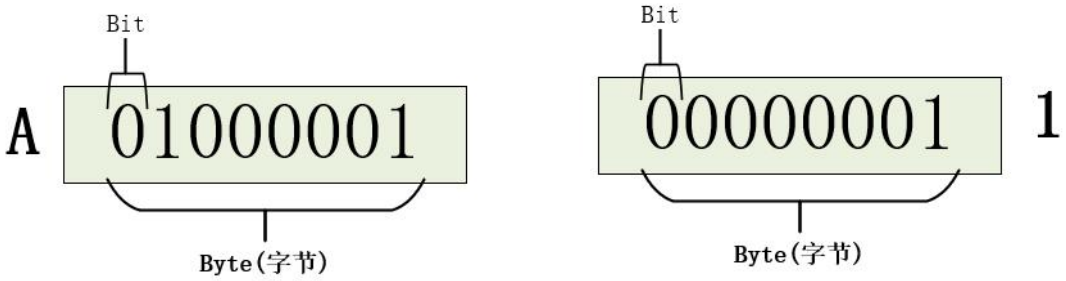
# BIO,NIO，AIO 笔记

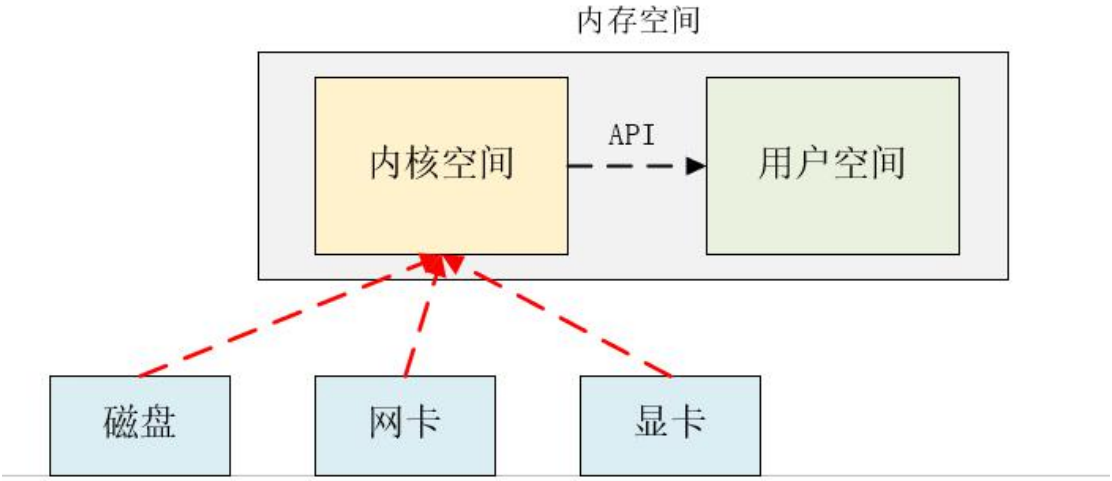
一、数据

1. 计算机中的数据都是以二进制为基础进行表达的，通常用 1bit 表示一个二进制位，8 个二进制位即 8bit 为一个表示单元 Byte,即字节。示例如下




二、数据的内核态和用户态

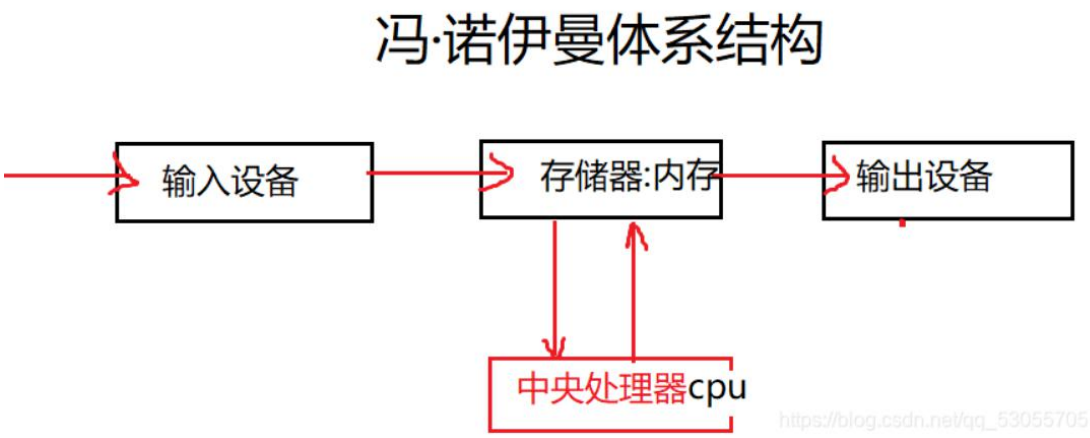
1. 为防止用户代码对计算机系统造成严重的损坏，计算机系统不允许用户代码直接调用系统的磁盘文件、网卡，显卡等，而是通过计算机系统给定的 API 来进行间接调用。
2. 这就导致计算机内存（RAM）分为大致的两块：用来存储用户代码的用户空间（如 JVM）
3. 其中数据在内核空间称为内核态，在用户空间称为用户态




三、IO 流

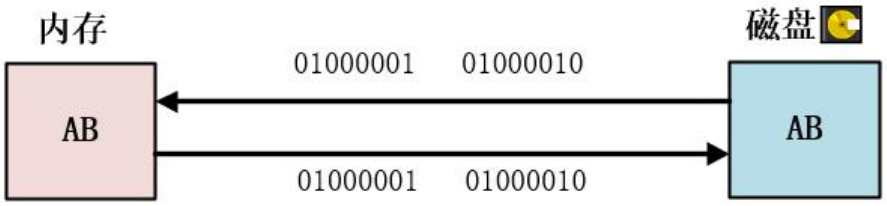
什么是 IO

IO 即 input 和 output 的缩写，根据冯诺依曼的计算机系统架构，一个计算机系统包括：输入设备、输出设备，CPU 和存储设备。




常见的输入设备：键盘、鼠标、磁盘、网卡等
常见的输出设备：显示器，磁盘、网卡等

1. 在计算机系统中，各个设备之间的数据交流都是以二进制形式,以字节 byte 为基本单元，进行交互的，如键盘、鼠标和主机的交互，内存和磁盘之间的交互，内存和网路之间的交互等；
2. 在 java 中，我们将这种二进制形式，以字节 byte 为基本单元的数据称之为 IO 流。其中获取到的流称之为输入流，向外输出的流称之为输出流；

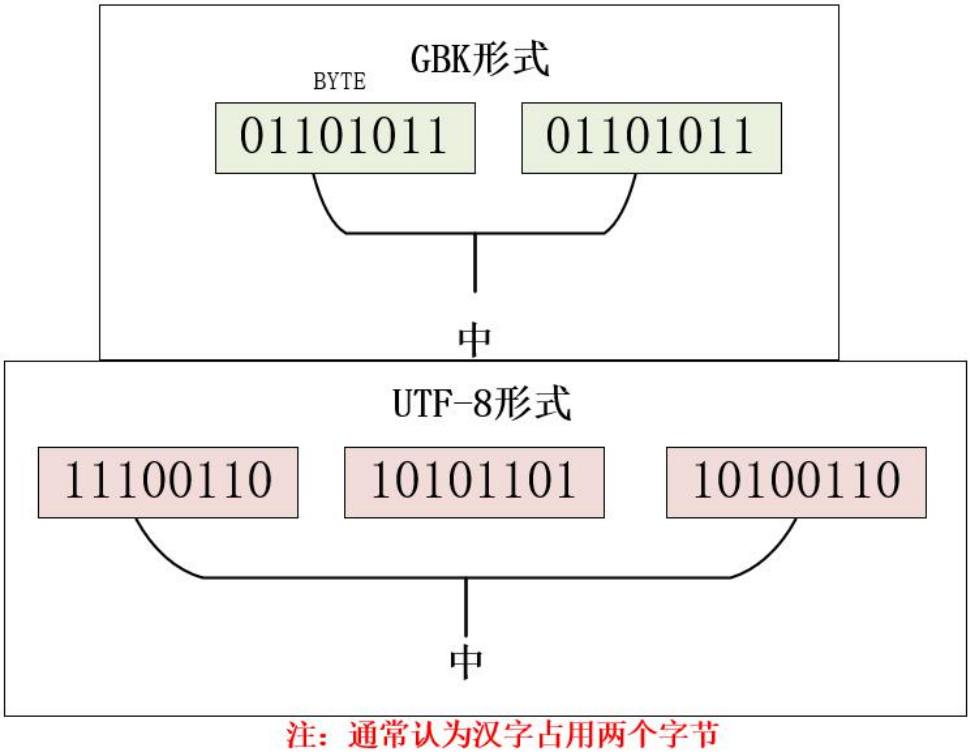



3. 在 Java 中，流的表现方式分为两种：字节流和字符流

(1) 字节流：以字节(Byte)为基本单位，可以处理任意形式的文件，抽象类为 InputStream 和 OutputStream；

(2) 字符流：

- 因为字节 Byte，并没有描述汉字以及其他字符的能力，所以需要对没用或者不常用的 byte 的二进制值,进行合理的分配组合，进而拼接成汉字等其他需要的字符；这种组合方式我们称之为编码，如 GBK，UTF-8，GB2312，Unicode 等。
- 所以，字符流是是为方便使用，对字节流的包装类，以字符(Char)为基本单位，只能处理文本形式的数据，抽象类为 Reader 和 Writer；
- 同时字符流的使用要注意编码问题，否则很容易乱码；




注：通常认为汉字占用两个字节

(3) 字节流和字符流都是通过输入流的 read 方法来读取数据，输出流的 write 方法来写入到流中；

(4) 另外，File 类是一种非流式的文件处理方式

Java 输入/输出流 体系中常用的流

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输入流 |
|---|---|---|---|---|
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | ***FileInputStream*** | ***FileoutputStream*** | ***FileReader*** | ***FileWriter*** |
| 访问数组 | ***ByteArrayInputStream*** | ***ByteArrayOutputStream*** | ***CharArrayReader*** | ***CharArrayReader*** |
| 访问管道 | ***PipedInputStream*** | ***PipedOutputStream*** | ***PipedReader*** | ***PipedWriter*** |
| 访问字符串 | | | ***StringReader*** | ***StringWriter*** |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| 抽象基类 | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |

*粗斜体：节点流，必须直接与指定物理节点关联*

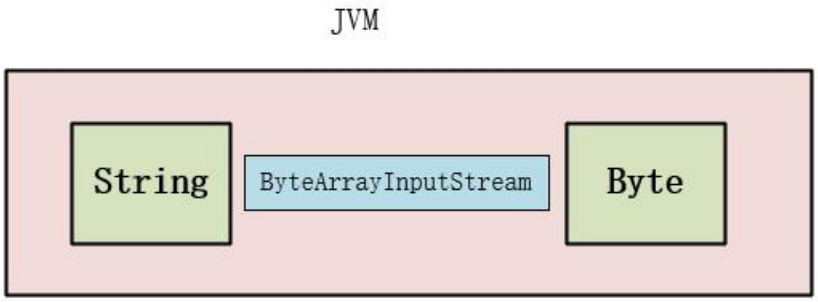
(5) 基于上述四种基本类，根据不同的数据类型，选择不同的具体类来处理文件，我们称之为 IO 管道，因为通过该类沟通了不同的媒介，

① 如：磁盘和内存的 IO 通道

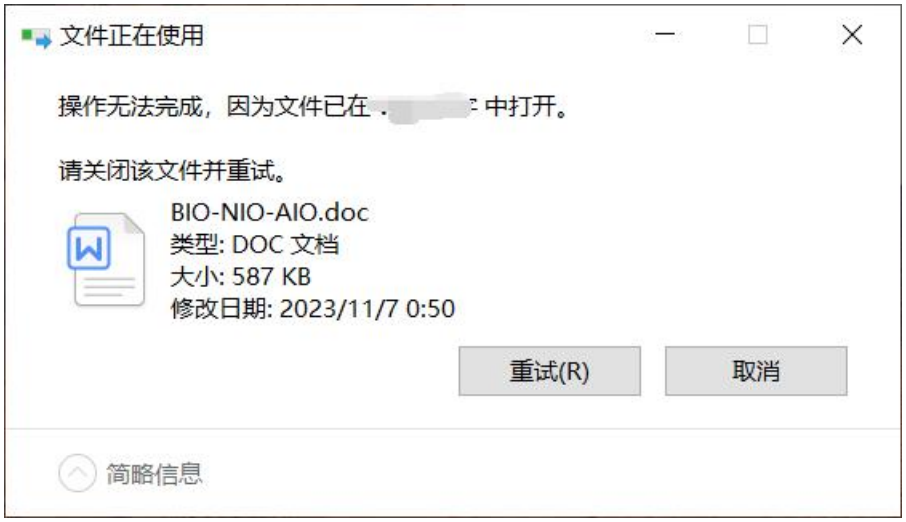
```
//磁盘IO
try {
    FileInputStream inputStream = new FileInputStream( name: "E:/test.txt");
}catch (Exception e){
    e.printStackTrace();
}
```


② 如：内存到内存的 IO 通道

```java
//内存
String str = "hello world";
ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(str.getBytes());
int i = 0;
while((i=byteArrayInputStream.read())!=-1){
    System.out.println((char)i);
}
```




