1 Ack=1 Win=2619648 Len=0
136	31.532852	169.254.170.6	169.254.170.6	TCP	44 9999 → 4825 [ACK] Seq=1 Ack=2 Win=2619648 Len=0
137	31.533448	169.254.170.6	169.254.170.6	TCP	44 9999 → 4825 [FIN, ACK] Seq=1 Ack=2 Win=2619648 Len=0
138	31.533615	169.254.170.6	169.254.170.6	TCP	44 4825 → 9999 [ACK] Seq=2 Ack=2 Win=2619648 Len=0

353 59.028321	127.0.0.1	127.0.0.1	TCP	44 1464 → 8080 [FIN, ACK] Seq=1 Ack=1 Win=8192 Len=0
354 59.028391	127.0.0.1	127.0.0.1	TCP	44 8080 → 1464 [ACK] Seq=1 Ack=2 Win=8192 Len=0
355 59.031519	127.0.0.1	127.0.0.1	TCP	44 8080 → 1464 [FIN, ACK] Seq=1 Ack=2 Win=8192 Len=0
356 59.031587	127.0.0.1	127.0.0.1	TCP	44 1464 → 8080 [ACK] Seq=2 Ack=2 Win=8192 Len=0

第一次"挥手"如下图所示

353 59.028321	127.0.0.1	127.0.0.1	TCP	44 1464 → 8080 [FIN, ACK] Seq=1 Ack=1 Win=8192 Len=0
---------------	-----------	-----------	-----	--

在第一次"挥手"时,客户端向服务器发送标志位FIN ACK,告知服务端客户端关闭了。Seq=1表示本次数据流的序号为1,Ack=1表示客户端期望服务端下一次发送的数据流的序号为1。len=0,

说明没有数据传输到服务端。

第二次"挥手"如下图所示

354 59.028391	127.0.0.1	127.0.0.1	TCP	44 8080 → 1464 [ACK] Seq=1 Ack=2 Win=8192 Len=0
---------------	-----------	-----------	-----	---

在第二次"挥手"时,服务端向客户端发送标志位ACK,Seq=1代表的正是客户端想看的Ack=1。Ack=2表示服务端期望下一次客户端发送的数据流的序号为2。len=0,说明没有数据传输到客户端。

第三次"挥手"如下图所示

355 59.031519	127.0.0.1	127.0.0.1	TCP	44 8080 → 1464 [FIN, ACK] Seq=1 Ack=2 Win=8192 Len=0
---------------	-----------	-----------	-----	--

在第三次"挥手"时,服务端向客户端发送标志位FIN ACK,告知客户端服务端关闭了。Seq=1代表的正是客户端想看的Ack=1。Ack=2表示服务端期望下一次客户端发送的数据流的序号为2。

len=0,说明没有数据传输到客户端。

第四次"挥手"如下图所示

356 59.031587	127.0.0.1	127.0.0.1	TCP	44 1464 → 8080 [ACK] Seq=2 Ack=2 Win=8192 Len=0
---------------	-----------	-----------	-----	---

在第四次"挥手"时,客户端向服务端发送标志位ACK,告知服务端客户端已经收到服务端关闭信息。Seq=2代表的正是服务端想看的Ack=2,ACK=2表示客户端期望下一次服务端发送的数据流的序号为2。

2、传统网络编程延迟消息发送

服务端

```
public static void main(String[] args) throws Exception {
    // 创建ServerSocket对象,用于客户端连接
    ServerSocket serverSocket = new ServerSocket(8080);
    // 使用输入流对象读取数据
    byte[] bytes = new byte[1024];
    try {
        while (true) {
            System.out.println("发生阻塞,等待客户端连接.....");
            // 调用accept方法(监听客户端),该方法是一个阻塞方法
            Socket accept = serverSocket.accept();
            // 调用Socket对象的方法获取输入流对象
            InputStream inputStream = accept.getInputStream();
            System.out.println("发生阻塞,等待客户端传输数据.....");
            int read = inputStream.read(bytes);
            System.out.println(new String(bytes, 0, read));
            //关闭资源
            accept.close();
        }
    }
}
```

```

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (serverSocket != null && !serverSocket.isClosed()) {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

}

