目录

[数据类型 5](#_Toc117608129)

[基本数据类型 5](#_Toc117608130)

[浮点型数据精度问题 6](#_Toc117608131)

[引用数据类型 6](#_Toc117608132)

[万物之父，Object类 7](#_Toc117608133)

[基本数据类型和引用数据类型的区别和共同处 8](#_Toc117608134)

[数据的比较equals和== 8](#_Toc117608135)

[字符串常量池和基本数据类型缓冲池 9](#_Toc117608136)

[Integer缓冲池 9](#_Toc117608137)

[String 缓冲池 10](#_Toc117608138)

[基本数据类型的装箱与拆箱与转换问题 10](#_Toc117608139)

[八种基本数据类型的封装类型 10](#_Toc117608140)

[装箱与拆箱 11](#_Toc117608141)

[常用方法 12](#_Toc117608142)

[java堆栈和常量池 13](#_Toc117608143)

[数据上下限以及数据的溢出问题 13](#_Toc117608144)

[原码反码和补码 13](#_Toc117608145)

[数据的上下限 14](#_Toc117608146)

[数据溢出问题 14](#_Toc117608147)

[数据的位运算\*\*\* 15](#_Toc117608148)

[数据在计算机中的存储方式以及计算方式 16](#_Toc117608149)

[带符号位运算>>,<< 16](#_Toc117608150)

[无符号位运算>>> 18](#_Toc117608151)

[java字符串 18](#_Toc117608152)

[字符串String，StringBuffer,StringBuilder 18](#_Toc117608153)

[String类的常用方法 19](#_Toc117608154)

[流程控制以及关键字 21](#_Toc117608155)

[逻辑运算和短路运算 22](#_Toc117608156)

[访问权限 22](#_Toc117608157)

[public 22](#_Toc117608158)

[protected 23](#_Toc117608159)

[private 23](#_Toc117608160)

[异常 23](#_Toc117608161)

[Exeception 23](#_Toc117608162)

[泛型 24](#_Toc117608163)

[泛型类 24](#_Toc117608164)

[泛型接口 24](#_Toc117608165)

[泛型方法 24](#_Toc117608166)

[Java Arrays类，数组 25](#_Toc117608167)

[初始化 25](#_Toc117608168)

[Java比较器 26](#_Toc117608169)

[Java IO流\*\*\* 27](#_Toc117608170)

[File类 28](#_Toc117608171)

[构造方法 28](#_Toc117608172)

[判断功能 28](#_Toc117608173)

[获取功能 29](#_Toc117608174)

[IO流 29](#_Toc117608175)

[字节流: 30](#_Toc117608176)

[FileInputStream 31](#_Toc117608177)

[OutputStream 31](#_Toc117608178)

[FileOuputStream 32](#_Toc117608179)

[字符流 34](#_Toc117608180)

[转换流 37](#_Toc117608181)

[内存输出流 38](#_Toc117608182)

[对象操作流 39](#_Toc117608183)

[随机读写流 40](#_Toc117608184)

[打印流 40](#_Toc117608185)

[JAVA异常处理机制 40](#_Toc117608186)

[JAVA并发\*\*\* 41](#_Toc117608187)

[基础知识补充： 41](#_Toc117608188)

[线程的生命周期 46](#_Toc117608189)

[实现线程的三种方式 48](#_Toc117608190)

[基础线程机制 48](#_Toc117608191)

[线程基础补充 48](#_Toc117608192)

[主线程的基本使用: 49](#_Toc117608193)

[线程的终止: 49](#_Toc117608194)

[线程的常用方法和基本属性 49](#_Toc117608195)

[线程的状态 50](#_Toc117608196)

[守护线程 51](#_Toc117608197)

[线程同步问题 51](#_Toc117608198)

[JAVA并发编程的三个核心思想 51](#_Toc117608199)

[互斥锁 52](#_Toc117608200)

[方法锁，类锁和对象锁 52](#_Toc117608201)

[synchronized的两种用法 53](#_Toc117608202)

[线程死锁 54](#_Toc117608203)

[死锁产生的必要条件 55](#_Toc117608204)

[释放锁 55](#_Toc117608205)

[JUC Lock锁 56](#_Toc117608206)

[synchronized和lock的对比 56](#_Toc117608207)

[线程的协作 57](#_Toc117608208)

[线程的通信-分析 57](#_Toc117608209)

[生产消费者问题 58](#_Toc117608210)

[线程池 58](#_Toc117608211)

[使用线程池 59](#_Toc117608212)

[JAVA反射，注解和代理 59](#_Toc117608213)

[JAVA反射机制 59](#_Toc117608214)

[静态语言和动态语言 59](#_Toc117608215)

[JAVA Reflection 60](#_Toc117608216)

[Class类详解 61](#_Toc117608217)

[注解 69](#_Toc117608218)

[内置注解 69](#_Toc117608219)

[元注解 69](#_Toc117608220)

[自定义注解 70](#_Toc117608221)

[代理 71](#_Toc117608222)

[静态代理 71](#_Toc117608223)

[动态代理 71](#_Toc117608224)

[JDBC 71](#_Toc117608225)

[JVM虚拟机\*\*\* 73](#_Toc117608226)

[JAVA内存分析 73](#_Toc117608227)

[类的加载过程 74](#_Toc117608228)

[什么时候会发生类的初始化 76](#_Toc117608229)

[JAVA GC 78](#_Toc117608230)

[Lamda表达式 79](#_Toc117608231)

[lamda表达式作用和条件 79](#_Toc117608232)

[函数式接口 79](#_Toc117608233)

[正则表达式 80](#_Toc117608234)

[概述 80](#_Toc117608235)

[单个符号和逻辑符号 80](#_Toc117608236)

[匹配次数的符号 80](#_Toc117608237)

[特殊符号 81](#_Toc117608238)

[快捷符号 81](#_Toc117608239)

[常用正则表达式和匹配方法 81](#_Toc117608240)

[Java容器\*\*\* 81](#_Toc117608241)

[Collection 82](#_Toc117608242)

[List接口 82](#_Toc117608243)

[Set接口 84](#_Toc117608244)

[Map接口 87](#_Toc117608245)

[HashMap 87](#_Toc117608246)

[LinkedHashMap 87](#_Toc117608247)

[TreeMap 87](#_Toc117608248)

[JAVA枚举类 88](#_Toc117608249)

[JAVA API 89](#_Toc117608250)

[JSOUP爬虫 89](#_Toc117608251)

[jsoup简介 89](#_Toc117608252)

[相关概念简介 89](#_Toc117608253)

[快速入门 90](#_Toc117608254)

[网站的跨域问题 91](#_Toc117608255)

[TIKA爬虫 91](#_Toc117608256)

[GUI编程 91](#_Toc117608257)

[JAVA 学习总结 92](#_Toc117608258)

[JAVA发展史 92](#_Toc117608259)

[JAVA编程理念 92](#_Toc117608260)

[编程法则以及注意事项 93](#_Toc117608261)

# 数据类型

java变量的命名规则：必须以字母，下划线，美刀符号开头，变量名不能与关键字冲突

## 基本数据类型

1字节：boolean和byte 2字节：char和short

4字节：int和float 8字节：long和double

补充:

double(双精度型)和float(单精度型)

float精确到第6位，float变量的声明需要加f后缀

double精确到第15位

正负无穷大：使用java内置常数Double.POSITIVE\_INFINITY和Double.NEGATIVE\_INFINITY。

基本数据类型之间的运算：低字节的数据可以赋值给高字节的数据，因为不会数据溢出。并且数据类型会自动转化成高字节型数据。高字节不能直接赋值给低字节，因为会数据溢出，如果强制类型转化，会丢弃溢出数据。其他运算也是如此。

总结：低字节和高字节数据进行运算，低字节数据会转化成高字节数据。高字节强制转化成低字节数据会丢弃溢出的数据部分。

基本数据作为类的成员变量默认值:

* 静态类成员在类的链接过程就完成默认值的设置，这也是 为什么static修饰的变量可以直接通过类名.静态成员使用，而不需要初始化。非static类变量在类初始化阶段完成默认值的设置，具体请看JVM目录

1. 基本数据类型的整数类型(byte、short、int、long)默认值是0；
2. 基本数据类型的浮点类型(float、double)默认值是0.0；
3. 基本数据类型的字符类型(char)默认值是'\u0000'；
4. 基本数据类型的布尔类型(boolean)默认值是false；

### 浮点型数据精度问题

浮点数存在的意义：在计算机中，由于数据位数有限，所以存储的数值就肯定有上限，因此，对于极大或者极小的数，就采用科学计数法表示，所以就有了浮点数！

浮点数有单精度浮点数和双精度浮点数，二者异曲同工之妙，这里以单精度浮点数为例：

#### 浮点数的表示

单精度浮点数4字节32位：

* 首位表示符号位：0为正，1为负
* 后八位存储指数即阶码位：IEEE754标准中规定阶码位存储的是指数对应的移码，而不是原码或者补码！因为阶码存储的是指数的移码，所以指数和阶码之间就有这换算关系！假设指数真值为e,阶码位E，则E = e+2^(n-1)-1; n为存储阶码的位数！单精度中n为8。2^(n-1)是IEEE754标准中规定的偏移量！

偏移量为什么规定是2^(n-1)?

一个字节能表示的数值范围是-128~127之间，将其偏移到正数域，即为0~255之间，在计算机中，规定阶码全为0和全为1分别代表机器0和无穷大，所以取值范围要去除这两个特殊值，即1~254！如果偏移量不变还是128,1~254减去128，指数取值范围最大值就是126，很明显缩小了浮点数取值范围，所以规定了偏移量为2^(n-1)-1,所以8位二进制偏移量就是127！

* 尾数位23位用来存储有效数字：有效数字满足二进制科学计数法,即1<=|a|<2!

#### 浮点数的计算

数学运算中，对小数进行加减运算，需要将小数点对齐在进行运算操作！在计算机中，通过对阶码判断是否相等，来判断小数点是否对齐，这一操作称之为对阶操作！具体计算步骤如下：

* + 零值检测：检测两个操作数是否有0值(浮点数中阶码和尾数全为0表示0)，有0则可以直接得出结果
  + 对阶：判断两个数阶码值是否相等，不等说明两个数小数位没有对齐，需要通过移动尾数使阶码相等，即对阶！尾数右移一位，阶码值加一，反之减一！如果左移，高位溢出造成的误差会更大，所以IEEE754标准规定了对阶操作，尾数只能右移！
  + 运算：对阶完成后，直接按位相加，如果是负数则需要转化成补码运算！
  + 结果规格化：运算结果如果符合规格，即二进制科学计数法！则不用操作。如果不符合规格，则使尾数左右移动以达到规格。称之为左规，或者右规！
  + 结果舍入：在对阶过程或者右规时，尾数右移，最右端数据丢弃，从而导致精度损失！为减少损失，丢弃的数据会被保存，等规格化完成后再舍入！

综上，为避免数据的精度损失，推荐使用浮点数时使用双精度数据！

### 基本数据类型的装箱与拆箱与转换问题

#### 八种基本数据类型的封装类型

1字节: Boolean、Byte 2字节:Short、Character 4字节: Integer、Float 8 字 节:Long、Double

对象型包装类(Object直接子类): Character 和 Boolean

数值型包装类(Number直接子类): Byte,Short,Integer,Float,Long,Double

#### 装箱与拆箱

Number是一个抽象类，里面一共定义了6中方法(拆箱方法):

byteValue(), shortValue(), intValue(), floatValue(), longValue(), doubleValue()

拆箱与装箱: Integer temp = new Integer(10); //装箱，不推荐使用构造方法创建

Integer temp = Integer.valueof(10); //通过缓冲池创建效率更高

int temp1 = temp.intvalue(); //拆箱

封装类型是对象，即引用数据类型。

数值型包装类和它对应的数据类型比较时，会自动拆箱。

特殊情况:

Integer t = new Integer(10)(new对象形式创建);

Integer t1 = 10(字面量形式创建);

t==t1的返回值是false.与字符串常量池一样，非new生成的Integer对象，它的地址指向的是java常量池中的对象，而new生成的变量指向的堆中的对象，二者地址不同。

八种封装类型中，Byte,Short,Integer,Long,Character 的定义中都有一个缓存机制，

-128~127的对象会缓存到缓存中，调用valueof()方法时，会判断数据是否在这个范围，如果在，则返回缓存对象，如果超出范围，则新建一个对象返回。

在这个范围的数值，用==比较会返回true，因为都是从同一个缓存地址得到的。

java不能隐式向下转换(高字节赋值给低字节)

例如: float a = 1.1; //1.1的double类型，有数据溢出

正确写法: float a = 1.1f;

### 常用方法

基本数据类型的封装方法除了构造方法(弃用)外，一般都有两种静态方法(同一方法的两种重载)获取当前基本数据类型的封装类型。

#### 装箱方法

1. static valueof(基本数据类型)
2. static valueof(String)

//valueof方法创建的数据类型从时间和空间上效率更高，取代了构造方法创建的方式

实例如下：

Integer a = Integer.*valueOf*(1);  
Integer a1 = Integer.*valueOf*("1");  
  
Float f = Float.*valueOf*(1.0f);  
Float f1 = Float.*valueOf*("1.0f");  
  
Long l = Long.*valueOf*(100000);  
Long l1 = Long.*valueOf*("100000");  
  
Double d = Double.*valueOf*(100.0);  
Double d1 = Double.*valueOf*("100.0");

#### 拆箱方法

intValue(), byteValue(),floatValue()……………………………..

#### 字符串转化基本数据封装类型方法

把字符串转化成字符串内容对应的数据类型方法

static int parseInt(String s);

static float parseFloat(String s);

…………………

更多方法查询jdk

## 引用数据类型

类、数组、接口

类型的引用类型(类、数组、接口、String)默认值是null.

### 类

#### 类的定义

由访问级别、类型、类名、是否抽象、是否静态、泛型标识、继承或者实现关键字、父类或者接口组成！

类的访问级别：public 和 无符号！

类的类型：class,interface和enum

#### 类的构成

类由属性和方法组成，定义类时，首先定义属性。由于共有方法往往是开发者最关心的方法，所以共有方法首屏展示，其次就是保护方法，最后才是私有方法！

#### 接口和抽象类

从语法上二者的区别：

|  |  |  |
| --- | --- | --- |
| 语法维度 | 抽象类 | 接口 |
| 定义关键字 | abstract | interface |
| 子类继承或者实现关键字 | extends | Implements |
| 方法实现即方法体 | 可以有也可以没有 | 不能有 |
| 方法访问控制符 | 无限制 | 有限制，默认public abstract |
| 属性访问控制符 | 无限制 | 有限制，默认是public static final |
| 静态方法 | 可以有 | 不能有，JDK8以后可以有 |
| 静态代码块 | 可以有 | 不能有 |
| 本类型之间扩展 | 单继承 | 多继承 |
| 本类型之间扩展关键字 | extends | extends |

抽象类被继承体现的是is-a关系，接口被实现体现的是can-do关系！

很好理解：抽象类被继承，父类is a bird,子类 is an Object,而接口的实现体现can-do关系！

实现接口的子类必须要具备执行接口中定义的行为的能力，否则就是方法污染！

抽象类是模板设计，接口是契约式设计：

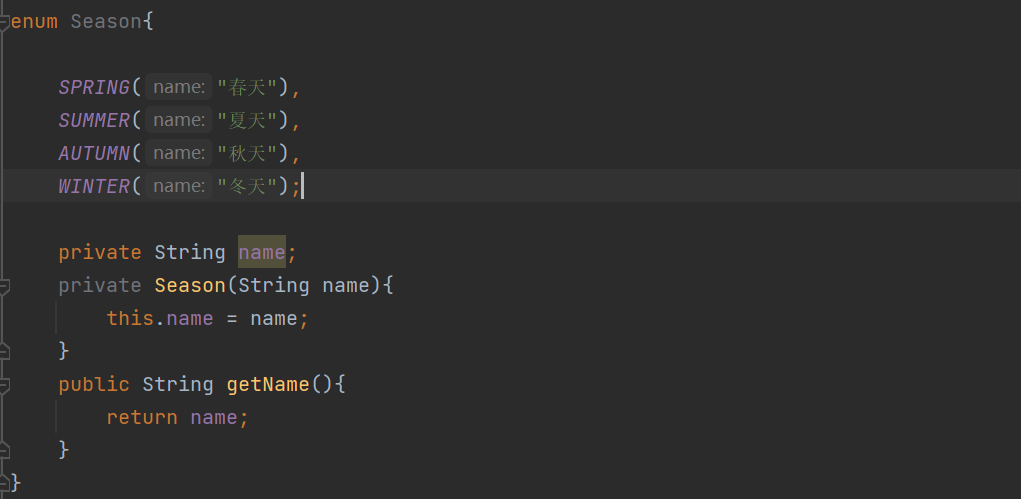
抽象类：一组物体所具有的共同特征！设计这个抽象类就是设计这一类物体的模板，是归纳行为！

接口：严格定义了属性方法的格式语法，任何物体都可以实现本接口，但是必须要遵守接口中定义的一切行为！是演绎行为！

综上，在java中抽象类的继承属于归纳行为，因此设计初严格要求继承是单继承，不能多继承！而实现则可以多种实现，属于演绎行为！

#### JAVA枚举类

针对于实例化对象是固定的类。例如季节的实例化对象只有春夏秋冬，性别对象只有男女，一周只有星期一到星期天等等。这时我们就需要枚举类。



使用enum关键字定义一个枚举类

#### 内部类

一个.java文件中，只能有一个类名与文件名相同的public修饰的类，该类作为该文件编译后的启动类。而任意类都可以在内部定义一个类，称之为内部类，内部类的声明定义和定义类一样！

内部类分为四种：

* + 静态内部类：类的所有对象共享
  + 成员内部类：作为类的一个属性
  + 匿名内部类：临时定义的一个内部类
  + 局部内部类：类的方法中或者表达式中定义的类

