* 1. **final关键字**

（表示不可变的，最终的）

1、final修饰的类无法被子类继承；

eg：public final class A{} 无法被继承

2、final修饰的方法无法被覆盖（重写），可以继承但不能重写；

eg：public final void m1(){} 父类存在m1()，子类可以继承但不能重写m1()

3、final修饰的变量无法第二次赋值；

局部变量：可以进行一次赋值，不能二次赋值。（注：final int a;此语句没有进行一次赋值，无初始默认值）

实例变量：与局部变量不同处在于，实例变量存在默认值，因此在创建实例变量时必须进行赋值。（实例变量被final修饰后不能采用默认值，因此才需要必须手动赋值）。

手动赋值

方法1：final int i = 1;

方法2：构造方法内进行赋值

final int i;

public 类名(){

this.i = 1;

}

注：其实底层中两种方法其实是一样的。

（原因：方法1实例变量的赋值运算符在构造方法执行时才会运算。构造方法默认对实例变量进行初始化）

4、final修饰的引用不能进行二次赋值。（引用实际存储在栈内存中，是一个局部变量）

eg：User a = new User(1);

a = new User(2);

注：引用a存储的地址被替换，第一个对象无引用指向，被回收。

final User a = new User(1); // 不能再被修改，不被回收。

// a = new User(2);

注：修改类中的实例变量值可以。

final User a = new User(1);

a.id = 2;

5、final修饰的实例变量，一般和static联合使用，被称为常量。

语法：public static final 数据类型常量名 = 值;

eg: public static final String GUO\_JIA = "中国";

// java规范中要求所有常量的名字全部大写，每个单词之间用下划线隔开。

* 1. **包、import、权限**
     1. **包机制**

1、包又称为package，java中引入package主要是为了方便程序的管理。

不同功能的类被分门别类到不同的软件包中，查找方便，容易维护。

2、怎样定义package？

- 在java源程序的第一行编写package语句。

- package只能编写一个语句。

语法结构：

package 包名;

3、包名命名规范。

公司域名倒序 + 项目名 + 模块名 + 功能名

eg：org.apache.tocat.core;

com.bjpowernode.oa.user.service;

4、包名要求全部小写，包名也是标识符，必须遵守命名规范。

5、一个包对应一个目录

eg：org.apache.tocat.core; // 四个目录

6、使用包机制在终端内怎样编译？怎样运行？

包的机制实际创建了多个树形目录。在编译和运行时遵循文件具体地址位置编译。（注意在相对地址下运行）

即，在终端内运行步骤：

- 编译：javac java原文件路径（生成.class文件）

- 创建目录：手动创建目录，将生成的.class文件放在指定包目录下

- 运行：java 包目录.文件名

7、不导入包时，调用其他.class文件需要注意文件是否与当前.class文件在同一文件目录下。

在同一个包目录时，直接调用该类

不在同一包目录时，需要将包目录地址写在类文件前做引用。

* + 1. **import关键字使用**

1、语法：

import 包名.文件名 // 导入指定的.class文件

import 包名. // 导入指定包的所有类文件

2、import编写在package语句之下，class语句之上。

3、String与System类为什么不需要进行导入包就可以直接调用？

// java.lang. 不需要手动导入，系统自动导入

// lang:language语言包，是java核心类，不需要手动导入。、

* + 1. **权限**

public int a1; // 公开的<公开属性，都可以访问>

protected int a2; // 受保护的<只能在同包或继承该类的子类中直接访问>

int a3; // 缺省的<只能在同一个包内直接访问>

private int a4; // 私有的<只能在变量所在类直接访问>

* 1. **object类中的方法**
     1. **toString()方法**

返回对象的字符串表示形式。

Object类中的toString方法：

源码：

public String toString(){

return this.getClass().getName() + "@" + Integer.toHexString(hashCode());

}

设计toString()方法的作用：

返回对象的字符串表示形式。（返回一个字符串）

默认返回值：类名@对象的内存地址

注：1、使用System.out.println()输出对象，或者将对象与字符串进行连接时，系统自动调用 该对象的toString()方法返回该对象的字符串表示。

2、调用toString()方法时，返回了内存地址，实际上就是打印输出引用变量

3、建议所有子类修改（重写）toString()方法<原因：不能输出想要输出的字符串(值)>

4、未重写toString()方法时，子类继承父类的toString()方法。即，未重写时继承Object中的toString()方法。

* + 1. **equals()方法**

目的：比较两个引用类型的对象实际内容是否相等。

源码：

public boolean equals(Object obj){

return (this == obj);

}

equals()方法作用：

未重写时，实际比较的是两个对象的地址是否相等

**为什么不用" == "进行判断？**

原因："=="是判断两个基本数据类型是否相等的，不能判断同时使用new出来的对象是否相等，因为new出来的对象实际是对象地址，不同的对象地址不同。

注：系统提供的类，如String类（引用类型），已经对toString()与equals()方法进行了重写。

java规范：引用类型使用equals()方法，基本数据类型使用"=="。

重写equals()方法：

例：

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Time02 time02 = (Time02) o;

return year == time02.year &&

month == time02.month &&

day == time02.day;

}

* + 1. **hashCode()**

源码：

public native int hashCode();

// 这个方法不是抽象方法。底层调用c++程序

hashCode()方法返回的是哈希码：

实际上就是一个java对象的内存地址，经过哈希算法，得到的一个值。

<等同于一个对象的地址>

例：<经过哈希算法转换的一个数字，等同地址>

Object o1 = new Object();

int hashCodeValue1 = o1.hashCode(); // 460141958

* 1. **内部类**

定义：内部类就是在一个类的内部进行其他类结构的嵌套操作。

优点：

1、内部类与外部类可以访问彼此的私有域（私有属性，私有方法）

2、内部类是另外一种封装，对外部的其他类隐藏。

3、内部类可以实现java的单继承局限。

内部类分类：

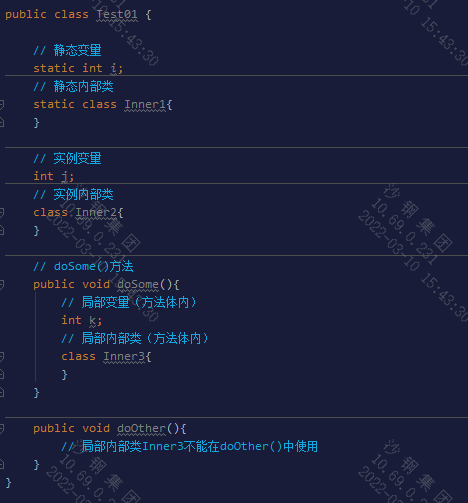
<成员内部类></>：类似成员变量

<静态内部类></>：类似静态变量

<局部内部类></>：类似局部变量

<匿名内部类></>：属于局部内部类一种

【例】



* + 1. **匿名内部类**

定义：没有类名的局部内部类（在方法内做接口的实现）

匿名内部类缺点：

1、太复杂，太乱，可读性差

2、匿名类没有类名，不能重复使用。

【例】

MyMath mm = new MyMath();

// mm.mySum(new Compute01(),1,"-",9); // 不使用匿名内部类实现接口时调用。

// Compute01是实现接口的实现类。

// 将new Compute01()替换为匿名内部类实现

/\*

new Compute() {

@Override

public int sum(int a, int b) {

return a+b;

}

};

\*/

// 匿名内部类new Compute() {}

mm.mySum(new Compute() { // mySum是一个计算方法，有4个参数

public int sum(int a, int b) {

return a \* b;

}

}, 1,"\*", 2);

在mysum()方法内作为变量。此时Compute接口被new为一个变量

<存放着对象内存地址></>。

注：实际上是直接在方法内做了接口的实现。

// Compute接口

interface Compute{

int sum(int a, int b);

}

* 1. **数组**

1、java中数组是一种引用数据类型。其实际是一种简单的数据结构。

2、数组既可以存储基本数据类型的数据，也可以存储引用数据类型的数据（包括类引用变量）

3、数组是一个引用数据类型。<数组对象存储在堆内存>

4、若数组内存储的是引用数据类型数据，实际上存储的是引用对象的内存地址。

5、数组一旦创建，在Java中规定，长度不可变

6、所有数组对象都有length**属性**（Java自带），用来获取数组元素个数。 // 注意是属性

7、java数组要求数组中的元素类型统一。 eg:int[]类型数组只能存储int类型的数据

8、数组中元素存储地址是连续的。并且第一个元素的内存地址就是整个数组的内存地址。

9、数组元素索引顺序从0开始依次加1，尾部位置(length - 1) // 注意java中没有-1位置

10、数组优点：

->每一个元素的内存地址是连续的。

->每一个元素的类型相同，所以每个元素占用的空间大小一样

->数组引用地址就是数组第一个元素的内存地址

->根据以上几点，内存可以直接计算出其他元素的内存地址，直接通过内存地址可以定位到数 组中的每个元素，效率很高。例如：int[]类型的数组，每个元素的数据类型也是int类型。而 int类型为4个bit，所以内存可以直接定位其他元素。

11、数组缺点：

->由于数组中每个元素是连续的，所以对数组进行删除和增加操作效率较低（除最后一个元素）。

原因：在删除与增加操作时，会涉及到后面元素与前面元素连续问题。（元素向前向后位移操作）

->数组存储量不能过大。

原因：在内存中很难找到一块非常大的连续内存空间。

* + 1. **一维数组**
* 数组声明与初始化

1、声明一维数组

数据类型[] 数组名; // eg：int[] array\_1; String[] array\_2; User[] array\_3;

2、初始化：

静态初始化：

数据类型[] 数组名 = {元素值1,元素值2,……};

// eg：int[] array\_1 = {1,2,3,4,5,6};

动态初始化：

数据类型[] 数组名 = new数据类型[元素个数];

// eg：int[] array\_1 = int[5];

注：默认数组初始化每个元素是0,null,false。

* 访问数组元素

语法：

数组名[位置]

* 遍历

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

} // length是属性，不是方法

* 修改数组元素

数组名[位置] = 值;

* 数组排序

导入：import java.util.Arrays;

方法：Arrays.sort(数组名); // 数值型数组

#### 数组转集合

Integer[] ars = {1,2,0,0,2,3,0,1};

List<Integer> list = Arrays.asList(ars);

注意：asList()返回值使用List接收，数组类型与List存储元素类型为Integer。

* + 1. **数组扩容**

定义后的数组不能修改其容量，怎样对其扩充容量？

方法：新建一个大容量数组，将小容量数组中的元素拷贝到大数组中。

（效率较低，尽量减少扩容操作）

拷贝方法：

System.arraycopy(参数1, 参数2, 参数3, 参数4, 参数5);

方法输入参数解析：

参数1：原数组。<原数组名src></>

参数2：元数组开始位置。<从src数组的第0位开始></>

参数3：目标数组。<目标数组名dest></>

参数4：目标数组开始位置。<从目标数组的第3位开始填充></>

参数5：拷贝个数。<总共拷贝2个元素></>

* + 1. **二维数组**

一维数组的每个元素也是一维数组

* 初始化

1、静态初始化

语法：

数据类型[][] 数组名 = {{},{},……};

eg：int[][] array01 = {{1,2,3},{4,5},{6,7,8},{0,9}};

2、动态初始化

语法：

数据类型[][] 数组名 = new 数据类型[][];

eg：int[][] array02 = new int[2][3];

* 元素访问

int[] array\_1 = array01[0];

int a0 = array\_1[0];

// 合并

int a1 = array01[0][0];

// 第一个0表示二维数组第一个元素。第二个0表示二维数组第一个元素的第一个元素位置

int a2 = array01[0][array01[0].length-1];

* 修改数组元素

array01[3][0] = 9;

array01[3][1] = 0;

* 遍历

for (int i = 0; i < array01.length; i++) {

for (int j = 0; j <array01[i].length; j++) {

}

}

* 1. **String类**

1、字符串存储位置：

在jdk中双引号引用的字符串是直接存储在方法区内存中的“字符串常量池”。

2、声明并初始化

1> String 变量名 = " 字符串 "; <实际创建一个对象></>

2> String 变量名 = new String("字符串"); <实际创建两个对象></>

当使用1>初始化：字符串直接存储在方法区的常量池中，常量池的字符串地址直接给String类型的局部变量。

当使用2>初始化：字符串依然保存在方法区的常量池中，但是常量池中的字符串地址没有直接给局部变量，而是在堆内存中创建了一个对象，

字符串地址在对象中保存，对象的地址在局部变量中保存。（变量是实例变量时，声明的实例变量在堆内存中存储。）

3、为什么String类型是引用类型，但在比较字符串相等时可以使用"=="？（一般还是使用equals()）

原因：String类型字符串直接保存在常量池中，使用1>方法初始化时，在堆内存中不创建对象，局部变量直接引用常量池内字符串地址。所以，不同的局部变量引用相同字符串时，内部存储地址不变（与常用基本数据类型比较时原理相同）

eg:

String var\_1 = "abc";

String var\_2 = "abc";

if(var\_1 == var\_2){

true; // 输出true

}false;

注意：

String var\_1 = new String("abc");

字符串“abc”是一个对象，可以当做引用使用。（常量池中是一个字符串对象）

5、String构造方法

->数字构造字符串

byte[] byte01 = {1,2,3,4,5};

String s = new String(byte01); // 输出：“12345”

String s = new String(byte01,2,2); // 从第三个元素开始，步长是2；输出：“34”

->字符构造字符串

char[] chr01 = {'a','b','c','d','e'};

String s = new String(chr01); // 输出：“abcde”

String s = new String(chr01,2,2); // 从第三个元素开始，步长是2；输出：“cd”

* + 1. **String类常用方法**
* chaAt(位置值) // 返回指定索引处的字符

【例】 String var\_01 = new String("abcdefg");

String var\_02 = "higklmn";

// chaAt() // 返回指定索引处的字符

System.out.println("1、字符串var\_01索引1处的字符：" + var\_01.charAt(1) + "\n 或：" + "abcdef".charAt(1));

* contains() // 判断字符串是否包含某字符串

【例】 System.out.println("2、字符串var\_01中是否包含“cde”字符串：" + var\_01.contains("cde"));

* endsWith() // 判断字符串是否以某字符串结尾

【例】 System.out.println("3、字符串是否以'f'结尾：" + var\_01.endsWith("f"));

* equals() // 判断两个字符串是否相等

【例】 System.out.println("4、判断两个字符串是否相等：" + "adadasa".equals("adsd"));

* getBytes() //将字符串对象转换为字节数组（unicode码）

【例】 System.out.println("5、字符串转换为字节数组");

byte[] bytes = "ada".getBytes(); // 注意：byte类型

* indexOf() // 判断某个字符串在当前字符串中第一次出现处的索引

【例】 System.out.println("6、判断字符串中指定字符串的首次出现索引位置：" + "fcjavaajavaq".indexOf("java"));

* isEmpty() // 判断某个字符串是否为空。

注意：不是null。也可以用"字符串".length() =0

【例】 System.out.println("7、判断字符串是否为空：\n" + " ->为空时：" + "".isEmpty() + "\n ->不为空时：" + "a".isEmpty());

返回值：为空时：true；不为空时：false

* length() // 字符串长度

注：数组的是属性length，字符串中的是方法length();

【例】 System.out.println("8、字符串长度：" + "adas".length());

* lastIndexOf() // 判断某子字符串在字符串中最后一次出现的首字符索引

【例】 System.out.println("9、判断字符串中指定字符串的最后一次出现索引位置：" + "ajavawajavac".lastIndexOf("java"));

* replace()/replaceAll() // 替换

// replaceAll() 替换字符串所有匹配给定的正则表达式的子字符串

【例】 "http://www.baidu.com".replace("http","https")；

* split() // 拆分字符串成一个字符串类型数组

【例】 String[] str01 = "1=2=3=4".split("=");

System.out.println("11、拆分字符串成一个字符串数组：" + str01[0]);

* startsWith() // 判断字符串是否以某子字符串开始

【例】 "javaawdawdawd".startsWith("java")

* substring() // 截取字符串

【例】 System.out.println("13、截取字符串：" + "quliujie".substring(2,5));

// 参数5不是步长，是结束索引（左闭右开） // 输出：liu

* toCharArray() // 字符串转换成char[]类型数组

【例】 char[] chr\_01 = "沙钢集团！".toCharArray();

System.out.println("14、字符串转换成char[]类型：" + chr\_01[0]);

* toLowerCase() // 字符串转化为小写

【例】 System.out.println("15、字符串转化为小写：" + "ABCDEFG".toLowerCase());

* toUpperCase() // 字符串转化为大写

【例】 System.out.println("16、字符串转化为小写：" + "abc".toUpperCase());

* trim() // 去除字符串前后空白

【例】 System.out.println("17、去除字符串前后空白：" + " my name is qlj！ ".trim());

* valueOf() // 将非字符串转换为字符串

注：String类中唯一的静态方法。不需要创建对象就可以使用

【例】 System.out.println("18、将非字符串转换为字符串：" + String.valueOf(1.23));

* append()/"+" // 字符串拼接

【例】 System.out.println("19、字符串连接：\n" + " 使用\"+\"时-> abc" + "de");

// append()方法具体看stringBuffer类

* compareTo() // 按(ASCII)字典顺序比较两个字符串(前减后)

返回值是整型，它是先比较对应字符的大小(ASCII码顺序)，如果第一个字符和参数的第一个字符不等，结束比较，返回他们之间的长度差值，如果第一个字符和参数的第一个字符相等，则以第二个字符和参数的第二个字符做比较，以此类推,直至比较的字符或被比较的字符有一方结束。

如果两个字符串不一样长，可对应字符又完全一样，则返回两个字符串的长度差值。

【例】 System.out.println("20、字符串比较：" + "abc".compareTo("abh"));

// 返回：-5

* + 1. **StringBuffer（字符串拼接）**

**为什么不推荐使用”+”进行连接字符串？**

【原因】

"+"字符串连接符在连接字符串时会在方法区常量池创建多个字符串对象，连接时还会创建新对象保存连接的新字符串

作用：保存追加的字符串<容量满了后自动扩容>

底层是创建一个初始容量为16的char[]数据。（字符串缓冲区）

例：

StringBuffer str01 = new StringBuffer(); // 创建StringBuffer对象

使用append()函数追加字符

注：任意类型的数据都可以追加

例： str01.append("a");

str01.append("bc");

str01.append(123);

str01.append(true);

1、如何优化StringBuffer的性能？

->在声明StringBuffer对象时，尽可能给对象初始化一个容量

->最好减少底层数组的扩容次数。尽可能预估容量使用的大小。

2、为什么String类型数据是不可变的？

String类中有一个value[]数组，这个数组被final修饰，数组的长度创建后不可变。并且被final修饰的引用一旦指向某个对象后，不可在指向其他对象。所以String不可变。

3、为什么StringBuffer与StringBuilder是可变的？

StringBuffer与StringBuilder底层实际是一个数组，这个数组没有被final修饰，并且初始化容量为16，当容量不够时，数组使用arraycopy()进行扩容，所以可以进行字符串拼接操作。

* + 1. **StringBuilder**

StringBuilder使用方式与StringBuffer类似，创建StringBuilder对象，然后使用append()函数追加字符即可。

StringBuffer与StringBuilder的区别？

-> StringBuffer类中的方法有synchronized关键字修饰。表示StringBuffer在多线程环境下运行是安全的。

-> StringBuilder类中没有synchronized关键字修饰。表示StringBuilder在多线程环境下运行是不安全的。

即：

-> StringBuffer是线程安全的

-> StringBuilder是非线程安全的

* 1. **包装类**

包装类：Java是一个面向对象的编程语言，但是8种基本数据类型却不是面向对象的，为了弥补这一不足，设计了8种基本数据类型的包装类。

为什么使用包装类？

原因：当类中的某一方法接收一个Object类型的形式参数，则在调用该方法时，实际参数也必须是Object类型的数据。但是8种基本数据类型并不是对象，不继承Object类。因此创建包装类这种引用数据类型

作用：

->编码过程中只接受对象的情况下，比如List中只存入对象，不能存基本数据类型；

->方便类型之间的转换，比如String和int之间的转换可以通过int包装类Integer来实现。

* + 1. **包装类的类别**

|  |  |
| --- | --- |
| **基本数据类型** | **引用数据类型** |
| byte | java.lang.Byte (父类Number) |
| short | java.lang.Short (父类Number) |
| int | java.lang.Integer (父类Number) |
| long | java.lang.Long (父类Number) |
| float | java.lang.Float (父类Number) |
| double | java.lang.Double (父类Number) |
| boolean | java.lang.Boolean (父类Object) |
| char | java.lang.Character (父类Object) |

注：

1、除了Boolean与Character类的父类是Object之外，其他6种包装类的父类都是Number类。

<Number是一个抽象类不能实例化>

* + 1. **类型转换**

1、Number抽象类中有6个方法，分别是intValue()、doubleValue()、floatValue()、byteValue()、shortValue()、longValue()。

方法作用：将引用数据类型的包装类（6种数字包装类）转换为基本数字类型。

2、装箱：将基本数据类型转换为包装类（通过创建包装类并初始化转换）

拆箱：将引用数据类型转化为基本数据类型（通过Number内部方法进行转换）

<注：对应的转换为基本数据类型>

3、自动装箱：Integer int02 = 10; // 直接将基本数据类型赋给包装类

自动拆箱：int int03 = int02; // 直接将包装类值赋给基本数据类型变量

* + 1. **构造方法**

关于Integer的构造方法

例：

Integer(int);

Integer(String);

* + 1. **比较包装类类型的数据大小**

注意：比较引用数据类型的数据是否相等使用equals()方法。

【例】下面例子中使用“==“比较。

Integer x = 127;

Integer y = 127;

System.out.println(x == y); // true（返回true是因为内存地址一样。）

Integer a = 128;

Integer b = 128;

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

**返回结果不同的原因？**

-> java为了提高执行效率，将[-128,127]之间所有的包装对象提前创建好，放到了一个方法区的“整数型常量池”中，目的是只要用这个区间的数据直接从整数型常量池中取。

* + 1. **Integer类中常用的方法**

// 初始化

Integer num01 = new Integer(123);

String str = "1232";

* intValue() // 拆箱

int n01 = num01.intValue(); // 返回int类型数字

* paserInt() // String类型数字转换int

注：

1、静态方法:类名.调用；

2、只能转换String型数字，文字、字母、符号等都不支持转换，转换其他类型会出现以下异常（NumberFormatException异常）

【例】

int n02 = Integer.parseInt(str);

System.out.println(n02);

System.out.println(Double.parseDouble("123"));

* valueOf() // 将int/String类型转换为Integer

1、int --> Integer

Integer num02 = Integer.valueOf(100);

2、String --> Integer

Integer num03 = Integer.valueOf("100");

* + 1. **String,int,Integer互相转换**
* int --> String

String str01 = String.valueOf(134);

System.out.println("int --> String：" + str01);

* String --> int

int int01 = Integer.parseInt("123");

System.out.println("String --> int：" + int01);

* String --> Integer

Integer num01 = Integer.valueOf("123");

System.out.println("String --> Integer：" + num01);

* Integer --> String

String str02 = String.valueOf(num01);

System.out.println("Integer --> String：" + str02);

* int --> Integer

Integer num02 = 123; // 自动装箱

System.out.println("自动装箱int --> Integer：" + num02);

Integer num03 = Integer.valueOf(123);

System.out.println("手动装箱int --> Integer：" + num03);

* Integer --> int

int int02 = num01; // 自动拆箱

System.out.println("自动拆箱Integer --> int：" + int02);

int int03 = num01.intValue(); // 手动拆箱

System.out.println("手动拆箱Integer --> int：" + int03);

* 1. **日期**

创建日期对象

Date date01 = new Date();<系统时间>

// Date中重写了toString()方法，直接输出date01即可。

* + 1. **SimpleDateFormat类 // 专门用于日期格式化的类**

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

* Date --> String (format)

SimpleDateFormat format01 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

// 指定格式对象

String s\_date = format01.format(date01);

// 将日期修改为创建的日期格式，返回String类型日期

* String --> Date (parse)

String s\_date02 = "2022-03-04 23:23:23 212";

SimpleDateFormat format02 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

注：必须与字符串的日期类型一致

Date date02 = format02.parse(s\_date02);

* + 1. **时间戳**

时间戳是从1970-1-1 00:00:00 000 到目前时间的总毫秒数

简单总结System类常用相关属性方法

1、System.out // out是System类的静态变量

2、System.out.println() // println()方法不是System类中的方法，是PrintStream类的方法

3、System.gc() // 建议启动垃圾回收器

4、System.currentTimeMillis() // 获取1970-1-1 00:00:00 000 到目前时间的总毫秒数

5、System.exit(0) // 退出JVM

【例】

// 创建一个时间戳对象

long time01 = System.currentTimeMillis();

System.out.println("1970-1-1 00:00:00 000 到目前时间的总毫秒数：" + time01);

