一 网络编程

课程介绍

在Java的软件设计开发中，通信架构是不可避免的，我们在进行不同系统或者不同进程之间的数据交互，或者在高并发下的通信场景下都需要用到网络通信相关的技术，对于一些经验丰富的程序员来说，Java早期的网络通信架构存在一些缺陷，其中最令人恼火的是基于性能低下的同步阻塞式的I/O通信（BIO），随着互联网开发下通信性能的高要求，Java在2002年开始支持了非阻塞式的I/O通信技术(NIO)。大多数读者在学习网络通信相关技术的时候，都只是接触到零碎的通信技术点，没有完整的技术体系架构，以至于对于Java的通信场景总是没有清晰的解决方案。本次课程将通过大量清晰直接的案例从最基础的BIO式通信开始介绍到NIO , AIO，读者可以清晰的了解到阻塞、同步、异步的现象、概念和特征以及优缺点。最终讲解到现今最流行的NIO框架Netty，Netty是很多开源软件如Dubbo，RocketMQ等的底层技术，Netty可用于构建高性能，分布式通信能力，尤其擅长高并发场景下通信性能的处理，是现今企业高性能通信下最优的解决方案之一，读者将从本课程中得到完整的通信体系技术栈，了解高性能网络编程的前世今生，轻松应对NIO技术的高薪面试，也为深入掌握Java高级通信技术打好坚实功底。

第一章 Linux网络IO模型

1.1 同步和异步，阻塞和非阻塞

a.同步和异步关注的是结果消息的通信机制

同步:同步的意思就是调用方需要主动等待结果的返回

异步:异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。

b.阻塞和非阻塞主要关注的是等待结果返回调用方的状态

阻塞:是指结果返回之前，当前线程被挂起，不做任何事

非阻塞:是指结果在返回之前，线程可以做一些其他事，不会被挂起。

1.2 组合起来

1.2.1 同步阻塞:

同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就在店里面一直等，期间不做任何事(包括看手机)，等着商家进货，直到有货为止，这个效率很低。

1.2.2 同步非阻塞:

同步非阻塞在编程中可以抽象为一个轮询模式，你去了商店之后，发现衣服卖完了，这个时候不需要傻傻的等着，你可以去其他地方比如奶茶店，买杯水，但是你还是需要时不时的去商店问老板新衣服到了吗。

1.2.3 异步阻塞:

异步阻塞这个编程里面用的较少，有点类似你写了个线程池,submit然后马上future.get()，这样线程其实还是挂起的。有点像你去商店买衣服，这个时候发现衣服没有了，这个时候你就给老板留给电话，说衣服到了就给我打电话，然后你就守着这个电话，一直等着他响什么事也不做。这样感觉的确有点傻，所以这个模式用得比较少。

1.2.4 异步非阻塞:

好比你去商店买衣服，衣服没了，你只需要给老板说这是我的电话，衣服到了就打。然后你就随心所欲的去玩，也不用操心衣服什么时候到，衣服一到，电话一响就可以去买衣服了。

1.3 5种I/O模型

1.3.1 阻塞I/O模型：

应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待....数据准备好了，从内核拷贝到用户空间,IO函数返回成功指示。当调用recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用recv()函数时，未必用户空间就已经存在数据，那么此时recv()函数就会处于等待状态。

1.3.2 非阻塞IO模型

我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。上述模型绝不被推荐。

把SOCKET设置为非阻塞模式，即通知系统内核：在调用Windows Sockets API时，不要让线程睡眠，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用recv()函数的过程。前三次调用recv()函数时，内核数据还没有准备好。因此，该函数立即返回WSAEWOULDBLOCK错误代码。第四次调用recv()函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv()函数返回成功指示，应用程序开始处理数据。