内部类的好处：

内部类可以访问外部类的所有静态属性和静态方法

通过外部类.内部类访问内部类

#### 类的实例化

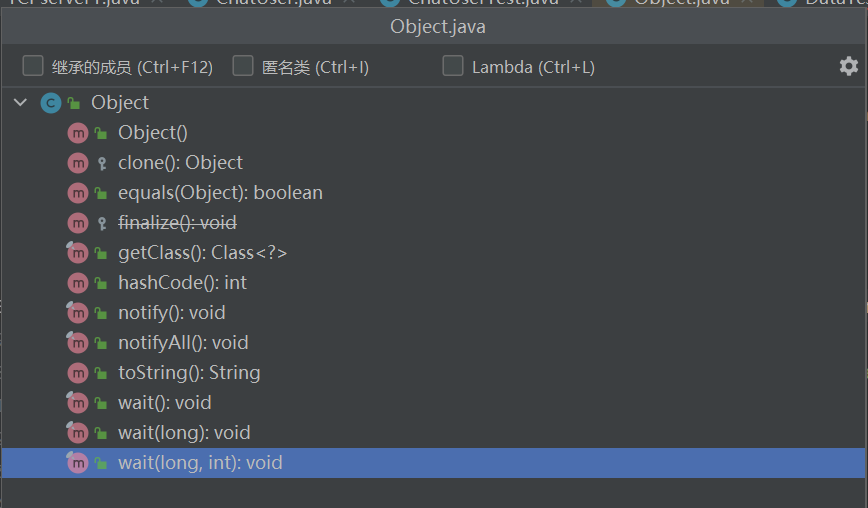
类的实例化使用到构造方法！

对象实例化，至少有一条从本类出发抵达Object的道路，而打通这条路的两个主要成员就是this和super，this指向自己，super指向父类

类的创建之初都有一个默认的空构造方法（如果父类没有构造方法就默认空构造方法），如果子类调用父类的构造方法，super就会不断向上源溯，如果没有指定，则默认调用super方法！

构造方法：构造方法是方法名和类名相同的方法，且没有返回值，不能被重写，不能被继承，不能被调用，可以被重载！构造方法可以私有化（枚举类的实现原理）

#### 万物之父Object类



父类方法解析：

1. Object:构造方法
2. clone:克隆方法
3. equals：比较方法
4. getClass:获取类方法，Java反射机制
5. hashcode：哈希码方法
6. notify:唤醒方法，用于线程同步
7. wait:等待方法，用于线程同步

JAVA创建对象的方式

1. new操作符创建对象
2. clone方法创建对象
3. java反射机制创建对象

clone涉及java浅拷贝和深拷贝，浅拷贝使用=对引用数据类型进行赋值就浅拷贝了这个引用数据类型的引用，使用clone方法拷贝引用数据类型既拷贝了引用，也在堆内存创建了一个新的该引用数据。

#### 类的关系

* 继承is-a
* 实现can-do
* 组合：类是成员变量
* 聚合：类是成员变量
* 依赖：
* 关联：

继承和实现关系很容易理解。

类关系中的组合是一种完全绑定的关系，所有成员共同完成一个任务，他们的生命周期是一样的。整体和部分之间同生共死！就好比头和身体一样都是人的一部分，虽然是一部分但是同生共死，二者完全不可分

聚合是可拆分关系，很松散的关系。部分可以被拆分给另一个整体。好比汽车的轮子和汽车一样，虽然是组合关系，但是汽车的轮子可以拆下来安装给另一辆车。

依赖关系是单向弱关系，类A用到类B，那么就是A依赖于B，这种关系是松散的，临时的。人养狗，狗这个类中包含主人这个类，狗依赖于人。

关联是平淡的相互依赖关系。人和信用卡之间的关系，人能使用信用卡，但是信用卡也绑定着人的信息。

#### 序列化

内存中的对象只有转化成二进制流才可以进行数据的持久化和网络传输，对象转化成二进制流就是序列化Serializable，反过来就是反序列化Deserialization。

java原生序列化：实现Serializable接口，该接口没有任何方法，起到标识作用。JAVA序列化保留了对象类的元数据以及对象数据，性能一般，不支持跨语言，性能一般。详细内容在码出高效p67页。

JSON序列化：码出高效p68页

Hessian序列化：码出高效p67页

## 基本数据类型和引用数据类型的区别和共同处

区别:

基本数据类型实际存放于栈内存中，内存存储的就是这个基本数据类型。

引用数据类型实际存放于堆内存中，但是它的引用存放在栈内存中，引用存储着指向它实际内容的地址(堆内存的地址)。

共同处:

不论是基本数据类型还是引用类型，他们都会先在栈中分配一块内存，对于基本类型

来说，这块区域包含的是基本类型的内容；而对于对象类型来说，这块区域包含的是

指向真正内容的指针，真正的内容被手动的分配在堆上。

## 数据的比较equals和==

区别：==是运算法，equals()是方法

对于==：比较基本数据类型，则比较数值是否相等。比较引用数据类型，则比较二者的内存地址是否相等。

对于equals方法：通过Object类的equals源码来分析。

public boolean equals(Object obj) {  
 return (this == obj);  
}

Object类的equals方法是通过==比较两个对象的地址是否相等。而一切类是继承Object类，在我们重写Object方法时，可以通过对象的成员变量值是否相等来比较两个对象是否相等。

equals方法只能用于比较对象，首先比较两个对象的地址是否相等(如果地址都相等那结果肯定是相等的)如果不相等，再可以通过二者的成员变量来进行比较。

所以equals方法是首先比较二者地址是否相等，如果不等再比较内容

## 字符串常量池和基本数据类型缓冲池

基本数据类型缓冲池如下:

boolean values true and false

all byte values

short values between -128 and 127

int values between -128 and 127

char in the range \u0000 to \u007F

使用基本数据类型对应的包装类型时，如果数据值在这个缓冲池范围中，就可以直接使用缓冲池中的对象。

例如：

new Integer(123) 使用new，每次都会新创建一个对象

Integer.valueof(123) 从缓冲池中获取对象，多次调用都是获取同一个引用。

使用基本数据类型的封装类时推荐优先使用valueof方法获取，通过缓冲池获取的值具有更加优异的空间和时间性能。

### Integer缓冲池

jdk 1.8 所有的数值类缓冲池中，Integer 的缓冲池 IntegerCache 很特殊，这个缓冲池的下界是 - 128，上界默认是 127，但是这个上界是可调的，在启动 jvm 的时候，通过

-XX:AutoBoxCacheMax= 来指定这个缓冲池的⼤⼩，该选项在 JVM 初始化的时候会设定⼀个名为 java.lang.IntegerCache.high 系统属性，然后 IntegerCache 初始化的时候就会读取该系统属性来决定上 界。

StackOverflow : Differences between new Integer(123), Integer.valueOf(123) and just 123

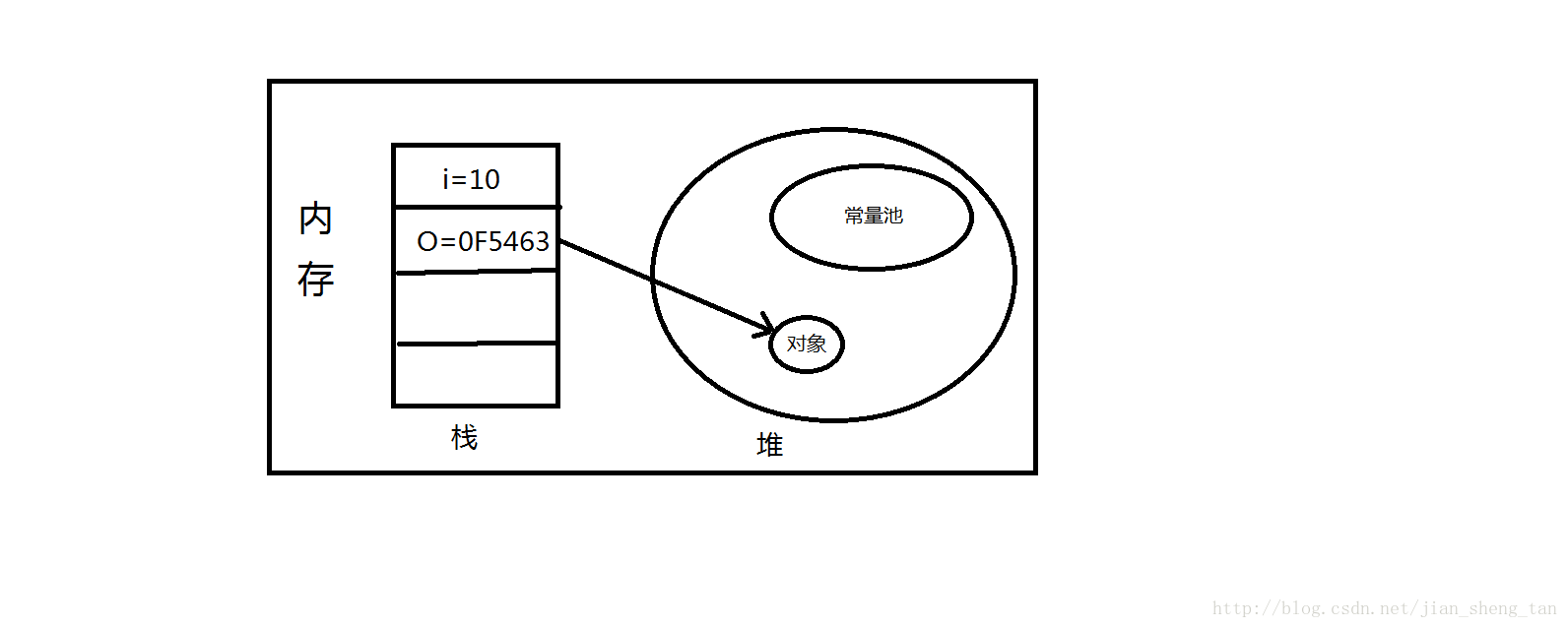
### String 缓冲池

通过字面量的形式创建字符串，字符串会在编译时被添加进入缓冲池以提高此类字符串的使用效率。不仅如此，还可以主动调用字符串的intern()方法将创建的字符串添加进入缓冲池！ 当⼀个字符串调⽤ intern() ⽅法时，如果 String Pool 中已经存在⼀个字符串和该字符串值相等（使⽤ equals() ⽅法进⾏确定），那么就会返回 String Pool 中字符串的引⽤；否则，就会在 String Pool 中添 加⼀个新的字符串，并返回这个新字符串的引⽤。

## java堆栈和常量池

java程序运行在JVM(java虚拟机)上，因此java的内存分配是在JVM中进行的，java程序运行涉及以下区域

* + - 1. 寄存器:JVM内部虚拟寄存器，存取速度非常快，程序不可控
      2. 栈:存放基本数据类型和对象的引用，对象本身存放在堆中
      3. 堆:存放new出来的对象，注意创建出来的对象只包含各自的承运变量，不包括成员方法。
      4. 常量池:存放常量，字面量，String还有基本数据类型的包装类，常量池位于堆中
      5. 代码段:存放从硬盘上读取的源程序代码
      6. 数据段:用于存放static修饰的静态成员变量



## 数据上下限以及数据的溢出问题

### 原码反码补码移码

在计算机中，一切数据由二进制数值表示，数据的正负由位数的首位表示，0为正，1为负！

为了应对减法以及正负0的问题，出现了反码和补码，在计算机中减法操作就是加负数，计算机采用补码进行数据的运算（所以负数在计算机中以补码形式存储）！因为在计算机中一切数据的本质都是0和1的指数幂组成，因此在计算机中一切数据都有一个上下限！

* 原码
  + 正数首位为0
  + 负数首位为1
* 反码
  + 正数和原码一致
  + 负数符号位不变，数值位取反
* 补码
  + 正数和原码一致
  + 负数符号位不变，数值位取反再加一
* 移码：每一个数值加上一个偏移常熟
  + 补码的符号位取反，其它不变！

### 数据的上下限

很明显，数据的上限和下限分别对应着正数和负数！以计算机一个字节八位二进制举例！

在计算机中数据的运算采用的是补码运算！

理论上一个字节数据的上下限：

正数的上限：01111111 = 2^7-1 = 127

负数的下限：11111111 = -2^7-1 = -127

实际上-127-1根据补码运算如下：



-127-1 = -128，而得到的结果为10000000(计算机数据溢出丢弃)，该数据是补码且无法计算得到它的原码，但是我们发现首位如果既用来表示符号位，也用来表示数值位的话，既可以满足算数需求，又不和符号发生冲突！因此规定八位二进制的下限就是-128，即10000000！,而计算机中一切数据都是2的指数倍个字节，所以数据上下限如下：

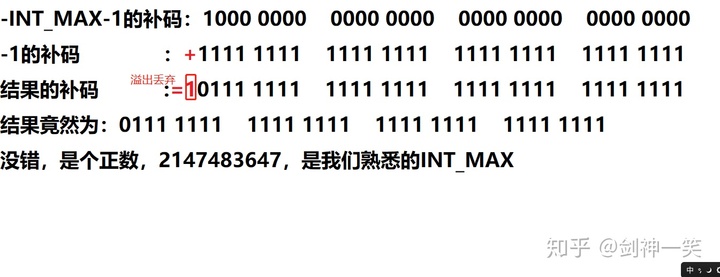
综上：假设数据的位数为x位二进制,去除首位符号位

* 数据的上限是：2^(x-1) -1
* 数据的下限是：-2^(x-1)

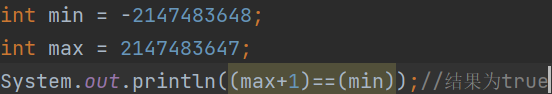
### 数据溢出问题

计算机中，数据的运算就是二进制按位运算，如果有数据溢出，则会丢失数据！

同理以一个字节的运算为例



数据的最小值减一会发生数据溢出，该数据就会变成最大值！



理解：

数据的上下限由数据的位数决定，如同一个钟表环（最大值也就是最小值），数据位数越多，这个环就越大。数据的定位只会在这个环中，当数据到达最大值或最小值时，再次增加或者再次减小就会发生数据的溢出（与运算的原理），由最大值变成最小值或者最小值变成最大值！

Integer.MAX\_VALUE+1 = Integer.MIN\_VALUE

Integer.MIN\_VALUE-1 = Integer.MAX\_VALUE

统称，数据的范围如同一个钟环，最大值也就是最小值处于0点位置，0处于6点位置。

数据类型的字节数越多，环就越大，数据取值范围也就在这个环内！数据最大值增大和数据最小值减小都会发生溢出，相当于带符号位运算！

天地之间(道一)，阴阳(二)相生相克相互转化(三)！

道生一，一生二，二生三，三生万物

## 数据的位运算\*\*\*

### 数据在计算机中的存储方式以及计算方式

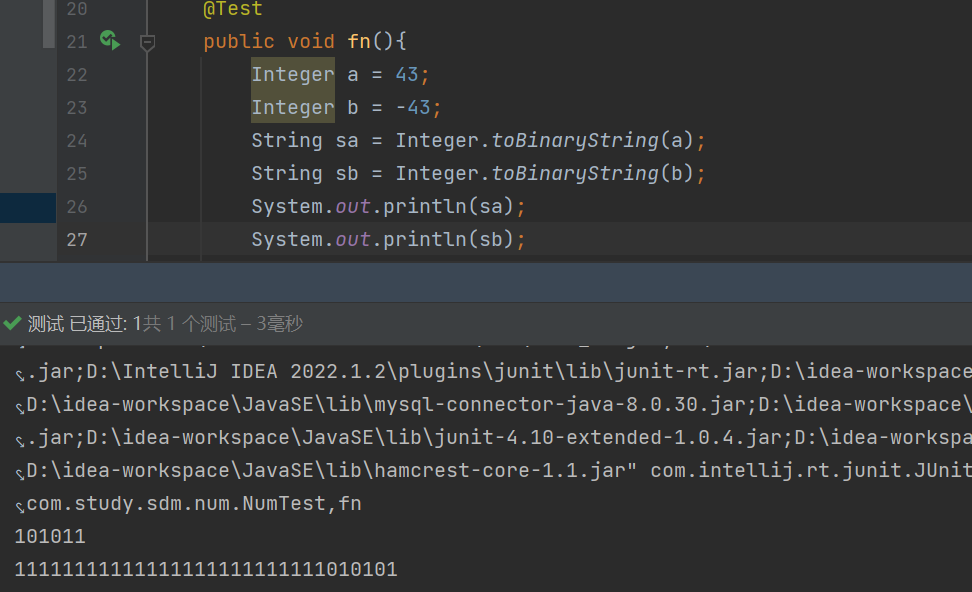
数据分类：

* 正数
* 负数

数据的计算方式以补码计算，存储方式正数原反补都一样，负数以补码方式存储，

总结：

数据以补码形式存储和计算

正数和负数在计算机中的存储形式：正数源码反码补码都一致，所以正数存储形式就是原码，负数在计算机中以补码形式存储。如图，43原码的0省略。负数补码如图！

位运算分类：针对正数和负数又有不同的情况！

* 带符号位运算
  + 左移n位：a<<n
  + 右移n位：a>>n
* 不带符号位运算
  + 左移n位：a<<<n
  + 右移n位：a>>>n

### 带符号位运算>>,<<

带符号位运算，正数首位补0，负数首位补1.因为携带符号运算，所以遵守进制法计算！

正数带符号位运算：正数情况下

*/\*\*带符号位运算，针对正数：左移存在上限溢出问题！  
 \*结果：在左移的情况下，数据未溢出，得到的值为x\*2!*

*数据溢出，得到的值是x\*2-2^(数据位数，如果是int，就是32)  
 \* 在右移动的情况下，奇数得到的值是(x-1)/2,偶数是x/2  
 \*  
 \*原理分析：  
 \*奇数：右移一位，末尾1会溢出，所以效果并非完全等于除以2！  
 \* 左移一位，如果数值没有达到数据类型所能存储的上限，由于高位皆为0，正数0溢出无影响，所以左移动等效于乘以2  
 \* 如果数值接近或达到数据类型所能存储的上限，左移一位发生溢出，效果等效于x\*2mod(2^数据位数)  
 \* 因为计算机内数据值根据数据类型成环形，  
 \* 所以最终的效果等于x\*2-2^数据位数！  
 \*偶数：右移一位，末尾0溢出，不受影响，效果等于除以2！  
 \* 左移一位，如果数值没有达到数据类型所能存储的上限，由于高位为0，所以左移动等效于乘以2  
 \* 如果数值接近或者达到数据类型所能存储的上限，左移发生溢出，原理同上。  
 \* 最终结果为 x\*2-(max)  
 \* \*/*

负数带符号位运算，情况如下

*/\*\*对于负数  
 \*  
 \* 结果：  
 \*  
 \*奇数：如果数据没有达到运算溢出后的上下限，负数根据补码运算，左移就是x\*2,右移就是(x-1)/2!  
 \* 如果数据左移发生数据溢出，结果为x\*2+2^(数据位数)  
 \* 以整形为例，结果为x\*2+2^32  
 \*  
 \*偶数：如果数据没有达到运算溢出后的上下限，负数根据补码运算，左移就是x\*2,右移就是(x-1)/2  
 \*  
 \* \*/*

