

Java IO

原作者github: <https://github.com/CyC2018/CS-Notes>

PDF制作github: <https://github.com/sjsdfg/CS-Notes-PDF>

一、概览

Java 的 I/O 大概可以分成以下几类:

- 磁盘操作: File
- 字节操作: InputStream 和 OutputStream
- 字符操作: Reader 和 Writer
- 对象操作: Serializable
- 网络操作: Socket
- 新的输入/输出: NIO

二、磁盘操作

File 类可以用于表示文件和目录的信息,但是它不表示文件的内容。

递归地列出一个目录下所有文件:

```
public static void listAllFiles(File dir) {
    if (dir == null || !dir.exists()) {
        return;
    }
    if (dir.isFile()) {
        System.out.println(dir.getName());
        return;
    }
    for (File file : dir.listFiles()) {
        listAllFiles(file);
    }
}
```

从 Java7 开始, 可以使用 Paths 和 Files 代替 File。

三、字节操作

实现文件复制

```
public static void copyFile(String src, String dist) throws IOException {
    FileInputStream in = new FileInputStream(src);
    FileOutputStream out = new FileOutputStream(dist);
}
```

```

byte[] buffer = new byte[20 * 1024];
int cnt;

// read() 最多读取 buffer.length 个字节
// 返回的是实际读取的个数
// 返回 -1 的时候表示读到 eof, 即文件尾
while ((cnt = in.read(buffer, 0, buffer.length)) != -1) {
    out.write(buffer, 0, cnt);
}

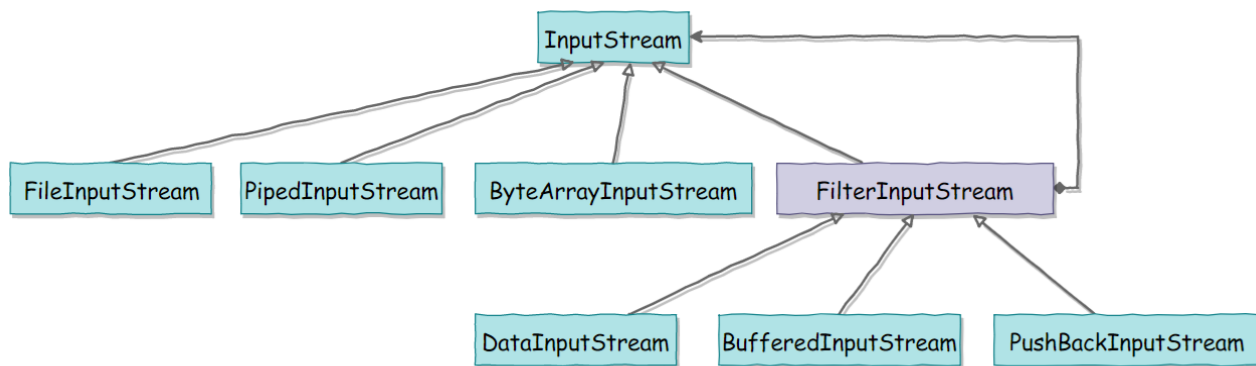
in.close();
out.close();
}

```

装饰者模式

Java I/O 使用了装饰者模式来实现。以 `InputStream` 为例，

- `InputStream` 是抽象组件；
- `FileInputStream` 是 `InputStream` 的子类，属于具体组件，提供了字节流的输入操作；
- `FilterInputStream` 属于抽象装饰者，装饰者用于装饰组件，为组件提供额外的功能。例如 `BufferedInputStream` 为 `FileInputStream` 提供缓存的功能。



@CyC2018

实例化一个具有缓存功能的字节流对象时，只需要在 `FileInputStream` 对象上再套一层 `BufferedInputStream` 对象即可。

```

FileInputStream fileInputStream = new FileInputStream(filePath);
BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);

```

`DataInputStream` 装饰者提供了对更多数据类型进行输入的操作，比如 `int`、`double` 等基本类型。

四、字符操作

编码与解码

编码就是把字符转换为字节，而解码是把字节重新组合成字符。

如果编码和解码过程使用不同的编码方式那么就出现了乱码。

- GBK 编码中，中文字符占 2 个字节，英文字符占 1 个字节；
- UTF-8 编码中，中文字符占 3 个字节，英文字符占 1 个字节；
- UTF-16be 编码中，中文字符和英文字符都占 2 个字节。

