课程目标

1、掌握 NIO 的核心组件 Buffer、Selector、Channel。

2、何谓多路复用?

3、Netty 支持的功能与特性。

内容定位

1、有网络 IO 操作业务场景。

2、有 Netty 使用经验的人群。

1 Java NIO 三件套

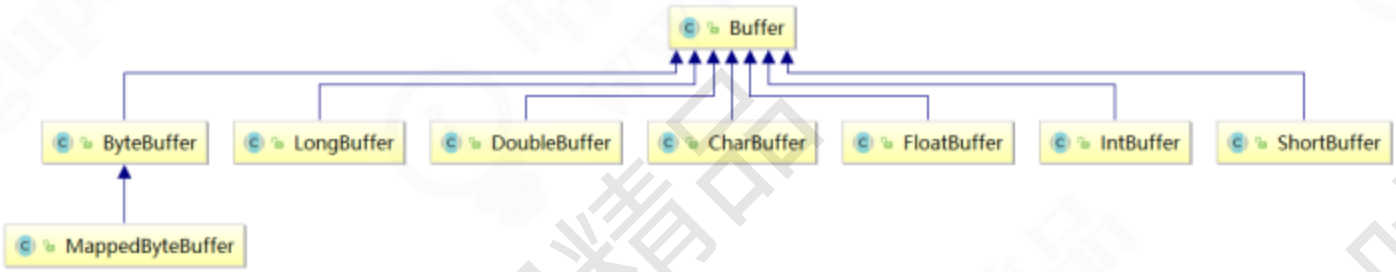
在 NIO 中有几个核心对象需要掌握：缓冲区（Buffer）、选择器（Selector）、通道（Channel）。

1.1 缓冲区 Buffer

1.Buffer 操作基本 API

缓冲区实际上是一个容器对象，更直接的说，其实就是一个数组，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，它也是写入到缓冲区中的；任何时候访问 NIO 中的数据，都是将它放到缓冲区中。而在面向流 I/O 系统中，所有数据都是直接写入或者直接将数据读取到 Stream 对象中。

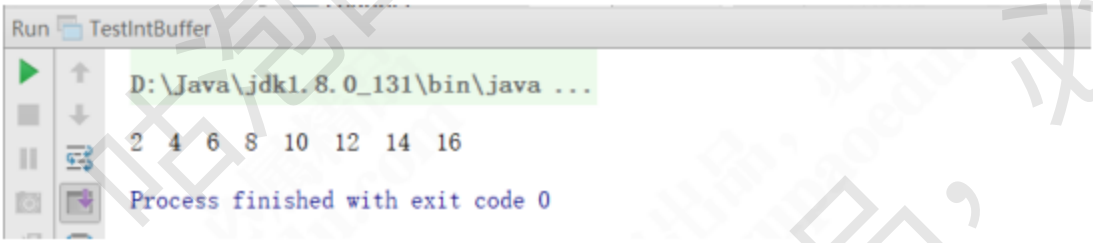
在 NIO 中，所有的缓冲区类型都继承于抽象类 Buffer，最常用的就是 ByteBuffer，对于 Java 中的基本类型，基本都有一个具体 Buffer 类型与之相对应，它们之间的继承关系如下图所示：



下面是一个简单的使用 IntBuffer 的例子：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.nio.IntBuffer;

public class IntBufferDemo {
    public static void main(String[] args) {
        // 分配新的 int 缓冲区，参数为缓冲区容量
        // 新缓冲区的当前位置将为零，其界限(限制位置)将为其容量。它将具有一个底层实现数组，其数组偏移量将为零。
        IntBuffer buffer = IntBuffer.allocate(8);

        for (int i = 0; i < buffer.capacity(); ++i) {
            int j = 2 * (i + 1);
            // 将给定整数写入此缓冲区的当前位置，当前位置递增
            buffer.put(j);
        }
        // 重设此缓冲区，将限制设置为当前位置，然后将当前位置设置为 0
        buffer.flip();
        // 查看在当前位置和限制位置之间是否有元素
        while (buffer.hasRemaining()) {
            // 读取此缓冲区当前位置的整数，然后当前位置递增
            int j = buffer.get();
            System.out.print(j + "  ");
        }
    }
}
```

运行后可以看到：



2.Buffer 的基本的原理

在谈到缓冲区时，我们说缓冲区对象本质上是一个数组，但它其实是一个特殊的数组，缓冲区对象内置了一些机制，能够跟踪和记录缓冲区的状态变化情况，如果我们使用 get()方法从缓冲区获取数据或者使用 put()方法把数据写入缓冲区，都会引起缓冲区状态的变化。

在缓冲区中，最重要的属性有下面三个，它们一起合作完成对缓冲区内部状态的变化跟踪：

position：指定下一个将要被写入或者读取的元素索引，它的值由 get()/put()方法自动更新，在新创建一个 Buffer 对象时，position 被初始化为 0。

limit：指定还有多少数据需要取出(在从缓冲区写入通道时)，或者还有多少空间可以放入数据(在从通道读入缓冲区时)。

capacity：指定了可以存储在缓冲区中的最大数据容量，实际上，它指定了底层数组的大小，或者至少是指定了准许我们使用的底层数组的容量。

以上三个属性值之间有一些相对大小的关系：0 <= position <= limit <= capacity。如果我们创建一个新的容量大小为 10 的 ByteBuffer 对象，在初始化的时候，position 设置为 0，limit 和 capacity 被设置为 10，在以后使用 ByteBuffer 对象过程中，capacity 的值不会再发生变化，而其它两个个将会随着使用而变化。

下面我们用代码来演示一遍，准备一个 txt 文档，存放的 E 盘，输入以下内容：

```
Tom.
```

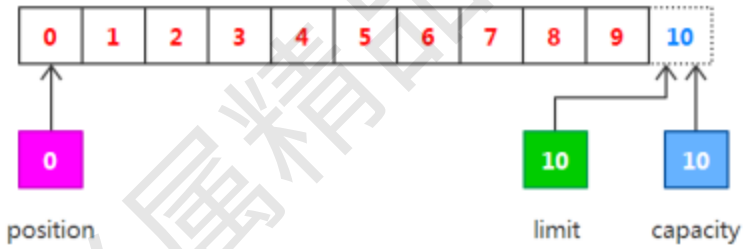
下面我们用一段代码来验证 position、limit 和 capacity 这几个值的变化过程，代码如下：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.io.FileInputStream;
import java.nio.*;
import java.nio.channels.*;

public class BufferDemo {
    public static void main(String args[]) throws Exception {
        //这用用的是文件 IO 处理
        FileInputStream fin = new FileInputStream("E://test.txt");
        //创建文件的操作管道
        FileChannel fc = fin.getChannel();

        //分配一个 10 个大小缓冲区，说白了就是分配一个 10 个大小的 byte 数组
        ByteBuffer buffer = ByteBuffer.allocate(10);
        output("初始化", buffer);

        //先读一下
        fc.read(buffer);
        output("调用 read()", buffer);
```