(6) 流用完一定要记得关闭，否则文件将一直处于占用的状态。




四、BIO、NIO 模型（非网络）

（一）BIO

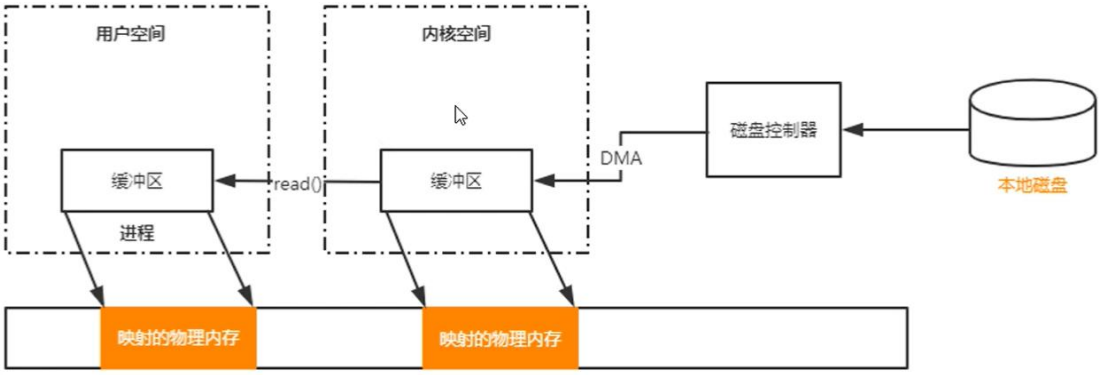
1. BIO（Blocking IO），同步阻塞 IO，是通过系统内核 API 调用系统文件（如磁盘文件、网卡，显卡等）的一种方式，采用同步阻塞等待的方式，将系统数据，转变为内核态再转变为用户态。
2. 当用户空间调用内核 API，进行读取数据操作时：

① 用户空间会在内存（RAM）上分配一块内存，作为缓冲池（大小可以默认，也可以进行自定义，默认通常是 1 字节大小（1byte=8bit），即几乎认为是直接面向数据源）（注：字符流是字节流的包装类，底层也是用字节流传输的）；

② 内核空间同时也会在内存上默认的分配一块内存作为缓冲池；

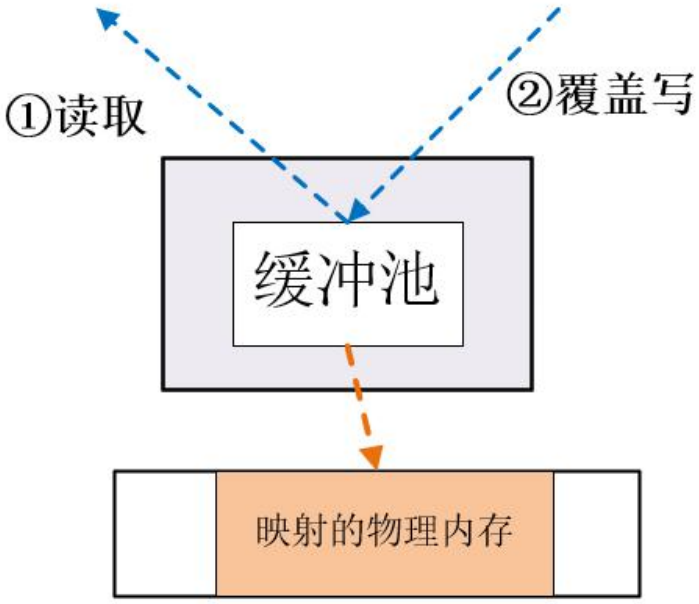
③ 用户空间需要一直处于阻塞等待状态，直至内核空间用缓冲池将用户空间的缓冲池填满或者将数据的剩余部分填充至用户空间的缓冲池中；

④ 然后用户空间获取到缓冲池中的流进行处理；

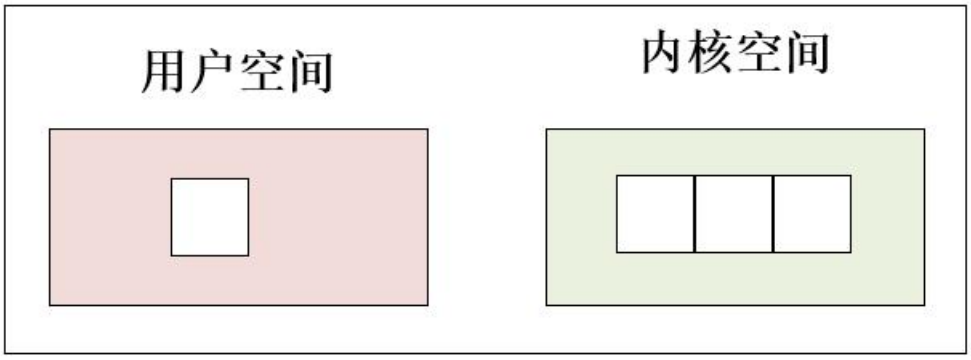


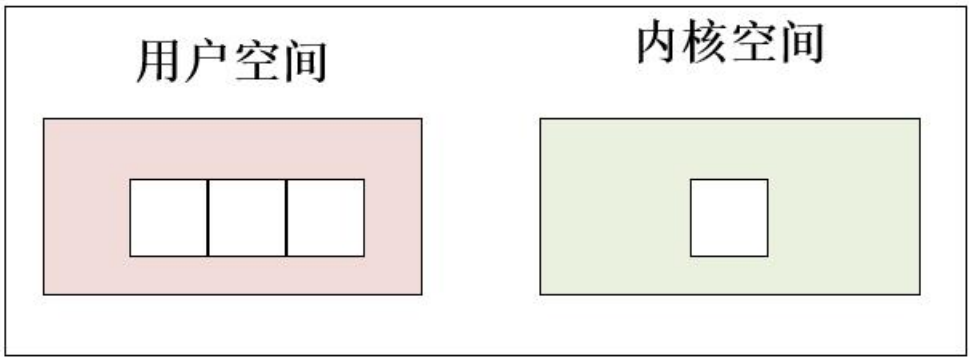

⑤ 需要注意的点（下同）：

- 缓冲区（内存/RAM）是可以被不断地覆写再利用的
- 在用户空间，缓冲区的表现为：缓冲区被读取到流中后，对应的内存空间就可以被下一次内核空间的数据覆写了；
- 在内核空间，缓冲区的表现为：缓冲区被用户空间读取的内存部分，就可以被 CPU 进行覆写了；

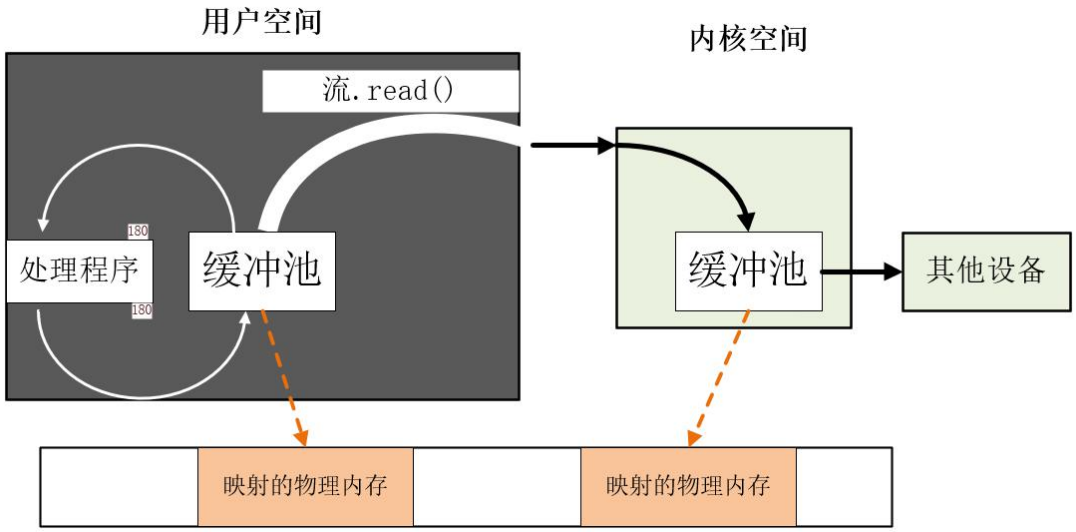



- 磁盘空间缓冲区的大小和用户空间缓冲区的大小并不是一致的，这就意味着用户缓冲区需要多次读内核缓冲区，或者内核缓冲需要多次覆写用户缓冲区。




- 流仅仅是一种 IO 管道，真正发送和接收的是，通过管道流入或流出的缓冲区中的数据，即内存中的二进制数据；




- 所以流并不一定要等到读完才可以进行处理，只要读取到缓冲区中的数据可以进行处理；

```java
public class ReadDemo {
    public static void main(String[] args) {
        try(FileInputStream fileInputStream = new FileInputStream( name: "E:/test.txt")){
            int i = 0;
            byte[] buffer = new byte[1024*1024];// 占用内存空间
            while((i=fileInputStream.read(buffer))!=-1){
                System.out.println(new String (buffer, offset: 0,i));
            }
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

缓冲区

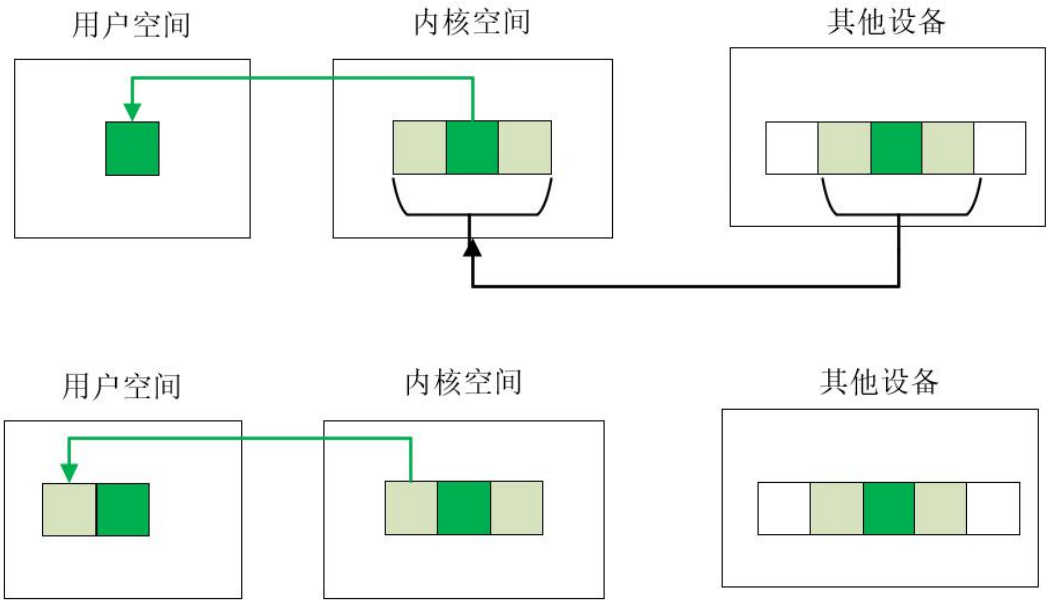
每次读到缓冲区，就可以拿到缓冲区的信息

如果返回-1说明文件已经读完

- 内核缓冲区的意义：

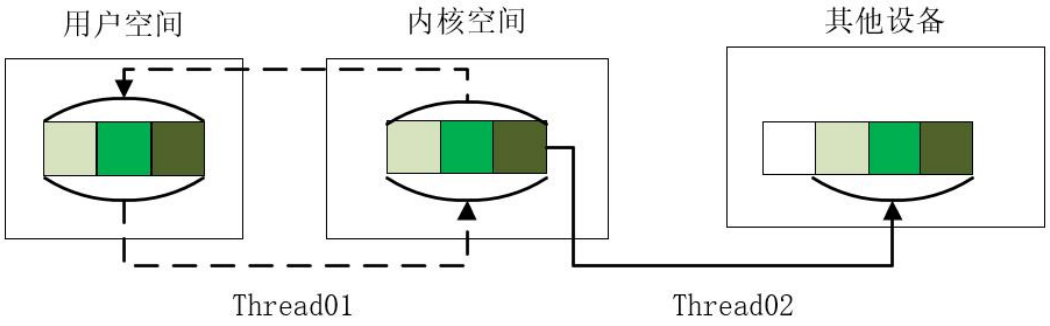
  内核缓冲区的存在提升了数据的读写效率，具体如下：

a. 当用户空间向内核空间发送读取数据的请求时，内核空间会将用户空间所需的数据，以及该数据领近的数据一起缓存到内核空间的缓冲池中；如果用户空间下一次读取附近的数据，将直接从内核空间的缓冲池中读取，提升了读取效率；如：磁盘文件通常 4KB 为一页，如果是读取磁盘文件，内核空间缓冲池将是 4KB 的整数倍；




下一次直接从内核空间读取

b. 当用户空间进行写操作时，内核空间的缓冲池会先将用户空间的数据缓存起来，进行异步写入，提高了数据的写入效率；



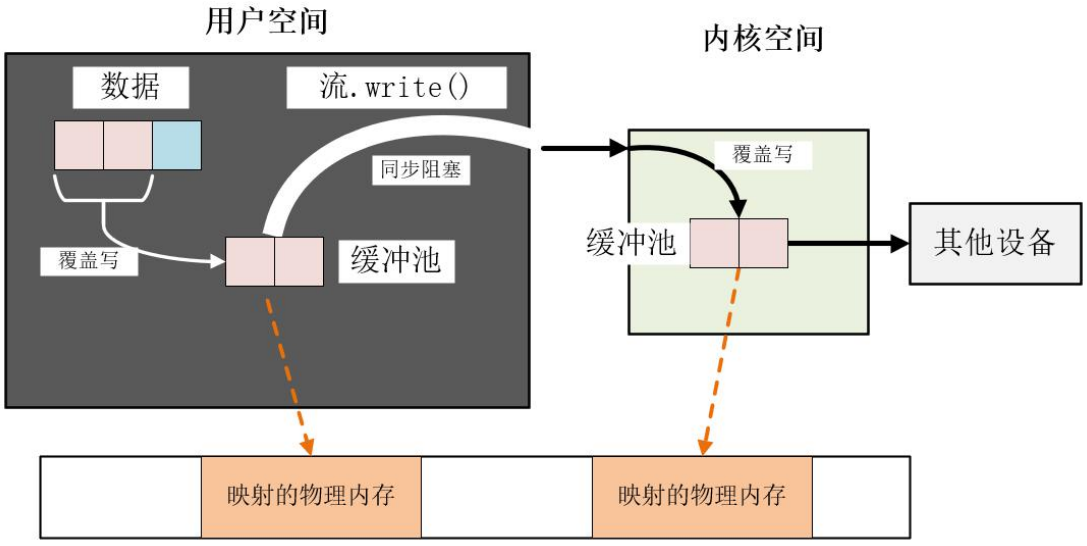

3. 当用户空间调用内核 API，进行写入数据操作时：

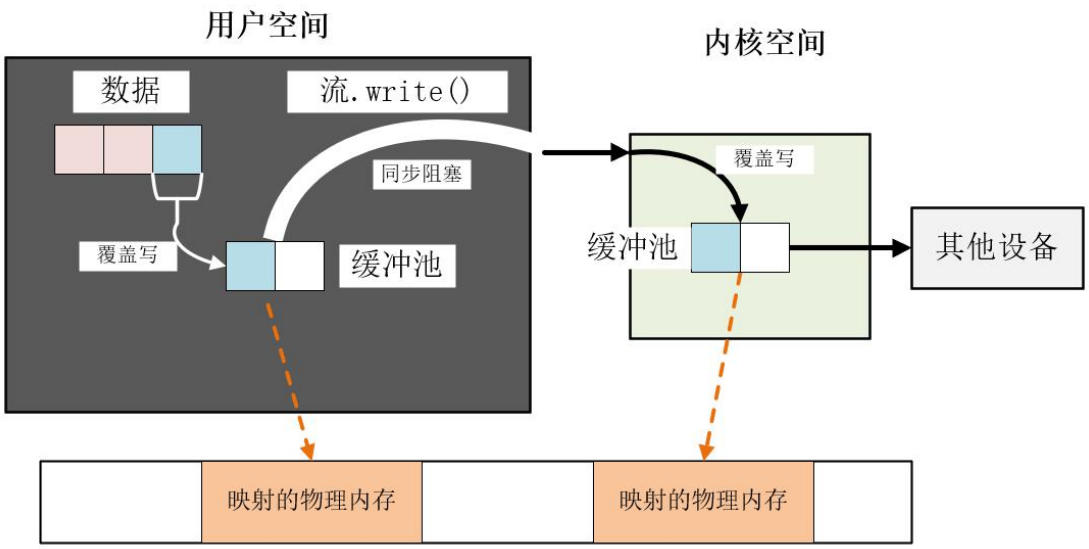
① 用户空间会在内存（RAM）上分配一块内存，作为缓冲池（大小可以默认，也可以进行自定义，默认通常是 1 字节大小（1byte=8bit），即几乎认为是直接面向数据源）（注：字符流是字节流的包装类，底层也是用字节流传输的）；

② 内核空间同时也会在内存上默认的分配一块内存作为缓冲池；

③ 数据先写入到用户空间的缓冲池中，然后由 IO 管道调用内核空间 API，将用户空间缓冲池的数据写入到内核空间缓冲池中，再由内核空间写入到相应设备中；

④ 在用户空间缓冲池未被内核空间缓冲池消费完前，线程将一直处于阻塞等待状态；





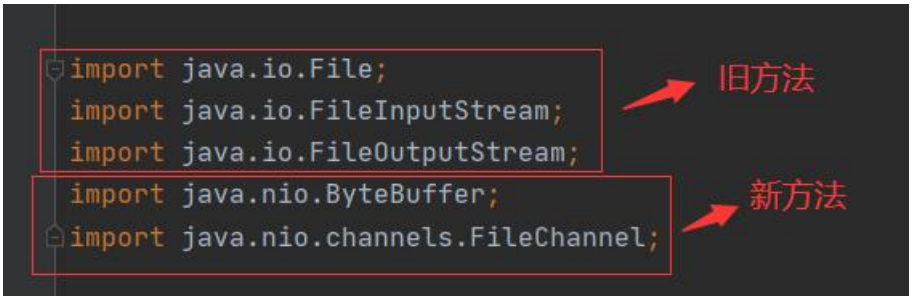

⑤ 当数据量过大时，用户空间缓冲池并不足以一次性将数据全部传输完毕，就需要 IO 流通道进行多次传输；

