# Windows快捷键

Alt + F4 ：关闭当前窗口

Shift + Delete ：删除选中的文件(不会进回收站)

Win + E : 打开文件资源管理器

Ctrl + Shift + Esc ：打开任务管理器

win+D ：最小化所有窗口

# Dos命令

盘符切换：如 E: 、F:

展示当前目录下的所有文件：dir

路径切换：cd 文件名1\文件名2

路径切换：cd /d 绝对路径。如：cd /d E:\BaiduNetdiskDownload\BiliBili视频下载工具 3.5

退回上一目录：cd..

退回根目录：cd\

清除屏幕：cls

退出屏幕：exit

查看ip相关信息：ipconfig

打开计算器：calc

打开画图工具：mspaint

打开记事本：notepad

查看连接情况：ping 网址 或者 ping ip地址

创建文件夹：md 文件夹名称

创建文件：cd>a.txt (前提是当前目录是文件夹)

删除文件：del a.txt

删除一堆后缀名一样的文件：del \*.txt

删除文件夹：rd文件夹名称。

如果文件夹里有文件，无法直接删除。此时可以使用

rd /s 文件夹名称 来确认删除

rd /s /q 文件夹名称 不确认直接删除

安装java之后：

java -version ：查看安装的java版本

javac Hello.java ：编译java文件

注意：文件的扩展名不能被隐藏。

javac其实是jdk的bin目录下的javac.exe文件

java Hello ：运行java文件

实际上 javac后跟的是文件名+扩展名

当你用javac编译java文件之后，会生成 类名.class文件。

所以java后跟的是类名

配置jdk的path环境之后，javac命令就可以在任意路径下使用。

配置jdk，只配置到bin目录，比如：C:\Program Files\Java\jdk1.8.0\_181\bin，这样当执行javac命令的时候，首先会去path环境中的路径中找有没有javac.exe，有就执行。

(一般不配)配置classpath路径，这样在任意路径下都能执行.class文件。

比如：a.class文件在E:\a 文件夹下，将D:\d配置在classpath后，在D:\d 文件夹下也

能执行a.class文件。但这样会很混乱。所以一般不配置。

# java基础

## 1.关键字

被java语言赋予特殊含义的单词；全部小写;

关键字有class、long、interface等。

goto和const作为保留字，暂不使用。

## 2.标识符

给类、接口、方法、变量等起名字的字符序列

组成规则：英文字母大小写、数字字符、$和\_

注意事项：不能以数字开头、不能是java中的关键字和保留字、区分大小写

## 3.命名规则

包：其实就是文件夹。包名全部小写。如单级包：liuyi 多级包：cn.itcast

类或者接口：单词的首字母大写，其他小写。如：Student、HelloWorld

方法或者变量：单词首字母小写，从第二个单词开始，首字母大写。如：student、showName

常量：全部大写，多个单词时用\_隔开。如：PI、STUDENT\_MAX

## 4.注释

单行注释：//注释文字

多行注释：/\*注释文字\*/

文档注释：/\*\*注释文字\*/

## 5.进制

1byte = 8bit、1k = 1024byte、1m = 1024k、1g = 1024m、1t = 1024g

二进制由0,1组成，以0b开头；八进制由0,1…7组成，以0开头；

十进制由0,1…9组成，整数默认十进制；十六进制由0,1…9,a,b…f(大小写均可)组成，以0x

开头

## 6.进制之间的转换

系数：每一个位上的数据值本身就是系数。

基数：x进制的基数就是x。

权：我们针对每一个位上的数据进行编号，从右边，并且是从0开始编号，这个编号就是该

位上数据的权值。

每一个位上的系数\*基数^权次幂相加，便可以转换为对应的十进制。例：

二进制：100

100 = 1\*2^2 + 0\*2^1 + 0\*2^0

= 4 + 0 + 0

= 4

八进制：100

100 = 1\*8^2 + 0\*8^1 + 0\*8^0

= 64 + 0 + 0

= 64

十六进制：100

100 = 1\*16^2 + 0\*16^1 + 0\*16^0

= 256 + 0 + 0

= 256

## 7.十进制到其他进制

除基取余，直到商为0，余数反转。

二进制是除2，八进制是除8，十六进制是除16。



## 8.基于8421码的简便转化法



十进制到二进制：

100 = 64 + 32 + 4，也就是对应的1100100。

二进制到八进制：

1011001，将其3个一组拆分，左边不够的补0。

001 011 001

1 3 1

就是八进制的131。

二进制到十六进制类似，只不过是4个一组拆分。

## 9.原码、反码、补码

所有的数据运算都是采用补码进行的。







特殊的数值：

-128在一个字节，也就是8位的时候，原码反码补码的计算方式不适用于它。8位时，

原码和反码的表示范围是-127~+127，没法表示-128。-128没有原码和反码，只有两个

补码的表示形式。11111111，10000000 都是-128的补码。10000000是 -127的补码-1

得到的，11111111是127的补码+1得到的。

补充：计算机中只有补码，补码会存储在内存中，没有原码和反码，原码和反码只是用于表

述数值的工具。

## 10.基本数据类型

在数字定义时，整数默认int，浮点数默认double

long型数据后要加后缀L，比如：long a = 100L;

float型数据后要加后缀F，比如：float b = 100F;



## 11.基本数据类型之间的默认转换

这是运行+ - \* / % 时的转换

byte -> short -> char -> int -> long -> float -> double

byte、short、char相互之间不转换，他们参与运算首先转换为int类型。

强制转换可能会损失精度。比如：byte c = (byte) (a + b);

byte b; b = 3 + 4;//不报错，因为常量会先计算结果，如果结果在byte范围内，就不报错

short a = 1; a+=1;//就相当于a = (short)(a + 1);

## 12.键盘录入

Scanner sc = new Scanner(System.in);

int a = sc.nextInt();//获取键盘录入的整形数据

System.out.println(a);//输出a

注意：main方法中的String[] args 其实也是用来接收键盘录入的数据的，只不过不再使用了。

## 13.switch中的可用类型

基本有int、char、short、byte及其对应的包装类型。后三者都是转化为int再进行判断；

而long 、float、double转换为int都会丢失精度，boolean则是无法转换为int，所以他

们不能用在switch中。

jdk1.5后增加了枚举，jdk1.7后增加了字符串String，判断字符串是用hashCode方法。

case后面的只能是常量，不能是变量，且多个case后面的值不能出现相同的。

default的位置不会影响它是匹配完所有的case之后，最会才会走它。

switch语句只有遇到break或者执行到程序的末尾才会结束。

## 14.方法介绍

方法格式：

修饰符 返回值类型 方法名(参数类型 参数名1,参数类型2 参数名2…){

方法体语句;

return 返回值;//有返回值的

}

方法重载：在同一个类中，方法名相同而参数列表不同的现象。

## 15.数组

数组定义格式：

数据类型[] 数组名;

数组类型 数组名[];

数组的初始化：就是为数组开辟内存空间，并为每个数组元素赋值。

动态初始化：只指定长度，由系统给出初始化值

数据类型[] 数据名 = new 数据类型[数组长度]

例：int[] arr = new int[3]

静态初始化：给出初始化值，由系统决定长度

数据类型[] 数据名 = new 数据类型[]{元素1,元素2,…}

数据类型[] 数据名 = {元素1,元素2,…}

例：int[] arr = new int[]{1,2,3};

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

注意：数组属于不可变数据，一但确定了长度，则长度不可再变，会一直占据内存中。

注意：数组的动态初始化和静态初始化不能同时使用。

int[] arr = new int[3]{1,2,3}; //错误

注意：数组名.length 返回该数组的长度，也就是该数组的元素的个数。

## 16.数组工具类

位置：jdk的java.util.Arrays

位置：apache的org.apache.commons.lang3.ArrayUtils

(1)返回指定数组的字符串表示形式

方法：static String toString(int[] a)

形参数组类型还可以是：boolean、byte、char、double、float、long、short、Object

int[] a = {6,7,8};

String arr = Arrays.toString(a);

(2)获取数组中的最大值，jdk8.0及以上版本：

int[] a = {6,7,8};

int maxVal = Arrays.stream(a).max().getAsInt();

(3)获取数组中的最小值，jdk8.0及以上版本：

int[] a = {6,7,8};

int minVal = Arrays.stream(a).min().getAsInt();

(4)比较两个相同类型的数组中的值是否完全相同

方法：public static boolean equals(int[] a, int[] a2)

int[] a = {6,7,8};

int[] b = {7,8,9};

boolean c = Arrays.equals(a,b);

形参数组类型还可以是：boolean、byte、char、double、float、long、short、Object

(5)数组排序

方法：public static void sort(int[] a)，底层是快速排序。

形参数组类型还可以是：byte、char、double、float、long、short、Object

int[] a = {6,7,8};

Arrays.sort(a);

(6)二分查找

方法：public static int binarySearch(int[] a,int key)

形参数组类型还可以是：byte、char、double、float、long、short、Object

int[] a = {6,7,8};

int index = Arrays. binarySearch(a,7); //该方法找不到元素就会返回负数

(7)反转数组元素

方法：public static void reverse(int[] array)

int[] a = {6,7,8};

ArrayUtils.reverse(a);

形参数组类型还可以是：boolean、byte、char、double、float、long、short、Object

注意：reverse()方法还有很多其它的形参格式，可以去源码查看。

(8)查询元素在数组中第一次出现时的索引

方法：public static int indexOf(int[] array, int valueToFind)

int[] a = {6,7,8};

ArrayUtils.indexOf(a,8);//该方法在数组中找不到对应元素时返回-1

注意：该方法的第一个参数也可以是其他类型的数组，甚至可以增加一个参数表示从

哪个索引开始查起，具体可以查询源码。

## 17.二维数组

含义：元素为一维数组的一个数组。

数组定义格式1：

数据类型[][] 变量名 = new 数据类型[m][n]; //m表示有几个一维数组，n表示一维数组

中的元素个数。

举例：int[][] arr = new int[3][2];

内存图实例：



数组定义格式2：

数据类型[][] 变量名 = new 数据类型[m][]; // m表示有几个一维数组，但每个一维数组

中具体有多少元素未指定，可以之后指定。

举例：

int[][] arr = new int[3][];

System.out.println(arr[0]); //输出null

arr[0] = new int[2]; //指定第一个一维数组的长度为2

arr[1] = new int[3];

arr[2] = new int[1];

注意：当你不给一维数组指定长度时，一维数组不会在内存中被创建。原因就是数组是不可

变数据。

数组定义格式3：

基本：数据类型[][] 变量名 = new 数据类型[][]{{元素1...},{元素1...},{元素1...}};

简化：数据类型[][] 变量名 = {{元素1...},{元素1...},{元素1...}};

## 18.内存分配

为了提高运算效率，对空间进行不同区域的划分，不同的区域有不同的处理数据方式和内存

管理方式。

垃圾数据是使用完毕的数据。

栈：存储局部变量(方法定义中或方法声明上的变量)，局部变量必须先赋值，后使用。

栈中的垃圾数据会被立即回收。

堆：存储new出来的东西

每一个new出来的东西都有地址值

每个变量都有默认值：

byte,short,int,long是0，float,double是0.0，char是’\u0000’，boolean是false，引用类

型是null。

堆中的垃圾数据会在垃圾回收器空闲的时候回收。

注意：java中的参数传递问题。当传递的是基本类型时，传递的是数据的值，所以形参的改

变对实际参数没有影响。但是如果传递的是引用类型，传递的是数据的地址，形参的改

变会影响实际参数。注意：参数传递时，String当作基本类型看待，而数组任何时候都

是引用类型。

## 19.面向对象

面向过程：强调的是每一个功能的步骤。

面向对象：强调的是由对象去调用功能，而不用自己具体实现每一个功能。

类：是一组相关的属性和行为的集合。是一个抽象的概念。

对象：是该类事务的具体表现形式。具体存在的个体。

创建对象：类名 对象名 = new 类名();

匿名对象：new 类名(); //匿名对象只能使用一次，可以作为参数传递

使用成员变量：对象名.变量名

成员变量随着对象的创建而存在，随着对象的消失而消失

使用成员方法：对象名.方法名(...)

内存图解：

两个对象的：



三个对象的：



补充：一个类可以产生无数个对象，但类就那么一个，方法也就那么一个。

## 20.关键字

(1)public关键字：

一个.java源文件中可以有多个类，但是只能有一个被public修饰的类，并且public的类

名必须跟文件名一致；如果只有一个类且是非public的，此类名可以跟文件名不同。

(2)private关键字：

可以修饰成员变量和成员方法

被修饰的成员只能在本类中才能访问

(3)this关键字：

表示当前类的对象引用

谁调用这个方法，该方法内部的this就代表谁

静态方法中，不能使用this关键字。

因为静态随类的加载而加载，this随对象的创建而存在，静态比对象先存在。

(4)static关键字

static可以修饰成员变量(静态变量)、成员方法(静态方法)。

静态变量存储于方法区的静态区。因为静态变量被所属类的所有对象共享，因此只

需要在内存中存储一份。

静态方法中无法直接访问非静态的变量和方法，但是可以用new一个对象的方式

解决。比如：

public static void show(){

User user = new User();

user.setAge(12);// setAge()方法是非静态的

}

特点：随类的加载而加载，随类的消失而消失

优先于对象存在

被类的所有对象共享

可以通过类名直接调用

使用介绍：一般使用在工具类的方法上，通过 类名.方法名 调用。

如果不想让别人创建工具类对象，可以用private修饰构造方法。

内存图解：



## 21.构造方法

作用：给对象中的数据进行初始化。

构造方法也可以重载。

如果我们自己写了构造方法，则系统默认的无参构造方法就会失效。

格式：

方法名和类名相同

没有返回值类型，也没有void

没有具体返回值

创建对象时的内存图解：



## 22.Math工具类

获取随机数：

public static double random():返回double类型的数值，该值大于等于0.0且小于1

int num = (int)(Math.random()\*100)+1; // num是1-100之间的随机数

java.lang包下的类不需要导入，其他的全部需要导入。

## 23.代码块

含义：java中，被{}括起来的代码被称为代码块。

根据其位置和声明的不同，可以分为：

局部代码块：局部位置(方法中)。用于限定变量的生命周期。

构造代码块：类中的成员位置(类中方法外)。每次调用构造方法都会执行，且在构造方

法前执行，所以可以将多个构造方法中重复的代码放到构造代码块中。

静态代码块：类中的成员位置(类中方法外)，被static修饰。在类的初始化过程中执行，

且在构造代码块的前面执行，只执行一次。

执行顺序：静态代码块 -- 构造代码块 -- 构造方法

## 24.继承

多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需再定义

这些属性和行为，只需要继承那个类即可。

java类只能单继承，但可以多层继承。

通过extends关键字可以实现类与类的继承

class 子类名 extends 父类名{}

父类又可以称为超类、基类

子类又可以称为派生类

优点：提高了代码的复用性

提高了代码的维护性

是多态的前提

缺点：增加了类的耦合性

开发原则：低耦合、高内聚

注意事项：

(1)子类只能继承父类所有非私有的成员(成员方法和成员变量)

(2)子类不能继承父类的构造方法，但是可以通过super()访问父类的构造方法

(3)不要为了部分功能而去继承

继承中成员变量的关系：

子类方法中访问一个变量的查找范围(就近原则)：

a:在子类的局部范围找

b:在子类的成员范围找

c:在父类的成员范围找

d:还找不到就报错

方法重写(Override)：子类中出现和父类中方法声明一模一样的方法。

父类中私有方法不能被重写，因为父类私有方法子类根本无法继承。

子类重写父类方法时，访问权限不能更低。

父类静态方法，子类也必须通过静态方法重写。(其实也不算重写)

方法重载(Overload)：本类中出现方法名一样而参数列表不同的方法，与返回值无关。

继承中成员方法的关系：

通过子类调用方法(就近原则)：

a:先找子类中

b:再找父类中

c:还找不到就报错

继承中构造方法的关系：

子类中所有的构造方法默认都会访问父类的空参构造方法。因为要先初始化父类。

子类每一个构造方法的第一条语句默认都是super();

注意：如果父类没有无参构造，子类的构造方法中第一条语句得用super(...)

访问父类其他的构造方法才能初始化父类数据，否则编译报错。

this和super关键字的区别：

this代表本类对象的引用。

super代表父类存储空间的标识，可以理解为父类引用。

this.成员变量 调用本类的成员变量

super.成员变量 调用父类的成员变量

this.成员方法 调用本类的成员方法

super.成员方法 调用父类的成员方法

this(...) 调用本类的构造方法

super(...)调用父类的构造方法

当父类、子类都存在静态代码块、构造代码块、构造方法时的执行顺序：

创建子类对象时，执行顺序是：

静态代码块Fu--静态代码块Zi--构造代码块Fu--构造方法Fu--构造代码块Zi--构造方法Zi

## 25.final关键字

final可以修饰类、方法、变量。

特点：

final修饰类，该类不能被继承

final修饰方法，该方法不能被重写

final修饰变量，该变量成为常量。如果变量是引用类型，则该变量不能指向新的地址。

## 26.多态

含义：某一个事物，在不同时刻表现出来的不同状态。

多态的前提：

有继承关系

有方法重写(没有方法重写就没有意义)

有父类引用指向子类对象，就是Fu f = new Zi();

多态中的成员访问特点：

A：成员变量

编译时看父类的，运行时也看父类的

B：构造方法

创建子类对象时，首先访问父类的构造方法，对父类的数据进行初始化。

C：成员方法

编译时看父类的，运行时看子类的(方法重写)

D：静态方法

编译时看父类的，运行时也看父类的(静态和类相关，算不上重写)

多态的优点：

A：提高了代码的维护性(继承保证)

B：提高了代码的扩展性

多态的弊端：

当父类没有方法A，而子类有方法A且想用A时无法使用

Fu f = new Zi(); // 向上转型

f.A(); // 报错

解决方法：强制转换

Zi z = (Zi) f; //向下转型

z.A(); //可以使用

## 27.抽象类

抽象类和抽象方法必须被abstract关键字修饰

格式：

abstract class 类名{}

public abstract void eat();

注意事项：

A：抽象类不一定有抽象方法，但是有抽象方法的类一定是抽象类。

B：抽象类有构造方法，但不能实例化，可以按照多态的方法由具体的子类实例化。

C：抽象类的子类，要么是抽象类，要么是具体类。是具体类时必须重写抽象类中的所

有抽象方法。

抽象类的成员特点：

成员变量：既可以是变量，也可以是常量

构造方法：有，用于子类访问父类数据的初始化

成员方法：既可以是抽象的，也可以是具体的

特性：抽象方法，强制要求子类做的事情。

具体方法，子类继承的事情，提高代码的复用性。

## 28.接口

接口用关键字interface表示

格式：

interface 接口名 {}

接口成员特点：

成员变量：只能是常量，并且是静态的。

默认修饰符：public static final

构造方法：没有构造方法。

成员方法：只能是公共的抽象方法。

默认修饰符：public abstract

类实现接口：

class 类名 implements 接口名1,接口名2... {}

接口继承接口：

interface 接口名1 extends 接口名2,接口名3...{}

所有的类最终都继承自一个类，java.lang包下的类Object。

接口无法实例化，可以按多态的方式由具体的子类实例化。

接口的子类，如果是抽象类，无要求。但如果是具体类，则要重写接口中所有的抽象方法。

## 29.修饰符





## 30.内部类

含义：类的内部的类。

内部类的访问特点：

A：内部类可以直接访问外部类的成员，包括私有

B：外部类要访问内部类的成员，必须创建对象

按照内部类在类中定义的位置不同，可以分为：

成员位置(成员内部类)

局部位置(局部内部类)

其他类如果想访问一个类中的成员内部类，需要创建对象，格式如下：

外部类名.内部类名 对象名 = 外部类对象.内部类对象;

例：Outer.Inner oi = new Outer().new Inner();

如果成员内部类被静态修饰后，访问格式如下：

外部类名.内部类名 对象名 = new 外部类名.内部类名();

例：Outer.Inner oi = new Outer.Inner();

成员内部类的修饰符：

private 为了保证数据的安全性

注意：如果内部类被private修饰，则在其他类中无法创建该内部类对象

static 为了让数据访问更方便

被static修饰的成员内部类只能访问外部类的静态成员

class A{

public static class B{

public static void hah(){

System.out.println(“调用”);

}

}

}

如上代码，成员内部类被static修饰，它的方法也被static修饰，则hah()可以这样调用：

A.B.hah();

局部内部类：

A：同样可以直接访问外部类的成员

B：在局部位置，可以创建内部类对象，通过对象调用内部类方法，来使用局部内部类

的功能。

注意：局部内部类访问局部变量，该变量必须用final修饰。

原因：局部变量会随着方法的调用完毕而消失，但局部对象并没有马上从堆内存中消失，

还要用这个变量。为了让数据还能继续使用，就用final修饰，这样，在堆内存里

面存储的其实是一个常量值。

匿名内部类(就是内部类的简化写法)：

A：前提是存在一个类或者接口

这里的类可以是具体类，也可以是抽象类

B：本质是一个继承了该类或者实现了该接口的子类匿名对象

格式：

new 类名或者接口名(){

重写方法;

}

C：优点是匿名内部类用完就是垃圾，可以被立即回收，节省内存

## 31.根类Object

介绍：Object是类层次结构的根类，所有类都直接或间接的继承自该类

构造方法只有一个：public Object(){} //这解释了为什么子类的构造方法都是默认访问

父类的无参构造方法。

方法：

public int hashCode()：返回该对象的哈希码值，用以提高哈希表(如Hashtable的哈希表)

的性能。

哈希码值是根据哈希算法计算出来的一个值，默认与对象的实际内存地址有关，但不是

实际的内存地址。

一般来说，不同对象返回的哈希码值一般不会相同，但是同一个对象的哈希码值一定

相同。

public final Class getClass()：返回调用这个方法的对象的字节码对象，也就是Class对象。

Class对象应用在反射技术中，在Class类中有许多操作数据的方法。比如：

public [String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) getName()：返回字符串表示的包名+类名，例：cn.itcast.Student

public String toString()：返回一个对象的字符串表示。默认是包名+类名+内存地址值，

比如：[cn.itcast.Student@42552c](mailto:cn.itcast.Student@42552c)。该返回值没有意义。建议所有子类重写该方法。

重写为返回该类的所有成员变量的字符串表示即可。

注意：内存地址值不是实际的内存地址，而是16进制的哈希码值。所以：Object类的

toString()方法就等同于：

this.getClass().getName()+’@’+Integer.toHexString(this.hashCode())。toHexString()方法

是将一个十进制的整数转化为十六进制。

注意：直接输出一个对象的名称，其实就是调用该对象的toString()方法。

public boolean equals(Object obj)：表示obj对象是否与调用这个方法的对象相等。

补充：== 可以比较基本类型，比较的是值是否相同。比较引用类型的时候，是比

较地址值是否相同。

Object类的equals()是调用 == 号比较的，比较的是地址值，因此一般没有意义。

所以建议子类重写equals方法，一般重写为比较成员变量值。可以用idea或者

eclipse自动生成。

protected void finalize()：当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃

圾回收器调用此方法，用于垃圾回收，但是什么时候回收，不确定。

protected Object clone() ：创建并返回此对象的一个副本。如何使用该方法，举例：

1.创建类Student

2.Student类重写clone方法

3.Student类实现接口Cloneable，该接口没有任何抽象方法，是一个标记接口

4.创建StudentDemo类，在该类中，调用Student类的clone方法。

注意：这是深克隆，也就是返回一个全新的对象，不和之前的对象共用一个内存地址。

## 32.Scanner类

JDK5以后用于获取用户的键盘输入。

Scanner类最常用于接收键盘录入的数据。比如：

Scanner sc = new Scanner(System.in);

实参中，System是一个类，in则是System类中的一个静态变量，定义格式是：

public static final InputStran in;// 标准的输入流

之所以用Scanner类来包装该输入流，是因为Scanner中有一些将输入的字符串数据转

为其它数据类型的方法。

方法：

hasNextXxx()：判断是否还有下一个某数据类型的输入项，其中Xxx可以是Int、Double

等。如果需要判断是否包含下一个字符串，则可以省略Xxx。该系列方法返回Boolean

类型。

nextXxx()：获取下一个输入项，Xxx的含义和上个方法中的Xxx相同。nextLine()是获取

下一个字符串数据(输入流输入的数据都是字符串)。

## 33.String类(非常重要，使用最广泛)

字符串是由多个字符组成的一串数据(字符序列)，可以看作是字符数组。

字符串是常量，它们的值创建了之后，在内存中便无法被修改。

问题：为什么将String设计为不可变的？

(1)字符串中常量池的需要

String不同于普通基础变量类型的地方在于对象。java中的字符串对象都保存在字

符串常量池中，创建字符串时首先会从这个常量池中查找是否已经存在相同的字符

串对象，不存在的情况下才会创建新的字符串对象。如果字符串被设计成可变的，

便存在当修改一个字符串的时候同时影响另一个指向常量池的相同对象(有潜在性

风险)。

(2)HashCode缓存

在HashMap或者HashSet数据结构中，存储String是经常需要用到哈希值，String

不可变的特征保证哈希值的唯一性，避免哈希引起冲突。也不必在每一次使用集合

中的String对象时候重复计算哈希值，高效率。

(3)线程安全

String的不可变性是指能够在不同线程中共享，同时确保线程安全，帮助开发者

减少线程同步的工作。

构造方法：



字符串直接赋值的方式是先到字符串常量池中去找这个字符串，没有的话就创建并返回这个

字符串。图解(注意是直接赋值的方式)：



如果是用构造方法赋值的方式则稍有不同：图解：



面试题：

String s1 = “hello”;

String s2 = “world”;

String s3 = “helloworld”

System.out.println(s3 == s1+s2);//false

System.out.println(s3 == ” hello”+” world”);//true

解析：

字符串如果是变量相加，先开空间，再拼接

字符串如果是常量相加，是先加，然后在常量池中找加之后的字符串，如果有就直

接返回，否则创建并返回。

方法：

int length()：返回此字符串的长度

boolean equals(Object obj)：比较字符串的内容是否相同，区分大小写

boolean equalsIgnoreCase(String str)：比较字符串的内容是否相同，不区分大小写

boolean contains(String str)：判断是否包含小字符串

boolean startsWith(String str)：判断是否以某个指定的字符串开头

boolean endsWith(String str)：判断是否以某个指定的字符串结尾

boolean isEmpty()：判断字符串是否为空。也就是判断””。

char charAt(int index)：获取指定索引位置的字符

int indexOf(int ch)：返回指定字符在此字符串中第一次出现处的索引

int indexOf(String str)：返回指定字符串在此字符串中第一次出现处的索引

int indexOf(int ch,int fromIndex)：返回指定字符在索引fromIndex后第一次出现处的索引

int indexOf(String str,int fromIndex)：返回指定字符串在索引fromIndex后第一次出现处

的索引

String substring(int start)：从指定位置开始截取字符串，默认到末尾

String substring(int start,int end)：从指定位置开始到指定位置结束截取字符串(包左不包

右)

byte[] getBytes()：把字符串转为字节数组

char[] toCharArray()：把字符串转为字符数组

static String valueOf(char[] chs)：把字符数组转为字符串

static String valueOf(int i)：把int类型的数据转为字符串

还有转化其他数据类型的valueOf方法

String toLowerCase()：把字符串转小写

String toUpperCase()：把字符串转大写

String concat(String str)：字符串拼接

String replace(char old,char new)：替换字符

String replace(String old,String new)：替换字符串

String trim()：去除字符串的两端空格

int compareTo(String str)：按照字典顺序比较字符串，区分大小写

int compareToIgnoreCase(String str)：按照字典顺序比较字符串，不区分大小写

## 34.StringBuffer

含义：线程安全的可变字符序列，与String不同的是它的内容和长度是可变的。

构造方法：

public StringBuffer()：无参构造，默认容量为16

public StringBuffer(int capacity)：指定初始化容量

public StringBuffer(String str)：指定初始化字符串，容量为str.length()+16

方法：

public int capacity()：返回当前容量

public int length()：返回长度(字符数)

public StringBuffer append(String str)：把任意类型数据添加到字符串缓冲区

public StringBuffer insert(int offset,String str)：在指定位置把任意类型的数据插入到字符

串缓冲区

public StringBuffer deleteCharAt(int index)：删除指定位置的字符

public StringBuffer delete(int start,int end)：删除从指定位置开始到指定位置结束的内容

(包左不包右)

public StringBuffer replace(int start,int end,String str)：从start开始到end结束用str替换

(包左不包右)

public StringBuffer reverse()：反转

public String substring(int start)：截取并返回一个字符串，StringBuffer本身内容不变  
 public String substring(int start,int end)：截取并返回一个字符串，StringBuffer本身内容不

变

public String toString()：将StringBuffer里的内容以字符串的形式返回

## 35.StringBuilder

StringBuilder与StringBuffer的所有方法基本一致。

区别是StringBuilder线程不安全，因此效率更高。

## 36.算法

(1)冒泡排序

相邻元素两两比较，大的往后放。第一次完毕，最大值就出现在了最大索引处。同理，继续，

即可得到一个排序好的数组。每一次比较完毕后，下一次比较的时候，就会减少一个

元素的比较，也就是每次比出来的最大值不用再和其他元素比。

核心代码：



(2)选择排序

从0索引开始，依次和后面元素比较，小的往前放，第一次完毕，最小值出现在最小索引处。

同理，继续，即可得到一个排序好的数组。每一次比较完毕后，下一次比较的时候，就

会减少一个元素的比较，也就是每次比出来的最小值不用再和其他元素比。

核心代码：



(3)二分查找

前提：数组必须有序。

思想：每次都只和中间的元素比较。



## 37.基本类型的包装类



(1)Integer

构造方法：

public Integer(int value)

public Integer(String s)

方法：

public static String toBinaryString(int i)：十进制转二进制

public static String toOctalString(int i)：十进制转八进制

public static String toHexString(int i)：十进制转十六进制

public static String toString(int i,int radix)：十进制转其他进制，范围：2-36，由radix控制

public static String toString(int i)：int数据转String数据

public static int parseInt(String s)：String数据转int数据

public static int parseInt(String s,int radix)：指定其他进制的字符串形式的数字转十进制，

范围：2-36，由radix控制

常量：

public static final int MAX\_VALUE：int类型数据的最大值

public static final int MIN\_VALUE：int类型数据的最小值

JDK1.5出现了自动拆装箱，也就是基本类型的包装类对象可以和对应的基本类型一样操作。

原因是编译器在编译后，会自动转换类型。比如：



面试题：

Integer i5 = 128;

Integer i6 = 128;

System.out.println(i5 == i6); // false

Integer i7 = 127;

Integer i8 = 127;

System.out.println(i7 == i8);//true

解析原因(valueOf方法从jdk1.5开始)：

因为 Integer i5 = 128; 编译后会变为Integer i5 = Integer.valueOf(128);

查看valueOf的源码可以发现，jdk内置了byte数组缓冲池，范围是-128~127，只要变

量在这个范围内,就会返回同一个Integer对象。反之就会返回一个新的Integer对象。

(2)Character

构造方法：

Character(char value)

方法：

public static boolean isUpperCase(char ch)：判断是否大写字符

public static boolean isLowerCase(char ch)：判断是否小写字符

public static boolean isDigit(char ch)：判断是否数字字符

public static char toUpperCase(char ch)：转换为大写字符

public static char toLowerCase(char ch)：转换为小写字符

## 38.Math类

作用：用于数学运算的类。

成员变量：

public static final double **E：**比任何其他值都更接近 *e*（即自然对数的底数）的 double

值。

public static final double **PI：**比任何其他值都更接近 *pi*（即圆的周长与直径之比）的

double值。

成员方法：

public static double **abs**(double a)：返回double值的绝对值

public static float **abs**(float a)：返回 float 值的绝对值

public static int **abs**(int a)：返回 int 值的绝对值

public static long **abs**(long a)：返回 long 值的绝对值

public static double **ceil**(double a)：向上取整 12.34向上取整为13.0

public static double **floor**(double a)：向下取整12.34向下取整为12.0

public static double max(double a,double b)：返回a,b之间的最大值

public static float max(float a,float b)

public static int max(int a, int b)

public static long max(long a,long b)

public static double min(double a,double b)：返回a,b之间的最小值

public static float min(float a,float b)

public static int min(int a, int b)

public static long min(long a,long b)

public static double pow(double a,double b)：返回a的b次幂

public static double **random**()：返回一个double随机值，范围：[0.0,1.0)