1.3.3 IO复用模型：

简介：主要是select和epoll；对一个IO端口，两次调用，两次返回，比阻塞IO并没有什么优越性；关键是能实现同时对多个IO端口进行监听；

I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。

当用户进程调用了select，那么整个进程会被block；而同时，kernel会“监视”所有select负责的socket；当任何一个socket中的数据准备好了，select就会返回。这个时候，用户进程再调用read操作，将数据从kernel拷贝到用户进程。 这个图和blocking IO的图其实并没有太大的不同，事实上还更差一些。因为这里需要使用两个系统调用(select和recvfrom)，而blocking IO只调用了一个系统调用(recvfrom)。但是，用select的优势在于它可以同时处理多个connection。（select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

1.3.4 信号驱动IO

简介：两次调用，两次返回；

首先我们允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

1.3.5 异步IO模型

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作

1.3.6 5个I/O模型的比较

select、poll、epoll的区别？** ：

1、支持一个进程所能打开的最大连接数

select	**单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是3232，*同理64位机器上FD_SETSIZE为3264*），可以对进行修改，然后重新编译内核，但是性能可能会受到影响。**
poll	poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的
epoll	连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接

2、FD剧增后带来的IO效率问题

select	**因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。**
poll	同上
epoll	因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select	**内核需要将消息传递到用户空间，都需要内核拷贝动作**
poll	同上
epoll	epoll通过内核和用户空间共享一块内存来实现的。

第二章 各种IO通信模式演示

Java BIO (blocking I/O)： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要对应一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销。

2.1 传统的BIO编程演示

网络编程的基本模型是Client/Server模型，也就是两个进程之间进行相互通信，其中服务端提供位置信（绑定IP地址和端口），客户端通过连接操作向服务端监听的端口地址发起连接请求，基于TCP协议下进行三次握手连接，连接成功后，双方通过网络套接字（Socket）进行通信。

传统的同步阻塞模型开发中，服务端ServerSocket负责绑定IP地址，启动监听端口；客户端Socket负责发起连接操作。连接成功后，双方通过输入和输出流进行同步阻塞式通信。 基于BIO模式下的通信，客户端 - 服务端是完全同步，完全耦合的。

客户端案例如下

```
package com.itheima._02bio01;

import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
/**
    目标：Socket网络编程。

    Java提供了一个包：java.net下的类都是用于网络通信。
    Java提供了基于套接字（端口）Socket的网络通信模式，我们基于这种模式就可以直接实现TCP通信。
    只要用Socket通信，那么就是基于TCP可靠传输通信。

    功能1：客户端发送一个消息，服务端接口一个消息，通信结束！！

    创建客户端对象：
          （1）创建一个Socket的通信管道，请求与服务端的端口连接。
          （2）从Socket管道中得到一个字节输出流。
          （3）把字节流改装成自己需要的流进行数据的发送
    创建服务端对象：
          （1）注册端口
          （2）开始等待接收客户端的连接，得到一个端到端的Socket管道
          （3）从Socket管道中得到一个字节输入流。
          （4）把字节输入流包装成自己需要的流进行数据的读取。

    Socket的使用：
        构造器：public Socket(String host, int port)
        方法：   public OutputStream getOutputStream()：获取字节输出流
                 public InputStream getInputStream() :获取字节输入流

    ServerSocket的使用：
        构造器：public ServerSocket(int port)

    小结：
        通信是很严格的，对方怎么发你就怎么收，对方发多少你就只能收多少！！

 */
public class ClientDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("==客户端的启动==");
        //  （1）创建一个Socket的通信管道，请求与服务端的端口连接。
        Socket socket = new Socket("127.0.0.1",8888);
```

```
        //  (2) 从Socket通信管道中得到一个字节输出流。
        OutputStream os = socket.getOutputStream();
        //  (3) 把字节流改装成自己需要的流进行数据的发送
        PrintStream ps = new PrintStream(os);
        //  (4) 开始发送消息
        ps.println("我是客户端，我想约你吃小龙虾！！！");
        ps.flush();
    }
}
```

