从NIO到AIO

理论知识

IO模型

磁盘IO

基于磁盘操作的IO接口:File

将数据持久化到磁盘,数据在磁盘上最小的描述就是文件，上层应用对磁盘的读和写都是针对文件而言的

网络IO

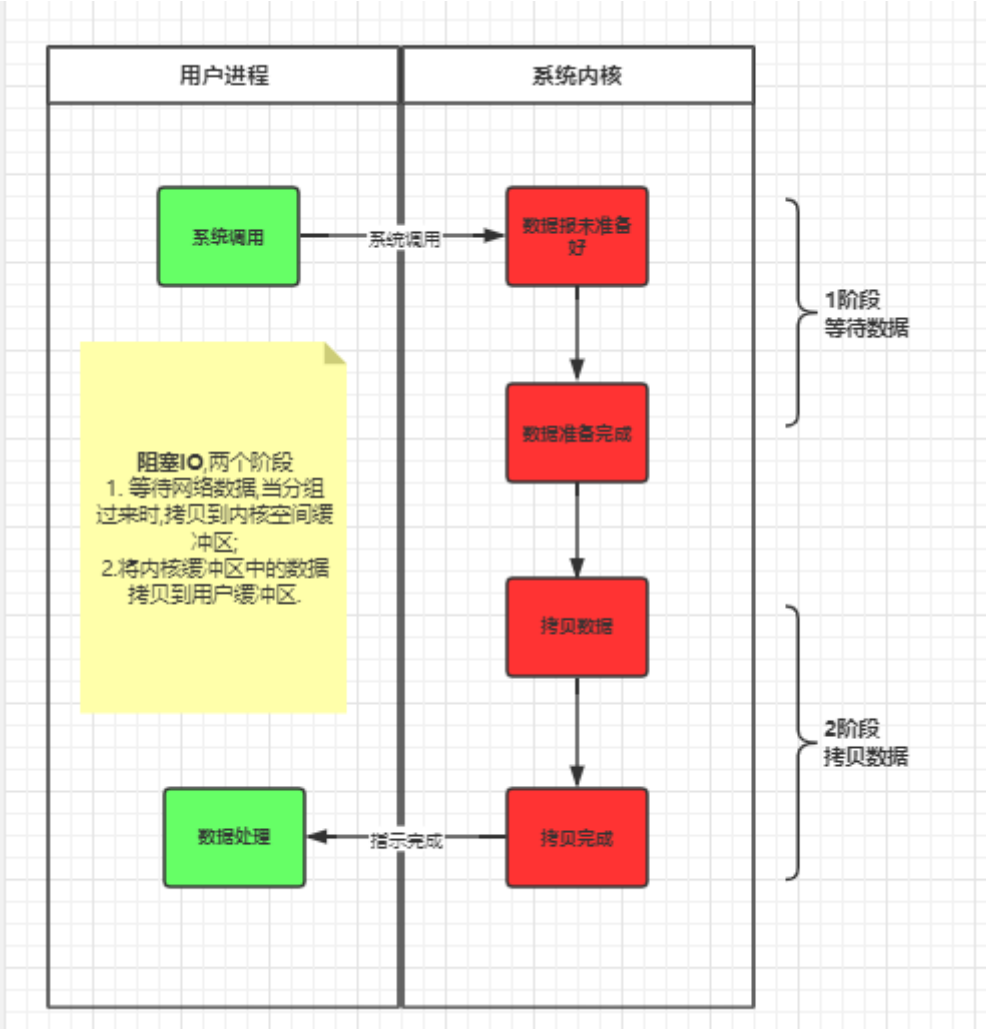
基于网络操作的io接口：Socket

Socket位于传输层和应用层之间，向应用层统一提供编程接口，应用层不必知道传输层的协议细节。Java中对Socket的支持主要是以下两种：

(1)基于TCP的Socket:提供给应用层可靠的流式数据服务，使用TCP的Socket应用程序协议：HTTP，FTP，TELNET等。优点：基于数据传输的可靠性。

(2)基于UDP的Socket：适用于数据传输可靠性要求不高的场合。

阻塞IO

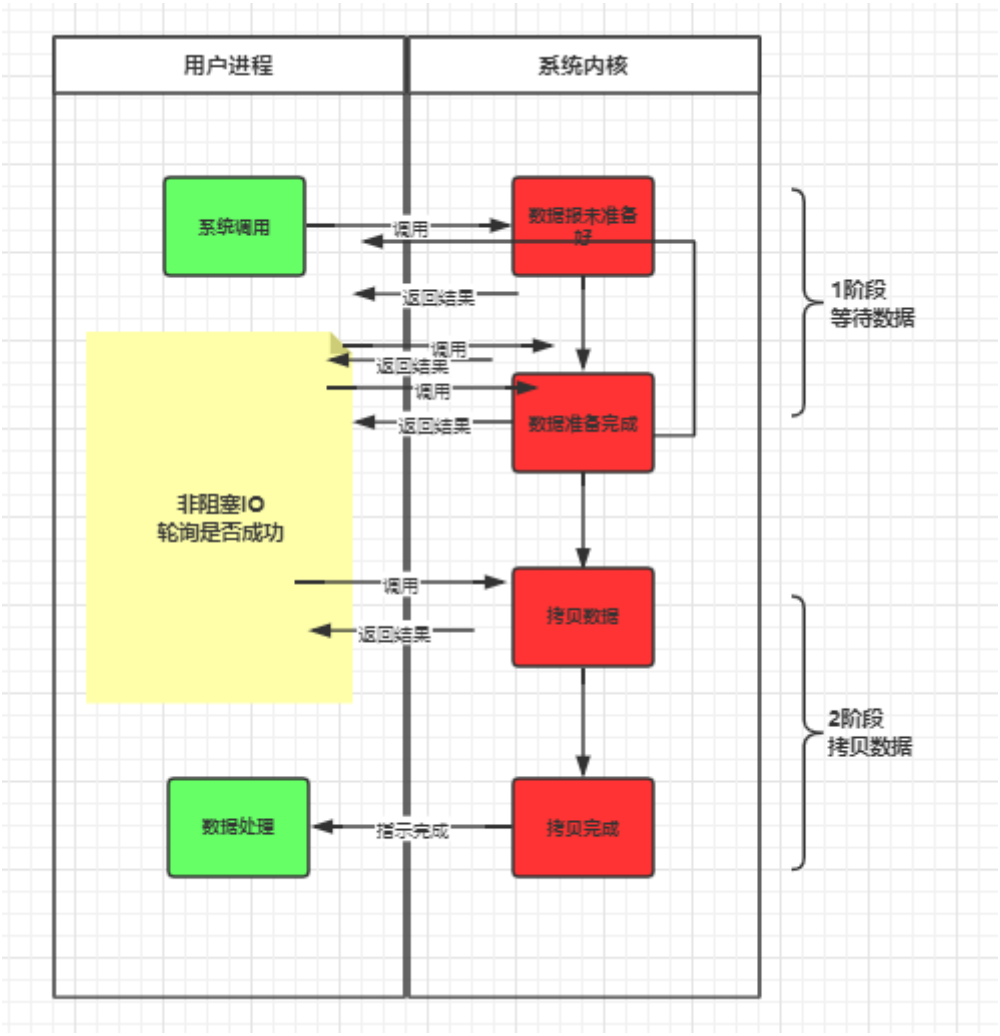


阻不阻塞只讨论第一阶段的,第二阶段该怎样怎样都是阻塞的.

阻塞IO: 比如上班需要等公交, 公交车没来 ,你就只能在那里等着,不许做其他事.

