Java高级基础笔记整理

# Java基础API整理

## 1. Number类

1. Java.lang.Number 是一个抽象类：

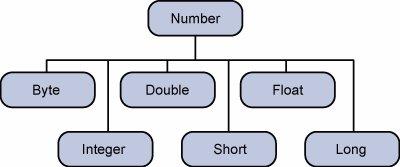
public abstract class Number

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

implements [Serializable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Serializable.html)

1. 它是所有数字型基本数据类型包装类的一个抽象超类:

Byte , Short , Integer, Long , Float , Double



3. 方法

|  |  |
| --- | --- |
| byte | [byteValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#byteValue--)()  返回指定号码作为值 byte ，这可能涉及舍入或截断。 |
| abstract double | [doubleValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#doubleValue--)()  返回指定数字的值为 double ，可能涉及四舍五入。 |
| abstract float | [floatValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#floatValue--)()  返回指定数字的值为 float ，可能涉及四舍五入。 |
| abstract int | [intValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#intValue--)()  返回指定号码作为值 int ，这可能涉及舍入或截断。 |
| abstract long | [longValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#longValue--)()  返回指定数字的值为 long ，可能涉及四舍五入或截断。 |
| short | [shortValue](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Number.html#shortValue--)()  返回指定号码作为值 short ，这可能涉及舍入或截断。 |

4. 练习

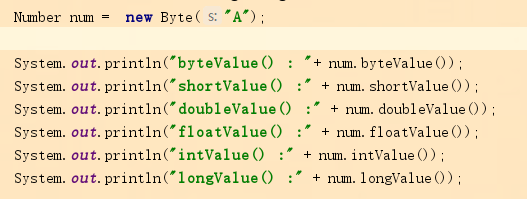
**整数型：**

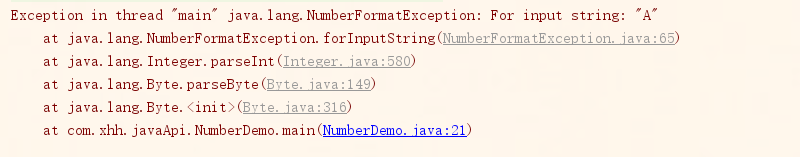
Number num = **new** Byte(**"1"**);  
  
System.***out***.println(**"byteValue() : "**+ num.byteValue());  
System.***out***.println(**"shortValue() :"** + num.shortValue());  
System.***out***.println(**"doubleValue() :"** + num.doubleValue());  
System.***out***.println(**"floatValue() :"** + num.floatValue());  
System.***out***.println(**"intValue() :"** + num.intValue());  
System.***out***.println(**"longValue() :"** + num.longValue());

结果：

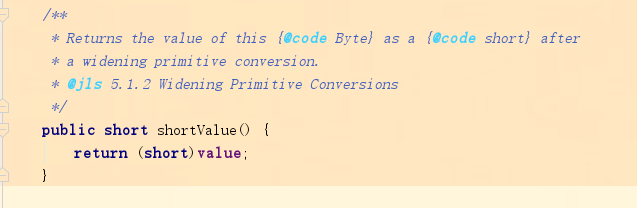


**注意： 如果Byte不是一个数字类型,当转成其他类型时会抛出异常**





**从源码可以看到其实是直接强制转换的:**



**浮点型：**

Number num = **new** Double(**"250.250"**);  
  
System.***out***.println(**"byteValue() : "**+ num.byteValue());  
System.***out***.println(**"shortValue() :"** + num.shortValue());  
System.***out***.println(**"doubleValue() :"** + num.doubleValue());  
System.***out***.println(**"floatValue() :"** + num.floatValue());  
System.***out***.println(**"intValue() :"** + num.intValue());  
System.***out***.println(**"longValue() :"** + num.longValue());

**结果：**



**注意：**

1. **当浮点型转换成整数型时，会截取掉后面的小数点。**

## 2. Character类

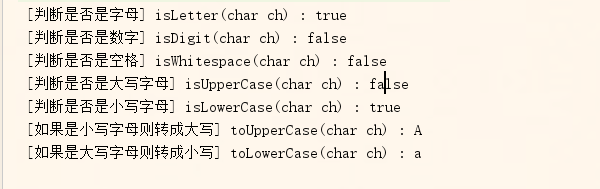
1. Character 类是char字符数据类型的包装类

2. 方法(静态方法)

|  |  |
| --- | --- |
| 序号 | 方法与描述 |
| 1 | isLetter() 是否是一个字母 |
| 2 | isDigit() 是否是一个数字字符 |
| 3 | isWhitespace() 是否一个空格 |
| 4 | isUpperCase() 是否是大写字母 |
| 5 | isLowerCase() 是否是小写字母 |
| 6 | toUpperCase() 指定字母的大写形式 |
| 7 | toLowerCase() 指定字母的小写形式 |
| 8 | toString() 返回字符的字符串形式，字符串的长度仅为1 |

3,方法练习

Character ch = **'a'**; *//自动装箱*System.***out***.println(**" [判断是否是字母] isLetter(char ch) : "** + Character.*isLetter*(ch));  
System.***out***.println(**" [判断是否是数字] isDigit(char ch) : "** + Character.*isDigit*(ch));  
System.***out***.println(**" [判断是否是空格] isWhitespace(char ch) : "** + Character.*isWhitespace*(ch));  
System.***out***.println(**" [判断是否是大写字母] isUpperCase(char ch) : "** + Character.*isUpperCase*(ch));  
System.***out***.println(**" [判断是否是小写字母] isLowerCase(char ch) : "** + Character.*isLowerCase*(ch));  
System.***out***.println(**" [如果是小写字母则转成大写] toUpperCase(char ch) : "** + Character.*toUpperCase*(ch));  
System.***out***.println(**" [如果是大写字母则转成小写] toLowerCase(char ch) : "** + Character.*toLowerCase*(ch));



## 3. Runtime类

### 1．类的作用

1. **在每一个JVM进程里面都会存在有一个Runtime类的对象，这个类的主要功能是取得一些运行时有关的环境属性或创建新的进程等操作。**
2. **Runtime类是单列设计模式**

**获取Runtime实例对象:**

**public static Runtime getRuntime();**

1. **Runtime类是单列设计模式**

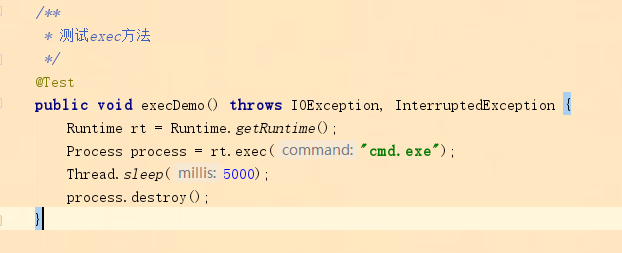
### 2．类提供的方法

**Runtime 中的方法：**

* [totalMemory](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Runtime.html#totalMemory--)() 返回Java虚拟机中的内存总量。
* [maxMemory](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Runtime.html#maxMemory--)() 返回Java虚拟机将尝试使用的最大内存量。
* [freeMemory](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Runtime.html#freeMemory--)() 返回Java虚拟机中的可用内存量。
* [gc](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Runtime.html#gc--)() 运行垃圾回收器。
* [exec](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Runtime.html#exec-java.lang.String-)([String](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/String.html) command) 在单独的进程中执行指定的字符串命令。

### 3．方法练习

**exec 方法：**



### 4．面试题

**1. 请解释什么叫GC，如何处理？**

* + GC(Garbage Collector)垃圾收集器，指的的是释放无用的内存空间
  + GC会有系统不定期的进行自动回收，或者调用Runtime类中的gc()方法手工回收。

## 4. System类

* public final class System
* extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)
  + - System类包含几个有用的类字段和方法。 它不能被实例化。
    - System类提供的System包括标准输入，标准输出和错误输出流;
    - 访问外部定义的属性和环境变量;
    - 一种加载文件和库的方法;
    - 以及用于快速复制阵列的一部分的实用方法。

### 1. 有关System类其他知识点

**1. System的构造器是私有的，类中的方法都是static修饰的静态方法。**

**2. System 中out、err、in 是静态常量而不是内部类。**

### 2. 类提供的方法

[currentTimeMillis](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#currentTimeMillis--)() 返回当前时间（以毫秒为单位）。

[console](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#console--)() 返回与当前Java虚拟机关联的唯一的[Console](mk:@MSITStore:D:\\my\\rxl\\API\\jdk%20api%201.8_google.CHM::/java/io/Console.html" \o "java.io中的类)对象（如果有）。

[exit](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#exit-int-)(int status) 终止当前运行的Java虚拟机。

[gc](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#gc--)() 运行垃圾回收器。

[getProperties](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#getProperties--)() 确定当前的系统属性。

[getProperty](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#getProperty-java.lang.String-)([String](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/String.html) key) 获取指定键指示的系统属性。

[setErr](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#setErr-java.io.PrintStream-)([PrintStream](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/PrintStream.html) err) 重新分配“标准”错误输出流。

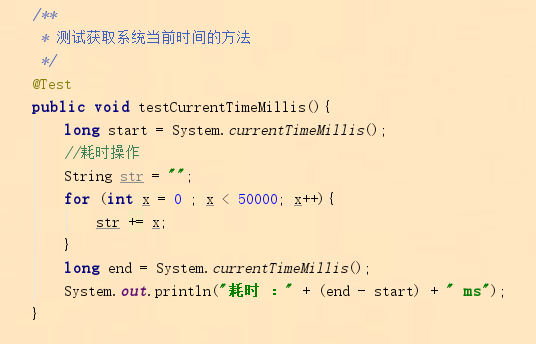
[setIn](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#setIn-java.io.InputStream-)([InputStream](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/InputStream.html) in) 重新分配“标准”输入流。

[setOut](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#setOut-java.io.PrintStream-)([PrintStream](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/PrintStream.html) out) 重新分配“标准”输出流。

[arraycopy](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#arraycopy-java.lang.Object-int-java.lang.Object-int-int-)([Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html) src, int srcPos, [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html) dest, int destPos, int length) 将指定源数组中的数组从指定位置复制到目标数组的指定位置。

### 3. 方法练习

#### [currentTimeMillis](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#currentTimeMillis--)()

耗时计算

#### [setOut](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#setOut-java.io.PrintStream-)([PrintStream](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/PrintStream.html) out)

#### [getProperties](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#getProperties--)()

#### [gc](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/System.html#gc--)()

**该方法实际上是直接调用了Runtime.getRuntime().gc()方法**

#### exit ()

**exit(int)方法终止当前正在运行的 Java 虚拟机，参数解释为状态码。根据惯例，非 0 的状态码表示异常终止。 而且，该方法永远不会正常返回。 这是唯一一个能够退出程序并不执行finally的情况。**

### 4. 面试题

**1. 请解释final、finally、finalize？**

* + final：关键字，定义不能被继承的类、不能被覆写的方法、常量;
  + finally：关键字，异常统一出口
  + finalize：方法，Object提供的方法(protected void finalize() throws Throwable),即使出异常了也不会导致中断执行。

## 5. 对象克隆

1. 对象Clone使用的是我们的Object类的中clone()方法：

* protected [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html) clone()

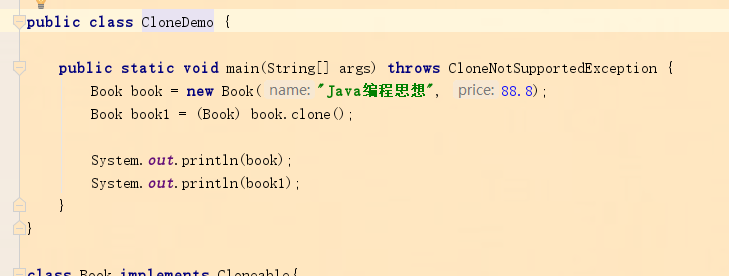
throws [CloneNotSupportedException](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/CloneNotSupportedException.html)

2. 如果此对象的类不实现接口Cloneable ，则抛出CloneNotSupportedException

3. Clone方法是一个protected方法，只能在本包中调用，所以需要覆写该方法。

4. Cloneable是一个空的接口定义，没有任何方法和常量。是一个表示接口，像通行证一样。

5.练习

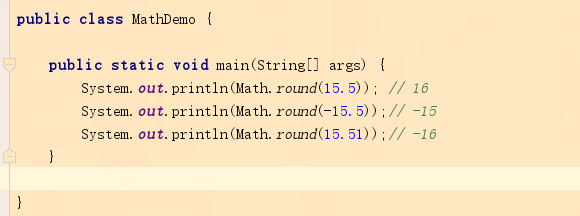


## 6. Java数学操作类

### 1. Math类

Math类中用得比较多的一个方法

|  |  |
| --- | --- |
| static long | [round](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Math.html#round-double-)(double a) |

 可能出现的面试题：

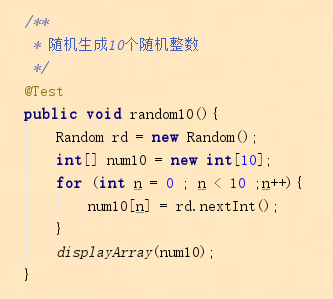
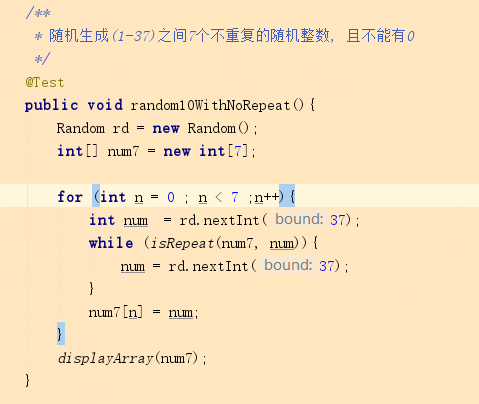
在Math类中，如果负数进行四舍五入，操作数的小数位必须大于0.5才进位，小于等于0.5不进位。

### 2. Random类

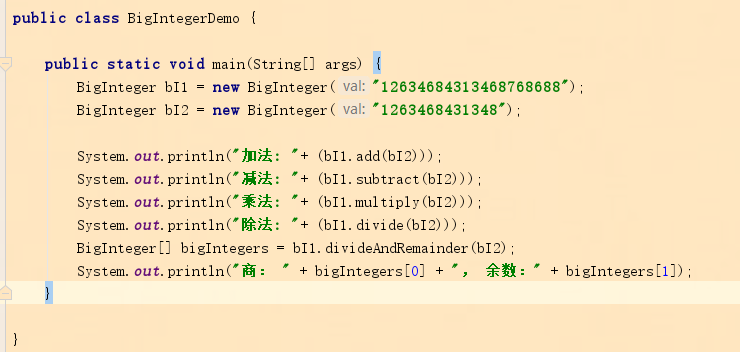
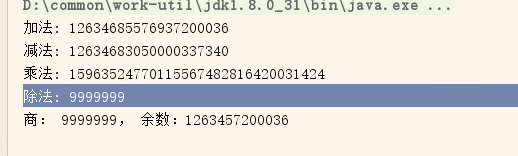
Random是一个取得随机数的一个类。

|  |  |
| --- | --- |
| int | [nextInt](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/util/Random.html#nextInt-int-)(int bound) |

我们来练习这一个方法，其他方法用法类似，只不过数据类型不一样。



### 3. BigInteger类

**支持****操作大整数的类。**

### 4. BigDecimal类

**支持大浮点型数的类。**

加法、减法、除法和BigInteger一样没啥区别。

除法不一样的地方：

[divide](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/BigDecimal.html#divide-java.math.BigDecimal-int-java.math.RoundingMode-)([BigDecimal](mk:@MSITStore:D:\\my\\rxl\\API\\jdk%20api%201.8_google.CHM::/java/math/BigDecimal.html" \o "class in java.math) divisor, int scale, [RoundingMode](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html) roundingMode)

* **divisor: 除数**
* **scale : 保留几位小数**
* **roundingMode :四舍五入模型**
  + **Enum RoundingMode**

|  |
| --- |
| [CEILING](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#CEILING)  圆形模式向正无穷大转弯。 |
| [DOWN](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#DOWN)  舍入模式向零舍入。 |
| [FLOOR](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#FLOOR)  舍入模式向负无穷大转弯。 |
| [HALF\_DOWN](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#HALF_DOWN)  四舍五入模式向“最近邻居”转弯，除非这两个邻居都是等距离的，在这种情况下，这是倒圆的。 |
| [HALF\_EVEN](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#HALF_EVEN)  四舍五入模式向“最近邻居”转弯，除非两个邻居都是等距离的，在这种情况下，向着邻居方向转移。 |
| [HALF\_UP](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#HALF_UP)  四舍五入模式向“最近邻居”转弯，除非两个邻居都是等距的，在这种情况下是圆括弧的。 |
| [UNNECESSARY](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#UNNECESSARY)  舍入模式来确定所请求的操作具有精确的结果，因此不需要舍入。 |
| [UP](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/math/RoundingMode.html#UP)  舍入模式从零开始。 |

## 7. Java日期操作类

### 1. Date类

**1. 获取当前日期**

*//获取当前日期*Date date = **new** Date();  
System.***out***.println(date);

**2. Long类跟日期类型的转换**

**构造方法：**

**public Date(Long date)**

**可以传入一个Long类的毫秒值**

*//Long类型和时间类的转换*Long time = System.*currentTimeMillis*();  
System.***out***.println(**new** Date(time));

### 2. SimpleDateFormate类

**SimpleDateFormate类是专门提供处理日期的一个工具类**

#### 1. 常用方法

⚫ 构造方法：public SimpleDateFormat(String pattern)，需要传递转换格式

⚫ 将 Date 转化为 String：public final String format(Date date)

⚫ 将 String 转化为 Date：public Date parse(String source)throws ParseException

#### 常用的转换格式

* yyyy - 年
* MM - 月
* dd - 日
* HH - 时
* mm - 分
* ss - 秒
* SSS - 毫秒

#### 练习

**public class** SimpleDateFormatDemo {  
  
 **public static final** String ***DATE\_PARTTEN*** = **"yyyy-MM-dd HH:mm:ss SSS"**;  
  
 */\*\*  
 \* 将Date类型转换为String类型  
 \*/* @Test  
 **public void** date2String(){  
 Date date = **new** Date();  
 SimpleDateFormat fmt = **new** SimpleDateFormat(***DATE\_PARTTEN***);  
 String dateStr = fmt.format(date);  
 System.***out***.println(dateStr);  
 }  
}

*/\*\*  
 \* 将String类型转换为Date类型  
 \** ***@throws*** *ParseException  
 \*/*@Test  
**public void** sting2Date() **throws** ParseException {  
 String dateStr = **"2018-12-05 15:15:15 555"**;  
 SimpleDateFormat fmt = **new** SimpleDateFormat(***DATE\_PARTTEN***);  
 Date date = fmt.parse(dateStr);  
 System.***out***.println(date);  
}

#### 4. 总结

**关于数据类型的转换：**

* **Date与String类型的转换主要依靠的是SimpleDateFormat类**
* **String与基本数据类型的转换主要依靠 .valueOf()方法。**
* **Long与Date之间的转换主要依靠getTime()方法。**

### 3. Calender类

public abstract class Calendar

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

1. Calender类提供简单的日期计算
2. 由于Calender类是一个抽象类，所以我们只能使用它的子类实现，

在Calender类中提供了一个静态方法获取该类的子类对象实现

|  |  |
| --- | --- |
| static [Calendar](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/util/Calendar.html) | [getInstance](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/util/Calendar.html#getInstance--)() |

# **2. String**

## 1. 不可变String

String a = “abc”;

String b = “abc”;

System.out.println(a == b);

答案：true

首先会去字符串常量池查找是否存在“abc”，这里a已经创建,所以给b直接赋值

## 2. String s = new String(“abc”);

String s = new String(“abc”)

首先会去到字符串常量池中查找是否存在“abc”，不存在则创建一个；

其次因为有new关键字，那么会在堆中创建一个对象,一共创建两个对象

## 3. 重载“+”,和StringBuilder

String a = “abc” ;

String b = a + “def” + 22；

当虚拟机遇到字符串“+“号时,会创建一个StringBuilder对象,追加字符串,所以当存在大量字符串相加的情况时，直接使用StringBuilder效果要好一点。

## 4. 格式化

当需要对数字、时间、日期等进行格式化的时候，我们可以采用

Java.util.Formatter类进行格式化，String类中的format()方法其内部也是调用的Formatter实现

**Java格式化大致分两类：常规类型的格式化和时间日期类型的格式化。**

### 常规类型格式化固定格式

**%[argument\_index$] [flags] [width] [.percision] conversion**

1. **argument\_index ：可选，是一个十进制整数，用于表明参数在参数列表中的位置。第一个参数由“1$“引用,第二个参数由”2$”引用，以此内推。**
2. **flags： 可选，用来控制输出格式。**
3. **width：可选，是一个正整数，表示输出的最小长度。**
4. **precision：可选，用来限定输出的精度。**
5. **converison：必须，用来表示如何格式化参数的字符。**

