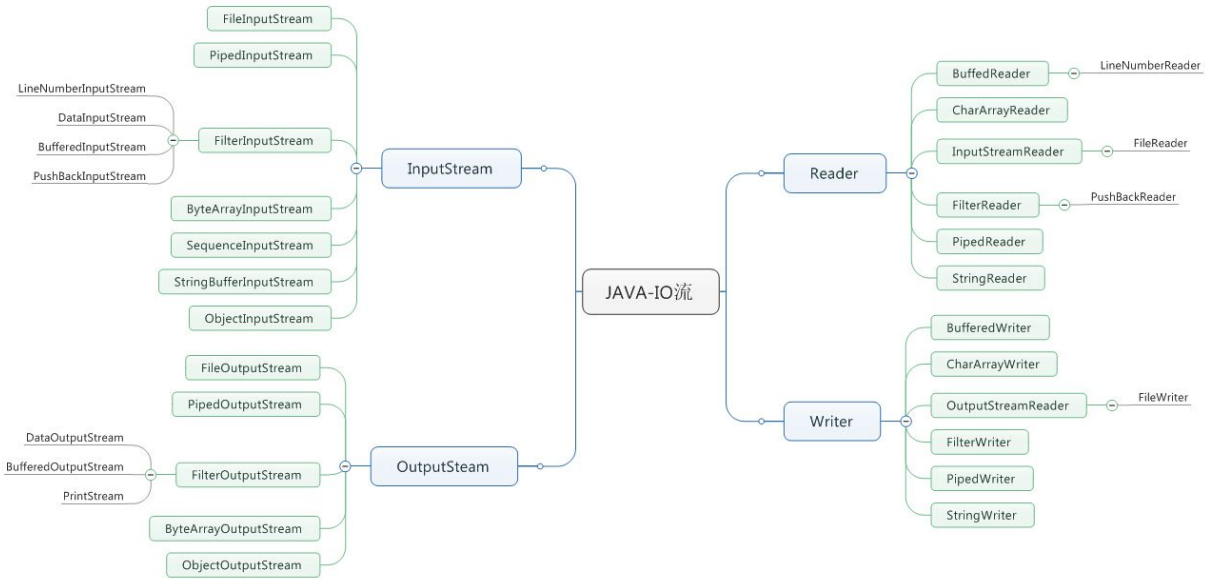
io流输入输出



理解java的io流

io流的分类

流向

- 输入流：只能从中读取数据，而不能向其中写入数据
- 输出流：只能向其写入数据，而不能从中读取数据

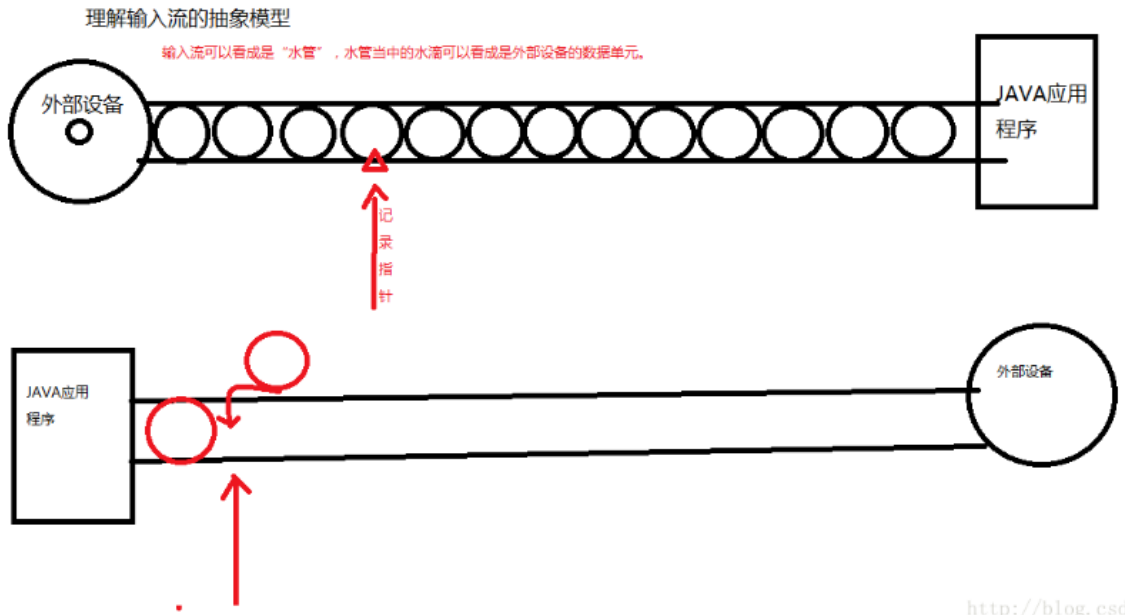
数据单元

- 字符流：操作的数据单元是16位的字符
- 字节流：操作的数据单元是8位的字节

流的角色

- 节点流：从一个特定的IO设备读/写数据的流，程序直接连到实际的数据源，和实际的输入/输出节点连接。
- 处理流：对一个已存在的流进行连接或封装，通过封装后的流来实现数据读/写功能。