阻塞IO表示调用者调用了IO操作,对程序的控制权就不在自己手中,登结束调用操作,才回到用户手中.这段时间内被阻塞了.做不了其他事情

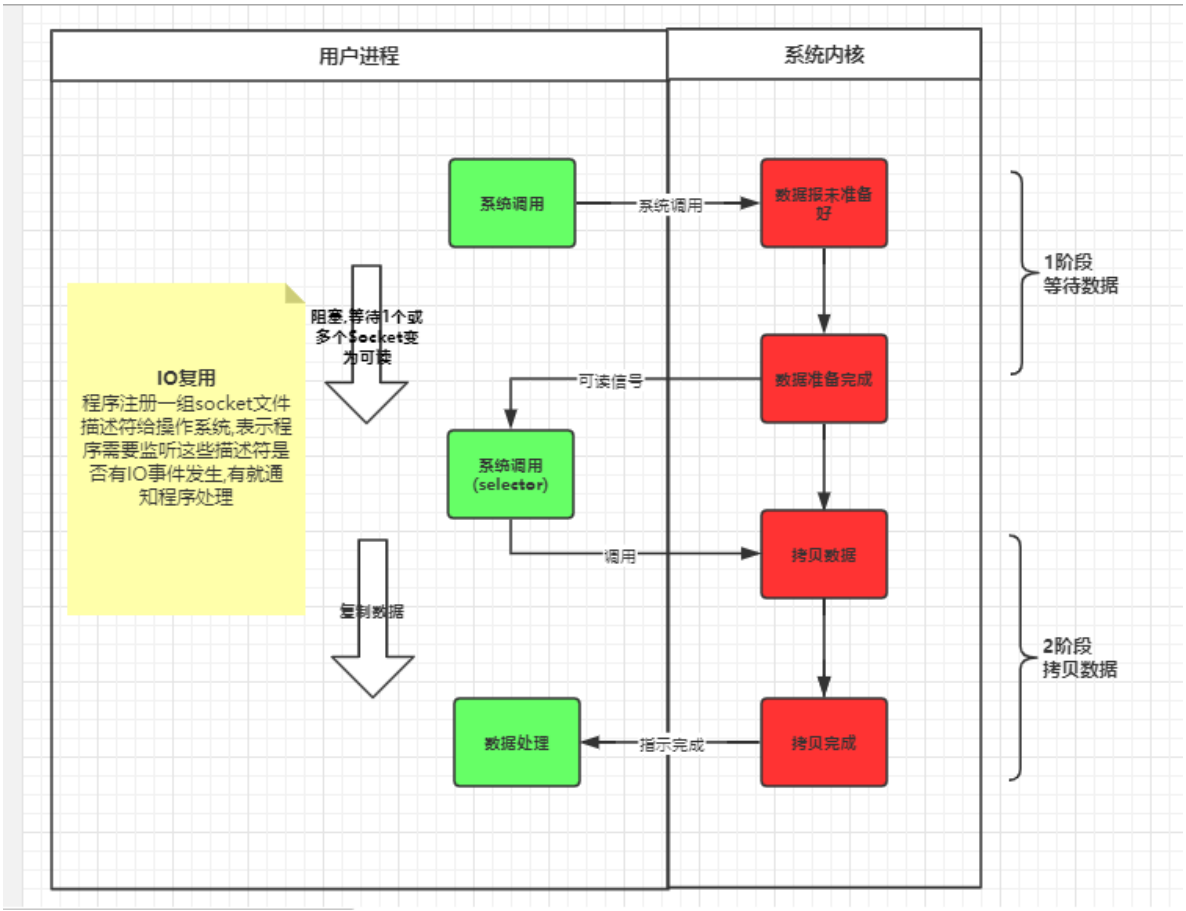
非阻塞IO



非阻塞: 等公交车的时候,你可以玩手机,不时看看车来没用.

非阻塞IO 立即将控制权返还给调用者,调用的人不需要等待,它获取两种结果 一种数据处理未完成 等下再调用,另一种数据处理完成返回 成功

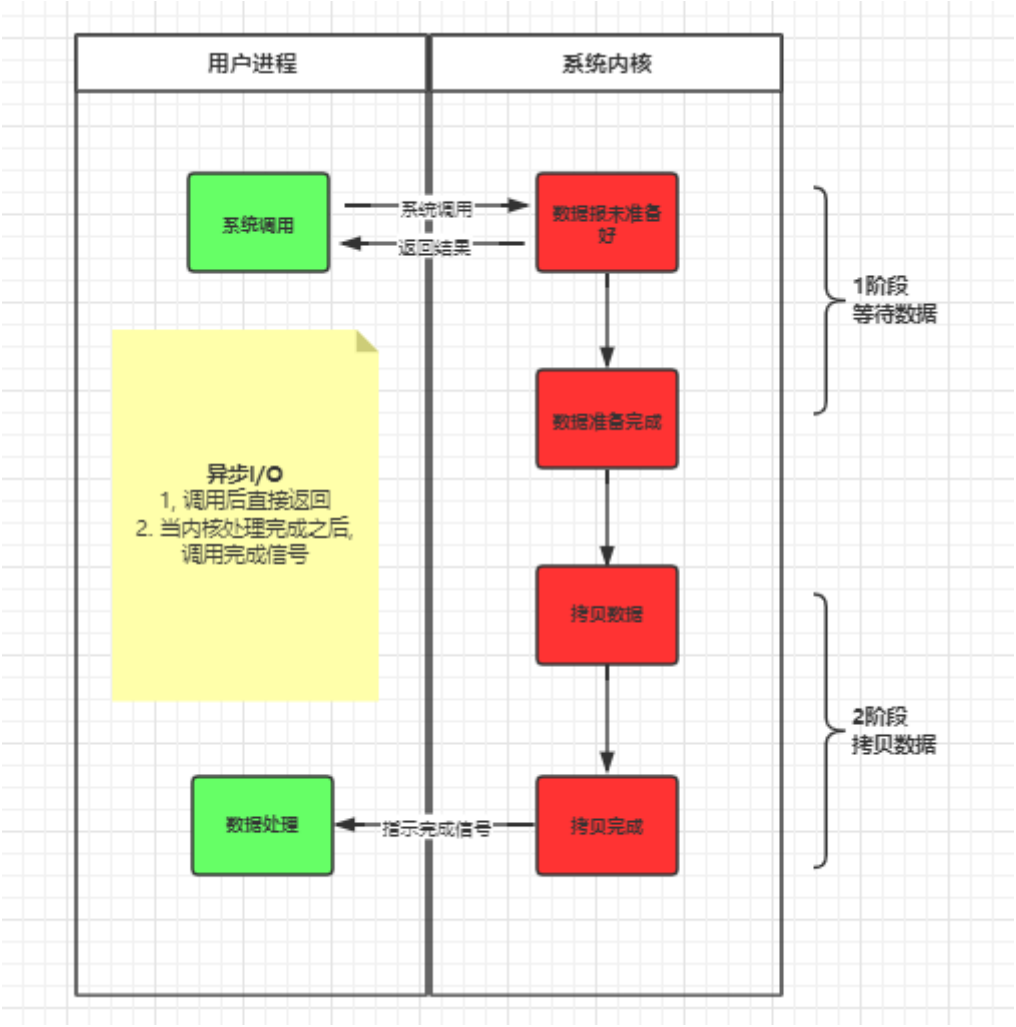
IO复用



IO复用: emmm.....不好举例,就是你可以做多辆车 (5 路, 7 路),来了就通知你

IO复用 调用者在selctor上注册自己敢兴趣的事件,你去干别的事,当事件发生 ,程序就会通知你去做

异步IO



异步IO: 你想起你还没吃饭 ,就到饭店吃饭,叫路人帮你看车,车来了叫你.

异步IO 调用函数返回时,需要告诉调用者,请求已经开始,此时系统会启用其他线程来完成本次调用操作,并在完成的时候回调调用者.

什么是NIO

NIO即 `NoneBlocking IO` 非阻塞IO,原理上已经知道什么是阻塞了,那具体实现上什么是阻塞呢?

- 系统调用 read 从socke里读取数据
- 系统调用从磁盘文件读取数据到内存

上面两种场景那种是阻塞呢?可能你会觉得两种都是.但是你的理解和Linux的理解不同 .linux 认为 :

- 对于第一种情况，算作block，因为Linux无法知道网络上对方是否会发数据。如果没数据发过来，对于调用read的程序来说，就只能“等”。
- 对于第二种情况，不算做block。

  所以对于磁盘IO , 都不视为阻塞

  > 一个解释是，所谓“Block”是指操作系统可以预见这个Block会发生才会主动Block。例如当读取TCP连接的数据时，如果发现Socket buffer里没有数据就可以确定定对方还没有发过来，于是Block；而对于普通磁盘文件的读写，也许磁盘运作期间会抖动，会短暂暂停，但是操作系统无法预见这种情况，只能视作不会Block，照样执行。

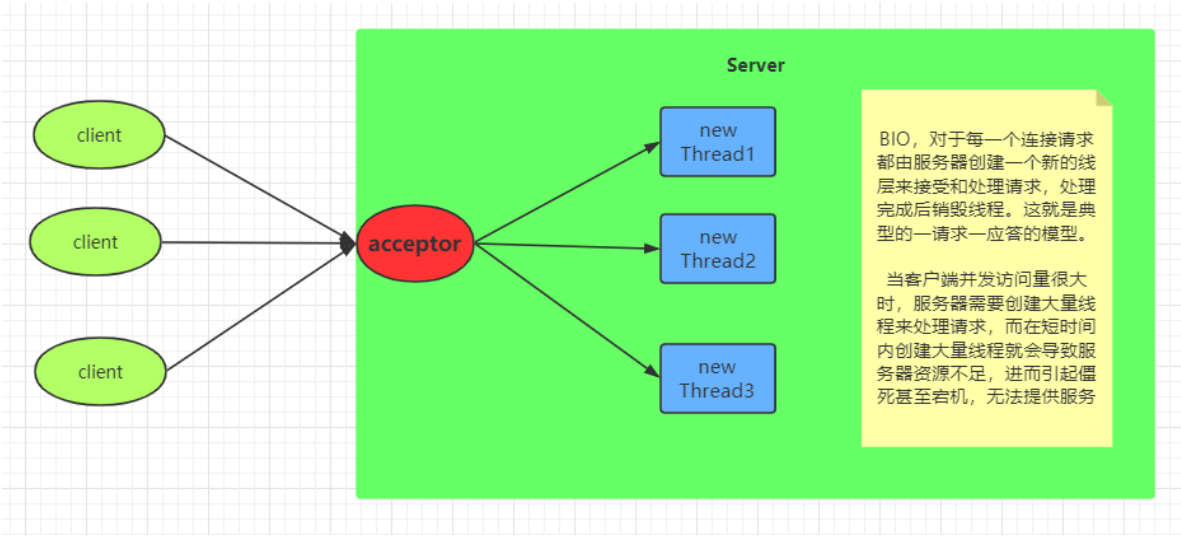
NIO和IO多路复用才有意义.

java中的NIO是JDK1.4引入的新的IO库(java.nio),NIO弥补了原来同步阻塞I/O的不足, **它在标准Java代码中提供了高速的、面向块的I/O. 通过定义包含数据的类, 以及通过以块的形式处理这些数据,**

为什么要用NIO

没有NIO之前,使用的是BIO,在网络操作中read和connect等操作是,系统调用会被卡住.(不会影响其他程序运行,操作系统多任务)

