1. 数据类型：



float：数据类型不精确，java中默认3.14为double类型，

eg：float a = 3.14F；//加上F才表示3.14为float类型

**注意：**不要使用float数据进行比较，需要使用BIGDecimal类进行比较。

eg：

**import** java.math. BigDecimal;

**public** **class** test {

**int** members;

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

BigDecimal bd = BigDecimal.*valueOf*(1);

bd = bd.subtract(BigDecimal.*valueOf*(0.1));

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

BigDecimal bd1 = BigDecimal.*valueOf*(0.10);

BigDecimal bd2 = BigDecimal.*valueOf*(1/10.0);

System.***err***.println(bd1.equals(bd2));

}

}

输出：

0.9

true

1. 布尔类型：boolean ：
2. 取模、取余运算：

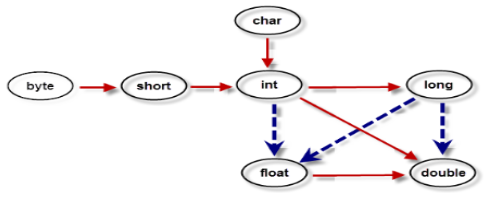
余数符号和左边操作数相同，eg：-7%3=-1；7%-3=1

1. 自动类型转换：

容量小的数据可自动转换为容量大的数据类型。

**实线：**无数据丢失的自动类型转换；（下图中）

**虚线：**在转换时可能会有精度的损失。



1. 输入输出：

**import** java.util.Scanner;

System.***out***.println("Input age");

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("\*\*\*\*\*INPUT DATA\*\*\*\*\*");

**int** age = scanner.nextInt();

System.***out***.println("\*\*\*\*\*OUTPUT DATA\*\*\*\*\*");

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

1. 方法：
2. 方法：就是一段用来完成特定功能的代码片段，类似于其它语言的函数。

eg：

/\*\* 求和的方法 \*/

**public** **static** **int** add(**int** n1, **int** n2) {

**int** sum = n1 + n2;

**return** sum;//使用return返回计算的结果

}

2）方法的重载：是指一个类中可以定义多个方法名相同，但参数不同的方法。 调用时，会根据不同的参数自动匹配对应的方法。

eg：

/\*\* 求和的方法 \*/

**public** **static** **int** add(**int** n1, **int** n2) {

**int** sum = n1 + n2;

**return** sum;

}

// 方法名相同，参数个数不同，构成重载

**public** **static** **int** add(**int** n1, **int** n2, **int** n3) {

**int** sum = n1 + n2 + n3;

**return** sum;

}

1. 递归：

任何能用递归解决的问题也能使用迭代解决。在要求高性能的情况下尽量避免使用递归，**递归调用既花时间又耗内存**。

1. 类：中的变量！

**public** **class** TestVariable {

**int** a; //成员变量, 从属于对象； 成员变量会自动被初始化

**static** **int** *size*; //静态变量，从属于类

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

{

**int** age; //局部变量，从属于语句块；

age = 18;

}

**int** salary = 3000; //局部变量，从属于方法

}

}

1. 类的定义：

**public** **class** SxtStu {

//属性（成员变量）

**int** id;

String sname;

**int** age;

//方法

**void** study(){

System.***out***.println("我正在学习！");

}

//构造方法：系统自动创建，不需要写明该方法，**该方法与类名相同**

SxtStu() {

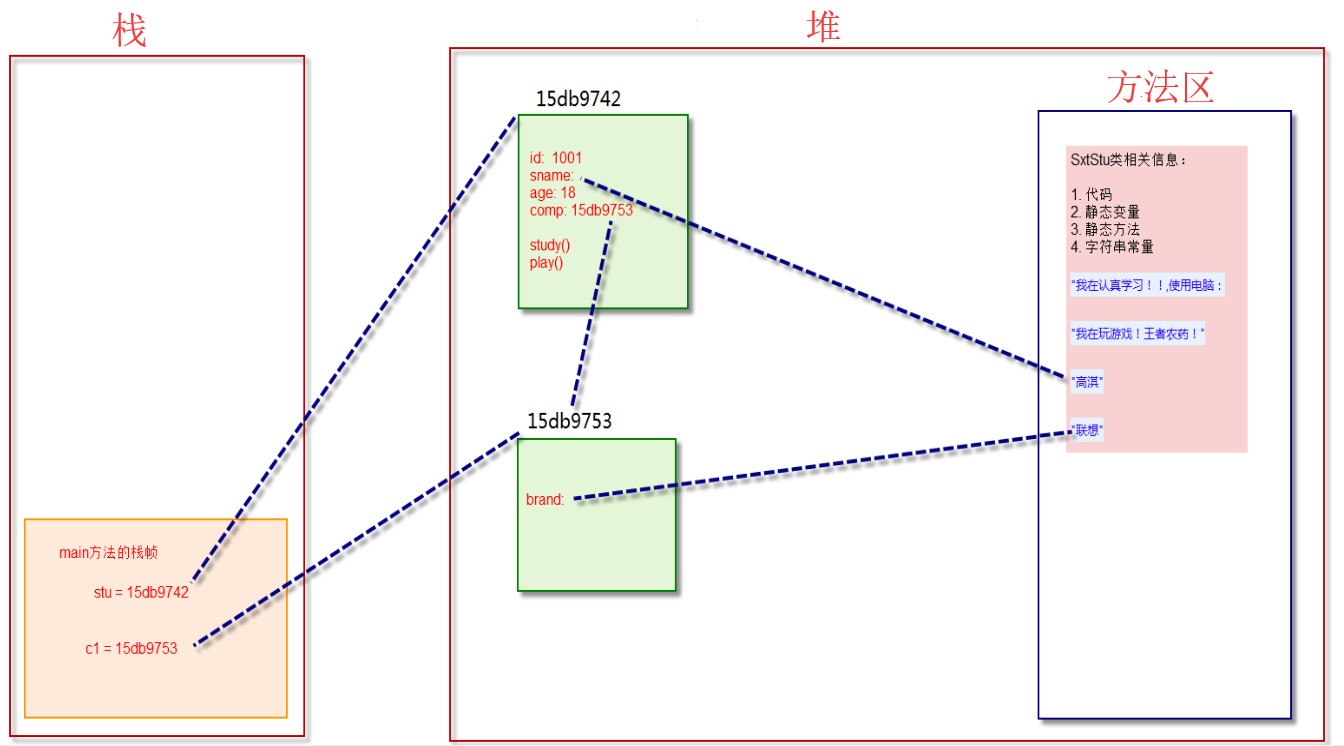
}

}

1. 内存分析：

java虚拟机的内存分为3个区域：堆heap、栈stack、方法区method area

1. **栈：**
2. 栈描述的是方法执行的内存模型，每一个方法被调用都会创建一个栈帧，用于**存储局部变量、操作数、方法出口**等等。
3. JVM为每个线程创建一个栈，存放该线程执行方法的信息（实际参数、局部变量等）
4. 栈属于线程私有，不能共享。
5. 栈有系统自动分配，速度快，连续的存储空间。
6. **堆：**
7. 堆用于**存储**创建好的**对象和数组**
8. **JVM只有一个堆**
9. 堆是不连续的内存空间，分配灵活，速度慢
10. **方法区：（特殊的堆）**
11. **JVM只有一个方法区**，被所有线程共享
12. 方法区实际上也是堆，只是用于运行时的**存储类、常量**等信息
13. 存放程序中的不变内容：**class对象、静态变量、方法、字符串常量**



1. 构造方法的使用：

**public** **class** Construct {

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

Point a = **new** Point(3.0, 4.0);

Point orign = **new** Point(0.0, 0.0);

System.***out***.println("dis = " + a.getDistance(orign));

}

}

**class** Point{

**double** x,y;

//构造方法：名字必须和类名相同

**public** Point(**double** \_x, **double** \_y) {

x = \_x;

y = \_y;

}

**public** **double** getDistance(Point p) {

**return** Math.*sqrt*((x - p.x)\*(x - p.x) + (y - p.y)\*(y - p.y));

}

}

1. **this的应用：**

**this本质：**当前对象的地址

**（1）**若方法构造中形参名与属性名相同时，需要**使用this关键字区分属性与形参**。

**public** **class** User {

**int** id; // id

**public** User(**int** id, String name) {

**super**();**//super:可用于调用离自己最近的父类的protected方法**

//this.id：表示属性id；id：表示形参id

**this**.id = id;

}

}

**（2）this的使用：this不能用于static方法中**

**public** **class** TestThis {

**int** a, b, c;

TestThis(**int** a, **int** b) {

**this**.a = a;

**this**.b = b;

}

TestThis(**int** a, **int** b, **int** c) {

**// 调用带参的构造方法，调用上面的TestThis（），并且必须位于第一行！**

**this**(a, b);

**this**.c = c;

}

}

1. **super()：**作用：**子类可以通过它调用父类的构造函数**。

当构造函数有参数时，**必须使用super，且放在第一行**，否则会报错。

当构造函数无参数时，可以省略super()

1. 垃圾回收机制：
2. 引用计数

堆中每个对象都有一个引用计数。被引用一次，计数加1. 被引用变量值变为null，则计数减1，直到计数为0，则表示变成无用对象。优点是算法简单，缺点是“循环引用的无用对象”无法别识别。

1. 引用可达法（根搜索算法，常用）

程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。

**（3）垃圾回收机制：原理：**

不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。我们将对象分为三种状态：年轻代、年老代、持久代。JVM将堆内存划分为 Eden、Survivor、enured/Old 空间。

**（4）清理区域分类：**

* Minor GC:用于清理年轻代区域。
* Major GC：用于清理老年代区域。
* Full GC：用于清理年轻代、年老代区域

1. **static**

**static的作用：方便在未创建对象时，调用方法、变量；**

**static不会影响方法、变量的访问权限。**

* **静态方法：**public **static** void method()。
* 静态方法不依赖任何对象，因此没有this；
* 静态方法中，不能调用其他非静态方法、成员（变量），因为非静态方法、成员都需要依赖具体对象才能够使用；
* 非静态方法中，可以调用静态方法、成员；
* **由于static方法可以在不创建对象时使用，因此main方法是static**;
* 调用：同一个类中：直接使用方法名；

不同类中：类名.方法名；

* **静态变量（少用）：**public **static** int para;
* **静态变量**，属于类，所有的对象都可以使用，**仅在加载类时被初始化**，**内存中仅有一个副本**；
* **非静态变量**，在实例化后可使用，**创建对象时被初始化**，**存在多个副本，副本之间不会相互影响**；
* 不存在静态局部变量。
* **非静态方法、变量：属于实例化对象，需要创建对象才能使用。**

**eg：static的使用示例**

public class BubbleStort {  
  **// 静态变量：少用，在多个实例化时，会改变之前实例化对象的静态变量，仅保留最后一次实例化对象的该静态变量值——因为静态变量在内存中仅有一个副本。**  
 private **static** int para；  
 public **static** void main(String[] args) {  
 int[] array = {1,20,3,4,1,2,5,0,11};  
 *Sort*(array);  
 }  
  **// 静态方法**  
 public **static** void Sort(int[] values){ }  
}  
**// 静态内部类**  
**static** class test{  
}

* **静态代码块：**
* **作用：**用于**初始化类**，在类调用时执行（**仅执行一次**）

（**构造函数**：用于**初始化对象**，可执行多次）

（**构造代码块**：给对象使用，也可执行多次）

* **执行顺序：**静态代码块 -> 构造代码块 -> 构造函数
* **示例代码：**

public class Hello {

**//构造函数**  
 public Hello(){  
 System.*out*.println("A的构造函数");  
 }

**//构造代码块**  
 {   
 System.*out*.println("A的构造代码块");  
 }

**//静态代码块**  
 static {   
 System.*out*.println("A的静态代码块");  
 }  
 public static void main(String[] args) {  
 Hello A = new Hello();  
 }  
}

**结果：**

A的静态代码块

A的构造代码块

A的构造函数

1. 包：
2. 包名：  
   域名倒着写即可，再加上模块名，便于内部管理类；

eg：

com.sun.test;

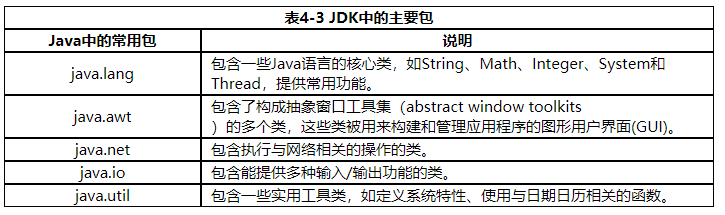
com.oracle.test;

注意事项：

com.gao和com.gao.car，这两个包没有包含关系，是两个完全独立的包。只是逻辑上看起来后者是前者的一部分。

这只是文件夹的包含关系：com包含gao，gao包含car

1. import：
2. 常用的包：



1. Java会默认导入java.lang包下所有的类，这些类可以直接使用。
2. **如果导入两个同名的类，只能用包名+类名来显示调用相关类：**

**eg：**

**import** java.sql.Date;

**import** java.util.\*;//导入该包下所有的类。会降低编译速度，但不会降低运行速度。

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

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

//这里指的是java.sql.Date

Date now;

//java.util.Date因为和java.sql.Date类同名，需要完整路径

java.util.Date now2 = **new** java.util.Date();

System.***out***.println(now2);

//java.util包的非同名类不需要完整路径

Scanner input = **new** Scanner(System.***in***);

}

}

**注意：**Scanner中**nextInt、nextLine**区别

1.nextInt()只读取数值，剩下"\n"没有读取，并将光标放在本行中。

2.nextLine()会读取"\n"，并结束

1. 静态导入：

**作用：**用于导入指定类的静态属性，就可以直接使用静态属性。

//以下两种静态导入的方式二选一即可

**import** **static** java.lang.Math.\*;//导入Math类的所有静态属性

**import** **static** java.lang.Math.***PI***;//导入Math类的PI属性

**public** **class** Test2{

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

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

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

}

}

1. **继承：Object类是所有Java类的根基类**，所有对象都拥有Object类的属性、方法。
2. **Java中类没有多继承，接口有多继承。**
3. **子类继承父类**，可以得到父类的全部属性和方法 (除了**父类的构造方法**)，**无法访问父类的私有属性、方法**。
4. **如果定义一个类时，没有调用extends，则它的父类是：java.lang.Object。**
5. 使用：

**public** **class** TestExtend {

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

Student s = **new** Student("Chris", 24, " engineering");

s.rest();

s.study();

//返回Boolean值

System.***out***.println(s **instanceof** People);

}

}

**class** People{

String name;

**int** age;

**public** **void** rest(){

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

}

}

**class** Student **extends** People{

String major;

**public** **void** study() {

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

}

**public** Student (String name, **int** age, String major) {

**this**.name = name;

**this**.age = age;

**this**.major = major;

}

}

1. instanceof：判断对象是否继承于该类。（或者该类的实例化）

**左边是对象，右边是类**；当对象是右面类或子类所创建对象时，返回true；否则，返回false.

**eg:**

//返回Boolean值

System.***out***.println(s **instanceof** People);

1. **方法重写：**

方法重写有下面几个注意事项：  
      1.“==”：方法名、形参列表相同。

    2.“≤”：返回值类型和声明异常类型，子类小于等于父类。

    3.“≥”：访问权限，子类大于等于父类。

1. toString：

下面两者等价：都是调用系统的方法**toString**打印出**对象的地址**

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

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

1. **final：**

1）**修饰变量**，则变量**不能重新赋值**；

final  int   MAX\_SPEED = 120;

2）**修饰方法**：该方法**不能被重写**；

final  void  study(){}

3）**修饰类**：该类**不能被继承**；

final   class  A {}

1. 数组：
2. 数组一旦被创建，就无法改变长度；
3. 数组可以看成对象，元素看成成员变量，java的对象存储在堆中；
4. **定义：**

一维：int[] a; or int a[];

二维：float [][]f = new float[6][6]; or float f[][] = new float[6][];

**分配空间：**

a = new int[10];

**初始化：**

1. **静态初始化：**

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

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

1. **动态初始化**

int[] a = new int[2];

a[0] = 1;

a[1] = 2;

1. 数组遍历：

**for-each循环**：**只用于读取数组，不能修改数据**

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

**int**[] b = **new** **int**[3];//静态初始化之后，默认值=0

**for**(**int** a : b){

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

}

}

1. 数组复制：System.*arraycopy*

//将s复制到s\_copy：

String[] s = {"aa","bb","cc"};

String[] s\_copy = **new** String[6];

System.*arraycopy*(s, 0, s\_copy, 0, s.length);

**for**(String a : s\_copy){

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

1. java.util.Arrays类：包含了常用的数组操作，包含了：排序、查找、填充、打印内容等常见的操作。
2. 打印数组：Arrays.toString(a); //这边的toString和Object的toString不同。
3. 元素排序：Array.sort(a); //可用于二分法查找

eg:

**import** java.util.Arrays;

**public** **class** binarySearch {

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

**int**[] a = {1,2,323,23,543,12,59};

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

//使用二分法查找，必须先对数组进行排序;

Arrays.*sort*(a);

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

//返回排序后新的索引位置,若未找到返回负数。

System.***out***.println("该元素的索引："+Arrays.*binarySearch*(a, 12));

}

}

1. **java.util.Arrays工具包**使用：
2. **打印**：Arrays.toString(a)
3. **数组排序**：Arrays.sort(a);
4. **二分法查找**（先排序）：Arrays.binarySearch(a, 12)
5. **填充**:Arrays.fill(a, 2, 4, 100);  //将[2,4）索引的元素替换为100;
6. 用二维数组保存表格：

**import** java.util.Arrays;

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

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

**//object不是对象，编译器将数据自动封装成对象，再将a1,a2,a3放到二维数组中**

Object[] a1 = {1001,"高淇",18,"讲师","2006-2-14"};

Object[] a2 = {1002,"高小七",19,"助教","2007-10-10"};

Object[] a3 = {1003,"高小琴",20,"班主任","2008-5-5"};

Object[][] emps = **new** Object[3][];

emps[0] = a1;

emps[1] = a2;

emps[2] = a3;

System.***out***.println(Arrays.*toString*(emps[0]));

System.***out***.println(Arrays.*toString*(emps[1]));

System.***out***.println(Arrays.*toString*(emps[2]));

}

}

1. **Comparable**：

用于数组为引用类型的排序，即：数组中含有标签

eg：

**import** java.util.Arrays;

**public** **class** bbb {

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

Man[] msMans = { **new** Man(3, "a"), **new** Man(60, "b"), **new** Man(2, "c") };

Arrays.*sort*(msMans); //根据索引数值（age）大小来排序

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

}

}

**class** Man **implements** Comparable {

**int** age;

**int** id;

String name;

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

**super**();

**this**.age = age;

**this**.name = name;

}

}

1. **访问权限修饰符：（变量、方法属性）**

1. private： 表示私有，只有在类中能访问，实例化后无法被访问.