// 统计程序运行时长

long start = System.currentTimeMillis();

eg(); // eg()方法是一个程序块（功能：打印1-10000数字）

long end = System.currentTimeMillis();

System.out.println("方法eg()执行耗时：" + (end - start) + "毫秒");

* 将毫秒数据（时间戳）转换为指定时间格式

Date date01 = new Date(213543213531); // long型时间戳

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

String strTime = sdf.format(date01);

System.out.println(strTime);

* 1. **异常处理机制**
     1. **定义**

在程序运行过程中出现的错误，称为异常。异常就是程序运行过程中出现了不正常现象导致程序的中断。

* + 1. **作用**

增强程序的健壮性，为程序调试提供了很大的方便，并能保证程序在出现异常情况下仍然可以继续运行下去，从而提高了程序的健壮性，

* + 1. **异常在java中的存在形式**

异常是以类的方式存在的，每个异常类都可以创建异常对象。

（在程序执行过程中，若程序某行出现异常，JVM会创建一个异常对象，并且JVM将new出来的对象抛出，打印到输出信息中）

* + 1. **异常类继承结构（分类）**

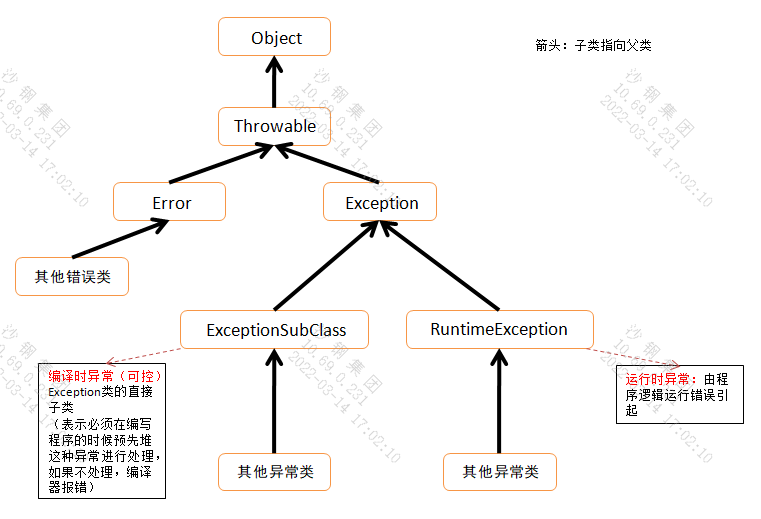


图1异常类UML图

* + 1. **处理异常方法**

1、在方法声明的位置上，使用throws关键字（抛给上一级<调用者>）

throws后面写异常的类名，也可写其异常的父类名，或多个异常类名。

throw，手动抛出异常！语法：throw new 异常名(“异常信息”);（抛出异常对象）

注：手动抛出异常后，该方法的方法名后也需要用throws抛出异常。

2、使用try……catch进行异常捕捉（解决异常）

注：

（1）只要异常没有捕捉，采用上报方式，此方法的后续代码不执行

try语句块中某一行出现异常，该行后面的代码不执行；try……catch捕捉异常后，后续代码可以执行。

（2）catch可以写多个，但必须从小到大，前面的catch不能包括下面的catch异常内容。

（3）jdk8以上，catch中可以写“|”，表示或。

怎样选择两种方法：

一般只有希望调用者处理异常，才使用throws。

* + 1. **异常对象常用方法**

1、获取异常简单的描述信息（实际是构造方法的参数信息，即，实参内容）

String msg = exception对象.getMessage();

2、打印异常追踪的堆栈信息（常用于catch内使用，查看异常追踪信息）

exception对象.printStackTrace();

注意：

在方法重写（覆盖）时

子类重写的方法不能比父类重写前的方法抛出更多的异常，可以更少！

例：

父类抛出Exception异常，子类可以抛出Exception异常或Exception类的子类异常；

父类抛出ArithmeticException异常，子类就不可以抛出Exception异常，只能抛出ArithmeticException异常或ArithmeticException类的子类异常。

* 1. **集合**
     1. **基础**
* 集合是java中提供的一种容器，可以用来存储多个数据。实际上是一个java类库，其中提供了已经实现的数据结构。
* 集合不能直接存储基本数据类型，集合也不能直接存储对象内容，集合中只能存储java对象的内存地址（引用数据类型）。
* 基本数据类型数据存储时，会自动装箱，转换成一个引用数据类型的数据。

例:Collection c = new ArrayList();

c.add(100); // 自动装箱 int -> Integer (储存对象的内存地址)

* 在Java中，每一个不同的集合，底层对应不同的数据结构，往集合中存储元素，等于将数据放到不同的数据结构中。

常见数据结构：数组、链表、二叉树、哈希表等

例：new ArrayList() 创建一个数组集合

new LinkedList() 创建一个链表集合

new TreeSet() 创建一个二叉树集合

* 集合包位置：java.util.

所有的集合类和集合接口都在java.util.内

* java的集合分类

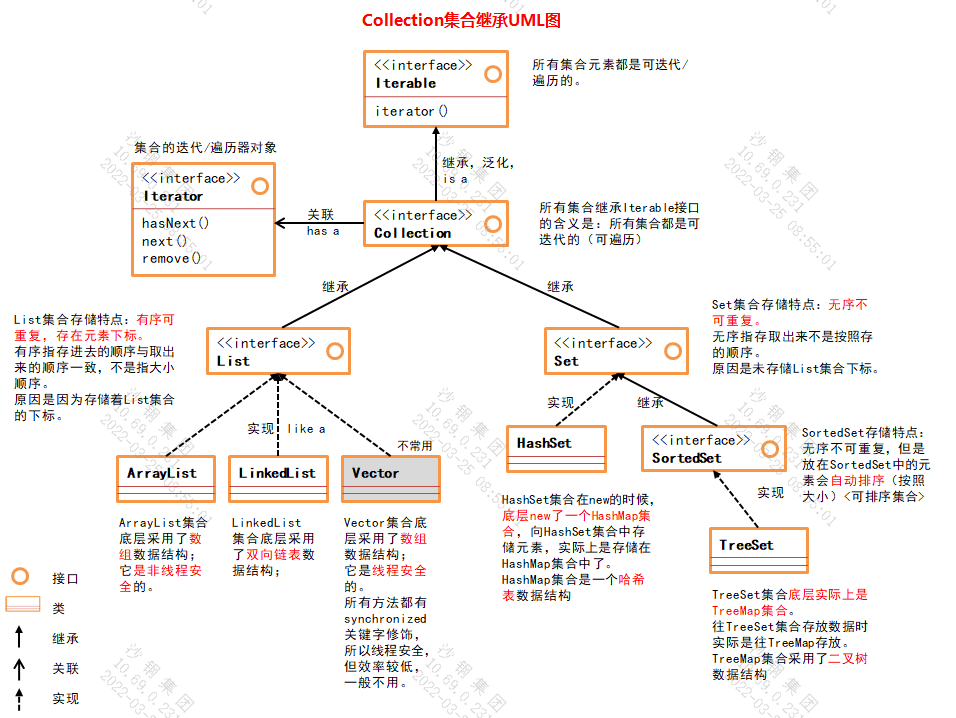
->单值方式存储元素

这一类集合的超级父接口：java.util.Collection

->以键值对方式存储

这一类集合的超级父接口：java.util.Map

* + 1. **Collection集合的继承结构图**



* + 1. **Collection集合的特点**

主要分为两种：

* List（ArrayList、LinkedList、Vector） // 有序可重复，存在元素下标。

1. ArrayList (数组，非线程安全)
2. LinkedList (双向链表)
3. Vector (数组，线程安全)

* Set（HashSet、TreeSet） // 无序不可重复。

1. HashSet (哈希表，HashMap集合key部分)
2. TreeSet (二叉树，TreeMap集合key部分，key自动排序)

注意：有序指存取顺序有序，并不是指元素的排序

* + 1. **Collection常用方法**
* boolean add(Object e) // 向集合中添加元素

例：

// 创建集合对象  
// Collection c = new Collection(); // 接口不能直接实例化  
// 多态创建对象  
Collection c = new ArrayList();

c.add(100);

c.add("abc");

c.add(true);

c.add(new Object());

* int size() // 获取集合中的元素个数

例：

int c\_size = c.size(); // 集合元素个数

System.out.println(c\_size);

* void clear() // 清空集合

例：

c.clear();

* boolean contains(Object o) // 判断当前集合是否包含元素o

①包含返回true,不包含返回false

②判断集合是否包含元素，底层需要比较元素是否相等。（涉及equals()使用）

注：集合的底层存储的是对象的内存地址，在判断时，底层使用indexOf()中的equals()进行比较。

若储存在集合中的元素是String类型，而String类型java内部已经重写了equals()方法，所以直接判断内容是否相等。

若集合中储存元素是自定义的引用类型，则必须重写equals()方法，contains()方法才可以比较对象内容，否则只能比较对象的内存地址

**contains()方法自定义的对象需要重写equals()方法。**

例：

c.contains("abc")

* boolean remove() // 删除集合中指定元素

删除指定元素底层也使用了equals()方法，比较元素是否相等

remove()方法与contains()方法底层都调用了equals方法，所以集合中存储的是自定义的引用类型，则需要重写equals()方法，才能正常删除元素。

例：

c.remove("aaaa");

* boolean isEmpty() // 判断集合是否为空

例：

c.isEmpty() // 不为空，返回false

* Object[] toArray() // 集合转换为数组（Object[] 类型）

注：转换为Object类型数组后，没有重写toString()的数据也可以保存，并且可以存储任意对象（不会违背数组只能保存同类型的数据规则）

例：

Object[] array\_c = c.toArray() // 可遍历查看数组

* + 1. **Collection中的迭代器Iterator（遍历）**

1. Iterator是为了实现对Java容器(Collection)进行遍历功能的接口
2. itertor()是Iterable 接口的一个Iterator返回类型的抽象方法，而Collection接口继承了Iterable接口，所以Collection接口也有了itertor()抽象方法。

源码：Iterator<E> iterator();

1. 迭代器迭代完成后就不能用了，要遍历其他集合需要重新创建迭代器对象。

* 迭代器使用原理

// 创建迭代器对象

Iterator<Object> it2 = c.iterator();

->在使用iterator()时，实际是具体的容器对象(例：ArrayList)调用了iterator()方法。

（原因：ArrayList集合实现了collection，ArrayList类中也有iterator()方法）

-> iterator()方法内部new了一个Itr对象，该对象实现Iterator接口

->所以c.iterator()实际就是创建了一个Iterator迭代器对象。

->实现了Iterator接口后，Collection容器中的所有元素相当于一个线性表(List)，而iterator中存在指针，开始时指针指向第一个元素之前。

* Iterator接口中的方法

hasNext() // 判断指针指向是否还存在下一个元素，存在返回true，否则返回false

next() // 返回迭代器的下一个元素，并更新迭代器的状态(向后移动一位)

remove() // 删除迭代器返回的元素（一般不用，可以使用集合的remove方法）

* 迭代器遍历Collection集合

注意：只能遍历Collection集合

1. 创建迭代器对象

Iterator it1 = c.iterator();

1. 使用hasnext()判断是否还有下一个元素

it2.hasNext(); // 有元素返回true

③使用next()返回迭代器指向的下一个元素，同时更新迭代器状态

Object c1 = it1.next();

例：

Iterator<Object> it2 = c.iterator(); // 创建迭代器对象

while (it2.hasNext()){

System.out.println(it2.next());

}

* 迭代器remove()与集合remove()方法区别

作用：两个remove()方法都是为了删除集合元素的

区别：两者的区别主要是使用范围不同。迭代器对象的remove()方法只能在迭代器内使用。 使用集合对象的remove()方法删除所有元素时（将方法写在遍历循环中），运行会出现异常(java.util.ConcurrentModificationException)。

* + 1. **List接口**

特点：有序可重复，存在元素下标（存取有序）

* List特有方法

注：不能使用Collection作为引用类型，使用多态时，若用Collection父类进行引用，则不能访问子类特有的方法。

1. void **add**(int index, Object element)

作用：在指定索引位置添加元素，原位置的元素向后移动，无索引参数时向集合末尾添加元素

【例】：myList.add("bbb");

1. Object **get**(int index)

作用：根据下标获取元素。（可以使用get()方法遍历集合）

【例】：Object obj1 = myList.get(0);

1. int **indexOf**(Object o)

作用：获取指定元素第一次出现的索引位置

【例】：myList.indexOf("bbb");

1. int **lastIndexOf**(Object o)

作用：获取指定元素最后出现的索引位置

【例】：myList.lastIndexOf("bbb");

1. Object **remove**(int index)

作用：删除集合指定位置/指定元素的值（尽量不使用指定元素删除）

注：若指定元素是数值，则默认删除的是指定索引位置的元素。

【例】：myList.remove("111"); // 此时删除的是字符串类型的数据

1. Object **set**(int index, Object element)

作用：修改指定位置元素

【例】：myList.set(3,111);

* ArrayList

1. ArrayList集合底层是一个Object[]数组类型
2. 默认初始容量为10;（高版本在此处优化：初始化对象时，为一个空数组。当插入第一个元素时，初始化容量就变为了10）
3. 集合扩容：若添加元素超出容量大小，集合会进行自动扩容，扩容大小为原容量的1.5倍
4. ArrayList集合的构造方法

-> new ArrayList(); // 默认容量为10

-> new ArrayList(20); // 指定容量大小

-> new ArrayList(Collection<? extends E> c)

// 将其他类型的集合转化为ArrayList集合

1. 创建集合对象：

// 可以根据不同构造方法创建对象

ArrayList array\_List01 = new ArrayList();

Collection array\_List02 = new ArrayList(); // 多态

1. ArrayList优缺点

ArrayList底层实际是数组结构，所以具备数组的特性

内存地址是连续，根据下标使得检索效率高。

大数据的集合一般不采用ArrayList集合。（找不到较大连续地址的内存）

1. 集合自动扩容

注意：尽量在创建集合对象进行初始化容量大小，减少扩容次数

源码：int newCapacity = oldCapacity + (oldCapacity >> 1);

// oldCapacity：表示扩容前集合容量

// newCapacity：表示扩容后的新容量

// 所以进行一次扩容，相当于对原有的集合容量扩充了1.5倍。

⑧ ArrayList线程安全化

Collections.synchronizedList(myArray);

// 此时再使用myArray对象就是线程安全的

* LinkedList

LinkedList集合底层是双向链表结构

**单向链表数据结构**

结点：分为数据域与指针域（数据域存放节点数据，指针域存放后继结点的地址，相当去指向后继结点）

头结点：链表第一个结点默认是null,可存储链表其他信息

首元结点：首元结点是除去头结点外的第一个结点，存有实际数据。

头指针：永远指向链表第一个结点的位置，用于指明链表的位置（若链表存在头结点，头指针指向头结点，否则指向首元结点）

存储：链表在存储时，是随机在内存中存储的，只是通过指针进行连接所有结点。

优点：随机增删改元素效率较高（增删元素不涉及大量元素作位移）

缺点：查询效率较低，每次查找元素都必须从头结点开始遍历。

**数组与链表的区别**

数组检索数据效率较高；

链表增删改数据效率较高；

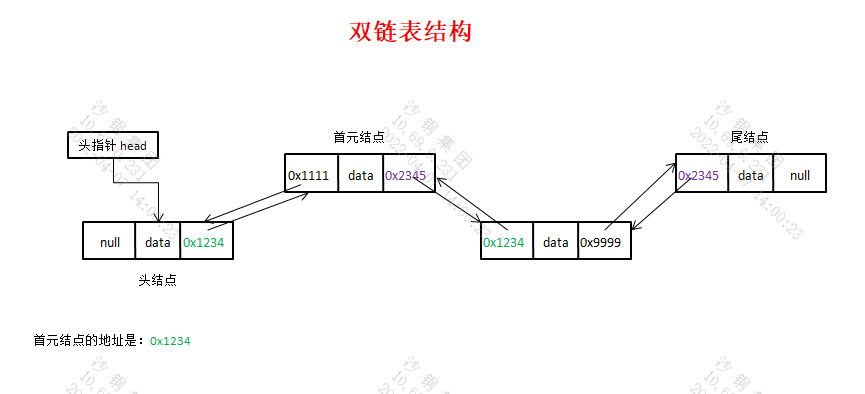
一般添加元素都是在末尾添加，数组结构在末尾添加时，不需要位移元素，因此ArrayList使用较多。

**双向链表结构**

双链表的结点由三部分组成。（前驱指针域、数据域、后继指针域）

前驱指针域：指向当前结点前驱结点的后继指针域

后继指针域：指向当前结点后继结点的前驱指针域



创建LinkedList对象

// LinkedList集合有两个构造方法

List myList01 = new LinkedList(); // 创建对象

LinkedList(Collection<? extends E> c)

// 将其他类型集合转化为LinkedList集合

插入数据

* 在链表后面添加元素<linkLast()>

myList01.add("aaa");

* 在链表内添加元素<linkBefore()>

myList01.add(1,"zzz");

* Vector

① Vector集合底层是数组结构

②初始化容量：10

③ Vector集合是线程安全的

④扩容机制：<默认一倍>

在grow()方法中源码：

newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity)

capacityIncrement：创建对象时Vector构造函数内输入一个实际参数，表示扩容的增量。

oldCapacity：表示原Vector集合容量大小。

解释源码：如果在创建Vector对象时，指定了扩容增量，则每次自动扩容就按照增量进行扩容；否则就将原容量加一倍。

⑤构造函数

public Vector()

public Vector(int initialCapacity) // 指定初始容量

public Vector(int initialCapacity, int capacityIncrement)

// 指定初始容量、扩容增量

* + 1. **Set接口**

特点：无序不可重复。

若添加重复元素，则添加元素失败。

* HashSet

原理：

HashSet集合在new的时候，底层new了一个HashMap集合，向HashSet集合中存储元素，实际上是存储在HashMap集合key部分中了。

HashMap集合是一个哈希表数据结构

* TreeSet

TreeSet并不直接实现Set接口，而是实现了SortedSet接口。

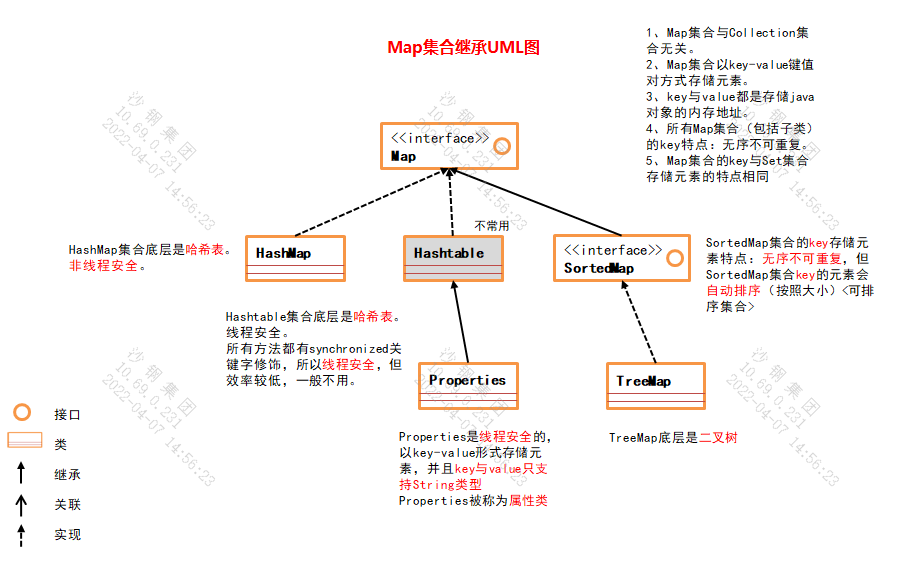
SortedSet接口特点：

无序不可重复，但是放在SortedSet中的元素会自动排序（按照大小）<可排序集合>

原理：

TreeSet集合底层实际上是TreeMap集合。往TreeSet集合存放数据时实际是往TreeMap存放。TreeMap集合采用了二叉树数据结构。

* + 1. **Map集合的继承结构图**



* + 1. **Map集合特点**

1、Map集合与Collection集合无关。

2、Map集合以key-value键值对方式存储元素。

3、key与value存储都是引用数据类型，存储对象的内存地址。

4、所有Map集合（包括子类）的key特点：无序不可重复。

5、Map集合的key与Set集合存储元素的特点相同

* + 1. **Map接口的常用方法：**
* V put(K key, V value); // 向Map集合中添加键值对

// 创建对象

Map<Integer,String> dict01 = new HashMap<>();

dict01.put(1,"qlj");

// key此时自动装箱

* V get(Object key); // 通过key获取value

dict01.get(3);

// 获取key = 3的value值

* int size(); // 获取集合键值对个数

dict01.size();

* boolean isEmpty(); // 判断Map集合中元素个数是否为0

dict01.isEmpty();

* boolean containsKey(Object key); // 判断集合是否包含指定key值

dict01.containsKey(6);

* boolean containsValue(Object value); // 判断集合是否包含指定value值

dict01.containsValue("py");

* V remove(Object key); // 通过key值删除键值对

dict01.remove(2);

// 可以返回删除的value值

* void clear(); // 清空Map集合

dict01.clear ();

* Set<K> keySet(); // 获取Map集合中所有key值

Set<Integer> mySet01 = dict01.keySet();

// 返回Set类型对象

* Collection<V> values(); // 获取Map集合中所有value值

Collection<String> myList = dict01.values();

* Set<Map.Entry<K, V>> entrySet(); // 将Map转换为Set集合

Set<Map.Entry<Integer,String>> mySet02 = dict01.entrySet();

// entrySet是将key值与value值用“=”连接，作为一个Set集合元素

* + 1. **遍历Map集合**

**方式一**：

获取所有的key，通过所有key访问value值

【代码】

// 获取所有key，返回Set集合

Set<Integer> myKey = dict01.keySet();

// 迭代器遍历(实际遍历的是Set集合)

Iterator<Integer> it01 = myKey.iterator();

while (it01.hasNext()){

// 取出Set集合中的key值

Integer key = it01.next();

// 通过取出的key值访问Map集合中的value

String value = dict01.get(key);

// 打印输出

System.out.println(key + ":" + value);

}

// foreach遍历

for (Integer key : myKey

) {

String value = dict01.get(key);

System.out.println(key + ":" + value);

}

// 不能使用for循环遍历（Set集合没有索引下标）

**方式二**：(一般使用这种方式)

Set<Map.Entry<K, V>> entrySet();

// 直接将键值对连接，返回一个Set类型的集合（元素类型：Map.Entry）

// 遍历返回的Set集合

Map.Entry类型：

Map.Entry：Entry是Map接口下的一个内部接口(内部类)

Entry内部接口内部方法：

-> K getKey(); // 获取Map.Entry类型的key值

-> V getValue(); // 获取Map.Entry类型的value值

【代码】

// 使用entrySet()方法将Map集合转换为Set集合

Set<Map.Entry<Integer,String>> mySet = dict01.entrySet();

// 迭代器遍历1

Iterator<Map.Entry<Integer,String>> it02 = mySet.iterator();

while (it02.hasNext()){

System.out.println(it02.next());

}

// 怎样修改key=value这种输出方式？

// 迭代器遍历2

Iterator<Map.Entry<Integer,String>> it03 = mySet.iterator();

while (it03.hasNext()){

Map.Entry<Integer,String> myMapEntry = it03.next();

// 使用Entry类的方法获取Set集合元素的key与value

Integer key = myMapEntry.getKey();

String value = myMapEntry.getValue();

System.out.println(key + ":" + value);

}

// foreach遍历

for ( Map.Entry<Integer,String> mapEntry : mySet

) {

Integer key = mapEntry.getKey();

String value = mapEntry.getValue();

System.out.println(key + "-->" + value);

}

* + 1. **HashMap**
* 使用HashMap的原因

数组：查询数据效率较高（根据下标对应查询），增删数据效率较低（增删元素需要其他元素位移）

链表：随机增删改元素效率较高（增删元素不涉及大量元素作位移）

HashMap：查询数据且随机增删数据的效率都很高。

* 哈希表存储结构

HashMap实现了Map接口，用于存储key-value键值对集合，每个键值对也叫Entry;

