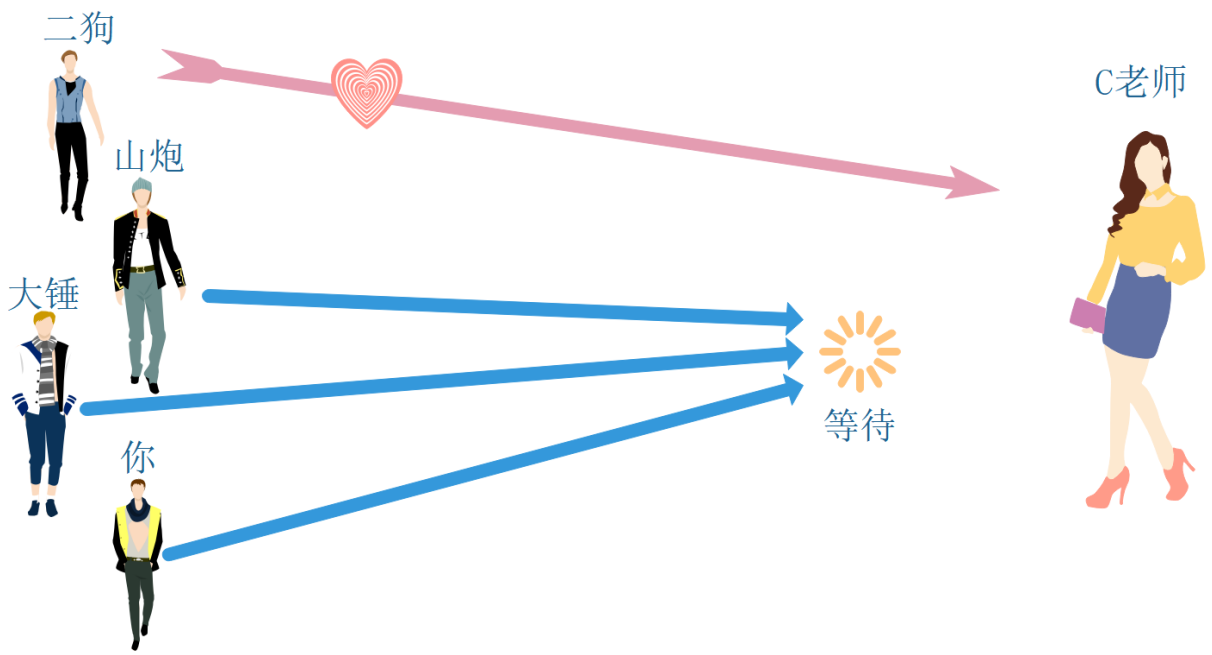
javaIO模型

BIO模型（Block -IO）

BIO模型又叫同步阻塞IO。就是我们学习的时候第一个学习的，印象最深刻的那一款！

直接上图：

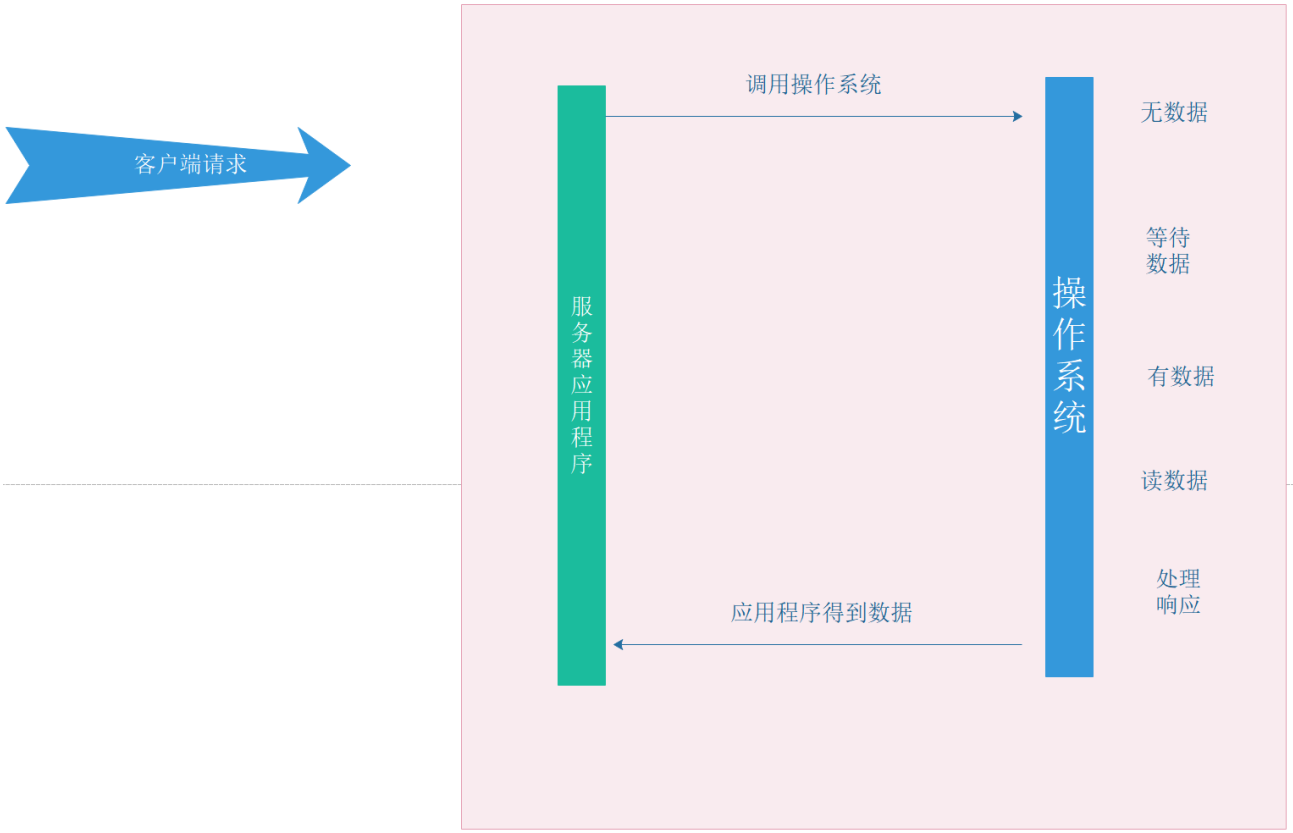


图中二狗在向老师提问的时候，其他的三位同学就职能等待！二狗的问题解答完成之后，才能继续解答其他同学，如果二狗故意降低自己的理解能力，增长了问问题的时间，其他同学的心里肯定是非常糟糕的。