```

客户端

```

public class SocketClient {
    public static void main(String[] args) throws Exception {
        // 创建Socket对象，与服务端建立连接
        Socket socket = new Socket("127.0.0.1", 8080);
        // 获取输出流对象
        OutputStream outputStream = socket.getOutputStream();
        System.out.println("客户端阻塞, 接受键盘录入...");
        // 接受键盘输入，模拟延迟发送消息
        Scanner scanner = new Scanner(System.in);
        String scannerString = scanner.next();
        // 使用输出流对象写数据
        outputStream.write(scannerString.getBytes());
        System.out.println("客户端完成录入...");
        // 释放资源
        socket.close();
    }
}

```

存在的问题：

- 1、客户端已经连接服务端，尚未发送数据，read阻塞
- 2、新的客户端无法正常连接

如何解决？

- 1、线程解决（mysql客户端连接服务器）
- 2、线程池解决（线程池泄露）
- 3、NIO解决
- 4、websocket解决（第二天详细讲解）

2.5.2 基于NIO多路复用解决并发问题

1) NIO知识点回顾

特点:

1. NIO是New IO的简称，是JDK1.4提出的一种新的IO模型，NIO所提供的类是在java.nio包中。
2. JAVA API中提供了两套NIO：一套是针对文件操作的NIO，另一套就是网络编程的NIO
3. NIO是一种非阻塞式IO

核心的部分

JAVA NIO中最为核心的部分：**Buffer(缓冲区)**，**Channel(通道)**。在网络编程的NIO模型中，还有一个核心部分：**Selector (选择器)**

1、Buffer(缓冲区)

缓冲区分为直接缓冲区和堆字节缓冲区。

我们既可以从Buffer中读取数据，也可以向Buffer中写数据，因此，

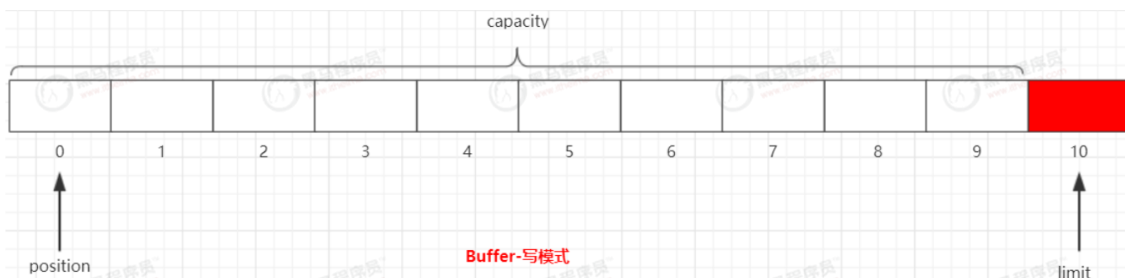
Buffer分为两种模式：**写模式**和**读模式**

为了理解Buffer的工作原理，需要熟悉它的三个属性：

1. capacity (容量)
2. position (位置)
3. limit(限制)