随机获取一个[start,end]之间的数，start和end都是int类型，返回的随机数也是int

类型：

int num = (int) (Math.random() \* (end - start + 1)) + start;

public static long **round**(double a)：四舍五入

public static int **round**(float a)：四舍五入

public static double **sqrt**(double a)：返回a的正平方根

## 39.Random类

作用：用于生成伪随机数流。此类使用48位的种子。

构造方法：

public **Random**()：使用默认种子(当前时间毫秒值)创建对象

public **Random**(long seed)：使用给定种子创建对象。

给定种子后，每次生成的随机数是相同的。

成员方法：

public int **nextInt**()：返回int范围内的随机数

public int **nextInt**(int n)：返回[0,n)范围内的随机数

## 40.System类

作用：包含一些静态字段和方法，该类不能实例化。

方法：

public static void gc()：运行垃圾回收器。

当一个堆内存中的对象没有被指向或者长时间没有被使用时，执行gc()方法会先执

行该对象的finalize()，来回收该对象所在内存。finalize()是Object类的方法，

子类可以重写它。

但是运行垃圾回收器要消耗很多资源，因此不推荐主动调用它。

public static void exit(int status)：终止当前正在运行的虚拟机。status非0时表示异常终

止。

public static long currentTimeMillis()：返回当前时间的毫秒值。

public static void arraycopy(Object src,int srcPos,Object dest,int destPos,int length)：

从指定源数组复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。

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-要复制的数组元素的数量。

## 41.BigInteger类

作用：可以让超出Integer范围内的数据进行运算。

构造方法：

public BigInteger(String val)

成员方法：

public BigInteger add(BigInteger val)：加

public BigInteger subtract(BigInteger val)：减

public BigInteger multiply(BigInteger val)：乘

public BigInteger divide(BigInteger val)：除

public BigInteger[] divideAndRemainder(BigInteger val)：返回商和余数的数组

## 42.BigDecimal类

作用：由于float和double型数据在进行运算的时候很容易丢失精度。所以创建了BigDecimal

用以进行浮点数的运算。

含义：不可变的，任意精度的有符号十进制数。

构造方法：

public **BigDecimal**(String val)

成员方法：

public BigDecimal **add**(BigDecimal augend)：加

public [BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) subtract([BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) subtrahend)：减

public [BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) multiply([BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) multiplicand)：乘

public [BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) divide([BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) divisor)：除

public [BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) divide([BigDecimal](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html) divisor,int scale,int roundingMode)：除

scale表示结果留几位小数。roundingMode表示舍入模式，其值为常量，有这些：

[ROUND\_UP](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_UP), [ROUND\_DOWN](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_DOWN), [ROUND\_CEILING](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_CEILING), [ROUND\_FLOOR](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_FLOOR), [ROUND\_HALF\_UP](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_HALF_UP),

[ROUND\_HALF\_DOWN](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_HALF_DOWN), [ROUND\_HALF\_EVEN](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_HALF_EVEN), [ROUND\_UNNECESSARY](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/math/BigDecimal.html#ROUND_UNNECESSARY)。

## 43.Date类

作用：jdk1.0加入的日期类，用于处理时间。但里面大部分的方法都已过时。

构造方法：

public Date()：根据当前时间毫秒值创建日期对象

public Date(long date)：根据给定的毫秒值创建日期对象

成员方法：

public long getTime()：获取时间的毫秒值

public void setTime(long time)：根据time的值设置时间

## 44.SimpleDateFormat类

作用：用于Date对象与String对象之间的相互转换。

构造方法：

public SimpleDateFormat()：默认模式创建对象

public SimpleDateFormat([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) pattern)：根据给定模式pattern创建对象

年-y、月-M、日-d、时(0~23)-H、时(0~11)-h、分-m、秒-s。举例：

SimpleDateFormat sdf = new SimpleDateFormat(“yyyy:MM:dd HH:mm:ss”);

成员方法：

public final [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) format([Date](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Date.html) date)：该方法继承自抽象父类DateFormat。将Date对象

格式化为String对象。

public [Date](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Date.html) parse([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) source)throws [ParseException](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/text/ParseException.html)：该方法继承自抽象父类

DateFormat。将String对象解析为Date对象。需要注意的是：source的日期模式

必须得和创建的SimpleDateFormat对象的模式保持一致，否则无法解析。

## 45.Calendar抽象类

作用：它为特定瞬间与一组诸如 YEAR、MONTH、DAY\_OF\_MONTH、HOUR 等[日历字段](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html#fields)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。

成员方法：

public static [Calendar](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html) getInstance()：使用默认时区和语言环境获得一个日历对象，该对

象为Calendar的子类对象。

public int get(int field)：返回给定日历字段的值。每个日历字段都是静态的成员变量，且

是int类型。field(日历字段)可以是[Calendar](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html).YEAR(年)、[Calendar](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html).MONTH(月)、

[Calendar](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html).DATE(日)。需要注意的是根据[Calendar](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Calendar.html).MONTH获取的月是从0开始的，

因此获取完月之后得再+1才是正确的月份。

public abstract void add(int field,int amount)：根据给定的日历字段加或减给定的数量。

当amount为正数就是加，为负数就是减。

public final void set(int year,int month,int date)：设置当前对象的年月日。需要注意的是：

month的值得是你想设置的月份-1，因为此对象的月份是从0开始的。

## 46.增强for

含义：是for循环的一种。

格式：

for(数据类型 变量 : 数组或者Collection集合){

操作变量;

}

## 47.静态导入(了解)

格式：import static 包名…类名.静态方法名;

使用：导入后，就可以在类中直接使用方法名来用这个方法。

注意：如果有同名的静态方法，使用时就必须得加前缀了，比如：

System.out.println(java.lang.Math.abs(-100));

## 48.可变参数(了解)

意义：定义方法时不知道该定义多少个形参时使用。

格式：

修饰符 返回值类型 方法名(数据类型… 变量名){

}

补充：格式中的变量名其实代表一个数组，看编译后的class文件就能看到。

## 49.异常

JAVA中的异常分为编译期异常和运行期异常。

关系图：

程序的异常：Throwable

严重问题：Error

异常：Exception

编译期异常：非RuntimeException的异常

运行期异常：RuntimeException以及继承自RuntimeException的异常

## 50.异常的处理方式

提示：

i.try里的代码越少越好。

ii.在try中发现问题后，jvm会生成一个异常对象，将这个对象抛出，和catch里的

类进行匹配，如果匹配上，就会执行该catch里面的代码。

(1)如果不手动做处理，那么jvm会将异常输出到控制台，并结束程序。

(2)try...catch...finally的处理格式：

try{

可能出现问题的代码;

}catch(异常名 变量){

针对问题的处理;

}finally{

释放资源;

}

(3) try...catch的处理格式：

try{

可能出现问题的代码;

}catch(异常名 变量){

针对问题的处理;

}

(4)多个catch的处理格式(加不加finally都可以)：

try{

可能出现问题的代码;

}catch(异常名 变量){

针对问题的处理;

} catch(异常名 变量){

针对问题的处理;

}...

说明：当出现多个catch时，异常的匹配是从上往下，因此最好把范围小的异常写在上面，

一旦执行了某一个catch中的语句，那么其他catch中的语句便不会再执行。

(5) try...finally的格式

这种方式不处理异常，仅仅是为了释放资源。

(6)jdk1.7出现的另一种格式：

try{

可能出现问题的代码;

}catch(异常名1 | 异常名2 | 异常名3 | ... 变量名){

针对问题的处理;

}

该格式中catch里的异常必须是平级关系，不能有父类子类的关系。

(7)使用throws

格式：throws异常类名1, 异常类名2...，用在方法声明后，表示可能抛出某种异常。

比如：

public void method() throws ParseException(){

代码...

}

效果是，如果method()出错了，哪个方法调用method()，哪个方法去处理异常。处理的

方式可以是try...catch的方式处理，也可以继续用throws抛出去。

(8)使用throw

格式：throw 异常类的对象; 例子：

public void method() throws Exception{

int a = 10;

int b = 0;

if(b == 0){

throw new Exception();

}

}

该例中，throw的是new Exception()，是一个编译期异常，所以得处理，这里是用throws处

理的。

注意：throw用在方法体内，只能抛出一个异常类对象，由方法体内处理异常。

如果抛的是运行期异常，方法体内可以不处理，但如果抛的是编译期异常，方法体内必须处

理，处理的方式可以是用throws，也可以用try...catch。

## 51.finally

此处讲的finally，是try…catch…finally类型格式中的finally。

说明：

finally下的语句一定会被执行，除非是jvm提前退出了。

作用：

用于释放资源，一般是用在IO流和数据库连接操作中。

问题：如果catch中有return语句且catch中的代码执行了，那么finally中的语句何时执行?

答：在return前执行，准确的说是在return中间执行。例子：

public static int getInt(){

int a = 10;

try{

a = 10/0;

}catch(Exception e){

a = 30;

return a;

}finally{

a = 40;

}

return a;

}

说明：该方法返回30。流程是，走return a的时候，a是30，也就是return 30。走到一半，

发现有finally，于是走finally中的语句，虽然a的值确实变了，但是return 30已无法

修改。走完finally，继续走return 30，所以返回的还是30。

注意，如果finally中有return语句的话，会走finally中的return语句，不走catch中的。

## 52.异常类Throwable中的方法

public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getMessage()：返回详细消息字符串。

public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) toString()：返回简短描述，是以下字符串的拼接：

此对象的类的name(全路径名)

“: ”(冒号和一个空格)

调用此对象getLocalizedMessage()方法的结果，如果此方法返回null，则只返回类名称

public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getLocalizedMessage()：返回本地化描述，子类可以重写此方法，若不重写，默

认返回与getMessage()相同的结果。

public void printStackTrace()：输出错误信息至控制台。

public void printStackTrace([PrintStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/PrintStream.html) s)：将错误信息输出到指定的输出流，该方法可以将错

误信息保存在日志文件中。

## 53.自定义异常类

说明：有时候需要我们自己定义自己代码中的异常，需要手动写一个异常类。该类必须继承

自Exception或者RuntimeException。继承自Exception意味着是编译期异常，继承自

RuntimeException意味着是运行期异常。为了简便起见，可以在自定义异常类的构造方

法中用super来调用父类对应的构造方法。例如：

public class MyException extends Exception(){

public MyException(){}

public MyException(String message){super(message);}

}

## 54.关于异常的注意事项

(1)子类重写父类方法时，如果父类的该方法使用throws抛出了异常，子类的重写方法必须

也用throws抛出父类该方法相同的异常或者该异常的子类。

(2)如果父类方法抛出了多个异常，子类重写父类该方法时， 只能抛出相同的异常或者多个

异常的子集，子类重写方法不能抛出父类该方法没有的异常。

(3)如果父类方法没有异常抛出，则子类的重写方法也不能抛出异常，最多用try…catch处理

异常。

## 55.File类(java.io包中)

说明：File类是文件和目录(文件夹)路径名的抽象表示形式。

构造方法：

public File(String pathname)：根据路径得到File对象。

public File(String parent,String child)：根据一个文件夹和一个子文件/子文件夹得到File

对象。

public File(File parent,String child)：根据父File对象和子文件/子文件夹得到File对象。

成员方法：

创建功能：

public boolean createNewFile()：创建文件，如果已存在同名且同类型的文件，就不会创

建新的并且该方法返回false。

注意：若想在某个文件夹下创建文件，前提是该文件夹存在，否则createNewFile()会报

错。

public boolean mkdir()：创建文件夹，如果已存在同名的文件夹，就不会创建新的并且

该方法返回false。

注意：若想在某个父文件夹下创建子文件夹，前提是父文件夹存在，否则mkdir()返回

false且不会创建文件夹。

public boolean mkdirs()：创建文件夹，如果父文件夹不存在，会自动创建出来。例如：

File file = new File(“e:\\aaa\\bbb\\ccc\\ddd”);

file.mkdirs();

注意：如果用构造方法创建File对象的时候，路径忘了写盘符(C盘或D盘等其他盘)，

则使用File类的方法时，默认根路径是当前项目路径。

删除功能：

public boolean delete()：删除文件、文件夹，该删除不走回收站。要删除一个文件夹时，

必须该文件夹不包含文件或者文件夹，否则无法删除且该方法返回false。

重命名功能：

public boolean renameTo(File dest)：给一个文件重命名。例：

File file1 = new File(“1.jpg”);

File file2 = new File(“2.jpg”);

file1.renameTo(file2);//将1.jpg改名为2.jpg

注意：如果file1对象中的路径和file2中的路径不同，则是重命名+剪切的效果。也就

是将1.jpg改名为2.jpg同时将2.jpg放到file2对象中的路径中。

判断功能：

public boolean isDirectorv()：判断是否是文件夹

public boolean isFile()：判断是否是文件

public boolean exists()：判断是否存在

public boolean canRead()：判断是否可读

public boolean canWrite()：判断是否可写

public boolean isHidden()：判断是否隐藏

基本获取功能：  
 public String getAbsolutePath()：获取绝对路径

public String getPath()：获取相对路径

public String getName()：获取名称 (若是文件，会带后缀)

public long length()：获取长度(字节数)

public long lastModified()：获取最后一次修改时间，是毫秒值

高级获取功能：

public String[] list()：返回指定目录下(一层)的所有文件和文件夹的名称数组

public File[] listFiles()：返回指定目录下(一层)的所有文件和文件夹的File数组

public String[] list(FilenameFilter filter)：过滤并返回指定目录下(一层)的文件和文件夹的

名称数组

如何过滤：FilenameFilter是一个接口，因此该方法需要一个FilenameFilter的实

现类的对象，实现类需要重写accept(File dir,String name)用以过滤。例如：

//获取e盘下的jpg文件的文件名称

File file = new File(“e:\\”);

String str = file.list(new FilenameFilter(){

@Override

public boolean accept(File dir,String name){//这里的dir就是上面的file对象，name就

是文件或文件夹的名称

return new File(dir,name). isFile()&&name.ends.with(“.jpg”);//返回true就是放行，

返回false就是不放行

}

});

for(String s : str){//遍历数组

System.out.println(s);

}

public File[] listFiles(FilenameFilter filter)：过滤并返回指定目录下(一层)的文件和文件夹

的File数组，用法和上一个方法list(FilenameFilter filter)差不多。

## 56.递归

含义：方法定义中调用方法本身的现象。

注意：A：递归必须要有出口，否则就是死循环。

B：递归的次数不能太多，否则会内存溢出。

C：构造方法不能递归使用。

例子：1.递归求阶乘的递归方法

public static int jieCheng(int n){

if(n==1){

return 1;

}else{

return n\*jieCheng(n-1);

}

}

2.求斐波那契数列的第二十个数字

public static int fib(int n){

if(n == 1 || n == 2){

return 1;

}else{

return fib(n-1) + fib(n-2);

}

}

3.输出E:\\JavaSE下所有的java文件的绝对路径



# 集合

1.概念

对象的容器，定义了对多个对象进行操作的常用方法。可实现数组的功能。

2.和数组的区别

(1)数组长度固定，集合长度不固定。

(2)数组可以存储基本类型和引用类型，集合只能存储引用类型。

3.位置

java.util.\* 这个包里

## 1.Collection

1.Collection体系



2. Collection特点

代表一组任意类型的对象，无序、无下标、不重复。

3. Collection方法



boolean containsAll(Collection c) //如果此collection包含c中的全部元素，则返回true

boolean removeAll(Collection c) //将此collection和c共有的元素从此collection中移除

boolean retainAll(Collection c) //将此collection和c非共有的元素从此collection中移除

Iterator<E> iterator() //返回在此collection的元素上进行迭代的迭代器

4.使用迭代器遍历Collection

//创建集合  
Collection<String> collection = new ArrayList<>();  
collection.add("苹果");  
collection.add("西瓜");  
collection.add("榴莲");

/\*  
\* 迭代器遍历  
\* hasNext():判断有没有下一个元素，有的话返回true  
\* next():返回下一个元素  
\* remove():删除当前元素  
\* \*/  
Iterator it = collection.iterator();  
while(it.hasNext()){  
 System.*out*.println(it.next());  
 it.remove();//想删除元素，只能使用迭代器的remove()  
}  
System.*out*.println("当前元素个数："+collection.size());

## 2.List子接口

1.特点：有序、有下标、元素可重复。

2.List子接口的方法(首先包含Collection父接口的所有方法)



int indexOf(Object o)//返回列表中首次出现的指定元素的索引

int lastIndexOf(Object o)//返回列表中最后出现的指定元素的索引

[ListIterator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html)<[E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)> [listIterator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html#listIterator())()//返回此列表元素的列表迭代器（按适当顺序）。

[ListIterator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html)<[E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)> [listIterator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html#listIterator())(int index)//返回此列表元素的列表迭代器（按适当顺序），从列表的指定位置开始。

列表迭代器的功能更多，可以正向、逆向迭代集合元素，也可以迭代过程中增加、删除、替换元素以及获取元素的索引。

E remove(int index)//移除此列表中指定位置上的元素，返回被移除的元素。例：

List list = new ArrayList();

list.add(20);

list.add(30);

list.remove(30);//默认30为索引值，而不是数据。导致报错。

list.remove((Object)30); 或者list.remove(new Integer(30));来根据元素删除数据。

E set(int index,E element)//用指定元素替代此列表中指定位置上的元素，返回被替代的元素。

Object clone()//返回此ArrayList的浅拷贝副本

浅拷贝：没有创建元素，只是拥有和之前list一样的指向而已。所以如果修改了浅拷

贝的列表元素，那么原来的list的列表元素也会发生更改。但是如果用remove()

就不会有问题，因为remove()只是移除了指向，而没有真正删除内存中的列表

元素。

注意：List是可以添加一个或多个null的，null将作为一个对象添加到List里，List长度会随

之增加，同时List不再为empty。

3.列表迭代器([ListIterator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html))

方法：

void add([E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html) e)：将指定的元素插入列表

boolean hasNext()：判断是否有下一个元素

[E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html) next()：返回下一个元素

int nextIndex()：返回当前指向的元素的索引

boolean hasPrevious()：判断是否有上一个元素

[E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html) previous()：返回上一个元素

int previousIndex()：返回当前指向的元素的上一个元素的索引

void remove()：移除元素

void set([E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/ListIterator.html) e)：替换元素

补充：List在遍历时想要添加、替换、删除元素时，必须要用迭代器遍历，用迭代器对象来

添加、替换、删除元素，不能使用for循环，否则报错。hasNext()和next()可以用来正

向遍历list，hasPrevious()和previous()可以用来逆向遍历list，需要注意的是必须先正向

遍历完，才能逆向遍历。

### ArrayList

数组结构实现，查询快，增删慢；

JDK1.2版本，运行效率快，线程不安全

java中标准数组是定长的，当数组被创建出来之后，它们不能被加长或者缩短。

源码中根据Arrays.copyOf(原数组,新数组容量)方法进行数组的扩容创建。

扩容时机：当数组的大小大于初始容量的时候(比如初始为10，当添加第11个元素的

时候)，就会进行扩容。

扩容机制：newCap = oldCap + oldCap >> 1。右移一位是向下取整，比如：2.5向下取整是2；

-3.5向下取整是-4。若指定原容量为15，则扩容一次大小为：15+15/2=15+7=22。

源码分析：默认容量DEFAULT\_CAPACITY=10。

注意：集合中没有元素时，容量为0，用add()添加一个元素后，容量扩

为10，也就是被赋值默认容量。

存放元素的数组elementData，被transient修饰，表示不会被序列化和反序

列化。

实际元素个数size

去除list中重复元素的方法：

(1)使用新集合的方式，核心代码：

//旧集合  
List array = new ArrayList(10);  
//新集合  
List newArray = new ArrayList(array.size());  
Iterator it = array.listIterator();  
while(it.hasNext()){  
 String s = (String)it.next();  
 if(!newArray.contains(s)){  
 newArray.add(s);  
 }  
}

(2)使用选择排序的循环和equals()，核心代码：

List array = new ArrayList(10);  
for(int x=0; x<array.size()-1;x++){  
 for(int y=x+1;y<array.size();y++){  
 if(array.get(x).equals(array.get(y))){  
 array.remove(y);  
 y--;  
 }  
 }  
}

### Vector(过时)

数组结构实现，查询快、增删慢；

JDK1.0版本，运行效率慢、线程安全

### LinkedList

双向链表结构实现，增删快，查询慢

源码分析：实际元素个数size

头节点 first

尾节点 last

特有方法：

public void addFirst([E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) e)：将指定元素插入此列表的开头。

public void addLast([E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) e)：将指定元素添加到此列表的结尾。

public [E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) getFirst()：返回此列表的第一个元素。

public [E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) getLast()：返回此列表的最后一个元素。

public [E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) removeFirst()：移除并返回此列表的第一个元素。

public [E](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/LinkedList.html) removeLast()：移除并返回此列表的最后一个元素。

集合的remove(对象)、contains(对象)和indexOf(对象)方法都会调用对象的equals()方法。

假设有一个Student类，成员属性有String name和int age，同时也有他们的getXxx()方法，举例来重写该类的equals()方法(idea有生成这种方法的快捷键)：

@Override

public boolean equals(Object obj){

if(this==obj){

return true;

}

if(obj==null){

return false;

}

if(obj instanceof Student){

Student s = (Student)obj;

if(this.name.equals(s.getName())&&this.age==s.getAge()){

return true;

}

}

return false;

}

## 3.Set子接口

特点：无序、无下标、元素不可重复。

该接口中的方法全部继承自Collection父接口，自己没有新方法。

### HashSet

底层实现：哈希表。

HashSet其实是用的HashMap。

基于HashCode计算元素存放位置。

基于HashCode实现元素不重复。当存入元素的哈希码相同时，会调用equals()进行确认，

如结果为true，则拒绝后者存入。

可以使用增强for和迭代器Iterator遍历。

add()方法的存储逻辑：

1. 根据hashcode计算保存的位置，如果此位置为空，则直接保存，如果不为空执行 第二步。
2. 执行equals方法，如果equals方法为true，则认为是重复，否则，形成链表。

想要实现对象按照自身属性判断是否重复，需要重写hashCode和equals方法。idea和eclipse

都有快捷键。

在重写hashCode方法时，会用到31。31是一个质数，可以减少地址值不同，而hashCode

值相同的情况发生；用31可以提高计算效率，31\*i=(i<<5)-i。

### LinkedHashSet

底层实现：哈希表和链表。哈希表保证元素的唯一性，链表保证有序。

### TreeSet

底层实现：红黑树。

TreeSet其实是用的TreeMap。

可以使用增强for和迭代器Iterator遍历。

实现了SortedSet接口，对集合元素自动排序。

排序规则：

(1)存储的元素对象的类型必须实现Comparable接口，并实现CompareTo()指定排序规则。例：

@Override

public int compareTo(Person o){

int n1 = this.getName().compareTo(o.getName());

int n2 = this.age-o.getAge();

return n1==0?n2:n1;

}

当实现的CompareTo()返回0表示两个元素重复，大于0表示元素不重复且应放置在右边的

节点上，小于0也表示元素不重复，但应放在左边的节点上。

remove()也会调用实现的CompareTo()方法判断删除哪个对象。

(2)也可以在创建TreeSet对象的时候，使用匿名内部类来指定如何比较所存储的对象。如：



## 4.Map集合



特点：用于存储任意键值对(Key-Value)

键：无序、无下标、不允许重复(唯一)

值：无序、无下标、允许重复

常用方法：



void clear()：移除所有键值对。

boolean [containsKey](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#containsKey(java.lang.Object))([Object](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Object.html) key)：是否包含key的键值对。

boolean [containsValue](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#containsValue(java.lang.Object))([Object](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Object.html) value)：是否包含value的键值对。

[Set](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Set.html)<[Map.Entry](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.Entry.html)<[K](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html),[V](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html)>> [entrySet](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#entrySet())()：返回存储所有键值对的Set集合。该方法用于遍历Map集

合。效率更高。例：

Map<String,String> map = new HashMap();  
map.put("cn","中国");  
for(Map.Entry<String,String> entry : map.entrySet()){  
 System.*out*.println(entry.getKey()+"-------" +entry.getValue());  
}

boolean [isEmpty](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#isEmpty())()：是否为空。

[Set](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Set.html)<[K](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html)> [keySet](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#keySet())()：返回存储所有key的Set集合。该方法用于遍历Map集合。例：

Map<String,String> map = new HashMap();  
map.put("cn","中国");  
for(String key : map.keySet()){  
 System.*out*.println(key+"-------" +map.get(key));  
}

void [putAll](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#putAll(java.util.Map))([Map](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html)<? extends [K](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html),? extends [V](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html)> m)：将一个Map集合中的元素放到另一个Map集合中。

V remove(Object key)：根据key移除对应的键值对并返回那个移除的键值对的值。

int size()：返回键值对的个数。

注意：当添加的键值对的键有重复的时候，后来的键值对会覆盖之前的。

### HashMap

jdk1.2版本引入。线程不安全。效率高。key可以出现一次null，而value可以是null。

底层：哈希表(数组+链表+红黑树)

红黑树是jdk1.8以后加上去的。

put方法根据什么来验证添加的键值对的键是否重复呢？

是存储(键)对象的hashCode和equals方法。如果想改变键是否重复的判断，可以按自

己的想法来重写hashCode和equals方法。eclipse和idea都有重写的快捷方式。

[containsKey](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Map.html#containsKey(java.lang.Object))方法也会使用存储对象的hashCode和equals方法判断键是否存在。

源码分析：

默认初始化容量DEFAULT\_INITIAL\_CAPACITY = 1 << 4;

最大容量MAXIMUM\_CAPACITY = 1 << 30;

加载因子DEFAULT\_LOAD\_FACTOR=0.75f;

即键值对数量达到或超出容量的75%时就扩容。扩容时，newCap = oldCap << 1;

链表长度TREEIFY\_THRESHOLD = 8;

集合元素个数 MIN\_TREEIFY\_CAPACITY = 64;

jdk1.8-当链表长度大于8，并且集合元素个数大于等于64时，链表调整成红黑树。

链表长度 UNTREEIFY\_THRESHOLD = 6;

jdk1.8-当链表长度小于6时，将红黑树转回链表

哈希表里的数组transient Node<K,V>[] table;

数组存储起始Node节点，也就是存储链表的起始结点。

键值对个数 transient int size;

当使用put()方法添加第一个键值对时，HashMap的容量变为16。

jdk1.8之前链表是头插入，就像压子弹一样。之后是尾插入。采用头插入是考虑到后添

加的元素最有可能被检索，从而提高检索的效率；采用尾插入是因为jdk1.8降低了

哈希碰撞的概率，从而间接提高了数据检索的效率，尾插入更符合链表的插入顺序。

为什么HashMap的加载因子是0.75？

答：首先加载因子是根据泊松分布计算出来的，泊松分布是单位时间、单位面积、单位

长度内等相互独立的随机事件发生的概率，简单说是求随机事件发生的概率。0.75是从

内存的利用率和数据的访问效率上折中取的。当加载因子越大则数据密度越大，发生碰

撞的几率越高，数组中的链表也越长，这样会造成查询和插入时的数据比较次数增加，

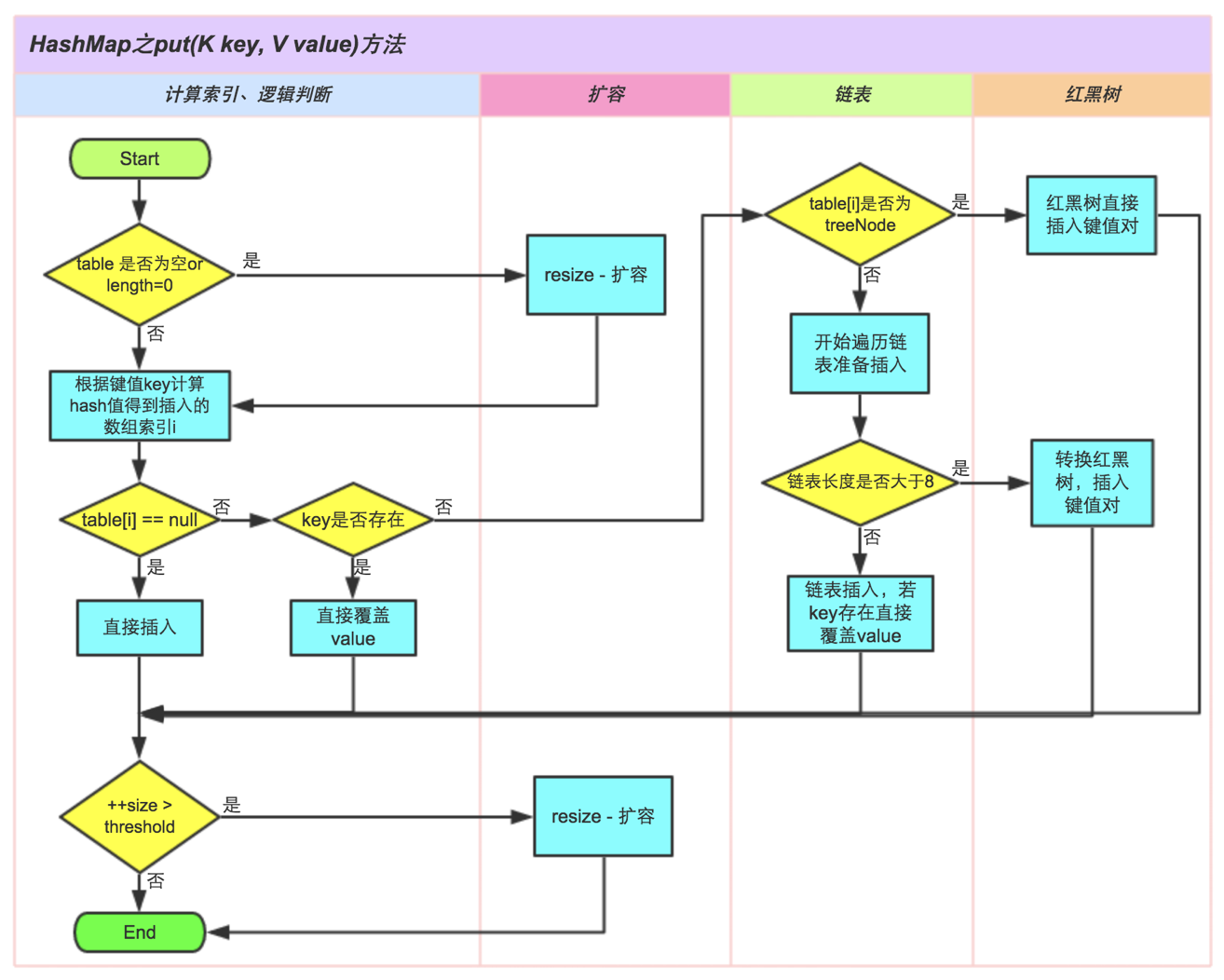
性能会下降。当加载因子越小，虽然数据密度越小，发生碰撞的几率也小，数组中的链

表也越短从而比较次数也减少，但是唯一的缺点是很容易扩容，扩容就会影响性能，即

便是初始化的时候给它大一点的空间，也有可能会浪费掉一部分内存。所以0.75是内

存利用率和数据访问效率的折中选择。

JDK1.8及之后版本的put方法详解顺序图：



补充：关于这个数组索引的计算看了源码是这样的。首先是HashMap的hash方法根据key

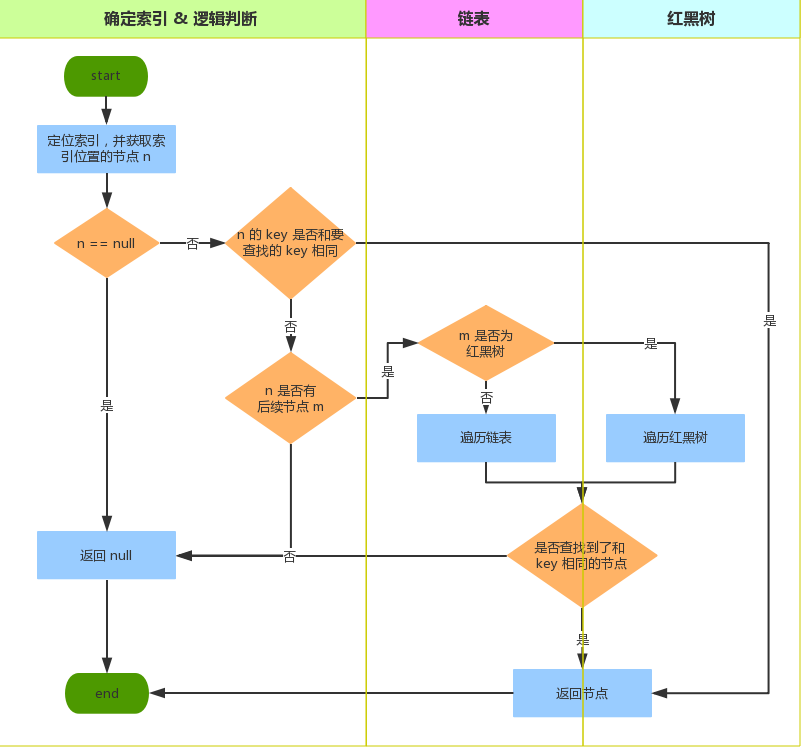
的hashCode()计算出一个新的hash值。然后putVal方法拿这个值会计算出一个table

