# JAVA基础

## 类命名

public 类只能有一个，而public后面名称必须要和文件名一样

其他类也会生成class文件

标识符

只能以字母，下划线，美元符号$开头

其他部分不能以字母，下划线，$，数字以外的字符组合

不可以是关键字

java是采用unicode 字符集 地球上所有字符

## 八大基本类型



byte -127-127 1字节

short -3万-3万 2字节

int -21亿-21亿 4字节

long -922亿亿-922亿 亿 8字节 64位

float 4字节

double 8字节 64位

BigDecimal表示精确小数

char 2个字节 0 - 65535

1个字节有2个状态（0和1），1个字节有8位。 boolean 占1位

final定义常量大写。比如：final int C=35; 不能改变C值

<<相当于乘以2

>>相当于除以2取商

**int** a = 2;

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

**int** b = a<<1;//表示乘以2的1次方

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

**int** b = a >> 2; //表示除以2的次方

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

### 进制

A:二进制的数据表现形式  
    \* 由0,1组成。以0b(b可以大写也可以小写)开头(JDK1.7版本可以表示二进制了)   
\* B:八进制的数据表现形式  
    \* 由0,1,…7组成。以0开头  
\* C:十进制的数据表现形式  
    \* 由0,1,…9组成。整数默认是十进制的   
\* D:十六进制的数据表现形式  
    \* 由0,1,…9,a,b,c,d,e,f(大小写均可)。以0x开头

### 赋值

 面试题:看下面的程序是否有问题，如果有问题，请指出并说明理由。  
    \* short s=1;s = s+1;有问题：s+1在运行过程中会提升为int类型，再赋值给short类型会出错，除非强制类型转换 short（s+1）  
    \* short s=1;s+=1;没有问题：因为+=是赋值，系统会帮你强转！

### 关系运算符

== 比较的是一个结果：true，或者 false

### 三目运算符

X？Y:Z

if语句和三元运算符的区别  
      
    \* 三元运算符实现的，都可以采用if语句实现。反之不成立。  
      
    \* 什么时候if语句实现不能用三元改进呢?  
        \* 当if语句控制的操作是一个输出语句的时候就不能。  
        \* 为什么呢?因为三元运算符是一个运算符，运算符操作完毕就应该有一个结果，而不是一个输出。

### Switch(e)等值判断

long 类型不能用在switch(key)的key里面

String（JDK1.7）和枚举（JDK1.5），byte，short，char，int可以

byte，short，char都自动转换为int

case e：

break； 每一个case都需要break。不然下面继续执行。case穿透现象

default：

return，2个作用：返回值和结束方法

总结switch语句和if语句的各自使用场景  
\*     switch建议判断固定值的时候用  
\*     if建议判断区间或范围的时候用

### 逻辑运算符

&逻辑与:有false则false。  
\* |逻辑或:有true则true。  
\* ^逻辑异或:相同为false，不同为true。  
\* !逻辑非:非false则true，非true则false。  
    \* 特点：偶数个不改变本身。  
  
###03.02\_Java语言基础(逻辑运算符&&和&的区别)  
\* A:案例演示  
    \* &&和&的区别?  
        \* a:最终结果一样。  
        \* b:&&具有短路效果。左边是false，右边不执行。

### For循环

**for** (**int** i = 0; i < 10; i++) {

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

}

for(初始化表达式;条件表达式;增量表达式) {  
            循环体;  
        }

增量表达式是最后执行的步骤！！在循环体完成后，才执行！！

### Continue和break

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

**if** (x % 3 == 0) {

// 在此处填写代码

System.***out***.println("x % 3 == 0");

**continue**;

}

System.***out***.println("Java基础班");

}

Continue会执行所有循环10次。但是注意一旦执行了**continue后，后面的语句不执行！！会立即执行下一次循环！！**

**Break会立即停止循环！！**

### Return

return的作用  
    \* 返回  
    \* 其实它的作用不是结束循环的，而是结束方法的。

## 对象和类的概念

### 类：描述同一类的对象的一个抽象概念。

类定义了这一类对象的静态和动态属性。

对象是一类事物的具体实例。

成员变量=静态=属性

方法=动态

### 成员变量和局部变量

在类中的位置不同  
    \* 成员变量：在类中方法外  
    \* 局部变量：在方法定义中或者方法声明上  
\* B:在内存中的位置不同  
    \* 成员变量：在堆内存(成员变量属于对象,对象进堆内存)  
    \* 局部变量：在栈内存(局部变量属于方法,方法进栈内存)  
\* C:生命周期不同  
    \* 成员变量：随着对象的创建而存在，随着对象的消失而消失  
    \* 局部变量：随着方法的调用而存在，随着方法的调用完毕而消失  
\* D:初始化值不同  
    \* 成员变量：有默认初始化值  
    \* 局部变量：没有默认初始化值，必须定义，赋值，然后才能使用。

\* 注意事项：  
    \* 局部变量名称可以和成员变量名称一样，在方法中使用的时候，采用的是就近原则。

### 类之间的关系

关联关系：一个类方法里面的参数是另一个类的对象

继承：XX是一种XX

聚合：XX是XX的一部分。聚集，组合。

is-a 用继承 是一种用继承

has-a 用组合 拥有关系用组合

实现关系：父类定义了方法。不同的子类实现了不同的方法

### 多态

\* B:多态前提  
    \* a:要有继承关系。  
    \* b:要有方法重写。  
    \* c:要有父类引用指向子类对象。

\* A:多态中的成员访问特点  
    \* a:成员变量  
        \* 编译看左边，运行看左边。  
    \* b:成员方法  
        \* 编译看左边，运行看右边。  
    \* c:静态方法  
        \* 编译看左边，运行看左边。  
        \* (静态和类相关，算不上重写，所以，访问还是左边的)

### 开发三步

1.有哪些类，哪些对象？

2.类和对象有哪些属性和方法？

3.类和类之间是什么关系？

成员变量可以声明后，直接使用。有默认的取值。

int 0， short 0， byte 0， long 0L， double 0.0D ，float 0.0F ，boolean false ，char '\u000'，所有引用类型 null

局部变量必须先声明，再赋值，再使用。

### 引用

除了8种基本类型以外，全部是引用类型。

基本类型占1个内存。引用占2个内存

不同的对象有不同的值，方法只有一份。只有在执行方法的时候，才占内存

用new创建，使用对象.成员变量来访问，使用对象.方法来调用

同一类，多个对象有不同存储空间

同一类，多个对象共有该类方法

### 构造函数

构造函数在调用多个自己的构造函数时，只能调用一次，而且必须在该方法的第一句！！

Person(){Person(int i)…}其中Person(int i)必须位于Person()方法里面的第一句，且不能再有其他Person构造方法

错误：Person(){Person(int i);Person(String s)…}

#### 构造函数的相互调用

**public** **class** Person {

**public** **int** age;

**public** String name;

**public** Person() {

**this**(123);//调用第二个

}

**public** Person(**int** age) {

**this**(age, "yang");//调用第三个

}

**public** Person(**int** age, String name) {

**super**();

**this**.age = age;

**this**.name = name;

}

}

### 构造方法

使用new 来 构造 新对象的方法

构造方法和类名一样，没有返回值

方法里面的参数是局部变量！

如果不写构造方法，系统默认空的构造方法

如果父类构造方法设定为private那么子类无法继承

### 方法重载

一个类中相同的名字，但是参数不同或个数不同

构造方法也可以重载

this

出现在类的方法里面。

this可以看成是一个变量，他的值是当前对象的引用，指向自身

### static

static声明的变量是静态变量。是该类的公用变量。

对于该类的对象，static成员变量只有一份

static声明的方法是静态方法。在static方法中不可访问非static成员。

比如main就是static方法，不能直接访问其他类的非static成员变量

非静态的方法只能针对某个对象来调用

通过对象引用。不需要实例化，来直接访问静态成员（类名.静态成员 访问 或者 每一个对象.静态成员 都可以访问）

### package

package要放在第一句

把类放在package下面的目录里面

Import引入包

如果class A分别在package com.bug.a和package com.bug.a.debug

那么import必须明确到底是那个包下面的class A。否则报歧义错误

### 继承

extends子类通过继承，自动拥有父类的protected和public成员变量和方法

继承：XX是一种XX

只支持单继承，不支持多继承

### 访问控制

用来限定其他对象对该类对象成员的访问权限

private 同一类

default 同一包

protected 子类与同一包也可以访问

public 任何

### 方法重写

子类对父类的方法进行重写

重写的方法和被重写的方法，具有相同的方法名称，参数列表，和返回值类型

要是jdk1.5及以上 支持返回值改变为原类型的子类

如：原方法是返回Object，子类重写方法后，返回值类型可以修改为String（是Object的子类）

### super

使用super来引用父类对象

### 继承中的构造方法

子类的构造方法中，必须 先调用父类的构造方法。要造子类，先造父类

如果调用super，必须写在子类构造方法当中的第一行

子类构造方法中，没有写super。则默认调用父类为（）空的构造方法。且父类必须要有（）空的构造方法

初始化注意！！

初始化顺序是按照编写顺序从上往下

Class Person{

int i = f();

int j = g(i);

}

但是不能这样

Class Person{

int i = f(j); //因为初始化i的时候，当f()方法运行时，j并没有初始化

int j = g();

}

## 类初始化顺序：

1. 从编写顺序一个成员变量一个成员变量的从上往下
2. 先把父类的构造方法调用，再把所有的成员变量初始化完成后，再进行子类的构造方法初始化

静态初始化在类具体用到的时候，才加载执行，只加载一次。

如果只是定义Person p;不会加载Person里面的静态。

必须要使用，比如new，才会加载

非静态成员，在每次new对象的时候都需要单独加载

先加载含有main的主类里面的static。

例如：static Person p = new Person();

static Dog d = new Dog();

然后再加载其他相关类里面的静态

例如：Person和Dog类里面的static元素

若Person或Dog里面有其他非静态成员变量，

那么先加载静态，再加载父类构造器，再按照编译顺序加载非静态成员，最后加载子类构造器

当以上静态加载完后，再new对象的时候，只需要加载该对象的非静态变量，再加载构造器即可

### 代码块

\*a:局部代码块

\* 在方法中出现；限定变量生命周期，及早释放，提高内存利用率

\* b:构造代码块

\* 在类中方法外出现；多个构造方法方法中相同的代码存放到一起，每次调用构造都执行，并且在构造方法前执行

\* c:静态代码块

\* 在类中方法外出现，加了static修饰

\* 在类中方法外出现，并加上static修饰；用于给类进行初始化，在加载的时候就执行，并且只执行一次。

**class** Student {

**static** {

System.***out***.println("Student 静态代码块");//类加载的时候，且仅仅执行一次

}

{

System.***out***.println("Student 构造代码块");

}//每次new对象，调用构造方法，都会执行构造代码块。并且在构造方法之前！！

**public** Student() {

System.***out***.println("Student 构造方法");

}

}

**public** **class** StudentDemo {

**static** {

System.***out***.println("静态代码块,带你装逼带你飞");

}

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

System.***out***.println("我是main方法");

Student s1 = **new** Student();

Student s2 = **new** Student();

}

}

结果：

静态代码块,带你装逼带你飞

我是main方法

Student 静态代码块

Student 构造代码块

Student 构造方法

Student 构造代码块

Student 构造方法

Object类

是所有的类的基本类，父类

toString

描述当前对象的有关信息

默认：类名@哈希编码

一般要重写toString

hashcode

哈希编码 独一无二代表了一个对象，通过哈希编码能找到这个对象的位置

equals方法

比较的是，X,Y是否同时指向一个对象。否则返回false

查看API文档不同参数下面的equals的说明

先比较hashcode。如果hashcode不同，则肯定不是相同对象

如果hashcode相同，则再比较equals，

Equals相同，则同一个对象

Equals不同，则不是同一个对象（这种情况很少出现但是有！）

Equals相同，hashcode一定相同

hashcode相同，Equals不一定相同

## 对象转型

父类对象指向子类引用

基类的引用类型变量可以（指）new向子类对象

基类的引用不可以访问子类新增加的成员

子类新增方法无法访问！！

instanceof 来查看 是否属于该类或父类

子类对象当成基类对象使用向上转型（100%可以），向下转型（就要注意不能访问子类新增加成员）

可以通过对象强制转换来实现访问新增方法。

Animal a = **new** Dog();

a.run();//a只能访问Animal的方法

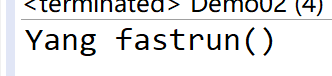
Dog a1 = (Dog) a;//强转后

a1.dogRun();//可以访问Dog的方法

如果不是新增方法，而且重写方法的话，则会触发多态

Person p = **new** Yang();

p.fastrun();//结果是执行的子类Yang的方法内容



## 多态，动态绑定，池绑定

在执行期间，判断所引用对象的实际类型，根据其实际类型调用相应的方法

父类变量 引入 子类变量的时候。执行的方法就是子类的方法

### 多态的3个条件

1.继承

2.要有方法的重写（父类和子类的方法不一样）

3.父类引用指向子类对象

## 抽象类

（抽象来配合实现多态1）100%要重写，100%要继承

abstract，只有定义。没有实现

一般对父类的 需要重写的方法 进行抽象

当类中有抽象方法的话，则必须把类，也进行抽象 抽象类是残缺的

抽象类不能被实例化，不能被new！

抽象类可以由抽象方法和普通方法一起组成

接口是用来统一行为的

抽象类是用来统一行为的具体实现（让每个子类实现共同的方法，都在抽象类里面实现）

**public** **interface** Running {

**void** run();

}

**public** **abstract** **class** AbsRunning **implements** Running {

@Override

**public** **void** run() {

System.***out***.println("各种Run共有的操作");//共有的内容在抽象类里面实现

}

}

**public** **class** FastRunning **extends** AbsRunning {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**super**.run();//很关键！具体的子类调用抽象父类共有部分！！

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

}

}

## Final

被Final修饰的成员变量，必须被初始化。否则就在构造方法中，给与初始化！！

3种给非静态final成员变量赋值

一：利用构造代码块初始化，在构造方法之前

**final** String name;

**public** Demo() {

**super**();

}

{

**this**.name = "aaa";

}

二：在构造方法中初始化

**public** Demo() {

**super**();

**this**.name = "aaa";

}

**public** Demo(String name) {

**super**();

**this**.name = name;

}

三：直接赋值

**final** String name = "yang";

2种方法给静态成员变量赋值

一：利用构造代码块初始化，在构造方法之前

**static** **final** String ***name***;

**static** {

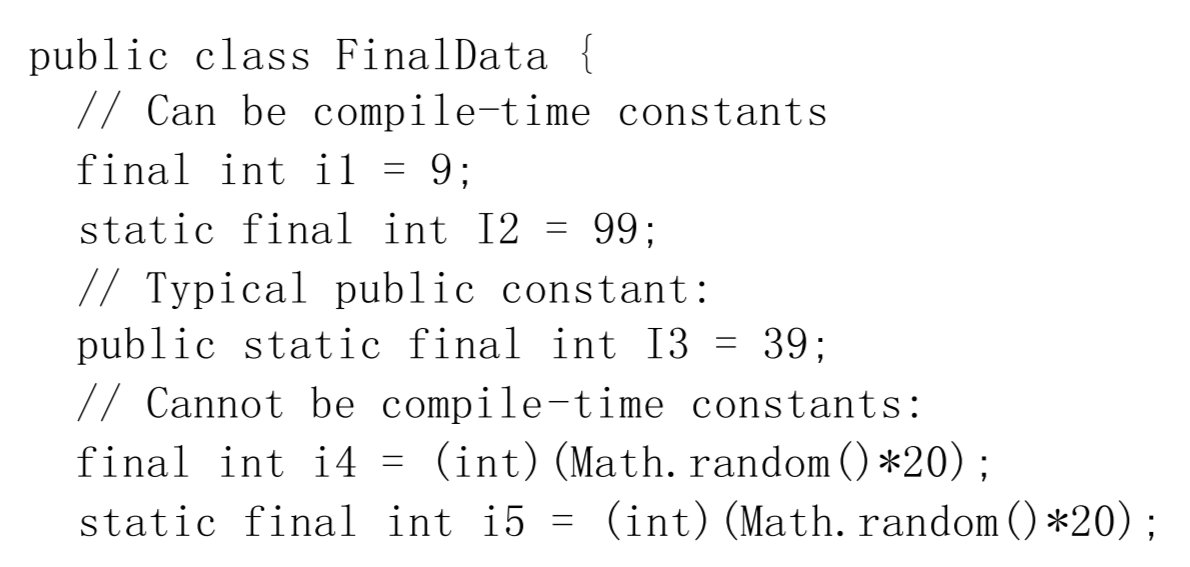
***name*** = "yyy";

}

三：直接赋值

static **final** String name = "yang";

final（方法100%不能被重写，类100%不能被继承，成员变量或局部变量的值100%不能被改变)

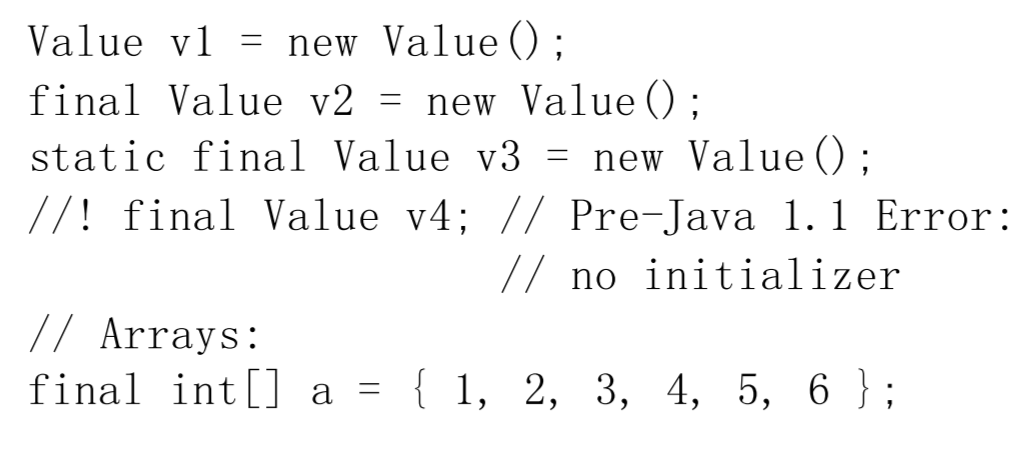


Static final编译器常数定义主数据类型表示是常数，变量要大写。如：I2，I3

由于再编译期间，无法确定i5的值，所以i5没有大写（经eclipse检验，I5大写亦可，为了规范还是小写）

i4和i5的值一旦被随机数确定后，也一样变成常量无法修改

由于i5是static。所以在类首次加载后。就被确定了。而i4由于不是static，所以i4会在每次创建对象后，随机到不一样的常数！！



final对象的时候，注意只是把引用固定。对象的值确可以改变！！

可以在声明中使用final。如：final Person p；但是一定要在构造方法中给Person p赋值。

类里面的方法如果定义为private，就等同于自动加上final

Final方法不能被重写，可以被继承

父类中如果是private final方法，则由于是private在子类中无法看见，所以子类可以重新创建一个和父类一模一样的方法。是重新创建而不是重写！！！

如果一个类为final类，由于final类无法被继承，所以等同于该类的方法都加上final

## 内部类

### 创建内部类的2个要点：

PS:实际开发中，一般不用内部类，如果非要用，则必须定义为静态内部类。因为非静态内部类会持有外部类对象的引用。这样的话，内部类对象置空，还必须要让外部类对象也置为空，才不会发生内存泄漏。如果只是把内部类对象置空，而外部类对象没有被置为空，那么外部类对象就会一直保持内部类对象的引用，就会发生内存泄漏。

1. 如果内部类是非static。那么需要拿到外部类的引用.new 内部类()。
2. 如果内部类是静态。那么创建内部类的方式：new 外部类类名.内部类()。

**new** Test().**new** InnerClass();//非static内部类

**new** Test.InnerClass();//static内部类

### 内部类的4种方式：

1. 静态内部类 A$StaticInA.class 表示类A里面有个静态内部类StaticInA
2. 非静态内部类 A$InA.class 表示类A里面有个成员内部类InA
3. 在方法体内部写类，又叫局部内部类

局部内部类，不能在外部访问

starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively fina。从javase1.8开始，局部内部类里面的方法可以访问方法里或者外部类的成员变量而不加final

A$1A\_Test.class表示类A的方法里面有个内部类A\_Test

1. 匿名内部类 A$1.class表示类A里面有个匿名内部类

A$表示成员内部类或者静态内部类

A$1XXX表示方法体内部类XXX

A$1.class表示匿名内部类

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

**private** CanFight canFight = **new** CanFight() {

@Override

**public** **void** fight() {

}

};

#### 内部类的注意事项：

1. 静态内部类不能访问外部类的非静态属性和方法
2. 内部类有一个成员（静态方法，静态变量（不包含基本数据类型或string常量），静态代码块）是static的话，那么内部类必须为静态的
3. 外类初始化，不会造成内部类的静态初始化
4. 一个类继承另一个类的内部类

**class** Outer {

**class** Inner {

**public** Inner() {

}

}

}

**public** **class** Test **extends** Outer.Inner {

**public** Test(Outer o) {

o.**super**();//此处注意，构造方法。必须先把外部类构造出来

//如果继承的Inner是static的话。则直接super()即可

}

## interface

特殊的抽象类

只有常量和方法定义！

没有变量和方法实现！

所有的方法都是abstract，所以不用注明abstract

声明的属性默认且只能是public static final。

接口还可以继承其他接口，并添加新属性和方法。

implements(实现)

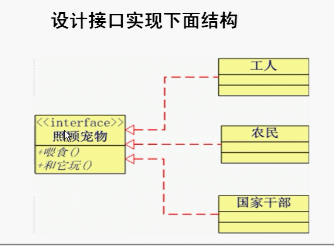
类继承类 接口和接口继承

类和接口不能继承只能实现

接口继承？多个接口？

1个类实现多个接口。多个类实现1个接口

接口的没有变量，抽象类可以有变量



## 打包

A:如何编译运行带包的类  
    \* a:javac编译的时候带上-d即可  
        \* javac -d . HelloWorld.java

-d：是指的自动创建目录

. ：这个点指的是在当前目录下。可以改为其他位置如：F:\盘

    \* b:通过java命令执行。  
        \* java 包名.HellWord

## Import

# 异常处理

## 异常类型

运行期间的错误，用于处理程序中错误的一种机制

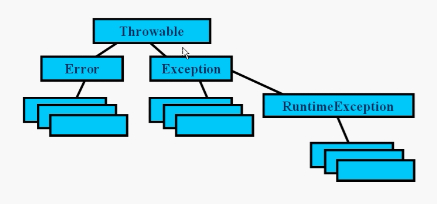
try 尝试 catch 捕获异常 1个try可以跟多个catch

Throwable Error和Exception Error是虚拟机出错了，系统错误，不用处理

其他的Exception 是一种经常出的错误，必须处理的异常！

Exception下面。其中一个是Runtime Exception

Runtime Exception 可以处理，也可以不处理



正常情况，try里面！一旦有语句报错，系统立即停止。

finally 无论是否异常，里面的代码都会执行（通常进行资源清理工作，如：关闭打开的文件，删除临时文件）

### 异常面试

**public** **class** Demo02 {

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

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

}

**private** **static** **int** getNum() {

// **TODO** Auto-generated method stub

**int** x = 0;

**try** {

x = 10;

System.***out***.println(x / 0);

**return** x;

} **catch** (Exception e) {

// **TODO**: handle exception

x = 20;

**return** x;

} **finally** {

x = 30;

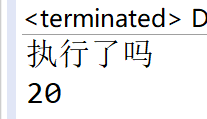
System.***out***.println("执行了吗");

// return x;

}

}

}

执行了catch里面的return后，方法返回x。但是finally里面肯定会执行。所以结果是这样。

**try** {

x = 10;

System.***out***.println(x / 0);

**return** x;

} **catch** (Exception e) {

// **TODO**: handle exception

x = 20;

**return** x;

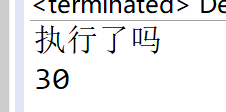
} **finally** {

x = 30;

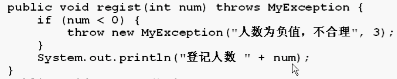
System.***out***.println("执行了吗");

**return** x;

}

因为走了catch代码块，return方法返回x=20.然后执行finally代码块。在这个代码块里面，通过return把x=30，返回了出去

5个关键字try catch finally throw



throws：方法名字后面抛出异常 先catch小的异常，再编译大的异常

## 自定义异常 class X extends Exception

子异常和父异常的关系，恰好是子类和父类的相反关系。子异常小于父异常。所以先catch捕获子异常。

如果interface接口和父类class同时有一个方法的不同异常，则以父类的方法异常为准

构造方法：子类任何的构造方法都必须包含（大于等于）父类构造方法中的全部异常

普通方法：子类方法可以声明任何父类异常的子类或者不声明亦可。（还是因为子异常小）

## 什么时候用throws，什么时候用try catch

1. 先看调用者是否需要处理异常（比如登录业务，是否需要具体区分用户名和密码错误，如果是需要区分，则在编写者位置throws用户名错误，密码错误，调用者位置，编写try-catch。如果只返回错误信息，不需要具体区分，则在编写处直接try-catch）
2. 如果自己的项目给自己用，则多用try-catch。如果jar包给别人用，则多用throws
3. Try-catch都可以用返回常量值的方式代替（比如用if-else或者switch。实际开发都用条件判断语句代替异常。效率较高）
4. Try – finally，当不想处理异常，但是又需要做某些操作的时候
5. 可以通过new RuntimeException来实现中断程序执行（最完美还是system.exit）

异常执行顺序：try，如果有异常走catch，然后在按照catch后面的代码顺序往下走，直到把方法体运行完。

当出现运行时异常，系统会在退出之前走finally区域，而不走其他代码

# 数组

## 数组注意

数组定义格式

数据类型[] 数组名 = new 数据类型[数组的长度];

不能声明数组的长度！！int a[5];非法！

动态初始化：定义和分配空间，赋值分开进行

静态初始化：定义的同时，分配空间和赋值

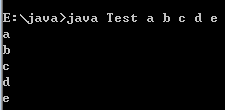
int [] a={ 1, 2, 3, 4};

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

无论一维还是二维数组。初始化的时候，都不能指定其长度！！

Int[] a = new int[i];这里i指的是数组长度，而不是索引。索引是长度-1。

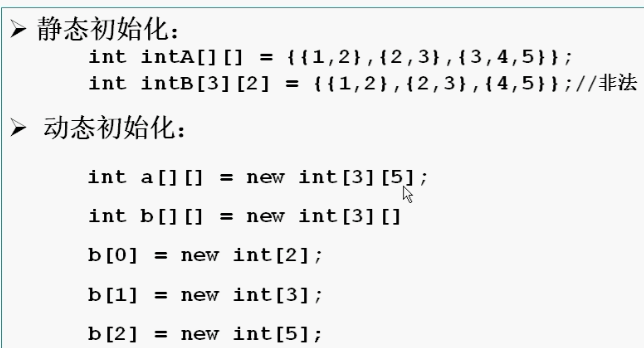
.length表明数组的长度

main(String[] args)其中args[]的作用：

parseDouble 字符串转化成Double类型

Double 转换 字符串 则用 toString

## 二维数组



二维数组的一维是引用，指向第二维的数组。

如果使用System.arraycopy（a，0，b，0，a.length）

把A的二维数组拷贝给B。则B和A同时指向第二维数组！！！所以改变B的第二维==改变A的第二维！！！

如果是一维数组，A拷贝B后，改变A，无法改变B。

### 二维数组注意

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

**int**[][] array1 = **new** **int**[3][2];//第一种创建格式

**int**[][] array2 = **new** **int**[3][];//第二种创建格式

System.***out***.println(array2[0]);//为null

array2[0] = **new** **int**[2];

array2[1] = **new** **int**[3];

array2[2] = **new** **int**[4];

System.***out***.println(array2[0]); //变成一维数组的地址

# 常用类

## String

String 不可变的字符序列

String内部是char[]数组

 new String（）实际上是创建一个对象指向char数组。然后数组再到常量池中取值

而String s = “a”。是s直接指向常量池中的值

StringBuild 可变字符序列，线程不安全的

StringBuffer 可变的字符序列，线程安全的

public static String valueOf（）将基本数据类型转换成字符串

public String[] split()可以将一个自从字符串按照指定的分隔符分隔。然后返回分隔后的字符串数组

String S=s1+s2 s1和s2先拷贝到新内存，然后S再指向到新内存

StringBuffer 可变，可以直接在后面加

String s = new String(“abc”);

创建了两个对象，一个对象在常量池，一个在堆（堆里面是常量池对象的副本）

以后创建String，都是现在常量池寻找，常量池没有的话，就创建2个对象

### 去除重复字符

String s1 = "aaa我我我.爱爱爱.爱...爱爱爱..你你";

String r1 = s1.replaceAll("\\.+", "");//首先去除.

String replaceAll = r1.replaceAll("(.)\\1+", "$1");//$1 是取分组中的内容。特定写法

//关键步骤，利用正则表达式，去掉重复内容！！

## 基础类型的包装类

Byte byte

Short short

Integer int

Character char

Boolean Boolean

Long long

Float float

JDK1.5后。会自动装箱，自动拆箱

Long a = 100L；//可以这样写

Integer i1 = 127;

Integer i2 = 127;//当值等于-128~127之间。Integer会直接从底层的一个数组中取值！。所以2个对象一样

//底层维护了一个-128~127的数组！！

i1 = 1;//当这里改变i1的值的时候，相当于又重新创建新对象，指向数组中的1！！！所以不会影响i2！！

Integer [] arr = {127};

System.***out***.println(arr[0]==i2);//一样等于true！

System.***out***.println(i1.equals(i2));

System.***out***.println(i2);

## Random

Random r2 = **new** Random();//没有种子

Random r1 = **new** Random(1L);//种子=1L

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println(r1.nextInt(10));

}

需求：如果希望每次随机出来的结果一样。则必须给一个固定种子

只要种子一样。随时生成的数字就一样！！

Random r2 = **new** Random(1L);

Random r1 = **new** Random(1L);

**for** (**int** i = 0; i < 10; i++) {

System.***out***.print(r1.nextInt(10));

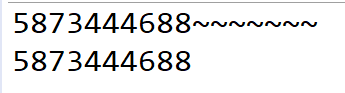
}

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

**for** (**int** i = 0; i < 10; i++) {

System.***out***.print(r2.nextInt(10));

}



如果构造方法没有传入种子，则每次生成随机数不一样

因为没有传固定种子。源码内部会根据时间毫秒数来计算随机。由于毫秒数每次不一样，所以每次随机生成不一样

public int nextInt(int n)(重点掌握) //获取0到n的随机数。含头不含尾

## Math

\* A:Math类概述  
    \* Math 类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。   
\* B:成员方法  
    \* public static int abs(int a)  
    \* public static double ceil(double a)  
    \* public static double floor(double a)  
    \* public static int max(int a,int b) min自学  
    \* public static double pow(double a,double b)  
    \* public static double random()  
    \* public static int round(float a) 参数为double的自学  
    \* public static double sqrt(double a)

Math.Random() //[0,1]含0不含1

## System

###14.13\_常见对象(System类的概述和方法使用)  
\* A:System类的概述  
    \* System 类包含一些有用的类字段和方法。它不能被实例化。   
\* B:成员方法  
    \* public static void gc()  
    \* public static void exit(int status)  
    \* public static long currentTimeMillis()  
\* C:案例演示  
    \* System类的成员方法使用

## BigInteger

###14.14\_常见对象(BigInteger类的概述和方法使用)  
\* A:BigInteger的概述  
    \* 可以让超过Integer范围内的数据进行运算  
\* B:构造方法  
    \* public BigInteger(String val)  
\* C:成员方法  
    \* public BigInteger add(BigInteger val)  
    \* public BigInteger subtract(BigInteger val)  
    \* public BigInteger multiply(BigInteger val)  
    \* public BigInteger divide(BigInteger val)  
    \* public BigInteger[] divideAndRemainder(BigInteger val)

## BigDecimal

###14.15\_常见对象(BigDecimal类的概述和方法使用)  
\* A:BigDecimal的概述  
    \* 由于在运算的时候，float类型和double很容易丢失精度，演示案例。  
    \* 所以，为了能精确的表示、计算浮点数，Java提供了BigDecimal  
  
    \* 不可变的、任意精度的有符号十进制数。  
\* B:构造方法  
    \* public BigDecimal(String val)  
\* C:成员方法  
    \* public BigDecimal add(BigDecimal augend)  
    \* public BigDecimal subtract(BigDecimal subtrahend)  
    \* public BigDecimal multiply(BigDecimal multiplicand)  
    \* public BigDecimal divide(BigDecimal divisor)  
\* D:案例演示  
    \* BigDecimal类的构造方法和成员方法使用

## Date

###14.16\_常见对象(Date类的概述和方法使用)  
\* A:Date类的概述  
    \* 类 Date 表示特定的瞬间，精确到毫秒。   
\* B:构造方法  
    \* public Date()  
    \* public Date(long date)  
\* C:成员方法  
    \* public long getTime()  
    \* public void setTime(long time)

## SimpleDateFormat

###14.17\_常见对象(SimpleDateFormat类实现日期和字符串的相互转换)  
\* A:DateFormat类的概述  
    \* DateFormat 是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。是抽象类，所以使用其子类SimpleDateFormat  
\* B:SimpleDateFormat构造方法  
    \* public SimpleDateFormat()  
    \* public SimpleDateFormat(String pattern)  
\* C:成员方法  
    \* public final String format(Date date)  
    \* public Date parse(String source)

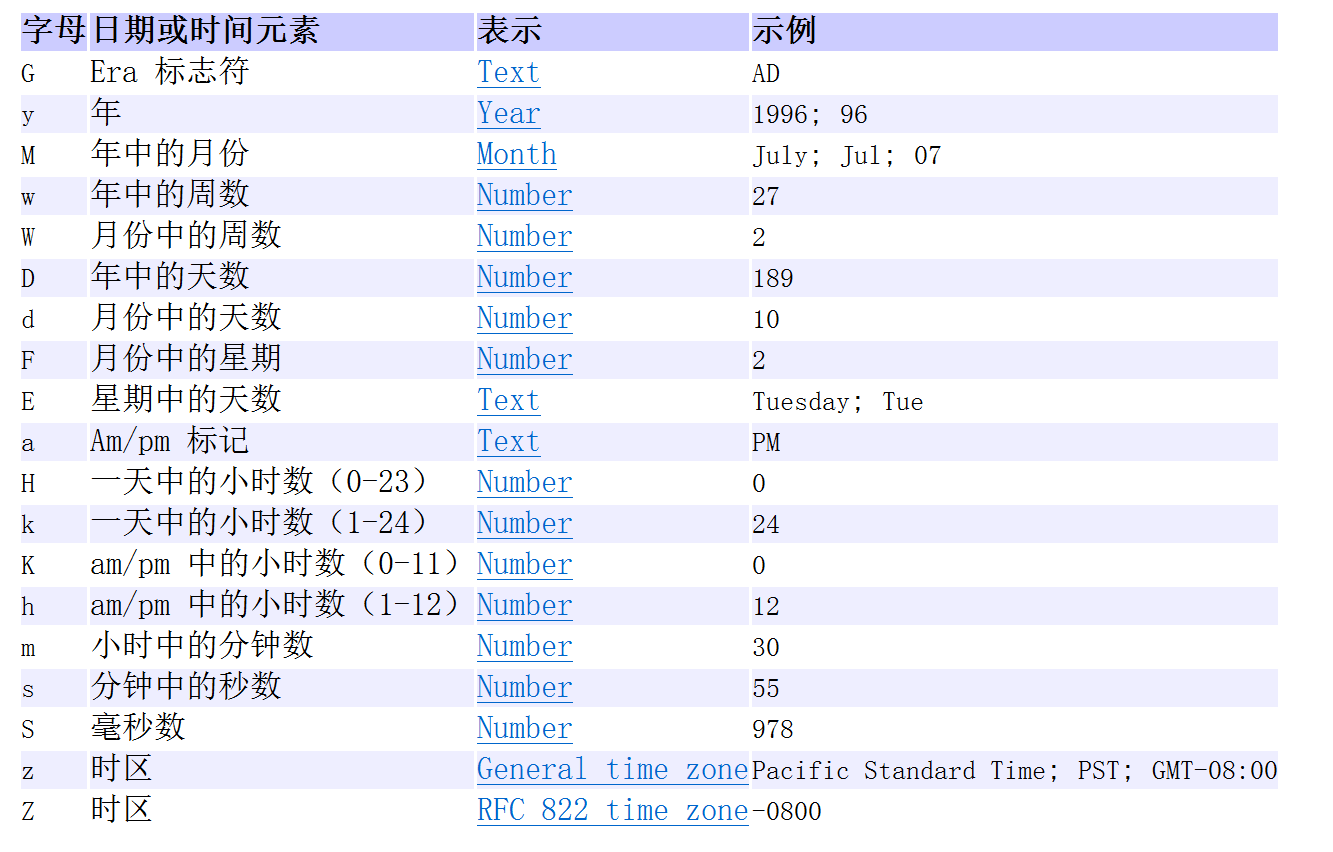
SimpleDateFormat s = **new** SimpleDateFormat("yyyy-MM-dd--hh--mm--ss");

System.***out***.println(s.format(**new** Date()));

按照自定义格式输出时间。MM是月。mm是分钟数



自定义格式具体代表如下：

  
  
###14.18\_常见对象(你来到这个世界多少天案例)  
\* A:案例演示  
    \* 需求：算一下你来到这个世界多少天?  
  
