**3、具体内容**

在java之中类库提供的是非常的多的，今天所讲解的类库只属于Java SE的范畴，日后还会学习更多的类库，包括许多的第三方类库，个人建议的学习方法：清楚每一个类的作用，而后其对应的方法操作通过文档查询，但是对于一些常用代码必须要求可以灵活编写。

**3.1、StringBuffer类（重点）**

在讲解StringBuffer类之前首先来简单回顾一下String类的特点：

· String类的对象有两种实例化方式，一种是直接赋值，只会开辟一块堆内存空间，而且对象可以自动入池，另外一种方式使用构造方法完成，会开辟两块空间，有一块空间将成为垃圾，并且不会自动入池，但是可以通过intern()方法手工入池；

· 字符串常量一旦声明则不可改变，而字符串对象可以改变，但是改变的是其内存地址的指向；

通过以上的几个特点就可以清楚的发现，String类是表示字符串使用最多的类，但是其不适合于被频繁修改的字符串操作上，所以在这种情况下，往往可以使用StringBuffer类，即：StringBuffer类方便用户进行内容的修改。在String类之中使用“+”作为数据库的连接操作，而在StringBuffer类之中使用append()方法进行数据的连接。

**范例：**使用StringBuffer操作，StringBuffer的内容可以改变

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  StringBuffer buf = **new** StringBuffer();  buf.append("Hello ").append("World ."); // 连接内容  *fun*(buf);  System.*out*.println(buf);  }  **public** **static** **void** fun(StringBuffer temp) {  temp.append("\n").append("Hello MLDN");  }  } |

**StringBuffer类在日后主要用于频繁修改字符串的操作上，但是在任何的开发之中，面对字符串的操作，98%都先考虑String，只有那2%会考虑StringBuffer。**

现在表示字符串的操作类就有了两个：String、StringBuffer，那么下面通过这两个类的定义来研究一下关系：

|  |  |
| --- | --- |
| **String类：** | **StringBuffer类：** |
| public final class String  extends Object  implements Serializable, Comparable<String>, **CharSequence** | public final class StringBuffer  extends Object  implements Serializable, **CharSequence** |

现在发现String和StringBuffer类都实现了一个CharSequence接口，日后一定要记住，如果看见了CharSequence最简单的理解做法就是传字符串，但是虽然这两个类是同一个接口的子类，不过这两个类对象之间却不能互相直接转型。

**操作一：**将String变为StringBuffer

· **方法一：**直接利用StringBuffer类的构造方法，public StringBuffer(String str)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "Hello World ." ;  StringBuffer buf = **new** StringBuffer(str);  System.*out*.println(buf);  }  } |

· **方法二：**利用StringBuffer类的append()方法

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "Hello World ." ;  StringBuffer buf = **new** StringBuffer();  buf.append(str) ;  System.*out*.println(buf);  }  } |

**操作二：**将StringBuffer变为String，利用StringBuffer类的toString()方法完成

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  StringBuffer buf = **new** StringBuffer();  buf.append("Hello World .") ;  String str = buf.toString() ;  System.*out*.println(str);  }  } |

在String类之中定义了许多的操作方法，同样，在StringBuffer类之中也定义了许多的操作方法，而且有些方法还是String类所有没有的支持。

**范例：**字符串反转操作，public **StringBuffer** reverse()

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  StringBuffer buf = **new** StringBuffer();  buf.append("Hello World .") ;  System.*out*.println(buf.reverse());  }  } |

**范例：**替换指定范围内的数据，public **StringBuffer** replace(int start, int end, String str)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  StringBuffer buf = **new** StringBuffer();  buf.append("Hello World .") ;  System.*out*.println(buf.replace(6, 12, "MLDN"));  }  } |

**范例：**在指定位置上插入数据，public StringBuffer insert(int offset, 数据类型 变量)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  StringBuffer buf = **new** StringBuffer();  buf.append("World .").insert(0, "Hello ") ;  System.*out*.println(buf);  }  } |

**面试题：**请解释String和StringBuffer的区别？

String的内容不可改变，而StringBuffer的内容可以改变。

**3.2、Runtime类（了解）**

在每一个JVM进程之中，都会**存在一个运行时的操作类的对象**，而这对象所属的类型就是Runtime类。打开这个类的文档可以发现，在这个类之中并没有构造方法定义，可是按照之前所学，每个类至少有一个构造方法，而这个类的构造方法实际上存在只是不被外部看见而已，因为构造方法被私有化了，这是一个标准的单例设计模式，既然是单例设计模式则在这个类就一定会存在一个static型的方法，可以取得本类的实例化对象：public static Runtime getRuntime()。

而当取得了这个类的实例化对象之后，可以利用这个类取得一些JVM的信息，例如：

· **取得最大可用内存：**public **long** maxMemory()；

· **总共可以使用的内存：**public **long** totalMemory()；

· **空闲的内存：**public **long** freeMemory()。

发现取得内存信息的时候所有的数据返回的类型是long，在之前讲解基本数据类型的时候强调long型数据的使用就在两种情况：表示文件大小、表示日期时间。

在Runtime类有一个非常重要的方法：public void gc()，运行垃圾收集器，释放垃圾空间。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Runtime run = Runtime.*getRuntime*() ; // 取得对象  System.*out*.println("1.MAX\_MEMORY：" + run.maxMemory());  System.*out*.println("1.TOTAL\_MEMORY：" + run.totalMemory());  System.*out*.println("1.FREE\_MEMORY：" + run.freeMemory());  String str = "" ;  **for** (**int** x = 0; x < 30000; x++) {  str += x ; // 产生垃圾  }  System.*out*.println("2.MAX\_MEMORY：" + run.maxMemory());  System.*out*.println("2.TOTAL\_MEMORY：" + run.totalMemory());  System.*out*.println("2.FREE\_MEMORY：" + run.freeMemory());  run.gc() ;  System.*out*.println("3.MAX\_MEMORY：" + run.maxMemory());  System.*out*.println("3.TOTAL\_MEMORY：" + run.totalMemory());  System.*out*.println("3.FREE\_MEMORY：" + run.freeMemory());  }  } |

**面试题：**请解释一下什么叫gc()？Java是如何处理的？

GC（Garbage Collector）：垃圾收集器，可以释放掉垃圾空间所占用的内存。在java之中GC有两种方式处理：一种是由JVM不定期的执行GC操作，另外一种是由用户自己手工调用Runtime类的gc()方法进行垃圾空间的释放。

