目录

[1. 重载 2](#_Toc24063)

[2. 封装：私有实例变量并赋予get和set方法。 2](#_Toc23292)

[3. toString代替print输出 2](#_Toc29224)

[4. \t=tab键 \n=换行 3](#_Toc24687)

[5. 基本数据类型占用字节 3](#_Toc8106)

[6. 逻辑运算 3](#_Toc4984)

[7. +=赋值运算 3](#_Toc29822)

[8. 读取键盘输出 4](#_Toc26951)

[9. if语句 4](#_Toc7114)

[10. switch语句 4](#_Toc2135)

[11. While循环 5](#_Toc21087)

[12. Do...while循环 5](#_Toc12656)

[13. Break语句 5](#_Toc22580)

[14. continue语句 6](#_Toc26034)

[15. 方法的定义 6](#_Toc3788)

[16. JVM内存结构 6](#_Toc23023)

[17. 方法递归 7](#_Toc13218)

[18. 面向对象的三大特征 7](#_Toc2529)

[19. this关键字 7](#_Toc24334)

[20. 抽象类 8](#_Toc4603)

[21. Interface 接口 8](#_Toc29507)

[22. 访问控制权限 8](#_Toc27885)

[23. Equals 9](#_Toc17496)

[24. Finalize 9](#_Toc23055)

[25. Clone 10](#_Toc18807)

[26. 匿名内部类 10](#_Toc7240)

[27. 数组Array 11](#_Toc26792)

[28. String工具类 11](#_Toc32061)

[29. StringBuffer：一个字符串缓冲区 13](#_Toc14803)

[30. 包装类 14](#_Toc7868)

[31. 日期处理 15](#_Toc17134)

[32. 数字格式化 16](#_Toc13509)

[33. 随机数 16](#_Toc32364)

[34. 枚举 16](#_Toc21666)

[35. 异常 17](#_Toc18032)

[36. UML统一建模语言 19](#_Toc10726)

[37. 集合 20](#_Toc21074)

[38. IO流 30](#_Toc10324)

[39. 多线程 36](#_Toc8557)

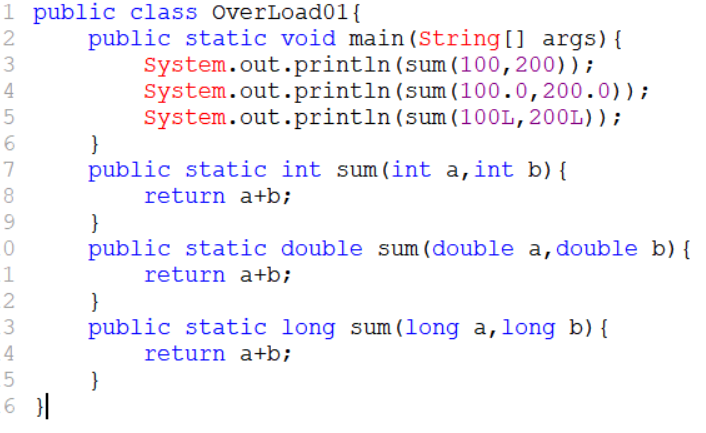
[40. 反射机制 42](#_Toc18932)

[41. 注解 46](#_Toc17754)

[学完javase之后还需要学数据库和JDBC以及需要了解java框架等考完研之后继续学习。 47](#_Toc8933)

# 重载

对于一些功能类似的方法，可以用一个方法名然后写不同的内容，主方法在调用时， 可以直接输入相应数据类型的数据，编译器会自动匹配并执行相应的方法：



方法的重载只与方法名和形参有关系，与返回值类型和修饰符无关，不管定义的方法是int还是double等不同的类型，如果形参相同那么编译器就不会通过，认为是方法重复。

本质上println方法其实也是重载，只不过println中的方法是sun公司已经定义好的，所以可以直接调用。

**小贴士**：可以先定义一个类，里面就专门重载System.out.Println方法，设类名为S，方法名为p，那么以后就可以用S.p来代替System.out.println来输出。相比一直用System.out.println要方便一点。

# 封装：私有实例变量并赋予get和set方法。

先定义私有变量private m；

然后用getset方法实例化

public void setID(int id){ this.id=id; }

public int getID(int id){ return id; }

引用实例：在主方法中用新建大类对象(Student stu=new Student();)

然后给stu赋值(stu.setID("值");)

最后输出用System.out.println(""+getID)

注意：在赋值时用set，在输出取值时用get。

# toString代替print输出

public String toString(){return ""+stu.getID+'';}

重写方法制定输出，最后再在主方法中用print实例化：

System.out.println(new shuchu());

toString方法时Object类中的方法，而所有的类都继承自Object类，所以在任何类中都可以重写toString方法。toString方法在直接输出类名或者对象名时，系统会自动在后边加上***.toString()方法，***未重写的toString方法格式为【类名@地址】，地址是由哈希算法算出来的。而重写toString方法后，由于继承关系以及系统自动调用的原理，直接输出类名或对象名时，会自动调用重写后的toString方法，这样就可以根据需求输出自己想要的结果。

# \t=tab键 \n=换行

\'=‘ \\=\ \"="

\u表示后面的是一个字符的Unicode编码(16进制) 例：char i='4e2d'输出为 “中”

# 基本数据类型占用字节

**byte 占1个字节，int 占4个字节，long占8个字节，short占2个字节**

byte、char、short做混合运算时，各自先转换成int在做运算。

多种数据类型做算法时，按占位最大的类型算。

long占用8个字节，float占用4个字节，但float容量>long容量。

注意：任意一个浮点型都比整数型空间大。

# **6. 逻辑运算**

逻辑与 &：有假为假，全真为真。

逻辑或 |：有真为真，全假为假。

逻辑取反。

短路现象：在进行逻辑与运算时，会将逻辑与两边的判断式都运行一遍在输出结果，即使已经知道结果也仍然会运行两边的表达式。

但在进行短路与时，如果仅仅判断短路与左边一个表达式就可以知道结果，那么右边的表达式就不会运行，直接输出结果。

例m=10,n=11,判断m>n & m>n++为假,那么n=12.

m=10,n=11,判断m>n && m>n++为假，那么n=11.

短路与&&：运算结果与逻辑与没有任何差别，但会出现短路现象。

短路或 | |：同上。

# 7. +=赋值运算

x+=1与x=x+1并不完全相同；

x+=1运算时，永远不会改变运算的类型，但x=x+1会改变，1系统默认为int类型，若x不是int编译会报错。

严格来讲，应该这么写，例：byte x +=1应该等于byte x=(byte)(x+1).

如果超过了数据类型的表示范围，则也会损失精度。

同类型如：-=，\*=，/=，%=都是一样的道理，同上。

条件运算符：（三目运算符）

格式为：布尔表达式 ？ 表达式1：表达式2

如果布尔表达式为真则执行表达式1，若为假则为表达式2。

例：boolean sex = false；char c=sex ？ '男' : '女' ;输出结果为：女

# 8. 读取键盘输出

1.创建一个键盘扫描器对象：

java.util.Scanner sc=new java.util.Scanner(System.in);

2.然后接受键盘上的输入，接受一个数据：int i=s.nextInt();

字符串输入：String str=s.next();

# if语句

例：boolean sex=true;

if(sex)

System.out.println("男");

System.out.println("helloworld!");

else

System.out.println("女");

会在else报错，如果if下只有一条java语句则可以不加大括号执行，但如果有两条语句则自动认为从第二条语句开始就跳出if语句，那么

else就缺少了之前的if语句，就会语法错误。

# switch语句

Switch中的变量只能是int和String类型。

switch(值/变量)

case 值1：

java语句；

java语句； break;

.....

case 值2：

java语句；

java语句；

break;

......

......

default:

......;

break;

switch的穿透现象：

正常switch语句若不加break语句，就会出现穿透现象，例：

int i=sc.nextInt();

String wk="周一";

switch(i){

case 1:

case 2:

wk="周二";

break;

case 3:

wk="周三";

break;

}

System.out.println("输入的是 "+wk);

}

若是case 1中没有break语句，那么在输入1时，不仅会产生case 1的结果，而且仍然会出现case 2的结果，若是case 2中也没有break语句，那么结果就会继续穿透，直到有break语句或者switch语句结束为止。如上述程序的运行结果为：输入的是：周二。如果case语句中直接用System.out.println语句输出，输出就会是周一和周二。

case合并：

Switch语句可以这么写：

switch(值):

case 值1: case 值2: case 值3:

java语句;

break;

case 值4:

.....

那么当变量满足值1或2或3时，都会输出相同的结果。

# While循环

本质上while与for循环的原理相同。

语法：while(布尔表达式){

循环体;

更新表达式;

}

//死循环

While(true){

System.out.println(“系循环”);

}

# Do...while循环

先执行循环体中的代码，执行一次循环体之后，在判断布尔表达式的结 果，true则继续执行，false则结束循环。（先执行，后判断。）

语法：do {

循环体

}while(); //封号不能丢。

与while的区别：do...while执行1~n次，while执行0~n次。

# Break语句

Break语句只能用在两个地方：

1. Switch语句当中，用来终止switch语句的执行。
2. 用在循环语句当中，用来终止循环的执行。

# continue语句

continue作用时终止本次循环，直接进入下次循环继续执行。

for(){

if(){

Continue;}

//当Continue执行时，以下代码不执行，直接执行更新表达式。

Code1;

Code2;

Code3;

Code4;...}

# 方法的定义

格式：[修饰符列表] 返回值类型 方法名 （形参列表）{

方法体；

}

修饰符列表是可选项，可写可不写，现在学习一边都用public static；

返回值类型：无返回值就用void代替，且不能省略，有返回值如String、int....等类型时，就要用return “abc”/123...；返回相应的类型。否则会报错。且方法遇到return时方法必然结束。

Break和Return的区别：

Break只是终止了循环，而return是直接终止方法，方法中出现return表示方法结束，若是循环之后还有输出，输出就不执行了。

# JVM内存结构

JVM主要分为三块内存空间：栈、堆区和方法区。运行程序时，方法区类加载器会先将.class文件放到方法区中，所以说方法区中存储的是一些代码片段。而代码中的一些变量需要的空间则是在栈分配的。

Java的两大变量分为局部变量和成员变量：

**局部变量：**局部变量一般在主方法main中，所以会在栈内存的main方法帧中创建，遵循栈的内存结构，先进后出，在main方法结束后会自动释放回收。局部变量没有默认值，需要手动赋值。

**成员变量：**成员变量是对象/类级别的变量，这类变量系统会自动赋值，成员变量又分为两种：一种是实例变量，另一种是静态变量：

*实例变量：*是对象级别的变量，又称属性，是对象的属性，在JVM内存结构中，实例变量存放在堆内存中。因此，在调用实例变量时，一定要实例化对象之后才能调用他的方法。

*静态变量：*静态变量是被static关键字修饰的变量，和静态方法一样都是被存放在JVM中的方法区中。静态变量是类级别的变量，不需要实例化对象就可以直接用【类名.】调用。

# 方法递归

定义：方法自己调用自己就叫方法递归。递归没有结束条件就会一直运行直到出现栈溢出错误（StackOverflowError）。所以递归必须要有结束条件。

递归假设是有结束条件的，也有可能会出现栈溢出错误，因为有可能递归的太深，一直在压栈，栈内存不够，仍然会出现栈溢出错误。

实际开发中，不建议轻易使用递归。递归使用不当，会导致JVM死掉。（极少数情况下，只能用递归）。

# 面向对象的三大特征

特征：封装、继承、多态

**封装：所有的类都继承自Object类，Object是所有类的父类！**

主要是为了数据的安全性，在设置变量为私有(private)时，可以用get/set方法将变量封装起来，这样就可以提高数据的安全性，同时还可以降低程序的耦合度。

[被private关键字修饰的变量只能在本类中调用，而如果要从其他类中调用就要进行封装，用get/set方法给外部类提供一个读取/修改的权限，这样在外部类访问该变量时就只能通过get/set方法来进行修改和读取，提高数据的安全性]

**继承：**当父类的功能或者属性无法满足子类的需求时，可以使用继承来拓展父类的功能，其语法是（子类方法 extends 父类方法）

**多态：**多态其实就是指实例化对象时可以有多种表达方式（例：Animal a=new Dog()）,多态可以应用在很多地方，最主要还是在其他方法中调用子类方法，当程序员不知道子类是否属于父类时可以使用，当然，在方法中调用的实参也是可以用多态的方式，本质上也是提高了效率，降低程序的耦合度

**注意：**在使用多态创建对象（例：Animal a=new Dog()）,在程序编译时，系统认为a变量是一个Animal类型的，但在程序运行时，创建的时Dog类型的对象，所以虽然编译上是Animal类型，但由于Dog是Animal的子类，所以a变量实际还是指向Dog。变量a可以调用子类的方法和父类的方法(继承)，这种方式叫做**向上转型，**属于**自动类型转换【小转大】**。

另一个方式就是创建父类型对象赋给变量,然后夫类型变量强制转换为子类型变量（这种方式叫做**向下转型**属于**强制类型转换【大转小】**），例如：Pet p1=new Pet();Dog d1=(Dog)p1;这时在使用a变量时最好要将a强制转换为Dog类型(在强转时，要注意使用【a instanceof Dog】来判断a变量是否指向Dog实例（是返回true否则返回false）。强转前p1只能调用父类Pet方法，强转后都可以调用。

【向下转型一般用的比较少】

# this关键字

this关键字其实就是指在本类中。在同一个类中直接使用实例变量时系统会自动给他添加一个***this.***所以在大多数情况下this可以省略 。

但是在构建有参构造方法时，为了区别与对象属性同名的实参时就需要用this来区分。

例如：public Animal(String name , int age){

this.name=name;

this.age=age; }

另外还有一种写法：***this()*** 这种写法是指对象本身，即像[Aimal()]这种形式。所以在无参构造方法中可以直接用【this(实参)】的形式给属性附上默认值。

**注意：**this()这种写法只能出现在构造方法的第一行！

# 抽象类

[修饰符] abstract 类名(){}

抽象类用abstract 修饰，abstract既可以修饰类也可以修饰方法。

在修饰方法时，方法只能在抽象类中，且当抽象类被继承时，该方法一定要被实现。且抽象方法不能带有方法体【**例：void doSome();**】,这里的方法可以不带控制符，因为抽象方法默认为公开的（public）。换句话说，抽象方法就是用来被重写（实现）的。

当修饰类时，抽象类可以被继承，抽象类中可以有非抽象方法，但又抽象方法时，子类必须实现他的抽象方法。抽象类依旧可以继承自抽象类。

**注意：抽象类其实就是用来被继承的，抽象类中可以有非抽象方法，但抽象方法必须在抽象类中。最后，抽象类是半抽象的。**

# Interface 接口

接口也是一种抽象类，他是完全抽象的，因此接口类中只能有抽象方法。

接口的作用顾名思义，就是用来连接两个类（英语中 like a 就是接口的意义），例如：厨师与菜单之间，菜单就是一个接口，他连接着顾客与厨师，对于顾客来说，厨师就像菜单，点什么就上什么。对于厨师来说菜单顾客也想菜单，点什么他就做什么。例2，电脑与硬件之间，硬件接口就是一种抽象接口。对于电脑来说，接口连接着硬件，让他具备硬件的能力，接口是可选的，电脑可以不带鼠标，也可以通过接口带上鼠标。

接口就是项目开发中最为关键的所在，接口与接口对应的功能可以单独分离出来，接口与实现接口的对象也可以分离出来，所以找到接口就可以大大提高开发项目的效率。

# 访问控制权限

访问控制权限的4个修饰符：private、protect、public、default(空白不写也可)