###14.19\_常见对象(日期工具类的编写和测试案例)  
\* A:案例演示  
    \* 日期工具类的编写  
    \* 日期工具类的测试

## Calendar

###14.20\_常见对象(Calendar类的概述和获取日期的方法)  
\* A:Calendar类的概述  
    \* Calendar 类是一个抽象类，它为特定瞬间与一组诸如 YEAR、MONTH、DAY\_OF\_MONTH、HOUR 等日历字段之间的转换提供了一些方法，并为操作日历字段（例如获得下星期的日期）提供了一些方法。  
\* B:成员方法  
    \* public static Calendar getInstance()  
    \* public int get(int field)  
  
  
###14.21\_常见对象(Calendar类的add()和set()方法)  
\* A:成员方法  
    \* public void add(int field,int amount)  
    \* public final void set(int year,int month,int date)  
\* B:案例演示  
    \* Calendar类的成员方法使用

## File类

是文件夹或者文件的路径！！！路径！

String path = "C:/Users/馒头大人/workspace/MyTest/file";

File dir = **new** File(path);

**if**(!dir.exists()){

dir.mkdirs();//如果目录不存在，则创建文件夹！！

//创建文件必须先创建文件夹！！

}

File f = **new** File(dir,"text.txt");//在该文件夹下，创建文件名称为text.txt的文件！！

    \* public boolean createNewFile():创建文件 如果存在这样的文件，就不创建了  
    \* public boolean mkdir():创建文件夹 如果存在这样的文件夹，就不创建了  
    \* public boolean mkdirs():创建文件夹,如果父文件夹不存在，会帮你创建出来

\* File类的删除功能  
      
    \* 注意事项：  
        \* Java中的删除不走回收站。  
        \* 要删除一个文件夹，请注意该文件夹内不能包含文件或者文件夹

## Enum

\* A:案例演示  
    \* 定义枚举类要用关键字enum  
    \* 所有枚举类都是Enum的子类  
    \* 枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，但是如果枚举类有其他的东西，这个分号就不能省略。建议不要省略  
    \* 枚举类可以有构造器，但必须是private的，它默认的也是private的。枚举项的用法比较特殊：枚举(“”);  
    \* 枚举类也可以有抽象方法，但是枚举项必须重写该方法  
    \* 枚举在switch语句中的使用

枚举类的常见方法  
    \* int ordinal() 返回该枚举类型在枚举类里面的位置。可以理解为数在数组中的索引，0开始  
    \* int compareTo(E o) 比较的就是ordinal返回的值！！  
    \* String name() 枚举项的名字  
    \* String toString()  
    \* <T> T valueOf(Class<T> type,String name)  
    \* values()   
    \* 此方法虽然在JDK文档中查找不到，但每个枚举类都具有该方法，它遍历枚举类的所有枚举值非常方便

Week[] values = Week.*values*();

System.***out***.println(Arrays.*toString*(values));

结果：

[MOM, TUE, WED]

枚举：相当于，类是Week，对象是MOM,TUE,WED

由于有抽象方法，所以相当于抽象类是Week，子类对象是MOM,TUE,WED

**public** **enum** Week {

***MOM***("星期一") {//默认无无参构造，如果是有参构造则要加括号

**public** **void** show() {

System.***out***.println("MOM(\"星期一\")");

}

},

***TUE***("星期二") {

**public** **void** show() {

System.***out***.println("MOM(\"星期二\")");

}

**public** **void** showTUE(){//该方法无法被调用！！

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

}

},

***WED***("星期三") {

**public** **void** show() {

System.***out***.println("MOM(\"星期三\")");

}

};

**private** String name;

**private** Week(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **abstract** **void** show();//所有子类必须实现的方法

**public** **void** showShare(){//所有子类公有的方法

System.***out***.println("Enum公有方法");

}

}

Week MOM = Week.***MOM***;

Week TUE = Week.***TUE***;

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

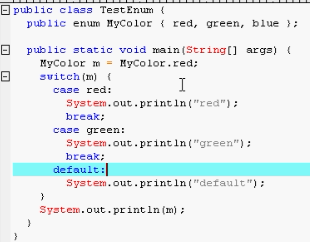
MOM.show();

MOM.showShare();

TUE.show();

TUE.showShare();

枚举类 只能在一个类型中选择



**public** **class** TestEnum {

**public** **enum** MyColor{

***red***,***bule***,***green***

};

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

MyColor m = MyColor.***red***;

**switch** (m) {

**case** ***red***:

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

System.***out***.println("switch--red");

**break**;

**default**:

**break**;

}

//枚举类重写了toString方法。返回的就是枚举名称，这里打印：red

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

}

}

## Pattern和Matcher

详见正则表达式

调用顺序是

Pattern p = Pattern.[compile](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/regex/Pattern.html#compile(java.lang.String))("a\*b");//regex

Matcher m = p.[matcher](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/regex/Pattern.html#matcher(java.lang.CharSequence))("aaaaab");//字符串

boolean b = m.[matches](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/regex/Matcher.html#matches())();//返回字符串是否匹配该regex

在仅使用一次正则表达式时，可以方便地通过此类定义 [matches](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/regex/Pattern.html#matches(java.lang.String, java.lang.CharSequence)) 方法。此方法编译表达式并在单个调用中将输入序列与其匹配。语句

boolean b = Pattern.matches("a\*b", "aaaaab");

### 需求

需求：从一段内容中，获取匹配正则表达式的内容！！

String str = "我用过的号码是:13628495530，我用过的号码是:13612345116";//一段内容

String regex = "1[358][0-9]{9}";//手机号码的正则

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

Matcher m = p.matcher(str);

**while**(m.find()){//如果能找到。则继续循环，没有找到跳出循环

System.***out***.println(m.group());//如果找到。打印找到的内容

}

Pattern p = Pattern.*compile*("");  
Matcher matcher = p.matcher("");  
while (matcher.find()){  
 String result = matcher.group();  
}

# 容器

## 容器介绍

Vector 线程安全，等同于arraylist

BitSet，位存储，放置大量正数（不能放负数）时，可以节约大量存储空间，但是读取效率较低，存的数字不可重复！

Stack，后入先出，LIFO，extends Vector

LinkedList，先进先出，FIFO

在把数组转换为list的时候，（Arrays.asList）由于返回的是固定长度的容器。所以，所有改变长度的操作都是不支持的（比如：add，remove，clear）。注意观察APi里面的（可选操作）

## 数组和集合

### 数组转集合

List = Arrays.asList(数组)

数组转集合后，不能使用add，remove改变内容！！！

意义在于，可以使用contains，或者方便的判断非空!!

String [] a = {"a","b","c"};

List<String> asList = Arrays.*asList*(a);

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

基本数据类型转List的时候，整体数组会当成1个对象！！！

只有String，或者其他引用类型会转换成每一个对象！

**int**[] b = { 11, 22, 33, 44 };

List<**int**[]> asListInt = Arrays.*asList*(b);

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

而利用Integer数组的自动拆装箱，可以完美转换成每一个对象

Integer[] b1 = { 11, 22, 33, 44 };

List<Integer> asListInt = Arrays.*asList*(b1);

**if** (asListInt.contains(11)) {

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

}

### 集合转数组

list.toArray

## Set

HashSet 底层是 HashMap的Key

TreeSet 底层是 TreeMap的Key

LinkedHashSet 底层是LinkedHashMap的Key

HashSet 无序，不重复

TreeSet 有序（按照compartor比较器来排序），不重复

LinkedHashSet 有序（按照add插入元素的顺序，来排序），不重复

## 排序

排序和搜索数组用Arrays

排序和搜索列表用Collections

\* a.自然顺序(Comparable)  
        \* TreeSet类的add()方法中会把存入的对象提升为Comparable类型  
        \* 调用对象的compareTo()方法和集合中的对象比较  
        \* 根据compareTo()方法返回的结果进行存储  
    \* b.比较器顺序(Comparator)  
        \* 创建TreeSet的时候可以制定 一个Comparator  
        \* 如果传入了Comparator的子类对象, 那么TreeSet就会按照比较器中的顺序排序  
        \* add()方法内部会自动调用Comparator接口中compare()方法排序  
    \* c.两种方式的区别  
        \* TreeSet构造函数什么都不传, 默认按照类中Comparable的顺序(没有就报错ClassCastException)  
        \* TreeSet如果传入Comparator, 就优先按照Comparator

## Map

HashMap 键值都可以传null

HashTable 键值不能为null，hashtable效率低，是同步版本的hashmap

### Map遍历

通过entrySet拿到键值对象

**for** (Entry<String, Person> entry : m.entrySet()) {

System.***out***.println(entry.getKey() + "--" + entry.getValue());

}

**for** (String s : m.keySet()) {

System.***out***.println(s + "--" + m.get(s));

}

#### 遍历HashMap的最佳方法

public static void printMap(Map mp) {

Iterator it = mp.entrySet().iterator();

while (it.hasNext()) {

Map.Entry pair = (Map.Entry)it.next();

System.out.println(pair.getKey() + " = " + pair.getValue());

it.remove(); // avoids a ConcurrentModificationException

}

}

**for** (String s : mp.keySet()) {

System.***out***.println(s + "--" + mp.get(s));

mp.remove(s);//这样会出错。只有用Iterator来删除才安全

}

### TreeMap注意

TreeSet和TreeMap 只在存入对象的一瞬间，自动排序。存入后，再修改，无法自动重新排序

如果需要增删改快，又需要排序快，键值对先用HashMap存，如果又希望排序，把hashmap添加到treemap里面。来实现自动排序。Treemap里面添加的是hashmap里面键值对的映射。

### TreeMap的排序注意

PS：Set是不可重复，但是TreeSet存入数据后，可以把某个数据改为和另一个数据一模一样。所以解决办法是，在成员变量前加final。让TreeSet存入数据后，无法更改

Arraylist，LinkedList，按照存入对象的时候，依次显示

HashSet，HashMap，存入对象，是按照hashcode取余存指定位置，所以显示的时候，不是按照存入顺序显示

### Comparable与Comparator的区别

一个类实现了Comparable接口，那么就表示该类支持排序。可以对该类的列表进行排序。比如：sort(List a)

一个比较器实现了Comparator接口。那么对列表排序时候，可以加入比较器。比如Collections.sort(List a,Comparator c);

#### Comparable接口

**Comparable接口只提供了 int compareTo(T o)方法：**也就是说假如我定义了一个Person类，这个类实现了 Comparable接口；

那么当我实例化Person类的person1后，我想比较person1和一个现有的Person对象person2的大小时，**我就可以这样来调用：**

person1.comparTo(person2),通过返回值就可以判断了；

#### Comparator接口

**而此时如果你定义了一个 PersonComparator（实现了Comparator接口）的话，那你就可以这样：**

PersonComparator comparator= new PersonComparator();

comparator.compare(person1,person2);

* **I1）改变该类的结构来实现：** 也即 要进行比较的对象所属类 实现 Comparable接口，重写方法 compareTo（）；**（涉及到 Comparable.compareTo()方法）**
* **I2）不改变该类的结果来实现：** 通过添加比较器 Comparator接口，通常的做法是 将 比较器接口 作为匿名内部类传递给 集合构造器，即可。然后在匿名内部类中重写 compare（）方法； **（涉及到 Comparator.compare()方法）**

**Comparator 是专用比较器，用 Comparator 是策略模式（strategy design pattern），就是不改变对象自身，而用一个策略对象（strategy object）来改变它的行为。**

**Comparator 是一个专用的比较器：**当这个对象不支持自比较或者自比较函数不能满足你的要求时，你可以写一个比较器来完成两个对象之间大小的比较。 可以说一个是自已完成比较，一个是外部程序实现比较的差别而已。

### Collections

排序和搜索数组用Arrays

排序和搜索列表用Collections

|  |  |  |
| --- | --- | --- |
| static   |  | | --- | | <T> boolean | | [**addAll**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#addAll(java.util.Collection, T...))**(**[**Collection**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collection.html)**<? super T> c, T... elements)            将所有指定元素添加到指定 collection 中。** |
| static boolean | [**disjoint**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#disjoint(java.util.Collection, java.util.Collection))([Collection](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collection.html)<?> c1, [Collection](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collection.html)<?> c2)            如果两个指定 collection 中没有相同的元素，则返回 true。 |
| static   |  | | --- | | <T> void | | [**copy**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#copy(java.util.List, java.util.List))**(**[**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<? super T> dest,** [**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<? extends T> src)            将所有元素从一个列表复制到另一个列表。** |
| static   |  | | --- | | <T> void | | [**fill**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#fill(java.util.List, T))**(**[**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<? super T> list, T obj)            使用指定元素替换指定列表中的所有元素。** |
| static int | [**frequency**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#frequency(java.util.Collection, java.lang.Object))**(**[**Collection**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collection.html)**<?> c,** [**Object**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/Object.html)**o)            返回指定 collection 中等于指定对象的元素数。** |
| static void | [**shuffle**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#shuffle(java.util.List))**(**[**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<?> list)            使用默认随机源对指定列表进行置换。对集合中元素洗牌** |
| static   |  | | --- | | <T extends [Comparable](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/Comparable.html)<? super T>>  void | | [**sort**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#sort(java.util.List))**(**[**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<T> list)            根据元素的自然顺序 对指定列表按升序进行排序。** |
| static void | [**swap**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#swap(java.util.List, int, int))**(**[**List**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/List.html)**<?> list, int i, int j)            在指定列表的指定位置处交换元素。** |

**Collections.**[**unmodifiableCollection**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#unmodifiableCollection(java.util.Collection))**(Collection c)让容器只读。不能修改**

**Collections.**[**synchronizedCollection**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Collections.html#synchronizedCollection(java.util.Collection))**(Collection c)返回一个线程安全（同步）的容器**

## Collection

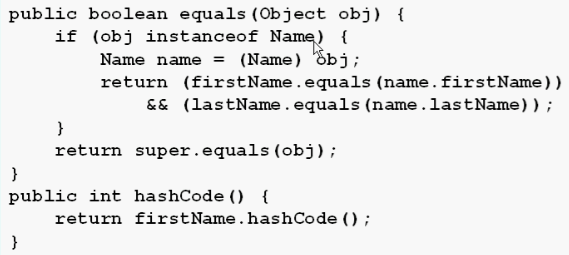
Set 没有顺序，不能重复

List 有顺序，能重复

map key键 value 值 key不能重复，value可以重复

## equals和hashCode

当集合类在调用remove，contains等方法时，需要比较对象是否相等，这会涉及对象类型的equals和hashCode方法。对于自定义的类，需要重写equals和hashCode方法来实现自定义对象的相等



重写equals方法 也要重写hashCode(这个对象当作键值，索引的时候，map)

首先equals与hashcode间的关系是这样的：

1、如果两个对象相同（即用equals比较返回true），那么它们的hashCode值一定要相同；

2、如果两个对象的hashCode相同，它们并不一定相同(即用equals比较返回false)

自我的理解：

由于为了提高程序的效率才实现了hashcode方法，先进行hashcode的比较，如果不同，那没就不必在进行equals的比较了，这样就大大减少了equals比较的次数，这对比需要比较的数量很大的效率提高是很明显的，一个很好的例子就是在集合中的使用；

我们都知道java中的List集合是有序的，因此是可以重复的，而set集合是无序的，因此是不能重复的，那么怎么能保证不能被放入重复的元素呢，但靠equals方法一样比较的话，如果原来集合中以后又10000个元素了，那么放入10001个元素，难道要将前面的所有元素都进行比较，看看是否有重复，欧码噶的，这个效率可想而知，因此hashcode就应遇而生了，java就采用了hash表，利用哈希算法（也叫散列算法），就是将对象数据根据该对象的特征使用特定的算法将其定义到一个地址上，那么在后面定义进来的数据只要看对应的hashcode地址上是否有值，那么就用equals比较，如果没有则直接插入，只要就大大减少了equals的使用次数，执行效率就大大提高了。

==是一个比较运算符号,既可以比较基本数据类型,也可以比较引用数据类型,基本数据类型比较的是值,引用数据类型比较的是地址值  
\* equals方法是一个方法,只能比较引用数据类型,所有的对象都会继承Object类中的方法,

如果没有重写Object类中的equals方法,equals方法和==号比较引用数据类型无区别,

重写后的equals方法比较的是对象中的属性

## 集合移除元素

1. **public** **class** test {
2. **public** **static** **void** main(String[] args) {
3. String str1 = **new** String("abcde");
4. String str2 = **new** String("abcde");
5. String str3 = **new** String("abcde");
6. String str4 = **new** String("abcde");
7. String str5 = **new** String("abcde");
8. List list = **new** ArrayList();
10. list.add(str1);
11. list.add(str2);
12. list.add(str3);
13. list.add(str4);
14. list.add(str5);
16. System.out.println("list.size()=" + list.size());
17. **for** (**int** i = 0; i < list.size(); i++) {
18. **if** (((String) list.get(i)).startsWith("abcde")) {
19. list.remove(i);
20. }
21. }
22. System.out.println("after remove:list.size()=" + list.size());
23. }
24. }

### ****倒过来遍历list****

**[java]** [view plain](http://blog.csdn.net/world_java/article/details/8665497) [copy](http://blog.csdn.net/world_java/article/details/8665497)

1. **for** (**int** i = list.size()-1; i > =0; i--) {
2. **if** (((String) list.get(i)).startsWith("abcde")) {
3. list.remove(i);
4. }
5. }

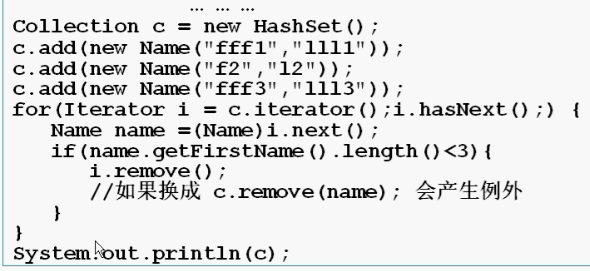
### Iterator

**使用iterator.remove()方法删除（推荐）**

**[java]** [view plain](http://blog.csdn.net/world_java/article/details/8665497) [copy](http://blog.csdn.net/world_java/article/details/8665497)

1. **for** (Iterator it = list.iterator(); it.hasNext();) {
2. String str = (String)it.next();
3. **if** (str.equals("chengang")){
4. it.remove();
5. }
6. }

Iterator对象的remove方法是在迭代过程中，删除元素的唯一安全方法。



### ListIterator

a:迭代器迭代元素，迭代器修改元素(ListIterator的特有功能add)  
 b:集合遍历元素，集合修改元素  
  
            ListIterator lit = list.listIterator();        //如果想在遍历的过程中添加元素,可以用ListIterator中的add方法  
            while(lit.hasNext()) {  
                String str = (String)lit.next();  
                if(str.equals("world")) {  
                    lit.add("javaee");      
                    //list.add("javaee");  
                }  
            }

## 底层结构

ArrayList底层实现是数组

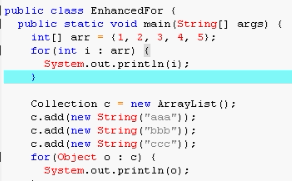
LinkedList底层实现是链表

HashMap底层实现是链表数组

HashSet底层实现是HashMap的K

### 增强的for循环

简单浏览集合内容。不能方便的访问下标值！不能方便的删除集合中的内容！



List<String> list = **new** ArrayList<>();

list.add("1");

list.add("2");

**for** (String string : list) {

list.add("3");//增强for只能遍历。不能在遍历的同时增加删除元素。否则，每次遍历都能往里面加内容。将出现永远遍历不完的情况！！

}

如果想在遍历的时候，增加或者删除元素，用listIterator

Set接口没有顺序，不能重复。有hashSet，TreeSet

List接口有顺序，可以重复。里面的元素都对应一个序号以便记住位置，可以根据序号存取集合里面的元素

Comparable接口 重写compareTo（）方法

Array 读快改慢

Linked 改快读慢

Hash 两者之间

Map接口有HashMap和TreeMap用来存储key和value。存储的键-值通过键来标识。所以键不能重复

比较每个对象的hashCode比较快。比较键很慢

key是String类型，value是Object类型·

key键 value 值都是对象 1.5后 value自动打包 自动将基础类型转换对象

1136

1个图

1个类

Collections

3个知识点

For Generic Auto-boxing/unboxing

6个接口

### 引用类型：强，软，弱，虚

强引用：StrongReference垃圾回收无论如何都不会回收

软引用：SoftReference垃圾回收时可能回收，jvm内存不够

弱引用：WeakReference垃圾回收运行时，立即回收

虚引用：PhantomReference跟踪对象回收状态，不能单独使用，必须与引用队列ReferenceQueue联合使用

### WeakHashMap

WeakHashMap会在垃圾回收运行后，立即回收new出来的KEY

虚引用：

### IdentityHashMap

IdentityHashMap比较的是地址。而不是hashcode，不是equals

（注意：如果是put的Key是常量(put(“a”,“a1”))，那么a地址是相同的，故只算1个）

IdentityHashMap:只有2个对象==的时候，（内存地址一样的时候），才不放进去

HashMap：只要内容equal一样，就不放进去

### EnumMap

EnumMap，key必须是枚举

# 泛型

定义集合的时候,同时定义集合中对象的类型

明确类型<类型>

## 泛型类

class Test <T1，T2> {T t；void test(T2 tt){}} 只能用在引用的成员变量上

**public** **class** Tool<E> {

**private** E e;

**public** E getE() {

**return** e;

}

**public** **void** setE(E e) {

**this**.e = e;

}

}

Tool<Person> tool = **new** Tool();

tool.setE(**new** Person());

Person e = tool.getE();

## 泛型接口

interface Test <T> { void test(T t);} 只能用在抽象方法上

**public** **interface** Test {

<E> **void** Test(E e);

}

**public** **interface** Test <E>{

**void** Test(E e);

}

第一种方式：直接写泛型的实现类

**public** **class** Testlmpl **implements** Test<Person>{

@Override

**public** **void** Test(Person e) {

// **TODO** Auto-generated method stub

}

}

第二种方式，在创建对象的时候，再根据泛型，创建具体的方法

**public** **class** Testlmpl<E> **implements** Test<E>{

@Override

**public** **void** Test(E e) {

// **TODO** Auto-generated method stub

}

}

## 泛型方法

用法：在public 或者 public static 之后 返回值之前。使用<T>

泛型方法：public <T> void test(T t){}

public class MethodGenericity {

public static void main(String[] args) {

String arr[]={"aaa","bbb","ccc","ddd"};

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

exchange(arr,0,3); //交换0,3位置元素

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

}

private static<T> void exchange(T[] arr, int i, int j) {

T temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

<T extends List> 类型只能接受List和List的子类

<T super List> 类型只能接受List和List的父类

## 通配符

?通配符，只能用在声明和方法的形参上

**带有通配符的参数不能使用与泛型相关的方法，例如：list.add(“hello”)编译不通过。**

**List<?> list参数中的通配符可以被赋任何值，但同时你也不知道通配符被赋了什么值。当你不知道“?”是什么时，会使你不能使用任何与泛型相关的方法**

### **List<? extends Number> list;**

**完全不能使用add方法，甚至add（new Number()）也不能用！！**

**只能使用addAll（添加一个Number集合）!!**

只能赋值给“?”Number或Number的子类型。虽然这多了一个限制，但也有好处，因为你可以用list的get()方法。就算你不知道“?”是什么类型，但你知道它一定是Number或Number的子类型。所以：Number num = list.get(0)是可以的。但是，还是不能调用list.add()方法！

/\*

\* 给通配符添加了限定：

\* 只能传递Number或其子类型

\* 子类通配符对通用性产生了影响，但使用形参更加灵活

\*/

public void print1(List<? extends Number> list) {

//list.add(new Integer(100));//错误！！！编译器报错，说明参数为泛型的方法还是不能使用（因为?也可能为Long型）

Number number = list.get(0);//正确！！！返回值为泛型的方法可用了！

}

### **List<? super Integer> list**

可以使用list.add方法

**list.add(new Integer(100))是正确的。因为无论“?”是Integer、Number、Object，list.add(new Integer(100))都是正确的**

/\*

\* 给通配符添加了限定

\* 只能传递Integer类型，或其父类型

\*/

public void print2(List<? super Integer> list) {

list.add(new Integer(100));//正确！！！参数为泛型的方法可以使用了

Object obj = list.get(0);//正确！！！但是只是得益于object类是所有类的父类，换成其他任何类编译器都会报错！说明返回值为泛型的方法，还是不能使用

}

### ****通配符小结****

1. 方法参数带有通配符会更加通用；  
2. 带有通配符类型的对象，被限制了与泛型相关方法的使用；  
3. 下边界通配符：可以使用返回值为泛型变量的方法；？super XXX？

4. 上边界通配符：可以使 方法。 ？extends XXX

* **上边界类型通配符（<? extends 父类型>）：因为可以确定父类型，所以可以以父类型去获取数据（向上转型）。但是不能写入数据。**
* **下边界类型通配符（<? super 子类型>）：因为可以确定最小类型，所以可以以最小类型去写入数据（向上转型）。而不能获取数据。**
* **无边界类型通配符（<?>） 等同于 上边界通配符<? extends Object>，所以可以以Object类去获取数据，但意义不大。**
* **下边界类型通配符（<? super 子类型>）下边界通配符<? super 子类型> + 上边界通配符<? extends Object>，所以可以以Object类去获取数据，但意义不大。**

# I/O 流

java.io

父目录不能拷贝到子目录（无限递归错误）

删除超长文件夹，需要工具：robocopy

数据流方向不同

输入流 输出流 站在程序的角度上

## 细节

### Len!=-1

每次读取字节只要是-1就返回表示结束呢？

因为int类型的-1，二进制表示是32个1.

如果是byte类型。-1是8个1。这种情况在文件内容中太容易出现了。而连续32个1，在文件内容中，是不存在的。只存在于文件结尾！！

所以while（len=fis。Read（）！=-1）。只要len不等于int类型的的-1，就表示还有内容！！

### try-with-resources

Jdk 1.7以后关闭io流写法

Try( 小括号 )

{

读写实现自动关闭

}

小括号里面的类，必须实现**interface** Closeable **extends** AutoCloseable接口

自动实现自动关闭流！

**try** (FileInputStream fis = **new** FileInputStream("a/file.txt");

)

{

**byte**[] b = **new** **byte**[1024];

**int** len;

**while** ((len = fis.read(b)) != -1) {

System.***out***.println(**new** String(b, 0, len));

}

}

## 流分类

处理数据单元不同

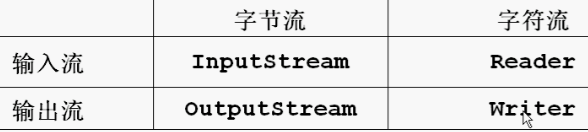
字节流 一个字节读010101 二进制，可以处理一切，文本，Doc，音频，视频

字符流 一个字符读，2个字节 文本文件，只能处理文本

功能不同

节点流 离文件最近的 从特定的数据源读写数据

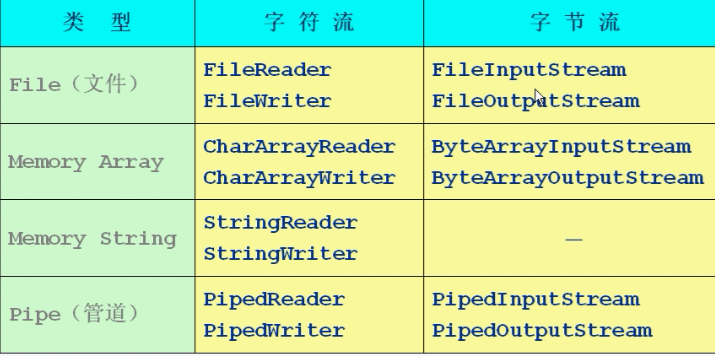
处理流 套在节点流之上的管道叫 处理流 提高效率



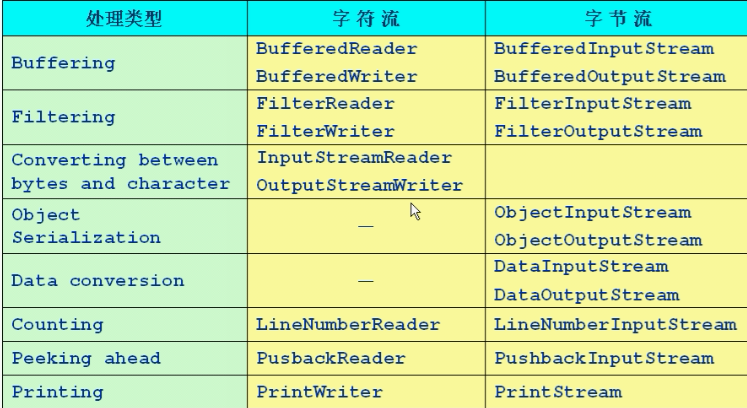
stream结尾的都是字节流 reader writer 字符流

只有OutputStream Write 要先close flush（）

## 节点流



## 处理流



## 文件输入输出流FileInputStream

FileInputStream

文件输入流。（文件路径和文件）都必须存在！！

FileoutputStream

文件输入流。文件不用存在，但是父路径必须存在！！如果不存在，需要手动创建其父文件夹！！

File parent = **new** File("ccc/ddd");

**if**(!parent.exists()){//父路径如果不存在先创建文件夹

parent.mkdirs();

}

File file = **new** File(parent,"eee.txt");//在父路径无误的前提下，就是该文件夹存在的前提下，创建名字为eee.txt的文件路径

FileOutputStream fos = **new** FileOutputStream(file);

fos.write(97);

fos.close();

New file（） 建立与文件或者文件夹的链接

### 输入顺序：

建立联系：new File();

选择流：InputStream FileInputStream

操作：byte[] car = new byte[10];//设置缓冲区大小 int len = 0 //实际读了多少

while(-1!=(len = [inputstream].read(car))){

String str = new String(car,0,len);//将字节数组转换成字符串

System.out.println(str);

}

关闭管道：finally{close();}

### 输出顺序：

建立联系：new File();

选择流：OutputStream FileOutputStream

操作：String str = “I am bug!”;

str. **getBytes**();

[Outputstream].write(str,0, str. **getBytes**().length);

.flush();

关闭管道：finally{close();}

### 注意细节：

new FileoutputStream(dest,true/flase)

如果为true则表示文件内容追加

如果为flase则表示文件内容覆盖

默认flase文件内容追加

**public** **int** read(**byte** b[]) **throws** IOException {

**return** readBytes(b, 0, b.length);

//read(字节数组)，实际上内部源码是调用readBytes(b, 0, b.length)这个方法

**public** **void** write(**byte** b[]) **throws** IOException {

writeBytes(b, 0, b.length, append);

//write(字节数组)，实际上内部源码是调用writeBytes(b, 0, b.length, append)这个方法

默认文件内容追加！

fis = **new** FileInputStream(file);

fos = **new** FileOutputStream(filecopy);

**int** len = 0;

**byte**[] byte1 = **new** **byte**[1024];

**while** (-1 != (len = fis.read(byte1))) {

//fis.read(byte1, 0, len)

fos.write(byte1, 0, len);

// fos.write(byte1);

length++;

}

## 对内存中字节数组流输入输出Bytearrayinputstream

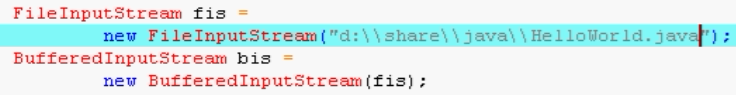
Bytearrayinputstream 对内存里面的字节数组进行读写

Bytearrayoutputstream

FileInputStream fis = new FileInputStream("a.txt");  
            ByteArrayOutputStream baos = new ByteArrayOutputStream();  
            int b;  
            while((b = fis.read()) != -1) {  
                baos.write(b);  
            }  
              
            //byte[] newArr = baos.toByteArray();                //将内存缓冲区中所有的字节存储在newArr中  
            //System.out.println(new String(newArr));  
            System.out.println(baos);  
            fis.close();

## 缓冲流

，对读写数据提供缓冲



只需要close（）buffer

## 转换流InputStreamReader

InputStreamReader

OutputStreamReader

字节流转换成字符流（处理乱码，编码集，解码集）

乱码出现2种情况：

1.解码与编码不统一（UTF-8,GBK）

2.字节缺少，长度丢失

要解码需要知道编码的字符集，否则无法处理

URL url = **new** URL("http://www.baidu.com");

//InputStreamReader可以设置解码格式：utf-8

InputStreamReader reader = **new** InputStreamReader(url.openStream(),"utf-8");

//BufferedReader专门给InputStreamReader这种类似reader的子类使用

BufferedReader bufferedReader = **new** BufferedReader(reader);

//BufferedInputStream是专门给inputstream的子类使用

// BufferedInputStream bufferedInputStream = new BufferedInputStream(reader);

String msg = **null**;

**while**((msg= bufferedReader.readLine())!=**null**){

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

}

//设置编码格式为utf-8.输出到当前工程目录下面。文件名称为：baidu.html

OutputStreamWriter writer = **new** OutputStreamWriter(**new** FileOutputStream("baidu.html"), "utf-8");

BufferedWriter bufferedWriter = **new** BufferedWriter(writer);

String msg = **null**;

**while**((msg= bufferedReader.readLine())!=**null**){

// System.out.println(msg);

bufferedWriter.append(msg);//append为BufferedWriter的新增方法

bufferedWriter.newLine();//换行

}

## 基本数据类型DataInputStream

输入输出基本数据类型（int double）！

输出，输入的基本数据类型顺序必须一致，否则会读取到错误数据！

DataInputStream

DataoutputStream

dis.readInt()

dis.readLong()

dis.readBoolean()

dis.readShort()

dis.readDouble()

dis.readFloat()

dis.readChar()

dis.readByte()

dis.read(**byte**[] b, **int** off, **int** len)

也可以输出String.