UTF-16be 中的 be 指的是 Big Endian，也就是大端。相应地也有 UTF-16le，le 指的是 Little Endian，也就是小端。

Java 的内存编码使用双字节编码 UTF-16be，这不是指 Java 只支持这一种编码方式，而是说 char 这种类型使用 UTF-16be 进行编码。char 类型占 16 位，也就是两个字节，Java 使用这种双字节编码是为了让一个中文或者一个英文都能使用一个 char 来存储。

String 的编码方式

String 可以看成一个字符序列，可以指定一个编码方式将它编码为字节序列，也可以指定一个编码方式将一个字节序列解码为 String。

```
String str1 = "中文";
byte[] bytes = str1.getBytes("UTF-8");
String str2 = new String(bytes, "UTF-8");
System.out.println(str2);
```

在调用无参数 getBytes() 方法时，默认的编码方式不是 UTF-16be。双字节编码的好处是可以使用一个 char 存储中文和英文，而将 String 转为 bytes[] 字节数组就不再需要这个好处，因此也就不再需要双字节编码。getBytes() 的默认编码方式与平台有关，一般为 UTF-8。

```
byte[] bytes = str1.getBytes();
```

Reader 与 Writer

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符。但是在程序中操作的通常是字符形式的数据，因此需要提供对字符进行操作的方法。

- InputStreamReader 实现从字节流解码成字符流；
- OutputStreamWriter 实现字符流编码成为字节流。

实现逐行输出文本文件的内容

```
public static void readFileContent(String filePath) throws IOException {

    FileReader fileReader = new FileReader(filePath);
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    String line;
    while ((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
    }
}
```

```
// 装饰者模式使得 BufferedReader 组合了一个 Reader 对象
// 在调用 BufferedReader 的 close() 方法时会去调用 Reader 的 close() 方法
// 因此只要一个 close() 调用即可
bufferedReader.close();
}
```

五、对象操作

序列化

序列化就是将一个对象转换成字节序列，方便存储和传输。

- 序列化：ObjectOutputStream.writeObject()
- 反序列化：ObjectInputStream.readObject()

不会对静态变量进行序列化，因为序列化只是保存对象的状态，静态变量属于类的状态。

Serializable

序列化的类需要实现 Serializable 接口，它只是一个标准，没有任何方法需要实现，但是如果不去实现它的话而进行序列化，会抛出异常。

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

    A a1 = new A(123, "abc");
    String objectFile = "file/a1";

    ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
    FileOutputStream(objectFile));
    objectOutputStream.writeObject(a1);
    objectOutputStream.close();

    ObjectInputStream objectInputStream = new ObjectInputStream(new FileInputStream(objectFile));
    A a2 = (A) objectInputStream.readObject();
    objectInputStream.close();
    System.out.println(a2);
}

private static class A implements Serializable {

    private int x;
    private String y;

    A(int x, String y) {
        this.x = x;
        this.y = y;
    }
}
```

```
@Override
public String toString() {
    return "x = " + x + " " + "y = " + y;
}
}
```

transient

transient 关键字可以使一些属性不会被序列化。

ArrayList 中存储数据的数组 elementData 是用 transient 修饰的，因为这个数组是动态扩展的，并不是所有的空间都被使用，因此就不需要所有的内容都被序列化。通过重写序列化和反序列化方法，使得可以只序列化数组中有内容的那部分数据。

```
private transient Object[] elementData;
```

六、网络操作

Java 中的网络支持：

- InetAddress：用于表示网络上的硬件资源，即 IP 地址；
- URL：统一资源定位符；
- Sockets：使用 TCP 协议实现网络通信；
- Datagram：使用 UDP 协议实现网络通信。

InetAddress

没有公有的构造函数，只能通过静态方法来创建实例。