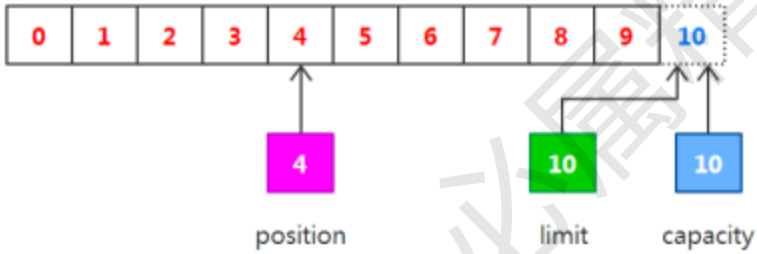
```
        //准备操作之前，先锁定操作范围
        buffer.flip();
        output("调用 flip()", buffer);

        //判断有没有可读数据
        while (buffer.remaining() > 0) {
            byte b = buffer.get();
            // System.out.print(((char)b));
        }
        output("调用 get()", buffer);

        //可以理解为解锁
        buffer.clear();
        output("调用 clear()", buffer);

        //最后把管道关闭
        fin.close();
    }

    //把这个缓冲里面实时状态给客应出来
    public static void output(String step, Buffer buffer) {
        System.out.println(step + " : ");
        //容量，数组大小
        System.out.print("capacity: " + buffer.capacity() + ", ");
        //当前操作数据所在的位置，也可以叫做游标
        System.out.print("position: " + buffer.position() + ", ");
        //锁定值，flip，数据操作范围索引只能在 position - limit 之间
        System.out.println("limit: " + buffer.limit());
        System.out.println();
    }
}
```

完成的输出结果为:

运行结果我们已经看到，下面呢对以上结果进行图解，四个属性值分别如图所示：



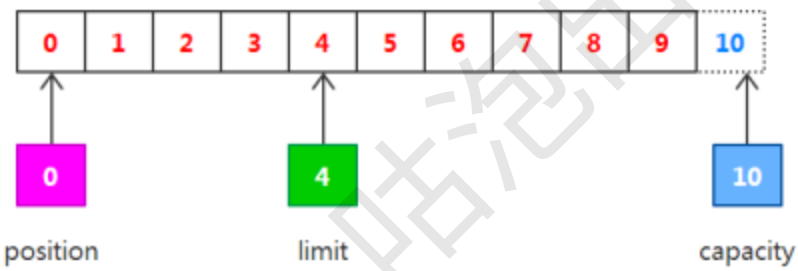
我们可以从通道中读取一些数据到缓冲区中，注意从通道读取数据，相当于往缓冲区中写入数据。如果读取 4 个自己的数据，则此时 position 的值为 4，即下一个将要被写入的字节索引为 4，而 limit 仍然是 10，如下图所示：



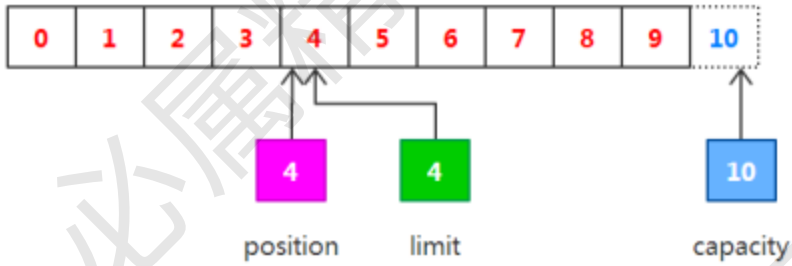
下一步把读取的数据写入到输出通道中，相当于从缓冲区中读取数据，在此之前，必须调用 flip()方法，该方法将会完成两件事情：

1. 把 limit 设置为当前的 position 值
2. 把 position 设置为 0

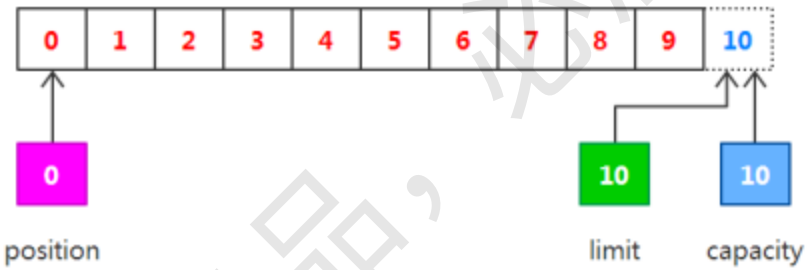
由于 position 被设置为 0，所以可以保证在下一步输出时读取到的是缓冲区中的第一个字节，而 limit 被设置为当前的 position，可以保证读取的数据正好是之前写入到缓冲区中的数据，如下图所示：



现在调用 get()方法从缓冲区中读取数据写入到输出通道，这会导致 position 的增加而 limit 保持不变，但 position 不会超过 limit 的值，所以在读取我们之前写入到缓冲区中的 4 个自己之后，position 和 limit 的值都为 4，如下图所示：



在从缓冲区中读取数据完毕后，limit 的值仍然保持在我们调用 flip()方法时的值，调用 clear()方法能够把所有的状态变化设置为初始化时的值，如下图所示：



3.缓冲区的分配

在前面的几个例子中，我们已经看过了，在创建一个缓冲区对象时，会调用静态方法 allocate()来指定缓冲区的容量，其实调用 allocate()相当于创建了一个指定大小的数组，并把它包装为缓冲区对象。或者我们也可以直接将一个现有的数组，包装为缓冲区对象，如下示例代码所示：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.nio.ByteBuffer;

/** 手动分配缓冲区 */
public class BufferWrap {

    public void myMethod() {
        // 分配指定大小的缓冲区
        ByteBuffer buffer1 = ByteBuffer.allocate(10);
```

```
        // 包装一个现有的数组
        byte array[] = new byte[10];
        ByteBuffer buffer2 = ByteBuffer.wrap( array );
    }
}
```

4.缓冲区分片

在 NIO 中，除了可以分配或者包装一个缓冲区对象外，还可以根据现有的缓冲区对象来创建一个子缓冲区，即在现有缓冲区上切出一片来作为一个新的缓冲区，但现有的缓冲区与创建的子缓冲区在底层数组层面上是数据共享的，也就是说，子缓冲区相当于是现有缓冲区的一个视图窗口。调用 slice()方法可以创建一个子缓冲区，让我们通过例子来看一下：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.nio.ByteBuffer;

/**
 * 缓冲区分片
 */
public class BufferSlice {
    static public void main( String args[] ) throws Exception {
        ByteBuffer buffer = ByteBuffer.allocate( 10 );

        // 缓冲区中的数据 0-9
        for (int i=0; i<buffer.capacity(); ++i) {
            buffer.put( (byte)i );
        }

        // 创建子缓冲区
        buffer.position( 3 );
        buffer.limit( 7 );
        ByteBuffer slice = buffer.slice();

        // 改变子缓冲区的内容
        for (int i=0; i<slice.capacity(); ++i) {
            byte b = slice.get( i );
            b *= 10;
            slice.put( i, b );
        }

        buffer.position( 0 );
        buffer.limit( buffer.capacity() );

        while (buffer.remaining()>0) {
            System.out.println( buffer.get() );
        }
    }
}
```

