# JAVA概述

## Java环境

jre-java虚拟机+核心类库（辅助虚拟机运行的文件）

jdk-jre+java开发工具

## JDK目录介绍

bin-存放开发工具：java.exe-启动java虚拟机解释并执行指定class文件；javac-启动java编译器对指定的java源文件进行编译

db-JDK7附带的轻量级数据库Derby

include-调用系统资源的接口文件

jre-java运行环境

lib-核心类库

src.zip-java源代码

# 基本知识

## 配置环境变量

Path环境变量引用系统变量：%JAVA\_HOME%\bin

配置临时环境变量：

set path 查看指定环境变量信息；

set path=E:/aa 设置临时环境变量

set path=E:/aa;%path% 原有环境变量基础上增加环境变量

配置classpath作用：java虚拟机在指定路径搜索class文件

set classpath=f:\; 设置classpath 无分号java虚拟机仅在classpath所保存的路径搜索class文件，加上分号会增加当前路径搜索

作业：

1、简述JVM、JRE、JDK的理解

JVM：JAVA虚拟机

JRE：JAVA运行环境，JRE=JAVA虚拟机+核心类库

JDK：JAVA开发工具集，JDK=JRE+JAVA开发工具

2、环境变量path和java\_home\classpath的作用

path: 控制台在任意路径下都能找到java的开发工具

java\_home: 通过引用系统变量，避免频繁修改path环境变量信息

classpath: 设置class文件的路径信息

3、class、java、javac的作用

class：定义类的关键字

java：启动java虚拟机解释并执行指定class文件

javac：启动java编译器并对源文件进行编译

4、main方法

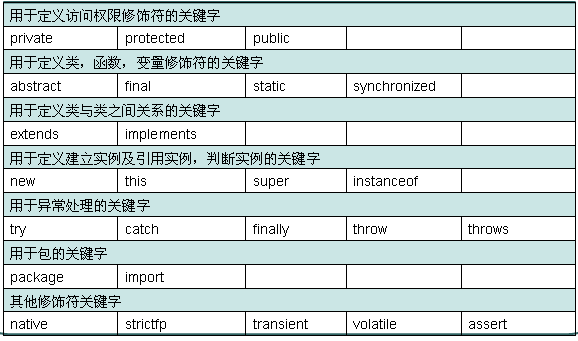
保证类独立运行

## 变量命名规范

标识符：

1. 由字母、数字、下划线组成
2. 不能以数字开头
3. 严格区分大小写
4. 命名有意义，见名知意
5. **类名与接口名单词首字母大写 DataSend**
6. **变量名与方法名首单词小写，其他单词首字母大写 dataReceive**
7. **包名全部小写 data\_analysis**
8. 常量全部大写，单词之间用下划线分割 TASK\_NAM
9. 关键字与保留字不能作为标识符

## 常用关键字



## 注释

单行：// 一般位于代码右边

多行：/\* xxx \*/ 一般位于代码上面

文档注释：/\*\* xxx \*/ 可以用于生成开发者文档，一般位于类或方法的上面

## 生成文档

javadoc -d 文档保存路径 java文件（类需要用public修饰，代表类是公共的），中文乱码：-encoding UTF-8

作者：author

版本：version

## 字符常量与字符串常量

字符常量：单个字符使用单引号

字符串常量：字符串使用双引号

## 进制

进制：有限数字表示所有数值

二进制表示：数字前加0b

八进制表示：数字前加0

十六进制表示：数字前加0x

## 基本数据类型

变量

变量命名规范：首单词小写，其他单词首字母大写

Java的基本数据类型：

（1）整数：byte：8bit -128-127（0属于正数）

short：16bit

int：32bit

long：64bit（声明时需要数字后加L，否则默认是int）

（2）小数：float（单精度浮点型）：32bit（声明时需要数字后加F，否则默认是double）

double（双精度浮点型）：64bit

（3）布尔：boolean：1字节或4字节，作为单独变量时占4字节，作为数组元素时占1字节

（4）字符：char：16bit（只能放单字符）

字符存储原理：字符转为数据，如a为97

变量声明：

int 变量1, 变量2;

# 运算符

## 强制数据类型转换

大数据类型→小数据类型（超范围可能会出错）

int i = 128;

byte b = (byte) i;（b=-128）

注：1、byte、short、char运算时都会自动转换成int计算，结果也为int类型，其他类型需要强转

2、不同类型数据运算时，结果取决于大数据类型

允许的情况：byte a = 10; 没有加任何标识的整数默认为整数，赋值给a时，编译器检查10并未超byte范围，允许赋值

不允许的情况：int i = 10; byte a = i; java编译器不能确定变量所存储的值，变量存储的值是在运行时才在内存中分配空间

## 负数的存储方式

**负数**在计算机中存储补码（原码取反+1）

二进制最高位是1肯定是负数

二进制最高位是0肯定是正数

## 算数运算符

+(加、正、连接（字符串拼接或数字与字符串拼接）)

1+2+3+"abc"+4+5+6：结果为：6abc456，说明从左至右依次相加，遇到字符串之后就变成了字符串连接

-

\*

/ 不同类型数据除取决于大类型数据

%(取余数，正负取决于被除数)

++(自增) 前自增：先自增 后使用；后自增：先使用，后自增

i = i++;

后自增运行原理：首先声明临时变量用于保存+1之前的值

int temp = i;

i = i + 1;

i = temp;

–(自减)

## 赋值运算符

= += -= \*= /= %=

## 比较运算符：可以比较不同类型数据，但是必须互相兼容

== （比较变量存储的值与数字 比较两个变量所存储的值 **比较引用变量所存储的地址是否一致**）

!= >= <= > <

## 逻辑运算符

& 与

| 或

^ 异或：两边不同为true

！非

&& 双与

|| 双或

## 位运算符：直接操作二进制位 应用：数据加密

&：与 有0（false）则0

|：或 有1则1（true）

^：异或 ：两边不同为1（true）。连续异或同一数，结果为本身（6^3^3）

~：取反

十进制转二进制补0的问题：不足8位不到8位，多余8位少于16位补到16位

位运算笔试题目：

1. 不用第三个变量交换两个变量的值：  
   相加法：

aa = aa + bb;

bb = aa - bb;

aa = aa - bb;

异或法：

aa = aa^bb; aa = 3^5

bb = aa^bb; bb = 3^5^5 = 3

aa = aa^bb; aa = 3^5^3

1. 取出一个二进制数据的指定位数：

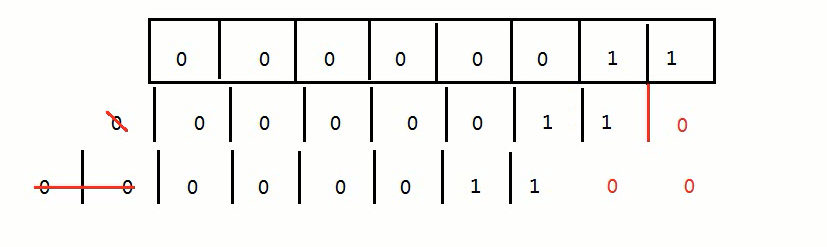
需要的与部分&1，不需要的部分&0

1. 移位运算符：

<< 左移：

先转成二进制，向左移动，超出的剔除，右边空缺用0补足

左移n位，变为原数值\*2^n



>> 右移

先转成二进制，向右移动，超出的剔除，左边空缺用0（正数）或1（负数）补足

右移n位，变为原数值/2^n

>>> 无符号右移

与右移的区别：左边空缺用0补足

由于负数的二进制位绝对值的二进制补码（反码+1），发生无符号位移之后容易变成一个很大的正数。

二进制转为负数：先看最高位是否为1，若为1说明是负数，然后减1，取反码，转为十进制加负号

笔试题目：用最高的效率计算2\*8

使用移位运算：2<<3;

## 三元运算符

结果需要变量接收或直接使用

布尔表达式? 值1:值2；

age > 18? “成年人”: “未成年人”，

## 运算优先级



练习

1、如果定义一个变量，如何使用变量。

2、基本数据类型有哪些？

byte short int long float double char boolean

3、'a'+1，结果是什么？为什么？都做了什么事情呢？

‘a’转为97 然后与1相加

4、++在前，在后的区别？

++在前为先自增，++在后为后自增

5、short s = 3; s = s+2; s+=2,有什么区别，为什么？

没区别

6、&和&&的区别?

&为与，两边条件都要判断，&&为双与，当左侧条件不满足时则右边条件不再判断

# 常用语句

## if语句

if (条件){执行语句};（如果只有一行语句，可以省略{}）

switch语句：优点是结构清晰，但不适用区间类型

switch (变量){

case 值1:

执行语句

break;

case 值2:

执行语句

break;

default:

不满足上述case时执行的语句

break;

}

注：（1）变量类型：byte short int char String 枚举

（2）case后需要常量

（3）程序执行case语句，然后执行default（无论顺序如何）

（4）语句在满足case后，如若没有break则继续执行，且不再判断case，直接执行case内的语句。

## while 语句

while (条件){执行语句}（一条语句可以省略大括号）

do while 循环语句

do {}while(条件)

与while语句的区别：while语句先判断后执行 do while 语句先执行后判断，二值条件是一样的

## for 语句

for (初始语句 终止条件; 步长或循环后的语句){循环语句}

for循环执行流程

第一次循环：初始化语句—判断条件—循环体—循环后语句

之后循环：判断条件—循环体—循环后语句

System.out.print(“\*\*”) //不换行打印

System.out.println(“\*\*”) //换行打印

## 转移字符

转义字符：\ （转化为字符本身）

“hello\” world”

\r 表示接受键盘输入，相当于按下回车。把光标移动到一行的首位置上，通常/r之前的会被之后的覆盖

\n 表示换行。

\t 制表符，相当于Table键（常用于对齐）

\b 退格键，相当于Back Space

\’ 单引号

\’’ 双引号

\\ 表示一个斜跨

windows 操作文件换行需要 \r\n

其他系统 仅需要\n

java 操作文件时需要咋main函数）后{前加throws Exception

## break与continue关键字

break: switch或循环中，用于结束switch 或跳出当前所在循环

笔试题目：break目前作用于内层for循环，如何让其作用于外层for循环？

## 给循环命名

f1:for

f2:for

XX;

break f1; //两层for循环都结束

continue：跳过本次循环体内容，继续下一循环

# 函数与数组

## 函数

函数：若没有返回值，则用void，可以有return，用于结束函数调用

若函数返回值为具体的数据类型，则函数需保证函数在任意情况下都有返回值

return关键字的作用：

1. 返回数据
2. 结束程序运行

## 函数重载

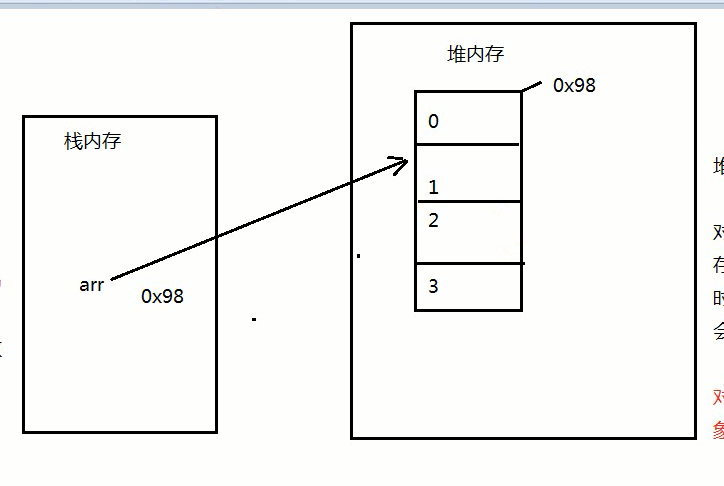
多个函数同一个函数名，但是形参的个数或类型不同，调用时自动匹配

## 数组

数组：同一类型集合容器

定义格式：double[] arr= new double[50];

=的作用：把数组对象内存地址赋予给arr变量



arr.length 查看数组长度

成员变量：方法之外，类之内

局部变量：方法之内

## Java虚拟机内存管理介绍

栈内存：存储局部变量，一旦超出作用域，释放内存空间

堆内存：存储对象数据（八种基本类型数据之外的引用数据类型，如数组，new出来的），对象使用完并不马上从内存中消失，而是等待垃圾回收（失去引用变成垃圾对象）

new创建的对象：在堆内存中开辟新空间，创建新对象

arr = null； 空指针（访问对象时会产生空指针异常）

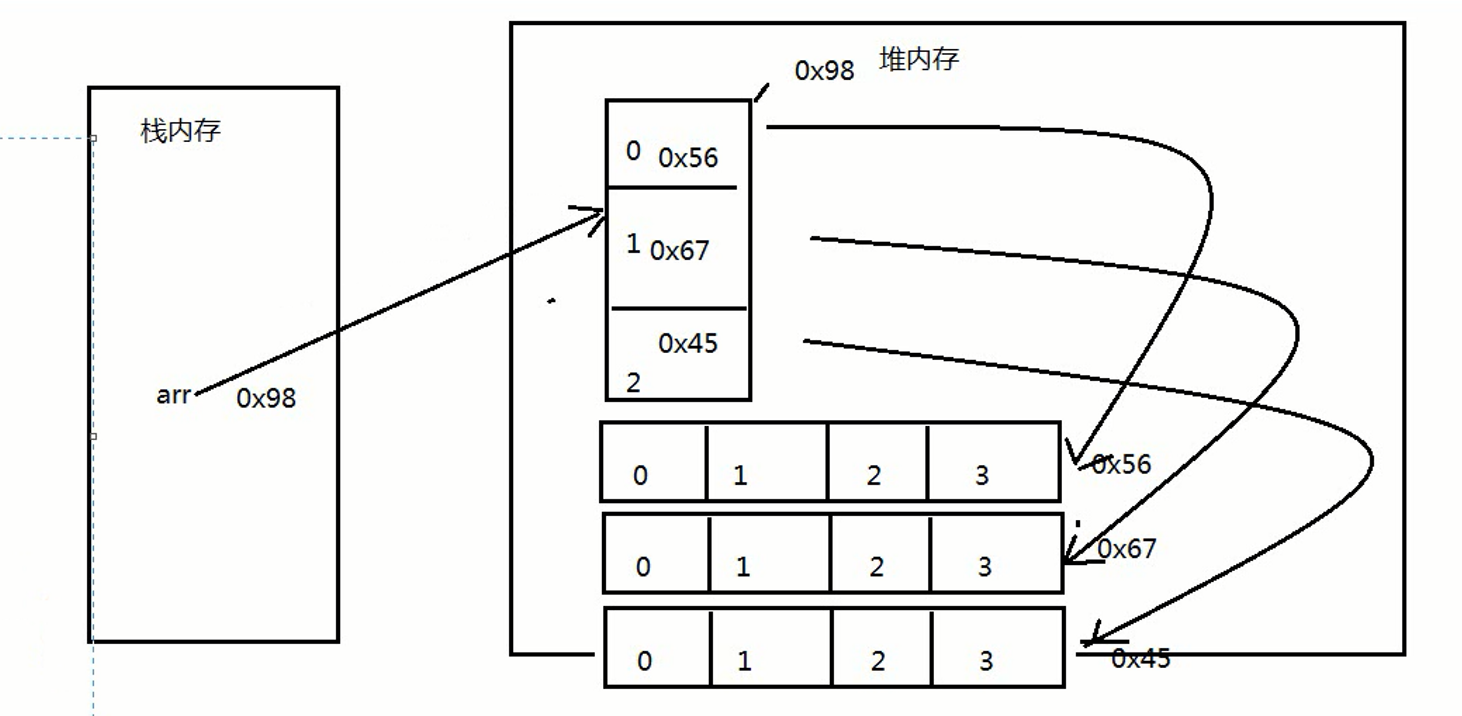
数组静态初始化：

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

二维数组：

double[][] a = new double[行][列];

double[][] a = {{},{},{}};