流的概念模型



对流的概念模型而言，字节流和字符流没有明显的区别。他们都是将输入/输出设备抽象成一个水管。在这个水管里，隐式地利用指针指出下次应该读取的数据位置，或者应该写入数据的位置。然后移动指针。

优点

- 性能的提高：主要以增加缓冲的方式来提高输入/输出效率
- 操作的便捷：处理流提供了一系列便捷的方法来一次输入/输出大批量的内容，同时可以“嫁接”在任何已存在流的基础之上

字节流和字符流

基本的输入流（所有输入流的抽象基类）

InputStream

- int read() :从输入流中读取单个字节，返回所读取的字节数据
- int read(byte[] b)：从输入流中最多读取b.length长度的字节数据，并将其存储在数组b中，返回实际读取的字节数
- int read(byte[] b, int off, int len)：从输入流中读入len长度的字节，存在数组b中，而且是从off位置开始。返回实际读取的字节数。

Reader

- int read()
- int read(char[])
- int read(char[]bb, int off, int len)

OutputStream

- void writer(int c): 将指定的字节/字符输出到输出流中，其中c既可以代表字节，也可以代表字符。
- void writer(byte[]/char[] b)：将字节数组、字符数组中的数据输出到指定的输出流中。
- void writer(byta[]/char[] b, int off, int len)：将字节数组/字符数组从off位置开始写入，总的写入长度的为len。

Writer

- void writer(String str)
- void writer(String str, int off, int len)

基本使用

基本输入

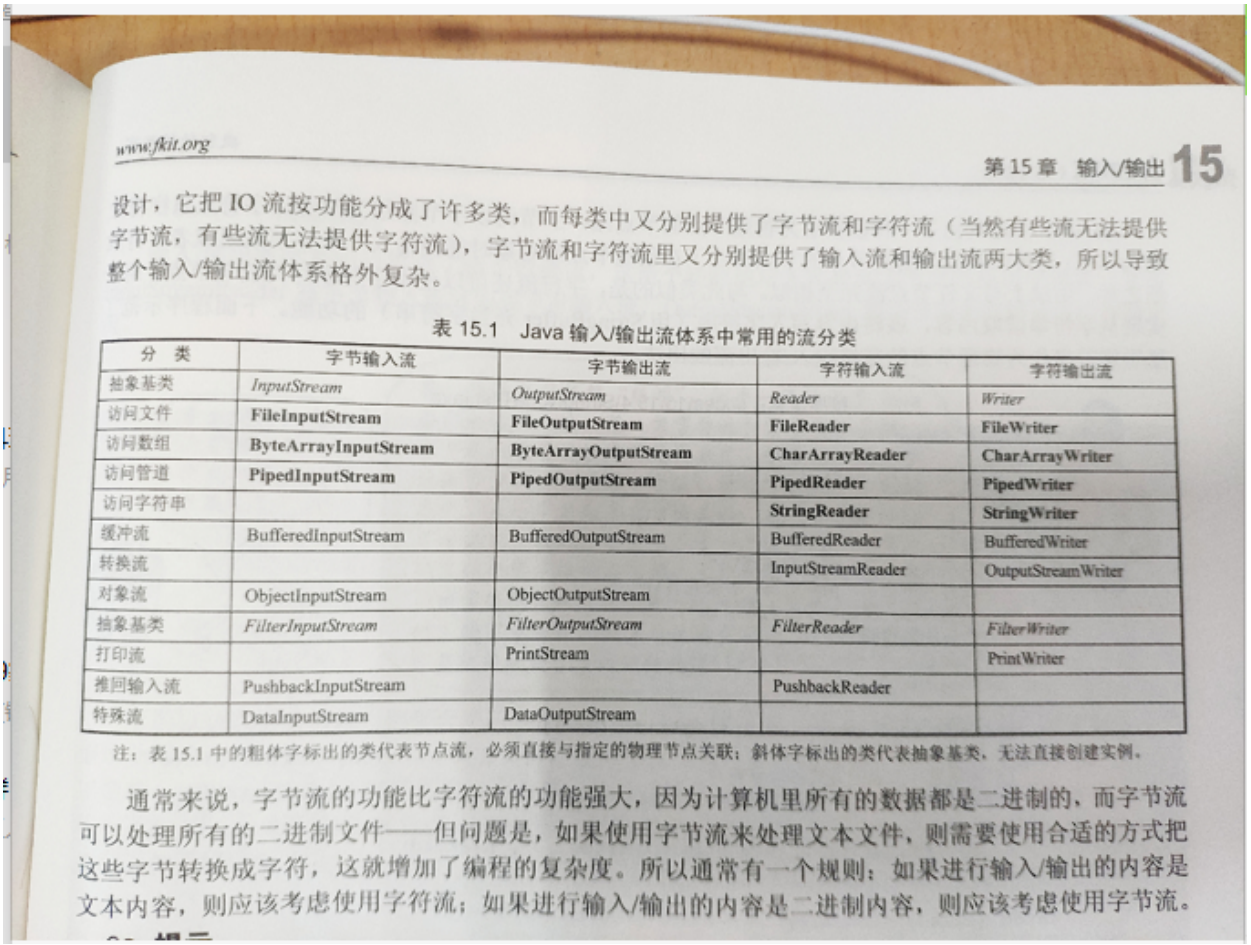
```
public static void main(String[] args) throws IOException {
        FileInputStream fileInputStream = new FileInputStream("src/Main.java");
        //创建竹筒用来存储
        byte[] bytes = new byte[1024];
        //用于保存实际读取的字节数
        int hasRead = 0;
        //循环取水
        try {
            while((hasRead = fileInputStream.read(bytes))>0){
                System.out.println(new String(bytes,0,hasRead));
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            fileInputStream.close();
        }
```

```
    }
```