总结

带符号运算：(以移动1位距离为例)

* 左移动：数据达不到上限没有溢出，左移一位就是x\*2。如果数据左移一位发生上限溢出，正数的值就是x\*2-2^(数据位数，int型就是32位)，负数的值就是x\*2+2^(数据位数)
* 右移动：类似于除法效果，不存在边界溢出，奇数就是(x-1)/2,偶数就是x/2

### 无符号位运算>>>

无符号位运算，补码右移一位，数值首位补0，所以无论正负，最终计算结果都是正数！

无符号运算只能向右位运算，不能向左！所以不存在正数越界溢出，只会负数越界溢出！

例如-1>>>1,计算机中-1的补码为11111111，无符号右移一位为01111111！，因为无符号运算的结果都是正数，所以01111111的原码也是它本身，即-1无符号右移一位变成了整形数据的最大值！

# 流程控制以及关键字

* switch case语句：不支持long,double,float三种类型

如果case语句没有break,一个case语句执行成功，后面的case语句不会判断是否符合，全部执行。

* if else if else:根据条件内的布尔值，true则执行，false执行else语句
* while循环和do while循环：根据while后的条件布尔值，为true则一直执行！

do while语句中do语句最少执行一次！

* final：

修改基本数据类型变量则变量值必须有初始值且以后不能更改。

修饰引用数据类型变量则引用数据类型的指向不能变更，但是指向的数据本身可以更改。

修饰方法，则方法不能被子类重写。

修饰类则类是最终类，不能被继承，例如String

* static
  + 静态变量: ⼜称为类变量，是成员变量的一种，静态变量存储在方法区中，所以可以通过类名.变量名访问，它在类加载的时候会进行初始化。类所有的实例都共享静态变量。静态变量在内存中只存在⼀份
  + 实例变量:每创建一个实例就会产生一个实例变量，它与该实例同生共死
  + 静态方法：在类加载的时候就会存在，静态方法必须实现，不能是抽象方法。且不依靠实例存在，通过类名.方法名就可以执行。方法中不能和this,super有关联或者有隐式关联
  + 静态代码块:在类加载的时候运行一次。

## 逻辑运算和短路运算

该运算只能针对于布尔值进行操作，常用于循环条件判断！

逻辑与&：当两个操作数结果全部计算出来，才会得出最终结果

短路与&&：只要前面能得出计算结果，后面的计算全部终止

逻辑或|

短路或||同理

boolean a = true;  
boolean b = true;  
boolean c = (a=(1==2))& (b=(1==2));//逻辑与运算，前后结果都会计算  
System.*out*.println(a);//a为false  
System.*out*.println(b);//b为false  
  
boolean d = true;  
boolean e = true;  
boolean f = (d=(1==2))&& (e=(1==2));//短路与运算，记过可以得出后面运算不进行  
System.*out*.println(d);//a为false  
System.*out*.println(e);//b为true

## 访问权限

## public

在一个java类文件中，可以有多个类，但是只能有一个类用public修饰，此类相当于该文件入口。编译时把.java文件编异成.class文件，根据这个类文件public修饰的类名编异成对应的.class文件，每个class文件都是唯一的。作为内部类，public可以修饰多个类。

public公开，都可以使用

java中访问权限修饰符有public,private,protect。

修饰符用于类或者类的成员(属性和方法)

类可见表示其他类可以使用这个类创建实例对象

成员可见表示其他类可以使用这个类的实例对象访问到该成员

## protected

用于修饰成员(属性和方法)，修饰类没有意义表示在继承体系中成员对于子类可见，父子友好.

缺省包内友好,只能同一个包内的访问(可能需要声明)。外包无法访问，声明无效。

## private

本类里可以使用 ，限制最高

接口的属性默认都是static和final，接口的方法默认都是public,如果定义为private,需要添加方法体。使用static修饰类中的代码块，当类加载时static修饰的代码块会运行一次！

重载：方法名重复，方法的参数个数，顺序，类型必须有一个不同。

# java字符串

### 字符串String，StringBuffer,StringBuilder

字符串的声明用双引号“”，字符串之间的加减操作会转化成字符串之间的拼接。

#### 字符串转义字符

使用/来表示字符串中的敏感字符：

String a = “abc”def”;错误

String a = “abc/”def” 这样就可以表示字符串中的双引号字符了。

#### String 类源码

public final class String

StringBuffer 和 StringBuilder 可变

String 因为不可变，所以线程安全

StringBuilder不是线程安全

StringBuffer线程安全

String是常量对象，当改变一个String类型的值，实际上是重新创建了一个对象，使这个String对象指向这个新的值。(效率低下，因为会频繁创建对象，更改指向)， String也经常用作为参数，因为可以保证参数不变。

StringBuffer线程安全(适用于多线程)，StringBuilder线程不安全(适用于单线程)

单线程操作字符串大量数据用StringBuilder

多线程操作字符串大量数据StringBuffer

运行效率:StringBuilder>StringBuffer>String

StringBuffer有线程安全锁。

### String类的常用方法

构造方法

//参数为字节数组，偏移量，长度，编码方式

public String(byte[] bytes, int offset, int length, Charset charset)

//获取字符串的静态方法

static String valueOf(基本数据类型)

字符串常用方法：

* int length()：获取字符串长度
* char charAt(int index)：获取索引位置的字符
* int indexOf(int ch):返回ch在字符串中第一次出现的位置
* int indexOf(String str)：返回字符串str在字符串中第一次出现的位置
* int lastindexOf(String str):返回字符串str在字符串最后一次出现的位置
* String concat(String str):返回一个new的拼接字符串，
* byte[] getbytes():返回一个字节数组
* valueof(int/float/…)：转化成字符串

转化成字符串：toString()犯法，拼串写法，构造方法

字符串的截取，字符串的拼接等！

详情查询JDK官方文档

# Java 数组

## 初始化

数组就是在内存上开辟的一串连续的空间，数组的初始化需要使用new开辟内存空间或者字面量方式直接初始化，因为数组是引用数据类型，所以数组初始化后，如果没有给其赋值，则会根据数组的数据类型产生一个默认值。

动态初始化:初始化时length不能为负，否则会编译出错

Int []a = new int [length];

静态初始化:

int [] a = new int []{x,x,,x,x,x,x};

字面量初始化，不可拆分步骤：

Int [] b = {x,x,x,x,x,x}

基本数据类型:byte,boolean,short,char,int,float,double,long;

引用数据类型对应的默认值:0, false ,0, [], 0, 0.0, 0.0, 0

引用数据类型(类):

默认值: null

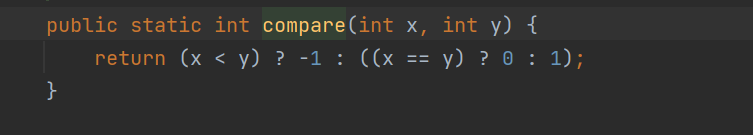
java数组的初始化需要给它分配内存。

java多维数组是数组的数组，例如:一个二维数组a[3][2],他的长度为3，而它的a[3][0]的长度为2。对数组进行操作的类如下:

## Java比较器

java中两个对象比较的方法通常是用在元素排序中，常用的两个接口Comparable和Comparator,前者是自己和自己比较，后者是第三方比较器，可以看做是平台比较器。前者的比较方法是CompareTo,后者是compare方法。

Integer的比较源码，compareTo方法调用compare方法，约定x<y返回-1，x=y返回0，x>y返回1



### equals和hashcode

两者用来标识对象，两个方法协同工作用于判断两个对象是否相等。

通过调用Object.hashcode()生成哈希值用于存储数据，但是不可避免存在hash值重复的情况，因此如果hash值相同，就再次调用equals方法判断二者是否相同。如果hash值不同，则直接判定Object不同。

Object类中对hashcode和equals要求如下：

* 如果两个对象的equals结果相同，则hashCode的返回结果必须相同
* 任何是否覆写equals，都必须同时覆写hashCode

## 动态数组

ArrayList和Vector

前者默认初始容量10，可动态扩容（一次扩容1.5倍），非线程安全

后者线程安全，但是效率太低，基本

## 数组和集合

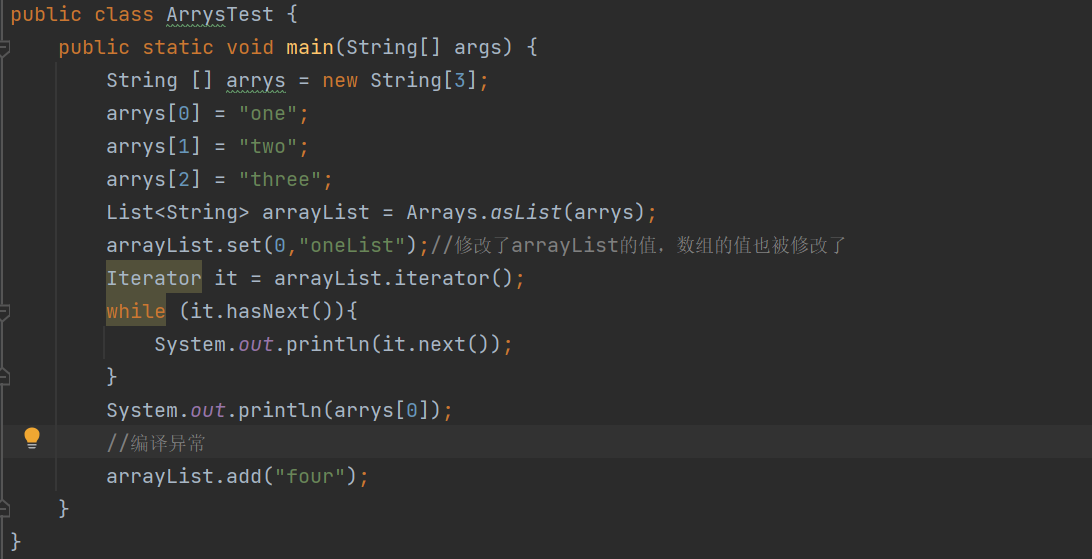
数组和集合都是用来存储对象的容器，前者性质单一，后者类型安全，功能强大，且两者之间存在必然的转化关系。

工具类Arrays类，包含大量静态方法用于操作数组

Arrays是可以对数组进行操作的工具类，包含大量静态方法用来操作数组，集合与数组的转化方法

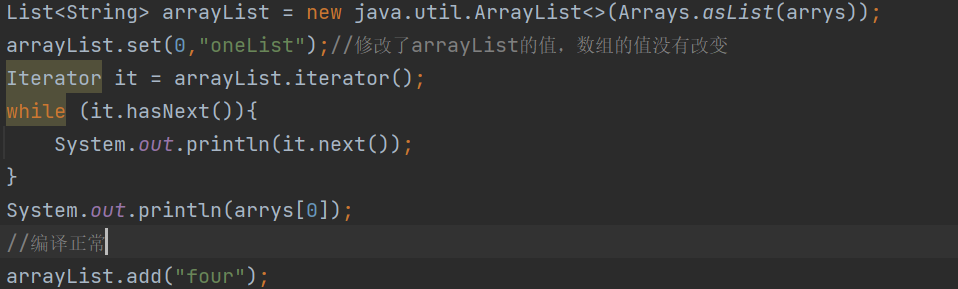
数组转集合：Arrays.aslist(array)

* list aslist(arrays) //把数组转化成集合
* x[] copyof(x [] array,length,class) //返回一个拷贝array数组的同类型数组

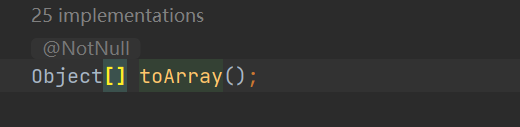


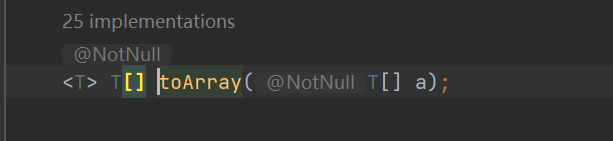
如图，使用Arrays的asList方法方法返回的集合并不是真正的集合，原因是Arrays类中有一个内部类为ArrayList,此ArrayList和集合框架中的ArrayList同名！根据作用域就近原则asList返回的就是这个内部ArrayList!

正确的写法：



集合转数组：list接口提供了toArray方法将集合转化成数组

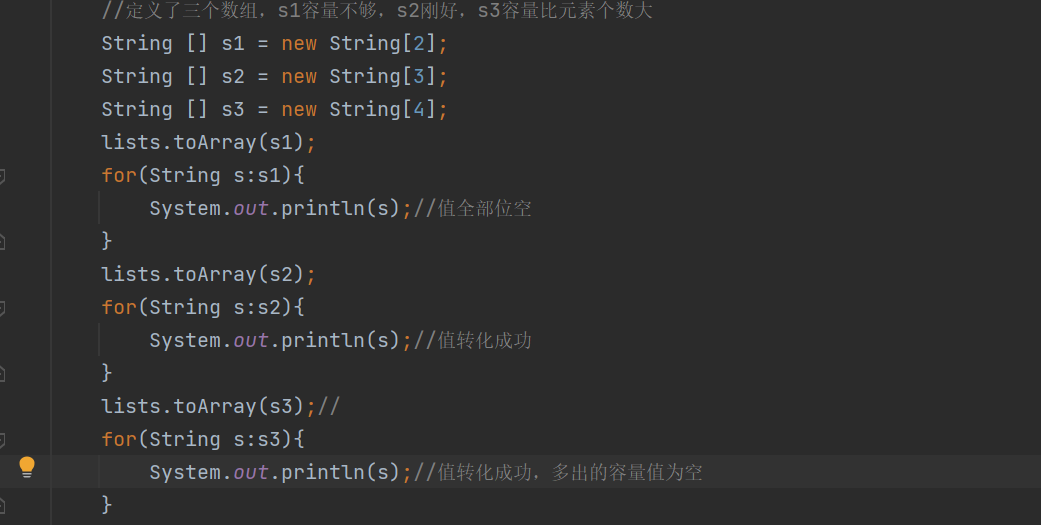




通过被子类重写，提供集合转化成数组的方法

无参转化方法会丢失泛型，不推荐使用。

有参转化方法,存在数组容量和集合元素个数的匹配问题



源码分析：

public <T> T[] toArray(T[] a) {  
 if (a.length < size)//如果数组容量不够，直接返回  
 // Make a new array of a's runtime type, but my contents:  
 return (T[]) Arrays.*copyOf*(elementData, size, a.getClass());  
 System.*arraycopy*(elementData, 0, a, 0, size);//数组容量刚好赋值  
 if (a.length > size)//数组容量过大，多出的容量的值为空  
 a[size] = null;  
 return a;  
}

当数组容量等于集合元素个数时，效率最高！所以，集合转化数组时需要注意构造的数组容量为list.size();

# 异常

## Exeception

Throwable表示任何可以作为异常抛出的类，分为error和Exception，error表示JVM无法处理的错误。Exception分为两种

受检异常:try catch用于捕获指定代码段的异常并处理恢复

非受检异常:程序运行错误。例如除0会导致程序崩溃且无法恢复

try{ 代码 1} catch(Exception e){代码2} finally{代码3}

根据代码1是否异常决定是否执行代码2，一定会执行代码3，如果三个代码段都有返回值，则开辟一块内存空间用于存储返回的值，由上而下依次覆盖。有finally则返回的值一定是代码段3的值。

# 泛型

泛型的使用，通过给定一个泛型的写法，表示只能用来传输该类型的数据！

特别写法：List<?>称之为通配符集合，在没有赋值之前可以接受任何数据，赋值以后就只能接受同类型数据。一般作为参数接受来自外部的集合或者返回一个不知道具体元素类型的集合。

List<T>最大的问题就是只能放置一种类型，如果可以随意转化类型的话，就是破窗理论，失去了类型安全的 意义。为了能够在泛型中放置多种受泛型约束的类型，JDK开发者实现了<? extends T>与<? super T>两种语法。二者的使用场景：

* <? extends T>是Get First,适用于消费集合元素为主的场景，它可以赋值给任何T以及T的子类，上界为T
* <? super T>是Put First,适用于生产集合元素为主的场景，它可以赋值给任何T以及T的父类，下界为T

# Java IO流\*\*\*

什么是IO流：

java程序操作数据的时候，就会开启一个通往数据源的流，这时候就可以通过这个抽象的流来操作数据。

IO流特点：

1. 先入先出：最先写入的数据在读取时会被最先读取到。
2. 顺序存取：可以一个一个往流中写入一串字节，也可以按写入顺序顺序读取
3. 只读或只写：每个流严格分为输入流或者输出流，一种流只能是一种功能，或读或写。如果需要读写文件，则需要两个流。

IO流分类

1. 输入输出流：根据数据流动方向分为输入输出流，以内存为核心，读入内存为输入流，从内存写到硬盘，是输出流。
2. 字符流和字节流：字节流每次操作8位二进制数据，字符流每次操作16位数据。字节流可以操作一切计算机数据，字符流只能操作文本数据。字节流本身没有缓冲区，缓冲字节流相对于字节流效率提升非常高。而字符流本身带有缓冲区，所以缓冲字符流相对于字符流提升没有那么高。
3. 节点流和处理流：节点流直接操作数据的读写，处理流对一个已经存在的流的链接和封装，加强流的功能。例如缓冲字节流。

在操作一个数据流时，如果路径不存在则需要创建路径

## File类

File类仅仅对文件系统中的文件路径，文件的创建以及判断等等进行操作，对于文件的读写功能 还需要创建输入输出流完成。File类仅作为辅助帮助输入输出流操作，本身不具备流的功能。

### 构造方法

File(String path)、File(String parent,String Child)、File(File parent,String chile)

后两个构造方法就是两个参数拼接以后形成一个新的路径，封装成File对象。

注:创建好File对象后，只是封装了一个路径，和磁盘是否有这个路径无关。

最终创建的是一个文件夹还是文件不取决于文件，而是取决于调用的方法。

createNewFile() :创建一个新的文件,文件所在路径如果没有则会报错

mkdir() :创建文件夹，如果父路径不存在，则创建失败

mkdirs() :创建文件夹，如果父路径不存在，就创建父路径再创建子路径。

delete() :删除文件或者文件夹，如果此路径名表示目录，则目录必须为空才能删除。

renameTo(File dest) : 重命名，方法调用者是一个File对象，参数也是一个File对象。调用者是修改之前的路径对象，参数是修改后的路径对象。如果改了父路径，就是剪切，如果不改父路径，就是重命名。

