目 录

[面向对象基本 8](#_Toc492405880)

[一、面向对象 8](#_Toc492405881)

[1、定义 8](#_Toc492405882)

[二、类和对象 8](#_Toc492405883)

[1、类描述了一类对象的基本属性和行为特征 8](#_Toc492405884)

[3、定义类 9](#_Toc492405885)

[5、定义属性 9](#_Toc492405886)

[6、定义方法 9](#_Toc492405887)

[7、构造器 9](#_Toc492405888)

[8、使用类 9](#_Toc492405889)

[9、dubag调试 9](#_Toc492405890)

[10、对象和引用 9](#_Toc492405891)

[三、方法 10](#_Toc492405892)

[1、方法的属性： 10](#_Toc492405893)

[2、方法的参数传递机制 10](#_Toc492405894)

[3、形参长度可变， 10](#_Toc492405895)

[4、方法的重载 10](#_Toc492405896)

[四、成员变量和局部变量 10](#_Toc492405897)

[1、定义位置不同，不在任何方法中的变量是全局的，定义在某一个类的方法中的变量是局部的 10](#_Toc492405898)

[2、成员变量初始化和在内存中运行的机制 10](#_Toc492405899)

[3、局部变量的初试化和在内存中的运行机制 10](#_Toc492405900)

[5、用来描述整个类或对象的固有信息 11](#_Toc492405901)

[6、多个方法共享 11](#_Toc492405902)

[五、封装 11](#_Toc492405903)

[1、理解封装 11](#_Toc492405904)

[2、private类 default包 protected子类 public公共 11](#_Toc492405905)

[3、package和import 12](#_Toc492405906)

[4、java常用包 13](#_Toc492405907)

[六、深入构造器 13](#_Toc492405908)

[1、使用构造器执行初试化 13](#_Toc492405909)

[2、构造器重载 13](#_Toc492405910)

[七、类的继承 13](#_Toc492405911)

[2、单继承。 13](#_Toc492405912)

[4、继承语法格式： 14](#_Toc492405913)

[5、重写父类的方法 14](#_Toc492405914)

[八、多态 14](#_Toc492405915)

[引用类型强制转换 15](#_Toc492405916)

[九、基本数据类型的包装类 15](#_Toc492405917)

[1、8中基本数据类型，提供类包装类，因为基础数据类型不是对象 15](#_Toc492405918)

[2、byte，short，int，long，double，boolean、float、char 15](#_Toc492405919)

[3、基本类型转化为包装类型 15](#_Toc492405920)

[4、jdk1.5自动装箱，自动开箱，可以吧基本类型赋给包装类 15](#_Toc492405921)

[5、包装类和字符串之间的转换： 15](#_Toc492405922)

[类和对象的处理 15](#_Toc492405923)

[一、处理对象 15](#_Toc492405924)

[1、打印对象和toString（）方法 16](#_Toc492405925)

[2、==和equals方法比较运算符 16](#_Toc492405926)

[二、类成员 16](#_Toc492405927)

[1、通过对象来调用类的成员，实际上是委托类来访问类的成员 16](#_Toc492405928)

[三、单例类 16](#_Toc492405929)

[1、单例类只能有一个实例。 17](#_Toc492405930)

[2、单例类必须自己自己创建自己的唯一实例。 17](#_Toc492405931)

[3、单例类必须给所有其他对象提供这一实例。 17](#_Toc492405932)

[四、final修饰符 20](#_Toc492405933)

[1、final修饰的变量，表示变量获得初始值后就不能改变 20](#_Toc492405934)

[2、final方法 20](#_Toc492405935)

[3、final类，final修饰的类没有子类。 20](#_Toc492405936)

[五、抽象类 21](#_Toc492405937)

[1、抽象类的作用 21](#_Toc492405938)

[六、接口 25](#_Toc492405939)

[1、接口不能创建实例，可以用来声明引用性的变量，这个引用型的变量必须引用到其他实现类的对象，另外一个用途被其实现类实现 25](#_Toc492405940)

[2、接口的定义 25](#_Toc492405941)

[3、接口的继承 28](#_Toc492405942)

[4、类实现接口规范： 28](#_Toc492405943)

[5、接口和抽象类的共同点和差别 28](#_Toc492405944)

[七、内部类 30](#_Toc492405945)

[1、把一个类放在其他类中定义，这个定义在其他类内部的类叫内部类。 31](#_Toc492405946)

[2、内部类的作用：能提供更好的封装性，内部类成员可以直接访问外部类的私有数据 31](#_Toc492405947)

[3、非静态内部类，静态内部类，局部内部类，匿名内部类 31](#_Toc492405948)

[八、枚举类 32](#_Toc492405949)

[1、一个类的对象有限且是固定的，叫枚举类 32](#_Toc492405950)

[2、实现枚举类 32](#_Toc492405951)

[3、示例 33](#_Toc492405952)

[4：jdk1.5以后增加了etum关键字，用于定义枚举类，一个java源文件只能定义一个public访问权限的枚举类，Java源文件名和该枚举类名一致 34](#_Toc492405953)

[九、对象垃圾回收 36](#_Toc492405954)

[1、垃圾回收机制的特征 36](#_Toc492405955)

[集合 37](#_Toc492405956)

[一、java集合概述 37](#_Toc492405957)

[二、Collection和Iterator接口 37](#_Toc492405958)

[1、  Collection操作 37](#_Toc492405959)

[2、  使用Iterator接口遍历集合元素 37](#_Toc492405960)

[三、set接口 37](#_Toc492405961)

[1、hashset类，treeset类，enumset类和collection基本相同，不允许添加重复元素。 37](#_Toc492405962)

[四、list接口 37](#_Toc492405963)

[1、list集合是一个有序的集合，集合中每个元素都有其对应的顺序索引。 37](#_Toc492405964)

[2、list接口的使用 37](#_Toc492405965)

[3、ArrayList和Vector实现类 37](#_Toc492405966)

[4、固定长度的List 38](#_Toc492405967)

[五、Queue接口 38](#_Toc492405968)

[1、Queue接口模拟接口这钟数据结构（先进先出）只能操作队首，或者队尾的元素 38](#_Toc492405969)

[2、LinkedList实现类 38](#_Toc492405970)

[3、几种集合性能比较 38](#_Toc492405971)

[六、map接口 38](#_Toc492405972)

[1、map是保存基友映射关系的数据，数据项一键值对存在，键key，值value，都可以是任意类型的数据key不允许重复， 38](#_Toc492405973)

[2、key和value是一一对应的，keySet（）会返回所有key的集合，所有value放在一起类似一个list 39](#_Toc492405974)

[3、常用函数，见api 39](#_Toc492405975)

[4、HashMap和HashTable 39](#_Toc492405976)

[七、操作类集合的工具类：collection 42](#_Toc492405977)

[1、将集合对象设置为不可变，对集合对象实现同步控制，并且该工具类提供了很多对集合排序、查询、修改的操作 42](#_Toc492405978)

[2、同步控制 43](#_Toc492405979)

[Java基础类库 43](#_Toc492405980)

[一、与用户互动 43](#_Toc492405981)

[1、运行程序的参数 43](#_Toc492405982)

[2、使用scanner获取键盘输入 43](#_Toc492405983)

[二、系统相关类 43](#_Toc492405984)

[1、System类 43](#_Toc492405985)

[2、Runtime类和java程序运行环境相关的类，每个java程序对应一个Runtime实例，应用程序通过对象与其运行时环境相关联 43](#_Toc492405986)

[三、常用类 46](#_Toc492405987)

[1、object类 46](#_Toc492405988)

[2、String类，stringBuffer和StringBuilder 46](#_Toc492405989)

[3、Math类 47](#_Toc492405990)

[4、BigDecimal类 48](#_Toc492405991)

[四、处理日期的类 48](#_Toc492405992)

[1、Date类 48](#_Toc492405993)

[2、Calender 48](#_Toc492405994)

[3、time包 48](#_Toc492405995)

[泛型 60](#_Toc492405996)

[一、泛型入门 60](#_Toc492405997)

[1、集合编译时不检查类型的异常 60](#_Toc492405998)

[2、使用泛型 60](#_Toc492405999)

[3、java7泛型的菱形语法 60](#_Toc492406000)

[二、深入泛型 60](#_Toc492406001)

[1、定义泛型接口、类 60](#_Toc492406002)

[3、从泛型类派生子类 62](#_Toc492406003)

[4、并不存在的泛型类 62](#_Toc492406004)

[三、类型通配符 62](#_Toc492406005)

[1、使用类型通配符 63](#_Toc492406006)

[2、设定类型通配符的上限 63](#_Toc492406007)

[3、类型通配符的下限：<?Super Type>,表示type本身或者type的父类 63](#_Toc492406008)

[四、泛型方法 64](#_Toc492406009)

[1、定义泛型方法 64](#_Toc492406010)

[2、泛型方法和类型通配符的区别和联系 64](#_Toc492406011)

[3、java7的菱形语法和泛构造器 65](#_Toc492406012)

[五、集合的擦除和转换 66](#_Toc492406013)

[1、在严格的泛型代码中，带泛型声明的类总应该带有类型参数，但为了和老版java保持一致也允许在使用带有泛型声明的类不指定类参数，会默认为原始的类型参数（声明该类型参数时的第一个上限参数） 66](#_Toc492406014)

[2、当把一个具有泛型信息的对象赋给一个没有泛型信息的对象时，所有<>内的类型信息都会被扔掉，一个List<String>类型被转换成List，则该List对集合元素的类型转变为类型参数的上限（object）这种情况叫做类型的擦除 66](#_Toc492406015)

[3、对泛型而言，可以直接把一个List对象赋给一个List<String>,编译器仅仅只提”未经检测的转换” 66](#_Toc492406016)

[异常处理 66](#_Toc492406017)

[一、异常概述 67](#_Toc492406018)

[二、异常处理机制 67](#_Toc492406019)

[1、Try{}catch（）{} 67](#_Toc492406020)

[2、异常类的继承体系 67](#_Toc492406021)

[3、访问异常信息 68](#_Toc492406022)

[4、使用finally回收资源 68](#_Toc492406023)

[5、异常处理嵌套 68](#_Toc492406024)

[三、Checked异常和Runtime异常 68](#_Toc492406025)

[2、Checked异常必须被处理，否则无法编译 68](#_Toc492406026)

[3、checked异常处理方式 69](#_Toc492406027)

[4、使用throws声明抛出异常 69](#_Toc492406028)

[四、使用throw声明抛出异常 69](#_Toc492406029)

[五、异常的跟踪栈 69](#_Toc492406030)

[六、异常处理的规则 69](#_Toc492406031)

[1、不要过度使用异常 69](#_Toc492406032)

[2、不要使用过于庞大的try 69](#_Toc492406033)

[3、避免使用catcha all 69](#_Toc492406034)

[AWT编程 69](#_Toc492406035)

[一、GUI（图形用户界面）和AWT （抽象窗口工具集） 69](#_Toc492406036)

[二、AWT容器 70](#_Toc492406037)

[2、Awt主要提供了两种主要的容器： 70](#_Toc492406038)

[3、 71](#_Toc492406039)

[三、布局管理器 71](#_Toc492406040)

[四、awt常用组件 73](#_Toc492406041)

[2、对话框 74](#_Toc492406042)

[五、事件处理 76](#_Toc492406043)

[1、事件模型的流程 76](#_Toc492406044)

[2、事件和事件监听器 76](#_Toc492406045)

[3、事件适配器 78](#_Toc492406046)

[六、AWT菜单 79](#_Toc492406047)