```
InetAddress.getByName(String host);
InetAddress.getByAddress(byte[] address);
```

URL

可以直接从 URL 中读取字节流数据。

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

    URL url = new URL("http://www.baidu.com");

    /* 字节流 */
    InputStream is = url.openStream();

    /* 字符流 */
    InputStreamReader isr = new InputStreamReader(is, "utf-8");
```

```

/* 提供缓存功能 */
BufferedReader br = new BufferedReader(isr);

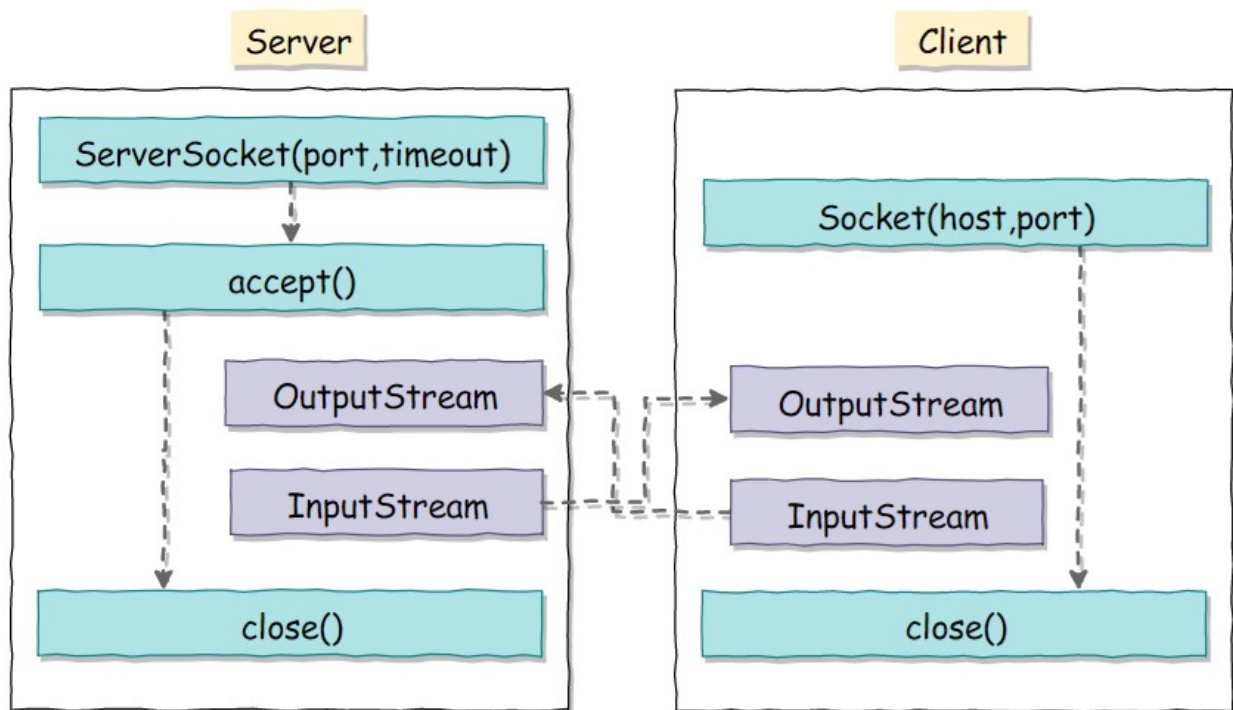
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}

br.close();
}

```

Sockets

- ServerSocket: 服务器端类
- Socket: 客户端类
- 服务器和客户端通过 InputStream 和 OutputStream 进行输入输出。



@CyC2018

Datagram

- DatagramSocket: 通信类
- DatagramPacket: 数据包类

七、NIO

新的输入/输出 (NIO) 库是在 JDK 1.4 中引入的，弥补了原来的 I/O 的不足，提供了高速的、面向块的 I/O。

流与块

I/O 与 NIO 最重要的区别是数据打包和传输的方式，I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

面向流的 I/O 一次处理一个字节数据：一个输入流产生一个字节数据，一个输出流消费一个字节数据。为流式数据创建过滤器非常容易，链接几个过滤器，以便每个过滤器只负责复杂处理机制的一部分。不利的一面是，面向流的 I/O 通常相当慢。