传统BIO编程是基于Client/Server模型, 其中服务端提供位置信息(绑定的IP地址和监听端口), 客户端通过连接操作向服务端监听的地址发起连接请过三次握手建立连接, 如果连接建立成功,双发就可以通过网络套接字(Socket)进行通信



BIO案例

Server

```
/**
 * 服务端
 *
 * @author miao.chen01@hand-china.com 2019-12-5
 */
@Slf4j
public class MyServer {
    public static void main(String[] args) {
        new MyServer().start();
    }

    void start() {
        try {
            ServerSocket serverSocket = new ServerSocket(12345);
            /**
             * 手动创建线程池 不要使用Executors
             * 参数
             * - corePoolSize:核心线程数
             * - maximumPoolSize: 最大线程数
             * - keepAliveTime: 非核心线程空闲存活时间
             * - TimeUnit 时间单位
             * - BlockingQueue 队列 (需要声明一个有界队列)
             * - ThreadFactory 线程工厂 (推荐) 使用guava的ThreadFactoryBuilder
             *
             */
```

```
            ThreadPoolExecutor pool = new ThreadPoolExecutor(10,
                    20,
                    60,
                    TimeUnit.SECONDS,
                    new LinkedBlockingQueue<>(1000),
                    new ThreadFactoryBuilder().build());
            while (true) {
                Socket client = serverSocket.accept();
                log.info("客户端连接到服务器.....");
                pool.execute(new ClientHandler(client));
            }
        } catch (IOException e) {
            log.error("服务器异常 ", e);
        }
    }
}

/**
 * 处理客户端信息
 */
@AllArgsConstructor
@Slf4j
class ClientHandler implements Runnable {
    private Socket client;

    public void sayHello() {
        try (
                BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
                BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(client.getOutputStream()));
        ) {
            //读客户端发送的数据
            while (true) {
                String line = bufferedReader.readLine();
                if(line !=null){
                    log.info("客户端说: {}", line);
                    break;
                }
            }

            client.shutdownInput();
            //向客户端发送数据
            bufferedWriter.write("hello 客户端!");
            // 必须要刷,不然会阻塞
            bufferedWriter.flush();
            client.shutdownOutput();
        } catch (IOException e) {
            log.info("服务器处理客户端错误: ", e);
        }
    }

    @Override
    public void run() {
        sayHello();

    }
}
```

client

```
/**
 * 客户端
 *
 * @author miao.chen01@hand-china.com 2019-12-5
 */
@Slf4j
public class MyClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("127.0.0.1", 12345);
        try (
                BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));
                BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        ) {
            bufferedWriter.write("hello! 服务器");
            // 必须要刷,不然会阻塞
            bufferedWriter.flush();
            socket.shutdownOutput();

            while (true) {
                String line = bufferedReader.readLine();
                if (line != null) {
                    log.info("客户端接收服务器信息: {}", line);
                    break;
                }
            }

        }
    }
}
```

结果

server

```
Connected to the target VM, address: '127.0.0.1:52908', transport: 'socket'
14:46:22.093 [main] INFO com.example.demoothers1.demo4.day1.MyServer - 客户端连接到服务
器.....
14:46:22.107 [pool-1-thread-1] INFO com.example.demoothers1.demo4.day1.ClientHandler -
客户端说: hello! 服务器
14:47:54.069 [main] INFO com.example.demoothers1.demo4.day1.MyServer - 客户端连接到服务
器.....
14:47:54.070 [pool-1-thread-2] INFO com.example.demoothers1.demo4.day1.ClientHandler -
客户端说: hello! 服务器
14:47:56.409 [main] INFO com.example.demoothers1.demo4.day1.MyServer - 客户端连接到服务
器.....
14:47:56.410 [pool-1-thread-3] INFO com.example.demoothers1.demo4.day1.ClientHandler -
客户端说: hello! 服务器
14:47:58.661 [main] INFO com.example.demoothers1.demo4.day1.MyServer - 客户端连接到服务
器.....
14:47:58.663 [pool-1-thread-4] INFO com.example.demoothers1.demo4.day1.ClientHandler -
客户端说: hello! 服务器
```

client

```
Connected to the target VM, address: '127.0.0.1:52955', transport: 'socket'
14:47:58.669 [main] INFO com.example.demoothers1.demo4.day1.MyClient - 客户端接收服务器信息: hello 客户端!
Disconnected from the target VM, address: '127.0.0.1:52955', transport: 'socket'
```

上面的代码 是原始bio的变种 , 通过线程池实现的伪异步通信框架能够缓解BIO面临的问题,但是无法从根本上解决问题,当并发量增加或者网络IO时延增大之后, 线程的执行时间会被拉长, 它导致缓存在任务队列中的任务不断堆积, 最终导致内存溢出或者拒绝新任务的执行.

NIO核心类

之前说了NIO的主要场景在网络IO中使用,但是磁盘IO中依然可以使用NIO进行处理.

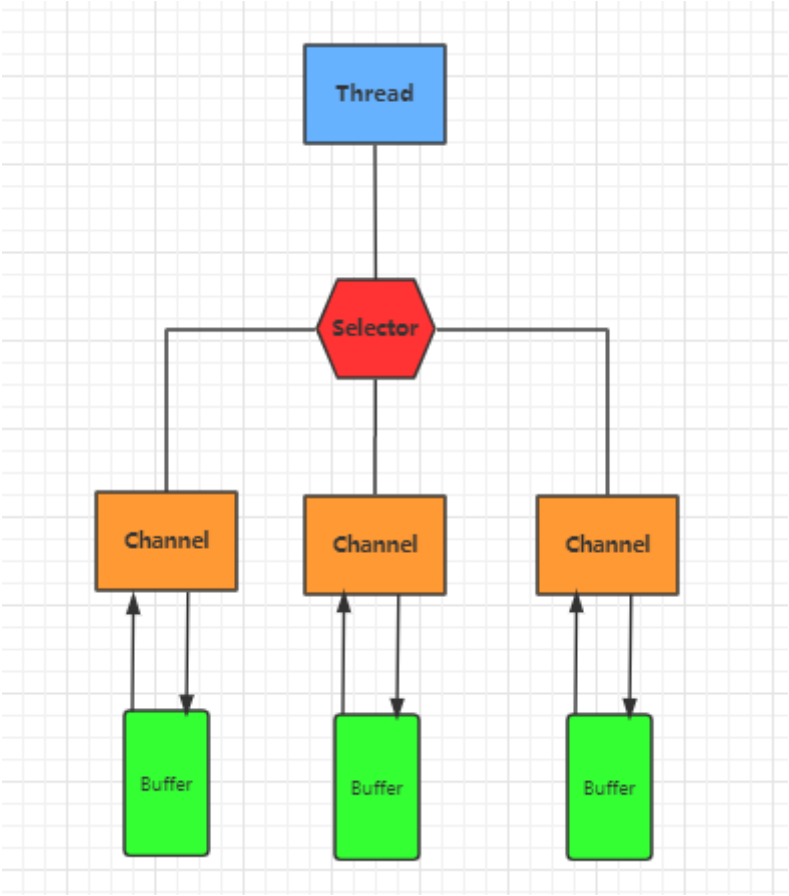
Java NIO 由以下几个核心部分组成:

- Channel
- Buffer
- Selector

Channel 就像火车,管理运输.运输什么?当然是数据了

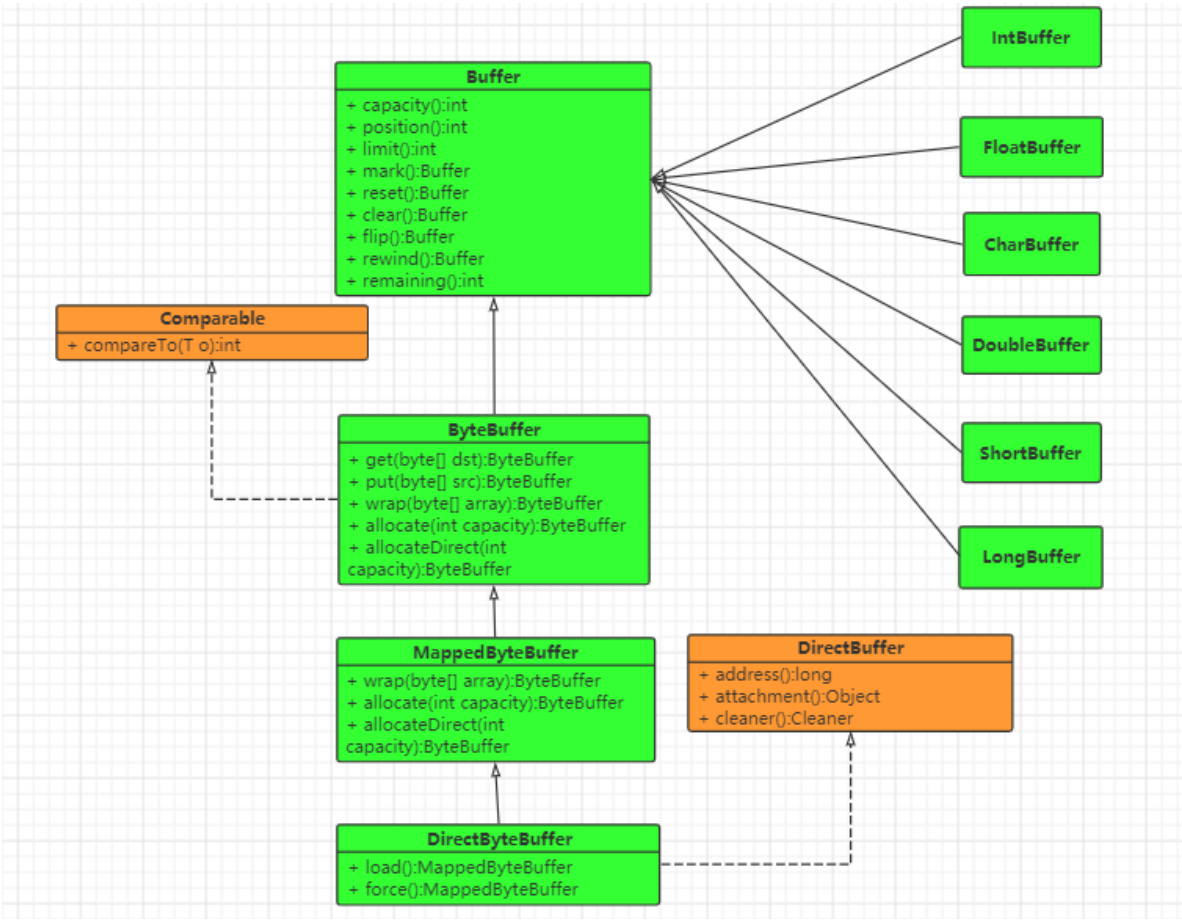
Buffer缓冲区,将数据冲Channel中拿出来,NIO之所以面向块,就是它是一个Buffer一个Buffer读写的,而buffer就是快内存.