#### Conversion

‘b’ : 将参数格式化为boolean输出，‘B’的效果相同。

‘h’ : 将参数格式化为散列输出，原理Integer.toHexString(arg.hashCode()) ‘H’ 效果相同。

‘s’：将参数格式话为字符串输出，如果参数实现了Formattable接口，则调用formatTo方法。‘S’效果相同。

‘c’：将参数格式化为Unicode字符，‘C’效果相同。

‘d’：将参数格式化为十进制整数。

‘o’：将参数格式化为八进制整数。

‘x’：将参数格式化为十六进制整数。

‘e’：将参数格式化为科学技术法的浮点数，‘E’效果相同。

‘f’：将参数格式化为十进制浮点数。

‘g‘：根据具体情况，自动选择用普通表示方式还是科学计数方法。’G’效果相同。

‘a‘：结果格式化为带有效位数和指数的十六进制浮点数。

‘t’ ：时间日期格式化前缀

‘%‘：输出%。%

‘n‘：平台独立的行分隔符

**补充**：

* 对于浮点转换符 'e' 、'E' 和 'f'，精度是小数点分隔符后的位数。如果转换符是 'g' 或 'G'，那么精度是舍入计算后所得数值的所有位数。如果转换是 'a' 或 'A'，则不必指定精度。
* 对于部分转换符，如 'b' 和 'B' ，两者转换效果是相似的，但是 'B' 会把输出中的字母都转换为大写，其它相似特性的转换符应该还有好多，大家可以去找一下O(∩\_∩)O哈哈~

#### Flags

**Flags是可选参数用于控制输出格式**

参数值：

‘-‘ ： 最小宽度左对齐,不可以与“0填充“同时使用

‘+’ ：总是包含一个符号

‘ ‘ ：正直前加空格，负值钱加负号

‘0’ ：结果用0填充

‘,‘ ： 每三位数字用‘,’隔开

‘(‘ ： 若参数是负数，则结果中不添加负号而是用圆括号把数据括起来

#### Width

**用于控制输出的宽度**

#### precision

**用于控制浮点型数值的精度。**

### 时间日期格式化固定格式

%[argument\_index$][flags][width]conversion

#### 常见的日期和时间组合

c ： 包括全部日期和时间信息：如 星期四 十一月 01 10:46:58 CST 2018

F ： “年-月-日”格式 ：如 2018-11-01

D ： 年/月/日’格式 : 如 2018/11/01

r ： “HH:MM:SS PM” 格式 ：如 11:10:07 上午

T ： “HH:MM:SS” 格式：如 11:22:47

#### 单个日期输出

b : 月份简称 ：如 十一月

a ：星期简称, A 效果相同 ：如 星期四

C ：年的前两位（不足前面补0） : 如 20

y ：年的后两位 （不足前面补0）: 如 18

j : 一年的第几天：如 305

m ：两位数字的月份 ： 如 11

d : 两位数字的日（不足两位前面补0） ： 如 01

#### 单个时间输出

H : 2位数字24小时制的小时 如： 15 (不足前面补0)

I : 2位数字的12小时制的小时 如： 03 (不足前面补0)

k : 2位数字24小时制的小时 如： 15 （不补0）

M ：2位数的分钟：如 22

S ：2位数的秒: 如 07

L ：3位数的秒: 如 015

N ：9位数字的毫秒数

p ： 上午或下午

## 5. 正则表达式

**正则表达式定义字符串的模式，可以用来搜索、编辑或处理文本。**

**从jdk1.4起Java提供了java.util.regex.\* 包支持正则表达式**

**元字符：**

\ ： 转移符，将特殊符号转为普通符号

^ ： 位于[]中表示取反，位于[]外面表示一行的开头

$ : 一行的结尾

\* : 0个或多个字符

+ ： 1个或多个字符

? : 0个或1个字符，当此字符跟在(\*、+、?、{n},{n,}, {n,m}) 之后时,匹配 模式是”非贪心的”。”非贪心”模式匹配的尽快能短的字符串，而默认的”贪心的” 模式匹配的是尽可能长的模式。例如 ：字符串 oooo 中 o+? 只匹配o 而o +则匹配oooo。

{n} : n是非负整数，正好匹配n个

{n,} ：n是非负整数，至少匹配n个

{n,m} ：至少匹配n个，至多m个

x|y : 匹配x或y

[ayz] : 匹配字符集中的任意一个字符

[^ayz] : 反向字符集

[^a-z] : 反正字符范围

\b :匹配字符边界,即字与空格间的位置

\B :非边界字符

\cx: 匹配 x 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。x 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。

\d : 数字字符匹配 。等效于[0-9]

\D : 非数字字符匹配。等效于[^0-9]

\f : 匹配换页符。等效于\x0c或者\cL

\n : 匹配换行符。等效于 \x0a 和 \cJ。

\r : 匹配一个回车符。等效于\x0d和\cM。

\s : 匹配任何空白字符，包括空格，制表符，换行符.与[\f\n\r\t\v]等效

\S ：匹配任何非空白字符，与[^\f\n\r\t\v]等效

\t ：匹配制表符.与\x09 和\cl等效

\v ：匹配垂直制表符,与 \x0b 和 \cK等效

\w : 匹配任何字符类，包括下划线，与[a-zA-Z0-9\_]等效

\W :匹配非任何字符类，等效于[^a-zA-Z0-9\_]

### NFA引擎匹配原理

#### 引擎分类

**正则引擎大致分两类：**

**DFA (Deterministic finite automaton) ： 确定型有穷自动机**

**NFA(Non-Deterministic finite automaton) : 非确定型有穷自动机**

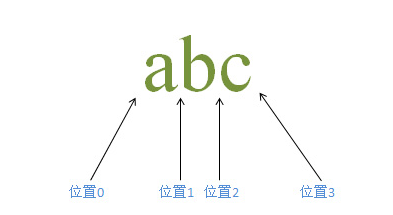
**DFA :**

**DFA从文本入手，从左到右，每个字符不会匹配两次，它的时间复杂度是多项式的，所以通常情况下它的速度更快，但支持的特性很少.**

**NFA :**

**NFA从正则表达式入手，不断读入字符，尝试是否匹配当前正则。不匹配则吐出字符重新匹配，通常速度较慢，最优时间是多项式。最差情况为指数级。但NFA支持更多的特性，因而绝大多数编程场景下(包括java, js)**

#### 字符串组成



**对于abc字符串而言，包含三个字符和四个位置。**

#### 占有字符和零宽度

正则表达式匹配过程中，如果子表达式匹配的是字符串，而非位置，并被保存到最终的匹配结果中，那么就认为这个子表达式是占有字符的；如果子表达式匹配的仅仅是位置，或者匹配的内容。

#### 控制权

源字符串："abcd"

正则表达式: "^abc" 这个表达式可以看做4个子表达式(^)(a)(b)(c)

匹配过程中第一个表达式获取控制权，从字符串某一位置开始匹配 一个子表达式开始尝试匹配的位置，是从前一子表达匹配成功的结束位置开始的

所以(^)匹配的是0位置，匹配成功后，把控制权给了表达式(a), 因为(^)匹配的是位置，并不匹配字符串，所有(a)还是匹配第一个字符串a, 匹配完成后，控制权交给下一个子表达式(b)，以此类推 (子表达式一)(子表达式二)

假设(子表达式一)为零宽度表达式，由于它匹配开始和结束的位置是同一个，如位置0，那么(子表达式二)是从位置0开始尝试匹配的。 假设(子表达式一)为占有字符的表达式，由于它匹配开始和结束的位置不是同一个，如匹配成功开始于位置0，结束于位置2，那么(子表达式二)是从位置2开始尝试匹配的

#### 传动

源字符串：my name is "sunny"

正则表达式: "sunny"

首先子表达式(")开始匹配字符m, 发现匹配失败了，那这个时候怎么办，正则引擎传动装置使正则向前传动，进入下一轮尝试， 然后子表达式(")开始匹配字符y， 匹配依然失败，正则在向前传动，直到匹配都字符"这个过程就是传动

#### 贪婪、非贪婪、独占

贪婪模式：在保证匹配成功的状态下，尽可能多的匹配 “{m,n}”、“{m,}”、“?”、“\*”和“+”

非贪婪： 在保证匹配成功的状态下，在尽可能少的匹配 “{m,n}?”、“{m,}?”、“??”、“\*?”和“+?”

独占：（*?+*、*\*+*、*++*、*{m,n}+*、*{m,}+）*

比如说"+"的原意是匹配1到多个，如果是贪婪模式，他会尽可能的匹配多个，如果是非贪婪模式，他只会匹配一个

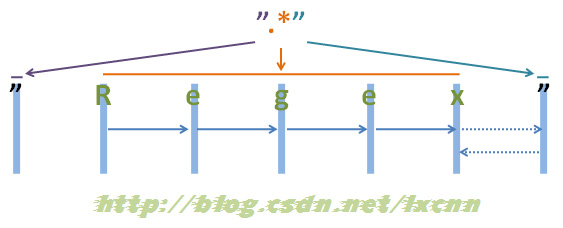
比如说"?"的原意是匹配0或者1个，如果是贪婪模式，他会尽可能的匹配1哥，如果是非贪婪模式，他就不匹配了

注意：这些选择都是需要保证匹配成功，也就是说最终如果"??"不匹配使得整个正则表达式匹配不成功，那么久需要回溯，尝试匹配

#### 基本匹配原理

源字符串："Regex"

正则表达式：".\*"



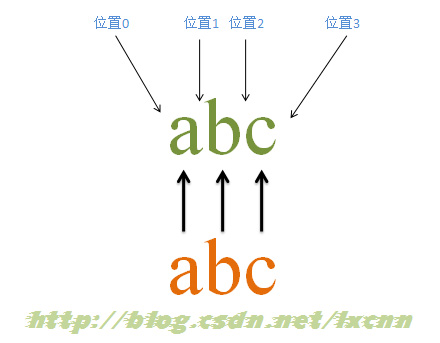
为了能够看清晰匹配过程，上面的空隙留得较大，实际源字符串为“”Regex””，下同。 来看一下匹配过程。首先由第一个“"”取得控制权，匹配位置0位的“"”，匹配成功，控制权交给“.*”。 “.*”取得控制权后，由于“*”是匹配优先量词，在可匹配可不匹配的情况下，优先尝试匹配。从位置1处的“R”开始尝试匹配，匹配成功，继续向右匹配，匹配位置2处的“e”，匹配成功，继续向右匹配，直到匹配到结尾的“””，匹配成功，由于此时已匹配到字符串的结尾，所以“.*”结束匹配，将控制权交给正则表达式最后的“"”。 “"”取得控制权后，由于已经在字符串结束位置，匹配失败，向前查找可供回溯的状态，控制权交给“.*”，由“.*”让出一个字符，也就是字符串结尾处的“””，再把控制权交给正则表达式最后的“"”，由“"”匹配字符串结尾处的“"”，匹配成功。 此时整个正则表达式匹配成功，其中“.\*”匹配的内容为“Regex”，匹配过程中进行了一次回溯

### 匹配过程

#### 正则表达式简单匹本过程

源字符串：abc

正则表达式：abc



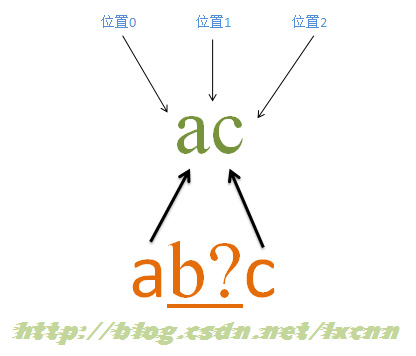
匹配过程：

首先由字符“a”取得控制权，从位置0开始匹配，由“a”来匹配“a”，匹配成功，控制权交给字符“b”；由于“a”已被“a”匹配，所以“b”从位置1开始尝试匹配，由“b”来匹配“b”，匹配成功，控制权交给“c”；由“c”来匹配“c”，匹配成功。 此时正则表达式匹配完成，报告匹配成功。匹配结果为“abc”，开始位置为0，结束位置为3

#### 含有匹配优先量词的匹配过程——匹配成功

源字符串：ac

正则表达式：ab?c



匹配过程：

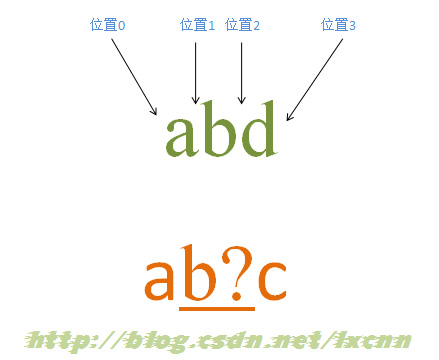
首先由字符“a”取得控制权，从位置0开始匹配，由“a”来匹配“a”，匹配成功，控制权交给字符“b?”；先尝试进行匹配，由“b?”来匹配“c”，同时记录一个备选状态，匹配失败，此时进行回溯，找到备选状态，“b?”忽略匹配，让出控制权，把控制权交给“c”；由“c”来匹配“c”，匹配成功。 此时正则表达式匹配完成，报告匹配成功。匹配结果为“ac”，开始位置为0，结束位置为2。其中“b?”不匹配任何内容。

需要注意的是子表达式(b?)会优先匹配字符,(?)也是贪婪表达式

#### 含有匹配优先量词的匹配过程——匹配失败

源字符串：abd

正则表达式：ab?c



匹配过程：

首先由字符“a”取得控制权，从位置0开始匹配，由“a”来匹配“a”，匹配成功，控制权交给字符“b?”；先尝试进行匹配，由“b?”来匹配“b”，同时记录一个备选状态，匹配成功，控制权交给“c”；由“c”来匹配“d”，匹配失败，此时进行回溯，找到记录的备选状态，“b?”忽略匹配，即“b?”不匹配“b”，让出控制权，把控制权交给“c”；由“c”来匹配“b”，匹配失败。此时第一轮匹配尝试失败。

当第一轮匹配失败后，正则表达式会向前传动

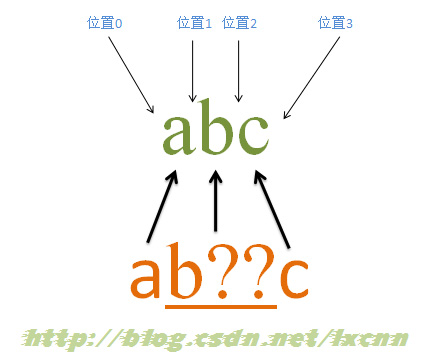
正则引擎使正则向前传动，由位置1开始尝试匹配，由“a”来匹配“b”，匹配失败，没有备选状态，第二轮匹配尝试失败。

继续向前传动，直到在位置3尝试匹配d失败，匹配结束。此时报告整个表达式匹配失败

#### 含有忽略优先量词的匹配过程——匹配成功

源字符串：abc

正则表达式：ab??c

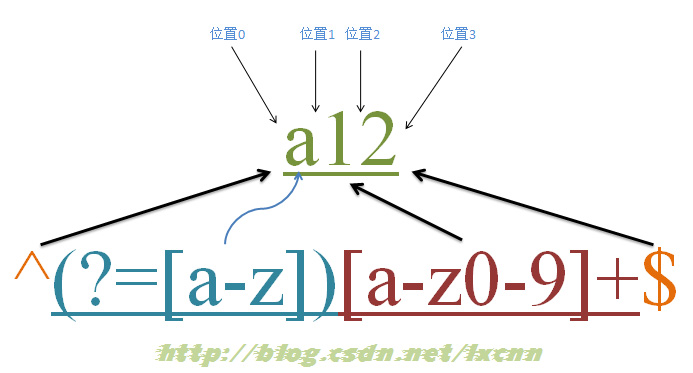


量词“??”属于忽略优先量词，在可匹配可不匹配时，会先选择不匹配，只有这种选择会使整个表达式无法匹配成功时，才会尝试进行匹配。这里的量词“??”是用来修饰字符“b”的，所以“b??”是一个整体。 匹配过程： 首先由字符“a”取得控制权，从位置0开始匹配，由“a”来匹配“a”，匹配成功，控制权交给字符“b??”；先尝试忽略匹配，即“b??”不进行匹配，同时记录一个备选状态，控制权交给“c”；由“c”来匹配“b”，匹配失败，此时进行回溯，找到记录的备选状态，“b??”尝试匹配，即“b??”来匹配“b”，匹配成功，把控制权交给“c”；由“c”来匹配“c”，匹配成功。 此时正则表达式匹配完成，报告匹配成功。匹配结果为“abc”，开始位置为0，结束位置为3。其中“b??”匹配字符“b”。

#### 零宽度匹配过程

源字符串：a12

正则表达式：^(?=[a-z])[a-z0-9]+$

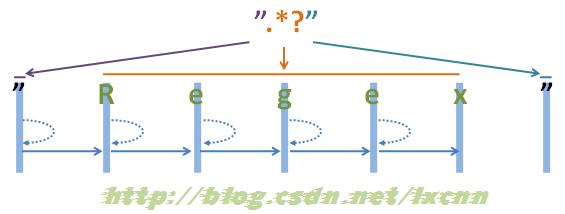


元字符“^”和“$”匹配的只是位置，顺序环视“(?=[a-z])”只进行匹配，并不占有字符，也不将匹配的内容保存到最终的匹配结果，所以都是零宽度的。 这个正则的意义就是匹配由字母或数字组成的，第一个字符是字母的字符串。 匹配过程： 首先由元字符“^”取得控制权，从位置0开始匹配，“^”匹配的就是开始位置“位置0”，匹配成功，控制权交给顺序环视“(?=[a-z])”； “(?=[a-z])”要求它所在位置右侧必须是字母才能匹配成功，零宽度的子表达式之间是不互斥的，即同一个位置可以同时由多个零宽度子表达式匹配，所以它也是从位置0尝试进行匹配，位置0的右侧是字符“a”，符合要求，匹配成功，控制权交给“[a-z0-9]+”； 因为“(?=[a-z])”只进行匹配，并不将匹配到的内容保存到最后结果，并且“(?=[a-z])”匹配成功的位置是位置0，所以“[a-z0-9]+”也是从位置0开始尝试匹配的，“[a-z0-9]+”首先尝试匹配“a”，匹配成功，继续尝试匹配，可以成功匹配接下来的“1”和“2”，此时已经匹配到位置3，位置3的右侧已没有字符，这时会把控制权交给“$”； 元字符“$”从位置3开始尝试匹配，它匹配的是结束位置，也就是“位置3”，匹配成功。 此时正则表达式匹配完成，报告匹配成功。匹配结果为“a12”，开始位置为0，结束位置为3。其中“^”匹配位置0，“(?=[a-z])”匹配位置0，“[a-z0-9]+”匹配字符串“a12”，“$”匹配位置3。

#### 非贪婪模式简单的匹配过程

源字符串："Regex"

正则表达式：".\*?"

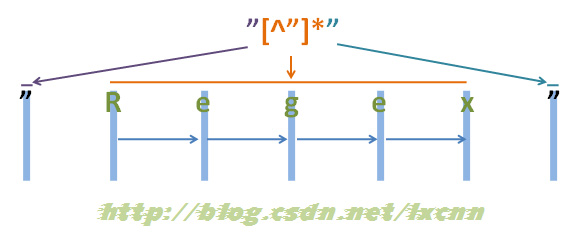


看一下非贪婪模式的匹配过程。首先由第一个“**"**”取得控制权，匹配位置0位的“**"**”，匹配成功，控制权交给“**.*?****”。 “****.*?**”取得控制权后，由于“***?****”是忽略优先量词，在可匹配可不匹配的情况下，优先尝试不匹配，由于“*”等价于“**{0,}**”，所以在忽略优先的情况下，可以不匹配任何内容。从位置1处尝试忽略匹配，也就是不匹配任何内容，将控制权交给正则表达式最后的“””。 “"”取得控制权后，从位置1处尝试匹配，由“"”匹配位置1处的“R”，匹配失败，向前查找可供回溯的状态，控制权交给“.*?”，由“.*?”吃进一个字符，匹配位置1处的“R”，再把控制权交给正则表达式最后的“"”。 “"”取得控制权后，从位置2处尝试匹配，由“"”匹配位置1处的“e”，匹配失败，向前查找可供回溯的状态，重复以上过程，直到由“.*?”匹配到“x”为止，再把控制权交给正则表达式最后的“"”。 “"”取得控制权后，从位置6处尝试匹配，由“"”匹配字符串最后的“"”，匹配成功。 此时整个正则表达式匹配成功，其中“.*?”匹配的内容为“Regex”，匹配过程中进行了五次回溯

#### 效率最优

源字符串："Regex"

给出正则表达式三："[^"]\*"



首先由第一个“"”取得控制权，匹配位置0位的“"”，匹配成功，控制权交给“["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)*”。 “*["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)”取得控制权后，由于“*”是匹配优先量词，在可匹配可不匹配的情况下，优先尝试匹配。从位置1处的“R”开始尝试匹配，匹配成功，继续向右匹配，匹配位置2处的“e”，匹配成功，继续向右匹配，直到匹配到“x”，匹配成功，再匹配结尾的“””时，匹配失败，将控制权交给正则表达式最后的“"”。 “””取得控制权后，匹配字符串结尾处的“””，匹配成功。 此时整个正则表达式匹配成功，其中“*["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)”匹配的内容为“Regex”，匹配过程中没有进行回溯。 将量词修饰的子表达式由范围较大的“.”，换成了排除型字符组“["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)”，使用的仍是贪婪模式，很完美的解决了需求和效率问题。当然，由于这一匹配过程没有进行回溯，所以也不需要记录回溯状态，这样就可以使用固化分组，对正则做进一步的优化。 给出正则表达式四："(?>["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)*)" 固化分组并不是所有语言都支持的，如.NET支持，而Java就不支持，但是在Java中却可以使用更简单的占有优先量词来代替："*["](https://sunnyingit.github.io/book/section_linux/regex.html#fn_)+"。

