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StringBulider

- **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
- **StringBulider**: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.equal
s("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 =
"aljlkdsflkjsadjfklhasdkjlflkajdflwoiudsafhaasdasd";

        // 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("aljlkdsflkjsadjfklhasdkjlflkajdflwoiudsafhaasdasd");
        bolean value=s.equals(s1);//比较值是否相等
        bolean 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Canlendar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(Calender.MONTH)+1;
        int day=c.get(Calendar.DAY_OF_MONTH);
        int hour=c.get(Calendar.HOUR_OF_DAY);
        int minute=c.get(Calendar.MIUNTE);
        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 = 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.抽象类由关键在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
    零个货多个抽象方法
}
```

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implemnets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.封装对象的属性的时候
  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
    obi2.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

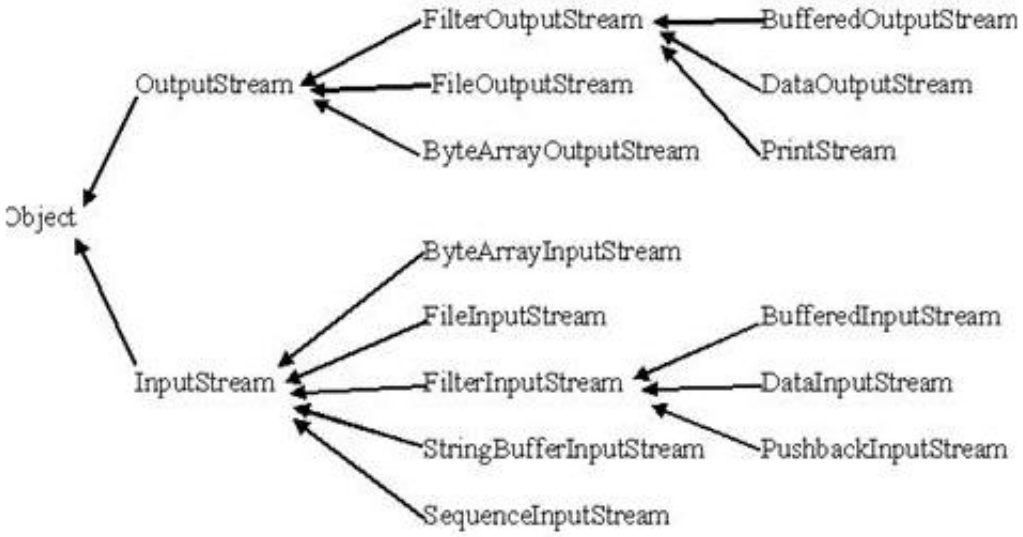
```
Dog dog=new DOg();
Animal animal=dog;//自动类型提升
Dog dog2=(Dog)animal;//向下转，强转

//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**

```
/**
 * 字符流操作用-->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++]=(char)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

整体框架

- **集合类map(映射)、set(集合)、list(列表)总结**

  1、JAVA集合类都放在java.util包中

  2、JAVA集合类不能存放基本数据类型，只能存放对象的引用，基本数据类型应存放在数组中。

  3、Set:集合中对象不按特定的方式排序，并且没有重复对象

  4、List:集合中的对象按照检索位置排序，可以有重复对象。

  5、Map:集合中每一个元素包含一对键-值对象，集合中没有重复的键对象，值对象可以重复

  6、Collection存放单一的元素，Map保存相关联键值对。

  7、ArrayList适用于大量的随机访问，LinkedList适用于经常从表中插入和删除元素

  8、各种队列Queue()和以及栈的行为，都有LinkedList支持。

  9、Map是一种对象与对象相关联的设计。HashMap设计用来快速访问，TreeMap保持键始终处于排序状态。

  10、Set不接受重复元素。HashSet提供最快的查询速度，而TreeSet保持元素处于排序状态

- **精炼的总结：**
  1、Collection 是对象集合， Collection 有两个子接口 List 和 Set
  2、List 可以通过下标 (1,2..) 来取得值，值可以重复
  而 Set 只能通过游标来取值，并且值是不能重复的
  3、ArrayList ， Vector ， LinkedList 是 List 的实现类
  4、ArrayList 是线程不安全的，便于快速查询， Vector 是线程安全的，这两个类底层都是由数组实现的
  5、LinkedList 是线程不安全的，便于插入和删除，底层是由链表实现的
  6、Map 是键值对集合
  7、HashTable 和 HashMap 是 Map 的实现类
  8、HashTable 是线程安全的，不能存储 null 值
  9、HashMap 不是线程安全的，可以存储 null 值

- **Collection ， List ， Set 和 Map 用法和区别:**

1、首先看一下他们之间的关系
Collection 接口的接口 对象的集合
├ List 子接口 按进入先后有序保存 可重复
│ ├ LinkedList 接口实现类 链表 插入删除 没有同步 线程不安全
│ ├ ArrayList 接口实现类 数组 随机访问 没有同步 线程不安全
│ └ Vector 接口实现类 数组 同步 线程安全
│ └ Stack
└ Set 子接口 仅接收一次，并做内部排序
├ HashSet
│ └ LinkedHashSet
└ TreeSet

2、对于 List ，关心的是顺序， 它保证维护元素特定的顺序（允许有相同元素），使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素。
对于 Set ，只关心某元素是否属于 Set （不 允许有相同元素 ），而不关心它的顺序。
Map 接口 键值对的集合
├ Hashtable 接口实现类 同步 线程安全
├ HashMap 接口实现类 没有同步 线程不安全
│ ├ LinkedHashMap
│ └ WeakHashMap
├ TreeMap 保持元素处于排序状态
└ IdentifyHashMap

- **集合的选择标准：**
  存放要求
  无序 - Set
  有序 - List
  “key-value”对 - Map

- **读和改的效率**
  Hash* - 两者都最高
  Array* - 读快改慢
  Linked* - 读慢改快

- **Collection接口的方法：**

