NIO

讲师：子慕

1. 什么是NIO

IO回顾

- IO：Input OutPut（输入 输出）
- IO技术的作用：解决设备和设备之间的数据传输问题
- IO的应用场景：图片上传、下载、打印机打印信息表、解析XML...

1.1 概念

- 即 `Java New IO`
- 是1个全新的、 `JDK 1.4`后提供的 `IO API`
- Java API中提供了两套NIO，一套是针对`标准输入输出NIO`，另一套就是`网络编程NIO`

1.2 作用

- `NIO`和`IO`有相同的作用和目的，但实现方式不同
- 可替代 标准`Java IO`的`IO API`
- IO是以**流**的方式处理数据，而NIO是以**块**的方式处理数据。

1.3 流与块的比较

- NIO和IO最大的区别是数据打包和传输方式。
- IO是以**流**的方式处理数据，而NIO是以**块**的方式处理数据。

面向流的IO一次一个字节的处理数据，一个输入流产生一个字节，一个输出流就消费一个字节。

面向块的IO系统以块的形式处理数据。每一个操作都在一步中产生或消费一个数据块。按块要比按流快的多

（举例：拿水龙头来比喻：流就像水龙头滴水，每次只有一滴；块就像水龙头往水壶放水，放满之后对一整个水壶的水进行操作）

1.4 新特性

对比于`Java IO`，`NIO`具备的新特性如下：

IO	NIO
面向流(Stream Oriented)	面向缓冲区(Buffer Oriented)
阻塞IO(Blocking IO)	非阻塞IO(Non Blocking IO)
(无)	选择器(Selectors)

- 可简单认为：**IO是面向流的处理，NIO是面向块(缓冲区)的处理**
  - 面向流的I/O 系统**一次一个字节地处理数据**
  - 一个面向块(缓冲区)的I/O系统**以块的形式处理数据**

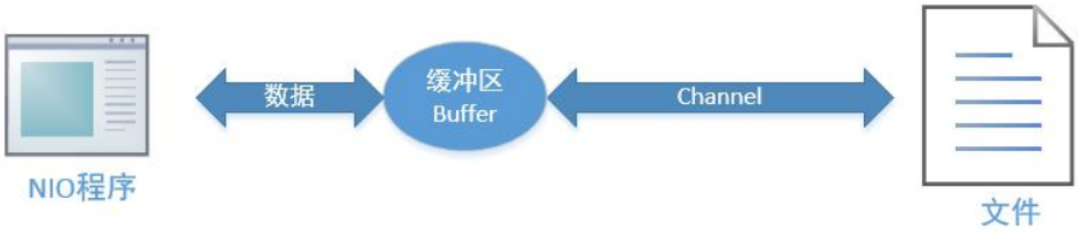
1.5 核心组件

`Java NIO`的核心组件 包括：

- 通道（`Channel`）
- 缓冲区（`Buffer`）
- 选择器（`Selector`）

在NIO中并不是以流的方式来处理数据的，而是以buffer缓冲区和Channel管道**配合使用**来处理数据。

Selector是因为NIO可以使用异步的非阻塞模式才加入的东西



简单理解一下：

- Channel管道比作成铁路，buffer缓冲区比作成火车(运载着货物)

  而我们的NIO就是**通过Channel管道运输着存储数据的Buffer缓冲区的来实现数据的处理**！

- 要时刻记住：Channel不与数据打交道，它只负责运输数据。与数据打交道的是Buffer缓冲区
-
  - **Channel-->运输**
  - **Buffer-->数据**

相对于传统IO而言，**流是单向的。**对于NIO而言，有了Channel管道这个概念，我们的**读写都是双向**的(铁路上的火车能从广州去北京、自然就能从北京返还到广州)！

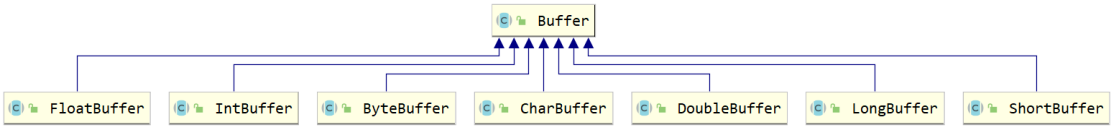
2. Buffer缓冲区

2.1 Buffer缓冲区概述

作用：缓冲区，用来存放具体要被传输的数据，比如文件、scoket 等。这里将数据装入 Buffer 再通过通道进行传输。

Buffer 就是一个数组，用来保存不同数据类型的数据

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



- ByteBuffer：存储字节数据到缓冲区
- ShortBuffer：存储字符串数据到缓冲区
- CharBuffer： 存储字符数据到缓冲区
- IntBuffer：存储整数数据到缓冲区
- LongBuffer：存储长整型数据到缓冲区
- DoubleBuffer：存储小数到缓冲区
- FloatBuffer：存储小数到缓冲区