### 判断功能

exists():判断是否存在

isFile():判断是否是文件

isDirectory():判断是否是文件夹

### 获取功能

获取路径:

getName():获取最短的那个文件或者文件夹名称

getPath():获取相对路径，构造方法中传入的那个字符串

getAbsolutePath():获取绝对路径

获取文件文件夹属性:

length():获取文件的字节个数，只能针对文件使用

lastModified():获取文件最后修改时间

获取文件夹中的内容:

String [] list():返回文件夹中的所有内容的名称(不包含子文件夹中的内容)

File [] listFiles():返回文件夹中的所有内容的File对象(不包含子文件夹中的内容)

注:这两个方法仅适用于文件夹。

## IO流

输入输出流不能读取文件夹，路径是要包含文件的。如果某个路径的文件夹不存在就根据路径创建该文件

io流的分类:

按照流向分类:以程序所在的内存为核心，从磁盘读入内存为输入流，从内存写到磁盘，为输出流。

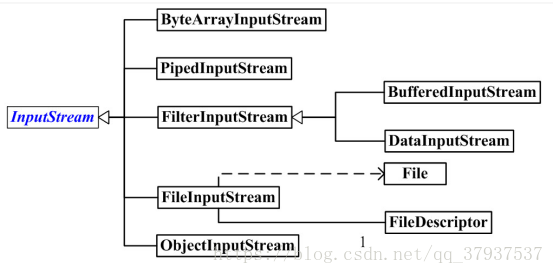
按照操作单元分类: 字符流(一次操作16位数据), 字节流(一次操作8位数据)



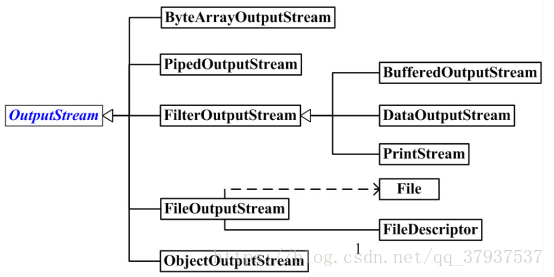
### 字节流:

对字节进行操作，常使用一个字节数组作为过渡！字节流将数据解释为原始的二进制数据，计算机中一切数据都是字节数据。所以不需要任何的编码和解码，比文本效率高。

InputStream(抽象类)



OutputStream(抽象类)



#### InputStream

字节输入流顶层抽象父类，常用方法:

int read():返回读取到的字符的ASCII码，如果返回-1,说明读取结束。

int read(byte[] arr):将读取到的数据存在arr数组中，返回读取到的数据个数

int read(byte[],int off,int len): off是起始偏移量，即从数组第几位开始读取。

int available():返回该流中还有多少字节数据的估计值

close():关闭流对象

#### **FileInputStream**

InputStream的子类，用户和磁盘交互的字节输入流。

构造方法: FileInputStream(File f) FileInputStream(String pathname)

解释:将一个File对象或者一个String描述的路径封装成一个输入流对象。

#### **OutputStream**

字节输出流的顶层抽象父类，常用方法：

write(int b)

将一个字节存储到目标文件中

write(byte[] arr)

将参数指定的字节数组中的全部字节，写出到目标文件中

write(byte[] arr, int offset, int len)

将参数指定的字节数组的一部分，写出到目标文件中

从offset索引开始，一共len个字节

close()

关流

#### **FileOuputStream**

OuputStream的子类，用户和磁盘交互的字节输出流。

构造方法: FileOuputStream(File f) FileOuputStream(String pathname)

解释:将一个File对象或者一个String描述的路径封装成一个输出流对象。如果路径是某一个磁盘，则如果File对象不存在则主动创建一个。

注:输入输出流不能读取文件夹，路径是要包含文件的。

#### 字节缓冲流

在读写数据时，让数据在缓冲区能够减少系统实际对原始数据来源的存取次数，因为一次能做多个数据单位的操作。相较而言，对于直接从文件读取数据或将数据写入文件，比起缓冲区的读写要慢多了。所以使用缓冲区的流，一般都会比没有缓冲区的流效率更高。拥有缓冲区的流被称为缓冲流，包括BufferedInputStream、 BufferedOutputStream类和BufferedReader、BufferedWriter类。缓冲流把数据从原始流成块读入或把数据积累到一个大数据块后再成批写出，通过减少资源的读写次数来加快程序的执行。

高效缓冲流(字节流): BufferedInputStream 和 BufferedOutputStream

是包装类，本身不能作为单独的字节流读写字节，用于加强基础的字节流对象，加强后，读写效率提升。

构造方法:

BufferedInputStream(InputStream is)

将一个普通的字节输入流，加强为一个高效字节输入流

BufferedOutputStream(OutputStream os)

将一个普通的字节输出流，加强为一个高效字节输出流

流的刷新与关闭:

flush(): 针对有缓冲区的.

java中的IO流中的输出流一般都有flush这个操作，这个操作的作用是强制将缓存中的输出流（字节流，字符流等）强制输出。

为什么会有这么个方法：

因为输出流在进行输出时，比如像某个文件中写入内容，其实是先将输出流写入到缓冲区，当缓冲区写满后才将缓冲区的内容输出到文件中。但是当主机完成输出流的输出后，有可能缓冲区这个时候还没有被填满，这样的话，就会一直等待主机发送内容，这时候，就可以使用flush将缓冲区的内容强制输出到文件中，清空缓冲区。

所以，一般在关闭输出流之前，要先调用flush方法强制缓冲区中的内容输出，并清空缓冲区。

close():关闭流，释放资源

1. 在带缓冲的输出流close方法中，包含一个flush。
2. close方法执行以后，流对象无法继续使用，flush方法执行后，流对象可以继续使用。
3. 不要再循环中频繁刷新，否则就失去了缓冲区的意义。

### 字符流

概述:用于直接操作字符的流对象，常使用字符数组作为过渡

FileReader 和FileWriter构造时，如果命名文件不存在，是一个目录，而不是常规文件，或者由于某些其他原因无法打开读取

字符输入流:reader(抽象类) 字符输出流:writer(抽象类)

#### Reader

顶层抽象父类，常用方法:

int read()

读取一个完整的字符，如果返回-1表示到达文件末尾

int read(char[] chs)

读取一系列字符到一个字符数组中

close()

关闭流对象

#### FileReader

reader子类，常用方法:

FileReader(String path)

FileReader(File f)

将路径封装成字符文件输入流对象

#### Writer

顶层父类，常用方法:

writer(int c)

将一个字符写出到目标文件(c整形数值对应的ASCII码字符)

write(char[] chs)

将一个字符数组写出到目标文件

write(char[] chs, int offset, int len)

将一个字符数组中的从offset开始一共len个字符写出到目标文件

write(String str)

写出一个字符串

write(String str, int offset, int len)

将字符串的一部分写出

flush()

刷新缓冲区

close()

关闭流对象

#### FileWriter

构造方法:

FileWriter(String path)

FileWrtier(File f)

将一个路径封装成字符输出流 ,如果文件存在，则先清空

FileWriter(String path, boolean append)

FileWriter(File f, boolean append)

将一个路径封装成字符输出流,如果第二个参数为true，则在文件存在的情况下，可以追加

注:字符流不能对非纯文本文件操作。因为读取到字节信息时，要对字节信息进行解码，因为是非纯文本文件，所以找不到字节信息对应的字符，只能用？替代，这就对文本信息进行了篡改。

#### 文本的拷贝

Reader中的

int read(char[] arr)

Writer中的

write(char[] arr, int offset, int len)

#### 缓冲字符流

BufferedReader 和 BufferedWriter

包装类，将普通字符流包装加强形成高效的缓冲字符流。

BufferedReader(Reader r)

将一个普通的Reader加强为BufferedReader

BufferedWriter(Writer w)

将一个普通的Writer加强为BufferedWriter

缓冲字符流特有的方法:

BufferedReader: readLine() 可以从源文件中一次读取一行

BufferedWriter: newline() 跨平台换行符

### 转换流

#### 编码表

1、GBK编码表

只定义了英文和中文的各种字符和数字的对应关系，最多使用2个字节就可以表示一个字符。英文使用1个字节表示一个字符，中文使用2个字节表示1个字符

2、UTF-8编码表

定义了全世界所有语言的符号和数字的对应关系，最多使用3个字节就可以表示一个字符。英文使用1个字节表示一个字符，中文使用3个字节表示1个字符

使用字节流和字符流拷贝不同编码的文件的问题：

1、目标文件将来的解码方式是UTF-8，现在却使用GBK的编码结果，写出到了目标文件，写出时的编码形式（GBK），和将来读取的解码格式（UTF-8）不同

2、希望写出时使用的编码表和将来读取后解码使用的编码表相同，只能使用能指定编码形式的“转换流”

#### InputStreamReader

是字节流到字符流的转换，通过流的构造方法可以完成字符流和字节流的转化

BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));

InputStreamReader(InputStream is, String charSetName)

is：用于读取磁盘字节信息的字节流

charSetName：用于解码读取到的字节信息使用的编码表名字

#### OutputStreamWriter

是字符流到字节流的转换，通过流的构造方法可以完成字符流和字节流的转化

OutputStreamWriter(OutputStream os, String charSetName)

os：用于将编码之后的字节信息写出到目标文件去

charSetName：用于解码当前流对象要写出的字符，使用的编码表

### 内存输出流

ByteArrayOutputStream

OutputStream 的子类，可以使用字节输出流的全部方法。

用于缓存不完整字节信息的一个缓冲区

方法:

toString()：将底层缓冲区中的字节解码返回字符串

toByteArray()：返回底层缓冲区数组中，有效的那部分字节

### 对象操作流

概述:读写对象的流，属于处理流

对象输出流:ObjectOutputStream

对象输入流:ObjectInputStream

对象输出流:

OutputStream 的子类，也是字节输出流

直接将内存中的对象写到目标文件的输出流

构造方法:ObjectOutputStream(OutputStream os):将一个没有对象输出能力的os加强成一个对象输出流

常用方法:writeObject(Object obj)

注:

1、在将对象输出到目标文件时，写出的对象所属的类型必须实现java.io.Serializable接口

2、写出对象时既没有编码也没有解码（在内存中，对象就是二进制的字节信息，写出到目标文件之后，仍然是二进制的字节信息），而使用文本编辑器打开文件时，强行解码，所以出现了乱码

3、将来能使用对象输入流将文件中的对象重新读取到内存中即可，不需要看懂.

对象输入流:

InputStream 的子类，

将文件中的一个完整对象读入内存

构造方法ObjectInputStream(InputStream is):把一个普通输入流is加强成一个对象输入流。

常用方法:

Object readObject();

注意事项:

序列化:把对象从内存到其他设备的过程

反序列化:把对象从其他设备到内存的过程

无论是序列化还是反序列化，操作对象都要实现Serializable接口。

随机读写流

RandomAccessFile

打印流

# JAVA异常处理机制

try catch finally return 语句的执行机制:

try{

return x;

}catch(Exception e){

return y;

}finally{

return z;

}

总结:

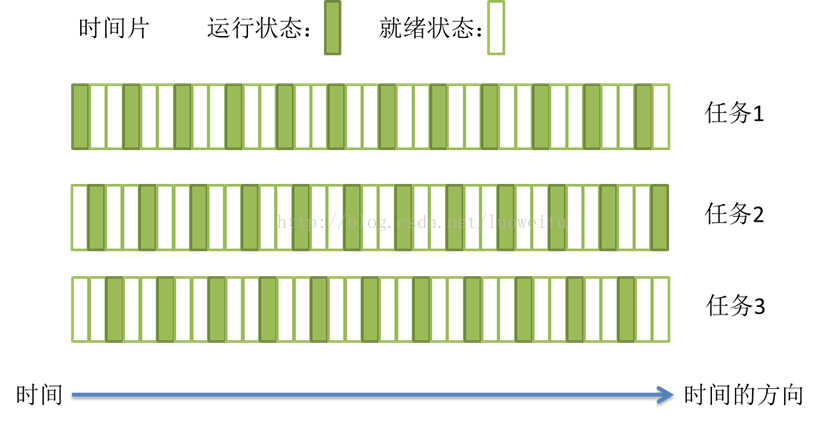
1.函数执行到try里面的内容，一定会执行finally里面的语句，除非try里面有类 似System.exit()这样的方法。catch语句是否执 行根据try语句能否捕获到异常来 决定。

2.函数执行到return不会直接返回，但是会开辟一块空间存储当前return返回值 该空间内的存储值会被第二个return返回值覆盖(return从上到下顺序覆盖)。

# JAVA并发\*\*\*

## 基础知识补充

* 任务调度：大部分操作系统，例如windows，Linux的任务调度是采用时间片轮转的方式进行任务的调度，任务获得时间片后，从就绪状态进入运行状态，当时间片用完，任务就从运行状态重新成为就绪状态，其他获得时间片的任务运行。多个任务抢占时间片进行状态的切换。
* 总结：在一段时间内，有多个任务交替执行，这就是并发机制。(线程中常使用sleep方法配合时间片轮转机制完成线程的并发执行)



并行机制：一段时间内，多个任务同时执行，多核CPU可以实现并行

* 进程：计算机的核心是CPU，而操作系统负责任务的调度，资源的分配管理。应用程序是具有某种功能的程序，运行在操作系统之上。

进程是一个具有一定独立功能的程序在一个数据集合上一次动态执行的过程(运行中的程序)。是操作系统进行资源分配和调度的一个独立单位，是程序运行的载体！进程一般分为三个部分：程序，数据集合，进程控制块！程序讲述进程需要完成的功能，数据集合是程序执行所需要的数据和工作区，程序控制块包括进程的描述信息和控制信息，是进程的唯一标志。

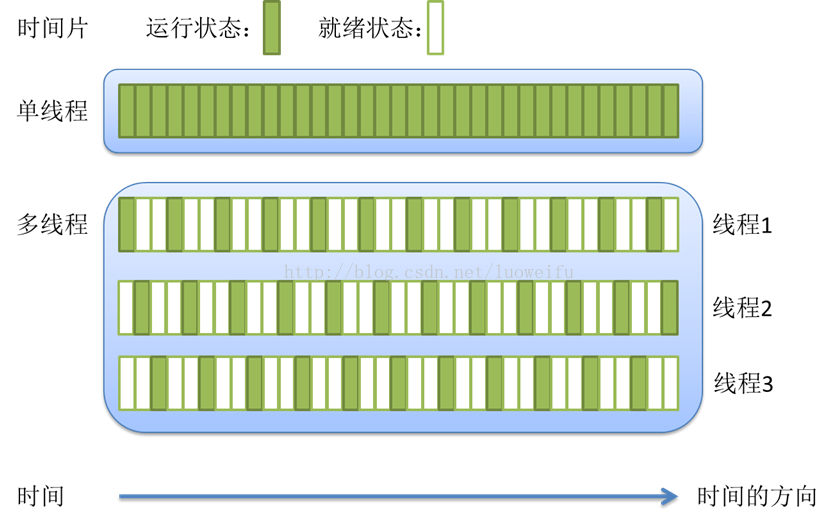
* 线程：在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。

后来，随着计算机的发展，对CPU的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了。于是就发明了线程，线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。一个标准的线程由线程ID、当前指令指针(PC)、寄存器和堆栈组成。而进程由内存空间(代码、数据、进程空间、打开的文件)和一个或多个线程组成。

* 进程和线程的区别：

1. 线程是程序执行的最小单位，进程是系统资源分配的最小单位
2. 一个进程由一个或者多个线程组成，线程是一个进程代码中不同执行路线
3. 进程与进程之间相互独立，而一个进程下的多个线程共享内存空间
4. 线程之间的切换比进程切换快

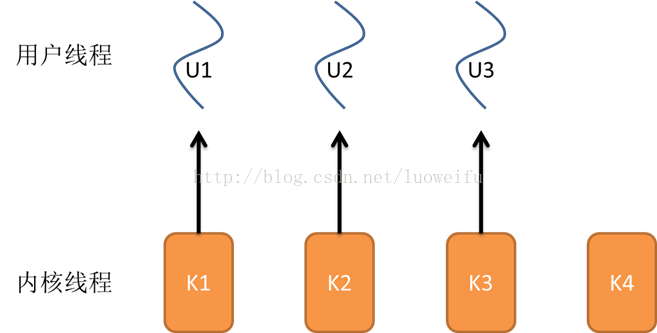
* 进程和线程之间的资源共享关系



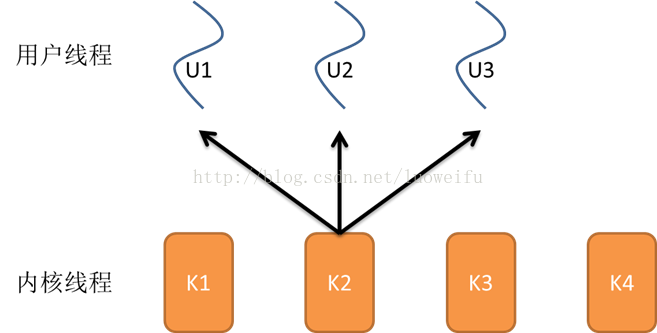
* 多线程和多核：cpu的四核八线程，8核16线程是利用超线程技术把一个物理核心分为多个逻辑处理核心，用于提高CPU运行效率。

程序一般不会去使用内核线程，而是去使用内核线程的高级接口——轻量级进程即用户线程，用户线程和内核线程有三种对应关系，一对一，多对一，多对多！

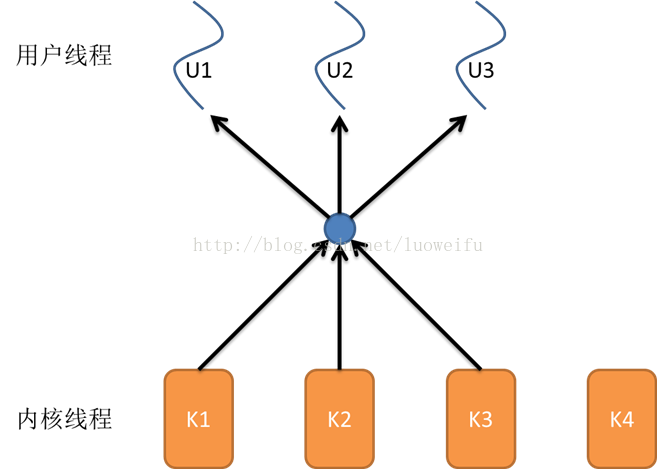
* 一对一，资源消耗过大，效率低！因为内核线程数量有限，从而限制了用户线程！



* 多对一，虽然可以有效发挥性能，但是也有很明显的缺点，如果一个用户线程阻塞，其他线程几乎都无法执行。



* 多对多模型合理避免了上两种的缺点，成为主流操作系统的应用首选



* 线程的生命周期：当线程的数量小于处理器的数量时，线程的并发是真正的并发，不同的线程运行在不同的处理器上。但当线程的数量大于处理器的数量时，线程的并发会受到一些阻碍，此时并不是真正的并发，因为此时至少有一个处理器会运行多个线程。