数组的索引，再根据上图的流程将键值对存储起来。

public V put(K key, V value) {  
 return putVal(*hash*(key), key, value, false, true);  
}

static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

JDK1.8及之后版本的get方法详解顺序图：



### LinkedHashMap

说明：LinkedHashMap继承自HashMap，同时实现了Map接口。

底层：哈希表、链表。

实现：由哈希表保证键的唯一性；由链表保证有序(存储和取出时的顺序一致)

### TreeMap

底层：红黑树。

实现了SortedMap接口(是Map的子接口)，可以对key自动排序。

TreeMap要求键值对中的键对象的类实现Comparable接口并重写compareTo()来指定排

序规则。compareTo()返回int类型的数据，TreeMap会按照该数据由小到大排列元素。

put(key,value)会根据compareTo()方法来判断key是否重复。若key重复，则后来的value会覆盖之前的value。

remove(key)方法也会根据compareTo()方法来判断是否有对应的键值对。

contains(key)方法也会根据compareTo()方法来判断是否有对应的键值对。

TreeMap的遍历同样可以使用keySet()和entrySet()实现。

注意：创建TreeMap对象时，可以直接使用参数为匿名内部类的构造方法来指定排序构造器。不用到key对象的类中再去实现Comparable接口并重写compareTo()来指定排

序规则。这点和TreeSet是一样的。

## 5.Hashtable(过时)

jdk1.0版本引入，线程安全，效率慢。不允许null作为key或者value。

底层：哈希表(数组+链表)。

初始容量为11，加载因子为0.75。

它的方法跟HashMap的方法基本一致。

### Properties - 属性集合类

Hashtable的子类，该类没有泛型，要求key和value都是String。通常用于配置文件的读取。

该类比较特殊，其对象可以保存在流中或从流中读取。

构造方法：

public Properties()：创建一个无默认值的空属性列表。

public Properties([Properties](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/Properties.html) defaults)：创建一个带有指定默认值的空属性列表

方法：

public [Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) setProperty([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) key, [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) value)：添加元素。该方法底层是用的Hashtable

的put方法。

public [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) getProperty([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) key)：获取元素。该方法底层是用的Hashtable的get方法。

public [Set](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/Set.html)<[String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html)> stringPropertyNames()：获取所有键的集合。

public void load([InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html) inStream)：将文件中的键值对数据读取到集合中

public void load([Reader](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/Reader.html) reader)：将文件中的键值对数据读取到集合中

public void store([OutputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/OutputStream.html) out, [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) comments)：把集合中的数据存储到文件，

comments是文件中的描述信息。

public void store([Writer](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/Writer.html) writer, [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) comments)

# Colletions工具类

概念:集合工具类，定义了除了存取以外的集合常用方法。

方法：



public static <T> void sort([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<T> list,[Comparator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Comparator.html)<? super T> c)//升序排序

注意：当元素即实现了Comparable接口，又用了sort([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<T> list,[Comparator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Comparator.html)<? super

T> c)方法的时候，排序规则按sort([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<T> list,[Comparator](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Comparator.html)<? super T> c)来。

static <T> int [binarySearch](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Collections.html#binarySearch(java.util.List, T))([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<? extends [Comparable](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Comparable.html)<? super T>> list, T key) //二分查找，前提是list是排好序的。

static <T> int [binarySearch](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Collections.html#binarySearch(java.util.List, T))([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<? extends [Comparable](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Comparable.html)<? super T>> list, T key, Comparator<? super T> c) //二分查找，前提是list是排好序的。第三个参数是一个比较器。

static <T> void [copy](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Collections.html#copy(java.util.List, java.util.List))([List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<? super T> dest, [List](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/List.html)<? extends T> src) 复制src到dest中，前提是dest的容量要大于等于src的容量。

public static <T> T max(Collection<?> coll)：获取最大值。

补充：

1.list转数组。例：

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

list.add(“haha”);

String[] array = list.toArray(new String[list.size()]);

2.数组转list。例：

String[] name = {“张三”,”里斯”};

List<String> list = Arrays.asList(names);

注意：该list不能添加和删除元素。因为其本质仍然是数组，而数组长度是固定的，无

法修改。

# IO流

## 1.初识IO流

作用：处理设备之间的数据传输，文件的上传和下载。

分类：

按流向分：

输入流-读取数据

输出流-写出数据

按数据类型分：

字节流-操作任意类型数据

字符流-操作文本数据

字节输入流 - 抽象基类 - InputStream

字节输出流 - 抽象基类 - OutputStream

字符输入流 - 抽象基类 - Reader

字符输出流 - 抽象基类 - Writer

上面的抽象基类的子类都是以父类名作为后缀名，比如：

XxxInputStream、XxxOutputStream、XxxReader、XxxWriter

(1)FileOutputStream - 文件输出字节流

作用：用于写入诸如图像数据之类的原始字节的流。

构造方法：

public FileOutputStream([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file)：根据File对象创建对象

public FileOutputStream(String name)：根据文件的路径创建对象

public FileOutputStream([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file,boolean append)：根据File对象创建对象，可以追加写

入文件数据

public FileOutputStream(String name,boolean append)：根据文件的路径创建对象，可以

追加写入文件数据

以上构造方法如果未指定文件的盘符，默认根路径为当前项目路径，且如果文件不存

在的话，会自动创建一个文件出来。

方法：

public void write(int b)：写入一个字节到文件中

FileOutputStream os = new FileOutputStream(“a.txt”);

os.write(97);//实际显示的是a

os.close();

public void write(byte[] b)：将字节数组b写入到文件中。

public void write(byte[] b, int off, int len)：将字节数组的一部分写入到文件中。

表示从数组的下标off开始写入len个到文件中。

public void close()：关闭此文件输出流并释放与此流有关的所有系统资源。

关闭流的意义：

(1)让流对象变成垃圾，便于回收

(2)通知系统释放与文件相关的资源

提示：一般流的关闭会放在try...catch...finally的finally语句中

问题1：如何让写入文件中的数据换行？

答：写入换行符号即可。例如：

FileOutputStream os = new FileOutputStream(“a.txt”);

os.write(“hello1”.getBytes());

os.write(“\n”.getBytes());// \n就是一种换行符号

os.close();

补充：windows的换行符是：\r\n，linux是\n，Mac是\r，一些高级记事本可以识别任

意换行符号。

问题2：如何追加写入文件数据？

使用对应的两种构造方法中的一个就可以。

一个标准的处理流的代码：



(2) FileInputStream - 文件输入字节流

作用：用于读取诸如图像数据之类的原始字节的流。

构造方法：

public FileInputStream([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file)：根据File对象创建对象

public FileInputStream([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name)：根据文件的路径创建对象

方法：

public int read()：一次读取一个字节，并移动到下一个字节位置，移动的逻辑类似于迭

代器的判断方法。如果已到达文件末尾，则返回-1。

public int read(byte[] b)：从输入流中一次读取一个字节数组长度的内容到字节数组中。

并移动到下一个字节位置，移动的逻辑类似于迭代器的判断方法。如果已到达文

件末尾，则返回-1，否则返回字节数组中当前读取的字节个数。

注意：每次读取都会覆盖掉字节数组中的内容，如果字节数组的长度是5，第一次读

了5个字节到字节数组中，而第二次只读到了3个字节，则只会覆盖字节数组中

前三个字节，后两个不会被覆盖。

public int read(byte[] b,int off,int len)：从输入流中将最多len个字节读入到一个字节数

组中，如果len为0，则不读取任何数据并返回0，否则返回当前读取的字节个数。

public void close()：关闭此文件输入流并释放与此流有关的所有系统资源。

提示：计算机中，中文字符由两个字节组成，左边的字节必定是个负数，右边的不一定。当

计算机读取到负数时，认定是中文字节，就会将当前的字节和下一个字节拼接成汉字。

当用字节输入流和字节输出流复制文件内容时，不需要专门处理汉字字符。

代码示例：控制台打印一个文件中的内容

FileInputStream fis = new FileInputStream(“a.txt”);

byte[] bys = new byte[1024];

int len = 0;

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

System.out.print(new String(bys,0,len));

}

fis.close();

文件复制的代码：



(3)BufferedOutputStream - 字节缓冲输出流

作用：高效率的写入文件内容

构造方法：

public BufferedOutputStream([OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) out)：使用默认的缓冲区大小创建对象。

public BufferedOutputStream([OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) out,int size)：使用指定的缓冲区大小size创

建对象。

问题：为什么构造方法需要传递[OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html)对象而不是一个文件路径呢？

因为BufferedOutputStream仅仅负责提供缓冲区，真正的读写操作还是得靠基本

的流对象来实现。

方法：

public void write(int b)：一次写一个字节到文件中。

public void write(byte[] b,int off,int len)：将流中off下标开始写入到字节数组中len个字

节。

public void write(byte[] b)：该方法继承自FileOutputStream。

public void flush()：刷新缓冲区。

public void close()：该方法继承自FileOutputStream。

代码示例：

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(“b.txt”));

bos.write(“hello”.getBytes());

bos.close();

(4)BufferedInputStream - 字节缓冲输入流

作用：高效率的读取文件内容

构造方法：

public BufferedInputStream([InputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/InputStream.html) in)

public BufferedInputStream([InputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/InputStream.html) in,int size)

方法：

public int read()

public int read(byte[] b,int off,int len)

public int read(byte[] b)：继承自FileInputStream

public void close()

(5)DataOutputStream - 数据输出流(了解)

作用：允许将基本java数据写入输出流中，然后程序可以使用数据输入流DataInputStream

读取数据。

构造方法：

public DataOutputStream([OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) out)

方法：

public final void writeBoolean(boolean v)

public final void writeByte(int v)

public final void writeBytes([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) s)

public final void writeChar(int v)

public final void writeChars([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) s)

public final void writeDouble(double v)

public final void writeFloat(float v)

public final void writeInt(int v)

public final void writeLong(long v)

public final void writeShort(int v)

(6)DataInputStream - 数据输入流(了解)

作用：读取由数据输出流DataOutputStream写入的数据。

构造方法：

public DataInputStream([InputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/InputStream.html) in)

方法：

public final boolean readBoolean()

public final byte readByte()

public final char readChar()

public final double readDouble()

public final float readFloat()

public final int readInt()

public final long readLong()

public final short readShort()

## 2.编码表



## 3.字符串中的编码问题

String类中的方法：

public String(byte[] bytes)：使用平台的默认字符集将解码字节数组为字符串。window

平台的默认字符集是GBK，而linux平台的默认字符集是UTF-8。

public String(byte[] bytes,String charsetName)：通过指定的字符集charsetName解码字节

数组。

public byte[] getBytes()：使用平台的默认字符集将字符串编码为字节数组。window平

台的默认字符集是GBK，而linux平台的默认字符集是UTF-8。

public byte[] getBytes([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) charsetName)：使用指定的字符集charsetName将字符串编

码为字节数组。

## 4.IO流中的字符流

(1) OutputStreamWriter - 字符输出流

构造方法：

public OutputStreamWriter([OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) out)：使用默认字符编码创建对象。

public OutputStreamWriter([OutputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) out,String charsetName)：使用指定字符编码创

建对象。

方法：

pulic void write(char[] char,int off,int len)：写入字符数组的一部分。

public void write(int c)：写入单个字符。

public void write(String str,int off,int len)：写入字符串的某一部分。

public void write(char[] cbuf)：写入字符数组。

public void write(String str)：该方法继承自Writer。向文件写入字符串str。

public void flush()：刷新缓冲区。

public void close()：关闭该流，但关闭前内部会先调用flush()刷新缓冲区。

注意：字符输出流和字节输出流有些区别，最大的区别是使用write系列方法时，字节输出

流是直接一个一个字节写到文件中的，不用经过内存临时匹配；而字符输出流的底层

用的是字节流，导致字节需要先在内存中缓存起来，匹配成字符，再使用flush()方法

将缓存中的字符写到文件中同时清空缓存，所以导致字符输出流使用write系列方法后

如果不使用flush()，是无法把缓存中的字符写到文件中的。值得一提的是，字符输出

流的close()会默认先调用flush()再关闭字符输出流，因此如果数据量不大的话，不需

要使用flush()来刷新缓冲区。

(2)InputStreamReader - 字符输入流

构造方法：

public InputStreamReader ([InputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) in)：使用默认字符编码创建对象。

public InputStreamReader ([InputStream](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/OutputStream.html) in,String charsetName)：使用指定字符编码创

建对象。

方法：

public int read()：读取单个字符并指向下个字符，若已到达流的末尾，则返回-1。

public int read(char[] cbuf,int off,int len)：读取字符数组的一部分，并指向下一部分，若

已到达流的末尾，则返回-1，否则返回实际读取的字符数。

public int read(char[] cbuf)：读取字符数组，若已到达流的末尾，则返回-1。该方法继承

自Reader。

public void close()：关闭流。

(3)FileWriter - 便捷的字符输出流

说明：该类是OutputStreamWriter的子类，为写入字符文件的便捷类。

构造方法：

public FileWriter([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file)：根据File对象创建对象。

public FileWriter([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file,boolean append)：根据File对象创建可追加写入的对象。

public FileWriter([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) fileName)：根据文件路径创建对象。

public FileWriter([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) fileName,boolean append)：根据文件路径创建可追加写入的对象

方法：全部继承自父类OutputStreamWriter。

(4)FileReader - 便捷的字符输入流

说明：该类是InputStreamReader的子类，为读取字符文件的便捷类。

构造方法：

public FileReader([File](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/File.html) file)：根据File对象创建对象。

public FileReader([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) fileName)：根据文件路径创建对象。

方法：全部继承自父类InputStreamReader。

(5)BufferedWriter - 字符缓冲输出流

说明：专门带缓冲区，高效率的字符输出流

构造方法：

public BufferedWriter([Writer](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Writer.html) out)：使用Writer对象创建对象。

public BufferedWriter([Writer](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Writer.html) out,int size)：使用Writer对象和指定的缓冲区大小创建对

象。

方法：

pulic void write(char[] char,int off,int len)：写入字符数组的一部分。

public void write(int c)：写入单个字符。

public void write(String str,int off,int len)：写入字符串的某一部分。

public void write(char[] cbuf)：写入字符数组。

public void write(String str)：该方法继承自Writer。向文件写入字符串str。

public void flush()：刷新缓冲区。

public void close()：关闭该流，但关闭前内部会先调用flush()刷新缓冲区。

public void newLine()：写入一个换行符，换行符由系统决定。此方法可以和

BufferedReader的readLine()配合使用。

(6)BufferedReader - 字符缓冲输入流

说明：专门带缓冲区，高效率的字符输入流

构造方法：

public BufferedReader([Reader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Reader.html) in)：使用Reader对象创建对象。

public BufferedReader([Reader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Reader.html) in,int size)：使用Reader对象和指定的缓冲区大小创建对

象。

方法：

public int read()：读取单个字符并指向下个字符，若已到达流的末尾，则返回-1。

public int read(char[] cbuf,int off,int len)：读取字符数组的一部分，并指向下一部分，若

已到达流的末尾，则返回-1，否则返回实际读取的字符数。

public int read(char[] cbuf)：读取字符数组，若已到达流的末尾，则返回-1。该方法继承

自Reader。

public void close()：关闭流。

public String readLine()：读取一行数据并指向下一行，若已到达流的末尾，则返回null。

根据字符串中的换行符、回车符或者回车符紧跟换行符来判断一行数据。注意：

正常返回一行数据的时候，是不包括换行符或者回车符的。此方法可以和

BufferedWriter的newLine()配合使用。

(7)字符缓冲流复制文本内容代码示例



(8)跟踪行号的缓冲字符输入流 - LineNumberReader

说明：该类是[java.io.BufferedReader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/BufferedReader.html)的子类，特殊方法为设置和获取行号。

构造方法：

public LineNumberReader([Reader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Reader.html) in)：根据Reader创建对象。

public LineNumberReader([Reader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/io/Reader.html) in,int size)：根据Reader和指定的缓冲区大小创建对象

特有方法：

public int getLineNumber()：获取行号。

public void setLineNumber(int lineNumber)：设置行号从几开始。

## 5.内存操作流(了解)

含义：用于处理临时数据，程序结束，数据从内存中消失。

(1)ByteArrayOutputStream操作字节数组的字节输出流

作用：此类实现了一个输出流，其中的数据被写入一个byte数组，缓冲区会随着数据的不

断写入而自动增长，可使用toByteArray()和toString()获取数据。此类不需要用close()

关闭。

构造方法：

public ByteArrayOutputStream()：创建对象，缓冲区大小为32个字节。

public ByteArrayOutputStream(int size)：创建对象，缓冲区大小为指定的size个字节。

(2)ByteArrayInputStream操作字节数组的字节输入流

作用：该类包含一个内部缓冲区，该区存储从流中读取的字节。该类不需要用close()关闭。

构造方法：

public ByteArrayInputStream(byte[] buf)：创建对象，buf作为其缓冲区数组。

public ByteArrayInputStream(byte[] buf,int offset,int length)：

使用案例：

ByteArrayOutputStream baos = new ByteArrayOutputStream();

for(int x = 0; x < 10; x++){baos.write((“hello”+x).getBytes());}

byte[] bys = baos.toByteArray();

ByteArrayInputStream bais = new ByteArrayInputStream(bys);

int by = 0;

while((by = bais.read()) != -1){System.out.println((char)by);}

(3)CharArrayReader - 操作字符数组的字符输入流

(4)CharArrayWriter - 操作字符数组的字符输出流

(5)StringReader - 操作字符串的字符输入流

(6)StringWriter - 操作字符串的字符输出流

补充：这六个类，三组流，其使用方法基本相同。

## 6.打印流

特点：

a.只能输出，没有对应的输入流

b.可以操作任意类型的数据

c.如果启动了自动刷新，则有些方法不需要再手动调flush()方法刷新缓冲区

d.可以直接操作文件

(1)PrintStream - 字节打印流

作用：打印各种类型的数据。

方法：

public void println()：根据平台输出换行符。

(2)PrintWriter - 字符打印流

作用：打印各种类型的数据。该类底层使用BufferedWriter(高效的字符输出流)。

构造方法：

public PrintWriter([Writer](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/Writer.html) out, boolean autoFlush)：创建对象，若autoFlush为true，则

println、printf、format方法将刷新缓冲区。

## 7.标准输入输出流

System类中的两个静态成员变量：

public static final InputStream in - 标准输入流

public static final PrintStream out - 标准输出流

说明：

PrintStream ps = System.out;

ps.println(“helloworld”);

键盘录入数据的三种方式：

(1)使用main方法的args接收参数

(2)使用Scanner接收参数(jdk1.5出现)

(3)通过字符缓冲流包装标准输入流实现

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

## 8.RandomAccessFile - 随机访问流

说明：该流的父类是object，兼具读和写的功能。

构造方法：

public RandomAccessFile([File](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/File.html) file, [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) mode)：创建对象，mode为操作文件的模式，

模式为”rw”最常用，表示即可以写数据，也可以读数据。

public RandomAccessFile([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name, [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) mode)：创建对象，mode为操作文件的模

式，模式为”rw”最常用，表示即可以写数据，也可以读数据。

方法：

public long getFilePointer()：返回读取到该文件中的偏移量。

public void seek(long pos)：设置读取到该文件中的偏移量。

## 9.SequenceInputStream - 合并流

作用：表示多个输入流的集合。

构造方法：

public SequenceInputStream([InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html) s1, [InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html) s2)

public SequenceInputStream([Enumeration](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/Enumeration.html)<? extends [InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html)> e)

案例：

(1)将两个文件复制到一个文件中



(2)将三个文件(或更多)复制到一个文件中



## 10.序列化流和反序列化流

序列化流：把对象按照流一样的方式存入文本文件或者在网络中传输。对象 -- 流数据

(ObjectOutputStream)

反序列化流：把文本文件中的流对象数据或者网络中的流对象数据还原成对象。流数据 --对象(ObjectInputStream)

如果一个对象要被序列化或者反序列化，必须实现序列化接口 - Serializable。Serializable接口中没有任何方法，是一个标记接口。

(1) ObjectOutputStream - 序列化流

构造方法：

public ObjectOutputStream([OutputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/OutputStream.html) out)

方法：

public final void writeObject([Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) obj)：以流的方式将对象写入到文件中。

public void close()：关闭流

(2) ObjectInputStream - 反序列化流

构造方法：

public ObjectInputStream([InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html) in)

方法：

public final [Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) readObject()：读取文件中流形式的对象数据

public void close()：关闭流

(3)详解序列化接口 - Serializable

当一个类实现序列化接口 - Serializable时，类中会产生一个默认的序列化标记值，即serialVersionUID，如果类中发生了变化，该标记值也会发生变化。当对象以流的形式写入文件中时会带着该标记值，读取以流形式存在文件中的对象时，也会读取该标记值，一旦读取的标记值和类的标记值不同，便会报错。

所以可以直接指定serialVersionUID的值，防止其发生改变，如：



(4)如何让对象的成员变量不被序列化

只需要让成员变量被transient关键字修饰即可。不被序列化意味着该成员变量的赋值不会

被传递，则会被赋予默认值，即byte,short,int,long是0，float,double是0.0，char是’\u0000’，boolean是false，引用类型是null。

## 11.字节流和字符流的继承关系图

(1)字节输入流



(2)字节输出流



(3)字符输入流



(4)字符输出流



# NIO(New IO)(了解)

说明：JDK1.4引入的NIO，为新IO，可以替代标准的IO的API，它与IO有同样的作用和目

的，但是使用方式完全不同。NIO 支持面向[缓冲区](https://www.zhihu.com/search?q=%E7%BC%93%E5%86%B2%E5%8C%BA&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22369062109%22%7D)的、基于通道的 IO 操作。NIO 将

以更加高效的方式进行文件的读写操作。

NIO与IO的区别：



Paths(jdk1.7后出现)类中的方法：

public static [Path](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/file/Path.html) get([URI](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/net/URI.html) uri)

Files(jdk1.7后出现)类中的方法：

public static long copy([Path](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/file/Path.html) source,[OutputStream](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/io/OutputStream.html) out)：复制文件

public static [Path](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/file/Path.html) write([Path](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/file/Path.html) path,[Iterable](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Iterable.html)<? extends

[CharSequence](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/CharSequence.html)> lines,[Charset](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/charset/Charset.html) cs,[OpenOption](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/nio/file/OpenOption.html)... options)：将集合lines中的数据根据

编码cs写入到path表示的文件中去。

案例代码：复制文件



案例代码：将集合中的数据根据编码写入到文件中



# 正则表达式

含义：符合一定规则的字符串

正则表达式的规则：可以查看jdk的文档中的Pattern类，里面有所有的规则描述。

常用规则：

A：字符

x 字符x。举例：’a’表示字符a

\\ 反斜线字符

\n 新行(换行)符 (‘\u000A’)

\r 回车符 (‘\u000D’)

B：字符类

[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)

[0-9] 0到9的字符都包括

C：预定义字符类

. 任何字符。\.表示.本身

\d 数字，[0-9]

\s 空白符

\S 非空白符

\w 单词字符，[a-zA-Z\_0-9]

在正则表达式里面组成单词的东西必须有这些东西组成。

\W 非单词字符

D：边界匹配器

^ 行的开头

$ 行的结尾

\b 单词边界，指的是 字符串中不是单词字符的地方

\B 非单词边界

E：Greedy 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

方法：

(1)String类的public boolean matches(String regex)：判断是否匹配规则regex

String phone = “13766668888”;

String regex = “1[38]\\d{9}”;

boolean flag = phone.matches(regex);

(2)String类的public String[] split(String regex)：根据规则regex分割字符串

String s1 = “aa,bb,cc”;

String [] arrayStr = s1.split(“,”);//分割,

s1 = “aa.bb.cc”;

arrayStr = s1.split(“\\.”);//分割.

s1 = “aa bb cc”;

arrayStr = s1.split(“ +”);//分割多个空格符

s1 = “aa\\bb\\cc”;

arrayStr = s1.split(“\\\\”);//分割\\

(3)String类的public String replaceAll(String regex,String replacement)：根据给定的

replacement替换此字符串所有匹配给定的正则表达式的子字符串。

String s = “hello465131jdlsjfd”;

String regex = “\\d+”;

String replacement = “\*”;

String result = s.replaceAll(regex, replacement);//将数字转换为\*，返回一个新字符串

结果是hello\*jdlsjfd

(4)Pattern类和Matcher类中的方法

两个类中的方法同样可以实现判断正则的功能：

// Pattern.[CASE\_INSENSITIVE](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/regex/Pattern.html#CASE_INSENSITIVE)实现匹配不区分大小写

Pattern p = Pattern.compile(“a\*b”, Pattern.[CASE\_INSENSITIVE](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/regex/Pattern.html#CASE_INSENSITIVE));

Matcher m = p.matcher(“aaab”);

boolean b = m.matches();//判断是否匹配给定的正则

从一个大字符串中获取符合正则规则的子串：

String s = “da jia ting wo shuo,jin tian yao”;

String regex = “\\b\\w{3}\\b”;// 规则-连续三个单词字符

Pattern p = Pattern.compile(regex);

Matcher m = p.matcher(s);

while(m.find()){//判断是否有下一个匹配正则的子序列

//必须先调用find()，再用group()，否则报错

System.out.println(m.group());//输出匹配的子序列

}

# 泛型(JDK1.5出现)

格式：

<数据类型>，此处的数据类型只能是引用类型

优点：

(1)把运行时期的问题提前到编译期间。

(2)避免了强制类型转换。

(3)优化程序设计，提高代码复用性。

泛型类：

把泛型定义在类上

格式：public class 类名<泛型类型1,…>

泛型方法：

把泛型定义在方法上

格式：public <泛型类型> 返回类型 方法名(泛型类型)

泛型接口：

把泛型定义在接口上

格式：public interface 接口名<泛型类型1,…>

泛型高级(通配符)：

<?>：表示任意的引用类型

<? extends E>：向下限定，E及其子类

<? super E>：向上限定，E及其父类

# JDK7的新特性(了解)

## 1.二进制字面量

在二进制数之前加0b或者0B就可以表示数字为二进制。

举例：int x = 0b110110;

## 2.数字字面量可以出现下划线

说明：为了增强对数值的阅读性，如我们经常把数据用逗号分隔一样。JDK7提供了\_对数据

分隔。

举例：int x = 100\_1000;

注意：下划线不能出现在进制标识和数值之间。

下划线不能出现在数值开头和结尾。

下划线不能出现在小数点旁边。

## 3.try-with-resource

作用：自动关闭需要释放资源的对象。

格式：

try(必须是java.lang.AutoCloseable的子类对象){…}

示例：

try(FileReader fr = new FileReader(“a.txt”);

FileWriter fw = new FileWriter(“b.txt”);){

int ch = 0;

while((ch = fr.read()) != -1){

fw.write(ch);

}

}catch(IOException e){

e.printStackTrace();

}

注意：IO流相关类基本都实现了AutoCloseable接口。

# JDK8的新特性

## 1.Lambda表达式

含义：是一个匿名函数，可以理解为一段可以传递的代码。示例：

//原来的匿名内部类  
Comparator<Integer> com1 = new Comparator<Integer>(){  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 return Integer.*compare*(o1, o2);  
 }  
};  
  
//使用Lambda  
Comparator<Integer> com2 = (x,y) -> Integer.*compare*(x,y);

1.Lambda基础

(1) Lambda表达式需要"函数式接口"的支持。

函数式接口：接口中只有一个抽象方法的接口，称为函数式接口。

可以使用注解@FunctionalInterface修饰来检查是否是函数式接口。

(2)Lambda表达式的基础语法：Java8中引入了一个新的操作符"->"，该操作符称为箭头操作符或Lambda操作符

箭头操作符将Lambda表达式拆分成两部分：

左侧：Lambda 表达式的参数列表，即接口中抽象方法的参数列表

右侧：Lambda表达式中所需执行的功能，即重写方法的主体

语法格式一：无参数，无返回值

() -> System.out.println( "Hello Lambda ! ");

示例：

final int num = 0;//jdk1.7之前，必须写final。jdk1.8之后默认会加上  
Runnable r = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Hello World!"+num);  
 }  
};

r.run();  
System.*out*.println("----------------------------");  
  
Runnable r1 = () -> System.*out*.println("Hello World!"+num);  
r1.run();

语法格式二：有一个参数，并且无返回值

(×) -> System.out.println(x)

示例：

@Test  
public void test2(){  
 Consumer<String> con1 = new Consumer<String>() {  
 @Override  
 public void accept(String x) {  
 System.*out*.println(x);  
 }  
 };  
 con1.accept("Lambda1");  
 Consumer<String> con2 = (x) -> System.*out*.println(x);  
 con2.accept("Lambda2");  
}

语法格式三：若只有一个参数，小括号可以省略不写

x -> system.out.println(x)

Consumer<String> con = x -> System.*out*.println(x);  
con.accept("Lambda");

语法格式四：有两个以上的参数，有返回值，并且Lambda体中有多条语句

Comparator<Integer> com = (x,y) -> {  
 System.*out*.println("函数时接口");  
 return Integer.*compare*(x,y);  
};

语法格式五：若Lambda体中只有一条语句,return和大括号都可以省略不写

Comparator<Integer> com = (x,y) -> Integer.*compare*(x,y);

语法格式六：Lambda表达式的参数列表的数据类型可以省略不写，因为JVN编译器通过上下文推断出数据类型，即"类型推断"。”类型推断”发生在编译期，因此只是会稍微增加项目编译的时间，对项目运行没有效率影响。

Comparator<Integer> com = (Integer x,Integer y) -> Integer.*compare*(x,y);

2.java8内置的四大核心函数式接口

出现原因：

使用Lambda表达式之前必须要一个函数式接口的支持，自己每次实现比较麻烦。

(1) **Consumer<T>** : 消费型接口

void accept(T t);

示例：

@Test  
public void test8(){  
 happy(10000, (m) -> System.*out*.println("消费：" + m + "元"));  
}  
public void happy(double money, Consumer<Double> con){  
 con.accept(money);  
}

(2) **Supplier<T>** : 供给型接口

T get();

示例：

@Test  
public void test9(){  
 List<Integer> numList = getNumList(10, () -> (int)(Math.*random*() \* 100));  
 for (Integer num : numList) {  
 System.*out*.println(num);  
 }  
}  
  
//需求：产生指定个数的整数，并放入集合中  
public List<Integer> getNumList(int num, Supplier<Integer> sup){  
 List<Integer> list = new ArrayList<>();  
 for (int i = 0; i < num; i++) {  
 Integer n = sup.get();  
 list.add(n);  
 }  
 return list;  
}

(3)**Function<T, R>** : 函数型接口

R apply(T t);

@Test  
public void test10(){  
 String subStr = strHandler("一二三四五六", (str) -> str.substring(2, 5));  
 System.*out*.println(subStr);  
}  
//需求：用于处理字符串  
public String strHandler(String str, Function<String, String> fun){  
 return fun.apply(str);  
}

(4)**Predicate<T>** : 断言型接口

boolean test(T t);