对于 Java 中的基本数据类型，都有一个 Buffer 类型与之相对应，最常用的自然是**ByteBuffer** 类（二进制数据）

2.2 ByteBuffer的创建方式

代码演示

- 在堆中创建缓冲区：allocate(int capacity)
- 在系统内存创建缓冲区：allocateDirect(int capacity)
- 通过普通数组创建缓冲区：wrap(byte[] arr)

```
import java.nio.ByteBuffer;

public class Demo01Buffer创建方式 {
    public static void main(String[] args) {
        //在堆中创建缓冲区：allocate(int capacity)*
        ByteBuffer buffer1 = ByteBuffer.allocate(10);

        //在系统内存创建缓冲区：allocateDirect(int capacity)
        ByteBuffer buffer2 = ByteBuffer.allocateDirect(10);

        //通过普通数组创建缓冲区：wrap(byte[] arr)
        byte[] arr = {97,98,99};
        ByteBuffer buffer3 = ByteBuffer.wrap(arr);
    }
}
```

2.3 常用方法

拿到一个缓冲区我们往往会做什么？很简单，就是**读取缓冲区的数据/写数据到缓冲区中。**

所以，缓冲区的核心方法就是:

- put(byte b) : 给数组添加元素
- get() :获取一个元素

```java
package com.lagou;

import java.nio.ByteBuffer;
import java.util.Arrays;

public class Demo02Buffer的方法 {

    public static void main(String[] args) {

    //创建对象
    ByteBuffer buffer = ByteBuffer.allocate(10);

    //put(byte b) : 给数组添加元素
    buffer.put((byte)10);
    buffer.put((byte)20);
    buffer.put((byte)30);

    //把缓冲数组变成普通数组
    byte[] arr = buffer.array();

    //打印
    System.out.println(Arrays.toString(arr));

    //get() :获取一个元素
    byte b = buffer.get(1);
    System.out.println(b);   //20

    }
}
```

Buffer类维护了4个核心变量属性来提供**关于其所包含的数组的信息。**它们是：

- 容量Capacity
- - **缓冲区能够容纳的数据元素的最大数量。**容量在缓冲区创建时被设定，并且永远不能被改变。(不能被改变的原因也很简单，底层是数组嘛)
- 界限Limit
- - **缓冲区中可以操作数据的大小**，代表了当前缓冲区中一共有多少数据（从limit开始后面的位置不能操作）。
- 位置Position
- - **下一个要被读或写的元素的位置。**Position会自动由相应的 `get( )` 和 `put( )` 函数更新。
  - 以上三个属性值之间有一些相对大小的关系：0 <= position <= limit <= capacity
    例：- 如果我们创建一个新的容量大小为20 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。
- 标记Mark
- - 一个备忘位置。**用于记录上一次读写的位置。**

2.4 buffer代码演示

首先展示一下**是创建缓冲区后，核心变量的值是怎么变化的**

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

        // 创建一个缓冲区
        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

        // 看一下初始时4个核心变量的值
        System.out.println("初始时-->limit--->"+byteBuffer.limit());
        System.out.println("初始时-->position--->"+byteBuffer.position());
        System.out.println("初始时-->capacity--->"+byteBuffer.capacity());
        System.out.println("初始时-->mark--->" + byteBuffer.mark());

        System.out.println("---------------------------------------");

        // 添加一些数据到缓冲区中
        String s = "JavaEE";
        byteBuffer.put(s.getBytes());

        // 看一下初始时4个核心变量的值
        System.out.println("put完之后-->limit--->"+byteBuffer.limit());
        System.out.println("put完之后-->position--->"+byteBuffer.position());
        System.out.println("put完之后-->capacity--->"+byteBuffer.capacity());
        System.out.println("put完之后-->mark--->" + byteBuffer.mark());
    }
```

运行结果：