服务端案例如下

```
package com.itheima._02bio01;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 服务端
 */
public class ServerDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("==服务器的启动==");
        //  (1) 注册端口
        ServerSocket serverSocket = new ServerSocket(8888);
        // (2) 开始在这里暂停等待接收客户端的连接,得到一个端到端的Socket管道
        Socket socket = serverSocket.accept();
        // (3) 从Socket管道中得到一个字节输入流。
        InputStream is = socket.getInputStream();
        // (4) 把字节输入流包装成自己需要的流进行数据的读取。
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        // (5) 读取数据
        String line ;
        while((line = br.readLine())!=null){
            System.out.println("服务端收到: "+line);
        }
    }
}
```

小结

- 在以上通信中，服务端会一致等待客户端的消息，如果客户端没有进行消息的发送，服务端将一直进入阻塞状态。
- 同时服务端是按照行获取消息的，这意味着客户端也必须按照行进行消息的发送，否则服务端将进入等待消息的阻塞状态！

2.2 BIO模式下多发和多收消息

在1.3的案例中，**只能实现客户端发送消息，服务端接收消息**，并不能实现反复的收消息和反复的发消息，我们只需要在客户端案例中，加上反复按照行发送消息的逻辑即可！案例代码如下：

客户端代码如下

```
package com.itheima._03bio02;

import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Scanner;

/**
    目标：Socket网络编程。

    功能1：客户端可以反复发消息，服务端可以反复收消息

    小结：
        通信是很严格的，对方怎么发你就怎么收，对方发多少你就只能收多少！！

 */
public class ClientDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("==客户端的启动==");
        //  (1) 创建一个Socket的通信管道，请求与服务端的端口连接。
        Socket socket = new Socket("127.0.0.1",8888);
        //  (2) 从Socket通信管道中得到一个字节输出流。
        OutputStream os = socket.getOutputStream();
        //  (3) 把字节流改装成自己需要的流进行数据的发送
        PrintStream ps = new PrintStream(os);
        //  (4) 开始发送消息
        Scanner sc = new Scanner(System.in);
        while(true){
            System.out.print("请说:");
            String msg = sc.nextLine();
            ps.println(msg);
            ps.flush();
        }
    }
}
```

服务端代码如下

```
package com.itheima._03bio02;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 服务端
 */
public class ServerDemo {
    public static void main(String[] args) throws Exception {
```

```
        String s = "886";
        System.out.println("886".equals(s));
        System.out.println("==服务器的启动==");
        // (1) 注册端口
        ServerSocket serverSocket = new ServerSocket(8888);
        // (2) 开始在这里暂停等待接收客户端的连接,得到一个端到端的Socket管道
        Socket socket = serverSocket.accept();
        // (3) 从Socket管道中得到一个字节输入流。
        InputStream is = socket.getInputStream();
        // (4) 把字节输入流包装成  自己需要的流进行数据的读取。
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        // (5) 读取数据
        String line ;
        while((line = br.readLine())!=null){
            System.out.println("服务端收到: "+line);
        }
    }
}
```