[分为:菜单条，菜单，菜单项。 79](#_Toc492406048)

[Swing 83](#_Toc492406049)

[一、Swing概述 83](#_Toc492406050)

[二、Swing基本组件 84](#_Toc492406051)

[三、Swing组件分类 84](#_Toc492406052)

[四、Swing组件的特别之处 84](#_Toc492406053)

[五、Applet基础知识 91](#_Toc492406054)

[IO 93](#_Toc492406055)

[二、File类 93](#_Toc492406056)

[1、访问文件和目录 93](#_Toc492406057)

[三、理解java的IO流 94](#_Toc492406058)

[1、流的分类 94](#_Toc492406059)

[2、流的概念模型 95](#_Toc492406060)

[3、字节流和字符流 95](#_Toc492406061)

[3、处理流的用法 96](#_Toc492406062)

[4、输入、输出流体系 96](#_Toc492406063)

[四、重定向标准输入/输出 97](#_Toc492406064)

[System类提供了三个重定向的方法： 97](#_Toc492406065)

[五、RandomAcessFile 103](#_Toc492406066)

[两个方法操作文件指针： 103](#_Toc492406067)

[两个构造器： 103](#_Toc492406068)

[六、对象序列化的含义和意义 105](#_Toc492406069)

[1、使用对象类实现序列化 105](#_Toc492406070)

[2、对象引用的序列化 105](#_Toc492406071)

[3、自定义序列化 105](#_Toc492406072)

[七、NIO 107](#_Toc492406073)

[1、java新IO概述 107](#_Toc492406074)

[2、使用Buffer 108](#_Toc492406075)

[3、使用Channel 109](#_Toc492406076)

[4、字符集和Charset 111](#_Toc492406077)

[5、文件锁 112](#_Toc492406078)

[八、NIO2 113](#_Toc492406079)

[1、Path、Paths和Files 113](#_Toc492406080)

[2、使用FileVisitor遍历文件和目录 113](#_Toc492406081)

[3、watchservice监控文件变化 115](#_Toc492406082)

[4、访问文件的属性 116](#_Toc492406083)

[多线程 116](#_Toc492406084)

[一、线程的概述 116](#_Toc492406085)

[1、进程和线程 116](#_Toc492406086)

[2、多线程的优势 117](#_Toc492406087)

[二 117](#_Toc492406088)

[1、继承Tread类创建线程 117](#_Toc492406089)

[2、示例 117](#_Toc492406090)

[3、实现Runnable接口创建线程类 118](#_Toc492406091)

[4、两种方式创建线程的对比 118](#_Toc492406092)

[三、线程的生命周期 119](#_Toc492406093)

[四、控制线程 120](#_Toc492406094)

**面向对象基本**

**一、面向对象**

1、定义

从客观世界存在的实际对象出发，来构造应用系统，特点封装、继承、多态

2、早期为结构化语言，随着技术发展逐步演化为面向对象语言，面向对象语言具有可读性，可扩展，可维护灯优势

3、类是组成程序的基本单位，面向对象的语言始终是围绕类、类层、数据类、继承多态进行的

4、Java语言提供了定义类和属性，方法的基本功能

5、类自己定义的数据类型，类是描述客观世界的某一类对象的共同特征；类是对象的抽象，对象是类的具体存在；

**二、类和对象**

1、类描述了一类对象的基本属性和行为特征

例如电脑的属性 cpu内存，硬盘，分辨率，价格等

2、行为特征：计算，打游戏等、

3、定义类

[修饰符]class类名{

属性：

方法（函数）：

构造函数：}

4、类名命名规则：（1）有意义（2）每个单词首字母大写（3）单词不能用分隔符（4）以字母开头，容许出现数字

5、定义属性

[修饰符]属性类型 属性名[=默认值]

修饰符，无 public private protected static final

属性类型：基本的数据类型和引用的数据类型

6、定义方法

{ 修饰符}返回类型 方法名 (形参列表){

可执行语句}

7、构造器

[修饰符]构造器名称（形参列表）{}

名称与类名一致

8、使用类

通过new实例化类

9、dubag调试

10、对象和引用

P引用变量，指向实际的Person对象，垃圾回收，对一个对象赋值null；

**三、方法**

1、方法的属性：

方法是类和对象行为特征的抽象，是类和对象最重要的组成部分

方法不能单独存在，必须包含在类中

Static修饰的方法叫类方法

方法只能通过类或对象来调用，其中类的方法，类，方法，对象，方发

2、方法的参数传递机制

在定义方法时，括号里面的参数叫形参列表。

调用方式，括号里值，叫实参。

 Java中实参值得传递，是将实参的副本传到方法内，参数本身不受影响。

3、形参长度可变，

最后一个形参增加…，表明这个形参可以接受多个参数，多个值当数组传入

4、方法的重载

一个类中方法名一样，参数不同，就叫方法的重载，返回值和修饰符对方法重载没有影响

**四、成员变量和局部变量**

1、定义位置不同，不在任何方法中的变量是全局的，定义在某一个类的方法中的变量是局部的

static类成员变量赋值时将整个类的值改变了

2、成员变量初始化和在内存中运行的机制

3、局部变量的初试化和在内存中的运行机制

    局部变量定义后必须经过显示的初始化后，才能使用。

    局部变量不属于任何类和实例。保存在栈内存中。

    局部变量的作用域，从初始化局部变量开始，到方法结束或代码块结束。

4、变量的使用规则

（1）、用于描述整个类或对象的固有信息。人的身高、体重、名称。眼睛、胳膊、腿，类的成员变量。

      （2）、如果某个类需要一个变量保存整个类或实例的运行状态。

      （3）、多个方法共享。

5、用来描述整个类或对象的固有信息

如果某个类需要保持整个类或实例的运行状态

6、多个方法共享

**五、封装**

1、理解封装

将类和对象的内部实现细节隐藏，不允许外部直接访问对象的内部信息，而是通过提供的方法访问

目的（1）隐藏类的实现细节（2）限制对属性的不合理访问（3）检测数据（4）便于修改提高代码的可维护程度

2、private类 default包 protected子类 public公共

1）一般情况下，类的绝大部分属性都应该使用private，除static、全局变量除外。

  （2）如果定义的类、方法、构造器，希望被别人访问，定义为public



3、package和import

相同功能类放到同一package，便于维护和管理。

一个类的第一行，指定包名称。

使用类时，要用包名+类。

通过import引入一个包。

选择具体要引入的类或接口。

4、java常用包

java核心包都放在java包及子包下，java的扩张类放在javax包及子包下。

这些包里的实用类就是java的api接口。sun按功能把不同类放到了不同的包下。

（1）、java.lang:核心语言包，String、Math、System、Thread。使用此包下的类，无需import。

（2）、java.util:主要包括工具类、接口，集合、框架类、接口。List、Map、Set、ArrayList、HashMap。

（3）、java.net:包括和网络编程相关的类和接口。Socket、URL

（4）、java.io:输入、输出相关的类和接口。InputStream、OutputStream、Reader、Writer。

（5）、java.text:格式化相关的类。DateFormat

（6）、java.sql:jdbc数据库编程有关的类和接口。Connection、Driver

（7）、java.awt:构建图像界面程序类和接口。

（8）、java.swing:图像界面程序类和接口。

**六、深入构造器**

1、使用构造器执行初试化

构造器最大的作用就是在创建对象时初始化，创建对象时，对属性初试化，基本数字初始化值为0，boolean设置为false,引用类型设置为null。

可以使用构造器，显示指定初始化值。

如果没有为程序编写一个构造器，系统会默认提供一个无参数默认构造器。

2、构造器重载

构造器名和类名一样，参数不同。

**七、类的继承**

1、继承是实现软件复用的重要手段。

2、单继承。

      3、继承的特点：extends，英文原意：扩展。

4、继承语法格式：

          修饰符 class 子类名称 extends 父类名称

{

}

Object是所有类的父类。

5、重写父类的方法

           （1）、一般情况下，子类基于父类，扩展属性和方法。另外，重写父类的方法。

           （2）、方法重写规则

                       A、“两同两小一大”：两同：方法名相同、形参相同。两小：子类的返回值更小或相等，抛出的异常类更小或相等。一大：子类方法的访问权限比父类更大或相等。

                      B、调用父类被覆盖的方法，spuer（实例方法）或父类名（类方法）。

           （3）父类实例的super引用

                 A、super是直接父类对象的默认引用。

                 B、如果在构造器中使用this，this指向正在初试化的对象，super指向父类对象。

           （4）重载和重写overload、override

重载：同一个类的多个同名方法之间。

重写：子类和父类的同名方法之间。

父类方法和子类方法之间也可以发生重载。

父类的属性和子类的属性，也可以发生重写。

           （5）调用父类的构造函数

Super调用父类的构造器。

子类对象在初始化时，总是先执行父类的构造器，再执行子类的构造器。

**八、多态**

java的引用变量有两种类型：编译时的类型，运行时的类型

如果编译时的类型和运行时的类型不一致，就叫多态。

编译时调用父类的方法，实际执行的是子类的方法，或者是父类表现出来的行为特性不像父类，像子类，这种现象叫多态

属性不具有多态性

优势、能够把多种类型的子类对象赋给同一种类型的应用变量，劣势、有些子类的函数和类型访问不了，会造成函数、属性的丢失。

引用类型强制转换

1)、基本类型转换只能在数字之间。

2）、引用类型的转换只能把一个父类转换成子类。没有关系的类之间不能进行转换，否则会发生编译错误

3）转换符（）

4）instanceof判断类形

**九、基本数据类型的包装类**

1、8中基本数据类型，提供类包装类，因为基础数据类型不是对象

2、byte，short，int，long，double，boolean、float、char

Byte，Short、Integer、Long、Character、Float、Boolean、Double

3、基本类型转化为包装类型

4、jdk1.5自动装箱，自动开箱，可以把基本类型赋给包装类

5、包装类和字符串之间的转换：

包装类的静态方法parseXxx（String s）字符串转化为基本类型

String类valueOf（）基本类型转化为字符串

+基本类型转换

**类和对象的处理**

**一、处理对象**

Object类是所有类的父类，该类定义的方法提供了处理java对象的基本方法。

1、打印对象和toString（）方法

2、==和equals方法比较运算符

**二、类成员**

Static关键字修饰的成员就是类成员。类属性、类方法、静态初始化块

Static修饰的成员属于类

Static不能修饰构造器

1、通过对象来调用类的成员，实际上是委托类来访问类的成员

**public class Acc {**

**private static void test()**

**{**

**System.out.println("静态方法我是");**

**}**

**private void info(){**

**System.out.println("普通的方法");**

**}**

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

**/\*test();**

**Acc p=new Acc();**

**p.info();\*/**

**Acc a=null;**

**a.test();**

**a.info();**

**}**

**}**

**三、单例类**

1. 构造器使用private修饰以达到隐藏构造器的目的，不允许任意创建类的实例，一个类始终只能创建一个实例，这个类就叫做单例类。
2. 因为构造器隐藏起来了，就提供一个public作为累的访问点，这个方法必须是static修饰的静态方法