```
boolean add(Object o)       ：向集合中加入一个对象的引用
void clear()：删除集合中所有的对象，即不再持有这些对象的引用
boolean isEmpty()     ：判断集合是否为空
boolean contains(Object o) ：  判断集合中是否持有特定对象的引用
Iterartor iterator()   ：返回一个Iterator对象，可以用来遍历集合中的元素
boolean remove(Object o) ：从集合中删除一个对象的引用
int size()        ：返回集合中元素的数目
Object[] toArray()     ：  返回一个数组，该数组中包括集合中的所有元素
关于：Iterator() 和toArray() 方法都用于集合的所有的元素，前者返回一个
Iterator对象，后者返回一个包含集合中所有元素的数组。
```

- **Iterator接口声明了如下方法：**

```
hasNext()：判断集合中元素是否遍历完毕，如果没有，就返回true
next() ：返回下一个元素
remove()：从集合中删除上一个有next()方法返回的元素。
```

- **Set的用法，存放对象引用，没有重复对象**

```
Set set=new HashSet();
String s1=new String("hello");
String s2=s1;
String s3=new String("world");
set.add(s1);
set.add(s2);
set.add(s3);
System.out.println(set.size());//打印集合中对象的数目 为 2。
```

- **List遍历方法一**

```
for(int i=0; i<list.size();i++){
System.out.println(list.get(i));
}
```

- **List遍历方法二**

```
Iterator it=list.iterator();
while(it.hashNext){
    System.out.println(it.next);
}
```

- **Map 的常用方法：**

```
1 添加，删除操作：
Object put(Object key, Object value)：  向集合中加入元素
Object remove(Object key)：  删除与KEY相关的元素
void putAll(Map t)：   将来自特定映像的所有元素添加给该映像
void clear()：从映像中删除所有映射
2 查询操作：
Object get(Object key)：获得与关键字key相关的值
```

- **题**

1、23.ArrayList和Vector有何异同点？
ArrayList和Vector在很多时候都很类似。
（1）两者都是基于索引的，内部由一个数组支持。
（2）两者维护插入的顺序，我们可以根据插入顺序来获取元素。
（3）ArrayList和Vector的迭代器实现都是fail-fast的。
（4）ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。
以下是ArrayList和Vector的不同点。
（1）Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。
（2）ArrayList比Vector快，它因为有同步，不会过载。
（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

2、HashMap和HashTable有何不同？
（1）HashMap允许key和value为null，而HashTable不允许。
（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。
（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。
（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。
（5）HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。

3、如何决定选用HashMap还是TreeMap？
对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

4、遍历一个List有哪些不同的方式？