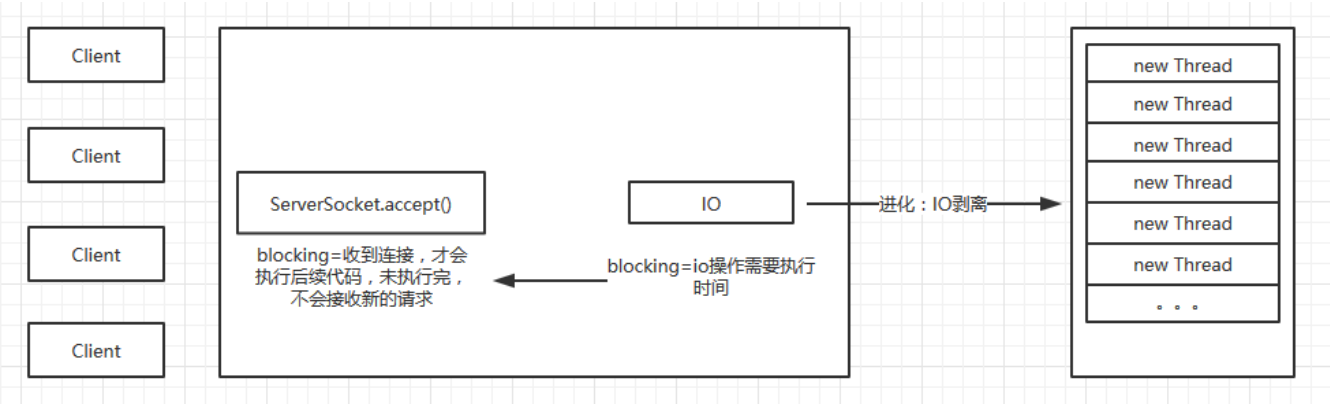
小结

- 本案例中确实可以实现客户端多发多收
- 但是服务端只能处理一个客户端的请求，因为服务端是单线程的。一次只能与一个客户端进行消息通信。

2.3 BIO模式下接受多个客户端

概述

在上述的案例中，一个服务端只能接收一个客户端的通信请求，**那么如果服务端需要处理很多个客户端的消息通信请求应该如何处理呢**，此时我们就需要在服务端引入线程了，也就是说客户端每发起一个请求，服务端就创建一个新的线程来处理这个客户端的请求，这样就实现了一个客户端一个线程的模型，图解模式如下：



客户端案例代码如下

```
/**
    目标：Socket网络编程。

    功能1：客户端可以反复发，一个服务端可以接收无数个客户端的消息！！

    小结：
        服务器如果想要接收多个客户端，那么必须引入线程，一个客户端一个线程处理！！
```

```
 */
public class ClientDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("==客户端的启动==");
        //  (1) 创建一个Socket的通信管道，请求与服务端的端口连接。
        Socket socket = new Socket("127.0.0.1",7777);
        //  (2) 从Socket通信管道中得到一个字节输出流。
        OutputStream os = socket.getOutputStream();
        //  (3) 把字节流改装成自己需要的流进行数据的发送
        PrintStream ps = new PrintStream(os);
        //  (4) 开始发送消息
        Scanner sc = new Scanner(System.in);
        while(true){
            System.out.print("请说:");
            String msg = sc.nextLine();
            ps.println(msg);
            ps.flush();
        }
    }
}
```

服务端案例代码如下

```
/**
    服务端
 */
public class ServerDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("==服务器的启动==");
        //  (1) 注册端口
        ServerSocket serverSocket = new ServerSocket(7777);
        while(true){
            // (2) 开始在这里暂停等待接收客户端的连接,得到一个端到端的Socket管道
            Socket socket = serverSocket.accept();
            new ServerReadThread(socket).start();
            System.out.println(socket.getRemoteSocketAddress()+"上线了! ");
        }
    }
}

class ServerReadThread extends Thread{
    private Socket socket;

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

    @Override
    public void run() {
        try{
            // (3) 从Socket管道中得到一个字节输入流。
            InputStream is = socket.getInputStream();
            // (4) 把字节输入流包装成自己需要的流进行数据的读取。
```

```
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            // (5) 读取数据
            String line ;
            while((line = br.readLine())!=null){
                System.out.println("服务端收到: "+socket.getRemoteSocketAddress()+":"+line);
            }
        }catch (Exception e){
            System.out.println(socket.getRemoteSocketAddress()+"下线了! ");
        }
    }
}
```

小结

- 1.每个Socket接收到，都会创建一个线程，线程的竞争、切换上下文影响性能；
- 2.每个线程都会占用栈空间和CPU资源；
- 3.并不是每个socket都进行IO操作，无意义的线程处理；
- 4.客户端的并发访问增加时。服务端将呈现1:1的线程开销，访问量越大，系统将发生线程栈溢出，线程创建失败，最终导致进程宕机或者僵死，从而不能对外提供服务。

