day12【JUnit单元测试、网络编程】

教学目标

- [ ] 能够使用Junit进行单元测试
- [ ] 能够辨别UDP和TCP协议特点
- [ ] 能够说出TCP协议下两个常用类名称
- [ ] 能够编写TCP协议下字符串数据传输程序
- [ ] 能够理解TCP协议下文件上传案例
- [ ] 能够理解TCP协议下BS案例

第一章 Junit单元测试

1.1 什么是JUnit

```
Junit是什么
    *  Junit是Java语言编写的第三方单元测试框架(工具类)
    *  类库 ==> 类  junit.jar

单元测试概念
    * 单元：在Java中，一个类就是一个单元
    * 单元测试：程序猿编写的一小段代码，用来对某个类中的某个方法进行功能测试或业务逻辑测试。

Junit单元测试框架的作用
    * 用来对类中的方法功能进行有目的的测试，以保证程序的正确性和稳定性。
    * 能够让方法独立运行起来。

Junit单元测试框架的使用步骤
    * 编写业务类，在业务类中编写业务方法。比如增删改查的方法
    * 编写测试类，在测试类中编写测试方法，在测试方法中编写测试代码来测试。
        * 测试类的命名规范：以Test开头，以业务类类名结尾，使用驼峰命名法
            * 每一个单词首字母大写，称为大驼峰命名法，比如类名，接口名...
            * 从第二单词开始首字母大写，称为小驼峰命名法，比如方法命名
            * 比如业务类类名：ProductDao，那么测试类类名就应该叫：TestProductDao
        * 测试方法的命名规则：以test开头，以业务方法名结尾
            * 比如业务方法名为：save，那么测试方法名就应该叫：testSave

测试方法注意事项
    * 必须是public修饰的，没有返回值，没有参数
    * 必须使注解@Test修饰

如何运行测试方法
    * 选中方法名 --> 右键 --> Run '测试方法名'  运行选中的测试方法
    * 选中测试类类名 --> 右键 --> Run '测试类类名'  运行测试类中所有测试方法
    * 选中模块名 --> 右键 --> Run 'All Tests'  运行模块中的所有测试类的所有测试方法

如何查看测试结果
    * 绿色：表示测试通过
    * 红色：表示测试失败，有问题
```

1.2 Junit常用注解(Junit4.x版本)

```
* @Before：用来修饰方法，该方法会在每一个测试方法执行之前执行一次。
* @After：用来修饰方法，该方法会在每一个测试方法执行之后执行一次。
* @BeforeClass：用来静态修饰方法，该方法会在所有测试方法之前执行一次。
* @AfterClass：用来静态修饰方法，该方法会在所有测试方法之后执行一次。
```

1.3 Junit常用注解(Junit5.x版本)

```
 * @BeforeEach：用来修饰方法，该方法会在每一个测试方法执行之前执行一次。
 * @AfterEach：用来修饰方法，该方法会在每一个测试方法执行之后执行一次。
 * @BeforeAll：用来静态修饰方法，该方法会在所有测试方法之前执行一次。
 * @AfterAll：用来静态修饰方法，该方法会在所有测试方法之后执行一次。
```

- 示例代码

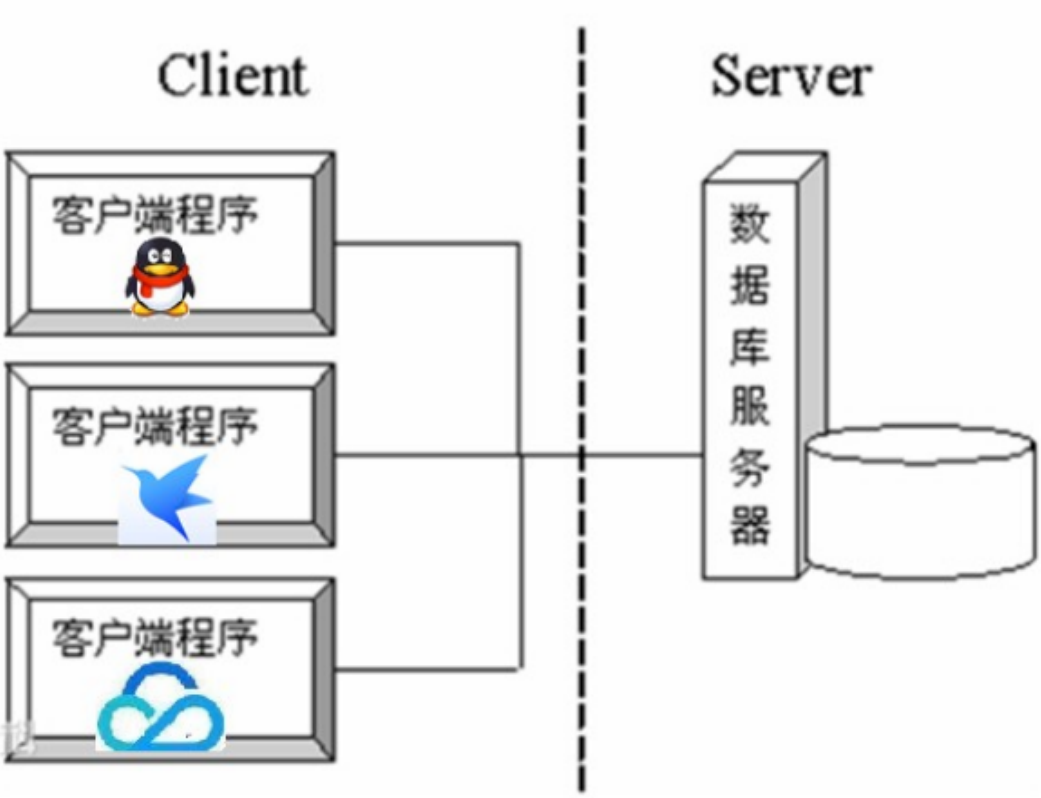
```
/**
 业务类：实现加减乘除运算
 */
public class Cacluate {
    /*
     业务方法1：求a和b之和
     */
    public int sum(int a,int b){
        return  a + b + 10;
    }
    /*
     业务方法2:求a和b之差
     */
    public int sub(int a,int b){
        return  a - b;
    }
}

public class TestCacluate {

    static Cacluate c = null;

    @BeforeClass // 用来静态修饰方法，该方法会在所有测试方法之前执行一次。
    public static void init(){
        System.out.println("初始化操作");
        // 创建Cacluate对象
        c = new Cacluate();
    }

    @AfterClass // 用来静态修饰方法，该方法会在所有测试方法之后执行一次。
    public static void close(){
        System.out.println("释放资源");
        c = null;
    }

   /* @Before // 用来修饰方法，该方法会在每一个测试方法执行之前执行一次。
    public void init(){
```

```
        System.out.println("初始化操作");
        // 创建Cacluate对象
        c = new Cacluate();
    }

    @After // 用来修饰方法，该方法会在每一个测试方法执行之后执行一次。
    public void close(){
        System.out.println("释放资源");
        c = null;
    }*/

    @Test
    public void testSum(){
        int result = c.sum(1,1);
        /*
            断言：预习判断某个条件一定成立，如果条件不成立，则直接奔溃。
            assertEquals方法的参数
            (String message, double expected, double actual)
            message： 消息字符串
            expected: 期望值
            actual: 实际值
         */
        // 如果期望值和实际值一致，则什么也不发生，否则会直接奔溃。
        Assert.assertEquals("期望值和实际值不一致",12,result);
        System.out.println(result);
    }

    @Test
    public void testSub(){
        // 创建Cacluate对象
        // Cacluate c = new Cacluate();

        int result = c.sub(1,1);
        // 如果期望值和实际值一致，则什么也不发生，否则会直接奔溃。
        Assert.assertEquals("期望值和实际值不一致",0,result);
        System.out.println(result);
    }
}
```

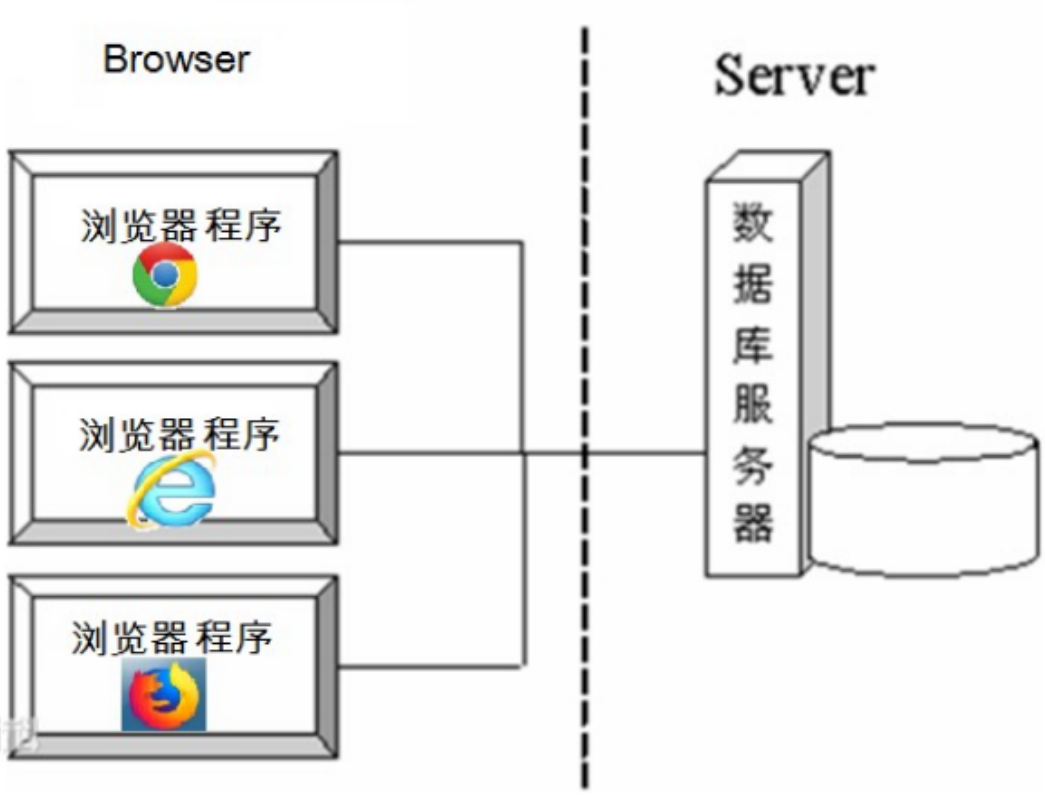
第二章 网络编程入门

2.1软件结构

- **C/S结构**： 全称为Client/Server结构，是指客户端和服务器结构。常见程序有ＱＱ、迅雷等软件。



B/S结构：全称为Browser/Server结构，是指浏览器和服务器结构。常见浏览器有谷歌、火狐等。

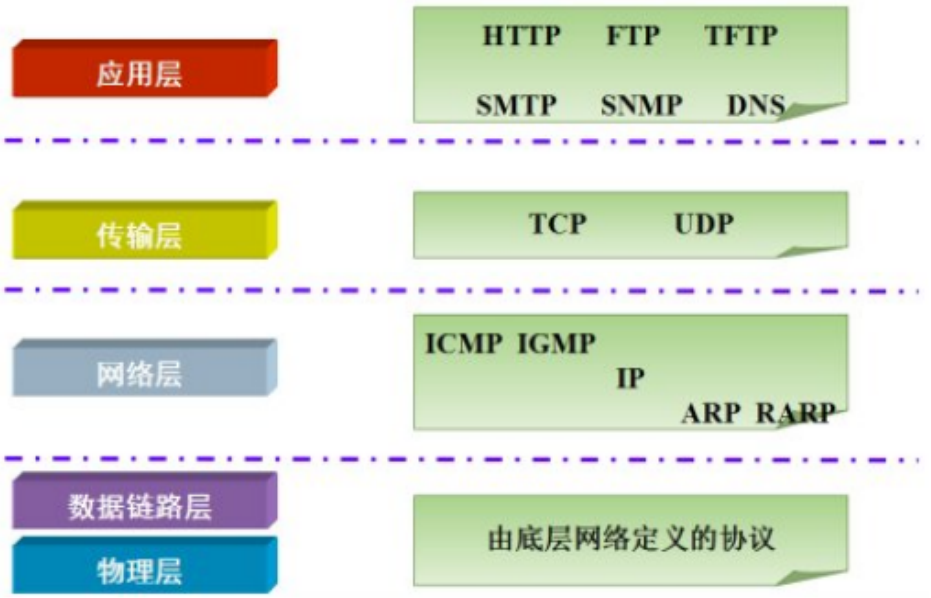


两种架构各有优势，但是无论哪种架构，都离不开网络的支持。**网络编程**，就是在一定的协议下，实现两台计算机的通信的程序。

2.2 网络通信协议

- **网络通信协议：**通信协议是对计算机必须遵守的规则，只有遵守这些规则，计算机之间才能进行通信。这就好比在道路中行驶的汽车一定要遵守交通规则一样，协议中对数据的传输格式、传输速率、传输步骤等做了统一规定，通信双方必须同时遵守，最终完成数据交换。