贪婪和非贪婪只是影响效率，不会影响结果

理解了以上之后，基本上能应付大部分的正则表达式需求

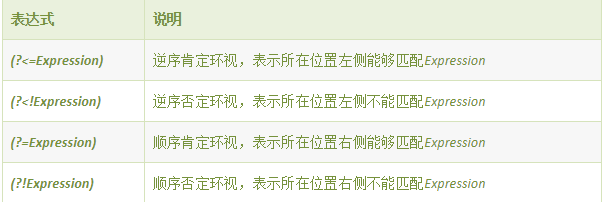
### 环视

#### 环视基础

环视只进行子表达式的匹配，不占有字符，匹配到的内容不保存到最终的匹配结果，是零宽度的。环视匹配的最终结果就是一个位置。

环视的作用相当于对所在位置加了一个附加条件，只有满足这个条件，环视子表达式才能匹配成功。

环视按照方向划分有顺序和逆序两种，按照是否匹配有肯定和否定两种，组合起来就有四种环视。顺序环视相当于在当前位置右侧附加一个条件，而逆序环视相当于在当前位置左侧附加一个条件。



对于环视的叫法，有的文档里叫预搜索，有的叫什么什么断言的，这里使用了更多人容易接受的《精通正则表达式》中“环视”的叫法，其实叫什么无所谓，只要知道是什么作用就是了，就这么几个语法规则， 还是很容易记的。

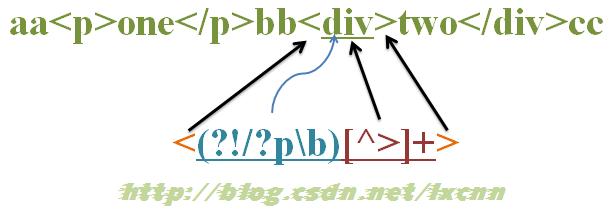
#### 环视匹配原理

环视相当于对“所在位置”附加了一个条件，环视的难点在于找到这个“位置”，这一点解决了，环视也就没什么秘密可言了。

##### 顺序环视匹配过程

对于顺序肯定环视(?=Expression)来说，当子表达式Expression匹配成功时，(?=Expression)匹配成功，并报告(?=Expression)匹配当前位置成功。

对于顺序否定环视(?!Expression)来说，当子表达式Expression匹配成功时，(?!Expression)匹配失败；当子表达式Expression匹配失败时，(?!Expression)匹配成功，并报告(?!Expression)匹配当前位置成功；

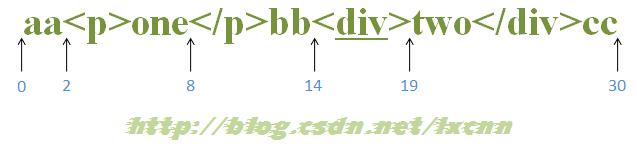


源字符串：aa<p>one</p>bb<div>two</div>cc

正则表达式：<(?!/?p\b)[^>]+>

**这个正则的意义就是匹配除<p…>或</p>之外的其余标签。**

**匹配过程：**



首先由字符“<”取得控制权，从位置0开始匹配，由于“<”匹配“a”失败，在位置0处整个表达式匹配失败，第一次迭代匹配失败，正则引擎向前传动，由位置1处开始尝试第二次迭代匹配。

重复以上过程，直到位置2，“<”匹配“<”成功，控制权交给“(?!/?p\b)”；“(?!/?p\b)”子表达式取得控制权后，进行内部子表达式的匹配。首先由“/?”取得控制权，尝试匹配“p”失败，进行回溯，不匹配，控制权交给“p”；由“p”来尝试匹配“p”，匹配成功，控制权交给“\b”；由“\b”来尝试匹配位置4，匹配成功。此时子表达式匹配完成，“/?p\b”匹配成功，那么环视表达式“(?!/?p\b)”就匹配失败。在位置2处整个表达式匹配失败，新一轮迭代匹配失败，正则引擎向前传动，由位置3处开始尝试下一轮迭代匹配。

在位置8处也会遇到一轮“/?p\b”匹配“/p”成功，而导致环视表达式“(?!/?p\b)”匹配失败，从而导致整个表达式匹配失败的过程。

重复以上过程，直到位置14，“<”匹配“<”成功，控制权交给“(?!/?p\b)”；“/?”尝试匹配“d”失败，进行回溯，不匹配，控制权交给“p”；由“p”来尝试匹配“d”，匹配失败，已经没有备选状态可供回溯，匹配失败。此时子表达式匹配完成，“/?p\b”匹配失败，那么环视表达式“(?!/?p\b)”就匹配成功。匹配的结果是位置15，然后控制权交给“[^>]+”；由“[^>]+”从位置15进行尝试匹配，可以成功匹配到“div”，控制权交给“>”；由“>”来匹配“>”。

此时正则表达式匹配完成，报告匹配成功。匹配结果为“<div>”，开始位置为14，结束位置为19。其中“<”匹配“<”，“(?!/?p\b)”匹配位置15，“[^>]+”匹配字符串“div”，“>”匹配“>”。

##### 逆序环视基础

对于逆序肯定环视(?<=Expression)来说，当子表达式Expression匹配成功时，(?<=Expression)匹配成功，并报告(?<=Expression)匹配当前位置成功。

对于逆序否定环视(?<!Expression)来说，当子表达式Expression匹配成功时，(?<!Expression)匹配失败；当子表达式Expression匹配失败时，(?<!Expression)匹配成功，并报告(?<!Expression)匹配当前位置成功；

顺序环视相当于在当前位置右侧附加一个条件，所以它的匹配尝试是从当前位置开始的，然后向右尝试匹配，直到某一位置使得匹配成功或失败为止。而逆序环视的特殊处在于，它相当于在当前位置左侧附加一个条件，所以它不是在当前位置开始尝试匹配的，而是从当前位置左侧某一位置开始，匹配到当前位置为止，报告匹配成功或失败。

顺序环视尝试匹配的起点是确定的，就是当前位置，而匹配的终点是不确定的。逆序环视匹配的起点是不确定的，是当前位置左侧某一位置，而匹配的终点是确定的，就是当前位置。

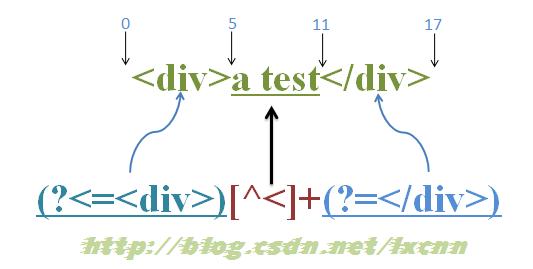
所以顺序环视相对是简单的，而逆序环视相对是复杂的。这也就是为什么大多数语言和工具都提供了对顺序环视的支持，而只有少数语言提供了对逆序环视支持的原因。

JavaScript中只支持顺序环视，不支持逆序环视。

Java中虽然顺序环视和逆序环视都支持，但是逆序环视只支持长度确定的表达式，逆序环视中量词只支持“?”，不支持其它长度不定的量词。长度确定时，引擎可以向左查找固定长度的位置作为起点开始尝试匹配，而如果长度不确定时，就要从当前位置向左逐个位置开始尝试匹配，不成功则回溯，再向左侧位置进行尝试匹配，然后重复以上过程，直到匹配成功，或是尝试到位置0处以后，报告匹配失败，处理的复杂度是显而易见的。

目前只有.NET中支持不确定长度的逆序环视

##### 逆序环视匹配过程



源字符串：<div>a test</div>

正则表达式：(?<=<div>)[^<]+(?=</div>)

这个正则的意义就是匹配<div>和</div>标签之间的内容，而不包括<div>和</div>标签本身。

**匹配过程：**

首先由“(?<=<div>)”取得控制权，从位置0开始匹配，由于位置0是起始位置，左侧没有任何内容，所以“<div>”必然匹配失败，从而环视表达式“(?<=<div>)”匹配失败，导致整个表达式在位置0处匹配失败。第一轮迭代匹配失败，正则引擎向前传动，由位置1处开始尝试第二次迭代匹配。

直到传动到位置5，“(?<=<div>)”取得控制权，向左查找5个位置，由位置0开始匹配，由“<div>”匹配“<div>”成功，从而“(?<=<div>)”匹配成功，匹配的结果为位置5，控制权交给“[^<]+”；“[^<]+”从位置5开始尝试匹配，匹配“a test”成功，控制权交给“(?=</div>)”；由“</div>”匹配“</div>”成功，从而“(?=</div>)”匹配成功，匹配结果为位置11。

此时正则表达式匹配完成，报告匹配成功。匹配结果为“a test”，开始位置为5，结束位置为11。其中“(?<=<div>)”匹配位置5，“[^<]+”匹配“a test”，“(?=</div>)”匹配位置11。

逆序否定环视的匹配过程与上述过程类似，区别只是当Expression匹配失败时，逆序否定表达式(?<!Expression)才匹配成功。

到此环视的匹配原理已基本讲解完，环视也就没有什么秘密可言了，所需要的，也只是多加练习而已。

### Java正则表达式

#### 基础

**java.util.regex** 包主要包括以下三个类：

**Pattern** 类：

pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

**Matcher** 类：

Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

**PatternSyntaxException**：

PatternSyntaxException 是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

#### 捕获组

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式 (dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C））），有四个这样的组：

* ((A)(B(C)))
* (A)
* (B(C))
* (C)

可以通过调用 matcher 对象的 groupCount 方法来查看表达式有多少个分组。groupCount 方法返回一个 int 值，表示matcher对象当前有多个捕获组。

还有一个特殊的组（group(0)），它总是代表整个表达式。该组不包括在 groupCount 的返回值中。

#### 基础语法练习

##### 校验邮政编码

规则：邮政编码由6位的数字组成。

由规则给出正则表达式：

Regex : \\d{6}

# 3. Java类型信息、枚举、反射、泛型、注解的应用

## 1. 注解

### 怎么定义注解

**注解的含义类似于标签，在Java中通过@interface关键字定义注解**

**public** @**interface** AnnotationFirstDemo {  
  
}

### Java中的元注解

**给注解用的注解叫元注解。**

#### @Retention

**该元注解定义注解的存活时间**

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***ANNOTATION\_TYPE***)  
**public** @**interface** Retention {  
 RetentionPolicy value();  
}

**该注解有一个 RetentionPolicy 枚举类型的属性**

**我们来看看这个枚举类**

**public enum** RetentionPolicy {  
 ***SOURCE***,  
 ***CLASS***,  
***RUNTIME***}

* RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。
* RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到 JVM 中。
* RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

**如果定义的注解不加** @Retention **那么默认是**CLASS

#### @Target

**该注解定义注解可以运用到哪些地方**

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***ANNOTATION\_TYPE***)  
**public** @**interface** Target {  
 ElementType[] value();  
}

**@Target 有一个ElementType类型的属性**

**我们来看一下**

**public enum** ElementType {  
 ***TYPE***,***FIELD***,***METHOD***,***PARAMETER***,***CONSTRUCTOR***,***LOCAL\_VARIABLE***,***ANNOTATION\_TYPE***,***PACKAGE***,

*// 1.8 以后* ***TYPE\_PARAMETER***,***TYPE\_USE***}

* ElementType.ANNOTATION\_TYPE 可以给一个注解进行注解
* ElementType.CONSTRUCTOR 可以给构造方法进行注解
* ElementType.FIELD 可以给属性进行注解
* ElementType.LOCAL\_VARIABLE 可以给局部变量进行注解
* ElementType.METHOD 可以给方法进行注解
* ElementType.PACKAGE 可以给一个包进行注解
* ElementType.PARAMETER 可以给一个方法内的参数进行注解
* ElementType.TYPE 可以给一个类型进行注解，比如类、接口、枚举

#### @Inherited

**这个元注解可以定义注解是可以继承的，并不是注解本身可以被继承**

**而是如果一个类被该注解运用，那么它的子类也拥有该注解。**

#### @Repeatable

**这个元注解可以定义注解可以重复运用在同一处地方。**

@Repeatable(Annotation2.**class**)  
**public** @**interface** AnnotationFirstDemo {  
  
}  
  
@**interface** Annotation2{  
 AnnotationFirstDemo[] value();  
}

**该注解必须要有一个可以存放注解的容器。**

#### @Documented

**顾名思义，这个元注解肯定是和文档有关。它的作用是能够将注解中的元素包含到 Javadoc 中去。**

### 注解中的属性

**基本类型属性**

@**interface** Annotation3{  
  
 *// 可以通过default 指定默认值* String id() **default "0"**;  
  
 *// 可以是8种数据类型, 但是其包装类型不行  
 // 如果注解仅仅只存在value 一个属性，那么在用的时候则不需要指定属性名* **int** value();  
  
}

**高级属性**

@**interface** Annotation4{  
  
 *// 数组类型属性* **int**[] array() **default** {1, 2, 3};  
 *// 枚举类型属性* MyEnum1 e();  
  
}

### 注解与反射

* **<A extends Annotation> getAnnotation(Class<A> annotationClass) 该元素如果存在指定类型的注解，则返回这些注解，否则返回 null。**
* **Annotation[] getAnnotations() 返回此元素上存在的所有注解，包括从父类继承的**
* **boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) 如果指定类型的注解存在于此元素上，则返回 true，否则返回 false。**
* **Annotation[] getDeclaredAnnotations() 返回直接存在于此元素上的所有注解，注意，不包括父类的注解，调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响，没有则返回长度为0的数组**

@Annotation4(e = MyEnum1.***GREE***)  
**class** AnnotationClass **extends** AnnotationClassP{  
  
 **public static void** main(String[] args) {  
 Class<?> clazz = AnnotationClass.**class**;  
 Annotation[] annotations = clazz.getAnnotations();  
 System.***out***.println(**"getAnnotations ："**);  
 **for** (Annotation a : annotations){  
 **if** ( a **instanceof** Annotation4){  
 System.***out***.println(((Annotation4)a).e().name());  
 }  
 **if** (a **instanceof** Annotation3){  
 System.***out***.println(((Annotation3)a).id());  
 }  
 }  
 Annotation4 annotation = clazz.getAnnotation(Annotation4.**class**);  
 System.***out***.println(annotation.array());  
 System.***out***.println(clazz.isAnnotationPresent(Annotation4.**class**));  
 *//返回直接的注解，不包括父类的* System.***out***.println(**"getDeclaredAnnotations ："**);  
 Annotation[] annotations1 = clazz.getDeclaredAnnotations();  
 **for** (Annotation a : annotations){  
 **if** ( a **instanceof** Annotation4){  
 System.***out***.println(((Annotation4)a).e().name());  
 }  
 **if** (a **instanceof** Annotation3){  
 System.***out***.println(((Annotation3)a).id());  
 }  
 }  
 }  
}

## 枚举

### 怎么定义枚举类型

**通过enum 关键字定义枚举类型**

**enum** MyColor{  
  
 ***GREE***, ***RED***, ***WHITE***}

### 2. 枚举的实现原理

枚举类型在编译的时候会生成一个Class文件，该类继承了Enum对象，且编译器会生成values() 和valueOf()方法

### Enum抽象类

我们知道了枚举实现实际上是继承了Enum类，那我们来看看Enum类

#### 定义

**public abstract class Enum<E extends Enum<E>>**

**extends Object implements Comparable<E>, Serializable**

#### 构造方法

* + **Enum是一个抽象类**
  + **它有一个唯一的构造方法**

#### 3. 常用方法

* **protected Object clone()** 抛出CloneNotSupportedException。
* **int** **compareTo(E o)** 将此枚举与指定的对象进行比较以进行订购。
* **boolean equals(Object other)**  如果指定的对象等于此枚举常量，则返回true。
* **protected void finalize()** 枚举类不能有finalize方法。
* **Class<E> getDeclaringClass()** 返回与此枚举常量的枚举类型相对应的Class对象。
* **int hashCode()**  返回此枚举常量的哈希码。
* **String name()**  返回此枚举常量的名称，与其枚举声明中声明的完全相同。
* **int ordinal()**  返回此枚举常数的序数（其枚举声明中的位置，其中初始常数的序数为零）。
* **String toString()** 返回声明中包含的此枚举常量的名称。
* **static <T extends Enum<T>> T valueOf(**Class<T> enumType, String name) 返回具有指定名称的指定枚举类型的枚举常量。

**方法解析：**

* Enum 类实现了Comparable接口，所以支持compareTo()
* Enum 重写了clone()方法
* Name() 方法一般和定义的时候相同
* Ordinal() 方法是定义的顺序，compareTo() 方法内部就是利用ordinal值进行比较

**方法练习：**

*/\*\*  
 \* 枚举原理练习  
 \*/***public enum** EnumPrinciple {  
 ***DAY1***, ***DAY2***, ***DAY3***}  
  
**class** Demo{  
 **public static void** main(String[] args) {  
 EnumPrinciple[] enumPrinciples =  
 {EnumPrinciple.***DAY1***,EnumPrinciple.***DAY2***,EnumPrinciple.***DAY3***};  
  
 **for** (EnumPrinciple e : enumPrinciples){  
 System.***out***.println(e.name());  
 System.***out***.println(e.ordinal());  
 }  
 }  
}

**DAY1**

**0**

**DAY2**

**1**

**DAY3**

**2**

### 4. 枚举类与Class对象

在Class对象中提供了两个与枚举有关的方法

* **T[] getEnumConstants()** 返回该枚举类型的所有元素，如果Class对象不是枚举类型，则返回null。
* **boolean isEnum()** 当且仅当该类声明为源代码中的枚举时返回 true

*/\*\*  
 \* 枚举与Class  
 \*/*@Test  
**public void** testEnumWithClass(){  
 *//等同于getClass()一样* Class<Day> eClass = Day.***DAY1***.getDeclaringClass();  
 System.***out***.println(**"是否是枚举类型:"** + eClass.isEnum());  
 Day[] es = eClass.getEnumConstants();*//返回枚举中的所有实例对象* **for** (Day enumPrinciple : es){  
 System.***out***.println(enumPrinciple.name());  
 System.***out***.println(enumPrinciple.ordinal());  
 }  
}

**是否是枚举类型:true**

**DAY1**

**0**

**DAY2**

**1**

**DAY3**

**2**

### 5. 枚举进阶用法

#### 向枚举中添加自定义构造和方法

1. **public class** EnumDefinedMethod {  
    @Test  
    **public void** test1(){  
    *// Day1 - Day7 实际上是MyEnum的一个实例对象* MyEnum[] myEnums = {MyEnum.***Day1***, MyEnum.***Day2***,MyEnum.***Day3***,  
    MyEnum.***Day4***, MyEnum.***Day5***, MyEnum.***Day6***,MyEnum.***Day7***};  
    **for** (MyEnum e : myEnums){  
    System.***out***.println(e.getDesc());  
    }  
    }  
   }  
   **enum** MyEnum{  
     
    ***Day1***(**"第一天"**),  
    ***Day2***(**"第二天"**),  
    ***Day3***(**"第三天"**),  
    ***Day4***(**"第四天"**),  
    ***Day5***(**"第五天"**),  
    ***Day6***(**"第六天"**),  
    ***Day7***(**"第七天"**);  
     
    **private** String **desc**;*//类描述* **public** String getDesc() {  
    **return desc**;  
    }  
    */\*\*  
    \* 私有化构造，防止外部调用  
    \** ***@param* desc** *\*/* **private** MyEnum(String desc){  
    **this**.**desc** = desc;  
    }  
   }

#### 2. 向枚举中定义抽象方法

**public class** EnumDefinedAbstractMethod {  
 @Test  
 **public void** test1(){  
 MyEnum1[] myEnums = {MyEnum1.***Day1***, MyEnum1.***Day2***};  
 **for** (MyEnum1 e : myEnums){  
 e.print();  
 }  
 }  
}  
**enum** MyEnum1{  
  
 ***Day1***(**"第一天"**){  
 **public void** print() {  
 System.***out***.println(**"这是"** + **this**.getDesc());  
 }  
 },  
 ***Day2***(**"第二天"**){  
 **public void** print() {  
 System.***out***.println(**"这是"** + **this**.getDesc());  
 }  
 };  
  
 **private** String **desc**;*//类描述* **public** String getDesc() {  
 **return desc**;  
 }  
 */\*\*  
 \* 私有化构造，防止外部调用  
 \** ***@param* desc** *\*/* **private** MyEnum1(String desc){  
 **this**.**desc** = desc;  
 }  
  
 **public abstract void** print();  
}

这是第一天

这是第二天

#### 枚举类实现接口

**枚举类可以实现接口**

**interface** Foot{  
 *//踢* **public void** kick();  
}  
  
**interface** head{  
 *//撞* **public void** strike();  
  
}  
  
**public enum** EnumWithInterface **implements** Foot, head{  
  
 ***Day1***,***Day2***;*//必须以分号结束* **public void** kick() {  
 System.***out***.println(**"向前踢"**);  
 }  
  
 **public void** strike() {  
 System.***out***.println(**"向前撞"**);  
 }  
  
 **public static void** main(String[] args) {  
 EnumWithInterface.***Day1***.kick();  
 EnumWithInterface.***Day1***.strike();  
 }  
}

**我们还可以通过接口来管理我们的枚举类:**

**通常我们在一个项目中会有很多个状态位，如：项目状态、订单状态、支付状态等等**

**下面通过接口枚举来管理我们的状态：**

#### 4. 枚举与switch

**public enum** EnumWithSwitch {  
  
 ***A***, ***B***, ***C***;  
  
 **public static void** main(String[] args) {  
  
 EnumWithSwitch e = EnumWithSwitch.***A***;  
  