```
初始时-->limit--->10
初始时-->position--->0
初始时-->capacity--->10
初始时-->mark--->java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
---------------------
put完后-->limit--->10
put完后-->position--->6
put完后-->capacity--->10
put完后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=10 cap=10]
```

现在**我想要从缓存区拿数据**，怎么拿呀？？NIO给了我们一个`flip()`方法。这个方法可以**改动position和limit的位置**！

还是上面的代码，我们`flip()`一下后，再看看4个核心属性的值会发生什么变化：

```
put完后-->position--->6
put完后-->limit--->10
put完后-->capacity--->10
put完后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=10 cap=10]
------------------
flip完后-->position--->0
flip完后-->limit--->6
flip完后-->capacity--->10
flip完后-->mark--->java.nio.HeapByteBuffer[pos=0 lim=6 cap=10]
```

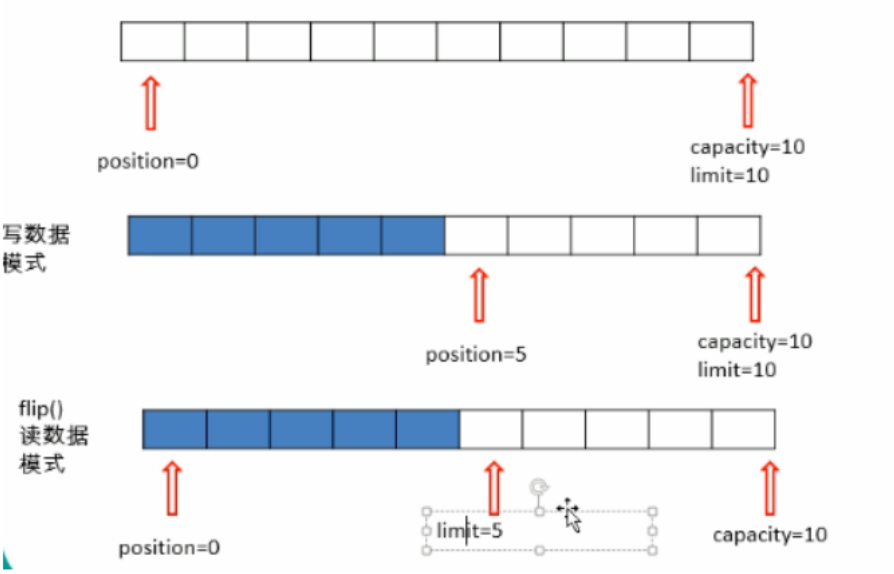
很明显的是：

- **limit变成了position的位置了**
- **而position变成了0**

看到这里的同学可能就会想到了：当调用完`filp()`时：**limit是限制读到哪里，而position是从哪里读**

一般我们称`filp()`为**“切换成读模式”**

- 每当要从缓存区的时候读取数据时，就调用`filp()`**“切换成读模式”**。



切换成读模式之后，我们就可以读取缓冲区的数据了：

```
        // 创建一个limit()大小的字节数组(因为就只有limit这么多个数据可读)
        byte[] bytes = new byte[byteBuffer.limit()];

        // 将读取的数据装进我们的字节数组中
        byteBuffer.get(bytes);

        // 输出数据
        System.out.println(new String(bytes, 0, bytes.length));
```

随后输出一下核心变量的值看看：

```
flip完后-->position--->0
flip完后-->limit--->6
flip完后-->capacity--->10
flip完后-->mark--->java.nio.HeapByteBuffer[pos=0 lim=6 cap=10]
--------------------
get完后-->position--->6
get完后-->limit--->6
get完后-->capacity--->10
get完后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=6 cap=10]
javaEE
```

get方法执行同样会影响position的变化

读完我们还想写数据到缓冲区，那就使用`clear()`函数，这个函数会"清空"缓冲区：

- 数据没有真正被清空，只是被**遗忘**掉了

