

# Java复习重点

Java复习

面向对象

多线程

IO

集合

泛型

反射

网络编程

设计模式

## 基础

### 自动递增/递减

- **前缀式**

前缀式：先执行运算，再生成值

- **后缀式**

后缀式：先生成值，再执行运算

### String、StringBuffer和StringBuilder

- **String**：提供值不可改变的字符串
- **StringBuffer**：提供值可改变的字符串，线程是安全的，效率低下
- **StringBuilder**：提供值可改变的字符串，线程不安全，效率高.在不考虑线程安全的情况下优先使用它

### Java 中基本类型和字符串之间的转换

其中，基本类型转换为字符串有三种方法：

1. 使用包装类的 toString() 方法
2. 使用String类的 valueOf() 方法
3. 用一个空字符串加上基本类型，得到的就是基本类型数据对应的字符串

```
//将基本数据类型转换为字符串
int c=10;
String str1=Integer.toString(c);
String str2=String.valueOf(c);
String str3=c+"";
```

再来看，将字符串转换成基本类型有两种方法：

1. 调用包装类的 parseXxx 静态方法
2. 调用包装类的 valueOf() 方法转换为基本类型的包装类，会自动拆箱

```
String str="8";
int a=Integer.parseInt(str);
int b=Integer.valueOf(str);
```

注：其他基本类型与字符串的相互转化这里不再一一列出，方法都类似

Java中常用类

• String类

String 类提供了许多用来处理字符串的方法，例如，获取字符串长度、对字符串进行截取、将字符串转换为大写或小写、字符串分割等，下面我们就来领略它的强大之处吧

方法	说明
int length()	返回当前字符串的长度
int indexOf(int ch)	查找ch字符在该字符串中第一次出现的位置
int indexOf(String str)	查找str子字符串在该字符串中第一次出现的位置
int lastIndexOf(int ch)	查找ch字符在该字符串中最后一次出现的位置
int lastIndexOf(String str)	查找str子字符串在该字符串中最后一次出现的位置
String substring(int beginIndex)	获取从beginIndex位置开始到结束的子字符串
String substring(int beginIndex, int endIndex)	获取从beginIndex位置开始到endIndex位置的子字符串
String trim()	返回去除了前后空格的字符串
boolean equals(Object obj)	将该字符串与指定对象比较，返回true或false
String toLowerCase()	将字符串转换为小写
String toUpperCase()	将字符串转换为大写
char charAt(int index)	获取字符串中指定位置的字符
String[] split(String regex,int limit)	将字符串分割为子字符串，返回字符串数组
byte[] getBytes()	将该字符串转换为byte数组

```
public class HelloWorld {
    public static void main(String[] args) {
        // Java文件名
        String fileName = "HelloWorld.jav";
        // 邮箱
        String email = "laurenyang@imooc.com";

        // 判断.java文件名是否正确：合法的文件名应该以.java结尾
        /*
        参考步骤：
        1、获取文件名中最后一次出现"."号的位置
        2、根据"."号的位置，获取文件的后缀
        3、判断"."号位置及文件后缀名
        */
        //获取文件名中最后一次出现"."号的位置
        int index = fileName.indexOf(".");

        // 获取文件的后缀
        String prefix =fileName.substring(index+1);
        // 获取文件的后缀方式二
        String [] arr=fileName.split(".");//按特定符号将字符串分割
        String prefix2=arr[1];

        // 判断必须包含"."号，且不能出现在首位，同时后缀名为"java"
        if
        (fileName.indexOf(".")!=-1&&fileName.indexOf(".")!=0&&prefix.equals("java")) {
            System.out.println("Java文件名正确");
        } else {
            System.out.println("Java文件名无效");
        }

        // 判断邮箱格式是否正确：合法的邮箱名中至少要包含"@"，并且"@"是在"."之前
        /*
        参考步骤：
        1、获取文件名中"@"符号的位置
        2、获取邮箱中"."号的位置
        3、判断必须包含"@"符号，且"@"必须在"."之前
        */
        // 获取邮箱中"@"符号的位置
        int index2 = email.indexOf("@");

        // 获取邮箱中"."号的位置
        int index3 = email.indexOf('.');

        // 判断必须包含"@"符号，且"@"必须在"."之前
```

```

        if (index2 != -1 && index3 > index2) {
            System.out.println("邮箱格式正确");
        } else {
            System.out.println("邮箱格式无效");
        }
    }
}

```

```

public class HelloWorld {
    public static void main(String[] args) {
        // 定义一个字符串
        String s =
        "aljlkdsflkjsadjfklhasdkjflkajdfldwoiudsafhaasdasd";

        // 出现次数
        int num = 0;

        // 循环遍历每个字符，判断是否是字符 a ，如果是，累加次数
        for (int i=0;i<s.length();i++)
        {
            // 获取每个字符，判断是否是字符a
            if (s.charAt(i)=='a') {
                // 累加统计次数
                num++;
            }
        }
        System.out.println("字符a出现的次数: " + num);

        // 将字符串转换为大写
        String str1=str.toUpperCase();

        //将字符串转化为字节数组byte[]
        byte [] b=s.getBytes();
        for(int i=0;i<b.length;i++){
            System.out.println(b[i]);
        }

        //==和equals的使用
        String s2=new
String("aljlkdsflkjsadjfklhasdkjflkajdfldwoiudsafhaasdasd");
        boolean value=s.equals(s1);//比较值是否相等
        boolean address=(s==s1);//比较地址是否相等，新new出来的地址肯定不
相等
    }
}