访问控制修饰符可以修饰**属性（4个都可），方法（4个都可），类（public&默认）、接口（public&默认）······**

private:Private修饰的只能在本类中使用

pulbic:在哪儿都可以访问

protect:protect修饰的只能在本类、同包、子类中访问

default(默认):表示只能在本类、同包中访问

**访问控制修饰符 本类 同包 子类 任意位置（其他包）**

**Public √ √ √ √**

**Protect √ √ √ ×**

**Default(默认) √ √ × ×**

**Private √ × × ×**

**范围从大到小：public > protect > 默认 > private**

# Equals

Object类中有一个equals方法，但他比较的是地址，所以和“==”没有区别，所以要使用equals方法一般都要重写,格式为：

**public boolean equals(Object obj){**

**//当对象是个空值时。直接返回false**

**if(obj == null){**

**return false;**

**}**

**//当对象不是比较对象时（本例中不是MyTime对象时）返回false**

**if(!(obj instanceof MyTime)){**

**return false;**

**}**

**//当对象和自己作比较（或者对象是相同的基本数据类型时），返回true**

**\*/**

**if(this == obj){**

**return true;**

**}**

**//到这一步说明对象既不是空值且是相同的类型且不是自己比较自己**

**MyTime t = (MyTime) obj;**

**if(this.year == t.year && this.mouth == t.mouth && this.day == t.day){**

**return true;**

**}**

**//属性不匹配，自然返回false**

**return false;**

**/\***

**当然用这种方式更好：**

**if(obj == null || !(obj instanceof MyTime)){**

**return false;**

**}**

**if(this == obj) return true;**

**return this.year == t.year && this.mouth == t.mouth && this.day == t.day**

**这是最为简便的比较方法也是IDEA自动生成的equals方法的格式。**

**\*/ }**

**注意：由于是覆盖，所以不管是方法名还是返回值类型又或者是实参的类型都不能更改。**

**“ == ”只能用于比较基本类型，不能用于比较两个引用类型。（“==”比的是地址！）**

# Finalize

protect void finalize（） throw Throwable{}

finalize是一种类似“遗嘱”的存在形式，这个方法不需要手动调用，他是JVM中GC（Garbage Collection垃圾回收器）自动调用的方法，源码中finalize并没有什么方法，要使用的话只需要重写即可，但垃圾回收机制并不是一定会启动，只有达到一定量的垃圾时才会启动。

**判别是否为垃圾：对象指向空(Person p = null)**

还有一种建议启动垃圾回收机制的方法：**System.gc()**；这种方法也不是一定会启动，只是启动的概率大大增加了而已。

# Clone

源码： protected native Object clone() throws CloneNotSupportedException;

Protect Object clone() throws CloneNotSupportedException(){

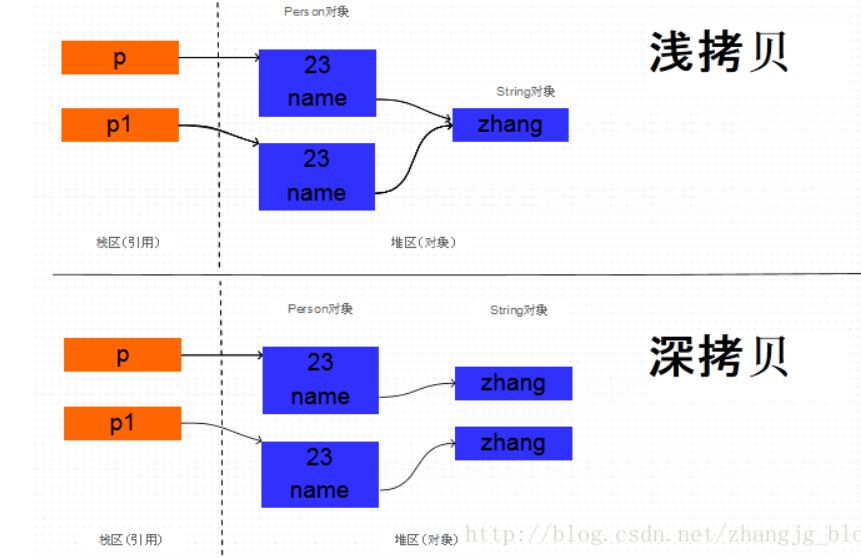
return (Person)super.clone();

}

Clone方法在使用时一般分为两种：**浅克隆**和**深克隆**

**·浅克隆：**浅克隆是Object类中自带的clone方法，在基本数据类型clone时没有什么问题， 但在引用类型clone时他是将新的对象的引用属性地址指向之前对象的引用类型地址，因此clone出来的对象，不仅基本数据类型是一样的，连引用类型数据的地址也是一样的。

**·深克隆：**深克隆需要自己重写clone方法，原理是新建引用类型的地址让两个对象的引用类 型地址不用，变量指向不同的地址



# 匿名内部类

匿名内部类顾名思义就是没有名字的类。

定义一个interface接口(例：Compute接口)，在实例化接口方法（例：sum方法）时，理论上是不能直接实例化接口的，但是可以用这种方式：

mm为对象，mm中有一个mySum方法调用sum方法。

mm.mySum(new Compute(){

public int sum(int a,int b){

return a+b;

}

},100,200);【接口后面的大括号绝对不能省略】

以这种在给对象赋值时临时实现的、没有名字的类中的方法叫做**匿名内部类，**实际上很变扭，可读性差且无法重复使用，只是看起来高端，用的比较少。

# 数组Array

int[] array; double[] array1；......

在C++中是：int array[] 这样定义的（java中不建议这样使用）

一维数组、二维数组、三维数组、多维数组....

1. 数组时一种引用类型变量，存放在堆内存中，自带length属性
2. 数组可以存储基本数据类型，也可以存储引用数据类型
3. 当数组中存放的时引用类型对象时，实际存放的是对象的地址
4. 数组中的对象类型一定是统一的。数组一旦创建长度不可变
5. 数组的地址其实就是第一个元素的对象地址
6. 数组中的元素是有下标的，从0开始，最后一个元素的下标是length-1
7. **数组的对象地址在内存空间上是连续的，这是一种数据结构**

**优点：** 由于在数组的地址是第一个元素的地址，而数组中的元素在内存空间上是连续的， 所以当知道了元素的数据类型（类型所占的字节，引用类型是地址所占的字节）和下标时， 就可以很轻松的算出某个数据的内存地址，从而找到那个数据，因此对于数组来说查找100 个元素中的某个元素和100w个元素，在效率上来讲是相同的，这是数组最大的优点。

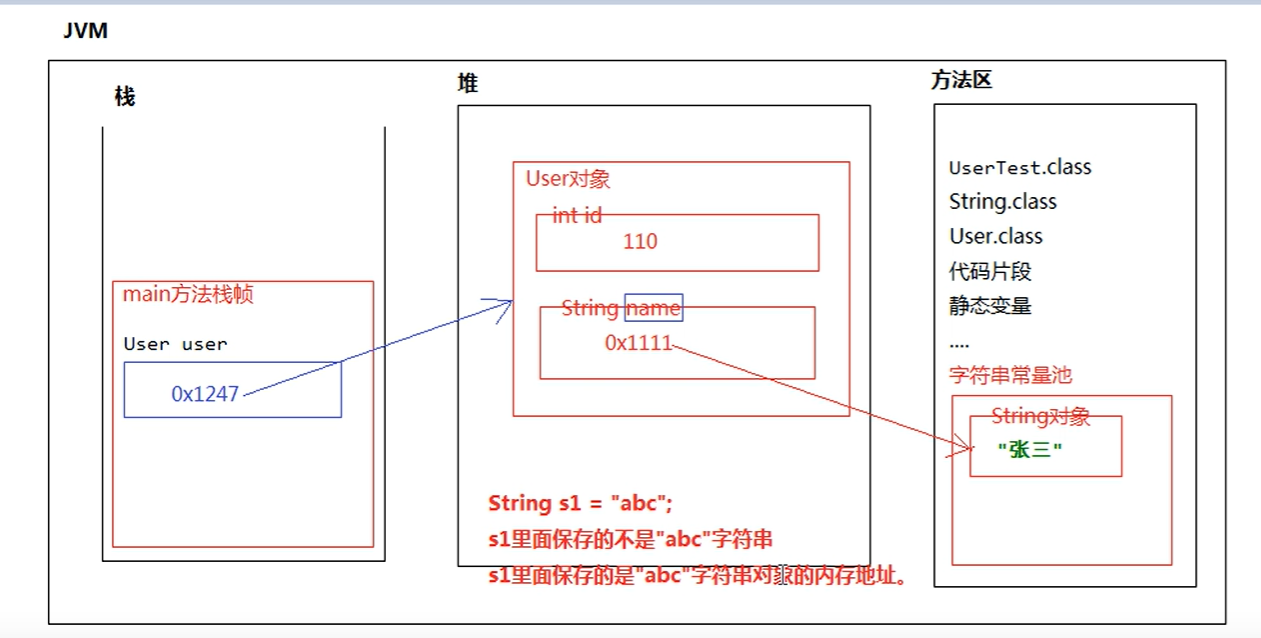
**缺点：** 显而易见，由于要求在内存空间上连续，所以在存储需要相当大的长度的数组时， 内存中很可能找不到这么大的内存。其次，由于在内存上要求连续，在数组中进行插入、 删除操作时，需要移动大量的数据元素地址，因此效率反而很低（但在数组末尾进行插入 删除操作时，并不会影响效率，因为后面没有数据，不需要更改地址）

**数组工具类**

Java.util.Arrays;

1. Arrays.sort(目标数组):排序
2. Arrays.binarySearch(目标数组,目标字符串):二分法查找，但要在排序好的基础上

# String工具类

1. String是**引用数据类型**，String类型使用双引号括起来的数据，所有用双引号括起来的数据都是一个String类型对象，这个对象的内容不可变
2. String字符串被存储在**方法区**的**“字符串常量池”**中，主要是因为字符串在实际开发中使用的太频繁，为了执行效率，所以将字符串存放在方法区中的”字符串常量池”中。
3. String字符串**对象**不能使用双等号（“==”），双等号比的是对象地址，用双等号比较时容易出错。可以用equals比较，因为在String方法中已经重写了equals方法。
4. **String的构造方法**

new String(“\*\*\*”):输入什么输出什么

new String(byte[]):输出ASCII码对应的值

new String(byte[],1,2):输出byte数组中，下标为1的值开始往后在输出两位

new String(char[]):挨个输出char数组中的所有字符

New String(char[],1,2):输出char数组中，下标为1开始往后在输出两位

1. **String的方法**

charAt(int i)：返回字符串中指定位置的值。

compareTo(String s):按字典顺序比较两个字符串大小,相等返回0，前大后小返回值大于零，前小后大返回值小于零。从第一个字符串的第一个字符开始比较，若不相等则直接出结果，相等就继续比。

**contains(String s):**字符串中是否包含s字符串，boolean类型

**endWith(String s):**判断字符串是否是以s字符串结尾，boolean类型

**equalsIgnoreCase(String s):**判断两个字符串是否相等，且忽略大小写

**byte[] bytes= s.ge**t**Bytes():**将字符串s转换为byte数组(即在ASCII码中的位置)

**indexOf(String s):**判断某个子字符串在当前字符串第一次出现的下标，int类型

**isEmpty():**判断字符串是否为空字符串。boolean类型

**length():**字符串长度。注意：在**字符串**中使用此方法是调用的length方法**(.length())**，而在数组中使用length，length是**数组**的**属性(.length)**。

**lastIndexOf(String s):**判断子字符串在当前字符串中最后一次出现的索引，int类型

**replace(CharSequence target,CharSequence replacement):**String类型

例:String s=“http://mail.com”.repalce(“http”,”https”)

这时字符串s就变成了“https://mail.com”。

注意：这个方法是**全部替换**。

**String[] s=String s1.split(String s):**拆分字符串

例：String[] s=”1999-04-25”.split(“-”).以“-”为分隔符进行拆分。

数组s就是{“1999”,”04”,”25”}.

**startsWith(String s):**判断某个字符串是否是以某个字符串开始的,boolean类型

**substring(int i):**从前面的字符串的下标为i开始截取到结束形成新的字符串，String类型

**substring(int i,int j):**从前面的字符串的下标为i开始截取到下标为j的字符，形成新的字符串，String类型,(左闭右开)

**toCharArray():**将字符串转换为char类型并放入char数组。char[]类型

**toLowerCase():**将字符串全部转换为小写。String类型

**toUpperCase():**将字符串全部转换为大写。String类型

**trim():**去除字符串前后的空白(字符之间的空白并不去除)。

**valueof(Objcet obj）:**将非字符串数据转换为字符串类型，如果是引用类型的对象，valueof是调用的obj对象的toString()方法。用println输出的值都是字符串形式，因为在println的底层代码都有valueof()方法，之前为何输出对象会调用toString方法，就是因为println方法输出对象时，会将对象用valueof()转换成字符串，而valueof()方法在对对象使用时，又会调用toString()方法，这就是原因。

**初始化数组**

**一维数组：**

**静态初始化：**

int[] array = {100,200,100,300};

**动态初始化：**

int[] array = new int[5] //这里的5表示长度

//初始化了一个长度为5，类型为int的int类数组，且每个元素的默认值为0

String[] array1 = new String[6] //同理，默认值为null

**数组的扩容：**

实际上就是建立一个更大的数组，将之前的数组中的数据拷贝进来。如果需要更大的数组长度，就需要新建更大的数组容量。（数组扩容效率较低）

扩容的方法：

**System.arraycopy(*源数组*,*源数组的拷贝起点*,*目标数组*,*目标数组的起始下标*,*拷贝的长度*);** 这里有5个实参，一个都不能少

**二维数组：**

二维数组其实就是特殊的一维数组，他的每个元素都是一个一维数组。

**静态初始化：**

int[][] array={{100,200},{5,6,4,8,5},{1}}

**动态初始化：**

int[][] array = new int[3][4];

**三维数组：三维数组其实就是特殊的二维数组，每个元素都是一个二维数组**

**多维数组也是这个道理。**

# StringBuffer：一个字符串缓冲区

StringBuffer是一个类名，所以在使用时要new一个对象：

StringBuffer stringbuffer = new StringBuffer();

之后用append(要加的内容)将数据拼接进去。

StringBuffer在初始化时是初始化了一个长度为16的字符串缓冲区。但在创建StringBuffer对象时最好是预估一下，给定一个大一点儿的初始化容量，这样可以优化StringBuffer这个类。

StringBuffer的构造方法：

StringBuffer():初始化长度为16的缓冲区

StringBuffer(int i)：初始化长度为i的缓冲区

StringBuffer(CharSequence seq)：初始化一个包含seq相同的字符的缓冲区

StringBuffer(String s):初始化一个内容为s的缓冲区

**StringBuilder与StringBuilder：**

与StringBuffer的区别：基本相同，但也有一些区别

1. StringBuffer中的所有方法都是**有synchronized（同步）**关键字修饰的，他表示StringBuffer在多线程环境下运行是**安全**的。
2. 而StringBuilder中的方法并**没有synchronized**关键字修饰，这就表示StringBuffer在多线程环境下运行是**不安全**的。

# 包装类

8种基本数据类型有对应的8种包装类

作用：

在有些方法中的实参为Object，那么8种基本的数据类型是无法直接传入的，而此时包装类就可以将原本是基本类型的数据转变成一个引用类（其实就是创建一个类，然后包装起来变成一个对象）。