 **switch** (e) {  
 **case *A***:  
 System.***out***.println(**"this is A"**);  
 **break**;  
 **case *B***:  
 System.***out***.println(**"this is B"**);  
 **break**;  
 **case *C***:  
 System.***out***.println(**"this is c"**);  
 **break**;  
 }  
 }  
}

## 3. 泛型

## 为什么需要RTTI（Run-Time Type Identification）

RTTI维护类型的信息，为多态机制提供基础

怎么为多态

## 反射

在整个反射的开发模式之中，有一个最重要的类：

* public final class Class<T>

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

### 获得Class类实例对象的三种方式

* **通过 对象.getClass() , 这个方法时Object中提供，所有类的实例化对象都可以调用。**
* **通过 类.class ，在一些开源框架中会大量使用到，例如：hibernate、mybatis**
* **利用Class中的forName方法，方法参数类全限定名称。这种方式可以用于工厂方法。**

### 对象实例化

#### **通过newInstance()方法实例化对象**

public [T](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Class.html) newInstance()

throws [InstantiationException](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/InstantiationException.html), -> **没有构造方法、类名错误**

[IllegalAccessException](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/IllegalAccessException.html) -> **构造方法私有化**

newInstance 是通过无参构造方法

**练习：**

**public static void** main(String[] args){  
 **try** {  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.Book1"**);  
 Object object = clazz.newInstance();  
 System.***out***.println(object);  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace(); *// 类找不到* } **catch** (IllegalAccessException e) {  
 e.printStackTrace(); *// 构造方法私有化* } **catch** (InstantiationException e) {  
 e.printStackTrace(); *//没有构造方法、类名错误* }  
}

**面试题：**

1. **谈谈通过new 实例化一个对象和通过反射实例化一个对象的区别？**

**如果只是单纯的创建一个类的实例对象，两者区别不大，非要说一个区别那就是反射实例化的灵活度要比new更高，如果运用到了多态模式，那么new这种方式造成了耦合性，解决这种方式最早是工厂设计模式**

**首先说说简单工厂：**

*/\*\*  
 \* 简单工厂  
 \*/***class** SimpleFactory{  
  
 **public static** Msg getNewInstance(String className){  
 **if** (**"News"**.equalsIgnoreCase(className)){  
 **return new** News();  
 }**else if**(**"Emails"**.equalsIgnoreCase(className)){  
 **return new** Emails();  
 }  
 **return null**;  
 }  
}

**简单工厂有一个非常大的缺陷，就是每当新增一个子类，都需要在工厂实例方法里面添加该类的构造代码。**

#### **通过**Constructor<T>类

**Constructor 提供构造函数信息的访问**

##### Class类中提供的获取构造类的方法

* **Constructor<?>[] getConstructors()**
* **Constructor<T> getConstructor(类<?>... parameterTypes)**

##### Constructor 类中的方法

* 获取构造方法名：**String getName()**
* 获取方法修饰符：**int getModifiers() 这个方法获取的是修饰符的数值，如果需要转换成关键字形式，那么需要通过调用**Modifier 类的toString(int mod)静态方法.
* 获取参数变量类型：**Class<?>[] getParameterTypes()**
* 获取异常：**Class<?>[] getExceptionTypes()**

##### Constructor类中方法练习

*/\*\*  
 \* Constructor类练习  
 \*/***public class** ConstructorDemo {  
  
 **public static void** main(String[] args) {  
 Class<?> clazz = Book.**class**;  
 Constructor<?>[] cons = clazz.getDeclaredConstructors();  
 **for** (Constructor<?> con : cons){  
 *//获取修饰符* System.***out***.print(Modifier.*toString*(con.getModifiers())+ **" "**);  
 *//获取构造方法名称* System.***out***.print(con.getName());  
 *//获取方法参数* System.***out***.print(**"("**);  
 Class<?>[] paramsType = con.getParameterTypes();  
 **for** (**int** i=0; i < paramsType.**length**; i++){  
 System.***out***.print(paramsType[i].getSimpleName() + **" arg"** + i);  
 **if** (i < paramsType.**length** - 1){  
 System.***out***.print(**", "**);  
 }  
 }  
 System.***out***.print(**")"**);  
 *//获取异常* Class<?>[] exceptions = con.getExceptionTypes();  
 **if** (exceptions.**length** > 0){  
 System.***out***.print(**" throws "**);  
 }  
 **for** (**int** i=0; i < exceptions.**length**; i++){  
 System.***out***.print(exceptions[i].getSimpleName());  
 **if** (i < paramsType.**length** - 1){  
 System.***out***.print(**", "**);  
 }  
 }  
 System.***out***.println();  
 }  
 }  
}  
**class** Book{  
  
 **private** String **name**;  
 **private** Double **price**;  
  
 **public** Book(String name, Double price)**throws** RuntimeException, NullPointerException {  
 **this**.**name** = name;  
 **this**.**price** = price;  
 }  
 **private** Book(String name) {  
 **this**.**name** = name;  
 }  
 **public** Book(){}  
  
 **public** String getName() {  
 **return name**;  
 }  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
 **public** Double getPrice() {  
 **return price**;  
 }  
 **public void** setPrice(Double price) {  
 **this**.**price** = price;  
 }  
}

##### 通过Constructor实例化对象

**public T newInstance(Object... initargs)**

**throws** **InstantiationException, -> 实例化异常**

**IllegalAccessException, ->非法方法**

**IllegalArgumentException, ->非法参数**

**InvocationTargetException -> 类错误**

### 普通方法调用

**获取本类声明的方法：**

* + - **Method** **getDeclaredMethod(String name, Class<?>... parameterTypes)**
    - **Method[] getDeclaredMethods()**

**获取本类的所有方法：**

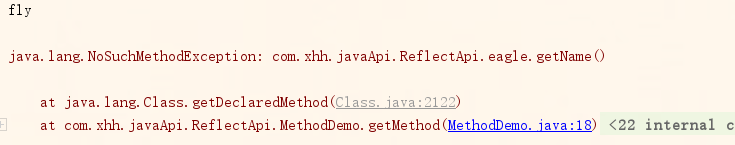
* + - **Method getMethod(String name, Class<?>... parameterTypes)**
    - **Method[] getMethods()**

#### 方法信息获取

**Method中的方法信息获取和构造方法一样，只不过多了一个返回类型**

**练习一下：**

@Test  
**public void** getMethod() **throws** Exception {  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.ReflectApi.eagle"**);  
 *// getDeclaredMethods()* Method[] methods = clazz.getDeclaredMethods();  
  
 **for** (Method m : methods){  
 System.***out***.println(m.getName());  
 }  
 Method method = clazz.getDeclaredMethod(**"getName"**);  
 System.***out***.println(method);  
}



**从这可以看出 getDeclaredMethod 方法和getDeclaredMethods方法只能获取到本类声明的方法，无法获取父类继承的方法。如果强行获取那么会抛出NoSuchMethodException异常。**

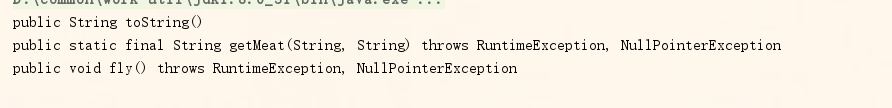
**另外一组方法：**

@Test  
**public void** getMethod2() **throws** Exception{  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.ReflectApi.eagle"**);  
  
 Method[] methods = clazz.getMethods();  
  
 **for** (Method m : methods){  
 System.***out***.println(m.getName());  
 }  
 Method method = clazz.getMethod(**"getName"**);  
 System.***out***.println(method.getName());  
}

**而这个两个方法可以获取继承来的方法。**

**查看方法信息：**

*/\*\*  
 \* 查看方法信息  
 \*/*@Test  
**public void** methodInfo() **throws** Exception{  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.ReflectApi.eagle"**);  
 Method[] methods = clazz.getDeclaredMethods();  
  
 **for** (**int** i=0; i < methods.**length**; i++){  
 *// 修饰符* System.***out***.print(Modifier.*toString*(methods[i].getModifiers()) +**" "**);  
 *//返回值类型* Class<?> returnType = methods[i].getReturnType();  
 System.***out***.print(returnType.getSimpleName() + **" "**);  
 *//方法名称* System.***out***.print(methods[i].getName());  
 *//方法参数* System.***out***.print(**"("**);  
 Class<?>[] paramTypes = methods[i].getParameterTypes();  
 **for** (**int** y=0; y< paramTypes.**length** ; y++){  
 System.***out***.print(paramTypes[y].getSimpleName());  
 **if** (y < paramTypes.**length** - 1){  
 System.***out***.print(**", "**);  
 }  
 }  
 System.***out***.print(**")"**);  
 *//异常信息* Class<?>[] exceptions = methods[i].getExceptionTypes();  
 **if** (exceptions.**length** > 0 ){  
 System.***out***.print(**" throws "**);  
 }  
 **for** (**int** j=0; j < exceptions.**length**; j++){  
 System.***out***.print(exceptions[j].getSimpleName());  
 **if** ( j < exceptions.**length** -1){  
 System.***out***.print(**", "**);  
 }  
 }  
 System.***out***.println();  
 }  
}



#### 方法调用

要想通过反射调用方法有三个前提条件:

* 1. **类确定**
  2. **方法名确定(在标准的JavaBean对象里面已知属性也可以)**
  3. **方法参数值确定**

*/\*\*  
 \* 测试方法调用  
 \*/*@Test  
**public void** callMethod() **throws** Exception{  
 String className = **"com.xhh.javaApi.ReflectApi.eagle"**;  
 String attr = **"name"**;  
 String val = **"老鹰"**;  
  
 Class<?> clazz = Class.*forName*(className);  
  
 Object obj = clazz.newInstance();  
  
 *//先设置值* Method method = clazz.getMethod(**"set"** + StringUtil.*initcap*(attr), String.**class**);  
 method.invoke(obj, val);  
 System.***out***.println(**"设置的值是："** + obj);  
  
 *//获取值* Method getMethod = clazz.getMethod(**"get"** + StringUtil.*initcap*(attr));  
 Object getVal = getMethod.invoke(obj);  
 System.***out***.println(**"获取的值是："** + getVal);  
}

### 4. 字段访问

在Class类中有两组方法访问类中的属性:

**第一组: 获取本类所有的字段包括私有的**

* + - **Field getDeclaredField(String name)**
    - **Field[] getDeclaredFields()**

**第二组：获取所有的公开（public修饰的）属性包括父类的**

* + - **Field getField(String name)**
    - **Field[] getFields()**

**那么怎么获取父类中私有的属性？**

*/\*\*  
 \* 查找字段  
 \** ***@param* clazz** *需要查找字段的所有类.class  
 \** ***@param* fieldName** *字段名  
 \** ***@return*** *找到返回Field，没找到返回Null  
 \*/***public static** Field getFiled(Class<?> clazz, String fieldName) {  
 **while** (clazz != **null**){  
 **try** {  
 Field field = clazz.getDeclaredField(fieldName);  
 **return** field;  
 } **catch** (NoSuchFieldException e) {  
 clazz = clazz.getSuperclass();  
 }  
 }  
 **return null**;  
}

**Field字段中的重要方法：**

* **get(Object obj); // 获取欧obj对象 该字段的值**
* **set(Object obj, Object value); // 给obj对象设置的该字段设置value**
* **getType(); // 获取字段类型**

**方法练习：**

@Test  
**public void** setField() **throws** Exception{  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.ReflectApi.Chinese"**);  
 Object obj = clazz.newInstance();  
 Field field = ReflectUtil.*getFiled*(clazz, **"name"**);  
 field.setAccessible(**true**);  
 field.set(obj, **"饶先龙"**);  
 System.***out***.println(**"设置的值："**+ obj);  
 System.***out***.println(**"获取的值:"** + field.get(obj));  
}

@Test  
**public void** getType() **throws** Exception{  
 Class<?> clazz = Class.*forName*(**"com.xhh.javaApi.ReflectApi.Chinese"**);  
 String attr = **"age"**;  
 Integer value = 23;  
  
 Object obj = clazz.newInstance();  
 Field field = ReflectUtil.*getFiled*(clazz, attr);  
 Method method = clazz.getMethod(**"set"** + StringUtil.*initcap*(attr), field.getType());  
 method.invoke(obj, value);  
 System.***out***.println(obj);  
}

## 6. 自定义MVC练习

**Github地址：**

<https://github.com/raoxianlong/learn-note/tree/learn_mymvc_20181220>

# 4. BIO

**Java IO的核心解释**：**如果抽象类或接口之中的抽象方法被子类覆写了，那么实例化这个子类的时候，所调用的方法一定是被覆写过的方法**

**在整个java.io包下面实际上就是五个类和一个接口：**

* **五个类：File、 InputStream、OutputStream、 Reader、Wirter;**
* **一个接口: Serializable**

## 1. 文件操作类 (File)

**在java.io包中File类是唯一的一个与文件本身操作有关的类。**

### 构造方法

|  |
| --- |
| [File](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/File.html#File-java.lang.String-)([String](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/String.html) pathname)  通过将给定的路径名字符串转换为抽象路径名来创建新的 File实例。 |
| [File](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/File.html#File-java.lang.String-java.lang.String-)([String](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/String.html) parent, [String](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/String.html) child)  从父路径名字符串和子路径名字符串创建新的 File实例。 |

*/\*\*  
 \* 创建文件  
 \*/*@Test  
**public void** creatFile() **throws** IOException {  
 File file = **new** File(**"D:"** + File.***separator*** + **"test.txt"**);  
 **if** (file.exists()){  
 file.delete();  
 }**else**{  
 file.createNewFile();  
 }  
}

### File的基础方法

#### **创建文件**

public boolean createNewFile() throws IOException

**抛IO异常的情况：**

**如果文件路径不存在**

**如果文件名称包含不合法字符**

#### **2. 删除文件**

public boolean delete()

#### **3. 查看文件是否存在**

public boolean exists()

#### **4. 取得文件名称**

public String getName()；

#### **5. 是否是文件**

public boolean isFile()

#### **6. 是否是文件夹**

public boolean isDirectory()

#### **7. 是否是隐藏文件**

public boolean isHidden()

#### **8. 文件最后一次修改日期**

Public long lastModified()

#### **9. 取得文件大小**

public long length()

#### **10. 获取父级目录**

public File getParentFile(); // 返回父级目录File文件形式

public String getParent(); // 返回父级目录Path

#### **11. 创建目录**

public boolean mkdir(); // 创建目录，该方法只能创建一级目录，不支持多级

public boolean mkdirs(); // 创建目录，可以创建多级目录

#### 12. 练习

*/\*\*  
 \* File基础方法  
 \*/*@Test  
**public void** fileBaseMethod() **throws** IOException {  
 File file = **new** File(**"D:"** + File.***separator*** +  
 **"raoxianlong"** + File.***separator*** + **"304318.jpg"**);  
 **if** (!file.exists()){

*// 因为该目录不存在，所以直接创建文件会抛 java.io.IOException: 系统找不到指定的路径。*  
 *// 所以在创建文件之前需要判断父路径是否存在* **if** (! file.getParentFile().exists()){  
  *// 不存在则需要创建父目录，注意这里如果用 mkdir()方法也是不会抛异常，  
 // 如果目录有多级则只能用mkdirs()，然后再创建文件* file.getParentFile().mkdirs();  
 }  
 file.createNewFile();  
 }  
 System.***out***.println(**"文件路径："** + file.getPath());  
 System.***out***.println(**"是否是文件："** + file.isFile()); *// 如果文件不存在，那么也会返回false* System.***out***.println(**"是否是文件夹："** + file.isDirectory());  
 System.***out***.println(**"是否是隐藏文件："** + file.isHidden());  
 System.***out***.println(**"文件/目录名称："** + file.getName());  
 System.***out***.println(**"文件大小:"** + **new** BigDecimal((**double**) file.length() / (1024 \* 1024)).  
 divide(**new** BigDecimal(1), 2, BigDecimal.***ROUND\_HALF\_UP***) + **"M"**);  
 System.***out***.println(**"文件最后一次修改时间:"** +  
 **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**).format(file.lastModified()));  
}

### 获取目录下的子文件

public File[] listFiles()

@Test  
**public void** fileList(){  
 File file = **new** File(**"d:"** + File.***separator***);  
 listFiles(file);  
}  
*/\*\*  
 \* 递归文件  
 \*/***private void** listFiles(File file){  
 **if** (file.isDirectory()){  
 File[] files = file.listFiles();  
 **if** (files != **null**){  
 System.***out***.printf(**"%1$-60s%2$-30s%3$-10s"**, **"文件名称"**, **"最后修改时间"**, **"文件大小"**);  
 System.***out***.println();  
 **for** (File f : files){  
 System.***out***.printf(**"%1$-60s%2$-30s%3$-10s"**, f.getName(),  
 **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**).format(file.lastModified())  
 , **new** BigDecimal((**double**) file.length() / (1024 \* 1024)).  
 divide(**new** BigDecimal(1), 2, BigDecimal.***ROUND\_HALF\_UP***) + **"M"**);  
 System.***out***.println();  
 }  
 }  
 }  
}

## 2. 字节流

**如果需要对文件里的内容做操作，那么就需要通过java中提供的两组方法：**

* **字节操作流【JDK1.0】：OutputStream、InputStream**
* **字符操作流【JDK1.1】：Writer、Reader**

**操作流的步骤：**

1. **需要知道流资源来源（可以是一个流、可以是一个文件等等）**
2. **通过字节流或字符流的子类创建字节流或字符流的实例对象**
3. **执行读/写操作**
4. **关闭流**

### OutputStream输出流

public abstract class OutputStream

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

implements [Closeable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Closeable.html), [Flushable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Flushable.html)

**OutputStream是所有字节输出流的抽象父类**

|  |  |
| --- | --- |
| **Closeable：JDK 1.5推出** | **Flushable：JDK 1.5推出** |
| **public** **interface** Closeable **extends** AutoCloseable {  **public** **void** close() **throws** IOException;  } | **public** **interface** Flushable {  **public** **void** flush() **throws** IOException;  } |

**提示：**对于Closeable继承的AutoCloseable接口

AutoCloseable是在JDK 1.7的时候又增加了一个新的接口，但是这个接口的定义和Closeable定义是完全一样的，我个人认为：有可能在一些其他的类上出现了自动的关闭功能，Closeable是手工关闭，AutoCloseable属于自动关闭。

但是对于Closeable和Flushable这两个接口实话而言用户不需要关注，因为从最早的习惯对于flush()和close()两个方法都是直接在OutputStream类之中定义的，所以很少去关心这些父接口问题。

对于OutputStream类而言发现其本身定义的是一个抽象类（abstract class），按照抽象类的使用原则来讲，需要定义抽象类的子类，而现在如果要执行的是文件操作，则可以使用FileOutputStream子类完成，如果按照面向对象的开发原则，子类要为抽象类进行对象的实例化，而后调用的方法以父类中定义的方法为主，而具体的实现找实例化这个父类的子类完成，也就是说在整个的操作之中，用户最关心的只有子类的构造方法。

**OutputStream中提供的方法：**

· **输出单个字节数据：**public abstract void write(**int** b) throws IOException；

· **输出一组字节数据：**public void write(**byte[] b**) throws IOException；

· **输出部分字节数据：**public void write(byte[] b, int off, int len) throws IOException；

* + **刷新流：**public void flush() throws IOException
  + **关闭流：**public void close() throws IOException

#### 1. 子类FileOutputStream(操作文件)

public class FileOutputStream

extends [OutputStream](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/OutputStream.html)

**有关文件内容的输出操作，使用该类**

##### 构造方法

**指定文件**：public FileOutputStream(File file) throws FileNotFoundException

**文件追加：**public FileOutputStream(File file,boolean append)throws FileNotFoundException

**文件路径：**public FileOutputStream(String name) throws FileNotFoundException

**追加：**public FileOutputStream(String name,boolean append) throws FileNotFoundException

##### 方法练习

@Test  
**public void** test() **throws** Exception {  
 *// 1.通过File定义一个文件资源* File file = **new** File(**"D:"** + File.***separator*** + **"raoxianlong"** + File.***separator*** + **"test.txt"**);  
 **if** (!file.getParentFile().exists()){  
 file.getParentFile().mkdirs();  
 }  
 *// 2.通过FileOutputStream创建OutputStream的子类* OutputStream out = **new** FileOutputStream(file);  
  
 *// 3.写入数据* **byte**[] test = **"饶先龙是傻逼66666!"**.getBytes();  
  
 *// 3.1 通过单个字节写入  
 /\*for (byte b : test){  
 out.write(b);  
 }\*/  
 // 3.2 通过全部字节写入  
 /\*out.write(test);\*/  
 // 3.3 通过部分字节写入 off 开始写入字节的位置 len 写入的字节长度 汉字占两个字节* out.write(test, 6, 6);  
  
 *// 4.关闭流* out.close();  
}

#### 2. 子类ByteArrayOutputStream (操作内存)

##### 构造方法

* **创建一个容量为32字节的缓充输出流 : public ByteArrayOutputStream()**
* **自定义缓充容量：public ByteArrayOutputStream(int size)**

##### 方法练习

###### 将一串字符串由小写转为大写

*/\*\*  
 \* 内存流测试  
 \*/*@Test  
**public void** test() **throws** IOException {  
 **byte**[] bytes = **"Hello World -Rxl wfwejlkjlkjs wefjklwwjljl!"**.getBytes();  
 InputStream in = **new** ByteArrayInputStream(bytes);  
 OutputStream out = **new** ByteArrayOutputStream();  
 **int** temp;  
 **while** ((temp = in.read()) != -1){  
 out.write((**char**)Character.*toUpperCase*(temp));  
 }  
 String str = ((ByteArrayOutputStream) out).toString();  
 in.close();  
 out.close();  
 System.***out***.println(str);  
  