在该示例中，分配了一个容量大小为 10 的缓冲区，并在其中放入了数据 0-9，而在该缓冲区基础之上又创建了一个子缓冲区，并改变子缓冲区中的内容，从最后输出的结果来看，只有子缓冲区"可见的"那部分数据发生了变化，并且说明子缓冲区与原缓冲区是数据共享的，输出结果如下所示：

5.只读缓冲区

只读缓冲区非常简单，可以读取它们，但是不能向它们写入数据。可以通过调用缓冲区的 asReadOnlyBuffer()方法，将任何常规缓冲区转 换为只读缓冲区，这个方法返回一个与原缓冲区完全相同的缓冲区，并与原缓冲区共享数据，只不过它是只读的。如果原缓冲区的内容发生了变化，只读缓冲区的内容也随之发生变化：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.nio.*;

/** 只读缓冲区 */
public class ReadOnlyBuffer {
    static public void main( String args[] ) throws Exception {
    ByteBuffer buffer = ByteBuffer.allocate( 10 );

    // 缓冲区中的数据 0-9
    for (int i=0; i<buffer.capacity(); ++i) {
        buffer.put( (byte)i );
    }

    // 创建只读缓冲区
    ByteBuffer readonly = buffer.asReadOnlyBuffer();

    // 改变原缓冲区的内容
    for (int i=0; i<buffer.capacity(); ++i) {
        byte b = buffer.get( i );
        b *= 10;
        buffer.put( i, b );
    }

    readonly.position(0);
    readonly.limit(buffer.capacity());

    // 只读缓冲区的内容也随之改变
    while (readonly.remaining()>0) {
        System.out.println( readonly.get());
    }
    }
}
```

如果尝试修改只读缓冲区的内容，则会报 ReadOnlyBufferException 异常。只读缓冲区对于保护数据很有用。在将缓冲区传递给某个 对象的方法时，无法知道这个方法是否会修改缓冲区中的数据。创建一个只读的缓冲区可以保证该缓冲区不会被修改。只可以把常规缓冲区转换为只读缓冲区，而不能将只读的缓冲区转换为可写的缓冲区。

6.直接缓冲区

直接缓冲区是为加快 I/O 速度，使用一种特殊方式为其分配内存的缓冲区，JDK 文档中的描述为：给定一个直接字节缓冲区，Java 虚拟机将尽最大努力直接对它执行本机 I/O 操作。也就是说，它会在每一次调用底层操作系统的本机 I/O 操作之前(或之后)，尝试避免将缓冲区的内容拷贝到一个中间缓冲区中或者从一个中间缓冲区中拷贝数据。要分配直接缓冲区，需要调用 allocateDirect() 方法，而不是 allocate()方法，使用方式与普通缓冲区并无区别，如下面的拷贝文件示例：

```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
/**
 * 直接缓冲区
 */
public class DirectBuffer {
   static public void main( String args[] ) throws Exception {

    //首先我们从磁盘上读取刚才我们写出的文件内容
       String infile = "E://test.txt";
       FileInputStream fin = new FileInputStream( infile );
       FileChannel fcin = fin.getChannel();

       //把刚刚读取的内容写入到一个新的文件中
       String outfile = String.format("E://testcopy.txt");
       FileOutputStream fout = new FileOutputStream( outfile );
       FileChannel fcout = fout.getChannel();

       // 使用 allocateDirect，而不是 allocate
       ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

       while (true) {
           buffer.clear();

           int r = fcin.read(buffer);

           if (r==-1) {
               break;
           }

           buffer.flip();

           fcout.write(buffer);
       }
   }
}
```

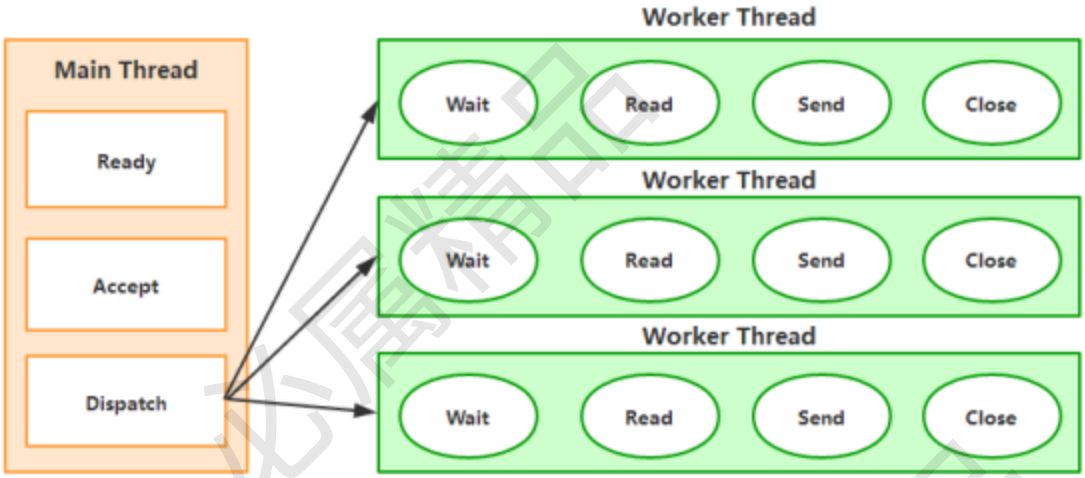
7.内存映射

内存映射是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的 I/O 快的多。内存映射文件 I/O 是通过使文件中的数据出现为 内存数组的内容来完成的，这其初听起来似乎不过就是将整个文件读到内存中，但是事实上并不是这样。一般来说，只有文件中实际读取或者写入的部分才会映射到内存中。如下面的示例代码：

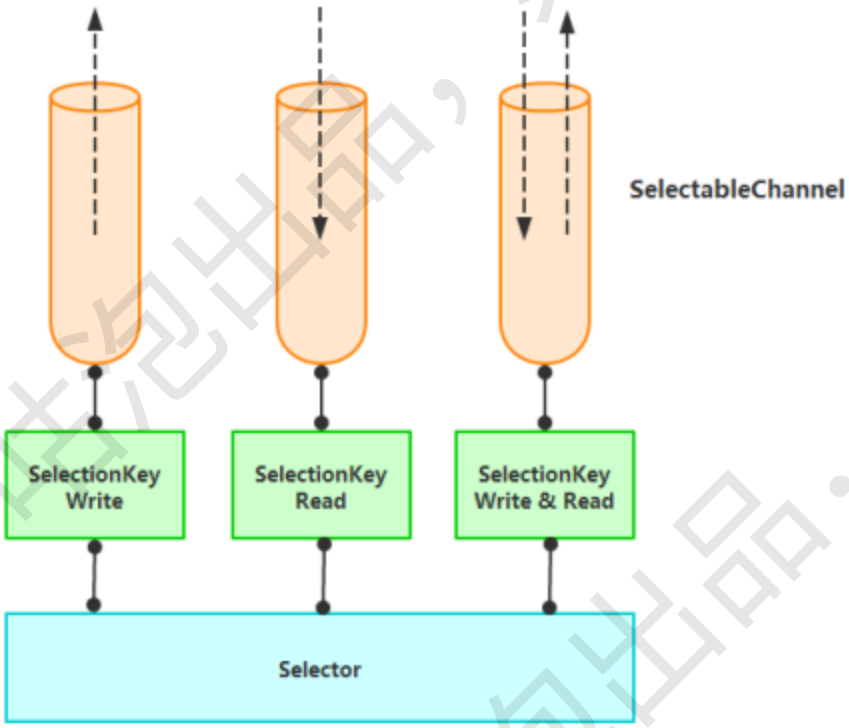
```
package com.gupaoedu.vip.netty.io.nio.buffer;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