我们并不需要自己写，SUN公司已经写好了。

只有int和char类型区别较大，除了Boolean和Char直接继承自Object，其他6种并不直接继承Objcet，而是继承Number

使用方法：

基本数据类型 包装类型

byte java.lang.Byte

short java.lang.Short

int java.lang.Integer

long java.lang.Long

float java.lang.Float

double java.lang.Double

boolean java.lang.Boolean

char java.lang.Character

Number类中的方法

Number是一个抽象类，他有6个方法**(用于拆箱)**：

byteVlue()、doubleValue()、floatValue()、intValue()、longValue()、

shortValue()

例：Integer i = new Integer(123);

float f = i.floatValue();//输出就是123.0

这就是引用类型-(转换为)->基本数据类型（俗称：**拆箱**）

而用Integer i = new Integer(123);这种方式是将

基本数据类型(123)-(转换为)->引用数据类型(俗称：**装箱**)

**以Integer类型为例：**

**Integer有两种构造方法**，一种是直接传入int类型数字Integer(int i)，直接对int数据进行装箱（即int-->Integer），还有一种是传入String类型的数字Integer(String s)，同样是将String类型数字转换为int进行装箱(String-->Integer)。

现在可以实现自动装箱和自动拆箱：

例：Integer i = 100; //自动装箱

int j = i; //自动拆箱

除了以上方式，自动拆箱只会在使用 + - \* / 运算时会才会触发 ，用 == 是不会触发自动拆箱的。

**注意：**java中为了提高效率，将[-128 到 127 ]之间所有的包装对象提前创建好，放在方法区中的“整数型常量池”中，目的是只要用这个区间的数据不需要在new对象，直接从常量池中取出，所以在这个区间内的数据比较时直接比较地址，而如果数据相同，那么地址也相同。

**Integer的方法：**

1. intValue()方法，手动拆箱
2. parseInt(String s)方法,静态方法,int类型,将字符串数字转化为整数型数字，如果字符串不是数字，则返回NumberFormatException异常。
3. 了解方法：
   * 1. toBinaryString(int i):静态方法，String类型，10进制转2进制
     2. toHexString(int i):静态方法，String类型，10进制转16进制
     3. toOctalString(int i):静态方法，String类型，10进制转8进制
4. valueOf(int i/String s):int/String类型的数字转换为Integer对象，返回Integer类型的对象。

# 日期处理

包地址：java.util.Date

方式：Date nowDate = new Date();但以这种方式输出的日期是以美国为标准，其他国家有自己的日期格式，可用SimpleDateFormat(java.text)格式化日期

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

y表示年(4位)，M表示月(2位),d表示日(2位)，H表示小时(2位),m表示分钟(2位)，s表示秒(2位),S表示毫秒(3位)。中间的间隔可用其他方式代替表示。

输出此格式的日期方式：sdf.format(nowDate);即可。（Date --> String）

也可以将String类型的日期转换位Date类型的日期

String s = “2008-08-08 08-08-08 888”;

SimpleDateFormat sdf1 = new SimpleDateFormat(“yyyy-MM-dd hh:mm:ss SSS”);

Date date = sdf1.format(s);

如果SimpleDateFormat的格式与String类型日期的格式不同，则会有ParseException异常。

**一些方法**

//获取自1970年1月1日 00：00：00 000到当前系统事件的毫秒数(静态方法)

long nowTimeMils = System.currentTimeMillis();//long类型

//获取一个程序运行的毫秒数  
 long begin = System.currentTimeMillis();  
 for (int j = 0; j < 10; j++) {  
 System.out.println("j="+j);  
 }  
 long end = System.currentTimeMillis();  
 System.out.println("0-10000循环所花的毫秒数："+(end-begin)+"毫秒");

**总结一些现在学到的System方法**  
 //System.out静态方法:  
 System.out.println();  
 //建议启动垃圾回收器  
 System.gc();  
 //获取自1970年1月1日 00：00：00 000到当前系统事件的毫秒数(静态方法)  
 System.currentTimeMillis();  
 //退出JVM虚拟机System.exit(0)是正常退出程序，而System.exit(1)或者说非0表示非正常退出程序  
 System.exit(0);

**Date的有参构造中的参数是一个long类型的值，就是毫秒数**  
 Date time1 = new Date(1);//这里就表示1970-01-01 08:00:00 001,即北京1970年毫秒为1的时间  
 SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss SSS");  
 String sdfStr= sdf.format(time1);  
 System.out.println(sdfStr);//1970-01-01 08:00:00 001  
 //1970年到昨天的此时的毫秒数  
 System.out.println(System.currentTimeMillis() - 1000 \* 60 \* 60 \* 24);  
 //往前10天的此时的时间  
 String s4 = sdf.format(System.currentTimeMillis() - 1000 \* 60 \* 60 \*24 \* 10);  
 *// String s4 = sdf.format(1000);1970-01-01 08:00:01 000*  
 System.out.println(s4);

# 数字格式化

Java.text.DecimalFormat;

**DecimalFormat df = new DecimalFormat(“###,###.##”);**

**数字格式：# 表示任意数字 ， 表示千分位 . 代表小数点**

**0 表示不够时补0**

**例：**String s = df.format(1234.56) //输出 1,234.56，用String接受

DecimalFormat df1 = new DecimalFormat(“###,###.0000”);

String s1 = df1.format(1234.56)//输出 1,234.5600,字符串接收

**BigDecimal**

**属于大数据，精度极高。不属于基本数据类型，属于java对象(引用数据类型)，这是一个类。用于财务软件中。**

BigDecimal v1 = new BigDecimal(100);//这里的100精度极高

BigDecimal v2 = new BigDecimal(200);//这里的200精度极高

BigDecimal v3 = v1.add(v2);//输出v3 = 300;由于是引用数据类型，不能使用简单的 + - \* / 符号进行操作，只能调用方法

# 随机数

Java.util.Random

**Random random = new Random();**

int i = random.nextInt();//产生int类型范围内的随机数

random.nextInt(101);//随机取0-100之间的随机数（左闭右开）

# 枚举

Enum枚举：由于开发过程中可能会出现多种结果，但又不是无数，用boolean类型不够，其他类型麻烦，所以就有了枚举，用以处理这种情况。

1. 枚举编译之后也是生成的class文件
2. 枚举也是一种引用数据类型
3. 枚举中的每一个值都可以看作是一个常量

例：enum Result{

//*SUCCESS,FAIL*都是Result类型中的一个值

*SUCCESS,FAIL*

}

调用: Restlt.***SUCCESS;***

用枚举的一般都是超过两种情况的，例如：颜色，四季，星期等

# 异常

try{ }catch(){ } //之后可以继续添加catch

throws

定义：程序执行过程中不正常的情况。

作用：增强程序的健壮性，程序在遇到异常时不会宕机(死机)。

程序出现的异常称为**“异常信息”**，是由JVM打印的供程序员参考。

异常在java中以类的形式存在。（可以new出来,例：new NullPointException()）

实际上程序在运行时，如果出现了异常，JVM会自动new一个异常对象并抛出，从而就出现了在控制台上显示的那些异常信息：

int c = 10 / 0;

//JVM执行到这里会new一个异常对象：new ArithemticException(“/ by zero”)

//然后JVM将new的异常对象抛出，打印到输出信息的控制台。

Exception的父类是Throwable，Throwable的子类除了Exception以外还有一个Error类，Error类是不可处理的，当程序遇到Error中的错误时，程序只有一个结果：退出。但Exception不同，程序出现的Exception异常是可以处理的。但不管是Exception还是Error都是可抛出的（throws）

所有Exception的直接子类都叫做编译时异常，所有的RuntimeException及其子类都属于运行时异常。编译时异常并不是在编译时发生的，编译时异常表示必须在编写程序的时候预先对这种异常进行处理 ，如果不处理编译器会报错。而运行时异常在编写程序阶段你可以选择处理，也可以不处理。

**编译时异常**(**CheckedException**、受检异常/受控异常)一般发生的概率比较**高**。

例：看到外面下雨，所以带一把伞，防止淋雨生病

异常：生病 预处理：带伞

**运行时异常**(**UnCheckedException**、未受检异常/非受控异常)一般发生的概率比较**低**。

例：出门在外可能会被飞机轮子砸。

异常：被飞机轮子砸 预处理：没必要处理，若是处理整个程序运行的会很累

若java中没有划分编译时异常和运行时异常，所有的异常都需要被预处理，首先程序会绝对的安全，但同时程序员的工作量会大大增加，代码到处都是异常。而实际上将所有异常都处理也并不现实，也不可能。

**注意：所有的异常都发生在运行阶段**

**java中对异常的处理方式：**

1. 在方法的声明上使用throws关键字，抛给上一级
   1. 谁调用就抛给谁，上级仍然需要对异常进行处理，也是两种方式
   2. 若果一直抛，抛给main方法，main方法抛给JVM虚拟机，JVM知道这个异常后只有一个结果：终止java程序的执行
   3. 在编写程序时，如果在方法名后面加了throws 【异常】，那么他的调用者一定要去处理这个异常，否则就要继续上抛，像他的子类一样同样使用throws 【异常】抛给上级，如果选择不上抛，就要用try...catch语句捕捉异常并处理。
   4. 在抛出异常时，父类如果也要抛，那么抛别的类是不行的，此问题仍然没有被处理，但可以抛出这个异常的父类异常类，这样是可行的。
   5. 在程序遇到异常时，异常语句忠厚的语句、方法、类等都不会运行，程序会回到捕捉异常的处理方式。如果父类仍然抛出就继续抛出，也不会运行父类异常之后的语句（类似return），直到catch语句捕捉。但try...catch语句之后的代码还是会执行，因为catch语句执行完之后表示异常已经处理完了，所以程序可以继续往下走。
2. 使用try...catch语句进行异常的捕捉
   1. try表示尝试执行，如果遇到异常就直接转到catch语句，catch语句中的类似**(NullPointException e)**【**异常名 变量名**】语句中的e其实就是保存了底层NullPointException异常对象的地址。
   2. 使用catch捕获异常时，建议不同的异常不同处理（精确处理异常）
   3. **JDK8的新特性:**在使用catch捕获异常时，catch中的异常可以用”|”(“或”)来表示多个异常.例：

**catch ( NullPointerException | StringIndexOutOfBoundsException | ClassCastException e){ }**

**异常对象的两个重要方法:**

exception.getMessage()//获取异常简单的描述信息

getMessage()方法获取的是异常构造方法传入的字符串。如果没写字符串则返回的是系统默认的字符串。

exception.printStackTrace()//打印异常追踪的堆栈信息

printStackTrace方法实际是由底层的一个线程(异步)专门负责，输出的是运行时异常类似的结果，但直接使用这个方法并不会让其后面的语句不执行，但由于是线程负责，所以输出的内容可能在最后，也可能在它后面的语句之前或者循环语句中间。实际开发中printStackTrace方法用的比较多。

**try...catch中的finally子句**

1. finally子句中的代码是一定会执行的，即使try中捕捉到了异常。finally子句必须要和try一起出现，不能单独编写。
2. 在有些情况，创建了流之后使用完需要关闭流(否则会占用资源)，而如果在创建流之后代码中出现了异常，而导致流的关闭代码不能运行，会使程序非常危险且占用资源，所以一般流的关闭是写在finally子句中。但由于try中的变量无法直接在finally中使用，所以可以在try外部创建流使他为空(null)，之后在finally中再用try...catch语句对close异常进行处理。

例：

FileInputStream fis = null;  
try {  
 fis = new FileInputStream("E:\\学习\\hello.java");  
 String s =null;  
 s.toString();//空指针异常

//fis.close();若之前有异常这里无法执行  
} catch (FileNotFoundException e) {  
 e.printStackTrace();  
}catch (NullPointerException e){  
 e.printStackTrace();  
}finally {  
 if(fis != null){  
 try {  
 fis.close();//所以放在finally中可以保证流的关闭  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. try和finally可以同时使用而不带catch，但try不能单独使用，若是try语句中有return，那么程序遇到return之后会先执行finally中的语句再return(try-->finally-->return)。

例：

try {  
 System.out.println("try....");  
 return;  
}finally {  
 System.out.println("finally....");  
}

1. 但如果try中有System.exit(0);//退出虚拟机,那finally就无法执行了，且只有退出虚拟机才能让程序不执行finally。
2. 一些规则。例：

int i =100;  
 try {  
 return i;  
 }finally {  
 i++;

}

以上代码执行之后返回的仍然是100。原因：通过底层代码得知，这个程序会将i的值赋给一个新的变量j，然后执行i++，但最后return的是j的值。这里有一个原则：方法体的代码一定是自上而下逐行执行的，所以在执行到try语句时，理论上应该直接返回，但是有finally的存在，所以只能先确定了返回值并赋给新的变量之后在进行finally子句中的语句，最后用return结束.

**自定义异常**

在实际开发中，sun公司的异常可能不够用，此时可以自己定义异常。

第一步：编写一个类继承Exception或者runtimeException

第二步：提供两个构造方法，一个无参数的，一个带String参数的

例：

public class MyException extends Exception{  
 public MyException(){}  
 public MyException(String s){  
 super(s);  
 } }

# UML统一建模语言

UML是一种统一建模语言。是一种图标式语言（画图）。UML不只是在JAVA中使用。只要是面向对象的编程语言，都有UML。

一般画UML图的都是软件工程架构师或者说是系统分析师这些级别的人员使用的。软件设计人员使用UML。

在UML图中可以描述类与类之间的关系，程序执行的流程，对象的状态等。在java软件开发中，软件分析师/设计师负责设计类，java软件开发人员必须能看懂。

**画UML的工具：Rational Rose(收费)、starUML等**

# 集合

**位置：java.util.\***

数组就是一种简单的集合。集合实际上是一种**容器**，可以容纳其他类型的容器。

集合在开发过程中使用很多。集合是一个容器，是一个载体，可以一次容纳多个对象。

在实际开发中，如果连接数据库，数据库中有10条记录，假设把这10条数据查询出来，在java程序中就会封装10个对象，然后将这10个对象(引用地址)放到某一个集合中，将集合传到前端并遍历，将这10个数据展现出来。

集合**不能**直接存储**基本数据类型**，也不能存储**java对象**，集合只能存储对象的内存地址(对象的引用)，**集合**在java中**本身**是一个容器，是一个**对象**。也就是说，多个集合可以将内存地址放在某一个集合中。就完成了在集合中存放集合。

集合在**任何时候存储**的都是**”引用”**。

不同的集合类型对应底层不同的数据结构：ArrayList、LinkedList、TreeSet.....（底层分别对应了数组集合、链表集合、二叉树集合）

**java中集合主要分为两大类：**

1. 以单个方式存储元素：

单个方式存储元素，这一类的集合中超级父接口是：java.util.Collection;

1. 以键值对的方式存储对象：

以键值对的方式存储对象，这一类集合的超级父接口是：java.util.Map;

**Collection的结构**

**如下图：**

Collection的父接口（继承、泛化）是Iterable接口，他从Iterable接口中可以继承一个iterable()方法，用于创建(返回)一个Iterator(迭代器【迭代的意思是遍历】)对象，他有3个方法：hasNext()、next()、remove(),这个迭代器与Collection是关联关系(has a)，说明所有集合都是可迭代的。

Collection有两个主要的接口：List、Set。当然还有其他的接口。

1. **List接口**

List集合存储元素是**有序的**(先进先出)、**可重复**的(元素可以相同)，存储的元素有下标(从0开始，以1递增)。

List接口的几个主要类：ArrayList类、LinkList类、Vector类(现在不常用)

* 1. **ArrayList类**

ArrayList集合底层采用了**数组**这种数据结构。ArrayList是非线程安全的（方法没有synchronized修饰）。ArrayList采用无参构造创建时，创建的是一个初始容量为10的集合，有参构造则创建一个初始化容量为参数的集合。但在JDK8以后，无参构造先是初始化长度为0的集合，之后在添加(add)第一个元素时，初始化容量变为10.