```
clear完后-->position--->0
clear完后-->limit--->10
clear完后-->capacity--->10
clear完后-->mark--->java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
j
```

"清空缓冲区" -> 核心变量回归"写模式" 缓冲区数据是
没有清空的，但被"遗忘了"，因为操作数据的核心变量都被还原了

3.Channel通道

3.1 Channel通道概述

通道（Channel）：由 java.nio.channels 包定义 的。Channel 表示 IO 源与目标打开的连接。

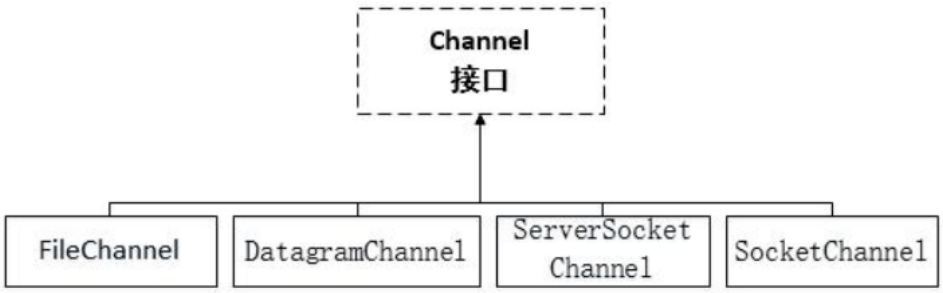
Channel 类似于传统的"**流**"。

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（白话: 就是数据传输用的通道，作用是打开到IO设备的连接，文件、套接字都行）

例：相当于一根管子，buffer中的数据可以通过管子写入被操作的资源当中，也可以将资源通过管子写入到buffer中去

3.2 Channel API

Java 为 Channel 接口提供的最主要实现类如下：



- FileChannel：用于读取、写入、映射和操作文件的通道。
- DatagramChannel：通过 UDP 读写网络中的数据通道。
- SocketChannel：通过 TCP 读写网络中的数据。
- ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来 的连接都会创建一个 SocketChannel。

3.3 FileChannel基本使用

- 使用FileChannel完成文件的复制

```
package com.lagou.buffer;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class Channel完成文件复制 {

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

        //通道依赖于IO流
        // 输入流
        FileInputStream fileInputStream = new
FileInputStream("C:\\Users\\sunzh\\Desktop\\wxy.png");
        // 输出流
        FileOutputStream fileOutputStream = new
FileOutputStream("C:\\Users\\sunzh\\Desktop\\lagou_myself\\nio\\复制.png");

        //使用流获取通道
        FileChannel f1 = fis.getChannel();
        FileChannel f2 = fos.getChannel();

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

        //循环
        while(f1.read(buffer) != -1){
            //切换
            buffer.flip();
            //输出
            f2.write(buffer);
            //还原所有指针位置
            buffer.clear();
```

```
        }

        //关流
        fos.close();
        fis.close();
    }
}
```

3.4 网络编程收发信息

- 客户端

```
package com.lagou.buffer;

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

public class Demo客户端 {

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

        //创建客户端
        SocketChannel sc = SocketChannel.open();
        //指定要连接的服务器ip和端口
        sc.connect(new InetSocketAddress("127.0.0.1",9000));

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

        //给数组添加数据
        buffer.put("哈哈哈".getBytes());

        //切换
        buffer.flip();

        //输出数据
        sc.write(buffer);

        //关闭资源
        sc.close();
    }
}
```

- 服务器端

```
package com.lagou.buffer;

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

public class Demo服务端 {
```

```
    //阻塞的
    public static void main(String[] args) throws IOException {

        //创建服务端对象
        ServerSocketChannel ssc = ServerSocketChannel.open();
        //绑定端口号
        ssc.bind(new InetSocketAddress(9000));

        //连接客户端
        SocketChannel sc = ssc.accept();

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

        //读取数据
        int len = sc.read(buffer);

        //打印
        System.out.println(new String(buffer.array(),0,len));
    }
}
```

3.5 accept阻塞问题

- 服务器端

```
package com.lagou.buffer;

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

public class Demo服务端非阻塞 {
    //阻塞的
    public static void main(String[] args) throws IOException,
InterruptedException {

        //创建服务端对象
        ServerSocketChannel ssc = ServerSocketChannel.open();
        //绑定端口号
        ssc.bind(new InetSocketAddress(9000));

        //设置非阻塞
        ssc.configureBlocking(false);

        while(true) {
            //连接客户端
            //如果连接成功就是sc对象,如果没有连接就是sc=null
            SocketChannel sc = ssc.accept();

            //判断
            if(sc != null) {
                //创建缓冲数组
```

```
                ByteBuffer buffer = ByteBuffer.allocate(1024);

                //读取数据
                int len = sc.read(buffer);