存储结构：数组+链表，结合数组与链表的优点。底层使用Entry[]键值对（jdk7使用Entry表示，jdk8使用Node）

Entry[]源码：Entry(int h, K k, V v, Entry<K,V> n) {

value = v; // value值

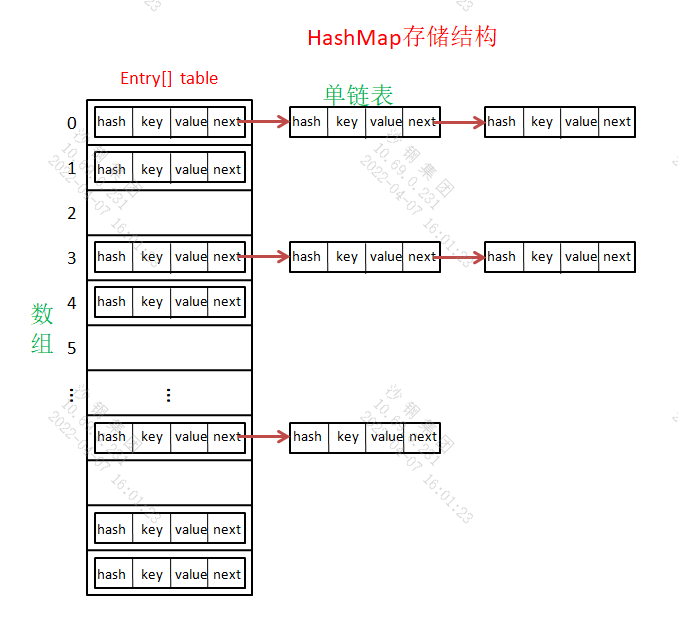
next = n; // 指向下一个结点的指针

key = k; // key值

hash = h; // hash值

}

// 每个结点中有4个参数：key、value、next、hash



* 哈希表（散列表）

根据关键码值(key、value)直接进行访问的数据结构。

也就是说：它通过把关键码值映射（哈希/散列函数）到表中的一个位置来访问记录，加快查询速度。

* 哈希函数

将任意长度的输入值通过函数映射关系变成固定长度的值（数值）。

（固定长度不是说每个转化值都一样长）

这个固定长度的值就是散列值（哈希值）。

哈希函数具有数学函数的映射特点。每个x只有唯一确定的y与之相对应。

* 哈希值(哈希码)

通过哈希函数准换后的值。

特点：

①哈希值的空间（区间）往往小于原集合数据空间；（压缩）

②不同的key值通过哈希函数得到的哈希值可能相同；

③相同的key值映射的哈希值一定相同

注：HashMap中hash值通过hash()方法转换为数组下标。

源码:<jdk8>

static final int hash(Object key){

int h;

return (key == null) ? 0 : (h = key.hashcode()) ^ (h >>> 16);

}

解释：

① h是hashcode计算得到的哈希码，然后使用hash()内算法得到数组下标。

② h >>> 16：经哈希码进行无符号右移16位；（无符号右移：无论最高位是0还是1，左侧被移空的高位都填入0）

③ (h = key.hashcode()) ^ (h >>> 16)：按位异或运算

* hashcode()

hashcode()方法是Object类中的方法，这个方法其实是实现哈希算法得到的(未重写时，使用native修饰)。native：说明被修饰方法是使用C/C++实现的

作用：用于返回任意key字符串/对象的哈希码值。

重写hashcode()：对象调用方法，返回一个定长的哈希码值。

重写hashcode()：对象根据hashcode方法内部哈希算法进行计算得到哈希值。

注：String、Integer、Boolean等类型底层已经重写hashcode()方法。

其他自定义对象一般需要重写，不重写会调用Object中的hashcode

* 重写hashcode()

为什么要重写hashcode()？

对象的equals()方法被重写时，相同内容的判断返回true;若没重写equals()方法，则底层使用"=="只能判断对象地址是否相等。

若未重写hashcode()方法，则可能出现相同值（equals判断）不同hashcode值现象，显然是不合理的现象。

key值相同时hashcode必须是相等的

【例】

Tem tem2 = new Tem(1,2);

Tem tem3 = new Tem(1,2);

tem3.hashCode(); // 30178960 重写hashcode:// 994

tem2.hashCode(); // 411640028 重写hashcode:// 994

tem2.equals(tem3); // true

**放在HashMap集合key部分的对象与HashSet对象都必须重写hashcode()方法与equals()方法。**

* 哈希冲突

产生原因：

哈希算法将大范围的集合数据压缩到了小范围，必定会有最终的哈希值是相同的（作为数组下标的哈希值），但是真正的数组下标只对应一个value，此时出现相同的数组下标就是哈希冲突。

怎样避免哈希冲突：

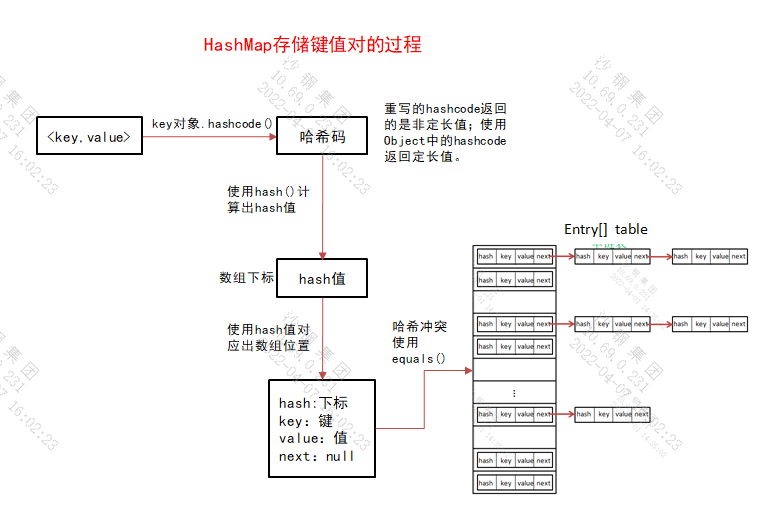
->开放地址法：出现冲突后按照一定的算法查找一个空数组位存放冲突数据。

->拉链法：将具有相同哈希地址的记录链成一个单链表。（java一般使用拉链方法）

->公共溢出区法：将所有产生冲突的数据存放到一个溢出表内。

jdk8后:HashMap冲突数据有8个以上就会使用红黑树结构存储冲突数据；当红黑树结点数小于6时，又会变成链表结构。

* HashMap实现原理



**put**():添加元素

①将键值对封装到Node对象中（Node类实现了Map.Entry<K,V>接口）

②底层调用key.hashcode()方法得到哈希码

③将hash码通过hash()方法转换成数组的下标

④当得到的索引下标处没有value，将封装的Node添加到当前下标位置；

当索引位置处已经存在数据时（哈希冲突以链表解决），此时使用equals()方法与链表每个节点的key比较进行判断：

->若比较key值相等时，说明当前添加的键值对key值重复，新结点覆盖之前的结点；

->若比较key值不相等，说明集合中没有当前键值对，将新结点直接添加到链表尾部。

**get**():获取元素

get的原理与set基本类似

如果key转化的数组下标位置没有任何数据，则返回null.

如果key转化的数组下标位置存在链表，则依然使用equals进行比较key值，没有相等的key值就返回null，否则返回实际value值。

* key对象与重写后hashcode的关系（类比数学函数）

->两个对象相等，hashcode一定相等：x值相等,y值一定相等

->两个对象不等，hashcode不一定不等：x不等，y值可能相等

-> hashcode相等，两个对象不一定相等：y值相等，x值不一定相等

-> hashcode不等，两个对象一定不等：y值不等，x值一定不等

* 为什么要使用hash()方法对应出下标？

hashcode已经将key对象转换为哈希数值，但是hashcode()返回的哈希值值域较大，使得put()添加数据后形成的哈希表空间较大，近似为一个一维数组结构，链表特性没有发挥出来。

当然也不可以通过哈希算法转换后，形成了数据链较长、数组元素较少的哈希表。

<散列分布不均匀>

* HashMap集合扩容机制

HashMap集合创建初始化容量为16，默认加载因子是0.75。

加载因子：当底层集合容量填充至75%时，集合开始扩容

官方推荐：HashMap集合初始化容量为2的幂次方（原因：为了达到散列均匀，提高集合存取效率）

源码：static final int DEFAULT\_INITIAL\_CAPACITY = 16;

扩容：每次扩容以2的倍数进行扩容

* HashMap集合可以存储空指针对象

HashMap集合中key与value都可以为null，但key只能有一个null。

为什么为null，没有引起空指针异常？

底层进行判断return (key == null) ? 0 : (h = key.hashcode()) ^ (h >>> 16);

返回0，规避了空指针异常。

// myMap.put(null,null);

* 创建并操作对象

// 创建对象

Map<Integer,String> myMap = new HashMap<>();

// 添加元素

myMap.put(4,"ddd");

myMap.put(null, "ddd")

myMap.put(null,null); // 覆盖"ddd"

// 注：添加第三条元素时，key=null，但是key值不能重复，所以会覆盖前面的值。

// 获取集合数据

myMap.get(2); // 用key获取value

* + 1. **HashTable**

1、HashTable的key与value不可以为null，HashMap可以为null。

2、HashTable与HashMap一样底层也是哈希表

3、HashTable扩容是原容量的2倍再加1。

4、HashTable方法线程安全的，但是一般不用

* 创建并操作对象

// 创建对象

Map<String,String> myTable = new Hashtable<>();

// 添加元素

myTable.put("1","1");

// myTable.put(null,"1"); // java.lang.NullPointerException空指针异常

* + 1. **Properties**

1、Properties是一个Map集合，继承了HashTable，它的key与value都是String类型

2、Properties被称为属性类对象

3、Properties主要方法：

->setProperty():添加元素 // 底层调用HashTable中的put方法

->getProperty():获取元素 // 底层调用HashTable中的set方法

* 创建并操作对象

// 创建Properties对象

Properties mypro = new Properties();

// 添加元素

mypro.setProperty("111","111");

mypro.setProperty("222","222");

mypro.setProperty("333","333");

// 获取元素

mypro.getProperty("111");

* + 1. **TreeMap**

1、TreeSet集合底层实际是一个TreeMap

2、TreeMap集合底层是一个二叉树(左小右大)

二叉树遍历方式

->前序遍历（根左右）

->中序遍历（左根右）

->后序遍历（左右根）

注：前、中、后说的是根的位置

->Iterator迭代器遍历TreeSet采用的是中序遍历

3、放在TreeSet集合中的元素，等同于放到TreeMap集合key部分了

4、TreeSet集合中的元素(key)：无序不可重复，但是可以按照元素的大小顺序自动排序

5、TreeSet对自定义类型的数据不能排序。（需要自己写排序器comparator）

6、添加元素不可以为null。

* 排序比较器

场景：自定义类作为TreeMap集合的key值时，不会自己进行排序，所以在添加元素时就会报错。此时需要对key写一个比较器。

注意：

1、比较器底层是通过对象的某个int属性进行比较的

2、要想遍历集合输出自定义对象的内容，必须重写toString()方法

3、比较器写好后创建对象时需要传入比较器作为参数。

用于传入比较器的构造方法：

public TreeMap(Comparator<? super K> comparator) {

this.comparator = comparator;

}

创建对象：

TreeMap<Teacher,String> myTeacher =

new TreeMap<>(new teacherComparator());

【例】

// 创建的比较器实现Comparator接口

class teacherComparator implements Comparator<Teacher>{

@Override

public int compare(Teacher o1, Teacher o2) {

return o1.age - o2.age;

}

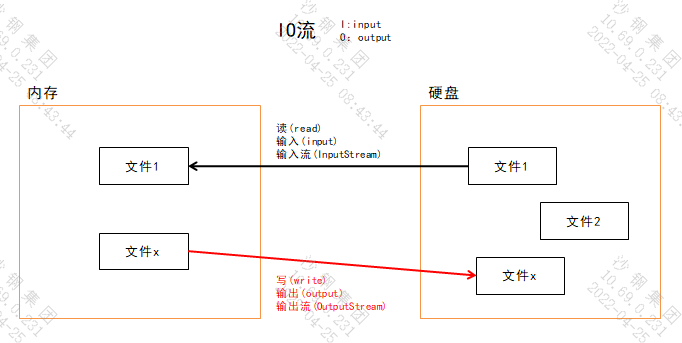
}

* 1. **IO流**
     1. **概述**
* io流概念

什么是IO流？

I：input; O:output

通过IO完成硬盘文件的读与写。



* 分类

①按照流的方向：

输入流：往内存中去，叫做输入/读

输出流：从内存中出，叫做输出/写

②按照读取数据方式：

字节流：按照字节的方式读取数据，一次读取1个字节byte，等同于一次读取8个二进制，这种流不管什么类型的数据都可以读取。

字符流：按照字符的方式读取数据，一次读取一个字符，这种流是为了方便文本文件存在的，只能读取纯文本文件，连word文件都无法读取。

* java中的IO流（四大家族）

java.io.InputStream字节输入流

java.io.OutputStream字节输出流

java.io.Reader字符输入流

java.io.Writer字符输出流

注：

-> Stream结尾是字节流，Reader与Write结尾是字符流。

->上述IO流都是抽象类。

->所有流都实现了java.io.Closeable接口，都是可以关闭的，都有close()方法。（每次需要关闭流通道）

->所有的输出流都实现了java.io.Flushable接口，都是可刷新的，都有Flush()方法。（将通道内还未输出的数据强行输出，清空通道）

如果没有flush,则可能会出现数据丢失。

->在java中只要“类名”以Stream结尾的都是字节流。以Reader/Writer结尾的都是字符流

* java.io包下需要掌握的16个流

①文件专属：

java.io.FileInputStream // 字节输入流，任何类型文件都可以采用该流来读

java.io.FileOutputStream

java.io.FileReader // 只能读取普通文本

java.io.FileWriter

②转换流：（将字节流转换为字符流）

java.io.InputStreamReader

java.io.OutputStreamWriter

③缓冲流专属：

java.io.BufferedReader

java.io.BufferedWriter

java.io.BufferedInputStream

java.io.BufferedOutputStream

④数据流专属：

java.io.DateInputStream

java.io.DateOutputStream

⑤标准流专属：

java.io.PrintWriter

java.io.PrintStream

⑥对象流专属：

java.io.ObjectInputStream

java.io.ObjectOutputStream

* + 1. **FileInputStream**
* 概念

1、字节输入流，万能的，任何类型的文件都可以采用这种流来读

2、字节方式，完成输入的操作，完成读的操作（硬盘到内存）

3、read()方法

// 从输入流读取一个数据字节，返回下一个数据字节，如果已经到达文件末尾，则返回-1。

缺点：

只能一次读取一个字节，内存与硬盘交互太频繁，时间与资源耗费较大。

4、如果想要完整读取文件数据字节，需要写一个while循环，遍历输出。（判断条件可以使用文件末尾必定返回-1）

5、使用流后必须关闭流。将close()方法放在finally中，必定关闭（注意空指针异常）。

* 改进FileInputStream

缺点：

1、read()方法一次只能读取一个字节，内存与硬盘交互太频繁，时间与资源耗费较大。

2、read()读取的是字节数据(a字符表示97)，不是实际字符。

改进：

1、减少内存与硬盘交互次数。

使用read(byte b[n])读取文件。

read(byte b[n])存储字节原理：

实际读取时是将字节读到了数组内进行储存。

进行读取文件时第一次将n个字节储存在byte[]内，之后每次读取都会覆盖byte[]中的元素，所以byte[]数组内元素在不断被替换。

而对于最后不足byte[].lenght个字节时，读取到几个就覆盖几个。

eg:

文件内容：abcdef；

byte[].lenght长度为4；

每次读取4个字节

第一次数组储存内容为：97,98,99,100

第二次数组储存内容为：101,102,99,100

【注意】

①实际读取时是将字节读到了数组内进行储存

②进行读取文件时第一次将n个字节储存在byte[]内，之后每次读取都会覆盖byte[]中的元素，所以byte[]数组内元素在不断被覆盖替换

③此时使用read(byte b[n])返回的是读取个数；

④未读取到数据返回-1。

2、怎样将字节转换为字符？

String的构造方法：String(byte bytes[])

原理：创建String对象的构造方法，通过byte[]数组创建字符串对象。

作用：可以将byte[]内所有元素(字节)转化为字符并以字符串输出

eg:

byte[] myByte = {97,98,99,100};

String myString = new String(myByte);

注意存在问题：

使用这种方式将获取到的文件字节码转换为对应字符是将数组内全部元素转换，而获取最后几个字符时，出现小于数组长度的情况，转换字符就会将未覆盖的元素也转化为字符。

所以，我们应该必须读取多少个数据转换多少个数据（read()方法返回读取个数）。

另外一个String构造方法：

String(byte bytes[], int offset, int length)构造方法

作用：指定开始索引与结束索引，将byte[]数组内指定索引区间的元素返回为字符串。

eg:

byte[] myByte = {97,98,99,100};

myByte = {101,102,99,100}

// 表示将myByte[]中索引0-2的字节转换为字符串。

// String myString = new String(myByte,0,2);

// read(byte b[])返回的恰好是读取字节数。

String myString = new String(myByte,0,read(myByte));

* FileInputStream读取文件步骤

代码详见：F:\java\_script\Project01\src\io\_flow\Flow01\_Test03.java

①定义储存流对象的全局变量（finally内close流时需要用到流对象）；

②创建文件字节输入流对象；

->此处需要处理异常，直接try…catch…finally；

->在finally中先进行空指针判断，再调用close()，同时处理close的异常。

③读数据

->先创建一个储存数据的byte[]数组,每次读取n个字节;

->循环读取文件内所有数据

【例】只展示核心代码

file01 = new FileInputStream("aaaa.txt");

byte[] myByte = new byte[4];

// 循环遍历读取输出

int count = 0;

while ((count = file01.read(myByte)) != -1){

System.out.print(new String(myByte,0,count));

}

* FileInputStream其他常用方法

// 以下方法使用该对象

FileInputStream file01 = new FileInputStream("aaaa.txt");

①longskip(long n) // 跳过几个字节不读

file01.skip(2);

②intavailable() // 返回流中剩余的没有读到的字节数量

file01.available();

妙用：

// 返回剩余未读字节数；

// 可以返回总字节数；

// 创建byte[]数组时，创建文件总字节数长的数组。一般不推荐

* + 1. **FileOutputStream**
* FileOutputStream构造方法

作用：创建流对象

① new FileOutputStream("aaaa.txt");

// 这种方式会将源文件的内容清空，然后再重新写入

② new FileOutputStream("aaaa.txt",true);

// 这种方式会在文档末尾处写入，不会清空原内容

* FileOutputStream写入步骤

代码详见：F:\java\_script\Project01\src\io\_flow\Flow02\_Test01.java

①创建输出流对象；

②处理异常，close()写在finally中，使用flush()刷新。

此处将close()与flush()先写到程序中。写操作执行完成后，先刷新再关闭。

②使用write()方法将字符串/字节写入到文件中；

注：字节流只能将任意类型的数据转为字节型，然后写入文件。（原因：write方法只能传入byte[]数组）

->创建字节类型数组byte[]，将数组传入到文件中。例：file\_out.write(myByte);

->创建字符串，将字符串转化为byte[]数组，然后传入到文件中。

字符串转换为byte[]：byte[] getBytes() // 实例方法，String类

注：传输byte[]数组，但是文档内保存的是将字节转换为对应的字符。

* FileInputStream与FileOutputStream联合使用完成文件拷贝

代码详见：F:\java\_script\Project01\src\io\_flow\Flow03\_Copy01.java

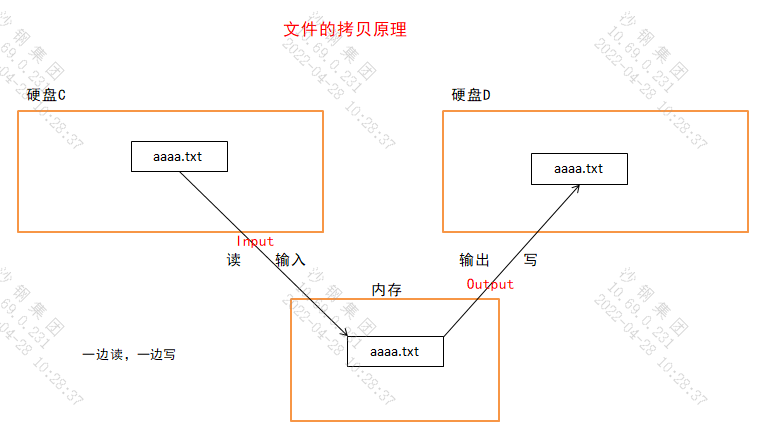
实现步骤：

①创建输入与输出流对象，指定被拷贝文件及指定目标文件；

②处理异常，关闭流及刷新输出流。创建byte[]数组；

③循环遍历边读边写，读多少写多少。

注：拷贝过程写入文件时，write()方法需要判断数组内读取到的数据有哪几个。此时：使用方法write(byte b[], int off, int len)。（len表示读取到数据的个数，与输入流中String方法作用类似）



* + 1. **FileReader**

1、只能读取普通文本。（字符输入流）<读>

2、读取文本内容时，比较方便、快捷

3、read()方法重载：

-> read()

-> read(char[] cbuf)

-> read(char[] cbuf, int off, int len)

4、底层创建的不是byte[]数组，而是char[]数组。

注：依然需要String类的构造方法将char[]内元素转换为字符串类型。

* + 1. **FileWriter**

1、只能输出普通文本（能用记事本编辑的）。（字符输出流）<写>

2、write()方法重载：

-> write(int c)

// 注：此方法写入时int c指的是ASCII值，写入后自动转化为对应字符。若想写入数字，只能以String类型写入。

-> write(char cbuf[]) // 可用作拷贝文件

-> write(String str)

-> write(String str, int off, int len) // 拷贝文件时有用到

FileReader和FileWriter联合使用拷贝文件步骤与FileInputStream和FileOutputStream拷贝步骤类似，此处不再赘述。

* + 1. **缓冲流**
* 常见缓冲流

① BufferedReader

② BufferedWriter

③ BufferedInputStream

④ BufferedOutputStream

以下只介绍BufferedReader流，其他缓冲流与其相似。

* BufferedReader概念

1、BufferedReader是一种带有缓冲区的字符输入流

2、使用BufferedReader创建对象后不需要创建byte[]或char[]类型的数组，自带缓冲。

3、怎样读取数据？

使用readLine()方法可以读取文本的一行数据。

String readLine()

注：

1、readLine()方法不会读取换行，如果使用print打印多行，依然可以打印在一行。

2、若读到最后一行没有字符，则readLine()方法返回null。（结束判断条件）

* 创建BufferedReader对象

构造方法：

BufferedReader(Reader in, int sz)

BufferedReader(Reader in) // 常用

注意：需要传递一个Reader类型的形参，这个Reader类型的流叫做节点流。

节点流：当一个流的构造方法需要传入一个流的时候，这个被传入的流被称为：节点流。

包装流：也称处理流。外部负责接收(包装)的流。（BufferedReader是包装流）

* Reader类型的流

一般字符流（输出）都继承了Reader抽象类。

源码：

public abstract class Reader implements Readable, Closeable{}

注：Reader是一个抽象类，不能直接创建对象。

一般使用已经继承了该类的流当做节点流。

例：FileReader类型的字符输入流。（FileReader类继承了Reader抽象类）

new BufferedReader(new FileReader(“src/io\_flow/Flow01\_Test01.java”));

* BufferedReader读取文件步骤

①创建一个实际字符输入流作为节点流（继承Reader抽象类的流）；

②创建BufferedReader对象（包装流），将节点流传入；

③处理异常，close()关闭流；

注：步骤①与②创建了两个流，此处只需关闭包装流。

原因：调用BufferedReader流中的close()方法关闭包装流，而这个close()方法的方法体中对节点流进行了关闭处理，所以只需要关闭包装流就可以同时关闭两个流。

④通过遍历读取文件中的字符。

注：使用readLine()方法读取文件时，读到最后一行没有字符会readLine()返回null，可以使得循环跳出。

读取时不会读换行。

代码详见：F:\java\_script\Project01\src\io\_flow\Flow06\_Test01.java

F:\java\_script\Project01\src\io\_flow\Flow07\_Test01.java

* + 1. **转换流**
* 概念

作用：将字节流转换为字符流。

使用场景1：当使用字节流作为节点流，BufferedReader作为包装流时，而由于字节流没有继承Reader抽象类不能当做形参传入到包装流中，此时使用转换流将字节流转换为字符流，这样就可以传入到包装流中。

分类：

-> InputStreamReader // 字节输入流转换为字符输入流

-> OutputStreamWriter // 字节输出流转换为字符输出流

* 使用转换流

此处只介绍InputStreamReader转换流，OutputStreamWriter与其使用类似。

源码：

public class InputStreamReader extends Reader {}

注：该转换流继承了Reader抽象类，所以InputStreamReader是一个字符流。

构造方法：

InputStreamReader(InputStream in)

将字节输入流传入，传入后此时InputStreamReader对象就是一个字符流。

注：创建了InputStreamReader流后，InputStreamReader就是一个包装流，参数为节点流。若将InputStreamReader流再传入给缓冲流,则缓冲流就是包装流，转换流就是节点流。