面向块的 I/O 一次处理一个数据块，按块处理数据比按流处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有优雅性和简单性。

I/O 包和 NIO 已经很好地集成了，`java.io.*` 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如，`java.io.*` 包中的一些类包含以块的形式读写数据的方法，这使得即使在面向流的系统中，处理速度也会更快。

通道与缓冲区

1. 通道

通道 Channel 是对原 I/O 包中的流的模拟，可以通过它读取和写入数据。

通道与流的不同之处在于，流只能在一个方向上移动(一个流必须是 `InputStream` 或者 `OutputStream` 的子类)，而通道是双向的，可以用于读、写或者同时用于读写。

通道包括以下类型：

- `FileChannel`：从文件中读写数据；
- `DatagramChannel`：通过 UDP 读写网络中数据；
- `SocketChannel`：通过 TCP 读写网络中数据；
- `ServerSocketChannel`：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 `SocketChannel`。

2. 缓冲区

发送给一个通道的所有数据都必须首先放到缓冲区中，同样地，从通道中读取的任何数据都要先读到缓冲区中。也就是说，不会直接对通道进行读写数据，而是要先经过缓冲区。

缓冲区实质上是一个数组，但它不仅仅是一个数组。缓冲区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。

缓冲区包括以下类型：

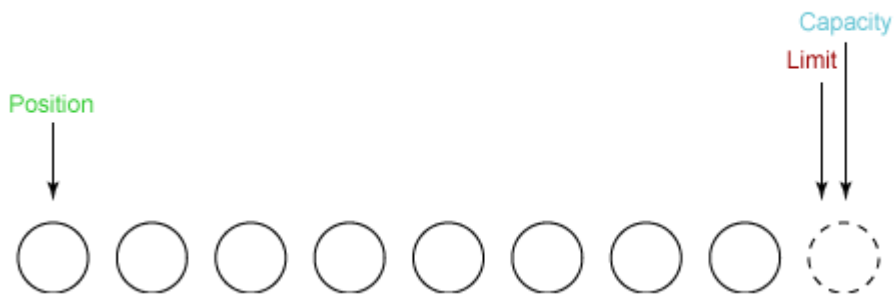
- `ByteBuffer`
- `CharBuffer`
- `ShortBuffer`
- `IntBuffer`
- `LongBuffer`
- `FloatBuffer`
- `DoubleBuffer`

缓冲区状态变量

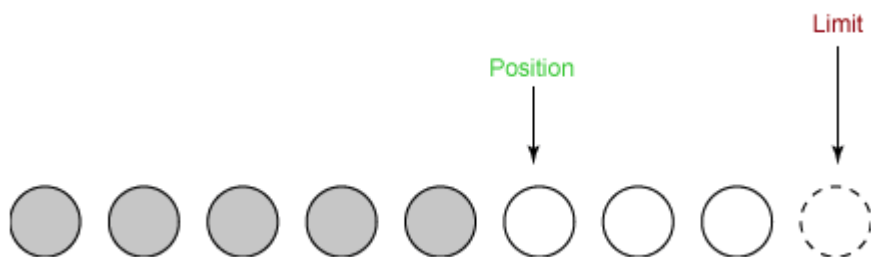
- capacity: 最大容量;
- position: 当前已经读写的字节数;
- limit: 还可以读写的字节数。

状态变量的改变过程举例:

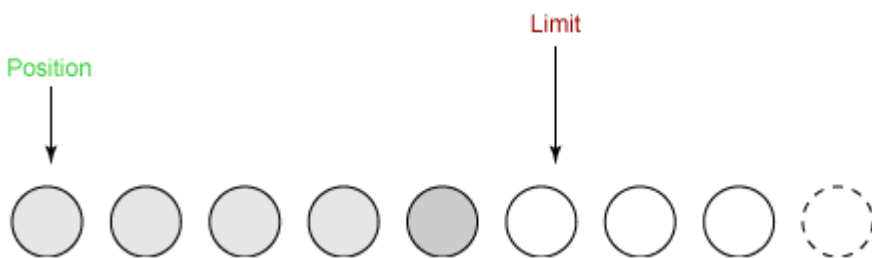
① 新建一个大小为 8 个字节的缓冲区, 此时 position 为 0, 而 limit = capacity = 8。capacity 变量不会改变, 下面的讨论会忽略它。