```java
public class Test02 {
    public static void main(String[] args) throws Exception {
        FileOutputStream file = new FileOutputStream( name: "E:/test.txt");
        String str = "hello world,How are you?";

        byte[] bys = new byte[3];

        InputStream in = new ByteArrayInputStream(str.getBytes());
        BufferedOutputStream out = new BufferedOutputStream(file);
        int len = 0;

        while ((len = in.read(bys)) != -1) {
            out.write(bys, off: 0, len);
        }

        in.close();
        out.close();

    }
}
```



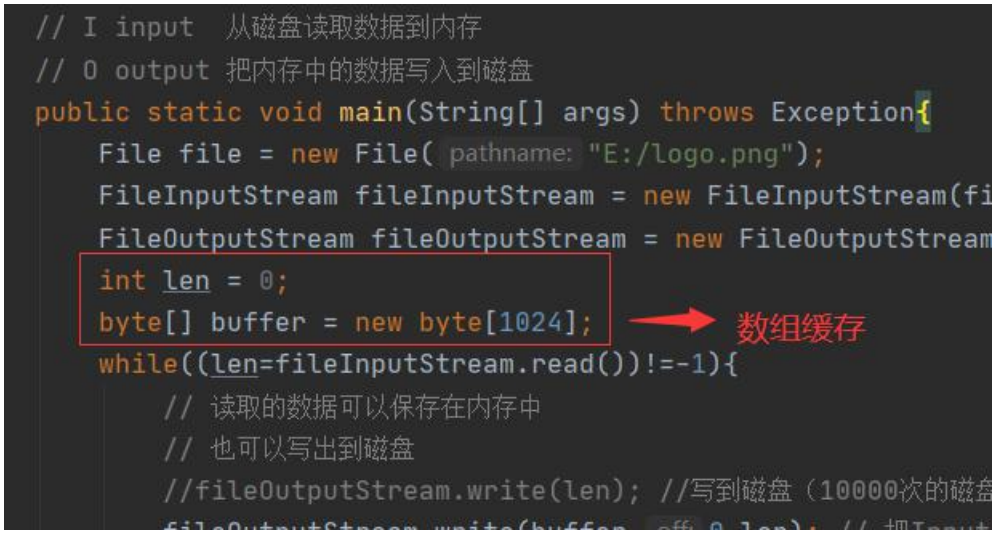
（二）NIO

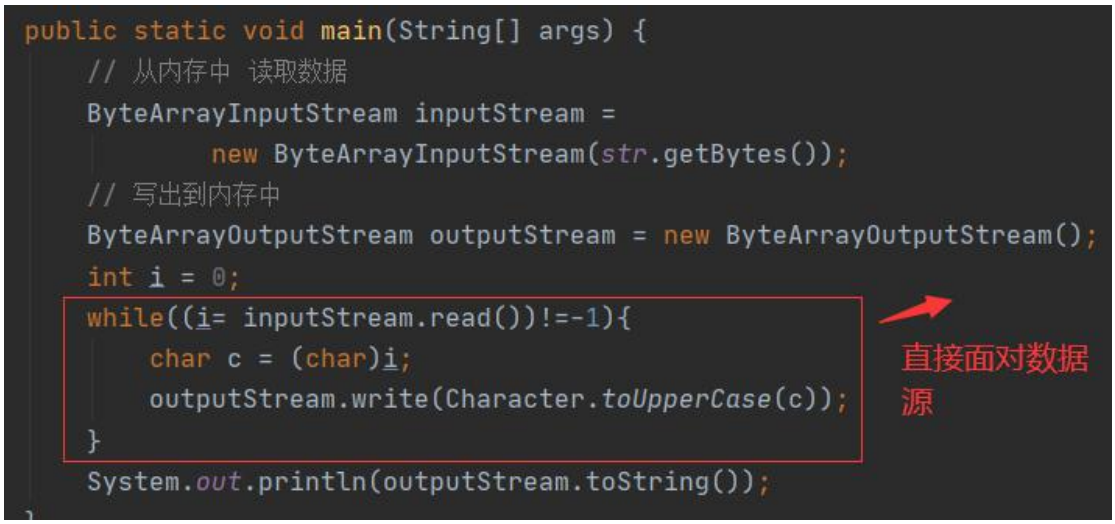
1. NIO：java 中的 NIO 又叫 NEW IO,是对旧 IO 的补充规范，即对 BIO 的补充规范。
2. 在 java 中，非网络的 NIO 同样是同步阻塞的，仅仅是对 BIO 的 IO 通道和缓冲池进行了升级，底层原理、写法几乎一致；
3. Java 的 NIO 位于 java.nio 包下面，可以配合之前的 java.io 包里的方法进行使用




4. 关于缓冲池 buffer 的升级：

- 在 BIO 中，缓存是以字节字节数组（Byte[]）进行规定的,或者直接不规定，使用默认值 1byte，直接面对数据源。




- NIO 中，使用的是缓存对象，结构复杂且使用灵活，本质也是 byte 数组，只是在 NIO 中封装成了 buffer 对象，并提供了一组方法来访问缓存，即 RAM 内存；
- Java 的 NIO（非网络）必须有缓存对象，而且必须手动声明，没有默认值，基本数据类型都有对应地缓冲对象，通常使用 ByteBuffer 较多；

**Buffer**

内存缓存区，用于暂存从Channel中读取的数据、以及即将写入Channel中的数据。

常见的Buffer有：

- ByteBuffer（主要使用）
  - MappedByteBuffer
  - DirectByteBuffer
  - HeapByteBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FloatBuffer
- DoubleBuffer
- CharBuffer

从名字上可以看出，这些Buffer支持的是不同数据类型的缓冲区，如short、int等。

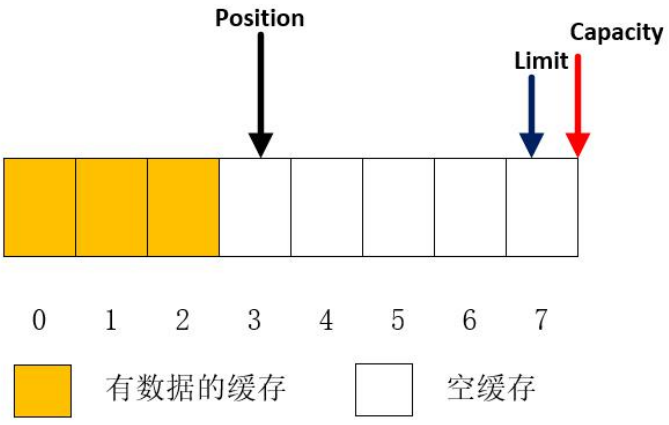
主要使用

```
@Test
public void test_04(){
    ByteBuffer byteBuffer_01 = ByteBuffer.allocate(10);

    byte[] bytes = new byte[10];
    ByteBuffer byteBuffer_02 = ByteBuffer.wrap(bytes);
}
```

声明方式1

声明方式2

- NIO 缓存对象三个重要的属性：
  ① Capacity：容量，缓冲池（缓存）的大小；
  ② Position：下一个要读取或写入的元素的索引；
  ③ Limit：读取或写入的上限索引值；