数组一旦被初始化，则长度固定

数组中元素之间的内存地址连续

# 类1

## 类定义

类：对某类型事物的**共性属性与行为**的抽取

自定义类的三步骤：

1. 自定义类
2. 通过类创建对象
3. 访问对象的属性或调用对象的功能

## 创建类

class{

成员变量

函数（方法）

}

成员属性有默认初始值 引用数据类型默认值为null

## 成员变量与局部变量的区别

1. 定义位置的区别：成员变量定义在类之内、方法之外；局部变量定于在方法之内
2. 作用上的区别：成员变量用于描述一类事物的公共属性；局部变量用方法内部使用
3. 生命周期：成员变量随对象存在；局部变量随方法存在
4. 初始值不同：成员变量有默认初始值，局部变量没有

## 匿名对象

匿名对象：没有引用类型变量指向的对象

一般不会个给匿名对象赋予属性值；

new classname().method;

应用场景：

1. 调用类方法，之后不再使用，执行完成后即成为垃圾对象；
2. 可作为实参调用函数

## 封装

权限修饰符

public修饰的成员变量或方法任何人都可以访问

private修饰的成员变量或方法只能在本类中访问

封装的步骤：

1. 使用private修饰需要被封装的属性
2. 提供公共方法设置或获取私有成员属性（setName getName）

一般实体类（描述一类事物的类）的所有成员变量都要封装

封装的好处：

1. 提高数据的安全性
2. 操作简单
3. 隐蔽了实现

类的划分：实体类、工具类

# 类2

## 构造函数

构造函数：给对应对象初始化，在创建对象时自动调用

修饰符 类名（形参）{

函数体（无返回值）

}

构造函数可以以函数重载的方式存在多个

类未设置构造函数时，编译器自动加，与类权限修饰符相同

java反编译工具：javap -c -l -private

## 构造代码块

构造代码块：给对象统一初始化

{

构造代码块

}

构造代码块作用：当两个构造函数时，可以将两个构造函数相同部分提取为构造代码块，无论用哪个构造函数构造对象，构造代码块都会执行

成员变量显示初始化、构造函数初始化、构造代码块初始化都对一个成员变量赋值时，Java编译器**将构造函数外的初始化移动到构造函数内（成员变量初始化与构造代码块初始化按顺序执行）**，且位置在构造函数初始化语句之前

局部代码块：大括号位于方法之内，作用：缩短局部变量的生命周期

## this关键字

this关键字：**表示所属函数调用者对象**

1. 解决同名成员变量与局部变量名相同问题，this.成员变量表示类的成员变量
2. 在一个构造函数中调用另外一个构造函数：this(传参数)，注意**该语句要放在其他初始化语句之前，**注意避免**递归调用**

## static修饰符

修饰成员变量，表示所有类对象**共享该成员变量**，一般显示初始化

静态成员变量只会在数据共享区维护一份

静态成员变量访问：对象.成员变量 或 类名.成员变量

## 静态成员变量与非静态成员变量的区别

1. 作用：静态成员变量共享一个数据给所有对象使用；非静态成员变量描述一类事物的公共属性
2. 存储：静态成员变量存储到数据共享区内存，且智慧存在一份数据；非静态成员变量存储在堆内存中，有n个对象就有n份数据
3. 生命周期：静态成员变量随类文件加载而存在，并随类文件消失；非静态成员数据随对象创建而存在。

## static修饰成员函数

通过对象或类名访问，推荐类名，好处是**不需要创建对象，节省内存**

静态成员函数可以访问静态成员（变量或函数），**不能直接访问非静态成员**

非静态成员函数可以访问静态或非静态成员

静态函数不能出现**this或super**关键字

## 静态数据的生命周期

类文件在**使用时**加载到内存中，静态代码块（代码块前加static）在类文件加载时被执行，静态成员变量即存放到数据共享区，因此，**静态成员变量优先于对象存在的**。但是**静态成员函数与非静态成员函数同时存在于内存中**

## static修饰一个函数的情况

一个函数不**直接**访问非静态成员，可以使用static，一般用于**工具类型方法**。

静态函数中存在对象时，可以访问非静态数据，只是不能直接访问

## main函数解析

public：保证JVM在任何情况下都能访问main方法

static：JVM不需要创建对象即可调用

void：不需要返回值

main：JVM可以识别的特殊函数名

String[] args：参数（字符串数组） 传参方式：java 类名 字符1 字符2 字符3

提供程序启动时传入需要的参数。

## 单例设计模式

单例设计模式：**保证一个类在内存中只有一个对象**

单例设计模式步骤：

饿汉单例设计模式（不管用不用，总会在类中声明一个类对象）

1. 私有化构造函数
2. 私有化声明并创建静态本类对象
3. 提供静态公共方法，获取本类对象

懒汉单例设计模式（用到时才会在类中声明一个对象）

1. 私有化构造函数
2. 私有化声明静态本类对象
3. 提供静态公共方法，获取本类对象（首次获取则创建本类对象）

推荐使用饿汉模式，保证线程（**操作系统能够进行运算调度的最小单位**）安全

# 类3

## ；类的继承

继承

学生类与人类是继承关系

继承格式：

**class 子类名 extends 父类名{**

**}**

父类**私有成员**和**构造函数**不能被子类继承（**构造函数**要注意！！！！！！！！！）

在子类构造函数中，默认会调用父类**无参构造函数；**

## super关键字

**super关键字：父类空间的引用**

**作用：**

（1）父类与子类具有同名成员（变量或方法）时，通过super访问父类成员（默认访问子类成员）

（2）子类构造函数中调用父类有参构造函数：**super(参数)，必须位于函数第一句，**且super与this不能同时出现在同一个构造函数中**调用其他构造函数**

## 方法重写

方法重写：父类与子类具有同名函数

1. 方法名与**形参列表**必须一致
2. 子类方法的权限要**大于等于父类方法权限**
3. 子类的返回值类型要**小于等于父类返回值类型**
4. **子类抛出的异常类型要小于或等于父类抛出的异常类型**（Exception-最坏；RuntimeException-小坏）（举例：线程的sleep方法）

## instanceof关键字

**instanceof关键字：判断一个对象是否属于指定类别**

前提：判断对象与指定类别必须存在**继承或实现**关系

使用格式：对象 instanceof 类别

## final关键字

**final关键字（最终修饰符）**

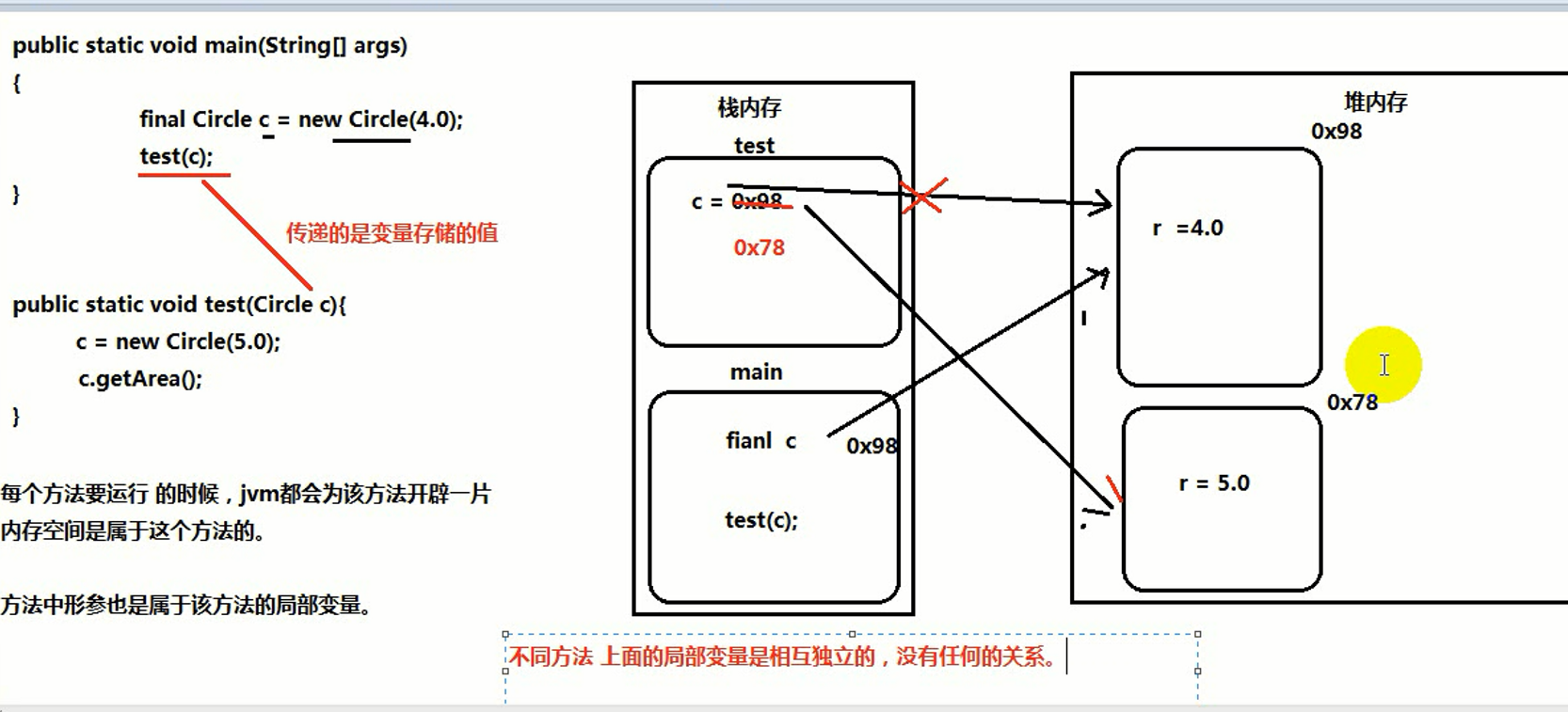
1. 修饰**基本类型变量**时，该变量值不能重新赋值，但要保证已经被赋值

final double pi = 3.14;

**常量的修饰符一般是public static final**

1. 修饰**引用类型变量**时，该变量**不能重新指向新的对象**

**每个方法运行时，JVM都会为该方法开辟一片属于该方法的内存空间，方法中的形参属于方法的局部变量**



1. 修饰**函数**时，该函数**不能被重写**
2. 修饰**类**时，该类不能被**继承**

# 抽象类与接口

## 抽象类

抽象类：**父类中某一行为难以统一描述，抽取行为声明，但不去实现，这种类称为抽象类**

**格式：public abstract void run();**

抽象方法没有函数主体，需要用**abstract**关键字修饰

若出现抽象函数，类也需要用**abstract**修饰，但是抽象类可以存在非抽象方法

抽象类不能创建对象，抽象类的构造函数用于子类创建对象时初始化父类属性

非抽象类继承抽象类，需要将抽象方法实现

**抽象类的优势：强制子类一定要实现指定的方法，否则编译报错**

abstract 不能与private\final\static共同修饰一个方法

值交换

当函数参数类型为**基本类型**时，调用方法时，传递给方法的变量实参，实际上是传递了**变量存储的值**。

**形参**是数据所属函数的**局部变量**

不同函数的局部变量之间互相独立

当函数参数类型为**引用类型**时，调用方法时，传递给方法的变量实参，实际上是传递了**地址**。

**一个类最多只能有一个直接父类，可以有多个间接父类**

## 接口（lambda表达式？）

格式

interface 接口名{

}

1. 接口编译之后会生成.class文件，**接口也是特殊的类**
2. 接口成员变量默认用**public、static、final**修饰，程序不写编译器会自动增加
3. 接口成员函数为**抽象方法**，默认用**public、abstract**修饰
4. 接口**不能创建对象、没有构造方法**。

接口的使用方式：用于类的实现，非抽象类实现一个接口时，必须要**把接口中所有方法实现**

class 类名 implements 接口名{

}

接口的作用：

1. 扩展功能
2. 定义约束规范（统一各模块的方法命名规范）
3. **程序解耦**

## 类与接口的关系

类与接口之间的关系：**实现**关系（类实现接口中定义的方法）

1. 非抽象类必须要实现接口中所有方法，抽象类允许实现接口中部分方法
2. 一个类可以实现多个接口（对于多个接口中的同名方法，只需要实现一个）

class 类名 implements 接口名1, 接口名2{

}

接口与接口之间的关系：**继承**关系

interface C extends A,B{ //C接口继承A\B接口

}

1. **一个接口可以继承多个接口**
2. 非抽象类实现接口时，需要将该接口继承的所有接口的方法实现

# 多态与异常处理

## 多态

1. **父类**的**引用类型变量**指向**子类**对象（**父类 a = new 子类()**）
2. **接口**的引用类型变量接口**实现类对象**

多态前提：存在**继承或实现**关系

注意事项

1. 子类与父类存在同名成员变量（静态、非静态）时，访问**父类**成员变量
2. 子类与父类存在同名**非静态**成员方法时，访问**子类**成员方法
3. 子类与父类存在同名**静态**成员方法时，访问**父类**成员方法
4. 不能访问**子类特有成员**

多态的应用

1. 用于**形参类型**时，可以接收更多类型的数据（父类作为形参时，可以接收子类）

需求：函数接收**任意类型**的图形对象，并打印面积与周长，保证代码可扩展

1. 用于**返回值类型**时，可返回更多类型数据（函数返回值为父类时，可以return子类，调用时用父类引用变量接收）

数据类型转换：

基本数据类型：小数据类型——大数据类型（**自动转换**）

大数据类型——小数据类型（**强制转换**）

引用数据类型：小数据类型——大数据类型（**自动转换**），如子类对象转为父类对象：Animal a = new Mouse(“老鼠”); **不能访问子类特有方法**

大数据类型——小数据类型（**强制转换**）， Mouse = (Mouse) a;