* 1. **LinkList类**

LinkList集合底层采用了双向链表数据结构。

* 1. **Vector类**

Vector集合底层也是采用了数组的数据结构，但与ArrayList集合不同的是，Vector类中的所有方法是有synchronized修饰的，Vector集合是线程安全的。但效率较低，而现在保证线程安全有其他更好的方式，所以现在不怎么用了

1. **Set接口**

Set集合存储元素是**无序**的（存入和取出的顺序可能不一样）、**不可重复的**(元素彼此之间不能相同)。

Set接口的几个主要类：HashSet类、TreeSet类。

1. **HashSet类**

HashSet集合在new时，实际上时new了一个HashMap集合，而向HashSet集合中存储元素，实际上存储到了HashMap集合中的key部分了。

HashMap集合是一个哈希表数据结构

1. **TreeSet类**

TreeSet类实现的直接父类接口并不是Set接口，他实现的直接父类接口是SortedSet接口，SortedSet集合存储元素由于继承了Set接口，所以他存储的数据也是无序不可重复的，但放在SortedSet集合中的元素可以自动排列，称为可排序集合。放在SortedSet集合中的元素自动按照大小顺序排列的。

TreeSet集合底层实际上是TreeMap，在new TreeSet集合时，实际上也是new了一个TreeMap集合。而往TreeSet集合中放数据时，实际上也是将数据放入TreeMap集合中。TreeMap集合底层采用的是二叉树数据结构。

**TreeSet的具体讲解放在TreeMap上**

List集合和Set集合特点上几乎是完全相反。

Collection的一些常用方法：以**Collection c = new ArrayList()**;为例。ArrayList()数组是有序的可重复的，存入和取出的顺序相同，但是Set类型的数组是无序不可重复的，所以取出顺序并不一定与输入顺序相同，相同的数据也只会存一次。

**Collection详解**

1. **Collection**

**Collection**在没有“泛化”之前，可以存储Object任意子类型。使用了“泛型”之后，就只能存储某个具体的类型。当然也不也能存基本数据类型和java对象，只能存地址。

Collection集合之间可以通过构造方法直接转换，但Map和Collection不能直接转换。

1. **Collection的常用方法**
   * 1. add(Object e):[c.add(new Object())]boolean类型，存基本数据类型时自动装箱，添加的元素在集合尾部，如果集合满了会自动扩容，容量为原来的1.5倍。自动扩容效率较低，建议创建之前预估集合的容量大小。
     2. size():[c.size()]int类型，返回集合中元素的个数。不是集合的长度。
     3. clear():[c.clear()]void类型，清空集合中的元素。
     4. contains(Object o):[c.contains(object o)]boolean类型，判断集合中是否有o元素，而判断的方法在contains的**底层其实用的是equals方法**，也就是说，当判断一个元素是否在集合中时，即使那个元素并不在集合中，但集合中有一个和那个元素**equals比较后**是返回00的是**true**，那么contains方法仍然会返回true。总结上来讲就是：**放在集合中的元素要重写equals方法。**
     5. remove():[c.remove(Object o)]:boolean类型，删除集合中o元素。与contains原理相同，如果一个删除非集合中的元素，而这个元素又恰好与集合中的某个元素的equals方法返回是true，那么集合中那个与非集合元素相同的元素会被删除，原因也是因为remove方法底层使用到了equals方法。

在使用迭代器迭代元素时不能使用集合中的该方法，否则会出现同步异常。

* + 1. isEmpty():[c.isEmpty()]:boolean类型，判断集合是否为空。
    2. toArray():[c.toArray()]:Object[]类型，是数组类型，调用这个方法可以将集合转化为数组。
    3. Collection的Iterator迭代器/遍历问题：
       1. Iterator迭代器是Collection的父类接口Iteration的方法，个人理解其实就是一个指针。Collection的任意集合都可以适用iterator迭代器，所以Map集合不能用。使用方法为：**Iteration it = c.iterator();it就是一个迭代器。**指向的是集合头部，并不直接指向集合中的元素。迭代器如果在集合添加元素之前就创建好了，那么迭代器指向的是空集合的头部，所以集合结构只要发生改变，迭代器就要重新建立。否则会出现**同步异常**：java.util.ConcurrentModificationException
       2. 迭代器iterator有三个方法：
          1. hasNext():[it.hasNext()]boolean类型，判断集合中是否有下一个元素。
          2. next():[it.next()]Object类型，iterator迭代器向后移一位，返回iterator迭代器当前指向的元素。

遍历集合中的元素：

while(it.hasNext()){

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

}

* + - * 1. remove():[it.remove()]boolean类型，删除迭代器当前指向的对象。

**2.1 List子接口特有的常用方法和特点**

特点：有序可重复。

* + 1. add(int index,Object obj):void类型，在列表指定位置（index）添加元素（obj）。用的较少，因为效率低。
    2. get(int index):返回Object类型，取得指定下标(index)的元素。可以根据List集合有序的特点，用get方法遍历集合。这是List集合特有的遍历方法。Set没有。
    3. indexOf(Object obj):int类型，获取指定元素(obj)第一次出现的索引。
    4. lastIndexOf(Object obj):int类型，获取指定元素(obj)最后一次出现的索引。
    5. remove(int index):Object类型，删除指定下标(index)的元素并返回。
    6. set(int index,Object obj):Object类型，修改指定位置(index)的元素，将这个位置的元素改为obj元素。
    7. **ArrayList类的常用方法和特点**

特点：检索效率高，随机增删元素效率低，但在数组末尾添加元素效率很高，不受影响，但数组类型集合无法存储大数据量。因为很难在内存上找到一块非常巨大的空间用来存放数据。数组集合在实际开发中使用的比较多。

1. ArrayList(Collection c)构造方法:将一个Collection的子集合转换为ArrayList集合。
2. 其他方法和前面差不多
   * 1. **LinkedList类 双向链表**

优点：由于在空间存储地址上是不连续的，所以不涉及大量元素的位移，因此随机增删元素效率比较高，以后开发中如果增删业务比较多时，选择LinkedList比较好。  
 缺点：查找效率低，每次查找某个元素都要从头结点开始往下遍历。  
 注意：链表也是有下标的，ArrayList之所以检索效率高并不单纯是因为有下标的原因，其实主要是底层数组发挥的作用。LinkedList也一样有下标，只是检索/查找某个元素的效率比较低，因为只能从头结点开始找。以后开发中主要是在末尾加元素，所以ArrayList在以后开发中比LinkedList使用的频繁的多。LinkedList没有初始化容量。

* + 1. **Vector类**

Vector底层也是数组，且他的初始化容量是10，自动扩容每次是之前容量的2倍，Vector所有的方法都是线程安全的，都有synchronized(线程同步)关键字修饰，效率低，使用较少。  
 **如何将一个ArrayList集合转换成线程安全的呢 ?**  
 使用集合工具类：Collections(java.util.Collections)

注意与Collection(java.util.Collection)的区别，Collection是一个接口类。

Collections工具类中有一个synchronizedList方法，将ArrayList数组传进去就行。

例：List mylist = new ArrayList();

Collections.synchronizedList(mylist);

mylist.add(1);.....

这样就可以将ArrayList类型的mylist变成线程安全的。以后可能会经常使用！

**2.2 Set接口**

Set接口底层是HashMap，他主要的两个子接口HashSet和TreeSet没有什么特别的地方，只是注意HashSet实际上是存在HashMap中的key部分，且是无序不可重复的，这里的无序是指存入与输出的顺序不一定相同。而TreeSet底层也是TreeMap，存入的数据也是在TreeMap的key中，需要注意的是这里存入的数据为什么是无序的，他输出时是按大小输出的，称为可排序集合，但他的存入与输出顺序仍然不同，所以也是无序的。

**2.3 增强for循环(foreach)**

语法：for(元素类型 变量名 ：数组或集合){

System.out.println(变量名);}

这样可以直接输出数组中的所有元素。但这种循环方式没有下标，在需要使用下标的循环中不建议使用。

增强for循环和普通for循环的区别：

当为数组时，效率几乎是差不多的

当为集合时效率却大不同，当为ArrayLIst时，普通for的效率高于增强for

当为LinkedList时，增强for的效率高于普通for

**Map的结构如下图**



Map存储的都键值对与Collection没有关系，他存储的元素都是**无需不可重复**的，Set集合就是Map的一种，Set集合存储数据其实就是在Map集合中的key部分存入数据。

1. **HashMap**

HashMap的底层结构是哈希表/散列表，是非线程安全的。

哈希表是一个数组和单向链表的结合体。数组查询效率高，随即增删效率低，而单向链表随即增删效率高，查询效率低。哈希表是将这两种数据结构结合到一起，充分发挥他们各自的优点。

在JDK8之后，如果哈希表中的单向链表长度超过8时，单向链表这种数据结构会变成二叉树(红黑树)数据结构。当二叉树上的结点数量小于6时，又会将二叉树重新变成单向链表数据结构。(其实也是为了提高查询效率)

HashMap集合初始化容量为16（若是自定义容量则必须是2的倍数，不然影响效率），默认加载因子是0.75(数组容量达到75%自动扩容)，自动扩容后是原容量的2倍。

哈希表底层是一个一维数组，每个元素是一个单向链表。单向链表中的每个节点都有4个变量:

Object k;//对应key，允许为null，但只能有一个

Object v;//对应value，允许为null

int hash;//通过key计算得出，同一单向链表的hash值可能相同也可能不 //同(不同时称为hash碰撞)，但对应数组的下标一定相同，因为 //下标是取余算出来的。

Node next;//指向下一个节点

所有存在HashMap集合中的key元素都要重写hashCode()和equals()这两个方法。因为HashMap的set/get方法底层调用了这两个方法。

1. **HashTable**

HashTable的底层结构也是哈希表，但他是线程安全(synchronize)的，因此效率比较低，用的比较少。但他有一个Properties类被称为属性类，他的键值只支持**String类型**，这个属性类也是线程安全的，用的比较多。

HashTable的key和value都是不允许null值的，若为null则抛出空指针异常。

HashTable的初始化容量是11，默认加载因子也是0.75，自动扩容是原来的容量的2倍在加1。

Properties（被称为属性类对象）是一个**Map对象**，继承HashTable，Properties的key和value都是String类型。是线程安全的。他有两个方法需要掌握：  
 创建对象：Properties pro = new Properties();

1. setProperty(String key,String value):Object类型 ，底层是HashTable的put方法，用于存放键值对。使用方法：pro.setProperty(String key,String value)
2. getProperty(String key):String类型(Properties的特性)，通过key(键)找到对应的value(值)。使用方法：pro.getProperty(String key)
3. **TreeMap**

TreeMap并不直接实现Map接口，他的直接父接口是SortrdMap接口，SortedMap接口的父接口才是Map接口，SortedMap接口的意思是：存储在SortedMap集合中的元素首先由于继承Map的关系，所以他也是无序不可重复的，但SortedMap集合中的key部分可以按照大小顺序排序。

TreeMap的底层结构是(自平衡)二叉树数据结构，遵循**左大右小**的原则存放

遍历二叉树有3种方式：

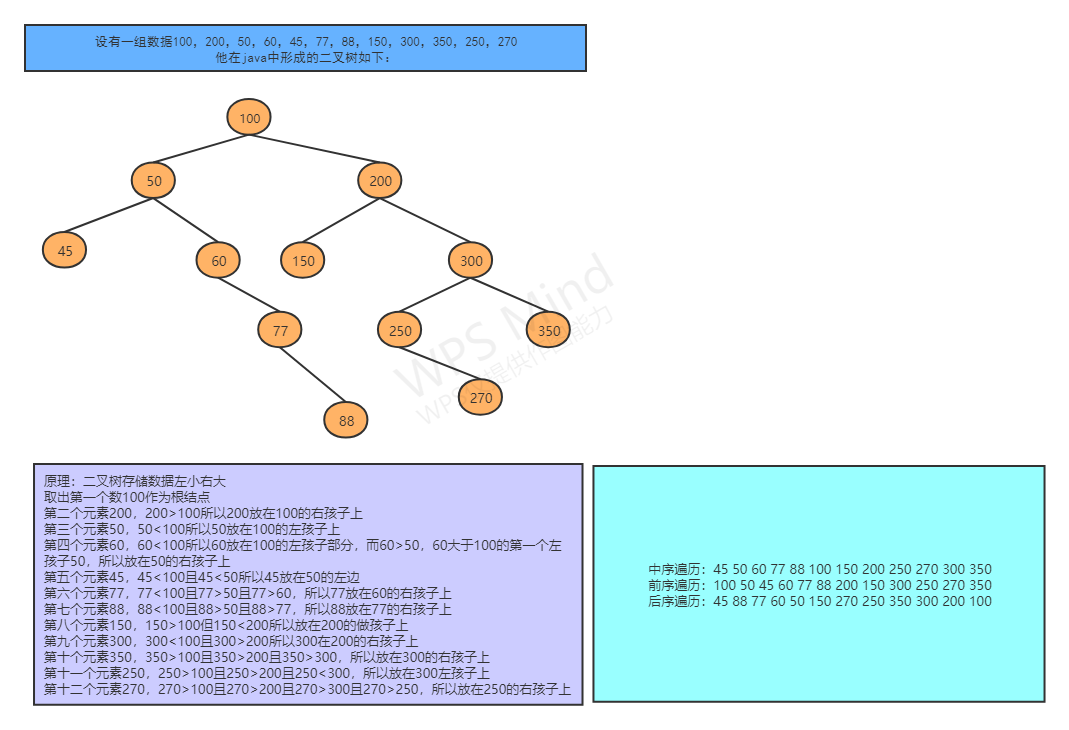
前序遍历：根左右

中序遍历：左根右

后序遍历：左右根

前中后指的是根的位置。TreeMap/TreeSet集合采用的是**中序遍历**方式。**Iterator迭代器也是采用的中序遍历。**

二叉树原理：



放到TreeSet集合中的元素，等同于放到了TreeMap集合的key值(数组)部分。放在TreeSet中的元素虽然无序不可重复，但可以按照元素大小自动排序(可排序集合)

自定义类型TreeSet不能排序，因为没有指定自定义类对象之间的比较规则。如果强行插入会报异常：ClassCastException。如果要使自定义类型加入到TreeSet集合中，那么自定义类型就要实现Comparable[java.lang]接口并重写的compareTo（Object o）方法或者自己定义一个比较器类继承Comparator()[java.util]。具体的计较规则需要自己定义。

若是重写了compareTo方法，那么equals方法可以不重写。

实现Comparable(java.lang)接口方法:

例：Person类中有age属性，根据age属性判断类对象的大小

lass Person implements Comparable<Person>{  
 int age;

public int compareTo(Person o) {  
 return this.age - o.age;//从小到大排序  
 }}

实现Comparator(java.util)接口方法：

例：Vip类中有age属性，根据age大小判断类的大小

class VipCompare implements Comparator<Vip>{  
 public int compare(Vip v1, Vip v2) {  
 return v1.age - v2.age;}}

**注意**

如果实现的是java.lang.Comparable接口实现的是compareTo方法，那么创建TreeMap/TreeSet集合对象时可以无参。但如果是用比较器，实现java.util.Comparator接口的compare方法，那么创建TreeMap/TreeSet集合对象时不能使用无参构造，且有参构造要将比较器对象放入实参，这样TreeMap/TreeSet集合才能正常比较。当然比较器可以以匿名内部类的方式直接放入实参，不需要类名，直接在参数位置new接口并实现他的compare方法。

如何选择Comparable(java.lang)和Comparator(java.util)?