      2. default：表示没有修饰符修饰，只有同一个包的类能访问

      3. protected：表示可以被同一个包的类以及其他包中的子类访问

      4. public：表示可以被该项目的所有包中的所有类访问

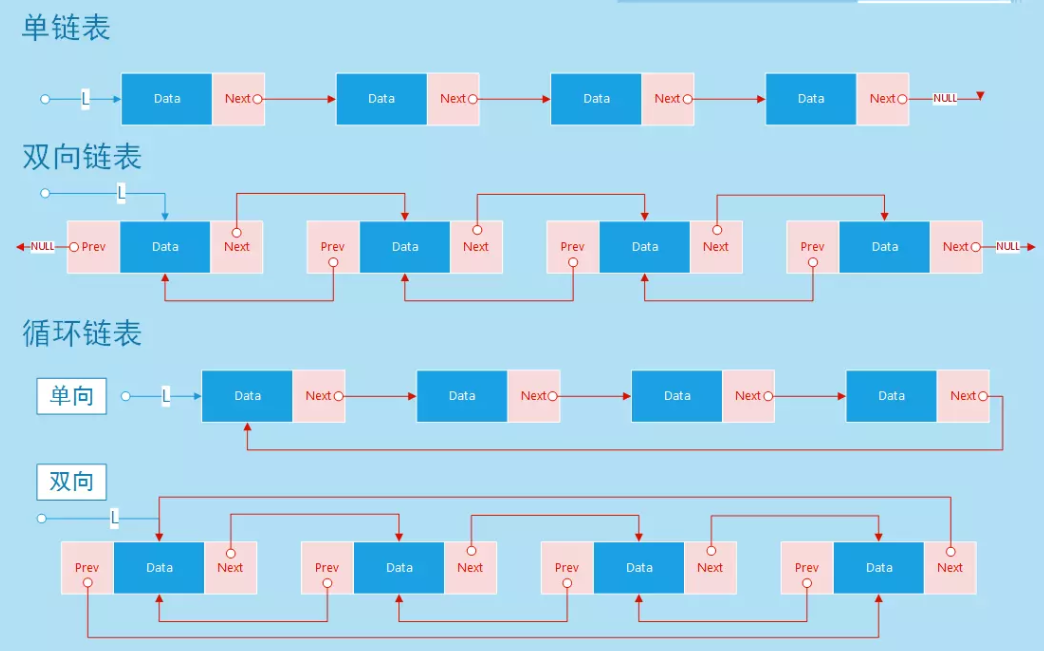
1. **链表：**
2. Item：

用在泛型数据：也就是占位符，可以匹配任意类型得数据

**泛型的本质**：数据类型的参数化（在调用泛型时必须传入实际参数）

1. 链表的分类：

* 单链表
* 双向链表
* 循环链表



1. 使用Java实现双向链表——LinkedList

手写链表的内容：节点类、双向链表类

**//节点类**

public class Node {  
 Node previous; **// 上一个节点，用于指向前一个结点**  
 Node next;  **// 下一个节点，用于指向下一个节点**  
 Object element; **// 本节点的数据**

**//实际上，这个构造函数并未使用**  
 public Node(Node previous, Node next, Object element){  
 this.previous = previous;  
 this.next = next;  
 this.element = element;  
 }  
 **//创建新节点时使用**  
 public Node(Object element){  
 this.element = element;  
 }  
}

**//双向链表类:内存结构:一个链表头、Data数据、一个链表尾**

public class LinkedListTest {  
 private Node first; **//链表头**  
 private Node last; **//链表尾**  
 private int size; **//链表节点个数**

**// 添加节点的操作**  
 public void add(Object obj){  
 Node newNode = new Node(obj);**//创建新节点**  
 if(first == null){ **//第一次添加节点，first为null**  
 first = newNode; **//第一次添加时，first和last指向相同的对象**  
 last = newNode;  
 }  
 else{  
  **//新节点的上一个节点-->last**  
 newNode.previous = last;  
  **//新节点的下一个节点-->null，因为链表尾的下一个节点为null（空）**  
 newNode.next = null;  
  **//将前一个链表的next指向新加入的节点（此时last节点不是链表尾）**  
 last.next = newNode;  
  **//last节点 = 新节点，此时新节点为链表尾，方便后续使用last操作**  
 last = newNode;  
 }  
 }

**//显示链表中的内容**  
 public void printList(){

**//注意StringBuilder的使用**  
 StringBuilder stringBuilder = new StringBuilder("[");  
 Node temp = first;  
 while (temp != null){  
 stringBuilder.append(temp.element + ",");  
 temp = temp.next;  
 }  
 stringBuilder.setCharAt(stringBuilder.length() - 1, ']');  
 System.*out*.println(stringBuilder);  
 }  
 public static void main(String[] args) {  
 LinkedListTest list = new LinkedListTest();  
 list.add("a");  
 list.add("b");  
 list.add("c");  
 list.printList();  
 }  
}

1. **抽象**
2. **抽象类：**public abstract class a{ }

**抽象类的用处：**

1. 为子类提供规范的模板，抽象类不能使用new（不能实例化）；
2. 抽象类只能用来继承。
3. **抽象方法：**abstract public void run();

**抽象方法的用处：**

1、没有实现；

2、子类必须实现（子类继承之后，必须再子类里面填写方法的具体内容）。

1. **接口：使用interface定义**
2. 定义：

public **interface** a{

void test();

}

1. 接口中定义的**全部都是抽象的方法**（也就是**定义规范**，没有实现）

这使得，程序比较稳定。

1. 接口中的**常量**：**自动修饰为public static final**：

eg：

int Fly = 100;

等同于：

public static final int Fly = 100；

1. 接口中的方法都是：abstract

eg：

public void test（）；

等同于：

public abstract void test（）；

**（注意：接口中不存在私有属性，仅有public和abstract）**

1. **关于extends和implements的区别：**
2. **extends**：主要用于**继承类**，只能继承一个父类（**类没有多继承**）
3. **implements**：用于**继承接口**，可实现多个接口（**接口有多继承**）  
   eg：class A implements B，C{ } //**interface B { }**
4. 示例代码：——interface的应用

**public** **class** MyInterface {

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

Angel myAngel = **new** Angel();

myAngel.fly();

myAngel.helpOthers();

System.***out***.println(myAngel.***FLY\_HIGH***);

}

}

**interface** Volant{

**int** ***FLY\_HIGH*** = 100;

**void** fly();

}

**interface** Honest{

**void** helpOthers();

}

// Angel 继承于Volant、Honest两个接口

**class** Angel **implements** Volant, Honest{

@Override

**public** **void** helpOthers() {

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

System.***out***.println("help others...");

}

@Override

**public** **void** fly() {

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

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

}

}

1. **内部类：**

内部类在编译后，会出现一个单独的.class文件，所以内部类是相对独立的存在。**作用:**

1. 内部类提供封装，**只允许外部类直接访问，不允许同一包内的其他类访问**。
2. 内部类可以访问外部类的私有属性，但外部类不能访问内部类的属性。

/\*\*外部类Outer\*/

**public** **class** InnerClass {

**private** **int** age = 10;

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

System.***out***.println(age);//10

}

/\*\*内部类Inner\*/

**public** **class** Inner {

//内部类中可以声明与外部类同名的属性与方法

**private** **int** age = 20;

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

System.***out***.println(age);//20

}

}

}

1. 非静态内部类：

非静态内部类，不能有静态方法、静态属性、静态初始化块。

**public** **class** InnerClass {

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

Outer.Inner test = **new** Outer().**new** Inner();

//这两种方式等价：inner 等同于test

Outer insOuter = **new** Outer();

Outer.Inner inner = insOuter.**new** Inner();

test.show();

}

}

/\*\*外部类Outer\*/

**class** Outer{

**private** **int** age = 10;

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

System.***out***.println(age);//10

}

/\*\*非静态内部类Inner\*/

**public** **class** Inner {

//内部类中可以声明与外部类同名的属性与方法

**private** **int** age = 20;

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

**int** age = 30;

System.***out***.println("内部类方法的局部变量"+age);//30

System.***out***.println("内部类成员变量" + **this**.age);//20

System.***out***.println("外部类成员变量" + Outer.**this**.age);//10

}

}

}

1. **静态内部类：**

1）使用static class Inner{}定义；

2）静态内部类不能访问外部类的实例化方法；

3）静态内部类可以看成外部类的一个静态成员，外部类可以使用“静态内部类.名字”访问静态内部类ide静态成员；

4）**不调用外部类时，静态内部类不会被调用，适用于调用次数少的类。**

eg：

**public** **class** InnerClass {

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

Outer.staticInner staticInner = **new** Outer.staticInner();

staticInner.show();

}

}

/\*\*外部类Outer\*/

**class** Outer{

**private** **int** age = 10;

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

System.***out***.println(age);//10

}

/\*\*静态内部类Inner\*/

**static** **class** staticInner{

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

System.***out***.println("show: static class ...");

}

}

}

1. 匿名内部类：

用于只是用一次的类：eg：键盘监听

1. string类：

**int** a = 10;

String string = "18" + a;

System.***out***.println(string); //string = 1810

**注意：==和equals()的区别：**

1）**==：**判两个变量指向的**对象是否相同**；

2）**equals()**：判断两个变量的**内容是否相同**

1. **常量池：**

用于存放初始化后的常量数据，eg：int a = 10；

**注意：**使用new String();初始化后的数据存放在与常量池不同的地方，因此这两个变量所谓指向对象是不同的。

eg：

String string2 = "1810";

String string3 = **new** String("1810");

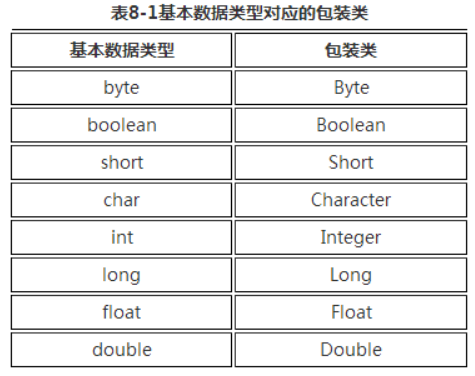
System.***out***.println(string2 == string3); //false



# 常用类

1. **常用类：**
2. **包装类：**Wrapper Class， 均位于java.lang包.

**包装类的用途：**将数据转变为包装类，使得该数据可以使用更多的方法：例如:求解属性最大值、最小值…



“数字型”都是java.lang.Number的子类。

Number类提供了抽象方法：intValue()、longValue()、floatValue()、doubleValue()，所有的“数字型”包装类都可以互相转型。

**eg：**

//基本类型转化为integer对象：两种方法一样，官方推荐第二种

//转化为integer对象后，int型数据就具有一些方法可以使用

Integer int1 = **new** Integer(10); //int1 = 10

Integer int2 = Integer.*valueOf*(20);//int2 = 20

//Integer int2 = 20，与之等价（**自动装箱**）

//将Integer对象转为int型

**int** a = int1.intValue(); //a = 10

//int a = int1；与之等价（**自动拆箱**）

//字符串转化为integer对象

Integer int3 = **new** Integer("999"); //int3 = 999

Integer int4 = Integer.*parseInt*("3334");

//integer对象转化为字符串

String str1 = int3.toString(); //str1 = 999

System.***out***.println("int能表示的最大整数：" + Integer.***MAX\_VALUE***);

注：

1. **自动装箱：**包装类.valueOf（数据）；
2. **自动拆箱**：参数名.xxValue（）；eg：a.intValue();
3. **String类：创建的string变量可以被修改。**

StringBuilder、Stringbuffer称为可变字符序列。两者的区别：

* **StringBuffer**：线程安全，做线程同步检查，效率较低。
* **StringBuilder** ：线程不安全，不做线程同步检查，因此效率较高。 建议采用该类，可以使用各种方法来添加元素。

（**StringBuffer、StringBuilder适合用于for循环**，都是对于字符串本身操作，不会拷贝对象，不占内存）

eg：**s.append((char)(‘a’)); //添加字符。**

* **String**：**无法改变内容**。对于string字符串实际上时对副本进行操作。（先拷贝）string不适合用于循环，会占用大量的内存。
* StringBuilder与String的区别：

String str8 = "";

//本质上使用StringBuilder拼接, 但是每次循环都会生成一个StringBuilder对象

**for** (**int** i = 0; i < 5000; i++) {

str8 = str8 + i;//相当于产生了10000个对象，占内存

}

StringBuilder sb1 = **new** StringBuilder("");

**for** (**int** i = 0; i < 5000; i++) {

sb1.append(i); //只有一个对象

}

1. **Date类：**
2. 时间类可以和字符串相互转化：

时间->字符串：SimpleDataFormat

字符串->时间：

String time = "2007-10-7";

Date date = s2.parse(time);

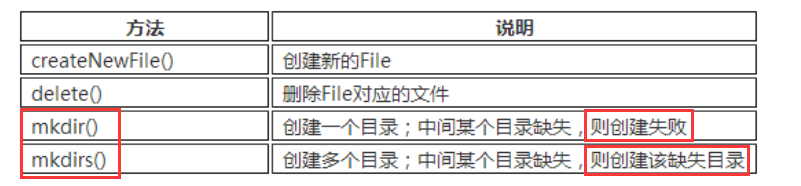
1. 注意 ：年取值范围：年份-1900

月：0-11

日：1-31

星期：1：星期天，2：星期一…

1. **Calendar类：**日历类
2. **Math类：**
3. **File**类：



File file = new File(path); **//创建文件**  
BufferedReader bufreader = new BufferedReader(new FileReader(file)); **// 读取文件**

