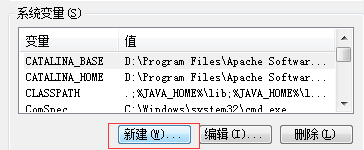
# Java语言基础

## JDK安装

* 安装JDK

在Oracle官网上下载适合运行操作系统版本的JDK，然后更具提示进行安装。安装完成需需要对JDK进行“环境变量”的配置（限于Windows操作系统）。

控制面板-----🡪系统-----🡪高级系统设置-----🡪环境变量-----🡪系统变量（新建）



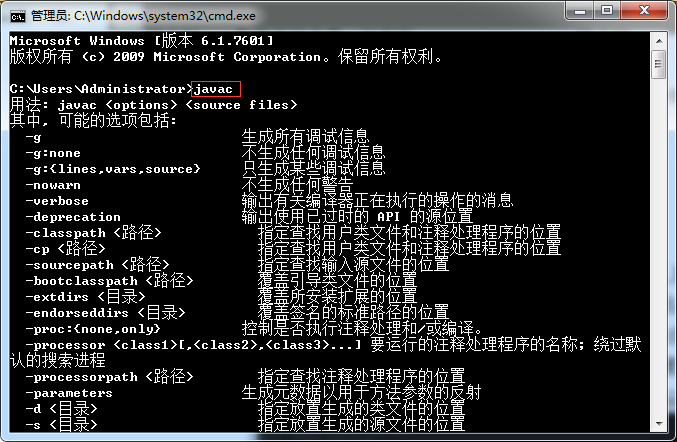
* 环境变量配置

点击新建弹出“新建系统变量”对话框，在“变量名”文本框中输入“JAVA\_HOME”，在“变量值”文本框中输入JDK的安装路径，如：“D:\Program Files\Java\jdk1.8.0\_144”。单击“确定”按钮，完成环境变量JAVA\_HOME的配置。

在系统变量中查找Path变量，如果不存在，则新建系统变量Path；选中该变量，点击“编辑”按钮，将“%JAVA\_HOME%\bin;”添加到“变量值”的文本框起始位置。注意，变量值之间必须用分号分割。

* 测试开发环境

打开“运行”敞口。输入“cmd”并单击“运行”按钮，进入DOS环境中。在命令提示符后面直接输入“javac”，按回车键，系统会输出javac的帮助信息，则说明成功配置了JDK。



## Java程序的基本结构

**package** com.ucai.test;

**public** **class** Test {

**static** **int** *ONE* = 1;

**public** **static** **void** main(String[] args) {

String var = "Hello";

System.***out***.println(*ONE*);

System.***out***.println(var);

}

}

第一条语句中的package关键字定义了Java程序中类所在的包是Mr，Mr是一个标识符。

第二条语句中的public class关键字定义了名为Example的类。public为一个修饰词，class为定义类的关键字。在Java程序中类名一般以大写字母开头。

第三条语句中static为一个修饰词，int关键字表明变量的数据类型。ONE为一个标识符，变量的名称。

第四条语句中public static void main(String[] args)是类的主方法，是Java应用程序的入口，Java程序是从该方法开始执行的。此段语句的格式固定不允许修改。

第五条语句定义了一个局部变量。

第六条语句中定义了一个输出语句，用于将结果输出到控制台。

其中Java程序中是区分大小写的，而且每段语句都以分号结尾区分。

## 标识符和关键字

### 标识符

标识符是Java程序中必须使用的，但也不是随便使用的，有一定的规则。

标识符可以简单地理解为一个名字，用来标志类名、变量名、方法名、数组名、文件名的有效字符序列。标识符是由程序员自定义的，必须是以遵循以下规则

1. Java语言中的标识符由字母、数字、下划线、美元符号自称，第一个字符不能是数字。
2. Java语言中的表示符可以是Unicode字符集中的任何字符。
3. 标识符不能是Java中的关键字和保留字。
4. Java语言中的标识符是区分大小写的。

### 关键字

在Java中定义了一些关键字用于表示特定的意义，不能用于其他使用。

|  |  |  |
| --- | --- | --- |
| **类型** | **关键字** | **描述** |
| 访问控制 | private | private 关键字表示私有的。使用了该关键字修饰的类、方法、变量只能在声明这些类、方法、变量的类中使用这些类、方法、变量。在声明类的外部或声明类的子类中都不能引用到这些修饰了的类、方法、变量。 |
| protected | protected关键字表示受保护的。使用该关键字修饰的类、方法、变量可以在这些类、方法、变量所在的同一个包中的任何地方使用。但是其他包中的数据不能引用这些数据。 |
| public | public关键字表示公共的。使用该关键字修饰的类、方法、变量程序的任何位置引用到这些数据 |
| 类、方法和变量修饰符 | abstract | abstract关键字表示抽象的，可以修饰类和方法。用于定义类和方法为抽象类和抽象方法。定义的抽象类和抽象方法 |
| class | 定义类的关键字 |
| extends | 表明一个类型是另一个类型的子类型，这里常见的类型有类和接口。定义继承关系 |
| final | final在Java中是一个保留的关键字，可以声明成员变量、方法、类以及本地变量。用来说明最终属性，表明一个类不能派生出子类，或者成员方法不能被覆盖，或者成员域的值不能被改变，用来定义常量 |
| implements | 表明一个类实现了给定的接口 |
| interface | 定义接口 |
| native | Java不是完美的， Java无法直接访问到操作系统底层（如系统硬件等)，为此Java使用native方法来扩展Java程序的功能。native是方法修饰符。Native方法是由另外一种语言（如c/c++，FORTRAN，汇编）实现的本地方法。因为在外部实现了方法，所以在java代码中，就不需要声明了，有点类似于借口方法。Native可以和其他一些修饰符连用，但是abstract方法和Interface方法不能用native来修饰 |
| new | 用来创建新实例对象 |
| static | 静态的，表明具有静态属性 |
| strictfp | strictfp 关键字可应用于类、接口或方法。使用 strictfp 关键字声明一个方法时，该方法中所有的float和double表达式都严格遵守FP-strict的限制,符合IEEE-754规范。当对一个类或接口使用 strictfp 关键字时，该类中的所有代码，包括嵌套类型中的初始设定值和代码，都将严格地进行计算。严格约束意味着所有表达式的结果都必须是 IEEE 754 算法对操作数预期的结果，以单精度和双精度格式表示。如果你想让你的浮点运算更加精确，而且不会因为不同的硬件平台所执行的结果不一致的话，可以用关键字strictfp. |
| synchronized | Java语言的关键字，可用来给对象和方法或者代码块加锁，当它锁定一个方法或者一个代码块的时候，同一时刻最多只有一个线程执行这段代码。当两个并发线程访问同一个对象object中的这个加锁同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。然而，当一个线程访问object的一个加锁代码块时，另一个线程仍然可以访问该object中的非加锁代码块。 |
| transient | java语言的关键字，变量修饰符，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。换句话来说就是，用transient关键字标记的成员变量不参与序列化过程。 |
| volatile | volatile是一个类型修饰符（type specifier），就像大家更熟悉的const一样，它是被设计用来修饰被不同线程访问和修改的变量。volatile的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。 |
| 程序控制语句 | break | 该关键字用于switch和循环语句中，用于退出代码块，不在执行代码。 |
| continue | 该关键字用于循环语句中，用于退出本次循环，继续执行下次循环。 |
| return | 该关键字用于方法中返回数据。 |
| do | 用于while循环中，实现do….while |
| while | while循环关键字 |
| if | if判断语句关键字 |
| else | 用于if判断语句中的另一种情况if….else |
| for | for循环的关键字 |
| instanceof | instanceof 运算符是用来在运行时指出对象是否是特定类的一个实例。 |
| switch | switch语句的关键字 |
| case | switch语句的关键字，case指定了匹配的项 |
| default | switch语句的关键字，default指定了case指定项以外的可能。 |
| 错误处理 | try | 定义异常处理关键字，定义可能出现异常的代码块 |
| catch | 定义异常处理关键字，定义抛出异常的类型 |
| throw | throw关键字通常用在方法体中，并且抛出一个异常对象。 |
| throws | throws关键字通常被应用在声明方法时，用来指定可能抛出的异常。多个异常可以使用逗号隔开。当在主函数中调用该方法时，如果发生异常，就会将异常抛给指定异常对象 |
| 包相关 | import | 导入包的关键字 |
| package | 定义包的关键字 |
| 基本类型 | boolean | 定义布尔值变量的关键字 |
| byte | 定义byte类型整型数值的关键字 |
| char | 定义char类型字符的关键字 |
| double | 定义double类型双精度的小数的关键字 |
| float | 定义float类型单精度的小数的关键字 |
| int | 定义int类型整型数值的关键字 |
| long | 定义long类型整型数值的关键字 |
| short | 定义short类型整型数值的关键字 |
| null | 表示空值 |
| true | 表示为真的关键字 |
| false | 表示为假的关键字 |
| 变量引用 | super | 该关键字是一个指向父类的引用，用于调用父类的方法或属性。 |
| this | 该关键字是一个指向本类对象的引用，直接指向本类的实例 |
| void | 表示方法没有返回值 |
| 保留字 | goto |  |
| const |  |
| native |  |

## 基本数据类型

在Java中有8种基本数据类型，其中6种是数值类型，另外两种分别是字符类型和布尔类型。而6种数值类型中有4种是整数类型，另外两种是浮点类型

### 整数类型

整数类型用来储存整数数值，即没有小数部分的数值，可以是正数、负数、零。根据所占用内存的大小不同，可以分为byte、short、int、long 4中类型。

|  |  |  |
| --- | --- | --- |
| **数据类型** | **内存空间** | **取值范围** |
| byte | 8位（1个字节） | -128~127 |
| short | 16位（2个字节） | -32768~32767 |
| int | 32位（4个字节） | -2147483648~2147483647 |
| long | 64位（8个字节） | -9223372036854775808~9223372036854775807 |

1. byte型

使用byte关键字来定义byte型变量，可以一次定义多个变量并对其进行赋值，也可以不进行赋值。因为这个类型是整数型中分配内存最小的，所以在使用的时候避免数据溢出（超出取值范围）。

**byte** x = 48, y = -108, z;

1. short型

short型即短整型，使用short关键字来定义short型变量。

1. int型

int型即整型，使用int关键字来定义int型变量。

1. long型

long型即长整型，使用long关键字来定义long型变量。在对long型变量赋值时结尾必须加上“L”，否则将不被认为是long型。

**long** x = 4556824L, y = -4556824L, z;

整数类型在Java程序中有3中表示形式，分别为十进制、八进制、十六进制。

### 浮点类型

浮点类型表示有小数部分的数字。在Java语言中，浮点类型分为单精度浮点类型和双精度浮点类型，它们有不同的取值范围。

|  |  |  |
| --- | --- | --- |
| 数据类型 | 内存空间 | 取值范围 |
| float | 32为（4个字节） | 1.4E-45~3.4028235-E38 |
| double | 64位（8个字节） | 4.9E-324~1.7976931348623157E-308 |

1. float型

float型即单精度浮点型，使用float关键字来定义float型变量，可以一次定义多个变量并对其进行赋值，也可以不进行赋值。

在float型进行赋值时在结尾必须添加“F”，如果不加系统自动将其定义为double型变量。

**float** x = 12.521F, y = -12.521F, z;

1. double型

double型即双精度浮点型，使用double关键字定义double型变量，可以一次定义多个变量并对其进行赋值，也可以不进行赋值。在给double型赋值时，可以使用后缀“D”来表明这个是double类型数据。

### 字符类型

char型即字符类型，使用char关键字进行声明，用于存储单个字符，系统会分配两个字节的内存空间。在定义字符型变量时，要用

单引号括起来。且单引号中只能有一个字符。与”A”不同，”A”是一个包含字符A的字符串。’A’是编码为65所对应的字符常量。

**char** x = 'a';

在Java语言中也可以把字符作为整数对待，只需要在Unicode字符集中找到对应的整数编号。

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int** i = 'd';

**char** c = 97;

System.***out***.println(i); //100

System.***out***.println(c); //a

}

}

在字符类型中有一个特殊的字符，以反斜线“\”开头，后跟一个或多个字符，具有特定的含义，不同于字符原有的意义，叫转义字

符。

|  |  |
| --- | --- |
| 转义字符 | 含义 |
| **\ddd** | 1~3位八进制数据所表示的字符，如\456 |
| **\dxxxx** | 4位十六进制所表示的字符，如\0052 |
| **\’** | 单引号字符 |
| **\\** | 反斜杠字符 |
| **\t** | 垂直制表符，将光标移动到下一个制表符的位置 |
| **\r** | 回车 |
| **\n** | 换行 |
| **\b** | 退格 |
| **\f** | 换页 |

转义字符本身也是字符，所以将转义字符赋值给字符变量时，与其他字符常量值一样需要使用单引号。

### 布尔类型

布尔类型又称逻辑类型，只有true和false两个值，分别代表布尔逻辑中的“真”和“假”。使用boolean关键字声明布尔类型变量，

通常被用在流程控制中作为判断条件。整型值和布尔值之间不能进行相互转换。

**boolean** b = **true**, a = **false**, c;

## 变量与常量

在程序执行过程中，其值不能改变的量成为常量，其值能被改变的量成为变量。变量与常量的声明都必须使用合法的标识符，所有变量与常量只有在声明之后才能使用。

### 声明变量

在程序设计中，变量的使用是一个十分重要的环节。定义一个变量，就是要告诉编译器这个变量属于哪一种数据类型，这样编译器才能知道需要分配多少空间，以及能存放什么样的数据。变量都有一个变量名，变量名必须是合法的标识符，内存空间内的值就是变量值，在声明时可以不给予赋值，也可以直接赋给初值。

**int** age;

**char** c = 'r';

如果需要声明的变量为同一类型，允许一起声明他们（以逗号分隔）：

**int** i, j, k;

**int** i = 1, j = 2, k = 3;

### 命名规则

1. 变量名必须是一个有效的标识符，
2. 变量名不能重复。
3. 选择有意义的单词作为变量名。

### 变量的有效范围

变量的有效范围是指程序代码能够访问该变量的区域，若超出变量所在区域方便变量则编译时会出现错误。在程序中，一般会根据变量能够访问的区域将变量分为“成员变量”和“局部变量”。

1. 成员变量

在类体中定义的变量被称为成员变量，成员变量的整个类中都是有效的。类的成员变量有可分为静态变量和实例变量。

**public** **class** Test {

**int** x = 45; //声明实例变量

**static** **int** *y* = 90; //声明静态变量

}

在定义成员变量的时候在其前面加上static关键字，表示该变量未静态变量，没有加static关键字的都属于实例变量。

静态变量的有效范围可以跨类，甚至可以达到整个应用程序之内。而实例变量只能在定义的类中使用。

跨类使用静态变量可以直接以“类名.静态变量”的方法在其他类中使用。

1. 局部变量

在类的方法体中定义的变量（花括号内）成为局部变量。局部变量只能在当前代码块中有效（花括号内有效）。

局部变量的生命周期取决于方法，当方法被调用时，Java虚拟机为方法中的局部变量分配内存空间，当方法调用结束后，则会释放方法中局部变量占用的内存空间，局部变量也随即销毁。

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int** i = 'd'; //定义的局部变量

System.***out***.println(i);

}

}

### 声明常量

在程序运行过程中一直不会改变的量成为常量，通常也被称为“final变量”。常量在整个程序中只能被赋值一次。在为所有对象共享值时，常量是非常有用的。

在Java语言中声明一个常量，除了要指定数据类型外，还需要通过final关键字子进行限定。常量名通常使用大写字母。

**final** **double** PI = 3.1415926F;

当定义的常量如果属于“成员变量”，则必须在定义时就赋给初值，否则将会在编译的时候出错。如果属于“局部变量”则可以在定义时可以不用初始化，但是赋值后就不能被修改。

## 运算符

运算符是一些特殊的符号，主要英语数学函数、一些类型的赋值语句和逻辑比较方面。

### 赋值运算符

赋值运算符即“=”，是一个二元运算符（即对两个操作数进行处理），其功能是将右边的操作数的值赋值给左边的操作数。

**int** age;

**char** c = 'r';

左边必须是一个变量，右边所赋的值可以是任何数值或表达式，包括变量、常量、表达式。

### 简捷元算符

简捷运算符就是将变量先进行算术运算后将结果赋值给该变量。

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 名称 | 举例 | 等价于 |
| **+=** | 加法赋值元算符 | i+=8 | i=i+8 |
| **-=** | 减法赋值运算符 | i-=8 | i=i-8 |
| **\*=** | 乘法赋值运算符 | i\*=8 | i=i\*8 |
| **/=** | 除法赋值运算符 | i/=8 | i=i/8 |
| **%=** | 求余赋值运算符 | i%=8 | i=i%8 |

### 算术运算符

Java中的算术运算符只要有+（加）、-（减）、\*（乘）、/（除）、%（求余），它们都是二元运算符。

其中“+”和“-”运算符还可以作为数据的正负符号，

除法运算时，0不能作为除数。不然系统会报异常

整数运算符在进行数学运算时，如果运算结果有小数，则会直接舍掉小数部分，返回整数部分。（结果为整数类型）

### 自增和自减运算符

自增、自减元算符是单目运算符，可以放在操作元之前，也可以放在操作元之后。操作元必须是一个整型或浮点型变量。

放在操作元前面的自增、自减运算符，会先将变量的值加1（减1），然后再使该变量参与表达式的运算。

放在操作元后面的自增、自减运算符，会先使变量参与表达式的运算，然后再将该变量加1（减1）。

**int** a = 4;

b = ++a; //a先自加1然后将结果赋值给b

b = a++; //a先将值赋值给b然后再自身加1

**int** a = 5;

**int** b = 10;

//5+11+6\*11

**int** c = (a++) + (++b) + a \* b; //82

System.***out***.println(c);

### 比较运算符

比较运算符属于二元运算符，用于程序中的变量和变量之间、变量与常量之间可以及其他类型的信息之间的比较。比较运算符的运

算结果是boolean型，运算符对应的关系成立时，运算结果是true，反之为false。比较运算符通常用在条件语句中来作为判断的依据。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 运算符 | 作用 | 操作数据 | 举例 | 结果 |
| **>** | 比较左方是否大于右方 | 整型、浮点型、字符型 | ‘a’ > ’b’ | false |
| **<** | 比较左方是否小于右方 | 整型、浮点型、字符型 | 156<456 | false |
| **==** | 比较左方是否等于右方 | 基本数据类型、引用型 | ‘c’ = ‘c’ | true |
| **>=** | 比较左方是否大于等于右方 | 整型、浮点型、字符型 | 479>=426 | true |
| **<=** | 比较左方是否小于等于右方 | 整型、浮点型、字符型 | 12.45<=45.5 | false |
| **!=** | 比较左方是否不等于右方 | 基本数据类型、引用型 | ‘y’ != ‘t’ | true |

其中字符串或者字符比较的是Unicode字符集中对应数值。

### 逻辑运算符

逻辑运算符包括&&（&）（逻辑与）、||（逻辑或）和！（逻辑非），返回值为布尔类型的表达式，操作元也必须是boolean型数据。其中除了“！”是一元运算符之外，其余的都是二元运算符。

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 含义 | 用法 | 结合方法 |
| **&&、&** | 逻辑与 | op1 && op2 | 左到右 |
| **||、|** | 逻辑或 | op1 || op2 | 左到右 |
| **！** | 逻辑非 | ! op | 右到左 |

用逻辑运算符进行逻辑判断时，不同的逻辑运算符与不同的操作元进行操作时，运行结果也不相同。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表达式1 | 表达式2 | 表达式1 && 表达式2 | 表达式1 || 表达式2 | ！表达式1 |
| true | true | true | true | false |
| true | false | false | true | false |
| false | false | false | false | true |
| false | true | false | true | true |

在Java找那个，逻辑运算符“&&”与“&”都表示“逻辑与”，他们的区别在于，“&”会去判断两个表达式；二“&&”则是针对boolean类型进行判断，当第一个表达式为false是则不去判断第二个表达式，直接输出结果。使用“&&”可以节省计算机判断的次数。通常将这种在逻辑表达式中从左端的表达式可推断出整个表达式的值称为“短路”，而那些始终执行逻辑运算符两边的表达式称为“非短路”。“||”和“|”也是如此

### 位运算符

位运算符用于处理整型和字符型的操作数，对其内存进行操作，数据在内存中一二进制的形式表示。

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 含义 | 用法 | 运算分类 |
| **~** | 按位取反 | ~op1 | 按位运算 |
| **&** | 按位与 | Op1 & op2 |
| **|** | 按位或 | Op1 | op2 |
| **^** | 按位异或 | Op1 ^ op2 |
| **<<** | 左移 | Op1 << op2 | 移位运算符 |
| **>>** | 右移 | Op1 >> op2 |
| **>>>** | 无符号右移 | Op1 >>> op2 |

### 三元运算符

三元运算符是Java中唯一一个三目运算符，其操作元有3个，第一个是条件比较，其余的是两个值，当条件表达式返回true，则

运算取得第一个值，返回false则取第二个值。

语法格式：

**i > j ? 100 : 200;**

实例：

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int** i = 46;

**int** j = 97;

**int** z = i > j ? 100 : 200;

System.***out***.println(z); //200

}

}

三元运算符嵌套使用：

**public** **static** **void** main(String[] args) {

/\*\*

\* 接收用户输入三个证书，打印出最大的数（三目运算符）

\* 使用Scanner对象接收用户的输入

\*/

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("请输入第一个整数");

**int** a1 = sc.nextInt();

System.***out***.println("请输入第二个整数");

**int** a2 = sc.nextInt();

System.***out***.println("请输入第三个整数");

**int** a3 = sc.nextInt();

**int** max = (a1 > a2) ? ((a1 > a3) ? a1 : a3) : ((a2 > a3) ? a2 : a3);

System.***out***.println("最大的数为：" + max);

}

### 运算符优先级

Java中的表达式就是使用运算符连接起来的符合Java规则的式子，运算符的优先级决定了表达式中运算执行的先后顺序。

|  |  |  |
| --- | --- | --- |
| 优先级 | 描述 | 运算符 |
| 1 | 括号 | **（）** |
| 2 | 正负号 | **+、—** |
| 3 | 一元运算符 | **++、——** |
| 4 | 乘除 | **\*、/、%** |
| 5 | 加减 | **+、-** |
| 6 | 移位运算 | **>>、>>>、<<** |
| 7 | 比较大小 | **<、>、>=、<=** |
| 8 | 比较是否相等 | **==、!=** |
| 9 | 按位与运算 | **&** |
| 10 | 按位异或运算 | **^** |
| 11 | 按位或运算 | **|** |
| 12 | 逻辑与运算 | **&&** |
| 13 | 逻辑或运算 | **||** |
| 14 | 三元运算符 | **？：** |
| 15 | 赋值运算符 | **=** |

如果两个运算符有相同的优先级，那么左边的表达式要比右边的表达式先被处理。

### 数学函数与常量

在Math类中，包含了各种各样的数学函数。在编写不同类别的程序时，可能需要的函数也不同。

计算一个数值的平方根，使用sqrt方法：

import static java.lang.Math.\*;

double x=4;

double y=Math.sqrt(x);

System.out.println(y); //2.0

在java中没有幂运算，因此需要借助Math类的pow方法。

double y=Math.pow(x, a);

将y的值设置为x的a次幂（x^a）。pow方法有两个double类型的参数，其返回结果也是double类型

## 基本数据类型转换

数据类型转换是将变量从一种类型更改为另一种类型的过程。Java对数据类型的转换有严格的规定，数据从占用存储空间较小的类

型转换为占用存储空间较大的数据类型时，则做自动类型转换（隐式类型转换），反之则必须做强制类型转换（显示类型转换）。

### 自动类型转换

Java中8种基本类型可以进行混合运算，不同类型的数据在运算过程中首先会自动转换为同一类型，再进行运算。数据类型根据占

用存储空间的大小分为高低不同的级别，占有空间小的级别低，占用空间大的级别高，自动类型转换遵循低级到高级转换的规则。

**float** number1 = 45F;

**int** number2 = 152;

System.***out***.println(number1 + number2); //197.0

8中基本数据类型占用存储空间的大小顺序及占用字节数

byte(1) = boolean(1) < short(2) = char(2) < int(4) = float(4) < long(8) = double(8)

隐式类型转换规则

|  |  |  |
| --- | --- | --- |
| 操作数1的数据类型 | 操作数2的数据类型 | 转换后的数据类型 |
| byte、short、char | int | int |
| byte、short、char、int | long | long |
| byte、short、char、int、long | float | float |
| byte、short、char、int、long、float | double | double |

注意：

byte、short在使用的时候JVM会将类型自动转换为int类型

**byte** a = 10;

**byte** b = 5;

**int** c = 10 + 5; //得到的结果为int类型

但是byte、short在进行简捷运算时（+=、-=、\*=、/=、%=）的时候得到的值为原类型（其实得到的值类型也是int，只是虚拟机会自动转为原类型，作了优化。）

**byte** a = 10;

**byte** b = 5;

a += b; //得到的结果为byte类型

**byte** a1 = 2;

**byte** b = 3;

/\*\*

\* 为什么这么写会报错：由于Java编译器会在编译期间或者运行期间将byte和short类型的数据带符号扩展为相应的int类型数据

\* 不能由大转小，所以会报错。

\*/

a1 = a1 + b;

/\*\*

\* 那么此处的写法为什么不报错呢？

\* 实际上虚拟机在底层作了优化处理，对a1 += b进行了强制类型转换，和a1 = (byte)(a1 + b);是等价的，所以不会报错。

\*/

a1 += b;

//对于short，byte，char 比int 字节数小的变量类型来说，运算结果会自动转换为int类型

字符类型在进行数学计算的时候会参照ASCII码表对应的进行转换为int类型的整数进行数学计算。

**char** a = '中';

**char** b = '国';

**int** c = a + b;

System.***out***.println(c); //42282

定义字符类型的变量时，如果值为数值，则会自动转换为ASCII码对应的字符或符号。

**char** a = 65;

System.***out***.println(a); //A

Char类型与整数类型的继续数学计算是，先将char进行ASCII码转换成int类型再进行计算。

### 强制类型转换

在做强制类型转换的时候，如果转换过程中超出数据类型的取值范围会出现数据溢出，造成数据丢失。所以在转换时要加倍小心，

不要超出变量的取值范围否则就得不到想要的结果。

语法格式：

**int** a = (**int**) 45.23; //45

**long** y = (**long**) 456.6F; //456

**int** b = (**int**) 'd'; //100

boolean型的值不能被转换为其他数据类型，其他类型的数据也不能转换为boolean型。

## 代码注释和编码规范

### 代码注释

1. 单行注释

“//”为单行注释标记，从符号“//”开始直到换行为止的所有内容均作为注释而被编译器忽略。

**int** a = (**int**) 45.23; //45

1. 多行注释

“/\*\*/”为多行注释标记，符号“/\*”与“\*/”之间的所有内容均为注释内容。注释中的内容可以换行。

/\*

\* int a = (int) 45.23; //45

\*/

**long** y = (**long**) 456.6F; //456

**int** b = (**int**) 'd'; //100

1. 文档注释

“/\*\*\*/”为文档注释标记。符号“/\*\*”与“\*/”之间的内容均为文档注释内容。当文档注释出现在任何声明之前时，会被Javadoc

文档工具读取作为Javadoc文档内容。文档注释的格式与多行注释的格式相同。

/\*\*

\* 程序名称：

\* 开发时间：

\* 作者：

\*/

编码规范

在开发中养成良好的编码规范，对程序的开发与日后的维护提供很大方便。

1. 每条语句要单独占一行。
2. 每条命令都要以分号结束。
3. 声明变量时要分行声明。
4. Java语句中多个空格看成一个，都与的空格均被视作一个。
5. 不要使用技术性很高、难懂、易混淆判断的语句。
6. 定义关键的方法要多加注释。

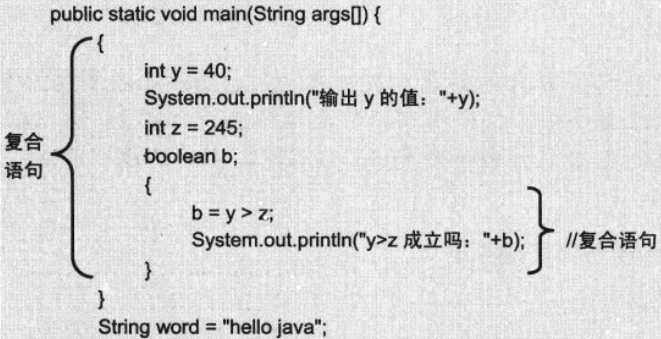
## 流程控制

### 复合语句

Java语言的符合语句是以整个块区为单位的语句，所以又称块语句。符合语句由开括号“{”开始，闭括号“}”结束。对于复合语

句中的每个语句都是从上到下地被执行。复合语句以整个块为单位，可以用在任何一个单独语句可以用到的地方，并且在符合语句

中还可以嵌套符合语句。



在使用复合语句时要注意，复合语句为局部变量创建了一个作用域，该作用域为程序的一部分，在该作用域中某个变量被创建并能

够被使用。如果在某个变量的作用域外使用该变量，则会发生错误。

### 条件语句

#### if条件语句

if条件语句是一个重要的编程语句，它用于告诉程序在某个条件成立的情况下执行某段程序，而在另一种情况下执行另外的语

句。if条件语句可分为简单的if条件语句、if…else语句和if…else if多分支语句。

1. **if语句**

**if**(布尔表达式){

//语句序列

}

其中布尔表达式是必须的参数，表示它最后返回的结果必须是一个布尔值。它可以是一个单纯的布尔变量或常量，或者使

用关系或布尔运算符的表达式。

语句序列是可选蚕食，可以是一条或多条语句，当表达式的值为true时执行这些语句。如果表达式的值为false时不执行

这些语句。

**int** a = 100;

**if**(a == 100){

System.***out***.println(a);

}

1. **if….else语句**

if….else语句是条件语句中最常用的一种形式，它会针对某种条件选择地做出处理。当表达式的结果为true，则执行if后面花括号中的代码，如果表达式的结果为false，则执行else后面花括号中的代码，if后面花括号中的代码不会执行。

**if**(表达式){

若干语句

}**else**{

若干语句

}

if….else if多分支语句

多分支语句一般用于处理有多种情况的事件。

**if**(表达式1){

若干语句

}**else** **if**(表达式2){

若干语句

}**else**{

若干语句

}

#### switch多分支语句

用于处理表达式输出的值与给定的值作匹配，并执行匹配项后面的代码，实现“多选一”的选择。

**public** **class** Test {

**public** **static** **void** main(String[] args) {

System.***out***.println("今天是星期几：");

**int** week = 2;

**switch**(week){

**case** 1:

System.***out***.println("Monday");

**break**;

**case** 2:

System.***out***.println("Tuesday");

**break**;

**case** 3:

System.***out***.println("Wednesday");

**break**;

**default**:

System.***out***.println("Soory,I don't Know");

}

}

}

switch语句中表达式的值必须是整型(long类型除外)或字符型，常量值1~常量值n必须也是整型(long类型除外)或字符型。switch语句首先计算表达式的值，如果表达式的值和某个case后面的变量值相同，则执行该case语句后面的若干个语句直到遇到break语句为止。此时如果该case语句中没有break语句，将继续执行后面case中的若干语句，直到遇到break语句为止。若没有一个变量的值于表达式的值相同，则执行default后面的语句。default语句为可选择的，如果它不存在，而且switch语句中表达式的值不与任何case的常量值相同，switch则不做任何处理。一个switch语句汇总，case的常量值必须互不相同。

### 循环语句

循环语句就是在满足一定条件的情况下反复执行某一个操作。

#### while循环语句

while循环语句也称为条件判断语句，它的循环方式为利用一个条件来控制是否要继续反复执行这个语句。

**int** x = 1;

**int** sum = 0;

**while**(x <= 10){

sum = sum + x;

x++;

}

System.***out***.println("1到10的和是：" + sum);

#### do…while循环语句

do…while循环语句与whlie循环语句类似，它们之间的区别是while循环语句为先判断条件是否成立再执行循环体，而do…while循环语句则先执行一次循环后，再判断条件是否成立。也就是说do…while循环语句中大括号中的程序段至少要被执行一次。

**int** x = 1;

**int** sum = 0;

**do**{

sum = sum + x;

x++;

}**while**(x > 10);

System.***out***.println("1到10的和是：" + sum);

#### for循环语句

1. **for语句**

**for**(表达式1; 表达式2; 表达式3){

语句序列

}

其中表达式1负责完成变量的初始化。表达式2用于指定循环条件，值为boolean型的表达式。表达式3赋值修正变量，改变循环条件。

其中程序的执行顺序为：先执行表达式1，然后再执行表达式2，再执行花括号中的语句，再执行表达式3，然后又开始执行表达式2，执行花括号中的语句，再执行表达式3….循环执行，直到表达式2的结果为false。

**int** sum = 0;

**for** (**int** i = 1; i <= 10; i++) {

sum = sum + i;

}

System.***out***.println("1到10的和是：" + sum);

1. **foreach语句**

foreach语句是for语句的特殊简化版本，foreach语句并不能完全取代for语句，然而任何foreach语句都可以改写为for语句版本。foreach并不是一个关键字，foreach语句在遍历数据等方面为程序员提供了很大的方便。

遍历一维数据：

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int** arr[] = {7, 10, 1};

System.***out***.println("一维数组中的元素分别为：");

**for** (**int** x : arr) {

System.***out***.println(x + "\t");

}

}

}

### 跳转语句

Java语言中提供了3种跳转语句，分别是break语句、continue语句和return语句。

#### break语句

break语句用于终止语句的执行，并且跳出当前循环语句体。一般用于switch语句，同样也可以用于for、while和do…while

循环语句中用于强行退出循环。当代码执行到break关键字的时候，直接退出当前循环体，继续执行循环体后面的其他代码。不会执行当前循环体中break关键字后面的代码。

**public** **void** test(){

**int** i = 0;

**while**(**true**){

**if**(0 == i){

**break**; //跳出循环体，不在执行循环体中的代码

}

//此条语句永远不会执行

System.***out***.println("循环执行");

}

//执行该行代码

System.***out***.println("循环体后面的代码");

}

#### continue语句

continue语句只能应用在for、while、do…while循环语句汇总，用于让程序直接跳过其后面的语句，进行下一次循环。也就是说，当代码执行到continue语句的时候，代码会直接返回到循环语句的开始，进行下一次的循环操作，该关键字后面的代码都不会执行

**public** **void** test(){

**int** i = 0;

**while**(**true**){

**if**(5 > i){

i++;

**continue**; //跳出本次循环，重新执行下一次的循环

}

//当i字增加到5后才会执行这些代码，并跳出循环体

System.***out***.println(i);

System.***out***.println("循环执行");

**break**;

}

//执行该行代码

System.***out***.println("循环体后面的代码");

}

#### retrun语句

retrun关键字用于跳出方法体，结束方法体中代码的执行。return关键字后面的代码都不会执行。如果在return关键字后面有表达式，则在调用该方法的时候会返回表达式对应的值（必须在方法上定义与返回值对应的数据类型）。

语法格式：

**return 表达式;**

其中表达式为可选参数，表示需要返回的值。它的数据类型必须与方法声明中的返回值类型一致，可以通过强制类型转换

实现。

retrun语句通常被放在被调用方法的最后，用于退出当前方法并返回一个值。当把单独的retrun语句放在一个方法的中间时，

会编译错误。但是可以通过把retrun语句用于if语句括起来的方法将retrun语句放在一个方法中，用于实现在程序未执行完方法中的全部语句时退出。

**public** **void** test(){

System.***out***.println("执行方法开始");

**return**; //直接退出方法体，后面的代码不会执行

//此代码不会执行

System.***out***.println("执行方法结束");

}

在代码块中添加return关键字：

**public** **void** test(){

**int** i = 1;

**if**(0 == i){

System.***out***.println("值为0");

}**else** **if**(1 == i){

System.***out***.println("值为1");

**return**; //代码执行到这里就会推出方法体

}

//这些代码都不会执行

System.***out***.println("这会执行吗？");

**return**;

}

## 方法

方法就是为完成某一功能操作而组合在一起的语句组。方便以后遇到同样的问题而不需要重复编写代码。直接调用编写好的方法来处理问题。

### 定义方法

语法格式：

修饰符 返回值类型 方法名(参数列表){

方法体;

}

修饰符：用于修饰方法，设置方法的有效范围。

返回值类型：设置返回值的数据类型，返回值的数据类型一定要 与此处设定的数据类型一致。如果方法不需要返回值，此处省

略数据类型，由void关键字代替。

方法名：此方法的名称，必须是有效的标识符。

参数：定义在方法头中的变量称为形式参数（形参）。参数就像占位符，通过参数可以向方法体中传递值。当调用方法时，就给参数传递一个值，这个值称为实际参数（实参）。参数列表指明方法中参数的类型、顺序和格式。方法中的参数是可选的。可以有参数，也可以没有。

方法体：方法体为语句集合，用时实现某种功能的代码块

实例：

**public** **static** **int** max(**int** num1, **int** num2){

**int** result;

**if**(num1 > num2){ //判断那个值大

result = num1;

}**else**{

result = num2;

}

**return** result; //返回结果

}

### 调用方法

当我们定义好了完成某种功能的方法后，想要使用该方法，则可以通过调用方法的形式来处理相同的问题。

调用方法有两种途径：

1. 如果方法返回一个值，对方法的调用通常就当作一个值来处理

**int** larger = *max*(3, 4); //将返回的结果赋值给变量

1. 如果方法返回void，对方法的调用必须是一条语句

getGrade(78.5); //调用没有返回值的方法

### 参数的值传递

方法的威力在于它处理参数的能力。调用方法时，需要提供实参，它们必须与定义方法签名中所对应的形参顺序相同和类型兼容（不需要经过显示的类型转换）。

**public** **static** **void** nP(String message, **int** n){

**for**(**int** i = 0; i < n; i++){

System.***out***.println(message);

}

}

**public** **static** **void** main(String[] args) {

*nP*("Hello", 3);

}

Java1.5增加了新特性“可变参数”。

**可变参数**

适用于参数个数不确定，类型确定的情况，java把可变参数当做数组处理。

注意：可变参数必须位于最后一项。当可变参数个数多余一个时，必将有一个不是最后一项，所以只支持有一个可变参数。因为参数个数不定，所以当其后边还有相同类型参数时，java无法区分传入的参数属于前一个可变参数还是后边的参数，所以只能让可变参数位于最后一项。

可变参数的特点：

1. 只能出现在参数列表的最后；
2. ...位于变量类型和变量名之间，前后有无空格都可以；
3. 调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中一数组的形式访问可变参数。

**public** **static** **void** test(**int** n, **int**...is){

**for**(**int** i = 0; i < is.length; i++){

System.***out***.println(is[i]);

}

}

**public** **static** **void** main(String[] args) {

*test*(4, 3, 2, 6, 7); //3, 2, 6, 7

}

### 方法的重载

在一个类中有两个方法，他们具有相同的名字，但有不同的参数列表。Java编译器会根据调用方法是指定的参数类型来识别调用哪个方法。这就是方法的重载。

所谓参数列表不同是指：如果参数数量相同，参数类型必须不同；如果参数类型相同，参数数量必须不同；

**public** **class** Test {

**public** **static** **void** nP(String message, **int** n){

**for**(**int** i = 0; i < n; i++){

System.***out***.println(message);

}

}

**public** **static** **void** nP(String message, **double** n){

**for**(**int** i = 0; i < n; i++){

System.***out***.println(message);

}

}

}

## 数据类型

### ****Java数据类型基本概念****

数据类型在计算机语言里面，是对内存位置的一个抽象表达方式，可以理解为针对内存的一种抽象的表达方式。接触每种语言的时候，都会存在数据类型的认识，有复杂的、简单的，各种数据类型都需要在学习初期去了解，Java是强类型语言，所以Java对于数据类型的规范会相对严格。数据类型是语言的抽象原子概念，可以说是语言中最基本的单元定义，在Java里面，本质上讲将数据类型分为两种：基本类型和引用数据类型。

基本类型：基本数据类型是不能简化的、内置的数据类型、由编程语言本身定义，它表示了真实的数字、字符和整数。基本数据类型包括byte、short、char、int、long、float、double、boolean

引用数据类型：Java语言本身不支持C++中的结构（struct）或联合（union）数据类型，它的复合数据类型一般都是通过类或接口进行构造，类提供了捆绑数据和方法的方式，同时可以针对程序外部进行信息隐藏。引用数据类型包括数组、类（对象）、接口

### ****Java数据类型在内存中的存储****

1. 基本数据类型的存储原理

所有的基本数据类型不存在“引用”的概念，基本数据类型都是直接存储在内存中的内存栈上的，数据本身的值就是存储在栈空间里面，而Java语言里面八种基本数据类型是这种存储模型；

1. 引用类型的存储原理

引用类型继承于Object类（也是引用类型）都是按照Java里面存储对象的内存模型来进行数据存储的，使用Java内存堆和内存栈来进行这种类型的数据存储，简单地讲，“引用”是存储在有序的内存栈上的，而对象本身的值存储在内存堆上的；

1. 区别

基本数据类型和引用类型的区别主要在于基本数据类型是分配在栈上的，而引用类型是分配在堆上的（需要java中的栈、堆概念）

1. 举例说明

int a;

a=250; //声明变量a的同时，系统给a分配了空间。

引用类型就不是了，只给变量分配了引用空间，数据空间没有分配，因为谁都不知道数据是什么，整数，字符？我们看一个错误的例子：

MyDate today;

today.day = 4; //发生错误，因为today对象的数据空间未分配。

那我们怎么给它赋值？引用类型变量在声明后必须通过实例化开辟数据空间，才能对变量所指向的对象进行访问。举个例子：

MyDate today; //将变量分配一个保存引用的空间

today = new MyDate(); //这句话是2步，首先执行new MyDate()，给today变量开辟数据空间，然后再执行赋值操作

//引用变量赋值

MyDate a，b; //在内存开辟两个引用空间

a = new MyDate(); //开辟MyDate对象的数据空间，并把该空间的首地址赋给a

b = a; //将a存储空间中的地址写到b的存储空间中

### 引用传递和值传递

1. 值传递

方法调用时，实际参数把它的值传递给对应的形式参数，函数接收的是原始值的一个copy，此时内存中存在两个相等的基本类型，即实际参数和形式参数，后面方法中的操作都是对形参这个值的修改，不影响实际参数的值。

1. 引用传递

也称为传地址。方法调用时，实际参数的引用(地址，而不是参数的值)被传递给方法中相对应的形式参数，函数接收的是原始值的内存地址；在方法执行中，形参和实参内容相同，指向同一块内存地址，方法执行中对引用的操作将会影响到实际对象。



**public** **class** Test {

**public** **void** test1(**int** a){

a = a++;

System.***out***.println(a);

}

**public** **void** test2(MyObj obj){

obj.b = 100;

System.***out***.println(obj.b);

}

**public** **static** **void** main(String[] args) {

Test t = **new** Test();

**int** a = 99;

t.test1(a); //这里传递的参数a就是按值传递

System.***out***.println(a);

MyObj obj = **new** MyObj();

t.test2(obj); //这里传递的参数obj就是引用传递

System.***out***.println(obj.b);

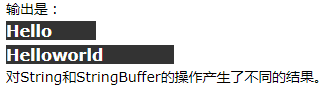
}

}

可以看到，int值没有发生变化，但是在test2方法中对obj类做的修改影响了obj这个对象。

这里要特殊考虑String，以及Integer、Double等几个基本类型包装类，它们都是immutable类型，

因为没有提供自身修改的函数，每次操作都是新生成一个对象，所以要特殊对待，可以认为是和基本数据类型相似，传值操作。



**public** **class** Test {

**public** **void** change(String str){

str = str + "world";

}

**public** **void** change(StringBuffer str){

str.append("world");

}

**public** **static** **void** main(String[] args) {

Test t = **new** Test();

//String类似基本类型，值传递，不会改变实际参数的值

String test1 = "Hello";

t.change(test1);

System.***out***.println(test1);

//StringBuffer和StringBuilder等是引用传递

StringBuffer test2 = **new** StringBuffer("Hello");

t.change(test2);

System.***out***.println(test2.toString());

}

}

### 总结

结合上面的分析，关于值传递和引用传递可以得出这样的结论：

1. 基本数据类型传值，对形参的修改不会影响实参；
2. 引用类型传引用，形参和实参指向同一个内存地址（同一个对象），所以对参数的修改会影响到实际的对象；
3. String, Integer, Double等immutable的类型特殊处理，可以理解为传值，最后的操作不会修改实参对象。

## 数组

数组是具有相同数据类型的一组数据的集合。当需要使用的变量很多，而且数据类型相同时，逐个声明就显得非常麻烦，这时可以声明

一个数组，然后对数组进行操作，从而省去了不少操作。

Java语言中虽然基本数据类型不是对象，但是由基本数据类型组成的数组则是对象。

数组根据维数的不同分为一维数组，二维数组和多维数组。Java中一位数组的每个基本单元都是基本数据类型的数据；二维数组的每个

基本元是一维数组的一维数组；n维数组的每个基本本单元都是n-1维数组的n-1维数组。

### 一维数组

一维数组实质上是一组相同类型数据的集合，当需要在程序中处理一组数据或者传递一组数据时，可以应用这种类型的数组。

#### 创建一维数组

数组作为对象允许使用new关键字进行内存分配。在使用数组之前，必须首先定义数组变量所属的类型，即声明数组。

语法格式：

声明数组

**int** month[];

**int**[] month;

创建数组对象

**int** month[] = **new** **int**[2];

int为数组变量的数据类型，指明数据的数据类型。

month为数组名称，必须是合法的标识符。

[]指明了该变量是为一个数组的引用。

new关键字用于为数组分配内存空间。

12指明数组的元素个数，即数组的长度。

当在创建数组对象的时候，必须指定该数组能够存储的元素个数来确定数组大小。创建数组之后就不能再修改它的大小。可以使用length属性来获得数组的大小。当数组创建后，如果不为其指明元素的值，则会为其赋予默认值。数值型基本数据类的默认值为0，char型的默认值为’\u0000’，boolean型的默认值为false。

数组中的每个元素（变量）可以通过数组的下标来发访问。数组第一个元素的下标值为0，最后一个下标值为arry.length-1。

month[0];

month[2];

#### 初始化一维数组

数组可以与基本数据类型一样进行初始化操作，数组的初始化可分别初始化数组中每个元素。

方法一：

**int**[] arr = **new** **int**[]{1, 2, 3, 4, 5};

方法二：

**int**[] arr = {1, 2, 3, 4, 5};

初始化数组时可以省略new运算符和数组的长度，编译器将根据初始化值的数量来自动计算数组长度，并创建数组。

在内存中数组的元素是以下标为0开始的，也就是数组中的第一个元素在内存中的下标值为0，以此类推。我们可以使用下标

的方式来访问到数组中的各个元素。

**int**[] arr = {1, 2, 3, 4, 5};

System.***out***.println(arr[0]); //1

### 多维数组

二维和多于二维的数组被统称为多维数组。

#### 二维数组

如果一维数组中的各个元素都是一维数组，那么这个数组就是一个二维数组。二维数组常用于表示表。表中的信息以行和列的形式组织，第一个下标代表元素所在的行，第二个下标代表元素的列。

**创建二维数组**

**int**[][] arr = **new** **int**[2][4];

int指定了二维数组的数据类型。

arr为二维数组的数组名称

[][]指明声明的是一个二维数组。

new关键字用于指明分配内存空间

[2]表示二维数组中一维数组的个数为2。

[4]表示为这些一维数组的数组长度为4。

**初始化二维数组**

**int**[][] arr = {{12, 0}, {45, 10}};

初始化二维数组后，要明确数组的下标都是从0开始。所以上面的代码中arr[1][1]的值为10。int型二维数组是以int arr[][]来定义的，所以可以直接给arr[x][y]赋值。

给arr[1]的第二个元素赋值：

arr[1][1] = 20;

**例句**：

输出一个3行4列且所有元素都是0的矩阵。

**public** **static** **void** main(String[] args) {

**int**[][] arr = **new** **int**[3][4]; //创建二维数组

System.***out***.println("输出3行4列的数组：");

**for** (**int** i = 0; i < arr.length; i++) {

**for** (**int** j = 0; j < arr[i].length; j++) { //遍历数组中的每个元素

System.***out***.println(arr[i][j] + ""); //输出元素

}

System.***out***.println(); //输出空格

}

}

对于整型二维数组，创建成功之后系统会赋给数组中每个元素初始化值0

#### 三维数组

三维数组的元素是二维数组，每个元素使用arr[x][y][z]的格式表示。

创建一个三维数组

**int**[][][] arr = **new** **int**[3][2][4];

[3]表示三维数组是由3个二维数组组成。

[2]表示二维数组中由2个一维数组组成。

[4]表示一维数组中由4个元素组成。

初始化三维数组

**int**[][][] arr = {

{{1, 2, 3, 4}, {4, 5, 6, 7}},

{{7, 8, 9, 10}, {11, 12, 13, 14}},

{{15, 16, 17, 18}, {19, 20, 21, 22}}

};

## 数组的基本操作

java.util包的Arrays类包含用来操作数组（如排序和搜索）的各种方法。

### 遍历数组

遍历数组就是获取数组中的每个元素。通常遍历数组都是使用for循环来实现。

#### 遍历一维数组

**public** **static** **void** main(String[] args) {

**int**[] dat = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

**for** (**int** i = 0; i < 12; i++) {

System.***out***.println((i + 1) + "月有" + dat[i] + "天" + "\t");

**if**((i + 1) % 3 == 0){

System.***out***.println("\n");

}

}

}

#### 遍历二维数组

**public** **static** **void** main(String[] args) {

**int**[][] arr = {{1, 5}, {2, 3, 4}, {5, 6, 7}};

System.***out***.println("二维数组是：");

**for** (**int** i = 0; i < arr.length; i++) {

**for** (**int** j = 0; j < arr[i].length; j++) {

System.***out***.println(arr[i][j] + " ");

}

System.***out***.println();

}

}

### 数组的查找

查找是在数组中寻找特定元素的过程

#### 线性查找法

线性查找法将要查找的关键字key与数组中的元素逐个进行比较。这个过程持续到在列表中找到与关键字匹配的元素，或者查完列表也没有找到关键字为止。如果匹配成功，线性查找法返回与关键字匹配的元素在数组中的下标。如果没有匹配成功，则返回-1.

**public** **int** linearSearch(**int**[] list, **int** key){

//遍历数组中的每个字在与key进行比较

**for** (**int** i = 0; i < list.length; i++) {

**if**(key == list[i]){

**return** i;

}

}

**return** -1;

}

线性查找法的关键是将关键字与数组中的每个元素进行比较。

#### 二分查找法

二分查找法是另一种常见的对数组列表的查找方法。使用二分查找法的前提是数组中的元素必须已经排好序且数组中没有值相等的项。

原理：二分查找算法就是不断将数组进行对半分割，每次拿中间元素和需要查找的元素值进行比较，

**public** **int** linearSearch(**int**[] arr, **int** key){

**int** low = 0; //定义最低位的数组下标值

**int** high = arr.length - 1; //定义最高位的数组下标值

**while**(high >= low){

//定义数组居中的下标（此处始终返回的是整数型，如果有小数位，则默认去除小数位）

**int** mid = (low + high) / 2;

//如果需要查询的值大于数组居中的值，则重写定义低位的值

**if**(key > arr[mid]){

low = mid + 1;

//如果需要查询的值小于数组居中的值，则重新定义高位的值

}**else** **if**(key < arr[mid]){

high = mid - 1;

//如果相等，则直接返回居中的值

}**else** **if**(key == arr[mid]){

**return** mid;

//如果没有，则返回-1

}**else**{

**return** -1;

}

}

//如果数组为空，返回-1

**return** -1;

}

### 填充替换数组元素

数组中的元素定义完成后，可以通过Arrays类的静态方法fill()来对数组中的元素进行替换。该方法通过各种重载形式可完成任意类型的数据元素的替换，该方法有两种参数类型。

第一种

语法格式：

Arrays.*fill*(**int** [] arr, **int** value);

arr[]：为需要进行元素替换的数组。

value：要存储到数组中所有元素的值。

实例：

**public** **static** **void** main(String[] args) {

**int**[] arr = {1, 2, 3, 4, 5, 2, 3, 9, 15, 3, 1};

Arrays.fill(arr, 0); //此方法会替换数组中所有的元素值

**for** (**int** i : arr) {

System.out.println(i);

}

}

注意：该方法会将数组中的所有元素值进行替换。

第二种

填充指点范围内的数组元素。

语法格式：

Arrays.*fill*(**int**[] arr, **int** fromIndex, **int** toIndex, **int** value);

arr[]：要进行填充的数组。

fromIndex：指定开始替换元素值的元素索引（会替换包含此下标的值）。

toIndex：指定结束替换元素值的元素索引（不会替换包含此下标的值）。

value：要填充数组元素的值。

实例：

**public** **static** **void** main(String[] args) {

**int**[] arr = {1, 2, 3, 4, 5, 2, 3, 9, 15, 3, 1};

//此方法会替换数组中指定范围内所有的元素值

Arrays.*fill*(arr, 5, arr.length, 0);

**for** (**int** i : arr) {

System.***out***.println(i);

}

}

注意：该方法会根据给定的开始下标（包含）和结束下标（不包含）范围来替换数组对象此范围内的所有值。

### 对数组进行排序

1. **选择排序**

选择排序法就是用数组的每一个值与其后面的所有值进行比较判断。找出数组中最小的值，然后再将最小的值以此放在前面。

**public** **static** **void** sortArray(**int**[] arr){

//循环遍历数组

**for** (**int** i = 0; i < arr.length; i++) {

**int** minNum = arr[i]; //定义一个变量用于存储遍历出的值

**int** minIndex = i; //定义一个变量用于存储下标值

/\*\*

\* 定义第二层循环，其初始值始终是从第一层循环后面的值开始比较

\*/

**for** (**int** j = 0; j < arr.length; j++) {

/\*\*

\* 将变量的值与其后面的所有值进行比较判断

\* 如果变量的值大于后面的值，则将小指赋值给变量，对应的下标也赋值给变量

\* 以此类推，找出其中最小的值和对应的下标赋值给变量

\*/

**if**(minNum > arr[j]){

minNum = arr[j];

minIndex = j;

}

}

/\*\*

\* 当第二层循环遍历完毕后，判断遍历的值是否改变。

\* 如果改变了，则证明第一层循环遍历出来的值不是最小值

\* 需将第一层遍历出的值与最小的值进行互换。让最小值处于第一位

\*/

**if**(minIndex != i){

arr[minIndex] = arr[i];

arr[i] = minNum;

}

}

}

1. 通过**Arrays**类的静态**sort()**方法可以实现对数组排序，sort()方法提供了许多种重载形式，可对任意类型数组进行升序排序。

语法格式：



object：指定需要进行排序的数组

实例：

**public** **static** **void** main(String[] args) {

**int**[] arr = {56,2,32,4,5,2,3,9,15,3,1};

//调用Arrays类的静态方法进行排序

Arrays.*sort*(arr);

**for** (**int** i = 0; i < arr.length; i++) {

System.***out***.println(arr[i]);

}

}

Java语言中的String类型数组的排序算法是根据字典编排顺序排序的，因此数字排在字母前面，大写字母排在小写字母前面。

1. **插入排序法**

插入排序的基本思想是将要排序的值逐个插入到其所合适的位置，其后的元素都向后移一个位置。

**public** **static** **void** sortArray(**int**[] arr){

**int** tmp;

**int** j;

//定义第一层循环体

**for** (**int** i = 0; i < arr.length; i++) {

//将循环的值赋值给tmp变量

tmp = arr[i];

/\*\*

\* 定义第二层循环体，当第一层循环体执行第一次的时候，第二层不会执行

\* 第二次循环开始执行下面的代码

\*/

**for** (j = i - 1; j >= 0 && arr[j] > tmp; j--) {

arr[j + 1] = arr[j]; //交换数组元素

}

arr[j + 1] = tmp; //在排序位置插入数据

}

}

1. **冒泡排序法**

冒泡排序的基本思想是对比相邻的元素值，如果满足条件就交换元素值，把较小的元素移动到数组前面，把大的元素移动到数组后面（交换两个元素的位置），这样数组元素就像气泡一样从底部上升到顶部。

**public** **static** **void** sortArray(**int**[] arr){

**for** (**int** i = 0; i < arr.length - 1; i++) {

**for** (**int** j = 0; j < arr.length - 1; j++) {

**if**(arr[j] < arr[j + 1]){ //判断数组中当前的值与其后一位的值

**int** temp = arr[j]; //定义一个变量用于在交换值是使用

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

冒泡排序法的算法比较简单，排序的结果稳定，但是比较耗时。

### 复制数组

Arrays类的copyOf()方法与copyOfRange()方法可实现对数组的复制。

1. **copyOf()**

该方法提供了很多种重载形式，来满足不同类型数组的复制。

语法格式：

Arrays.copyOf(arr, **int** newlength)

arr：要进行复制的数组。

newlength：int型常量，指赋值后的新数组的长度。如果新数组的长度大于数组arr的长度，则用0填充。（整数以0填充，char型以null填充）。如果复制后的数组长度小于数组arr的长度，则会从数组arr的第一个元素开始截取至满足新数组长度为止。

返回值：复制后的数组

实例：

**public** **static** **void** main(String[] args) {

**int**[] arr = {45,12,2,10,1};

**int**[] newarr = Arrays.*copyOf*(arr, 6);

**for** (**int** i = 0; i < newarr.length; i++) {

System.***out***.println(newarr[i]);

}

}

1. **copyOfRange()**

该方法同样提供了多种重载形式。

语法格式：

Arrays.copyOfRange(arr, **int** formIndex, **int** toIndex)

arr：要进行赋值的数组对象。

formIndex：指定开始赋值数组的索引位置。formIndex必须在0至整个数组长度之间。新数组包括索引是formIndex的元素。

toIndex：要复制范围的最后索引位置。可以大于数组arr的长度。新数组不包括索引是toIndex的元素。多出的部分则用0填充。（整数以0填充，char型以null填充）

返回值：复制指定位置后的数组。

实例：

**public** **static** **void** main(String[] args) {

**int**[] arr = {45,12,2,10,1};

//调用copyOfRange()复制数组

**int**[] newarr = Arrays.*copyOfRange*(arr, 0, 6);

**for** (**int** i = 0; i < newarr.length; i++) {

System.***out***.println(newarr[i]);

}

}

### 判断两个数组是否相等

使用Arrays类的equals()方法检测两个数组是否相等。如果它们的内容相同，那么这两个数组相等。

**int**[] list1 = {2,4,7,10};

**int**[] list2 = {2,4,7,10};

**int**[] list3 = {4,2,7,10};

System.***out***.println(Arrays.*equals*(list1, list2));

System.***out***.println(Arrays.*equals*(list2, list3));

## 字符串

String类即字符串类型，并不是Java的基本数据类型，但可以像基本数据类型一样使用双引号括起来进行声明。

String str = " 1452 12";

在Java中也可以使用String类的构造方法来创建字符串变量。常用的构造方法如下：

### String()

一个String对象，可表示一个空字符序列。

使用String()方法初始化一个新创建的String对象

String str = **new** String();

### String(char a[])

用一个字符数组a创建String对象

使用String(char a[])方法创建String对象



### String(char a[],int offset,int length)

提取字符数组a中的一部分创建一个字符串对象，参数offset表示开始截取字符串的位置，length表示截取字符串的长度。

使用String(char a[],int offset,int length)方法创建String对象



### 字符串声明的区别

1. 使用“=”赋值的方法定义字符串，JVM会在常量池中开辟空间出来将字符串存储进常量池。在常量池中的字符串常量不能重复出现；在定义新的字符串时JVM会先去常量池中查找有没有存在相同的字符串，如果有则新创建的String类型的变量直接指向常量池中的字符串常量；如果没有，则会新开辟个空间将字符串常量存储进常量池中。
2. 使用“new”的方式定义字符串，首先会在堆中开辟一个内存空间。然后再去常量池中查找有没有存在相同的字符串常量，如果有则在常量池中拷贝字符串常量的对象存储在堆中；如果没有则先在常量池中创建字符串常量，然后在拷贝字符串常量对象到堆中。

### 字符串的特性

String类是一个final类型，表明该类是一个不能被继承的类，所以字符串是不可变的。一个字符串对象一旦被创建，其内容是不可变的。

## 字符串类的常用方法

### lenght()

该方法用于返回字符串的字符长度。

**public** **static** **void** main(String[] args) {

String str = "na,sskdn ";

System.***out***.println(str.length()); //9

}

### charAt()

该方法用于返回字符串给定索引值对应的字符

**public** **static** **void** main(String[] args) {

String str = "na,sskdn ";

System.***out***.println(str.charAt(2)); //,

}

### equals()

该方法用于比较两个字符串的值是否相等，返回boolean值。该方法重写了父类object类的equals方法。object类中equals方法比较对象的内存地址。在做判断的时候，该方法严格区分大小写。

**public** **static** **void** main(String[] args) {

String str1 = "na,sskdn ";

String str2 = "na,sskdn";

System.***out***.println(str1.equals(str2)); //false

}

### equalsIgnoreCase()

使用equalsIgnoreCase()方法是在忽略大小写的情况下比较两个字符串是否相等，如果两个字符串具有相同的字符和长度，则返回true，否则返回false。

**public** **static** **void** main(String[] args) {

String str1 = "na,sskdn ";

String str2 = "nA,sskdN ";

System.***out***.println(str1.equalsIgnoreCase(str2)); //true

}

### indexOf()

该方法重载了几种不同的用法，用于返回给定字符/字符串从左向右开始查找第一个匹配的索引值。

indexOf(char ch)

返回指定字符在此字符串中第一次出现处的索引。

indexOf(char ch, int fromIndex)

返回指定字符在此字符串中第一次出现处的索引。从fromIndex给定的索引开始匹配。

indexOf(String str)

返回指定子字符串在此字符串中第一次出现处的索引。

indexOf(String str, int fromIndex)

返回指定子字符串在此字符串中第一次出现处的索引。从fromIndex给定的索引开始匹配。

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.indexOf("an", 2)); //6

}

### lastIndexOf()

该方法用于从右开始向左查询，返回给定字符/字符串在字符串中第一次出现的索引值。

lastIndexOf(char ch)

返回指定字符在此字符串中从右至做查询第一次出现处的索引。

lastIndexOf(char ch, int fromIndex)

返回指定字符在此字符串中从右至做查询第一次出现处的索引。从fromIndex给定的索引开始匹配。

lastIndexOf(String str)

返回指定子字符串在此字符串中从右至做查询第一次出现处的索引。

lastIndexOf(String str, int fromIndex)

返回指定子字符串在此字符串中从右至做查询第一次出现处的索引。从fromIndex给定的索引开始匹配。

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.lastIndexOf("an", 5)); //0

}

### startsWith()

该方法用于判断字符串是否以给定字符串前缀开始。返回boolean值

startsWith(String prefix)

测试此字符串是否以指定prefix的前缀开始。

startsWith(String prefix, int toffset)

测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.startsWith("an", 6)); //true

}

### endsWith()

该方法用于判断字符串是否以给定字符串后缀结束。返回boolean值

endsWith(String suffix)

测试此字符串是否以指定prefix的后缀结束。

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.endsWith("an")); //false

}

### substring()

该方法用于截取字符串后返回一个截取后的新字符串。

substring(int beginIndex)

返回指定索引值开始至结尾的所有字符串。

substring(int beginIndex, int endIndex)

返回指定开始索引值（包含）至结束索引值（不包含）之间的所有字符串

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.substring(2, 6)); //,ssk

}

### replace()

该方法用于将给定的字符替换字符串中指定的字符。返回替换后的新的字符串

replace(char oldChar, char newChar)

用newChar字符替换字符串中所有的oldChar字符。

replace(CharSequence target, CharSequence replacement)

使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。该替换从字符串的开头朝末尾执行，例如，用 "b" 替换字符串 "aaa" 中的 "aa" 将生成 "ba" 而不是 "ab"。

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.replace("aa", "b")); //ba

}

### replaceAll()

该方法用于替换字符串中所有给定正则表达式所匹配的值，并返回替换后的新字符串。

replaceAll(String regex, String replacement)

使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

### replaceFirst()

该方法用于替换字符串中第一个匹配给定正则表达式的值，并返回替换后的新字符串。

replaceFirst(String regex, String replacement)

使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

### trim()

该方法用于去除字符串前后的空格，如果字符串中间有空格则不会去掉。返回去除前后空格的新字符串

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

System.***out***.println(str1.trim()); //an,sskan

}

### concat()

concat(String str)

该方法将字符串给定字符串添加到原字符串的后面，并返回一个拼接后的新字符串

**public** **static** **void** main(String[] args) {

String str1 = "an,sskan ";

String str2 = "123";

System.***out***.println(str1.concat(str2)); //an,sskan 123

}

### split()

该方法用于给定的正则表达式来拆分字符串，返回一个字符串数组。

split(String regex)

根据给定正则表达式的匹配拆分此字符串。

split(String regex, int limit)

根据匹配给定的正则表达式来拆分此字符串。

此方法返回的数组包含此字符串的子字符串，每个子字符串都由另一个匹配给定表达式的子字符串终止，或者由此字符串末尾终止。数组中的子字符串按它们在此字符串中出现的顺序排列。如果表达式不匹配输入的任何部分，那么所得数组只具有一个元素，即此字符串。

limit 参数控制模式应用的次数，因此影响所得数组的长度。如果该限制 n 大于 0，则模式将被最多应用 n - 1 次，数组的长度将不会大于 n，而且数组的最后一项将包含所有超出最后匹配的定界符的输入。如果 n 为非正，那么模式将被应用尽可能多的次数，而且数组可以是任何长度。如果 n 为 0，那么模式将被应用尽可能多的次数，数组可以是任何长度，并且结尾空字符串将被丢弃。

**public** **static** **void** main(String[] args) {

String str1 = "ljansakan ";

String[] arr = str1.split("a", 3); //拆分两次，生成3个数组

System.***out***.println(Arrays.*toString*(arr)); //[lj,ns,kan ]

}

### toLowerCase()

该方法将字符串中的大写字母转换为小写字母，如果原来就是小写字母，则不改变，数字或非字符不受影响。最后返回一个新的字符串，字符串长度与原字符串长度相同。

**public** **static** **void** main(String[] args) {

String s1 = "T am";

s1 = s1.toLowerCase();

System.***out***.println(s1); //"i am"

}

### toUpperCase()

该方法将字符串中的小写字母转换为大写字母，如果原来就是大写字母，则不改变，数字或非字符不受影响。最后返回一个新的字符串，字符串长度与原字符串长度相同。

**public** **static** **void** main(String[] args) {

String s1 = "T am";

s1 = s1.toUpperCase();

System.***out***.println(s1); //"T AM"

}

### toCharArray()

将此字符串转换为一个新的字符数组。返回一个新分配的字符数组，它的长度是此字符串的长度，它的内容被初始化为包含此字符串表示的字符序列。

**public** **static** **void** main(String[] args) {

String s1 = "ljansakan";

**char**[] arr = s1.toCharArray();

System.***out***.println(Arrays.*toString*(arr)); //[l,j,a,n,s,a,k,a,n]

}

### valueOf()

该方法用于将基本类型转换为字符串。该方法按照基本类型重载了此方法。

## 格式化字符串

格式化字符串是程序中经常用到的，主要包括日期格式化。时间格式化、日期/时间组合的格式化和常规类型格式化。

### 格式化方法

在应用程序设计中，经常需要显示时间和日期。format()方法通过给定的特殊转换符作为参数来实现对日期和时间的格式化。format（）方法有以下两种重载形式。

#### format(String format,Object…args)方法

该方法使用指定的格式字符串和参数，格式化后的新字符串使用本地默认的语言环境，返回格式化后的字符串。

语法格式：

str.format(String format, Object...args)

str：需要格式化的字符串对象

format：格式化转换符

args：格式化字符串中格式说明符引用的参数。如果还是格式说明符以外的参数，则忽略这些额外的参数。此参数的

数目是可变的，可以是0

返回值：一个格式化的字符串

#### format(Local l,String format,Object…args)方法

该方法使用指定的格式字符串和参数，格式化后的新字符串使用参数所规定的语言环境，返回格式化后的字符串

语法格式：

str.format(Local 1, String format, Object...args)

str：需要格式化的字符串对象

l：格式化过程中要应用的语言环境。如果l为null，则不进行本地化。

format：格式化转换符

args：格式化字符串中格式说明符引用的参数。如果还有格式说明符以外的参数，则忽略这些额外的参数。此参数的

数目是可变的，可以为0

返回值：一个格式化字符串

### 日期格式化

使用format()方法对日期进行格式化事，会用到日期格式化转换符。常用的日期格式转换符如下：

|  |  |  |
| --- | --- | --- |
| 转换符 | 说明 | 示例 |
| %te | 一个月中的某一天（1~31） | 6 |
| %tb | 指定语言环境的月份简称 | Feb(英文)、二月（中文） |
| %tB | 指定语言环境的月份全称 | February（英文）、二月（中文） |
| %tA | 指定语言环境的星期几全称 | Monday（英文）、星期一（中文） |
| %ta | 指定语言环境的星期几简称 | Mon(英文)、星期一（中文） |
| %tc | 包括全部日期和时间信息 | 星期二 三月 25 13:37:22 CST 2008 |
| %tY | 4位年份 | 2008 |
| %tj | 一年中的第几天（001~366） | 085 |
| %tm | 月份 | 03 |
| %td | 一个月中的第几天（01~31） | 02 |
| %ty | 2位年份 | 08 |

实例：

**public** **static** **void** main(String[] args) {

Date today = **new** Date();

String a = String.*format*(Locale.***US***, "%tb", today);

String b = String.*format*(Locale.***US***, "%tB", today);

String c = String.*format*("%ta", today);

String d = String.*format*("%tA", today);

String e = String.*format*("%tY", today);

String f = String.*format*("%ty", today);

String g = String.*format*("%tm", today);

System.***out***.println("格式化后的字符串为月份的英文缩写：" + a);

System.***out***.println("格式化后的字符串为月份的英文全写：" + b);

System.***out***.println("格式化后的字符串为星期：" + c);

System.***out***.println("格式化后的字符串为星期：" + d);

System.***out***.println("格式化后的字符串为4位的年份值：" + e);

System.***out***.println("格式化后的字符串为2位的年份值：" + f);

System.***out***.println("格式化后的字符串为2位的月份值：" + g);

}

### 时间格式化

使用format()方法对时间进行格式化时，会用到时间格式化转换符。时间格式化转换符要比日期转换符更多、更精确，它可以将时间格式化为时、分、秒、毫秒。

|  |  |  |
| --- | --- | --- |
| 转换符 | 说明 | 示例 |
| %tH | 2位数字的24时制的小时（00~23） | 14 |
| %tI | 2位数字的12时制的小时（01~12） | 05 |
| %tk | 2为数字的24时制的小时（0~23） | 5 |
| %tl | 2位数字的12时制的小时（1~12） | 10 |
| %tM | 2位数字的分钟（00~59） | 05 |
| %tS | 2为数字的秒数（00~60） | 12 |
| %tL | 3位数字的毫秒数（000~999） | 920 |
| %tN | 9位数字的微秒数（000000000~999999999） | 062000000 |
| %tp | 指定语言环境下上午或下午标记 | 下午（中文）、pm（英文） |
| %tz | 相对于GMT RFC 82格式的数字时区偏移量 | +0800 |
| %tZ | 时区缩写形式的字符串 | CST |
| %ts | 1970-01-01 00:00:00至现在经过的秒数 | 1206426646 |
| %tQ | 1970-01-01 00:00:00至现在经过的毫秒数 | 1206426737453 |

实例：

**public** **static** **void** main(String[] args) {

Date date = **new** Date();

System.***out***.println("默认时间格式" + date);

System.***out***.println("两个24小时制的小时" + String.*format*("%tH", date));

}

### 日期和时间组合格式化

因为日期与时间经常是同时出现的，所以格式化转换符还定义了各种日期和时间组合的格式。

|  |  |  |
| --- | --- | --- |
| 转换符 | 说明 | 示例 |
| %tF | “年-月-日”格式（4位年份） | 2008-03-25 |
| %tD | “月/日/年”格式（2位年份） | 03/25/08 |
| %tc | 全部日期和时间信息 | 星期二 三月 25 15:20:00 CST 2008 |
| %tr | “时:分:秒 PM(AM)”格式（12时制） | 03:22:06 下午 |
| %tT | “时:分:秒”格式（24时制） | 15:23:50 |
| %tR | “时:分”格式（24时制） | 15:25 |

### 常规类型格式化

在程序设计过程中，经常需要对常规类型的数据进程格式化，如格式化为整数、格式化为科学记数表示等。

|  |  |  |
| --- | --- | --- |
| 转换符 | 说明 | 示例 |
| %b、%B | 结果被格式化为布尔类型 | true |
| %h、%H | 结果被格式化为散列码 | A05A5198 |
| %s、%S | 结果被格式化为字符串类型 | “abcd” |
| %c、%C | 结果被格式化为字符类型 | ‘a’ |
| %d | 结果被格式化为十进制整数 | 40 |
| %o | 结果被格式化为八进制整数 | 11 |
| %x、%X | 结果被格式化为十六进制整数 | 4b1 |
| %e | 结果被格式化为用计算机科学计数法表示的十进制数 | 1.700000e+01 |
| %a | 结果被个书画为带有效位数和指数的十六进制浮点值 | 0X1.C000000000001P4 |
| %n | 结果为也定于平台的行分割符 |  |
| %% | 结果为字面值‘%’ | % |

## 字符串生成器

字符串生成器即StringBuilder类，是字符串中一个重要的常用类，在程序中会经常用到，另外在程序运行时间上的节省也是很显著的

#### StringBuilder类

StringBuilder类即字符串生成器，新创建的StringBuilder对象初始容量是16个字符，可以自行制定初始长度，也可以动态执行添加、删除和插入等字符串的编辑操作，大大提高了频繁增加字符串的效率。如果附加的字符超过容纳的长度，则StringBuilder对象将自动增加长度以容纳被附加的字符。