**3.3、System类（了解）**

System类一直很熟悉，从未被了解，从最早的系统输出，后来的数组拷贝都属于System类的操作，下面来看一下数组排序的方法定义：

· **之前的格式：**System.arraycopy(源数组名称，源数组开始点，目标数组名称，目标数组开始点，长度) ;

· **System类定义：**public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)。

在System类之中有一个取得当前日期时间的方法：public static **long** currentTimeMillis()。

**范例：**使用System类的currentTimeMillis()方法来统计一个操作所花费的时间

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **long** start = System.*currentTimeMillis*();  String str = "";  **for** (**int** x = 0; x < 30000; x++) {  str += x; // 产生垃圾  }  **long** end = System.*currentTimeMillis*();  System.*out*.println("花费的时间：" + (end - start));  }  } |

在日后的许多开发之中，都会出现自动统计操作时间的功能，出现之后应该可以立刻清楚是通过System.currentTimeMilllis()方法完成的。

可是在System类之中还存在了一个很有意思的方法：public static void gc()，但是这个gc()方法并不是一个新的操作方法，而是间接调用了Runtime类之中的gc()方法，不表示一个重写的方法。

**问题：**现在如果说一个对象产生的话，可以通过构造方法处理一些对象产生时的操作，但是，当一个对象被回收了呢？发现没有像C++那样的析构函数（对象回收前的收尾），如果现在希望在一个对象收尾的时候执行一些收尾工作，则对象所在的类可以实现一个finalize()方法，此方法由Object类定义：protected void finalize() throws **Throwable**。在对象回收之前有可能出现异常或者是错误，但是即使出现了，这些错误或者是异常都不会影响程序的执行，即：不会因为异常而导致程序的中断执行。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {  **public** Person() {  System.*out*.println("狂风乱起，地动山摇，海啸，告诉大家：妖孽出生了 —— 玉史");  }  @Override  **protected** **void** finalize() **throws** Throwable {  System.*out*.println("妖孽被除了。。。玉史：老子18年后又是一条好虫。");  **throw** **new** Exception("祸害人间。"); // 抛异常了  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person();  per = **null**; // 消失了  System.*gc*() ;  }  } |

**面试题：**请解释final、finally、finalize的区别？

· final表示终结器，用于定义不能被继承的父类，不能被覆写的方法，常量；

· finally是异常处理的出口；

· finalize()是Object类定义的一个方法，用于执行对象回收前的收尾操作。

## 3.4、日期操作类（核心）

### 3.4.1、java.util.Date

学习到了今天，对于之前的简单java类，是可以和数据表进行完整映射的，但是对于数据表的日期型字段却一直没有映射，而在Java之中，如果要想表示出日期型，则使用java.util.Date类完成。

如果要想通过这个类取得当前的日期时间，那么只需要直接实例化Date类对象即可：public Date()。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Date date = **new** Date();  System.*out*.println(date); // Tue Dec 18 10:02:28 CST 2012  }  } |

但是发现，这个时候返回的日期时间实在是不能够被一些人所看懂，所以在之后需要对其进行转换。

**提示：**关于long和日期时间的转换

在Date对象之中本身是包含了日期时间数据，但是如果要想让其和long数据之间互相转换，则需要以下方式：

**方式一：**将Date型数据变为long型数据，public long getTime()

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Date date = **new** Date();  **long** num = date.getTime() ;  System.*out*.println(num);  }  } |

**方式二：**将long型变为日期型数据，public Date(long date)

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Date date = **new** Date(System.*currentTimeMillis*()); // long --> Date  System.*out*.println(date);  }  } |

以上的两种转换操作，在日后的开发之中一定会出现，必须使用熟练。

### 3.4.2、日期格式化操作类：SimpleDateFormat（死了都要会）

在Oracle之中存在了TO\_CHAR()函数（最终的数据是字符串），可以将一个日期型的数据进行格式化的操作显示，而在Java之中也可以通过java.text.**SimpleDateFormat**类完成TO\_CHAR()函数的功能和TO\_DATE()函数的功能。

但是考虑到实际问题，对于SimpleDateFormat类并不太需要关心它的继承结构了，只关心三个方法：

· **构造方法：**public SimpleDateFormat(String pattern)；

· **将日期格式化为字符串：**public final String format(Date date)；

· **将字符串格式化为日期：**public Date parse(String source) throws ParseException；

但是除了以上的三个核心操作方法之外，如果要想正常的完成格式化的操作，还需要准备出一些标记：**年（yyyy）、月（MM）、日（dd）、时（HH）、分（mm）、秒（ss）、毫秒（SSS）**。

**范例：**将日期变为字符串，格式化显示

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Date date = **new** Date(); // 当前日期  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");  String str = sdf.format(date); // 将日期格式化为字符串  System.*out*.println(str);  }  } |

**范例：**将字符串格式化为日期

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "1980-09-19 12:12:12.122";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");  Date date = sdf.parse(str); // 将字符串变为日期  System.*out*.println(date);  }  } |

**提示：**完成了以上的代码之后，实际上对于数据表和简单Java类的映射就应该彻底完成了，数据库之中的日期时间可以通过java.util.Date进行表示。

而且此时发现字符串可以向基本数据类型（包装类）、日期类型（SimpleDateFormat）进行转换，反过来，基本数据类型（String.valueOf()）和日期类型（SimpleDateFormat）也可以向字符串进行转换。

**3.5、随机数类：Random（了解）**

java.util.Random的主要目的是产生随机数，下面直接通过一个代码来观察就行。

**范例：**产生10个0~100之间的正整数

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Random;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Random rand = **new** Random();  **for** (**int** x = 0; x < 10; x++) {  System.*out*.println(rand.nextInt(101));  }  }  } |

**3.6、数学公式类：Math（了解）**

在java.lang.Math类之中定义了所有的与数学有关的基本公式，在这个类之中所有的方法都是static型的方法，而在这个类只强调一个方法：round()，public static long round(double a)，表示四舍五入。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(Math.*round*(15.5)); // 16  System.*out*.println(Math.*round*(15.51)); // 16  System.*out*.println(Math.*round*(15.6)); // 16  System.*out*.println(Math.*round*(15.2356)); // 15  System.*out*.println(Math.*round*(-15.5)); // -15  System.*out*.println(Math.*round*(-15.51)); // -16  System.*out*.println(Math.*round*(-15.6)); // -16  }  } |