- **TCP/IP协议：** 传输控制协议/因特网互联协议(Transmission Control Protocol/Internet Protocol)，是Internet最基本、最广泛的协议。它定义了计算机如何连入因特网，以及数据如何在它们之间传输的标准。它的内部包含一系列的用于处理数据通信的协议，并采用了4层的分层模型，每一层都呼叫它的下一层所提供的协议来完成自己的需求。

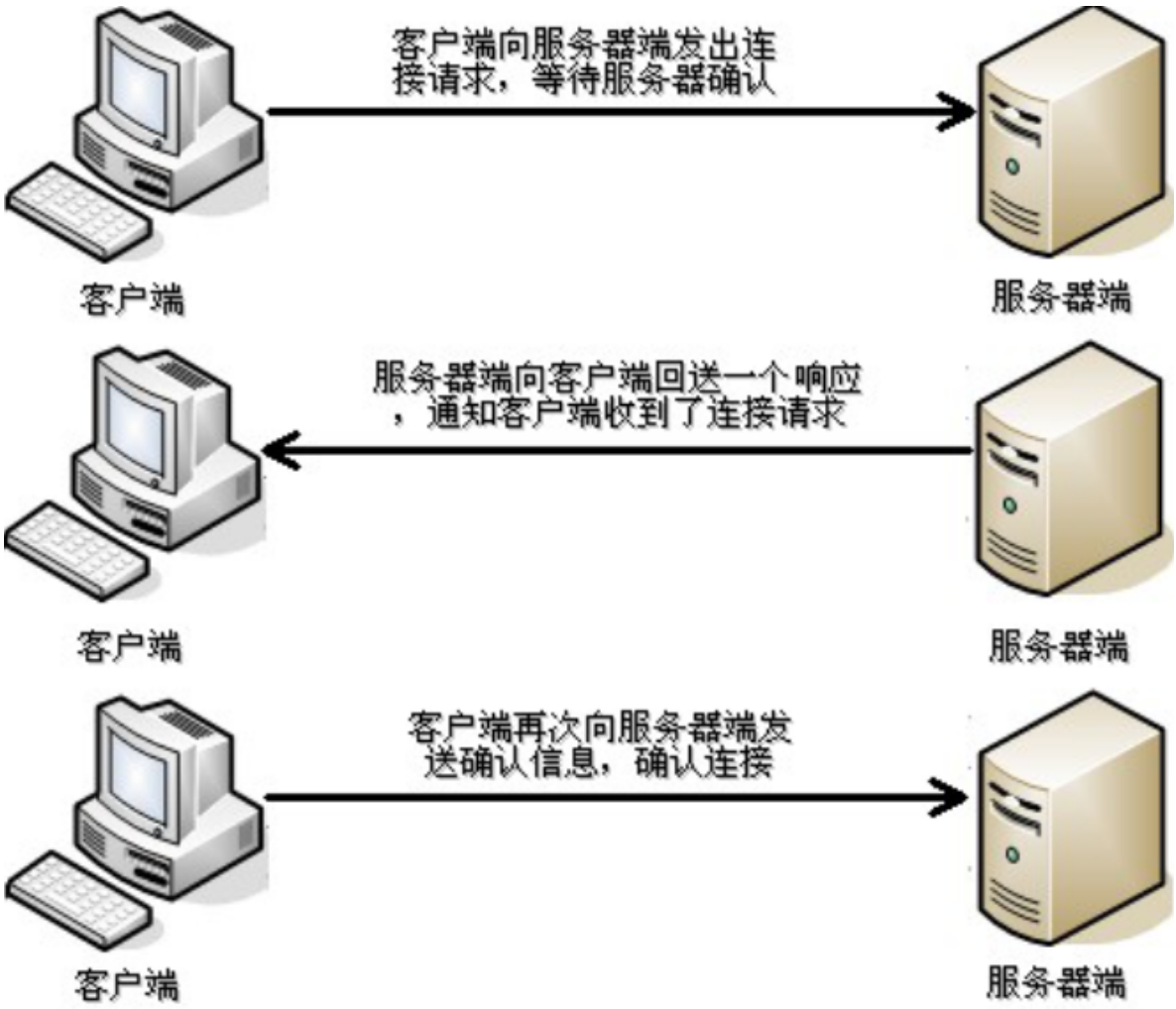


2.3 协议分类

通信的协议还是比较复杂的，`java.net` 包中包含的类和接口，它们提供低层次的通信细节。我们可以直接使用这些类和接口，来专注于网络程序开发，而不用考虑通信的细节。

`java.net` 包中提供了两种常见的网络协议的支持：

- **TCP**：传输控制协议 (Transmission Control Protocol)。TCP协议是**面向连接**的通信协议，即传输数据之前，在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。
    - 三次握手：TCP协议中，在发送数据的准备阶段，客户端与服务器之间的三次交互，以保证连接的可靠。
        - 第一次握手，客户端向服务器端发出连接请求，等待服务器确认。服务器你死了吗?
        - 第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求。我活着啊！！
        - 第三次握手，客户端再次向服务器端发送确认信息，确认连接。整个交互过程如下图所示。我知道了！！



完成三次握手，连接建立后，客户端和服务器就可以开始进行数据传输了。由于这种面向连接的特性，TCP协议可以保证传输数据的安全，所以应用十分广泛，例如下载文件、浏览网页等。

- **UDP**：用户数据报协议(User Datagram Protocol)。UDP协议是一个**面向无连接**的协议。传输数据时，不需要建立连接，不管对方端服务是否启动，直接将数据、数据源和目的地都封装在数据包中，直接发送。每个数据包的大小限制在64k以内。它是不可靠协议，因为无连接，所以传输速度快，但是容易丢失数据。日常应用中,例如视频会议、QQ聊天等。

2.4 网络编程三要素

协议

- **协议：**计算机网络通信必须遵守的规则，已经介绍过了，不再赘述。

IP地址

- **IP地址：指互联网协议地址（Internet Protocol Address）**，俗称IP。IP地址用来给一个网络中的计算机设备做唯一的编号。假如我们把“个人电脑”比作“一台电话”的话，那么“IP地址”就相当于“电话号码”。

IP地址分类 [][][][] o o o o

- IPv4：是一个32位的二进制数，通常被分为4个字节，表示成`a.b.c.d`的形式，例如`192.168.65.100`。其中a、b、c、d都是0~255之间的十进制整数，那么最多可以表示42亿个。
- IPv6：由于互联网的蓬勃发展，IP地址的需求量愈来愈大，但是网络地址资源有限，使得IP的分配越发紧张。有资料显示，全球IPv4地址在2011年2月分配完毕。

为了扩大地址空间，拟通过IPv6重新定义地址空间，采用128位地址长度，每16个字节一组，分成8组十六进制数，表示成`ABCD:EF01:2345:6789:ABCD:EF01:2345:6789`，号称可以为全世界的每一粒沙子编上一个网址，这样就解决了网络地址资源数量不够的问题。

常用命令

- 查看本机IP地址，在控制台输入：

```
ipconfig
```

- 检查网络是否连通，在控制台输入：

```
ping 空格 IP地址
ping 220.181.57.216
ping www.baidu.com
```

特殊的IP地址

- 本机IP地址：`127.0.0.1`、`localhost`。

端口号

网络的通信，本质上是两个进程（应用程序）的通信。每台计算机都有很多的进程，那么在网络通信时，如何区分这些进程呢?

如果说**IP地址**可以唯一标识网络中的设备，那么**端口号**就可以唯一标识设备中的进程（应用程序）了。

- **端口号：用两个字节表示的整数，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，普通的应用程序需要使用1024以上的端口号。如果端口号被另外一个服务或应用所占用，会导致当前程序启动失败。

利用`协议`+`IP地址`+`端口号`三元组合，就可以标识网络中的进程了，那么进程间的通信就可以利用这个标识与其它进程进行交互。ddress

```
/**
    InetAddress类概述
        * 一个该类的对象就代表一个IP地址对象。

    InetAddress类成员方法
        * static InetAddress getLocalHost()
            * 获得本地主机IP地址对象
        * static InetAddress getByName(String host)
            * 根据IP地址字符串或主机名获得对应的IP地址对象

        * String getHostName();获得主机名
        * String getHostAddress();获得IP地址字符串
 */
public class InetAddressDemo01 {
    public static void main(String[] args) throws Exception {
        // 获得本地主机IP地址对象
        InetAddress inet01 = InetAddress.getLocalHost();
        // pkxingdeMacBook-Pro.local/10.211.55.2
        // 主机名/ip地址字符串
        System.out.println(inet01);
        // 根据IP地址字符串或主机名获得对应的IP地址对象
        // InetAddress inet02 = InetAddress.getByName("192.168.73.97");
        InetAddress inet02 = InetAddress.getByName("baidu.com");
```

```
        System.out.println(inet02);

        // 获得主机名
        String hostName = inet01.getHostName();
        System.out.println(hostName);
        // 获得IP地址字符串
        String hostAddress = inet01.getHostAddress();
        System.out.println(hostName);
        System.out.println(hostAddress);
    }
}
```

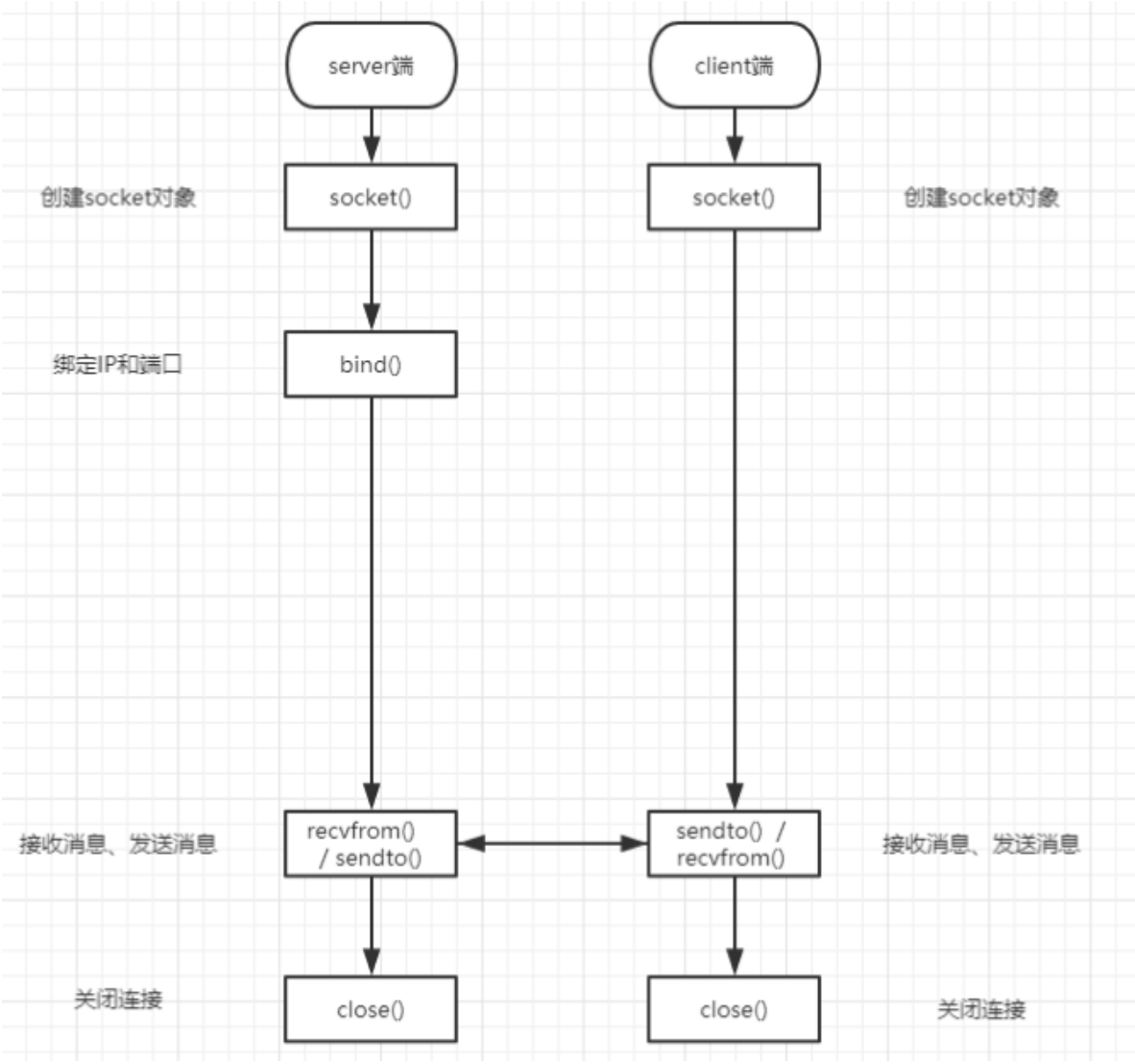
第三章 UDP通信程序

3.1 UDP协议概述

UDP是无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输例如视频会议都使用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