在单个处理器运行多个线程时，并发是一种模拟出来的状态。操作系统采用时间片轮转的方式轮流执行每一个线程。

现在，几乎所有的现代操作系统采用的都是时间片轮转的抢占式调度方式，如我们熟悉的Unix、Linux、Windows及Mac OS X等流行的操作系统。我们知道线程是程序执行的最小单位，也是任务执行的最小单位。在早期只有进程的操作系统中，进程有五种状态，创建、就绪、运行、阻塞(等待)、退出。早期的进程相当于现在的只有单个线程的进程，那么现在的多线程也有五种状态，现在的多线程的生命周期与早期进程的生命周期类似。

进程的生命周期:



## 线程的生命周期

如图：有五大状态

* 创建状态
* 就绪状态
* 运行状态
* 阻塞状态
* 退出



总结：系统采用时间片轮转方式进行将资源分配给进程，进程采用时间片轮转方式将资源分配给线程。进程和线程的五个状态类似！

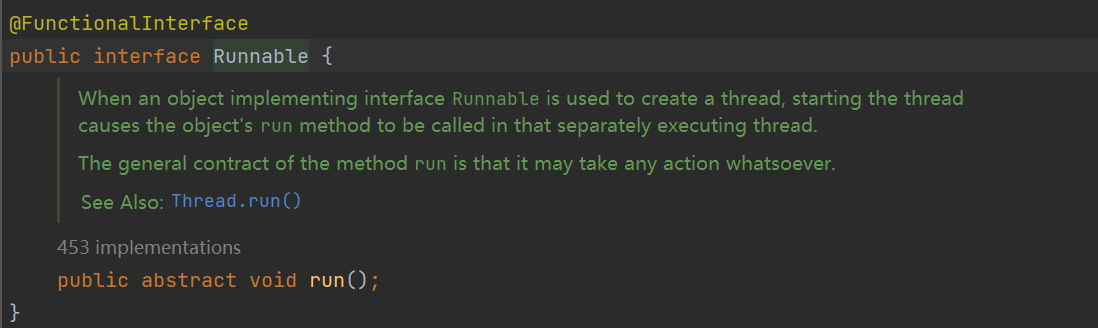
### 线程的创建new

三种创建方式：

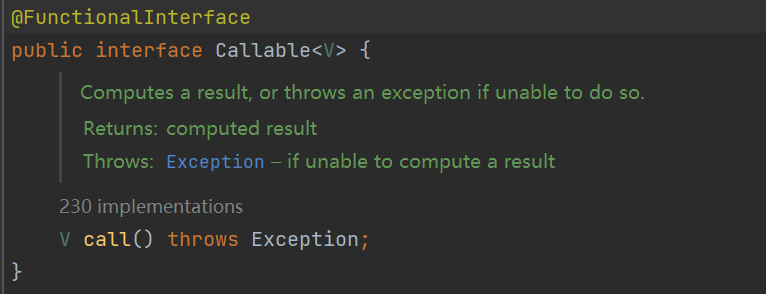
1. 继承Thread类
2. 实现Runnable接口（函数式接口，只有一个run方法），该类线程的启动通过Thread的有参构造方法启动（参数是runnable接口实现类实例）
3. 实现Callable接口（函数式接口，只有一个call方法）

推荐使用第二种方式，因为继承Thread类往往不符合里氏代换法则，实现Runnable接口可以使编程更加灵活，对外暴露的细节更少。

runnable接口源码



callable接口源码：可以看出实现callable接口的优点就是可以获取一个返回值且能够抛出异常。



### 线程就绪RUNNABLE

就绪状态，调用start之后运行之前的状态，线程的start方法不能多次调用，否则会抛出ILLegalStateException

### 运行状态RUNNING

运行状态，是run方法正在执行时线程的状态，线程可能会因为某些原因退出RUNNING状态，如时间，异常，锁，调度等。

### 阻塞状态BLOCKED

阻塞状态分为三种情况：

1. 主动阻塞：调用了Thread类的某些方法，让出cpu主动权。如sleep,join方法
2. 同步阻塞：锁被其他线程占有
3. 等待阻塞：执行了wait方法

### 死亡状态DEAD

终止状态，run方法执行完毕或者因为异常退出后的状态，此状态不可逆转。

## 基础线程机制

### 线程基础补充

main作为主线程，不会阻塞！会和子线程交替执行！如果CPU有多核，可能会并行和并发执行都有！

如果主线程运行结束，子线程还在运行，整个进程还不会结束，会继续运行。主线程的结束不会影响子线程的结束，当所有子线程全部结束整个进程才会结束。

线程的退出只取决于线程的存活时间！

如果子线程有返回值给主线程，则前提条件是主线程的存活时间一定高于有返回值的子线程

### 线程的终止

1. 线程完成任务后，会自动退出
2. 还可以通过使用变量的方式控制run方法退出的方式停止线程，即通知方式

### 线程的常用方法

1. setName //设置线程名称，使之与参数name相同
2. getName //返回线程名称
3. start //启动线程
4. run //线程启动后自动调用run方法
5. setPriority(int newPriority) //更改线程的优先级
6. getPriority //获取线程优先级
7. sleep //在指定毫秒数内让正在执行的线程休眠，根据操作系统内核的时间片轮转执行任务机制，使线程休眠就是为配合时间片轮转机制让其他线程得到时间片资源，以此完成线程的并发执行
8. interrupt //中断线程

常用方法注意事项:

1. run方法不会启动线程，这只是一个单纯的方法！start底层会创建一个新的线程并且调用run方法！
2. 线程优先级的范围
3. void interrupt():中断线程，但是没有真正结束线程，一般用于中断正在休眠线程
4. static void sleep(long millis):线程的静态方法，使当前线程休眠，参数是毫秒数。sleep方法不要插在判断操作代码之间，会破坏数据的的一致性
5. static void yield():线程的礼让(静态方法)，让出cpu，让其他线程先执行，但是礼让时间不确定(由cpu决定)，所以不一定礼让成功。用法就是线程调用自身yield方法，表示该线程礼让。(正在执行的线程暂停，不阻塞，将线程从运行状态转化为就绪状态)
6. void join():线程的插队，插队线程一旦插队成功，则肯定先将插队线程的所有任务全部完成再去执行其他线程！用法调用其他线程的join方法，表示让其他线程插队！
7. boolean isAlive():返回线程的存活状态

线程的停止不推荐使用stop或者destroy等方法，具有风险，一般通过设置通知变量的方式使其停止。

线程的基本属性优先级：

Thread.MIN\_PRIORITY = 1;

Thread.MAX\_PRIORITY = 10;

Thread.NORM\_PRIORITY = 5;

对线程优先级的操控方法，getPriority()和setPriority(int xx);

线程的调度取决于cpu,设置线程的优先级越大，分配到的资源就越多，cpu调度的可能性就越高。不是说优先级大的就一定执行

### 用户线程和守护线程

1. 用户线程:也叫工作线程，当线程的任务执行完或通知方式结束
2. 守护线程:为工作线程服务，当所有的工作线程结束，守护线程也自动结束。
3. 常见的守护线程:垃圾回收机制

守护线程举例：

一个进程下有一个main主线程，还有一个无线循环的子线程，当主线程已经结束，子线程因为无限循环并不会结束。所谓守护线程就是当主线程结束后，无限循环的子线程也可以自动结束。

使用方法:子线程实例.setDaemon(true)

注意：守护线程先设置再启动

线程的休眠(sleep):sleep存在异常情况，sleep时间到达后进入就绪状态。每一个对象都有一个锁，sleep不会释放锁。

线程的等待(wait):wait方法是Object的方法，使用wait方法会释放锁且wait方法依赖synchronized关键字。wait方法需要被唤醒

## 线程安全问题

线程的安全问题只会出现在多线程上，当有多个线程操作同一个数据时，某个线程正在写数据而此时其他的线程来读数据，便会出现脏读现象，很明显这种现象是不被允许的。为了保证线程的安全问题，从以下四个维度考量：

1. 数据单线程内可见：单线程内总是安全的，通过限制数据在单线程内可见，可以避免数据被其他线程篡改，最典型的是线程局部变量（函数体内的变量），它独立存储在独立虚拟机栈的局部变量表中
2. 只读对象：只读对象总是安全的。最典型的对象有Integer,String（其他基本数据类型的封装类）。一个对象想要拒绝任何写入，必须要满足以下条件：使用final关键字修饰类，避免被继承。使用private final修饰的属性避免被中途修改，没有任何更新方法，返回值不可变。
3. 线程安全类：例如StringBuffer，内部有着非常明确的线程安全机制
4. 同步和锁机制：如果相要对某个对象进行并发更新操作，且该对象不符合以上三种情况，需要我们在代码中实现安全的同步机制。线程安全的核心理念是“要么只读，要么加锁”,Java并发包：java.util.concurrent,JUC
5. Java并发包：java.util.concurrent,JUC
   1. 线程同步类
   2. 并发集合类
   3. 线程管理类：线程池，使用Executor静态工厂或者使用ThreadPoolExecutor等
   4. 锁相关类，以下

### 线程同步

资源共享的两个原因是资源紧缺和共建需求，线程共享cpu是从资源紧缺的维度来考虑的，而多线程共享一个变量则是共建需求。在多线程操作同一个变量时，如果操作没有原子性，就可能产生脏数据。最典型的i++操作并不具备原子性！

### 线程锁的要求

使用synchronized修饰代码时，尽量只修饰读写数据部分的代码。synchronized修饰范围越大，越影响线程的并发执行，效率低（能锁对象就不要锁类，能锁代码块，就不要锁方法）!不要再同步代码块内写入线程的操作方法。

同步的局限性：使程序执行效率下降。

### 互斥锁

基本介绍：

Java语言中，引入对象互斥锁概念，用来保证共享数据操作的完整性。

每个对象都对应于一个可称之为互斥锁的标记，这个标记用来保证任意时刻，只能有一个线程访问该对象。

关键字synchronized与对象的互斥锁联系，当某个对象用synchronized修饰时，表明该对象任意时刻只能由一个线程来访问。

### 方法锁，类锁和对象锁

方法锁属于对象锁，所以总体可以分为两类，对象锁和类锁。

类锁和对象锁简介和区别:

一个类可以有多个对象，所以一个类可以有多个对象锁

当一个类中有多个synchronized修饰的同步方法时，其中一个方法被线程访问持有锁，其他方法同样被锁住，其他线程依旧访问其他同步方法，这时相当于该类的整个对象被锁住

一个类只能有一个.class，所以一个类只能有一个类锁！

方法锁(synchronized修饰方法)：

在方法声明中加入synchronized关键字声明。每个类的实例对象对应一把锁，每个synchronized方法都必须获得调用该方法的类实例的锁才能执行，执行期间独占该锁，其他线程进入阻塞状态，直至该方法执行完毕。方法锁也属于对象锁

对象锁(synchronized修饰方法和非静态代码块)：

该对象被线程访问时，因为方法被synchronized修饰，导致同一时间只能有一个线程调用该对象的同步方法，相当于锁被加在对象上，即对象锁(this)！多个线程访问的锁对象必须是同一个对象

类锁(synchronized修饰静态方法，静态方法中的代码块以及静态代码块):

无论一个类被实例化多少次，它的静态方法和静态变量内存中只有一份。当静态变量或者方法被synchronized修饰时，此类的所有实例化对象都在共用这一个方法，共用一把锁，所以这就是类锁(类.class)！

### synchronized的两种用法

synchronized方法和synchronized块

sychronized方法只需要在方法修饰符后添加synchronized关键字即可。

sychronized块需要在关键字后面添加锁的对象(对象锁或者类锁)。默认是this

切记锁的对象一定是变化的量，即需要读写的数据



synchronized方法控制对对象的访问，每个对象都有一把锁。每个synchronized方法都必须获得调用该方法的对象的锁才能执行，否则线程会阻塞。方法一旦执行就独占该锁，该方法执行完毕后释放锁，后面的线程继续争取锁，争取到的执行。

### 线程死锁

多个线程都占用了一些共享资源，并且相互等待其他线程占有的资源才能运行，从而导致两个或者多个线程都在等待对方释放资源，都停止执行的情况。某一个同步代码块同时拥有两个以上的对象锁的时候，就可能发生死锁问题

### 死锁产生的必要条件

1. 互斥条件：一个资源每次只能被一个进程使用
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放
3. 不剥夺条件：进程已获得的资源在未使用完之前不能强行剥夺
4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系

破坏死锁的方法就是破坏以上任意一个或多个条件就可以

### 释放锁

以下操作会释放锁

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，同步方法执行中遇到break,return
3. 当前线程在同步代码块，同步方法中遇到未处理的异常或者error,导致程序直接结束
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

以下操作不会释放锁

1. 当前线程执行同步代码块，同步方法时，调用sleep方法，yield方法
2. 当前线程执行同步代码块，同步方法时，其他线程调用了该线程的suspend()方法，将该方法挂起，该方法也不会释放锁

## JUC Lock锁

从JDK5.0开始，java提供更加强大的线程同步机制，通过显示定义同步锁实现对象同步。同步锁使用Lock对象充当

java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问。每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得lock对象

ReentrantLock类实现了lock,它拥有与synchronized相同的并发性和内存语义，在实现线程安全的控制中常用，可以显示加锁和释放锁。

ReentrantLock基本使用实例代码(创建实例化对象然后对数据操作代码加锁，操作完以后解锁);



## synchronized和lock的对比

* lock是显示锁，需要手动上锁放锁。synchronized是隐式锁，离开作用域自动解锁
* lock只能锁代码块，synchronized能锁代码块和方法
* 使用lock锁效率更高，拓展性更好(子类更多)
* 推荐使用顺序:lock>同步代码块>同步方法

## 线程的协作

### 线程的通信-分析

这是一个线程同步问题，经典案例生产消费者问题！生产者和消费者共享一个资源，并且生产者和消费者之间相互依赖！

对于生产者，在没有生产产品 之前需要通知消费者等待。而生产力产品之后又要通知消费者消费

对于消费者，在消费之后，要通知生产者自己已经结束了消费，需要生产者继续生产产品以供消费。

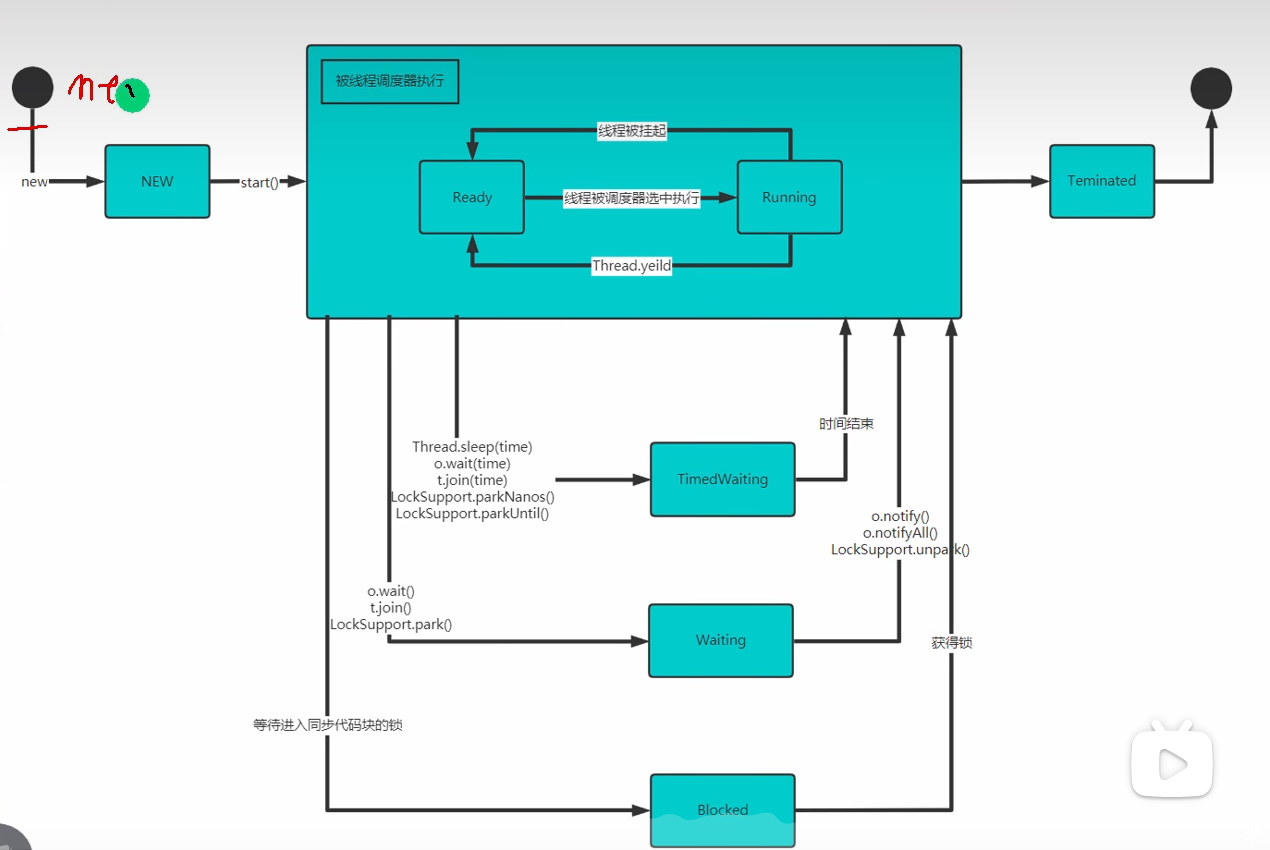
生产消费者问题中，仅仅一个synchonized是不够的。

synchronized可以完成资源的同步，但是不能用来实现不同线程之间消息的传递，即通信

java提供以下方法解决线程之间的通信

* wait():表示线程会一直等待，直到有其他线程通知，与sleep不同的是会wait会释放锁
* wait(long time):指定等待的毫秒数
* notify():唤醒一个处于等待状态的线程
* notifyAll():唤醒同一个对象上所有调用了wait方法的线程，根据优先级大小来进行优先调度。