```java
@Test
public void test_01(){
    LongBuffer buffer  = LongBuffer.allocate(10);
    buffer.put(new long[]{1l,2l,3l,4l,5l});
    System.out.println("capacity: "+buffer.capacity());
    System.out.println("position: "+buffer.position());
    System.out.println("limit: "+buffer.limit());
    System.out.println("数据: "+ Arrays.toString(buffer.array()));
}
```

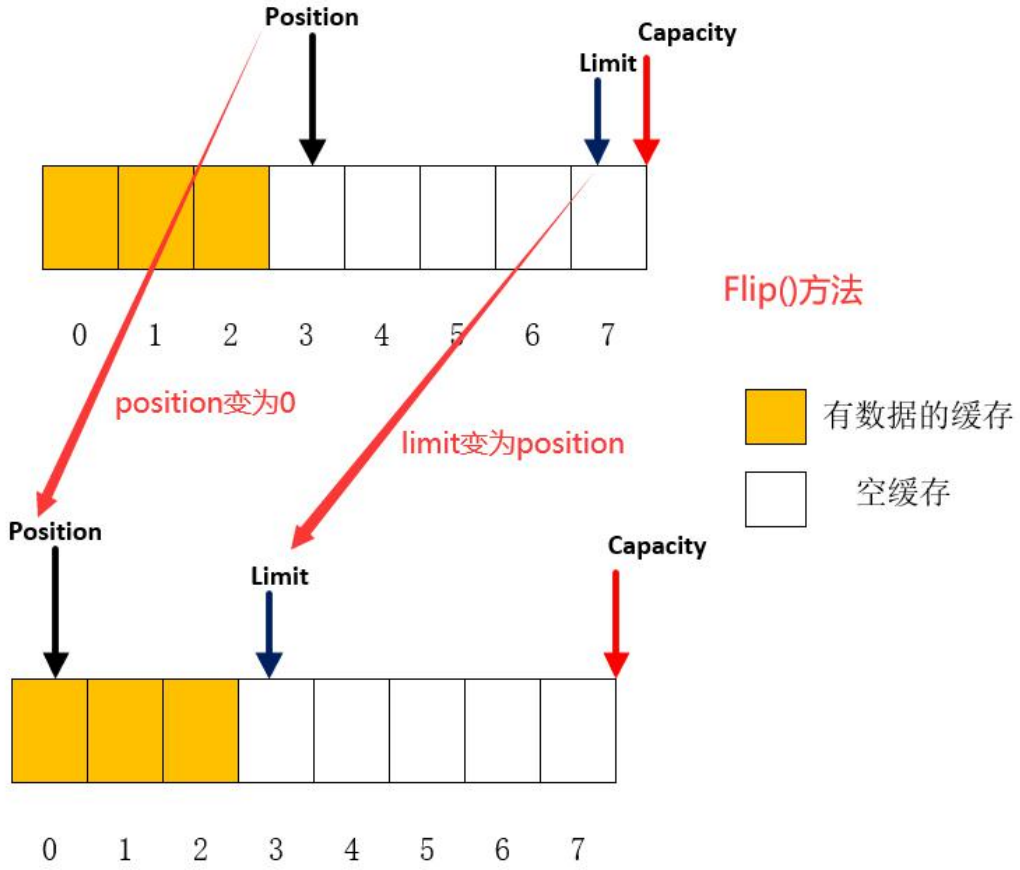
```
"E:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
capacity: 10
position: 5
limit: 10
数据: [1, 2, 3, 4, 5, 0, 0, 0, 0, 0]
```

- NIO 缓存对象两个重要的方法：

① Filp(): 很多技术文档说是，缓冲池读模式和写模式的互相切换，其实并不是很准确，因为所谓的读模式下，也可以进行数据的插入，所谓的写模式下也可以进行数据的读取；

官方的解释为：将缓冲池的 limit 值设置为 position 值，并且将 position 值设置为 0；

```java
/** 翻转当前缓冲池
 * Flips this buffer.  The limit is set to the current position and then
 * the position is set to zero.  If the mark is defined then it is
 * discarded.
 *                                  将limit设置为position，并且将position设置为
 * <p> After a sequence of channel-read or <i>put</i> operations, invoke
 * this method to prepare for a sequence of channel-write or relative
 * <i>get</i> operations.  For example:
 *
 *                                  通常是在读操作后调用，方便写入；
 *                                  或者是在写操作后调用，方便读取；
 * <blockquote><pre>
 * buf.put(magic);    // Prepend header
 * in.read(buf);      // Read data into rest of buffer
 * buf.flip();        // Flip buffer
 * out.write(buf);    // Write header + data to channel</pre></blockquote>
 *
 * <p> This method is often used in conjunction with the {@link
 * java.nio.ByteBuffer#compact compact} method when transferring data from
 * one place to another.  </p>
 *
 * @return  This buffer
 */
@NotNull
public final Buffer flip() {
```




② Put()：向 NIO 缓存中插入数据，每插入一个，Position 就向后移动一位；

```
@Test
public void test_02(){
    LongBuffer buffer  = LongBuffer.allocate(10);
    System.out.println("position: "+buffer.position());
    buffer.put(new long[]{1l,2l,3l,4l,5l});
    System.out.println("position: "+buffer.position());
    buffer.put(new long[]{6l});
    System.out.println("position: "+buffer.position());
}
```

插入5个

插入1个

```
"E:\Program Files\Ja
position: 0
position: 5
position: 6
```

③ Get():从 NIO 缓存中获取数据，每获取一个，position 就向后移动一位；

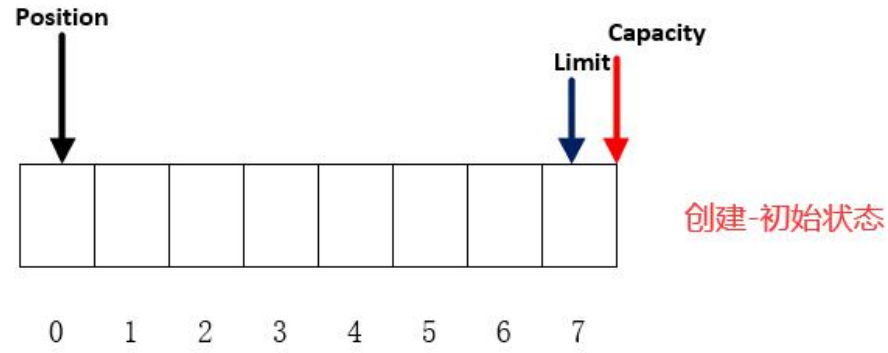
```java
@Test
public void test_03(){
    LongBuffer buffer  = LongBuffer.allocate(7);
    buffer.put(new long[]{1l,2l,3l,4l,5l});
    //翻转
    buffer.flip();

    System.out.println("position："+buffer.position());
    System.out.println("获取数据："+buffer.get()+"===position："+buffer.position());
    System.out.println("获取数据："+buffer.get()+"===position："+buffer.position());
}
```

填充数据

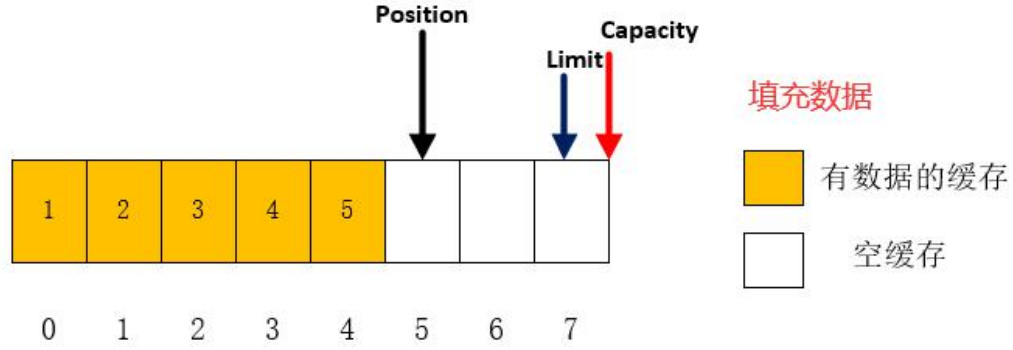
缓冲翻转

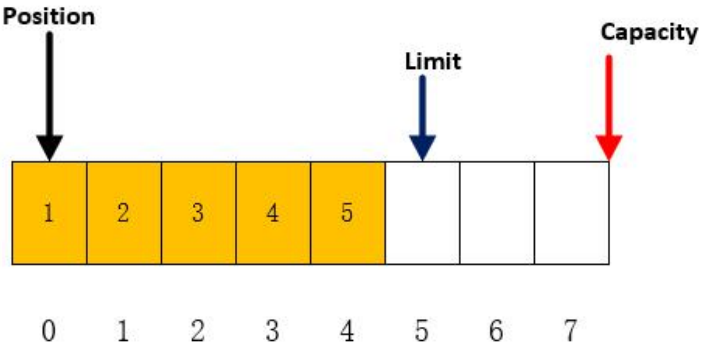
每获取一位，position就加一

```
"E:\Program Files\Java\jdk1.8.0_181\bin\java.exe"
position：0
获取数据：1===position：1
获取数据：2===position：2

Process finished with exit code 0
```

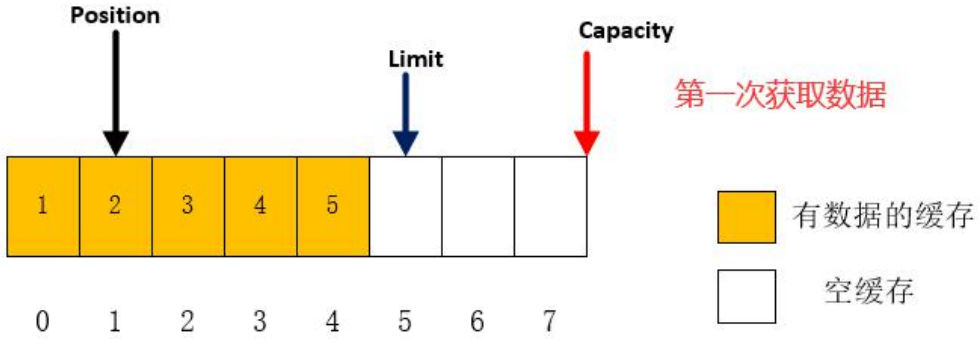

Position
Capacity
Limit
0 1 2 3 4 5 6 7

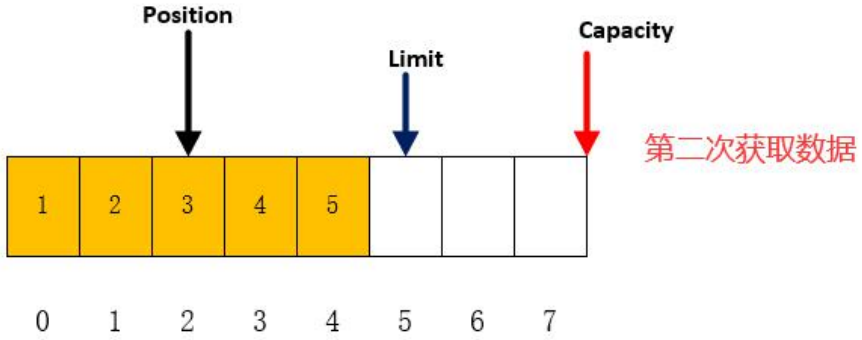
创建-初始状态


Position
Capacity
Limit
1 2 3 4 5
0 1 2 3 4 5 6 7
填充数据
有数据的缓存
空缓存


Position
Limit
Capacity
1 2 3 4 5
0 1 2 3 4 5 6 7

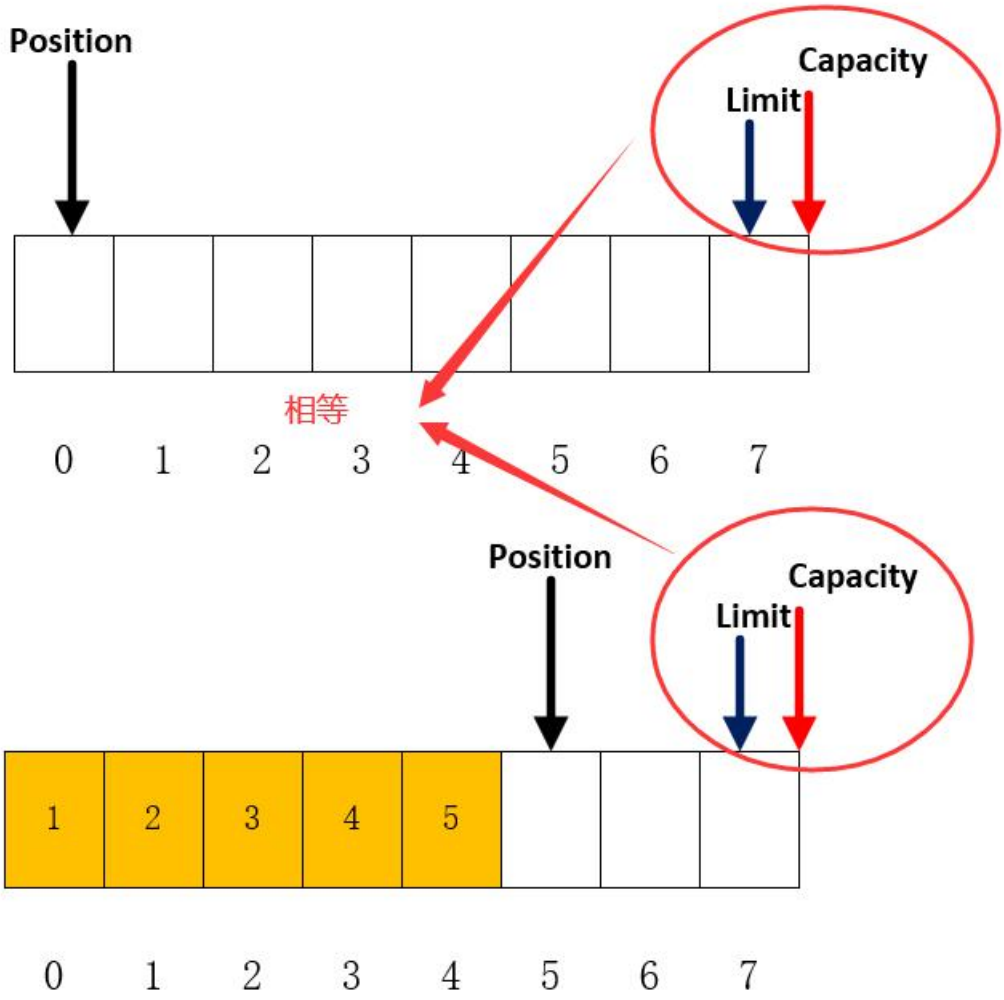
翻转



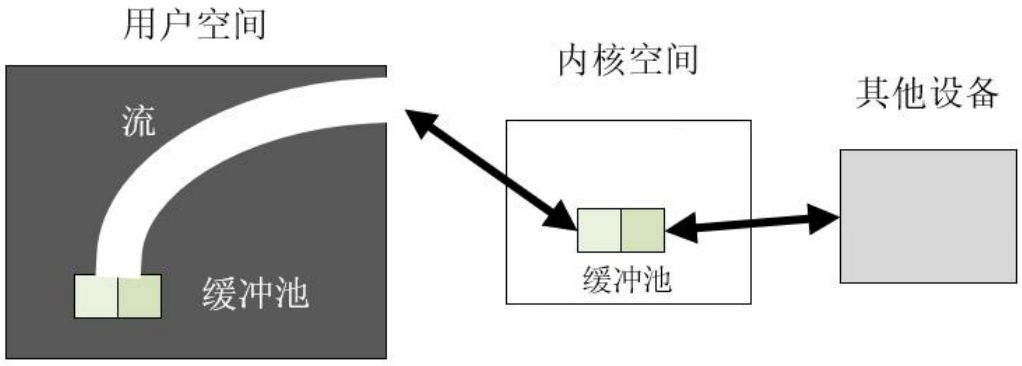

① Clear()：清空缓存；

- NIO 缓存默认情况下，limit 和 Capacity 是相等的；




5. 关于通道的升级

1) BIO 中 IO 流和缓存是直接交互的，如:



```java
public class ReadDemo {
    public static void main(String[] args) {
        try(FileInputStream fileInputStream = new FileInputStream( name: "E:
            int i = 0;
            byte[] buffer = new byte[1024*1024];// 占用内存空间
            while((i=fileInputStream.read(buffer))!=-1){
                System.out.println(new String (buffer, offset: 0,i));
            }
        }catch (Exception e){
            e.printStackTrace();
        }
```

定义缓存

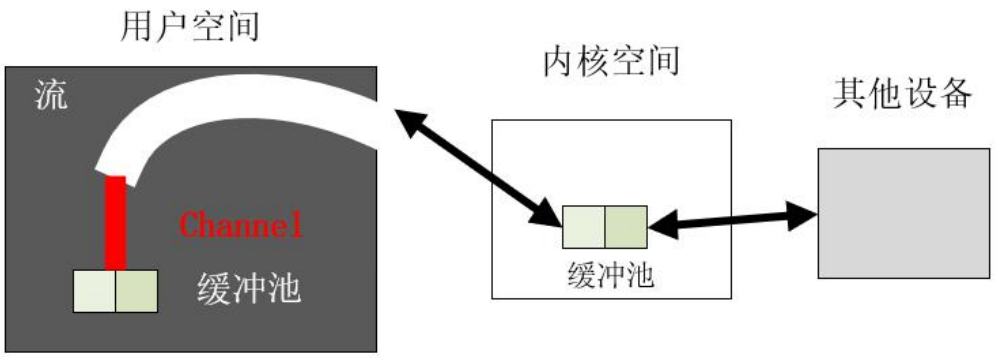
直接从流中读取

```java
    // I input  从磁盘读取数据到内存
    // O output 把内存中的数据写入到磁盘
    public static void main(String[] args) throws Exception{
        File file = new File( pathname: "E:/logo.png");
        FileInputStream fileInputStream = new FileInputStream(file);
        FileOutputStream fileOutputStream = new FileOutputStream( name: "E:/logo_
        int len = 0;
        byte[] buffer = new byte[1024];
        while((len=fileInputStream.read())!=-1){
            // 读取的数据可以保存在内存中
            // 也可以写出到磁盘
            //fileOutputStream.write(len); //写到磁盘（10000次的磁盘交互）
            fileOutputStream.write(buffer, off: 0,len); // 把InputStream的输入字节写
        }
        fileOutputStream.close();
        fileInputStream.close();

    }
}
```

定义缓存

直接将数据写道输出流中

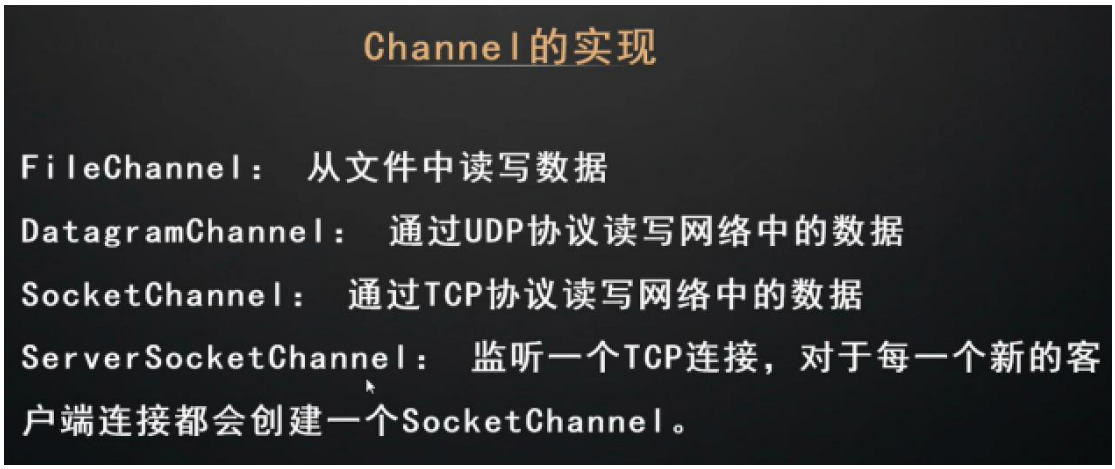
2) NIO 中，缓存没有默认，必须自定义。且流和缓冲池不可以直接对接，必须通过 channel 管道进行数据交换。



```
public static void main(String[] args) {
    try {
        FileInputStream fis = new FileInputStream(new File( pathnam
        FileOutputStream fos = new FileOutputStream(new File( pathna

        FileChannel fin = fis.getChannel();
        FileChannel fout = fos.getChannel();
        //初始化一个缓冲区
        ByteBuffer buffer  = ByteBuffer.allocate(1024);
        fin.read(buffer);//读取数据到缓冲区
        buffer.flip();  // 表示从读转化为写
        fout.write(buffer);
        buffer.clear();//重置缓冲区
    }catch (Exception e){
        e.printStackTrace();
    }
}
```

输入流
输出流
输入流的管道
输出流的管道
缓存
管道读
管道写

3) Channel 常用的文件文件实现类为 FileChannel，当然还有其他场景下的实现类（了解）：




6. Java 中 NIO（非网络）部分的读写过程，同 BIO 相差不大，可概述为以下内容

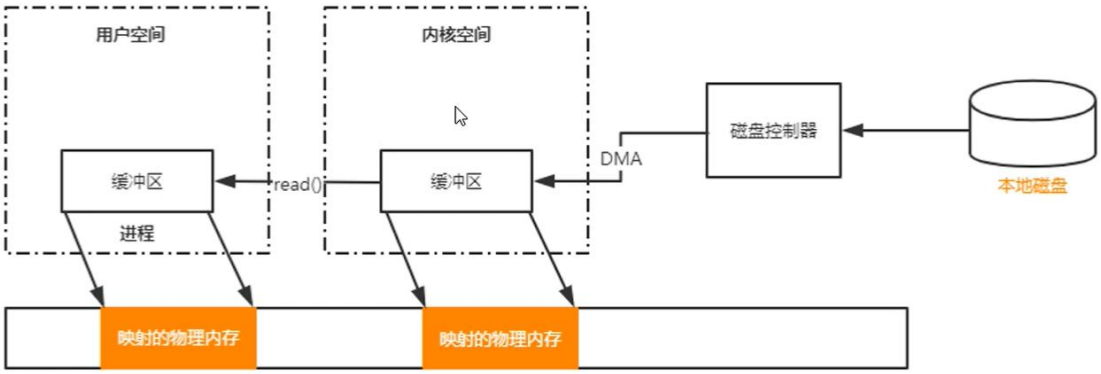
(1) 当用户空间调用内核 API，进行读取数据操作时：

① 用户空间会在内存（RAM）上分配一块内存，作为缓冲池（必须手动声明）（注：字符流是字节流的包装类，底层也是用字节流传输的）；

② 内核空间同时也会在内存上默认的分配一块内存作为缓冲池；

③ 用户空间需要一直处于阻塞等待状态，直至内核空间用缓冲池将用户空间的缓冲池填满或者将数据的剩余部分填充至用户空间的缓冲池中；

④ 然后用户空间获取到缓冲池中的流进行处理；




⑤ 注意点同上 BIO

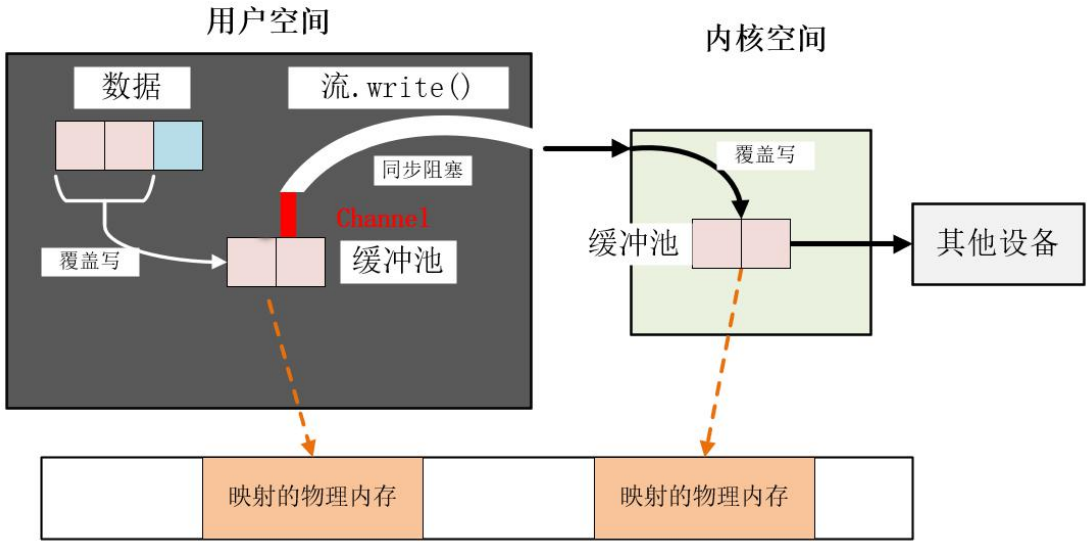
(2) 当用户空间调用内核 API，进行写入数据操作时：

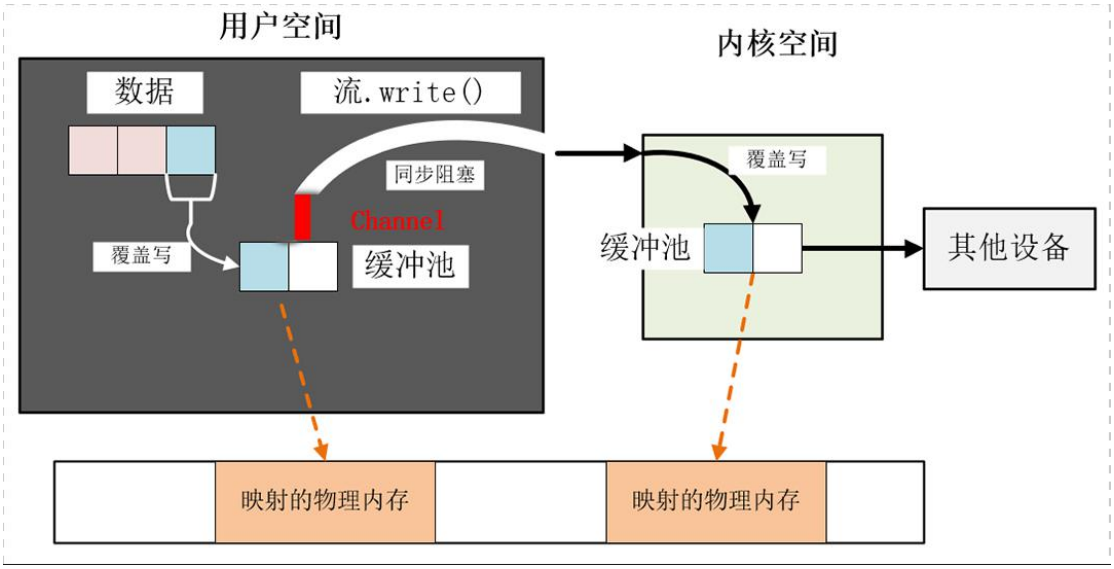
① 用户空间会在内存（RAM）上分配一块内存，作为缓冲池（必须手动声明）（注：字符流是字节流的包装类，底层也是用字节流传输的）；

② 内核空间同时也会在内存上默认的分配一块内存作为缓冲池；

③ 数据先写入到用户空间的缓冲池中，然后由 IO 管道调用内核空间 API，将用户空间缓冲池的数据写入到内核空间缓冲池中，再由内核空间写入到相应设备中；

④ 在用户空间缓冲池未被内核空间缓冲池消费完前，线程将一直处于阻塞等待状态；





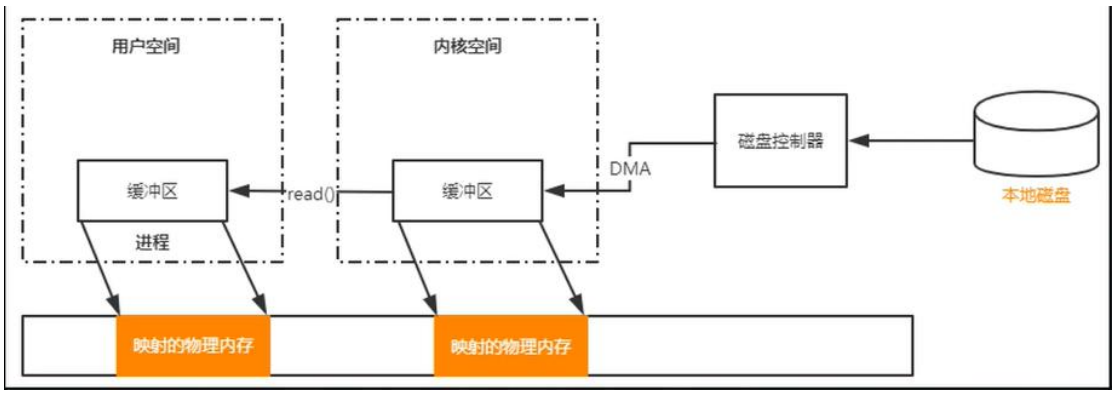

⑤ 当数据量过大时，用户空间缓冲池并不足以一次性将数据全部传输完毕，就需要 IO 流通道进行多次传输；

