一 概述

1 nio 的作用：传输数据

2 nio 的分类

1.a. BIO - Blocking IO - 同步阻塞式 IO

1.b. NIO - New IO - NonBlocking IO - 同步非阻塞式 IO - JDK1.4

1.c. AIO - Asynchronous IO - 异步非阻塞式 IO - JDK1.7 - AIO 是基于 NIO 进行了改进，也把 AIO 称之为 NIO.2；因为 NIO 出现时间比较早，所以市面上很多框架的底层都是基于 NIO 构建，对 NIO 进行了改进，导致 AIO 在市面上的占有率并不高

3 基本概念

1.d. 同步和异步：如果一个对象或者一段逻辑在一个时间段内允许被多个线程同时使用，此时称之为异步；如果一个对象或者一段逻辑在一个时间段内只允许被一个线程使用，此时称之为同步

1.e. 阻塞和非阻塞：一个线程在没有获取到结果之前会持续等待也不执行后续逻辑也不抛出异常，这种现象称之为阻塞；一个线程即使没有获取到结果也会继续往下执行或者抛出异常，此时称之为非阻塞。

4 NIO 的三大组件：Buffer、Channel、Selector

二 BIO 的缺点

1 阻塞

导致任务的执行效率变低

2 一对一连接

每当客户端产生一个连接，服务器端都需要产生一个线程去处理这个连接；如果产生了大量的客户端连接，服务器需要产生大量线程去处理这些连接；服务器端所能产生和承载的线程数量是有限的，如果线程过多会导致服务器的卡顿甚至崩溃

3 服务端线程大量占用

如果客户端连接之后不做任何操作而是恶意保持连接，导致服务器端的线程无法释放；如果产生大量的恶意连接，导致服务器端的线程被大量占用

三 Buffer-缓冲区

1 作用：存储数据

2 基于数组实现

Buffer 针对基本类型来进行存储（没有布尔类型的）：
ByteBuffer，ShortBuffer，IntBuffer，LongBuffer，FloatBuffer，DoubleBuffer，CharBuffer

3 重要位置：capacity >= limit >= position

3.A capacity：容量位。

用于表示缓冲区的容量，指定之后大小不能变

```
//输出当前 capacity 位置
System.out.println(buffer.capacity());
```

3.B limit：限制位。

用于限制 position 所能达到的最大下标。当 limit 和 position 重合的时候，就表示所有元素已经遍历完毕。当缓冲区刚创建的时候，limit 默认和 capacity 重合

```
//输出当前 limit 位置
System.out.println(buffer.limit());
//移动 limit 到 0
buffer.limit(0);
```

3.C position：操作位。

用于指向要读写的位置的。在缓冲区刚创建的时候，position 默认为 0。当对缓冲区进行读写操作的时候，position 自动后挪。

```
//输出当前 position 位置
System.out.println(buffer.position());
//移动 position 到 0
buffer.position(0);
```

4 使用

4.A 创建添加数据(方式一数据未知)

创建 capacity 大小为 10 的缓冲区（position 在 6），添加 abcdef 数据（0-5），此时 position 在 6。

```
// 表示给底层的字节数组来指定大小
// 缓冲区在给定之后，长度就不能改变了
ByteBuffer buffer =ByteBuffer.allocate(10);
System.out.println(buffer.position());//0
System.out.println(buffer.capacity());//10
System.out.println(buffer.limit());//10
// 添加数据
buffer.put("abc".getBytes());
buffer.put("def".getBytes());
System.out.println(buffer.position());//6
//position 为 6，获取的第 7 字节为数组未赋值时的默认值 0
byte b = buffer.get();
//移动 position 到 0
buffer.position(0);
byte b1 = buffer.get();
System.out.println(b1);//97
 byte b2 = buffer.get();
System.out.println(b2);//98
```

1.A 创建添加数据(方式二数据已知)

这种方式会覆盖？有待考证

```
// 数据已知 创建时 position 在 0
ByteBuffer buffer =ByteBuffer.wrap("abcdef".getBytes());
buffer.put("hello1".getBytes());
System.out.println(buffer.position());//6
System.out.println(buffer.limit());//6
System.out.println(buffer.capacity());//6
System.out.println(new String(buffer.array()));//hello1 这种方式会覆盖？
```