注意：以上均是Object类的方法，都只能在同步方法或者同步代码块中使用，否则会抛出异常IIIegalMonitorStateException



### 生产消费者问题

并发协作模型：生产消费者问题(管程法，信号灯法)

生产者：产生数据的模块

消费者：处理数据的模块

缓冲区：作为二者之间的一个缓冲，类似于仓库存储

管程法：设立数据缓冲区，通过对缓冲区数据的判断通知生产者和消费者。

信号灯法：不使用缓冲区设置一个变量充当信号灯，通过更改变量值来通知消费者和生产者。

示例代码路径: D:\idea-workspace\JavaSE\src\com\study\sdm\Thread\CusAndPro

## 线程池

背景:经常创建和销毁使用量特别大的资源，比如并发情况下的线程，对性能影响特别大。

作用：提前创建好多个线程放入线程池中，使用时就从线程池中获取，使用完放回池中。

可以避免频繁创建销毁线程，实现同一个线程的重复利用。

有限：

* 降低资源消耗，提高性能。
* 便于线程管理(…)
  + corePoolSize:核心池的大小。
  + maximumPoolSize:最大线程数。
  + keepAliveTime:线程没有任务时最多保持多长时间后会终止。

### 使用线程池

JDK5.0起提供了线程池API:ExecutorService接口和Executors工厂

* ExecutorService:真正的线程池接口，常见子类ThreadPoolExecutor
  + void execute(Runnable command):执行线程任务，没有返回值，一般用来执行Runnable线程
  + <T>Future<T>submit(Callable<T>task):执行线程任务，有返回值，一般用来执行Callable
  + void shutdown():关闭连接池
* Executors:工具类，线程池的工厂类，用于创建并返回不同类型的线程池。

//创建线程池  
ExecutorService service = Executors.*newFixedThreadPool*(5);  
service.execute(new MyThread());

# JAVA反射，注解和代理

### 静态语言和动态语言

1. 动态语言
   1. 是一类在运行时可以改变其结构的语言，例如新的函数，对象，甚至代码可以被引进，已有的函数可以被删除或者是其他结构上的变化。通俗来说，就是在运行时代码可以根据某些条件改变自身结构

主要动态语言如：javascript,python,c#,php

1. 静态语言
   1. 运行时结构不可变的语言就是静态语言，如java,c,c++
   2. java不是动态语言，但java可以被称作“准动态语言”，即java具有一定的动态性，我们可以利用反射机制获得类似动态语言的特性。java的动态性让编程更加灵活。

### JAVA Reflection

Reflection反射是java被视为动态语言的关键，反射机制允许程序在执行期间借助于Reflection API获取任何类的内部信息，并能直接操作任意对象的内部属性以及方法。

#### 反射的API——Class类

java.lang.Class:代表一个类

java.lang.reflect.Method:代表类的方法

java.lang.reflect.Field:代表类的成员变量

java.lang.reflect.Constructor:代表类的构造器

实例代码如下：c1 c2 c3哈希码一样，说明是同一个对象

public class Reflection {  
 public static void main(String[] args) throws Exception {  
 //通过反射获取类的Class对象  
 Class<?> c1 = Class.*forName*("com.study.sdm.reflection.User");  
 Class<?> c2 = Class.*forName*("com.study.sdm.reflection.User");  
 Class<?> c3 = Class.*forName*("com.study.sdm.reflection.User");

//一个类在内存中只有一个Class对象

//一个类被加载后，整个类的结构都会被封装在Class对象中  
 System.*out*.println(c1.hashCode());  
 System.*out*.println(c2.hashCode());  
 System.*out*.println(c3.hashCode());  
 }  
}  
  
class User{}

加载完类之后，在堆内存的方法区就产生了一个Class类型的对象(一个类只有一个Class对象)，这个对象包含了完整的类的结构信息，我们可以通过这个对象得到类的结构。这个对象就如同一面镜子，镜子完整的显示了目标类的全部结构，我们称之为反射。

正常方式：引入jar包——>new实例化——>获取实例对象

反射方式：实例化对象——>getClass()方法——>得到完整的包类名称

理解：每一个类都对应唯一一个Class类，Class类好比是类的镜子！通过这个Class类，我们可以获取任意一个类的所有结构信息！详情查询JDK API文档Class类的详细信息！

#### java反射机制提供的功能

* + - 1. 在运行时判断任意一个对象所属的类
      2. 在运行时构造任意一个类的对象
      3. 在运行时判断任意一个类所具有的成员变量和方法
      4. 运行时获取泛型信息
      5. 运行时调用任意一个对象的成员变量和方法
      6. 运行时处理注解
      7. 生成动态代理

#### 反射的优缺点

1. 优点
   1. 可以实现动态创建对象和编译，灵活性强
2. 缺点
   1. 损耗性能，使用反射基本上是一种解释操作，我们可以告诉JVM,我们希望做什么并且让他满足我们的需求，这类操作总是慢于直接执行的其他操作

## Class类详解

对照镜子后得到的信息，某个类的属性方法和构造器，以及他的父类接口等。对于每个类而言，JRE都为其保留了一个不变的Class对象，一个Class对象包含了特定某个结构的有关信息。

* Class本身也是一个类
* Class对象只能由系统建立对象
* 一个加载的类在JVM中只会有一个Class实例
* 一个Class对象对应的是一个加载到JVM虚拟机中的一个.class文件
* 每个类的实例都会记得自己是由哪个Class实例生成
* 通过Class对象可以完整得到一个类的所有结构信息
* Class类是Reflection的根源，针对任何你想动态加载运行的类，唯有先获得相应的Class对象。

#### 常用方法



##### 获取Class类的实例

//已经类的信息，通过类的class属性获取，效率最高

Class user\_cl = User.class;  
System.*out*.println(user\_cl);

//已知某个类的实例，调用实例的getClass()方法获取Class对象

User u = new User();  
Class c = u.getClass();  
System.*out*.println(c.getName());

//已知一个类的全类名，且该类在此路径下，通过Class的静态方法forName()获取，可能会抛出异常ClassNotFoundException

Class cl\_user = Class.*forName*("com.study.sdm.reflection.Me");  
System.*out*.println(cl\_user.getName());

//Java基本数据类型的包装类都有内置的静态属性，用于返回它对应的Class实例

Class c\_i = Integer.*TYPE*;

#### 有Class对象的类型

1. class:外部类，内部类(成员内部类，静态内部类)，局部内部类，匿名内部类
2. interface:接口
3. 数组
   1. :数组的Class对象需要注意的是只要数组类型和维度一样，就是同一个Class对象，数组长度不影响Class对象
4. 枚举类
5. 注解
6. 基本数据类型封装类
7. void

#### 获取运行时类的完整结构

通过反射获取运行时类的完整结构,详情查询JDK API .java.lang.reflect包下提供了对应的类以获取Class对象封装的信息。注意：在获取方法时需要注意方法的修饰符。

1. Field类（属性）
2. Method类（方法）
3. Constructor类（构造器）

//实例代码，通过Class对象的方法和反射提供的API完成对Class对象信息的获取

public static void main(String[] args) throws Exception {  
 Class c\_user = Class.*forName*("com.study.sdm.reflection.User");  
 System.*out*.println(c\_user.getName());//包名+类名  
 System.*out*.println(c\_user.getSimpleName());//类名  
 Field [] fields = c\_user.getFields();//只能找到public属性  
 fields = c\_user.getDeclaredFields();//找到全部属性  
 Field name = c\_user.getDeclaredField("name");//定向获取类的属性  
 Method [] methods = c\_user.getDeclaredMethods();  
}

#### 通过反射动态创建对象

Class类中提供了泛型方法 newInstance();该方法有两大限制

* 创建对象对应的类必须要有无参构造器，本质上是调用了类无参构造方法
* 类的构造器访问权限需要足够
* //实例代码如下
* public static void main(String[] args) throws Exception {  
   User user  
   = (User) Class.*forName*("com.study.sdm.reflection.User").newInstance();//本质调用了无参构造器  
   System.*out*.println(user);  
  }

如果希望通过有参构造器反射创建对象，需要先获得构造器方法，然后创建实例，步骤如下：

* 通过Class类的getDecaredConstructor(参数)取得本类的指定形参类型构造器
* 向构造器形参传递一个对象数组进去，里面包含了构造器中所需的各个参数
* 通过Constructor的newInstance()方法实例化对象

示例代码如下：

Class cuser = Class.*forName*("com.study.sdm.reflection.User");  
Constructor constructor = cuser.getDeclaredConstructor(String.class,int.class,int.class);  
constructor.newInstance("sdm",99,99);  
System.*out*.println(cuser);

#### 通过反射调用方法

//通过反射实现对象的属性注入，方法invoke

User user = (User) cuser.newInstance();  
Method setname = cuser.getDeclaredMethod("setName", String.class);  
setname.invoke(user,"sdm");//invoke激活，参数是该方法的参数以及所属对象  
System.*out*.println(user.getName());

#### 反射之权限检测关闭

java对象属性 的封装利用private修饰符，表示属性只能被本类内部访问，相当于一个权限的检测！这种修饰符修饰的属性不可以直接操作！但是在反射机制中可以通过Field类的setAccessible(boolean)方法关闭权限！

//关闭检测权限

User user = (User) cuser.newInstance();  
Field f = cuser.getDeclaredField("name");  
f.setAccessible(true);//关闭属性检测权限  
f.set(user,"aaaaaoo");  
System.*out*.println(user.getName());

##### Accessible方法

此三个类都具有该方法，该方法是启用和禁用安全检查开关

1. Method
2. Field
3. Constructor

该方法参数为boolean

1. true:取消安全检测开关，提高反射效率，如果代码中常常使用反射，而该句代码需要被频繁调用，就可以设置为true
2. false:安全检测开关启用

#### 反射操作泛型

针对泛型，Method和Fileld,Constructor类中提供了特定的方法(Generic)

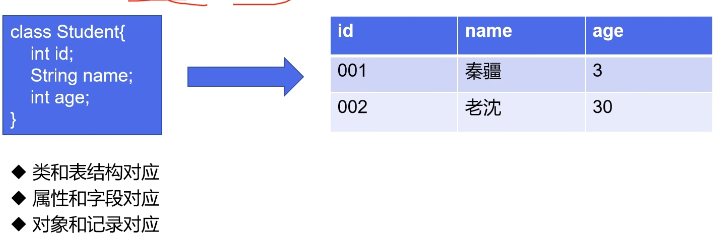
Method m1 = Test11.class.getMethod("test01", Map.class, List.class);  
Type [] genericPT = m1.getGenericParameterTypes();  
for(Type g:genericPT){  
 System.*out*.println("#"+g);  
 if(g instanceof ParameterizedType){//ParameterizedType：参数化类型  
 Type [] acctually = ((ParameterizedType) g).getActualTypeArguments();//获取真实参数类型，需要强转  
 for(Type t:acctually){  
 System.*out*.println(t);  
 }  
 }  
}

#### 反射操作注解

Java.lang.reflect提供的API(Method,Constructor,Field)都具有getAnnotation方法!

了解ORM——》Object relectionship Mapping对象关系映射

Spring框架底层包含了利用java反射机制对注解的操控完成spring框架中注解功能的实现



示例：利用注解和反射完成类和表结构的映射关系(java类的一维表通过注解映射成为数据库的二维表，这就是ORM)

//自定义两个注解，类的注解

@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@interface TableSDM{  
 String value();  
}  
  
//属性的注解  
@Target(ElementType.*FIELD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@interface FieldSDM{  
 String cname();  
 String type();  
 int length();  
}

java Student2表有三个属性，通过自定义注解给每个属性又分配了三个数据库属性，至此完成了ORM映射

@TableSDM("db\_student")  
class Student2{  
 @FieldSDM(cname = "db\_id",type = "int",length = 10)  
 private int id;  
 @FieldSDM(cname = "db\_age",type = "int",length = 10)  
 private int age;  
 @FieldSDM(cname = "db\_name",type = "varchar",length = 10)  
 private String name;  
 public Student2(){  
 }

具体获取注解的操作，api应用

Class c1 = Class.*forName*("com.study.sdm.reflection.Student2");  
Annotation[]annotations = c1.getAnnotations();//获得注解数组  
//获取注解的值  
TableSDM tableSDM = (TableSDM)c1.getAnnotation(TableSDM.class);  
String value = tableSDM.value();  
System.*out*.println(value);  
//获得类指定的注解  
Field f = c1.getDeclaredField("name");  
FieldSDM fieldSDM = f.getAnnotation(FieldSDM.class);  
System.*out*.println(fieldSDM.cname());  
System.*out*.println(fieldSDM.type());  
System.*out*.println(fieldSDM.length());

## 注解

Annotation是从JDK5.0开始引入的技术

* Annotation的作用
  + 不是程序本身，可以对程序作出解释以及约束的作用
  + 可以被其他程序读取
* Annotation的格式
  + @注释名
* Annotation在哪里使用
  + 可以在package,class,method,field上面使用，相当于给他们添加了额外的辅助信息，我们可以通过反射机制编程实现对这些元数据的访问。

### 内置注解

* @Override:定义在java.lang.Override中，此注释只适用于修辞方法，表示一个方法声明打算重写超类中的另外一个方法
* @SuppressWarnings:定义在java.lang.SuppressWarnings中，用来抑制编译时的警告信息，使用时带有内置参数
  + @SuppressWarnings(“all”)
  + @SuppressWarnings(“unchecked”)
  + @SuppressWarnings(value={“unchecked”,”deprecation”})
  + 等等…

### 元注解

元:最小单位，不可分割

作用：负责注解其他注解，java中定义了四个标准的meta-annotation类型，他们被用来提供对其他annotation类型做说明

所在jar包以及注解：java.lang.annotation

* @Target:用于描述注解的使用范围
* @Retention:表示需要在什么级别保持该注释信息，用于描述注解的生命周期，（SOURCE<CLASS<RUNTIME）
* @Document:说明该注解将被包含在javadoc中
* @Inherited:说明子类可以继承父类中的注解

### 自定义注解

使用@interface自定义注解，自动继承java.lang.annotation.Annotation接口

分析：

* @interface声明一个注解，格式： 修饰符 @interface 注解名{内容}
* 其中的每一个方法实际上是声明了一个配置参数
* 方法的名称就是参数的名称
* 返回值类型就是参数的类型（返回值只能是基本类型(class,String,enum)）
* 可以通过default声明参数的默认值
* 如果只有一个参数成员，一般参数名为value，value可以省略不写
* 注解元素必须要有值，定义注解元素时，经常使用空字符串0作为默认值

示例代码如下：

public class DataTest {  
 //使用自定义注解，注解内的参数如果有设置默认值使用时就可以不写  
 @Myannotation(ss={"道"})//ss没有设置默认值，所以需要手动设置  
 public void test(){  
 }  
}  
@Target({ElementType.*TYPE*,ElementType.*METHOD*})  
@Retention(RetentionPolicy.*RUNTIME*)  
@interface Myannotation{  
 //注解的参数:参数类型+参数名();  
 String name() default "";  
 int age() default 0;  
 int id() default -1;  
 String [] ss();  
}

## 代理

### 静态代理

代理思想在spring框架和线程等其他方面也频繁使用，某个类或者接口中某个方法A因为底层代码限制了A的存在且方法的形式不可更改而开发者又需要对方法A实现其他功能，这时就可以利用代理的思想创建一个代理，代理在完成其他功能的同时又调用了方法A，这样既满足的了A功能的需要又增强了方法的功能

总结：代理可以实现原方法所不具备的功能

条件：代理对象和真实对象需要实现同一个接口，并且代理对象中需要注入真实对象

### 动态代理

# JDBC

概述：JDBC是一种用于执行SQL语句的Java API,是连接数据库和Java应用程序的纽带

java操作数据库流程如下：

1. 导入依赖mysql-connector-java
2. 注册驱动,常使用反射机制注册驱动
3. 获取连接Connection
4. 创建数据操作Statement

常用的接口和方法：

1. Connection
   1. 数据库连接对象，代表一次会话，可以创建statement对象
      1. 方法createStatement()
2. DriverManager
   1. 驱动管理服务，可以创建数据库连接对象
      1. 方法static Connection getConnection(url,username,password)
3. Statement
   1. 在已经建立连接的基础上向数据库发送语句，一个statement对象代表一个原始语句
      1. 方法execute()执行sql语句
4. PrepareStatement
   1. 用来动态执行SQL语句，推荐使用
      1. 增删改方法executeUpdate()
      2. 查询方法executeQuery()
      3. set…设置参数值
      4. get…获取值
5. ResultSet
   1. 存放查询数据库的结果集

//数据的查询

try {  
 //注册驱动  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 String username = "root";  
 String password = "SDM4444NOG";  
 //通过驱动管理获取连接  
 connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/chapter2"  
 ,username,password);  
 String sql1 = "select \* from user where id = ?";  
 Integer id = Integer.*valueOf*(1);  
 //通过会话对象创建动态语句  
 preparedStatement = connection.prepareStatement(sql1);  
 //给动态语句设置参数值，索引从1开始  
 preparedStatement.setObject(1,id.intValue());  
 //查询语句返回结果集，结果集是二维数据形式  
 ResultSet set = preparedStatement.executeQuery();  
 User user = new User();  
 while (set.next()){  
 //获取结果集的每一列  
 user.setId(set.getInt(1));  
 user.setUsername(set.getString(2));  
 user.setPassword(set.getString(3));  
 break;  
 }