同步：同时只能给二狗一位同学解答问题。

阻塞：二狗故意放慢说话速度，当二狗的一句话没有说完的时候，C老师只能等待，等二狗说完她才能继续解答。

正经图：



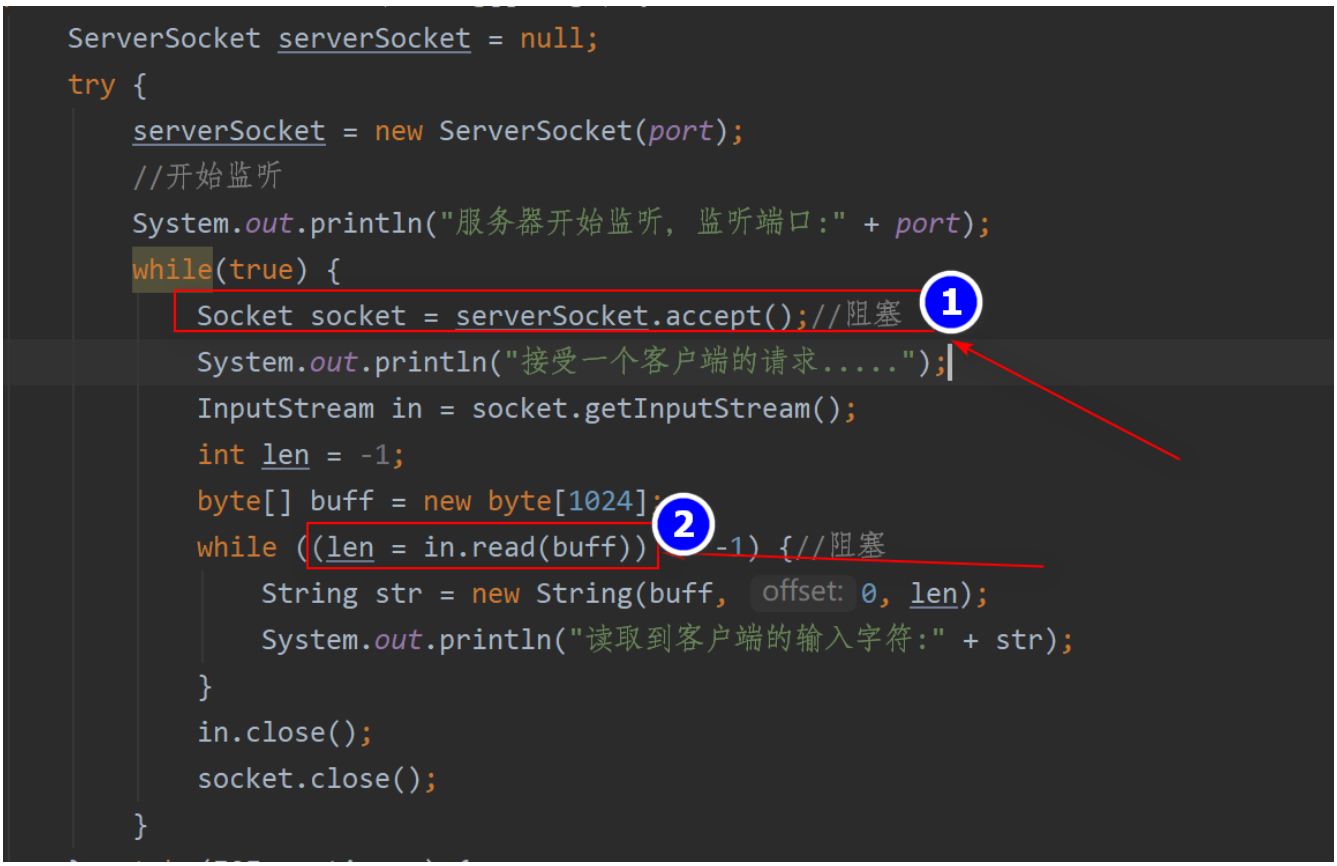
上程序：

服务端：

```
/**
 * BIO模型服务器端
 */
public class Server {
    private static int port = 8585;
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(port);
            //开始监听
            System.out.println("服务器开始监听，监听端口:" + port);
            while(true) {
                Socket socket = serverSocket.accept();//阻塞
                System.out.println("接受一个客户端的请求.....");
                InputStream in = socket.getInputStream();
                int len = -1;
                byte[] buff = new byte[1024];
                while ((len = in.read(buff)) != -1) {//阻塞
                    String str = new String(buff, 0, len);
                    System.out.println("读取到客户端的输入字符:" + str);
                }
                in.close();
                socket.close();
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

在这个案例中的服务器端的程序中会有两处阻塞，



阻塞1是等待客户端连接，阻塞2是等待客户端发送数据。

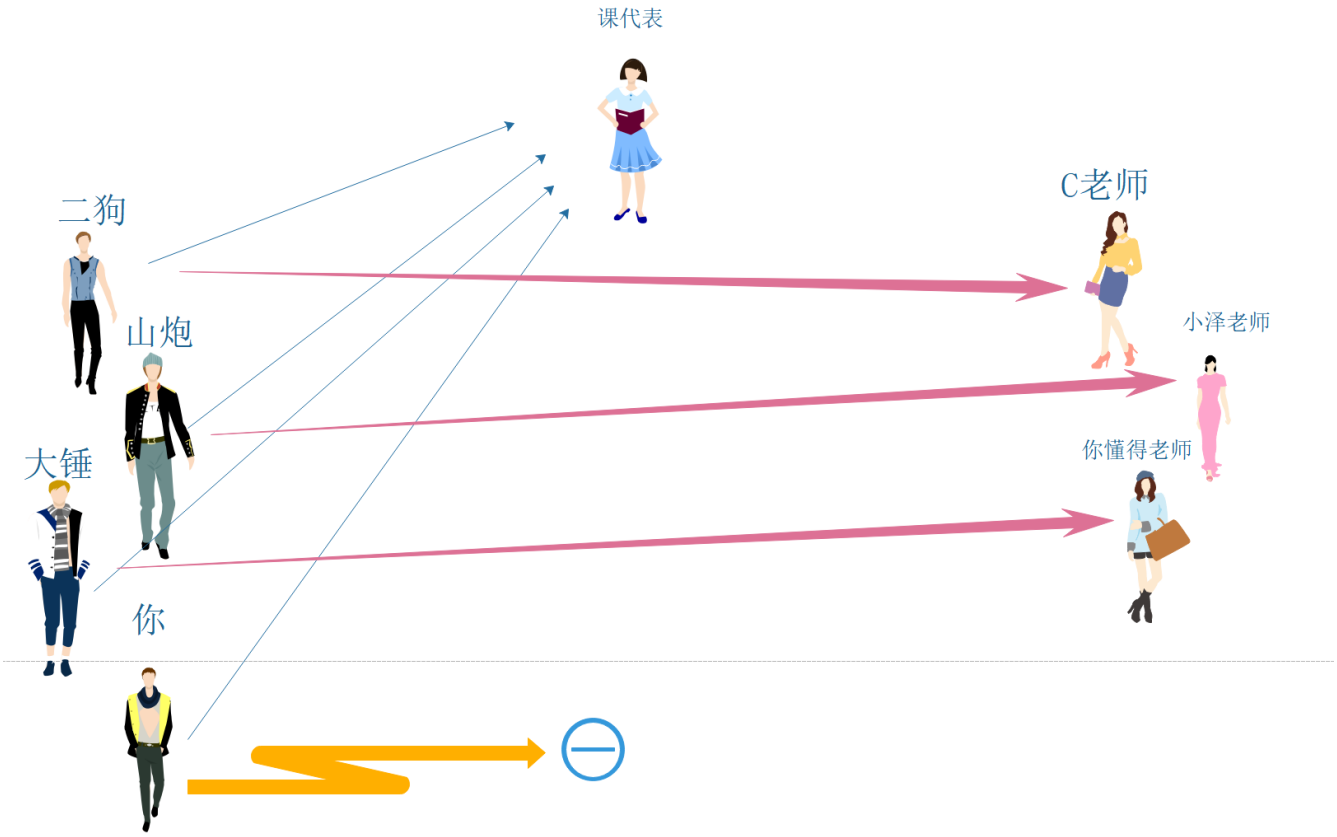
客户端程序：