写模式

Buffer底层是通过数组进行实现的，下图展示的就是一个Buffer的简易结构：

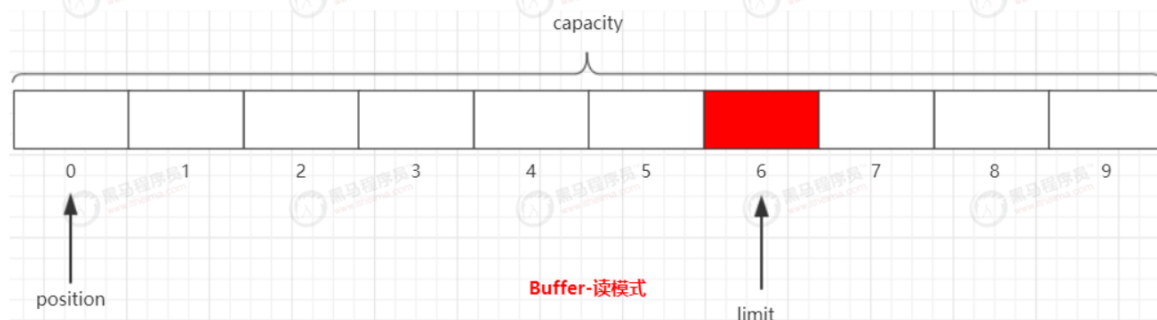


capacity: 数组中可以存储元素的个数。

position: 下一次可插入元素的位置，默认值为0，每添加一次元素向后移动一位。最大值：**capacity - 1**

limit : 在写模式下，**limit**表示第一个不可写的位置（默认第一个不可写的位置，应该是数组容量值的下一个位置，**limit**是从0开始计算，因此默认值等于**capacity**的值）。

读模式



flip, 写---读

capacity: 数组中可以存储元素的个数

position: 当读取数据时, 也是从某个特定位置读。当将Buffer从写模式切换到读模式(一般情况下我们在进行读取数据之前都需要对Buffer做一个读写转换), position会被重置为0。当从

- Buffer的position处读取数据时, position向前移动到下一个可读的位置。

limit : 在读模式下, limit表示第一个不可读的位置, 从0开始进行计算。当切换Buffer到读模式时, limit会被设置成写模式下的position值。换句话说, 你能读到之前写入的所有数据:

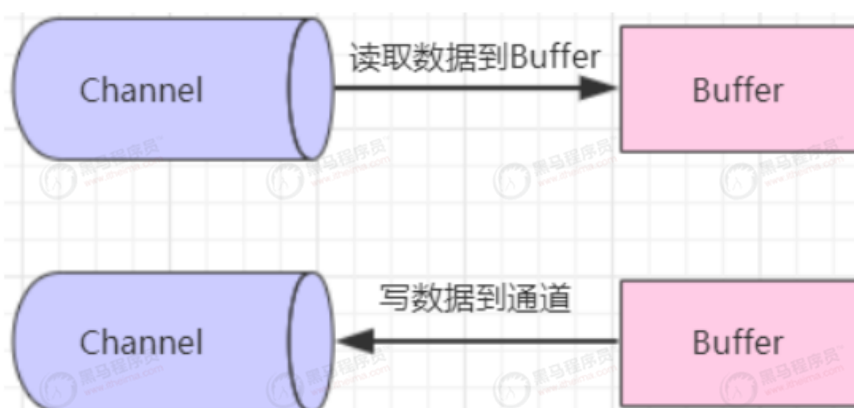
2、Channel(通道)