                //打印
                System.out.println(new String(buffer.array(), 0, len));

                //结束循环
                break;
            }else{
                //没有客户访问
                //在这里可以写别的业务代码
                System.out.println("去忙点别的事儿...");
                Thread.sleep(3000);

            }
        }
    }
}
```

4.Selector选择器

3.1 多路复用的概念

一个选择器可以同时监听多个服务器端口, 帮多个服务器端口同时等待客户端的访问

3.2 Selector的和Channel的关系

Channel和Buffer比较好理解 ，联系也比较密切，他们的关系简单来说就是：数据总是从通道中读到buffer缓冲区内，或者从buffer写入到通道中。

选择器和他们的关系又是什么？

选择器（Selector）是 Channel（通道）的多路复用器，Selector 可以同时监控多个 通道的 IO（输入输出）状况。

Selector的作用是什么？

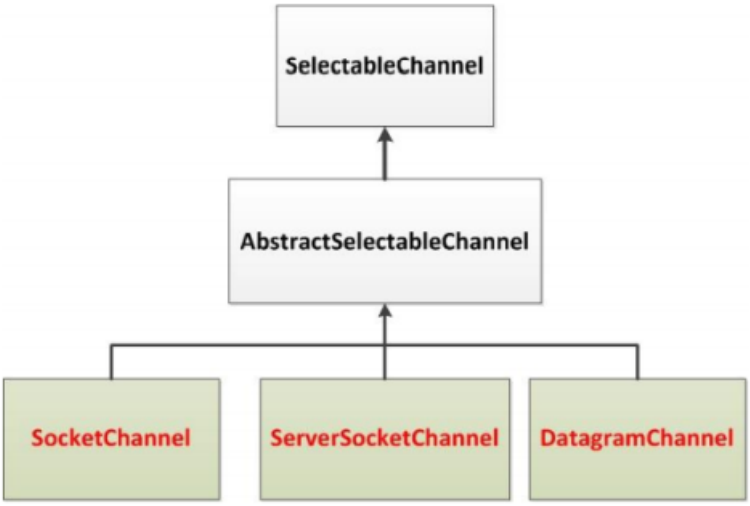
选择器提供选择执行已经就绪的任务的能力。从底层来看，Selector提供了询问通道是否已经准备好执行每个I/O操作的能力。Selector 允许单线程处理多个Channel。仅用单个线程来处理多个Channels的好处是，只需要更少的线程来处理通道。事实上，可以只用一个线程处理所有的通道，这样会大量的减少线程之间上下文切换的开销。

3.3 可选择通道(SelectableChannel)

注意：并不是所有的Channel，都是可以被Selector 复用的。比方说，FileChannel就不能被选择器复用。为什么呢？

判断一个Channel 能被Selector 复用，有一个前提：判断他是否继承了一个抽象类SelectableChannel。如果继承了SelectableChannel，则可以被复用，否则不能。

SelectableChannel 的结构如下图：

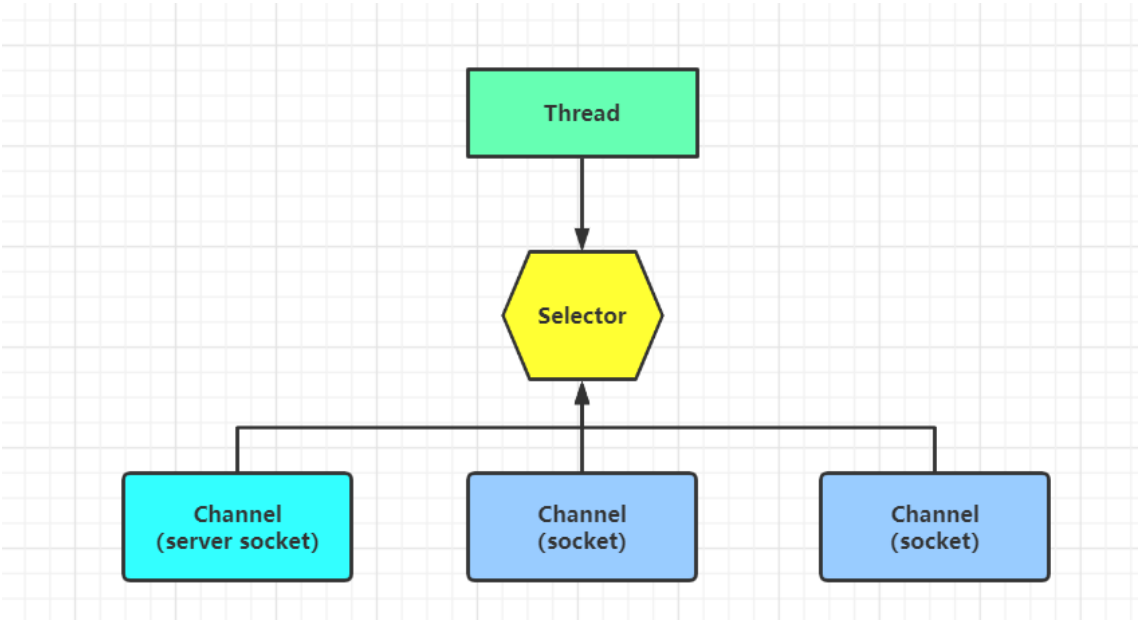


SelectableChannel类提供了实现通道的可选择性所需要的公共方法

通道和选择器注册之后，他们是绑定的关系吗？

答:不是。不是一对一的关系。一个通道可以被注册到多个选择器上，但对每个选择器而言只能被注册一次。

通道和选择器之间的关系，使用注册的方式完成。SelectableChannel可以被注册到Selector对象上，在注册的时候，需要指定通道的哪些操作，是Selector感兴趣的。



3.4 Channel注册到Selector

使用Channel.register（Selector sel，int ops）方法，将一个通道注册到一个选择器时。

第一个参数：指定通道要注册的选择器是谁

第二个参数：指定选择器需要查询的通道操作

可以供选择器查询的通道操作，从类型来分，包括以下四种：

（1）可读 : SelectionKey.OP_READ

（2）可写 : SelectionKey.OP_WRITE

（3）连接 : SelectionKey.OP_CONNECT

（4）接收 : SelectionKey.OP_ACCEPT

如果Selector对通道的多操作类型感兴趣，可以用“位或”操作符来实现：int key = SelectionKey.OP_READ | SelectionKey.OP_WRITE ;

3.5 选择键(SelectionKey)

Channel和Selector的关系确定好后，并且一旦通道处于某种就绪的状态，就可以被选择器查询到。这个工作，使用选择器Selector的select（）方法完成。select方法的作用，对感兴趣的通道操作，进行就绪状态的查询。

Selector可以不断的查询Channel中发生的操作的就绪状态。并且挑选感兴趣的操作就绪状态。一旦通道有操作的就绪状态达成，并且是Selector感兴趣的操作，就会被Selector选中，放入选择键集合中。