```
/**
 * BIO模型客户端
 */
public class Client {
    private static int port = 8585;
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.println("请输入客户端编号:");
        int no = sc.nextInt();

```

```
        Socket socket = null;
        try {
            System.out.println("客户端"+no+"开始连接服务器..");
           socket = new Socket("127.0.0.1",port);
           if(socket!=null){
               System.out.println("客户端:"+no+"连接服务器成功!");
           }
            OutputStream out = socket.getOutputStream();

           while(true){
               System.out.println("客户端"+no+"请输入要发送字符(输入quit表示结束):");
               String str = sc.next();
               if(str.trim().equalsIgnoreCase("quit"))
                   break;
               out.write((no+":"+str).getBytes());
           }
            System.out.println("客户端"+no+"连接中断");
           out.close();
           socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

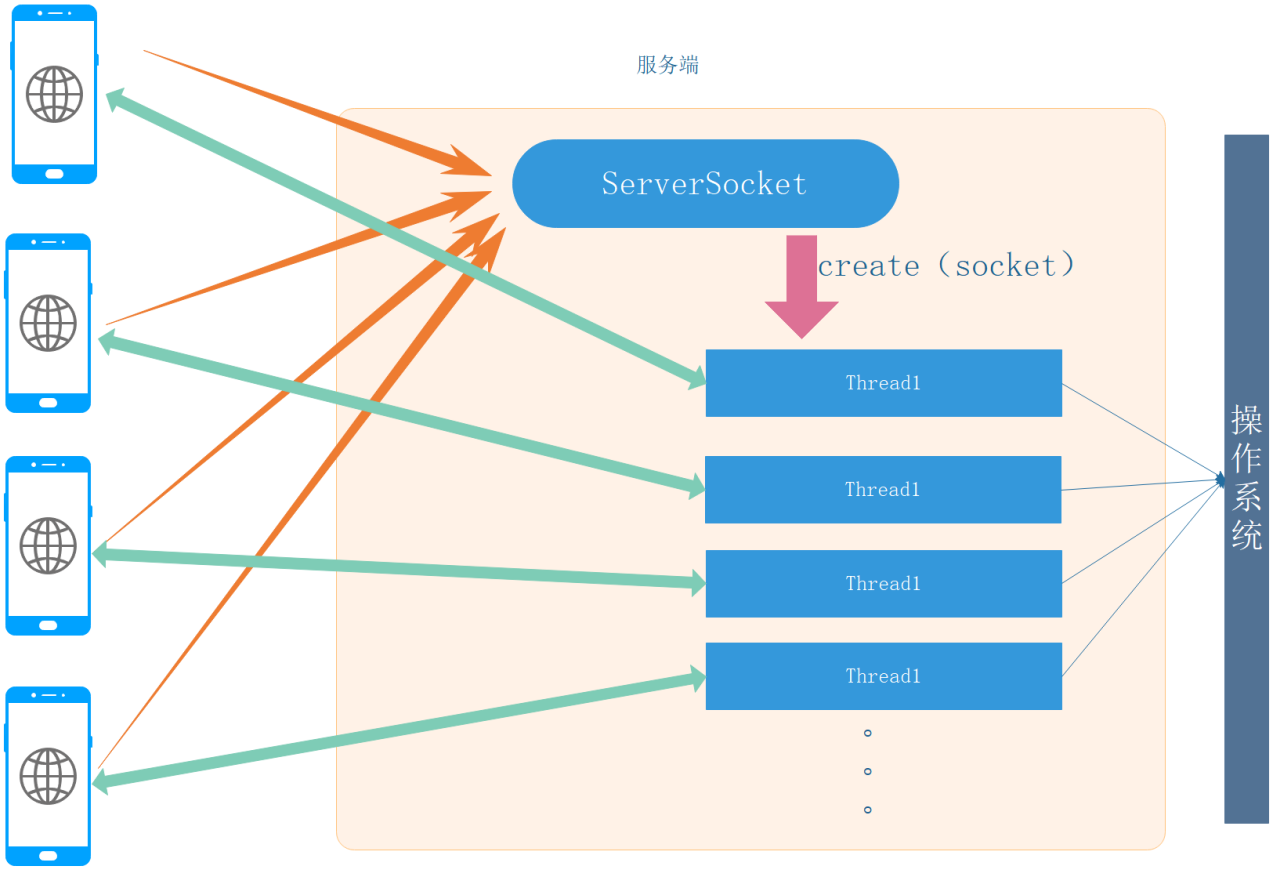
调整上面的程序结构

上面的程序我们觉得很不合理（虽然现实中有时就是这样的）。所以我们来调整这段程序。调整方式是:准备多个老师，每个老师负责一位同学的问题。安排课代表为大家安排老师。



在这幅图中，我们有多个老师，当然就可以同时为多个同学解答问题，但是老师的数量总归是有限的。学生的需求太多，这时就会出现新的问题。我们需要解决。

正经图：



我们的ServerSocket只负责接收请求，接收到请求之后，就创建一个新的线程，并且将socket交给新线程，由新线程处理客户端的连接请求。serversocket就可以继续监听了。

但是随着线程数量的上升，系统的压力自然就越大。所以还是有问题（就好像老师不够）。

上程序：

服务端：

```
/**
 * BIO模型服务器端
 */
public class Server1 {
    private static int port = 8585;
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(port);
            //开始监听
            System.out.println("服务器开始监听，监听端口:" + port);
            while(true) {
                final Socket socket = serverSocket.accept();
                System.out.println("接受一个客户端的请求.....");
                new Thread(){//开启新的线程处理socket
                    public void run(){
                        try{
                        InputStream in = socket.getInputStream();
                        int len = -1;
```

```
                        byte[] buff = new byte[1024];
                        while ((len = in.read(buff)) != -1) {
                            String str = new String(buff, 0, len);
                            System.out.println("读取到客户端的输入字符:" + str);
                        }
                        in.close();
                        socket.close();
                        }catch (Exception e){
                            e.printStackTrace();
                        }
                    }
                }.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

客户端：