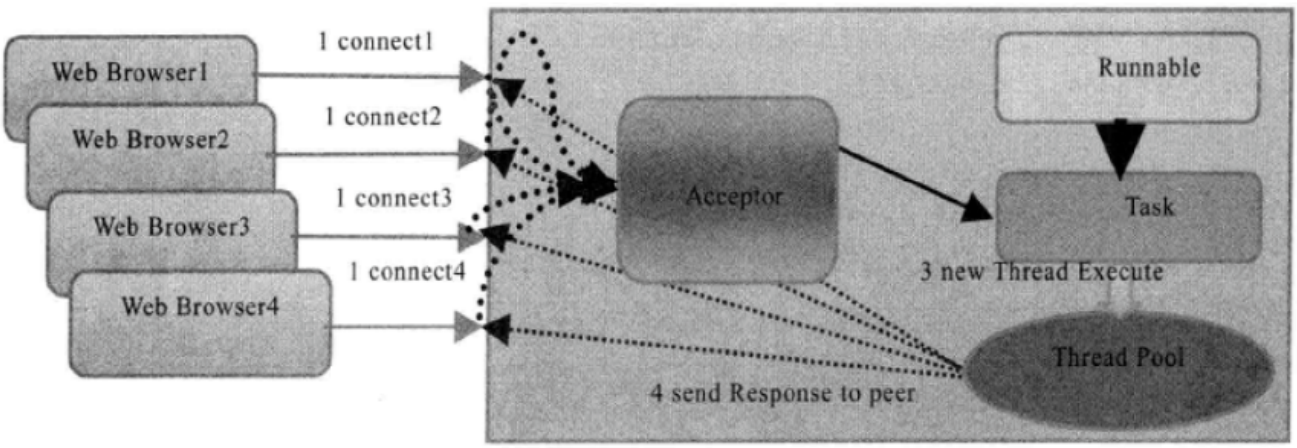
2.4 伪异步I/O编程

概述

在上述案例中：客户端的并发访问增加时。服务端将呈现1:1的线程开销，访问量越大，系统将发生线程栈溢出，线程创建失败，最终导致进程宕机或者僵死，从而不能对外提供服务。

接下来我们采用一个伪异步I/O的通信框架，采用线程池和任务队列实现，当客户端接入时，将客户端的Socket封装成一个Task(该任务实现java.lang.Runnable线程任务接口)交给后端的线程池中进行处理。JDK的线程池维护一个消息队列和N个活跃的线程，对消息队列中Socket任务进行处理，由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。

图示如下:



客户端源码分析

```
public class Client {
   public static void main(String[] args) {
      try {
```

```
            // 1.简历一个与服务端的Socket对象：套接字
            Socket socket = new Socket("127.0.0.1", 9999);
            // 2.从socket管道中获取一个输出流，写数据给服务端
            OutputStream os = socket.getOutputStream() ;
            // 3.把输出流包装成一个打印流
            PrintWriter pw = new PrintWriter(os);
            // 4.反复接收用户的输入
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String line = null ;
            while((line = br.readLine()) != null){
               pw.println(line);
               pw.flush();
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

线程池处理类

```
// 线程池处理类
public class HandlerSocketThreadPool {

   // 线程池
   private ExecutorService executor;

   public HandlerSocketThreadPool(int maxPoolSize, int queueSize){

      this.executor = new ThreadPoolExecutor(
            3, // 8
            maxPoolSize,
            120L,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(queueSize) );
   }

   public void execute(Runnable task){
      this.executor.execute(task);
   }
}
```

服务端源码分析

```
public class Server {
   public static void main(String[] args) {
      try {
         System.out.println("----------服务端启动成功------------");
         ServerSocket ss = new ServerSocket(9999);

         // 一个服务端只需要对应一个线程池
         HandlerSocketThreadPool handlerSocketThreadPool =
```

```
                new HandlerSocketThreadPool(3, 1000);

            // 客户端可能有很多个
            while(true){
                Socket socket = ss.accept() ; // 阻塞式的!
                System.out.println("有人上线了！！");
                // 每次收到一个客户端的socket请求，都需要为这个客户端分配一个
                // 独立的线程 专门负责对这个客户端的通信！！
                handlerSocketThreadPool.execute(new ReaderClientRunnable(socket));
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
class ReaderClientRunnable implements Runnable{

    private Socket socket ;