但是在使用UDP协议传送数据时，由于UDP的面向无连接性，不能保证数据的完整性，因此在传输重要数据时不建议使用UDP协议。UDP通信过程如下图所示：



UDP协议的特点

* 面向无连接的协议
* 发送端只管发送，不确认对方是否能收到。
* 基于数据包进行数据传输。
* 发送数据的大小限制64K以内
* 因为面向无连接，速度快，但是不可靠。

UDP协议的使用场景

* 即时通讯
* 在线视频
* 网络语音电话

UDP协议相关的两个类

* DatagramPacket
    * 数据包对象
    * 作用：用来封装要发送或要接收的数据，比如：集装箱
* DategramSocket
    * 发送对象
    * 作用：用来发送或接收数据包，比如：码头

DatagramPacket类构造方法

* DatagramPacket(byte[] buf, int length, InetAddress address, int port)
    * 创建发送端数据包对象
    * buf：要发送的内容，字节数组
    * length：要发送内容的长度，单位是字节
    * address：接收端的IP地址对象

```
        * port：接收端的端口号
    * DatagramPacket(byte[] buf, int length)
        * 创建接收端的数据包对象
        * buf：用来存储接收到内容
        * length：能够接收内容的长度

DatagramPacket类常用方法
    * int getLength() 获得实际接收到的字节个数

DatagramSocket类构造方法
    * DatagramSocket() 创建发送端的Socket对象，系统会随机分配一个端口号。
    * DatagramSocket(int port) 创建接收端的Socket对象并指定端口号

DatagramSocket类成员方法
    * void send(DatagramPacket dp) 发送数据包
    * void receive(DatagramPacket p) 接收数据包
```

3.2 UDP通信案例

- 需求：教师的电脑的一个程序发送数据，一个程序接收数据，使用的教师本机的ip。

3.2.1 UDP发送端代码实现

```
// UDP发送端代码实现
public class UDPSender {
    public static void main(String[] args)throws Exception{
        // 定义一个字符串：要发送的内容
        String message = "约吗";
        // 字符串转字节数组
        byte[] buf = message.getBytes();
        // 创建数据包对象
        DatagramPacket dp = new DatagramPacket(buf,buf.length,
                InetAddress.getLocalHost(),6666);
        // 创建发送端的发送对象
        DatagramSocket ds = new DatagramSocket(8888);
        // 发送数据包
        ds.send(dp);
        // 关闭发送对象释放端口号
        ds.close();
    }
}
```

3.2.2 UDP接收端代码实现

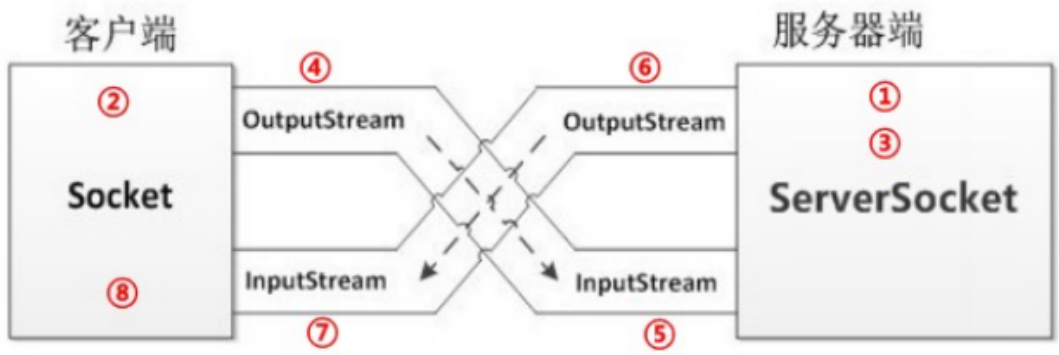
```
/**
    UDP协议接收端代码实现
 */
public class UDPReceive {
    public static void main(String[] args)throws Exception{
        // 创建接收对象DatagramSocket
        DatagramSocket ds = new DatagramSocket(6666);
        // 创建字节数组用来存储接收接收到的内容
        byte[] buf = new byte[1024];
        // 创建数据包对象
        DatagramPacket dp = new DatagramPacket(buf,buf.length);
```

```
        // 接收数据包
        ds.receive(dp);

        // 获得实际接收到的字节个数
        int len = dp.getLength();
        System.out.println("len = " + len);
        // 将字节数组的内容转换为字符串输出
        System.out.println(new String(buf,0,len));

        // 获得发送端的ip地址
        String sendIp = dp.getAddress().getHostAddress();
        // 获得发送端的端口号
        int port  = dp.getPort();
        System.out.println(sendIp);
        System.out.println(port);

        // 关闭Socket对象
        ds.close();
    }
}
```

第四章 TCP通信程序

4.1 TCP协议概述

- TCP协议是面向连接的通信协议，即在传输数据前先在客户端和服务器端建立逻辑连接，然后再传输数据。它提供了两台计算机之间可靠无差错的数据传输。TCP通信过程如下图所示：




```
TCP ==> Transfer Control Protocol ==> 传输控制协议
TCP协议的特点
    * 面向连接的协议
    * 只能由客户端主动发送数据给服务器端，服务器端接收到数据之后，可以给客户端响应数据。
    * 通过三次握手建立连接，连接成功形成数据传输通道。
    * 通过四次挥手断开连接
    * 基于IO流进行数据传输
    * 传输数据大小没有限制
    * 因为面向连接的协议，速度慢，但是是可靠的协议。

TCP协议的使用场景
    * 文件上传和下载
    * 邮件发送和接收
    * 远程登录
```

TCP协议相关的类
    * Socket
        * 一个该类的对象就代表一个客户端程序。
    * ServerSocket
        * 一个该类的对象就代表一个服务器端程序。

Socket类构造方法
    * Socket(String host, int port)
        * 根据ip地址字符串和端口号创建客户端Socket对象
        * 注意事项：只要执行该方法，就会立即连接指定的服务器程序，如果连接不成功，则会抛出异常。
            如果连接成功，则表示三次握手通过。

Socket类常用方法
    * OutputStream getOutputStream(); 获得字节输出流对象
    * InputStream getInputStream();获得字节输入流对象

4.2 TCP通信案例

4.2.2 客户端向服务器发送数据

```
/*
TCP客户端代码实现步骤
        * 创建客户端Socket对象并指定服务器地址和端口号
        * 调用Socket对象的getOutputStream方法获得字节输出流对象
        * 调用字节输出流对象的write方法往服务器端输出数据
        * 调用Socket对象的getInputStream方法获得字节输入流对象
        * 调用字节输入流对象的read方法读取服务器端返回的数据
        * 关闭Socket对象断开连接。
 */
// TCP客户端代码实现
public class TCPClient {
    public static void main(String[] args) throws Exception{
        // 要发送的内容
        String content = "你好TCP服务器端，约吗";
        // 创建Socket对象
        Socket socket = new Socket("192.168.73.99",9999);
        // System.out.println(socket);
        // 获得字节输出流对象
        OutputStream out = socket.getOutputStream();
        // 输出数据到服务器端
        out.write(content.getBytes());

        // 获得字节输入流对象
        InputStream in = socket.getInputStream();
        // 创建字节数组：用来存储读取到服务器端数据
        byte[] buf = new byte[1024];
        // 读取服务器端返回的数据
        int len = in.read(buf);
        System.out.println("len = " + len);
        System.out.println("服务器端返回的内容 = " + new String(buf,0,len));

        // 关闭socket对象
        socket.close();
    }
```

```
}
```

4.2.3 服务器向客户端回写数据

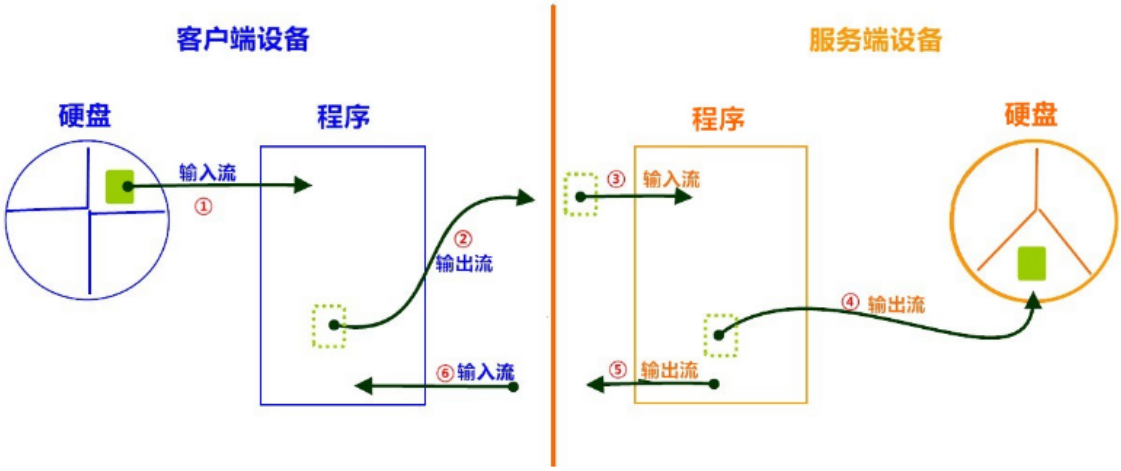
```
/**
    TCP服务器端代码实现

    ServerSocket类构造方法
        * ServerSocket(int port) 根据指定的端口号开启服务器。

    ServerSocket类常用方法
        *  Socket accept() 等待客户端连接并获得与客户端关联的Socket对象

    TCP服务器端代码实现步骤
        * 创建ServerSocket对象并指定端口号(相当于开启了一个服务器)
        * 调用ServerSocket对象的accept方法等待客端户连接并获得对应Socket对象
        * 调用Socket对象的getInputStream方法获得字节输入流对象
        * 调用字节输入流对象的read方法读取客户端发送的数据
        * 调用Socket对象的getOutputStream方法获得字节输出流对象
        * 调用字节输出流对象的write方法往客户端输出数据
        * 关闭Socket和ServerSocket对象
 */
public class TCPServer {
    public static void main(String[] args)throws Exception{
        // 创建服务器ocket对象
        ServerSocket serverSocket = new ServerSocket(9999);
        // 等待客户端连接并获得与客户端关联的Socket对象
        Socket socket = serverSocket.accept();
        // 获得字节输入流对象
        InputStream in = socket.getInputStream();
        // 创建字节数组：用来存储读取到客户端发送的数据
        byte[] buf = new byte[1024];
        // 读取客户端发送过来的数据
        int len = in.read(buf);
        System.out.println("len = " + len);
        System.out.println("客户端发送的数据 = " + new String(buf,0,len));

        // 获得字节输出流对象
        OutputStream out = socket.getOutputStream();
        // 往客户端输出数据
        out.write("约你妹".getBytes());



        // 关闭socket
        socket.close();
        // 关闭服务器（在实际开发中，服务器一般不会关闭）
        serverSocket.close();
    }
}
```

第五章 综合案例

5.1 文件上传案例

文件上传分析图解

1. 【客户端】输入流，从硬盘读取文件数据到程序中。
2. 【客户端】输出流，写出文件数据到服务端。
3. 【服务端】输入流，读取文件数据到服务端程序。
4. 【服务端】输出流，写出文件数据到服务器硬盘中。
5. 【服务端】获取输出流，回写数据。
6. 【客户端】获取输入流，解析回写数据。