```

- **StringBuilder 类的常用方法**

StringBuilder 类提供了很多方法来操作字符串：

方法	说明
StringBuilder append(参数)	追加内容到当前StringBuilder对象的末尾
StringBuilder insert(位置, 参数)	将内容插入到StringBuilder对象的指定位置
String toString()	将StringBuilder对象转换为String对象
int length()	获取字符串的长度

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // 创建一个空的StringBuilder对象  
        StringBuilder str=new StringBuilder();  
  
        // 追加字符串  
        str.append("jaewkjldfxmopzdm");  
  
        // 从后往前每隔三位插入逗号  
        for(int i=str.length()-3;i>0;i-=3)  
            str.insert(i,',');  
    }  
  
    // 将StringBuilder对象转换为String对象并输出  
    System.out.print(str.toString());  
}  
}
```

- **Date和SimpleDateFormat**

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) throws ParseException {

        // 使用format()方法将日期转换为指定格式的文本
        SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy年MM月dd
        日 HH时mm分ss秒");
        SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy/MM/dd
        HH:mm");
        SimpleDateFormat sdf3 = new SimpleDateFormat("yyyy-MM-dd
        HH:mm:ss");

        // 创建Date对象，表示当前时间

        Date now=new Date();
        // 调用format()方法，将日期转换为字符串并输出
        System.out.println(sdf1.format(now));
        System.out.println(sdf2.format(now));
        System.out.println(sdf3.format(now));

        // 使用parse()方法将文本转换为日期
        String d = "2014-6-1 21:05:36";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
        HH:mm:ss");

        // 调用parse()方法，将字符串转换为日期
        Date date =sdf.parse(d);

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

## • Calendar

1.Date 类最主要的作用就是获得当前时间，同时这个类里面也具有设置时间以及一些其他的功能，但是由于本身设计的问题，这些方法却遭到众多批评，不建议使用，更推荐使用 Calendar 类进行时间和日期的处理。

2.java.util.Calendar 类是一个抽象类，可以通过调用 getInstance() 静态方法获取一个 Calendar 对象，此对象已由当前日期时间初始化，即默认代表当前时间，如 Calendar c = Calendar.getInstance();

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) {
        // 创建Calendar对象
        Calendar c = Calendar.getInstance();

        // 使用Calendar获取年月日时分秒
        int year=c.get(Calendar.YEAR);
        int month=c.get(Calendar.MONTH)+1;
        int day=c.get(Calendar.DAY_OF_MONTH);
        int hour=c.get(Calendar.HOUR_OF_DAY);
        int minute=c.get(Calendar.MINUTE);
        int second=c.get(Calendar.SECOND);

        // 将Calendar对象转换为Date对象
        Date date = c.getTime();
        System.out.println("当前时间: "+date);

        // 还可以Calendar的时间值，以毫秒为单位
        Long time=c.getTimeInMillis();
        System.out.println("当前毫秒数: "+time);

        // 创建SimpleDateFormat对象，指定目标格式
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

        // 将日期转换为指定格式的字符串
        String now = sdf.format(date);
        System.out.println("当前时间: " + now);
    }
}
```

#### • Math类

Math 类位于 java.lang 包中，包含用于执行基本数学运算的方法，Math 类的所有方法都是静态方法，所以使用该类中的方法时，可以直接使用类名.方法名，如：

Math.round();

返回值	方法名	解释
long	round()	返回四舍五入后的整数
double	floor()	返回小于参数的最大整数
double	ceil()	返回大于参数的最小整数
double	random()	返回 [0, 1) 之间的随机数浮点数

```

public class HelloWorld {

    public static void main(String[] args) {

        // 定义一个整型数组，长度为10
        int[] nums = new int[10];

        //通过循环给数组赋值
        for (int i = 0; i < nums.length; i++) {
            //随机数产生方法一
            // 产生10以内的随机数
            //其中Math.random()方法是一个可以产生[0.0,1.0]区间内的一个双
            //精度浮点数的方法
            //产生一个1-50之间的随机数: int x=1+(int)
            (Math.random()*50)
            int x = (int)(Math.random()*10);

            nums[i] = x; // 为元素赋值
        }

        // 使用foreach循环输出数组中的元素
        for (int random:nums) {
            System.out.print(num + " ");
        }

        // 四舍五入
        double a=12.81;
        long c=Math.round(a); //结果为13
        double d=Math.floor(a); //结果为12.0 向下取整
        double e=Math.ceil(a); //结果为13.0 向上取整
    }

    //随机数产生方法二
    //通过java.util包中的Random类的nextInt方法来得到1-10的int随机数
    Random ra = new Random();
    for (int i=0;i<30;i++)
    {System.out.println(ra.nextInt(10)+1);}
    //Random可以产生随机整数、随机float、随机double，随机long，带种子和不
    //带种子
}