Selector 选择器,选择那辆火车装货(卸货) 关系如下



NIO核心的类图(**类中只显示常用的方法,黄色表示接口,绿色表示类**)

Buffer(缓冲器)



Buffer 用于与Channel通道交互,数据通过Buffer读写入通道中.Buffer实现包括7种基本数据类型实现(排除Boolean),上图中我只画了ByteBuffer的详细信息,没种的方法差不多,并且每种Buffer都有与之对应的DirectByteBuffer

Buffer属性

属性	说明
capacity	容量。即可以容纳的最大数据量, 在缓冲区创建时被设定不能改变(1024倍数)
limit	界限，即表明还有多少数据需要取出，或者还有多少空间能够写入
position	位置。下一个要被读或写的元素的索引, 每次读写缓冲区时都会改变此值, 为下次读写作准备
mark	标记。调用mark()来设置mark=position, 调用reset()可以让position恢复到标记的位置

ByteBuffer方法

方法	描述
allocate(int capacity)	分配一个容量大小为capacity的byte数组作为缓冲区的byte数据存储器
allocateDirect(int capacity)	分配直接缓冲区 不使用JVM堆栈
wrap(byte[] array)	bytes数组或buff缓冲区任何一方中数据的改动都会影响另一方。
reset()	把position设置成mark的值，相当于之前做过一个标记，现在要退回到之前标记的地方
clear()	position = 0;limit = capacity;mark = -1; 有点初始化的味道，但是并不影响底层byte数组的内容. 切换到写模式
flip()	limit = position;position = 0;mark = -1; 切换到读模式
rewind()	把position设为0，mark设为-1，不改变limit的值 重新读
compact()	position = limit - position，limit=capacity(当position=limit，等同于clear) 清空已读取的部分
get()	相对读，从position位置读取一个byte，并将position+1，为下次读写作准备
put(byte b)	相对写，向position的位置写入一个byte，并将postion+1，为下次读写作准备
remaining()	返回limit-position的值 , 可读的长度

测试

```
/**
 * @author miao.chen01@hand-china.com 2019-12-6
 */
@Slf4j
public class TestBuffer {
    static void bufferInfo(Buffer buffer) {
        log.info("Buffer信息如下:");
        log.info("capacity : {}", buffer.capacity());
        log.info("limit : {}", buffer.limit());
        log.info("position : {}", buffer.position());
        log.info("remaining : {}",buffer.remaining());
        System.out.println();
    }

    public static void main(String[] args) {
        // 1.初始化Buffer,打印信息
        ByteBuffer byteBuffer = ByteBuffer.allocate(10240);
        bufferInfo(byteBuffer);
        // 2.放数据
        byteBuffer.put("测试".getBytes());
        bufferInfo(byteBuffer);
        // 3.读数据
        byteBuffer.flip();
        byteBuffer.get();
        bufferInfo(byteBuffer);
        // 4.重读
        byteBuffer.rewind();
```

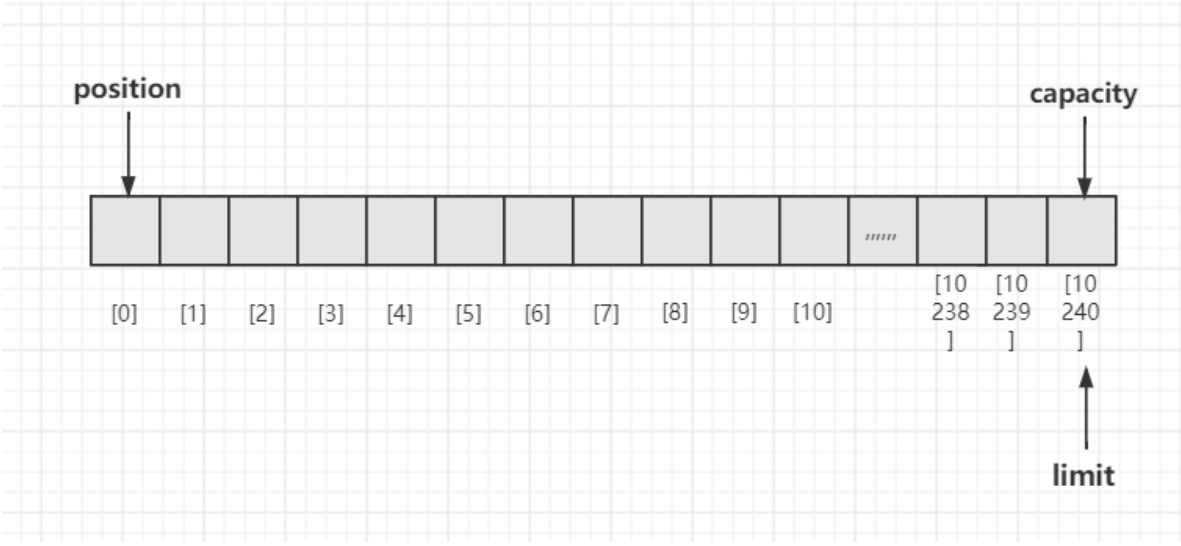
```
        bufferInfo(byteBuffer);
        // 5.读取两个字节并且设置mark
        byteBuffer.mark();
        byteBuffer.get(new byte[2]);
        bufferInfo(byteBuffer);
        // 6.回到mark
        byteBuffer.reset();
        bufferInfo(byteBuffer);
        // 7.clear
        byteBuffer.clear();
        bufferInfo(byteBuffer);
    }
}
```

结果分析

1. 初始信息

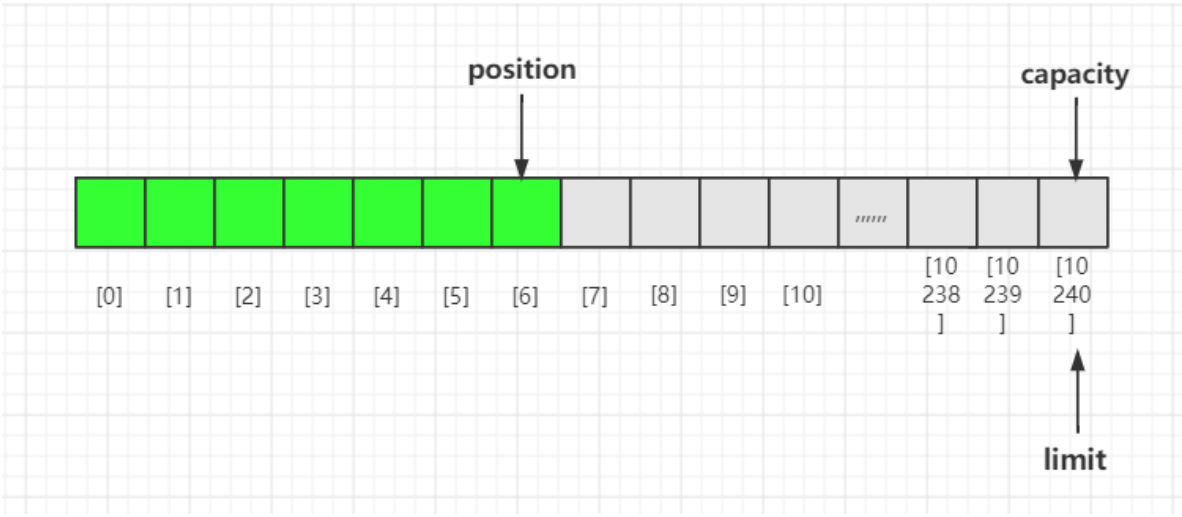
```
20:12:28.979 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:12:28.983 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 10240
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 0
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 10240
```