案例实现

服务器端实现：

```
public class FileUpload_Server {
    public static void main(String[] args) throws IOException {
        System.out.println("服务器 启动.....  ");
        // 1. 创建服务端ServerSocket
        ServerSocket serverSocket = new ServerSocket(6666);
        // 2. 循环接收,建立连接
        while (true) {
            Socket accept = serverSocket.accept();
            /*
            3. socket对象交给子线程处理,进行读写操作
               Runnable接口中,只有一个run方法,使用lambda表达式简化格式
            */
            new Thread(() -> {
                try (
                    //3.1 获取输入流对象
                    BufferedInputStream bis = new
BufferedInputStream(accept.getInputStream());
                    //3.2 创建输出流对象, 保存到本地 .
                    FileOutputStream fis = new
FileOutputStream(System.currentTimeMillis() + ".jpg");
                    BufferedOutputStream bos = new BufferedOutputStream(fis);
                ) {
                    // 3.3 读写数据
                    byte[] b = new byte[1024 * 8];
                    int len;
                    while ((len = bis.read(b)) != -1) {
                        bos.write(b, 0, len);
                    }
```

```
                    // 4.=======信息回写==========================
                    System.out.println("back ........");
                    OutputStream out = accept.getOutputStream();
                    out.write("上传成功".getBytes());
                    out.close();
                    //==============================

                    //5. 关闭 资源
                    bos.close();
                    bis.close();
                    accept.close();
                    System.out.println("文件上传已保存");
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}
```

客户端实现:

```
public class FileUpload_Client {
    public static void main(String[] args) throws IOException {
        // 1.创建流对象
        // 1.1 创建输入流,读取本地文件
        BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("test.jpg"));
        // 1.2 创建输出流,写到服务端
        Socket socket = new Socket("localhost", 6666);
        BufferedOutputStream bos = new
BufferedOutputStream(socket.getOutputStream());

        //2.写出数据.
        byte[] b  = new byte[1024 * 8 ];
        int len ;
        while (( len  = bis.read(b))!=-1) {
            bos.write(b, 0, len);
        }
        // 关闭输出流,通知服务端,写出数据完毕
        socket.shutdownOutput();
        System.out.println("文件发送完毕");
        // 3. =====解析回写============
        InputStream in = socket.getInputStream();
        byte[] back = new byte[20];
        in.read(back);
        System.out.println(new String(back));
        in.close();
        // ============================

        // 4.释放资源
        socket.close();
        bis.close();
    }
}
```

5.2 模拟B\S服务器

模拟网站服务器，使用浏览器访问自己编写的服务端程序，查看网页效果。

案例分析

1. 准备页面数据，web文件夹。
2. 我们模拟服务器端，ServerSocket类监听端口，使用浏览器访问，查看网页效果

案例实现

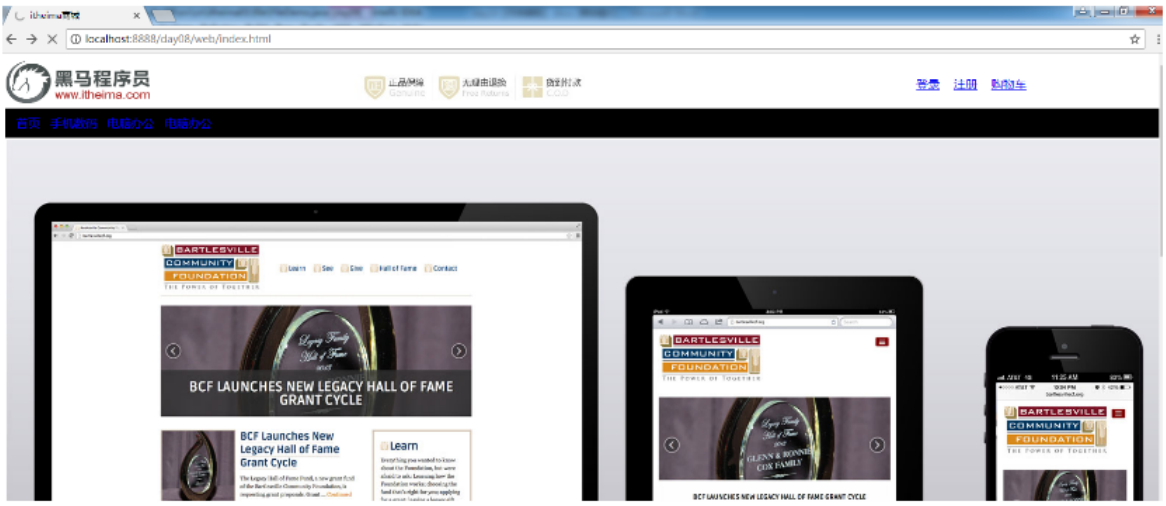
浏览器工作原理是遇到图片会开启一个线程进行单独的访问,因此在服务器端加入线程技术。

```
public class ServerDemo {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(8888);
        while(true){
            Socket socket = server.accept();
            new Thread(new Web(socket)).start();
        }
    }
}
```

```
class Web implements Runnable{
    private Socket socket;

    public Web(Socket socket){
        this.socket=socket;
    }

    public void run() {
        try{
            //转换流,读取浏览器请求第一行
            BufferedReader readWb = new
                    BufferedReader(new
InputStreamReader(socket.getInputStream()));
            String requst = readWb.readLine();
            //取出请求资源的路径
            String[] strArr = requst.split(" ");
            System.out.println(Arrays.toString(strArr));
            String path = strArr[1].substring(1);
            System.out.println(path);

            FileInputStream fis = new FileInputStream(path);
            System.out.println(fis);
            byte[] bytes= new byte[1024];
            int len = 0 ;

            //向浏览器 回写数据
            OutputStream out = socket.getOutputStream();
            out.write("HTTP/1.1 200 OK\r\n".getBytes());
            out.write("Content-Type:text/html\r\n".getBytes());
            out.write("\r\n".getBytes());
            while((len = fis.read(bytes))!=-1){
                out.write(bytes,0,len);
            }
```

```
            fis.close();
            out.close();
            readWb.close();
            socket.close();
        }catch(Exception ex){

        }
    }
}
```

访问效果：



图解：

第章 NIO

5.1 NIO概述

5.1.1 NIO引入

在我们学习Java的NIO流之前，我们都要了解几个关键词

- 同步与异步（synchronous/asynchronous）：**同步**是一种可靠的有序运行机制，当我们进行同步操作时，后续的任务是等待当前调用返回，才会进行下一步；而**异步**则相反，其他任务不需要等待当前调用返回，通常依靠事件、回调等机制来实现任务间次序关系

- 阻塞与非阻塞：在进行**阻塞**操作时，当前线程会处于阻塞状态，无法从事其他任务，只有当条件就绪才能继续，比如ServerSocket新连接建立完毕，或者数据读取、写入操作完成；而**非阻塞**则是不管IO操作是否结束，直接返回，相应操作在后台继续处理

在Java1.4之前的I/O系统中，提供的都是面向流的I/O系统，系统一次一个字节地处理数据，一个输入流产生一个字节的数据，一个输出流消费一个字节的数据，面向流的I/O速度非常慢，而在Java 1.4中推出了NIO，这是一个面向块的I/O系统，系统以块的方式处理处理，每一个操作在一步中产生或者消费一个数据库，按块处理要比按字节处理数据快的多。

在 Java 7 中，NIO 有了进一步的改进，也就是 NIO 2，引入了异步非阻塞 IO 方式，也有很多人叫它 AIO（Asynchronous IO）。异步 IO 操作基于事件和回调机制，可以简单理解为，应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

NIO之所以是同步，是因为它的accept/read/write方法的内核I/O操作都会阻塞当前线程

首先，我们要先了解一下NIO的三个主要组成部分：Buffer（缓冲区）、Channel（通道）、Selector（选择器）

5.2 NIO-Buffer类

5.2.1 Buffer概述

Buffer是一个对象，它包含一些要写入或者读到Stream对象的。应用程序不能直接对 Channel 进行读写操作，而必须通过 Buffer 来进行，即 Channel 是通过 Buffer 来读写数据的。

在NIO中，所有的数据都是用Buffer处理的，它是NIO读写数据的中转池。Buffer实质上是一个数组，通常是一个字节数据，但也可以是其他类型的数组。但一个缓冲区不仅仅是一个数组，重要的是它提供了对数据的结构化访问，而且还可以跟踪系统的读写进程。

使用 Buffer 读写数据一般遵循以下四个步骤：

1.写入数据到 Buffer；

2.调用 flip() 方法；

3.从 Buffer 中读取数据；

4.调用 clear() 方法或者 compact() 方法。

当向 Buffer 写入数据时，Buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip() 方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 Buffer 的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear() 或 compact() 方法。clear() 方法会清空整个缓冲区。compact() 方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

Buffer主要有如下几种：

- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

5.2.2 创建ByteBuffer

- ByteBuffer类内部封装了一个byte[]数组，并可以通过一些方法对这个数组进行操作。

- 创建ByteBuffer对象
  - 方式一：在堆中创建缓冲区：allocate(int capacity)

```
public static void main(String[] args) {
        //创建堆缓冲区
        ByteBuffer byteBuffer = ByteBuffer.allocate(10);
}
```

  - 在系统内存创建缓冲区：allocatDirect(int capacity)

```
public static void main(String[] args) {
        //创建直接缓冲区
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(10);
}
```

    - 在堆中创建缓冲区称为：间接缓冲区
    - 在系统内存创建缓冲区称为：直接缓冲区
  - 间接缓冲区的创建和销毁效率要高于直接缓冲区
    - 间接缓冲区的工作效率要低于直接缓冲区
  - 方式三：通过数组创建缓冲区：wrap(byte[] arr)

```
 public static void main(String[] args) {
        byte[] byteArray = new byte[10];
        ByteBuffer byteBuffer = ByteBuffer.wrap(byteArray);
 }
```

    - 此种方式创建的缓冲区为：间接缓冲区

5.2.3 向ByteBuffer添加数据

- public ByteBuffer put(byte b)：向当前可用位置添加数据。

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    buf.put((byte) 10);
    buf.put((byte) 20);

    System.out.println(Arrays.toString(buf.array()));
}
```

打印结果：

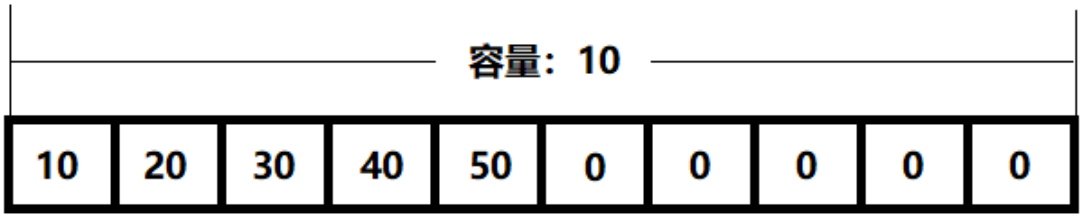
```
[10, 20, 0, 0, 0, 0, 0, 0, 0, 0]
```

- public ByteBuffer put(byte[] byteArray)：向当前可用位置添加一个byte[]数组

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    buf.put((byte) 10);
    buf.put((byte) 20);

    byte[] byteArray = {30, 40, 50};
    buf.put(byteArray);//添加整个数组

    System.out.println(Arrays.toString(buf.array()));
}
```

打印结果：

```
[10, 20, 30, 40, 50, 0, 0, 0, 0, 0]
```



- public ByteBuffer put(byte[] byteArray,int offset,int len)：添加一个byte[]数组的一部分

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    buf.put((byte) 10);
    buf.put((byte) 20);

    byte[] byteArray = {30, 40, 50};
    buf.put(byteArray,0,2);//只添加byteArray的前两个元素

    System.out.println(Arrays.toString(buf.array()));
}
```

打印结果：

```
[10, 20, 30, 40, 0, 0, 0, 0, 0, 0]
```

5.2.4 容量-capacity

- Buffer的容量(capacity)是指：Buffer所能够包含的元素的最大数量。定义了Buffer后，容量是不可变的。

- 示例代码：