这样才**能访问子类特有方法**

接口**实现**关系下的多态

与继承关系的多态类似：**接口名 a = new 实现类();** 调用实现类的方法

内部类：包括**成员内部类和局部内部类**

**成员内部类**：一个类定义在另一个类的内部，该类称为内部类（类的嵌套）

内部类编译class文件名：**外部类名$外部类名**

成员内部类的访问方式：

1. 在外部类中提供一个方法，在方法中构建内部类对象；
2. 直接创建内部类对象**：外部类.内部类 变量名 = new 外部类(). new 内部类();**

成员内部类应用场景：

**事物A内部存在较为复杂的事物B，且事物B需要访问事物A的成员。**内部类可以直接访问外部类的所有成员。

使用要求：

1. 内部类与外部类存在同名成员变量，默认访问内部类成员变量（强制访问外部成员变量：外部类.this.同名成员变量）
2. **私有成员内部类**只能在外部类提供方法访问；
3. 成员内部类有**静态成员**时，内部类需用**static修饰（保证不依赖对象访问内部类）**

**静态内部类创建方式：外部类.内部类 变量名 = new 外部类.内部类();**

**局部内部类**：定义在方法内部

由于其只能在方法内部使用，所以其访问方式为：

在方法内部创建对象并使用（**局部内部类不能是静态类**）

使用细节：

局部内部类访问局部变量，局部变量需用**final**修饰（方法执行完毕后，局部成员变量在内存中消失，但是局部内部类对象还存在（等待垃圾回收），使用**final修饰成员变量**后，保证**成员变量不被修改**，然后复制给**局部类对象**使用）

**匿名内部类：无类名**

使用前提：存在**继承或实现**关系

语言：

new 父类名或接口名(){

//匿名内部类成员

}.方法();

访问匿名内部类的多个方法：

1. 方法1返回父类（return this），然后接着访问下一个方法

**new 父类名(){**

**//匿名内部类成员**

**}.方法1().方法2();**

1. 多态形式

**父类名 变量名 = new 父类名(){**

**//匿名内部类成员**

**};**

**变量名.方法1();**

**变量名.方法2();**

匿名内部类使用场景：

传递**实参（某函数形参为接口，实参需要传递接口的实现类时，则可用匿名内部类作为实参传递）**

**函数名（匿名内部类）**

**Test(new Dao(){**

**//实现**

**})**

## 异常

描述程序不正常的情况的类称为异常类，多个异常类形成异常体系

Throwable：所有错误或异常的超类

Error: 错误（AWT错误 JVM错误（StackOverFlow OutOfMemory））

Exception: 异常，包括运行时异常和编译时异常

## Throwable常用方法

1. toString(): 返回当前异常对象的完整类名+病态信息
2. getMessage(): 返回创建异常对象传入的字符串
3. printStackTrace(): 打印异常栈信息，定位代码异常位置

Error: JVM虚拟机或硬件引发的问题，一般不需要修改代码

Exception: 需要通过修改代码解决

JVM异常发现：

程序发现不正常情况时，立马创建对应的异常对象并调用printStackTrace方法

## 异常处理

**捕获处理（与用户直接打交道的代码出现异常进行捕获处理）**：

try{

可能异常代码

}catch (**捕获的异常类型 变量名**){

处理异常的代码

}

try{

int c =a/b;

}catch(ArithmeticException e){

XXX

}

捕获处理使用细节：

1. 经过异常处理的代码，后续代码可以继续执行
2. try块中出现异常语句，try块中后续代码不会继续执行
3. **多个catch可以捕获多种异常类型，但是须从小到大捕获**

**抛出处理（通知调用者）：**

（1）方法内部抛出异常时，需要在**方法声明上**抛出：

**public static void div(int a, int b) throws Exception{**

**}**

（2）调用声明异常的方法，调用者必须**处理异常（向上一级抛出或捕获）**

（3）方法内部抛出异常对象（**throw new Exception()**），throw语句代码后不再执行，方法停止执行

（4）一种情况下只能抛出一种异常对象

（5）throws与throw的区别：

**throws用于方法声明，声明抛出类型，throw用于方法内部，用于抛出异常对象**

**throws可以声明抛出多种类型异常，throw后面只能由一个异常对象**

# 异常处理与Jar包

## Exception分类

自定义异常类（自定义一个继承Exception的类）

Exception分类：运行时异常（RuntimeException及其子类）、编译时异常

方法内部抛出**运行时异常**，方法可不声明，调用者可不处理；方法内部抛出**编译时异常**，方法必须声明，调用者必须处理

## Finally块

Finally块：与Try块配合使用

**try{**

**int c =a/b;**

**}catch(ArithmeticException e){**

**XXX**

**}finally{**

**XXX**

**}**

finally不论是否异常都会执行，适合完成**资源文件释放**工作。

**try{**

**int c =a/b;**

**} finally{ //适用于内部抛出运行时异常，并且有资源被释放**

**XXX**

**}**

## 导包

有了包之后，类与类之间的访问需要加包名，可能通过**导包**解决，导包仅为了简化书写，但不会加载到内存。

**import 包名.类名**

**import 包名.\*（导出该包的所有的类，但不会作用于子包）**

权限修饰符：控制被修饰成员的范围可见性

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 可见范围 | public | protected | default | private |
| 同一类 | √ | √ | √ | √ |
| 同一包 | √ | √ | √ | × |
| 子父类 | √ | √ | × | × |
| 不同包 | √ | × | × | × |

## 打Jar包（工具jar.exe）

jar cvf 文件名.jar class文件或文件夹

打包之后，需要在清单文件制定入口类 格式：Main-Class: 包名.类名

jar包双击运行仅对图形化界面的程序起作用

jar文件作用：

1. 方便用户快速运行项目
2. 提供工具类给别人使用
3. 使用jar包的类时，需要设置classpath

## 模板模式

解决某类事情有些步骤是固定的，有些是变化的，可以采用模板，从而提高效率。

模板模式步骤

1. 写出该类问题中具体问题的解决方案
2. 分析代码，抽取会发生变化的代码并独立成一个方法（描述为**抽象方法**）
3. 使用final修饰模板方法，防止别人重写

# 常用类1

## Object类

Object是所有类的终极父类

常用方法

toString() 返回对象的字符串表示：**完整类名+@+对象hash码**

**可以通过对**toString()进行**重写**，进而输出需要的对象信息

equals() **默认**比较两个对象的内存地址，判读是否为同一个对象；

hashCode(): 返回对象的哈希码（可以理解为对象的内存）

**一般重写equals时需要同时重写hashCode();**

## 字符串类

创建字符串的方式

**（1）String s1 = “hello”;**

jvm首先检查字符串常量池（在方法区）是否该字符串对象，若有，则返回**内存地址**；若无则先在字符串常量池创建字符串对象再返回。

**String s1 = “hello”; 与 String s2 = “hello”; 内存地址是一样的 ==判断内存地址是否相等**

**（2）String s2 = new String(“hello”);（实际创建了两个对象）**

jvm首先检查字符串常量池（在方法区）是否该字符串对象，若有，则返回**内存地址**；若无则先在字符串常量池中创建字符串对象。最后在**堆内存复制一份字符串对象（无论字符串常量池是否存在字符串对象）**，再返回**堆内存**中字符串对象的内存地址。

* **String s1 = “hello”; 与 String s2 = new String(“hello”); 内存地址不一样（一个是字符串常量池地址，一个是堆内存地址）**
* **String s2 = new String(“hello”); 与 String s3 = new String(“hello”); 内存地址不一样（在堆内存中地址不一样）**
* s2.equals(s3) true 原因：String类对equals方法进行了重写，比较的是两个字符串内容是否一致。
* 字符串判断：**if (“hello”.equals(s2))** 避免**s2.equals(“中国”)中s2为空时程序报错**

## String构造方法

1. String s = new String(); //创建空字符串
2. String s = new String(byte[] bytes); //使用byte数组创建一个字符串对象，结果为数字对应的字母
3. String s = new String(byte[] bytes, int offset, int length); //bytes要解码的数组、offset解码开始的索引值、length字符长度
4. **String s = new String(char[] c); //使用字符数组构建字符串**
5. String s = new String(char[] c, int offset, int length); //指定索引值和长度
6. String s = new String(String s);
7. String s = new String(int[] i, int offset, int length);

## String方法

1. 获取方法

int length() 获取字符串长度

char charAt(int index) 获取指定位置字符

int indexOf(String s, int index) 查找子串s从index开始第一次出现的索引值，没有返回-1

int lastIndexOf(String s) 查找子串最后出现的索引值，没有返回-1

1. 判断方法

boolean endsWith(String s) 判断是否以指定字符串结束

boolean isEmpty() 判断长度是否为0

boolean contains(String s) 判断是否包含子串s

boolean equals(String s) 判断字符串**内容**是否一致（区分大小写）

boolean equalsIgnoreCase(String s) 判断字符串**内容**是否一致（忽略大小写）

1. 转换方法

char[] toCharArray() 字符串转为字符数组

byte[] getBytes() 字符串转为字节数组

1. 其他方法

替换：

String replace(char oldChar, char newChar)

String replace(charSequence oldChar, Sequenc newChar) // charSequence为String类的接口，可以传入字符串

分割

String[] split(String s) 根据s分割

截取

String substring(int index) 从index截取至最后

String substring(int startindex, int endindex) 从startindex截取至endindex-1（**左闭右开**）

转大小写

String toUpperCase(String s) //转大写

String toLowerCase(String s) //转小写

去空格

String trim(String s) //去除首尾空格

# 常用类2

## StringBuffer类

字符串的特点：**字符串为常量**，当字符串内容发生变化时，系统会**重新创建一个新的对象**，因此字符串不适宜频繁修改。

对于需要频繁修改的字符串，可以使用StringBuffer（字符串缓冲类）

构造无参数StringBuffer

StringBuffer s1 = new StringBuffer(); （默认初始容量为16字符（依赖长度为16的字符数组），不够时自动增加1倍+2）

## StringBuffer常用方法

返回StringBuffer

（1）增

**append(xxx)：尾部添加任意类型数据**

**insert(int index, xxx)：在索引值index处插入内容**

（2）删

delete(int start,int end): 根据**索引值区间**删除子串

deleteCharAt(int index): 删除index处的一个字符

（3）改

replace(int start, int end, String x): 将指定索引值区间内的内容**替换**为x

reverse(): 字符串翻转

setCharAt(int index, char c): 将index处的字符替换为c

substring(int start,int end): 截取指定索引值区间内的子串

ensureCapacity(int minvalue): 指定内部数组长度（默认16）（一般构造时输入int指定）

（4）查

int indexOf(String s, int fromIndex) 从fromIndex处开始查找子串s，返回索引值

capacity(): 查看**字符数组**长度（字符串长度小于16时，返回16，大于16时返回ceil(len/16)+2）

length(): 返回**字符串**长度

char charAt(int index): 返回index处的字符

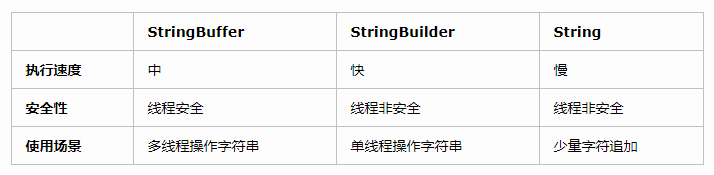
toString(): StringBuffer转换为String

## StringBuilder类

字符串另外一个缓冲类：StringBuilder

两个字符串缓冲类的区别：

1. StringBuffer是线程安全（**同一时间，只允许一个线程操作代码**）的，StringBuilder是线程非安全的
2. StringBuffer是jdk1.0出现，StringBuilder是jdk1.5出现，推荐使用StringBuilder（效率高）



## System类

1. 常用方法

* void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)：**将一个数组的指定位置、指定长度数组元素复制到另外一个数组的指定位置**

src-源数组；srcPos-源数组起始位置；dest-目标数组；destPos-目标数组起始位置；length-复制数组长度

* **long currentTimeMillis()**：获取以**毫秒**为单位的当前时间
* exit(int status)：退出JVM status=0表示正常退出
* gc()：**建议**JVM启动垃圾回收

当对象被垃圾回收时，会调用对象的**finalize()**方法

* getenv(String name)：获取环境变量
* getProperties(): 获取当前所有系统属性
* getProperty(String name): 获取当前指定名称系统属性

## RunTime类

表示应用程序的运行环境

每个 Java 应用程序都有一个 Runtime 类实例（单例设计模式），应用程序不能创建自己的 Runtime 类实例

Runtime getRuntime()：返回当前应用程序运行环境

Process exec(String[] cmdarray)：返回一个新的 Process 对象，用于管理子**进程**，**cmdarray可以是可执行文件的路径**

**注：**void destroy()：Process的方法，用于杀死子进程

long freeMemory(): 返回JVM空闲内存，以字节为单位

long maxMemory(): 返回JVM**试图**管理的最大内存

long totalMemory(): 返回JVM的**实际**内存总量

## Date日期类

Date date = new Date(); //获取当前系统时间

equals: 判断相等

Calendar：日历类

Calendar c = Calendar.getInstance(); //获取当前系统时间

获取年：c.get(c.YEAR)