如果比较规则只有一个，建议使用Comparable接口。如果比较规则有多个，需要多个规则之间来回切换，建议使用Comparator接口。

**Map接口详解**

**Map**

1. Map和Collection没有继承关系
2. Map集合是以键值对的方式存储数据：key和value
   1. key和value都是引用数据类型
   2. key和value都是存储对象的内存地址
   3. key在Map中起到主导作用，value是key的一个附属品
3. Map的key和value都可以使用泛型

**Map接口的常用方法**

1. clear():void类型

作用：清空Map集合

1. containsKey(Object key)：boolean类型

作用：判断Map集合中是否包含某个key

1. containsValue(Object value):boolean类型

作用：判断Map集合中是否包含某个value

这些contains方法底层调用的都是equals方法来比较的，所以自定义的类型需要重写equal s方法

1. get(Object key):泛型V（value）类型

作用：通过key获取value

1. isEmpty():boolean类型

作用：判断Map集合中的元素个数是否为0

1. keySet():Set泛型K（key）类型

作用：获取Map中所有的key(所有的键(key)是一个Set集合)

1. put(K key,V value):泛型V(value)类型

作用：向Map集合中添加键值对

1. remove(Object key):泛型V(value)类型

作用：通过key删除键值对，返回value类型

1. size():int类型

作用：获取Map集合中键值对的个数

1. values():Collection<V>类型

作用：获取Map集合中所有的value，返回一个Collection集合

1. entrySet():Set<Map.Entry<K,V>>类型

作用：将Map集合转换成Set集合，Set集合中存储的数据是Map.Entry<K,V>类型，实际上这也是个泛型，限制的类型就是Map.Entry<K,V>。Map.Entry和String一样，是一种类型的名字，但Map.Entry是Map集合中的静态内部类。Map.Entry<K,V>源码中其实就是一个节点，这个结点有getKey和getValue方法，entrySet的底层就是Set集合中存这种结点。

设有一个Map集合：map1

key value

1. zhangsan
2. lisi
3. wangwu
4. zhaoliu

在使用了entrySet()方法后(map1.entrySet()),可以用Set集合接收：

Set set = map1.entrySet();

就将上述格式的Map集合转化为：

1 = zhangsan

2 = lisi

3 = wangwu

4 = zhaoliu

这种形式，并将其加入到Set集合中。

以上的方法在**IDEA的CollectionTest包中的MapTest类**中都有测试，详情查看具体的测试类。Map集合中遍历集合的各种方法比较重要，在开发中遍历使用的很多。foreach和entrySet方法的结合使用可以让Map集合的效率比较高，适用于大数据量。其他的获取键放入Set集合，再根据Set集合用Map的get方法遍历也可，但效率较低，数据量少时并不影响。

**Map接口存入取出原理**

**map.put(k,v)原理**

首先，先将k(key),v(value)封装进Node对象中，然后底层会将k使用hashCode()方法将k转化成哈希值，然后通过哈希算法/哈希函数转化成数组下标，如果下标位置上没有任何元素，就把Node对象加入到这个位置上，如果下标对应的位置上有链表，那么就会拿着k和链表上的每一个节点的k进行equals比较，如果返回的都是false那么这个新节点就会添加到这个链表的末尾，如果有一个结点返回了true，那么就会将新节点的value值覆盖原来节点的value值。

**map.get(k)原理**

先调用k(key)的hashCode()方法得出哈希值，通过哈希算法转化成数组下标，然后通过数组下标快速定位到某个位置，如果这个位置什么也没有，就返回null。如果这个位置有单向链表，那么就会拿着这个k值和链表中的每一个节点的k值进行equals比较，如果都返回false，那么get方法就返回null，如果有一个节点的k equals比较之后返回的是true，那么就返回这个结点的v(value)值。

**总结原理**

通过以上原理分析可知，HashMap无序是因为存入的数据不知道是放在那么链表上或者数组中，所以输出时也不会按照存入时的顺序输出。而不可重复是因为put方法会将重复的key对应的value值覆盖旧的value值。存在HashMap中的某个位置上单链表上所有结点的hash值都是一样的，他们对应了他们所在的数组下标，但他们的k与k之间的equals方法返回的肯定都是false，都是不同的。

为什么哈希表的增删查询效率比较高? 因为他的增删是在链表上实现的，而查询也只需要部分扫描所以增删查询效率都比较高。

**重点：**Map接口的put和get方法的k底层都调用了这两个方法：hashCode()和equals()。因此所有存在HashMap集合中的key元素都要重写这两个方法！而HashMap的key集合实际上就是Collection中HashSet集合，所以所有存在HashSet集合中的元素也都要重写以上两个方法。

equals方法无论是在get还是put方法中，只有当哈希表的数组下标对应的是null时不调用，其他都会调用。而且在用Map集合时，总是先进行hashCode算出hash值之后才可能使用equals方法，所以如果对象不同但内容且没有重写hashCode方法，那么两个相同内容的对象很可能就存放在不同的下标中，而重写了hashCode方法但不重写equals方法的话，两个对象hash值相同但equals方法是false，很可能就会出现一个单链表上有相同的数据。就违背了Map集合的特点，所以：**Map集合的key元素对象一定要重写equals方法和hashCode方法！**

HashMap集合的初始化容量为16，默认加载因子是0.75(即当集合容量达到75%时开始扩容)。为了防止**散列分布不均匀**(hashCode返回固定值或返回的值都不一样导致数组某一位置的单链表过长或数组长度过长)，HashMap集合的初始化容量必须为**2的倍数**，这也是官方推荐的，为了达到散列分布均匀，是提高HashMap的存取效率所必需的。

hashCode方法和equals方法可以用IDEA方法直接生成，但如果是在类似Map集合这种需要同时使用hashCode和equals方法的集合，IDEA直接生成时要**同时生成**。

**Collections集合工具类**

Collections集合工具类的方法

1. synchronizedList:用法：Collections.synchronizedList(List list),将一个非线程安全的List集合转换为线程安全的。
2. sort:用法：Collections.sort(List list),对一个List数组进行排序。这里要注意不是所有的List集合都可以排序，如果用这个方法排序则需要保证List集合中的元素对象实现了Comparable/Comparator接口并重写了比较方法之后才可以使用。
3. 若需要对Set集合排序则需要将Set集合转换为List集合之后才能进行排序

**泛型**

泛型是在JDK5之后才推出的一种机制。属于编译阶段的一个特性。Collection类的迭代器也可以使用泛型，这样可以直接调用泛型类型的方法，但其子类仍然要“向下转型”。

**优点**：集合中存储的元素类型统一。从集合中取出的元素类型是泛型指定的类型，不需要进行大量的”向下转型”。

**缺点：**泛型使集合中的元素缺乏多样性。

例：

Animal cat = new Cat();  
Animal dog = new Dog();  
List<Animal> mylist = new ArrayList<Animal>();  
mylist.add(cat);  
mylist.add(dog);  
//mylist.add(23);不是Animal类型无法存入  
Iterator<Animal> it = mylist.iterator();  
while (it.hasNext()){  
 Animal a = it.next();  
 a.move();  
}

如果要调用Animal类中的子类特有方法，其实还是避免不了转型，泛型只是方便了程序中的某个操作，子类特有方法依旧与要转型。

例：iterator迭代器的对子类特有方法的遍历。

if(a instanceof Cat){  
 Cat c1 = (Cat)a;  
 c1.catchmice();  
 }else if(a instanceof Dog){  
 Dog d1 = (Dog)a;  
 d1.bark();}

**钻石表达式**

JDK8之后引用了自动类型推断机制（又称为钻石表达式）

例：List<Animal> mylist = new ArrayList<>();

省去了对象的泛型限制，编译器会根据前面的泛型推断对象的泛型限制。之后与前面区别不大。在使用泛型特定方法时可以省去instanceof的判断，用泛型指定类型接受元素后可以直接使用泛型的特有方法。以后开发中经常使用。

**自定义泛型**

**例：class Student<asda>{  
 public void dosome(asda a){  
 System.out.println("zz");}}**

这个例子就表示如果new了一个Student对象且使用了泛型限制，那么在调用该对象的方法时传入的实参就一定要是泛型所限制的类型，否则无法调用。如果在new对象时不使用泛型，那么在调用他的方法时也要传入一个Object类型对象。

**注意**：泛型源码中“<E>”的E其实只是一个标识符，可以用任何字符代替，他表示了该类可以使用包括Object及其子类的所有类型的泛型限制，而一旦限制，那么该类中用到泛型的方法也与之相同，不相同则无法使用这个方法。

源码中主要用<E>(Element)和<T>(Type)来表示泛型。

**总结集合**

**各类集合的特点**

* ArrayList：底层是数组。
* LinkList：底层是双向链表。
* Vector：底层是数组，线程安全的，效率较低，使用较少
* HashSet：底层是HashMap，放到HashSet集合中的元素等同于将元素放到HashMap中的key部分了。
* TreeSet：底层是TreeMap，放到TreeSet集合中的元素等同于将元素放到TreeMap中的key部分了。
* HashMap：底层是哈希表。
* HashTable：底层也是哈希表，但是是线程安全的，效率较低，使用较少。
* Properties：是线程安全的，并且key和value是能存储String类型的值。
* TreeMap：底层是二叉树。TreeMap集合中的key可以自动按照大小顺序排列
* List、Set（Map）、SortedSet（SortedMap）集合的特点：
* List集合存储的特点：

有序可重复。

有序：元素有下表，先进先出。

可重复：元素值可以相同。

* Set（Map）集合存储的特点：

无序不可重复。

无序：元素无下标，存入顺序与输出顺序不一定相同。

不可重复：元素值不能相同。

* SortedSet（SortedMap）集合存储的特点：

无序不可重复，但SortedSet中的元素是可排序的。

可排序：可以按照大小排序

* Map集合的Key就是一个Set集合。往Set集合中放数据，实际上就是往Map集合中的key放数据，数据到了Map集合中的key部分中。

# IO流

IO流：I(input) O(output)

将硬盘上的文件传到内存上的这个操作被称为：

读（Read）、输入（Input）、输入流（InputStream）

将内存上的数据文件传到硬盘上的这个操作被称为：

写（Write）、输出（Output）、输出流（OutputStream）

**IO流的分类**

按照流的方向分类：  
 输入和输出都是以内存为参照物，数据传到内存中就是输入/读，数据从内存中出来就是输出/写。

按照读取数据的方式不同分类：

1. 按照字节的方式读取数据，一次读取一个字节byte(8个二进制位)。这种流是万能的，什么类型的文件都可以读取，例如：文本文件、声音文件、图片、视频  
    例：a阿萨sda2  
    第一次：”a” 第二次：”阿”的一半 第三次:”阿” ......
2. 按照字符的方式读取数据，一次读取一个字符，这种流只是为了方便读取普通文本文件(.txt/.java等能用记事本编辑的文件)，图片、声音、视频等文件都不能读取，甚至word也不行。  
    例：a阿萨sda2

第一次：”a” 第二次：”阿” 第三次：”萨” ......

综上所述，IO流的分类：

输入流、输出流 字节流、字符流

IO流的4大家族，都是抽象类

Java.io.InputStream 字节输入流

Java.io.outputStram 字节输出流

Java.io.Reader 字符输入流

Java.io.Writer 字符输出流

注意：java中的”类名”只要是以**Stream结尾**的都是**字节流**。以**Reader/Writer结尾**的都是**字符流**

所有的流都实现了Closeable接口，说明所有的流都是可关闭的，都有close()方法。流毕竟是一个内存与硬盘之间的管道，用完之后一定要关闭，否则会耗费(占用)很多资源。

所有的输出流(Reader/Output)都继承了Flushable接口，都有flush()方法，都是可刷新的。输入流在最终输出之后一定要使用flush()方法刷新一下。刷新管道会将管道中未输出的数据强行输出完(清空管道)，刷新的作用就是清空管道。如果不使用flush()清空管道，可能会导致丢失数据。

Java.io包下需要掌握的16个流：

文件专属流：

java.io.FileInputStream;   
 java.io.FileOutputStream;  
 java.io.FileReader;  
 java.io.FileWriter;

转换流：将字节流转换成字符流  
 java.io.InputStreamReader;  
 java.io.OutputStreamWriter;

缓冲流专属：  
 java.io.BufferedReader;  
 java.io.BufferedWriter;  
 java.io.BufferedInputStream;  
 java.io.BufferedOutputStream;

数据流专属：  
 java.io.DataInputStream;  
 java.io.DataOutputStream;  
 对象专属流：

java.io.ObjectInputStream;  
 java.io.ObjectOutputStream;  
标注输出流：

java.io.PrintStream;  
 java.io.PrintWriter;

**IO流详解**

1. FileInputStream

FileInputStream是字节流输入，构造方法为：

FileInput fis = new FileInputStream(“文件路径”);

这里的路径有绝对路径和相对路径两种：

绝对路径：即精确到某盘某文件

相对路径：从IDEA的默认(工程根目录)路径开始。

FileInputStream的方法：

1. read()：int类型，读取文件中的数据，每次一个字节，返回的是字节的ASCII码，如果没有数据了，这个方法会返回-1。

这种方法内存和硬盘之间的交互太频繁，不但耗费时间和内存且大部分时间都用在了数据传输中。

1. read(byte[] length):int类型，读取文件中的数据，每次最多读取byte数组长度的字节数，返回的数据是读到的字节数而不再是ASCII码，所以这个方法返回的最大值就是数组的长度，如果没有数据可以读取，就返回-1。

这个方法每次读取byte数组长度的数据，当数据不够byte长度时，就会将数据从0位置开始继续存放，比如6个数据：abcdef，byte数组长度为4 ，数组名为bytes，这个方法返回的数据用int data接收

这个方法第一次使用后在数组中的数据是abcd，但在第二次使用后数组中的数据为efcd。此时如果我们输出数组，那么会有一些我们并不需要的数据：第二次使用后数组中的”cd”。那么我们就可以用String工具类中的构造方法：

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

这样就可以只输出需要的数据了。

1. available()：int类型，返回的是文件中还没有读的字节数量。

用处：可以在使用read(byte[] length)读取文件之前，直接将byte数组的长度设置为available，此时available返回的是文件的字节数，这是就不需要使用while循环，一次可以读完全部数据。这种方式不适合大型文件，因为内存中可能不存在存放这种文件长度的数组对象。

1. sikp(int n): long类型，跳过n个字符。
2. FileOutputStream

文件字节输出流，负责写。从内存到硬盘。

FileOutputStream的构造方法：

FileOutputStream fos = new FileOutputStream(“文件路径”);

这里的文件路径如果不存在会新建这个文件，如果存在就清空文件后再写，如果不想清空而是再原文件后面添加数据则使用以下构造方法：

FileOutputStream fos = new FileOutputStream(“文件路径” ， boolean append);

若append为true则再不清空源文件的基础上添加数据，否则先清空数据。

FileOutputStream的write方法一般只能放byte类型数组，字符串类型（String）需要先用getBytes()方法将字符串转换为byte数组类型之后才能用write方法写入元素。

FileOutputStream的方法

1. write(int n):void类型，将n对应的ASCII码中代表的元素写入到文件中。
2. write(byte[] bytes):void类型，将数组bytes中元素代表的ASCII码表上所代表的元素写入到文件中。
3. Write(byte[] bytes , int start , int length):void类型，将byte数组中从start位置开始写入长度为length的数据。
4. flush():void类型，清空FileOutputStream管道，再写入输出数据之后必须使用，不然以后可能会丢失数据。
5. 文件复制copy

使用FileInputStream和FileOutputStream可以完成对文件的复制，这两种流的结合使用可以完成对任意文件的复制。文件复制（拷贝）的原理就是将文件从硬盘中读取到内存中，在从内存将数据写入到硬盘中，这两个操作时同步执行的（边读边写）。