```
public static void main(String[] args) {
    ByteBuffer b1 = ByteBuffer.allocate(10);
    System.out.println("容量：" + b1.capacity());//10。之后不可改变

    byte[] byteArray = {97, 98, 99, 100};
    ByteBuffer b2 = ByteBuffer.wrap(byteArray);
    System.out.println("容量：" + b2.capacity());//4。之后不可改变
}
```

- 结果：

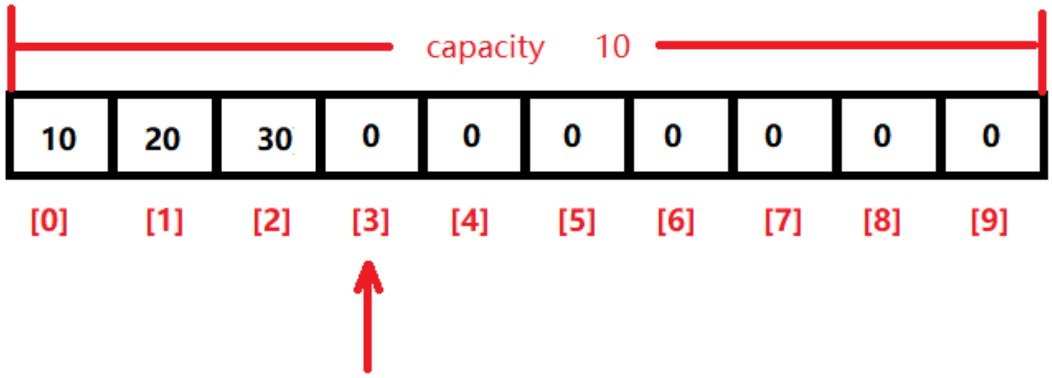
```
容量：10
容量：4
```

5.2.5 限制-limit

- 限制limit是指：第一个不应该读取或写入元素的index索引。缓冲区的限制(limit)不能为负，并且不能大于容量。
- 有两个相关方法：
  - public int limit()：获取此缓冲区的限制。
  - public Buffer limit(int newLimit)：设置此缓冲区的限制。
- 示例代码：

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    System.out.println("初始容量：" + buf.capacity() +
                                "  初始限制：" + buf.limit());//10

    buf.limit(3);//设置限制为：索引3

    buf.put((byte) 10);//索引：0
    buf.put((byte) 20);//索引：1
    buf.put((byte) 30);//索引：2
    buf.put((byte) 40);//抛出异常

}
```

图示：



limit：3，后面的位置将不可用

5.2.6 位置-position

- 位置position是指：当前可写入的索引。位置不能小于0，并且不能大于"限制"。
- 有两个相关方法：
  - public int position()：获取当前可写入位置索引。
  - public Buffer position(int p)：更改当前可写入位置索引。
- 示例代码：

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    System.out.println("初始容量：" + buf.capacity() +
                       " 初始限制：" + buf.limit() +
                       " 当前位置：" + buf.position());//0

    buf.put((byte) 10);//position = 1
    buf.put((byte) 20);//position = 2
    buf.put((byte) 30);//position = 3
    System.out.println("当前容量：" + buf.capacity() +
                       " 初始限制：" + buf.limit() +
                       " 当前位置：" + buf.position());//3

    buf.position(1);//当position改为：1

    buf.put((byte) 2);//添加到索引：1
    buf.put((byte) 3);//添加到索引：2


    System.out.println(Arrays.toString(buf.array()));
}
```

打印结果：

```
初始容量：10 初始限制：10 当前位置：0
初始容量：10 初始限制：10 当前位置：3
[10, 2, 3, 0, 0, 0, 0, 0, 0, 0]
```

5.2.7 标记-mark

- 标记mark是指：当调用缓冲区的reset()方法时，会将缓冲区的position位置重置为该索引。不能为0，不能大于position。
- 相关方法：
  - public Buffer mark()：设置此缓冲区的标记为当前的position位置。
- 示例代码：

```
public static void main(String[] args) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    System.out.println("初始容量：" + buf.capacity() +
                       " 初始限制：" + buf.limit() +
                       " 当前位置：" + buf.position());//初始标记不确定

    buf.put((byte) 10);
    buf.put((byte) 20);
    buf.put((byte) 30);
    System.out.println("当前容量：" + buf.capacity() +
                       " 当前限制：" + buf.limit() +
                       " 当前位置：" + buf.position());
    buf.position(1);//当position改为：1
    buf.mark();//设置索引1位标记点

    buf.put((byte) 2);//添加到索引：1
    buf.put((byte) 3);//添加到索引：2

    //当前position为：3

    //将position设置到之前的标记位：1
    buf.reset();
    System.out.println("reset后的当前位置：" + buf.position());

    buf.put((byte) 20);//添加到索引：1

    System.out.println(Arrays.toString(buf.array()));
}
```

打印结果：

```
初始容量：10 初始限制：10 当前位置：0
当前容量：10 当前限制：10 当前位置：3
reset后的当前位置：1
[10, 20, 3, 0, 0, 0, 0, 0, 0, 0]
```

5.2.8 其它方法

- public int remaining()：获取position与limit之间的元素数。
- public boolean isReadOnly()：获取当前缓冲区是否只读。
- public boolean isDirect()：获取当前缓冲区是否为直接缓冲区。
- public Buffer clear()：还原缓冲区的状态。
  - 将position设置为：0
  - 将限制limit设置为容量capacity；

    - 丢弃标记mark。
- public Buffer flip()：缩小limit的范围。
    - 将limit设置为当前position位置；
    - 将当前position位置设置为0；
    - 丢弃标记。
- public Buffer rewind()：重绕此缓冲区。
    - 将position位置设置为：0
    - 限制limit不变。
    - 丢弃标记。

5.3 Channel（通道）

Channel（通道）：Channel是一个对象，可以通过它读取和写入数据。可以把它看做是IO中的流，不同的是：

- 为所有的原始类型提供（Buffer）缓存支持；
- 字符集编码解决方案（Charset）；
- Channel：一个新的原始I/O抽象；
- 支持锁和内存映射文件的文件访问接口；
- 提供多路（non-bloking）非阻塞式的高伸缩性网路I/O。

正如上面提到的，所有数据都通过Buffer对象处理，所以，您永远不会将字节直接写入到Channel中，相反，您是将数据写入到Buffer中；同样，您也不会从Channel中读取字节，而是将数据从Channel读入Buffer，再从Buffer获取这个字节。

因为Channel是双向的，所以Channel可以比流更好地反映出底层操作系统的真实情况。特别是在Unix模型中，底层操作系统通常都是双向的。

在Java NIO中的Channel主要有如下几种类型：

- FileChannel：从文件读取数据的
- DatagramChannel：读写UDP网络协议数据
- SocketChannel：读写TCP网络协议数据
- ServerSocketChannel：可以监听TCP连接

5.3.1 FileChannel类的基本使用

- java.nio.channels.FileChannel (抽象类)：用于读、写文件的通道。
- FileChannel是抽象类，我们可以通过FileInputStream和FileOutputStream的getChannel()方法方便的获取一个它的子类对象。

```
FileInputStream fi=new FileInputStream(new File(src));
FileOutputStream fo=new FileOutputStream(new File(dst));
//获得传输通道channel
FileChannel inChannel=fi.getChannel();
FileChannel outChannel=fo.getChannel();
```

- 我们将通过CopyFile这个实力让大家体会NIO的操作过程。CopyFile执行三个基本的操作：创建一个Buffer，然后从源文件读取数据到缓冲区，然后再将缓冲区写入目标文件。

```
public static void main(String[] args) throws Exception {
        //声明源文件和目标文件
```

```
        FileInputStream fi=new FileInputStream("d:\\视频.itcast");
        FileOutputStream fo=new FileOutputStream("e:\\视频_copy.itcast");
        //获得传输通道channel
        FileChannel inChannel=fi.getChannel();
        FileChannel outChannel=fo.getChannel();
        //获得容器buffer
        ByteBuffer buffer= ByteBuffer.allocate(1024);
        int eof = 0;
        while((eof =inChannel.read(buffer)) != -1){//读取的字节将会填充buffer的
position到limit位置
            //重设一下buffer：limit=position , position=0
            buffer.flip();
            //开始写
            outChannel.write(buffer);//只输出position到limit之间的数据
            //写完要重置buffer，重设position=0，limit=capacity，用于下次读取
            buffer.clear();
        }
        inChannel.close();
        outChannel.close();
        fi.close();
        fo.close();

    }
```

5.3.2 FileChannel结合MappedByteBuffer实现高效读写

- 上例直接使用FileChannel结合ByteBuffer实现的管道读写，但并不能提高文件的读写效率。
- ByteBuffer有个子类：MappedByteBuffer，它可以创建一个“直接缓冲区”，并可以将文件直接映射至内存，可以提高大文件的读写效率。
  - ByteBuffer(抽象类)

    |--MappedByteBuffer(抽象类)
- 可以调用FileChannel的map()方法获取一个MappedByteBuffer，map()方法的原型：

  MappedByteBuffer map(MapMode mode, long position, long size);

  说明：将节点中从position开始的size个字节映射到返回的MappedByteBuffer中。
- 示例：复制d:\b.rar文件，此文件大概600多兆，复制完毕用时不到2秒。此例不能复制大于2G的文件，因为map的第三个参数被限制在Integer.MAX_VALUE(字节) = 2G。

```
public static void main(String[] args) throws Exception {
        try {
            //java.io.RandomAccessFile类，可以设置读、写模式的IO流类。
            //"r"表示：只读--输入流，只读就可以。
            RandomAccessFile source = new RandomAccessFile("d:\\b.rar",
"r");
            //"rw"表示：读、写--输出流，需要读、写。
            RandomAccessFile target = new RandomAccessFile("e:\\b.rar",
"rw");
            //分别获取FileChannel通道
            FileChannel in = source.getChannel();
            FileChannel out = target.getChannel();
            //获取文件大小
            long size = in.size();
            //调用Channel的map方法获取MappedByteBuffer
```

```
            MappedByteBuffer mbbi = in.map(FileChannel.MapMode.READ_ONLY, 0,
size);
            MappedByteBuffer mbbo = out.map(FileChannel.MapMode.READ_WRITE,
0, size);
            long start = System.currentTimeMillis();
            System.out.println("开始...");
            for (int i = 0; i < size; i++) {
                byte b = mbbi.get(i);//读取一个字节
                mbbo.put(i, b);//将字节添加到mbbo中
            }
            long end = System.currentTimeMillis();
            System.out.println("用时: " + (end - start) + " 毫秒");
            source.close();
            target.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
```

- 代码说明：
  - map()方法的第一个参数mode：映射的三种模式，在这三种模式下得到的将是三种不同的MappedByteBuffer：三种模式都是Channel的内部类MapMode中定义的静态常量，这里以FileChannel举例： 1). **FileChannel.MapMode.READ_ONLY**：得到的镜像只能读不能写（只能使用get之类的读取Buffer中的内容）；

    2). **FileChannel.MapMode.READ_WRITE**：得到的镜像可读可写（既然可写了必然可读），对其写会直接更改到存储节点；

    3). **FileChannel.MapMode.PRIVATE**：得到一个私有的镜像，其实就是一个(position, size)区域的副本罢了，也是可读可写，只不过写不会影响到存储节点，就是一个普通的ByteBuffer了！！
  - 为什么使用RandomAccessFile?

    1). 使用InputStream获得的Channel可以映射，使用map时只能指定为READ_ONLY模式，不能指定为READ_WRITE和PRIVATE，否则会抛出运行时异常！

    2). 使用OutputStream得到的Channel不可以映射！并且OutputStream的Channel也只能write不能read！

    3). 只有RandomAccessFile获取的Channel才能开启任意的这三种模式！
- 下例使用循环，将文件分块，可以高效的复制大于2G的文件：要复制的文件为：d:\测试13G.rar，此文件13G多，复制完成大概30秒左右。

```
import sun.nio.ch.FileChannelImpl;

import java.io.*;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

public class MappedFileChannelTest {
    public static void main(String[] args) throws Exception {
        try {
            RandomAccessFile source = new RandomAccessFile("d:\\测试
13G.rar", "r");
```

```
            RandomAccessFile target = new RandomAccessFile("e:\\测试
13G.rar", "rw");
            FileChannel in = source.getChannel();
            FileChannel out = target.getChannel();
            long size = in.size();//获取文件大小
            long count = 1;//存储分的块数，默认初始化为：1
            long copySize = size;//每次复制的字节数，默认初始化为：文件大小
            long everySize = 1024 * 1024 * 512;//每块的大小，初始化为：512M
            if(size > everySize){//判断文件是否大于每块的大小
                //判断"文件大小"和"每块大小"是否整除，来计算"块数"
                count = (int)(size % everySize != 0 ? size / everySize + 1 :
size / everySize);
                //第一次复制的大小等于每块大小。
                copySize = everySize;
            }