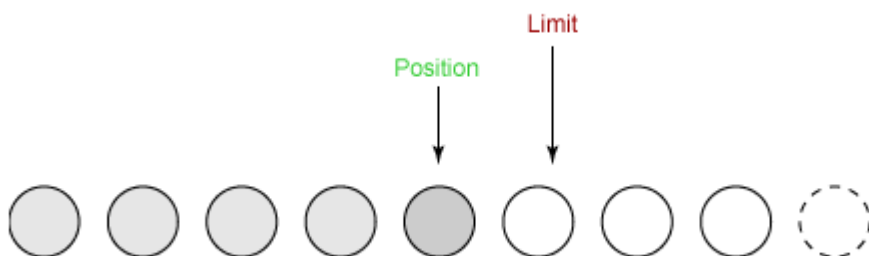
② 从输入通道中读取 5 个字节数据写入缓冲区中, 此时 position 为 5, limit 保持不变。



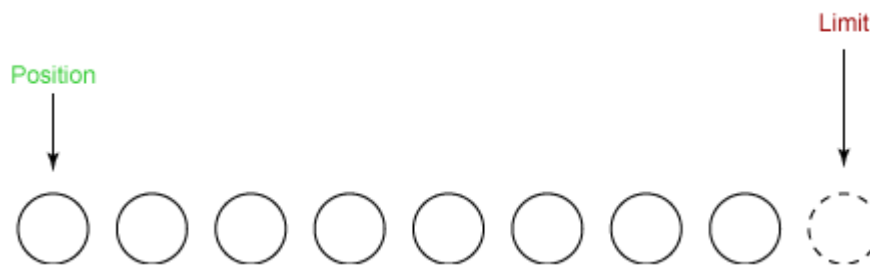
③ 在将缓冲区的数据写到输出通道之前, 需要先调用 flip() 方法, 这个方法将 limit 设置为当前 position, 并将 position 设置为 0。



④ 从缓冲区中取 4 个字节到输出缓冲中, 此时 position 设为 4。



⑤ 最后需要调用 clear() 方法来清空缓冲区, 此时 position 和 limit 都被设置为最初位置。



文件 NIO 实例

以下展示了使用 NIO 快速复制文件的实例：

```
public static void fastCopy(String src, String dist) throws IOException {

    /* 获得源文件的输入字节流 */
    FileInputStream fin = new FileInputStream(src);

    /* 获取输入字节流的文件通道 */
    FileChannel fcin = fin.getChannel();

    /* 获取目标文件的输出字节流 */
    FileOutputStream fout = new FileOutputStream(dist);

    /* 获取输出字节流的文件通道 */
    FileChannel fcout = fout.getChannel();

    /* 为缓冲区分配 1024 个字节 */
    ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

    while (true) {

        /* 从输入通道中读取数据到缓冲区中 */
        int r = fcin.read(buffer);

        /* read() 返回 -1 表示 EOF */
        if (r == -1) {
            break;
        }

        /* 切换读写 */
        buffer.flip();

        /* 把缓冲区的内容写入输出文件中 */
        fcout.write(buffer);

        /* 清空缓冲区 */
        buffer.clear();
    }
}
```