}

#### 装饰者模式FilterOutputStream(Filter模式)

* 1. 底层依然是OutputStream,
  2. 需要接受一个OutputStream
  3. 提供一系列附加功能。
  4. 为了解决流子类爆炸的问题。
  5. 自身不负责与文件或其他源打交道，需要依赖底层流。

##### 1. 子类BufferedOutputStream (缓冲区)

**BufferedOutputStream提供带缓冲功能的写操作，防止一个一个写减少物理IO的频率**

**提高效率**

###### 构造方法

* **BufferedOutputStream(OutputStream out);** //使用默认大小、底层字节输出流构造bos。默认缓冲大小是 8192 字节( 8KB )
* **BufferedOutputStream(OutputStream out, int size);**  //使用指定大小、底层字节输出流构造bos

##### 2. 子类ZipOutputStream(压缩)

**使用****ZipOutputStream对文件进行压缩**

**ZipOutputStream类接受一个OutputStream流作为写入，其底层压缩依赖**

**Deflater类**

*/\*\*  
 \* 压缩  
 \*/*@Test  
**public void** compress() **throws** IOException {  
 ZipOutputStream zipOutputStream = **new** ZipOutputStream(**new** FileOutputStream(**new** File(*file*, **"demo.zip"**)));  
 compress(*file*, zipOutputStream, **""**);  
 zipOutputStream.close();  
}  
  
*/\*\*  
 \* 递归压缩文件  
 \** ***@param* f** *压缩文件目录  
 \** ***@param* zipOutputStream** *压缩流  
 \** ***@param* parentPath** *压缩文件的父路径  
 \** ***@throws*** *IOException  
 \*/***private void** compress(File f, ZipOutputStream zipOutputStream, String parentPath) **throws** IOException {  
 **if** (f.isDirectory()){  
 parentPath += f.getName() + File.***separator***;  
 **for** (File f1 : f.listFiles()){  
 **if** (f1.getName().equals(**"demo.zip"**)){  
 **continue**;  
 }  
 compress(f1, zipOutputStream, parentPath);  
 }  
 }**else**{  
 InputStream in = **new** FileInputStream(f);  
 ZipEntry zipEntry = **new** ZipEntry(parentPath + f.getName());  
 zipOutputStream.putNextEntry(zipEntry);  
  
 **byte**[] buffer = **new byte**[1024];  
 **int** len;  
 **while** ((len = in.read(buffer)) != -1){  
 zipOutputStream.write(buffer, 0, len);  
 }  
 in.close();  
 zipOutputStream.closeEntry();  
 }  
}

### InputStream输入流

public abstract class InputStream

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

implements [Closeable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Closeable.html)

**InputStream中提供的方法：**

* + **读取单个字节:**  **public abstract int read() throws IOException**
  + **读取b.length个字节: public int read(byte[] b) throws IOException**
    - **该方法会将读取的数据缓存到b数组汇中, 相当于 read(b, 0, b.length);**
    - **该方法返回读取字节的长度, 如果没有内容可读，返回 -1**
  + **读取指定长度的字节：public int read(byte[] b,int off,int len) throws IOException**

#### 1. 子类FileInputStream(操作文件)

##### 构造方法

**指定文件**：public FileInputStream(File file) throws FileNotFoundException

**文件路径：**public FileInputStream(String name) throws FileNotFoundException

##### 方法练习

###### 单个字节读取

*/\*\*  
 \* 单个字节读取  
 \** ***@throws*** *FileNotFoundException  
 \*/*@Test  
**public void** readOne() **throws** IOException {  
 InputStream in = **new** FileInputStream(*file*);  
 **byte**[] data = **new byte**[1024];  
 **int** temp;  
 **int** flag = 0;  
 **while** ((temp=in.read()) != -1){  
 data[flag++] = (**byte**) temp;  
 }  
 in.close();  
}

###### 全部字节读取

*/\*\*  
 \* 全部字节读取:  
 \* 要注意的是读取的长度，跟缓存数组的长度有关，  
 \* 数组长度大于输入流中字节的长度则可以全部读完。  
 \* 相当于：read(data, 0, data.length);  
 \*/*@Test  
**public void** readAll() **throws** IOException {  
 InputStream in = **new** FileInputStream(*file*);  
 **byte**[] data = **new byte**[1024];  
 **int** len = in.read(data);  
 in.close();  
 System.***out***.println(**new** String(data, 0, len));  
}

###### 部分字节读取

*/\*\*  
 \* 读取部分字节  
 \*/*@Test  
**public void** readMany() **throws** IOException {  
 InputStream in = **new** FileInputStream(*file*);  
  
 **byte**[] temp = **new byte**[2];  
 **byte**[] data = **new byte**[1024];  
  
 **int** off = 0; *// 角标* **int** foot = 0; *// 角标* **int** len; *// 定义每次读取两个字节* **while** ((len = in.read(temp, off, 2)) != -1){  
 **for** (**int** i=0; i < len;i++){  
 data[foot++]=temp[i];  
 }  
 }  
 in.close();  
 System.***out***.println(**new** String(data, 0, foot));  
}

###### 文件拷贝

*/\*\*  
 \* 文件拷贝  
 \*/*@Test  
**public void** copy() **throws** IOException {  
 InputStream in = **new** FileInputStream(*file*);  
 File outFile = **new** File(*file*.getParentFile(), **"test\_copy.txt"**);  
  
 OutputStream out = **new** FileOutputStream(outFile);  
  
 **byte**[] data = **new byte**[1024];  
 **int** len = 0;  
 **while** ((len = in.read(data)) != -1){  
 out.write(data, 0, len);  
 }  
 in.close();  
 out.close();  
}

#### 子类ByteArrayInputStream (操作内存)

##### 构造方法

* **使用buf作为数据源：public ByteArrayInputStream(byte[] buf)**
* **截取buf数组中的数据作为数据源：public ByteArrayInputStream(byte[] buf, int offset, int length)**

#### 装饰者模式FilterInputStream(Filter模式)

* 1. 底层依然是InputStream,
  2. 需要接受一个InputStream
  3. 提供一系列附加功能。
  4. 为了解决流子类爆炸的问题。
  5. 自身不负责与文件或其他源打交道，需要依赖底层流。

##### 1. 子类BufferedInputStream(缓冲区)

BufferedInputStream提供带缓冲功能的输入流，以及支持mark()标记和reset()重置方法

###### 构造方法

* **BufferedInputStream(InputStream in)** *//使用默认buf大小、底层字节输入流构建bis*
* **BufferedInputStream(InputStream in, int size)** *//使用指定buf大小、底层字节输入流构建bis*

###### 练习

*/\*\*  
 \* 通过缓冲区实现文件拷贝  
 \*/*@Test  
**public void** copy(){  
 **long** start = System.*currentTimeMillis*();  
 InputStream in = **null**;  
 OutputStream out = **null**;  
 **try** {  
 in = **new** BufferedInputStream(**new** FileInputStream(*file*));  
 File outFile = **new** File(*file*.getParentFile(), **"copyBuffered.txt"**);  
 out = **new** BufferedOutputStream(**new** FileOutputStream(outFile));  
 **byte**[] data = **new byte**[1024];  
 **int** len;  
 **while** ((len = in.read(data)) != -1){  
 out.write(data, 0, len);  
 }  
 out.close();  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }**finally** {  
 **if** (in != **null**){  
 **try** {  
 in.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **if** (out != **null**){  
 **try** {  
 out.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 **long** end = System.*currentTimeMillis*();  
 System.***out***.println(**"耗时:"** + (end - start) + **"ms"**);  
}

##### 2. 子类ZipInputStream(解压缩)

**使用ZipInputStream对文件进行压缩**

**ZipInputStream类接受一个InputStream流作为读取，其底层解压缩依赖**

**Inflater类**

*/\*\*  
 \* 解压缩  
 \*/*@Test  
**public void** decompression() **throws** IOException {  
 File dcFile = **new** File(*file*, **"demo.zip"**); *// 压缩文件* File pFile = **new** File(dcFile.getParentFile(), **"dDemo"**); *// 解压缩到该文件夹下* ZipInputStream zipInputStream = **new** ZipInputStream(**new** FileInputStream(dcFile));  
 decompression(zipInputStream, pFile);  
}  
  
*/\*\*  
 \* 解压缩文件  
 \** ***@param* zipInputStream** *\** ***@throws*** *IOException  
 \*/***private void** decompression(ZipInputStream zipInputStream, File pFile) **throws** IOException {  
 ZipEntry zipEntry;  
 **while** ((zipEntry = zipInputStream.getNextEntry()) != **null**){  
 File outFile = **new** File(pFile, zipEntry.getName());  
 **if** (!outFile.getParentFile().exists()){  
 outFile.getParentFile().mkdirs();  
 }  
 OutputStream out = **new** FileOutputStream(outFile);  
 **byte**[] buffer = **new byte**[1024 \* 100];  
 **int** len;  
 **while** ((len = zipInputStream.read(buffer)) != -1){  
 out.write(buffer, 0, len);  
 }  
 zipInputStream.closeEntry();  
 out.close();  
 }  
 zipInputStream.close();  
}

## 字符流

### 1. Writer字符输出流

public abstract class Writer

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

implements [Appendable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Appendable.html), [Closeable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Closeable.html), [Flushable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Flushable.html)

**Writer抽象类中提供的方法：**

* **写入单个字符：public void write(int c) throws IOException**
* **写入字符串：public void write(String str) throws IOException**
* **写入部门字符串: public void write(String str,int off,int len) throws IOException**
* **写入字符数组：public void write(char[] cbuf) throws IOException**
* **写字符数组部分:public abstract void write(char[] cbuf, int off, int len) throwsIOException**
* **追加一个字符: public Writer append(char c) throws IOException**
* **追加一个字符序列：public Writer append(CharSequence csq) throws IOException**
* **追加部分字符序列：public Writer append(CharSequence csq, int start,int end)throws IOException**

#### 子类FileWriter(操作文件)

##### 构造方法

* **指定文件：public FileWriter(File file) throws IOException**
* **指定文件是否追加：public FileWriter(File file, boolean append) throws IOException**
* **指定文件路径：public FileWriter(String fileName) throws IOException**
* **追加：public FileWriter(String filename, blooean append) throws IOException**

##### 方法练习

###### 写入一个字符

*/\*\*  
 \* 写入一个字符  
 \*/*@Test  
**public void** writerChar() **throws** IOException {  
 Writer writer = **new** FileWriter(*file*, **true**);  
  
 **char**[] chars = **"饶先龙是傻逼"**.toCharArray();  
  
 **for** (**char** c : chars){  
 writer.write(c);  
 }  
 *// 关闭之前刷新* writer.close();  
}

###### 写入字符串

*/\*\*  
 \* 写入字符串  
 \*/*@Test  
**public void** writerStr() **throws** IOException {  
 Writer writer = **new** FileWriter(*file*, **true**);  
  
 String str = **"饶先龙是大傻逼"**;  
  
 writer.write(str);  
 *// 关闭之前刷新* writer.close();  
}

###### 追加字符串

*/\*\*  
 \* 追加字符串  
 \* 必须在构造时设置是否追加  
 \*/*@Test  
**public void** appendStr() **throws** IOException {  
 Writer writer = **new** FileWriter(*file*, **true**);  
  
 String str = **"饶先龙是大傻逼"**;  
  
 writer.append(str);  
 *// 关闭之前刷新* writer.close();  
}

#### 子类OutputStreamWriter(转换流)

**我们可以使用两个类来将字节流转换为字符流**

* + 将字节输出流转换为字符输出流：**OutputStreamWriter** (**Writer的子类**)

**实例：**

*/\*\*  
 \* 字节输出流转换为字符输出流  
 \*/*@Test  
**public void** output2Writer() **throws** IOException {  
 OutputStream out = **new** FileOutputStream(*file*);  
 String str = **"字节流、字符流、转换流"**;  
 Writer writer = **new** OutputStreamWriter(out);  
 writer.append(str);  
 writer.close();  
}

#### 子类BufferedWriter(缓冲区)

BufferedWriter 提供缓冲的功能，可以更加高效的写入字符。

###### 构造方法

* BufferedWriter(Writer out) 创建使用默认大小的输出缓冲区的缓冲字符输出流。
* BufferedWriter(Writer out, int sz) 创建一个新的缓冲字符输出流，使用给定大小的输出缓冲区。

### 2. Reader字符输入流

public abstract class Reader

extends [Object](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Object.html)

implements [Readable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/lang/Readable.html), [Closeable](mk:@MSITStore:D:\my\rxl\API\jdk%20api%201.8_google.CHM::/java/io/Closeable.html)

**Reader抽象类中提供的方法：**

* **读取单个字符：public void read() throws IOException**
* **读取多个字符：public int read(char[] cbuf) throws IOException**
  + **读取的大小跟cbuf字符数组的大小一样**
  + **如果没有内容可读，返回 -1**
  + **读取的数据缓存到cbuf中**
* **读取指定长度的字符：public abstract int read(char[] cbuf,int off,int len)throws IOException**

**注意：**

**在Writer类之中存在了直接输出一个字符串数据的方法，可是在Reader类之中并没有定义这样的方法，只是定义了三个按照字符读取的方法，为什么会这样？**

**原因:**

因为在使用OutputStream输出数据的时候，其程序可以输出的大小一定是程序可以承受的数据大小，那么如果说使用InputStream读取的时候，可能被读取的数据非常的大，那么如果一次性全读进来了，就会出现问题，所以只能一个一个的进行读取。

#### 子类FileReader(操作文件)

##### 构造方法

* **指定文件路径：public FileReader(String fileName) throws FileNotFoundException**
* **指定文件： public FileReader(File file) throws FileNotFoundException**

##### 方法练习

###### 单个字符读取

*/\*\*  
 \* 单个字符读取  
 \*/*@Test  
**public void** readOne() **throws** IOException {  
 Reader reader = **new** FileReader(*file*);  
  
 StringBuilder stringBuilder = **new** StringBuilder();  
  
 **int** temp;  
  
 **while** ((temp = reader.read()) != -1){  
 stringBuilder.append((**char**) temp);  
 }  
 System.***out***.println(stringBuilder.toString());  
}

###### 多个字符读取

*/\*\*  
 \* 读取多个字符  
 \*/*@Test  
**public void** readMany() **throws** IOException{  
 Reader reader = **new** FileReader(*file*);  
  
 StringBuilder stringBuilder = **new** StringBuilder();  
  
 *// 每次读取4个字符* **char**[] temp = **new char**[4];  
 **int** len;  
  
 **while** ((len = reader.read(temp)) != -1){  
 stringBuilder.append(temp, 0, len);  
 }  
 System.***out***.println(stringBuilder.toString());  
}

#### 子类InputStreamReader(转换流)

**我们可以使用两个类来将字节流转换为字符流**

* + 将字节输入流转换为字符输入流：**InputStreamReader (Reader的子类)**

**实例：**

*/\*\*  
 \* 字节输入流转换为字符输入流  
 \*/*@Test  
**public void** input2Reader() **throws** IOException {  
 InputStream inputStream = **new** FileInputStream(*file*);  
 **char**[] chars = **new char**[1024 \* 1024];  
 Reader reader = **new** InputStreamReader(inputStream);  
 **int** len = reader.read(chars);  
 reader.close();  
 System.***out***.println(**new** String(chars, 0 ,len));  
}

#### 3. 子类BufferedReader(缓冲区)

BufferedReader提供缓冲的功能，可以高效的读取字符串数据。

###### 构造方法

* BufferedReader(Reader in) 创建使用默认大小的输入缓冲区的缓冲字符输入流。
* BufferedReader(Reader in, int sz) 创建使用指定大小的输入缓冲区的缓冲字符输入流。

###### 方法

除了普通的读取方法外，BufferedReader还提供了一个特别的方法：

public String readLine() throws IOException

它提供读取一行文本的操作，而且返回的是String类型，非常方便开发者使用。

###### 方法练习

**public static void** main(String[] args) **throws** IOException {  
 BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
 **boolean** exit = **false**;  
  
 **while** (!exit){  
 System.***out***.print(**"请输入姓名:"**);  
 String name = reader.readLine();  
 **if** (StringUtil.*isNotBlank*(name)){  
 System.***out***.println(**"您的名字是："** + name);  
 exit = **true**;  
 }  
 }  
}

## 字节流和字符流的区别

### 区别

**字节流在进行IO操作的时候，直接针对的是操作的数据终端(例如：文件), 而字符流操作的时候不是直接针对终端，而是针对缓存区(内存)操作，而后由缓存取操作终端，属于间接操作，这样操作在不关闭字符流的情况下，那么缓存区的内容也不会清空。这个时候可以使用flush()手动清空缓存。**

### 使用场景

**字节流：处理图片、电影、音乐等二进制数据时**

**字符流：处理中文**

## 字符编码

### **常见编码**

* **GBK/GB2312 ： 表示国标中文编码，其中GBK包含简体中文和繁体中文GB2312只包含简体中文**
* **ISO8859-1 ： 是一种国际通用编码，可以表示任何文字，但是对于窗格需要转码**
* **Unicode ： 使用了十六进制编码，可以准备的表示任何语言文字**
* **UTF(UTF-8) : 部分编码使用ISO8859-1适用于网络传输，部分使用Unicode**

### **乱码**

**乱码的本质上的原因就是：编码和解码的字符集不统一**

## 6. 打印流

### 1. 自定义打印流

**package** com.xhh.javaApi.io;  
  
  
**import** java.io.\*;  
**import** java.util.Formatter;  
**import** java.util.Locale;  
  
*/\*\*  
 \* 自定义打印流  
 \*/***public class** PrintUtil **extends** FilterOutputStream **implements** Appendable{  
  
 **private** Formatter **formatter**;  
  
 **public** PrintUtil(OutputStream out){  
 **super**(out);  
 **formatter** = **new** Formatter(**this**, Locale.*getDefault*());  
 }  
  
 **public** PrintUtil(File file) **throws** FileNotFoundException {  
 **super**(**new** FileOutputStream(file));  
 **formatter** = **new** Formatter(**this**, Locale.*getDefault*());  
 }  
  
 **private void** writer(String str){  
 **try** {  
 **this**.**out**.write(str.getBytes());  
 **this**.**out**.flush();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **public void** print(**int** b){  
 writer(String.*valueOf*(b));  
 }  
  
 **public void** print(**byte** b){  
 writer(String.*valueOf*(b));  
 }  
  
 **public void** print(Character b){  
 writer(String.*valueOf*(b));  
 }  
  
 **public void** print(String str){  
 writer(str);  
 }  
  
 **public void** printf(String fmt, Object... vars){  
 **formatter**.format(fmt, vars);  
 }  
  
 @Override  
 **public** PrintUtil append(CharSequence csq) **throws** IOException {  
 writer(csq.toString());  
 **return this**;  
 }  
  
 @Override  
 **public** PrintUtil append(CharSequence csq, **int** start, **int** end) **throws** IOException {  
 writer(csq.subSequence(start, end).toString());  
 **return this**;  
 }  
  
 @Override  
 **public** PrintUtil append(**char** c) **throws** IOException {  
 print(c);  
 **return this**;  
 }  
  
}

**在实际使用过程中，我们发现使用OutputStream的wirte方法进行输出太过繁琐，那么我们可以对OutputStream进行装饰一下让打印更加便捷，上面是自定义的打印流。**

### JDK提供的打印流

**JDK实际上也提供了一组打印流来方便开发者使用。**

**原理：**

**打印流内部本质上还是需要依赖OutputStream的子类来完成，内部还是调用的write方法，只不过是进行了一层包装。这个就是装饰者模式。**

#### 字节打印流PrintStream

**内部提供了一系列的print方法，便捷输出。**

#### 字符打印流PrintWriter

**和PrintStream一样只不过PrintWriter方便中文输出。**

### 练习

**public static void** main(String[] args) **throws** IOException {  
*// PrintUtil print = new PrintUtil(System.out);* PrintUtil print = **new** PrintUtil(**new** File(**"D:"** + File.***separator*** + **"raoxianlong"** + File.***separator*** + **"testPrint.txt"**));  
 print.printf(**"%88s"**, **"饶先龙是傻逼"**);  
 print.print(**"a"**);  
 print.print(3);  
 print.print(**"饶先龙是傻逼222"**);  
 }

## 9. Filter模式

**Filter模式是为了解决子类数量爆炸的问题。**

**直接提供数据的InputStream：**

**FileInputStream**

**ByteArrayInputStream**

**ServletInputStream**

**提供附加功能的InputStream从FilterInputStream派生**

**BufferedInputStream**

**DigestInputStream**

**CipherInputStream**

**GZIPInputStream**

**Filter模式又称Decorator模式，通过少量的类实现了各种功能的组合。**

**FilterOutputStream和FilterInputStream类似**

## 10. 扫描流

Scanner 类是作为InputStrem类的一个方便读取数据的一个补充。

### 构造方法

* Scanner(File source) 通过一个文件
* Scanner(InputStream source) 通过一个输入流

### 方法

**有两大组方法：**

* + - hasNext();
    - hasNextXxx();
    - next();
    - nextXxx();

**另外next()方法和nextLine()的区别**

**next()方法：**

* **一定要读取到有效字符后才可以结束输入。**
* **对输入有效字符之前遇到的空白，next() 方法会自动将其去掉。**
* **只有输入有效字符后才将其后面输入的空白作为分隔符或者结束符。**
* **next() 不能得到带有空格的字符串。**