获取月：c.get(c.MONTH)+1

**SimpleDateFormat** 日期格式化类

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

format(Date d)：日期转化为指定格式字符串

parser(String s)：把字符串转为日期

## Math类

ceil: 向上取整; floor：向下取整; round: 四舍五入取整

random：0-1随机数

## Random随机数类

Random r = new Random();

nextInt(10) 产生0-10随机整数

# 进程与线程

## 进程定义

进程是一个具有一定**独立功能**的程序在一个数据集上的一次动态执行的过程，是**操作系统进行资源分配和调度的独立单位**，是**应用程序运行的载体**。

**进程**一般由**程序，数据集合和进程控制块**三部分组成。程序用于描述进程要完成的功能，是控制进程执行的指令集；数据集合是程序在执行时所需要的数据和工作区；程序控制块包含进程的描述信息和控制信息是进程存在的唯一标志。

任务调度采用的是**时间片轮转的抢占式调度**方式，而**进程是任务调度的最小单位**，**每个进程有各自独立的一块内存**，使得各个进程之间内存地址相互隔离。

## 进程特征

动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的；

并发性：任何进程都可以同其他进行一起并发执行；

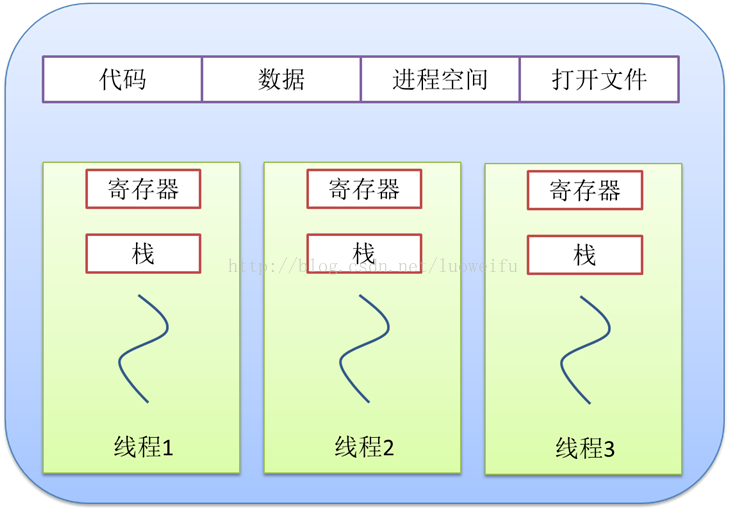
独立性：进程是系统进行资源分配和调度的一个独立单位；

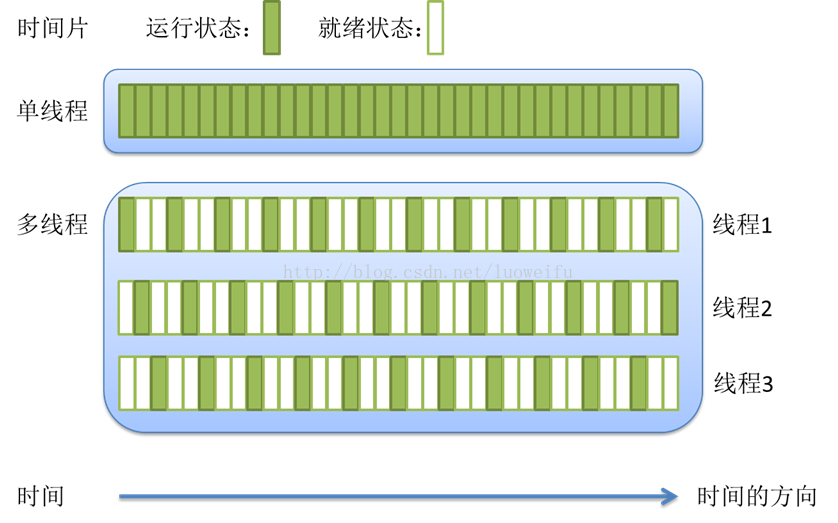
结构性：进程由**程序，数据和进程控制块**三部分组成。

**一个CPU在一个时间片内只能执行一个进程，通过短时间切换进程从而实现执行多进行进程任务。**

## 线程定义

线程是**程序执行的最小单位**，一个进程可以有一个或多个线程**同时执行任务**，**各个线程之间共享程序的内存空间(也就是所在进程的内存空间)**。一个标准的**线程由线程ID，当前指令指针PC，寄存器和堆栈组成**。





**运行程序时，JVM都会创建一个main线程执行main方法。**

**一个Java应用程序至少几个线程？**

1. **主线程执行main方法**
2. **垃圾回收线程**

## 多线程的优缺点

多线程好处：

1. 一个进程可以同时多个任务
2. 提高**资源利用率，而非效率**

多线程弊端：

1. 增加CPU负担
2. 降低一个进程中线程被执行的概率
3. 引发线程安全问题
4. 存在死锁现象

## 创建多线程

（1）方法一：

* 继承Thread类
* 重写run方法：将本线程需要执行的代码写在run方法中
* 创建Thread子类对象，调用start()方法开启线程

（2）方法二（**推荐使用，原因JAVA单继承，多实现**）：

* 自定义类实现**Runnable接口**
* 实现Runnable接口的run方法，并把线程的任务定义在run方法上
* 创建Runnable实现类对象
* 创建**Thread类对象，并把Runnable实现类对象作为实参传入**
* 调用Thread类对象的start()方法，开启多线程

注：

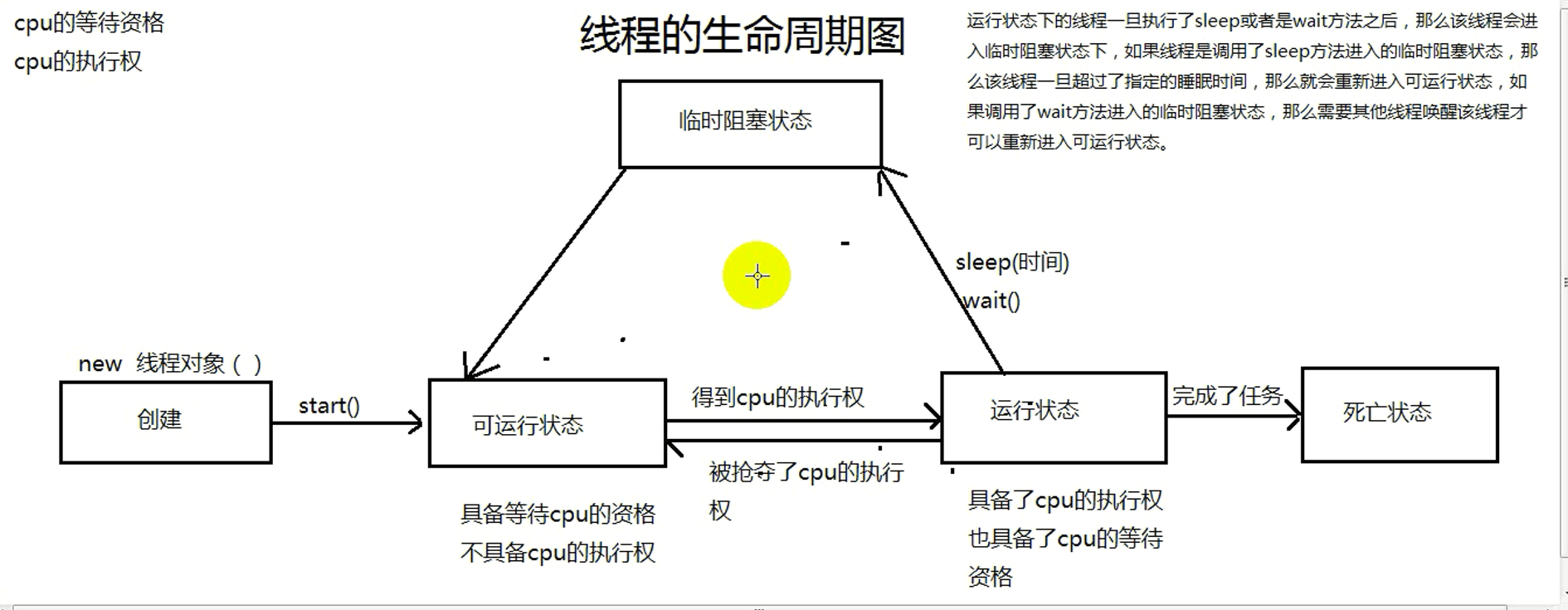
Runnable实现类对象并**不是线程对象**

Runnable实现类对象作为实参传递给Thread对象的原因：

**Runnable实现类对象的run方法作为线程的任务代码取执行**。

## 线程生命周期

* 创建线程子类对象（**创建状态**：不具备CPU等待资格和执行权）
* 执行start()：**可运行状态**，具备CPU等待资格
* 获得CPU的执行权时，则进入**运行状态**，同时具备CPU等待资格
* 完成任务后，进入**死亡状态**
* 线程调用**slpeep**方法：进入**临时阻塞状态**，超过睡眠时间时，重新进入**可运行状态**
* 线程调用**wait**方法：进入**临时阻塞状态**，需要**其他线程唤醒**才能重新进入可运行装填



## 线程常用方法

Thread(String name): 构造有名字的线程子类对象

getName(): 获取线程名字

setName(String name): 设置线程名字

sleep(): 睡眠 静态方法，哪个线程执行该方法哪个线程睡眠，一般用**Thread.sleep()，静态方法，**使用该方法时，会有异常，但是只能捕获，不能抛出，**原因：父类Thread的run()方法没有抛出异常，而子类异常范围不能大于父类，因此不能抛出。**

**currentThread(): 静态方法**，返回当前**线程对象**，Thread. currentThread()，代码写在哪里就返回哪个线程的对象。在线程子类中，Thread. currentThread()相当于this

getPriority(): 返回当前线程的优先级，默认为5，优先级越大，执行概率越高。

setPriority(): 设置优先级，范围1~10；

# 线程控制

## 线程安全问题

**出现线程安全问题的情况：**

1. 存在两个及以上线程对象，线程之间**共享一个资源**
2. 存在多条语句操作共享资源。

## 线程同步机制

线程安全问题解决方式：

线程同步机制：

（1）同步代码块

**synchronized(锁对象){**

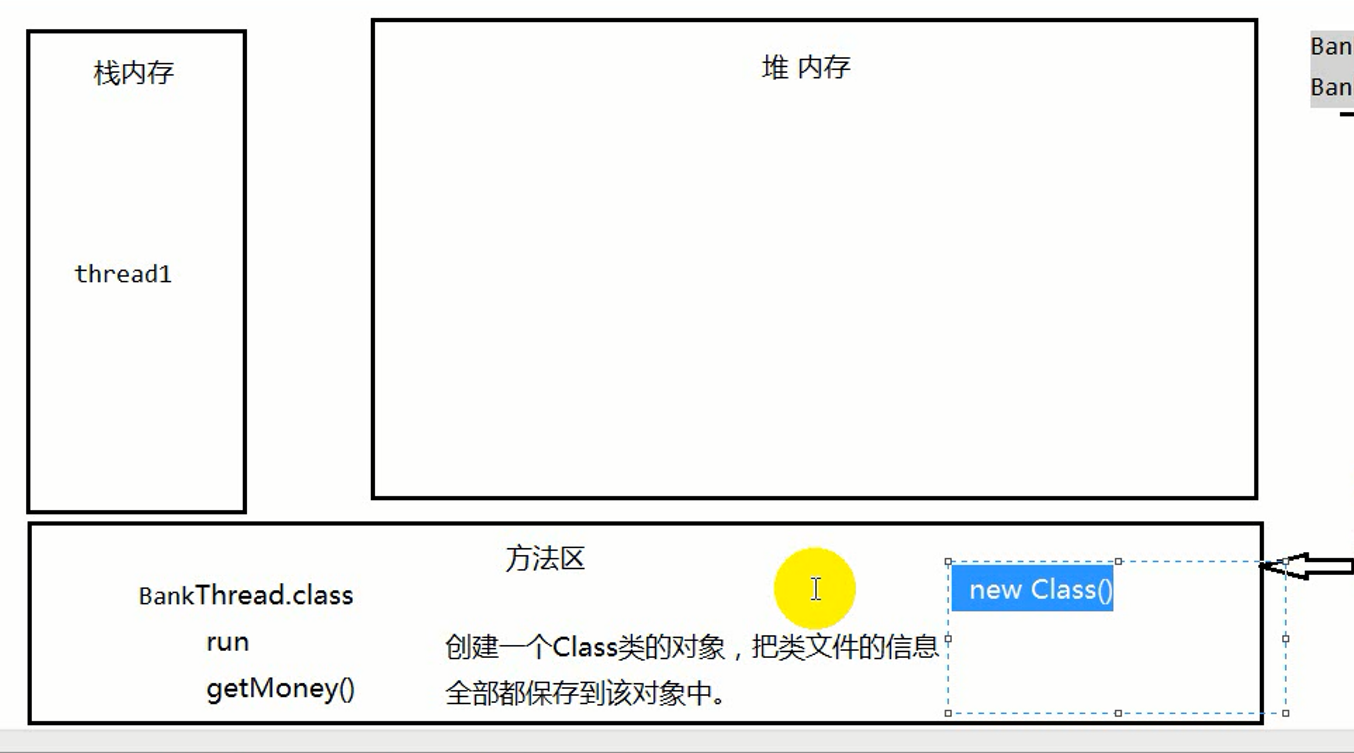
**需要同步的代码块**

**}**

* 任意对象都可以作为锁对象，**任何对象都有状态，java同步机制是操作对象的状态；**
* 同步代码块内执行sleep并不会解锁
* 多线程操作的锁对象必须是**唯一共享（Static修饰）**的，否则无效。
* 最简单的锁对象：**synchronized(“锁”) 原因：字符串常量在内存中是唯一的**

（2）同步函数：**使用synchronized修饰函数**

**非静态**同步函数的锁对象是**this对象**，**静态同步函数**的锁对象是当前所属类的字节码文件（**Class对象—Class也是Java中的一个类，用于存储类的信息，当存在多个类对象时，Class对象只有一份，因此是唯一的，可以作为锁对象**）。



推荐使用同步代码块：

1. 同步代码块锁对象可任意指定，同步函数的锁对象是股sing的
2. 同步代码块方便控制同步代码范围，同步函数的控制范围是整个函数。

## 死锁现象

存在两个以上共享资源的线程，互相等待对方解锁。

**synchronized（A）{**

**XXX**

**synchronized（B）{**

**XXX**

**}**

**}**

**synchronized（B）{**

**XXX**

**synchronized（A）{**

**XXX**

**}**

**}**

假设两个线程对象分别执行上下两块代码，分别锁定了A和B，然后分别等待B和A解锁，于是就发生了死锁现象。

## 线程通信

一个线程完成任务后，通知另外的线程取完成其他任务

**wait():** 等待，执行该方法时，线程进入等待状态，其他线程调用notify()方法时，才能唤醒；（进入以**锁对象**为标识符的线程池中等待）

**notify()**: 唤醒等待的线程（唤醒线程池中以锁对象为标识符的等待线程中的一个）

**notifyAll()**:唤醒线程池中以锁对象为标识符的所有等待线程

1. wait()方法与notify()方法属于**Object对象**
2. wait()方法与notify()方法必须由**同一个锁对象**调用
3. wait()方法与notify()方法必须在**同步代码块或同步函数**中使用

## 线程停止

1. 通过变量（flag）控制循环的结束。
2. 针对进入**等待**状态的线程，修改变量后要**唤醒（notify()或（3） interrupt()）**，不然不会生效。
3. interrupt(): 清除线程的等待状态，被清除状态的线程会收到InterruptedException

## 守护线程（后台线程）

在一个进程中，如果只剩守护线程，则守护线程自动死亡

设置守护线程：线程类对象.setDaemon(true);

判断是否为守护线程：线程类对象.isDaemon();

加入线程（在一个线程执行程序中加入新线程）

当新线程执行**join()**方法时，则原线程任务让步给新线程执行，等新线程执行完毕后继续执行原线程任务。

# 集合类

## 集合概述

数组：存储同一种数据类型的集合容器（**Object类型数组可存储任意类型数据**）

数组一旦初始化，长度就固定；数组元素之间的内存地址是连续的。

集合：存储对象类型数据的数据

1. 集合可以**存储任意类型数据**
2. **长度可变**

## Collection分类

**Collection（单列集合的根接口）**

1. List（Collection子接口）：**有序（按照添加顺序），可重复**
2. Set（Collection子接口）：**无序，不可重复**

## Collection方法

1. **增**

boolean add(E e) : 添加元素

boolean addAll(Collection<? extends E> c)： 把一个集合添加到另外一个集合

1. **删**

void [clear](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html#clear())(): 清空集合

boolean remove(Object o)；删除指定元素，不存在则返回false

boolean removeAll(Collection<?> c): 删除集合中与c的交集元素  
[retainAll](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html#retainAll(java.util.Collection))([Collection](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html)<?> c): 保留集合中与c的交集元素，其余删除

1. **查**

int [size](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html#size())(): 返回集合元素个数