声明字符串生成器

StringBuilder builder = **new** StringBuilder();

StringBuilder类是一个可变的字符序列，提供一个与StringBuffer类兼容的API，但是并不保证与之同步。该类是StringBuffer类的一个简易替换，在字符串缓存被单个线程使用时要比StringBuffer类快。但是将StringBuilder的实例用于多个线程是不安全的，这时就要使用StringBuffer类来实现。

#### StringBuilder类的常用方法

1. **StringBuilder append(String str)方法**

该方法将参数str指定的字符串追加到字符串生成器中

语法格式：

builder.append(String str);

builder：声明的字符串生成器

str：追加到字符串生成器中的字符串

返回值：返回字符串生成器的值，String型

实例：

String s = "goods";

StringBuilder builder = **new** StringBuilder();

builder.append(s);

System.***out***.println(builder); //"goods"

append()方法不仅可以实现字符串的追加，还可以实现任何类型的数据追加，如int、boolean、char等，甚至其他的字符串生成器也可以往里追加。

1. **StringBuilder append(StringBuffer sb)方法**

该方法将参数sb指定的字符串缓存StringBuffer的值追加到字符串生成器中。

语法格式：

builder.append(StringBuffer sb);

builder：声明的字符串生成器

sb：缓存中的值，追加到字符串生成器中。

返回值：返回将缓存的值追到字符串生成器后的值，String型。

实例：

**public** **static** **void** main(String[] args) {

StringBuffer buffer = **new** StringBuffer("Sunday"); //创建初始内容为Sunday的字符串缓存

StringBuilder builder = **new** StringBuilder(); //构造字符串生成器

builder.append(buffer); //向字符串生成器中追加字符串缓存

System.***out***.println("追加后的值为：" + builder);

}

1. **StringBuilder insert(int offset,String str)方法**

该方法将参数指定的字符串str添加到offset指定的位置。

语法格式：

builder.insert(**int** offset, String str);

builder：声明的字符串生成器

offset：指定字符串添加的下标索引位置

str：添加的字符串

返回值：返回添加字符串后的字符串生成器的值，String型

实例：

**public** **static** **void** main(String[] args) {

String s1 = "int";

String s2 = "ser";

StringBuilder builder = **new** StringBuilder(s1);

builder.insert(2, s2);

System.***out***.println(builder); //insert

}

1. **StringBuilder delete(int start,String end)方法**

该方法移除字符串生成器中的子字符串，该子字符串从指定的start处开始，到指定的end处结束。

语法格式：

builder.delete(**int** start, String end);

builder：声明的字符串生成器

start：删除子字符串的起始索引位置，（包含此索引上的值）

end：删除子字符串的结尾索引位置，（不包含此索引上的值）

返回值：返回删除子字符串后的字符串生成器的值，String型

实例：

**public** **static** **void** main(String[] args) {

StringBuilder builder = **new** StringBuilder("StringBuilder");

StringBuilder s1 = builder.delete(5, 10);

System.***out***.println(s1); //Strinder

}

1. **String toString()方法**

该方法返回字符串生成器的字符串表示，即将字符串生成器转换为字符串，转换后字符串生成器的值不变。

语法格式：

str.toString();

str：字符串生成器的值，转换字符串。

返回值：返回转换后的字符串。

实例：

**public** **static** **void** main(String[] args) {

StringBuilder builder = **new** StringBuilder("HelloWord");

String s = builder.toString();

System.***out***.println(s); //HelloWord

System.***out***.println(builder); //HelloWord

}

#### StringBuffer类

java.lang.StringBuffer类代表的是可变的字符序列。它是一个容器，可以对字符串内容进行增删。StringBuffer是可变长度的。

StringBuffer类有三个构造方法：

StringBuffer()：初始容量为16的字符串缓冲区。

StringBuffer(int size)：构造指定容量的字符串缓冲区

StringBuffer(String str)：将内容初始化为指定字符串内容。

#### StringBuffer类的常用方法

**length()**

返回长度（字符数）。

**append()**

该方法用于将给定值添加到字符串的后面。该方法重载了多种数据类型的方法，返回的是StringBuffer类型的数据

**append(char[] str)**

将 char 数组参数的字符串表示形式追加到此序列。

**append(char[] str, int offset, int len)**

将 char 数组参数的子数组的字符串表示形式追加到此序列。

将 char 数组 str 中的字符按顺序追加到此序列的内容中，从索引 offset 开始。此字符的长度将增加 len。

**capacity()**

返回当前容量。容量指可用于最新插入的字符的存储量，超过这一容量就需要再次进行分配。

**charAt(int index)**

返回此序列中指定索引处的 char 值。

**delete(int start, int end)**

移除此序列的子字符串中的字符。

**deleteCharAt(int index)**

移除此序列指定位置的 char。

**indexOf(String str)**

返回第一次出现的指定子字符串在该字符串中的索引。

**indexOf(String str, int fromIndex)**

从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。

**lastIndexOf(String str)**

返回最右边出现的指定子字符串在此字符串中的索引。

**lastIndexOf(String str, int fromIndex)**

返回最后一次出现的指定子字符串在此字符串中的索引。

**insert()**

该方法用于将给定值插入到字符串的指定索引位置

**insert(int offset, char[] str)**

将 char 数组参数的字符串表示形式插入此序列中。

**insert(int index, char[] str, int offset, int len)**

将 str 数组参数的子数组的字符串表示形式插入此序列中。其中子数组从指定的 offset 开始，包含 len 个 char。子数组的字符将被插入 index 所指示的位置。此序列的长度将增加 len 个 char。

**reverse()**

将此字符序列用其反转形式取代。

**replace(int start, int end, String str)**

使用给定 String 中的字符替换此序列的子字符串中的字符。

**setCharAt(int index, char ch)**

将给定索引处的字符设置为 ch。

**setLength(int newLength)**

设置字符序列的长度。

**substring(int start)**

返回一个新的 String，它包含此字符序列当前所包含的字符子序列。该子字符串始于指定索引处的字符，一直到此字符串末尾。

**substring(int start, int end)**

返回一个新的 String，它包含此序列当前所包含的字符子序列。该子字符串从指定的 start 处开始，一直到索引 end - 1 处的字符。

**toString()**

返回此序列中数据的字符串表示形式。分配一个新的 String 对象，并将它初始化，以包含当前由此对象表示的字符串序列。然后返回此 String。对此序列的后续更改不影响该 String 的内容。

#### String、StringBuilder、StringBuffer的比较

String类定义的字符串长度无法更改，且字符串值也无法更改。但是StringBuilder和StringBuffer是字符串容器，可以修改字符串的内容及长度。

String、StringBuilder、StringBuffer中执行效率最高的是StringBuilder，但是线程不安全，而StringBuffer则仅次于，是线程安全的，执行效率最低的是String。

## 正则表达式

### 什么是正则表达式

正则表达式定义了字符串的模式。正则表达式可以用来搜索、编辑或处理文本。正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。Java正则表达式和Perl的是最为相似的。

Java 正则表达式和 Perl 的是最为相似的。Java正则表达式的类在 java.util.regex 包中，包括三个类：Pattern、Matcher和PatternSyntaxException。

1. Pattern对象是正则表达式的已编译版本。他没有任何公共构造器，我们通过传递一个正则表达式参数给公共静态方法compile来创建一个pattern对象。
2. Matcher是用来匹配输入字符串和创建的pattern对象的正则引擎对象。这个类没有任何公共构造器，我们用patten对象的matcher方法，使用输入字符串作为参数来获得一个Matcher对象。然后使用matches方法，通过返回的布尔值判断输入字符串是否与正则匹配。
3. 如果正则表达式语法不正确将抛出PatternSyntaxException异常。

**public** **class** RegexExamples {

**public** **static** **void** main(String[] args) {

// using pattern with flags

Pattern pattern = Pattern.*compile*("ab", Pattern.***CASE\_INSENSITIVE***);

Matcher matcher = pattern.matcher("ABcabdAb");

// using Matcher find(), group(), start() and end() methods

**while** (matcher.find()) {

System.***out***.println("Found the text \"" + matcher.group()

+ "\" starting at " + matcher.start()

+ " index and ending at index " + matcher.end());

}

// using Pattern split() method

pattern = Pattern.*compile*("\\W");

String[] words = pattern.split("one@two#three:four$five");

**for** (String s : words) {

System.***out***.println("Split using Pattern.split(): " + s);

}

// using Matcher.replaceFirst() and replaceAll() methods

pattern = Pattern.*compile*("1\*2");

matcher = pattern.matcher("11234512678");

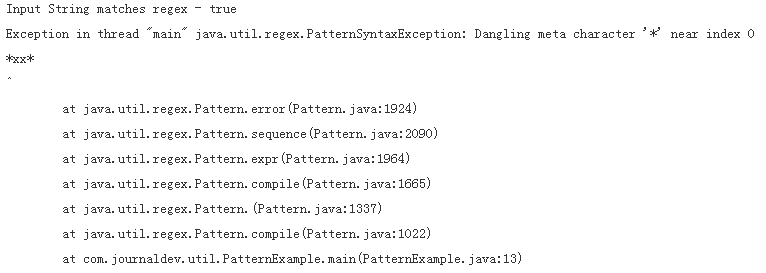
System.***out***.println("Using replaceAll: " + matcher.replaceAll("\_"));

System.***out***.println("Using replaceFirst: " + matcher.replaceFirst("\_"));

}

}

上述程序的输出是：



既然正则表达式总是和字符串有关， Java 1.4对String类进行了扩展，提供了一个matches方法来匹配pattern。在方法内部使用Pattern和Matcher类来处理这些东西，但显然这样减少了代码的行数。

Pattern类同样有matches方法，可以让正则和作为参数输入的字符串匹配，输出布尔值结果。

下述的代码可以将输入字符串和正则表达式进行匹配。

String str = "bbb";

System.***out***.println("Using String matches method: "+str.matches(".bb"));

System.***out***.println("Using Pattern matches method: "+Pattern.*matches*(".bb", str));

所以如果你的需要仅仅是检查输入字符串是否和pattern匹配，你可以通过调用String的matches方法省下时间。只有当你需要操作输入字符串或者重用pattern的时候，你才需要使用Pattern和Matches类。

注意由正则定义的pattern是从左至右应用的，一旦一个原字符在一次匹配中使用过了，将不会再次使用。

例如，正则“121”只会匹配两次字符串“31212142121″，就像这样“\_121\_\_\_\_121″。

### 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：

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (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。

下面的例子说明如何从一个给定的字符串中找到数字串：

**public** **class** RegexMatches

{

**public** **static** **void** main( String args[] ){

// 按指定模式在字符串查找

String line = "This order was placed for QT3000! OK?";

String pattern = "(\\D\*)(\\d+)(.\*)";

// 创建 Pattern 对象

Pattern r = Pattern.*compile*(pattern);

// 现在创建 matcher 对象

Matcher m = r.matcher(line);

**if** (m.find( )) {

System.***out***.println("Found value: " + m.group(0) );

System.***out***.println("Found value: " + m.group(1) );

System.***out***.println("Found value: " + m.group(2) );

System.***out***.println("Found value: " + m.group(3) );

} **else** {

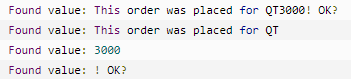
System.***out***.println("NO MATCH");

}

}

}

以上实例编译运行结果如下：



### 正则表达式语法

在其他语言中，\\ 表示：我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。

在 Java 中，\\ 表示：我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。

所以，在其他的语言中，一个反斜杠\就足以具有转义的作用，而在正则表达式中则需要有两个反斜杠才能被解析为其他语言中的转义作用。也可以简单的理解在正则表达式中，两个 \ 代表其他语言中的一个 \，这也就是为什么表示一位数字的正则表达式是 \\d，而表示一个普通的反斜杠是 \\\\。

|  |  |
| --- | --- |
| 字符 | 说明 |
| \ | 将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\\\"匹配"\\"，"\\("匹配"("。 |
| ^ | 匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与"\n"或"\r"之后的位置匹配。 |
| $ | 匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与"\n"或"\r"之前的位置匹配。 |
| \* | 零次或多次匹配前面的字符或子表达式。例如，zo\* 匹配"z"和"zoo"。\* 等效于 {0,}。 |
| + | 一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 {1,}。 |
| ? | 零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。 |
| {n} | n 是非负整数。正好匹配 n 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。 |
| {n,} | n 是非负整数。至少匹配 n 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o\*"。 |
| {n,m} | M 和 n 是非负整数，其中 n <= m。匹配至少 n 次，至多 m 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。 |
| ? | 当此字符紧随任何其他限定符（\*、+、?、{n}、{n,}、{n,m}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。 |
| . | 匹配除"\r\n"之外的任何单个字符。若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。 |
| (pattern) | 匹配 pattern 并捕获该匹配的子表达式。可以使用 $0…$9 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\("或者"\)"。 |
| (?:pattern) | 匹配 pattern 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。 |
| (?=pattern) | 执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 pattern 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。 |
| (?!pattern) | 执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 pattern 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的 "Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。 |
| x|y | 匹配 x 或 y。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。 |
| [xyz] | 字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。 |
| [^xyz] | 反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。 |
| [a-z] | 字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。 |
| [^a-z] | 反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。 |
| \b | 匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。 |
| \B | 非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。 |
| \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 | 换页符匹配。等效于 \x0c 和 \cL。 |
| \n | 换行符匹配。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、换页符等。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 | 制表符匹配。与 \x09 和 \cI 等效。 |
| \v | 垂直制表符匹配。与 \x0b 和 \cK 等效。 |
| \w | 匹配任何字类字符，包括下划线。与"[A-Za-z0-9\_]"等效。 |
| \W | 与任何非单词字符匹配。与"[^A-Za-z0-9\_]"等效。 |
| \xn | 匹配 n，此处的 n 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。 |
| \num | 匹配 num，此处的 num 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。 |
| \n | 标识一个八进制转义码或反向引用。如果 \n 前面至少有 n 个捕获子表达式，那么 n 是反向引用。否则，如果 n 是八进制数 (0-7)，那么 n 是八进制转义码。 |
| \nm | 标识一个八进制转义码或反向引用。如果 \nm 前面至少有 nm 个捕获子表达式，那么 nm 是反向引用。如果 \nm 前面至少有 n 个捕获，则 n 是反向引用，后面跟有字符 m。如果两种前面的情况都不存在，则 \nm 匹配八进制值 nm，其中 n 和 m 是八进制数字 (0-7)。 |
| \nml | 当 n 是八进制数 (0-3)，m 和 l 是八进制数 (0-7) 时，匹配八进制转义码 nml。 |
| \un | 匹配 n，其中 n 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。 |

### Matcher 类的方法

**索引方法**

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

|  |  |
| --- | --- |
| 序号 | 方法及说明 |
| 1 | public int start()  返回以前匹配的初始索引。 |
| 2 | public int start(int group)  返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引 |
| 3 | public int end()  返回最后匹配字符之后的偏移量。 |
| 4 | public int end(int group)  返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。 |

**研究方法**

研究方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

|  |  |
| --- | --- |
| 序号 | 方法及说明 |
| 1 | public boolean lookingAt()  尝试将从区域开头开始的输入序列与该模式匹配。 |
| 2 | public boolean find()  尝试查找与该模式匹配的输入序列的下一个子序列。 |
| 3 | public boolean find(int start）  重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。 |
| 4 | public boolean matches()  尝试将整个区域与模式匹配。 |

**替换方法**

替换方法是替换输入字符串里文本的方法：

|  |  |
| --- | --- |
| 序号 | 方法及说明 |
| 1 | public Matcher appendReplacement(StringBuffer sb, String replacement)  实现非终端添加和替换步骤。 |
| 2 | public StringBuffer appendTail(StringBuffer sb)  实现终端添加和替换步骤。 |
| 3 | public String replaceAll(String replacement)  替换模式与给定替换字符串相匹配的输入序列的每个子序列。 |
| 4 | public String replaceFirst(String replacement)  替换模式与给定替换字符串匹配的输入序列的第一个子序列。 |
| 5 | public static String quoteReplacement(String s)  返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。 |

**start 和 end 方法**

下面是一个对单词 "cat" 出现在输入字符串中出现次数进行计数的例子：

**public** **class** RegexMatches{

**private** **static** **final** String ***REGEX*** = "\\bcat\\b";

**private** **static** **final** String ***INPUT*** = "cat cat cat cattie cat";

**public** **static** **void** main( String args[] ){

Pattern p = Pattern.*compile*(***REGEX***);

Matcher m = p.matcher(***INPUT***); // 获取 matcher 对象

**int** count = 0;

**while**(m.find()) {

count++;

System.***out***.println("Match number "+count);

System.***out***.println("start(): "+m.start());

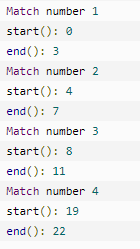
System.***out***.println("end(): "+m.end());

}

}

}

以上实例编译运行结果如下：



可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start 方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end 方法最后一个匹配字符的索引加 1。

**matches 和 lookingAt 方法**

matches 和 lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是 matches 要求整个序列都匹配，而lookingAt 不要求。

lookingAt 方法虽然不需要整句都匹配，但是需要从第一个字符开始匹配。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

**public** **class** RegexMatches{

**private** **static** **final** String ***REGEX*** = "foo";

**private** **static** **final** String ***INPUT*** = "fooooooooooooooooo";

**private** **static** **final** String ***INPUT2*** = "ooooofoooooooooooo";

**private** **static** Pattern *pattern*;

**private** **static** Matcher *matcher*;

**private** **static** Matcher *matcher2*;

**public** **static** **void** main( String args[] ){

*pattern* = Pattern.*compile*(***REGEX***);

*matcher* = *pattern*.matcher(***INPUT***);

*matcher2* = *pattern*.matcher(***INPUT2***);

System.***out***.println("Current REGEX is: "+***REGEX***);

System.***out***.println("Current INPUT is: "+***INPUT***);

System.***out***.println("Current INPUT2 is: "+***INPUT2***);

System.***out***.println("lookingAt(): "+*matcher*.lookingAt());

System.***out***.println("matches(): "+*matcher*.matches());

System.***out***.println("lookingAt(): "+*matcher2*.lookingAt());

}

}

以上实例编译运行结果如下：



**replaceFirst 和 replaceAll 方法**

replaceFirst 和 replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

**public** **class** RegexMatches{

**private** **static** String *REGEX* = "dog";

**private** **static** String *INPUT* = "The dog says meow. " +

"All dogs say meow.";

**private** **static** String *REPLACE* = "cat";

**public** **static** **void** main(String[] args) {

Pattern p = Pattern.*compile*(*REGEX*);

// get a matcher object

Matcher m = p.matcher(*INPUT*);

*INPUT* = m.replaceAll(*REPLACE*);

System.***out***.println(*INPUT*);

}

}

以上实例编译运行结果如下：



**appendReplacement 和 appendTail 方法**

Matcher 类也提供了appendReplacement 和 appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

**public** **class** RegexMatches{

**private** **static** String *REGEX* = "a\*b";

**private** **static** String *INPUT* = "aabfooaabfooabfoob";

**private** **static** String *REPLACE* = "-";

**public** **static** **void** main(String[] args) {

Pattern p = Pattern.compile(*REGEX*);

// 获取 matcher 对象

Matcher m = p.matcher(*INPUT*);

StringBuffer sb = **new** StringBuffer();

**while**(m.find()){

m.appendReplacement(sb,*REPLACE*);

}

m.appendTail(sb);

System.***out***.println(sb.toString());

}

}

以上实例编译运行结果如下：



### PatternSyntaxException 类的方法

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

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

|  |  |
| --- | --- |
| 序号 | 方法及说明 |
| 1 | public String getDescription()  获取错误的描述。 |
| 2 | public int getIndex()  获取错误的索引。 |
| 3 | public String getPattern()  获取错误的正则表达式模式。 |
| 4 | public String getMessage()  返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。 |

## java代码块

所谓代码块就是用大括号({})将多行代码封装在一起，形成一个独立的数据体，用于实现特定的算法。一般来说代码块是不能单独运行的，它必须要有运行主体。在Java中代码块主要分为四种

### 普通代码块

普通代码块是我们用得最多的也是最普遍的，它就是在方法名后面用{}括起来的代码段。普通代码块是不能够单独存在的，它必须要紧跟在方法名后面。同时也必须要使用方法名调用它。

**public** **void** test(){

System.***out***.println("普通代码块");

}

### 静态代码块

想到静态我们就会想到static，静态代码块就是用static修饰的用{}括起来的代码段，它的主要目的就是对静态属性进行初始化。

**public** **class** Test {

**static**{

System.***out***.println("静态代码块");

}

}

### 同步代码块

使用 synchronized 关键字修饰，并使用“{}”括起来的代码片段，它表示同一时间只能有一个线程进入到该方法块中，是一种多线程保护机制。

**public** **void** test(){

**synchronized**(**this**){

//同步代码块

}

}

### 构造代码块

在类中直接定义没有任何修饰符、前缀、后缀的代码块即为构造代码块。我们明白一个类必须至少有一个构造函数，构造函数在生成对象时被调用。构造代码块和构造函数一样同样是在生成一个对象时被调用。

那么构造代码在什么时候被调用？如何调用的呢？看如下代码：

**public** **class** Test {

//构造代码块

{

System.***out***.println("构造代码块");

}

//无参构造函数

**public** Test(){

System.***out***.println("无参构造函数");

}

//有参构造函数

**public** Test(**int** n){

System.***out***.println("有参构造函数");

}

/\*\*

\* 运行结果：

\* 1、构造代码块

\* 2、无参构造函数

\* 3、构造代码块

\* 4、有参构造函数

\*/

**public** **static** **void** main(String[] args) {

**new** Test();

**new** Test(1);

}

}

上面定义了一个非常简单的类，该类包含无参构造函数、有参构造函数以及构造代码块，同时在上面也提过代码块是没有独立运行的能力，他必须要有一个可以承载的载体，那么编译器会如何来处理构造代码块呢？编译器会将代码块按照他们的顺序(假如有多个代码块)插入到所有的构造函数的最前端，这样就能保证不管调用哪个构造函数都会执行所有的构造代码块。

有一点需要注意构造代码不是在构造函数之前运行的，它是依托构造函数执行的。正是由于构造代码块有这几个特性，所以它常用于如下场景：

1. 初始化实例变量

如果一个类中存在若干个构造函数，这些构造函数都需要对实例变量进行初始化，如果我们直接在构造函数中实例化，必定会产生很多重复代码，繁琐和可读性差。这里我们可以充分利用构造代码块来实现。这是利用编译器会将构造代码块添加到每个构造函数中的特性。

1. 初始化实例环境

一个对象必须在适当的场景下才能存在，如果没有适当的场景，则就需要在创建对象时创建此场景。我们可以利用构造代码块来创建此场景，尤其是该场景的创建过程较为复杂。构造代码会在构造函数之前执行。

上面两个常用场景都充分利用构造代码块的特性，能够很好的解决在实例化对象时构造函数比较难解决的问题，利用构造代码不仅可以减少代码量，同时也是程序的可读性增强了。特别是当一个对象的创建过程比较复杂，需要实现一些复杂逻辑，这个时候如果在构造函数中实现逻辑，这是不推荐的，因为我们提倡构造函数要尽可能的简单易懂，所以我们可以使用构造代码封装这些逻辑实现部分。

# Java语言进阶

## Java中内存的分配

Java内存分配主要包括以下几个区域

1. 寄存器：我们在程序中无法操控
2. 栈：存放基本类型的数据和对象的引用，但对象本身不存放在栈中，而是存放在堆中
3. 堆：存放用new关键字产生的数据
4. 静态域：存放在对象中用static定义的静态成员
5. 常量池：存放常量
6. 非RAM（随机存取存储器）存储：硬盘等永久存储空间

## 类和对象

面向对象程序设计（简称OOP）就是使用对象进行程序设计。对象（Object）代表显示世界中可以明确标识的一个实体。例如：一个学生、一个圆都可以看作是一个对象。每个对象都有自己独特的标识、状态和行为。

1. 一个对象的状态（state，也称之为特征或属性）是指那些具有他们当前值得数据域。例如：苑对象具有一个数据域radius，它是标识圆的属性。一个矩形对象具有数据域width和height，他们都是标识矩形的属性。
2. 一个对象的行为（behavior，也称之为动作）是有方法定义的。调用对象的一个方法就是要求对象完成一个动作。

### 面向对象的特点

1. **封装**

封装是面向对象编程的核心思想，将对象的属性和行为封装起来，而将对象的属性和行为封装起来的载体就是类，类通常对客户隐藏其实现细节，这就是封装的思想。采用封装的思想保证了类内部数据结构的完整性，应用该类的用户不能轻易直接操纵此数据结构，而只能执行类允许公开的数据。这样避免了外部对内部数据的影响，提高程序的可维护性。

1. **继承**

类与类之间同样具有关系，被称为关联。关联是描述两个类之间的一般二元关系。两个类之间的关联有很多种，继承就是关联中的一种。当处理一个问题时，可以将一些有用的类保留下来，当遇到同样的问题时拿来复用。软件的代码使用继承思想可以缩短软件开发的事件，复用那些已经定义好的类可以提高系统性能，减少系统在使用过程中出现错误的几率。继承性主要利用特定对象之间的共有属性。

1. **多态**

将父类对象应用于子类的特征就是多态。多态性允许以统一的风格编写程序，以处理种类繁多的已存在的类以及相关类。该统一风格可以由父类来实现，根据父类统一风格的处理，就可以实例化子类的对象。由于整个事件的处理都只依赖于父类的方法，所以日后只要维护和调整父类的方法即可，这样既降低了维护的难度，又节省了时间。

在提到多态的同时，不得不提到抽象类和接口，因为多态的实现并不依赖具体类，而是依赖于抽象类和接口。

### 类

类是一个模板、蓝本或者说是合约，用来定义对象的数据域是什么以及方法使做什么的。一个对象是类的一个实例。可以从一个类中创建多个实例。创建实例的过程称为实例化。类和对象之间的关系似于苹果派配方和苹果派之间的额关系，可以用一种配方做出任意多的苹果派。

#### 定义类

类的定义需要使用关键字class加上类名，类名首字母需大写。

**class** Test {

}

当我们在class关键字前面加了public修饰符后，则该类定义为一个公共类，其他的类就可以访问该类。且该文件的文件名必须和公共类的类名保持一致。一个java文件中允许有多个类，但是只允许只有一个公共类型。



**public** **class** Test {

}

#### 类的成员变量

在Java中对象的属性称为成员变量，也可以称为属性。成员变量的类型可以设置为Java语言中合法的数据类型，其实成员变量就是普通的变量，可以为它设置初始值，也可以不为其设置。成员变量定义在类中，类的属性也可以是对象。例如：如下代码中的String name中String也是一个类，name为String类型的对象。

**public** **class** Test {

**int** id; //成员变量

String name; //成员变量

}

#### 类的成员方法

在Java语言中使用成员方法对应于类对象的行为。

语法格式：

权限修饰符 返回值类型 方法名(参数类型 参数名){

...//方法体

**return** 返回值;

}

一个成员方法可以有参数，这个参数可以是对象也可以是基本数据类型的变量，同时成员方法有返回值和不返回任何值的选择，方法需要返回结果，需要在方法体中使用return关键字，使用这个关键字后，方法的执行就将终止。不需要返回值可以使用void关键字表示。

**public** **class** Test {

**int** id; //成员变量

String name; //成员变量

**private** **void** setName(String name){ //成员方法

**this**.name = name;

}

}

#### 类方法的重载

在同一个类中，我们可以对类中的方法（成员方法和构造方法）进行重载。所谓重载指在一个类中定义多个同名的方法，但要求每个方法具有不同的参数的类型或参数的个数或参数的顺序。调用重载方法时，Java编译器能通过检查调用的方法的参数类型和个数及顺序选择一个恰当的方法。方法重载通常用于创建完成一组任务相似但参数的类型或参数的个数不同的方法。

**public** **class** Test {

**public** **void** receive(**int** i){

System.***out***.println("Receivd onr int data");

System.***out***.println("i=" + i);

}

**public** **void** receive(**float** f){

System.***out***.println("Receivd onr int data");

System.***out***.println("f=" + f);

}

**public** **void** receive(String s){

System.***out***.println("Receivd onr int data");

System.***out***.println("s=" + s);

**public** **static** **void** main(String[] args) {

Test test = **new** Test();

test.receive(3456);

test.receive(34.56f);

test.receive("方法重载");

}

}

}

**具体规范**

一.方法名一定要相同。

二.方法的参数列表必须不同，包括参数的类型或个数或顺序，以此区分不同的方法体。

1.如果参数个数不同，就不管它的参数类型了。

2.如果参数个数相同，那么参数的类型必须不同。

3.如果参数个数和类型相同，那么参数的顺序必须不同。

三.方法的返回类型、修饰符可以相同，也可不同。

#### 类的局部变量

如果在成员方法内部定义一个变量，那么这个变量被称为局部变量。

局部变量是在方法被执行时创建，在方法执行结束时被销毁。局部变量在使用时必须进行赋值操作或被初始化，否则会出现编译错误。

**public** String getName(){

**int** id = 0; //局部变量

setName("java"); //调用其他方法

**return** id + **this**.name; //设置返回值

}

局部变量的有效范围：

可以将局部变量的有效范围称为变量的作用域，局部变量的有效范围从该变量的声明开始到该变量的结束为止。

在相互不嵌套的作用域中可以同时声明两个名称、类型相同的局部变量。（因为局部变量的作用域只是在定义它的方法体内部，只要出了方法体改局部变量就会被销毁）

#### 类的构造方法

在类中除了成员方法之外，还存在一种特殊的方法，那就是构造方法。构造方法是一个与类同名的方法，对象的创建就是通过构造方法完成的，每当类实例化一个对象时，类都会自动调用构造方法

构造方法的特点：

1. 构造方法没有返回值
2. 构造方法的名称要与本类的名称相同
3. 构造方法使在创建一个对象是使用new操作符时调用的。构造方法的作用是初始化对象。

在定义构造方法时，构造方法没有返回值，但这与普通没有返回值的方法不同，普通没有返回值的方法使用public void methodEx()这种形式进行定义，但构造方法并不需要void关键字进行修饰。

语法格式：

**public** **class** Test {

**public** Test(){

//...构造方法体

}

}

public：构造方法修饰符。

Test：构造方法的名称，要与类名一致。

在构造方法中可以为成员变量赋值，这样当实例化一个本类的对象时，相应的成员变量也将被初始化。（方法的重载）

注意：如果在类中定义的构造方法都是带参数的构造方法，则编译器不会为类设置一个默认的不带参数的构造方法，当试图调用不带参数的构造方法实例化一个对象时，编译器会报错。所以只有在类中没有定义任何构造方法时，编译器才会在该类中自动创建一个不带参数的构造方法。所以我们在定义了有参数的构造方法的时候必须显示的定义无参的构造方法。

#### 主方法

主方法是程序的入口点，它定义了程序从何处开始；主方法提供对程序流向的控制。Java编译器通过主方法来执行程序。主方法只能依托在类中，一个类中只能有一个主方法。（在一个完整的程序中也只能有一个主方法，它是整个程序的入口。JVM在执行程序时会从主方法开始执行）

语法格式：

**public** **static** **void** main(String[] args) {

//主方法

}

特点：

1. 主方法是静态的。要直接在主方法中调用其他方法，则该方法必须也是静态的。
2. 主方法没有返回值。
3. 主方法的形参为数组。

**public** **class** Test { //定义一个类

**public** **static** **void** main(String[] args) {

//主方法

}

}

#### 静态变量、常量、方法和静态代码块

在Java中有static关键字修饰的变量、常量和方法被称为静态变量、常量和方法。实现在不同的类和对象中共享一个数据（因为成员变量、常量和方法的作用域只能是类结构体中）。我们在调用静态变量、常量和方法的时候，不需要new一个对象出来，直接通过“类名.变量”的方式就可以使用。被static关键字修饰的变量、常量和方法会在加载类的时候优先被初始化，而且只能初始化一次。

类名.静态变量 //其中类名为承载该变量的类名称

类名.静态常量 //其中类名为承载该常量的类名称

类名.静态方法 //其中类名为承载该方法的类名称

实例：

**public** **class** Test { //定义一个类

**static** **double** *pi* = 3.1415;

**static** **int** *id*;

**public** **static** **void** mothod1(){

//方法体

}

**public** **void** method2(){

System.***out***.println(Test.*pi*);

System.***out***.println(Test.*id*);

Test.*mothod1*();

}

}

在Java中静态成员同样遵循这public、private、protected修饰符的约束。并且静态成员被所有类和对象所共享。

**静态方法**

static方法一般称作静态方法，由于静态方法不依赖于任何对象就可以进行访问，因此对于静态方法来说，是没有this的，因为它不依附于任何对象，既然都没有对象，就谈不上this了。并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。

**public** **class** Test { //定义一个类

**static** **double** *pi* = 3.1415;

**static** **int** *id*;

**public** **static** **void** mothod1(){

//方法体

}

**public** **void** method2(){

Test.*mothod1*();

**this**.*mothod1*(); //应该以静态方式访问类型 Test 中的静态方法 mothod1()

}

}

对静态方法有以下两点规定：

1. 在静态方法中不可以使用this关键字。
2. 在静态方法中不可以直接调用非静态方法，但是在非静态方法中可以调用静态方法。

**public** **class** Test { //定义一个类

**private** **static** String *str1* = "staticProperty";

**private** String str2 = "property";

**public** Test(){

}

**public** **void** print1(){

System.***out***.println(*str1*);

System.***out***.println(str2);

*print2*();

}

**public** **static** **void** print2(){

System.***out***.println(*str1*);

System.***out***.println(str2); //不能对类型 Test 中的非静态方法 print1（）进行静态引用

print1(); //不能对类型 Test 中的非静态方法 print1（）进行静态引用

}

}

在上面的代码中，由于print2方法是独立于对象存在的，可以直接用过类名调用。假如说可以在静态方法中访问非静态方法/变量的话，那么如果在main方法中有下面一条语句：MyObject.print2();此时对象都没有，str2根本就不存在，所以就会产生矛盾了。同样对于方法也是一样，由于你无法预知在print1方法中是否访问了非静态成员变量，所以也禁止在静态成员方法中访问非静态成员方法。而对于非静态成员方法，它访问静态成员方法/变量显然是毫无限制的。因此，如果说想在不创建对象的情况下调用某个方法，就可以将这个方法设置为static。我们最常见的static方法就是main方法，至于为什么main方法必须是static的，现在就很清楚了。因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。另外记住，即使没有显示地声明为static，类的构造器实际上也是静态方法。

**静态变量**

static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。static成员变量的初始化顺序按照定义的顺序进行初始化。

注意：在Java语言中规定不能将方法体内的局部变量声明为static。

**静态代码块**

static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。

**public** **class** Test {

**private** Date birthDate;

**public** Test(Date birthDate){

**this**.birthDate = birthDate;

}

**boolean** isBornBoomer(){

Date startDate = Date.valueOf("1946");

Date endDate = Date.valueOf("1964");

**return** birthDate.compareTo(startDate) >= 0 && birthDate.compareTo(endDate) < 0;

}

}

isBornBoomer是用来这个人是否是1946-1964年出生的，而每次isBornBoomer被调用的时候，都会生成startDate和endDate两个对象，造成了空间浪费，如果改成这样效率会更好：

**public** **class** Test {

**private** Date birthDate;

**private** **static** Date *startDate*, *endDate*;

**static**{

*startDate* = Date.valueOf("1946");

*endDate* = Date.valueOf("1964");

}

**public** Test(Date birthDate){

**this**.birthDate = birthDate;

}

**boolean** isBornBoomer(){

**return** birthDate.compareTo(*startDate*) >= 0 && birthDate.compareTo(*endDate*) < 0;

}

}

因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

#### 权限修饰符

Java语言中的权限修饰符主要包括private、public、default和protected，这些修饰符控制着对类和类的成员变量以及成员方法的访问。如果一个类的成员变量或成员方法被修饰符private，则该成员变量只能被本类中被使用，在子类中是不可见的，并且对其他包的类也是不可见的。

如果将类的成员变量和成员方法的访问权限设置为public，则除了可以在本类使用这些数据之外，还可以在子类和其他包中的类中使用。

如果一个类的访问权限被设置为private，这个类将隐藏其内的所有数据，以免用户直接访问它，只能在同一个类中访问。

如果一个类的访问权限被设置为public，这个类中的数据可以被子类或其他包中的类使用。

如果一个类使用protected修饰符，那么只有本包内的其他类和子类可以访问此类中的成员变量和成员方法。

如果一个类不使用任何的修饰符设置权限或default，则这个类预设为包存取范围，即只有本包中的类可以调用这个类的成员变量或成员方法，但是子类不能访问。



注意：Java语言规定，类的权限设定会约束类成员上的权限设定，类中的成员变量和成员方法的访问权限受限与类的访问权限。

等同于

**public** **class** Test {

**void** doString(){

//....方法体

}

}

**public** **class** Test {

**public** **void** doString(){

//....方法体

}

}

我们可以使用权限修饰符来实现类的封装

//类对数据和实现细节的疯涨，外部只能通过公开的方法来获取和设置属性值

**public** **class** Test {

**private** String age;

**private** String name; //设置私有的属性

//定义修改age属性值的方法供外部调用

**public** **void** setAge(String age){

**this**.age = age;

}

//定义访问age属性值的方法供外部调用

**public** String getAge(){

**return** **this**.age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

#### this关键字

在Java语言中规定使用this关键来调用类的成员变量和成员方法，即this引用就是对一个对象的引用。

**public** **class** Test {

String name;

**private** **void** setName(String name){

**this**.name = name; //相当于Test.name = name

}

}

this除了可以调用类的成员变量或成员方法之外，还可以作为方法的返回值，返回对象的引用。

**public** **class** Test {

**public** Test getTest(){

**return** **this**; //返回对象

}

}

在类中this还可以在构造方法中调用类的另一个构造方法。

**public** **class** Test {

**public** Test(){

**this**("姓名"); //通过无参构造方法调用有参构造方法

}

**public** Test(String name){

}

}

#### 类的设计原则

1. 内聚性

类应该描述一个单一的实体，而所有的类操作应该在逻辑上相互配合，支持一个连贯的目标。例如：可以专门为学生使用一个类，但不应该将学生与教职工组合在同一个类中，因为学生和教职工是不同的实体。

1. 一致性

遵循标准Java程序设计风格和命名习惯。给类、数据域和方法选择有信息量的名字。流行的风格是将数据声明置于构造方法之前，并且将构造方法置于方法之前。

1. 封装性

一个类应该使用private修饰符隐藏其数据，以免用户直接访问它。这使得类更易于维护。如果想让数据域刻度，只需要提供get方法。如果想让数据域可更新，应该提供set方法。

1. 清晰性

为使设计清晰，内聚性、一致性和封装性都是很好的设计原则。除此之外，类应该有一个很清晰的合约，易于解释和理解。用户可以以多种不同组合和顺序，在许多不同环境中结合多个类。因此，应该设计一个类，这个类应该没有对用户使用目的及使用时间的限制，设计属性以容许用户按值的任何顺序和任何组合来设置，所设计的方法实现的功能应该与它们出现的顺序无关。

1. 完整性

类经常是为许多不同用户的使用而设计的。为了能在一个广泛的应用中使用，一个类应该通过属性和方法提供多种方案以适应用户的不同需求。

1. 实例和静态

依赖于类的具体实例的变量或方法应该是一个实例变量或方法。如果一个变量被类的所有实例所共享，那就应该将它声明为静态的。

### JavaBean

### 对象

Java是一门面向对象的程序设计语言，对象是由类抽象出来的，所有的问题都是通过对象来处理，对象可以操作类的属性和方法解

决相应的问题。

#### 对象的创建

在Java语言中通过new操作符调用构造方法来创建对象。

语法格式：

Test test = **new** Test();

Test test = **new** Test("a");

Test：类名

test：创建Test类对象

new：创建对象操作符

“a”：构造方法的参数

test对象被创建出来时，test对象就是一个对象的引用，这个引用在内存中为对象分配了存储空间，每个对象都是相互独立的，

在内存中占据独立的内存地址，并且每个对象都具有自己的生命周期。

实例：

**public** **class** Test {

**public** Test(){

System.***out***.println("创建对象");

}

**public** **static** **void** main(String[] args){

**new** Test(); //创建对象

}

}

#### 访问对象的属性和方法

当使用new操作符创建一个对象后，可以使用“对象.属性名”的方式来获取对象的属性和方法。

**public** **class** Test {

**public** Test(){}

**int** i;

**public** **void** call(){

System.***out***.println("调用对象的方法");

}

**public** **static** **void** main(String[] args) {

Test t = **new** Test(); //创建对象

t.i = 60; //给对象的属性赋值

System.***out***.println(t.i);

t.call(); //调用对象的方法

}

}

#### 对象的引用

在Java语言中尽管一切都可以看作对象，但真正操作标识符实质上是一个引用。

语法格式：

Test test = **new** Test();

Test：类名

test：对象引用

new：创建对象操作符

引用只是存放一个对象的内存地址，并非存放一个对象，对象是存放在内存中的数据集合。可以通过存放的book中的内存地址来访问内存中的对象数据。

#### 对象的比较

在Java语言中有两种对象的比较方法，分别为“==”运算符与equals()方法。实质上这两种方式有着本质区别。equals()方法是Object类的方法，其实质底层也是运用了”==”运算符比较对象的内存地址，但是在String类中重写了Object类的此方法，（重写equals()方法后必须重写hashCode()方法）实现了比较两个引用类型所对应的值是否相等；而“==”运算符比较的是两个对象引用的地址是否相等。

**public** **static** **void** main(String[] args) {

String c1 = **new** String("abc");

String c2 = **new** String("abc");

String c3 = c1;

System.***out***.println("c2==c3的运算结果为：" + (c2 == c3)); //false

System.***out***.println("c2.equalsc3的运算结果为：" + c2.equals(c3)); //true

}

#### 对象的销毁

每个对象都有生命周期，当对象的生命周期结束时，分配给该对象的内存空间将会被回收，垃圾回收器将回收无用的占内存的

资源。

以下两种对象会被Java虚拟机回收：

1. 对象引用超过其作用范围，则这个对象将被视为垃圾。
2. 将对象赋值为null。

虽然垃圾回收机制已经很完善，但垃圾回收器只能回收由new操作符创建的对象，如果某些对象不是通过new操作符在内存中获取一块内存区域，这种对象可能不被垃圾回收机制所识别。

在Java语言中提供了一个finalize()方法，这个方法是Object类的方法，它被声明为protected，用户可以在自己的类中定义这个方法，如果用户在类中定义了finalize()方法，在垃圾回收时首先调用该方法，并且在下一次垃圾回收动作发生时，才能真正回收对象占用的内存。

由于垃圾回收不受人为控制，具体执行时间也不确定，所有finalize()方法也就无法执行，为此Java提供了System.gc()方法强制启动垃圾回收器，告知垃圾回收器来清理。

## 封装、继承与多态

封装、继承和多态是面向对象开发语言中非常重要的一个环节，如果在程序中使用得当，可以将整个程序的架构变得非常有弹性，同时可以减少代码的冗余性。继承机制的使用可以复用一些定义好的类，减少重复代码的编写。多态机制的使用可以动态调整对象的调用，降低对象之间的依存关系。

### 类的封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

Java是注重于类的编写，那么封装自然也是对类的封装。对类封装有很多好处

1. 可以彻底隐藏方法的内部实现，仅仅提供一个调用的方法给其他人，让其他使用这个类的人不需要关心是如何实现的，只要知道该如何调用就行。
2. 隐藏方法的内部实现的好处，可以让保留调用方法不变的同时，随意修改类的结构，而不影响其他人运行结果。
3. 封装还会分开类的属性，将类的属性分成私有属性和公共属性。私有属性仅供类自身调用，和公共属性也仅提供一个供外部调用的方法。
4. 良好的封装是能够减少耦合，达到代码复用。

如何对一个类进行封装，则需要根据这个类本身的客观属性与实际的需要。

比如一个Test类

**public** **class** Test {

//设置日期格式

**private** SimpleDateFormat df = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

**private** String date = df.format(**new** Date()); //对获取的当前日期格式初始化

**public** String getDate(){

**return** date;

}

}

显然这个类是只提供了显示系统的时间，不提供外部去修改系统时间的方法。这个也是根据类本身的客观属性而言，时间是固有不会人为改变的属性，当然诸如此类的例子，还有很多。我们只是简单举例来表明我们该如何设计类的外部访问方法。

**访问权限**

在上面的代码中，我们已经看到可以调用getData()方法对date进行访问，但是date的修饰符是public ，换句话说，我们完全可以new 一个MyTime对象对date属性进行修改和读值，这就违背了我们封装的初衷。所以，Java引入了不同的访问权限来限定这件事情。

1. public：接口访问权限。也就是说public修饰的成员方法，属性都是可以在任何地方所访问的。
2. private: 你无法访问 。也就说private修饰的成员方法，属性 都是只能在这个类中被访问，出了这个类以外的地方，你是无法访问到这个类里面的内容，这十分的有用，我们上面的例子代码，因为是public修饰，所以会出现封装不彻底的问题，如果我们的成员变量date用private修饰，那么就会不出现上述的问题。
3. protected: 继承访问权限。在说到继承访问权限之前，我们思考这样一个问题，我们有一个基类，是需要被继承的。基类中的成员属性，我们是希望能被继承的类所访问，那么我们是可以把他设定成public，但是一旦这样的话，除了这个继承的类，其他类也会访问到。这个显然不是我们所期望的事情，所以Java引入protected修饰符，来表示所修饰的部分是能被派生的类所访问的，同时他也能被同一包里面的其他类所访问。

所以，总结一下，如果我们是希望类中的方法和属性能被所有类所访问，就用public,如果只是能被继承的类和同一包内的类所访问就用protected，如果只是希望在当前类中所访问，不能被其他任何类访问就要用private。

**public** **class** Student {

**private** String name;

**private** String age;

**private** String handleName(String name){

**return** "I m" + name;

}

**private** String handleAge(String age){

**return** age + "岁";

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = **this**.handleName(name);

**public** String getAge() {

**return** age;

}

**public** **void** setAge(String age) {

**this**.age = **this**.handleAge(age);

}

}

在上面的例子中，我们对Student类进行了简单的封装，首先我们切断了外部直接调用到name,age的可能，通过set和get方法来提供访问到name，age的方法，同时，上面的例子中也体现出来了，在外部访问方法不变的情况下，自己随意修改内部的实现，我们首先编写了只供类内部调用的Handle的方法，用Handle方法来修改name，age的显示结果。这样确保了修改显示结果的方法只能自己调用，而且在外部访问不变的情况下，根据需求修改内部的实现。这也是一个简单的封装例子。其调用的例子如下：student对象只能调用如下的内容，其他我们自己编写的内容是无法调用的。

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Student student = **new** Student();

student.setName("byhieg");

student.setAge("23");

System.***out***.println(student.getName() + " " + student.getAge());

}

}

### 类的继承

#### 继承的实现

在Java语言中，继承通过extends关键字来实现。也就是用extends指明当前类是子类，并指明从哪个类继承而来。即在子类的声明中，通过使用extends关键字来显示地指明其父类。子类只能继承父类非私有的属性和方法。

语法格式：

修饰符 **class** 子类名 **extends** 父类名{

类体

}

修饰符：可选参数，用于指定类的访问权限，可选值为public、abstract和final

子类名：必选参数，用于指定子类的名称，类名必须是合法的Java标识符。一般情况下，要求首字母大写

extends 父类名：必选参数，用于指定要定义的子类继承于哪个父类。

实例：

//定义父类

**public** **class** Bird { //定义一个鸟类

String color = "白色"; //定义颜色成员变量

String shin = "羽毛";

}

//定义子类

**public** **class** Pigeon **extends** Bird{ //创建Pigeon类继承自Bird类

**public** **static** **void** main(String[] args) {

Pigeon pigeon = **new** Pigeon(); //实例化一个对象

System.***out***.println(pigeon.color); //输出继承的属性

}

}

注意：在Java中只能有单继承，不能有多继承。一个子类只能继承一个父类不能继承多个父类。

#### 继承中的重写

重写是指父子类之间的关系，当子类继承父类中所有可能被子访问的成员方法时，如果子类的方法名与父类的方法名相同，那么子类就不能继承父类的方法，此时，称为子类的方法重写了父类的方法。重写体现了子类补充或者改变父类方法的能力。通过重写，可以使一个方法在不同的子类中表现出不同的行为。

//定义父类

**public** **class** Animal {

**public** **void** cry(){

System.***out***.println("动物发出叫声");

}

}

//定义Dog类继承自Animal类

**public** **class** Dog **extends** Animal{

**public** **void** cry(){

System.***out***.println("狗发出的叫声");

}

}

**public** **class** Say{

**public** **static** **void** main(String[] args) {

Animal a = **new** Animal();

Dog d = **new** Dog();

System.***out***.println(a.cry()); //"动物发出叫声"

System.***out***.println(d.cry()); //"狗发出的叫声"

}

}

注意：重写父类的方法，控制权限必须不能低于父类方法的控制权限。

#### final关键字

final关键字“最终的”。该关键字可以修饰类、成员变量、局部变量、方法。

当修饰类的时候，该类则不能被继承。

当修饰成员变量的时候，该成员变量在定义的时候必须初始化，且不能被修改值（常量）。

当修饰局部变量的时候，该局部变量在定义的时候可以不用初始化，但是在赋值后就不能被修改。

当修饰方法的时候，该方法可以被重载，但是不能被重写。

#### 使用super关键字

子类可以继承父类的非私有成员变量和成员方法（不是以private关键字修饰的）作为自己的成员变量和成员方法。但是，如果子类中声明的成员变量与父类的成员变量同名，则子类不能继承父类的成员变量（成员方法也是如此，会被重写），这时如果想在子类中访问父类中被子类隐藏的成员方法或成员变量，则需要使用super关键字。

super可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

super关键字主要有两种用途：

1. **调用父类的构造方法**

当子类在实例化一个对象的时候，JVM首先会去找到子类的父类，先会实例化出父类的对象后，再去实例化子类的对象。如果我们没有显示的定义构造方法的时候，JVM虚拟机会为父类和子类添加一个无参构造方法，在实例化子类的对象的时候，JVM虚拟机会通过子类的无参构造方法去调用其父类的无参构造方法先实例化父类的对象，然后在调用子类的无参构造方法实例化子类的对象。如果我们在子类的构造方法中显示的调用父类的构造方法，必须在子类的构造方法中通过super关键字去调用父类的构造方法，而且调用父类的构造方法语句必须在子类构造函数的第一行（因为JVM需要先实例化父类的对象）。

语法格式：

**super**(参数列表);

如果父类的构造方法中包括参数，则参数列表为必选项，用于指定父类构造方法的入口参数。

实例：

//定义父类

**public** **class** Beast {

String str = "";

**public** Beast(){}

**public** Beast(String str){

**this**.str = str;

}

**public** **void** move(){

System.***out***.println("跑");

}

}

//定义Tiger类继承自Animal类

**public** **class** Tiger **extends** Beast{

**public** Tiger(){

**super**("条纹"); //调用父类中带参数的构造方法

}

}

如果我们在实例化子类的对象的时候，通过子类的有参构造函数的方式去调用父类的有参构造函数，并且完成对父类和子类的属性赋值，则必须先通过super关键字调用父类的有参构造方法实例化父类对象后才能再为子类属性赋值。

//定义父类

**public** **class** ManKind {

**protected** **int** sex;

**private** **int** salary;

**public** ManKind(){} //无参构造

**public** ManKind(**int** sex, **int** salary){ //有参构造方法

**this**.sex = sex;

**this**.salary = salary;

}

}

//子类

**public** **class** Kids **extends** ManKind{

**private** **int** yearsOld;

**public** Kids(){} //无参构造

**public** Kids(**int** sex, **int** salary, **int** yearsOld){

**super**(sex, salary); //调用父类的有参构造

**this**.yearsOld = yearsOld

}

//主方法

**public** **static** **void** main(String[] args) {

Kids someKid = **new** Kids(0, 500, 30);

}

}

**子类为什么会默认访问父类的默认构造函数：**

因为子类继承了父类，获取到了父类中的内容（属性），所以在使用父类内容之前，要先看父类是如何对自己的内容进行初始化，所以子类在构造对象时候，必须访问父类的构造函数，为了完成这个必须的动作，就在子类的构造函数中加入了super()语句。如果父类中没有定义空参数构造函数，那么子类的构造函数必须使用super明确要调用父类中那个构造函数，否则子类无法完成初始化。

**为什么super(…)或this(…)调用语句只能作为构造方法中的第一句出现？**

子类构造函数如果使用this调用了本类构造函数时，那么super就没有了，因为super和this都只能定义在第一行，所以只能有一个.

1. **操作被隐藏的成员变量和被重写的成员方法**

如果想在子类中操作父类中被隐藏的成员变量和被重写的成员方法，也可以使用super关键字

语法格式：

**super**.成员变量名

**super**.成员方法名(参数列表)

实例：

//定义父类

**public** **class** ManKind {

**protected** **int** sex;

**public** **void** move(){

System.***out***.println("跑");

}

}

//子类

**public** **class** Kids **extends** ManKind{

**public** Kids(){

**super**.sex = 20;

**super**.move();

}

}

1. **继承的缺陷**

父类变，子类就必须变。

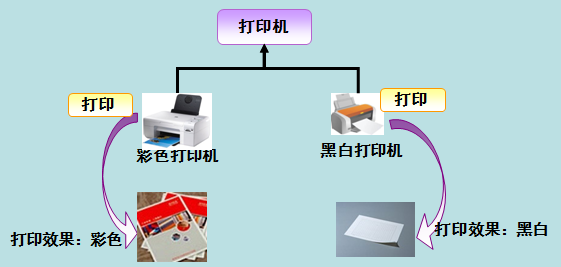
继承破坏了封装，对于父类而言，它的实现细节对于子类来说都是透明的。

继承是一种强耦合关系。

### 多态

#### 多态的理解

多态是同一个行为具有多个不同表现形式或形态的能力。



现实中，比如我们按下 F1 键这个动作：

如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；

如果当前在 Word 下弹出的就是 Word 帮助；

在 Windows 下弹出的就是 Windows 帮助和支持。

同一个事件发生在不同的对象上会产生不同的结果。

#### Java中多态的体现

多态性是面向对象程序设计的重要部分。在Java中，通常使用方法的重载和重写实现类的多态性。

1. 重写之所以具有多态是因为父类的方法在子类中被重写，子类和父类的方法名称相同，但完成的功能却不一样，所以重写也具有多态性。
2. 重载是指在一个类中出现多个方法名相同，但参数个数或参数类型不同的方法，则成为方法的重载。这些方法在被调用的时候会根据参数的个数和参数类型来判断具体调用哪个方法

注意：在进行方法的重载时，方法返回值的类型不能作为区分方法的标志。

#### 多态的优点

1. 消除类型之间的耦合关系
2. 可替换性
3. 可扩充性
4. 接口性
5. 灵活性
6. 简化性

#### 多态存在的三个必要条件

1. 继承
2. 重写
3. 父类引用指向子类对象

#### 多态的具体实现

例如：

定义了一个动物Pet类，该类中定义了一个public eat()方法。再定义了一个Cat和Dog类，Cat类和Dog类都继承自Pet类。在Cat类和Dog类中重写了父类的eat()方法，实现了子类的特有方法。我们在定义一个Master主人类，主人类中定义了一个public foot(Cat cat)方法喂食。（这里我们不使用多态，当我们在完成对不同动物喂食的同一动作是，需要重载Master类中的foot()方法。这样既增加了代码的冗余，也不够灵活；如果增加了其他的动物，则需要在Master类中继续重载foot()方法。）

//动物类，父类

**public** **abstract** **class** Pet {

**protected** **int** health;

**public** **abstract** **void** eat();

}

//狗类，该类继承自动物类并重写了eat()方法

**public** **class** Dog **extends** Pet {

**public** **void** eat(){

System.***out***.println("狗吃骨头");

**this**.health += 4;

System.***out***.println(**this**.health);

}

}

//猫类，该类继承自动物类并重写了eat()方法

**public** **class** Cat **extends** Pet {

**public** **void** eat(){

System.***out***.println("猫吃鱼");

**this**.health++;

System.***out***.println(**this**.health);

}

}

//主人类，定义了一个food()方法用于喂食

**public** **class** Master {

//给猫喂食

**public** **void** food(Cat cat){

cat.eat();

}

//给猫喂食

**public** **void** food(Dog dog){

dog.eat();

}

}

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Master master = **new** Master(); //定义主人对象

Cat cat = **new** Cat(); //定义猫对象

Dog dog = **new** Dog(); //定义狗对象

master.food(cat); //实现喂猫

master.food(dog); //实现喂狗

}

}

以上实例中的情况我们可以使用Java的多态来实现对象和类型的解耦，减少Master类中的代码冗余，增加扩展性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。

1. Master类中去掉food()方法的重载，将参数类型修改为动物父类的类型

//主人类，定义了一个food()方法用于喂食

**public** **class** Master {

//给动物喂食

**public** **void** food(Pet pet){

pet.eat();

}

}

1. 在主方法中定义父类类型的变量指向子类的对象

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Pet cat = **new** Cat(); //定义Pet类型的猫对象

Pet dog = **new** Dog(); //定义Pet类型的狗对象

cat.eat(); //父类的类型引用实际调用的是子类中重写的方法

dog.eat();

}

}

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Master master = **new** Master(); //定义主人对象

Pet cat = **new** Cat(); //定义猫对象

Pet dog = **new** Dog(); //定义狗对象

master.food(cat); //实现喂猫

master.food(dog); //实现喂狗

}

}

在主方法中定义父类类型的子类对象，JVM虚拟机会自动将子类的对象类型进行向上转型，在内存栈中Pet cat变量保存的地址是指向堆内存中cat对象。所以我们在调用master对象的food()方法时，将对象的内存地址复制给pet变量。所以pet.eat()方法实际执行的是cat对象的eat()方法。dog对象也相同的情况

注意：Java中像多态这样通过父类类型的变量来指向子类对象的方式，父类类型的变量只能调用子类中重写了父类的方法。如果子类中有自定义的属性和方法，则父类类型的变量无法调用子类的自定义方法和属性。

如果需要调用子类对象的自定义属性和方法，有两种方式可以实现：

1. 重写实例化一个子类类型的对象。

//猫类，该类继承自动物类并重写了eat()方法

**public** **class** Cat **extends** Pet {

**public** **void** eat(){

System.***out***.println("猫吃鱼");

**this**.health++;

System.***out***.println(**this**.health);

}

**public** **void** run(){

System.***out***.println("猫在抓老鼠");

}

}

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Master master = **new** Master(); //定义主人对象

Pet cat = **new** Cat(); //定义猫对象

Pet dog = **new** Dog(); //定义狗对象

master.food(cat); //实现喂猫

master.food(dog); //实现喂狗

//重新实例化了一个对象

Cat cat1 = **new** Cat();

cat1.run();

}

}

1. 将父类类型的变量向下转型，转换为对象自己的类型引用

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Master master = **new** Master(); //定义主人对象

Pet cat = **new** Cat(); //定义猫对象

Pet dog = **new** Dog(); //定义狗对象

master.food(cat); //实现喂猫

master.food(dog); //实现喂狗

//向下转型，没有产生新的对象

Cat cat1 = (Cat) cat;

cat1.run(); //调用子类中定义的方法

}

}

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处：可以使程序有良好的扩展（在扩展子类的时候，无需理会其他的代码，只需要实现子类即可），并可以对所有类的对象进行通用处理。

## 类的高级特性

### 抽象类

在继承的层次结构中，随着每一个新子类的出现，类会变得越来越明确和具体。如果从一个子类追溯到父类，类就会变得更通用、更加不明确。类的设计应该确保父类包含它的子类的共同特征。有时候，一个父类设计得非常抽象，以至于它都没有任何具体的实例。这样的类称为抽象类。

抽象类是Java对现实生活的描述，我们定义抽象类的作用是用于集合一类事物共同特性和行为，但它又不能代表具体的某一个事物，在抽象类中可以派生出这一类事物中的具体子类，而子类则可以具体代表某一具体事物从而可以实例化出具体的对象。

使用abstract关键字修饰的类称为抽象类。抽象类的作用是将所有子类中的相同数据提取出来定义的类，主要实现子类的继承。也是面向对象编程的客观世界描述。抽象类因为具有子类的共同属性，而没有实例化具体对象的特殊属性，所以抽象类是不能实例化一个具体的对象。抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

语法格式：

**abstract** **class** 类名{

类体

}

实例：

//定义动物抽象类

**public** **abstract** **class** Pet {

**protected** **int** health;

**public** Pet(){} //构造方法

**public** **abstract** **void** eat(); //抽象方法

**public** **void** run(){ //普通方法

System.***out***.println("run");

}

}

抽象类中可以定义构造方法，但是为什么不能直接实例化对象？

当一个类可以被实例化的时候，则可以调用类中除被static修饰的属性和方法。但是在抽象类里存在抽象方法，而抽象方法没有方法体，没有方法体就无法进行调用。既然无法进行方法调用的话，又怎么去产生实例化对象呢。

抽象类中有构造方法主要是因为在抽象类中可以定义属性和普通的方法，而在使用属性和普通方法之前必须得初始化。在其抽象类的实现子类中，构造方法中默认隐式使用super()方式调用父类的构造方法来初始化父类的属性和方法。如果需要调用父类的属性和方法，可以在子类中使用super关键字来调用父类的属性和方法。子类对象实例化的时候，依然满足先执行父类构造，再执行子类构造的顺序。

//猫类，该类继承自动物类并重写了eat()方法

**public** **class** Cat **extends** Pet {

**public** Cat(){ //显示定义无参构造方法

**super**(); //显示定义了调用父类的无参构造方法

}

**public** **void** eat(){

System.***out***.println("猫吃鱼");

**this**.health++;

**super**.run(); //调用父类的普通方法

System.***out***.println(**this**.health);

}

}

### 抽象方法

使用abstract关键字修饰的方法被称为抽象方法，抽象方法必须存在于抽象类中不能存在与非抽象类中。抽象方法只有方法名、返回值、参数表，没有具体的实现过程。抽象方法的具体实现必须在继承该抽象类的子类中定义实现。如果子类中没有对父类中抽象方法的实现，则子类也必须声明为抽象类。

语法格式：

**abstract** 返回值类型 方法名(参数列表);

方法返回值类型：必选参数，用于指定方法的返回值类型，如果该方法没有返回值，可以使用关键字void进行标识。方法返回

值的类型可以是任何Java数据类型。

方法名：必选参数，用于指定抽象方法的明细，方法名必须是合法的Java标识符。

参数列表：方法的参数可以是任何Java数据类型。

实例：

**public** **abstract** **void** eat(); //抽象方法

抽象方法不能使用private或static关键字进行修饰。

完整实例：

//定义动物抽象类

**public** **abstract** **class** Pet {

**protected** **int** health;

**public** Pet(){} //构造方法

**public** **abstract** **void** eat(); //抽象方法

**public** **void** run(){ //普通方法

System.***out***.println("run");

}

}

### 抽象类的使用原则

1. 抽象方法必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public；
2. 抽象类不能直接实例化，需要依靠子类采用向上转型的方式处理；
3. 抽象类必须有子类，使用extends继承，一个子类只能继承一个抽象类；
4. 子类（如果不是抽象类）则必须覆写抽象类之中的全部抽象方法（如果子类没有实现父类的抽象方法，则必须将子类也定义为为abstract类。）；

### 抽象类的使用及注意

1. 抽象类中有构造方法吗？

由于抽象类里会存在一些属性，那么抽象类中一定存在构造方法，其存在目的是为了属性的初始化。 并且子类对象实例化的时候，依然满足先执行父类构造，再执行子类构造的顺序。

1. 抽象类可以用final声明吗？

不能，因为抽象类必须有子类，而final定义的类不能有子类；

1. 抽象类能否使用static声明吗？

先看一个关于外部抽象类的范例：

//定义动物抽象类

**static** **abstract** **class** Pet {

**public** **abstract** **void** eat(); //抽象方法

}

//实现子类

**class** Cat **extends** Pet{

@Override

**public** **void** eat() {

System.***out***.println("\*\*\*\*\*\*\*\*\*\*");

}

//主方法,编译的时候会报错

**public** **static** **void** main(String[] args) {

Pet pet = **new** Cat(); //向上转型

pet.eat();

}

}

编译包类型错误：



再看一个关于内部抽象类：

//定义动物抽象类

**abstract** **class** Pet {

//定义内部类

**static** **abstract** **class** pet{

**public** **abstract** **void** eat(); //抽象方法

}

}

执行结果： 

//实现子类

**class** Cat **extends** Pet.pet{

@Override

**public** **void** eat() {

System.***out***.println("\*\*\*\*\*\*\*\*\*\*");

}

//主方法,编译的时候会报错

**public** **static** **void** main(String[] args) {

Pet.pet pet = **new** Cat(); //向上转型

pet.eat();

}

}

由此可见，外部抽象类不允许使用static声明，而内部的抽象类运行使用static声明。使用static声明的内部抽象类相当于一个外部抽象类，继承的时候使用“外部类.内部类”的形式表示类名称。

1. 可以直接调用抽象类中用static声明的方法么？

任何时候，如果要执行类中的static方法的时候，都可以在没有对象的情况下直接调用，对于抽象类也一样。

1. 有时候由于抽象类中只需要一个特定的系统子类操作，所以可以忽略掉外部子类。这样的设计在系统类库中会比较常见，目的是对用户隐藏不需要知道的子类。

//定义动物抽象类

**public** **abstract** **class** Pet {

**public** **abstract** **void** eat(); //抽象方法

//集成了抽象类的内部子类

**private** **static** **class** Cat **extends** Pet{

@Override

**public** **void** eat() { //实现父类的方法

System.***out***.println("Hello World!");

}

}

//定义一个静态方法用于创建子类的实例

**public** **static** Pet getCat(){

**return** **new** Cat();

}

}

//主方法类

**public** **class** index{

**public** **static** **void** main(String[] args) {

//此时取得抽象类对象的时候完全不需要知道Cat类这个子类的存在

Pet pet = Pet.*getCat*();

pet.eat();

}

}

### 内部类

如果在一个类中再定义一个类，就将在类中再定义的那个类称为内部类。内部类可分为成员内部类、局部内部类、匿名内部类和静态内部类。

#### 成员内部类

在一个类中使用内部类可以在内部类中直接存取其所在类的所有成员变量。

语法格式：

**public** **class** OuterClass {

**private** **class** innerClass{

//.......