```
select()            :选择器等待客户端连接的方法
                    阻塞问题:
                        1.在开始没有客户访问的时候是阻塞的
                        2.在有客户来访问的时候方法会变成非阻塞的
                        3.如果客户的访问被处理结束之后,又会恢复成阻塞的

selectedKeys()      :选择器会把被连接的服务端对象放在Set集合中,这个方法就是返回一个Set集合
```

3.6 Selector的使用流程

3.6.1 创建Selector

Selector对象是通过调用静态工厂方法open()来实例化的，如下：

```
        // 1、获取Selector选择器
           Selector selector = Selector.open();
```

3.6.2 将Channel注册到Selector

要实现Selector管理Channel，需要将channel注册到相应的Selector上，如下：

```
            // 2、获取通道
            ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();

            // 3.设置为非阻塞
            serverSocketChannel.configureBlocking(false);

            // 4、绑定连接
            serverSocketChannel.bind(new
InetSocketAddress(SystemConfig.SOCKET_SERVER_PORT));

            // 5、将通道注册到选择器上,并制定监听事件为:“接收”事件
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

上面通过调用通道的register()方法会将它注册到一个选择器上。

首先需要注意的是：

与Selector一起使用时，Channel**必须处于非阻塞模式**下，否则将抛出异常IllegalBlockingModeException

3.6.3 轮询查询就绪操作

万事俱备，下一步是查询就绪的操作。

通过Selector的 select() 方法，可以查询出已经就绪的通道操作，这些就绪的状态集合，包存在一个元素是SelectionKey对象的Set集合中。

select()方法返回的int值，表示有多少通道已经就绪

而一旦调用select()方法，并且返回值不为0时，下一步工干啥？

通过调用Selector的selectedKeys()方法来访问已选择键集合，然后迭代集合的每一个选择键元素，根据就绪操作的类型，完成对应的操作：

3.6.4 NIO 编程实例

客户端

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

        //创建客户端
        SocketChannel sc = SocketChannel.open();
        //指定要连接的服务器ip和端口
        sc.connect(new InetSocketAddress("127.0.0.1",9000));