Java NIO的通道类似流(用于进行数据传输), 但和流是不同的。

区别: 通道是双向的, 而流是单向的(大部分流对象的功能都是比较单一, 要么进行读数据要么进行写数据)。

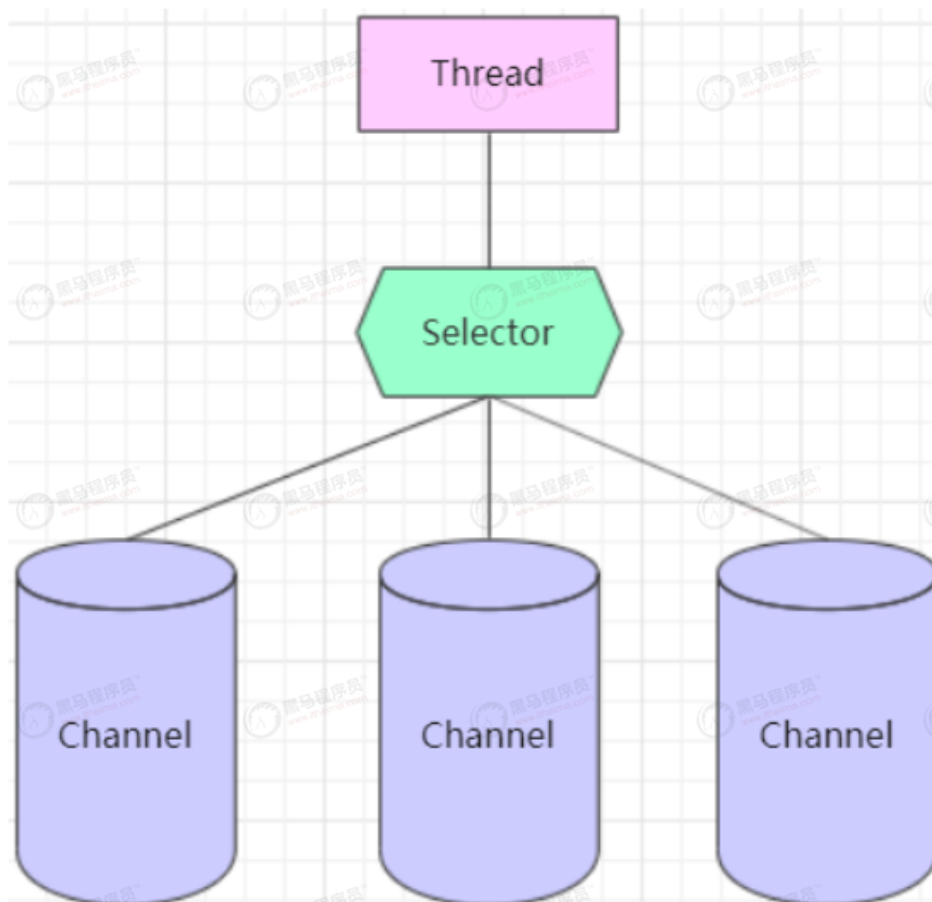
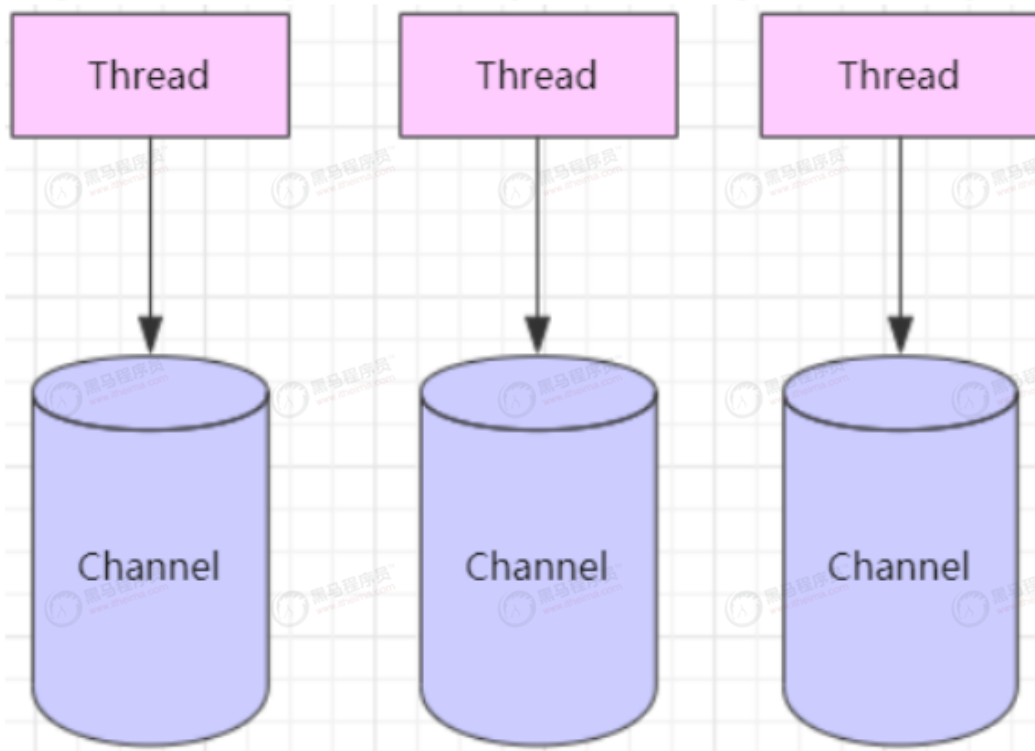
通道使用的时候需要结合Buffer。要将操作的数据从源打包到缓冲区中, 而缓冲区中的数据想要传输到目的地是要依赖于通道的。NIO 技术中的数据要放在缓冲区中进行管理, 再使用通道将缓冲区中