使用步骤：

①创建一个字节输入流FileInputStream对象，

②创建一个InputStreamReader转换流，将字节输入流传入转换流；

③创建一个BufferedReader缓冲流，将转换流对象传入缓冲流；

④处理异常，关闭流。（只需要关闭最外层的包装流）

⑤通过遍历读取文件中的字符。

注：第③、④点具体操作参照缓冲流使用步骤

代码详见：F:\java\_script\Project01\src\io\_flow\Flow06\_Test02.java

* + 1. **数据流**
* 常用数据专属流

① DateInputStream

② DateOutputStream

* DateOutputStream

作用：可以将数据连同数据类型一并写入文件。

注：使用数据流写入的数据不能打开查看，不是普通文本文件。

构造方法：

DataOutputStream(OutputStream out) // 注意传入一个字符输入流

// 创建数据专属流

DataOutputStream dataOut = new DataOutputStream(new FileOutputStream("aaaa"));

写入方法：

->writeByte(int v) // 也可以传入byte类型数据，自动转型

->writeShort(int v)

->writeInt(int v)

->writeLong(long v)

->writeFloat(float v)

……

* DateOutputStream

作用：读取用DateInputStream写的数据文件。

注：读取数据时，必须按照写入时的顺序才能正确读取数据文件。

构造方法：

DataInputStream(InputStream in)

// 创建数据流对象

DataInputStream in = new DataInputStream(new FileInputStream("aaaa"));

读取方法：

-> byte readByte()

-> short readShort()

-> int readInt()

-> long readLong()

-> float readFloat()

……

* + 1. **标准流**

作用：

标准流分为标准输入流与标准输出流，分别负责控制台的输入与输出。

本质：

在java中系统类System实现标准输入/输出的功能。因为其是基于原始的输入输出流实现的子类，所以标准输入输出流本质上仍然是一个字节输入/输出流。

标准输入流：

System.in是InputStream，标准输入流，默认可以从键盘输入读取字节。

标准输出流：

System.out是PrintStream，标准输出流，默认可以向Console（控制台）中输入字符和字节数据。

修改输入输出流方向：

修改输入流：System.setIn(InputStream);

修改输出流：System.setOut(PrintStream);

* 常用标准流

PrintWriter、PrintStream

// 标准输入输出流不需要进行关闭操作。

* 使用PrintStream改变输出流输出方向

// 默认输出到控制台，与System.Out类似

①创建一个标准输出流对象，让其指向新建的log文件内；

PrintStream printl = new PrintStream(new FileOutputStream("log"));

②修改输出方向

System.setOut(printl);

③输出

System.out.println("Hello World!");

// 此时输出不再是控制台，而是输出在log文件中。

* 案例：日志工具类

代码详见：F:\java\_script\Project01\src\io\_flow\Flow10\_Logger.java

首先创建一个工具类方法，然后根据以下步骤实现日志工具类。

①指定一个日志文件

PrintStream out = new PrintStream(new FileOutputStream("log.txt",true));

②改变输出方向

System.setOut(out);

③获取日期当前时间

Date now = new Date();

④指定时间格式

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

⑤指定格式对象

String strtime = sdf.format(now);

⑥输出

System.out.println(now + "：" + s); // s是工具类方法形参变量

* + 1. **对象流**

作用：用于存储和读取基本数据类型或对象的处理流。主要作用就是把java中的对象写入到文件源中（序列化），也可以把对象从文件源中还原回来（反序列化）

序列化：(Serialize)将对象的状态信息转换为可以存储或传输的形式存储（二进制内容）

序列化目的：为了对象可以跨平台存储，和进行网络传输。

反序列化：(DeSerialize)把硬盘中的字节序列恢复为对象的过程。

* 常见对象流

① ObjectInputStream<反序列化>：将硬盘中的数据（java对象）恢复到内存中

② ObjectOutputStream<序列化>：将内存中的数据（java对象）保存在硬盘中

* 序列化实现

实现步骤：

->将需要被序列化的类实现Serializable接口。

public interface Serializable {} // 标识性接口

该接口没有任何属性与方法，只是为了标注该类可以被序列化。

作用：JVM看到类实现这个接口后，会对该类自动生成一个序列版本号

->将一个输出流对象传入到ObjectOutputStream对象流中实例化一个对象流对象。

ObjectOutputStream oos =

new ObjectOutputStream(new FileOutputStream("students"));

->使用ObjectOutputStream对象的writeObject(Object o)将对象o写入到输出流指定文件中。

oos.writeObject(stu);

->记得对输出流刷新与关闭操作。

transient关键字：

表示游离的，被修饰的属性不参与序列化。

可以通过反序列化到内存测试，被修饰的属性值为null。

代码详见：F:\java\_script\Project01\src\io\_flow\ObjectOutputStream\_Test01.java

* 序列化版本号

1、Java语言怎样区分类？

①首先通过对比类名，不同类名，则为不同类。

②如果类名一致，通过比较序列版本号，序列化版本号不同，表示不同类。

2、序列化版本号：

同名，不同序列化版本号，代表不同的类。

只有当同名且序列化版本号都相同，才代表是同一个类。

3、序列化版本号使用场景：

十年前：写了一个Student类（该类实现了Serializable接口，自动生成版本号）

十年后：此时版本需要升级，必须改动Student类，修改类之后，此时重新编译会重新生成一个序列化版本号。

注：此时修改前后两个自动生成的序列化版本号不同，java认定编译后的两个类不是同一个类。在反序列化时，运行就会报错。

4、自动生成序列化版本号的缺陷

自动生成序列化版本号，在修改/更新类时，编译后会重新生成一个序列化版本号，这个版本号与修改前的序列化版本号不同，java虚拟机将修改前后看做不同的类。

所以自动生成序列化版本号不能修改类的代码。

**在序列化时，一般给定一个固定不变的序列版本号，不自动生成版本号**

eg:

① ArrayList类中定义的固定序列化版本号

private static final long serialVersionUID = 8683452581122892189L

② String类中定义的固定序列化版本号

private static final long serialVersionUID = -6849794470754667710L;

5、小知识

IDEA生成序列化版本号

settings -> Editor -> Inspections ->搜索Serializable

->勾选"Serializable class without serialVersionUID"

在实现了Serializable的"类名"上 alt+enter。

* 多个类对象怎样序列化

将对象存入集合中，将集合对象序列化。

将对象add到集合中后，按照序列化实现步骤对集合对象进行序列化即可。

* 反序列化实现

实现步骤：

->将一个输入流对象传入到ObjectInputStream对象流中实例化一个对象流对象。

ObjectInputStream ois =

new ObjectInputStream(new FileInputStream("students"));

->使用ObjectInputStream对象的readObject()方法将指定文件读到。

ois.readObject();

->关闭流。

* + 1. **IO + Properties**

IO：文件的读与写

Properties：Map集合，key与value都是String类型

作用：项目的配置文件中使用

将配置文件中的数据使用io流加载到properties集合中。

java配置文件规范：

->属性配置文件：key=value形式；

->建议以.properties为后缀存储（非必须）

->配置文件中以#为注释符，key与value尽可能不出现空格

* 配置文件加载步骤

①首先必须创建一个后缀为.properties的配置文件（按照规范将属性值写在配置文件中）；

②编写代码，创建Properties集合对象（key与value都是String类型）；

③创建输入流对象；

④调用Properties对象的load()实例方法，将文件的数据加载到集合中。

synchronized void load(Reader reader)

⑤使用getProperty()方法用key值获取value值。

* + 1. **拷贝目录**

将文件夹拷贝到另外一个文件下（包括文件内容）。

代码详见：F:\java\_script\Project01\src\io\_flow\Test01\_Eg.java

* 1. **多线程**
     1. **线程与进程概念**
* 进程

一个内存中运行的应用程序。每个进程都有自己独立的一块内存空间，一个进程可以有多个线程。

进程是程序的一次执行过程，是一个动态概念，是程序在执行过程中分配和管理资源的基本单位，每一个进程都有自己的内存空间。

例：一个车间 = 一个进程(即一个运行的程序)

* 线程

线程是进程中的一个执行任务(控制单元)，负责当前进程中程序的执行。一个进程中至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

线程是CPU调度和分派的基本单位，它可与同属一个进程的其他的线程共享进程所拥有的全部资源。至少有5种基本状态。(创建、就绪、执行、阻塞、终止)

例：车间内多个工人 = 多个线程

* 线程的五种状态转换(生命周期)

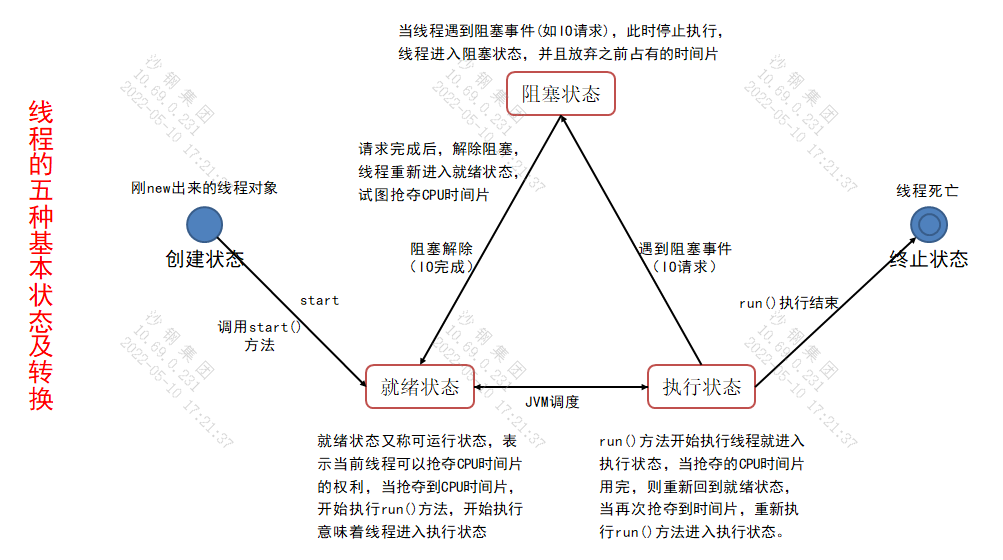


图2操作系统线程状态图

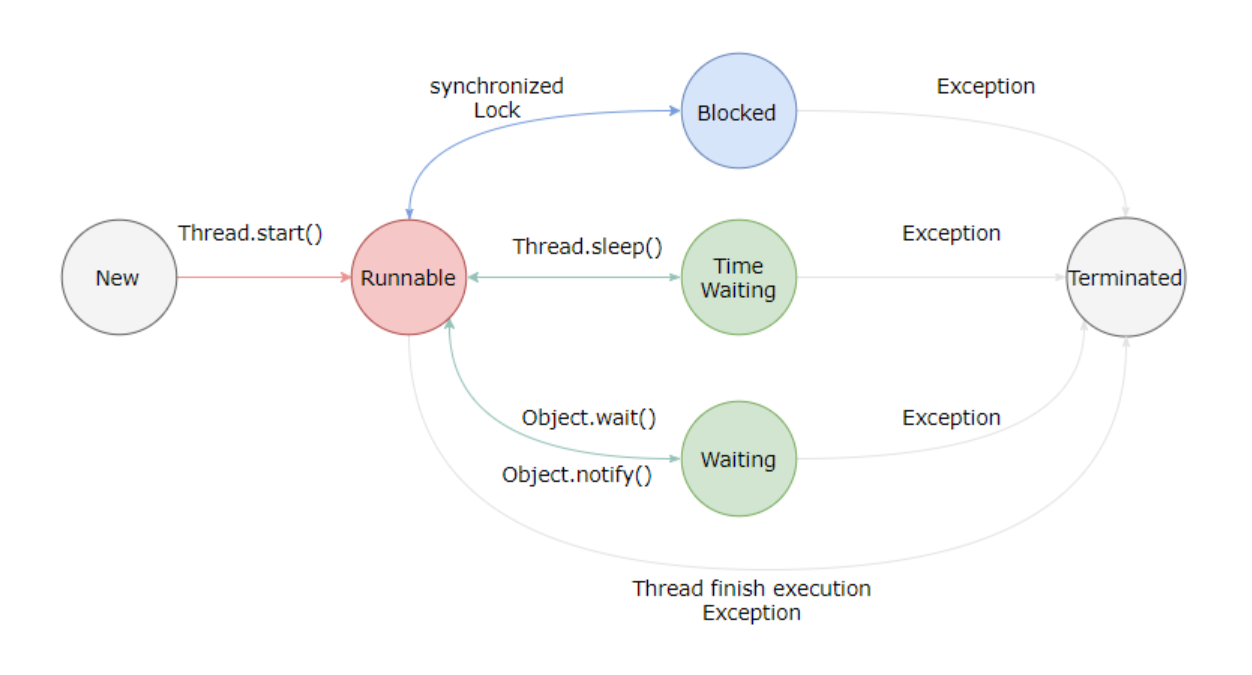


图3java线程状态图

**New(创建)**：创建线程但尚未启动。

**Runnable(可运行)**：可能正在运行，也可能正在等待CPU时间片。

注意：

①包含了操作系统中的就绪(Ready)与执行(Running)状态。

②处于就绪中的线程，会存储在一个就绪队列中，等待着被操作系统的调度机制选到，进入CPU 中运行。

③RUNNING和READY 状态，是我们自己为了方便描述而造出来的。无论是Java语言不区分这两种状态，在Java中统统叫RUNNABLE。

**Blocked(阻塞)**：等待获取一个排它锁，如果其他线程释放了锁就会结束此状态。

在不考虑虚拟机对synchronized的极致优化前提下。

当进入 synchronized 块或方法，获取不到锁时，线程会进入一个该锁对象的同步队列。

当持有锁的这个线程，释放了锁之后，会唤醒该锁对象同步队列中的所有线程，这些线程会继续尝试抢锁。如此往复。

例：线程a,b,c,d在同时运行，并且每个线程都有synchronized块，锁对象相同。当a线程优先抢 到了时间片执行时，得到了锁对象，则其余线程无法得到锁对象，会使得线程变为Blocked状态，等 到a线程释放锁对象，则其余线程重新回到Runnable状态(可运行线程)进行抢锁，以此往复执行。

**Waiting(无限期等待)**：等待其他线程显式的唤醒，否则不会分配CPU时间片。



① **wait**()

作用：

释放锁对象lock（隐含着必须先获取到这个锁才行）；线程状态变成WAITING(等待状态)； 当前线程进入lock对象的**等待队列**。

② **notify**()

作用：唤醒线程。

必须由另一个线程执行同一个对象锁的notify，唤醒后线程从等待队列中释放，变为可运行线程 (Runnable)，但是线程依然需要抢锁，如果抢不到会进入对象锁的同步队列中。

③ **join**()

作用：先将当前线程挂起，待其他线程结束后在执行当前线程的代码。

<实际底层依然是wait()/notify()配合执行>

例：

当主线程执行到t.join()的时候，主线程会变成Waiting状态，直到线程t执行完毕，主线 程才会变回RUNNABLE状态，继续往下执行。

**Timed Waiting(限期等待)**：无须等待其他线程显式的唤醒，在一定时间内就会被系统自动唤醒。

注意：

sleep表示使一个线程睡眠；wait()表示将一个线程挂起

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

**阻塞与等待的区别**：

阻塞是被动的，它是在等待获取一个排它锁。而等待是主动的，通过调用Thread.sleep()和Object.wait()等方法进入。



**Terminated(死亡)**：可以是线程结束任务之后自己结束，或者执行stop，或者产生了异常而结束

java.lang.IllegalThreadStateException // 异常类型

实际上在start方法中直接进行了异常控制。

源码：

public synchronized void start() {

if (threadStatus != 0)

throw new IllegalThreadStateException();

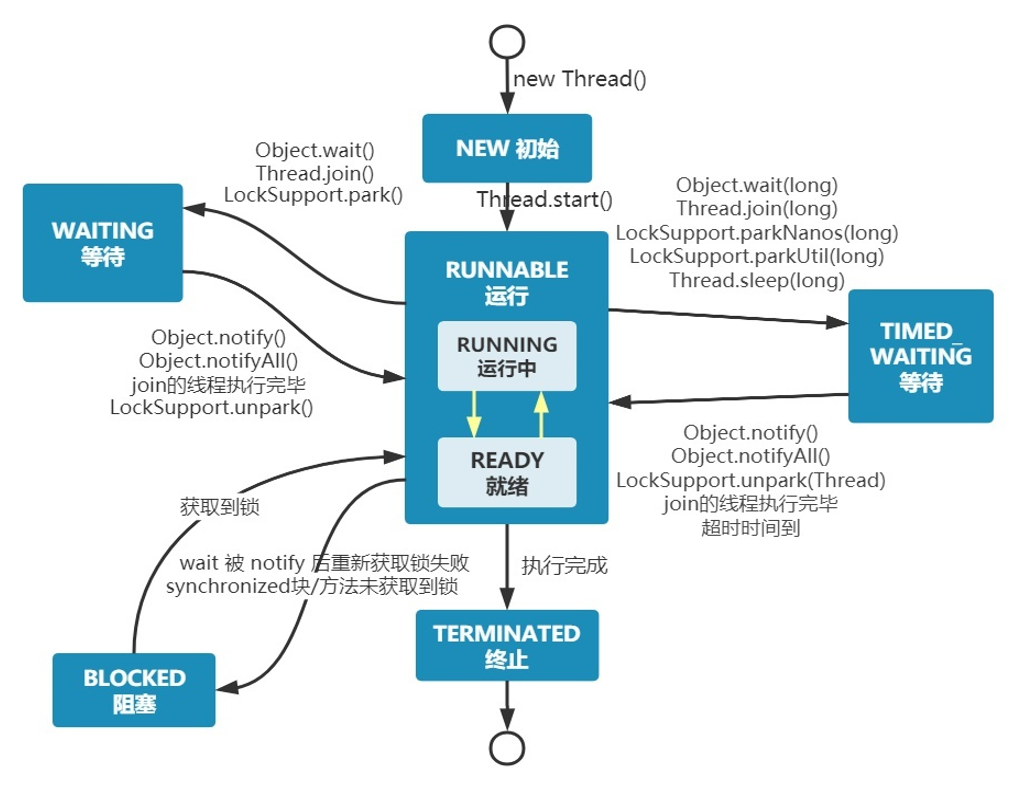
...

}

#### 操作系统与java线程状态区别

Java将操作系统到的五态模型中的就绪和执行，都统一成RUNNABLE，将阻塞（即不可能得到CPU 运行机会的态）细分为了BLOCKED、WAITING、TIMED\_WAITING。

BLOCKED、WAITING、TIMED\_WAITING这几个状态，线程都不可能得到CPU的运行权，你叫它挂起、阻塞、睡眠、等待，都可以。



* java中的进程与线程

在dos窗口中输入一个.class文件时，计算机会先启动JVM，而JVM就是一个进程。JVM再启动一个主线程调用main方法。同时再启动一个垃圾回收线程负责看护，回收垃圾。现在java程序至少有了两个线程并发执行。

注意：main()方法结束，程序不一定就执行结束了，只是main()方法所在的主线程执行完毕，其他线程未执行完依然会压栈弹栈。

* 进程与线程的关系

进程A与进程B的内存不共享。（例如：京东与阿里的资源不共享）

线程A与线程B，在**java中堆内存与方法区内存共享**(只有一块)。**栈内存独立，一个线程一个栈**

区别：

进程是操作系统资源分配的基本单位，线程是处理器任务调度和执行的基本单位。

* 多线程并发

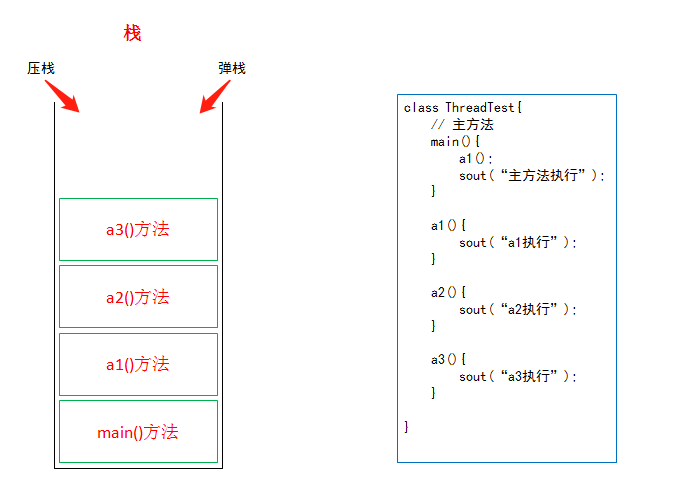
真正的多线程并发是多个线程同时在执行。

多线程的目的：

提高程序的执行效率。

单核CPU怎样做到多线程并发执行的？

单核CPU就相当于只有一个人在处理事情(线程任务)，而由于计算机的执行速度足够快，因此，在不断地切换多个线程执行，给人的感觉就像是在同时并发执行多个线程。



### 线程实现

#### 常见线程的实现

① 通过继承Thread类本身

② 通过实现Runnable接口(开发常用)

③ 通过Callable和Future创建线程

#### 继承Thread类(方式一)

步骤：

① 编写一个线程类，直接继承java.lang.Thread，重写run()方法

public class MyThread extends **Thread**{

// 重写

public void **run**() {

// run()方法体

}

}

② 创建线程对象，启动线程。(在主方法main()中)

// 创建线程类对象

MyThread mt = new MyThread();

// 启动线程

mt.**start**();

注意：

1、线程类中的run()方法不能在main()方法中直接调用，直接调用run()方法只是普通方法，并 不会开辟新的分支线程。

2、start()方法的作用(线程创建启动原理)：

启动一个分支线程，在jvm中开辟一个新的栈空间，start()方法任务完成之后瞬间就结束 了。开辟出新的栈空间后，start()方法就结束了。此时线程启动成功。启动成功线程之后会自 动调用run()方法，并且run()方法在分支栈的底部(压栈)。run()方法在分支栈的底部，main() 方法在主栈的底部。两个方法同级。

3、start()方法不结束，主方法内start()方法下方的代码是不会执行的。

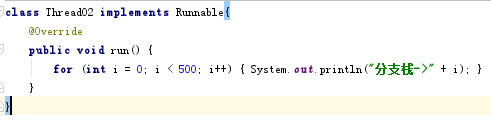
原因:java中必须自上而下执行代码。只是因为start()方法仅仅是开辟了一个分支栈，而分支 栈开辟结束， start()方法就结束了。执行run()方法时间是start()执行结束后开始执行的。 并且与start()方法下面的for循环一起并发执行。



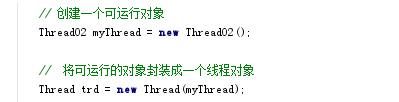
#### 实现Runnable接口(方式二)

步骤：

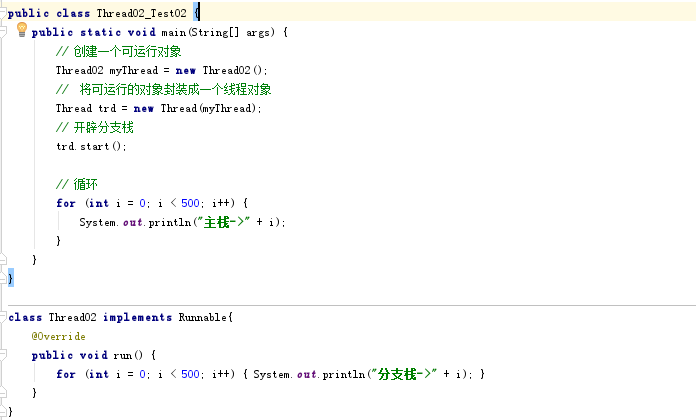
① 编写一个线程类，实现java.lang.Runnable接口，实现run()方法



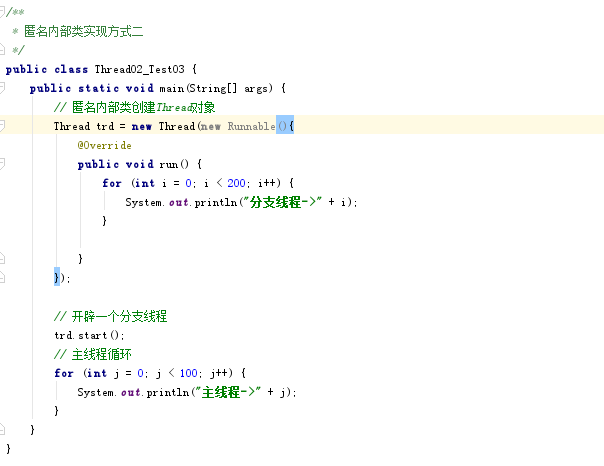
② 创建可运行类对象，将可运行对象传入Thread对象中



③ 使用Thread类中的start()方法开辟分支栈



匿名内部类实现上面步骤：



#### 实现Callable接口(java8新特性)

@FunctionalInterface

public interface **Callable<V>** {

V **call()** throws Exception;