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

    @Override
    public void run() {
        try {
            // 读取一行数据
            InputStream is = socket.getInputStream() ;
            // 转成一个缓冲字符流
            Reader fr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(fr);
            // 一行一行的读取数据
            String line = null ;
            while((line = br.readLine())!=null){ // 阻塞式的！！
                System.out.println("服务端收到了数据: "+line);
            }
        } catch (Exception e) {
            System.out.println("有人下线了");
        }

    }
}
```

小结

- 伪异步io采用了线程池实现，因此避免了为每个请求创建一个独立线程造成线程资源耗尽的问题，但由于底层依然是采用的同步阻塞模型，因此无法从根本上解决问题。
- 如果单个消息处理的缓慢，或者服务器线程池中的全部线程都被阻塞，那么后续socket的i/o消息都将在队列中排队。新的Socket请求将被拒绝，客户端会发生大量连接超时。

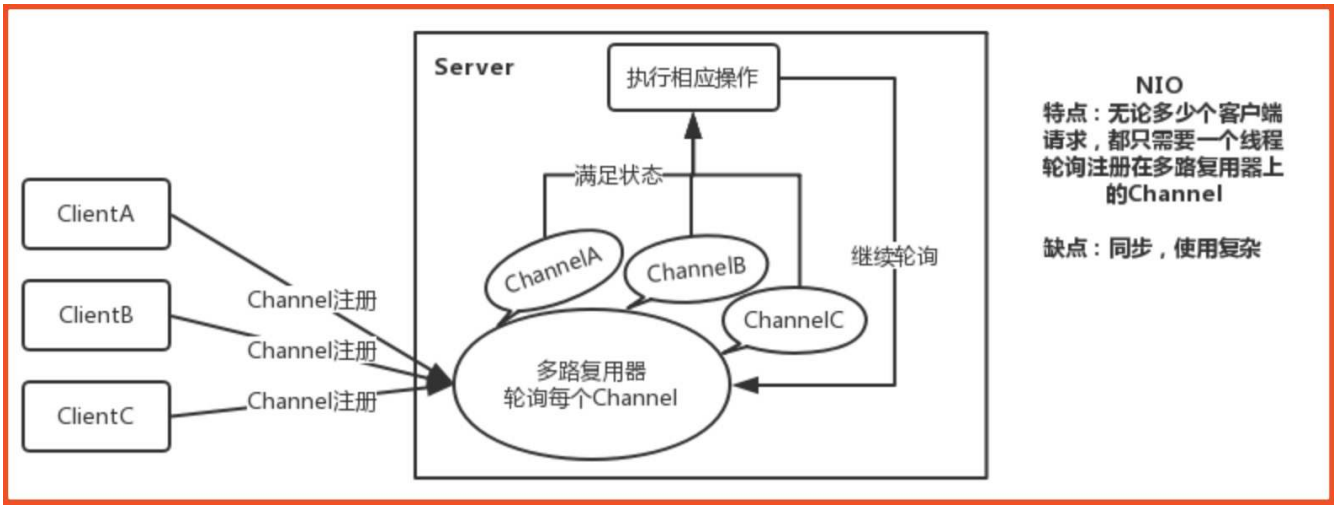
2.5 基于BIO模式下即时通信

基于BIO模式下的即时通信，我们需要解决客户端到客户端的通信，也就是需要实现客户端与客户端的端口消息转发逻辑。

2.6 NIO的使用

NIO（Non-blocking IO） 同步非阻塞式IO 通信， NIO提供了与传统BIO模型中的Socket和ServerSocket 相对应的SocketChannel和ServerSocketChannel两种不同的套接字通道实现。

```
（同步阻塞式通信）          （非阻塞式通信）
Socket                    SocketChannel
ServerSocket              ServerSocketChannel
```



Java NIO: Channels and Buffers（通道和缓冲区）

标准的IO基于字节流和字符流进行操作的，而NIO是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

Java NIO: Selectors（多路复用器）

Java NIO引入了多路复用器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

```
新增的这两种通道都支持阻塞和非阻塞两种模式。
 阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；
 非阻塞模式正好与之相反。 对于低负载、低并发的应用程序，
 可以使用同步阻塞I/O(BIO)来提升开发速率和更好的维护性；
 对于高负载、高并发的（网络）应用，应使用NIO的非阻塞模式来开发。