}

}

在内部类中可以随意使用外部类的成员方法以及成员变量，尽管这些类成员被修饰为private。但是外部类不能直接访问并使用内部内的成员变量以及成员方法。

内部类的实例一定要绑定在外部类的实例上，如果在外部类中初始化一个内部类对象，那么内部类对象就会绑定在外部类对象上。内部类初始化与其他类初始化方式相同，都是使用new关键字。

实例：

**public** **class** OuterClass {

innerClass in = **new** innerClass();

**public** **void** outf(){

in.inf(); //调用内部类的方法

}

**private** **class** innerClass{

**public** **void** inf(){

System.***out***.println("内部类的方法");

}

}

}

内部类的对象实例化操作必须在外部类或外部类中的非静态方法中实现。如果在外部类和非静态方法之外实例化内部类对

象，需要使用“外部类.内部类”的形式指定该对象的类型。

**public** **static** **void** main(String[] args) {

OuterClass out = **new** OuterClass();

OuterClass.innerClass in = out.**new** innerClass();

}

在实例化内部类对象时，不能在new操作符之前使用外部类名称那种形式实例化内部类对象，而是应该使用外部类的对象

来创建其内部类的对象。内部类对象会依赖于外部类对象，除非已经存在一个外部类对象，否则类中不会出现内部类对象。

**使用this关键字获取内部类与外部类的引用**

在类中如果内部类与外部类遇到成员变量重名的情况可以使用this关键字进行处理。在内部类中使用this.x语句可以调用内部类的成员变量x，而使用“外部类名.this.x”语句可以调用外部类的成员变量x

**public** **class** OuterClass {

**private** **int** x;

//内部类

**private** **class** innerClass{

**private** **int** x = 9;

**public** **void** doit(**int** x){

x++; //调用的是参数

**this**.x++; //调用的是内部类的变量

OuterClass.**this**.x++; //调用的是外部类的变量

}

}

}

#### 局部内部类

局部内部类是指在类的方法中定义的内部类，它的作用范围也是在这个方法体内。

**public** **class** OuterClass {

**public** **void** sell(**int** price){

**class** Apple{ //定义局部类

**int** innerPrice = 0;

**public** **void** Apple(){}

}

Apple apple = **new** Apple(); //实例化局部类

System.***out***.println(apple.innerPrice); //调用局部类的变量

}

}

在Java中外部类不能访问局部类的成员方法和成员变量，但是局部类可以访问当前代码块的常量以及此外部类的所有成员。

#### 匿名内部类

在编写程序代码时，不一定要给内部类取一个名字，可以直接以对象名来代替。匿名内部类的所有实现代码都需要在大括号之间进行编写。

语法格式：

**return** **new** A(){

...//内部类体

}

A：表示对象名

由于匿名内部类没有名称，所以匿名内部类使用默认构造方法来生成匿名内部类的对象。在匿名内部类定义结束后，需要

加分号标识，这个分号并不代表定义内部类结束的标识，而代表创建匿名内部类的引用表达式的标识。

**public** **class** OuterClass {

**public** **static** **void** main(String[] args) {

**final** String str = "javase";

IStringDeal s = **new** IStringDeal(){ //定义匿名内部类

**public** String filter(){

**return** str;

}

};

System.***out***.println(s.filter());

}

}

#### 静态内部类

在内部类前面添加修饰符static，这个内部类就是静态内部类。一个静态内部内中可以声明static成员，但是在非静态内部类中不可以声明静态成员。静态内部类有一个最大的特点，就是不可以使用外部类的非静态成员，所以静态内部类在程序开发中比较少见。

静态内部类的特点：

1. 创建静态内部类的对象，不需要其外部类的对象。
2. 不能从静态内部类的对象中访问非静态外部类的对象。

**public** **class** OuterClass {

**int** x = 100;

**static** **class** Inner{ //定义静态内部类

**public** **void** doInner(){

System.***out***.println("外部类" + x); //调用报错

}

}

}

#### 内部类的继承

内部类和其他普通类一样可以被继承，但是继承内部类比继承普通类要复杂，需要设置专门的语法来完成

**class** ClassA{

**class** ClassB{

}

}

**public** **class** OuterClass **extends** ClassA.ClassB {

**public** OuterClass(ClassA a){

a.**super**();

}

}

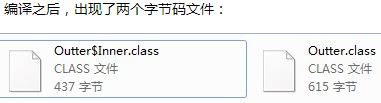
在某个类继承内部类时，必须硬性给予这个类一个带参数的构造方法，并且该构造方法的参数为需要继承内部类的外部类的引

用，同时在构造方法体中使用a.super()语句，这样才为继承提供了必要的对象引用。

### 深入理解内部类

1. **为什么成员内部类可以无条件访问外部类的成员**

在此之前，我们已经讨论过了成员内部类可以无条件访问外部类的成员，那具体究竟是如何实现的呢？事实上，编译器在进行编译的时候，会将成员内部类单独编译成一个字节码文件。



**public** **class** Outter {

**private** Inner inner = **null**;

**public** Outter(){

}

**public** Inner getInnerInstance(){

**if**(inner == **null**){

inner = **new** Inner();

}

**return** inner;

}

**protected** **class** Inner{ //内部类

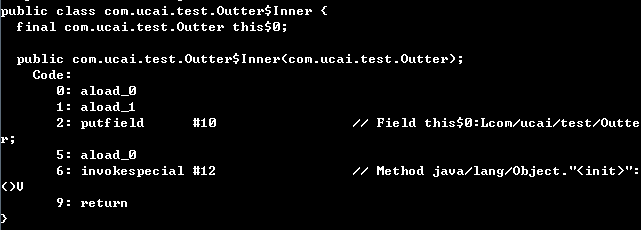
**public** Inner(){

}

}

}

反编译Outter$Inner.class文件得到下面信息：javap –c Outter$Inner.class



首先编译器会默认为成员内部类添加了一个指向外部类对象的引用，



虽然我们在定义的内部类的构造器是无参构造器，编译器还是会默认添加一个参数，该参数的类型为指向外部类对象的一个引用，所以成员内部类中的Outter this&0 指针便指向了外部类对象，因此可以在成员内部类中随意访问外部类的成员。从这里也间接说明了成员内部类是依赖于外部类的，如果没有创建外部类的对象，则无法对Outter this&0引用进行初始化赋值，也就无法创建成员内部类的对象了。



1. 为什么局部内部类和匿名内部类只能访问局部final变量

**public** **class** Test{

**public** **static** **void** main(String[] agrs){

}

**public** **void** test(**final** **int** b){

**final** **int** a = 10;

**new** Thread(){

**public** **void** run(){

System.***out***.println(a);

System.***out***.println(b);

};

}.start();

}

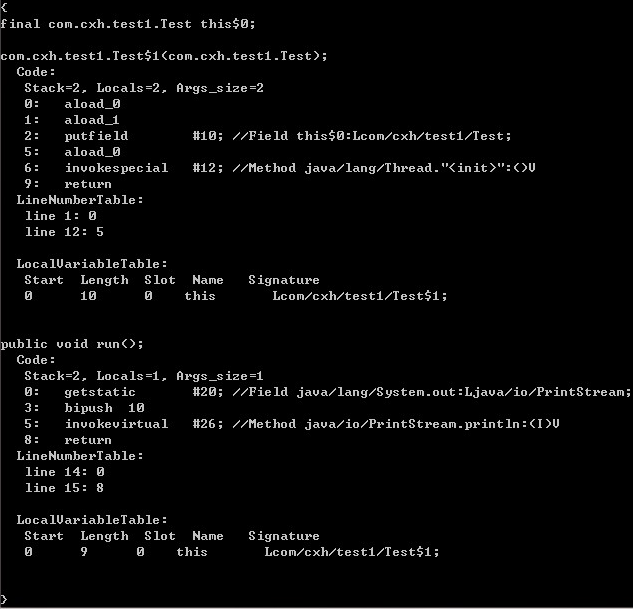
}

这段代码会被编译成两个class文件：Test.class和Test。默认情况下，编译器会为匿名内部类和局部内部类起名为1.class。默认情况下，编译器会为匿名内部类和局部内部类起名为Outterx.class（x为正整数）。



如果把变量a和b前面的任一个final去掉，这段代码都编译不过。当test方法执行完毕之后，变量a的生命周期就结束了，而此时Thread对象的生命周期很可能还没有结束，那么在Thread的run方法中继续访问变量a就变成不可能了，但是又要实现这样的效果，怎么办呢？Java采用了 复制 的手段来解决这个问题。

将这段代码的字节码反编译可以得到下面的内容：



我们看到在run方法中有一条指令



这条指令表示将操作数10压栈，表示使用的是一个本地局部变量。这个过程是在编译期间由编译器默认进行，如果这个变量的值在编译期间可以确定，则编译器默认会在匿名内部类（局部内部类）的常量池中添加一个内容相等的字面量或直接将相应的字节码嵌入到执行字节码中。这样一来，匿名内部类使用的变量是另一个局部变量，只不过值和方法中局部变量的值相等，因此和方法中的局部变量完全独立开。

下面再看一个例子：

**public** **class** Test{

**public** **static** **void** main(String[] agrs){

}

**public** **void** test(**final** **int** b){

**new** Thread(){

**public** **void** run(){

System.***out***.println(a);

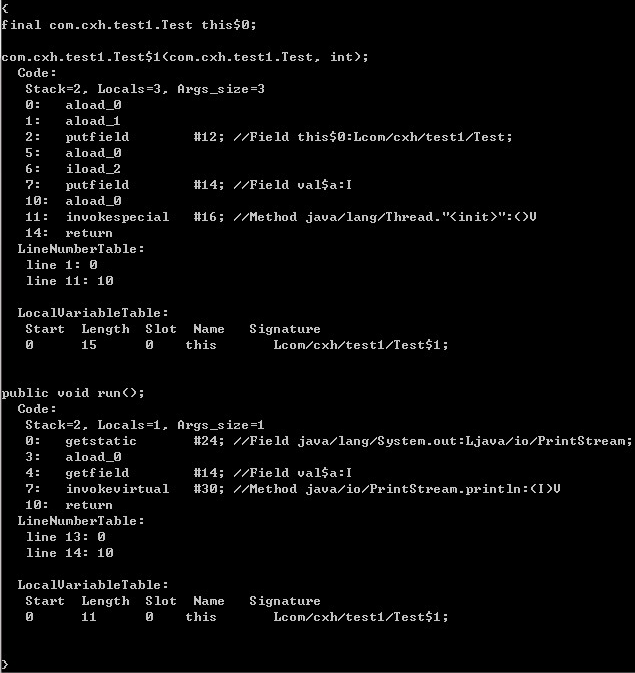
};

}.start();

}

}

反编译得到：



我们看到匿名内部类Test$1的构造器含有两个参数，一个是指向外部类对象的引用，一个是int型变量，很显然，这里是将变量test方法中的形参a以参数的形式传进来对匿名内部类中的拷贝（变量a的拷贝）进行赋值初始化。

也就说如果局部变量的值在编译期间就可以确定，则直接在匿名内部里面创建一个拷贝。如果局部变量的值无法在编译期间确定，则通过构造器传参的方式来对拷贝进行初始化赋值。

从上面可以看出，在run方法中访问的变量a根本就不是test方法中的局部变量a。这样一来就解决了前面所说的 生命周期不一致的问题。但是新的问题又来了，既然在run方法中访问的变量a和test方法中的变量a不是同一个变量，当在run方法中改变变量a的值的话，会出现什么情况？

对，会造成数据不一致性，这样就达不到原本的意图和要求。为了解决这个问题，java编译器就限定必须将变量a限制为final变量，不允许对变量a进行更改（对于引用类型的变量，是不允许指向新的对象），这样数据不一致性的问题就得以解决了。

到这里，想必大家应该清楚为何 方法中的局部变量和形参都必须用final进行限定了。

1. 静态内部类有特殊的地方吗

从前面可以知道，静态内部类是不依赖于外部类的，也就说可以在不创建外部类对象的情况下创建内部类的对象。另外，静态内部类是不持有指向外部类对象的引用的，这个读者可以自己尝试反编译class文件看一下就知道了，是没有Outter this&0引用的。

### 内部类的使用场景和好处

每个内部类都能独立的继承一个接口的实现，所以无论外部类是否已经继承了某个(接口的)实现，对于内部类都没有影响。内部类使得多继承的解决方案变得完整，

1. 方便将存在一定逻辑关系的类组织在一起，又可以对外界隐藏。
2. 方便编写事件驱动程序
3. 方便编写线程代码

个人觉得第一点是最重要的原因之一，内部类的存在使得Java的多继承机制变得更加完善。

## 接口的使用

Java语言只支持单重继承，不支持多继承，即一个类只能有一个父类。但实际应用中又经常要使用多继承来解决问题，因此Java语言提供了接口来实现类的多重继承功能。

#### 接口的定义

使用interface来定义一个接口。接口的定义与类的定义类似，也是分为接口的声明和接口体，其中接口体由静态常量和抽象方法定义两部分组成。

语法格式：

修饰符 **interface** 接口名 **extends** 父接口名列表{

**public** **static** **final** 大写变量名;

**public** **abstract** 方法名;

}

修饰符：可选参数，用于指定接口的访问权限，可选值为public。如果省略则使用默认的访问权限

接口名：必选参数，用于指定接口的名称，接口名必须是合法的Java标识符。一般情况下，要求首字母大写

extends 父接口名列表：可选参数，用于指定要定义的接口继承于哪个父接口。当使用extends关键字时，父接口名为必选参数。接口只能继承接口不能继承一般的类，且接口可以继承多个接口。

属性：接口中可以定义变量，但是变量必须被static和final关键字所修饰，且变量名必须全部大写。（定义常量）

方法：在java8之前接口中的方法只有定义而没有被实现。在Java8开始，接口可以包含多个默认方法和静态方法。默认方法和静态方法都可以在接口中实现方法体。

实例：

**public** **interface** MyInterface {

**void** printIt(String text);

//默认方法

**default** **public** **void** printUtf8To(String text, OutputStream outputStream){

**try** {

outputStream.write(text.getBytes("UTF-8"));

} **catch** (IOException e) {

**throw** **new** RuntimeException("Error writing String as UTF-8 to OutputStream", e);

}

}

//静态方法

**static** **void** printItToSystemOut(String text){

System.***out***.println(text);

}

}

**public** **interface** ICalculate{

**final** **float** ***PI*** = 3.14159F; //定义一个常量表示圆周率

**float** getArea(**float** r); //定义计算面积的方法

**float** getCircumference(**float** r); //定义计算周长的方法

}

在Java中接口文件的文件名必须与接口名相同。如果一个接口只有一个抽象方法，那么这个接口会默认自动变成函数式接口。

如果使用了@FunctionalInterface 注解对接口进行修饰，说明这个接口是一个函数式接口，在该接口只能有一个抽象方法 (不限制静态方法和默认方法)。一个函数式接口可以通过 Lambda 表达式来创建该接口的对象。

#### 接口的实现

接口在定义后，就可以在类中实现该接口。在类中使用接口可以使用implements关键字。实现类中必须实现接口中定义的所有方法，否则该类必须声明为抽象的类。

语法结构：

修饰符 **class** 类名 **extends** 父类名 **implements** 接口列表{

}

修饰符：可选参数，用于指定类的访问权限，可选值为public、abstract、final

类名：必选参数，用于指定类的名称，类名必须是合法的Java标识符。一般情况下，要求首字母大写

extends 父类名：可选参数，用于指定要定义的类继承于哪个父类。当使用extends关键字时，父类名为必选参数

implements 接口列表：可选参数，用于指定该类实现的是哪些接口。当使用implements关键字时，接口列表为必选参数。当接口列表中存在多个接口名时，各个接口名之间使用逗号分隔。

在类中实现接口时，方法名、返回值类型、参数的个数及类型必须与接口中的完全一致，并且必须实现接口中的所有方法。接口中定义的默认方法和静态方法不需要在实现类中实现方法体。静态方法属于类级别的方法，可以直接通过接口调用且不会被实现类继承；接口中的默认方法可以被实现类继承且可以在实现类中重写接口中的默认方法。

注意：如果一个类实现了多个接口，并且这些接口之间没有相互继承关系，同时存在相同的 default 方法时会报错，不过你可以在实现类中重写 default 方法并通过 <接口>.super.<方法名>(); 形式指定调用哪个父接口中的 default 方法。

实例：

**public** **interface** ICalculate{

**final** **float** ***PI*** = 3.14159F; //定义一个常量表示圆周率

**float** getArea(**float** r); //定义计算面积的方法

**float** getCircumference(**float** r); //定义计算周长的方法

}

//定义一个实现上述接口的类

**public** **class** Cire **implements** ICalculate{

@Override

**public** **float** getArea(**float** r) {

**float** area = **this**.***PI*** \* r \* r;

**return** area;

}

@Override

**public** **float** getCircumference(**float** r) {

**float** circumference = 2 \* **this**.***PI*** \* r;

**return** circumference;

}

}

#### 接口的应用

接口的灵活性就在于“规定一个类必须做什么，而不管你如何做”。我们可以定义一个接口类型的引用变量来引用实现接口的类的实例。

**在接口中使用多态**

//定义动物接口

**public** **interface** Pet {

**public** **void** say();

}

//实现动物接口的狗类

**public** **class** Dog **implements** Pet {

@Override

**public** **void** say() {

System.***out***.println("小狗叫");

}

}

//实现动物接口的猫类

**public** **class** Cat **implements** Pet {

@Override

**public** **void** say() {

System.***out***.println("小猫叫");

}

}

//主人类

**public** **class** Master {

//定义传入Pet类型的参数

**public** **void** play (Pet pet){

pet.say();;

}

}

//主方法类

**public** **class** Index {

**public** **static** **void** main(String[] args) {

Master master = **new** Master();

Pet cat = **new** Cat(); //通过多态的方法实例化一个Cat实例

Pet dog = **new** Dog();

master.play(cat); //小猫叫！

master.play(dog); //小狗叫

}

}

#### 标记接口

标识接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标识接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event 包中的 MouseListener 接口继承的 java.util.EventListener 接口定义如下：

**public** **interface** ICalculate{}

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

**建立一个公共的父接口：**

正如EventListener接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了EventListener接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

**向一个类添加数据类型：**

这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

#### 接口的特点

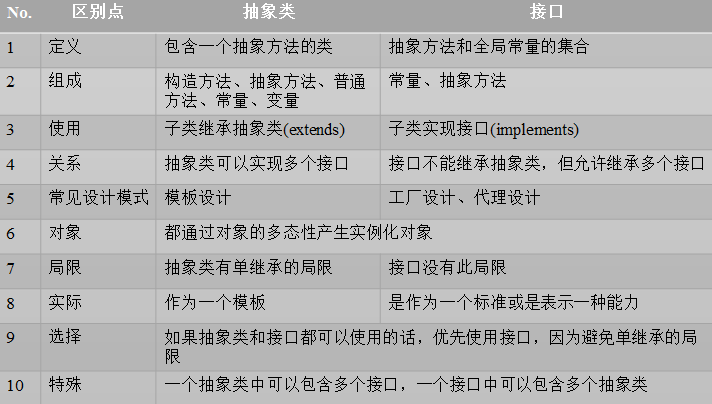
在java中，接口被看作是一种特殊的类。就像常规类一样，每个接口都被编译为独立的字节码文件。与抽象类相似，不能使用new操作符创建接口的实例，但是大多数情况下，使用接口或多或少有点像使用抽象类。

**接口的特点**：

1. 用interface来定义
2. 接口中的所有变量会被隐式的指定为 public static final 变量（并且只能是 public，用 private 修饰会报编译错误）。
3. 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 public abstract（只能是 public abstract，其他修饰符都会报错）。
4. 接口没有构造器。
5. 接口采用多继承机制，且只能继承接口，不能继承一般的类。
6. 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法
7. 接口的主要用途就是被实现类实现（面向接口编程）
8. 实现接口的类中必须提供接口中所有方法的具体实现内容，方可实例化。否则，仍为抽象类。
9. 与继承关系类似，接口与实现类之间存在多态性
10. 定义Java类的语法格式：先写extends，后写implements

**抽象类和接口的区别**

1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的。
3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。



## 泛型

泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。 Java语言引入泛型的好处是安全简单。

集合中可以存放Object类型的对象（所有类型的对象）但是如果集合中存放了自定义的类型时，在遍历集合的时候就会出现问题，很难进行向下转型。所以泛型就是强制规定了集合中只能存放给定的数据类型。

List<Student> a = **new** ArrayList<Student>();

**private** Map<Integer, Role> roleMap = **new** HashMap<Integer, Role>();

泛型用于限制在集合中存储的数据类型。如果需要向集合中添加基本数据类型，则需先将基本数据类型转换为对应的包装类。

### 定义简单泛型类

一个泛型类就是具有一个或多个类型变量的类。对于这个类来说，我们只关注泛型，而不会为数据存储的细节烦恼。

**public** **class** Pair<T>{

**private** T first;

**private** T second;

**public** Pair(){

first = **null**;

second = **null**;

}

**public** Pair(T first,T second){

**this**.first = first;

**this**.second = second;

}

**public** **void** setFirst(T first) {

**this**.first = first;

}

**public** **void** setSecond(T second) {

**this**.second = second;

}

}

Pair类引入了一个类型变量T，用尖括号（<>）括起来，并放在类名的后面。泛型可以有多个类型变量。

**public** **class** Pair<T, U>{

…

}

类定义中的类型变量指定方法的返回类型以及域和局部变量的类型。

**private** T first;

用具体的类型替换类型变量就可以实例化泛型类型。

Pair<String>

### 泛型方法

前面介绍了如何定义一个泛型类，实际上还可以定义一个带有类型参数的简单方法

**public** **class** ArrayAlg{

**public** **static** <T> T getMiddle(T...a){

**return** a[a.length / 2];

}

}

泛型方法可以定义在普通类中，也可以定义的在泛型类中。当调用一个泛型方法时，在方法名前的尖括号中放入具体的类型：

String middle = ArrayAlg.<String>*getMiddle*("John", "Q");

## Java集合类

Java中提供了不同的集合类，这些类具有不同的存储对象的方法，并提供了相应的方法方便用户对集合进行遍历、添加、删除以及查

找指定的对象。

### 集合类概述

Java语言的java.util包中提供了一些集合类，这些集合类又称为容器。集合中只能存放对象的引用，如果需要存储基本数据类型则需将其转换为对应的包装类。集合的长度是可变的。常用的集合有List集合、Set集合、Map集合。其中List与Set实现了Collection接口，各接口还提供了不同的实现类。

### 集合类接口的常用方法

Collection接口是层次结构中的根接口。Collection 表示一组对象，这些对象也称为 collection 的元素。Collection接口通常不能直接使用，但该接口提供了添加和删除元素、管理数据的方法。由于List接口与Set接口都实现了Collection接口，因此这些方法对List集合与Set集合是通用的。

而Map接口也是一个层次结构中的根接口，其也有很多的继承子接口和实现子类。Map接口中定义了操作集合的许多方法。其子接口和实现类都适用。

#### List接口的常用方法

List接口继承了Collection接口，因此包含Collection中的所有方法，又因为List是列表类型，所以List接口还提供了一些适合于自身的常用方法

|  |  |  |
| --- | --- | --- |
| 方法 | 返回值 | 功能描述 |
| add(int index,Object obj) | void | 用来向集合中的指定索引位置添加对象，集合的索引位置从0开始，其他对象的索引位置相对向后移一位。 |
| allAll(int index,Collection col) | boolean | 向集合的指定索引位置添加指定的集合对象 |
| remove(int index) | Object | 用来移除集合中指定索引位置的对象 |
| get(int index) | Object | 用于获取指定索引位置的对象 |
| indexOf(Object obj) | int | 该方法返回列表中对象第一次出现的索引位置，如果集合中不包含该元素，则返回-1 |
| lastIndexOf(Object obj) | int | 该方法返回列表中对象最后一次出现的索引位置，如果集合中不包含该元素，则返回-1 |
| subList(int formIndex,int toIndex) | List | 获取从索引formIndex到toIndex之间的元素对象 |
| set(int index,E element) | Object | 用指定元素替换列表中指定位置的元素，返回以前在指定位置的元素 |
| listIterator() | ListIteratot | 用来获得一个包含所有对象的listIterator列表迭代器 |

由于List集合是以线性方式存储对象的，所以可以通过对象的索引来操作对象

#### Set接口的常用方法

Set集合由Set接口和Set接口的实现类组成。Set接口继承了Collection接口，因此包含Collection接口的所有方法。

|  |  |  |
| --- | --- | --- |
| 方法 | 返回值 | 功能描述 |
| add(Object obj) | boolean | 如果此Set集合中尚未存在指定的元素，则添加此元素 |
| addAll(Collection coll) | boolean | 将参数集合中所有元素添加到此Set集合的尾部 |
| remove(Object obj) | boolean | 将指定的参数对象移除集合 |
| retainAll(Collection c) | boolean | 只保存Set集合中包含在指定Collection集合中的内容 |
| removeAll(Collection c) | boolean | 在Set集合中移除包含在指定Collection中的元素 |
| clear() | void | 移除此Set中的所有元素 |
| iterator() | Iterator | 返回此Set中的元素上进行迭代的迭代器 |
| size() | int | 返回此Set集合中的所有元素数 |
| isEmpty() | boolean | 如果Set不包含元素，则返回true |

Set集合在存放数据的时候会调用该数据的equals()方法比较集合中是否有相同的数据。

Set集合中的数据如果重写了equals()和hashCode()，则不允许放重复数据，如果没有重写这两个方法，则允许放重复数据（虽然对象的属性值都相等，但是其对象地址不相等。equals()方法默认是比较地址是否相等，重写后才是比较的值而非地址）。

#### Map接口的常用方法

Map接口提供了将键映射到值的对象。一个映射不能包含重复的键，每个键最多只能映射到一个值。Map接口中同样提供了集合的常用方法

|  |  |  |
| --- | --- | --- |
| 方法 | 返回值 | 功能描述 |
| put(key k,value v) | Object | 向集合中添加指定的key与value的映射关系 |
| containskey(Object key) | boolean | 如果此映射包含指定键的映射关系，则返回true |
| containsValue(Object value) | boolean | 如果此映射将一个或多个键映射到指定值，则返回true |
| get(Object key) | Object | 如果存在指定的键对象，则返回该对象对应的值，否则返回null |
| keySet() | Set | 返回该集合中的所有键对象组成的Set集合 |
| values() | Collection | 返回该集合中所有值对象形成的Collection集合 |
| clear() | void | 移除此Map中的所有映射关系 |
| isEmpty() | boolean | 如果此Map中不包含指定键的映射关系，则返回true |
| size() | int | 返回此Map集合中的所有映射关系数 |

由于Map集合中的元素是通过key、value进行存储的，要获取集合中指定的key值或value值，需要先通过相应的方法获取key集合或value集合，再遍历key集合或value集合获取指定值。

### 集合类接口的实现类

#### List接口的实现类

要使用List集合，通常情况下需要声明为List类型，然后通过List接口的实现类来对集合进行实例化。List接口的实现类常用的有ArrayList、LinkedList、Vector。

1. **ArrayList类**

该类实现了可变的数组，底层以数组的数据结构存储对象引用。允许存储重复数据和null值。其存储的数据会按照存入的先后顺序存储。该集合查询速度快，增删速度慢且线程不安全。

语法格式：

List<String> list = **new** ArrayList<String>();

1. **LinkedList类**

该类采用链表数据结构保存对象。允许存储重复数据和null值。其存储的数据会按照存入的先后顺序存储。该集合增删速度快，查询速度慢且线程不安全。

语法格式：

List<String> list = **new** LinkedList<String>();

1. **Vector类**

该类实现了可变的数组，底层也是以数组的数据结构存储对象引用。允许存储重复数据，但是不允许存储null值。其存储的数据会按照存入的先后顺序存储。该集合整体效率较低，但是是线程安全的。

List<String> list = **new** Vector<String>();

#### Set接口的实现类

要使用Set集合，通常情况下需要声明为Set类型，然后通过Set接口的实现类来实例化。Set接口的实现类常用的有HashSet、TreeSet类

1. **HashSet类**

该类底层是以哈希表（散列）的数据结构存储对象引用。不允许存储重复数据，可以存储null值。其存储的数据的顺序不一定是按照添加的先后顺序存储。该集合线程不安全。该集合是以hashCode()&&equals()来实现判断数值是否重复的。内部是基于HashMap实现。

语法格式：

Set<String> set = **new** HashSet<String>();

1. **TreeSet类**

该类底层是以二叉树的数据结构存储对象引用。不允许存储重复数据，且存储的数据是已经排序后的方式存储的（内部基于TreeMap实现）。该集合线程不安全。

语法格式：

Set<String> set = **new** TreeSet<String>();

首先要明白一点：加入Set里面的元素必须定义equals()方法以确保对象的唯一性。

TreeSet的底层实现是采用红-黑树的数据结构，采用这种结构可以从Set中获取有序的序列，但是前提条件是：元素必须实现Comparable接口，该接口中只用一个方法，就是compareTo()方法。当往Set中插入一个新的元素的时候，首先会遍历Set中已经存在的元素（当然不是采用顺序遍历，具体采用什么方法，建议自己去看看源码），并调用compareTo()方法，根据返回的结果，决定插入位置。进而也就保证了元素的顺序。

上面已经说过，加入Set里面的元素必须定义自己的equals()方法，但是对于良好的设计风格，最好在覆盖equals()方法的同时，也覆盖hashCode()方法，当然，对于TreeSet而言不用覆盖hashCode()方法也可。请记住：覆盖hashCode()方法的目的，只有一个原因就是提高效率。

在往Set中插入新的对象时，首先会用该对象的hashCode()与已经存在对象的hashCode()做比较，如果相等，那就不能插入，如果不等，才会调用equals()方法，如果equals结果为true，说明已经存在，就不能再插入，如果为false，可以插入。

注：如果没有覆盖hashCode()方法，那就是只比较equals().对两个对象equals运算，是判断两个对象是否相等的关键。

**HashSet与TreeSet区别:**

1. TreeSet 是二差树实现的,Treeset中的数据是自动排好序的，不允许放入null值
2. HashSet 是哈希表实现的,HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束
3. HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例

#### Map接口的实现类

Map接口常用的实现类有HashMap、TreeMap和HashTable类。通常建议使用HashMap实现类实现Map集合，因为由HashMap类实现的Map集合对于添加和删除映射关系效率更高。如果希望Map集合中的对象存在一定的顺序，则使用TreeMap类实现Map集合。

1. **HashMap类**

该类基于哈希表（散列）的Map接口的实现，此实现提供所有可选择的映射操作，并允许使用null值和null键，但必须保证键的唯一性。HashMap通过哈希码对其内部的映射关系进行快速查找，但不能保证映射的顺序，特别是不保证该顺序的恒久不变。线程不安全，但是效率高。

语法格式：

Map<Integer, String> map = **new** HashMap<Integer, String>();

1. **TreeMap类**

该类不仅实现了Map接口，还实现了java.util.SortedMap接口，因此集合根据其键的自然顺序进行排序。但在添加、删除和定位映射关系上，TreeMap类比HashMap类的性能要差一些。由于TreeMap类实现的Map集合中的映射关系是根据键对象按照一定的顺序排列的，因此不允许键对象是null，但值可以为null。键不允许重复。该类是以二叉树的数据结构存储对象引用，线程不安全。

语法格式：

Map<Integer, String> map = **new** TreeMap<Integer, String>();

1. **HashTable类**

此类实现一个哈希表，该哈希表将键映射到相应的值。任何非null对象都可以用作键或值并且键不允许重复。为了成功地在哈希表中存储和获取对象，用作键的对象必须实现 hashCode 方法和 equals 方法。该集合线程安全，但是效率低。

语法格式：

Map<Integer, String> map = **new** Hashtable<Integer, String>();

### 集合的操作

#### 添加数据

向集合中添加数据，其中实现List和Set接口的集合类都使用add(E e)方法向集合中添加数据。实现Map接口的集合类则使用put(K key, V value)方法向集合中添加数据。

List、Set接口：

List<String> list = **new** ArrayList<>();

list.add("abc");

Map接口：

Map<Integer, String> map = **new** Hashtable<>();

map.put(1, "abc");

#### 删除数据

删除集合中的数据，其中实现List和Set接口的集合类都使用clear()和remove(Object o)方法，而实现Map接口的集合类中使用clear()和remove(Object key)方法。clear()方法用于删除集合中的所有数据，而remove()方法用于删除给定的数据。

Lset、Set接口：

List<String> list = **new** ArrayList<>();

list.clear(); //删除所有数据

list.remove("abc"); //删除指定数据

Map接口：

Map<Integer, String> map = **new** Hashtable<>();

map.clear(); //删除所有数据

map.remove(1); //删除指定键对应的值

#### 替换数据

替换集合中的数据，其中实现了List接口的集合类可以使用set(int index, E element)方法实现替换集合值。实现Set、Map接口的集合类只能通过删除后再添加的方法来实现数据的替换。

List接口：

List<String> list = **new** ArrayList<>();

list.set(0, "efg"); //替换下标0对应的值

#### 遍历数据

遍历集合数据可以通过以下四种方式来实现：

1. **Iterator**：使用Iterator接口实现一个迭代器。通过接口中定义的方法来实现集合的遍历

List、Set接口：

List<String> list = **new** ArrayList<>();

Iterator<String> iter = list.iterator(); //返回Iterator迭代器

**while** (iter.hasNext()) { //遍历

System.***out***.println(iter.next()); //获取值

}

Map接口：

Map<Integer, String> map = **new** Hashtable<>();

//调用entrySet()方法返回Entry类型的Set集合

Set<Entry<Integer, String>> set = map.entrySet();

//调用Set集合的iterator()方法返回Iterator类型的对象

Iterator<Entry<Integer, String>> iterator = set.iterator();

**while**(iterator.hasNext()){

//遍历获取每个Entry

Entry<Integer, String> entry = iterator.next();

System.***out***.println(entry.getKey() + ":" + entry.getValue());

}

1. **ListIterator**：是Iterator的子接口，专门用于输出List中的内容。

List接口：

List<String> list = **new** ArrayList<>();

//也可以Iterator<String> iter = list.listIterator();

ListIterator<String> iter = list.listIterator();

**while**(iter.hasNext()){

System.***out***.println(iter.next());

}

1. **foreach**遍历：JDK1.5之后提供的新功能，可以输出数组或集合。

List、Set接口：

List<String> list = **new** ArrayList<>();

**for** (String str : list) {

System.***out***.println(str);

}

Map接口：

Map<Integer, String> map = **new** Hashtable<>();

//调用Map集合的keySet()方法，返回所有键的Set集合

Set<Integer> set = map.keySet();

**for** (Integer integer : set) {

System.***out***.println(map.get(integer));

}

1. **for**循环

List接口：

List<String> list = **new** ArrayList<>();

**for** (**int** i = 0; i < list.size(); i++) {

System.***out***.println(list.get(i));

}

Set接口：

Set<String> set = **new** HashSet<>();

//将集合转换为数组，因为返回的是一个Object[]数组，所以需要向下转型

String[] strs = (String[]) set.toArray();

**for** (**int** i = 0; i < strs.length; i++) {

System.***out***.println(strs[i]);

}

Map接口：

Map<Integer, String> map = **new** Hashtable<>();

//调用Map集合的keySet()方法，返回所有键的Set集合

Set<Integer> set = map.keySet();

Integer[] in = (Integer[]) set.toArray();

**for** (**int** i = 0; i < in.length; i++) {

System.***out***.println(map.get(in[i]));

}

#### 数据排序

1. **Collections类**

java集合的工具类java.util.Collections类中提供了sort()静态方法用于实现对集合数据的排序，该方法重载实现了两种排序方式：

**Collections.****sort(List list)**

该方法根据元素的“自然顺序”对指定列表按升序进行排序。列表中的所有元素都必须实现Comparable接口，并实现compareTo()方法。此排序方法具有稳定性：不会因调用sort 方法而对相等的元素进行重新排序。

**java.lang.Comparable<T>**接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的compareTo方法被称为它的自然比较方法。

实现此接口的对象列表（和数组）可以通过 Collections.sort（和 Arrays.sort）进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器（public interface Comparator<T>）。

compareTo(T o)：比较此对象与指定对象的顺序。

该方法返回int类型的值（负数、零还是正数）。

实现Comparable接口

**public** **class** Student **implements** Comparable<Student> {

**private** String name;

**public** Student(){}

**public** Student(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **int** compareTo(Student o) {

**if**(**this**.name.hashCode() > o.getName().hashCode()){

**return** 1;

}**else** **if**(**this**.name.hashCode() < o.getName().hashCode()){

**return** -1;

}**else**{

**return** 0;

}

}

}

调用Collections类的sort(List list)静态方法对集合进行排序

**public** **class** StudentTest {

**public** **static** **void** main(String[] args) {

Student s1 = **new** Student("王");

Student s2 = **new** Student("赵");

Student s3 = **new** Student("李");

List<Student> list = **new** ArrayList<>();

list.add(s1);

list.add(s2);

list.add(s3);

Collections.*sort*(list);

**if**(!list.isEmpty()){

**for** (Student student : list) {

System.***out***.println(student.getName());

}

}

}

}

备注：排序方法适用于所有的集合或数组排序。(其中TreeSet和TreeMap集合默认实现了Comparable接口。在JDK类库中有一部分类实现了Comparable接口，如：Integer、Double和String等。)

**Collections.sort(List list,Comparator c)**

该方法根据给定的比较器（Comparator）产生的顺序对指定列表进行排序。此列表内的所有元素都必须可使用指定比较器相互比较。列表中所有需要进行排序的元素必须实现Comparator接口，并实现接口中的compare()方法。此排序被保证是稳定的：不会因调用 sort 而对相等的元素进行重新排序。

1. **Comparator接口**

java.util.Comparator<T>接口强行对某个对象Collection进行整体排序的比较函数。可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort），从而允许在排序顺序上实现精确控制。还可以使用 Comparator 来控制某些数据结构（如有序 set或有序映射）的顺序，或者为那些没有自然顺序的对象Collection提供排序。

compare(T o1, T o2)：比较用来排序的两个参数。根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数。

实例：

1.封装一个Student类

**public** **class** Student {

**private** String name;

**public** Student(){}

**public** Student(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

2.定义一个比较用于实现比较的功能

//定义一个比较类，实现Comparator<>比较接口

**public** **class** Comp **implements** Comparator<Student> {

@Override //重写compare方法

**public** **int** compare(Student o1, Student o2) {

Student a = o1;

Student b = o2;

**if**(a.getName().hashCode() < b.getName().hashCode()){

**return** -1;

}**else** **if**(a.getName().hashCode() > b.getName().hashCode()){

**return** 1;

}**else** {

**return** 0;

}

}

}

3.实现比较

**public** **class** StudentTest {

**public** **static** **void** main(String[] args) {

Student s1 = **new** Student("王");

Student s2 = **new** Student("赵");

Student s3 = **new** Student("李");

List<Student> list = **new** ArrayList<>();

list.add(s1);

list.add(s2);

list.add(s3);

Collections.*sort*(list, **new** Comp());

**if**(!list.isEmpty()){

**for** (Student student : list) {

System.***out***.println(student.getName());

}

}

}

}

### Collections集合工具类

java.util.Collections 是一个包装类。它包含各种有关集合操作的静态方法。此类不能实例化，就像一个工具类，服务于Java的Collection框架。

常用方法：

**public static <T> boolean addAll(Collection<? super T> c, T... elements)**

将所有指定元素添加到指定 Collection 中。可以分别指定要添加的元素，或者将它们指定为一个数组。此便捷方法的行为与 c.addAll(Arrays.asList(elements)) 的行为是相同的，但在大多数实现下，此方法运行起来可能要快得多。c - 要插入 elements 的 Collection、elements - 插入 c 的元素。

**public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)**

使用二分搜索法搜索指定列表，以获得指定对象。在进行此调用之前，必须根据列表元素的自然顺序对列表进行升序排序（通过 sort(List) 方法）。如果没有对列表进行排序，则结果是不确定的。如果列表包含多个等于指定对象的元素，则无法保证找到的是哪一个。list - 要搜索的列表、key - 要搜索的键。

**public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)**

使用二分搜索法搜索指定列表，以获得指定对象。在进行此调用之前，必须根据指定的比较器对列表进行升序排序（通过 sort(List, Comparator) 方法）。如果没有对列表进行排序，则结果是不确定的。如果列表包含多个等于指定对象的元素，则无法保证找到的是哪一个。list - 要搜索的列表、key - 要搜索的键、c - 排序列表的比较器。null 值指示应该使用元素的自然顺序。

**public static <T> void copy(List<? super T> dest, List<? extends T> src)**

将所有元素从一个列表复制到另一个列表。执行此操作后，目标列表中每个已复制元素的索引将等同于源列表中该元素的索引。目标列表的长度至少必须等于源列表。如果目标列表更长一些，也不会影响目标列表中的其余元素。dest - 目标列表、src - 源列表。

**public static boolean disjoint(Collection<?> c1, Collection<?> c2)**

如果两个指定 Collection 中没有相同的元素，则返回 true。

**public static final <T> List<T> emptyList()**

返回空的列表（不可变的）。此列表是可序列化的。

**public static final <K,V> Map<K,V> emptyMap()**

返回空的映射（不可变的）。此映射是可序列化的。

**public static final <T> Set<T> emptySet()**

返回空的 set（不可变的）。此 set 是可序列化的。

**public static <T> void fill(List<? super T> list, T obj)**

使用指定元素替换指定列表中的所有元素。

**public static int frequency(Collection<?> c, Object o)**

返回指定 collection 中等于指定对象的元素数。更确切地讲，返回 collection 中满足 (o == null ? e == null : o.equals(e)) 的 e 元素的数量。

**public static int indexOfSubList(List<?> source, List<?> target)**

返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。source - 在该列表中搜索第一次出现的 target、target - 作为 source 子列表搜索的列表。

**public static int lastIndexOfSubList(List<?> source, List<?> target)**

返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。source - 在该列表中搜索最后一次出现的 target、target - 作为 source 子列表搜索的列表。

**public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)**

根据元素的自然顺序，返回给定 Collection 的最大元素。Collection 中的所有元素都必须实现 Comparable 接口。此外，Collection 中的所有元素都必须是可相互比较的（也就是说，对于 Collection 中的任意 e1 和 e2 元素，e1.compareTo(e2) 不得抛出 ClassCastException）。

**public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)**

根据指定比较器产生的顺序，返回给定 Collection 的最大元素。Collection 中的所有元素都必须可通过指定比较器相互比较（也就是说，对于 Collection 中的任意 e1 和 e2 元素，comp.compare(e1, e2) 不得抛出 ClassCastException）。

**public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)**

根据元素的自然顺序 返回给定Collection 的最小元素。Collection 中的所有元素都必须实现 Comparable 接口。此外，Collection 中的所有元素都必须是可相互比较的（也就是说，对于 Collection 中的任意 e1 和 e2 元素，e1.compareTo(e2) 不得抛出 ClassCastException）。

**public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)**

根据指定比较器产生的顺序，返回给定 Collection 的最小元素。Collection 中的所有元素都必须可通过指定比较器相互比较（也就是说，对于 Collection 中的任意 e1 和 e2 元素，comp.compare(e1, e2) 不得抛出 ClassCastException）。

**public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)**

使用另一个值替换列表中出现的所有某一指定值。更确切地讲，使用 newVal 替换 list 中满足 (oldVal==null ? e==null : oldVal.equals(e)) 的每个 e 元素。（此方法对列表的大小没有任何影响。）

**public static void reverse(List<?> list)**

反转指定列表中元素的顺序。

**public static <T extends Comparable<? super T>> void sort(List<T> list)**

根据元素的自然顺序 对指定列表按升序进行排序。列表中的所有元素都必须实现 Comparable 接口。此外，列表中的所有元素都必须是可相互比较的（也就是说，对于列表中的任何 e1 和 e2 元素，e1.compareTo(e2) 不得抛出 ClassCastException）。

**public static <T> void sort(List<T> list, Comparator<? super T> c)**

根据指定比较器产生的顺序对指定列表进行排序。此列表内的所有元素都必须可使用指定比较器相互比较（也就是说，对于列表中的任意 e1 和 e2 元素，c.compare(e1, e2) 不得抛出 ClassCastException）。

## Java枚举

枚举（enum），是指一个经过排序的、被打包成一个单一实体的项列表。一个枚举的实例可以使用枚举项列表中任意单一项的值。枚举在各个语言当中都有着广泛的应用，通常用来表示诸如颜色、方式、类别、状态等等数目有限、形式离散、表达又极为明确的量。Java从JDK5开始，引入了对枚举的支持。

### 为什么使用枚举

在枚举出现之前，如果想要表示一组特定的离散值，往往使用一些常量。例如：

**public** **class** Entity {

**public** **static** **final** **int** ***VIDEO*** = 1;

**public** **static** **final** **int** ***AUDIO*** = 2;

**public** **static** **final** **int** ***TEXT*** = 3;

**public** **static** **final** **int** ***IMAGE*** = 4;

**private** **int** id;

**private** **int** type;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **int** getType() {

**return** type;

}

**public** **void** setType(**int** type) {

**this**.type = type;

}

}

当然，常量也不仅仅局限于int型，诸如char和String等也是不在少数。然而，无论使用什么样的类型，这样做都有很多的坏处。这些常量通常都是连续、有无穷多个值的量，而类似这种表示类别的量则是离散的，并且通常情况下只有有限个值。用连续的量去表示离散量，会产生很多问题。例如，针对上述的Entity类，如果要对Entity对象的type属性进行赋值，一般会采用如下方法：

**public** **static** **void** main(String[] args) {

Entity e = **new** Entity();

e.setId(10);

e.setType(2);

}

这样做的缺点有：

1）代码可读性差、易用性低。

由于setType()方法的参数是int型的，在阅读代码的时候往往会让读者感到一头雾水，根本不明白这个2到底是什么意思，代表的是什么类型。当然，要保证可读性，还有这样一个办法：

Entity e = **new** Entity();

e.setType(Entity.***AUDIO***);

而这样的话，问题又来了。这样做，客户端必须对这些常量去建立理解，才能了解如何去使用这个东西。说白了，在调用的时候，如果用户不到Entity类中去看看，还真不知道这个参数应该怎么传、怎么调。像是setType(2)这种用法也是在所难免，因为它完全合法，不是每个人都能够建立起用常量名代替数值，从而增加程序可读性、降低耦合性的意识。

2）类型不安全。

在用户去调用的时候，必须保证类型完全一致，同时取值范围也要正确。像是setType(-1)这样的调用是合法的，但它并不合理，今后会为程序带来种种问题。也许你会说，加一个有效性验证嘛，但是，这样做的话，又会引出下面的第(3)个问题。

3）耦合性高，扩展性差。

假如，因为某些原因，需要修改Entity类中常量的值，那么，所有用到这些常量的代码也就都需要修改——当然，要仔细地修改，万一漏了一个，那可不是开玩笑的。同时，这样做也不利于扩展。例如，假如针对类别做了一个有效性验证，如果类别增加了或者有所变动，则有效性验证也需要做对应的修改，不利于后期维护。

枚举就是为了这样的问题而诞生的。它们给出了将一个任意项同另一个项相比较的能力，并且可以在一个已定义项列表中进行迭代。

### 如何使用枚举

创建枚举类型要使用 enum 关键字，隐含了所创建的类型都是 java.lang.Enum 类的子类（java.lang.Enum是一个抽象类，不能手工进行子类定义。）。枚举类型符合通用模式 Class Enum<E extends Enum<E>>，而 E 表示枚举类型的名称。枚举类型的每一个值都将映射到 protected Enum(String name, int ordinal) 构造函数中，在这里，每个值的名称都被转换成一个字符串，并且序数设置表示了此设置被创建的顺序。

一个简单的枚举可以是这样：

**public** **enum** TypeEnum {

***VIDEO***, ***AUDIO***, ***TEXT***, ***IMAGE***;

}

上面的Entity类就可以改成这样：

**public** **class** Entity {

**private** **int** id;

**private** TypeEnum type;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** TypeEnum getType() {

**return** type;

}

**public** **void** setType(TypeEnum type) {

**this**.type = type;

}

}

在为Entity对象赋值的时候，就可以这样：

**public** **static** **void** main(String[] args) {

Entity e = **new** Entity();

e.setId(10);

e.setType(TypeEnum.***AUDIO***);

}

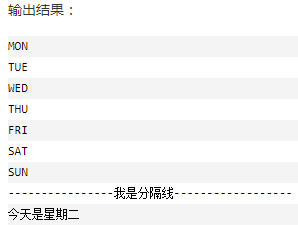
怎么看，都是好了很多。在调用setType()时，可选值只有四个，否则会出现编译错误，因此可以看出，枚举是类型安全的，不会出现取值范围错误的问题。同时，客户端不需要建立对枚举中常量值的了解，使用起来很方便，并且可以容易地对枚举进行修改，而无需修改客户端。如果常量从枚举中被删除了，那么客户端将会失败并且将会收到一个错误消息。枚举中的常量名称可以被打印，因此除了仅仅得到列表中项的序号外还可以获取更多信息。这也意味着常量可用作集合的名称，例如HashMap。

**遍历、switch 等常用操作**

**public** **enum** EnumTest {

***MON***, ***TUE***, ***WED***, ***THU***, ***FRI***, ***SAT***, ***SUN***;

}



**public** **static** **void** main(String[] args) {

**for** (EnumTest e : EnumTest.*values*()) {

System.***out***.println(e.toString());

}

System.***out***.println("----------------------------");

EnumTest test = EnumTest.***TUE***;

**switch**(test){

**case** ***MON***:

System.***out***.println("今天星期一");

**break**;

**case** ***TUE***:

System.***out***.println("今天星期二");

**break**;

//........

**default**:

System.***out***.println(test);

**break**;

}

}

### 枚举的特性

因为在Java中一个枚举就是一个类，它也可以有属性和方法，并且实现接口。只是所有的枚举都继承自java.lang.Enum类，因此enum不可以再继承其他的类、但是可以实现接口。

下面给出在枚举中声明属性和方法的示例：

**public** **enum** TypeEnum {

***VIDEO***(1), ***AUDIO***(2), ***TEXT***(3), ***IMAGE***(4);

**int** value;

TypeEnum(**int** value){

**this**.value = value;

}

**public** **int** getValue(){

**return** value;

}

}

在这个枚举中，每个枚举的值都有一个对应的int型字段，而且不同的枚举值也会有不同的int数值。同时，它和普通的类一样，可以声明构造器和各种各样的方法。

**public** **static** **void** main(String[] args) {

TypeEnum type = TypeEnum.***TEXT***; //type的value属性值为3

System.***out***.println(type.getValue());

}

如果要为每个枚举值指定属性，则在枚举中必须声明一个参数为属性对应类型的构造方法（不能是public）。否则编译器将给出The constructor TypeEnum(int, String) is undefined的错误。在此例中，属性为int型，因此构造方法应当为int型。除此之外，还可以为枚举指定多个属性。

**public** **enum** TypeEnum {

***VIDEO***(1, "视频"), ***AUDIO***(2, "音频"), ***TEXT***(3, "文本"), ***IMAGE***(4, "图像");

**int** value;

String name;

TypeEnum(**int** value, String name){

**this**.value = value;

**this**.name = name;

}

**public** **int** getValue(){

**return** value;

}

**public** String getName(){

**return** name;

}

}

### enum的内置方法

1. **public final int compareTo(E o)**

比较此枚举与指定对象的顺序。在该对象小于、等于或大于指定对象时，分别返回负整数、零或正整数。 枚举常量只能与相同枚举类型的其他枚举常量进行比较。该方法实现的自然顺序就是声明常量的顺序。o - 要比较的对象。

1. **public final Class<E> getDeclaringClass()**

返回与此枚举常量的枚举类型相对应的 Class 对象。当且仅当 e1.getDeclaringClass() == e2.getDeclaringClass() 时，两个枚举常量 e1 和 e2 的枚举类型才相同。（由该方法返回的值不同于由 Object.getClass() 方法返回的值，Object.getClass() 方法用于带有特定常量的类主体的枚举常量。）

1. **public final String name()**

返回此枚举常量的名称，在其枚举声明中对其进行声明。与此方法相比，大多数程序员应该优先考虑使用 toString() 方法，因为 toString 方法返回更加用户友好的名称。该方法主要设计用于特殊情形，其正确性取决于获取正确的名称，其名称不会随版本的改变而改变。

1. **public final int ordinal()**

返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）。 大多数程序员不会使用此方法。它被设计用于复杂的基于枚举的数据结构，比如 EnumSet 和 EnumMap。

1. **public String toString()**

返回枚举常量的名称，它包含在声明中。可以重写此方法，虽然一般来说没有必要。当存在更加“程序员友好的”字符串形式时，应该使用枚举类型重写此方法。

1. **public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)**

返回带指定名称的指定枚举类型的枚举常量。名称必须与在此类型中声明枚举常量所用的标识符完全匹配。（不允许使用额外的空白字符。）enumType - 要从中返回常量的枚举类型的 Class 对象、name - 要返回的常量名称。

1. **public static T[] values()**

返回该枚举的所有值。

### 实例

假设要为该枚举实现一个根据整数值生成枚举值的方法，可以这样做：

**public** **enum** TypeEnum {

***VIDEO***(1, "视频"), ***AUDIO***(2, "音频"), ***TEXT***(3, "文本"), ***IMAGE***(4, "图像");

**int** value;

String name;

TypeEnum(**int** value, String name){

**this**.value = value;

**this**.name = name;

}

**public** **int** getValue(){

**return** value;

}

**public** String getName(){

**return** name;

}

**public** TypeEnum getByValue(**int** value){

**for** (TypeEnum typeEnum : TypeEnum.*values*()) {

**if**(typeEnum.value == value){

**return** typeEnum;

}

}

//定义抛出异常

**throw** **new** IllegalArgumentException("No element matches" + value);

}

}

getByValue(int)即为整数值转枚举值的方法。调用values()方法获取到该枚举下的所有值，然后遍历该枚举下面的每个值和给定的整数是否匹配，若匹配直接返回，若无匹配值则抛出IllegalArgumentException异常，表示参数不合法，兼有有效性验证的作用。

enum 的语法结构尽管和 class 的语法不一样，但是经过编译器编译之后产生的是一个class文件。该class文件经过反编译可以看到实际上是生成了一个类，该类继承了java.lang.Enum<E>。所以，实际上 enum 就是一个 class，只不过 java 编译器帮我们做了语法的解析和编译而已。

综上，我们可以看到，在JDK5中新引入的枚举完美地解决了之前通过常量来表示离散量所带来的问题，大大加强了程序的可读性、易用性和可维护性，并且在此基础之上又进行了扩展，使之可以像类一样去使用，更是为Java对离散量的表示上升了一个台阶。因此，如果在Java中需要表示诸如颜色、方式、类别、状态等等数目有限、形式离散、表达又极为明确的量，应当尽量舍弃常量表示的做法，而将枚举作为首要的选择。

### 枚举集合

EnumSet和EnumMap的应用

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//EnumSet的使用

EnumSet<EnumTest> weekSet = EnumSet.*allOf*(EnumTest.**class**);

**for** (EnumTest day : weekSet) {

System.***out***.println(day);

}

//EnumMap的使用

EnumMap<EnumTest, String> weekMap = **new** EnumMap<>(EnumTest.**class**);

weekMap.put(EnumTest.MON, "星期一");

weekMap.put(EnumTest.TUE, "星期二");

//.....

**for** (Tterator<Entry<EnumTest, String>> iter = weekMap.entrySet().iterator(); iter.hasNext; ) {

Entry<EnumTest, String> entry = iter.next();

System.***out***.println(entry.getKey().name() + ":" + entry.getValue());

}

}

}

## Java异常

异常指不期而至的各种状况，如：文件找不到、网络连接失败、非法参数等。异常是一个事件，它发生在程序运行期间，干扰了正常的指令流程。Java通 过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例，描述了出现在一段编码中的 错误条件。当条件生成时，错误将引发异常。



在 Java 中，所有的异常都有一个共同的祖先 Throwable（可抛出）。Throwable 指定代码中可用异常传播机制通过 Java 应用程序传输的任何问题的共性

Throwable： 有两个重要的子类：Exception（异常）和 Error（错误），二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

Exception（异常）:是程序本身可以处理的异常。

Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

### 异常的分类

#### 编译时异常

在Java中把那些可以预知的错误，例如从文件中读取数据、对数据库进行操作等，在程序编译时就能对程序中可能存在的错误进行处理，并给出具体的错误信息，我们把这些错误称为可控式异常。

|  |  |
| --- | --- |
| 异常 | 说明 |
| IOException | 当发生某种I/O异常时，抛出此异常 |
| SQLException | 提供关于数据库访问错误或其他错误信息的异常 |
| ClassNotFoundException | 类没有找到异常 |
| NoSuchFieldException | 类不包含指定名称的字段时产生的信号 |
| NoSuchMethodException | 无法找到某一特定方法时，抛出该异常 |

#### 运行时异常

在Java中有些错误是不能被编译器检测到的，例如，在进行除法运算时，除数为零；试图把一个不是由数字组成的字符串使用Integer类的parseInt()方法转换为整数等，Java的编译器是检测不到的，因而能够正常编译，但是在运行时就会发生异常，我们把这些异常称为运行时异常

|  |  |
| --- | --- |
| 方法 | 说明 |
| IndexOutOfBoundsException | 指示某集合或数组的索引值超出范围时抛出该异常 |
| NullPointerException | 当应用程序试图在需要对象的地方使用null时，抛出该异常 |
| ArithmeticException | 当出现异常的运算条件时，抛出此异常 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数 |
| ClassCastException | 当试图将对象强制转换为不是实例的子类时，抛出该异常 |

我们在编写代码的时候尽量避免出现运行时异常，将代码写的健壮写。

### 处理异常

在Java语言中当程序发生异常时，可以使用try…..catch、try….catch…..finally、try…..finally进行处理

#### 使用try….catch处理异常

对于程序中可能发生异常的语句，可以将其添加到try…catch语句块中，这样当程序发生异常时，就可以对其进行相应的处理 。如果我们对可能发生异常的代码不进行任何的处理，当发生异常时jvm会抛出产生异常的说明并终止后面代码的运行。

语法格式：

**try**{

//可能发生异常的程序代码

}**catch**(type1 id1){

//捕获并处理try抛出的异常类型type1

}**catch**(type2 id2){

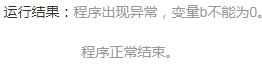
//捕获并处理try抛出的异常类型type2

}

关键词try后的一对大括号将一块可能发生异常的代码包起来，称为监控区域。Java方法在运行过程中出现异常，则创建异常对象。将异常抛出监控区域之 外，由Java运行时系统试图寻找匹配的catch子句以捕获异常。若有匹配的catch子句，则运行其异常处理代码，try-catch语句结束。

匹配的原则是：如果抛出的异常对象属于catch子句的异常类，或者属于该异常类的子类，则认为生成的异常对象与catch块捕获的异常类型相匹配。

1. 捕捉throw语句抛出的“除数为0”异常。



**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int** a = 6;

**int** b = 0;

**try**{

**if**(b == 0){

**throw** **new** ArithmeticException(); //通过throw语句抛出异常

}

System.***out***.println("a/b的值是：" + a / b);

}**catch**(ArithmeticException e){ //捕获异常

System.***out***.println("程序出现异常，变量b不能为0。"); //处理异常

}

System.***out***.println("程序正常结束。");

}

}

1. 捕捉运行时系统自动抛出“除数为0”引发的ArithmeticException异常。



**public** **static** **void** main(String[] args) {

**int** a = 6;

**int** b = 0;

**try**{

System.***out***.println("a/b的值是：" + a / b);

}**catch**(ArithmeticException e){ //捕获异常

System.***out***.println("程序出现异常，变量b不能为0。"); //处理异常

}

System.***out***.println("程序正常结束。");

}

在运行中出现“除数为0”错误，引发ArithmeticException异常。运行时系统创建异常对象并抛出监控区域，转而匹配合适的异常处理器catch，并执行相应的异常处理代码。

#### 使用try…catch….finally处理异常

对于程序中可能发生异常的语句，可以将其添加到try…catch….finally语句块中，这样当程序发生异常时，就可以对其进行相应的处理。其中finally代码块中的无论是否出现异常都会去执行其中的代码。

语法格式：

**try**{

//可能发生异常的程序代码

}**catch**(type id){

//捕获并处理try抛出的异常类型type

}**finally**{

//无论是否发生异常，都会执行的语句块

}

try、catch和finally是进行异常处理的关键字。

try和catch之间的两个大括号内是程序需要正常执行但有可能发生异常的语句。

catch后的两个小括号内是程序需要处理的异常类型。

catch后的两个大括号内是对程序发生的异常进行处理的语句。

finally后的两个大括号内的语句，不管程序是否发生异常都要执行（也就是说程序执行完try和catch之间的语句或执行完catch后两个大括号内的语句都将执行finally后的语句），因此finally语句块通常用于执行垃圾回收、释放资源等操作。

在Java中进行异常处理时，应该尽量使用finally块进行资源回收，因为在try…catch…finally语句块中，不管程序是否发生常，最终都会执行finally语句块，因此可以在finally块中添加释放资源的代码。

在以下4种特殊情况下，finally块不会被执行：

1. 在finally语句块中发生了异常。
2. 在前面的代码中用了System.exit()退出程序。
3. 程序所在的线程死亡。
4. 关闭CPU。

#### 使用try…finally处理异常

对于程序中可能发生异常的语句，可以将其添加到try….finally语句块中，这样当程序发生异常时，就可以在finally语句块中对其进行相应的处理，另外当程序没有发生异常时，执行完try和finally之间的语句后，也将执行finally语句块中的代码，因此可以在finally语句块中放置一些必须执行的代码，如释放内存资源的代码等。

语法格式：

**try**{

//可能发生异常的程序代码

} **finally**{

//无论是否发生异常，都会执行的语句块

}

try和finally是进行异常处理的关键字。

try和finally之间的两个大括号内是程序需要正常执行但又可能发生异常的语句。

finally后两个大括号内的语句是不管程序是否发生异常最终都要执行的语句，因此finally语句块通常用于放置程序中必须执行的代码，如关闭数据库连接、关闭IO流等。

#### try、catch、finally语句块的执行顺序

1. 当try没有捕获到异常时：try语句块中的语句逐一被执行，程序将跳过catch语句块，执行finally语句块和其后的语句；
2. 当try捕获到异常，catch语句块里没有处理此异常的情况：当try语句块里的某条语句出现异常时，而没有处理此异常的catch语句块时，此异常将会抛给JVM处理，finally语句块里的语句还是会被执行，但finally语句块后的语句不会被执行；
3. 当try捕获到异常，catch语句块里有处理此异常的情况：在try语句块中是按照顺序来执行的，当执行到某一条语句出现异常时，程序将跳到catch语句块，并与catch语句块逐一匹配，找到与之对应的处理程序，其他的catch语句块将不会被执行，而try语句块中，出现异常之后的语句也不会被执行，catch语句块执行完后，执行finally语句块里的语句，最后执行finally语句块后的语句；
4. 当发生异常时，如果catch块中包含了return语句，则代码在执行时会先将finally代码块中的语句执行完毕之后再去执行catch块中的return语句。

### 抛出异常

对于程序中发生的异常，除了可以使用try…catch语句块处理之外，还可以使用throws声明或throw语句抛出异常。

#### 使用throws声明抛出异常

throws通常用于方法声明，当方法中可能存在异常，却不想在该方法中对异常进行处理时，就可以在声明方法时使用throws

声明抛出的异常，然后在调用该方法的其他方法中对异常进行处理（如使用try…catch语句或使用throws声明抛出的异常）

如果需要使用throws声明抛出多个异常，各异常之间要用逗号分隔。

语法格式：

数据类型 方法名(参数列表) **throws** 异常类1, 异常类2,..{

方法体;

}

数据类型是基本数据类型或对象类型。

方法名是Java语言的合法标识符。

throws是抛出异常的关键字

异常类是Java语句的异常类或自定义异常类。

方法体是该方法需要执行的语句。

**public** **class** Test {

**public** **static** **void** pop() **throws** NegativeArraySizeException{

//定义方法并抛出NegativeArraySizeException异常类

**int**[] arr = **new** **int**[-1]; //创建数组

}

**public** **static** **void** main(String[] args) {

**try**{ //try语句处理异常信息

*pop*(); //调用方法

}**catch**(NegativeArraySizeException e){

System.***out***.println("pop()方法抛出的异常");

}

}

}

使用throws关键字将异常抛给调用者后，如果调用者不想处理该异常，可以继续向上抛出，但最终要有能够处理该异常的调用者。

**Throws抛出异常的规则：**

1. 如果是不可查异常（unchecked exception），即Error、RuntimeException或它们的子类，那么可以不使用throws关键字来声明要抛出的异常，编译仍能顺利通过，但在运行时会被系统抛出。
2. 必须声明方法可抛出的任何可查异常（checked exception）。即如果一个方法可能出现受可查异常，要么用try-catch语句捕获，要么用throws子句声明将它抛出，否则会导致编译错误
3. 仅当抛出了异常，该方法的调用者才必须处理或者重新抛出该异常。当方法的调用者无力处理该异常的时候，应该继续抛出，而不是囫囵吞枣。
4. 调用方法必须遵循任何可查异常的处理和声明规则。若覆盖一个方法，则不能声明与覆盖方法不同的异常。声明的任何异常必须是被覆盖方法所声明异常的同类或子类。

#### 使用throw语句抛出异常

throw总是出现在函数体中，用来抛出一个Throwable类型的异常。程序会在throw语句后立即终止，它后面的语句执行不到，然后在包含它的所有try块中（可能在上层调用函数中）从里向外寻找含有与其匹配的catch子句的try块。

我们知道，异常是异常类的实例对象，我们可以创建异常类的实例对象通过throw语句抛出。

语法格式：

**throw** **new** Exception("对异常的说明");

throw是抛出异常的关键字

Exception是异常类（通常使用自定义异常类）

**public** **class** Test {

**public** **static** **int** quotient(**int** x, **int** y) **throws** MyException{

//定义方法并抛出MyException自定义的异常类

**if**(y < 0){

**throw** **new** MyException("除数不能是负数"); //异常信息

}

**return** x / y;

}

**public** **static** **void** main(String[] args) {

**int** a = 3;

**int** b = 0;

**try**{ //try语句包含可能发生异常的语句

**int** result = *quotient*(a, b);

}**catch**(MyException e){

System.***out***.println(e.getMessage); //输出异常信息

}**catch**(ArithmeticException e){ //处理ArithmeticException异常

System.***out***.println("除数不能为0"); //输出提示信息

}**catch**(Exception e){ //处理其他异常

System.***out***.println("程序发生了其他的异常");

}

}

}

如果抛出了检查异常，则还应该在方法头部使用throws关键字声明方法可能抛出的异常类型。该方法的调用者也必须检查处理抛出的异常。

### 自定义异常类

使用Java内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户还可以自定义异常。

#### 为什么要用自定义用异常，有什么好处

1. 我们在工作的时候，项目是分模块或者分功能开发的 ,基本不会你一个人开发一整个项目，使用自定义异常类就统一了对外异常展示的方式。
2. 有时候我们遇到某些校验或者问题时，需要直接结束掉当前的请求，这时便可以通过抛出自定义异常来结束，如果你项目中使用了SpringMVC比较新的版本的话有控制器增强，可以通过@ControllerAdvice注解写一个控制器增强类来拦截自定义的异常并响应给前端相应的信息（关于springMVC控制器增强的知识有空再和大家分享）。
3. 自定义异常可以在我们项目中某些特殊的业务逻辑时抛出异常，比如"中性".equals(sex)，性别等于中性时我们要抛出异常，而Java是不会有这种异常的。系统中有些错误是符合Java语法的，但不符合我们项目的业务逻辑。
4. 使用自定义异常继承相关的异常来抛出处理后的异常信息可以隐藏底层的异常，这样更安全，异常信息也更加的直观。自定义异常可以抛出我们自己想要抛出的信息，可以通过抛出的信息区分异常发生的位置，根据异常名我们就可以知道哪里有异常，根据异常提示信息进行程序修改。比如空指针异常NullPointException，我们可以抛出信息为“xxx为空”定位异常位置，而不用输出堆栈信息。

#### 自定义异常的缺陷

毋庸置疑，我们不可能期待JVM（Java虚拟机）自动抛出一个自定义异常，也不能够期待JVM会自动处理一个自定义异常。发现异常、抛出异常以及处理异常的工作必须靠编程人员在代码中利用异常处理机制自己完成。这样就相应的增加了一些开发成本和工作量，所以项目没必要的话，也不一定非得要用上自定义异常，要能够自己去权衡。

#### 自定义异常类注意事项

1. 所有异常都必须是 Throwable 的子类。
2. 如果希望写一个检查性异常类，则需要继承 Exception 类。
3. 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

#### 创建自定义异常类

创建自定义的异常类需要继承自Exception类，并提供含有一个String类型形参的构造方法，该形参就是异常的描述信息，可以通过getMessage()方法获得

**public** **class** NewException **extends** Exception {

**public** NewException(String s){

**super**(s);

}

}

上面代码创建了一个自定义异常类NewException，也就是说NewException是自定义异常类的名称，该类继承自Exception类，该类构造方法的形参s是需要传递的异常描述信息，该信息可以通过异常类的getMessage()方法获得。

#### 使用自定义异常类

创建完自定义异常类后，就可以在程序中使用了。使用自定义异常类可以通过throw语句抛出异常，接下来通过实例来说明自定义异常类的使用。

**public** **class** NewException **extends** Exception {

**private** **static** **final** **long** ***serialVersionUID*** = -7329427261042123777L;

**public** NewException(){}

**public** NewException(String str){

**super**(str);

}

}

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\text.txt");

InputStream is = **null**;

**try** {

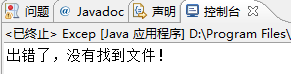
is = **new** FileInputStream(file);

} **catch** (FileNotFoundException e) {

System.***out***.println(**new** NewException("出错了，没有找到文件").getMessage());

}

}



### 异常的使用原则

在程序中使用异常，可以捕获程序中的错误，但是异常的使用也要遵循一定的规则。下面是异常类的几项使用原则：

1. 不要过多地使用异常，这样会增加系统的负担
2. 在方法中使用try…catech语句块捕获异常时，要对异常作出处理。
3. try…catch语句块的范围不要太大，这样不利于对异常的分析。
4. 一个方法被覆盖时，覆盖它的方法必须抛出相同的异常或子异常。

## 输入/输出

### 流概述

流是一组有序的数据序列，根据操作的类型，可分为输入流和输出流两种。I/O(Input/Output)流提供了一条通道程序，可以使用这条通道把源中的字节序列送到目的地。虽然I/O流经常与磁盘文件存取有关，但是程序的源和目的地也可以是键盘、鼠标、内存或显示器窗口等。

Java由数据流处理输入/输出模式，程序从指向源的输入流中读取源中的数据。源可以是文件、网络、压缩包或者其他数据源。

输出流的指向是数据要到达的目的地，程序通过向输出流中写入数据把信息传递到目的地。输出流的目标可以是文件、网络、压缩包、控制台和其他数据输出目标。

java流在处理上分为字符流和字节流。字符流处理的单元为 2 个字节的 Unicode 字符，分别操作字符、字符数组或字符串，而字节流处理单元为 1 个字节，操作字节和字节数组。

Java 内用 Unicode 编码存储字符，字符流处理类负责将外部的其他编码的字符流和 java 内 Unicode 字符流之间的转换。而类 InputStreamReader 和 OutputStreamWriter 处理字符流和字节流的转换。字符流（一次可以处理一个缓冲区）一次操作比字节流（一次一个字节）效率高。

### 输入/输出流

Java语言定义了许多类专门负责各种方式的输入/输出，这些类都被放在java.io包中。其中所有输入流都是抽象类InputStream(字节输入流)或抽象类Reader(字符输入流)的子类；而所有输出流都是抽象类OutputStream(字节输出流)或抽象类Writer(字符输出流)的子类。

#### 输入流

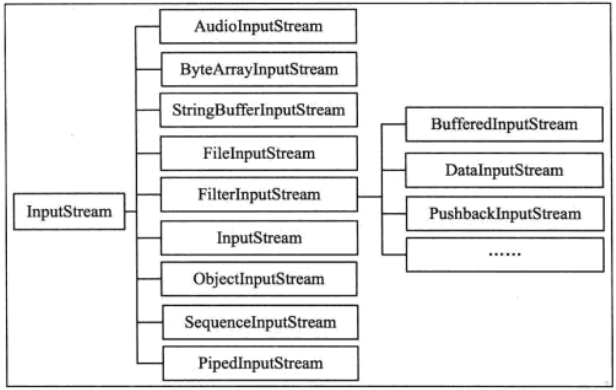
* **字节输入流**

InputStream类是字节输入流的抽象类，是所有字节输入流的父类。该类中所有方法遇到错误时都会引发IOException异常。

InputStream类常用的方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| available() | 返回stream中的可读字节数，inputstream类中的这个方法始终返回的是0，这个方法需要子类去实现。 |
| read() | 从输入流中读取数据的下一个字节。返回0~255范围内的int字节值。如果因为已经到达流末尾而没有可用的字节，则返回值-1 |
| read(byte[] b) | 从输入流中读入一定长度的字节，并以整数的形式返回字节数 |
| mark(int readlimit) | 在输入流的当前位置放置一个标记，readlimit参数告知此输入流在标记位置失效之前允许读取的字节数 |
| reset() | 将输入指针返回到当前所做的标记处 |
| skip(long n) | 跳过输入流上的n个字节并返回实际跳过的字节数 |
| markSupported() | 如果当前流支持mark()/reset()操作就返回true |
| close() | 关闭此输入流并释放与该流关联的所有系统资源，这个是每次在用完流之后必须调用的方法。 |

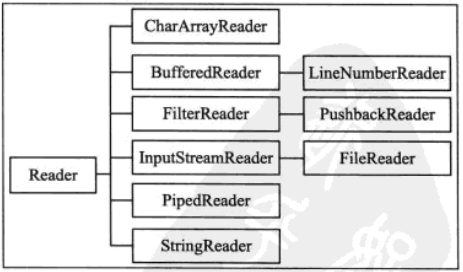
InputStream类的层次结构：



* **字符输入流**

Java中的字符是Unicode编码，是双字节的，InputStream是用来处理字节的，在处理字符文本时不是很方便。Java为字符文本的输入提供了专门一套单独的类Reader，但Reader类并不是InputStream类的替换者，只是在处理字符串时简化了编程。Reader类是字符输入流的抽象类，所有字符输入流的实现都是它的子类。Reader类中的方法与InputStream中方法类似。

Reader类的层次结构：



#### 输出流

* **字节输出流**

OutputStream类是字节输出流的抽象类，此抽象类表示输出字节流的所有类的超类。OutputStream类中的所有方法均返回void，在遇到错误时会引发IOException异常

OutputStream类常用的方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| write(int b) | 将指定的字节写入此输出流 |
| write(byte[] b) | 将b.length个字节从指定的byte数组写入此输出流 |
| write(byte[] b,int off,int len) | 将指定byte数组中从偏移量off开始的len个字节写入此输出流 |
| flush() | 彻底完成输出并清空缓存区 |
| close() | 关闭输出流 |

OutputStream类的层次结构：



* **字符输出流**

Writer类是字符输出流的抽象类，所有字符输出类的实现都是它的子类。

Writer类的层次机构：



### File类

File类是io包中唯一代表磁盘文件本身的对象。File类定义了一些与平台无关的方法来操作文件，可以通过调用File类中的方法，实现创建、删除、重命名文件等。File类的对象主要用来获取文件本身的一些信息，如文件所在的目录、文件的长度、文件读写权限等。数据流可以将数据写入到文件中，而文件也是数据流最常用的数据媒体。

#### 文件的创建与删除

java.io.File类以抽象的方式代表文件名和目录路径名。该类主要用于文件和目录的创建、文件的查找和文件的删除等。File对象代表磁盘中实际存在的文件和目录。通常使用以下3中构造方法来创建文件对象

1. File(String pathname)

该构造方法通过将给定路径名字符串转换为抽象路径名来创建一个新File实例。

语法格式：

**new** File(String pathname);

pathname：是指路径名称（包含文件名称），如果pathname是相对路径，则默认的当前路径在系统属性user.dir中存储。

实例：

File file = **new** File("d:\\text.txt");

1. File(String parent,String child)

该构造方法根据定义的父路径字符串（包含文件名）创建一个新的File对象。

语法格式：

**new** File(String parent, String child);

parent：父路径字符串。

child：子路径字符串

1. File(File f,String child)

该构造方法根据parent抽象路径名和child路径名字符串创建一个新File实例

语法格式：

**new** File(File f, String child);

f：父路径对象

child：子路径字符串

**File类中提供了几个静态属性**：

File.separatorChar(public static final char separatorChar)

与系统有关的默认名称分隔符。此字段被初始化为包含系统属性 file.separator 值的第一个字符。在 UNIX 系统上，此字段的值为 '/'；在 Microsoft Windows 系统上，它为 '\\'。

File.separator(public static final String separator)

与系统有关的默认名称分隔符，为了方便，它被表示为一个字符串。此字符串只包含一个字符，即 separatorChar。

File.pathSeparatorChar(public static final char pathSeparatorChar)

与系统有关的路径分隔符。此字段被初始为包含系统属性 path.separator 值的第一个字符。此字符用于分隔以路径列表 形式给定的文件序列中的文件名。在 UNIX 系统上，此字段为 ':'；在 Microsoft Windows 系统上，它为 ';'。

File.pathSeparator(public static final String pathSeparator)

与系统有关的路径分隔符，为了方便，它被表示为一个字符串。此字符串只包含一个字符，即 pathSeparatorChar。

注意：为了实现代码的跨平台型，可以用这些静态属性来代表路径的分隔符。

1. 使用File类创建一个文件对象

语法格式：

File file = **new** File("d:\\myword", "word.txt");

如果D:/myword目录中没有名称为word的文件，File类对象可通过调用createNewFile()方法创建一个名称为word.txt的文件；如果word.txt文件存在，可以通过文件对象的delete()方法将其删除。

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\myword", "word.txt");

**if**(file.exists()){ //如果文件存在则删除原有的文件

file.delete();

System.***out***.println("文件已删除");

}**else**{

**try** {

file.createNewFile(); //创建文件

System.***out***.println("文件已创建");

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

在Java中文件夹也是一个特殊的文件，如果需要删除文件夹，需先将文件夹下的所有文件删除后才能成功删除文件夹。

#### 常用方法

File类提供了很多方法用于获取文件本身的一些信息

|  |  |
| --- | --- |
| 方法 | 说明 |
| getName() | 返回由此抽象路径名表示的文件或目录的名称。 |
| canRead() | 判断文件是否是可读的 |
| canWrite() | 判断文件是否可被写入 |
| exits() | 测试此抽象路径名表示的文件或目录是否存在。 |
| length() | 获取文件的长度（以字节为单位） |
| getAbsolutePath() | 获取文件的绝对路径 |
| getParent() | 获取文件的父路径 |
| isFile() | 测试此抽象路径名表示的文件是否是一个标准文件。 |
| isDirectory() | 测试此抽象路径名表示的文件是否是一个目录。 |
| isHidden() | 判断文件是否是隐藏文件 |
| lastModified() | 获取文件最后修改时间 |
| createNewFile() | 当且仅当不存在具有此抽象路径名指定名称的文件时，不可分地创建一个新的空文件。 |
| delete() | 删除此抽象路径名表示的文件或目录。 |
| mkdir() | 创建此抽象路径名指定的目录。 |
| mkdirs() | 创建此抽象路径名指定的目录，包括所有必需但不存在的父目录。 |
| public File[] listFiles() | 返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。 |

#### 遍历所有的文件

使用递归遍历出所有的文件

/\*\*

\* 使用递归调用遍历出所有的文件

\* **@param** rootDir

\*/

**public** **void** getAllFile(String rootDir){

File file = **new** File(rootDir);

**if**(file.isDirectory()){ //判断是否是文件

File[] folders = file.listFiles(); //获取文件下的所有文件及文件夹

**for** (File f : folders) {

**if**(f.isDirectory()){ //如果是文件夹则调用自身

getAllFile(f.getAbsolutePath());

}**else**{ //如果是文件则输出文件名称

System.***out***.println(f.getName());

}

}

}

}

### 文件输入/输出流

程序运行期间，大部分数据都是在内存中进行操作，当程序结束或关闭时，这些数据将消失。如果需要将数据永久保存，可使用文件输入/输出流与指定的文件建立连接，将需要的数据永久保存到文件中。

#### FileInputStream与FileOutputStream类

FileInputStream类与FileOutputStream类都是用来操作磁盘文件的。如果用户的文件读取需求比较简单，则可以使用FileInputStream类，该类继承自InputStream类。FileOutputStream类与FileInputStream类对应，提供了基本的文件写入能力。FileOutputStream类是OutoputStream类的子类。

FileInputStream类常用的构造方法如下：

1. FileInputStream(String name)

通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的路径名 name 指定。

1. FileInputStream(File file)

通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的 File 对象 file 指定。

**常用方法：**

**public abstract int read() throws IOException**

读取一个字节并以整数的形式返回（0~255），如果返回-1表示到达输入流的结尾

**public int read(byte[] buffer) throws IOException**

读取一系列字节并存储到一个数组buffer，返回实际读取的字节数，到末尾返回-1

**public long skip(long n) throws IOException**

跳过n个字节不读，返回实际跳过的字节数

**public void close( ) throws IOException**

关闭流释放内存资源

FileOutputStream类有与FileInputStream类相同参数的构造方法，创建一个FileOutputStream对象时，可以指定不存在的文件名，但此文件不能是一个已被其他程序打开的文件。