@Test  
public void test11(){  
 List<String> list = Arrays.*asList*("Hello", "atguigu", "Lambda", "www", "ok");  
 List<String> strList = filterStr(list, (s) -> s.length() > 3);  
 for (String str : strList) {  
 System.*out*.println(str);  
 }  
}  
//需求：将满足条件的字符串，放入集合中  
public List<String> filterStr(List<String> list, Predicate<String> pre){  
 List<String> strList = new ArrayList<>();  
 for (String str : list) {  
 if(pre.test(str)){  
 strList.add(str);  
 }  
 }  
 return strList;  
}

(5)其他接口



3.方法引用

若Lambda体中的内容(比如下方的println方法和getAge方法)有类已经实现了，我们可以

使用"方法引用"(可以理解为方法引用是Lambda表达式的另外一种表现形式)。主要有

三种语法格式。

(1) 对象::实例方法名

@Test  
public void test1(){  
 Consumer<String> con1 = (x) -> System.*out*.println(x);  
 //前提是println这个方法的返回值类型和形参列表与accept方法保持一致才能

正常执行  
 Consumer<String> con2 = System.*out*::println;  
 con1.accept("hello1");  
 con2.accept("hello2");  
}

@Test  
public void test2(){  
 Employee emp = new Employee();

Supplier<Integer> sup1 = () -> emp.getAge();

//前提是getAge这个方法的返回值类型和形参列表与get方法保持一致才能

正常执行  
 Supplier<Integer> sup = emp::getAge;  
 Integer num = sup.get();  
 System.*out*.println(num);  
}

(2) 类::静态方法名

@Test  
public void test3(){  
 Comparator<Integer> com1 = (x,y) -> Integer.*compare*(x,y);  
 Comparator<Integer> com2 = Integer::*compare*;  
 System.*out*.println(com2.compare(8,4));  
}

(3) 类::实例方法名

@Test  
public void test4(){  
 BiPredicate<String,String> bp1 = (x,y) -> x.equals(y);  
 //equals是String类中的方法  
 BiPredicate<String,String> bp2 = String::equals;  
}

注意：仅当实现方法的第一个参数是实例方法的调用者，比如x，x调用equals；而第二个

参数是实例方法的参数时，比如y，y是equals方法的参数，才可以用使用 类::实例方

法名的格式。

4.构造器引用

格式：

ClassName::new

@Test  
public void test5(){  
 Supplier<Employee> sup1 = () -> new Employee();  
 //自动根据抽象方法的形参选择构造器方法  
 Supplier<Employee> sup2 = Employee::new;  
 Employee emp = sup2.get();  
 System.*out*.println(emp);  
}

@Test  
public void test6(){  
 Function<Integer,Employee> fun1 = (x) -> new Employee(x);  
 //根据接口Function中的唯一一个抽象方法的形参来选择对应的构造器方法  
 Function<Integer,Employee> fun2 = Employee::new;  
 Employee emp = fun2.apply(101);  
 System.*out*.println(emp);  
  
 //这个则是对应俩形参的构造器方法  
 BiFunction<Integer, Integer, Employee> bf = Employee::new;  
}

5.数组引用

格式：

Type::new

@Test  
public void test7(){  
 Function<Integer,String[]> fun1 = (x) -> new String[x];  
 String[] str1 = fun1.apply(10);  
 System.*out*.println(str1.length);  
  
 Function<Integer,String[]> fun2 = String[]::new;  
 String[] str2 = fun2.apply(20);  
 System.*out*.println(str2.length);  
}

## 2.StreamAPI

1.介绍

流(Stream)是数据渠道，用于操作数据源(集合、数组等)所生成的元素序列。

注意：

Stream自己不会存储元素。

Stream不会改变源对象。相反，他们会返回一个持有结果的新Stream。

Stream操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

Stream的三个操作步骤：

1.创建Stream

2.中间操作

3.终止操作（终端操作）

2.创建Stream

(1) 可以通过Collection系列集合提供的stream()(顺序流)或parallelStream()(并行流)

List<String> list = new ArrayList<>();  
Stream<String> stream1 = list.stream();

(2) 通过Arrays中的静态方法stream()获取数组流

Employee[] emps = new Employee[10];  
Stream<Employee> stream2 = Arrays.*stream*(emps);

(3) 通过Stream类中的静态方法of()

Stream<String> stream3 = Stream.*of*("aa","bb","cc");

(4) 创建无限流

//迭代  
Stream<Integer> stream4 = Stream.*iterate*(0,(x) -> x + 2);  
stream4.limit(10).forEach(System.*out*::println);  
//生成  
Stream.*generate*(() -> Math.*random*())  
 .limit(5)  
 .forEach(System.*out*::println);

3.中间操作

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为”惰性求值”。

(1)筛选与切片

Stream<T> filter(Predicate<? super T> predicate)：接收Lambda ,从流中排除某些元素

补充：从下图代码中可以看到Lambda表达式有一个形参e，e是一个Employee对象，只用

形参e是无法完成对集合employees中所有元素的过滤的。其实filter方法内部实现了

迭代器机制，可以对集合进行内部迭代来保证过滤集合中的所有元素。

@Test  
public void test1(){  
 //中间操作，不会执行任何操作  
 Stream<Employee> stream = employees.stream()  
 .filter((e) -> {  
 System.*out*.println("Stream API 的中间操作");  
 return e.getAge() > 35;  
 });  
 //终止操作，一次性执行全部内容  
 stream.forEach(System.*out*::println);  
}

Stream<T> limit(long maxsize)：截断流，使其元素不超过给定数量。

补充：看下图代码。因为有limit(2)，也就是只取符合条件的前两个元素。为了提高运行的

效率，java在使用filter方法过滤元素时，如果已经有两条符合条件的元素，那么

employees集合中的其他元素将不会再进行过滤以提高运行的效率，这就叫做短路。

@Test  
public void test3(){  
 employees.stream()  
 .filter((e) -> {  
 System.*out*.println("短路");//类似于&&和||，可以提高效率  
 return e.getSalary() > 5000;  
 })  
 .limit(2)  
 .forEach(System.*out*::println);  
}

Stream<T> skip(long n)：跳过元素，返回一个扔掉了前n个元素的流。若流中元素不足n个，则返回一个空流。与limit(n)互补

@Test  
public void test4(){  
 employees.stream()  
 .filter((e) -> e.getSalary() > 5000)  
 .skip(2)  
 .forEach(System.*out*::println);  
}

Stream<T> distinct()：筛选，通过流所生成元素的 hashCode()和equals()去除重复元素。

所以，使用distinct筛选的前提是元素重写了hashCode()和equals()

@Test  
public void test5(){  
 employees.stream()  
 .filter((e) -> e.getSalary() > 5000)  
 .skip(2)  
 .distinct()  
 .forEach(System.*out*::println);  
}

(2)映射

<R> Stream<R> map(Function<? super T,? extends R> mapper)：接收Lambda，将元素转换成其

他形式或提取信息。接收一个函数作为参数，该函数会被应用到每个元素上，并将其映

射成一个新的元素。

List<String> list = Arrays.*asList*("aaa","bbb","ccc","ddd","eee");  
list.stream()  
 .map((str) -> str.toUpperCase())  
 .forEach(System.*out*::println);  
System.*out*.println("--------------------------");  
  
employees.stream()  
 .map(Employee::getName)  
 .forEach(System.*out*::println);  
System.*out*.println("--------------------------");

<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)：接收一个

函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

Stream<Stream<Character>> stream = list.stream()  
 .map(TestStreamAPI2::*filterCharacter*);  
 stream.forEach((sm) -> {  
 sm.forEach(System.*out*::println);  
 });  
 System.*out*.println("--------------------------");  
  
 Stream<Character> sm = list.stream()  
 .flatMap(TestStreamAPI2::*filterCharacter*);  
 sm.forEach(System.*out*::println);  
}  
public static Stream<Character> filterCharacter(String str){  
 List<Character> list = new ArrayList<>();  
 for (Character ch : str.toCharArray()){  
 list.add(ch);  
 }  
 return list.stream();  
}

map 和flatMap的关系类似于list集合的add和addAll方法之间的关系。

(3)排序

Stream<T> sorted()：自然排序(Comparable)，即按照字典排序

List<String> list = Arrays.*asList*("ccc","aaa","bbb","eee","ddd");  
list.stream()  
 .sorted()  
 .forEach(System.*out*::println);

Stream<T> sorted(comparator<? super T> comparator)：定制排序(Comparator)，下面的两个

compareTo方法都是升序排序。

employees.stream()  
 .sorted((e1,e2) -> {  
 if(e1.getAge().equals(e2.getAge())){  
 return e1.getName().compareTo(e2.getName());  
 }else{  
 return e1.getAge().compareTo(e2.getAge());  
 }  
 }).forEach(System.*out*::println);

(4)查找与匹配

boolean allMatch(Predicate<? super T> predicate)：检查是否匹配所有元素

boolean b1 = employees.stream()  
 .allMatch((e) -> e.getStatus().equals(Employee.Status.*BUSY*));  
System.*out*.println(b1);//false

boolean anyMatch(Predicate<? super T> predicate)：检查是否至少匹配一个元素

boolean b2 = employees.stream()  
 .anyMatch((e) -> e.getStatus().equals(Employee.Status.*BUSY*));  
System.*out*.println(b2);//true

boolean noneMatch(Predicate<? super T> predicate)：检查是否不匹配所有元素

boolean b3 = employees.stream()  
 .noneMatch((e) -> e.getStatus().equals(Employee.Status.*BUSY*));  
System.*out*.println(b3);//false

Optional<T> findFirst()：返回第一个元素

//Optional是jdk8引用的用于解决空指针的容器类。findFirst()可能返回null  
Optional<Employee> op1 = employees.stream()  
 .sorted((e1, e2) -> -Double.*compare*(e1.getSalary(),e2.getSalary()))  
 .findFirst();  
//op.get()返回该类存储的对象，没有对象会报NoSuchElementException  
System.*out*.println(op1.get());

Optional<T> findAny()：返回当前选中的任意元素

//findAny()在顺序流中一般会返回第一个结果；在并行流中随机返回一个合适的结果  
Optional<Employee> op2 = employees.stream()//串行流  
 .filter((e) -> e.getStatus().equals(Employee.Status.*FREE*))  
 .findAny();  
Optional<Employee> op3 = employees.parallelStream()//并行流  
 .filter((e) -> e.getStatus().equals(Employee.Status.*FREE*))  
 .findAny();  
System.*out*.println("串行流："+op2.get()+"\n"+"并行流："+op3.get());

long count()：返回流中元素的总个数

Long count = employees.stream()  
 .count();  
System.*out*.println(count);

Optional<T> max(comparator<? super T> comparator)：返回流中最大值

Optional<Employee> op1 = employees.stream()  
 .max((e1, e2) -> Double.*compare*(e1.getSalary(),e2.getSalary()));  
System.*out*.println(op1.get());

Optional<T> min(Comparator<? super T> comparator)：返回流中最小值

Optional<Double> op2 = employees.stream()  
 .map(Employee::getSalary)  
 .min(Double::*compare*);  
System.*out*.println(op2.get());

(5)规约

T reduce(T identity, BinaryOperator<T> accumulator)：identity是起始值，accumulator是带

有运算方法的对象，可以将流中元素反复结合起来，得到一个值。

List<Integer> list = Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
//元素累加  
Integer sum = list.stream()  
 .reduce(0,(x,y) -> x+y);  
System.*out*.println(sum);//结果为55

补充：上面代码的逻辑为，首先将起始值0作为x，集合的第一个元素1作为y，二者相加

得到1；然后将这个1当作x，将集合的第二个元素2当作y，二者继续相加，以此类

推将集合中的元素加完得到一个值55。

[Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<[T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/stream/Stream.html)> reduce([BinaryOperator](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/function/BinaryOperator.html)<[T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/stream/Stream.html)> accumulator)：将流中元素反复结合起来，得到一个值，

但是这个方法跟上面的方法不同的是它没有指定的起始值，所以它的返回值为Optional，

防止其最后运行的结果为null。

//计算工资共和  
Optional<Double> op = employees.stream()  
 .map(Employee::getSalary)  
 .reduce(Double::*sum*);  
System.*out*.println(op.get());

补充：上面代码逻辑为，用map方法获取到每一个Employee对象的salary，然后用reduce

方法将所有的salary相加到一起得到工资总和。

(6)收集

<R,A> R collect(Collector<? super T,A,R> collector)：将流转换为其他形式。接收一个Collector

接口的实现，用于给Stream中元素做汇总的方法。这个Collector接口的实现，一般是

具体类Collectors的某个方法返回的。比如下面代码中的toList()、toSet()、counting()等。

//将list中存放的对象中的name放到另一个list中  
List<String> list = employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toList*());  
list.forEach(System.*out*::println);

//将list中存放的对象中的name放到一个set中(存放的name无序且唯一)  
Set<String> set = employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toSet*());  
set.forEach(System.*out*::println);

//可以将值存储在任意集合中  
HashSet<String> hs = employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toCollection*(HashSet::new));  
hs.forEach(System.*out*::println);

//得到总数  
Long count = employees.stream()  
 .collect(Collectors.*counting*());  
System.*out*.println(count);

//得到平均值  
Double avg = employees.stream()  
 .collect(Collectors.*averagingDouble*(Employee::getSalary));  
System.*out*.println(avg);

//得到工资总和  
Double sum = employees.stream()  
 .collect(Collectors.*summingDouble*(Employee::getSalary));  
System.*out*.println(sum);

//最大工资的对象  
Optional<Employee> max = employees.stream()  
 .max(Comparator.*comparingDouble*(Employee::getSalary));  
System.*out*.println(max.get());

//最小工资  
Optional<Double> min = employees.stream()  
 .map(Employee::getSalary)  
 .collect(Collectors.*minBy*(Double::*compare*));  
System.*out*.println(min.get());

//按照员工对象中的状态分组，状态Status是一个枚举对象  
Map<Employee.Status,List<Employee>> map = employees.stream()  
 .collect(Collectors.*groupingBy*(Employee::getStatus));  
System.*out*.println(map);

//多级分组  
Map<Employee.Status,Map<String,List<Employee>>> map = employees.stream()  
 .collect(Collectors.*groupingBy*(Employee::getStatus,Collectors.*groupingBy*((e) -> {  
 if(e.getAge() <= 35){  
 return "青年";  
 }else if(e.getAge() <= 50){  
 return "中年";  
 }else{  
 return "老年";  
 }  
 })));  
System.*out*.println(map);

//分区  
Map<Boolean,List<Employee>> map = employees.stream()  
 .collect(Collectors.*partitioningBy*((e) -> e.getSalary() > 8000));  
System.*out*.println(map);

//另外的方式获取总数、平均值、最大值等  
DoubleSummaryStatistics dss = employees.stream()  
 .collect(Collectors.*summarizingDouble*(Employee::getSalary));  
System.*out*.println(dss.getSum());  
System.*out*.println(dss.getAverage());  
System.*out*.println(dss.getMax());

//Collectors.joining()：将元素值拼接起来，其有几个重载的方法。  
String str = employees.stream()  
 //假设只有三个对象，其name分别为张三、李四、王五  
 .map(Employee::getName)  
 //第一个重载的方法，结果为：张三李四王五  
// .collect(Collectors.joining());  
 //第二个重载的方法，结果为：张三,李四,王五  
// .collect(Collectors.joining(","));  
 //第三个重载的方法，结果为：==张三,李四,王五===  
.collect(Collectors.*joining*(",","==","==="));  
System.*out*.println(str);

## 3.Optional类

1.说明

Optional<T>类(java.util.Optional)是一个容器类，代表一个值存在或不存在，原来用null表示

一个值不存在，现在Optional可以更好的表达这个概念。并且可以避免空指针异常。

2.方法

public static <T> [Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<T> of(T value)：创建一个Optional实例，若value为null报

[NullPointerException](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/NullPointerException.html)

public static <T> [Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<T> empty()：创建一个空的Optional实例

public static <T> [Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<T> ofNullable(T value)：若value不为null，创建一个Optional

实例，否则创建一个空实例。

public boolean isPresent()：判断是否包含值

public [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html) get()：返回实例的值，若没值，报[NoSuchElementException](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/NoSuchElementException.html)

public [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html) orElse([T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html) other)：如果调用对象包含值，返回该值，否则返回other

public [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html) orElseGet([Supplier](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/function/Supplier.html)<? extends [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)> other)：如果调用对象包含值，返回该值，否则

返回other获取的值。

public <U> [Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<U> map([Function](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/function/Function.html)<? super [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html),? extends U> mapper)：如果有值对其处理

，并返回处理后的Optional，否则返回Optional.empty()

public <U> [Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<U> flatMap([Function](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/function/Function.html)<? super [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html),[Optional](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Optional.html)<U>> mapper)：与map方法类

似，要求返回值必须是Optional。

## 4.接口中的默认方法与静态方法

1.接口中的默认方法(jdk1.8出现)

说明：在JDK8之后，接口中可以写默认的具体方法。方法由default修饰。如：

public interface MyFun{

default String getName(){

return “哈哈哈”;

}

}

由此出现两个问题：

(1)假设接口A有一个默认方法getName()，具体类B也有一个方法getName()，而具体类C

继承了类B，同时实现了接口A。那么当我们创建了C的对象并调用getName()时，会

执行接口A中的方法，还是具体类B中的方法呢？答案是具体类B中的getName()，这

就是”类优先”原则。

(2)假设接口A有一个默认方法getName()，接口B也有一个默认方法getName()，具体类C

实现了接口A和B，那么C就必须重写getName()来解决冲突。在类C的方法中，如果

想调用接口A的getName()，可以这么写：A.super.getName()，如果想调用接口B的getName()，可以这么写：B.super.getName()。

2.接口中的静态方法(jdk1.8出现)

说明：接口中的静态方法同样是由static修饰，调用方式为：接口名.方法名(参数列表)。

## 5.新时间日期API

1.说明：由于之前的日期API存在线程安全问题以及诸多不方便之处，JDK8提供了全新的日

期API。

2.LocalDateTime(日期时间)类

方法：

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) now()：从默认时区中的系统时钟中获取当前日期时间对象。

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) now([ZoneId](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZoneId.html) zone)：按设置的时区获取当前日期时间对象。

public [ZonedDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZonedDateTime.html) atZone([ZoneId](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZoneId.html) zone)：结合时间和时区创建[ZonedDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZonedDateTime.html) 对象。

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-int-int-int-int-)(int year, int month, int dayOfMonth, int hour, int minute)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-int-int-int-int-int-)(int year, int month, int dayOfMonth, int hour, int minute,

int second)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-int-int-int-int-int-int-)(int year, int month, int dayOfMonth, int hour, int minute,

int second, int nanoOfSecond)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-java.time.Month-int-int-int-)(int year, [Month](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Month.html) month, int dayOfMonth, int hour, int minute)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-java.time.Month-int-int-int-int-)(int year, [Month](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Month.html) month, int dayOfMonth, int hour, int minute,

int second)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-int-java.time.Month-int-int-int-int-int-)(int year, [Month](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Month.html) month, int dayOfMonth, int hour, int minute,

int second, int nanoOfSecond)

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) [of](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html#of-java.time.LocalDate-java.time.LocalTime-)([LocalDate](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDate.html) date, [LocalTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalTime.html) time)

以上7个of方法都是设置时间来获取[LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html)对象的。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusYears(long years)：增加years年

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusWeeks(long weeks)：增加weeks周

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusSeconds(long seconds)：增加seconds秒

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusNanos(long nanos)：增加nanos纳秒

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusMonths(long months)：增加months月

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusMinutes(long minutes)：增加minutes分钟

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusHours(long hours)：增加hours小时

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) plusDays(long days)：增加days天

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusDays(long days)：减去days天

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusHours(long hours)：减去hours小时

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusMinutes(long minutes)：减去minutes分钟

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusMonths(long months)：减去months月

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusNanos(long nanos)：减去nanos纳秒

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusSeconds(long seconds)：减去seconds秒

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusWeeks(long weeks)：减去weeks周

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) minusYears(long years)：减去years年

public int getDayOfMonth()：获取月中的一天

public [DayOfWeek](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/DayOfWeek.html) getDayOfWeek()：获取一周中的一天，[DayOfWeek](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/DayOfWeek.html) 是一个枚举

public int getDayOfYear()：获取年中的一天

public int getHour()：获取小时

public int getMinute()：获取分钟

public int getMonthValue()：获取月份

public int getNano()：获取纳秒

public int getSecond()：获取分钟

public int getYear()：获取年

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withYear(int year)：设置年份为year。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withMonth(int month)：设置一年中的月份为month。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withDayOfMonth(int dayOfMonth)：将月中的天数设置为

dayOfMonth。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withDayOfYear(int dayOfYear)：将年中的天数设置为dayOfYear。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withHour(int hour)：设置一天中的小时数为hour。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withMinute(int minute)：设置小时中的分钟数为minute。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) withSecond(int second)：设置分钟中的秒数为second。

public [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) with([TemporalAdjuster](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAdjuster.html) adjuster)：返回时间校正后的对象。

public [String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) format([DateTimeFormatter](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/format/DateTimeFormatter.html) formatter)：返回格式化后的时间字符串。

public static [LocalDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDateTime.html) parse([CharSequence](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/CharSequence.html) text,[DateTimeFormatter](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/format/DateTimeFormatter.html) formatter)：解析

时间。

3.LocalDate和LocalTime类

这俩类和LocalDateTime类中的方法几乎一样，不同的是LocalDate中没有操作时、分、秒、

毫秒、纳秒的方法；LocalTime没有操作年、月、日、周的方法。

4.Instant(时间戳)类

时间戳：以Unix元年：1970年1月1日00:00:00到某个时间之间的毫秒值。

方法：

public static [Instant](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Instant.html) now()：获取当前时间对应的[Instant](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Instant.html) 对象。默认获取UTC时区，而不

是(UTC-8)东八区。

public [OffsetDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/OffsetDateTime.html) atOffset([ZoneOffset](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZoneOffset.html) offset)：设置时间偏移量。例：

[OffsetDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/OffsetDateTime.html) odt = Instant.now().atOffset(ZoneOffset.ofHours(8));//设置增加8个小

时，即东八区的时间。[OffsetDateTime](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/OffsetDateTime.html) 为时间偏移类。

public long toEpochMilli()：返回1970-01-01 00:00:00距现在的毫秒数。

public long getEpochSecond()：返回1970-01-01 00:00:00距现在的秒数。

public static [Instant](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Instant.html) ofEpochSecond(long epochSecond)：返回自1970-01-01 00:00:00增加

epochSecond秒的[Instant](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Instant.html) 对象。

5.Duration(时间间隔)类

方法：

public static [Duration](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Duration.html) between([Temporal](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/Temporal.html) startInclusive,[Temporal](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/Temporal.html) endExclusive)：返回两个时

间对象之间的间隔时间。

public long toMillis()：返回间隔时间的毫秒值。例：

Instant t1 = Instant.now();

Instant t2 = Instant.now();

Duration d = Duration.between(t1,t2);

System.out.println(d.toMillis());//输出间隔时间的毫秒值

public long getSeconds()：返回间隔时间的秒值。

public long toDays()：返回间隔时间的天数。

public long toHours()：返回间隔时间的小时数。

public long toMinutes()：返回间隔时间的分钟数。

代码演示：

LocalDateTime t1 = LocalDateTime.*of*(2019,12,12,13,14);  
LocalDateTime t2 = LocalDateTime.*of*(2021,1,10,16,25);  
//使用between方法时，第一个参数的时间必须比第二个参数小，否则输出的时间间隔是负数  
Duration d = Duration.*between*(t1,t2);  
System.*out*.println(d.toDays());//395  
System.*out*.println(d.toHours());//9486  
System.*out*.println(d.toMinutes());//568991  
System.*out*.println(d.getSeconds());//34139460  
//对比  
Duration d2 = Duration.*between*(t2,t1);  
System.*out*.println(d2.toDays());//-395  
System.*out*.println(d2.toHours());//-9486  
System.*out*.println(d2.toMinutes());//-568991  
System.*out*.println(d2.getSeconds());//-34139460

6.Period(日期间隔)类

方法：

public static [Period](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/Period.html) between([LocalDate](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDate.html) startDateInclusive,[LocalDate](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDate.html) endDateExclusive)：返

回两个日期对象之间的间隔时间。

public int getDays()：返回间隔时间的天数。

public int getMonths()：返回间隔时间的月数。

public int getYears()：返回间隔时间的年数。

代码演示：

LocalDate ld1 = LocalDate.*of*(2023,6,6);  
LocalDate ld2 = LocalDate.*of*(2011, 1, 1);  
//使用between方法时，第一个参数时间必须比第二个参数小，否则输出的时间间隔是负数  
Period pe = Period.*between*(ld2, ld1);  
System.*out*.println(pe.getYears());//12  
System.*out*.println(pe.getMonths());//5  
System.*out*.println(pe.getDays());//5  
//对比  
Period pe2 = Period.*between*(ld1, ld2);  
System.*out*.println(pe2.getYears());//-12  
System.*out*.println(pe2.getMonths());//-5  
System.*out*.println(pe2.getDays());//-5

7.TemporalAdjuster(时间校正器)接口和TemporalAdjusters(时间校正器)工具类

说明：TemporalAdjusters通过静态方法提供了大量的常用TemporalAdjuster的实现。

方法：

public static [TemporalAdjuster](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAdjuster.html) firstDayOfMonth()：设置时间为当月第一天。

public static [TemporalAdjuster](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAdjuster.html) firstDayOfNextMonth()：设置时间为下月第一天。

public static [TemporalAdjuster](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAdjuster.html) lastDayOfMonth()：设置时间为当月最后一天。

类似的方法还有几个，可以去API文档查看。这些方法可以和LocalDateTime、[LocalDate](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/LocalDate.html)、

LocalTime的with([TemporalAdjuster](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAdjuster.html) adjuster)配合使用。

public static TemporalAdjuster next(DayOfWeek dayOfWeek)：dayOfWeek参数是一个枚举

对象，可以指定周一至周天。所以该方法表示下一次的周一至周天的时间。

方法示例：

//下一个周日  
LocalDateTime ldt3 = LocalDateTime.*now*().with(TemporalAdjusters.*next*(DayOfWeek.*SUNDAY*));  
System.*out*.println(ldt3);//2023-06-11T17:18:55.476

//自定义：获取下一个工作日的时间(周一至周五)  
LocalDateTime ldt5 = ldt.with((l) -> {  
 LocalDateTime ldt4 = (LocalDateTime) l;  
 //获取周几  
 DayOfWeek dow = ldt4.getDayOfWeek();  
 //如果今天是周五，那么当前天数加3天就是下一个工作日  
 if(dow.equals(DayOfWeek.*FRIDAY*)){  
 return ldt4.plusDays(3);  
 }else if(dow.equals(DayOfWeek.*SATURDAY*)){  
 //如果今天是周六，那么当前天数加2天就是下一个工作日  
 return ldt4.plusDays(2);  
 }else{  
 //除以上两种情况，当前天数加1天就是下一个工作日  
 return ldt4.plusDays(1);  
 }  
});  
System.*out*.println(ldt5);

8.DateTimeFormatter(格式化时间)类

说明：该类用作格式化时间，它有一些指定的格式化类型，由静态常量指定。

静态常量：



方法：

public [String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) format([TemporalAccessor](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/temporal/TemporalAccessor.html) temporal)：格式化时间。

public static [DateTimeFormatter](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/format/DateTimeFormatter.html) ofPattern([String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) pattern)：创建自定义的格式化类型。

代码示例：

DateTimeFormatter m = DateTimeFormatter.*ofPattern*("yyyy年MM月dd日 HH:mm:ss");  
LocalDateTime t = LocalDateTime.*now*();  
String strDate = m.format(t);//格式化时间  
System.*out*.println(strDate);//2023年06月06日 20:00:58  
LocalDateTime t2 = LocalDateTime.*parse*(strDate,m);//解析时间  
System.*out*.println(t2);//2023-06-06T20:00:58

9.ZoneId(时区)类

方法：

public static [Set](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/util/Set.html)<[String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html)> getAvailableZoneIds()：返回所有可用的时区集合。

public static [ZoneId](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/time/ZoneId.html) of([String](mk:@MSITStore:F:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) zoneId)：设置时区，zoneId可以从getAvailableZoneIds方法

返回的所有时区字符串中选择一个。

10.ZonedDateTime类

说明：LocalDateTime无法与时间戳进行转换，因为LocalDateTime没有时区，无法确定某一

时刻，而ZonedDateTime可以表示一个带时区的日期和时间。

常用方法：

static ZonedDateTime now()：从默认时区中的系统时钟中获取当前日期时间

static ZonedDateTime now(Clock clock)：从指定的时钟中获取当前日期时间

static ZonedDateTime now(ZoneId zone)：从指定时区中的系统时钟中获得当前日期时间

static ZonedDateTime of(int year,int month,int dayOfMonth,int hour,int minute,int

second,int nanoOfSencond,ZoneId zone)：从年，月，日，小时，分钟，秒，纳秒和

时区综合选择下获得 ZonedDateTime实例

static ZonedDateTime of(LocalDate date,LocalTime time,ZoneId zone)：从本地日期和时间

获得ZonedDateTime实例

static ZonedDateTime of(LocalDateTime localDateTime,ZoneId zone)：从本地日期时间获得

ZonedDateTime实例

static ZonedDateTime parse(CharSequence text)：从一个文本字符串获得 ZonedDateTime

实例，如 2007-12-03T10:15:30+01:00[Europe/Paris]

static ZonedDateTime parse(CharSequence text,DateTimeFormatter formatter)：从使用特定

格式的文本字符串获得ZonedDateTime实例

ZonedDateTime withZoneSameInstant(ZoneId zone)：以不同的时区返回此日期时间的副

本，保留即时

补充：重点说下时区的转换，比如当前的上海时间对比的纽约时间是多少。代码示例：

LocalDateTime localDateTime = LocalDateTime.*of*(2023,6,6,20,46,20);  
ZonedDateTime zonedDateTime1 = ZonedDateTime.*of*(localDateTime,ZoneId.*of*("Asia/Shanghai"));  
ZonedDateTime zonedDateTime2 = zonedDateTime1.withZoneSameInstant(ZoneId.*of*("America/New\_York"));  
System.*out*.println(zonedDateTime1);//2023-06-06T20:46:20+08:00[Asia/Shanghai]  
System.*out*.println(zonedDateTime2);//2023-06-06T08:46:20-04:00[America/New\_York]

可以看到纽约时间比上海时间晚12个小时。

# 多线程

## 1.什么是进程？

在任务管理器中我们可以看到进程，只有运行的程序才会有进程。所以进程是正在运行的程

序。进程是系统进行资源分配和调用的独立单位，每一个进程都有它自己的内存空间和

系统资源。

## 2.多进程的意义

单进程的计算机只能做一件事情，而现代的多进程计算机可以同时做多件事情，即在一个时

间段内执行多个任务，这可以提高CPU的使用率。

## 3.什么是线程？

在一个进程中可以执行多个任务，这每一个任务都可以看作是一个线程。线程是程序的执行

单元，是程序使用CPU的最基本单位。比如：6核12线程的CPU最多可以模拟12个

核心，也就是最多可以同时执行12个线程。

## 4.多线程的意义

多线程不是为了提高程序的执行速度，而是为了提高应用程序的使用率。线程的执行具有随

机性，能抢到CPU就执行，抢不到就不执行。

## 5.线程的生命周期

新建：创建线程对象

就绪：有执行资格，没有执行权

运行：有执行资格，有执行权

阻塞：由于一些操作让线程处于该状态，没有执行资格，没有执行权；而另一些操作却可以把它给激活，激活后处于就绪状态。

死亡：线程对象变成垃圾，等待被回收



## 6.线程的状态转换图



## 7.并行和并发

并行是指在某一个时间点同时运行多个程序。

并发是指在某一个时间内同时运行多个程序。

## 8.Java程序运行原理

java命令会启动java虚拟机，启动JVM，等于启动了一个应用程序，也就是启动了一个进程。

该进程会自动启动一个“主线程”，然后主线程去调用某个类的main方法。所以main方法运行在主线程中。在此之前的所有程序都是单线程的。

jvm虚拟机的启动是多线程的，因为垃圾回收线程也要先启动。

Java本身无法直接调用系统功能去实现多线程，只能调用C/C++写好的模块来实现多线程，

由C/C++来负责调用系统功能创建进程。

## 9.线程的两种调度模型

分时调度模型：所有线程轮流使用CPU的使用权，平均分配每个线程占用CPU的时间片。

抢占式调度模型：优先让优先级高的线程使用CPU，如果线程的优先级相同,那么会随机选

择一个,优先级高的线程获取的CPU时间片相对多一些。Java使用的便是这种模型。

## 10.创建线程的几种方式

(1)继承Thread类

A：自定义MyThread类去继承Thread类。

B：MyThread类重写run()方法，run()方法中写入需要多线程执行的代码。

C：创建MyThread类的对象，使用start()方法。

start()方法会启动线程，然后由jvm去调用该线程的run()方法。

D：如果想多个线程同时执行，需要创建多个MyThread类的对象来调用start()方法。

(2)实现Runnable接口

A：自定义MyRunnable类去实现Runnable接口。

B：MyRunnable类重写run()方法，run()方法中写入需要多线程执行的代码。

C：创建MyRunnable类的对象。

D：创建Thread类的对象，并把C步骤的对象作为构造参数传递。

Thread(Runnable target)