FileWriter writer = new FileWriter(file); **// 写入文件**

1. **枚举：**

**用于定义一组常量。枚举实质上还是类。**

**示例代码：**

**import** java.util.Random;

**enum** Season{

***SPRING***, ***SUMMER***, ***AUTUMN***, ***WINTER***

}

**public** **class** Enum {

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

**int** a = **new** Random().nextInt(4);

**switch**(Season.*values*()[a]){

**case** ***SPRING***:

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

**break**;

**case** ***SUMMER***:

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

**break**;

**case** ***AUTUMN***:

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

**break**;

**case** ***WINTER***:

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

**break**;

}

}

}



# 异常处理

1. **异常处理：**
2. 异常捕获、处理：

异常捕获：try-catch-finally；

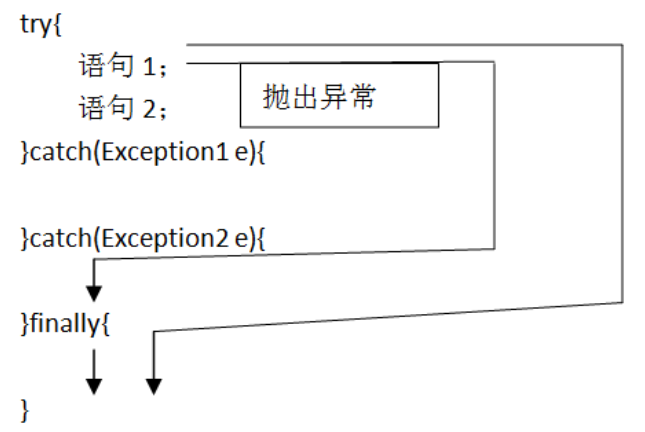
声明异常：throws

**try：**执行一段程序（抛出异常，只能有一个try）；

**catch：**捕捉异常（可以有多个catch）

**finally：一定会执行的代码**（无论程序是否抛出异常）

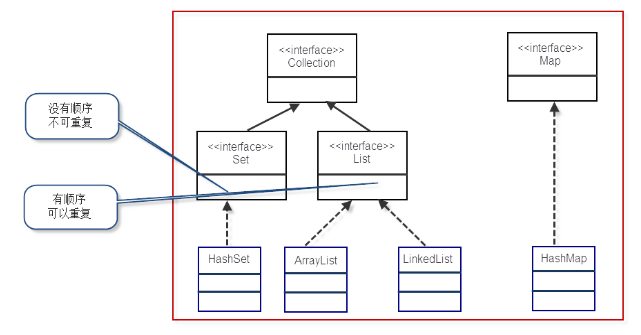
1. 处理流程：



1. Exception类：是所有异常的父类，其子类对应了各种异常事件。

# 容器

1. **容器：**也就是集合，数组也是一种容器
2. **接口结构：**



1. **Collection接口：**List 、Set是Collection的子接口

* Collection所具有的方法：

也就是Set、HashSet、List、ArrayList、LinkedList所具有的方法。

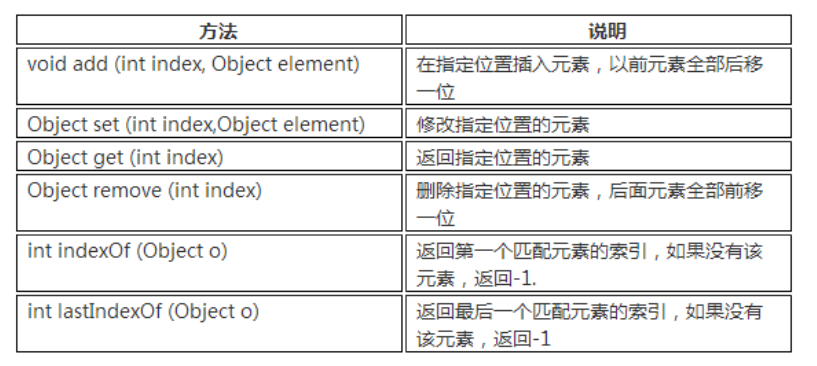


* 容器中泛型的使用：

public class CollectionTest**<E>** { **//<E>：表示使用泛型E，E可以是自定义类**  
 Object[] obj = new Object[5];**//定义object数组，用于存放数据**  
 public **E** get(int i){ **//返回数据类型为，泛型**  
 return **(E)**obj[i];  **//取变量时，需要强制转型**  
 }  
 public void set(**E** data,int i){  
 obj[i] = data;  
 }  
 public static void main(String[] args) {  
 CollectionTest**<String>** myCollection = new CollectionTest();  
 myCollection.set("Chris",0);  
 System.*out*.println(myCollection.get(0));  
 }  
}

1. **List（继承自Collection）:**

* 特点：有顺序、**可重复**
* List接口的实现类：
* ArrayList：底层用数组实现
* LinkedList：双向链表
* Vector：List
* List中的方法：



* 使用List：ArrayList

**//使用泛型：String**

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

**//添加元素**  
arrayList.add("Chris");

**//转为Object数组**  
Object[] objects = arrayList.**toArray**();  
System.*out*.println("objects[0] = "+objects[0]);

* 使用List：LinkedList——双向链表（手写LinkedList，详见21点）

List<String> linkedList = new LinkedList<String>();  
linkedList.add("aa");

* 使用List：Vector：

**如何选用ArrayList、LinkedList、Vector?**

* **需要线程安全**时用Vector。(在多线程中使用,保证线程同步)
* 不存在线程安全问题时，**查找**较多用ArrayList
* 不存在线程安全问题时，**增加或删除元素**较多用LinkedList。

1. **Set（继承自Collection）**：

* 特点：无顺序、**不可重复**
* 分类：**TreeSet 、HashSet（常用，本质是HashMap但只添加key）**
* **HashSet**：使用HashMap实现，本质上就是简化版的HashMap
* **TreeSet**：使用TreeMap实现，TreeSet会对元素进行排序，需要在对应类的内部实现Comparab接口，才能根据compareTo（）方法比较对象之间的大小。将元素存储在**红-黑二叉树**中。
* 使用HashSet

public class SetTest {  
 public static void main(String[] args) {  
 **Set**<String> set1 = new **HashSet**<>();  
 **//add本质就是使用map，添加键，键不能重复，所以Set不能重复**

set1.add("aa");

set1.add("bb");  
 set1.add("bb"); **//添加元素：相同的元素不会被重复添加**  
 System.*out*.println("set1=" + set1);  
 set1.remove("bb");**//删除元素**  
 System.*out*.println("remove 'bb'" + set1);  
 Set<String> set2 = new HashSet<>();  
 set2.add("cc");  
 set2.addAll(set1);**//将其他的set元素添加进来**  
 System.*out*.println("set2=" + set2);  
 }  
}

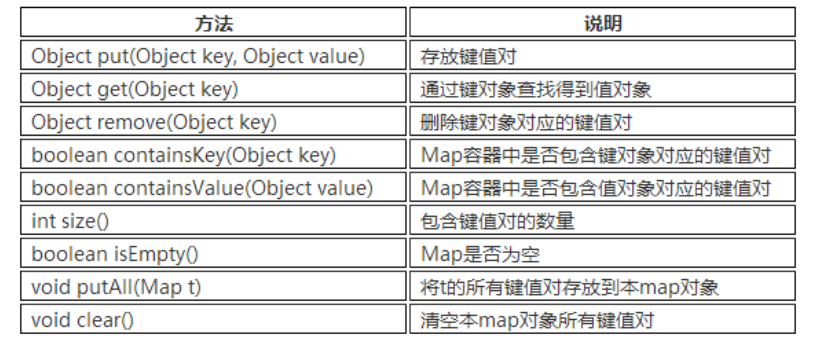
* 使用TreeSet：

public class TreeSet\_Test {  
 public static void main(String[] args) {  
 Set<Integer> set = new **TreeSet**<>();  
 set.add(300);  
 set.add(200);  
 set.add(600);  
 for (Integer m : set)  
 System.*out*.println(m);  
 Set<Emp1> set1 = new **TreeSet**<>();  
 set1.add(new Emp1(100, "CC", 3200));  
 set1.add(new Emp1(502, "DD", 2000));  
 set1.add(new Emp1(30, "EE", 50000));  
 for (Emp1 m : set1)   
 System.*out*.println(m);  
 }  
}  
class Emp1 implements **Comparable**<Emp1> {  
 int id;  
 String name;  
 double salary;  
 public Emp1(int id, String name, double salary) {  
 super();  
 this.id = id;  
 this.name = name;  
 this.salary = salary;  
 }  
 @Override  
 public int compareTo(Emp1 o) { **//需要自己实现compareTo比较功能**

if (this.salary > o.salary)  
 return 1;  
 else if (this.salary < o.salary) return -1;  
 else {  
 if (this.id > o.id) return 1;  
 else if (this.id < o.id) return -1;  
 else eturn 0;  
 }  
 }  
 @Override  
 public String toString() {  
 return "id: " + id + ",name: " + name + ",salary: " + salary;  
 }  
}

1. **Map**接口：

* Map中有3个类：HashMap、TreeMap、HashTable
* 用来存储键值对，实现成对的存储关系。（**键：不能重复**）



* **HashMap**：哈希表
* **hash表：**

**1、本质：**数组+链表（单向链表）

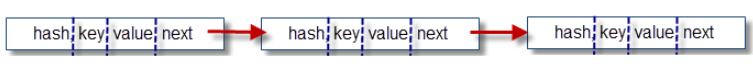
**2、数组**：占比空间连续、寻址容易、查询速度快，但增删效率低。

**3、链表**：占空间不连续，寻址困难，查询速度慢，但增删效率高。

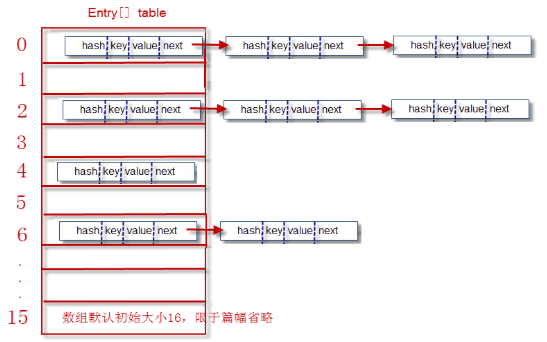
* **try[] table：用于存放哈希表+数据**

1、是hashMap的核心数组（位桶数组）；

2、Entry对象存储：key（键对象），value（值对象），next（下一个节点），hash（键对象的hash值），因此Entry对象是一个**单向链表结构**。



* **Entry 数组存储结构：**



* 数据存储流程：
* 获得键对象（类型Object）的hashcode()方法：使用hashcode()得到hash码，用hash码计算得到的**hash值[0，数组长度-1]**之间。计算hash值方法如下：

**1、相除取余法**：

hash值 = hash码 % 数组长度，约定**数组长度为2的整数幂**

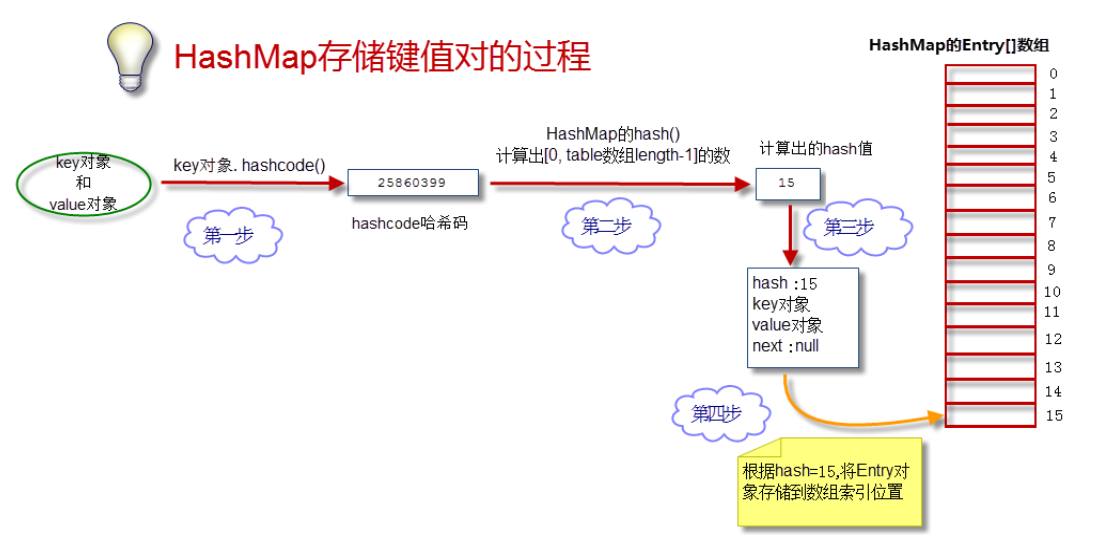
**2、示例代码：**计算hash值

法一：System.*out*.println(h & (length - 1)); **//位操作，效率高**  
法二：System.*out*.println(h % length); **//取余**

* 生成Entry对象：

包含hash、key、value、next（指向下一个entry对象）

* 将Entry对象存入table数组中，如果该**索引位置**（即：**hash值**在表中的位置）已经有entry对象，则将已存在对象的next指向本entry对象，形成链表。



* 数据获取流程：
* 使用key查找：先获取哈希码（hashCode），key.hashCode；
* 使用哈希码获取哈希值（hash）：HashMap的hash()，得到的hash值为HashMap的Entry[]数组的索引；
* 使用equals()来找到对应的值（value）；

**注意：Java中两个对象的equals()为true，则他们的hashCode（哈希码）一样。**

* **TreeMap：**红黑二叉树的实现，一般用于排序

1. 使用Iterator迭代器遍历List、Set、Map

public class Iterator\_Test {  
 public static void main(String[] args) {  
 System.*out*.println("Iterator for List...");  
 *testIteratorList*();  
 }  
 /\*\* **iterator遍历List**  
 \* **静态方法属于类：可以在类中直接使用方法名调用**  
 \*/  
 public static void testIteratorList(){  
 List<String> list = new ArrayList<>();  
 list.add("aa");  
 list.add("cc");  
 list.add("bb");  
  
 for(**Iterator**<String> iter = list.**iterator**(); iter.**hasNext**();){  
 String temp = iter.**next**();  
 System.*out*.println(temp);  
 }  
 }

}

1. **Collections工具类：与collection类不是同一个东西**

java.util.Collection提供对Set、list、Map进行paixue、填充、查找元素的方法。

**1、 void sort(List)** ：升序排序。

**2、 void reverse(List)** ：逆序排列（反转数列）

**3、 void shuffle(List)** ：**行**随机排列。

**4、 void fill(List, Object)** ：用特定的对象重写整个List容器。

**5、 int binarySearch(List, Object)**：对于顺序的List容器，采用**二分法**查找特定对象。

public class Collections\_Test {  
 public static void main(String[] args) {  
 List<String> lits = new ArrayList<String>();  
 for(int i = 0; i< 5; i++)  
 lits.add("a" + i);  
 System.*out*.println("原数组:" + lits);  
 Collections.*shuffle*(lits);  
 System.*out*.println("行随机排序:" + lits);  
 }  
}

1. **iterator、iterable：迭代器**

**提供统一的方法来遍历容器**

遍历的3种方法：

* for
* 增强for循环：for each
* Iterator：迭代器

1. **iterator、****iterable的区别：**
2. **iterator：**

**用于操作集合对象**，该迭代器提供了对集合对象（collection）遍历的操作方法，无需关心集合对象内部的实现方法，**iterator只能向前移动。需要遍历、删除元素时，可使用该方法。**

**（1）java的声明如下：**

public interface Iterator<E> {

boolean hasNext();

E next();

void remove();

}

**（2）常见用法：**

public static void testIteratorList() {  
 **List**<String> list = new **ArrayList**<>();  
 list.add("aa");  
 list.add("cc");  
 list.add("bb");  
  
 for (**Iterator**<String> iter = list.**iterator**(); iter.**hasNext**(); ) {  
 **//hasNext：有没有下一个元素**

**/\*next：返回值类型是Object，需要类型转换。指定类型就不用转换**

**\* 返回当前内容，并指向下一个**

**\*/**

String temp = iter.next();  
 System.*out*.println(temp);  
 }  
}

1. **iterable：**

**在iterator的基础上加入了for each循环**。直接返回iterator对象。**所有的iterator对象均可使用foreach遍历**。

**（1）java声明如下：**

public interface Iterable<T> {

Iterator<T> iterator();

}

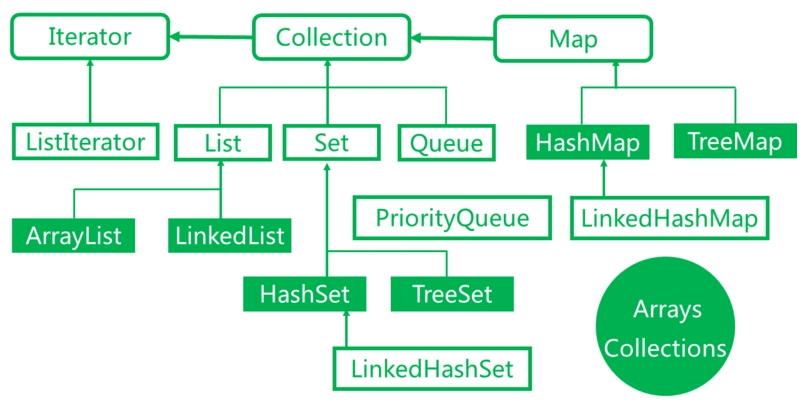
1. **Collections工具类：**

Collections和Collection容器不一样。Collection工具类给Collection容器提供**查找、排序、填充**等方法。

public static void main(String[] args) {  
 **List**<String> aList = new **ArrayList**<String>();  
 for (int i = 0; i < 5; i++){  
 aList.add("a" + i);  
 }  
 System.*out*.println(aList);  
 Collections.**shuffle**(aList); **// 随机排列**  
 System.*out*.println(aList);  
 Collections.**reverse**(aList); **// 逆续**  
 System.*out*.println(aList);  
 Collections.**sort**(aList); **// 排序**  
 System.*out*.println(aList);  
 System.*out*.println(Collections.**binarySearch**(aList, "a2")); **//二分查找**  
 Collections.**fill**(aList, "hello"); **//填充**  
 System.*out*.println(aList);  
}

1. **java容器结构图：**

**绿色填充框：常用的类**



# IO篇

1. **核心类：**
2. **File**：文件类
3. **InputStream**：字节输入流，**字节流操作字节**
4. **OutputStream**：字节输出流
5. **Reader**：字符输入流，**字符流操作纯文本，字符流底层基于字节流**
6. **Writer**：字符输出流
7. **Closeable**：关闭流接口，**释放资源**
8. **Flushable**：刷新流接口，这边的**flush**在**outputStream**中使用，强制输出字节，防止数据驻留在内存中（**需要手动调用flush**）
9. **Serializable**：序列化接口
10. **IO简介（imput、output）**
11. **流分类：**

**1）、按方向**

**输入流**：数据源-->程序,InputStream、Reader读进来；

**输出流**：程序-->目的地，OutputStream、Writer写出去；



2**）、按功能**

**节点流**：直接从数据源、目的地读取数据

**处理流（包装流）**：其他流的封装，目的：提高性能、简化操作。

（**节点流、处理流的关系：**

* 节点流：处于IO操作的第一线，所有操作必须先经过节点流
* 处理流：可以对其他流进行处理，优化性能）

1. **四大抽象类：**
2. **InputStream：**

字节输入流的所有类的父类，不可实例化，派生不同的节点流子类。继承自InputStream的流都是用于向程序中输入数据，数据的单位为8bit。

**常用方法：**

* int read()
* void close()

eg：

**int read();** 读取一个字节的数据，将字节值作为int类型返回（0-255），**未读出字节则返回-1（-1表示结束）**

1. **OutputStream**

字节输出流的所有类的父类，发送字节到某个地方。

**常用方法:**

* void write(int)
* void flush()：强制刷新，输出字节
* coid close()

eg：

void write(int n); //向目的地写入一个字节

1. **Reader：**

读取字符流抽象类，数据单位：字符

**常用方法:**

* int read()
* void close()

eg：

int read();//读取一个字符，将字符值作为int型返回（0-655535），未读出则返回-1.