1. **判断**

boolean isEmpty(): 判断集合是否为空

boolean contains(Object o): 判断集合中是否存在元素o（依赖于实参的equals方法）

boolean [containsAll](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html#containsAll(java.util.Collection))([Collection](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html)<?> c): 判断集合中是否存在集合c的所有元素

boolean equals(Object o): 判断集合与指定对象o是否相等。

1. **迭代**

Object[] toArray()：返回包含此集合中所有元素的数组（Object类型数组）。返回数组的运行时类型与指定数组的运行时类型相同。

**Iterator<E> iterator()**：返回在对集合元素进行迭代的迭代器（**Iterator<E>为接口类型，但是实际是接口的实现类对象**）。（抓取集合中的元素）

迭代器方法

boolean hasNext(): 判断是否有元素可以遍历

next(): 返回迭代的下一个元素

void remove(): 移除迭代器返回的**最后一个**元素

Collection c = new ArrayList();

//用迭代器遍历集合元素

**Iterator it = c.iterator(); //返回迭代器**

**while (it.hasNext()){ //判断是否有元素可以遍历（当前指针是否有指向元素）**

**it.next(); //获取当前指针指向的元素并返回，然后指针向下移动一个单位**

**it.remove(); //删除当前遍历到的元素**

**}**

## List 接口的特有方法

List 接口的特有方法都与索引值有关

1. 添加

void add(int index, E element); //把元素插入到指定位置

void addAll(int index, Collection<? extends E> c); //把集合c元素插入到指定位置

1. 获取

E get(int index); //获取指定集合中的元素，返回声明类型

int indexOf(Object o); //找出指定元素**第一次**在集合中的索引

int lastIndexOf(Object o); //找出指定元素**最后一次**在集合中的索引

**List<E>** subList(int fromIndex, int toIndex); //根据索引区间获取子列表

1. 修改

set(int index, E element); //修改指定索引值位置元素

1. 迭代

ListIterator<E> listIterator(); //**返回ListIterator迭代器**

**ListIterator迭代器的超级接口为Iterator**

**特有方法（这三个方法不能同时在一个循环中使用）：**

add(E e); //**把元素插入到当前指针指向的位置**

set(E e); //**替代迭代器最后一次返回的元素**

remove(); //**删除迭代器最后一次返回的元素**

hasPrevious(); //判断是否有上一个元素

previous(); //指针向上移动一个单位，然后再取出当前指针指向的元素

**注：这里与next()方法不同，next()是先取值然后再向下移动指针**

**迭代器遍历的注意事项**

迭代器迭代过程中，不允许对通过list的方法对改变集合的元素个数，需要改变可用**ListIterator**的方法（add、remove）

## List接口实现类

**常用List接口实现类（ArrayList、LinkedList、Vector）**

## ArrayList

ArrayList特有方法

ensureCapacity(int minCapacity)：**增加ArrayList的容量**，以确保它至少能够容纳最小容量参数所指定的元素数。

trimToSize(): 将ArrayList对象的容量调整为列表的当前大小（一般不用）

**ArrayList使用特点及原理**

（1）使用ArrayList无参构造函数创建对象时，默认容量是多少？

ArrayList**底层维护一个object数组**，使用无参构造函数时，Object数组的默认长度是**10**，当长度不够时，默认增长**0.5倍**

（2）数组的地址是连续的，所以**查询速度快**（指针偏移 ），**增删速度慢**（添加时，当容量不够时，需要将旧数组内容拷贝到新数组；删除时：需要将删除位置后的元素向前移动）

## LinkedListed

1. 底层使用**链表**数据结构实现的，链表的内存地址是**不连续**的，则特点是**查询速度快、增删快**
2. 链表数据存储分为两部分：一部分为当前元素，另一部分存下一元素的内存地址
3. 查询需要挨个遍历
4. 增删只需要改变指向的内存地址就可以

**LinkedListed常用方法**

（1）方法介绍

addFirst(E e); //**添加元素至首位**

addLast(E e); //**添加元素至末位**

getFirst(); //**获取首位元素**

getLast(); //**获取末位元素**

removeFirst(); //**删除首位元素，返回删除后的集合**

removeLast(); //**删除末位元素**

**如果集合中没有元素，获取或者删除元**

**素抛：NoSuchElementException**

（2）数据结构

**栈：**先进后出（模拟堆栈数据结构的存储方式）

push(E e): 将元素插入集合的末位

pop():**移除并返回集合的第一个元素（**从此列表所表示的堆栈处弹出一个元素**）**

**队列：**先进先出（模拟 队列数据结构的存储方式）

offer(E e): 将元素插入集合的末位

poll(): **获取并移除此列表的头（第一个元素）**

（2）返回逆序的迭代器对象

**descendingIterator()** 返回逆序的迭代器对象（Iterator）

# 集合类

## Vector

Vector：底层维护Object数组，是**线程安全**的，线程安全意味着操作效率低。

ArrayList与Vector的区别：

二者底层都是通过Object数组实现的

不同点是：ArrayList是**线程不同步**的，操作效率高，Vector是**线程同步**的，操作效率低；ArrayList是**JDK1.2**出现的，操作效率高，Vector是**JDK1.0** 出现的。

## Set接口实现类

Set接口实现类（**元素无序、不可重复）HashSet TreeSet**

## HashSet

**HashSet：底层使用Hash表支持，存取速度快**

**实现原理：**

向HashSet添加元素时，首先调用元素的**HashCode**方法，得到元素的Hash值，并经过移位等运算等，计算出该元素在哈希表中的存储位置。

情况一：计算出该元素存储位置**没有元素**，则直接存储到该位置

情况二：计算出该元素存储位置已经存在元素，则调用该元素的equals方法，若返回为true，则认为是同一个元素，不再添加入HashSet

因此：当HashSet元素类型为自定义类对象时，如果需要根据某个属性判断对象相等，一般需要重写equals方法和hashCode方法

Hash表的特点：**桶式结构**，一个坑对应多个萝卜

**注：HashCode默认表示内存地址，String类重写了HashCode方法：当两个字符串内容一致时，返回的Hash码相同**

HashSet的add可返回boolean，判断元素能不能加入；

## TreeSet

1. 底层使用**二叉树**数据结构实现，存储规则：**左小右大**

**构建二叉树时，当连续三个节点没有分叉时，则自动调整树结构。**

<https://blog.csdn.net/lsr40/article/details/85230703>

1. 若元素具备自然顺序特性，则**自动按照自然顺序特性进行单例存储**
2. 若元素不具备具备自然顺序特性，则元素所所属类**须实现Comparable接口**，将比较规则定义在**compareTo方法**
3. 向TressSet添加元素时，**若compareTo()方法返回值为0，则认为是重复元素，不再添加**
4. 若元素不具备具备自然顺序特性，且元素所所属类**没有实现Comparable接口**时，则需要在创建TreeSet时传入一个比较器（推荐使用该方法）

**比较器定义：**

**class 类名 implements Comparator{**

**//重写compare方法，写入比较规则**

**}**

TreeSet aa = new TreeSet(comparator)

1. 若元素不具备具备自然顺序特性，且元素所所属类**有实现Comparable接口**时，同时在创建TreeSet时传入一个比较器时，优先使用比较器的规则
2. **TreeSet可以实现对字符串排序，原因：字符串已经实现Comparable接口，比较规则：**

* **对应位置有字符，则比较字符**
* **对应位置字符相同，则比较字符串长度**

## 泛型

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

约定集合的元素类型，其他类型不能添加。

**右边的String可以省略**

自定义泛型：一个**数据类型的占位符**或**数据类型的变量**

## 方法上自定义泛型

修饰符 <**声明自定义的泛型>**返回值类型 函数名(**使用自定义泛型)**{

}

public static <**T**>**T** getData(**T** o)

（1）在泛型中不允许使用基本数据类型，需要使用基本数据类型时，则**使用基本数据类型对应的包装类型**。

byte----Byte

short----Short

int----Integer

long----Long

double----Double

float---Float

boolean----Boolean

char----Character

（2）在方法上自定义泛型时，自定义泛型的数据类型在调用该方法时**根据传入的实参数据类型确定**

## 泛型类（在类上自定义泛型）

class 类名<声明自定义泛型>{

}

**泛型类具体数据类型在创建类型对象时确定**，不指定时为Object类型

**类名<Integer> tool = new 类名<Integer>();**

在类上自定义的泛型**不能作用于静态方法**，若静态方法需要使用自定义泛型时，**需要在自己方法上定义使用，**实际数据类型在调用该方法时**根据传入的实参数据类型确定**

## 泛型接口

**interface 接口名<声明自定义泛型>{**

**}**

1、接口上自定义泛型的**具体数据类型在实现接口时指定，不指定时默认Object**

class 实现类名 implements 接口名<**String**>{

}

2、实现类时不确定类型时，可以指定为自定义泛型，在创建对象时确定实际数据类型（类名和接口名后都需要自定义泛型）

class 实现类名<**T**> implements 接口名<**T**>{

}

## 泛型上下限

通配符：？

print (**Collection<? super Integer> c**) 限定集合的自定义泛型只能是Integer或父类元素，限定泛型的下限

print (**Collection<? extends Number>**) 限定集合的自定义泛型只能是Number或子类元素，限定泛型的上限

# 双例集合

## 根接口Map

**双例集合：存储键值对（Key-Value）**

根接口：Map：键不允许重复

**实现类：HashMap TreeMap HashTable**

声明：

**Map<key类型,value类型> map = new HashMap< key类型,value类型>**

## Map接口方法

1. 添加

* **put(key, value)**：添加键值对，如果存在该键，则返回键对应的值，否则返回null
* **putaAll(map2)**：添加整个Map集合

注：null也可以作为键或值

1. 查询

* **get(key)**：根据指定的键获取对应的值
* **size()**：获取Map中的键值对个数

1. 判断

* containsKey(key)：判断Map集合是否包含指定的键
* containsValue(value)：判断Map集合是否包含指定的值
* isEmpty()：判断Map集合是否为空

1. 删除

* **remove(key)**：根据键删除对应数据，并返回删除的数据
* **clear()**：清空Map集合中的所有数据

1. 迭代

* **keySet()**方法把Map集合的键转为Set，但是不能获取值

Set<String> keys = **map.keySet()**;

Iterator<String> it = keys.iterator();

while(it.hasNext()){

String key = it.next();

System.out.println("键："+ key+" 值："+ **map.get(key)**);

}

* **values()**方法把Map集合的值转为Collection，但是不能获取键

Collection<String> c = **map.values**();

Iterator<String> it = c.iterator();

while(it.hasNext()){

System.out.println("值："+ it.next());

}

* **entrySet方法遍历（推荐）**

Set<**Map.Entry<String,String>**> entrys = **map.entrySet**();

Iterator<Map.Entry<String,String>> it = entrys.iterator();

while(it.hasNext()){

Map.Entry<String,String> entry = it.next();

System.out.println("键："+ **entry.getKey()+**" 值："+ **entry.getValue**());

}

## HashMap（底层数据结构是重点）

（1）存储原理：向HashMap中添加元素时，首先调用键的hashCode方法确定哈希码值，并经过运算计算元素在哈希表中的存储位置：

* 存储位置为空时：直接存入
* 存储位置不为空时，调用equals方法比较两个键，返回为false时则存储，否则不存储。

（2）**出现重复键时，先前存储键值对会被替代**

## TreeMap

1. 存储原理：底层基于**红黑树**数据结构实现，会对**键**进行排序
2. 键不具备自然顺序时，键所属类需实现Comparable接口或在创建TreeMap对象时传入自定义比较器（Comparator）

## HashTable（了解）

1. 底层基于哈希表实现
2. 线程安全，但操作效率低，目前已经被HashMap取代

## 集合工具类

* Collections

与Collection区别？

Collection是**单例集合的根接口**

Collections是**操作对象集合的工具类**

**Collections的常用方法：**

|  |
| --- |
| 1. 对list进行二分查找：   前提该集合一定要**有序**。  **int binarySearch(list,key);**  //必须根据元素自然顺序对列表进行升级排序  //要求list 集合中的元素都是Comparable 的子类。  int binarySearch(list,key,Comparator);  2，对list集合进行排序。  **sort(list);**  //对list进行排序,其实使用的事list容器中的对象的compareTo方法  **sort(list,comaprator); //针对不具有自然顺序的元素**  //按照指定比较器进行排序  3，对集合取最大值或者最小值。  max(Collection)  max(Collection,comparator)  min(Collection)  min(Collection,comparator) 4，对list集合进行反转。  **reverse(list)**;  5，对比较方式进行强行逆转。  Comparator reverseOrder();  Comparator reverseOrder(Comparator);  6，对list集合中的元素进行位置的置换。  swap(list,x,y);  7，对list集合进行元素的替换。如果被替换的元素不存在，那么原集合不变。  replaceAll(list,old,new);  8，可以将不同步的集合变成同步的集合。  创建线程安全集合  Set synchronizedSet(Set<T> s)  Map synchronizedMap(Map<K,V> m)  List synchronizedList(List<T> list)  9. 如果想要将集合变数组：  可以使用Collection 中的toArray 方法。注意：是Collection不是Collections工具类  传入指定的类型数组即可，该数组的长度最好为集合的size。 |

## 数组工具类

**Arrays:用于对数组操作的工具类**

**常用方法**

|  |
| --- |
| 1，二分查找,数组需要有序  binarySearch(int[])  binarySearch(double[])  2，数组排序  sort(int[])  sort(char[])……   1. 将数组变成字符串。   toString(int[])   1. 复制数组。  copyOf(原数组，新数组长度（元素不足用0或null补足）); 2. 复制部分数组。   copyOfRange(原数组, 范围索引1，范围索引2):   1. 比较两个数组（对应位置元素）是否相同。   equals(int[],int[]);   1. 将数组变成集合。   **List asList(T[]);**  这样可以通过集合的操作来操作数组中元素，  但是**不可以使用增删方法，add，remove**。因为数组长度是固定的，会出现  UnsupportOperationExcetion。  **可以使用的方法：contains，indexOf**。。。  如果数组中存入的基本数据类型，那么asList会将数组实体作为集合中的元素。  如果数组中的存入的引用数据类型，那么asList会将数组中的元素作为集合中  的元素。 |

# 正则表达式

**正则表达式：**用于操作字符串的规则

## 预定义字符

|  |  |
| --- | --- |
| **.** | 任何字符（与行结束符可能匹配也可能不匹配） |
| **\d** | 数字：[0-9] |
| \D | 非数字： [^0-9] |
| \s | 空白字符：[ \t\n\x0B\f\r] |
| \S | 非空白字符：[^\s] |
| **\w** | 单词字符：[**a-zA-Z\_0-9**] |
| \W | 非单词字符：[^\w] |

注：不加数量词，预定义字符只能匹配一个字符

## 数量词

|  |  |
| --- | --- |
| X? | X，**一次或一次也没有** |
| X\* | X，零次或多次 |
| X+ | X，一次或多次 |
| X{n} | X，恰好n次 |
| X{n,} | X，至少n次 |
| X{n,m} | X，至少n次，但是不超过m次 |