//数据的增删改，set方法设置sql语句的参数，方法参数的表的列索引

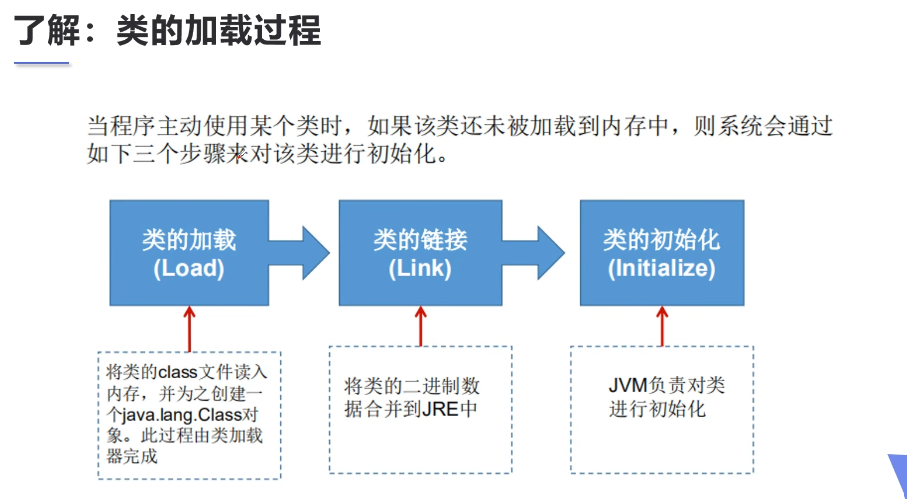
connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/chapter2"  
 ,username,password);  
String sql1 = "insert into user values (?,?,?)";  
User user = new User(7,"sss","ddd");  
//通过会话对象创建动态语句  
preparedStatement = connection.prepareStatement(sql1);  
//给动态语句设置参数值，索引从1开始  
preparedStatement.setInt(1,user.getId());  
preparedStatement.setString(2,user.getUsername());  
preparedStatement.setString(3,user.getPassword());  
//返回修改结果行数  
int rows = preparedStatement.executeUpdate();

# JVM虚拟机\*\*\*

## JAVA内存分析

1. 堆
   1. 存放new的对象和数组
   2. 可以被所有线程共享，不会存放别的对象引用
2. 栈
   1. 存放基本数据类型(包含具体数值)
   2. 引用数据类型的引用(引用指向堆内存的具体地址)
3. 方法区
   1. 可以被所有线程共享
   2. 包含了所有class和static变量

## 类的加载过程



1. 加载
   1. 将class文件加载到内存中，并将这些静态数据转化成在方法区运行时的数据结构，然后生成一个Java.lang.Class对象(Class对象在类加载中完成，所以我们无法主动创建，都是通过方法去获取)
2. 链接：将Java类的二进制代码合并到JVM的运行状态中的过程
   1. 验证：确保加载的类信息符合JVM规范，没有错误
   2. 正式为类变量(static)分配内存并设置它的默认初始值的阶段，这些内存都将在方法区中进行分配.类的静态常量(final)在此阶段调入类的常量池
   3. 解析：虚拟机常量池内的符号引用(常量名)替换为直接引用(地址)的过程
3. 初始化
   1. 执行类构造器<clinit>()的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量(static)的赋值动作和静态代码块中的语句合并产生的。(类构造器是构造类的，不是人为手写的对象构造器，这个步骤人为无法控制)
   2. 初始化类时，如果发现父类没有初始化，就需要先触发父类的构造器(super()方法)
   3. 虚拟机会保证一个类的类构造器<clinit>()方法在多线程环境中被正确加锁和同步

总结：

1. Class对象实在类加载过程完成的
2. 类成员(静态代码块，静态变量)的初始化是在类的链接过程完成的
3. 类的初始通过类构造器<clinit>()方法合并所有静态语句完成
4. 三个步骤，加载——》链接——》初始化

实例代码以及过程如下：

public class TestJVM {  
 public static void main(String[] args) {  
 A a = new A();  
 System.*out*.println(A.*m*);  
 */\*\*  
 \* 1.加载到内存方法区，产生一个类对应的Class对象  
 \* 2.链接，链接结束后，m=0，  
 \* 3.初始化<clinit>(){  
 \* System.out.println("A类静态代码块");  
 \* m=300  
 \* m=100  
 \* }  
 \* m最终等于100  
 \*/* }  
}  
class A{  
 static{  
 System.*out*.println("A类静态代码块");  
 *m* = 300;  
 }  
 static int *m* = 100;  
 public A(){  
 System.*out*.println("A类的无参构造");  
 }  
}

### 什么时候会发生类的初始化

#### 类的主动引用一定会发生类的初始化

1. 虚拟机启动，先初始化main方法所在的类
2. new一个类的对象
3. 调用类的静态成员变量和静态方法(除了final常量)
4. 使用java.lang.reflection包的方法对类进行反射调用
5. 初始化一个类，如果父类没有初始化，就先初始化它的父类

总结：

1. new会初始化
2. 初始化子类会先初始化其父类(如果其父类还没有初始化的话)
3. 反射会初始化
4. 调用类的静态方法和静态成员变量
5. main方法所在类会被初始化

#### 类的被动引用不会发生类的初始化

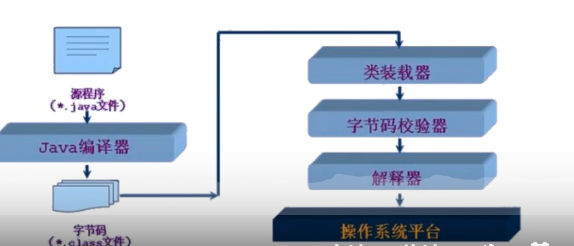
1. 访问一个静态域时，只有真正声明这个域的类才会被初始化，例如：通过子类去引用父类的静态变量，不会导致子类的初始化
2. 通过数组定义类引用，不会触发类的初始化
3. 引用常量(final)不会触发其包装类的初始化（常量在链接阶段就存入调用类的常量池中了）

实例代码如下：

public class TestJVM01 {  
 static {  
 System.*out*.println("main类被加载");  
 }  
 public static void main(String[] args) throws ClassNotFoundException {  
 //主动引用和父类先初始化  
 Son son = new Son();  
 //反射产生主动引用  
 Class.*forName*("com.study.sdm.JVM.Son");  
 //不会产生初始化,子类调用父类的静态成员  
 System.*out*.println(Son.*b*);  
 //使用数组  
 Son[]array = new Son[5];  
 //调用类的静态常量  
 System.*out*.println(Son.*x*);  
 }  
}  
class Father{  
 static int *b* = 2;  
 static {  
 System.*out*.println("父类被加载");  
 }  
}  
class Son extends Father{  
 static {  
 System.*out*.println("子类被加载");  
 *m* = 300;  
 }  
 static int *m* = 100;  
 static final int *x* = 1;  
}

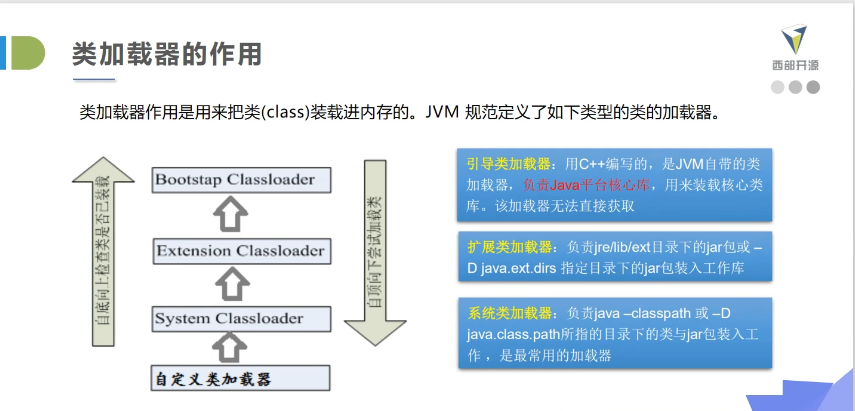
#### 类缓存

标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间，不过JVM垃圾回收机制可以回收这些Class对象



#### 类加载器

类的加载器作用是用来把类装载进内存，JVM定义了如下类型的类加载器



## JAVA GC

# Lamda表达式

## lamda表达式作用和条件

* 避免匿名内部类使用过多
* 简洁代码
* 去掉无意义代码，只留下核心部分

条件：只能针对函数式接口使用

总结表达式：

变量名（函数式接口的实例化对象） = 参数名（接口内方法参数）->{函数式接口方法体}

## 函数式接口

定义：任何接口如果只包含唯一一个抽象方法，那么它就是一个函数式接口

作用：对于函数式接口，我们可以通过lamda表达式来创建该接口的对象

简化的流程(针对的对象是函数式接口的实现类)：了解即可

外部类 -> 静态内部类 -> 局部类 -> 匿名类 -> lamda表达式

代码演示：D:\idea-workspace\JavaSE\src\com\study\sdm\Lambda\TestLamda

lamda表达式的简化过程：\*\*\*

去掉参数类型 -> 去掉参数括号 ->去掉花括号

代码演示：D:\idea-workspace\JavaSE\src\com\study\sdm\Lambda\TestLamda2

去掉参数类型中，如果有多个参数也可以去掉参数类型，要全部去掉，保持一致

去掉花括号的前提条件是方法内代码只有一行

# 正则表达式

### 概述

字符串匹配规则，应用于字符串的匹配。

java的匹配规则是贪婪匹配，当有符合的最长匹配时，优先匹配最长的。如果要使用非贪婪匹配，在限定符后面加上？变成非贪婪匹配。

### 单个符号和逻辑符号

英文句号.符号：匹配单个任意字符

例如：a.b可以匹配 aab,abb,acb,adb,a#b等等

中括号[]:只有括号内的字符才能参与匹配，也只能匹配单个字符

[^]：括号内的不接受

-：连字符

例如A-Z:表示从A到Z

[-]表示从哪到哪，是一个范围

例如[0-9]：表示从0到9，短横线出现在括号外需要转义\,表示-本身

|符号：或

例如a[a|d|cc|x]y 可以匹配aay,ady,accy,axy

^符号: [^]方括号以内的表示括号内的不能匹配。

指定的起始符号例如^[AD],表示A或者D开头的字符串

$符合：指定的结束字符。

### 匹配次数的符号

\*：0次或者多次，指定的字符重复0次或者n次

+：一次或者多次，指定的字符重复最少1次。

？：0次或者1次

{n}:n次

{n,}:最少n次

{n,m}： n次到m次之间

### 特殊符号

\S:非空字符

\s:空字符,可以匹配一个空格，制表符，回车符，换页符

\r:空格符

### 快捷符号

\d表示[0-9]，使用的时候需要带一个\转义，以下一致

\D表示[^0-9]，

\w表示[0-9a-z\_A-Z]

\W表示[^0-9A-Z\_a-z]

### 正则表达式的使用

java正则表达式通过Pattern和Matcher类实现！

Pattern：作为正则表达式的编译表示，必须将正则表达式（字符串）编译成此类的实例，然后将所得的图案可以被用来创建一个Matcher对象可以匹配任意针对正则表达式的字符或字符串！

创建实例的静态方法：static compile(String regex)

该类自带的matches方法返回匹配结果的布尔值

Matcher：

Pattern类的matcher方法会返回一个matcher实例！

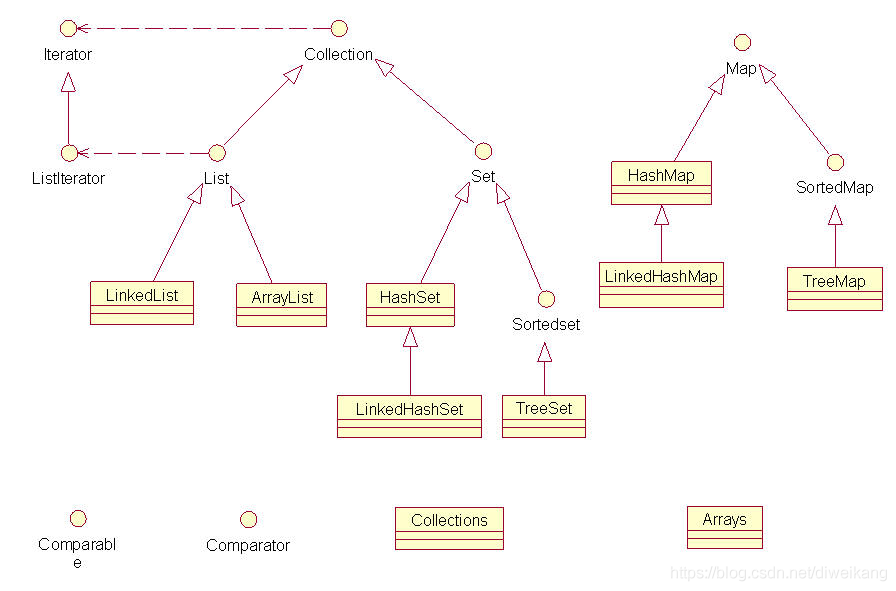
常用方法：public String replaceAll(String replacement)

将与模式匹配的输入序列的每个子序列替换为给定的替换字符串，并返回整个字符串！



# Java容器\*\*\*

框架简略图:



## Collection

Collection可以容纳一组集合元素

collection包含List 和Set，List存放有序可重复数据，Set存放无序不可重复数据

### List接口

list接口:按照元素插入的顺序存放，有序且元素可以重复。

常用方法:

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator<E> iterator();

Object[] toArray();

<T> T[] toArray(T[] a);

boolean add(E e);

boolean remove(Object o);

boolean containsAll(Collection<?> c);

boolean addAll(int index, Collection<? extends E> c);

boolean removeAll(Collection<?> c);

boolean equals(Object o);

int hashCode();

E get(int index);

E set(int index, E element);

void add(int index, E element);

E remove(int index);

int indexOf(Object o);

int lastIndexOf(Object o);

List<E> subList(int fromIndex, int toIndex);

toArray();转化成数组

#### List的实现类ArrayList集合

特点：

1. 实现了List接口，可以动态扩容，随着集合不断被添加元素，ArrayList的长度不断增加
2. 底层是数组，访问速度快，增删慢。

//源码：一个Object[]数组

transient Object[] elementData; // non-private to simplify nested class access

1. 可以存储任意类型数据。如源码所示

常用方法：

boolean add(E e) //添加元素

int size()//返回集合长度

E get(int index);//获取元素

E set(int index, E element);//设置元素

int indexof(Object o)//返回参数在集合中首次出现的位置，返回该数据的下表

boolean contains(Object o)//返回集合是否包含这个元素

boolean remove(Object o)//移除元素

boolean isEmpty()//返回是否为空

ArrayList遍历

1. 利用for循环和size方法遍历
2. 利用Iterator迭代器完成遍历

ArrayList的扩容问题：源码如下

顺序表默认初始容量是10

private static final int *DEFAULT\_CAPACITY* = 10;

private Object[] grow(int minCapacity) {  
 int oldCapacity = elementData.length;  
 if (oldCapacity > 0 ||

elementData != *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*) {  
 int newCapacity = ArraysSupport.*newLength*(oldCapacity,  
 minCapacity – oldCapacity, /\* minimum growth \*/  
 oldCapacity >> 1 /\*初始容量的一般\*/);

//扩容调用了Arrays.compyOf方法  
 return elementData = Arrays.*copyOf*(elementData, newCapacity);  
 } else {  
 return elementData = new Object[Math.*max*(*DEFAULT\_CAPACITY*, minCapacity)];  
 }  
}

*DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA为顺序表默认容量，为10，当元素达到上限且依旧要存储元素时，该顺序表会扩容至原来的1.5倍，而每一次扩容都会调用Arrays.copyof方法，创建数组再复制。如果需要存储数据量非常大，便会反复调用Arrays.compyOf方法直至容量足够，那么无形之中的性能损耗非常严重。*

*所以如果使用顺序表存储元素量比较大，可以初始话顺序表时给定一个容量参数。*

//数组初始化

public ArrayList(int initialCapacity) {  
 if (initialCapacity > 0) {  
 this.elementData = new Object[initialCapacity];  
 } else if (initialCapacity == 0) {  
 this.elementData = *EMPTY\_ELEMENTDATA*;  
 } else {  
 throw new IllegalArgumentException("Illegal Capacity: "+  
 initialCapacity);  
 }  
}

#### List的实现类LinkedList集合

底层实现原理是链表，所以查询效率低增删效率高。

LinkedList因为是链式存储，所以没有扩容问题，该类有两个构造方法，一个无参（忽略），还有一个构造方法如下，参数是一个Conllection对象！方法内部调用了addAll方法，所以链表可以将Collection（List,Set）对象转化成链式对象（ArrayList也有该构造方法）。

public LinkedList(Collection<? extends E> c) {  
 this();  
 addAll(c);  
}

#### List的实现类Vector

特点线程安全

### Set接口

无序且set对象包含的元素不能重复，常用方法:

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator<E> iterator();

<T> T[] toArray(T[] a);

boolean add(E e);

boolean remove(Object o);

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c);

boolean retainAll(Collection<?> c);

boolean removeAll(Collection<?> c);

void clear();

#### Set的实现类HashSet

实现set接口的两个子类HashSet和TreeSet:

HashSet能快速定位一个元素，但是你放到HashSet中的对象需要实现hashCode()方法。HashSet判断两个元素相等的标准:两个对象通过hashCode()方法比较相等，并且两个对象的equals()方法的返回值也相等。存放在HashSet中的元素一定要重写hashcode和equlas方法。以实现相同对象的散列码相同。

HashSet添加元素的过程:

当向HashSet添加一个元素时，HashSet会调用该对象的hashcode()方法来得到该对象的hashcode值，根据这个值决定该元素在hashset中存放的位置，如果该元素存放的位置没有元素则该元素就存放在这里，如果已经有了，则以链表形式和该位置上的元素形成一个链表。如果添加的元素和hashset中的某个元素hashcode值相同，则再比较equlas方法返回的值，如果相同，则添加失败，反转添加成功。

#### Set的实现类LinkedHashSet

HashSet的子类，底层由链表实现，元素迭代插入。

#### Set的实现类TreeSet

1. 需要添加的元素实现Comparable接口。
2. TreeSet底层使用红黑树(平衡二叉查找树)结构存储数据，TreeSet默认采用自然排序。
3. 向TreeSet中添加数据，要求数据必须是相同类的对象。

排序:

TreeSet会调用集合元素的compareTo(Object obj)方法来比较元素之间的大小关系，然后根据默认升序方式排列。如果想把一个元素加入TreeSet中，则该对象的类必须要实现