1. FileOutputStream(File file)

创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

1. FileOutputStream(String name)

创建一个向具有指定名称的文件中写入数据的输出文件流。

常用方法：

**public abstract void write(int b) throws IOException**

向输出流中写入一个字节数据，该字节数据为参数b的低8位

**public void write(byte[] data) throws IOException**

将一个字节类型的数组中的数据写入输出流

**public void close( ) throws IOException**

关闭流释放内存资源

**public void flush( ) throws IOException**

将输出流中缓冲的数据全部写出到目的地

**实例：**

以字节流的方式将内容写入文件中：

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\tet.txt");

OutputStream os = **null**;

**try** {

//定义一个字节输出流

**new** FileOutputStream(file);

//定义一个byte数组及字符编码

**byte**[] bytes = **new** String().getBytes("UTF-8");

os.write(bytes); //将字符输入byte[]数组中

os.flush(); //将数组中的内容写文件中

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{ //关闭IO流

**try** {

os.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

以字节流的方式读取文件内容：

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\tet.txt");

InputStream is = **null**;

**try** {

//定义一个字节输出流

**new** FileInputStream(file);

//定义一个byte数组及字符编码

**byte**[] bytes = **new** **byte**[2048];

**int** len = 0; //定义一个遍历用于获取实际读取的字节长度

**while**((len = is.read(bytes)) != -1){ //获取实际读取内容的长度

System.***out***.println(**new** String(bytes, 0, len, "UTF-8"));

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{ //关闭IO流

**try** {

is.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

注意：在字节流去读文件内容的时候，如果读取的内容小于定义的字节数组长度，则数组中的空位会以默认值填充，如果再去读数组的时候也会将默认值读取出来。为了解决这个问题，可以获取读取的实际字节长度来输出实际长度的内容。

#### FileReader类和FileWriter类

使用FileOutputStream类向文件中写入数据与使用FileInputStream类从文件中将内容读出来，存在一点不足，即这两个类都只提供了对字节或字节数组的读取方法。由于汉字在文件中占用两个字节，如果使用字节流，读取不好可能会出现乱码现象。此时采用字符流FileReader或FileWriter类即可避免这种现象。

FileReader、FileWriter字符流对应了FileInputStream、FileOutputStream类。FileReader流顺序地读取文件，只要不关闭流，每次调用read()方法就顺序地读取源中其余的内容，直到源的末尾或流被关闭。

实例：

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\tet.txt");

FileWriter fw = **null**;

**try** {

**new** FileWriter(file);

fw.write("abcd");

fw.flush();

} **catch** (IOException e) {

// **TODO** 自动生成的 catch 块

e.printStackTrace();

}**finally**{

**try** {

fw.close();

} **catch** (IOException e) {

// **TODO** 自动生成的 catch 块

e.printStackTrace();

}

}

}

### 带缓存的输入/输出流

缓存可以说是I/O的一种性能优化。缓存流为I/O流增加了内存缓存区。有了缓存区，使得在流上执行skip()、mark()和reset()方法都成为可能。

#### BufferedInputStream类与BufferedOutputStream类

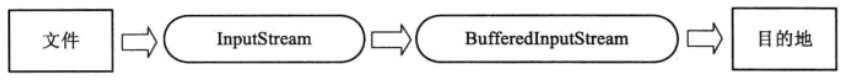
BufferedInputStream类可以对任意的InputStream类进行带缓存区的包装以达到性能的优化。

BufferedInputStream类有两个构造函数：

1. BufferedInputStream(InputStream in)
2. BufferedInputStream(InputStream in,int size)

第一种形式的构造函数创建了一个带有32个字节的缓存流；第二种形式的构造函数按指定的大小来创建缓存区。一个最优的缓存区的大小，取决于它所在的操作系统、可用的内存空间以及机器配置。从构造函数可以看出，BufferedInputStream对象位于InputStream类对象之前

下图秒数了字节数据读取文件的过程：



使用BufferedOutputStream输出信息和向OutputStream输入信息完全一样，只不过BufferedOutputStream有一个flush()方法用来将缓存区的数据强制输出完。BufferedOutputStream类也有两个构造方法：

1. BufferedOutputStream(OutputStream in)
2. BufferedOutputStream(OutputStream in,int size)

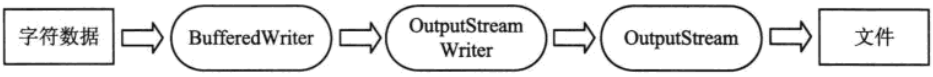
第一种构造函数创建一个32个字节的缓存区，第二种形式以指定的大小来创建缓存区。

flush()方法就是用于即使缓存区么有满的情况下，也将缓存区的内容强制写入到外设。flush()方法只对使用缓存区的OutputStream类的子类有效，当调用close()方法时，系统在关闭流之前，也会将缓存区中信息刷新到磁盘文件中。

#### BufferedReader类与BufferedWriter类

BufferedReader类与BufferedWriter类分别继承Reader类与Writer类。这两个类同样具有内部缓存机制，并可以行为单位进行输入/输出。

下图为字符数据读取文件的过程：



BufferedReader类常用的方法：

read()方法：读取单个字符。

readLine()方法：读取一个文本行，并将其返回为字符串；若无数据可读，则返回null。

write(String s,int off,int len)方法：写入字符串的某一部分。

flush()方法：刷新该流的缓存。

newLine()方法：写入一个行分隔符。

在使用BufferedWriter类的Write()方法时，数据并没有立刻被写入至输出流中，而是首先进入缓存区中。如果想立刻将缓存区中的数据写入输出流中，一定要调用flush()方法

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\tet.txt");

BufferedWriter bw = **null**;

**try** {

**new** BufferedWriter(**new** FileWriter(file));

bw.write("abc");

bw.flush();

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**try** {

bw.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

File file = **new** File("d:\\tet.txt");

BufferedReader br = **null**;

**try** {

**new** BufferedReader(**new** FileReader(file));

String str = "";

**while**((str = br.readLine()) != **null**){

System.***out***.println(str);

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**try** {

br.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

### 数据输入/输出流

数据输入/输出流（DataInputStream类与DataOutputStream类）允许应用程序以与机器无关的方法从底层输入流中读取基本Java数据类型。也就是说，当读取一个数据时，不必再关心这个数值应当是什么字节

DataInputStream类与DataOutputStream类的构造方法：

1. DataInputStream(InputStream in)：使用指定的基础InputStream创建一个DataInputStream。
2. DataOutputStream(OutputStream out)：创建一个新的数据输出流，将数据写入指定基础输出流。

DataOutputStream类提供了3中写入字符串的方法：

1. writeBytes(String s)
2. writeChars(String s)
3. writeUTF(String s)

DataInputStream类只提供了一个readUTF()方法返回字符串。

### 对象序列化和反序列化

#### 什么是序列化

Java序列化是指把Java对象转换为字节序列的过程，用于将对象转换成二进制数据存储到磁盘上或者用于网络传输。而Java反序列化是指把字节序列恢复为Java对象的过程。

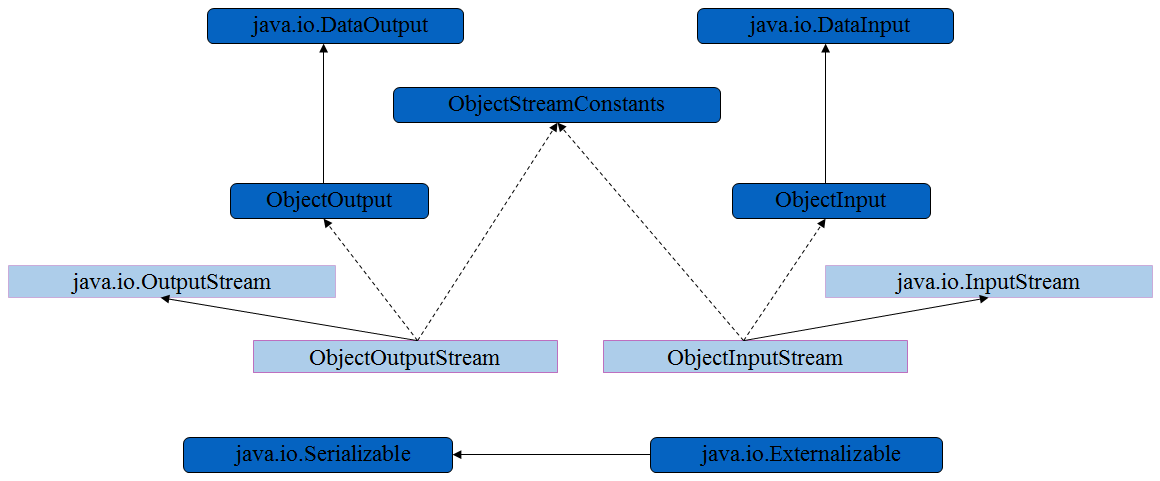
#### 为什么需要序列化与反序列化

我们知道，当两个进程进行远程通信时，可以相互发送各种类型的数据，包括文本、图片、音频、视频等， 而这些数据都会以二进制序列的形式在网络上传送。那么当两个Java进程进行通信时，能否实现进程间的对象传送呢？答案是可以的。如何做到呢？这就需要Java序列化与反序列化了。换句话说，一方面，发送方需要把这个Java对象转换为字节序列，然后在网络上传送；另一方面，接收方需要从字节序列中恢复出Java对象。

当我们明晰了为什么需要Java序列化和反序列化后，我们很自然地会想Java序列化的好处。其好处一是实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上（通常存放在文件里），二是，利用序列化实现远程通信，即在网络上传送对象的字节序列。

#### 实现序列化和反序列化

对象序列化API层次



**序列化**

java.io.ObjectOutputStream类用于实现对象的序列化，该方法提供了一个writeObject(Object obj)方法可以对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。

前提：需要被序列化的类必须实现java.io.Serializable接口，实现该接口后会生成一个serialVersionUID值。

**public** **class** Person **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**实现细节**

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

File file = **new** File("D:\\test.txt"); //建立与文件的链接

OutputStream os = **new** FileOutputStream(file); //实例化一个字节输出流

ObjectOutputStream oos = **new** ObjectOutputStream(os); //实例化一个序列化对象

Person person = **new** Person(); //实例化一个需要序列化的对象

oos.writeObject(person); //实现序列化

oos.flush(); //刷新缓存区

os.close();

oos.close(); //关闭流

}

**反序列化**

注意事项：反序列化时，项目中需要有被序列化相同的包名和类名才能成功的反序列化。

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

File file = **new** File("D:\\test.txt"); //建立与文件的链接

InputStream is = **new** FileInputStream(file); //实例化一个字节输入流

ObjectInputStream ois = **new** ObjectInputStream(is); //实例化一个反序列化对象

Person person = (Person) ois.readObject(); //实现反序列化

is.close();

ois.close(); //关闭流

}

#### 序列化的数据保护

在对象序列化过程中，其所属类的static属性和方法代码不会被序列化处理。除实现Serializable接口生成的serialVersionUID属性除外。

对于个别不希望被序列化的非static属性，可以在属性声明时使用transient关键字进行标明。

**public** **class** Person **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **transient** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

#### 总结：

1. Java序列化就是把对象转换成字节序列，而Java反序列化就是把字节序列还原成Java对象。
2. 采用Java序列化与反序列化技术，一是可以实现数据的持久化，在MVC模式中很是有用；二是可以对象数据的远程通信。

### ZIP压缩输入/输出流

使用ZIP压缩管理文件，是一种十分典型的文件压缩形式，可以节省存储空间。Java的java.util.zip包中提供了非常好用的相关类。其中ZipOutputStream类与ZipInputStream类实现文件的压缩/解压缩。如果要从ZIP压缩管理文件内读取某个文件，要先找到对应该文件的“目录进入点”（从它可知该文件在ZIP文件内的位置），才能读取这个文件的内容。如果要将文件内容写至ZIP文件内，必须先写入对应于该文件的“目录进入点”，并且把要写入文件内容的位置移到此进入点所指的位置，然后再写入文件内容。

Java实现了I/O数据流与网络数据流的单一接口，因此数据的压缩、网络传输和解压缩的实现比较容易。ZipEntry类产生的对象，是用来代表一个ZIP压缩文件内的进入点（entry）。ZipInputStream类用来读取ZIP压缩格式的文件，所支持的包括已压缩机为压缩的进入点（entry）。ZipOutputStream类用来写入ZIP压缩格式的文件，而且所支持的包括已压缩及未压缩的进入点（entry）。

#### 压缩文件

利用ZipOutputStream类对象，可将文件压缩为.zip文件。

ZipOutputStream类的构造函数：

ZipOutputStream(OutputStream out);

ZipOutputStream类的常用方法:

|  |  |
| --- | --- |
| 方法 | 说明 |
| putNextEntry(ZipEntry e) | 开始写一个新的ZipEntry，并且将流内的位置移至此entry所指数据的开头 |
| write(byte[] b,int off,int len) | 将字节数组写入当前ZIP条目数据 |
| finish() | 完成写入ZIP输出流的内容，无须关闭它所配合的OutputStream |
| setComment(String comment) | 可设置此ZIP文件的注释文字 |

#### 解压缩ZIP文件

ZipInputStream类可读取ZIP压缩格式的文件，包括对已压缩和未压缩条目的支持（entry）。

ZipInputStream类的构造函数：

ZipInputStream(InputStream in);

ZipInputStream类的常用方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| read(byte[] b,int off,int len) | 读取目标b数组内off偏移量的位置，长度是len字节 |
| available() | 判断是否已读完目前entry所指定的数据。已读完返回0，否则返回1 |
| closeEntry() | 关闭当前ZIP条目并定位流以读取下一个条目 |
| skip(long n) | 跳过当前ZIP条目中指定的字节数 |
| getNextEntry() | 读取下一个ZipEntry，并将流内的位置移至该entry所指数据的开头 |
| createZipEntry(String name) | 以指定的name参数新建一个ZipEntry对象 |

### java操作xml文件

XML(eXtensible Markup Language，可扩展标记语言)是一种用于存储和传输数据的技术。它现在已经成为一种通用的数据交换格式，平台的无关性，语言的无关性，系统的无关性使得很多场合都需要用到XML，给数据集成与交互带来了极大的方便。

目前Java中用于解析XML的技术很多，主流的有DOM、SAX、JDOM、DOM4j。下面我们分别介绍这4种技术的使用、优缺点及性能。

下面首先给出这四种方法的jar包下载地址

DOM：在现在的Java JDK里都自带了，在xml-apis.jar包里

SAX：http://sourceforge.net/projects/sax/

JDOM：http://jdom.org/downloads/index.html

DOM4J：http://sourceforge.net/projects/dom4j/

#### 介绍及优缺点分析

1. DOM（Document Object Model)

DOM是用与平台和语言无关的方式表示XML文档的官方W3C标准。DOM是以层次结构组织的节点或信息片断的集合。这个层次结构允许开发人员在树中寻找特定信息。分析该结构通常需要加载整个文档和构造层次结构，然后才能做任何工作。由于它是基于信息层次的，因而DOM被认为是基于树或基于对象的。

【优点】

允许应用程序对数据和结构做出更改。

访问是双向的，可以在任何时候在树中上下导航，获取和操作任意部分的数据。

【缺点】

通常需要加载整个XML文档来构造层次结构，消耗资源大。

1. SAX（Simple API for XML)

SAX处理的优点非常类似于流媒体的优点。分析能够立即开始，而不是等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这对于大型文档来说是个巨大的优点。事实上，应用程序甚至不必解析整个文档；它可以在某个条件得到满足时停止解析。一般来说，SAX还比它的替代者DOM快许多。

选择DOM还是选择SAX？ 对于需要自己编写代码来处理XML文档的开发人员来说， 选择DOM还是SAX解析模型是一个非常重要的设计决策。 DOM采用建立树形结构的方式访问XML文档，而SAX采用的是事件模型。

DOM解析器把XML文档转化为一个包含其内容的树，并可以对树进行遍历。用DOM解析模型的优点是编程容易，开发人员只需要调用建树的指令，然后利用navigation APIs访问所需的树节点来完成任务。可以很容易的添加和修改树中的元素。然而由于使用DOM解析器的时候需要处理整个XML文档，所以对性能和内存的要求比较高，尤其是遇到很大的XML文件的时候。由于它的遍历能力，DOM解析器常用于XML文档需要频繁的改变的服务中。

SAX解析器采用了基于事件的模型，它在解析XML文档的时候可以触发一系列的事件，当发现给定的tag的时候，它可以激活一个回调方法，告诉该方法制定的标签已经找到。SAX对内存的要求通常会比较低，因为它让开发人员自己来决定所要处理的tag.特别是当开发人员只需要处理文档中所包含的部分数据时，SAX这种扩展能力得到了更好的体现。但用SAX解析器的时候编码工作会比较困难，而且很难同时访问同一个文档中的多处不同数据。

【优势】

不需要等待所有数据都被处理，分析就能立即开始。

只在读取数据时检查数据，不需要保存在内存中。

可以在某个条件得到满足时停止解析，不必解析整个文档。

效率和性能较高，能解析大于系统内存的文档。

【缺点】

需要应用程序自己负责TAG的处理逻辑（例如维护父/子关系等），文档越复杂程序就越复杂。

单向导航，无法定位文档层次，很难同时访问同一文档的不同部分数据，不支持XPath。

1. JDOM(Java-based Document Object Model)

JDOM的目的是成为Java特定文档模型，它简化与XML的交互并且比使用DOM实现更快。由于是第一个Java特定模型，JDOM一直得到大力推广和促进。正在考虑通过“Java规范请求JSR-102”将它最终用作“Java标准扩展”。从2000年初就已经开始了JDOM开发。

JDOM与DOM主要有两方面不同。首先，JDOM仅使用具体类而不使用接口。这在某些方面简化了API，但是也限制了灵活性。第二，API大量使用了Collections类，简化了那些已经熟悉这些类的Java开发者的使用。

JDOM文档声明其目的是“使用20%（或更少）的精力解决80%（或更多）Java/XML问题”（根据学习曲线假定为20%）。JDOM对于大多数Java/XML应用程序来说当然是有用的，并且大多数开发者发现API比DOM容易理解得多。JDOM还包括对程序行为的相当广泛检查以防止用户做任何在XML中无意义的事。然而，它仍需要您充分理解XML以便做一些超出基本的工作（或者甚至理解某些情况下的错误）。这也许是比学习DOM或JDOM接口都更有意义的工作。

JDOM自身不包含解析器。它通常使用SAX2解析器来解析和验证输入XML文档（尽管它还可以将以前构造的DOM表示作为输入）。它包含一些转换器以将JDOM表示输出成SAX2事件流、DOM模型或XML文本文档。JDOM是在Apache许可证变体下发布的开放源码。

【优点】

使用具体类而不是接口，简化了DOM的API。

大量使用了Java集合类，方便了Java开发人员。

【缺点】

没有较好的灵活性。

性能较差。

1. DOM4J(Document Object Model for Java)

虽然DOM4J代表了完全独立的开发结果，但最初，它是JDOM的一种智能分支。它合并了许多超出基本XML文档表示的功能，包括集成的XPath支持、XML Schema支持以及用于大文档或流化文档的基于事件的处理。它还提供了构建文档表示的选项，它通过DOM4J API和标准DOM接口具有并行访问功能。从2000下半年开始，它就一直处于开发之中。

为支持所有这些功能，DOM4J使用接口和抽象基本类方法。DOM4J大量使用了API中的Collections类，但是在许多情况下，它还提供一些替代方法以允许更好的性能或更直接的编码方法。直接好处是，虽然DOM4J付出了更复杂的API的代价，但是它提供了比JDOM大得多的灵活性。

在添加灵活性、XPath集成和对大文档处理的目标时，DOM4J的目标与JDOM是一样的：针对Java开发者的易用性和直观操作。它还致力于成为比JDOM更完整的解决方案，实现在本质上处理所有Java/XML问题的目标。在完成该目标时，它比JDOM更少强调防止不正确的应用程序行为。

DOM4J是一个非常非常优秀的Java XML API，具有性能优异、功能强大和极端易用使用的特点，同时它也是一个开放源代码的软件。如今你可以看到越来越多的Java软件都在使用DOM4J来读写XML，特别值得一提的是连Sun的JAXM也在用DOM4J.

【优点】

大量使用了Java集合类，方便Java开发人员，同时提供一些提高性能的替代方法。

支持XPath。

有很好的性能。

【缺点】

大量使用了接口，API较为复杂。

**比较**

1. DOM4J性能最好，连Sun的JAXM也在用DOM4J。目前许多开源项目中大量采用DOM4J，例如大名鼎鼎的Hibernate也用DOM4J来读取XML配置文件。如果不考虑可移植性，那就采用DOM4J.
2. JDOM和DOM在性能测试时表现不佳，在测试10M文档时内存溢出，但可移植。在小文档情况下还值得考虑使用DOM和JDOM.虽然JDOM的开发者已经说明他们期望在正式发行版前专注性能问题，但是从性能观点来看，它确实没有值得推荐之处。另外，DOM仍是一个非常好的选择。DOM实现广泛应用于多种编程语言。它还是许多其它与XML相关的标准的基础，因为它正式获得W3C推荐（与基于非标准的Java模型相对），所以在某些类型的项目中可能也需要它（如在JavaScript中使用DOM）。
3. SAX表现较好，这要依赖于它特定的解析方式－事件驱动。一个SAX检测即将到来的XML流，但并没有载入到内存（当然当XML流被读入时，会有部分文档暂时隐藏在内存中）。

如果XML文档较大且不考虑移植性问题建议采用DOM4J；如果XML文档较小则建议采用JDOM；如果需要及时处理而不需要保存数据则考虑SAX。但无论如何，还是那句话：适合自己的才是最好的，如果时间允许，建议大家讲这四种方法都尝试一遍然后选择一种适合自己的即可。

#### DOM

DOM的全称是Document Object Model，即文档对象模型。由W3C提供的接口，它将整个XML文档读入内存，构建一个DOM树来对各个节点（Node）进行操作。通过DOM接口，应用程序可以在任何时候访问XML文档中的任何一部分数据，因此这种利用DOM接口的机制也被称为随机访问机制。DOM接口是访问XML最底层的接口之一，都以此为基础。

**DOM的XML文件读取:**

1. DocumentBuilderFactory.newInstance() 创建DocumentBuilderFactory的对象
2. DocumentBuilder 获得具体的DOM解析器。
3. Parse(new File()) 获取文档xml路径。

示例代码：

准备XML文档：这里我用的是languages.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* standalone=*"no"*?>

<father>

<son id=*"001"*>

<name>老大</name>

<age>20</age>

</son>

<son id=*"002"*>

<name>老二</name>

<age>18</age>

</son>

<son id=*"003"*>

<name>老三</name>

<age>13</age>

</son>

</father>

DOM操作XML文件的例子

**import** java.io.File;

**import** javax.xml.parsers.DocumentBuilder;

**import** javax.xml.parsers.DocumentBuilderFactory;

**import** javax.xml.transform.Transformer;

**import** javax.xml.transform.TransformerFactory;

**import** javax.xml.transform.dom.DOMSource;

**import** javax.xml.transform.stream.StreamResult;

**import** javax.xml.xpath.XPath;

**import** javax.xml.xpath.XPathConstants;

**import** javax.xml.xpath.XPathExpressionException;

**import** javax.xml.xpath.XPathFactory;

**import** org.w3c.dom.Document;

**import** org.w3c.dom.Element;

**import** org.w3c.dom.Node;

**import** org.w3c.dom.NodeList;

**public** **class** XMLWriter {

**private** **static** String *xmlPath* = "src\\com\\zc\\homeWork18\\MyXml.xml";

**public** **static** **void** getFamilyMemebers() {

\*//创建文件工厂实例

DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();

// 如果创建的解析器在解析XML文档时必须删除元素内容中的空格，则为true，否则为false

dbf.setIgnoringElementContentWhitespace(**true**);

**try** {

\*//创建文件对象

DocumentBuilder db = dbf.newDocumentBuilder();// 创建解析器，解析XML文档

Document doc = db.parse(*xmlPath*); // 使用dom解析xml文件

//历遍列表，进行XML文件的数据提取

// 根据节点名称来获取所有相关的节点

NodeList sonlist = doc.getElementsByTagName("son");

**for** (**int** i = 0; i < sonlist.getLength(); i++) // 循环处理对象

{

// 节点属性的处理

Element son = (Element) sonlist.item(i);

// 循环节点son内的所有子节点

**for** (Node node = son.getFirstChild(); node != **null**; node = node

.getNextSibling()) {

// 判断是否为元素节点

**if** (node.getNodeType() == Node.***ELEMENT\_NODE***) {

String name = node.getNodeName();

String value = node.getFirstChild().getNodeValue();

System.***out***.println(name + " : " + value);

}

}

}

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

}

// 修改

**public** **static** **void** modifySon() {

// 创建文件工厂实例

DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();

dbf.setIgnoringElementContentWhitespace(**true**);

**try** {

// 从XML文档中获取DOM文档实例

DocumentBuilder db = dbf.newDocumentBuilder();

// 获取Document对象

Document xmldoc = db.parse(*xmlPath*);

// 获取根节点

Element root = xmldoc.getDocumentElement();

// 定位id为001的节点

Element per = (Element) *selectSingleNode*("/father/son[@id='001']",

root);

// 将age节点的内容更改为28

per.getElementsByTagName("age").item(0).setTextContent("28");

// 保存

TransformerFactory factory = TransformerFactory.*newInstance*();

Transformer former = factory.newTransformer();

former.transform(**new** DOMSource(xmldoc), **new** StreamResult(**new** File(

*xmlPath*)));

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

}

// 获取目标节点，进行删除，最后保存

**public** **static** **void** discardSon() {

DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();

dbf.setIgnoringElementContentWhitespace(**true**);

**try** {

DocumentBuilder db = dbf.newDocumentBuilder();

Document xmldoc = db.parse(*xmlPath*);

// 获取根节点

Element root = xmldoc.getDocumentElement();

// 定位根节点中的id=002的节点

Element son = (Element) *selectSingleNode*("/father/son[@id='002']", root);

// 删除该节点

root.removeChild(son);

// 保存

TransformerFactory factory = TransformerFactory.*newInstance*();

Transformer former = factory.newTransformer();

former.transform(**new** DOMSource(xmldoc), **new** StreamResult(**new** File(*xmlPath*)));

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

}

// 新增节点

**public** **static** **void** createSon() {

// 创建文件工厂实例

DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();

dbf.setIgnoringElementContentWhitespace(**false**);

**try** {

DocumentBuilder db = dbf.newDocumentBuilder();

// 创建Document对象

Document xmldoc = db.parse(*xmlPath*);

// 获取根节点

Element root = xmldoc.getDocumentElement();

// 创建节点son，设置对应的id为004

Element son = xmldoc.createElement("son");

son.setAttribute("id", "004");

// 创建节点name

Element name = xmldoc.createElement("name");

name.setTextContent("小儿子");

son.appendChild(name);

// 创建节点age

Element age = xmldoc.createElement("age");

age.setTextContent("0");

son.appendChild(age);

// 把son添加到根节点中

root.appendChild(son);

// 保存

TransformerFactory factory = TransformerFactory.*newInstance*();

Transformer former = factory.newTransformer();

former.transform(**new** DOMSource(xmldoc), **new** StreamResult(**new** File(

*xmlPath*)));

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

}

// 修改节点信息

**public** **static** Node selectSingleNode(String express, Element source) {

Node result = **null**;

//创建XPath工厂

XPathFactory xpathFactory = XPathFactory.*newInstance*();

//创建XPath对象

XPath xpath = xpathFactory.newXPath();

**try** {

result = (Node) xpath.evaluate(express, source, XPathConstants.***NODE***);

System.***out***.println(result);

} **catch** (XPathExpressionException e) {

System.***out***.println(e.getMessage());

}

**return** result;

}

// 打印

**public** **static** **void** main(String[] args) {

*getFamilyMemebers*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

*modifySon*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

System.***out***.println("修改数据");

*getFamilyMemebers*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

*discardSon*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

System.***out***.println("删除数据");

*getFamilyMemebers*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

*createSon*();

System.***out***.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

System.***out***.println("添加数据");

*getFamilyMemebers*();

}

}

#### DOM4J

dom4j是一个Java的XML API，类似于jdom，用来读写XML文件的，dom4j是一个十分优秀的JavaXML API，具有性能优异、功能强大和极其易使用的特点，同时它也是一个开放源代码的软件，可以在SourceForge上找到它。在IBM developerWorks上面还可以找到一篇文章，对主流的Java XML API进行的性能、功能和易用性的评测，所以可以知道dom4j无论在哪个方面都是非常出色的。如今可以看到越来越多的Java软件都在使用dom4j来读写XML，特别值得一提的是连Sun的JAXM也在用dom4j。这已经是必须使用的jar包， Hibernate也用它来读写配置文件。

下载：dom4j架包！

book.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<books>

<book id=*"book1"*>

<title>JAVA编程思想</title>

<price>80.00</price>

</book>

<book id=*"book2"*>

<title>JAVA 编程理论</title>

<price>100.00</price>

</book>

</books>

Book.java

**public** **class** Book {

**public** String title;

**public** **double** price;

**public** String id;

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** String toString() {

**return** "图书ISBN为：" + id + " 书名为：" + title + " 价格为：" + price;

}

}

DOM4j.java

**import** java.io.FileWriter;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.dom4j.Document;

**import** org.dom4j.Element;

**import** org.dom4j.io.SAXReader;

**import** org.dom4j.io.XMLWriter;

**import** com.zc.homeWork19.Book;

**public** **class** DOM4j {

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

/\*\*

\* 第一步，得到document对象。

\*/

Document document = *getDocument*();

/\*\*

\* 第二步，修改得到的document对象

\*/

//首先，读取功能

List<Book> books = *readAllElementsFromXMLDocument*(document);

*traverseBooks*(books);

//其次，修改功能 修改内容：将id为b002的元素的title改为Java Core，Price改为100.01

*ModifyInformationOfXMLDocument*(document);

//再者：实现删除功能 删除内容：删除掉id为book1的元素内容。

*deleteInformationOfXMLDocument*(document);

//最后：实现添加i新元素功能 添加内容：id为book3，title内容为：凤姐玉照，price内容为10000.00

*addNewBookToXMLDocument*(document);

/\*\*

\* 第三步：将得到的document对象持久化保存到硬盘（XML）

\*/

*writeToNewXMLDocument*(document);

}

//实现了添加新节点：book的功能

**private** **static** **void** addNewBookToXMLDocument(Document document) {

Element root = document.getRootElement();

Element newBook = root.addElement("book");

newBook.addAttribute("id", "book3");

Element title = newBook.addElement("title");

title.setText("凤姐玉照");

Element price = newBook.addElement("price");

price.setText("10000.01");

}

//该方法实现了使用dom4j的删除元素的功能

**private** **static** **void** deleteInformationOfXMLDocument(Document document) {

Element root = document.getRootElement();

**for** (Iterator it = root.elementIterator(); it.hasNext();) {

Element book = (Element) it.next();

String id = book.attributeValue("id");

**if** ("book1".equals(id)) {

Element parent = book.getParent();

parent.remove(book);

}

}

}

//该方法的作用是修改document中的内容 将id为b002的元素的title改为Java Core，Price改为100.01

**private** **static** **void** ModifyInformationOfXMLDocument(Document document) {

Element root = document.getRootElement();

List books = root.elements();

**for** (**int** i = 0; i < books.size(); i++) {

Element book = (Element) books.get(i);

**if** ("book2".equals(book.attributeValue("id"))) {

**for** (Iterator it = book.elementIterator(); it.hasNext();) {

Element node = (Element) it.next();

String type = node.getName();

**if** ("title".equals(type)) {

node.setText("JAVA Core");

}

**if** ("price".equals(type)) {

node.setText("100.01");

}

}

}

}

**try** {

*writeToNewXMLDocument*(document);

} **catch** (Exception e) {

e.printStackTrace();

}

}

//遍历集合

**private** **static** **void** traverseBooks(List<Book> books) {

**for** (Iterator<Book> iterator = books.iterator(); iterator.hasNext();) {

Book book = iterator.next();

System.***out***.println(book);

}

}

//该方法实现了对xml文档的读取功能

**private** **static** List<Book> readAllElementsFromXMLDocument(Document document) {

List<Book> books = **new** ArrayList<Book>();

Element root = document.getRootElement();

List list = root.elements();

**for** (**int** i = 0; i < list.size(); i++) {

Element book = (Element) list.get(i);

Book b = **new** Book();

String id = book.attributeValue("id");

List ll = book.elements();

b.setId(id);

System.***out***.println(id);

**for** (**int** j = 0; j < ll.size(); j++) {

Element element = (Element) ll.get(j);

**if** ("title".equals(element.getName())) {

String title = element.getText();

b.setTitle(title);

System.***out***.println(title);

}

**if** ("price".equals(element.getName())) {

String price = element.getText();

**double** p = Double.*parseDouble*(price);

b.setPrice(p);

System.***out***.println(price);

}

}

books.add(b);

}

**return** books;

}

//通过document对象将内存中的dom树保存到新的xml文档。

**private** **static** **void** writeToNewXMLDocument(Document document) **throws** Exception {

XMLWriter writer = **new** XMLWriter(**new** FileWriter("src/com/zc/homeWork19/newbooks.xml"));

writer.write(document);

writer.close();

}

//该方法用于得到document对象。

**private** **static** Document getDocument() **throws** Exception {

SAXReader sr = **new** SAXReader();

Document document = sr.read("src\\books.xml");

**return** document;

}

}

#### JDOM

JDOM是使用Java语言编写的，用于读写及操作XML的一套组件，Jdom同时具有DOM修改文件的优点和SAX读取快速的优点。首先下载JDOM的JAR包，下载地址：http://www.jdom.org/downloads/index.html

创建XML文档

XML文件是一种典型的树形文件，每个文档元素都是一个document元素的子节点。而每个子元素都是一个Element对象，对象可以向下包含。

1. 因此我们可以通过先创建元素再将元素添加到父元素中，最后将顶层元素添加到根元素中。
2. 创建完文档元素后，就可以把元素添加到document对象中，然后写入文件。

主要使用的函数：

Element.setAttribute 为元素添加信息

Element.addContent(String,String) 为元素添加子元素内容，也可以直接添加另一个元素节点

Document.setRootElement(Element) 为文档添加根元素

XMLOutputter.output(Document,FileWriter) 将Docuemnt写入到FileWriter文件流中

下面是主要的操作过程，写入文件的过程放到了saveXML中

@SuppressWarnings("null")

**public** **static** **void** createXML() {

// 创建document

Document mydoc = **new** Document();

// 创建元素person1

Element person1 = **new** Element("person");

person1.setAttribute("id", "ID001");

// 添加注释

person1.addContent(**new** Comment("this is person1"));

person1.addContent(**new** Element("name").setText("xingoo"));

person1.addContent(**new** Element("age").setText("25"));

person1.addContent(**new** Element("sex").setText("M"));

// 可以嵌套添加子元素

Element address1 = **new** Element("address");

address1.setAttribute("zone", "province");

address1.addContent("LiaoNing");

person1.addContent(address1);

// 创建元素person2

Element person2 = **new** Element("person");

person2.setAttribute("id", "ID002");

// 添加注释

person2.addContent(**new** Comment("this is person2"));

person2.addContent(**new** Element("name").setText("xhalo"));

person2.addContent(**new** Element("age").setText("26"));

person2.addContent(**new** Element("sex").setText("M"));

// 可以嵌套添加子元素

Element address2 = **new** Element("address");

address2.setAttribute("zone", "province");

address2.addContent("JiLin");

person2.addContent(address2);

// 在doc中添加元素Person

Element info = **new** Element("information");

info.addContent(person1);

info.addContent(person2);

mydoc.setRootElement(info);

saveXML(mydoc);

}

saveXML()代码：

**public** **static** **void** saveXML(Document doc) {

// 将doc对象输出到文件

**try** {

// 创建xml文件输出流

XMLOutputter xmlopt = **new** XMLOutputter();

// 创建文件输出流

FileWriter writer = **new** FileWriter("person.xml");

// 指定文档格式

Format fm = Format.getPrettyFormat();

// fm.setEncoding("GB2312");

xmlopt.setFormat(fm);

// 将doc写入到指定的文件中

xmlopt.output(doc, writer);

writer.close();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**读取XML文档**

读取文档，首先需要一个xml的解析器，它可以自动的解析出各个元素，并且把子元素作为自己的孩子节点，方便操作。

主要使用的函数：

SAXBuilder.build("xxx.xml") 解析XML文档

Document.getRootElement() 获取根元素

Element.getChildren() 获取根元素下的子元素，返回List<Element>

Element.getAttributeValue(String) 获取指定元素的信息

Element.getChildText 获取指定元素的内容

**public** **static** **void** readXML() {

// 使用SAXBuilder解析器解析xml文件

SAXBuilder sb = **new** SAXBuilder();

Document doc = **null**;

**try** {

doc = sb.build("person.xml");

Element root = doc.getRootElement();

List<Element> list = root.getChildren("person");

**for** (Element el : list) {

String id = el.getAttributeValue("id");

String name = el.getChildText("name");

String age = el.getChildText("age");

String sex = el.getChildText("sex");

System.out.println("id:" + id);

System.out.println("name:" + name);

System.out.println("age:" + age);

System.out.println("sex:" + sex);

System.out.println("--------------------------");

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

**修改XML文档**

修改XML文档，也是先利用解析器找到指定的元素，使用setText或者setAttributeValue来修改元素内容

修改后记得要保存到文件中，即在此调用saveXML()

**public** **static** **void** updateXML(){

SAXBuilder sb = **new** SAXBuilder();

Document doc = **null**;

**try** {

doc = sb.build("person.xml");

Element root = doc.getRootElement();

List<Element> list = root.getChildren("person");

**for** (Element el : list) {

**if** (el.getAttributeValue("id").equals("ID001")) {

Element name = el.getChild("name");

name.setText("xingoo---update");

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

saveXML(doc);

}

**删除XML文档元素**

删除操作类似修改，调用removeCotent就可以根据内容删除指定的元素了。但是要用他的父节点调用。最后也需要保存到文件中才可以。

**public** **static** **void** removeXML() {

SAXBuilder sb = **new** SAXBuilder();

Document doc = **null**;

**try** {

doc = sb.build("person.xml");

Element root = doc.getRootElement();

List<Element> list = root.getChildren("person");

**for** (Element el : list) {

**if** (el.getAttributeValue("id").equals("ID001")) {

root.removeContent(el);

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

saveXML(doc);

}

#### SAX

SAX的全称是Simple APIs for XML，也即XML简单应用程序接口。与DOM不同，SAX提供的访问模式是一种顺序模式，这是一种快速读写XML数据的方式。当使用SAX分析器对XML文档进行分析时，会触发一系列事件，并激活相应的事件处理函数，应用程序通过这些事件处理函数实现对XML文档的访问，因而SAX接口也被称作事件驱动接口。

**读取XML文件**

首先，要读取的目标XML文件如下：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<oes:Notifications xmlns:oes=*"http://xml.sax.test.com/oesAccessNotification"*>

<oes:Notification>

<oes:NotificationID>11111</oes:NotificationID>

<oes:NotificationType>AlarmNew</oes:NotificationType>

<oes:timeStamp>2009-02-25T08:57:17</oes:timeStamp>

<oes:Appendix>

<oes:MapItem key=*"key"* value=*"value"*/>

</oes:Appendix>

<oes:Content>

<alarmNew systemDN=*"PLMN-1/S3SN-1/SRME-BSS-2/SBSS-0"*>

<alarmId>400951</alarmId>

<alarmText>PIPE 0 IS SLOW OR NOT WORKING</alarmText>

<eventTime>2009-02-25T08:57:17+02:00</eventTime>

<eventType>processingError</eventType>

<perceivedSeverity>critical</perceivedSeverity>

<probableCause>0</probableCause>

<specificProblem>86600</specificProblem>

<additionalText1>A Raised by pipe supervision script, process ID 20848</additionalText1>

<additionalText2>A test additional text2</additionalText2>

<additionalText3>A test additional text3</additionalText3>

<additionalText4>A test additional text4</additionalText4>

<additionalText5>A Original Additional text: test alarm1 | Original Probable Cause: Toxic Leak1 Detected |

Original alarm time: 20090901183006+0530 | Automatic clearing:Y

</additionalText5>

<additionalText6>Original

</additionalText6>

</alarmNew>

</oes:Content>

</oes:Notification>

</oes:Notifications>

SAX读取该XML文件的过程如下：

1. 定义XML中各种标签：

**class** Constant {

**public** **static** **final** String ***NAME\_SPACE*** = "xmlns:oes";

**public** **static** **final** String ***SCHEMA*** = "http://xml.sax.test.com/oesAccessNotification";

**public** **static** **final** String ***NOTIFICATIONS*** = "oes:Notifications";

**public** **static** **final** String ***NOTIFICATION*** = "oes:Notification";

**public** **static** **final** String ***NOTIFICATION\_ID*** = "oes:NotificationID";

**public** **static** **final** String ***NOTIFICATION\_TYPE*** ="oes:NotificationType";

**public** **static** **final** String ***TIME\_STAMP*** = "oes:timeStamp";

**public** **static** **final** String ***APPENDIX*** = "oes:Appendix";

**public** **static** **final** String ***MAP\_ITEM*** = "oes:MapItem";

**public** **static** **final** String ***KEY*** = "key";

**public** **static** **final** String ***VALUE*** = "value";

**public** **static** **final** String ***CONTENT*** = "oes:Content";

**public** **static** **final** String ***ALARM\_NEW*** = "alarmNew";

**public** **static** **final** String ***SYSTEM\_DN*** = "systemDN";

**public** **static** **final** String ***ALARM\_ID*** = "alarmId";

**public** **static** **final** String ***ALRAM\_TEXT*** = "alarmText";

**public** **static** **final** String ***EVENT\_TIME*** = "eventTime";

**public** **static** **final** String ***EVENT\_TYPE*** = "eventType";

**public** **static** **final** String ***PERCEIVED\_SEVERITY*** = "perceivedSeverity";

**public** **static** **final** String ***PROBABLE\_CAUSE***= "probableCause";

**public** **static** **final** String ***SPECIFIC\_PROBLEM*** = "specificProblem";

**public** **static** **final** String ***ADDITION\_TEXT1*** = "additionalText1";

**public** **static** **final** String ***ADDITION\_TEXT2*** = "additionalText2";

**public** **static** **final** String ***ADDITION\_TEXT3*** = "additionalText3";

**public** **static** **final** String ***ADDITION\_TEXT4*** = "additionalText4";

**public** **static** **final** String ***ADDITION\_TEXT5*** = "additionalText5";

**public** **static** **final** String ***ADDITION\_TEXT6*** = "additionalText6";

**public** **static** **final** String ***ADDITION\_TEXT7*** = "additionalText7";

}

这些定义会在读取XML的处理过程中用到。

1. 定义XML文件节点对应的java对象：

**class** EventFactory {

**private** XMLReader xmlReader;

**public** **static** **class** InternalEvent {

**private** String notificationType = "";

**private** Map<String, String> props = **new** HashMap<String, String>();

**public** String getNotificationType() {

**return** notificationType;

}

**public** String getProp(String name) {

String str = props.get(name);

**if** (str == **null**) {

**return** "";

} **else** {

**return** str;

}

}

**public** Map<String, String> getProps(){

**return** props;

}

**public** **void** setNotificationType(String notificationType) {

**this**.notificationType = notificationType;

}

**public** **void** putAttribute(String name, String value) {

**this**.props.put(name, value);

}

}

//调用SAX读取XML的方法，XML文件的数据会被存放到该List中

**public** List<InternalEvent> read(String xmlPath) **throws** ParserConfigurationException, SAXException {

SAXParserFactory spf = SAXParserFactory.newInstance();

SAXParser saxParser = spf.newSAXParser();

xmlReader = saxParser.getXMLReader();

List<InternalEvent> container = **new** LinkedList<InternalEvent>();

ContentHandler handler = **new** ReadXMLHandler(container);

xmlReader.setContentHandler(handler);

**try** {

xmlReader.parse(**new** InputSource(xmlPath));

} **catch** (IOException e) {

e.printStackTrace();

}

**return** container;

}

}

该对象会在SAX读取XML文件时，将XML数据转换为内存中的java对象。

1. SAX读取XML文件：

**class** ReadXMLHandler **extends** DefaultHandler {

**private** List<EventFactory.InternalEvent> eventContainer;

**private** StringBuilder buf = **new** StringBuilder();

**private** EventFactory.InternalEvent event;

**private** **static** **final** Set<String> ***ATTR\_TAGS*** = **new** HashSet<String>()

**static** {

***ATTR\_TAGS***.add(Constant.EVENT\_TIME);

***ATTR\_TAGS***.add(Constant.SPECIFIC\_PROBLEM);

***ATTR\_TAGS***.add(Constant.ALRAM\_TEXT);

***ATTR\_TAGS***.add(Constant.PERCEIVED\_SEVERITY);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT1);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT2);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT3);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT4);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT5);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT6);

***ATTR\_TAGS***.add(Constant.ADDITION\_TEXT7);

***ATTR\_TAGS***.add(Constant.EVENT\_TYPE);

}

**public** ReadXMLHandler(List<EventFactory.InternalEvent> eventContainer) {

**this**.eventContainer = eventContainer;

}

@Override

**public** **void** startElement(String uri, String localName, String qName,

Attributes attributes) **throws** SAXException {

buf.setLength(0);

**if** (qName.equals("oes:Notification")) {

event = **new** EventFactory.InternalEvent();

eventContainer.add(event);

}

**else** **if**(qName.equals(Constant.MAP\_ITEM)){

//获取元素中的属性值，如<a key="key" value="value"/>，获取key和value

String key = attributes.getValue(Constant.KEY);

event.putAttribute(Constant.KEY, key);

String value = attributes.getValue(Constant.VALUE);

event.putAttribute(Constant.VALUE, value);

}

**else** **if**(qName.equals(Constant.ALARM\_NEW)){

String systemDn = attributes.getValue(Constant.SYSTEM\_DN);

event.putAttribute(Constant.SYSTEM\_DN, systemDn);

}

}

@Override

**public** **void** endElement(String uri, String localName, String qName)

**throws** SAXException {

**if** (qName.equals(Constant.NOTIFICATION\_TYPE)) {

event.setNotificationType(buf.toString());

}

**else** **if** (***ATTR\_TAGS***.contains(qName)){

event.putAttribute(qName, buf.toString());

}

}

//获取元素值，如<a>abc</a>，获取其中的abc

@Override

**public** **void** characters(**char**[] ch, **int** start, **int** length)

**throws** SAXException {

buf.append(ch, start, length);

}

}

**转换并写出XML文件**

比起SAX读取XML来，SAX写XML要相对复杂一些，流程如下：

1. 对读取的XML对象做一个简单的转换：

**class** Convert {

**public** **static** String convertString(String value){

**return** value + "\_TEST";

}

}

转换很简单，即将XML标签加一个“\_TEST”，同时给值也加一个“\_TEST“

1. 写XML文件

**class** WriteXML {

SAXTransformerFactory fac = (SAXTransformerFactory) SAXTransformerFactory.newInstance();

**private** TransformerHandler handler = **null**;

**private** OutputStream outStream = **null**;

**private** String fileName;

**private** AttributesImpl atts;

**private** String rootElement;

//元素层次，用于控制XML缩进

**private** **static** **int** *level* = 0;

//每个层次父级缩进4个空格，即一个tab

**private** **static** String *tab* = " ";

//系统换行符，Windows为："\n"，Linux/Unix为："/n"

**private** **static** **final** String ***separator*** = System.*getProperties*().getProperty("os.name").toUpperCase().indexOf("WINDOWS") != -1 ? "\n" : "/n";

**public** WriteXML(String fileName, String rootElement) {

**this**.fileName = fileName;

**this**.rootElement = rootElement;

init();

}

**public** **void** init() {

**try** {

handler = fac.newTransformerHandler();

Transformer transformer = handler.getTransformer();

//设置输出采用的编码方式

transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

//是否自动添加额外的空白

transformer.setOutputProperty(OutputKeys.INDENT, "yes");

//是否忽略xml声明

transformer.setOutputProperty(OutputKeys.OMIT\_XML\_DECLARATION, "no");

outStream = **new** FileOutputStream(fileName);

Result resultxml = **new** StreamResult(outStream);

handler.setResult(resultxml);

atts = **new** AttributesImpl();

start();

}**catch** (Exception e) {

e.printStackTrace();

}

}

**private** **void** start() {

**try** {

handler.startDocument();

//设置schema和名称空间

atts.addAttribute("", "", Constant.NAME\_SPACE, String.**class**.getName(), Constant.SCHEMA);

handler.startElement("", "", rootElement, atts);

} **catch** (Exception e) {

e.printStackTrace();

}

}

//元素里面会嵌套子节点，因此元素的开始和结束分开写

//如：<a><b>bcd</b></a>

**private** **void** startElement(String objectElement, AttributesImpl attrs)

**throws** SAXException {

**if**(attrs == **null**){

attrs = **new** AttributesImpl();

}

*level*++;

appendTab();

**if** (objectElement != **null**) {

//注意，如果atts.addAttribute设置了属性，则会输出如：<a key="key" value="value">abc</a>格式

//如果没有设置属性，则输出如：<a>abc</a>格式

handler.startElement("", "", objectElement, attrs);

}

}

//正常元素结束标记，如：</a>

**private** **void** endElement(String objectElement) **throws** SAXException{

*level*--;

appendTab();

**if** (objectElement != **null**) {

handler.endElement("", "", objectElement);

}

}

//自封闭的空元素，如<a key="key" value="value"/>，不用换行，写在一行时XML自动会自封闭

**private** **void** endEmptyElement(String objectElement) **throws** SAXException{

handler.endElement("", "", objectElement);

}

//无子节点的元素成为属性，如<a>abc</a>

**private** **void** writeAttribute(String key, String value) **throws** SAXException{

atts.clear();

*level*++;

appendTab();

handler.startElement("", "", key, atts);

handler.characters(value.toCharArray(), 0, value.length());

handler.endElement("", "", key);

*level*--;

}

**public** **void** end() {

**try** {

handler.endElement("", "", rootElement);

// 文档结束,同步到磁盘

handler.endDocument();

outStream.close();

}**catch** (Exception e) {

e.printStackTrace();

}

}

//Tab缩进，SAX默认不自动缩进，因此需要手动根据元素层次进行缩进控制

**private** **void** appendTab() **throws** SAXException{

String indent = ***separator*** + " ";

**for**(**int** i = 1 ; i< *level*; i++){

indent += *tab*;

}

handler.characters(indent.toCharArray(), 0, indent.length());

}

**public** **void** writeNotification(InternalEvent event) **throws** SAXException{

Map<String, String> props = event.getProps();

Set<String> keys = props.keySet();

*level* = 0;

//写<oes:Notification>节点

startElement(Constant.NOTIFICATION, **null**);

//写oes:NotificationID

writeAttribute(Convert.convertString(Constant.NOTIFICATION\_ID), Convert.convertString(props.get(Constant.NOTIFICATION\_ID)));

keys.remove(Constant.NOTIFICATION\_ID);

//写oes:NotificationType

writeAttribute(Convert.convertString(Constant.NOTIFICATION\_TYPE), Convert.convertString(event.getNotificationType()));

//写oes:timeStamp

writeAttribute(Convert.convertString(Constant.TIME\_STAMP), Convert.convertString(props.get(Constant.TIME\_STAMP)));

keys.remove(Constant.TIME\_STAMP);

//写<oes:Appendix>节点

startElement(Constant.APPENDIX, **null**);

//写oes:MapItem

atts = **new** AttributesImpl();

atts.addAttribute("", "", Convert.convertString(Constant.KEY), String.**class**.getName(), Convert.convertString(props.get(Constant.KEY)));

keys.remove(Constant.KEY);

atts.addAttribute("", "", Convert.convertString(Constant.VALUE), String.**class**.getName(), Convert.convertString(props.get(Constant.VALUE)));

keys.remove(Constant.VALUE);

startElement(Constant.MAP\_ITEM, atts);

//结束oes:MapItem，由于MapItem是个自封闭的元素，需要特殊处理

endEmptyElement(Constant.MAP\_ITEM);

keys.remove(Constant.MAP\_ITEM);

//结束oes:MapItem节点

endElement(Constant.APPENDIX);

keys.remove(Constant.APPENDIX);

//写oes:Content节点

startElement(Constant.CONTENT, **null**);

keys.remove(Constant.CONTENT);

//写alarmNew节点

atts = **new** AttributesImpl();

atts.addAttribute("", "", Convert.convertString(Constant.SYSTEM\_DN), String.**class**.getName(), Convert.convertString(props.get(Constant.SYSTEM\_DN)));

startElement(Constant.ALARM\_NEW, atts);

keys.remove(Constant.ALARM\_NEW);

//写Alarm节点内的属性

**for**(String key : keys){

writeAttribute(Convert.convertString(key), Convert.convertString(props.get(key)));

}

//结束alarmNew节点

endElement(Constant.ALARM\_NEW);

//结束oes:Content节点

endElement(Constant.CONTENT);

//结束<oes:Notification>节点

endElement(Constant.NOTIFICATION);

}

}

1. 先用SAX读取XML文件，然后使用SAX处理写出的demo程序如下：

**public** **class** FlexMapping {

**private** **static** String *inputFile* = "input/input.xml";

**private** **static** String *outputFile* = "output/output.xml";

**private** **static** List<InternalEvent> *events*;

**public** **static** **void** main(String[] args) {

**long** start = System.*currentTimeMillis*();

**try** {

*events* = **new** EventFactory().read(*inputFile*);

WriteXML xml = **new** WriteXML(*outputFile*, Constant.NOTIFICATIONS);

**for**(InternalEvent event : *events*){

xml.writeNotification(event);

}

xml.end();

}**catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println("耗时：" + (System.*currentTimeMillis*() - start) + "ms.");

}

}

## Socket编程

### 网络基础知识

1. 两台计算机间进行通讯需要以下三个条件：

IP地址、网络协议、端口号

1. TCP/IP协议

是目前世界上应用最为广泛的协议，是以TCP和IP为基础的不同层次上多个协议的集合，也成为TCP/IP协议族、或TCP/IP协议栈。

TCP ：Transmission Control Protocol 传输控制协议

IP：Internet Protocol 互联网协议

1. TCP/IP五层模式

应用层：HTTP、FTP、SMTP、Telnet等

传输层：TCP/IP

网络层

数据链路层

物理层：网线、双绞线、网卡等

1. IP地址

为实现网络中不同计算机之间的通信，每台计算机都必须有一个唯一的标识---IP地址。

32位二进制

1. 端口

逻辑意义上的数据传输通道、或者说模拟通道。TCP/IP协议约定，每台计算机拥有65536个这种逻辑通信端口。

区分一台主机的多个不同应用程序，端口号范围为0-65535，其中0-1023位为系统保留。

如：HTTP：80 FTP：21 Telnet：23

IP地址+端口号组成了所谓的Socket，Socket是网络上运行的程序之间双向通信链路的终结点，是TCP和UDP的基础

1. Socket套接字

网络上具有唯一标识的IP地址和端口组合在一起才能构成唯一能识别的标识符套接字。

Socket原理机制：

通信的两端都有Socket

网络通信其实就是Socket间的通信

数据在两个Socket间通过IO传输

1. Java中的网络支持

针对网络通信的不同层次，Java提供了不同的API，其提供的网络功能有四大类：

InetAddress:用于标识网络上的硬件资源，主要是IP地址

URL：统一资源定位符，通过URL可以直接读取或写入网络上的数据

Sockets：使用TCP协议实现的网络通信Socket相关的类

Datagram:使用UDP协议，将数据保存在用户数据报中，通过网络进行通信。

### InetAddress

InetAddress类用于标识网络上的硬件资源，标识互联网协议(IP)地址。

该类没有构造方法

//获取本机的InetAddress实例

InetAddress address = InetAddress.*getLocalHost*();

address.getHostName(); //获取计算机名

address.getHostAddress(); //获取IP地址

**byte**[] bytes = address.getAddress(); //获取字节数组形式的IP地址，以点分割的四部分

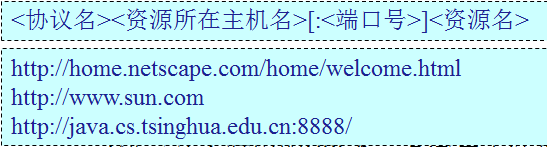
//获取其他主机的InetAddress实例

InetAddress address2 = InetAddress.*getByName*("其他主机名");

InetAddress address3 = InetAddress.*getByName*("IP地址");

### URL类

URL（Uniform Resource Locator，统一资源定位器）用于表示Internet上资源的地址。



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

//使用URL读取网页内容

//创建一个URL实例

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

InputStream is = url.openStream(); //通过openStream方法获取资源的字节输入流

//将字节输入流转换为字符输入流，如果不指定编码，中文可能会出现乱码

InputStreamReader isr = **new** InputStreamReader(is, "UTF-8");

BufferedReader br = **new** BufferedReader(isr); //将字符输入流添加缓冲，提高读取效率

String data = "";

**while**((data = br.readLine()) != **null**){

System.***out***.println(data);

}

br.close();

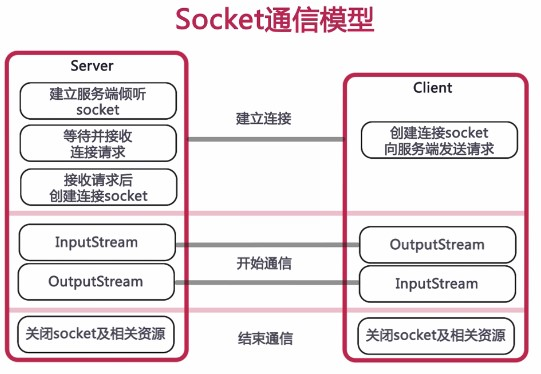
isr.close();

is.close();

}

### TCP编程

1. TCP协议是面向连接的、可靠的、有序的、以字节流的方式发送数据，通过三次握手方式建立连接，形成传输数据的通道，在连接中进行大量数据的传输，效率会稍低
2. Java中基于TCP协议实现网络通信的类：客户端的Socket类、服务端的ServerSocket类



1. Socket通信的步骤
2. 创建ServerSocket和Socket
3. 打开连接到Socket的输入/输出流
4. 按照协议对Socket进行读/写操作
5. 关闭输入输出流、关闭Socket
6. 服务器端的实现
7. 创建ServerSocket对象，绑定监听端口
8. 通过accept()方法监听客户端请求
9. 连接建立后，通过输入流读取客户端发送的请求信息
10. 通过输出流向客户端发送响应信息
11. 关闭相关资源

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

//基于TCP协议的Socket通信，实现用户登录，服务端

//1.创建一个服务端Socket，即ServerSocket，指定绑定的端口，并监听此端口

ServerSocket serverSocket = **new** ServerSocket(10086);//1024-65535的某个端口

//2.调用accept()方法开始监听，等待客户端的连接

Socket socket = serverSocket.accept();

//3.获取输入流，并读取客户端信息

InputStreamReader isr = **new** InputStreamReader(socket.getInputStream());

BufferedReader br = **new** BufferedReader(isr);

String str = "";

**while**((str = br.readLine()) != **null**){

System.***out***.println("我是服务器，客户端说：" + str);

}

socket.shutdownInput(); //关闭输入流

//4.获取输出流，响应客户端的请求

PrintWriter pw = **new** PrintWriter(socket.getOutputStream());

pw.write("欢迎您！");

pw.flush();

//5.关闭资源

pw.close(); br.close(); isr.close(); socket.close(); serverSocket.close();

}

1. 客户端实现
2. 创建Socket对象，指明需要连接的服务器的地址和端口号
3. 连接建立后，通过输出流想服务器端发送请求信息
4. 通过输入流获取服务器响应的信息
5. 关闭响应资源

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

//客户端

//1.创建客户端Socket，指定服务器地址和端口

Socket socket = **new** Socket("localhost", 10086);

//2.获取输出流，向服务器端发送信息

PrintWriter pw = **new** PrintWriter(socket.getOutputStream()); //将输出流包装成打印流

pw.write("用户名：admin;密码：123");

pw.flush();

socket.shutdownOutput();

//3.获取输入流，并读取服务器端的响应信息

InputStreamReader isr = **new** InputStreamReader(socket.getInputStream());

BufferedReader br = **new** BufferedReader(isr);

String str = "";

**while**((str = br.readLine()) != **null**){

System.***out***.println("我是客户端，服务器说：" + str);

}

//4.关闭资源

br.close(); isr.close(); socket.close(); pw.close();

}

### UDP编程

UDP协议（用户数据报协议）是无连接的、不可靠的、无序的、速度快。

进行数据传输时，首先将要传输的数据定义成数据报（Datagram），大小限制在64k，在数据报中指明数据索要达到的Socket（主机地址和端口号），然后再将数据报发送出去。

DatagramPacket类:表示数据报包

DatagramSocket类：进行端到端通信的类

1. 服务端实现
2. 创建DatagramSocket，指定端口号
3. 创建DatagramPacket
4. 接受客户端发送的数据信息
5. 读取数据

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

//服务器端，实现基于UDP的用户登录

//1.创建服务器端DatagramSocket，指定端口

DatagramSocket socket = **new** DatagramSocket(10010);

//2.创建数据报，用于接收客户端发送的数据

**byte**[] data = **new** **byte**[1024];

DatagramPacket packet = **new** DatagramPacket(data, data.length);

//3.接收客户端发送的数据

socket.receive(packet); //此方法在接收数据报之前会一直阻塞

//4.读取数据

String str = **new** String(data, 0, data.length);

System.***out***.println("我是服务器，客户端说：" + str);

//==================================================

//向客户端响应数据

//1.定义客户端的地址、端口号、数据

InetAddress address = packet.getAddress();

**int** port = packet.getPort();

**byte**[] data2 = "欢迎您".getBytes();

//2.创建数据报，包含响应的数据信息

DatagramPacket packet2 = **new** DatagramPacket(data2, data2.length, address, port);

//3.响应客户端

socket.send(packet2);

//关闭资源

socket.close();

}

1. 客户端实现
2. 定义发送信息
3. 创建DatagramPacket，包含将要发送的信息
4. 创建DatagramSocket
5. 发送数据

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

//客户端

//1.定义服务器的地址、端口号、数据

InetAddress address = InetAddress.*getByName*("localhost");

**int** port = 10010;

**byte**[] data = "用户名：admin;密码：123".getBytes();

//2.创建数据报，包含发送的数据信息

DatagramPacket packet = **new** DatagramPacket(data, data.length, address, port);

//3.创建DatagramSocket对象

DatagramSocket socket = **new** DatagramSocket();

//4.向服务器发送数据

socket.send(packet);

//==============================================

//接收服务器端响应的数据

//1.创建数据报，用户接收服务器响应的数据

**byte**[] data2 = **new** **byte**[1024];

DatagramPacket packet2 = **new** DatagramPacket(data2, data2.length);

//2.接收服务器响应的数据

socket.receive(packet2);

String str = **new** String(data2, 0, packet2.getLength());

System.***out***.println("我是客户端，服务器说：" + str);

//4.关闭资源

socket.close();

}

## I/O编程

### Java的I/O演进之路

JDK1.4之前的早期版本，Java对I/O的支持并不完善，开发人员在开发高性能I/O程序的时候，会面临一些巨大的挑战和困难，主要问题如下：

1. 没有数据缓冲区，I/O性能存在问题；
2. 没有C或者C++中的Channel概念，只有输入和输出流；
3. 同步阻塞式I/O通信（BIO），通常会导致通信线程被长时间阻塞；
4. 支持的字符集有限，硬件可移植性不好。

在Java支持异步I/O之前的很长一段时间里，高性能服务端开发领域一直被C++和C长期占据，Java的同步阻塞I/O被大家所诟病。

在JDK1.4推出Java NIO之前，基于Java的所有Socket通信都采用了同步阻塞模式（BIO），这种一请求一应答的通信模型简化了上层的应用开发，但是在性能和可靠性方面却存在着巨大的瓶颈。因此，在很长一段时间里，大型的应用服务器都采用C或者C++语言开发，因为它们可以直接使用操作系统提供的异步I/O或者AIO能力。当并发访问量增大、响应时间延迟增大之后，采用Java BIO开发的服务端软件只有通过硬件的不断扩容来满足高并发和延时，极大地增加了企业的成本，并且随着集群规模的不断膨胀，系统的可维护性也面临巨大的挑战，只能通过采购性能更高的硬件服务器来解决问题，这会导致恶性循环。

正是由于Java传统BIO的拙劣表现，才使得Java支持非阻塞I/O的呼声日渐高涨，最终，JDK1.4版本提供了新的NIO类库，Java终于也可以支持非阻塞I/O了。

JDK1.4推出 NIO 1.0 。新增java.nio包，提供了很多进行异步I/O开发的API和类库，主要的类和接口如下。

1. 进行异步I/O操作的缓冲区ByteBuffer等；
2. 进行异步I/O操作的管道Pipe；
3. 进行各种I/O操作（异步或者同步）的Channel，包括ServerSocketChannel和SocketChannel;
4. 多种字符集的编码能力和解码能力；
5. 实现非阻塞I/O操作的多路复用器selector;
6. 基于流行的Perl实现的正则表达式类库；
7. 文件通道FileChannel。

新的NIO类库的提供，极大地促进了基于Java的异步非阻塞编程的发展和应用，但是，它依然有不完善的地方，特别是对文件系统的处理能力仍显不足，主要问题如下。

1. 没有统一的文件属性（例如读写权限）；
2. API能力比较弱，例如目录的级联创建和递归遍历，往往需要自己实现；
3. 底层存储系统的一些高级API无法使用；
4. 所有的文件操作都是同步阻塞调用，不支持异步文件读写操作。

JDK1.7推出 NIO 2.0。主要提供了如下三个方面的改进。

1. 提供能够批量获取文件属性的API，这些API具有平台无关性，不与特性的文件系统相耦合，另外它还提供了标准文件系统的SPI，供各个服务商扩展实现；
2. 提供AIO功能，支持基于文件的异步I/O操作和针对网络套接字的异步操作；
3. 完成JSR定义的通道功能，包括对配置和多播数据报的支持等。

### 传统的BIO编程

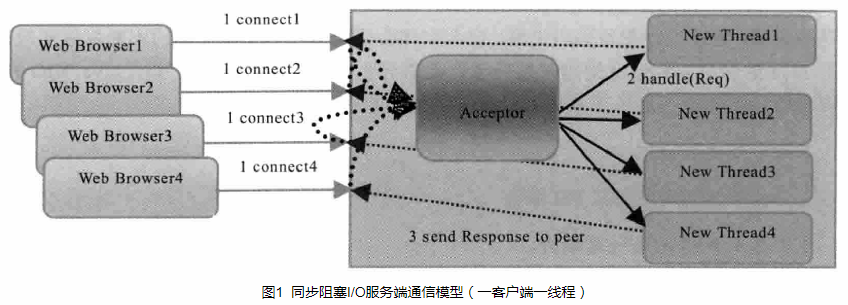
网络编程的基本模型是Client/Server模型，也就是两个进程之间进行相互通信，其中服务端提供位置信息（绑定的IP地址和监听端口），客户端通过连接操作向服务端监听的地址发起连接请求，通过三次握手建立连接，如果连接建立成功，双方就可以通过网络套接字（Socket）进行通信。

在基于传统同步阻塞模型开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功之后，双方通过输入和输出流进行同步阻塞式通信。

以经典的时间服务器(TimeServer)为例，通过代码分析来回顾和熟悉下BIO编程。

#### BIO通信模型图

如下图BIO的服务端通信模型：采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的一请求一应答通信模型。



该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈1:1 的正比关系，犹豫线程是Java虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

#### 同步阻塞式I/O创建的TimeServer

**package** joanna.yan.bio;

**import** java.io.IOException;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** TimeServer {

**public** **static** **void** main(String[] args) {

**int** port=9090;

**if**(args!=**null**&&args.length>0){

**try** {

port=Integer.valueOf(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

ServerSocket server=**null**;

**try** {

server=**new** ServerSocket(port);

System.out.println("The time server is start in port:"+port);

Socket socket=**null**;

**while**(**true**){//通过一个无限循环来监听客户端的连接

socket=server.accept();//如果没有客户端接入，则主线程阻塞在ServerSocket的accept操作上。

**new** Thread(**new** TimeServerHandler(socket)).start();

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(server!=**null**){

System.out.println("The time server close");

**try** {

server.close();

server=**null**;

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

**package** joanna.yan.bio;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** java.util.Date;

**public** **class** TimeServerHandler **implements** Runnable{

**private** Socket socket;

**public** TimeServerHandler(Socket socket) {

**this**.socket = socket;

}

@Override

**public** **void** run() {

BufferedReader in=**null**;

PrintWriter out=**null**;

**try** {

in=**new** BufferedReader(**new** InputStreamReader(**this**.socket.getInputStream()));

out=**new** PrintWriter(**this**.socket.getOutputStream(), **true**);

String currentTime=**null**;

String body=**null**;

**while**(**true**){

body=in.readLine();

**if**(body==**null**){

**break**;

}

System.***out***.println("The time server receive order:"+body);

//如果请求消息为查询时间的指令"QUERY TIME ORDER"则获取当前最新的系统时间。

currentTime="QUERY TIME ORDER".equalsIgnoreCase(body) ?

**new** Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";

out.println(currentTime);

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(in!=**null**){

**try** {

in.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if**(out!=**null**){

out.close();

out=**null**;

}

**if**(**this**.socket!=**null**){

**try** {

**this**.socket.close();

**this**.socket=**null**;

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

#### 同步阻塞式I/O创建的TimeClient

**package** joanna.yan.bio;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**public** **class** TimeClient {

**public** **static** **void** main(String[] args) {

**int** port=9090;

**if**(args!=**null**&&args.length>0){

**try** {

port=Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

Socket socket=**null**;

BufferedReader in=**null**;

PrintWriter out=**null**;

**try** {

socket=**new** Socket("127.0.0.1",port);

in=**new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));

out=**new** PrintWriter(socket.getOutputStream(),**true**);

out.println("QUERY TIME ORDER");

System.***out***.println("Send order 2 server succeed.");

String resp=in.readLine();

System.***out***.println("Now is:"+resp);

} **catch** (UnknownHostException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(out!=**null**){

out.close();

out=**null**;

}

**if**(in!=**null**){

**try** {

in.close();

} **catch** (IOException e) {

e.printStackTrace();

}

in=**null**;

}

**if**(socket!=**null**){

**try** {

socket.close();

} **catch** (IOException e) {

e.printStackTrace();

}

socket=**null**;

}

}

}

}

我们发现，BIO主要的问题在于每当有一个新的客户端请求接入时，服务端必须创建一个新的线程处理新接入的客户端链路，一个线程只能吃力一个客户端连接。在高性能服务器应用领域，往往需要面向成千上万个客户端的并发连接，这种模型显然无法满足高性能、高并发接入的场景。

为了改进一线程一连接模型，后来又演进出了一种通过线程池或者消息队列实现1个或者多个线程处理N个客户端的模型，由于它的底层通信机制依然使用同步阻塞I/O，所以被称为“伪异步”。后面我们将通过对伪异步代码的分析，看看伪异步能否满足我们对高性能、高并发接入的诉求。

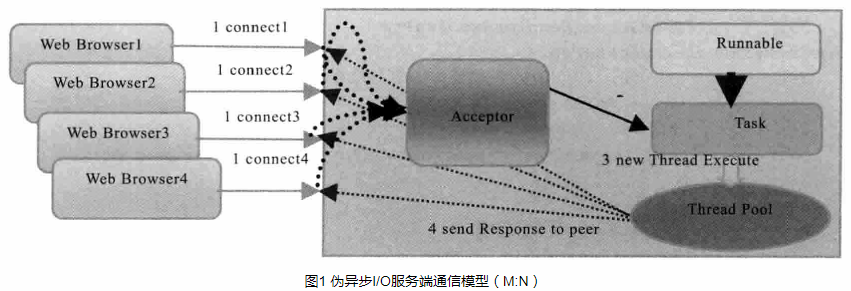
### 伪异步IO编程

为了解决同步阻塞I/O面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化，后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数M：线程池最大线程数N的比例关系，其中M可以远远大于N，通过线程池可以灵活的调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

#### 伪异步I/O模型图

采用线程池和任务队列可以实现一种叫做伪异步的I/O通信框架，它的模型图如下所示。

当有新的客户端接入的时候，将客户端的Socket封装成一个Task（该任务实现java.lang.Runnable接口）投递到后端的线程池中进行处理，JDK的线程池维护一个消息队列和N个活跃线程对消息队列中的任务进行处理。由于线程池可以设置消息队列的大小和最大线程数。因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。



下面我们依然采用时间服务器程序，将其改造成为伪异步I/O时间服务器，然后通过对代码进行分析，找出其弊端

#### 伪异步式I/O创建的TimeServer源码分析

**package** joanna.yan.poio;

**import** java.io.IOException;

**import** java.net.ServerSocket;

**import** java.net.Socket;

/\*\*

\* 伪异步式I/O

\* **@author** Joanna.Yan

\* **@date** 2017年10月24日上午10:16:10

\*/

**public** **class** TimeServer {

**public** **static** **void** main(String[] args) {

**int** port=9090;

**if**(args!=**null**&&args.length>0){

**try** {

port=Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

ServerSocket server=**null**;

**try** {

server=**new** ServerSocket(port);

System.***out***.println("The time server is start in port :"+ port);

Socket socket=**null**;

//创建一个时间服务器类的线程池

TimeServerHandlerExecutePool singleExecutor=**new**

TimeServerHandlerExecutePool(50, 10000);//创建I/O任务

**while**(**true**){

socket=server.accept();

//当接收到新的客户端连接时，将请求Socket封装成一个Task，然后调用execute方法执行。从而避免了每个请求接入都创建一个新的线程。

singleExecutor.execute(**new** TimeServerHandler(socket));

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(server!=**null**){

**try** {

System.***out***.println("The time server close");

server=**null**;

server.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

**package** joanna.yan.poio;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.ThreadPoolExecutor;

**import** java.util.concurrent.TimeUnit;

/\*\*

\* 由于线程池和消息队列都是有界的，因此，无论客户端并发连接数多大，它都不会导致线程个数过于膨胀或者内存溢出，

\* 相比于传统的一连接一线程模型，是一种改良。

\* **@author** Joanna.Yan

\* **@date** 2017年10月24日下午2:39:49

\*/

**public** **class** TimeServerHandlerExecutePool {

**private** ExecutorService executor;

**public** TimeServerHandlerExecutePool(**int** maxPoolSize,**int** queueSize){

executor=**new** ThreadPoolExecutor(Runtime.*getRuntime*().availableProcessors(),

maxPoolSize, 120L, TimeUnit.***SECONDS***,

**new** ArrayBlockingQueue<java.lang.Runnable>(queueSize));

}

**public** **void** execute(java.lang.Runnable task){

executor.execute(task);;

}

}

**package** joanna.yan.poio;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** java.util.Date;

**public** **class** TimeServerHandler **implements** Runnable{

**private** Socket socket;

**public** TimeServerHandler(Socket socket) {

**this**.socket = socket;

}

@Override

**public** **void** run() {

BufferedReader in=**null**;

PrintWriter out=**null**;

**try** {

in=**new** BufferedReader(**new** InputStreamReader(**this**.socket.getInputStream()));

out=**new** PrintWriter(**this**.socket.getOutputStream(), **true**);

String currentTime=**null**;

String body=**null**;

**while**(**true**){

body=in.readLine();

**if**(body==**null**){

**break**;

}

System.***out***.println("The time server receive order:"+body);

//如果请求消息为查询时间的指令"QUERY TIME ORDER"则获取当前最新的系统时间。

currentTime="QUERY TIME ORDER".equalsIgnoreCase(body) ?