基本输出

```
public static void main(String[] args) throws FileNotFoundException , IOException {
    FileInputStream fileInputStream = new FileInputStream("src/Main.java");
    FileOutputStream fileOutputStream = new FileOutputStream("newFile.txt");

    byte[] bytes = new byte[32];
    int hasRead = 0;
    while((hasRead = fileInputStream.read(bytes))>0){
        fileOutputStream.write(bytes,0,hasRead);
    }
}
```

输入/输出流体系



www.fkit.org

设计，它把 IO 流按功能分成了许多类，而每类中又分别提供了字节流和字符流（当然有些流无法提供字节流，有些流无法提供字符流），字节流和字符流里又分别提供了输入流和输出流两大类，所以导致整个输入/输出流体系格外复杂。

表 15.1 Java 输入/输出流体系中常用的流分类

分 类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	*InputStream*	*OutputStream*	*Reader*	*Writer*
访问文件	**FileInputStream**	**FileOutputStream**	**FileReader**	**FileWriter**
访问数组	**ByteArrayInputStream**	**ByteArrayOutputStream**	**CharArrayReader**	**CharArrayWriter**
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		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

注：表 15.1 中的粗体字标出的类代表节点流，必须直接与指定的物理节点关联；斜体字标出的类代表抽象基类，无法直接创建实例。

通常来说，字节流的功能比字符流的功能强大，因为计算机里所有的数据都是二进制的，而字节流可以处理所有的二进制文件——但问题是，如果使用字节流来处理文本文件，则需要使用合适的方式把这些字节转换成字符，这就增加了编程的复杂度。所以通常有一个规则：如果进行输入/输出的内容是文本内容，则应该考虑使用字符流；如果进行输入/输出的内容是二进制内容，则应该考虑使用字节流。

转换流

转换流的目的是用于实现将字节流转换成字符流。主要是InputStreamReader，和OutputStreamReader两个方法。

```
    public static void main(String[] args) throws IOException {
        InputStreamReader inputStreamReader = new InputStreamReader(System.in);
        //bufferedReader的readLine方法可以一行一行读取，适合前端的文本输入
        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
        String line = null;
        while((line = bufferedReader.readLine())!=null){
            if(line.equals("exit"))
                System.exit(1);
            System.out.println(line);
        }
    }
```

推回输入流

推回输入流，提供了三个方法

- unread(byte[]/char[] buf)：将一个字节/字符数组内容推回到缓冲区，从而允许重复读取刚刚的内容
- unread(byte[]/char[] buf,int off, int len)：指定开始位置，长度len，然后重复读取
- unread(int a)：将一个字节/字符推回到缓冲区李，从而允许重复读取。