```

## 抽象类与接口

- 1.抽象类由关键字abstract修饰
- 2.应用场景：



- 某个父类只是知道子类应该包含怎样的方法，但无法准确的知道这些子类如何实现这些方法。
- 从多个具有相同特征的类中抽象出一个抽象类，有这个抽象类为模板，从而避免了子类设计的随意性

3.作用：限制规定子类必须实现某些方法，但不关注实现细节

4.使用规则：

- abstract定义抽象类
- abstract定义抽象方法，只声明，不需要实现
- 包含抽象方法的类是抽象类
- 抽象类中可以包含普通方法，也可以没有抽象方法（抽象方法没有方法体）
- 抽象类不能直接创建，可以定义引用变量

```
public abstract class Phone{
    public abstract void call();
    public abstract void send();
}
```

```
public class Nokin extends Phone{

    public void call(){
        syso("a");
    }
    public void send(){
        syso("b");
    }
}
```

```
public class Android extends Phone{
    public void call(){
        syso("c");
    }
    public void send(){
        syso("d");
    }
}
```

```
public class Test{
    public void main(){
        //通过父类的引用创建子类对象
        Phone nokin=new Nokin();
        nokin.call();
        Phone android=new Phone();
        android.send();
    }
}
```

1.接口定义了某一些类所要遵守的 **规范**，接口不关心这些类的内部数据，也不关心内部的实现细节，它只规定这些类里必须提供某些方法。

2.定义时关键字为interface

```
[修饰符] interface 接口名 [extends 父接口1, 父接口2...]{
    零个或多个常量定义
    零个或多个抽象方法
}
```

3.使用接口

一个类可以实现一个或多个接口，实现关键字为implements,通过实现多个接口是对java中类的单继承做一个补充

```
[修饰符] class 类名 extends 父类 implements 接口1, 接口2...{

}
```

4.示例

```
public interface IplayGame{
    public void playGame();
}
```

```
public class Android extends implements IplayGame{

    public void playGame(){
        syso("android play Game");
    }
}
```

```
public class Psp implements IplayGame{

    public void playGame(){
        syso("psp play Game");
    }
}
```

```
public class Test{

    public void main(){
        //接口的引用指向实现了接口的对象
        IplayGame ip1=new Android();
        ip1.playGame();
        IplayGame ip2=new Psp();
        ip2.playGame();

        //使用匿名内部类的方式实现接口
        new IplayGame(){
            public void playGame(){
                syso("使用匿名内部类的方式实现接口");}.playGame();
        }
    }
}
```

## 面向对象

### 类与对象

类：确定对象将会拥有的特征。类是对象的类型，具有相同属性和方法的一组对象集合，是模板，是抽象概念

对象：客观存在的事物，是具体的实体

面向对象：人关注事物信息

### 封装

- **概念**：将类的信息隐藏在类的内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问，隐藏该隐藏的，暴露该暴露的
- **好处**
  - 1.只能通过规定的方法来访问数据
  - 2.提高安全性
  - 3.隐藏类的实现细节，方便修改和实现

- **实现**

- 1.修该可见属性Private
- 2.创建get/set方法
- 3.在getter/settter方法中加入属性的控制语句

- **this和super**

- 1.this表示当前类或当前对象
- 2.this关键字代表当前对象。this.属性：操作当前对象的属性；this.方法：调用当前对象的方法
- 3.封装对象的属性时候
- 4.super表示当前类或对象的父类.访问父类属性：super.age;访问父类方法：super.eat();
- 5.如果显示的调用构造方法，super()一般要将此句放在构造函数第一句
- 6.如果子类的构造方法没有显示的调用父类的构造方法，则系统会默认调用父类的无参构造方法
7. 子类的构造的过程当中必须调用父类的构造方法

- **Java中内部类**

- 1.成员内部类：外部类中嵌套内部类
- 2.局部内部类：定义在方法中的类
- 3.静态内部类：由static修饰的成员内部类
- 4.匿名内部类：通常出现在点击事件、多线程。btn.setOnClickListen(new OnClikListen({}));

## 继承

- **概念**

继承是类与类的一种关系，是一种“is a”的关系，java是单继承

- **好处**

- 1.子类拥有父类的所有属性和方法（不能是private修饰）
- 2.复用父类所写的代码

- **重写和重载**

- 1.重写(父子类)

如果子类对父类的方法不满意，是可以重写父类继承的个方法的，调用方法是优先调用子类的方法。语法规则：a.返回值类型，b.方法名，c.参数类型及个数.都要与父类继承的方法相同