如果现在使用的是Oracle的round()函数，那么对于-15.5的结果一定是-16，但是Java的Math.round()方法不太一样：如果是负数，其小数位的数值小于5的话，那么不会进位，如果大于了5，才会进位。

**3.7、大数字操作类（次重点）**

如果说现在有两个非常大的数字要进行数学操作，你们认为该怎么做？这个时候数字已经超过了double的范围，那么只能利用字符串来表示，取出每一位字符变为数字后进行数学计算，但是这种方式的难度较高，为了解决这样的问题，在Java里面提供了两个大数字操作类：java.math.BigInteger、java.math.BigDecimal，而这两个类是属于Number的子类。

**1、 大整数操作类：BigInteger**

之前已经强调过了，如果数字较大，肯定按照String来处理，所以这一点可以通过BigInteger的构造方法来观察：

· 构造：public BigInteger(String val)；

而且在BigInteger类之中定义了一些基本的数学计算：

· 加法：public BigInteger add(BigInteger val)；

· 减法：public BigInteger subtract(BigInteger val)；

· 乘法：public BigInteger multiply(BigInteger val)；

· 除法（不保存余数）：public BigInteger divide(BigInteger val)；

· 除法（保存余数）：public BigInteger[] divideAndRemainder(BigInteger val)，数组第一个元素是商，第二个元素是余数；

**范例：**进行BigInteger的演示

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.math.BigInteger;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  BigInteger bigA = **new** BigInteger("234809234801");  BigInteger bigB = **new** BigInteger("8939834789");  System.*out*.println("加法结果：" + bigA.add(bigB));  System.*out*.println("减法结果：" + bigA.subtract(bigB));  System.*out*.println("乘法结果：" + bigA.multiply(bigB));  System.*out*.println("除法结果：" + bigA.divide(bigB));  BigInteger result[] = bigA.divideAndRemainder(bigB);  System.*out*.println("商：" + result[0] + "，余数：" + result[1]);  }  } |

**2、 大小数操作类：BigDecimal**

BigDecimal类表示的是大小数操作类，但是这个类也具备了与之前同样的基本计算方式，而在实际的工作之中，使用这个类最多的情况是进行准确位数的四舍五入操作，如果要完成这一操作需要关心BigDecimal类中的以下定义：

· 构造：public BigDecimal(double val)；

· 除法：public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)；

· 进位模式：public static final int ROUND\_HALF\_UP。

**范例：**完成准确位的四舍五入操作

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.math.BigDecimal;  **class** MyMath {  **public** **static** **double** round(**double** num, **int** scale) {  BigDecimal big = **new** BigDecimal(num);  BigDecimal result = big.divide(**new** BigDecimal(1), scale,  BigDecimal.*ROUND\_HALF\_UP*);  **return** result.doubleValue(); // Number类的方法  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(MyMath.*round*(15.5, 0)); // 16  System.*out*.println(MyMath.*round*(-15.5, 0)); // -15  System.*out*.println(MyMath.*round*(168.98765, 2)); // 168.99  }  } |

这个代码在日后的开发之中，就作为工具代码出现了，可以不用自己再去编写。

**3.8、数组操作类：Arrays（了解）**

排序操作：java.util.Arrays.sort(数组名称)，对于Arrays类一直是进行数组排序的操作，而Arrays类是定义在java.util包下的一个操作类，在这个类之中定义了所有的与数组有关的基本操作：二分查找、相等判断、填充等。

**范例：**测试一些

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Arrays;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **int**[] dataA = **new** **int**[] { 1, 2, 3 };  **int**[] dataB = **new** **int**[] { 1, 2, 3 };  System.*out*.println(Arrays.*equals*(dataA, dataB));  Arrays.*fill*(dataA, 3) ; // 填充数组  System.*out*.println(Arrays.*toString*(dataA));  }  } |

以上的代码只是作为Arrays类的基本使用，但是在Arrays类之中定义了一个方法：

· 为对象数组排序：public static void sort(Object[] a)。

**范例：**编写操作方法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Arrays;  **class** Person {  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]\n";  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per[] = **new** Person[] { **new** Person("张三", 20),  **new** Person("李四", 19), **new** Person("王五", 21) };  Arrays.*sort*(per) ; // 排序  System.*out*.println(Arrays.*toString*(per));  }  } |

这个时候没有任何的语法错误，即：程序的代码是正确的，但是在程序执行的时候出现了以下的问题：

|  |
| --- |
| Exception in thread "main" java.lang.ClassCastException: cn.mldn.demo.Person cannot be cast to java.lang.Comparable |

明确的告诉用户现在发生了“ClassCastException”，类转换异常，Person类不能变为Comparable实例。

## 3.9、比较器（重点）

如果现在要想为一组对象进行排序，那么必须有一个可以区分出对象大小的关系操作，而这个操作在Java之中就是利用比较器完成的。

### 3.9.1、常用比较器：Comparable（核心）

如果要为对象指定比较规则，那么对象所在的类必须实现Comparable接口，下面首先来看一下这个接口的定义：

|  |
| --- |
| **public** **interface** Comparable<T> {  **public** **int** compareTo(T o);  } |

根据文档的要求：要排序的数组所在的类一定要实现此接口，此接口返回的是int型数据，而用户覆写此方法的时候只需要返回三种结果：1（>0）、-1（<0）、0（=0）即可。

**范例：**实现比较器

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Arrays;  **class** Person **implements** Comparable <Person> {  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]\n";  }  @Override  **public** **int** compareTo(Person o) {  **if** (**this**.age > o.age) {  **return** 1;  } **else** **if** (**this**.age < o.age) {  **return** -1;  } **else** {  **return** 0;  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per[] = **new** Person[] { **new** Person("张三", 20),  **new** Person("李四", 19), **new** Person("王五", 21) };  Arrays.*sort*(per) ; // 排序  System.*out*.println(Arrays.*toString*(per));  }  } |

以后不管是何种情况下，只要牵扯到对象数组排序的操作，永远都是比较器Comparable。

### 3.9.2、Binary Tree，二叉树（了解）

二叉树是一种排序的基本的数据结构，而如果要想为多个对象进行排序，那么就必须可以区分出对象的大小，那么就必须依靠Comparable接口完成。

二叉树的基本原理：取第一个元素作为根节点，之后每一个元素的排列要求：如果比根节点小的数据放在左子树，如果比根节点大的数据放在右子树，在输出的时候采用中序遍历（左-根-右）的方式完成。