```
List<String> strList = new ArrayList<>();
//使用for-each循环
for(String obj : strList){
  System.out.println(obj);
}
//using iterator
Iterator<String> it = strList.iterator();
while(it.hasNext()){
  String obj = it.next();
  System.out.println(obj);
}
```

5、Iterater和ListIterator之间有什么区别？
Iterator接口提供遍历任何Collection的接口。我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。
（1）我们可以使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。
（2）Iterator只可以向前遍历，而LIstIterator可以双向遍历。
（3）ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。
6、hashCode()和equals()方法有何重要性？
（1）如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。
（2）如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。
7、Collections类是什么？
Java.util.Collections是一个工具类仅包含静态方法，它们操作或返回集合。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如折半搜索、排序、混编和逆序等。
8、Comparable和Comparator接口有何区别？
Comparable和Comparator接口被用来对对象集合或者数组进行排序。Comparable接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。
Comparator接口被用来提供不同的排序算法，我们可以选择需要使用的Comparator来对给定的对象集合进行排序。

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
        // 首先需要获得与该方法对应的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

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了

定义

线程是进程中某个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

进程与线程的区别

1.线程的划分比进程小。支持多线程的系统比支持多进程的系统并发度高。

2.进程能独立运行，父进程和子进程都有各自独立的数据空间和代码；线程不能独立运行，父线程和子线程共享相同的数据空间并共享系统资源。

3.进程是相对静止的，它代表代码和数据存放的地址空间；而线程是动态的，每个线程代表进程内的一个执行流。

线程的生命周期

Java多线程实现的三种方式：

`继承Thread类`、`实现Runnable接口`、`使用ExecutorService`、Callable、Future实现有返回结果的多线程。其中前两种方式线程执行完后都没有返回值，只有最后一种是带返回值的。

- **继承Thread类来实现多线程**
  到底用Thread呢？还是Runnable。Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口好

```
public class MyThread extends Thread{
    public void run(){
        Syso("MyThread.run");
    }
}
```

```
MyThread myThread1=new MyThread();
MyThread myThread2=new MyThread();
myThread1.start();
myThread2.start();
```

- **实现Runnable接口方式实现多线程**

```
public class MyRunnable extends OtherClass implements Runnable {
    public void run() {
     System.out.println("MyThread.run()");
    }
}
```

```
MyRunnable myRun = new MyRunnable();
Thread thread = new Thread(myRun);
thread.start();
```

- **线程池**
  1、线程池的引入：由于线程的生命周期中包括创建、就绪、运行、阻塞、销毁阶段，当我们待处理的任务数目较小时，我们可以自己创建几个线程来处理相应的任务，但当有大量的任务时，由于创建、销毁线程需要很大的开销，运用线程池这些问题就大大的缓解了
  2、四种线程池的使用
  `newCachedThreadPool`创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
  `newFixedThreadPool`创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
  `newScheduledThreadPool`创建一个定长线程池，支持定时及周期性任务执行。
  `newSingleThreadExecutor`创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
- **(1) newCachedThreadPool**
  创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

```
package test;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
 public static void main(String[] args) {
  ExecutorService cachedThreadPool =
Executors.newCachedThreadPool();
  for (int i = 0; i < 10; i++) {
   final int index = i;
   try {
    Thread.sleep(index * 1000);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
   cachedThreadPool.execute(new Runnable() {
    public void run() {
     System.out.println(index);
    }
   });
  }
 }
}
```

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

- **(2) newFixedThreadPool**
  创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下：

```
package test;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
 public static void main(String[] args) {
  ExecutorService fixedThreadPool =
Executors.newFixedThreadPool(3);
  for (int i = 0; i < 10; i++) {
   final int index = i;
   fixedThreadPool.execute(new Runnable() {
    public void run() {
     try {
      System.out.println(index);
      Thread.sleep(2000);
     } catch (InterruptedException e) {
      e.printStackTrace();
     }
    }
   });
  }
 }
}
```

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。
定长线程池的大小最好根据系统资源进行设置。如
Runtime.getRuntime().availableProcessors()

- **(3)newScheduledThreadPool**
  创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

```
package test;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
public class ThreadPoolExecutorTest {
 public static void main(String[] args) {
  ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(5);
  scheduledThreadPool.schedule(new Runnable() {
   public void run() {
    System.out.println("delay 3 seconds");
   }
  }, 3, TimeUnit.SECONDS);
 }
}
```

表示延迟3秒执行。

- **(4) newSingleThreadExecutor**
  创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

```
package test;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
 public static void main(String[] args) {
  ExecutorService singleThreadExecutor =
Executors.newSingleThreadExecutor();
  for (int i = 0; i < 10; i++) {
   final int index = i;
   singleThreadExecutor.execute(new Runnable() {
    public void run() {
     try {
      System.out.println(index);
      Thread.sleep(2000);
     } catch (InterruptedException e) {
      e.printStackTrace();
     }
    }
   });
  }
 }
}
```

线程的同步

同步：就如当客户端向服务端发送一个请求时，要等到服务端做出回应并返回，客户端才可以进行下一次的请求。如窗口买票就要实现同步，所谓同步就是在同一段时间中只有一个线程运行，其他的线程必须等到这个线程结束后才能继续执行。
同步解决方案：阻塞（wait/notify），互斥（synchronized），信号量（一般不用，不好实现）

- **采用（互斥）同步的话，可以使用 `同步代码块` 和 `同步方法` 两种来完成。**

同步代码块实现：

