day24【Junit单元测试、NIO】

今日内容

- Junit单元测试
- NIO

教学目标

- [ ] 能够使用Junit进行单元测试
- [ ] 能够说出同步和异步的概念
- [ ] 能够说出阻塞和非阻塞的概念
- [ ] 能够创建和使用ByteBuffer
- [ ] 能够使用MappedByteBuffer实现高效读写
- [ ] 能够使用ServerSocketChannel和SocketChannel实现连接并收发信息

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、一个方法就是一个单元
    * 单元测试：程序员编写的一小段代码，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
    * 必须使用JUnit的注解@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，而且只执行一次。
* @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，而且只执行一次。
```

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执行之前执行一次。
 * @AfterAll：用来静态修饰方法，该方法会在所有测试方法执行之后执行一次。
```

- 示例代码(JUnit4.x)

```java
/**
 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 NIO

2.1 NIO概述

1.1.1 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数据，每一个操作在一步中产生或者消费一个数据，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的三个重要组成部分：Buffer（缓冲区）、Channel（通道）、Selector（选择器）

第三章 Buffer类（缓冲区）

3.1 Buffer概述

Buffer是一个对象，它对某种基本类型的数组进行了封装。NIO开始使用的Channel(通道)就是通过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

3.2 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

3.3 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：

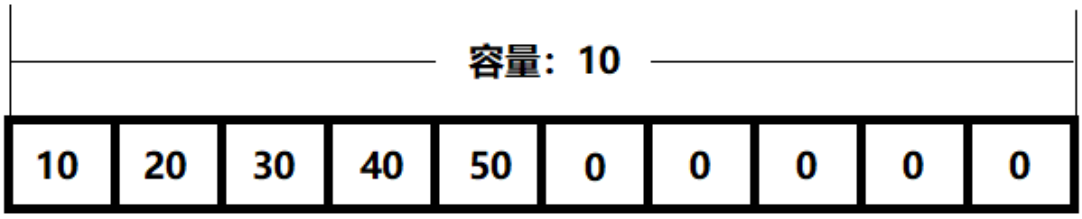
```
[10, 20, 0, 0, 0, 0, 0, 0, 0, 0]
```

- 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]
```

3.4 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
```

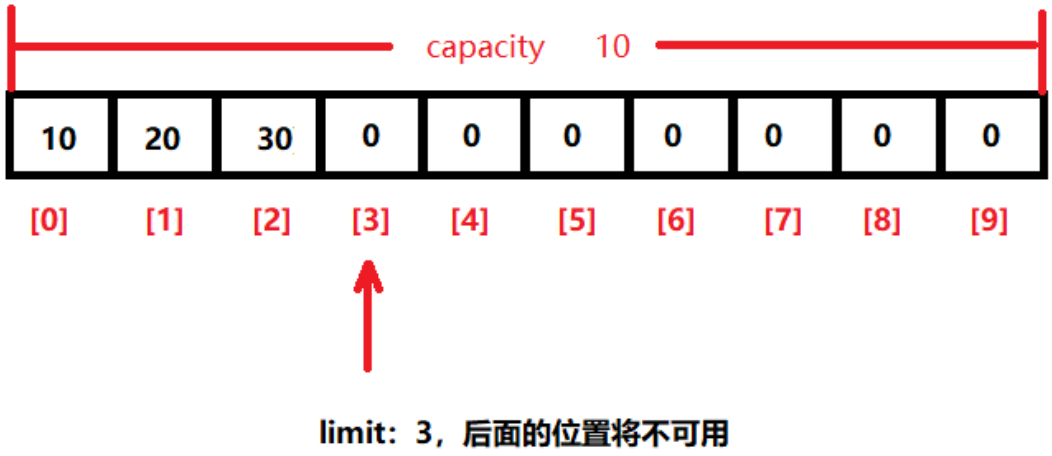
3.5 限制-limit

- 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：



3.6 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]
```

3.7 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]
```

3.8 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。

第四章 Channel（通道）

4.1 Channel概述

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

4.2 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这个示例让大家体会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();

    }
```

4.3 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2GB以下的文件**
  - 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！

- **示例：复制2GB以上文件**
  - 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();
        }
    }
}
```

4.4 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：

```
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！");
    }
}
```

4.5 ServerSocketChannel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！
```