的数据传输到目的地。通道中的数据总是要先读到一个Buffer(从通道读取数据到缓冲区), 或者总是要从一个Buffer中写入(从缓冲区写入数据到通道)。如下图所示:



3、Selector(选择器)

每一个通道都存在一个线程对其进行处理。如果在高并发环境下, 就会存在很多个通道, 那么就会创建很多的线程对象, 造成内存占用率升高, 增加CPU在多个线程之间切换的时间, 因此, 此种设计就不适用于高并发的场景。如下图所示:



这种通过一个线程来处理多个通道任务的机制，在NIO技术中称为"IO多路复用"。

使用了I/O多路复用后，只需要使用1个线程就可操作多个通道，这对高并发高频段处理的业务环境有非常重要的优势。

注：线程数会随着通道的多少动态地增减以进行适配，在内部其实并不永远是一个线程，多路复用的核心目的就是使用最少的线程去操作更多的通道JDK的源代码中，创建线程的个数是根据通道的数量来决定的，每注册1023个通道就创建1个新的线程。

2) NIO多路复用解决BIO并发

```
package com.itheima.socket.tcp;

import java.io.InputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

/**
 * @Class: SocketServer
 * @Package com.itheima.socket.tcp
 * @Description: NIO改造
 * 1.解决并发问题（BIO串行问题）
 * 1.解决传统BIO其中一个客户端连接成功，不发送数据，其他客户端无法连接
 * @Company: http://www.itheima.com/
 */
public class SocketNioServer {

    public static void main(String[] args) throws Exception {

        // 打开一个ServerSocketChannel通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

        // 为ServerSocketChannel绑定地址信息(主机地址和端口号)
        serverSocketChannel.bind(new InetSocketAddress("127.0.0.1", 8080));

        // 设置ServerSocketChannel为非阻塞
        serverSocketChannel.configureBlocking(false);

        // 开启一个选择器
        Selector selector = Selector.open();

        // 将ServerSocketChannel通道注册到选择器上
        //SelectionKey.OP_ACCEPT接收连接进行事件，表示服务器监听到了客户连接，那么服务器可以接收这个连接了
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        // 需要不断的进行选择操作，选择哪些具有就绪任务的通道信息
        while (true) {
            System.out.println("等待连接，阻塞中....");
            // 调用选择器select()方法进行选择操作
            int count = selector.select(); // 此方法是一个阻塞方法，如果没有就绪任务的通道此方法阻塞
            if (count != 0) {
```

```

// 调用Selector的selectedKeys()方法，获取"已选择的键的集合"
Set<SelectionKey> selectedKeys = selector.selectedKeys();
// 遍历集合
Iterator<SelectionKey> iterator = selectedKeys.iterator();
while (iterator.hasNext()) {

    // 获取每一个SelectionKey
    SelectionKey selectionKey = iterator.next();

    // 判断每一个SelectionKey的就绪任务类型，针对不同的任务给出不同的处理
    if (selectionKey.isAcceptable()) { // 可接收连接任务就绪
        System.out.println("客户端连接成功，但尚未发送数据");
        // 获取该SelectionKey所关联的通道
        ServerSocketChannel ssc = (ServerSocketChannel)
selectionKey.channel();
        SocketChannel socketChannel = ssc.accept();
        // 获取一个连接
        socketChannel.configureBlocking(false);
        // 将其设置为非阻塞
        // 将该通道注册到选择器上，读就绪事件，表示通道中已经有了可读的数据，可以执行读操作了
        socketChannel.register(selector, SelectionKey.OP_READ);

    } else if (selectionKey.isReadable()) { // 可读取数据的任务就绪
        System.out.println("客户端成功发送数据");
        // 获取该SelectionKey所关联的通道
        SocketChannel socketChannel = (SocketChannel)
selectionKey.channel();

        // 读取通道中的数据
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int read = socketChannel.read(buffer);
        while (read > 0) {
            // 将Buffer从写模式切换到读模式
            buffer.flip();
            System.out.print(new String(buffer.array(), 0,
read));
            //清除此缓冲区(数据还在)。位置设置为零，限制设置为容量
            buffer.clear();
            read = socketChannel.read(buffer);
        }

        // 释放资源
        socketChannel.close();

    }

    // 任务处理完毕以后，将SelectionKey从"已选择的键的集合"移除掉
    iterator.remove();
}
}
}