```
好像和前面的客户端是一样的(￣▵￣; )
```

通过上面的分析和调整，我们发现了新的问题：虽然我们通过开启子线程，但是线程数量的增加让我们的系统压力增加，所以出现NIO（单线程处理并发问题）。

NIO模型（New-IO/NO-Block-IO）

多路复用：

多路复用 编辑

数据通信系统或计算机网络系统中，传输媒体的带宽或容量往往会大于传输单一信号的需求，为了有效地利用通信线路,希望一个信道同时传输多路信号，这就是所谓的多路复用技术(Multiplexing)。采用多路复用技术能把多个信号组合起来在一条物理信道上进行传输，在远距离传输时可大大节省电缆的安装和维护费用。频分多路复用FDM (Frequency Division Multiplexing)和时分多路复用TDM (Time Division Multiplexing)是两种最常用的多路复用技术。

体现在我们的IO模型中，就一个线程处理多个读写请求。

Socket底层是如何实现IO操作的?

通过检查源码，发现最终accept方法最终调用了一个本地方法：

```
static native int accept0(int fd, InetSocketAddress[] isaa) throws IOException;
```

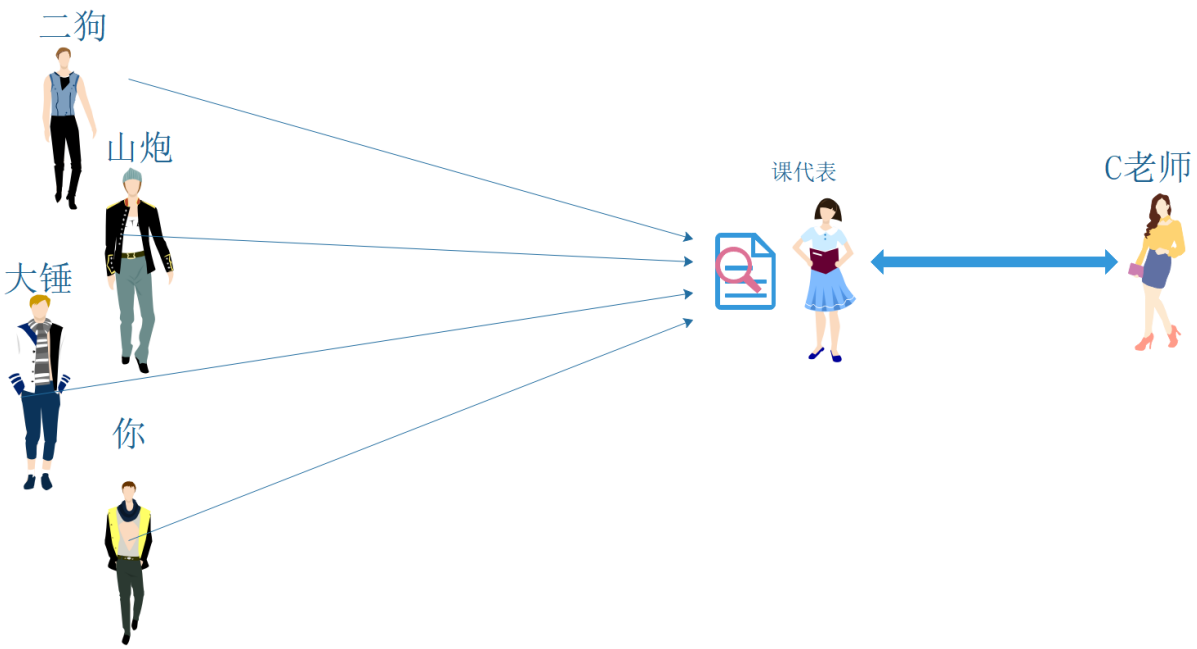
最终我们要知道，网络监听，接收请求，从IO进行数据读写都是由操作系统内核完成的。

[1]NIO要解决的问题

①让accept()和read()不在阻塞

②实现单线程解决并发问题（多路复用）。

先上图：



其他老师生病了，只有一个C老师在，课代表用一个表格记录所有同学的问题，任何以为同学提问，都是找课代表，课代表告诉老师，老师就来处理这位同学的问题，解答过程中，如果这位同学正在思考，这是C老师不会等待这位同学，而是继续解答其他同学的问题，课代表会不断的询问所有同学是不是思考好了，凡是思考好的，课代表才会让他继续和C老师沟通。

这个过程中

C老师不等待学生思考：不阻塞。

C老师一个人在宏观上可以同时解决多个同学的问题：多路复用。

注意：在某一个时刻，C老师同时只能解决一个同学的问题。

开始上程序：

如何解决不阻塞问题?

为什么会阻塞?

因为线程在等待资源：等待客户端连接，等待客户端输入数据。

解决办法就是不让它等待，如果没有连接就不等他。如果没有数据就不等它。

于是我们需要修改ServereSocket和socket这两个类中的API。让他们对应的方法不要再等待。

如果真的修改了这两个类，sum的这个项目经理就可以下岗了。

所以sum没有修改，而是提供了另外一套接口，实现不阻塞的socket。

> java.nio
> java.nio.channels
> java.nio.channels.spi
> java.nio.charset
> java.nio.charset.spi
> java.nio.file
> java.nio.file.attribute
> java.nio.file.spi

类介绍：

java.nio.channels

Class ServerSocketChannel

accept

```
public abstract SocketChannel accept()
                                throws IOException
```

如果该信道是在非阻塞模式，则此方法将立即返回null，如果没有未决的连接。 否则，它将无限期地阻塞，直到有新的连接或发生I / O错误。

configureBlocking

```
public final SelectableChannel configureBlocking(boolean block)
                                          throws IOException
```

如果给定阻塞模式与当前阻塞模式不同，则该方法调用implConfigureBlocking方法，同时保持适当的锁，以便更改模式。如果参数block为true那么这个通道将被置于阻塞模式; 如果false那么它将被放置为非阻塞模式。

java.nio.channels

Class SocketChannel

configureBlocking

```
public final SelectableChannel configureBlocking(boolean block)
                                          throws IOException
```

如果给定阻塞模式与当前阻塞模式不同，则该方法调用implConfigureBlocking方法，同时保持适当的锁，以便更改模式。如果参数block为true那么这个通道将被置于阻塞模式; 如果false那么它将被放置为非阻塞模式。

read

```
public abstract int read(ByteBuffer dst)
                  throws IOException
```

从该通道读取到给定缓冲区的字节序列。

读取操作可能不会填充缓冲区，实际上它可能不会读取任何字节。 是否这样做取决于渠道的性质和状态。 例如，非阻塞模式的套接字通道不能再读取比从套接字的输入缓冲区可以立即获得的任何字节数; 类似地，文件通道不能读取比保留在文件中的任何更多的字节。 但是，如果通道处于阻塞模式，并且缓冲区中至少剩余一个字节，则该方法将被阻塞，直到读取至少一个字节为止。

结果：读取的字节数，可能为零，如果通道已达到流出端， 则为-1

上测试程序：

```
/**
 * NIO
 */
public class Server0 {
    public static void main(String[] args) {
        try {
            // 创建ServerSocketChannel通道，绑定监听端口为8585
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.socket().bind(new InetSocketAddress(8585));
            // 设置为非阻塞模式
            serverSocketChannel.configureBlocking(false);
            //开始监听
            System.out.println("开始监听....");
            SocketChannel accept = serverSocketChannel.accept();
            System.out.println("accept:"+accept);
            if (accept != null) {
                System.out.println("接受到请求...");
                //设置为非阻塞
                accept.configureBlocking(false);
                //读取数据？？？
            }
            System.out.println("OVER");
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

```
开始监听....
accept:null
OVER
```

一旦启动，就直接执行结束。不会阻塞。

问题：单线程，非阻塞，如何保证监听到请求，并且读取到数据?