            MappedByteBuffer mbbi = null;//输入的MappedByteBuffer
            MappedByteBuffer mbbo = null;//输出的MappedByteBuffer
            long startIndex = 0;//记录复制每块时的起始位置
            long start = System.currentTimeMillis();
            System.out.println("开始...");
            for (int i = 0; i < count; i++) {
                mbbi =
in.map(FileChannel.MapMode.READ_ONLY,startIndex,copySize);
                mbbo = out.map(FileChannel.MapMode.READ_WRITE,
startIndex,copySize);

                for (int j = 0; j < copySize; j++) {
                    byte b = mbbi.get(i);
                    mbbo.put(i, b);
                }
                startIndex += copySize;//计算下一块的起始位置
                //计算下一块要复制的字节数量。
                copySize = in.size() - startIndex > everySize ? everySize :
in.size() - startIndex;
            }


            long end = System.currentTimeMillis();
            source.close();
            target.close();
            System.out.println("用时: " + (end - start) + " 毫秒");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.3.3 ServerSocketChannel和SocketChannel创建连接

- **服务器端：ServerSocketChannel**类用于连接的服务器端，它相当于：ServerSocket。

  1). 调用ServerSocketChannel的静态方法open()：打开一个通道，新频道的套接字最初未绑定; 必须通过其套接字的bind方法将其绑定到特定地址，才能接受连接。

```
ServerSocketChannel serverChannel = ServerSocketChannel.open()
```

2). 调用ServerSocketChannel的实例方法bind(SocketAddress add)：绑定本机监听端口，准备接受连接。

注：java.net.SocketAddress(抽象类)：代表一个Socket地址。

我们可以使用它的子类：java.net.InetSocketAddress(类)

构造方法：InetSocketAddress(int port)：指定本机监听端口。

```
serverChannel.bind(new InetSocketAddress(8888));
```

3). 调用ServerSocketChannel的实例方法accept()：等待连接。

```
SocketChannel accept = serverChannel.accept();
System.out.println("后续代码...");
```

示例：服务器端等待连接(默认-阻塞模式)

```
public class Server {
    public static void main(String[] args) {
        try (ServerSocketChannel serverChannel = ServerSocketChannel.open())
{
            serverChannel.bind(new InetSocketAddress(8888));
            System.out.println("【服务器】等待客户端连接...");
            SocketChannel accept = serverChannel.accept();
            System.out.println("后续代码.....");
            ......
            ......
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行后结果：

```
【服务器】等待客户端连接...
```

我们可以通过ServerSocketChannel的configureBlocking(boolean b)方法设置accept()是否阻塞

```
public class Server {
    public static void main(String[] args) throws Exception {
        try (ServerSocketChannel serverChannel = ServerSocketChannel.open())
{
            serverChannel.bind(new InetSocketAddress(8888));
            System.out.println("【服务器】等待客户端连接...");
       //   serverChannel.configureBlocking(true);//默认--阻塞
            serverChannel.configureBlocking(false);//非阻塞
            SocketChannel accept = serverChannel.accept();
```

```
            System.out.println("后续代码......");
            //......
            //......
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

运行后结果：

```
【服务器】等待客户端连接...
后续代码......
```

可以看到，accept()方法并没有阻塞，而是直接执行后续代码，返回值为null。

这种非阻塞的方式，通常用于"客户端"先启动，"服务器端"后启动，来查看是否有客户端连接，有，则接受连接；没有，则继续工作。

- **客户端：SocketChannel**类用于连接的客户端，它相当于：Socket。

  1). 先调用SocketChannel的open()方法打开通道：

```
ServerSocketChannel serverChannel = ServerSocketChannel.open()
```

  2). 调用SocketChannel的实例方法connect(SocketAddress add)连接服务器，默认：阻塞

```
socket.connect(new InetSocketAddress("localhost", 8888));
```

  示例：客户端连接服务器：

```
public class Client {
    public static void main(String[] args) {
        try (SocketChannel socket = SocketChannel.open()) {
            socket.connect(new InetSocketAddress("localhost", 8888));
            System.out.println("后续代码......")
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("客户端完毕！");
    }
}
```

SocketChannel默认是"阻塞"的，表现是：connect()方法会尝试几次连接，比较慢，如果连接失败，会抛出异常。

可以调用SocketChannel的configureBlocking(false)方法，设置为"非阻塞"。

```
public class Client {
    public static void main(String[] args) {
        try (SocketChannel socket = SocketChannel.open()) {
            socket.configureBlocking(false);//设置为"非阻塞"
            boolean b = socket.connect(new InetSocketAddress("localhost",
8888));
            System.out.println(b);
          System.out.println("后续代码......")
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("客户端完毕！");
    }
}
```

"非阻塞"情况下，表现是：connect()方法会立即返回，如果建立连接会返回true，如果连接失败，或者此通道处于非阻塞模式并且连接操作正在进行中，会返回false。

下面看一个完整的例子：先启动"客户端"，客户端采用"阻塞"模式反复尝试连接，后启动服务器端，服务器端采用"非阻塞"模式，如果没有客户端连接，就等待500毫秒，重新等待。

客户端：采用默认"阻塞"模式：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;

public class Client {
    public static void main(String[] args) {
        while (true) {
            try (SocketChannel socket = SocketChannel.open()) {
                System.out.println("客户端连接服务器......");
                socket.connect(new InetSocketAddress("localhost", 8888));
                System.out.println("客户端连接成功！！");
                break;
            } catch (IOException e) {
                System.out.println("连接失败，重连......");
            }
        }
    }
}
```

服务器端：采用"非阻塞"模式：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;

public class Server {
    public static void main(String[] args) {
        try (ServerSocketChannel serverChannel = ServerSocketChannel.open()) {
            serverChannel.bind(new InetSocketAddress(8888));
            System.out.println("【服务器】等待客户端连接...");
            serverChannel.configureBlocking(false);//非阻塞
            while (true) {
```

```
                SocketChannel accept = serverChannel.accept();
                if (accept != null) {
                    System.out.println("【服务器】收到连接...");
                    break;
                }else{
                    System.out.println("【服务器】继续等待...");
                    Thread.sleep(500);
                }
                System.out.println("后续代码......");
                //......
                //......
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

上例无论哪一端先启动，都会实现连接。

5.3.4 ServlerSocketChannel和SocketChannel收发信息

接下来我们看一下客户端和服务器端实现信息交互的过程。

- 创建服务器端如下：

```
public class Server {
    public static void main(String[] args) {
        try (ServerSocketChannel serverChannel = ServerSocketChannel.open())
{
            serverChannel.bind(new InetSocketAddress("localhost", 8888));
            System.out.println("【服务器】等待客户端连接...");
            SocketChannel accept = serverChannel.accept();
            System.out.println("【服务器】有连接到达...");
            //1.先发一条
            ByteBuffer outBuffer = ByteBuffer.allocate(100);
            outBuffer.put("你好客户端，我是服务器".getBytes());
            outBuffer.flip();//limit设置为position,position设置为0
            accept.write(outBuffer);//输出从position到limit之间的数据

            //2.再收一条，不确定字数是多少，但最多是100字节。先准备100字节空间
            ByteBuffer inBuffer = ByteBuffer.allocate(100);
            accept.read(inBuffer);
            inBuffer.flip();//limit设置为position,position设置为0
            String msg = new String(inBuffer.array(),0,inBuffer.limit());
            System.out.println("【服务器】收到信息：" + msg);
            accept.close();


        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- 创建客户端如下：

```
public class Client {
    public static void main(String[] args) {
        try (SocketChannel socket = SocketChannel.open()) {
            socket.connect(new InetSocketAddress("localhost", 8888));

            //1.先发一条
            ByteBuffer buf = ByteBuffer.allocate(100);
            buf.put("你好服务器，我是客户端".getBytes());
            buf.flip();//limit设置为position,position设置为0
            socket.write(buf);//输出从position到limit之间的数据

            //2.再收一条，不确定字数是多少，但最多是100字节。先准备100字节空间
            ByteBuffer inBuffer = ByteBuffer.allocate(100);
            socket.read(inBuffer);
            inBuffer.flip();//limit设置为position,position设置为0
            String msg = new String(inBuffer.array(),0,inBuffer.limit());
            System.out.println("【客户端】收到信息：" + msg);
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("客户端完毕！");
    }
}
```

- 服务器端打印结果：

```
【服务器】等待客户端连接...
【服务器】有连接到达...
【服务器】收到信息：你好服务器，我是客户端
```

- 客户端打印结果：

```
【客户端】收到信息：你好客户端，我是服务器
客户端完毕！
```

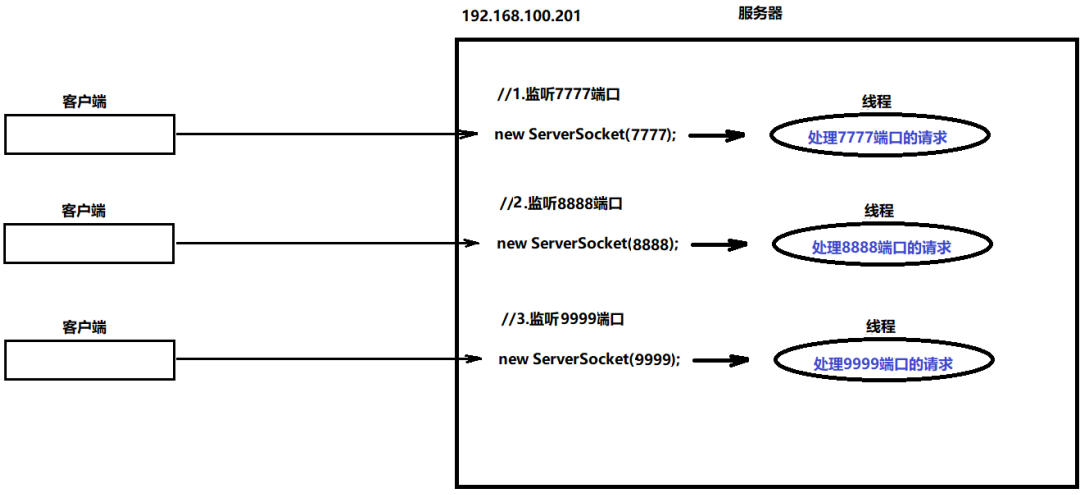
5.4 Selector(选择器)

5.4.1 多路复用的概念

选择器Selector是NIO中的重要技术之一。它与SelectableChannel联合使用实现了非阻塞的多路复用。使用它可以节省CPU资源，提高程序的运行效率。

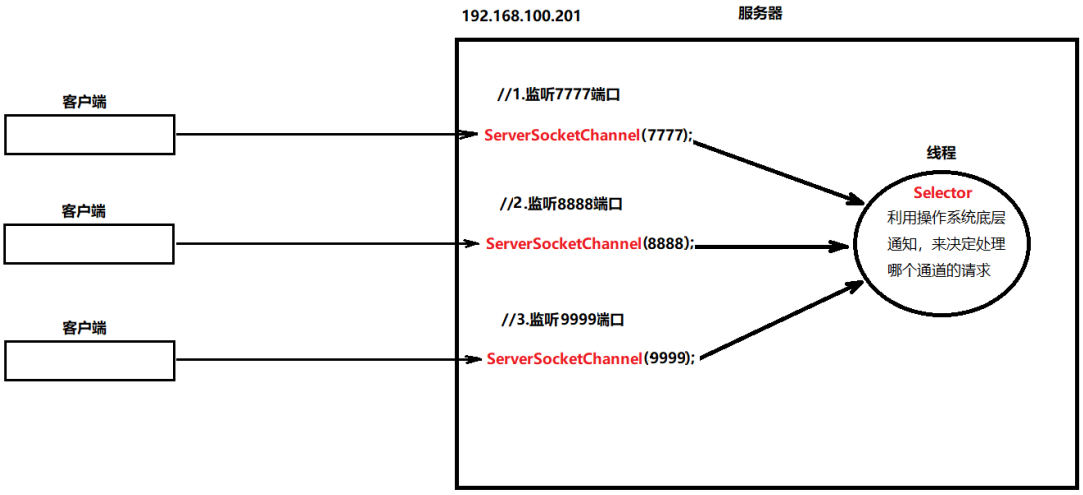
"多路"是指：服务器端同时监听多个"端口"的情况。每个端口都要监听多个客户端的连接。