但是，不管是何种方式操作，一定要记住，这种数据结构的实现永远都需要依靠节点类，而这个时候的节点类要保存两个子节点：左、右。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Student **implements** Comparable<Student> {  **private** String name;  **private** **int** age;  **public** Student(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Student [name=" + name + ", age=" + age + "]\n";  }  @Override  **public** **int** compareTo(Student o) {  **if** (**this**.age > o.age) {  **return** -1;  } **else** **if** (**this**.age < o.age) {  **return** 1;  } **else** {  **return** 0;  }  }  }  **class** BinaryTree {  @SuppressWarnings("rawtypes")  **private** **class** Node {  **private** Comparable data;  **private** Node left;  **private** Node right;  **public** Node(Comparable data) {  **this**.data = data;  }  @SuppressWarnings("unchecked")  **public** **void** addNode(Node newNode) {  **if** (**this**.data.compareTo(newNode.data) < 0) {  **if** (**this**.left == **null**) { // 没有左子树  **this**.left = newNode;  } **else** {  **this**.left.addNode(newNode);  }  } **else** {  **if** (**this**.right == **null**) {  **this**.right = newNode;  } **else** {  **this**.right.addNode(newNode);  }  }  }  **public** **void** printNode() {  **if** (**this**.left != **null**) {  **this**.left.printNode();  }  System.*out*.println(**this**.data);  **if** (**this**.right != **null**) {  **this**.right.printNode();  }  }  }  **private** Node root; // 根节点  **public** **void** add(Student data) {  **if** (data == **null**) {  **return**;// 返回调用处  }  Node newNode = **new** Node(data); // 将数据封装为节点  **if** (**this**.root == **null**) { // 没有根节点  **this**.root = newNode;  } **else** { // 保存到合适的位置  **this**.root.addNode(newNode);  }  }  **public** **void** print() {  **if** (**this**.root != **null**) {  **this**.root.printNode();  }  }  }  **public** **class** BinaryTreeDemo {  **public** **static** **void** main(String[] args) {  BinaryTree tree = **new** BinaryTree();  tree.add(**new** Student("张三", 20));  tree.add(**new** Student("李四", 19));  tree.add(**new** Student("王五", 21));  tree.print();  }  } |

如果不是笔试之中出现，基本上就出现不了了，所以以上的代码，作为了解，清楚就行了，实际上也别用。

### 3.9.3、挽救的比较器（了解）

之前使用的Comparable实际上是在一个类定义的时候就已经具备的功能了，但是如果说现在一个类已经定义完成了。

**范例：**已经定义好了一个Person类

|  |
| --- |
| **class** Person {  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]\n";  }  **public** **int** getAge() {  **return** age;  }  **public** String getName() {  **return** name;  }  } |

这个类在设计之初并没有考虑到排序的要求，可是后来又需要进行排序，但是以上的类已经无法修改了，那么如何使用Arrays.sort()排序呢？为了解决这样的问题，在java里面又提供了另外一个比较器接口：java.util.Comparator接口，这个接口的定义如下：

|  |
| --- |
| **public** **interface** Comparator<T> {  **public** **int** compare(T o1, T o2);  **public** **boolean** equals(Object obj);  } |

在compare()方法上存在了两个参数用于比较大小，而要想使用这个接口，需要单独定义一个比较规则类。

**范例：**专门为Person类补救一个比较规则类

|  |
| --- |
| **class** PersonComparator **implements** Comparator<Person> {  @Override  **public** **int** compare(Person o1, Person o2) {  **if** (o1.getAge() > o2.getAge()) {  **return** 1;  } **else** **if** (o1.getAge() < o2.getAge()) {  **return** -1;  } **else** {  **return** 0;  }  }  } |

而后如果要进行排序的话，依然使用Arrays类的sort()方法：public static <T> void sort(T[] a, Comparator<? super T> c)

**范例：**实现排序

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per[] = **new** Person[] { **new** Person("张三", 20),  **new** Person("李四", 19), **new** Person("王五", 21) };  Arrays.*sort*(per, **new** PersonComparator()); // 排序  System.*out*.println(Arrays.*toString*(per));  }  } |

很明显，使用Comparator要比Comparable麻烦许多，所以以后别使它。

**面试题：**请解释Comparable和Comparator的区别？（请解释两种比较器的区别）

· java.lang.Comparable是在一个类定义的时候默认实现好的接口，里面只有一个compareTo()方法；

· java.util.Comparator是需要单独定义一个比较的规则类，里面有两个方法；compare()、equals()。

**3.10、对象克隆（理解）**

克隆就是对象的复制操作，在Object类中存在一个clone()方法：

· 克隆方法：**protected** Object clone() throws CloneNotSupportedException；

但是在这个方法上抛出了一个“CloneNotSupportedException”，不支持的克隆异常，这个异常表示的是，要克隆对象的类必须实现Cloneable接口，但是这个接口没有任何的方法，这个接口属于标识接口，只表示一种能力。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person **implements** Cloneable{ // 表示一种能力，没有方法覆写  **private** String name;  **private** **int** age;  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]\n";  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return** **super**.clone(); // 调用父类克隆  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person("张三", 20) ;  Person p = (Person) per.clone() ;  p.setName("王五") ;  System.*out*.println(per);  System.*out*.println(p);  }  } |

对于此类操作，清楚clone()方法和Cloneable接口的作用就行了，至于代码的实现，个人认为：意义不大。

## 3.11、正则表达式（重点）

### 3.11.1、正则表达式的引出

下面完成一个简单的程序，在之前曾经讲解过这样一段代码：判断一个字符串是否由数字组成。

**范例：**代码实现

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = " 13412a 4124214141232" ;  **if** (*isNumber*(str)) {  System.*out*.println("是由数字所组成！");  } **else** {  System.*out*.println("不是由数字所组成！");  }  }  **public** **static** **boolean** isNumber(String data) {  **char** arr [] = data.toCharArray() ; // 字符串变为字符数组  **for** (**int** x = 0; x < arr.length; x++) {  **if** (arr[x] < '0' || arr[x] > '9') {  **return** **false** ;  }  }  **return** **true** ;  }  } |