Thread(Runnable target,String name)

E：使用Thread类的对象的start()方法。

(3)实现[Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)<V>接口，直接上示例

A：做一个类MyThread实现[Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)<V>接口并实现call()，call方法可以有返回值。

B：使用FutureTask类的有参构造搭配MyThread类的对象创建一个FutureTask对象。

C：使用Thread类的有参构造搭配FutureTask对象创建Thread对象，使用Thread对象执行

start()即可执行线程。

D：执行完线程后，FutureTask对象调用get()可以得到call()的返回值。

补充：get()可能会出现阻塞，因为要等待call()执行完毕，最好使用异步通信。

补充：C步中start()执行的是call()中的代码，并将call()的返回值缓存起来。

(4)依赖线程池(了解)

A：JDK5出现的Executors工厂类有方法，创建一个线程池对象，控制要容纳几个线程对象。方法：

public static [ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html) newFixedThreadPool(int nThreads)

B：做一个类实现[Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)<V>接口并实现call方法。

C：A中返回的[ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html)可以调用方法：<T> [Future](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Future.html)<T> submit([Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)<T> task)来执行

线程。

D：由于重写的call方法带返回值，有种方式可以获取到这个返回值。C中的submit方法会

返回一个[Future](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Future.html)对象，调用该对象的[V](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Future.html) get()即可得到返回值。

E：当使用线程后，A中返回的[ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html)可以调用void shutdown()关闭线程池。

(4)匿名内部类实现多线程(了解)

代码示例：



还有一种更复杂的：



这种方式执行起来实际只走world那行所在的线程。

(5)有了方式一，为什么还要方式二呢？

A：可以避免由于Java单继承带来的局限性。

B：适合多个相同程序的代码去处理同一个资源的情况，把线程同程序的代码，数

据有效分离，较好的体现了面向对象的设计思想。

## 11.详解继承Thread类的多线程方式

Thread类中的方法：

(1)构造方法

public Thread()：分配新的 Thread 对象

public Thread([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)：分配新的 Thread 对象，name是指定的线程名称。

(2)成员方法

public final [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) getName()：返回该线程的名称。

public final void setName([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)：设置该线程的名称。

public static [Thread](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html) currentThread()：获取当前正在执行的线程对象。该方法可以让不是

Thread类的子类来获取它所处的线程对象，比如在main方法中使用它。

public final int getPriority()：返回该线程的优先级。默认是5。

public final void setPriority(int newPriority)：更改线程的优先级。范围：[1,10]之间的整数，

越大表示获取CPU时间片的几率越高。

public static void sleep(long millis)：让当前正在执行的线程休眠millis毫秒。特点：sleep

系列方法不会释放锁。

public static void sleep(long millis, int nanos)：让当前正在执行的线程休眠millis毫秒

+nanos纳秒。nanos的范围是0-999999。

public final void join()：其他线程一直等待该线程终止。

public final void join(long millis)：其他线程等待该线程执行millis毫秒。

public final void join(long millis, int nanos)：其他线程等待该线程执行millis毫秒+ nanos

纳秒。nanos的范围是0-999999。

public static void yield()：暂停当前正在执行的线程对象，并执行其他线程。原理是让正

在执行的线程变为就绪状态，然后CPU根据调度算法重新从就绪状态的线程中选

择一个执行，因此礼让不一定成功。跟sleep方法一样，它也不会释放锁。

public final void setDaemon(boolean on)：on为true时表示将该线程标记为守护线程，on

为false时标记为用户线程，线程默认都是用户线程。当正在运行的线程都是守护

线程时，Java 虚拟机退出。该方法必须在启动线程前调用。

public final void stop()：中断线程。该方法已过时。该方法不会让正在休眠的线程抛出异

常，因此不建议使用。建议使用其替代方法interrupt()。

public void interrupt()：中断线程。如果线程在调用 [Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) 类的 [wait()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#wait())、[wait(long)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#wait(long)) 或

[wait(long, int)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#wait(long, int)) 方法，或者该类的 [join()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html#join())、[join(long)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html#join(long))、[join(long, int)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html#join(long, int))、[sleep(long)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html#sleep(long)) 或 [sleep(long, int)](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Thread.html#sleep(long, int)) 方法过程中受阻，则其中断状态将被清除，它还将收到一个 [InterruptedException](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/InterruptedException.html)。

public final boolean isAlive()：测试线程是否处于活动状态。如果线程已经启动且尚未终

止，则为活动状态。

public [Thread.State](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Thread.State.html) getState()：获取当前线程的状态，返回值是一个枚举对象。NEW(新

建)、RUNNABLE(运行)、BLOCKED(其他阻塞)、WAITING(不限时间的等待阻塞)、

TIMED\_WAITING(限定时间的等待阻塞)、TERMINATED(终止)。

## 12.详解实现Runnable接口的多线程方式

Thread类中的方法：

(1)构造方法

public Thread([Runnable](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Runnable.html) target)：分配新的Thread对象。target为其run方法被调用的对

象。

public Thread([Runnable](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Runnable.html) target, [String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name)：分配新的Thread对象。target为其run方法

被调用的对象。name为新线程的名称。

## 13.线程安全问题

出现线程安全问题的条件：

A：处于多线程环境

B：有共享数据

C：有多条语句操作共享数据

解决方式：

将多条语句操作共享数据的代码给包成一个整体，让某个线程在执行的时候，其他线程

不能执行。

解决方案：

(1)同步代码块

格式：

synchronized(对象){需要同步的代码}

这里的对象起着锁的功能，多个线程必须用同一把锁(同一个对象)，才能实现某个线程

在执行同步的代码时，其他线程不会抢过执行权去执行同步代码。这里的对象可以

是任意对象。必须用同一把锁的原因是：只要锁被其中一个线程使用，那么即使有其他的线程抢到了执行权，也会因为锁被其他线程占用而无法运行，导致进入等待。

synchronized是非公平锁。

好处：解决了多线程的安全问题。

缺点：当线程相当多时，因为每个线程都得判断同步上的锁，这是很耗费资源的，无形中降

低了程序的运行效率。

(2)同步方法的格式及其锁对象

说明：如果一个方法内的所有代码都在synchronized的代码块内，则这个方法可以被定义为

同步方法。

格式：

让synchronized修饰方法即可。例如：

private synchronized void sellTicket(){…}

那么同步方法的锁对象是哪个对象呢？

this。

那么静态同步方法的锁对象是哪个对象呢？

因为静态方法的执行优先于对象，因此锁对象不可能是this，而是当前类的class对象。例如：SellTicket.class。

补充：获取线程安全的ArrayList。

List<String> list = Collections.synchronizedList(new ArrayList<String>());

或者

CopyOnWriteArrayList list = new CopyOnWriteArrayList();

## 14.jdk5中的Lock锁

说明：虽然我们可以理解同步代码块和同步方法的锁对象问题，但是我们并没有直接看到在

哪里加上了锁，在哪里释放了锁，为了更清晰的表达如何加锁和释放锁，JDK5以后提

供了一个新的锁对象Lock。

(1)Lock – 接口

抽象方法：

void lock()：获取锁(加锁)。如果锁不可用，出于线程调度目的，将禁用当前线程，并且

在获得锁之前，该线程将一直处于休眠状态。

void unlock()：释放锁。

(2)ReentrantLock(可重入锁) – 类。实现了Lock接口。

构造方法：

public ReentrantLock()：创建一个实例。等同于ReentrantLock(false)。

public ReentrantLock(boolean fair)：创建一个实例。如果fair为true，意味着此锁使用公

平的排序策略，反之是不公平的排序策略。

方法：

void lock()：获取锁(加锁)。如果锁不可用，出于线程调度目的，将禁用当前线程，并且

在获得锁之前，该线程将一直处于休眠状态。

void unlock()：释放锁。

public boolean tryLock()：尝试获取锁，成功则返回true；失败则返回false，这个方法无

论如何都会立即返回。不会像synchronized一样，一个线程获取锁之后，其他线程

只能等待那个线程释放锁之后才能有获取锁的机会。

public boolean tryLock(long timeout, TimeUnit unit)：规定时间内尝试获取锁，unit是枚举

类型，表示使用哪种时间，比如秒、分钟...。

加锁和释放锁的代码示例：



(3)Condition接口

介绍：

Lock取代了synchronized方法和语句的使用，而Condition则取代了对象监视器的使用

方法，即Condition中有取代wait、notify、notifyAll的方法。

方法：

void await()：当前线程等待。

void signal()：唤醒一个等待的线程。

void signalAll()：唤醒所有等待线程。

使用方式(示例，只写了一个生产者线程)：

public class TestCondition{  
 Lock lock = new ReentrantLock();  
 Condition condition = lock.newCondition();//获取Condition对象。  
 int number;  
 public void increment(){  
 lock.lock();  
 try{  
 //业务代码  
 while(number != 0){  
 condition.await();  
 }  
 number++;  
 condition.signalAll();  
 }catch(Exception e){  
 e.printStackTrace();  
 }finally{  
 lock.unlock();  
 }  
 }  
}

(4)ReentrantLock和synchronized的区别



## 15.死锁问题

说明：是指两个或两个以上的线程在执行的过程中，因争夺资源产生的一种互相等待现象。

例如：当线程1走if中的语句使用锁objA，然后线程2抢到执行权，走else中的语句也想

用锁objA，就会陷入等待(因为objA正在被线程1使用)，之后线程1又抢到了执行权，

想用锁对象objB，但是objB正在被线程2使用，所以线程1也陷入等待。就这样两个

线程都想用对方的资源，形成一种互相等待的现象，形成死锁。



产生死锁的四个必要条件：

(1)互斥条件：一个资源每次只能被一个进程使用；

(2)请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放；

(3)不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺；

(4)循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系；

## 16.线程中的生产者和消费者问题

说明：一般来说，生产者是设置数据的那些线程，消费者是使用数据的那些线程。二者共用

一些数据资源。例如：线程1给Student对象中的name变量赋值，线程2打印同一个

Student对象中的name变量值，如此，线程1是生产者，线程2是消费者。为了解决

线程安全问题，给这些线程加上同一把锁。但即使加了锁，仍然会有非理想中的执行顺

序。

问题所在：

A：如果消费者先抢到CPU的执行权，就会去消费数据，但是现在的数据是默认值。没

有意义,应该等着数据有意义，再消费。

B：如果生产者先抢到CPU的执行权，就会去产生数据，但是呢，它产生完数据后，还

继续拥有执行权,它又继续产生数据。这是有问题的，你应该等着消费者把数据消费掉，然后再生产。

正常思路：

A：生产者 - 先看是否有数据，有就等待，没有就生产，生产完之后通知消费者消费

B：消费者 - 先看是否有数据，有就消费，没有就等待，通知生产者生产数据

解决问题的方法：Java的等待唤醒机制

首先介绍Object类中相关方法：

public final void wait()：在其他线程调用此对象的 [notify()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notify()) 方法或 [notifyAll()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notifyAll()) 方法

前，当前线程一直等待。特点：调用此方法使线程等待时，会释放锁。

public final void wait(long timeout)：如果timeout为0，则此方法和wait()方法

效果是一样的。如果timeout不为0，则表示等待timeout毫秒后，若仍没有

被[notify()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notify())或[notifyAll()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notifyAll())唤醒，则自己会自动唤醒，进入就绪状态。

public final void wait(long timeout,int nanos)：如果timeout和nanos都为0，则此方

法和wait()方法效果是一样的。如果有不为0，则表示等待timeout毫秒+

nanos纳秒后，若仍没有被[notify()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notify())或[notifyAll()](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html#notifyAll())唤醒，则自己会自动唤醒，进

入就绪状态。

public final void notify()：唤醒在此对象监视器上等待的单个线程。但是唤醒并不表

示可以立即执行，还得继续抢CPU的执行权。

public final void notifyAll()：唤醒在此对象监视器上等待的所有线程。

在解决这个问题的例子中，上面五个方法由锁对象调用。

问题：为什么wait、notify、notifyAll三个方法定义在Object类中？

因为这些方法的调用依赖于锁对象，而锁对象可以是任意对象。

补充：

(1)虚假唤醒。

虚假唤醒是一种现象，它只会出现在多线程环境中，指的是在多线程环境下，多个

线程等待在同一个条件上，等到条件满足时，所有等待的线程都被唤醒，但由于多个线程执行的顺序不同，后面竞争到锁的线程在获得时间片时条件已经不再满足，线程应该继续睡眠但是却继续往下运行的一种现象。

解决方式：将判断线程等待的条件由if换成while。原因是代码在哪里等待，唤醒时就会在哪里继续执行下去。如果用if的话，线程醒来就不会再判断是否有可运行的环境，它会直接运行下去；如果用while的话，线程醒来，会走while中的条件判断，当不满足while的条件时会跳出while循环，从而继续执行下面的代码，反之会继续执行wait()导致线程等待。





## 17.线程组

说明：Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理，Java允许

程序直接对线程组进行控制。默认情况下，所有的线程都属于主(main)线程组。

Thread类相关方法：

构造方法：

public Thread([ThreadGroup](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ThreadGroup.html) group,[Runnable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Runnable.html) target)

public Thread([ThreadGroup](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ThreadGroup.html) group,[Runnable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Runnable.html) target,[String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)

public Thread([ThreadGroup](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ThreadGroup.html) group,[String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)

以上三个构造方法都可以指定线程归属于哪个线程组。

方法：

public final [ThreadGroup](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ThreadGroup.html) getThreadGroup()：返回该线程所属的线程组。如果该线程

已停止运行，该方法返回null。

ThreadGroup类相关方法：

构造方法：

public ThreadGroup([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)：构造一个新线程组。新线程组的父线程组是目前

正在运行线程的线程组。

public ThreadGroup([ThreadGroup](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ThreadGroup.html) parent,[String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) name)：创建一个新线程组。新线程

组的父线程组是指定的线程组。

方法：

public final [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) getName()：返回此线程组的名称。

public final void setDaemon(boolean daemon)：若daemon为true，则该组中的所有

线程变为守护线程(后台线程)。

public final void setMaxPriority(int pri)：统一设置该组中的所有线程的优先级，该组

中已有更高优先级的线程不受影响。如果pri小于1或大于10，则组中的所有

线程的优先级不会发生变化。

public final void destroy()：销毁线程组及其子组，前提是该组以及子组中已没有还

在运行的线程。

public boolean isDestroyed()：判断此线程组是否已经被销毁。

public final void interrupt()：此方法将对此线程组及其所有子组中的所有线程调用

interrupt 方法。

## 18.线程池

说明：创建和销毁线程比较消耗资源，因此可以考虑创建线程池，让线程可以反复使用，不

用的线程放到线程池中，用的时候，再从池子中拿。JDK5之后，Java内置线程池。

JDK5新增了一个Executors工厂类来产生线程池，方法如下：

public static [ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html) newCachedThreadPool()：创建一个可缓存的线程池，其容纳

的线程数会在运行时动态调整。

public static [ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html) newFixedThreadPool(int nThreads)：创建线程池，nThreads

表示池中可容纳的线程数。nThreads<=0时报[IllegalArgumentException](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/IllegalArgumentException.html)。

public static [ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html) newSingleThreadExecutor()：创建一个只容纳一个线程的线程

池，它可以保证先进先出的执行顺序。

这些方法返回的[ExecutorService](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/ExecutorService.html)(接口)对象，表示一个线程池，可以执行Runnable对象

或者Callable对象代表的线程。

其他几种方式：

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)：创建一

个可以执行延迟任务的线程池。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,

ThreadFactory threadFactory)：创建一个可以执行延迟任务的线程池。

public static ScheduledExecutorService newSingleThreadScheduledExecutor()：创建一个单

线程的可以执行延迟任务的线程池。

public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory

threadFactory)：创建一个单线程的可以执行延迟任务的线程池。

public static ExecutorService newWorkStealingPool()：创建一个抢占式执行的线程池（任

务执行顺序不确定）【JDK 1.8 添加】。

ExecutorService接口中的方法：

[Future](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Future.html)<?> submit([Runnable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Runnable.html) task)：提交一个 Runnable 任务用于执行，并返回一个表示

该任务的 Future。该Future的get方法在成功完成时将会返回 null。

<T> [Future](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Future.html)<T> submit([Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)<T> task)：提交一个[Callable](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/util/concurrent/Callable.html)任务用于执行，返回一个表

示任务的未决结果的 Future。

void execute(Runnable command)：提交一个 Runnable 任务用于执行。此方法继承自

Executor接口。

void shutdown()：启动一次顺序关闭，执行以前提交的任务，但不接受新任务。如果已

经关闭，则调用此方法无效果。

补充：Executors创建线程池是通过ThreadPoolExecutor的方式。线程池不推荐使用Executors

去创建，而是通过ThreadPoolExecutor的方式，这样可以更清晰线程池的运行规则，避

免资源耗尽的风险。Executors的主要缺点：

(1)newFixedThreadPool和newSingleThreadExecutor使用的阻塞队列都是

LinkedBlockingQueue-链表阻塞队列，其默认容量为Integer.MAX\_VALUE，可能会堆

积大量的请求，导致OOM。

(2)newCachedThreadPool和newScheduleThreadPool允许创建的线程数量为

Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM。

Executors创建线程池及调用示例(简单展示)：

//容纳单个线程  
ExecutorService threadPool = Executors.*newSingleThreadExecutor*();  
//容纳固定线程数  
//ExecutorService threadPool = Executors.newFixedThreadPool(5);  
//运行时动态调整线程池的容量  
//ExecutorService threadPool = Executors.newCachedThreadPool();  
try{  
 for(int i=0;i<100;i++){  
 threadPool.execute(()->{  
 System.*out*.println(Thread.*currentThread*().getName()+"OK");  
 });  
 }  
}catch (Exception e){  
 e.printStackTrace();  
}finally {  
 //关闭线程池  
 threadPool.shutdown();  
}

## 19. ThreadPoolExecutor

说明：Executors中创建线程池的方法，底层都是通过调用ThreadPoolExecutor类实现的。

源码调用示例：

Executors.newSingleThreadExecutor();对应：

public static ExecutorService newSingleThreadExecutor() {  
 return new FinalizableDelegatedExecutorService  
 (new ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingQueue<Runnable>()));  
}

Executors.newFixedThreadPool(5); 对应：

public static ExecutorService newFixedThreadPool(int nThreads) {  
 return new ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingQueue<Runnable>());  
}

Executors.newCachedThreadPool();对应：

public static ExecutorService newCachedThreadPool() {  
 return new ThreadPoolExecutor(0, Integer.*MAX\_VALUE*,  
 60L, TimeUnit.*SECONDS*,  
 new SynchronousQueue<Runnable>());  
}

ThreadPoolExecutor类创建线程池的源码示例：

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 if (corePoolSize < 0 ||  
 maximumPoolSize <= 0 ||  
 maximumPoolSize < corePoolSize ||  
 keepAliveTime < 0)  
 throw new IllegalArgumentException();  
 if (workQueue == null || threadFactory == null || handler == null)  
 throw new NullPointerException();  
 this.acc = System.*getSecurityManager*() == null ?  
 null :  
 AccessController.*getContext*();  
 this.corePoolSize = corePoolSize;  
 this.maximumPoolSize = maximumPoolSize;  
 this.workQueue = workQueue;  
 this.keepAliveTime = unit.toNanos(keepAliveTime);  
 this.threadFactory = threadFactory;  
 this.handler = handler;  
}

构造方法7大形参解析：

**(1) corePoolSize 线程池核心线程大小**

corePoolSize 是线程池中的一个最小的线程数量，即使这些线程处理空闲状态，他们也不会被销毁，除非设置了allowCoreThreadTimeOut。

核心线程池大小，即在没有任务需要执行的时候线程池的大小，并且只有在工作队列满了的情况下才会创建超出这个数量的线程。这里需要注意的是：在刚刚创建ThreadPoolExecutor的时候，线程并不会立即创建，而是要等到有任务提交时才会创建，除非调用了prestartCoreThread/prestartAllCoreThreads事先创建核心线程。再考虑到keepAliveTime和allowCoreThreadTimeOut超时参数(executor.allowCoreThreadTimeOut(true))的影响，所以没有任务需要执行的时候，线程池的大小不一定是corePoolSize。

**(2) maximumPoolSize 线程池最大线程数量**

线程池能够容纳同时执行的最大线程数,此值大于等于1。一个任务被提交到线程池以后，首先会找有没有空闲并且存活线程，如果有则直接将任务交给这个空闲线程来执行，如果没有则会放到工作队列中，直到工作队列满了，才会创建一个新线程，然后从工作队列的头部取出一个任务交由新线程来处理，而将刚提交的任务放入工作队列尾部。线程池不会无限制的去创建新线程，它会有一个最大线程数量的限制，这个数量即由maximunPoolSize指定。工作队列满，且线程数等于最大线程数，此时再提交任务则会调用拒绝策略。

这里值得一提的是largestPoolSize，该变量记录了线程池在整个生命周期中曾经出现的最大线程个数。为什么说是曾经呢？因为线程池创建之后，可以调用setMaximumPoolSize()改变运行的最大线程的数目。

**(3) keepAliveTime 多余的空闲线程存活时间**

当线程空闲时间达到keepAliveTime值时,多余的线程会被销毁直到只剩下corePoolSize个线程为止。默认情况下:只有当线程池中的线程数大于corePoolSize时keepAliveTime才会起作用，直到线程中的线程数不大于corePoolSIze。

**(4) unit 空闲线程存活时间单位**

unit是keepAliveTime的计量单位。有以下七种种计量单位：

TimeUnit:days:天

TimeUnit:Hours:小时

TimeUnit:Minutes:分钟

TimeUnit:seconds:秒

TimeUnit:milliseconds:毫秒

TimeUnit:microseconds:微秒

TimeUnit:nanoseconds:纳秒

**(5) workQueue 工作队列**

任务被提交给线程池时，会先进入工作队列，任务调度时再从工作队列中取出。

常用工作队列有以下几种：

<1> ArrayBlockingQueue（数组的有界阻塞队列）

ArrayBlockingQueue 在创建时必须设置大小，按FIFO排序（先进先出）。新任务进来后，会放到该队列的队尾，有界的数组可以防止资源耗尽问题。当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

<2>LinkedBlockingQueue（链表的无界阻塞队列）

按 FIFO 排序任务，可以设置容量(有界队列)，不设置容量则默认使用 Integer.Max\_VALUE 作为容量 （无界队列）。该队列的吞吐量高于 ArrayBlockingQueue。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。有两个快捷创建线程池的工厂方法 Executors.newSingleThreadExecutor、Executors.newFixedThreadPool，使用了这个队列，并且都没有设置容量（无界队列）。

<3>SynchronousQueue（一个不缓存任务的阻塞队列）

生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。其 吞 吐 量 通 常 高 于LinkedBlockingQueue。 快捷工厂方法 Executors.newCachedThreadPool 所创建的线程池使用此队列。与前面的队列相比，这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

<4>PriorityBlockingQueue（具有优先级的无界阻塞队列）

优先级通过参数Comparator实现。

<5>DelayQueue（这是一个无界阻塞延迟队列）

底层基于 PriorityBlockingQueue 实现的，队列中每个元素都有过期时间，当从队列获取元素（元素出队）时，只有已经过期的元素才会出队，而队列头部的元素是过期最快的元素。快捷工厂方法 Executors.newScheduledThreadPool 所创建的线程池使用此队列。

Java 中的阻塞队列（BlockingQueue）与普通队列相比，有一个重要的特点：在阻塞队列为空时，会阻塞当前线程的元素获取操作。具体来说，在一个线程从一个空的阻塞队列中取元素时，线程会被阻塞，直到阻塞队列中有了元素；当队列中有元素后，被阻塞的线程会自动被唤醒（唤醒过程不需要用户程序干预）。

**(6) threadFactory 线程工厂**

创建一个线程工厂用来创建线程，可以用来设定线程名、是否为daemon线程等等。

**(7) handler 拒绝策略**

AbortPolicy：丢弃任务并抛出 RejectedExecutionException 异常。（线程池默认这种）。

DiscardPolicy：默默丢弃任务，不进行任何通知。

DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程） 。也就是当任务被拒绝添加时，会抛弃任务队列中最旧的任务也就是最先加入队列的，再把这个新任务从队尾添加进去，等待执行。（丢弃掉在队列中存在时间最久的任务）

CallerRunsPolicy：谁调用，谁处理。由调用线程（即提交任务给线程池的线程）处理该任务，如果线程池已经被shutdown则直接丢弃。

自定义一个线程池：

ExecutorService threadPool = new ThreadPoolExecutor(  
 2,  
 5,  
 3,  
 TimeUnit.*SECONDS*,  
 new LinkedBlockingQueue<>(),  
 Executors.*defaultThreadFactory*(),  
 new ThreadPoolExecutor.AbortPolicy()  
);

如何选定最大核心线程数的数量呢？

答：分CPU密集型和IO密集型两种。

CPU密集型时，cpu的逻辑处理线程数就是最合适的最大核心线程数。比如：6核12线程的cpu，逻辑处理线程数就是12，Runtime.getRuntime().availableProcessors()可以获取cpu的逻辑处理线程数，所以填写最大核心线程数时，不要写死，用方法来获取。

IO密集型时，需要判断程序中十分耗费IO 的线程，然后设置最大核心线程数大于这个线程数即可，可以是两倍。

# 定时器

## 1.说明

定时器是一个应用十分广泛的线程工具，可用于调用多个定时任务，以后台线程的方式执行。

在Java中，可以通过Timer(负责定时)和TimerTask(负责任务)类来实现定义调度的功能。

实际开发中可以使用Quartz这个开源框架实现定时器的功能。

## 2.Timer(JDK1.3出现)类介绍

构造方法：

public Timer()：创建一个新计时器。相关的线程不作为守护程序运行。

public Timer(boolean isDaemon)：创建一个新计时器，isDaemon为true时表示将相关线

程作为守护线程。

public Timer([String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name)：创建一个新计时器，name是相关线程的名称。

public Timer([String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name, boolean isDaemon)：创建一个新计时器，name是相关线程的

名称，isDaemon为true时表示将相关线程作为守护线程。

方法：

public void schedule([TimerTask](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/TimerTask.html) task, long delay)：安排在指定延迟后执行指定的任务。

task - 所要安排的任务。delay - 执行任务前的延迟时间，单位是毫秒。

public void schedule([TimerTask](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/TimerTask.html) task,long delay,long period)：安排指定的任务从指定的延迟

后开始进行重复的固定延迟执行。task - 所要安排的任务。 delay - 执行任务前的

延迟时间，单位是毫秒。 period - 执行各后续任务之间的时间间隔，单位是毫秒。public void schedule([TimerTask](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/TimerTask.html) task,[Date](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Date.html) time)：安排在指定的时间执行指定的任务。如

果此时间已过去，则安排立即执行该任务。task - 所要安排的任务。 time - 执行

任务的时间。

public void schedule([TimerTask](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/TimerTask.html) task,[Date](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.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。task - 所要安排的任务。firstTime - 首次

执行任务的时间。period - 执行各后续任务之间的时间间隔，单位是毫秒。

public void cancel()：终止此计时器，丢弃Timer中所有当前已安排的任务，但不会影响

当前正在执行的任务。该方法同时会终止它的执行线程。可以重复调用此方法，但

第二次和后续调用无效。

## 3.TimerTask(JDK1.3出现)抽象类介绍

构造方法：

protected TimerTask()：创建一个新的计时器任务。

方法：

public abstract void run()：此计时器任务要执行的操作。

public boolean cancel()：取消此计时器的后续任务，但当前正在运行的任务可以运行完。

只会取消所在TimerTask对象中的任务，而它所在的Timer中的其他对象的任务不

受影响。

## 4.定时器任务的执行流程。

A：定义一个类来继承TimerTask并重写run方法，方法内为定时执行的任务。

B：创建Timer对象，并使用该对象的schedule方法来执行继承TimerTask类的任务。

C：使用Timer对象调用cancel方法终止计时器。

## 5.代码参考，代码中没有终止定时任务。



# 设计模式

## 1.面向对象思想设计原则

(1)单一职责：

就是高内聚，低耦合。每个类应该只有一个职责，对外提供一种功能。

(2)开闭原则

一个对象对扩展开放，对修改关闭。一旦写出了可以运行的代码，就不应该去带动它。

这就要求将可能变化的内容抽象出来。而具体的实现是可以改变和扩展的。

(3)里氏替换原则

在任何父类出现的地方都可以用它的子类来替代。也就是说同一个继承体系中的对象应

该有共同的行为特征。

(4)依赖注入原则

要依赖于抽象，不要依赖于具体实现。也就是说所有的类如果使用或依赖于其他的类，

则应该依赖这些其他类的抽象类，而不是这些其他类的具体类。为了实现这一原则，就

要求我们在编程的时候针对抽象类或者接口编程，而不是针对具体实现编程。

(5)接口分离原则

不应该强迫程序依赖它们不需要使用的方法。也就是说一个接口不需要提供太多的行为，

一个接口应该只提供一种对外的功能,不应该把所有的操作都封装到一个接口中。

(6)迪米特原则

一个对象应当对其他对象尽可能少的了解。也就是说降低各个对象之间的耦合，提高系

统的可维护性。在模块之间应该只通过接口编程，而不理会模块的内部工作原理，它可

以使各个模块耦合度降到最低，促进软件的复用。

## 2.设计模式概述

设计模式(Design pattern)是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

设计模式不是一种方法和技术，而是一种思想。设计模式和具体的语言无关，学习设计模式就是要建立面向对象的思想，尽可能的面向接口编程，低耦合，高内聚，使设计的程序可复用。学习设计模式能够促进对面向对象思想的理解，反之亦然，他们相辅相成。

## 3.设计模式的分类

创建型模式 – 对象的创建

简单工厂模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式、单例模式

结构型模式 – 对象的组成(结构)

外观模式、适配器模式、代理模式、装饰模式、桥接模式、组合模式、享元模式

行为型模式 – 对象的行为

模板方法模式、观察者模式、状态模式、职责链模式、命令模式、访问者模式、策略模

式、备忘录模式、迭代器模式、解释器模式。

## 4.简单工厂模式

(1)动物抽象类：public abstract Animal { public abstract void eat(); }

(2)具体狗类：public class Dog extends Animal{}

(3)具体猫类：public class Cat extends Animal{}

(4)具体工厂类：public class AnimalFactorty{}

在该类中，私有化构造方法，然后写几个静态的方法用来创建狗类和猫类的对象。

优点：客户端不需要负责对象的创建，明确了各个类的职责。

缺点：如果有新的对象增加，或者某些对象的创建方式不同，需要不断的修改工厂类，不利

于后期的维护。

## 5.工厂方法模式

(1)动物抽象类：public abstract Animal { public abstract void eat(); }

(2)具体狗类：public class Dog extends Animal{}

(3)具体猫类：public class Cat extends Animal{}

(4)工厂接口：public interface Factory{ public abstract Animal createAnimal(); }

(5)具体狗的工厂类：public class DogFactory implements Factory{}

(6)具体猫的工厂类：public class CatFactory implements Factory{}

优点：客户端不需要负责对象的创建，明确了各个类的职责。如果有新的对象增加，只需要

增加一个具体的类和具体的工厂类即可，不影响已有的代码，后期维护容易，增强了系统的扩展性。

缺点：需要额外的编写代码，增加了工作量。

## 6.单例设计模式

概述：确保类在内存中只有一个对象，该实例必须自动创建，并且对外提供。

(1)饿汉式(开发用它)

