

# Java

[01\\_ Java入门第一课：课程介绍](#)  bilibili

## Java基础

### Java环境配置与概述

#### 1. DOS常用命令：

D: 切换到D盘

dir 查看当前路径文件信息

cd 进入目录 (在窗口中鼠标右键为粘贴)

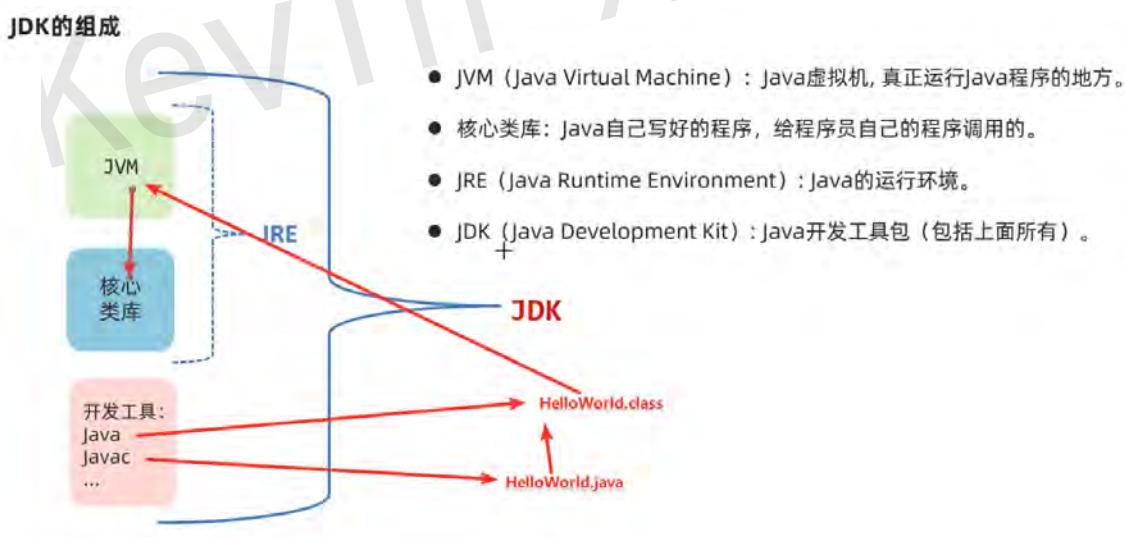
#### 2. Java程序开发步骤

编写代码——>.java源代码文件——>使用javac编译——>.class字节码文件——>使用java运行

用命令行编译代码：javac 文件名.java

用命令行运行代码：java 文件名

#### 3. JDK的组成



#### 4. Path环境变量：用于记住程序路径，方便在命令行任意目录直接启动程序

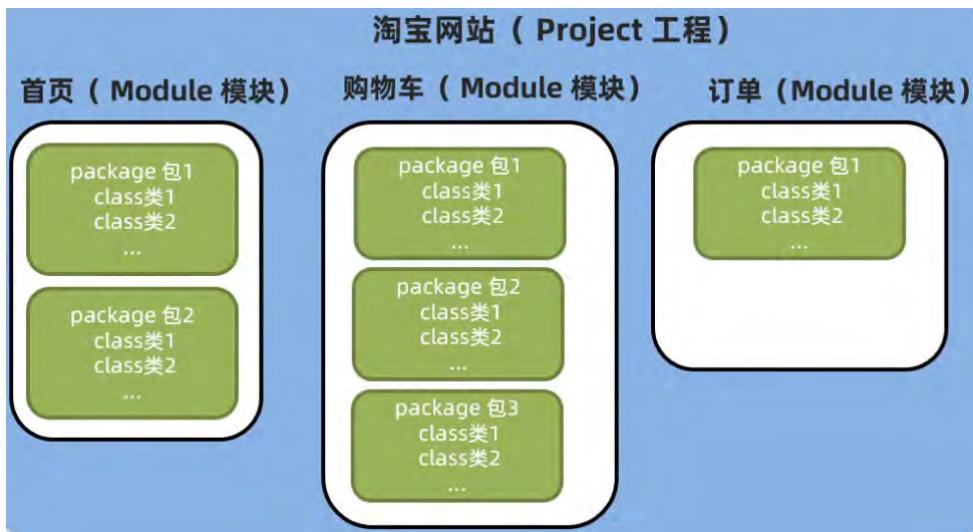
Path环境变量位置：此电脑——>属性——>高级系统设置——>高级——>环境变量

对于版本较低的JDK，需要手动将JDK的bin包配置到环境变量中

JAVA\_HOME：告诉操作系统JDK安装在哪个位置

JAVA\_HOME配置：在环境变量中新建，放置路径为JDK的位置

5. IDEA管理Java程序的结构: project——>module (文件夹) ——>package (文件夹) ——>class (文件)



先建立空project，创建新module，在src下新建包（包的名字不能是关键字，不然建立的是dictionary），在包下建类

IDEA编译后的class文件放在out文件夹中

## 6. IDEA快捷键

快捷键	功能效果
main/psvm、sout、...	快速键入相关代码
Ctrl + D	复制当前行数据到下一行
Ctrl + Y	删除所在行，建议用Ctrl + X
Ctrl + ALT + L	格式化代码
ALT + SHIFT + ↑, ALT + SHIFT + ↓	上下移动当前代码
Ctrl + /, Ctrl + Shift + /	对代码进行注释（分别是单行和多行）
数组名.fori	对数组使用for循环
Alt + Enter	选中一个未定义的方法，可以创建方法；也可以继承重写方法和实现接口
Ctrl + Alt + t	选中一段代码，可以为其添加外层嵌套（如while）

修改模块/类名称：右键refactor——>rename

导入模块：File——>new——>module from existing resources，输入模块路径，选择iml点击（有黑点的文件）

删除模块：在IDEA中右键remove module，再到文件夹删除模块

## 基础语法

### 注释

1. 单行注释 //
2. 多行注释 /\* \*/
3. 文档注释 /\*\* \*/，可以提取到文档说明器中

## 数据存储方式

1. 字符：ASCII编码的二进制表示
2. 图片：由像素点组成，每个像素点用三元组表示
3. 声音：给波形图建立坐标，波形图上每个点有二进制表示

## 数据类型

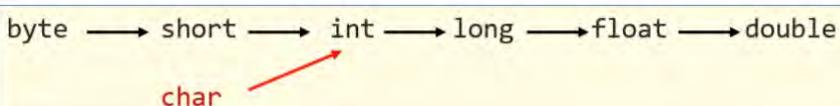
数据类型	内存占用(字节数)	数据范围
整型	byte	1 -128~127
	short	2 -32768~32767
	int(默认)	4 -2147483648~2147483647 (10位数，大概21亿多)
	long	8 -9223372036854775808 ~ 9223372036854775807 (19位数)
浮点型(小数)	float	4 1.401298 E -45 到 3.4028235 E +38
	double (默认)	8 4.9000000 E -324 到 1.797693 E +308
字符型	char	2 0-65535
布尔型	boolean	1 true, false

当希望随便写的数据为long，在数据后加L

## 类型转换

1. 自动类型转换

类型范围小的变量，可以直接赋给类型范围大的变量



2. 表达式的自动类型转换

表达式的最终结果类型由表达式中最高类型决定；

在表达式中，byte、short、char是直接转化为int参与运算的

3. 强制类型转换

强行将类型范围大的变量、数值赋值给类型范围小的变量

数据类型 变量2 = (数据类型)变量1、数据

## 运算符

1. 数据类型 变量1 += 变量2； 表达式的最终类型为变量1的类型
2. 在Java中，与是&，或是|，异或是^，&&是短路与，||是短路或
3. 三元运算符：条件表达式？值1：值2

## 流程控制

### 1. switch注意事项

- (1) 表达式类型只能是byte、short、int、char、String
- (2) case给出的值不允许重复，且只能是字面量，不能是变量
- (3) 穿透性：没有break会顺序执行

## 数组

### 1. 静态初始化数组

完整形式：数据类型[] 数组名 = new 数据类型[] {元素1, 元素2, .....};

简化形式：数据类型[] 数组名 = {元素1, 元素2, .....};

也可以写：数据类型 数组名[] = {元素1, 元素2, .....};

### 2. 动态初始化数组

定义数组时先不存入具体的元素值，只确定数组存储的数据类型和数组的长度

数据类型[] 数组名 = new 数据类型[长度];

动态初始化时数组元素默认初值为0

```
int a = 20; //a存在栈中
int[] arr = new int(20); //arr存储数组基地址，在栈中；new出来的数据元素存在堆中
```

3. 如果某个数组变量存储的地址是null，那么该变量将不再指向任何数组对象。此时可以输出这个变量，但不能用这个数组变量去访问其中的数据或者访问数组长度。

# 方法

方法是一种语法结构，它可以把一段代码封装成一个功能，以便重复调用。

## 1. 方法的完整格式

```
修饰符 返回值类型 方法名(形参列表){  
    方法体代码（需要执行的功能代码）；  
    return 返回值；  
}  
大多数方法的修饰符为 public static
```

方法不能定义在另一个方法里面。

## 2. 参数传递

基本类型和引用类型（数组、String等）的参数传递都是值传递，但基本类型参数传输存储的参数值，引用类型的参数传输存储的地址值。

## 3. 方法重载

一个类中，出现多个方法的名称相同，但是它们的形参列表是不同的，则称这些方法为方法重载。

形参列表不同：形参的个数、类型、顺序不同，不关心形参的名称。

方法重载应用场景：处理一类业务，提供多种解决方案。

## 4. return关键字

在无返回值的方法中，使用return可以立即跳出并结束当前方法的执行。

# 输入与随机数

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt(); //接受一个整数  
  
Random r = new Random();  
int i = r.nextInt(33) + 1; //生成1-33的整数
```

# 面向对象

## 面向对象入门

### 1. 如何理解对象

对象实质上是一种特殊的数据结构，可以理解为一张表，表当中记录什么数据，对象就处理什么数据。数据表中的数据由类来设计。

```
public class 类名{  
    属性;  
    方法;  
}  
类名 对象名 = new 类名();
```

## 2. 对象在计算机中的执行原理

```
Student s1 = new Student();  
每次new Student(), 就是在堆内存中开辟一块内存区域代表一个学生对象;  
s1变量里记住的是学生对象的地址;  
访问属性时, 通过s1记住的地址访问堆内存; 访问方法时, 由于堆内存记录类的地址,  
通过s1地址访问堆内存, 再通过堆内存中的类地址访问类中定义的方法
```

3. 一个代码文件中, 可以写多个class类, 但只有一个可以用public修饰, 且public修饰的类名必须为代码文件名。

## 4. this关键字

this是一个变量, 可以用在方法中, 来拿到当前对象。

应用场景: 解决变量名称冲突问题。

## 5. 构造器

```
public class Student {  
    //构造器, 也能重载  
    public Student() {  
  
    }  
    //有参数构造器  
    public Student(String name, double score) {  
        this.name = name;  
        this.score = score;  
    }  
}  
//创建对象时, ()即调用构造器  
Student s1 = new Student(); //调用无参数构造器  
Student s2 = new Student("xzs", 100); //调用有参数构造器
```

- (1) 类在设计时, 如果不写构造器, Java会为类自动生成一个无参数构造器
- (2) 一旦定义了有参数构造器, Java不会自动生成无参数构造器, 此时需要自己实现无参数构造器

## 6. 封装

封装，就是用类设计对象处理某一个事物的数据时，应该把要处理的数据，以及处理数据的方法，都设计到一个对象中去。

被private修饰的变量或者方法，只能在本类中被访问。

一般在设计一个类时，会将成员变量隐藏，然后把操作成员变量的方法对外暴露。常常设置赋值和取值两个方法，若想赋值使用方法set，取值用方法get。对于成员方法，看是否需要在外访问，若需要则暴露。

```
public class Student{
    private double score;
    //将方法暴露
    public void setScore(double score){
        if (score >= 0 && score <= 100)
            this.score = score;
    }

    public void getScore(){
        return score;
    }
}
```

## 7. 实体类：实体JavaBean

实体类就是一种特殊的类，它需要满足下面的要求：

- (1) 这个类中成员变量都要私有，并且要对外提供相应的set、get方法。
- (2) 类中必须要有一个公共的无参的构造器。

IDEA中，右键——>Generate——>getter and setter可以自动生成get、set函数，按住shift可以多选变量

IDEA中，右键——>Generate——>Constructor可以自动生成构造器，按住shift可以多选变量

实体类特点：仅仅用来保存数据，可以用它创建对象保存某个事物的数据。

实体类应用场景：实体类只负责数据存取，而对数据的处理交给其他类完成，以实现数据和数据业务处理相分离。

## 8. 成员变量与局部变量的区别

区别	成员变量	局部变量
类中位置不同	类中，方法外	常见于方法中
初始化值不同	有默认值，不需要初始化赋值	没有默认值，使用之前必须完成赋值
内存位置不同	堆内存	栈内存
作用域不同	整个对象	在所归属的大括号中
生命周期不同	与对象共存亡	随着方法的调用而生，随着方法的运行结束而亡

## static关键字

1. static叫静态，可以修饰成员变量与成员方法。

static修饰成员变量

```
public class Student{
    //类变量，属于类，在计算机中只有一份，会被类的全部对象共享（在堆内存）
    static String name; //类变量一般是public

    //实例变量，属于每个对象
    int age;
}
```

对于类变量的访问，一般使用 类名.类变量

```
Student.name = "xzs";
```

对于实例对象的访问，使用 对象.实例变量

```
Student s1 = new Student();
s1.age = 22;
```

应用场景：在开发中，如果某数据只需要一份，且希望能被共享（访问、修改），则该数据可以定义成类变量。

当访问自己类中的类变量，可以省略类名不写。

2. static修饰成员方法

```
public class Student{
    //类方法，属于类
    public static void print(){

    }

    //实例方法，属于每个对象
    public void print2(){
}
```

```
    }  
}
```

对于类变量的访问，一般使用 类名.类变量  
`Student.print();`

对于实例对象的访问，使用 对象.实例变量  
`Student s1 = new Student();  
s1.print2();`

main方法：是个类方法，当执行程序时，虚拟机直接调用该类方法。

类方法的应用场景：设计工具类，工具类中的方法都是类方法，每个方法用于完成某个功能。由于工具类没有创建对象的需求，建议将工具类的构造器进行私有。

### 3. static注意事项

- (1) 类方法中可以直接访问类的成员，不可以直接访问实例成员。
- (2) 实例方法中既可以直接访问类成员，也可以直接访问实例成员。
- (3) 实例方法中可以出现this关键字，类方法中不可以出现this关键字。

### 4. 代码块

代码块是类的五大成分之一（成员变量、构造器、方法、代码块、内部类）

代码块分为两种：静态代码块和实例代码块

- (1) 静态代码块（自己开发用的比较少）

格式：`static{}`

特点：类加载时自动执行，由于类只会加载一次，所以静态代码块也只会执行一次。

作用：完成类的初始化，如对类变量的初始化赋值。

- (2) 实例代码块

格式：`{}`

特点：每次创建对象时，执行实例代码块并在构造器前执行。

作用：和构造器一样，用来完成对象的初始化，如对实例变量的初始化赋值。

```
public class Student {  
    static int number = 80;  
    static String schoolName = "zju";  
    // 静态代码块  
    static {  
        System.out.println("静态代码块执行了~~");  
    }  
}
```

```

        schoolName = "zju";
    }

}

public class Student{
    //实例变量
    int age;
    //实例代码块：实例代码块会执行在每一个构造方法之前
    {
        System.out.println("实例代码块执行了");
        age = 18;
        System.out.println("有人创建了对象：" + this);
    }

    public Student(){
        System.out.println("无参数构造器执行了");
    }

    public Student(String name){
        System.out.println("有参数构造器执行了");
    }
}

```

## 继承

### 1. 继承定义与特点

```

//A类称为父类，B类称为派生类
public class B extends A{

}

```

继承特点：子类能继承父类的非私有成员（变量和方法）。

继承后对象的创建：子类的对象是由子类、父类共同创建的。

```

public class A{
    //公开的成员
    public int i;
    public void print1(){
        System.out.println("==print1===");
    }

    //私有的成员
    private int j;
    private void print2(){

}

```

```

        System.out.println("==print2==");
    }

}

public class B extends A{
    public void print3(){
        //由于i和print1是属于父类A的公有成员，在子类中可以直接被使用
        System.out.println(i); //正确
        print1(); //正确

        //由于j和print2是属于父类A的私有成员，在子类中不可以被使用
        System.out.println(j); //错误
        print2();
    }
}

public class Test{
    public static void main(String[] args){
        B b = new B();
        //父类公有成员，子类对象是可以调用的
        System.out.println(i); //正确
        b.print1();

        //父类私有成员，子类对象时不可以调用的
        System.out.println(j); //错误
        b.print2(); //错误
    }
}

```

## 2. 权限修饰符

修饰符	在本类中	同一个包下的其他类里	任意包下的子类里	任意包下的任意类里
<b>private</b>	√			
<b>缺省</b>	√	√		
<b>protected</b>	√	√	√	
<b>public</b>	√	√	√	√

3. 单继承：Java是单继承的，不支持多继承，但是支持多层继承（即继承传递）。
4. Object类：Java中所有类的祖宗。
5. 方法重写

当子类觉得父类方法不好用，或者无法满足父类需求时，子类可以重写一个方法名称、参数列表一样的方法，去覆盖父类的这个方法，这就是方法重写。重写后，方法的访问遵循就近原则。

方法重写的注意事项：

(1) 重写的方法上面，可以加一个注解@Override,用于标注这个方法是复写的父类方法。

```
public class B extends A{  
    @override  
    public void print(){  
  
    }  
}
```

(2) 子类复写父类方法时，访问权限必须大于或者等于父类方法的权限。

(3) 重写的方法返回值类型，必须与被重写的方法返回值类型一样，或者范围更小。

(4) 私有方法、静态方法不能被重写，如果重写会报错。

实际上我们实际写代码时，只要和父类写的一样就可以（总结起来就8个字：声明不变，重新实现）

## 6. 子类中访问成员的特点

原则：就近原则

super关键字：如果子类和父类出现同名变量或者方法，优先使用子类的；此时如果一定要在子类中使用父类的成员，可以加this或者super进行区分。

```
public class Z extends F {  
    String name = "子类名称";  
  
    public void showName(){  
        String name = "局部名称";  
        System.out.println(name); //输出“局部名称”  
        System.out.println(this.name); //输出“子类名称”  
        System.out.println(super.name); //输出“父类名称”  
    }  
  
    @Override  
    public void print1(){  
        System.out.println("==子类的print1方法执行了=");  
    }  
  
    public void showMethod(){  
        print1(); //调用子类的函数  
        super.print1(); //调用父类的  
    }  
}
```

```
}
```

## 7. 子类中访问构造器的特点

子类全部构造器，都会先调用父类构造器，再执行自己。

子类默认调用父类的无参构造器，默认情况子类构造器的第一行即为super(); 当父类使用有参构造器，子类需要在构造器中用super(参数)访问父类的构造器。

this(...)能调用兄弟构造器，super(...)能调用父类构造器。两者都需要出现在构造器第一行，但两者不能同时出现在一个构造器中。

this和super的总结：

访问本类成员：

```
this.成员变量 //访问本类成员变量  
this.成员方法 //调用本类成员方法  
this() //调用本类空参数构造器  
this(参数) //调用本类有参数构造器
```

访问父类成员：

```
super.成员变量 //访问父类成员变量  
super.成员方法 //调用父类成员方法  
super() //调用父类空参数构造器  
super(参数) //调用父类有参数构造器
```

注意：this和super访问构造方法，只能用到构造方法的第一句，否则会报错。

## 多态

1. 多态：在继承/实现情况下的一种现象，表现为对象多态和行为多态。

多态的前提：有继承/实现关系；存在父类引用子类对象；存在方法重写。

左边是父类，右边是子类。

```
People p1 = new Teacher();  
p1.run(); //调用Teacher类中的run方法  
//编译看左边：编译时看左边People类，发现有run方法，不会报错  
//运行看右边：运行时看右边Teacher对象，执行Teacher的run方法  
//对于变量，编译和运行都是看左边  
  
People p2 = new Student();  
p2.run(); //调用Student类中的run方法  
  
//要求People中有run方法，Teacher和Student类重写run方法
```

2. 多态的好处

- (1) 可以实现解耦合，右边对象可以随时切换。
- (2) 可以使用父类类型的变量作为形参放入方法中，则一切子类对象都能调用该方法。

### 3. 类型转换

多态存在的问题：不能使用子类的独有功能。解决方法即类型转换。

自动类型转换：父类 变量名 = new 子类();

强制类型转换：子类 变量名 = (子类)父类变量

转换后就可以使用子类功能。

```
People p = new Teacher(); //自动  
Teacher t = (Teacher)p; //强制  
  
if (p1 instanceof Student) { //强转前先用instanceof关键字判断  
    Student s1 = (Student) p1;  
    s1.test();  
}
```

## final关键字

### 1. final关键字可以修饰类、方法、变量。

- (1) 修饰类：该类称为最终类，特点是不能被继承。
- (2) 修饰方法：该方法称之为最终方法，特点是不能被重写。
- (3) 修饰变量：该变量只能被赋值一次。

### 2. 常量：用 public static final 修饰。

好处：编译后会被宏替换，出现常量的地方全部会被替换成字面量。

### 3. final修饰基本类型变量，变量存储的数据不能修改；修饰引用类型的变量，变量存储的地址不能修改，但内容可以。

## 抽象类

### 1. 用abstract关键字修饰类，就称为抽象类；修饰方法就称为抽象方法。

```
//abstract修饰类，这个类就是抽象类  
public abstract class A{  
    //abstract修饰方法，这个方法就是抽象方法  
    //抽象方法不能有方法体  
    public abstract void test();  
}
```

### 2. 抽象类的特点

- (1) 抽象类中不一定有抽象方法，有抽象方法的类一定是抽象类。
- (2) 类该有的成员（成员变量、方法、构造器）抽象类也可以有。
- (3) 抽象类最主要的特点：抽象类不能创建对象，仅作为一种特殊的父类，让子类继承并实现。
- (4) 一个类继承抽象类，必须重写完抽象类的全部抽象方法，否则这个类也必须定义为抽象类。

### 3. 抽象类的好处和应用

用抽象类可以把父类中相同的代码，包括方法声明都抽取到父类，这样能更好的支持多态，以提高代码的灵活性。父类定义出抽象方法，子类去重写实现，调时可以使用不同子类的方法。

抽象类不是必须的，只是看起来更加专业，同时支持多态更好。

应用：模板方法设计模式

## 接口

### 1. 接口是一种特殊的结构

```
//接口中只能有两种成员，不能有构造器、代码块等
public interface 接口名{
    //成员变量（常量）
    //成员方法（抽象方法）
}
```

### 2. 接口不能创建对象。接口是用来被类实现的，实现接口的类被称为实现类

```
修饰符 class 实现类 implements 接口1, 接口2, 接口3.....{  
}
```

一个类可以实现多个接口。实现类实现多个接口，必须重写全部接口的全部抽象方法，否则实现类需要定义成抽象类。

### 3. 接口的好处

- (1) 弥补了类单继承的不足，一个类同时可以实现多个接口。
- (2) 让程序可以面向接口编程，这样程序员可以灵活方便的切换各种业务实现。

```
class Student{  
}
```

```

interface Driver{
    void drive();
}

interface Singer{
    void sing();
}

//A类是Student的子类，同时也实现了Dirver接口和Singer接口
class A extends Student implements Driver, Singer{
    @Override
    public void drive() {

    }

    @Override
    public void sing() {

    }
}

public class Test {
    public static void main(String[] args) {
        //想唱歌的时候，A类对象就表现为Singer类型
        Singer s = new A();
        s.sing();

        //想开车的时候，A类对象就表现为Driver类型
        Driver d = new A();
        d.drive();
    }
}

```

#### 4. 接口新特性

```

public interface A {
    /**
     * 1、默认方法：必须使用default修饰，默认会被public修饰
     * 实例方法：对象的方法，必须使用实现类的对象来访问。
     */
    default void test1(){
        System.out.println("==默认方法==");
        test2();
    }

    /**

```

```
* 2、私有方法：必须使用private修饰。(JDK 9开始才支持的)
*   实例方法：对象的方法。
*/
private void test2(){
    System.out.println("==私有方法==");
}

/**
* 3、静态方法：必须使用static修饰，默认会被public修饰
*/
static void test3(){
    System.out.println("==静态方法==");
}

void test4();
void test5();
default void test6(){

}

}

public class B implements A{
    @Override
    public void test4() {
    }

    @Override
    public void test5() {

    }
}

public class Test {
    public static void main(String[] args) {
        // 目标：掌握接口新增的三种方法形式
        B b = new B();
        b.test1(); //默认方法使用对象调用
        // b.test2(); //A接口中的私有方法，B类调用不了
        A.test3(); //静态方法，使用接口名调用
    }
}
```

好处：增强了接口的能力，因为接口中可以写有方法体的方法，便于项目扩展和维护。

## 5. 接口多继承

一个接口可以继承多个接口，这样相当于一个接口结合了多个接口，方便实现类去实现

```
public class Test {  
    public static void main(String[] args) {  
        // 目标：理解接口的多继承。  
    }  
}  
  
interface A{  
    void test1();  
}  
interface B{  
    void test2();  
}  
interface C{  
}  
  
//比如：D接口继承C、B、A  
interface D extends C, B, A{  
}  
  
//E类在实现D接口时，必须重写D接口、以及其父类中的所有抽象方法。  
class E implements D{  
    @Override  
    public void test1() {  
  
    }  
  
    @Override  
    public void test2() {  
  
    }  
}
```

接口多继承的注意事项：

- (1) 一个接口继承多个接口，如果多个接口中存在相同的方法声明（方法名相同），则此时不支持多继承。
- (2) 一个类实现多个接口，如果多个接口中存在相同的方法声明，则此时不支持多实现。
- (3) 一个类继承了父类，又同时实现了接口，父类中和接口中有同名的默认方法，实现类会优先使用父类的方法。

- (4) 一个类实现类多个接口，多个接口中有同名的默认方法，则这个类必须重写该方法。

## 内部类

### 1. 内部类

内部类是类中的五大成分之一（成员变量、方法、构造器、内部类、代码块），如果一个类定义在另一个类的内部，这个类就是内部类。

当一个类的内部，包含一个完整的事物，且这个事物没有必要单独设计时，就可以把这个事物设计成内部类。

比如：汽车、的内部有发动机，发动机是包含在汽车内部的一个完整事物，可以把发动机设计成内部类。

### 2. 内部类的四种形式

#### (1) 成员内部类：就是类中的一个普通成员

```
public class Outer {  
    private int age = 99;  
    public static String a="黑马";  
  
    // 成员内部类  
    public class Inner{  
        private String name;  
        private int age = 88;  
  
        //在内部类中既可以访问自己类的成员，也可以访问外部类的成员  
        public void test(){  
            System.out.println(age); //88  
            System.out.println(a); //黑马  
  
            int age = 77;  
            System.out.println(age); //77  
            System.out.println(this.age); //88  
            System.out.println(outer.this.age); //99 外部类  
名.this.成员  
        }  
  
        public String getName() {  
            return name;  
        }  
  
        public void setName(String name) {  
            this.name = name;  
        }  
}
```

```

    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

创建对象方法：

```

//外部类.内部类 变量名 = new 外部类().new 内部类();
Outer.Inner in = new Outer().new Inner();
//调用内部类的方法
in.test();

```

(2) 静态内部类：有static修饰的内部类，属于外部类自己持有的

```

//外部类.内部类 变量名 = new 外部类.内部类();
Outer.Inner in = new Outer.Inner();

public class Outer {
    private int age = 99;
    public static String schoolName="黑马";

    // 静态内部类
    public static class Inner{
        //静态内部类访问外部类的静态变量，是可以的;
        //静态内部类访问外部类的实例变量，是不行的
        public void test(){
            System.out.println(schoolName); //99
            //System.out.println(age); //报错
        }
    }
}

```

(3) 局部内部类：定义在方法中的类，和局部变量一样，只能在方法中有效。

(4) 匿名内部类

格式：

```

new 父类/接口(参数值){

```

```

@Override
重写父类/接口的方法;
}

//遇到匿名内部类，计算机首先会把这个匿名内部类编译成一个子类，然后会立即创建
一个子类对象

public class Test{
    public static void main(String[] args){
        //这里后面new 的部分，其实就是一个Animal的子类对象
        //这里隐含的有多态的特性： Animal a = Animal子类对象；
        Animal a = new Animal(){
            @Override
            public void cry(){
                System.out.println("猫喵喵喵的叫~~~");
            }
        }
        a.cry(); //直线上面重写的cry()方法
    }
}

```

特点：匿名内部类本质上是一个没有名字的子类对象、或者接口的实现类对象。

好处：更方便的创建一个子类对象。有时候需要使用这个对象，不需要专门为这个对象去设计一个类。

应用场景：在调用方法时，当方法的形参是一个接口或者抽象类，为了简化代码书写，而直接传递匿名内部类对象给方法。

```

public class Test{
    public static void main(String[] args){
        Swimming s1 = new Swimming(){ //创建对象
            public void swim(){
                System.out.println("狗刨飞快");
            }
        };
        go(s1);

        //形参是Swimming接口，实参可以接收任意Swimming接口的实现类对象
        public static void go(Swimming s){
            System.out.println("开始~~~~~");
            s.swim();
            System.out.println("结束~~~~~");
        }
    }
}

```

```
public interface Swimming{  
    public void swim();  
}
```

## 枚举

1. 枚举是一种特殊的类，格式为

```
public enum 枚举类名{  
    枚举项1, 枚举项2, 枚举项3; //第一行必须是枚举项  
    .....  
}
```

### 获取枚举项

```
public class Test{  
    public static void main(String[] args){  
        //获取枚举A类的，枚举项  
        A a1 = A.X;  
        A a2 = A.Y;  
        A a3 = A.Z;  
    }  
}
```

枚举项都是常量，且每个常量记住的都是枚举类的一个对象。

枚举类的构造器都是私有的，因此不能创建对象。

枚举类都是最终类，不能被继承。

2. 枚举类也能加构造器和方法

```
public class SeasonTest1 {  
    public static void main(String[] args) {  
        Season1 summer = Season1.SUMMER;  
  
        System.out.println(summer);  
        System.out.println(Season1.class.getSuperclass());  
        //class java.lang.Enum  
    }  
}  
  
//使用enum关键字定义枚举类  
enum Season1 {  
    //1. 先显示列出枚举类的对象，以","分隔， ";"结尾  
    SPRING("春天", "春暖花开"),
```

```

SUMMER("夏天", "烈日炎炎"),
AUTUMN("秋天", "秋高气爽"),
WINTER("冬天", "冰天雪地");
//2. 声明Season对象的属性: private final修饰
private final String seasonName;
private final String seasonDesc;
//3. 私有化类的构造器, 并给对象赋值
Season1(String seasonName, String seasonDesc) {
    this.seasonName = seasonName;
    this.seasonDesc = seasonDesc;
}
//4. 其他诉求: 获取枚举类的属性
public String getSeasonName() {
    return seasonName;
}

public String getSeasonDesc() {
    return seasonDesc;
}
//5. 其他诉求: 提供toString()
@Override
public String toString() {
    return "Season1{" +
        "seasonName='" + seasonName + '\'' +
        ", seasonDesc='" + seasonDesc + '\'' +
        '}';
}