选择器

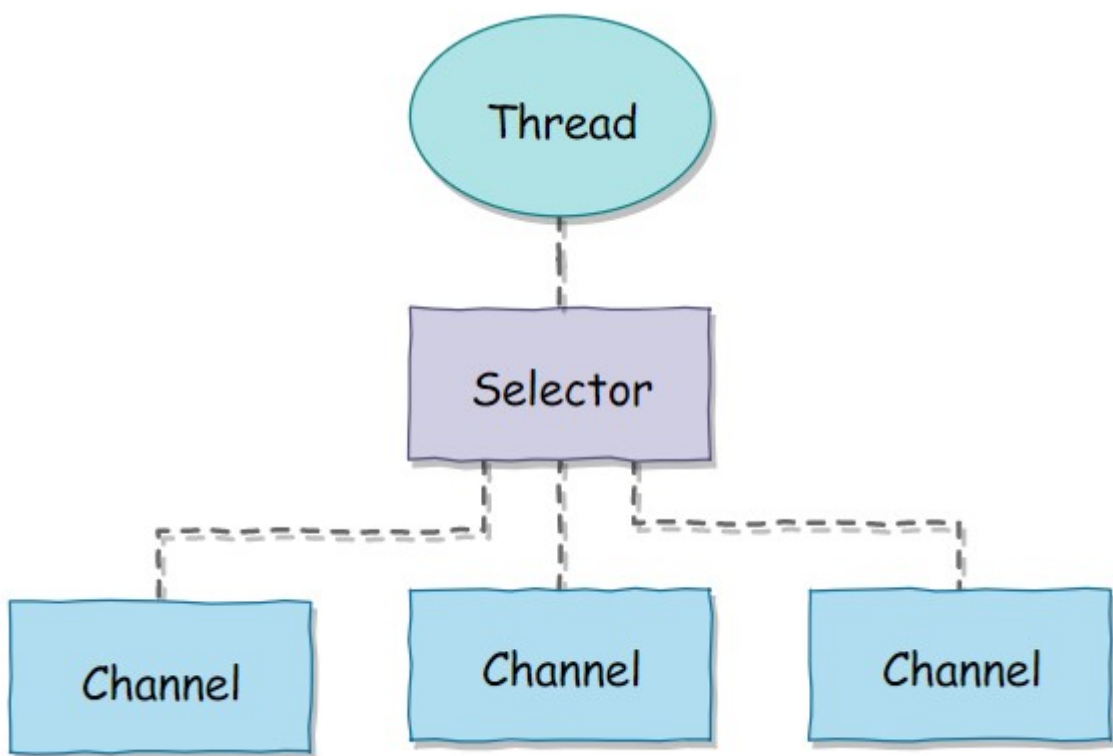
NIO 常常被叫做非阻塞 IO，主要是因为 NIO 在网络通信中的非阻塞特性被广泛使用。

NIO 实现了 IO 多路复用中的 Reactor 模型，一个线程 Thread 使用一个选择器 Selector 通过轮询的方式去监听多个通道 Channel 上的事件，从而让一个线程就可以处理多个事件。

通过配置监听的通道 Channel 为非阻塞，那么当 Channel 上的 IO 事件还未到达时，就不会进入阻塞状态一直等待，而是继续轮询其它 Channel，找到 IO 事件已经到达的 Channel 执行。

因为创建和切换线程的开销很大，因此使用一个线程来处理多个事件而不是一个线程处理一个事件，对于 IO 密集型的应用具有很好性能。

应该注意的是，只有套接字 Channel 才能配置为非阻塞，而 FileChannel 不能，为 FileChannel 配置非阻塞也没有意义。



@CyC2018

1. 创建选择器

```
Selector selector = Selector.open();
```

2. 将通道注册到选择器上

```
ServerSocketChannel ssChannel = ServerSocketChannel.open();  
ssChannel.configureBlocking(false);  
ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

通道必须配置为非阻塞模式，否则使用选择器就没有任何意义了，因为如果通道在某个事件上被阻塞，那么服务器就不能响应其它事件，必须等待这个事件处理完毕才能去处理其它事件，显然这和选择器的作用背道而驰。

在将通道注册到选择器上时，还需要指定要注册的具体事件，主要有以下几类：

- `SelectionKey.OP_CONNECT`
- `SelectionKey.OP_ACCEPT`
- `SelectionKey.OP_READ`
- `SelectionKey.OP_WRITE`

它们在 `SelectionKey` 的定义如下：

```
public static final int OP_READ = 1 << 0;
public static final int OP_WRITE = 1 << 2;
public static final int OP_CONNECT = 1 << 3;
public static final int OP_ACCEPT = 1 << 4;
```

可以看出每个事件可以被当成一个位域，从而组成事件集整数。例如：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

3. 监听事件

```
int num = selector.select();
```

使用 `select()` 来监听到达的事件，它会一直阻塞直到有至少一个事件到达。

4. 获取到达的事件