1. **Writer：**

用于写入字符流抽象类，数据单位：字符

**常用方法:**

* void write()
* void flush()：强制刷新，避免出现数据无法即使输出
* void close()

eg：

void write(int n); //向输入流写入一个字符

1. **void close()；** //关闭对象，释放资源**（1-4的四个操作之后，必须加此函数）**
2. **IO的标准操作步骤：**
3. 创建源
4. 选择流
5. 具体操作
6. 释放资源

public static void main(String[] args) throws IOException {  
 **// 1、创建操作源**  
 File src = new File(System.*getProperty*("user.dir") + "\\data\\test.txt");  
 **// 2、选择流: FileInputStream，文件输入流，用于读取文件**  
 InputStream is = new **FileInputStream**(src);  
  **// 3、具体操作：读取操作，read**  
  **// 方式一：一个一个字节读取**  
 int temp;  
 while((temp = is.read()) != -1)  
 System.*out*.print((char)temp);  
 **// 方式二：一次性读取多个字节**  
 byte[] datas = new byte[1024];  
 int len = -1;  
 while((len = is.read(datas)) != -1){ **//read读取的数据存放到datas自己数组中**  
  **// 解码：字节数组->字符串，因为，datas是字节数组**  
 String res = new String(datas, 0, len);  
 System.*out*.println(res);  
 }  
 **// 4、释放资源**  
 if(null !=is)  
 is.close();  
}

1. 字节数组：

**任何数据都可以转换成字节数组，字节数组方便在网络中传输。实际中，除了字符串的之外的数据转换，都是使用字节数组来对接。**

**例如：**图片的传输

图片🡪字节数组ByteArrayInputStream、ByteArrayOutputStream🡪图片。

**流和数组之间的转换**：ByteArrayInputStream、ByteArrayOutputStream。

1）**FileInputSream**：把文件当成数据源；

2）**ByteArrayInputStream**：把内存中某个字节数组对象当成数据源。不需要释放，所有的数据都可以转换成字节数组（就是将数据转成二进制），方便后面的网络数据传输。

（**ByteArrayInputStream的操作和InputStream一样**）

3）**ByteArrayOutputStream：**

数据转为字节数组。

eg：

**//1、创建源与目的地**  
File src = new File(filePath);  
byte[] dest =null;  
**//2、选择流**  
InputStream is = new FileInputStream(src);  
ByteArrayOutputStream baos = new ByteArrayOutputStream();  
**//3、操作 (分段读取)**  
byte[] datas = new byte[1024\*10]; **//缓冲容器**  
int len = -1; **//接收长度**  
while((len=is.read(datas))!=-1) { **//读取文件，存到datas字节数组中，返回字节数**  
 baos.write(datas, 0, len);  **//输出datas字节数组**  
}

**//4、强制刷新**  
baos.flush();  
return baos.toByteArray(); **//将接收的数据转换成字节数组**

1. **设计模式：**IO装饰类：
2. **抽象组件：**提供接口；
3. **具体组件**：需要装饰的对象；
4. **抽象装饰类**：包含对抽象组件的引用 + 装饰共有的方法；

**implement**：用于继承**接口；**

**extends**：用于继承**类；**

1. **具体装饰类**：被装饰的对象；
2. **装饰流共5种：**

* 字节、字符缓冲流：字节（二进制数据）、字符（文本）
* 字符转换流
* 数据流
* 对象流
* 打印流（了解，不常用）

1. **装饰流1：**字符缓冲流、字节**缓冲流**

**缓冲流的作用**：提升读写的操作速度

* **BufferedWriter**：写入，写入分隔符——newLine()
* **BufferedReader**：读取，可以逐行读取——readLine()
* **BufferedInputStream**：和InputStream一样，默认缓冲空间8k
* **BufferedOutputStream**：和OutputStream一样

**（所有的输入、输出，都是指java的输入、输出，不是所操作的文件）**

**eg：缓冲流例程，在操作InputStream、OutputStream时加入缓冲流，可提升性能**

public static void main(String[] args) throws IOException {  
 **// 1、创建操作源**  
 File src = new File(System.*getProperty*("user.dir") + "\\data\\test.txt");  
 **// 2、选择流: 使用缓冲流——BufferedOutputStream，提升读取速度**  
 InputStream is = new **BufferedOutputStream**(new **FileInputStream**(src));  
  **// 3、具体操作：读取操作，read**  
 byte[] datas = new byte[1024]; **// 方式二：一次性读取多个字节**  
 int len = -1;  
 while((len = is.read(datas)) != -1){ **//read读取的数据存放到datas自己数组中**  
  **// 解码：字节数组->字符串，因为，datas是字节数组**  
 String res = new String(datas, 0, len);  
 System.*out*.println(res);  
 }  
 **// 4、释放资源**  
 if(null !=is)  
 is.close();  
}

**eg：文件缓冲流例程，在操作FileReader、FileWriter时，加入缓冲流**

public static void copyFile(String srcPath,String destPath) {  
 **//1、创建源**  
 File src = new File(srcPath);  **//源头**  
 File dest = new File(destPath);**//目的地**  
  **//2、选择流：使用缓冲流——BufferedReader、BufferedWriter**  
 BufferedReader br=new **BufferedReader**(new **FileReader**(src));  
 BufferedWriter bw =new **BufferedWriter**( new **FileWriter**(dest));  
 **//3、操作 (逐行读取)**  
 String line =null;  
 while((line=br.readLine())!=null) {  
 bw.write(line); **//逐行写出**  
 bw.newLine()； **//写入新的一行——分隔符**   
 }  
 bw.flush();  
}

1. **装饰流2：转换流**：

**作用：**字节流<-->字符流（转换） ，**用与处理纯文本的数据**

* **InputStreamReader**：数据读取，可用于**解码**：字节数组->其他数据
* **OutputStreamWriter**：数据存储，可用于**编码**：其他数据->字节数据

eg：获取键盘输入字符，并输出

public static void main(String[] args) {  
  **//使用InputStreamReader、 OutputStreamWriter操作System.in、System.out**

**（这里：使用了缓冲流 + 转换流）**  
 try(BufferedReader reader = new BufferedReader(new **InputStreamReader**(System.*in*));  
 BufferedWriter writer = new BufferedWriter(new **OutputStreamWriter**(System.*out*));) {  
 **//获取键盘的数据**  
 String msg = "";  
 while (!msg.equals("exit")){  
 msg = reader.readLine();**//循环读取**  
 writer.write(msg);  **//循环写出**  
 writer.newLine(); **//分隔符**  
 writer.flush(); **//强制刷新**  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

1. **装饰流3**：数据流

**作用：操作基本数据类型、字符串。**方便处理数据类型，保留数据的类型。

**注意：这里读取、写入的顺序要保证一致**

* DataInputStream：
* DataOutputStram：

public static void main(String[] args) throws IOException {  
 **//写出：将数据写出到字节数组，使用里缓冲流，提升速度**

**//改成new FileOutputSream(“xx.txt”)就可以写入文件xx.txt**  
 ByteArrayOutputStream baos =new ByteArrayOutputStream();  
 DataOutputStream dos =new **DataOutputStream**(new **BufferedOutputStream**(baos));  
  **//操作数据类型 +数据**  
 dos.writeUTF("编码辛酸泪"); **// 一个中文：3个字节**  
 dos.writeInt(30);  **// 一位数字：1个字节**  
 dos.writeBoolean(false);  **// 一个字母：1个字节**  
 dos.flush();  
 byte[] datas =baos.toByteArray();  
 System.*out*.println(datas.length);  
 **//读取**  
 DataInputStream dis =new DataInputStream(new BufferedInputStream(new ByteArrayInputStream(datas)));  
 **//获取数据时，顺序与写出一致**  
 String msg = dis.readUTF();  
 int age = dis.readInt();  
 boolean flag = dis.readBoolean();  
}

1. **装饰流4：**对象流

* **作用：**操作所有对象、基本数据类型、字符串。**使用和数据流相同**，**也需要保证读写顺序一致。**
* **概念：序列化Serializable，不是所有的对象都可以序列化。**
* **ObjectInputStram**：读取数据，反序列化
* **ObjectOuputStream**：写入数据，序列化。**读取对象之后，需要还原对象，根据对象类型，进行强制转换。**

**transient**：用于防止属性序列化。

**对象可序列化**：必须拥有serializable接口。

1. **装饰流5：**打印流

**作用：System.out就是这个流**

* PrintStream：

1. commens IO：Appach提供的IO操作，已经封装好API
2. 注解@

添加注解后，可以在后面使用该注解的数据——通过解析注解来获得数据。

作用：

1. 生成文档：常见的注解：@see、@param、@return；
2. 跟踪代码依赖性，替代配置文件：在sping2.5中，使用注解来减少配置文件；
3. 在编译时检查格式：例如使用@override，若该方法不是覆盖了父类方法，在编译时就会检测出。
4. java中包的区别：java.\* 、javax.\* 、org.\* 、com.sun.\*
5. **java.\*：**java SE的标准库，java标准的一部分，不会轻易修改，保持向后兼容，所有jdk的实现在java.\*都是一样的；
6. **javax.\***：也属于java标准的一部分，但没有包含在标准库中，属于标准库的扩展；
7. **com.sun.\***：sun是hotspot虚拟机中java.\*与javax.\*的实现类，包含在rt中，可能不会向后兼容，不推荐使用；
8. **org.\***：有企业、组织提供的java类库，没有向后兼容性，常用的是w3c提供的XML、网页、服务器的类和接口。

# 多线程

1. **多线程： （少用继承、多用实现，java是单继承，方便之后代码重构）**

一共有三种创建线程的方法：

1. 继承**Thread类**（常用），使用**extend**继承类——**单继承；**
2. 实现**Runnable接口**（常用），使用**implements**继承接口——**多继承；**
3. 实现**Callable接口**：在**并发**中使用。

**eg：使用继承Thread，实例化1次，只能创建一个线程**

public class StartThread extends Thread {  
 **// 线程入口:start方法会自动调用run方法。**

**// 直接调用run，不是执行多线程**  
 @Override  
 public void run(){  
 for(int i = 0; i < 500; i++)  
 System.*out*.println("sing song");  
 }  
 public void like(){  
 for(int i = 0; i < 500; i++)  
 System.*out*.println("I just know you like him");  
 }  
 public static void main(String[] args) {  
 StartThread startThread = new StartThread();  
 **// 仅仅是开启线程，等待cup自己调用**  
 startThread.start();  
 startThread.like();  
 }  
}

**eg：使用继承Runnable，实例化1次，可以创建多个线程**

public class StartRunnable implements Runnable {// **使用Runnable创建线程**  
  **// 线程入口:start方法会自动调用run方法。**  
 public void run(){  
 for(int i = 0; i < 500; i++)  
 System.*out*.println("THREADNAME = "+Thread.*currentThread*().getName());  
 }  
 public void like(){  
 for(int i = 0; i < 500; i++)  
 System.*out*.println("I just know you like him");  
 }  
 public static void main(String[] args) {   
 StartRunnable std = new StartRunnable();

// **仅仅是开启线程，等待cup自己调用，创建两个线程--使用静态代理(类已经构建好)**

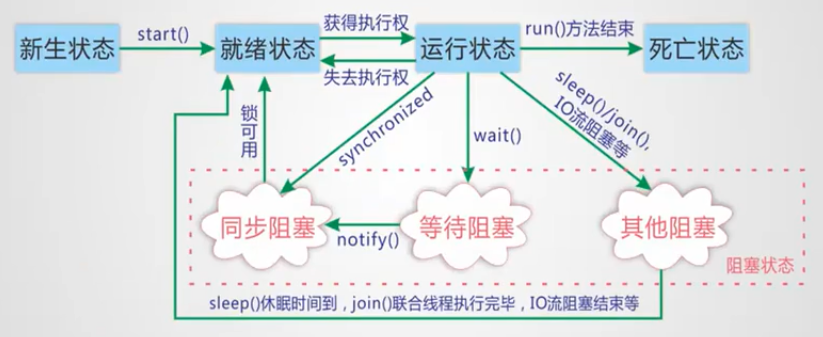
Thread T = new Thread(std,"std-1");// **不使用匿名的方式创建线程,线程别名：std-1**

T.start();  
 new Thread(std,"std-1").start(); // **使用匿名的方式创建线程，给线程取名字：std-1**  
 new Thread(std,"std-2").start(); // **使用匿名的方式创建线程，给线程取名字：std-2** std.like();  
 }  
}

1. Lambda表达式：用于简化线程表达式**（两种方法）**

public class Lambda {  
 public static void main(String[] args) {  
 **// 方法一：直接写方法**  
 new Thread(()->{  
 System.*out*.println("Lambda Thread, 减少匿名内部类的使用");  
 }).start();  
 **//方法二：使用interface接口：实现参数、返回值**  
 Interest interest = (int a, int b) -> {  
 System.*out*.println("Lambda method run = " + (a + b));  
 return (a+b);  
 };  
 interest.lambda(100,200);  
 }  
}  
interface Interest{  
 int lambda(int a, int b);  
}

1. **线程状态**



**当run()结束后，线程自动停止。**

1. **线程方法**
2. sleep()：线程停止运行一段时间，处于阻塞状态；（**用于倒计时、模拟网络延时：找出可能出现的问题**）
3. join()：阻塞指定线程，等待另一个线程完成后再继续运行；**（用于合并线程）**
4. yield()：暂停当前运行的线程，使线程进入就绪状态，不是阻塞线程；
5. setDaemon()：将线程设置为后台线程——守护线程；
6. setPriority(int newPriority)、getPriority()：调整线程优先级，取值1-10，默认为5；**这里的优先级指的是概率的大小，并不是高的先执行。**
7. stop()：停止线程，**一般不用这个方法停止线程**
8. 线程停止的示例代码

public class ThreadStop implements Runnable{  
  **// 标志位，控制run方法的结束与否，即线程的结束与否**  
 private boolean flag = true;  
 @Override  
 public void run(){  
 int i = 0;  
 while (flag){  
 System.*out*.println(Thread.*currentThread*().getName() + "-->" + i);  
 i++;  
 }  
 }  
 **// 手动将标志位置为false，结束run()方法—>结束线程**  
 public void stopthread(){  
 flag = false;  
 System.*out*.println(Thread.*currentThread*().getName() + " ---- END ---- ");  
 }  
 public static void main(String[] args){  
 ThreadStop threadStop = new ThreadStop();  
 new Thread(threadStop, "CHRIS").start();  
  
 for(int i = 0; i < 100; i++){  
 System.*out*.println("th: "+ i +" main RUNING...");  
 if(i == 90)  
 threadStop.stopthread();  
 }  
 }  
}

1. yield线程暂停的示例代码

public class ThreadYield {  
 public static void main(String[] args) {  
 MyYield myYield = new MyYield();  
 new Thread(myYield, "thread-1").start();  
 new Thread(myYield, "thread-2").start();  
 }  
 static class MyYield implements Runnable{  
 @Override  
 public void run() {  
 System.*out*.println(Thread.*currentThread*().getName() + " RUN ");  
 Thread.*yield*(); // **暂停当前线程**  
 System.*out*.println(Thread.*currentThread*().getName() + " END ");  
 }  
 }  
}

1. join插入线程

public class ThreadJoin {  
 public static void main(String[] args) {  
 MyJoin myJoin = new MyJoin();  
 Thread T = new Thread(myJoin, "Join");  
 T.start();  
 for(int i = 0; i < 100; i++){  
 if(i == 30) { **// 当i=30时，main线程被中断，执行myJoin线程**  
 try {  
 T.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();**//在命令行打印异常信息在程序中出错的位置及原因**  
 System.*out*.println("ERROR");  
 }  
 }  
 System.*out*.println("th :"+ i + " Thread: main runing...");  
 }  
 }  
 static class MyJoin implements Runnable{  
 @Override  
 public void run() {  
 System.*out*.println("Thread: lambda runing ...");  
 for(int i = 0; i < 100; i++)  
 System.*out*.println("th :"+ i + " Thread: myJoin runing...");  
 }  
 }  
}