4.B 翻转缓冲区->遍历数据

```
// 遍历
/*
    手动遍历缓冲区
    1.将 limit 挪动到position 位置
    2.将position 位置归0
    3.循环读取[position,limit)的位置

  position（0）               limit（6）
     |                        |
    [ a , b , c , d , e , f , 0 , 0 , 0 , 0 ]
                                              |
                                         capacity（10）
*/

buffer.limit(buffer.position());
buffer.position(0);
/* 上述两部操作称之为翻转缓冲区
等价于下面：*/
 buffer.flip();
//遍历
while(buffer.position() < buffer.limit()){
    byte b = buffer.get();
    System.out.println(b);
}

//#######上诉遍历方式有已被封装的方法
while(buffer.hasRemaining()){
    byte b = buffer.get();
    System.out.println(b);
}
```

四 Channel-通道

1. 作用：**传**输数据
1. Channel 在传输的时候是针对缓冲区进行操作
2. 常用的 Channel
    1.a. 文件：FileChannel
    1.b. UDP：DatagramChannel
    1.c. **TCP**：SocketChannel，ServerSocketChannel
2. Channel 默认是阻塞的，手动设置为非阻塞
3. Channel 可以实现双向传输

1 测试

服务端监听 8090 端口，循环接收客户端请求

客户端连接 8090 端口，发送消息，程序结束

服务端打印消息，程序结束

1.A 服务端

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

    // 开启服务器端通道
    ServerSocketChannel ssc = ServerSocketChannel.open();
    // 绑定监听的端口
    ssc.bind(new InetSocketAddress(8090));
    // 设置为非阻塞
    ssc.configureBlocking(false);
    // 接收连接
    SocketChannel sc = ssc.accept();
    // 因为是非阻塞的，所以连接无论是否接收，都会继续向下执行
    // 判断是否接收到连接
    // 服务器不计数
    while (sc == null){
        sc = ssc.accept();
    }
    // 读取数据
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    sc.read(buffer);
    System.out.println(new String(
            buffer.array(), 0, buffer.position()));
    // 关流
    ssc.close();
}
```

1.B 客户端

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

    // 开启客户端的通道
    SocketChannel sc = SocketChannel.open();
    // 设置为非阻塞
    sc.configureBlocking(false);
    // 发起连接
    // 因为是非阻塞的，所以无论是否建立连接，都会继续往下执行
    sc.connect(new
            InetSocketAddress("localhost", 8090));
    // 判断连接是否建立
    // 如果连接多次都没有成功，那么说明连接无法建立
    while(!sc.isConnected()) {
        // 如果没有建立连接，试图再次建立
        // finishConnect 方法底层自动计数
```

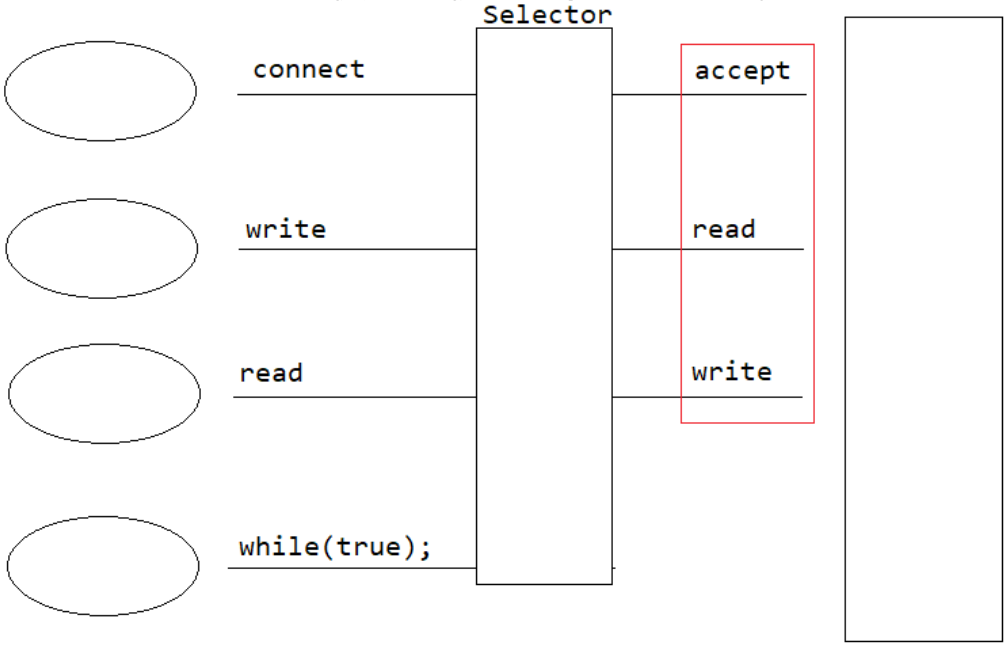
```
            // 如果计数多次依然没有建立连接，抛出异常
            sc.finishConnect();
        }
        // 连接建立
        // 写出数据
        sc.write(ByteBuffer.wrap("hello server".getBytes()));
        // 关流
        sc.close();

    }
```