调用PushbackInputStream/PushbackReader的read方法，会先从缓冲区进行调用，然后再从输入流中读取

```
public static void main(String[] args) {
        try{
            PushbackReader pushbackReader = new PushbackReader(new
FileReader("newFile.txt"),128);
            {
                char[] chars = new char[64];
                String lastContent = "";
                int hasRead =0 ;
                while((hasRead = pushbackReader.read(chars))>0){
                    String content = new String(chars,0,hasRead);
                    int targetIndex = 0;
                    if((targetIndex = (lastContent+content).indexOf("test test1"))>0){
                        pushbackReader.unread((lastContent+content).toCharArray());
                        if(targetIndex > 64) {
                            chars = new char[targetIndex];
                        }
                        pushbackReader.read(chars,0,targetIndex);
                        System.out.println(new String(chars,0,targetIndex));
                        System.exit(0);
                    }else {
                        System.out.println(lastContent);
                        lastContent = content;
                    }
                }

            }
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }
```

对象序列化

序列化机制允许将实现序列化的java对象转换成字节序列，这些字节序列可以保存在磁盘上，或通过网络传输，以后可以重新恢复成原来的对象。序列化机制就是使得对象可以脱离程序的运行而独立存在。

主要是通过实现Serializable的接口来进行标志，这只是一个标记接口，说明它是可序列化的，无其他方法需要实现。

基本代码实现

```
//实体类
public class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

//序列化
public class WriteObject {
    public static void main(String[] args) throws IOException {
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream("Object.txt"));
        Person gumpgan = new Person("gumpgan", 5);
        objectOutputStream.writeObject(gumpgan);

    }
}
//反序列化
```

```
public class ReadObject {
    public static void main(String[] args) throws ClassNotFoundException {
        try {
            ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream("Object.txt"));
            Person o = (Person) objectInputStream.readObject();
            System.out.println(o.getName()+ o.getAge());

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

对象引用的序列化

如果在A类中持有B类的引用，只有B类是可序列化的前提，我们才可以对A类进行序列化操作，否则会报错。

若两个A类对象a,b，同时持有同一个B类对象c，我们将a,b,c序列化后，如果进行反序列化操作，可能会出现，三个对象c。这个时候，我们是通过一种特殊的序列化算法来解决这个问题。

- 所有保存在磁盘中的对象都会有一个序列化对象
- 当程序视图序列化一个对象时，程序先检查该对象是否已经被序列化过，只有该对象被序列化过，系统才会将该对象转换成字节序列并输出
- 如果某个对象已经被序列化过了，系统只是直接输出一个序列化编号，而不是再次重新序列化该对象。

自定义序列化

1. 通过transient关键字来修饰变量，那么java在序列化时便不会理会该实例变量。同样的在反序列化时也会得不到该变量的值。

2. 通过重写实体类的 writeObject方法和readObject方法来对部分变量在传送过程中进行变形（加密）

```
public class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    private void writeObject(java.io.ObjectOutputStream out) throws IOException {
        out.writeObject(new StringBuffer(name).reverse());
        out.writeInt(age);
    }

    private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException {
        this.name = ((StringBuffer)in.readObject()).reverse().toString();
        this.age = in.readInt();
    }
```

3. 通过writeReplace()方法对序列化机制进行调用，可以直接进行对象的替换。

```
//在实体类中重写该方法
private Object writePlace() throws ObjectStreamException{
    ArrayList<Object> list = new ArrayList<Object>();
    list.add(name) ;
    list.add(age);
    return list;
}

//调用实现
public class ReplaceTest {
    public static void main(String[] args) {

        try (
                ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("Object.txt"));
                ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("Object.txt"));

        ) {
            Person gan = new Person("GAN", 23);
            Person sun = new Person("SUN",21);
            oos.writeObject(gan);
            oos.writeObject(sun);
            ArrayList A;
            while ((A=(ArrayList) ois.readObject())!= null){
                System.out.println(A);
            }
           /* ArrayList a = (ArrayList) ois.readObject();
            ArrayList b = (ArrayList) ois.readObject();
            System.out.println(a);
            System.out.println(b);*/
```