1. 获取线程状态

State st = 线程名.getState();

System.out.println(st);

1. 优先级

public class ThreadPriority {  
 public static void main(String[] args) {  
 System.*out*.println("优先级高的，仅表示先执行的概率大，并不是先执行");  
 MyPriority myPriority = new MyPriority();  
 Thread T1 = new Thread(myPriority,"T1");  
 Thread T2 = new Thread(myPriority,"T2");  
 Thread T3 = new Thread(myPriority,"T3");  
 // **设置优先级：所有的设置必须在线程启动前，即start之前**  
 T1.setPriority(Thread.*MAX\_PRIORITY*);  
 T2.setPriority(Thread.*NORM\_PRIORITY*);  
 T3.setPriority(Thread.*MIN\_PRIORITY*);s  
 T1.start();  
 T2.start();  
 T3.start();  
 }  
 static class MyPriority implements Runnable{  
 public void run(){  
 System.*out*.println(Thread.*currentThread*().getName() + "-->" + Thread.*currentThread*().getPriority());  
 }  
 }  
}

1. Daemon设置后台线程

MyDaemon myDaemon = new MyDaemon(); //创建线程类，对象  
Thread T1 = new Thread(myDaemon, "T1");  
T1.setDaemon(true); **//设置T1为后台线程**  
T1.start();

1. **线程并发**

**线程不安全的情况：**一般存在于对数据的修改时，数据读取不会存在不安全。

**//** **这是一个测试线程不安全的例程：结婚基金的账户余额会存在负数** 2019/10/6.  
public class ThreadSafe {  
 public static void main(String[] args) {  
 Account account = new Account("MARRIAGE FUNDATION", 100);  
 Withdrawer chris = new Withdrawer(account,"chris",40);  
 Withdrawer yj = new Withdrawer(account,"jy",80);  
 Thread TChris = new Thread(chris);  
 Thread TYJ = new Thread(yj);  
 TChris.start();  
 TYJ.start();  
 }  
 static class Account{  
 String AccountName; //账户名  
 int money; //账户里的钱  
 public Account(String accountName, int money){  
 this.AccountName = accountName;  
 this.money = money;  
 }  
 }  
 **// 使用extends继承Account类， 使用implements继承Runnable接口**  
 static class Withdrawer **extends** Account **implements** Runnable{  
 Account account; // 提款账户  
 int withdrawals; //提款金额  
 int packetMoney; //余额  
 **// 将实例化的类account， 作为参数传入**  
 public Withdrawer(Account **account**, String accountName, int withdrawals){  
  **// 调用父类的构造函数,**初始化提款账户的名字，初始余额  
 super(accountName, 0);  
 // 子类的属性赋值  
 this.account = account;  
 this.withdrawals = withdrawals;  
 }  
 @Override  
 public void run() {  
 account.money -= withdrawals;  
 packetMoney += withdrawals;  
 System.*out*.println(this.AccountName+ " pocket money =" + packetMoney);  
 System.*out*.println(this.account.AccountName + " remaining ballance =" + account.money);  
 }  
 }  
}

1. **出现线程并发的情况：**

存在多个用户对统一资源进行操作时。

1. **线程同步：**

（1）线程加入队列 + 使用锁机制（**synchronized**）

（2）**线程锁**：可能导致调度延时，引起性能问题。

（3）synchronized包含两种：同步方法、同步块

1、同步方法：

直接选择锁定的方法即可。

2、同步块：

**必须选择正确的方法、块才能实现线程同步，否则容易出现错误.**

**选择的范围太大，导致性能下降；**

**范围太小，导致无法锁定，无法实现同步。**

**（这边的范围：指数据的完整性，不是代码的范围）**

（4）eg：同步方法

public **synchronized** void method(int i){ }

eg：同步块

**synchronized**(**需要锁定的块**：类名){ (添加处理代码、方法) }

public void run() {  
 method();  
}  
private void method(){

// 使用同步块，实现同步——锁定**account：类名**

synchronized (**account**){

**（添加处理代码、方法）**

}

}

1. CopyOnWriteArrayList：锁定线程，用于高并发状态
2. **并发处理：**使用**生产者、消费者模式**处理，有两种方法
3. 管程法：使用wait()阻塞线程，等待满足一定条件，再开启方法
4. 信号灯法：使用标志位，来控制wait()，唤醒哪个方法

线程间通信方法：

1. wait()：线程等待，与sleep()不同，wait会解锁线程；
2. wait(long timeout)：等待多少毫秒；
3. notify()：唤醒处于等待状态的线程；
4. notifyAll()：唤醒统一个对象上所有调用wait()方法的线程；
5. Timer定时器使用：

public class ThreadTimer {  
 public static void main(String[] args) {  
 Task task = new Task();  
 Timer timer = new Timer();  
 // 定时调度：延时1000ms，周期21000ms循环执行  
 timer.schedule(task, 1000,2000);  
 }

**// TimerTask是类，所以使用extends继承**  
 static class Task extends TimerTask {  
 int count=0;  
 @Override  
 public void run() {  
 for(int i = 0; i< 2; i++){  
 System.*out*.println("th; " + count + " RUN TimerTask");  
 count++;  
 }  
 System.*out*.println("========END=======");  
 }  
 }  
}

1. 任务调度：一般使用下面四个
2. Scheduler：调度器，控制所有调度；
3. Trigger：触发条件，采用DSL模式（domain-specific language领域特定语言：方法链、函数嵌套。。。）；
4. JobDetail：需要处理的JOB；
5. Job：执行逻辑。

1. 关于Quartz使用：在Spring中会用到 。

测试代码：见ThreadAdvanced.Quartz包

1. HappenBefore：重排代码，用于调整代码运行的顺序

指令重排：指的是，两个线程之间，不存在数据交互、依赖时，会出现线程执行错误的情况。

解决方案：使用flag标志位，可以解决。

1. volatitle：保证线程间变量的可见性、数据同步

当线程对变量进行修改之后，立刻将变量回写到主内存，保证变量最新。

可以避免重排

1. **单例模式**（dcl：double check locking）：在多线程情况下，对外存在一个对象

**一个类只有一个实例，使用getInstance创建（创建的对象属于静态对象），不使用new object。**

**getInstance创建的对象可以跨栈区域使用，new只能在本地使用。**

单例模式的实现：可以使用double-check、volatile实现

步骤：

1. 提供私有的静态属性：使用getInstance创建静态对象

// 1、提供私有属性,**volatile**，**避免其他线程访问不存在的对象**  
private static volatile DoubleCheckLock *instance*;

1. 构造器私有化

// 2、构造器私有化  
private DoubleCheckLock(){  
}

1. 提供公共的静态方法：访问属性

// 3、公共方法，访问私有属性，**加入延时，可以模拟网络延时**  
public static DoubleCheckLock **getInstance**(long time){

}

1. ThreadLocal:

每个线程都有自己的空间，ThreadLocal可以在多线程的情况下保证成员变量的安全，常用的方法如下：

1. get()
2. set()
3. initialValue()
4. inheritableThreadLocal：继承父类线程中线程的inheritableThreadLocal的空间大小

一般，**ThreadLocal定义为private static**。

示例代码：

public class ThreadLocalTest {  
 // **1、创建ThreadLocal**  
  **private static** ThreadLocal<Integer> *threadLocal* = new ThreadLocal<>();  
 // **赋初值：方式一**  
 private static ThreadLocal<Integer> *threadLocal\_1* = *threadLocal*.*withInitial*(()->{  
 return 200;  
 });  
 // **使用InheritableThreadLocal继承起点线程的Threadlocal\_2的空间大小**  
 private static ThreadLocal<Integer> *threadLocal\_2* = new **InheritableThreadLocal**<>();  
  
 public static void main(String[] args) {  
 // 没赋值之前，为null  
 System.*out*.println(Thread.*currentThread*().getName() + "--" + *threadLocal*.get());  
 // **2、设置初始值：方式二**  
 *threadLocal*.set(100);  
 *threadLocal\_2*.set(5);  
 System.*out*.println(Thread.*currentThread*().getName() + "--" + *threadLocal*.get() + "--" + *threadLocal\_2*.get());  
 // **3、体现InheritableThreadLocal**  
 new Thread(()->{  
 System.*out*.println("InheritableThreadLocal继承main线程中threadLocal\_2的空间大小："+Thread.*currentThread*().getName() + "--" + *threadLocal\_2*.get());  
 }).start();  
 }  
}

1. 可重入锁：锁可以延续使用 ，并且每个锁都有一个计数器

锁：用于保证并发时数据的一致性

可重入锁实现：**ReenTrantLock** lock = new **ReenTrantLock**();

1. CAS：Compare and Swap，比较并交换

通过不断比较数据、版本的不同，来决定是否更新数据。

AtomicInteger：直接对cup进行操作，速度快

private static **AtomicInteger** stock = new AtomicInteger(5);



# 网络编程

1. **基本概念：**
2. **网络编程：**强调数据传输，TCP、UDP等协议的使用。
3. **网页：**强调上层应用开发，在服务器上，编写应用等等。
4. **IP：**用于定位设备，分类：
5. **IPV4：**32为地址，以点分十表示：192.168.11.11；
6. **IPV6：**128位地址，16个字节，写成8个16位无符号整数，每个整数用4个十六进制表示：3ffe:3201:1401:1280:c8ff:fe20:db39:1945
7. **特殊的IP：**

127.0.0.1：主机回环地址--本机地址；

192.168.0.0——192.168.255.255：私有地址，局域网使用。

1. **Port（**端口）：**用于区分不同软件；**

**电脑拥有的端口数量：65536，TCP、UDP都拥有65536个端口，相同协议下的端口不能重复。**

**关于端口的使用：**

1. 0-1023：公认端口，eg：80为www使用，21为FTP；
2. 1024-49151：注册端口，分配给用户进程、应用程序；
3. 49152-65536：动态、私有端口；

**查看端口的指令：**

1. 查看所有端口：netstat –ano
2. 查看指定端口：netstat –ano|findstr ”808”
3. 查看指定进程：tasklist|findstr “808”
4. **URL：**用于区分网络资源，就是一个网址；
5. **协议：**用于资源、设备之间的通信。
6. **http：**
7. **TCP：需要先建立，再进行通信**。面向安全，但效率低。（**一对一**）
8. **UDP：不用先建立连接，直接发送信息，只管发送、接收信息**。非面向安全，但效率高。（**一对多**）
9. **ICMP：**网络控制信息协议，命令ping使用的，和TCP、UDP不同。
10. **B/S（browser/server）：**在公网上，使用公共的标准进行开发（eg：http），通过浏览器访问公网；

**C/S（client/server）：**在局域网上，自定义标准，需要自己开发client、server，传速速度快。

1. **网络涉及的术语：**
2. **LAN**（Local Area network）：局域网，使用路由器组建的网络；
3. **WAN**（Wide Area Network）：广域网，光猫引出来的线，接在路由器上；
4. **WLAN**（Wireless Local Area Network）：无线局域网，也就是wifi；
5. **eth0：**网卡，使用命令ipconfig得到的eth0所对应的IP地址为机器的**本地IP，**Hwaddr为网卡的**MAC地址**（注：本地IP，在网络中经常使用，用于定位局域网内的一台机器，但是**只要机器重启、重新接入网络，IP都可能会变化，MAC地址可用于标识入网设备**）
6. **lo：**主机的回环地址，使用命令ifconfig得到（127.0.0.1），用于测试网络程序。局域网、外网的设备无法查看该网络程序。在浏览器中，输入127.0.0.1就可以运行相应的WEB网站。
7. **OSI七层模型：**
8. OSI模型：物理层->数据链路层->网络层->传输层->会话层->表示层->应用层；
9. TCP/IP协议参考OSI模型，只有四层:

网络接口->网络层->传输层->应用层



1. **PDU（Data Encapsulation）：**协议数据封装单元，负责OSI模型每层之间的通信。

**处理方式：**从高层到底层，进行数据封装。

**目的：**将数据先分成几段，减小数据包的大小，从应用层开始进行封装，到物理层就可以发送出去。

**步骤：**

1. **应用层：**准备数据，传入下层——传输层；
2. **传输层：**接收应用层数据，添加TCP信息（TCP头部）；传入下层——网络层；
3. **网络层：**接收传输层数据，添加IP头部（添加后的数据单元称为**“包”**）；传入下层——数据链路层；
4. **数据链路层：**接收网络层数据，添加MAC头部、尾部（添加后的数据单元称为**“帧”**），传入下层——物理层；
5. **物理层：**接收数据链路层数据，将其转化为比特流（0、1电平，也就是二进制），在网络上传输。
6. **数据解封（Data dismantling）：**

相对于PDU，是反向的。

1. **DNS（Domain Name System）**：域名系统，将服务器IP地址转换成字符串（网址）。
2. **URL**：（Universal Resource Locator）统一资源定位符，

包括：protocol、host、port、path、parameter、anchor

**URI**：（Universal Resource Identifier）统一资源标识符

**URN**：（Universal Resource Name）统一资源名称，标识资源名称。

1. **URL的组成：**协议、存放资源的主机名、端口号、资源文件名

eg：http://www.google.com:80/index.html

1. **通信接口：**用于建立两个节点通信的工具。分类：
2. 硬件装置：节点间的信息传送；
3. 软件装置：规定节点间的通信协议；
5. java中对于IP的封装：**InetAdress**()

常用2个方法：

1. **getLocalhost()：**获取计算机名+IP地址；
2. **getByName()：**域名、IP之间的相互转换。

上面两个方法，都包含下面两个子方法：

1. **getHostAddress()**：获取IP地址；
2. **getHostName()**：获取计算机名；

public class InetAddressTest {  
 public static void main(String[] args) throws UnknownHostException {  
 **// 1、获取本机IP、计算机名**  
 InetAddress inetAddress = InetAddress.*getLocalHost*();  
 System.*out*.println("本地IP:"+inetAddress.getHostAddress());  
 System.*out*.println("计算机名："+inetAddress.getHostName());  
  **// 2、域名转换成IP**  
 inetAddress = InetAddress.*getByName*("www.baidu.com");  
 System.*out*.println("域名->IP："+inetAddress.getHostAddress());  
 System.*out*.println("域名：" + inetAddress.getHostName());  
  **// 3、IP转换成域名**  
 inetAddress = InetAddress.*getByName*("182.61.200.6");  
 System.*out*.println("IP:"+inetAddress.getHostAddress());  
 System.*out*.println("域名：" + inetAddress.getHostName());  
 }  
}

1. 包含端口的IP，需要使用socket通信，使用**InetSocketAddress**

该类，实现IP套接字地址（IP地址、本机名+端口号）

public class InetSocketTest {  
 public static void main(String[] args) {  
 InetSocketAddress inetSocketAddress = new InetSocketAddress("192.168.154.1",8080);  
 InetSocketAddress inetSocketAddress\_1 = new InetSocketAddress("localhost",9000);  
 System.*out*.println(inetSocketAddress.getHostName());  
 System.*out*.println(inetSocketAddress.getAddress());  
 }  
}

1. URL使用：

简单为网络爬虫示例，获取百度网页的源码。

public class ScrapyTest {  
 public static void main(String[] args) throws MalformedURLException {  
 URL url = new URL("http://www.baidu.com");  
 **//模拟浏览器**  
 HttpURLConnection conn = null;  
 try {  
 conn = (HttpURLConnection) url.openConnection();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 try {

**// 获取：GET**  
 conn.setRequestMethod("GET");  
 } catch (ProtocolException e) {  
 e.printStackTrace();  
 }

**// 设置：模拟浏览器的值**  
 conn.setRequestProperty("User-Agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36");  
  
  **// 输入流**  
 BufferedReader br = null;  
 try {  
 br = new BufferedReader(new InputStreamReader(conn.getInputStream(),"UTF-8"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 **//爬取信息**  
 String msg = null;  
 while (true){  
 try {  
 if (!(null != (msg=br.readLine()))) break;  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(msg);  
 }  
 }  
}

1. TCP详解：transfer control protocol

面向连接，基于字节流的传输层通信协议。

**其上层应用：**HTTP、FTP、SMTP

**特点：**传输可靠，需要建立连接，**一对一**，效率较低，但安全。存在server和client。

1. UDP详解：user datagram protocol：

无连接的传输协议，提供简单的信息传输服务。传输数据时，不能太大，太大需要分开来传送。

**其上层应用：**DNS、SNMP（监控系统、视频管理系统）

**特点：**传输不可靠、不确认数据是否送达，广播发送（**一对多**），协议简单。可能会丢包。

**（注：TCP、UDP属于传输层底层协议，将数据转换成字节数组）**

java下的编程类：

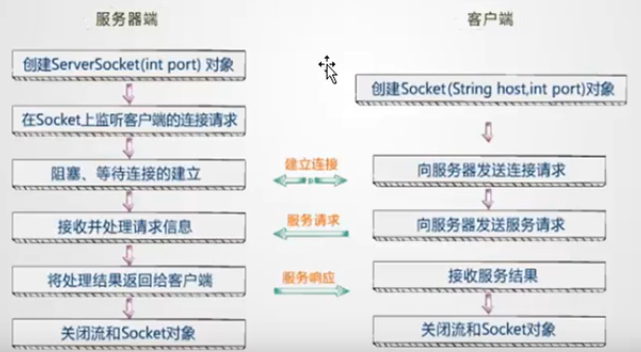
1. **DatagramSocket：**用于发送、接收数据包的套接字**（创建端口）**
2. **DatagramPacket：**数据包**（创建接收数据的容器）**
3. socket：套接字：应用于传输层和应用层通信