```

## 泛型

### 1. 认识泛型

泛型指的是, 在定义类、接口、方法时, 同时声明了一个或者多个类型变量(如: ) , 称为泛型类、泛型接口、泛型方法、它们统称为泛型。

### 2. 自定义泛型类

```

//这里的<T,W>其实指的就是类型变量, 可以是一个, 也可以是多个。
public class 类名<T,W>{
}
```

```

//定义一个泛型类, 用来表示一个容器
//容器中存储的数据, 它的类型用<E>先代替用着, 等调用者来确认<E>的具体类型。
public class MyArrayList<E>{
```

```

private Object[] array = new Object[10];
//定一个索引，方便对数组进行操作
private int index;

//添加元素
public void add(E e){
    array[index]=e;
    index++;
}

//获取元素
public E get(int index){
    return (E)array[index];
}

}

//定义一个泛型类，用来表示一个容器
//容器中存储的数据，它的类型用<E>先代替用着，等调用者来确认<E>的具体类型。
public class MyArrayList<E>{
    private Object[] array = new Object[10];
    //定一个索引，方便对数组进行操作
    private int index;

    //添加元素
    public void add(E e){
        array[index]=e;
        index++;
    }

    //获取元素
    public E get(int index){
        return (E)array[index];
    }
}

```

### 3. 自定义泛型接口

```

//这里的类型变量，一般是一个字母，比如<E>
public interface 接口名<类型变量>{

}

```

```

public interface Data<T>{
    public void add(T t);
}

```

```
    public ArrayList<T> getByName(String name);  
}  
  
//此时确定Data<E>中的E为Teacher类型,  
//接口中add和getByName方法上的T也都会变成Teacher类型  
public class TeacherData implements Data<Teacher>{  
    public void add(Teacher t){  
  
    }  
  
    public ArrayList<Teacher> getByName(String name){  
  
    }  
}
```

#### 4. 自定义泛型方法

```
public <泛型变量,泛型变量> 返回值类型 方法名(形参列表){  
  
}
```

#### 5. 泛型限定

泛型限定的意思是对泛型的数据类型进行范围的限制。有如下的三种格式

- (1) <?> 表示任意类型，? 是通配符，在使用泛型（如ArrayList）时表示一切类型。
- (2) <? extends 数据类型> 表示指定数据类型或者指定类型的子类。
- (3) <? super 数据类型> 表示指定类型或者指定类型的父类。

#### 6. 泛型擦除

- (1) 泛型只能编译阶段有效，一旦编译成字节码，字节码中是不包含泛型的。
- (2) 泛型只支持引用数据类型，不支持基本数据类型。

```
ArrayList<int> //error  
ArrayList<double> //error  
ArrayList<Integer> //yes  
ArrayList<Double> //yes
```

# API

## 包

### 1. 在程序中调包的规则

- (1) 如果当前程序中，要调用自己所在包下的其他程序，可以直接调用。（同一个包下的类，互相可以直接调用）
- (2) 如果当前程序中，要调用其他包下的程序，则必须在当前程序中导包，才可以访问！

导包格式：`import 包名.类名`

(3) 如果当前程序中，要调用Java.lang包下的程序，不需要我们导包的，可以直接使用。调用其他Java程序需要导包。

(4) 如果当前程序中，要调用多个不同包下的程序，而这些程序名正好一样，此时默认只能导入一个程序，另一个程序必须带包名访问。

## String

### 1. 创建对象

```
//1. 直接双引号得到字符串对象
String name = "xzs";

//2. new String() 使用构造器进行初始化
String n1 = new String(); //无参数构造器

String n2 = new String("xzs"); //构造器public String(String
original)

char[] chars = {'a', 'b', 'c'};
String n3 = new String(chars); //构造器public String(char[]
chars), 把字符连接形成字符串
System.out.println(n3); //输出abc

byte[] bytes = {97, 98, 99};
String n4 = new String(bytes); //构造器public String(byte[]
bytes), 把整数转成ASCII的字符，再连接形成字符串
System.out.println(n4); //输出abc
```

### 2. 常用方法

方法名	说明
public int length()	获取字符串的长度返回（就是字符个数）
public char charAt(int index)	获取某个索引位置处的字符返回
public char[] toCharArray():	将当前字符串转换成字符数组返回
public boolean equals(Object anObject)	判断当前字符串与另一个字符串的内容一样，一样返回true
public boolean equalsIgnoreCase(String anotherString)	判断当前字符串与另一个字符串的内容是否一样(忽略大小写)
public String substring(int beginIndex, int endIndex)	根据开始和结束索引进行截取，得到新的字符串（包前不包后）
public String substring(int beginIndex)	从传入的索引处截取，截取到末尾，得到新的字符串返回
public String replace(CharSequence target, CharSequence replacement)	使用新值，将字符串中的旧值替换，得到新的字符串
public boolean contains(CharSequence s)	判断字符串中是否包含了某个字符串
public boolean startsWith(String prefix)	判断字符串是否以某个字符串内容开头，开头返回true，反之
public String[] split(String regex)	把字符串按照某个字符串内容分割，并返回字符串数组回来

```

public class StringDemo2 {
    public static void main(String[] args) {
        //目标：快速熟悉String提供的处理字符串的常用方法。
        String s = "黑马Java";
        // 1、获取字符串的长度
        System.out.println(s.length());

        // 2、提取字符串中某个索引位置处的字符
        char c = s.charAt(1);
        System.out.println(c); //马

        // 字符串的遍历
        for (int i = 0; i < s.length(); i++) {
            // i = 0 1 2 3 4 5
            char ch = s.charAt(i);
            System.out.println(ch);
        }

        System.out.println("-----");

        // 3、把字符串转换成字符数组，再进行遍历
        char[] chars = s.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            System.out.println(chars[i]);
        }

        // 4、判断字符串内容，内容一样就返回true
        String s1 = new String("黑马");
        String s2 = new String("黑马");
        System.out.println(s1 == s2); // false, 比较地址
        System.out.println(s1.equals(s2)); // true

        // 5、忽略大小写比较字符串内容
        String c1 = "34AeFG";
    }
}

```

```

String c2 = "34aEfg";
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equalsIgnoreCase(c2)); // true

// 6、截取字符串内容（包前不包后的）
String s3 = "Java是最好的编程语言之一";
String rs = s3.substring(0, 8);
System.out.println(rs);

// 7、从当前索引位置一直截取到字符串的末尾
String rs2 = s3.substring(5);
System.out.println(rs2);

// 8、把字符串中的某个内容替换成新内容，并返回新的字符串对象给我们
String info = "这个电影简直是个垃圾，垃圾电影！！";
String rs3 = info.replace("垃圾", "***");
System.out.println(rs3);

// 9、判断字符串中是否包含某个关键字
String info2 = "Java是最好的编程语言之一，我爱Java，Java不爱
我！";
System.out.println(info2.contains("Java"));
System.out.println(info2.contains("java"));
System.out.println(info2.contains("Java2"));

// 10、判断字符串是否以某个字符串开头。
String rs4 = "张三丰";
System.out.println(rs4.startsWith("张"));
System.out.println(rs4.startsWith("张三"));
System.out.println(rs4.startsWith("张三2"));

// 11、把字符串按照某个指定内容分割成多个字符串，放到一个字符串数
组中返回给我们
String rs5 = "张无忌,周芷若,殷素素,赵敏";
String[] names = rs5.split(",");
for (int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}
}

```

### 3. String注意事项

- (1) String对象是不可变字符串对象。每次试图改变字符串对象实际上是新产生新的字符串对象，变量每次都是指向了新的字符串对象，之前字符串对象内容不变。

(2) 用" "写出的字符串对象，会存储到字符串常量池，且相同内容的字符串只存储一份；但通过new方式创建字符串对象，每new一次都会产生一个新对象放在堆内存中。

```
String s1 = "abc";
String s2 = "abc";
System.out.println(s1 == s2); //true

char[] chars = {'a','b','c'};
String s3 = new String(chars);
String s4 = new String(chars);
System.out.println(s3 == s4); //false
```

## ArrayList

ArrayList表示一种集合，它是一个容器，用来装数据的，类似于数组。集合是大小可变的。

### 1. 常用方法

集合最常用的方法就是增删改查。

#### ArrayList<E>

- 是用的最多、最常见的一种集合。

构造器	说明
public ArrayList()	创建一个空的集合对象
常用方法名	说明
public boolean add(E e)	将指定的元素添加到此集合的末尾
public void add(int index,E element)	在此集合中的指定位置插入指定的元素
public E get(int index)	返回指定索引处的元素
public int size()	返回集合中的元素的个数
public E remove(int index)	删除指定索引处的元素，返回被删除的元素
public boolean remove(Object o)	删除指定的元素，返回删除是否成功
public E set(int index,E element)	修改指定索引处的元素，返回被修改的元素

```
ArrayList list = new ArrayList(); //未使用泛型
ArrayList<int> list = new ArrayList<>(); //使用泛型

public class ArrayListDemo1 {
    public static void main(String[] args) {
        // 1、创建一个ArrayList的集合对象
        // ArrayList<String> list = new ArrayList<String>();
```

```

// 从jdk 1.7开始才支持的
ArrayList<String> list = new ArrayList<>();

list.add("黑马");
list.add("黑马");
list.add("Java");
System.out.println(list);

// 2、往集合中的某个索引位置处添加一个数据
list.add(1, "MySQL");
System.out.println(list);

// 3、根据索引获取集合中某个索引位置处的值
String rs = list.get(1);
System.out.println(rs);

// 4、获取集合的大小（返回集合中存储的元素个数）
System.out.println(list.size());

// 5、根据索引删除集合中的某个元素值，会返回被删除的元素值给我们
System.out.println(list.remove(1)); //删除索引1的元素
System.out.println(list);

// 6、直接删除某个元素值，删除成功会返回true，反之
System.out.println(list.remove("Java"));
System.out.println(list);

list.add(1, "html");
System.out.println(list);

// 默认删除的是第一次出现的这个黑马的数据的
System.out.println(list.remove("黑马"));
System.out.println(list);

// 7、修改某个索引位置处的数据，修改后会返回原来的值给我们
System.out.println(list.set(1, "黑马程序员"));
System.out.println(list);
}
}

```

## 2. 从集合筛选元素并删除

```

public class ArrayListTest2 {
    public static void main(String[] args) {
        // 1、创建一个ArrayList集合对象
        ArrayList<String> list = new ArrayList<>();

```

```
list.add("枸杞");
list.add("Java入门");
list.add("宁夏枸杞");
list.add("黑枸杞");
list.add("人字拖");
list.add("特级枸杞");
list.add("枸杞子");
System.out.println(list);
//运行结果如下: [Java入门, 宁夏枸杞, 黑枸杞, 人字拖, 特级枸杞,
枸杞子]

// 2、开始完成需求: 从集合中找出包含枸杞的数据并删除它
for (int i = 0; i < list.size(); i++) {
    // i = 0 1 2 3 4 5
    // 取出当前遍历到的数据
    String ele = list.get(i);
    // 判断这个数据中包含枸杞
    if(ele.contains("枸杞")){
        // 直接从集合中删除该数据
        list.remove(ele);
    }
}
System.out.println(list);
//删除后结果如下: [Java入门, 黑枸杞, 人字拖, 枸杞子]
//原因: 删除后i会变化
}

//正确做法
// 方式一: 每次删除一个数据后, 就让i往左边退一步
for (int i = 0; i < list.size(); i++) {
    // i = 0 1 2 3 4 5
    // 取出当前遍历到的数据
    String ele = list.get(i);
    // 判断这个数据中包含枸杞
    if(ele.contains("枸杞")){
        // 直接从集合中删除该数据
        list.remove(ele);
        i--;
    }
}

// 方式二: 从集合的后面倒着遍历并删除
for (int i = list.size() - 1; i >= 0; i--) {
    // 取出当前遍历到的数据
```

```

String ele = list.get(i);
// 判断这个数据中包含枸杞
if(ele.contains("枸杞")){
    // 直接从集合中删除该数据
    list.remove(ele);
}

```

## Object

### 1. 常用方法

#### Object类的常见方法

方法名	说明
public String <b>toString()</b>	返回对象的字符串表示形式。
public boolean <b>equals(Object o)</b>	判断两个对象是否相等。
protected Object <b>clone()</b>	对象克隆

- (1) IDEA输入toS，回车后会自动重写我们想要的toString内容。 (好用！！！ )
- (2) equals方法比较的是地址。 输入eq一直回车会重写该方法。

#### public **String toString()**

返回对象的字符串表示形式。默认的格式是：“包名.类名@哈希值16进制”

【子类重写后，返回对象的属性值】

#### public **boolean equals(Object o)**

判断此对象与参数对象是否“相等”。默认比较对象的地址值，和“==”没有区别

【子类重写后，比较对象的属性值】

#### public class Student{

    private String name;

    private int age;

        public Student(String name, int age){

            this.name=name;

            this.age=age;

        }

    @Override

        public String **toString()**{

            return "Student{name='"+name+"', age="+age+"'}";

```

    }

    //重写equals方法，按照对象的属性值进行比较
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Student student = (Student) o;

        if (age != student.age) return false;
        return name != null ? name.equals(student.name) :
student.name == null;
    }
}

```

(3) clone方法会复制一个一模一样的新对象。

```

//1.必须重写
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}

//2.要实现接口
public class A implements Cloneable{
}

```

## Objects

### 1. 常用方法

Objects类的常见方法

方法名	说明
public static boolean equals(Object a, Object b)	先做非空判断，再比较两个对象
public static boolean isNull(Object obj)	判断对象是否为null，为null返回true，反之
public static boolean nonNull(Object obj)	判断对象是否不为null，不为null则返回true，反之

```

public class Test{
    public static void main(String[] args){
        String s1 = null;
    }
}

```

```

String s2 = "itheima";

//这里会出现NullPointerException异常，调用者不能为null
System.out.println(s1.equals(s2));
//此时不会有NullPointerException异常，底层会自动先判断空
System.out.println(Objects.equals(s1,s2));

//判断对象是否为null，等价于 ==
System.out.println(Objects.isNull(s1)); //true
System.out.println(s1==null); //true

//判断对象是否不为null，等价于 !=
System.out.println(Objects.nonNull(s2)); //true
System.out.println(s2!=null); //true
}
}

```

## 基本类型包装类

- 包装类就是把基本类型的数据包装成对象。

基本数据类型	对应的包装类（引用数据类型）
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

### 1. 装箱和拆箱

```

//1. 创建Integer对象，封装基本类型数据10
Integer a = new Integer(10);

//2. 使用Integer类的静态方法valueOf(数据)
Integer b = Integer.valueOf(10);

//3. 还有一种自动装箱的写法（意思就是自动将基本类型转换为引用类型）
Integer c = 10;

//4. 有装箱肯定还有拆箱（意思就是自动将引用类型转换为基本类型）
int d = c;

```

```
//5.装箱和拆箱在使用集合时就有体现
ArrayList<Integer> list = new ArrayList<>();
//添加的元素是基本类型，实际上会自动装箱为Integer类型
list.add(100);
//获取元素时，会将Integer类型自动拆箱为int类型
int e = list.get(0);
```

## 2. 数据类型转换

```
//1.字符串转换为数值型数据
String ageStr = "29";
int age1 = Integer.parseInt(ageStr);

Integer age2 = Integer.valueOf(age1);

String scoreStr = 3.14;
double score = Double.parseDouble(scoreStr);

//2.整数转换为字符串，以下几种方式都可以（挑中你喜欢的记一下）
Integer a = 23;
String s1 = Integer.toString(a);
String s2 = a.toString();
String s3 = a+"";
String s4 = String.valueOf(a);
```

# StringXXX

## 1. StringBuilder

StringBuilder代表可变字符串对象，相当于是一个容器，它里面的字符串是可以改变的，就是用来操作字符串的。

### 常用方法

构造器	说明
public StringBuilder()	创建一个空白的可变的字符串对象，不包含任何内容
public StringBuilder(String str)	创建一个指定字符串内容的可变字符串对象

方法名称	说明
public StringBuilder append(任意类型)	添加数据并返回StringBuilder对象本身
public StringBuilder reverse()	将对象的内容反转
public int length()	返回对象内容长度
public String toString()	通过toString()就可以实现把StringBuilder转换为String

```
public class Test{
```

```

public static void main(String[] args){
    StringBuilder sb = new StringBuilder("itheima");

    //1.拼接内容
    sb.append(12);
    sb.append("黑马");
    sb.append(true);

    //2.append方法，支持临时编程
    sb.append(666).append("黑马2").append(666);
    System.out.println(sb); //打印: itheima12黑马true666黑马
    2666

    //3.反转操作
    sb.reverse();
    System.out.println(sb); //打印: 6662马黑666马黑21

    //4.返回字符串的长度
    System.out.println(sb.length());

    //5.StringBuilder还可以转换为字符串
    String s = sb.toString();
    System.out.println(s); //打印: 6662马黑666马黑21
}
}

```

## 2. StringBuffer

与StringBuilder用法相同，但StringBuilder不是线程安全的，而StringBuffer是线程安全的。

## 3. StringJoiner

构造器	说明
public StringJoiner (间隔符号)	创建一个StringJoiner对象，指定拼接时的间隔符号
public StringJoiner (间隔符号, 开始符号, 结束符号)	创建一个StringJoiner对象，指定拼接时的间隔符号、开始符号、结束符号

方法名称	说明
public StringJoiner add (添加的内容)	添加数据，并返回对象本身
public int length()	返回长度（字符出现的个数）
public String toString()	返回一个字符串（该字符串就是拼接之后的结果）

```

public class Test{
    public static void main(String[] args){
        StringJoiner s = new StringJoiner(",");
        s.add("java1");
        s.add("java2");
    }
}

```

```

        s.add("java3");
        System.out.println(s); //结果为: java1,java2,java3

        //参数1: 间隔符
        //参数2: 开头
        //参数3: 结尾
        StringJoiner s1 = new StringJoiner(",","[","]");
        s1.add("java1");
        s1.add("java2");
        s1.add("java3");
        System.out.println(s1); //结果为: [java1,java2,java3]
    }
}

```

## Math

```

public class MathTest {
    public static void main(String[] args) {
        // 目标: 了解下Math类提供的常见方法。
        // 1、public static int abs(int a): 取绝对值（拿到的结果一定是正数）
        //     public static double abs(double a)
        System.out.println(Math.abs(-12)); // 12
        System.out.println(Math.abs(123)); // 123
        System.out.println(Math.abs(-3.14)); // 3.14

        // 2、public static double ceil(double a): 向上取整
        System.out.println(Math.ceil(4.0000001)); // 5.0
        System.out.println(Math.ceil(4.0)); // 4.0

        // 3、public static double floor(double a): 向下取整
        System.out.println(Math.floor(4.999999)); // 4.0
        System.out.println(Math.floor(4.0)); // 4.0

        // 4、public static long round(double a): 四舍五入
        System.out.println(Math.round(3.4999)); // 3
        System.out.println(Math.round(3.50001)); // 4

        // 5、public static int max(int a, int b): 取较大值
        //     public static int min(int a, int b): 取较小值
        System.out.println(Math.max(10, 20)); // 20
        System.out.println(Math.min(10, 20)); // 10

        // 6、public static double pow(double a, double b): 取次方
        System.out.println(Math.pow(2, 3)); // 2的3次方 8.0
    }
}

```

```
System.out.println(Math.pow(3, 2)); // 3的2次方 9.0

// 7、public static double random(): 取随机数 [0.0 , 1.0)
// (包前不包后)
System.out.println(Math.random());
}
```

## System

### 1. 常用：获取系统时间

```
/***
 * 目标：了解下System类的常见方法。
 */
public class SystemTest {
    public static void main(String[] args) {

        // 1、public static void exit(int status):
        // 终止当前运行的Java虚拟机。
        // 该参数用作状态代码；按照惯例，非零状态代码表示异常终止。
        System.exit(0); // 人为的终止虚拟机。（不要使用）

        // 2、public static long currentTimeMillis():
        // 获取当前系统的时间
        // 返回的是long类型的时间毫秒值：指的是从1970-1-1 0:0:0开始
        // 走到此刻的总的毫秒值，1s = 1000ms
        long time = System.currentTimeMillis();
        System.out.println(time);

        for (int i = 0; i < 1000000; i++) {
            System.out.println("输出了：" + i);
        }

        long time2 = System.currentTimeMillis();
        System.out.println((time2 - time) / 1000.0 + "s");
    }
}
```

## Runtime

### 1. 作用：获取JVM的一些信息，也可以用这个类去执行其他的程序。

```
/***
 * 目标：了解下Runtime的几个常见方法。
 */
```

```

/*
public class RuntimeTest {
    public static void main(String[] args) throws IOException,
InterruptedException {

        // 1、public static Runtime getRuntime() 返回与当前Java应用程序关联的运行时对象。
        Runtime r = Runtime.getRuntime();

        // 2、public void exit(int status) 终止当前运行的虚拟机，该参数用作状态代码；按照惯例，非零状态代码表示异常终止。
        // r.exit(0);

        // 3、public int availableProcessors()：获取虚拟机能够使用的处理器数。
        System.out.println(r.availableProcessors());

        // 4、public long totalMemory() 返回Java虚拟机中的内存总量。
        System.out.println(r.totalMemory()/1024.0/1024.0 +
"MB"); // 1024 = 1K      1024 * 1024 = 1M

        // 5、public long freeMemory() 返回Java虚拟机中的可用内存量
        System.out.println(r.freeMemory()/1024.0/1024.0 +
"MB");

        // 6、public Process exec(String command) 启动某个程序，并返回代表该程序的对象。
        // r.exec("D:\\soft\\XMind\\XMind.exe");
        Process p = r.exec("QQ");
        Thread.sleep(5000); // 让程序在这里暂停5s后继续往下走！！
        p.destroy(); // 销毁！关闭程序！

    }
}

```

## BigDecimal

1. 作用：解决计算精度损失问题

## BigDecimal的常见构造器、常用方法

构造器	说明
public BigDecimal(double val) 注意：不推荐使用这个	将 double转换为 BigDecimal
public BigDecimal(String val)	把String转成BigDecimal
方法名	说明
public static BigDecimal valueOf(double val)	转换一个 double成 BigDecimal
public BigDecimal add(BigDecimal b)	加法
public BigDecimal subtract(BigDecimal b)	减法
public BigDecimal multiply(BigDecimal b)	乘法
public BigDecimal divide(BigDecimal b)	除法
public BigDecimal divide (另一个BigDecimal对象, 精确几位, 舍入模式)	除法、可以控制精确到小数几位
public double doubleValue()	将BigDecimal转换为double

```
public class Test2 {  
    public static void main(String[] args) {  
        // 目标：掌握BigDecimal进行精确运算的方案。  
        double a = 0.1;  
        double b = 0.2;  
  
        // 1、把浮点型数据封装成BigDecimal对象，再来参与运算。  
        // a、public BigDecimal(double val) 得到的BigDecimal对象  
        // 是无法精确计算浮点型数据的。 注意：不推荐使用这个，  
        // b、public BigDecimal(string val) 得到的BigDecimal对象  
        // 是可以精确计算浮点型数据的。 可以使用。  
        // c、public static BigDecimal valueOf(double val)：通过  
        // 这个静态方法得到的BigDecimal对象是可以精确运算的。是最好的方案。原理是先  
        // 把double转成string，再用b方法  
        BigDecimal a1 = BigDecimal.valueOf(a);  
        BigDecimal b1 = BigDecimal.valueOf(b);  
  
        // 2、public BigDecimal add(BigDecimal augend)：加法  
        BigDecimal c1 = a1.add(b1);  
        System.out.println(c1);  
  
        // 3、public BigDecimal subtract(BigDecimal augend)：减  
        // 法  
        BigDecimal c2 = a1.subtract(b1);  
        System.out.println(c2);  
  
        // 4、public BigDecimal multiply(BigDecimal augend)：乘  
        // 法  
        BigDecimal c3 = a1.multiply(b1);  
    }  
}
```

```
System.out.println(c3);

// 5、public BigDecimal divide(BigDecimal b): 除法
BigDecimal c4 = a1.divide(b1);
System.out.println(c4);

//      BigDecimal d1 = BigDecimal.valueOf(0.1);
//      BigDecimal d2 = BigDecimal.valueOf(0.3);
//      BigDecimal d3 = d1.divide(d2);
//      System.out.println(d3);

// 6、public BigDecimal divide(另一个BigDecimal对象, 精确
// 几位, 舍入模式) : 除法, 可以设置精确几位。
BigDecimal d1 = BigDecimal.valueOf(0.1);
BigDecimal d2 = BigDecimal.valueOf(0.3);
BigDecimal d3 = d1.divide(d2, 2,
RoundingMode.HALF_UP); // 0.33
System.out.println(d3);

// 7、public double doubleValue() : 把BigDecimal对象又转
// 换成double类型的数据。
//print(d3);
//print(c1);
double db1 = d3.doubleValue();
double db2 = c1.doubleValue();
print(db1);
print(db2);
}

public static void print(double a){
    System.out.println(a);
}
}
```

## Date&SimpleDateFormat

### 1. Date

## Date

- 代表的是日期和时间。

构造器	说明
public Date()	创建一个Date对象，代表的是系统当前此刻日期时间。
public Date(long time)	把时间毫秒值转换成Date日期对象。

常见方法	说明
public long getTime()	返回从1970年1月1日 00:00:00走到此刻的总的毫秒数
public void setTime(long time)	设置日期对象的时间为当前时间毫秒值对应的时间

## 2. SimpleDateFormat

可以把日期对象、时间毫秒形式格式化成我们想要的形式。



常见构造器	说明
public SimpleDateFormat(String pattern)	创建简单日期格式化对象，并封装时间的格式

格式化时间的方法	说明
public final String format(Date date)	将日期格式化成日期/时间字符串
public final String format(Object time)	将时间毫秒值式化成日期/时间字符串

```
public class Test2SimpleDateFormat {
    public static void main(String[] args) throws
ParseException {
        // 目标: 掌握SimpleDateFormat的使用。
        // 1、准备一些时间
        Date d = new Date();
        System.out.println(d);

        long time = d.getTime();
        System.out.println(time);
    }
}
```

```

// 2、格式化日期对象，和时间 毫秒值。
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月
dd日 HH:mm:ss EEE a");

String rs = sdf.format(d);
String rs2 = sdf.format(time);
System.out.println(rs);
System.out.println(rs2);
System.out.println("-----");
}

// 目标：掌握SimpleDateFormat解析字符串时间 成为日期对象。
String dateStr = "2022-12-12 12:12:11";
// 1、创建简单日期格式化对象，指定的时间格式必须与被解析的时间格
式一模一样，否则程序会出bug。
SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
Date d2 = sdf2.parse(dateStr);
System.out.println(d2);
}
}

```

## Calendar

- 作用：Calendar类提供了方法可以直接对日历中的年、月、日、时、分、秒等进
行运算。

### Calendar日历类的常见方法

方法名	说明
public static Calendar getInstance()	获取当前日历对象
public int get(int field)	获取日历中的某个信息。
public final Date getTime()	获取日期对象。
public long getTimeInMillis()	获取时间毫秒值
public void set(int field,int value)	修改日历的某个信息。
public void add(int field,int amount)	为某个信息增加/减少指定的值

```

public class Test4Calendar {
    public static void main(String[] args) {
        // 目标：掌握Calendar的使用和特点。
        // 1、得到系统此刻时间对应的日历对象。
        Calendar now = Calendar.getInstance();
        System.out.println(now);

        // 2、获取日历中的某个信息
        int year = now.get(Calendar.YEAR);
        System.out.println(year);

        int days = now.get(Calendar.DAY_OF_YEAR);
        System.out.println(days);
    }
}

```

## JDK8新时间（日期、时间）

### 1. 传统时间与新时间对比

JDK8之前 传统的时间API	JDK8开始之后 新增的时间API
1、设计不合理，使用不方便，很多都被淘汰了。 2、都是可变对象，修改后会丢失最开始的时间信息。 3、线程不安全。 4、只能精确到毫秒。	1、设计更合理，功能丰富，使用更方便。 2、都是不可变对象，修改后会返回新的时间对象，不会丢失最开始的时间。 3、线程安全。 4、能精确到毫秒、纳秒。

### 2. 日期时间类

LocalDate：代表本地时间（年、月、日、星期）

LocalTime：代表本地时间（时、分、秒、纳秒）

LocalDateTime（用最多）：代表本地日期、时间（年、月、日、星期、时、分、秒、纳秒）

#### 它们获取对象的方案

方法名	示例
public static Xxx now(): 获取系统当前时间对应的该对象	LocalDate ld = LocalDate.now(); LocalTime lt = LocalTime.now(); LocalDateTime ldt = LocalDateTime.now();
public static Xxx of(...): 获取指定时间的对象	LocalDate localDate1 = LocalDate.of(2099, 11, 11); LocalTime localTime1 = LocalTime.of(9, 8, 59); LocalDateTime localDateTime1 = LocalDateTime.of(2025, 11, 16, 14, 30, 01);

## LocalDateTime类的基本使用

LocalDateTime的常用API（可以处理年、月、日、星期、时、分、秒、纳秒等信息）

方法名	说明
getYear、getMonthValue、getDayOfMonth、getDayOfYear getDayOfWeek、getHour、getMinute、getSecond、getNano	获取年月日、时分秒、纳秒等
withYear、withMonth、withDayOfMonth、withDayOfYear withHour、withMinute、withSecond、withNano	修改某个信息，返回新日期时间对象
plusYears、plusMonths、plusDays、plusWeeks plusHours、plusMinutes、plusSeconds、plusNanos	把某个信息加多少，返回新日期时间对象
minusYears、minusMonths、minusDays、minusWeeks minusHours、minusMinutes、minusSeconds、minusNanos	把某个信息减多少，返回新日期时间对象
equals isBefore isAfter	判断2个时间对象，是否相等，在前还是在后

```
public class Test3_LocalDateTime {
    public static void main(String[] args) {
        // 0、获取本地日期和时间对象。
        LocalDateTime ldt = LocalDateTime.now(); // 年 月 日 时
        // 分 秒 纳秒
        System.out.println(ldt);

        // 1、可以获取日期和时间的全部信息
        int year = ldt.getYear(); // 年
        int month = ldt.getMonthValue(); // 月
        int day = ldt.getDayOfMonth(); // 日
        int dayOfYear = ldt.getDayOfYear(); // 一年中的第几天
        int dayOfWeek = ldt.getDayOfWeek().getValue(); // 获取
        // 是周几

        int hour = ldt.getHour(); // 时
        int minute = ldt.getMinute(); // 分
        int second = ldt.getSecond(); // 秒
        int nano = ldt.getNano(); // 纳秒

        // 2、修改时间信息:
        // withYear withMonth withDayOfMonth withDayOfYear
        withHour
        // withMinute withSecond withNano
        LocalDateTime ldt2 = ldt.withYear(2029);
        LocalDateTime ldt3 = ldt.withMinute(59);

        // 3、加多少:
        // plusYears plusMonths plusDays plusWeeks plusHours
        plusMinutes plusSeconds plusNanos
    }
}
```

```

    LocalDateTime ldt4 = ldt.plusYears(2);
    LocalDateTime ldt5 = ldt.plusMinutes(3);

    // 4、减多少:
    // minusDays minusYears minusMonths minusWeeks
    minusHours minusMinutes minusSeconds minusNanos
    LocalDateTime ldt6 = ldt.minusYears(2);
    LocalDateTime ldt7 = ldt.minusMinutes(3);

    // 5、获取指定日期和时间的LocalDateTime对象:
    // public static LocalDateTime of(int year, Month
    month, int dayOfMonth, int hour,
    //                                         int minute, int
    second, int nanoOfSecond)
    LocalDateTime ldt8 = LocalDateTime.of(2029, 12, 12,
    12, 12, 1222);
    LocalDateTime ldt9 = LocalDateTime.of(2029, 12, 12,
    12, 12, 1222);

    // 6、判断2个日期、时间对象，是否相等，在前还是在后: equals、
    isBefore、isAfter
    System.out.println(ldt9.equals(ldt8));
    System.out.println(ldt9.isAfter(ldt));
    System.out.println(ldt9.isBefore(ldt));

    // 7、可以把LocalDateTime转换成LocalDate和LocalTime
    // public LocalDate toLocalDate()
    // public LocalTime toLocalTime()
    // public static LocalDateTime of(LocalDate date,
    LocalTime time)
    LocalDate ld = ldt.toLocalDate();
    LocalTime lt = ldt.toLocalTime();
    LocalDateTime ldt10 = LocalDateTime.of(ld, lt);

}
}