**nextLine()方法：**

* **以Enter为结束符,也就是说 nextLine()方法返回的是输入回车之前的所有字符；**
* **可以获得空白，都会读入，空格等均会被识别；**

### 方法练习

*/\*\*  
 \* 扫描流  
 \*/***public class** ScannerDemo {  
  
 **public static void** main(String[] args) {  
 Scanner scanner = **new** Scanner(System.***in***);  
  
 System.***out***.println(**"请输入内容:"**);  
 **if** (scanner.hasNext()){  
 System.***out***.println(**"输入的内容是:"** + scanner.nextLine());  
 }  
 }  
  
}

## 11. 对象序列化和反序列化

### 1. ObjectOutputStream序列化

我们使用ObjectOutputStream对对象进行序列化操作

#### 构造方法

ObjectOutputStream(OutputStream in)

### 2. ObjectInputStream反序列化

#### 构造方法

ObjectInputStream(InputStream in)

#### 练习

**package** com.xhh.javaApi.bio;  
  
  
**import** java.io.\*;  
  
*/\*\*  
 \* 序列化  
 \*/***public class** SerializeDemo {  
  
 **static** File *file* = **new** File(**"D:"** + File.***separator*** + **"raoxianlong"**);  
  
 **public static void** main(String[] args) **throws** IOException, ClassNotFoundException {  
 *// 序列化* ObjectOutputStream out = **new** ObjectOutputStream(**new** FileOutputStream(*file* + File.***separator*** + **"book.txt"**));  
 Book book = **new** Book(**"Java编程思想"**, 58.8);  
 out.writeObject(book);  
 out.close();  
 *// 反序列化* ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream(*file* + File.***separator*** + **"book.txt"**));  
 Book book1 = (Book) in.readObject();  
 System.***out***.println(book1);  
 }  
  
}  
  
**class** Book **implements** Serializable{  
  
 String **name**;  
 **double price**;  
  
 **public** Book(String name, **double** price) {  
 **this**.**name** = name;  
 **this**.**price** = price;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public double** getPrice() {  
 **return price**;  
 }  
  
 **public void** setPrice(**double** price) {  
 **this**.**price** = price;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Book{"** +  
 **"name='"** + **name** + **'\''** +  
 **", price="** + **price** +  
 **'}'**;  
 }  
}

# 5. NIO

## 1. 缓冲区(Buffer)

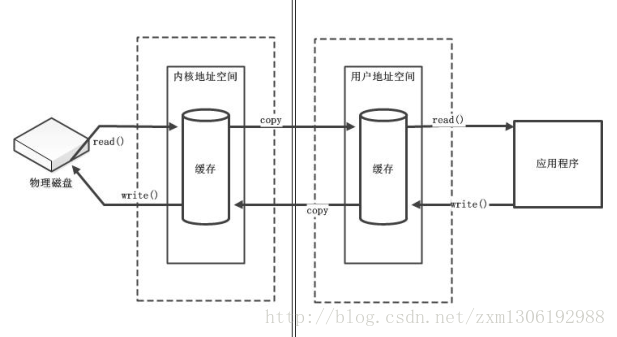
**缓冲区在Java Nio中负责数据的存储。缓冲区就是一个数组，用于存储不同的数据类型的数据**

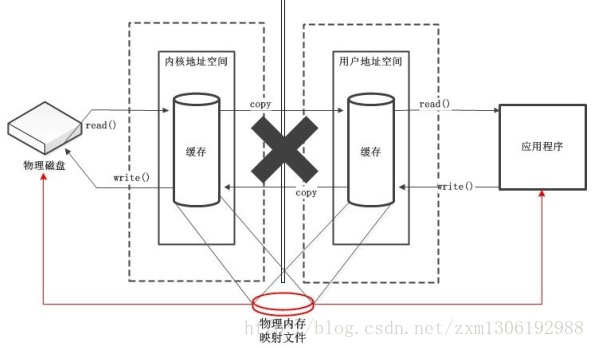
**根据数据类型（boolean除外），提供了相应类型的缓冲区：**

* **ByteBuffer**
* **CharBuffer**
* **ShortBuffer**
* **IntBuffer**
* **LongBuffer**
* **FloatBuffer**
* **DoubleBuffer**

### 1. 直接缓冲区和非直接缓冲区

* 非直接缓冲区：通过allocate()方法创建，缓冲区建立在JVM的内存中。
* 直接缓冲区： 通过allocateDirect()方法创建，缓冲区可以创建在物理内存中。





### 2. 缓冲区的核心属性

* **capactity : 容量，表示缓冲区最大的数据存储容量。一旦声明不能改变。**
* **limit : 界限，表示缓冲区可以操作数据的大小。**
* **position：位置，表示缓冲区正在操作数据的位置。**
* **mark ： 标记，表示记录当前position的位置。可以通过reset()恢复到mark()的位置。**

**位置关系:**

**mark <= position <= limit <= capcatity**

**当mark的位置大于position时，mark位置会清除变为 -1;**

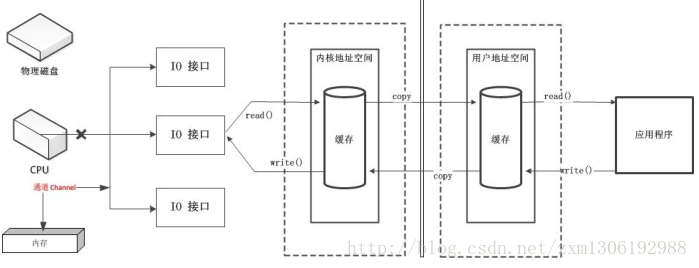
### 3. 缓冲区的方法

* + **static allocate() ：创建一个缓冲区,可以给缓冲区指定大小。**
  + **put() : 存入数据到缓冲区。**
    - **每put一个位置，position位置 +1, limit和capactity位置不变**
  + **get()：获取缓冲区的数据。**
  + **flip() : 切换读取数据模式。**
    - **切换到flip()模式之后，position位置变为0，limit位置变为之前存储的position位置。**
  + **rewind() : 清除读取的位置，可重复读。**
  + **clear() : 恢复到开始的位置，但是数据不会清空。**
  + **mark() : 打一个标记，通过reset()恢复到mark()的位置。**

### 4. 方法练习

@Test  
**public void** testBuffer(){  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
  
 *// put 数据* buffer.put(**"abc"**.getBytes());  
 System.***out***.println(**"----------------------put--------------------------"**);  
 System.***out***.println(**"position :"** + buffer.position());  
 System.***out***.println(**"limit :"** + buffer.limit());  
 System.***out***.println(**"capacity :"** + buffer.capacity());  
  
 *// flip() 切换到数据读取模式* buffer.flip();  
 System.***out***.println(**"----------------------flip--------------------------"**);  
 System.***out***.println(**"position :"** + buffer.position());  
 System.***out***.println(**"limit :"** + buffer.limit());  
 System.***out***.println(**"capacity :"** + buffer.capacity());  
  
 *// get() 获取数据* **byte**[] temp = **new byte**[buffer.limit()];  
 buffer.get(temp);  
 System.***out***.println(**"----------------------get--------------------------"**);  
 System.***out***.println(**"position :"** + buffer.position());  
 System.***out***.println(**"limit :"** + buffer.limit());  
 System.***out***.println(**"capacity :"** + buffer.capacity());  
 System.***out***.println(**"获取的数据是："** + **new** String(temp, 0, buffer.limit()));  
 *// 判断缓冲区是否还有数据* System.***out***.println(**"是否还有数据："** + buffer.hasRemaining());  
  
 *// mark() 标记* buffer.mark();  
 *// 清楚当前位置 rewind clear 会清除标记的位置  
 // buffer.clear();  
 // 重复读* buffer.rewind();  
 System.***out***.println(**"----------------------rewind--------------------------"**);  
 System.***out***.println(**"position :"** + buffer.position());  
 System.***out***.println(**"limit :"** + buffer.limit());  
 System.***out***.println(**"capacity :"** + buffer.capacity());  
  
 *// 恢复到标记的位置* buffer.reset();  
 System.***out***.println(**"----------------------reset--------------------------"**);  
 System.***out***.println(**"position :"** + buffer.position());  
 System.***out***.println(**"limit :"** + buffer.limit());  
 System.***out***.println(**"capacity :"** + buffer.capacity());  
  
}

## 2. 通道(Channce)

通道用于源节点与目标节点的连接。在Java NIO中负责数据缓冲区中的数据传输。Channel本身不负责存储数据，因此需要配合缓冲区进行传输。

### 1. 通道的主要实现类

Channel 接口

|- FileChannel

|- SocketChannel

|- ServerSocketChannel

|- DatagramChannel

### 2. 获取通道

#### 1. Java针对支持通道的类提供了getChannel()方法

本地IO：

FileInputStream/FileOutputStream

RandomAccessFile

网络IO：

Socket

ServerSocket

DatagramSocket

#### 2. 在JDK1.7中的NIO.2 针对各个通道提供了open();方法

#### 3. 在JDK1.7中的NIO.2的Files工具类中的newBytes();

### 3. 练习

#### 1. 使**用非直接缓冲区对文件进行复制**

*/\*\*  
 \* 使用通道进行文件复制  
 \*/*@Test  
**public void** copy(){  
 FileInputStream inputStream = **null**;  
 FileOutputStream outputStream = **null**;  
 FileChannel inChannel = **null**;  
 FileChannel outChannel = **null**;  
 **try** {  
 inputStream = **new** FileInputStream(**new** File(*file*, **"b.jpg"**));  
 File file1 = **new** File(*file*,**"c.jpg"**);  
 outputStream = **new** FileOutputStream(file1);  
  
 inChannel = inputStream.getChannel();  
 outChannel = outputStream.getChannel();  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 **while** (inChannel.read(buffer) != -1){  
 buffer.flip();  
 outChannel.write(buffer);  
 buffer.clear();  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **if** (outChannel != **null**){  
 **try** {  
 outChannel.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **if** (inChannel != **null**){  
 **try** {  
 inChannel.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **if** (outputStream != **null**){  
 **try** {  
 outputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **if** (inputStream != **null**){  
 **try** {  
 inputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

#### 2. 使用直接缓冲区对文件进行复制

*/\*\*  
 \* 通过直接内存进行Copy操作  
 \*/*@Test  
**public void** directCopy(){  
 FileChannel inChannel = **null**;  
 FileChannel outChannel = **null**;  
  
 **try** {  
 inChannel = FileChannel.*open*(Paths.*get*(**"D:"**,**"raoxianlong"**, **"b.jpg"**), StandardOpenOption.***READ***);  
 outChannel = FileChannel.*open*(Paths.*get*(**"D:"**,**"raoxianlong"**, **"d.jpg"**),StandardOpenOption.***READ***, StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);  
  
 MappedByteBuffer inBuffer = inChannel.map(FileChannel.MapMode.***READ\_ONLY***,0, inChannel.size());  
 MappedByteBuffer outBuffer = outChannel.map(FileChannel.MapMode.***READ\_WRITE***,0, inChannel.size());  
  
 **byte**[] temp = **new byte**[inBuffer.limit()];  
 inBuffer.get(temp);  
 outBuffer.put(temp);  
  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }**finally** {  
 **if** (outChannel != **null**){  
 **try** {  
 outChannel.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **if** (inChannel != **null**){  
 **try** {  
 inChannel.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

#### 3. 通道之间的数据传输

*/\*\*  
 \* 通道之间的数据传输(直接内存缓冲区)  
 \*/*@Test  
**public void** transfer() **throws** IOException {  
 FileChannel in = FileChannel.*open*(Paths.*get*(**"D:"**,**"raoxianlong"**, **"b.jpg"**), StandardOpenOption.***READ***);  
 FileChannel out = FileChannel.*open*(Paths.*get*(**"D:"**,**"raoxianlong"**, **"e.jpg"**), StandardOpenOption.***READ***, StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);  
 in.transferTo(0, in.size(), out);  
 in.close();  
 out.close();  
}

#### 4. 分散读取和聚集写入

*/\*\*  
 \* 分散读取和聚集写入  
 \*/*@Test  
**public void** disperse() **throws** IOException {  
 *// 1. 获取渠道(三种方式)* RandomAccessFile randomAccessFile = **new** RandomAccessFile(**new** File(*file*, **"222.txt"**), **"rw"**);  
 FileChannel fileChannel = randomAccessFile.getChannel();  
 *// 2. 按顺序分散读取数据到缓冲区* ByteBuffer buff1 = ByteBuffer.*allocate*(500);  
 ByteBuffer buff2 = ByteBuffer.*allocate*(500);  
 ByteBuffer buff3 = ByteBuffer.*allocate*(500);  
 fileChannel.read(**new** ByteBuffer[]{buff1, buff2, buff3});  
 *// 3. 从缓冲区获取数据* buff1.flip();  
 buff2.flip();  
 buff3.flip();  
 System.***out***.println(**new** String(buff1.array(), 0, buff1.limit()));  
 System.***out***.println(**"-----------------------------------------------------------------------------------------------------------"**);  
 System.***out***.println(**new** String(buff2.array(), 0, buff2.limit()));  
 System.***out***.println(**"-----------------------------------------------------------------------------------------------------------"**);  
 System.***out***.println(**new** String(buff3.array(), 0, buff3.limit()));  
 fileChannel.close();  
 *// 4. 聚集写入* FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"D:"**,**"raoxianlong"**, **"333.txt"**), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);  
 outChannel.write(**new** ByteBuffer[]{buff1, buff2, buff3});  
 outChannel.close();  
}

#### 5. 字符集编码

*/\*\*  
 \* 字符编码和解码  
 \*/*@Test  
**public void** charset() **throws** CharacterCodingException {  
 Charset charset = Charset.*forName*(**"GBK"**);  
  
 String str = **"水电费聚隆科技的双方各"**;  
  
 CharsetEncoder encoder = charset.newEncoder();  
  
 CharBuffer charBuffer = CharBuffer.*allocate*(1024);  
 charBuffer.put(str);  
 charBuffer.flip();  
  
 ByteBuffer buffer = encoder.encode(charBuffer);  
 buffer.flip();  
  
 **while** (buffer.remaining() > 0){  
 System.***out***.println(buffer.get());  
 }  
}

## 3. 使用NIO完成网络通信

### 1. 通道(Channel) – 负责连接

**Channel**

SelectableChannel

* + - * SocketChannel
      * ServerSocketChannel
      * DatagramChannel

### 2. 缓冲区-负责数据存储

### 3. 选择器（Selector）

是SelectableChannel的多路由复用器。用于监控SelectableChannel的IO装

### 4. 管道

管道是2个线程之间的单向数据连接。

Pipe有一个source通道和一个sink通道。数据会被写道sink通道,从source通道读取。

### 5. 练习

#### 使用阻塞模式传递文件

1. **package** com.xhh.javaApi.nio;  
     
   **import** org.junit.Test;  
     
   **import** java.io.IOException;  
   **import** java.net.InetSocketAddress;  
   **import** java.nio.ByteBuffer;  
   **import** java.nio.channels.FileChannel;  
   **import** java.nio.channels.ServerSocketChannel;  
   **import** java.nio.channels.SocketChannel;  
   **import** java.nio.file.Paths;  
   **import** java.nio.file.StandardOpenOption;  
     
   */\*\*  
    \*  
    \*/***public class** BlockChannelDemo {  
     
    */\*\*  
    \* 使用阻塞模式传递数据  
    \* 客户端  
    \*/* @Test  
    **public void** client() **throws** IOException {  
    *// 获取通道* SocketChannel socketChannel = SocketChannel.*open*(**new** InetSocketAddress(8089));  
    *// 获取要传递的文件* FileChannel inChannel = FileChannel.*open*(Paths.*get*(**"D:/raoxianlong/b.jpg"**),StandardOpenOption.***WRITE***, StandardOpenOption.***READ***);  
    *// 创建缓冲区* ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
    **while** (inChannel.read(buffer) != -1){  
    buffer.flip();  
    socketChannel.write(buffer);  
    buffer.clear();  
    }  
    socketChannel.shutdownOutput();  
    inChannel.close();  
     
    **int** len;  
    **while** ((len = socketChannel.read(buffer)) != -1){  
    System.***out***.println(**new** String(buffer.array(), 0, len));  
    }  
    socketChannel.close();  
    }  
     
    */\*\*  
    \* 接受数据  
    \* 服务端  
    \*/* @Test  
    **public void** server() **throws** IOException {  
    *// 创建通道* ServerSocketChannel server = ServerSocketChannel.*open*();  
    server.bind(**new** InetSocketAddress(8089));  
    *// 获取连接* SocketChannel channel = server.accept();  
     
    FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"D:/raoxianlong/f2.jpg"**),  
    StandardOpenOption.***CREATE***, StandardOpenOption.***WRITE***, StandardOpenOption.***READ***);  
    ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
     
    **while** (channel.read(buffer) != -1){  
    buffer.flip();  
    outChannel.write(buffer);  
    buffer.clear();  
    }  
    buffer.put(**"数据接受完毕"**.getBytes());  
    buffer.flip();  
    channel.write(buffer);  
    channel.shutdownOutput();  
    }  
     
   }

#### 2. 使用非阻塞模式进行网络通讯

*/\*\*  
 \* 服务端  
 \** ***@param* args** *\*/***public static void** main(String[] args) **throws** IOException {  
 ServerSocketChannel server = ServerSocketChannel.*open*();  
 *// 切换成非阻塞模式* server.configureBlocking(**false**);  
 *// 绑定端口号* server.bind(**new** InetSocketAddress(8089));  
 *// 获取选择器* Selector selector = Selector.*open*();  
 *// 将通道注册到选择器中* server.register(selector, SelectionKey.***OP\_ACCEPT***);  
 *// 轮询获取选择器上准备好的事件* **while** (selector.select() > 0){  
 *// 获取所有准备好的选择键* Iterator<SelectionKey> iter = selector.selectedKeys().iterator();  
  
 **while** (iter.hasNext()){  
 SelectionKey key = iter.next();  
 *// 具体判断是什么事件准备就绪* **if** (key.isAcceptable()){  
 SocketChannel channel = server.accept();  
 channel.configureBlocking(**false**);  
 channel.register(selector, SelectionKey.***OP\_READ***);  
 }**else if** (key.isReadable()){  
 *// 获取当前选择器上准备就绪的通道* SocketChannel read = (SocketChannel)key.channel();;  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 **int** len;  
 **while** ((len = read.read(buffer)) != -1){  
 buffer.flip();  
 String str = **new** String(buffer.array(), 0, len);  
 **if** (StringUtil.*isNotBlank*(str)){  
 System.***out***.println(str);  
 }  
 buffer.clear();  
 }  
 }  
 iter.remove();  
 }  
 }  
}

*/\*\*  
 \* 客户端 (阻塞模式)  
 \*  
 \* 传递一张图片  
 \*/***public static void** main(String[] args) **throws** IOException {  
  
 *// 获取通道* SocketChannel socketChannel = SocketChannel.*open*(**new** InetSocketAddress(8089));  
  
 socketChannel.configureBlocking(**false**);  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 Scanner scanner = **new** Scanner(System.***in***);  
 *// 写入数据* **while** (scanner.hasNext()){  
 String str = **"天子一号 "**+ **new** SimpleDateFormat(**"HH:mm:ss"**).format(**new** Date()) + **"\n"** + scanner.nextLine();  
 buffer.put(str.getBytes());  
 buffer.flip();  
 socketChannel.write(buffer);  
 buffer.clear();  
 }  
 socketChannel.close();  
}

#### 3. 使用UDP进行网络通讯

*/\*\*  
 \* 服务端  
 \** ***@param* args** *\*/***public static void** main(String[] args) **throws** IOException {  
 DatagramChannel dc = DatagramChannel.*open*();  
 dc.bind(**new** InetSocketAddress(8890));  
 dc.configureBlocking(**false**);  
  
 Selector selector = Selector.*open*();  
 dc.register(selector, SelectionKey.***OP\_READ***);  
  
 **while** (selector.select() > 0){  
 Iterator<SelectionKey> it = selector.selectedKeys().iterator();  
  
 **while** (it.hasNext()){  
 SelectionKey sk = it.next();  
  
 **if** (sk.isReadable()){  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 DatagramChannel dc1 = (DatagramChannel) sk.channel();  
 dc1.receive(buffer);  
 buffer.flip();  
 System.***out***.println(**new** String(buffer.array(), 0, buffer.limit()));  
 buffer.clear();  
 }  
 }  
 it.remove();  
 }  
}

*/\*\*  
 \* 客户端  
 \** ***@param* args** *\** ***@throws*** *IOException  
 \*/***public static void** main(String[] args) **throws** IOException {  
 DatagramChannel dc = DatagramChannel.*open*();  
 dc.configureBlocking(**false**);  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
  
 Scanner scanner = **new** Scanner(System.***in***);  
 **while** (scanner.hasNext()){  
 String str = scanner.nextLine();  
 buffer.put((**new** Date().toString() + **"\n"** + str).getBytes());  
 buffer.flip();  
 dc.send(buffer, **new** InetSocketAddress(**"127.0.0.1"**, 8890));  
 buffer.clear();  
 }  
 dc.close();  
}

#### 4. 管道

**public class** PipDemo {  
  
 @Test  
 **public void** pip() **throws** IOException, InterruptedException {  
 Pipe pipe = Pipe.*open*();  
 MyThread t1 = **new** MyThread(pipe, 0);  
 MyThread t2 = **new** MyThread(pipe, 1);  
 t1.run();  
 Thread.*sleep*(2000);  
 t2.run();  
 Thread.*sleep*(2000);  
 pipe.sink().close();  
 pipe.source().close();  
 }  
}  
  