具体学生类：

public class Student{

private Student(){}

private static Student s = new Student();

public static Student getStudent(){ return s; }

}

这样，别人只能够调用静态的getStudent()来获取唯一一个Student对象。且只要该类一

加载，立马就会创建Student对象。

(2)懒汉式(面试用它)

具体学生类：

public class Student{

private Student(){}

//用volatile修饰是为了在new Student()过程中禁止指令重排，从而避免发生创建对象

过程中的错误

private volatile static Student s = null;

public static Student getStudent(){

//使用双重if判断的方式被称为DCL懒汉式

if(s==null){

synchronized(Student.class){

if(s == null){

s = new Student();

}

}

}

return s;

}

}

这样，别人只能够调用静态的getStudent()来获取唯一一个Student对象。但是Student

对象的创建会等到别人调用getStudent()时才会创建。

优点：在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销

毁的对象单例模式无疑可以提高系统的性能。

缺点：没有抽象层，因此扩展很难。职责过重，在一定程度上违背了单一职责。

具体应用：JDK中的Runtime类使用的是单例模式的饿汉式。

## 7. Runtime类的使用

方法：

public static [Runtime](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Runtime.html) getRuntime()：返回与当前 Java 应用程序相关的 Runtime 对象。

public [Process](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Process.html) exec([String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) command)：在单独的进程中执行指定的字符串命令。

代码实例：

Runtime r = Runtime.getRuntime();

r.exec(“calc”);//调用计算器

r.exec(“shutdown –s –t 10000”);//10000秒后关机

r.exec(“shutdown –a”);//取消关机

## 8.模板设计模式

说明：就是定义一个算法的骨架，而将具体的算法延迟到子类中来实现。

代码示例：

(1)public abstract class GetTime{

public long getTime(){

long start = System.currentTimeMillis();

code();

long end = System.currentTimeMillis();

return end – start;

}

public abstract code();

}

(2)public class ForDemo extends GetTime{

@Override

public void code(){

for(int x = 0; x<10000;x++){

System.out.println(x);

}

}

}

(3)public class GetTimeDemo{

public static void main(String[] args){

GetTime gt = new ForDemo();

System.out.println(gt.getTime() + “毫秒”);

}

}

(4)你可以定义更多的类来继承GetTime以便重写code方法，这样在测试类中使用多态，就

可以计算各种重写的code方法的运行时间。

优点：使用模版方法模式，在定义算法骨架的同时，可以很灵活的实现具体的算法，满足用

户灵活多变的需求。

缺点：如果算法骨架有修改的话，则需要修改抽象类。

## 9.装饰设计模式

说明：就是使用被装饰类的一个子类的实例，在客户端将这个子类的实例交给装饰粪。是继

承的替代方案。

代码示例：

(1)interface Robot{

void doSomething();

}

(2)class FirstRobot implements Robot{

@Override

public void doSomething(){

System.out.println(“对话”);

System.out.println(“唱歌”);

}

}

(3)class RobotDecorator implements Robot{

private Robot robot;

public RobotDecorator(Robot robot){

this.robot = robot;

}

@Override

public void doSomething(){

robot.doSomething();

}

public void doMoreThing(){

robot.doSomething();

System.out.println(“拖地”);

}

}

(4)public class Test(){

public static void main(String[] args){

new RobotDecorator(new FirstRobot()).doMoreThing();

}

}

(5) RobotDecorator类也可以是抽象的，定义它的子类去实现更多的装饰器。Java的IO流大

量的使用了装饰设计模式。

优点：使用装饰模式，可以提供比继承更灵活的扩展对象的功能，它可以动态的添加对象的

功能，并且可以随意的组合这些功能。

缺点：因为可以随意组合，所以可能出现一些不合理的逻辑。

# 网络编程(了解)

## 1.计算机网络

是指将地理位置不同的具有独立功能的多台计算机及其外部设备，通过通信线路连接起来，

在网络操作系统，网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息

传递的计算机系统。

## 2.网络编程

就是用来实现网络互连的不同计算机上运行的程序间可以进行数据交换。

## 3.网络模型七层概述

(1)物理层:主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的

传输速率等。它的主要作用是传输比特流(就是由1、0转化为电流强弱来进行传输，

到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换)。这一层的数据

叫做比特。

(2)数据链路层:主要将从物理层接收的数据进行MAC地址(网卡的地址)的封装与解封装。常

把这一层的数据叫做帧。在这一层工作的设备是交换机，数据通过交换机来传输。

(3)网络层:主要将从下层接收到的数据进行IP地址(例192.168.0.1)的封装与解封装。在这一

层工作的设备是路由器，常把这一层的数据叫数据包。

(4)传输层:定义了一些传输数据的协议和端口号(www端口80等)，如:TCP(传输控制协议，传

输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据)，UDP(用户数据报协

议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如Q聊天数

据就是通过这种方式传输的)。主要是将从下层接收的数据进行分段和传输，到达目的

地址后再进行重组。常常把这一层数据叫做段。

(5)会话层:通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的

系统之间发起会话或者接受会话请求(设备之间需要互相认识可以是IP也可以是MAC

或者是主机名)

(6)表示层:主要是进行对接收的数据进行解释、加密与解密、压缩与解压缩等(也就是把计

算机能够识别的东西转换成人能够能识别的东西(如图片、声音等)。

(7)应用层:主要是一些终端的应用，比如说FTP(各种文件下载)，WEB(IE浏览)，QQ之类的

(可以把它理解成我们在电脑屏幕上可以看到的东西．就是终端应用）。

## 4.七层协议



## 5.网络编程三要素

A：IP地址

网络中计算机的唯一标识。比如：192.168.1.100。计算机只能识别二进制数据，

192.168.1.100可以和11000000 10101000 00000001 01100100互相转化，之所以用

192.168.1.100(点分十进制)只是为了好记。

私有地址：指在互联网上不使用，而被用在局域网上。

保留地址：不怎么使用的地址，有专门的用途。

IP地址的分类：

A类1.0.0.1 --- 127.255.255.254 私有地址：10.X.X.X

B类128.0.0.1 --- 191.255.255.254私有地址：172.16.0.0 --- 172.31.255.255

保留地址：169.254.X.X

C类192.0.0.1 --- 223.255.255.254私有地址：192.168.X.X

该类地址最常见

D类 224.0.0.1 --- 239.255.255.254 为保留地址，多用于多点广播

E类 240.0.0.1 --- 255.255.255.254 为保留地址，用于科学实验

IP地址的组成：

A类：第一号段为网络号段+后三段的主机号段

一个网络号配：256\*256\*256=16777216个主机

B类：前二号段为网络号段+后二段的主机号段

一个网络号配：256\*256=65536个主机

C类：前三号段为网络号段+后一段的主机号段

一个网络号配：256个主机

两个DOS命令：ipconfig 查看本机ip地址

ping ip/ping 域名 测试本机与指定的ip地址或域名之间的通信是否有

问题

特殊的IP地址：

127.0.0.1 回环地址(表示本机)

x.x.x.255 广播地址 - 用于向所属网络中所有工作站进行发送信息的一个地址

x.x.x.0 网络地址

B：端口

正在运行的程序的标识。

有效端口：0 ~ 65535，其中0~1024为系统使用或保留端口。

C：协议

通信的规则。

UDP协议：

将数据源和目的封装成数据包，不需要建立连接；每个数据包的大小限制在

64k；因无连接，所以是不可靠协议，但也因此速度快。

TCP协议：

建立连接，形成传输数据的通道；在连接中进行大数据量传输；通过三次握竽完成

连接，是可靠协议；必须建立连接，因此效率会稍低

## 6.Socket套接字

说明：

网络上具有唯一标识的IP地址和端口号组合在一起的数据就是Socket套接字。

原理机制：

通信的两端都有Socket。

网络通信其实就是Socket间的通信。

数据在两个Socket间通过IO流传输。

## 7.InetAddress类

方法：

public static [InetAddress](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/InetAddress.html) getByName([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) host)：根据主机名返回类对象，主机名可以是

计算机名或者IP地址的文本形式。

public [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) getHostName()：返回主机名

public [String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) getHostAddress()：返回文本形式的IP地址

## 8.UDP协议发送与接收相关的类

(1)DatagramSocket类

说明：用于UDP协议发送和接收数据包。

构造方法：

public DatagramSocket()：构造数据报套接字并将其绑定到本地主机上任何可用的端口。

public DatagramSocket(int port)：创建数据报套接字并将其绑定到本地主机上的指定端口

port上。一般用于接收方。

方法：

public void close()：关闭此数据报套接字。

public void send([DatagramPacket](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/DatagramPacket.html) p)：从此套接字发送数据报包。DatagramPacket 包含的

信息指示：将要发送的数据、其长度、远程主机的 IP 地址和远程主机的端口号。public void receive([DatagramPacket](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/DatagramPacket.html) p)：从此套接字接收数据报包。此方法在接收到数据

报前一直阻塞。

(2)DatagramPacket类

说明：此类表示数据报包。

构造方法：

public DatagramPacket(byte[] buf,int length,[InetAddress](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/InetAddress.html) 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。buf - 包数据。length -

包长度。address - 目的地址。port - 目的端口号。

public DatagramPacket(byte[] buf,int length)：构造 DatagramPacket，用来接收长度为

length 的数据包。

方法：

public byte[] getData()：返回用来接收或发送数据的缓冲区。

public int getLength()：返回将要发送或接收到的数据的实际长度。

public [InetAddress](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/InetAddress.html) getAddress()：返回地址对象，该对象表示的机器为此数据报将要发往

或接收到的地方。

## 9.UDP发送与接收信息的代码示例

发送方：

public class SendDemo{

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

DatagramSocket ds = new DatagramSocket();

byte[] bys = “hello”.getBytes();

int length = bys.length;

//如果把192.168.12.92改为192.168.12.255(广播地址)，则

192.168.12.1~192.168.12.254的机器都可以收到发送的信息！

InetAddress address = InetAddress.getByName(“192.168.12.92”);

int port = 10086;

DatagramPacket dp = new DatagramPacket(bys,length,address,port);

ds.send(dp);

ds.close();

}

}

接收方：

public class ReceiveDemo{

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

DatagramSocket ds = new DatagramSocket(10086);

byte[] bys = new byte[1024];

DatagramPacket dp = new DatagramPacket(bys, bys.length);

ds.receive(dp);

InetAddress address = dp.getAddress();

String ip = address.getHostAddress();

byte[] bys2 = dp.getData();

int len = dp.getLength();

String s = new String(bys2,0,len);

System.out.println(ip + “传递的数据是：” + s);

//真正在服务器时，接收方任何时候都不会关闭，这里关闭只是为了演示

ds.close();

}

}

注意：重复运行接收方类会报错：Address already in user(端口被占用)。

## 10. TCP协议发送与接收相关的类

(1)Socket类

说明：此类实现客户端套接字。

构造方法：

public Socket([InetAddress](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/InetAddress.html) address,int port)

public Socket([String](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/String.html) host,int port)

方法：

public [OutputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/OutputStream.html) getOutputStream()：返回此套接字的输出流。可以用write系列方法

将字节写入到该输出流中。

public [InputStream](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/io/InputStream.html) getInputStream()：返回此套接字的输入流。

public boolean isClosed()：返回套接字的关闭状态。如果已关闭了套接字，返回true。

public void close()：关闭此套接字。

public void shutdownOutput()：禁用此套接字的输出流。对于 TCP 套接字，任何以前写

入的数据都将被发送，并且后跟 TCP 的正常连接终止序列。 如果在套接字上调用

shutdownOutput() 后写入套接字输出流，则该流将抛出 IOException。这个方法用

于数据传输完成后关闭通道流，然后服务端就不会一直等待接收通道中的数据。

(2)ServerSocket类

说明：此类实现服务器套接字。

构造方法：

public ServerSocket(int port)：创建绑定到特定端口的服务器套接字。

方法：

public [Socket](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/Socket.html) accept()：侦听并接受到此套接字的连接。此方法在连接传入之前一直阻塞。

public [InetAddress](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/net/InetAddress.html) getInetAddress()：返回此服务器套接字的本地地址。

## 11. TCP发送与接收信息的代码示例

(1)客户端(发送方)：

public class ClientDemo{

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

Socket s = new Socket(“192.168.12.92”,8888);

OutputStream os = s.getOutputSream();

os.write(“hello”.getBytes());

//客户端除了可以发送信息给服务端，也可以接收服务端发送的信息

InputStream is = s.getInputStream();

byte[] bys = new byte[1024];

int len = is.read(bys); //read()会阻塞

String client = new String(bys,0,len);

System.out.println(“client:”+client);

s.close();

}

}

(2)服务端(接收方)：

public class ServerDemo{

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

ServerSocket ss = new ServerSocket(8888);

Socket s = ss.accept();//accept()会阻塞

InputStream is = s.getInputStream();

byte[] bys = new byte[1024];

int len = is.read(bys);//read()会阻塞

String str = new String(bys,0,len);

System.out.println(str);

//服务端除了可以接收信息，也可以发送信息给客户端

OutputStream os = s.getOutputStream();

os.write(“数据已收到”.getBytes());

s.close();

//服务器不应该关闭，应该一直开着

//ss.close();

}

}

注意：先运行客户端会报错：Connection refused: connect(连接被拒绝)。

# 反射

## 1.类的加载

注意：类的加载是在编译之后！编译会根据类(.java文件)生成对应的.class文件。

当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化

三步来实现对这个类进行初始化。

(1)加载

过程：将.class文件转化为二进制字节流读入内存的方法区，并通过类加载器为之创建一个

代表该类的Class对象，Class对象存在于堆中。

如果保证在多线程的情况下，一个类只会生成一个Class对象呢？

通过synchronized锁。下图中name即为被加载的类的全路径。



任何类被使用时系统都会建立一个Class对象。

分类：

静态加载：编译时加载相关的类，如果没有则报错，依赖性太强。

动态加载：运行时加载需要的类，即便没有，只要运行时不用该类，则不报错，降低了

依赖性。

当加载完成后，内存会生成方法区和堆。方法区中有字节码文件(.class文件)的二进制数据形

式，堆中有对应的Class对象。



补充：堆区存放new的对象和数组，可以被所有线程共享；栈区存放基本变量类型(会包含这个基本类型的具体数值)和引用对象的变量(存放这个引用在堆里面的具体地址)；方法区是一个特殊的堆，可以被所有线程共享，包含了所有的static变量、常量、方法。

(2)连接

验证：

目的：确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚

拟机自身的安全。

包括：文件格式验证(是否以魔数oxcafebabe开头)、元数据验证、字节码验证和符合引

用验证。以魔数oxcafebabe开头意味着该文件为.class文件。

可以考虑使用-Xverify:none参数来关闭大部分的类验证措施，缩短虚拟机类加载的时间。

准备：

JVM会在该阶段对静态变量，分配内存并默认初始化(对应数据类型的默认初始值，如0、

0L、null、false等)。这些变量所使用的内存都将在方法区中进行分配。

举例：

public int n1 = 10;

public static int n2 = 20;

public static fial int n3 = 30;

说明：n1是实例变量，不是静态变量，因此在准备阶段，不会被分配内存。

n2是静态变量，在准备阶段分配内存，默认初始化其值为0，等到初始化阶段，

才会被再次赋值为20。

n3是静态常量，在准备阶段分配内存，直接赋值为30。

解析：

虚拟机将常量池中类的二进制数据中的符号引用替换为直接引用。

符号引用就是用符号来表示一个类，直接引用就是用内存地址来表示一个类。

连接会将类的二进制数据合并到JRE中。

(3)初始化

到初始化阶段，才真正开始执行类中定义的Java程序代码，此阶段是执行<clinit>()方法

的过程。

<clinit>()方法是由编译器按语句在源文件中出现的顺序，依次自动收集类中的所有静态

变量的赋值动作和静态代码块中的语句，并进行合并。

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁、同步，如果多个线

程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。

## 2.类的加载时机

(1)创建类的实例，也就是new一个对象 --属于静态加载

(2)访问类的静态变量，或者为静态变量赋值

(3)调用类的静态方法

(4)使用反射方式来强制创建某个类或接口对应的java.lang.Class对象 --属于动态加载

(5)初始化某个类的子类

(6)直接使用java.exe命令来运行某个主类 --属于静态加载

## 3.类加载器

作用：负责将.class文件加载到内存中，并为之生成对应的Class对象。所以Class对象是系

统创建的，只创建一个，存在于堆内存。一个类的Class对象无论怎么获取，都是同一

个。

(1)类加载器的组成

Bootstrap ClassLoader根类加载器

也被称为引导类加载器，无法直接获取。负责Java核心类的加载，比如System.String等。在

JDK中JRE的lib目录下rt.jar文件中，为C++编写。

Extension ClassLoader扩展类加载器

负责JRE的扩展目录中jar包的加载。在JDK中JRE的lib目录下ext目录。

Sysetm ClassLoader系统类加载器

负责在JVM启动时加载来自java命令的class文件，以及classpath环境变量所指定的jar

包和类路径。

代码获取：

//获取系统类加载器  
ClassLoader systemClassLoader = ClassLoader.*getSystemClassLoader*();  
//获取扩展类加载器  
ClassLoader extensionClassLoader = systemClassLoader.getParent();  
//无法获取根类加载器，其值为null  
ClassLoader parent = extensionClassLoader.getParent();

(2)类缓存

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

(3)获取系统属性(额外内容)

System类有一个方法：public static [Properties](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/Properties.html) getProperties()，此方法以key-value的格式返回当前

的所有系统属性。

System类还有一个方法：public static [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getProperty([String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) key)，此方法根据key拿到对象的系统

属性，这个key就是getProperties()方法返回的key-value格式数据中的key。

案例：打印java加载的所有类的类路径

System.*out*.println(System.*getProperty*("java.class.path"));

## 4.反射的概述

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

## 5.反射的使用

(1)首先需要得到class文件对象，也就是Class类的对象

Class类：

成员变量 Field

构造方法 Constructor

成员方法 Method

(2)获取class文件对象的方式

A：Object类的getClass()方法 运行阶段

B：数据类型的静态属性class 加载阶段

C：Class类中的静态方法 编译阶段

public static Class<?> forName(String className)

补充：className是全路径，包含包名。

实际应用中第二种更方便。但开发选第三种，因为直观。

D：基本数据类型可以用.class得到Class对象，该对象实际是其包装类的Class对象。

Class<Integer> integerClass = int.class

E：基本数据类型的包装类可以用.TYPE得到其Class对象。

Class<Integer> integerClass = Integer.TYPE

(3)如何操作构造方法

Class类的方法：

a.public [Constructor](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html)<?>[] getConstructors()//获取本类所有公共(public修饰的)构造方法。

之所以此方法无法获取父类的构造方法，是因为子类无法继承父类的构造方法。

b.public [Constructor](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html)<?>[] getDeclaredConstructors()//获取本类所有构造方法。

c.public [Constructor](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html)<[T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Class.html)> getConstructor(Class<?>... parameterTypes)//获取单个公共的构造

方法。

d.public [Constructor](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html)<[T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Class.html)> getDeclaredConstructor(Class<?>... parameterTypes)//获取单个构

造方法，形参的值表示你要获取的构造方法的参数个数以及参数数据类型的class

字节码文件对象。

Constructor类的方法：

a.public [T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html) newInstance([Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html)... initargs)//根据实参的不同获取类的对象

例：

Class c = Class.forName(“cn.itcast.Person”);//实参可以通过配置文件获取，而不写死路径

Constructor con = c.getConstructor();//获取Person类的无参构造方法

Object obj = con.newInstance();//这种方式获取Person对象而不用导入Person类

//以上语句相当于

Person p = new Person();

b.public void setAccessible(boolean flag)//该方法继承自类AccessibleObject，形参为true时表示取消Java语言访问检查，可以搭配public [Constructor](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Constructor.html)<[T](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Class.html)> getDeclaredConstructor(Class<?>... parameterTypes)来访问一个类的私有构造方法。

例：

Class c = Class.forName(“cn.itcast.Person”);

Constructor con = c.getDeclaredConstructor (String.class);

con. setAccessible(true);

Object obj = con.newInstance(“风清扬”);

System.out.println(obj);

c. public int getModifiers()//以数字的形式返回该变量的修饰符。默认修饰符是0，public是1，

private是2，protected是4。

d. public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getName()：返回构造方法的名称。

e. public [Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<?>[] getParameterTypes()//返回该方法的形参类型的Class对象组成的数组。

(4)如何操作成员变量

Class类的方法：

a.public [Field](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Field.html)[] getFields()//获取本类和父类的公共(public修饰)的成员变量。

b.public [Field](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Field.html)[] getDeclaredFields()//获取本类所有成员变量

c.public [Field](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Field.html) getField([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name)//获取单个公共的成员变量，形参的值为成员变量的名

称

d.public [Field](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Field.html) getDeclaredField([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name)//获取单个成员变量

Field类的方法：

a.public void set([Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html) obj,[Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html) value)//obj是通过反射获取的类的对象，value是成员变量的值。如果该变量为静态属性，则参数obj可以写成null。

b.public void setAccessible(boolean flag)//该方法继承自类AccessibleObject，形参为true时表示取消Java语言访问检查，可以搭配public [Field](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/reflect/Field.html) getDeclaredField([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name)来访问一个类的私有成员变量。

例：

Class c = Class.forName(“cn.itcast.Person”);

Constructor con = c.getConstructor();

Object obj = con.newInstance();

Field addressField = c. getDeclaredField (“address”);

addressField. setAccessible(true);

addressField.set(obj,”北京”);//给obj对象中的字段addressField的值设为北京。

System.out.println(obj);

c. public int getModifiers()//以数字的形式返回该变量的修饰符。默认修饰符是0，public是1，

private是2，protected是4，static是8，final是16。若被多个修饰符修饰，则返回这

些修饰符代表的数字相加之和。例：public(1)+static(8)=9。

d. public [Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<?> getType()//返回该变量的类型的Class对象，比如：int的Class对象。

e. public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getName()//返回变量的名称。

f. public [Object](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Object.html) get([Object](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Object.html) obj)//获取对象上的该变量的值。如果该变量为静态属性，则参数obj可以写成null。

(5)如何操作成员方法

Class类的方法：

a.public Method [] getMethods()//获取自己的和父类的所有(public修饰的)公共方法

b.public Method [] getDeclaredMethods()//获取本类的所有方法(不包括父类的)

c.public Method getMethod([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name,Class <?>... parameterTypes)//获取单个公共的方

法。

//name是要获取的方法名称，而parameterTypes是该方法形参类型的Class对象

d.public Method  getDeclaredMethod([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name,Class <?>... parameterTypes)

//获取单个的方法，可搭配Method类的setAccessible(true)来访问私有方法

Method类的方法：

a. public [Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html) invoke([Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html) obj,[Object](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/Object.html)... args)//obj表示要调用的对象，args表示方法的实参。

该方法的返回值Object表示要调用的方法的返回值。如果此方法是静态的，则参数obj

可以写成null。

例：

Class c = Class.forName(“cn.itcast.Person”);

Constructor con = c.getConstructor();

Object obj = con.newInstance();

Method m1 = c.getMethod(“show”,String.class);

Object o = m1.invoke(obj,”方法的实参”);

以上语句相当于：

Person p = new Person(); Object o = p.show(”方法的实参”);

b.public void setAccessible(boolean flag)//该方法继承自父类AccessibleObject，形参为true时

表示取消Java语言访问检查，可以搭配Class类的public Method  getDeclaredMethod([String](mk:@MSITStore:E:\个人资料\技术文档\jdk1.8中文版.CHM::/java/lang/String.html) name,Class <?>... parameterTypes)方法来访问私有方法。

c. public int getModifiers()//以数字的形式返回该变量的修饰符。默认修饰符是0，public是1，private是2，protected是4，static是8，final是16。若被多个修饰符修饰，则返回这些修饰符代表的数字相加之和。例：public(1)+static(8)=9。

d. public [Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<?> getReturnType()//返回该方法的返回值类型的Class对象。

e. public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getName()//返回该方法的名称。

f. public [Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<?>[] getParameterTypes()//返回该方法的形参类型的Class对象组成的数组。

## 6.读取配置文件中的内容

假设项目中有个class.txt，里面的内容是：

className=cn.itcast.Student

methodName=love

那么可以使用以下代码获取其值：

Properties prop = new Properties();

FileReader fr = new FileReader(“class.txt”);

prop.load(fr);

fr.close();

String className = prop.getProperty(“className”);// cn.itcast.Student

String methodName = prop.getProperty(“methodName”);// love

## 7.通过反射可以越过泛型检查

原因：泛型的检查只存在于编译的过程中，而反射的使用是在运行过程中，运行过程中不会

检查泛型。

代码示例：

ArrayList<Integer> array = new ArrayList<Integer>();

Class c = array.getClass();

Method m = c.getMethod(“add”,Object.class);

m.invoke(array,”hello”);

System.out.println(array);

## 8.Class类的其他方法

public [T](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html) newInstance()：创建一个实例，底层仅仅通过空参构造方法创建一个实例，没有空参构

造方法则报错。

public [ClassLoader](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/ClassLoader.html) getClassLoader()：返回该类的类加载器。

public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getName()：返回全限定类名，包含包名和类名。

public [String](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) getSimpleName()：只获取类名。

public [Package](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Package.html) getPackage()：获取此类的包。

public [Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<? super [T](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)> getSuperclass()：返回超类的Class对象。

public [Class](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<?>[] getInterfaces()：返回此Class对象继承/实现的所有接口的Class对象。

public boolean isArray()：如果此对象表示一个数组类，则返回 true，否则返回 false。

public [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)[] getAnnotations()：返回此对象上存在的所有注解。

## 9.反射的优缺点

优点：可以动态的创建和使用对象，比较灵活。

缺点：使用反射为解释执行方式，因此执行速度有影响。

## 10.哪些类型有Class对象

(1) 外部类,成员内部类,静态内部类,局部内部类,匿名内部类 String.class、Class.class

(2) 接口 Serializable.class

(3) 数组 Integer[].class、float[][].class

(4) 枚举 Thread.State.class

(5) 注解 Deprecated.class

(6) 基本数据类型 int.class

(7) void void.class

## 11.反射操作泛型(了解)

(1) Java采用泛型擦除的机制来引入泛型，Java中的泛型仅仅是给编译器javac使用的，确保

数据的安全性和免去强制类型转换问题，但是，一旦编译完成，所有和泛型有关的类型

全部擦除。

(2)为了通过反射操作这些类型，Java新增了ParameterizedType、GenericArrayType、

TypeVariable和WildcardType几种类型来代表不能被归一到Class类中的类型但是又和原

始类型齐名的类型。

(3)类型介绍

ParameterizedType：表示一种参数化类型，比如Collection<String>。

GenericArrayType：表示一种元素类型是参数化类型或者类型变量的数组类型。

TypeVariable：是各种类型变量的公共父接口。

WildcardType：代表一种通配符类型表达式。

(4)操作泛型

涉及到的方法：

Method类的：

public [Type](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/reflect/Type.html)[] getGenericParameterTypes()：返回该方法的所有形参类型，如果没有形参，

返回一个空数组。

public [Type](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/reflect/Type.html) getGenericReturnType()：返回该方法的返回值的类型。

public [Type](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/reflect/Type.html)[] getGenericExceptionTypes()：返回该方法的抛出的异常的类型，如果该方法

没有throws一个异常，则返回一个空数组。

ParameterizedType接口的方法：

[Type](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/reflect/Type.html)[] getActualTypeArguments()：返回参数化类型中的泛型实际的对象类型，如果不是

参数化类型，则返回一个空数组。

代码实例：

public class AnnotationGeneric {  
 public Map<String,User> test01(Map<String,User> map, List<User> list){  
 System.*out*.println("test01");  
 return null;  
 }  
 public static void main(String[] args) throws Exception{  
 Method method = AnnotationGeneric.class.getDeclaredMethod("test01",Map.class,List.class);  
 //返回该方法中的形参类型  
 Type[] genericParameterTypes = method.getGenericParameterTypes();  
 for (Type genericParameterType : genericParameterTypes){  
 System.*out*.println(genericParameterType);  
 //如果形参类型是参数化类型，比如：Collection<String>  
 if(genericParameterType instanceof ParameterizedType){  
 //拿到里面泛型的实际类型  
 Type[] actualTypeArguments = ((ParameterizedType) genericParameterType).getActualTypeArguments();  
 for (Type actualTypeArgument : actualTypeArguments) {  
 System.*out*.println(actualTypeArgument);  
 }  
 }  
 }  
 //返回该方法中的返回值类型  
 Type genericReturnType = method.getGenericReturnType();  
 //如果返回值类型是参数化类型，比如：Collection<String>  
 if(genericReturnType instanceof ParameterizedType){  
 //拿到里面泛型的实际类型  
 Type[] actualTypeArguments = ((ParameterizedType) genericReturnType).getActualTypeArguments();  
 for (Type actualTypeArgument : actualTypeArguments) {  
 System.*out*.println(actualTypeArgument);  
 }  
 }  
 }  
}

# 注解

## 1.介绍

(1)来源：注解由jdk1.5引入。

(2)作用：可以对程序做出解释，被其他程序(比如：编译器等)读取并做相关操作。

(3)格式：注解以”@注解名称”在代码中中存在，还可以添加一些参数值。如：

@SuppressWarnings(value=”unchecked”)

(4)使用范围：可以修饰包、类、方法、属性等。可以通过反射来实现对这些元数据的访问。

## 2.jdk的内置注解

(1) @Override：定义在java.lang.Override中，此注释只适用于修辞方法，表示一个方法声明

打算重写超类中的另一个方法声明。



(2) @Deprecated：定义在java.lang.Deprecated中，此注释可以用于修辞方法、属性、类，表

示反对程序员使用这样的元素，通常是因为它很危险或者有更好的选择。



(3) @SuppressWarnings：定义在java.lang.SuppressWarnings中，用来抑制编译时的警告信息。

与前两个注释有所不同，你需要添加一个参数才能正确使用，这些参数都是已经定义好

了的，我们选择性的使用就好了。比如：@SuppressWarnings("all")、

@SuppressWarnings("unchecked")、

@SuppressWarnings(value="unchecked" ,"deprecation"})



## 3.元注解

(1)作用：

负责注解其他注解，Java定义了4个标准的meta-annotation类型，他们被用来提

供对其他注解类型做说明。

(2)位置：

在java.lang.annotation包中可以找到。分别是@Target、@Retention、@Documented、

@Inherited。

(3)使用：

@Target用于描述注解的使用范围(即:被描述的注解可以用在什么地方)，其值为一个数

组类型，也可以只赋一个值，值的含义如下：

*\** ***@Target(ElementType.TYPE)——接口、类、枚举、注解****\** ***@Target(ElementType.FIELD)——字段、枚举的常量****\** ***@Target(ElementType.METHOD)——方法****\** ***@Target(ElementType.PARAMETER)——方法参数****\** ***@Target(ElementType.CONSTRUCTOR)*** *——构造函数  
\** ***@Target(ElementType.LOCAL\_VARIABLE)——局部变量****\** ***@Target(ElementType.ANNOTATION\_TYPE)——注解****\** ***@Target(ElementType.PACKAGE)——包***

@Retention表示需要在什么级别保存该注释信息，用于描述注解的生命周期，其值为：

*\* RetentionPolicy.SOURCE:这种类型的Annotations只在源代码级别保留,编译时就会被忽略,在class字节码文件中不包含。  
\* RetentionPolicy.CLASS:这种类型的Annotations编译时被保留,默认的保留策略,在class文件中存在,但JVM将会忽略,运行时无法获得。  
\* RetentionPolicy.RUNTIME:这种类型的Annotations将被JVM保留,所以他们能在运行时被JVM或其他使用反射机制的代码所读取和使用。*

@Documented说明该注解将被包含在javadoc中；

@Inherited说明子类可以继承父类中的该注解。

## 4.自定义注解

(1) 例子

@Target({*TYPE*,*FIELD*,*METHOD*,*PARAMETER*,*CONSTRUCTOR*,*LOCAL\_VARIABLE*,*TYPE\_PARAMETER*})  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface MyAnnotation {  
 String value() default "defaultValue";  
}

(2) @interface

[使用@interface自定义注解时，会自动继承java.annotation.Annotation](mailto:使用@interface自定义注解时，会自动继承java.annotation.Annotation)接口