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream(**new** File("")));

dos.writeUTF(str);

DataInputStream dis = **new** DataInputStream(**new** FileInputStream(**new** File("")));

dis.readUTF();

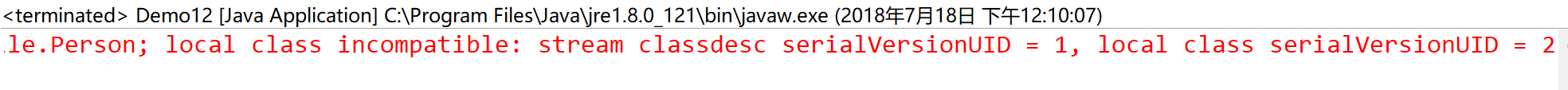
## Object流ObjectInputStream

直接将对象Object写入或读出

ObjectInputStream 反序列化 readObject()

ObjectOutputStream 序列化 writeObject()

### serialVersionUID



**类成员方法全部不变。只是修改**serialVersionUID。则会报错！！

因为java是根据serialVersionUID，来确认类版本是否相同，类似数据库升级！！

**public** **class** Person **implements** Serializable{

/\*\*

\* 反序列化的时候，如果serialVersionUID更改了。则java会认为，读的对象，不是原对象。

\* 应该先在此serialVersionUID下，先保存一遍，才可以读取！！

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

### 序列化

把一个Object对象直接转化成字节流，写到硬盘，或者网络

不是所有对象都可以序列化

**Serializable接口 是空接口。只是标识，告诉JVM该对象可以序列化**

**把类的对象**序列化成字节流 （标记性的接口 不用重写方法） JDK帮你控制序列化过程

不想序列化的属性使用transient 修饰成员变量

英文：透明的意思。在写入硬盘的时候，不予考虑

不是所有属性都需要序列化

…表示可变参数。只能是形参的最后一个位置。表示该类型不限个数。只要是该类型或者该类型的子类都可以，处理方式与数组一致

public static void close(Closeable … io){

for(Closeable temp: io){

if(null!=temp){

temp.close();

}

}

}

public static <T extends Closeable> void close(T … io){

for(Closeable temp: io){

if(null!=temp){

temp.close();

}

}

}

JDK1.7新特性：try—with—resource。try后面跟()。然后操作部分用{}。可以实现自动关闭

## Print流

打印流。没有read 只有writer

PrintWriter 针对字符

PrintStream 针对字节

Print有自动的Flush功能

输出重定向，true表示刷新flush功能开启

System.setOut(new PrintStream(new BufferedOutputStream(new FileOutputStream("c:/Test/1.txt")),true));

externalizable 接口 外部化 自己控制自己序列化过程

# 多线程

## 线程定义

线程是一个程序里面不同的执行路径

一个进程中的线程共享相同的内存单元/内存地址空间

可以访问相同的变量和对象，而且他们从同一堆中分配对象，通信，数据交换，同步操作

由于线程间的通讯在同一地址空间上进行，所以不需要额外的通讯机制

## 守护线程

非守护线程全部执行完毕后，守护线程马上挂掉

主线程也是属于非守护线程！！

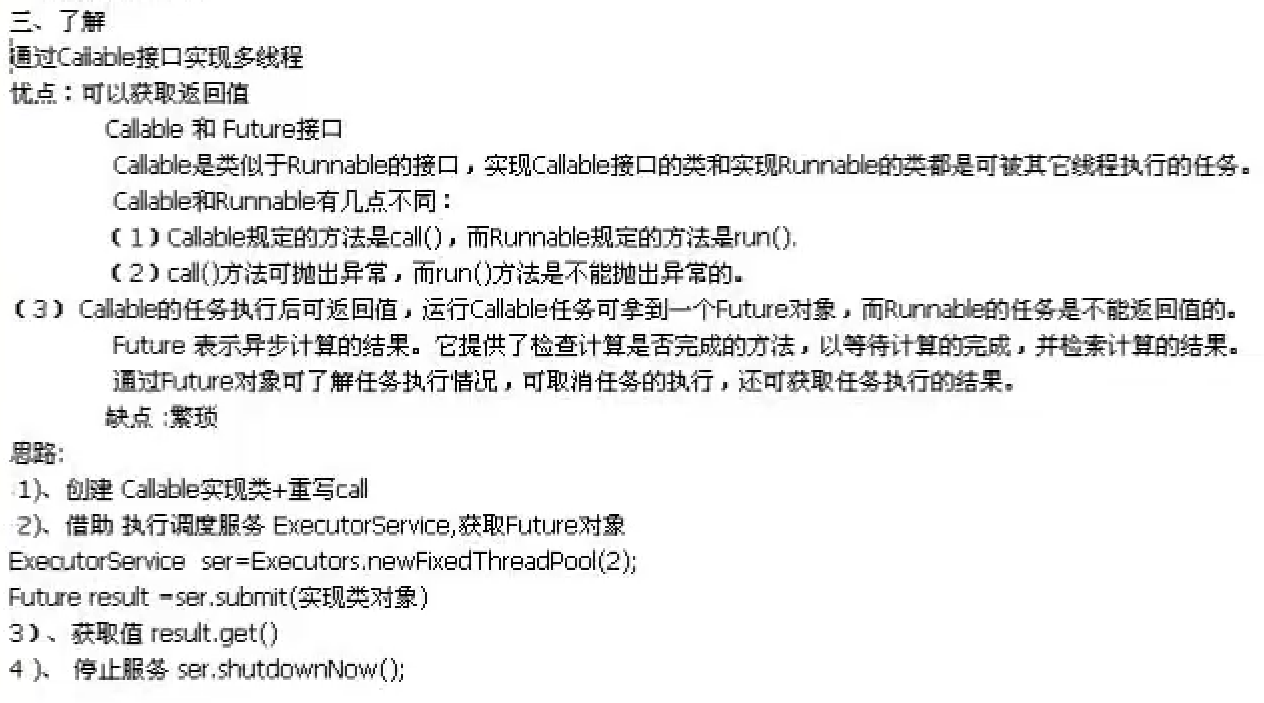
所以主线程和其他非守护线程挂掉后，守护线程立即挂掉！！

守护线程不能单独运行。

###23.11\_多线程(守护线程)  
\* setDaemon(), 设置一个线程为守护线程, 该线程不会单独执行, 当其他非守护线程都执行结束后, 自动退出  
    \*   
            Thread t1 = new Thread() {  
                public void run() {  
                    for(int i = 0; i < 50; i++) {  
                        System.out.println(getName() + "...aaaaaaaaaaaaaaaaaaaaaa");  
                        try {  
                            Thread.sleep(10);  
                        } catch (InterruptedException e) {  
                            e.printStackTrace();  
                        }  
                    }  
                }  
            };  
              
            Thread t2 = new Thread() {  
                public void run() {  
                    for(int i = 0; i < 5; i++) {  
                        System.out.println(getName() + "...bb");  
                        try {  
                            Thread.sleep(10);  
                        } catch (InterruptedException e) {  
                            e.printStackTrace();  
                        }  
                    }  
                }  
            };  
              
            t1.setDaemon(true);                        //将t1设置为守护线程  
              
            t1.start();  
            t2.start();

## Callable

PS：Callable和Runnable都是线程任务接口。不过Callable有返回值，Runnable没有



## 两种方式创建新的线程

1.实现Runnable接口

（类implements Runnable。创建该类对象，newThread（该类对象），start启动）

2.通过继承Thread类并重写run方法来启动多线程

（类extends Thread。创建该类对象，start启动）

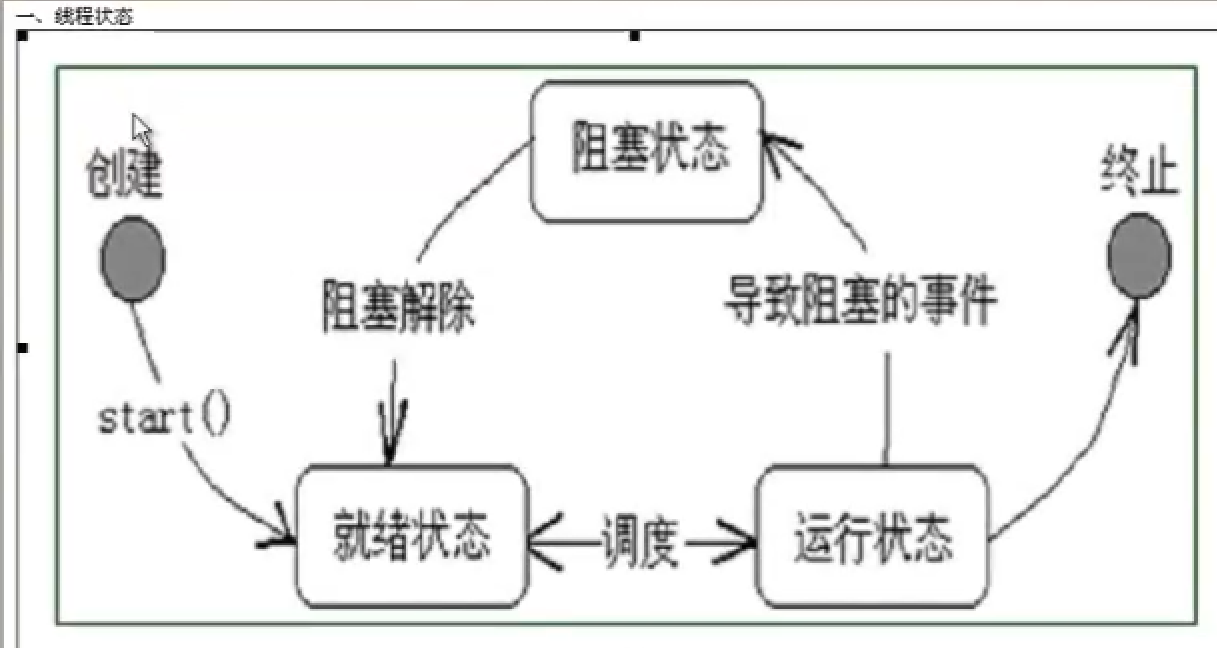
start（）就是启动线程，而不是直接调用run（）方法来启动线程

runnable接口 run方法

能使用接口。就不要通过类来继承。推荐第一种创建新线程

## 线程状态

线程状态：PS，阻塞状态解锁后是进入就绪状态，而不是运行状态！！



终止线程：1，自然终止（结束代码运行）2.外部干涉（线程内部定义标识flag，在外部调用方法来改变flag的值，来终止循环）

### sleep

sleep 使当前线程休眠。由于是静态方法，直接用类名调用Thread.sleep()

1. 模拟时间倒数
2. 模拟网络延时

### join

join 合并某个线程。

PS：join是抢夺CPU的执行权，所以必须要让join先执行完

1. 必须要在该线程start，才能合并
2. 调用者执行a.join()后，a线程里面先执行完，才执行调用者线程

a.join()后，a线程里面先执行完！！

Join(long time):

在B线程里面A.join(100):在100毫秒里面，A先执行完，B完全阻塞，相当于只阻塞100毫秒。过了100毫秒后，放行。A和B都可以执行

### yield

yield让出CPU，给其他线程执行机会。静态，直接用类名调用Thread.yield()

yield在哪个线程里面就暂停哪个线程。但是注意是这个暂停可能很快又会被CPU重新调用！！

线程的优先级别

priority优先级 默认是5 从1-10。优先级高，运行时间片多一些

## 线程同步

### synchronized

synchronized 锁定当前对象

synchronized(this){}

PS：synchronized (this)锁的是对象，而synchronized(xxx.class)锁的是xxx类！！

public synchronized void 方法名（）{}

PS：同步方法用的锁是锁this对象

如果是静态同步方法，锁的是XX类.class。字节码对象

线程死锁 锁的粒度加粗

死锁是锁里面再嵌套锁造成的！

wait（）线程等待

PS：wait是释放锁。sleep是锁定！！

notify（）唤醒在此对象监视器上等待的单个线程。

### Timer

Timer，一个后台的，一直执行的线程

使用cancel取消任务。不然会在后台一直执行

[**schedule**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, java.util.Date))([TimerTask](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/TimerTask.html) task, [Date](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Date.html) time)   
          安排在指定的时间执行指定的任务。（类似定时炸弹，定时闹钟）

TimeTask是实现Runnabe的接口，相当于也是线程

定时器调度方式

[**schedule**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, java.util.Date))([TimerTask](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/TimerTask.html) task, [Date](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Date.html) time)   
          安排在指定的时间执行指定的任务。

PS:直接设定某个特定的时间，只执行一次

[**schedule**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, java.util.Date, long))([TimerTask](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/TimerTask.html) task, [Date](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Date.html) firstTime, long period)   
          安排指定的任务在指定的时间开始进行重复的*固定延迟执行*。

PS:直接设定某个特定的时间，以后重复执行，多次，时间间隔不一定准确，比如系统延迟，垃圾回收等

[**scheduleAtFixedRate**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Timer.html#scheduleAtFixedRate(java.util.TimerTask, java.util.Date, long))([TimerTask](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/TimerTask.html) task, [Date](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Date.html) firstTime, long period)   
          安排指定的任务在指定的时间开始进行重复的*固定速率执行*。

PS:直接设定某个特定的时间，以后重复执行，多次，时间间隔相对准确，系统延迟，垃圾回收后，会通过执行多次拉加快，缩短后续的执行时间稳定！！

[**schedule**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, long))([TimerTask](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/TimerTask.html) task, long delay)   
          安排在指定延迟后执行指定的任务。

PS:当代码运行到这里时候，延迟多少时间后来执行任务，只执行一次

专门的日期工具类：quartz

可以指定星期一到天，每年的日历

**static** **class** MyChangeTimerTask **extends** TimerTask {

**static** **int** *count* = 0;

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

// if (count % 2 == 0) {

// System.out.println("boom！");

//

// new Timer().schedule(new MyChangeTimerTask(), 4000);

// } else {

// System.out.println("boom！");

//

// new Timer().schedule(new MyChangeTimerTask(), 2000);

// }

//

// count++;

*count* = (*count* + 1) % 2;//巧妙的让count一直在0和1之间无限切换！！比上面判断要有效率的多！！

System.***out***.println("boom！");

**new** Timer().schedule(**new** MyChangeTimerTask(), 2000 + 2000 \* *count*);

}

}

### Interrupted

Interrupted 强制恢复处在冻结状态的线程，不是停止线程

冻结状态：sleep，wait，join

[**setDaemon**](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/Thread.html#setDaemon(boolean))(boolean on)          将该线程标记为守护线程或用户线程。

该方法必须在启动线程前调用。

如果前台线程运行完，则jvm把守护线程也立即关闭

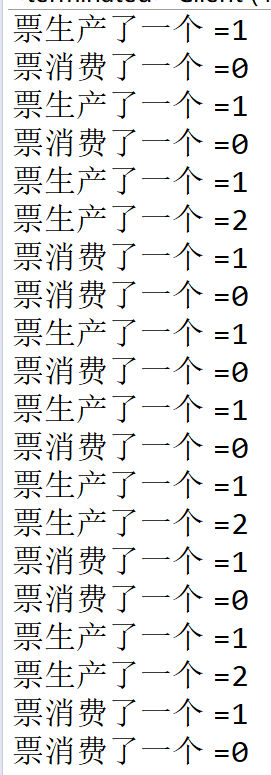
### Atomic

Atomic包里面提供了原子性操作。PS：解决多线程同时访问该对象的问题，保证同步

可以给基本数据类型，或者引用数据类型提供原子性操作。

AtomicInteger atomicInteger = new AtomicInteger(1);//new Integer(1)不自带同步

当希望100个生产者能够同时生产100个，100个消费者也能同时消费100个事件，并且没有冲突，就用Atomic

生产只管生产，消费只管消费，并且没有冲突，不会出现消费了-1的情况

**public** **class** Ticket {

**public** AtomicInteger j = **new** AtomicInteger(0);

**public** **synchronized** **void** inc() {

System.***out***.println("票生产了一个 =" + j.incrementAndGet());

**this**.notifyAll();//每生产完，就唤醒消费

}

**public** **synchronized** **void** dec() {

**while** (j.get() <= 0) {//当值小于等于0时候，表示生产者还没有生产事件，等待一下

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

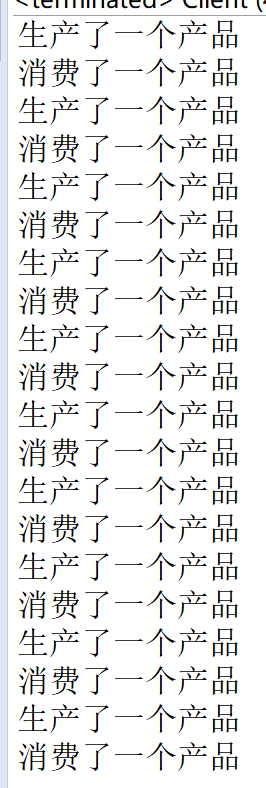
}

System.***out***.println("票消费了一个 =" + j.decrementAndGet());

}

}

**当希望无论多少个生产者，也无论多少个消费者，都是只能生产1个，然后必须等待消费1个后，再生产。就用相互唤醒模式**

**和生产者消费者数量无关，都是只生产1个后，再消费1个**

**public** **class** Goods {

**public** **static** **int** *goods* = 0;

/\*\*

\* isProducing为false时，生产 isProducing为true时，消费

\*/

**private** **boolean** isProducing = **false**;

**public** **void** produceGoods() {

**synchronized** (*this*) {

**while** (isProducing) {

**try** {

*this*.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

*goods*++;

System.***out***.println("生产了一个产品");

isProducing = **true**;

*this*.notifyAll();

}

}

**public** **void** consumerGoods() {

**synchronized** (*this*) {

**while** (!isProducing) {

**try** {

*this*.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

*goods*--;

System.***out***.println("消费了一个产品");

isProducing = **false**;

*this*.notifyAll();

}

}

}

单例模式

饿汉式

class Single {

private static Single s = new Single();

private single(){}

public static Single getInstance(){

return s;

}

}

懒汉式

PS：目的是延迟加载。但是有创建多个对象隐患，所以要加同步锁。双重判断为了提高效率

Class Single{

private static volatile Single s = null;// 必须加volatile，完美的解决双重bug

private Single(){}

public static Single getInstance(){

if(null==s){

synchronized(Single.class){

if(null==s){

s = new Single();

}

}

}

return s;

}

}

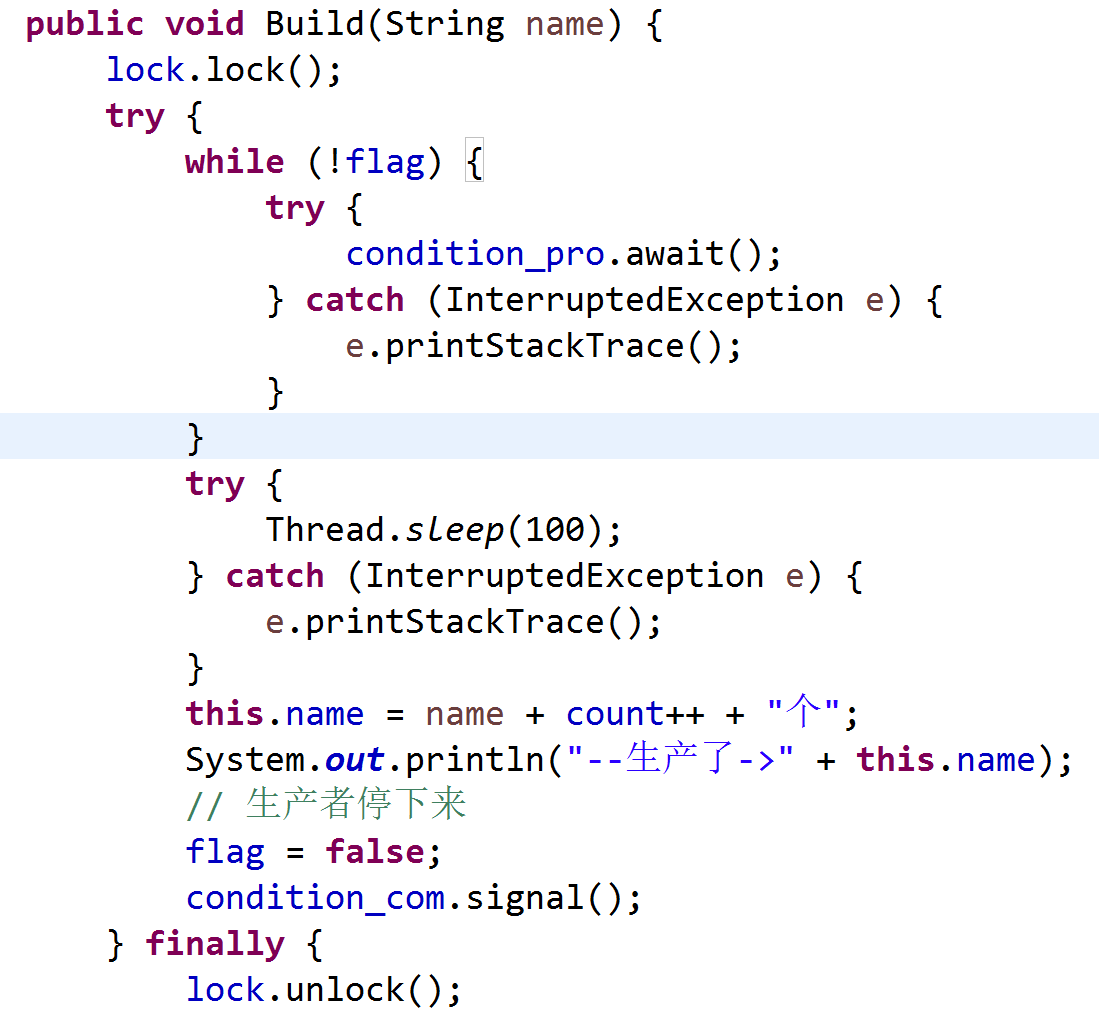
PS:写同步的时候，把run里面的内容抽到单独的一个类的一个方法里面。

在这个方法里面写synchronized,wait,notify

## Lock和Condition代替synchronized

Condition可以创建多个等待和唤醒条件对象。所以可以指定某个线程唤醒（notify无法指定）

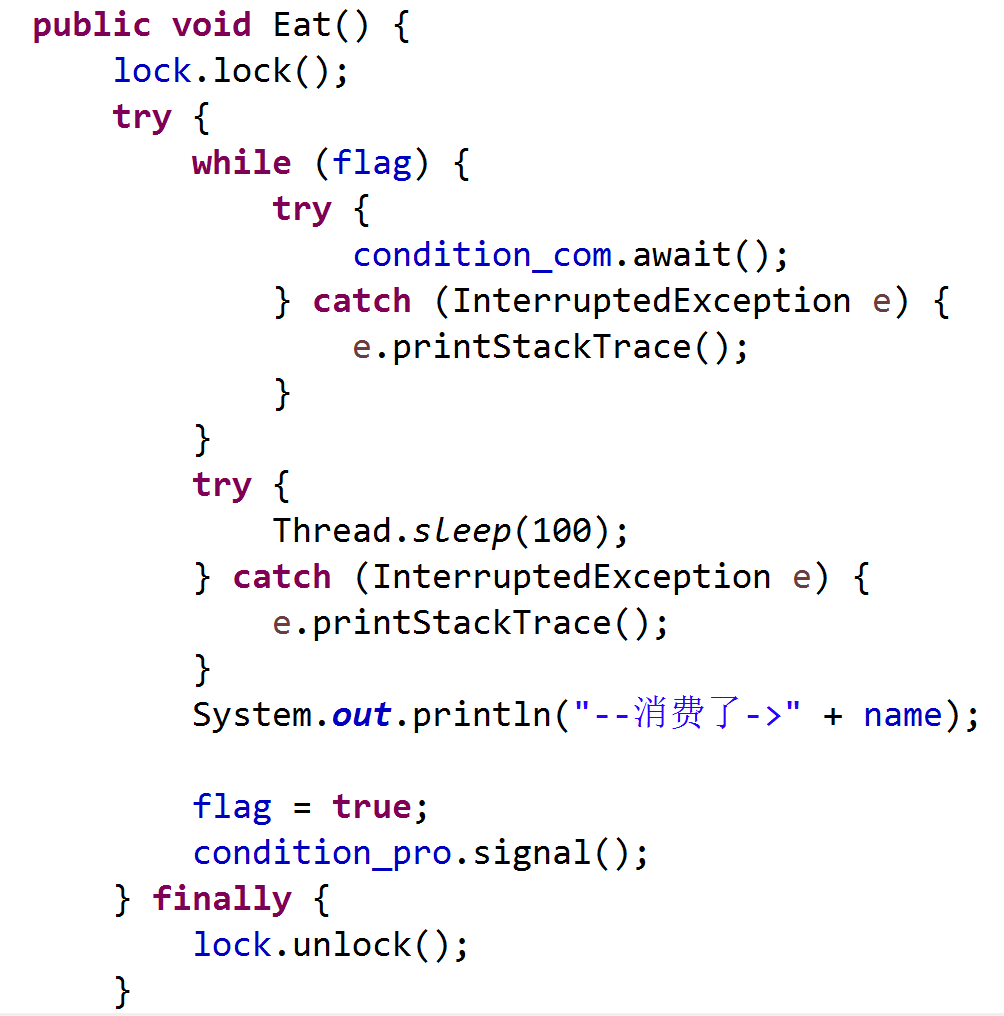




PS：

while(flag)这里一定要用while。确保线程唤醒后，再判断一次flag！！

while是多次判断，而if的话，wait唤醒后不判断flag，直接执行后面代码！！



PS：

当多个线程需要同时访问，一个线程run范围内的共享变量：可以用hashmap<Thread>，变量值>来进行多线程操作。通过调用不同的get（Thread）来获取不同的结果

## ThreadLocal

线程范围内的共享变量！

需要在线程中共享，在线程外独立！！

线程1，当中有A,B两个模块，同时访问static变量x。

线程2，当中也有A,B两个模块，也同时访问static变量x，但是希望x结果和线程1中的x互不影响

希望在线程1中，X变量取值后，给A,B两个模块访问。

线程2中，X变量取值后，给A,B两个模块访问。

线程1中的X取值，和线程2中的X取值，互不干扰！！

ThreadLocal<T> 可以方便的存取线程内的共享变量，

希望每一个线程各自处理各自的共享变量！

一个ThreadLocal只能存放一个变量。如果有多个变量，2种方式：

1. 定义多个ThreadLocal，来存放多个变量
2. 还可以把多个变量打包封装到一个对象，然后把该对象存放到ThreadLocal里面

一般采用第二种方式，把多个变量封装到一个类中ThreadScopeData。然后用单例+threadLocal存储即可！

最佳写法

## 线程池

是开启同时可运行线程的数量的池子。如：固定数量10。表示可以开启10个线程。通过executorService.execute来给池子里面增加执行任务（Runnable对象，run方法）。比如开启10个线程，但是任务加了15个任务，那么10个线程会执行10个任务，另外5个任务会等待加入，10个线程中的任务，其中一个任务执行完5个任务中就选择一个就添加进去。

执行完所有任务后，线程池中的线程还是一直保持监听状态，等待任务加入。所以如果要停止监听，只有shutdown，关闭线程池。

ExecutorService executorService = Executors.newSingleThreadExecutor();

//创建单个线程池

ExecutorService executorService = Executors.newCachedThreadPool();

//动态创建线程池（其实内部也是固定）

PS：有多个线程进来就动态增加，执行完后，线程变成空闲状态，如果空闲时间超时，就回收线程。

ExecutorService executorService = Executors.newFixedThreadPool(10);

//New 固定数量（10个）的线程池

executorService.execute(Runnable 对象);

//给线程池添加任务并执行start任务

executorService.shutdown();

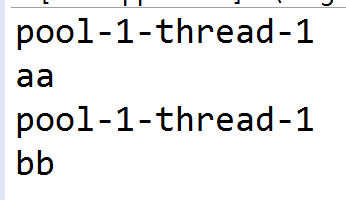
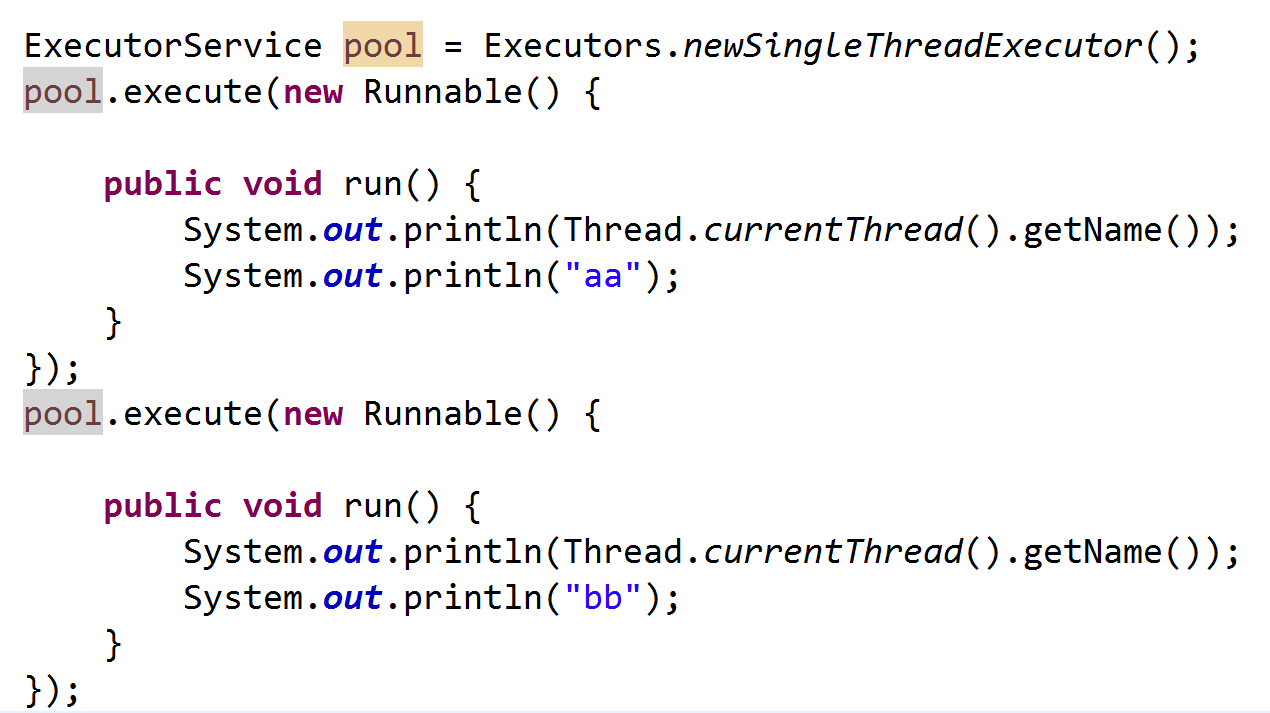
//关闭线程池，结束监听

PS：shutdown会等待，已经加入线程池的线程全部执行完毕后，才关闭线程池。***不允许新线程加入。比如延迟5秒钟后，才执行线程。如果在5秒钟之前执行shutdown，则线程无法加入***

executorService.shutdownNow();

//立即结束线程池，并且返回一个list。里面装的是还未执行的线程

### 单线程池



PS：单线程池是内部while(true)一直监听有无新线程进来，有就执行，然后把该线程删除掉，再保持继续监听，再有线程进来就再执行，再删除。一直保持监听状态。除非pool.shutdown()来结束监听。

### ScheduledExecutorService

Timer的加强版本

ScheduledExecutorService threadPool = Executors.newScheduledThreadPool(2);

//创建固定数量（2个）的延迟执行线程池

#### 和timer的区别

1. Timer不支持多线程。全部挂在Timer下的任务都是单线程的，任务仅仅能串行运行。假设当中一个任务运行时间过长。会影响到其它任务的运行，然后就可能会有各种接踵而来的问题。
2. Timer的线程不捕获异常。TimerTask假设抛出异常，那么Timer唯一的进程就会挂掉，这样挂在Timer下的全部任务都会无法继续运行。

#### scheduleAtFixedRate

Timer t = new Timer();

t.执行一个任务延期2秒

t.执行另一个任务延期2秒

以上效果等同于scheduleAtFixedRate。但是timer下的任务是单线程运行。ScheduledExecutorService是多线程同时运行！

threadPool.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + "--aa--" + task);

try {

Thread.sleep(3000);//任务消耗了3秒。

} catch (InterruptedException e){

e.printStackTrace();}

}

}, 5, 10, TimeUnit.SECONDS); //首次执行任务，延迟5秒，然后任务消耗了3秒，那么10-3=7秒。下一次任务再开启的间隔时间为7秒！

//表示线程任务初始化的时候，延迟5秒后执行，下次再执行该任务的最大间隔时间不超过10秒！！

2次任务的间隔要考虑任务本身的执行时间

#### scheduleWithFixedDelay

pool.scheduleWithFixedDelay(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(10000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}, 2, 5, TimeUnit.***SECONDS***); //首次执行任务，延迟2秒，然后任务消耗了10秒。以后每次任务执行，始终需要在上一次任务执行完毕后等待5秒！

//表示线程任务初始化的时候，延迟2秒后执行，下次再执行该任务的时间是：每次任务执行完毕后间隔5秒

2次任务的间隔不考虑任务本身的执行时间

### CompletionService

ExecutorService pool = Executors.*newFixedThreadPool*(10);