方法的重写要遵循“两同两小一大”规则：

方法名相同

形参列表相同

子类的方法返回值类型应小于或等于父类的方法返回值类型

子类方法声明抛出的异常类应小于或等于父类方法声明抛出的异常类

子类方法的访问权限应大于或等于父类方法的访问权限。

## 2.重载

如果 **同一个类** 中包含了两个以上方法的方法名相同，但形参列表不同，则被称为方法重载。方法重载其实就是上述要素的“两同一不同”。

确定一个方法的三要素：

调用者-->同一个类

方法名-->方法名相同

形参列表-->形参个数和类型不同

- **继承的初始化顺序**

1.初始化父类再初始化子类

2.先执行初始化对象中的属性，再执行构造方法中的的初始化

- **Java中的final和static**

1.final关键字

使用final关键字做标识有“最终的”含义

final可以修饰类、方法、属性和变量

final->修饰类，该类不允许被继承

final->修饰方法，该方法不允许被重写

final->修饰属性，不会自动进行隐式的初始化，需人为在构造方法中进行初始化

final->修饰变量，变量在声明时只能赋一次值，也即变为 **常量**

## 2.static关键字

Java 中被 static 修饰的成员称为静态成员或类成员。它属于整个类所有，而不是某个对象所有，即被类的所有对象所共享。静态成员可以使用类名直接访问。

- 1、静态方法中可以直接调用同类中的静态成员，但不能直接调用非静态成员。
- 2、如果希望在静态方法中调用非静态变量，可以通过创建类的对象，然后通过对象来访问非静态变量。
- 3、在普通成员方法中，则可以直接访问同类的非静态变量和静态变量
- 4、静态方法中不能直接调用非静态方法，需要通过对象来访问非静态方法。
- 5、静态初始化块只在类加载时执行，且`只会执行一次`，同时静态初始化块只能给静态变量赋值，不能初始化普通的成员变量。
- 6、执行顺序  
静态变量->静态代码块->变量->初始化块->构造器

## 多态

对象具有多种形态，其主要体现在1.引用多态，2.方法多态

### 1.引用多态

- 父类的引用可以指向本类的对象
- 父类的引用可以指向子类的对象

```
public class Animal{
    public void eat(){
        syso("Animal");
    }
    public void drink(){
        syso("Animal_drink");
    }
}
```

```
public class Dog extends Animal{
    public void eat(){
        syso("Dog");
    }
}
```

```
public class Test{
    public static void main(String[] args)
        //父类的引用可以指向本类的对象
        Animal obj1=new Animal();
        //父类的引用可以指向子类的对象
        Animal obj2=new Dog();

        obj1.eat();//调用父类方法
        obj2.eat();//调用子类重写的方法

        obj2.drink();//调用子类继承父类后的方法
}
```

## 2.方法多态

- 创建本类对象时，调用方法为本类方法
- 创建子类对象时，调用的方法为子类重写的方法或者继承的方法
- 如果子类中有独有的方法，就不允许有父类引用的子类对象来调用此方法。而只能通过该类引用的该类对象来调用此方法

### 3.引用类型转换

- 向上类型转换（隐式/自动类型转化），是小类型到大类型的转换→无风险（杯子里的水倒到茶壶里）
- 向下类型转换（强制类型转化），是大类型转化小类型→有风险（茶壶里的水倒到杯子，有可能会溢出）
- instanceof运算符，来解决引用对象的类型，避免类型转换的安全性问题

```

Dog dog=new DOg();
Animal animal=dog;//自动类型提升
Dog dog2=(Dog)animal;//向下转，强转

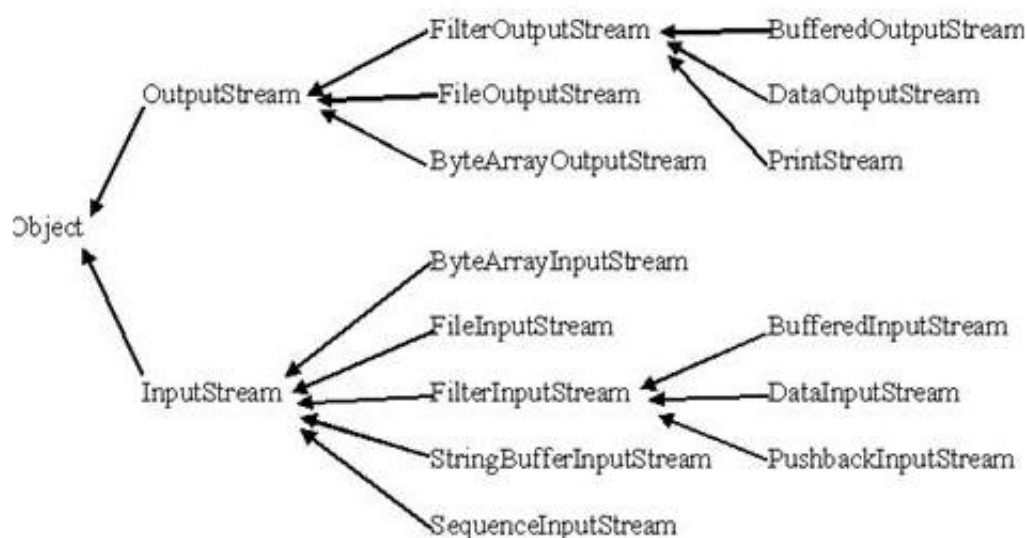
//通过instanceof运算符来避免类型转化的安全性问题
if(animal instanceof Cat){
    Cat cat=(Cat)animal;
}

```

注意：java中A instanceof B中instanceof用来判断内存中实际对象A是不是B类型

## 文件IO

输入流/输出流层次图



### • 创建一个新文件

```

import java.io.*;
class hello{
    public static void main(String[] args) {
        File f=new File("D:\\hello.txt");
        try{
            f.createNewFile();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

### • File类的两个常量

```
import java.io.*;