## 范围词

|  |  |
| --- | --- |
| [abc] | a、b 或 c（简单类） |
| [^abc] | 任何字符，除了 a、b 或 c（否定） |
| [a-zA-Z] | a 到 z 或 A 到 Z，**两头的字母**包括在内（范围） |
| [a-d[m-p]] | a 到 d 或 m 到 p：[**a-dm-p**]（并集） |
| [a-z&&[def]] | d、e 或 f（交集） |
| [a-z&&[^bc]] | a 到 z，除了 b 和 c：[**ad-z**]（减去） |
| [a-z&&[^m-p]] | a 到 z，而非 m 到 p：[**a-lq-z**]（减去） |

**注：不加数量词，范围词只能匹配一个字符**

## 正则表达式的主要操作

1. **匹配**

matches(String regex)方法

1. **切割**

split(String regex)方法

(.)：分组语法，**目的是为了让正则内容复用性**，组号从1开始

((A)B(C)))：第一组：((A)B(C)))；第二组：(A)；第三组：(B(C))；

第四组：(C)

\1：引用第一组匹配的内容

1. **替换**

replaceAll(String regex, String replacement)方法

s.replaceAll("(.)\\1+","$1"); //正则表达式外引用内容时用：$组号

1. **查找**

需要使用的对象：

1. Pattern（正则对象）
2. Matcher（**匹配器对象**）

常用方法：

find()：通知匹配器去匹配字符串，查找符合规则的字符串，有则返回true

group(): 获取负荷规则的字符串（调用group之前需要调用find方法）

**一次只能获取一个，获取全部时需要用循环**

指定为字符串的正则表达式必须首先**创建Pattern类**的实例。然后，可将得到的模式用于**创建**[**Matcher**](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Matcher.html)**对象**，依照正则表达式，**该对象可以与任意**[**字符序列**](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html)**匹配**。执行匹配所涉及的**所有状态都驻留在匹配器**中，所以多个匹配器可以共享同一模式。

因此，典型的调用顺序是

Pattern p = Pattern.[compile](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html#compile(java.lang.String))("a\*b");

Matcher m = p.[matcher](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html#matcher(java.lang.CharSequence))("aaaaab");

boolean b = m.[matches](mk:@MSITStore:C:\Users\王嗣鑫\Desktop\JAVA\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Matcher.html#matches())();

查找单词时，一般需要**单词边界（空格、逗号都可以作为边界）匹配器**

\b：单词边界匹配器只是**表示单词的结束部分**，不匹配任何字符。

# JDK1.5新特性

## 静态导入

import **static** java.util.Collections.sort 导入Collections的sort方法

import **static** java.util.Collections.\* 导入Collections的所有方法

可以避免重复在代码中使用类名

静态导入成员方法与本类成员同名时，**默认使用本类成员**，即静态导入失效

## 增强For循环

作用：简化迭代器的书写格式

适用范围：实现了**Iterable接口的对象**或**数组对象，底层使用迭代器方式**

语法：

for (元素类型 元素变量：集合或素组){

XXX

}

for (String item: set){

XXX

}

迭代器遍历与增强for循环的区别：迭代器遍历过程可以进行删除、修改等，增强for循环不可以

普通for循环与增强for循环的区别：普通for循环可以没有遍历的目标，增强for循环必须有

Map集合不能直接使用增强for循环（没有实现Iterable接口），需要使用时，要通过**entrySet**方法转为set。

Set<**Map.Entry<String,String>**> entrys = **map.entrySet**();

自定义类要使用挣增强for循环，需要实现Iterable接口

## 可变参数

用于函数形参，调用该方法是可传入不定长参数

格式：数据类型… 变量名，实际是一个**数组**

**可变参数必须位于形参中的最后一个，因此一个函数最多有一个可变参数**

void add(**int… arr**){

XXXX

}

**调用：add(1,2,3); add(1,2,3,4,5);**

## 自动装箱与拆箱

（1）自动装箱：把**基本数据类型数据**转换成**对象类型（引用类型）数据**

* Integer b = 10; //自动装箱
* Integer b = new Integer(10); //非自动装箱

（2）自动拆箱：把**对象类型（引用类型）数据**转换成**基本数据类型**数据

* Integer b = 10; //自动装箱
* Int a = b; //自动拆箱
* Int a = b.intValue(); //非自动拆箱

（3）

Integer e = 127;

Integer f = 127;

是同一个对象么：e == f true

Integer e = 128;

Integer f = 128;

是同一个对象么：e == f false

原因：**Integer内部维护了一个缓存数组，数据范围是-128-127，如果数据在该范围内，则直接取出，否则创建新的对象**。

## 枚举

某些方法接收的数据在**固定范围内**（可枚举）。

方案一：需要自定义类，私有化构造方法，然后创建本类对象供外部使用。

class Gender{

String value;

public static final Gender man = new Gender(“男”);

public static final Gender woman = new Gender(“女”);

private Gender(String value){

this.value = value;

}

}

方案二：使用**自定义枚举类**

enum Sex{

woman, man; //枚举值

}

枚举值相当于：

**public static final Sex man = Sex();**

**public static final Sex woman = Sex();**

* 枚举值默认修饰符：**public static final**
* 枚举值的数据类型为所属**类**，枚举值指向本类对象
* 枚举类构造方法为private
* 枚举类可以定义成员变量和成员方法
* 枚举类可自定义构造函数，但必须私有
* 枚举类可以存在抽象方法，但是枚举值须实现抽象方法
* 枚举值必须位于枚举类的第一个语句
* switch用于枚举类时，case后直接用枚举值

枚举类使用场景：方向、星期、性别、季度

# IO技术

## File类

IO技术主要解决**设备之间的数据传输问题**，如硬盘—内存，键盘数据—内存

File类用于描述一个文件或文件夹，可以读取文件或文件夹属性信息。

（1）构造方法

* **File(String pathname)：指定文件或文件夹路径创建一个File文件**

File file = new File(“XXXXXXXXXXX”);

* File(File parent, String name)：parent表示父路径创建的File对象，name表示子路径（适用操作同一文件夹下多个文件）
* File(String parent, String name)：parent表示父路径创建的路径（字符串），name表示子路径

（2）目录分隔符

* windows: \\
* linux: /

File.separator：目录分隔符，可以自动识别系统

（3）路径

* 绝对路径：完整路径
* 相对路径：.当前路径 ..上一级路径 ./xxx/下一级路径（前提：在同一个硬盘）

file.getAbsolutePath()：获取绝对路径

（4）常用方法

* 创建：

createNewFile()：在指定位置创建空文件，若已存在则不创建，创建成功返回true

mkdir()：创建**单级**文件夹，创建成功返回true

mkdirs()：创建**多级**文件夹，创建成功返回true

renameTo(File xxx)：重命名或剪切（不在同一文件夹下时），**剪切不能操作文件夹（目标文件存在时则不操作）**

* 删除：

delete(): 删除文件或删除空文件夹，不能删除非空

deleteOnExit(): **JVM退出**时删除文件，一般用于删除临时文件、

* 判断：

exit(): 判断文件是否存在

isFile(): 判断是否是一个文件

isDirectory(): 判断是否是一个文件夹

isHidden(): 判断是否是隐藏文件或目录

isAbsolute(): 判断是否是绝对路径

* 获取：

getName(): 获取文件或文件夹的名称，不包含上级路径（**不管存不存在都会返回**）。

getPath(): 回绝对路径，可以是相对路径，但是目录要指定（**传入路径与传出路径相同**）

getAbsolutePath(): 获取文件的绝对路径，与文件是否存在没关系

length(): 获取文件的大小（字节数），如果文件不存在则返回0L，如果是文件夹也返回0L。

getParent(): 返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回null，**传入相对路径则返回相对路径**。

lastModified(): 获取**最后一次被修改的时间，返回值为long，可以先转为Date，然后用SimpleDateFormat转为年月日时分秒**。

File[] listRoots(): 列出所有的**根目录**（Windows中就是所有系统的盘符），返回数组

**String[]** list(): 返回当前路径下的文件或者文件名，包含**隐藏文件**，返回字符串。**对于文件这样操作会返回null**。

**String[]** list(FilenameFilter filter) 返回指定当前目录中**符合过滤条件的子文件或子目录**。返回字符串数组，对于文件这样操作会返回null。

File[] llistFiles()：返回目录下的文件或者目录对象（**File类实例**），包含隐藏文件。对于文件这样操作会返回null。

File[] listFiles(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录，返回File数组。对于文件这样操作会返回null。（**FilenameFilter：文件名过滤器**）

## FileInputStream

（1）IO流分类：

* 按照数据流向划分：输入流与输出流
* 按照处理单位划分：字节流（读取二进制数据），字符流（字节流+解码）

（2）输入字节流：

InputStream 所有输入字节流的基类（抽象类）

FileInputStream 读取文件数据的输入字节流

（3）使用FileInputStream读取文件步骤：

* 找到数据文件：创建File对象
* 建立数据输入通道：创建FileInputStream对象
* 读取文件中的数据：调用FileInputStream对象的read()方法
* 默认一次读取一个字节数据
* 使用循环读取：read()方法读取到文件末尾时，返回-1，可以作为读取文件全部内容的循环结束条件。
* 使用缓冲字节数组读取：read(buf)，传入一个字节数组，返回值为读取到的字节长度，也可以指定读取文件长度：read(buf,0,len)（缺点是不能读取全部文件）
* 使用缓冲数据配合循环读取：read(buf,0,len)读取到文件末尾时，返回-1

while (length = fileInputStream.read(buf) != -1){

xxxx

sout(new String(buf,0,length)) //每次循环，buf直接被覆盖原内容，在内容末不够buf长度时，不加length会**打印出前一次循环的部分内容**。

}

这种方法效率更高，缓冲//大小一般是**1024**的倍数

* 关闭数据通道：调用FileInputStream对象的close()方法（不关闭其他程序不能操作文件）

## FileOutputStream

（1）输出字节流：

OutputStream 所有输出字节流的基类（抽象类）

FileOutputStream文件数据写入的输出字节流

（2）使用FileOutputStream读取文件步骤：

* 找到数据文件：创建File对象
* 建立数据输入通道：创建FileOutputStream对象
* 向文件写入数据：调用FileOutputStream对象的write()方法
* 默认每次写入一个字节数据
* 使用字节数组把数据写入：String转为byte数组传入write()方法

write可以指定写入byte数组的索引区间

* 关闭数据通道：调用FileOutputStream对象的close()方法

（3）注意事项

* 使用FileOutputStream创建对象时，若目标文件不存在，则会创建，若目标文件存在会默认清空原文件内容
* 需要追加内容时，创建对象时传入（File file, boolean true）。
* 写入Int（32位）数据类型时，只会取前8位，高位切掉

## 异常处理

针对IOException，一般不采用抛出的方式（调用不方便），而采用处理的方式：

1. 阻止后面的代码执行
2. 把IOException传递给RuntimeException（调用者可以不处理），然后抛出

throw new RuntimeException(IOException e); // 创建RuntimeException对象可以传入异常

1. finally关闭文件

try{

XXX

}catch (IOException e) {

throw new RuntimeException(e);

}finally{ //**一层不够可以嵌套一层**

try {

if (fileInputStream != null) {

fileInputStream.close();

System.out.println("关闭资源成功...");

}

} catch (IOException e) {

System.out.println("关闭资源失败...");

throw new RuntimeException(e);

}

}

## 缓冲输入字节流BufferedInputStream

作用：提高读取文件的效率，减少与文件的交互次数

内部维护长度为8192（8KB）的字节数组，本质与缓冲字节数组方式读取文件是相同的。

BufferedInputStream没有文件读写能力，必须建立数据传输通道

（1）使用BufferedInputStream读取文件步骤：

* 找到数据文件：创建File对象
* 建立数据输入通道：创建FileInputStream对象
* 创建BufferedInputStream对象，创建时传入FileInputStream对象
* 调用read()方法获取文件内容（**一次只能读取一个字节**）
* 调用close()关闭文件

注：

* BufferedInputStream的close()实际上是在调用FileInputStream的方法；
* **read()一次性读取8KB数据到内部维护的字节数组**，然后用户调用时一次返回一个数组元素，到字节数组最后一个元素时，再从文件中取8KB内容，而FileInputStream的read()方法每次直接从文件读取字节。

## 缓冲输出字节流BufferedOutputStream

内部维护长度为8192（8KB）的字节数组，本质与缓冲字节数组方式写入文件是相同的。

（1）使用BufferedOutputStream写入文件步骤：

* 找到数据文件：创建File对象
* 建立数据输入通道：创建FileOutputStream对象
* 创建BufferedOutputStream对象，创建时传入FileOutputStream对象
* 调用**write()方法将内容写入内部维护的字节数组，**当调用flush()或close()或数组已满时，将字节数组内容写入文件中
* 调用close()方法关闭文件

## 输入、输出字符流

字节流不能输出中文，需要借助String的getBytes()方法对中文进行编码才可以，也不能输入中文，因为一个中文占**两个字节**。

**（1）码表**

ASCII：美国标准信息交换码。用一个字节的**7位**可以表示。

ISO8859-1：拉丁码表。欧洲码表，用一个字节的**8位**表示。

GB2312：中国的中文编码表。

**GBK**：中国的中文编码表升级，融合了更多的中文文字符号。

Unicode：国际标准码，融合了多种文字。所有文字都用两个字节来表示，Java语言使用的就是unicode。

UTF-8：**最多用三个字节来表示一个字符**。万国码，推行的。是1~3个字节不等长。英文存的是1个字节，中文存的是3个字节

UTF-16：中英文均是两个字符

（我们以后接触最多的是iso8859-1、gbk、utf-8）

**（2）编码**

字符串---》字节数组

String类的getBytes() 方法进行编码，将字符串，转为对映的二进制，并且这个方法可以指定编码表。如果没有指定码表，该方法会使用**操作系统默认码表**。

注意：中国大陆的Windows系统上默认的编码一般为**GBK**。在Java程序中可以使用System.getProperty("file.encoding")方式得到当前的默认编码。

**（3）解码**

字节数组---》字符串

String类的构造函数完成。

String(byte[] bytes) **使用系统默认码表**

String(byte[],charset)指定码表

注意：我们使用什么字符集（码表）进行编码，就应该使用什么字符集进行解码，否则很有可能出现乱码（兼容字符集不会）。

**（4）字符流**

字符流 = 字节流 + 编码（解码）：把读取的二进制进行编码和解码工作

输入字符流：

| Reader：输入字符流的基类（抽象类）

| FileReader：读取数据

基本流程：

* 找到目标文件：创建File对象
* 建立数据输入通道：创建FileReader对象，传入File对象
* 调用read()：每次读取一个字符（返回值为int）。
* 每次读取一个字符
* 借助char[] 数组，读取多个字符
* 关闭文件