(3) 注意事项

@interface用来声明一个注解，格式：[public] @interface注解名{定义内容}；

其中的每一个方法实际上是声明了一个配置参数；

方法的名称就是参数的名称；

返回值类型就是参数的类型(返回值只能是基本类型：Class、String、enum)；

可以通过default来声明参数的默认值；

如果只有一个参数成员，一般参数名为value；

当只有一个参数成员，且参数名为value时，给注解赋值时可以省略value，比如：

@MyAnnotation("World")  
public void show(String str){}

注解元素必须要有值，我们定义注解元素时，经常使用空字符串、0作为默认值。

## 5.实现注解

上面的步骤仅仅是定义注解，注解的功能实现，需要通过反射来实现。

第一步是获取注解，具体方法为：

Class类的：

public <A extends [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)> A getAnnotation([Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<A> annotationClass)：通过注解类的

Class对象来得到修饰该类的注解，如果没有该类型的注解，返回null。

public [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)[] getAnnotations()：返回修饰该类的所有注解。

Method类的：

public <T extends [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)> T getAnnotation([Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<T> annotationClass)：通过注解类的

Class对象来得到修饰该方法的注解，如果没有该类型的注解，返回null。

public [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)[] getAnnotations()：返回修饰该方法的所有注解。

Field类的：

public <T extends [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)> T getAnnotation([Class](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<T> annotationClass)：通过注解类的

Class对象来得到修饰该属性的注解，如果没有该类型的注解，返回null。

public [Annotation](mk:@MSITStore:F:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/annotation/Annotation.html)[] getAnnotations()：返回修饰该属性的所有注解。

第二步是拿到注解中的信息并实现注解的功能，具体方式为：

通过上一步拿到Annotation类型对象，再直接通过里面的方法拿到对应的参数，用这

些参数来实现功能。

代码举例：

public class NewAnnotation {  
 @Table(column = "列名",num = 20)  
 String field;  
 public static void main(String[] args) throws Exception{  
 Field fe = NewAnnotation.class.getDeclaredField("field");  
 Table table = fe.getAnnotation(Table.class);  
 System.*out*.println(table.annotationType());//interface jdk8.annotation.Table  
 System.*out*.println(table.column());//列名  
 System.*out*.println(table.num());//20  
 }  
}  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target({ElementType.*FIELD*})  
@interface Table {  
 String column();  
 int num();  
}

# 动态代理

代理：本来应该自己做的事情，却请了别人来做，被请的人就是代理对象。

举例：火车票请人代买。

动态代理：在程序运行过程中产生的这个现象。

程序运行过程中产生对象其实就是反射的内容，动态代理其实就是通过反射来生成一个

代理。

## 1.Proxy类(JDK1.3出现)

在java中java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过使用

这个类和接口就可以生成动态代理对象，JDK提供的代理只能针对接口做代理。我们有

更强大的代理cglib。

创建动态代理类对象的(Proxy)方法：

public static [Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) newProxyInstance([ClassLoader](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/ClassLoader.html) loader, [Class](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Class.html)<?>[] interfaces,

InvocationHandler h)：该方法最终会调用参数InvocationHandler接口的实现类中的方法。

loader - 定义了由哪个ClassLoader对象来对生成的代理对象进行加载。

interfaces - 表示的是我将要给需要代理的对象提供一组什么接口，如果我提供了

一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这

组接口中的方法了。

h - 为实现了InvocationHandler接口的类对象

该方法的返回值为一个代理对象。

InvocationHandler接口(JDK1.3出现)中的方法：

[Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) invoke([Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html) proxy,[Method](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/reflect/Method.html) method,[Object](mk:@MSITStore:C:\Users\Lijun\Desktop\JDK%201.6%20API.chm::/java/lang/Object.html)[] args)：在代理实例上处理方法调

用并返回结果。

proxy - 代理对象

method - 方法类对象

args - 方法类对象的实参

该方法的返回值为调用的method对象的invoke()的返回值。

## 2.动态代理的代码示例

(1)public class Test{

public static void main(String[] args){

StudentDao dao = new StudentDaoImpl();

MyInvocationHandler handler = new MyInvocationHandler(ud);

StudentDao proxy = (StudentDao) Proxy.newProxyInstance(dao.getClass().getClassLoader(),dao.getClass().getInterfaces(),handler);

proxy.login();

proxy.regist();

}

}

(2)public class MyInvocationHandler implements InvocationHandler{

private Object targer;//目标对象

public MyInvocationHandler(Object target){

this.target = target;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwanle{

System.out.println(“方法执行前执行”);

Object result = method.invoke(target, args);

System.out.println(“方法执行后执行”);

return result;

}

}

(3)public Interface StudentDao{

void login();

void regist();

}

(4)public class StudentDaoImpl implements StudentDao{

@Override

public void login(){

System.out.println(“登录功能”);

}

@Override

public void regist(){

System.out.println(“注册功能”);

}

}

## 3.动态代理的本质

通过反射来执行方法，在执行前和执行后，添加需要处理的代码。

# 枚举

## 1.概述

是指将变量的值一一列出来，变量的值只限于列举出来的值的范围内。举例：一周

只有7天，一年只有12个月等。

## 2.三版自己定义的枚举类

(1)public class Direction{

//创建几个实例

public static final Direction FRONT = new Direction();

public static final Direction BEHIND = new Direction();

public static final Direction LEFT = new Direction();

public static final Direction RIGHT = new Direction();

//构造私有

private Direction(){}

}

(2)public class Direction2{

//创建几个实例

public static final Direction2 FRONT = new Direction2(“前”);

public static final Direction2 BEHIND = new Direction2(“后”);

public static final Direction2 LEFT = new Direction2(“左”);

public static final Direction2 RIGHT = new Direction2(“右”);

//加入成员变量

private String name;

//构造私有

private Direction2(String name){this.name = name;}

public String getName(){

return name;

}

}

(3)public abstract class Direction3{

//创建几个实例

public static final Direction3 FRONT = new Direction3(“前”){

@Override

public void show(){

System.out.println(“前”);

}

};

public static final Direction3 BEHIND = new Direction3(“后”){

@Override

public void show(){

System.out.println(“后”);

}

};

public static final Direction3 LEFT = new Direction3(“左”){

@Override

public void show(){

System.out.println(“左”);

}

};

public static final Direction3 RIGHT = new Direction3(“右”){

@Override

public void show(){

System.out.println(“右”);

}

};

//加入成员变量

private String name;

//构造私有

private Direction3(String name){this.name = name;}

public String getName(){

return name;

}

//加入抽象方法

public abstract void show();

}

## 3.enum

说明：java提供了枚举类供我们使用。

格式：只有枚举项的枚举类

public enum 枚举类名{

枚举项1,枚举项2,枚举项3…;

}

三版枚举类：

(1)public enum Direction{

FRONT, BEHIND, LEFT, RIGHT;

}

从编译后的.class文件可以看到，该类重写了toString()，返回枚举常量的名称。

(2) public enum Direction2{

//创建几个实例

FRONT(“前”), BEHIND(“后”), LEFT(“左”), RIGHT(“右”);

//加入成员变量

private String name;

//构造私有

private Direction2(String name){this.name = name;}

public String getName(){

return name;

}

}

(3) public enum Direction3{

//创建几个实例

FRONT (“前”){

@Override

public void show(){

System.out.println(“前”);

}

},

BEHIND (“后”){

@Override

public void show(){

System.out.println(“后”);

}

},

LEFT (“左”){

@Override

public void show(){

System.out.println(“左”);

}

},

RIGHT (“右”){

@Override

public void show(){

System.out.println(“右”);

}

};

//加入成员变量

private String name;

//构造私有

private Direction3(String name){this.name = name;}

public String getName(){

return name;

}

//加入抽象方法

public abstract void show();

}

## 4.注意事项

(1)定义枚举类要用关键字enum

(2)所有枚举类都是Enum的子类

(3)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堂略的，但是如果枚

举类有其他的代码，这个分号就不能省略。建议不要省略。

(4)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：枚举项名("")。参考上面的第二版。

(5)枚举类也可以有抽象方法，但是枚举项必须重写该方法。

5.enum中的方法

public final int compareTo([E](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Enum.html) o)：比较此枚举与指定对象的顺序。在该对象小于、等于或

大于指定对象时，分别返回负整数、零或正整数。 枚举常量只能与相同枚举类型

的其他枚举常量进行比较。该方法实现的自然顺序就是声明常量的顺序。

public final [String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name()：返回此枚举常量的名称，在其枚举声明中对其进行声明。 与

此方法相比，大多数程序员应该优先考虑使用 [toString()](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Enum.html#toString()) 方法，因为 toString 方法

返回更加用户友好的名称。

public final int ordinal()：返回枚举常量的序数（它在枚举声明中的位置，其中初始常量

序数为零）。 大多数程序员不会使用此方法。它被设计用于复杂的基于枚举的数据

结构，比如 [EnumSet](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/EnumSet.html) 和 [EnumMap](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/util/EnumMap.html)。

public [String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) toString()：返回枚举常量的名称，它包含在声明中。可以重写此方法，虽

然一般来说没有必要。当存在更加“程序员友好的”字符串形式时，应该使用枚举类

型重写此方法。

public static <T extends [Enum](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Enum.html)<T>> T valueOf([Class](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/Class.html)<T> enumType, [String](mk:@MSITStore:E:\个人资料\技术文档\JDK%201.6%20API.chm::/java/lang/String.html) name)：返回带指

定名称的指定枚举类型的枚举常量。名称必须与在此类型中声明枚举常量所用的标

识符完全匹配。（不允许使用额外的空白字符。）

举例：

Direction d = Enum.valueOf(Direction.class,”FRONT”);

System.out.println(d.getName());

说明：从Direction这个类中拿枚举项名叫做FRONT的枚举项。

values()：此方法在文档中查不到，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。

举例：

Direction[] dirs = Direction.values();

for(Direction d : dirs){

System.out.println(d);

}

# JVM

## 1.JVM定义

JVM是一种用于计算设备的规范，它是一个虚拟出来的计算机，是通过在实际的计算机上仿

真模拟计算机的各个功能来实现的。Java语言的一个非常重要的特点就是与平台的无关

性，而使用Java虚拟机是实现这一特点的关键。

JVM分为三种：

(1)SUN公司的HotSpot(JDK自带的虚拟机)。

(2)BEA公司的JRockit，最快的JVM，适合财务前端办公、军事指挥与控制和电信网络的

需要。

(3)IBM公司的J9。

下面的JVM介绍都以HotSpot为准。

## 2.JVM的生命周期

(1)启动。启动一个java程序的时候就产生了一个JVM实例。

(2)运行。Main方法是程序的入口，任何其他线程均由它启动。

(3)消亡。当程序中所有非守护线程都终止时，JVM才退出。若安全管理器允许，程序也可

以使用Runtime类或者System.exit()来退出JVM。

## 3.JVM的组成

JVM由类加载器、字节码执行引擎、运行时数据区(堆、栈、本地方法栈、方法区、程序计

数器)组成。

在运行时数据区里，栈、本地方法栈、程序计数器是不会有垃圾的产生的，垃圾主要是在堆

里，方法区里很少，这意味着JVM调优主要是调堆。下图是JVM的内存模型图：



## 4.具体解释

(1)堆(Heap)

被所有线程共享的一块内存区域，在虚拟机启动时创建；

用来存储对象实例；

可以通过-Xms和-Xmx控制堆空间的大小；

OutOfMemory：当堆中没有完成实例分配，且堆再也无法扩展时；

JVM堆是垃圾收集器管理的主要区域；

堆空间又细分为：

新生代(Eden/Young)。新建的对象都由新生代分配内存。新生代常常又被划分为Eden

和Survivor区(包含s0和s1)，Eden空间不足时会把存活的对象转移到Survivor。新

生代的大小可以使用-Xmn控制，也可用-XX:SurvivorRatio控制Eden和Survivor的

比例。新生代占堆内存的1/3。

年老代(Tenured/Old)。存放经过多次(默认15次)垃圾回收依然存活的对象。年老代占堆

内存的2/3。

持久代(Permanent)，是方法区概念的一种实现。存放静态文件，如Java类、方法等。

持久代对垃圾回收没有显著影响。jdk1.8之后改名为元空间(Metaspace)，元空间没

有使用堆内存，而是与堆不相连的本地内存，理论上，本地内存多大，元空间就有

多大。

(2)方法区(Method Area)

线程间共享；

jdk1.7之前，方法区位于堆中的持久代。1.8后，方法区由元空间实现，即方法区位于元空

间。

用于存储已被虚拟机加载的类信息(构造方法、接口定义)、常量、静态变量、即时编译后的

代码等数据等；

OutOfMemoryError：当方法区无法满足内存分配时；

运行时常量池是方法区的一部分，用于编译期生成的各种字面量与符号引用，如String类型

的引用就存放在字符串常量池；

OutOfMemoryError：当常量池无法在申请到内存时；

(3)栈(Stack)

线程私有，生命周期与线程相同；

存储方法的局部变量表（基本类型、对象引用）、操作数栈、动态链接、方法出口等信息；

Java方法执行的内存模型，每个方法执行的同时都会创建一个栈帧，每个方法被调用直至完

成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程；

栈不存在垃圾回收的问题；

StackOverflowError：当线程请求的栈深度大于虚拟机所允许的深度；

OutOFMemoryError：如果栈扩展时无法申请到足够的内存；

JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈，JVM栈中存放的为当前线程

中局部基本类型的变量、部分的返回结果以及Stack Frame。其他引用类型的对象在JVM

栈上仅存放变量名和指向堆上对象实例的首地址。

(4)本地方法栈(Native Method Stack)

与虚拟机栈相似，主要是为虚拟机使用到的本地方法服务，用于登记native方法，在执行引

擎(Execution Engine)执行的时候加载本地库(Native Libraies)。在HotSpot虚拟机中和虚拟

机栈合二为一。

(5)程序计数器(Programma Counter Register)

当前线程所执行的字节码的行号指示器；

当前线程私有；

不会出现OutOfMemoryError；

占内存非常小，几乎可以忽略不计。

## 5.双亲委派机制

(1)四种类加载器

Bootstrap ClassLoader(启动类加载器)：主要负责加载Java核心类库，%JRE\_HOME%\lib下的

rt.jar、resources.jar、charsets.jar和class文件等。注意：该加载器是由C/C++语言写的，

所以用getClassLoader()是拿不到这个加载器的，会返回null。

Extention ClassLoader(扩展类加载器)：主要负责加载目录%JRE\_HOME%\lib\ext目录下的jar

包和class文件；

Application ClassLoader(应用程序类加载器)：主要负责加载当前应用的classpath(类路径)下的

所有类；

User ClassLoader(用户自定义类加载器)：可加载指定路径的class文件。

(2)双亲委派机制的定义

当一个类加载器收到了类加载的请求的时候，他不会直接去加载指定的类，而是把这个请求

委托给自己的父加载器去加载。只有父加载器无法加载这个类的时候，才会由当前这个

加载器来负责类的加载。

其委托的顺序为：User ClassLoader --> Application ClassLoader --> Extention ClassLoader -->

Bootstrap ClassLoader。也就是说，即便是需要User ClassLoader加载的类，也会一层层

的委托到Bootstrap ClassLoader，然后判断Bootstrap ClassLoader无法加载，再判断

Extention ClassLoader无法加载、再判断Application ClassLoader无法加载，最后再判断

User ClassLoader是否可以加载。

双亲委派机制是在ClassLoader里的loadClass方法里实现的。

(3)双亲委派机制的意义

通过委派的方式，可以避免类的重复加载，当父加载器已经加载过某一个类时，子加载器就

不会再重新加载这个类；

通过双亲委派的方式，还保证了安全性。比如：可以避免有人自定义一个有破坏功能的

java.lang.Integer被加载。因为jdk本身已经有java.lang包下的Integer类，该类由

Bootstrap ClassLoader加载，所以其他的加载器无法再加载这个包名和类名都相同的类。

(4)非双亲委派机制的地方

双亲委派机制有他存在的意义，不过也存在许多场景是需要破坏这个机制的，所以双亲委派

机制也非必然。比如 tomcat web容器里面部署了很多的应用程序，但是这些应用程序

对于第三方类库的依赖版本却不一样，但这些第三方类库的路径又是一样的，如果采用

默认的双亲委派类加载机制，那么是无法加载多个相同的类。所以，Tomcat破坏双亲

委派原则，提供隔离的机制，为每个web容器单独提供一个WebAppClassLoader加载

器。

Tomcat的类加载机制：为了实现隔离性，优先加载 Web 应用自己定义的类，所以没有遵

照双亲委派的约定，每一个应用自己的类加载器——WebAppClassLoader负责加载本身

的目录下的class文件，加载不到时再交给CommonClassLoader加载，这和双亲委派刚

好相反。

## 6.沙箱安全机制

(1)定义

沙箱是一个限制程序运行的环境(沙箱主要限制系统资源的访问，如cpu，内存等等。不同级

别的沙箱对这些资源的访问限制也不一样)。

(2)作用

就是将java代码限定在虚拟机(jvm)特定的运行范围中，并且严格限制代码对本地系统资源

的访问，通过这样的措施来保证对代码的有效隔离，防止对系统造成破坏。

(3)发展

jdk1.0安全模型：

在java中将程序分为本地代码和远程代码两种，本地代码默认是可信任的，而远程代码则

被看作是不可信任的。对于授信的本地代码可以访问一切本地资源，而对于非授信的远

程代码在早期java实现中，安全依赖于沙箱机制。



缺点：如此严格的机制为程序的功能扩展带来了障碍，比如当用户希望远程代码访问本地系

统文件时，就无法实现。

jdk1.1安全模型：

增加了安全策略，允许用户指定远程代码对资源的访问权限。



jdk1.2安全模型：

再次改进了安全机制，增加了代码签名。无论是本地代码还是远程代码，都会按照策略设定，

由类加载器加载到不同的空间，来实现差异化的代码执行权限控制。



jdk1.6安全模型(当前最新安全机制)：

引入了域的概念，虚拟机会把所有代码加载到不同的系统域和应用域，系统域部分专门负责

与关键资源进行交互，各个应用域部分则通过系统域代理来对各种需要的资源进行访问。

虚拟机中不同的受保护域对应不一样的权限。存在于不同域的类文件就具有了当前域的

全部权限。



(4)沙箱的基本组件

字节码校验器：确保Java类文件遵循java语言规范。这样可以帮助Java实现内存保护。但

并不是所有类文件都会经过字节码校验，比如核心类。

类装载器：在三个方面对沙箱起作用。

1.防止恶意代码去干涉善意代码(双亲委派机制)；

2.守护了被信任的类库边界；

3.它将代码归入保护域，确定了代码可以执行哪些操作。

## 7.native关键字

在Java代码中，使用native关键字表示要调用其他语言写成的方法。在Thread类中有类似

的例子，比如：private native void start0(); 即调用C/C++的方法start0来启动线程。

使用native会进入本地方法栈，调用本地方法接口(JNI)，JNI的出现，就是为了扩展Java的

使用，融合不同的编程语言。

## 8.JVM的内存介绍

(1)查询虚拟机可用的堆的总内存和初始化时的内存

public void testMemory(){  
 //虚拟机的堆的可用的最大内存(单位是字节)，约为实际物理内存的1/4。  
 long max = Runtime.*getRuntime*().maxMemory();  
 //虚拟机的堆的初始化的总内存(单位是字节)，约为实际物理内存的1/64。  
 long total = Runtime.*getRuntime*().totalMemory();  
 System.*out*.println("最大内存："+(max/(double)1024/1024)+"MB");  
 System.*out*.println("初始化总内存："+(total/(double)1024/1024)+"MB");  
}

(2)在idea里使用参数修改虚拟机堆的最大内存和初始化内存





参数：-Xms1024m -Xmx1024m -XX:+PrintGCDetails。其中-Xms1024m表示堆的初始内存为

1024M；-Xmx1024m表示堆的最大内存为1024M；-XX:+PrintGCDetails表示打印GC详细

信息。

(3)堆和非堆

堆：

初始分配的内存由-Xms指定，默认是物理内存的1/64；

最大分配的内存由-Xmx指定，默认是物理内存的1/4；

-Xmn2G：设置新生代大小为2G；

-XX:SurvivorRatio，设置新生代中Eden区与Survivor区的比值；

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于

70%时，JVM会减少堆直到 -Xms的最小限制。因此服务器一般设置-Xms、-Xmx相

等以避免在每次GC 后调整堆的大小。对象的堆内存由称为垃圾回收器的自动内存

管理系统回收。

非堆：

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；

由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4；

## 9. JProfiler

(1)介绍

JProfiler是由ej-technologies GmbH公司开发的一款性能瓶颈分析工具，用于分析JVM内存

和其中的存储数据。

(2)特点

使用方便；界面操作友好；对被分析的应用影响小；CPU,Thread,Memory分析功能尤其强大；

支持对jdbc,noSql, jsp, servlet, socket等进行分析；支持多种模式(离线，在线)的分析；

跨平台(Windows、Mac OS X、Linux、FreeBSD等)。

(3)使用方式

a.idea安装Jprofiler插件，然后重启idea

b.下载Jprofiler客户端并安装，官方网址：

https://www.ej-technologies.com/download/jprofiler/version\_111

c.使用license key生成器生成一个license key用于破解Jprofiler客户端

d.在idea上配置上已经安装好的Jprofiler客户端。图解：



e.写一个用于测试内存溢出错误的死循环用来测试，示例：

@org.junit.Test  
public void testOutOfMemory(){  
 ArrayList list = new ArrayList();  
 byte[] array = new byte[1\*1024\*1024];//1MB  
 int count = 0;  
 //用死循环来模拟一个OutOfMemoryError   
 while(true){  
 list.add(new Test());  
 count = count+1;  
 }  
}

f.修改这个方法的堆的大小，同时设置当出现OutOfMemoryError时，Dump出一个文件。



参数：-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError。其中-Xms1m是设置堆的初始

化容量为1MB，-Xmx8m是设置堆的最大容量为8MB，改小堆的容量只是测试方便，正

式开发时，不要改这么小。-XX:+HeapDumpOnOutOfMemoryError是设置当出现堆的

OutOfMemoryError时，Dump出一个文件。

g.运行这个方法，报OutOfMemoryError，同时查看Dump出的一个文件，该文件在项目下。



h.双击这个文件，用Jprofiler打开。



i.用Jprofiler打开此文件之后，会在项目下产生一些关联文件，不要忘了最后删除这些文件。



## 10.GC(Garbage Collection，即垃圾回收)

(1)主要是回收堆中新生代的Eden区和年老代(Old)区的内存空间。

(2)当新生代的Eden区满了的时候，会触发一次GC的轻量级回收(Minor GC)，因为新生代的

Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般在这里使用速度快、效率高

的算法，使垃圾回收能尽快完成。轻量级回收之后，会将幸存的对象转移到新生代的

Survivor区。Minor GC会引发STW，暂停其他用户线程，等垃圾回收结束，用户线程才

恢复运行

(3)当年老代满了的时候，会触发一次major GC。major GC速度比Minor GC慢10倍，STW

的时间更长。

(3)Full GC (重量级回收)，他会对整个堆进行回收，包括新生代、年老代和持久代，所

以速度较慢，因此应该尽量减少Full GC的次数。导致Full GC的原因包括：新生代被写

满、年老代被写满、永久代被写满和System.gc()被显式调用等。major GC和full GC通

常是等价的，收集整个GC堆。

(4)引用计数法(了解，已经不再使用)

介绍：属于GC算法的一种。假设A对象用了一次就给它的计数器为1，假设对象B用了两

次给它的计数器为2，假设对象C一次也没有用它的计数器为0，那么对象C就会被垃

圾回收进行清除。

缺点：比较难处理循环引用。又因为每个对象都分配一个计数器，比较消耗内存。

(5)复制回收算法(GC 90%用的都是这个，这个算法一般用在新生代)

介绍：属于GC算法的一种。新生代有Survivor区，在Survivor区里有s0和s1区，其实s0

和s1正确来说是from区和to区。在第一次Minor GC后，Eden区里幸存的对象会复制

到from区里，Eden区会被清空。第二次Minor GC后，Eden区里幸存的对象和from区

中的对象会复制到to区，这时，from区会被置空，转为to区，原来的to区有了对象，

转为from区。以后的Minor GC都会循环第二次的模式。图解：



遵循规则：谁空谁是to区。

优点：不会产生内存碎片，效率高。

缺点：from区和to区容量相同，但是其中一个总是空的，浪费内存空间。

补充：默认经过15次垃圾回收仍存活的对象会进入年老代，这个次数也是可调的。参数：

-XX:MaxTenuringThreshold=20，设置次数为20。

(6)标记-清除算法(适用于年老代)

介绍：属于GC算法的一种。如它的名字一样，算法分为“标记”和“清除”两个阶段：首

先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。之所以说

它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进

而得到的。图解：



优点：不需要额外的内存空间

缺点：两次扫描，浪费时间，且清除对象数据时会产生大量不连续的内存碎片，可能会导致

大对象分配不到连续的内存空间，触发GC。

(7)标记-整理算法(适用于年老代)

介绍：属于GC算法的一种，是标记-清除算法的再优化方案。步骤是：先标记需要回收的对

象，然后将所有存活的对象都向一端移动，最后直接回收掉端边界以外的内存。图解：



优点：相比标记-清除算法不会产生内存碎片。

缺点：相比标记-清除算法，增加了移动对象数据的资源消耗。

(8)算法总结

对于不同生命周期的对象，JVM把Java堆分为了新生代和年老代，由此，GC总体采用的是

分代收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，

那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中

因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标

记-整理”算法来进行回收。

# JUC

## 1.含义

JUC指的是JAVA自带的并发包，java.util.concurrent的首字母集合。



并发编程的本质：充分利用CPU的资源。

补充：查询CPU的逻辑核心(线程)数：Runtime.getRuntime().availableProcessors()。

## 2.wait和sleep方法的区别

(1)wait定义在Object类中，sleep定义在Thread类中。一旦一个对象调用了wait方法，必

须要采用notify()和notifyAll()方法唤醒该进程。

补充：TimeUnit.DAYS.sleep(1);//睡眠一天

TimeUnit.SECONDS.sleep(1);//睡眠1秒

类似的更多方法可以查看TimeUnit类。

(2)wait会释放锁，sleep不会释放锁。

(3)wait只能在synchronized的同步方法或是代码块中使用，而sleep可以在任何地方使用。

(4)wait不需要捕获异常，sleep必须要捕获异常。

## 3.线程的状态

说明：线程在源码中定义了几个状态，位置在Thread类的内部枚举类State中。

状态分别为NEW(新建)、RUNNABLE(运行)、BLOCKED(阻塞)、WAITING(等待)、

TIMED\_WAITING(超时等待)、TERMINATED(终止)。

## 4.并发下的List

说明：首先List在并发环境下是不安全的，在并发修改时会报ConcurrentModificationException，

即并发修改异常。示例：

List<String> list = new ArrayList<>();  
for (int i = 1; i <= 30; i++) {  
 new Thread(() -> {  
 list.add(UUID.*randomUUID*().toString().substring(0, 5));  
 System.*out*.println(list);  
 }, String.*valueOf*(i)).start();  
}

解决方式：

(1)使用Vector，Vector是线程安全的。示例：

List<String> list = new Vector<>();

注意：此种方式效率很低，已经淘汰。

(2)使用Collections类的方法将ArrayList转成线程安全的。示例：

List<String> list = Collections.synchronizedList(new ArrayList<>());

补充：Collections类也有将Set、Map等转成线程安全的方法，查JDK文档。

(3)使用java.util.Concurrent包下的并发类。

替代ArrayList的是CopyOnWriteArrayList。示例：

List<String> list = new CopyOnWriteArrayList();

补充：CopyOnWrite意为写入时复制，COW是计算机程序设计领域的一种优化策略，适合读多写少的并发场景。其核心思想是，如果有多个调用者（Callers）同时要求相同的资源（如内存或者是磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的（transparently）。此做法主要的优点是如果调用者没有修改资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。缺点是内存占用会增加很多，而且此种方式只能保证数据的最终一致性，不能保证数据的实时一致性。

补充：CopyOnWriteArrayList为什么并发安全且性能比Vector好？

我们都知道Vector是增删改查方法都加了synchronized，保证同步，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而CopyOnWriteArrayList 只是在增删改上加锁，但是读不加锁，在读方面的性能就好于Vector，CopyOnWriteArrayList支持读多写少的并发情况。

## 5.并发下的Set

说明：首先Set在并发环境下也不安全，在并发修改时会报ConcurrentModificationException，

即并发修改异常。示例和List差不多。

解决方式：

(1)使用Collections类的方法将HashSet转成线程安全的。示例：

Set<String> set = Collections. synchronizedSet(new HashSet<>());

(2)使用java.util.Concurrent包下的并发类。

替代HashSet的是CopyOnWriteArraySet。示例：

Set<String> set = new CopyOnWriteArraySet ();

## 6.并发下的Map

说明：Map在并发修改时也会报ConcurrentModificationException。

解决方式：

(1)使用Collections类的方法将HashMap转成线程安全的。示例：

Map map = Collectios. synchronizedMap(new HashMap());

(2)使用java.util.Concurrent包下的并发类。

替代HashMap的是ConcurrentHashMap。示例：

Map map = new ConcurrentHashMap();

补充：ConcurrentHashMap的key和value不能是null，否则报异常。

## 7.并发辅助类CountDownLatch

说明：计数减一的类，用于等待线程执行完毕时，再执行接下来代码的辅助类。

示例：

CountDownLatch latch = new CountDownLatch(6);//参数必须是线程执行的次数，这里是6  
for(int i=1;i<=6;i++){  
 new Thread(()->{  
 System.*out*.println(Thread.*currentThread*().getName()+"Go out");  
 latch.countDown();//计数减一  
 },String.*valueOf*(i)).start();  
}  
latch.await();//等待计数归零，然后再向下执行  
System.*out*.println("Close Door");

## 8.并发辅助类CyclicBarrier

说明：计数加一的类，用于等待线程执行完毕时，再执行接下来代码的辅助类。

构造方法：

public CyclicBarrier(int parties)：设置等待线程执行parties次

public CyclicBarrier(int parties, Runnable barrierAction)：设置等待线程执行parties次，然

后执行线程barrierAction

示例：

CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{  
 System.*out*.println("召唤神龙");  
});  
for(int i=1;i<=7;i++){  
 final int temp = i;  
 new Thread(()->{  
 System.*out*.println(Thread.*currentThread*().getName()+"收集"+temp+"个龙珠");  
 try{  
 cyclicBarrier.await();//计数加一并等待  
 }catch(InterruptedException e){  
 e.printStackTrace();  
 }catch(BrokenBarrierException e){  
 e.printStackTrace();  
 }  
 }).start();  
}

补充：如果创建CyclicBarrier对象时，参数parties大于线程的执行次数，则CyclicBarrier对

象会一直等待。

## 9.并发辅助类Semaphore

说明：用于线程数量限流的的辅助类。

示例：

//设置最大的线程运行数   
Semaphore semaphore = new Semaphore(3);  
for(int i=1;i<=6;i++){  
 new Thread(()->{  
 try{  
 semaphore.acquire();//获取运行机会  
 System.*out*.println(Thread.*currentThread*().getName()+"抢到车位");  
 TimeUnit.*SECONDS*.sleep(2);  
 System.*out*.println(Thread.*currentThread*().getName()+"离开车位");  
 }catch(InterruptedException e){  
 e.printStackTrace();  
 }finally{  
 semaphore.release();//释放运行的机会  
 }  
 },String.*valueOf*(i)).start();  
}