但是现在一个简单的问题就出现了，这个验证应该算是不难的，但是面对这样一个不麻烦的验证，代码写了9行代码，如果是一些更为复杂的验证呢？那么对于整体操作就更加的麻烦了，现在有另外一种做法：

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "1341224124214141232" ;  **if** (str.matches("**\\d+**")) {  System.*out*.println("是由数字所组成！");  } **else** {  System.*out*.println("不是由数字所组成！");  }  }  } |

很明显，现在对于程序而言，第二种方式更加的简单，而第二种方式就是使用了正则表达式的概念，而其中的“\\d+”代码就属于正则表达式的程序代码。

正则表达式最早是在Linux下发展起来的，但是由于其使用方便，在JDK 1.4的时候将其正式引入到了Java的开发行列之中，而在JDK 1.4之前如果要想使用正则表达式，那么需要单独配置一个Apache的开发包，而JDK 1.4之后除了引入了正则的支持之外，同时也对String类进行了一些修改，可以让其直接操作正则。

在JDK 1.4之后专门引入了一个java.util.regex开发包，这个包之中有两个主要类负责完成正则的开发：Pattern（定义并且编译正则的匹配模板）、Matcher（匹配应用），之所以现在不去关心这两个类，主要是因为JDK 1.4之后对String类的修改达到了一个很好的正则操作效果，主要使用String类完成。

### 3.11.2、常用正则匹配符号（背，死了都要背）

所有的正则匹配的符号都在java.util.regex.Pattern类之中进行定义，下面分别对这些常用的符号做一些介绍。

1、 字符：匹配**单个**字符

· a：表示匹配字母a；

· \\：匹配转义字符“\”；

· \t：匹配转义字符“\t”；

· \n：匹配转义字符“\n”；

2、 一组字符：任意匹配里面的一个单个字符；

· [abc]：表示可能是字母a，可能是字母b或者是字母c；

· [^abc]：表示不是字母a、字母b、字母c的任意一个；

· [a-zA-Z]：表示全部字母中的任意一个；

· [0-9]：表示全部数字的任意一个；

3、 边界匹配：在以后编写JavaScript的时候使用正则中要使用到；

· ^：表示一组正则的开始；

· $：表示一组正则的结束；

4、 简写表达式：每一位出现的简写标记也只表示一位；

· .：表示任意的一位字符；

· \d：表示任意的一位数字，等价于“[0-9]”；

· \D：表示任意的一位非数字，等价于“[^0-9]”；

· \w：表示任意的一位字母、数字、\_，等价于“[a-zA-Z0-9\_]”；

· \W：表示任意的一位非字母、数字、\_，等价于“[^a-zA-Z0-9\_]”；

· \s：表示任意的一位空格，例如：\n、\t等；

· \S：表示任意的一位非空格；

5、 数量表示：之前的所有正则都只是表示一位，如果要想表示多位，则就需要数量表示。

· 正则表达式?：此正则出现0次或1次；

· 正则表达式\*：此正则出现0次、1次或多次；

· 正则表达式+：此正则出现1次或多次；

· 正则表达式{n}：此正则出现正好n次；

· 正则表达式{n,}：此正则出现n次以上；

· 正则表达式{n,m}：此正则出现n ~ m次。

6、 逻辑表示：与、或、非

· 正则表达式A正则表达式B：表达式A之后紧跟着表达式B；

· 正则表达式A|正则表达式B：表示表达式A或者是表达式B，二者任选一个出现；

· (正则表达式)：将多个子表达式合成一个表示，作为一组出现。

### 3.11.3、String类对正则的支持

在JDK 1.4之后，String类对正则有了直接的方法支持，只需要通过如下的几个方法就可以操作正则。

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public boolean matches(String regex) | 普通 | 与指定正则匹配 |
| 2 | public String replaceAll(String regex, String replacement) | 普通 | 替换满足指定正则的全部内容 |
| 3 | public String replaceFirst(String regex, String replacement) | 普通 | 替换满足指定正则的首个内容 |
| 4 | public String[] split(String regex) | 普通 | 按照指定正则全拆分 |
| 5 | public String[] split(String regex, int limit) | 普通 | 按照指定的正则拆分为指定个数 |

**Pattern类之中存在的方法：**

· 字符串全拆分：public String[] split(CharSequence input)；

· 字符串部分拆分：public String[] split(CharSequence input, int limit)；

**Matterh类之中存在的方法：**

· 字符串匹配：public boolean matches()；

· 字符串全替换：public String replaceAll(String replacement)；

· 字符串替换首个：public String replaceFirst(String replacement)。

正是因为String类支持的方法比较全面，所以在开发之中，主要的都是String类操作正则，因为方便。下面编写若干个操作实例来进行正则的验证。

**范例：**字符串拆分

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "a1bb2ccc3dddd4eeeee5fffffff6ggggggg7" ;  String regex = "[a-zA-Z]+" ;// 字母出现1次或多次  System.*out*.println(str.replaceAll(regex, ""));  System.*out*.println(str.replaceFirst(regex, ""));  }  } |

**范例：**字符串拆分

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "a1bb2ccc3dddd4eeeee5fffffff6ggggggg7" ;  String regex = "\\d" ;// 数字出现1次，[0-9]  String result [] = str.split(regex) ;  **for** (**int** x = 0; x < result.length; x++) {  System.*out*.print(result[x] + "、");  }  }  } |

有了正则，对于字符串日后的种种操作就会变的相当的方便。而正则在使用的过程之中最为重要的部分就是验证部分，因为一些字符串必须满足于指定的格式才可以操作。

**范例：**做一个简单的验证，要求字符串格式“aaaaab”，在b之前可以有无数多个a，但是不能没有，至少1个。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "aaaaaab" ;  String regex = "a+b" ;  System.*out*.println(str.matches(regex));  }  } |

**范例：**验证一个字符串是否是整型数据，如果是则将其变为int型数据，而后执行乘法操作

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "123" ;  String regex = "**\\d+**" ;  **if** (str.matches(regex)) { // 符合于验证要求  **int** data = Integer.*parseInt*(str) ; // 字符串变为int型数据  System.*out*.println(data \* data);  } **else** {  System.*out*.println("字符串不是数字所组成！");  }  }  } |