1、单例类只能有一个实例。

2、单例类必须自己自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

**public class Singl {**

**private static Singl instance;**

**private Singl(){**

**}**

**public static Singl getInstence(){**

**if(instance==null){**

**instance=new Singl();//实例化这个对象**

**}**

**return instance;**

**}**

**}**

**public class TestSingl {**

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

**Singl s1=Singl.getInstence();//调用这个方法，肯定会实例化一个对象，可以用这个唯一的对象调用其他函数。**

**Singl s2=Singl.getInstence();**

**System.out.println(s1==s2);**

**}**

**}**

流量统计

**package xxx;**

**public class WebJsQ {**

**private static WebJsQ instence;**

**private int count;**

**public WebJsQ() {**

**// TODO Auto-generated constructor stub**

**}**

**public static WebJsQ getInstence(){**

**if (instence==null){**

**instence=new WebJsQ();**

**}return instence;**

**}**

**public void visted(){**

**System.out.println("当前访问人数"+ ++count);**

**}**

**public int getCount() {**

**return count;**

**}**

**public void setCount(int count) {**

**this.count = count;**

**}**

**}**

**package xxx;**

**public class WebJsQTest {**

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

**WebJsQ man1=WebJsQ.getInstence();**

**man1.visted();**

**man1.visted();**

**WebJsQ man2=WebJsQ.getInstence();**

**man2.visted();**

**man1.visted();**

**WebJsQ man3=WebJsQ.getInstence();**

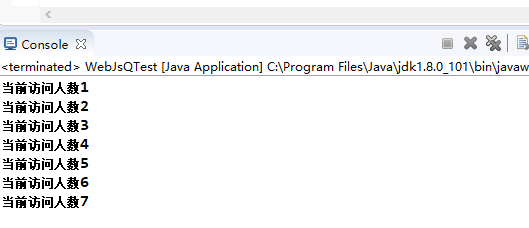
**man3.visted();**

**man2.visted();**

**man2.visted();**

**}**

**}**



打印机

**package xxx;**

**public class Dayin {**

**private static Dayin p;**

**private boolean free=true;**

**public Dayin() {**

**// TODO Auto-generated constructor stub**

**}**

**public static Dayin getIndtance()**

**{**

**if(p==null){**

**p=new Dayin();**

**}return p;**

**}**

**public void print(String s){**

**if(free){**

**free=false;**

**System.out.println("打印内容"+s);**

**free=true;**

**}else{**

**System.out.println("打印机在被别人使用");**

**}**

**}**

**public boolean isFree(){**

**return free;**

**}**

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

**Dayin d1=Dayin.getIndtance();**

**d1.print("打印内容1.0");**

**d1.print("打印内容1.1");**

**Dayin d2=Dayin.getIndtance();**

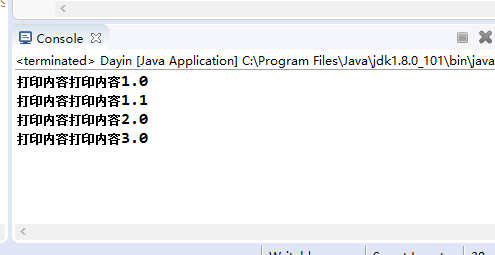
**Dayin d3=Dayin.getIndtance();**

**d2.print("打印内容2.0");**

**d3.print("打印内容3.0");**

**}**

**}**



**四、final修饰符**

final修饰的类，变量，方法不可变

1、final修饰的变量，表示变量获得初始值后就不能改变

1. 修饰成员变量

类变量：声明成员变量，在静态域初始化块

实例变量：声明变量、非静态初始化块，构造器

1. final修饰局部变量，系统不会对局部变量初始化，由程序员显示初始化，由final修饰的局部变量只能被赋值一次
2. final修饰的基本变量和引用变量的区别，

基本变量不能重复赋值，变量不能改变

引用变量，它仅仅是一个引用，final只保证这个引用地址不变，即一直引用同一个对象，但是对象是可以改变的。

2、final方法

final修饰的方法不能被重写，不希望子类重写父类的方法。Object中的getCless方法不能被重写，

private 修饰的方法在子类不可见，子类无法重写，即使在子类定义一个方法，方法名，参数，返回值都相同的方法，也不是重写父类的方法，而是重新定义了一个新方法。

3、final类，final修饰的类没有子类。

**public class TestFinalVar {**

**private final int a=6;**

**private final String str;**

**private final int c;**

**final static double d;**

**final static double h;**

**static**

**{**

**d=35.424;h=21.890;**

**}**

**{**

**str="hello";**

**//非静态初始化块**

**}**

**public TestFinalVar(){**

**c=23;**

**}**

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

**TestFinalVar r=new TestFinalVar();**

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

**System.out.println(r.c);**

**System.out.println(r.d);**

**System.out.println(r.h);**

**System.out.println(r.str);**

**final String tr="hello";**

**final int i;**

**i=1;**

**}**

**五、抽象类**

abstract修饰抽象类，使用abstract修饰的方法称为抽象方法，抽象方法只有方法头没有具体实现。在一个类里如果存在抽象方法，这个类必须被定义成抽象类，一个抽象类可以没有抽象方法

1. 抽象类和抽象方法必须使用abstract修饰，抽象方法没有方法体
2. 抽象类不能被实例化，即使抽象类没有抽象方法
3. 抽象类中可以包括属性，方法，构造器，初始化块，内部类，枚举类，不能创建实例，只能被子类调用
4. 含有抽象方法（直接定义的抽象方法或从父类继承的抽象方法，并且抽象方法没有完全实现，继承了接口并且没有完全实现接口的抽象方法）的类只能被定义为抽象类
5. 定义抽象方法，就是在普通方法上加上abstract，去掉{}
6. 定义抽象类，在普通类上加abstract

1、抽象类的作用

抽象类是子类更高程度的抽象是子类的模板，避免了子类设计的随意性。体现了模板设计模式的思想

**public class SDBcer extends SDB {**

**/\*\***

**\***

**\*/**

**public SDBcer() {**

**// TODO Auto-generated constructor stub**

**}**

**/\* (non-Javadoc)**

**\* @see SDB#getRadius()**

**\*/**

**@Override**

**public double getRadius() {**

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

**return 3;**

**}**

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

**SDB sm=new SDBcer();**

**sm.setTurnRate(15);**

**System.out.println(sm.getSpeed());**

**}**

**}**

**public abstract class SDB {**

**private double turnRate;**

**public abstract double getRadius();**

**public SDB() {**

**// TODO Auto-generated constructor stub**

**}**

**public double getSpeed(){**

**return 2\*this.getRadius()\*Math.PI\*turnRate;**

**}**

**public double getTurnRate() {**

**return turnRate;**

**}**

**public void setTurnRate(double turnRate) {**

**this.turnRate = turnRate;**

**}**

**}**

**public abstract class Testfinal {**

**{**

**System.out.println("执行testfinal的初始化块");**

**}**

**private String color;**

**public abstract double calPerimeter();**

**public abstract String getType();**

**public Testfinal(){**

**}**

**public Testfinal(String color){**

**this.color=color;**

**System.out.println("抽象类被调用");**

**}**

**public String getColor() {**

**return color;**

**}**

**public void setColor(String color) {**

**this.color = color;**

**}**

**}**

**public class Tringls extends Testfinal {**

**private double a;**

**private double b;**

**private double c;**

**public Tringls(String color,double a,double b,double c){**

**super(color);**

**if((a+b>c||a+c>b||b+c>a)&&(a-b<c||b-a<c||b-c<a||c-a<b||a-c<b)){**

**this.a=a;**

**this.b=b;**

**this.c=c;**

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

**return ;**

**}**

**}**

**public double calPerimeter() {**

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

**return a+b+c;**

**}**

**public String getType(){**

**return "三角形";**

**}**

**}**

**public class Circle extends Shape {**

**private double radius;**

**public Circle(String color,double radius){**

**super(color);**

**this.radius = radius;**

**}**

**public double getRadius() {**

**return radius;**

**}**

**public void setRadius(double radius) {**

**this.radius = radius;**

**}**

**/\* (non-Javadoc)**

**\* @see class16.Shape#calPerimeter()**

**\*/**

**@Override**

**public double calPerimeter() {**

**return 2\*radius\*Math.PI;**

**}**

**/\* (non-Javadoc)**

**\* @see class16.Shape#getType()**

**\*/**

**@Override**

**public String getType() {**

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

**return "圆形";**

**}**

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

**Shape s1 = new Triangle("红色",2,3,4);**

**Shape s2 = new Circle("黑色",2);**

**System.out.println(s1.getType());**

**System.out.println(s1.calPerimeter());**

**System.out.println(s2.getType());**

**System.out.println(s2.calPerimeter());**

**}**

**}**

**六、接口**

1、接口不能创建实例，可以用来声明引用性的变量，这个引用型的变量必须引用到其他实现类的对象，另外一个用途被其实现类实现

一个类可以实现多个接口

接口就是模板，体现的思想是规范和分离。

软件系统各模块间，两个系统之间进行设计时。应采用接口设计，这样能降低他们的耦合度，系统能具有更好的扩展性和可维护性

2、接口的定义

1. 关键字interface

[修饰符]interface接口名extends 父接口1，父接口2{常量定义

抽象方法定义}

1. 接口中定义的是多个类的共同行为，因此都是public访问权限，接口定义的常量是接口相关的，只能是public、static、final，没有指定修饰符系统默认是这三个
2. 接口的方法public abstract修饰
3. 示例

**public interface Output {**

**int MAX\_CACHE\_LINE = 50;**

**public void out();**

**void getDate(String msg);**

**}**

**public class TestOutput {**

**/\*\***

**\* @param args**

**\*/**

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

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

**System.out.println(Output.MAX\_CACHE\_LINE);**

**}**

**}**

**public interface InterfaceA {**

**int PROP\_A = 5;**

**public void testA();**

**}**

**public interface InterfaceB {**

**int PROP\_B = 6;**

**public void testB();**

**}**

**public interface InterfaceC extends InterfaceA, InterfaceB {**

**int PROP\_C = 7;**

**public void testC();**

**}**

**public class TestInterface {**

**/\*\***

**\* @param args**

**\*/**

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

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

**System.out.println(InterfaceC.PROP\_A);**

**System.out.println(InterfaceC.PROP\_B);**

**System.out.println(InterfaceC.PROP\_C);**

**}**

**}**

**public interface Product {**

**public int getProductTime();**

**}**

**public class Printer implements Output, Product {**

**private String[] printdata = new String[MAX\_CACHE\_LINE];**

**private int dataNum = 0;**

**/\* (non-Javadoc)**

**\* @see class6.Product#getProductTime()**

**\*/**

**@Override**

**public int getProductTime() {**

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

**return 30;**

**}**

**/\* (non-Javadoc)**

**\* @see class6.Output#out()**

**\*/**

**@Override**

**public void out() {**

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

**while (dataNum > 0) {**

**System.out.print("正在打印"+printdata[0]);**

**System.arraycopy(printdata, 1, printdata, 0, --dataNum);**

**}**

**}**

**/\* (non-Javadoc)**

**\* @see class6.Output#getDate(java.lang.String)**

**\*/**

**@Override**

**public void getDate(String msg) {**

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

**if (dataNum >= MAX\_CACHE\_LINE) {**

**System.out.println("打印队列已满，添加失败");**

**} else {**

**printdata[dataNum++] = msg;**

**}**

**}**

**/\*\***

**\* @param args**

**\*/**

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

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

**Output o = new Printer();**

**o.getDate("西游记");**

**o.getDate("红楼梦");**

**o.out();**

**o.getDate("三国演义");**

**o.getDate("水浒传");**

**o.out();**

**Product p = new Printer();**

**System.out.println(p.getProductTime());**

**}**

**}**

3、接口的继承

接口支持多继承，可以有多个父接口，继承关系和类类似

4、类实现接口规范：

[修饰符]class 类名 继承关键字（extends）父类

implement 接口1，接口2{}

作用，从父类继承了属性，抽象方法等，并要实现所有继承接口的抽象方法。

5、接口和抽象类的共同点和差别

1. 都不能被实例化，只能被继承或实现，并且都包含了抽象方法，被集成的子类要实现抽象方法
2. 差别：主要体现在设计目标上

接口：更多体现一种规范，对于接口的实现者而言，接口规定了实现者必须像外界提供哪些服务，对于接口的调用者，规定了调用者可以调用那些方法以及如何调用服务。程序内部的调用接口，接口就是多模块之间耦合的标准，多程序之间的接口就是程序间的通信标准。接口一旦发布，不能轻易的变更。