编写UDP server、client：**所有操作的数据，均是字节数组（需要数据转换）**

public class **UDPClient** {  
 public static void main(String[] args) throws Exception {  
 System.*out*.println("SENDING...");  
  **// 1、datagramsocket创建发送端**  
 DatagramSocket client = new DatagramSocket(8888);  
 **// 2、数据转化为字节数组**  
 String data = "华东理工大学";  
 byte[] data2byte = data.getBytes();  
  **// 3、封装成datagramsocket包裹，指定目的地：UDP server端口：6666**  
 DatagramPacket packet = new DatagramPacket(data2byte,0,data2byte.length,  
 new InetSocketAddress("localhost",6666));  
  **// 4、发送包裹**  
 client.send(packet);  
  **// 5、关闭**  
 client.close();  
 }  
}

public class **UDPServer** {  
 public static void main(String[] args) throws Exception {  
 System.*out*.println("RECEVING...");  
  **// 1、datagramsocket创建端口**  
 DatagramSocket server = new DatagramSocket(6666);  
  **//2、准备容器，封装成datagrampacket包裹**  
 byte[] container = new byte[1024\*60];  
 DatagramPacket packet = new DatagramPacket(container,0,container.length);  
  **// 3、阻塞式接收：开启阻塞式，运行程序后，只有当接收数据后才会自动停止，因此不需要手动关闭**  
 server.receive(packet);  
 **//4、分析数据**  
 byte[] data = packet.getData();  
 int len = packet.getLength();  
 System.*out*.println(new String(data,0,len));  
 }  
}

1. TCP编程流程：



1. 关于多线程中，Closeable的使用：

由于Closeable可以用于释放资源，且Closeable… 可用于传入不定参数，因此在关闭多个线程时，可以使用如下代码：

public class Closethread{  
 public static void close(Closeable... targets){  
 for(Closeable target : targets){  
 try{  
 if(null != target)  
 target.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 }  
 }  
}

**// 关闭多个线程:a、b、c**

Closethread.close(a, b,c);

1. CopyOnWriteArrayList适合用于并发中，解决：对于List边读取边修改的问题，保证数据的同步。

若是ArrayList，在多线程并发的情况下，可能会出现数据不同步。



# Webserver

1. 反射：reflection

把Java类中的各种结构：方法、属性、构造器、类名，映射成java对象，可以用于对类进行解剖。

1. 关于Class的应用：

public class ClassTest {  
 public static void main(String[] args) throws Exception {  
 ECUST ecust = new ECUST();  
 **// 使用class获取类：的三种方式**  
  **// 1、getClass：只使用getClass()----得到当前运行的类**  
 Class cl = ecust.getClass();  
  **// 2、class**  
 cl = ECUST.class;  
 **// 3、Class.forName（“包名.类名”）：重点掌握**  
 cl = Class.*forName*("ClassTest.ECUST");  
  
 **// 创建对象**  
 ECUST ecust2 = (ECUST)cl.getConstructor().newInstance();  
 System.*out*.println(ecust2);  
 }  
}  
class ECUST{  
 public ECUST() {  
 }  
}

1. **XML：**extensible markup language，可扩展标记语言。

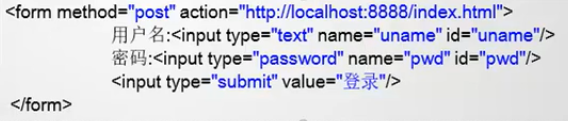
作为数据的一种存储格式，可用于存储软件的参数，程序解析、配置xml文件，就可以不用修改代码，达到更改程序的目的。

1. html(Hyper Text Markup Language)常用标签：

* h1-h6：标题
* p：段落
* div：指定一块区域
* span：
* form：表单
* input：输入
* method：http中的模式，用于和服务器进行交互
* **GET**（查，获取）：量小，请求参数url可见，不安全。一般用于查找/获取资源。浏览器使用GET获取网络资源（URL）。

GET请求数据组成：URL + ？ + 传输数据；（？用于分隔）

* **POST**（改，提交）：量大，请求参数url不可见，安全。一般用于更新资源；
* **PUT**（增）：
* **DELETE**（删）：
* action：请求web服务器的资源，写网址
* name：（用于后端）只要请求服务器，就一定要使用name
* id：（用于前端）区分唯一性
* <ber/>：换行
* <hr>：绘制横线



1. **html固定格式：**

**<html>**开始标签

**<head>**网页上的控制信息

**<title>**页面标题**</title>**

**</head>**

**<body>**页面显示的内容**</body>**

**</html>**

**示例代码：**

<html>  
 <head>  
 <title>Login</title>  
 </head>  
 <body>  
 <pre>  
 Test html<br/>  
 Author:chris<br/>  
 br/:换行  
 </pre>  
 <form method = "post" action = "http://localhost:9999/index.html/">  
 UserName:<input type = "text" name = "uname" id = "uname"/>  
 Password:<input type="password" name = "pwd" id="pwd"/>  
 <input type = "submit" value="Login"/>  
 </form>  
 </body>  
</html>

1. http：Hyper Text Transfer Protocol超文本传输协议
2. **请求协议：**

请求行：方法（get、post）、URI、协议/版本

请求头：Request Header

请求正文：

**eg：**

**1、请求行**

GET /index.html? name=test&pwd=12345 HTTP/1.1

**2、请求体**

**Accept:** text/html, application/xhtml+xml, \* / \*

**Accept-Language:** zh-CN

**User-Agent:** Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36

**Host:** localhost

**Connection:** Keep-Alive

**3、请求正文**

1. **响应协议：**

状态行：协议/版本、状态码、状态描述

响应头：Response header

响应正文：

eg：

1. **请求行：**

POST /index.html HTTP/1.1

1. **请求体**

**Accept:** text/html, application/xhtml+xml, \* / \*

**Accept-Language:** zh-CN

**User-Agent:** Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36

**Host:** localhost

**Connection:** Keep-Alive

1. **请求正文**

name = test&pwd=12345

1. **\r和\n的区别：**

* \r：回车，回到当前行的行首，不会换到到下一行
* \n： 换行，换到下一行的当前位置，不会回到行首。

windows下：所有行的结尾都是：\r\n

1. **手写服务器：**

**网页输入（GET）**：<http://localhost:8888/chris?fav=1&fav=2&uname>=洁&age=18&others=

**分析：**http://IP：端口/URL？请求参数；

* **手写服务器内容包含的内容：**接收浏览器请求内容（Request类），反馈数据给浏览器（Response类）；
* Response类：服务器返回给浏览器的数据——后端人员干的事情

1. 响应行 + 响应头；
2. 响应内容（结尾需要换行）；
3. 拼接：响应行 + 响应头 + 响应内容，转成字符格式；
4. 使用输出流，输出到客户端。

* Request类：服务器获取浏览器发送来的信息

1. 接收client的请求信息；
2. 解析请求信息的步骤：

* GET方法中：分离URL、请求参数 （两者用？分割）；
* POST方法中（不含URL，只有请求参数）：去除&；
* 获取方法、URL、请求参数；
* 请求参数分离：使用HashMap保存分离的键值对，以便使用；

1. **servlet：**（server applet）

* **概念：**java编写的服务器端程序，主要用于交互式浏览数据、修改数据、生成动态web内容。

**本质：**狭义的servlet是一个接口，广义的servlet指的是**实现**了某个**servlet接口的类**。

**应用：**servlet只用来扩展基于HTTP协议的Web服务器。







# Annotation注解

1. Annotation概念：

* JDK5.0引入；
* 作用：对程序做出解释，但不是程序——和注释一样的效果；

可以被其他程序读取，注解信息处理流程 ；

* 格式：@注解名，可以添加参数；

eg：@SupperssWarnings(value=”unchecked”)；

* 使用范围：附加在package、class、method、field中，通过反射机制访问这些数据。
* 常见的内置注解：
* **@Override**：重写
* **@Deprecated**：不建议使用的方法，调用对应方法时会出现删除线
* **@SupperssWarnings**：抑制编译时的警告信息

eg：@SupperssWarnings(“all”)：抑制所有警告信息

* 元注解：用于注解其他注解
* **@Target：**指出被描述的注解，可用于什么地方；

**取值：**PACKAGE、TYPE、CONSTRUCTUOR、FIELD、METHOD、LOCAL\_VALRIABLE、PARMETER；

* **@Retention：**表示需要在什么级别保存该注释，用于描述注解的生命周期；

**取值：**SOURCE、CLASS、RUNTIME（运行时有效，可被反射机制读取）

* **@Documented**：

**取值：**

* **@Inherited**
* **注解，需要有注解解析的机制，才能够发挥作用——就是注解能够被其他程序读取，才有意义。**

1. 自定义注解：

**/\*\*\*\*\*\* 自定义注解 \*\*\*\*\*\*/**

**@Target**(value = {ElementType.**METHOD**, ElementType.**TYPE**})**//说明注解使用范围**  
**@Retention**(value = RetentionPolicy.**RUNTIME**) **//说明注解生命周期，RUNTIME运行时读取**  
public **@interface** **Annotation** {  
 **//类型、注解名、默认值**  
 String **studentName()** **default** **"";** int **age() default 0;**  
 **//默认值为负数，一般表示不存在**  
 int id() default -1;  
 String[] schools() default{"ECUST", "SUIU"};  
}

**/\*\*\*\*\*\* 使用自定义注解 \*\*\*\*\*\*/**

**@Annotation**  
public class Test {  
 **@Annotation**(age = 10, studentName = "chris", id = 242, schools = "ECUST")  
 public void test(){  
 }  
}

1. **ORM**（Object Relationship Mapping）：对象关系映射

* **作用**：使用注解，完成类和表结构的映射关系：即数据转化。

**流程：**java语言转化为SQL语言，生成SQL表格

* **示例代码：**解析注解，获取注解信息。

public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {

**//0、使用反射解析：StudentTable类注解设置为@Retention(RetentionPolicy.RUNTIME)**  
 Class cla = Class.***forName***("Annotation.ParseAnnotation.StudentTable");  
  **// 1、获取所有类的注解**

Annotation[] annotations = cla.**getAnnotations**();  
 for(Annotation a : annotations){  
 System.*out*.println(a);  
 **// 2、获取指定类的注解: 解析类对应的注解**  
 TableAnnotation tableAnnotation = (TableAnnotation) cla.getAnnotation(TableAnnotation.class);  
 System.*out*.println(tableAnnotation.value());  
 **// 3、获取类属性的注解:这里只获取一个，解析属性对应的注解**  
 Field field = cla.getDeclaredField("studentName");  
 ParaAnnotation paraAnnotation = field.getAnnotation(ParaAnnotation.class);  
 System.*out*.println(paraAnnotation.paraName() + "--" + paraAnnotation.type() + "--" + paraAnnotation.length());  
 }  
}

1. **JAVA动态特性: Java动态的实现，使用字节码、反射**

* **动态语言**
* Python、ruby、javascript（解释型语言）
* 在程序运行时，可以改变程序结构、变量类型.
* **反射机制: reflect，会降低程序运行的速度**
* 程序加载、运行时，可以使用未知的类、数组、枚举类型、注解、interface等等。

加载完类之后可以使用这个类的所有属性、方法，**同样的类只能被反射一次**；简单的来说，JVM会将创建一个对应的Class对象。

* **反射：**Class cla = Class.forName(“cc.User”);
* **基本方法：**
* **反射类：**Class.forName(path)；
* **实例化反射类：**newInstance()——**实例化之后的对象可以使用该类所有的方法、属性；**
* **获取类名：**类名+包名getName()、类名getSimpleName()；
* **获取类的属性：**所有属性getDeclaredField()；

public属性getDeclaredFields()；

* **获取类的方法**：所有方法getDeclaredMethods()；

某个方法，可以有参数getDeclaredMethod()；

* **获取类的构造器：**getDeclaredConstructors()；
* **示例代码**：

**//等待反射的类**

package Reflection;

public class **User** {  
 private int id;  
 private int age;  
 private String username;  
 public int getId() {return id;}  
 public void setId(int id) {this.id = id;}  
 public int getAge() {return age; }  
 public void setAge(int age) {this.age = age;}  
 public String getUsername() {return username; }  
 public void **setUsername**(**String username**) {this.username = username;}  
 public void **setUname**() {this.username = "Chris";}  
 public User(int id, int age, String username) {  
 super();  
 this.id = id;  
 this.age = age;  
 this.username = username;  
 }  
 **//javabean必须要有无参的构造方法！**  
 public User() {}  
}

**//反射的应用**

public static void main(String[] args) {  
 String path = "Reflection.User";  
 try {  
 **Class<User>** reflectClass = **(Class<User>)** **Class.forName**(path);**//类路径**  
 **//通过反射API，调用构造方法，实例化对象**  
  **//1、调用无参构造器：实例化类**  
 **User** u1 = reflectClass.**newInstance();//u1、u2…都可以使用反射类的方法**  
 System.*out*.println(u1);  
 **//2、调用有参构造器**  
 Constructor<User> c = reflectClass.**getDeclaredConstructor(int.class,int.class,String.class);**  
 User u2 = c.newInstance(10001,24,"Chris");  
 System.*out*.println(u2);

**//反射API，调用普通方法**  
 User u3 = reflectClass.newInstance();  
 **Method** method = reflectClass.**getDeclaredMethod("setUsername", String.class); //方法名+参数类对象** method.**invoke**(u3,"Vicky"); **// 等同于 u3.setUname("Vicky");**  
 System.*out*.println(u3.getUsername());

**//反射API，操作属性**  
 User u4 = reflectClass.newInstance();  
 **Field** field = reflectClass.**getDeclaredField("username");//属性名**  
 field.**setAccessible**(true); **//禁止安全检查的开关，禁用：提升反射速度**

**//默认为false：只能调用public属性。**

**//true: 调用所有属性**

**//可以用于破解、调用私有属性、破解单例模式** field.**set**(u4,"Sunshine"); **//设置属性的值**  
 System.*out*.println(u4.getUsername());

**//等同于System.out.println(field.get(u4));**  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

* **动态编译**
* **应用：编译.java文件源码**

浏览器端编写java代码，上传到服务器编译、运行，服务器需要动态加载文件进行编译。

* **编译方式：**

RunTime；

反射;

* **示例代码：**

public static void main(String[] args) throws Exception {  
 String path = "./data/HelloWorld.java";  
 **/\*\*  
 \* 方法一：使用RunTime编译、运行  
 \*/**  
  **//1、编译**  
 **JavaCompiler** compiler = **ToolProvider.getSystemJavaCompiler();**  
 int result = compiler.**run**(null,null,null,path);  
 System.*out*.println(result);  
  **//2、执行**  
 **Runtime** runtime = **Runtime.getRuntime();**  
  **//操作 + 路径 + 文件名**  
 **Process** process = runtime.**exec**("java -cp ./data HelloWorld");

**//读取改文件运行后的输出内容**  
 InputStream in = process.getInputStream();  
 BufferedReader reader = new BufferedReader(new InputStreamReader(in));  
 String data = "";  
 while ((data = reader.readLine())!=null){  
 System.*out*.println(data);  
 }  
 **/\*\*  
 \* 方法二：使用反射编译、运行  
 \*/**  
 **URL**[] urls = new **URL[]**{new **URL**("file:/" + System.*getProperty*("user.dir") + "/data/")};

**//类加载器**  
 **URLClassLoader** loader = new URLClassLoader(urls);

**//加载类：返回一个示例化后的对象**  
 Class cla = **loader.loadClass**("HelloWorld");

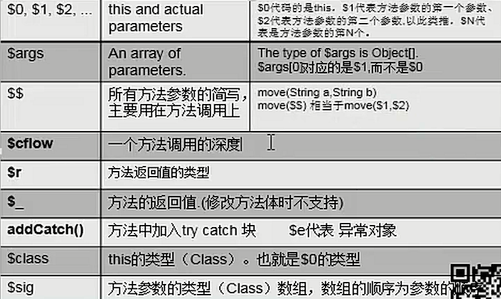
**//获取方法**  
 **Method** method = cla.getMethod("main", String[].class);

**//执行、调用方法**  
 method.**invoke**(null,(Object)new String[]{});  
}

* **动态执行javascript代码**
* **作用：**将功能写成脚本，是需要调用对应的脚本就可以实现对应功能。使用JAVA的脚本引擎执行；
* **示例代码：**

public static void main(String[] args) throws Exception {  
 **//获得脚本引擎对象**  
 **ScriptEngineManager** sem = new ScriptEngineManager();  
 **ScriptEngine** engine = sem.getEngineByName("javascript");  
  