**new** Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";

out.println(currentTime);

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(in!=**null**){

**try** {

in.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if**(out!=**null**){

out.close();

out=**null**;

}

**if**(**this**.socket!=**null**){

**try** {

**this**.socket.close();

**this**.socket=**null**;

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

伪异步I/O通信框架采用了线程池实现，因此避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。但是由于它底层的通信依然采用同步阻塞模型，因此无法从根本上解决问题。

#### 伪异步I/O弊端分析

当对Socket的输入流进行读取操作的时候，它会一直阻塞下去，直到发生如下三种事件。

1. 有数据可读；
2. 可用数据已经读取完毕；
3. 发生空指针或者I/O异常。

这意味着当对方发送请求或者应答消息比较缓慢、或者网络传输较慢时，读取输入流一方的通信线程将被长时间阻塞，如果对方要60s才能够将数据发送完成，读取一方的I/O线程也将会被同步阻塞60s，在此期间，其他接入消息只能在消息队列中排队

当调用OutputStream的write方法写输出流的时候，它将会被阻塞，直到要发送的字节全部写入完毕，或者发生异常。学习过TCP/IP相关知识的人都知道，当消息的接收方处理缓慢的时候，将不能及时地从TCP缓冲区读取数据，这将会导致发送方的TCP window size不断减小，直到为0，双方处于Keep-Alive状态，消息发送方将不能再向TCP缓冲区写入消息，这是如果采用的是同步阻塞I/O，write操作将会被无限期阻塞，直到TCP window size大于0或者发生I/O异常。

通过对输入和输出流的API文档进行分析，我们了解到读和写操作都是同步阻塞的，阻塞的时间取决于对方I/O线程的处理速度和网络I/O传输速度。本质上来讲，我们无法保证生产环境的网络状况和对端的应用程序能够足够快，如果我们的应用程序依赖对方的处理速度，它的可靠性就非常差。

伪异步I/O实际上仅仅只是对之前I/O线程模型的一个简单优化，它无法从根本上解决同步I/O导致的通信线程阻塞问题。下面我们就简单分析下如果通信对方返回应答时间过长，会引起的级联故障

1. 服务端处理缓慢，返回应答消息耗费60s，平时只需要10ms。
2. 采用伪异步I/O的线程正在读取故障服务节点的响应，由于读取输入流是阻塞的，因此，它将会被同步阻塞60s。
3. 假如所有的可用线程都被故障服务器阻塞，那后续所有的I/O消息都将在队里中排队。
4. 由于线程池采用阻塞队里实现，当队列积满之后，后续入队的操作将被阻塞。
5. 由于前端只有一个Accptor线程接收客户端接入，它被阻塞在线程池的同步阻塞队列之后，新的客户端请求消息将被拒绝， 客户端会发生大量的连接超时。
6. 由于几乎所有的连接都超时，调用者会认为系统已经崩溃，无法接收新的请求消息。

### NIO编程

NIO，即New I/O，这是官方叫法，因为它相对于之前的I/O类库是新增的。但是，由于之前老的I/O类库是阻塞I/O，New I/O类库的目标就是要让Java支持非阻塞I/O，所以，更多的人喜欢称之为非阻塞I/O（Non-block I/O），由于非阻塞I/O更能够体现NIO的特点，所以这里使用的NIO都是指非阻塞I/O。

与Socket类和ServerSocket类相对应，NIO也提供了SocketChannel和ServerSocketChannel两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞则正好相反。开发人员一般可以根据自己的需要来选择合适的模式，一般来说，低负载、低并发的应用程序可以选择同步阻塞I/O以降低编程复杂度，但是对于高负载、高并发的网络应用，需要使用NIO的非阻塞模式进行开发。

#### NIO类库简介

新的输入/输出（NIO）库是在JDK1.4中引入的。NIO弥补了原来同步阻塞I/O的不足，它在标准Java代码中提供了高速的、面向块的I/O。通过定义包含数据的类，以及通过以块的形式处理这些数据，NIO不使用本机代码就可以利用低级优化，这是原来的I/O包所无法做到的。下面对NIO的一些概念和功能做下简单介绍，以便大家能够快速地了解NIO类库和相关概念。

1. **缓冲区Buffer**

Buffer是一个对象，它包含一些要写入或者要读出的数据。在NIO类库中加入Buffer对象，体现了新库与原I/O的一个重要区别。在面向流的I/O中，可以将数据直接写入或者将数据直接读到Stream对象中。

在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

缓冲区实质上是一个数组。通常它是一个字节数组(ByteBuffer)，也可以使用其他种类的数组。但是缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置(limit)等信息。

最常用的缓冲区是ByteBuffer，一个ByteBuffer提供了一组功能用于操作byte数组。除了ByteBuffer，还有其他的一些缓冲区，事实上，每一种Java基本类型（除了Boolean类型）都对应有一种缓冲区，具体如下：

ByteBuffer：字节缓冲区

CharBuffer：字符缓冲区

ShortBuffer：短整型缓冲区

IntBuffer：整型缓冲区

LongBuffer：长整型缓冲区

FloatBuffer：浮点型缓冲区

DoubleBuffer：双精度浮点型缓冲区

每一个Buffer类都是Buffer接口的一个子实例。除了ByteBuffer,每一个Buffer类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准I/O操作都是使用ByteBuffer，所以它除了具有一般缓冲区的操作之外还提供一些特有的操作，方便网络读写。

1. **通道Channel**

Channel是一个通道，可以通过它读取和写入数据，它就像自来水管一样，网络数据通过Channel读取和写入。通道与流的不同之处在于通道是双向的，流只是在一个方向上移动（一个流必须是InputStream或者OutputStream的子类），而且通道可以用于读、写或者同时读写。因为Channel是全双工的，所以它可以比流更好地映射底层操作系统的API。

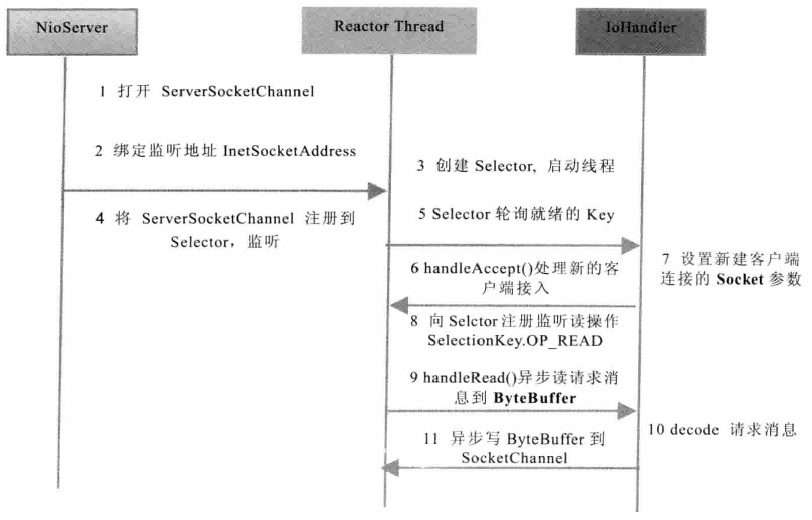
1. **多路复用器Selector**

多路复用器Selector是Java NIO编程的基础，熟练地掌握Selector对于掌握NIO编程至关重要。多路复用器提供选择已经就绪的任务的能力。简单来讲，Selector会不断地轮询注册在其上的Channel，如果某个Channel上面有新的TCP连接接入、读和写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。

一个多路复用器Selector可以同时轮询多个Channel，由于JDK使用了epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2048的限制。这也就意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端，这确实是个非常巨大的进步。

#### NIO服务端序列图

NIO服务端通信序列图如下图所示：



下面，我们对NIO服务端的主要创建过程进行讲解和说明，作为NIO的基础入门，我们将忽略掉一些在生产环境中部署所需要的一些特性和功能。

步骤一：打开ServerSocketChannel，用于监听客户端的连接，它是所有客户端连接的父管道。

ServerSocketChannel acceptorSvr=ServerSocketChannel.open();

步骤二：绑定监听端口，设置连接为非阻塞模式。

acceptorSvr.socket().bind(**new** InetSocketAddress(InetAddress.getByName("IP"),port));

acceptorSvr.configureBlocking(**false**);

步骤三：创建Reactor线程，创建多路复用器并启动线程。

Selector selector=Selector.open();

New Thread(**new** ReactorTask()).start();

步骤四：将ServerSocketChannel注册到Reactor线程的多路复用器Selector上，监听ACCEPT事件。

SelectionKey key=acceptorSvr.register(selector,SelectionKey.OP\_ACCEPT,ioHandler);

步骤五：多路复用器在线程run方法的无线循环体内轮询准备就绪的Key。

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

Set selectedKeys=selector.selectedKeys();

Iterator it=selectedKeys.iterator();

**while**(it.hasNext()){

SelectionKey key=(SelectionKey )it.next();

//...deal with I/O event...

}

步骤六：多路复用器监听到有新的客户端接入，处理新的接入请求，完成TCP三次握手，建立物理链路。

SocketChannel channel=svrChannel.accpet();

步骤七：设置客户端链路为非阻塞模式。

channel.configureBlocking(**false**);

channel.socket().setReuseAddress(**true**);

......

步骤八：将新接入的客户端连接注册到Reactor线程的多路复用器，监听读操作，用来读取客户端发送的网络消息。

SelectionKey key=socketChannel.register(selector,SelectionKey.OP\_READ,ioHandler);

步骤九：异步读取客户端请求消息到缓冲区。

**int** readNumber=channel.read(receivedBuffer);

步骤十：对ByteBuffer进行编解码，如果有半包消息指针reset，继续读取后续的报文，将解码成功的消息封装成Task，投递到业务线程池中，进行业务逻辑编排。

Object message=**null**;

**while**(buffer.hasRemain()){

byteBuffer.mark();

Object message=decode(byteBuffer);

**if**(message==**null**){

　　byteBuffer.reset();

**break**;

}

messageList.add(message);

}

**if**(!byteBuffer.hasRemain()){

byteBuffer.clear();

}**else**{

byteBuffer.compact();

}

**if**(messageList!=**null**& !messageList.isEmpty()){

**for**(Object messageE: messageList){

　　handlerTask(messageE);

}

}

步骤十一：将POJO对象encode成ByteBuffer,调用SocketChannel的异步write接口，将消息异步发送给客户端。

socketChannel.write(buffer);

#### NIO创建的TimeServer源码分析

**package** joanna.yan.nio;

**public** **class** TimeServer {

**public** **static** **void** main(String[] args) {

**int** port=9090;

**if**(args!=**null**&&args.length>0){

**try** {

port=Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

MultiplexerTimeServer timeServer=**new** MultiplexerTimeServer(port);

**new** Thread(timeServer, "NIO-MultiplexerTimeServer-001").start();

}

}

**package** joanna.yan.nio;

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.SelectionKey;

**import** java.nio.channels.Selector;

**import** java.nio.channels.ServerSocketChannel;

**import** java.nio.channels.SocketChannel;

**import** java.util.Date;

**import** java.util.Iterator;

**import** java.util.Set;

/\*\*

\* 多路复用类

\* 它是一个独立的线程，负责轮询多路复器Selector,可以处理多个客户端的并发接入。

\* **@author** Joanna.Yan

\* **@date** 2017年11月6日下午3:51:41

\*/

**public** **class** MultiplexerTimeServer **implements** Runnable{

**private** Selector selector;//多路复用器

**private** ServerSocketChannel servChannel;

**private** **volatile** **boolean** stop;

/\*\*

\* 初始化多路复用器、绑定监听端口

\* **@param** port

\*/

**public** MultiplexerTimeServer(**int** port){

**try** {

selector=Selector.*open*();

servChannel=ServerSocketChannel.*open*();

servChannel.configureBlocking(**false**);

servChannel.socket().bind(**new** InetSocketAddress(port), 1024);

servChannel.register(selector, SelectionKey.***OP\_ACCEPT***);

System.***out***.println("The time server is start in port: "+port);

} **catch** (IOException e) {

e.printStackTrace();

System.*exit*(1);

}

}

**public** **void** stop(){

**this**.stop=**true**;

}

　　　　@Override

**public** **void** run() {

**while**(!stop){

**try** {

//设置selector的休眠时间为1s，无论是否有读写等事件发生，selector每隔1s都被唤醒一次。

selector.select(1000);

//当有处于就绪状态的Channel时，selector就返回就绪状态的Channel的SelectionKey集合。

Set<SelectionKey> selectedKeys=selector.selectedKeys();

Iterator<SelectionKey> it=selectedKeys.iterator();

SelectionKey key=**null**;

//通过对就绪状态的Channel集合进行迭代，可以进行网络的异步读写操作。

**while**(it.hasNext()){

key=it.next();

it.remove();

**try** {

handleInput(key);

} **catch** (Exception e) {

**if**(key!=**null**){

key.cancel();

**if**(key.channel()!=**null**){

key.channel().close();

}

}

}

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

　　　　　　　/\*

\* 多路复用器关闭后，所有注册在上面的Channel和Pipe等资源都会被自动去注册并关闭，所以不需要重复释放资源。

\*/

**if**(selector!=**null**){

**try** {

selector.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**private** **void** handleInput(SelectionKey key) **throws** IOException{

**if**(key.isValid()){

//处理新接入的请求消息

//通过SelectionKey的操作位进行判断即可获知网络事件类型

**if**(key.isAcceptable()){

//Accept the new connection

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

SocketChannel sc=ssc.accept();

//-----以上操作相当于完成了TCP的三次握手，TCP物理链路正式建立------

//将新创建的SocketChannel设置为异步非阻塞，同时也可以对其TCP参数进行设置，例如TCP接收和发送缓冲区的大小等。

sc.configureBlocking(**false**);

//Add the new connection to the selector

sc.register(selector, SelectionKey.OP\_READ);

}

**if**(key.isReadable()){

//Read the data

SocketChannel sc=(SocketChannel) key.channel();

//由于实现我们得知客户端发送的码流大小，作为例程，我们开辟一个1K的缓冲区

ByteBuffer readBuffer=ByteBuffer.allocate(1024);

//由于已经设置SocketChannel为异步非阻塞模式，因此它的read是非阻塞的。

**int** readBytes=sc.read(readBuffer);

/\*

\* readBytes>0 读到了字节，对字节进行编解码；

\* readBytes=0 没有读取到字节，属于正常场景，忽略；

\* readByte=-1 链路已经关闭，需要关闭SocketChannel，释放资源

\*/

**if**(readBytes>0){

//将缓冲区当前的limit设置为position，position设置为0，用于后续对缓冲区的读取操作。

readBuffer.flip();

//根据缓冲区可读的字节个数创建字节数组

**byte**[] bytes=**new** **byte**[readBuffer.remaining()];

//调用ByteBuffer的get操作将缓冲区可读的字节数组复制到新创建爱你的字节数组中

readBuffer.get(bytes);

String body=**new** String(bytes, "UTF-8");

System.out.println("The time server receive order: "+body);

//如果请求指令是"QUERY TIME ORDER"则把服务器的当前时间编码后返回给客户端

String currentTime="QUERY TIME ORDER".equalsIgnoreCase(body) ? **new** Date(

System.currentTimeMillis()).toString() : "BAD ORDER";

doWrite(sc,currentTime);

}**else** **if**(readBytes<0){

　　　　　　　　　　　　 //对端链路关闭

key.cancel();

sc.close();

}**else**{

//读到0字节，忽略

}

}

}

}

**private** **void** doWrite(SocketChannel channel,String response) **throws** IOException{

**if**(response!=**null**&& response.trim().length()>0){

**byte**[] bytes=response.getBytes();

ByteBuffer writeBuffer=ByteBuffer.*allocate*(bytes.length);

//调用ByteBuffer的put操作将字节数组复制到缓冲区

writeBuffer.put(bytes);

writeBuffer.flip();

channel.write(writeBuffer);

/\*

\* 需要指出的是，由于SocketChannel是异步非阻塞的，它并不保证一次性能够把需要发送的字节数组发送完，

\* 此时会出现“写半包”问题，我们需要注册写操作，不断轮询Selector，将没有发送完毕的ByteBuffer发送完毕，

\* 可以通过ByteBuffer的hasRemaining()方法判断消息是否发送完成。

\* 此处仅仅是各简单的入门级例程，没有演示如何处理“写半包”场景，后面会说到。

\*/

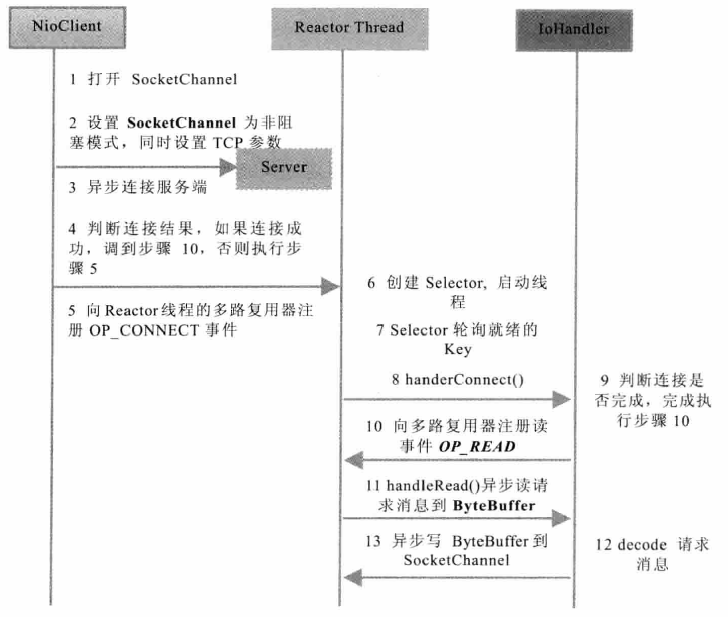
}

}

}

#### NIO客户端序列图

NIO客户端创建序列图如图所示：



步骤一：打开SocketChannel，绑定客户端本地地址（可选，默认系统会随机分配一个可用的本地地址）

SocketChannel clientChannel=SocketChannel.open();

步骤二：设置SocketChannel为非阻塞模式，同时设置客户端连接的TCP参数。

clientChannel.configureBlocking(**false**);

socket.setReuseAddress(**true**);

socket.setReceiveBufferSize(BUFFER\_SIZE);

socket.setSendBufferSize(BUFFER\_SIZE);

步骤三：异步连接服务端。

**boolean** connected=clientChannel.connect(**new** InetSocketAddress("ip",port));

步骤四：判断是否连接成功，如果连接成功，则直接注册读状态位到多路复用器中，如果当前没有连接成功（异步连接，返回false，说明客户端已经发送sync包，服务端没有返回ack包，物理链路还没有建立）。

**if**(connected){

clientChannel.register(selector,SelectionKey.OP\_READ,ioHandler);

}**else**{

clientChannel.register(selector,SelectionKey.OP\_CONNECT,ioHandler);

}

步骤五：向Reactor线程的多路复用器注册OP\_CONNECT状态位，监听服务端的TCP ACK应答。

clientChannel.register(selector,SelectionKay.OP\_CONNECT,ioHandler);

步骤六：创建Reactor线程，创建多路复用器并启动线程。

Selector selector=Selector.open();

**new** Thread(**new** ReactorTask()).start();

步骤七：多路复用器在线程run方法的无限循环体内轮询准备就绪的key。

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

Set selectedKeys=selector.selectedKeys();

Iterator it=selectedKeys.iterator();

**while**(it.hasNext()){

SelectionKey key=(SelectionKey)it.next();

//...deal with I/O event...

}

步骤八：接收connect事件进行处理。

**if**(key.isConnectable()){

//handlerConnect();

}

步骤九：判断连接结果，如果连接成功，注册读事件到多路复用器。

**if**(channel.finishConnect()){

registerRead();

}

步骤十：注册读事件到多路复用器。

clientChannel.register(selector,SelectionKey.OP\_READ,ioHandler);

步骤十一：异步读客户端请求消息到缓冲区。

**int** readNumber=channel.read(receivedBuffer);

步骤十二：对ByteBuffer进行编解码，如果有半包消息接收缓冲区Reset，继续读取后续的报文，将解码成功的消息封装成Task,投递到业务线程池中，进行业务逻辑编排。

Object message=**null**;

**while**(buffer.hasRemain()){

byteBuffer.mark();

Object message=decode(byteBuffer);

**if**(message==**null**){

　　byteBuffer.reset();

**break**;

}

messageList.add(message);

}

**if**(!byteBuffer.hasRemain()){

byteBuffer.clear();

}**else**{

byteBuffer.compact();

}

**if**(messageList!=**null** & !messageList.isEmpty()){

**for**(Object messageE:messageList){

　　handlerTask(messageE);

}

}

步骤十三：将POJO对象encode成ByteBuffer,调用SocketChannel的异步write接口，将消息异步发送给客户端。

socketChannel.wirte(buffer);

#### NIO创建的TimeClient源码分析

**package** joanna.yan.nio;

**public** **class** TimeClient {

**public** **static** **void** main(String[] args) {

**int** port=9090;

**if**(args!=**null**&&args.length>0){

**try** {

port=Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

**new** Thread(**new** TimeClientHandle("127.0.0.1", port),"TimClient-001").start();

}

}

**package** joanna.yan.nio;

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.ClosedChannelException;

**import** java.nio.channels.SelectionKey;

**import** java.nio.channels.Selector;

**import** java.nio.channels.SocketChannel;

**import** java.util.Iterator;

**import** java.util.Set;

/\*\*

\* 处理异步连接和读写操作

\* **@author** Joanna.Yan

\* **@date** 2017年11月6日下午4:33:14

\*/

**public** **class** TimeClientHandle **implements** Runnable{

**private** String host;

**private** **int** port;

**private** Selector selector;

**private** SocketChannel socketChannel;

**private** **volatile** **boolean** stop;

/\*\*

\* 初始化NIO的多路复用器和SocketChannel对象

\* **@param** host

\* **@param** port

\*/

**public** TimeClientHandle(String host,**int** port){

**this**.host=host==**null** ? "127.0.0.1" : host;

**this**.port=port;

**try** {

selector=Selector.*open*();

socketChannel=SocketChannel.*open*();

//设置为异步非阻塞模式，同时还可以设置SocketChannel的TCP参数。例如接收和发送的TCP缓冲区大小

socketChannel.configureBlocking(**false**);

} **catch** (IOException e) {

e.printStackTrace();

System.*exit*(1);

}

}

@Override

**public** **void** run() {

**try** {

doConnect();

} **catch** (IOException e) {

e.printStackTrace();

System.exit(1);

}

**while**(!stop){

**try** {

selector.select(1000);

Set<SelectionKey> selectedKeys=selector.selectedKeys();

Iterator<SelectionKey> it=selectedKeys.iterator();

SelectionKey key=**null**;

**while**(it.hasNext()){//轮询多路复用器Selector，当有就绪的Channel时

key=it.next();

it.remove();

**try** {

handleInput(key);

} **catch** (Exception e) {

**if**(key!=**null**){

key.cancel();

**if**(key.channel()!=**null**){

key.channel().close();

}

}

}

}

} **catch** (IOException e) {

e.printStackTrace();

System.exit(1);

}

}

//多路复用器关闭后，所有注册在上面的Channel和Pipe等资源都会被自动注册并关闭，所以不需要重复释放资源。

/\*

\* 由于多路复用器上可能注册成千上万的Channel或者pipe，如果一一对这些资源进行释放显然不合适。

\* 因此，JDK底层会自动释放所有跟此多路复用器关联的资源。

\*/

**if**(selector!=**null**){

**try** {

selector.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**private** **void** handleInput(SelectionKey key) **throws** ClosedChannelException, IOException {

**if**(key.isValid()){

//判断是否连接成功

SocketChannel sc=(SocketChannel) key.channel();

**if**(key.isConnectable()){//处于连接状态，说明服务器已经返回ACK应答消息

**if**(sc.finishConnect()){//对连接结果进行判断

/\*

\* 将SocketChannel注册到多路复用器上，注册SelectionKey.OP\_READ操作位，

\* 监听网络读操作，然后发送请求消息给服务端。

\*/

sc.register(selector, SelectionKey.OP\_READ);

doWrite(sc);

}**else**{

System.exit(1);//连接失败，进程退出

}

}

**if**(key.isReadable()){

//开辟缓冲区

ByteBuffer readBuffer=ByteBuffer.allocate(1024);

//异步读取

**int** readBytes=sc.read(readBuffer);

**if**(readBytes>0){

readBuffer.flip();

**byte**[] bytes=**new** **byte**[readBuffer.remaining()];

readBuffer.get(bytes);

String body=**new** String(bytes, "UTF-8");

System.out.println("Now is: "+body);

**this**.stop=**true**;

}**else** **if**(readBytes<0){

//对端链路关闭

key.cancel();

sc.close();

}**else**{

//读到0字节，忽略

}

}

}

}

**private** **void** doConnect() **throws** IOException {

//如果直接连接成功，则将SocketChannel注册到多路复用器Selector上，发送请求消息，读应答

**if**(socketChannel.connect(**new** InetSocketAddress(host, port))){

socketChannel.register(selector, SelectionKey.***OP\_READ***);

doWrite(socketChannel);

}**else**{

/\*

\* 如果没有直接连接成功，则说明服务端没有返回TCP握手应答信息，但这并不代表连接失败，

\* 我们需要将SocketChannel注册到多路复用器Selector上，注册SelectionKey.OP\_CONNECT，

\* 当服务端返回TCP syn-ack消息后，Selector就能轮询到整个SocketChannel处于连接就绪状态。

\*/

socketChannel.register(selector, SelectionKey.***OP\_CONNECT***);

}

}

**private** **void** doWrite(SocketChannel sc) **throws** IOException {

**byte**[] req="QUERY TIME ORDER".getBytes();

ByteBuffer writeBuffer=ByteBuffer.*allocate*(req.length);

//写入到发送缓冲区中

writeBuffer.put(req);

writeBuffer.flip();

//由于发送是异步的，所以会存在"半包写"问题，此处不赘述

sc.write(writeBuffer);

**if**(!writeBuffer.hasRemaining()){//如果缓冲区中的消息全部发送完成

System.***out***.println("Send order 2 server succeed.");

}

}

}

通过源码对比分析发现，NIO编程难度确实比同步阻塞BIO大很多，此处我们的NIO例程并没有考虑“半包读”和“半包写”，如果加上这些，代码会更加复杂。NIO代码既然这么复杂，为什么它的应用却越来越广泛呢，使用NIO编程的优点总结如下：

1. 客户端发起的连接操作是异步的，可以通过多路复用器注册OP\_CONNECT等待后续结果，不需要像之前的客户端那样被同步阻塞。
2. SocketChannel的读写操作都是异步的，如果没有可读写的数据它不会同步等待，直接返回，这样I/O通信线程就可以处理其他的链路，不需要同步等待这个链路可用。
3. 线程模型的优化：由于JDK的Selector在Linux等主流操作系统上通过epoll实现，它没有连接句柄数的限制（只受限于操作系统的最大句柄数或者对单个进程的句柄限制），这意味着一个Selector线程可以同时处理成千上万个客户端连接，而且性能不会随着客户端的增加而线性下降，因此，它非常适合做高性能、高负载的网络服务器。

JDK1.7升级了NIO类库，升级后的NIO类库被称为NIO 2.0。引入注目的是，Java正式提供了异步文件I/O操作，同时提供了与UNIX网络编程事件驱动I/O对应的AIO。

### AIO编程

NIO2.0引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。异步通道提供两种方式获取操作结果

1. 通过java.util.concurrent.Future类来表示异步操作的结果；
2. 在执行异步操作的时候传入一个java.nio.channels。

CompletionHandler接口的实现类作为操作完成的回调

NIO2.0的异步套接字通道是真正的异步非阻塞I/O，它对UNIX网络编程中的事件驱动I/O（AIO），它不需要通过多路复用器（Selector）对注册的通道进行轮询操作即可实现异步读写，从而简化了NIO的编程模型。

下面通过代码来熟悉NIO2.0 AIO的相关类库，仍旧以时间服务器为例程进行讲解。

#### AIO创建的TimeServer源码分析

**package** joanna.yan.aio;

**public** **class** TimeServer {

**public** **static** **void** main(String[] args) {

**int** port = 9090;

**if** (args != **null** && args.length > 0) {

**try** {

port = Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

AsyncTimeServerHandler timeServer = **new** AsyncTimeServerHandler(port);

**new** Thread(timeServer, "AIO-AsyncTimeServerHandler-001").start();

}

}

**package** joanna.yan.aio;

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.nio.channels.AsynchronousServerSocketChannel;

**import** java.util.concurrent.CountDownLatch;

**public** **class** AsyncTimeServerHandler **implements** Runnable {

**private** **int** port;

CountDownLatch latch;

AsynchronousServerSocketChannel asynchronousServerSocketChannel;

**public** AsyncTimeServerHandler(**int** port) {

**this**.port = port;

**try** {

// 创建一个异步的服务端通道AsynchronousServerSocketChannel

asynchronousServerSocketChannel = AsynchronousServerSocketChannel.*open*();

asynchronousServerSocketChannel.bind(**new** InetSocketAddress(port));

System.***out***.println("The time server is start in port: " + port);

} **catch** (IOException e) {

e.printStackTrace();

}

}

@Override

**public** **void** run() {

/\*

\* 初始化CountDownLatch对象。 它的作用是，在完成一组正在执行的操作之前，允许当前的线程一直阻塞。

\* 在本例中，我们让线程在此阻塞，防止服务器执行完成退出。

\* 在实际项目应用中，不需要启动独立的线程来处理AsynchronousServerSocketChannel，这里仅仅是个demo演示。

\*/

latch = **new** CountDownLatch(1);

doAccept();

**try** {

latch.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 接收客户端的连接。 由于这里是异步操作。我们可以传递一个CompletionHandler<AsynchronousSocketChannel,?

\* super A>类型 的handler实例接收accept操作成功的通知消息

\*/

**public** **void** doAccept() {

asynchronousServerSocketChannel.accept(**this**, **new** AcceptCompletionHandler());

}

}

**package** joanna.yan.aio;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.AsynchronousSocketChannel;

**import** java.nio.channels.CompletionHandler;

/\*\*

\* 接收accept操作成功的通知消息

\*

\* **@author** Administrator

\*

\*/

**public** **class** AcceptCompletionHandler **implements** CompletionHandler<AsynchronousSocketChannel, AsyncTimeServerHandler> {

@Override

**public** **void** completed(AsynchronousSocketChannel result, AsyncTimeServerHandler attachment) {

/\*

\* 疑惑：既然已经接收客户端成功了，为什么还要再次调用accept方法呢？

\* 原因：当我们调用AsynchronousServerSocketChannel的accept方法后，如果有新的客户端连接接入，

\* 系统将回调我们传入的CompletionHandler实例的completed方法，表示新的客户端已经接入成功，

\* 因为一个AsynchronousServerSocketChannel可以接收成千上万个客户端，所以我们需要继续调用它的accep方法，

\* 接收其他的客户端连接，最终形成一个循环。 每当接收一个客户读连接成功之后，再异步接收新的客户端连接。

\*/

attachment.asynchronousServerSocketChannel.accept(attachment, **this**);

ByteBuffer buffer = ByteBuffer.*allocate*(1024);

/\*

\* 参数一：接收缓冲区，用于从异步Channel中读取数据包； 参数二：异步Channel携带的附件，通知回调的时候作为入参使用；

\* 参数三：接收通知回调的业务handler

\*/

result.read(buffer, buffer, **new** ReadCompletionHandler(result));

}

@Override

**public** **void** failed(Throwable exc, AsyncTimeServerHandler attachment) {

exc.printStackTrace();

attachment.latch.countDown();

}

}

**package** joanna.yan.aio;

**import** java.io.IOException;

**import** java.io.UnsupportedEncodingException;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.AsynchronousSocketChannel;

**import** java.nio.channels.CompletionHandler;

**import** java.util.Date;

/\*\*

\* 主要用于读取半包消息和发送应答。 本例不对半包读写进行具体说明，在后面的Netty半包处理中会介绍。

\*

\* **@author** Administrator

\*/

**public** **class** ReadCompletionHandler **implements** CompletionHandler<Integer, ByteBuffer> {

**private** AsynchronousSocketChannel channel;

**public** ReadCompletionHandler(AsynchronousSocketChannel channel) {

**if** (**this**.channel == **null**) {

**this**.channel = channel;

}

}

@Override

**public** **void** completed(Integer result, ByteBuffer attachment) {

// 为后续从缓冲区读取数据做准备

attachment.flip();

**byte**[] body = **new** **byte**[attachment.remaining()];

attachment.get(body);

**try** {

String req = **new** String(body, "UTF-8");

System.***out***.println("The time server receive order : " + req);

String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(req)

? **new** Date(System.*currentTimeMillis*()).toString()

: "BAD　ORDER";

doWrite(currentTime);

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

/\*\*

\* 发生异常的时候调用。 对异常Throwable进行判断，如果是I/O异常，就关闭链路，释放资源； 如果是其它异常，按照业务自己的逻辑进行处理。

\* 本例作为简单demo，没有对异常进行分类判断，只要发生了读写异常，就关闭链路，释放资源。

\*/

@Override

**public** **void** failed(Throwable exc, ByteBuffer attachment) {

**try** {

**this**.channel.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**private** **void** doWrite(String currentTime) {

**if** (currentTime != **null** && currentTime.trim().length() > 0) {

**byte**[] bytes = currentTime.getBytes();

ByteBuffer writeBuffer = ByteBuffer.*allocate*(bytes.length);

writeBuffer.put(bytes);

writeBuffer.flip();

channel.write(writeBuffer, writeBuffer, **new** CompletionHandler<Integer, ByteBuffer>() {

@Override

**public** **void** completed(Integer result, ByteBuffer buffer) {

// 如果没有发送完成，继续发送

**if** (buffer.hasRemaining()) {

channel.write(buffer, buffer, **this**);

}

}

@Override

**public** **void** failed(Throwable exc, ByteBuffer attachment) {

**try** {

channel.close();

} **catch** (IOException e) {

// ingnore on close

}

}

});

}

}

}

#### AIO创建的TimeClient源码分析

**package** joanna.yan.aio;

**public** **class** TimeClient {

**public** **static** **void** main(String[] args) {

**int** port = 9090;

**if** (args != **null** && args.length > 0) {

**try** {

port = Integer.*valueOf*(args[0]);

} **catch** (Exception e) {

// 采用默认值

}

}

/\*

\* 通过一个独立的I/O线程创建异步时间服务器客户端handler。

\* 在实际项目中，我们不需要独立的线程创建异步连接对象，因为底层都是通过JDK的系统回调实现的。

\*/

**new** Thread(**new** AsyncTimeClientHandler("127.0.0.1", port), "AIO-AsyncTimeClientHandle-001").start();

/\*

\* 需要指出的是，正如之前的NIO例程，我们并没有完整的处理网络的半包读写，在对例程进行功能测试的是还没有问题，

\* 但是，如果对代码稍加改造，进行压力或者性能测试，就会发现输出结果存在问题。 这里只集中将NIO的入门知识，后面会详细讲到半包读写

\*/

}

}

**package** joanna.yan.aio;

**import** java.io.IOException;

**import** java.io.UnsupportedEncodingException;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.AsynchronousSocketChannel;

**import** java.nio.channels.CompletionHandler;

**import** java.util.concurrent.CountDownLatch;

**public** **class** AsyncTimeClientHandler **implements** CompletionHandler<Void, AsyncTimeClientHandler>, Runnable {

**private** AsynchronousSocketChannel client;

**private** String host;

**private** **int** port;

**private** CountDownLatch latch;

**public** AsyncTimeClientHandler(String host, **int** port) {

**this**.host = host;

**this**.port = port;

**try** {

client = AsynchronousSocketChannel.*open*();

} **catch** (IOException e) {

e.printStackTrace();

}

}

@Override

**public** **void** completed(Void result, AsyncTimeClientHandler attachment) {

**byte**[] req = "QUERY TIME ORDER".getBytes();

ByteBuffer writeBuffer = ByteBuffer.*allocate*(req.length);

writeBuffer.put(req);

writeBuffer.flip();

client.write(writeBuffer, writeBuffer, **new** CompletionHandler<Integer, ByteBuffer>() {// 用于写操作完成后的回调

@Override

**public** **void** completed(Integer result, ByteBuffer buffer) {

**if** (buffer.hasRemaining()) {

client.write(buffer, buffer, **this**);

} **else** {

ByteBuffer readBuffer = ByteBuffer.*allocate*(1024);

client.read(readBuffer, readBuffer, **new** CompletionHandler<Integer, ByteBuffer>() {// 当读取完成被JDK回调时，构造应答消息。

@Override

**public** **void** completed(Integer result, ByteBuffer buffer) {

buffer.flip();

**byte**[] bytes = **new** **byte**[buffer.remaining()];

buffer.get(bytes);

String body;

**try** {

body = **new** String(bytes, "UTF-8");

System.***out***.println("Now is : " + body);

latch.countDown();

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

@Override

**public** **void** failed(Throwable exc, ByteBuffer attachment) {

**try** {

client.close();

// 让AsyncTimeClientHandler线程执行完毕，客户端退出执行

latch.countDown();

} **catch** (IOException e) {

// ingnore on close

}

}

});

}

}

@Override

**public** **void** failed(Throwable exc, ByteBuffer attachment) {

**try** {

client.close();

latch.countDown();

} **catch** (IOException e) {

// ingnore on close

}

}

});

}

@Override

**public** **void** failed(Throwable exc, AsyncTimeClientHandler attachment) {

exc.printStackTrace();

**try** {

client.close();

latch.countDown();

} **catch** (IOException e) {

// ingnore on close

}

}

@Override

**public** **void** run() {

// 创建CountDownLatch进行等待，防止异步操作没有执行完成线程就退出。

latch = **new** CountDownLatch(1);

/\*

\* 参数二：AsynchronousSocketChannel的附件，用于回调通知时作为入参被传递，调用者可以自定义

\* 参数三：异步参数回调通知接口，由调用者实现。

\*/

client.connect(**new** InetSocketAddress(host, port), **this**, **this**);

**try** {

latch.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**try** {

client.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

异步SocketChannel是被动执行对象，我们不需要像NIO编程那样创建一个独立I/O线程来处理读写操作。对于AsynchronousServerSocketChannel和 AsynchronousSocketChannel，它们都由JDK底层的线程池负责回调并驱动读写操作。正因为如此，基于NIO2.0新的异步非阻塞Channel进行编程比NIO编程更为简单。

后面，我们将对前面讲到的4种I/O进行概念澄清和比较，让大家从整体上掌握这些I/O模型的差异。以便在未来的工作中能够根据产品的实际情况选择合适的I/O模型。

### 4种I/O的对比与选型

为了防止由于对一些技术概念和术语的理解或者叫法不一致而引起歧义，这里对涉及到的专业术语或者技术用语做下声明：如果它们与其他一些地方的称呼不一致，请以本解释为准。

1. **异步非阻塞I/O**

很多人喜欢将JDK1.4提供的NIO框架成为异步非阻塞I/O，但是，如果严格按照UNIX网络编程模型和JDK的实现进行区分，实际上它只能被称为非阻塞I/O，不能叫异步非阻塞I/O。在早期的JDK1.4和1.5 update10版本之前，JDK的Selector基于select/poll模型实现，它是基于I/O复用技术的非阻塞I/O，不是异步I/O。在JDK1.5 update10和Linux core2.6以上版本，Sun优化了Selector实现，它在底层使用epoll替换了select/poll，上层的API并没有变化，可以认为是JDK NIO的一次性能优化，但是它仍旧没有改变I/O的模型。

由JDK1.7提供的NIO 2.0，新增了异步的套接字通道，它是真正的异步I/O，在异步I/O操作的时候可以传递信号变量，当操作完成之后会回调相关的方法，异步I/O也被称为AIO。

NIO类库支持非阻塞读和写操作，相比于之前的同步阻塞读和写，它是异步的，NIO类库支持非阻塞读和写操作，相比于之前的同步阻塞读和写，它是一部的，因此很多人习惯称NIO为异步非阻塞I/O，包括很多介绍NIO编程的书籍也沿用了这个说法。为了符合大家的习惯，这里也会将NIO称为异步非阻塞I/O或者非阻塞I/O。请大家理解，不要过分纠结在一些技术术语的咬文嚼字上

1. **多路复用器Selector**

几乎所有的中文技术书籍都将Selector翻译为选择器，但是实际上我认为这样的翻译并不恰当，选择器仅仅是字面上的意思，体现不出Selector的功能和特点。

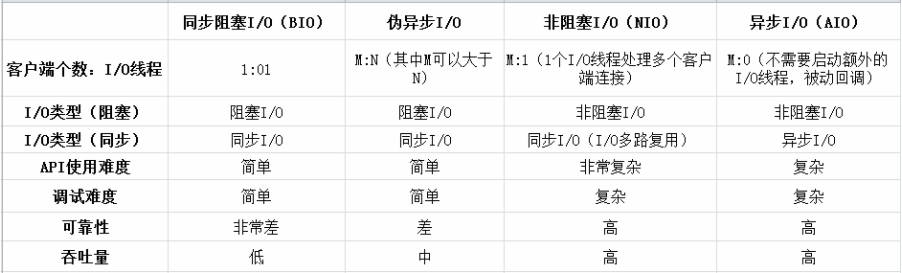
前面介绍过Java NIO的实现关键是多路复用I/O技术，多路复用的核心就是通过Selector来轮询注册在其上的Channel，当发现某个或者多个Channel处于就绪状态后，从阻塞状态返回就绪的Channel的选择键集合，进行I/O操作。由于多路复用器是NIO实现非阻塞I/O的关键，它又是主要通过Selector实现的，所以这里将Selector翻译为多路复用器，与其他技术书籍所说的选择器是同一个东西，请大家了解。

1. **伪异步I/O**

伪异步的概念完全来源于实现。在JDK NIO编程没有流行之前，为了解决Tomcat通信线程同步I/O导致业务线程被挂住的问题，大家想到了一个办法：在通信线程和业务线程之前做个缓冲区，这个缓冲区用于隔离I/O线程和业务线程间的直接访问，这样业务线程就不会被I/O线程阻塞。而对于后端的业务侧来说，将消息或者Task放到线程池后就返回了，它不再直接访问I/O线程或者进行I/O读写，这样就不会被同步阻塞。类似的设计还包括前端启动一组线程，将接收到的客户端封装成Task，放到后端的线程池执行，用于解决一连接一线程问题。像这样通过线程池做缓冲区的做法，这里习惯于称它为伪异步I/O，而官方并没有伪异步I/O这种说法，请大家注意。

#### 不同I/O模型对比

不同的I/O模型由于线程模型、API等差别很大，所以用法的差异也非常大。



实际开发中，具体选择什么样的I/O模型或者NIO框架，完全基于业务的实际应用场景和性能诉求，如果客户端并发连接数不多，周边对接的网元不多，服务器的负载也不重，那就完全没必要选择NIO做服务器；如果是相反情况，那就要考虑选择合适的NIO框架进行开发。

#### 选择Netty的理由

开发出高质量的NIO程序并不是一件简单的事情，除去NIO固有的复杂性和BUG不谈，作为一个NIO服务器，需要能够处理网络的闪断、客户端的重复接入、客户端的安全认证、消息的编解码、半包读写等情况，如果你没有足够的NIO编程经验积累，一个NIO框架的稳定往往需要半年甚至更长的时间。更为糟糕的是，一旦在生产环境中发生问题，往往会导致跨节点的服务调用中断，严重的可能会导致整个集群环境都不可用，需要重启服务器，这种非正常停机会带来巨大的损失。

从可维护性角度看，由于NIO采用了异步非阻塞编程模型，而且是一个I/O线程处理多条链路，它的调试和跟踪非常麻烦，特别是生产环境中的问题，我们无法进行有效的调试和跟踪，往往只能靠一些日志来辅助分析，定位难度很大。

1. **不选择Java原生NIO编程的原因**

现在我们总结一下我们不建议开发者直接使用JDK的NIO类库进行开发，具体原因如下。

1. NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。
2. 需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程设计到Reactor模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的NIO程序。
3. 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。
4. JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU100%，官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生频率降低了一些而已，它并没有被根本解决。

由于上述原因，在大多数场景下，不建议大家直接使用JDK的NIO类库，除非你精通NIO编程或者有特殊的需求。在绝大多数的业务场景中，我们可以使用NIO框架Netty来进行NIO编程，它既可以作为客户端也可以作为服务端，同时支持UDP和异步文件传输，功能非常强大。

1. **为什么选择Netty**

Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商业项目验证，例如Hadoop的RPC框架avro使用Netty作为底层通信框架；很多其他业界主流的RPC框架，也使用Netty来构建高性能的异步通信能力。

Netty的优点总结如下：

1. API使用简单，开发门槛低；
2. 功能强大，预置了多种编解码功能，支持多种主流协议；
3. 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；
4. 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；
5. 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；
6. 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；
7. 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它已经完全能够满足不同行业的商业应用了。

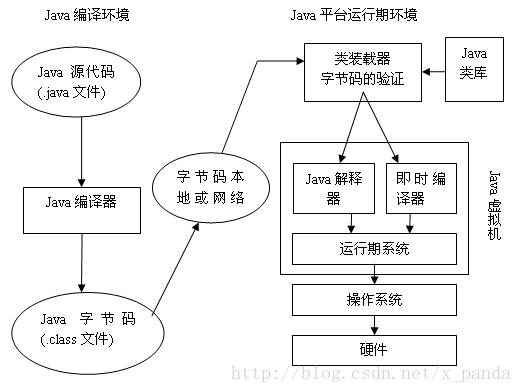
正是因为这些优点，Netty逐渐成为Java NIO编程的首选框架。

## 类的加载和反射

### JVM

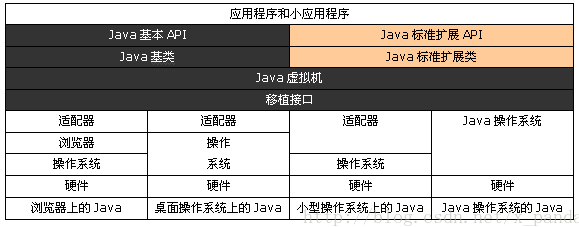
说起Java，人们首先想到的是Java编程语言，然而事实上，Java是一种技术，它由四方面组成：Java编程语言、Java类文件格式、Java虚拟机和Java应用程序接口(Java API)。

它们的关系如下图所示：



运行期环境代表着Java平台，开发人员编写Java代码(.java文件)，然后将之编译成字节码(.class文件)，再然后字节码被装入内存，一旦字节码进入虚拟机，它就会被解释器解释执行，或者是被即时代码发生器有选择的转换成机器码执行。

Java平台由Java虚拟机和Java应用程序接口搭建，Java语言则是进入这个平台的通道，用Java语言编写并编译的程序可以运行在这个平台上。



在Java平台的结构中, 可以看出，Java虚拟机(JVM) 处在核心的位置，是程序与底层操作系统和硬件无关的关键。它的下方是移植接口，移植接口由两部分组成：适配器和Java操作系统, 其中依赖于平台的部分称为适配器；JVM 通过移植接口在具体的平台和操作系统上实现；在JVM 的上方是Java的基本类库和扩展类库以及它们的API， 利用Java API编写的应用程序(application) 和小程序(Java applet) 可以在任何Java平台上运行而无需考虑底层平台, 就是因为有Java虚拟机(JVM)实现了程序与操作系统的分离，从而实现了Java 的平台无关性。

JVM在它的生存周期中有一个明确的任务，那就是运行Java程序，因此当Java程序启动的时候，就产生JVM的一个实例；当程序运行结束的时候，该实例也跟着消失了。下面我们从JVM的体系结构和它的运行过程这两个方面来对它进行比较深入的研究。

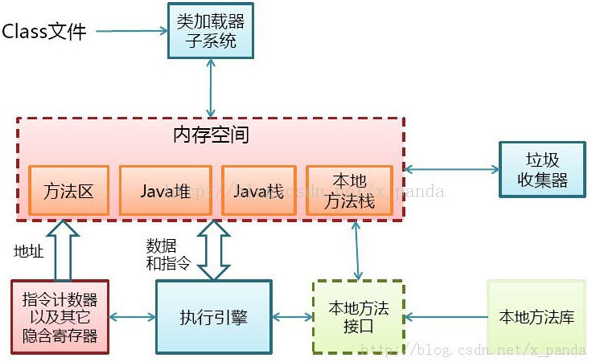
1. Java虚拟机的体系结构

每个JVM都有两种机制：

1. 类装载子系统：装载具有适合名称的类或接口
2. 执行引擎：负责执行包含在已装载的类或接口中的指令

每个JVM都包含：

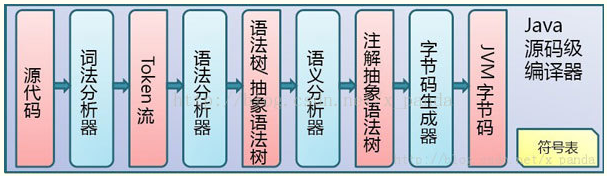
方法区、Java堆、Java栈、本地方法栈、指令计数器及其他隐含寄存器



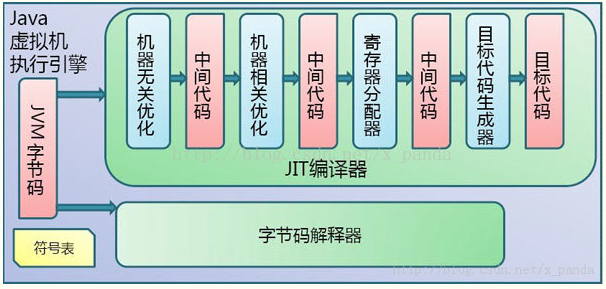
1. Java代码编译和执行的整个过程

也正如前面所说，Java代码的编译和执行的整个过程大概是：开发人员编写Java代码(.java文件)，然后将之编译成字节码(.class文件)，再然后字节码被装入内存，一旦字节码进入虚拟机，它就会被解释器解释执行，或者是被即时代码发生器有选择的转换成机器码执行。

Java代码编译是由Java源码编译器来完成，也就是Java代码到JVM字节码（.class文件）的过程。 流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：



Java代码编译和执行的整个过程包含了以下三个重要的机制:

Java源码编译机制

类加载机制

类执行机制

1. Java源码编译机制

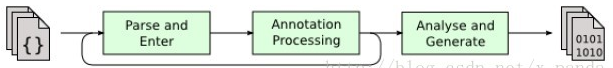
Java 源码编译由以下三个过程组成：

①分析和输入到符号表

②注解处理

③语义分析和生成class文件

流程图如下所示：



最后生成的class文件由以下部分组成：

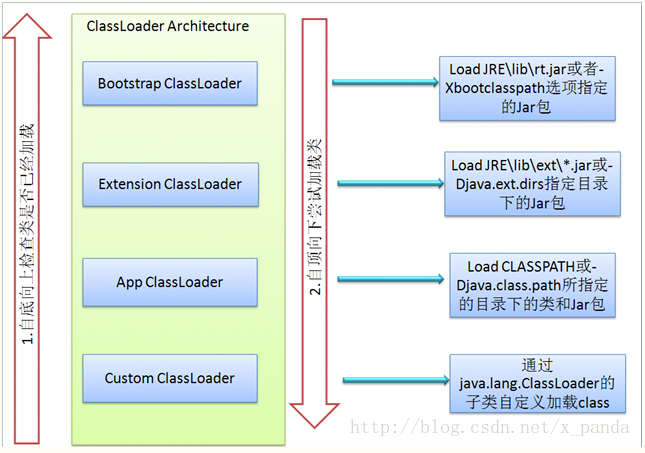
1. 构信息：包括class文件格式版本号及各部分的数量与大小的信息

②元数据：对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池

③方法信息：对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息

1. 类加载机制

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



①Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

②Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

③App ClassLoader

负责记载classpath中指定的jar包及目录中class

1. Custom ClassLoader

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

1. 类执行机制

JVM是基于堆栈的虚拟机。JVM为每个新创建的线程都分配一个堆栈.也就是说,对于一个Java程序来说，它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。

JVM执行class字节码，线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：



1. JVM内存管理及垃圾回收机制

JVM内存结构分为：方法区（method），栈内存（stack），堆内存（heap），本地方法栈（java中的jni调用），结构图如下所示：



1. 堆内存（heap）

所有通过new创建的对象的内存都在堆中分配，其大小可以通过-Xmx和-Xms来控制。

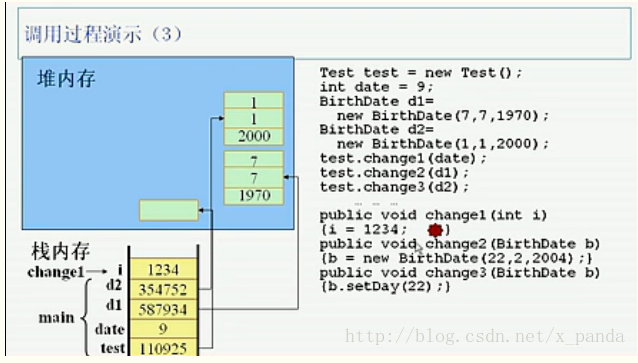
操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的delete语句才能正确的释放本内存空间。但由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。这时由new分配的内存，一般速度比较慢，而且容易产生内存碎片，不过用起来最方便。另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，它不是在堆，也不是在栈，而是直接在进程的地址空间中保留一块内存，虽然这种方法用起来最不方便，但是速度快，也是最灵活的。堆内存是向高地址扩展的数据结构，是不连续的内存区域。由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

1. 栈内存（stack）

在Windows下, 栈是向低地址扩展的数据结构，是一块连续的内存区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是固定的（是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。 由系统自动分配，速度较快。但程序员是无法控制的。

**堆内存与栈内存需要说明：**

基础数据类型直接在栈空间分配，方法的形式参数，直接在栈空间分配，当方法调用完成后从栈空间回收。引用数据类型，需要用new来创建，既在栈空间分配一个地址空间，又在堆空间分配对象的类变量 。方法的引用参数，在栈空间分配一个地址空间，并指向堆空间的对象区，当方法调用完成后从栈空间回收。局部变量new出来时，在栈空间和堆空间中分配空间，当局部变量生命周期结束后，栈空间立刻被回收，堆空间区域等待GC回收。方法调用时传入的literal参数，先在栈空间分配，在方法调用完成后从栈空间收回。字符串常量、static在DATA区域分配，this在堆空间分配。数组既在栈空间分配数组名称，又在堆空间分配数组实际的大小。



1. 本地方法栈（java中的jni调用）

用于支持native方法的执行，存储了每个native方法调用的状态。对于本地方法接口，实现JVM并不要求一定要有它的支持，甚至可以完全没有。Sun公司实现Java本地接口(JNI)是出于可移植性的考虑，当然我们也可以设计出其它的本地接口来代替Sun公司的JNI。但是这些设计与实现是比较复杂的事情，需要确保垃圾回收器不会将那些正在被本地方法调用的对象释放掉。

1. 方法区（method）

它保存方法代码(编译后的java代码)和符号表。存放了要加载的类信息、静态变量、final类型的常量、属性和方法信息。JVM用持久代（Permanet Generation）来存放方法区，可通过-XX:PermSize和-XX:MaxPermSize来指定最小值和最大值。

**垃圾回收机制**

堆里聚集了所有由应用程序创建的对象，JVM也有对应的指令比如 new, newarray, anewarray和multianewarray，然并没有向 C++ 的 delete，free 等释放空间的指令，Java的所有释放都由 GC 来做，GC除了做回收内存之外，另外一个重要的工作就是内存的压缩，这个在其他的语言中也有类似的实现，相比 C++ 不仅好用，而且增加了安全性，当然她也有弊端，比如性能这个大问题。

1. Java虚拟机的运行过程示例

上面对虚拟机的各个部分进行了比较详细的说明，下面通过一个具体的例子来分析它的运行过程。

虚拟机通过调用某个指定类的方法main启动，传递给main一个字符串数组参数，使指定的类被装载，同时链接该类所使用的其它的类型，并且初始化它们。例如对于程序：

**public** **class** HelloApp {

**public** **static** **void** main(String[] args){

System.***out***.println("Hello World!");

**for** (**int** i = 0; i < args.length; i++) {

System.***out***.println(args[i]);

}

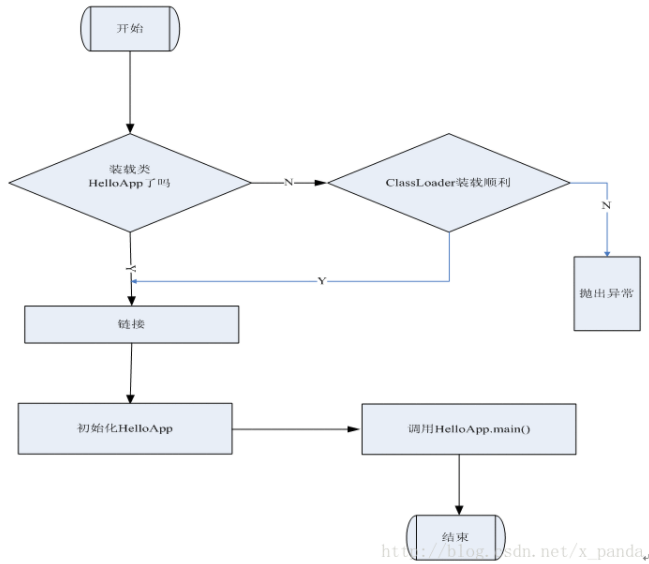
}

}

编译后在命令行模式下键入： java HelloApp run virtual machine

将通过调用HelloApp的方法main来启动java虚拟机，传递给main一个包含三个字符串"run"、"virtual"、"machine"的数组。现在我们略述虚拟机在执行HelloApp时可能采取的步骤。

开始试图执行类HelloApp的main方法，发现该类并没有被装载，也就是说虚拟机当前不包含该类的二进制代表，于是虚拟机使用ClassLoader试图寻找这样的二进制代表。如果这个进程失败，则抛出一个异常。类被装载后同时在main方法被调用之前，必须对类HelloApp与其它类型进行链接然后初始化。链接包含三个阶段：检验，准备和解析。检验检查被装载的主类的符号和语义，准备则创建类或接口的静态域以及把这些域初始化为标准的默认值，解析负责检查主类对其它类或接口的符号引用，在这一步它是可选的。类的初始化是对类中声明的静态初始化函数和静态域的初始化构造方法的执行。一个类在初始化之前它的父类必须被初始化。整个过程如下：



### 类的加载

当使用JAVA命令运行某个JAVA程序时，该命令都将会启动一条JAVA虚拟机进程，这个程序所有的线程都将位于该进程内，同一个JVM的所有线程、变量都使用该JVM进程的内存区。当系统退出时将出现以下几种情况，JVM进程将被终止：

1. 程序运行到最后正常结束。
2. 程序运行遇到Sytem.exit()或Runtime.getRuntime().exit()代码结束程序。
3. 程序执行过程中碰到了未捕获的异常或错误而结束。
4. 程序所有在平台强制结束了JVM进程。

注意：当JVM进程结束，该进程内存中状态将会丢失。

**public** **class** A{

**public** **static** **int** *a* = 6;

}

**public** **class** TestA1{

**public** **static** **void** main(String[] args) {

A a = **new** A();

a.*a*++;

System.***out***.println(a.*a*); //7

}

}

**public** **class** TestA2{

**public** **static** **void** main(String[] args) {

A b = **new** A();

System.***out***.println(b.*a*); //6

}

}

JAVA中的类初始化分为三个步骤：加载、连接、初始化。

以上三个步骤统称为类的加载或类初始化。

所谓类的加载是指将class文件读入内存，并为之创建一个java.lang.Class对象，也就是说当程序使用任何类，都将会为之建立一个java.lang.Class对象。

不同的类加载器可以加载不同的来源加载二进制数据，通常有如下几种：

1. 从本地文件系统加载class文件。
2. 从JAR包中加载class文件。
3. 通过网络加载class文件。
4. 把一个java源文件动态编译、并执行加载。

### 类的连接

当类被加载完成后将形成一个Class对象，接着进入连接阶段，在连接阶段将会把类的二进制数据合并到一个JRE中。类连接又可分为如下三个阶段：

1. 验证：验证被加载的类是否有正确的内部结构，并和其他类协调一致。
2. 准备：为类的静态属性分配内存，并设置默认初始值。
3. 解析：将类的二进制数据中的符号引用替换成直接引用。

类的初始化

类的初始化阶段，虚拟机负责对类进行初始化，主要是静态属性初始化。

JAVA对静态属性初始化的方式：

1. 申明静态属性时执行初始值。
2. 使用静态初始化块为静态属性指定初始值。

**static** **int** a = 5;

**static** **int** b;

JVM初始化一个类包含如下几个步骤：

1. 假如这个类还没被加载连接，程序先加载和连接该类。
2. 假如该类的直接父类还没被初始化，则先初始化其直接父类。
3. 假如类中有 初始化语句，则系统依次执行这些初始化的语句。

### 类的加载器

类装载器负责加载所有的类，装载成功后都会为该类生成一个java.lang.Class实例，一个类被加载到JVM中，同一个类就不会再被载入了。

当JVM启动时，会形成由三个类加载器组成的初始化类加载器层次结构：

1. 根类加载器（Bootstrap ClassLoader）（该类不是ClassLoader的子类）负责加载java核心类
2. 扩展类加载器（Extension ClassLoader）负责加载JRE的扩展目录中的JAR的类包。
3. 系统加载器（System ClassLoader）加载来自命令java中指定的类文件或者CLASSPAT环境变量所指定的JAR包和类路径。

### 反射

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

反射机制提供的功能：

在运行时判断任意一个对象所属的类。

在运行时构造任意一个类的对象。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法

1. **关于Class**
2. Class是一个类，一个描述类的类（也就是描述类本身），封装了描述方法的Method，描述字段的Filed，描述构造器的Constructor等属性
3. 对象照镜子后（反射）可以得到的信息：某个类的数据成员名、方法和构造器、某个类到底实现了哪些接口。
4. 对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。
5. Class 对象只能由系统建立对象
6. 一个类在 JVM 中只会有一个Class实例
7. **获得Class对象的5种方式**：
8. Object.getClass()：如果一个类的对象可用，则可以使用对象的.getClass()方法获取当前对象的Class对象。只对引用类型有效。
9. 类名.class：如果类型可用但没有对象，则可以在类型后面加上.class来获得该类的Class对象。这也是使用原始类型获得Class对象最简单的方式
10. Class.forName()：如果知道类的全名（包.类名），则可以使用Class类的静态方法forName()来获得Class对象，它不能用在原始类型上，但可以用在原始类型数组上。（注：此方式会抛出ClassNotFoundException异常）。该方法告知JVM开始加载给定的类，也就是说JVM会执行该类的静态代码段。
11. 包装类名.TYPE：每个基本类型和void都有包装类。利用其TYPE域就可以获得Class对象。
12. 以Class为返回值的方法：如获取内部类的getDeclaredClasses()方法，此时返回一个对象数组

**public** **static** **void** main(String[] args) {

System.***out***.println("第一种方式--用Object.getClass()方法获取到的String类的对象");

Class class1 = **new** String().getClass();

System.***out***.println(class1.getName());

System.***out***.println("第二种方式--用.class获取到的String类的对象");

class1 = String.**class**;

System.***out***.println(class1.getName());

System.***out***.println("第三种方式--用.forName()方法获取到的String类的对象");

class1 = Class.*forName*("java.lang.String");

System.***out***.println(class1.getName());

System.***out***.println("第四种方式--用包装类的TYPE域获取到的Integer类的对象");

class1 = Integer.***TYPE***;

System.***out***.println(class1.getName());

System.***out***.println("第五种方式--用getDeclaredClasses()获取String类的内部类对象");

Class<?>[] clazz = **new** String().getClass().getDeclaredClasses();

**for** (Class<?> class2 : clazz) {

System.***out***.println("\t" + class2);

}

}

1. **反射API**

**java.lang.Class类**

Class类：代表的是java中的一个类。

Class类的实例用于表示运行时的JAVA数据类型，包括类、接口、数组、枚举、注解和基本数据类型。

在类加载时，Java虚拟机会自动创建Class对象

1. 常用方法：

**public <U> Class<? extends U> asSubclass(Class<U> clazz)**

强制转换该 Class 对象，以表示指定的 class 对象所表示的类的一个子类。检查强制转换的有效性，如果无效则抛出 ClassCastException。如果此方法成功了，它将始终返回对此 class 对象的一个引用。

**public T cast(Object obj)**

将一个对象强制转换成此 Class 对象所表示的类或接口。若 obj 为 null 则返回 null

**public static Class<?> forName(String className)**

返回与带有给定字符串名的类或接口相关联的 Class 对象。其中className为指定的类的完整名称（包.类）

**public static Class<?> forName(String name, boolean initialize, ClassLoader loader)**

使用给定的类加载器，返回与带有给定字符串名的类或接口相关联的 Class 对象。

**public <A extends Annotation> A getAnnotation(Class<A> annotationClass)**

如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。

**public Annotation[] getAnnotations()**

返回此元素上存在的所有注释。（如果此元素没有注释，则返回长度为零的数组。）该方法的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。

**public ClassLoader getClassLoader()**

返回该类的类加载器。有些实现可能使用 null 来表示引导类加载器。如果该类由引导类加载器加载，则此方法在这类实现中将返回 null。（类加载器是负责加载类的对象。如使用类加载器的方法用于加载java配置文件）

**public Field getDeclaredField(String name)**

返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段。name 参数是一个 String，它指定所需字段的简称。注意，此方法不反映数组类的 length 字段。

**public Field[] getDeclaredFields()**

返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段。包括公共、保护、默认（包）访问和私有字段，但不包括继承的字段。返回数组中的元素没有排序，也没有任何特定的顺序。如果该类或接口不声明任何字段，或者此 Class 对象表示一个基本类型、一个数组类或 void，则此方法返回一个长度为 0 的数组。

**public Method getDeclaredMethod(String name, Class<?>... parameterTypes)**

返回一个 Method 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。name 参数是一个 String，它指定所需方法的简称，parameterTypes 参数是 Class 对象的一个数组，它按声明顺序标识该方法的形参类型。如果在某个类中声明了带有相同参数类型的多个方法，并且其中有一个方法的返回类型比其他方法的返回类型都特殊，则返回该方法；否则将从中任选一个方法。如果名称是 "<init>” 或 “<clinit>"，则引发一个 NoSuchMethodException。

**public Method[] getDeclaredMethods()**

返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。返回数组中的元素没有排序，也没有任何特定的顺序。如果该类或接口不声明任何方法，或者此 Class 对象表示一个基本类型、一个数组类或 void，则此方法返回一个长度为 0 的数组。类初始化方法 <clinit> 不包含在返回数组中。如果该类声明带有相同参数类型的多个公共成员方法，则它们都包含在返回的数组中。

**public Field getField(String name)**

返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段。name 参数是一个 String，用于指定所需字段的简称。

**public Field[] getFields()**

返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段。返回数组中的元素没有排序，也没有任何特定的顺序。如果类或接口没有可访问的公共字段，或者表示一个数组类、一个基本类型或 void，则此方法返回长度为 0 的数组。

特别地，如果该 Class 对象表示一个类，则此方法返回该类及其所有超类的公共字段。如果该 Class 对象表示一个接口，则此方法返回该接口及其所有超接口的公共字段。

该方法不反映数组类的隐式长度字段。用户代码应使用 Array 类的方法来操作数组。

**public Method getMethod(String name, Class<?>... parameterTypes)**

返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。name 参数是一个 String，用于指定所需方法的简称。parameterTypes 参数是按声明顺序标识该方法形参类型的 Class 对象的一个数组。如果 parameterTypes 为 null，则按空数组处理。

**public Method[] getMethods()**

返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。数组类返回从 Object 类继承的所有（公共）member 方法。返回数组中的元素没有排序，也没有任何特定的顺序。如果此 Class 对象表示没有公共成员方法的类或接口，或者表示一个基本类型或 void，则此方法返回长度为 0 的数组。

**public String getName()**

以 String 的形式返回此 Class 对象所表示的实体（类、接口、数组类、基本类型或 void）名称。

**public Package getPackage()**

获取此类的包。此类的类加载器用于查找该包。如果该类是通过引导类加载器加载的，则搜索从 CLASSPATH 加载的包的集合，以查找该类的包。如果所有包对象都不是用该类的类加载器加载的，则返回 null。

**public T newInstance()**

创建此 Class 对象所表示的类的一个新实例。如同用一个带有一个空参数列表的 new 表达式实例化该类。如果该类尚未初始化，则初始化这个类。

1. **java.lang.reflect包**

该包中提供的类和接口，以获得关于类和对象的反射信息。

1. java.lang.reflect.Field类

Field 类：代表类的成员变量（成员变量也称为类的属性）。

Field 提供有关类或接口的单个属性的信息，以及对它的动态访问权限。反射的属性可能是一个类（静态）属性或实例属性。

获取Field类对象的方法：（使用Class类的如下方法）

getFields(): 获取类中public类型的属性。

getField(String name)： 获取类特定的方法，name参数指定了属性的名称

getDeclaredFields(): 获取类中所有的属性(public、protected、default、private),但不包括继承的属性。

getDeclaredField(String name): 获取类特定的方法，name参数指定了属性的名称。

Field类中常用的方法：

**public boolean equals(Object obj)**

将此 Field 与指定对象比较。如果对象是相同的，则返回 true。如果由相同的类声明并且具有相同的名称和类型，那么这两个 Field 对象是相同的。

**public Object get(Object obj)**

返回指定对象上此 Field 表示的字段的值。如果该值是一个基本类型值，则自动将其包装在一个对象中。

**public Type getGenericType()**

返回一个 Type 对象，它表示此 Field 对象所表示字段的声明类型。如果 Type 是一个参数化类型，则返回的 Type 对象必须准确地反映源代码中使用的实际类型参数。如果底层字段的类型是一个类型变量或者是一个参数化类型，则创建它。否则将解析它。

**public String getName()**

返回此 Field 对象表示的字段的名称。

**public Class<?> getType()**

返回一个 Class 对象，它标识了此 Field 对象所表示字段的声明类型。

1. java.lang.reflect.Method类

Method类：代表类的方法。

Method 提供关于类或接口上单独某个方法（以及如何访问该方法）的信息。所反映的方法可能是类方法或实例方法（包括抽象方法）。

获取Method类对象的方法：（使用Class类的如下方法）

getDeclaredMethod(String name, Class<?>... parameterTypes)：获取给定名称和参数列表对应的方法对象

getDeclaredMethods()：返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。

getMethod(String name, Class<?>... parameterTypes)：返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。

getMethods()：返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。

Method类中常用方法：

**public Object invoke(Object obj, Object... args)**

对带有指定参数的指定对象调用由此Method 对象表示的底层方法。个别参数被自动解包，以便与基本形参相匹配，基本参数和引用参数都随需服从方法调用转换。

**public String getName()**

以 String 形式返回此 Method 对象表示的方法名称。

**public Type getGenericReturnType()**

返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象。如果返回类型是参数化类型，则返回的 Type 对象必须实际反映源代码中所用参数的实际类型。如果返回类型是类型变量或参数化类型，则创建它。否则将解析它。

1. java.lang.reflect.Constructor类

Constructor 类：代表类的构造方法。

Constructor 提供关于类的单个构造方法的信息以及对它的访问权限。Constructor 允许在将实参与带有底层构造方法的形参的 newInstance() 匹配时进行扩展转换，但是如果发生收缩转换，则抛出 IllegalArgumentException。

获取Constructor类对象的方法：（使用Class类的如下方法）

getConstructor(Class<?>... parameterTypes)：返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共构造方法。parameterTypes - 参数数组。

getConstructors()：返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法。

getDeclaredConstructor(Class<?>... parameterTypes)：返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法。

getDeclaredConstructors()：返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。它们是公共、保护、默认（包）访问和私有构造方法。

Constructor类的常用方法：

**public Type[] getGenericParameterTypes()**

按照声明顺序返回一组 Type 对象，这些对象表示此 Constructor 对象所表示的方法的形参类型。如果底层方法不带任何参数，则返回一个长度为 0 的数组。

**public String getName()**

以字符串形式返回此构造方法的名称。它总是与构造方法的声明类的简单名称相同。

**public Class<?>[] getParameterTypes()**

按照声明顺序返回一组 Class 对象，这些对象表示此 Constructor 对象所表示构造方法的形参类型。如果底层构造方法不带任何参数，则返回一个长度为 0 的数组。

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

使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，并用指定的初始化参数初始化该实例。个别参数会自动解包，以匹配基本形参，必要时，基本参数和引用参数都要进行方法调用转换。initargs - 将作为变量传递给构造方法调用的对象数组；基本类型的值被包装在适当类型的包装器对象（如 Float 中的 float）中。

**public String toGenericString()**

返回描述此 Constructor 的字符串，其中包括类型参数。该字符串是作为构造方法访问修饰符（如果有）格式化的，其后面是构造方法的类型参数（如果有）的加尖括号的、逗号分隔的列表，再往后是声明类的完全限定名，再往后是该构造方法的一般形参类型的加括号的、逗号分隔的列表。空格用来分隔访问修饰符与访问修饰符，以及访问修饰符与类型参数或返回值。如果没有类型参数，则省略类型参数列表；如果存在类型参数列表，则用空格分隔列表与类名。如果声明构造方法抛出异常，则参数列表后跟着一个空格，再往后是单词 "throws" ，再往后是用逗号分隔的所抛出异常类型的列表。

1. java.lang.reflect.Array类

Array类：提供了动态创建数组，以及访问数组的元素的静态方法

Array 类提供了动态创建和访问 Java 数组的方法。Array 允许在执行 get 或 set 操作期间进行扩展转换，但如果发生收缩转换，则抛出 IllegalArgumentException。

获取Array类对象的方法：（使用Class类的如下方法）

getComponentType()：返回表示数组组件类型的 Class。如果此类不表示数组类，则此方法返回 null。如果此类是数组，则返回表示此类组件类型的 Class

Array类常用的方法：

**public static Object get(Object array, int index)**

返回指定数组对象中索引组件的值。如果该值是一个基本类型值，则自动将其包装在一个对象中。array - 数组、index - 索引。

**public static Object newInstance(Class<?> componentType, int... dimensions)**

创建一个具有指定的组件类型和维度的新数组。如果 componentType 表示一个非数组类或接口，则新数组具有 dimensions.length 维度，并且将 componentType 作为其组件类型。如果 componentType 表示一个数组类，则新数组的维数等于 dimensions.length 和 componentType 的维数的总和。在这种情况下，新数组的组件类型为 componentType 的组件类型。新数组的维数不能超过该实现所支持的数组维数（通常为 255）。componentType - 表示新数组的组件类型的 Class 对象、dimensions - 表示新数组维度的 int 数组。

**public static Object newInstance(Class<?> componentType, int length)**

创建一个具有指定的组件类型和长度的新数组。调用此方法等效于创建如下数组：int[] x = {length}; Array.newInstance(componentType, x); componentType - 表示新数组的组件类型的 Class 对象、length - 新数组的长度。

**public static void set(Object array, int index, Object value)**

将指定数组对象中索引组件的值设置为指定的新值。如果数组的类型为基本组件类型，则新值第一个被自动解包。array - 数组、index - 数组内部的索引、value - 索引组件的新值。

## 代理模式

代理(Proxy)是一种设计模式,提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能。

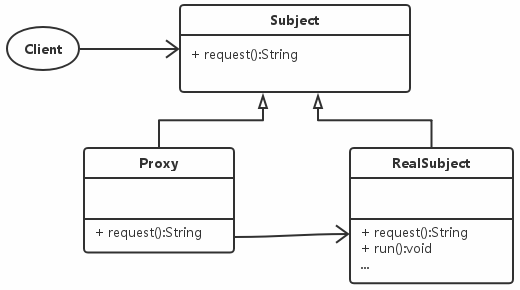
这里使用到编程中的一个思想:不要随意去修改别人已经写好的代码或者方法,如果需改修改,可以通过代理的方式来扩展该方法

举个例子来说明代理的作用:假设我们想邀请一位明星,那么并不是直接连接明星,而是联系明星的经纪人,来达到同样的目的.明星就是一个目标对象,他只要负责活动中的节目,而其他琐碎的事情就交给他的代理人(经纪人)来解决.这就是代理思想在现实中的一个例子。

代理模式的关键点是:代理对象与目标对象.代理对象是对目标对象的扩展,并会调用目标对象。

### 静态代理

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类。



RealSubject 是委托类，Proxy 是代理类； Subject 是委托类和代理类的接口； request() 是委托类和代理类的共同方法；

父类接口：

**public** **interface** Subject {

**void** request();

}

委托类：

**public** **class** RealSubject **implements** Subject {

@Override

**public** **void** request() {

System.***out***.println("RealSubject");

}

}

代理类：

**public** **class** Proxy **implements** Subject {

**private** Subject subject;

**public** Proxy(RealSubject realSubject) {

**this**.subject = realSubject;

}

@Override

**public** **void** request() {

System.***out***.println("begin");

subject.request();

System.***out***.println("end");

}

}

测试类：

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Subject subject = **new** Proxy(**new** RealSubject());

subject.request();

}

}

需要注意的是,代理对象与目标对象要实现相同的接口,然后通过调用相同的方法来调用目标对象的方法。

静态代理总结:

1. 可以做到在不修改目标对象的功能前提下,对目标功能扩展.
2. 缺点:

因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.