}

注：call()方法可以抛出异常(throws),有返回值,run不能够抛出异常,并且没有返回值

获取返回值：

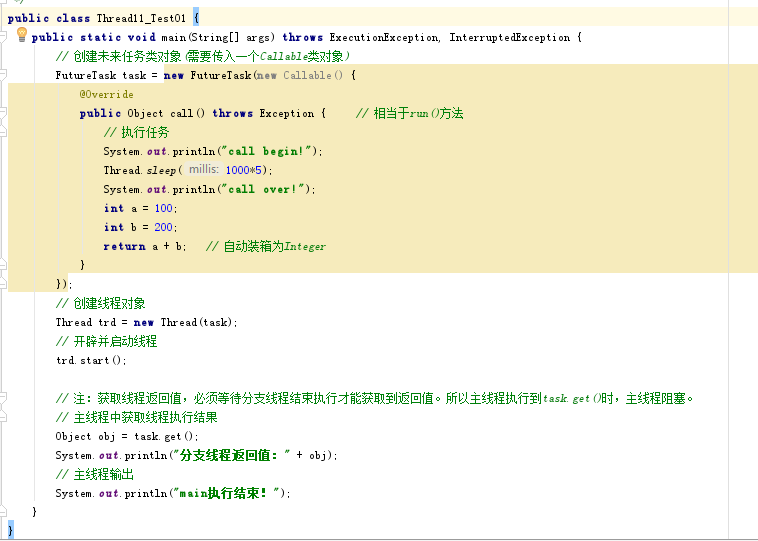
**V get()** // FutureTask类中的方法

// 这种方式获取分支线程返回值时，分支线程必须执行结束才可以获取到返回值。所以主线程 执行获取分支线程返回值程序时，主线程阻塞，等分支线程执行结束才可以继续执行主线程剩余 代码。

优缺点：

优点：可以获取分支线程返回值

缺点：执行效率低。(执行过程主线程阻塞)



### 线程常用方法

#### start() // 启动线程，调用该线程的run方法

详细参见线程实现，此处不再赘述。

#### run()

详细参见线程实现，此处不再赘述。

#### currentThread() //获取当前线程对象

**static native Thread currentThread();**



注意：

① currentThread()方法是静态方法，需要使用Thtead类进行调用。

②Thread.currentThread()出现在哪获取的就是哪个线程对象。要想获取分支线程的对象，将 方法写在Thread类run()方法中即可。

#### getName() // 获取当前线程名称

**final String getName()** // 实例方法

未修改线程名称时，会默认提供一个名称。

例：Thread-0，Thread-1，Thread-2



#### setName(name) // 修改线程对象的名字

**final synchronized void setName(String name)** // 实例方法



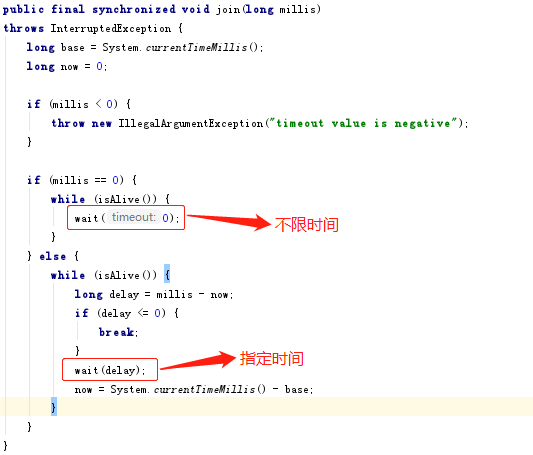
#### join(long millisec) // 等待该线程终止的时间最长为millis毫秒。

**final void join()** // 实例方法

作用：先将当前线程挂起，待其他线程结束后在执行当前线程的代码。

<实际底层依然是wait()/notify()配合执行>

底层原理：



既然join()底层是由wait()进行控制的，如何唤醒线程？

答：Thread类线程执行完run()方法后，一定会自动执行notifyAll()方法

坑点：

注意：join()方法作用是阻塞线程，等待wait方法被调用的线程对象执行结束后，调用语句所 在的线程才会被释放执行。

如果执行**Thread.currentThread().join()**;则线程会一直被阻塞，无法终止。

因为它一直在等待自己结束，形成死锁。

#### sleep(long millisec) //让当前正在执行的线程休眠(进入阻塞状态)

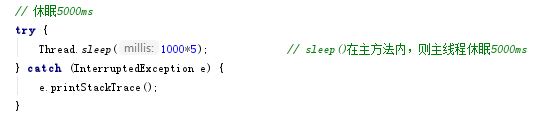
**static native void sleep(long millis)** // 静态方法

作用：

让当前线程进入休眠状态，进入“阻塞状态”，放弃占有的CPU时间片，让给其他线程使用。

效果：

将方法可以放在程序特定的位置，让程序间隔一段时间执行再执行(模拟网络延迟)



注意：

① 静态方法可以通过对象去调用，但是并不会受到对象的影响，依然遵循sleep()的位置 决定休眠的线程。

即：sleep()在主方法内，则主方法进行休眠。

② 参数是毫秒。

③ sleep()方法存在异常，需要将异常抛出或处理。

**如何终止sleep()？**

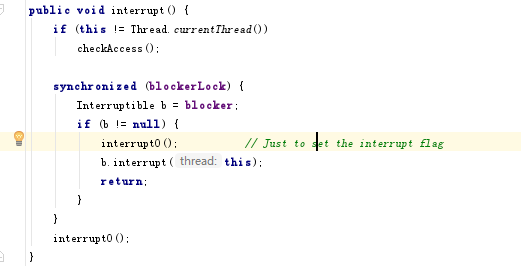
在线程睡眠未结束时打断睡眠，让线程执行程序。

解决方案：使用方法interrupt()进行终止睡眠。

原理：<sleep()方法中可以抛出异常>

通过异常处理机制结束睡眠。执行interrupt()方法后，线程对象中的sleep出现异常， 执行catch语句块内容，然后将run()结束

interrupt()源码：



代码详见：D:\javascript\Project01\src\xiancheng\Thread04\_Test02.java

#### interrupt() //中断线程

详细参见sleep()方法终止睡眠，此处不再赘述。

#### stop()/添加终止线程标记 // 强行终止线程

stop()这种方式容易丢失数据，因为这种方式是将线程直接杀死。

**添加终止线程标记**

->在run()方法外打一个boolean标记

boolean sign = true;

->在run()方法内判断sign是否为true，如果为true，则执行run()方法内容，否则return；

->在main()方法内只需要在合适位置将标记改为false即可。

注意：

修改标记时，在具体的类对象下的属性访问标记。不是在封装后的线程对象。

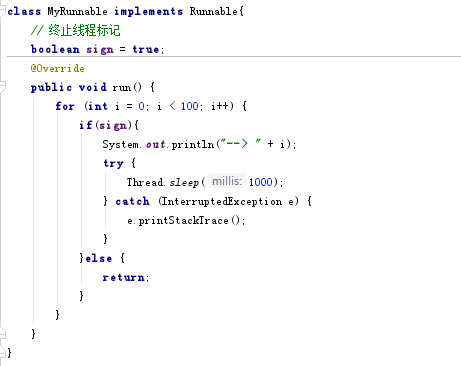


图4分支线程类

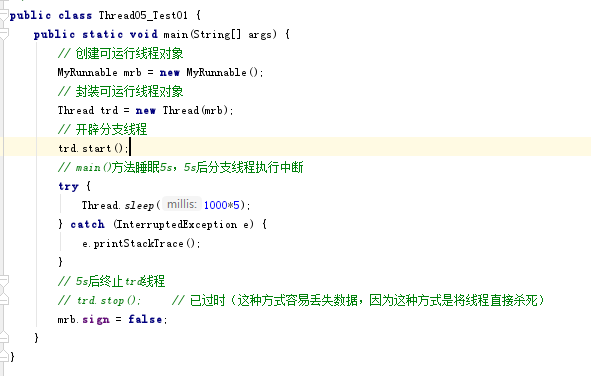


图5主方法进行强制终止分支线程运行

#### wait(long s) // 将当前运行的线程挂起(进入等待状态)

final native void wait(long timeoutMillis)

注意lock.wait()方法一经调用，则会发生三件事情：

① 释放锁对象lock（隐含着必须先获取到这个锁才行）；

<线程必须执行中才能让其等待挂起>

② 线程状态变成WAITING(等待状态)；

③ 线程进入lock对象的**等待队列**。

#### notify()/notifiAll() // 唤醒指定对象线程池中的当前线程/所有线程

作用：唤醒被wait()后存入锁对象的等待队列中的线程

注意：必须由另一个线程执行同对象的notify，唤醒后线程从等待队列中释放，变为可运行线程 (Runnable)，但是线程依然需要抢锁，如果抢不到会进入对象锁的同步队列中。

<**注意**：

① wait()与notify()不是线程类Thread中的方法，而是Object类中的方法，每个对象都有

② wait()与notify()方法是建立在synchronized线程同步基础上

>

### 线程调度

#### 常见线程调度

①抢占式调度模型

哪个线程优先级高，抢占到的CPU时间片概率就高一些。<java采用这种方式></>

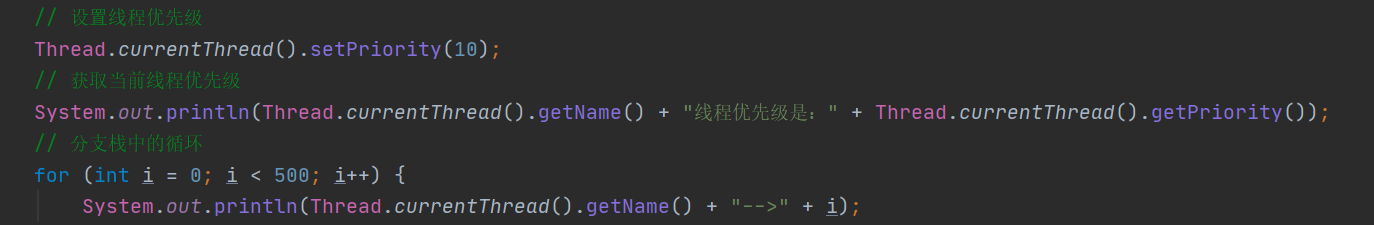
②均分式调度模型

平均分配CPU时间片。每个线程占有的CPU时间片时间长度一样。

#### 线程调度相关方法

①setPriority(int priority) // 更改线程优先级

**final void setPriority(int newPriority)**



②getPriority() // 获取线程优先级

**final int getPriority()**

线程优先级：

最低优先级：1

默认优先级：5 // java默认优先级为5

最高优先级：10

优先级较高，获取到的CPU时间片就可能多一些。

注意：

* 线程优先级高的不一定先执行，优先级低只是意味着获得调度的概率低，并不是优先级低就不会被调用了，都是看CPU的调度。
* 优先级的设定建议在start()调度前

③yield() // 暂停当前正在执行的线程对象，并执行其他线程(进入就绪状态)

**static native void yield()**

静态方法：使用Thread.yield()调用

注意：

yield()方法不是阻塞。让当前线程让位，给其他线程使用；

原理：

底层是从'运行状态'回到'就绪状态'，但回到就绪状态后会再次参与到CPU时间片的抢夺。

### 线程安全

#### 概念

线程安全：

多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

线程不安全：

不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

#### 多线程非线程安全前提

①多线程并发执行；

②有共享数据；

③共享数据有修改行为。

#### 避免非线程安全(同步与异步)

采用线程同步机制。

同步：

就是线程排队的意思。牺牲多线程并发的高效率，保证线程执行的安全性。

线程t1与线程t2，有顺序的执行两个线程，t1执行时必须等t2执行完成，或者t2执行时必须等t1执行完成.

两个线程存在等待关系，效率较低。<线程排队执行>

异步：

线程t1与线程t2，各自执行各自，谁也不需要等谁。<多线程并发>

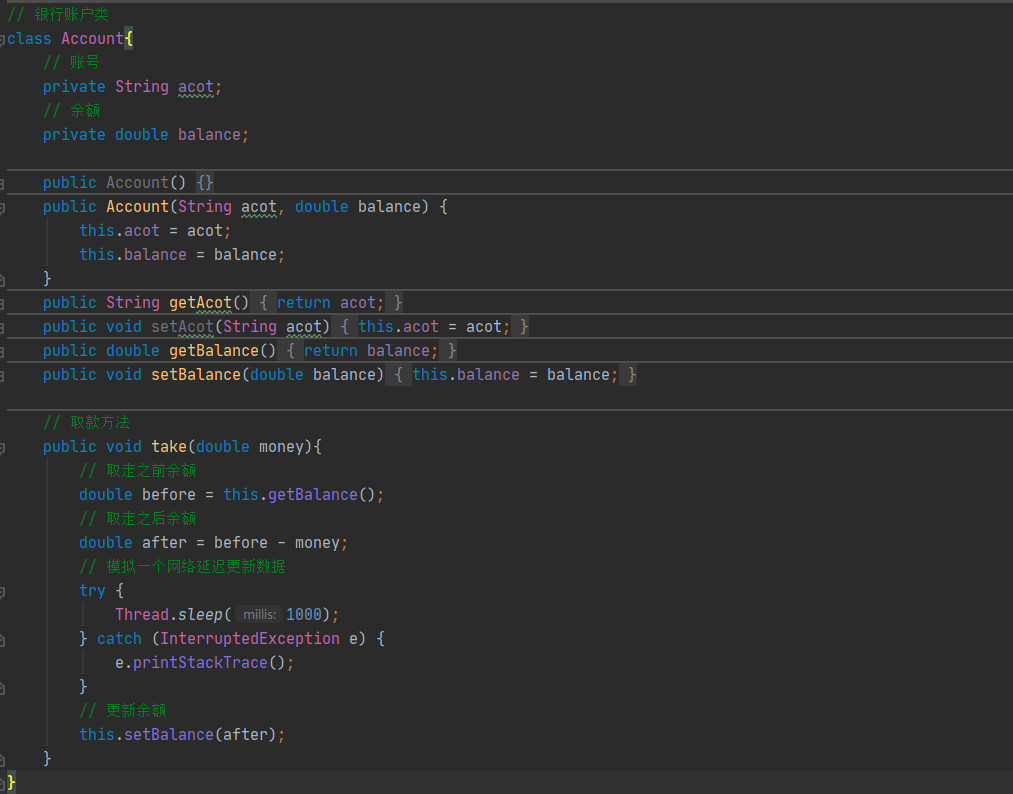
显然，同步最安全，最保险的。而异步不安全，容易导致死锁，这样一个线程死掉就会导致整个进程崩溃，但没有同步机制的存在，性能会有所提升。

**线程安全问题发生原因：**

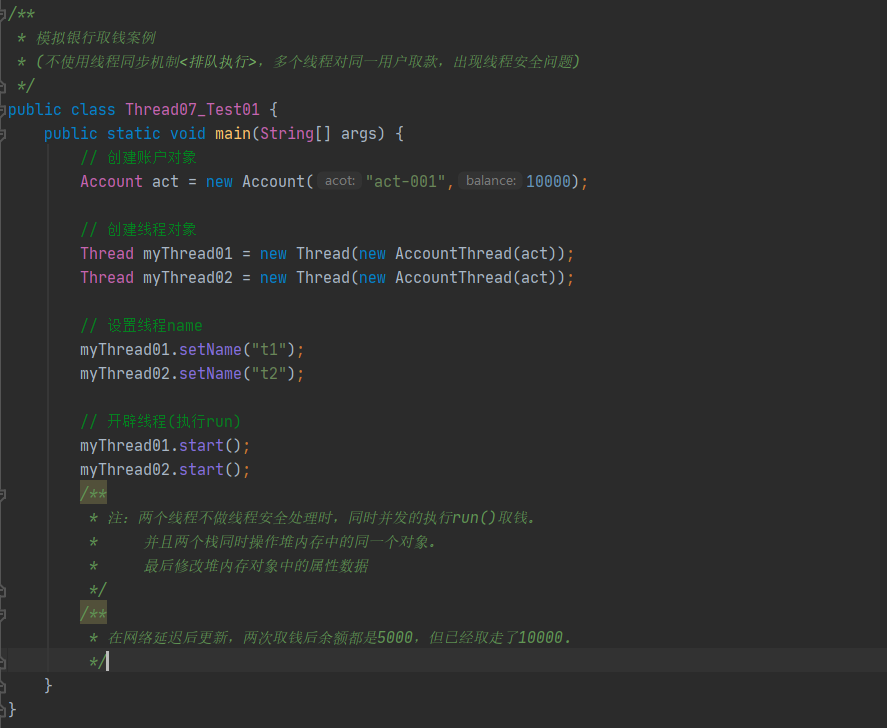
线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量，静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

#### 银行取钱案例

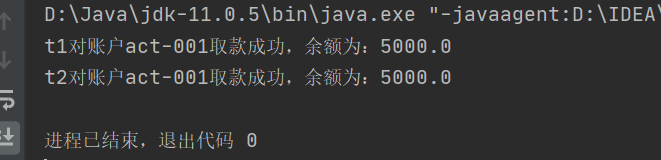
情景：用户A与用户B在同一时间分别在ATM与柜台取用户C的钱，假若C账户只有10000元，用户A、B同时取钱每人可以取出10000元。







输出结果：



解释：

在取钱过程中，要想让两个线程同时取钱，而取钱成功的根本是更新系统中账户余额。所以，在更新前让当前线程休眠1s钟，保证两个线程同时发生更新操作，但实际两个线程都已经取到钱，因为after变量都是5000。

结果让两个不同的客户端，在账户只有10000的前提下，同时取走了5000元，但是系统剩余是5000。<非线程安全>

问题：银行系统的账户信息是共享的，属于共享资源，同时进行修改时就会有线程不同步的现象。

### 线程同步(排队)机制

<以上面的银行取钱业务为例>

问题出现的原因：当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

如何解决：对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。

当一个线程t1在操作账户act-001的时候，其他线程不能参与进来。直到线程t1操作完act-001时(余额修改完成时)，其他线程才可以开始操作act-001。这种情况即使线程t1出现了阻塞，也不能被改变。

线程同步过程：

**当进入 synchronized 块或方法，获取不到锁时，线程会进入一个该锁对象的同步队列。**

**当持有锁的这个线程，释放了锁之后，会唤醒该锁对象同步队列中的所有线程，这些线程会继续 尝试抢锁。如此往复。**

#### 线程同步机制语法

**同步代码块：**

**synchronized**(对象){

// 线程代码块

}

// 要同步静态方法，需要一个用于整个类对象的锁，synchronized括号后的对象为Xxx.class

**同步方法：**

修饰符**synchronized**返回类型方法名(){

可能会产生线程安全问题的代码;

}

**在静态方法上使用**

<保证静态变量安全></>

表示类锁，不是对象锁了；类锁只有一个。

就算创建100个对象，只会有1个类锁。

① 只能同步方法，不能同步变量和类；

② 每个对象只有一个锁；(当想要同步时，必须清楚在哪个对象上进行同步)

③ 不必同步类中的所有方法，类可以同时拥有同步和非同步方法；

④ 如果线程拥有同步和非同步方法，则非同步方法可以被多个线程自由访问而不受锁的限制；

⑤ 线程睡眠时sleep，它所持有的任何锁都不会被释放；

⑥ 线程可以获得多个锁(例：在一个对象的同步方法里面调用另外一个对象的同步方法)。

⑦ 同步会损坏并发性，应该尽可能缩小同步范围。同步不但可以同步整个方法，还可以同步方法中一部分代码

#### 线程同步原理

假设t1与t2线程并发，开始执行代码时，抢夺时间片后，会有先后顺序；

若t1线程执行到synchronized代码块时，先到的自动找到‘共享对象’的对象锁，然后占有对象锁(标记被占用)；

当t2线程抢到时间片也执行到synchronized代码块时，发现共享对象被占有，此时t2只能在对象锁的同步队列中等待；

t1执行完代码块中的程序后，释放对象锁。此时t2得到共享对象锁再执行synchronized代码块中的程序。

这样就达到了排队执行的效果。

**深入理解synchronized**：

1、在账户类中分别创建一个实例变量和一个局部变量，synchronized小括号中分别写两个对象

-> 实例变量可以实现线程同步

原因：创建的obj是一个在账户类中的实例变量，实例变量存储在堆内存中，堆内存只有一个， 所以对象唯一且是两个线程都使用的共享对象，即可以实现线程同步。

-> 局部变量不可以实现线程同步

原因：局部变量存储在栈内存中，每个栈中都会有一个相对应的局部变量，是不同的对象，而一 个线程就是一个栈。所以这儿局部变量不共享

<局部变量永远都不存在线程安全问题<局部变量不共享，一个栈一个线程>

2、在账户类中创建一个储存在常量池中的字符串局部变量，将对象传入synchronized小括号中。可 以实现线程同步。

原因：虽然该字符串是局部变量，但确定的字符串被存储在字符串常量池中(方法区)，并不是在栈中 存储，所以不同的栈执行程序中用到的字符串对象并没有区别，即实现线程同步。

3、空指针不可以实现线程同步

以上三点在本例中可以使用的原因是：

由于t1与t2两个线程都需要线程排队执行，所以可以在账户类中写实例变量和常量池变量作为共享对象。

若重新创建一个账户对象

Acct act02 = new Acct("act-002",10000);

增加一个t3线程。

t3线程并没有处理t1与t2共享的对象，而是对act02对象进行账户操作了。所以t3线程不需要进行线程同步。此时如果使用实例变量与常量池常量作为共享对象，那么t3线程就会进行线程同步，所以这两个作为共享对象就不合理了。

即，synchronized中一般是填入this。本例中this指act对象。若添加t3线程，则this指act02对象，t3线程不和t1、t2 进行排队，它会单独拿到自己处理的act02对象的锁执行代码块。

(this不是通用的，需要注意synchronized代码块所在的位置，位置变了this不一定就是共享对象了)

故：小括号内共享对象需要看线程的具体共享对象是谁，它是变化的。

#### 开发中怎样避免和使用线程安全问题？

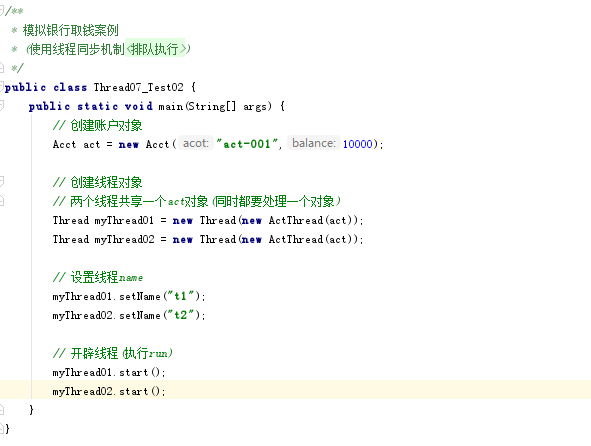
1、尽量使用局部变量代替"实例变量"与"静态变量"；

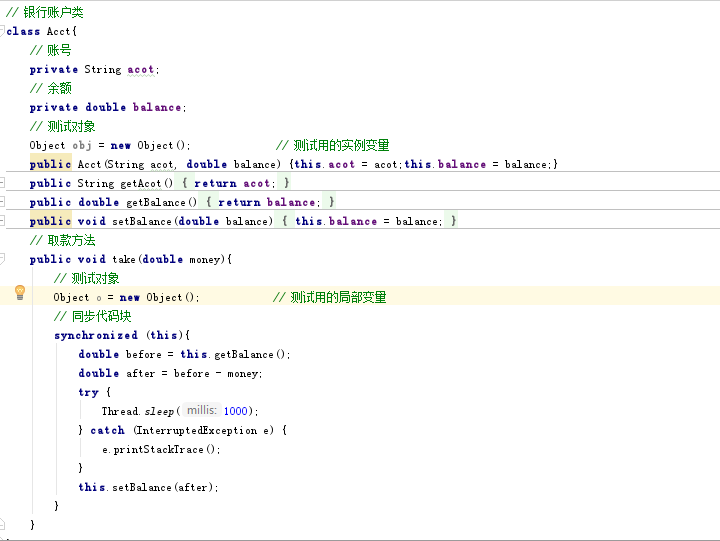
2、如果必须使用实例变量，则考虑创建多个对象，这样实例变量内存就不会共享。

3、如果不能使用局部变量代替，对象也不能创建多个，这时再使用synchronized，线程同步机制。

#### 银行取钱案例同步实现







注意：上面的同步代码块中的就是同步信息。

#### 锁的原理

Java中每个对象都有一个内置锁。

当程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this）有关的锁。获得一个对象的锁也称为获取锁、锁定对象、在对象上同步。

当程序运行到synchronized同步方法或代码块时对象锁才起作用。

一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获取锁，直到第一个线程释放锁。即，被占用了对象锁后其他线程都不能获取该对象上的synchronized方法或代码块，直至锁释放。