/**
 * IO 映射缓冲区
 */
public class MappedBuffer {
    static private final int start = 0;
    static private final int size = 1024;

    static public void main( String args[] ) throws Exception {
        RandomAccessFile raf = new RandomAccessFile( "E://test.txt", "rw" );
        FileChannel fc = raf.getChannel();

        //把缓冲区跟文件系统进行一个映射关联
        //只要操作缓冲区里面的内容，文件内容也会跟着改变
        MappedByteBuffer mbb = fc.map( FileChannel.MapMode.READ_WRITE,start, size );

        mbb.put( 0, (byte)97 );
        mbb.put( 1023, (byte)122 );

        raf.close();
    }
}
```

1.2 选择器 Selector

传统的 Server/Client 模式会基于 TPR（Thread per Request），服务器会为每个客户端请求建立一个线程，由该线程单独负责处理一个客户请求。这种模式带来的一个问题就是线程数量的剧增，大量的线程会增大服务器的开销。大多数的实现为了避免这个问题，都采用了线程池模型，并设置线程池线程的最大数量，这又带来了新的问题，如果线程池中有 200 个线程，而有 200 个用户都在进行大文件下载，会导致第 201 个用户的请求无法及时处理，即便第 201 个用户只想请求一个几 KB 大小的页面。传统的 Server/Client 模式如下图所示：



NIO 中非阻塞 I/O 采用了基于 Reactor 模式的工作方式，I/O 调用不会被阻塞，相反是注册感兴趣的特定 I/O 事件，如可读数据到达，新的套接字连接等等，在发生特定事件时，系统再通知我们。NIO 中实现非阻塞 I/O 的核心对象就是 Selector，Selector 就是注册各种 I/O 事件地方，而且当那些事件发生时，就是这个对象告诉我们所发生的事件，如下图所示:



从图中可以看出，当有读或写等任何注册的事件发生时，可以从 Selector 中获得相应的 SelectionKey，同时从 SelectionKey 中可以找到发生的事件和该事件所发生的具体的 SelectableChannel，以获得客户端发送过来的数据。

使用NIO 中非阻塞I/O 编写服务器处理程序，大体上可以分为下面三个步骤：

1. 向 Selector 对象注册感兴趣的事件。

2. 从 Selector 中获取感兴趣的事件。

3. 根据不同的事件进行相应的处理。

接下来我们用一个简单的示例来说明整个过程。首先是向 Selector 对象注册感兴趣的事件：

```
/*
 * 注册事件
 */
private Selector getSelector() throws IOException {
    // 创建 Selector 对象
    Selector sel = Selector.open();

    // 创建可选择通道，并配置为非阻塞模式
    ServerSocketChannel server = ServerSocketChannel.open();
    server.configureBlocking(false);

    // 绑定通道到指定端口
    ServerSocket socket = server.socket();
    InetSocketAddress address = new InetSocketAddress(port);
    socket.bind(address);

    // 向 Selector 中注册感兴趣的事件
    server.register(sel, SelectionKey.OP_ACCEPT);
    return sel;
}
```

创建了 ServerSocketChannel 对象，并调用 configureBlocking()方法，配置为非阻塞模式，接下来的三行代码把该通道绑定到指定端口，最后向 Selector 中注册事件，此处指定的是参数是 OP_ACCEPT，即指定我们想要监听 accept 事件，也就是新的连接发 生时所产生的事件，对于 ServerSocketChannel 通道来说，我们唯一可以指定的参数就是 OP_ACCEPT。

从 Selector 中获取感兴趣的事件，即开始监听，进入内部循环：

```
/*
* 开始监听
*/
public void listen() {
   System.out.println("listen on " + port);
   try {
       while(true) {
           // 该调用会阻塞，直到至少有一个事件发生
           selector.select();
           Set<SelectionKey> keys = selector.selectedKeys();
```

```
            Iterator<SelectionKey> iter = keys.iterator();
            while (iter.hasNext()) {
                SelectionKey key = (SelectionKey) iter.next();
                iter.remove();
                process(key);
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

在非阻塞 I/O 中，内部循环模式基本都是遵循这种方式。首先调用 select()方法，该方法会阻塞，直到至少有一个事件发生，然后再使用 selectedKeys()方法获取发生事件的 SelectionKey，再使用迭代器进行循环。

最后一步就是根据不同的事件，编写相应的处理代码:

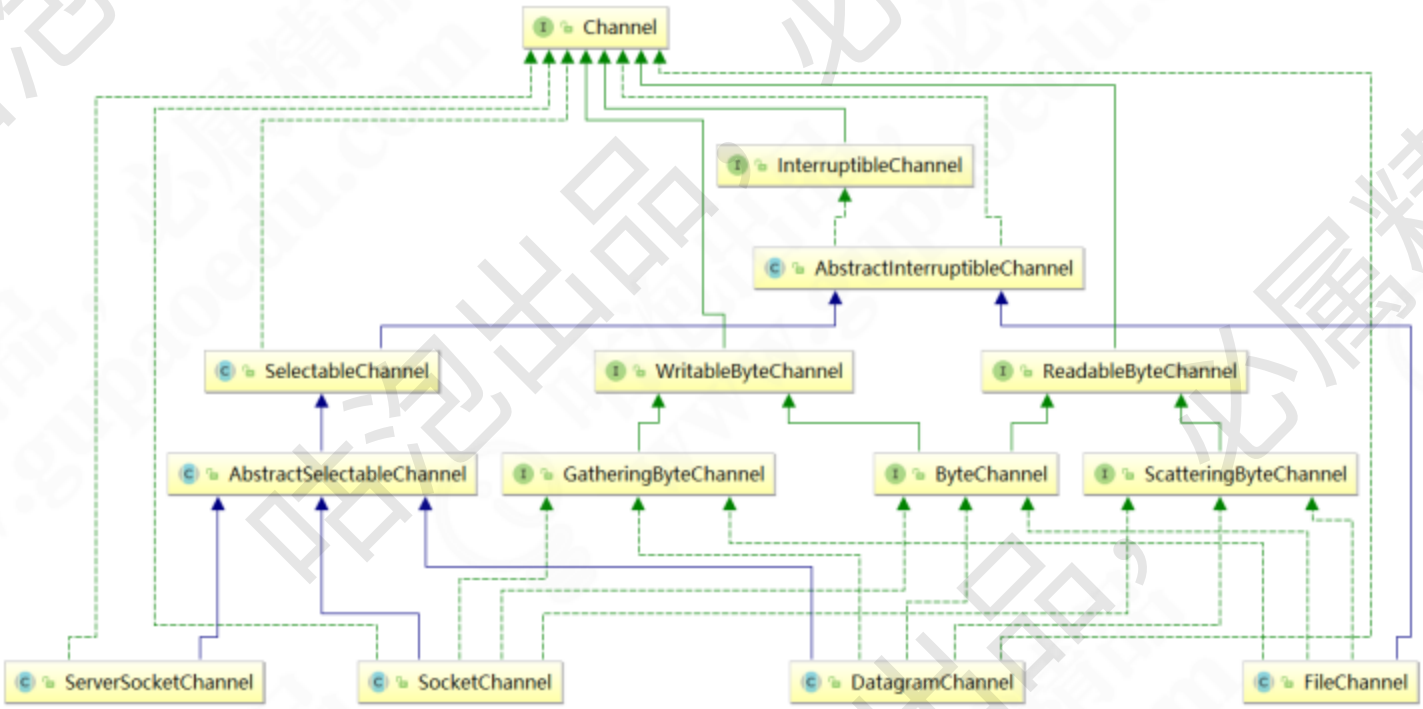
```
/*
 * 根据不同的事件做处理
 */
private void process(SelectionKey key) throws IOException{

    // 接收请求
    if (key.isAcceptable()) {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        SocketChannel channel = server.accept();
        channel.configureBlocking(false);
        channel.register(selector, SelectionKey.OP_READ);
    }

    // 读信息
    else if (key.isReadable()) {
        SocketChannel channel = (SocketChannel) key.channel();
        int len = channel.read(buffer);
        if (len > 0) {
            buffer.flip();
            content = new String(buffer.array(),0,len);
            SelectionKey sKey = channel.register(selector, SelectionKey.OP_WRITE);
            sKey.attach(content);
        } else {
            channel.close();
        }
        buffer.clear();
    }

    // 写事件
    else if (key.isWritable()) {
        SocketChannel channel = (SocketChannel) key.channel();
        String content = (String) key.attachment();
        ByteBuffer block = ByteBuffer.wrap(("输出内容: " + content).getBytes());
        if(block != null){
            channel.write(block);
        }else{
            channel.close();
        }
    }
}
```