方法：首先创建一个输入流对象和输出流对象，然后用read方法读取文件，若是知道文件的大小，可以设定数组长度让read方法读取效率更高，而写入方法也可以用

1. FileReader

文件字符输入流，只能读取普通文本，他读取的数组类型就是char[ ]类型，不再是byte[ ]类型。比Stream类读取效率更高，比较方便快捷。但使用方法和FileInputStream区别不大，所以不在细讲。

1. FileWriter

文件字符输出流，能读取char、String和char[]等类型，使用方式和FileOutStream差不多，也不再细讲。

1. BufferedReader

BufferedReader是带缓冲区的字符输入流，使用这个流不需要自定义char数组，也不需要自定义byte数组，他自己带缓冲。但在使用这个流是，这个流的构造方法中需要传一个Reader类型的实参流(例：FileReader)，如果不是Reader类型，也可以用InputStreamReader将字节流转换成字符流，这个传入的实参流被称为：**节点流**。而类似BufferedReader这样外部的流被称为：**包装流**或**处理流**。注意这种节点流和包装流都是相对的。

这种包装流的close()方法中自带有节点流的close()方法，也就是说，如果要关闭流只需要关闭这个包装流，实参 中的节点流不用手动关闭，会随着包装流的关闭而关闭。

BufferedReader的常用方法：

1. readLine():String类型，是这个缓冲输入流最重要的方法，读取一个文本行但**不带换行符**，返回的是一行的数据，若是读完了就返回null，所以输出readLine()方法的数据时最好用println。这种方法并不比FileInputStream的读取速度快，因为这毕竟是个包装类，但是比较方便所以很常用。
2. BufferedWriter

和FileWriter之类的其实没有什么本质区别。对应BufferedReader可以一行一行的写入。注意这里BufferedWriter的有参构造方法中也只能放字符的输出流(类似FileWriter等Writer类型的流),如果要放入OutputStream这样的字节输出流，就需要用OutputStreamWriter这样的转换流将字节流转换成字符流。

1. InputStreamReader

将字节流转换为字符流。这个类有一个构造方法可以传入一个字节输入流（Stream），这个构造方法可以将字节流转换为字符流。例：

BufferedReader br = new BufferedReader(new **InputStreanReader**(new FileInputStream));

1. OutputStreamWriter

将字节输出流转换为字符流，有参构造方法需要传入一个字节输出流(Stream)，就可以将字节输出流转换为字符流。例：

BufferedWriter bw = new BufferedWriter(new **OutputStreamWriter**(new FileOutputStream));

1. BufferedInputStream

缓冲输入流和其他的输入流区别不大，只不过是有参构造方法中，参数需要是字节输入流（Stream）类型，其他和普通的输入流差别不大

1. BufferedOutputStream

缓冲输出流和其他输出流区别也不大，有参构造方法需要传入一个字节输出流(Stream)类型,其他的方法和普通的输出流差别不大

1. DataOutputStream

数据专属的输出/写入流，有参构造方法中的实参必须是(File/Buffered)OutputStream类型。这个流可以将数据连同数据类型一起写入到文件中，但这个文件不是普通文本，因此记事本是打不开的（打开了也是乱码）。不同的类型有不同的write方法，如writeByte()方法、writeInt()方法等。DataOutputStream写的文件，只能用DataInputStream去读，并且读取的时候要提前知道写入的顺序。当读顺序和写的顺序相同时，才能正常取出数据。

1. DataInputStream

数据专属的输入\读取流，有参构造方法中的实参必须是(File/Buffered)InputStream类型。

读取的是DataOutputStream写入的数据，顺序必须和写入顺序相同才能正常读取数据。不同类型的read方法也是不同的，比如：readInt()、readByte()、readFloat()....

1. PrintStream

标出输出流，这个流不需要手动关闭，他的构造方法需要一个FileOutputStream类型对象，如果不加这个对象，直接写路径经过测试也是可以的，但他的底层仍然是使用了FileOutputStream,所以最好还是自己加。System.out.后边的方法都是PrintStream类型，可以用PrintStream类型对象接收，之后这个对象就可以直接使用println、print方法输出到控制台。

System有一个setOut()方法，这个方法可以改变输出的“方向”，不改变这个setOut方法，那么println、print方法都是直接输出到工作台。如果写了这个方法则可以自定义输出的位置。

方法：System.setOut(PrintStream ps):PrintStream对象指定了路径，将这个对象传入到setOut()方法的实参部分，那么以后使用println、print方法时都不会再向控制台输出，而是向PrintStream对象指定的路径输出。

主要用于日志文件。

1. PrintWriter

这个与PrintStream的区别就是：PrintWriter是字符方式，有参构造的实参必须是字符流(Writer)而PrintStream是字节方式，有参构造的实参必须是字节流(Stream)。其他作用区别不大。

1. ObjectOutputStream

对象序列化（Serialize），从内存中拆分对象放入硬盘中，将java对象的状态保存下来，这个对象再硬盘中被拆分成好几块，但都有序列，所以在放回内存时，也不会丢失数据。要是要用这个流，就需要让被序列化的对象实现Serializable接口，否则会报错：对象无法序列化。其次这个对象序列化流是一个包装流，内部需要一个OutputStream流。

例：

Student s = new Student(1111,”zhangsan”);

ObjectOutputStream oos = new ObjectOutputStream(new OutputStream(“students”));

oos.writeObject(s);

这个程序执行后，会创建一个students文件，里面就是存储的对象的信息，但这个文件用记事本无法打开。

1. ObjectInputStream

对象反序列化（DeSerialize），将被拆分放在硬盘中的对象恢复到内存中。

参与系列化和反序列化的对象必须实现Serializable接口。

Serializable接口是一个标志性接口，里面什么代码也没有。它的作用主要是起到标识作用，java虚拟机看到这个类实现了这个接口，可能会对这个类进行特殊待遇。这个标志性接口主要是给java虚拟机参考的，java虚拟机看到了这个接口，会为这个类自动生成一个序列化版本号。

**序列化版本号**

如果程序中没有手动写出来序列化版本号，那么java虚拟机会根据编译了当前类之后形成的新的class文件自动生成序列化版本号。虚拟机在对比两个类是否相同时，会先比较类名，再比较类的序列化版本号。但自动生成的序列化版本号会在改变类的代码之后也改变，会形成新的序列化版本号，这样就会导致在使用ObjectInputStream反序列化对象时出现错误。因此，我们在写继承了Serializable接口的类时，建议手动添加一个序列化版本号，这个序列化版本号是long类型，数字可以随便写，但在项目中需要区分开来。例：Private static final serialVersionUID = 12312343L;

序列化版本号应用场景：十年前的类十年后添加属性，反序列化十年前的对象时。

ObjectInputStream也是一个包装流，他的节点流必须是InputStream类型的。

例：

ObjectInputStream ois = new ObjectInputStream(new FileInptStream(“students”));

Object obj = Ois.readObject();//读取这个对象

System.out.println(obj);//输出这个对象的toString方法。

序列化对象也可以同时**序列化多个对象**，只要将这些对象放在集合中，但这些对象类也要实现Serializable接口，不然无法实现。

例：序列化：

List<User> users = new ArrayList<>();  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("users"));  
 users.add(new User(1,"zhangsan"));  
 users.add(new User(2,"lisi"));  
 users.add(new User(3,"wangwu"));  
 users.add(new User(4,"zhaoliu"));  
 oos.writeObject(users);   
 oos.flush();  
 oos.close();

反序列化：

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("users"));  
 List<User> userList = (List<User>) ois.readObject();

用List集合接受之后可以用增强for遍历出来。

若是并不像序列化对象所有的属性，可以用transient(游离的)关键字修饰这个属性，那么这个属性就会不参与序列化，反序列化时这个属性的所有值都时null

**File**

java.io.File

File和四大家族没有关系，所以和流也没有关系，File类不能完成读和写的操作。

File对象代表的是：文件和目录路径的抽象表示。(既可以是某个文件也和一是某个目录)，File类只是一个路径名的抽象表示。

**File的常用方法**

1. 构造方法：File类的有参构造传入的实参是一个路径或文件名

File file = new File(“文件路径”)

1. exist()：boolean类型，判断文件是否存在，存在返回true，不存在返回false

file.exist();

1. createNewFile()：void类型，创建一个新的“文件”
2. mkdir()：void类型，创建一个目录
3. mkdirs()：void类型，创建一个多重目录
4. getParant()：String类型，获取当前目录的父目录路径
5. getParantFile()：File类型，也可以获取当前目录的父目录路径，但要用File类型接收，再用这个File对象中的方法。

他的一些方法：

getAbsolutePath()：String类型，获取当前目录的父目录的绝对路径

1. getName()：String类型，获取文件名
2. isFile()：boolean类型，判断是否为文件类型，是返回true否则返回false
3. isDirectory()：boolean类型，判断是否为目录，是返回true，否则返回false
4. lastModify()：long类型，返回的是最后一次修改的时间，但他是从1970

年到最后一次修改时间的总毫秒数。这个就要用日期类，将毫秒数传进去，再用SimpleDateFormat设置日期格式，最后用format()方法将数据传入String对象，出来的就是可以看的懂的日期。

1. length()：long类型，返回文件的大小，但返回的是字节数。
2. listFile()：File数组(File[])类型，获取当前目录下的所有子文件。

例：File f = new File("F:/data");  
 File[] files = f.listFiles();  
 for(File file : files){  
 System.out.println(file.getAbsolutePath()); }

listFile用File[]接收后用增强for循环输出文件的绝对路径，这样就可以做到遍历子目录。

运用以上方法并结合File和FileInputStream、FileOutputStream可以实现目录的拷贝，具体实现再IDEA中有写（比较难，建议多看多学）。目录拷贝难的地方主要是思路逻辑上的，具体的实现并不难。

**IO&Properties的联合使用**

设计理念：

以后经常要改变的数据，可以单独写到一个文件中，使用程序动态读取。将来只要改变这个文件中的内容就可以直接读取，java代码不需要改变，不需要重新编译，服务器也不需要重启就可以拿到动态的信息。

类似这种机制的文件被称为**配置文件。**并且当这种配置文件中的内容格式是：key=value(“=”可以用“：”代替，但不建议用“：”。)等号“=”的左右两边不要加空格。这种格式的文件被称为**属性配置文件。**这种属性文件在java中建议用.properties结尾，但并不是必须的，txt文档也可以。所以这种以.Properties结尾的文件被称为属性配置文件。Properties类其实就是专门用来存放属性配置内容的一个类。

例：IO&Properties的测试

FileReader fr = new FileReader("userInfo");  
 Properties pt = new Properties();  
 pt.load(fr);  
 String username = pt.getProperty("username");  
 String password = pt.getProperty("password");  
 String userid = pt.getProperty("userid");  
 String birthdate = pt.getProperty("birthdate");

/\*

\*先将文件传入到内存中（用字节流输入流或者字符流输入流都可）

\*然后创建Properties集合

\*用Properties类的load方法将输入流中读取到的文件传入到Properties集合中

\*用getProperties(String key)方法通过key值找到对应的value值

\*之后输出即可

\*注意：这里的数据都是动态的，即使不重新编译，改变了文件中的内容，依旧可以动态读取到\*文件里面的数据

\*/

# 多线程

**进程和线程的区别**

进程是一个应用程序(可以理解为1个进程就是1个软件)。

线程是1个进程中的执行场景/执行单元。

一个进程可以启动多个线程。

对于java来说，当运行一个程序时，会先启动JVM虚拟机，而JVM其实就是一个进程，这个进程会启动至少两个线程并发。一个主线程调用执行main方法，另一个线程是垃圾回收线程，遵循垃圾回收机制专门处理java中的垃圾。

在java语言中，线程A和线程B的堆内存和方法区内存共享，但是栈内存独立，一个线程一个栈。进程相当于从应用软件，他们的内存是独立不共享的。

假设启动10个线程，就会有10个栈空间，栈之间互相独立互不干扰各自执行各自的程序，这就是多线程并发。java中的多线程机制就是为了提高程序的处理效率。

使用了多线程机制之后，main方法结束了，有可能程序并没有结束。main方法结束了只是主线程结束了。主栈空了，其他的栈（线程）可能还在弹栈压栈。

多线程并发的意思是t1线程执行t1的，t2线程执行t2的，两个线程互不干扰。在单核的CPU中（表示只有一个大脑），是不能真正做到多线程并发的，他在某一个时间点上实际上只能处理一件事情，但由于这个“大脑”的处理速度非常快，多个线程之间来回切换的速度非常快，给人的感觉就好像是多个线程在同时执行。

**实现线程有两种方法**

**第一种：编写一个类，直接继承java.lang.Thread类,重写run()方法**

当这个类继承了Thread，并重写了run()方法，那么只需要创建这个类对象，并启动他的start()方法就可以启动线程了。start()方法不需要重写，这是父类Thread里面的方法。

JVM虚拟机在执行到对象线程的start()方法时，这个方法就会压栈启动一个新的线程，在启动出这个线程之后这个start()方法就结束（弹栈）了，新的线程会自动调用类的run()方法并执行，而main主栈则会继续执行start()代码之后的方法，实现两个线程并发。

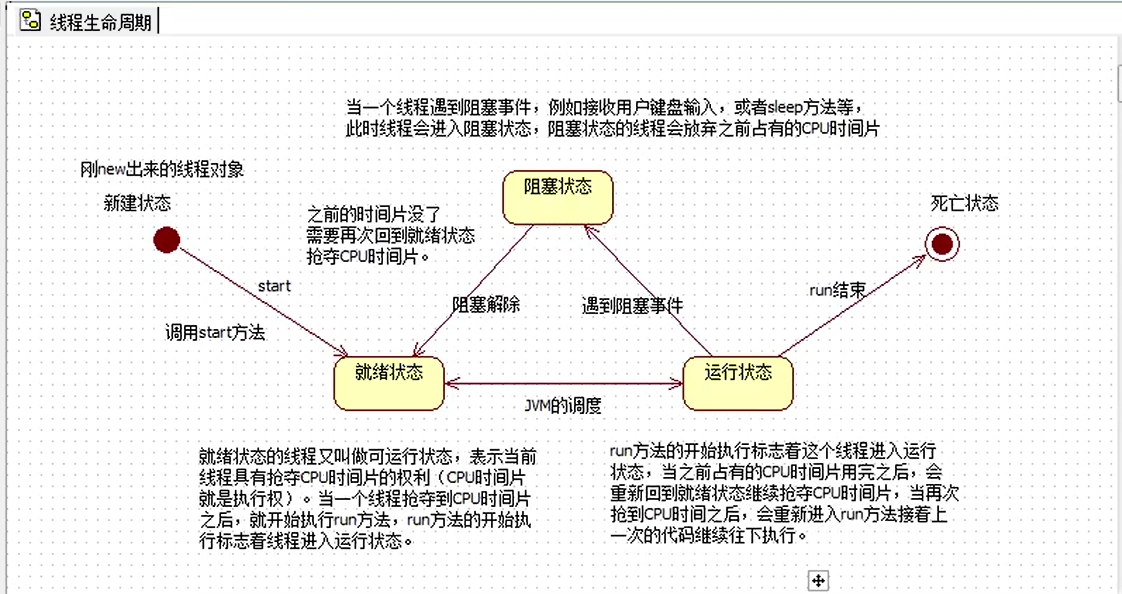
**第二种：编写一个类实现Runnable接口，实现run方法**

实现这个Runnable接口后，需要实现他的run方法，之后在main方法中创建一个Thread对象，将实现了Runnable接口的类new一个对象放入Thread对象的有参构造方法中，并调用这个Thread对象的start()方法即可启动这个线程，这种方式也可以实现多线程。