  **//定义变量、存储到引擎**  
 engine.**put**("msg","Welcome to ECUST");  
 String str = "var user = {name:'chris',schools:['MIT','Harvard']};";  
 str += "print(user.name);";  
  
 **//执行脚本**  
 engine.**eval**(str);  
 engine.eval("msg = 'Hello World';");  
 System.*out*.println(engine.**get**("msg"));  
  
  **//定义函数：实现两数之和**  
 engine.eval("function add(a,b){var sum = a+b; return sum;}");  
  **//获取接口**  
 **Invocable** jsInvoke = **(Invocable)**engine;  
 **//执行脚本中的方法**  
 Object result = jsInvoke.**invokeFunction**("add", new Object[]{13,10});  
 System.*out*.println(result);  
  **//导入其他java包**  
 }

* **动态字节码操作**
* **作用：**使用java代码来编写类，生成.class文件，需要使用XJAD反编译才能得到对应的java文件，而不是自己创建java文件来填写类；
* **常见字节码库：**BCEL、ASM、CGLIB、Javassit（常见于开源框架中）；
* **Javassit使用:**
* **方法操作：**



* **示例代码：需要导入Javassit包**

public static void main(String[] args) throws Exception {  
  **//1、创建类：类池 + 类名**  
 **ClassPool** classPool = ClassPool.**getDefault**();  
 **CtClass** **ctClass** = classPool.**makeClass**("JavassitTestClass");  
  **//2、创建属性**  
 **CtField** ctField01 = CtField.**make**("private int id;",ctClass);  
 CtField ctField02 = CtField.*make*("private String name;",ctClass);  
 **ctClass.addField**(ctField01);  
 ctClass.addField(ctField02);  
  **//3、创建方法**  
 **CtMethod** ctMethod01 = CtMethod.**make**("public int getId(){return id;}", ctClass);  
 CtMethod ctMethod02 = CtMethod.*make*("public void setId(int id){this.id = id;}", ctClass);  
 **ctClass.addMethod**(ctMethod01);**//在对应类中，添加方法**  
 ctClass.addMethod(ctMethod02);  
  **//4、添加构造器**  
 **CtConstructor** ctConstructor = new CtConstructor(new CtClass[]{CtClass.*intType*, classPool.get("java.lang.String")},ctClass);  
 ctConstructor.**setBody**("{this.id = id; this.name = name;}");**//构造器内容**  
 **ctClass.addConstructor**(ctConstructor);**// 在对应类中，添加构造器**  
  **//5、保存Javassit创建的类为字节码文件.class，需用XJAD反编译得到.java文件**  
 **ctClass.writeFile**("./data/JavassitTest");  
 System.*out*.println("SUCCESS");  
}

# JVM核心机制

1. **JAVA文件说明：**

* **.class：**可执行文件，编译后的文件。cmd下的执行命令：java xxx
* **.java：**java文件，源码。cmd下的编译命令：javac –d . xxx.java

1. **类加载全过程**

* **类加载机制：**

JVM把.class文件加载到内存，进行数据校对、解析、初始化，得到JVM可以运行的Java类的过程。具体流程如下：

* **加载：**

类加载器->加载.class文件到内存->方法区存储运行时的数据、类对象；

* **连接：**
* 验证：保证加载类符合JVM规范；
* 准备：为static变量分配内存，存于方法区；
* 解析：虚拟机常量内存池的符号引用，替换为直接引用；
* **初始化：**
* 执行类的构造器，**一定会先初始化父类**，**初始化流程如下**：

**（先加载属性）static代码块 -> 代码块 -> 构造器;**

* 类的主动引用：一定会有类的初始化
* new一个对象；
* 调用类的静态成员、静态方法（除了final常量）；
* 使用反射方法，调用类；
* 启动main()方法的类；
* **初始化一个类时，一定会先初始化其父类**；
* 类的被动引用：不会有类的初始化
* 子类引用父类的静态变量时，不会初始化子类；
* 通过数组定义类的引用，不会初始化类；
* 引用常量不会初始化类；

1. **类加载器：**

* **类加载器原理**
* **类缓存：**

Java SE类加载器会查找已加载的类，并维持一段时间的缓存，避免用一个类的重复加载；

* **java.class.ClassLoader类：**
* 作用：根据类，找到或生成对应的字节码，然后从字节码中定义一个类——生成实例对象；也可记载资源、文件；
* 相关方法：
* getParent()：返回该类加载器的父类加载器；
* loadClass(String name)：加载名为name的类，返回一个实例对象；
* findClass(String name)：查找名为name的类；
* findLoadedClass(String name)：查找名为name的类，是否被加载过；
* defineClass(String name, byte[] b, int off, int len)：将字节数组b的内容转化为java类，返回final的实例对象；
* **类加载器树状结构、代理机制**
* **引导类加载器：**

**加载Java核心库：**jre/lib/tr.jar、sun.boot.class.path；

不继承java.lang.ClassLoader；

* **扩展类加载器：**

**加载Java扩展库：**jre/ext/\*.jar、java.ext.dirs；

**实现：**使用sun.misc.Launcher$ExtClassLoader；

* **应用程序类加载器：**

加载Java应用类路径中的内容：classpath、java.class.path；

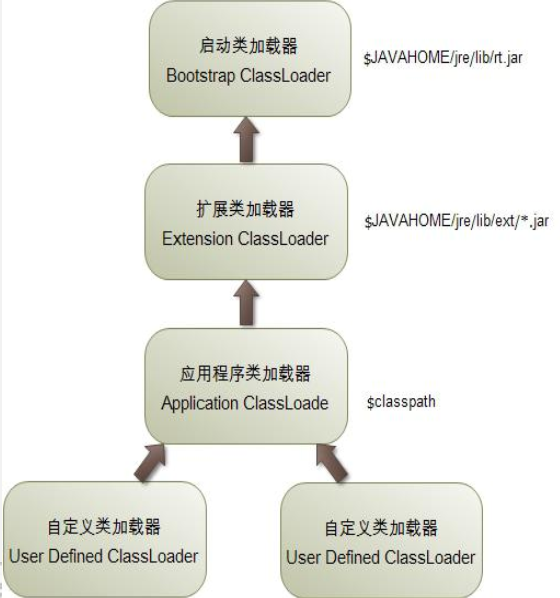
**实现：**使用sun.misc.Launcher$AppClassLoader

* **自定义类加载器：**

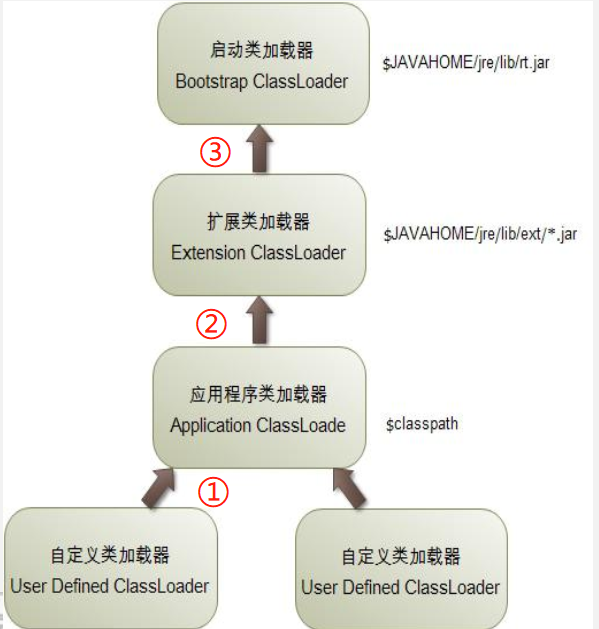
**实现：**使用java.lang.ClassLoader

* **四个加载器的树状结构：**

**这四个加载器是组合关系（上下级关系），不是继承关系；**



* **代理机制：**
* **双亲委托机制：**
* **作用：确保用户不能自己定义java.lang.Object类，即使用户定义这样的类，也无法使用。**
* **加载流程：**逐级寻问**上一级（父类）加载器**，询问是否可加载。若最上一级加载器无法加载，则逐级返回下一级加载器询问。若还是无法加载，则抛出异常。



* **其他机制：eg：Tomcat服务器**

先加载某个类，若不行，再询问**上一级（父类）加载器**。

* **自定义类加载器：文件、网络、加密**
* **加载流程：**
* 检查该类是否被加载：findLoadedClass(String classname)
* 没加载过：使用父类加载，父类加载成功，直接返回，加载失败进入下一步
* 使用该类的加载器方法findClass（已被重写）：

获取所加载类的字节码 -> 将字节码转换成java类defineClass()

* **文件类加载器：**

加载.class文件（可被执行的文件，想看源码需要使用XJAD反编译得到.java文件）

* **网络类加载器：**

和文件类加载器相似。但使用URL + inputStream读取类的字节码；

* **加密类加载器**：
* 取反操作，DES对称加密；
* **本质上：**就是将文件中读取的数据，进行取反操作，然后再将加密的数据输出即可。
* **java中取反操作：**

toBinaryString(data ^ 0xff); //数据转换成二进制，并取反

* **线程上下文加载器**
* **JAVA默认使用的加载机制为：双亲委派加载机制**(AppClassLoader)，可能会导致第三方jar包无法被正确加载，例如：JDBC；
* **示例代码：**

**//获取当前线程的加载器**  
System.*out*.println(Thread.*currentThread*().**getContextClassLoader**());  
**//改变线程的类加载器：loader01对象，是已经初始化后的自定义文件类加载器**  
Thread.*currentThread*().**setContextClassLoader**(loader01);   
System.*out*.println(Thread.*currentThread*().getContextClassLoader());

* **服务器类加载原理、OSGI**
* **OSIG：open service gateway initative**

面向java的动态模块系统，提供面向服务和基于组件的运行环境，每一个OSGI模块都对应的类加载器，每个模块只会调用自己的类加载器，即时被其他模块调用，也会自动调用自己模块对应的类加载器，保证每个模块的独立性。

* **应用：**

Eclipse就是用OSGI实现的

# 设计模式---GOF（Gang of Four）

1. **设计模式分类：**

* **创建模式：**
* **单例模式：**
* **作用：保证一个类只有一个实例，提供一个访问该实例的全局访问点。**
* **应用：**windows任务管理器、回收站、读取配置文件的类、数据库连接、Servlet中Application、spring MVC的控制器；
* **优点：**占用系统资源少，优化资源环共享访问，提供全局访问点；
* **常见的单例模式：**
* **饿汉式：**

不涉及多个线程对象访问该对象问题，JVM保证只会加载一次该类，无并发访问问题，可省略synchronized关键字；

public class SingletonDemo1 {  
  **//类初始化时，立即加载这个对象。加载类时，天然的是线程安全的！**  
 private static SingletonDemo1 ***instance***= new SingletonDemo1();   
 **//构造器私有化**  
 **private** SingletonDemo1(){  
 }  
 **//用户使用该方法获得实例。该方法没有同步，调用效率高！**  
 public static SingletonDemo1 getInstance(){  
 return **instance**;  
 }  
}

* **懒汉式：**

**延时加载：**在真正使用时，才加载类。

public class SingletonDemo2 {  
  **//类初始化时，不初始化该对象（延时加载：真正用的时候再初始化对象）**  
 private static SingletonDemo2 *instance*;  
 **//私有化构造器**  
 private SingletonDemo2(){   
 }  
 **//方法同步，调用效率低，需要等待线程调用完，才能再调用**  
 public static **synchronized** SingletonDemo2 getInstance(){  
 if(*instance*==null){  
 *instance* = new SingletonDemo2();  
 }  
 return *instance*;  
 }  
}

* **双重检测锁式：不使用，因为可能会出错**

使用if结合懒汉式 + 饿汉式，简化为只在第一次调用时创建一个对象，其余时候都是直接使用。

* **静态内部类式**：

使用静态内部类初始化对象，并且使用final保证值初始化一个对象。有延时加载。

public class SingletonDemo4 {  
 **//静态内部类**  
 private **static class** SingletonClassInstance {  
 private static **final** SingletonDemo4 *instance* = new SingletonDemo4();  
 }  
 **//私有化构造器**  
 private SingletonDemo4(){  
 }  
 **//方法没有同步，调用效率高！**  
 public static SingletonDemo4 getInstance(){  
 return **SingletonClassInstance.instance**;  
 }  
}

* **枚举单例：**

枚举类都是单例的，从JVM机制上提供保障。没有延时加载。

public **enum** SingletonDemo5 {  
 **//这个枚举元素，本身就是单例对象！**  
 *INSTANCE*;  
 **//添加自己需要的操作！**  
 public void singletonOperation(){  
 }  
}

* **反射、反序列化方式：破解除枚举之外的单例模式**

**使用反射：**获取类的反射对象，setAccessible(true)：访问私有属性、私有构造器。

**使用反序列化：**先序列化类的实例化对象，并保存为文件。再使用反序列化，从文件中读取，获得新的实例化对象。

* **选用单例模式：**

1. 需求：占用资源少、不需要延时加载。

选择枚举式，比饿汉式好

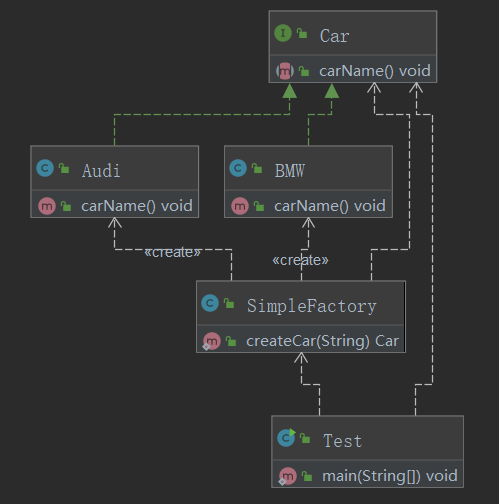
1. 需求：占用资源多，需要延时加载。

选择静态内部式，比懒汉式好（时间长）

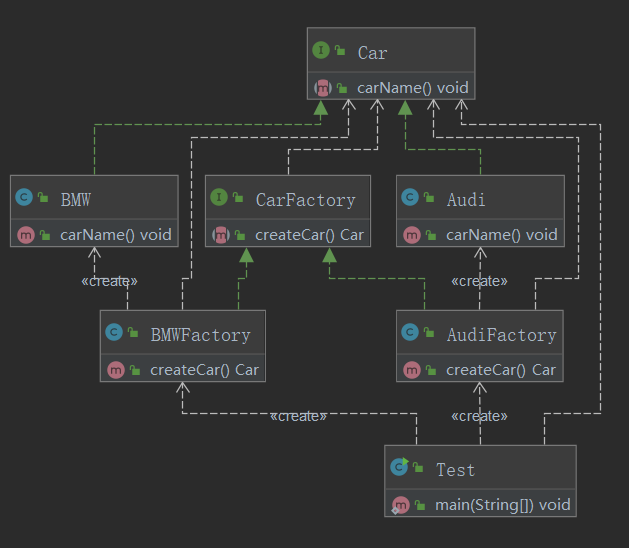
* **工厂模式Factory：**

**本质：**实例化对象，用工厂方法代替new操作。把选择实现类、创建对象统一管理，从而实现类之间的解耦。

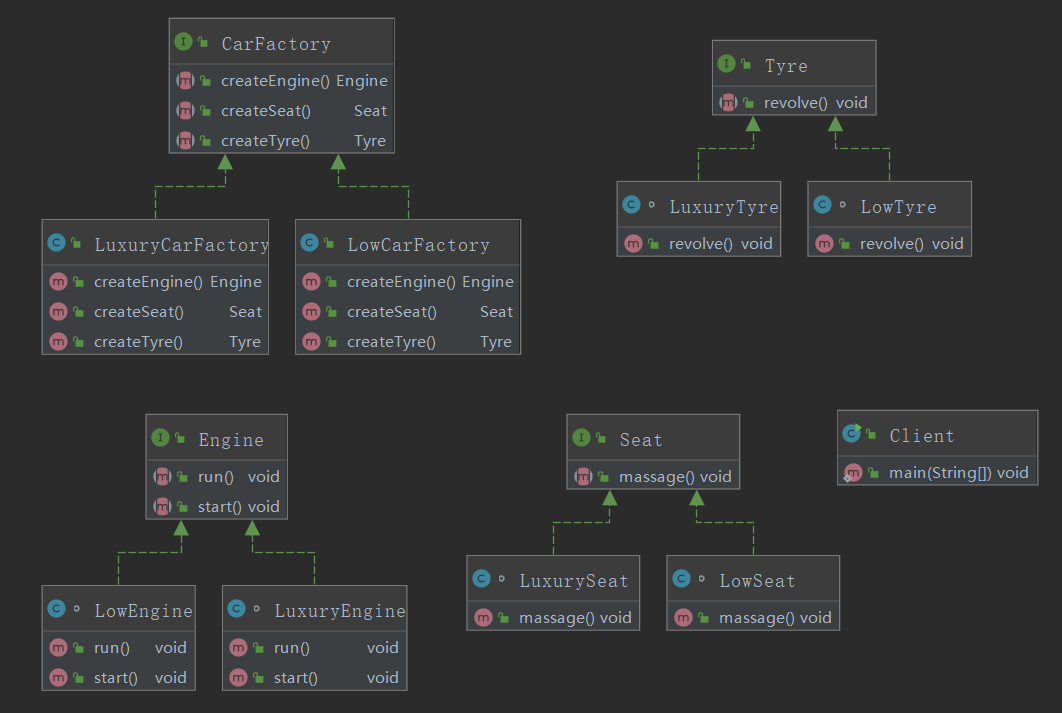
* **简单工厂模式：**
* **设计模式：**每个类继承类的接口。一个工厂中，实现所有类的初始化。
* **使用：**使用者只需知道，工厂名、接口的方法，即可调用相应的类。
* **问题：**想增加新的类，除了增加对应的类文件，还需要修改工厂类中的代码（实现新类的初始化），不符合开闭原则（OCP）