循环监听，一旦接受到请求，就处理数据。

```
while(true) {
        // 设置为非阻塞模式
        serverSocketChannel.configureBlocking(false);
        SocketChannel accept = serverSocketChannel.accept();//非阻塞
        if (accept != null) {
           System.out.println("接受到请求...");
           //设置为非阻塞
           accept.configureBlocking(false);
           ByteBuffer buffer = ByteBuffer.allocate(1024);
           int read = sc.read(buffer);//非阻塞状态
           if (read > 0) {
                buffer.flip();
                 String str = new String(buffer.array(), 0, buffer.limit());
                 System.out.println("服务器端接受到数据：" + str);
            }
         }
   }
```

上面的程序中我们虽然可以做到循环的监听，但是无法保证一定能读取到客户端的数据。

因为read方法也是非阻塞的，一旦客户端没有数据，就会直接跳过，进行下一次监听，那么本次的所有资源都会丢失。

继续修改程序，调整思路：将每次获取的连接放到一个集合中，防止丢失。不断的循环遍历这个集合，从每一个连接中不断的尝试读取数据。

```
public static void main(String[] args) {
        //创建一个集合用来存储接收到的请求
        List<SocketChannel> socketChannels = new ArrayList<SocketChannel>();
        try {
            // 创建ServerSocketChannel通道，绑定监听端口为8585
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.socket().bind(new InetSocketAddress(8585));
            while(true) {
                // 设置为非阻塞模式
```

```
                serverSocketChannel.configureBlocking(false);
                SocketChannel socketChannel = serverSocketChannel.accept();
                if (socketChannel != null) {
                    System.out.println("接受到请求...");
                    //设置为非阻塞
                    socketChannel.configureBlocking(false);
                    socketChannels.add(socketChannel);
                    System.out.println("socketChannels-size:"+socketChannels.size());
                }
                //每次遍历所有的SocketChannl
                for (int x = 0;x < socketChannels.size();x++) {
                    SocketChannel sc = socketChannels.get(x);
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    try {
                        //判断是否可读?
                        int read = sc.read(buffer);
                        if (read > 0) {
                            buffer.flip();
                            String str =
                                new String(buffer.array(), 0, buffer.limit());
                            System.out.println("服务器端接受到数据: " + str);
                        }
                    }catch(IOException e){
                        System.out.println("关闭了一个连接");
                        socketChannels.remove(sc);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

我们已经解决了问题：单线程，不阻塞处理并发问题（多路复用）。

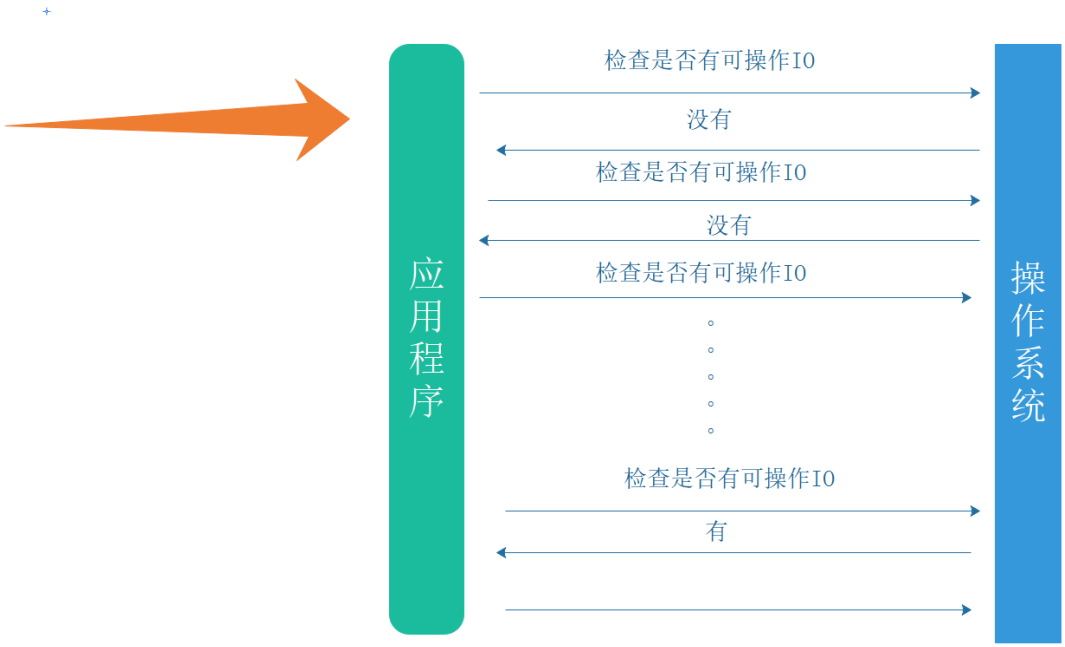
客户端程序：

```
    public static void main(String[] args) throws IOException, InterruptedException {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入客户端编号:");
        int no = sc.nextInt();
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.connect(new InetSocketAddress("127.0.0.1",8585));
        //
        ByteBuffer buff = ByteBuffer.allocate(1024);
        String str = "";
        while(true){
            if(!socketChannel.finishConnect()){
                Thread.sleep(100);
                continue;
            }
            System.out.println("客户端"+no+"请输入要发送的内容:");
            str = sc.next();
```