Buffer内部存储示意图



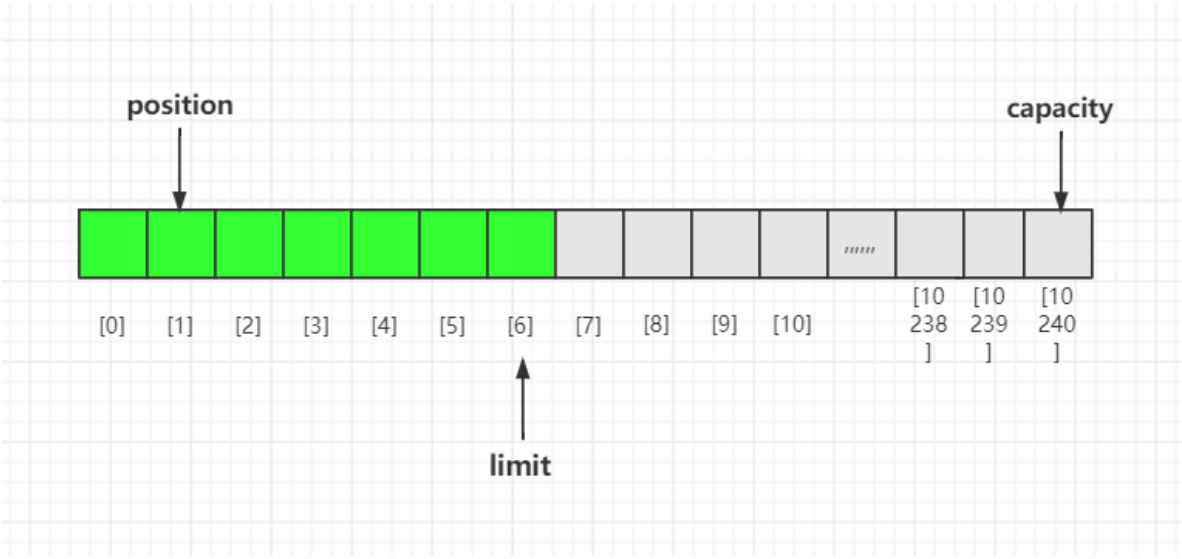
2. 放数据

```
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 10240
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 6
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 10234
```



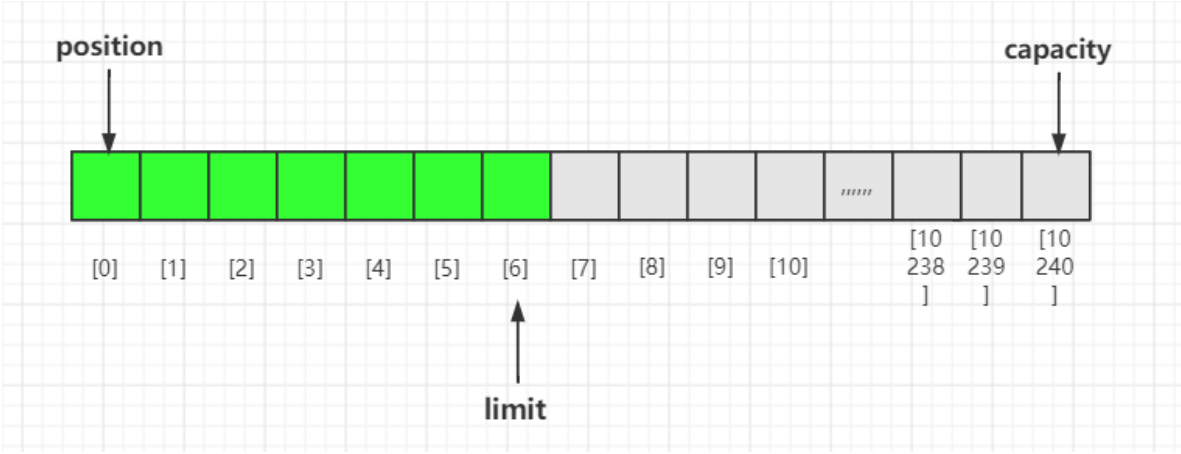
3. 读数据

```
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 6
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 1
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 5
```



4. 重读

```
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:12:28.986 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:12:28.987 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 6
20:12:28.987 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 0
20:12:28.987 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 6
```



5. 读取两个字节并且设置mark

```
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 6
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 2
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 4
```

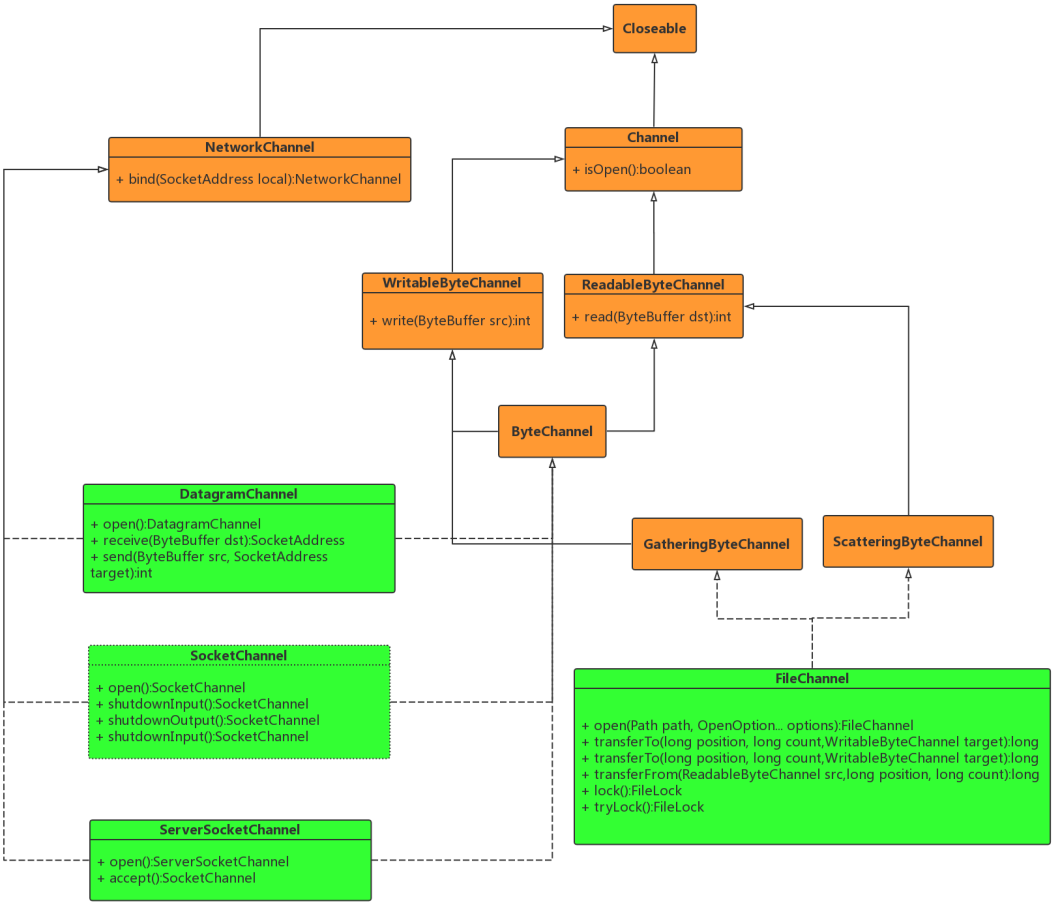
6. 回到mark

```
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 6
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 0
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 6
```

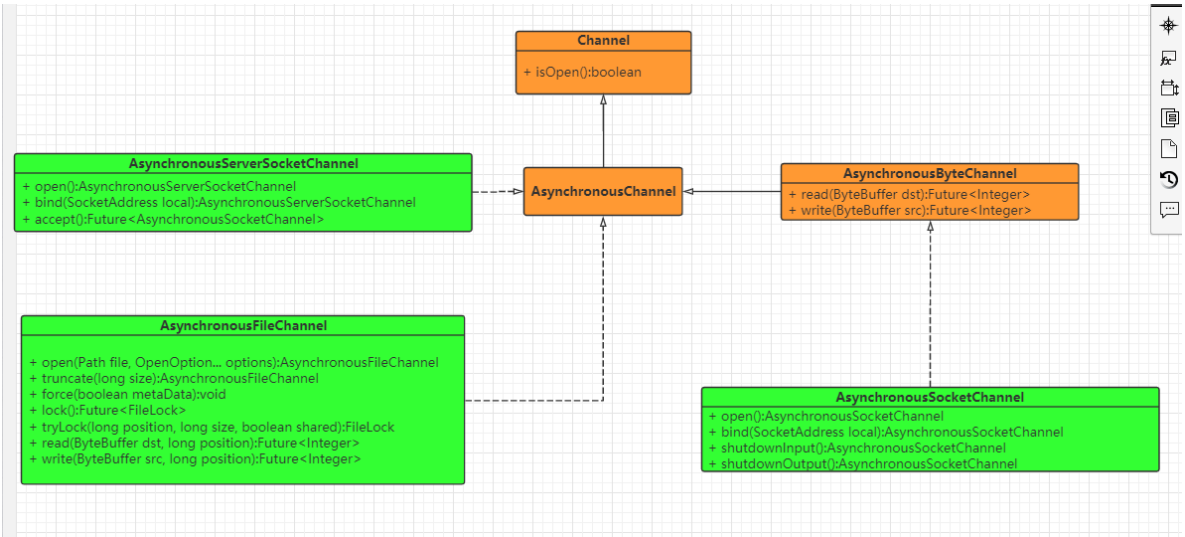
7. Clear

```
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - Buffer信息如下:
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - capacity : 10240
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - limit : 10240
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - position : 0
20:35:52.502 [main] INFO com.example.demoothers1.demo4.day1.TestBuffer - remaining : 10240
```