**范例：**验证字符串是否是小数，如果是则将其变为double型数据，而后执行乘法操作

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "123.1" ;  String regex = "**\\d+(\\.\\d+)?**" ;  **if** (str.matches(regex)) { // 符合于验证要求  **double** data = Double.*parseDouble*(str) ; // 字符串变为double型数据  System.*out*.println(data \* data);  } **else** {  System.*out*.println("字符串不是数字所组成！");  }  }  } |

**范例：**输入一个字符串，按照年-月-日 时-分-秒的形式，如果正确，则将其变为Date型数据

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "1998-09-18 10:10:10.100" ;  String regex = "**\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}\\.\\d{3}**" ;  **if** (str.matches(regex)) { // 符合于验证要求  Date date = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")  .parse(str);  System.*out*.println(date);  } **else** {  System.*out*.println("字符串不是日期！");  }  }  } |

**范例：**一个用户名只能由字母、数字、\_所组成，其长度只能是6~15位

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "343234FDSF\_";  String regex = "**\\w{6,15}**";  **if** (str.matches(regex)) { // 符合于验证要求  System.*out*.println("用户名合法。");  } **else** {  System.*out*.println("用户名非法！");  }  }  } |

**范例：**现在编写一个正则表达式要求验证电话号码是否正确，例如：现在写上咱们电话号码，有如下几种写法：

· **写法一：**51283346 🡺 \\d{7,8}；

· **写法二：**01051283346 🡺 (\\d{3,4})?\\d{7,8}；

· **写法三：**010-51283346 🡺 (\\d{3,4})?-?\\d{7,8}；

· **写法四：**(010)-51283346 🡺 ((\\d{3,4}|\\(\\d{3,4}\\))-?)?\\d{7,8}；

· **写法五：**(010)51283346 🡺 ((\\d{3,4}|\\(\\d{3,4}\\))-?)?\\d{7,8}。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "51283346";  // 第一步：\\d{7,8}，因为电话号码可能是7~8位所组成；  // 第二步：(\\d{3,4})?\\d{7,8}，因为区号由3~4位所组成；  // 第三步：(\\d{3,4})?-?\\d{7,8}，因为-可能出现也可能不出现  // 第四步：((\\d{3,4}|\\(\\d{3,4}\\))-?)?\\d{7,8}  String regex = "((\\d{3,4}|\\(\\d{3,4}\\))-?)?\\d{7,8}";  **if** (str.matches(regex)) { // 符合于验证要求  System.*out*.println("TRUE，电话号码输入合法。");  } **else** {  System.*out*.println("FLASE，电话号码输入非法！");  }  }  } |

**范例：**现在要求验证一个email地址，email地址的用户名由字母、数字、\_、.所组成，其中不能以数字和.开头，而且email地址的域名只能是.com、.cn、.net

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "mldnqa@163.net";  String regex = "[a-zA-Z\_][a-zA-Z\_0-9\\.]\*@[a-zA-Z\_0-9\\.]+\\.(com|cn|net)";  **if** (str.matches(regex)) { // 符合于验证要求  System.*out*.println("TRUE，EMAIL输入合法。");  } **else** {  System.*out*.println("FLASE，EMAIL输入非法！");  }  }  } |

**范例：**现在判断一个字符串的组成，其组成原则“姓名:年龄:成绩|姓名:年龄:成绩|姓名:年龄:成绩|…”

例如，现在有一个字符串内容如下：“SMITH:19:90.1|ALLEN:18:90.1|KING:20:95.2|”，其中姓名的组成只能是字母，年龄只能是数字，成绩可能是小数。

**第一问：**写出来验证的正则表达式

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "SMITH:19:90.1|ALLEN:18:90.1|KING:20:95.2";  String regex = "([a-zA-Z]+:\\d{1,3}:\\d{1,3}(\\.\\d{1,2})?\\|)+([a-zA-Z]+:\\d{1,3}:\\d{1,3}(\\.\\d{1,2})?)?";  **if** (str.matches(regex)) { // 符合于验证要求  System.*out*.println("TRUE，输入合法。");  } **else** {  System.*out*.println("FLASE，输入非法！");  }  }  } |

**第二问：**将所有的数据排序，按照成绩由高到低排序，如果成绩相同，则按照年龄由低到高排序

既然是排序操作，现在可以想到的方法就是：Arrays.sort()，但是现在这些数据可以将其变为对象数组排序，利用比较器完成。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Arrays;  **class** Student **implements** Comparable<Student> {  **private** String name ;  **private** **int** age ;  **private** **double** score ;  **public** Student(String name,**int** age,**double** score) {  **this**.name = name ;  **this**.age = age ;  **this**.score = score ;  }  @Override  **public** String toString() {  **return** "Student [name=" + name + ", age=" + age + ", score=" + score  + "]\n";  }  @Override  **public** **int** compareTo(Student o) {  **if** (**this**.score > o.score) {  **return** -1 ;  } **else** **if** (**this**.score < o.score) {  **return** 1 ;  } **else** {  **if** (**this**.age > o.age) {  **return** 1 ;  } **else** **if** (**this**.age < o.age){  **return** -1 ;  } **else** {  **return** 0 ;  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "SMITH:19:90.1|ALLEN:18:90.1|KING:20:95.2";  String regex = "([a-zA-Z]+:\\d{1,3}:\\d{1,3}(\\.\\d{1,2})?\\|)+([a-zA-Z]+:\\d{1,3}:\\d{1,3}(\\.\\d{1,2})?)?";  **if** (str.matches(regex)) { // 符合于验证要求  String result [] = str.split("\\|") ; // 拆分  Student stu [] = **new** Student[result.length]; // 此时的数组大小就是对象数组大小  **for** (**int** x = 0; x < result.length; x++) {  String temp [] = result[x].split(":") ;  stu[x] = **new** Student(temp[0], Integer.*parseInt*(temp[1]),  Double.*parseDouble*(temp[2]));  }  Arrays.*sort*(stu) ;  System.*out*.println(Arrays.*toString*(stu));  } **else** {  System.*out*.println("FLASE，输入非法！");  }  }  } |

因为在日后的所有项目的开发之中，只要是输入的数据都是利用字符串接收的，那么字符串又可以向各个数据类型转换，而要转换之前为了保证转换成功一定需要进行验证，验证最方便的就是正则表达式。