```
public static void main(String[] args) {
    try{
        FileInputStream fin = new FileInputStream( name: "E:/test.txt")
        FileOutputStream fout = new FileOutputStream( name: "E:/test_cp
        FileChannel fcin = fin.getChannel();
        FileChannel fcout = fout.getChannel();

        ByteBuffer byteBuffer = ByteBuffer.allocate(10);
        while(true){ ——————————→ 循环写
            System.out.println("come in");
            int r = fcin.read(byteBuffer);
            if(r == -1){break;}
            // 每读到 10个字节 ，写入缓冲区
            byteBuffer.flip();
            fcout.write(byteBuffer);
            byteBuffer.clear();//清空
        }
    }catch (Exception e){
        e.printStackTrace();
    }
}
```

7. Java 的 NIO（非网络），因为对 buffer 进行了规范化，所以速度相对于 BIO 有所提升，且相对于旧的 IO 的数组缓存，新的 NIO 缓存对数据的操作以及移动，更加方便；但是访问数据的方式仍然是同步阻塞的方式；
8. 此外，java 的 NIO 还提供了更为快捷的数据访问技术：内存映射（MMAP）技术

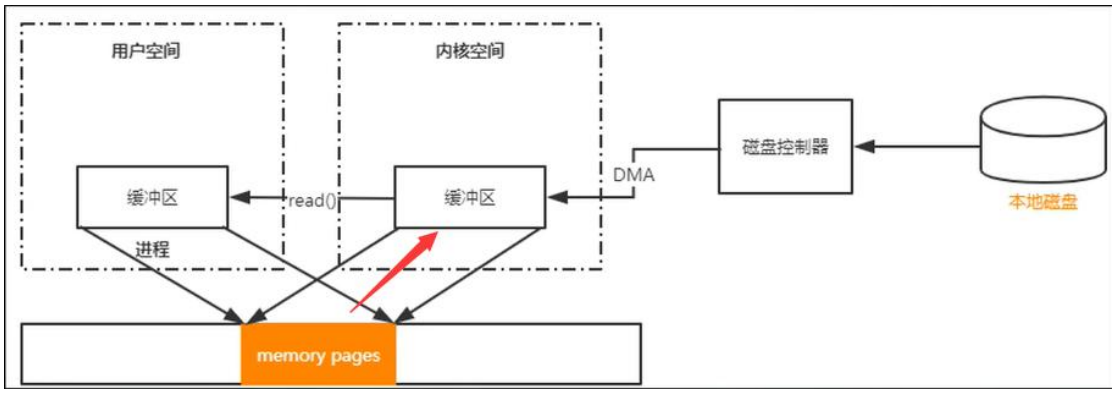
① 由上可知：内存中用户空间在对其他设备（如磁盘 ，网卡等）进行 IO 数据读写时，出于系统安全性考量，必须通过调用内核空间的 API 才能实现；并且需要在内存条上（RAM）分别为用户空间和内核空间分配一块内存空间

② 当进行读操作时，设备上的数据先要存储在内核空间的内存，然后再拷贝至用户空间的内存；当进行写操作时，用户空间的数据需要先拷贝至内核空间的内存上，然后才能到其他设备中（磁盘、网卡）




③ 这样显然增加了 CPU 的负担，降低了数据传输的效率；

④ Java 的 NIO 内存映射机制可以使，用户空间和内核空间共享内核空间的物理内存，减少了数据的拷贝次数，提高了数据的传输效率；（这里需要注意 MMAP 和零拷贝(零拷贝见 Kafka 笔记)的区别，下面会讲）




```
public static void main(String[] args) throws Exception{
    FileChannel inChannel = FileChannel.open(Paths.get( first: "E:/logo.png")
            , StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get( first: "E:/logo_cp.png")
            , StandardOpenOption.READ,StandardOpenOption.CREATE,StandardOpenOption.WRITE);

    MappedByteBuffer inMappedBuffer = inChannel.map(FileChannel.MapMode.READ_ONLY, position: 0,inChannel.size());
    MappedByteBuffer outMappedBuffer = outChannel.map(FileChannel.MapMode.READ_WRITE, position: 0,inChannel.size());