- 服务器端的非多路复用效果



如果不使用“多路复用”，服务器端需要开很多线程处理每个端口的请求。如果在高并发环境下，造成系统性能下降。

- 服务器端的多路复用效果



使用了多路复用，只需要一个线程就可以处理多个通道，降低内存占用率，减少CPU切换时间，在高并发、高频段业务环境下有非常重要的优势

5.4.2 选择器Selector

Selector被称为：选择器，也被称为：多路复用器，它可以注册到很多个Channel上，监听各个Channel上发生的事件，并且能够根据事件情况决定Channel读写。这样，通过一个线程管理多个Channel，就可以处理大量网络连接了。

有了Selector，我们就可以利用一个线程来处理所有的channels。线程之间的切换对操作系统来说代价是很高的，并且每个线程也会占用一定的系统资源。所以，对系统来说使用的线程越少越好。

- **如何创建一个Selector**

Selector 就是您注册对各种 I/O 事件兴趣的地方，而且当那些事件发生时，就是这个对象告诉您所发生的事件。

```
Selector selector = Selector.open();
```

- **注册Channel到Selector**

为了能让Channel和Selector配合使用，我们需要把Channel注册到Selector上。通过调用channel.register（）方法来实现注册：

```
channel.configureBlocking(false);
SelectionKey key =channel.register(selector,SelectionKey.OP_READ);
```

注意，注册的Channel 必须设置成异步模式才可以,否则异步IO就无法工作，这就意味着我们不能把一个FileChannel注册到Selector，因为FileChannel没有异步模式，但是网络编程中的SocketChannel是可以的。

register()方法的第二个参数：是一个int值，意思是在**通过Selector监听Channel时对什么事件感兴趣。**可以监听四种不同类型的事件，而且可以使用SelectionKey的四个常量表示：

1. 连接就绪--常量：SelectionKey.OP_CONNECT
2. 接收就绪--常量：SelectionKey.OP_ACCEPT (ServerSocketChannel在注册时只能使用此项)
3. 读就绪--常量：SelectionKey.OP_READ
4. 写就绪--常量：SelectionKey.OP_WRITE

注意：对于ServerSocketChannel在注册时，只能使用OP_ACCEPT，否则抛出异常。

- 示例：下面的例子，服务器创建3个通道，同时监听3个端口，并将3个通道注册到一个选择器中

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;

public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();

        //注册三个通道
        channelA.register(selector, SelectionKey.OP_ACCEPT);
        channelB.register(selector, SelectionKey.OP_ACCEPT);
        channelC.register(selector, SelectionKey.OP_ACCEPT);
    }
}
```

接下来，就可以通过选择器selector操作三个通道了。

5.4.3 多路连接

- **Selector的keys()方法**
  - 此方法返回一个Set集合，表示：已注册通道的集合。每个已注册通道封装为一个SelectionKey对象。
- **Selector的selectedKeys()方法**
  - 此方法返回一个Set集合，表示：当前已连接的通道的集合。每个已连接通道同一封装为一个SelectionKey对象。
- **Selector的select()方法**
  - 此方法会阻塞，直到有至少1个客户端连接。
  - 此方法会返回一个int值，表示有几个客户端连接了服务器。
- **示例：** 客户端：启动两个线程，模拟两个客户端，同时连接服务器的7777和8888端口：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;

public class Client {
    public static void main(String[] args) {
        new Thread(()->{
            while (true) {
                try (SocketChannel socket = SocketChannel.open()) {
                    System.out.println("7777客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
7777));
                    System.out.println("7777客户端连接成功....");
                    break;
                } catch (IOException e) {
                    System.out.println("7777异常重连");
                }
            }
        }).start();

        new Thread(()->{
            while (true) {
                try (SocketChannel socket = SocketChannel.open()) {
                    System.out.println("8888客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
8888));
                    System.out.println("8888客户端连接成功....");
                    break;
                } catch (IOException e) {
                    System.out.println("8888异常重连");
                }
            }
        }).start();
    }
}
```

服务器端：

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
```

```
public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();

        //注册三个通道
        channelA.register(selector, SelectionKey.OP_ACCEPT);
        channelB.register(selector, SelectionKey.OP_ACCEPT);
        channelC.register(selector, SelectionKey.OP_ACCEPT);

        Set<SelectionKey> keys = selector.keys();//获取已注册通道的集合
        System.out.println("注册通道数量：" + keys.size());
        Set<SelectionKey> selectionKeys = selector.selectedKeys();//获取已连接通
道的集合
        System.out.println("已连接的通道数量：" + selectionKeys.size());
        System.out.println("-----------------------------------------------");

        System.out.println("【服务器】等待连接......");
        int selectedCount = selector.select();//此方法会"阻塞"
        System.out.println("连接数量：" + selectedCount);

      System.out.println("-----------------------------------------------");
        Set<SelectionKey> keys1 = selector.keys();
      System.out.println("注册通道数量：" + keys1.size());
        Set<SelectionKey> selectionKeys1 = selector.selectedKeys();
      System.out.println("已连接的通道数量：" + selectionKeys1.size());
    }
}
```

测试采用两种运行方式：

1. 先启动服务器，再启动客户端。会看到"服务器端"打印：

```
注册通道数量：3
已连接的通道数量：0
-----------------------------------------------
 【服务器】等待连接......
连接数量：1
-----------------------------------------------
 注册通道数量：3
 已连接的通道数量：1
```

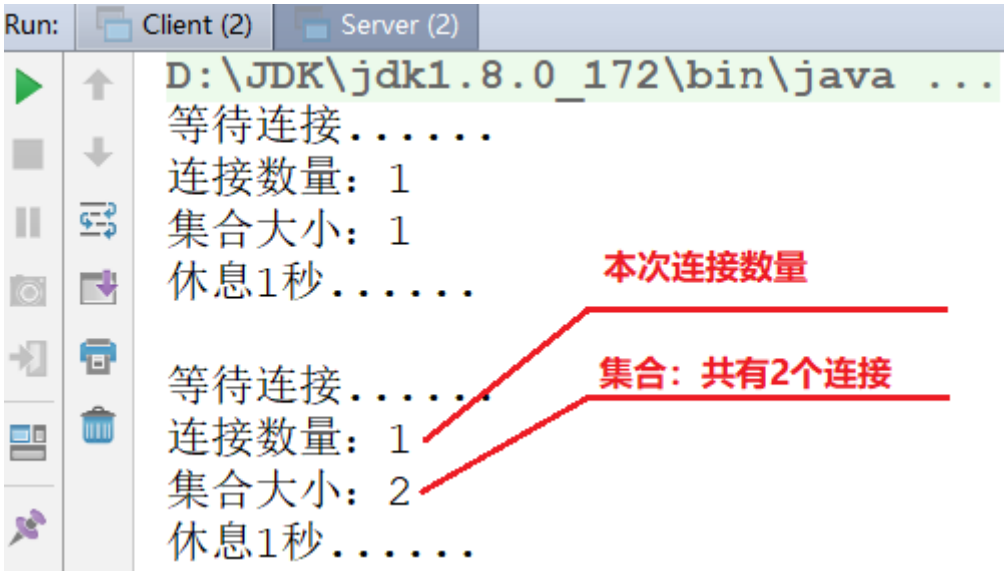
2. 先启动客户端，再启动服务器。会看到"服务器端"打印：

```
注册通道数量：3
已连接的通道数量：0
------------------------------------------------
【服务器】等待连接......
连接数量：2
------------------------------------------------
 注册通道数量：3
已连接的通道数量：2
```

在**"服务器端"**加入循环，确保接收到每个通道的连接：(下面的代码去掉了一些测试代码)

```
    public static void main(String[] args) throws Exception {
          //创建3个通道，同时监听3个端口
            ServerSocketChannel channelA = ServerSocketChannel.open();
            channelA.configureBlocking(false);
            channelA.bind(new InetSocketAddress(7777));

            ServerSocketChannel channelB = ServerSocketChannel.open();
            channelB.configureBlocking(false);
            channelB.bind(new InetSocketAddress(8888));

            ServerSocketChannel channelC = ServerSocketChannel.open();
            channelC.configureBlocking(false);
            channelC.bind(new InetSocketAddress(9999));

            //获取选择器
            Selector selector = Selector.open();

            //注册三个通道
            channelA.register(selector, SelectionKey.OP_ACCEPT);
            channelB.register(selector, SelectionKey.OP_ACCEPT);
            channelC.register(selector, SelectionKey.OP_ACCEPT);

            while(true) {
                System.out.println("等待连接......");
                int selectedCount = selector.select();
                System.out.println("连接数量：" + selectedCount);
                //获取已连接的通道对象
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                System.out.println("集合大小：" + selectionKeys.size());

                System.out.println("休息1秒......");
                Thread.sleep(1000);
                System.out.println();//打印一个空行

            }

        }
```

先运行"服务器"，再运行"客户端"。服务器打印如下：



注意：此例会有一个问题——服务器端第一次select()会阻塞，获取到一次连接后再次循环时，select()将不会再阻塞，从而造成死循环，所以这里加了一个sleep()，这个我们后边解决！！！

接下来，我们获取"已连接通道"的集合，并遍历：

客户端

> 使用上例的客户端即可

服务器端

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.util.Set;

public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();

        //注册三个通道
        channelA.register(selector, SelectionKey.OP_ACCEPT);
        channelB.register(selector, SelectionKey.OP_ACCEPT);
```

```
            channelC.register(selector, SelectionKey.OP_ACCEPT);

            while(true) {
                System.out.println("等待连接......");
                int selectedCount = selector.select();
                System.out.println("连接数量: " + selectedCount);
                //获取已连接的通道对象
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                System.out.println("集合大小: " + selectionKeys.size());

                //遍历已连接通道的集合
                Iterator<SelectionKey> it = selectionKeys.iterator();
                while (it.hasNext()) {
                    //获取当前连接通道的SelectionKey
                    SelectionKey key = it.next();
                    //从SelectionKey中获取通道对象
                    ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                    //看一下此通道是监听哪个端口的
                    System.out.println("监听端口: " + channel.getLocalAddress());
                }
                System.out.println("休息1秒......");
                Thread.sleep(1000);
                System.out.println();//打印一个空行

            }

        }
    }
```

- 先启动服务器端，再启动客户端。可以看到"服务器端"如下打印：

```
等待连接......
连接数量：1
集合大小：1
监听端口：/0:0:0:0:0:0:0:0:8888
休息1秒......

等待连接......
连接数量：1
集合大小：2
监听端口：/0:0:0:0:0:0:0:0:8888
监听端口：/0:0:0:0:0:0:0:0:7777
休息1秒......
```

- **关于SelectionKey**

```
- 当一个"通道"注册到选择器Selector后，选择器Selector内部就创建一个SelectionKey对象，里面封装了这个通道和这个选择器的映射关系。
- 通过SelectionKey的channel()方法，可以获取它内部的通道对象。
```

- **解决select()不阻塞，导致服务器端死循环的问题**

- 原因：在将"通道"注册到"选择器Selector"时，我们指定了关注的事件SelectionKey.OP_ACCEPT，而我们获取到管道对象后，并没有处理这个事件，所以导致select()方法一直循环。
- 解决：处理SelectionKey.OP_ACCEPT事件

  更改服务器端代码

```
public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ...略...

        //获取选择器
        ...略...

        //注册三个通道
        ...略...

        while(true) {
            ......
            ......
            while (it.hasNext()) {
                //获取当前连接通道的SelectionKey
                SelectionKey key = it.next();
                //从SelectionKey中获取通道对象
                ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                //看一下此通道是监听哪个端口的
                System.out.println("监听端口：" + channel.getLocalAddress());
                SocketChannel accept = channel.accept();//处理accept事件(非阻
塞)
            }
        }

    }
}
```