/*
 *我直接在windows下使用\进行分割不行吗？当然是可以的。但是在linux下就不是\了。所以，要想使得我们的代码跨平台，更加健壮，所以，大家都采用这两个常量吧，
 */
class hello{
    public static void main(String[] args) {
        System.out.println(File.separator);
        System.out.println(File.pathSeparator);
    }
}
```

【运行结果】：

```
\
;
```

#### • 改写后

```
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        try{
            f.createNewFile();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

#### • 删除一个文件



```
/**
 * 删除一个文件
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        if(f.exists()){
            f.delete();
        }else{
            System.out.println("文件不存在");
        }
    }
}
```

- 创建一个文件夹

```
/**
 * 创建一个文件夹
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator+"hello";
        File f=new File(fileName);
        f.mkdir();
    }
}
```

- 列出指定目录的全部文件（包括隐藏文件）：list

```
/**
 * 使用list列出指定目录的全部文件
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator;
        File f=new File(fileName);
        String[] str=f.list();
        for (int i = 0; i < str.length; i++) {
            System.out.println(str[i]);
        }
    }
}
```

- 列出指定目录的全部文件（包括隐藏文件）：listFiles

```
/**
 * 使用listFiles列出指定目录的全部文件
 * listFiles输出的是完整路径
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator;
        File f=new File(fileName);
        File[] str=f.listFiles();
        for (int i = 0; i < str.length; i++) {
            System.out.println(str[i]);
        }
    }
}
```

- 判断一个指定的路径是否为目录

```
/**
 * 使用isDirectory判断一个指定的路径是否为目录
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator;
        File f=new File(fileName);
        if(f.isDirectory()){
            System.out.println("YES");
        }else{
            System.out.println("NO");
        }
    }
}
```

- 搜索指定目录的全部内容

```
/**
 * 列出指定目录的全部内容
 * */
import java.io.*;
class hello{
    public static void main(String[] args) {
        String fileName="D:"+File.separator;
        File f=new File(fileName);
        print(f);
    }
    public static void print(File f){
        if(f!=null){
            if(f.isDirectory()){
                File[] fileArray=f.listFiles();
                if(fileArray!=null){
                    for (int i = 0; i < fileArray.length; i++) {
                        //递归调用
                        print(fileArray[i]);
                    }
                }
            }
            else{
                System.out.println(f);
            }
        }
    }
}
```

- 使用RandomAccessFile写入文件

```
/**
 * 使用RandomAccessFile写入文件
 * */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        RandomAccessFile demo=new RandomAccessFile(f,"rw");
        demo.writeBytes("asdsad");
        demo.writeInt(12);
        demo.writeBoolean(true);
        demo.writeChar('A');
        demo.writeFloat(1.21f);
        demo.writeDouble(12.123);
        demo.close();
    }
}
```

- 向文件中写入数据

```
/**
 * 字符流操作-->Reader和 Writer者两个抽象类
 * 写入数据
 * */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        Writer out =new FileWriter(f);
        String str="hello";
        out.write(str);
        out.close();
    }
}
```

- 从文件中读内容：

```

/**
 * 字符流
 * 从文件中读出内容
 * */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        char[] ch=new char[100];
        Reader read=new FileReader(f);
        int count=read.read(ch);
        read.close();
        System.out.println("读入的长度为: "+count);
        System.out.println("内容为"+new String(ch,0,count));
    }
}

```

- 采用循环读取的方式，因为我们有时候不知道文件到底有多大

```

/**
 * 字符流
 * 从文件中读出内容
 * */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName="D:"+File.separator+"hello.txt";
        File f=new File(fileName);
        char[] ch=new char[100];
        Reader read=new FileReader(f);
        int temp=0;
        int count=0;
        while((temp=read.read())!=(-1)){
            ch[count++]=temp;
        }
        read.close();
        System.out.println("内容为"+new String(ch,0,count));
    }
}

```

- 文件的复制

```
/**
 * 文件的复制
 */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        if(args.length!=2){
            System.out.println("命令行参数输入有误，请检查");
            System.exit(1);
        }
        File file1=new File(args[0]);
        File file2=new File(args[1]);

        if(!file1.exists()){
            System.out.println("被复制的文件不存在");
            System.exit(1);
        }
        InputStream input=new FileInputStream(file1);
        OutputStream output=new FileOutputStream(file2);
        if((input!=null)&&(output!=null)){
            int temp=0;
            while((temp=input.read())!=(-1)){
                output.write(temp);
            }
        }
        input.close();
        output.close();
    }
}
```

- **OutputStreramWriter 和InputStreamReader类**

- 1.整个IO类中除了 字节流 和 字符流 还包括字节和字符 转换流。
- 2.OutputStreramWriter将输出的字节流转化为字符流
- 3.InputStreamReader将输入的字节流转换为字符流

- **将字节输出流转化为字符输出流**

```
/**
 * 将字节输出流转化为字符输出流
 */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName= "d:"+File.separator+"hello.txt";
        File file=new File(fileName);
        Writer out=new OutputStreamWriter(new
FileOutputStream(file));
        out.write("hello");
        out.close();
    }
}
```

- 将字节输入流变为字符输入流

```
/**
 * 将字节输入流变为字符输入流
 */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String fileName= "d:"+File.separator+"hello.txt";
        File file=new File(fileName);
        Reader read=new InputStreamReader(new
FileInputStream(file));
        char[] b=new char[100];
        int len=read.read(b);
        System.out.println(new String(b,0,len));
        read.close();
    }
}
```