此处分别判断是接受请求、读数据还是写事件，分别作不同的处理。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。

1.3 通道 Channel

通道是一个对象，通过它可以读取和写入数据，当然了所有数据都通过 Buffer 对象来处理。我们永远不会将字节直接写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区。同样不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

在 NIO 中，提供了多种通道对象，而所有的通道对象都实现了 Channel 接口。它们之间的继承关系如下图所示：



1.使用 NIO 读取数据

在前面我们说过，任何时候读取数据，都不是直接从通道读取，而是从通道读取到缓冲区。所以使用 NIO 读取数据可以分为下面三个步骤：

1. 从 FileInputStream 获取 Channel

2. 创建 Buffer

3. 将数据从 Channel 读取到 Buffer 中

下面是一个简单的使用 NIO 从文件中读取数据的例子：

2.使用NIO写入数据

使用 NIO 写入数据与读取数据的过程类似，同样数据不是直接写入通道，而是写入缓冲区，可以分为下面三个步骤：

1. 从 FileInputStream 获取 Channel。

2. 创建 Buffer。

3. 将数据从 Channel 写入到 Buffer 中。

```
package com.gupaoedu.vip.netty.io.nio.channel;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class FileInputDemo {
    static public void main( String args[] ) throws Exception {
        FileInputStream fin = new FileInputStream("E://test.txt");

        // 获取通道
        FileChannel fc = fin.getChannel();

        // 创建缓冲区
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 读取数据到缓冲区
        fc.read(buffer);

        buffer.flip();

        while (buffer.remaining() > 0) {
            byte b = buffer.get();
            System.out.print(((char)b));
        }

        fin.close();
    }
}
```

下面是一个简单的使用 NIO 向文件中写入数据的例子：

```
package com.gupaoedu.vip.netty.io.nio.channel;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class FileOutputDemo {
```

```
    static private final byte message[] = { 83, 111, 109, 101, 32,
        98, 121, 116, 101, 115, 46 };

    static public void main( String args[] ) throws Exception {
        FileOutputStream fout = new FileOutputStream( "E://test.txt" );

        FileChannel fc = fout.getChannel();

        ByteBuffer buffer = ByteBuffer.allocate( 1024 );

        for (int i=0; i<message.length; ++i) {
            buffer.put( message[i] );
        }

        buffer.flip();
        fc.write( buffer );
        fout.close();
    }
}
```

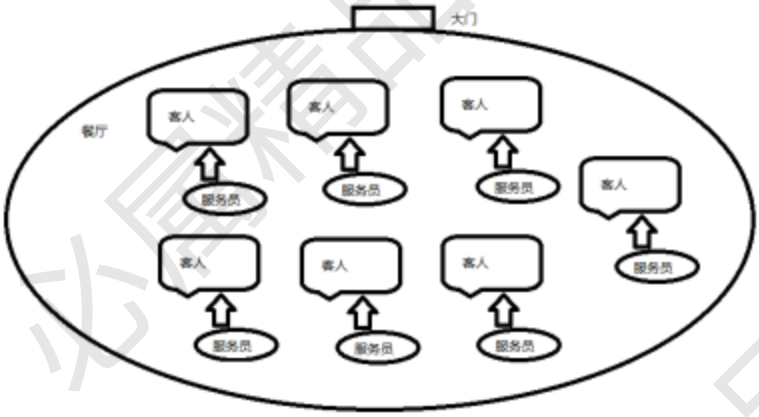
3.IO 多路复用

我们试想一下这样的现实场景：

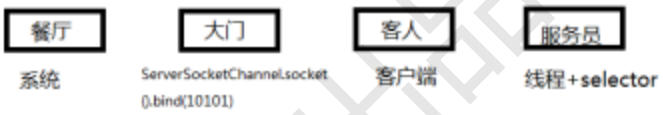
一个餐厅同时有 100 位客人到店，当然到店后第一件要做的事情就是点菜。但是问题来了，餐厅老板为了节约人力成本目前只有一位大堂服务员拿着唯一的一本菜单等待客人进行服务。

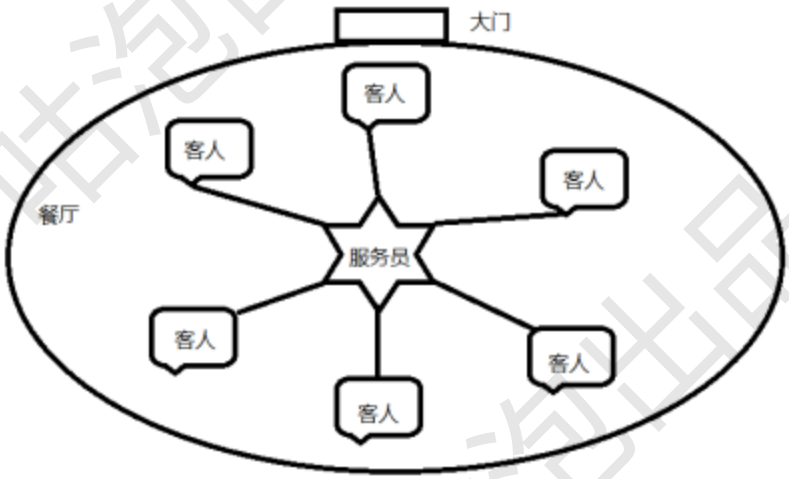
那么最笨（但是最简单）的方法是（方法 A），无论有多少客人等待点餐，服务员都把仅有的一份菜单递给其中一位客人，然后站在客人身旁等待这个客人完成点菜过程。在记录客人点菜内容后，把点菜记录交给后堂厨师。然后是第二位客人。。。。然后是第三位客人。很明显，只有脑袋被门夹过的老板，才会这样设置服务流程。因为随后的 80 位客人，再等待超时后就会离店（还会给差评）。