CompletionService<String> c = **new** ExecutorCompletionService(pool);

**for** (**int** i = 0; i < 10; i++) {

**final** **int** seq = i;

c.submit(**new** Callable<String>() {

**public** String call() **throws** Exception {

**return** "bug" + seq;

}

});

}

**for** (**int** i = 0; i < 10; i++) {

**try** {

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

} **catch** (InterruptedException e) {

e.printStackTrace();

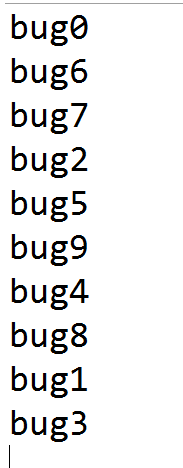
} **catch** (ExecutionException e) {

e.printStackTrace();

}

}

pool.shutdown();

 CompletionService 类似于收麦子，哪个callable里面的线程先返回其值，就先拿到其值

c.take().get()//take：表示主线程运行到这里时，一直等待取结果。阻塞式

c.poll().get()//poll：表示当主线程运行到这里时，如果callable里面还没有返回值，则直接返回null

## ReentrantReadWriteLock

读写锁

rwl.readLock()//允许多用户同时read同一数据

rwl,writeLock()//用户同时修改一个数据的时候，则必须保持同步，同一时刻只能有一个线程来访问

PS：读写锁必须在同一个线程里面才有效果

以下例子是缓存数据的读写锁嵌套使用：

class CachedData {

Object data;

volatile boolean cacheValid;

ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

void processCachedData() {

rwl.readLock().lock();

if (!cacheValid) {

// Must release read lock before acquiring write lock

rwl.readLock().unlock();

rwl.writeLock().lock();

// Recheck state because another thread might have acquired

// write lock and changed state before we did.

if (!cacheValid) {

data = ...

cacheValid = true;

}

// Downgrade by acquiring read lock before releasing write lock

rwl.readLock().lock();//在写锁释放之前加入读锁，可以让写锁降级为读锁，反之不行

rwl.writeLock().unlock(); // Unlock write, still hold read

}

use(data);

rwl.readLock().unlock();

}

}

PS：

1. 读取锁在没有unlock释放之前，无法加入writeLock。

## 类 Semaphore。信号灯

类似厕所。比如：线程池可以放10个固定***（需要比信号灯数量多，不然信号灯无意义）***或者动态创建线程的位置。但是只开启5个位置（通过信号灯来开启），来了8个人，就只能有5个位置执行，当执行完1个线程任务后，再在剩下3个人中选择1个再来执行，总之：只有拿到1个信号灯，才有权力执行任务

***可以一个线程获取许可，再由另一个线程来释放许可，用于死锁恢复***

Semaphore semaphore = new Semaphore(int permits);

//创建信号灯个数

Semaphore.acquire()

//获取一个许可，来执行任务

Semaphore.release()

//释放一个许可。和acquire匹配使用

## CyclicBarrier类

设置一个集合点（路障），等所有***线程任务都执行***到该集合点（路障）后，再一起执行剩下线程

PS：最好配合动态创建线程池newCachedThreadPool使用，或者配合固定线程池newFixedThreadPool(10)来使用。但是特别注意：固定线程池的大小一定一定要大于等于[CyclicBarrier](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/CyclicBarrier.html#CyclicBarrier(int))(int parties)里面的parties数量。因为比如路障设置需要集合5个线程。但是固定线程池却开启3个，那么当线程任务执行到路障的时候，线程无法继续执行，进入等待状态。但是线程池中的3个线程由于进入等待状态，还没有结束，不会释放线程池中的位置，所以由于新线程无法加入路障从而造成死锁！

[CyclicBarrier](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/CyclicBarrier.html#CyclicBarrier(int))(int parties)

//创建需要集合多少个线程后才放行的一个路障

[await](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/CyclicBarrier.html#await())()

//在满足指定数量的线程任务全部到达路障后，再一起执行任务

## CountDownLatch类

设置一个倒计时路障。在运行线程任务的过程中，当执行到countDownLatch.countDown()的时候计数器减1。直到计数器为0的时候，路障才放行。和CyclicBarrier不同的是，CyclicBarrier是阻塞的，每个线程任务在执行的时候cyclicBarrier.[await](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/CyclicBarrier.html#await())(),所有线程任务都是等待阻塞状态。而执行countDownLatch.countDown()的时候，不是阻塞，线程程序会继续向下执行，只是计数器减1。

CountDownLatch countDownLatch = new CountDownLatch(10);

//创建一个倒计时

countDownLatch.countDown();

//计数器减1

countDownLatch.await();

//当计数器=0的时候，线程任务全部放行

## Exchanger类

主要用户交换任意2个线程任务里面的任何对象，PS：如其中1个线程任务先到达交换点，则处于阻塞状态，一直等待到数据交换成功后才解除阻塞

Exchanger<Integer> exchanger = new Exchanger<Integer>();

//创建传送交换空间

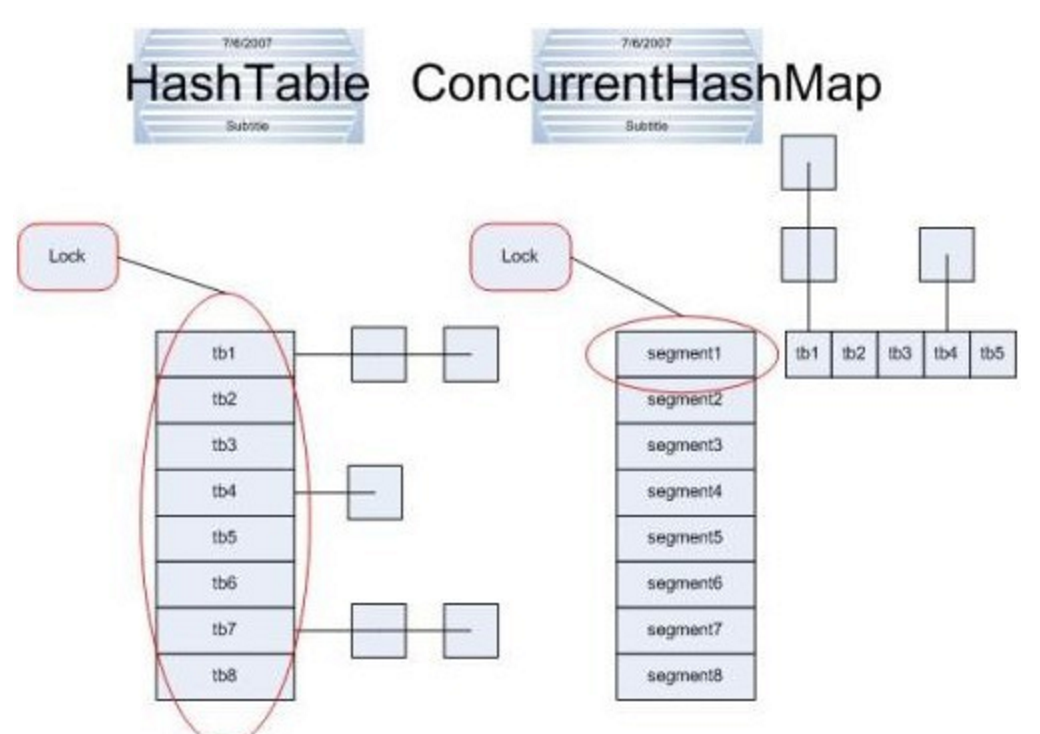
Integer exchangeI = exchanger.exchange(i);

//分别在不同的2个线程任务里面，设置要交换的数据，即可实现数据交换，注意泛型和返回值

## ConcurrentHashMap

类 ConcurrentHashMap<K,V> 等同于 HashMap的同步版本

concurrenthashmap和hashtable区别：主要是读写的同步方式完全不同。



hashtable每次同步执行的时候都要锁住整个结构。

ConcurrentHashMap锁的方式是稍微细粒度的。

更令人惊讶的是ConcurrentHashMap的读取并发，因为在读取的大多数时候都没有用到锁定，所以读取操作几乎是完全的并发操作，而写操作锁定的粒度又非常细，比起之前又更加快速（这一点在桶更多时表现得更明显些）。只有在求size等操作时才需要锁定整个表。

而在迭代时，ConcurrentHashMap使用了不同于传统集合的快速失败迭代器的另一种迭代方式，我们称为弱一致迭代器。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在***改变时new新的数据从而不影响原有的数据***，iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变，更重要的，这保证了多个线程并发执行的连续性和扩展性，是性能提升的关键。

类 ConcurrentSkipListSet<E> 等同于 TreeSet 的同步版本

类 ConcurrentSkipListMap<K,V>等同于TreeMap的同步版本

类 CopyOnWriteArrayList<E>等同于ArrayList的同步版本

## CopyOnWriteArrayList

public class TestCopyOnWriteArrayList {

public static void main(String[] args) {

// ArrayList arrayList = new ArrayList();

CopyOnWriteArrayList arrayList = new CopyOnWriteArrayList();

arrayList.add(1);

arrayList.add(2);

arrayList.add(3);

Iterator iterator = arrayList.iterator();

while (iterator.hasNext()) {

Integer Integer = (Integer) iterator.next();

arrayList.add(4);//模拟遍历的时候，正在对集合增加数据

if (Integer == 1) {

arrayList.remove(Integer);

// iterator.remove();

} else {

System.out.println(Integer);

}

}

}

}

传统的ArrayList在进行遍历的时候，无法自身修改arrayList.remove(Integer)。只能iterator.remove()。这样的话同时并发访问遍历的时候，就无法做到去修改数据。或者修改数据的时候去遍历也会报错

而CopyOnWriteArrayList的强大在于允许并发遍历的时候，也可以同时修改数据。原理是每次创建遍历Iterator iterator = arrayList.iterator()的时候，先拷贝Copy一份List（所以开销很大），把遍历和修改分开操作。然后修改后，再把迭代器指向新的List

类 CopyOnWriteArraySet<E>等同于ArrayList的同步Set版本（里面内容不能有重复）

## ThreadPoolExecutor

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

### 构造方法参数说明

* corePoolSize

核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。

* maximumPoolSize

线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的LinkedBlockingDeque时，这个值无效。

* keepAliveTime

非核心线程的闲置超时时间，超过这个时间就会被回收。

* unit

指定keepAliveTime的单位，如TimeUnit.SECONDS。当将allowCoreThreadTimeOut设置为true时对corePoolSize生效。

* workQueue

线程池中的任务队列.

常用的有三种队列，SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue。

* threadFactory

使用默认即可

线程工厂，提供创建新线程的功能。ThreadFactory是一个接口，只有一个方法

public interface ThreadFactory {

Thread newThread(Runnable r);

}

通过线程工厂可以对线程的一些属性进行定制。

* RejectedExecutionHandler

通常也是默认，新添加的线程会被抛异常，抛弃不处理，对正在处理的线程没有影响

RejectedExecutionHandler也是一个接口，只有一个方法

public interface RejectedExecutionHandler {

void rejectedExecution(Runnable var1, ThreadPoolExecutor var2);

}

当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution方法。

### 线程池规则

线程池的线程执行规则跟任务队列有很大的关系。

* 下面都假设任务队列没有大小限制：

其实源码LinkedBlockingDeque数字默认是Integer.MaxValue

* 1. 如果线程数量<=核心线程数量，那么直接启动一个核心线程来执行任务，不会放入队列中。
  2. 如果线程数量>核心线程数，并且无论是>还是<=最大线程数，只要任务队列是LinkedBlockingDeque的时候，超过核心线程数量的任务会全部放在任务队列中排队。因为源码LinkedBlockingDeque默认是Integer.MaxValue。会让所有的线程全部排队！除非排队数量超过Integer.MaxValue就抛异常！
  3. 如果线程数量>核心线程数，但<=最大线程数，并且任务队列是SynchronousQueue的时候，线程池会创建新线程执行任务，这些任务也不会被放在任务队列中。因为SynchronousQueue队列数量是1个。所以会立即启用非核心线程来处理过剩的线程！这些非核心线程，在任务完成后，闲置时间达到了超时时间就会被清除。如果线程数量>核心线程数，并且>最大线程数，当任务队列是SynchronousQueue的时候，会因为线程池拒绝添加任务而抛出异常。
* 任务队列大小有限时
  1. 当LinkedBlockingDeque塞满时，新增的任务会直接创建新线程来执行，当创建的线程数量超过最大线程数量时会抛异常。LinkedBlockingDeque（6），当同一时间有7个线程过来会抛异常
  2. SynchronousQueue没有数量限制。SynchronousQueue（默认就是1）因为他根本不保持这些任务，而是直接交给线程池去执行。当任务数量超过最大线程数时会直接抛异常。

#### 规则详解

##### 情况一：几乎不用

非核心线程一辈子不启用！永远不抛弃任何一个线程，不抛异常放弃

ThreadPoolExecutor(3,10,60L,TimeUnit.Second,new LinkedBlockingQueue<Runnable>(), ThreadFactory threadFactory, RejectedExecutionHandler handler)

先来3个线程。调用3个核心线程执行

再来1000个线程，全部存进LinkedBlockingQueue。由于还没有超出队列，所以不启用非核心线程。结果非核心线程7个无法使用

##### 情况二：

带队列（缓存）的线程池！会抛弃过量线程

ThreadPoolExecutor(3,10,60L,TimeUnit.Second,new LinkedBlockingQueue<Runnable>(20), ThreadFactory threadFactory, RejectedExecutionHandler handler)

假定：每个线程处理时间非常长！！！

先来3个线程。调用3个核心线程执行

再来20个线程，全部存进LinkedBlockingQueue。

再来7个的线程，启用非核心线程处理。

（当非核心线程启用后，处理完这7个线程，会提取LinkedBlockingQueue队列里面的线程，来处理。非核心线程，休息下来没有任务处理的时候，会启动超时时间，过了时间就回收关闭非核心线程！）

当LinkedBlockingQueue队列也满了，所有线程数量也满了，这个时候，再来线程，抛异常！

##### 情况三：

不带队列（缓存）的线程池！会抛弃过量线程

ThreadPoolExecutor(3,10,60L,TimeUnit.Second,new SynchronousQueue<Runnable>(), ThreadFactory threadFactory, RejectedExecutionHandler handler)

假定：每个线程处理时间非常长！！！

先来3个线程。调用3个核心线程执行I

再来7个的线程，立即启用非核心线程处理。

由于没有缓存队列！！！，所有此时能够处理线程的数量已经满了，这个时候，再来线程，抛异常！

#### 推荐心得

对于过剩线程的等待时间，处理方式来区分

* 任何线程都不能等待，要求立即处理！！处理不过来就抛弃，用情况三。注意要根据服务器性能设置好maximumPoolSize。
* 抛弃等待太久的线程，用情况二。注意要根据服务器性能设置好maximumPoolSize。和corePoolSize，LinkedBlockingQueue（队列大小）。3个值
* 线程随便等待多久都可以，但是不希望抛弃任何一个线程，用情况一。注意要根据服务器性能设置好corePoolSize

## 接口 BlockingQueue<E>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 抛出异常 | 特殊值 | 阻塞 | 超时 | 备注 |
| 插入 | [add(e)](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#add(E)) | [offer(e)](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#offer(E)) | [put(e)](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#put(E)) | [offer(e, time, unit)](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#offer(E, long, java.util.concurrent.TimeUnit)) | 当队列空间满的时候，发生不一样的情况 |
| 移除 | [remove()](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#remove(java.lang.Object)) | [poll()](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#poll(long, java.util.concurrent.TimeUnit)) | [take()](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#take()) | [poll(time, unit)](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/concurrent/BlockingQueue.html#poll(long, java.util.concurrent.TimeUnit)) | 当队列空间为null的时候，发生不一样的情况 |
| 检查 | [element()](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Queue.html#element()) | [peek()](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/util/Queue.html#peek()) | 不可用 | 不可用 | 当遍历队列发现为null的时候，发生不一样的情况 |

非常灵活的队列

类 ArrayBlockingQueue<E>//按照数组排列的可以设定队列（缓冲区）长度的阻塞队列

类 LinkedBlockingQueue<E>//按照链表排列的阻塞单向队列

类 LinkedBlockingDeque<E>//按照链表排列的阻塞双向队列

类 SynchronousQueue<E>//一种阻塞队列，队列长度1，放一个只有取走才能解除同步

BlockingQueue 和Semaphore。区别

阻塞队列是一种集合。集合里面存放满数据后，必须取走一个后，才能再放入另一个数据

Semaphore是控制线程运行，拿到信号灯才能执行线程。一个线程内部自己设置和释放信号

## 不可变类

**不可变类**：所谓的不可变类是指这个类的实例一旦创建完成后，就不能改变其成员变量值。如JDK内部自带的很多不可变类：Interger、Long和String等。  
**可变类**：相对于不可变类，可变类创建实例后可以改变其成员变量值，开发中创建的大部分类都属于可变类。

下面的理解可能会易懂一些：

{概念：不可变类的意思是创建该类的实例后，该实例的属性是不可改变的。[**Java**](http://lib.csdn.net/base/javase)中的8个包装类和String类都是不可变类。所以不可变类并不是指该类是被final修饰的，而是指该类的属性是被final修饰的。

自定义不可变类遵守如下原则：

1、使用private和final修饰符来修饰该类的属性。

2、提供带参数的构造器，用于根据传入的参数来初始化属性。

3、仅为该类属性提供getter方法，不要提供setter方法。

4、如果有必要，重写hashCode和equals方法，同时应保证两个用equals方法判断为相等的对象，其hashCode也应相等。}

### 不可变类的优点

说完可变类和不可变类的区别，我们需要进一步了解为什么要有不可变类？这样的特性对JAVA来说带来怎样的好处？

1. 线程安全  
   不可变对象是线程安全的，在线程之间可以相互共享，不需要利用特殊机制来保证同步问题，因为对象的值无法改变。可以降低并发错误的可能性，因为不需要用一些锁机制等保证内存一致性问题也减少了同步开销。
2. 易于构造、使用和测试
3. ...

### 不可变类的设计方法

对于设计不可变类，个人总结出以下原则：

**1. 类添加final修饰符，保证类不被继承**。  
如果类可以被继承会破坏类的不可变性机制，只要继承类覆盖父类的方法并且继承类可以改变成员变量值，那么一旦子类以父类的形式出现时，不能保证当前类是否可变。

**2. 保证所有成员变量必须私有，并且加上final修饰**  
通过这种方式保证成员变量不可改变。但只做到这一步还不够，因为如果是对象成员变量有可能再外部改变其值。所以第4点弥补这个不足。

**3. 不提供改变成员变量的方法，包括setter**  
避免通过其他接口改变成员变量的值，破坏不可变特性。

**4.通过构造器初始化所有成员，进行深拷贝(deep copy)**

如果构造器传入的对象直接赋值给成员变量，还是可以通过对传入对象的修改进而导致改变内部变量的值。例如：

public final class ImmutableDemo {

private final int[] myArray;

public ImmutableDemo(int[] array) {

this.myArray = array; // wrong

}

}

这种方式不能保证不可变性，myArray和array指向同一块内存地址，用户可以在ImmutableDemo之外通过修改array对象的值来改变myArray内部的值。  
为了保证内部的值不被修改，可以采用深度copy来创建一个新内存保存传入的值。正确做法：

public final class MyImmutableDemo {

private final int[] myArray;

public MyImmutableDemo(int[] array) {

this.myArray = array.clone();

}

}

**5. 在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝**  
这种做法也是防止对象外泄，防止通过getter获得内部可变成员对象后对成员变量直接操作，导致成员变量发生改变。

### String对象的不可变性

string对象在内存创建后就不可改变，不可变对象的创建一般满足以上5个原则，我们看看String代码是如何实现的。

public final class String

implements java.io.Serializable, Comparable<String>, CharSequence

{

/\*\* The value is used for character storage. \*/

private final char value[];

/\*\* The offset is the first index of the storage that is used. \*/

private final int offset;

/\*\* The count is the number of characters in the String. \*/

private final int count;

/\*\* Cache the hash code for the string \*/

private int hash; // Default to 0

....

public String(char value[]) {

this.value = Arrays.copyOf(value, value.length); // deep copy操作

}

...

public char[] toCharArray() {

// Cannot use Arrays.copyOf because of class initialization order issues

char result[] = new char[value.length];

System.arraycopy(value, 0, result, 0, value.length);

return result;

}

...

}

如上代码所示，可以观察到以下设计细节:

1. String类被final修饰，不可继承
2. string内部所有成员都设置为私有变量
3. 不存在value的setter
4. 并将value和offset设置为final。
5. 当传入可变数组value[]时，进行copy而不是直接将value[]复制给内部变量.
6. 获取value时不是直接返回对象引用，而是返回对象的copy.

这都符合上面总结的不变类型的特性，也保证了String类型是不可变的类。

### String对象的不可变性的优缺点

从上一节分析，String数据不可变类，那设置这样的特性有什么好处呢？我总结为以下几点：

**1.字符串常量池的需要**.  
字符串常量池可以将一些字符常量放在常量池中重复使用，避免每次都重新创建相同的对象、节省存储空间。但如果字符串是可变的，此时相同内容的String还指向常量池的同一个内存空间，当某个变量改变了该内存的值时，其他遍历的值也会发生改变。所以不符合常量池设计的初衷。

**2. 线程安全考虑**。  
同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

**3. 类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载**。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。

**4. 支持hash映射和缓存。**  
因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

缺点：

1. 如果有对String对象值改变的需求，那么会创建大量的String对象。

### String对象的是否真的不可变

虽然String对象将value设置为final,并且还通过各种机制保证其成员变量不可改变。但是还是可以通过反射机制的手段改变其值。例如：

//创建字符串"Hello World"， 并赋给引用s

String s = "Hello World";

System.out.println("s = " + s); //Hello World

//获取String类中的value字段

Field valueFieldOfString = String.class.getDeclaredField("value");

//改变value属性的访问权限

valueFieldOfString.setAccessible(true);

//获取s对象上的value属性的值

char[] value = (char[]) valueFieldOfString.get(s);

//改变value所引用的数组中的第5个字符

value[5] = '\_';

System.out.println("s = " + s); //Hello\_World

打印结果为：

s = Hello World

s = Hello\_World

发现String的值已经发生了改变。也就是说，通过反射是可以修改所谓的“不可变”对象的

### 总结

不可变类是实例创建后就不可以改变成员遍历的值。这种特性使得不可变类提供了线程安全的特性但同时也带来了对象创建的开销，每更改一个属性都是重新创建一个新的对象。JDK内部也提供了很多不可变类如Integer、Double、String等。String的不可变特性主要为了满足常量池、线程安全、类加载的需求。合理使用不可变类可以带来极大的好处。

### 自定义不可变类

//1.类被final修饰，不可继承

**public** **final** **class** ImmutableClass {

**private** **final** **int** id;//2.内部所有成员都设置为私有变量和final

**private** **final** String name;

**private** **final** Person p;

//所有变量赋值通过构造方法一次性传入后就不可变了

**public** ImmutableClass(**int** id,String name,Person p){

**this**.id = id;

**this**.name = name;

**this**.p = **new** Person(**new** Name());//4.当传入对象的时候，进行copy而不是直接将对象赋值给内部变量.

}

**public** **int** getId(){//3.不存在所有变量的的setter方法

**return** id;//基本数据类型自带深克隆，所以可以直接返回

}

**public** String getName(){

**return** name;

}

**public** Person getPerson(){

**return** **new** Person(p.getName());//5.获取内部对象时，不是直接返回对象引用，而是返回对象的copy

//不可变类提供了线程安全的特性但同时也带来了对象创建的开销，每更改一个属性都是重新创建一个新的对，每次获取类中的对象，都需要创建copy

}

}

## Volitae

1）Java 中能创建 volatile 数组吗？  
能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。我的意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。volatile 数组的效果和对象一样，只维护引用，不维护引用内部的值

2）volatile 能使得一个非原子操作变成原子操作吗？  
一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

3）volatile 修饰符的有过什么实践？  
一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

4）volatile 类型变量提供什么保证？([答案](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html))  
volatile 变量提供顺序和可见性保证，例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的，但 volatile 类型的 double 和 long 就是原子的。

# 网络编程

IP地址，确定电脑在网络上的地址

端口号的作用是区分软件用的。0-65535.一共65536个端口号

TCP和UDP协议各自有各自的端口号，都是0-65535

在同一个协议下，端口号不能重复

1024以下端口不要使用。给知名厂商预留的。比如80端口给http。21端口给ftp

URL统一资源定位符 URI统一资源。URL多了一个L。location。定位

## IP

数据传输步骤：

1.先选择协议（比如游戏，一般建立链接用tcp，具体不重要的数据（CS中的子弹）用udp）

TCP 可靠的链接 效率低

UDP 不可靠的链接 效率快

2.再把数据封装。对方收到数据后需要拆封装

类：

## InetAddress InetSocketAddress

InetAddress封装IP地址和DNS InetSocketAddress还封装了端口

InetAddress inetAddress = InetAddress.getLocalHost();//返回本机IP地址对象

inetAddress.getHostAddress()返回本机IP地址

inetAddress.getHostName()返回本机计算机名

## URL URI

URL统一资源标识符具体位置

URL包含协议，存放资源主机域名，端口号，资源文件名

http://dota2.uuu9.com:80/201704/542625.shtml

http协议。dota2.uuu9.com域名。端口80。201704/542625.shtml资源文件名。

URI统一资源标识符

[URL](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/net/URL.html#URL(java.lang.String))([String](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/String.html) spec) //绝对路径构建

URL url2 = new URL("http://dota2.uuu9.com/"); 在绝对路径最后要加/  
          根据 String 表示形式创建 URL 对象。

[URL](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/net/URL.html#URL(java.net.URL, java.lang.String))([URL](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/net/URL.html) context, [String](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/String.html) spec) //相对路径构建

URL url3 = new URL(url2, "201704/542625.shtml");在相对路径前面不要加/  
          通过在指定的上下文中对给定的 spec 进行解析创建 URL。

System.out.println("网址全称是：" + url3.toString());

System.out.println("协议是：" + url3.getProtocol());

System.out.println("域名是：" + url3.getHost());

System.out.println("端口是：" + url3.getPort());

System.out.println("资源是：" + url3.getFile());

System.out.println("相对路径是：" + url3.getPath());



锚点是网页内部写死的，网页内部跳转用的

System.out.println("锚点是：" + url3.getRef());//锚点aa?uname=bjsxt

System.out.println("参数是：" + url3.getQuery());//如果没有锚点返回? uname =bjsxt。

如果有锚点，就把参数当成是锚点的一部分，返回null

流

## openStream

## TCP核心思想：

TCP:ServerSocket用在服务端 Socket用在客户端

### Tcp服务端

1. 服务端需指定端口，客户端是系统随机创建端口

服务端是用ServerSocket。而客户端是用Socket。效率低，安全高。有明确的主次之分。服务端必须先开启，客户端才能发数据。如果服务端没有开启。则单独开启客户端会报错

/\*

\* 创建TCP服务端步骤

\* 1.建立服务端和指定端口

\* 2.创建监听

\* 3.发送数据或者接收数据

\*/

public class MyTCPServer {

public static void main(String[] args) throws IOException {

// 建立服务端和指定端口

ServerSocket server = new ServerSocket(8888);

// 创建监听

Socket serversocket = server.accept();//阻塞式的方法

System.out.println("一个客户端链接成功");

// 发送数据

String msg = "服务器发送给客户端-->欢迎访问";

OutputStream serveroutputStream = serversocket.getOutputStream();

BufferedWriter wr = new BufferedWriter(new OutputStreamWriter(serveroutputStream));

wr.write(msg);

wr.newLine();

wr.flush();

//wr.close();服务器不要关闭通道。因为服务器是不间断运行

}

}

### Tcp客户端

/\*

\* 创建TCP客户务端步骤

\* 1.建立客户端

\* Socket(String host, int port) 必须指定服务器主机和服务器端口号

\* 2.接收数据或者发送数据

\*/

public class MyTCPClient {

public static void main(String[] args) throws UnknownHostException, IOException {

//1.指定服务器主机和服务器端口号

Socket clientSocket = new Socket("192.168.1.104",8888);

//接收数据

InputStream inputStream = clientSocket.getInputStream();

BufferedReader br = new BufferedReader(new InputStreamReader(inputStream));

// char[] flush = new char[1024];

// int len = 0;

// while(-1!=(len = br.read(flush))){

// System.out.println(new String(flush, 0, len));

// }

String msg = br.readLine();

System.out.println(msg);

Socket clientSocket2 = new Socket("192.168.1.104",8888);

Socket clientSocket3 = new Socket("192.168.1.104",8888);

Socket clientSocket4 = new Socket("192.168.1.104",8888);

}

}

## UDP核心思想：

UDP：DatagramSocket DatagramPacket

1. 服务端和客户端两边都是DatagramSocket。
2. 通讯方式都是用DatagramPacket封装数据来传输。
3. Send发送数据的顺序。要和receive接收数据的顺序完全一致。才能解析到正确的数据！！是靠顺序来区分不同的数据！！
4. 所以任意一方发送数据后，不需要管对方到底有没有收到。效率高，安全低。没有明确的主次之分

### UDP客户端：

/\*

\* 创建客户端步骤

\* 1.创建客户端+端口

\* 2.准备数据

\* 3.封装包DatagramPacket(byte[] buf, int length, InetAddress address, int port)

构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。

\* 4.发送数据

\* 5.释放数据

\*/

public class MyClient {

public static void main(String[] args) throws IOException {

// 创建客户端+端口

DatagramSocket client = new DatagramSocket(6666);

// 准备数据

String str = "udp 编程";

byte[] data = str.getBytes();

double f = 89.12; //定义一个doubule类型

ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();//创建字符数组输出类，准备把double转换成byte[]数组

DataOutputStream dataout = new DataOutputStream(byteArrayOutputStream); //DataOutputStream是专门用于输出基本数据类型，里面放置byteArrayOutputStream对象

dataout.writeDouble(f); //把double写进去

dataout.flush();

byte[] byteArray = byteArrayOutputStream.toByteArray();//写进去后返回byte[]数组

byteArrayOutputStream.flush();//不用关

dataout.close();

byteArrayOutputStream.close();//关闭无效

// 核心：把数据封装成DatagramPacket！！！

DatagramPacket packet = new DatagramPacket(data, data.length,

new InetSocketAddress("127.0.0.1", 8888));

DatagramPacket packet2 = new DatagramPacket(byteArray, byteArray.length,

new InetSocketAddress("127.0.0.1", 8888));

// 核心：这里发送数据的顺序，是先发送packet，再发送packet2

client.send(packet);

client.send(packet2);

// 释放数据

client.close();

}

}

### UDP服务端

/\*

\* 创建服务端步骤

\* 1.创建服务端+端口

\* 2.准备接受容器

\* 3.封装包DatagramPacket(byte[] buf, int length)

\* 4.接收数据

\* 5.分析数据

\* 6.释放数据

\*/

public class MyServer {

public static void main(String[] args) throws IOException {

// 创建服务端+端口

DatagramSocket server = new DatagramSocket(8888);

// 准备接受容器

byte[] container = new byte[1024];

byte[] doublecon = new byte[1024];

// 封装包DatagramPacket(byte[] buf, int length) 用来接收长度为 length 的数据包。

DatagramPacket packet = new DatagramPacket(container, container.length);

DatagramPacket packet2 = new DatagramPacket(doublecon, doublecon.length);

// 核心：这里接收数据的顺序，要和发送数据的顺序，完全一致！！

//客户端先发送packet，那么就先接收packet

//客户端后发送packet2，那么就后接收packet2

server.receive(packet);//receive是阻塞，会一直保持监听接收状态

server.receive(packet2);

// 分析数据

byte[] data = packet.getData();

int len = packet.getLength();

System.out.println(new String(data, 0, len));

byte[] data2 = packet2.getData();

ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(data2);

DataInputStream dataInputStream = new DataInputStream(byteArrayInputStream);

double readDouble = dataInputStream.readDouble();

System.out.println(readDouble);

dataInputStream.close();

byteArrayInputStream.close();//关闭无效

// 释放数据

server.close();