```
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }
```

NIO

Channel和Buffer是NIO中的两个核心对象。channel是对传统的输入/输出系统的模拟，在新IO系统中所有的数据都需要通过通道传输。与传统的inputstream，outputstream相比，最大的区别是提供了map方法，可以直接一块数据映射到内存中，可以理解为是面向块的输入输出。

buffer可以理解为一个容器，本质是一个数组，发送到Channel中所有的对象都必须先放在buffer中，而从channel中读取数据也需要先将channel中数据放到到buffer中再去读取。

Buffer

基本参数

- 容量capacity：缓冲区所能容纳的最大数据量。
- 界限limit：第一个不允许被读出或者写入的位置
- 位置position:用于指明下一个可以被读出或者被写入的位置。

重要方法

- put():往buffer中注入数据，位置可以理解为position指向的地方
- flip()：是在装入数据结束后使用，会将limit指向position的位置，position会变成0，这样就把之后允许读取的位置限定在之前put的范围内
- clear()：不是清空数据，仅仅将position指向0，limit指向capacity。

基本使用

```
public class BufferTest {
    public static void main(String[] args) {
        //创建buffer
        CharBuffer cb = CharBuffer.allocate(8);
        System.out.println(cb.capacity() + " "+ cb.position()+ cb.limit());
        //注入数据
        cb.put('a');
        cb.put('b');
        cb.put('c');
        System.out.println(cb.position());
        //调用flip方法
        cb.flip();
```

```
        System.out.println(cb.capacity() + " "+cb.position()+ cb.limit());
        //取出元素
        System.out.println(cb.get());
        System.out.println(cb.position());
        //调用clear方法
        cb.clear();
        //绝对位置的get，position位置不会改变
        System.out.println(cb.get(2));
    }
}
```

Chanel

基本概念

- channel可以将制定文件的部分或全部直接映射成buffer
- Channel只能和Buffer进行交互，我们队Channel的读写操作都需要经过Buffer。

重要方法

- map():用于将Channel对应的数据映射成ByteBuffer
- read()：从buffer中读取数据
- write()：往buffer中写入数据

基本使用

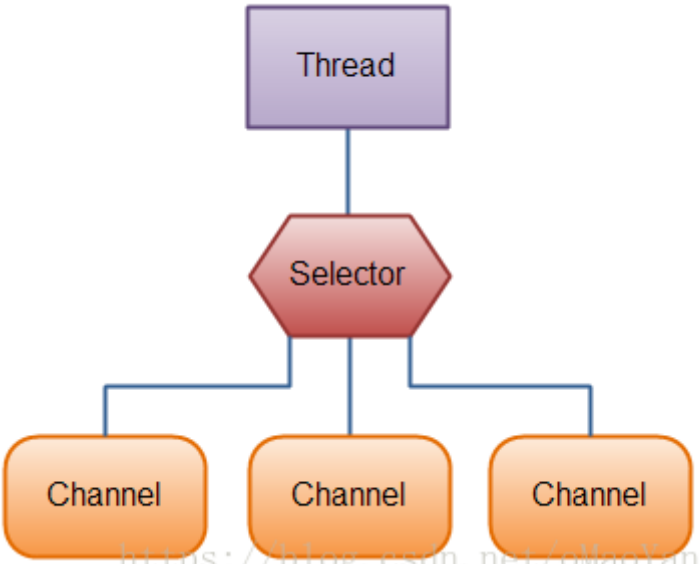
```
public class FileChannelTest {
    public static void main(String[] args) {
        File file = new File("src/io/nio/FileChannelTest.java");
        try {
            FileChannel inChannel = new FileInputStream(file).getChannel();
            FileChannel outChannel = new FileOutputStream("channel.txt").getChannel();
            //将数据映射到buffer
            MappedByteBuffer map = inChannel.map(FileChannel.MapMode.READ_ONLY, 0,
file.length());
            Charset gbk = Charset.forName("GBK");
            //将数据通过channel写入到channel.txt
            outChannel.write(map);
            map.clear();
            //对buffer进行解码，然后打印
            CharsetDecoder charsetDecoder = gbk.newDecoder();
            CharBuffer decode = charsetDecoder.decode(map);
            System.out.println(decode);

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

selecotr

基本概念

Selector是Java NIO中用于管理一个或多个Channel的组件，控制决定对哪些Channel进行读写；通过使用Selector让一个单线程可以管理多个Channel甚至多个网络连接。

使用Selector最大的优势就是可以在较少的线程中控制更多的Channel。事实上我们可以使用一个线程控制需要使用的所有Channel。操作系统线程的运行和切换需要一定的开销，使用的线程越小，系统开销也就越少；因此使用Selector可以节省很多系统开销。下图展示了一个线程使用Selector控制三个Channel的情形。



重要方法

基本使用