- 内存操作流

- 1.前面列举的输出输入都是以文件进行的，现在我们以内容为输出输入目的地，使用内存操作流
- 2.ByteArrayInputStream 主要将内容写内存
- 3.ByteArrayOutputStream 主要将内容从内存输出

- 使用内存操作流将一个大写字母转化为小写字母

```
/**
 * 使用内存操作流将一个大写字母转化为小写字母
 */
import java.io.*;
class hello{
    public static void main(String[] args) throws IOException {
        String str="ROLLENHOLT";
        ByteArrayInputStream input=new
ByteArrayInputStream(str.getBytes());
        ByteArrayOutputStream output=new ByteArrayOutputStream();
        int temp=0;
        while((temp=input.read())!=-1){
            char ch=(char)temp;
            output.write(Character.toLowerCase(ch));
        }
        String outStr=output.toString();
        input.close();
        output.close();
        System.out.println(outStr);
    }
}
```

- 管道流

- 1.管道流主要可以进行两个线程之间的通信。
- 2.PipedOutputStream 管道输出流
- 3.PipedInputStream 管道输入流



```
/**
 * 验证管道流
 */
import java.io.*;

/**
 * 消息发送类
 */
class Send implements Runnable{
    private PipedOutputStream out=null;
    public Send() {
        out=new PipedOutputStream();
    }
    public PipedOutputStream getOut(){
        return this.out;
    }
    public void run(){
        String message="hello , Rollen";
        try{
            out.write(message.getBytes());
        }catch (Exception e) {
            e.printStackTrace();
        }try{
            out.close();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/**
 * 接受消息类
 */
class Recive implements Runnable{
    private PipedInputStream input=null;
    public Recive(){
        this.input=new PipedInputStream();
    }
    public PipedInputStream getInput(){
        return this.input;
    }
    public void run(){
        byte[] b=new byte[1000];
        int len=0;
        try{
            len=this.input.read(b);
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

    }try{
        input.close();
    }catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println("接受的内容为 " + (new String(b, 0, len)));
}
}
/**
 * 测试类
 * */
class hello{
    public static void main(String[] args) throws IOException {
        Send send=new Send();
        Recive recive=new Recive();
        try{
//管道连接
            send.getOut().connect(recive.getInput());
        }catch (Exception e) {
            e.printStackTrace();
        }
        new Thread(send).start();
        new Thread(recive).start();
    }
}

```

## • 打印流

```

/**
 * 使用PrintStream进行输出
 * */
import java.io.*;

class hello {
    public static void main(String[] args) throws IOException {
        PrintStream print = new PrintStream(new
        FileOutputStream(new File("d:"
            + File.separator + "hello.txt")));
        print.println(true);
        print.println("Rollen");
        print.close();
    }
}

```

## • 使用OutputStream向屏幕上输出内容

```
/**
 * 使用OutputStream向屏幕上输出内容
 */
import java.io.*;
class hello {
    public static void main(String[] args) throws IOException {
        OutputStream out=System.out;
        try{
            out.write("hello".getBytes());
        }catch (Exception e) {
            e.printStackTrace();
        }
        try{
            out.close();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

#### • 输入输出重定向

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

/**
 * 为System.out.println() 重定向输出
 */
public class systemDemo{
    public static void main(String[] args){
        // 此刻直接输出到屏幕
        System.out.println("hello");
        File file = new File("d:" + File.separator + "hello.txt");
        try{
            System.setOut(new PrintStream(new
FileOutputStream(file)));
        }catch (FileNotFoundException e){
            e.printStackTrace();
        }
        System.out.println("这些内容在文件中才能看到哦！");
    }
}
```

```

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

/**
 * System.err重定向 这个例子也提示我们可以使用这种方法保存错误信息
 * */
public class systemErr{
    public static void main(String[] args){
        File file = new File("d:" + File.separator + "hello.txt");
        System.err.println("这些在控制台输出");
        try{
            System.setErr(new PrintStream(new
FileOutputStream(file)));
        }catch(FileNotFoundException e){
            e.printStackTrace();
        }
        System.err.println("这些在文件中才能看到哦！");
    }
}

```

#### • 数据操作流DataOutputStream、DataInputStream类

```

import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataOutputStreamDemo{
    public static void main(String[] args) throws IOException{
        File file = new File("d:" + File.separator + "hello.txt");
        char[] ch = { 'A', 'B', 'C' };
        DataOutputStream out = null;
        out = new DataOutputStream(new FileOutputStream(file));
        for(char temp : ch){
            out.writeChar(temp);
        }
        out.close();
    }
}

```

#### • DataInputStream读出内容

```
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class DataOutputStreamDemo{
    public static void main(String[] args) throws IOException{
        File file = new File("d:" + File.separator + "hello.txt");
        DataInputStream input = new DataInputStream(new
FileInputStream(file));
        char[] ch = new char[10];
        int count = 0;
        char temp;
        while((temp = input.readChar()) != 'C'){
            ch[count++] = temp;
        }
        System.out.println(ch);
    }
}
```