```

}

}

2.5.3 WebSocket实战与应用

WebSocket

[编辑](#)[讨论](#)[上传视频](#)[+](#) [★ 收藏](#) | [👍 446](#) | [🔗 174](#)

本词条由“科普中国”科学百科词条编写与应用工作项目 审核。

WebSocket是一种在单个TCP连接上进行全双工通信的协议。WebSocket通信协议于2011年被IETF定为标准RFC 6455，并由RFC7936补充规范。WebSocket API也被W3C定为标准。

WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

中文名	WebSocket	解 释	基于TCP的全双工通信协议
外文名	WebSocket	优 点	服务器可以主动传送数据给客户端
		功 能	实现了浏览器与服务全双工通信

- 1、WebSocket是双向通信协议，模拟Socket协议，可以双向发送或接受信息
- 2、WebSocket和Socket编程一样，都是基于TCP的，也是可靠性传输协议
- 3、WebSocket也可以做Socket可以多做的事情比如IM
- 4、WebSocket在性能上可以解决socket存在的一些问题

2.6 网络编程常用类深入剖析

2.6.1 TCP编程常用类

TCP通信（编程）常用类包含：ServerSocket 类和 Socket 类

TCP 网络程序是指利用 Socket 编写的通信程序。利用 TCP 协议进行通信的两个应用程序是有主次之分的，一个是服务器程序，一个是客户端程序，两者的功能和编写方法不太一样。其中 ServerSocket 类表示 Socket 服务器端，Socket 类表示 Socket 客户端，两者之间的交互过程如下：

1. 服务器端创建一个 ServerSocket（服务器端套接字），调用 accept() 方法等待客户端来连接。
2. 客户端程序创建一个 Socket，请求与服务器建立连接。
3. 服务器接收客户端的连接请求，同时创建一个新的 Socket 与客户端建立连接，服务器继续等待新的请求。

1) ServerSocket类剖析

ServerSocket 构造方法

- ServerSocket(): 无参构造方法。
- ServerSocket(int port): 创建绑定到特定端口的服务器套接字。
- ServerSocket(int port,int backlog): 使用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口。

- `ServerSocket(int port,int backlog,InetAddress bindAddr)`: 使用指定的端口、监听 backlog 和要绑定到本地的 IP 地址创建服务器

`port` 指的是本地 TCP 端口, `backlog` 指的是监听 backlog, `bindAddr` 指的是要将服务器绑定到的 `InetAddress`**

ServerSocket 的常用方法

- `Server accept()`: 监听并接收到此套接字的连接。
- `void bind(SocketAddress endpoint)`: 将 `ServerSocket` 绑定到指定地址 (IP 地址和端口号)。
- `void close()`: 关闭此套接字。
- `InetAddress getInetAddress()`: 返回此服务器套接字的本地地址。
- `int getLocalPort()`: 返回此套接字监听的端口。
- `SocketAddress getLocalSocketAddress()`: 返回此套接字绑定的端口的地址, 如果尚未绑定则返回 `null`。
- `int getReceiveBufferSize()`: 获取此 `ServerSocket` 的 `SO_RCVBUF` 选项的值, 该值是从 `ServerSocket` 接收的套接字的建议缓冲区大小。