如何解决静态代理中的缺点呢?答案是可以使用动态代理方式

### 动态代理

动态代理中，代理类并不是在Java代码中实现，而是在运行时期生成，相比静态代理，动态代理可以很方便的对委托类的方法进行统一处理，如添加方法调用次数、添加日志功能等等，动态代理分为jdk动态代理和cglib动态代理，下面通过一个例子看看如何实现jdk动态代理。

动态代理有以下特点:

1. 代理对象,不需要实现接口
2. 代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)
3. 动态代理也叫做:JDK代理,接口代理

**JDK中生成代理对象的API**

代理类所在包:java.lang.reflect.Proxy

现在要生成某一个对象的代理对象，这个代理对象通常也要编写一个类来生成，所以首先要编写用于生成代理对象的类。在java中如何用程序去生成一个对象的代理对象呢，java在JDK1.5之后提供了一个"java.lang.reflect.Proxy"类，通过"Proxy"类提供的一个newProxyInstance方法用来创建一个对象的代理对象，如下所示：



newProxyInstance方法用来返回一个代理对象，这个方法总共有3个参数，ClassLoader loader用来指明生成代理对象使用哪个类装载器，Class<?>[] interfaces用来指明生成哪个对象的代理对象，通过接口指定，InvocationHandler h用来指明产生的这个代理对象要做什么事情。所以我们只需要调用newProxyInstance方法就可以得到某一个对象的代理对象了。

在java中规定，要想产生一个对象的代理对象，那么这个对象必须要有一个接口，所以我们第一步就是设计这个对象的接口，在接口中定义这个对象所具有的行为(方法)

定义对象的行为接口：

**public** **interface** Person {

String sing(String name);

String dance(String name);

}

定义目标业务对象类：

**public** **class** LiuDeHua **implements** Person {

@Override

**public** String sing(String name) {

System.***out***.println("刘德华唱"+name+"歌！！");

**return** "歌唱完了，谢谢大家！";

}

@Override

**public** String dance(String name) {

System.***out***.println("刘德华跳"+name+"舞！！");

**return** "舞跳完了，多谢各位观众！";

}

}

创建生成代理对象的代理类：

**public** **class** ProxyFactory {

**private** Object target;

**public** ProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyInstance() {

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces()

, **new** InvocationHandler() {

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

**if**(method.getName().equals("sing")) {

System.***out***.println("我是他的经纪人，要找他唱歌得先给十万块钱！！");

**return** method.invoke(target, args);

}

**if**(method.getName().equals("dance")) {

System.***out***.println("我是他的经纪人，要找他跳舞得先给二十万块钱！！");

**return** method.invoke(target, args);

}

**return** **null**;

}

});

}

}

测试类：

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ProxyFactory proxyFactory = **new** ProxyFactory(**new** LiuDeHua());

Person person = (Person)proxyFactory.getProxyInstance();

System.***out***.println(person.sing("冰雨"));

System.***out***.println(person.dance("江南style"));

}

}

Proxy类负责创建代理对象时，如果指定了handler（处理器），那么不管用户调用代理对象的什么方法，该方法都是调用处理器的invoke方法。

由于invoke方法被调用需要三个参数：代理对象、方法、方法的参数，因此不管代理对象哪个方法调用处理器的invoke方法，都必须把自己所在的对象、自己（调用invoke方法的方法）、方法的参数传递进来。

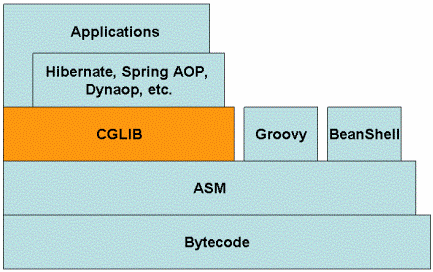
**jdk动态代理使用的局限性**

通过反射类Proxy和InvocationHandler回调接口实现的jdk动态代理，要求委托类必须实现一个接口，但事实上并不是所有类都有接口，对于没有实现接口的类，便无法使用该方式实现动态代理。

### CGLib代理

CGLib（Code Generator Library）是一个强大的、高性能的代码生成库。其被广泛应用于AOP框架（Spring、dynaop）中，用以提供方法拦截操作。Hibernate作为一个比较受欢迎的ORM框架，同样使用CGLib来代理单端（多对一和一对一）关联（延迟提取集合使用的另一种机制）。

CGLib组成结构：



上面的静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式类实现代理,这种方法就叫做: CGLib代理。

使用CGLib实现动态代理，完全不受代理类必须实现接口的限制，而且CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，因为CGLib原理是动态生成被代理类的子类。

CGLib子类代理实现方法:

使用CGLib动态代理之前需要在项目中引入CGLib的jar包。

创建目标对象：

**public** **class** UserDao {

**public** **void** save() {

System.***out***.println("----已经保存数据!----");

}

}

创建代理工厂类：

**public** **class** ProxyFactorys **implements** MethodInterceptor {

**private** Object target;

**public** ProxyFactorys(Object target) {

**this**.target = target;

}

//给目标对象创建一个代理对象

**public** Object getProxyInstance() {

//1.工具类

Enhancer en = **new** Enhancer();

//2.设置父类

en.setSuperclass(target.getClass());

//3.设置回调函数

en.setCallback(**this**);

//4.创建子类(代理对象)

**return** en.create();

}

@Override

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

System.***out***.println("开始事务...");

//执行目标对象的方法

Object returnValue = method.invoke(target, args);

System.***out***.println("提交事务...");

**return** returnValue;

}

}

测试类：

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ProxyFactorys proxyFactorys = **new** ProxyFactorys(**new** UserDao());

UserDao userDao = (UserDao)proxyFactorys.getProxyInstance();

userDao.save();

}

}

由于CGLIB的大部分类是直接对Java字节码进行操作，这样生成的类会在Java的永久堆中。如果动态代理操作过多，容易造成永久堆满，触发OutOfMemory异常。

## Java注解

### 概述

注解（Annotation），也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。

注解（Annotation）提供了一种安全的类似注释的机制，为我们在代码中添加信息提供了一种形式化得方法，使我们可以在稍后某个时刻方便的使用这些数据（通过解析注解来使用这些数据），用来将任何的信息或者元数据与程序元素（类、方法、成员变量等）进行关联。其实就是更加直观更加明了的说明，这些说明信息与程序业务逻辑没有关系，并且是供指定的工具或框架使用的。Annotation像一种修饰符一样，应用于包、类型、构造方法、方法、成员变量、参数及本地变量的申明语句中。

Annotation其实是一种接口。通过Java的反射机制相关的API来访问Annotation信息。相关类（框架或工具中的类）根据这些信息来决定如何使用该程序元素或改变它们的行为。Java语言解释器在工作时会忽略这些Annotation，因此在JVM中这些Annotation是“不起作用”的，只能通过配套的工具才能对这些Annotation类型的信息进行访问和处理。

### 元注解

元注解是指JDK提供的用于注释自定义注解的注解，定义自定义注解的一些行为。元注解有 @Retention、@Documented、@Target、@Inherited、@Repeatable 5 种。

@Retention

描述该注解的生命周期，表示在什么编译级别上保存该注解的信息。Annotation被保留的时间有长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取（请注意并不影响class的执行，因为Annotation与class在使用上是被分离的）：

取值范围：

1. RetentionPolicy.SOURCE：注解将被编译器丢弃（该类型的注解信息只会保留在源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的class文件里）。
2. RetentionPolicy.CLASS：注解在class文件中可用，但会被JVM丢弃（该类型的注解信息会保留在源码里和class文件里，在执行的时候，不会加载到虚拟机中），请注意，当注解未定义Retention值时，默认值是CLASS，如Java内置注解，@Override、@Deprecated、@SuppressWarnning等
3. RetentionPolicy.RUNTIME：注解信息将在运行期(JVM)也保留，因此可以通过反射机制读取注解的信息（源码、class文件和执行的时候都有注解的信息），如SpringMvc中的@Controller、@Autowired、@RequestMapping等。

@Documented

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***ANNOTATION\_TYPE***)

**public** **@interface** Documented {

}

@Documented

@Documented注解的作用是在生成javadoc文档的时候将该Annotation也写入到文档中。

@Target

描述该注解修饰的范围，可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。

@Target(ElementType.***TYPE***)

@Retention(RetentionPolicy.***RUNTIME***)

@Documented

**public** **@interface** Indexed {

}

@Target注解有如下的取值范围：

1. ElementType.TYPE：类、接口(包括注释类型)或枚举声明
2. ElementType.FIELD：字段声明(包括枚举常量)
3. ElementType.METHOD：方法声明
4. ElementType.PARAMETER：形式参数声明
5. ElementType.CONSTRUCTOR：构造函数声明
6. ElementType.LOCAL\_VARIABLE：局部变量声明
7. ElementType.ANNOTATION\_TYPE：注释类型声明
8. ElementType.PACKAGE：包声明
9. ElementType.TYPE\_PARAMETER：类型参数声明,表示该注解能写在类型变量的声明语句中。
10. ElementType.TYPE\_USE：使用类型，表示该注解能写在使用类型的任何语句中（例如声明语句、泛型和强制转换语句中的类型）。

类型注解:

类型注解扩展了注解使用的范围，可以使用在创建类实例、类型映射、implements语句、throw exception声明中的类型前面。例如：

创建类实例： new @Interned MyObject();

类型映射：myString = (@NonNull String) str;

implements 语句中 ：class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }

throw exception声明:void monitorTemperature() throws @Critical TemperatureException { ... }

@Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})

**public** **@interface** Encrypted {

}

**public** **class** MyTypeAnnotation {

@Encrypted String data;

List<@Encrypted String> strings;

}

类型注解的作用：

首先，局域变量声明中的类型注解也可以保留在类文件中，完整泛型被保留，并且在运行期可以访问，从而有助于我们获取更多的代码信息；其次，类型注解可以支持在的程序中做强类型检查。配合第三方工具check framework，可以在编译的时候检测出runtime error，以提高代码质量；最后，代码中包含的注解清楚表明了编写者的意图，使代码更具有表达意义，有助于阅读者理解程序，毕竟代码才是“最根本”的文档、“最基本”的注释。

@Inherited

@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

@Inherited

@Retention(RetentionPolicy.RUNTIME)

**@interface** Test {}

@Test

**public** **class** A {}

**public** **class** B **extends** A {}

@Repeatable

@Repeatable 是 Java 1.8加入的，该注解表明标记的注解可以多次应用于相同的声明或类型使用。

没有使用@Repeatable注解的方式：

**public** **@interface** Authority {

String role();

}

**public** **@interface** Authorities {

Authority[] value();

}

**public** **class** RepeatAnnotationUseOldVersion {

@Authorities({@Authority(role="Admin"),@Authority(role="Manager")})

**public** **void** doSomeThing(){

}

}

使用@Repeatable注解的方式：

@Repeatable(Authorities.**class**)

**public** **@interface** Authority {

String role();

}

**public** **@interface** Authorities {

Authority[] value();

}

**public** **class** RepeatAnnotationUseNewVersion {

@Authority(role="Admin")

@Authority(role="Manager")

**public** **void** doSomeThing(){ }

}

### JDK自带的注解

@Override：该注解作用于方法，表示覆盖或重写父类的方法。

**public** **interface** Person {

**public** String name();

**public** **int** age();

/\*@Deprecated 表示该方法已经过时了\*/

@Deprecated

**public** **void** ~~sing~~();

/\*注：

当方法或是类上面有@Deprecated注解时，

说明该方法或是类都已经过期不能再用，

但不影响以前项目使用，提醒你新替代待的方法或是类

\*/

}

@Deprecated：该方法也作用于方法，表式该方法已经过时。（当方法或是类上面有@Deprecated注解时，说明该方法或是类都已经过期不能再用，但不影响以前项目使用，提醒你新替代待的方法或是类。如果程序员不小心使用了它的元素,那么编译器会发出警告信息。）

**public** **class** Child **implements** Person {

/\*@Override 表示覆盖或重写父类的方法\*/

@Override

**public** String name() {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

@SuppressWarnings：表示忽略指定警告，比如@Suppvisewarnings("Deprecation")

**public** **class** Test {

/\*@SuppressWarnings 表示忽略指定警告，比如@Suppvisewarnings("Deprecation")\*/

@SuppressWarnings("deprecation")

**public** **void** sing(){

Person p=**new** Child();

p.sing();

}

}

### 注解的分类

1. 按照运行机制分

源码注解：注解只在源码中存在，编译成.class文件就不存在了

编译时注解：注解在源码和.class文件中都存在（如：JDK内置系统注解）

运行时注解：在运行阶段还起作用，甚至会影响运行逻辑的注解（如：Spring中@Autowried）

1. 按照来源分

JDK内置系统注解、元注解、自定义注解、第三方注解

### 自定义注解

1. 语法格式

注解是java中一种数据类型，使用@interface关键字进行修饰。

**public** **@interface** Ety {

}

我们可以给注解定义成员方法。但是对成员方法还是有一些限制。

1. 定义的成员方法不能有参数。
2. 成员方法的返回类型是受限的，合法的返回类型包括原始类型及String、Class、Annotation、Enumeration。
3. 如果注解只有一个成员方法，则方法名必须取名为value()，在使用时可以忽略方法名和赋值号（=）。
4. 注解类可以没有成员方法，没有成员方法的注解成为标识注解。
5. 成员方法可以包含默认值，使用default关键字定义。

**public** **@interface** Description{//使用@interface关键字注解

String name();//成员以无参无异常方式声明

String author();

**int** age() **default** 19;//可以用default为成员方法指定一个默认值

}

自定义一个完整的注解还需要给该注解注释元注解来规定该注解的行为。

@Target(ElementType.***TYPE***)

@Retention(RetentionPolicy.***RUNTIME***)

@Documented

@Indexed

**public** **@interface** Component {

String value() **default** "";

}

1. 使用自定义注解



1. 解析注解

Java使用java.lang.annotation.Annotation接口来代表程序元素的注解，该接口是所有Annotation类型的父接口。相应地Java在java.lang.reflect包下新增了AnnotatedElement接口，该接口代表程序中可以接受注解的程序元素。

实际上，java.lang.reflect 包所有提供的反射API扩充了读取运行时Annotation信息的能力。当一个Annotation类型被定义为运行时的Annotation后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。

AnnotatedElement接口是所有程序元素（Field、Method、Package、Class和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的如下七个方法来访问Annotation信息：

1. <T extends Annotation> T getAnnotation(Class<T> annotationClass)：返回程序元素上存在的、制定类型的注解，如果该类型注解不存在，则返回null。
2. Annotation[] getDeclaredAnnotation(Class<T>)：返回该程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null；与此接口中的其他方法不同，该方法将忽略继承的注解。
3. Annotation[] getAnnotations()：返回该程序元素上存在的所有注解。
4. Annotation[] getDeclaredAnnotations()：返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注解。
5. Annotation[] getAnnotationsByType(Class<T>)：返回直接存在于此元素上指定注解类型的所有注解。
6. Annotation[] getDeclaredAnnotationsByType(Class<T>)：返回直接存在于此元素上指定注解类型的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注解。
7. boolean isAnnotationPresent(Class<?extends Annotation> annotationClass)：判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false。

/\*\*\*\*\*\*\*\*\*\*\*注解声明\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 水果名称注解

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitName {

String value() **default** " ";

}

/\*\*

\* 水果颜色注解

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitColor {

/\*\*

\* 颜色枚举

\*/

**public** **enum** Color{***BLUE***, ***RED***, ***GREEN***};

/\*\*

\* 颜色属性

\* **@return**

\*/

Color fruitColor() **default** Color.***GREEN***;

}

/\*\*

\* 水果供应商注解

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitProvider {

/\*\*

\* 供应商编号

\* **@return**

\*/

**public** **int** id() **default** -1;

/\*\*

\* 供应商名称

\* **@return**

\*/

**public** String name() **default** " ";

/\*\*

\* 供应商地址

\* **@return**

\*/

**public** String address() **default** " ";

}

/\*\*\*\*\*\*\*\*\*\*\*注解使用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** Apple {

@FruitName("Apple")

**private** String appleName;

@FruitColor(fruitColor = FruitColor.Color.***RED***)

**private** String appleColor;

@FruitProvider(id = 1, name = "陕西红富士集团", address = "陕西红富士大厦")

**private** String appleProvider;

//省略setter、getter方法

}

/\*\*\*\*\*\*\*\*\*\*\*注解信息获取\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** AnnotationParser {

**public** **static** **void** main(String[] args) {

Field[] fields = Apple.**class**.getDeclaredFields();

**for** (Field field : fields) {

//System.out.println(field.getName().toString());

**if** (field.isAnnotationPresent(FruitName.**class**)){

FruitName fruitName = field.getAnnotation(FruitName.**class**);

System.out.println("水果的名称：" + fruitName.value());

}**else** **if** (field.isAnnotationPresent(FruitColor.**class**)){

FruitColor fruitColor = field.getAnnotation(FruitColor.**class**);

System.out.println("水果的颜色："+fruitColor.fruitColor());

}**else** **if** (field.isAnnotationPresent(FruitProvider.**class**)){

FruitProvider fruitProvider = field.getAnnotation(FruitProvider.**class**);

System.out.println("水果供应商编号:" + fruitProvider.id() + " 名称:" + fruitProvider.name() + " 地址:" +

fruitProvider.address());

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*输出结果\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

水果的名称：Apple

水果的颜色：RED

水果供应商编号:1 名称:陕西红富士集团 地址:陕西红富士大厦

## Java配置文件

**Properties类**

java.util.Properties类主要用于读取Java的配置文件，它表示了一个持久的属性集，它是在一个文件中存储键值对的，其中键值对以等号分隔。Properties可保存在IO流中或从IO流中加载。属性列表中的每个键及其所对应的值都是字符串。Properties类是线程安全的：多个线程可以共享单个Properties对象而无需进行外部同步。

java的配置文件常为.properties文件，格式为文本文件，文件的内容的格式是“键=值”的格式，文本注释信息可以用"#"来注释。

#数据库URL

url=jdbc:mysql://localhost:3306/javase?useUnicode=true&characterEncoding=utf8

#JDBC驱动类

driver=com.mysql.jdbc.Driver

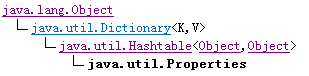
#数据库登录用户名

username=root

#数据库登录密码

password=

Properties类继承自Hashtable类，拥有Hashtable类的所有方法



**构造方法：**

public Properties()：创建一个无默认值的空属性列表。

public Properties(Properties defaults)：创建一个带有指定默认值的空属性列表。

**常用方法：**

|  |  |
| --- | --- |
| 方法 | 说明 |
| public String getProperty(String key) | 用指定的键在此属性列表中搜索属性。如果在此属性列表中未找到该键，则接着递归检查默认属性列表及其默认值。如果未找到属性，则此方法返回 null。 |
| public void load(InputStream inStream) | 从输入流中读取属性列表（键和元素对）。 |
| public void load(Reader reader) | 按简单的面向行的格式从输入字符流中读取属性列表（键和元素对）。 |
| public void loadFromXML(InputStream in) | 将指定输入流中由 XML 文档所表示的所有属性加载到此属性表中。 |
| public Enumeration<?> propertyNames() | 返回属性列表中所有键的枚举，如果在主属性列表中未找到同名的键，则包括默认属性列表中不同的键。 |
| public Set<String> stringPropertyNames() | 返回此属性列表中的键集，其中该键及其对应值是字符串，其键或值不是 String 类型的属性被忽略。 |
| public Object setProperty(String key, String value) | 调用 Hashtable 的方法 put 。他通过调用基类的put方法来设置 键 - 值对。 |
| public void list(PrintStream out) | 将属性列表输出到指定的输出流。此方法对调试很有用 |
| public void list(PrintWriter out) | 将属性列表输出到指定的输出流。此方法对调试很有用。 |
| public void clear() | 将此哈希表清空，使其不包含任何键。 |

**Java读取Properties文件**

1. 使用java.util.Properties类的load()方法

该方法需传入InputStream类型的输入流，配置文件可以放在项目中的任何地方。

**public** **static** **void** main(String[] args) {

Properties ps = **new** Properties();

FileInputStream fs = **null**;

InputStream is = **null**;

**try** {

//使用文件IO流类加载外部文件

fs = **new** FileInputStream(System.*getProperty*("user.dir") + "\\src\\jdbc.properties");

//使用缓冲流实例化一个输入流InputStream对象

is = **new** BufferedInputStream(fs);

//使用Properties对象的load()方法获取输入流中的属性列表

ps.load(is);

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

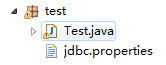
e.printStackTrace();

}

}

1. 使用java.lang.Class对象的getResourceAsStream()方法加载配置文件

该方法需要将配置文件放与类所在的包中。因为Class对象的getResourceAsStream()方法会以加载配置文件类所在的包名开始查找相应的资源。如果这个文件名是以 '/' 开头的，那么就会从classpath的根路径下开始查找。



**public** **static** **void** main(String[] args) {

Properties ps = **new** Properties();

//利用反射加载配置文件

InputStream is = Test.**class**.getResourceAsStream("jdbc.properties");

**try** {

ps.load(is);

} **catch** (IOException e) {

e.printStackTrace();

}

}

1. 使用java.lang.Class对象的public ClassLoader getClassLoader()方法返回ClassLoader对象，再调用ClassLoader对象的getResourceAsStream(String name)方法加载配置文件。

该方法需将配置文件放与项目的src文件夹下：

**public** **static** **void** main(String[] args) {

Properties ps = **new** Properties();

//利用反射加载配置文件

InputStream is = Test.**class**.getClassLoader().getResourceAsStream("jdbc.properties");

**try** {

ps.load(is);

} **catch** (IOException e) {

e.printStackTrace();

}

}

1. 使用java.lang.ClassLoader类的public static InputStream getSystemResourceAsStream(String name)静态方法加载配置文件。该方法的配置文件需放在项目src文件夹下

**public** **static** **void** main(String[] args) {

Properties ps = **new** Properties();

//利用ClassLoader的静态方法加载配置文件

InputStream is = ClassLoader.*getSystemResourceAsStream*("jdbc.properties");

**try** {

ps.load(is);

} **catch** (IOException e) {

e.printStackTrace();

}

}

**Properties获取数据乱码解决**

1. 原因

Properties调用load(InputStream)时，读取文件时使用的默认编码为ISO-8859-1；当我们将中文放入到properties文件中，通过getProperty(key)获取值时，取到得数据是ISO-8859-1格式的，但是ISO-8859-1是不能识别中文的。

1. 解决方法

通过getProperty()获取的数据data既然是ISO-8859-1编码的，就通过data.getByte(“iso-8859-1”)获取获取，使用new String(data.getByte(“iso-8859-1”),”UTF-8”)进行转换。当然properties文件的编码类型需要和new String(Byte[],charset)中的第二个参数的编码类型相同。

# Java语言高级

## 多线程

在Java中，多个任务同时进行的机制被称为并发，而将并发完成的每一件事情称为线程。每一个线程完成一个功能，并与其他线程并发执行，这种机制被成为多线程。

每个线程内部都有自己的程序执行流，线程与线程之间属于并发执行，且互不影响。

### 线程和进程

**进程**：每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销，一个进程包含1--n个线程。（进程是资源分配的最小单位）。

**线程**：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。（线程是cpu调度的最小单位）。

线程是共享地址空间的，也就是说多线程可以同时读取相同的地址空间，并且利用这个空间进行交换数据

### 线程的生命周期

线程具有生命周期，其中包含新建状态、就绪状态、运行状态、等待状态、计时等待状态、阻塞状态和结束状态7中状态。

**新建状态：**当用户在创建线程后与使用线程实例调用start()方法之前的状态属于新建状态。

**就绪状态：**当用户调用start()方法后，线程处于就绪状态（可执行状态）

**运行状态：**当线程得到系统资源后就进入运行状态

**等待状态：**当线程在运行状态下调用Thread类中的wait()方法是，则线程进入等待状态。可以调用Thread类中的notify()方法

唤醒线程。notifyAll()方法则是唤醒所有处于等待状态的线程。线程唤醒后会重写进入就绪状态。

**计时等待状态**：当线程在运行状态下调用Thread类的sleep()方法，则线程进入计时等待状态。当设定的等待时间结束后，线程进

入就绪状态。

**结束状态：**线程执行完了或者因异常退出了run()方法，该线程结束生命周期

### 线程的特点

1. 多个线程在运行时，系统自动在线程之间进行切换；
2. 由于多个线程共存于同一块内存，线程之间的通信非常容易；
3. Java将线程视为一个对象。线程要么是Thread类的对象，要么是接口Runnable的对象。
4. 当多个线程并行执行时，具有较高优先级的线程将获得较多的CPU时间片；
5. 优先级是从0到10的整数，并且它仅表示线程之间的相对关系；
6. 多个线程共享一组资源，有可能在运行时产生冲突。必须采用synchronized关键字协调资源，实现线程同步。

### 实现线程的两种方法

在Java中主要提供了两种方式实现线程，分别为继承java.lang.Thread类与实现java.lang.Runnable接口。

#### 继承Thread类

Thread类是java.lang包中的一个类，该类实现了java.lang.Runnable接口。从这个类中实例化的对象代表线程，程序员启动一个新线程需要建立Thread实例。

Thread类常用的两个构造方法：

**public** Thread(String threadName)

**public** Thread()

其中第一个构造方法是创建一个名称为threadName的线程对象。

继承Thread类创建一个新的线程的语法格式：

**public** **class** ThreadTest **extends** Thread {

//.....

}

继承了Thread类后需要重写其run()方法。完成线程真正功能的代码放在重写的run()方法中，，然后同时调用Thread类中的start()方法执行线程，也就是调用run()方法。

Thread对象需要一个任务来执行，任务是指线程在启动时执行的工作，该工作的功能代码被写在run()方法中。

run()方法的语法格式：

**public** **void** run(){

//.....

}

如果start()方法调用一个已经启动的线程，系统会抛出异常

当执行一个线程程序时，就自动产生一个线程，主方法正式在这个线程上运行的。当不再启动其他线程时，该程序就为单线程程序。主方法线程启动由Java虚拟机负责，程序员负责启动自己的线程。

语法格式：

**public** **static** **void** main(String[] args) {

**new** ThreadTest().start();

}

实例：

**public** **class** ThreadTest **extends** Thread {

**private** **int** count = 10;

**public** **void** run(){ //重写Thread类的run()方法

**while**(**true**){

System.***out***.println(count + " "); //打印count变量

**if**(--count == 0){

**return**;

}

}

}

**public** **static** **void** main(String[] args) {

**new** ThreadTest().start(); //启动线程

}

}

#### 实现Runnable接口

到目前为止，线程都是通过扩展Thread类来创建的，如果程序员需要继承其他类（非Thread类），并且该程序可以使用线程，则需要使用Runnable接口。（java中类的继承属于单继承）

实现Runnable接口的语法格式：

**public** **class** Thread **extends** Object **implements** Runnable{

}

实现Runnable接口的程序会创建一个Thread对象，并将Runnable对象与Thread对象相关联。

Thread类中有两个构造方法：

**public** Thread(Runnable r)

**public** Thread(Runnable r, String name)

这两个构造方法的参数都存在Runnable实例，使用以上构造方法就可以将Runnable实例与Thread实例相关联

使用Runnable接口启动新的线程的步骤如下：

1. 建立Runnable对象
2. 使用参数为Runnable对象的构造方法将建Thread实例
3. 调用start()方法启动线程

说明：启动一个新的线程，不是直接调用Thread子类对象的run()方法，而是调用Thread子类的start()方法，Thread类的start()方法产生一个新的线程，该线程运行Thread子类的run()方法。

实现Runnable接口：

**public** **class** ThreadTest **implements** Runnable {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println(Thread.*currentThread*().getName());

}

}

}

启动线程：

**public** **static** **void** main(String[] args) {

//使用多态

Runnable threadTest = **new** ThreadTest();

Thread thread = **new** Thread(threadTest);

thread.start(); //启动线程

}

#### Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

实现Runnable接口比继承Thread类所具有的优势：

1. 适合多个相同的程序代码的线程去处理同一个资源
2. 可以避免java中的单继承的限制
3. 增加程序的健壮性，代码可以被多个线程共享，代码和数据独立
4. 线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

提醒一下大家：main方法其实也是一个线程。在java中所以的线程都是同时启动的，至于什么时候，哪个先执行，完全看谁先得到CPU的资源。

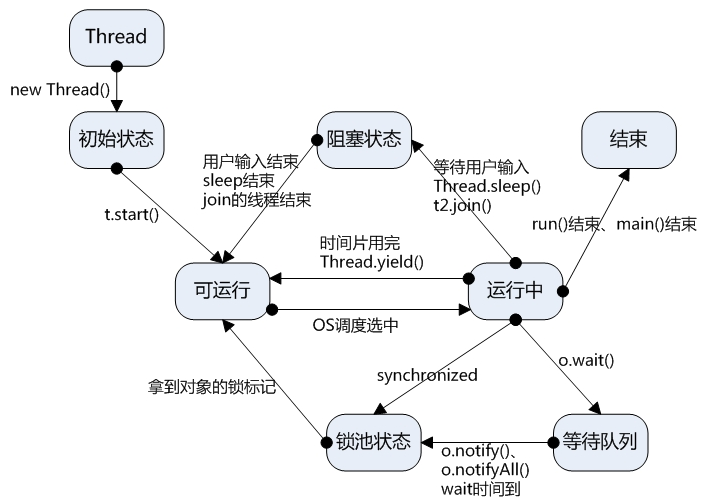
在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。因为每当使用java命令执行一个类的时候，实际上都会启动一个JVM，每一个JVM就是在操作系统中启动了一个进程。

#### start()方法和run()方法的区别

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

### 操作线程的方法

操作线程有很多方法，这些方法可以使线程从某一种状态过渡到另一种状态



#### 线程的休眠

Thread.sleep()方法用于将线程过渡到休眠状态。sleep()方法属于Thread类的静态方法，该方法需要一个参数用于指定该线程休眠的时间，该时间使用毫秒为单位。当休眠时间结束后，线程自动被唤醒，进入就绪状态。

sleep()方法的语法格式：

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

上述代码不一定会使线程在2秒之内进入就绪状态。由于sleep()方法的执行有可能抛出InterruptedException异常，所以将sleep()方法的调用放在try…catch块中。虽然使用了sleep()方法的线程在一段时间内会醒来，但是并不能保证它醒来后进行运行状态，只能保证它进入就绪状态。

sleep()方法让正在执行的线程主动让出CPU（然后CPU就可以去执行其他任务），在sleep指定时间后CPU再回到该线程继续往下执行(注意：sleep方法只让出了CPU，而并不会释放监视器锁！！！)。

#### 线程的等待

Object类的wait()方法用于将正在执行的线程跳转到等待状态。不带参数的wait()方法会将线程设置为等待状态，如果需要将等待状态的线程唤醒，需要其他线程调用Object类的notify()或notifyAll()方法才能唤醒。（其中notify()用于唤醒单个等待的线程，而notifyAll()用于唤醒所有等待的线程。）如果给wait()添加一个参数（毫秒数），则该线程会在给定的毫秒数之后自动唤醒，并处于就绪状态。

**public** **synchronized** **void** getNum(){

**for** (**int** i = 0; i < nums.length; i++) {

System.***out***.println(nums[i]);

**try** {

//设置当线程调用该方法是先唤醒其他处于等待状态的线程，然后再将该线程设置为等待状态

**this**.notify();

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

注意：当设置线程处于等待状态后（wait()方法），如果需要将线程手动唤醒，需要其他的线程来唤醒处于等待队列中的线程。处于等待状态的线程无法自己将自己唤醒。

**public** **synchronized** **void** getNum(){

**for** (**int** i = 0; i < nums.length; i++) {

System.***out***.println(nums[i]);

**try** {

//设置当线程调用该方法是先唤醒其他处于等待状态的线程，

//设置调用该方法的线程先处于等待状态，2秒后自动处于就绪状态，等待获取监视器锁。

**this**.notify();

**this**.wait(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

在调用wait()方法之前，该线程必须要获得该对象的监视器锁，即Object类的wait()、notify()和notifyAll()方法只能与synchronized关键字一起使用，否则会抛出java.lang.IllegalMonitorStateException的异常。因为这些方法依赖于monitor对象。而synchronized关键字可以实现与monitor对象的交互。（Synchronized的语义底层是通过一个monitor的对象来完成）

在Java中，每个类和对象都具有一个monitor对象（监视器锁），且在同一时间只允许有一个线程获取该对象的监视器锁（monitor对象）。获取了该监视器锁的线程会执行访问该对象资源的代码。其余需要访问该对象资源的线程则会处于阻塞状态，直到该线程执行完毕后，释放了该对象的监视器锁。此时，其余的线程会去试图获取该监视器锁。

**wait()方法：**

public final void wait() throws InterruptedException,IllegalMonitorStateException

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。在调用wait（）之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用wait（）方法。进入wait（）方法后，当前线程释放锁。在从wait（）返回前，线程与其他线程竞争重新获得锁。如果调用wait（）时，没有持有适当的锁，则抛出IllegalMonitorStateException，它是RuntimeException的一个子类，因此，不需要try-catch结构。

**wait(long)和wait(long,int)方法：**

显然，这两个方法是设置等待超时时间的，后者在超值时间上加上ns，精度也难以达到，因此，该方法很少使用。对于前者，如果在等待线程接到通知或被中断之前，已经超过了指定的毫秒数，则它通过竞争重新获得锁，并从wait(long)返回。另外，需要知道，如果设置了超时时间，当wait()返回时，我们不能确定它是因为接到了通知还是因为超时而返回的，因为wait()方法不会返回任何相关的信息。但一般可以通过设置标志位来判断，在notify之前改变标志位的值，在wait()方法后读取该标志位的值来判断，当然为了保证notify不被遗漏，我们还需要另外一个标志位来循环判断是否调用wait()方法。

**notify()方法：**

public final native void notify() throws IllegalMonitorStateException

该方法也要在同步方法或同步块中调用，即在调用前，线程也必须要获得该对象的对象级别锁，的如果调用notify()时没有持有适当的锁，也会抛出IllegalMonitorStateException。

该方法用来通知那些可能等待该对象的对象锁的其他线程。如果有多个线程等待，则线程规划器任意挑选出其中一个wait()状态的线程来发出通知，并使它等待获取该对象的对象锁（notify后，当前线程不会马上释放该对象锁，wait所在的线程并不能马上获取该对象锁，要等到程序退出synchronized代码块后，当前线程才会释放锁，wait所在的线程也才可以获取该对象锁），但不惊动其他同样在等待被该对象notify的线程们。当第一个获得了该对象锁的wait线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程由于没有得到该对象的通知，会继续阻塞在wait状态，直到这个对象发出一个notify或notifyAll。这里需要注意：它们等待的是被notify或notifyAll，而不是锁。这与下面的notifyAll()方法执行后的情况不同。

**notifyAll()方法：**

public final native void notifyAll() throws IllegalMonitorStateException

该方法与notify()方法的工作方式相同，重要的一点差异是：

notifyAll使所有原来在该对象上wait的线程统统退出wait的状态（即全部被唤醒，不再等待notify或notifyAll，但由于此时还没有获取到该对象锁，因此还不能继续往下执行），变成等待获取该对象上的锁，一旦该对象锁被释放（notifyAll线程退出调用了notifyAll的synchronized代码块的时候），他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出synchronized代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

**深入理解：**

如果线程调用了对象的wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的notifyAll()方法（唤醒所有wait线程）或notify()方法（只随机唤醒一个wait线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了synchronized代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

#### 线程的加入

指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。

如果当前某程序为多线程程序，假如存在一个线程A，现在需要插入线程B，并要求线程B先执行完毕，然后再继续执行线程A，此时可以使用Thread类中的join()方法来完成。

当某个线程使用join()方法加入到另一个线程时，另一个线程会等待该线程支持完毕再继续执行。（注意：在被加入的线程必须在被启用后在加入，如果没有启用则依然会按照并发执行。）

**没有使用join()方法的情况：**

//通过Thread类的构造函数及Runnable接口的内部类方法定义第一个线程

Thread thread1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(3000); //设置第一个线程运行是先休眠3秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("第一个线程启动！");

}

});

执行结果：

//定义的第二个线程

Thread thread2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

thread1.start(); //启动第一个线程

System.***out***.println("第二个线程启动！");

}

});

thread2.start(); //启动第二个线程

**使用join()方法后：**

//通过Thread类的构造函数及Runnable接口的内部类方法定义第一个线程

Thread thread1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(3000); //设置第一个线程运行是先休眠3秒

} **catch** (InterruptedException e) {

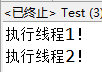
e.printStackTrace();

}

System.***out***.println("第一个线程启动！");

}

});

执行结果：

//定义的第二个线程

Thread thread2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

thread1.start(); //启动第一个线程

**try** {

thread1.join(); //将第一个线程加入到第二个线程中

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("第二个线程启动！");

}

});

thread2.start(); //启动第二个线程

join()方法实现是通过wait（小提示：Object 提供的方法）。 当thread2线程调用thread1.join时候，thread2线程会获得线程对象thread1的锁（wait意味着拿到该对象的锁),调用该对象的wait(等待时间)，直到该对象唤醒thread2线程 ，比如退出后。这就意味着thread2线程调用thread1.join时，必须能够拿到线程thread1对象的锁。

从代码上看，如果线程被生成了，但还未被起动，调用它的 join() 方法是没有作用的，将直接继续向下执行

JDK中join()方法的源码：

**public** **final** **void** join() **throws** InterruptedException {

join(0);

}

**public** **final** **synchronized** **void** join(**long** millis) **throws** InterruptedException {

**long** base = System.*currentTimeMillis*();

**long** now = 0;

**if** (millis < 0) {

**throw** **new** IllegalArgumentException("timeout value is negative");

}

**if** (millis == 0) {

**while** (isAlive()) {

wait(0);

}

} **else** {

**while** (isAlive()) {

**long** delay = millis - now;

**if** (delay <= 0) {

**break**;

}

wait(delay);

now = System.*currentTimeMillis*() - base;

}

}

}

#### 线程的中断

如果想让某个线程停止，可以在run()方法中使用无限循环的形式，然后使用一个布尔型标记控制循环的停止。

#### 线程的礼让

Thread类中提供了一种礼让方法，使用yield()静态方法表示，它只是给当前正处于运行状态下的线程一个提醒，告知它可以将资源礼让给其他线程，但这仅仅是一种暗示，没有任何一种机制保证当前线程会将资源礼让。

yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

#### sleep()与wait()的区别

1. sleep是Thread类的方法，而wait是Object类的方法
2. sleep方法让当前线程对象处于休眠状态，当定义的时间到达后会继续执行线程。而wait方法（不带参数的）则会将当前线程设置为等待状态，直到其他线程调用Object类的notify或notifyAll方法来唤醒。
3. sleep方法是一个静态方法，直接使用Thread.sleep(n)来调用，而wait方法使用对象来调用。
4. sleep方法可以在任何地方使用，而wait方法只能在被synchronized关键字修饰的方法或代码块中使用，在这之外使用会抛出java.lang.IllegalMonitorStateException的异常
5. sleep方法会让当前正在执行的线程让出CPU资源供其他代码使用（不会抛出监视器锁），当休眠时间到达后又会重新获取CPU资源执行线程；而wait方法则会让当前正在执行的线程放弃监视器锁的使用权限，使其处于等待状态。直到其他线程调用Object类的notify或notifyAll方法来唤醒线程，使其处于就绪状态。

### 线程的优先级

每个线程都具有各自的优先级，线程的优先级可以在程序中表明该线程的重要性，如果有很多线程处于就绪状态，系统会根据优先级来决定首先使那个线程进入运行状态。但这并不意味着低优先级的线程得不到运行，而只是它允许的几率比较小，如垃圾回收线程的优先级就较低。

在多任务操作系统中，每个线程都会得到一小段CPU时间片运行，在时间结束时，将轮换另一个线程进入运行状态，这时系统会选择与当前线程优先级相同的线程予以运行。系统始终选择就绪状态下优先级较高的线程进入运行状态。

线程的优先级可以使用setPriority()方法调整，如果使用该方法设置的优先级不在1~10之内，将产生一个异常。

Thread类中也提供了用于设置线程优先级的三个常量：

MAX\_PRIORITY：线程可以具有的最高优先级。（10）

MIN\_PRIORITY：线程可以具有的最低优先级。（1）

NORM\_PRIORITY：分配给线程的默认优先级。（5）

### 线程同步

在单线程程序中，每次只能做一件事情，后面的事情需要等待前面的事情完成后才可以进行；但是如果使用多线程程序，就会发生两个线程抢占资源的问题。所以在多线程编程中，需要防止这些资源访问的冲突。Java提供线程同步的机制来防止资源访问的冲突。

#### 线程安全

线程的安全问题来源于多个线程同时存取单一对象的数据，出现数据错乱的情况。不能保证数据的准确性、一致性和可见性。

例如：两个线程同时访问一个对象资源，当线程A修改对象资源的数据，但是线程B在线程A修改数据之前获取了原始的数据，线程B就是拿着原始数据进行业务处理，最终会导致数据的混乱。

#### 线程同步机制

如何解决资源共享的问题（即多个线程同时访问一个资源，保证资源数据的正确性和每个线程对资源数据执行的结果是对每个线程都是可见的）？基本上所有解决多线程资源冲突问题都会采用给定时间只允许一个线程访问共享资源，这时就需要给共享资源上一道锁。

##### synchronized关键字

**同步代码块**

在Java中提供了同步机制，可以有效的防止资源冲突。同步机制使用synchronized关键字(JVM提供了synchronized关键字的实现)。

语法格式：

**synchronized**(Objec){

//......

}

通常是将共享资源的操作放置在synchronized定义的区域内，这样当其他线程也获取到这个锁时，必须等待锁被释放时才能进入该区域。Object为需要获取那个对象的监视器锁，每个对象都存在一个标志位，并具有两个值，分别为0和1.一个线程运行到同步块时首先检查该对象的标志位，如果是0状态，表明此同步块中存在其他线程在运行。这时该线程处于就绪状态，直到处于同步块中的线程执行完同步块中的代码为止。这时该对象的标志位被设置为1，该线程才能执行同步块中的代码，并将Object对象的标志位设置为0,放置其他线程执行同步块中的代码。

**public** **class** ThreadTest {

**public** **void** test(){

**synchronized**(**this**){

System.***out***.println("执行资源数据");

}

}

}

代码块必须依赖于方法中，不能独立存在。

**同步方法**

同步方法就是在方法前面修饰synchronized关键字的方法

语法格式：

**public** **synchronized void** test(){

}

当某个对象调用了同步方法时，该对象上的其他同步方法必须等待该同步方法执行完毕才能被执行。必须将每个能访问共享资源的方法修饰为synchronized，否则就会出错。synchronized关键字同样可以修饰静态方法。

**public** **synchronized** **void** test(){

**int** num = 0;

System.***out***.println("这是共享资源" + num++);

}

##### Lock接口

Java5.0中新增了一个java.util.concurrent包来支持同步。Lock类来实现和synchronized一样的功能，并且还提供了Condition来显示线程间通信。Lock类是Java类来提供的功能，丰富的api使得Lock类的同步功能比synchronized的同步更强大。

1. **Lock类**

Lock类实际上是一个接口，我们在实例化的时候实际上是实例化实现了该接口的类Lock lock = new ReentrantLock();我们可以用lock.lock来加锁，用lock.unlock来释放锁。在两者中间放置需要同步处理的代码。

**public** **class** MyConditionService {

**private** Lock lock = **new** ReentrantLock();

**public** **void** testMethod(){

lock.lock();

**for** (**int** i = 0 ;i < 5;i++){

System.***out***.println("ThreadName = " + Thread.*currentThread*().getName() + (" " + (i + 1)));

}

lock.unlock();

}

}

测试代码：

MyConditionService service = **new** MyConditionService();

**new** Thread(service::testMethod).start();

**new** Thread(service::testMethod).start();

**new** Thread(service::testMethod).start();

**new** Thread(service::testMethod).start();

**new** Thread(service::testMethod).start();

Thread.sleep(1000 \* 5);

Lock接口及实现类ReentrantLock还提供了其他关于锁机制的其他实现。

**实现公平锁**

在实例化锁对象的时候，构造方法有2个，一个是无参构造方法，一个是传入一个boolean变量的构造方法。当传入值为true的时候，该锁为公平锁。默认不传参数是非公平锁。

公平锁：按照线程加锁的顺序来获取锁

非公平锁：随机竞争来得到锁

**获取当前线程锁定的个数**

getHoldCount()方法来获取当前线程的锁定个数。所谓锁定个数就是当前线程调用lock方法的次数。一般一个方法只会调用一个lock方法，但是有可能在同步代码中还有调用了别的方法，那个方法内部有同步代码。这样，getHoldCount()返回值就是大于1。

1. **Condition类**

Condition是Java提供了来实现等待/通知的类，Condition类还提供比wait/notify更丰富的功能，Condition对象是由lock对象所创建的。但是同一个锁可以创建多个Condition的对象，即创建多个对象监视器。这样的好处就是可以指定唤醒线程。notify唤醒的线程是随机唤醒一个。

**public** **class** ConditionWaitNotifyService {

**private** Lock lock = **new** ReentrantLock();

**public** Condition condition = lock.newCondition();

**public** **void** await(){

**try**{

lock.lock();//给线程获取锁

System.***out***.println("await的时间为 " + System.*currentTimeMillis*());

condition.await();//将线程设置为等待状态

System.***out***.println("await结束的时间" + System.*currentTimeMillis*());

}**catch** (Exception e){

e.printStackTrace();

}**finally** {

lock.unlock();//释放当前线程的锁

}

}

**public** **void** signal(){

**try**{

lock.lock();//给当前线程获取锁

System.***out***.println("sign的时间为" + System.*currentTimeMillis*());

condition.signal();//唤醒一个等待的线程

}**finally** {

lock.unlock();//释放当前线程的锁

}

}

}

测试类：

ConditionWaitNotifyService service = **new** ConditionWaitNotifyService();

**new** Thread(service::await).start();

Thread.sleep(1000 \* 3);

service.signal();

Thread.sleep(1000);

condition对象通过lock.newCondition()来创建，用condition.await()来实现让线程等待，是线程进入阻塞。用condition.signal()来实现唤醒线程。唤醒的线程是用同一个conditon对象调用await()方法而进入阻塞。并且和wait/notify一样，await()和signal()也是在同步代码区内执行。

对于等待/通知机制，简化而言，就是等待一个条件，当条件不满足时，就进入等待，等条件满足时，就通知等待的线程开始执行。为了实现这种功能，需要进行wait的代码部分与需要进行通知的代码部分必须放在同一个对象监视器里面。执行才能实现多个阻塞的线程同步执行代码，等待与通知的线程也是同步进行。对于wait/notify而言，对象监视器与等待条件结合在一起 即synchronized（对象）利用该对象去调用wait以及notify。但是对于Condition类，是对象监视器与条件分开，Lock类来实现对象监视器，condition对象来负责条件，去调用await以及signal。

1. **ReentrantReadWriteLock类**

Lock类除了提供了ReentrantLock的锁以外，还提供了ReentrantReadWriteLock的锁。读写锁分成两个锁，一个锁是读锁，一个锁是写锁。读锁与读锁之间是共享的，读锁与写锁之间是互斥的，写锁与写锁之间也是互斥的。

下面的读读共享的例子：

**public** **class** ReadReadService {

**private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();

**public** **void** read(){

**try**{

**try**{

lock.readLock().lock();

System.***out***.println("获得读锁" + Thread.*currentThread*().getName() +

" " + System.*currentTimeMillis*());

Thread.*sleep*(1000 \* 10);

}**finally** {

lock.readLock().unlock();

}

}**catch** (InterruptedException e){

e.printStackTrace();

}

}

}

测试的代码：

ReadReadService service = **new** ReadReadService();

Thread a = **new** Thread(service::read);

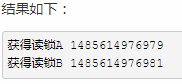
a.setName("A");

Thread b = **new** Thread(service::read);

b.setName("B");

a.start();

b.start();



两个线程几乎同时执行同步代码。

下面的例子是写写互斥的例子：

**public** **class** WriteWriteService {

**private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();

**public** **void** write(){

**try**{

**try**{

lock.writeLock().lock();

System.***out***.println("获得写锁" + Thread.*currentThread*().getName() +

" " +System.*currentTimeMillis*());

Thread.*sleep*(1000 \* 10);

}**finally** {

lock.writeLock().unlock();

}

}**catch** (InterruptedException e){

e.printStackTrace();

}

}

}

测试类：

WriteWriteService service = **new** WriteWriteService();

Thread a = **new** Thread(service::write);

a.setName("A");

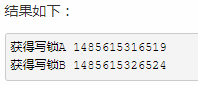
Thread b = **new** Thread(service::write);

b.setName("B");

a.start();

b.start();

Thread.sleep(1000 \* 30);



两个线程同步执行代码

读写互斥的例子：

**public** **class** WriteReadService {

**private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();

**public** **void** read(){

**try**{

**try**{

lock.readLock().lock();

System.***out***.println("获得读锁" + Thread.*currentThread*().getName()

+ " " + System.*currentTimeMillis*());

Thread.*sleep*(1000 \* 10);

}**finally** {

lock.readLock().unlock();

}

}**catch** (InterruptedException e){

e.printStackTrace();

}

}

**public** **void** write(){

**try**{

**try**{

lock.writeLock().lock();

System.***out***.println("获得写锁" + Thread.*currentThread*().getName()

+ " " + System.*currentTimeMillis*());

Thread.*sleep*(1000 \* 10);

}**finally** {

lock.writeLock().unlock();

}

}**catch** (InterruptedException e){

e.printStackTrace();

}

}

}

测试代码：

WriteReadService service = **new** WriteReadService();

Thread a = **new** Thread(service::write);

a.setName("A");

a.start();

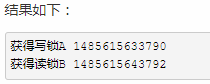
Thread.sleep(1000);

Thread b = **new** Thread(service::read);

b.setName("B");

b.start();

Thread.sleep(1000 \* 30);



两个线程读写之间也是同步执行代码。

### 线程数据传递

在传统的同步开发模式下，当我们调用一个函数时，通过这个函数的参数将数据传入，并通过这个函数的返回值来返回最终的计算结果。但在多线程的异步开发模式下，数据的传递和返回和同步开发模式有很大的区别。由于线程的运行和结束是不可预料的，因此，在传递和返回数据时就无法像函数一样通过函数参数和return语句来返回数据。

#### 通过构造方法传递数据

在创建线程时，必须要建立一个Thread类或其子类的实例。因此，我们不难想到在调用start方法之前通过线程类的构造方法将数据传入线程。并将传入的数据使用类变量保存起来，以便线程使用(其实就是在run方法中使用)。

下面的代码演示了如何通过构造方法来传递数据：

**public** **class** ThreadTest **extends** Thread {

**private** String name;

**public** ThreadTest(){}

**public** ThreadTest(String name){

**this**.name = name;

}

**public** **void** run(){

System.***out***.println("Hello" + **this**.name);

}

}

**public** **static** **void** main(String[] args) {

ThreadTest test = **new** ThreadTest("world");

test.start();

}

由于这种方法是在创建线程对象的同时传递数据的，因此，在线程运行之前这些数据就就已经到位了，这样就不会造成数据在线程运行后才传入的现象。如果要传递更复杂的数据，可以使用集合、类等数据结构。使用构造方法来传递数据虽然比较安全，但如果要传递的数据比较多时，就会造成很多不便。由于Java没有默认参数，要想实现类似默认参数的效果，就得使用重载，这样不但使构造方法本身过于复杂，又会使构造方法在数量上大增。因此，要想避免这种情况，就得通过类方法或类变量来传递数据

#### 通过变量和方法传递数据

向对象中传入数据一般有两次机会，第一次机会是在建立对象时通过构造方法将数据传入，另外一次机会就是在类中定义一系列的public的方法或变量（也可称之为字段）。然后在建立完对象后，通过对象实例逐个赋值。

下面的代码是对MyThread1类的改版，使用了一个setName方法来设置 name变量：

**public** **class** ThreadTest **implements** Runnable {

**private** String name;

**public** **void** setName(String name){

**this**.name = name;

}

**public** **void** run(){

System.***out***.println("Hello" + **this**.name);

}

}

**public** **static** **void** main(String[] args) {

ThreadTest test = **new** ThreadTest();

test.setName("world");

Thread thread = **new** Thread(test);

thread.start();

}

#### 通过回调函数传递数据

上面讨论的两种向线程中传递数据的方法是最常用的。但这两种方法都是main方法中主动将数据传入线程类的。这对于线程来说，是被动接收这些数据的。然而，在有些应用中需要在线程运行的过程中动态地获取数据，如在下面代码的run方法中产生了3个随机数，然后通过Work类的process方法求这三个随机数的和，并通过Data类的value将结果返回。从这个例子可以看出，在返回value之前，必须要得到三个随机数。也就是说，这个 value是无法事先就传入线程类的。

**public** **class** Data{

**public** **int** value = 0;

}

**public** **class** Work{

**public** **void** process(Data data, Integer…numbers){

**for** (**int** n : numbers) {

data.value += n;

}

}

}

**public** classMyThread3 **extends** Thread{

**private** Work work;

**public** MyThread3(Work work){

**this**.work = work;

}

**public** **void** run(){

Random rd = **new** Random();

Data data = **new** Data();

**int** n1 = rd.nextInt(1000);

**int** n2 = rd.nextInt(2000);

**int** n3 = rd.nextInt(3000);

work.process(data, n1, n2, n3); //使用回调函数

System.out.println(String.valueOf(n1) + "+" + String.valueOf(n2) + "+" String.valueOf(n3) + "=" + data.value);

}

}

**public** **static** **void** main(String[] args) {

Thread thread = **new** MyThread3(**new** Work);

thread.start();

}

### Callable接口

java.util.concurrent.Callable<V>接口类似于 Runnable，也可以用于实现线程类。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常。而Callable功能更强大一些，实现了Callable接口的线程类，执行线程后可以返回结果，且可以抛出经过检查的异常。

Callable接口源码：

**public** **interface** Callable<V> {

/\*\*

\* Computes a result, or throws an exception if unable to do so.

\*

\* **@return** computed result

\* **@throws** Exception if unable to compute a result

\*/

V call() **throws** Exception;

}

可以看到，这是一个泛型接口，call()函数返回的类型就是客户程序传递进来的V类型。其中Callable接口定义了一个call()方法，所有实现了该接口的类必须实现此方法。

**public** **class** ThreadTest **implements** Callable<V> {

@Override

**public** V call() **throws** Exception {

**return** **null**;

}

}

Callable并不像Runnable那样通过Thread的start方法就能启动实现类的run方法，所以它通常利用ExecutorService接口的submit方法去启动call方法自执行任务，而ExecutorService的submit又返回一个Future类型的结果，因此Callable通常也与Future一起使用

**public** **static** **void** main(String[] args) {

ExecutorService pool = Executors.*newCachedThreadPool*();

Future<String> future = pool.submit(**new** Callable<String>() {

@Override

**public** String call() **throws** Exception {

**return** **null**;

}

});

}

或者利用FutureTask封装Callable再由Thread去启动（少用）

**public** **static** **void** main(String[] args) {

FutureTask<String> task = **new** FutureTask<>(**new** Callable<String>() {

**public** String call() **throws** Exception {

**return** **null**;

}

});

Thread thread = **new** Thread(task);

thread.start();

}

**Runnable与Callable不同点：**

1. 两者最大的不同点是：实现Callable接口的任务线程能返回执行结果；而实现Runnable接口的任务线程不能返回结果；
2. Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛；

### Future接口

java.util.concurrent.Future<V>接口保存异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get 方法来获取结果，如有必要，计算完成前可以阻塞此方法。取消则由 cancel 方法来执行。还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。

**方法：**

**boolean cancel(boolean mayInterruptIfRunning)**

试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。

**boolean isCancelled()**

判断任务是否在正常执行完前被取消的，如果是则返回true

**boolean isDone()**

如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。

**V get()**

如有必要，等待计算完成，然后获取其结果。

**V get(long timeout, TimeUnit unit)**

如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。timeout - 等待的最大时间、unit - timeout 参数的时间单位。

**public** **static** **void** main(String[] args) {

ExecutorService pool = Executors.*newCachedThreadPool*();

Future<String> future = pool.submit(**new** Callable<String>() {

@Override

**public** String call() **throws** Exception {

**return** **null**;

}

});

**try** {

//等待1秒钟后获取返回值

System.***out***.println(future.get(1, TimeUnit.***SECONDS***));

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (ExecutionException e) {

e.printStackTrace();

} **catch** (TimeoutException e) {

e.printStackTrace();

}

}

### FutureTask类

java.util.concurrent.FutureTask<V>类可取消的异步计算。利用开始和取消计算的方法、查询计算是否完成的方法和获取计算结果的方法，此类提供了对 Future 的基本实现。仅在计算完成时才能获取结果；如果计算尚未完成，则阻塞 get 方法。一旦计算完成，就不能再重新开始或取消计算。该类实现了RunnableFuture<V>接口，而RunnableFuture<V>接口又继承自Runnable和Future<V>接口。

可使用 FutureTask 包装 Callable 或 Runnable 对象。因为 FutureTask 实现了 Runnable，所以可将 FutureTask 提交给 Executor 执行。

除了作为一个独立的类外，此类还提供了 protected 功能，这在创建自定义任务类时可能很有用。

**构造方法：**

**public FutureTask(Callable<V> callable)**

创建一个 FutureTask，一旦运行就执行给定的 Callable。callable - 可调用的任务。

**public FutureTask(Runnable runnable, V result)**

创建一个 FutureTask，一旦运行就执行给定的 Runnable，并安排成功完成时 get 返回给定的结果 。runnable - 可运行的任务、result - 成功完成时要返回的结果。如果不需要特定的结果，则考虑使用下列形式的构造：Future<?> f = new FutureTask<Object>(runnable, null)

### 守护线程

守护线程（Daemon Thread）区分与用户线程（User Thread），主要为JVM中的非守护线程服务。当JVM中存在有未结束的非守护线程，所有守护线程都会全部工作。只有当JVM中所有非守护线程结束工作，则守护线程线程会随着JVM一起结束工作。守护线程最经典的应用是GC（垃圾回收器）。

守护线程并非只有虚拟机内部提供，用户在编程时也可以自己设置守护线程。

//创建一个线程

Thread thread = **new** Thread();

//将其定义为守护线程，false为非守护线程

thread.setDaemon(**true**);

//判断是否是守护线程

thread.isDaemon();

**注意事项：**

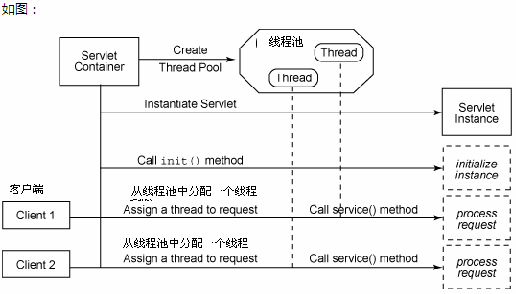
1. thread.setDaemon(true)必须在thread.start()之前设置，否则会抛出IllegalThreadStateException异常。不能把正在运行的常规线程设置为守护线程。
2. 在Daemon线程中产生的新线程也是Daemon线程。
3. 不是所有的应用都可以分配给Daemon来进行服务，比如读写操作或者计算逻辑。

**实际应用例子：**

在使用长连接的comet服务端推送技术中，消息推送线程设置为守护线程，服务于ChatServlet的servlet用户线程，在servlet的init启动消息线程，servlet一旦初始化后，一直存在服务器，servlet摧毁后,消息线程自动退出

容器收到一个Servlet请求，调度线程从线程池中选出一个工作者线程,将请求传递给该工作者线程，然后由该线程来执行Servlet的 service方法。当这个线程正在执行的时候,容器收到另外一个请求,调度线程同样从线程池中选出另一个工作者线程来服务新的请求,容器并不关心这个请求是否访问的是同一个Servlet.当容器同时收到对同一个Servlet的多个请求的时候，那么这个Servlet的service()方法将在多线程中并发执行。

Servlet容器默认采用单实例多线程的方式来处理请求，这样减少产生Servlet实例的开销，提升了对请求的响应时间，对于Tomcat可以在server.xml中通过<Connector>元素设置线程池中线程的数目。



## 多线程进阶

### 核心理论

并发编程是Java程序员最重要的技能之一，也是最难掌握的一种技能。它要求编程者对计算机最底层的运作原理有深刻的理解，同时要求编程者逻辑清晰、思维缜密，这样才能写出高效、安全、可靠的多线程并发程序。本系列会从线程间协调的方式（wait、notify、notifyAll）、Synchronized及Volatile的本质入手，详细解释JDK为我们提供的每种并发工具和底层实现机制。在此基础上，我们会进一步分析java.util.concurrent包的工具类，包括其使用方式、实现源码及其背后的原理。本文是该系列的第一篇文章，是这系列中最核心的理论部分，之后的文章都会以此为基础来分析和解释。

#### 共享性

数据共享性是线程安全的主要原因之一。如果所有的数据只是在线程内有效，那就不存在线程安全性问题，这也是我们在编程的时候经常不需要考虑线程安全的主要原因之一。但是，在多线程编程中，数据共享是不可避免的。最典型的场景是数据库中的数据，为了保证数据的一致性，我们通常需要共享同一个数据库中数据，即使是在主从的情况下，访问的也同一份数据，主从只是为了访问的效率和数据安全，而对同一份数据做的副本。

我们现在，通过一个简单的示例来演示多线程下共享数据导致的问题：

**public** **class** ShareData {

**public** **static** **int** *count* = 0;

**public** **void** addCount(){

*count*++;

}

**public** **static** **void** main(String[] args){

**final** ShareData data = **new** ShareData();

**for** (**int** i = 0; i < 10; i++) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

//进入的时候暂停1毫秒，增加并发问题出现的几率

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**for** (**int** j = 0; j < 100; j++) {

data.addCount();

}

System.***out***.println(*count* + " ");

}

}).start();;

}

**try** {

//主线程暂停3秒，以保证上面的程序执行完成

Thread.*sleep*(3000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("count = " + *count*);

}

}

上述代码的目的是对count进行加一操作，执行1000次，不过这里是通过10个线程来实现的，每个线程执行100次，正常情况下，应该输出1000。不过，如果你运行上面的程序，你会发现结果却不是这样。下面是某次的执行结果（每次运行的结果不一定相同，有时候也可能获取到正确的结果）：



可以看出，对共享变量操作，在多线程环境下很容易出现各种意想不到的的结果。

#### 互斥性

资源互斥是指同时只允许一个访问者对其进行访问，具有唯一性和排它性。我们通常允许多个线程同时对数据进行读操作，但同一时间内只允许一个线程对数据进行写操作。所以我们通常将锁分为共享锁和排它锁，也叫做读锁和写锁。如果资源不具有互斥性，即使是共享资源，我们也不需要担心线程安全。例如，对于不可变的数据共享，所有线程都只能对其进行读操作，所以不用考虑线程安全问题。但是对共享数据的写操作，一般就需要保证互斥性，上述例子中就是因为没有保证互斥性才导致数据的修改产生问题。Java 中提供多种机制来保证互斥性，最简单的方式是使用Synchronized。

现在我们在上面程序中加上Synchronized再执行：

**public** **class** ShareData {

**public** **static** **int** *count* = 0;

//增加 synchronized 关键字

**public** **synchronized** **void** addCount(){

*count*++;

}

**public** **static** **void** main(String[] args){

**final** ShareData data = **new** ShareData();

**for** (**int** i = 0; i < 10; i++) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

//进入的时候暂停1毫秒，增加并发问题出现的几率

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**for** (**int** j = 0; j < 100; j++) {

data.addCount();

}

System.***out***.println(*count* + " ");

}

}).start();;

}

**try** {

//主线程暂停3秒，以保证上面的程序执行完成

Thread.*sleep*(3000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("count = " + *count*);

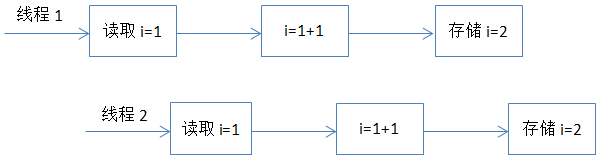
}

}

现在再执行上述代码，会发现无论执行多少次，返回的最终结果都是1000。

#### 原子性

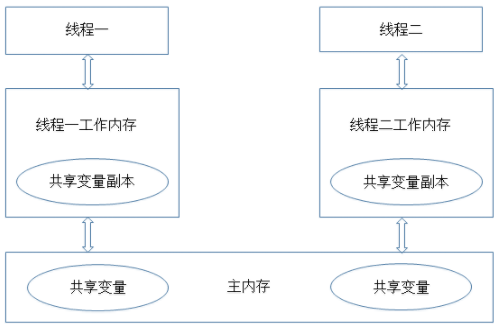
原子性就是指对数据的操作是一个独立的、不可分割的整体。换句话说，就是一次操作，是一个连续不可中断的过程，数据不会执行的一半的时候被其他线程所修改。保证原子性的最简单方式是操作系统指令，就是说如果一次操作对应一条操作系统指令，这样肯定可以能保证原子性。但是很多操作不能通过一条指令就完成。例如，对long类型的运算，很多系统就需要分成多条指令分别对高位和低位进行操作才能完成。还比如，我们经常使用的整数 i++ 的操作，其实需要分成三个步骤：（1）读取整数 i 的值；（2）对 i 进行加一操作；（3）将结果写回内存。这个过程在多线程下就可能出现如下现象：



这也是代码段一执行的结果为什么不正确的原因。对于这种组合操作，要保证原子性，最常见的方式是加锁，如Java中的Synchronized或Lock都可以实现，代码段二就是通过Synchronized实现的。除了锁以外，还有一种方式就是CAS（Compare And Swap），即修改数据之前先比较与之前读取到的值是否一致，如果一致，则进行修改，如果不一致则重新执行，这也是乐观锁的实现原理。不过CAS在某些场景下不一定有效，比如另一线程先修改了某个值，然后再改回原来值，这种情况下，CAS是无法判断的。

#### 可见性

要理解可见性，需要先对JVM的内存模型有一定的了解，JVM的内存模型与操作系统类似，如图所示：



从这个图中我们可以看出，每个线程都有一个自己的工作内存（相当于CPU高级缓冲区，这么做的目的还是在于进一步缩小存储系统与CPU之间速度的差异，提高性能），对于共享变量，线程每次读取的是工作内存中共享变量的副本，写入的时候也直接修改工作内存中副本的值，然后在某个时间点上再将工作内存与主内存中的值进行同步。这样导致的问题是，如果线程1对某个变量进行了修改，线程2却有可能看不到线程1对共享变量所做的修改。

通过下面这段程序我们可以演示一下不可见的问题：

**public** **class** VisibilityTest {

**private** **static** **boolean** *ready*;

**private** **static** **int** *number*;

**private** **static** **class** ReaderThread **extends** Thread {

**public** **void** run() {

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**if** (!*ready*) {

System.***out***.println(*ready*);

}

System.***out***.println(*number*);

}

}

**private** **static** **class** WriterThread **extends** Thread {

**public** **void** run() {

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

*number* = 100;

*ready* = **true**;

}

}

**public** **static** **void** main(String[] args) {

**new** WriterThread().start();

**new** ReaderThread().start();

}

}

从直观上理解，这段程序应该只会输出100，ready的值是不会打印出来的。实际上，如果多次执行上面代码的话，可能会出现多种不同的结果，下面是我运行出来的某两次的结果：

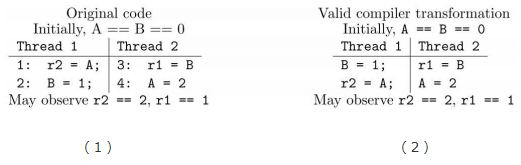
当然，这个结果也只能说是有可能是可见性造成的，当写线程（WriterThread）设置ready=true后，读线程（ReaderThread）看不到修改后的结果，所以会打印false，对于第二个结果，也就是执行if (!ready)时还没有读取到写线程的结果，但执行System.out.println(ready)时读取到了写线程执行的结果。不过，这个结果也有可能是线程的交替执行所造成的。Java 中可通过Synchronized或Volatile来保证可见性

#### 有序性

为了提高性能，编译器和处理器可能会对指令做重排序。重排序可以分为三种：

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

我们可以直接参考一下JSR 133 中对重排序问题的描述：



先看上图中的（1）源码部分，从源码来看，要么指令 1 先执行要么指令 3先执行。如果指令 1 先执行，r2不应该能看到指令 4 中写入的值。如果指令 3 先执行，r1不应该能看到指令 2 写的值。但是运行结果却可能出现r2==2，r1==1的情况，这就是“重排序”导致的结果。上图（2）即是一种可能出现的合法的编译结果，编译后，指令1和指令2的顺序可能就互换了。因此，才会出现r2==2，r1==1的结果。Java 中也可通过Synchronized或Volatile来保证顺序性。

### Synchronized及其实现原理

#### Synchronized的基本使用

Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：（1）确保线程互斥的访问同步代码（2）保证共享变量的修改能够及时可见（3）有效解决重排序问题。从语法上讲，Synchronized总共有三种用法：

1. 修饰普通方法
2. 修饰静态方法
3. 修饰代码块

接下来我就通过几个例子程序来说明一下这三种使用方式（为了便于比较，三段代码除了Synchronized的使用方式不同以外，其他基本保持一致）。

##### 没有设置同步的情况

**public** **class** SynchronizedTest {

**public** **void** method1(){ //定义方法method1

System.***out***.println("Method 1 start");

**try** {

System.***out***.println("Method 1 execute");

Thread.*sleep*(3000); //设置线程休眠3秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 1 end");

}

**public** **void** method2(){ //定义方法method2

System.***out***.println("Method 2 start");

**try** {

System.***out***.println("Method 2 execute");

Thread.*sleep*(1000); //设置线程休眠1秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 2 end");

}

}

**public** **static** **void** main(String[] args) {

**final** SynchronizedTest test = **new** SynchronizedTest();

**new** Thread(**new** Runnable() { //定义线程1

@Override

**public** **void** run() {

}

}).start();

**new** Thread(**new** Runnable() { //定义线程2

@Override

**public** **void** run() {

test.method2();

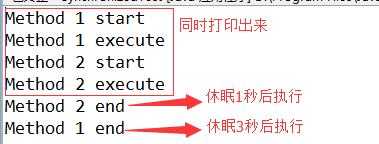
}

}).start();

}

test.method1();

执行结果如下，线程1和线程2同时进入执行状态，线程2执行速度比线程1快，所以线程2先执行完成，这个过程中线程1和线程2是同时执行的。



##### 对普通方法同步

**public** **class** SynchronizedTest {

**public synchronized** **void** method1(){ //定义方法method1

System.***out***.println("Method 1 start");

**try** {

System.***out***.println("Method 1 execute");

Thread.*sleep*(3000); //设置线程休眠3秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 1 end");

}

**public synchronized** **void** method2(){ //定义方法method2

System.***out***.println("Method 2 start");

**try** {

System.***out***.println("Method 2 execute");

Thread.*sleep*(1000); //设置线程休眠1秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 2 end");

}

}

**public** **static** **void** main(String[] args) {

**final** SynchronizedTest test = **new** SynchronizedTest();

**new** Thread(**new** Runnable() { //定义线程1

@Override

**public** **void** run() {

test.method1();

}

}).start();

**new** Thread(**new** Runnable() { //定义线程2

@Override

**public** **void** run() {

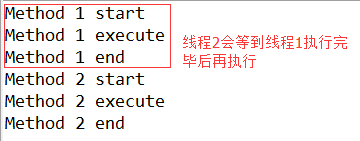
test.method2();

}

}).start();

}

执行结果如下，跟代码段一比较，可以很明显的看出，线程2需要等待线程1的method1执行完成才能开始执行method2方法。



##### 静态方法（类）同步

**public** **class** SynchronizedTest {

**public static synchronized** **void** method1(){ //定义方法method1

System.***out***.println("Method 1 start");

**try** {

System.***out***.println("Method 1 execute");

Thread.*sleep*(3000); //设置线程休眠3秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 1 end");

}

**public static synchronized** **void** method2(){ //定义方法method2

System.***out***.println("Method 2 start");

**try** {

System.***out***.println("Method 2 execute");

Thread.*sleep*(1000); //设置线程休眠1秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Method 2 end");

}

}

**public** **static** **void** main(String[] args) {

**final** SynchronizedTest test = **new** SynchronizedTest();

**final** SynchronizedTest test2 = **new** SynchronizedTest();

**new** Thread(**new** Runnable() { //定义线程1

@Override

**public** **void** run() {

test.method1();

}

}).start();

**new** Thread(**new** Runnable() { //定义线程2

@Override

**public** **void** run() {

test2.method2();

}

}).start();

}

执行结果如下，对静态方法的同步本质上是对类的同步（静态方法本质上是属于类的方法，而不是对象上的方法），所以即使test和test2属于不同的对象，但是它们都属于SynchronizedTest类的实例，所以也只能顺序的执行method1和method2，不能并发执行。



##### 代码块同步

**public** **class** SynchronizedTest {

//定义方法method1

**public** **void** method1() **throws** InterruptedException{

System.***out***.println("Method 1 start");

**synchronized** (**this**) { //同步代码块

System.***out***.println("Method 1 execute");

Thread.*sleep*(3000);

}

System.***out***.println("Method 1 end");

}

//定义方法method2

**public** **void** method2() **throws** InterruptedException{

System.***out***.println("Method 2 start");

**synchronized** (**this**) { //同步代码块

System.***out***.println("Method 2 execute");

Thread.*sleep*(1000);

}

System.***out***.println("Method 2 end");

}

}

**public** **static** **void** main(String[] args) {

**final** SynchronizedTest test = **new** SynchronizedTest();

**new** Thread(**new** Runnable() { //定义线程1

@Override

**public** **void** run() {

test.method1();

}

}).start();

**new** Thread(**new** Runnable() { //定义线程2

@Override

**public** **void** run() {

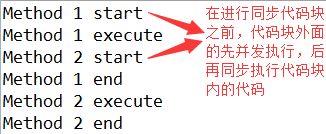
test.method2();

}

}).start();

}

执行结果如下，虽然线程1和线程2都进入了对应的方法开始执行，但是线程2在进入同步块之前，需要等待线程1中同步块执行完成。



#### Synchronized 原理

如果对上面的执行结果还有疑问，也先不用急，我们先来了解Synchronized的原理，再回头上面的问题就一目了然了。我们先通过反编译下面的代码来看看Synchronized是如何实现对代码块进行同步的：

**public** **class** SynchronizedDemo {

**public** **void** method(){

**synchronized** (**this**) {

System.***out***.println("Method 1 start");

}

}

}

反编译结果：



关于这两条指令的作用，我们直接参考JVM规范中描述：

monitorenter ：

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref, as follows:

• If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.