抽象类：是多个子类的父类，体现模板设计思想

**package wangleba;**

**public interface Usb {**

**public abstract void insertUsb();**

**}**

**package wangleba;**

**public interface Mouse extends Usb {**

**public abstract void clickReinstall();**

**}**

**package wangleba;**

**public interface Key extends Usb{**

**public abstract void password();**

**}**

**package wangleba;**

**public interface Screen {**

**public abstract void start();**

**public abstract void stop();**

**}**

**package wangleba;**

**public class Computer implements Mouse, Key, Screen {**

**private int usb;**

**public Computer(int usb) {**

**// TODO Auto-generated constructor stub**

**this.usb=usb;**

**}**

**@Override**

**public void insertUsb() {**

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

**System.out.println("请插入usb设备");**

**}**

**@Override**

**public void start() {**

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

**System.out.println("开始");**

**}**

**@Override**

**public void stop() {**

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

**System.out.println("结束");**

**}**

**@Override**

**public void password() {**

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

**System.out.println("从键盘输入");**

**}**

**@Override**

**public void clickReinstall() {**

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

**System.out.println("鼠标点击确定");**

**}**

**public void PANduan(){**

**if(usb==0){**

**start();**

**password();**

**clickReinstall();**

**stop();**

**}else{**

**insertUsb();**

**}**

**}**

**}**

**package wangleba;**

**public class TestComputer {**

**public TestComputer() {**

**// TODO Auto-generated constructor stub**

**}**

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

**Computer q=new Computer(1);**

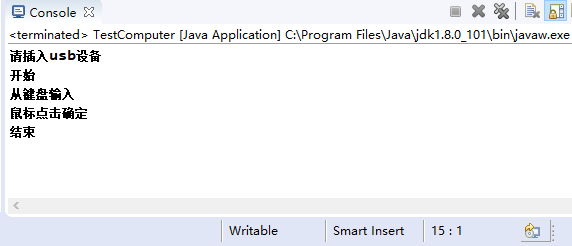
**Computer w=new Computer(0);**

**q.PANduan();**

**w.PANduan();**

**}**

**}**



**七、内部类**

1、把一个类放在其他类中定义，这个定义在其他类内部的类叫内部类。

2、内部类的作用：能提供更好的封装性，内部类成员可以直接访问外部类的私有数据

3、非静态内部类，静态内部类，局部内部类，匿名内部类

**package neibulei;**

**public class Car {**

**private String color;**

**private String form;**

**private int lenth;**

**public Car() {**

**// TODO Auto-generated constructor stub**

**}**

**public Car(String color,String form,int lenth){**

**this.color=color;**

**this.form=form;**

**this.lenth=lenth;**

**}**

**public void test(){**

**Wheel l=new Wheel("黑色",12.2);**

**l.info();**

**}**

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

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

**Car c=new Car("绿色","跑车",4);**

**c.test();**

**}**

**private class Wheel{**

**private final int WHEEL=4;**

**private String Wcolor;**

**private double diameter;**

**public Wheel(){**

**}**

**public Wheel(String Wcolor,double diameter){**

**this.diameter=diameter;**

**this.Wcolor=Wcolor;**

**}**

**public void info(){**

**System.out.println("汽车有"+WHEEL+"个轮子"+"颜色是"+this.Wcolor+",直径是"+this.diameter);**

**Inside i=new Inside("红色",3);**

**i.fun();**

**}**

**}**

**private class Inside{**

**private String color2;**

**private int row;**

**public Inside(){**

**}**

**public Inside(String color2,int row){**

**this.color2=color;**

**this.row=row;**

**}**

**public void fun(){**

**System.out.println("汽车内部的颜色是"+this.color2+"有"+row+"排");**

**System.out.println("汽车外观是"+color+"长"+lenth+"米，的"+form);**

**}**

**}**

**}**

**八、枚举类**

1、一个类的对象有限且是固定的，叫枚举类

2、实现枚举类

设计原则：1）private修饰 隐藏构造器，类所有实现使用public static final 修饰的属性保存。2）提供静态方法，允许其他程序通过静态方法根据特定参数获取与之匹配的实例

3、示例

**package neibulei;**

**public class Year {**

**private final String name;**

**private final String desc;**

**public String getName() {**

**return name;**

**}**

**public String getDesc() {**

**return desc;**

**}**

**private Year(String name,String desc) {**

**// TODO Auto-generated constructor stub**

**this.desc=desc;**

**this.name=name;**

**}**

**public static final Year Spring=new Year("春天","春天天来了，又到了动物交配的时候了");**

**public static final Year Summer=new Year("夏天","热死人了");**

**public static final Year Fall=new Year("秋天","123123");**

**public static final Year Winter=new Year("冬天","冷啊");**

**public static Year getYear(int yearNum){**

**switch(yearNum){**

**case 1:**

**return Spring;**

**case 2:**

**return Summer;**

**case 3:**

**return Fall;**

**case 4:**

**return Winter;**

**default:**

**return null;**

**}**

**}**

**public void TestYear(){**

**System.out.println(this.name+"是"+this.desc);**

**}**

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

**Year s=getYear(2);**

**s.TestYear();**

**}**

**}**

4：jdk1.5以后增加了etum关键字，用于定义枚举类，一个java源文件只能定义一个public访问权限的枚举类，Java源文件名和该枚举类名一致

1. 枚举类可以实现一个接口，或多个接口，其中etum类默认继承java。long。Etum类，java.long.Etum类实现了java.long.Serializable和java.long.Comparable接口
2. 枚举类的构造器只能用于private
3. 枚举类的所有实现必须在枚举类中显式列出。这些实例必须有 public static final 修饰
4. 枚举类都有一个values方法，遍历所有枚举值

**package neibulei;**

**public enum Season {**

**SPRING("春天","春天来了，又到了动物交配的季节了"),SUMMER("夏天","热死个人喽"),FALL("秋天","这阵风咋感觉阴飕飕地捏"),WINTER("冬天","怪地很，今个天真冷");**

**private String desc;**

**private String name;**

**private Season(String name,String desc)**

**{**

**this.name=name;**

**this.desc=desc;**

**}**

**public String getDesc() {**

**return desc;**

**}**

**public void setDesc(String desc) {**

**this.desc = desc;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**}**

**package neibulei;**

**public class TestSeason {**

**public TestSeason() {**

**// TODO Auto-generated constructor stub**

**}**

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

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

**System.out.println(Season.SPRING.getName());**

**System.out.println(Season.SPRING.getDesc());**

**for(Season season:Season.values()){**

**System.out.println(season+","+season.getName()+","+season.getDesc());**

**}**

**}**

**}**

**package neibulei;**

**public enum Gender {**

**NAEL,FEMALE;**

**private String name;**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**switch(this){**

**case NAEL:**

**if(name.equals("man"))**

**{**

**this.name = name;**

**}else{**

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

**return ;**

**}**

**case FEMALE:**

**if(name.endsWith("woman"))**

**{**

**this.name = name;**

**}else{**

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

**return ;**

**}**

**break;**

**}**

**}**

**package neibulei;**

**public class TestGendle {**

**public TestGendle() {**

**// TODO Auto-generated constructor stub**

**}**

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

**Gender g=Enum.valueOf(Gender.class,"FEMALE");**

**g.setName("man");**

**System.out.println(g+"代表的"+g.getName());**

**}**

**}**

**九、对象垃圾回收**

1、垃圾回收机制的特征

1. 垃圾回收，只会收内存中的对象，不会回收任何物理资源
2. 程序无法精确的回收，垃圾回收只会在合适的时间进行
3. 垃圾回收任何对象前，总会调用他的finalize（）该函数可能让对象复活，从而导致垃圾回收取消

2、对象在内存状态的演变

**集合**

**一、java集合概述**

Java集合一种工具类，可以存储多个对象。不能存储数组，并且数组声明大小后不能改变，在处理数据时，数据对于长度变化或者有映射关系的数据，数组无法处理，为了解决这个问题，提供了集合类，集合类负责盛装其他数据，因此集合类也叫容器类，所有集合类包在java.util下

Java的集合类主要有两个接口派生出，collection和map

Java的集合分为三大类：set，list‘map

Set里的元素无序，不能添加重复元素

List是一个长度可变的数组

Map每项数据都是两个值组成的，一个key一个value

**二、Collection和Iterator接口**

1、  Collection操作

2、  使用Iterator接口遍历集合元素

主要由3个方法，hasNext（），next（），remove（）

           3、使用foreach循环遍历集合元素

**三、set接口**

1、hashset类，treeset类，enumset类和collection基本相同，不允许添加重复元素。

**四、list接口**

1、list集合是一个有序的集合，集合中每个元素都有其对应的顺序索引。

2、list接口的使用

3、ArrayList和Vector实现类

共同特点：都是动态再分配的数据，初始大小10，一般情况下，不需要关注大小，当增加元素时，系统会跟据实际情况，增加大小，但是当添加大量元素时，可以一次性增加capacity值，这样可以减少重新分配次数提高性能

区别ArrayList线程不安全，Vector线程安全ArrayList操作效率更高，Vector效率稍低。

4、固定长度的List

Arrays.ArrayList是一个固定长度的List ，Arrays是操作数据的工具类。Arrays.ArrayList，程序只能遍历集合中的元素，不能添加删除元素。

List a=Arrays.asList()//把数据或对象转换成list集合

不能添加元素和删除元素，程序只能遍历集合中的元素

**五、Queue接口**

1、Queue接口模拟接口这钟数据结构（先进先出）只能操作队首，或者队尾的元素

2、LinkedList实现类

List接口的实现类，可以随机访问集合中的元素。LinkedList实现了Deque接口，Deque是queue接口的子接口，是一个双向队列。功能强大。

3、几种集合性能比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 实现机制 | 随机访问 | 迭代操作 | 插入操作 | 删除操作 |
| 数组 | 连续内存区保存元素 | 1 | 不支持 | 不支持 | 不支持 |
| ArrayList | 内部以数据保存元素 | 2 | 2 | 2 | 2 |
| Vector | 同上 | 3 | 3 | 3 | 3 |
| LinkedList | 内部以列表形式保存元素 | 4 | 1 | 1 | 1 |

操作建议：ArryList、Vector集合使用随机访问集合get，性能较好，linkedlist使用迭代器lterator性能较好，需要经常插入、删除集合使用LinkedList较好