}

}

端口区分软件，IP地址区分电脑

端口65536个 端口号是0-65535

不同的协议下，端口数可以重复。同一个协议下，端口数不能重复

TCP端口和UDP端口分开的。每一个都有65536个

端口号1024以下不要使用，比如：80端口是给Http。21端口是给ftp

Server端必须要指定端口，client系统随机指定端口

UDP链接

DatagramPacket

DatagramSocket

## 手写服务器httpserver

成对编程

<html>

<head>

<title>第一个Html</title>

</head>

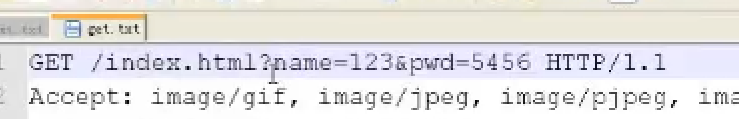
<body>

<pre>

### method:请求方式 get/post

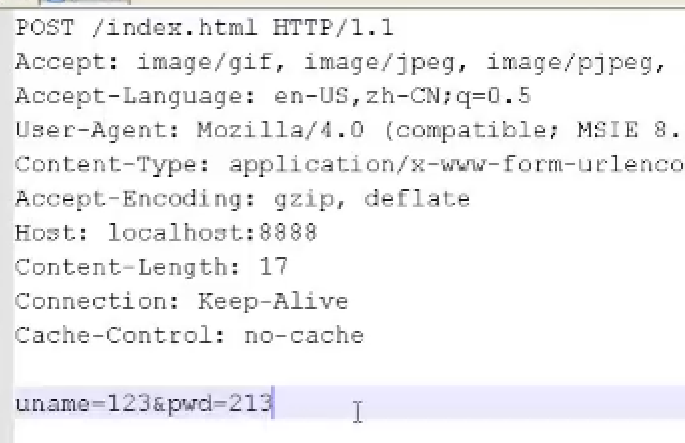
get：数据量小，安全性不高，默认方式

如果是get请求，用户名和密码在URL里面



post：量大，安全性高

如果是post请求，用户名和密码单独在最下面



action：请求服务器的路径

id：编号，前端区分唯一性，js中使用

name：名称，后端（服务器）区分唯一性，获取值

</pre>

<form method="get" action="http://localhost:8888/index.html">

用户名：<input type="test" name="uname" id="name"/>

密码：<input type="password" name="pwd" id="pass"/>

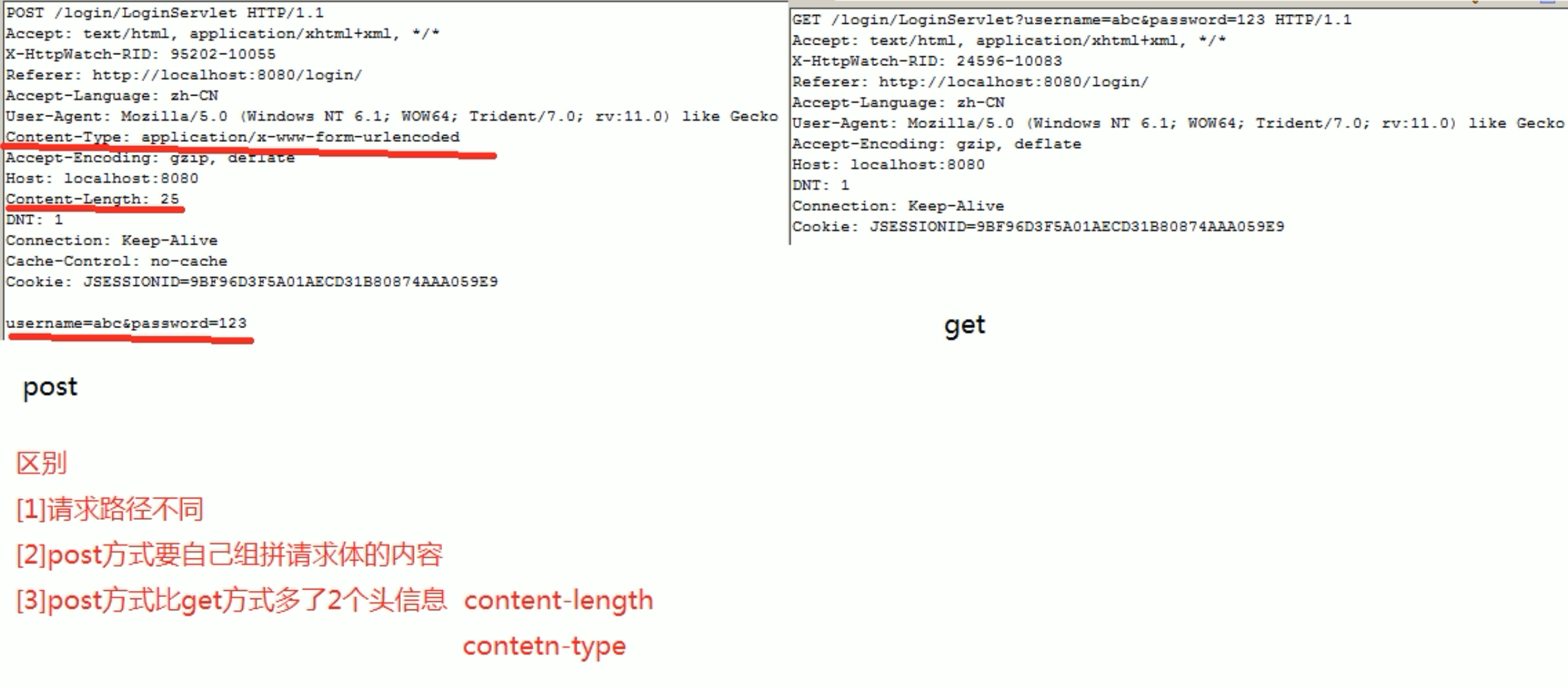
<input type="submit" value="登录"/>

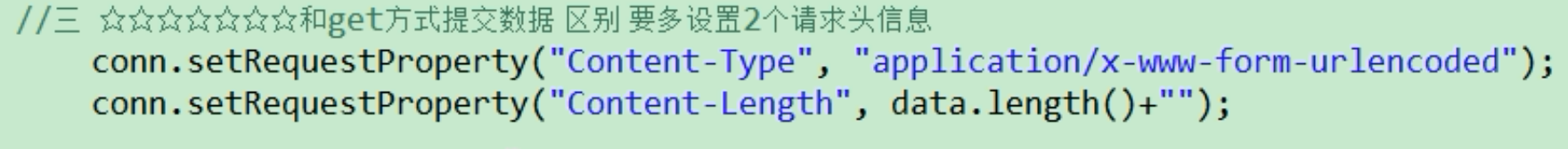
</form>

</body>

</html>

### Get和Post区别





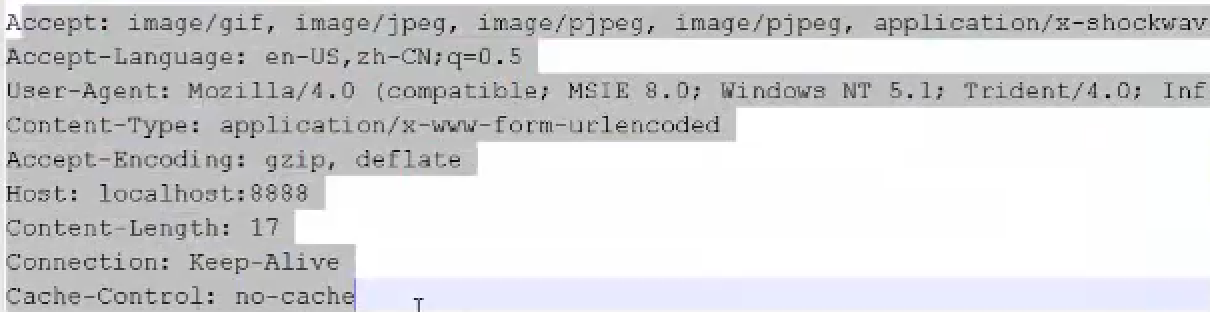
Data.length 是请求体的长度

### Http请求格式分为3部分

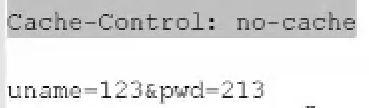
##### 请求方式（空格）URL（空格）http协议/版本

或者

##### 请求头包含了许多客户端环境和请求正文的有用信息



##### 请求正文(只有在post里面才有)

和请求头必须要有明显空行间隔

### Http响应格式分为3部分

响应与请求一样



#### Http协议版本，状态代码，描述



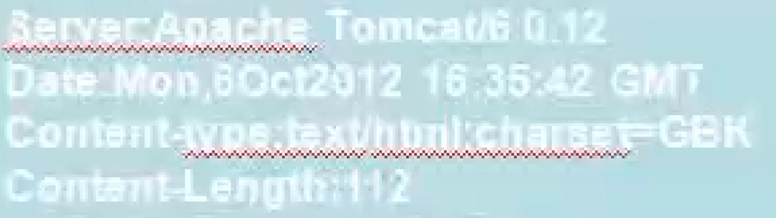
##### 常用状态代码：

成功类200：表示用户请求被正确接收

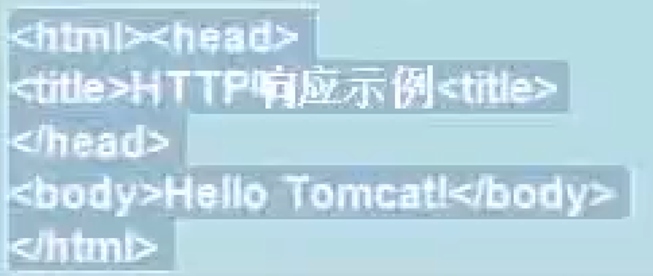
客户端错误404：请求中引用的文档不存在

服务端错误500：表示服务器不能完成对请求的处理

#### 响应头



#### 响应正文



## XML

基本解析xml的2种方式

SAX是基于事件流的解析，一行一行的逐步往下解析。适合XML内容多

DOM是基于XML文档树结构解析。DOM解析是把XML所有内容全部拷贝到内存，然后需要用哪块就提取哪块。适合XML内容比较少。

常见的4种解析方式

1. DOM生成和解析XML文档
2. SAX生成和解析XML文档
3. DOM4J生成和解析XML文档
4. JDOM生成和解析XML文档

Android解析xml是sax，dom，pull解析。用的最多的是pull解析

解析json，主要是原生json，GSON，Jackson。最多的是Gson

SAX解析步骤

1. 获得解析工厂 SAXParserFactory
2. 工厂获取解析器SAXParser
3. 加载文档Document注册处理器

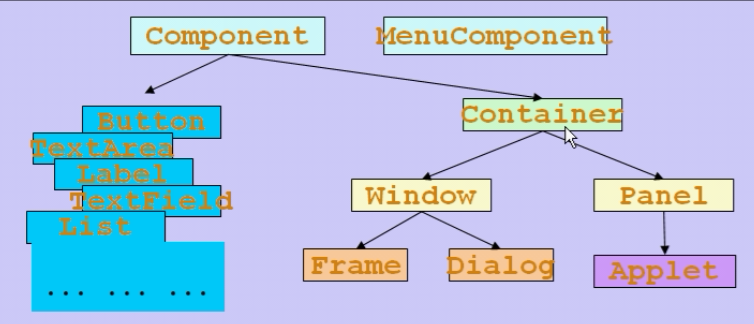
# GUI（了解）

AWT中的核心2个类：

Component是java用户界面的最基本组成部分

Container是可以容纳其他component的元素

Component对象不能独立显示，必须放在Container对象中才可以显示



Frame是Window的子类，独立的窗口显示

# 注解\_Annotation

注解是不仅是对相关语句的解释，还可以被相关程序读取

表示方式是：@注释名

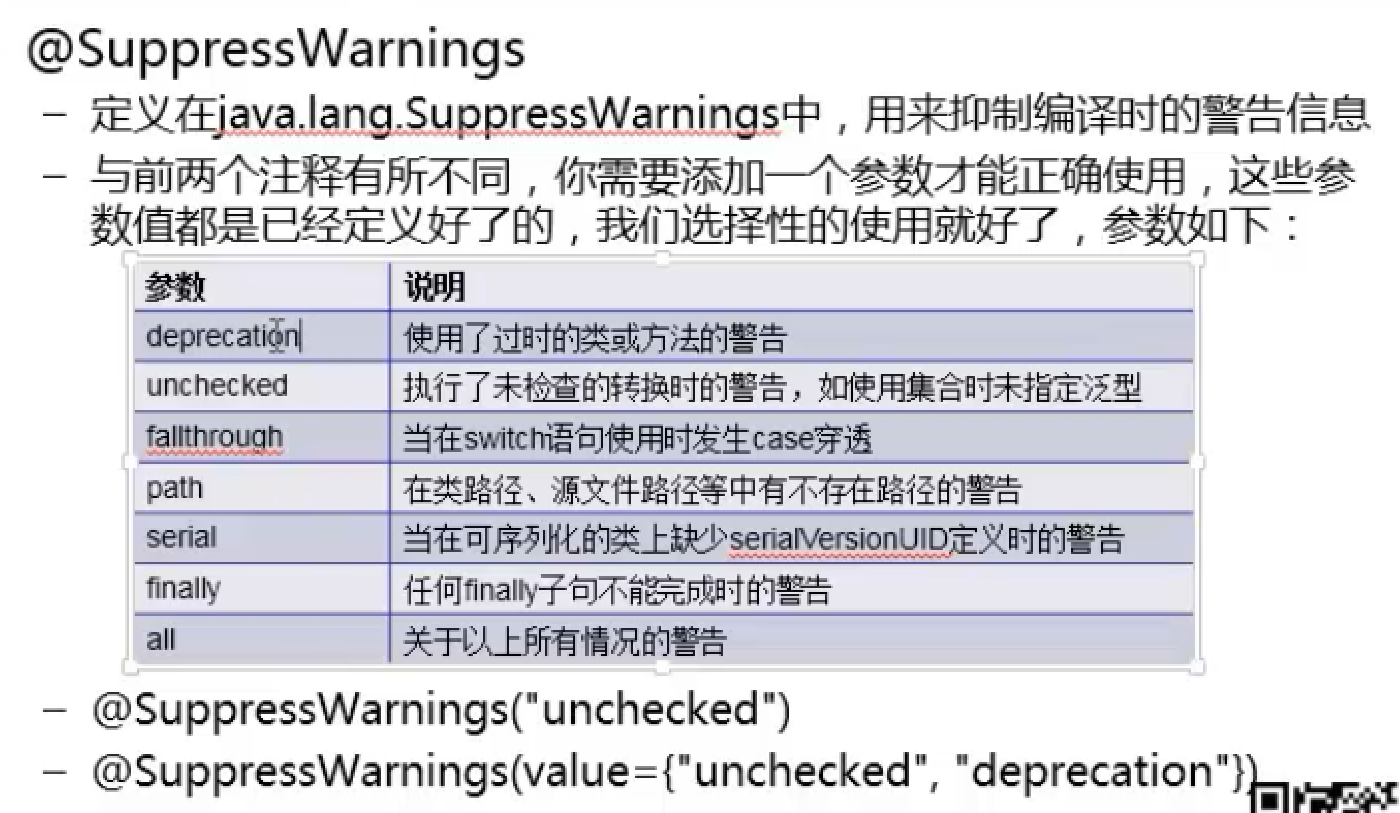
## 内置注解

加了内置注解后，类似泛型一样，系统会帮我们按照注解规范来检查！

@Override表示下面的方法是重写了父类的方法

@Deprecated表示不鼓励程序员使用这样的元素，通常原因是因为危险或者有更好的选择

@SuppressWarnings用来抑制编译时候的警告信息



## 自定义注解

### 自定义注解格式

@Target(ElementType.***METHOD***)//表示注解的使用范围，作用域！

@Retention(RetentionPolicy.***SOURCE***)//注解的生命周期。一般自定义的话为RetentionPolicy.RUNTIME，才可以被反射调用。否则和注释没有区别

**public** **@interface** Override {

}

@Target({***TYPE***, ***FIELD***, ***METHOD***, ***PARAMETER***, ***CONSTRUCTOR***, ***LOCAL\_VARIABLE***})

@Retention(RetentionPolicy.***SOURCE***)

**public** **@interface** SuppressWarnings {

String[] value();//Value是参数名。String[]是参数类型

}

### 自定义注解写法

自定义注解之前要加元注解

元注解的作用是负责注释其他的注解

@Target(value=ElementType.METHOD)

public @interface MyAnnotation {

String name() default "";//定义自定义注解，一般String类型默认””,int类型默认0，-1表示不存在的含义

String[] value() default {“aaa”,”bbb”};//定义数组自定义注解，如果内部只有一个值，一般定义为value

}

#### 注解含多个参数

##### 注解写法

@Target({***TYPE***,***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)//一般都要设置为运行时有效。可以通过反射调用

**public** **@interface** MyAnnotation {

String[] value() **default** {"yangke"};//第一个参数

**int** length();//第二个参数

}

##### 注解使用

@MyAnnotation(value={"class","clazz"}, length = 20)

**public** **class** AnnotationDemo01 {

@MyAnnotation(value={"abc","aaa"}, length = 30)

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

}

}

##### 注解获取多个参数

MyAnnotation aa = clazz.getAnnotation(MyAnnotation.**class**);

aa.length();//由于在MyAnnotation中定义了length参数，所以可以通过aa.length()直接找到对应的值

String[] value = aa.value();

**for** (String string : value) {

System.***out***.println(string+"--"+aa.length());

}

#### 注解defaule

String name() default “默认的内容” 如果在使用的时候，不使用。则用默认的内容

**public** **class** AnnotationDemo01 {

@MyAnnotation//使用注解的时候，使用默认的值

//@MyAnnotation(value={"abc","aaa"})。如果设置了值，结果就为自己设置的值

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

}

}

@Target({***TYPE***,***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** MyAnnotation {

String[] value() **default** {"yangke"};//设置默认值，不设置的时候，使用默认值

}

1. 选择要获取方法的注解
2. 拿到方法，才能获得该方法的注解

Class<?> clazz = Class.*forName*("com.test.Annotation.AnnotationDemo01");

Method[] declaredMethods = clazz.getDeclaredMethods();

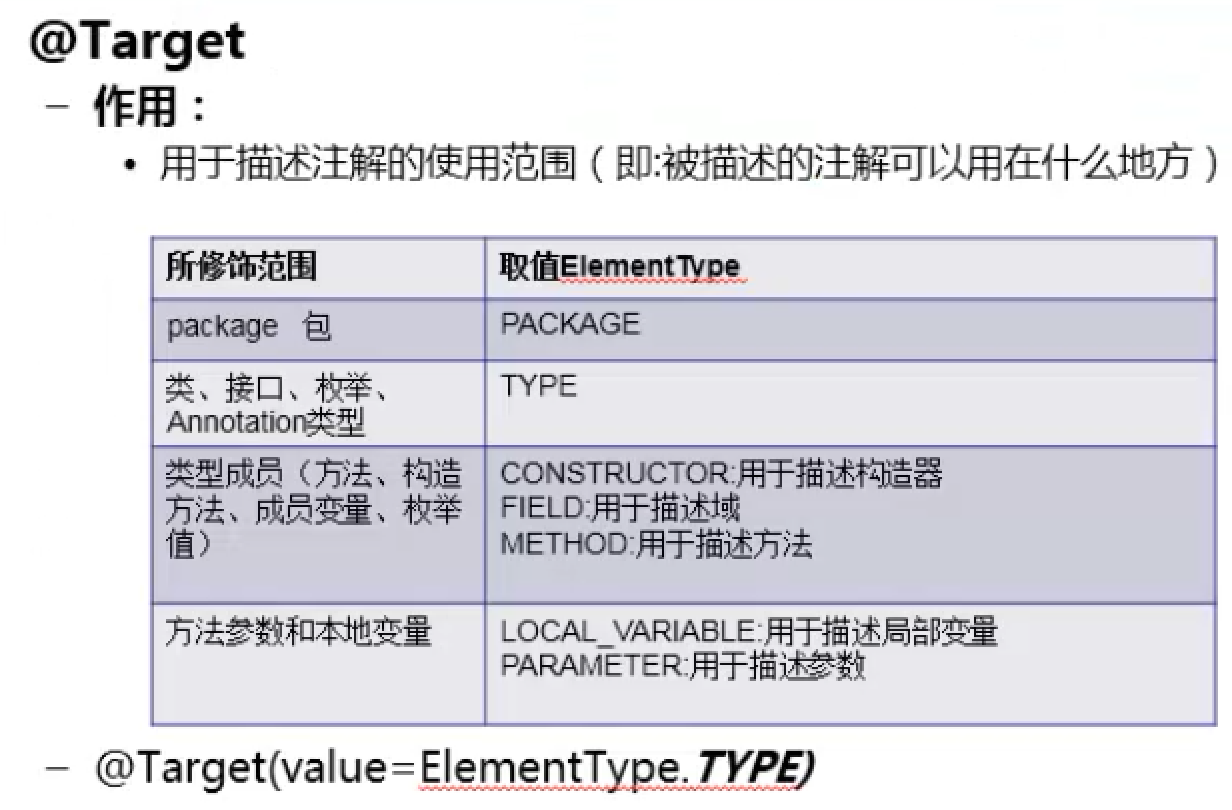
Annotation[] annotations = declaredMethods[0].getAnnotations();

System.***out***.println(annotations[0]);



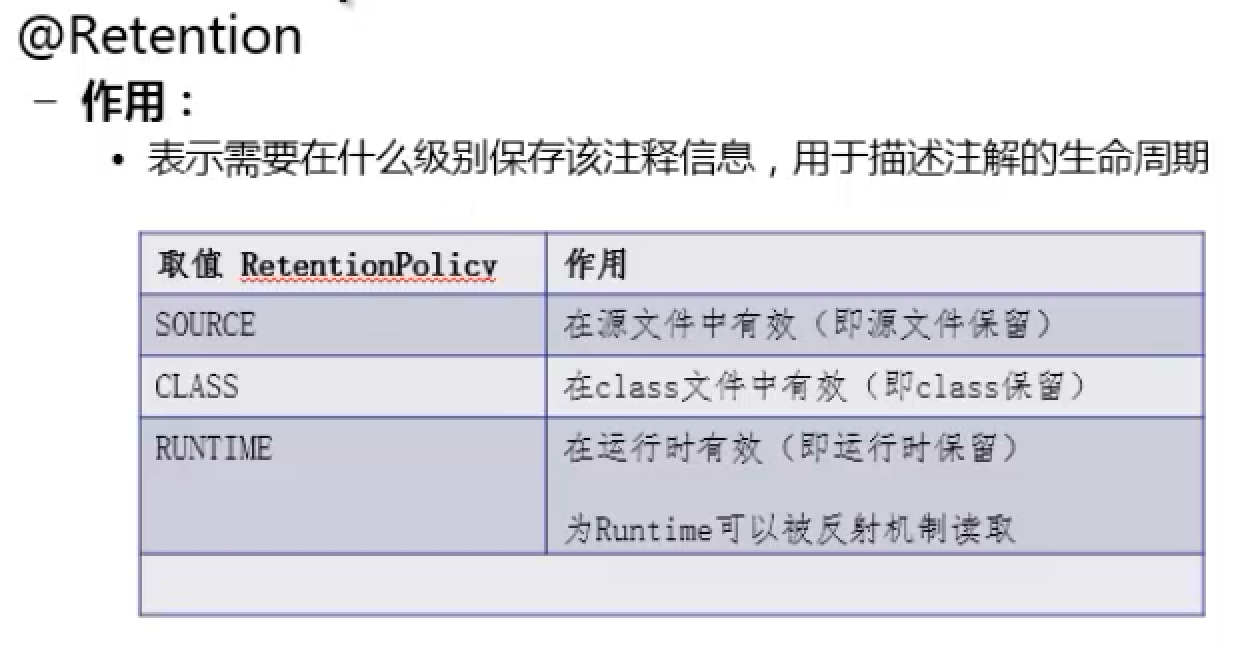
#### @Target

描述注解的使用范围



#### @Retention

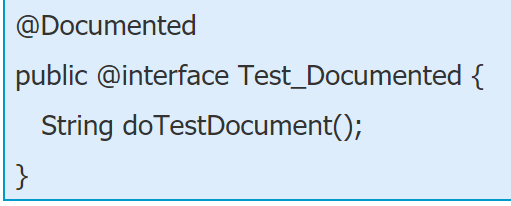
一般都要设置为RUNTIME。不然和一般注释没有什么区别



#### @Documented

Documented 注解表明这个注解应该被 javadoc工具记录



可以看见 在javadoc文档里面。由于注解类型是 Documented。所以会被javadoc文档保存

#### @Inherited

**通过对注解上使用元注解Inherited声明出的注解，在使用时用在类上，可以被子类所继承，对属性或方法无效**

## ORM

ORM是建立对象和数据库之间关系

类对应表

属性对应字段

一个对象对应一条记录

## 获得注解

1. 选择要获取哪个部分的注解
2. 拿到部分，才能获得该部分的注解

Class cla = Class.forName("com.bug.test.Annotation.Demo01");

//想获得类名上面的注解，就先获取类。

//调用类的字节码来获取类的注解cla.getAnnotations();

//获得该类的全部注解

Annotation[] annotation = cla.getAnnotations();

for (Annotation a : annotation) {

System.out.println(a);

}

//获得类的指定注解

//参数传入的是注解名字的class

MyAnnotation annotation2 = (MyAnnotation)cla.getAnnotation(MyAnnotation.class);

String[] value = annotation2.value();

System.out.println(value[0]);

//想获得类的属性上面的注解，就先获取属性。

//调用属性获取属性的注解field.getAnnotation(MyAnnotation.class);

//获得类的属性的注解

Field field = cla.getDeclaredField("name");

MyAnnotation annotation3 = field.getAnnotation(MyAnnotation.class);

System.out.println(annotation3.value()[0]);

### 自定义注解

@Target({***TYPE***,***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)//一般都要设置为运行时有效。可以通过反射调用

**public** **@interface** MyAnnotation {

String[] value() **default** {"yangke"};

}

### 使用注解类

@MyAnnotation(value={"class","clazz"})

**public** **class** AnnotationDemo01 {

@MyAnnotation(value={"abc","aaa"})

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

}

}

### 解析注解

//这里直接传入@MyAnnotation(value={"class","clazz"})中的MyAnnotation的class

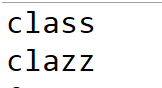
MyAnnotation aa = clazz.getAnnotation(MyAnnotation.**class**);

String[] value = aa.value();

**for** (String string : value) {

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

}



# JAVA动态性反射

## 反射

关键点：调用已经存在的class文件，并且操作属性，方法

获取类元码的方式：对象名.getClass() Class.forName(“类完整路径”) 类名.class

setAccessible//使用反射效率要比直接new对象操作低很多。可以通过不检查安全机制来提高性能

Declared和没有Declared的区别关键在于是否考虑继承问题

Declared，表示自身的所有（方法，属性，构造方法，注解）

没有Declared，表示自身和父类的所有public（方法，属性，构造方法，注解）

// 获取对象类名字

System.out.println(clazz.getName());// 获取对象完整路径的类名字

System.out.println(clazz.getSimpleName());// 获取对象的类名字

DeclaredFields() 仅仅是自身的所有属性（无论父类是什么属性，都无法访问！！）

Fields() 自身和父类的public属性（父类protected属性也无法访问！！）

方法也是一样，DeclaredMethods。仅仅是自身的所有方法（无论父类是什么方法，都无法访问！！）

getMethods 自身和父类的public方法（父类protected方法也无法访问！！）

#### 能不能访问父类的protected方法

**public** **class** Demo02 {

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

Person p = **new** Person();

Person instance = Person.**class**.newInstance();

// instance.jump2(1);

如果该Demo02和Person是在同一个包下，则instance.jump2(1)可以访问。可以访问Person的父类的protected方法！！

要看调用类，是否和父类在同一个包下

在，则可以调用

不再，则无法调用

但是无论是否在同一个包下，反射都无法访问父类的protected方法！！

### 反射数组

Person[] array = { **new** Person("yang1"), **new** Person("yang2"), **new** Person("yang3"), **new** Person("yang4") };

Class<?> componentType = array.getClass().getComponentType();//如果是数组，通过getComponentTyep，拿到数组里面对象的类型

Class<? **extends** Person[]> class1 = array.getClass();

System.***out***.println(componentType.getSimpleName());//这里返回Person

### 反射属性

// 获取对象类属性

Field[] f = clazz.getFields();// getFields()获取包含自己，父类，超父类的所有public属性

Field[] declaredFields = clazz.getDeclaredFields();// getDeclaredFields()只能获取自己的所有属性含private

Field f\_id = clazz.getDeclaredField("id");// 如果id为private，则只能用getDeclaredField

### 反射方法

// 获取对象类方法

Method[] methods = clazz.getMethods();// clazz.getMethods()//仅仅获取包含父类，超父类的所有public方法，如果重写了方法，methods也只能获取1个方法。不能获取protected方法

Method[] declaredMethods = clazz.getDeclaredMethods();// getDeclaredMethods()获取自己的所有方法

#### 方法参数是数组

**public** **void** runDis(Object[] d) {//参数需要传入的是数组

System.***out***.println("welcome to heima!"+"-->"+d[0]);

}

Method method = clazz.getDeclaredMethod("runDis", Object[].**class**);

Object o1 = **new** Object();

Object o2 = **new** Object();

Object[] o = { o1, o2 };

method.invoke(instance, **new** Object[]{o});//必须用Object包装一下

必须把数组用Object包装一下！！

method.invoke(instance, o);//这样写会报错！！！

### 反射内部类

// 获取对象类的内部类

Class<?>[] declaringClass = clazz.getDeclaredClasses();// getDeclaredClasses()获取自己的所有内部类

Class<?>[] classes = clazz.getClasses();// getClasses()获取包含父类，超父类的所有public内部类

### 反射构造方法

//获取对象类的构造方法

Constructor<?>[] constructors = clazz.getConstructors();//getConstructors()获取所有的public构造方法Constructor<?>[] constructors2 = clazz.getDeclaredConstructors();//getDeclaredConstructors()获取所有的构造方法

### 反射注解

clazz.getDeclaredAnnotations()//获取当前类的注解（仅仅是类上面的，类里面方法，属性的注解不获取）

clazz.getAnnotations()//获取当前类及父类的注解

### 以下是获取具体的构造方法，属性，方法

Class<?> clazz = Class.forName("com.bug.test.reflection.User");

User u1 = (User) clazz.newInstance();//相当于调用User的无参构造器

Constructor<User> c = (Constructor<User>) clazz.getDeclaredConstructor(int.class,String.class);//如果构造方法里面有参数，则在newInstance（里面传入参数值！）

c.setAccessible(true);//设置不判断访问控制权限

User uyang = c.newInstance(32,"yangke");//根据构造器反射创造对象。不能强转成User的子类！！

Field f = clazz.getDeclaredField("id");

f.setAccessible(true);

f.set(uyang, 500105);//根据成员变量名字通过反射给成员变量赋值

System.out.println(f.get(uyang));

Method m = clazz.getDeclaredMethod("setAge", int.class);//这里不能是Integer.class！

//方法对象.invoke(对象,该方法要传入的参数)。

//如果子类重写了方法，则这里的对象可以传子类，并且调用子类中重写的方法！！

//如果是m.invoke(uyang)等同执行了uyang对象的 m 方法。该方法没有参数

m.invoke(uyang, 35);//根据方法名字通过反射执行方法

**还可以获取泛型，详见**

|  |  |
| --- | --- |
| [Type](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/reflect/Type.html)[] | 方法.[getGenericParameterTypes](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/reflect/Method.html#getGenericParameterTypes())()  按照声明顺序返回 Type 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型的。 |
| [Type](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/reflect/Type.html) | 方法.[getGenericReturnType](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/reflect/Method.html#getGenericReturnType())() 返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象。 |

### 反射注意

1. 根据反射创造对象。和New出来对象效果一样
2. 通过反射调方法，传入（对象，参数），该对象即使为父类，也执行反射时的类的方法

Class<?> clazz = Class.*forName*("com.test.reflect.Person");

Object instance = **null**;

instance = (Person) clazz.newInstance();//instance是Object对象，所以不能通过instance直接访问Person类里面的方法

Method jumpMethod = clazz.getDeclaredMethod("jump", **int**.**class**);