释放锁是指获取了锁的线程退出了synchronized同步方法或代码块。

如何释放锁？

①当前线程的同步方法/同步代码块执行结束；

②当前线程在同步代码块、同步方法中遇到break、return终止了该代码块/该方法的继续执行；

③当前线程在同步代码块/同步方法中出现了未处理的Error或Exception，导致异常结束；

④当前线程在同步代码块/同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

不会释放锁：

①线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行；

②线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（不同监视器）；应尽量避免使用suspend()和resume()来控制线程；

#### 死锁

指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现 象，若无外力作用，它们都将无法执行下去

现象：程序不会出现异常，也不会报错，而是一直僵持在控制台中，线程之间相互等待所导致。

例：

当线程a拥有了A对象的锁，想要去获取B对象的锁；线程b拥有了B对象的锁，想要拥有A对 象的锁，两个线程在获取锁的时候，都不会释放已经持有的锁，于是，就造成了死锁。

<无限期阻塞>

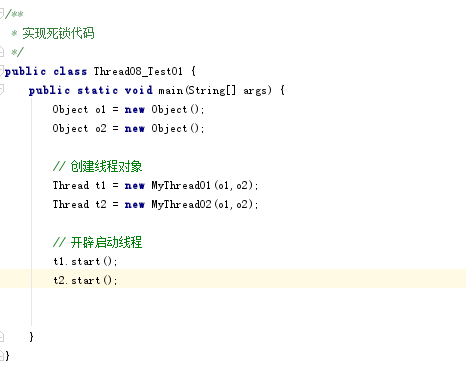
**死锁产生的原因(必要条件)：**

互斥使用：当资源被一个线程使用或者占用时，别的线程不能使用该资源；

不可抢占：获取资源的一方，不能从正在使用资源的一方抢占掠夺资源，资源只能被使用者主动 释放；

请求和保持：资源请求者在请求别的资源时，同时保持对已有资源的占有；

循环等待：即 p1 占有 p2 的资源，p2 占有 p3 的资源，p3 占有 p1 的资源，这样形成了一个 等待环路。





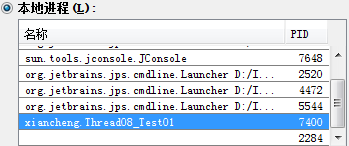


**如何使用idea查看死锁的线程？**

① 点击idea下面的Terminal 

② 在命令窗口输入jconsole，然后回车；

③ 双击运行中发生死锁对应的类；



④ 切换到线程，点击下面的'检查死锁'



**如何避免死锁？**

死锁的产生必须满足互斥使用，不可抢占，请求和保持，循环等待这四个条件，但是只要破坏其 中任意一个条件即可破坏死锁，其中最容易破坏的就是循环等待这个条件。

<调换某一个线程中的对象锁获取顺序>



### 守护线程

#### 概念

java语言中线程分两类：

① 用户线程

② 守护线程(后台线程)

其中最有代表性的是：垃圾回收线程(守护线程)

守护线程特点：

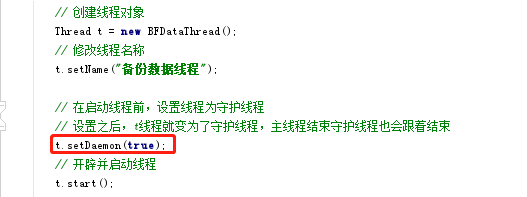
一般守护线程是一个死循环，所有用户线程只要结束了，守护线程就会自动结束。

注：主线程main是一个用户线程

守护线程的实现：

**final void setDaemon(boolean on)** // 实例方法

将线程设置为守护线程的方法，参数为true时，表示设置为守护线程。



守护线程用在什么地方？

例：系统在每天00:00进行数据自动备份。

此时需要一个定时器，可以将定时器设置为一个守护线程。

#### 定时器

作用：间隔一定的时间执行特定的程序。

实现方式：

最原始方式：使用sleep()方法，设置睡眠时间，每到特定时间点执行任务。

在java类库中有一个定时器：java.util.Timer。可以直接使用，但在开发中不常用这种方式。

实际开发中，使用较多的是Spring框架中的SpringTask框架。只需要简单配置就可以完成定时 任务。

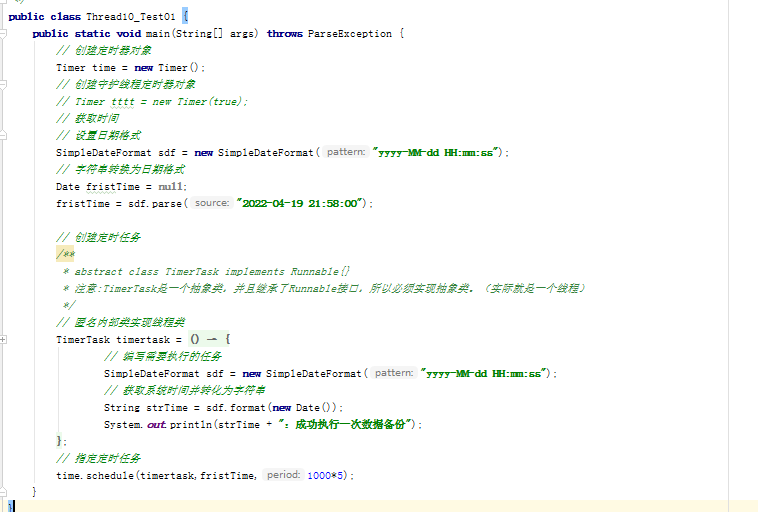


图 6实现从2022-04-19 21:58:00开始每5s执行一次备份操作

指定定时任务

**time.schedule(定时任务,第一次执行时间,步长间隔多久执行一次);**

### 生产者与消费者模式(wait/notify使用)

① 目的：

生产线程负责生产，消费线程负责消费

最终生产与消费线程要求达到均衡

这是一种特殊需求

② wait()与notify()不是线程对象的方法，是普通的继承Object类所有对象都有的方法。

③ wait()与notify()方法是建立在synchronized线程同步基础上的。因为生产与消费两个线程共享一个操作仓库对象

④ wait()作用：o.wait()方法让正在o对象上活动的当前线程进入等待状态，并且释放当前占有的对象锁。

⑤ notify()作用：o.notify()方法唤醒当前进入等待状态的线程，不会释放对象锁。

【案例】

需求：

仓库使用List集合；

List集合中假设只存一个元素；

List中有1个元素就表示仓库已满；

List中有0个元素就表示仓库已空；

最终效果：生产一个消费一个(最佳效果)

详解：D:\javascript\Project01\src\xiancheng\Thread12\_Test01.java

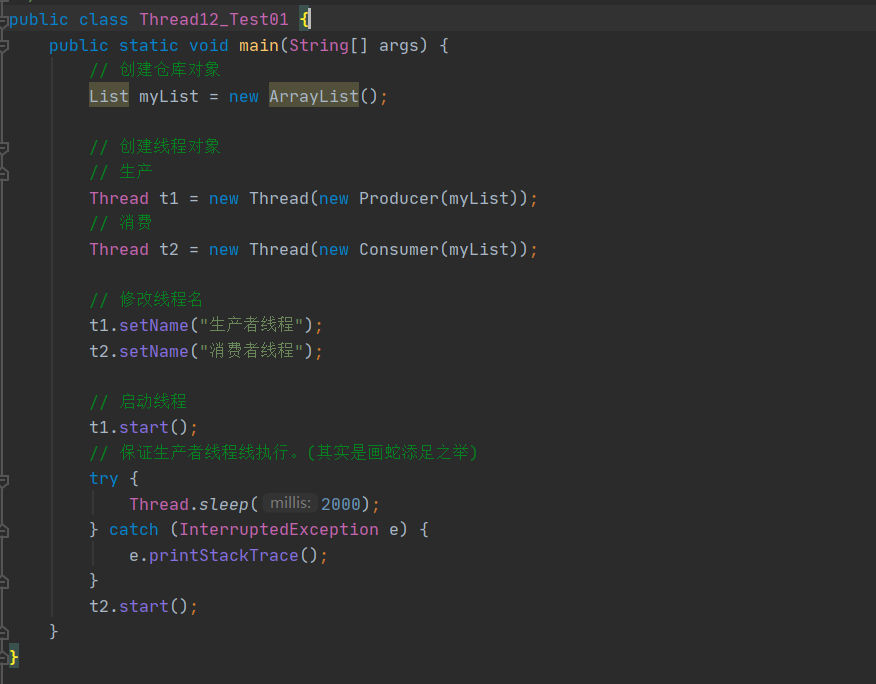


图 7 主线程main方法



图 8 生产者线程

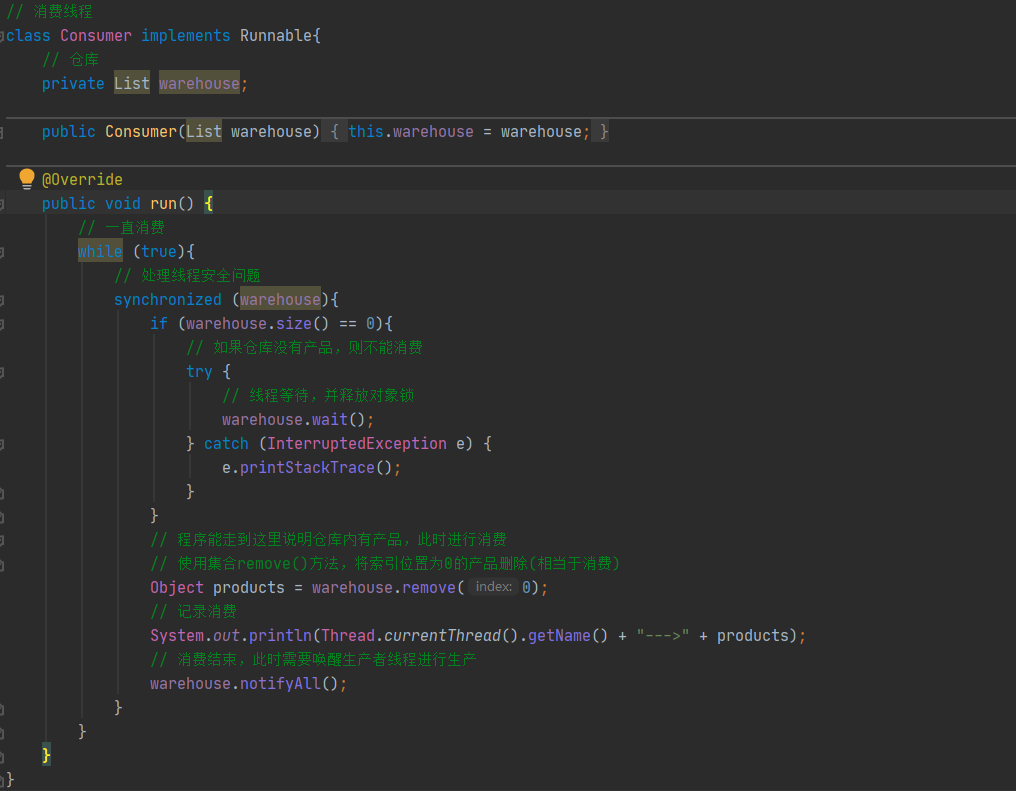


图 9 消费者线程

## 反射机制

### 概述

#### 反射简介

反射就是把Java类中的各个部分，映射成一个个的Java对象，拿到这些对象后可以做一些事情。

<用于操作字节码.class文件>

例如，一个类有：成员变量，方法，构造方法，等信息，利用反射技术咱们可以把这些组成部分映射成一个个对象

使用的**前提条件**：必须先得到代表的字节码的Class，Class类用于表示.class文件（字节码）

#### 反射机制相关重要的类

-> java.lang.**Class**

// 代表整个字节码，代表一个类型/一个类。eg:String.class

-> java.lang.reflect.**Method**

// 代表字节码文件中的方法。代表类中的方法

-> java.lang.reflect.**Constructor**

// 代表字节码文件中的构造方法。代表类中的构造方法

-> java.lang.reflect.**Field**

// 代表字节码文件中的属性。代表类中的成员变量(静态变量+实例变量)

#### Java创建对象方式

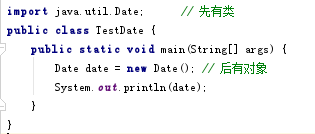
常见的三种方式：

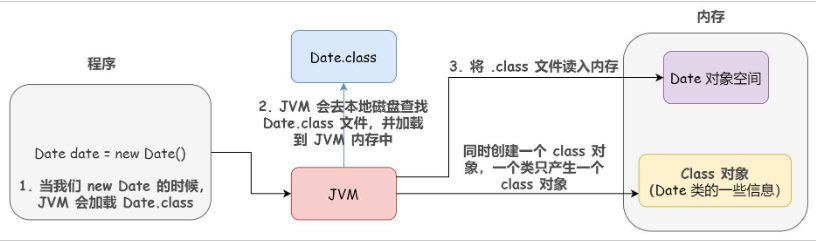
1、使用new关键字：这是我们最常见的也是最简单的创建对象的方式

2、使用Clone的方法：无论何时我们调用一个对象的clone方法，JVM就会创建一个新的对象，将前面的对象的内容全部拷贝进去

3、使用反序列化：当我们序列化和反序列化一个对象，JVM会给我们创建一个单独的对象

#### 反射原理





首先JVM会将你的代码编译成一个.class字节码文件，然后被类加载器（ClassLoader）加载进JVM的内存中，同时会创建一个Date类的Class对象存到堆中（注意：这个不是new出来的对象，而是类的类型对象）。JVM在创建Date对象前，会先检查其类是否加载，寻找类对应的Class对象，若加载好，则为其分配内存，然后再进行初始化new Date()。

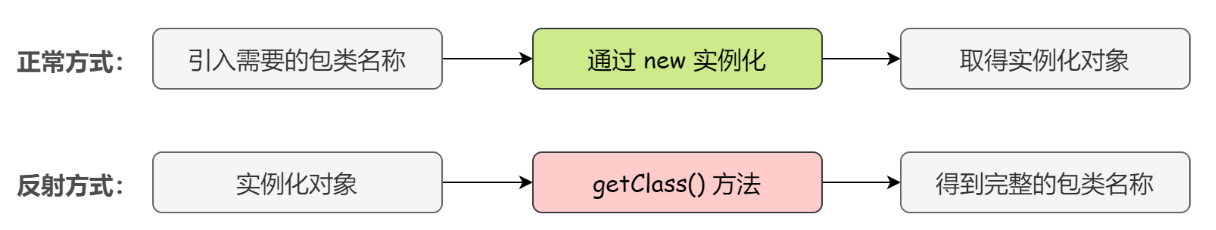
需要注意的是，每个类只有一个Class对象，也就是说如果我们有第二条new Date()语句，JVM不会再生成一个Date的Class对象，因为已经存在一个了。这也使得我们可以利用==运算符实现两个类对象比较的操作：

System.out.println(date.getClass() == Date.getClass()); // true

那么在加载完一个类后，堆内存的方法区就产生了一个 Class 对象，这个对象就包含了完整的类的结构信息，我们可以通过这个 Class 对象看到类的结构，就好比一面镜子。所以我们形象的称之为：反射。

说的再详细点，再解释一下。上文说过，在通常情况下，一定是先有类再有对象，我们把这个通常情况称为“正”。那么反射中的这个“反”我们就可以理解为根据对象找到对象所属的类（对象的出处）

通过反射，也就是调用了getClass()方法后，我们就获得了Date类对应的Class对象，看到了Date类的结构，输出了Date对象所属的类的完整名称，即找到了对象的出处。当然，获取Class对象的方式不止这一种。



### 获取字节码Class并创建

#### Class类

java代码是存储在后缀名是.java的文件里的，但是它会被编译，最终真正去执行的是编译后的.class文件。Java是面向对象的语言，一切皆对象，所以java认为这些编译后的class文件，这种事物也是一种对象，它也给抽象成了一种类，这个类就是Class。

实际Class类对象代表整个字节码，代表一个类型/一个类。eg:String.class

Class类的实例表示正在运行的Java应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。（包括基本数据类型以及关键字void）

Class没有公共构造方法。Class对象是在加载类时由Java虚拟机以及ClassLoader类加载器中的defineClass方法自动构造的。也就是这不需要我们自己去处理创建，JVM已经帮我们创建好了。

作用：

获取到.class文件对象后，存储到Class类型的变量中，通过Class类中的方法操作.class对象中的各种属性、方法等

#### Class.forName(String className) //方式一

**static Class<?> forName(String className)**

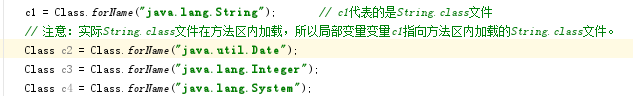
注意：

1、该方法是一个静态方法，使用Class.forName()调用；

2、参数是一个字符串，是某一具体类的全限定包名，例：java.lang.String；

3、正因为方法的参数接收的是全限定类名字符串，所以一般框架中将它写在配置文件中，然后 利用反射生成对应的对象；

4、forName()方式获取字节码文件时，一般直接将代码写在静态代码块中，直接在类加载时运行。



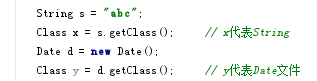
#### 对象.getClass() //方式二

**final native Class<?> getClass();**

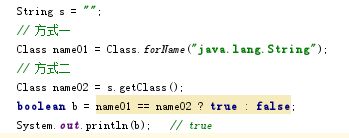
注意：

1、getClass()方法在Object类中，是一个实例方法，所以必须使用具体的对象进行调用；

2、Class<?>:返回Class类型值



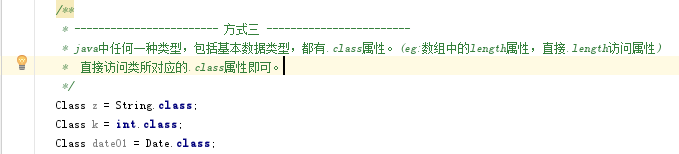
比较方式一与方式二的获取字节码对象后Class对象是否相同：



可以看出，同一个类加载器加载的文件在一次程序运行过程中，只会被加载一次，无论使用哪种方式获得的类对象都是同一个

#### 类型.class //方式三

java中任何一种类型，包括基本数据类型，都有.class属性。(eg:数组中的length属性，直接.length访问属性)



注意：三种获取Class文件的方式在不同的场景使用。

Class.forName(“类全名”)：多用于配置文件，将类名定义在配置文件中，读取配置文件，加 载类；

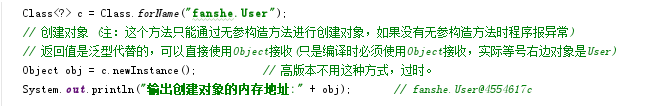
类名.class:多用于参数的传递

对象名.getClass()：多用于对象获取类对象

#### 创建实例对象

newInstance() //实例方法，实例化对象

重点：只能调用类中的无参构造方法创建对象。必须保证无参构造方法存在。



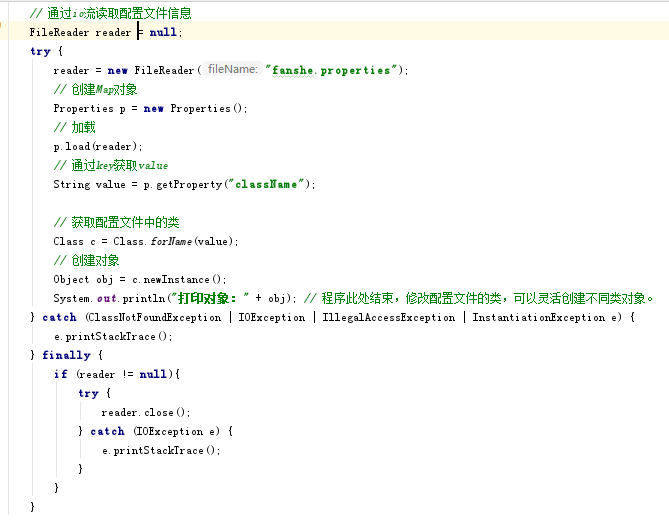
Class的newInstance()方法进行实例化的意义：

通过反射机制灵活创建对象。

【例】配置文件读取并使用反射创建对象

写一个properties类型的配置文件，将编译的字节码文件位置写入配置文件中，当修改配置文件中的字节码文件名，就可以灵活的创建不同类型的对象。

fanshe.properties配置文件内容：className=java.util.Date



#### 配置文件存储位置

1、idea中，默认的相对路径的根路径是在Project项目路径内的。

2、将配置文件存储在idea根路径下的缺点：移植性差。

如果代码离开了idea，换到其他位置，可能当前路径就不是项目根路径了。

FileReader read = new FileReader("F:\\java\_script\\Project01\\src\\fanshe2.properties")

3、通用存放位置：配置文件存储在类路径下。

类路径：src下的都是类路径(类的根路径下)。(与包同路径，或在包内，都叫src下)

即：properties配置文件一般存在项目src目录下

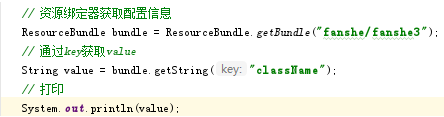


#### 资源绑定器获取properties配置文件信息

1、这种方式只能获取properties后缀的文件内容

2、文件必须存储在类路径下

3、路径不能写后缀properties后缀



#### Class类的方法

① 获取Class字节码文件名称

String getName() // 全限定类名(java.lang.String)

String getSimpleName() // 简类名(String)

② 获取Class类的父类

**native Class<? super T> getSuperclass(); // 实例方法**

注意：java的继承是单继承形式，所以只会返回一个Class类。

③ 获取Class类所实现的接口

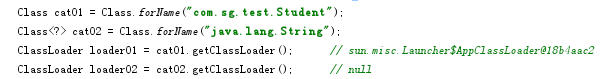
**Class<?>[] getInterfaces() // 实例方法**

注意：实现可以有多个实现接口，所以返回一个Class数组

④ 获取类加载器

**ClassLoader getClassLoader()**

注意：如果此类由引导类加载器加载，则此方法将在此类实现中返回null。



### 类加载器

专门负责加载类的命令/工具(ClassLoader)

jdk中类加载器分类：

1、引导类加载器(Bootstrap classLoader)：rt.jar

2、扩展类加载器(Extensions ClassLoader)：ext\\*.jar

3、应用类加载器(Application ClassLoader)：classpath

#### 类加载器原理

代码执行之前，首先会将所需要的类加载到方法区内存中。

加载类使用的就是类加载器，而类加载器会用以下方式加载类。

-> 首先通过"启动类加载器"加载

引导类加载器专门加载：C:\Program Files\java\jdk1.7.0\_17\jre\lib\rt.jar

rt.jar包是jdk中最核心的类库。

-> 若没有在rt.jar中找到对应类，则会通过"扩展类加载器"加载

扩展类加载器专门用来加载：C:\Program Files\Java\jdk1.7.0\_17\jre\lib\ext

这个位置下的所有jar文件都会被访问，直至找到对应类。

-> 若扩展类加载器没有加载到对应类，则通过"应用类加载器"加载。

应用类加载器专门加载：classpath中的类

#### 双亲委派机制

java为了保证类加载的安全，使用了双亲委派机制

场景：

自定义一个类，java.lang.String。

再创建一个测试Test类，使用String类，猜一猜Test中使用的是自定义的String，还是jdk 中的String类。

实际上Test中使用的是jdk中的String类，双亲委派机制保护了jdk核心API不被破坏。

作用：

1、保护程序安全，防止核心API被随意篡改。在java.lang包下，开发者自定义的类中的main方法不允许执行，防止恶意代码对程序产生破坏。

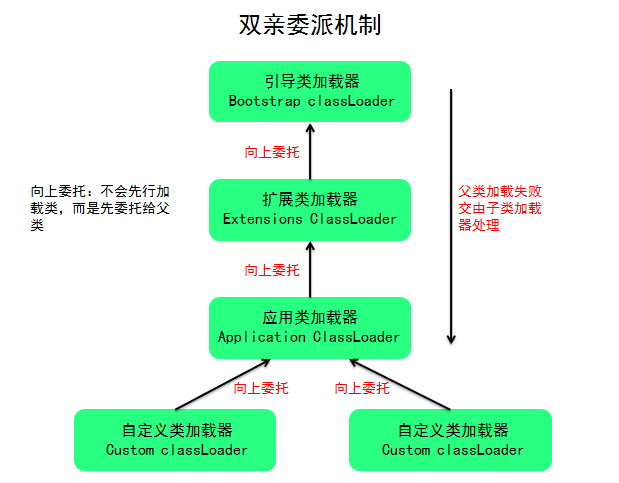
2、避免类的重复加载。一个类只会被加载一次。

原理：

1、如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类 的加载器去执行。

2、如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归请求最终将到达顶层的 启动类加载器。

3、如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务， 子加载器才会尝试自己去加载，这就是双亲委派模式。



### 反射方法(Method)

1、首先必须获取对应类文件

获取方式参见上面创建字节码对象三种方式。

2、通过Class中的方法获取方法对象

① Method[] **getMethods**() // 实例方法

// 获取Class对象代表的类的所有的非私有方法，数组，包含从父类继承而来的方法

② Method[] **getDeclaredMethods**() // 实例方法

// 获取Class对象代表的类定义的所有的方法，数组，但是不包含从父类继承而来的方法

③ Method **getMethod**(String name , Class<?>... parameterTypes) // 实例方法

// 获取Class对象代表的类的指定方法名的非私有方法

Class<?>... parameterTypes：是方法的形参列表中形参的类型。

例：myClass.getMethod("login", String.class, String.class); // 类型必须一致

④ Method **getDeclaredMethod**(String name, Class<?>... parameterTypes) // 实例方法

// 获取Class对象代表的类的指定方法名的方法

注意：Method[]数组中存储的是Method类型的方法对象。

#### 反射机制Method类中的方法

① String getName() // 获取方法名

② Class<?> getReturnType() // 获取返回值类型

④ int getModifiers()

// 获取方法的修饰符序号(返回的是int类型，int数据代表着修饰符。)

例：默认修饰符0，public 1，private 2，protected 4，static 8，final 16，

组合修饰的话返回和的数字：public static 9；

⑤ static String toString(int mod) // 这个方法在java.lang.reflect.Modifier内

// 将getModifiers()方法返回的int类型转换为String类型的修饰符字符串

例：Modifier.toString(m.getModifiers()) // 两个方法配合使用

⑥ Class<?>[] getParameterTypes()

// 获取方法形参类型列表（只获取类型，返回一个Class[]类型数组）

例：doSome(String s, int i)方法，获取结果：[String, int]

⑦ String getSimpleName()

// 获取形参类型名(这个方法是Class类的方法)

与getParameterTypes()组合使用。

使用：使用for循环，取出形参类型的数组中元素，然后使用该方法获取形参类型

#### 反射的方法如何调用< invoke() >

依然是java.lang.reflect. Method的方法

**Object invoke(Object obj, Object... args)**

参数：

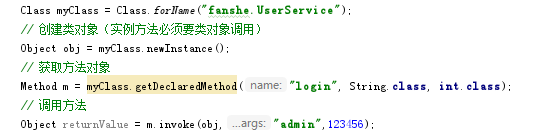
Student s01 = new Student("qlj", "男");

s01.get Mark("qlj"); // 通过姓名获取成绩

**Object obj**表示调用反射的方法的对象<类对象>，形如：s01对象

**Object... args**表示方法的实际参数<实参列表>，形如：get Mark()中传入的参数"qlj"。

如何使用：



### 反射构造方法(Constructor)

构造方法的作用实际是为了创建对象。

#### 步骤

1、首先必须获取对应字节码文件

例：Class c = Class.forName("fanshe.VipUser");

2、创建对象

① 调用无参数构造方法

Object obj = c.newInstance();

c.getDeclaredConstructor().newInstance();

注：这两个newInstance()方法不是同一个，前者是Class类中的方法，后者是Constructor 类中的方法。

Class直接调用newInstance实际上底层是使用Constructor中的newInstance()方法.