  现在我们的服务器端可以很好的接收客户端连接了，但还有一个小问题，在接下来的互发信息的例子中我们可以看到这个问题并解决它。

5.4.4 多路信息接收

- 服务器端代码：

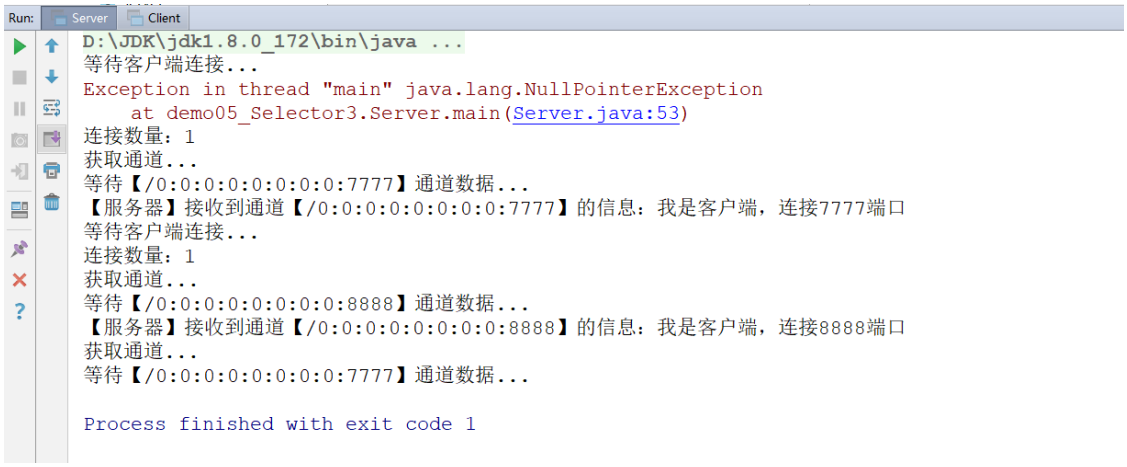
```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

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

```
        //1.同时监听三个端口：7777,8888,9999
        ServerSocketChannel serverChannel1 = ServerSocketChannel.open();
        serverChannel1.bind(new InetSocketAddress(7777));
        serverChannel1.configureBlocking(false);

        ServerSocketChannel serverChannel2 = ServerSocketChannel.open();
        serverChannel2.bind(new InetSocketAddress(8888));
        serverChannel2.configureBlocking(false);

        ServerSocketChannel serverChannel3 = ServerSocketChannel.open();
        serverChannel3.bind(new InetSocketAddress(9999));
        serverChannel3.configureBlocking(false);

        //2.获取一个选择器
        Selector selector = Selector.open();

        //3.注册三个通道
        SelectionKey key1 = serverChannel1.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key2 = serverChannel2.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key3 = serverChannel3.register(selector,
SelectionKey.OP_ACCEPT);

        //4.循环监听三个通道
        while (true) {
            System.out.println("等待客户端连接...");
            int keyCount = selector.select();
            System.out.println("连接数量：" + keyCount);

            //遍历已连接的每个通道的SelectionKey
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> it = keys.iterator();
            while (it.hasNext()) {
                SelectionKey nextKey = it.next();
                System.out.println("获取通道...");
                ServerSocketChannel channel = (ServerSocketChannel)
nextKey.channel();
                System.out.println("等待【" + channel.getLocalAddress() + "】
通道数据...");
                SocketChannel socketChannel = channel.accept();
                //接收数据
                ByteBuffer inBuf = ByteBuffer.allocate(100);
                socketChannel.read(inBuf);
                inBuf.flip();
                String msg = new String(inBuf.array(), 0, inBuf.limit());
                System.out.println("【服务器】接收到通道【" +
channel.getLocalAddress() + "】的信息：" + msg);

            }
        }
    }
}
```

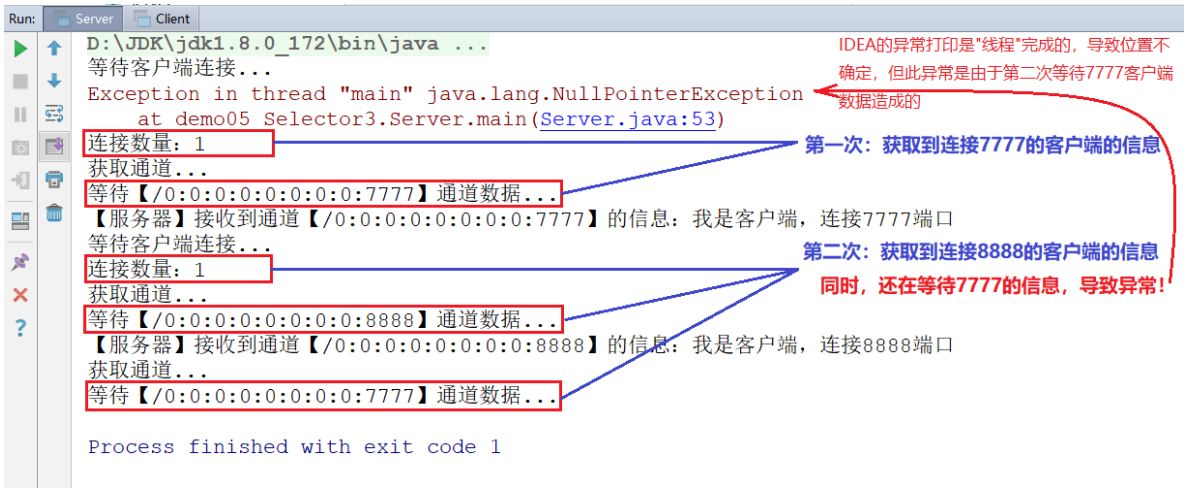
- 客户端代码：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class Client {
    public static void main(String[] args) throws InterruptedException {
        //两个线程，模拟两个客户端，分别连接服务器的7777,8888端口
        new Thread(()->{
            while (true) {
                try(SocketChannel socket = SocketChannel.open())  {

                    System.out.println("7777客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
7777));
                    System.out.println("7777客户端连接成功....");
                    //发送信息
                    ByteBuffer outBuf = ByteBuffer.allocate(100);
                    outBuf.put("我是客户端，连接7777端口".getBytes());
                    outBuf.flip();
                    socket.write(outBuf);
                    break;
                } catch (IOException e) {
                    System.out.println("7777异常重连");
                }
            }
        }).start();
        new Thread(()->{
            while (true) {
                try(SocketChannel socket = SocketChannel.open())  {
                    System.out.println("8888客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
8888));
                    System.out.println("8888客户端连接成功....");
                    //发送信息
                    ByteBuffer outBuf = ByteBuffer.allocate(100);
                    outBuf.put("我是客户端，连接8888端口".getBytes());
                    outBuf.flip();
                    socket.write(outBuf);
                    break;
                } catch (IOException e) {
                    System.out.println("8888异常重连");
                }
            }
        }).start();
    }
}
```

先启动服务器，再启动客户端，打印结果：



可以看到，出现了异常，为什么会这样？



问题就出现在获取selectedKeys()的集合。

- 第一次的7777连接，selectedKeys()获取的集合中只有一个SelectionKey对象。
- 第二次的8888连接，selectedKeys()获取的集合中有2个SelectionKey对象，一个是连接7777客户端的，另一个是连接8888客户端的。而此时应该只处理连接8888客户端的，所以在上一次处理完7777的数据后，应该将其SelectionKey对象移除。

更改服务器端代码：

```
public class Server {
    public static void main(String[] args) throws Exception {
        //1.同时监听三个端口：7777,8888,9999
        ServerSocketChannel serverChannel1 = ServerSocketChannel.open();
        serverChannel1.bind(new InetSocketAddress(7777));
        serverChannel1.configureBlocking(false);

        ServerSocketChannel serverChannel2 = ServerSocketChannel.open();
        serverChannel2.bind(new InetSocketAddress(8888));
        serverChannel2.configureBlocking(false);

        ServerSocketChannel serverChannel3 = ServerSocketChannel.open();
        serverChannel3.bind(new InetSocketAddress(9999));
        serverChannel3.configureBlocking(false);

        //2.获取一个选择器
        Selector selector = Selector.open();
```

```
        //3.注册三个通道
        SelectionKey key1 = serverChannel1.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key2 = serverChannel2.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key3 = serverChannel3.register(selector,
SelectionKey.OP_ACCEPT);

        //4.循环监听三个通道
        while (true) {
            System.out.println("等待客户端连接...");
            int keyCount = selector.select();
            System.out.println("连接数量: " + keyCount);

            //遍历已连接的每个通道的SelectionKey
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> it = keys.iterator();
            while (it.hasNext()) {
                SelectionKey nextKey = it.next();
                System.out.println("获取通道...");
                ServerSocketChannel channel = (ServerSocketChannel)
nextKey.channel();
                System.out.println("等待【" + channel.getLocalAddress() + "】通道
数据...");
                SocketChannel socketChannel = channel.accept();
                //接收数据
                ByteBuffer inBuf = ByteBuffer.allocate(100);
                socketChannel.read(inBuf);
                inBuf.flip();
                String msg = new String(inBuf.array(), 0, inBuf.limit());
                System.out.println("【服务器】接收到通道【" +
channel.getLocalAddress() + "】的信息: " + msg);
                //移除此SelectionKey
                it.remove();
            }
        }
    }
}
```

测试：先启动服务器，再启动客户端，可以正常接收客户端数据了(客户端可以再添加一个线程连接9999端口)。

5.5 NIO2-AIO(异步、非阻塞)

5.5.1 AIO概述

AIO是异步IO的缩写，虽然NIO在网络操作中，提供了非阻塞的方法，但是NIO的IO行为还是同步的。对于NIO来说，我们的业务线程是在IO操作准备好时，得到通知，接着就由这个线程自行进行IO操作，IO操作本身是同步的。

但是对AIO来说，则更加进了一步，它不是在IO准备好时再通知线程，而是在IO操作已经完成后，再给线程发出通知。因此AIO是不会阻塞的，此时我们的业务逻辑将变成一个回调函数，等待IO操作完成后，由系统自动触发。

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。 即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。 在JDK1.7中，这部分内容被称作NIO.2，主要在Java.nio.channels包下增加了下面四个异步通道：

- AsynchronousSocketChannel
- AsynchronousServerSocketChannel
- AsynchronousFileChannel
- AsynchronousDatagramChannel

在AIO socket编程中，服务端通道是AsynchronousServerSocketChannel，这个类提供了一个open()静态工厂，一个bind()方法用于绑定服务端IP地址（还有端口号），另外还提供了accept()用于接收用户连接请求。在客户端使用的通道是AsynchronousSocketChannel,这个通道处理提供open静态工厂方法外，还提供了read和write方法。

在AIO编程中，发出一个事件（accept read write等）之后要指定事件处理类（回调函数），AIO中的事件处理类是CompletionHandler<V,A>，这个接口定义了如下两个方法，分别在异步操作成功和失败时被回调。

void completed(V result, A attachment);

void failed(Throwable exc, A attachment);

5.5.2 AIO 服务器端代码示例

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.util.concurrent.Future;

public class Server {
    public static void main(String[] args) throws Exception {
        AsynchronousServerSocketChannel serverSocketChannel =
AsynchronousServerSocketChannel.open();
        serverSocketChannel.bind(new InetSocketAddress(8888));
        Future<AsynchronousSocketChannel> accept;
        while (true) {
            accept = serverSocketChannel.accept();// accept()不会阻塞。
            System.out.println("=================");
            System.out.println("服务器等待连接...");
            AsynchronousSocketChannel socketChannel = accept.get(); //get()方法将
阻塞。

            System.out.println("服务器接受连接");
            System.out.println("服务器与" + socketChannel.getRemoteAddress() + "建
立连接");
            ByteBuffer buffer = ByteBuffer.wrap("你好客户端，我是服务
器".getBytes());
            Future<Integer> write = socketChannel.write(buffer);
            while (!write.isDone()) {//判断数据是否已发出。如果当前网络堵塞，尝试循环等
待
                Thread.sleep(10);
            }
            System.out.println("服务器发送数据完毕.");
            socketChannel.close();
```

```
        }
    }

}
```

5.5.3 AIO 客户端代码示例

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.util.concurrent.Future;

public class Client {
    public static void main(String[] args) {
        AsynchronousSocketChannel socketChannel = null;
        try {
            socketChannel = AsynchronousSocketChannel.open();

            Future<Void> connect = socketChannel.connect(new
InetSocketAddress("localhost", 8888));//不阻塞
            while (!connect.isDone()) {//判断是否连接完毕
                Thread.sleep(10);//如果连接未完成，等待10毫秒。
            }
            System.out.println("建立连接" + socketChannel.getRemoteAddress());
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            Future<Integer> read = socketChannel.read(buffer);
            while (!read.isDone()) {//判断数据是否已接收完毕。如果当前网络堵塞，尝试循环
等待
                Thread.sleep(10);
            }
            System.out.println("接收服务器数据:" + new String(buffer.array(), 0,
read.get()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```