## • 对象序列化

- 1.对象序列化就是把一个对象变为二进制数据流的一种方法。
- 2.一个类要想被序列化，就必须实现java.io.Serializable接口。虽然这个接口中没有任何方法，就如同之前的cloneable接口一样。实现了这个接口之后，就表示这个类具有被序列化的能力。

```
import java.io.*;
/**
 * 实现具有序列化能力的类
 * */
public class SerializableDemo implements Serializable{
    private String name;
    private int age;
    public SerializableDemo(){

    }
    public SerializableDemo(String name, int age){
        this.name=name;
        this.age=age;
    }
    @Override
    public String toString(){
        return "姓名: "+name+"  年龄: "+age;
    }
}
```

在继续将序列化之前，先说一下ObjectInputStream和ObjectOutputStream这两个类

- ObjectOutputStream的例子

```
import java.io.Serializable;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

/**
 * 实现具有序列化能力的类
 * */
public class Person implements Serializable{
    public Person(){

    }

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString(){
        return "姓名: " + name + " 年龄: " + age;
    }

    private String name;
    private int age;
}

/**
 * 示范ObjectOutputStream
 * */
public class ObjectOutputStreamDemo{
    public static void main(String[] args) throws IOException{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectOutputStream oos = new ObjectOutputStream(new
        FileOutputStream(
            file));
        oos.writeObject(new Person("rollen", 20));
        oos.close();
    }
}
```

此时运行结果是查看hello.txt文件，看到的是乱码，因为是二进制文件

- 通过ObjectInputStream类来查看上面写入的文件

```
import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;

/**
 * ObjectInputStream示范
 * */
public class ObjectInputStreamDemo{
    public static void main(String[] args) throws Exception{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectInputStream input = new ObjectInputStream(new
        FileInputStream(
            file));
        Object obj = input.readObject();
        input.close();
        System.out.println(obj);
    }
}
```

姓名：rollen 年龄：20

到底序列化什么内容呢？其实只有属性才会被序列化。如果想自定义序列化的内容的时候，就需要实现Externalizable接口。

Serializable接口实现的操作其实是吧一个对象中的全部属性进行序列化，当然也可以使用我们上使用的是Externalizable接口以实现部分属性的序列化，但是这样的操作比较麻烦，

当我们使用Serializable接口实现序列化操作的时候，如果一个对象的某一个属性不想被序列化保存下来，那么我们可以使用transient关键字进行说明：

- 又一序列化与反序列化的完整示例

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

/**
 * 序列化一组对象
 * */
public class SerDemo1{
    public static void main(String[] args) throws Exception{
        Student[] stu = { new Student("hello", 20), new
Student("world", 30),
            new Student("rollen", 40) };
        ser(stu);
        Object[] obj = dser();
        for(int i = 0; i < obj.length; ++i){
            Student s = (Student) obj[i];
            System.out.println(s);
        }
    }

    // 序列化(对象-->二进制数据流)
    public static void ser(Object[] obj) throws Exception{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(
            file));
        out.writeObject(obj);
        out.close();
    }

    // 反序列化(二进制数据流-->对象)
    public static Object[] dser() throws Exception{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectInputStream input = new ObjectInputStream(new
FileInputStream(
            file));
        Object[] obj = (Object[]) input.readObject();
        input.close();
        return obj;
    }
}

class Student implements Serializable{
    public Student(){
```



```

    }

    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString(){
        return "姓名:  " + name + "  年龄: " + age;
    }

    private String name;
    private int age;
}

```

## • 总结

- 1.在java.io包中操作文件内容的主要有两大类：字节流、字符流，两类都分为输入和输出操作。在字节流中输出数据主要是使用OutputStream完成，输入使用的是InputStream，在字符流中输出主要是使用Writer类完成，输入流主要使用Reader类完成。（这四个都是抽象类）<—>(读入写出)
- 2.java中提供了专用于输入输出功能的包Java.io,其中包括：  
InputStream,OutputStream,Reader,Writer  
InputStream 和OutputStream,两个是为字节流设计的,主要用来处理字节或二进制对象，Reader和Writer.两个是为字符流（一个字符占两个字节）设计的,主要用来处理字符或字符串.
- 3.关于字节流和字符流的区别  
实际上字节流在操作的时候本身是不会用到缓冲区的，是文件本身的直接操作的，但是字符流在操作的时候下后是会用到缓冲区的，是通过缓冲区来操作文件的。
- 4.使用字节流好还是字符流好呢？答案是字节流。首先因为硬盘上的所有文件都是以字节的形式进行传输或者保存的，包括图片等内容。但是字符只是在内存中才会形成的，所以在开发中，字节流使用广泛。

## 集合\泛型

## 反射

**反射**：是指在 **运行时** 状态中，获取类中的属性和方法，以及调用其中的方法的一种机制。这种机制的作用在于获取运行时才知道的类（Class）及其中的属性（Field）、方法（Method）以及调用其中的方法，也可以设置其中的属性值。

### Java Reflection API简介

位于java.lang.reflect包中

- Class类：代表一个类，位于java.lang包下。
- Field类：代表类的成员变量（成员变量也称为类的属性）。

- Method类：代表类的方法。
- Constructor类：代表类的构造方法。
- Array类：提供了动态创建数组，以及访问数组的元素的静态方法。

## Class对象

- Java中，无论生成某个类的多少个对象，这些对象都会对应于同一个Class对象。
- 这个Class对象是由JVM生成的，通过它能够获悉整个类的结构
- 在Java中实现反射最重要的一步，也是第一步就是获取Class对象，得到Class对象后可以通过该对象调用相应的方法来获取该类中的属性、方法以及调用该类中的方法。

### 常用的获取Class对象的3种方式：

#### 1.使用Class类的静态方法