jumpMethod.invoke(instance, 1);//传入instance是Object对象，但是运行结果却是执行了Person方法

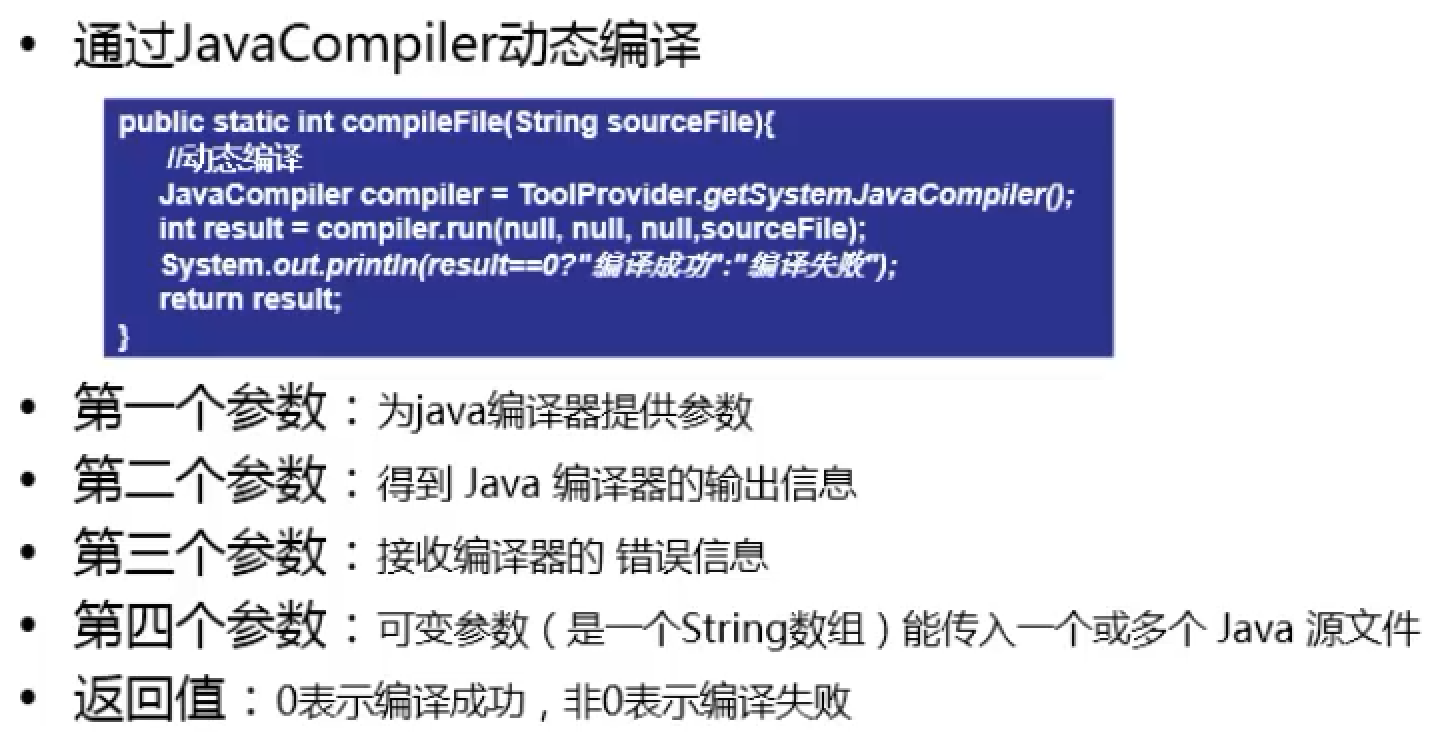
## 动态编译

关键点：通过string字符串动态创建类，属性，方法。

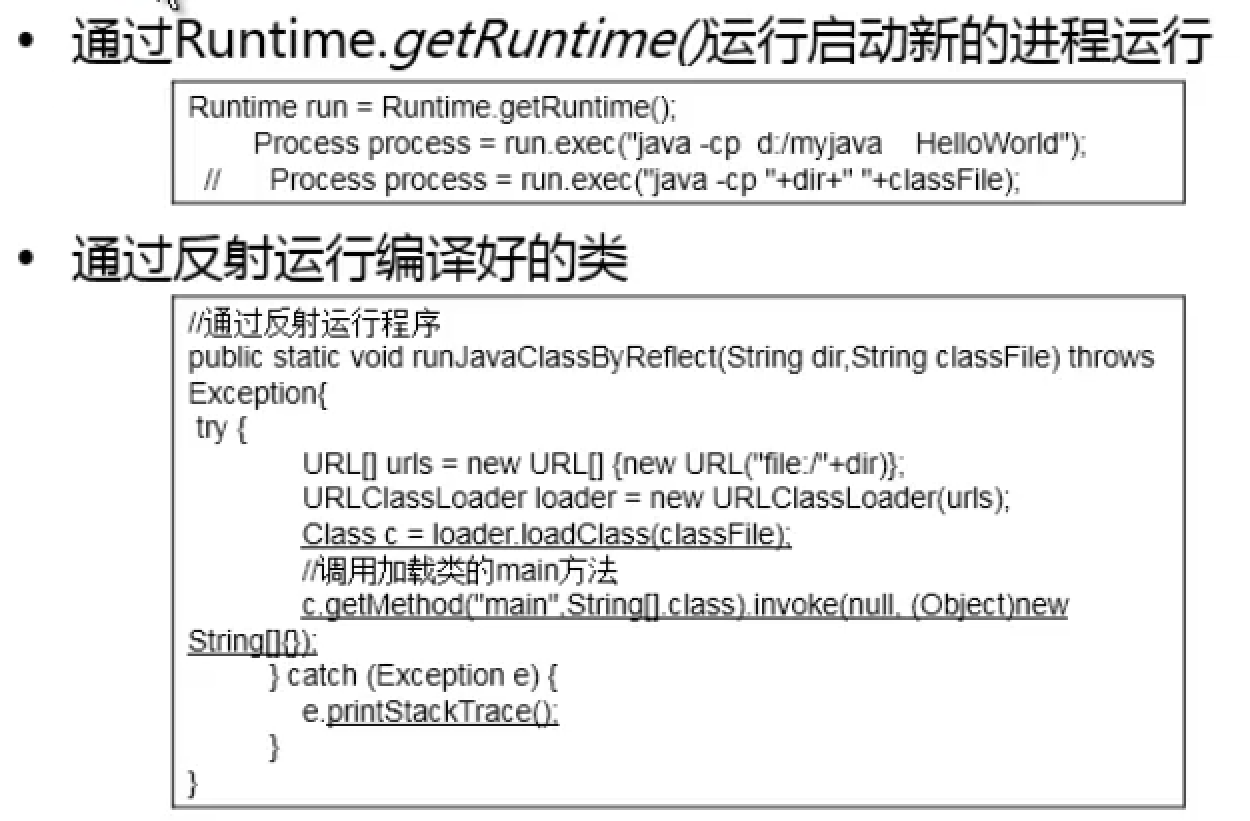
第一个参数为null：表示默认控制台输入

第二个参数为null：表示默认控制台输出

第三个参数为null：表示默认控制台输出错误信息

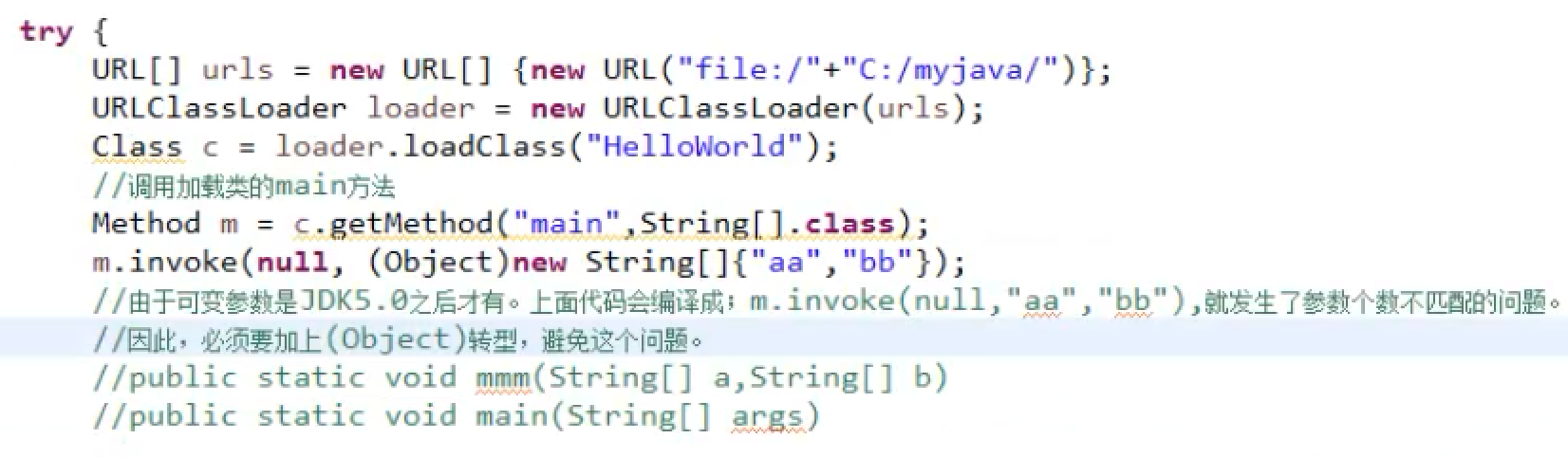


动态运行

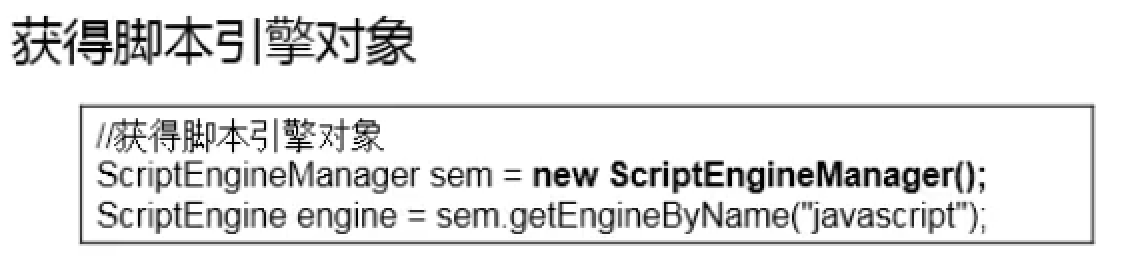


m.invoke(null,(Object)new String[]{});//因为main方法是静态方法，所以对象可以是null

关于必须强制转换成Object的问题如下图：



**获得脚本引擎**



# JVM核心

## JVM运行和类加载过程

类加载和初始化只有一次。

### 加载

通过类的加载器，把class字节码文件加载到内存中

将静态数据转换成方法区中的运行时数据结构

在堆中生成代表这个类的Class对象，作为方法区类数据访问的入口

### 链接

将Java类的二进制代码合并到JVM的运行状态之中的过程

#### 验证

确保加载类的信息，符合JVM规范，没有安全问题

#### 准备

为类的所有变量的（包含静态变量）分配内存，并设置类的所有变量为初始值

如:int a = 9; 这个阶段int a = 0；只赋初始值！

这些内存将在方法区中进行分配

#### 解析

虚拟机常量池内的符号引用替换为直接引用的过程

符号引用是指存在class文件中的常量池，包括类和接口的全限定名、字段的名称和描述符以及方法的名称和描述符。

符号引用可以个人理解为一些类，方法的加载规范，用符号标注。

实际的类名，方法名，需要通过解析，替换成直接引用。

直接引用就是偏移量，通过偏移量虚拟机可以直接在该类的内存区域中找到方法字节码的起始位置。

### 初始化\*\*

初始化过程是执行类构造器<clinit>()的过程（不是对象构造器）

类构造器方法是由编译器自动收集类中所有变量赋值，静态语句块中的合并产生，静态语句合并是编译顺序从上往下执行

初始化类，先初始化父类

虚拟机会保证一个类的构造器方法，在多线程环境中被正确的加锁和同步

当访问一个Java类的静态域时，只有真正声明这个域的类才会被加载

**类构造器是在堆的方法区里面创建**

**类加载就是静态成员和静态代码块的加载**

## 类的主动引用（一定会发生类的初始化）

1. **new一个类**
2. **调用类的静态成员（除了final成员）**
3. **调用类的静态方法**
4. **反射调用类** Class.forName("com.test.JVM.Demo01");
5. **如果该类是main所在的主类，则启动jvm时候，一定会先加载主类**
6. **当子类被调用的时候，先加载父类**

## 类的被动引用（一定不会发生类的初始化）

1. **当通过子类来调用父类的静态成员或方法，且子类没有重写父类的静态成员或方法的时候，不会初始化子类，只初始化父类**
2. **定义类的数组(Dog[] d = new Dog[10];)**
3. **调用final常量，在编译阶段就把常量存入常量池**

## 类加载器

**代理模式**

**双亲委托机制也是代理模式的一种。不是所有的类加载模式都是执行双亲委托机制。比如tomcat的代理模式就必须要求自己来实现类加载器**

loadClass(String name)**默认双亲委托机制，可以重写方法修改机制**

****

**启动类加载，负责加载java核心库**

**扩展类加载，负责加载第三方jar包库**

**应用程序类加载，负责加载eclipse程序classpath下的类**

**自定义类加载器，负责满足特殊需求**

### 类加载顺序

**类加载的时候，执行顺序是，先从启动类加载器（祖辈），扩展类加载（爷辈），应用程序类加载（父辈），自定义加载（子辈），从上往下查询类（好处是防止篡改核心类），上面能加载，下面就不加载。**

### 注意

**必须要完全相同的类加载器对象（new的对象一样），去加载同一个类，类才能加载一次**

**不同的自定义类加载，加载同一个类，JVM会多次加载。**

**同一个自定义类加载器对象new多次，然后去加载同一个类，JVM也会多次加载**

## 自定义类加载器

**加密解密**

**关键点在于把文件转换成字节数组的时候，进行加密解密**

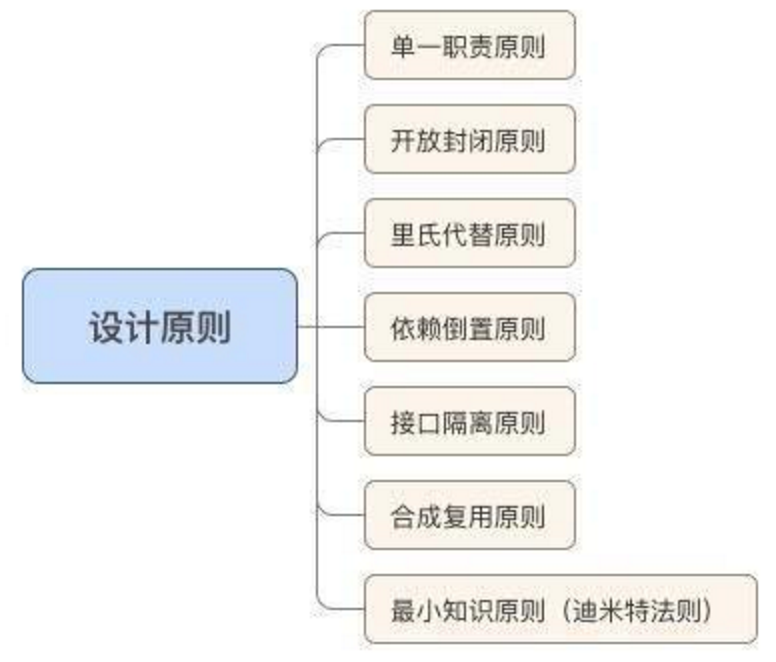
**线程上下文加载器**

//比如当前线程默认为应用加载器，可以修改成自定义加载器

Thread对象.[setContextClassLoader](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/Thread.html#setContextClassLoader(java.lang.ClassLoader))([ClassLoader](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/ClassLoader.html) cl) 设置该线程的上下文 ClassLoader。

Thread对象.[getContextClassLoader](mk:@MSITStore:C:\Users\馒头大人\Desktop\学习\JDK_API_1_6_zh_CN1.CHM::/java/lang/Thread.html#getContextClassLoader())() 返回该线程的上下文 ClassLoader。

# 设计原则



## 单一职责原则

一个类=只有一个引起它变化的原因。

如果一个类承担的职责过多，即耦合性太高=一个职责的变化可能会影响到其他的职责

## 开放封闭原则

一个实体（类、函数、模块等）应该对外扩展开放，对内修改关闭

1. 即每次发生变化时，要通过添加新的代码来增强现有类型的行为，而不是修改原有的代码。
2. 符合开放封闭原则的最好方式是提供一个固有的接口，然后让所有可能发生变化的类实现该接口，让固定的接口与相关对象进行交互。

## 里氏代替原则

子类必须替换掉它们的父类型。

1. 在软件开发过程中，子类替换父类后，程序的行为是一样的。
2. 只有当子类替换掉父类后软件的功能不受影响时，父类才能真正地被复用，而子类也可以在父类的基础上添加新的行为。

所有引用基类的地方必须能透明地使用其子类对象。本质上就是说要好好利用继承和多态，从而以父类的形式来声明变量（或形参），为变量（或形参）赋值任何继承于这个父类的子类。

## 依赖倒置原则

细节应该依赖于抽象，而抽象不应该依赖于细节。

所谓的的 “面向接口编程，而不是面向实现编程”。这样可以降低客户与具体实现的耦合。

在我们用的Java语言中，抽象就是指接口或者抽象类，二者都是不能直接被实例化；细节就是实现类，实现接口或者继承抽象类而产生的类，就是细节。使用Java语言描述就是：各个模块之间相互传递的参数声明为抽象类型，而不是声明为具体的实现类；

## 接口隔离原则

使用多个专门功能的接口，而不是使用单一的总接口。

不要让一个单一的接口承担过多的职责，而应把每个职责分离到多个专门的接口中，进行接口分离。

## 合成复用原则

在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分。

新对象通过向这些对象的委派达到复用已用功能的目的。简单地说，就是要尽量使用合成/聚合，尽量不要使用继承。

## 最少知识原则（迪米特法则）

一个模块或对象应尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立，这样当一个模块修改时，影响的模块就会越少，扩展起来更加容易。

1. 关于迪米特法则的其他描述：只与你直接的朋友们通信；不要跟“陌生人”说话。
2. 外观模式（Facade Pattern)和中介者模式（Mediator Pattern）就使用了迪米特法则。

假设类A实现了某个功能，类B需要调用类A的去执行这个功能，那么类A应该只暴露一个函数给类B，这个函数表示是实现这个功能的函数，而不是让类A把实现这个功能的所有细分的函数暴露给B。

# 设计模式

23种设计模式

## 创建型模式：关注对象的创建过程

封装了具体类的信息

隐藏了类的实例化过程

### 单例模式

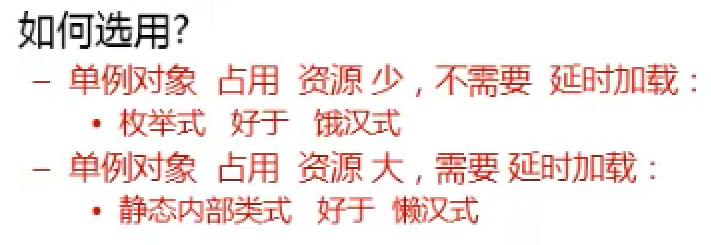
#### 适用性

* 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
* 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

单例模式：保证一个类只有一个实例，减少系统开销。通过启动应用时产生单例对象，永驻内存来解决开销问题

单例应用场景：比如：负责所有数据表的映射处理

实现单例模式5种方式



#### 饿汉式（线程安全，调用效率高，不能延时加载）

/\*

\* 单例模式：饿汉式

\*/

public class SingletonDemo01 {

//第二步，在类加载的时候，立即创建实例对象，线程安全不延时

private static SingletonDemo01 instance = new SingletonDemo01();

//第一步，私有化构造器

private SingletonDemo01(){

}

//第三步，创建公开静态方法来获取唯一实例

public static SingletonDemo01 getInstance(){

return instance;

}

}

#### 懒汉式（线程安全，调用效率不高，延时加载）

/\*

\* 单例模式：懒汉式

\*/

public class SingletonDemo02 {

// 第二步，在类加载的时候，声明静态变量，但是不加载

private static SingletonDemo02 instance;

// 第一步，私有化构造器

private SingletonDemo02() {

}

// 第三步，创建公开静态方法，在调用静态方法的时候，才创建唯一实例

//当有多线程进来的时候，确保多线程访问的时候，只创建唯一实例

public static synchronized SingletonDemo02 getInstance() {

if (instance == null) {

instance = new SingletonDemo02();

}

return instance;

}

}

其他：

双重检测锁式（由于JVM底层内部模型原因，偶尔出问题，不建议使用, 但是可以用volatile完美解决BUG!!!）

使用了**双重检查成例**的“懒汉式”单例类，不能工作的基本原因在于，在Java编译器中，**LazySingleton类的初始化与m\_instance变量赋值的顺序不可预料**。如果一个线程在没有同步化的条件下读取m\_instance引用，并调用这个对象的方法的话，可能会发现对象的初始化过程尚未完成，从而造成崩溃。

public class SingletonDemo02 {

// 第二步，在类加载的时候，声明静态变量，用volatile完美解决BUG!!!确保每次返回最新的值

private static volatile SingletonDemo02 instance;

// 第一步，私有化构造器

private SingletonDemo02() {

}

// 第三步，创建公开静态方法，在调用静态方法的时候，才创建唯一实例

//当有多线程进来的时候，确保多线程访问的时候，只创建唯一实例

public static SingletonDemo02 getInstance() {

if (instance == null) {

synchronized(SingletonDemo02.class){

if (instance == null) {

instance = new SingletonDemo02();

}

}

}

return instance;

}

}

#### 静态内部类式（线程安全，调用效率高，可以延时加载）

/\*

\* 单例模式：静态内部类模式，也是一种懒汉式，兼备了高并发安全和延迟加载效率

\*/

public class SingletonDemo03 {

// 第二步，创建静态内部类，在外部类加载时候，内部类不加载，起到延迟加载效果！

private static class SingletonInnerClass {

private static final SingletonDemo03 instance = new SingletonDemo03();

}

// 第一步，私有化构造器

private SingletonDemo03() {

}

// 第三步，创建公开静态方法，在调用静态方法的时候，才加载内部类，来创建唯一实例

public static SingletonDemo03 getInstance() {

return SingletonInnerClass.instance;

}

}

#### 枚举单例（天然线程安全，天然抵制反序列化和反射漏洞，调用效率高，不能延时加载）

\* 单例模式：枚举单例，立即加载

\*/

public enum SingletonDemo04 {

//这个枚举元素本身就算单例，天然线程安全

INSTANCE;

//根据情况添加相关操作

public void SingletonDemo04Operation() {

}

}

class Resource{

//类Resource是我们要应用单例模式的资源，具体可以表现为网络连接，数据库连接，线程池等等。

}

#### 枚举获取单例

//获取资源的方式很简单，只要 SomeThing.INSTANCE.getInstance() 即可获得所要实例

public enum SomeThing {

INSTANCE;

private Resource instance;

SomeThing() {

instance = new Resource();

}

public Resource getInstance() {

return instance;

}

}

测试各种单例类型的执行效率



long start = System.currentTimeMillis();

int ThreadCount = 10;

CountDownLatch c = new CountDownLatch(ThreadCount);

for (int i = 0; i < ThreadCount; i++) {

new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000000; i++) {

Object o = SingletonDemo01.getInstance();//饿汉式效率高：16ms/1000万次

Object o = SingletonDemo02.getInstance();//懒汉式效率低：962ms/1000万次

Object o = SingletonDemo03.getInstance();//静态内部类效率高：53ms/1000万次

Object o = SingletonDemo04.INSTANCE;//枚举单例效率高：16ms/1000万次

}

c.countDown();

}

}).start();

}

c.await();

long end = System.currentTimeMillis();

System.out.println("执行效率时间：" + (end - start));

#### 反射或反序列化漏洞

通过反射或反序列化漏洞，创建多个不同的实例对象（枚举单例除外）

//先创建一个实例对象

SingletonDemo05 d1 = SingletonDemo05.getInstance();

//在先创建一个实例的前提下，再通过反射创建多个不同的对象

Class<SingletonDemo05> clzz = (Class<SingletonDemo05>)Class.forName("com.test.GOF.SingletonDemo05");

Constructor<SingletonDemo05> c = clzz.getDeclaredConstructor(null);

c.setAccessible(true);

//s1,s2和d1不是同一个对象

SingletonDemo05 s1 = c.newInstance(null);

SingletonDemo05 s2 = c.newInstance(null);

//通过反序列化，来创建多个不同的对象

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("c:/a.txt"));

oos.writeObject(d1);

oos.flush();

oos.close();

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("c:/a.txt"));

//o1和d1不是同一个对象

SingletonDemo05 o1 = (SingletonDemo05)ois.readObject();

ois.close();

##### 解决方案：

// 第一步，私有化构造器

private SingletonDemo05() {

//保证在先调用getInstance()创建1个实例后，无法通过反射创建多个不同的对象

if (instance!=null) {

//确保有了实例对象后，无法用反射创建对象

PS：//但是如果之前未调用SingletonDemo05.getInstance()，而直接调用反射还是可以创建多个对象

throw new RuntimeException();

}

}

//在类里面加入方法readResolve()，来防止反序列化创建不同的对象

// readResolve()的作用就是当反序列化读取类的对象时，直接返回instance

private Object readResolve(){

return instance;

}

### 工厂模式

实现了创建者和调用者的分离

#### 适用性

* 当一个类不知道它所必须创建的对象的类的时候。
* 当一个类希望由它的子类来指定它所创建的对象的时候。
* 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

#### 简单工厂模式

客户端不需要知道具体奥迪车如何造出来，只需要调用工厂创造即可

对于新增加产品则必须要修改工厂源代码，违反开闭原则！

例一：//Car c = FactorySimple.createCar("奥迪");直接调取静态方法创建对象

public class FactorySimple {

static Car createCar(String Type) {//核心1.返回接口类型

if (Type.equals("奥迪")) {

return new AudiCar();//核心2.根据不同的type，来返回不同实现接口的内容

} else if (Type.equals("长安")) {

return new ChangAnCar();

}

return null;

}

}

例二：// Car c = FactorySimple.createAudiCar();

public class FactorySimple {

static AudiCar createAudiCar(){

return new AudiCar();

}

static ChangAnCar createChangAnCar(){

return new ChangAnCar();

}

}

#### 工厂方法模式

每新增加一个产品则必须新增一个相应的工厂源代码，完全符合开闭原则！但是相应类的数量大大增加。有1万种产品，就有1万个相应产品的类工厂。客户端需要知道1万个类的名字，管理维护难度大大增加，而简单工厂只需修改1个工厂总类即可

//Car c = new BenZFactoryMethod().createCar();需要创建实例对象，非静态

public class BenZFactoryMethod implements FactoryMethodCar{

public Car createCar() {

return new BenZ();

}

}

public class AudiCarFactoryMethod implements FactoryMethodCar{

public Car createCar() {

return new AudiCar();

}

}

public class ChangAnCarFactoryMethod implements FactoryMethodCar{

public Car createCar() {

return new ChangAnCar();

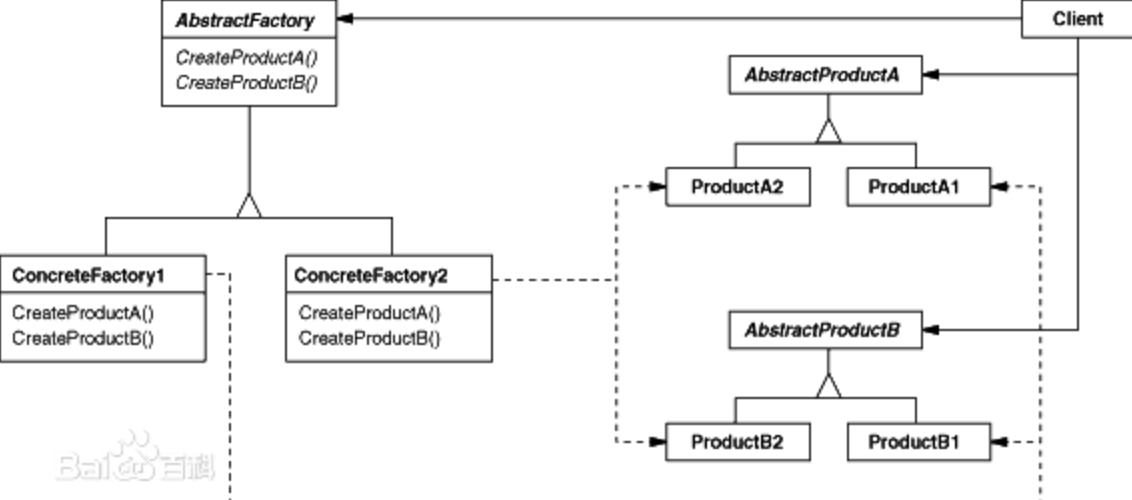
}

}

### 抽象工厂模式

#### 适用性

* 一个系统要独立于它的产品的创建、组合和表示时。
* 一个系统要由多个产品系列中的一个来配置时。
* 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
* 当你提供一个产品类库，而只想显示它们的接口而不是实现时。



解决生产不同产品族的所有产品问题

关键点：创建每一个产品都增加接口，实现不同产品灵活组合

//创建组合产品的接口。这样可以灵活创建具体实现类

**public** **interface** ISeat {

**void** comfort();

}

**public** **interface** IEngine {

**void** run();

}

**public** **interface** ITyre {

**void** wearproof();

}

//灵活创建具体的发动机实现类，以便组合成不同的车

**public** **class** MyLowEngine **implements** IEngine{

@Override

**public** **void** run() {

System.***out***.println("低端发动机跑得慢");

}

}

**public** **class** MyLuxuryEngine **implements** IEngine{

@Override

**public** **void** run() {

System.***out***.println("高端发动机跑得快");

}

}

//创建汽车工厂接口，这样可以灵活组合成不同产品具体的车

**public** **interface** ICarFactory {

IEngine creatEngine();

ISeat creatSeat();

ITyre creatTyre();//核心1.定义组合不同类型的组合接口

}

//灵活创建具体的汽车工厂实现类，以便组合成不同产品的车

**public** **class** MyLowCarFactory **implements** ICarFactory{//核心2.定义不同组合的具体实现

//可以查看在低端汽车工厂类下，所有包含的零件。

@Override//灵活选择发动机

**public** IEngine creatEngine() {

**return** **new** MyLowEngine();

}

@Override//灵活选择座椅

**public** ISeat creatSeat() {

**return** **new** MyLowSeat();

}

@Override//灵活选择轮胎

**public** ITyre creatTyre() {

**return** **new** MyLowTyre();

}

}

**public** **class** Client {

//调用情况

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

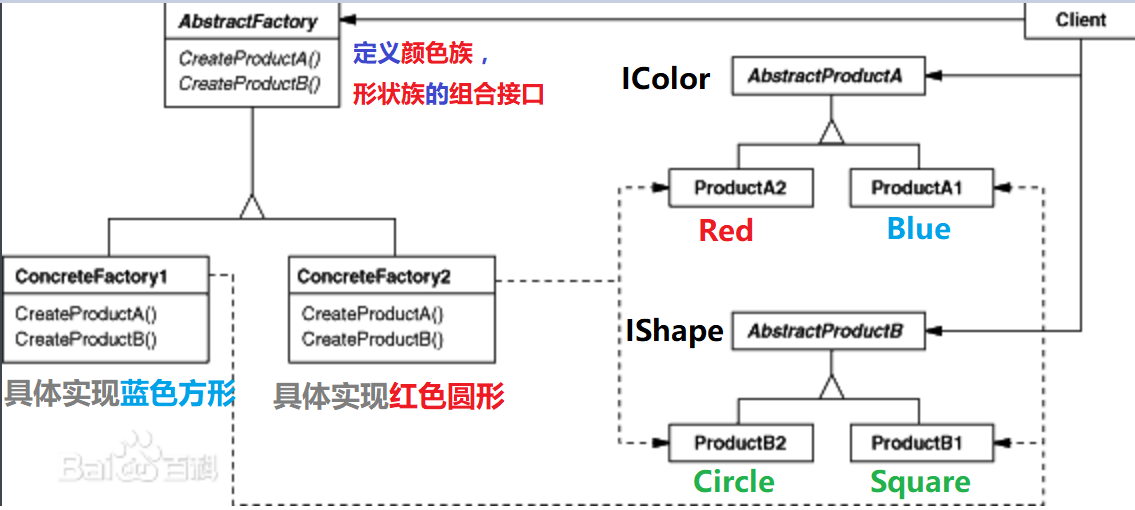
ICarFactory lowCar = **new** MyLowCarFactory();

lowCar.creatEngine().run();

}

}

#### 抽象工厂解释图



#### 抽象工厂，简单工厂，工厂方法的区别

简单工厂：工厂对象.静态方法来创建对象

对于新增加产品则必须要修改工厂源代码，违反开闭原则！但是简单粗暴！

工厂方法：先new工厂对象，通过调用对象.方法来创建对象

每新增加一个产品则必须新增一个相应的工厂源代码，完全符合开闭原则！但是相应类的数量大大增加。有1万种产品，就有1万个相应产品的类工厂。客户端需要知道1万个类的名字，管理维护难度大大增加，而简单工厂只需修改1个工厂总类即可

核心总结：一个工厂类，里面多个新建对象方法

抽象工厂：产品全部接口化，接口实现不同的具体类型来灵活组合产品

由于工厂接口已经写死，所以对于该接口下的具体类，无法新增产品族。只能增加产品。其实还可以通过增加新的工厂接口继承原接口，来扩展新的产品。抽象工厂把组合定死了。要想把构造和装配分离，就用建造者模式

核心总结：多个工厂类，里面是相同的一个新建对象方法

### 建造者模式

#### 适用性

* 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
* 当构造过程必须允许被构造的对象有不同的表示时。

实现了构造和装配的分离

关键点：构造负责builder建造，装配director负责组装

//构建者接口

**public** **interface** IAirShipBuilder {

OrbitalModule buildOrbitalModule();

Engine buildEngine();

EscapeTower buildEscapeTower();

//特别注意：在构造者接口里面直接定义飞船整体模型

AirShip buildAirShip();

}

构造者负责构建

**public** **class** MyAirShipBuilder **implements** IAirShipBuilder {

//构建着内部定义完整飞船模型

**private** AirShip myairship;

//创建获得飞船模型方法

**public** AirShip buildAirShip() {

**return** myairship;

}

//在建造者的构造方法中，创建飞船模型

**public** MyAirShipBuilder() {

**super**();

myairship = **new** AirShip();

}

@Override//创建轨道舱

**public** OrbitalModule buildOrbitalModule() {

**return** **new** OrbitalModule("MyOrbitalModule");

}

@Override//创建发动机

**public** Engine buildEngine() {

**return** **new** Engine("MyEngine");

}

@Override//创建逃逸塔

**public** EscapeTower buildEscapeTower() {

**return** **new** EscapeTower("MyEscapeTower");

}

}

设计师负责装配

//核心。建造者模式，把组装对象的过程分离了出来，由组装类来完成组装！！

**public** **class** MyAirShipDirector **implements** IAirShipDirector {

**private** IAirShipBuilder airShipBuilder;

**public** MyAirShipDirector(IAirShipBuilder airShipBuilder) {

**this**.airShipBuilder = airShipBuilder;

}

@Override

**public** AirShip directAirShip() {

//直接组合装配各个组件

airShipBuilder.buildOrbitalModule();

airShipBuilder.buildEngine();

airShipBuilder.buildEscapeTower();

//按照指定顺序装配设计后，直接返回该对象

**return** airShipBuilder.buildAirShip();

}

}

//调用情况

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

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

//new一个设计师来（装配）指挥一个建造者来（建造）

AirShip MyShip = **new** MyAirShipDirector(**new** MyAirShipBuilder()).directAirShip();

MyShip.launch();

}

}

### 原型模式prototype

#### 适用性

* 当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者
* 为了避免创建一个与产品类层次平行的工厂类层次时；或者
* 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

克隆模式

关键点：浅克隆和深克隆的区别用法注意

//实现Cloneable接口

**public** **class** Person **implements** Cloneable {

String name;

**int** age，salary;

Dog dog;//当克隆对象里面有对象成员的时候，对象成员也要实现Cloneable接口

@Override//如果不加具体成员的克隆方法就是浅克隆

**protected** Object clone() **throws** CloneNotSupportedException {//深克隆核心步骤！！

Object obj = **super**.clone();

Person p = (Person) obj;

// 深克隆：针对Person类里面的Dog成员进行克隆，要求Dog也实现Cloneable接口并且重写clone方法

p.dog = (Dog) **this**.dog.clone();

**return** obj;

}

。。。。。。//省略setget构造等方法

}

//Person类中的成员Dog对象，也要实现Cloneable接口！

//只有8大基本数据类型和String类型自带深克隆！

//其余的要先看API里面有没有实现Cloneable接口！

//如果没有实现Cloneable接口，就需要自己手动实现！

**public** **class** Dog **implements** Cloneable{

String name;

@Override//手动实现Cloneable接口，重写clone()方法

**protected** Object clone() **throws** CloneNotSupportedException {

// **TODO** Auto-generated method stub

**return** **super**.clone();

}。。。。。。//省略setget构造等方法

}

//调用情况

**public** **class** Client {

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

Person p = **new** Person("杨科",33,33000,**new** Dog("大狗"));

Person c = (Person) p.clone();

//通过hashcode来判断是否是深克隆或浅克隆

System.***out***.println(p.dog.hashCode()==c.dog.hashCode());

//因为浅克隆克隆的成员对象和源克隆成员对象是同一个。

//所以修改其中任何一个Dog对象里面的值的时候，源克隆和浅克隆体对象的值都会同时发生改变。

p.dog.setName("小小狗");

//是把另外一个完全不同的Dog对象交给P。要特别注意和深克隆的区别

p.setDog(**new** Dog("小狗"));

}

}

还可以通过序列化和反序列化来实现深克隆

//工厂模式结合原型模式来快速创建对象

**public** **class** PersonFactory {

**private** **static** Person *person*;

**public** **static** Person createPerson() {

//如果没有对象则，创建一个对象，并且保存原型

**if** (*person* == **null**) {

*person* = **new** Person();

**return** *person*;

}

**try** {//如果已经存在一个对象，则通过原型创建一个克隆对象

**return** (Person) *person*.clone();

} **catch** (CloneNotSupportedException e) {

e.printStackTrace();

**return** **null**;

}

}

}

### 创建型模式总结

单例模式：保证一个类只有一个实例

工厂模式

1. 简单工厂：用以生产同一等级结构中的任意产品。（增加新产品需修改源代码）
2. 工厂方法：用以生产同一等级结构中的固定产品。（支持增加任意新产品）

抽象工厂模式：用来生产不同产品族的全部产品。（支持增加产品族，无法新增产品）因为多个工厂类，都是拥有相同的一个创建对象方法，该方法的返回值，已经定死了只能是某个接口类型的产品，所以无法新增其他接口产品

建造者模式：实现了构造和装配的分离

原型模式：当通过new一个对象非常耗时的时候，通过克隆快速创建新对象

## 结构型模式：关注对象和类的组织，方便扩展升级

类的结构：采用继承来组合接口的实现

组合对象的方式实现新的功能

代理模式：为真实对象提供一个代理，从而控制对真实对象的访问

适配器模式：使原本由于接口不兼容不能一起工作的类可以一起工作

桥接模式：处理多层继承结构，处理多维度变化场景，使各个维度独立扩展

组合模式：统一处理部分和整体

装饰模式：增加一个对象的额外功能

外观模式：最少知道原则

享元模式：共享技术管理对象，节约内存

#### 桥接模式的区别

桥接模式的目的是为了将抽象部分与实现部分分离，使他们都可以独立地进行变化，所以说他们两个部分是独立的，没有实现自同一个接口，这是桥接模式与代理模式，装饰者模式的区别。