回顾 TCP/UDP 异同

五　Selector – 多路复用选择器

1. 作用：针对通道的指定事件来进行**选**择
2. Selector 在使用的时候针对**非阻塞**通道进行操作



1　测试

服务端一直监听 8090 端口，等待客户端连接
客户端连接发送“hello server”，服务端打印，并回复“收到消息了”
客户端打印收到的消息，程序结束
服务端继续监听

1.A　服务端

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

```
    // 开启服务器端通道
    ServerSocketChannel ssc = ServerSocketChannel.open();
    // 设置非阻塞
    ssc.configureBlocking(false);
    // 绑定端口
    ssc.bind(new InetSocketAddress(8090));
    // 开启选择器
    Selector selc = Selector.open();
    // 将通道注册到选择器上
    ssc.register(selc, SelectionKey.OP_ACCEPT);
    // 模拟：服务器开启之后不关闭
    while (true) {
        // 随着运行时间的延长，接收到的请求会越来越多
        // 需要针对这些请求进行选择，将能触发事件的请求留下
        // 将不能触发事件的请求过滤掉
        selc.select();
        // 选完之后，留下来的请求都是有用的请求
        // connect/read/write -> accept/write/read
        // 因为这些请求中不一定存在所有的事件
        // 所以需要获取请求的事件类型
        Set<SelectionKey> set = selc.selectedKeys();
        // 需要针对请求的不同类型来进行分门别类的处理
        Iterator<SelectionKey> it = set.iterator();
        while (it.hasNext()) {
            SelectionKey key = it.next();
            // 触发服务器的 accept 操作
            // -> 说明客户端一定调用了 connect 方法
            if (key.isAcceptable()) {
                // 从事件中获取通道
                ServerSocketChannel sscx =
                        (ServerSocketChannel) key.channel();
                // 接收连接
                SocketChannel sc = ssc.accept();
                sc.configureBlocking(false);
                // 根据需求确定，如果需要读操作，那么就给 READ
                // 如果需要写操作，那么就给 WRITE
                // 如果存在多个 register，那么后边的会覆盖前边的
                sc.register(selc,
                        //SelectionKey.OP_READ +
SelectionKey.OP_WRITE);
                        // SelectionKey.OP_READ |
SelectionKey.OP_WRITE);
                        SelectionKey.OP_READ ^ SelectionKey.OP_WRITE);
            }
            if (key.isReadable()) {
                SocketChannel sc = (SocketChannel) key.channel();
                // 读取数据
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                sc.read(buffer);
                System.out.println(new String(buffer.array(), 0,
                        buffer.position()));
                // 读取完成之后，需要将 READ 事件从通道身上移除掉
                // key.interestOps() - 获取到所有事件
                sc.register(selc,
                        // key.interestOps() - SelectionKey.OP_READ);
```

```
                        key.interestOps() ^ SelectionKey.OP_READ);
            }
            if (key.isWritable()) {
                SocketChannel sc = (SocketChannel) key.channel();
                sc.write(ByteBuffer.wrap("收到数据啦~~~".getBytes()));
                sc.register(selc,
                        key.interestOps() - SelectionKey.OP_WRITE);
            }
            // 处理完成之后，需要将这一大类事件移除掉
            it.remove();
        }
    }

}
```

1.B 客户端

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

    SocketChannel sc = SocketChannel.open();
    sc.connect(
            new InetSocketAddress("localhost", 8090));
    sc.write(ByteBuffer.wrap("hello server".getBytes()));
    // 读取数据
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    sc.read(buffer);
    System.out.println(
            new String(buffer.array(), 0, buffer.position()));
    sc.close();
}
```