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

        //给数组添加数据
        buffer.put("拉勾教育".getBytes());

        //切换
        buffer.flip();

        //输出数据
```

```
        sc.write(buffer);

        //关闭资源
        sc.close();
    }
```

服务端

```
package com.lagou.selector;

import java.io.IOException;
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 Demo服务端 {
    public static void main(String[] args) throws IOException {
        //创建服务端对象
        ServerSocketChannel ssc1 = ServerSocketChannel.open();
        ssc1.bind(new InetSocketAddress(8000));
        //设置非阻塞
        ssc1.configureBlocking(false);

        //创建服务端对象
        ServerSocketChannel ssc2 = ServerSocketChannel.open();
        ssc2.bind(new InetSocketAddress(9000));
        ssc2.configureBlocking(false);

        //创建服务端对象
        ServerSocketChannel ssc3 = ServerSocketChannel.open();
        ssc3.bind(new InetSocketAddress(10001));
        ssc3.configureBlocking(false);

        //创建选择器对象
        Selector s = Selector.open();

        //两个服务器都要交给选择器来管理
        ssc1.register(s, SelectionKey.OP_ACCEPT);
        ssc2.register(s, SelectionKey.OP_ACCEPT);
        ssc3.register(s, SelectionKey.OP_ACCEPT);

        //获取集合
        //selectedKeys() :返回集合,集合作用存放的是被连接的服务对象的key
        Set<SelectionKey> set = s.selectedKeys();

        System.out.println("集合中元素的个数: " + set.size());  //0(没有服务端被访问的时候显示0)

        //select():这是选择器连接客户端的方法
        s.select();

        System.out.println("集合中元素的个数: " + set.size());  //1(有一个服务端被访问的时候显示1)
```

```
    }
}
```

```
package com.lagou.selector;

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;

public class Selector服务端 {

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

        // 1、获取Selector选择器
        Selector selector = Selector.open();

        // 2、获取通道
        ServerSocketChannel ssc1 = ServerSocketChannel.open();
        ServerSocketChannel ssc2 = ServerSocketChannel.open();
        ServerSocketChannel ssc3 = ServerSocketChannel.open();

        // 3.设置为非阻塞
        ssc1.configureBlocking(false);
        ssc2.configureBlocking(false);
        ssc3.configureBlocking(false);

        // 4、绑定连接
        ssc1.bind(new InetSocketAddress(8000));
        ssc2.bind(new InetSocketAddress(9000));
        ssc3.bind(new InetSocketAddress(10000));


        // 5、将通道注册到选择器上,并注册的操作为：“接收”操作
        ssc1.register(selector, SelectionKey.OP_ACCEPT);
        ssc2.register(selector, SelectionKey.OP_ACCEPT);
        ssc3.register(selector, SelectionKey.OP_ACCEPT);


        // 6、采用轮询的方式，查询获取"准备就绪"的注册过的操作
        while (selector.select() > 0) {
            // 7、获取当前选择器中所有注册的选择键（“已经准备就绪的操作”）
            Iterator<SelectionKey> selectedKeys =
selector.selectedKeys().iterator();
            while (selectedKeys.hasNext()) {

                // 8、获取"准备就绪"的事件
                SelectionKey selectedKey = selectedKeys.next();

                    // 9、获取ServerSocketChannel
```

```
                ServerSocketChannel serverSocketChannel = (ServerSocketChannel)
selectedKey.channel();
                // 10、接受客户端发来的数据
                SocketChannel socketChannel = serverSocketChannel.accept();

                // 11、读取数据
                    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                    int length = 0;
                    while ((length = socketChannel.read(byteBuffer)) != -1) {
                        byteBuffer.flip();
                        System.out.println(new String(byteBuffer.array(), 0,
length));

                        byteBuffer.clear();
                    }
                    socketChannel.close();
                }
                // 12、移除选择键
                selectedKeys.remove();
            }
        // 13、关闭连接
         ssc1.close();
         ssc2.close();
         ssc3.close();
        }

}
```