**六、map接口**

1、map是保存基友映射关系的数据，数据项一键值对存在，键key，值value，都可以是任意类型的数据key不允许重复，

2、key和value是一一对应的，keySet（）会返回所有key的集合，所有value放在一起类似一个list

3、常用函数，见api

4、HashMap和HashTable

区别：HashTable线程安全，HashMap线程不安全，HashMap效率更高

HashTable不允许null作为key和value，hashmap允许null值作为key和value但只能有一个

共同点都具有map接口特点

**public class TestList {**

**/\*\***

**\* @param args**

**\*/**

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

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

**List books = new ArrayList();**

**books.add("book1");**

**books.add("book2");**

**books.add("book3");**

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

**books.add(1,"book4");**

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

**for (int i = 0; i < books.size(); i++) {**

**System.out.println(books.get(i));**

**}**

**books.remove(2);**

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

**System.out.println(books.indexOf("book3"));**

**books.set(1, "book2");**

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

**List sub = books.subList(1, 2);**

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

**}**

**}**

**public class TestLinkedList {**

**/\*\***

**\* @param args**

**\*/**

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

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

**LinkedList books = new LinkedList();**

**books.offer("book1");**

**//将book2字符串入栈**

**books.push("book2");**

**books.offerFirst("book0");**

**for (int i = 0;i < books.size(); i++) {**

**System.out.println(books.get(i));**

**}**

**// 访问、并不删除队列的第一个元素**

**System.out.println(books.peekFirst());**

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

**// 访问、并不删除队列的最后一个个元素**

**System.out.println(books.peekLast());**

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

**//采用出栈的方式将删除第一个元素**

**System.out.println(books.pop());**

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

**//访问、并删除队列的最后一个元素**

**System.out.println(books.pollLast());**

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

**}**

**}**

**public class TestPerformance {**

**/\*\***

**\* @param args**

**\*/**

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

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

**String[] tst1 = new String[900000];**

**for (int i=0; i < tst1.length; i++) {**

**tst1[i] = String.valueOf(i);**

**}**

**ArrayList al = new ArrayList();**

**for (int i=0; i < tst1.length; i++) {**

**al.add(tst1[i]);**

**}**

**LinkedList ll = new LinkedList();**

**for (int i=0; i < tst1.length; i++) {**

**ll.add(tst1[i]);**

**}**

**long start = System.currentTimeMillis();**

**for (Iterator it = al.iterator();it.hasNext();) {**

**it.next();**

**}**

**System.out.println("迭代Arraylist集合元素的时间："+(System.currentTimeMillis()-start));**

**long start1 = System.currentTimeMillis();**

**for (Iterator it = ll.iterator();it.hasNext();) {**

**it.next();**

**}**

**System.out.println("迭代Linkedlist集合元素的时间："+(System.currentTimeMillis()-start1));**

**}**

**}**

**public class FixedSizeList {**

**/\*\***

**\* @param args**

**\*/**

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

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

**//把一个数据或指定个数的对象转换成List集合**

**List fixedList = Arrays.asList("book1","book2");**

**System.out.println(fixedList.getClass());**

**for (int i = 0; i < fixedList.size(); i++) {**

**System.out.println(fixedList.get(i));**

**}**

**//fixedList.add("books");**

**fixedList.remove("book1");**

**java.lang.Class c = ArrayList.class;**

**}**

**}**

**public class TestMap {**

**/\*\***

**\* @param args**

**\*/**

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

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

**Map map = new HashMap();**

**map.put("id1", "book1");**

**map.put("id2", "book2");**

**map.put("id3", "book3");**

**Iterator it = map.keySet().iterator();**

**while (it.hasNext()) {**

**String bookid = (String)it.next();**

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

**String bookvalue = (String)map.get(bookid);**

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

**}**

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

**map.remove("id2");**

**Iterator it1 = map.keySet().iterator();**

**while (it1.hasNext()) {**

**String bookid = (String)it1.next();**

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

**String bookvalue = (String)map.get(bookid);**

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

**}**

**System.out.println(map.containsKey("id2"));**

**System.out.println(map.containsKey("id1"));**

**System.out.println(map.containsValue("book2"));**

**System.out.println(map.containsValue("book1"));**

**map.clear();**

**System.out.println(map.isEmpty());**

**}**

**七、操作类集合的工具类：collection**

1、将集合对象设置为不可变，对集合对象实现同步控制，并且该工具类提供了很多对集合排序、查询、修改的操作

**package** xx;

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** TestSort {

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

ArrayList a1=**new** ArrayList();

a1.add(2);

a1.add(-5);

a1.add(3);

a1.add(9);

System.***out***.println(a1);

Collections.*reverse*(a1);//逆序

System.***out***.println(a1);

Collections.*sort*(a1);//升序排序

System.***out***.println(a1);

Collections.*shuffle*(a1);//随机

System.***out***.println(a1);

Collections.*swap*(a1, 1, 2);//交换位置

System.***out***.println(a1);

}

}

2、同步控制

Collections类提供了多个synchronizedXxx方法，返回指定集合对象对应的同步对象，从而解决多线程并发访问的线程安全问题

**Java基础类库**

通过这些基础类库可以提高开发效率，降低开发难度

**一、与用户互动**

1、运行程序的参数

2、使用scanner获取键盘输入

**二、系统相关类**

Java具有平台相关性，如果想获得平台的信息，或调用平台命令完成相关工作，通过System类和Runtime类

1、System类

代表当前java程序的运行平台，不能创建对象，通过System类调用方法和属性

属性：in，out，err

[getenv](mk:@MSITStore:D:\api\jdk1.6API中文文档%20CHM\JDK_API_1_6_zh_CN.CHM::/java/lang/System.html#getenv())()   
          返回一个不能修改的当前系统环境的字符串映射视图

2、Runtime类和java程序运行环境相关的类，每个java程序对应一个Runtime实例，应用程序通过对象与其运行时环境相关联

程序不能自己创建Runtime实例，通过getRuntime获得与之相关联的实例

**public class TestSort {**

**/\*\***

**\* @param args**

**\*/**

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

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

**ArrayList nums = new ArrayList();**

**nums.add(2);**

**nums.add(-5);**

**nums.add(3);**

**nums.add(0);**

**nums.add(-5);**

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

**System.out.println(Collections.max(nums));**

**System.out.println(Collections.min(nums));**

**Collections.replaceAll(nums, 0, 1);**

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

**System.out.println(Collections.frequency(nums, -5));**

**/\*System.out.println(nums);**

**Collections.reverse(nums);**

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

**Collections.sort(nums);**

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

**//Collections.shuffle(nums);**

**//System.out.println(nums);**

**Collections.swap(nums, 1, 2);**

**System.out.println(nums);\*/**

**Collection c = Collections.synchronizedCollection(nums);**

**List list = Collections.synchronizedList(nums);**

**Set set = Collections.synchronizedSet(new HashSet());**

**Map map = Collections.synchronizedMap(new HashMap());**

**}**

**}**

**public class TestArgs {**

**/\*\***

**\* @param args**

**\*/**

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

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

**System.out.println(args.length);**

**for (int i = 0; i < args.length; i++) {**

**System.out.println(args[i]);**

**}**

**}**

**}**

**public class TestScanner {**

**/\*\***

**\* @param args**

**\*/**

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

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

**//Scanner sc = new Scanner(System.in);**

**/\*while (sc.hasNext()) {**

**System.out.println(sc.next());**

**}\*/**

**/\*while (sc.hasNextLong()) {**

**System.out.println(sc.nextLong());**

**}\*/**

**/\*while (sc.hasNextLine()) {**

**System.out.println(sc.nextLine());**

**}\*/**

**try {**

**Scanner sc =**

**new Scanner(new File("src/class8/a.txt"));**

**while (sc.hasNextLine()) {**

**System.out.println(sc.nextLine());**

**}**

**} catch (FileNotFoundException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**}**

**public class TestSystem {**

**/\*\***

**\* @param args**

**\*/**

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

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

**//Map<String,String> map = System.getenv();**

**/\*for (String str : map.keySet()) {**

**System.out.println(str + ":" +map.get(str));**

**}\*/**

**System.out.println(System.getenv("JAVA\_HOME"));**

**Properties pros = System.getProperties();**

**try {**

**pros.store(new FileOutputStream("pros.txt"), "system proprites");**

**} catch (FileNotFoundException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**} catch (IOException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**System.out.println(System.getProperty("os.name"));**

**}**

**}**

**public class TestRuntime {**

**/\*\***

**\* @param args**

**\*/**

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

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

**Runtime rt = Runtime.getRuntime();**

**System.out.println("处理器数量："+rt.availableProcessors());**

**System.out.println("空闲内存数："+rt.freeMemory());**

**System.out.println("总内存数："+rt.freeMemory());**

**try {**

**rt.exec("notepad.exe");**

**rt.exec("calc.exe");**

**} catch (IOException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**}**

**三、常用类**

1、object类

所有类的父类，java允许把任何类型的对象赋给Object，任何Java对象都可以调用object类这个方法

2、String类，stringBuffer和StringBuilder

String类和StringBuffer，都是封装字符串，都提供了一系列操作字符串的方法

初始化后的String字符串不可变，StringBuffer创建一个字符串对象后，可以通过一系列的操作来改变字符串

StringBuffer类，addend追加信息，insert把要插入的信息插入到指定位置、reverse（）反转、setCharAt（）指定位置设置字符、setLength（）设置长度、toString（）StringBuffer转换为String类型。

StringBuilder，和StringBuffer类似，但是，是线程不安全的，效率高一些

**package** xx;

**public** **class** Stringtest {

**public** Stringtest() {

// **TODO** Auto-generated constructor stub

}

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

String s = **new** String("0123456789");

System.***out***.println(s.charAt(5));

String s1=**new** String("\*");

String s2=**new** String("\*abcdefg\*");

System.***out***.println(s2.startsWith(s1));

System.***out***.println(s2.endsWith(s1));

String s3=**new** String("\*ABCdefg\*");

System.***out***.println(s2.equals(s3));

System.***out***.println(s2.equalsIgnoreCase(s3));//比较时忽略大小写

System.***out***.println(s2.indexOf(s1));

String s4=**new** String("sihfiu wefu iow ewo iuw o ow j");

System.***out***.println(s4.indexOf("o"));

System.***out***.println(s4.indexOf(" ",(s4.indexOf(" ")+1)));

System.***out***.println(s.indexOf("u",6));

System.***out***.println(s4.lastIndexOf(" "));

System.***out***.println(s4.substring(s4.lastIndexOf(" ")));

System.***out***.println(s4.substring(s4.indexOf("i"), s4.lastIndexOf("o")));

System.***out***.println(s4.toUpperCase());//转换大小写

System.***out***.println("ERUIWTGHW".toLowerCase());

StringBuilder w1=**new** StringBuilder();

w1.append("java");

System.***out***.println(w1.toString());

w1.insert(0, "hello");

System.***out***.println(w1);

w1.replace(5, 6, "1");

System.***out***.println(w1);

System.***out***.println(w1.length());

System.***out***.println(w1.capacity());

w1.setLength(2);

System.***out***.println(w1.toString());

}

}

3、Math类

Java中提供了加减乘除基本运算符，更复杂的要用Math类

见api

4、BigDecimal类

不可变的、任意精度的有符号十进制数。BigDecimal 由任意精度的整数非标度值 和 32 位的整数标度 (scale) 组成。

BigDecimal 类提供以下操作：算术、标度操作、舍入、比较、哈希算法和格式转换。

Flout、double类型容易丢失精度，使用BigDecimal

BigDecimal b1=**new** BigDecimal("1.234");

BigDecimal b2=**new** BigDecimal("0.14");

BigDecimal b3=**new** BigDecimal("0.05");

BigDecimal b4=**new** BigDecimal(1.234);

System.***out***.println(b1.add(b2));

System.***out***.println(b1.add(b3));

System.***out***.println(b1.add(b4));

System.***out***.println(b2.add(b4));

System.***out***.println(b1.divide(b2,3,2));