 --- 缓冲区 Buffer
    Buffer是一个对象，包含一些要写入或者读出的数据。
    在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，也是写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。
    缓冲区实际上是一个数组，并提供了对数据结构化访问以及维护读写位置等信息。
    具体的缓存区有这些：ByteBuffe、CharBuffer、 ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。他们实现了相同的接口：Buffer。

 ---  通道 Channel
 ---  多路复用器 Selector
```

```
NIO: 如何实现客户端与客户端的通信。(同步非阻塞)

同步 ： 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；
异步 ： 委托一小弟拿银行卡到银行取钱，然后给你
          （使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS
      （银行卡和密码），OS需要支持异步IO操作API）；

阻塞 ： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；
非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，
          没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，
如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进
行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多
路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，
                   客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，
```

2.7 AIO编程

- BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中， 编程比较复杂，JDK1.4开始支持。
- AIO方式使用于连接数目多且连接比较长（重操作）的架构， 比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

```
AIO
    异步非阻塞，基于NIO的，可以称之为NIO2.0
    BIO                    NIO                              AIO
Socket                SocketChannel                    AsynchronousSocketChannel
ServerSocket          ServerSocketChannel          AsynchronousServerSocketChannel
```

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,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
```

第三章 Netty框架的选择

3.1 为何选择Netty

1、虽然JAVA NIO框架提供了 多路复用IO的支持，但是并没有提供上层"信息格式"的良好封装。例如前两者并没有提供针对 Protocol Buffer、JSON这些信息格式的封装，但是Netty框架提供了这些数据格式封装（基于责任链模式的编码和解码功能）；

2、直接使用NIO需要需要额外的技能，例如Java多线程，网络编程；

3、要编写一个可靠的、易维护的、高性能的NIO服务器应用并不是一个简单的事情，如果没有足够的NIO编程经验，那么一个NIO框架的稳定性往往需要半年或者更长的时间，如果在生产环境中可能引起整个服务集群的关联性奔溃，这种非正常停机会带来巨大损失。除此之外，框架本身要兼容实现各类操作系统的实现外。更重要的是它应该还要处理很多上层特有服务，例如：客户端的权限、还有上面提到的信息格式封装、简单的数据读取，断连重连，半包读写，心跳等等，这些Netty框架都提供了响应的支持。

4、JAVA NIO框架存在一个poll/epoll bug：Selector doesn't block on Selector.select(timeout)，不能block意味着CPU的使用率会变成100%（这是底层JNI的问题，上层要处理这个异常实际上也好办）。当然这个bug只有在Linux内核上才能重现。这个问题在JDK 1.7版本中还没有被完全解决，但是Netty已经将这个bug进行了处理。

这个Bug与操作系统机制有关系的，JDK虽然仅仅是一个兼容各个操作系统平台的软件，但在JDK5和JDK6最初的版本中（严格意义上来将，JDK部分版本都是），这个问题并没有解决，而将这个帽子抛给了操作系统方，这也就是这个bug最终一直到2013年才最终修复的原因(JDK7和JDK8之间)。

Netty的优点如下：

- API 使用简单，开发门槛低；
- 功能强大，预置了多种编解码功能，支持多种主流协议；
- 定制能力强，可以通过 ChannelHandler 对通信框架进行灵活地扩展；
- 性能高，通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优；
- 成熟、稳定，Netty 修复了已经发现的所有 JDK NIO BUG，业务开发人员不需要再为 NIO 的 BUG 而烦恼；
- 社区活跃，版本迭代周期短，发现的 BUG 可以被及时修复，同时，更多的新功能会加入；
- 经历了大规模的商业应用考验，质量得到验证。Netty 在互联网、大数据、网络游戏、企业应用、电信软件等众多行业已经得到了成功商用，证明它已经完全能够满足不同行业的商业应用了。

正是因为这些优点，Netty 逐渐成为了 Java NIO 编程的首选框架。

3.2 Netty入门应用

3.3 Netty粘包与拆包

敬请期待！