核心：把xxx的方法体内部内容，抽取成接口，通过实现不同的接口内容，来改变不同的xxx方法体内部内容

桥接模式强调以继承子类+方法内部接口，的方式来实现对一个方法流程的扩展

装饰模式，用了一部分继承：生成抽象装饰，使用继承抽象装饰类的方式来扩展原方法流程

代理模式，完全不用继承！！

装饰模式，一般不看重扩展的这一部分，在原方法部分的位置，或者一般在原方法之后即可！

而代理模式，更加看重，扩展原方法部分的位置！！不如在原方法之前或者之后！

#### 装饰模式和代理模式的关系

装饰者模式和代理模式都是持有实现并持有被装饰或被代理对象的引用。

装饰者模式结构上类似于代理模式，但是和代理模式的目的是不一样的，装饰者是用来动态地给一个对象添加一些额外的职责，装饰者模式为对象加上行为，而代理则是控制访问。

相同点：2个模式的整体写法，结构非常相似

不同点：

装饰模式：1.装饰类一般是abstract，不可以直接使用，而使用各种具体实现了的装饰类。2.内部持有的是被装饰对象方法的接口。3.目的是：在不能直接继承，或者继承类太多的前提下，为了更好的扩展被装饰类的方法的功能

代理模式：1.代理类一般是public，可以直接使用。2.内部持有的是被代理对象。（虽然也可以持有接口，但是为了突出代理核心思想，一般代理是持有具体对象）。3.目的是：为了控制和隐藏被代理对象

### 适配器模式adapter

Android中listView和Adapter的关系就是适配器模式

#### 适用性

* 你想使用一个已经存在的类，而它的接口不符合你的需求。
* 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。//类适配
* （仅适用于对象Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。//对象适配

关键点：无法给第三方类直接添加目标接口，但是又需要调用其方法

中国插头，通过适配德国插头规范，用在德国酒店的插座上的故事

**public** **interface** IGermanySocket {

**void** chargeGermany();//定义德国插口规范

}

**public** **class** GermanySocketImp **implements** IGermanySocket{

@Override

**public** **void** chargeGermany() {//定义德国插头

System.***out***.println("GermanySocketImp--德国插头");

}

}

**public** **interface** IChineseSocket {

**void** chargeChinese();//定义中国插口规范

}

**public** **class** ChineseSocketImp **implements** IChineseSocket{

@Override

**public** **void** chargeChinese() {//定义中国插头

System.***out***.println("ChineseSocketImp--中国插头");

}

}

**public** **class** GermanyHotel {//在德国酒店只能用德国插口规范

IGermanySocket germanySocket;

**public** GermanyHotel(){

}

**public** **void** setSocket(IGermanySocket germanySocket){//把德国插头传进去

**this**.germanySocket = germanySocket;

}

**public** **void** charge(){//用德国插头充电

germanySocket.chargeGermany();

}

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

GermanyHotel germanyHotel = **new** GermanyHotel();

// germanyHotel.setSocket(new GermanySocketImp());直接使用德国插头

// germanyHotel.charge();

// germanyHotel.setSocket(new MySocketAdapter(new ChineseSocketImp()));对象适配方式

germanyHotel.setSocket(**new** MySocketAdapterClass());//类适配方式

germanyHotel.charge();

}

}

//核心.1创建适配类，创建转换头

**public** **class** MySocketAdapter **implements** IGermanySocket {//对象适配方式

//定义一个适配器，实现了德国插口规范的中国插头

IChineseSocket chineseSocket;//核心.2创建准备去适配的接口

**public** MySocketAdapter(IChineseSocket chineseSocket) {

**super**();

**this**.chineseSocket = chineseSocket;

}

@Override//核心.3，在适配接口方法里面，调用准备适配的方法

**public** **void** chargeGermany() {//实际调用，是调用的中国插头的方法

chineseSocket.chargeChinese();

}

}

**public** **class** MySocketAdapterClass **extends** ChineseSocketImp **implements** IGermanySocket {

@Override//类适配方式

**public** **void** chargeGermany() {

**super**.chargeChinese();

}

}

### 代理模式proxy pattern

装饰者模式为对象加上行为，而代理则是控制访问

#### 代理模式和回调函数的区别

相同点：都是修改一部分流程内容

不同点：

代理模式重在原有流程(这个流程无法改变已经写死)之前，或者之后，或者之前之后新增

回调模式重在原有流程（这个流程由自己掌握）上，有一部分需要外界动态实现

如果是自己写的流程框架，当这部分代码需要外界来实现的时候，推荐用回调函数。方便！

如果是第三方框架，或者源码的类，这部分代码我无法从内部修改，这时使用代理模式，可以动态在原方法之前，之后添加新内容

还有一点，回调函数的位置相对比较死板。是固定在某个位置

而代理模式，非常方便，在原方法之前，之后添加，很灵活

#### 适用性

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一 些可以使用Proxy模式常见情况：

1. 远程代理（Remote Proxy）为一个对象在不同的地址空间提供局部代表。
2. 虚代理（Virtual Proxy）根据需要创建开销很大的对象。
3. 保护代理（Protection Proxy）控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。
4. 智能指引（Smart Reference）取代了简单的指针，它在访问对象时执行一些附加操作。 它的典型用途包括：
   * 对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它（也称为SmartPointers）。
   * 当第一次引用一个持久对象时，将它装入内存。
   * 在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

通过代理控制对对象的访问

关键点：详细控制某个对象的方法，进行方法的前置后置处理。对外只暴露代理类，从而实现真实角色的隐藏

#### 静态代理：自己写的代理类

//代理类同样实现了IParty接口，并且代理其中sing()方法

**public** **class** ProxySinger **implements** IParty{

**private** String name;

//持有被代理角色的引用

**private** IParty s;

//由于无法修改接口方法的参数，所以只有在构造方法中把真实角色传进来

**public** ProxySinger(String name, IParty s) {

**super**();

**this**.name = name;

**this**.s = s;

}

@Override//需要代理的方法。PS:无法修改接口方法的参数和返回值

**public** **void** sing() {

s.sing();

}

//省略setget接口其他方法

}

//调用情况

**public** **class** Client {

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

//创建真实角色

Singer s = **new** Singer("Jay");

//创建代理角色

ProxySinger proxySinger = **new** ProxySinger("杨科",s);

//通过只操作代理角色，从而隐藏真实角色

proxySinger.confer();

proxySinger.bookticket();

proxySinger.sing();//需要代理处理的方法

proxySinger.getMoney();

}

}

#### 动态代理：直接调用程序写好的代理类

JDK自带的Proxy代理类。原理是通过需要被代理的类实现的接口，来创建新的代理类去实现接口。填写接口的方法。

CGLIB原理是创建需要被代理对象的子类，重写父类的方法，所以需要被代理的类不能加final（必须能被继承），方法不能加final（必须能被重写）

1. **package** com.meituan.hyt.test3.service;
3. //1.接口
4. **public** **interface** IUserService {
5. **public** String getName(**int** id);
7. **public** Integer getAge(**int** id);
8. }
9. **package** com.meituan.hyt.test3.service.impl;
11. **import** com.meituan.hyt.test3.service.UserService;
13. //2.实现接口的类。。该类的所有接口的所有方法会全部被代理类去代理
14. **public** **class** UserServiceImpl **implements** IUserService {
15. @Override
16. **public** String getName(**int** id) {
17. System.out.println("------getName------");
18. **return** "Tom";
19. }
21. @Override
22. **public** Integer getAge(**int** id) {
23. System.out.println("------getAge------");
24. **return** 10;
25. }
26. }
27. **package** com.meituan.hyt.test3.jdk;
29. **import** java.lang.reflect.InvocationHandler;
30. **import** java.lang.reflect.Method;
32. //3.创建一个“代理处理规则”的类，作用是代理类通过“什么规则”来处理被代理对象方法
33. **public** **class** MyInvocationHandler **implements** InvocationHandler {
34. **private** Object target;
36. MyInvocationHandler() {
37. **super**();
38. }
40. MyInvocationHandler(Object target) {
41. **super**();
42. **this**.target = target;
43. }
45. @Override
46. **public** Object invoke(Object o, Method method, Object[] args) **throws** Throwable {
47. //如果被代理类的方法名称=代理规则回调函数所拿到的名称。则进行处理
49. **if**("getName".equals(method.getName())){
50. System.out.println("++++++before " + method.getName() + "++++++");  //因为接口中的方法会被全部代理。所以需要针对特定的方法调用做处理
51. Object result = method.invoke(target, args);//这里的args是代理对象调用方法后，原方法中的参数，所需的值
52. System.out.println("++++++after " + method.getName() + "++++++");
53. **return** result; //代理对象调用方法后，原方法返回的值
54. }**else**{
55. Object result = method.invoke(target, args);
56. **return** result;
57. }
59. }
60. }

系统jdk的代理具体用法

1. 创建被代理类
2. 创建代理规则类

InvocationHandler代理规则处理类。

专门动态的处理代理的方法

* + - 1. 可以处理代理方法的返回值
      2. 可以处理代理方法的内容
      3. 可以在原方法前后新添加新内容

1. 创建代理类
2. 代理类调用被代理类的方法
3. **public** **class** Main1 {
4. **public** **static** **void** main(String[] args) {
5. //1.创建被代理类，真实角色
6. IUserService userService = **new** UserServiceImpl();
7. //2.创建代理规则，传入真实角色，并设置如何去处理
8. InvocationHandler invocationHandler = **new** MyInvocationHandler(userService);
9. //3.创建代理类，传入接口，代理规则（系统自带JDK是传接口）
10. IUserService userServiceProxy = (IUserService)Proxy.newProxyInstance(userService.getClass().getClassLoader(),
11. userService.getClass().getInterfaces(), invocationHandler);
12. System.out.println(userServiceProxy.getName(1));
13. System.out.println(userServiceProxy.getAge(1));
14. }
15. }

invoke(Object proxy, Method method, Object[] args)

args[] 为 调用方法参数传入的值，有多个参数，所以是数组

p2.runDistance(111)。只有1个参数。所以args[0] = 111

//调用情况

Person person = new Person();//先创建了一个真实对象

Object obj = Proxy.newProxyInstance(Test.class.getClassLoader(),person.getClass().getInterfaces(),new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if ("eat".equals(method.getName())) {

method.invoke(person); //method执行，需要依据对象

System.out.println("eat-------invoke()");

}else if("jump".equals(method.getName())){

method.invoke(person);

System.out.println("jump-------invoke()");

}else if("runDistance".equals(method.getName())){ //3.在InvocationHandler处理规则里面，完全自定义重写方法内容

//4.如果要用到代理对象传入的值，则到Object[] args里面去取。并做处理 System.out.println("method.invoke(person,1280)-------before");

int dis = (int) method.invoke(person,12800);//5.被代理类的原方法可使用，也可以不使用。//这里是原方法被调用后，返回的值为dis

//method.invoke(person,12800)等同于执行person.runDistance(12800)!!!

System.out.println("method.invoke(person,1280)-------after");//6.可以添加新的业务逻辑

return dis+(int)args[0];//7.但是不管原方法使不使用，内容的返回值类型，都必须和原方法返回值类型一致！！//这里是最后代理对象调用方法后，返回的值

}if("fastrun".equals(method.getName())){

System.out.println("eat-------invoke()");

}

return null;

}

});

IRunDistance p2 = (IRunDistance) obj;//1.代理对象只能根据接口强转成相应的类，通过接口强转的方式隐藏方法。这里p2对象只能访问runDistance方法

System.out.println(p2.runDistance(111));//2.调用相应的方法传入值。只有1个值，所以值赋给了(int)args[0]//只能强制转换为需要代理类的接口，因为动态代理实现类是Proxy，没有方法

CGLIB的方式

1. **public** **class** CglibProxy **implements** MethodInterceptor {
2. @Override
3. **public** Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {
4. System.out.println("++++++before " + methodProxy.getSuperName() + "++++++");
5. System.out.println(method.getName());
6. Object o1 = methodProxy.invokeSuper(o, args);//调用被代理对象的方法
7. System.out.println("++++++before " + methodProxy.getSuperName() + "++++++");
8. **return** o1;
9. }
10. }
11. **package** com.meituan.hyt.test3.cglib;
13. **import** com.meituan.hyt.test3.service.IUserService;
14. **import** com.meituan.hyt.test3.service.impl.UserServiceImpl;
15. **import** net.sf.cglib.proxy.Enhancer;


19. **public** **class** Main2 {
20. **public** **static** **void** main(String[] args) {
21. CglibProxy cglibProxy = **new** CglibProxy(); //创建代理规则
23. Enhancer enhancer = **new** Enhancer();  //由于不是按照接口代理，所以要创建子类
24. enhancer.setSuperclass(UserServiceImpl.**class**); //子类设置父类
25. enhancer.setCallback(cglibProxy);  //子类设置代理规则
27. IUserService o = (IUserService)enhancer.create();  //子类创建对象
28. //这里创建的对象，可以转换为接口也可以转换为完整UserServiceImpl对象。不像JDK自带代理只能转换接口
29. o.getName(1);
30. o.getAge(1);
31. }
32. }

//类加载器的4种方式

//当前main方法的所在主类的类加载器

ClassLoader classLoader1 = Test.**class**.getClassLoader();

//当前线程的类加载器

ClassLoader classLoader2 = Thread.*currentThread*().getContextClassLoader();

//当前真实角色的类加载器

ClassLoader classLoader3 = s.getClass().getClassLoader();

//直接获取当前系统的类加载器

ClassLoader classLoader4 = ClassLoader.*getSystemClassLoader*();

}

}

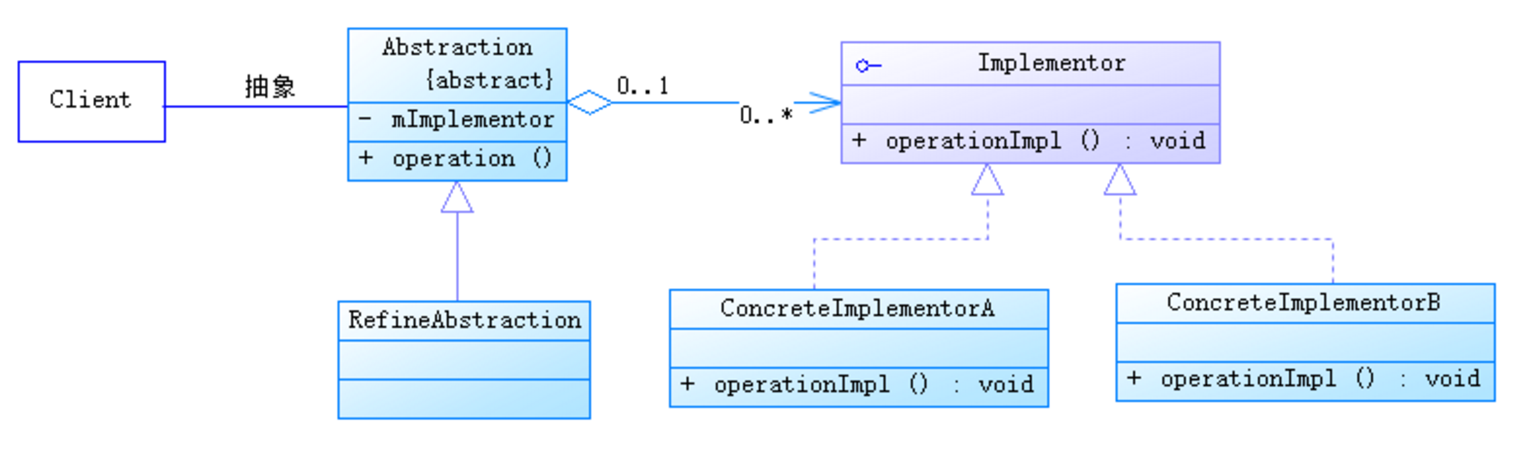
### 桥接模式bridge

实现多维度的扩展

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化

关键点：把类的某个方法实现部分抽取出来，让类的这个方法变成抽象方法，让其与具体实现该方法的方式分离。

抽象部分和  实现部分进行分离   使得他们可以独立的变化，这里面的抽象指的是方法的抽象或者叫功能的抽象，不是类。



Abstraction类中的operation方法，该方法的具体实现方式，方法体被抽取了出来。

通过接口的方式，来传入Abstraction类的operation方法体中！

通过接口的任意扩展，来达到operation方法体中的具体实现方式的变化

抽象类持有方法接口的引用

例如android中 AdapterView和Adapter的关系就是桥接模式。AdapterView中有个Adapter方法，AdapterView类包含Adapter的引用，这样就可以做到AdapterView维度的任意扩展和Adapter维度的任意扩展

**public** **interface** Brand { //定义品牌接口维度

**void** sale();

}

**public** **class** Lenovo **implements** Brand{ //任意品牌接口维度的扩展

@Override

**public** **void** sale() {

System.***out***.println("销售联想品牌");

}

}

**public** **abstract** **class** Machine {//定义设备抽象类维度

//机器抽象类里面持有品牌的接口引用

Brand brand;

//通过构造方法建立联系

**public** Machine(Brand brand) {

**super**();

**this**.brand = brand;

}

**public** **void** saleMachine(){//核心：把saleMachine的方法体内部内容，抽取成接口，通过实现不同的接口内容，来改变不同的saleMachine方法体内部内容

brand.sale();

}

}

**public** **class** DesktopComputer **extends** Machine{//任意设备维度的扩展

//继承机器抽象类，构造方法传入品牌类

**public** DesktopComputer(Brand brand) {

**super**(brand);

}

@Override

**public** **void** saleMachine() {

**super**.sale();

System.***out***.println("销售笔记本电脑");

}

}

//调用情况

**public** **class** Client {

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

//可以创建任意设备的任意品牌，设备和品牌2个维度随意扩展

Machine computer = **new** DesktopComputer(**new** Dell());

computer.sale();

}

}

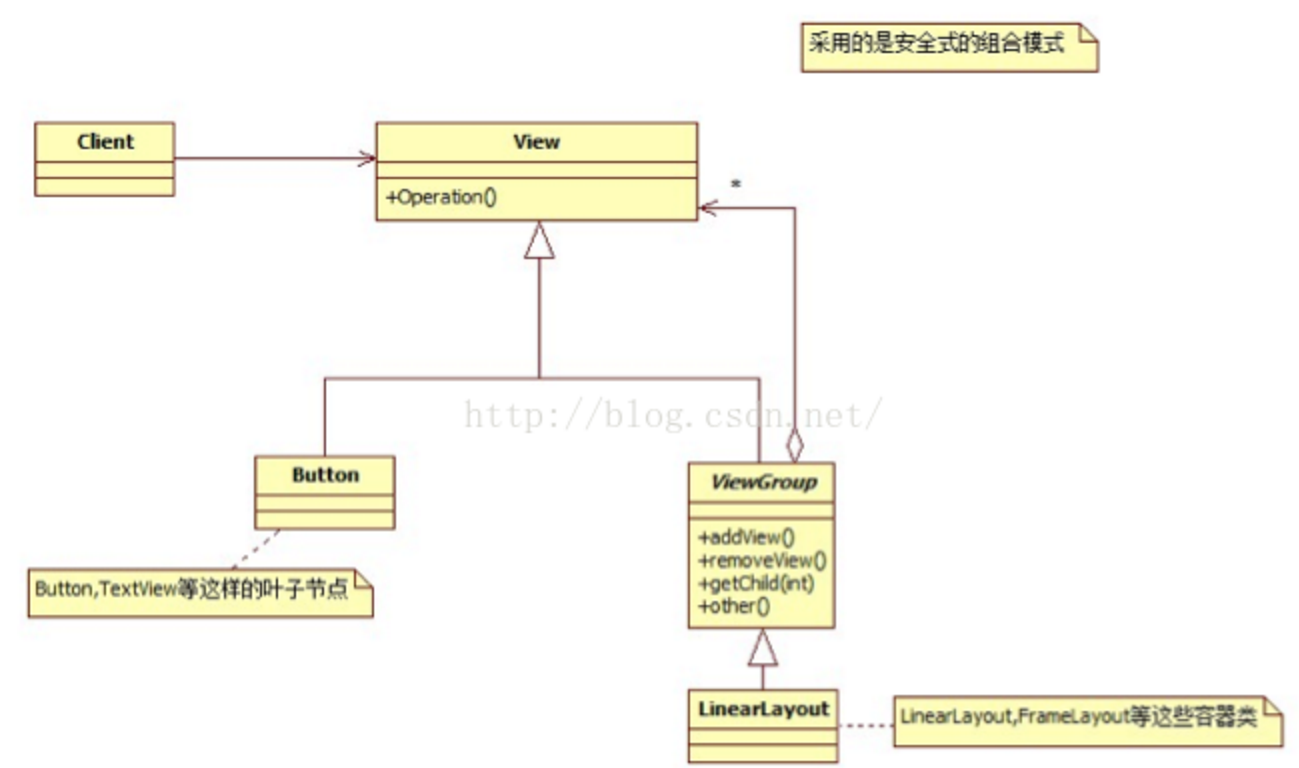
### 组合模式composite

#### 适用性

* 你想表示对象的部分—整体层次结构。
* 你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

核心思想：将客户代码与复杂的对象容器结构解耦

在android中，View类是典型的组合模式（如下图）



处理树形结构

关键点：客户端用统一的方式处理整体（容器）和部分（节点）

**public interface** IComposite **extends** IComponent**{**

**public void add(IComponent c);**

**public void remove(IComponent c);**

**public void getchild(int index);**

**}**

**核心：把叶子和容器共同处理的方法抽象出来。然后容器接口继承该接口，叶子接口直接使用该接口即可**

**public** **interface** IComponent {

**public** **void** test();

}

//容器类

**public** **class** TestComposite **implements** IComposite {

String name;

// 由于泛型是IComponent，所以既可以保存子节点又可以保持容器

List<IComponent> list = **new** ArrayList<IComponent>();

@Override//在整体和部分共有的方法中，进行统一的处理

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

System.***out***.println("整体名：" + name + "正在测试中...");

**for** (Component component : list) {

component.test();

}

}

@Override

**public** **void** add(Component c) {

list.add(c);

}

@Override

**public** **void** remove(Component c) {

list.remove(c);

}

@Override

**public** **void** getchild(**int** index) {

list.get(index);

}

**public** TestComposite(String name) {

**super**();

**this**.name = name;

}

}

//子节点类

**public** **class** TestLeaf **implements** IComponent {

String name;

**public** TestLeaf(String name) {

**super**();

**this**.name = name;

}

@Override // 子节点的方法

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

System.***out***.println("部分名：" + name + "正在测试中...");

}

}

**public** **class** Client {

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

IComponent A = **new** TestLeaf("子节点：A");

IComponent B = **new** TestLeaf("子节点：B");

IComposite S1 = **new** TestComposite("整体容器：S1");

// 把A,B，2个子节点添加到S1容器中

S1.add(A);

S1.add(B);

IComponent C = **new** TestLeaf("子节点：C");

IComposite S2 = **new** TestComposite("整体容器：S2");

// 把C子节点添加到S2中，再把S2容器添加到S1中。

S2.add(C);

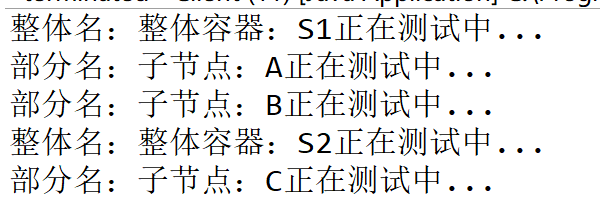
S1.add(S2);

// 最后通过统一处理S1，来达到统一处理全部节点和容器的效果

S1.test();

}

}



#### 安全式的组合模式：避免安全性的问题

//把部分和整体需要统一处理的方法抽象出来

**public** **interface** Component {

**void** test();

}

//容器接口继承统一抽象接口

**public** **interface** Composite **extends** Component{

**void** add(Component c);//把管理子节点的方式放在分支。属于安全式的组合模式

**void** remove(Component c);

**void** getchild(**int** index);

}

//节点接口继承统一抽象接口

**public** **interface** Leaf **extends** Component{

}

#### 透明式的组合模式：让客户看起来在接口层次上树叶和分支没有区别

//把部分和整体需要统一处理的方法抽象出来

**public** **interface** Component {

**void** test();

**void** add(Component c);//把管理子节点的方式放在分支。属于透明式的组合模式

**void** remove(Component c);

**void** getchild(**int** index);

}

//容器接口继承统一抽象接口

**public** **interface** Composite **extends** Component{

}

//节点接口继承统一抽象接口

**public** **interface** Leaf **extends** Component{

}

### 装饰模式decorator

比如一个人，给他装上火箭就能上天了，装上潜水服就能下海了，但本身还是个人，人没有任何变化。

核心：在被装饰对象无法继承的基础上，扩展部分方法的模式

* 需要拓展一个类的功能，增加附加职责时。
* 需要动态增加功能，并动态删除功能时。
* 当不能使用继承，但要提供继承的功能时。

#### 小结

装饰模式和前面的代理模式有点类似，容易把装饰模式看成代理模式。装饰模式是继承的一种替代方案，主要为所装饰的对象增强功能，动态的增加方法。而代理模式主要是为了控制对原有对象的访问权限，不对原有对象进行功能增强。

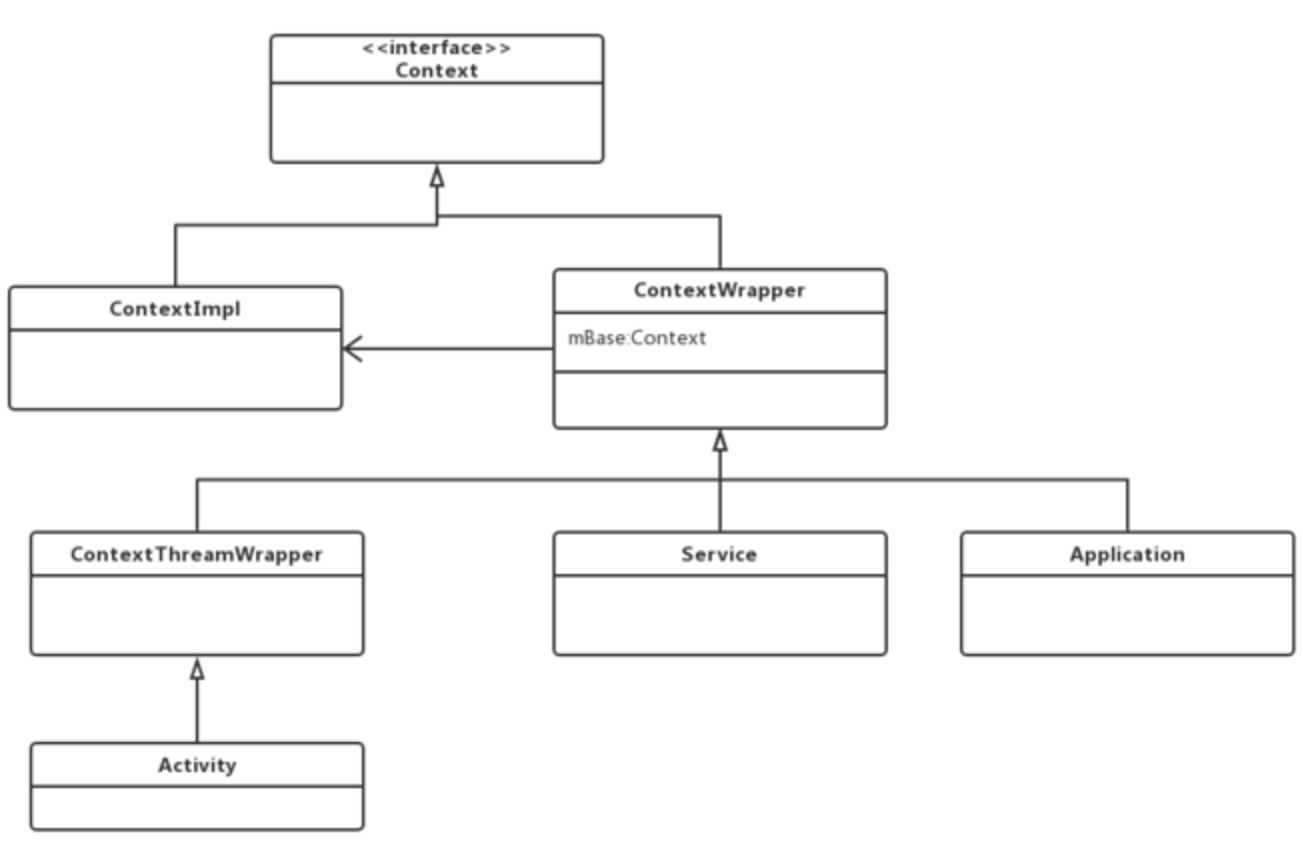
我觉得两者的区别主要是使用目的的区别。

Android中，上下文就是装饰模式。

ContextImpl是被包装具体的对象

ContextWrapper是装饰角色

Service Application是对具体的ContextImpl的功能的扩充



关键点：装饰角色持有被装饰角色接口的引用（接口里面是需要新增功能的方法）

//接口里面是需要新增功能的方法

**public** **interface** ICar {

**void** move();

}

//具体被装饰角色

**public** **class** Car **implements** ICar{

@Override//真实对象里面需要被增强的功能

**public** **void** move() {

System.***out***.println("车在陆地移动");

}

}

//装饰角色

**public** **abstract class** SuperCar **implements** ICar{//一定要实现ICar接口。因为这样才能实现类似new RedHair(new WhiteJacket(new WhitePants(new Worrior())));的嵌套调用

//持有被装饰角色接口的引用

**protected** ICar c;

//通过构造方法把被装饰角色传进来

**public** SuperCar(ICar c) {

**super**();

**this**.c = c;

}

@Override//通过重写方法来增强新功能

**public** **void** move() {

c.move();

}

}

//具体装饰角色

**public** **class** FlyCar **extends** SuperCar{

**public** FlyCar(ICar c) {

**super**(c);

}

@Override//具体增加新功能

**public** **void** move() {

//可以在老功能之前加新功能

//xxxx

c.move();//也可以完全重写老功能

//可以在老功能之后加新功能

System.***out***.println("车子现在会飞了！");

}

}

//调用情况

**public** **class** Client {

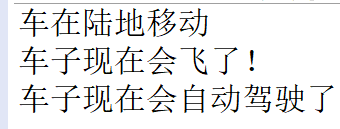
**public** **static** **void** main(String[] args) {

//新功能一层一层增加

**new** AICar(**new** FlyCar(**new** Car())).move();

}

}



#### 装饰模式调用注意

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

IAvatar worrior = **new** RedHair(**new** WhiteJacket(**new** WhitePants(**new** Worrior())));

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

}

调用结果：战士+白色裤子+白色上衣+红头发

注意：先调用Worrior的describe方法，再调WhitePants的describe方法，再调WhiteJacket的describe方法，再调RedHair的describe方法

像洋葱一样，由内向外依次调用

### 外观模式

关键点：迪米特法则，最少知道原则

把相对复杂的创建对象，调用方法过程单独封装到一个类中

### 享元模式FlyWeight

#### 核心

享元模式中会有一个工厂，工厂维护着一个容器，容器以键值对的方式存储，键是对象的内部状态，也就是共享的部分，值就是对象本身。

客户端从这个工厂获取对象，如果容器中存在这个对象就直接返回，不存在再创建新的对象并存入容器，避免了大量重复创建对象。

关键点：内部状态，可以共享。外部状态，不可以共享。享元池

//享元池

**public** **class** ChessFlyWeightFactory {

//定义map用来存取内部状态

**private** **static** Map<String, Chess> *chessmap* = **new** HashMap<String, Chess>();

**public** **static** Chess getChess(String color){

//如果在map里面已经有对象，则直接取出来

**if**(*chessmap*.containsKey(color)){

**return** *chessmap*.get(color);

}**else**{//如果没有，则创建一个对象放入map，并且返回对象

Chess temp\_c = **new** Chess(color);//核心：内部状态是key，通过构造方法传入方便

*chessmap*.put(color, temp\_c);

**return** temp\_c;

}

}

}

//调用情况

**public** **class** Client {

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

//只能创建唯一黑色棋子或者唯一白色棋子

Chess blackchess = ChessFlyWeightFactory.*getChess*("黑色");

Chess whitechess = ChessFlyWeightFactory.*getChess*("白色");

//显示坐标不一样。坐标是外部状态。外部状态变化需要每次都new

blackchess.display(**new** Coordinate(10, 20));

blackchess.display(**new** Coordinate(330, 120));

}

}

//外部状态，单独定义类

**public** **class** Coordinate {

**private** **int** x,y;

**。。。。。。**//省略构造方法，setget方法

}

//棋子类

**public** **class** Chess **implements** IChessFlyWeight{

**private** String color;

@Override//省略构造方法，setget方法

**public** **void** display(Coordinate c) {

System.***out***.println("棋子颜色是"+color+"---位置是在"+c.getX()+"---"+c.getY());

}

}

## 行为型模式：关注系统中对象之间的交互，研究系统在运行时对象之间相互通讯和协作。进一步明确对象职责。

类的行为模式：使用继承关系在几个类之间分配行为

对象的行为模式：使用对象聚合的方式来分配行为

### 责任链模式chain of responsibility

关键点：自己持有自己的引用，设置nextleader来处理责任链

关键点在于： //具体的处理方式交给具体的责任人来处理

**public** **abstract** **void** handlerRequest(LeaveRequest req);

LeaveRequest类的创建！！==处理链的关键！！

//抽象责任链类

**public** **abstract** **class** Leader {

**protected** String name;

//持有自己对象的引用

**protected** Leader nextLeader;

**public** Leader(String name) {

**super**();

**this**.name = name;

}

//设置下一级处理人

**public** **void** setNextLeader(Leader nextLeader) {

**this**.nextLeader = nextLeader;

}

//具体的处理方式交给具体的责任人来处理

**public** **abstract** **void** handlerRequest(LeaveRequest req);

}

**public** **class** Director **extends** Leader {

@Override//具体责任人的处理流程

**public** **void** handlerRequest(LeaveRequest req) {

**if** (req.getLeaveDay() < 3) {

System.***out***.println("主任：" + name + "，审批请假天数" + req.getLeaveDay() + "天通过！");

} **else** {//如果不满足条件则下一个责任人处理

**if** (nextLeader != **null**) {

**this**.nextLeader.handlerRequest(req);

}

}

}

**。。。。。。//省略构造方法**

}

//调用方法

**public** **class** Client {

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

//创建责任人

Leader a = **new** Director("马主任");

Leader b = **new** Manager("杨经理");

Leader c = **new** CEO("孙总裁");

//设置责任链关系

a.setNextLeader(b);

b.setNextLeader(c);

//创建请假条

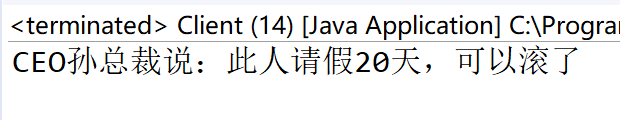
LeaveRequest req1 = **new** LeaveRequest("bug", 20, "不想做了");

//调用责任链头的方法，一层一层处理

a.handlerRequest(req1);

}

}



**public** **class** LeaveRequest {

String name;

**int** day;

String reason;