```
语法格式:
synchronized（同步对象）{
 //需要同步的代码
}
```

```
class hello implements Runnable {
    public void run() {
        for(int i=0;i<10;++i){
            synchronized (this) {
                if(count>0){
                    try{
                        Thread.sleep(1000);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    System.out.println(count--);
                }
            }
        }
    }

    public static void main(String[] args) {
        hello he=new hello();
        Thread h1=new Thread(he);
        Thread h2=new Thread(he);
        Thread h3=new Thread(he);
        h1.start();
        h2.start();
        h3.start();
    }
    private int count=5;
}
```

【运行结果】：（每一秒输出一个结果）
5 4 3 2 1

同步方法实现：

```
也可以采用同步方法。

语法格式为synchronized 方法返回类型 方法名（参数列表）{
    // 其他代码
}
```

```
class hello implements Runnable {
    public void run() {
        for (int i = 0; i < 10; ++i) {
            sale();
        }
    }

    public synchronized void sale() {
        if (count > 0) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(count--);
        }
    }

    public static void main(String[] args) {
        hello he = new hello();
        Thread h1 = new Thread(he);
        Thread h2 = new Thread(he);
        Thread h3 = new Thread(he);
        h1.start();
        h2.start();
        h3.start();
    }

    private int count = 5;
}
```

- **通过等待唤醒操作实现(阻塞方式)**

```
class Info {

    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;
    }

    public synchronized void set(String name, int age){
        if(!flag){
            try{
                super.wait();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
        this.name=name;
        try{
            Thread.sleep(100);
        }catch (Exception e) {
            e.printStackTrace();
        }
        this.age=age;
        flag=false;
        super.notify();
    }

    public synchronized void get(){
        if(flag){
            try{
                super.wait();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }

        try{
            Thread.sleep(100);
```

```
            }catch (Exception e) {
                e.printStackTrace();
            }
            System.out.println(this.getName()+"<===>"+this.getAge());
            flag=true;
            super.notify();
        }
        private String name = "Rollen";
        private int age = 20;
        private boolean flag=false;
    }

    /**
     * 生产者
     * */
    class Producer implements Runnable {
        private Info info = null;

        Producer(Info info) {
            this.info = info;
        }

        public void run() {
            boolean flag = false;
            for (int i = 0; i < 25; ++i) {
                if (flag) {

                    this.info.set("Rollen", 20);
                    flag = false;
                } else {
                    this.info.set("ChunGe", 100);
                    flag = true;
                }
            }
        }
    }

    /**
     * 消费者类
     * */
    class Consumer implements Runnable {
        private Info info = null;

        public Consumer(Info info) {
            this.info = info;
        }

        public void run() {
            for (int i = 0; i < 25; ++i) {
```

```
                try {
                    Thread.sleep(100);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                this.info.get();
            }
        }
    }

    /**
     * 测试类
     * */
    class hello {
        public static void main(String[] args) {
            Info info = new Info();
            Producer pro = new Producer(info);
            Consumer con = new Consumer(info);
            new Thread(pro).start();
            new Thread(con).start();
        }
    }
```

线程中注意的问题

- **Java中如何停止一个线程？**

```
    private class Runner extends Thread{
        boolean bExit = false;//设置一个boolean变量来使用
        public void exit(boolean bExit){
            this.bExit = bExit;
        }
        @Override
        public void run(){
            while(!bExit){
                System.out.println("Thread is running");
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException ex) {