• If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry count.

• If another thread already owns the monitor associated with objectref, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

这段话的大概意思为：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit：

The thread that executes monitorexit must be the owner of the monitor associated with the instance referenced by objectref.

The thread decrements the entry count of the monitor associated with objectref. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

这段话的大概意思为：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过这两段描述，我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

我们再来看一下同步方法的反编译结果：

**public** **class** SynchronizedDemo {

**public** **void** method(){

**synchronized** (**this**) {

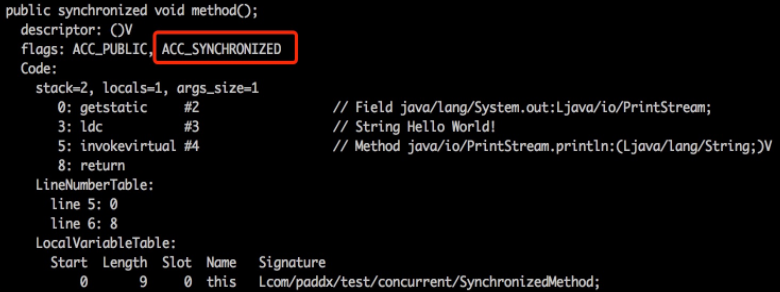
System.***out***.println("Method 1 start");

}

}

}

反编译结果：



从反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法被调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

#### 运行结果解释

有了对Synchronized原理的认识，再来看上面的程序就可以迎刃而解了。

1. 代码段2结果：

虽然method1和method2是不同的方法，但是这两个方法都进行了同步，并且是通过同一个对象去调用的，所以调用之前都需要先去竞争同一个对象上的锁（monitor），也就只能互斥的获取到锁，因此，method1和method2只能顺序的执行。

1. 代码段3结果：

虽然test和test2属于不同对象，但是test和test2属于同一个类的不同实例，由于method1和method2都属于静态同步方法，所以调用的时候需要获取同一个类上monitor（每个类只对应一个class对象），所以也只能顺序的执行。

1. 代码段4结果：

对于代码块的同步实质上需要获取Synchronized关键字后面括号中对象的monitor，由于这段代码中括号的内容都是this，而method1和method2又是通过同一的对象去调用的，所以进入同步块之前需要去竞争同一个对象上的锁，因此只能顺序执行同步块。

#### 总结

Synchronized是Java并发编程中最常用的用于保证线程安全的方式，其使用相对也比较简单。但是如果能够深入了解其原理，对监视器锁等底层知识有所了解，一方面可以帮助我们正确的使用Synchronized关键字，另一方面也能够帮助我们更好的理解并发编程机制，有助我们在不同的情况下选择更优的并发策略来完成任务。对平时遇到的各种并发问题，也能够从容的应对。

### Synchronized底层优化（轻量级锁、偏向锁）

#### 重量级锁

Java的并发编程中Synchronized是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。因此，这种依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”。JDK中对Synchronized做的种种优化，其核心都是为了减少这种重量级锁的使用。JDK1.6以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

#### 轻量级锁

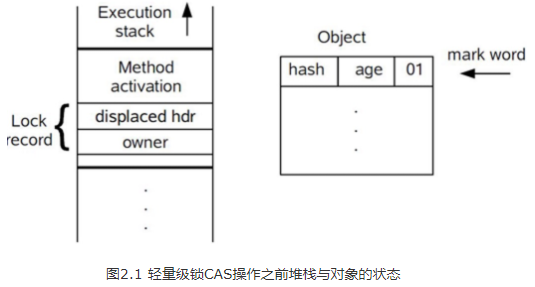
锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。JDK 1.6中默认是开启偏向锁和轻量级锁的，我们也可以通过-XX:-UseBiasedLocking来禁用偏向锁。

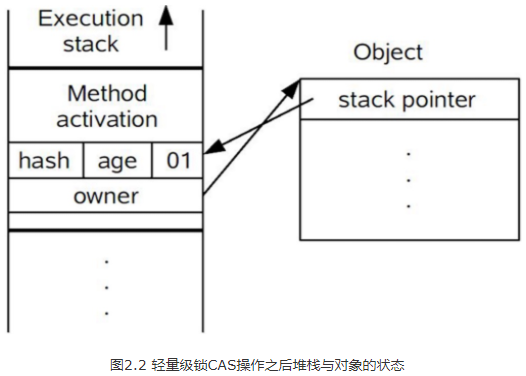
锁的状态保存在对象的头文件中，以32位的JDK为例：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 锁状态 | 25bit | | | 4bit | 1bit | 2bit |
| 23bit | | 2bit | 是否是偏向锁 | 锁标志位 |
| 轻量级锁 | 指向栈中锁记录的指针 | | | | | 00 |
| 重量级锁 | 指向互斥量（重量级锁）的指针 | | | | | 10 |
| GC标记 | 空 | | | | | 11 |
| 偏向锁 | 线程ID | Epoch | | 对象分代年龄 | 1 | 01 |
| 无锁 | 对象的hashCode | | | 对象分代年龄 | 0 | 01 |

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

1. 轻量级锁的加锁过程
2. 在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示
3. 拷贝对象头中的Mark Word复制到锁记录中。
4. 拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（3），否则执行步骤（4）。
5. 如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图2.2所示。
6. 如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。 而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。





1. 轻量级锁的解锁过程：
2. 通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word。
3. 如果替换成功，整个同步过程就完成了。
4. 如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

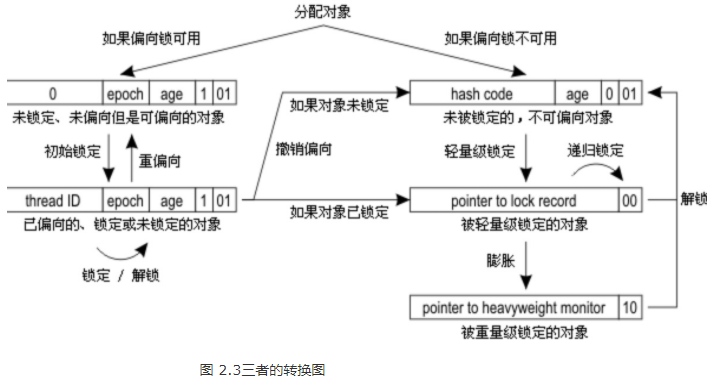
#### 偏向锁

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

1. 偏向锁获取过程：
2. 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——确认为可偏向状态。
3. 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤（5），否则进入步骤（3）。
4. 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。
5. 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。
6. 执行同步代码。
7. 偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

1. 重量级锁、轻量级锁和偏向锁之间转换



#### 其他优化

1. **适应性自旋（Adaptive Spinning）:** 从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。
2. **锁粗化（Lock Coarsening）**：锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。

举个例子：

**public** **class** StringBufferTest {

StringBuffer stringBuffer = **new** StringBuffer();

**public** **void** append(){

stringBuffer.append("a");

stringBuffer.append("b");

stringBuffer.append("c");

}

}

这里每次调用stringBuffer.append方法都需要加锁和解锁，如果虚拟机检测到有一系列连串的对同一个对象加锁和解锁操作，就会将其合并成一次范围更大的加锁和解锁操作，即在第一次append方法时进行加锁，最后一次append方法结束后进行解锁。

1. **锁消除（Lock Elimination）：**锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。

**public** **class** SynchronizedTest02 {

**public** **void** append(String str1, String str2){

StringBuffer sb = **new** StringBuffer();

sb.append(str1).append(str2);

}

**public** **static** **void** main(String[] args) {

SynchronizedTest02 test02 = **new** SynchronizedTest02();

//启动预热

**for** (**int** i = 0; i < 10000; i++) {

i++;;

}

**long** start = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 100000000; i++) {

test02.append("abc", "def");

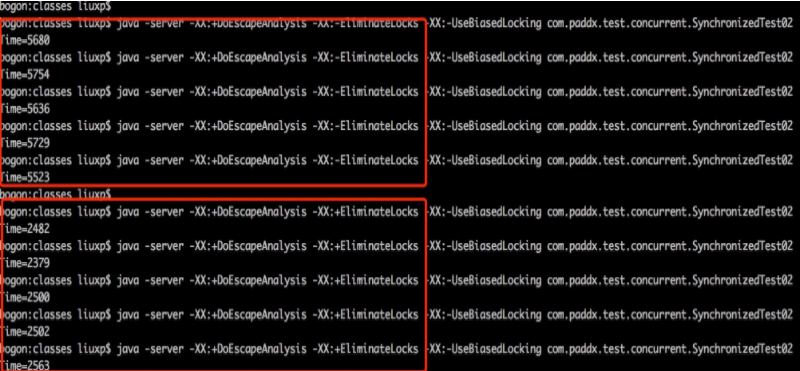
}

System.***out***.println("Time" + (System.*currentTimeMillis*() - start));

}

}

虽然StringBuffer的append是一个同步方法，但是这段程序中的StringBuffer属于一个局部变量，并且不会从该方法中逃逸出去，所以其实这过程是线程安全的，可以将锁消除。下面是我本地执行的结果：



为了尽量减少其他因素的影响，这里禁用了偏向锁（-XX:-UseBiasedLocking）。通过上面程序，可以看出消除锁以后性能还是有比较大提升的。

注：可能JDK各个版本之间执行的结果不尽相同，我这里采用的JDK版本为1.6。

#### 总结

本文重点介绍了JDk中采用轻量级锁和偏向锁等对Synchronized的优化，但是这两种锁也不是完全没缺点的，比如竞争比较激烈的时候，不但无法提升效率，反而会降低效率，因为多了一个锁升级的过程，这个时候就需要通过-XX:-UseBiasedLocking来禁用偏向锁。

下面是这几种锁的对比：



### 生产者消费者模式

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这种生产消费能力不均衡的问题，所以便有了生产者和消费者模式。

#### 什么是生产者消费者模式

生产者消费者模式并不是GOF提出的23种设计模式之一，23种设计模式都是建立在面向对象的基础之上的，但其实面向过程的编程中也有很多高效的编程模式，生产者消费者模式便是其中之一，它是我们编程过程中最常用的一种设计模式。

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大多数设计模式，都会找一个第三者出来进行解耦，如工厂模式的第三者是工厂类，模板模式的第三者是模板类。在学习一些设计模式的过程中，如果先找到这个模式的第三者，能帮助我们快速熟悉一个设计模式。

#### 实现生产者消费者

要实现生产者消费者模式，需要用到线程之间的通信的功能。

##### synchronized、wait和notify

该方式使用的是synchronized关键字及object对象中的wait方法和notify方式实现生产者和消费者模式。

生产者类：

**public** **class** Producer **extends** Thread {

**private** Resource resource;

**public** Producer(Resource resource){

**this**.resource = resource;

}

@Override

**public** **void** run() {

**while** (**true**) {

resource.produce();

}

}

}

消费者类：

**public** **class** Consumer **extends** Thread {

**private** Resource resource;

**public** Consumer(Resource resource){

**this**.resource = resource;

}

@Override

**public** **void** run() {

**while** (**true**){

resource.consume();

}

}

}

资源类：

**public** **class** Resource {

//定义存放的最小数量

**private** Integer minNum = 0;

//定义存放的最大数量

**private** Integer maxNum = 10;

/\*\*

\* 定义生产方法（线程安全）

\*/

**public** **synchronized** **void** produce(){

**if** (minNum < maxNum){

minNum++;

**this**.notifyAll();//唤醒所有等待的线程

System.***out***.println("生产线程开始生产");

}**else**{

**try** {

System.***err***.println("生产线程开始等待......");

**this**.wait();//设置当前生产线程为等待状态

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 定义消费方法(线程安全)

\*/

**public** **synchronized** **void** consume(){

**if**(minNum > 0){

minNum--;

**this**.notifyAll();//唤醒所有处于等待状态的线程

System.***out***.println("消费线程开始消费");

}**else**{

**try** {

System.***err***.println("消费线程开始等待......");

**this**.wait();//设置当前消费线程为等待状态

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

测试类：

**public** **class** Test {

**public** **static** **void** main(String[] args){

Resource resource = **new** Resource();

Producer producer1 = **new** Producer(resource);//生产者线程

Consumer consumer1 = **new** Consumer(resource);//消费者线程

Consumer consumer2 = **new** Consumer(resource);//消费者线程

producer1.start();

consumer1.start();

consumer2.start();

}

}

##### lock和condition的await、signalAll

该方式使用了java5.0中新增加的锁的机制来实现生产者和消费者模式。

公共资源类：

**class** Resource2 {

**private** **int** num = 0;// 当前资源数量

**private** **int** size = 10;// 资源池中允许存放的资源数目

**private** Lock lock;

**private** Condition producerCondition;

**private** Condition consumerCondition;

**public** Resource2(Lock lock, Condition producerCondition, Condition consumerCondition) {

**this**.lock = lock;

**this**.producerCondition = producerCondition;

**this**.consumerCondition = consumerCondition;

}

/\*\*

\* 向资源池中添加资源

\*/

**public** **void** add() {

lock.lock();

**try** {

**if** (num < size) {

num++;

System.***out***.println(Thread.*currentThread*().getName() + "生产一件资源,当前资源池有" + num + "个");

// 唤醒等待的消费者

consumerCondition.signalAll();

} **else** {

// 让生产者线程等待

**try** {

producerCondition.await();

System.***out***.println(Thread.*currentThread*().getName() + "线程进入等待");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

} **finally** {

lock.unlock();

}

}

/\*\*

\* 从资源池中取走资源

\*/

**public** **void** remove() {

lock.lock();

**try** {

**if** (num > 0) {

num--;

System.***out***.println("消费者" + Thread.*currentThread*().getName() + "消耗一件资源," + "当前资源池有" + num + "个");

producerCondition.signalAll();// 唤醒等待的生产者

} **else** {

**try** {

consumerCondition.await();

System.***out***.println(Thread.*currentThread*().getName() + "线程进入等待");

} **catch** (InterruptedException e) {

e.printStackTrace();

} // 让消费者等待

}

} **finally** {

lock.unlock();

}

}

}

生产者线程:

**class** ProducerThread2 **extends** Thread{

**private** Resource2 resource;

**public** ProducerThread2(Resource2 resource){

**this**.resource = resource;

setName("生产者");

}

**public** **void** run(){

**while**(**true**){

**try** {

Thread.*sleep*((**long**) (1000 \* Math.*random*()));

} **catch** (InterruptedException e) {

e.printStackTrace();

}

resource.add();

}

}

}

消费者线程:

**class** ConsumerThread2 **extends** Thread{

**private** Resource2 resource;

**public** ConsumerThread2(Resource2 resource){

**this**.resource = resource;

//setName("消费者");

}

**public** **void** run(){

**while**(**true**){

**try** {

Thread.*sleep*((**long**) (1000 \* Math.*random*()));

} **catch** (InterruptedException e) {

e.printStackTrace();

}

resource.remove();

}

}

}

测试类：

**public** **class** LockCondition {

**public** **static** **void** main(String[] args) {

Lock lock = **new** ReentrantLock();

Condition producerCondition = lock.newCondition();

Condition consumerCondition = lock.newCondition();

Resource2 resource = **new** Resource2(lock, producerCondition, consumerCondition);

// 生产者线程

ProducerThread2 producer1 = **new** ProducerThread2(resource);

// 消费者线程

ConsumerThread2 consumer1 = **new** ConsumerThread2(resource);

ConsumerThread2 consumer2 = **new** ConsumerThread2(resource);

ConsumerThread2 consumer3 = **new** ConsumerThread2(resource);

producer1.start();

consumer1.start();

consumer2.start();

consumer3.start();

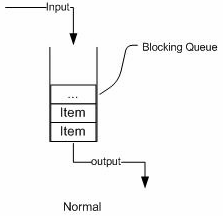
}

}

##### BlockingQueue（阻塞队列）

java5.0中新增加的Concurrent包中，BlockingQueue很好的解决了多线程中，如何高效安全“传输”数据的问题。通过这些高效并且线程安全的队列类，为我们快速搭建高质量的多线程程序带来极大的便利。

阻塞队列，顾名思义，首先它是一个队列，而一个队列在数据结构中所起的作用大致如下图所示：



从上图我们可以很清楚看到，通过一个共享的队列，可以使得数据由队列的一端输入，从另外一端输出；

常用的队列主要有以下两种：（当然通过不同的实现方式，还可以延伸出很多不同类型的队列，DelayQueue就是其中的一种）

先进先出（FIFO）：先插入的队列的元素也最先出队列，类似于排队的功能。从某种程度上来说这种队列也体现了一种公平性。

后进先出（LIFO）：后插入队列的元素最先出队列，这种队列优先处理最近发生的事件。

多线程环境中，通过队列可以很容易实现数据共享，比如经典的“生产者”和“消费者”模型中，通过队列可以很便利地实现两者之间的数据共享。假设我们有若干生产者线程，另外又有若干个消费者线程。如果生产者线程需要把准备好的数据共享给消费者线程，利用队列的方式来传递数据，就可以很方便地解决他们之间的数据共享问题。但如果生产者和消费者在某个时间段内，万一发生数据处理速度不匹配的情况呢？理想情况下，如果生产者产出数据的速度大于消费者消费的速度，并且当生产出来的数据累积到一定程度的时候，那么生产者必须暂停等待一下（阻塞生产者线程），以便等待消费者线程把累积的数据处理完毕，反之亦然。然而，在concurrent包发布以前，在多线程环境下，我们每个程序员都必须去自己控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。好在此时，强大的concurrent包横空出世了，而他也给我们带来了强大的BlockingQueue。（在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒）

**常见BlockingQueue：**

**ArrayBlockingQueue**

基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护了一个定长数组，以便缓存队列中的数据对象，这是一个常用的阻塞队列，除了一个定长数组外，ArrayBlockingQueue内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置。

ArrayBlockingQueue在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于LinkedBlockingQueue；按照实现原理来分析，ArrayBlockingQueue完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。Doug Lea之所以没这样去做，也许是因为ArrayBlockingQueue的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。 ArrayBlockingQueue和LinkedBlockingQueue间还有一个明显的不同之处在于，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的Node对象。这在长时间内需要高效并发地处理大批量数据的系统中，其对于GC的影响还是存在一定的区别。而在创建ArrayBlockingQueue时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

**LinkedBlockingQueue**

基于链表的阻塞队列，同ArrayListBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

作为开发者，我们需要注意的是，如果构造一个LinkedBlockingQueue对象，而没有指定其容量大小，LinkedBlockingQueue会默认一个类似无限大小的容量（Integer.MAX\_VALUE），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。

ArrayBlockingQueue和LinkedBlockingQueue是两个最普通也是最常用的阻塞队列，一般情况下，在处理多线程间的生产者消费者问题，使用这两个类足以。

生产者线程：

**public** **class** Producer **implements** Runnable {

**private** **volatile** **boolean** isRunning = **true**;

**private** BlockingQueue queue;

**private** **static** AtomicInteger *count* = **new** AtomicInteger();

**private** **static** **final** **int** ***DEFAULT\_RANGE\_FOR\_SLEEP*** = 1000;

**public** Producer(BlockingQueue queue) {

**this**.queue = queue;

}

**public** **void** run() {

String data = **null**;

Random r = **new** Random();

System.***out***.println("启动生产者线程！");

**try** {

**while** (isRunning) {

System.***out***.println("正在生产数据...");

Thread.*sleep*(r.nextInt(***DEFAULT\_RANGE\_FOR\_SLEEP***));

data = "data:" + *count*.incrementAndGet();

System.***out***.println("将数据：" + data + "放入队列...");

**if** (!queue.offer(data, 2, TimeUnit.SECONDS)) {

System.***out***.println("放入数据失败：" + data);

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

Thread.*currentThread*().interrupt();

} **finally** {

System.***out***.println("退出生产者线程！");

}

}

**public** **void** stop() {

isRunning = **false**;

}

}

消费者线程：

**public** **class** Consumer **implements** Runnable {

**private** BlockingQueue<String> queue;

**private** **static** **final** **int** ***DEFAULT\_RANGE\_FOR\_SLEEP*** = 1000;

**public** Consumer(BlockingQueue<String> queue) {

**this**.queue = queue;

}

**public** **void** run() {

System.***out***.println("启动消费者线程！");

Random r = **new** Random();

**boolean** isRunning = **true**;

**try** {

**while** (isRunning) {

System.***out***.println("正从队列获取数据...");

String data = queue.poll(2, TimeUnit.SECONDS);

**if** (**null** != data) {

System.***out***.println("拿到数据：" + data);

System.***out***.println("正在消费数据：" + data);

Thread.*sleep*(r.nextInt(***DEFAULT\_RANGE\_FOR\_SLEEP***));

} **else** {

// 超过2s还没数据，认为所有生产线程都已经退出，自动退出消费线程。

isRunning = **false**;

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

Thread.*currentThread*().interrupt();

} **finally** {

System.***out***.println("退出消费者线程！");

}

}

}

测试类：

**public** **class** BlockingQueueTest {

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

// 声明一个容量为10的缓存队列

BlockingQueue<String> queue = **new** LinkedBlockingQueue<String>(10);

Producer producer1 = **new** Producer(queue);

Producer producer2 = **new** Producer(queue);

Producer producer3 = **new** Producer(queue);

Consumer consumer = **new** Consumer(queue);

// 借助Executors

ExecutorService service = Executors.newCachedThreadPool();

// 启动线程

service.execute(producer1);

service.execute(producer2);

service.execute(producer3);

service.execute(consumer);

// 执行10s

Thread.*sleep*(10 \* 1000);

producer1.stop();

producer2.stop();

producer3.stop();

Thread.*sleep*(2000);

// 退出Executor

service.shutdown();

}

}

**DelayQueue**

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

使用场景：

DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个超时未响应的连接队列。

**PriorityBlockingQueue**

基于优先级的阻塞队列（优先级的判断通过构造函数传入的Compator对象来决定），但需要注意的是PriorityBlockingQueue并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。在实现PriorityBlockingQueue时，内部控制线程同步的锁采用的是公平锁。

**SynchronousQueue**

一种无缓冲的等待队列，类似于无中介的直接交易，有点像原始社会中的生产者和消费者，生产者拿着产品去集市销售给产品的最终消费者，而消费者必须亲自去集市找到所要商品的直接生产者，如果一方没有找到合适的目标，那么对不起，大家都在集市等待。相对于有缓冲的BlockingQueue来说，少了一个中间经销商的环节（缓冲区），如果有经销商，生产者直接把产品批发给经销商，而无需在意经销商最终会将这些产品卖给那些消费者，由于经销商可以库存一部分商品，因此相对于直接交易模式，总体来说采用中间经销商的模式会吞吐量高一些（可以批量买卖）；但另一方面，又因为经销商的引入，使得产品从生产者到消费者中间增加了额外的交易环节，单个产品的及时响应性能可能会降低。

声明一个SynchronousQueue有两种不同的方式，它们之间有着不太一样的行为。公平模式和非公平模式的区别:

如果采用公平模式：SynchronousQueue会采用公平锁，并配合一个FIFO队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；

但如果是非公平模式（SynchronousQueue默认）：SynchronousQueue采用非公平锁，同时配合一个LIFO队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有某些生产者或者是消费者的数据永远都得不到处理。

## JDBC操作数据库

JDBC是连接数据库和Java程序的桥梁，通过JDBC API可以方便地实现对各种主流数据库的操作。

### JDBC技术

JDBC制定了统一的访问各类关系数据库的标准接口，为各个数据库厂商提供了标准接口的实现。JDBC并不能直接访问数据库，需要依赖于数据库厂商提供的JDBC驱动程序。

JDBC的优点

1. JDBC与ODBC十分相似，便于软件开发人员理解
2. JDBC使软件开发人员从复杂的驱动程序编写工作中解脱出来，可以完全专注于业务逻辑的开发
3. JDBC支持多种关系型数据库，大大增加了软件的可移植性
4. JDBC API是面向对象的，软件开发人员可以将常用的方法进行二次封装，从而提高代码的重用性。

### JDBC的常用类和接口

#### DriverManager类

java.sql.DriverManager类用来管理数据库中的所有驱动程序，是JDBC的管理层，作用于用户和驱动程序之间，跟踪可用的驱动程序，并在数据库的驱动程序之间建立连接。此外，DriverManager类也处理诸如驱动程序登录时间限制及登录和跟踪信息的显示等事务。DriverManager类中的方法都是静态方法，所以在程序中无须对它进行实例化，直接通过类名就可以调用。

DriverManager类的常用方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| public static Connection getConnection(String url) | 该方法获取一个Connection接口的实例，该对象存储的是数据库的连接，参数url为连接数据库的URL：jdbc:mysql://localhost:3306/myDB?user=soft&password=1234&useUnicode=true&characterEncoding=UTF-8 |
| public static Connection getConnection(String url, Properties info) | 该方法获取一个Connection接口的实例，该对象存储的是数据库的连接，参数url为连接数据库的URL：jdbc:mysql://localhost:3306/myDB? useUnicode=true&characterEncoding=UTF-8  参数info为Properties对象，其中保存着xxx.properties配置文件中的内容。（通过该类的setProperty(String key, String value)方法） |
| public static Connection getConnection(String url, String user, String password) | 该方法获取一个Connection接口的实例，该对象存储的是数据库的连接，参数url为连接数据库的URL：jdbc:mysql://localhost:3306/myDB? useUnicode=true&characterEncoding=UTF-8  user:数据库用户名。password：数据库密码 |
| setLoginTimeout() | 获取驱动程序试图登录到某一数据库时可以等待的最长时间，以秒为单位 |
| println(String message) | 将一条消息打印到当前JDBC日志流中 |

#### Connection接口

Connection接口代表与特定的数据库的连接。要对数据表中的数据进行操作，首先要获取数据库连接。Connection实例就是像在应用程序与数据库之间开通了一条渠道。

可通过DriverManager类的getConnection()方法获取Connection实例

Connection接口的常用方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| createStatement() | 创建一个 Statement 类型的对象来将 SQL 语句发送到数据库。 |
| createStatement(int resultSetType,int resultSetConcurrency) | 创建一个Statement类型的对象，该对象将生成具有给定类型、并发性和可保存性的ResultSet对象 |
| prepareStatement(String sql) | 创建预处理对象PreparedStatement，其中sql为需要语出的sql语句 |
| isReadOnly() | 查看当前Connection对象的读取模式是否是只读形式 |
| setReadOnly() | 设置当前Connection对象的读写模式，默认是非只读模式 |
| commit() | 使所有上一次提交、回滚后进行的更该为持久更改，并释放此Connection对象当前持有的所有数据库锁 |
| roolback() | 取消在当前事务中进行的所有更改，并释放此Connection对象当前持有的所有数据库锁 |
| close() | 立即释放此Connection对象的数据库和JDBC资源，而不是等待他们被自动释放 |

#### Statement接口

Statement接口用于创建向数据库中传递SQL语句的对象，该接口提供了一些方法可以实现对数据库的常用操作。

Statement接口的常用方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| execute(String sql) | 用于执行返回多个结果集、多个更新计数或二者组合的语句。因为多数程序员不会需要该高级功能。execute方法应该仅在语句能返回多个ResultSet对象、多个更新计数或ResultSet对象与更新计数的组合时使用。 |
| executeQuery(String sql) | 执行给定的查询SQL语句，该语句返回单个ResultSet对象 |
| clearBatch() | 清空此Statement对象的当前SQL命令列表 |
| executeBatch() | 将一批命令提交给数据库来执行，如果全部命令执行成功，则返回更新计数组成的数组。数组元素的排序与SQL语句的添加顺序对应 |
| executeUpdate() | 执行给定SQL语句，该语句可以为INSERT、UPDATE或DELETE语句 |
| addBatch(String sql) | 将给定的SQL命令添加到此Statement对象的当前命令列表中。如果驱动程序不支持批量处理将抛出异常 |
| close() | 释放Statement实例占用的数据库和JDBC资源 |

#### PreparedStatement接口

PreparedStatement接口继承Statement，用于执行动态的SQL语句，通过PreparedStatement实例执行的SQL语句，将被预编译并保存到PreparedStatement实例中，从而可以反复地执行该SQL语句。

PreparedStatement接口的常用方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| execute() | 用于执行返回多个结果集、多个更新计数或二者组合的语句。因为多数程序员不会需要该高级功能。execute方法应该仅在语句能返回多个ResultSet对象、多个更新计数或ResultSet对象与更新计数的组合时使用。 |
| executeQuery() | 在此PreparedStatement对象中执行查询的SQL查询语句，返回结果为查询结果集ResultSet对象 |
| executeUpdate() | 在此PreparedStatement对象中执行的SQL语句，该SQL语句必须是一个INSERT、UPDATE、DELETE语句，或者是没有返回值的DDL语句 |
| setByte(int pIndex,byte bt) | 将参数pIndex位置上设置为给定的byte型参数bt |
| setDouble(int pIndex,double dou) | 将参数pIndex位置上设置为给定的double型参数值dou |
| setInt(int pIndex,int x) | 将参数pIndex位置上设置为给定的int型参数值x |
| setObject(int pIndex,Object o) | 将参数pIndex位置上设置为给定的Object型参数值 |
| setString(int pIndex,String str) | 将参数pIndex位置上设置为给定的String型参数值 |

#### ResultSet接口

ResultSet接口类似于一个临时表，用于暂时存放数据库查询操作所获得的结果集。可以通过定义的方法来操作结果集中的数据。

ResultSet接口常用的方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| getInt() | 以int形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回值是0 |
| getFloat() | 以float形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回值是0 |
| getDate() | 以Date形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回值是null |
| getBoolean() | 以boolean形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回null |
| getString() | 以String形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回null |
| getObject() | 以Object形式获取ResultSet对象的当前行的指定列值。如列值是null，则返回null |
| getMetaData() | 获取此ResultSet对象的列的编号、类型和属性。该方法返回一个ResultSetMetaData接口，该接口可用于获取关于 ResultSet 对象中列的类型和属性信息的对象。比如：列数、列名称等信息 |
| next() | 将指针向下移一行 |
| updateInt() | 用int值更新指定列 |
| updateFloat() | 用float值更新指定列 |
| updateLong() | 用指定的long值更新指定列 |
| updateString() | 用指定的String值更新指定列 |
| updateObject() | 用Object值更新指定列 |
| updateNull() | 将指定的列值修改为null |
| updateDate() | 用指定的Date值更新指定列 |
| updateDouble() | 用指定的double值更新指定列 |

#### ResultSetMetaData接口

该接口通过ResultSet接口的getMetaData()方法获取。该接口表示可用于获取关于 ResultSet 对象中列的类型和属性信息的对象。表示ResultSet结果集的元数据。其中给定的方法用于获取ResultSet结果集中的数据

ResultSetMetaData接口常用的方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| getCatalogName(int column) | 获取指定列的表目录名称。其中第一列为1，第二列为2。 |
| getColumnClassName(int column) | 如果调用方法 ResultSet.getObject 从列中获取值，则返回构造其实例的 Java 类的完全限定名称。ResultSet.getObject 可能返回此方法所返回的类的子类。 |
| getColumnCount() | 返回此 ResultSet 对象中的列数。 |
| getColumnName(int column) | 获取指定列的名称。其中第一列为1，第二列为2。 |

### 通过JDBC操作数据库

如果需要访问数据库，首先要加载数据库驱动，数据库驱动只需要在第一次访问数据库时加载一次，然后在每次访问数据库时创建一个Connection实例，获取数据库连接， 就可以执行操作数据库的SQL语句。最后在完成数据库操作时，释放与数据库的连接。

#### 加载数据库驱动

将下载的数据库驱动文件添加到项目中后，首先需要加载数据库驱动程序，才能进行数据库操作。Java加载数据库驱动的方法是调用Class类的静态方法forName()

语法格式：

Class.forName(String driverManager)

driverManager：所需类的完全限定名。

返回值：返回与带有给定字符串名的类或接口相关联的Class对象。

forName()方法参数指定要加载的数据库驱动，加载成功，会将加载的驱动注册给DriverManager。如果加载失败，则会抛出ClassNotFoundException异常

实例：

**public** **static** **void** main(String[] args) {

**try** { //加载MySQL数据库驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

}

#### 建立连接

加载完数据库驱动即可建立数据库的连接，要连接数据库可以使用DriverManager类的静态方法getConnection()来实现，并分别将数据库的URL、数据库用户名和密码作为该方法的参数，即可建立到指定数据库的连接。该方法返回一个Connection类型的对象存储这数据库的连接。

实例：

**public** **static** **void** main(String[] args) {

**try** { //加载MySQL数据库驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

//建立数据连接

String url = "jdbc:mysql://localhost:3306/javase?useUnicode=true&charterEncoding=utf-8";

String username = "root";

String password = "";

Connection conn = DriverManager.*getConnection*(url, username, password);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

由于在一个程序中经常需要对数据库进行操作，如果每次操作数据库都要建立数据库的连接，这样不但会出现代码重复，而且也极大地影响了编程效率，为此可以为数据库连接单独创建一个类，在该类中定义一个进行数据库连接的静态方法，当需要使用数据库连接时，可以通过类名调用该方法获得数据库的连接。

#### 向数据库添加数据

在Java程序中，一旦建立了数据库连接，就可以使用Connection接口的createStatement()方法来获得Statement对象，也可调用prepareStatement()方法获得PreparedStatement对象，通过executeUpdate()方法来执行SQL语句，就可以向数据库添加数据了。

**实例1：Statement接口实现**

//………………………………..

//生成一条SQL语句

String sql = "insert into t\_user(username,password,sex,age)values('张三','111','男','22')";

Statement stmt = conn.createStatement(); //创建一个Statement对象

stmt.executeUpdate(sql); //执行SQL语句

conn.close(); //关闭资源

//………………………………..

**实例2：PreparedStatement接口实现**

//………………………………..

//生成一条SQL语句

String sql = "insert into t\_user(username,password,sex,age)values(?,?,?,?)";

PreparedStatement ps = conn.prepareStatement(sql); //创建一个PreparedStatement对象

ps.setString(1, "李四");

ps.setString(2, "aaa");

ps.setString(3, "男");

ps.setInt(4, 23);

ps.executeUpdate(); //执行SQL语句

conn.close(); //关闭资源

//………………………………..

#### 获取数据库数据

Statement接口和PreparedStatement接口的executeUpdate()或executeQuery()方法可以执行SQL语句，executeUpdate()方法用于执行数据的插入、修改或删除操作，返回影响数据库记录的条数。executeQuery()方法用于执行SELECT查询语句，将返货一个ResultSet型的结果集。通过遍历结果集的内容，才可获取SQL语句执行的查询结果。

ResultSet对象具有指向当前数据行的光标。最初，光标被置于第一行之前，可以通过该对象的next()方法将光标移动到下一行；如果ResultSet对象没有下一行，next()方法返回false，所以可以在while循环使用next()方法迭代结果集。

//………………………………..

//生成一条SQL语句

String sql = "select \* from t\_user";

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery(sql);

**int** id,age;

String username,pwd,sex;

**while**(rs.next()){

id = rs.getInt("id"); //根据数据库的字段名称来访问

username = rs.getString(2); //根据列数来访问

pwd = rs.getString("password");

sex = rs.getString(4);

age = rs.getInt("age");

System.***out***.println(id + username + pwd + sex + age);

}

conn.close(); //关闭资源

//………………………………..

说明：ResultSet对象的getXXX()方法可获取查询结果集中数据，由于ResultSet中保存的数据是表的形式，因此可通过使用getXXX()方法指定列的序号与列的名称来获取数据。如上述实例中的代码“rs.getString(“username”)”，username表示表的列名。如果此时username为查询结果集中的第2列，也可通过代码“rs.getString(2)”得到查询结果。

**PreparedStatement接口实现**

//………………………………..

//生成一条SQL语句

String sql = "select \* from t\_user where username=? and password=?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, "小明");

ps.setString(2, "111");

ResultSet rs = ps.executeQuery();

**int** id,age;

String username,pwd,sex;

**while**(rs.next()){

id = rs.getInt("id"); //根据数据库的字段名称来访问

username = rs.getString(2); //根据列数来访问

pwd = rs.getString("password");

sex = rs.getString(4);

age = rs.getInt("age");

System.***out***.println(id + username + pwd + sex + age);

}

conn.close(); //关闭资源

//………………………………..

#### 更改数据库中的数据

使用Statement接口中的executeUpdate()方法可以修改数据库表中的数据，也可以使用PreparedStatement接口中的executeUpdate()方法对数据库中的表进行修改操作。

实例：

//………………………………..

//生成一条SQL语句

String sql = "update t\_user set age=20 where id=1";

Statement stmt = conn.createStatement();

stmt.executeUpdate(sql);

conn.close(); //关闭资源

//………………………………..

#### 删除数据库中的数据

使用Statement接口和PreparedStatement接口中的executeUpdate()方法同样可以删除数据库中的数据

//………………………………..

//生成一条SQL语句

String sql = "delete from t\_user where id=?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, 1);

ps.executeUpdate();

conn.close(); //关闭资源

//………………………………..

#### Statement接口和PrepareStatement接口比较

1. 执行SQL语句的格式不同，PreparedStatement更具可读性。

**Statement：**

//生成一条SQL语句

String sql = " select \* from t\_user where username='" + username + "' and password='" + pwd + "' ";

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery(sql);

**PreparedStatement：**

String sql = "select \* from t\_user where username=? and password=?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, "小明");

ps.setString(2, "111");

ResultSet rs = ps.executeQuery();

1. PreparedStatement可以写动态参数化的查询

用PreparedStatement你可以写带参数的sql查询语句，通过使用相同的sql语句和不同的参数值来做查询比创建一个不同的查询语句要好，下面是一个参数化查询：

SELECT interest\_rate FROM loan WHERE loan\_type=?

现在你可以使用任何一种loan类型如：”personal loan”,”home loan” 或者”gold loan”来查询，这个例子叫做参数化查询，因为它可以用不同的参数调用它，这里的”?”就是参数的占位符。

1. PreparedStatement比 Statement 更快

使用 PreparedStatement 最重要的一点好处是它拥有更佳的性能优势，SQL语句会预编译在数据库系统中。执行计划同样会被缓存起来，它允许数据库做参数化查询。使用预处理语句比普通的查询更快，因为它做的工作更少（数据库对SQL语句的分析，编译，优化已经在第一次查询前完成了）。为了减少数据库的负载，生产环境中德JDBC代码你应该总是使用PreparedStatement 。值得注意的一点是：为了获得性能上的优势，应该使用参数化sql查询而不是字符串追加的方式。

下面两个SELECT 查询，第一个SELECT查询就没有任何性能优势。

SQL Query 1:字符串追加形式的PreparedStatement

String loanTement prestmt = conn.prepareStatement("select banks from loan where loan\_type=" + loanType);

SQL Query 2：使用参数化查询的PreparedStatement

PreparedStatement prestmt = conn.prepareStatement("select banks from loan where loan\_type=?");

prestmt.setString(1,loanType);

第二个查询就是正确使用PreparedStatement的查询，它比SQL1能获得更好的性能。

1. PreparedStatement可以防止SQL注入式攻击

如果你是做Java web应用开发的，那么必须熟悉那声名狼藉的SQL注入式攻击。去年Sony就遭受了SQL注入攻击，被盗用了一些Sony play station（PS机）用户的数据。在SQL注入攻击里，恶意用户通过SQL元数据绑定输入，比如：某个网站的登录验证SQL查询代码为：

strSQL = "SELECT \* FROM users WHERE name = '" + userName + "' and pw = '"+ passWord +"';"

恶意填入：

userName = "1' OR '1'='1";

passWord = "1' OR '1'='1";

那么最终SQL语句变成了：

strSQL = "SELECT \* FROM users WHERE name = '1' OR '1'='1' and pw = '1' OR '1'='1';"

因为WHERE条件恒为真，这就相当于执行：

strSQL = "SELECT \* FROM users;"

因此可以达到无账号密码亦可登录网站。如果恶意用户要是更坏一点，用户填入：

strSQL = "SELECT \* FROM users;"

SQL语句变成了：

strSQL = "SELECT \* FROM users WHERE name = 'any\_value' and pw = ''; DROP TABLE users"

这样一来，虽然没有登录，但是数据表都被删除了。

然而使用PreparedStatement的参数化的查询可以阻止大部分的SQL注入。在使用参数化查询的情况下，数据库系统（eg:MySQL）不会将参数的内容视为SQL指令的一部分来处理，而是在数据库完成SQL指令的编译后，才套用参数运行，因此就算参数中含有破坏性的指令，也不会被数据库所运行。

补充：避免SQL注入的第二种方式：

在组合SQL字符串的时候，先对所传入的参数做字符取代（将单引号字符取代为连续2个单引号字符，因为连续2个单引号字符在SQL数据库中会视为字符中的一个单引号字符，譬如：

strSQL = "SELECT \* FROM users WHERE name = '" + userName + "';"

传入字符串：

userName = " 1' OR 1=1 "

把userName做字符替换后变成：

userName = " 1'' OR 1=1"

最后生成的SQL查询语句为：

strSQL = "SELECT \* FROM users WHERE name = '1'' OR 1=1'

这样数据库就会去系统查找name为“1′ ‘ OR 1=1”的记录，而避免了SQL注入。

#### 事务处理

1. 什么是事务

通常的观念认为，事务仅与数据库相关。

事务必须服从ISO/IEC所制定的ACID原则。ACID是原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）的缩写。

原子性：表示事务执行过程中的任何失败都将导致事务所做的任何修改失效。

一致性：表示当事务执行失败时，所有被该事务影响的数据都应该恢复到事务执行前的状态。

隔离性：表示在事务执行过程中对数据的修改，在事务提交之前对其他事务不可见。

持久性：表示已提交的数据在事务执行失败时，数据的状态都应该正确。

通俗的理解，事务是一组原子操作单元，从数据库角度说，就是一组SQL指令，要么全部执行成功，若因为某个原因其中一条指令执行有错误，则撤销先前执行过的所有指令。更简答的说就是：要么全部执行成功，要么撤销不执行。

既然事务的概念从数据库而来，那Java事务是什么？之间有什么联系？

实际上，一个Java应用系统，如果要操作数据库，则通过JDBC来实现的。增加、修改、删除都是通过相应方法间接来实现的，事务的控制也相应转移到Java程序代码中。因此，数据库操作的事务习惯上就称为Java事务。

1. 为什么需要事务

事务是为解决数据安全操作提出的，事务控制实际上就是控制数据的安全访问。举一个简单例子：比如银行转帐业务，账户A要将自己账户上的1000元 转到B账户下面，A账户余额首先要减去1000元，然后B账户要增加1000元。假如在中间网络出现了问题，A账户减去1000元已经结束，B因为网络中断而操作失败，那么整个业务失败，必须做出控制，要求A账户转帐业务撤销。这才能保证业务的正确性，完成这个操走就需要事务，将A账户资金减少和B账户资 金增加方到一个事务里面，要么全部执行成功，要么操作全部撤销，这样就保持了数据的安全性。

1. java事务的类型

Java事务的类型有三种：JDBC事务、JTA(Java Transaction API)事务、容器事务。

* JDBC事务

JDBC 事务是用 Connection对象控制的。JDBC Connection接口( java.sql.Connection )提供了两种事务模式：自动提交和手工提交。java.sql.Connection 提供了以下控制事务的方法：

**public void setAutoCommit(boolean)**

当该方法的参数为false时，则JDBC事务为手动提交，参数为true时，JDBC事务为自动提交。即每执行一个SQL语句，如果成功，则作为一个事务提交。默认为自动提交。

**public boolean getAutoCommit()**

获取此 Connection 对象的自动提交模式的当前状态

**public void commit()**

该方法用于JDBC事务设置为手动提交时提交JDBC事务。

**public void rollback()**

该方法用于JDBC事务设置为手动提交时提交事务出错，进行手动回滚事务。

使用 JDBC 事务界定时，您可以将多个 SQL 语句结合到一个事务中。JDBC 事务的一个缺点是事务的范围局限于一个数据库连接。一个 JDBC 事务不能跨越多个数据库

**try** { //加载MySQL数据库驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

//建立数据连接

String url = "jdbc:mysql://localhost:3306/javase?useUnicode=true&charterEncoding=utf-8";

String user = "root";

String password = "";

Connection conn = DriverManager.*getConnection*(url, user, password);

conn.setAutoCommit(**false**); //将事务设置为手动

//生成一条SQL语句

String sql = "insert into t\_user(name,pwd)values(?,?)";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, "xxx");

ps.setString(2, "xxx");

ps.executeUpdate();

conn.commit(); //当代码执行成功后提交事务

conn.close(); //关闭资源

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

* JTA(Java Transaction API)事务

JTA是一种高层的，与实现无关的，与协议无关的API，应用程序和应用服务器可以使用JTA来访问事务。

JTA允许应用程序执行分布式事务处理在两个或多个网络计算机资源上访问并且更新数据，这些数据可以分布在多个数据库上。JDBC驱动程序的JTA支持极大地增强了数据访问能力。

如果计划用 JTA 界定事务，那么就需要有一个实现javax.sql.XADataSource、javax.sql.XAConnectio和javax.sql.XAResource 接口的 JDBC 驱动程序。一个实现了这些接口的驱动程序将可以参与JTA事务。

一个 XADataSource对象就是一个XAConnection 对象的工厂。XAConnection s 是参与 JTA 事务的 JDBC 连接。 您将需要用应用服务器的管理工具设置 XADataSource 。从应用服务器和 JDBC 驱动程序的文档中可以了解到相关的指导。

J2EE 应用程序用 JNDI 查询数据源。一旦应用程序找到了数据源对象，它就调用 javax.sql.DataSource.getConnection() 以获得到数据库的连接。

XA 连接与非 XA 连接不同。一定要记住 XA 连接参与了 JTA 事务。这意味着 XA 连接不支持 JDBC 的自动提交功能。同时，应用程序一定不要对 XA 连接调用 java.sql.Connection.commit() 或者 java.sql.Connection.rollback() 。相反，应用程序应该使用 UserTransaction.begin()、UserTransaction.commit()和serTransaction.rollback() 。

* 容器事务

容器事务主要是J2EE应用服务器提供的，容器事务大多是基于JTA完成，这是一个基于JNDI的，相当复杂的API实现。相对编码实现JTA事 务管理，我们可以通过EJB容器提供的容器事务管理机制（CMT）完成同一个功能，这项功能由J2EE应用服务器提供。这使得我们可以简单的指定将哪个方法加入事务，一旦指定，容器将负责事务管理任务。这是我们土建的解决方式，因为通过这种方式我们可以将事务代码排除在逻辑编码之外，同时将所有困难交给J2EE容器去解决。使用EJB CMT的另外一个好处就是程序员无需关心JTA API的编码，不过，理论上我们必须使用EJB。

1. 三种事务的差异

* JDBC事务控制的局限性在一个数据库连接内，但是其使用简单。
* JTA事务的功能强大，事务可以跨越多个数据库或多个DAO，使用也比较复杂。
* 容器事务，主要指的是J2EE应用服务器提供的事务管理，局限于EJB应用使用。

1. 事务隔离

为了应对多线程并发读取数据时出现的问题，事务有了“隔离级别”特性，多线程并发读取数据一般会引发如下三个问题：脏读、不可重复读、幻读。

**脏读**：一个事务正在访问数据，并对数据进行了修改，而这种修改还没有提交到数据库中，与此同时，另一个事务读取了这些数据，因为这些数据还没有提交，所以另一个事务读取的数据是脏数据，依据脏数据进行的操作可能是不正确的。如果前一个事务发生了回滚，那么后一个事务读取的将是无效的数据。

**不可重复读**：一个事务读取了一行数据，在这个事务结束前，另一个书屋访问了同一行数据，并对数据进行了修改，当第一个事务再次读取这行数据时，得到了一个不同的数据。这样，在同一个事务内两次读取的数据不同。

**幻读**：一个事务读取了满足条件的所有行后，第二个事务插入了一行数据，第一个事务再次读取同样条件的数据时，却发现多出了一行数据，这好像出现了幻觉一样。

为了处理上面的读数据问题，java事务提供了4种隔离级别：

* Read Uncommitted

最低等级的事务隔离，它仅仅保证在读取过正中不会读取到非法数据。在这种隔离等级下，上述3种不确定的情况均有可能发生。

* Read Committed

此级别的事务隔离保证了一个事务不会读取到另一个并行事务已修改但未提交的数据。也就是说，这个等级的事务级别避免了“脏读”。

* Repeatable Read

此级别的事务隔离避免了“脏读”和“不可重复读”，这也意味着，一个事务在执行过程中可以看到其他事务已经提交的新插入的数据，但是不能看到其他事务对已有记录的更新。

* Serializable

最高等级的事务隔离，也提供了最严格的隔离机制。上述3种情况都将被避免。在此级别下，一个事务在执行过程中完全看不到其他事务对数据库所做的更新。当两个事务访问相同的数据时，如果第一个事务已经在访问该数据，那么第二个事务只能停下来等待，必须等到第一个事务结束后才能恢复运行，因此这两个事务实际上以串行化方式在运行。

Connection接口的setTransactionIsolation()方法，用于设置事务的隔离级别，同时Connection接口还定义了如下5个常量，用作setTransactionIsolation()方法的参数。

* TRANSACTION\_NONE

不支持事务

* TRANSACTION\_READ\_UNCOMMITTED

指定可以发生脏读、不可重复读和幻读、这个事务隔离级别表示在一个事务对一个数据行的所有修改操作提交之前，允许另一个事务读取这一行。如果第一个事务的所有更改操作被回滚，则第二个事务将获取到一个无效的数据行。

* TRANSACTION\_READ\_COMMMITTED

指定禁止脏读，但可以发生不可重复读和幻读。也就是说，这个事务隔离级别只禁止事务在数据行的修改操作被提交之前读取这个数据

* TRANSACTION\_REPEATABLE\_READ

指定禁止脏读和不可重复读，但可以发生幻读。这个事务隔离级别禁止事务读取一个还没有提交修改操作的数据行，并且它还禁止不可重复度的情况：一个事务读取了一个数据行，而另一个事务修改了这一行，然后第一个事务重新读取这个数据行，并在第二次读取时得到了不同的数据值。

* TRANSACTION\_SERIALIZABLE

指定禁止脏读、不可重复读和幻读。这个事务隔离级别包括了TRANSACTION\_REPEATABLE\_READ隔离级别禁止的事项，同时还禁止出现幻读的情况：当一个事务读取满足WHERE条件的所有数据行后，另一个事务插入了一个满足WHERE条件的数据行，然后第一个事务再次读取满足相同条件的数据行时，将会得到一个新增的数据行。

Connection conn = DriverManager.*getConnection*(url, user, password);

conn.setAutoCommit(**false**); //将事务设置为手动

conn.setTransactionIsolation(Connection.***TRANSACTION\_REPEATABLE\_READ***); //设置事务级别

Statement stmt = conn.createStatement();

#### 可滚动和可更新的结果集

从JDBC2.1开始，对结果集提供了更多的增强特性，支持可滚动和可更新的结果集，支持批量更新。

1. **可滚动的结果集**

我们通过Statement对象所创建的结果集只能向前滚动，也就是说只能调用next()方法向前得到数据行，当到达最后一条记录时，next()方法将返回false，我们无法想后读取数据行。

如果想获得一个可滚动的结果集，需要在创建Statement对象是，调用Connection对象的另一个重载方法createStatement()方法：

Statement createStatement(int resultSetType, int resultSetConcurrency)

创建一个 Statement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。

*resultSetType*用于指定结果集的类型，可以有如下3个取值：

* ResultSet.TYPE\_FORWARD\_ONLY

结果集只能向前移动。这是调用不带参数的createStatement()方法的默认类型。

* ResultSet.TYPE\_SCROLL\_INSENSITIVE

结果集可以滚动，但是对数据库的变化不敏感

* ResultSet.TYPE\_SCROLL\_SENSITIVE

结果集可以滚动，并且对数据库的变化很敏感。例如，在程序中通过查询返回了10行数据，如果另一个程序删除了其中的2行，那么这个查询结果集中就只有8行数据了。

*resultSetConcurrency*用于指定并发性类型，可以有如下的两个取值：

* ResultSet.CONCUR\_READ\_ONLY

结果集不能用于更新数据库。这是调用不带参数的createStatement()方法的默认类型

* ResultSet.CONCUR\_UPDATABLE

结果集可以用于更新数据库。使用这个选项，就可以在结果集中插入、删除或更新数据行，而这种改变将反映在数据库中。

**try** { //加载MySQL数据库驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://localhost:3306/javase?useUnicode=true&charterEncoding=utf-8";

String user = "root";

String password = "";

Connection conn = DriverManager.*getConnection*(url, user, password);

//设置可滚动的结果集

Statement stmt = conn.createStatement(ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet.***CONCUR\_READ\_ONLY***);

ResultSet rs = stmt.executeQuery("select \* from t\_user");

rs.last(); //将游标移动到最后一行

System.***out***.println(rs.getString(1)); //ID等于4

rs.first(); //将游标移动到第一行

System.***out***.println(rs.getString(1)); //ID等于1

conn.close(); //关闭资源

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

对于PreparedStatement对象，在Connection接口中同样提供了另外一个重载的prepareStatement()方法：

PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)

创建一个 PreparedStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。

**try** { //加载MySQL数据库驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

//建立数据连接

String url = "jdbc:mysql://localhost:3306/javase?useUnicode=true&charterEncoding=utf-8";

String user = "root";

String password = "";

Connection conn = DriverManager.*getConnection*(url, user, password);

//设置可滚动的结果集

String sql = "select \* from t\_user";

PreparedStatement ps = conn.prepareStatement(sql, ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet. ***CONCUR\_READ\_ONLY***);

ResultSet rs = ps.executeQuery();

rs.last(); //将游标移动到最后一行

System.***out***.println(rs.getString(1)); //ID等于4

rs.first(); //将游标移动到第一行

System.***out***.println(rs.getString(1)); //ID等于1

conn.close(); //关闭资源

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

1. **可更新的结果集**

如果希望编辑结果集中的数据，并且将结果集上的数据变更自动反应到数据库中， 那么就必须使用可更新的结果集。

可更新的结果集并非必须是可滚动的，但如果将数据提供给用户去编辑，那么通常也会希望结果集时可滚动的；

如果要获得可更新的结果集，应该使用以下方法创建一条语句：

PreparedStatement ps = conn.prepareStatement(sql, ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet.***CONCUR\_UPDATABLE***);

并非所有的查询都会返回可更新的结果集；如果查询涉及多个表的连接操作， 那么它所产生的结果集将是不可更新的；可以调用 ResultSet 接口中的 getConcurrency 方法来确定结果集是否是可更新的；

Connection conn = DriverManager.*getConnection*(url, user, password);

//设置可滚动和可更新的结果集

Statement stmt = conn.createStatement(ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet.***CONCUR\_UPDATABLE***);

ResultSet rs = stmt.executeQuery("select \* from t\_user");

**while**(rs.next()){

**if**("xxx".equals(rs.getString("pwd"))){

rs.updateString("pwd", "123"); //定义修改结果集中的值

rs.updateRow(); //执行修改数据库中的值

}

}

在使用第一个参数为列序号的updateXXX 方法时，请注意这里的列序号指的是该列在结果集中的序号，而不是数据库的中的列序号；

udpateXXX 方法：改变的只是结果集中的行值，而非数据库中的值；当更新完字段后，必须调用updateRow方法，这个方法将当前行中的更新信息发送给数据库；

cancelRowUpdates方法：取消对当前行的更新；

1. 添加一列数据

如果需要在结果集中添加一行数据，并且需要将添加的数据反应在数据库中，可以通过以下步骤来实现：

使用 moveToInsertRow 方法：将游标移动到特定的位置，我们称之为插入行；

调用 udpateXXX 方法 在插入行的位置上创建一个新的行；

调用insertRow 方法：将新建的行发送给数据库；

完成插入后，调用moveToCurrentRow方法：将游标移动到调用moveToCurrentRow方法之前的位置；

//设置可滚动和可更新的结果集

Statement stmt = conn.createStatement(ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet.***CONCUR\_UPDATABLE***);

ResultSet rs = stmt.executeQuery("select \* from t\_user");

//将光标移动到插入行。插入一行

rs.moveToInsertRow();

//向插入一行设置值,如果数据库中的字段为自增长，则无需设置值

//如果设置了值，则会按照设置的值来存储

rs.updateString("name", "nnn");

rs.updateString("pwd", "nnn");

//向数据库中添加一行数据

rs.insertRow();

//将光标移动到记住的光标位置，通常为当前行

rs.moveToCurrentRow();

你无法控制在结果集或数据库中添加新数据的位置；

对于在插入行中没有指定值的列，将被设置为null，而如果这个列有not null约束的话，那么将会抛出异常，而这一行无法插入；

1. 删除指定的行

想要删除结果集中指定的行，需要将游标移动到需要删除行上。

Connection conn = DriverManager.*getConnection*(url, user, password);

//设置可滚动和可更新的结果集

Statement stmt = conn.createStatement(ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,

ResultSet.***CONCUR\_UPDATABLE***);

ResultSet rs = stmt.executeQuery("select \* from t\_user");

//将光标移动到最后一行上

rs.last();

//删除结果集和数据库光标所在的行数据

rs.deleteRow();

#### 实例

数据库连接的封装（配置文件jdbc.properties）

配置文件放与src的根目录下

url=jdbc:mysql://localhost:3306/javase?useUnicode=true&characterEncoding=utf-8

driver=com.mysql.jdbc.Driver

userName=root

pwd=

//单例模式封装连接数据库（原生的JDBC数据库连接）

**public** **class** JdbcUtil {

**private** String driver;**private** **static** JdbcUtil *jdbcUtil*;**private** String url;

**private** String userName;**private** String pwd;

**private** JdbcUtil() {

Properties pro = **new** Properties();

//定义初始化类是加载数据库连接的驱动

**try** {

pro.load(JdbcUtil.**class**.getClassLoader().getResourceAsStream("jdbc.properties"));

**this**.driver = pro.getProperty("driver");

**this**.url = pro.getProperty("url");

**this**.userName = pro.getProperty("userName");

**this**.pwd = pro.getProperty("pwd");

Class.*forName*(driver);

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

//定义获取该对象的方法

**public** **static** JdbcUtil getInstance() {

**if** (**null** == *jdbcUtil*) {

**synchronized** (JdbcUtil.**class**) {

**if**(**null** == *jdbcUtil*) {

*jdbcUtil* = **new** JdbcUtil();

}

}

}

**return** *jdbcUtil*;

}

//获取Connection对象的方法

**public** Connection getConnection() **throws** SQLException {

Connection conn = DriverManager.*getConnection*(**this**.url, **this**.userName, **this**.pwd);

**return** conn;

}

//关闭数据库练级的方法

**public** **void** closeConnection(Connection conn){

**if** (**null** != conn){

**try** {

conn.close();

}

**catch** (SQLException e){

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

数据库操作语句的封装

//定义操作数据库的类：增、删、查、改

**public** **class** OperationData {

ConnJDBC c = ConnJDBC.*getConnJDBC*();

**private** Connection conn = c.getConn();

**private** PreparedStatement ps = **null**;

**private** ResultSet rs = **null**;

//定义增、删、修操作的方法

**public** **int** operation(String sql, Object...objects){

**int** num = 0;

**try** {

ps = conn.prepareStatement(sql);

**for**(**int** i = 0; i < objects.length; i++){//遍历传入的可变参数，将参数设置进SQL语句中

ps.setObject(i + 1, objects[i]);

}

num = ps.executeUpdate();

} **catch** (SQLException e) {

**return** num;

}

**return** num;

}

//定义查询数据库表中所有的数据结果

**public** List getDataAll(String sql, Class clazz, Object...objects){

List<Object> lists = **new** ArrayList<>();

ps = conn.prepareStatement(sql);

**if**(**null** != objects){

**for**(**int** i = 0; i < objects.length; i++){

ps.setObject(i + 1, objects[i]);

}

}

rs = ps.executeQuery();

**if**(**null** != rs){

ResultSetMetaData rsmd = rs.getMetaData();//获取原数据ResultSetMetaData 对象

**int** num = rsmd.getColumnCount();//获取原数据中的列数

**while**(rs.next()){ //遍历结果集

Object obj = clazz.newInstance();//利用反射获取类的对象

**for**(**int** i = 1; i <= num; i++){

String name = rsmd.getColumnName(i); //获取原数据中的列名称

//通过数据库的列名获取对象的属性；前提是属性名要和数据库字段名相同

Field field = clazz.getDeclaredField(name);

field.setAccessible(**true**); //强制忽略属性的权限修饰符

Object value = rs.getObject(name);

field.set(obj, value); //对属性进行赋值

}

lists.add(obj);

}

}

**return** lists;

}

}

### 数据库连接池

#### DBCP开源连接池

是Apache上的一个 Java连接池项目，也是 tomcat使用的连接池组件。单独使用dbcp需要导入common-dbcp.jar包。由于建立数据库连接是一个非常耗时耗资源的行为，所以通过连接池预先同数据库建立一些连接，放在内存中，应用程序需要建立数据库连接时直接到连接池中申请一个就行，用完后再放回去。dbcp没有自动的去回收空闲连接的功能。

1. 下载DBCP工具包：http://commons.apache.org/proper/commons-dbcp/download\_dbcp.cgi 解压后将commons-dbcp2-2.1.1.jar 导入项目bil文件夹。
2. 实现DBCP数据库连接池的方式：

**使用简单版配置文件的方式（其中连接池参数需要代码来配置）**

**配置文件（jdbc.properties）**

#数据库URL

url=jdbc:mysql://localhost:3306/javase?useUnicode=true&characterEncoding=utf8

#JDBC驱动类

driver=com.mysql.jdbc.Driver

#数据库登录用户名

username=root

#数据库登录密码

password=

**实现代码**

//获取ProperTies对象用于操作配置文件数据

Properties p = **new** Properties();

//利用反射机制加载配置文件

p.load(ConnectionUtil.**class**.getClassLoader().getResourceAsStream("jdbc.properties"));

String driver = p.getProperty("driver");

String url = p.getProperty("url");

String user = p.getProperty("user");

String pwd = p.getProperty("pwd");

//定义一个连接池对象BasicDataSource

BasicDataSource bds = **new** BasicDataSource();

//设置数据库连接参数

bds.setDriverClassName(driver);

bds.setUrl(url);

bds.setUsername(user);

bds.setPassword(pwd);

//设置连接池参数

bds.setMaxActive(50); //设置最大的激活的连接数

bds.setMaxIdle(50); //设置闲置的连接数

bds.setMaxWait(1000); //设置连接的等待时间

//获取Connection连接对象

Connection conn = bds.getConnection();

}

**连接池的参数设置：**

setInitialSize(int initialSize)

初始化连接，连接池启动时创建的初始化连接数量（默认值为0）

setMaxActive(int maxActive)

**使用标准的配置文件的方式**

使用读取配置文件的方式，新建一个dbcp.properties配置文件，并且放在src根目录下。

#############DBCP配置文件##############

#驱动名

driverClassName=com.mysql.jdbc.Driver

#url

url=jdbc:mysql://localhost:3306/mydb

#用户名

username=root

#密码

password=123456

#初始连接数

initialSize=30

#最大活跃数

maxTotal=30

#最大idle数

maxIdle=10

#最小idle数

minIdle=5

#最长等待时间（毫秒）

maxWaitMillis=1000

#程序中的连接不使用后是否被连接池回收

#（该版本要使用removeAbandonedOnMaintenance和removeAbandonedOnBorrow）

#removeAbandoned=true

removeAbandonedOnMaintenance=true

removeAbandonedOnBorrow=true

#连接在所指定的藐视内未使用才会被删除（秒）

removeAbandonedTimeout=1

**实现连接池：**

**public** **class** DBCPUtil {

**private** **static** Properties *p* = **new** Properties();

**private** **static** DataSource *data*;

//利用静态代码块加载配置文件

**static**{

//利用反射加载配置文件

**try** {

*p*.load(DBCPUtil.**class**.getClassLoader().getResourceAsStream("dbcp.properties"));

*data* = BasicDataSourceFactory.*createDataSource*(*p*);

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (Exception e) {

e.printStackTrace();

}

}

//从连接池中获取一个连接

**public** Connection getConn() **throws** SQLException{

**return** *data*.getConnection();

}

}

#### **C3P0开源连接池**

C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。c3p0是异步操作的，缓慢的JDBC操作通过帮助进程完成。扩展这些操作可以有效的提升性能。目前使用它的开源项目有hibernate，spring等。c3p0有自动回收空闲连接功能。

使用C3P0连接池具体实现步奏：

1. 下载C3P0工具包：https://sourceforge.net/projects/c3p0/files/latest/download 解压后将lib文件夹下的3个jar包导入到你的项目中。
2. 配置C3P0连接池

主要包括：初始化连接池时建立多少个连接、连接池最少多少个连接最多容纳多少连接、每个连接的生存时间、连接池能同时允许多少个操作进行，以及对具体数据库连接的配置：数据库的驱动、数据库的URL、数据库登录名、数据库密码、对这个数据库的连接池的细化配置（比如初始化时建立多少连接，最多最少连接数等等）。一个数据库的连接池配置用一个<named-config name="标识"> </name-config>节点来定义。在C3P0Utils中创建连接池时把 “标识” 作为连接池的构造函数的参数传入，则C3P0在配置文件中找到同名节点，按照这个节点的配置来创建相应配置的连接池。

c3p0的配置方式分为三种，分别是：

1. setXXX方法一个个地设置各个配置项
2. 类路径下提供一个c3p0.properties文件
3. 类路径下提供一个c3p0-config.xml文件

第一种：使用setXXX方法设置属性值：

**public** **class** Test {

Connection conn = **null**;

ComboPooleDataSource cd = **new** ComboPooleDataSource();

**public** Connection getConn(){

**try**{

cd.setDriverClass("com.mysql.jdbc.Driver");

cd.setJdbcUrl("jdbc:mysql://localhost:3306/javase??useUnicode=true&charterEncoding=utf-8");

cd.setUser("root");

cd.setPassword("123456");

conn = cd.getConnection();

}**catch**(SQLException e){

e.printStackTrace();

}**catch**(PropertyVetoException e){

e.printStackTrace();

}

**return** conn;

}

}

第二种：使用读取配置文件的方式,要求是配置文件必须命名为c3p0-config.xml,并且放在src根目录下，ComboPooledDataSource对象会自动读取名为c3p0-config.xml配置文件中的配置。配置文件如下:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<c3p0-config>

<!-- 默认配置 -->

<default-config>

<property name=*"initialPoolSize"*>10</property>

<property name=*"maxIdleTime"*>30</property>

<property name=*"maxPoolSize"*>100</property>

<property name=*"minPoolSize"*>10</property>

<property name=*"maxStatements"*>200</property>

</default-config>

<!-- 配置连接池mysql -->

<named-config name=*"mysql"*>

<property name=*"driverClass"*>com.mysql.jdbc.Driver</property>

<property name=*"jdbcUrl"*>jdbc:mysql://localhost:3306/javase</property>

<property name=*"user"*>root</property>

<property name=*"password"*>123456</property>

<property name=*"initialPoolSize"*>10</property>

<property name=*"maxIdleTime"*>30</property>

<property name=*"maxPoolSize"*>100</property>

<property name=*"minPoolSize"*>10</property>

<property name=*"maxStatements"*>200</property>

</named-config>

<!-- 配置连接池2 -->

...........