```

## JDK8新时间（时区）

### 1. 方法（代替Calendar）

## ZonedDateTime 时区的常见方法

方法名	说明
public static Set<String> getAvailableZoneIds()	获取Java中支持的所有时区
public static ZoneId systemDefault()	获取系统默认时区
public static ZoneId of(String zoneId)	获取一个指定时区

## ZonedDateTime 带时区时间的常见方法

方法名	说明
public static ZonedDateTime now()	获取当前时区的ZonedDateTime对象
public static ZonedDateTime now(ZoneId zone)	获取指定时区的ZonedDateTime对象
getYear、getMonthValue、getDayOfMonth、getDayOfYear、getDayOfWeek、getHour、getMinute、getSecond、getNano	获取年月日、时分秒、纳秒等
public ZonedDateTime withXxx(时间)	修改时间系列的方法
public ZonedDateTime minusXxx(时间)	减少时间系列的方法
public ZonedDateTime plusXxx(时间)	增加时间系列的方法

```
public class Test4_ZoneId_ZonedDateTime {
    public static void main(String[] args) {
        // 目标：了解时区和带时区的时间。
        // 1、ZoneId的常见方法：
        // public static ZoneId systemDefault()：获取系统默认的时
        // 区
        ZoneId zoneId = ZoneId.systemDefault();
        System.out.println(zoneId.getId());
        System.out.println(zoneId);

        // public static Set<String> getAvailableZoneIds()：获
        取Java支持的全部时区ID
        System.out.println(zoneId.getAvailableZoneIds());

        // public static ZoneId of(String zoneId)：把某个时区id
        封装成ZoneId对象。
        ZoneId zoneId1 = ZoneId.of("America/New_York");

        // 2、ZonedDateTime：带时区的时间。
        // public static ZonedDateTime now(ZoneId zone)：获取某
        个时区的ZonedDateTime对象。
        ZonedDateTime now = ZonedDateTime.now(zoneId1);
        System.out.println(now);

        // 世界标准时间
    }
}
```

```

        zonedDateTime now1 =
zonedDateTime.now(clock.systemUTC());
System.out.println(now1);

        // public static ZonedDateTime now(): 获取系统默认时区的
ZonedDateTime对象
        ZonedDateTime now2 = ZonedDateTime.now();
System.out.println(now2);

        // Calendar instance =
Calendar.getInstance(TimeZone.getTimeZone(zoneId1));
    }
}

```

## JDK8新时间 (Instant)

### 1. 方法 (代替Date类)

**Instant** 时间线上的某个时刻/时间戳

- 通过获取Instant的对象可以拿到此刻的时间，该时间由两部分组成：从1970-01-01 00:00:00 开始走到此刻的总秒数 + 不够1秒的纳秒数



作用：可以用来记录代码的执行时间，或用于记录用户操作某个事件的时间点。

```

/**
 * 目标：掌握Instant的使用。
 */
public class Test5_Instant {
    public static void main(String[] args) {
        // 1、创建Instant的对象，获取此刻时间信息
        Instant now = Instant.now(); // 不可变对象

        // 2、获取总秒数
        long second = now.getEpochSecond();
        System.out.println(second);

        // 3、不够1秒的纳秒数
        int nano = now.getNano();
        System.out.println(nano);
    }
}

```

```

        System.out.println(now);

        Instant instant = now.plusNanos(111);

        // Instant对象的作用：做代码的性能分析，或者记录用户的操作时间点
        Instant now1 = Instant.now();
        // 代码执行。。。
        Instant now2 = Instant.now();

        LocalDateTime l = LocalDateTime.now();
    }
}

```

## JDK8新时间（格式化器）

### 1. 方法（代替SimpleDateFormat类）

#### DateTimeFormatter

方法名	说明
public static DateTimeFormatter ofPattern(时间格式)	获取格式化器对象
public String format(时间对象)	格式化时间

#### LocalDateTime提供的格式化、解析时间的方法

方法名	说明
public String format(DateTimeFormatter formatter)	格式化时间
public static LocalDateTime parse(CharSequence text, DateTimeFormatter formatter)	解析时间

```

/**
 * 目标：掌握JDK 8新增的DateTimeFormatter格式化器的用法。
 */
public class Test6_DateTimeFormatter {
    public static void main(String[] args) {
        // 1、创建一个日期时间格式化器对象出来。
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH:mm:ss");

        // 2、对时间进行格式化
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);

        String rs = formatter.format(now); // 正向格式化
        System.out.println(rs);
    }
}

```

```

// 3、格式化时间，其实还有一种方案。
String rs2 = now.format(formatter); // 反向格式化
System.out.println(rs2);

// 4、解析时间：解析时间一般使用LocalDateTime提供的解析方法来解析。
String dateStr = "2029年12月12日 12:12:11";
LocalDateTime ldt = LocalDateTime.parse(dateStr,
formatter);
System.out.println(ldt);
}
}

```

## JDK8新时间（计算间隔）

### 1. Period类

- 可以用于计算两个 LocalDate对象 相差的年数、月数、天数。

方法名	说明
public static Period between(LocalDate start, LocalDate end)	传入2个日期对象，得到Period对象
public int getYears()	计算隔几年，并返回
public int getMonths()	计算隔几个月，年返回
public int getDays()	计算隔多少天，并返回

```

/**
 * 目标：掌握Period的作用：计算机两个日期相差的年数，月数、天数。
 */
public class Test7_Period {
    public static void main(String[] args) {
        LocalDate start = LocalDate.of(2029, 8, 10);
        LocalDate end = LocalDate.of(2029, 12, 15);

        // 1、创建Period对象，封装两个日期对象。
        Period period = Period.between(start, end);

        // 2、通过period对象获取两个日期对象相差的信息。
        System.out.println(period.getYears());
        System.out.println(period.getMonths());
        System.out.println(period.getDays());
    }
}

```

## 2. Duration类

- 可以用于计算两个时间对象相差的天数、小时数、分钟数、秒数、纳秒数；支持LocalTime、LocalDateTime、Instant等时间。

方法名	说明
public static Duration between(开始时间对象1,截止时间对象2)	传入2个时间对象，得到Duration对象
public long toDays()	计算隔多少天，并返回
public long toHours()	计算隔多少小时，并返回
public long toMinutes()	计算隔多少分，并返回
public long toSeconds()	计算隔多少秒，并返回
public long toMillis()	计算隔多少毫秒，并返回
public long toNanos()	计算隔多少纳秒，并返回

```
public class Test8_Duration {  
    public static void main(String[] args) {  
        LocalDateTime start = LocalDateTime.of(2025, 11, 11,  
11, 10, 10);  
        LocalDateTime end = LocalDateTime.of(2025, 11, 11, 11,  
11, 11);  
        // 1、得到Duration对象  
        Duration duration = Duration.between(start, end);  
  
        // 2、获取两个时间对象间隔的信息  
        System.out.println(duration.toDays()); // 间隔多少天  
        System.out.println(duration.toHours()); // 间隔多少小时  
        System.out.println(duration.toMinutes()); // 间隔多少分  
        System.out.println(duration.toSeconds()); // 间隔多少秒  
        System.out.println(duration.toMillis()); // 间隔多少毫秒  
        System.out.println(duration.toNanos()); // 间隔多少纳秒  
    }  
}
```

## Arrays

### 1. 基本使用

Arrays类是用来操作数组的一个工具类。

## Arrays类提供的的常见方法

方法名	说明
public static String <u>toString</u> (类型[] arr)	返回数组的内容
public static int[] <u>copyOfRange</u> (类型[] arr, 起始索引, 结束索引)	拷贝数组 (指定范围)
public static <u>copyOf</u> (类型[] arr, int newLength)	拷贝数组
public static <u>setAll</u> (double[] array, IntToDoubleFunction generator)	把数组中的原数据改为新数据
public static void <u>sort</u> (类型[] arr)	对数组进行排序(默认是升序排序)

```
/***
 * 目标: 掌握Arrays类的常用方法。
 */
public class ArraysTest1 {
    public static void main(String[] args) {
        // 1、public static String toString(类型[] arr): 返回数组
        // 的内容
        int[] arr = {10, 20, 30, 40, 50, 60};
        System.out.println(Arrays.toString(arr));
        // 输出[10,20,30,40,50,60]

        // 2、public static 类型[] copyOfRange(类型[] arr, 起始索
        // 引, 结束索引): 拷贝数组 (指定范围, 包前不包后)
        int[] arr2 = Arrays.copyOfRange(arr, 1, 4);
        System.out.println(Arrays.toString(arr2)); // 输出
        [20,30,40]

        // 3、public static copyOf(类型[] arr, int newLength):
        // 拷贝数组, 可以指定新数组的长度。
        int[] arr3 = Arrays.copyOf(arr, 10);
        System.out.println(Arrays.toString(arr3));
        // 输出[10,20,30,40,50,60,0,0,0,0]

        // 4、public static setAll(double[] array,
        IntToDoubleFunction generator): 把数组中的原数据改为新数据又存进去。
        double[] prices = {99.8, 128, 100};
        //          0      1      2
        // 把所有的价格都打八折, 然后又存进去。
        Arrays.setAll(prices, new IntToDoubleFunction() {
            @Override
            public double applyAsDouble(int value) { //value会
                取出数组索引
                // value = 0  1  2
            }
        });
    }
}
```

```

        return prices[value] * 0.8;
    }
});

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

// 5、public static void sort(类型[] arr): 对数组进行排序
//（默认是升序排序）
Arrays.sort(prices);
System.out.println(Arrays.toString(prices));
}
}

```

## 2. 自定义排序

**自定义排序规则时，需要遵循的官方约定如下：**

如果认为左边对象 大于 右边对象 应该返回正整数  
 如果认为左边对象 小于 右边对象 应该返回负整数  
 如果认为左边对象 等于 右边对象 应该返回0整数

----> 升序

当Arrays操作自定义的对象数组时，不知道根据什么规则进行排序，此时有两种解决办法。

(1) 让要排序的类实现Comparable接口，同时重写compareTo方法

Arrays的sort方法底层会根据compareTo方法的返回值是正数、负数、还是0来确定谁大、谁小、谁相等。

```

public class Student implements Comparable<Student>{
    private String name;
    private double height;
    private int age;

    //...get、set、空参数构造方法、有参数构造方法...自己补全

    // 指定比较规则
    // this o
    @Override
    public int compareTo(Student o) {
        // 约定1：认为左边对象 大于 右边对象 请您返回正整数
        // 约定2：认为左边对象 小于 右边对象 请您返回负整数
        // 约定3：认为左边对象 等于 右边对象 请您一定返回0
        /* if(this.age > o.age){
            return 1;
        }else if(this.age < o.age){
```

```

        return -1;
    }
    return 0;/*
    //上面的if语句，也可以简化为下面的一行代码
    return this.age - o.age; // 按照年龄升序排列
    // return o.age - this.age; // 按照年龄降序排列
}

@Override
public String toString() {
    return "Student{" +
        "name='" + name + '\'' +
        ", height=" + height +
        ", age=" + age +
        '}';
}

```

(2) 在调用 `Arrays.sort(数组, Comparator比较器);` 时，除了传递数组之外，传递一个Comparator比较器对象。

`Arrays` 的 `sort` 方法底层会根据 `Comparator` 比较器对象的 `compare` 方法方法的返回值是正数、负数、还是0来确定谁大、谁小、谁相等。

```

public class ArraysTest2 {
    public static void main(String[] args) {
        // 目标：掌握如何对数组中的对象进行排序。
        Student[] students = new Student[4];
        students[0] = new Student("蜘蛛精", 169.5, 23);
        students[1] = new Student("紫霞", 163.8, 26);
        students[2] = new Student("紫霞", 163.8, 26);
        students[3] = new Student("至尊宝", 167.5, 24);

        // 2. public static <T> void sort(T[] arr, Comparator<?
super T> c)
        // 参数一：需要排序的数组
        // 参数二：Comparator比较器对象（用来制定对象的比较规则）
        Arrays.sort(students, new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                // 制定比较规则了：左边对象 o1    右边对象 o2
                // 约定1：认为左边对象 大于 右边对象 请您返回正整数
                // 约定2：认为左边对象 小于 右边对象 请您返回负整数
                // 约定3：认为左边对象 等于 右边对象 请您一定返回0
            }
        });
    }
}

```

```

//             if(o1.getHeight() > o2.getHeight()){
//                 return 1;
//             }else if(o1.getHeight() < o2.getHeight()){
//                 return -1;
//             }
//             return 0; // 升序
//             return Double.compare(o1.getHeight(),
o2.getHeight()); // 升序
//             return Double.compare(o2.getHeight(),
o1.getHeight()); // 降序
}
});
System.out.println(Arrays.toString(students));
}
}

```

## 异常

### 1. 异常解决

(1) 使用throws在方法上声明，意思就是告诉下一个调用者，这里面可能有异常啊，你调用时注意一下。

```

public class ExceptionTest1 {
    public static void main(String[] args) throws
ParseException{
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
    Date d = sdf.parse("2028-11-11 10:24");
    System.out.println(d);
}
}

```

(2) 使用try...catch语句块异常进行处理。

```
public class ExceptionTest1 {
    public static void main(String[] args) throws
ParseException{
        try {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-
MM-dd HH:mm:ss");
            Date d = sdf.parse("2028-11-11 10:24");
            System.out.println(d);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

## 2. 自定义异常

需求：判断年龄是否合法

```
// 当类继承自Exception，才能成为一个编译时异常类。
// 当类继承自RuntimeException，成为一个运行时异常类（只有跑起来才报错）
public class AgeIllegalException extends Exception{ //异常类的名字自己取
    public AgeIllegalException() { //无参构造器
    }

    public AgeIllegalException(String message) { //有参构造器，接收报错提示
        super(message);
    }
}

// 右键Generate构造器选前两个即可
```

```
public class ExceptionTest2 {
    public static void main(String[] args) {
        // 需求：保存一个合法的年
        try {
            saveAge2(225);
            System.out.println("saveAge2底层执行是成功的！");
        } catch (AgeIllegalException e) {
            e.printStackTrace(); //嵌套选择try catch，IDEA自动补全
            System.out.println("saveAge2底层执行是出现bug的！");
        }
    }
}
```

```

//2、在方法中对age进行判断，不合法则抛出AgeIllegalException
public static void saveAge(int age) throws
AgeIllegalException{
    if(age > 0 && age < 150){
        System.out.println("年龄被成功保存: " + age);
    }else {
        // 用一个异常对象封装这个问题
        // throw 抛出去这个异常对象
        // throws 用在方法上，抛出方法内部异常至上层
        throw new AgeIllegalRuntimeException("/age is
illegal, your age is " + age);
    }
}

```

逻辑：try先传入年龄，若合法则保存；若不合法，会将异常抛出，当运行时异常抛出即可，当编译时异常只有方法throws后才可以在编译时不报错；出现异常后上层会catch，并打印报错信息

### 1. 如果自定义异常类继承Exception，则是编译时异常。

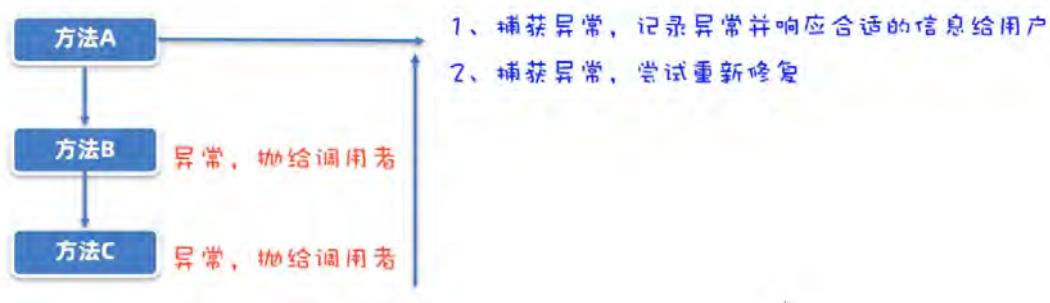
特点：方法中抛出的是编译时异常，必须在方法上使用throws声明，强制调用者处理。

### 2. 如果自定义异常类继承RuntimeException，则运行时异常。

特点：方法中抛出的是运行时异常，不需要在方法上用throws声明。

## 3. 异常处理

### 开发中对于异常的常见处理方式



### (1) 捕获异常，记录异常并响应合适的信息给用户

```

public class ExceptionTest3 {
    public static void main(String[] args) {
        try {
            test1();
        } catch (FileNotFoundException e) {
            System.out.println("您要找的文件不存在！！");
        }
    }
}

```

```

        e.printStackTrace(); // 打印出这个异常对象的信息。记录下来。
    } catch (ParseException e) {
        System.out.println("您要解析的时间有问题了！");
        e.printStackTrace(); // 打印出这个异常对象的信息。记录下来。
    }
}

public static void test1() throws FileNotFoundException,
ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
    Date d = sdf.parse("2028-11-11 10:24:11");
    System.out.println(d);
    test2();
}

public static void test2() throws FileNotFoundException {
    // 读取文件的。
    InputStream is = new FileInputStream("D:/meinv.png");
}
}

```

实际开发中，throws后面建议直接写Exception，这样catch也只需catch(Exception)

## (2) 捕获异常，尝试重新修复

```

public class ExceptionTest4 {
    public static void main(String[] args) {
        // 需求：调用一个方法，让用户输入一个合适的价格返回为止。
        // 尝试修复
        while (true) {
            try {
                System.out.println(getMoney());
                break;
            } catch (Exception e) {
                System.out.println("请您输入合法的数字！！");
            }
        }
    }

    public static double getMoney(){
        Scanner sc = new Scanner(System.in);

```

```

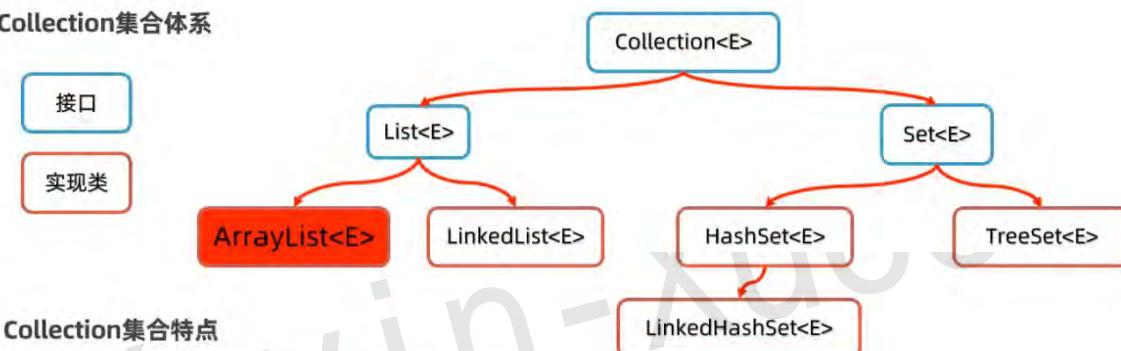
        while (true) {
            System.out.println("请您输入合适的价格: ");
            double money = sc.nextDouble();
            if(money >= 0){
                return money;
            }else {
                System.out.println("您输入的价格是不合适的! ");
            }
        }
    }
}

```

# 集合

## Collection

Collection集合体系



Collection集合特点

- **List系列集合：**添加的元素是**有序**、**可重复**、**有索引**。
  - ◆ ArrayList、LinkedList：有序、可重复、有索引。
- **Set系列集合：**添加的元素是**无序**、**不重复**、**无索引**。
  - ◆ HashSet: 无序、不重复、无索引；
  - ◆ LinkedHashSet: **有序**、不重复、无索引。
  - ◆ TreeSet: 按照大小默认**升序排序**、不重复、无索引。

### 1. 常用方法

Collection的常用方法和遍历方式可以应用到所有单列集合

## Collection的常见方法如下：

方法名	说明
public boolean add(E e)	把给定的对象添加到当前集合中
public void clear()	清空集合中所有的元素
public boolean remove(E e)	把给定的对象在当前集合中删除
public boolean contains(Object obj)	判断当前集合中是否包含给定的对象
public boolean isEmpty()	判断当前集合是否为空
public int size()	返回集合中元素的个数。
public Object[] toArray()	把集合中的元素，存储到数组中

```
Collection<String> c = new ArrayList<>();
//1.public boolean add(E e): 添加元素到集合
c.add("java1");
c.add("java1");
c.add("java2");
c.add("java2");
c.add("java3");
System.out.println(c); //打印: [java1, java1, java2, java2,
java3]

//2.public int size(): 获取集合的大小
System.out.println(c.size()); //5

//3.public boolean contains(Object obj): 判断集合中是否包含某个元素
System.out.println(c.contains("java1")); //true
System.out.println(c.contains("Java1")); //false

//4.pubilc boolean remove(E e): 删除某个元素，如果有多个重复元素只能删除第一个
System.out.println(c.remove("java1")); //true
System.out.println(c); //打印: [java1,java2, java2, java3]

//5.public void clear(): 清空集合的元素
c.clear();
System.out.println(c); //打印: []

//6.public boolean isEmpty(): 判断集合是否为空 是空返回true 反之返回false
System.out.println(c.isEmpty()); //true

//7.public Object[] toArray(): 把集合转换为数组，转换后用Object接收
```

```

Object[] array = c.toArray();
System.out.println(Arrays.toString(array)); // [java1, java2,
java2, java3]

// 8. 如果想把集合转换为指定类型的数组，可以使用下面的代码
String[] array1 = c.toArray(new String[c.size()]);
System.out.println(Arrays.toString(array1)); // [java1, java2,
java2, java3]

// 9. 还可以把一个集合中的元素，添加到另一个集合中
Collection<String> c1 = new ArrayList<>();
c1.add("java1");
c1.add("java2");
Collection<String> c2 = new ArrayList<>();
c2.add("java3");
c2.add("java4");
c1.addAll(c2); // 把c2集合中的全部元素，添加到c1集合中去
System.out.println(c1); // [java1, java2, java3, java4]

```

## 2. 遍历方式

之前学习过的遍历方式，只能遍历List集合，不能遍历Set集合，因为以前的普通for循环遍历需要索引，只有List集合有索引，而Set集合没有索引。

### (1) 迭代器遍历

#### Collection集合获取迭代器的方法

方法名称	说明
Iterator<E> iterator()	返回集合中的迭代器对象，该迭代器对象默认指向当前集合的第一个元素

#### Iterator迭代器中的常用方法

方法名称	说明
boolean hasNext()	询问当前位置是否有元素存在，存在返回true，不存在返回false
E next()	获取当前位置的元素，并同时将迭代器对象指向下一个元素处。

```

Collection<String> c = new ArrayList<>();
c.add("赵敏");
c.add("小昭");
c.add("素素");
c.add("灭绝");
System.out.println(c); // [赵敏, 小昭, 素素, 灭绝]

```

// 第一步：先获取迭代器对象

// 解释：Iterator就是迭代器对象，用于遍历集合的工具

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

//第二步：用于判断当前位置是否有元素可以获取
//解释：hasNext()方法返回true，说明有元素可以获取；反之没有
while(it.hasNext()){
    //第三步：获取当前位置的元素，然后自动指向下一个元素。
    String e = it.next();
    System.out.println(s);
}
```

## (2) 增强for遍历

```
for(数据类型 变量名:数组/集合){

}

Collection<String> c = new ArrayList<>();
c.add("赵敏");
c.add("小昭");
c.add("素素");
c.add("灭绝");

//1. 使用增强for遍历集合
for(String s: c){
    System.out.println(s);
}

//2. 再尝试使用增强for遍历数组
String[] arr = {"迪丽热巴", "古力娜扎", "稀奇哈哈"};
for(String name: arr){
    System.out.println(name);
}
```

## (3) forEach遍历

```
Collection<String> c = new ArrayList<>();
c.add("赵敏");
c.add("小昭");
c.add("素素");
c.add("灭绝");

//调用forEach方法
//由于参数是一个Consumer接口，所以可以传递匿名内部类
c.forEach(new Consumer<String>{
    @Override
```

```

    public void accept(String s){
        System.out.println(s);
    }
});

//也可以使用lambda表达式对匿名内部类进行简化
c.forEach(s->System.out.println(s)); //赵敏, 小昭, 素素, 灭绝

```

## List

### 1. list特有方法

方法名称	说明
void add(int index, E element)	在此集合中的指定位置插入指定的元素
E remove(int index)	删除指定索引处的元素，返回被删除的元素
E set(int index, E element)	修改指定索引处的元素，返回被修改的元素
E get(int index)	返回指定索引处的元素

```

//1. 创建一个ArrayList集合对象（有序、有索引、可以重复）
List<String> list = new ArrayList<>();
list.add("蜘蛛精");
list.add("至尊宝");
list.add("至尊宝");
list.add("牛夫人");
System.out.println(list); //蜘蛛精, 至尊宝, 至尊宝, 牛夫人

//2. public void add(int index, E element): 在某个索引位置插入元素
list.add(2, "紫霞仙子");
System.out.println(list); //蜘蛛精, 至尊宝, 紫霞仙子, 至尊宝, 牛夫人

//3. public E remove(int index): 根据索引删除元素，返回被删除的元素
System.out.println(list.remove(2)); //紫霞仙子
System.out.println(list); //蜘蛛精, 至尊宝, 至尊宝, 牛夫人

//4. public E get(int index): 返回集合中指定位置的元素
System.out.println(list.get(3));

//5. public E set(int index, E e): 修改索引位置处的元素，修改后，会返回原数据
System.out.println(list.set(3, "牛魔王")); //牛夫人
System.out.println(list); //蜘蛛精, 至尊宝, 至尊宝, 牛魔王

```

## 2. 遍历

```
List<String> list = new ArrayList<>();
list.add("蜘蛛精");
list.add("至尊宝");
list.add("糖宝宝");

//1. 普通for循环
for(int i = 0; i < list.size(); i++){
    //i = 0, 1, 2
    String e = list.get(i);
    System.out.println(e);
}

//2. 增强for遍历
for(String s : list){
    System.out.println(s);
}

//3. 迭代器遍历
Iterator<String> it = list.iterator();
while(it.hasNext()){
    String s = it.next();
    System.out.println(s);
}

//4. Lambda表达式遍历
list.forEach(s->System.out.println(s));
```

## ArrayList

### 1. 底层原理

- (1) 利用无参构造器创建的集合，会在底层创建一个默认长度为0的数组。
- (2) 添加第一个元素时，底层会创建一个新的长度为10的数组。
- (3) 存满时，会扩容1.5倍。
- (4) 如果一次添加多个元素，1.5倍还放不下，则新创建的数组的长度以实际为准。

## LinkedList

### 1. 新增方法

LinkedList的底层是双链表实现

方法名称	说明
public void addFirst(E e)	在该列表开头插入指定的元素
public void addLast(E e)	将指定的元素追加到此列表的末尾
public E getFirst()	返回此列表中的第一个元素
public E getLast()	返回此列表中的最后一个元素
public E removeFirst()	从此列表中删除并返回第一个元素
public E removeLast()	从此列表中删除并返回最后一个元素

## 2. 应用场景：设计队列、栈（有push、pop方法）

## Set

**Set系列集合特点：**无序：添加数据的顺序和获取出的数据顺序不一致； 不重复； 无索引；

- HashSet：无序、不重复、无索引。
- LinkedHashSet：有序、不重复、无索引。
- TreeSet：排序、不重复、无索引。

Set常用功能就是继承Collection的

## HashSet

1. 底层原理：哈希表
2. 去重原理

确认相同需要确认两点：一个是hashCode方法用来确定在底层数组中存储的位置相同，另一个是用equals方法判断新添加的元素是否和集合中已有的元素相同。

```
public class Student{
    private String name; //姓名
    private int age; //年龄
    private double height; //身高

    //无参数构造方法
    public Student(){}
    //全参数构造方法
    public Student(String name, int age, double height){
        this.name=name;
        this.age=age;
        this.height=height;
    }
    //...get、set、toString()方法自己补上..
}
```

```

//按快捷键生成hashCode和equals方法（IDEA自动生成）
//alt+insert 选择 hashCode and equals
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;

    Student student = (Student) o;

    if (age != student.age) return false;
    if (Double.compare(student.height, height) != 0)
return false;
    return name != null ? name.equals(student.name) :
student.name == null;
}

@Override
public int hashCode() {
    int result;
    long temp;
    result = name != null ? name.hashCode() : 0;
    result = 31 * result + age;
    temp = Double.doubleToLongBits(height);
    result = 31 * result + (int) (temp ^ (temp >>> 32));
    return result;
}

```

### 3. LinkedHashSet

底层采用的是也是哈希表结构，只不过额外新增了一个双向链表来维护元素的存取顺序。每次添加元素，就和上一个元素用双向链表连接一下。第一个添加的元素是双向链表的头节点，最后一个添加的元素是双向链表的尾节点。

### 4. TreeSet

TreeSet集合的特点是可以对元素进行排序，但是必须指定元素的排序规则。

当存的是自己定义的数据类型，必须指定排序规则。

#### (1) 实现Comparable接口

```

//第一步：先让Student类，实现Comparable接口
//注意：Student类的对象是作为TreeSet集合的元素的
public class Student implements Comparable<Student>{
    private String name;
    private int age;

```

```

private double height;
//无参数构造方法
public Student(){}
//全参数构造方法
public Student(String name, int age, double height){
    this.name=name;
    this.age=age;
    this.height=height;
}
//...get、set、toString()方法自己补上..

//第二步：重写compareTo方法
//按照年龄进行比较，只需要在方法中让this.age和o.age相减就可以。
/*
原理：
在往TreeSet集合中添加元素时，add方法底层会调用compareTo方法，根据该
方法的
结果是正数、负数、还是零，决定元素放在后面、前面还是不存。
*/
@Override
public int compareTo(Student o) {
    //this: 表示将要添加进去的Student对象
    //o: 表示集合中已有的Student对象
    return this.age-o.age;
}
}

```

## (2) 传递比较器对象

```

//创建TreeSet集合时，传递比较器对象排序
/*
原理：当调用add方法时，底层会先用比较器，根据Comparator的compare方法是正
数、负数、还是零，决定谁在后，谁在前，谁不存。
*/
//下面代码中是按照学生的年龄升序排序
Set<Student> students = new TreeSet<>(new Comparator<Student>{
    @Override
    public int compare(Student o1, Student o2){
        //需求：按照学生的身高排序
        return Double.compare(o1,o2);
    }
});

```

## 并发修改异常

- 迭代器遍历集合时，可能存在并发修改异常。该问题是边遍历边删除元素，由于迭代器会向前移动，删除某元素时后一个元素又会向前移动，导致删除不完全。

```
List<String> list = new ArrayList<>();
list.add("王麻子");
list.add("小李子");
list.add("李爱花");
list.add("张全蛋");
list.add("晓李");
list.add("李玉刚");
System.out.println(list); // [王麻子, 小李子, 李爱花, 张全蛋, 晓李,
李玉刚]

//需求：找出集合中带"李"字的姓名，并从集合中删除
Iterator<String> it = list.iterator();
while(it.hasNext()){
    String name = it.next();
    if(name.contains("李")){
        //list.remove(name); //并发修改异常
        it.remove(); //当前迭代器指向谁，就删除谁
    }
}
System.out.println(list);
```

## Collections工具类

### 1. 可变参数

定义：一种特殊形参，定义在方法、构造器的形参列表里，格式是：数据类型...  
参数名称

特点和好处：可以不传数据给它，可以传一个或者多个数据给它，也可以传一个数组给它，因此常常用来灵活的接收数据。

```
public class ParamTest{
    public static void main(String[] args){
        //不传递参数，下面的nums长度则为0， 打印元素是[]
        test();

        //传递3个参数，下面的nums长度为3， 打印元素是[10, 20, 30]
        test(10, 20, 30);

        //传递一个数组，下面数组长度为4， 打印元素是[10, 20, 30, 40]
        int[] arr = new int[]{10, 20, 30, 40}
```

```

        test(arr);
    }

    public static void test(int...nums){
        //可变参数在方法内部，本质上是一个数组
        System.out.println(nums.length);
        System.out.println(Arrays.toString(nums));
        System.out.println("-----");
    }
}