    Logger.getLogger(ThreadTester.class.getName()).log(Level.SEVERE,
    null, ex);
                    }
            }
        }
    }
```

- **如何在两个线程间共享数据？**
  可以通过共享对象来实现这个目的，或者是使用像阻塞队列这样并发的数据结构。这篇教程《Java线程间通信》(涉及到在两个线程间共享对象)用wait和notify方法实现了生产者消费者模型。
- **Java中notify 和 notifyAll有什么区别？**
  这又是一个刁钻的问题，因为多线程可以等待单监控锁，Java API 的设计人员提供了一些方法当等待条件改变的时候通知它们，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以只有一个线程在等 待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。
- **什么是线程池？ 为什么要使用它？**
  创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时 候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短 的任务的程序的可扩展线程池）。
- **Java多线程中的死锁**
  1.死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务
  2.避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。
- **有三个线程T1，T2，T3，怎么确保它们按顺序执行？**
  在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。
- **Thread类中的yield方法有什么作用？**
  Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。
- **Java多线程中调用wait() 和 sleep()方法有什么不同？**
  Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而 sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

Java网络编程

TCP与UDP

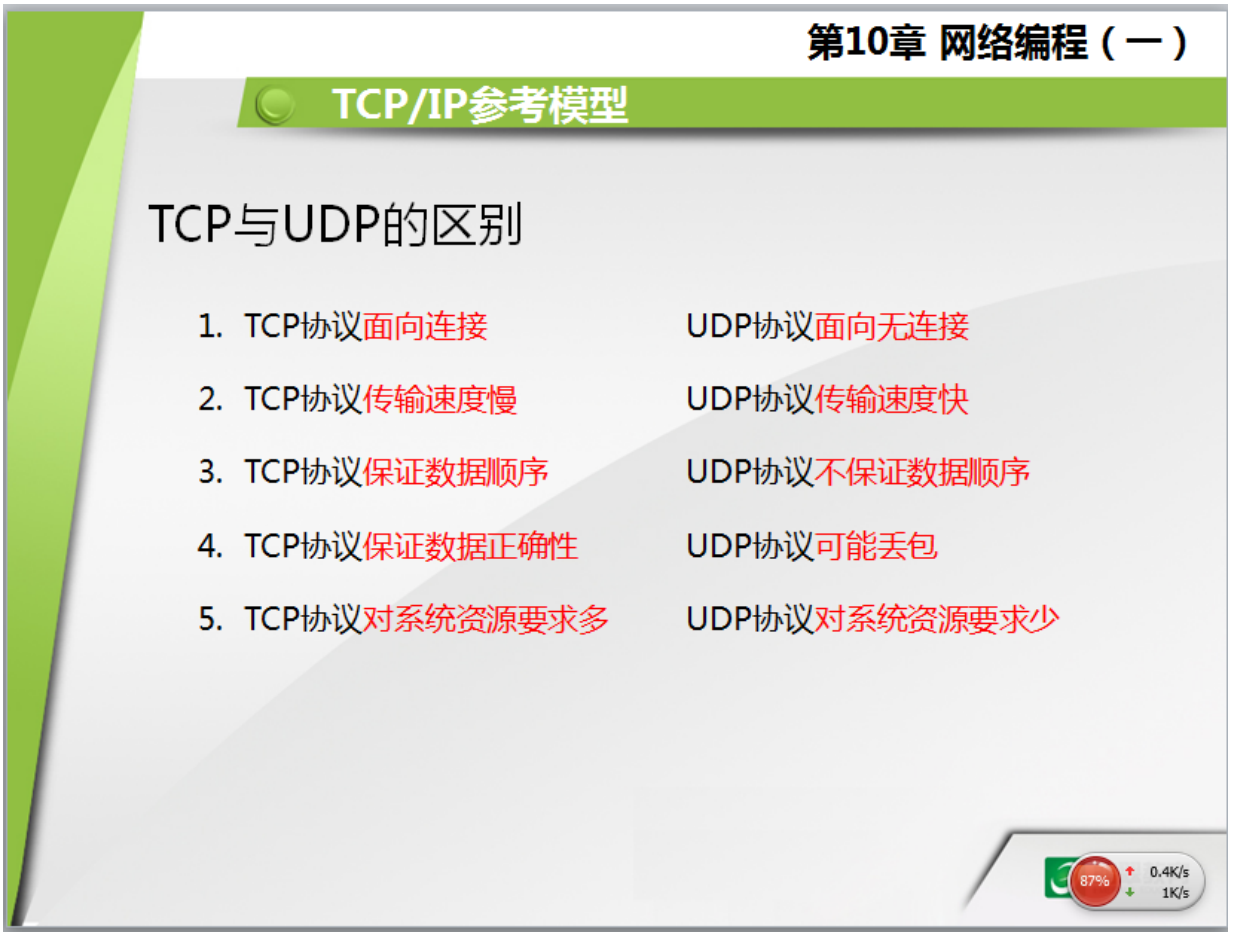
java.net包中提供了两种常见的网络协议的支持：

1.TCP： 传输控制协议（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层通信协议。TCP在任意两台主机之间建立通信链接，两台主机可以通过该链接连接到一起，直到链接终止，连接才回断开。面向连接协议的一个经典例子就是电话通话，电话链接建立起来，通信发生，通讯终止，链接终止。