② 调用有参构造方法

-> 获取有参的构造方法对象；

-> 调用构造方法创建对象

#### 获取有参的构造方法对象

**Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)**

Class<?>... parameterTypes：形式参数列表

例：c.getDeclaredConstructor(String.class, String.class);

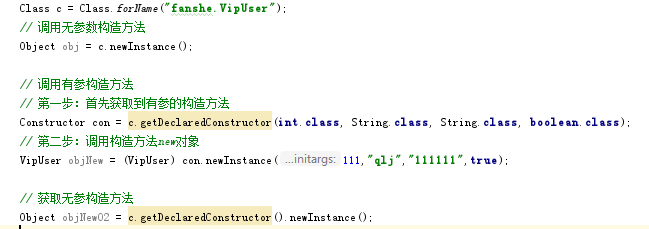
#### 调用构造方法创建对象

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

Object ... initargs：实际参数列表

例：con.newInstance("qlj","111111");

注：实际newInstance的对象就是指定对象，只是编译的时候要使用Object接收。



### 反射属性<成员变量>(Field)

#### 获取属性对象

首先必须获取字节码对象，通过字节码对象获取属性对象。

① 指定属性名称获取

**Field getField(String name)**

注：只能获取public修饰的成员变量

② 获取类中所有属性

**Field[] getFields()**

注：返回一个Field[]类型数组；只能获取public修饰的成员变量，如果没有public修饰的成 员变量，则返回一个空数组。数组存储Field类型的数据，并不是对象属性的类型。

③ 获取类中的所有属性

**Field[] getDeclaredFields()**

注：可以获取任意权限的属性；数组存储Field类型的数据，并不是对象属性的类型。

④ **Field[] getDeclaredField(String name) // 不限制权限**

#### Field类常用方法

① String getName()

// 获取属性名

② Class<?> getType()

// 获取属性类型的字节码对象

例：field.getType() // class java.lang.String

也可以使用Class类方法getSimpleName()获取简单字节码对象名

③ int getModifiers()

// 获取修饰符关键字序号(与方法的修饰符使用方式一样)

配合Modifier.toString(field.getModifiers())使用。

④ void set(Object obj, Object value)

// 给属性赋值

注意：虽然私有属性对象可以使用getDeclaredField()获取，但是并不能进行set()赋值操作。

**打破私有封装限制<不安全>**：

void setAccessible(boolean flag)

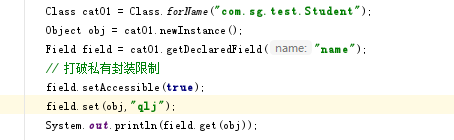
例：fieldObj.setAccessible(true) // true表示打破私有限制

⑤ Object get(Object obj)

// 获取属性值

注意：

get/set()方法必须先创建一个实际对象，这个对象并不是指字节码对象，而是new出来的 对象，在反射中使用newInstance()创建对象。



## 注解(Annotation)

### 概述

#### 简介

Java注解（Annotation）又称Java标注，是JDK5.0引入的一种注释机制。

Java语言中的类、方法、变量、参数和包等都可以被标注。和Javadoc不同，Java标注可以通过反射获取标注内容。在编译器生成类文件时，标注可以被嵌入到字节码中。Java虚拟机可以保留标注内容，在运行时可以获取到标注内容。当然它也支持自定义Java标注。

注解Annotation是一种引用数据类型.编译后生成.class文件。

注解使用：

->注解使用时的语法格式：

@注解类型名

->注解可以出现在类、属性、方法、变量等位置上

注解也可以出现在注解类型上。（默认情况下注解可以出现在任何位置上）

#### 作用

1、编写文档：通过代码里标识的注解生成文档 【生成文档doc文档】

2、代码分析：通过代码里标识的注解对代码进行分析 【使用反射】

3、编译检查：通过代码里标识的注解让编译器能够实现基本的编译检查 【Override】

#### 注解分类

1、Java自带的标准注解

包括@Override、@Deprecated和@SuppressWarnings，分别用于标明重写某个方法、标明某个类或方法过时、标明要忽略的警告，用这些注解标明后编译器就会进行检查。

2、元注解

元注解是用于定义注解的注解，包括@Retention、@Target、@Inherited、@Documented，@Retention用于标明注解被保留的阶段，@Target用于标明注解使用的范围，@Inherited用于标明注解可继承，@Documented用于标明是否生成javadoc文档。

3、自定义注解

可以根据自己的需求定义注解，并可用元注解对自定义注解进行注解。

### 自定义注解

#### 创建自定义注解

【语法】

元注解

​ **public @interface 注解名称**{

​ **属性列表 // 例：int num();**

​ }

注解本质其实上是一个接口，该接口默认继承Annotation接口

相当于：

public interface 注解名称extends java.lang.annotation.Annotation {}

【属性列表】

1、属性列表相当于接口中的抽象方法的定义，但在注解中叫做属性。

2、属性类型：byte、short、int、long、float、double、boolean、char、String、Class、enum、 以及这些类型的数组类型。

3、注解定义中如果有属性时，使用注解时必须给属性指定值。(相当于必须实现接口方法)

4、指定默认属性值：

例：int num() **default** 20;

#### 使用自定义注解

注解直接在类、属性、方法、变量等位置上添加标注即可。

注解使用的位置通过@Retention进行控制。

几种使用场景：

1、@MyAnnotation // 注解中没有定义属性

2、@MyAnnotation("qlj") // 注解中只有一个属性(例：String value();)

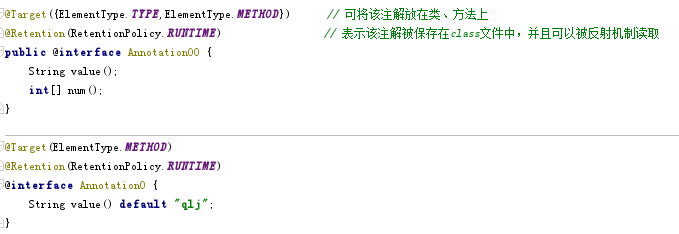
3、@MyAnnotation(name = "qlj", age = 23)

// 注解中有两个属性，必须有属性名。

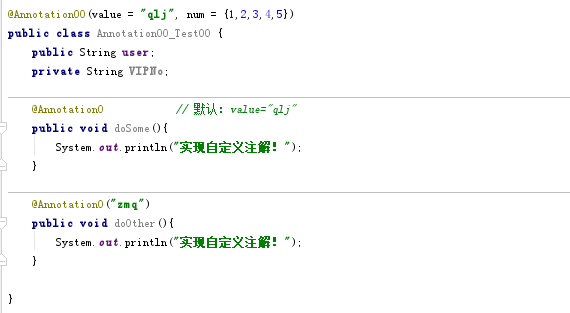
4、@MyAnnotation(num = {1,2,3,4})

// 注解中只有一个属性，但接收类型为数组类型，也可以省略属性名。

【创建】



【使用】



### 元注解

**@Target**

标记这个注解应该是哪种 Java 成员。

**@Retention**

标识该注解只被保留在java源文件中、class文件中、保存在class中并可以被反射读取。

**@Documented**

标记这些注解是否包含在用户文档中。

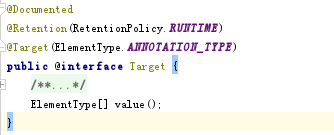
**@Inherited**

标记这个注解是继承于哪个注解类(默认：注解并没有继承于任何子类)

#### @Target // 被修饰的注解可以用在什么地方

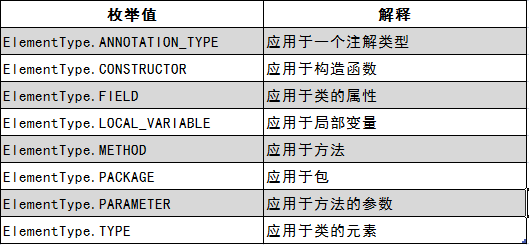
描述注解的使用范围（即：被修饰的注解可以用在什么地方）

【源码】



解释：

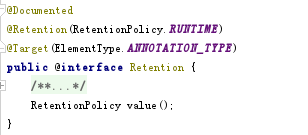
ElementType[]：是一个枚举类型，由jdk直接定义。



#### @Retention // 注解被保留的阶段

描述注解保留的时间范围（即：被描述的注解在它所修饰的类中可以被保留到何时）。

【源码】



解释：

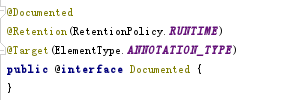
RetentionPolicy：是一个枚举类型。



#### @Documented

描述在使用javadoc工具为类生成帮助文档时是否要保留其注解信息。

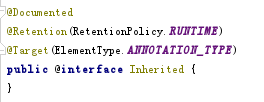
【源码】



#### @Inherited

被它修饰的Annotation将具有继承性。如果某个类使用了被@Inherited修饰的Annotation，则其子类将自动具有该注解。

【源码】

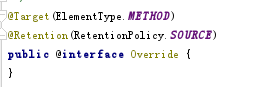


### 内置注解

#### @Override // 重写编译检查

告诉编译器被修饰的方法是重写的父类的中的相同签名的方法，编译器会对此做出检查，若发现父类中不存在这个方法或是存在的方法签名不同，则会报错

【源码】



#### @Deprecated // 是否过时

告诉编译器被修饰的程序元素已被“废弃”，不再建议用户使用。

【源码】

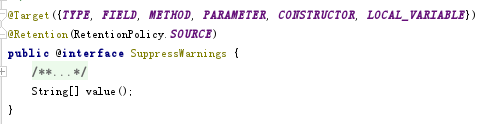


#### @SuppressWarnings

告诉编译器忽略指定的警告信息。

一般用在类的前面，传递参数"all"，表示压制该类的所有警告

【源码】



例：@SuppressWarnings(value={"deprecation","unchecked"})

// 表示对"它所标注的内容"中的"SuppressWarnings不再建议使用警告"和"未检查的转换时的警告"保持沉默。

### 在反射中使用注解(获取注解中的内容)

反射包java.lang.reflect下的AnnotatedElement接口提供这些方法。

注意：只有@Retention注解被定义为RUNTIME时，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 接口是所有反射对象（Class、Method、Field、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方法来访问Annotation信息。

#### 常用方法(AnnotatedElement类)

以下案例只针对Field对象的注解操作，其他反射对象类似。

**1、isAnnotationPresent()**

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。

annotationClass：注解的字节码对象。

例：

// 可用于判断field成员变量上是否包含@Annotation02注解。

field.isAnnotationPresent(Annotation02.class)

**2、getAnnotation()**

<T extends Annotation> T **getAnnotation**(Class<T> annotationClass)

// 返回该程序元素上存在的、指定类型的注解对象，如果该类型注解不存在，则返回null。

例：

// 获取当属性反射对象field指定的注释对象

Annotation02 annotation = field.getAnnotation(Annotation02.class);

**3、getAnnotations()**

Annotation[] **getAnnotations**()

// 返回该程序元素上存在的所有注解，若没有注解，返回长度为0的数组。

**4、getAnnotationsByType()**

<T extends Annotation> T[] **getAnnotationsByType**(Class<T> annotationClass)

返回该程序元素上存在的、指定类型的注解数组。没有注解对应类型的注解时，返回长度为0的数组。该方法的调用者可以随意修改返回的数组，而不会对其他调用者返回的数组产生任何影响。

与getAnnotation区别：

getAnnotationsByType方法与 getAnnotation的区别在于，getAnnotationsByType会检测注解对应的重复注解容器。若程序元素为类，当前类上找不到注解，且该注解为可继承的，则会去父类上检测对应的注解。

**5、getDeclaredAnnotation()**

<T extends Annotation> T **getDeclaredAnnotation**(Class<T> annotationClass)

返回直接存在于此元素上的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注释。如果没有注释直接存在于此元素上，则返回null

**6、getDeclaredAnnotationsByType()**

<T extends Annotation> T[] **getDeclaredAnnotationsByType**(Class<T> annotationClass)

返回直接存在于此元素上的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注释。

**7、getDeclaredAnnotations()**

Annotation[] getDeclaredAnnotations()

返回直接存在于此元素上的所有注解及注解对应的重复注解容器。与此接口中的其他方法不同，该方法将忽略继承的注解。如果没有注释直接存在于此元素上，则返回长度为零的一个数组。该方法的调用者可以随意修改返回的数组，而不会对其他调用者返回的数组产生任何影响

**重点**：由于上面的方法来自AnnotatedElement接口，而反射对象（Class、Method、Field、Constructor等）是它的子接口，所以**想要使用上面的方法，必须先获取指定的反射对象**

**8、获取注解属性值**

**annotationObj.注解属性名()**

直接使用getAnnotation()得到的注解对象调用属性。

例：

String name = annotation.name();

// annotation是一个注解对象；name是注释中的属性名称。

#### 案例

需求：

定义一个Annotation02的注解，该注解只能在类的成员变量上面添加。创建Annotation02\_Test01类，给属性name添加注解，并在注解中指定值，通过反射进行获取该注解的值，并赋值给成员变量。最终使用成员变量的get方法读取值。

【注解】

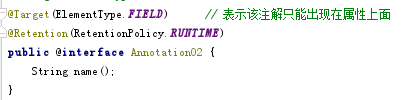


图 10定义注解

【被注解的类】



图 11 被注解的类

【获取注解的属性值】



图 12 获取注解的属性值

【调用被注解类中的成员变量】

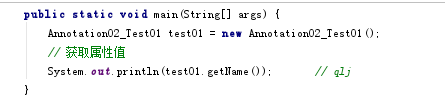


图 13 调用被注解类中的成员变量

## Java8新特性

### lambda表达式

#### 概念

实际上就是在不直接interface接口的情况下，实现函数式接口中的那个唯一方法。

能够使用Lambda表达式的一个重要依据是必须有相应的函数接口。所谓函数接口，是指内部有且仅有一个抽象方法的接口。

Lambda表达式的另一个依据是类型推断机制。在上下文信息足够的情况下，编译器可以推断出参数表的类型，而不需要显式指名。

#### 基本语法

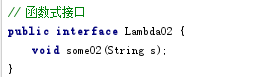
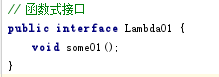
**(参数列表)->表达式** 或 **(parameters)->{代码块}**

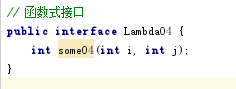
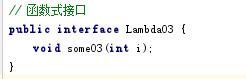
解释：

|  |  |
| --- | --- |
| paramaters： | 类似方法中的形参列表，这里的参数是函数式接口里的参数。这里的参数类型可以明确的声明，也可以不声明而由JVM隐含的推断。另外当只有一个推断类型时可以省略掉圆括号。 |
| **->：** | 可理解为“被用于”的意思 |
| 方法体： | 可以是表达式也可以代码块，是函数式接口里方法的实现。代码块可返回一个值或者什么都不返回，这里的代码块等同于方法的方法体。如果是表达式，也可以返回一个值或者什么都不返回。 |

#### 实现

接口：





lambda表达式实现：



语法精简：

1、参数类型可以省略，如果需要省略，参数列表中的每个参数的类型都要省略；

2、参数的小括号里面只有一个参数，那么小括号可以省略；

3、如果方法体当中只有一句代码，那么大括号可以省略；

4、如果方法体中只有一条return语句，那么大括号可以省略，且去掉return关键字。

#### 变量作用域

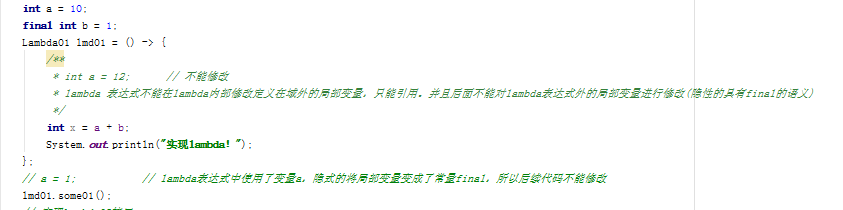
1、访问局部变量

①可以直接在lambda表达式中访问外层的局部变量。但是和匿名对象不同的是，lambda表达式 的局部变量可以不用声明为final。

②在Lambda表达式当中被引用的变量的值不可以被更改。

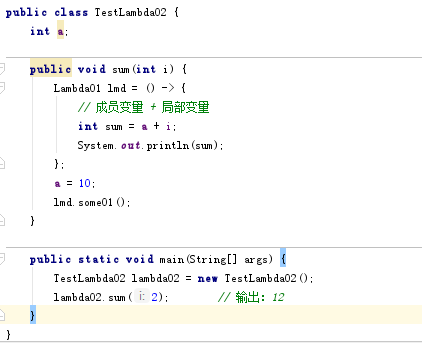
如果lambda表达式中使用了局部变量，则实际上是隐式的将局部变量变成了常量final，所 以后续代码不能修改。

③在Lambda表达式当中不允许声明一个与局部变量同名的参数或者局部变量。



2、访问成员变量

前提条件：lambda表达式必须在实例方法中，并且。

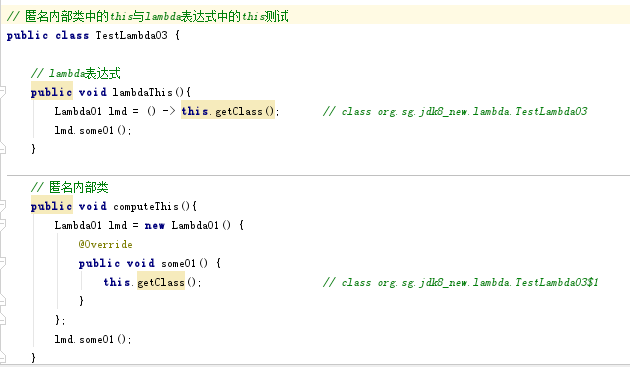


3、不能访问接口的默认方法

Lambda表达式中是无法访问到默认方法的。

4、Lambda表达式中使用this会引用创建该Lambda表达式的方法的this参数。

与匿名内部类相同，匿名内部类与lambda表达式中使用this都表示其所在的类对象。



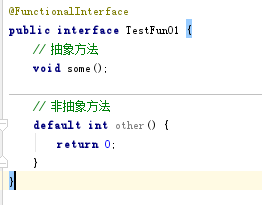
### 函数式接口

#### 概念

函数式接口是只包含一个抽象方法的接口，但也可以包含多个非抽象方法(default修饰)。函数式接口有时候被称为SAM类型，意思是单抽象方法。

一般来说，这个抽象方法指明了接口的目标用途。因此，函数式接口通常表示单个动作。

例如：标准接口Runnable是一个函数式接口，因为它只定义了一个方法run()；因此，run()定义了Runnable的动作。



#### 函数式接口注解

**@FunctionalInterface**

可以在任意函数式接口上面使用@FunctionalInterface来标识它是一个函数式接口，但是该注解不是强制的。

当接口被该注释修饰后，表示该接口只能是函数式接口，不符合函数式接口规范则编译报错。

### 方法引用

#### 概念

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。计算时，方法引用会创建函数式接口的一个实例。

**当Lambda表达式中只是执行一个方法调用时**，不用Lambda表达式，直接通过方法引用的形式可读性更高一些。方法引用是一种更简洁易懂的Lambda表达式。

<lambda表达式方法体中只是执行一个方法调用时才会用方法引用>

作用：

方法引用的唯一用途是支持Lambda的简写。

方法引用提高了代码的可读性，也使逻辑更加清晰。（优点）

#### 使用方式

使用**::**操作符将方法名和对象或类的名字分隔开。

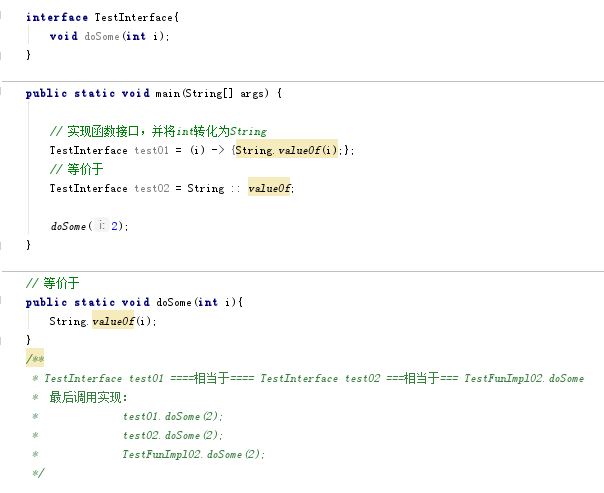
“::”是域操作符（也可以称作定界符、分隔符）。

表格1方法引用与lambda等价关系

|  |  |  |
| --- | --- | --- |
| 方法引用 | 等价的lambda表达式 | 分类 |
| String::valueOf | x->String.valueOf(x) | 静态方法引用 |
| Object::toString | x->x.toString() | 实例方法引用 |
| obj::toString | ()->obj.toString() | 实例方法引用 |
| ArrayList::new | ()->newArrayList<>() | 构造方法引用 |

1、静态方法引用

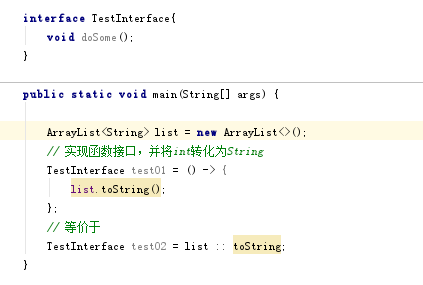
静态方法引用比较容易理解，和静态方法调用相比，只是把.换为::