**四、处理日期的类**

1、Date类

2、Calender

3、time包

LocalDate类 jdk1.8

Static LocalTime now（）构造一个表示当前日期的对象

Static LocalTime of（int year，int month，int day）构造一个表示给定日期的对象

Int getYear（）

Int getMonthValue（）；

Int getDayOfMonth（）

得到当前日期的年月日

DayOfWeek getDayOfWeek得到星期

LocalDate plusDays（int n）

LocalDate minusDays（int n）

生成当前日期之前货之后n天的日期

对于新API：

非常有用的值类型：

Instant ----- 与java.util.Date相似

ZonedDateTime ----- ZoneId -时区很重要的时候使用

OffsetDateTime ----- OffsetTime, ZoneOffset -对UTC的偏移处理

Duration, Period ----- 但如果你想找到两个日期之间的时间量，你可能会寻找ChronoUnit代替（详情见下文）

其他有用的类型：

DateTimeFormatter ----- 将日期类型转换成字符串类型

ChronoUnit ----- 计算出两点之间的时间量，例如ChronoUnit.DAYS.between(t1, t2)

TemporalAdjuster ----- 例如date.with(TemporalAdjuster.firstDayOfMonth())

Clock类

Clock类提供了访问当前日期和时间的方法。Clock使用时区来访问当前的instant, date和time。Clock类可以替换 System.currentTimeMillis() 和 TimeZone.getDefault()。

1. //Clock 时钟
2. Clock clock1 = Clock.systemDefaultZone();//获取系统默认时区 (当前瞬时时间 )
3. System.out.println( "系统时间日期："+clock1.instant() );
4. System.out.println( "时间毫秒："+clock1.millis() );
6. **final** Clock clock = Clock.systemUTC();//获取系统时钟，并将其转换成使用UTC时区的日期和时间
7. System.out.println( "时间日期："+clock.instant() );
8. System.out.println( "时间毫秒值："+clock.millis() )

某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象

1. Instant instant = clock1.instant();
2. Date javadate = Date.from(instant);
3. System.out.println( "date："+javadate);

 ZoneId（时区）

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of()来获取到。时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

1. System.out.println(ZoneId.getAvailableZoneIds());
3. ZoneId zone1 = ZoneId.of("Europe/Berlin");
4. ZoneId zone2 = ZoneId.of("Brazil/East");
5. System.out.println(zone1.getRules());
6. System.out.println(zone2.getRules());
7. //输出结果： ZoneRules[currentStandardOffset=+01:00]
8. //输出结果： ZoneRules[currentStandardOffset=-03:00]

 LocalTime（本地时间）

（1）获取现在的本地时间

LocalTime 定义了一个没有时区信息的时间。

1. // Get the local date and local time
2. **final** LocalTime time = LocalTime.now();
3. **final** LocalTime timeFromClock = LocalTime.now( clock );
4. System.out.println( time );
5. System.out.println( timeFromClock );

（2）按时区显示时间

1. ZoneId zone1 = ZoneId.of("Europe/Berlin");
2. ZoneId zone2 = ZoneId.of("Brazil/East");
4. LocalTime now1 = LocalTime.now(zone1);
5. LocalTime now2 = LocalTime.now(zone2);
6. System.out.println("时区：Europe/Berlin---"+now1);
7. System.out.println("时区：Brazil/East---"+now2);

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串

1. LocalTime late = LocalTime.of(22, 12, 18);//时分秒
2. System.out.println(late); // 输出结果：22:12:18
4. DateTimeFormatter germanFormatter = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)
5. .withLocale(Locale.GERMAN);
7. LocalTime leetTime = LocalTime.parse("15:39", germanFormatter);
8. System.out.println(leetTime); // 输出结果： 15:39

LocalDate(本地日期)

LocalDate 表示了一个确切的日期（eg： 2014-03-11）。该对象值是不可变的，使用方式和LocalTime基本一致。

1. Clock clock = Clock.systemDefaultZone();// 获取系统默认时区 (当前瞬时时间 )
3. // Get the local date and local time
4. **final** LocalDate date = LocalDate.now();
5. **final** LocalDate dateFromClock = LocalDate.now(clock);
6. System.out.println(date);
7. System.out.println(dateFromClock);

从字符串解析一个LocalDate类型和解析LocalTime一样简单**.**

1. DateTimeFormatter germanFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)
2. .withLocale(Locale.GERMAN);
4. LocalDate xmas = LocalDate.parse("25.10.2016", germanFormatter);
5. System.out.println(xmas);

LocalDateTime（本地日期时间）

表示了具体时间和日期。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

**1）**

1. Clock clock = Clock.systemDefaultZone();// 获取系统默认时区 (当前瞬时时间 )
3. // Get the local date/time
4. **final** LocalDateTime datetime = LocalDateTime.now();
5. **final** LocalDateTime datetimeFromClock = LocalDateTime.now(clock);
6. System.out.println(datetime);
7. System.out.println(datetimeFromClock);

**2）**

1. LocalDateTime sylvester = LocalDateTime.of(2016, Month.DECEMBER, 31, 23, 59, 59);
3. DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
4. System.out.println(dayOfWeek);
6. Month month = sylvester.getMonth();
7. System.out.println(month);
9. **long** minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);
10. System.out.println(minuteOfDay)

只要附加上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

1. LocalDateTime sylvester = LocalDateTime.of(2016, Month.DECEMBER, 31, 23, 59, 59);
2. Instant instant = sylvester
3. .atZone(ZoneId.systemDefault())
4. .toInstant();
6. Date legacyDate = Date.from(instant);
7. System.out.println(legacyDate);

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我也可以自定义格式。

1. DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM dd, yyyy - HH:mm");
2. LocalDateTime parsed = LocalDateTime.parse("05 03, 2016 - 07:13", formatter);
3. String string = formatter.format(parsed);
4. System.out.println(string);

ZonedDateTime（日期时间和时区信息）

使用ZonedDateTime，它保存有ISO-8601日期系统的日期和时间，而且有时区信息。

1. Clock clock = Clock.systemDefaultZone();// 获取系统默认时区 (当前瞬时时间 )
2. // Get the zoned date/time
3. **final** ZonedDateTime zonedDatetime = ZonedDateTime.now();
4. **final** ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now(clock);
5. **final** ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now(ZoneId.of("America/Los\_Angeles"));
6. System.out.println(zonedDatetime);
7. System.out.println(zonedDatetimeFromClock);
8. System.out.println(zonedDatetimeFromZone);

Duration类

Duration持有的时间精确到纳秒。很容易计算两个日期中间的差异。

eg：求时间差

1. // Get duration between two dates
2. **final** LocalDateTime from = LocalDateTime.of(2014, Month.APRIL, 16, 0, 0, 0);//年月日时分秒
3. **final** LocalDateTime to = LocalDateTime.of(2015, Month.APRIL, 16, 23, 59, 59);
4. **final** Duration duration = Duration.between(from, to);
5. System.out.println("Duration in days: " + duration.toDays());
6. System.out.println("Duration in hours: " + duration.toHours());</span>

**还有一种获取时间差值的方式：**ChronoUnit

1. ZoneId zone1 = ZoneId.of("America/Cuiaba");
2. ZoneId zone2 = ZoneId.of("Brazil/East");
3. LocalTime now1 = LocalTime.now(zone1);
4. LocalTime now2 = LocalTime.now(zone2);
5. **long** hoursBetween = ChronoUnit.HOURS.between(now1, now2);
6. **long** minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
7. System.out.println(hoursBetween); // 1
8. System.out.println(minutesBetween); // 60

Api

1. **mport** java.time.Clock;
2. **import** java.time.Duration;
3. **import** java.time.Instant;
4. **import** java.time.LocalDateTime;
5. **import** java.time.ZoneId;
6. **import** java.time.ZonedDateTime;
7. **import** java.time.chrono.ChronoLocalDateTime;  z
8. **import** java.time.chrono.Chronology;
9. **import** java.time.chrono.HijrahChronology;
10. **import** java.time.format.DateTimeFormatter;
11. **import** java.time.temporal.IsoFields;
12. **import** java.util.Date;
14. **public** **class** TimeTest {
16. **public** **static** **void** main(String[] args) **throws** InterruptedException {
17. testClock();
18. //      testInstant();
19. //      testLocalDateTime();
20. //      testZonedDateTime();
21. //      testDuration();
22. //      testChronology();
23. //      testNewOldDateConversion();
24. }
25. **public** **static** **void** testClock() **throws** InterruptedException {
26. // 时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
27. Clock c1 = Clock.systemUTC(); // 系统默认UTC时钟（当前瞬时时间
28. System.out.println(c1.millis()); // 每次调用将返回当前瞬时时间（UTC）
30. //相当于System.currentTimeMillis()）
31. Clock c2 = Clock.systemDefaultZone(); // 系统默认时区时钟（当前瞬时时间）
33. Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); // 巴黎时区
34. System.out.println(c31.instant()); // 每次调用将返回当前瞬时时间（UTC）
35. Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));// 上海时区
36. System.out.println(c32.instant());// 每次调用将返回当前瞬时时间（UTC）
38. Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));// 固定上海时区时钟
39. System.out.println(c4.millis());
40. Thread.sleep(1000);
41. System.out.println(c4.millis()); // 不变 即时钟时钟在那一个点不动
43. Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); // 相对于系统默认时钟两秒的时钟
44. System.out.println(c1.millis());
45. System.out.println(c5.millis());
46. }
48. **public** **static** **void** testInstant() {
49. // 瞬时时间 相当于以前的System.currentTimeMillis()
50. Instant instant1 = Instant.now();
51. System.out.println(instant1.getEpochSecond());// 精确到秒 得到相对于1970-01-01
52. // 00:00:00 UTC的一个时间
53. System.out.println(instant1.toEpochMilli()); // 精确到毫秒
54. Clock clock1 = Clock.systemUTC(); // 获取系统UTC默认时钟
55. Instant instant2 = Instant.now(clock1);// 得到时钟的瞬时时间
56. System.out.println(instant2.toEpochMilli());
57. Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); // 固定瞬时时间时钟
58. Instant instant3 = Instant.now(clock2);// 得到时钟的瞬时时间
59. System.out.println(instant3.toEpochMilli());// equals instant1
60. }
62. **public** **static** **void** testLocalDateTime() {
63. // 使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于
64. // ZoneId.systemDefault()默认时区
65. LocalDateTime now = LocalDateTime.now();
66. System.out.println(now);
67. // 自定义时区
68. LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Europe/Paris"));
69. System.out.println(now2);// 会以相应的时区显示日期
70. // 自定义时钟
71. Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
72. LocalDateTime now3 = LocalDateTime.now(clock);
73. System.out.println(now3);// 会以相应的时区显示日期
74. // 不需要写什么相对时间 如java.util.Date 年是相对于1900 月是从0开始
75. // 2013-12-31 23:59
76. LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);
77. // 年月日 时分秒 纳秒
78. LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59, 59, 11);
79. // 使用瞬时时间 + 时区
80. Instant instant = Instant.now();
81. LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault());
82. System.out.println(d3);
83. // 解析String--->LocalDateTime
84. LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
85. System.out.println(d4);
86. LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");// 999毫秒
87. // 等价于999000000纳秒
88. System.out.println(d5);
89. // 使用DateTimeFormatter API 解析 和 格式化
90. DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
91. LocalDateTime d6 = LocalDateTime.parse("2013/12/31 23:59:59", formatter);
92. System.out.println(formatter.format(d6));
93. // 时间获取
94. System.out.println(d6.getYear());
95. System.out.println(d6.getMonth());
96. System.out.println(d6.getDayOfYear());
97. System.out.println(d6.getDayOfMonth());
98. System.out.println(d6.getDayOfWeek());
99. System.out.println(d6.getHour());
100. System.out.println(d6.getMinute());
101. System.out.println(d6.getSecond());
102. System.out.println(d6.getNano());
103. // 时间增减
104. LocalDateTime d7 = d6.minusDays(1);
105. LocalDateTime d8 = d7.plus(1, IsoFields.QUARTER\_YEARS);
106. // LocalDate 即年月日 无时分秒
107. // LocalTime即时分秒 无年月日
108. // API和LocalDateTime类似就不演示了
109. }
111. **public** **static** **void** testZonedDateTime() {
112. // 即带有时区的date-time 存储纳秒、时区和时差（避免与本地date-time歧义）。
113. // API和LocalDateTime类似，只是多了时差(如2013-12-20T10:35:50.711+08:00[Asia/Shanghai])
114. ZonedDateTime now = ZonedDateTime.now();
115. System.out.println(now);
116. ZonedDateTime now2 = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
117. System.out.println(now2);
118. // 其他的用法也是类似的 就不介绍了
119. ZonedDateTime z1 = ZonedDateTime.parse("2013-12-31T23:59:59Z[Europe/Paris]");
120. System.out.println(z1);
121. }
123. **public** **static** **void** testDuration() {
124. // 表示两个瞬时时间的时间段
125. Duration d1 = Duration.between(Instant.ofEpochMilli(System.currentTimeMillis() - 12323123), Instant.now());
126. // 得到相应的时差
127. System.out.println(d1.toDays());
128. System.out.println(d1.toHours());
129. System.out.println(d1.toMinutes());
130. System.out.println(d1.toMillis());
131. System.out.println(d1.toNanos());
132. // 1天时差 类似的还有如ofHours()
133. Duration d2 = Duration.ofDays(1);
134. System.out.println(d2.toDays());
135. }
137. **public** **static** **void** testChronology() {
138. // 提供对java.util.Calendar的替换，提供对年历系统的支持
139. Chronology c = HijrahChronology.INSTANCE;
140. ChronoLocalDateTime d = c.localDateTime(LocalDateTime.now());
141. System.out.println(d);
142. }
144. /\*\*
145. \* 新旧日期转换
146. \*/
147. **public** **static** **void** testNewOldDateConversion() {
148. Instant instant = **new** Date().toInstant();
149. Date date = Date.from(instant);
150. System.out.println(instant);
151. System.out.println(date);
152. }