```
Class.forName("java.lang.String");
```

#### 2.使用类的.class语法

```
String.class;
```

#### 3.使用对象的getClass()方法

```
String str="aa";  
Class<?> classType1=str.getClass();
```

- 获取方法

```
import java.lang.reflect.Method;

public class DumpMethods
{
    public static void main(String[] args) throws Exception //在方法后加上这句，异常就消失了
    {
        //获得字符串所标识的类的class对象
        Class<?> classType = Class.forName("java.lang.String");//在此处传入字符串指定类名，所以参数获取可以是一个运行期的行为，可以用args[0]

        //返回class对象所对应的类或接口中，所声明的所有方法的数组（包括私有方法）
        Method[] methods = classType.getDeclaredMethods();
        //遍历输出所有方法声明
        for(Method method : methods)
        {
            System.out.println(method);
        }
    }
}
```

- 通过反射调用方法

```
import java.lang.reflect.Method;

public class InvokeTester
{
    public int add(int param1, int param2)
    {
        return param1 + param2;
    }

    public String echo(String message)
    {
        return "Hello: " + message;
    }

    public static void main(String[] args) throws Exception
    {
        // 以前的常规执行手段
        InvokeTester tester = new InvokeTester();
        System.out.println(tester.add(1, 2));
        System.out.println(tester.echo("Tom"));
        System.out.println("-----");

        // 通过反射的方式

        // 第一步，获取Class对象
        // 前面用的方法是：Class.forName()方法获取
        // 这里用第二种方法，类名.class
        Class<?> classType = InvokeTester.class;

        // 生成新的对象：用newInstance()方法
        Object invokeTester = classType.newInstance();
        System.out.println(invokeTester instanceof InvokeTester);

        // 输出true

        // 通过反射调用方法
        // 首先需要获得与方法对应的Method对象
        Method addMethod = classType.getMethod("add", new Class[]
        { int.class,
            int.class });
        // 第一个参数是方法名，第二个参数是这个方法所需要的参数的Class对象的
        数组

        // 调用目标方法
        Object result = addMethod.invoke(invokeTester, new
        Object[] { 1, 2 });
        System.out.println(result); // 此时result是Integer类型
    }
}
```

```

        //调用第二个方法
        Method echoMethod = classType.getDeclaredMethod("echo",
new Class[]{String.class});
        Object result2 = echoMethod.invoke(invokeTester, new
Object[]{"Tom"});
        System.out.println(result2);

    }
}

```

- 生成对象

若想通过类的不带参数的构造方法来生成对象，我们有两种方式：

1.先获得Class对象，然后通过该Class对象的newInstance()方法直接生成即可：

```

Class<?> classType = String.class;
Object obj = classType.newInstance();

```

2.先获得Class对象，然后通过该对象获得对应的Constructor对象，再通过该Constructor对象的newInstance()方法生成（其中Customer是一个自定义的类，有一个无参数的构造方法，也有带参数的构造方法）：

```

Class<?> classType = Customer.class;
// 获得Constructor对象,此处获取第一个无参数的构造方法的
Constructor cons = classType.getConstructor(new Class[] {});
// 通过构造方法来生成一个对象
Object obj = cons.newInstance(new Object[] {});

```

若想通过类的带参数的构造方法生成对象，只能使用下面这一种方式：（Customer为一个自定义的类，有无参数的构造方法，也有一个带参数的构造方法，传入字符串和整型）

```

Class<?> classType = Customer.class;

Constructor cons2 = classType.getConstructor(new Class[]
{String.class, int.class});

Object obj2 = cons2.newInstance(new Object[] {"ZhangSan",20});

```

## Java多线程

## Java网络编程

# 序列化与反序列化

## 示例代码

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

/**
 * 序列化一组对象
 * */
public class SerDemo1{
    public static void main(String[] args) throws Exception{
        Student[] stu = { new Student("hello", 20), new
Student("world", 30),
            new Student("rollen", 40) };
        ser(stu);
        Object[] obj = dser();
        for(int i = 0; i < obj.length; ++i){
            Student s = (Student) obj[i];
            System.out.println(s);
        }
    }

    // 序列化
    public static void ser(Object[] obj) throws Exception{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(
            file));
        out.writeObject(obj);
        out.close();
    }

    // 反序列化
    public static Object[] dser() throws Exception{
        File file = new File("d:" + File.separator + "hello.txt");
        ObjectInputStream input = new ObjectInputStream(new
FileInputStream(
            file));
        Object[] obj = (Object[]) input.readObject();
        input.close();
        return obj;
    }
}

class Student implements Serializable{
    public Student(){
```

```
}

public Student(String name, int age){
    this.name = name;
    this.age = age;
}

@Override
public String toString(){
    return "姓名:  " + name + "  年龄: " + age;
}

private String name;
private int age;
}
```

## 设计模式（常见13种）