Channel(通道)



java NIO



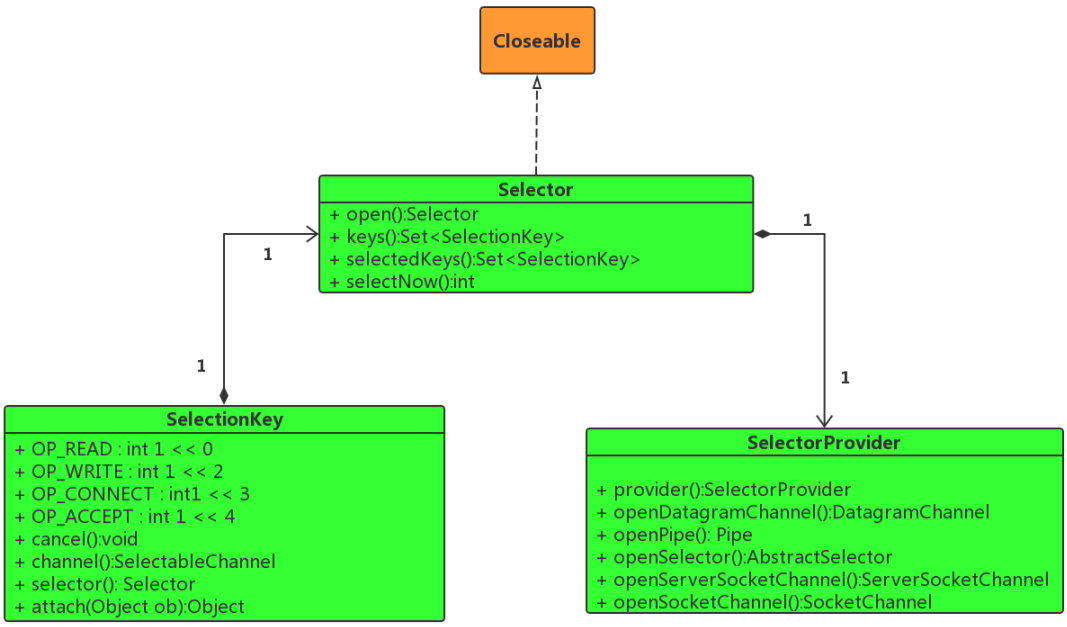
NIO2(AIO)类图.java1.7引入

Channel 可以全双工读写数据,而流只能单向操作.主要的Channel即上面NIO和NIO2

类名	描述
FileChannel	文件读写数据通道
SocketChannel	TCP读写网络数据通道
ServerSocketChannel	服务端网络数据读写通道，可以监听TCP连接
DatagramChannel	UDP读写网络数据通道

类名	描述
AsynchronousServerSocketChannel	异步SocketChannel
AsynchronousFileChannel	异步FileChannel
AsynchronousSocketChannel	异步ServerSocketChannel

Selector(选择器)



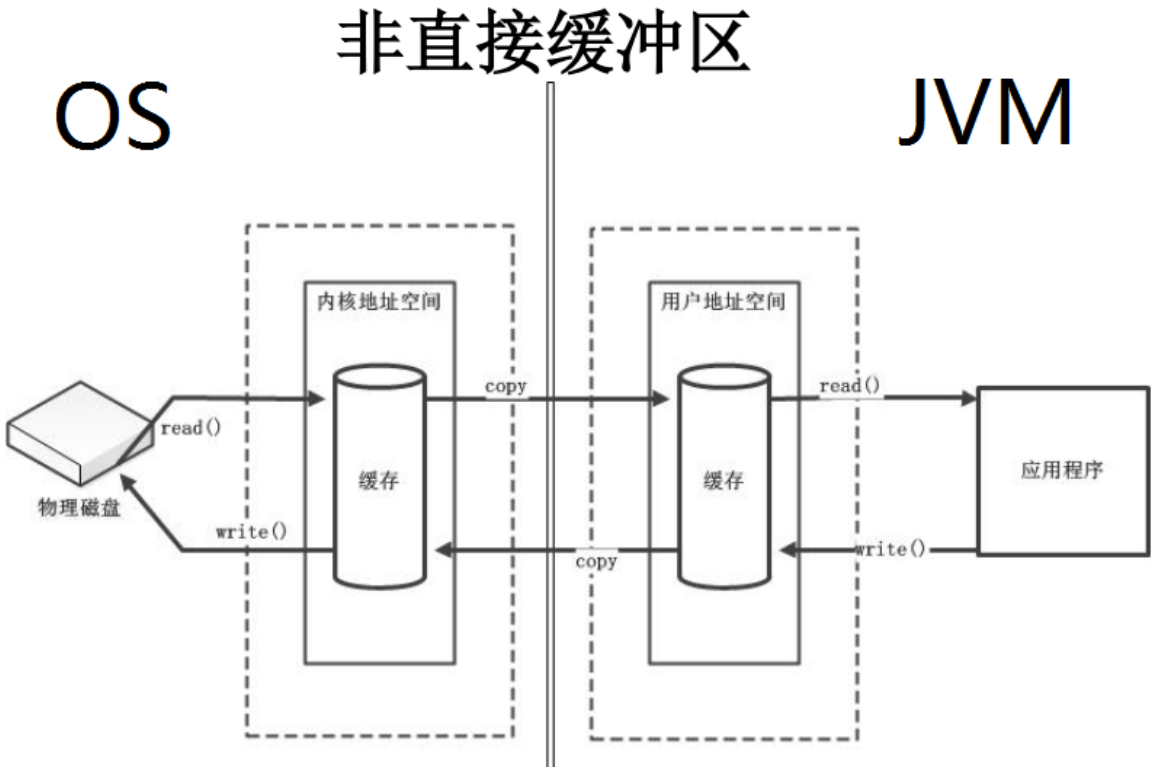
I/O复用模型。通过只阻塞Selector一个线程，通过Selector不断的查询Channel中发生事件，将可处理的事件返回给出现处理.

事件	描述
SelectionKey.OP_CONNECT	连接继续事件，表示服务器监听到了客户连接，服务器可以接收这个连接了
SelectionKey.OP_ACCEPT	连接就绪事件，服务端收到客户端的一个连接请求会触发
SelectionKey.OP_READ	读就绪事件，表示通道中已经有可读的数据了，可以执行读操作
SelectionKey.OP_WRITE	写就绪事件，表示已经可以向通道写数据

文件NIO

非直接缓冲区和直接缓冲区

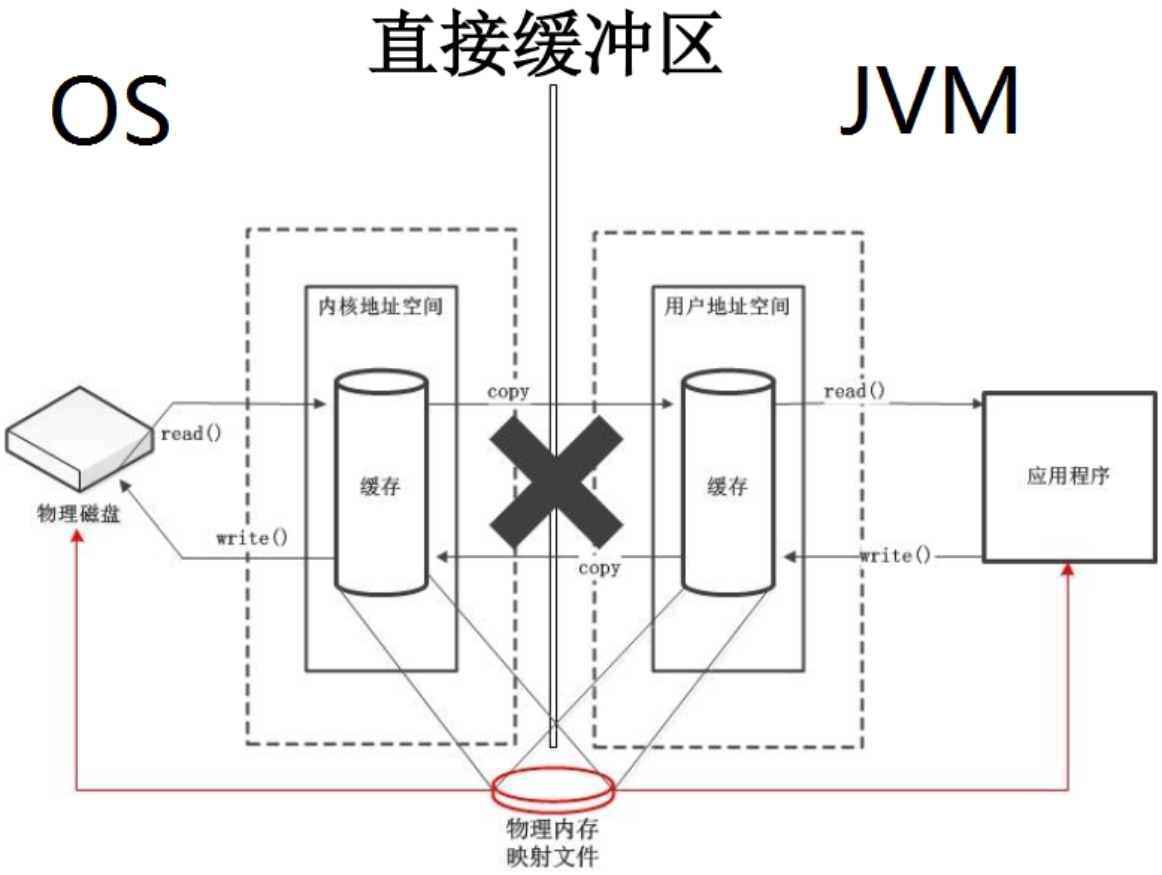
NIO是通过通道连接磁盘文件和应用程序,使用缓冲区存取数据进行双向的数据传输,物理磁盘被操作系统管理,操作物理磁盘的数据需要经过内核地址空间,而java应用程序上分配的缓冲区 是用户地址空间,也就是说文件拷贝 要经过 用户地址空间-----> 内核地址空间.