```

## 注意事项

- (1) 一个形参列表中，只能有一个可变参数。
- (2) 一个形参列表中如果有多个参数，可变参数需要写在最后，否则报错。

## 2. Collections工具类

Collections是用来操作Collection的工具类。它提供了一些好用的静态方法，如下

方法名称	说明
public static <T> boolean addAll(Collection<? super T> c, T... elements)	给集合批量添加元素
public static void shuffle(List<?> list)	打乱List集合中的元素顺序
public static <T> void sort(List<T> list)	对List集合中的元素进行升序排序
public static <T> void sort(List<T> list, Comparator<? super T> c)	对List集合中元素，按照比较器对象指定的规则进行排序

```

public class CollectionsTest{
    public static void main(String[] args){
        //1.public static <T> boolean addAll(Collection<?
super T> c, T...e)
        List<String> names = new ArrayList<>();
        Collections.addAll(names, "张三", "王五", "李四", "张麻子");
        System.out.println(names);

        //2.public static void shuffle(List<?> list): 对集合打乱
顺序
        Collections.shuffle(names);
        System.out.println(names);

        //3.public static <T> void sort(List<T> list): 对List集
合排序
        List<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(5);
    }
}

```

```

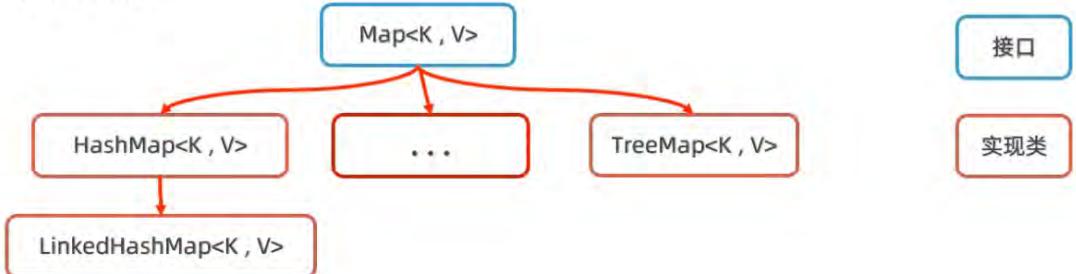
        list.add(2);
        Collections.sort(list);
        System.out.println(list);
    }
}

//当对自定义数据类型的List进行排序，需要像TreeSet一样实现接口

```

## Map

### Map集合体系



### Map集合体系的特点

注意：Map系列集合的特点都是由键决定的，值只是一个附属品，值是不做要求的

- HashMap（由键决定特点）：无序、不重复、无索引； **（用的最多）**
- LinkedHashMap（由键决定特点）：由键决定的特点：**有序**、不重复、无索引。
- TreeMap（由键决定特点）：**按照大小默认升序排序**、不重复、无索引。

#### 1. 定义

Map是双列集合，就是说集合中的元素是一对一对的。Map集合中的每一个元素是以 key=value 的形式存在的，一个 key=value 就称之为一个键值对，而且在 Java 中有一个类叫 Entry 类，Entry 的对象用来表示键值对对象。

Map特点：键不能重复，值可以重复，每一个键只能找到自己对应的值。

```

public class MapTest1 {
    public static void main(String[] args) {
        // Map<String, Integer> map = new HashMap<>(); // 一行
        经典代码。 按照键 无序，不重复，无索引。
        Map<String, Integer> map = new LinkedHashMap<>(); // 有
        序，不重复，无索引。
        map.put("手表", 100);
        map.put("手表", 220); // 后面重复的数据会覆盖前面的数据（键）
        map.put("手机", 2);
        map.put("Java", 2);
        map.put(null, null);
        System.out.println(map);
    }
}

```

```

        Map<Integer, String> map1 = new TreeMap<>(); // 可排序,
不重复, 无索引
        map1.put(23, "Java");
        map1.put(23, "MySQL");
        map1.put(19, "李四");
        map1.put(20, "王五");
        System.out.println(map1);
    }
}

```

## 2. 常用方法

Map的常用方法如下：

方法名称	说明
public V put(K key,V value)	添加元素
public int size()	获取集合的大小
public void clear()	清空集合
public boolean isEmpty()	判断集合是否为空, 为空返回true , 反之
public V get(Object key)	根据键获取对应值
public V remove(Object key)	根据键删除整个元素
public boolean containsKey(Object key)	判断是否包含某个键
public boolean containsValue(Object value)	判断是否包含某个值
public Set<K> keySet()	获取全部键的集合
public Collection<V> values()	获取Map集合的全部值

```

public class MapTest2 {
    public static void main(String[] args) {
        // 1.添加元素：无序, 不重复, 无索引。
        Map<String, Integer> map = new HashMap<>();
        map.put("手表", 100);
        map.put("手表", 220);
        map.put("手机", 2);
        map.put("Java", 2);
        map.put(null, null);
        System.out.println(map);
        // map = {null=null, 手表=220, Java=2, 手机=2}

        // 2. public int size():获取集合的大小
        System.out.println(map.size());

        // 3. public void clear():清空集合
        //map.clear();
        //System.out.println(map);
    }
}

```

```
// 4.public boolean isEmpty(): 判断集合是否为空，为空返回  
true ,反之!  
System.out.println(map.isEmpty());  
  
// 5.public v get(Object key): 根据键获取对应值  
int v1 = map.get("手表");  
System.out.println(v1);  
System.out.println(map.get("手机")); // 2  
System.out.println(map.get("张三")); // null  
  
// 6. public v remove(Object key): 根据键删除整个元素(删除  
键会返回键的值)  
System.out.println(map.remove("手表"));  
System.out.println(map);  
  
// 7.public boolean containsKey(Object key): 判断是否包  
含某个键，包含返回true ,反之  
System.out.println(map.containsKey("手表")); // false  
System.out.println(map.containsKey("手机")); // true  
System.out.println(map.containsKey("java")); // false  
System.out.println(map.containsKey("Java")); // true  
  
// 8.public boolean containsValue(Object value): 判断是  
否包含某个值。  
System.out.println(map.containsValue(2)); // true  
System.out.println(map.containsValue("2")); // false  
  
// 9.public Set<K> keySet(): 获取Map集合的全部键。  
Set<String> keys = map.keySet();  
System.out.println(keys);  
  
// 10.public Collection<v> values(); 获取Map集合的全部值。  
Collection<Integer> values = map.values();  
System.out.println(values);  
  
// 11.把其他Map集合的数据倒入到自己集合中来。(拓展)  
Map<String, Integer> map1 = new HashMap<>();  
map1.put("java1", 10);  
map1.put("java2", 20);  
Map<String, Integer> map2 = new HashMap<>();  
map2.put("java3", 10);  
map2.put("java2", 222);  
map1.putAll(map2); // putAll: 把map2集合中的元素全部倒入一份  
到map1集合中去。  
System.out.println(map1);
```

```
        System.out.println(map2);
    }
}
```

### 3. 遍历方式

#### (1) 先获取键，再获取值

```
/*
 * 目标：掌握Map集合的遍历方式1：键找值
 */
public class MapTest1 {
    public static void main(String[] args) {
        // 准备一个Map集合。
        Map<String, Double> map = new HashMap<>();
        map.put("蜘蛛精", 162.5);
        map.put("蜘蛛精", 169.8);
        map.put("紫霞", 165.8);
        map.put("至尊宝", 169.5);
        map.put("牛魔王", 183.6);
        System.out.println(map);
        // map = {蜘蛛精=169.8, 牛魔王=183.6, 至尊宝=169.5, 紫霞=165.8}

        // 1、获取Map集合的全部键
        Set<String> keys = map.keySet();
        // System.out.println(keys);
        // [蜘蛛精, 牛魔王, 至尊宝, 紫霞]
        // key
        // 2、遍历全部的键，根据键获取其对应的值
        for (String key : keys) {
            // 根据键获取对应的值
            double value = map.get(key);
            System.out.println(key + "====>" + value);
        }
    }
}
```

#### (2) 同时获取键值对

键值对构成的集合为Entry类型，调用entrySet方法可以获取键值对

Map提供的方法	说明
Set<Map.Entry<K, V>> entrySet()	获取所有“键值对”的集合

```
/*
```

```

* 目标：掌握Map集合的第二种遍历方式：键值对。
*/
public class MapTest2 {
    public static void main(String[] args) {
        Map<String, Double> map = new HashMap<>();
        map.put("蜘蛛精", 169.8);
        map.put("紫霞", 165.8);
        map.put("至尊宝", 169.5);
        map.put("牛魔王", 183.6);
        System.out.println(map);
        // map = {蜘蛛精=169.8, 牛魔王=183.6, 至尊宝=169.5, 紫霞=165.8}
        // entries = [(蜘蛛精=169.8), (牛魔王=183.6), (至尊宝=169.5), (紫霞=165.8)]
        // entry = (蜘蛛精=169.8)
        // entry = (牛魔王=183.6)
        // ...

        // 1、调用Map集合提供entrySet方法，把Map集合转换成键值对类型的
        // Set集合
        Set<Map.Entry<String, Double>> entries =
            map.entrySet();
        for (Map.Entry<String, Double> entry : entries) {
            String key = entry.getKey();
            double value = entry.getValue();
            System.out.println(key + "---->" + value);
        }
    }
}

```

### (3) Lambda表达式

```

//遍历map集合，传递Lambda表达式
map.forEach(( k, v) -> {
    System.out.println(k + "---->" + v);
});

```

## 4. 底层原理

HashMap：哈希表

LinkedHashMap：哈希表+双链表

TreeHashMap：红黑树

# JDK8新特性

## Lambda表达式

### 1. 作用

用于简化匿名内部类，即可以代替匿名内部类的写法。

使用要求：Lambda表达式并不能简化所有的匿名内部类，只能简化函数式接口的匿名内部类。函数式接口指有且只有一个抽象方法的接口。

```
//大部分函数式接口上面都可能会有@FunctionalInterface的注解，有该注解一定是函数式接口
//函数式接口
public interface Swimming{
    void swim();
}
```

### 2. 格式

```
(被重写方法的形参列表) -> {
    被重写方法的方法体代码;
}
```

```
public class LambdaTest1 {
    public static void main(String[] args) {
        // 目标：认识Lambda表达式。
        //1. 创建一个Swimming接口的匿名内部类对象
        Swimming s = new Swimming() {
            @Override
            public void swim() {
                System.out.println("学生快乐的游泳~~~~");
            }
        };
        s.swim();

        //2. 使用Lambda表达式对Swimming接口的匿名内部类进行简化
        Swimming s1 = () -> {
            System.out.println("学生快乐的游泳~~~~");
        };
        s1.swim();
    }
}
```

### 3. Lambda表达式省略规则

### 1. Lambda的标准格式

```
(参数类型1 参数名1, 参数类型2 参数名2) -> {  
    ...方法体的代码...  
    return 返回值;  
}
```

### 2. 在标准格式的基础上()中的参数类型可以直接省略

```
(参数名1, 参数名2) -> {  
    ...方法体的代码...  
    return 返回值;  
}
```

### 3. 如果{}总的语句只有一条语句，则{}可以省略、return关键字、以及最后的“;”都可以省略

```
(参数名1, 参数名2) -> 结果
```

### 4. 如果()里面只有一个参数，则()可以省略

```
(参数名) -> 结果
```

```
Arrays.setAll(prices, new IntToDoubleFunction() {  
    @Override  
    public double applyAsDouble(int value) {  
        // value = 0 1 2  
        return prices[value] * 0.8;  
    }  
});
```

//2.需求：对数组中的每一个元素\*0.8，使用Lambda表达式标准写法

```
Arrays.setAll(prices, (int value) -> {  
    return prices[value] * 0.8;  
});
```

//3. 使用Lambda表达式简化格式1--省略参数类型

```
Arrays.setAll(prices, (value) -> {  
    return prices[value] * 0.8;  
});
```

//4. 使用Lambda表达式简化格式2--省略()

```
Arrays.setAll(prices, value -> {  
    return prices[value] * 0.8;  
});
```

//5. 使用Lambda表达式简化格式3--省略{}

```
Arrays.setAll(prices, value -> prices[value] * 0.8 );
```

# Stream

## 1. Stream流使用步骤



## 2. Stream流的创建

- 获取 **集合** 的Stream流,

Collection提供的如下方法	说明
<code>default Stream&lt;E&gt; stream()</code>	获取当前集合对象的Stream流

- 获取 **数组** 的Stream流

Arrays类提供的如下方法	说明
<code>public static &lt;T&gt; Stream&lt;T&gt; stream(T[] array)</code>	获取当前数组的Stream流

Stream类提供的如下方法	说明
<code>public static&lt;T&gt; Stream&lt;T&gt; of(T... values)</code>	获取当前接收数据的Stream流

```
/**  
 * 目标: 掌握Stream流的创建。  
 */  
public class StreamTest2 {  
    public static void main(String[] args) {  
        // 1、如何获取List集合的Stream流?  
        List<String> names = new ArrayList<>();  
        Collections.addAll(names, "张三丰", "张无忌", "周芷若", "赵  
敏", "张强");  
        Stream<String> stream = names.stream();  
  
        // 2、如何获取Set集合的Stream流?  
        Set<String> set = new HashSet<>();  
        Collections.addAll(set, "刘德华", "张曼玉", "蜘蛛精", "马  
德", "德玛西亚");  
        Stream<String> stream1 = set.stream();  
    }  
}
```

```

        stream1.filter(s -> s.contains("德")).forEach(s ->
System.out.println(s));

        // 3、如何获取Map集合的Stream流?
        Map<String, Double> map = new HashMap<>();
        map.put("古力娜扎", 172.3);
        map.put("迪丽热巴", 168.3);
        map.put("马尔扎哈", 166.3);
        map.put("卡尔扎巴", 168.3);

        // 分别获取键和值
        Set<String> keys = map.keySet();
        Stream<String> ks = keys.stream();

        Collection<Double> values = map.values();
        Stream<Double> vs = values.stream();

        // 直接获取entryset
        Set<Map.Entry<String, Double>> entries =
map.entrySet();
        Stream<Map.Entry<String, Double>> kvs =
entries.stream();
        kvs.filter(e -> e.getKey().contains("巴"))
            .forEach(e -> System.out.println(e.getKey()+
"-->" + e.getValue()));

        // 4、如何获取数组的Stream流?
        String[] names2 = {"张翠山", "东方不败", "唐大山", "独孤求
败"};
        Stream<String> s1 = Arrays.stream(names2);
        Stream<String> s2 = Stream.of(names2);
    }
}

```

### 3. Stream中间方法

Stream提供的常用中间方法	说明
<code>Stream&lt;T&gt; filter(Predicate&lt;? super T&gt; predicate)</code>	用于对流中的数据进行过滤。
<code>Stream&lt;T&gt; sorted()</code>	对元素进行升序排序
<code>Stream&lt;T&gt; sorted(Comparator&lt;? super T&gt; comparator)</code>	按照指定规则排序
<code>Stream&lt;T&gt; limit(long maxSize)</code>	获取前几个元素
<code>Stream&lt;T&gt; skip(long n)</code>	跳过前几个元素
<code>Stream&lt;T&gt; distinct()</code>	去除流中重复的元素。
<code>&lt;R&gt; Stream&lt;R&gt; map(Function&lt;? super T, ? extends R&gt; mapper)</code>	对元素进行加工，并返回对应的新流
<code>static &lt;T&gt; Stream&lt;T&gt; concat(Stream a, Stream b)</code>	合并a和b两个流为一个流

```

/**
 * 目标：掌握Stream流提供的常见中间方法。
 */
public class StreamTest3 {
    public static void main(String[] args) {
        List<Double> scores = new ArrayList<>();
        Collections.addAll(scores, 88.5, 100.0, 60.0, 99.0,
9.5, 99.6, 25.0);
        // 需求1：找出成绩大于等于60分的数据，并升序后，再输出。
        scores.stream().filter(s -> s >=
60).sorted().forEach(s -> System.out.println(s));

        List<Student> students = new ArrayList<>();
        Student s1 = new Student("蜘蛛精", 26, 172.5);
        Student s2 = new Student("蜘蛛精", 26, 172.5);
        Student s3 = new Student("紫霞", 23, 167.6);
        Student s4 = new Student("白晶晶", 25, 169.0);
        Student s5 = new Student("牛魔王", 35, 183.3);
        Student s6 = new Student("牛夫人", 34, 168.5);
        Collections.addAll(students, s1, s2, s3, s4, s5, s6);
        // 需求2：找出年龄大于等于23，且年龄小于等于30岁的学生，并按照年龄
降序输出。
        students.stream().filter(s -> s.getAge() >= 23 &&
s.getAge() <= 30)
            .sorted((o1, o2) -> o2.getAge() - o1.getAge())
            .forEach(s -> System.out.println(s));

        // 需求3：取出身高最高的前3名学生，并输出。
        students.stream().sorted((o1, o2) ->
Double.compare(o2.getHeight(), o1.getHeight()))
            .limit(3).forEach(System.out::println);
        System.out.println("-----");
    }
}

```

```

        // 需求4：取出身高倒数的2名学生，并输出。 s1 s2 s3 s4 s5 s6
        students.stream().sorted((o1, o2) ->
Double.compare(o2.getHeight(), o1.getHeight()))
            .skip(students.size() - 2).forEach(System.out::println);

        // 需求5：找出身高超过168的学生叫什么名字，要求去除重复的名字，再输出。
        students.stream().filter(s -> s.getHeight() > 168).map(s -> s.getName())
            .distinct().forEach(s -> System.out.println(s));

        // distinct去重复，自定义类型的对象（希望内容一样就认为重复，重写 hashCode, equals）
        students.stream().filter(s -> s.getHeight() > 168)
            .distinct().forEach(System.out::println);

        Stream<String> st1 = Stream.of("张三", "李四");
        Stream<String> st2 = Stream.of("张三2", "李四2", "王五");
        Stream<String> allst = Stream.concat(st1, st2);
        allst.forEach(System.out::println);
    }
}

```

#### 4. Stream流终结方法

终结方法指使用后，返回的不再是Stream流，不能继续使用中间方法

Stream提供的常用终结方法	说明
void forEach(Consumer action)	对此流运算后的元素执行遍历
long count()	统计此流运算后的元素个数
Optional<T> max(Comparator<? super T> comparator)	获取此流运算后的最大值元素
Optional<T> min(Comparator<? super T> comparator)	获取此流运算后的最小值元素

Stream提供的常用终结方法	说明
R collect(Collector collector)	把流处理后的结果收集到一个指定的集合中去
Object[] toArray()	把流处理后的结果收集到一个数组中去

```

/**
 * 目标：Stream流的终结方法
 */
public class StreamTest4 {

```

```
public static void main(String[] args) {
    List<Student> students = new ArrayList<>();
    Student s1 = new Student("蜘蛛精", 26, 172.5);
    Student s2 = new Student("蜘蛛精", 26, 172.5);
    Student s3 = new Student("紫霞", 23, 167.6);
    Student s4 = new Student("白晶晶", 25, 169.0);
    Student s5 = new Student("牛魔王", 35, 183.3);
    Student s6 = new Student("牛夫人", 34, 168.5);
    Collections.addAll(students, s1, s2, s3, s4, s5, s6);
    // 需求1：请计算出身高超过168的学生有几人。
    long size = students.stream().filter(s ->
        s.getHeight() > 168).count();
    System.out.println(size);

    // 需求2：请找出身高最高的学生对象，并输出。
    Student s = students.stream().max((o1, o2) ->
        Double.compare(o1.getHeight(), o2.getHeight())).get();
    System.out.println(s);

    // 需求3：请找出身高最矮的学生对象，并输出。
    Student ss = students.stream().min((o1, o2) ->
        Double.compare(o1.getHeight(), o2.getHeight())).get();
    System.out.println(ss);

    // 需求4：请找出身高超过170的学生对象，并放到一个新集合中去返回。
    // 流只能收集一次。
    List<Student> students1 = students.stream().filter(a ->
        a.getHeight() > 170).collect(Collectors.toList());
    System.out.println(students1);

    Set<Student> students2 = students.stream().filter(a ->
        a.getHeight() > 170).collect(Collectors.toSet());
    System.out.println(students2);

    // 需求5：请找出身高超过170的学生对象，并把学生对象的名字和身高，存入到一个Map集合返回。
    Map<String, Double> map =
        students.stream().filter(a -> a.getHeight() > 170)
            .distinct().collect(Collectors.toMap(a -> a.getName(), a -> a.getHeight()));
    System.out.println(map);

    // Object[] arr = students.stream().filter(a ->
    //     a.getHeight() > 170).toArray();
```

```

        Student[] arr = students.stream().filter(a ->
a.getHeight() > 170).toArray(len -> new Student[len]);
        System.out.println(Arrays.toString(arr));
    }
}

```

# IO

## File

### 1. 介绍

File是java.io包下的类， File类对象用于代表当前操作系统的文件或文件夹。

File类提供的方法能获取文件信息、判断文件类型、创建文件/文件夹、删除文件/文件夹，但其只能对文件本身进行操作，不能读写文件里面存储的数据。

File负责操作文件本身，而IO流负责读写数据。

### 2. 创建对象

#### 创建File类的对象

构造器	说明
public File(String pathname)	根据文件路径创建文件对象
public File(String parent, String child)	根据父路径和子路径名字创建文件对象
public File(File parent, String child)	根据父路径对应文件对象和子路径名字创建文件对象

路径中"\\"要写成"\\\"， 路径中"/"可以直接用  
因为若文件名中有n等，会出现\n变为转义字符

```

/**
 * 目标：掌握File创建对象，代表具体文件的方案。
 */
public class FileTest1 {
    public static void main(String[] args) {
        // 1、创建一个File对象，指代某个具体的文件。
        File f1 = new File("D:/resource/ab.txt");
        File f1 = new File("D:\\resource\\ab.txt");
        // 路径分隔符，自动选择是正斜杠还是反斜杠
        File f1 = new File("D:" + File.separator +"resource" +
File.separator + "ab.txt");
        System.out.println(f1.length()); // 文件大小

        File f2 = new File("D:/resource");
        System.out.println(f2.length());
    }
}

```

```

    // 注意: File对象可以指代一个不存在的文件路径
    File f3 = new File("D:/resource/aaaa.txt");
    System.out.println(f3.length()); // length为0
    System.out.println(f3.exists()); // false

    // 我现在要定位的文件是在模块中, 应该怎么定位呢?
    // 绝对路径: 带盘符的
    // File f4 = new File("D:\\code\\javasepromax\\file-
    io-app\\src\\itheima.txt");
    // 相对路径(重点): 不带盘符, 默认是直接去工程下寻找文件的。这里
    project名字是file-io-app
    File f4 = new File("file-io-app\\src\\itheima.txt");
    System.out.println(f4.length());
}
}

```

### 3. 判断和获取方法

方法名称	说明
public boolean exists()	判断当前文件对象, 对应的文件路径是否存在, 存在返回true
public boolean isFile()	判断当前文件对象指代的是不是文件, 是文件返回true, 反之。
public boolean isDirectory()	判断当前文件对象指代的是不是文件夹, 是文件夹返回true, 反之。
public String getName()	获取文件的名称(包含后缀)
public long length()	获取文件的大小, 返回字节个数
public long lastModified()	获取文件的最后修改时间。
public String getPath()	获取创建文件对象时, 使用的路径
public String getAbsolutePath()	获取绝对路径

```

/**
 * 目标: 掌握File提供的判断文件类型、获取文件信息功能
 */
public class FileTest2 {
    public static void main(String[] args) throws
UnsupportedEncodingException {
        // 1. 创建文件对象, 指代某个文件
        File f1 = new File("D:/resource/ab.txt");
        //File f1 = new File("D:/resource/");

        // 2. public boolean exists(): 判断当前文件对象, 对应的文件路
        //径是否存在, 存在返回true.
        System.out.println(f1.exists());

        // 3. public boolean isFile() : 判断当前文件对象指代的是否是
        //文件, 是文件返回true, 反之。
    }
}

```

```

System.out.println(f1.isFile());

// 4. public boolean isDirectory() : 判断当前文件对象指代的是是否是文件夹，是文件夹返回true，反之。
System.out.println(f1.isDirectory());

// 5.public String getName(): 获取文件的名称（包含后缀）
System.out.println(f1.getName());

// 6.public long length(): 获取文件的大小，返回字节个数
System.out.println(f1.length());

// 7.public long lastModified(): 获取文件的最后修改时间。
long time = f1.lastModified();
SimpleDateFormat sdf = new
SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
System.out.println(sdf.format(time));

// 8.public String getPath(): 获取创建文件对象时，使用的路径
File f2 = new File("D:\\resource\\ab.txt");
File f3 = new File("file-io-app\\src\\itheima.txt");
System.out.println(f2.getPath());
System.out.println(f3.getPath());

// 9.public String getAbsolutePath(): 获取绝对路径
System.out.println(f2.getAbsolutePath());
System.out.println(f3.getAbsolutePath());
}
}

```

#### 4. 创建和删除

##### File类创建文件的功能

方法名称	说明
public boolean createNewFile()	创建一个新的空的文件
public boolean mkdir()	只能创建一级文件夹
public boolean mkdirs()	可以创建多级文件夹

##### File类删除文件的功能

方法名称	说明
public boolean delete()	删除文件、空文件夹

/\*\*

```

* 目标：掌握File创建和删除文件相关的方法。
*/
public class FileTest3 {
    public static void main(String[] args) throws Exception {
        // 1、public boolean createNewFile(): 创建一个新文件（文件
        // 内容为空），创建成功返回true，反之。
        File f1 = new File("D:/resource/itheima2.txt");
        System.out.println(f1.createNewFile());

        // 2、public boolean mkdir(): 用于创建文件夹，注意：只能创建
        // 一级文件夹
        File f2 = new File("D:/resource/aaa");
        System.out.println(f2.mkdir());

        // 3、public boolean mkdirs(): 用于创建文件夹，注意：可以创建
        // 多级文件夹
        File f3 = new
        File("D:/resource/bbb/ccc/ddd/eee/fff/ggg");
        System.out.println(f3.mkdirs());

        // 3、public boolean delete(): 删除文件，或者空文件，注意：不
        // 能删除非空文件夹。
        System.out.println(f1.delete());
        System.out.println(f2.delete());
        File f4 = new File("D:/resource");
        System.out.println(f4.delete());
    }
}

```

## 5. 遍历文件夹

方法名称	说明
public String[] list()	获取当前目录下所有的“一级文件名称”到一个字符串数组中去返回。
public File[] listFiles()	获取当前目录下所有的“一级文件对象”到一个文件对象数组中去返回（重点）

```

/**
 * 目标：掌握File提供的遍历文件夹的方法。
*/
public class FileTest4 {
    public static void main(String[] args) {
        // 1、public String[] list(): 获取当前目录下所有的“一级文件名
        // 称”到一个字符串数组中去返回。
        File f1 = new File("D:\\course\\待研发内容");
        String[] names = f1.list();
        for (String name : names) {

```

```

        System.out.println(name);
    }

    // 2、public File[] listFiles(): (重点) 获取当前目录下所有的"一级文件对象"到一个文件对象数组中去返回 (重点)
    File[] files = f1.listFiles();
    for (File file : files) {
        System.out.println(file.getAbsolutePath());
    }

    File f = new File("D:/resource/aaa");
    File[] files1 = f.listFiles();
    System.out.println(Arrays.toString(files1));
}
}

```

## 6. 递归文件搜索

```

/**
 * 目标: 掌握文件搜索的实现。
 */
public class RecursionTest3 {
    public static void main(String[] args) throws Exception {
        searchFile(new File("D:/"), "QQ.exe");
    }
}

/**
 * 去目录下搜索某个文件
 * @param dir 目录
 * @param fileName 要搜索的文件名称
 */
public static void searchFile(File dir, String fileName)
throws Exception {
    // 1、把非法的情况都拦截住
    if(dir == null || !dir.exists() || dir.isFile()){
        return; // 代表无法搜索
    }

    // 2、dir不是null,存在, 一定是目录对象。
    // 获取当前目录下的全部一级文件对象。
    File[] files = dir.listFiles();

    // 3、判断当前目录下是否存在一级文件对象, 以及是否可以拿到一级文件
    // 对象。
    if(files != null && files.length > 0){
        // 4、遍历全部一级文件对象。
    }
}

```

```

        for (File f : files) {
            // 5、判断文件是否是文件,还是文件夹
            if(f.isFile()){
                // 是文件, 判断这个文件名是否是我们要找的
                if(f.getName().contains(fileName)){
                    System.out.println("找到了: " +
f.getAbsolutePath());
                    Runtime runtime =
Runtime.getRuntime();
                    runtime.exec(f.getAbsolutePath());
                }
            }else {
                // 是文件夹, 继续重复这个过程(递归)
                searchFile(f, fileName);
            }
        }
    }
}

```

## 字符集

### 1. 字符集归纳

**ASCII**字符集：《美国信息交换标准代码》，包含英文字母、数字、标点符号、控制字符

特点：1个字符占1个字节

**GBK**字符集：中国人自己的字符集，兼容**ASCII**字符集，还包含2万多个汉字

特点：1个字母占用1个字节；1个汉字占用2个字节

**Unicode**字符集：包含世界上所有国家的文字，有三种编码方案，最常用的是**UTF-8**

**UTF-8**编码方案：英文字母、数字占1个字节兼容(**ASCII**编码)、汉字字符占3个字节

#### UTF-8编码方式(二进制)

0xxxxxxx (ASCII码)

110xxxxx 10xxxxxx

1110xxxx 10xxxxxx 10xxxxxx

11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

确定的前缀

UTF-8编码以1-4个字节表示，每种长度有

### 2. 编码和解码

编码：把字符按照指定字符集编码成字节。

解码：把字节按照指定字符集解码成字节。

Java代码完成对字符的编码

String提供了如下方法	说明
byte[] getBytes()	使用平台的默认字符集将该 String编码为一系列字节，将结果存储到新的字节数组中
byte[] getBytes(String charsetName)	使用指定的字符集将该 String编码为一系列字节，将结果存储到新的字节数组中

Java代码完成对字符的解码

String提供了如下方法	说明
String(byte[] bytes)	通过使用平台的默认字符集解码指定的字节数组来构造新的 String
String(byte[] bytes, String charsetName)	通过指定的字符集解码指定的字节数组来构造新的 String

```
public class Test {
    public static void main(String[] args) throws Exception {
        // 1、编码
        String data = "a我b";
        byte[] bytes = data.getBytes(); // 默认是按照平台字符集（UTF-8）进行编码的。
        System.out.println(Arrays.toString(bytes));

        // 按照指定字符集进行编码。
        byte[] bytes1 = data.getBytes("GBK");
        System.out.println(Arrays.toString(bytes1));

        // 2、解码
        String s1 = new String(bytes); // 按照平台默认编码（UTF-8）解码
        System.out.println(s1);

        String s2 = new String(bytes1, "GBK");
        System.out.println(s2);
    }
}
```

# IO流简介

## IO流的分类

按流的方向分为：

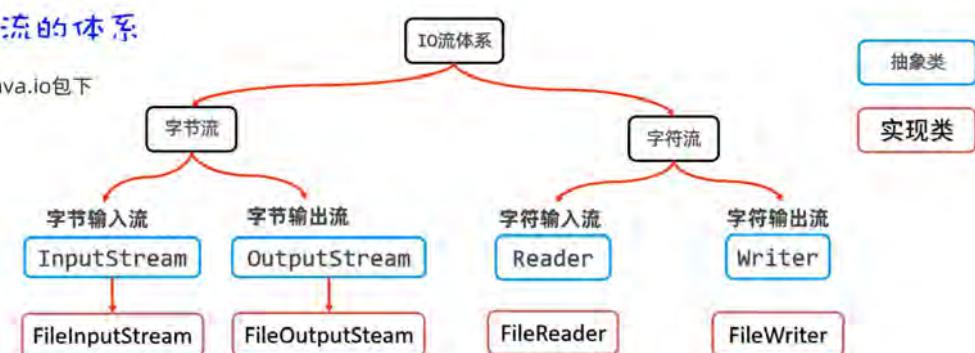


按流中数据的最小单位，分为：



## IO流的体系

● Java.io包下



字节流：适合一切文件的拷贝。

字符流：适合做文本文件的读写。

## FileInputStream

### 1. 作用

以内存为基准，可以把磁盘文件中的数据以字节形式读入到内存中去。

### 2. 读取

构造器	说明
public <b>FileInputStream(File file)</b>	创建字节输入流管道与源文件接通
public <b>FileInputStream(String pathname)</b>	创建字节输入流管道与源文件接通

方法名称	说明
public int <b>read()</b>	每次读取一个字节返回，如果发现没有数据可读会返回-1.
public int <b>read(byte[] buffer)</b>	每次用一个字节数组去读取数据，返回字节数组读取了多少个字节，如果发现没有数据可读会返回-1.