输出字符流：

| Writer：输出字符流的基类（抽象类）

| FileWriter：写入数据

基本流程：

* 找到目标文件：创建File对象
* 建立数据输入通道：创建FileWriter对象，传入File对象
* 调用write()：可以每次传入一个字符串。
* FileWriter**内部维护了1KB的字符数组**，写入数据时会先写入内部维护的字符数组，如需真正写到文件，需要调用flush或close方法或内部数组已满
* 目标文件不存在会自动创建，目标文件已经存在时会默认清空，需要追加时，需要在创建对象时加true
* 关闭文件

使用输入、输出字符流复制图片会丢**失数据**的原因：

FileReader读到数据，但在码表中找不到对应字符时，会返回一个未知字符对应的数据，从而出现数据丢失的情况

**使用字符流的场景：读写字符数据（字母、汉字）**

**使用字节流的场景：读写数据不需要转换成字符的场景，如图片**

## 缓冲输入、输出字符流

缓冲输入字符流BufferedReader：提高读取文件效率，**扩展FileReader的扩展功能**

扩展功能：

readline(): 一次读取一行内容，返回字符串，读到最后返回null

缓冲输出字符流BufferedWriter

扩展功能：

newline()：换行符

## 装饰者设计模式

作用：增强类的功能，且装饰类之间可以互相装饰

装饰器模式：

* 使用分层对象来动态透明的向单个对象中添加责任（功能）。
* 装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口。
* 某些对象是可装饰的，可以通过将其他类包装在这个可装饰对象的四周，来将功能分层。
* 装饰器必须具有和他所装饰的对象相同的接口。

JavaIO中的应用：

* Java I/O类库需要多种不同的功能组合，所以使用了装饰器模式。
* FilterXxx类是JavaIO提供的装饰器基类，即我们要想实现一个新的装饰器，就要继承这些类。

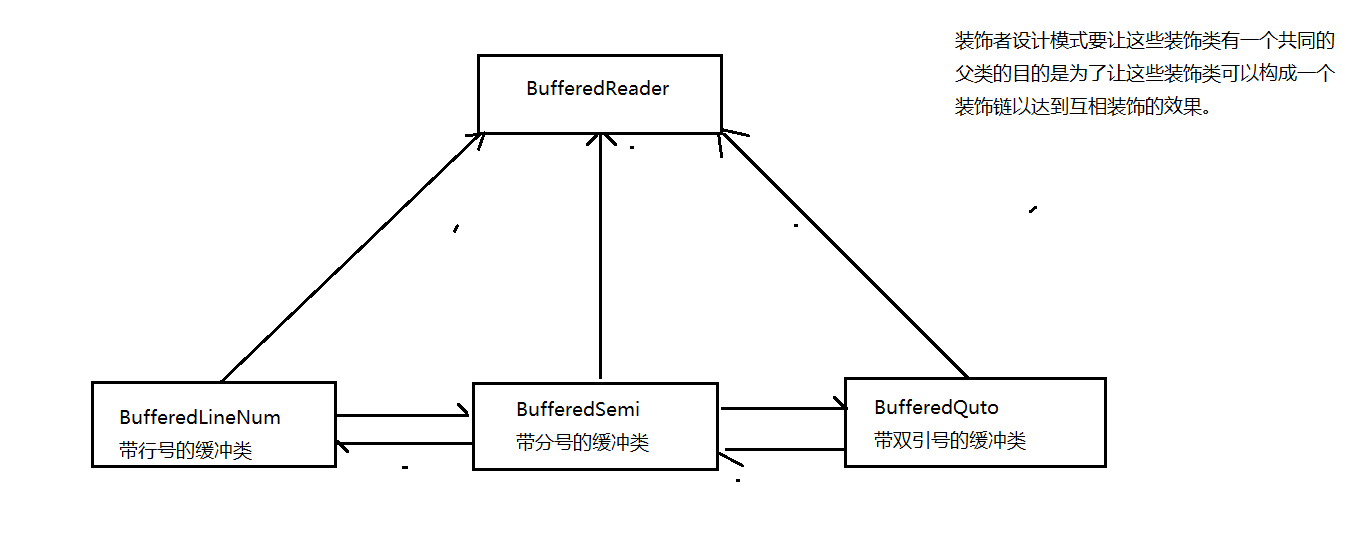
装饰器与继承：

（1）继承实现的增强类：

* 优点：代码结构清晰，而且实现简单
* 缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

（2）修饰模式实现的增强类：

* 优点：内部可以通过多态技术对多个需要增强的类进行增强
* 缺点：需要内部通过多态技术维护需要增强的类的实例。进而使得代码稍微复杂。



## SequenceInputStream（序列流）

序列流，对多个流进行合并。

SequenceInputStream 表示其他输入流的**逻辑串联**。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

构造函数：

SequenceInputStream(InputStream s1, InputStream s2)

多个文件合并：

将多个输入流打包为Vector

获取迭代器：Enumeration<FileInputStream> e = vector.**elements**();

## 对象的输入输出流（序列化与反序列化）

作用：通过文件读写对象的信息，保证对象的持久化存储

对象的输出流：ObjectOutputStream

对象输出（序列化）的基本流程：

1. 创建File对象
2. 创建FileOutputStream对象
3. 创建对象输出流对象：new ObjectOutputStream(FileOutputStream f)
4. 调用**writeObject()** 方法写出（对象所属类需要实现**Serializable**接口（标志接口，无方法））：这个过程称为**序列化（读取文件的对象信息称为反序列化）**
5. 调用close()方法关闭资源

对象的输入流：ObjectInputStream

1. 创建File对象
2. 创建FileInputStream对象
3. 创建对象输出流对象：new ObjectInputStream(FileInputStream f)
4. 调用**readObject()** 方法写入（返回Object对象，需要进行强转，**此创建对象过程不要要调用构造方法**）
5. 调用close()方法关闭资源

注：

* 反序列化时，ObjectInputStream会先读取文件中的serialVersionUID（**根据类名、成员、包名、工程名计算出**），然后与本地class文件的serialVersionUID进行对比，如果编码不一致，则反序列化失败
* 考虑到序列化之后会修改类成员，则需要在序列化之前**指定**类的serialVersionUID，则系统不会自动生成serialVersionUID：private static final long serialVersionUID = 1L
* 若某个对象的成员不想被序列化，可使用关键字**transient**修饰（透明）
* 一个对象内部成员变量包含另一个类（内部维护另一个的引用），要实现反序列化，另一个类也需要实现**Serializable**接口

## Properties（配置文件类）

作用：生成与读取配置信息

HashTable的子类，实现Map接口（添加键值对：**setProperty(key,value)**）

生成配置文件（文件名后缀properties）

p.**store**(FileOutputStream f, String s): s：文件描述

p.**store**(FileWriter w, String s): 支持中文

读取配置文件信息

p.**load(**FileReader r**);**

## 打印流printStream

可以打印**任意类型的数据**，打印数据之前会**把数据转换为字符串**

创建：

new printStream(File f);

new printStream(FileOutputStream f);

方法：

println() print()：把括号内信息转换为字符串，打印对象会调用对象的toString()方法

异常日志信息收集

e.printStackTrace(printStream p)

## 编码与转码

字符串的编码：getBytes()：默认系统编码表，可在括号内指定

字符串编码：new String(byte[] b)：默认系统编码表，可在括号内指定

注意：

* 编码与解码应该是一套码表
* 如果指定码表为unicode，实际上使用的是utf-16（一个字符两个字节），特殊之处：utf-16编码的前两个字节是-2和-1（**标志位**）

## 转换流

（1）输入字节流转换为字符流：

**InputStreamReader()**

创建：

new InputStreamReader(InputStream in)：返回

（2）输出字节流转为字符流

OutputStreamWriter

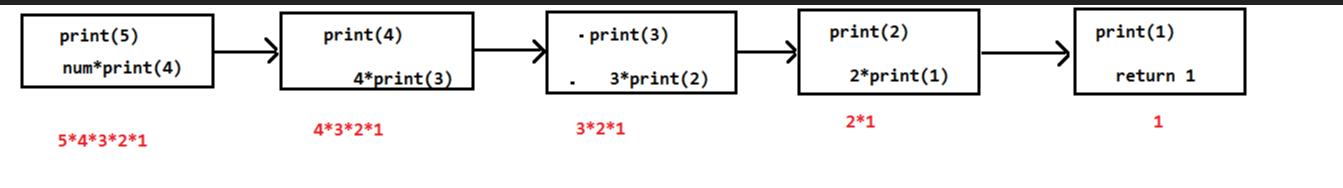
创建：

new OutputStreamWriter(OutStream o)

注：可**指定编码表**进行读写文件

## 递归

函数自己调用自己：必须要特定条件下调用

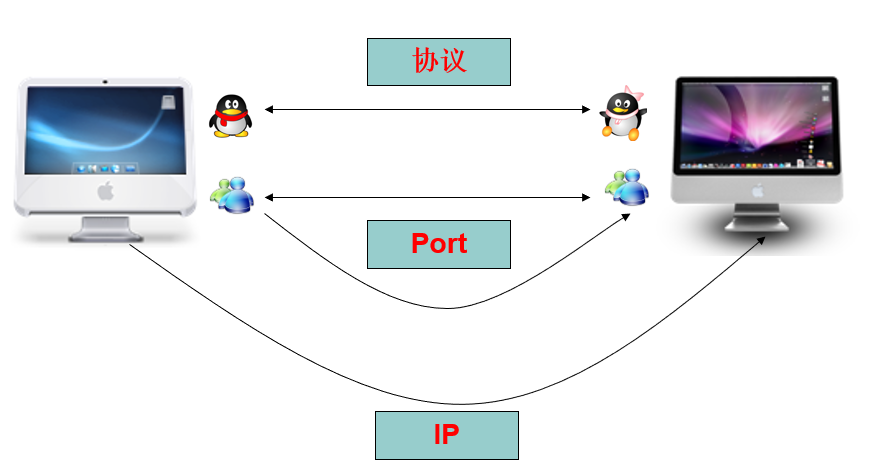


# 网络编程

解决计算机之间的数据传输问题

网络通信三要素

* IP地址
* 端口号：标识消息发给哪个应用程序（0-65535）
* 公认端口（WellKnownPorts）：**从0到1023，它们紧密绑定（binding）于一些服务**。
* 注册端口（RegisteredPorts）：**从1024到49151**。它们松散地绑定于一些服务（**可用**）。
* 动态和/或私有端口（Dynamicand/orPrivatePorts）：**从49152到65535**。
* 常用端口：**21：FTP；80：HTTP；443：HTTPS**
* 通信协议



## IP地址

IP地址：**由32位二进制数组成（每8位组成一个网端）：本机回环地址：127.0.0.1**

* A类地址：1个网络号+3个主机号 2^24 政府
* B类地址：1个网络号+3个主机号 2^16 事业单位
* C类地址：3个网络号+1个主机号 2^8 私人使用

子网掩码：255.255.255.0表明前三段为网络号，最后一段为主机号

InetAddress(IP类)：无构造方法，只能通过静态方法创建对象

常用方法：

* 获取本机信息
* getLocalHost()：获取本机IP地址对象（返回InetAddress类对象）
* getHostAddress()：获取**IP地址**（返回字符串）
* getHostName()：获取IP地址**主机名**（返回字符串）
* 获取别人机器IP信息

getByName()：传入IP地址或主机名，返回IP地址对象

getAllByName()：传入域名字符串（“www.baidu.com”），返回IP地址数组

## 网络通信协议介绍

**（1）UDP**

**UDP： User Datagram Protocol**的简称，中文名是**用户数据包协议**，是 OSI 参考模型中一种**无连接的传输层协议**，提供**面向事务的简单不可靠信息传送服务**。

**特点：**

* 每个数据包（**数据包含数据、源、目的地址**）中都给出了完整的地址信息，因此**无需要建立发送方和接收方的连接（速度快）**。
* UDP传输数据时是有大小限制的，每个被传输的数据报必须限定在**64KB**之内。
* UDP是一个**不可靠的协议（丢包）**，发送方所发送的数据报并不一定以相同的次序到达接收方。

**（2）TCP**

**TCP**：**Transmission Control Protocol，**中文名是**传输控制协议**。TCP是一种**面向连接**（连接导向）的、**可靠**的、**基于字节流的运输**层（Transport layer）通信协议。

**特点：**

* 面向连接的协议，数据传输必须要**建立连接**，所以在TCP中需要连接时间。
* 传输数据大小限制，一旦连接建立，双方可以**按统一的格式传输大的数据**。
* 通过三次握手机制连接，保证可靠的协议，**确保接收方完全正确地获取发送方所发送的全部数据**。

## UDP协议下的socket

Java中的网络通信称为socket（插座）通信，要求通信机器安装socket

**DatagramSocket（UDP插座服务）：使用发送端与接收端**

**DatagramPacket（数据包类）**

**发送端使用步骤：**

* 创建DatagramSocket对象：无参构造
* 创建DatagramPacket对象：**传参：buf（byte数组）：数据包内容；length：数据包长度；address：IP地址；port：端口号**
* 调用UDP服务发送数据包：send
* 关闭资源：释放端口号

**接口端使用步骤：**

* 创建DatagramSocket对象：传入**端口号**（用作监听）
* 创建DatagramPacket对象（准备空数据包用于存储数据）：**传参：buf：空数据包；length：数据包长度**
* 调用UDP服务接收数据：receive（未收到数据阻塞）
* 调用UDP服务发送数据包：send，通过数据包获取字节长度
* 关闭资源

UDP给FeiQ发送消息

每个网络程序都**有自己所处理的特定格式数据**,如果接收到的数据不符合指定的格式，那么就会被当成垃圾数据丢弃。(加密..)