Comparable接口。实现Comparable接口的类必须实现compareTo(Object obj)方法。

向TreeSet中添加元素时，只有第一个元素无需比较compareTo()方法，后面添加的所有元素都会调用这个方法比较。

向TreeSet中添加的元素应该是同一个类的对象。

对于TreeSet集合而言，它判断连个对象是否相等的唯一标准是连个对象通过compareTo(Object obj)方法比较返回值。

Set是一种不包含重复的元素的Collection，即任意的两个元素e1和e2都有e1.equals(e2)=false，Set最多有一个null元素。

实现Set接口子类的特点：

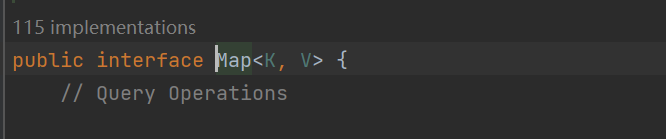
——HashSet：底层数据结构由HashMap的键来实现。不保证集合中元素的顺序，即不能保证迭代的顺序与插入的顺序一致。是线程不安全的。

——TreeSet：有序的存放，线程不安全，可以对Set集合中的元素进行排序，由红黑树来实现排序，TreeSet实际上也是SortedSet接口的子类，其在方法中实现了SortedSet的所有方法，并使用comparator()方法进行排序。

——LinkedHashSet：底层由链表实现，按照元素插入的顺序进行迭代，即迭代输出的顺序与插入的顺序保持一致

## Map接口

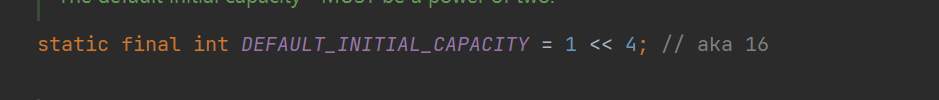
Map集合是以Key-Value键值对作为存储元素实现的哈希结构，key按照某种哈希函数计算后是唯一的，Value是可以重复的且可以是list,set,map类对象。顶层接口如下：

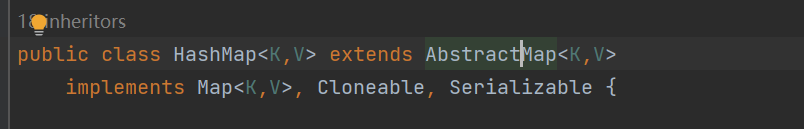


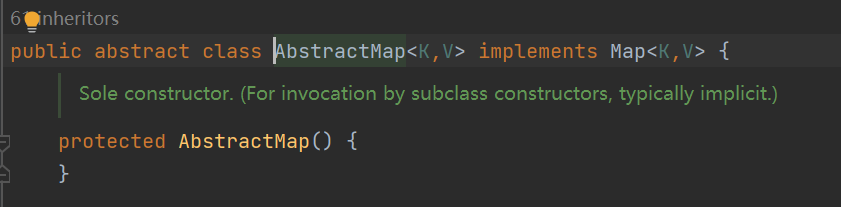
顶层Map接口提供的方法（查看源码）

### HashMap

如图：源码中hashmap的初始容量是16







1. 允许使用null键和null值（其他类map不允许key为空）
2. 所有key构成的集合是一个set，无序不可重复，所以key所在的类需要重写equals方法和hashcode方法
3. 所有的value构成的集合是collection无序可以重复，所以value所在的类需要重写equlas方法
4. 一个key一个value构成一个entry
5. 所有entry构成的集合是一个set,无序不可重复
6. 判断key相等的标准就是key的hashcode相等，equals返回true
7. 基于hash表

大多数情况下直接使用ConcurrentHashMap代替HashMap没有任何问题，性能区别不大，且线程安全。（在任何map类集合中都要避免将key设置为空，ConcurrentHashMap键不能 为空）

### LinkedHashMap

HashMap的子类

底层使用了链表

### TreeMap

底层使用了红黑树，存储entry时保证所有entry有序

存储键值对时需要给键值对排序

排序实现:键值对中的key需要实现compable接口，且key是同一个类的对象

TreeMap通过key的compareTo()方法来判断key是否相等

### 数据结构补充

#### 树

一种常用数据结构，由一个有限节点构成的一个具有层级关系的集合。数据就存储在这些节点中！根节点为最顶层的单个节点，某个节点下方没有任何节点的话就是叶子节点，从根节点到某个节点边的条数，称之为该节点的深度。

#### 二叉树

每个节点最多只有两个子节点的树为二叉树

#### 平衡二叉树

二叉树的左右高度差不能超过1，没有任何节点的空树和只有根节点的树也是平衡二叉树

#### 二叉查找树

java集合的目的就是为了加工数据，二叉查找树就是为了查找数据。

二叉查找树的定义：左子树上的所有节点值都小于它的父节点值，右子树的所有节点值都大于它的父节点值。

遍历方式：前序遍历，中序遍历，后序遍历

#### AVL树

随着二叉查找树的数据量不断增加，容易产生失衡问题，为了 保存平衡，引入了很多算法的实现，AVL树和红黑树。AVL树了解即可。

AVL树是一个二叉查找树，增加或删除节点后通过旋转树形结构重新达到平衡。

#### 红黑树

红黑树和AVL树类似，都是在插入或者删除元素时通过旋转保持平衡。红黑树确保从根节点到叶尾节点的最长路径不超过最短路径的两倍，所以它最差的运行时间是lgn

红黑树引入了5个条件：

1. 节点只能是红色或者黑色
2. 根节点是黑色
3. 所以NIL节点都是黑色
4. 一条路径上不能出现相邻的两个红色节点（有红必有黑，红红不相连）
5. 在任何递归子树内，根节点到叶子节点的所有路径上包含相同数目的黑色节点。

# JAVA API

## JSOUP爬虫

### jsoup简介

Jsoup是一款基于JAVA的HTML解析器，它提供了一套API可以直接解析某个url,html文本内容，而且可以通过类似于DOM,CSS,Jquery的方法来操作数据，所以也经常被作为爬虫工具使用。

### 相关概念简介

JSOP类的所有方法均是静态方法！

* Document:文档对象，每一个html页面都是一个文档对象，Document是jsoup体系中最顶层的结构
* Element节点对象：一个Document中可以包含着多个Element对象，可以使用Element对象来遍历节点提取数据或者直接操作HTML.(通过id选择器获取的节点就是element)
* Elements节点对象集合：类似于List<Element>（类选择器获取的节点集合就是elements对象）
* Node：节点对象，标签名称属性都是节点对象，节点对象用来存储数据

### 快速入门

public static void main(String[] args) throws Exception {  
 //目标网站  
 String url = "https://game.doaxvv.com/en/special.html";  
 //获取连接  
 Connection connection = Jsoup.*connect*(url);  
 //发生请求获取网页对象document  
 Document document = connection.get();  
 //通过类名选择器获取目标所在的容器集合  
 Elements boxs\_els = document.getElementsByClass("centering");  
 //遍历容器集合  
 for(int i = 0;i<boxs\_els.size();i++){  
 //获取每一个容器下的类名为lazy的节点集合  
 Elements img\_lists = boxs\_els.get(i).getElementsByClass("lazy");  
 for(int j = 0;j<img\_lists.size();j++){  
 String imgUrl = "https://game.doaxvv.com"+img\_lists.get(j).attr("src");  
 *download*(imgUrl);  
 }  
 }  
}  
public static void download(String imgUrl) throws IOException {  
 FileUtils.*copyURLToFile*(new URL(imgUrl),new File("D:\\DOA\\"+imgUrl.substring(imgUrl.lastIndexOf("/")+1)));  
 System.*out*.println("下载完成");  
}

### 网站的跨域问题

url的IP或者端口号不一致即为跨域！

## TIKA爬虫

## GUI编程

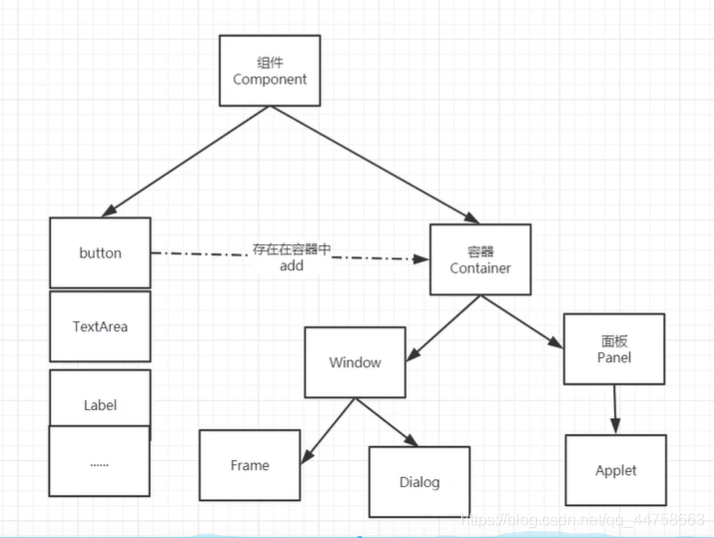
GUI！Graphical User Interface即用户图形界面编程！

GUI核心技术：Swing和AWT

编程思想：类的main方法下只能有一行代码，其作用就是启动！

### AWT

简介：用于创建图形用户界面的工具包！它提供了一系列用于实现图形界面的组件！如窗口，按钮，文本框，对话框等！这些类都位于java.awt包中



#### 组件和容器

##### Frame窗口

另一种窗口JFame,完善了关闭窗口的方法

Frame frame = new Frame("窗口"+(*f\_number*++));  
//设置窗口可见  
frame.setVisible(true);  
//设置背景色  
frame.setBackground(Color.*red*);

//设置布局

frame.setLayout(null);

//设置窗口大小  
frame.setSize(400,400);  
//设置窗口开始位置  
frame.setLocation(500,400);  
//设置窗口不可调整尺寸,窗口全屏效果关闭  
frame.setResizable(false);  
return frame;

Frame关闭窗口，原理窗口监听器

//配置监听事件：窗口关闭监听System.exit(0)  
//适配器模式  
frame.addWindowListener(new WindowAdapter() {  
 //窗口关闭的时候需要做到事情  
 public void windowClosing(WindowEvent e) {  
 System.*exit*(0);  
 }  
});

##### panel面板

内置与frame窗口，不可独立存在！面板与面板之间可以重叠放入！

另一种容器Jpanel，详情查询官方文档

//把panel添加到frame中

frame.add(panel)

//panel面板属性设置  
//相对于窗口的位置  
panel.setBounds(50,50,400,400);  
panel.setBackground(Color.*RED*);

##### button按钮

按钮的大小取决于容器以及按钮的数量，不需要手动设置！

##### TextField文本框

文本框用于获取用户输入的数据

常用方法：

1. setText():设置文本框中的内容
2. getText():获取文本框中的内容
3. TextField(int length):构造该文本框内数据内容的长度

##### 标签label

图形显示按钮

##### 三种布局管理

* 流式布局：从左到右

frame.setLayout(new FlowLayout(FlowLayout.*LEFT*));

* 东西南北中

frame.setLayout(new BorderLayout());

panel.add(b1, BorderLayout.*EAST*);

* 表格布局

frame.setLayout(new GridLayout(1,3));两个参数表示行列

##### 画笔paint

Frame窗口中重写paint方法，该方法中内置Graphics对象，会自动执行

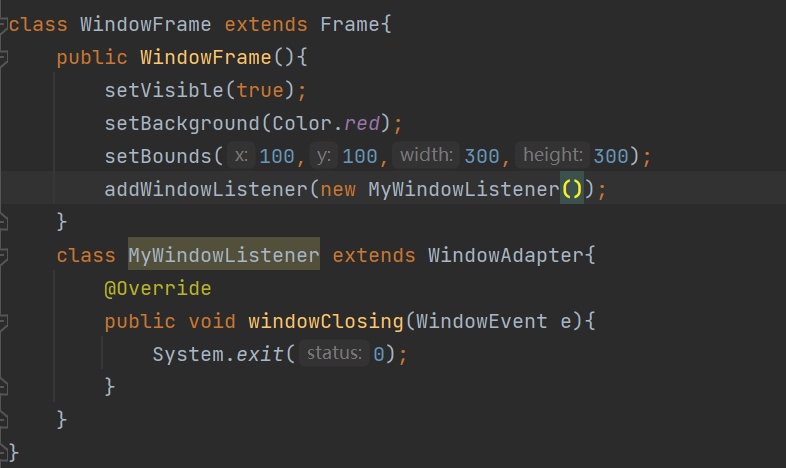
##### 鼠标监听

监听器MouseLintener

适配器MouseAdapter

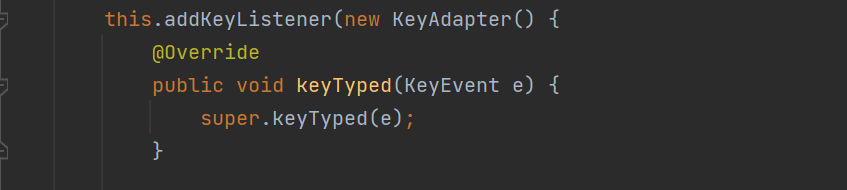
##### 窗口监听

采用窗口适配器模式实现窗口的关闭功能



##### 键盘监听

同理采用适配器类，拥有三个方法对应按键的三种状态



##### 事件监听

事件监听：监听事件的发生通过按钮button的addActionListener方法添加ActionListener的实例化对象即可！

而ActionListener是一个函数式接口！我们只需要将监听事件添加到ActionListener的唯一方法中即可！

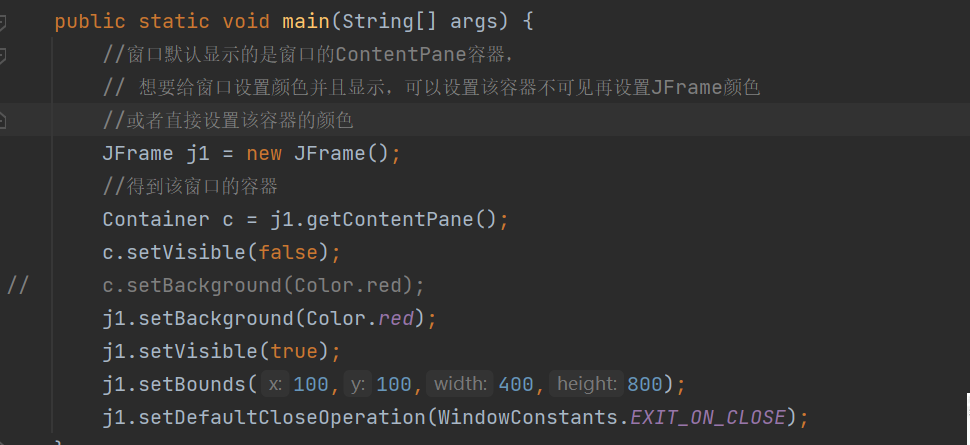
//b1是button的实例

b1.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 *WindowClose*(frame);

### Swing

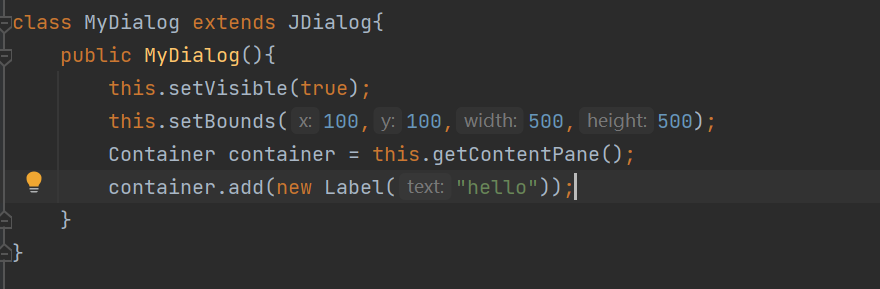
#### 窗口

Jframe默认内置一个Container容器用于显示内容



#### 弹窗

如图，弹窗和窗口类似，不过窗口设置了默认关闭方式，弹窗就不需要设置了！



#### 标签

label

#### 图标

icon

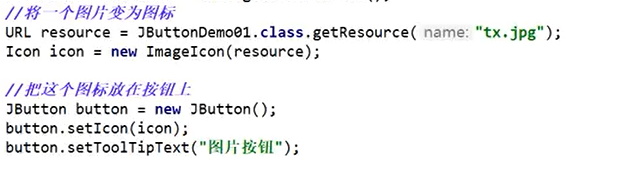
#### 面板

JPanel

#### 按钮

JButton

* 单选按钮:JRadioButton
* 多选按钮:JCheckBox



#### 列表

#### 滚动条

Jscroll

#### 文本框

JTextArea

# JAVA 学习总结

## JAVA发展史

JAVA语言于1995由Sun公司发布，次年Java开发工具包发布，即JDK,此后不断完善逐步发展成为一门成熟的语言！

JRE，即java运行环境，包括JVM虚拟机，核心类库，核心配置工具等！它是整个java的底层支撑平台！通过把源文件编译成与平台无关的字节码文件，屏蔽了平台与java源码之间的相关信息，所以java源代码不需要修改平台信息就可以跨平台运行！JVM还支撑Kotalin,Python等语言！

## JAVA编程理念

OOP理念：Object-Oriented Programming面向对象编程，万物皆对象，物指心之所指，而非只是物理存在的物！组合大于继承！

传统思维面向对象的三大特征：封装，继承，多态！

面向对象的另一核心思维：抽象

抽象可分为归纳和演绎！

归纳是从具体到本质，从个性到共性，是将一类对象的共同特征进行归一化的逻辑思维过程！属于逆向工程！

演绎是从本质到具体，从共性到个性，逐步演化的一个过程！

java之父Gosling设计的Object类，即默认父类，是对一切万事万物具体对象的抽象描述！

我是谁？getClass()

我从哪里来？Object()

我要去哪里？finalize():对象销毁时触发的方法

我和别人的是否相同：hashcode()，equals()

如何与他人协调：wait(),notify()

## 编程法则以及注意事项

设计模式七大原则之迪米特法则：

A模块使用B模块的某个接口行为，对B模块除此行为之外的其他信息尽可能知道的少。模块与模块之间忠于接口，忠于接口功能的实现即可！

因此，诞生了封装，封装属性和操作属性的最大区别，封装属性处了可以操作属性外还可以执行额外的行为，增强功能。例如监权控制，日志记录，这些都是在访问属性中无形完成的！所以在不知道什么样的访问权限合适的时候，默认private！

里氏代换法则：LSP

实际代码环境中，如果父类引用使用子类引用来代替，如果输出结果符合子类场景预期，即两个类符合LSP原则，可以使用继承关系！

方法污染：父类具备的行为通过继承传递给子类，但是子类却不具备执行此行为的能力，这就是方法污染。好比鸟会飞，鸵鸟继承鸟，但是鸵鸟不会飞一样！滥用继承会导致底层子类方法过多冗余，实际开发中产生选择困难症！

多态：继承，重写，重载的表现，一种行为的多种不同的表现！