这两种方式虽然都可以实现多线程，但第一种（extends继承）Thread方式会让这个类无法再继承其他的类（java只支持单继承），而第二种（implements实现）Runnable的方式允许这个类去继承其他的类，可以使这个类更加灵活，并且我们编程是以面向接口编程为主，因此第二种用的比较多。

如果在main方法中直接调用线程对象的run()方法，那么这还是单线程，并不会自动启动另一个线程，这就相当于调用了一个类中的方法，要启动线程一定要用start()方法来启动。

**线程的生命周期**



线程有5种状态：新建状态、就绪状态、运行状态、死亡状态、阻塞状态

**新建状态**：线程对象刚被new出来时，被称为新建状态

**就绪状态**：线程对象调用start()方法时，线程就处于就绪状态（又称可运行状态）。这个状态表示当前线程具有抢夺CPU时间片的权力（执行权）。当一个线程抢夺到CPU时间片之后，就开始执行run()方法，run()方法的执行代表着线程进入运行状态。run方法不能抛出异常，因为他的父类没有抛出异常，根据异常定义，子类不能抛出比父类更多的异常这个规则，所以重写run方法时，不能抛出异常，只能try...catch捕捉。

**运行状态**：线程对象中的run()方法开始执行时，标志着线程进入运行状态，当这个线程将之前抢夺到的时间片用完时，线程会再次进入就绪状态，继续抢夺CPU时间片，当再次抢夺到CPU时间片时，run()方法会接着上次的代码继续往下执行。

运行状态下如果遇到了synchronized代码块，这个线程会进入**锁池（lockpool）,**在这里找共享对象的对象锁时，会释放之前占有的CPU时间片，有可能找到了，也可能没找到，没找到就在锁池里面等待，如果找到了就会进入就绪状态继续抢夺CPU时间片来执行这里面的代码。线程进入锁池也可以理解为一种阻塞状态。

**死亡状态**：当线程中的run()方法执行完毕之后，线程进入死亡状态。

**阻塞状态**：阻塞状态是当线程处于运行状态时遇到阻塞事件，例如接受用户键盘输入，或者**sleep()、join()**方法等，此时线程会进入阻塞状态，阻塞状态的线程会放弃之前所占有的CPU时间片，但其他线程仍然会继续执行，并不会影响，而本线程将阻塞事件解除后线程并不会立刻回到运行状态，而是回到就绪状态继续抢夺CPU时间片，但这次抢到之后不会重新执行run()方法，而是接着上次停下来的位置继续往下执行。

**线程的一些方法**

1. getName():String类型，返回线程的名称，如果没有设置，那么线程的名称都是Thread-0、Thread-1这种形式。用法：线程对象.getName()；
2. setName(String name):void类型，设置线程的名字。用法：线程对象.setName(“...”)。
3. currentThread()：Thread类型，Thread类的静态方法。直接用Thread就可以调用，返回的是当前线程的引用，且这个方法被谁调用就返回谁的引用。当这个方法在main中使用，并用getName()方法获取当前线程的名字时，会返回main。当Thread线程对象中使用了这个方法且main中用setName()方法设置了名字，那么在线程运行时，这个方法的获取的线程名就是设置的名字。

这个方法有便于在输出时分辨是哪个线程在运行，线程多了以后不同的线程有各自能区分的名字，此时这个方法就会让人很方便的看出是哪个线程在运行，而getName方法的调用者必须是Thread对象，而分支线程自己的run方法中无法创建自己的对象（强行创建会形成死循环），所以这个方法有发挥了作用。当然如果只是为了看到线程的名字super和this也可以实现，因为调用this和super的类刚好是Thread类，而如果在其他非线程类中，要知道那个线程在执行这个类中的方法，就只能用currentThread方法。

例：线程对象Thread01中的run方法

Thread t1 = Thread.currentThread();  
for (int i = 0; i < 100; i++) {  
 System.out.println(t1.getName()+"------>"+i);  
}

main方法中

Thread01 t1 = new Thread();

t1.setName(“T01”)

t1.start()

//这里执行时线程Thread01输出的就是“T01”这个线程名。

//T01 ----->1 ......

1. sleep(long millis)：void类型，Thread类的静态方法，让线程处于阻塞状态millis毫秒，经过millis毫秒之后再让这个线程恢复就绪状态抢夺CPU时间片。这个方法出现在哪个线程中，这个线程就会进入休眠。对for循环使用可以实现让代码每隔millis毫秒执行一次。

sleep方法是静态方法，和对象无关，所以即使用多态创建了线程对象，在main方法中用这个对象去调用sleep方法，这个线程对象也不会休眠，休眠的是main主线程。反正记住，sleep方法出现在那个线程中，哪个线程就休眠。

1. interrupt()：void类型，让处于睡眠状态的线程醒来，可以在mian方法中唤醒其他线程。这种中断睡眠的方式是依靠的异常捕捉一个正在睡眠的线程如果被唤醒，他会抛出捕捉到的异常，并输出异常信息，然后继续执行try...catch之后的语句。用法：线程对象.interrupt()

这个方法的使用场景可以是在一个有先后顺序的事件之间，需要先做完一件事之后，再继续另一件事。

1. stop()：void类型（已过时，不推荐使用）。强行终止线程。用法：线程对象.stop()。这种方法现在不使用了，因为这种方法就相当于任务管理器的终止进程，会强行退出线程，容易丢失数据，所以一般用其他方法代替。

现在终止线程的方法：给线程对象添加一个boolean类型的变量run，run的默认值为true，当run为true时就继续执行线程的程序，当run为false时，就用return终止线程，而在return之前也可以用程序将需要保存的数据保存下来。这种方法是比较好的方法，也是目前的主流方法。

**线程的调度**

**常见的线程调度方式有两种：**

抢占式调度模型：

那个线程的优先级较高，抢到的CPU时间片的概率就高一些/多一些。Java就是采用这种调度模型。

均分式调度模型：

平均分配CPU时间片，每个线程占有的CPU时间片时间长度相同，平均分配，有一些编程语言就是采用的这种方式。

**Java中关于线程调度的一些方法：**

1. setPriority(int i)：void类型，实例方法，设置线程的优先级，最高为10，最低为1，默认是5。优先级较高的线程获取的CPU时间片会多一些（但不完全是，大概率是），不是抢到的频率高。
2. getPriority()：int类型，实例方法，获取线程的优先级。
3. yield()：void类型，静态方法，让位方法，暂停正在执行的线程，并执行其他线程。注意yield方法不是阻塞方法，它是将处于“运行状态”的线程回到“就绪状态”。回到“就绪状态”有可能会再次抢到。
4. join()：void类型，实例方法。合并线程。让当前线程处于阻塞状态，并让调用这个方法的线程对象先执行，知道调用这个方法的线程执行结束，在执行当前线程。相当于将调用这个方法的线程加入到当前线程中，合并成一个线程，程序线性执行，所以要先执行调用这个方法的线程，这个线程结束之后才能继续执行当前线程。

**多线程并发环境下的数据安全问题（synchronized）**

这个是多线程这方面非常重要的问题。

以后开发中，项目是运行在服务器中的，服务器已经将线程的定义、创建、启动等都已经实现了，这些代码不需要我们写。但是，我们所写的程序放到一个多线程环境下运行，我们最需要关注的就是数据的安全问题。例如：在取银行账户中的钱时，有两个线程同时再取，而这两个线程同时读取到数据，对原数据取完钱后将数据传回银行账户时发生了延迟，两个人读取到的是原有的钱，取出来之后传回去的是2个线程对原有数据进行操作之后的数据，就会让两个人取出来的钱的总数超过账户里原本有的钱，数据就不安全了。

所以，多线程并发环境下，数据出现安全问题的情况主要有三个条件：

1. 多线程并发
2. 有共享数据
3. 共享数据有修改行为

这些条件都满足之后就可能出现线程安全问题。

如何解决线程安全问题呢？

线程排队执行（不能并发）。

用排队执行解决线程安全问题，这种机制被称为：线程排队机制，又叫线程同步。

线程同步会牺牲一部分效率。但数据安全最重要，所以不可避免。

1. **关于线程同步的两个专业术语：**

异步编程模型：

线程t1和t2各自执行各自的，互不干扰。其实就是多线程并发（效率较高）

同步编程模型：

线程t1和t2，在线程t1执行时，必须等待t2线程执行结束，或者说在t2线程执行时，必须等待t1线程执行结束，两个线程发生了等待关系，这就是同步编程模型（效率较低），线程排队执行。

1. **线程同步机制的语法：**

synchronized(){ } //线程同步代码块

使用synchronized代码块，将需要排队执行的代码放入代码块中。

Synchronized代码块最关键的是他的“( )”中需要填什么，填入这个小括号的数据必须是多线程共享的数据，只有这样才能达到多线程排队。

“( )”中具体写什么数据，主要看你想让那些线程同步。

假设有5个线程t1,t2,t3,t4,t5，而你只想让t1,t2,t3线程排队而t4,t5不需要排队，那么这里面填写的数据一定是t1,t2,t3线程共享而t4,t5线程不共享的对象。

假如是一个类中的自己的方法需要同步，那么“( )”里面也可以传this（this代表当前对象），甚至是共享对象中的任意一个成员变量类型的对象，都可以使这个类同步。

个人理解为：可以想象一个类是一个房间，而这个类的成员变量相当于是床，当只有一个房间且设置为线程同步(需要排队，一次只能进来一个)时，如果要占用这个床，就同时也要占用这个房间，所以当床为同步对象时，这个类也同步。

1. Java中的三大变量：实例变量（堆中）、静态变量（方法区中）、局部变量（栈中），只有局部变量是不会存在数据安全问题，因为栈不共享。
2. synchronized关键字也可以使用在实例方法上，使用在实例方法上的synchronized关键字表示同步这个方法，不需要传入同步对象，但默认是this。也就是说只要多个线程使用的是相同类的同步方法，这些线程就需要排队。所以这种方式并不灵活，而且整个方法都需要同步可能会无故扩大同步的范围，导致程序执行效率降低。但如果需要共享的对象就是this，且这个方法体确实需要同步，那么还是建议使用这种方式的。

**锁**

在java语言中，任何对象都有一把“锁”，其实这把“锁”就是一个标记(只是把它叫做锁)。

假设t1和t2线程并发，开始执行代码时，一定有一个先后顺序。

假设t1先执行了，遇到synchronzed关键字，这个时候线程就会自动进入锁池找后面“共享对象”的对象锁，找到之后就会占有这把“锁”，然后执行同步代码块中的程序，在程序执行过程中一直都是占有这把锁的。知道同步代码快执行结束，t1就会归还这把锁，此时t2等到了这把锁，然后t2就占有这把锁执行同步代码块。

这样就达到了线程排队执行，synchronized其实就是把这把“锁”唯一且共享了，只能给一个线程使用，这个线程使用完了才轮到下一个线程。

这里要注意的是，这个共享对象一定要选好了。这个共享对象一定是你所需要排队的线程对象所共享的对象。

个人总结：synchronized小括号中的同步对象，指的是线程对象传入的实参。

synchronized的三种写法：

第一种：同步代码块（灵活）

synchronized(线程共享对象){ //锁的是对象

同步代码块;

}

第二种：在实例方法上用synchronized

表示共享对象一定是this //锁的也是对象

并且同步的代码块是整个方法体。

第三种：在静态变量上使用synchronized

表示找类锁

同一个类永远只有一把类锁

就算有多个对象，也只有一把类锁

对象锁：每创建一个对象就有一把锁，100个对象100把对象锁

类锁：类专有的锁，100个对象可能也只有1把类锁

死锁

假如有2个线程，一个线程执行时锁住了A，A中锁住了B，另一个线程在执行时，先锁住了B，然后再锁A，如果两个线程同时发生，那么第一个线程执行到A时在等待B的锁，而第二个线程锁住了B，在等待A的锁，所以两个线程僵持住了，就会导致死锁。为了减少这种情况的发生，**synchronized最好不要嵌套使用。**

**守护线程**

Java语言中线程分为两大类：

用户线程：普通的线程都是用户线程

用户线程的代表线程：main

守护线程：后台程序。

守护线程的代表线程：垃圾回收器(gc)

守护线程的特点：

一般守护线程是一个死循环，所有的用户线程只要结束，守护线程也会自动结束。

守护线程的用途：

例：每天00：00系统数据自动备份

这个时候就需要定时器，并且我们要将定时器设置为守护线程。后台一直执行，每到时间就备份一次。如果所有的用户线程都结束了，守护线程自动退出，没有必要进行数据备份了。

守护线程的实现方式：

非常简单，再线程执行之前，使用setDeamon()方法（线程对象.setDeamon()）。

**定时器**

定时器的作用：

间隔特定的时间执行特定的任务。

例如：每周要进行的银行账户的总账操作。每天的数据备份操作...

在实际开发中，没隔多久进行一段特定的程序，这种需求很常见，java可以有多种实现方式：

1. 使用sleep方法，睡眠，设置睡眠时间，每到这个点醒来执行任务，这种方式是最原始的定时器（比较low）。
2. 在java类库中已经写好了一个定时器：java.util.Timer;可以直接拿来用，不过这种方式在目前的开发中也很少使用，因为现在有很多高级的框架都是支持定时任务的。在实际开发中，目前用的最多的就是SprintTask框架，这个框架只要进行简单的配置，就可以完成定时器的任务。TImer实现了Runnerable接口，所以实际上也是一个线程，但它不需要用start方法开启。

Timer方法的具体使用参见IDEA中的TimerTest包中的测试。

**实现进程的第三种方式(JDK8之后才有)**

实现Callable接口。他与之前两种线程的区别就是：这种方法创建出来的线程有返回值。创建FutureTask对象时，传入Callable类型对象的实参可以用匿名内部类表示，影响不大。但如果是在Thread传入FutureTask类型对象的实参就使用了匿名内部类，再将在FutureTask的实参部分的Callable类型对象也用匿名内部类/或者正常的类，这样虽然对于线程本身来说影响不大，但没有了FutureTask对象变量名就无法使用FutureTask对象的get()方法获取线程的返回值。

**使用方法：**

1. 在main方法种创建一个FutuerTask类对象，他需要传入一个对象类型为Callable的实参。
2. 创建一个类继承Callable接口（这个接口支持泛型）并实现他的call方法(这里的call方法和run方法一样，没什么区别)，但这个call方法有一个返回值，需要用return来返回，返回的类型取决于你的泛型，若是没写泛型就是Object类型。将这个类的对象new出来并传入FutureTask类构造方法的实参。
3. 创建一个Thread线程对象，将这个FutureTask对象传入Thread对象的实参，然后启动这个线程。
4. 线程的返回值用FutureTask对象的方法get()获取（ft.get()），返回类型用自己定义的泛型接收，若没有泛型就用Object类型对象接收。

**这种方式的优缺点：**

优点：这种方式确实可以获取到线程的返回值，就相当于获取到了线程的执行结果，而且Callable类对象的方法可以抛异常。

缺点：效率比较低，在一个要获取这个线程的返回值的线程种，这个线程（即当前线程）必须要等到获得有返回值线程的返回值之后程序才能继续执行，不然就一直处于“阻塞状态”，效率比较低。

**生产者和消费者模式(wait() 和 notify())**

wait()和notify()方法都不是线程对象的方法，是java中任何一个对象都有的方法，这两个方法是Object类中自带的

wait()方法的作用：

Object obj = new Object();

obj.wait();

让正在obj对象上的线程进入等待状态，无期限的等待，直到被唤醒为止。

obj.wait()方法的调用会让“当前线程（即活动在obj对象上的线程）”进入等待状态。

notify()方法的作用：

Object obj = new Object();

obj.notify();