## 3.12、反射机制（理解，基本上本人不太奢求你们会）

反射机制如果只是针对于普通开发者而言，意义不大，一般都是作为一些系统的架构设计去使用的，包括以后学习的开源框架，那么几乎都反射机制。

### 3.12.1、认识反射（可以不会的重点）

反射之中包含了一个“反”的概念，所以要想解释反射就必须先从“正”开始解释，一般而言，当用户使用一个类的时候，应该先知道这个类，而后通过这个类产生实例化对象，但是“反”指的是通过对象找到类。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {}  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person() ; // 正着操作  System.*out*.println(per.getClass().getName()); // 反着来  }  } |

以上的代码使用了一个getClass()方法，而后就可以得到对象所在的“包.类”名称，这就属于“反”了，但是在这个“反”的操作之中有一个getClass()就作为发起一切反射操作的开端。

Person的父类是Object类，而上面所使用getClass()方法就是Object类之中所定义的方法。

· **取得Class对象：**public final Class<?> getClass()，**反射之中的所有泛型都定义为?，返回值都是Object。**

而这个getClass()方法返回的对象是Class类的对象，所以这个Class就是所有反射操作的源头。但是在讲解其真正使用之前还有一个需要先解释的问题，既然Class是所有反射操作的源头，那么这个类肯定是最为重要的，而如果要想取得这个类的实例化对象，Java中定义了三种方式：

**方式一：**通过Object类的getClass()方法取得，基本不用：

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {}  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person() ; // 正着操作  Class<?> cls = per.getClass() ; // 取得Class对象  System.*out*.println(cls.getName()); // 反着来  }  } |

**方式二：**使用“类.class”取得，在日后学习Hibernate开发的时候使用

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {}  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Person.**class** ; // 取得Class对象  System.*out*.println(cls.getName()); // 反着来  }  } |

**方式三：**使用Class类内部定义的一个static方法，主要使用

· **取得Class类对象：**public static Class<**?**> forName(String **className**) throws ClassNotFoundException；

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {}  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  System.*out*.println(cls.getName()); // 反着来  }  } |

那么现在一个新的问题又来了，取得了Class类的对象有什么用处呢？对于对象的实例化操作之前一直依靠构造方法和关键字new完成，可是有了Class类对象之后，现在又提供了另外一种对象的实例化方法：

· **通过反射实例化对象：public T newInstance() throws InstantiationException, IllegalAccessException**；

**范例：**通过反射实例化对象

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person {  @Override  **public** String toString() {  **return** "Person Class Instance .";  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("**cn.mldn.demo.Person**") ; // 取得Class对象  Object obj = cls.newInstance() ; // 实例化对象，和使用关键字new一样  Person per = (Person) obj ; // 向下转型  System.*out*.println(per);  }  } |

那么现在可以发现，对于对象的实例化操作，除了使用关键字new之外又多了一个反射机制操作，而且这个操作要比之前使用的new复杂一些，可是有什么用？

对于程序的开发模式之前一直强调：尽量减少耦合，而减少耦合的最好做法是使用接口，但是就算使用了接口也逃不出关键字new，所以实际上new是造成耦合的关键元凶。

**范例：**回顾一下之前所编写的工厂设计模式

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Fruit {  **public** **void** eat() ;  }  **class** Apple **implements** Fruit {  **public** **void** eat() {  System.*out*.println("吃苹果。");  };  }  **class** Factory {  **public** **static** Fruit getInstance(String className) {  **if** ("apple".equals(className)){  **return** **new** Apple() ;  }  **return** **null** ;  }  }  **public** **class** FactoryDemo {  **public** **static** **void** main(String[] args) {  Fruit f = Factory.*getInstance*("apple") ;  f.eat() ;  }  } |

以上为之前所编写最简单的工厂设计模式，但是在这个工厂设计模式之中有一个最大的问题：如果现在接口的子类增加了，那么工厂类肯定需要修改，这是它所面临的最大问题，而这个最大问题造成的关键性的病因是new，那么如果说现在不使用关键字new了，变为了反射机制呢？

反射机制实例化对象的时候实际上只需要“包.类”就可以，于是根据此操作，修改工厂设计模式。

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Fruit {  **public** **void** eat() ;  }  **class** Apple **implements** Fruit {  **public** **void** eat() {  System.*out*.println("吃苹果。");  };  }  **class**Orange**implements** Fruit {  **public** **void** eat() {  System.*out*.println("吃橘子。");  };  }  **class** Factory {  **public** **static** Fruit getInstance(String className) {  Fruit f = **null** ;  **try** {  **f = (Fruit) Class.*forName*(className).newInstance() ;**  } **catch** (Exception e) {  e.printStackTrace();  }  **return** f ;  }  }  **public** **class** FactoryDemo {  **public** **static** **void** main(String[] args) {  Fruit f = Factory.*getInstance*("**cn.mldn.demo.Orange**") ;  f.eat() ;  }  } |

发现，这个时候即使增加了接口的子类，工厂类照样可以完成对象的实例化操作，这个才是真正的工厂类，可以应对于所有的变化。如果单独从开发角度而言，与开发者关系不大，但是对于日后学习的一些框架技术这个就是它实现的命脉，**在日后的程序开发上，如果发现操作的过程之中需要传递了一个完整的“包.类”名称的时候几乎都是反射机制作用。**

### 3.12.2、反射的深入应用（看看就行了）

以上只是利用了Class类作为了反射实例化对象的基本应用，但是对于一个实例化对象而言，它需要调用类之中的构造方法、普通方法、属性，而这些操作都可以通过反射机制完成。

#### 3.12.2.1、调用构造

使用反射机制也可以取得类之中的构造方法，这个方法在Class类之中已经明确定义了：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public Constructor<?>[] getConstructors() throws SecurityException | 普通 | 取得一个类的全部构造 |
| 2 | public Constructor<T> getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException | 普通 | 取得一个类的指定参数构造 |

现在发现以上的两个方法返回的都是java.lang.reflect.Constructor类的对象。

**范例：**取得一个类之中的全部构造

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Constructor;  **class** Person { // CTRL + K  **public** Person() {}  **public** Person(String name) {}  **public** Person(String name,**int** age) {}  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  Constructor<?> cons [] = cls.getConstructors() ; // 取得全部构造  **for** (**int** x = 0; x < cons.length; x++) {  System.*out*.println(cons[x]);  }  }  } |