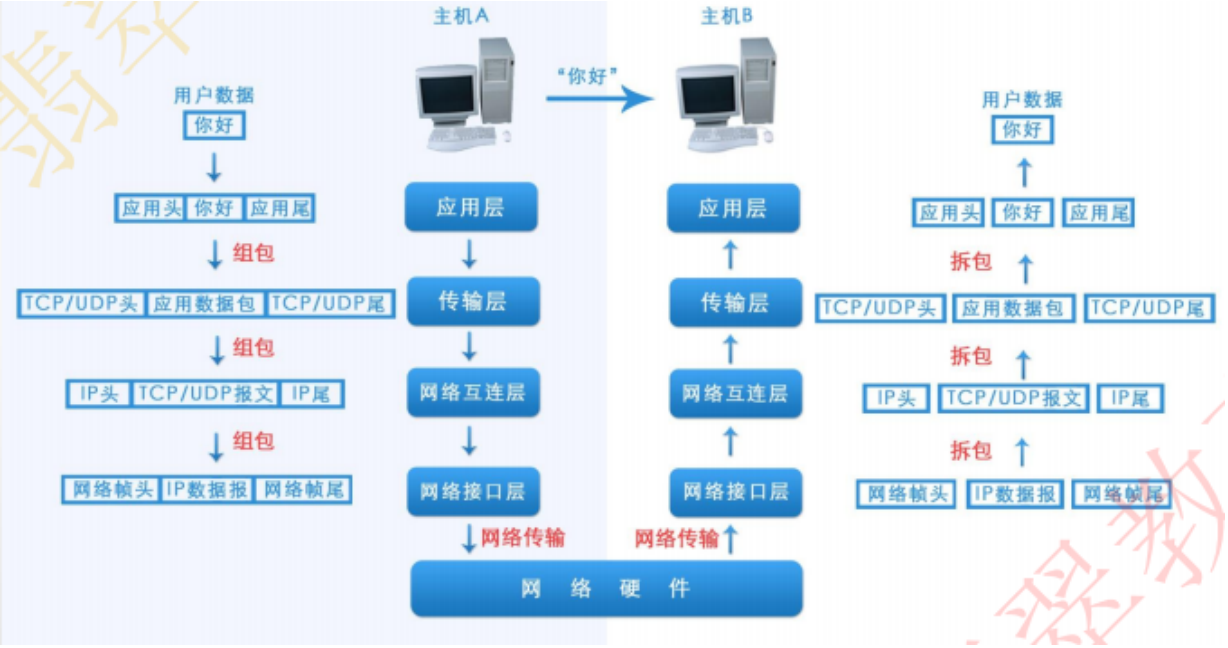
2.UDP: 用户数据包协议（User Datagram Protocol）,它是面向无连接的传输层通讯协议，此中方式以数据包的形式传送，易发生丢包现象，UDP不提供数据报的分组、组装，不能对数据进行排序，也就是说，当保文发送后无法得知是否完整安全的到达。此链接协议用发短信为例：当发短信时，是不需要与对方建立连接的，只需要把短信发送给对方，无论对方是关机还是欠费，短信已经发出，无法确认对方是否收到。

3.IP: 网络之间互联协议（Internet Protocol）为计算机网络互相连接进行通信而设计的协议。只用遵循了IP协议就可以与因特网互联互通。

TCP与UDP的区别



两台主机间的数据传递



网络编程的三大要素

1.IP地址：网络中设备的标识

2.端口号：用于标识进程的逻辑地址，不同进程标识

3.传输协议：通讯的规则

TCP程序开发

- **TCP通信经常使用在客户/服务器编程模型（C/S模型）。**
- **TCP程序开发步骤**

Socket 编程

套接字使用TCP提供了两台计算机之间的通信机制。 客户端程序创建一个套接字，并尝试连接服务器的套接字。

当连接建立时，服务器会创建一个Socket对象。客户端和服务器现在可以通过对Socket对象的写入和读取来进行进行通信。

java.net.Socket类代表一个套接字，并且java.net.ServerSocket类为服务器程序提供了一种来监听客户端，并与他们建立连接的机制。

以下步骤在两台计算机之间使用套接字建立TCP连接时会出现：

(1)服务器实例化一个ServerSocket对象，表示通过服务器上的端口通信。

(2)服务器调用 ServerSocket类 的accept（ ）方法，该方法将一直等待，直到客户端连接到服务器上给定的端口。

(3)服务器正在等待时，一个客户端实例化一个Socket对象，指定服务器名称和端口号来请求连接。

(4)Socket类的构造函数试图将客户端连接到指定的服务器和端口号。如果通信被建立，则在客户端创建一个Socket对象能够与服务器进行通信。

(5)在服务器端，accept()方法返回服务器上一个新的socket引用，该socket连接到客户端的socket。

连接建立后，通过使用I/O流在进行通信。每一个socket都有一个输出流和一个输入流。客户端的输出流连接到()(6)服务器端的输入流，而客户端的输入流连接到服务器端的输出流。

TCP是一个双向的通信协议，因此数据可以通过两个数据流在同一时间发送.以下是一些类提供的一套完整的有用的方法来实现sockets。

- **ServerSocket 类的方法**
  服务器应用程序通过使用java.net.ServerSocket类以获取一个端口,并且侦听客户端请求。

- **Socket 类的方法**
  java.net.Socket类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个Socket对象通过实例化 ，而 服务器获得一个Socket对象则通过accept()方法的返回值。
  构造方法：