**class** MyThread **extends** Thread{  
  
 **private int flag**;  
 **private** Pipe **pipe**;  
 **public** MyThread(Pipe pipe, **int** flag){  
 **this**.**pipe** = pipe;  
 **this**.**flag** = flag;  
 }  
  
 @Override  
 **public void** run() {  
 **if** (**flag** == 0){  
 **try** {  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 Pipe.SinkChannel sinkChannel = **pipe**.sink();  
 buffer.put(**"饶先龙是傻逼"**.getBytes());  
 buffer.flip();  
 sinkChannel.write(buffer);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }**else** {  
 **try** {  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 Pipe.SourceChannel sourceChannel = **pipe**.source();  
 sourceChannel.read(buffer);  
 buffer.flip();  
 System.***out***.println(**new** String(buffer.array(), 0, buffer.limit()));  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

# 6. Socket网络编程

## 1. 网络编程基本概念

### 1. 报文段

* 报文段是指TCP/IP协议网络传输过程中，起着路由导航作用
* 用以查询各个网络路由网段、IP地址、交换协议等IP数据包
* 报文段充当整个TCP/IP协议数据包的导航路由功能
* 报文在传输过程中会不断封装成组、包、帧来传输
* 封装的信息就是添加一些控制信息组成的首部，即报文头

### 2. 传输协议

* 协议顾名思义，一种规定、约束
* 约定大于配置，在网络传输中依然适用;网络的传输流程是健壮的、稳定的，得益于基础的协议构成
* 简单来说：A->B的传输数据，B能识别，反之B->A的传输数据A也能识别，这就是传输协议。

### 3. Mac地址

* Media Access Control 或者Medium Access Control
* 意为媒体访问控制，或称为物理地址、硬件地址
* 用来定义网络设备的地址
* 形如：44-45-53-54-00-00;与身份证相似

### 4. IP地址

* 互联网协议地址(Internet Protocl Address)，又意为网际协议地址，缩写为IP(IP Address)
* 是分配给网络上使用网际协议的设备的数字标签
* 常见的IP地址分为IPv4和IPv6两大类
* IP地址由32位二进制数组成，常以xxx.xxx.xxx.xxx形式表现，每组xxx代表小于或等于255的10进制数，如：208.80.152.2
* 分为A、B、C、D、E五大类，其中E类属于特殊保留地址
* IPv4总数量：42亿个;最终于2011年2月3日用尽
* 如果主机号是全1,那么这个地址为直接广播地址
* IP地址：255.255.255.255 为受限广播地址
* IPv6 总共有128位长，IPv6地址的表达形式，一般采用32个十六进制数。也可以想象为1632个

### 5. 端口

* 如果把IP地址比作一间房子，端口就是出入这间房子的门或窗户。
* 0到1023号端口以及1024到49151端口都是特殊端口
* 计算机之间依照互联网传输层TCP/IP协议的协议通信，不同的协议都对应不同的端口
* 49152到65535号端口属于”动态端口”范围，没有端口可以被正式注册占用

#### 常见特殊端口

|  |  |  |
| --- | --- | --- |
| 服务名称 | 端口号 | TCP/UDP |
| FTP(文件传输协议) | **20,21** | **TCP** |
| SSH（Shell 协议） | **22** | **TCP** |
| Telnet | **23** | **TCP** |
| SMTP(简单邮件协议) | **25** | **TCP** |
| DNS | **53** | **UDP** |
| TFTP | **69** | **UDP** |
| HTTP | **80** | **TCP** |
| POP3 | **110** | **TCP** |
| IMAP4 | **143** | **TCP** |
| HTTPS | **443** | **TCP** |

### 6. 什么是网络

* **在计算机了领域之中，网络是信息传输、接受、共享的虚拟平台。**
* **通过它把各个点、面、体的信息联系到一起，从而实现资源的共享。**

### C:/Users/002559.XHH/Downloads/未命名文件.jpg7. 网络模型

## 2. 网络编程

### 1. 什么是Socket

* 简单来说是IP地址与端口的结合协议(RFC 793)
* 一种地址与端口的结合描述协议
* TCP/IP协议的相关API的总称;是网络API的集合实现
* 涵盖了：Stream Socket/Datagram Socke

### 2. Socket的组成

* 在网络传输中用于唯一标识两个端口之间的链接
* 短点：（IP + Port）
* 4个要素：客户端地址、客户端端口、服务器地址、服务器端口

### TCP

* TCP是面向连接的通信协议
* 通过三次握手建立连接，通讯完成时就要拆除连接
* 由于TCP是面向连接的所以只能用于端到端的通讯

#### 1. TCP是什么

* 英语:Transmission Control Protocol，缩写TCP
* TCP是传输控制协议;是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793定义

#### 2. TCP的机制

* 三次握手、四次挥手
* 具有校验机制、可靠、数据传输稳定

#### 3. TCP能做什么

* 聊天消息传输、推送
* 单人语音、视频聊天等
* 几乎UDP能做的都能做，但需要考虑复杂性、性能
* 限制：无法进行广播，多播等操作。

#### TCP核心API

**服务器端：**

**ServerSocket(服务器套接字):**

**构造方法：**

* + - * **ServerSocket()：创建未绑定的服务器套接字。**
      * **ServerSocket(int port)：创建绑定到指定端口的服务器套接字。**

**方法：**

* + - * **bind (SocketAddress endpoint)： 将 ServerSocket绑定到特定地址（IP地址和端口号）。**
      * **Socket accept() ：侦听要连接到此套接字并接受它。**
      * **InetAddress getInetAddress()：返回此服务器套接字的本地地址。**
      * **int getLocalPort()：返回此套接字正在侦听的端口号。**

**客户端：**

**Socket(客户端套接字):**

**构造方法：**

* + - * **Socket() ：创建一个未连接的套接字，并使用系统默认类型的SocketImpl。**
      * **Socket(InetAddress address, int port) ：创建流套接字并将其连接到指定IP地址的指定端口号。**
      * **Socket(String host, int port) ：创建流套接字并将其连接到指定主机上的指定端口号。**

**方法：**

* + - * **bind(SocketAddress bindpoint) ：将套接字绑定到本地地址。**
      * **close() ：关闭此套接字。**
      * **connect(SocketAddress endpoint) ：将此套接字连接到服务器。**
      * **connect(SocketAddress endpoint, int timeout) ：将此套接字连接到具有指定超时值的服务器。**
      * **getOutputStream() ：返回此套接字的输出流。**
      * **getInputStream() ：返回此套接字的输入流。**

#### TCP服务端和客户端接连接流程

**服务端：**

第一步：创建ServerSocket套接字

第二步：bind绑定本地端口

第三步： accept客户端套接字 🡨--🡪远程套接字服务器

**客户端：**

第一步：创建Socket套接字

第二步：bind绑定本地端口

第三步： connect远程套接字 🡨--🡪远程套接字服务器

#### TCP连接的可靠性

##### 三次握手（连接）

**第一次： 发送者给接受者发送syn命令 携带一个随机值参数x**

**第二次： 接收者接收到syn命令将参数值+1，发送一个并同时发送syn命令和ack命令**

**携带刚刚 +1的x参数，以及重新生成一个随机的y值给发送方。**

**第三次：发送方收到消息之后将回复一个ack命令给接收方,同时将参数+1。**

**其原理就像两个人电话测试：**

**甲(Sender)：你能听到我说话吗，我是甲。**

**乙(Receiver)：我能听到你说话的甲，你能听到我说话吗。**

**甲(Sender): 我也能听到你说话。**

**这样的话就代表两边都能发送以及接受到消息，相互连接成立。**

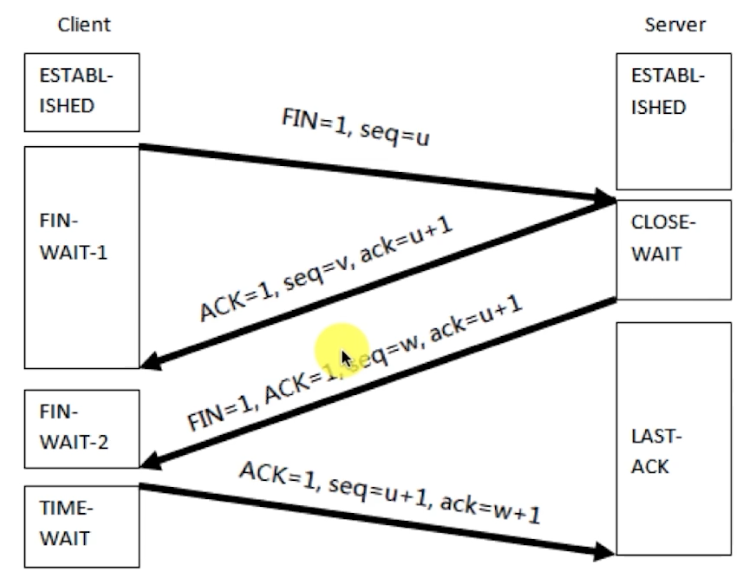
##### 四次挥手（断开连接）

**第一次：client发送一个FIN命令，携带一个seq随机数参数**

**第二次：server端收到FIN命令之后，会断开来自Client端的输入流，并回送ack命令确认可以断开。Client收到ack命令之后会断开输出流不再像Server端发送消息。**

**第三次：当Server端确认数据已经全部发送给Client端之后，会不断向Client发送FIN命令**

**第四次：当Client完全接收数据完成之后，回复一个Ack命令，告诉Server可以断开输出流**



**因为TCP连接是相互的，所以要确保两边的数据全部送达完成之后，才能断开各自的输出流。**

#### TCP传输的可靠性

##### 排序、顺序发送、顺序组装

**当进行一条数据发送的时候，TCP会将数据拆分成不同的片段， 然后将拆分的片段顺序组装之后进行发送，保证数据的有序性。**

**拆分数据的原因：**

**因为有的时候数据比较大，如果是一个整个传输，在传输50%后如果出现错误，那么前面传输的50%都将被丢弃。那如果是拆分成小的数据片段，如果一个小的片段丢失了，也不会影响其他的数据片段。服务端收到数据之后会发送一个确认消息给客户端说收到某个数据片段了，如果client端发送一个数据片段之后，在30秒之后仍然没有收到服务端的确认消息，那么久说明这条数据丢失了。需要客户端重发。**

##### 丢弃、超时

##### 重发机制-定时器

#### 练习

* **package** com.xhh.javaApi.socket;  
    
  **import** com.xhh.javaApi.utils.StringUtil;  
    
  **import** java.io.\*;  
  **import** java.net.InetSocketAddress;  
  **import** java.net.ServerSocket;  
  **import** java.net.Socket;  
  **import** java.util.Date;  
  **import** java.util.Scanner;  
    
  **public class** SocketServer {  
    
   **public static void** main(String[] args) **throws** IOException {  
   ServerSocket server = **new** ServerSocket();  
   server.bind(**new** InetSocketAddress(8890));  
    
   **while** (**true**){  
   Socket socket = server.accept();  
   System.***out***.println(**"客户端："** + socket.getInetAddress()+**":"** + socket.getPort() + **"已连接"**);  
   **new** Thread(**new** ServerThread(socket)).start();  
   }  
   }  
  }  
    
  **class** ServerThread **implements** Runnable{  
    
   **private** Socket **socket**;  
    
   **public** ServerThread(Socket socket){  
   **this**.**socket** = socket;  
   }  
   @Override  
   **public void** run() {  
   BufferedReader reader = **null**;  
   PrintStream print = **null**;  
   **try** {  
   reader = **new** BufferedReader(**new** InputStreamReader(**socket**.getInputStream()));  
   print = **new** PrintStream(**socket**.getOutputStream());  
    
   **while** (**true**){  
   String clientStr = reader.readLine();  
   **if** (**"bye"**.equals(clientStr)){  
   print.print(**"客户小鑫 : 您慢走"**);  
   **break**;  
   }**else**{  
   System.***out***.println(clientStr);  
   clientStr = clientStr.substring(clientStr.indexOf(**":"**) + 1);  
   clientStr = clientStr.replace(**"吗"**, **""**)  
   .replace(**"谁"**, **"客户小鑫"**)  
   .replace(**"你"**, **"我"**)  
   .replace(**"?"**, **"!"**)  
   .replace(**"？"**, **"!"**);  
   System.***out***.println(**"客户小鑫："** + clientStr);  
   print.println(**"客户小鑫："** + clientStr);  
   }  
   }  
   } **catch** (IOException e) {  
   System.***out***.println(**"客户端："** + **socket**.getInetAddress()+**":"** + **socket**.getPort() + **"异常退出"**);  
   e.printStackTrace();  
   }**finally** {  
   System.***out***.println(**"客户端："** + **socket**.getInetAddress()+**":"** + **socket**.getPort() + **"退出连接"**);  
   **try** {  
   **socket**.close();  
   } **catch** (IOException e) {  
   e.printStackTrace();  
   }  
   }  
    
   }  
  }
* **package** com.xhh.javaApi.socket;  
    
  **import** com.xhh.javaApi.utils.StringUtil;  
  **import** org.junit.Test;  
    
  **import** java.io.\*;  
  **import** java.net.InetSocketAddress;  
  **import** java.net.ServerSocket;  
  **import** java.net.Socket;  
  **import** java.util.Date;  
  **import** java.util.Scanner;  
    
  */\*\*  
   \*  
   \*/***public class** SocketClient {  
    
    
   **public static void** main(String[] args) **throws** IOException {  
   *// 创建Socket 连接服务器地址和端口* Socket socket = **new** Socket(**"127.0.0.1"**, 8890);  
    
   System.***out***.println(**"已连接到服务器:"** + socket.getInetAddress() + **":"** + socket.getPort());  
   OutputStream out = socket.getOutputStream();  
   PrintStream print = **new** PrintStream(out);  
    
   InputStream in = socket.getInputStream();  
   BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(in));  
    
   Scanner scanner = **new** Scanner(System.***in***);  
    
   **while** (**true**){  
   String str = scanner.nextLine();  
   System.***out***.println(**"张三:"** + str);  
   print.println(**"张三:"** + str);  
   String inStr = reader.readLine();  
   **if** (**"bey"**.equals(inStr)){  
   System.***out***.println(**"退出连接"**);  
   **break**;  
   }  
   System.***out***.println(inStr);  
   }  
   socket.close();  
   }  
  }

### UDP

* UDP是面向无连接的通信协议
* UDP数据包括目的端口号和源端口号信息
* 由于通信不需要连接，所以可以实现广播发送，并不局限于端到端
* 一种用户**数据报协议**，又称用户**数据报文协议, 非连接协议**
* 是一个简单的面向数据报的传输协议，正式规范为RFC 768

#### 1. UDP是不可靠的

* 它一旦把应用程序发给网路层的数据发送出去,就不保留数据备份。
* UDP在IP数据报的头部仅仅加入了复用和数据校验
* 发送端生产数据，接收端从网络抓取数据

#### 2. UDP可以做什么

* DNS、TFTP、SNMP
* 视频、音频、普通数据（无关紧要的数据）

#### 3. UDP最大长度

* 16位->2字节存储长度信息
* 65507 byte

#### 4. UDP核心API

##### 1. DatagramSocket(接受和发送)

###### 构造方法

* **DatagramSocket(int port)** 构造数据报套接字并将其绑定到本地主机上的指定端口。
* **DatagramSocket(int port, InetAddress laddr)**  创建一个数据报套接字，绑定到指定的本地地址。
* **DatagramSocket(SocketAddress bindaddr)** 创建一个数据报套接字，绑定到指定的本地套接字地址。
* **DatagramSocket()** 构造数据报套接字并将其绑定到本地主机上的任何可用端口。

###### 方法

* **receive(DatagramPacket p)**  从此套接字接收数据报包。
* **send(DatagramPacket p)**  从此套接字发送数据报包。
* **setSoTimeout(int timeout)**  以指定的超时（以毫秒为单位）\
* **close()** 关闭此数据报套接字。
* **connect(InetAddress address, int port)** 将套接字连接到此套接字的远程地址。
* **connect(SocketAddress addr)** 将此套接字连接到远程套接字地址（IP地址+端口号）。

##### 2. DatagramPacket(数据包)

###### 构造方法

* **DatagramPacket(byte[] buf, int length)** 构造一个 DatagramPacket用于接收长度的数据包 length 。
* **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** 构造用于发送长度的分组的数据报包 length指定主机上到指定的端口号。
* **DatagramPacket(byte[] buf, int offset, int length)** 构造一个 DatagramPacket用于接收长度的分组 length ，指定偏移到缓冲器中。
* **DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)** 构造用于发送长度的分组数据报包 length具有偏移 ioffset指定主机上到指定的端口号。
* **DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)** 构造用于发送长度的分组数据报包 length具有偏移 ioffset指定主机上到指定的端口号。
* **DatagramPacket(byte[] buf, int length, SocketAddress address)** 构造用于发送长度的分组的数据报包 length指定主机上到指定的端口号。

###### 方法

* **netAddress getAddress()** 返回该数据报发送或接收数据报的计算机的IP地址。
* **byte[] getData()** 返回数据缓冲区。
* **int getLength()** 返回要发送的数据的长度或接收到的数据的长度。
* **int getOffset()** 返回要发送的数据的偏移量或接收到的数据的偏移量。
* **int getPort()** 返回发送数据报的远程主机上的端口号，或从中接收数据报的端口号。
* **SocketAddress getSocketAddress()** 获取该数据包发送到或正在从其发送的远程主机的SocketAddress（通常为IP地址+端口号）。
* **void setAddress(InetAddress iaddr)** 设置该数据报发送到的机器的IP地址。
* **void setData(byte[] buf)** 设置此数据包的数据缓冲区。
* **void setData(byte[] buf, int offset, int length)** 设置此数据包的数据缓冲区。
* **void setLength(int length)** 设置此数据包的长度。
* **void setPort(int iport)** 设置发送此数据报的远程主机上的端口号。
* **void setSocketAddress(SocketAddress address)** 设置该数据报发送到的远程主机的SocketAddress（通常是IP地址+端口号）。

##### 3. 练习

**package** com.xhh.javaApi.socket.udp;  
  
**import** java.io.IOException;  
**import** java.net.\*;  
**import** java.util.HashSet;  
**import** java.util.Scanner;  
**import** java.util.Set;  
  
**public class** UDPSearcher {  
  
 **public static void** main(String[] args) **throws** IOException {  
 DatagramSocket ds = **new** DatagramSocket();  
 Scanner scanner = **new** Scanner(System.***in***);  
  
 **new** ReceiveThread(ds).start();  
  
 **while** (scanner.hasNext()){  
 **byte**[] data = scanner.nextLine().getBytes();  
 DatagramPacket datagramPacket = **new** DatagramPacket(data, data.**length**, InetAddress.*getLocalHost*(), 2000);  
 ds.send(datagramPacket);  
 }  
 }  
  
 **public static class** ReceiveThread **extends** Thread{  
  
 DatagramSocket **ds**;  
  
 **public** ReceiveThread(DatagramSocket ds){  
 **this**.**ds** =ds;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
  
 **byte**[] data = **new byte**[1024];  
 DatagramPacket packet = **new** DatagramPacket(data, data.**length**);  
 **while** (**true**){  
 **try** {  
 **ds**.receive(packet);  
 String ip = packet.getAddress().getHostAddress();  
 **int** port = packet.getPort();  
 String reData = **new** String(packet.getData(), 0, packet.getLength());  
 System.***out***.println(ip + **":"** + port + **"\n"**+ reData);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
}

**package** com.xhh.javaApi.socket.udp;  
  
**import** java.io.IOException;  
**import** java.net.\*;  
**import** java.util.HashSet;  
**import** java.util.Scanner;  
**import** java.util.Set;  
  
**public class** UDPSearcher {  
  
 **public static void** main(String[] args) **throws** IOException {  
 DatagramSocket ds = **new** DatagramSocket();  
 Scanner scanner = **new** Scanner(System.***in***);  
  
 **new** ReceiveThread(ds).start();  
  
 **while** (scanner.hasNext()){  
 **byte**[] data = scanner.nextLine().getBytes();  
 DatagramPacket datagramPacket = **new** DatagramPacket(data, data.**length**, InetAddress.*getLocalHost*(), 2000);  
 ds.send(datagramPacket);  
 }  
 }  
  
 **public static class** ReceiveThread **extends** Thread{  
  
 DatagramSocket **ds**;  
  
 **public** ReceiveThread(DatagramSocket ds){  
 **this**.**ds** =ds;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
  
 **byte**[] data = **new byte**[1024];  
 DatagramPacket packet = **new** DatagramPacket(data, data.**length**);  
 **while** (**true**){  
 **try** {  
 **ds**.receive(packet);  
 String ip = packet.getAddress().getHostAddress();  
 **int** port = packet.getPort();  
 String reData = **new** String(packet.getData(), 0, packet.getLength());  
 System.***out***.println(ip + **":"** + port + **"\n"**+ reData);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
}

#### 5. UDP发送和接收数据过程

**发送：**

第一步：创建DatagramSocket套接字->监听端口

第二步：创建DatagramPacket数据发送包->指定发送的数据、地址和端口

第三步：通过DatagramSocket->send(packet)发送数据包

**接收（监听）：**

第一步：创建DatagramSocket套接字->监听端口

第二步：创建DatagramPacket数据接收包->指定数据缓存，

第三步：通过DatagramSocket-> receive(packet)接收数据包

#### 6. UDP单播、广播、多播

## 3. 服务器性能数据分析

**CPU：取决于数据的频繁性、数据转发的复杂性**

**内存：取决于客户端的数量、客户端发送数据的大小**

**线程：取决于客户端连接的数量**

### 1. 服务器优化方案分析

**减少线程数量**

**增加线程执行繁忙状态**

**客户端Buffer复用机制**

# 7. Java发送邮件

## 邮件的基本概念

我们想要发送或者接受邮件，必须要有专门的邮件服务器提供这个服务。就像邮局一样，那么在网络上也有专门的邮件服务提供商：sina、sohu、163等等他们都有自己的邮件服务器。