于是还有一种办法（方法 B），老板马上新雇佣 99 名服务员，同时印制 99 本新的菜单。每一名服务员手持一本菜单负责一位客人（关键不只在于服务员，还在于菜单。因为没有菜单客人也无法点菜）。在客人点完菜后，记录点菜内容交给后堂厨师（当然为了更高效，后堂厨师最好也有 100 名）。这样每一位客人享受的就是 VIP 服务咯，当然客人不会走，但是人力成本可是一个大头哦（亏死你）。



另外一种办法（方法 C），就是改进点菜的方式，当客人到店后，自己申请一本菜单。想好自己要点的才后，就呼叫服务员。服务员站在自己身边后记录客人的菜单内容。将菜单递给厨师的过程也要进行改进，并不是每一份菜单记录好以后，都要交给后堂厨师。服务员可以记录号多份菜单后，同时交给厨师就行了。那么这种方式，对于老板来说人力成本是最低的；对于客人来说，虽然不再享受 VIP 服务并且要进行一定的等待，但是这些都是可接受的；对于服务员来说，基本上她的时间都没有浪费，基本上被老板压杆了最后一滴油水。





如果您是老板，您会采用哪种方式呢?

到店情况：并发量。到店情况不理想时，一个服务员一本菜单，当然是足够了。所以不同的老板在不同的场合下，将会灵活选择服务员和菜单的配置。

客人：客户端请求

点餐内容：客户端发送的实际数据

老板：操作系统

人力成本：系统资源

菜单：文件状态描述符（FD）。操作系统对于一个进程能够同时持有的文件状态描述符的个数是有限制的，在 linux 系统中$ulimit -n 查看这个限制值，当然也是可以（并且应该）进行内核参数调整的。

服务员：操作系统内核用于 IO 操作的线程（内核线程）

厨师：应用程序线程（当然厨房就是应用程序进程咯）

方法 A：同步 IO

方法 B：同步 IO

方法 C：多路复用 IO

目前流行的多路复用 IO 实现主要包括四种：select、poll、epoll、kqueue。下表是他们的一些重要特性的比较：

IO模型	相对性能	关键思路	操作系统	JAVA支持
select	较高	Reactor	windows/Linux	支持,Reactor 模式(反应器设计模式)。Linux 操作系统的 kernels 2.4 内核版本之前，默认使用 select；而目前 windows 下对同步 IO 的支持，都是 select 模型。

poll	较高	Reactor	Linux	Linux 下的 JAVA NIO 框架，Linux kernels 2.6 内核版本之前使用 poll 进行支持。也是使用的 Reactor 模式。
epoll	高	Reactor/Proactor	Linux	Linux kernels 2.6 内核版本及以后使用 epoll 进行支持；Linux kernels 2.6 内核版本之前使用 poll 进行支持；另外一定注意，由于 Linux 下没有 Windows 下的 IOCP 技术提供真正的 异步 IO 支持，所以 Linux 下使用 epoll 模拟异步 IO。
kqueue	高	Proactor	Linux	目前 JAVA 的版本不支持。

多路复用 IO 技术最适用的是"高并发"场景，所谓高并发是指 1 毫秒内至少同时有上千个连接请求准备好。其他情况下多路复用 IO 技术发挥不出来它的优势。另一方面，使用 JAVA NIO 进行功能实现，相对于传统的 Socket 套接字实现要复杂一些，所以实际应用中，需要根据自己的业务需求进行技术选择。

2 NIO 源码初探

说到源码先得从 Selector 的 open 方法开始看起.java.nio.channels.Selector:

```
public static Selector open() throws IOException {
    return SelectorProvider.provider().openSelector();
}
```

看看 SelectorProvider.*provider*()做了什么:

```
public static SelectorProvider provider() {
        synchronized (lock) {
            if (provider != null)
                return provider;
            return AccessController.doPrivileged(
                new PrivilegedAction<SelectorProvider>() {
                    public SelectorProvider run() {
                            if (loadProviderFromProperty())
                                return provider;
                            if (loadProviderAsService())
                                return provider;
                            provider = sun.nio.ch.DefaultSelectorProvider.create();
                            return provider;
```

```
                }
            });
        }
}
```

其中 *provider* = sun.nio.ch.DefaultSelectorProvider.*create*();会根据操作系统来返回不同的实现类，windows 平台就返回

WindowsSelectorProvider；而 **if** (*provider* != **null**) **return** *provider*;

保证了整个 server 程序中只有一个 WindowsSelectorProvider 对象；

再看看 WindowsSelectorProvider. openSelector():

```
public AbstractSelector openSelector() throws IOException {
    return new WindowsSelectorImpl(this);
}
```

new WindowsSelectorImpl(SelectorProvider)代码：

```
WindowsSelectorImpl(SelectorProvider sp) throws IOException {
    super(sp);
    pollWrapper = new PollArrayWrapper(INIT_CAP);
    wakeupPipe = Pipe.open();
    wakeupSourceFd = ((SelChImpl)wakeupPipe.source()).getFDVal();

    // Disable the Nagle algorithm so that the wakeup is more immediate
    SinkChannelImpl sink = (SinkChannelImpl)wakeupPipe.sink();
    (sink.sc).socket().setTcpNoDelay(true);
    wakeupSinkFd = ((SelChImpl)sink).getFDVal();

    pollWrapper.addWakeupSocket(wakeupSourceFd, 0);
}
```

其中 Pipe.*open*()是关键，这个方法的调用过程是：

```
public static Pipe open() throws IOException {
    return SelectorProvider.provider().openPipe();
}
```

SelectorProvider 中：

```
public Pipe openPipe() throws IOException {
    return new PipeImpl(this);
}
```

再看看怎么 new PipeImpl()的：

```
PipeImpl(SelectorProvider sp) {
    long pipeFds = IOUtil.makePipe(true);
    int readFd = (int) (pipeFds >>> 32);
    int writeFd = (int) pipeFds;
    FileDescriptor sourcefd = new FileDescriptor();
    IOUtil.setfdVal(sourcefd, readFd);
    source = new SourceChannelImpl(sp, sourcefd);
    FileDescriptor sinkfd = new FileDescriptor();
    IOUtil.setfdVal(sinkfd, writeFd);
    sink = new SinkChannelImpl(sp, sinkfd);
}
```

其中 IOUtil.makePipe(**true**)是个 native 方法：

```
/**
  * Returns two file descriptors for a pipe encoded in a long.
  * The read end of the pipe is returned in the high 32 bits,
  * while the write end is returned in the low 32 bits.
  */
staticnativelong makePipe(boolean blocking);
```