```
public Socket(InetAddress host, int port) throws IOException
创建一个流套接字并将其连接到指定 IP 地址的指定端口号
```

- **Socket 客户端实例**
  如下的GreetingClient 是一个客户端程序，该程序通过socket连接到服务器并发送一个请求，然后等待一个响应。

```
// 文件名 GreetingClient.java

import java.net.*;
import java.io.*;

public class GreetingClient
{
   public static void main(String [] args)
   {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try
      {
         System.out.println("Connecting to " + serverName
                            + " on port " + port);
         Socket client = new Socket(serverName, port);
         System.out.println("Just connected to "
                     + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out =
                      new DataOutputStream(outToServer);

         out.writeUTF("Hello from "
                     + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in =
                       new DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF());
         client.close();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

- **Socket 服务端实例**
  如下的GreetingServer 程序是一个服务器端应用程序，使用Socket来监听一个指定的端口。

```
// 文件名 GreetingServer.java

import java.net.*;
import java.io.*;

public class GreetingServer extends Thread
{
   private ServerSocket serverSocket;

   public GreetingServer(int port) throws IOException
   {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }

   public void run()
   {
      while(true)
      {
         try
         {
            System.out.println("Waiting for client on port " +
            serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();
            System.out.println("Just connected to "
                  + server.getRemoteSocketAddress());
            DataInputStream in =
                  new DataInputStream(server.getInputStream());
            System.out.println(in.readUTF());
            DataOutputStream out =
                 new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to "
              + server.getLocalSocketAddress() + "\nGoodbye!");
            server.close();
         }catch(SocketTimeoutException s)
         {
            System.out.println("Socket timed out!");
            break;
         }catch(IOException e)
         {
            e.printStackTrace();
            break;
         }
      }
   }
   public static void main(String [] args)
   {
      int port = Integer.parseInt(args[0]);
```

```
        try
        {
            Thread t = new GreetingServer(port);
            t.start();
        }catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

编译以上 java 代码，并执行以下命令来启动服务，使用端口号为 6066：

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

像下面一样开启客户端：

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

UDP程序开发步骤

UDP程序开发示例(以DatagramSocket实现短信的发送接收)

- **发送方代码：**

```
public class UdpSender{
    public void static main(String []args) throws IOException{
        //创建DatagramSocket发送端
        DatagramSocket sender=new DatagramSocket();
        String senStr="hello,I send you a text!"
        byte[] sendBuf;
        sendBuf=senStr.getByte();
        InteAddress addr=InetAddress.getByName("182.168.1.1");
        int port=5050;
        //创建data数据包
        DatagramPacket sendPacket=new
DatagramPacket(sendBuf,sendBuf.length,addr,port);
        sender.send(sendPacket);//发送数据包
        System.out.println("已发送");
        sender.close（）；
    }
}
```

- **接受代码：**

```
public class UdpReceiver{
    public void static main(String []args) throws IOException{
        //创建DatagramSocket并监听5050端口
        DatagramSocket receiver=new DatagramSocket(5050);
        byte[] recvBuf=new byte[100];
        //创建接受的数据包
        DatagramPacket recPacket=new
DatagramPacket(recvBuf,recvBuf.length);
        //接受客户端消息
        receiver.rfeceive(recvpacket);
        String recvStr=new
String(recvPacket.getData(),0,recvPacket.getLength());
        System.out.println(recvStr);
        receiver.cloase();
}
```

HttpURLConnection

HttpURLConnection实例可用于生成单个请求。作用是通过HTTP协议向服务器发送请求，并可以霍去病服务器返回的数据

- **HttpURIConnection使用步骤**

- **HttpURLConnection实例（下载网页）**