唤醒正在obj对象上等待的线程。

此外还有一个notifyAll()方法，可以唤醒所有obj对象上处于等待状态的线程。

生产者消费者模式：

生产者和消费者模式是为了专门解决某个特定需求的。

假设有一个仓库o，有两个线程：生产者(Producer)和消费者(Consumer)

生产者将生产的产品放入仓库，若是仓库满了就不再生产，消费者消费商品，若是商品没有了就停止消费，最终必须要达到生产者和消费者的均衡。

仓库这个对象最终稿都会用到wait()和notify()方法，且仓库是多线程共享的，所以也要考虑到线程安全问题。

Wait()和notify()方法都是建立在synchronized线程同步的基础上的，

注意：o.wait()是让处于o对象上的线程进入等待状态，并且是释放之前占有当前对象的锁。o.notify()只会通知，并不会释放之前占有o对象的锁。

# 反射机制

反射机制的作用：

通过java语言中的反射机制可以操作字节码文件。类似于黑客（可以读和修改字节码文件），通过反射机制可以操作代码片段。（class文件）

反射机制的相关类的包：java.lang.reflect.\*;

反射机制相关的几个重要的类：

java.lang.Class：代表整个字节码，代表一个类型，代表整个类。

java.lang.reflect.Method：代表字节码中的方法字节码。代表类中的方法。

java.lang.reflect.Constructor：代表字节码中的构造方法字节码。代表类中的构造方法。

java.lang.reflect.Field：代表字节码中的属性字节码。代表类中的成员变量（静态变量和实例变量）

若是要对一个类的字节码文件进行读和修改，那么首先就应该获取这个类的字节码文件。

方法1：使用Class类的静态方法forName

forName(String s):返回Class类型，是Class类的静态方法，方法的参数是一个字符串，这个字符串必须是一个完整的类名，这个完整的类名必须含有包名，即使是java.lang包也不能省略。

例：Class c1 = Class.forName(“java.lang.String”);//此时c1就代表String.class文件，或者说c1代表String类。

方法2：使用Object类中的实例方法getClass();

getClass()：返回Class类型，是Object对象的方法，可以返回对象的类型文件字节码。例：String s= “abc”;

Class c = s.getClass();//这里c代表的是String.class字节码文件，代表了String类型。这种方式和第一种方法获取的String.class的地址是一样的，也就是说方法1中的Class类型变量c1 == 方法2中的Class类型变量c，因为字节码文件装在到JVM中只装载一份。

方法3：直接获取.class属性

在java语言中，任何一种类型，包括基本数据类型，他都有.class属性

例：Class c2 = String.class;//c2就代表了String类型

这里的c2和方法1中的c1和方法2中的对应的地址都是相同的。

获取到Class类的字节码文件之后就可以对Class类里面的方法、构造方法或者属性进行操作。

1. 通过Class的newInstance()方法来实例化对象。注意：newInstance()方法底层实际上是调用了无参构造方法，必须保证无参构造存在才行。

Class c = Class.forName(“User”);

Object obj = c.newInstance();

这样就创建了一个User的无参构造对象，并将地址赋给obj。但newInstance()这个方法从JDK9开始就不用了，是已过时的方法。

这种方法比正常的实例化对象灵活很多，为什么？

将这种创建对象的方式与IO流的属性配置文件Properties结合起来，可以从Propertirs中直接获取到forName()方法所需要的实参，之后通过newInstance()方法就可以实例化对象，而如果要实例化其它类型的对象，只需要修改Properties里面key对应的value值，而实例化代码一行都不需要改就可以完成实例化其它类型的对象。

**具体实现见IDEA的ReflectTest/ReflectANDProperties包下有测试。**

**总结**：反射机制更加灵活，可以访问Class文件，使程序更加灵活。其次有一些需要注意的点，如果只希望某一个类的静态代码块执行，其他代码一律不执行，那么用Class.forName()方法是最好的，因为这个方法会导致类加载，而静态代码块会在类加载的时候运行一次，且只运行一次，所以若是用这中方法获取类，那么既可以不创建对象，又可以直接获取静态代码块内容。

这里研究一下文件的路径问题。

上面那种获取文件路径的方法有一个弊端，就是他这个路径是在IDEA里面的相对路径，程序找文件时自动就从项目的根目录(src)下找，但若是换了个环境，或是换了个工具，没有盘符或者不是从项目的根目录开始寻找，那么文件很可能会找不到，因此我们可以将这种获取“相对路径”的方式转变为获取文件的**“绝对路径”**。

方法：Thread.currentThread().getContextClassLoad().getResource(“文件从根目录(src)开始的路径”).getPath();

解释：用当前线程（currentThread()）的类加载器（getContextClassLoad()），找到这个文件（getResource()）并获取他的路径(getPath)。

这种方法只能用在文件时在类路径下（就是文件必须在src下），因为类加载器是从类的根路径（src）下开始找的。

以后凡是牵扯到路径问题基本上都用这种方式，而如果需要将文件的内容放入流中进行读取或者修改，可以用以下方式：

**以流的方式直接将文件放入流中：**

Thread.currentThread().getContextClassLoad().getResourceAsStream(“相对路径”);

这种方式也可，但只能用InputStream类型接收。如是要放入Propertires集合中，就可以将这个InputStream类型对象直接放入。

**资源绑定器ResourceBundle**

ResourceBundle rb = ResourceBundle.getBundle(“类目录以下的文件名”);

资源绑定器ResourceBundle只能绑定“xxx.properties”文件(属性配置文件)，且这个文件必须在类路径下，文件扩展名也必须是properties。在写路径时，后面的扩展名不能写，从src文件下的包开始写路径一直写到xxx.properties，但properties不用加上。

这个类可以直接通过他的getString(String key)方法，通过key获取到对应的value值。

1. 获取了类之后，获取类的属性Field

方法：

1. getFileds()：Filed[]数组类型，获取类中所有public修饰的属性。可以用Filed对象方法getName()方法获取这个属性名。
2. getDeclaredFileds()：Filed[]数组类型，获取类中所有的属性(包括public\default\private\protected)。Filed数组中的Filed对象存储的是整个属性，包括修饰符，类型和变量名。

通过获取到的Filed对象可以用Field对象方法getTyper()来获取属性的类型，但这个getType()方法的返回值是Class，所以用Class类型对象接受后，仍然要使用Class类型的对象方法getName()来过去属性名称。

通过获取道德Field对象还可以使用Filed对象方法getModifiers()方法来获取修饰符，但这个方法返回的是修饰符代号，是int类型。但有一个Modifier类，这个类有一个toString(int i)方法，可以将之前获取道德修饰符代号作为实参传进去，就可以通过代号获取到对应的修饰符，并将修饰符转化为String类型，输出就是对应的修饰符。

当然getName()获取的是完整的类名，也可以用getSimpleName()方法来获取简单的类名。

**使用反射机制访问对象的属性：**

**方法：**

**若这个属性是public**

首先获取类文件（Class.forName(“路径”)）赋给Class类型变量

其次用这个Class变量的实例方法newInstance方法创建一个类对象并用Object类型对象接收。

然后用这个Class类的实例方法getDeclaredFied(String name)这个方法将对象属性名传入实参name，这个方法返回的是Field类型，所以用Field类型变量接收。

最后这个Field对象有一个实例方法set(Object obj,Object value),obj为这个对象，即用newInstance()方法创建出来的对象，value是想修改的属性的值，最后用Filed对象的get(对象obj)方法就可以获取属性的值，这个意思是获取obj对象的属性值（之前用Field接受了）。这样就完成了使用反射机制访问对象属性。

作用：这种方法虽然编码难度大，但它的灵活性很高，而且比较通用，传入的实际参数都可以放在配置文件中，不同的配置文件可以实现相同的操作且发挥不一样的作用。

底层框架都是这么写的，以后学习框架会非常常用。

**若这个属性是private（重要）**

步骤与上面一样，只不过在调用Field的set方法之前，先用Field的实例方法setAccessible(true);这样再插入数据时就不会出错。

但这也暴露了反射机制的缺点：打破封装。这可能会给不法分子留下机会！

1. 获取类的方法Method

可变长度参数：类型...（例：int...），传入0-N个同一类型的数据

一般用在方法的参数中

可变长度参数要求的参数个数是：0-N个

可变长度参数在参数列表中必须处于最后一个位置上,而且可变长度参数只能有1个

可变长度参数可以看作是一个数组，他有length属性。甚至传入数组也可。

测试在IDEA的ReflectTest包下的ArgTest01

**使用反射机制获取类的方法及其修饰符和返回值类型**

设已经获取了Class c = Class.forName(“User”);

//获取类的方法

Method[ ] methods = c.getDeclareMethods();

for(Method m : methods){

//获取方法的修饰符

String modifier = Modifier.toString(m.getModifiers());

//获取方法的返回值类型

Class cType = m.getReturnType();

String cTypeName = cType.getSimpleName();

//获取方法的名称

String mName = m.getName();

//获取方法的参数列表,首先参数列表返回的是参数类型，肯定是Class类型

//其次，参数可能有多个所以一定是一个数组

//最后根据作用推断方法名

Class[ ] classParameters = m.getParameterType();

for(Class classParameter : classParameters){

String classParameterName = classParameter.getSimpleName();

}

}

以上方法已在IDEA的ReflectTest包下的ReflectTest03中测试

**反编译Method（方法）**

通过反射机制打印出某个类的所有方法

具体测试在IDEA的ReflectTest包下的ReflectTest04中测试

**通过反射机制调用方法（很重要）**

首先获取Class类赋给Class类型Class c = Class.forName(“Reflect.User”)

然后创建这个类的对象Object obj = c.newInstance()

然后获取这个c的方法c.getDeclareMethod(c,String.class,int.class);

getDeclareMethod()方法返回的是Method类型，调用方法的办法：对象+方法名+实参+返回值，获取一个方法需要对象（c）+方法名(getDeclareMethod)+实参（String.class,int.class）所以就获取了对应的对象方法

**Method m = c.getDeclareMethod(“doSome”,String.class,int.class);**

**对应：返回的类型 获取方法 <通过> 方法名 方法的实参列表类型**

而调用方法需要使用invoke()方法，是Method方法调用的方法,返回方法的返回类型

用法：**Object obj = m.invoke(obj,”mcl”,11);**

**对应： 返回值 方法 对象 实参**

1. 获取类的构造方法

方法基本上和上面差不多，不在细讲。

通过反射机制获取(拼接)构造方法

通过反射机制使用构造方法创建对象(比上面这种方法重要)

**以上两种使用方法在IDEA的ReflectTest包下的ReflectTest06中都有测试**

1. 获取类继承的父类和实现的接口

首先获取类：Class c = Class.forName(“java.lang.String”);

获取父类：Class superC = c. getSuperClass();

获取实现的接口：Class[ ] cInterfaces = c.getInterfaces();

具体在IDEA的ReflectTest包下的ReflectTest07中有测试使用。

# 注解

注解，又被称为注释类型，英文单词是Annotation

注解Annotation是一种引用数据类型，编译之后也是产生xxx.class文件

如何自定义注解？

[修饰符列表] @interface 注解类型名{

//

}

注解如何使用，用在什么地方?

注解使用时的语法：@注解类型名

注解可以用在几乎任何地方：类、属性、方法、变量甚至注解等都可以使用。

修饰注解的注解被称为元注解。常见的元注解有：Target和Retention

**Override注解**

源代码:public @interface Override{

}

标识性注解，只是给编译器做参考。编译器在看到方法上有这个注解时，编译器会自动检查该方法是否重写了弗雷德方法。如果没有重写就报错。

注解只在编译阶段起作用，和运行期无关。

凡是java中的方法带有这个@Override注解的，编译器都会进行编译检查，如果这个方法不是 重写父类的方法，编译器报错。

@Override这个注解只能注解方法。且这个注解时给编译器做参考的，和运行阶段没关系。

**元注解**

用来就是注解的注解，就是元注解。

常见的元注解：Target和Retention；

Target注解：

这是一个元注解，用来标注“注解类型”的“注解”

这个Target注解用来标注“被标注的注解”可以出现在哪些位置上。

@Target(ElementType.METHOD):表示“被标注的注解”只能出现在方法上。

Retention注解：

这是一个元注解，用来标注“注解类型”的“注解”

这个Retention注解用来标注“被标注的注解”最终保存在哪里。

@Retenton(RetentionPolicy.SOURCE):表示该注解只保留在java源文件当中。

@Retenton(RetentionPolicy.CLASS):表示该注解只保留在CLASS文件当中。

@Retenton(RetentionPolicy.SOURCE):表示该注解被保存在CLASS文件中，且可以被反射机制所读取。

**Deprecated注解**

这个注解标识的元素，说明这个元素已经过时。

主要是向其他程序员传达一个信息，告知已过时，有其他更好的解决方案存在。

用法：在过时的元素（方法、类等）前加@Deprecated

**自定义注解**

格式：[修饰符列表] @interface 注解类型名{ }

例：public @interface myAnnotation{ }

若是注解中有属性，则这个属性的格式为：

类型 类型名();//例：String name();

然后在使用这个注解时，需要给类型名赋类型值：

例：@myAnnotation(name = “xxx”);

若有多个属性中间用逗号隔开。

若是注解中的属性有了默认值：

例：String name() default “xxx”;

那么在使用这个注解时可以不赋值，直接使用。但若是要赋默认值则一定要按以上这种方式，其他一律报错。

**注意**：若是属性名字就叫**value**，且只有value这一个属性，属性类型可以是系统允许的任意一种类型，那么再调用这个注解并给这个属性赋值时可以**直接写值**，不用按照之前赋值的格式。(测试JDK14即使不叫value，在只有一个变量的情况下，也可以直接写值，而且若是有多个属性，并不一定要全部赋值，运行时仍然没有错误，可能是JDK14更新过)

注解中属性的类型可以是：

Byte short int long float double boolean char String Class enum(枚举)

以及以上任意类型的数组形式。数组形式用：数组名“=”大括号（“{}”），将元素填入大括号中，中间用逗号（“，”）隔开。若是只有一个数组，可以不加大括号，直接就“=”元素。

**通过反射机制获取注解**

**这种注解**

首先获取类Class c = Class.forName(“类地址”)；

且有一个自定义注解myAnnotation

几个类Class的方法：

1. isAnnotationPresent(注解.class)：boolean类型，判断这个类是否被注解修饰。

用法：c.isAnnotationPresent(myAnnotation.class);

1. getAnnotation(注解.class)：返回注解类型，获取类的注解

用法：(myAnnotation)c.getAnnotation(myAnnotation.class)注意获取了这个注解之后要进行强转才能被myAnnotation对象接收。

注解类型也有一个方法：value()【并不一定时value，若是其他名字就是其他的方法】

value()：返回注解中的属性类型，若是没有额外赋值则返回默认值，若再用这个注解时赋了默认值，就返回赋的值。不能没有返回值，若是没有默认值而不赋值编译器会报错。

一些注意：通过反射机制获取注解以及注解的属性值时，只能通过具体的对象获取。例如：类的注解就调用类变量的获取方法（c.getAnnotation(???)）,若是属性(Field)就用属性的获取方法（f.getAnnotation(???)）......。不能通过其他的类型获取于此类型不同的注解。

其次，若是没有指定Target元注解，那么这个注解可以用在任何地方，但若是没有指定Rentation元注解，那么他的默认Rentation是Class类型(即生成Class文件并存放在src的目录下，但无法被反射机制获取到)，反射机制无法获取这个注解，只能将Rentation设置成RUNTIME类型，才可以被反射机制获取。

# 学完javase之后还需要学数据库和JDBC以及需要了解java框架等考完研之后继续学习。