具体实现：

```
JNIEXPORT jlong JNICALL
Java_sun_nio_ch_IOUtil_makePipe(JNIEnv *env, jobject this, jboolean blocking)
{
 int fd[2];

 if (pipe(fd) < 0) {
     JNU_ThrowIOExceptionWithLastError(env, "Pipe failed");
     return 0;
 }
 if (blocking == JNI_FALSE) {
     if ((configureBlocking(fd[0], JNI_FALSE) < 0)
         || (configureBlocking(fd[1], JNI_FALSE) < 0)) {
         JNU_ThrowIOExceptionWithLastError(env, "Configure blocking failed");
         close(fd[0]);
         close(fd[1]);
         return 0;
     }
 }
 return ((jlong) fd[0] << 32) | (jlong) fd[1];
}
static int
configureBlocking(int fd, jboolean blocking)
{
 int flags = fcntl(fd, F_GETFL);
 int newflags = blocking ? (flags & ~O_NONBLOCK) : (flags | O_NONBLOCK);

 return (flags == newflags) ? 0 : fcntl(fd, F_SETFL, newflags);
}
```

正如这段注释所描述的：

```
    /**
      * Returns two file descriptors for a pipe encoded in a long.
      * The read end of the pipe is returned in the high 32 bits,
      * while the write end is returned in the low 32 bits.
      */
```

High32 位存放的是通道 read 端的文件描述符 FD（file descriptor），low 32 bits 存放的是 write 端的文件描述符。所以取到 makepipe（）返回值后要做移位处理。

```
pollWrapper.addWakeupSocket(wakeupSourceFd, 0);
```

这行代码把返回的 pipe 的 write 端的 FD 放在了 pollWrapper 中（后面会发现，这么做是为了实现 selector 的 wakeup()）

ServerSocketChannel.open()的实现：

```
    public static ServerSocketChannel open() throws IOException {
```

```
        return SelectorProvider.provider().openServerSocketChannel();
    }
```

SelectorProvider:

```
public ServerSocketChannel openServerSocketChannel() throws IOException {
    return new ServerSocketChannelImpl(this);
}
```

可见创建的 ServerSocketChannelImpl 也有 WindowsSelectorImpl 的引用。

```
public ServerSocketChannelImpl(SelectorProvider sp) throws IOException {
    super(sp);
    this.fd =  Net.serverSocket(true);
    this.fdVal = IOUtil.fdVal(fd);
    this.state = ST_INUSE;
}
```

然后通过 serverChannel1.register(selector, SelectionKey.OP_ACCEPT);把 selector 和 channel 绑定在一起，也就是把 new

ServerSocketChannel 时创建的 FD 与 selector 绑定在了一起。

到此，server 端已启动完成了，主要创建了以下对象：

WindowsSelectorProvider：单例

WindowsSelectorImpl 中包含：

pollWrapper：保存 selector 上注册的 FD，包括 pipe 的 write 端 FD 和 ServerSocketChannel 所用的 FD

wakeupPipe：通道（其实就是两个 FD，一个 read，一个 write）

再到 Server 中的 run():

selector.select();主要调用了 WindowsSelectorImpl 中的这个方法：

```
protected int doSelect(long timeout) throws IOException {
        if (channelArray == null)
            throw new ClosedSelectorException();
        this.timeout = timeout; // set selector timeout
        processDeregisterQueue();
        if (interruptTriggered) {
            resetWakeupSocket();
            return 0;
        }
        // Calculate number of helper threads needed for poll. If necessary
        // threads are created here and start waiting on startLock
        adjustThreadsCount();
        finishLock.reset(); // reset finishLock
        // Wakeup helper threads, waiting on startLock, so they start polling.
        // Redundant threads will exit here after wakeup.
        startLock.startThreads();
        // do polling in the main thread. Main thread is responsible for
        // first MAX_SELECTABLE_FDS entries in pollArray.
        try {
            begin();
            try {
```

```
                subSelector.poll();
            } catch (IOException e) {
                finishLock.setException(e); // Save this exception
            }
            // Main thread is out of poll(). Wakeup others and wait for them
            if (threads.size() > 0)
                finishLock.waitForHelperThreads();
          } finally {
              end();
          }
        // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
        finishLock.checkForException();
        processDeregisterQueue();
        int updated = updateSelectedKeys();
        // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
        resetWakeupSocket();
        return updated;
    }
```

其中 subSelector.poll()是核心，也就是轮训 pollWrapper 中保存的 FD；具体实现是调用 native 方法 poll0:

```
  private int poll() throws IOException{ // poll for the main thread
      return poll0(pollWrapper.pollArrayAddress,
             Math.min(totalChannels, MAX_SELECTABLE_FDS),
             readFds, writeFds, exceptFds, timeout);
  }
  private native int poll0(long pollAddress, int numfds,
             int[] readFds, int[] writeFds, int[] exceptFds, long timeout);
    // These arrays will hold result of native select().
    // The first element of each array is the number of selected sockets.
    // Other elements are file descriptors of selected sockets.
    private final int[] readFds = new int [MAX_SELECTABLE_FDS + 1];//保存发生 read 的 FD
    private final int[] writeFds = new int [MAX_SELECTABLE_FDS + 1]; //保存发生 write 的 FD
    private final int[] exceptFds = new int [MAX_SELECTABLE_FDS + 1]; //保存发生 except 的 FD
```

这个 poll0（）会监听 pollWrapper 中的 FD 有没有数据进出，这会造成 IO 阻塞，直到有数据读写事件发生。比如，由于 pollWrapper 中保存的也有 ServerSocketChannel 的 FD，所以只要 ClientSocket 发一份数据到 ServerSocket,那么 poll0（）就会返回；又由于 pollWrapper 中保存的也有 pipe 的 write 端的 FD，所以只要 pipe 的 write 端向 FD 发一份数据，也会造成 poll0（）返回；如果这两种情况都没有发生，那么 poll0（）就一直阻塞，也就是 selector.select()会一直阻塞；如果有任何一种情况发生，那么 selector.select()就会返回，所有在 OperationServer 的 run()里要用 **while**(**true**){，这样就可以保证在 selector 接收到数据并处理完后继续监听 poll();

这时再来看看 WindowsSelectorImpl. Wakeup():

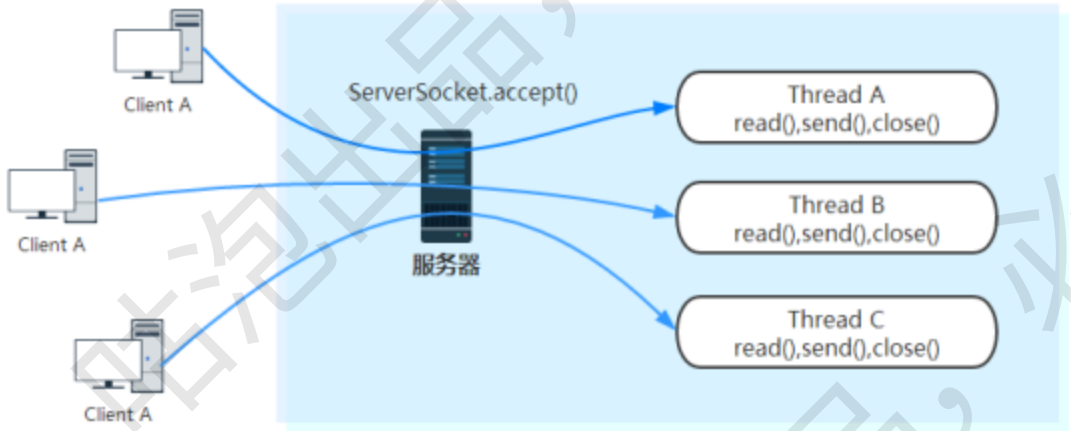
```
  public Selector wakeup() {
      synchronized (interruptLock) {
          if (!interruptTriggered) {
              setWakeupSocket();
              interruptTriggered = true;
          }
      }
      return this;
  }
```

```
// Sets Windows wakeup socket to a signaled state.
    private void setWakeupSocket() {
        setWakeupSocket0(wakeupSinkFd);
    }
private native void setWakeupSocket0(int wakeupSinkFd);
JNIEXPORT void JNICALL
Java_sun_nio_ch_WindowsSelectorImpl_setWakeupSocket0(JNIEnv *env, jclass this,
                                                     jint scoutFd)
{
    /* Write one byte into the pipe */
    const char byte = 1;
    send(scoutFd, &byte, 1, 0);
}
```