调用 `accept()` 方法会返回一个和客户端 `Socket` 对象相连接的 `Socket` 对象, 服务器端的 `Socket` 对象使用 `getOutputStream()` 方法获得的输出流将指定客户端 `Socket` 对象使用 `getInputStream()` 方法获得的那个输入流。同样, 服务器端的 `Socket` 对象使用的 `getInputStream()` 方法获得的输入流将指向客户端 `Socket` 对象使用的 `getOutputStream()` 方法获得的那个输出流。也就是说, 当服务器向输出流写入信息时, 客户端通过相应的输入流就能读取, 反之同样如此。

2) Socket 类剖析

Socket 的构造方法

- `Socket()`: 无参构造方法。
- `Socket(InetAddress address,int port)`: 创建一个流套接字并将其连接到指定 IP 地址的指定端口。
- `Socket(InetAddress address,int port,InetAddress localAddr,int localPort)`: 创建一个套接字并将其连接到指定远程地址上的指定远程端口。
- `Socket(String host,int port)`: 创建一个流套接字并将其连接到指定主机上的指定端口。
- `Socket(String host,int port,InetAddress localAddr,int localPort)`: 创建一个套接字并将其连接到指定远程地址上的指定远程端口。Socket 会通过调用 `bind()` 函数来绑定提供的本地地址及端口。

`address` 指的是远程地址, `port` 指的是远程端口, `localAddr` 指的是要将套接字绑定到的本地地址, `localPort` 指的是要将套接字绑定到的本地端口

Socket 的常用方法

- `void bind(SocketAddress bindpoint)`: 将套接字绑定到本地地址。
- `void close()`: 关闭此套接字。
- `void connect(SocketAddress endpoint)`: 将此套接字连接到服务器。
- `InetAddress getInetAddress()`: 返回套接字的连接地址。
- `InetAddress getLocalAddress()`: 获取套接字绑定的本地地址。
- `InputStream getInputStream()`: 返回此套接字的输入流。
- `OutputStream getOutputStream()`: 返回此套接字的输出流。

- `SocketAddress getLocalSocketAddress()`: 返回此套接字绑定的端点地址，如果尚未绑定则返回 `null`。
- `SocketAddress getRemoteSocketAddress()`: 返回此套接字的连接的端点地址，如果尚未连接则返回 `null`。
- `int getLocalPort()`: 返回此套接字绑定的本地端口。
- `int getPort()`: 返回此套接字连接的远程端口

Internet 上的主机有两种方式表示地址，分别为域名和 **IP** 地址。**java.net** 包中的 **InetAddress** 类对象包含一个主机地址的域名和 **IP** 地址。

InetAddress 类提供操作 **IP** 地址的各种方法。该类本身没有构造方法，而是通过调用相关静态方法获取实例。**InetAddress** 类中的常用方法如下表所示：

方法名称	说明
<code>boolean equals(Object obj)</code>	将此对象与指定对象比较
<code>byte[] getAddress()</code>	返回此 <code>InetAddress</code> 对象的原始 IP 地址
<code>static InetAddress[] getAByName(String host)</code>	在给定主机名的情况下，根据系统上配置的名称，服务器返回其 IP 地址所组成的数组
<code>static InetAddress getByAddress(byte[] addr)</code>	在给定原始 IP 地址的情况下，返回 <code>InetAddress</code> 对象
<code>static InetAddress getByAddress(String host)</code>	在给定主机名的情况下确定主机的 IP 地址
<code>String getCanonicalHostName()</code>	获取此 IP 地址的完全限定域名
<code>String.getHostAddress()</code>	返回 IP 地址字符串（以文本表现形式）
<code>String getHostName()</code>	返回此 IP 地址的主机名
<code>static InetAddress getLocalHost()</code>	返回本地主机