</c3p0-config>

实现方法：

**public** **class** Test {

Connection conn = **null**;

ComboPooleDataSource cd = **new** ComboPooleDataSource("mysql");

**public** Connection getConn(){

**try**{

conn = cd.getConnection();

}**catch**(SQLException e){

e.printStackTrace();

}

**return** conn;

}

}

第三种：使用读取配置文件的方式,要求是配置文件必须命名为c3p0.properties,并且放在src目录下，ComboPooledDataSource对象会自动读取名为c3p0.properties配置文件中的配置配置文件如下:

c3p0.driverClass=com.mysql.jdbc.Driver

c3p0.jdbcUrl=jdbc:mysql://localhost:3306/javase

c3p0.user=root

c3p0.password=123456

c3p0.initialPoolSize=3

c3p0.minPoolSize=3

c3p0.maxPoolSize=15

c3p0.acquireIncrement=3

c3p0.maxIdleTime=1000

c3p0.acquireRetryAttempts=30

c3p0.acquireRetryDelay=1000

实现方法：

**public** **class** Test {

Connection conn = **null**;

ComboPooleDataSource cd = **new** ComboPooleDataSource();

**public** Connection getConn(){

**try**{

conn = cd.getConnection();

}**catch**(SQLException e){

e.printStackTrace();

}

**return** conn;

}

}

## 国际化

## 安全

Java技术提供了以下三种确保安全的机制：

1. 语言涉及特性（对数组的边界进行检查，无不受检查的类型转换，无指针算法等）
2. 访问控制机制，用于控制代码能够执行的操作（比如文件访问，网络访问等）
3. 代码签名，利用该特性，代码的作者就能够用标准的加密算法来认证Java代码。这样该代码的使用者就能够准确地知道谁创建了该代码，以及代码被标识后是否被修改过。

### 类加载器

Java编译器会为虚拟机转换源指令。虚拟机代码存储在以.class为扩展名的类文件中，每个类文件都包含某个类或者接口的定义和代码实现。这些类文件必须由一个程序进行解析，该程序能够将虚拟机的指令集翻译成目标机器的机器语言。

### 字节码校验

### 安全管理器与访问权限

### 用户认证

### 数字签名

### 代码签名

### 加密

## 脚本、编译与注解处理

## 分布式对象

## 远程方法调用

Java远程方法调用，即Java RMI（Java Remote Method Invocation）是Java编程语言里，一种用于实现远程过程调用的应用程序编程接口。它使客户机上运行的程序可以调用远程服务器上的对象。远程方法调用特性使Java编程人员能够在网络环境中分布操作。RMI全部的宗旨就是尽可能简化远程接口对象的使用。

## 本地方法

## 拉姆达表达式

Java lambda表达式是Java 8中的新增功能.Java lambda表达式是Java进入函数式编程的第一步。因此，Java lambda表达式是可以在不属于任何类的情况下创建的函数。Java lambda表达式可以作为对象传递，并按需执行。

Java lambda表达式通常用于实现简单的事件侦听器/回调，或者使用Java Streams API进行函数式编程。

### Java Lambdas和单一方法接口

函数式编程经常用于实现事件侦听器。Java中的事件侦听器通常使用单个方法定义为Java接口。这是一个虚构的单一方法接口示例：

**public** **interface** StateChangeListener {

**public** **void** onStateChange(State oldState, State newState);

}

此Java接口定义了一个方法，只要状态发生变化（无论观察到什么），就会调用该方法。

在Java 7中，您必须实现此接口才能侦听状态更改。想象一下，你有一个StateOwner可以注册状态事件监听器的类。这是一个例子：

**public** **class** StateOwner {

**public** **void** addStateListener(StateChangeListener listener) { ... }

}

在Java 7中，您可以使用匿名接口实现添加事件侦听器，如下所示：

StateOwner stateOwner = **new** StateOwner();

stateOwner.addStateListener(**new** StateChangeListener(){

**public** **void** onStateChange(State oldState, State newState){

//使用旧状态和新状态执行某些操作。

}

});

首先StateOwner创建一个实例。然后StateChangeListener，在StateOwner实例上添加接口的匿名实现 作为侦听器。

在Java 8中，您可以使用Java lambda表达式添加事件侦听器，如下所示：

StateOwner stateOwner = **new** StateOwner();

stateOwner.addStateListener(

//lambda表达式

(oldState, newState) - > System.out.println("State changed")

);

lambda表达式与addStateListener()方法参数的参数类型匹配。如果lambda表达式与参数类型（在本例中为StateChangeListener接口）匹配，则lambda表达式将转换为实现与该参数相同的接口的函数。

Java lambda表达式只能在与它们匹配的类型是单个方法接口的情况下使用。在上面的示例中，lambda表达式用作参数，其中参数类型是 StateChangeListener接口。此接口只有一个方法。因此，lambda表达式与该接口成功匹配。

#### 匹配Lambda到接口

单个方法接口有时也称为功能接口。将Java lambda表达式与功能接口匹配分为以下步骤：

1. 接口是否只有一个抽象（未实现）方法？
2. lambda表达式的参数是否与单个方法的参数匹配？
3. lambda表达式的返回类型是否与单个方法的返回类型匹配？

如果对这三个问题的答案是肯定的，则给定的lambda表达式与接口成功匹配。

#### 与默认和静态方法的接口

在java 8中，java接口可以同时包含默认方法和静态方法。默认方法和静态方法都有一个直接在接口声明中定义的实现。这意味着，一个java lambda表达式可以使用多个方法实现接口-只要接口只有一个未实现的(也称为抽象)方法。换句话说，接口仍然是一个函数接口，即使它包含默认和静态方法，只要接口只包含一个未实现的(抽象)方法。

可以使用lambda表达式实现以下接口：

**import** java.io.IOException;

**import** java.io.OutputStream;

**public** **interface** MyInterface {

**void** printIt(String text);

**default** **public** **void** printUtf8To(String text, OutputStream outputStream){

**try** {

outputStream.write(text.getBytes("UTF-8"));

} **catch** (IOException e) {

**throw** **new** RuntimeException("Error writing String as UTF-8 to OutputStream", e);

}

}

**static** **void** printItToSystemOut(String text){

System.***out***.println(text);

}

}

尽管这个接口包含3个方法，但它可以由lambda表达式实现，因为只有其中一个方法未实现。下面是实现的样子：

MyInterface myInterface = (String text) -> {

System.out.print(text);

};

### Lambda表达式与匿名接口实现

尽管lambda表达式接近匿名接口实现，但有一些差异值得注意。主要区别在于，匿名接口实现可以具有状态(成员变量)，而lambda表达式不能。

**public** **interface** MyEventConsumer {

**public** **void** consume(Object event);

}

可以使用匿名接口实现来实现这个接口，如下所示：

MyEventConsumer consumer = **new** MyEventConsumer() {

**public** **void** consume(Object event){

System.out.println(event.toString() + " consumed");

}

};

这个匿名MyEventConsumer implementation可以有它自己的内部状态。看看这个重新设计：

MyEventConsumer myEventConsumer = **new** MyEventConsumer() {

**private** **int** eventCount = 0;

**public** **void** consume(Object event) {

System.out.println(event.toString() + " consumed " + **this**.eventCount++ + " times.");

}

};

注意匿名MyEventConsumer implementation现在定义了一个名为Eventcount的字段的。lambda表达式不能有这样的字段。因此，lambda表达式被认为是无状态的。

### Lambda型推理

在Java 8之前，您必须在进行匿名接口实现时指定要实现的接口。

stateOwner.addStateListener(**new** StateChangeListener() {

**public** **void** onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

使用lambda表达式，通常可以从周围的代码推断类型。例如，可以从addstatelistener()方法(statechangelistener接口上的单个方法)的方法声明中推断参数的接口类型。这被称为类型推断。编译器通过在其他地方查找类型来推断参数的类型-在本例中是方法定义。

stateOwner.addStateListener(

(oldState, newState) -> System.out.println("State changed")

);

在lambda表达式中，通常也可以推断参数类型。在上面的示例中，编译器可以从onstatechange()方法声明中推断它们的类型。因此，从onstatechange()方法的方法声明中可以推断出oldstate和newstate参数的类型。

### Lambda参数

因为java lambda表达式实际上只是方法，所以lambda表达式可以像方法一样接受参数。前面显示的lambda表达式的(oldstate，Newstate)部分指定lambda表达式接受的参数。这些参数必须与单个方法接口上的方法参数相匹配。

**public** **void** onStateChange(State oldState, State newState);

作为最低限度，lambda表达式和方法中的参数必须匹配。

其次，如果在lambda表达式中指定了任何参数类型，这些类型也必须匹配。

#### 零参数

如果要匹配lambda表达式的方法不接受参数，那么可以这样编写lambda表达式

() -> System.out.println("Zero parameter lambda");

注意括号之间是如何没有内容的。这是表示lambda不接受任何参数。

#### 一个参数

如果将Java lambda表达式与之匹配的方法接受一个参数，则可以编写如下所示的lambda表达式：

(param) -> System.out.println("One parameter: " + param);

注意，参数列在括号内。

当lambda表达式接受单个参数时，还可以省略括号，如下所示：

param -> System.out.println("One parameter: " + param);

#### 多个参数

如果与java lambda表达式匹配的方法接受多个参数，则需要在括号中列出参数。下面是java代码中的内容：

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

只有当方法接受单个参数时，才能省略括号。

#### 参数类型

如果编译器无法从lambda匹配的函数接口方法推断参数类型，则有时可能需要为lambda表达式指定参数类型。别担心，编译器会告诉您什么时候是这样的。下面是一个java lambda参数类型示例

(Car car) -> System.out.println("The car is: " + car.getName());

如您所见，car参数的类型(Car)写在参数名称本身的前面，就像在其他地方的方法中声明参数时一样，或者在对接口进行匿名实现时。

### Lambda函数体

lambda表达式的主体，以及它所代表的函数/方法的主体，在lambda声明中->的右边指定：下面是一个示例：

(oldState, newState) -> System.out.println("State changed")

如果您的lambda表达式需要由多行组成，则可以将lambda函数体括在{}括号内，在其他地方声明方法时，java也需要该括号。以下是一个例子：

(oldState, newState) -> {

System.out.println("Old state: " + oldState);

System.out.println("New state: " + newState);

}

### 从Lambda表达式返回值

您可以从java lambda表达式返回值，就像从方法返回值一样。只需向lambda函数体添加一个返回语句，如下所示：

(param) -> {

System.out.println("param: " + param);

**return** "return value";

}

如果您的lambda表达式所做的只是计算返回值并返回它，您可以用更短的方式指定返回值。而不是这个

(a1, a2) -> { **return** a1 > a2; }

你可以写：

(a1, a2) -> a1 > a2;

然后编译器计算出表达式A1>a2是lambda表达式的返回值(因此命名为lambda表达式-因为表达式返回某种类型的值)。

### Lambdas作为对象

java lambda表达式本质上是一个对象。您可以将lambda表达式赋值给变量并传递给它，就像处理任何其他对象一样。以下是一个例子：

**public** **interface** MyComparator {

**public** **boolean** compare(**int** a1, **int** a2);

}

MyComparator myComparator = (a1, a2) -> **return** a1 > a2;

**boolean** result = myComparator.compare(2, 5);

第一个代码块显示lambda表达式实现的接口。第二个代码块显示了lambda表达式的定义，如何将lambda表达式分配给变量，以及如何通过调用它实现的接口方法调用lambda表达式。

### 变量捕获

在某些情况下，java lambda表达式能够访问在lambda函数体之外声明的变量。

Javalambda可以捕获以下类型的变量：局部变量、实例变量、静态变量

#### 局部变量捕获

java lambda可以捕获在lambda主体之外声明的局部变量的值。为了说明这一点，首先看一下这个单一的方法接口：

**public** **interface** MyFactory {

**public** String create(**char**[] chars);

}

现在，查看实现myFactory接口的lambda表达式：

MyFactory myFactory = (chars) -> {

**return** **new** String(chars);

};

现在，这个lambda表达式只引用传递给它的参数值(Chars)。但我们可以改变这种状况。下面是引用在lambda函数体之外声明的字符串变量的更新版本：

String myString = "Test";

MyFactory myFactory = (chars) -> {

**return** myString + ":" + **new** String(chars);

};

如您所见，lambda主体现在引用局部变量myString，该变量在lambda主体之外声明。这是有可能的，只有当变量是引用是“有效的final”时，这意味着它在被赋值后不会改变它的值（常量）。如果myString变量的值后来发生了更改，编译器就会从lambda正文中抱怨对它的引用。

#### 实例变量捕获

lambda表达式还可以捕获创建lambda的对象中的实例变量。下面的一个例子表明：

**public** **class** EventConsumerImpl {

**private** String name = "MyConsumer";

**public** **void** attach(MyEventProducer eventProducer){

eventProducer.listen(e -> {

System.out.println(**this**.name);

});

}

}

请注意lambda主体中对this.name的引用。这捕获了包围事件使用者冲击对象的Name实例变量。甚至可以在实例变量捕获后更改其值，并且该值将反映在lambda中。

它的语义实际上是java lambda与接口的匿名实现不同的地方之一。匿名接口实现可以有自己的实例变量，通过该引用这些实例变量。然而，lambda不能有它自己的实例变量，所以这总是指向封闭的对象。

注：上述设计的事件消费者并不是特别优雅。我这样做就是为了说明实例变量捕获。

#### 静态变量捕获

java lambda表达式也可以捕获静态变量。这并不奇怪，因为只要静态变量是可访问的(打包的、作用域的或公共的)，就可以从Java应用程序中的任何地方访问静态变量。

下面是一个示例类，它创建一个lambda，它从lambda主体中引用一个静态变量：

**public** **class** EventConsumerImpl {

**private** **static** String *someStaticVar* = "Some text";

**public** **void** attach(MyEventProducer eventProducer){

eventProducer.listen(e -> {

System.out.println(someStaticVar);

});

}

}

静态变量的值也允许在lambda捕获它之后进行更改。

再一次，上面的设计是有点荒谬的。别想太多了。该类主要用于向您展示lambda可以访问静态变量。

### Lambda表达式的方法引用

如果您的lambda表达式只调用传递给lambda的参数的另一个方法，则java lambda实现提供了一种更短的方法来表示方法调用。首先，这里是单个函数接口的示例：

**public** **interface** MyPrinter{

**public** **void** print(String s);

}

下面是一个创建Java lambda实例的示例，该实例实现myprint接口：

MyPrinter myPrinter = (s) -> { System.out.println(s); };

因为lambda主体只包含一个语句，所以我们实际上可以省略包含{}括号的内容。另外，由于lambda方法只有一个参数，所以我们可以省略参数周围的括号。下面是生成的lambda声明的外观：

MyPrinter myPrinter = s -> System.out.println(s);

由于lambda主体所做的所有操作都是将字符串参数转发给system.out.println()方法，所以我们可以用方法引用替换上面的lambda声明。以下是lambda方法引用的外观：

MyPrinter myPrinter = System.out::println;

注意两个冒号：。这些信号向java编译器表示这是一个方法引用。所引用的方法是在双冒号之后出现的。拥有引用方法的任何类或对象都位于双冒号之前。

可以引用以下方法类型：参数对象实例方法、构造函数上的静态方法、实例方法。

#### 静态方法参考

最容易引用的方法是静态方法。这里首先是一个函数接口的示例：

**public** **interface** Finder {

**public** **int** find(String s1, String s2);

}

下面是要创建方法引用的静态方法：

**public** **class** MyClass{

**public** **static** **int** doFind(String s1, String s2){

**return** s1.lastIndexOf(s2);

}

}

最后，这里是一个引用静态方法的java lambda表达式：

Finder finder = MyClass::doFind;

由于finder.find()和myclass.dofind()方法的参数匹配，所以可以创建实现finder.find()并引用myclass.dofind()方法的lambda表达式。

#### 参数方法参考

还可以将其中一个参数的方法引用到lambda。假设有一个函数接口，如下所示：

**public** **interface** Finder {

**public** **int** find(String s1, String s2);

}

该接口用于表示能够搜索s1以查找s2出现的组件。下面是一个java lambda表达式的示例，该表达式调用string.indexof()来搜索

Finder finder = String::indexOf;

这相当于lambda的定义：

Finder finder = (s1, s2) -> s1.indexOf(s2);

请注意快捷版本如何引用单个方法。Java编译器将尝试将引用的方法与第一个参数类型相匹配，使用第二个参数类型作为被引用方法的参数。

#### 实例方法参考

第三，还可以从lambda定义引用实例方法。首先，让我们看看一个方法接口定义：

**public** **interface** Deserializer {

**public** **int** deserialize(String v1);

}

此接口表示能够将字符串“反序列化”为int的组件。

现在看一下这个字符串转换器类：

**public** **class** StringConverter {

**public** **int** convertToInt(String v1){

**return** Integer.*valueOf*(v1);

}

}

Converttoint()方法具有与反序列化器反序列化()方法的反序列化()方法相同的签名。因此，我们可以创建一个String转换器的实例，并从一个java lambda表达式中引用它的Converttoint()方法，如下所示：

StringConverter stringConverter = **new** StringConverter();

Deserializer des = stringConverter::convertToInt;

由这两行中的第二行创建的lambda表达式引用了在第一行上创建的字符串转换器实例的Converttoint方法。

#### 构造函数参考

最后，可以引用类的构造函数。要做到这一点，可以编写类名后面跟着：new，如下所示：

MyClass::**new**

也可以看到如何使用构造函数作为lambda表达式，请查看这个接口定义：

**public** **interface** Factory {

**public** String create(**char**[] val);

}

这个接口的create()方法匹配String类中一个构造函数的签名。因此，此构造函数可用作lambda。下面是一个这样的例子：

Factory factory = String::**new**;

这相当于这个java lambda表达式：

Factory factory = chars -> **new** String(chars);

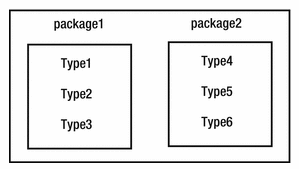
## 模块化编程

### java9之前

在 JDK 9之前，开发一个 Java 应用程序通常包括以下步骤：

1. Java源代码以Java类型（如类，接口，枚举和注释）的形式编写。
2. 不同的Java类型被安排在一个包（package）中，而且始终属于一个明确或默认的包。 一个包是一个逻辑的类型集合，本质上为它包含的类型提供一个命名空间。 即使声明为public，包可能包含公共类型，私有类型和一些内部实现类型。
3. 编译的代码被打包成一个或多个JAR文件，也称为应用程序JAR，因为它们包含应用程序代码。 一个程序包中的代码可能会引用多个JAR。
4. 应用程序可能使用类库。 类库作为一个或多个JAR文件提供给应用程序使用。
5. 通过将所有JAR文件，应用程序JAR文件和JAR类库放在类路径上来部署应用程序。

下图显示了JAR文件中打包的代码的典型布局。 该图仅显示了包和Java 类型，不包括其他内容，如manifest.mf文件和资源文件。



20多年来，Java社区以这种编写，编译，打包和部署Java代码的方式开发。 但是，20年漫长的旅程并没有像你所希望的一样顺利！ 这样安排和运行Java代码就存在固有的问题：

1. 一个包只是一个类型的容器，而不强制执行任何可访问性边界。包中的公共类型可以在所有其他包中访问；没有办法阻止在一个包中公开类型的全局可见性。
2. 除了以java和javax开头的包外，包应该是开放扩展的。如果你在具有包级别访问的JAR中进行了类型化，则可以在其他JAR中访问定义与你的名称相同的包中的类型。
3. Java运行时会看到从JAR列表加载的一组包。没有办法知道是否在不同的JAR中有多个相同类型的副本。Java运行时首先加载在类路径中遇到的JAR中找到的类型。
4. Java运行时可能会出现由于应用程序在类路径中需要的其中一个JAR引起的运行时缺少类型的情况。当代码尝试使用它们时，缺少的类型会引起运行时错误。
5. 在启动时没有办法知道应用程序中使用的某些类型已经丢失。还可以包含错误的JAR文件版本，并在运行时产生错误。

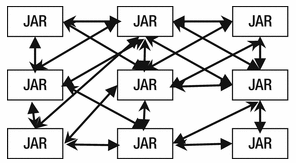
这些问题在Java社区中非常频繁和臭名昭着，他们得到了一个名字 ——JAR-hell。

包装JDK和JRE也是一个问题。 它们作为一个整体作为使用，从而增加了下载时间，启动时间和内存占用。 单体JRE使得Java不可能在内存很小的设备上使用。 如果将Java应用程序部署到云端，则需要支付更多的费用购买更多的使用内存。 大多数情况下，单体JRE使用的内存比所需的内存多，这意味着需要为云服务支付更多的内存。 Java 8中引入的Compact配置文件通过允许将JRE的一个子集打包在称为紧凑配置文件的自定义运行时映像中，大大减少了JRE大小，从而减少了运行时内存占用。

可以将JDK 9之前的JDK/JRE中的这些问题分为三类：

1. 不可靠的配置
2. 弱封装
3. JDK/JRE的单体结构

下图显示了Java运行时如何看到类路径上的所有JAR，以及如何从其他JAR访问一个JAR中的代码，没有任何限制，除了在访问控制方面由类型声明指定的代码。



Java 9通过引入开发，打包和部署Java应用程序的新方法来解决这些问题。 在Java 9中，Java应用程序由称为模块的小型交互组件组成。 Java 9也已经将JDK/JRE组织为一组模块。

### 全新的模块系统

Java 9引入了一个称为模块的新的程序组件。 您可以将Java应用程序视为具有明确定义的边界和这些模块之间依赖关系的交互模块的集合。 模块系统的开发具有以下目标：

1. 可靠的配置
2. 强封装
3. 模块化JDK/JRE

这些目标是解决Java 9之前开发和部署Java应用程序所面临的问题。

可靠的配置解决了用于查找类型的容易出错的类路径机制的问题。 模块必须声明对其他模块的显式依赖。 模块系统验证应用程序开发的所有阶段的依赖关系 —— 编译时，链接时和运行时。 假设一个模块声明对另一个模块的依赖，并且第二个模块在启动时丢失。 JVM检测到依赖关系丢失，并在启动时失败。 在Java 9之前，当使用缺少的类型时，这样的应用程序会生成运行时错误（不是在启动时）。

强大的封装解决了类路径上跨JAR的公共类型的可访问性问题。 模块必须明确声明其中哪些公共类型可以被其他模块访问。 除非这些模块明确地使其公共类型可访问，否则模块不能访问另一个模块中的公共类型。 Java 9中的公共类型并不意味着程序的所有部分都可以访问它。 模块系统增加了更精细的可访问性控制。

Tips：

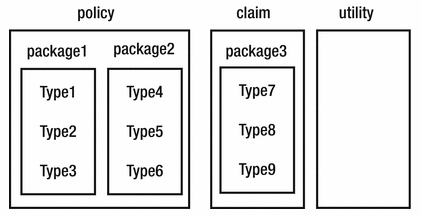
Java 9通过允许模块在开发的所有阶段声明明确的依赖关系并验证这些依赖关系来提供可靠的配置。它通过允许模块声明其公共类型可以访问其他模块的软件包来提供强大的封装。

JDK 9通过将其前身的体结构分解成一组称为平台模块的模块来重写。 JDK 9还引入了一个可选的阶段，称为链接时，这可能在编译时和运行时之间发生。 在链接期间，使用一个链接器，它是JDK 9附带的一个名为jlink的工具，用于创建应用程序的自定义运行时映像，其中仅包含应用程序中使用的模块。 这将运行时的大小调整到最佳大小。

### 什么是模块化

模块是代码和数据集合。 它可以包含Java代码和本地代码。 Java代码被组织为一组包含诸如类，接口，枚举和注解等类型的类。 数据可以包括诸如图像文件和配置文件的资源。

对于Java代码，模块可以看做零个或多个包的集合。 下图显示了三个名为policy，claim和utility的模块，其中policy模块包含两个包，claim模块包含一个包，而utility模块不包含任何包。



一个模块不仅仅是一个包的容器。 除了其名称，模块定义包含以下内容：

1. 所需的其他模块（或依赖于）的列表
2. 导出的软件包列表（其公共API），其他模块可以使用
3. 开放的软件包（其整个API，公共和私有）到其他反射访问模块的列表
4. 使用的服务列表（或使用java.util.ServiceLoader类发现和加载）
5. 提供的服务的实现列表

在使用这些模块时，可以使用这些方面中的一个或多个。

Java SE 9平台规范将平台划分为称为平台模块的一组模块。 Java SE 9平台的实现可能包含一些或所有平台模块，从而提供可扩展的Java运行时。 标准模块的名字是以Java 为前缀。 Java SE标准模块的示例有java.base，java.sql，java.xml和java.logging。 支持标准平台模块中的API，供开发人员使用。

非标准平台模块是JDK的一部分，但未在Java SE平台规范中指定。 这些JDK特定的模块的名称以jdk为前缀。 JDK特定模块的示例是jdk.charsets，jdk.compiler，jdk.jlink，jdk.policytool和jdk.zipfs。 JDK特定模块中的API不适用于开发人员。 这些API通常用于JDK本身以及不能轻易获得使用Java SE API所需功能的库开发人员使用。 如果使用这些模块中的API，则可能会在未经通知的情况下对其进行支持或更改。

JavaFX不是Java SE 9平台规范的一部分。 但是，在安装JDK/JRE时，会安装与JavaFX相关的模块。 JavaFX模块名称以javafx为前缀。 JavaFX模块的示例是javafx.base，javafx.controls，javafx.fxml，javafx.graphics和javafx.web。

作为Java SE 9平台的一部分的java.base模块是原始模块。 它不依赖于任何其他模块。 模块系统只知道java.base模块。 它通过模块中指定的依赖关系发现所有其他模块。 java.base模块导出核心Java SE软件包，如java.lang，java.io，java.math，java.text，java.time，java.util等。

### 模块依赖关系

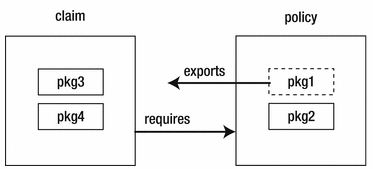
包括JDK 8之前的版本，一个包中的公共类型可以被其他包访问，没有任何限制。 换句话说，包没有控制它们包含的类型的可访问性。 JDK 9中的模块系统对类型的可访问性提供了细粒度的控制。

模块之间的可访问性是所使用的模块和使用模块之间的双向协议：模块明确地使其公共类型可供其他模块使用，并且使用这些公共类型的模块明确声明对第一个模块的依赖。 模块中的所有未导出的软件包都是模块的私有的，它们不能在模块之外使用。

将包中的 API 设置为公共供其他模块使用被称之为导出包。如果名为policy的模块将名为pkg1的包设置为公共类型可用于其他模块访问，则说明policy模块导出包pkg1。如果名为claim的模块声明对policy模块的依赖性，则称之为claim模块读取（read）policy模块。这意味着，在claim模块内部可以访问policy模块导出包中的所有公共类型。模块还可以选择性地将包导出到一个或多个命名模块。这种导出成为qualified导出或module-friendly导出。 qualified导出中的包中的公共类型只能访问指定的命名模块。

在模块系统的上下文中，可以互换使用三个术语 —— 需要（require），读取（read）和依赖（depend）。 以下三个语句意思相同：P读取Q，P需要Q，P依赖Q，其中P和Q指的是两个模块。

下图描述了两个名为policy和claim的模块之间的依赖关系。 policy模块包含两个名为pkg1和pkg2的包，它导出包pkg1，该包使用虚线边界显示，以将其与未导出的包pkg2区分开来。 claim模块包含两个件包pkg3和pkg4，它不导出包。 它声明了对policy模块的依赖。



在JDK 9中，您可以如下声明这两个模块：

module policy {

exports pkg1;

}

module claim {

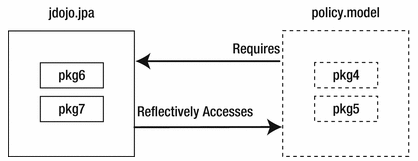
requires policy;

}

Tips：

用于指示模块中的依赖关系的语法是不对称的 ——导出一个包，但需要一个模块。

如果你的模块依赖于另一个模块，则该模块声明要求知道模块名称。几个Java框架和工具在很大程度上依赖于反射来在运行时访问未导出的模块的代码。它们提供了很大的功能，如依赖注入，序列化，Java Persistence API的实现，代码自动化和调试。Spring，Hibernate和XStream是这样的框架和库的例子。这些框架和库不了解你的应用程序模块。 但是，他们需要访问模块中的类型来完成他们的工作。 他们还需要访问模块的私有成员，这打破了JDK 9中强封装的前提。当模块导出软件包时，依赖于第一个模块的其他模块只能访问导出的软件包中的公共API。 在运行时，在模块的所有软件包上授予深入的反射访问权限（访问公共和私有API），可以声明一个开放的模块。



在JDK 9中，可以如下声明这两个模块：

open module policy.model {

requires jdojo.jpa;

}

module jdojo.jpa {

// The module exports its packages here

}

#### 模块图

模块系统只知道一个模块：java.base。 java.base模块不依赖于任何其他模块。 所有其他模块都隐含地依赖于java.base模块。

应用程序的模块化结构可以被视为一个称为模块图。 在模块图中，每个模块都表示为一个节点。 如果第一个模块依赖于第二个模块，则存在从模块到另一个模块的有向边。 通过将称为根模块的一组初始模块的依赖关系与称为可观察模块的模块系统已知的一组模块相结合来构建模块图。

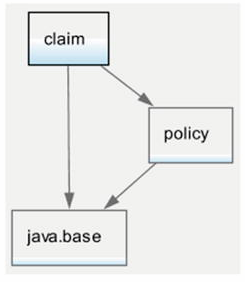
Tips

模块解析意味着该模块所依赖的模块可用。 假设名为P的模块取决于两个名为Q和R的模块。解析模块P表示您定位模块Q和R，并递归地解析模块Q和R。

构建模块图旨在在编译时，链接时和运行时解析模块依赖关系。 模块解析从根模块开始，并遵循依赖关系链接，直到达到java.base模块。 有时，可能在模块路径上有一个模块，但是会收到该模块未找到的错误。 如果模块未解析，并且未包含在模块图中，则可能会发生这种情况。 对于要解决的模块，需要从根模块开始依赖关系链。 根据调用编译器或Java启动器的方式，选择一组默认的根模块。 还可以将模块添加到默认的根模块中。 了解在不同情况下如何选择默认根模块很重要：

1. 如果应用程序代码是从类路径编译的，或者主类是从类路径运行的，则默认的根模块将由java.se模块和所有非“java.”系统模块组成，如“jdk.”和“JavaFX.”。 如果java.se模块不存在，则默认的根模块将由所有“java.”和非“java.\*”模块组成。
2. 如果您的应用程序由模块组成，则默认的根模块将依赖于以下阶段：
3. 在编译时，它由所有正在编译的模块组成。
4. 在链接时，它是空的。
5. 在运行时，它包含有主类的模块。 在java命令中使用--module或-m选项指定要运行的模块及其主类。

继续介绍policy和claim模块的例子，假设pkg3.Main是claim模块中的主类，并且两个模块都作为模块化JAR打包在C:\ Java9Revealed\lib目录中。下图显示了使用以下命令运行应用程序时在运行时构建的模块图：



C:\Java9Revealed>java -p lib -m claim/pkg3.Main

claim模块包含应用程序的主类。 因此，claim是创建模块图时唯一的根模块。 policy模块需要被解决，因为claim模块依赖于policy模块。 还需要解析java.base模块，因为所有其他模块都依赖于它，这两个模块也是如此。

模块图的复杂性取决于根模块的数量和模块之间的依赖关系。 假设除了依赖于policy模块之外，claim模块还取决于java.sql的平台模块。 claim模块的新声明如下所示：

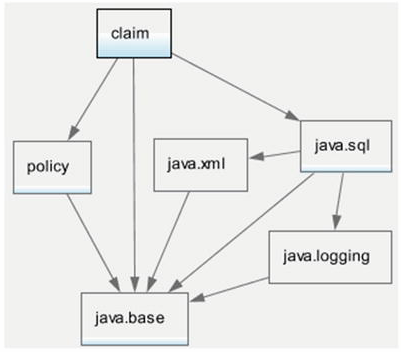
module policy {

requires policy;

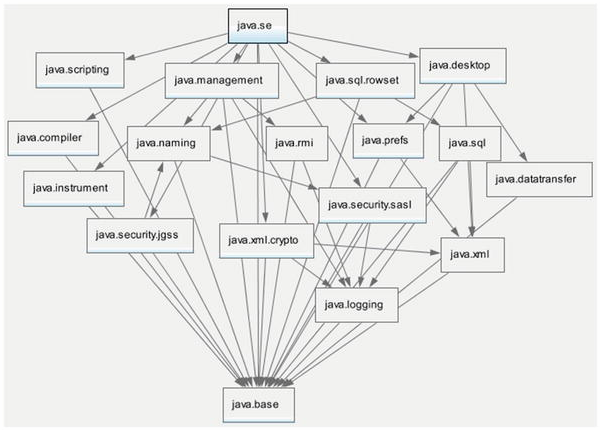
requires java.sql;

}

如下图，显示在claim模块中运行pkg3.Main类时将构建的模块图。 请注意，java.xml和java.logging模块也存在于图中，因为java.sql模块依赖于它们。 在图中，claim模块是唯一的根模块。



下图显示了java.se的平台模块的最复杂的模块图形之一。 java.se模块的模块声明如下：



module java.se {

requires transitive java.sql;

requires transitive java.rmi;

requires transitive java.desktop;

requires transitive java.security.jgss;

requires transitive java.security.sasl;

requires transitive java.management;

requires transitive java.logging;

requires transitive java.xml;

requires transitive java.scripting;

requires transitive java.compiler;

requires transitive java.naming;

requires transitive java.instrument;

requires transitive java.xml.crypto;

requires transitive java.prefs;

requires transitive java.sql.rowset;

requires java.base;

requires transitive java.datatransfer;

}

有时，需要将模块添加到默认的根模块中，以便解析添加的模块。 可以在编译时，链接时和运行使用--add-modules命令行选项指定其他根模块：



这里的<module-list>是逗号分隔的模块名称列表。

可以使用以下特殊值作为具有特殊含义的--add-modules选项的模块列表：

1. ALL-DEFAULT
2. ALL-SYSTEM
3. ALL-MODULE-PATH

所有三个特殊值在运行时都有效。 只能在编译时使用ALL-MODULE-PATH。

如果使用ALL-DEFAULT作为模块列表，则从应用程序从类路径运行时使用的默认的根模块集将添加到根集中。 这对于作为容器的应用程序是有用的，托管可能需要容器应用程序本身不需要的其他模块的其他应用程序。 这是一种使所有Java SE模块可用于容器的方法，因此任何托管的应用程序都可能使用到它们。

如果将ALL-SYSTEM用作模块列表，则将所有系统模块添加到根集中。 这对于运行测试时非常有用。

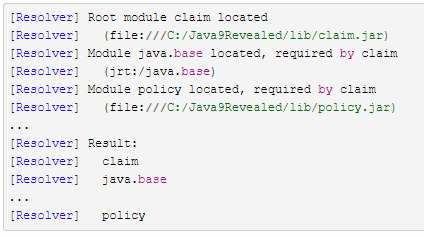
如果使用ALL-MODULE-PATH作为模块列表，则在模块路径上找到的所有模块都将添加到根集中。 这对于诸如Maven这样的工具非常有用，这确保了应用程序需要模块路径上的所有模块。

Tips：

即使模块存在于模块路径上，也可能会收到模块未找到的错误。 在这种情况下，需要使用--add-modules命令行选项将缺少的模块添加到默认的根模块中。

JDK 9支持一个有用的非标准命令行选项，它打印描述在构建模块图时用于解析模块的步骤的诊断消息。 选项是-Xdiag:resolver。 以下命令在声明模块中运行pkg3.Main类。 显示部分输出。 在诊断消息的结尾，你会发现一个结果：部分列出了解决模块。

使用命令C:\Java9Revealed>java -Xdiag:resolver -p lib -m claim/pkg3.Main，会得到如下输出：



### 聚合模块

你可以创建一个不包含任何代码的模块。 它收集并重新导出其他模块的内容。 这样的模块称为聚合模块。假设有几个模块依赖于五个模块。 您可以为这五个模块创建一个聚合模块，现在，你的模块只能依赖于一个模块 —— 聚合模块。

为了方便， Java 9包含几个聚合模块，如java.se和java.se.ee。 java.se模块收集Java SE的不与Java EE重叠的部分。 java.se.ee模块收集组成Java SE的所有模块，包括与Java EE重叠的模块。

### 声明模块

使用模块声明来定义模块，是Java编程语言中的新概念。其语法如下：

[open] module <module> {

<module-statement>;

<module-statement>;

...

}

open修饰符是可选的，它声明一个开放的模块。 一个开放的模块导出所有的包，以便其他模块使用反射访问。 <module>是要定义的模块的名称。 <module-statement>是一个模块语句。 模块声明中可以包含零个或多个模块语句。 如果它存在，它可以是五种类型的语句之一：

1. 导出语句（exports statement）；
2. 开放语句（opens statement）；
3. 需要语句（requires statement）；
4. 使用语句（uses statement）；

提供语句（provides statement）。

导出和开放语句用于控制对模块代码的访问。 需要语句用于声明模块对另一个模块的依赖关系。 使用和提供的语句分别用于表达服务消费和服务提供。 以下是名为myModule的模块的模块声明示例：

module myModule {

// Exports the packages - com.jdojo.util and

// com.jdojo.util.parser

exports com.jdojo.util;

exports com.jdojo.util.parser;

// Reads the java.sql module

requires java.sql;

// Opens com.jdojo.legacy package for reflective access

opens com.jdojo.legacy;

// Uses the service interface java.sql.Driver

uses java.sql.Driver;

// Provides the com.jdojo.util.parser.FasterCsvParser

// class as an implementation for the service interface

// named com.jdojo.util.CsvParser

provides com.jdojo.util.CsvParser

with com.jdojo.util.parser.FasterCsvParser;

}

你可以使用模块声明中的open修饰符来创建一个开放模块。 一个开放模块可以将其所有软件包的反射访问授予其他模块。 你不能在open模块中再使用open语句，因为所有程序包都是在open模块中隐式打开的。 以下代码段声明一个名为myLegacyModule的开放模块：

open module myLegacyModule {

exports com.jdojo.legacy;

requires java.sql;

}

#### 模块命名

模块名称可以是Java限定标识符。 合法标识符是一个或多个由点分隔的标识符，例如policy，com.jdojo.common和com.jdojo.util。 如果模块名称中的任何部分不是有效的Java标识符，则会发生编译时错误。 例如，com.jdojo.common.1.0不是有效的模块名称，因为名称中的1和0不是有效的Java标识符。

与包命名约定类似，使用反向域名模式为模块提供唯一的名称。 使用这个惯例，名为com.jdojo.common的最简单的模块可以声明如下：

module com.jdojo.common {

// No module statements

}

模块名称不会隐藏具有相同名称的变量，类型和包。 因此，可以拥有一个模块以及具有相同名称的变量，类型或包。 他们使用的上下文将区分哪个名称是指什么样的实体。

在JDK 9中， open, module, requires, transitive, exports, opens, to, uses, provides 和 with是受限关键字。只有当具体位置出现在模块声明中时，它们才具有特殊意义。 可以将它们用作程序中其他地方的标识符。 例如，以下模块声明是有效的，即使它不使用直观的模块名称：

// Declare a module named module

module module {

// Module statements go here

}

第一个模块字被解释为一个关键字，第二个是一个模块名称。

你可以在程序中的任何地方声明一个名为module的变量：

String module = "myModule";

#### 模块的访问控制

导出语句将模块的指定包导出到所有模块或编译时和运行时的命名模块列表。 它的两种形式如下：

exports <**package**>;

exports <**package**> to <module1>, <module2>...;

以下是使用了导出语句的模块示例：

module M {

exports com.jdojo.util;

exports com.jdojo.policy

to com.jdojo.claim, com.jdojo.billing;

}

开放语句允许对所有模块的反射访问指定的包或运行时指定的模块列表。 其他模块可以使用反射访问指定包中的所有类型以及这些类型的所有成员（私有和公共）。 开放语句采用以下形式：

opens <**package**>;

opens <**package**> to <module1>, <module2>...;

使用开放语句的实例：

module M {

opens com.jdojo.claim.model;

opens com.jdojo.policy.model to core.hibernate;

opens com.jdojo.services to core.spring;

}

Tips:

对比导出和打开语句。 导出语句允许仅在编译时和运行时访问指定包的公共API，而打开语句允许在运行时使用反射访问指定包中的所有类型的公共和私有成员。

如果模块需要在编译时从另一个模块访问公共类型，并在运行时使用反射访问类型的私有成员，则第二个模块可以导出并打开相同的软件包，如下所示：

module N {

exports com.jdojo.claim.model;

opens com.jdojo.claim.model;

}

阅读有关模块的时候会遇到三个短语：

1. 模块M导出包P
2. 模块M打开包Q
3. 模块M包含包R

前两个短语对应于模块中导出语句和开放语句。 第三个短语意味着该模块包含的包R既不导出也不开放。 在模块系统的早期设计中，第三种情况被称为“模块M隐藏包R”。

#### 声明依赖关系

需要（require）语句声明当前模块与另一个模块的依赖关系。 一个名为M的模块中的“需要N”语句表示模块M取决于（或读取）模块N。语句有以下形式：

requires <module>;

requires transitive <module>;

requires **static** <module>;

requires transitive **static** <module>;

require语句中的静态修饰符表示在编译时的依赖是强制的，但在运行时是可选的。requires static N语句意味着模块M取决于模块N，模块N必须在编译时出现才能编译模块M，而在运行时存在模块N是可选的。require语句中的transitive修饰符会导致依赖于当前模块的其他模块具有隐式依赖性。假设有三个模块P，Q和R，假设模块Q包含requires transitive R语句，如果如果模块P包含包含requires Q语句，这意味着模块P隐含地取决于模块R。

#### 配置服务

Java允许使用服务提供者和服务使用者分离的服务提供者机制。 JDK 9允许使用语句（uses statement）和提供语句（provides statement）实现其服务。

使用语句可以指定服务接口的名字，当前模块就会发现它，使用 java.util.ServiceLoader类进行加载。格式如下：



使用语句的实例如下：

module M {

uses com.jdojo.prime.PrimeChecker;

}

com.jdojo.PrimeChecker是一个服务接口，其实现类将由其他模块提供。 模块M将使用java.util.ServiceLoader类来发现和加载此接口的实现。

提供语句指定服务接口的一个或多个服务提供程序实现类。 它采取以下形式：



相同的模块可以提供服务实现，可以发现和加载服务。 模块还可以发现和加载一种服务，并为另一种服务提供实现。 以下是例子：

module P {

uses com.jdojo.CsvParser;

provides com.jdojo.CsvParser

with com.jdojo.CsvParserImpl;

provides com.jdojo.prime.PrimeChecker

with com.jdojo.prime.generic.FasterPrimeChecker;

}

### 模块描述符

在了解上一节中如何声明模块之后，你可能会对模块声明的源代码有几个疑问：

1. 在哪里保存模块声明的源代码？ 是否保存在文件中？ 如果是，文件名是什么？
2. 在哪里放置模块声明源代码文件？
3. 模块的声明的源代码如何编译？

#### 编译模块声明

模块声明存储在名为module-info.java的文件中，该文件存储在该模块的源文件层次结构的根目录下。 Java编译器将模块声明编译为名为module-info.class的文件。 module-info.class文件被称为模块描述符，它被放置在模块的编译代码层次结构的根目录下。 如果将模块的编译代码打包到JAR文件中，则module-info.class文件将存储在JAR文件的根目录下。

模块声明不包含可执行代码。 实质上，它包含一个模块的配置。 那为什么我们不在XML或JSON格式的文本文件中保留模块声明，而是在类文件中？ 类文件被选为模块描述符，因为类文件具有可扩展，明确定义的格式。 模块描述符包含源码级模块声明的编译形式。 它可以通过工具来增强，例如 jar工具，在模块声明初始编译之后，在类文件属性中包含附加信息。 类文件格式还允许开发人员在模块声明中使用导入和注解。

#### 模块版本

在模块系统的初始原型中，模块声明还包括模块版本的。 包括模块版本在声明中使模块系统的实现复杂化，所以模块版本从声明中删除。

模块描述符（类文件格式）的可扩展格式被利用来向模块添加版本。 当将模块的编译代码打包到JAR中时，该jar工具提供了一个添加模块版本的选项，最后将其添加到module-info.class文件中。

#### 模块源文件结构

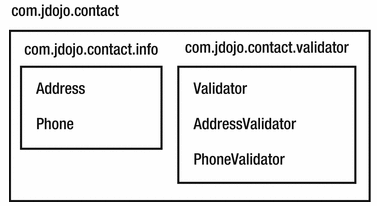
我们来看一个组织源代码和一个名为com.jdojo.contact的模块的编译代码的例子。 该模块包含用于处理联系信息的包，例如地址和电话号码。 它包含两个包：

com.jdojo.contact.info

com.jdojo.contact.validator

com.jdojo.contact.info包中包含两个类 —— Address 和 Phone。 com.jdojo.contact.validator包中包含一个名为Validator的接口和两个名为AddressValidator和PhoneValidator的类。

下图显示了com.jdojo.contact模块中的内容



在Java 9中，Java编译器工具javac添加了几个选项。 它允许一次编译一个模块或多个模块。 如果要一次编译多个模块，则必须将每个模块的源代码存储在与模块名称相同的目录下。 即使只有一个模块，也最好遵循此源目录命名约定。

假设你想编译com.jdojo.contact模块的源代码。 可以将其源代码存储在名为C:\j9r\src的目录中，其中包含以下文件：



请注意，需要遵循包层次结构来存储接口和类的源文件。

如果要一次编译多个模块，则必须将源代码目录命名为com.jdojo.contact，这与模块的名称相同。 在这种情况下，可以将模块的源代码存储在名为C:\j9r\src的目录中，其目录如下：



模块的编译后代码将遵循与之前看到的相同的目录层次结构。

### 打包模块

模块的artifact可以存储在：

目录中

模块化的JAR文件中

JMOD文件中，它是JDK 9中引入的一种新的模块封装格式

#### 目录中的模块

当模块的编译代码存储在目录中时，目录的根目录包含模块描述符（module-info.class文件），子目录是包层次结构的镜像。 继续上一节中的示例，假设将com.jdojo.contact模块的编译代码存储在C:\j9r\mods com.jdojo.contact目录中。 目录的内容如下：



#### 模块化JAR中的模块

JDK附带一个jar工具，以JAR（Java Archive）文件格式打包Java代码。 JAR格式基于ZIP文件格式。 JDK 9增强了在JAR中打包模块代码的jar工具。 当JAR包含模块的编译代码时，JAR称为模块化JAR。 模块化JAR在根目录下包含一个module-info.class文件。

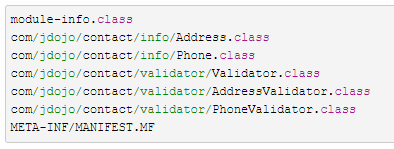
无论在JDK 9之前使用JAR，现在都可以使用模块化JAR。 例如，模块化JAR可以放置在类路径上，在这种情况下，模块化JAR中的module-info.class文件将被忽略，因为module-info在Java中不是有效的类名。

在打包模块化JAR的同时，可以使用JDK 9中添加的jar工具中可用的各种选项，将模块描述符中的信息例如模块版本添加到主类中。

Tips:

模块化JAR在各个方面来看都是一个JAR，除了它在根路径下包含的模块描述符。通常，比较重要的Java应用程序由多个模块组成。 模块化JAR可以是一个模块，包含编译的代码。 需要将应用程序的所有模块打包到单个JAR中。

继续上一节中的示例，com.jdojo.contact模块的模块化JAR内容如下。 请注意，JAR在META-INF目录中始终包含一个MANIFEST.MF文件。



#### JMOD文件中的模块

JDK 9引入了一种称为JMOD的新格式来封装模块。 JMOD文件使用.jmod扩展名。 JDK模块被编译成JMOD格式，放在JDK\_HOME jmods目录中。例如，可以找到一个包含java.base模块内容的java.base.jmod文件。 仅在编译时和链接时才支持JMOD文件。 它们在运行时不受支持。

### 模块路径

自JDK开始以来，类路径机制查找类型已经存在。 类路径是一系列目录，JAR文件和ZIP文件。 当Java需要在各个阶段（编译时，运行时，工具使用等）中查找类型时，它会使用类路径中的条目来查找类型。

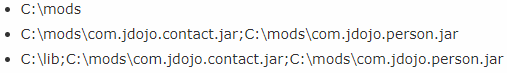
Java 9类型作为模块的一部分存在。 Java需要在不同阶段查找模块，而不是类似于Java 9之前的模块。Java 9引入了一种新的机制来查找模块，它被称为模块路径。

模块路径是包含模块的路径名称序列，其中路径名可以是模块化JAR，JMOD文件或目录的路径。 路径名由特定于平台的路径分隔符分隔，在UNIX平台上为冒号（:)，Windows平台上分号（;）。

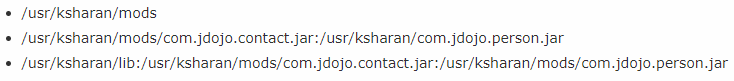
当路径名称是模块化的JAR或JMOD文件时，很容易理解。 在这种情况下，如果JAR或JMOD文件中的模块描述符包含要查找的模块的模块定义，则会找到该模块。 如果路径名是目录，则存在以下两种情况：

1. 如果类文件存在于根目录，则该目录被认为具有模块定义。 根目录下的类文件将被解释为模块描述符。 所有其他文件和子目录将被解释为此一个模块的一部分。 如果根目录中存在多个类文件，则首先找到的文件被解释为模块描述符。 经过几次实验，JDK 9似乎以按字母排列的顺序拾取了第一类文件。 这种存储模块编译代码的方式肯定会让你头疼。 因此，如果目录在根目录中包含多个类文件，请避免向模块路径添加目录。
2. 如果根目录中不存在类文件，则目录的内容将被不同的解释。 目录中的每个模块化JAR或JMOD文件被认为是模块定义。 每个子目录，如果它包含在它的根一个 module-info.class文件，被认为具有展开目录树格式的模块定义。 如果一个子目录的根目录不包含一个module-info.class文件，那么它不会被解释为包含一个模块定义。 请注意，如果子目录包含模块定义，则其名称不必与模块名称相同。 模块名称是从module-info.class文件中读取的。

以下是Windows上的有效模块路径：



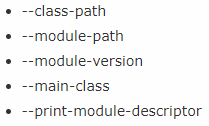
第一个模块路径包含名为C:\mods的目录的路径。 第二个模块路径包含两个模块化JAR——com.jdojo.contact.jar和com.jdojo.person.jar的路径。 第三个模块路径包含三个元素 —— 目录C:\lib的路径，以及两个模块化JAR——com.jdojo.contact.jar和com.jdojo.person.jar的路径。 在类似UNIX的平台上显示相当于这些路径：



避免模​​块路径问题的最佳方法是不要将分解的目录用作模块定义。

有两个目录作为模块路径 —— 一个包含所有应用程序模块化JAR的目录，另一个包含用于外部库的所有模块化JAR的目录。例如，可以使用C:\applib 和 C:\extlib作为Windows上的模块路径，其中C:\applib目录包含所有应用程序模块化JAR，C:\extlib目录包含所有外部库的模块化JAR。

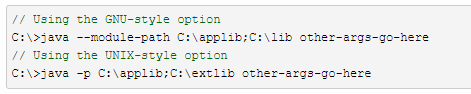
JDK 9已经更新了所有的工具来使用模块路径来查找模块。这些工具提供了指定模块路径的新选项。到JDK 9，已经看到以一个连字符（-）开头的UNIX样式选项，例如-cp和-classpath。在JDK 9中有如此多的附加选项，JDK设计人员对于开发人员来说也用完了有意义的短名称的选项。因此，JDK 9开始使用GNU样式选项，其中选项以两个连续的连字符开头，并且单词由连字符分隔。以下是GNU样式命令行选项的几个示例：



Tips:

要打印工具支持的所有标准选项的列表，使用--help或-h选项运行该工具，对于所有非标准选项，使用-X选项运行该工具。 例如，java -h和java -X命令将分别打印java命令的标准和非标准选项列表。

JDK 9中的大多数工具（如javac，java和jar）都支持两个选项来在命令行上指定一个模块路径。 它们是-p和--module-path。 将继续支持现有的UNIX样式选项以实现向后兼容性。 以下两个命令显示如何使用两个选项来指定java工具的模块路径：



当您使用GNU样式选项时，可以使用以下两种形式之一指定该选项的值：



上面的命令也可以写成如下形式：



当使用空格作为名称值分隔符时，需要至少使用一个空格。 您使用=作为分隔符时，不得在其周围包含任何空格。

### 可观察模块

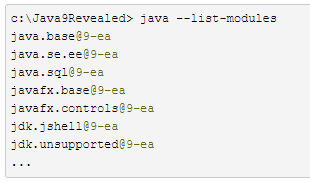
在模块查找过程中，模块系统使用不同类型的模块路径来定位模块。 在模块路径上与系统模块一起发现的一组模块被称为可观察模块。 可以将可观察模块视为模块系统在特定阶段可用的所有模块的集合，例如编译时，链接时和运行时，或可用于工具。

JDK 9为java命令添加了一个名为--list-modules的新选项。 该选项可用于打印两种类型的信息：可观察模块的列表和一个或多个模块的描述。 该选项可以以两种形式使用：



在第一种形式中，该选项没有跟随任何模块名称。 它打印可观察模块的列表。 在第二种形式中，该选项后面是逗号分隔的模块名称列表，用于打印指定模块的模块描述符。

以下命令打印可观察模块的列表，其中仅包括系统模块：



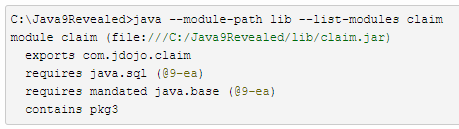
上面显示的是输出部分内容。 输出中的每个条目都包含两个部分—— 一个模块名称和一个由@符号分隔的版本字符串。 第一部分是模块名称，第二部分是模块的版本字符串。 例如，在java.base@9-ea中，java.base是模块名称，9-ea是版本字符串。 在版本字符串中，数字9表示JDK 9，ea代表早期访问。 运行命令时，你可能会得到不同的版本字符串输出。

现在在C:\Java9Revealed\lib目录中放置了三个模块化JAR。 如果提供此目录作为java命令的模块路径，这些模块将被包含在可观察模块列表中。以下命令显示了改变指定一个模块路径后，观察到的模块列表。 这里，lib目录是相对路径，C:\Java9Revealed是当前目录。



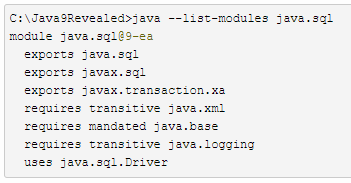
注意，对于应用程序模块，--list-modules选项还会打印它们的位置。 当获得意想不到的结果，并且不知道正在使用哪些模块以及哪些位置时，此信息有助于排除故障。

以下命令将com.jdojo.intro模块指定为--list-modules选项的参数，以打印模块的描述：



输出的第一行包含模块名称和包含该模块的模块化JAR位置。 第二行表示该模块导出com.jdojo.claim模块。 第三行表示该模块需要java.sql模块。 第四行表示模块强制依赖于java.base模块。 回想一下，除了java.base模块之外的每个模块都取决于java.base模块。 除了java.base模块，在每个模块的描述中看到需要强制的java.base模块。 第五行声明该模块包含一个名为pkg3的包，既不导出也不开放。

你还可以使用--list-modules打印系统模块的描述，例如java.base和java.sql。 以下命令打印出java.sql模块的描述。



### 总结

Java中的包已被用作类型的容器。 应用程序由放置在类路径上的几个JAR组成。 软件包作为类型的容器，不强制执行任何可访问性边界。 类型的可访问性内置在使用修饰符的类型声明中。 如果包中包含内部实现，则无法阻止程序的其他部分访问内部实现。 类路径机制在使用类型时线性搜索类型。 这导致在部署的JAR中缺少类型时，在运行时接收错误的另一个问题 —— 有时在部署应用程序后很长时间。 这些问题可以分为两种类型：封装和配置。

JDK 9引入了模块系统。 它提供了一种组织Java程序的方法。 它有两个主要目标：强大的封装和可靠的配置。 使用模块系统，应用程序由模块组成，这些模块被命名为代码和数据的集合。 模块通过其声明来控制模块的其他模块可以访问的部分。 访问另一个模块的部分的模块必须声明对第二个模块的依赖。 控制访问和声明依赖的是达成强封装的基础。 在应用程序启动时解决了一个模块的依赖关系。 在JDK 9中，如果一个模块依赖于另一个模块，并且运行应用程序时第二个模块丢失，则在启动时将会收到一个错误，而不是应用程序运行后的某个时间。 这是一个可靠的基础配置。

使用模块声明定义模块。 模块的源代码通常存储在名为module-info.java的文件中。 一个模块被编译成一个类文件，通常命名为module-info.class。 编译后的模块声明称为模块描述符。 模块声明不允许指定模块版本。 但诸如将模块打包到JAR中的jar工具的可以将模块版本添加到模块描述符中。

使用module关键字声明模块，后跟模块名称。 模块声明可以使用五种类型的模块语句：exports，opens，require，uses和provide。 导出语句将模块的指定包导出到所有模块或编译时和运行时的命名模块列表。 开放语句允许对所有模块的反射访问指定的包或运行时指定的模块列表， 其他模块可以使用反射访问指定包中的所有类型以及这些类型的所有成员（私有和公共）。 使用语句和提供模块语句用于配置模块以发现服务实现并提供特定服务接口的服务实现。

从JDK 9开始，open， module， requires， transitive, exports，opens，to，uses，provides和with都是受限关键字。 只有当具体位置出现在模块声明中时，它们才具有特殊意义。

模块的源代码和编译代码被安排在目录，JAR文件或JMOD文件中。 在目录和JAR文件中，module-info.class文件位于根目录。

与类路径类似，JDK 9引入了模块路径。 但是，它们的使用方式有所不同。 类路径用于搜索类型的定义，而模块路径用于查找模块，而不是模块中的特定类型。 Java工具（如java和javac）已经被更新为使用模块路径和类路径。 您可以使用--module-path或-p选项指定这些工具的模块路径。

JDK 9引入了与工具一起使用的GNU风格选项。 选项以两个破折号开头，每个单词用短划线分隔，例如--module-path，--class-path，--list-modules等。如果选项接受一个值，则该值可以跟随选项加上空格或=。 以下两个选项是一样的：



模块系统在某个阶段（编译时，运行时，工具等）中可用的模块列表被称为可观察模块。 可以使用--list-modules选项与java命令列出运行时可用的可观察模块。 还可以使用此选项打印模块的描述。

## Java日志

# 常用API

## 基本类型包装类

我们在开发中想要对基本类型数据进行更多的操作，最方便的方式就是将其封装成对象。包装类用于把基本数据类型转换为对象的类。每个基本类型在java.lang包中都有一个相应的包装类型。

|  |  |
| --- | --- |
| 基本数据类型 | 包装类 |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |

**基本数据类型包装成包装类的实例----装箱**

通过包装类的构造器实现：

**int** i = 500;

Integer a = **new** Integer(i);

还可以通过字符串参数构造包装类型对象

Float f = **new** Float("4.56");

Long l = **new** Long("asdf"); //NumberFormatException

**获得包装类对象中包装的基本类型变量---拆箱**

调用包装类的.xxxValue()方法

Boolean bobj = **new** Boolean(**true**);

**boolean** b = bobj.booleanValue();

JDK1.5之后，支持自动装箱，自动拆箱。但是类型必须匹配。

**字符串转成基本数据类型**

通过包装类的构造器实现

**int** i = **new** Integer("12");

通过包装类的parseXxx(String)静态方法

Float f = Float.*parseFloat*("12.1");

**基本数据类型转成字符串**

调用字符串重载的valueOf()方法

String fstr = String.*valueOf*(2.34f);

更直接的方式

String intStr = 5 + "";

## Object类

类 Object 是类层次结构的根类。每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。该类是所有类的父类，自定义类自动继承自Object类型。

* **常用方法**

**equals(Object obj)**

该方法比较两个对象是否“相等”。该方法比较的是两个对象的内存地址是否相等。所有类都继承了Object，也就获得了equals()方法，还可以重写

特例：当使用equals()方法进行比较时，对类File、String、Date及包装类来说，是比较类型及内容是否相等，而不考虑引用的是否是同一个对象。因为在这些类中重写了equalse()方法。

**equals 方法在非空对象引用上实现相等关系：**

* 自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。
* 对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。
* 传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。
* 一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。
* 对于任何非空引用值 x，x.equals(null) 都应返回 false。

Object 类的 equals 方法实现对象上差别可能性最大的相等关系；即，对于任何非空引用值 x 和 y，当且仅当 x 和 y 引用同一个对象时，此方法才返回 true（x == y 具有值 true）。

注意：当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

实例：

编写Order类，有int型的orderId，String型的orderName，相应的getter()和setter()方法，两个参数的构造器，重写父类的equals()方法：public boolean equals(Object obj)，并判断测试类中创建的两个对象是否相等

@Override //重写equals方法判断两个对象的值是否相等

**public** **boolean** equals(Object obj) {

**if** (obj == **null**){

**return** **false**;

}

Order other = (Order) obj;

**if**(other.getOrderId() == **this**.orderId && other.getOrderName().equals(**this**.orderName)){

**return** **true**;

}

**return** **false**;

}

**hashCode()**

该方法返回该对象的散列码。返回值是int类型的散列码。对象的散列码是为了更好的支持基于哈希机制的Java集合类，例如 Hashtable, HashMap, HashSet 等。

**hashCode 的常规协定是：**

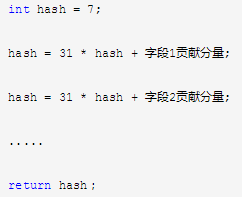
* 在 Java 应用程序执行期间，在对同一对象多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是将对象进行 equals 比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。
* 如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果。
* 如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么对这两个对象中的任一对象上调用 hashCode 方法不 要求一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能。

实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧。）

**hashCode编写指导**

在编写hashCode时，你需要考虑的是，最终的hash是个int值，而不能溢出。不同的对象的hash码应该尽量不同，避免hash冲突。那么如果做到呢？下面是解决方案。

* 定义一个int类型的变量 hash,初始化为 7。
* 接下来让你认为重要的字段（equals中衡量相等的字段）参入散列运，算每一个重要字段都会产生一个hash分量，为最终的hash值做出贡献（影响）
* 最后把所有的分量都总和起来，注意并不是简单的相加。选择一个倍乘的数字31，参与计算。然后不断地递归计算，直到所有的字段都参与了。



@Override //重写hashCode方法

**public** **int** hashCode() {

**int** result = 7;

result = 31 \* result + **this**.orderId;

result = 31 \* result + ((**this**.orderName == **null**) ? 0 : **this**.orderName.hashCode());

**return** result;

}

注意：equals()相等的两个对象，hashcode()一定相等；反过来：hashcode()不等，一定能推出equals()也不等；hashcode()相等，equals()可能相等，也可能不等。

**toString()**

toString()方法在Object类中定义，其返回值是String类型，返回类名和它的引用地址。

在进行String与其它类型数据的连接操作时，自动调用toString()方法

Date now = **new** Date();

System.***out***.println("now = " + now);

System.***out***.println("now = " + now.toString());

基本类型数据转换为String类型时，调用了对应包装类的toString()方法

可以根据需要在用户自定义类型中重写toString()方法。如String 类重写了toString()方法，返回字符串的值。

String str = "123";

//System.out.println(str.toString());

System.***out***.println(str); //123

## System类

System类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于java.lang包。由于该类的构造方法是private的，所以无法创建该类的对象，也就是无法实例化该类。其内部的成员变量和成员方法都是static的，所以也可以很方便的进行调用

* 成员变量

System类内部包含in、out和err三个成员变量，分别代表标准输入流(键盘输入)，标准输出流(显示器)和标准错误输出流(显示器)。

例如：

System.out.println(“Test”);

该行代码的作用是将字符串”Test”输出到系统的标准输出设备上，也就是显示在屏幕上。

后续在学习完IO相关的知识以后，可以使用System类中的成员方法改变标准输入流等对应的设备，例如可以将标准输出流输出的信息输出到文件内部，从而形成日志文件等。

* 常用成员方法

System类中提供了一些系统级的操作方法，这些方法实现的功能分别如下：

**arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**

该方法的作用是数组拷贝，也就是将一个数组中的内容复制到另外一个数组中的指定位置，由于该方法是native方法，所以性能上比使用循环高效。

**int**[] a = {1,2,3,4};

**int**[] b = **new** **int**[5];

System.*arraycopy*(a, 1, b, 3, 2); //[0,0,0,2,3]

该代码的作用是将数组a中，从下标为1开始，复制到数组b从下标3开始的位置，总共复制2个。也就是将a[1]复制给b[3]，将a[2]复制给b[4]，这样经过复制以后数组a中的值不发生变化，而数组b中的值将变成{0,0,0,2,3}。

**currentTimeMillis()**

该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和GMT时间(格林威治时间)1970年1月1号0时0分0秒所差的毫秒数。

例如，计算程序运行需要的时间则可以使用如下的代码

**long** start = System.*currentTimeMillis*();

**for**(**int** i = 0;i < 100000000; i++){

**int** a = 0;

}

**long** end = System.*currentTimeMillis*();

**long** time = end - start; //得出for循环代码执行总共用了好多毫秒

**getProperty(String key)**

该方法的作用是获得系统中属性名为key的属性对应的值。

|  |  |
| --- | --- |
| 属性名 | 属性说明 |
| java.version | Java 运行时环境版本 |
| java.vendor | Java 运行时环境供应商 |
| java.vendor.url | Java 供应商的 URL |
| java.home | Java 安装目录 |
| java.vm.specification.version | Java 虚拟机规范版本 |
| java.vm.specification.vendor | Java 虚拟机规范供应商 |
| java.vm.specification.name | Java 虚拟机规范名称 |
| java.vm.version | Java 虚拟机实现版本 |
| java.vm.vendor | Java 虚拟机实现供应商 |
| java.vm.name | Java 虚拟机实现名称 |
| java.specification.version | Java 运行时环境规范版本 |
| java.specification.vendor | Java 运行时环境规范供应商 |
| java.specification.name | Java 运行时环境规范名称 |
| java.class.version | Java 类格式版本号 |
| java.class.path | Java 类路径 |
| java.library.path | 加载库时搜索的路径列表 |
| java.io.tmpdir | 默认的临时文件路径 |
| java.compiler | 要使用的 JIT 编译器的名称 |
| java.ext.dirs | 一个或多个扩展目录的路径 |
| os.name | 操作系统的名称 |
| os.arch | 操作系统的架构 |
| os.version | 操作系统的版本 |
| file.separator | 文件分隔符（在 UNIX 系统中是“/”） |
| path.separator | 路径分隔符（在 UNIX 系统中是“:”） |
| line.separator | 行分隔符（在 UNIX 系统中是“/n”） |
| user.name | 用户的账户名称 |
| user.home | 用户的主目录 |
| user.dir | 用户的当前工作目录 |

String osName = System.*getProperty*("os.name");

String user = System.*getProperty*("user.name");

System.***out***.println("当前操作系统是：" + osName);

System.***out***.println("当前用户是：" + user);

## Random类

Random类是java.util包中的类，它实现的随机算法是伪随机，也就是有规则的随机。在进行随机时，随机算法的起源数字称为种子数(seed)，在种子数的基础上进行一定的变换，从而产生需要的随机数字。

Random使用一个48位的种子，相同种子数的Random对象，相同次数生成的随机数字是完全相同的。也就是说，两个种子数相同的Random对象，第一次生成的随机数字完全相同，第二次生成的随机数字也完全相同。这点在生成多个随机数字时需要特别

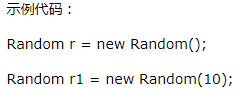
**Random类的构造方法：**

**Random()**

该构造方法使用一个和当前系统时间对应的相对时间有关的数字作为种子数，然后使用这个种子数构造Random对象。

**Random(long seed)**

该构造方法可以通过制定一个种子数进行创建。



注意：种子数只是随机算法的起源数字，和生成的随机数字的区间无关。

**常用方法**

**protected int next(int bits)**

生成下一个伪随机数。当被所有其他方法使用时，子类应该重写此方法。

**public boolean nextBoolean()**

返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 boolean 值。nextBoolean 的常规协定是，伪随机地生成并返回一个 boolean 值。值 true 和 false 的生成概率（大致）相同。

**public void nextBytes(byte[] bytes)**

生成随机字节并将其置于用户提供的 byte 数组中。所生成的随机字节数等于该 byte 数组的长度。

**public double nextDouble()**

返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 double 值。

**public float nextFloat()**

返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 float 值。

**public int nextInt()**

返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。nextInt 的常规协定是，伪随机地生成并返回一个 int 值。所有 232 个可能 int 值的生成概率（大致）相同。

**public int nextInt(int n)**

返回一个伪随机数，它是取自此随机数生成器序列的、在 0（包括）和指定值（不包括）之间均匀分布的 int 值。

**public long nextLong()**

返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 long 值。

## Date类

java.util.Date类表示特定的瞬间，精确到毫秒。在 JDK 1.1 之前，类 Date 有两个其他的函数。它允许把日期解释为年、月、日、小时、分钟和秒值。它也允许格式化和解析日期字符串。不过，这些函数的 API 不易于实现国际化。从 JDK 1.1 开始，应该使用 Calendar 类实现日期和时间字段之间转换，使用 DateFormat 类来格式化和解析日期字符串。Date 中的相应方法已废弃。

**构造方法**：

**public Date()**

分配 Date 对象并初始化此对象，以表示分配它的时间（精确到毫秒）。

**public Date(long date)**

分配 Date 对象并初始化此对象，以表示自从标准基准时间（称为“历元（epoch）”，即 1970 年 1 月 1 日 00:00:00 GMT）以来的指定毫秒数。

**常用方法：**

**public boolean after(Date when)**

测试此日期是否在指定日期之后。

**public boolean before(Date when)**

测试此日期是否在指定日期之前。

**public int compareTo(Date anotherDate)**

比较两个日期的顺序。如果参数 Date 等于此 Date，则返回值 0；如果此 Date 在 Date 参数之前，则返回小于 0 的值；如果此 Date 在 Date 参数之后，则返回大于 0 的值。

**public long getTime()**

返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。

**public void setTime(long time)**

设置此 Date 对象，以表示 1970 年 1 月 1 日 00:00:00 GMT 以后 time 毫秒的时间点。

**实例：**

Date date = **new** Date();

System.***out***.println(date.getTime()); //1508055933934

date.setTime(1508055933934L);

System.***out***.println(date); //Sun Oct 15 16:25:33 CST 2017

## SimpleDateFormat类

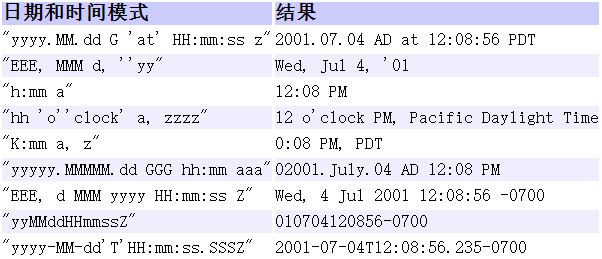
java.text.SimpleDateFormat类是一个以与语言环境有关的方式来格式化和解析日期的具体类。它允许进行格式化（日期 -> 文本）、解析（文本 -> 日期）和规范化。SimpleDateFormat 使得可以选择任何用户定义的日期-时间格式的模式。但是，仍然建议通过 DateFormat 中的 getTimeInstance、getDateInstance 或 getDateTimeInstance 来创建日期-时间格式器。每一个这样的类方法都能够返回一个以默认格式模式初始化的日期/时间格式器。

日期和时间模式：



示例：

以下示例显示了如何在美国语言环境中解释日期和时间模式。给定的日期和时间为美国太平洋时区的本地时间 2001-07-04 12:08:56。



**构造方法：**

**public SimpleDateFormat()**

用默认的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。注：此构造方法可能不支持所有语言环境。要覆盖所有地区，请使用 DateFormat 类中的工厂方法。

**public SimpleDateFormat(String pattern)**

用给定的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。注：此构造方法可能不支持所有语言环境。要覆盖所有语言环境，请使用 DateFormat 类中的工厂方法。

**public SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)**

用给定的模式和日期符号构造 SimpleDateFormat。pattern - 描述日期和时间格式的模式、formatSymbols - 要用来格式化的日期格式符号。

**public SimpleDateFormat(String pattern, Locale locale)**

用给定的模式和给定语言环境的默认日期格式符号构造 SimpleDateFormat。注：此构造方法可能不支持所有语言环境。要覆盖所有语言环境，请使用 DateFormat 类中的工厂方法。pattern - 描述日期和时间格式的模式、locale - 其日期格式符号要被使用的语言环境。

**常用方法：**

**public void applyLocalizedPattern(String pattern)**

将给定的本地化模式字符串应用于此日期格式。pattern - 要被映射到此格式的新的日期和时间格式模式的字符串。

**public void applyPattern(String pattern)**

将给定模式字符串应用于此日期格式。

**public Date parse(String source)**

解析字符串的文本，生成 Date。

**public Date parse(String text, ParsePosition pos)**

解析字符串的文本，生成 Date。此方法试图解析从 pos 给定的索引处开始的文本。如果解析成功，则将 pos 的索引更新为所用最后一个字符后面的索引（不必对直到字符串结尾的所有字符进行解析），并返回解析得到的日期。更新后的 pos 可以用来指示下次调用此方法的起始点。如果发生错误，则不更改 pos 的索引，并将 pos 的错误索引设置为发生错误处的字符索引，并且返回 null。

**public String toLocalizedPattern()**

返回描述此日期格式的本地化模式字符串。

**public String toPattern()**

返回描述此日期格式的模式字符串。

**public final String format(Date date)**

将给定的日期格式化后返回以字符串的表示。

**public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos)**

将给定的 Date 格式化为日期/时间字符串，并将结果添加到给定的 StringBuffer。date - 要被格式化为日期-时间字符串的日期-时间值、toAppendTo - 新的日期-时间文本被添加的位置、pos - 格式化的位置。输入时：如果需要，是一个对齐字段。输出时：为对齐字段的偏移量。

实例：

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat();

sdf.applyPattern("yyyy-MM-dd : HH-mm-ss");

date.setTime(1508055933934L);

System.***out***.println(sdf.format(date)); //2017-10-15 : 16-57-23

**try**{

//Sun Oct 15 16:57:23 CST 2017

System.***out***.println(sdf.parse(sdf.format(date)));

}**catch** (ParseException e) {

// **TODO**: handle exception

e.printStackTrace();

}

## Calendar类型

## Math类型

## BigInteger类和BigDecimal类

# Java加密解密算法