## 10.ReadWriteLock

说明：ReadWriteLock是一个读写锁(接口)，对于锁的分类更具体，其实现类

ReentrantReadWriteLock一般最常用。存在的意义是将锁分为读锁和写锁，读锁同时能

多个线程访问，写锁同时只能一个线程访问。

ReentrantReadWriteLock的方法：

public ReentrantReadWriteLock.ReadLock readLock()：返回一个读锁

public ReentrantReadWriteLock.WriteLock writeLock()：返回一个写锁

示例：

public class ReadWriteLockDemo {  
 public static void main(String[] args) {  
 MyCacheLock myCacheLock = new MyCacheLock();  
 //测试写锁  
 for(int i=1;i<=5;i++){  
 final int temp = i;  
 new Thread(()->{  
 myCacheLock.put(temp+"",temp+"");  
 },String.*valueOf*(i)).start();  
 }  
 //测试读锁  
 for(int i=1;i<=10;i++){  
 final int temp = i;  
 new Thread(()->{  
 myCacheLock.get(temp+"");  
 },String.*valueOf*(i)).start();  
 }  
 }  
}  
class MyCacheLock{  
 private volatile Map<String,Object> map = new HashMap<>();  
 //读写锁  
 private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();  
 //写入时，只支持同时只有一个线程写入  
 public void put(String key, Object value){  
 readWriteLock.writeLock().lock();  
 try{  
 System.*out*.println(Thread.*currentThread*().getName()+"写入开始");  
 map.put(key,value);  
 System.*out*.println(Thread.*currentThread*().getName()+"写入完毕");  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 readWriteLock.writeLock().unlock();  
 }  
  
 }  
 //读取时，支持同时多个线程读取  
 public void get(String key){  
 readWriteLock.readLock().lock();  
 try{  
 System.*out*.println(Thread.*currentThread*().getName()+"读取开始");  
 Object o = map.get(key);  
 System.*out*.println(Thread.*currentThread*().getName()+"读取完毕");  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally{  
 readWriteLock.readLock().unlock();  
 }  
 }  
}

## 11.阻塞队列BlockingQueue(接口)

说明：队列是一种数据结构，遵循先进先出的原则。BlockingQueue常用的实现类有

ArrayBlockingQueue和LinkedBlockingQueue。

补充：阻塞队列何时会阻塞呢？两种情况，一种队列中元素已满，无法再插入元素时阻塞；

一种队列中无元素，获取队列中的元素时阻塞。

BlockingQueue的四组API：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方式 | 抛出异常 | 不抛出异常 | 阻塞 等待 | 超时 退出 |
| 添加元素并返回添加是否成功 | add(E e) | offer(E e) | put(E e) | offer(E e, long timeout, TimeUnit unit) |
| 移除并返回移除的元素 | remove() | poll() | take() | poll(long timeout, TimeUnit unit) |
| 获取队列首元素 | element() | peek() | - | - |

示例：

//设置队列的容量  
ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);  
  
System.*out*.println(blockingQueue.add("a"));  
System.*out*.println(blockingQueue.add("b"));  
System.*out*.println(blockingQueue.add("c"));  
//add方法在添加元素超过队列的容量时会抛出异常-java.lang.IllegalStateException: Queue full  
//System.out.println(blockingQueue.add("d"));  
  
System.*out*.println(blockingQueue.remove());  
System.*out*.println(blockingQueue.remove());  
System.*out*.println(blockingQueue.remove());  
//remove方法在队列中已无元素时会抛出异常-java.util.NoSuchElementException  
//System.out.println(blockingQueue.remove());

补充：

(1)offer(E e)在添加元素超过队列的容量时会返回false。

(2)poll()在队列中已无元素时会返回null。

(3)element()在队列中已无元素时会报java.util.NoSuchElementException。

(4)peek()在队列中已无元素时则会返回null。

## 12. SynchronousQueue<E>(特殊的阻塞队列-同步队列)

说明：SynchronousQueue<E>实现了BlockingQueue，是一种特殊的阻塞队列，其本身是没有

容量大小的，进去一个元素，必须等待取出之后，才能再往里面放一个元素。

SynchronousQueue在消息队列技术中间件中被大量使用。

示例：

BlockingQueue<String> blockingQueue = new SynchronousQueue<>();  
new Thread(()->{  
 try{  
 System.*out*.println(Thread.*currentThread*().getName()+" put 1");  
 blockingQueue.put("1");  
 System.*out*.println(Thread.*currentThread*().getName()+" put 2");  
 blockingQueue.put("2");  
 System.*out*.println(Thread.*currentThread*().getName()+" put 3");  
 blockingQueue.put("3");  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
},"T1").start();  
  
new Thread(()->{  
 try{  
 TimeUnit.*SECONDS*.sleep(3);  
 System.*out*.println(Thread.*currentThread*().getName()+" get "+blockingQueue.take());  
 TimeUnit.*SECONDS*.sleep(3);  
 System.*out*.println(Thread.*currentThread*().getName()+" get "+blockingQueue.take());  
 TimeUnit.*SECONDS*.sleep(3);  
 System.*out*.println(Thread.*currentThread*().getName()+" get "+blockingQueue.take());  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
},"T2").start();

补充：put(E e)和take()都是在运行过程中就阻塞了。比如：put(E e)运行之后，它下面的代码

不会执行，只有等到take()执行之后，原来put(E e)下的代码才会继续执行。

## 13.ForkJoin

说明：Fork/Join 框架是 Java7 提供的一个用于并行执行任务的框架， 是一个把大任务分割

成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。类似于Java 8中

的parallel Stream。注意：【只能将任务1个切分为两个，不能切分为3个或其他数量】

相关类说明：

ForkJoinTask : 基本任务，使用forkJoin框架必须创建的对象，提供forkJoin操作，其常

用的三个子类如下：

(1) RecursiveAction-无结果返回的任务

(2) RecursiveTask-有返回结果的任务

(3) CountedCompleter-无返回值任务，完成任务后可以触发回调

ForkJoinTask提供了两个重要的方法-fork(让task异步执行)、join(让task同步执行，

可以获取返回值)

ForkJoinPool：专门用来运行 ForkJoinTask 的线程池，（在实际使用中，也可以接收

Runnable/Callable 任务，但在真正运行时，也会把这些任务封装成 ForkJoinTask 类

型的任务）。他是 fork/join 框架的核心，是 ExecutorService 的一个实现，用于管

理工作线程，并提供了一些工具来帮助获取有关线程池状态和性能的信息。工作线

程一次只能执行一个任务。ForkJoinPool 线程池并不会为每个子任务创建一个单独

的线程，相反，池中的每个线程都有自己的双端队列用于存储任务(double-ended

queue)。这种架构使用了一种名为工作窃取(work-stealing)算法来平衡线程的工作负

载。

其执行方法：

invoke(ForkJoinTask)：提交任务并一直阻塞直到任务执行完成返回合并结果。

execute(ForkJoinTask)：异步执行任务，无返回值。

submit(ForkJoinTask)：异步执行任务，返回task本身，可以通过task.get()方法获取

合并之后的结果。

ForkJoinTask 在不显示使用 ForkJoinPool.execute/invoke/submit() 方法进行执行的情况

下，也可以使用自己的fork/invoke方法进行执行。

工作窃取(work-stealing)算法：简单来说，就是空闲的线程试图从繁忙线程的 deques 中 窃

取工作。默认情况下，每个工作线程从其自己的双端队列中获取任务。但如果自己的双

端队列中的任务已经执行完毕，双端队列为空时，工作线程就会从另一个忙线程的双端

队列尾部或全局入口队列中获取任务，因为这是最大概率可能找到工作的地方。这种方

法最大限度地减少了线程竞争任务的可能性。它还减少了工作线程寻找任务的次数，因

为它首先在最大可用的工作块上工作。

详细介绍ForkJoin的CSDN网页：https://blog.csdn.net/qq\_41432730/article/details/121308729

简单示例：

public class ForkJoinDemo extends RecursiveTask<Long> {  
  
 private Long start;  
 private Long end;  
  
 private Long temp = 10000L;  
  
 public ForkJoinDemo(Long start, Long end){  
 this.start = start;  
 this.end = end;  
 }  
  
 @Override  
 protected Long compute() {  
 if(end-start < temp){  
 Long sum = 0L;  
 for(Long i=start;i<=end;i++){  
 sum+=i;  
 }  
 return sum;  
 }  
 Long middle = (start+end)/2;  
 ForkJoinDemo task1 = new ForkJoinDemo(start, middle);  
 task1.fork();//拆分任务，把任务压入线程队列  
 ForkJoinDemo task2 = new ForkJoinDemo(middle+1, end);  
 task2.fork();//拆分任务，把任务压入线程队列  
 return task1.join() + task2.join();  
 }  
  
 public static void main(String[] args) {  
 Long sum = 0L;  
 long start = System.*currentTimeMillis*();  
  
 //1.直接for循环累加求值 10181毫秒  
 for(Long i=1L;i<=10\_0000\_0000;i++){  
 sum+=i;  
 }  
  
 //2.forkJoin 9422毫秒  
// ForkJoinPool forkJoinPool = new ForkJoinPool();  
// ForkJoinTask<Long> task = new ForkJoinDemo(0L, 10\_0000\_0000L);  
// ForkJoinTask<Long> submit = forkJoinPool.submit(task);  
// try {  
// sum = submit.get();  
// } catch (InterruptedException e) {  
// e.printStackTrace();  
// } catch (ExecutionException e) {  
// e.printStackTrace();  
// }  
  
 //3.并行流 最快 930毫秒  
// sum = LongStream.rangeClosed(0L,10\_0000\_0000L).parallel().reduce(0,Long::sum);  
  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("sum："+sum+"时间："+(end-start));  
 }  
}

## 14.Future(异步回调)

异步回调：指的是当一个系统发出请求并且需要等待另一个系统处理完成并返回结果后才能

继续后续操作时，采用异步的方式来进行回调通知，即不是同步等待返回结果，而是在

接收到返回结果后进行回调处理。这种方式能够有效提高系统的响应速度和可伸缩性，

适用于大量的请求场景。通俗易懂地举一个例子，比如您去餐馆点餐，等餐的时候您不

需要一直等待，可以先去逛逛街或者做其他事情，餐馆做好餐后会给您打电话或者发短

信通知您来取餐。这里的通知就是一个异步回调，您不需要一直等待餐馆做好餐，而是

在餐做好后再回来取餐。

详细介绍Future的CSDN网页<https://blog.csdn.net/honey_qin/article/details/116379489>

前景：Runnable接口是在Java多线程中表示线程的业务代码的抽象接口。但是Runnable没

有返回值，为了解决这个问题，Java定义了一个和Runnable类似的接口 — Callable接

口。并将业务处理方法名为call。Callable接口是一个范型接口，也声明为了函数式接

口。唯一的抽象方法call有返回值，返回值类型为范型形参的实际类型。

介绍：当使用Future模式时，可以让调用方立即返回，然后它自己会在后面慢慢处理，此

时调用者拿到的仅仅是一个凭证，调用者可以先去处理其它任务，在真正需要用到调用

结果的场合，再使用凭证去获取调用结果。这个凭证就是这里的Future。

说明：Future接口并不复杂，主要是对并发任务的执行及获取其结果的一些操作。主要有三

大功能。

(1) 判断并发任务是否执行完。

(2) 获取并发的任务完成后的结果

(3) 取消并发执行的任务

Future详细方法：

(1) V get()：获取并发任务执行的结果。这个方法是阻塞的，如果并发任务没有执行完成，

则调用此方法的线程会一直阻塞。

(2) V get(Long timeout, TimeUtil unit)：获取并发任务执行的结果。也是阻塞的，但是有

阻塞的时间限制，如果阻塞时间超过设定的时间，该方法将会抛出异常。

(3) boolean isDone()：获取并发任务的状态是否结束。

(4) boolean isCancelled()：获取并发任务的取消状态。如果任务被取消返回true。

(5) boolean cancel(boolean mayInterruptIfRunning)：取消并发任务的执行。

子类：

(1)FutureTask。在FutureTask类中，有一个Callable的私有成员，Futuretask内部有一个

run方法。这个run方法是Runable接口的抽象方法，在FutureTask类的内部提供了自己的实现。在Thread线程实例执行时，会将这个run方法作为target目标去异步执行。在FutureTask内部的run方法中实际是会执行Callable的call方法。执行完后结果会保存在私有成员 -- outcome属性中，outcome负责保存结果。然后FutureTask通过get方法获取这个object的值，那这个FutureTask的任务也就能成功完成了。

代码示例：

public class FutureTaskDemo {  
 public static void main(String[] args) {  
 FutureTask<Boolean> taskA = new FutureTask<>(()->{  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 System.*out*.println("A任务准备好了");  
 }catch (Exception e){  
 System.*out*.println("A任务出问题了");  
 return false;  
 }  
 System.*out*.println("A任务运行结束");  
 return true;  
 });  
  
 FutureTask<Boolean> taskB = new FutureTask<>(()->{  
 try {  
 TimeUnit.*SECONDS*.sleep(4);  
 System.*out*.println("B任务准备好了");  
 }catch (Exception e){  
 System.*out*.println("B任务出问题了");  
 return false;  
 }  
 System.*out*.println("B任务运行结束");  
 return true;  
 });  
  
 Thread threadA = new Thread(taskA);  
 Thread threadB = new Thread(taskB);  
  
 threadA.start();  
 threadB.start();  
  
 Thread.*currentThread*().setName("主线程");  
 try {  
 boolean a = taskA.get();  
 boolean b = taskB.get();  
 *isReady*(a,b);  
 }catch (Exception e){  
 System.*out*.println("发生了中断");  
 }  
 System.*out*.println("运行结束");  
 }  
  
 public static void isReady(boolean a,boolean b){  
 if(a && b){  
 System.*out*.println("都准备好了");  
 }else if (!a){  
 System.*out*.println("A没准备好");  
 }else{  
 System.*out*.println("B没准备好");  
 }  
 }  
}

补充：要是将上面的代码跑一下会发现，这里的FutureTask类的get方法，异步获取结果的同时，主线程是阻塞的。所以可以将其归为异步阻塞模式。异步阻塞的效率往往是比较低的，被阻塞的主线程不能干任何事。并没有实现非阻塞的异步结果获取方法。如果需要用到获取异步结果，则需要引入一些框架，比如：Google的Guava框架和Netty的异步回调。

## 15.JMM(Java Memory Model)

详细介绍JMM的CSDN网页<https://blog.csdn.net/zjcjava/article/details/78406330、>

https://blog.csdn.net/LYQ20010417/article/details/124138635

含义：Java内存模型，是一种约定，或者说概念。

相关内容：

(1) Java内存模型规定了所有的变量都存储在主内存(Main Memory)中。主内存可以类比

成物理硬件的主内存，但此处仅是虚拟机内存的部分。

(2) 每条线程还有自己的工作内存(Working Memory)。工作内存是一个理论上的概念，

通常是寄存器或者CPU的高速缓存充当工作内存。

(3) 线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝， 线程对变量

的所有操作(读取、赋值等)都必须在工作内存中进行，而不能直接读写主内存中的

变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传

递均需要通过主内存来完成。

约定的一些内容：

(1)线程解锁前，必须把工作内存中拷贝的共享变量立刻刷回主存

(2)线程加锁前，必须读取主存中的最新值到工作内存中

(3)加锁和解锁必须是同一把锁

JMM中的八种操作：为了支持 JMM，Java 定义了8种原子操作，用来控制主存与工作内存

之间的交互。

·read 读取：作用于主内存，将共享变量从主内存传送到线程的工作内存中。

·load 载入：作用于工作内存，把 read 读取的值放到工作内存中的副本变量中。

·store 存储：作用于工作内存，把工作内存中的变量传送到主内存中。

·write 写入：作用于主内存，把从工作内存中 store 传送过来的值写到主内存的变量

中。

·use 使用：作用于工作内存，把工作内存的值传递给执行引擎，当虚拟机遇到一个需

要使用这个变量的指令时，就会执行这个动作。

·assign 赋值：作用于工作内存，把执行引擎获取到的值赋值给工作内存中的变量，当

虚拟机栈遇到给变量赋值的指令时，就执行此操作。

·lock锁定： 作用于主内存，把变量标记为线程独占状态。

·unlock解锁： 作用于主内存，它将释放独占状态。

图解：



## 16.Volatile

Volatile的定义：java 编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致

的更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某

些情况下比锁更加方便。如果一个字段被声明成volatile，java线程内存模型确保所有

线程看到这个变量的值是一致的。

说明：Volatile是Java虚拟机提供的轻量级同步机制，可以理解为轻量级的synchronized。

volatile 是变量修饰符，其修饰的变量具有可见性。

Volatile特性内容：

(1)保证可见性

在上面的JMM中，我们知道线程可以操作主存中的共享变量，方式是通过拷贝共

享变量到自己的工作内存中。可见性也就是一旦某个线程修改了该共享变量的值且

刷回了主存，则其他线程读值时可以立即获取其修改之后的值。可见性的原理是使

用了MESI嗅探机制。

(2)不保证原子性

原子性是指一段不可分割的逻辑整体，当执行时，要么同时成功，要么同时失败。

而Volatile是无法保证原子性的。JMM的8种原子操作见上图。

(3)禁止指令重排

验证可见性的代码示例：

public class JMMDemo {  
 */\*\*  
 \* 此处如果不加volatile修饰num的话，程序不会停止，因为线程不知道主线程修改了  
 \* num的值，这个while会一直循环判断下去  
 \*/* private volatile static int *num* = 0;  
  
 public static void main(String[] args) {  
 new Thread(()->{  
 while(*num*==0){  
 */\*\*  
 \* 如果此处加了打印输出，则即使没有volatile修饰num，程序也会  
 \* 停止，因为println()方法中的语句有synchronized代码块保  
 \* 证可见性。  
 \*/* System.*out*.println("可见性");  
 }  
 }).start();  
 try{  
 //主线程睡眠1s  
 TimeUnit.*SECONDS*.sleep(1);  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 *num*=1;  
 System.*out*.println(*num*);  
 }  
}

原子性验证示例：

在java中，即使是num++这样的一条语句也不是线程安全的，因为它在字节码文件中是由

几步操作共同执行的。

验证步骤：

(1)创建一个类，执行main方法并得到该类的class文件，类的内容如下：

public class VDemo {  
 private volatile static int *num* = 0;  
 public static void add(){  
 *num*++;  
 }  
 public static void main(String[] args) {  
 *add*();  
 }  
}

(2)使用javap命令打开class字节码文件，看num++有几步原子操作组成。

命令示例：javap -c VDemo.class



含义解析：getstatic-获取值；iadd-值加1；putstatic-写回这个值；

JVM中 iconst 是一个入栈指令，其作用是用来将 int 类型的数字、取值在 -1 到 5 之间的

整数压入栈中。当取值等于 -1 时，采用 iconst\_m1 指令，当取值在 0 到 5 之间时，

分别对应 iconst\_0、iconst\_1、iconst\_2、iconst\_3、iconst\_4、iconst\_5 这几个指令。

在不使用Lock和synchronized的情况下，如何使这些操作保证原子性？

答：可以使用一系列的原子类，比如：AtomicBoolean、AtomicInteger、AtomicLong等，其位

置位于java.util.concurrent.atomic包下。每个原子类都自己的方法，这些方法都可以保

证原子性、可见性，且效率极高，其底层使用了CAS(compare and swap)，中文译为 比

较并交换。

实例：使用AtomicInteger中的自增方法

public class VDemo {  
 private volatile static AtomicInteger *num* = new AtomicInteger();  
 public static void add(){  
 //获取值并加一，该方法可保证原子性  
 *num*.getAndIncrement();  
 }  
 public static void main(String[] args) {  
 *add*();  
 }  
}

补充：volatile只能修饰实例变量和类变量，当修饰的变量为引用类型时，仅表示对象内存

地址的可见性，不保证内容的可见性。

被volatile关键字修饰的对象作为类变量或实例变量时，其对象中携带的类变量和实例

变量也相当于被volatile关键字修饰了。

## 17.指令重排

含义：你写的源代码，在不影响结果的前提下，计算机会调整代码的执行顺序，对代码进行

优化，以提高执行效率。这里的不影响结果只是考虑到数据之间的依赖性。

源代码->编译器优化重排->指令并行也可能重排->内存系统也会重排->执行

指令重排在单线程环境下不会有任何影响，但是在多线程环境下可能会有影响。

(1)关联变量

public class D {  
 static Integer *a*;  
 static Boolean *flag*;  
 public static void writer(){  
 *a* = 1;  
 *flag* =true;  
 }  
 public static void reader(){  
 if(*flag*!=null && *flag*){  
 System.*out*.println(*a*);  
 *a* = 0;  
 *flag* = false;  
 }  
 }  
}

说明：在单线程环境下，writer方法中的语句即使有指令重排也不会影响程序结果。但如果

在多线程环境下，writer方法中的语句发生了指令重排，也就是a=1;和flag=true;互换了

顺序，则控制台本该一直打印1，却会出现打印0的情况。这是因为，一个线程执行

flag=true;后发生了中断，而另一个线程在执行reader时判断flag=true就会打印a的值，

此时a可能为0。

(2)new创建对象

使用关键字new创建对象时，因其非原子操作，故存在指令重排，指令重排在多线程环境

下会带来负面影响。

public class Teacher {  
 private Teacher(){}  
 private static Teacher *t* = null;  
 public static Teacher getTeacher(){  
 //使用双重if判断的方式被称为DCL懒汉式  
 if(*t* == null){  
 synchronized (Teacher.class){  
 if(*t* == null){  
 *t* = new Teacher();  
 }  
 }  
 }  
 return *t*;  
 }  
}

说明：使用new关键字创建对象时有以下几个过程。

在栈空间创建引用地址->以类文件为模版在堆空间对象分配内存->成员变量初始化->使用构造函数初始化->将引用值赋值给左侧存储变量。因为存在指令重排，有可能执行的顺序是 在栈空间创建引用地址->将引用值赋值给左侧存储变量，此时CPU调度时间片耗尽出现线程中断，后续线程在检测t变量不为空时则直接使用，但因为t并未实例化完成，所以直接使用会导致意想不到的结果。

如何避免指令重排？

答：a.使用AtomicReference原子类，将一组相关联的变量封装成一个对象，利用原子操作

的特性，可以有效回避指令重排问题。实例：

@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class ValueModel {  
 private Integer value;  
 private Boolean flag;  
}

public class E {  
 private static final AtomicReference<ValueModel> *ar* = new AtomicReference<>(new ValueModel());  
 public static void writer(){  
 *ar*.set(new ValueModel(1,true));  
 }  
 public static void reader(){  
 ValueModel valueModel = *ar*.get();  
 if(valueModel.getFlag()!=null && valueModel.getFlag()){  
 System.*out*.println(valueModel.getValue());  
 *ar*.set(new ValueModel(0,false));  
 }  
 }  
}

b.使用volatile，其可以禁止编译器进行指令重排。原理是编译器会将非volatile的变量

缓存到CPU寄存器或其他高速缓存中以提高访问速度。而被volatile修饰的变量不会被

缓存，每次访问时都会从内存中读取，这样可以保证程序的正确性，避免指令重排的错

误。在CPU层面，读写volatile变量时，通常需要使用memory barrier(内存屏障)指令来

保证指令执行的顺序和正确性。实例：

public class Teacher {  
 private Teacher(){}  
 //用volatile修饰是为了在new Teacher()过程中禁止指令重排，从而避免发生创建对象过程

中的错误  
 private volatile static Teacher *t* = null;  
 public static Teacher getTeacher(){  
 //使用双重if判断的方式被称为DCL懒汉式  
 if(*t* == null){  
 synchronized (Teacher.class){  
 if(*t* == null){  
 *t* = new Teacher();  
 }  
 }  
 }  
 return *t*;  
 }  
}

## 18.CAS

含义：CAS，compare and swap的缩写，中文翻译成比较并交换。

说明：CAS 操作包含三个操作数 —— 内存位置(V)、预期原值(A)和新值(B)。如果内存位置

的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做

任何操作。CAS是java一系列原子类的底层原理。

通过原子类AtomicInteger的getAndIncrement()方法来查看CAS的使用。

代码示例：

public static void main(String[] args) {  
 AtomicInteger atomicInteger = new AtomicInteger(10);  
 //得到并+1，类似于i++  
 Integer value = atomicInteger.getAndIncrement();  
 System.*out*.println(value);//10  
 System.*out*.println(atomicInteger.get());//11  
}

// AtomicInteger类的getAndIncrement()代码：

补充：this是AtomicInteger类的对象。valueOffset是变量value相对于AtomicInteger对象的起始地址的偏移量。1是待会value要加的值。

public final int getAndIncrement() {  
 return *unsafe*.getAndAddInt(this, *valueOffset*, 1);  
}

//valueOffset的得到方式：

static {  
 try {  
 *valueOffset* = *unsafe*.objectFieldOffset  
 (AtomicInteger.class.getDeclaredField("value"));  
 } catch (Exception ex) { throw new Error(ex); }  
}

//Unsafe的getAndAddInt()方法

补充：var1是AtomicInteger类的对象，var2是valueOffset，var4是1。

this.getIntVolatile(var1, var2)是获取value的值。

this.compareAndSwapInt(var1, var2, var5, var5 + var4)是根据var1和var2拿到value的值，将

value和var5做比较，如果二者值相同，则将var5+var4的值在内存的相同位置将var5

替换掉。

这里展示的do()while{}是一个无限循环的自旋锁。

public final int getAndAddInt(Object var1, long var2, int var4) {  
 int var5;  
 do {  
 var5 = this.getIntVolatile(var1, var2);  
 } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));  
  
 return var5;  
}

CAS的缺点：

1.循环时间长开销大。高并发下N多线程同时去操作一个变量，会造成大量线程CAS失败，

然后处于自旋状态，导致严重浪费CPU资源，降低了并发性。

解决 CAS 恶性空自旋的较为常见的方案为：

(1)分散操作热点，使用 LongAdder 替代基础原子类 AtomicLong。

(2)使用队列削峰，将发生 CAS 争用的线程加入一个队列中排队，降低 CAS 争用

的激烈程度。JUC 中非常重要的基础类 AQS（抽象队列同步器）就是这么做的。

2. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则

更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会

发现它的值没有发生变化，但是实际上却变化了。

JDK 提供了两个类 AtomicStampedReference、AtomicMarkableReference 来解决 ABA 问

题。

3. 只能保证一个共享变量的原子操作。一个比较简单的规避方法为：把多个共享变量合并

成一个共享变量来操作。 JDK 提供了 AtomicReference 类来保证引用对象之间的原子

性，可以把多个变量放在一个 AtomicReference 实例后再进行 CAS 操作。比如有两个

共享变量 i＝1、j=2，可以将二者合并成一个对象，然后用 CAS 来操作该合并对象的

AtomicReference 引用。

## 19.ABA

说明：ABA问题是在多线程并发情况下发生的一种现象，与CAS有关。

ABA发生的情况：有一个共享变量a，其值为1。线程t1在使用CAS时判断a的值为1就更

新a的值，但是线程t2比线程t1先执行了，t2修改了若干次变量a的值并最终让a的

值又变回了1。此时线程t1仅仅判断a的值为1就进行更新操作而忽略了a的状态，

即变量a的修改次数、变量a是否被修改过。有些业务不关心中间过程，只要前后值一

样就行，而有些业务则要求变量在中间过程中值不能被修改，这就要求判断变量是否被

修改过以及修改的次数。

解决方式：使用AtomicStampedReference的版本号机制。

代码示例：

*/\*\*  
 \* 这里泛型是使用的包装类Integer。构造方法第一个参数是比较并交换的值，第二个是标志，  
 \* 类型为int，可以理解为第一个参数的版本号  
 \*/*AtomicStampedReference<Integer> atomicStamped =  
 new AtomicStampedReference<>(2,1);  
new Thread(()->{  
 //获取版本号  
 System.*out*.println("a1->"+atomicStamped.getStamp());  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 */\*\*  
 \* compareAndSet()方法比较值时用的是==，对于Integer来说，当值不在-127~128之间时，  
 \* 会new一个新的Integer对象，因为==比较的两个对象的地址值从而一直更新失败。  
 \* 所以这里用的是-127~128之间的值，比如2和4  
 \*/* System.*out*.println(atomicStamped.compareAndSet(2, 4,  
 atomicStamped.getStamp(), atomicStamped.getStamp() + 1));  
 System.*out*.println("a2->"+atomicStamped.getStamp());  
 System.*out*.println(atomicStamped.compareAndSet(4, 2,  
 atomicStamped.getStamp(), atomicStamped.getStamp() + 1));  
 System.*out*.println("a3->"+atomicStamped.getStamp());  
},"a").start();  
  
new Thread(()->{  
 //获取版本号  
 int stamp = atomicStamped.getStamp();  
 System.*out*.println("b1->"+stamp);  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(atomicStamped.compareAndSet(2, 6,  
 stamp, stamp + 1));  
 System.*out*.println("b2->"+atomicStamped.getStamp());  
},"b").start();

## 20.锁的种类

公平锁：指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，每次都由队首

的线程获取锁，当其释放锁之后，再按照排队的顺序由下一个线程获取锁，类似于超市中排队付款。优点是等待锁的线程不会饿死，也就是只要你排队，最后你肯定能付款。缺点是整体吞吐量低，等待队列中除队首的线程外其他线程都会阻塞。

非公平锁：指多个线程同时尝试获取锁，能抢到锁就直接占有，抢不到就到等待队列的队尾

去等待。如果抢到了锁，那该线程就无需阻塞，所有会有后申请锁的线程反而先获取到锁的场景。优点是可以减少唤醒线程的开销，整体吞吐量更高。缺点是可能会有线程一直处于饥饿状态，即一直都获取不到锁。

synchronized是非公平锁，而Lock可以通过构造方法中的不同参数分别得到公平和非公平锁。

可重入锁：指一个线程不用释放，可以重复的获取一个锁n次，只是在释放的时候，也需要

相应的释放n次。（简单来说：A线程在某上下文中或得了某锁，当A线程想要再次获取该锁时，不会因为锁已经被自己占用，而需要先等到锁的释放）假使A线程已经获得了锁，又在等待锁的释放，就会造成死锁。

synchronized和ReentrantLock都是可重入锁。

自旋锁(spinlock)：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线

程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。获取锁的线程一直处于活跃状态，但是并没有执行任何有效的任务，使用这种锁会造成busy-waiting。代码示例：

public class SpinLock {  
 private AtomicReference<Thread> cas = new AtomicReference<Thread>();  
  
 public void lock() {  
 Thread current = Thread.*currentThread*();  
 // 利用CAS  
 while (!cas.compareAndSet(null, current)) {  
 // DO nothing  
 }  
 }  
  
 public void unlock() {  
 Thread current = Thread.*currentThread*();  
 cas.compareAndSet(current, null);  
 }  
}

示例说明：lock()方法利用的CAS，当第一个线程A获取锁的时候，能够成功获取到锁，

不会进入while循环，如果此时线程A没有释放锁，另一个线程B又来获取锁，此

时由于不满足CAS，所以就会进入while循环，不断判断是否满足CAS，直到A线程调用unlock方法释放了该锁。

自旋锁的问题：

1. 如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。

2. 上面Java实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

自旋锁的优点：

1. 自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；

不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快。

补充：自旋锁本身无法实现公平性和可重入性，但是可以添加这两种性质。CLHLock和

MCSLock是高性能的公平自旋锁，它们都通过链表的方式避免了减少了处理器缓存同步，极大的提高了性能，区别在于CLHLock是通过轮询其前驱节点的状态，而MCS则是查看当前节点的锁状态。CLHLock在NUMA架构下使用会存在问题。在没有cache的NUMA系统架构中，由于CLHLock是在当前节点的前一个节点上自旋,NUMA架构中处理器访问本地内存的速度高于通过网络访问其他节点的内存，所以CLHLock在NUMA架构上不是最优的自旋锁。

乐观锁：首先乐观锁和悲观锁都是用于解决并发场景下的数据竞争问题，但是却是两种完全

不同的思想。它们的使用非常广泛，也不局限于某种编程语言或数据库。乐观锁指的是在操作数据的时候非常乐观，乐观地认为别人不会同时修改数据，因此乐观锁默认是不会上锁的，只有在执行更新的时候才会去判断在此期间别人是否修改了数据，如果别人修改了数据则放弃操作，否则执行操作。冲突比较少的时候, 适合使用乐观锁，由于乐观锁的不上锁特性，所以在性能方面要比悲观锁好，比较适合用在DB的读大于写的业务场景。Java中乐观锁的实现方式为CAS和版本号机制。

悲观锁：指的是在操作数据的时候比较悲观，悲观地认为别人一定会同时修改数据，因此悲

观锁在操作数据时是直接把数据上锁，直到操作完成之后才会释放锁，在上锁期间其他

人不能操作数据。悲观锁适用于写大于读的业务场景。Java中其实现机制为synchronized

和Lock的加锁方法，Mysql数据库中则是有对应的排他锁。

死锁：指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一

种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系

统产生了死锁，这些永远在互相等待的线程称为死锁线程。