    byte[] bytes = new byte[inMappedBuffer.limit()];
    inMappedBuffer.get(bytes);
    outMappedBuffer.put(bytes);
    inChannel.close();
    outChannel.close();
}
```

MMAP技术

五、BIO、NIO 模型（网络）

1. 上述讲了 java 中 BIO、NIO 在读写非网络数据的原理，下面讲解 BIO、NIO 在读写网路数据的原理。
2. Linux、Unix 的 5 种 IO 网络模型

1) BIO / 同步阻塞 IO/ Blocking IO

- 原理同磁盘 BIO

- 说明：数据在网络发送/接受时，数据在用户空间、内核空间以及网卡中的缓冲池，采用同步阻塞等待的方式进行传输。

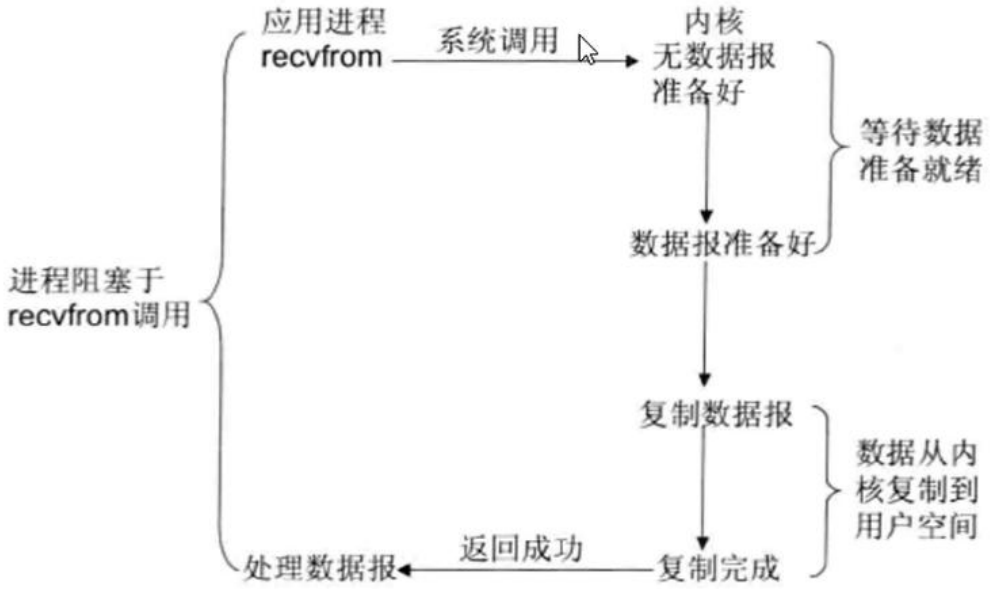
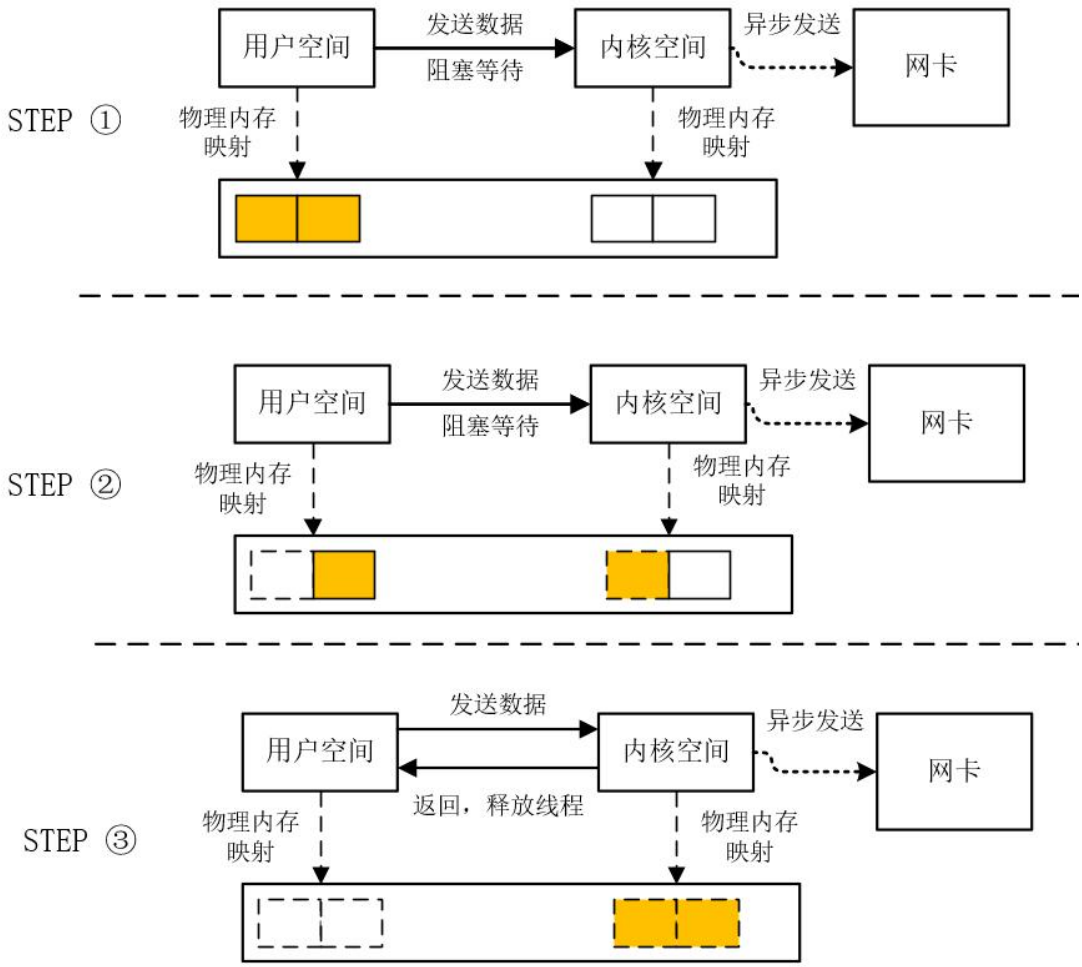
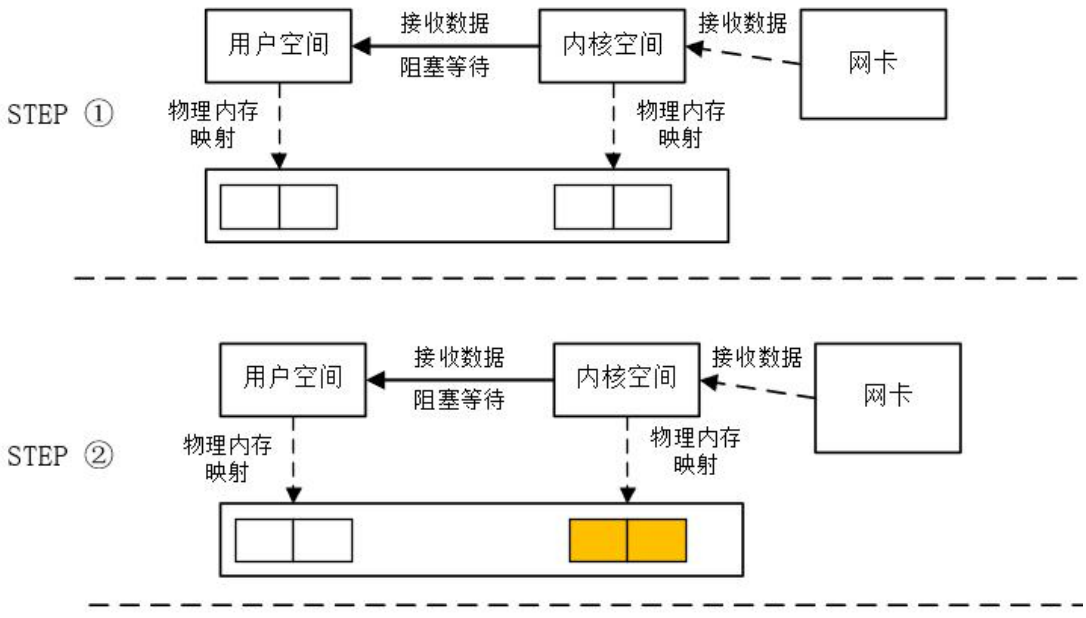


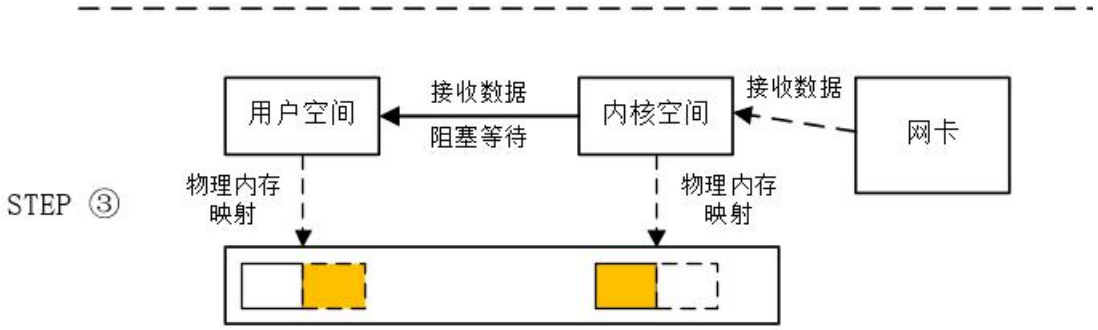

图 1-1　阻塞 I/O 模型

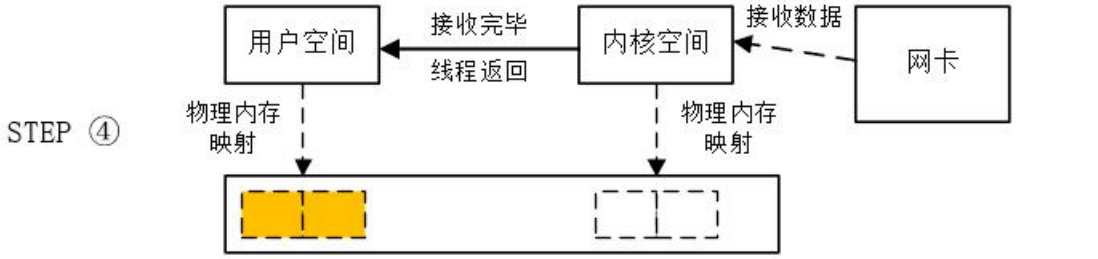
(1) 如用户空间的线程数据进行发送时，要阻塞等待内核缓冲区将用户空间缓冲区的数据全部消费完毕，直至线程返回。由内核缓冲区异步发送至网卡缓冲区，由网卡进行数据发送。




(2) 当用户空间的线程向内核空间读取数据时，线程将处于阻塞等待状态，直至内核空间将数据准备好且复制到用户空间的缓冲池中，线程携带用户缓冲池中的数据返回。

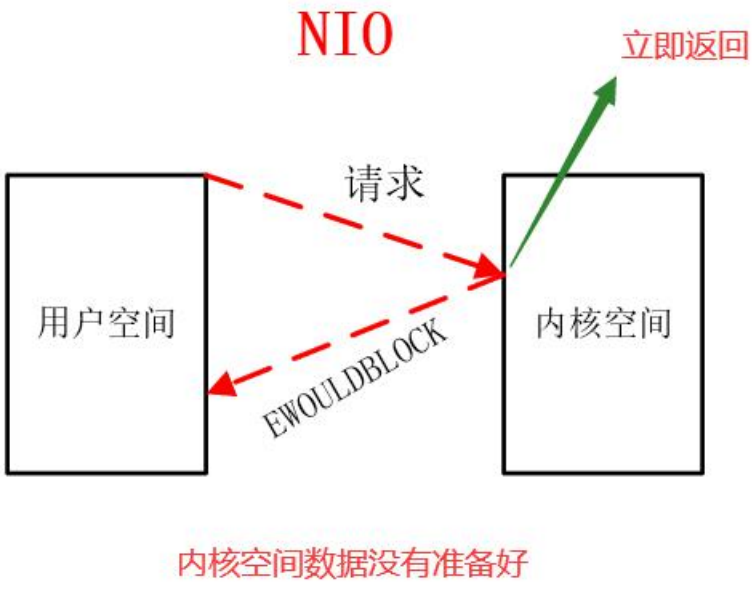


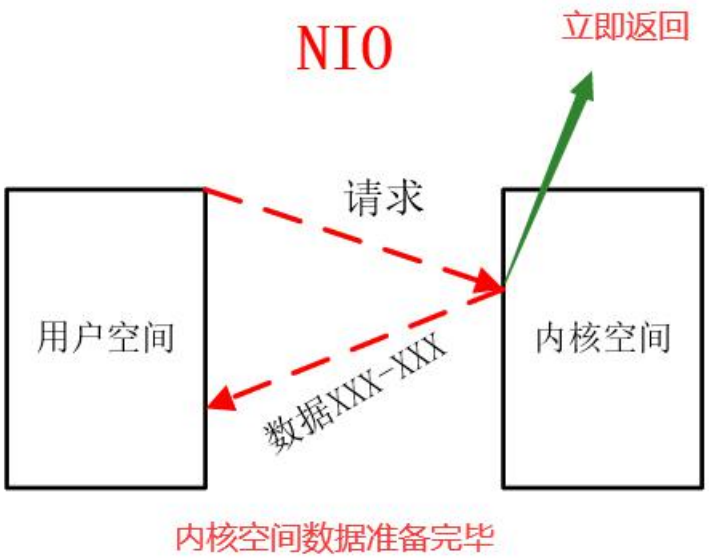



2) NIO/非阻塞 IO/Non-Blocking I/O

- 在 java 中，NIO 叫 New IO，区别于 BIO（Blocking IO）,表示对 BIO 的升级
- 在 Linux 或者 Unix 系统的网络模型中，NIO 又叫 Non-Blocking I/O，区别于 BIO（Blocking IO）,表示另一种网络 IO 方式；
- 在网络模型中，NIO 叫非阻塞 IO，这里的非阻塞是相对于用户线程来说的，即不会阻塞用户空间的线程；只要有线程传过来，马上就会将线程返回。如果数据没有准备好，返回编码 EWOULDBLOCK（E 是 error,WOULD BLOCK 是可能会被阻塞的意思），如果数据已经准备好，则返回数据；






- 所以在网络模型 NIO 中，用户空间调用内核空间的读、写 API 不再受到阻塞，而是会直接返回。当内核空间未执行完读、写操作，用户空间进行询问时，返回的是 EWOULDBLOCK 编码；当内核空间已执行完读、写操作，用户空间进行询问时，返回的是对应地数据。
- 这就需要用户空间不断地去询问内核空间，被调用的 API 是否已经执行完毕；
- 这么做的好处是，释放了内核空间的网络线程，一定程度上增加了内核空间连接网络请求的数量；

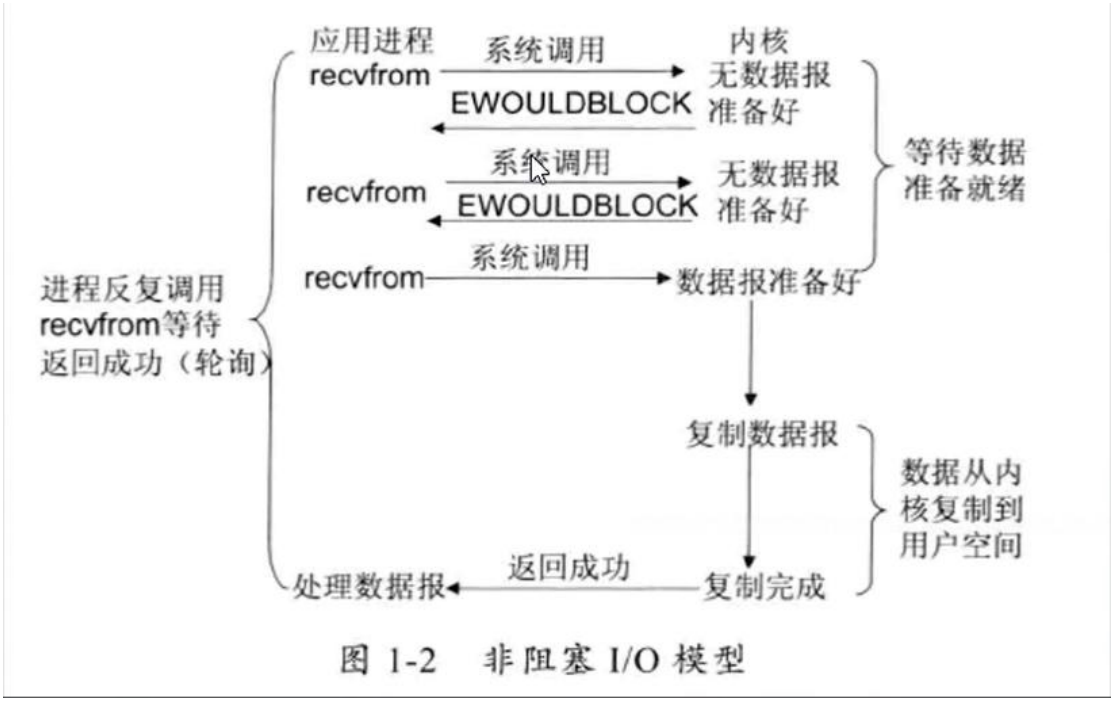



图 1-2　非阻塞 I/O 模型

3) IO 复用

- IO 复用包括：select、poll、epoll、AIO
- 关于 select、poll、epoll 的讲解，见 IO 复用专栏
- 因为 AIO 技术不成熟，所以这里不进行讲解
- Java 的 NIO 的网络部分，在 1.5 之前使用的是 select/poll 网络模型，在 1.6 之后使用的

是 epoll 模型；

3. 上述讲了 linux、Unix 系统的 5 中网络模型，下面讲解 JDK 的 BIO、NIO 网络部分的情况
4. Java(JDK)中的 BIO，网络部分，原理同磁盘部分，只不过数据源从磁盘换成了网卡
5. Java(JDK)中的 NIO,网络部分，融合了 IO 复用进去，相对来说变得复杂，下面将详细讲解

1) 上面说了，java(JDK)非网络部分的 NIO 的新特性，java(JDK)网络部分的 NIO 新特性如下：

- 增加了通道和缓冲区的概念，进行数据的读写时必须基于通道和缓冲区（和 JDK 非网络的是一样的）
- 非阻塞特性，java(JDK)的 NIO 的非网络部分是阻塞的，前面讲过；java(JDK)的 NIO 网络部分，因为整合了 IO 复用模型，所以是非阻塞的
- 选择器，java(JDK)的 NIO 网络部分独有的，用于注册网络事件的监听对象
- 同时提供了字符集编码等解决方案（网络和非网络部分共有）

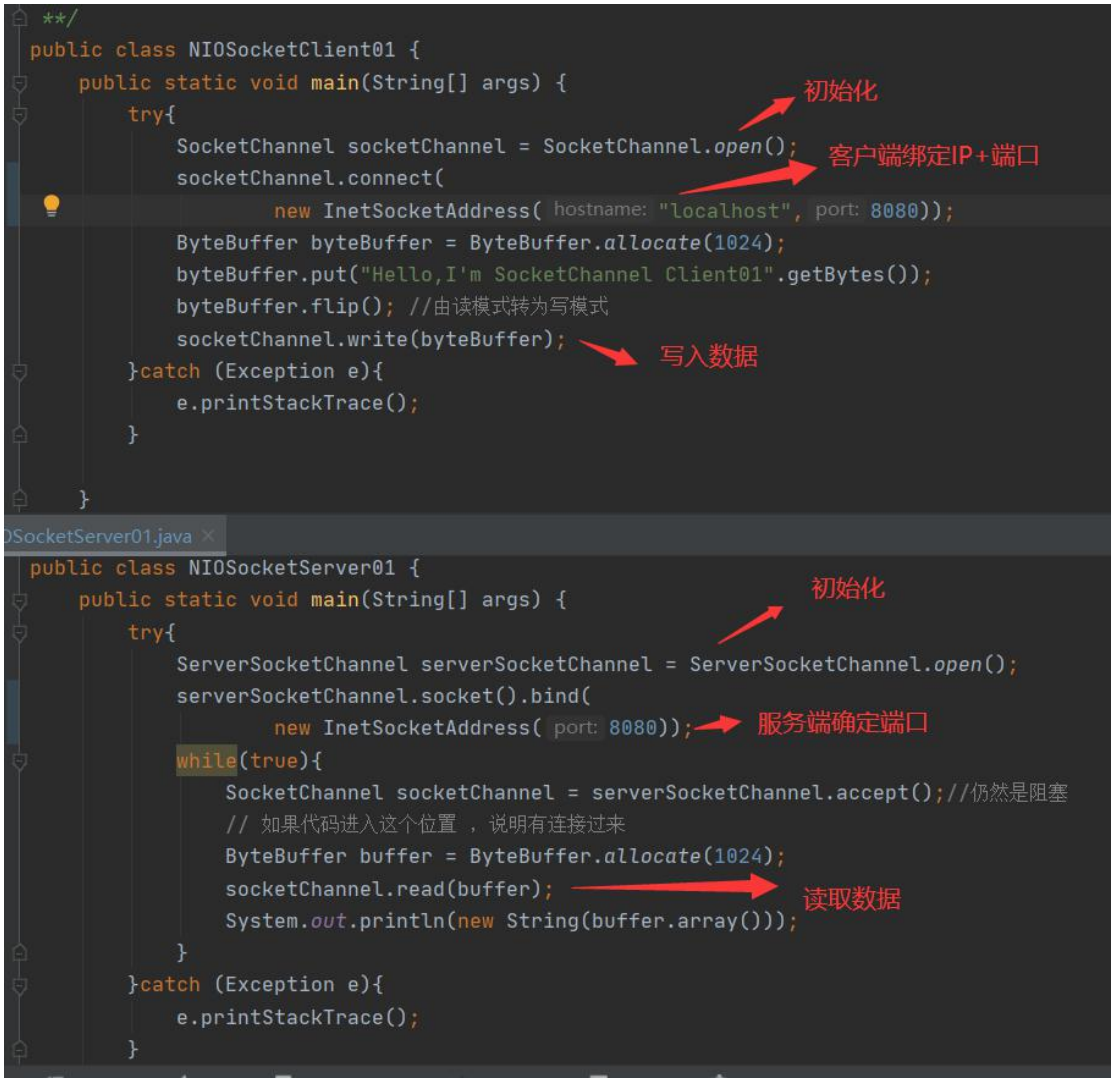
6. 关于通道和缓冲区的相关内容见 NIO 非网络部分，下面将 NIO 网络部分独有的
7. Java(JDK)网络部分核心组件：

- 通道
- 缓冲区
- 选择器

8. Java(JDK) 的 NIO 网络部分具体的实现方案

(1) 基本的实现方案

- Java(JDK)的 NIO 的网络部分，是使用 SocketChannel 和 ServerSocketChannel 两个通道，绑定 IP 和端口，通过读写缓存之间的数据，来实现交互；读数据就是获取另一个通道的数据，写数据就是传输给另一个通道数据；




- 需要注意的是，JDK 的 NIO 网络部分，默认情况下是阻塞的，如果要使用非阻塞的需要专门声明，但是非阻塞模式下一定要注意处理好是否读取到对方数据的异常，因为非阻塞模式下，即使读取不到也会返回