(1) 每次读取一个字节

```
public class FileInputStreamTest1 {
```

```

public static void main(String[] args) throws Exception {
    // 1、创建文件字节输入流管道，与源文件接通。
    InputStream is = new FileInputStream("file-io-
app\\src\\itheima01.txt"));

    // 2、开始读取文件的字节数据。
    // public int read():每次读取一个字节返回，如果没有数据了，返
回-1。
    int b; // 用于记住读取的字节。
    while ((b = is.read()) != -1){
        System.out.print((char) b);
    }

    //3、流使用完毕之后，必须关闭！释放系统资源！
    is.close();
}

//缺点：无法读取汉字

```

## (2) 每次读取多个字节（字符数组）

```

public class FileInputStreamTest2 {
    public static void main(String[] args) throws Exception {
        // 1、创建一个字节输入流对象代表字节输入流管道与源文件接通。
        InputStream is = new FileInputStream("file-io-
app\\src\\itheima02.txt");

        // 2、开始读取文件中的字节数据：每次读取多个字节。
        // public int read(byte b[]) throws IOException
        // 每次读取多个字节到字节数组中去，返回读取的字节数量，读取完毕会
        返回-1。

        // 3、使用循环改造。
        byte[] buffer = new byte[3];
        int len; // 记住每次读取了多少个字节。 abc 66
        while ((len = is.read(buffer)) != -1){
            // 注意：读取多少，倒出多少。
            String rs = new String(buffer, 0 , len);
            System.out.print(rs);
        }
        // 性能得到明显的提升！！
        // 这种方案也不能避免读取汉字输出乱码的问题！！

        is.close(); // 关闭流
    }
}

```

```
}

//缺点：读取汉字还是有可能乱码
```

### (3) 读取全部字节

```
// 1、一次性读取完文件的全部字节到一个字节数组中去。
// 创建一个字节输入流管道与源文件接通
InputStream is = new FileInputStream("file-io-
app\\src\\itheima03.txt");

// 2、准备一个字节数组，大小与文件的大小正好一样大。
File f = new File("file-io-app\\src\\itheima03.txt");
long size = f.length();
byte[] buffer = new byte[(int) size];

int len = is.read(buffer);
System.out.println(new String(buffer));

//3、关闭流
is.close();
```

```
// 1、一次性读取完文件的全部字节到一个字节数组中去。
// 创建一个字节输入流管道与源文件接通
InputStream is = new FileInputStream("file-io-
app\\src\\itheima03.txt");

//2、调用方法读取所有字节，返回一个存储所有字节的字节数组。
byte[] buffer = is.readAllBytes();
System.out.println(new String(buffer));

//3、关闭流
is.close();
```

## FileOutputStream

### 1. 作用

以内存为基准，把内存中的数据以字节的形式写出到文件中去。

构造器	说明
public FileOutputStream(File file)	创建字节输出流管道与源文件对象接通
public FileOutputStream(String filepath)	创建字节输出流管道与源文件路径接通
public FileOutputStream(File file, boolean append)	创建字节输出流管道与源文件对象接通, 可追加数据
public FileOutputStream(String filepath, boolean append)	创建字节输出流管道与源文件路径接通, 可追加数据
方法名称	说明
public void write(int a)	写一个字节出去
public void write(byte[] buffer)	写一个字节数组出去
public void write(byte[] buffer, int pos, int len)	写一个字节数组的一部分出去。
public void close() throws IOException	关闭流。

```
/*
 * 目标: 掌握文件字节输出流FileOutputStream的使用。
 */
public class FileOutputStreamTest4 {
    public static void main(String[] args) throws Exception {
        // 1、创建一个字节输出流管道与目标文件接通。
        // 覆盖管道: 覆盖之前的数据
        // OutputStream os =
        //         new FileOutputStream("file-io-
        app/src/itheima04out.txt");

        // 追加数据的管道, 参数为true表示写数据时是在原有数据上追加
        OutputStream os =
            new FileOutputStream("file-io-
        app/src/itheima04out.txt", true);

        // 2、开始写字节数据出去了
        os.write(97); // 97就是一个字节, 代表a
        os.write('b'); // 'b'也是一个字节
        // os.write('磊'); // [ooo] 默认只能写出去一个字节

        byte[] bytes = "我爱你中国abc".getBytes();
        os.write(bytes);

        os.write(bytes, 0, 15);

        // 换行符
        os.write("\r\n".getBytes());

        os.close(); // 关闭流
    }
}
```

# 文件复制

## 1. 字节流适合做一切文件的复制操作

```
/*
 * 目标：使用字节流完成对文件的复制操作。
 */
public class CopyTest5 {
    public static void main(String[] args) throws Exception {
        // 需求：复制照片。
        // 1、创建一个字节输入流管道与源文件接通
        InputStream is = new FileInputStream("D:/resource/meinv.png");
        // 2、创建一个字节输出流管道与目标文件接通。
        OutputStream os = new FileOutputStream("C:/data/meinv.png");

        // 3、创建一个字节数组，负责转移字节数据。
        byte[] buffer = new byte[1024]; // 1KB.
        // 4、从字节输入流中读取字节数据，写出去到字节输出流中。读多少写出
        // 多少。
        int len; // 记住每次读取了多少个字节。
        while ((len = is.read(buffer)) != -1){
            os.write(buffer, 0, len);
        }
        os.close();
        is.close();
        System.out.println("复制完成！！");
    }
}
```

# 资源释放

## 1. try...catch...finally

```
try{
    //有可能产生异常的代码
} catch(异常类 e){
    //处理异常的代码
} finally{
    //释放资源的代码
    //finally里面的代码有一个特点，不管异常是否发生，finally里面的代码都
    //会执行。
}
```

finally作用：一般用于在程序执行完后进行资源的释放操作。

## 2. try-with-resource

```
try(资源对象1; 资源对象2;){ //这里可以写流对象
    使用资源的代码
}catch(异常类 e){
    处理异常的代码
}

//注意：注意到没有，这里没有释放资源的代码。它会自动是否资源

public class Test3 {
    public static void main(String[] args) {
        try (
            // 1、创建一个字节输入流管道与源文件接通
            InputStream is = new
FileInputStream("D:/resource/meinv.png");
            // 2、创建一个字节输出流管道与目标文件接通。
            OutputStream os = new
FileOutputStream("C:/data/meinv.png");
        ){
            // 3、创建一个字节数组，负责转移字节数据。
            byte[] buffer = new byte[1024]; // 1KB.
            // 4、从字节输入流中读取字节数据，写出去到字节输出流中。读多
少写出去多少。
            int len; // 记住每次读取了多少个字节。
            while ((len = is.read(buffer)) != -1){
                os.write(buffer, 0, len);
            }
            System.out.println(conn);
            System.out.println("复制完成！！");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## 字符流(FileReader/Writer)

1. 字符流适合读取文本文件数据。

2. FileReader

作用：以内存为基准，可以把文件中的数据以字符的形式读入到内存中去。

构造器	说明
public FileReader(File file)	创建字符输入流管道与源文件接通
public FileReader(String pathname)	创建字符输入流管道与源文件接通

方法名称	说明
public int read()	每次读取一个字符返回，如果发现没有数据可读会返回-1.
public int read(char[] buffer)	每次用一个字符数组去读取数据，返回字符数组读取了多少个字符，如果发现没有数据可读会返回-1.

```
/*
 * 目标：掌握文件字符输入流。
 */
public class FileReaderTest1 {
    public static void main(String[] args)  {
        try ( // try-with-resource形式
            // 1、创建一个文件字符输入流管道与源文件接通
            Reader fr = new FileReader("io-
app2\\src\\itheima01.txt");
        ){
            // 2、一个字符一个字符的读（性能较差）
            int c; // 记住每次读取的字符编号。
            while ((c = fr.read()) != -1){
                System.out.print((char) c);
            }
            // 每次读取一个字符的形式，性能肯定是比较差的。
            // 3、每次读取多个字符。（性能是比较不错的！）
            char[] buffer = new char[3];
            int len; // 记住每次读取了多少个字符。
            while ((len = fr.read(buffer)) != -1){
                // 读取多少倒出多少
                System.out.print(new String(buffer, 0, len));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### 3. FileWriter

作用：以内存为基准，把内存中的数据以字符的形式写出到文件中去。

构造器	说明
public FileWriter(File file)	创建字节输出流管道与源文件对象接通
public FileWriter(String filepath)	创建字节输出流管道与源文件路径接通
public FileWriter(File file, boolean append)	创建字节输出流管道与源文件对象接通, 可追加数据
public FileWriter(String filepath, boolean append)	创建字节输出流管道与源文件路径接通, 可追加数据
方法名称	说明
void write(int c)	写一个字符
void write(String str)	写一个字符串
void write(String str, int off, int len)	写一个字符串的一部分
void write(char[] cbuf)	写入一个字符数组
void write(char[] cbuf, int off, int len)	写入字符数组的一部分

```

/**
 * 目标: 掌握文件字符输出流: 写字符数据出去
 */
public class FileWriterTest2 {
    public static void main(String[] args) {
        try {
            // 0、创建一个文件字符输出流管道与目标文件接通。
            // 覆盖管道
            // writer fw = new FileWriter("io-
app2/src/itheima02out.txt");
            // 追加数据的管道
            writer fw = new FileWriter("io-
app2/src/itheima02out.txt", true);
        }{
            // 1. public void write(int c):写一个字符出去
            fw.write('a');
            fw.write(97);
            //fw.write('磊'); // 写一个字符出去
            fw.write("\r\n"); // 换行

            // 2. public void write(string c)写一个字符串出去
            fw.write("我爱你中国abc");
            fw.write("\r\n");

            // 3. public void write(string c ,int pos ,int
len):写字符串的一部分出去
            fw.write("我爱你中国abc", 0, 5);
            fw.write("\r\n");

            // 4. public void write(char[] buffer):写一个字符数组
出去
            char[] buffer = {'黑', '马', 'a', 'b', 'c'};
            fw.write(buffer);
        }
    }
}

```

```

        fw.write("\r\n");
    }
    // 5. public void write(char[] buffer ,int pos ,int len):写字符数组的一部分出去
    fw.write(buffer, 0, 2);
    fw.write("\r\n");
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}

```

## 注意事项

FileWriter写完数据之后，必须刷新或者关闭，写出去的数据才能生效。

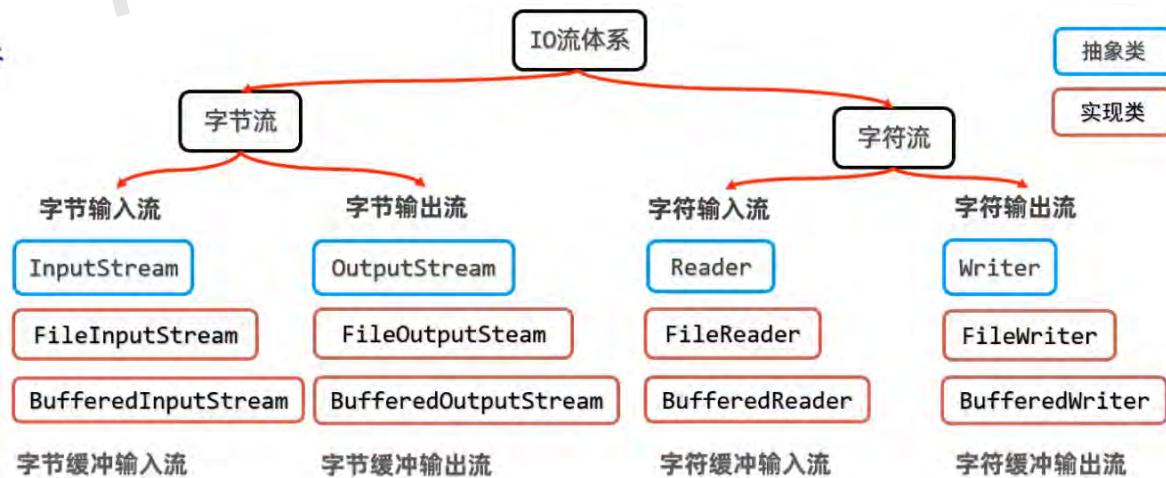
方法名称	说明
public void flush() throws IOException	刷新流，就是将内存中缓存的数据立即写到文件中去生效！
public void close() throws IOException	关闭流的操作，包含了刷新！

FileWriter是把内容先写进缓冲区，刷新后才会从缓冲区写入文件。

用flush()刷新后的对象可以继续写，关闭流后的对象不能继续写。

当缓冲区被写满，会自动写入文件中去。

## 缓冲流



### 1. 作用

对原始流进行包装，以提高原始流读写数据的性能。

### 2. 原理

在内存中开辟较大的输入缓冲区和输出缓冲区，这样内存外存交互次数减少，更多操作是在内存中进行，速度加快。

### 3. 字节缓冲流

构造器	说明
<code>public BufferedInputStream(InputStream is)</code>	把低级的字节输入流包装成一个高级的缓冲字节输入流，从而提高读数据的性能
<code>public BufferedOutputStream(OutputStream os)</code>	把低级的字节输出流包装成一个高级的缓冲字节输出流，从而提高写数据的性能

```
public class BufferedInputStreamTest1 {
    public static void main(String[] args) {
        try {
            InputStream is = new FileInputStream("io-
app2/src/itheima01.txt");
            // 1、定义一个字节缓冲输入流包装原始的字节输入流
            InputStream bis = new BufferedInputStream(is);

            OutputStream os = new FileOutputStream("io-
app2/src/itheima01_bak.txt");
            // 2、定义一个字节缓冲输出流包装原始的字节输出流
            OutputStream bos = new
BufferedOutputStream(os);
        }{

            byte[] buffer = new byte[1024];
            int len;
            while ((len = bis.read(buffer)) != -1){
                bos.write(buffer, 0, len);
            }
            System.out.println("复制完成！！");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### 4. 字符缓冲流

- 作用：自带8K（8192）的字符缓冲池，可以提高字符输入流读取字符数据的性能。

构造器	说明
<code>public BufferedReader(Reader r)</code>	把低级的字符输入流包装成字符缓冲输入流管道，从而提高字符输入流读字符数据的性能

字符缓冲输入流新增的功能：按照行读取字符

方法	说明
<code>public String readLine()</code>	读取一行数据返回，如果没有数据可读了，会返回null

```

public class BufferedReaderTest2 {
    public static void main(String[] args) {
        try {
            Reader fr = new FileReader("io-
app2\\src\\itheima04.txt");
                // 创建一个字符缓冲输入流包装原始的字符输入流
            BufferedReader br = new BufferedReader(fr);
        }{
            String line; // 记住每次读取的一行数据
            while ((line = br.readLine()) != null){
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

### BufferedWriter(字符缓冲输出流)

- 作用：自带8K的字符缓冲池，可以提高字符输出流写字符数据的性能。

构造器	说明
public BufferedWriter(Writer r)	把低级的字符输出流包装成一个高级的缓冲字符输出流管道，从而提高字符输出流写数据的性能

### 字符缓冲输出流新增的功能：换行

方法	说明
public void newLine()	换行

ps：缓冲流的性能不一定比低级流高，其实低级流自己加一个数组，性能其实不差。只不过缓冲流帮你加了一个相对而言大小比较合理的数组。

## 转换流

### 1. 作用

将字节流转换为字符流，并且可以指定编码方案。

### 2. InputStreamReader

字符输入转换流，是Reader的子类，解决不同编码时，字符流读取文本内容乱码的问题。

解决思路：先获取文件的原始字节流，再将其按真实的字符集编码转成字符输入流，这样字符输入流中的字符就不乱码了。

构造器	说明
<code>public InputStreamReader(InputStream is)</code>	把原始的字节输入流，按照代码默认编码转成字符输入流（与直接用FileReader的效果一样）
<code>public InputStreamReader(InputStream is, String charset)</code>	把原始的字节输入流，按照指定字符集编码转成字符输入流（重点）

```

public class InputStreamReaderTest2 {
    public static void main(String[] args) {
        try {
            // 1、得到文件的原始字节流（GBK的字节流形式）
            InputStream is = new FileInputStream("io-
app2/src/itheima06.txt");
            // 2、把原始的字节输入流按照指定的字符集编码转换成字符输入流
            Reader isr = new InputStreamReader(is, "GBK");
            // 3、把字符输入流包装成缓冲字符输入流
            BufferedReader br = new BufferedReader(isr);
        }{
            String line;
            while ((line = br.readLine()) != null){
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

### 3. OutputStreamWriter

是Writer的子类，也是不能单独使用的，它内部需要封装一个OutputStream的子类对象，再指定一个编码表，如果不指定编码表，默认会按照UTF-8形式进行转换。

```

public class OutputStreamWriterTest3 {
    public static void main(String[] args) {
        // 指定写出去的字符编码。
        try {
            // 1、创建一个文件字节输出流
            OutputStream os = new FileOutputStream("io-
app2/src/itheima07out.txt");
            // 2、把原始的字节输出流，按照指定的字符集编码转换成字符输出转换流。
        }
    }
}

```

```

        Writer osw = new OutputStreamWriter(os,
        "GBK");
        // 3、把字符输出流包装成缓冲字符输出流
        BufferedWriter bw = new BufferedWriter(osw);
    }
    bw.write("我是中国人abc");
    bw.write("我爱你中国123");

} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 打印流

### 1. 作用

实现更方便、更高效的打印数据出去，能实现打印啥出去就是啥出去。比如97，打印出去就是97而不是a。

### 2. PrintStream和PrintWriter

打印流有两个，一个是字节打印流PrintStream，一个是字符打印流PrintWriter。

#### PrintStream提供的打印数据的方案

构造器	说明
public PrintStream(OutputStream/File/String)	打印流直接通向字节输出流/文件/文件路径
public PrintStream(String fileName, Charset charset)	可以指定写出去的字符编码
public PrintStream(OutputStream out, boolean autoFlush)	可以指定实现自动刷新
public PrintStream(OutputStream out, boolean autoFlush, String encoding)	可以指定实现自动刷新，并可指定字符的编码

方法	说明
public void println(Xxx xx)	打印任意类型的数据出去
public void write(int/byte[]/byte[]一部分)	可以支持写字节数据出去

#### PrintWriter提供的打印数据的方案

构造器	说明
public PrintWriter(OutputStream/Writer/File/String)	打印流直接通向字节输出流/文件/文件路径
public PrintWriter(String fileName, Charset charset)	可以指定写出去的字符编码
public PrintWriter(OutputStream out/Writer, boolean autoFlush)	可以指定实现自动刷新
public PrintWriter(OutputStream out, boolean autoFlush, String encoding)	可以指定实现自动刷新，并可指定字符的编码

```

public class PrintTest1 {
    public static void main(String[] args) {

```

```

try (
    // 1、创建一个打印流管道
    PrintStream ps =
        new PrintStream("io-
app2/src/itheima08.txt", charset.forName("GBK"));
    PrintStream ps =
        new PrintStream("io-
app2/src/itheima08.txt");
    PrintWriter ps =
        new PrintWriter(new
FileOutputStream("io-app2/src/itheima08.txt", true));
) {
    ps.print(97); //文件中显示的就是:97
    ps.print('a'); //文件中显示的就是:a
    ps.println("我爱你中国abc"); //文件中显示的就是:我爱
你中国abc
    ps.println(true); //文件中显示的就是:true
    ps.println(99.5); //文件中显示的就是99.5

    ps.write(97); //文件中显示a, 发现和前面println方法的
区别了吗?
}

```

### 3. 重定向

System.out.println默认打印位置是控制台，通过setOut方法可以重定向打印位置。

```

public class PrintTest2 {
    public static void main(String[] args) {
        System.out.println("老骥伏枥");
        System.out.println("志在千里");

        try ( PrintStream ps = new PrintStream("io-
app2/src/itheima09.txt"); ) {
            // 把系统默认的打印流对象改成自己设置的打印流
            System.setOut(ps);

            System.out.println("烈士暮年");
            System.out.println("壮心不已");
        } catch (Exception e) {

```

```

        e.printStackTrace();
    }
}
}

```

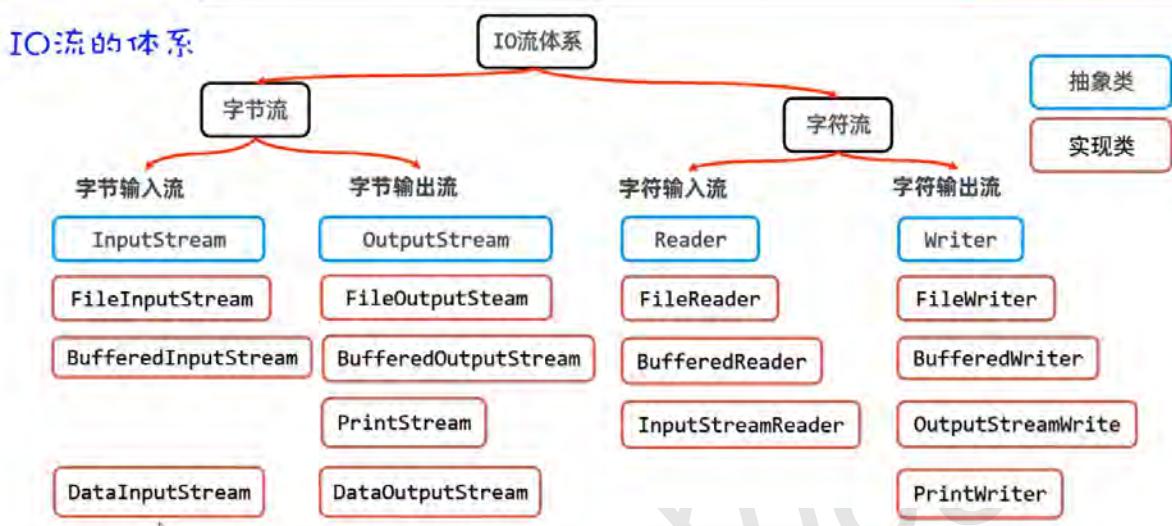
重定向代码

```

PrintStream ps = new PrintStream("文件地址");
System.setOut(ps);

```

## 数据流



### 1. DataOutputStream

允许把数据和其类型一并写出去。

构造器	说明
public DataOutputStream(OutputStream out)	创建新数据输出流包装基础的字节输出流
方法	说明
public final void writeByte(int v) throws IOException	将byte类型的数据写入基础的字节输出流
public final void writeInt(int v) throws IOException	将int类型的数据写入基础的字节输出流
public final void writeDouble(Double v) throws IOException	将double类型的数据写入基础的字节输出流
public final void writeUTF(String str) throws IOException	将字符串数据以UTF-8编码成字节写入基础的字节输出流
public void write(int/byte[]/byte[]一部分)	支持写字节数组出去

```

public class DataOutputStreamTest1 {
    public static void main(String[] args) {
        try (
            // 1、创建一个数据输出流包装低级的字节输出流
            DataOutputStream dos =
                new DataOutputStream(new
                FileOutputStream("io-app2/src/itheima10out.txt"));
        ) {
    }
}

```

```

        dos.writeInt(97);
        dos.writeDouble(99.5);
        dos.writeBoolean(true);
        dos.writeUTF("黑马程序员666! ");

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

## 2. DataInputStream

读取数据输出流写出的数据。

构造器	说明
public DataInputStream(InputStream is)	创建新数据输入流包装基础的字节输入流
方法	说明
Public final byte readByte() throws IOException	读取字节数据返回
public final int readInt() throws IOException	读取int类型的数据返回
public final double readDouble() throws IOException	读取double类型的数据返回
public final String readUTF() throws IOException	读取字符串数 (UTF-8) 据返回
public int readInt()/read(byte[])	支持读字节数据进来

```

public class DataInputStreamTest2 {
    public static void main(String[] args) {
        try {
            DataInputStream dis =
                new DataInputStream(new
FileInputStream("io-app2/src/itheima10out.txt"));
        }{
            int i = dis.readInt(); //dataoutputstream先写什么，就得先读什么
            System.out.println(i);

            double d = dis.readDouble();
            System.out.println(d);

            boolean b = dis.readBoolean();
            System.out.println(b);

            String rs = dis.readUTF();
            System.out.println(rs);
        } catch (Exception e) {

```

```
        e.printStackTrace();
    }
}
}
```

## 序列化流

### 1. 概念

序列化：意思就是把对象写到文件或者网络中去。（简单记：写对象）

反序列化：意思就是把对象从文件或者网络中读取出来。（简单记：读对象）

### 2. ObjectOutputStream

可以把Java对象存入到文件中去。

构造器	说明
<code>public ObjectOutputStream(OutputStream out)</code>	创建对象字节输出流，包装基础的字节输出流
方法	说明
<code>public final void writeObject(Object o) throws IOException</code>	把对象写出去

要将对象序列化，必须让Java类实现Serializable接口。

加transient关键字的成员变量将不参与序列化。

```
// 注意：对象如果需要序列化，必须实现序列化接口。
public class User implements Serializable {
    private String loginName;
    private String userName;
    private int age;
    // transient 这个成员变量将不参与序列化。
    private transient String password;

    public User() {
    }

    public User(String loginName, String userName, int age,
String password) {
        this.loginName = loginName;
        this.userName = userName;
        this.age = age;
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
    }}
```

```

        "loginName'" + loginName + '\'' +
        ", userName'" + userName + '\'' +
        ", age=" + age +
        ", password'" + password + '\'' +
        '}';
    }

public class Test1ObjectOutputStream {
    public static void main(String[] args) {
        try {
            // 2、创建一个对象字节输出流包装原始的字节 输出流。
            ObjectOutputStream oos =
                new ObjectOutputStream(new
FileOutputStream("io-app2/src/itheima1out.txt"));
            ){
                // 1、创建一个Java对象。
                User u = new User("admin", "张三", 32,
"666888xyz");

                // 3、序列化对象到文件中去
                oos.writeObject(u);
                System.out.println("序列化对象成功！！");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

### 3. ObjectInputStream

构造器	说明
public ObjectInputStream(InputStream is)	创建对象字节输入流，包装基础的字节输入流

方法	说明
public final Object readObject()	把存储在文件中的Java对象读出来

## IO框架

### 1. 框架

定义：解决某类问题，编写的一套类、接口等。

形式：一般是把类、接口等编译成class形式，再压缩成一个.jar结尾的文件发出去。

IO框架：封装了Java提供的对文件、数据进行操作的代码。

## 2. Commons-io

由apache开源基金组织提供了一组有关IO流小框架，可以提高IO流的开发效率。

FileUtils类提供的部分方法展示		说明
public static void	copyFile(File srcFile, File destFile)	复制文件。
public static void	copyDirectory(File srcDir, File destDir)	复制文件夹
public static void	deleteDirectory(File directory)	删除文件夹
public static String	readFileToString(File file, String encoding)	读数据
public static void	writeStringToFile(File file, String data, String charname, boolean append)	写数据

IOUtils类提供的部分方法展示		说明
public static int	copy(InputStream inputStream, OutputStream outputStream)	复制文件。
public static int	copy(Reader reader, Writer writer)	复制文件。
public static void	write(String data, OutputStream output, String charsetName)	写数据

```
public class CommonsIOTest1 {
    public static void main(String[] args) throws Exception {
        //1. 复制文件
        FileUtils.copyFile(new File("io-
app2\\src\\itheima01.txt"), new File("io-app2/src/a.txt"));

        //2. 复制文件夹
        FileUtils.copyDirectory(new File("D:\\resource\\私人珍
藏"), new File("D:\\resource\\私人珍藏3"));

        //3. 删除文件夹
        FileUtils.deleteDirectory(new File("D:\\resource\\私人
珍藏3"));

        // Java提供的原生的一行代码搞定很多事情
        Files.copy(Path.of("io-app2\\src\\itheima01.txt"),
Path.of("io-app2\\src\\b.txt"));
        System.out.println(Files.readString(Path.of("io-
app2\\src\\itheima01.txt")));
    }
}
```

# 特殊文件与日志

## 属性文件

### 1. 概述

- (1) 后缀为.properties的文件，称之为属性文件，它可以很方便的存储一些类似于键值对的数据。经常当做软件的配置文件使用。
- (2) 而xml文件能够表示更加复杂的数据关系，比如要表示多个用户的用户名、密码、家乡、性别等。在后面，也经常当做软件的配置文件使用。

### 2. Properties属性文件

文件格式：

- (1) 属性文件后缀以 .properties 结尾
- (2) 属性文件里面的每一行都是一个键值对，键和值中间用=隔开。比如:  
admin=123456
- (3) # 表示这样是注释信息，是用来解释这一行配置是什么意思。
- (4) 每一行末尾不要习惯性加分号，以及空格等字符；不然会把分号，空格会当做值的一部分。
- (5) 键不能重复，值可以重复



Properties：是Map接口下面的一个实现类，所以Properties也是一种双列集合，用来存储键值对。Properties类的对象，用来表示属性文件，可以用来读取属性文件中的键值对。使用Properties读取属性文件中的键值对，需要用到的方法如下

### 使用Properties读取属性文件里的键值对数据

构造器	说明
public Properties()	用于构建Properties集合对象（空容器）
常用方法	说明
public void load(InputStream is)	通过字节输入流，读取属性文件里的键值对数据
public void load(Reader reader)	通过字符输入流，读取属性文件里的键值对数据
public String getProperty(String key)	根据键获取值(其实就是get方法的效果)
public Set<String> stringPropertyNames()	获取全部键的集合 (其实就是keySet方法的效果)

## 使用Properties读取属性文件：

- 1、创建一个Properties的对象出来（键值对集合，空容器）
- 2、调用load(字符输入流/字节输入流)方法，开始加载属性文件中的键值对数据到properties对象中去
- 3、调用getProperty(键)方法，根据键取值

```
/**  
 * 目标：掌握使用Properties类读取属性文件中的键值对信息。  
 */  
public class PropertiesTest1 {  
    public static void main(String[] args) throws Exception {  
        // 1、创建一个Properties的对象出来（键值对集合，空容器）  
        Properties properties = new Properties();  
        System.out.println(properties);  
  
        // 2、开始加载属性文件中的键值对数据到properties对象中去  
        properties.load(new FileReader("properties-xml-log-  
app\\src\\users.properties"));  
        System.out.println(properties);  
  
        // 3、根据键取值  
        System.out.println(properties.getProperty("赵敏"));  
        System.out.println(properties.getProperty("张无忌"));  
  
        // 4、遍历全部的键和值  
        //获取键的集合  
        Set<String> keys = properties.stringPropertyNames();  
        for (String key : keys) {  
            //再根据键获取值  
            String value = properties.getProperty(key);  
            System.out.println(key + "---->" + value);  
        }  
  
        properties.forEach((k, v) -> {  
            System.out.println(k + "---->" + v);  
        });  
    }  
}
```