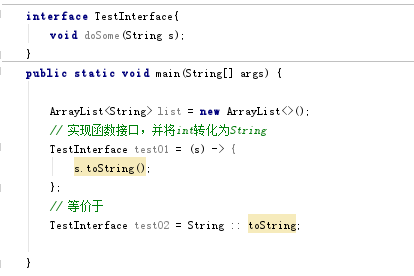
2、实例方法引用

这种语法与用于静态方法的语法类似，只不过这里使用对象引用而不是类名。

①接口中抽象方法无参



②接口中抽象方法有参

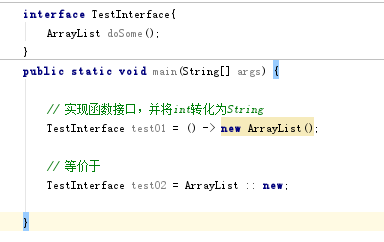


3、构造方法引用

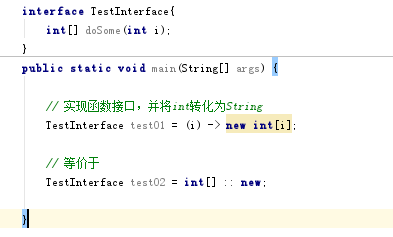
构造方法引用又分构造方法引用和数组构造方法引用。

构造函数本质上是静态方法，只是方法名字比较特殊，使用的是new关键字。

①构造方法引用（也可以称作构造器引用）



②数组构造方法引用



### Stream

一个序列的元素支持顺序和并行聚合操作。

Stream是元素的集合，这点让Stream看起来用些类似Iterator；

可以支持顺序和并行的对原Stream进行聚合（聚合也可以理解为聚汇、合并）的操作。

理解：

可以把Stream当成一个高级版本的Iterator。原始版本的Iterator，用户只能一个一个的遍历元素并对其执行某些操作；高级版本的Stream，用户只要给出需要对其包含的元素执行什么操作，如“过滤掉长度大于10的字符串”、“获取每个字符串的首字母”等，具体这些操作如何应用到每个元素上，交给Stream的方法完成。

例： List<Integer> nums = Lists.new ArrayList(1,null,3,4,null,6);

nums.stream().filter(num -> num != null).count();

#### 使用Stream基本步骤

->创建Stream；

->转换Stream，每次转换原有Stream对象不改变，返回一个新的Stream对象（可以有多次转换）；

->对Stream进行聚合（Reduce）操作，获取想要的结果。

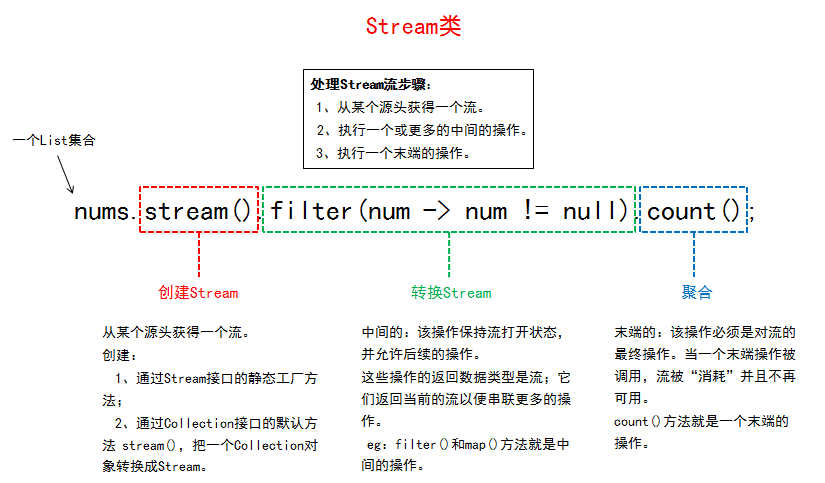


图14剖析Stream通用语法

红色框：Stream的生命开始的地方，负责创建一个Stream实例；

绿色框：把一个Stream流转换成另外一个Stream，红框的语句生成的是一个包含所有nums变量的Stream，进入绿框的filter方法以后，重新生成了一个过滤掉原nums列表所有null以后的Stream。

蓝色框：把Stream的里面包含的内容按照某种方法来汇聚成一个值。

#### parallelStream()

对于ParallelStream，需要知道的是里面的执行是异步的，并且使用的线程池是ForkJoinPool.common，可以通过设置-Djava.util.concurrent.ForkJoinPool.common.parallelism=N来调整线程池的大小；

parallelStream的作用：

Stream具有平行处理能力，处理的过程会分而治之，也就是将一个大任务切分成多个小任务，这表示每个任务都是一个操作

例：

List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);

numbers.parallelStream().forEach(System.out::print);//输出：635821749

注意：

得到的展示顺序不一定会是1、2、3、4、5、6、7、8、9，而可能是任意的顺序。得到的结论就是parallelStream()每次执行的结果都不相同，与多线程程序中执行的结果类似。如果希望最后顺序是按照原来Stream的数据顺序，那可以调用forEachOrdered()。

parallelStream与Stream区别：

stream()是一种串行流；

parallelStream()是一种并行流，支持并行执行，提高程序运行效率。但是如果使用不当可能会发生线程安全的问题。

parallelStream线程不安全

parallelStream的效率，因为是多线程，默认线程数量是计算器处理器的数量

#### 创建Stream

两种方式创建：

①通过Stream接口的静态工厂方法（Note：Java8里接口可以带静态方法）。

②通过Collection接口的默认方法stream()，把一个Collection对象转换成Stream。

1、Stream流中的静态方法创建

①of方法:

有两个，一个接受单一值，一个接受变长参数。

static <T> Stream <T> of(T t) //接受单一值

static <T> Stream <T> of(T...values) //接受变长参数

案例：

Stream<Integer> integerStream = Stream.of(1,5,9,5);

Stream<String> stringStream = Stream.of("qlj");

②generator方法

生成一个无限长度的Stream，其元素的生成是通过给定的Supplier（这个接口可以看成一 个对象的工厂，每次调用返回一个给定类型的对象）

static <T> Stream<T> generate(Supplier<T> s)

案例：

//匿名内部类、lambda、方法引用都可以

Stream.generate(()->Math.random());

Stream.generate(Math::random);

③iterate方法

生成无限长度的Stream，和generator不同的是，其元素的生成是重复对给定的种子值(seed) 调用用户指定函数来生成的。其中包含的元素可以认为是：seed，f(seed),f(f(seed))无限循环。

static <T> Stream<T> iterate(final T seed,final UnaryOperator<T> f)

案例：

//先获取一个无限长度的正整数集合的Stream，然后取出前10个打印。

Stream.iterate(1,item -> item + 1).limit(10).forEach(System.out::println);

此方法必须配合limit()方法使用，不然会无限打印下去。

2、通过Collection接口的默认方法stream()

Collection接口有一个stream方法，所以其所有子类都都可以获取对应的Stream对象。从List对象获取其对应的Stream对象。

default Stream<E> stream()

案例：

Stream<Integer> stream = list.stream();

这种方式创建Stream流经常被使用。

#### 转换Stream

转换Stream其实就是把一个Stream通过某些行为转换成一个新的Stream。

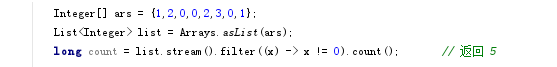
<转换后实际是一个新的Stream，所以支持多个方法进行多次转换>

1、性能问题：

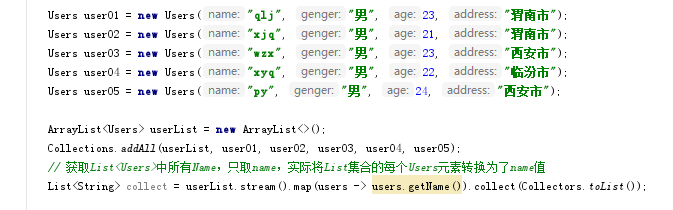
在对于一个Stream进行多次转换操作，每次都对Stream的每个元素进行转换，而且是执行多次，这样时间复杂度就是一个for循环里把所有操作都做掉的N（转换的次数）倍。但是事实上不是这样的，转换操作都是lazy(延迟)的，多个转换操作只会在聚合（reduce）操作的时候融合起来，一次循环完成。我们可以这样简单的理解，Stream里有个操作函数的集合，每次转换操作就是把转换函数放入这个集合中，在聚合操作的时候循环Stream对应的集合，然后对每个元素执行所有的函数。

2、常用转换过滤方法：

①filter()-->(过滤)排除所有与条件不匹配的元素。(与sql中的where类似)



②map()-->(转换)通过Function将每个元素映射转换为对应的结果。



③flatMap()-->通过FlatMapper将每个元素转变为无或更多的元素。

④peek()-->对每个遇到的元素执行一些操作。主要对调试很有用。

⑤distinct()-->(去重)根据equals行为排除所有重复的元素。这是一个有状态的操作。

⑥sorted()-->确保流中的元素在后续的操作中，按照比较器（Comparator）决定的顺序访问。 这是一个有状态的操作。

⑦limit()-->保证后续的操作所能看到的最大数量的元素。这是一个有状态的短路的操作。

⑧substream()-->确保后续的操作只能看到一个范围的（根据index）元素。像不能用于流的 String.substring一样。也有两种形式，一种有一个开始索引，一种有一个结束索引。二者都是 有状态的操作，有一个结束索引的形式也是一个短路的操作。

流操作的特性：

有状态的：有状态的操作给流增加了一些新的属性，如:元素的唯一性，或者元素的最大数量，或者保证元素以排序的方式被处理。这些典型的要比无状态的中间操作代价大。

短路：短路操作潜在的允许对流的操作尽早停止，而不去检查所有的元素。这是对无限流的一个特殊设计的属性；如果对流的操作没有短路，那么代码可能永远也不会终止。

#### Reduce（聚合）Stream

聚合（也称为折叠）接受一个元素序列为输入，反复使用某个合并操作，把序列中的元素合并成一个汇总的结果。

1、聚合分类

可变聚合：把输入的元素们累积到一个可变的容器中，eg:Collection或者StringBuilder。

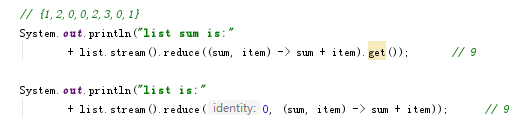
其他聚合：除去可变汇聚剩下的，一般都不是通过反复修改某个可变对象，而是通过将前一次的 聚合（汇聚）结果当成下一次的入参，反复如此。eg:reduce()，count()，allMatch()。

2、常见聚合方法

①forEach()-->对流中的每个元素执行一些操作。

②toArray()-->将流中的元素倾倒入一个数组。

③reduce()-->通过一个二进制操作将流中的元素合并到一起。



④collect()-->把Stream中的所有元素收集到一个结果容器中，eg：Collection或Map。



Collector的工具类Collectors：

Collectors.toList()收集到List中；

Collectors.toSet()收集到Set中。

⑤min()-->根据一个比较器找到流中元素的最小值。

⑥max()-->根据一个比较器找到流中元素的最大值。

⑦count()-->计算流中元素的数量。

⑧anyMatch()-->判断流中是否至少有一个元素匹配断言。这是一个短路的操作。

⑨allMatch()-->判断流中是否每一个元素都匹配断言。这是一个短路的操作。

⑩noneMatch()-->判断流中是否没有一个元素匹配断言。这是一个短路的操作。

⑪findFirst()-->查找流中的第一个元素。这是一个短路的操作。

⑫findAny()-->查找流中的任意元素，可能对某些流要比findFirst代价低。这是一个短路 的操作。

**注意**：sum()等方法不是所有的Stream对象都有的，只有IntStream、LongStream和DoubleStream 是实例才有。

* 1. **其他常用类**
     1. **随机数**

类导入：importjava.util.Random;

类方法：nextInt(参数1)

// 随机产生int类型范围内的数字

// 参数1指随机数的最大值（不包含该值）

其他方法：nextDouble()、nextFloat()、nextBoolean()等

【例1】

Random random = new Random();

int num01 = random.nextInt(); // 随机产生int类型范围内的数字

【例2】

int num02 = random.nextInt(101); // 随机生成[0,101)的int类型数据

* + 1. **泛型**

1、泛型是一种语法机制，只在程序编译起到作用（运行阶段泛型没有！）

2、泛型的优点：

-> 集合中存储的元素类型统一

-> 从集合中取出的类型是泛型指定的类型，不需要进行大量的向下转型

缺点：

-> 集合类型缺乏多样性

3、钻石表达式（自动类型推断机制）jdk7后才可以使用

// List<Animal> myArrayList03 = new ArrayList<>() 可以省略<>中的内容

4、自定义泛型

①自定义泛型类：类中的属性的类型可以有很多种。

只能在成员变量上使用，只能使用引用类型。

【例】List、Set、Map等类上。

②自定义泛型方法：方法的参数类型可以有很多种。

只能在抽象方法上使用

【例】

class Some<AAAAA>{

//实际是将数据类型参数化（让调用者去指定实参的数据类型）

//调用者不指定类型时，默认为Object类型

public void doSome(AAAAA obj){

System.out.println("调用doSome()方法!");

}

}

### Arrays

Arrays类位于java.util包中，主要包含了操纵数组的各种方法

注：Arrays类中的方法都是static修饰的静态方法,使用的时候可以直接使用类名进行调用

#### Arrays.toString() //快速输出数组内容

例：

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

System.out.println(Arrays.toString(a));

//Arrays类中重写了toString()方法。

#### Arrays.sort() //给数组排序(默认升序)

例：

int[] a={3,2,6,4,1,5};

Arrays.sort(a); // 升序

System.out.println(Arrays.toString(a)); // [1,2,3,4,5,6]

#### Arrays.asList() //数组转换为集合(列表)

例：

Integer[] num = {1,2,3,4,5};

List<Integer> ints = Arrays.asList(num);

注意：asList()中的参数必须是引用类型数组，asList()源码中实际创建了一个ArrayList对象。

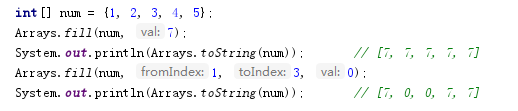
#### Arrays.copyOf() //复制数组

例：

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

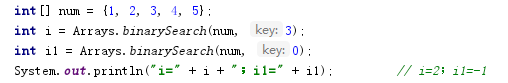
int[] copy = Arrays.copyOf(num,10); // [1,2,3,4,5,0,0,0,0,0]

#### Arrays.fill() //数组填充



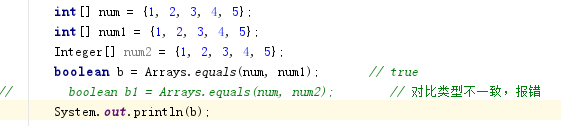
注意：fill()修改的参数与数组元素类型必须一致。

#### Arrays.binarySearch() //查找数组元素，返回索引



注意：查询不到对应值返回任意负数。

#### Arrays.equals() //判断两个数组是否相等（比较数组长度与元素值都相等）

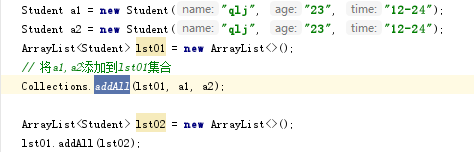


### Collections

Collections类是Java提供的一个操作Set、List和Map等集合的工具类。Collections类提供了许多操作集合的静态方法，借助这些静态方法可以实现集合元素的排序、查找替换和复制等操作。

#### addAll() //给集合对象批量添加元素

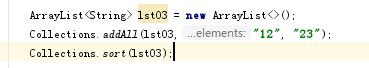
static <T> boolean addAll(Collection <? super T> c, T...elements)



注意：Collections类中的addAll()方法与集合自身的addAll()方法不同。

#### sort() //集合排序（从小到大）

static <T extends Comparable<? super T>> void sort(List<T> list)



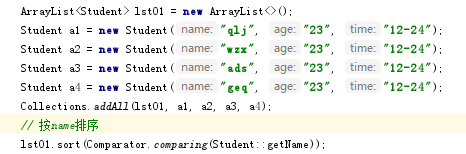
注意：

使用sort()方法时，传入的list集合中的元素必须是一个继承了Comparable类型的元素 (例： Integer、String、...)，

public static <T> void sort​(List<T> list, Comparator<? super T> c)

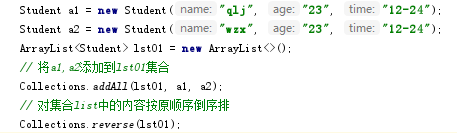
也可以传入一个比较器参数，此处不做深入。

【集合自身排序方法】



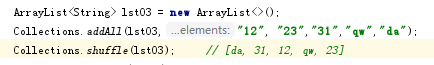
#### reverse() //对集合list中的内容按原顺序倒序排列

static void reverse(List<?> list)



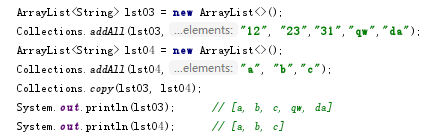
#### shuffle() //对集合list中的内容随机进行排序

static void shuffle(List<?> list)



#### copy() //将集合src中的元素全部复制到dest中,并且覆盖相应索引的元素

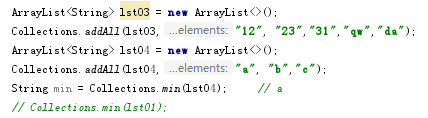
static <T> void copy(List <? super T> dest,List<? extends T> src)



注意：dest.size()>=src.size()

#### min()/max() //获取最小值/最大值

static <T extends Object & Comparable <? super T>> T min(Collection <? extends T> coll)



注意：list集合元素必须继承Comparable。也可以传入一个比较器参数，此处不做深入。

### 枚举(Enum)

#### 概述

1、enum是一个修饰关键字，相当于class、interface。

实际上Enum是一种受限制的类，并且具有自己的方法。

2、枚举类一般表示一组常量集。

在没有enum枚举的时候，java一般使用final关键字进行常量定义

例：public static final int i = 1;

3、Java枚举类使用enum关键字来定义，各个常量使用逗号','来分割。

4、枚举类的实质：创建enum时，编译器会为你生成一个相关的类，这个类继承java.lang.Enum

java.lang.Enum类声明

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

implements Comparable<E>, Serializable { ... }

#### 作用

1、出于类型安全考虑，没用枚举类之前，常用静态常量来表示。

例：public static final int WOMAN = 0;

public static final int MAN = 1;

2、代码更优雅、易读

在大项目中静态常量会有很多，如果都使用final定义在一个类中，容易造成命名混乱。

3、枚举类可以帮助我们定义所需的类型

枚举易于记忆和使用，相当于一个接口。使用时，只需封装内部数据类型并限制数据字段。此外， 可以为不同的枚举变量调用不同的处理方法（这可以通过实现枚举类的抽象方法来实现）。

#### 枚举类声明

声明枚举时必须使用 enum 关键字，然后定义枚举的名称、可访问性、基础类型和成员等

任意两个枚举成员不能具有相同的名称，多个枚举成员之间使用逗号分隔。

【语法】

**enum-modifiers enum enumname {**

**enum-body1, enum-body2 ...;**

**}**

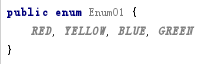
其中：

enum-modifiers表示枚举的修饰符主要包括public、private和internal；enumname表示声 明的枚举名称； enum-body表示枚举的成员，它是枚举类型的命名常数。

相当于：

public class EnumTest extends Enum{}

【例】



实际上底层相当于创建了四个Enum对象：

new Enum<EnumTest>("RED", 0);

new Enum<EnumTest>("YELLOW", 1);

new Enum<EnumTest>("BLUE", 2);

new Enum<EnumTest>("GREEN", 3);

注意：虽然底层Enum类存在惟一的构造方法，但是程序员不被允许调用此方法，即不能new对象。

构造方法：protected Enum​(String name, int ordinal)

name：此枚举常量的名称，它是用于声明它的标识符。

ordinal：这个枚举常数的序数（它在枚举声明中的位置，其中初始常数被分配为零的序数）。

**内部类声明枚举：**

与class的内部类效果一致。，此处不做赘述。

#### 使用枚举常量

语法：枚举类名.枚举成员名

例：Enum01.RED、Enum01.YELLOW

#### 枚举类成员

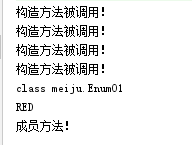
枚举跟普通类一样可以用自己的变量、方法和构造函数，构造函数只能使用private或者省略访问修饰符，所以外部无法调用。

枚举既可以包含具体方法，也可以包含抽象方法。如果枚举类具有抽象方法，则枚举类的每个实例都必须实现它。

【例】



【输出】



注意：

1、构造方法依然是被调用了，也就是说每个枚举成员(RED、YELLOW、...)底层都是通过构造方法创 建的。

2、构造方法创建时必须使用private修饰或者省略。

3、想要使用成员实例方法必须通过Enum对象进行调用，而Enum的构造方法不允许被外界获取并创 建Enum对象，因为底层在定义枚举成员时就已经创建了对象的Enum对象，所以直接调用枚举成员就 可以获取Enum对象(Enum01.RED)。

#### 详解Enum枚举成员及构造方法

上面的Enum01类方法中枚举成员分别是：

RED, YELLOW, BLUE, GREEN;

// 每个枚举成员都相当于一个枚举对象，而每个枚举对象中并没有实际的参数，实际上相当于底层 调用了一个无参的构造方法，相当于：new String()；只是这个Enum对象名称叫做RED。

**如何对Enum枚举成员对象传参？**

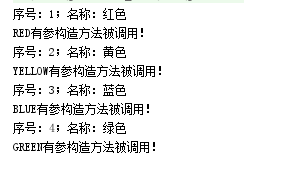
例：RED(1,"红色"), YELLOW(2,"黄色"), BLUE(3,"蓝色"), GREEN(4,"绿色");

注：传参后必须创建对应参数类型的构造方法，否则无法创建Enum对象，程序报错。

【例】：



【输出】：



注意：

1、枚举对象的参数类型必须与构造方法中的参数类型一致；

2、构造方法中的参数名称是任意名称，由于上面的Enum类中并没有显示的定义成员变量，所以 构造方法中不需要通过形参列表给成员变量赋值。

3、由于上面案例没有成员变量，所以如果Enum类中定义方法时不可直接使用枚举对象的常量值。 要想在Enum类中使用定义的枚举常量，必须定义成员变量，用来接收枚举对象的常量。

改进上面的案例：



#### 枚举常用方法

1、**valueOf()**

// 返回具有指定名称的指定枚举类型的枚举常量。

static <T extends Enum<T>> T **valueOf**(Class<T> enumType, String name)

enumType ：jvm进行编译的时候添加的，使用时不需要添加。

name：枚举成员对象名称。

例：Enum01.valueOf("RED"); // RED

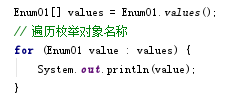
2、**values()**

// 将枚举类转变为一个枚举类型的数组

作用：

枚举中没有下标，没有办法通过下标来快速找到需要的枚举类，这时候，转变为数组之后， 就可以通过数组的下找到需要的枚举类。

注：values()方法是一个特殊方法，在Enum的API文档中没有定义该方法，通过反射class文 档，可以看出底层实际是由编辑器添加的一个方法



3、**name()、toString()**

// 返回当前枚举对象的名称。

例：Enum01.RED.name(); // RED

Enum01.BLUE.toString(); // BLUE

4、**ordinal**()

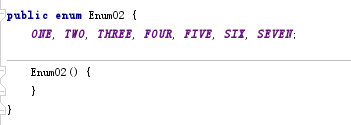
// 返回实例声明时的次序，从0开始

final int **ordinal**()

注：枚举类会给所有的枚举变量一个默认的次序，是根据我们定义的次序来排序的。

例：red.ordinal(); // 0

#### 在switch 中使用枚举类





## 小知识点

### idea设置

1、切换主题

Editor > Color Scheme > Scheme

2、鼠标悬停提示方法注释

Editor -> General -> 勾选Show quick documentation on mouse move

3、自动优化导包（自动删除、导入包）

Editor>General>Auto Import



4、IDEA给选中内容添加双引号""

Editor > General > Smart Keys > 勾选Surround selection on typing quote or brace

5、鼠标滚轮调节字体大小

Editor > General > 勾选Change font size with Ctr+Mouse Wheel

6、修改快捷键(Eclipse)

Keymap => Keymaps改为 Eclipse copy

7、显示行号 ， 方法和方法间的分隔符

Editor > General > Appearance > 勾选Show line numbers 和 Show method separators

8、忽略大小写

Editor > General > code Completion > 不勾选Match case

9、设置项目文件编码

Editor > File Encodings

10、自动编译

Build > Compiler > 勾选Build Project automatically 和 Compile independent modules ...

### 快捷键

keymap=idea default

1、ctrl+f12 // 当前类的所有属性、方法(悬浮窗)

2、ctrl+H // 显示类结构图（类的继承层次）

3、alt+/ // 快速补全代码

4、ctrl+alt+L // 快速格式化代码

5、ctrl+alt+T // 将选中的代码使用 if、while、try/catch 等代码块包装起来（功能强大）

6、ctrl+alt+delete // 去除相关的代码块包装，是Ctrl + Alt + T的相反操作