```java
public class NIOSocketClient02 {
    public static void main(String[] args) {
        try{
            SocketChannel socketChannel = SocketChannel.open();
            socketChannel.configureBlocking(Boolean.FALSE);
            // 在非阻塞模式下，这段代码并不一定是等到链接建立之后再往下执行
            socketChannel.connect(new InetSocketAddress( hostname: "localhost", port: 8
            if(socketChannel.isConnectionPending()){...}
            ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
            byteBuffer.put("Hello,I'm SocketChannel Client01".getBytes());
            byteBuffer.flip(); //由读模式转为写模式
            socketChannel.write(byteBuffer);

            // 读取服务端返回的数据
            byteBuffer.clear();
            int r =  socketChannel.read(byteBuffer);// 非阻塞模式，这里不阻塞
            if(r>0){
                System.out.println("收到服务端的消息："+new String(byteBuffer.array()))
            }else{
                System.out.println("服务端的数据还未返回");
            }
            System.in.read();
```

声明非阻塞

处理读取数据异常

```java
public class NIOSocketServer02 {
    public static void main(String[] args) {
        try{
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.configureBlocking(Boolean.FALSE);
            serverSocketChannel.socket().bind(new InetSocketAddress( port: 8080));
            while(true){
                //此处不再阻塞
                SocketChannel socketChannel = serverSocketChannel.accept();
                if(socketChannel!=null){
                    // 如果代码进入这个位置 ，说明有连接过来
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    socketChannel.read(buffer);
                    System.out.println(new String(buffer.array()));

                    //再把消息写回到客户端
                    Thread.sleep( millis: 10000);
                    //buffer.flip();
                    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                    byteBuffer.put("Hello I'm Server Channel".getBytes());
                    byteBuffer.flip();
                    socketChannel.write(byteBuffer);
                    //socketChannel.write(buffer);
                }else{
                    Thread.sleep( millis: 1000);
                    System.out.println("没有客户端连接过来");
                }
            }
        }catch (Exception e){
```

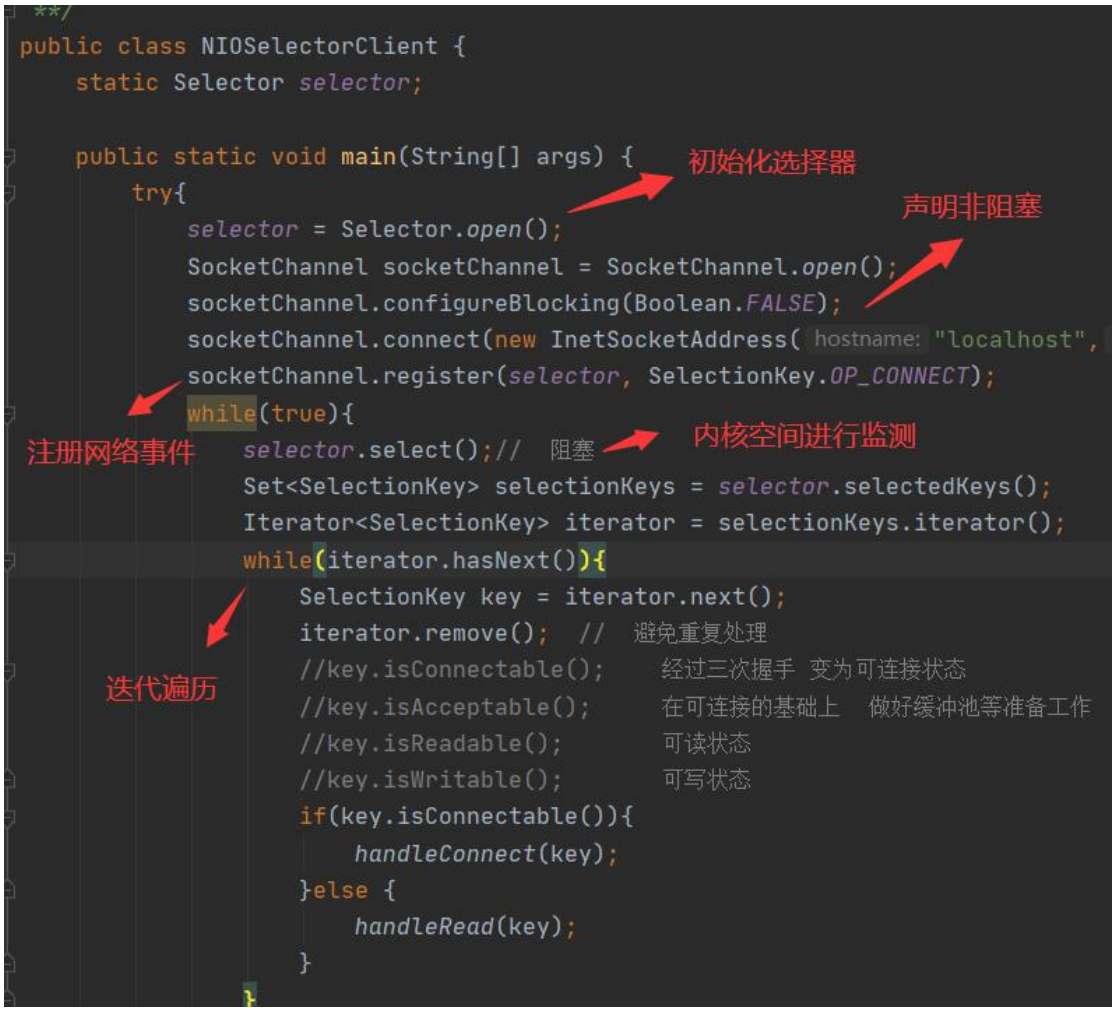
声明非阻塞

处理异常

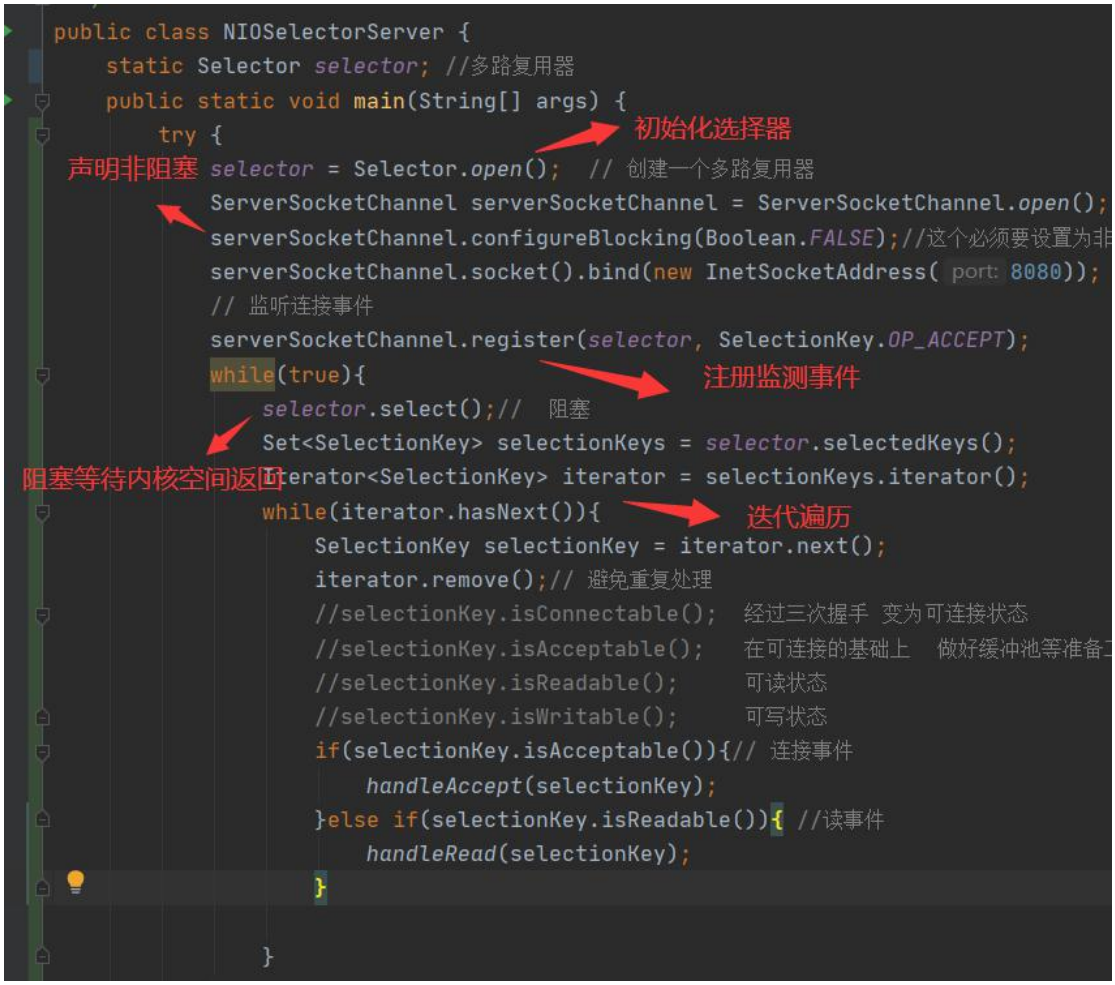
(2) 整合 IO 复用的实现方案

如果网络连接比较多，就考虑使用 IO 复用提高效率，原理上面讲过，下面讲实践
如果要使用 IO 复用的话，就需要用到选择器（Selector），将需要监测的网络事件注册到选择器中，Selector 会执行 select 方法，调用内核空间的 IO 复用模型来监测需要注意的网络事件；
需要注意的是：

- Selector 的 select 方法是阻塞的，需要一直等到内核空间将就绪的网络事件返回，才能进行网络事件数据的操作；
- Java 通常是对这些网络事件进行迭代遍历操作，（详情见代码）
- 总之，selector 就是使用 IO 复用，用来注册网络事件的实现类




```
**/
public class NIOSelectorClient {
    static Selector selector;

    public static void main(String[] args) {
        try{
            selector = Selector.open();
            SocketChannel socketChannel = SocketChannel.open();
            socketChannel.configureBlocking(Boolean.FALSE);
            socketChannel.connect(new InetSocketAddress( hostname: "localhost",
            socketChannel.register(selector, SelectionKey.OP_CONNECT);
            while(true){
                selector.select();//  阻塞
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while(iterator.hasNext()){
                    SelectionKey key = iterator.next();
                    iterator.remove();  //  避免重复处理
                    //key.isConnectable();     经过三次握手 变为可连接状态
                    //key.isAcceptable();      在可连接的基础上  做好缓冲池等准备工作
                    //key.isReadable();        可读状态
                    //key.isWritable();        可写状态
                    if(key.isConnectable()){
                        handleConnect(key);
                    }else {
                        handleRead(key);
                    }
                }
```




```java
public class NIOSelectorServer {
    static Selector selector; //多路复用器
    public static void main(String[] args) {
        try {
            selector = Selector.open();  // 创建一个多路复用器
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.configureBlocking(Boolean.FALSE);//这个必须要设置为非
            serverSocketChannel.socket().bind(new InetSocketAddress( port: 8080));
            // 监听连接事件
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
            while(true){
                selector.select();//  阻塞
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while(iterator.hasNext()){
                    SelectionKey selectionKey = iterator.next();
                    iterator.remove();// 避免重复处理
                    //selectionKey.isConnectable();  经过三次握手 变为可连接状态
                    //selectionKey.isAcceptable();   在可连接的基础上  做好缓冲池等准备
                    //selectionKey.isReadable();     可读状态
                    //selectionKey.isWritable();     可写状态
                    if(selectionKey.isAcceptable()){// 连接事件
                        handleAccept(selectionKey);
                    }else if(selectionKey.isReadable()){ //读事件
                        handleRead(selectionKey);
                    }

                }
```