这样操作下来 文件多了一个拷贝过程.

java NIO中提供了一个直接拷贝文件不需要在,用户地址空间拷贝的方法

`ByteBuffer.allocateDirect(int capacity)` 直接在堆外分配内存.



对比如下

	非直接缓冲区	直接缓冲区
底层实现	数组，JVM 内存	unsafe.allocateMemory(size) 返回直接内存
分配大小限制	-Xms-Xmx 配置的 JVM 内存相关	可以通过 -XX:MaxDirectMemorySize 参数从 JVM 层面去限制
垃圾回收	gc	当 DirectByteBuffer 不再被使用时，会出发内部 cleaner 的钩子，
内存复制	堆内内存 -> 堆外内存 -> pageCache	堆外内存 -> pageCache

建议

- 当需要申请大块的内存时，堆内内存会受到限制，只能分配堆外内存。
- 字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 isDirect() 方法来确定。
- 创建堆外内存的消耗要大于创建堆内内存的消耗，所以当分配了堆外内存之后，尽可能复用它。
- 最好仅在直接缓冲区能在程序性能方面带来明显好处时分配它们。

磁盘IO-文件读写的几种方式对别

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.StandardOpenOption;
import java.time.Duration;
import java.time.Instant;

import lombok.extern.slf4j.Slf4j;

/**
 * 文件复制的几种方式
 *
 * @author miao.chen01@hand-china.com 2019-12-14
 */
@Slf4j
public class FileCopy {

    /**
     * 缓冲大小8k
     */
    private static final int BUF_SIZE = 8192;
    /**
     * copy小文件 4.73M
     */
    private static final File SRC_SMALL = new
File("C:\\Users\\chen\\Desktop\\temp1.pdf");
    /**
     * copy大文件 1.04G
     */
```

```
    private static final File SRC_BIG = new
File("C:\\Users\\chen\\Desktop\\temp2.ISO");
    /**
     * 目标小文件 4.73M
     */
    private static final File DES_SMALL = new File("temp1.pdf");
    /**
     * 目标大文件 1.04G
     */
    private static final File DES_BIG = new File("temp2.ISO");

    public static void main(String[] args) {
        test4();
    }


    /**
     * 测试FileChannel通道传输
     */
    private static void test4() {
        FileCopy fileCopy = new FileCopy();
        FileCopyDoing smallFileChannel = () -> fileCopy.fileChannelCopy3(SRC_SMALL,
DES_SMALL);
        smallFileChannel.calculateTime("FileChannel 通道传输 Copy SMALL File 耗时");
        FileCopyDoing bigFileChannel = () -> fileCopy.fileChannelCopy3(SRC_BIG,
SRC_BIG);
        bigFileChannel.calculateTime("FileChannel 通道传输 Copy big File 耗时");
    }

    /**
     * 通道之间的数据传输
     * FileChannel的transferFrom()方法将数据从源通道传输到FileChannel
     * 另一种是FileChannel的transferTo() 不做演示
     *
     * @param src
     * @param des
     */
    private void fileChannelCopy4(File src, File des) {
        try (
                FileChannel srcChannel = FileChannel.open(src.toPath(),
StandardOpenOption.READ);
                FileChannel desChannel = FileChannel.open(des.toPath(),
StandardOpenOption.WRITE, StandardOpenOption.CREATE);
        ) {
            desChannel.transferFrom(srcChannel, 0L, srcChannel.size());
        } catch (IOException e) {
            log.info("FileChannel copy 异常: ", e);
        }
    }


    /**
     * 测试使用直接缓冲区
     */
    private static void test3() {
        FileCopy fileCopy = new FileCopy();
        FileCopyDoing smallFileChannel = () -> fileCopy.fileChannelCopy3(SRC_SMALL,
DES_SMALL);
```

```
        smallFileChannel.calculateTime("FileChannel direct Copy SMALL File 耗时");
        FileCopyDoing bigFileChannel = () -> fileCopy.fileChannelCopy3(SRC_BIG,
SRC_BIG);
        bigFileChannel.calculateTime("FileChannel direct Copy big File 耗时");
    }

    /**
     * FileChannel 进行文件copy 使用直接缓冲区
     *
     * @param src
     * @param des
     */
    private void fileChannelCopy3(File src, File des) {
        try (
                FileChannel srcChannel = FileChannel.open(src.toPath(),
StandardOpenOption.READ);
                FileChannel desChannel = FileChannel.open(des.toPath(),
StandardOpenOption.WRITE, StandardOpenOption.CREATE);
        ) {
            ByteBuffer buffer = ByteBuffer.allocateDirect(BUF_SIZE);
            while (srcChannel.read(buffer) != -1) {
                buffer.flip();
                desChannel.write(buffer);
                buffer.clear();
            }
        } catch (IOException e) {
            log.info("FileChannel copy 异常: ", e);
        }
    }


    /**
     * 测试FileChannel copy
     */
    private static void test2() {
        FileCopy fileCopy = new FileCopy();
        FileCopyDoing smallFileChannel = () -> fileCopy.fileChannelCopy2(SRC_SMALL,
DES_SMALL);
        smallFileChannel.calculateTime("FileChannel Copy SMALL File 耗时");
        FileCopyDoing bigFileChannel = () -> fileCopy.fileChannelCopy2(SRC_BIG,
SRC_BIG);
        bigFileChannel.calculateTime("FileChannel Copy big File 耗时");
    }

    /**
     * FileChannel 进行文件copy
     *
     * @param src
     * @param des
     */
    private void fileChannelCopy2(File src, File des) {
        try (
                FileChannel srcChannel = FileChannel.open(src.toPath(),
StandardOpenOption.READ);
                FileChannel desChannel = FileChannel.open(des.toPath(),
StandardOpenOption.WRITE, StandardOpenOption.CREATE);
        ) {
            ByteBuffer buffer = ByteBuffer.allocate(BUF_SIZE);
```

```
            while (srcChannel.read(buffer) != -1) {
                buffer.flip();
                desChannel.write(buffer);
                buffer.clear();
            }
        } catch (IOException e) {
            log.info("FileChannel copy 异常: ", e);
        }
    }


    /**
     * 传统 BIO 测试
     */
    private static void test1() {
        FileCopy fileCopy = new FileCopy();
        FileCopyDoing smallBIO = () -> fileCopy.BlockIO(SRC_SMALL, DES_SMALL);
        smallBIO.calculateTime("BIO Copy SMALL File 耗时");
        FileCopyDoing bigBIO = () -> fileCopy.BlockIO(SRC_BIG, DES_BIG);
        bigBIO.calculateTime("BIO copy Big File 耗时");
    }


    /**
     * 传统BIO 文件拷贝
     * @param src
     * @param des
     */
    private void BlockIO(File src, File des) {
        try (
                BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(src));
                BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(des))
        ) {
            byte[] buf = new byte[BUF_SIZE];
            int len;
            while ((len = bis.read(buf)) != -1) {
                bos.write(buf, 0, len);
            }
            bos.flush();
        } catch (IOException e) {
            log.info("BIO copy 异常: ", e);
        }
    }


}

interface FileCopyDoing {

    /**
     * 分割
     */
    String SPLIT = " : ";
    /**
     * 毫秒
     */
```

```
    String TIME_MS = "ms";

    /**
     * 计算文件拷贝时间
     *
     * @param des
     */
    default void calculateTime(String des) {
        Instant start = Instant.now();
        doFileCopy();
        Instant end = Instant.now();
        Duration between = Duration.between(start, end);
        long total = between.toMillis();
        System.out.println(des + SPLIT + total + TIME_MS);

    }

    /**
     * 执行文件拷贝
     */
    void doFileCopy();
}
```

测试结果如下

大家在测试的时候需要多次测试，但是连续多次运行也会对结果有印象

- 传统BIO测试

```
BIO Copy SMALL File 耗时 : 6ms
BIO copy Big File 耗时 : 2734ms
```

- 使用FileChannel

```
FileChannel Copy SMALL File 耗时 : 15ms
FileChannel Copy big File 耗时 : 1490ms
```

- FileChannel 直接缓冲区

```
FileChannel direct Copy SMALL File 耗时 : 12ms
FileChannel direct Copy big File 耗时 : 1239ms
```

- FileChannel通道传输

```
FileChannel 通道传输 Copy SMALL File 耗时 : 12ms
FileChannel 通道传输 Copy big File 耗时 : 1160ms
```

可以看出,在读写小文件的时候,传统BIO运行的还是比较快的, 当文件比较大的时候,可以考虑使用FileChannel的通道传输(2G以内),写代码也方便.如果堆内存不够,可以考虑使用直接缓冲区.

网络IO

BIO已经在之前的例子给出,现在我们来学习,网络中如何使用NIO.

java实现NIO 使用的是I/O多路复用,I/O多路复用依赖一个事件多路分离器(就是java中的Selector),可以将时间源的I/O事件分离处理(比如分离accept,read,write...),分发给对应的事件处理器(处理read 事件,write事件),程序员需要先注册需要处理的事件和处理器.

有两种模式Reactor和Proactor. **Reactor采用同步IO,Proactor采用异步IO**

Reactor 事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

Proactor 处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作.IO操作本身由操作系统来完成。**能够胜任那些重量级，读写过程长的任务** Proactor 模式简化了Reactor 模式,让开发更简单.

Reactor 模式

eg Server端