**public** LeaveRequest(String name, **int** day, String reason) {

**super**();

**this**.name = name;

**this**.day = day;

**this**.reason = reason;

}

**public** **int** getLeaveDay() {

**return** day;

}

}

### 中介者模式 mediator

关键点：解决多个同事对象相互复杂的关系。每个同事对象都持有中介者对象的引用。中介者对象持有所有同事对象。所有的同事对象之间的交互，统一由中介者对象处理

#### 方式一：

//中介者模式接口

**public** **interface** IMediator {

**void** register(String dname,IDepartment d);

**void** command(String dname);

}

//中介者类

**public** **class** President **implements** IMediator{

//存储多个同事对象

//方式一：用map来存放和管理所有的同事对象

**private** Map<String, IDepartment> m = **new** HashMap<String, IDepartment>();

@Override//注册同事对象

**public** **void** register(String dname, IDepartment d) {

m.put(dname, d);

}

@Override//取出同事对象调用命令

**public** **void** command(String dname) {

m.get(dname).selfAction();

}

}

**public** **interface** IDepartment {

//需要中介者统一调用的方法

**void** selfAction();

//需要各个同事对象自己调用的方法

**void** outAction();

}

**public** **class** Market **implements** IDepartment{

//持有中介者的引用

**private** IMediator m;

//通过构造方法传入中介者，并且调用register方法来注册到中介者的map中

**public** Market(IMediator m) {

**super**();

**this**.m = m;

m.register("市场部", **this**);

}

@Override

**public** **void** selfAction() {

System.***out***.println("市场部做自己的事，规划用钱");

}

@Override

**public** **void** outAction() {

System.***out***.println("市场部请求财务部支持");

//通过中介者的方式来达到访问财务部的效果

m.command("财务部");

}

}

//调用情况

**public** **class** Client {

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

//创建中介者

IMediator m = **new** President();

//创建各个同事对象并导入中介者

IDepartment financial = **new** Financial(m);

IDepartment market = **new** Market(m);

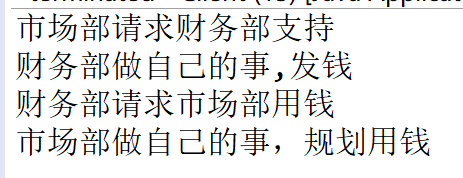
//各个同事对象的方法都是通过中介者来传送

market.outAction();

financial.outAction();

}

}



#### 方式二：

//中介者类持有各个同事实现类的对象

//在中介者类里面，统一处理这一类colleague的业务逻辑

**public** **class** MainBoard **implements** IMediator {

**private** CDDevice cdDevice;

**private** CPU cpu;

**private** GraphicsCard graphicsCard;

**private** SoundCard soundCard;

@Override

**public** **void** change(Colleague colleague) {

**if** (colleague == cdDevice) {

handleCD((CDDevice) colleague);

}

**if** (colleague == cpu) {

handleCPU((CPU) colleague);

}

}

**private** **void** handleCD(CDDevice cdDevice) {

cpu.decodeData(cdDevice.read());

}

**private** **void** handleCPU(CPU cpu) {

soundCard.playSound(cpu.getDataSound());

graphicsCard.vidoePlay(cpu.getDataVideo());

}

**public** **void** setCdDevice(CDDevice cdDevice) {

**this**.cdDevice = cdDevice;

}

**public** **void** setCpu(CPU cpu) {

**this**.cpu = cpu;

}

**public** **void** setGraphicsCard(GraphicsCard graphicsCard) {

**this**.graphicsCard = graphicsCard;

}

**public** **void** setSoundCard(SoundCard soundCard) {

**this**.soundCard = soundCard;

}

}

//同事类持有中介者对象的引用

**public** **abstract** **class** Colleague {

**public** IMediator mediator;

**public** Colleague(IMediator mediator) {

**super**();

**this**.mediator = mediator;

}

}

//同事实现类，调用中介者对象的方法

**public** **class** CPU **extends** Colleague {

**private** String dataVideo, dataSound;

**public** CPU(IMediator mediator) {

**super**(mediator);

// **TODO** Auto-generated constructor stub

}

**public** String getDataVideo() {

**return** dataVideo;

}

**public** String getDataSound() {

**return** dataSound;

}

//解析数据，分割音频和视频

**public** **void** decodeData(String data){

String[] tmp = data.split("，");

dataVideo=tmp[0];

dataSound=tmp[1];

mediator.change(**this**);//本类只处理本类自己的业务逻辑，其他逻辑交给中介者统一处理

}

}

**public** **class** CDDevice **extends** Colleague {

**private** String data;

**public** CDDevice(IMediator mediator) {

**super**(mediator);

}

**public** String read() {

**return** data;

}

**public** **void** load() {

data = "视频数据，音频数据";

mediator.change(**this**); //本类只处理本类自己的业务逻辑，其他逻辑交给中介者统一处理

}

}

**public** **class** GraphicsCard **extends** Colleague {

**public** GraphicsCard(IMediator mediator) {

**super**(mediator);

}

**public** **void** vidoePlay(String data) {

System.***out***.println("播放视频：" + data);

}

}

**public** **class** SoundCard **extends** Colleague {

**public** SoundCard(IMediator mediator) {

**super**(mediator);

}

**public** **void** playSound(String data) {

System.***out***.println("播放音频：" + data);

}

}

**public** **class** Client {

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

MainBoard mainBoard = **new** MainBoard();

CPU cpu = **new** CPU(mainBoard);

CDDevice cdDevice = **new** CDDevice(mainBoard);

GraphicsCard graphicsCard = **new** GraphicsCard(mainBoard);

SoundCard soundCard = **new** SoundCard(mainBoard);

mainBoard.setCpu(cpu);

mainBoard.setCdDevice(cdDevice);

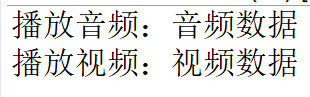
mainBoard.setGraphicsCard(graphicsCard);

mainBoard.setSoundCard(soundCard);

cdDevice.load();//调用业务处理起始点，load方法内部具体逻辑非常复杂，自己（CDDevice）只处理自己的业务逻辑，其他逻辑交给中介者来协调！！

}

}



### 备忘录模式memento

保存某个对象内部状态的拷贝，便于恢复

关键点：源发器类，备忘录类，负责人类

源发器类

**public** **class** Person {

String name;

**int** age;

**int** salary;

// 进行备忘操作并返回备忘录对象

**public** MementoPerson memento() {

**return** **new** MementoPerson(**this**);

}

// 数据恢复当前对象

**public** **void** recovery(MementoPerson m) {

**this**.name = m.getName();

**this**.age = m.getAge();

**this**.salary = m.getSalary();

}

......//省略set，get方法

}

备忘录类

**public** **class** MementoPerson {

String name;

**int** age;

**int** salary;

//直接在备份对象的构造方法里面，添加源数据进行备份操作

**public** MementoPerson(Person p) {

**this**.name = p.getName();

**this**.age = p.getAge();

**this**.salary = p.getSalary();

}

......//省略set，get方法

}

负责人类//负责人类专门管理和保存备份录对象

**public** **class** CareTaker {

//直接利用栈的FILO。来保存多个备忘录对象

Stack<MementoPerson> s = **new** Stack<MementoPerson>();

**public** CareTaker() {

**super**();

}

**public** MementoPerson getMementoPerson() {

**return** s.pop();

}

**public** **void** setMementoPerson(MementoPerson m) {

s.push(m);

}

}

//调用情况

**public** **class** Client {

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

//创建源数据

Person p = **new** Person("杨科", 33, 35000);

//因为不是去new MementoPerson所以必须增加负责人类来管理

CareTaker careTaker = **new** CareTaker();

//调用负责人的管理备份方法，方法里面传备份对象

careTaker.setMementoPerson(p.memento());

//调用Person的回复数据方法，方法里面传的是负责人类里面保存的备份对象

p.recovery(careTaker.getMementoPerson()); }

}

### 观察者模式Observer

定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。

关键点：这个主题对象在状态上发生变化时，会通知所有观察者对象，让它们能够自动更新自己。

//主题对象

**public** **class** Subject **implements** ISubject {

**private** String subjectState; //记录主题对象的状态

**private** List<Observer> list = **new** ArrayList<Observer>();//容器用来存放观察者

@Override//推送消息给观察者

**public** **void** inform() {

**for** (Observer o : list) {

o.update(**this**);

}

}

**public** String getSubjectState() {

**return** subjectState;

}

//当主题对象状态改变的时候，对观察者进行推送状态

**public** **void** setSubjectState(String subjectState) {

**this**.subjectState = subjectState;

**this**.inform();

}

@Override//对容器里面增加观察者

**public** **void** addObserver(Observer o) {

list.add(o);

}

@Override//对容器里面删除观察者

**public** **void** removeObserver(Observer o) {

list.remove(o);

}

}

//观察者对象

**public** **class** Observer **implements** IObserver {

**private** String obState; //观察者对象状态

@Override //观察者对象状态改变为主题对象状态

**public** **void** update(Subject subject) {

obState = subject.getSubjectState();

}

**public** String getObState() {

**return** obState;

}

**public** **void** setObState(String obState) {

**this**.obState = obState;

}

}

//调用情况

**public** **class** Client {

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

Observer o1 = **new** Observer();

Observer o2 = **new** Observer();

Observer o3 = **new** Observer();//创建多个观察者

Subject s = **new** Subject();//创建主题对象

s.addObserver(o1);

s.addObserver(o2);

s.addObserver(o3);//将观察者添加到主题对象中

s.setSubjectState("果然是个bug");//更新主题对象状态并推送给观察者

}

}

#### Java提供工具类实现观察者

让主题类继承Observable类，观察者实现Observer接口

**import** java.util.Observable;

//主题对象继承Observable类

**public** **class** Announcer **extends** Observable{

**private** String message;//设置状态

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String ss) {

**this**.message = ss;

**this**.setChanged();//标记对象为已改变对象

**this**.notifyObservers(message); //当主题对象状态改变的时候，对观察者进行推送状态

}

}

#### notifyObservers的内部源码：

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

**if** (!changed)// **this**.setChanged();changed=true

**return**;

arrLocal = obs.toArray();

clearChanged();

}

**for** (**int** i = arrLocal.length-1; i>=0; i--)

((Observer)arrLocal[i]).update(**this**, arg);//遍历Object数组，把实现了Observer的对象取出来，进行更新

// Observable o指的是主题对象，

//Object arg指的是主题对象**this**.notifyObservers(Object arg)方法中推送的内容arg

**public** **void** update(Observable o, Object arg)

//观察者实现Observer接口

**public** **class** Listener **implements** Observer{

**private** String myMessage;

**public** String getMyMessage() {

**return** myMessage;

}

**public** **void** setMyMessage(String myMessage) {

**this**.myMessage = myMessage;

}

@Override//观察者对象状态改变为主题对象状态

**public** **void** update(Observable o, Object arg) {

myMessage = ((Announcer)o).getMessage();

//也可以这样 myMessage = (String)arg;

}

}

### 状态模式state

不同的状态需要切换不同的行为

关键点：在适合不同状态下，同样的行为有不同的对应实现方式！

//遥控器有开和关2种状态

**public** **class** Controller {

**private** TvState state;

**public** Controller(TvState state) {

**this**.state = state;

}

**public** **void** setState(TvState state) {

**this**.state = state;

}

**public** **void** turnOn() {

state.tvOn(**this**);

}

**public** **void** turnOff() {

state.tvOff(**this**);

}

**public** **void** nextChinal() {

state.nextChinal(**this**);

}

**public** **void** lastChinal() {

state.lastChinal(**this**);

}

}

//定义开和关的接口

**public** **interface** ITvState {

**void** tvOn(Controller controller);

**void** tvOff(Controller controller);

**void** nextChinal(Controller controller);

**void** lastChinal(Controller controller);

}

//开状态的实现

**public** **class** OnState **implements** ITvState {

@Override

**public** **void** tvOn(Controller controller) {

}

@Override

**public** **void** tvOff(Controller controller) {

System.***out***.println("关机");

controller.setState(**new** OffState());

}

@Override

**public** **void** nextChinal(Controller controller) {

System.***out***.println("下个频道");

}

@Override

**public** **void** lastChinal(Controller controller) {

System.***out***.println("上个频道");

}

}

//关状态的实现

**public** **class** OffState **implements** ITvState {

@Override

**public** **void** tvOn(Controller controller) {

System.***out***.println("开机");

controller.setState(**new** OnState());

}

@Override

**public** **void** tvOff(Controller controller) {

}

@Override

**public** **void** nextChinal(Controller controller) {

}

@Override

**public** **void** lastChinal(Controller controller) {

}

}

//调用情况

**public** **class** Client {

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

Controller controller = **new** Controller(**new** OffState());

System.***out***.println(1 + "点击上一个频道");

controller.lastChinal();

System.***out***.println(2 + "点击关机");

controller.turnOff();

System.***out***.println(3 + "点击开机");

controller.turnOn();

System.***out***.println(4 + "点击下一个频道");

controller.nextChinal();

System.***out***.println(5 + "点击开机");

controller.turnOn();

System.***out***.println(6 + "点击关机");

controller.turnOff();

System.***out***.println(7 + "点击下一个频道");

controller.nextChinal();

}

}

//结果

1点击上一个频道

2点击关机

3点击开机

开机

4点击下一个频道

下个频道

5点击开机

6点击关机

关机

7点击下一个频道

//可以发现，同样的类，相同的方法，根据不同的状态，实现了不同的实现方式

#### 状态模式和策略模式的区别：

•可以通过环境类状态的个数来决定是使用策略模式还是状态模式。

•策略模式的环境类自己选择一个具体策略类，具体策略类无须关心环境类；而状态模式的环境类由于外在因素需要放进一个具体状态中，

以便通过其方法实现状态的切换，因此环境类和状态类之间存在一种双向的关联关系。

•使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环境类的状态会根据用户的操作自动转换。

•如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式；

如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式。

### 策略模式Strategy

分离算法选择实现

关键点：构建类的时候，在构造方法中，选择调用具体策略

对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某个问题，同时可以方便更换算法或者增加新的算法。由客户端决定调用哪个算法。

\* 实现起来比较容易，符合一般人员开发思路

\* 假如类型特别多，策略算法特别复杂，整个条件语句会变得很长，难以维护

\* 如果有新增类，要修改原代码

\* 不符合开闭原则

缺点：用户要知道所有的策略类。

//原来写法

public class TestStrategy {

public double getPrice(String type, double price) {

if (type.equals("普通客户小批量")) {

System.out.println("不打折，原价");

return price;

} else if (type.equals("普通客户大批量")) {

System.out.println("打九折");

return price \* 0.9;

} else if (type.equals("老客户小批量")) {

System.out.println("打八五折");

return price \* 0.85;

} else if (type.equals("老客户大批量")) {

System.out.println("打八折");

return price \* 0.8;

}

return price;

}

}

//调用情况

public class Client {

public static void main(String[] args) {

double p = new TestStrategy().getPrice("老客户小批量", 1000);

System.out.println(p);

}

}

public class Context {

//持有价格策略接口的引用

private PriceStrategy priceStrategy;

//通过构造方法把各种实现了价格策略的具体对象导入进来

public Context(PriceStrategy priceStrategy) {

super();

this.priceStrategy = priceStrategy;

}

//通过具体价格策略对象调用方法

public void printPrice(double standardPrice){

System.out.println(priceStrategy.getPrice(standardPrice));

}

}

//创建价格策略接口

public interface PriceStrategy {

public double getPrice(double standardPrice);

}

//具体的价格策略实现类

public class NewFew implements PriceStrategy {

@Override

public double getPrice(double standardPrice) {

System.out.println("新客户少量购买不打折，原价");

return standardPrice;

}

}

//调用情况

public class Client {

public static void main(String[] args) {

new Context(new NewFew()).printPrice(1000);

//非常方便，扩展性强，便于修改

new Context(new NewMany()).printPrice(1000);

}

}

//需要新增价格策略，只需要新增价格策略的实现类，其他全部不用修改

public class OldFew implements PriceStrategy {

@Override

public double getPrice(double standardPrice) {

System.out.println("老客户少量购买打八五折");

return standardPrice \* 0.85;

}

}

Client调用只需要新增一句话

new Context(new OldFew()).printPrice(1000);

### 模板方法模式template method

多态钩子函数

关键点：父类定义处理步骤，具体实现延迟到子类

public abstract class Restaurant {

protected void seat() {

System.out.println("找个座位坐下");

}

// 钩子函数，具体实现由子类去实现

protected abstract void eatsomething();

protected void checkout() {

System.out.println("吃完了，买单结账");

}

// 模板方法核心，定义一系列固定处理步骤。其中一个步骤具体如何处理，暂时未知，由子类去决定

public final void process() {

this.seat();

this.eatsomething();//吃什么，要由子类来决定

this.checkout();

}

}

//调用情况

public class Client {

public static void main(String[] args) {

// 使用匿名内部类的方式，来定义具体实现过程

Restaurant p2 = new Restaurant() {

@Override

protected void eatsomething() {

System.out.println("今天我要吃烤鸭");

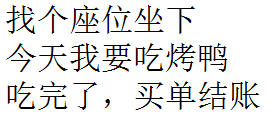
}

};

p2.process();

}

}



### 命令模式（不常用）底层的软件command

发送者-桥接模式-命令者-桥接模式-接收者

用2层桥。组成一个命令模式

关键点：将一个请求封装成一个对象

//发送者

**public** **class** Invoke {

//持有命令者的引用

**private** Command c;

**public** Invoke(Command c) {

**super**();

**this**.c = c;

}

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

c.execute();//发送者与命令者打交道

}

}

//命令者

**public** **class** Command {

//持有接收者的引用

**private** Receiver r;

**public** Command(Receiver r) {

**super**();

**this**.r = r;

}

//发送者不直接和接收者打交道，而是和命令者打交道

**public** **void** execute(){

r.action();//命令者与接收者打交道

}

}

//接收者

**public** **class** Receiver {

**public** **void** action(){

System.***out***.println("Receiver...正在被执行");

}

}

//调用情况

**public** **class** Client {

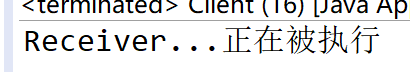
**public** **static** **void** main(String[] args) {

//发送者调用命令者，命令者再调用接收者，最终执行任务

**new** Invoke(**new** Command(**new** Receiver())).call();

}

} ·



### 迭代器模式（不常用）iterator

### 解释器模式（不常用）Interpreter：用JAVA构建自己的脚本语言

当我们需要解析字符串“1+2”的时候，我们会转换成js代码，jruby等来代替解释器。正则表达式的解释器，SQL语法的解释器，数学表达式的解释器。

### 访问者模式（不常用）Visitor

对于存储在一个集合里面的不同类型（同一个公共接口）的对象，要针对不同类型的对象使用不同的处理，用访问者模式

## Android开发中的设计模式

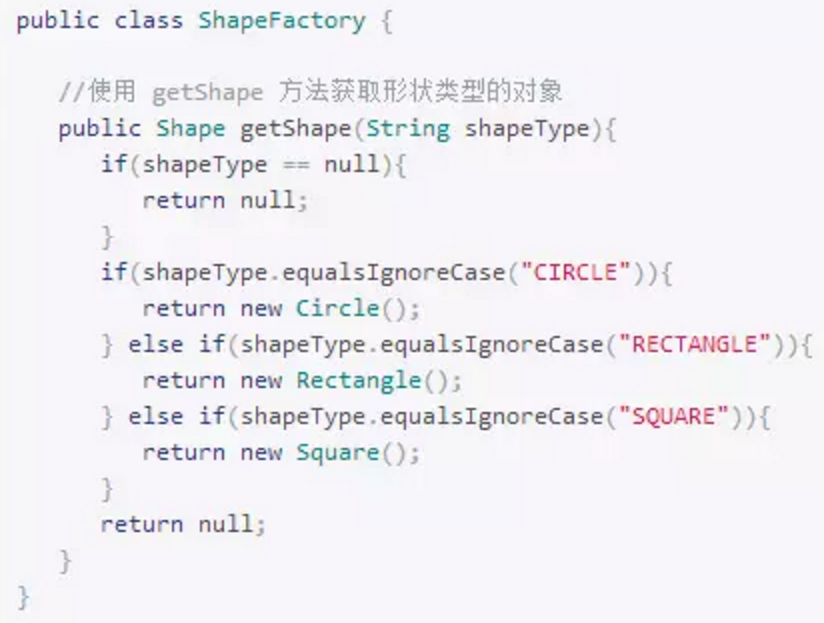
### 工厂模式：

当产品经理要求，同样的功能，有多种实现方式的时候，使用工厂模式

1.把功能方法变成接口

2.创建具体实现类

3.在实现类里面，通过type返回不同的实现功能类



### 中介者模式

中介者模式的精髓是管理同一类型(同一个接口，同一个抽象父类的不同实现类)对象的相互复杂访问。这样的中介者对象，目标明确，就是统一处理这一类型(同一个接口，同一个抽象父类的不同实现类)的对象的逻辑运算！

所以最佳的处理：中介者对象持有各种同事对象（同一个接口或同一个抽象父类的不同实现类对象），对这一类对象做统一处理。同事对象可以不持有中介者对象的引用

如果按照下面描述操作：中介者对象会变成杂牌军，无论什么类型，都交给该中介者对象。那么中介者对象的职责会造成复杂，而不是单一。因为什么类型的逻辑都要处理。

在项目中实际使用中介者模式来说，有时候将同事子类抽象出一个 Colleague 父类是不太合理的，因为子类之间的业务逻辑的不同，导致他们很难抽象出一些公用方法，所以这时候使用中介者模式，可以省去 Colleague 这个角色，让 Mediator 直接依赖于几个同事子类；

同时也可以不定义Mediator接口，把具体的中介者对象实现成为单例，这样同事对象不再持有中介者，而是在需要的时候直接获取中介者对象并调用；（这样可以，同事对象可以不持有中介者对象的引用）

中介者也不再持有同事对象，而是在具体处理方法里面去创建，或获取，或从数据传入需要的同事对象。（这样不好，中介者对象一定要持有同事对象）

### 回调函数

平时开发中，经常写回调函数

回调函数是桥接模式的单维度简化版

桥接模式是把一个方法的内容完全抽象成接口，然后持有该接口，通过方法把接口对象传进去，来实现内容维度的变化。

回调函数是桥接模式中省去了自己还可以被继承的维度。

由于方法内容完全被抽象出来，所以一般桥接模式方法内容为null。

但是回调函数一般是对方法的步骤的补充

回调函数是观察者模式的简化版

两者非常相似！

**public** **class** Apple **extends** Fruit {

**private** ISee isee;//相当于观察者map集合

@Override

**public** **void** eatFruit() {

// **TODO** Auto-generated method stub

System.***out***.println("Apple--eatFruit");

}

**public** **void** seeApple(){

**if**(isee!=**null**){

isee.see();//相当于从观察者集合中取对象，然后遍历调用观察者方法

}

}

**public** **void** setSee(ISee is){//相当于把观察者注册到主题对象中

isee = is;

}

}

**public** **class** Demo03 {

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

Apple apple = **new** Apple();

System.***out***.println("~~Log.i~~setSee~~before");

apple.seeApple();

apple.setSee(**new** ISee() {

@Override

**public** **void** see() {

// **TODO** Auto-generated method stub

System.***out***.println("custom--see");

}

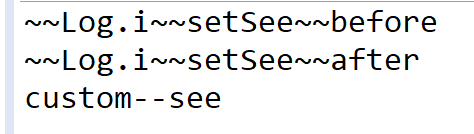
});//相当于把观察者注册到主题对象中

System.***out***.println("~~Log.i~~setSee~~after");

apple.seeApple();//相当于把主题对象内容推送给观察者

}

}



### 策略模式

animation.setInterpolator()其实用的就是策略模式，Android中有很多不同的插值器，都实现了Interpolator接口。

然后通过setInterpolator设置给Animation。

在动画执行的时候，会通过设置的计时器来在不同的动画时间执行不同的动画效果。

Animation animation = new AlphaAnimation(1,0);

animation.setInterpolator(new AccelerateDecelerateInterpolator());

imageView.setAnimation(animation);

animation.start();

Animatioin部分源码：

Interpolator mInterpolator；//相当于声明回调接口

public void setInterpolator(Interpolator i) {

mInterpolator = i;//相当于给回调函数的接口，赋对象

}

protected void ensureInterpolator() {

if (mInterpolator == null) {

//默认就是加速度插值器,在Animation初始化的时候执行。

mInterpolator = new AccelerateDecelerateInterpolator();

}

}

public boolean getTransformation(long currentTime, Transformation outTransformation) {

......

if ((normalizedTime >= 0.0f || mFillBefore) && (normalizedTime <= 1.0f || mFillAfter)) {

......

//通过策略模式来获取不同的值。//相当于回调函数

final float interpolatedTime = mInterpolator.getInterpolation(normalizedTime);

applyTransformation(interpolatedTime, outTransformation);

}

......

}

# 正则表达式

Java里面\需要转义。所以正则表达式\d 在java String里面是\\d

String matches（regex） 表示该字符串是否匹配该正则表达式！

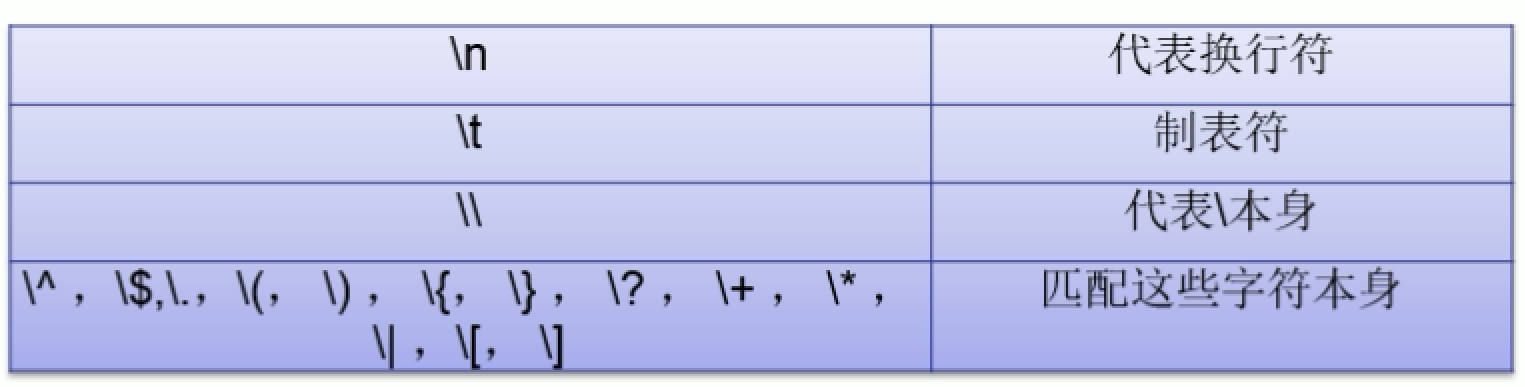
普通字符：字母，数字，汉字，下划线，没有特殊定义的标点符号

## 简单转义字符

\n 换行符

\t 制表符

\\ \本身



标准字符集合：能够与多种字符匹配的表达式

PS：大写是取反的意思

\d 任意一个数字：0-9任意一个

\D 表示非数字字符

\w 任意一个字母或数字或下划线：A-Z，a-z，0-9，\_ 中任意一个

\W 表示非字母，数字，下划线

\s 空格，制表符，换行符等空白符的其中任意一个

. 匹配任意一个字符（不含换行符）。[\s\S]是所有字符（含换行符）

单行模式下.可以包含换行符

## 自定义字符集合

[]表示单个字符！！

PS：[里面的关系是或的关系]。如[abc]=包含a，或b，或c

[ab5@] 匹配a或b或5或@

[^abc] 匹配abc以外的任意一个字符

[f-k] 匹配f到k之间的字符

[^A-F0-3] 匹配A到F和0到3以外的任意一个字符

## 量词

\d\d{6} 匹配寻找连续7位的数字。\d1位+\d{6}6位=7位

PS：花括号{}只修饰紧挨着的表达符

如果要表达12位。2种方式：\d{12}或者(\d\d){6}

[]表示单个字符！{}表示前面单个字符重复多少次！！

{n} 表示恰好重复n次

String regex = "[ab]{2}";

只有几种情况匹配，

“aa”算1次 “ab”算1次

“bb”算1次 “ba”算1次

其他均不算

{m,n} 表示至少重复m次，最多重复n次 大于等于m 小于等于n

PS：{m,n}默认是贪婪模式，先从最大n开始匹配。

如果是非贪婪模式，就是从最小m开始匹配。{m,n}?就等同于{m}

String regex = "[ab]{2，3}";

只有几种情况匹配，

“aa”算1次 “ab”算1次

“bb”算1次 “ba”算1次

“aaa”算1次 “aba”算1次

“bbb”算1次 “baa”算1次。。。等3个字符由a，b组成

其他均不算

{m,} 表示至少重复m次

String regex = "[ab]{2，}";

只有无数种情况匹配，但是不能出现其他非a，b字符

“aa”

“aaaaaa”…等无数情况

? 匹配表达式0次或者1次相当{0,1}

String regex = "[ab]?";

只有3种情况匹配，

“a”算1次

“b”算1次

“”空字符 算0次

+ 表达式至少出现1次相当于{1,}

String regex = "[ab]+";

只有无数种情况匹配，但是不能出现其他非a，b字符

“aaaabbbb”其中ab可以出现多次！！

“b”算！！

“”空字符 不算！！

“abcd”不算，因为出现非ab字符

\* 表达式不出现或出现任意次相当于{0,}

String regex = "[ab]\*";

只有无数种情况匹配，但是不能出现其他非a，b字符

“aaaabbbb”其中ab可以出现多次！！

“b”算！！

“”空字符 ！！！算！！

“abcd”不算，因为出现非ab字符

## 字符边界

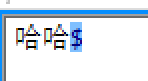
PS：匹配的不是字符而是位置，零宽

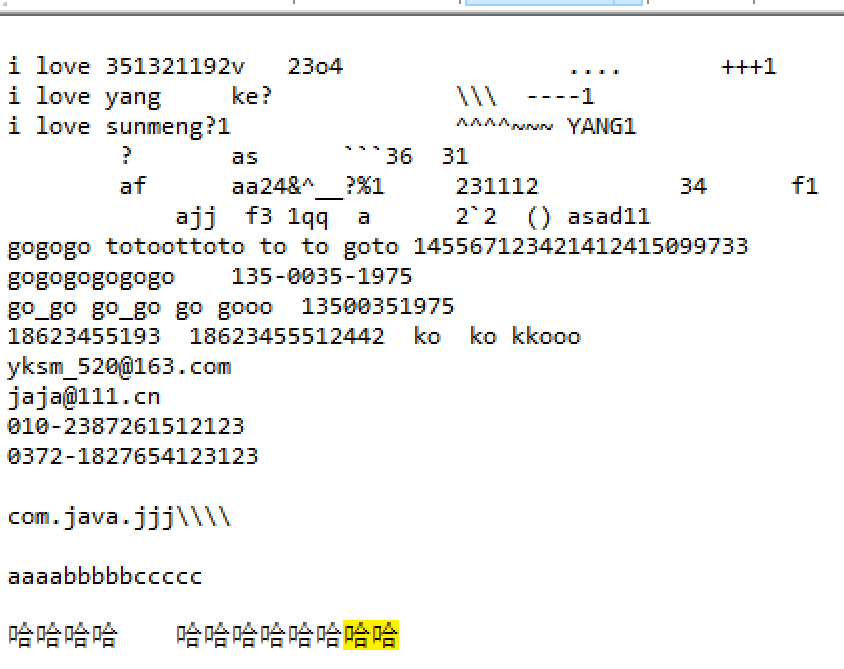
^ 与字符串开始的地方匹配（只找最开始位置）

查询一个字符串是否是以xxx来开头

$ 与字符串结尾的地方匹配（只找结尾位置）

查询一个字符串是否是以xxx来结尾





整个字符串。如果是以哈哈$结尾，则匹配成功！！

\b 以\W的方式匹配一个单词边界

如：\ba。表示匹配以a开头的。a前面必须是\W（就是指a前面不能有字母或数字或下划线）

\b哈哈\b

表示字符串，中是否有，哈哈，并且哈哈的前面和后面都不能有，字母，数字，下划线！！

忽略大小写模式

单行模式

整个文本看成一行

.可以包含换行符

多行模式

每一行都是单独的字符串

\A和\Z是不管当前是什么模式，都来强制表示开始位置和结束位置

## 选择符和分组

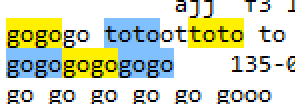
| 匹配左边或者右边

() 捕获组并且保持内容

分组的作用是，先按组匹配字符，然后再看后续的字符是否和之前匹配的相同。

如果相同。则结果为匹配字符\*2

([a-z]{2})\1



可以看见先存储go。因为go满足[a-z]{2}

然后再看go后续紧挨着的字符，是否和go相同。

如果相同。则结果为gogo

(?:表达式) 捕获组不保存内容，无法被反向引用

反向引用

\nnn 引用捕获的数据

每一个()都会分配一个编号，从最左边开始，计数1

通过反向引用，对分组捕获的字符串进行引用

## 预搜索（零宽断言）

PS：定位位置后。\w(?=表达式)。其中\w指的是位置左一个。如：

#匹配后面为\_path，结果为t  
  'product\_path'.scan /\w (?=\_path)/

(?=表达式) 定位的位置在表达式的前面

**(?=exp):零宽度正预测先行断言，它断言自身出现的位置的后面能匹配表达式exp。**

#匹配后面为\_path，结果为product  
  'product\_path'.scan /(product)(?=\_path)/

(?<=表达式)定位的位置在表达式的后面

**(?<=exp):零宽度正回顾后发断言，它断言自身出现的位置的前面能匹配表达式exp**

#匹配前面为name:，结果为wangfei  
'name:wangfei'.scan /(?<=name:)(wangfei)/ #wangfei

(?!表达式)

**(?!exp):零宽度负预测先行断言，断言此位置的后面不能匹配表达式exp。**

#匹配后面不是\_path  
'product\_path'.scan /(product)(?!\_path)/  #nil  
#匹配后面不是\_url  
'product\_path'.scan /(product)(?!\_url)/  #product

(?<!表达式)

**(?<!exp):零宽度负回顾后发断言来断言此位置的前面不能匹配表达式exp**

#匹配前面不是name:  
'name:angelica'.scan /(?<!name:)(angelica)/  #nil  
#匹配前面不是nick\_name:  
'name:angelica'.scan /(?<!nick\_name:)(angelica)/#angelica

取电话号码座机：(0\d{2}-\d{8})|(0\d{3}-\d{7})

翻译：以0开头如果0后面是2位，则-后面取8位。如果0后面是3位，则-后面取7位

取手机号码：1[35678]\d{9}

翻译：以1开头，第2位是3或5或6或7或8都可以，再后面必须紧跟9位数字