```
public class CreateBaiduHtml{
    public void static main(String []args) throws IOException{
    File file=new File(D:/baidu.html);
    URL url=new URL("http://www.baidu.com");
    HttpURLConnection conn=(HttpURLConnection
)url.openConnection();
    conn.setConnectTime(5000);//设置超时
    conn.setRequestMethod("GET");//设置请求方式
    if(conn.getResponseCode()==200){
        InputStream in=conn.getInputStream();//获取到读取流
        //创建输出流
        BufferedOutputStream fosOut=new  BufferedOutputStream(new
FileOutPutStream(file));
        byte []buffer=new byte[1024];
        int len=0;
        while((len=in.read(buffer))!=-1){
            fosOut.Write(buffer,0,len);
        }
        fosOut.close();
        in.close();
        }
    }
}
```

序列化与反序列化

1.序列化是将对象转变成二进制数据流便于传输，使其保存到本地文件、数据库、网络流等。这种传输可以是程序内的也可以是程序间的，而Android的Parcelable的设计初衷是因为Serializable效率过慢，为了在程序内不同组件间以及不同Android程序间(AIDL)高效的传输数据而设计，这些数据仅在内存中存在，Parcelable是通过IBinder通信的消息的载体。
2.实现序列化有Parcelable和Serializable两种方式
3.Parcelable的性能比Serializable好，在内存开销方面较小，所以在内存间数据传输时推荐使用Parcelable，如activity间传输数据
4.编程实现：Serializable实现较简单，而Parceables实现较负责对于Serializable，类只需要实现Serializable接口，并提供一个序列化版本id(serialVersionUID)即可。而Parcelable则需要实现writeToParcel、describeContents函数以及静态的CREATOR变量，实际上就是将如何打包和解包的工作自己来定义，而序列化的这些操作完全由底层实现。

Serializable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;
}
```

Parcelable序列化示例代码

- **我们将要传送的对象的类实现Parcelable接口。**

```
package com.zeph.android.Parcelable;

import android.os.Parcel;
import android.os.Parcelable;

public class BenParcelable implements Parcelable {

    public String name;

    public int age;

    public String profession;

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

    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;
    }

    public String getprofession() {
        return profession;
    }

    public void setprofession(String profession) {
        this.profession = profession;
    }

    //重写describeContents方法，内容接口描述，默认返回0就可以，好像然并卵
    @Override
    public int describeContents() {
        return 0;
```

```
    }
    //重写writeToParcel方法，将你的对象序列化为一个Parcel对象，即：将类的数据写入外部提供的Parcel中
    @Override
    public void writeToParcel(Parcel parcel, int flag) {
        parcel.writeString(name);
        parcel.writeInt(age);
        parcel.writeString(profession);
    }
    //实例化静态内部对象CREATOR实现接口Parcelable.Creator
    public static final Creator<BenParcelable> CREATOR = new Creator<BenParcelable>() {
        public BenParcelable createFromParcel(Parcel in) {
            return new BenParcelable(in);
        }

        public BenParcelable[] newArray(int size) {
            return new BenParcelable[size];
        }
    };

    private BenParcelable(Parcel in) {
        name = in.readString();
        age = in.readInt();
        profession = in.readString();
    }
}
```

- **ParcelableActivity类，传递对象的Activity类**

```
package com.zeph.android.Parcelable;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class ParcelableActivity extends Activity {

    private Button myButton;

    @Override
    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        myButton = (Button) findViewById(R.id.myButton);

        myButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View arg0) {

                BenParcelable benParcelable = new
BenParcelable("BenZeph", 23,
                        "Java/Android Engineer");

                Intent intent = new Intent();

                intent.setClass(getApplicationContext(),
                        GetParcelableActivity.class);

                Bundle bundle = new Bundle();

                bundle.putParcelable("Ben", benParcelable); //添加
序列化数据

                intent.putExtras(bundle);

                startActivity(intent);
            }
        });
    }
}
```

- **GetParcelableActivity类，接收序列化对象的Activity类**

```
package com.zeph.android.Parcelable;

import android.app.Activity;
import android.os.Bundle;

public class GetParcelableActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        BenParcelable parcelable =
getIntent().getParcelableExtra("Ben");//返回序列化对象数据
System.out.println(parcelable.getName());
System.out.println(parcelable.getAge());
System.out.println(parcelable.getprofession());
    }
}
```

设计模式（常见13种）

详解

福利

Android App优化