```
package com.example.demoothers1.demo4.day3.Reactor;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.Iterator;
import java.util.Objects;

import lombok.extern.slf4j.Slf4j;

/**
 * NIO Reactor 模式 服务器
 *
 * @author miao.chen01@hand-china.com 2019-12-15
 */
@Slf4j
public class ReactorServer {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel server = ServerSocketChannel.open();
        // 切换非阻塞模式
        server.configureBlocking(false);
        server.bind(new InetSocketAddress(12345));
        // 获取选择器
        Selector selector = Selector.open();
        // 将通道注册到选择器上，指定接收“连接就绪”事件
        server.register(selector, SelectionKey.OP_ACCEPT);
        // 轮询 已就绪的事件!=0
        while (selector.select() != 0) {
            Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                // 必须在selectedKeys中删除,迭代中删除,需要使用迭代器
                iterator.remove();
                // 连接就绪
                if (selectionKey.isAcceptable()) {
```

```
                    try {
                        SocketChannel client = server.accept();
                        // 设置客户端非阻塞
                        client.configureBlocking(false);
                        // 注册读就绪事件 注意客户端注册到服务器的selector上
                        client.register(selector, SelectionKey.OP_READ);
                        log.info("accept {}", client.hashCode());
                    } catch (IOException e) {
                        log.info("连接accept 错误!", e);
                    }
                } else if (selectionKey.isReadable()) {
                    // 读事件就绪,获取通道
                    SocketChannel client = (SocketChannel) selectionKey.channel();
                    log.info("read {}", client.hashCode());
                    ByteBuffer buffer = ByteBuffer.allocate(8196);
                    FileChannel outChannel = null;
                    try {
                        // 将客户端的文件写到本地项目下
                        outChannel = FileChannel.open(Paths.get("temp1.pdf"),
StandardOpenOption.WRITE, StandardOpenOption.CREATE);
                    } catch (IOException e) {
                        log.info("打开文件通道失败! ", e);
                    }
                    while (client.read(buffer) > 0) {
                        buffer.flip();
                        Objects.requireNonNull(outChannel).write(buffer);
                        buffer.clear();
                    }

                    //发送写完
                    log.info("发送中...");
                    ByteBuffer buf = ByteBuffer.allocate(1024);
                    buf.put("文件发送成功! ".getBytes());
                    buf.flip();
                    client.write(buf);
                    log.info("发送中....结束");
                    Objects.requireNonNull(outChannel).close();
                    client.close();
                }

            }

        }
    }
}
```

客户端

```
package com.example.demoothers1.demo4.day3.Reactor;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
```

```java
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.Iterator;

import lombok.extern.slf4j.Slf4j;

/**
 * NIO Reactor 模式 客户端
 *
 * @author miao.chen01@hand-china.com 2019-12-15
 */
@Slf4j
public class ReactorClient {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open(new
InetSocketAddress("127.0.0.1", 12345));
        // 设置非阻塞
        socketChannel.configureBlocking(false);
        // 完成之后获取服务器消息,也需要注册轮询事件
        Selector selector = Selector.open();
        // 注册read事件,读取服务器发送消息
        socketChannel.register(selector, SelectionKey.OP_READ);
        //发送的文件
        FileChannel inChannel =
FileChannel.open(Paths.get("C:\\Users\\chen\\Desktop\\temp1.pdf"),
StandardOpenOption.READ);
        ByteBuffer buffer = ByteBuffer.allocate(8196);
        while (inChannel.read(buffer) != -1) {
            buffer.flip();
            socketChannel.write(buffer);
            buffer.clear();
        }
        socketChannel.shutdownOutput();
        while (selector.select() != 0) {
            Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                iterator.remove();
                if (selectionKey.isReadable()) {
                    SocketChannel channel = (SocketChannel) selectionKey.channel();
                    ByteBuffer buf = ByteBuffer.allocate(1024);
                    int len;
                    if ((len=channel.read(buffer)) != -1) {
                        buf.flip();
                        log.info("客户端接收通知: {}", new String(buffer.array(), 0,
len));
                        return;
                    }
                }
            }

        }
        inChannel.close();
        socketChannel.close();

    }
}
```

结果

```
12:28:18.108 [main] INFO com.example.demoothers1.demo4.day3.Reactor.ReactorClient - 客户端接收通知: 文件发送成功!
```

Proactor模式

```
package com.example.demoothers1.demo4.day3.Proactor;

import com.google.common.util.concurrent.ThreadFactoryBuilder;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousChannelGroup;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import lombok.extern.slf4j.Slf4j;

/**
 * Proactor 模式 服务端
 *
 * @author miao.chen01@hand-china.com 2019-12-15
 */
@Slf4j
public class ProactorServer {
    public static void main(String[] args) throws IOException, InterruptedException {
        ThreadPoolExecutor pool = new ThreadPoolExecutor(10,
                20,
                60,
                TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(1000),
                new ThreadFactoryBuilder().build());
        // 线程池绑定到通道组
        AsynchronousChannelGroup channelGroup =
AsynchronousChannelGroup.withCachedThreadPool(pool, 1);
        // 开启异步ServerSocket
        AsynchronousServerSocketChannel assc =
AsynchronousServerSocketChannel.open(channelGroup);
        assc.bind(new InetSocketAddress(12345));

        MyMessage message = new MyMessage();
        assc.accept(message, new CompletionHandler<AsynchronousSocketChannel,
MyMessage>() {
            @Override
            public void completed(AsynchronousSocketChannel result, MyMessage
attachment) {
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                Integer size = null;
```

```
                try {
                    size = result.read(buffer).get();
                    if (size > 0) {
                        attachment.setInfo(new String(buffer.array(), 0, size));
                        log.info("read message");
                    }
                    // 发数据
                    result.write(ByteBuffer.wrap("server 发送数据测试".getBytes()));
                } catch (InterruptedException | ExecutionException e) {
                    log.info("server log err", e);
                } finally {
                    try {
                        result.close();
                    } catch (IOException e) {
                        log.info("关闭失败 ", e);
                    }
                }


            }

            @Override
            public void failed(Throwable exc, MyMessage attachment) {
                log.info("accpet 错误!", exc);
            }
        });

        while (message.getInfo() == null) {
            Thread.sleep(500);
        }
        log.info("get message" + message.getInfo());

        // 关闭
        assc.close();
        channelGroup.awaitTermination(10, TimeUnit.SECONDS);
        if (!channelGroup.isTerminated()) {
            channelGroup.shutdownNow();
        }

        pool.awaitTermination(10, TimeUnit.SECONDS);
        if (!pool.isTerminated()) {
            pool.shutdownNow();
        }
    }
}
```

```
package com.example.demoothers1.demo4.day3.Proactor;


import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
```

```
import lombok.Data;
import lombok.extern.slf4j.Slf4j;

/**
 * Proactor 模式客户端
 *
 * @author miao.chen01@hand-china.com 2019-12-15
 */
@Slf4j
public class ProactorClient {
    public static void main(String[] args) throws IOException, InterruptedException {
        AsynchronousSocketChannel asc = AsynchronousSocketChannel.open();
        // 监听连接事件,发送数据
        asc.connect(new InetSocketAddress("127.0.0.1",12345),
                null, new CompletionHandler<Void, Object>() {
            @Override
            public void completed(Void result, Object attachment) {
                try {
                    asc.write(ByteBuffer.wrap("客户端 test".getBytes())).get();
                } catch (InterruptedException | ExecutionException e) {
                    log.info("发送消息失败 !", e);
                }
            }

            @Override
            public void failed(Throwable exc, Object attachment) {
                log.info("连接事件失败:", exc);
            }
        });

        //监听读数据
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        MyMessage message = new MyMessage();
        asc.read(buffer, message, new CompletionHandler<Integer, MyMessage>() {
            @Override
            public void completed(Integer result, MyMessage attachment) {
                if (result > 0) {
                    attachment.info = new String(buffer.array(), 0, result);
                }
            }

            @Override
            public void failed(Throwable exc, MyMessage attachment) {
                log.info("read事件失败! ", exc);
            }
        });

        // 当没有消息
        while (message.info == null) {
            log.info("info is null");
            Thread.sleep(500);
        }
        log.info("get message : " + message.getInfo());
        asc.close();
    }
}

@Data
```

```
class MyMessage {
    String info;

}
```

结果

```
# 服务端

13:08:10.505 [main] INFO com.example.demoothers1.demo4.day3.Proactor.ProactorClient -
info is null
13:08:11.009 [main] INFO com.example.demoothers1.demo4.day3.Proactor.ProactorClient -
get message : server 发送数据测试

# 客户端
13:08:10.503 [pool-1-thread-2] INFO
com.example.demoothers1.demo4.day3.Proactor.ProactorServer - read message
13:08:10.696 [main] INFO com.example.demoothers1.demo4.day3.Proactor.ProactorServer -
get message客户端 test
```

学习资料

JAVA中的I/O和NIO

https://juejin.im/post/5bac9392e51d450e805b679c

https://www.jianshu.com/p/8dbb0686fb8b

https://www.cnblogs.com/dolphin0520/p/3916526.html

https://www.jianshu.com/p/c5c37e464d0f

https://blog.csdn.net/elricboa/article/details/79034617

https://www.jianshu.com/p/e55312ec5b03

https://www.jianshu.com/p/dfd940e7fca2