```
Set<SelectionKey> keys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = keys.iterator();
while (keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if (key.isAcceptable()) {
        // ...
    } else if (key.isReadable()) {
        // ...
    }
    keyIterator.remove();
}
```

5. 事件循环

因为一次 `select()` 调用不能处理完所有的事件，并且服务器端有可能需要一直监听事件，因此服务器端处理事件的代码一般会放在一个死循环内。

```
while (true) {
    int num = selector.select();
    Set<SelectionKey> keys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = keys.iterator();
    while (keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        if (key.isAcceptable()) {
```

```

        // ...
    } else if (key.isReadable()) {
        // ...
    }
    keyIterator.remove();
}
}

```

套接字 NIO 实例

```

public class NIOServer {

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

        Selector selector = Selector.open();

        ServerSocketChannel ssChannel = ServerSocketChannel.open();
        ssChannel.configureBlocking(false);
        ssChannel.register(selector, SelectionKey.OP_ACCEPT);

        ServerSocket serverSocket = ssChannel.socket();
        InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8888);
        serverSocket.bind(address);

        while (true) {

            selector.select();
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> keyIterator = keys.iterator();

            while (keyIterator.hasNext()) {

                SelectionKey key = keyIterator.next();

                if (key.isAcceptable()) {

                    ServerSocketChannel ssChannel1 = (ServerSocketChannel) key.channel();

                    // 服务器会为每个新连接创建一个 SocketChannel
                    SocketChannel sChannel = ssChannel1.accept();
                    sChannel.configureBlocking(false);

                    // 这个新连接主要用于从客户端读取数据
                    sChannel.register(selector, SelectionKey.OP_READ);

                } else if (key.isReadable()) {

                    SocketChannel sChannel = (SocketChannel) key.channel();
                    System.out.println(readDataFromSocketChannel(sChannel));
                    sChannel.close();

                }

            }

        }

    }

}

```

```

        keyIterator.remove();
    }
}

private static String readDataFromSocketChannel(SocketChannel sChannel) throws IOException {

    ByteBuffer buffer = ByteBuffer.allocate(1024);
    StringBuilder data = new StringBuilder();

    while (true) {

        buffer.clear();
        int n = sChannel.read(buffer);
        if (n == -1) {
            break;
        }
        buffer.flip();
        int limit = buffer.limit();
        char[] dst = new char[limit];
        for (int i = 0; i < limit; i++) {
            dst[i] = (char) buffer.get(i);
        }
        data.append(dst);
        buffer.clear();
    }
    return data.toString();
}
}

```

```

public class NIOClient {

    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("127.0.0.1", 8888);
        OutputStream out = socket.getOutputStream();
        String s = "hello world";
        out.write(s.getBytes());
        out.close();
    }
}

```

内存映射文件

内存映射文件 I/O 是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的 I/O 快得多。

向内存映射文件写入可能是危险的，只是改变数组的单个元素这样的简单操作，就可能会直接修改磁盘上的文件。修改数据与将数据保存到磁盘是没有分开的。

下面代码行将文件的前 1024 个字节映射到内存中，map() 方法返回一个 MappedByteBuffer，它是 ByteBuffer 的子类。因此，可以像使用其他任何 ByteBuffer 一样使用新映射的缓冲区，操作系统会在需要时负责执行映射。

```
MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, 1024);
```

对比

NIO 与普通 I/O 的区别主要有以下两点：

- NIO 是非阻塞的；
- NIO 面向块，I/O 面向流。

八、参考资料

- Eckel B, 埃克尔, 昊鹏, 等. Java 编程思想 [M]. 机械工业出版社, 2002.
- [IBM: NIO 入门](#)
- [Java NIO Tutorial](#)
- [Java NIO 浅析](#)
- [IBM: 深入分析 Java I/O 的工作机制](#)
- [IBM: 深入分析 Java 中的中文编码问题](#)
- [IBM: Java 序列化的高级认识](#)
- [NIO 与传统 IO 的区别](#)
- [Decorator Design Pattern](#)
- [Socket Multicast](#)

github: <https://github.com/sjsdfg/CS-Notes-PDF>