155. }

类之间转换的示例：

LocalDateTime localDateTime = LocalDateTime.now();

System.out.println("localDateTime :" + localDateTime);

LocalDate localDate = LocalDate.now();

System.out.println("localDate :" + localDate);

LocalTime localtime = LocalTime.now();

System.out.println("localtime :" + localtime);

// 获取当前年份

Year year = Year.now();

System.out.println("year :" + year);

// 从Year获取LocalDate

LocalDate localDate1 = year.atDay(59);

System.out.println("localDate1 :" + localDate1);

// 把LocalTime关联到一个LocalDate得到一个LocalDateTime

LocalDateTime localDateTime1 = localtime.atDate(localDate1);

System.out.println("localDateTime1 :" + localDateTime1);

// 用指定的年获取一个Year

Year year1 = Year.of(2012);

System.out.println("year1 :" + year1);

// 从Year获取YearMonth

YearMonth yearMonth = year1.atMonth(2);

System.out.println("yearMonth :" + yearMonth);

// YearMonth指定日得到 LocalDate

LocalDate localDate2 = yearMonth.atDay(29);

System.out.println("localDate2 :" + localDate2);

// 判断是否是闰年

System.out.println("isLeapYear :" + localDate2.isLeapYear());

//自动处理闰年的2月日期

//创建一个 MonthDay

MonthDay monthDay = MonthDay.of(2, 29);

LocalDate leapYear = monthDay.atYear(2012);

System.out.println("leapYear :" + leapYear);

//同一个 MonthDay 关联到另一个年份上

LocalDate nonLeapYear = monthDay.atYear(2011);

System.out.println("nonLeapYear :" + nonLeapYear);

上面代码的输出结果为：

localDateTime :2013-10-15T15:11:57.489

localDate :2013-10-15

localtime :15:11:57.489

year :2013

localDate1 :2013-02-28

localDateTime1 :2013-02-28T15:11:57.489

year1 :2012

yearMonth :2012-02

localDate2 :2012-02-29

isLeapYear :true

leapYear :2012-02-29

nonLeapYear :2011-02-28

格式化与时间计算

[DateTimeFormatter](http://download.java.net/jdk8/docs/api/java/time/format/DateTimeFormatter.html)：在日期对象与字符串之间进行转换。

[ChronoUnit](http://download.java.net/jdk8/docs/api/java/time/temporal/ChronoUnit.html)：计算出两个时间点之间的时间距离，可按多种时间单位计算。

[TemporalAdjuster](http://download.java.net/jdk8/docs/api/java/time/temporal/TemporalAdjuster.html)：各种日期计算功能。

续前面的代码：

DayOfWeek dayOfWeek = DayOfWeek.of(1);

System.out.println("dayOfWeek :" + dayOfWeek);

//计算两个日期之间的天数，还可以按其他时间单位计算两个时间点之间的间隔。

long between = ChronoUnit.DAYS.between(localDate, leapYear);

System.out.println("between :" + between);

// 线程安全的格式化类，不用每次都new个SimpleDateFormat

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("uuuu MM dd");

// 把日期时间转换为字符串标识

System.out.println("date formatter :" + dateTimeFormatter.format(nonLeapYear));

// 解析字符串形式的日期时间

TemporalAccessor temporalAccessor = dateTimeFormatter.parse("2013 01 15");

System.out.println("temporalAccessor :" + LocalDate.from(temporalAccessor));

Instant instant = Instant.now(); // 时间戳

System.out.println("instant :" + instant);

//计算某月的第一天的日期

LocalDate with = nonLeapYear.with(TemporalAdjuster.firstDayOfMonth());

System.out.println("with :" + with);

// 计算某月的第一个星期一的日期

TemporalAdjuster temporalAdjuster = TemporalAdjuster.firstInMonth(DayOfWeek.MONDAY);

LocalDate with1 = localDate.with(temporalAdjuster);

System.out.println("with1 :" + with1);

// 计算localDate的下一个星期一的日期

LocalDate with2 = localDate.with(TemporalAdjuster.next(DayOfWeek.MONDAY));

System.out.println("with2 :" + with2);

输出：

dayOfWeek :MONDAY

between :-594

date formatter :2011 02 28

temporalAccessor :2013-01-15

instant :2013-10-15T07:55:30.964Z

with :2011-02-01

with1 :2013-10-07

with2 :2013-10-21

java.util.Date到新库类的转换

转换可通过下面的方法进行。

Date.toInstant()

Date.from(Instant)

Calendar.toInstant()

方法概览

该包的API提供了大量相关的方法，这些方法一般有一致的方法前缀：

of：静态工厂方法。

parse：静态工厂方法，关注于解析。

get：获取某些东西的值。

is：检查某些东西的是否是true。

with：不可变的setter等价物。

plus：加一些量到某个对象。

minus：从某个对象减去一些量。

to：转换到另一个类型。

at：把这个对象与另一个对象组合起来，例如：date.atTime(time)。

**泛型**

Jdk1.5后增加了泛型支持，是为了集合能够记住其他元素类型，没有泛型值，一旦把对象放进集合中，集合就会忘了对象的类型，把所有对象当成object类型处理，把元素从集合中取出来，需要进行强制转换，很容易引起ClassCastExction

**一、泛型入门**

1、集合编译时不检查类型的异常

Java集合的两个缺点：

集合对元素类型没有限制，可能会引起一些问题（2）把元素放进集合中，集合就会忘了对象的类型，取出都是object，需要强制转换

2、使用泛型

参数化类型：允许程序创建集合时指定集合元素类型，java的参数化类型被称为泛型

3、java7泛型的菱形语法

Java7之前，使用泛型接口、类、定义变量时，调用构造器后面必须带泛型

**二、深入泛型**

所谓泛型，就是允许在定义类、接口、方法时允许指定类型形参、这个类型形参在声明变量、创建对象、调用方法时动态指定。Java5改写了集合框架中的全部接口和类，为这些接口、类提供泛型支持

1、定义泛型接口、类

<>里的内容就是泛型，允许在定义接口和类声明类型形参，类型形参在接口和类体内被当成类型形参，几乎所有可以使用普通类型的地方都可以使用这种类型形参。Iterator<e>和iterator是不同的数据类型，可以认为前者是后者的子类

2、使用list类型时，如果E形参传为String实参时，会产生一种新的数据类型，List<String>

**package** FanXing;

**public** **class** Apple<T> {

**private** T info;

**public** Apple(){

}

**public** Apple(T info){

**this**.info=info;

}

**public** T getInfo() {

**return** info;

}

**public** **void** setInfo(T info) {

**this**.info = info;

}

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

Apple<String> a1=**new** Apple<>("banana");

System.***out***.println(a1.getInfo());

Apple<Double>a2=**new** Apple<>(2.34);

System.***out***.println(a2.getInfo());

}

}

**import** java.util.List;

**import** java.util.Map;

**public** **class** Fanxing1Text {

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

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

str.add("book1");

str.add("book2");

str.add("book3");

str.add("book4");

//str.add(5);

**for**(**int** i=0;i<str.size();i++){

String s=(String) str.get(i);

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

Map<String,List<String>> sc=**new** HashMap<>();

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

scc.add("dadada");

scc.add("xiaoxiaoxiao");

scc.add("miaomiaomiao");

sc.put("lala",scc);

**for**(String key:sc.keySet()){

List<String> list= sc.get(key);

System.***out***.println(key+"\*\*\*\*\*\*\*\*\*"+list);

}

}

}

}

3、从泛型类派生子类

当创建了带泛型声明接口或父类类后，可以为接口创建实现类，或从该父类派生子类，当使用这些接口、父类时不能在包括类型形参

**public** **class** A<T> **extends** Apple {}

4、并不存在的泛型类

ArrayList<String>不是一个新的类，仍然是ArrayList

**三、类型通配符**

使用泛型时（声明变量，和创建对象），都应该为这个泛型传入一个实参，如果没有传会有警告

**public** **class** TestArrayList {

**public** TestArrayList() {

// **TODO** Auto-generated constructor stub

}

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

List<String> l1=**new** ArrayList<>();

List<Integer> l2=**new** ArrayList<>();

System.***out***.println(l1.getClass()==l2.getClass());

}

**public** **void** setU(List<?> c){

**for**(**int** i=0;i<c.size();i++){

System.***out***.println(c.get(i));

}

}

**public** **void** Test(){

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

setU(l);

}

}

List<String>不是List<Object>的子类

1、使用类型通配符

（1）为了表示各种泛型List的父类，可以使用类型通配符，用？来表示

将？作为类型实参传给List集合 List<?>意思是元素类型未知的Lsit，可以匹配任何类型

这种带有通配符的List仅仅表示它是各种泛型List的父类，并不能吧元素加入该List

List<?> l1=new ArrayList<>();

l1.add(new Object());

程序无法确定集合中元素类型，无法添加元素，根据Li add（）方法有类型参数E作为集合的元素类型，所以传入的参数必须是E类的对象，或者E的子类。

程序可以调 L?>集合指定索引处的元素，返回的值是未知的类型

st接口定义可以发现，

2、设定类型通配符的上限

List<? extends XingZuang> ？表示任意类型，但必须是xingzuang的子类

3、类型通配符的下限：<?Super Type>,表示type本身或者type的父类

**public** **class** TestDrr {

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

List<Yuan> list=**new** ArrayList<>();