```
            if(str.equalsIgnoreCase("quit"))
                break;
            byte[] bytes = (no+":"+str).getBytes();
            buff = ByteBuffer.allocate(bytes.length);
            buff.put(bytes);
            buff.flip();
            socketChannel.write(buff);
        }
    }
```

注意：单线程在某一时刻同时只能处理一个事件。看上去是并发，其实是因为轮询导致的。

看图：



新的问题

如果某个时间段里面没有任何客户端连接请求，那么我们的while循环就一直在空循环，占用着CPU资源。

如果我们现在有10000个请求连接，但是这个10000个连接中只有100个连接是有交互的。其他的9900个连接暂时没有交互，那么我们每次都会把这个10000个连接逐个询问一次是不是在浪费资源和时间。

问题:java中的IO中读写阻塞初衷是什么?

一旦阻塞就会释放CPU资源。

解决新问题：①进行阻塞。 ②避免不必要的轮询。

让轮询器阻塞。（上面例子中就是课代表阻塞。当课代表阻塞的时候，C老师就处于休息状态。）

轮询器阻塞的意思就是：轮询器去判断是否有请求事件发生，是否有读数据的事件发生，是否有写数据的事件发生。如果都没有，那么轮询器就阻塞等待。轮询器一旦阻塞，整个线程就阻塞，就让出CPU的资源。

任何一个事件（请求，可读，可写）一旦打通阻塞，轮询器阻塞就被打通，这时主线程开始处理事件。

NIO提供的多路复用器:

```
public abstract class Selector
extends Object
implements Closeable

        SelectableChannel对象的多路复用器 。
```

提供方法:

---

select() public abstract int select() throws IOException

选择一组其相应通道准备好进行I / O操作的键。

---

selectedKeys() public abstract Set<SelectionKey> selectedKeys()

返回此选择器的选择键集。

我们可以将连接请求，读数据，写数据三种请求全部放入Selector中。让Selector进行轮询选择。

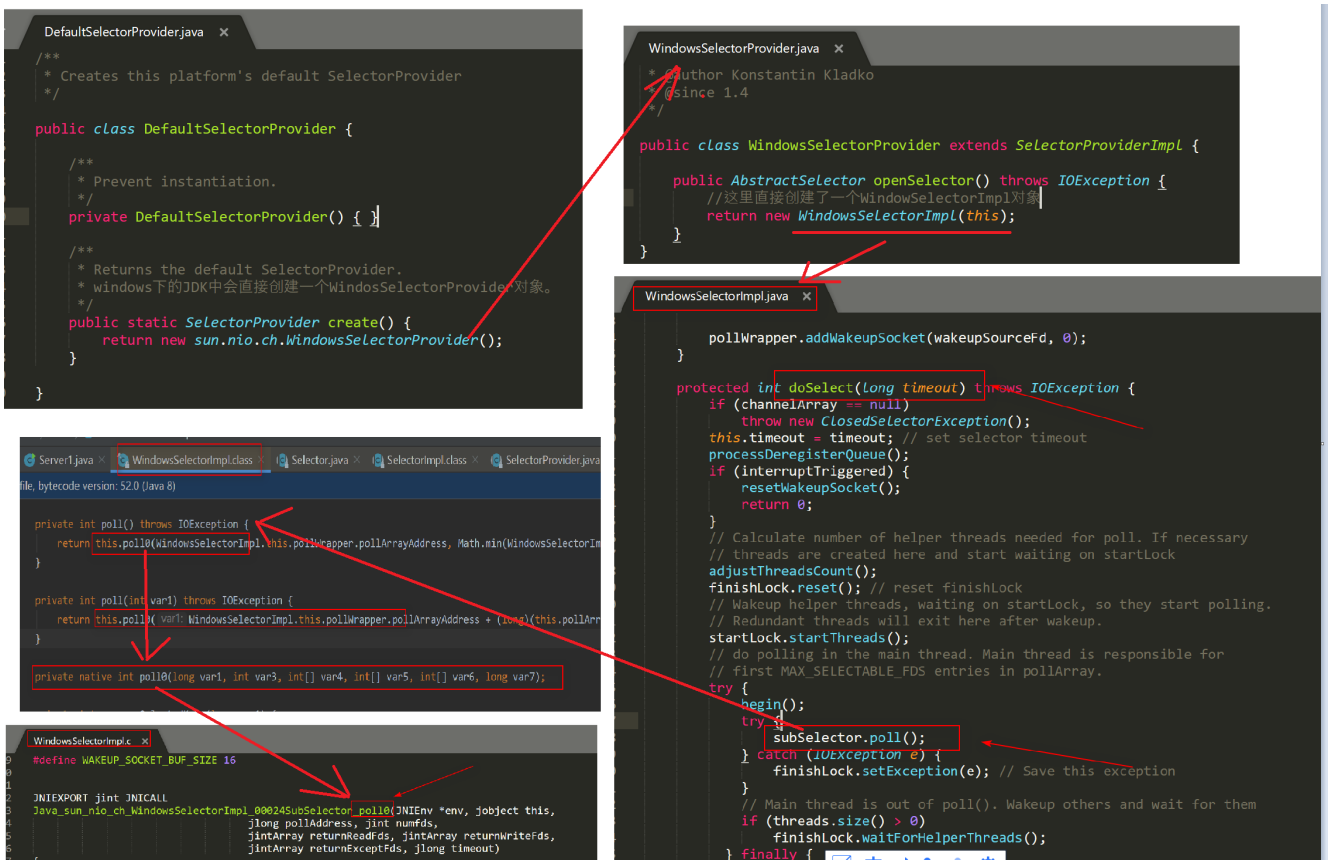
上代码:

```
    public static void main(String[] args) throws InterruptedException {
        try {
            // 创建ServerSocketChannel通道，绑定监听端口为8585
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.socket().bind(new InetSocketAddress(8585));
            serverSocketChannel.configureBlocking(false);
            //创建一个选择器对象
            Selector selector = Selector.open();
            // 注册选择器,设置选择器选择的操作类型
            serverSocketChannel.register(selector,SelectionKey.OP_ACCEPT);
            while(true) {
                Thread.sleep(1000);
                //选择事件,这里会阻塞
                selector.select();
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while(iterator.hasNext()){
                    System.out.println("处理事件");
                    System.out.println(selectionKeys.size()+"----");
                    SelectionKey next = iterator.next();
                    iterator.remove();
                    if(next.isReadable()){
                        SocketChannel channel = (SocketChannel)next.channel();
```

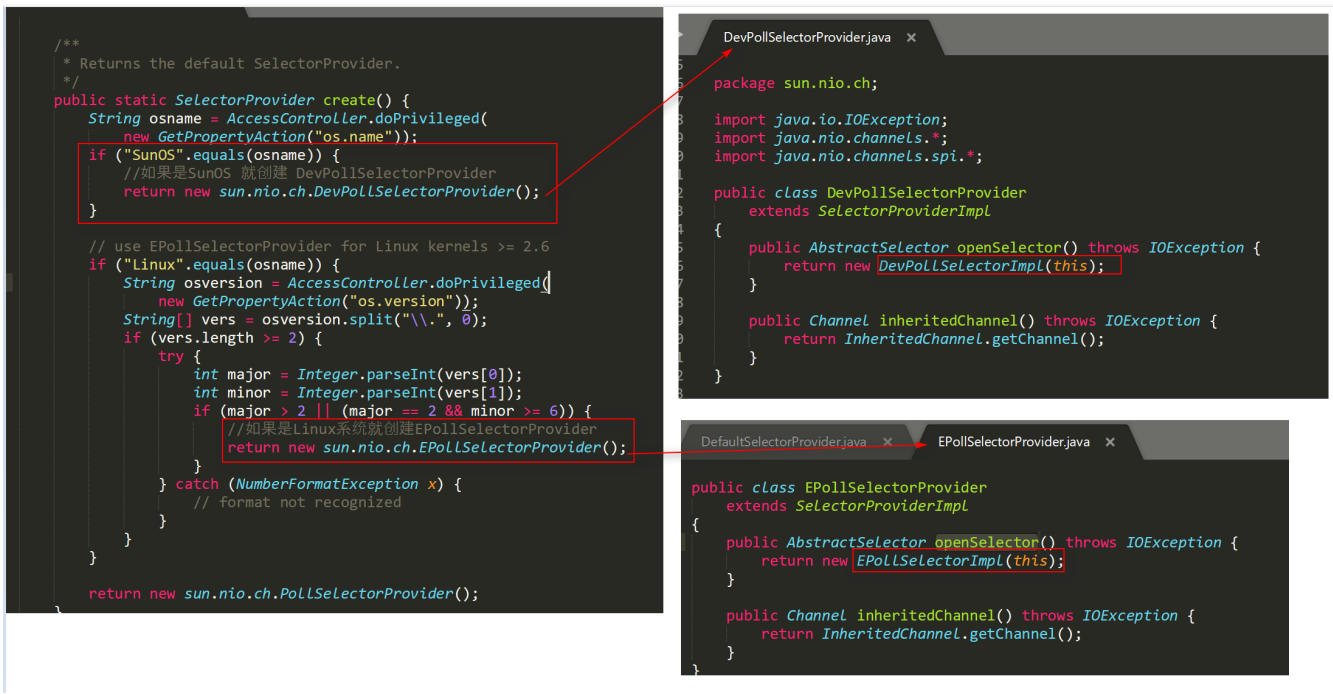
```
                        ByteBuffer dest = ByteBuffer.allocate(1024);
                        int read = channel.read(dest);
                        System.out.println(new String(dest.array(),0,read));
                    }else  if(next.isAcceptable()){
                        SocketChannel accept = serverSocketChannel.accept();
                        accept.configureBlocking(false);
                        //接收到请求，将连接注册到选择器中，并且设置监听类型为read
                        accept.register(selector,SelectionKey.OP_READ);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

查看Selector的底层实现：

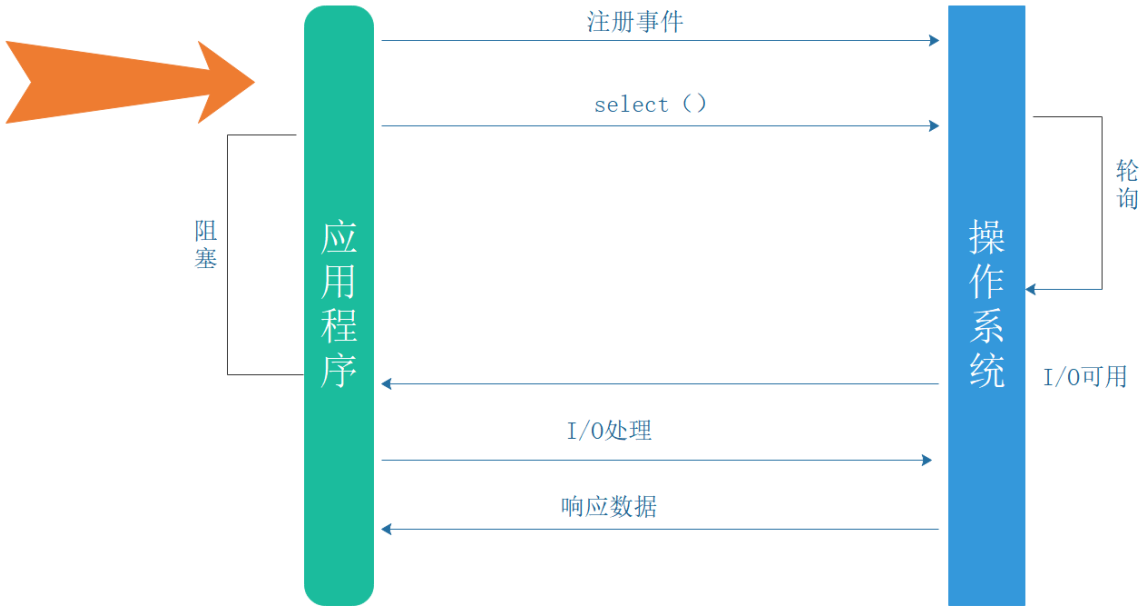
windows



其他系统：



从上面的程序中可以看出，其实NIO的select是调用了操作系统的API。也就是说对于事件的轮询是由操作系统内核完成的。而且socket本身也是通过os内核实现的。这样的话NIO的请求模型图就是下面的情况：



多路复用器的三种实现方式：

select模型：是由一个数组管理的，也就是每当注册一个感兴趣的事件时，就会占用一个数组的位置。由于是使用数组来存储事件的注册，所以就有长度限制，在32位机器上限制为1024，在64位机器上限制为2048。 每次系统请求时都会线性遍历整个数组看是否有可处理的事件，若没有则睡眠，直到超时或被事件触发唤醒后重新遍历，性能自然不怎么样。