使用Properties往属性文件中写键值对，需要用到的方法如下

## 使用Properties把键值对数据写出到属性文件里去

构造器	说明
public Properties()	用于构建Properties集合对象（空容器）
常用方法	说明
public ObjectsetProperty(String key, String value)	保存键值对数据到Properties对象中去。
public void store(OutputStream os, String comments)	把键值对数据，通过字节输出流写出到属性文件里去
public void store(Writer w, String comments)	把键值对数据，通过字符输出流写出到属性文件里去

往Properties属性文件中写键值对：

- 1、先准备一个.properties属性文件，按照格式写几个键值对
- 2、创建Properties对象出来，
- 3、调用setProperty存储一些键值对数据
- 4、调用store(字符输出流/字节输出流，注释)，将Properties集合中的键和值写到文件中

注意：第二个参数是注释，必须得加；

```
public class PropertiesTest2 {  
    public static void main(String[] args) throws Exception {  
        // 1、创建Properties对象出来，先用它存储一些键值对数据  
        Properties properties = new Properties();  
        properties.setProperty("张无忌", "minmin");  
        properties.setProperty("殷素素", "cuishan");  
        properties.setProperty("张翠山", "susu");  
  
        // 2、把properties对象中的键值对数据存入到属性文件中去  
        properties.store(new FileWriter("properties-xml-log-  
app/src/users2.properties")  
                , "i saved many users!");  
  
    }  
}
```

## XML文件

1. XML是可扩展的标记语言，意思是它是由一些标签组成的，而这些标签是自己定义的。本质上一种数据格式，可以用来表示复杂的数据关系。

XML文件有如下的特点：

- XML中的<标签名>称为一个标签或者一个元素，一般是成对出现的。
- XML中的标签名可以自己定义（可扩展），但是必须要正确的嵌套
- XML中只能有一个根标签。

- XML标准中可以有属性
- XML必须第一行有一个文档声明，格式是固定的 `<?xml version="1.0" encoding="UTF-8"?>`
- XML文件必须是以.xml为后缀结尾

## 2. 特殊字符

有一些特殊字符不能直接写。

```
&lt; 表示 <  
&gt; 表示 >  
&amp; 表示 &  
&apos; 表示 '  
&quot; 表示 "
```

如果在标签文本中，出现大量的特殊字符，不想使用特殊字符，此时可以用 CDATA区，格式如下

```
<data1>  
  <! [CDATA[  
    3 < 2 && 5 > 4  
  ]]>  
</data1>
```

## 3. 应用场景

作为系统的配置文件；作为一种特殊的数据结构，在网络中进行传输。

## 4. 解析XML文件

使用程序读取XML文件中的数据称为解析XML文件。

# 日志技术

## 1. 概述

定义：程序中的日志，通常就是一个文件，里面记录了程序运行过程中产生的各种数据。

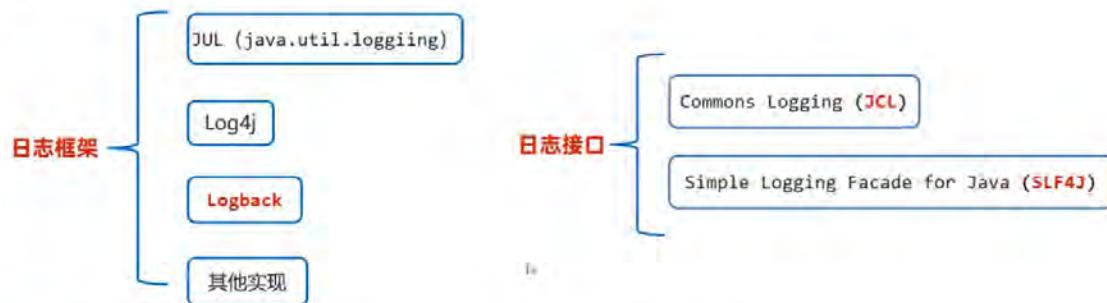
日志技术有如下好处

(1) 日志可以将系统执行的信息，方便的记录到指定位置，可以是控制台、可以是文件、可以是数据库中。

(2) 日志可以随时以开关的形式控制启停，无需侵入到源代码中去修改。

## 2. 日志技术的体系结构

## 日志技术的体系结构



- **日志框架**: 牛人或者第三方公司已经做好的实现代码，后来者直接可以拿去使用。
- **日志接口**: 设计日志框架的一套标准，日志框架需要实现这些接口。

目前一般使用Logback+SLF4J

Logback日志框架有以下几个模块：



要想使用Logback日志框架，至少需要在项目中整合三个模块：slf4j-api+logback-core+logback-classic

### 3. Logback

创建日志记录对象

```
public static final Logger LOGGER = LoggerFactory.getLogger("当前类名");
```

记录日志

```
public class LogBackTest {  
    // 创建一个Logger日志对象  
    public static final Logger LOGGER =  
        LoggerFactory.getLogger("LogBackTest");  
  
    public static void main(String[] args) {  
        //while (true) {  
        try {  
            LOGGER.info("chu法方法开始执行~~~");  
            chu(10, 0);  
            LOGGER.info("chu法方法执行成功~~~");  
        } catch (Exception e) {  
            LOGGER.error("chu法方法执行失败了，出现了bug~~~");  
        }  
    }  
}
```

```

        }
    //}

}

public static void chu(int a, int b){
    LOGGER.debug("参数a:" + a);
    LOGGER.debug("参数b:" + b);
    int c = a / b;
    LOGGER.info("结果是: " + c);
}

```

#### 4. 日志配置文件

(1) 配置日志输出的位置是文件、还是控制台

- 通常可以设置2个输出日志的位置：一个是控制台、一个是系统文件中

```
<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
```

```
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
```

(2) 开启/关闭日志

**开启日志 (ALL) , 取消日志(OFF)**

```
<root level="ALL">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE" />
</root>
```

(3) 选择是否要输出到文件或者控制台

在 (2) 中选择不要输出的进行删去

#### 5. Logback设置日志级别

- 日志级别指的是日志信息的类型，日志都会分级别，常见的日志级别如下（优先级依次升高）：

日志级别	说明
trace	追踪，指明程序运行轨迹
debug	调试，实际应用中一般将其作为最低级别，而 trace 则很少使用
info	输出重要的运行信息，数据连接、网络连接、IO操作等等，使用较多
warn	警告信息，可能会发生问题，使用较多
error	错误信息，使用较多

配置日志级别

```
<root level='info'>
</root>
```

Logback只输出大于或者等于核心配置文件配置的日志级别信息。小于配置级别的日志信息，不被记录。

# Java 高级

## 多线程

### 概述与线程创建

#### 1. 定义

线程：一个程序内部的一条执行流程。

多线程：从软硬件上实现的多条执行流程的技术（多条线程由CPU负责调度执行）

#### 2. 多线程创建

通过java.lang.Thread类的对象来代表线程。

##### (1) 创建方式1：继承Thread类

1. 定义一个子类继承**Thread**类，并重写**run**方法
2. 创建**Thread**的子类对象
3. 调用**start**方法启动线程（启动线程后，会自动执行**run**方法中的代码）

代码如下

```
public class MyThread extends Thread{  
    // 2、必须重写Thread类的run方法  
    @Override  
    public void run() {  
        // 描述线程的执行任务。  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("子线程MyThread输出：" + i);  
        }  
    }  
}
```

再定义一个测试类，在测试类中创建MyThread线程对象，并启动线程

```

public class ThreadTest1 {
    // main方法是由一条默认的主线程负责执行。
    public static void main(String[] args) {
        // 3、创建MyThread线程类的对象代表一个线程
        Thread t = new MyThread();
        // 4、启动线程（自动执行run方法的）
        t.start();

        for (int i = 1; i <= 5; i++) {
            System.out.println("主线程main输出: " + i);
        }
    }
}

```

缺点：线程类已经继承Thread，无法继承其他类，不利于功能扩展。

## (2) 创建方式2：实现Runnable接口

1. 先写一个Runnable接口的实现类，重写run方法（这里面就是线程要执行的代码）
2. 再创建一个Runnable实现类的对象
3. 创建一个Thread对象，把Runnable实现类的对象传递给Thread
4. 调用Thread对象的start()方法启动线程（启动后会自动执行Runnable里面的run方法）

代码如下：先准备一个Runnable接口的实现类

```

/**
 * 1、定义一个任务类，实现Runnable接口
 */
public class MyRunnable implements Runnable{
    // 2、重写runnable的run方法
    @Override
    public void run() {
        // 线程要执行的任务。
        for (int i = 1; i <= 5; i++) {
            System.out.println("子线程输出 ==> " + i);
        }
    }
}

```

再写一个测试类，在测试类中创建线程对象，并执行线程

```
public class ThreadTest2 {  
    public static void main(String[] args) {  
        // 3、创建任务对象。  
        Runnable target = new MyRunnable();  
        // 4、把任务对象交给一个线程对象处理。  
        // public Thread(Runnable target)  
        new Thread(target).start();  
  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("主线程main输出 ===》 " + i);  
        }  
    }  
}
```

运行上面代码，结果如下图所示（注意：没有出现下面交替执行的效果，也是正常的）

```
主线程main输出 ===》 1  
主线程main输出 ===》 2  
主线程main输出 ===》 3  
子线程输出 ===》 1  
子线程输出 ===》 2  
子线程输出 ===》 3  
子线程输出 ===》 4  
子线程输出 ===》 5  
主线程main输出 ===》 4  
主线程main输出 ===》 5
```

匿名方式类改写实现类：直接创建Runnable接口的匿名内部类对象，传递给Thread对象。

```
public class ThreadTest2_2 {  
    public static void main(String[] args) {  
        // 1、直接创建Runnable接口的匿名内部类形式（任务对象）  
        Runnable target = new Runnable() {  
            @Override  
            public void run() {  
                for (int i = 1; i <= 5; i++) {  
                    System.out.println("子线程1输出: " + i);  
                }  
            }  
        };  
        new Thread(target).start();  
    }  
}
```

```

// 简化形式1:
new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("子线程2输出: " + i);
        }
    }
}).start();

// 简化形式2:
new Thread(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("子线程3输出: " + i);
    }
}).start();

for (int i = 1; i <= 5; i++) {
    System.out.println("主线程main输出: " + i);
}
}
}

```

### (3) 创建方式3：实现Callable接口

前两种方法存在的问题：假设线程执行完毕之后有一些数据需要返回，前面两种方式重写的run方法均没有返回结果。

JDK5提供了Callable接口和FutureTask类来创建线程，它最大的优点就是有返回值。

在Callable接口中有一个call方法，重写call方法就是线程要执行的代码，它是有返回值的

```

public T call(){
    ...线程执行的代码...
    return 结果;
}

```

1. 先定义一个**Callable**接口的实现类，重写**call**方法
2. 创建**Callable**实现类的对象
3. 创建**FutureTask**类的对象，将**Callable**对象传递给**FutureTask**
4. 创建**Thread**对象，将**FutureTask**对象传递给**Thread**
5. 调用**Thread**的**start()**方法启动线程(启动后会自动执行**call**方法)  
等**call()**方法执行完之后，会自动将返回值结果封装到**FutureTask**对象中
6. 调用**FutureTask**对的**get()**方法获取返回结果

代码如下：先准备一个**Callable**接口的实现类

```
/*
 * 1、让子类实现Callable接口。
 */
public class MyCallable implements Callable<String>{
    private int n;
    public MyCallable(int n) {
        this.n = n;
    }

    // 2、必须重写Callable接口的call方法
    @Override
    public String call() throws Exception {
        // 描述线程的执行任务。
        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += i;
        }
        return "线程求出的和为" + sum;
    }
}
```

再定义一个测试类，在测试类中创建线程并启动线程，还要获取返回结果

```
public class ThreadTest3 {
    public static void main(String[] args) throws Exception {
        // 3、创建一个Callable的对象
        Callable<String> call = new MyCallable(100);
        // 4、把Callable的对象封装成一个FutureTask对象（任务对象）
        // 未来任务对象的作用？
        // 1、是一个任务对象，实现了Runnable对象。
        // 2、可以在线程执行完毕之后，用未来任务对象调用get方法获取线程执行完毕后的结果。
        FutureTask<String> f1 = new FutureTask<>(call);
        // 5、把任务对象交给一个Thread对象
    }
}
```

```
new Thread(f1).start();  
  
Callable<String> call1 = new MyCallable(200);  
FutureTask<String> f1 = new FutureTask<>(call1);  
new Thread(f1).start();  
  
  
// 6、获取线程执行完毕后返回的结果。  
// 注意：如果执行到这儿，假如上面的线程还没有执行完毕  
// 这里的代码会暂停，等待上面线程执行完毕后才会获取结果。  
String rs = f1.get();  
System.out.println(rs);  
  
String rs2 = f2.get();  
System.out.println(rs2);  
}  
}
```

优点：可以在线程执行完毕后去获取线程执行的结果。

缺点：编码较为复杂。

### 3. 多线程的注意事项

(1) 启动线程必须是调用start方法，不是调用run方法。若调用run只是把run当成普通方法执行，不会形成多线程。

(2) 不要把主线程任务放在启动子线程之前。因为这样永远会把主线程任务先跑完。

# Thread的常用方法

Thread提供了很多与线程操作相关的方法

Thread提供的常用方法	说明
public void run()	线程的任务方法
public void start()	启动线程
public String getName()	获取当前线程的名称，线程名称默认是Thread-索引
public void setName(String name)	为线程设置名称
public static Thread currentThread()	获取当前执行的线程对象
public static void sleep(long time)	让当前执行的线程休眠多少毫秒后，再继续执行
public final void join()...	让调用当前这个方法的线程先执行完！

Thread提供的常见构造器	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	封装Runnable对象成为线程对象
public Thread(Runnable target, String name)	封装Runnable对象成为线程对象，并指定线程名称

```
public class MyThread extends Thread{
    public MyThread(String name){
        super(name); //1.执行父类Thread(String name)构造器，为当前线程
设置名字了
    }
    @Override
    public void run() {
        //2.currentThread() 哪个线程执行它，它就会得到哪个线程对象。
        Thread t = Thread.currentThread();
        for (int i = 1; i <= 3; i++) {
            //3.getName() 获取线程名称
            System.out.println(t.getName() + "输出：" + i);
        }
    }
}
```

```
public class ThreadTest2 {
    public static void main(String[] args) throws Exception {
        // join方法作用：让当前调用这个方法的线程先执行完。
        Thread t1 = new MyThread("1号线程");
        t1.start();
        t1.join();

        Thread t2 = new MyThread("2号线程");
        t2.start();
        t2.join();
    }
}
```

```
    Thread t3 = new MyThread("3号线程");
    t3.start();
    t3.join();
}
}

//最后的执行顺序严格为t1 t2 t3
```

## 线程安全与线程同步

1. 线程安全定义：线程安全问题指的是，多个线程同时操作同一个共享资源的时候，可能会出现业务安全问题。
2. 线程同步：解决线程安全问题的方案

思想：让多个线程实现先后依次访问共享资源。

线程同步一般有如下三种方法：

(1) 同步代码块

作用：把访问共享资源的核心代码给上锁，以此保证线程安全。

原理：每次只允许一个线程加锁后进入，执行完毕后自动解锁，其他线程才可以进来执行。

```
//锁对象：必须是一个唯一的对象（同一个地址）
synchronized(锁对象){
    //...访问共享数据的代码...
}
```

```
// 小明 小红线程同时过来的
public void drawMoney(double money) {
    // 先搞清楚是谁来取钱？
    String name = Thread.currentThread().getName();
    // 1、判断余额是否足够
    // this正好代表共享资源！
    synchronized (this) {
        if(this.money >= money){
            System.out.println(name + "来取钱" + money + "成功！");
            this.money -= money;
            System.out.println(name + "来取钱后，余额剩余：" + this.money);
        }else {
            System.out.println(name + "来取钱：余额不足~");
        }
    }
}
```

```
}
```

对于同步代码块的锁对象，选择方法一般为

1. 建议把共享资源作为锁对象，不要将随便无关的对象当做锁对象
2. 对于实例方法，建议使用**this**作为锁对象
3. 对于静态方法，建议把类的字节码(类名.**class**)当做锁对象

## (2) 同步方法

作用：把访问共享资源的核心方法给上锁，以此保证线程安全。

```
修饰符 synchronized 返回值类型 方法名称(形参列表) {  
    操作共享资源的代码  
}
```

```
// 同步方法  
public synchronized void drawMoney(double money) {  
    // 先搞清楚是谁来取钱?  
    String name = Thread.currentThread().getName();  
    // 1、判断余额是否足够  
    if(this.money >= money){  
        System.out.println(name + "来取钱" + money + "成功!");  
        this.money -= money;  
        System.out.println(name + "来取钱后，余额剩余：" +  
this.money);  
    }else {  
        System.out.println(name + "来取钱：余额不足~");  
    }  
}
```

## 同步方法底层原理

1. 底层有隐式锁对象，只是锁的范围是整个方法代码
2. 如果方法是实例方法，同步方法默认用**this**作为锁对象
3. 如果方法是静态方法，同步方法默认用类名.**class**作为锁对象

## (3) Lock锁

构造器	说明
public ReentrantLock()	获得Lock锁的实现类对象

## Lock的常用方法

方法名称	说明
void lock()	获得锁
void unlock()	释放锁

```
// 创建了一个锁对象
private final Lock lk = new ReentrantLock();

public void drawMoney(double money) {
    // 先搞清楚是谁来取钱?
    String name = Thread.currentThread().getName();
    try {
        lk.lock(); // 加锁
        // 1、判断余额是否足够
        if(this.money >= money){
            System.out.println(name + "来取钱" + money + "成功!");
            this.money -= money;
            System.out.println(name + "来取钱后，余额剩余：" +
this.money);
        }else {
            System.out.println(name + "来取钱：余额不足~");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lk.unlock(); // 解锁
    }
}
```

## 线程通信

### 1. 定义

当多个线程共同操作共享资源时，线程间通过某种方式互相告知自己的状态，以相互协调，并避免无效的资源争夺。

### 2. 等待与唤醒方法

## Object类的等待和唤醒方法：

方法名称	说明
void wait()	让当前线程等待并释放所占锁，直到另一个线程调用notify()方法或 notifyAll()方法
void notify()	唤醒正在等待的单个线程
void notifyAll()	唤醒正在等待的所有线程

注意：上述方法应该使用当前同步锁对象进行调用，如同步锁对象是this就该用this.wait()

### 3. 生产者与消费者模型

```
public class Desk {  
    private List<String> list = new ArrayList<>();  
  
    // 放1个包子的方法  
    // 厨师1 厨师2 厨师3  
    public synchronized void put() {  
        try {  
            String name = Thread.currentThread().getName();  
            // 判断是否有包子。  
            if(list.size() == 0){  
                list.add(name + "做的肉包子");  
                System.out.println(name + "做了一个肉包子~~");  
                Thread.sleep(2000);  
  
                // 唤醒别人，等待自己  
                this.notifyAll();  
                this.wait();  
            }else {  
                // 有包子了，不做了。  
                // 唤醒别人，等待自己  
                this.notifyAll();  
                this.wait();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    // 吃货1 吃货2  
    public synchronized void get() {  
        try {  
            String name = Thread.currentThread().getName();  
            if(list.size() == 1){  
                list.remove(0);  
                System.out.println(name + "吃了包子");  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

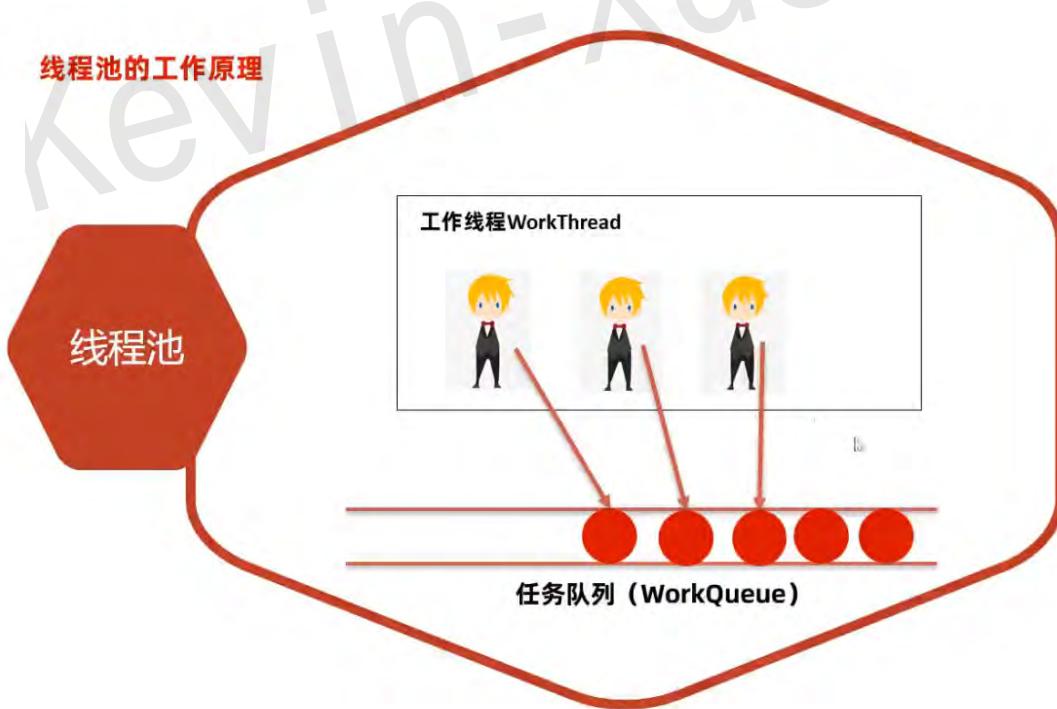
        // 有包子，吃了
        System.out.println(name + "吃了：" + 
list.get(0));
        list.clear();
        Thread.sleep(1000);
        this.notifyAll();
        this.wait();
    } else {
        // 没有包子
        this.notifyAll();
        this.wait();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 线程池

### 1. 概述

定义：线程池就是一个线程复用技术，它可以提高线程的利用率。



线程池中的线程称为工作线程，要执行的任务放入任务队列中（只有实现了 Runnable或者Callable接口的任务才能放入任务队列）。线程池可以设置工作线程和任务队列中任务的数量。

### 2. 线程池的创建

在JDK5版本中提供了代表线程池的接口ExecutorService，而这个接口下有一个实现类叫ThreadPoolExecutor类，使用ThreadPoolExecutor类就可以用来创建线程池对象。

实现类的构造器如下

#### ThreadPoolExecutor构造器

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
    BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

- 参数一：corePoolSize：指定线程池的核心线程的数量。  
除了核心线程，还有临时线程
- 参数二：maximumPoolSize：指定线程池的最大线程数量。
- 参数三：keepAliveTime：指定临时线程的存活时间。过了存活时间，临时线程会kill
- 参数四：unit：指定临时线程存活的时间单位(秒、分、时、天)
- 参数五：workQueue：指定线程池的任务队列。
- 参数六：threadFactory：指定线程池的线程工厂。用来创建线程
- 参数七：handler：指定线程池的任务拒绝策略（线程都在忙，任务队列也满了的时候，新任务来了该怎么处理）

#### ThreadPoolExecutor构造器

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
    BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

- 参数一：corePoolSize：指定线程池的核心线程的数量。  
正式工：3
- 参数二：maximumPoolSize：指定线程池的最大线程数量。  
最大员工数：5  
临时工：2
- 参数三：keepAliveTime：指定临时线程的存活时间。  
临时工空闲多久被开除
- 参数四：unit：指定临时线程存活的时间单位(秒、分、时、天)
- 参数五：workQueue：指定线程池的任务队列。  
客人排队的地方
- 参数六：threadFactory：指定线程池的线程工厂。  
负责招聘员工的 (hr)
- 参数七：handler：指定线程池的任务拒绝策略（线程都在忙，任务队列也满了的时候，新任务来了该怎么处理）  
忙不过来咋办？

```
ExecutorService pool = new ThreadPoolExecutor(
    3, //核心线程数有3个
    5, //最大线程数有5个。    临时线程数=最大线程数-核心线程数=5-3=2
    8, //临时线程存活的时间8秒。 意思是临时线程8秒没有任务执行，就会被销毁掉。
    TimeUnit.SECONDS, //时间单位（秒）
    new ArrayBlockingQueue<>(4), //任务阻塞队列，没有来得及执行的任务在，任务队列中等待，idea会补全
    Executors.defaultThreadFactory(), //用于创建线程的工厂对象，工具类Executors的方法
    new ThreadPoolExecutor.CallerRunsPolicy() //拒绝策略，记住即可
);
```

新任务拒绝策略

## 新任务拒绝策略

策略	详解
<code>ThreadPoolExecutor.AbortPolicy</code>	丢弃任务并抛出 <code>RejectedExecutionException</code> 异常。 <b>是默认的策略</b>
<code>ThreadPoolExecutor.DiscardPolicy</code> :	丢弃任务，但是不抛出异常 这是不推荐的做法
<code>ThreadPoolExecutor.DiscardOldestPolicy</code>	抛弃队列中等待最久的任务 然后把当前任务加入队列中
<code>ThreadPoolExecutor.CallerRunsPolicy</code>	由主线程负责调用任务的 <code>run()</code> 方法从而绕过线程池直接执行

### 注意

(1) 临时线程何时创建?

新任务提交时，发现核心线程都在忙、任务队列满了、并且还可以创建临时线程，此时会创建临时线程。

(2) 什么时候开始拒绝新的任务?

核心线程和临时线程都在忙、任务队列也满了、新任务过来时才会开始拒绝任务。

### 3. 线程池处理Runnable任务

#### ExecutorService的常用方法

方法名称	说明
<code>void execute(Runnable command)</code>	执行 Runnable 任务
<code>Future&lt;T&gt; submit(Callable&lt;T&gt; task)</code>	执行 Callable 任务，返回未来任务对象，用于获取线程返回的结果
<code>void shutdown()</code>	等全部任务执行完毕后，再关闭线程池！
<code>List&lt;Runnable&gt; shutdownNow()</code>	立刻关闭线程池，停止正在执行的任务，并返回队列中未执行的任务

```
ExecutorService pool = new ThreadPoolExecutor(
    3, //核心线程数有3个
    5, //最大线程数有5个。 临时线程数=最大线程数-核心线程数=5-3=2
    8, //临时线程存活的时间8秒。 意思是临时线程8秒没有任务执行，就会被销毁掉。
    TimeUnit.SECONDS, //时间单位（秒）
    new ArrayBlockingQueue<>(4), //任务阻塞队列，没有来得及执行的任务在，任务队列中等待
    Executors.defaultThreadFactory(), //用于创建线程的工厂对象
    new ThreadPoolExecutor.CallerRunsPolicy() //拒绝策略
);

Runnable target = new MyRunnable();
pool.execute(target); // 线程池会自动创建一个新线程，自动处理这个任务，自动执行的！
```

```

pool.execute(target); // 线程池会自动创建一个新线程，自动处理这个任务，  

                     自动执行的！  

pool.execute(target); // 线程池会自动创建一个新线程，自动处理这个任务，  

                     自动执行的！  

//下面4个任务在任务队列里排队  

pool.execute(target);  

pool.execute(target);  

pool.execute(target);  

pool.execute(target);  
  

//下面2个任务，到了临时线程的创建时机了  

pool.execute(target);  

pool.execute(target);  

// 到了新任务的拒绝时机了！  

pool.execute(target);

```

pool.shutdown(); //等线程池任务全部完成后，再关闭线程池  
pool.shutdownNow(); //立刻关闭线程池

#### 4. 线程池处理Callable任务

方法名称	说明
void execute(Runnable command)	执行任务/命令，没有返回值，一般用来执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行任务，返回未来任务对象获取线程结果，一般拿来执行 Callable 任务
void shutdown()	等任务执行完毕后关闭线程池
List<Runnable> shutdownNow()	立刻关闭，停止正在执行的任务，并返回队列中未执行的任务

先准备一个Callable线程任务

```

public class MyCallable implements Callable<String> {  

    private int n;  

    public MyCallable(int n) {  

        this.n = n;  

    }  
  

    // 2、重写call方法  

    @Override  

    public String call() throws Exception {  

        // 描述线程的任务，返回线程执行返回后的结果。  

        // 需求：求1-n的和返回。  

        int sum = 0;  

        for (int i = 1; i <= n; i++) {  

            sum += i;  

        }
    }
}

```

```
        return Thread.currentThread().getName() + "求出了1-" +  
n + "的和是: " + sum;  
    }  
}
```

再准备一个测试类，在测试类中创建线程池，并执行callable任务。

```
public class ThreadPoolTest2 {  
    public static void main(String[] args) throws Exception {  
        // 1、通过ThreadPoolExecutor创建一个线程池对象。  
        ExecutorService pool = new ThreadPoolExecutor(  
            3,  
            5,  
            8,  
            TimeUnit.SECONDS,  
            new ArrayBlockingQueue<>(4),  
            Executors.defaultThreadFactory(),  
            new ThreadPoolExecutor.CallerRunsPolicy());  
  
        // 2、使用线程处理Callable任务。  
        Future<String> f1 = pool.submit(new MyCallable(100));  
        Future<String> f2 = pool.submit(new MyCallable(200));  
        Future<String> f3 = pool.submit(new MyCallable(300));  
        Future<String> f4 = pool.submit(new MyCallable(400));  
  
        // 3、执行完Callable任务后，需要获取返回结果。  
        System.out.println(f1.get());  
        System.out.println(f2.get());  
        System.out.println(f3.get());  
        System.out.println(f4.get());  
    }  
}
```

## 5. Executors工具类实现线程池

Executors是一个线程池的工具类，提供了很多静态方法用于返回不同特点的线程池对象。这些方法的底层都是通过线程池的实现类ThreadPoolExecutor创建的线程池对象。

注意：大型并发系统环境中使用Executors可能会出现系统风险。

方法名称	说明
<code>public static ExecutorService newFixedThreadPool(int nThreads)</code>	创建固定线程数量的线程池。如果某个线程因为执行异常而结束，那么线程池会补充一个新线程替代它。
<code>public static ExecutorService newSingleThreadExecutor()</code>	创建只有一个线程的线程池对象。如果该线程出现异常而结束，那么线程池会补充一个新线程。
<code>public static ExecutorService newCachedThreadPool()</code>	线程数量随着任务增加而增加。如果线程任务执行完毕且空闲了60s则会被回收掉。
<code>public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)</code>	创建一个线程池，可以实现在给定的延迟后运行任务，或者定期执行任务。

```

public class ThreadPoolTest3 {
    public static void main(String[] args) throws Exception {
        // 1、通过Executors创建一个线程池对象。
        ExecutorService pool =
        Executors.newFixedThreadPool(17);
        // 老师：核心线程数量到底配置多少呢？？
        // 计算密集型的任务：核心线程数量 = CPU的核数 + 1
        // IO密集型的任务：核心线程数量 = CPU核数 * 2

        // 2、使用线程处理callable任务。
        Future<String> f1 = pool.submit(new MyCallable(100));
        Future<String> f2 = pool.submit(new MyCallable(200));
        Future<String> f3 = pool.submit(new MyCallable(300));
        Future<String> f4 = pool.submit(new MyCallable(400));