**验证：**在之前强调的一个简单Java类必须存在一个无参构造方法

**范例：**观察没有无参构造的情况

|  |
| --- |
| **package** cn.mldn.demo;  **class** Person { // CTRL + K  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]";  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  Object obj = **cls.newInstance()** ; // 实例化对象  System.*out*.println(obj);  }  } |

此时程序运行的时候出现了错误提示“**java.lang.InstantiationException**”，因为以上的方式使用反射实例化对象时需要的是类之中要提供无参构造方法，但是现在既然没有了无参构造方法，那么就必须明确的找到一个构造方法，而后利用Constructor类之中的新方法实例化对象：

· 实例化对象：public T newInstance(Object... initargs) throws InstantiationException, IllegalAccessException,

IllegalArgumentException, InvocationTargetException

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Constructor;  **class** Person { // CTRL + K  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + "]";  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  // 取得指定参数类型的构造方法  Constructor<?> cons = cls.getConstructor(String.**class**,**int**.**class**) ;  Object obj = cons.newInstance("张三", 20); // 为构造方法传递参数  System.*out*.println(obj);  }  } |

很明显，调用无参构造方法实例化对象要比调用有参构造的更加简单、方便，所以在日后的所有开发之中，凡是有简单Java类出现的地方，都一定要提供无参构造。

#### 3.12.2.2、调用普通方法

当取得了一个类实例化对象之后，下面最需要调用的肯定是类之中的方法，所以可以继续使用Class类取得一个类中所定义的方法定义：

· 取得全部方法：public Method[] getMethods() throws SecurityException；

· 取得指定方法：public Method getMethod(String name, Class<?>... parameterTypes) throws

NoSuchMethodException, SecurityException

发现以上的方法返回的都是java.lang.reflect.Method类的对象。

**范例：**取得一个类之中所定义的全部方法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Method;  **class** Person {  **private** String name ;  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getName() {  **return** name;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  Method met [] = cls.getMethods() ; // 取得全部方法  **for** (**int** x = 0; x < met.length; x++) {  System.*out*.println(met[x]);  }  }  } |

但是取得了Method类对象最大的作用不再于方法的列出（方法的列出都在开发工具上使用了），但是对于取得了Method类对象之后还有一个最大的功能，就是可以利用反射调用类中的方法：

· 调用方法：public **Object** invoke(Object obj, Object... args) throws IllegalAccessException,

IllegalArgumentException, InvocationTargetException

之前调用类中方法的时候使用的都是“对象.方法”，但是现在有了反射之后，可以直接利用Object类调用指定子类的操作方法。（同时解释一下，为什么setter、getter方法的命名要求如此严格）。

**范例：**利用反射调用Person类之中的setName()、getName()方法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Method;  **class** Person {  **private** String name ;  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getName() {  **return** name;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  Object obj = cls.newInstance(); // 实例化对象，没有向Person转型  String attribute = "name" ; // 要调用类之中的属性  Method setMet = cls.getMethod("set" + *initcap*(attribute), String.**class**);// setName()  Method getMet = cls.getMethod("get" + *initcap*(attribute));// getName()  setMet.invoke(obj, "张三") ; // 等价于：Person对象.setName("张三")  System.*out*.println(getMet.invoke(obj));// 等价于：Person对象.getName()  }  **public** **static** String initcap(String str) {  **return** str.substring(0,1).toUpperCase().concat(str.substring(1)) ;  }  } |

在日后的所有框架技术开发之中，简单Java类都是如此应用的，所以必须按照标准进行。

#### 3.12.2.3、调用成员

类之中最后一个组成部分就是成员（Field，也可以称为属性），如果要通过反射取得类的成员可以使用方法如下：

· 取得本类的全部成员：public Field[] getDeclaredFields() throws SecurityException；

· 取得指定的成员：public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException；

这两个方法的返回值类型是java.lang.reflect.Field类的对象，下面首先观察如何取得一个类之中的全部属性。

**范例：**取得一个类之中的全部属性

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Field;  **class** Person {  **private** String name ;  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person") ; // 取得Class对象  Field field [] = cls.getDeclaredFields() ; // 取得全部属性  **for** (**int** x = 0; x < field.length; x++) {  System.*out*.println(field[x]);  }  }  } |

但是找到Field实际上就找到了一个很有意思的操作，在Field类之中提供了两个方法：

· **设置属性内容（类似于：对象.属性 = 内容）：**public void set(Object obj, Object value)

throws IllegalArgumentException, IllegalAccessException；

· **取得属性内容（类似于：对象.属性）：**public Object get(Object obj)

throws IllegalArgumentException, IllegalAccessException

可是从类的开发要求而言，一直都强调类之中的属性必须封装，所以现在调用之前要想办法解除封装。

· **解除封装：**public void setAccessible(boolean flag) throws SecurityException；

**范例：**利用反射操作类中的属性

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.lang.reflect.Field;  **class** Person {  **private** String name;  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> cls = Class.*forName*("cn.mldn.demo.Person"); // 取得Class对象  Object obj = cls.newInstance(); // 对象实例化属性才会分配空间  Field nameField = cls.getDeclaredField("name") ; // 找到name属性  nameField.setAccessible(**true**) ; // 解除封装了  nameField.set(obj, "张三") ; // Person对象.name = "张三"  System.*out*.println(nameField.get(obj)); // Person对象.name  }  } |

虽然反射机制运行直接操作类之中的属性，可是不会有任何一种程序直接操作属性，都会通过setter、getter方法。

# 4、总结

1、 频繁修改字符串使用StringBuffer，append()为连接字符串使用；

2、 日期、基本类型、String之间的互相转换；

3、 使用BigDecimal进行准确的四舍五入操作；

4、 使用Comparable实现的比较器；

5、 正则表达式一定要背下来，包括讲解过的各种使用；

6、 使用反射机制依然可以进行对象的实例化操作。

# 5、预习任务

File、InputStream、OutputStream、Writer、Reader、FileInputStream、FileOutputStream、FileWriter、FileReader、ByteArrayOutputStream、ByteArrayInputStream、PrintStream、OutputStreamWriter、InputStreamReader、BufferedReader、Scanner、ObjectOutputStream、ObjectInputStream、Serializable。

# 6、作业

今天又给出了三个重要的代码模型：

1、 使用SimpleDateFormat类格式化；

2、 比较器的使用；

3、 正则表达式。