可见 wakeup()是通过 pipe 的 write 端 send(scoutFd, &byte, 1, 0)，发生一个字节 1，来唤醒 poll（）。所以在需要的时候就可以调用 selector.wakeup()来唤醒 selector。

3 反应堆 Reactor

现在我们已经对阻塞 I/O 已有了一定了解，我们知道阻塞 I/O 在调用 InputStream.read()方法时是阻塞的，它会一直等到数据到来时（或超时）才会返回；同样，在调用 ServerSocket.accept()方法时，也会一直阻塞到有客户端连接才会返回，每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。阻塞 I/O 的通信模型示意图如下：



如果你细细分析，一定会发现阻塞 I/O 存在一些缺点。根据阻塞 I/O 通信模型，我总结了它的两点缺点：

1. 当客户端多时，会创建大量的处理线程。且每个线程都要占用栈空间和一些 CPU 时间

2. 阻塞可能带来频繁的上下文切换，且大部分上下文切换可能是无意义的。在这种情况下非阻塞式 I/O 就有了它的应用前景。

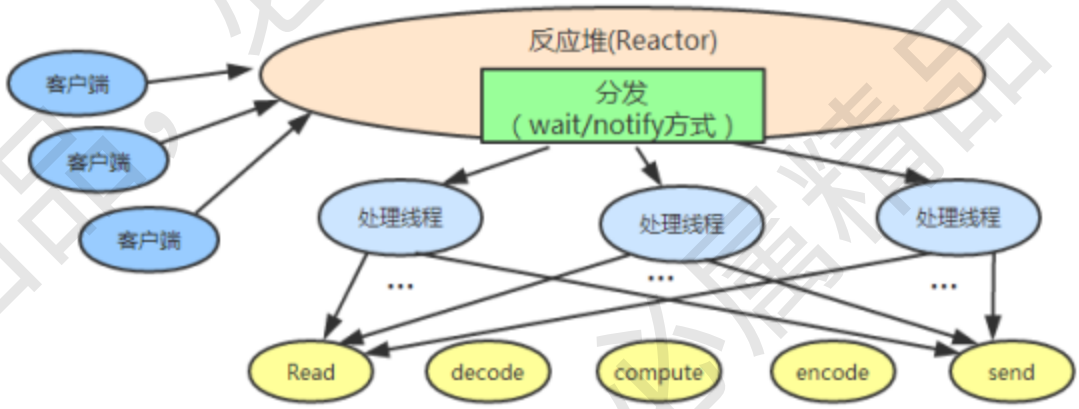
Java NIO 是在 jdk1.4 开始使用的，它既可以说成"新 I/O"，也可以说成非阻塞式 I/O。下面是 Java NIO 的工作原理：

1. 由一个专门的线程来处理所有的 IO 事件，并负责分发。

2. 事件驱动机制：事件到的时候触发，而不是同步的去监视事件。

3. 线程通讯：线程之间通过 wait,notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的线程切换。

下面贴出我理解的 Java NIO 反应堆的工作原理图：



（注：每个线程的处理流程大概都是读取数据、解码、计算处理、编码、发送响应。）

4 Netty 与 NIO

4.1 Netty 支持的功能与特性

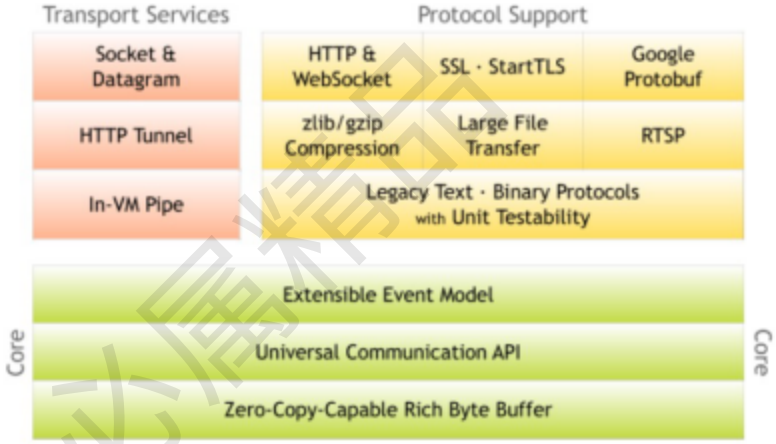
按照定义来说，Netty 是一个异步、事件驱动的用来做高性能、高可靠性的网络应用框架。主要的优点有：

1. 框架设计优雅，底层模型随意切换适应不同的网络协议要求。

2. 提供很多标准的协议、安全、编码解码的支持。

3. 解决了很多 NIO 不易用的问题。

4. 社区更为活跃，在很多开源框架中使用，如 Dubbo、RocketMQ、Spark 等。



上图体现的主要是 Netty 支持的功能或者特性：

1.底层核心有：Zero-Copy-Capable Buffer，非常易用的灵拷贝 Buffer（这个内容很有意思，稍后专门来说）；统一的 API；标准可扩展的时间模型

2.传输方面的支持有：管道通信（具体不知道干啥的，还请老司机指教）；Http 隧道；TCP 与 UDP

3.协议方面的支持有：基于原始文本和二进制的协议；解压缩；大文件传输；流媒体传输；protobuf 编解码；安全认证；http 和 websocket

4.2 Netty 采用 NIO 而非 AIO 的理由

1.Netty 不看重 Windows 上的使用，在 Linux 系统上，AIO 的底层实现仍使用 EPOLL，没有很好实现 AIO，因此在性能上没有明显的优势，而且被 JDK 封装了一层不容易深度优化

2.Netty 整体架构是 reactor 模型，而 AIO 是 proactor 模型，混合在一起会非常混乱,把 AIO 也改造成 reactor 模型看起来是把 epoll 绕个弯又绕回来

3.AIO还有个缺点是接收数据需要预先分配缓存，而不是NIO那种需要接收时才需要分配缓存，所以对连接数量非常大但流量小的情况，内存浪费很多

4.Linux 上 AIO 不够成熟，处理回调结果速度跟不到处理需求，比如外卖员太少，顾客太多，供不应求，造成处理速度有瓶颈（待验证）