        System.out.println(f1.get());
        System.out.println(f2.get());
        System.out.println(f3.get());
        System.out.println(f4.get());
    }
}

```

## 并发并行&生命周期

### 1. 进程与线程

正常运行的程序（软件）就是一个独立的进程。

线程是属于进程，一个进程中包含多个线程，进程中的线程其实并发和并行同时存在。

### 2. 并发与并行

并发：进程中的线程由CPU负责调度执行，但是CPU同时处理线程的数量是优先的，为了保证全部线程都能执行到，CPU采用轮询机制为系统的每个线程服务，由于CPU切换的速度很快，给我们的感觉这些线程在同时执行，这就是并发。

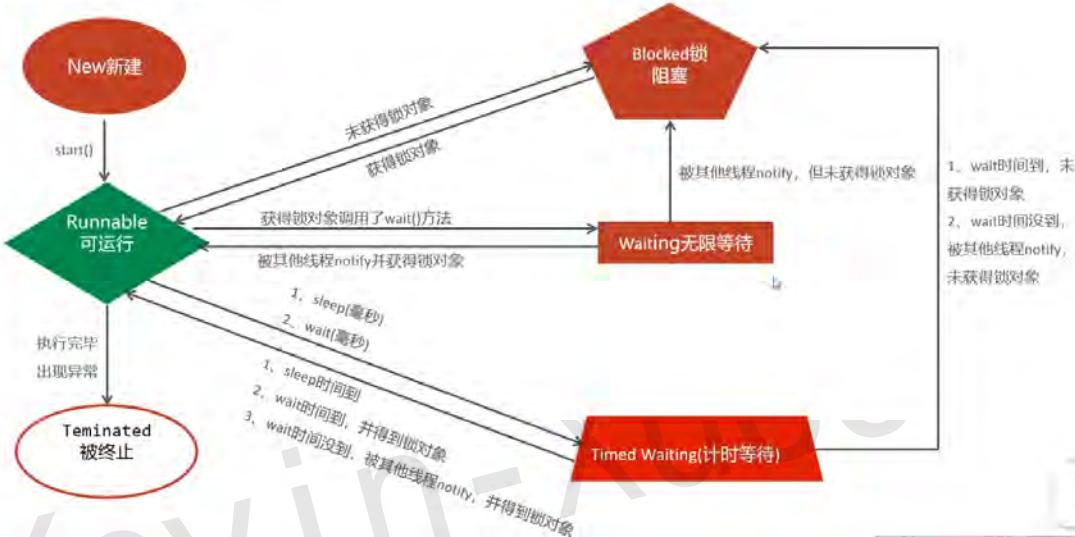
并行：并行指的是，多个线程同时被CPU调度执行。

### 3. 线程的生命周期

Java总共定义了6种状态，6种状态都定义在Thread类的内部枚举类中。

```
public class Thread{  
    ...  
    public enum State {  
        NEW,  
        RUNNABLE,  
        BLOCKED,  
        WAITING,  
        TIMED_WAITING,  
        TERMINATED;  
    }  
    ...  
}
```

线程的6种状态互相转换



线程状态	说明
NEW(新建)	线程刚被创建，但是并未启动。
Runnable(可运行)	线程已经调用了start(), 等待CPU调度
Blocked(锁阻塞)	线程在执行的时候未竞争到锁对象，则该线程进入Blocked状态；
Waiting(无限等待)	一个线程进入Waiting状态，另一个线程调用notify或者notifyAll方法才能够唤醒
Timed Waiting(计时等待)	同Waiting状态，有几个方法(sleep,wait)有超时参数，调用他们将进入Timed Waiting状态。
Terminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

## 悲观锁与乐观锁

1. 悲观锁：一上来就加锁，每次只能一个线程进入访问完毕后，再解锁。

之前讲解的都是悲观锁，线程安全，但性能较差，因为整个进程中大量线程阻塞和等待。

2. 乐观锁：一开始不上锁，认为是没有问题的，等要出现线程安全问题的时候才开始控制。线程安全，且性能较好。

乐观锁基于CAS算法：假设线程操作对整数10加1，线程A取出10后，进行加1，然后检查原地址中的数是否还是10。若还是10，说明没有其他线程修改，将加1后的结果存入原地址；若不是10，说明已有其他线程改动了这个数，线程A会取出新数并进行加1操作。

乐观锁现在已有原子类实现，写代码时使用封装的原子类即可。

```
public class MyRunnable implements Runnable {  
    //整数修改的乐观锁（不同功能的乐观锁不同）  
    private AtomicInteger count = new AtomicInteger();  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
  
            System.out.println(Thread.currentThread().getname() +  
                "count====>" +  
                count.incrementAndGet());  
        }  
    }  
}
```

## 网络编程

### 概述

1. 网络编程：编写的应用程序可以与网络上其他设备中的应用程序进行数据交互。

java.net.\*包下提供网络编程的解决方案。

2. 通信的基本架构

(1) CS架构 (Client 客户端/Server服务端)：CS架构需要用户在自己的电脑或者手机上安装客户端软件，然后由客户端软件通过网络连接服务器程序，由服务器把数据发给客户端，客户端就可以在页面上看到各种数据了。

(2) BS架构 (Brower 浏览器/Server服务端)：BS架构不需要开发客户端软件，用户只需要通过浏览器输入网址就可以直接从服务器获取数据，并由服务器将数据返回给浏览器，用户在页面上就可以看到各种数据了。

### 网络编程三要素

1. IP地址

IP地址全称互联网协议地址，是分配给网络设备的唯一表示。

127.0.0.1和localhost都表示本机IP地址。

两个和IP地址相关的命令：

**ipconfig:** 查看本机的ip地址

**ping 域名/ip** 检测当前电脑与指定的ip是否连通

## 2. InetAddress类

Java中有一个类用来表IP地址，这个类是InetAddress类。我们在开发网络通信程序的时候，可能有时候会获取本机的IP地址，以及测试与其他地址是否连通，这个时候就可以使用InetAddress类来完成。

### InetAddress

- 代表IP地址。

#### InetAddress的常用方法如下

名称	说明
public static InetAddress getLocalHost()	获取本机IP，会以一个InetAddress的对象返回
public static InetAddress getByName(String host)	根据ip地址或者域名，返回一个InetAddress对象
public String getHostName()	获取该ip地址对象对应的主机名。
public String getHostAddress()	获取该ip地址对象中的ip地址信息。
public boolean isReachable(int timeout)	在指定毫秒内，判断主机与该ip对应的主机是否能连通

```
public class InetAddressTest {  
    public static void main(String[] args) throws Exception {  
        // 1、获取本机IP地址对象的  
        InetAddress ip1 = InetAddress.getLocalHost();  
        System.out.println(ip1.getHostName());  
        System.out.println(ip1.getHostAddress());  
  
        // 2、获取指定IP或者域名的IP地址对象。  
        InetAddress ip2 =  
            InetAddress.getByName("www.baidu.com");  
        System.out.println(ip2.getHostName());  
        System.out.println(ip2.getHostAddress());  
  
        // ping www.baidu.com  
        System.out.println(ip2.isReachable(6000)); //判断服务器是否宕机  
    }  
}
```

## 3. 端口号

端口号：指的是计算机设备上运行的应用程序的标识，被规定为一个16位的二进制数据，范围（0~65535）

端口号分为一下几类

- (1) 周知端口：0~1023，被预先定义的知名应用程序占用（如：HTTP占用80，FTP占用21）
  - (2) 注册端口：1024~49151，分配给用户经常或者某些应用程序
  - (3) 动态端口：49152~65536，之所以称为动态端口，是因为它一般不固定分配给某进程，而是动态分配的。

自己开发的程序一般使用注册端口。

#### 4. 协议

## 传输层协议： UDP、 TCP

## UDP通信

## 1. 构造器和方法

**DatagramSocket**: 用于创建客户端、服务端

构造器	说明
<code>public DatagramSocket()</code>	创建 <b>客户端</b> 的Socket对象，系统会随机分配一个端口号。
<code>public DatagramSocket(int port)</code>	创建 <b>服务端</b> 的Socket对象，并指定端口号

方法	说明
public void send(DatagramPacket dp)	发送数据包
public void receive(DatagramPacket p)	使用数据包接收数据

**DatagramPacket:** 创建数据包

构造器	说明
<code>public DatagramPacket(byte[] buf, int length, InetAddress address, int port)</code>	创建发出去的数据包对象
<code>public DatagramPacket(byte[] buf, int length)</code>	创建用来接收数据的数据包

客户端负责发送数据，服务端负责接收数据。

## 2. 客户端程序

```
/**  
 * 目标：完成UDP通信快速入门：实现客户端反复的发。  
 */  
public class Client {  
    public static void main(String[] args) throws Exception {  
        // 1、创建客户端对象（发韭菜出去的人）  
        DatagramSocket socket = new DatagramSocket();  
  
        // 2、创建数据包对象封装要发出去的数据（创建一个韭菜盘子）  
        /* public DatagramPacket(byte buf[], int length,  
           InetAddress address, int port)  
    }  
}
```

```
参数一：封装要发出去的数据。  
参数二：发送出去的数据大小（字节个数）  
参数三：服务端的IP地址（找到服务端主机）  
参数四：服务端程序的端口。  
*/  
Scanner sc = new Scanner(System.in);  
while (true) {  
    System.out.println("请说: ");  
    String msg = sc.nextLine();  
  
    // 一旦发现用户输入的exit命令，就退出客户端  
    if("exit".equals(msg)){  
        System.out.println("欢迎下次光临！退出成功！");  
        socket.close(); // 释放资源  
        break; // 跳出死循环  
    }  
  
    byte[] bytes = msg.getBytes();  
    DatagramPacket packet = new DatagramPacket(bytes,  
bytes.length  
        , InetAddress.getLocalHost(), 6666);  
  
    // 3、开始正式发送这个数据包的数据出去了  
    socket.send(packet);  
}  
}
```

### 3. 服务端程序

```
/*
 * 目标：完成UDP通信快速入门-服务端反复的收
 */
public class Server {
    public static void main(String[] args) throws Exception {
        System.out.println("----服务端启动----");
        // 1、创建一个服务端对象（创建一个接韭菜的人） 注册端口
        DatagramSocket socket = new DatagramSocket(6666);

        // 2、创建一个数据包对象，用于接收数据的（创建一个韭菜盘子）
        byte[] buffer = new byte[1024 * 64]; // 64KB.
        DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);

        while (true) {
            // 3、开始正式使用数据包来接收客户端发来的数据
        }
    }
}
```

```

        socket.receive(packet);

        // 4、从字节数组中，把接收到的数据直接打印出来
        // 接收多少就倒出多少
        // 获取本次数据包接收了多少数据。
        int len = packet.getLength();

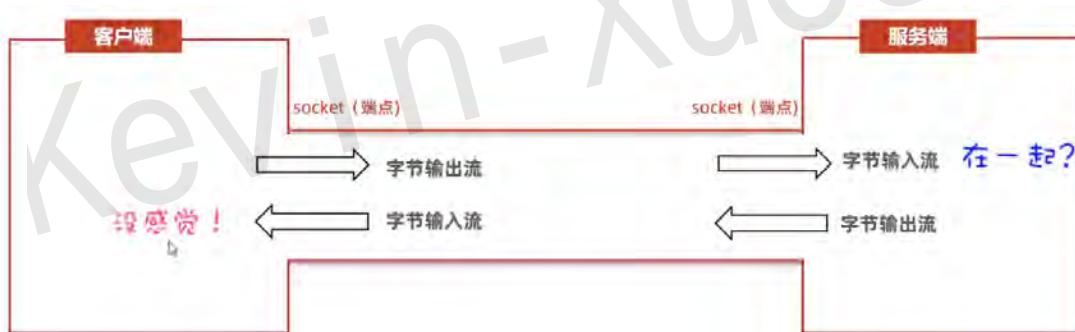
        String rs = new String(buffer, 0 , len);
        System.out.println(rs);

        System.out.println(packet.getAddress().getHostAddress());
        System.out.println(packet.getPort());
        System.out.println("-----");
    }
}
}

```

## TCP通信

### 1. 模型



TCP是双端通信的

### 2. 客户端程序

- 客户端程序就是通过java.net包下的Socket类来实现的。

构造器	说明
public Socket(String host , int port)	根据指定的服务器ip、端口号请求与服务端建立连接，连接通过，就获得了客户端socket
方法	说明
public OutputStream getOutputStream()	获得字节输出流对象
public InputStream getInputStream()	获得字节输入流对象

```

/**
 * 目标：完成TCP通信快速入门-客户端开发：实现客户端可以反复的发消息出去
 */

```

```

public class Client {
    public static void main(String[] args) throws Exception {
        // 1、创建Socket对象，并同时请求与服务端程序的连接。
        Socket socket = new Socket("127.0.0.1", 8888);

        // 2、从socket通信管道中得到一个字节输出流，用来发数据给服务端程序。
        OutputStream os = socket.getOutputStream();

        // 3、把低级的字节输出流包装成数据输出流
        DataOutputStream dos = new DataOutputStream(os);

        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.println("请说：");
            String msg = sc.nextLine();

            // 一旦用户输入了exit，就退出客户端程序
            if("exit".equals(msg)){
                System.out.println("欢迎您下次光临！退出成功！");
                dos.close();
                socket.close();
                break;
            }

            // 4、开始写数据出去了
            dos.writeUTF(msg);
            dos.flush();
        }
    }
}

```

### 3. 服务端程序

#### ServerSocket

构造器	说明
public ServerSocket(int port)	为服务端程序注册端口

方法	说明
public Socket accept()	阻塞等待客户端的连接请求，一旦与某个客户端成功连接，则返回服务端这边的Socket对象。

```

/**
 * 目标：完成TCP通信快速入门-服务端开发：实现服务端反复发消息
 */
public class Server {

```

```
public static void main(String[] args) throws Exception {
    System.out.println("-----服务端启动成功-----");
    // 1、创建ServerSocket的对象，同时为服务端注册端口。
    ServerSocket serverSocket = new ServerSocket(8888);

    // 2、使用serverSocket对象，调用一个accept方法，等待客户端的连接请求
    Socket socket = serverSocket.accept();

    // 3、从socket通信管道中得到一个字节输入流。
    InputStream is = socket.getInputStream();

    // 4、把原始的字节输入流包装成数据输入流
    DataInputStream dis = new DataInputStream(is);

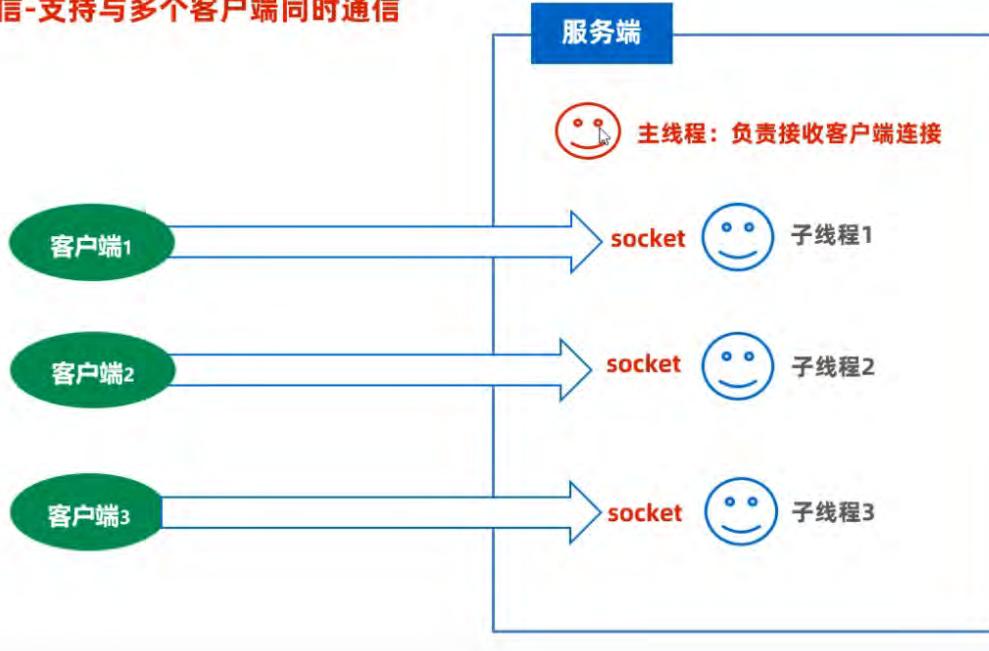
    while (true) {
        try {
            // 5、使用数据输入流读取客户端发送过来的消息
            String rs = dis.readUTF();
            System.out.println(rs);
        } catch (Exception e) {

            System.out.println(socket.getRemoteSocketAddress() + "离线了！");
            dis.close();
            socket.close();
            break;
        }
    }
}
```

## TCP支持与多个客户端通信

### 1. 原理

## TCP通信-支持与多个客户端同时通信



### 2. 代码

首先，我们需要写一个服务端的读取数据的线程类，代码如下

```
public class ServerReaderThread extends Thread{  
    private Socket socket;  
    public ServerReaderThread(Socket socket){  
        this.socket = socket;  
    }  
    @Override  
    public void run() {  
        try {  
            InputStream is = socket.getInputStream();  
            DataInputStream dis = new DataInputStream(is);  
            while (true){  
                try {  
                    String msg = dis.readUTF();  
                    System.out.println(msg);  
  
                } catch (Exception e) {  
                    System.out.println("有人下线了：" +  
socket.getRemoteSocketAddress());  
                    dis.close();  
                    socket.close();  
                    break;  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }  
}
```

接下来，再改写服务端的主程序代码，如下：

```
/**  
 * 目标：完成TCP通信快速入门-服务端开发：要求实现与多个客户端同时通信。  
 */  
public class Server {  
    public static void main(String[] args) throws Exception {  
        System.out.println("----服务端启动成功-----");  
        // 1、创建ServerSocket的对象，同时为服务端注册端口。  
        ServerSocket serverSocket = new ServerSocket(8888);  
  
        while (true) {  
            // 2、使用serverSocket对象，调用一个accept方法，等待客户  
            // 端的连接请求  
            Socket socket = serverSocket.accept();  
  
            System.out.println("有人上线了：" +  
                socket.getRemoteSocketAddress());  
  
            // 3、把这个客户端对应的socket通信管道，交给一个独立的线程负  
            // 责处理。  
            new ServerReaderThread(socket).start();  
        }  
    }  
}
```

## 单元测试

### 1. 概念

针对最小的功能单元（方法），编写测试代码对其进行正确性测试。

### 2. 之前测试存在的问题

- (1) 只能在main里编写测试代码，去调用其他方法进行测试。
- (2) 无法实现自动化测试，一个方法测试失败，可能影响其他方法的测试。
- (3) 无法得到测试报告，需要程序员自己观察是否成功。

### 3. Junit单元测试框架

## Junit单元测试框架

- 可以用来对方法进行测试，它是第三方公司开源出来的（很多开发工具已经集成了Junit框架，比如IDEA）

### 优点

- 可以灵活的编写测试代码，可以针对某个方法执行测试，也支持一键完成对全部方法的自动化测试，且各自独立。
- 不需要程序员去分析测试的结果，会自动生成测试报告出来。

UserServiceTest (com.itheima_01单元测试)	10 ms
UserServiceTest.testChu	10 ms
UserServiceTest.testLogin	0 ms



### 具体步骤

- ① 将Junit框架的jar包导入到项目中（注意：IDEA集成了Junit框架，不需要我们自己手工导入了）
- ② 为需要测试的业务类，定义对应的测试类，并为每个业务方法，编写对应的测试方法（必须：公共、无参、无返回值）
- ③ 测试方法上必须声明@Test注解，然后在测试方法中，编写代码调用被测试的业务方法进行测试；
- ④ 开始测试：选中测试方法，右键选择“JUnit运行”，如果测试通过则是绿色；如果测试失败，则是红色

```
public class StringUtil{  
    public static void printNumber(String name){  
        System.out.println("名字长度: "+name.length());  
    }  
}  
  
//测试类  
public class StringUtilTest{  
    @Test //必须要写Test注解  
    public void testPrintNumber(){  
        StringUtil.printNumber("admin");  
        StringUtil.printNumber(null);  
    }  
}
```

## 4. 单元测试断言

断言：程序员可以预测程序的运行结果，检查程序的运行结果是否与预期一致。

原先的测试只能判断程序有无语法错误以及能否考虑到所有情况，断言能判断程序有无逻辑问题。

```
public static int getMaxIndex(String data){  
    if(data == null){  
        return -1;  
    }  
    return data.length();  
}
```

```

public class StringUtilTest{
    @Test
    public void testGetMaxIndex(){
        int index1 = StringUtil.getMaxIndex(null);
        System.out.println(index1);

        int index2 = StringUtil.getMaxIndex("admin");
        System.out.println(index2);

        //断言机制：预测index2的结果
        Assert.assertEquals("方法内部有Bug", 4, index2); //正确是4,
程序是index2
    }
}

```

## 5. Junit框架常用注解

Junit单元测试框架的常用注解(Junit 4.xxxx版本)

注解	说明
@Test	测试类中的方法必须用它修饰才能成为测试方法，才能启动执行
@Before	用来修饰一个实例方法，该方法会在每一个测试方法执行之前执行一次。
@After	用来修饰一个实例方法，该方法会在每一个测试方法执行之后执行一次。
@BeforeClass	用来修饰一个静态方法，该方法会在所有测试方法之前只执行一次。
@AfterClass	用来修饰一个静态方法，该方法会在所有测试方法之后只执行一次。

- 在测试方法执行前执行的方法，常用于：初始化资源。
- 在测试方法执行完后再执行的方法，常用于：释放资源。

Junit单元测试框架的常用注解(Junit 5.xxxx版本)

注解	说明
@Test	测试类中的方法必须用它修饰才能成为测试方法，才能启动执行
@BeforeEach	用来修饰一个实例方法，该方法会在每一个测试方法执行之前执行一次。
@AfterEach	用来修饰一个实例方法，该方法会在每一个测试方法执行之后执行一次。
@BeforeAll	用来修饰一个静态方法，该方法会在所有测试方法之前只执行一次。
@AfterAll	用来修饰一个静态方法，该方法会在所有测试方法之后只执行一次。

- 开始执行的方法：初始化资源。
- 执行完之后的方法：释放资源。

# 反射

## 反射&获取类

### 1. 定义

反射就是：加载类，并允许以编程的方式解剖类中的各种成分。

## 2. 反射学习内容

### 反射学什么？

学习获取类的信息、操作它们

1、反射第一步：加载类，获取类的字节码：Class对象

2、获取类的构造器：Constructor对象

3、获取类的成员变量：Field对象

4、获取类的成员方法：Method对象

### 3. 获取类（获取类的字节码）

获取Class对象的三种方式

- Class c1 = 类名.class
- 调用Class提供方法： public static Class forName(String package);
- Object提供的方法： public Class getClass(); Class c3 = 对象.getClass();

```
public class Test1Class{  
    public static void main(String[] args){  
        Class c1 = Student.class;  
        System.out.println(c1.getName()); //获取全类名,  
com.xxx.student  
        System.out.println(c1.getSimpleName()); //获取简单类名,  
即student  
  
        Class c2 =  
Class.forName("com.itheima.d2_reflect.Student");  
        System.out.println(c1 == c2); //true  
  
        Student s = new Student();  
        Class c3 = s.getClass();  
        System.out.println(c2 == c3); //true  
    }  
}
```

### 获取类的构造器并使用

#### 1. 获取

- Class提供了从类中获取构造器的方法。

方法	说明
Constructor<?>[] getConstructors()	获取全部构造器 (只能获取public修饰的)
Constructor<?>[] getDeclaredConstructors()	获取全部构造器 (只要存在就能拿到)
Constructor<T> getConstructor(Class<?>... parameterTypes)	获取某个构造器 (只能获取public修饰的)
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)	获取某个构造器 (只要存在就能拿到)

## 获取全部构造器

```
public class Test2Constructor{
    @Test
    public void testGetConstructors(){
        //1、反射第一步：必须先得到这个类的Class对象
        Class c = Cat.class;

        //2、获取类的全部构造器
        Constructor[] constructors =
c.getDeclaredConstructors();

        //3、遍历数组中的每一个构造器对象。
        for(Constructor constructor: constructors){
            System.out.println(constructor.getName()+"---> 参数
个数: "+constructor.getParameterCount());
        }
    }
}
```

## 获取单个构造器

```
public class Test2Constructor{
    @Test
    public void testGetConstructor(){
        //1、反射第一步：必须先得到这个类的Class对象
        Class c = Cat.class;

        //2、获取类public修饰的空参数构造器
        Constructor constructor1 = c.getConstructor();
        System.out.println(constructor1.getName()+"---> 参数个
数: "+constructor1.getParameterCount());

        //3、获取private修饰的有两个参数的构造器，第一个参数String类
型，第二个参数int类型
        Constructor constructor2 =
c.getDeclaredConstructor(String.class,int.class);
    }
}
```

```

        System.out.println( constructor2.getName() + "----> 参数个
数: "+constructor1.getParameterCount());
    }
}

```

## 2. 获取类构造器作用：初始化对象返回

Constructor提供的方法	说明
T newInstance(Object... initargs)	调用此构造器对象表示的构造器，并传入参数，完成对象的初始化并返回
public void setAccessible(boolean flag)	设置为true，表示禁止检查访问控制（暴力反射）

```

public class Test{
    @Test
    public void testGetConstructor throws Exception {
        //反射第一步：先得到这个类的class对象
        Class c = Cat.class;
        //获取类的某个构造器
        Constructor constructor = c.getDeclaredConstructor();
        System.out.println(constructor.getName());
        constructor.setAccessible(true); //禁止检查访问权限，能使用
        私有构造器进行初始化
        Cat cat = (Cat) constructor.newInstance(); //用无参构造器
        构造，方法返回默认为Object

        //获取private修饰的有两个参数的构造器，第一个参数String类型，第
        二个参数int类型
        Constructor constructor2 =
            c.getDeclaredConstructor(String.class, int.class);

        System.out.println( constructor2.getName() + "----> 参数个
数: "+constructor1.getParameterCount());
        constructor.setAccessible(true); //禁止检查访问权限，能使用
        私有构造器进行初始化
        Cat cat = (Cat) constructor2.newInstance("xzs", 3); //用
        无参构造器构造，方法返回默认为Object
    }
}

```

## 获取成员变量和成员方法

### 1. 获取成员变量与使用

- Class提供了从类中获取成员变量的方法。

方法	说明
public Field[] getFields()	获取类的全部成员变量（只能获取public修饰的）
public Field[] getDeclaredFields()	获取类的全部成员变量（只要存在就能拿到）
public Field getField(String name)	获取类的某个成员变量（只能获取public修饰的）
public Field getDeclaredField(String name)	获取类的某个成员变量（只要存在就能拿到）

获取到成员变量的作用：依然是赋值、取值。

方法	说明
void set(Object obj, Object value);	赋值
Object get(Object obj)	取值
public void setAccessible(boolean flag)	设置为true，表示禁止检查访问控制（暴力反射）

```

public class Test2Constructor(){
    @Test
    public void testGetFields() throws Exception{
        //1、反射第一步：必须先得到这个类的Class对象
        Class c = Cat.class;

        //2、获取类的全部成员变量
        Field[] fields = c.getDeclaredFields();

        //3、定位某个成员变量
        Field fName = c.getDeclaredField("name");
        System.out.println(fName.getName() + "---->" +
fName.getType());

        //4、赋值
        Cat cat = new Cat();
        fName.setAccessible(true);
        fName.set(cat, "kafe");

        //5、取值
        String name = (String) fName.get(cat);
    }
}

```

## 2. 获取成员方法和使用

- Class提供了从类中获取成员方法的API。

方法	说明
Method[] getMethods()	获取类的全部成员方法 (只能获取public修饰的)
Method[] getDeclaredMethods()	获取类的全部成员方法 (只要存在就能拿到)
Method getMethod(String name, Class<?>... parameterTypes)	获取类的某个成员方法 (只能获取public修饰的)
Method getDeclaredMethod(String name, Class<?>... parameterTypes)	获取类的某个成员方法 (只要存在就能拿到)

成员方法的作用：依然是执行

Method提供的方法	说明
public Object invoke(Object obj, Object... args)	触发某个对象的该方法执行。
public void setAccessible(boolean flag)	设置为true，表示禁止检查访问控制（暴力反射）

```

public class Test3Method{
    public static void main(String[] args){
        //1、反射第一步：先获取到Class对象
        Class c = Cat.class;

        //2、获取类中的全部成员方法
        Method[] methods = c.getDeclaredMethods();

        //3、遍历这个数组中的每一个方法对象
        for(Method method : methods){
            System.out.println(method.getName()+"-->" +method.getParameterCount()+"-->" +method.getReturnType());
        }

        System.out.println("-----");
        //4、获取private修饰的run方法，得到Method对象
        Method run = c.getDeclaredMethod("run"); //拿run方法，该方法无形参
        //执行run方法，在执行前需要取消权限检查
        Cat cat = new Cat();
        run.setAccessible(true);
        Object rs1 = run.invoke(cat);
        System.out.println(rs1)

        //5、获取private 修饰的eat(String name)方法，得到Method对象
        Method eat = c.getDeclaredMethod("eat",String.class);
        eat.setAccessible(true);
        Object rs2 = eat.invoke(cat,"鱼儿");
        System.out.println(rs2)
    }
}

```

# 反射的作用

## 1. 反射作用

- (1) 得到一个类的全部成分然后操作。
- (2) 可以破坏封装性。
- (3) 适合做Java的框架。基本上，主流的框架都会基于反射设计出一些通用的功能。

## 2. 简易版框架实现

### 需求：

- 对于任意一个对象，该框架都可以把对象的字段名和对应的值，保存到文件中去。

### 实现步骤

- ① 定义一个方法，可以接收任意对象。
- ② 每收到一个对象后，使用反射获取该对象的Class对象，然后获取全部的成员变量。
- ③ 遍历成员变量，然后提取成员变量在该对象中的具体值。
- ④ 把成员变量名、和其值，写出到文件中去即可。

写一个ObjectFrame表示自己设计的框架，代码如下图所示

```
public class ObjectFrame{  
    public static void saveObject(Object obj) throws  
Exception{  
    PrintStream ps =  
        new PrintStream(new FileOutputStream("模块名  
\\src\\data.txt",true));  
        //1)参数obj对象中有哪些属性，属性名是什么实现值是什么，只有对象自  
己最清楚。  
        //2)接着就通过反射获取类的成员变量信息了（变量名、变量值）  
        Class c = obj.getClass(); //获取字节码  
        ps.println("-----"+c.getSimpleName()+"-----  
-----");  
  
        Field[] fields = c.getDeclaredFields(); //获取所有成员变  
量  
        //3)把变量名和变量值写到文件中去  
        for(Field field : fields){  
            String name = field.getName();  
            Object value = field.get(obj)+"";  
        }  
    }  
}
```

```
        ps.println(name);
    }
    ps.close();
}
}
```

使用自己设计的框架，往文件中写入Student对象的信息和Teacher对象的信息。

先准备好Student类和Teacher类

```
public class Student{
    private String name;
    private int age;
    private char sex;
    private double height;
    private String hobby;
}
```

```
public class Teacher{
    private String name;
    private double salary;
}
```

创建一个测试类，在测试中类创建一个Student对象，创建一个Teacher对象，用ObjectFrame的方法把这两个对象所有的属性名和属性值写到文件中去。

```
public class Test5Frame{
    @Test
    public void save() throws Exception{
        Student s1 = new Student("黑马吴彦祖", 45, '男', 185.3,
"篮球, 冰球, 阅读");
        Teacher s2 = new Teacher("播妞", 999.9);

        ObjectFrame.save(s1);
        ObjectFrame.save(s2);
    }
}
```

打开data.txt文件，内容如下图所示，就说明我们这个框架的功能已经实现了。

# 注解

## 自定义注解

### 1. 注解 (Annotation) 定义

注解是Java代码里的特殊标记，比如@Override，作用是让其他程序根据注解信息来决定怎么执行该程序。

注解不光可以用在方法上，还可以用在类上、变量上、构造器上等位置。

### 2. 自定义注解

格式为

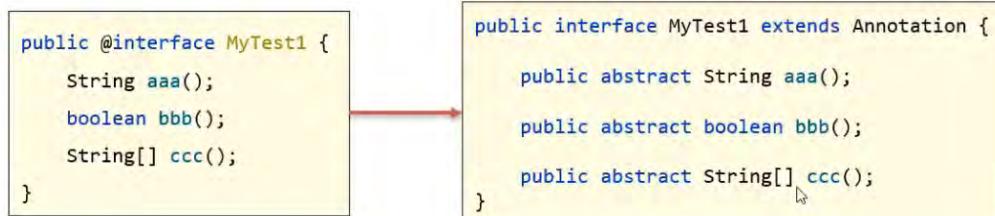
```
public @interface 注解名称 {  
    属性类型 属性名() default 默认值 ;  
}
```