### 邮件发送协议

我们使用SMTP协议来发送电子邮件

### 邮件接受协议

使用pop3协议来接受电子邮件

### 电子邮件发送和接受过程

#### ****以163邮箱为例→telnet发邮件****

通过CMD 手动发送邮件 使用 tnlnet smtp.163.com 25 连接到163邮箱的SMTP服务器

1.连接→telnet smtp.163.com 25

回显”220″

2.报名字(随便写)→helo vkill

回显”250 ok”

3.登陆模式(这时选一般登陆模式)→AUTH LOGIN

回显”334 dXNlcm5hbWU6″

(注:这时的用户和密码都为Base64编码后的字符,手头没有Base64编码加密器的话,偶提供个在线的:http://www.yx127.com/base64.html)、

4.用户名→\*\*\*\*

回显”334 UGFzc3dvcmQ6″

5.密码→\*\*\*\*

回显”235 Authentication successful” 表示验证通过

6.发件人→mail from: <XXXXX@163.com >

回显”250 Mail OK”

7.收件人→rcpt to: <XXXXX@56.com >

回显”250 Mail OK”

8.开始写正文→data

输入正文内容就ok,正文要符合MIME规范

例子:

**From: he200377@163.com ←电子邮件的发信人**

**To: he200377@56.com ←电子邮件的收信人**

**Cc: he200377@56.com ←接收通过抄送方式发送的邮件的收件人**

**Subject: test ←电子邮件的主题**

**MIME-Version: 1.0 ←MIME 协议的版本**

**Content-Type: text/plain ←邮件格式text**

**hello vkill ←内容**

**当行输入. 回车,回显”250 OK”发送成功**

***9.断开连接→quit***

#### ****以163邮箱为例→telnet收邮件****

**1.连接→telnet pop.163.com 110**

**2.用户名→user \*\*\*\***

**3.密码→pass \*\*\*\***

**4.验证ok后,终端会显示如下信息:**

+OK 60 message(s) [72208371 byte(s)]

数字60表示有60封邮件，72208371表示这60封邮件总的字节数(即72208371B)

这时可使用的命令如下:

**1.stat命令 格式：stat 无需参数**

stat命令,查看统计,执行后,POP3服务器会响应一个正确应答,它以“+OK”开头,接着是两个数字,第一个是邮件数目,第二个是邮件的大小

**2.list命令 格式：list [n] 参数n可选，n为邮件编号**

list命令,查看邮件列表,可以利用不带参数的list命令,获得各邮件的编号,并且每一封邮件均占用一行显示,前面的数为邮件的编号,后面的数为邮件的大小

**3.uidl命令 格式：uidl [n] 参数n可选，n为邮件编号**

uidl命令,与list同,不过显示邮件的信息比list更详细,更具体

**4.retr命令 格式：retr n 参数n不可省，n为邮件编号**

retr命令,查看邮件的内容

**5.dele命令 格式：dele n 参数n不可省，n为邮件编号**

dele命令,删除指定的邮件(注意:dele n命令只是给邮件做上删除标记,只有在执行quit命令之后,邮件才会真正删除)

**6.top 命令 格式：top n m 参数n,m不可省,n为邮件编号,m为行数**

top命令,读取指定邮件正文的行数,如果m=0,则只读出邮件的邮件头部分

**7.noop命令 格式：noop 无需参数**

noop命令,该命令发出后,POP3服务器不作任何事,仅返回一个正确响应”+OK”

8.quit命令 格式：quit 无需参数

#### C:\Users\Administrator\Downloads\未命名文件.jpg**流程图**

## 使用JAVA创建邮、发送邮件以及接受邮件

### RFC822文档

1. RFC822文档规定了如何编写一封简单的邮件(纯文本邮件)。

2.一封简单的邮件包含邮件头和邮件体两部分，邮件头和邮件体使用空行分隔。

**邮件头包含:**

1. from --用于指明发件人

2. to --用于指明收件人

3. subject --用于说明邮件主题

4. cc --抄送，收件人可以看到抄送人员

5. bcc --密送，收件不能看到抄送人员.

### MIME协议

1. 实际开发过程，我们在邮件可能不只是纯文本邮件，需要带图片，附件等，这样情况下RFC822文档规定就无法满足了。
2. MIME协议是对RFC822文档的补充升级，MIME协议描述了如何产生一封复杂的邮件。
3. MIME协议规定了要使用分割线对多段数据进行分隔,并使用Content-Type头字段对数据类型、以及多个数据之间的关系进行描述。

### 使用JavaMail创建和发送邮件

**Maven依赖：**

<**dependency**>  
 <**groupId**>javax.mail</**groupId**>  
 <**artifactId**>mail</**artifactId**>  
 <**version**>1.4.6</**version**>  
</**dependency**>

#### ****API设计****

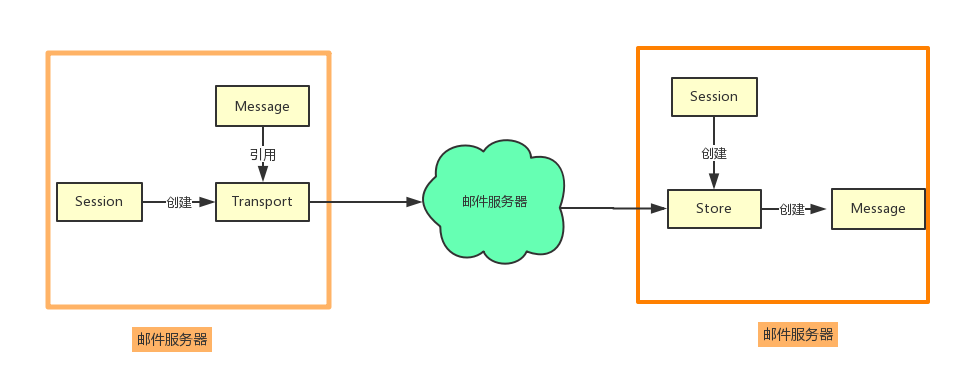
JavaMail API 按其功能划分通常可分为三大类：

**1、创建和解析邮件内容的API**：**Message**类是创建和解析邮件的核心API，它的实例对象代表一封电子邮件

**2、发送邮件的API ： Transport** 类是发送邮件的核心API类，它的实例对象代表实现了某个邮件发送协议的邮件发送对象，例如SMTP协议。

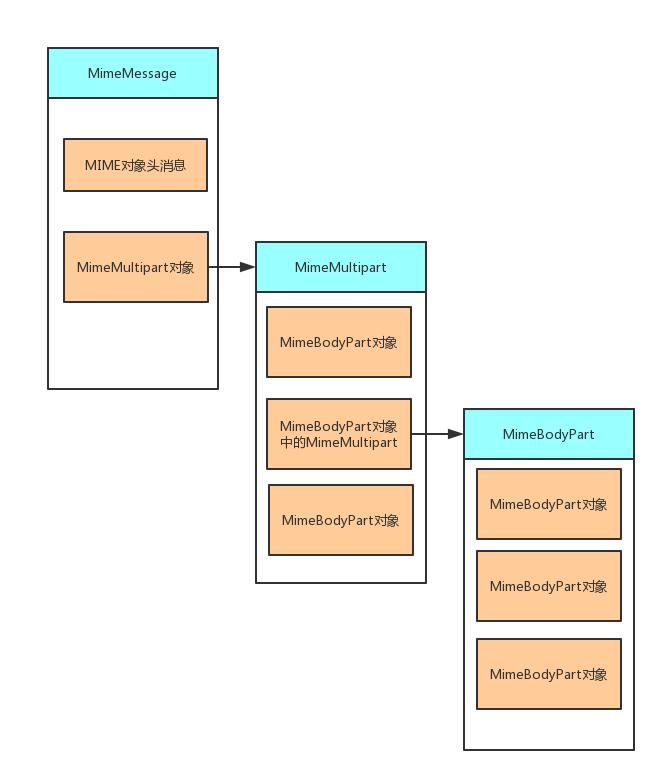
**3、接受邮件的API ：** **Store**类是接受邮件的核心API类，它的实例对象代表实现实现了某个邮件接受协议的邮件接受对象，例如POP3协议。

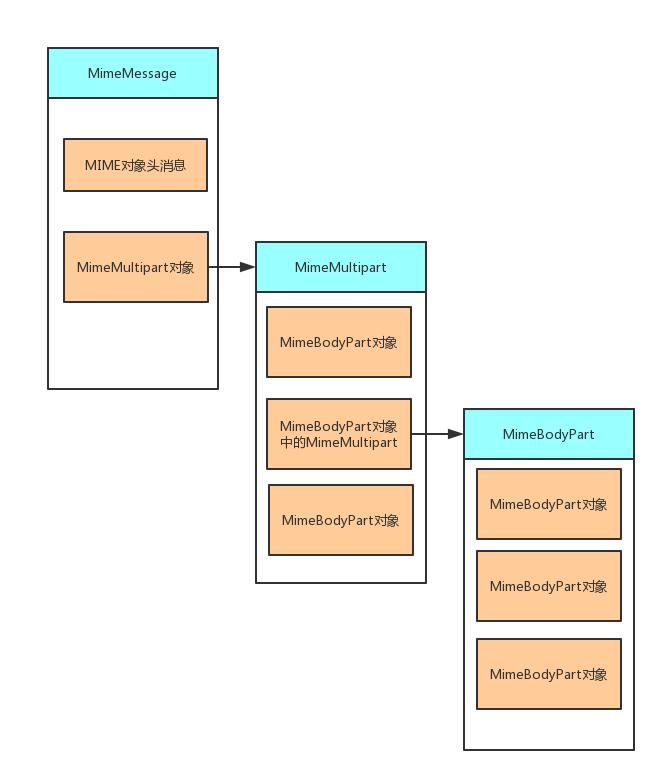
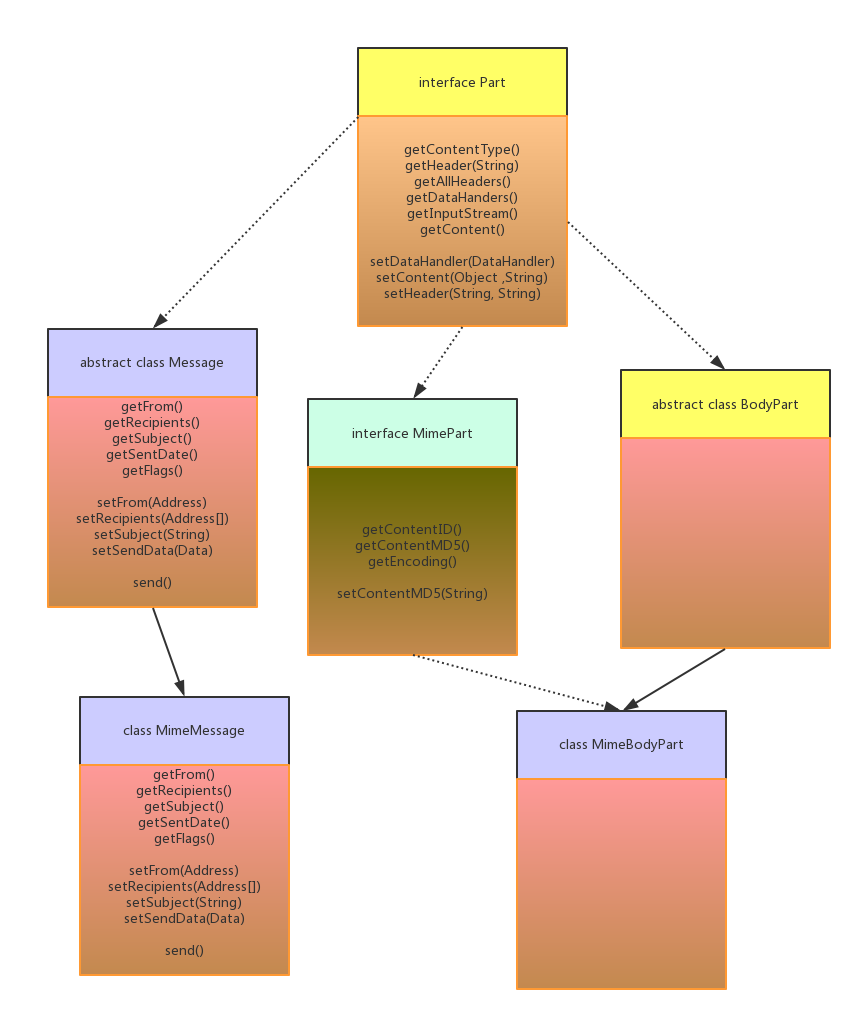
**Session类**：

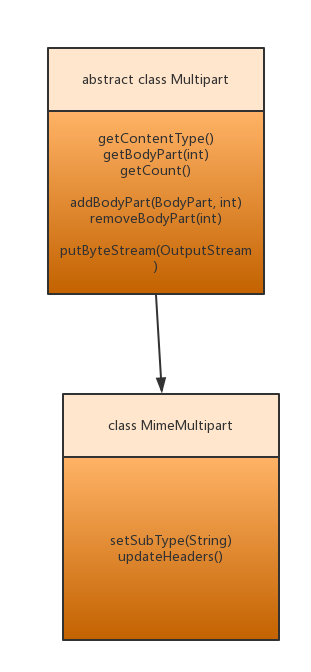
 Session类用于定义整个应用程序所需的环境信息，以及收集客户端与邮件服务器建立网络连接的会话信息，如邮件的服务器的主机名、端口号、采用的邮件发送和接受协议等。Session对象根据这些信息构建用于收发的Transport和Store对象，以及为客户端创建Message对象时提供信息支持。

#### ****使用JavaMail API创建和发送邮件****

JavaMail API使用Javax.mail.Message类表示一封邮件,我们使用它的子类javax.mail.internet.MimeMessage类来创建Message的实例对象。如果要创建负责的邮件，还需要配合使用java.mail.internet.MimeBodyPart和javax.mail.internet.MimeMultipart等类。

1. MimeMessage类表示整封邮件
2. MimeBodyPart类表示邮件的一个MIME消息
3. MimeMultPart类表示一个由多个MIME消息组合成的MIME

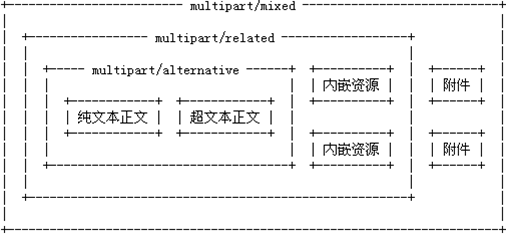
**继承关系**

****

**组合关系：**

Multipart对象创建，其默认的实例对象的MIME类型为mixed，第二种制定一个类型来创建MimeMultipart类的实例对象，其有三种常用的类型：

**mixed，****related，alternative**



#### ****代码实现****

##### 发送一封简单的邮件

**try** {  
 Properties properties = **new** Properties();  
 *//使用的协议(JavaMail规范要求)* properties.setProperty(**"smtp.transport.protocol"**, **"smtp"**);  
 *//发件人邮箱的SMTP服务器地址* properties.setProperty(**"mail.smtp.host"**, **"smtp.163.com"**);  
 *//请求认证，参数名称与具体事项有关* properties.setProperty(**"mail.smtp.auth"**, **"true"**);  
  
 *//创建Session对象* Session session = Session.*getDefaultInstance*(properties);  
 session.setDebug(**true**);  
 *//创建MimeMessages实例对象* MimeMessage msg = **new** MimeMessage(session);  
 *//设置发件人* msg.setFrom(**new** InternetAddress(**"m17601377795@163.com"**));  
 *//设置收件人* msg.setRecipient(Message.RecipientType.***TO***, **new** InternetAddress(**"1203502972@qq.com"**));  
 *//设置抄送* msg.setRecipient(Message.RecipientType.***CC***, **new** InternetAddress(**"raoxianlong@xinhehui.com"**));  
 *//设置发送日期* msg.setSentDate(**new** Date());  
 *//设置邮件主题* msg.setSubject(**"饶先龙，Hello!"**);  
 *//如果需要发送HTML内容* msg.setContent(**"<html><font color='red'>饶先龙</font></html>"**, **"text/html;charset=gb2312"**);  
 *//保存最终邮件内容* msg.saveChanges();  
 Transport ts = session.getTransport(**"smtp"**);  
 ts.connect(**"smtp.163.com"**, **"m17601377795@163.com"**, **"long520"**);  
 ts.sendMessage(msg, msg.getAllRecipients());  
 ts.close();  
 System.***out***.println(**"邮件发送成功！"**);  
} **catch** (MessagingException e) {  
 e.printStackTrace();  
}

##### 发送一封带图片的邮件

因为163邮箱发送图片一直不成功，所以改成QQ邮箱了，QQ登录需要用到授权码。

**try** {  
 Properties properties = **new** Properties();  
 *//使用的协议(JavaMail规范要求)* properties.setProperty(**"smtp.transport.protocol"**, **"smtp"**);  
 *//发件人邮箱的SMTP服务器地址* properties.setProperty(**"mail.smtp.host"**, **"smtp.qq.com"**);  
 *//请求认证，参数名称与具体事项有关* properties.setProperty(**"mail.smtp.auth"**, **"true"**);  
 *//创建Session对象* Session session = Session.*getDefaultInstance*(properties);  
 session.setDebug(**true**);  
  
 Transport ts = session.getTransport(**"smtp"**);  
 ts.connect(**"smtp.qq.com"**, **"1203502972@qq.com"**, **"zefiprafzxzngfaf"**);  
  
 *//发送简单邮件  
 //Message msg = createTextMail(session);  
  
 //发送带图片的邮件* Message msg = *createImageMail*(session);  
  
 ts.sendMessage(msg, msg.getAllRecipients());  
 ts.close();  
 System.***out***.println(**"邮件发送成功！"**);  
} **catch** (MessagingException e) {  
 e.printStackTrace();  
}

**创建图片Message方法:**

*/\*\*  
 \* 创建一封带图片的邮件  
 \** ***@param* session** *\** ***@return*** *\*/***private static** Message createImageMail(Session session){  
 Message msg = **new** MimeMessage(session);  
 **try** {  
 msg.setFrom(**new** InternetAddress(**"1203502972@qq.com"**));  
 *//设置收件人* msg.setRecipient(Message.RecipientType.***TO***, **new** InternetAddress(**"m17601377795@163.com"**));  
 *//设置抄送* msg.setRecipient(Message.RecipientType.***CC***, **new** InternetAddress(**"raoxianlong@xinhehui.com"**));  
 msg.setSentDate(**new** Date());  
 msg.setSubject(**"JavaMail测试发送图片"**);  
  
 MimeBodyPart text = **new** MimeBodyPart();  
 text.setContent(**"看美女<br/><img src='cid:aaa.jpg'>"**,**"text/html;charset=UTF-8"**);  
 *//图片* MimeBodyPart image = **new** MimeBodyPart();  
 image.setDataHandler(**new** DataHandler(**new** FileDataSource(**"src\\b.jpg"**)));  
 image.setContentID(**"aaa.jpg"**);  
 *//描述数据关系* MimeMultipart mp = **new** MimeMultipart();  
 mp.addBodyPart(text);  
 mp.addBodyPart(image);  
 mp.setSubType(**"related"**);  
  
 msg.setContent(mp);  
 msg.saveChanges();  
 } **catch** (MessagingException e) {  
 e.printStackTrace();  
 }  
 **return** msg;  
}



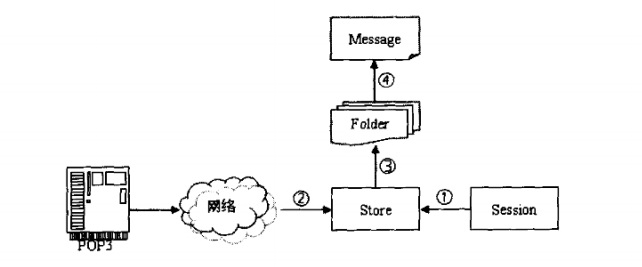
##### 发送一封带附件的邮件

### 使用JavaMail接受邮件

#### ****API设计****

JavaMail使用java.mail.Store类用于执行邮件的执行接受任务,我们可以在程序中调用这个类中的方法可以获取邮箱中的各个**邮件夹**信息。JavaMail使用**Folder**对象表示邮件夹,通过Folder对象的方法可以获取邮件夹中的所有邮件信息，而邮件则用Message对象表示，可以获取主题，文件正文，发送时间等。

#### ****接受流程****



1. 从Session对象中获得实现了某种邮件发送协议的Store对象
2. 登录邮箱，连接POP3或者IMAP4服务器
3. 调用Store的getFolder方法，获取邮箱中的某个邮件夹的Folder对象’
4. 调用额Folder对象中的getMessage或getMessage方法，获取邮件夹中的某一封邮件后者所有邮件，每一封的邮件以一个Message对象返回。

#### ****代码实现****

这里实现使用的是QQ邮箱，需要SSL加密进行登录



## 3. 测试账号

测试邮箱（163邮箱）：

账号：m17601377795@163.com

Base64： bTE3NjAxMzc3Nzk1QDE2My5jb20=

密码：long520

Base64：bG9uZzUyMA==