poll和select很类似，最大的区别在于它不是用数组实现的，而是使用链表实现，因此它就没有select最大数量的限制了。

epoll基于事件回调机制，即回调时直接通知进程，而无须使用某种方式来查看状态。它通过mmap映射内存来实现，不用像select和poll一样需要内存拷贝。

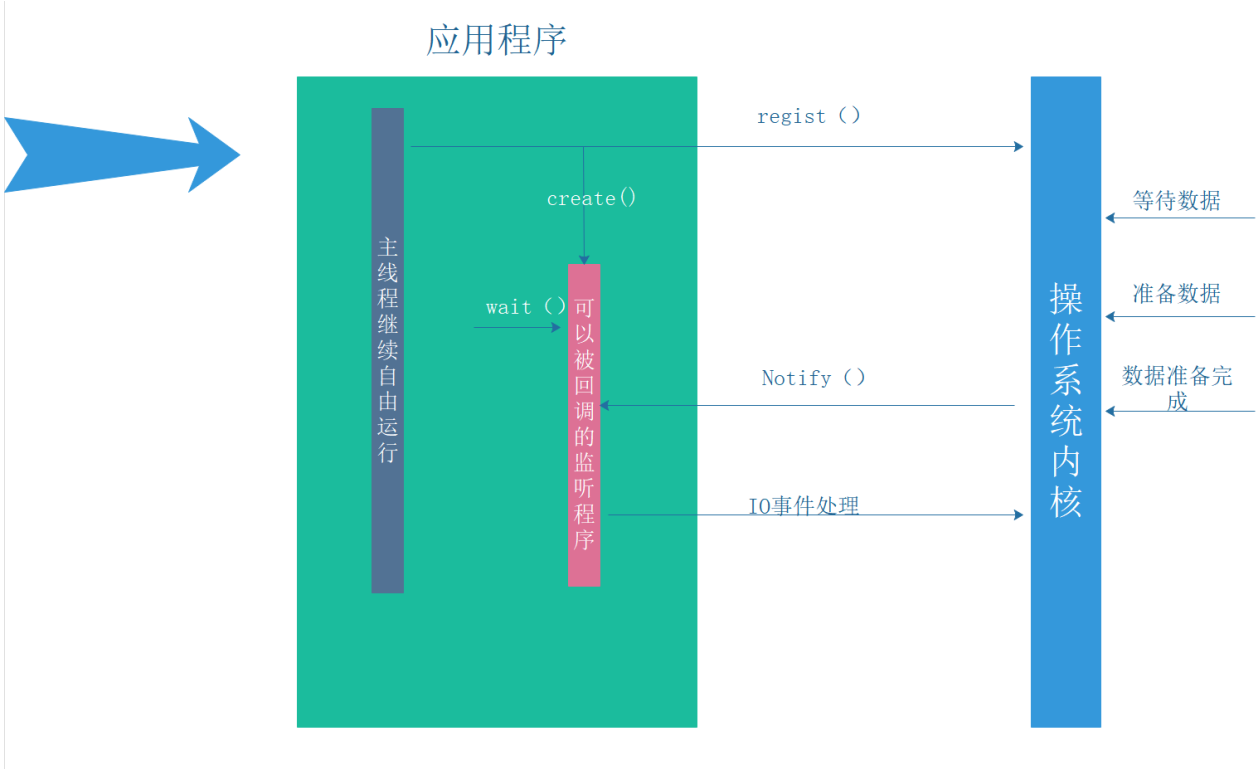
AIO模型（异步IO模型）

AIO是java中IO模型的一种，作为NIO的改进和增强随JDK1.7版本更新被集成在JDK的nio包中，因此AIO也被称作是NIO2.0。区别于传统的BIO(Blocking IO,同步阻塞式模型,JDK1.4之前就存在于JDK中，NIO于JDK1.4版本发布更新)的阻塞式读写，AIO提供了从建立连接到读、写的全异步操作。AIO可用于异步的文件读写和网络通信。

上图：

山炮提问之后，留一个回调的电话，然后就可以继续和二狗扯淡。 C老师做好结果，电话通知山炮过来拿结果。在这个过程中，山炮和C老师就是异步操作的。

正经图：



程序实现：

实现一个最简单的AIO socket通信server、client，主要需要这些相关的类和接口：

AsynchronousServerSocketChannel 服务端Socket通道类，负责服务端Socket的创建和监听；

AsynchronousSocketChannel 客户端Socket通道类，负责客户端消息读写；

CompletionHandler<A,V> 消息处理回调接口，是一个负责消费异步IO操作结果的消息处理器；

ByteBuffer 负责承载通信过程中需要读、写的消息。

AsynchronousServerSocketChannel的监听方法accept有两个重载的方法：

```
public abstract <A> void accept(A,CompletionHandler<AsynchronousSocketChannel,? super A>);
public abstract Future<AsynchronousSocketChannel> accept();
```

这两个重载方法的行为方式完全相同一种基于Future，一种基于回调。基于Future接口的就相当于同步阻塞。

先看基于Future接口的服务端：

```
/**
 * AIO-Future
 */
public class Server {
    public static void main(String[] args) throws IOException, ExecutionException, InterruptedException {
        //创建一个AsynchronousServerSocketChannel
        AsynchronousServerSocketChannel asynchronousServerSocketChannel = AsynchronousServerSocketChannel.open();
```

```
        //绑定地址
        asynchronousServerSocketChannel.bind(new InetSocketAddress("127.0.0.1",8585));
        while(true) {
            //Future方式开始监听
            System.out.println("开始监听8585");
            Future<AsynchronousSocketChannel> future =
asynchronousServerSocketChannel.accept();
            AsynchronousSocketChannel asynchronousSocketChannel = future.get();
            System.out.println("接收到请求...");
            ByteBuffer dst = ByteBuffer.allocate(1024);
            //循环读取数据
            while (asynchronousSocketChannel.isOpen()){
                Future<Integer> read = asynchronousSocketChannel.read(dst);
                Integer integer = read.get();
                if(integer>0){
                    System.out.println(new String(dst.array(),0,integer,"utf-8"));
                    dst.clear();
                }
            }
        }
    }
}

```

客户端：

```
public class Client {
    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入客户端编号:");
        int no = sc.nextInt();
        AsynchronousSocketChannel asynchronousSocketChannel =
AsynchronousSocketChannel.open();
        asynchronousSocketChannel.connect(new InetSocketAddress("127.0.0.1",8585));
        ByteBuffer buff = ByteBuffer.allocate(1024);
        String str = "";
        while(true){
            System.out.println("客户端"+no+"请输入要发送的内容:");
            str = sc.next();
            if(str.equalsIgnoreCase("quit"))
                break;
            byte[] bytes = (no+":"+str).getBytes();
            buff = ByteBuffer.allocate(bytes.length);
            buff.put(bytes);
            buff.flip();
            asynchronousSocketChannel.write(buff);
        }
    }
}
```

基于回调的方式：

服务器端：

```java
/**
 * AIO
 */
public class Server1 {
    public static void main(String[] args) throws Exception {
        //创建一个AsynchronousServerSocketChannel
       final  AsynchronousServerSocketChannel asynchronousServerSocketChannel = AsynchronousServerSocketChannel.open();
        //绑定地址
        asynchronousServerSocketChannel.bind(new InetSocketAddress("127.0.0.1",8585));
            //Future方式开始监听
            System.out.println("开始监听8585");
            asynchronousServerSocketChannel.accept(null,
                    new CompletionHandler<AsynchronousSocketChannel, Void>() {

                        @Override
                        public void completed(AsynchronousSocketChannel asynchronousSocketChannel, Void attachment) {
                            asynchronousServerSocketChannel.accept(null,this);
                            // 接收到新的客户端连接时调用，result就是和客户端的连接对话，此时可以通过result和客户端进行通信
                            System.out.println("accept completed");
                            ByteBuffer dst = ByteBuffer.allocate(1024);
                            //循环读取数据
                            while (asynchronousSocketChannel.isOpen()){
                                Future<Integer> read = asynchronousSocketChannel.read(dst);
                                try {
                                    Integer integer = read.get();
                                    if (integer > 0) {
                                        System.out.println(new String(dst.array(), 0, integer, "utf-8"));
                                        dst.clear();
                                    }
                                }catch (Exception e)
                                {}
                            }
                        }

                        @Override
                        public void failed(Throwable exc, Void attachment) {
                            // accept失败时回调
                            System.out.println("accept failed");
                        }
                    });
            //让程序暂停
            System.in.read();
    }
}
```