```
public @interface MyTest{  
    String aaa();  
    boolean bbb() default true; //default true 表示默认值为true，  
    使用时可以不赋值。  
    String[] ccc();  
}
```

特殊属性名value：如果注解中只有一个value属性，使用注解时，value名称可以不写。

### 3. 注解的本质

#### 注解的原理



1. 注解本质上是接口，每一个注解接口都继承自Annotation接口
2. 注解中的属性本质上是抽象方法
3. @MyTest1实际上是作为MyTest接口的实现类对象
4. @MyTest1(aaa="孙悟空", bbb=false, ccc={"Python", "前端", "Java"})里面的属性值，可以通过调用aaa()、bbb()、ccc()方法获取到。

# 元注解

1. 元注解：修饰注解的注解。元注解主要是两个，分别是Target和Retention。
2. @Target

## @Target

作用：声明被修饰的注解只能在哪些位置使用

```
@Target(ElementType.TYPE)
1. TYPE, 类, 接口
2. FIELD, 成员变量
3. METHOD, 成员方法
4. PARAMETER, 方法参数
5. CONSTRUCTOR, 构造器
6. LOCAL_VARIABLE, 局部变量
```

```
@Target(ElementType.TYPE) //声明@MyTest3注解只能用在类上，用在其他
                           //地方会报错
//@Target(ElementType.TYPE, ElementType.METHOD) 声明@MyTest3注解
                           //只能用在类上和方法上
public @interface MyTest3{
}
```

3. @Retention

## @Retention

作用：声明注解的保留周期。

```
@Retention(RetentionPolicy.RUNTIME)
1. SOURCE
● 只作用在源码阶段，字节码文件中不存在。
2. CLASS (默认值)
● 保留到字节码文件阶段，运行阶段不存在。
3. RUNTIME (开发常用)
● 一直保留到运行阶段。
```

```

//声明@MyTest3注解只能用在类上和方法上
@Target({ElementType.TYPE, ElementType.METHOD})
//控制使用了@MyTest3注解的代码中，@MyTest3保留到运行时期
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTest3{

}

```

## 解析注解

1. 注解的解析：判断类上、方法上、成员变量上是否存在注解，并把注解里的内容给解析出来。
2. 解析注解的方法

**指导思想：**要解析谁上面的注解，就应该先拿到谁（class对象、method对象等）

Class、Method、Field、Constructor都实现了AnnotatedElement接口，它们都拥有解析注解的能力。

解析注解套路如下

1. 如果注解在类上，先获取类的字节码对象，再获取类上的注解
  2. 如果注解在方法上，先获取方法对象，再获取方法上的注解
  3. 如果注解在成员变量上，先获取成员变量对象，再获取变量上的注解
- 总之：注解在谁身上，就先获取谁，再用谁获取谁身上的注解

AnnotatedElement接口提供了解析注解的方法	说明
public Annotation[] getDeclaredAnnotations()	获取当前对象上面的注解。
public T getDeclaredAnnotation(Class<T> annotationClass)	获取指定的注解对象
public boolean isAnnotationPresent(Class<Annotation> annotationClass)	判断当前对象上是否存在某个注解

根据方法完成下列需求：

解析注解的案例，具体需求如下：

① 定义注解MyTest4，要求如

- 包含属性：String value()
- 包含属性：double aaa(), 默认值为 100
- 包含属性：String[] bbb()
- 限制注解使用的位置：类和成员方法上
- 指定注解的有效范围：一直到运行时

② 定义一个类叫：Demo，在类中定义一个test1方法，并在该类和其方法上使用MyTest4注解

③ 定义AnnotationTest3测试类，解析Demo类中的全部注解。

① 先定义一个MyTest4注解

```
//声明@MyTest4注解只能用在类上和方法上
@Target({ElementType.TYPE, ElementType.METHOD})
//控制使用了@MyTest4注解的代码中，@MyTest4保留到运行时期
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTest4{
    String value();
    double aaa() default 100;
    String[] bbb();
}
```

② 定义有一个类Demo

```
@MyTest4(value="蜘蛛侠",aaa=99.9, bbb={"至尊宝","黑马"})
public class Demo{
    @MyTest4(value="孙悟空",aaa=199.9, bbb={"紫霞","牛夫人"})
    public void test1(){

    }
}
```

③ 写一个测试类AnnotationTest3解析Demo类上的MyTest4注解

```
public class AnnotationTest3{
    @Test
    public void parseClass(){
        //1.先获取Class对象
        Class c = Demo.class;
```

```

//2. 解析Demo类上的注解
if(c.isAnnotationPresent(MyTest4.class)){
    //获取类上的MyTest4注解
    MyTest4 myTest4 =
    (MyTest4)c.getDeclaredAnnotation(MyTest4.class);
    //获取MyTest4注解的属性值
    System.out.println(myTest4.value());
    System.out.println(myTest4.aaa());
    System.out.println(myTest4.bbb());
}
}

@Test
public void parseMethods(){
    //1. 先获取Class对象
    Class c = Demo.class;

    //2. 解析Demo类中test1方法上的注解MyTest4注解
    Method m = c.getDeclaredMethod("test1");
    if(m.isAnnotationPresent(MyTest4.class)){
        //获取方法上的MyTest4注解
        MyTest4 myTest4 =
        (MyTest4)m.getDeclaredAnnotation(MyTest4.class);
        //获取MyTest4注解的属性值
        System.out.println(myTest4.value());
        System.out.println(myTest4.aaa());
        System.out.println(myTest4.bbb());
    }
}
}

```

3. 应用：利用注解解析和反射可以实现框架，如模拟JUnit框架。

## 动态代理

### 1. 代理介绍

对象可以通过代理来转移部分职责。对象有什么方法想被代理，代理就一定要有对应的方法。

对象和代理之间通过中介来连接，中介是一个接口，里面有对象的方法，代理和对象都是这个中介的实现类。

### 2. 生成动态代理对象

```

public class ProxyUtil {
    public static Star createProxy(BigStar bigStar){

```

```

/* newProxyInstance(ClassLoader loader,
    Class<?>[] interfaces,
    InvocationHandler h)
参数1：用于指定一个类加载器
参数2：指定生成的代理长什么样子，也就是有哪些方法
参数3：用来指定生成的代理对象要干什么事情
*/
// Star starProxy = ProxyUtil.createProxy(s);
// starProxy.sing("好日子") starProxy.dance()
Star starProxy = (Star)
Proxy.newProxyInstance(ProxyUtil.class.getClassLoader(),
    new Class[]{Star.class}, new
InvocationHandler() {
    @Override // 回调方法
    public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
        // 代理对象要做的事情，会在这里写代码
        if(method.getName().equals("sing")){
            System.out.println("准备话筒，收钱20
万");
        }else
        if(method.getName().equals("dance")){
            System.out.println("准备场地，收钱
1000万");
        }
        return method.invoke(bigStar, args);
    }
}
);
return starProxy;
}
}

```

调用我们写好的ProxyUtil工具类，为BigStar对象生成代理对象

```

public class Test {
    public static void main(String[] args) {
        BigStar s = new BigStar("杨超越");
        Star starProxy = ProxyUtil.createProxy(s);

        String rs = starProxy.sing("好日子");
        System.out.println(rs);

        starProxy.dance();
    }
}

```

# Java Web

## HTTP

### 1. B/S架构

Browser/Server，浏览器/服务器 架构模式，它的特点是，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web资源，服务器把Web资源发送给浏览器。

B/S架构的好处：易于维护升级：服务器端升级后，客户端无需任何部署就可以使用到新的版本。

### 2. Java Web交互过程



### 3. HTTP协议特点

- (1) 基于TCP协议：面向连接，安全
- (2) 基于请求-响应模型的：一次请求对应一次响应
- (3) HTTP协议是无状态协议：对于事物处理没有记忆能力。每次请求-响应都是独立的。

缺点：多次请求间不能共享数据

### 4. HTTP请求数据格式

#### HTTP-请求数据格式

- 请求数据分为3部分：
  - 请求行：请求数据的第一行。其中GET表示请求方式，/ 表示请求资源路径，HTTP/1.1表示协议版本
  - 请求头：第二行开始，格式为key: value形式。
  - 请求体：POST请求的最后一部分，存放请求参数

```
GET / HTTP/1.1
Host: www.itcast.cn
Connection: keep-alive
Cache-Control: max-age=0 Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 Chrome/91.0.4472.106
...
```

请求方式中GET和POST有区别，抓包观察请求参数时要注意请求参数位置：

- GET请求和 POST请求区别：
  1. GET请求参数在请求行中，没有请求体。  
POST请求参数在请求体中
  2. GET请求参数大小有限制，POST没有

## 5. HTTP响应数据格式

● 响应数据分为3部分：

1. **响应行**：响应数据的第一行。其中HTTP/1.1表示协议版本，200表示响应状态码，OK表示状态码描述
2. **响应头**：第二行开始，格式为key: value形式
3. **响应体**：最后一部分。存放响应数据

状态码分类	说明
1xx	响应中——临时状态码，表示请求已经接收到，告诉客户端应该继续处理请求或者如果它已经完成则忽略它。
2xx	成功——表示请求已经被成功接收、处理已完毕。
3xx	重定向——通常指让其它地方：它让客户端再发起一个请求以完成整个处理。
4xx	客户端错误——通常发生在客户端，如：客户端的请求一个不存在的资源，或者缺少权限，禁止访问等。
5xx	服务器端错误——通常发生在服务器端，如：服务器挂出异常，稍后出现，HTTP版本不支持等。

```

HTTP/1.1 200 OK
Server: Tengine
Content-Type: text/html
Transfer-Encoding: chunked...
<html>
<head>
  <title></title>
</head>
<body></body>
</html>
  
```

● 常见的HTTP响应头：

- Content-Type: 表示该响应内容的类型，例如text/html, image/jpeg;
- Content-Length: 表示该响应内容的长度（字节数）；
- Content-Encoding: 表示该响应压缩算法，例如gzip；
- Cache-Control: 指示客户端应如何缓存，例如max-age=300表示可以最多缓存300秒

### 响应状态码

状态码	英文描述	解释
200	OK	客户端请求成功，即 <b>处理成功</b> ，这是我们最想看到的状态码
302	Found	指示所请求的资源已移动到由 Location响应头给定的 URL，浏览器会自动重新访问到这个页面
304	Not Modified	告诉客户端，你请求的资源至上次取得后，服务端并未更改，你直接用你本地缓存吧。隐式重定向
400	Bad Request	客户端请求有 <b>语法错误</b> ，不能被服务器所理解
403	Forbidden	服务器收到请求，但是 <b>拒绝提供服务</b> ，比如：没有权限访问相关资源
404	Not Found	<b>请求资源不存在</b> ，一般是URL输入有误，或者网站资源被删除了
428	Precondition Required	<b>服务器要求有条件的请求</b> ，告诉客户端要想访问该资源，必须携带特定的请求头

状态码	英文描述	解释
429	Too Many Requests	<b>太多请求</b> , 可以限制客户端请求某个资源的数量, 配合 Retry-After(多长时间后可以请求)响应头一起使用
431	Request Header Fields Too Large	<b>请求头太大</b> , 服务器不愿意处理请求, 因为它的头部字段太大。请求可以在减少请求头域的大小后重新提交。
405	Method Not Allowed	请求方式有误, 比如应该用GET请求方式的资源, 用了POST
500	Internal Server Error	<b>服务器发生不可预期的错误</b> 。服务器出异常了, 赶紧看日志去吧
503	Service Unavailable	<b>服务器尚未准备好处理请求</b> , 服务器刚刚启动, 还未初始化好
511	Network Authentication Required	<b>客户端需要进行身份验证才能获得网络访问权限</b>

## Tomcat

### 1. Web服务器

Web服务器是一个软件, 对http协议进行封装, 使得程序员不必直接对协议进行操作。具体来说, 接收浏览器的请求, 进行HTTP解析, 并返回响应给浏览器。

同时可以将Web项目部署到服务器中, 对外提供网上浏览服务。

### 2. Tomcat简介

Tomcat支持Servlet/JSP规范, 所以Tomcat也被称为Web容器、Servlet容器。Servlet需要依赖Tomcat才能运行。

### 3. 部署

将项目放到webapps目录下, 即部署完成。一般javaweb项目会被打包成war包, 将war包放到webapps目录下, Tomcat会自动解压缩war文件。

### 4. 创建maven-web项目

## IDEA中创建 Maven Web项目

- Web项目结构：



- 部署的JavaWeb项目结构：开发完成，可以部署的项目
- 编译后的Java字节码文件和resources的资源文件，放到WEB-INF下的classes目录下
- pom.xml中依赖坐标对应的jar包，放入WEB-INF下的lib目录下

# Servlet

## 流程&生命周期&方法&体系

### 1. 快速入门

是什么：动态资源，处理服务器端的逻辑。

#### Servlet 快速入门

##### 1. 创建 web项目，导入 Servlet依赖坐标

```
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>3.1.0</version>
<scope>provided</scope>
</dependency>
```

##### 2. 创建：定义一个类，实现 Servlet接口，并重写接口中所有方法，并在 service方法中输入一句话

```
public class ServletDemo1 implements Servlet {
    public void service(){}
}
```

##### 3. 配置：在类上使用@.WebServlet 注解，配置该 Servlet的访问路径

```
@WebServlet("/demo1")
public class ServletDemo1 implements Servlet {
```

##### 4. 访问：启动 Tomcat，浏览器输入URL 访问该Servlet

```
http://localhost:8080/web-demo/demo1
```

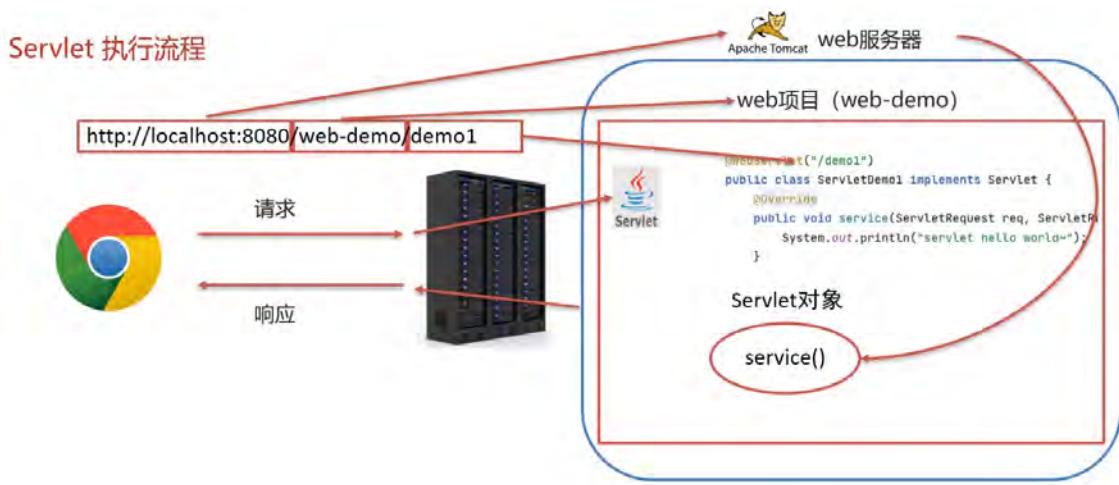
高级软件人才

(1) provided必须写上，provided指的是在编译和测试过程中有效,最后生成的war包时不会加入。

(2) service方法会自动执行，需要打印一句话。

(3) 注解路径以/开头。启动服务器，在服务器的网址后加上注解路径，即可访问Servlet。

### 2. Servlet执行流程



Servlet由web服务器创建，Servlet方法由web服务器调用。

服务器知道Servlet中有service方法是因为我们必须复写service方法。

### 3. Servlet生命周期

- Servlet运行在Servlet容器(web服务器)中，其生命周期由容器来管理，分为4个阶段：
  1. 加载和实例化：默认情况下，当Servlet第一次被访问时，由容器创建Servlet对象
  2. 初始化：在Servlet实例化之后，容器将调用Servlet的init()方法初始化这个对象，完成一些如加载配置文件、创建连接等初始化的工作。该方法只调用一次
  3. 请求处理：每次请求Servlet时，Servlet容器都会调用Servlet的service()方法对请求进行处理。
  4. 服务终止：当需要释放内存或者容器关闭时，容器就会调用Servlet实例的destroy()方法完成资源的释放。在destroy()方法调用之后，容器会释放这个Servlet实例，该实例随后会被Java的垃圾收集器所回收

```

package com.itheima.web;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import java.io.IOException;
/**
 * Servlet生命周期方法
 */
@WebServlet(urlPatterns = "/demo2", loadOnStartup = 1) //多个参数
时要写出参数名字
public class ServletDemo2 implements Servlet {

    /**
     * 初始化方法
     * 1. 调用时机：默认情况下，Servlet被第一次访问时，调用
     *          * LoadOnStartup：默认为-1，修改为0或者正整数，则会在服务器
     * 启动的时候，调用
     * 2. 调用次数：1次
     * @param config
     * @throws ServletException
}

```

```
/*
public void init(ServletConfig config) throws
ServletException {
    System.out.println("init...");
}

/***
 * 提供服务
 * 1. 调用时机:每一次Servlet被访问时, 调用
 * 2. 调用次数: 多次
 * @param req
 * @param res
 * @throws ServletException
 * @throws IOException
 */
public void service(ServletRequest req, ServletResponse
res) throws ServletException, IOException {
    System.out.println("servlet hello world~");
}

/***
 * 销毁方法
 * 1. 调用时机: 内存释放或者服务器关闭的时候, Servlet对象会被销毁, 调用
 * 2. 调用次数: 1次
 */
public void destroy() {
    System.out.println("destroy...");
}

public ServletConfig getServletConfig() {
    return null;
}

public String getServletInfo() {
    return null;
}

}
```

#### 4. Servlet方法

Servlet共五个方法，最常用的三个方法已介绍。

```

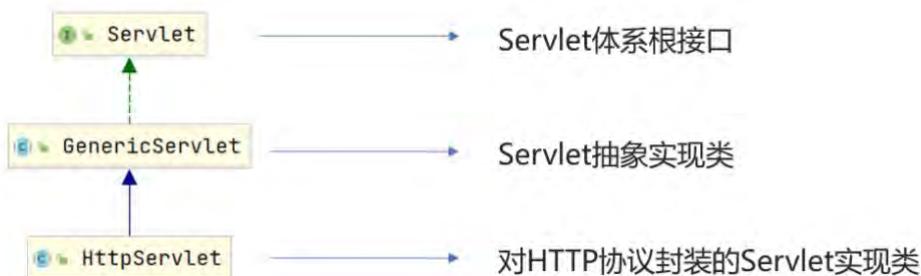
String getServletInfo()
//该方法用来返回Servlet的相关信息，没有什么太大的用处，一般我们返回一个空字符串即可
public String getServletInfo() {
    return "";
}

ServletConfig getServletConfig() //需将init的ServletConfig对象定义为类中局部变量

```

## 5. Servlet体系结构

### Servlet 体系结构



### HttpServlet使用步骤

- (1) 写一个类继承HttpServlet
- (2) 重写doGet方法和doPost方法

```

@WebServlet("/demo4")
public class ServletDemo4 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException
    {
        //TODO GET 请求方式处理逻辑
        System.out.println("get..."); 
    }
    @Override
    protected void doPost(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException
    {
        //TODO Post 请求方式处理逻辑
        System.out.println("post..."); 
    }
}

```

### HttpServlet原理

获取请求方式，根据不同请求方式，调用不同doXXX方法。HttpServlet重写了Servlet中的service()方法。

## 6. urlPattern配置

urlPattern是Servlet的访问路径，一个Servlet可以配置多个urlPattern。

配置规则：

### 2. urlPattern配置规则：

#### ① 精确匹配：

- 配置路径：@WebServlet("/user/select")
- 访问路径：localhost:8080/web-demo/user/select

#### ④ 任意匹配：

- 配置路径：@WebServlet("/")
- 访问路径：localhost:8080/web-demo/hehe
- 访问路径：localhost:8080/web-demo/haha

#### ② 目录匹配：

- 配置路径：@WebServlet("/user/\*")
- 访问路径：localhost:8080/web-demo/user/aaa
- 访问路径：localhost:8080/web-demo/user/bbb

#### ③ 扩展名匹配：

- 配置路径：@WebServlet("\*.do")
- 访问路径：localhost:8080/web-demo/aaa.do
- 访问路径：localhost:8080/web-demo/bbb.do

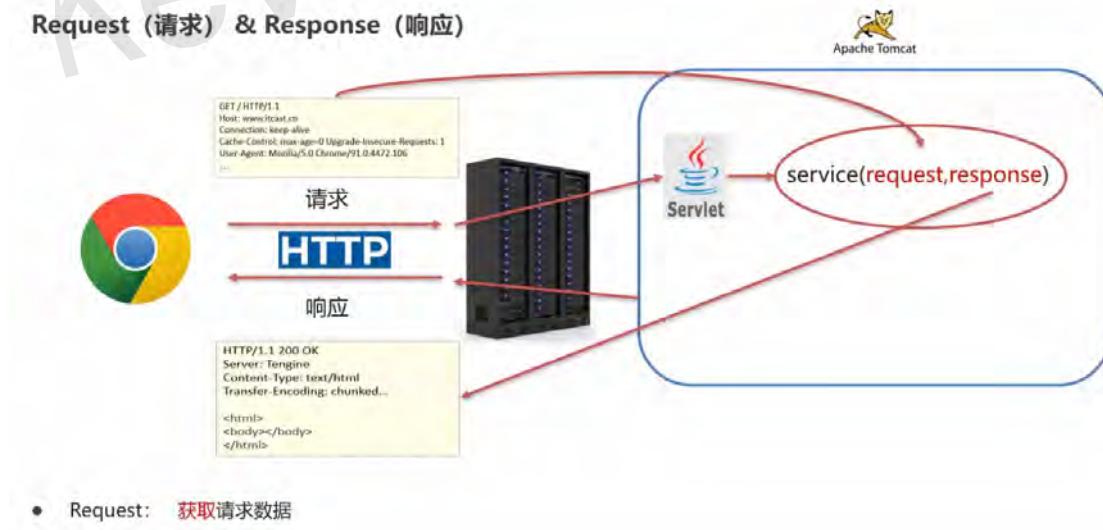
高级软

配置优先级：1>2>3>/\*>/

## Request&Response

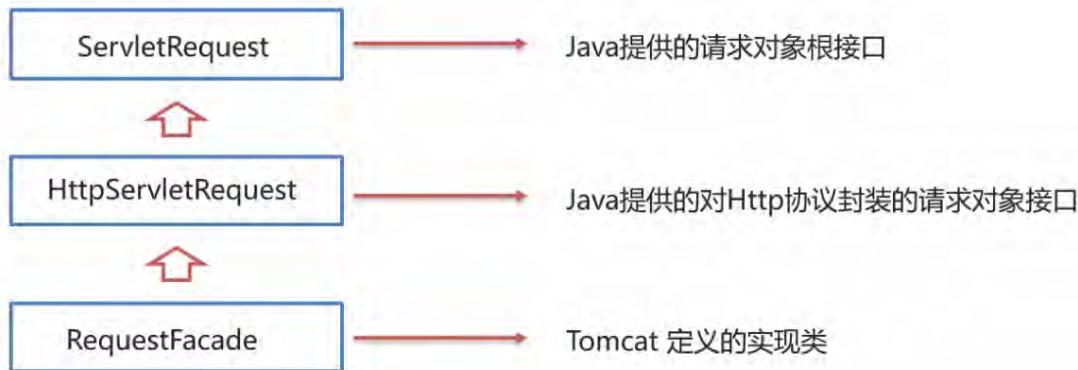
### 1. 介绍

#### Request (请求) & Response (响应)



### 2. Request继承体系

## Request 继承体系



上面两个都是接口，无法创建对象，最下面的是实现类。

### 3. Request 获取请求数据

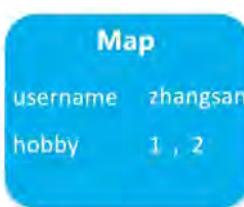
#### Request 获取请求数据

- 请求数据分为3部分：

- 请求行： `GET /request-demo/req1?username=zhangsan HTTP/1.1`
  - `String getMethod()`: 获取请求方式: GET
  - `String getContextPath()`: 获取虚拟目录(项目访问路径): /request-demo
  - `StringBuffer getRequestURL()`: 获取URL(统一资源定位符): http://localhost:8080/request-demo/req1
  - `String getRequestURI()`: 获取URI(统一资源标识符): /request-demo/req1
  - `String getQueryString()`: 获取请求参数 (GET方式) : username=zhangsan&password=123
- 请求头： `User-Agent: Mozilla/5.0 Chrome/91.0.4472.106`
  - `String getHeader(String name)`: 根据请求头名称, 获取值
- 请求体： `username=superbaby&password=123`
  - `ServletInputStream getInputStream()`: 获取字节输入流
  - `BufferedReader getReader()`: 获取字符输入流

#### 获取请求参数的通用方式

- `Map<String, String[]> getParameterMap()`: 获取所有参数Map集合
- `String[] getParameterValues(String name)`: 根据名称获取参数值 (数组)
- `String getParameter(String name)`: 根据名称获取参数值 (单个值)



`Map<String, String[]>`

- (1) 由于键值对中值可能不唯一, map的第二个参数是数组。
- (2) 第二、三个方法传参都是键值对的键名称。

#### 代码通用格式

```

public class RequestDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException
    {
        //采用request提供的获取请求参数的通用方式来获取请求参数
        //编写其他的业务代码...
    }
    @Override
    protected void doPost(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException
    {
        this.doGet(req,resp);
    }
}

```

#### 4. IDEA创建Servlet

##### (1) 修改模板创建内容

settings——>File and Code Templates——>Other——>Web——>Java code templates——>Servlet Annotated Class.java

##### (2) 创建java类时也可以选择下面的Servlet

#### 5. Post请求中文乱码问题

##### 解决方法

```
request.setCharacterEncoding("UTF-8");
```

注意：设置的字符集要和页面保持一致

```

/**
 * 中文乱码问题解决方案
 */
@WebServlet("/req4")
public class RequestDemo4 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        //1. 解决乱码：POST getReader()
        //设置字符输入流的编码，设置的字符集要和页面保持一致
        request.setCharacterEncoding("UTF-8");
        //2. 获取username
        String username = request.getParameter("username");
        System.out.println(username);
    }
}

```

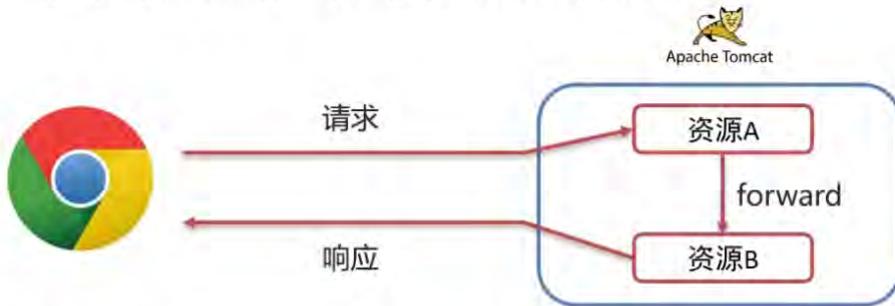
```

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    this doGet(request, response);
}
}

```

## 6. Request请求转发

- 请求转发(forward): 一种在服务器内部的资源跳转方式



- 实现方式:

```
req.getRequestDispatcher("资源B路径").forward(req,resp);
```

- 请求转发资源间共享数据: 使用Request对象

- void setAttribute(String name, Object o): 存储数据到 request域中
- Object getAttribute(String name): 根据 key, 获取值
- void removeAttribute(String name): 根据 key, 删除该键值对

其中req resp是request和response对象。

## 7. Response设置响应数据

- 响应数据分为3部分:

1. 响应行: `HTTP/1.1 200 OK`

- void setStatus(int sc): 设置响应状态码

2. 响应头: `Content-Type: text/html`

- void setHeader(String name, String value): 设置响应头键值对

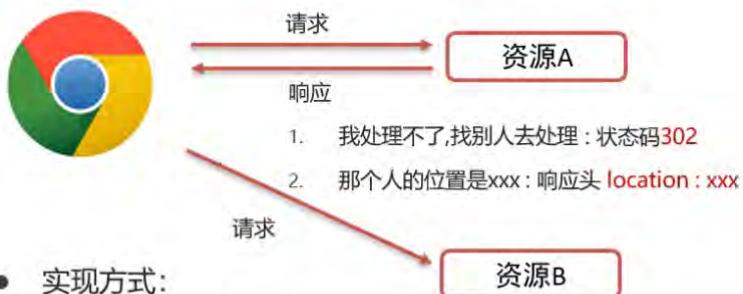
3. 响应体: `<html><head>head</body></html>`

- PrintWriter getWriter(): 获取字符输出流
- ServletOutputStream getOutputStream(): 获取字节输出流

## 8. Response完成重定向

## Response 完成重定向

- 重定向(Redirect): 一种资源跳转方式



- 实现方式:

```
resp.setStatus(302);  
resp.setHeader("location", "资源B的路径");  
resp.sendRedirect("资源B的路径");
```

- 重定向特点:

- 浏览器地址栏路径发生变化
- 可以重定向到任意位置的资源 (服务器内部、外部均可)
- 两次请求, 不能在多个资源使用request共享数据

```
@WebServlet("/resp1")  
public class ResponseDemo1 extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException,  
                         IOException {  
        System.out.println("resp1....");  
        //重定向  
        //1. 设置响应状态码 302  
        response.setStatus(302);  
        //2. 设置响应头 Location  
        response.setHeader("Location", "/request-demo/resp2");  
        //简化形式  
        response.sendRedirect("/request-demo/resp2");  
    }  
  
    @Override  
    protected void doPost(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException,  
                         IOException {  
        this.doGet(request, response);  
    }  
}
```

## 9. 动态获取虚拟目录

```
String contextPath = request.getContextPath();
response.sendRedirect(contextPath + "/resp2");
```

## 10. Response响应字符数据

- 使用：

1. 通过Response对象获取字符输出流

```
PrintWriter writer = resp.getWriter();
```

2. 写数据

```
writer.write("aaa");
```

- 注意：

- 该流不需要关闭，随着响应结束，response对象销毁，由服务器关闭
- 中文数据乱码：原因通过Response获取的字符输出流默认编码：ISO-8859-1

```
resp.setContentType("text/html;charset=utf-8");
```

```
/*
 * 响应字符数据：设置字符数据的响应体
 */
@WebServlet("/resp3")
public class ResponseDemo3 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        response.setContentType("text/html;charset=utf-8");
        //1. 获取字符输出流
        PrintWriter writer = response.getWriter();
        writer.write("aaa");

        PrintWriter writer = response.getWriter();
        //content-type, 告诉浏览器返回的数据类型是HTML类型数据，这样浏览器才会解析HTML标签
        response.setHeader("content-type", "text/html");
        writer.write("<h1>aaa</h1>");
    }
    @Override
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        this.doGet(request, response);
    }
}
```

## 11. Response响应字节数据

```
/**  
 * 响应字节数据：设置字节数据的响应体  
 */  
  
@WebServlet("/resp4")  
public class Responseemo4 extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException,  
                         IOException {  
        //1. 读取文件  
        FileInputStream fis = new  
FileInputStream("d://a.jpg");  
        //2. 获取response字节输出流  
        ServletOutputStream os = response.getOutputStream();  
        //3. 完成流的copy  
        IOUtils.copy(fis,os);  
        fis.close();  
    }  
  
    @Override  
    protected void doPost(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException,  
                         IOException {  
        this.doGet(request, response);  
    }  
}
```