飞Q接收的数据格式：

**version:time :sender : ip: flag:content ;**

版本号 时间 发送人 :IP： 发送的标识符(32): 真正的内容;、

广播地址：

在udp协议中，有一个IP地址称作为*广播地址*，广播地址就是**主机号为255地址**。

**给广播IP地址发送消息的时候，在同一个网络段的机器都可以接收到信息。**

//192.168.15.255

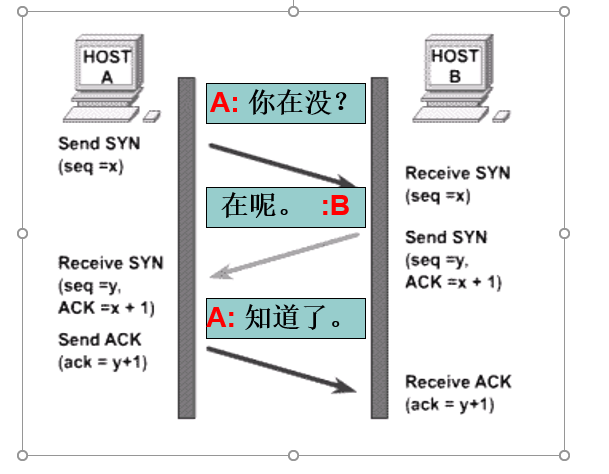
UDP数据丢失的情况：

1. 带宽不够
2. CPU的处理能力不足（**寄存器可暂存1KB**）

## TCP协议下的socket

基于**IO流**进行数据传输，面向连接，通过**三次握手**保证数据的完整性

保证A到B是通的，B到A是通的



（1）Socket（客户端）

TCP客户端一旦启动马上与服务端通信

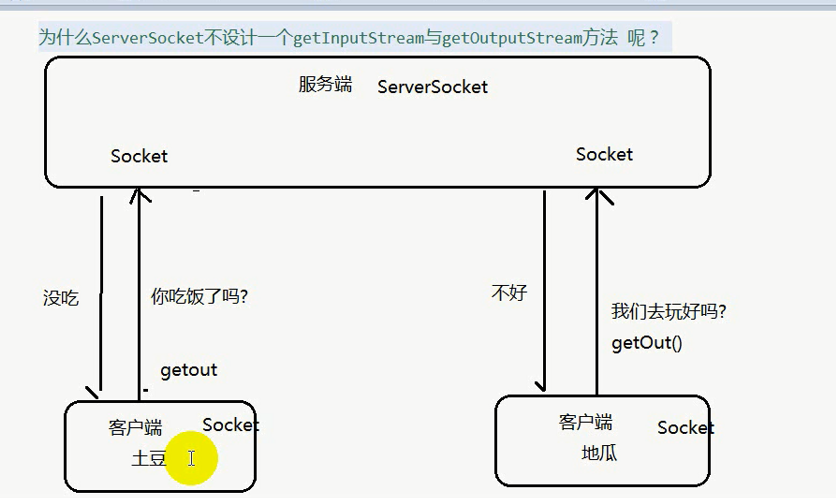
使用步骤：

* 建立TCP客户端服务：**new Socket(InetAddress ip, int port)**;
* 获取Socket的输出流对象：调用getOutputStream()，返回OutputStream对象
* 写数据：调用OutputStream对象的write方法
* 关闭资源：关闭Socket对象

（2）ServerSocket（服务端）

* 建立TCP服务端服务：**new ServerSocket(int port) //监听端口**;
* 接受客户端连接：调用accept()方法，返回Socket对象（阻塞型方法，未收到一直等待）
* 调用Socket对象的getInputStream()方法，返回InputStream对象
* 调用InputStream对象的read()方法读取客户端发送的内容

**ServerSocket不设置输入输出流的原因：存在多个客户端，通过accept方法实现与客户端一一对应，**



# 图形化界面

## 概述

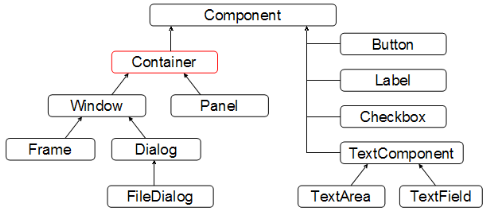
java.awt使用的图形类（称为组件类）依赖系统的图形库

java.swing使用的图形类均是sun自己实现

GUI API包含的类分为三个部分:**组件类**(component class) **容器类**(container class),和**辅助类**(helper class)，**组件类是用来创建用户图形界面的**,例如JButton,JLabel,JTextField。**容器类是用来包含其他组件的**,例如JFrame,JPanel，**辅助类是用来支持GUI组件的**,例如Color,Font

组件分类：

* 容器组件
* 非容器组件



## 容器组件

窗体：JFrame jFrame = new JFrame("第一个窗体");

对话框：

* 消息

JOptionPane.showMessageDialog(jFrame,"明天下雨","通知

",JOptionPane.INFORMATION\_MESSAGE);

* 警告

JOptionPane.showMessageDialog(jFrame,"迟到扣钱","警告

",JOptionPane.WARNING\_MESSAGE);

* 错误

JOptionPane.showMessageDialog(jFrame,"路径不存在","错误

",JOptionPane.ERROR\_MESSAGE);

* 输入框

JOptionPane.showInputDialog("请输入金额：");

* 确认框

JOptionPane.showConfirmDialog(jFrame,"确认是否要删除文件？");

面板：JPanel jPanel = new JPanel();

文件对话框：

FileDialog fileDialog = new FileDialog(jFrame,"请选择打开的文件",FileDialog.LOAD);

fileDialog.setVisible(true);

System.out.println("文件目录：" + fileDialog.getDirectory());

System.out.println("文件名称：" + fileDialog.getFile());

## 非容器组件

Jlabel：标签

JTextField文本域

JRadioButton单选按钮

JComboBox 下拉框

JCheckBox复选框

JTextArea文本区域

JList 列表框

JButtion普通按钮

## 菜单组件

JmenuBar菜单条

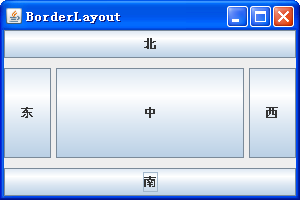
JMenu菜单

JMenultem菜单项

复选菜单：菜单添加菜单

## 布局管理器

BorderLayout边框布局：窗体默认的布局方式，未指定位置添加到中间；没有东、西、南、北其中之一，中间会占用其位置



FlowLayout流式布局：面板默认的布局方式，可指定对齐方式（默认居中）；水平间隙；垂直间隙



GridLayout网格布局：可指定行数、列数、水平间隙、垂直间隙

CardLayout卡片布局：如计算器

## 事件

事件与事件源：

一个事件是事件类的实例对象。事件类的根类是java.util.EventObject。

事件对象包含事件相关的属性，可以使用EventObject类中的实例方法getSource获得事件的源对象。

EventObject类的子类可以描述特定类型的事件

例如：

|  |  |  |
| --- | --- | --- |
| 用户动作 | 源对象 | 触发的事件类型 |
| 点击按钮 | JButton | ActionEvent |
| 文本域按回车 | JTextField | ActionEvent |
| 窗口打开，关闭，最小化，关闭 | Window | WindowEvent |
| 单击，双击，移动，鼠标 | Component | MouseEvent |
| 点击单选框 | JradioButton | ItemEvent ActionEvent |
| 点击复选框 | JcheckBox | ItemEvent ActionEvent |

监听器：

* 添加动作事件监听器：addActionListener（鼠标点击或空格）
* 添加鼠标事件监听器：addMouseListener（鼠标）

传入适配器Mouseadapter可仅对需要的鼠标事件重写

* 添加键盘事件监听器addKeyListener

# 知识进阶

## 批处理文件（bat）

批处理的作用就是**自动的连续执行多条命令**，编写bat处理文件可以使用记事本的方式:

常见批处理文件的命令：

echo 表示显示此命令后的字符

tiltle 设置窗口的标题。

echo off 表示在此语句后所有运行的命令都不显示命令行本身

color 设置窗体的字体颜色。

@与echo off相象，但它是加在每个命令行的最前面，表示运行时不显示这一行的命令行（只能影响当前行）。

pause 运行此句会暂停批处理的执行并在屏幕上显示Press any key to continue...的提示，等待用户按任意键后继续

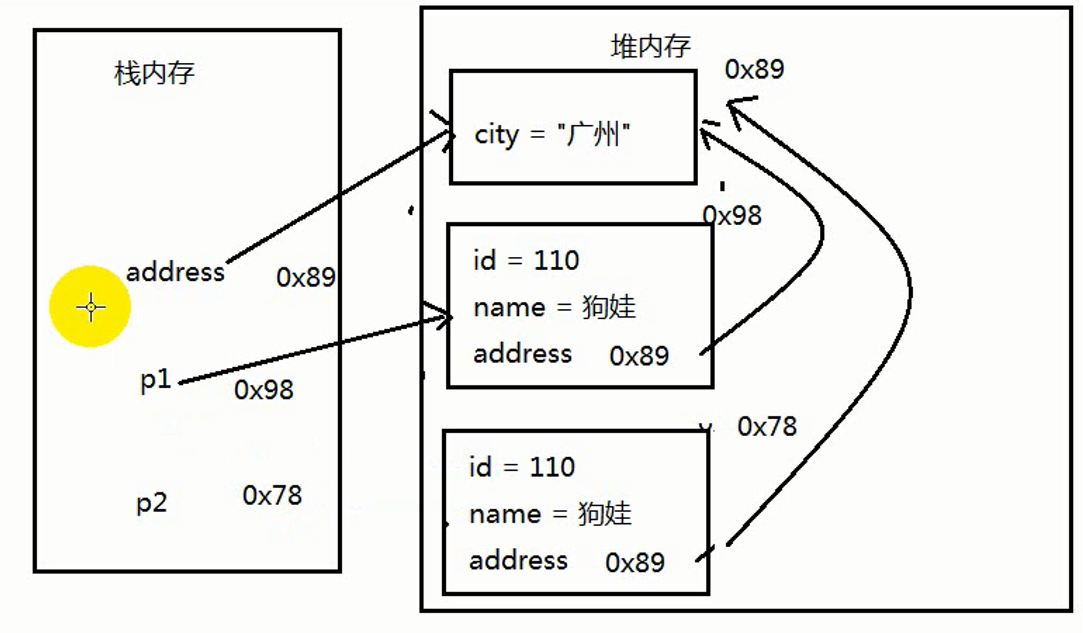
rem 表示此命令后的字符为解释行（注释），不执行，只是给自己今后参考用的（相当于程序中的注释）或者%注释的内容%

%[1-9]表示参数，参数是指在运行批处理文件时在文件名后加的以空格（或者Tab）分隔的字符串

## 对象的克隆

**浅克隆**：若被克隆对象中维护另一一个类对象，则只**克隆另一个对象的地址**（被克隆对象与克隆对象只维护一份）

1. 对象所属类需实现Cloneable接口（标识接口）
2. 重写对象所属类的clone()方法
3. 调用clone()方法



深克隆：利用对象的持久化（先写入文件后读取对象）

## 内存溢出问题

当变量不再被使用时，应及时指向null，以等待垃圾回收

从栈中弹出的对象如果不会作为垃圾回收，即使程序不再使用这些对象，因为栈内部继续维护着这些对象。最终可能会导致内存占用的不断增加，程序性能降低，这就是内存泄漏。

## 单例设计模式线程安全问题

1. 判断单例是否存在前加锁（解决线程安全问题）
2. 锁外层加一层单例是否存在的判断（解决效率问题，第一次之后无需再去判断锁）

**if (s == null){**

synchronized(“lock”){

**if (s ==null){**

s = new S();

}

}

}

## 观察者模式

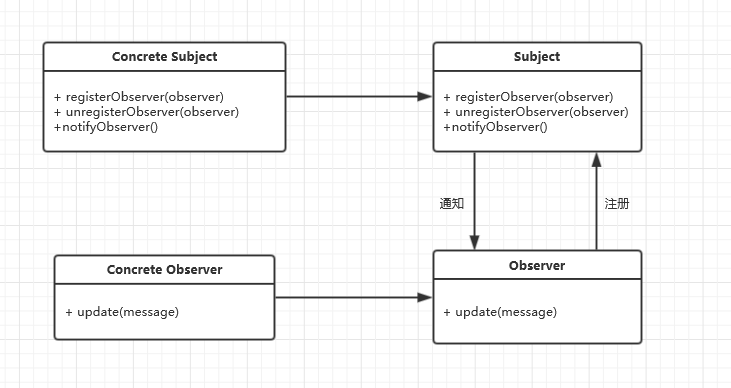
解决的问题：**当一个对象发生指定的动作时，通过另外一个对象作出相应的处理**

又被称为**发布-订阅<Publish/Subscribe>模式、模型-视图<Model/View>模式**，是软件设计模式的一种。

应用场景：

（1）发布者发布信息，订阅者获取信息，订阅了就能收到信息，没订阅就收不到信息。

（2）发布至订阅天气信息（按时更新），订阅者（工人、学生等）获取天气信息并作出相应的动作



观察者模式定义了四种角色：抽象主题、具体主题、抽象观察者、具体观察者

1）抽象主题（Subject）：通常为接口或抽象类，定义一系列操作，如订阅，取消订阅，通知等方法

2）具体主题（Concrete Subject）：具体主题，继承或实现抽象主题，实现相应的逻辑，如维护与观察者的关系，将自身的状态变化告知观察者。

3）抽象观察者（Observer）：定义观察者行为

4）具体观察者（Concrete Observer）：实现观察者行为，实现收到被观察者的通知后将要处理的逻辑

## 反射

（1）反射基本原理

**类字节码文件**是在硬盘上存储的，是一个个的.class文件。我们在**new一个对象时**，JVM会先**把字节码文件的信息读出来放到内存中**，第二次用时，直接使用之前缓存的这个字节码信息。

**字节码的信息包括：类名、声明的方法、声明的字段**等信息，**这些信息需要封装一个对象**，这就是Class类、Method类、Field类。

通过**Class类、Method类、Field类**等可以得到这个类型的一些信息，可以**不用new关键字就创建一个实例，可以执行一个对象中的方法，设置或获取字段的值，这就是反射技术**。

（2）获取Class类对象的方式

Java的反射机制中重要的部分就是获取Class对象，对于Class对象描述了类的信息，可以**通过获取Class对象获取类的成员，方法，注解**等

获取Class对象的三种方法包括：

* Class.forName(“类的路径”) **最常用：给个字符串就能获取对应的Class对象**
* 类对象.getClass()
* 类名.class

（3）通过Class类对象获取类的信息

* getName()：获得类的完整名字
* getSimpleName()：类的简单名称（不带包名）
* getModifiers()：类的修饰符
* newInstance()：无参数构造创建对象（返回object）
* constructor.newInstance("wsx",12); 利用获取的构造函数创建对象，传入对应的参数，没有传null
* constructor.setAccessible(true); 强制使用私有构造函数前需要设置为可访问；
* getConstructors()：获取所有公有构造函数，返回constructor[]
* get DeclaredConstructors()：获取所有构造函数（包括私有），返回constructor[]
* getConstructor(String.class, int.class), getConstructor(null)：获取指定参数构造方法

（4）通过Class对象获方法信息

* getMethods() 获取公有方法返回一个数组,元素类型是Method
* getDeclaredMethods()：获取全部方法，包括私有
* getMethod("setName", String.class)：获取指定参数的公有方法，传入方法名和形参
* getDeclaredMethod("setName", String.class)：获取指定参数的任意方法
* method.invoke(new Person(), null) 调用非静态方法，传入类对象和所需实参，无实参传null
* method.invoke(null, null) 调用静态方法，不传类对象（写null），无实参传null
* method.setAccessible(true); 调用私有方法，需先设置方法可访问；

（5）通过Class对象获成员信息

* getFields()：获取公共属性，返回Field数组
* getField(String name)：获取指定公共属性，传入属性名称，返回Field
* getDeclaredFields()：获取全部属性
* getDeclaredField(String name)：获取任意指定属性；
* field.set(new Person(), int age)：设置获取属性的值，传入类对象（静态属性传null）和属性值
* field.setAccessible(true); 访问私有属性，需先设置属性可访问；

## 工厂模式

用于批量创建对象，通常与反射技术配合使用

## lambda表达式