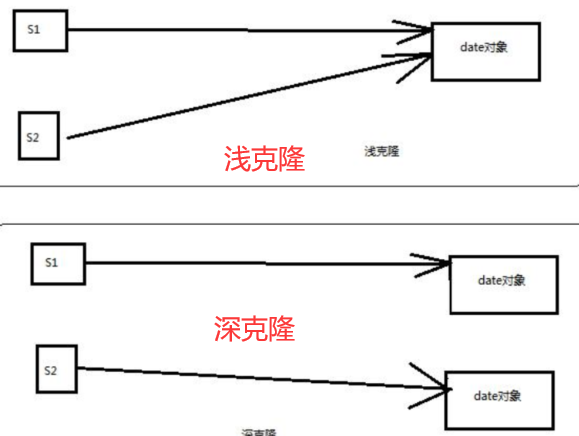
* **工厂方法模式：**
* **设计模式：**类继承类的接口，工厂继承工厂的接口。
* **使用：**使用者只需知道，各个类的工厂名、各类的方法即可调用相应的类；想增加新的类，只需增加对应的类文件、类的工厂文件。
* **问题：**每个类都要有自己的工厂，会增加文件中类的数量——因此，**一般都是用简单工厂模式。**



* **抽象工厂模式：**
* **设计模式：**适合一个产品簇的开发——产品中含有低、中、高等不同级别的产品
* 使用：
* 问题：



* **建造者模式Builder：常结合工厂模式使用**
* 设计模式：
* 使用：
* 问题：
* **原型模式Prototype：**
* 设计模式：使用克隆的概念（需要重写clone接口），将一个类进行复制
* 分类：
* 浅克隆：被复制对象的变量和原对象的变量有相同值，且指向原对象
* 深克隆（克隆属性）：将变量复制到新的对象，和原对象不同

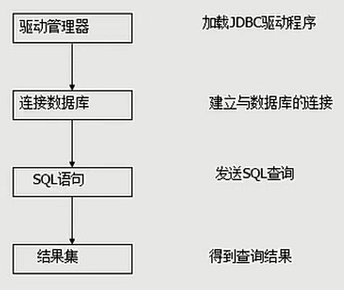


* **结构模式：**
* 适配器模式：
* 桥接模式：
* 装饰模式：
* 组合模式：
* 外观模式：
* 享元模式：
* 代理模式：
* **行为模式：**
* 模板方法模式：
* 命令模式：
* 迭代器模式：
* 观察者模式：
* 中介者模式：
* 备忘录模式：
* 解释器模式：
* 状态模式：
* 决策模式：
* 职责链模式：
* 访问者模式：



## JDBC

1. **访问数据库的基本流程：**



1. **JDBC连接数据库代码：**

public static void main(String[] args) {  
 RunTime runTime = new RunTime();  
 Connection connection = null;

Statement stmt = null;

try {  
  **//加载驱动类**  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 runTime.startProgram();  
  **//需要填写时区，否则报时区错误：UTC（Coordinated Universal Time）：调节世界时**  
 connection = DriverManager.*getConnection* ("jdbc:mysql://localhost:3306/dangdang?serverTimezone=UTC","chris","1230re0321re");  
 runTime.endProgram();

**//插入数据**

stmt = connection.createStatement();

String sql = "insert into table01 (id,name) value ('01','chris')";

stmt.execute(sql);

} catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
}

1. **Java访问Mysql时注意事项：**

需要mysql-java的驱动和mysql匹配，否则无法连接。

1. **常用接口：使用的是java.sql库，不是com.mysql.sql（仅负责驱动MySQL）**

* **继承关系：**

Statement（PreparedStatement） implements Connection

ResultSet implements Statement（PreparedStatement）

* Statement：使用createStatement发送简单的sql语句（执行sql语句），不带参数。可能存在注入的危险
* PreparedStatement：继承自Statement，发送含参数的sql语句（执行sql语句，**主要使用的方法）**
* CallableStatement：继承自PreparedStatement，用方法prePareCall创建，调用存储
* execute()：运行语句
* executeQuery()：运行select语句**（可用于返回查询语句的结果）**
* executeUpdate()：运行insert、update、delete操作，返回更新的条数
* addBatch（）：批量处理数据
* ResultSet：结果集，用于查找数据

1. **事务的基本概念：**

* 一组sql语句只能时同时执行成功、或者失败，即：将这些操作看成一个单元；
* **事务开始：**

连接数据库后，执行DML（data manipulation languages）：insert、update…

* **事务截至：**
* 执行commit、rollback（回滚）
* 执行DDL（data definition languages）：create table（自动执行commit）
* 执行DCL（data control languages）：grant…
* 断开数据库连接
* 执行DML失败时，默认前面的操作无效，且执行rollback（回滚）将数据还原。

1. **时间类型：**

* date：存储了时间+日期，但只显示日期
* timestamp：时间+日期，和时区有关
* datetime：时间+日期，和时区无关

1. **CLOB：character large object，大文本对象**

* 使用流对象来处理
* 数据类型：TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT

1. **BLOB：Binary large object，二进制文件处理**

* 可以存放图片，需要使用FileReader

1. **使用资源文件存储数据库连接信息**

* 使用.properties资源文件保存参数，方便后期修改参数，给文件应存放在源文件的根目录下。和xml类似
* **示例代码：**

资源文件db.properties内容：

mysqlDriver:com.mysql.cj.jdbc.Driver  
mysqlURL:jdbc:mysql://localhost:3306/jdbctest?serverTimezone=UTC  
mysqlUser:root  
mysqlPasswd:1230re0321re

调用方法：

static Properties ***properties***= null; **//资源文件操作对象**  
**//在类被加载时，初始化**  
static{  
 ***properties***= new Properties();  
  **//加载资源文件：资源文件放在根目录**  
 try {  
***properties.load***(Thread.*currentThread*().getContextClassLoader().getResourceAsStream("db.properties"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

public Connection MysqlConnection(){  
 try{  
 Class.*forName*(*properties*.***getProperty***("mysqlDriver"));  
 return DriverManager.*getConnection*(*properties*.***getProperty***("mysqlURL"),  
 *properties*.***getProperty***("mysqlUser"),  
 *properties*.***getProperty***("mysqlPasswd"));  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 return null;  
 } catch (SQLException e) {  
 e.printStackTrace();  
 return null;  
 }  
}

1. **java使用JDBC示例代码：**

public static void main(String[] args) {  
 Connection connection = null;  
 Statement statement = null;  
 PreparedStatement preparedStatement = null;  
 ResultSet resultSet = null;  
 try {  
  **//1、加载驱动类：**   
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 **//此处连接需要填写时区，否则会报时区错误：UTC调节世界时** connection = DriverManager.*getConnection* ("jdbc:mysql://localhost:3306/jdbctest?serverTimezone=UTC","chris","1230re0321re");  
 connection.setAutoCommit(false); **//默认为true，手动提交数据：需用commit，才能存数据**

**//2、插入数据: 方式一**  
 statement = connection.createStatement();  
 String sql = "insert into table01 (id,name) value ('2','chris')";  **//插入的值，必须使用char、string类型** statement.execute(sql);

**//3、插入数据: 方式二**  
 sql = "insert into table01 (id,name) values (?,?)"; **//?为占位符：需要后面填入参数**  
 preparedStatement = connection.prepareStatement(sql);  
 preparedStatement.**setString**(1,"4"); **//参数索引从1开始（第一个？），不是0**  
 preparedStatement.**setObject**(2,"Alice"); **//setObject不需要参数类型（第二个？）**  
 preparedStatement.**execute**(); **//执行sql语句**  
  
  **//4、查找数据**  
 sql = "select id,name from table01 where id>?";  **//查找的sql语句**  
 preparedStatement = connection.prepareStatement(sql); **//获取对象**  
 preparedStatement.setObject(1,2); **//把id>2的数据记录下来**  
 resultSet = preparedStatement.**executeQuery**();  **//获取查找的数据**  
 while (resultSet.next()){ **//打印第1、2列数据**  
 System.*out*.println(resultSet.getObject(1)+"--"+resultSet.getObject(2) );  
 }

**//5、批量添加数据**  
 for(int i = 0; i < 100; i ++){  
 preparedStatement.**addBatch**("insert into table01 (id,name) values (**"+i+"**,'chris**"+i+"**')"); **//添加变量时，使用"+参数名+"**  
 }  
 preparedStatement.executeBatch();  
 connection.commit(); **//若前面关闭自动提交，则需手动提交数据，否则数据不会存入数据库**  
  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }finally{  
 **//5、关闭操作：**

**顺序：result-->statement-->connection，需要将try，catch分开（防止关闭时出错）**  
 try {  
 if(resultSet != null){  
 resultSet.close();   
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }

…………

}

1. **ORM（object relationship mapping）对象关系映射**

* **思想：**
* 表结构和类对应；表中字段和类的属性对应；表中记录和对象对应；
* javabean的属性名和类型，与数据库一致；
* 一条记录对应一个对象，使用容器（List、Map、Set）存放查找到的对象



# 补充知识

1. **serialVersionUID：**

* **作用：**

serialVersionUID是实现Serializable接口而来的，应用于对象于反序列化、序列化。

serialVersionUID**控制每个序列化类版本**的标识，保证在反序列化时，发送方、接收方是可兼容的对象。当serialVersionUID不同时，接收方会抛出异常。

**对象序列化的用途：**

* 将对象序列化成字节码、保存到存储介质：如硬盘等
* 用于网络传输
* **使用：**
* 一般来说，serialVersionUID是private static final long的变量
* 其数值为：1L、2L、3L….**数字越大，标识该版本越高，且不会向下兼容**
* **例程：**

public class Person {  
 private static final long *serialVersionUID* = 1L; **//设定版本**

……

}

1. 输出函数的区别：

* printf：C语言的方式，可以格式化输出

eg：System.out.printf(“number = d”, num); //d，整数输出。f：浮点数

* println：输出换行
* print：输出不换行

1. **方法参数Object … args**：

… ：表示任意个数的args参数，在调用该方法时，可以在这个位置传入多个参数。可用于，当参数个数不确定时

eg：

public String getAll(String sql, Object **... args**){

}

//调用该方法时，**此处传入多个参数**

getAll(sql, **ce.getName(), ce.getTel()**);

**常用函数**

1. format：

String.format用于创建格式化的字符串，连接多个字符数据，和c语言的sprintf功能类似。

eg：

public String toString() {  
 return String.*format*("%d->%d %.2f", v, w, weight);  
}

1. substring:

作用：用于字符串类型数据，返回从第i到第j的元素。

eg：

String str = new String(“01234567”);

System.out.println(str.substring(3,5));

结果：345

应用：可以结合compareTo使用。

// 判断字符串v，w的第d位元素的大小

private static boolean less(String v, String w, int d) {  
 return v.substring(d).compareTo(w.substring(d)) < 0;  
}

1. string类的compareTo方法：只会从左到右的访问字符串种的所有字母，当首字母不同时，不会访问第二个字母了。
2. **List中包含ArraytList**

**关于ArrayList的使用：**

**addAll**：添加元素到数组的最后面

**removeAll**：从数组中删除指定的元素

static class Cinema{  
 List<Integer> available;

**// 删除available中的seats元素**

public boolean bookTicket(List<Integer> seats){  
 System.*out*.println("SEAT LEFT: " + available);  
 List<Integer> copy = new ArrayList<Integer>();  
 **// 将avaliable的元素添加到copy数组中**  
 copy.addAll(available);  
 **// 从copy中删除seats中包含的元素**  
 copy.removeAll(seats);

available = copy;

}

}

1. 关于null ！= str 和str ！= null的区别：
2. **null ！= str：**这种写法永远不会出现空指针异常，建议这样写。就算str为空，还是会返回false。

eg：“123”.equals.(str)

1. **null ！= str：**有可能str不存在，产生异常。str为空，就会产生空指针异常。

1. 关于throw和try catch的区别：
2. **throw：**将本函数的异常抛出，扔给上一层处理。一个函数可以throw多个异常。
3. **try catch：**运行某几行代码，可能产生异常，并且将异常留在该函数内，可用于在该函数内部处理异常。

1. .trim() ：用于删除字符串的头尾空白符
2. **split：去除字符串中的某些字符**

根据给定的正则表达式，去除某些字符。

**注意：| . \* ? 等特殊字符，必须在前面加上\\**

eg：

String str2 = new String("www.runoob.com");

System.out.println("转义字符返回值 :" );

for (String retval: str2.**split**("\\.", 3)){

System.out.println(retval);

}

结果：

www

runoob

com

1. **.getBytes()**  ，[**String**](mk:@MSITStore:D:\Course\尚学堂JAVA\课堂笔记和资料\jdk%20api%201.8_google\jdk%20api%201.8_google.CHM::/java/lang/String.html#String-byte:A-int-int-)(byte[] bytes, int offset, int length, String charsetName)

（1）.getBytes():编码函数，将字符串转成字节数组

（2）[String](mk:@MSITStore:D:\Course\尚学堂JAVA\课堂笔记和资料\jdk%20api%201.8_google\jdk%20api%201.8_google.CHM::/java/lang/String.html#String-byte:A-int-int-)(byte[] bytes, int offset, int length,String charsetName)：

解码函数，将字节数组转换成字符串，可以指定编码方式

String msg = "chris";  
// 编码  
byte[] dates = msg.getBytes();  
System.*out*.println(dates.length);  
//解码  
msg = new String(dates,0,dates.length);  
System.*out*.println(msg);

1. setChartAt：

是StringBuilder、StringBuffer的方法，用于在指定位置添加元素。

public void printList(){

**//初始化时可以添加元素**  
 **StringBuilder** stringBuilder = new StringBuilder("[");

**//在指定位置添加字符’]’**  
 stringBuilder.setCharAt(stringBuilder.length() - 1, ']');  
 System.*out*.println(stringBuilder);  
}

1. RuntimeException：

抛出异常，可在运行程序时，手动抛出异常。

throw new **RuntimeException**("填写输出内容");

1. Class.forName()、newInstance()：

* **Class.forName(String className)：**

**概念：**加载类的静态方法 ，返回一个类名为“className内容”的类。

也可以放入包名.类名，加载一个类——获得该类的所有信息。

**作用：**常用于加载数据库驱动，用于选择不同名字的驱动（为了不改代码，实现调用不同驱动对象）；

* **newInstance()：**

实例化对象，调用所反射类的默认构造器（无参构造器）；

* **getConstructor()：**

获取构造器对象，并调用其newInstance()方法创建对象；

* **示例代码：**

Class clz;  
try {  
  **//得到一个类名为“className内容”的类**  
 clz = Class.*forName*(className);  
  **/\*创建一个对象：**

**1、getConstructor：获得其对应的构造器  
 2、newInstance（无参数）：调用其实例化方法，实例化对象（servlet对象）  
 3、这里实现了，根据URL获得的servlet-class，去实例化、调用对应的servlet**

**（Servlet是一个接口，这里实现了几个servlet对象）**  
  **\*/**  
 Servlet servlet =(Servlet)clz.getConstructor().newInstance();  
 **//返回对应的servlet对象，以便后期调用**  
 return servlet;

} catch (Exception e) {  
}



# 基本算法

1. **二分法查找：(**折半检索**)**

条件：所有的数据必须是有序的（按照大小排列）：

1. 递归方式实现：

**public** **static** **int** rank(**int** key, **int**[] a) {

**return** *rank*(key, a, 0, a.length - 1);

}

**public** **static** **int** rank(**int** key, **int**[] a, **int** lo, **int** hi) {

**if**(lo > hi)

**return** -1;

**int** mid = lo + (hi - lo) /2;

**if**(key < a[mid])

**return** *rank*(key, a, lo, mid - 1);

**else** **if** (key > a[mid])

**return** *rank*(key, a, mid + 1, hi);

**else**

**return** mid;

}

1. while方式实现：

**int** lo = 0;

**int** hi = a.length;

**int** mid = 0;

**while**(lo <= hi){

mid = lo + (hi - lo) / 2;

**if**(key < a[mid])

hi = mid - 1;

**else** **if**(key > a[mid])

lo = mid + 1;

**else**

**break**;

}

System.***out***.println("result = " + mid);