List<JuXing> list1=**new** ArrayList<>();

Yuan y=**new** Yuan();

JuXing t=**new** JuXing();

list1.add(t);

list.add(y);

TestDrr td=**new** TestDrr();

td.draw(list);

td.draw(list1);

}

**public** **void** draw(List<? **extends** XingZuang> shs){

/\*for(Object s:shs){

XingZuang ss=(XingZuang)s;

ss.draw(this);}\*/

**for**(XingZuang s:shs){

s.draw(**this**);

}

}

}

**public** **abstract** **class** XingZuang {

**public** **abstract** **void** draw(TestDrr td);

**public** XingZuang() {

// **TODO** Auto-generated constructor stub

}

}

**public** **class** Yuan **extends** XingZuang {

@Override

**public** **void** draw(TestDrr td) {

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

System.***out***.println(td+"被调用");

}

}

**public** **class** JuXing **extends** XingZuang {

@Override

**public** **void** draw(TestDrr td) {

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

System.***out***.println(td+"被调用2");

}

}

**四、泛型方法**

类和接口使用类型形参，在该类的方法定义和成员变量定义，这些类型形参会当作普通的类型使用

1、定义泛型方法

修饰符 <T,S>返回值类型 方法名（型参 列表）{}

2、泛型方法和类型通配符的区别和联系

（1）、一般情况下可以使用泛型方法代替类型通配符，使用泛型方法实现

<T>Boolean containsall(Collection<T>c);

<T extends E>Boolean addall（<T>c);

上面方法中，类型形参只使用了一次，类型形参T产生的唯一效果就是在不同的调用点传入不同的实际参数，对于这钟情况应该用类型通配符，因为通配符就是被设计用来支持灵活的子类化的。

2）泛型方法允许类型形参被用来表示方法的一个或多个参数之间的依赖关系，或者方法返回值和参数之间的依赖关系。如果没有依赖关系就不要用泛型方法

3)泛型方法和通配符可以同时使用

**public** **class** formArrayToCollection {

**public** **static** <T>**void** formArrayListToCollection(T[] a,Collection<T> c){

**for**(T o:a){

c.add(o);}

}

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

Object []oa=**new** Object[100];

Collection<Object> co=**new** ArrayList<>();

*formArrayListToCollection*(oa, co);

String[] sa=**new** String[100];

Collection<String>cs=**new** ArrayList<>();

*formArrayListToCollection*(sa, cs);

Integer []ss=**new** Integer[100];

Collection<Integer>ws=**new** ArrayList<>();

*formArrayListToCollection*(ss, ws);

}

}

1. 类型通配符和泛型方法的显著区别

类型通配符即可以在方法签名中定义参数类型，也可以定义变量的类型，泛型方法的类型形参必须在对应的方法中显示声明

3、java7的菱形语法和泛构造器

Java允许在构造器签名中声明类型参数，一旦定义了泛型构造器，在调用泛型构造器时不仅可以让java根据数据参数的类型来推断类型参数的类型，而且程序员也可以显示的为构造器的类型形参指定实际参数

**public** **class** FOO {

**public** <T> FOO(T t) {

// **TODO** Auto-generated constructor stub

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

}

}

**class** tee{

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

**new** FOO("");

**new** FOO(666);

**new** <String>FOO("eie");

}

}

Java7的菱形语法允许调用构造器时在构造器后使用一对<>来代表泛型信息，但如果显示指定了泛型构造器中声明的类型参数的实际参数，就不能用菱形语法

**class** mm<E>{

**public** <T>mm(T t){

System.***out***.println("t的参数为"+t);

}

}

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

mm<String> m1=**new** mm<>(5);

mm<String> m2=**new**<Integer> mm<String>(5);

}

**五、集合的擦除和转换**

1、在严格的泛型代码中，带泛型声明的类总应该带有类型参数，但为了和老版java保持一致也允许在使用带有泛型声明的类不指定类参数，会默认为原始的类型参数（声明该类型参数时的第一个上限参数）

2、当把一个具有泛型信息的对象赋给一个没有泛型信息的对象时，所有<>内的类型信息都会被扔掉😂，一个List<String>类型被转换成List，则该List对集合元素的类型转变为类型参数的上限（object）这种情况叫做类型的擦除

3、对泛型而言，可以直接把一个List对象赋给一个List<String>,编译器仅仅只提⑩”未经检测的转换”

**异常处理**

**一、异常概述**

异常处理已成为一门语言成熟的标志，有了完善的异常处理机制，应用程序的容错性更好。

Java异常处理：try 、catch、finally、throw、throws

编译阶段的异常叫做checked异常，有些异常只有在运行时才能被发现，在运行时被发现的异常叫做Runtime异常

希望有一种机制能够一次处理所有异常：如用户录入不合法等，程序只处一次。

If（用户录入正常或其他正常业务）{正常业务处理}else{异常处理机制处理}

**二、异常处理机制**

1、Try{}catch（）{}

Try程序块发生异常，系统生成异常对象，并把异常抛出来，java运行环境会接受到异常对象，会找到处理异常的catch块，如果捕获不到异常，程序就会退出😡

2、异常类的继承体系

（1）java捕获异常的流程

系统可能发生不同意外情况，系统会生成不同的异常对象，对应不同的catch块处理，但是只能有一个catch块被执行

Try{}catch（ExceptionClass1 a）{}catch（ExceptionClass b）{}

（2）异常类

A、java把非正常情况分为两类：异常（exception）和错误（error）

B、error：虚拟机相关问题，这中错误无法恢复，也不能被捕获到

C、java常见的异常体系



D、先捕获小异常再捕获大异常

3、访问异常信息

通过catch后的形参获得异常信息

所有的异常对象都包含了一下几种常用的方法

1）getMessage（）返回异常信息的详细描述

2）printStackTrace（）打印异常跟踪栈的信息

3）printStackTrace（PrintStreams）打印到指定输出流

4）getStackTrace（）返回该异常的跟踪栈信息

4、使用finally回收资源

1）当try中打开了数据库连接、网络连接、磁盘文件，这些物理资源都需要显示回收

2）在哪里进行资源回收?

3)try、catch都不能保证会执行到、java提供了finally一定会被执行

4）try是必须的。Finally和catch有一个就可以

5、异常处理嵌套

在try、catch、finally中又包含了完整的异常处理称为异常处理嵌套

**三、Checked异常和Runtime异常**

1、所有RuntimeException类及其子类的实例被称为Runtime其他被称为Checked

2、Checked异常必须被处理，否则无法编译

3、checked异常处理方式

1）明确知道异常信息，提示用户异常信息

2）不知道异常信息如何处理将异常抛给上一级

4、使用throws声明抛出异常

1）思路，当前不知道如何处理异常，这个异常交给上一级调用者处理

2）主要在方法定义时使用，可以抛出多个异常，用，隔开

3）如果某段代码调用了一个带有throws声明的方法希望调用者来处理异常。调用者处理异常有两种方式，一个是把方法放到try中，第二把异常也抛出去

**四、使用throw声明抛出异常**

前面的内容都是系统出现异常时，系统抛出异常，同时，java 允许自行抛出异常。

很多时候，如果业务需要不符合，程序员可以自定义抛出异常

**五、异常的跟踪栈**

异常对象的printStackTrace方法，打印异常的跟踪栈信息。根据输出结构，我可以找到异常的源头信息，并跟踪异常一路触发的过程。

**六、异常处理的规则**

从程序性能优化、结构优化的角度给出一般处理规则

1、不要过度使用异常

不要使用异常处理代替流程控制

2、不要使用过于庞大的try

3、避免使用catcha all

4、不要忽略捕获到的异常

**AWT编程**

**一、GUI（图形用户界面）和AWT （抽象窗口工具集）**

Java使用AWT和swing类完成用户图形界面编程，GUI（graphics user interfance）AWT（abstract window toolkit），awt是早期的用户图形，后来有的swing库

AWT创建的图形界面应用和他所运行的平台有相同的风格，windows和linux不同

Awt相关类放在java.Awt相关子包下



**二、AWT容器**

1、容器（Container）是Component的子类，容器本身也是一个组件，具有组件的所有属性和方法

Component类有如下几个常用方法来设置组件的大小、位置、可见性等。

（1）setLocation(int x ,int y)设置组件的位置

（2）setSize（int width，int height）；同时设置组件大小

（3）setLocation(int x ,int y，int width，int height)

（4）setVisible（boolean b）设置组件可见性

Component add （Componentcomp）想容器中添加组件，并返回添加的组件

Component getComponentAt（int x，int y）返回指定点的组件

Int getComponentCount（）返回该容器内组件的数量

Component【】getComponents（）返回该容器所有组件

2、Awt主要提供了两种主要的容器：

1)windows可独立存在的顶级容器

2）Panel可作为容器容纳其他组件，但不能独立存在，必须添加到其他容器中



3、

（1）Frame

（2）Panel

Panel是一个比较金典的容器，是一个矩形容器，该区域能够承装其他组件。Panel存在的意义就是为其他容器提供空间

特点：

为放置其他组件提供空间，不能单独存在，默认使用FlowLayout布局管理器

（3）ScorllPane

是一个带滚动条的容器，其他特性和Panel类似，不能独立存在。

特点：内部组件占用空间过大时，自动产生滚动条，也可以通过参数设置指定默认有滚动条。默认布局管理器BorderLayout

**三、布局管理器**

为了使生成的图形界面具有两个好的平台无关系，java语言提供了布局管理器来管理组件在容器中的布局，而不是设置大小和位置的方式，windows—》到linux可能涉及大量代码的修改。布局管理器可以根据平台来调整组件的大小，从而保证了程序修改的地方很小

没有设置布局管理器会采用默认布局管理器

Awt提供了五种布局管理器：Flowlayout，Borderlayout，gridlayout，gridbaglayout，cardlayout

Flowlayout：一行排满，排下一行，默认左到右。有3个构造器

Flowlayout（）默认对齐和间距Flowlayout（int align）使用指定对其方式

Flowlayout（int align ，int h，v）使用指定对其方式，垂直，水平间距

Align表明组件的排列方式，左到右，右到左，中间到两边Flowlayout.left

Flowlayout.right, Flowlayout.center

Borderlayout将容器分为五个部分，east，West，south，north，center

普通组件可以放到其中任何一个位置，没有指定区域默认添加到中间

向一个区域添加多个组件，后放的覆盖前面的组件

Frame，dialog，scollpane 默认采用Borderlayout

Borderlayout构造器Borderlayout（）使用默认水平，垂直间距Borderlayout（int x，int y）指定

GridLayout将容器分割成纵横线分割的网格。每个网格大小相同。默认左到右，上到下添加。组件大小由组件所在区域决定

GridLayout构造器：GridLayout（int rows，int cols）采用指定行列数，采用默认间距GridLayout（int rows，int cols，int h，int v）指定间距

GridBagLayout 功能比GridLayout复杂，增加一个组件可跨越多个网络，并可以设置大小互不相同，增加了布局灵活性

CardLayout 以时间而非空间来管理他的组件，他将加入容器的组件看成是一叠卡片，每次只有最上面的一张是可见的，很类似我们的扑克牌

CardLayout（）创建默认构造器 CardLayout（int h，int v）左右边距和上下边距

常用方法：frist（Container target）显示target容器的第一个卡片

Last（Container target）显示target容器中的最后一张卡片

Previous（Container target）显示target容器中的上一张卡片

Next（Container target）显示target容器中的下一张卡片

Show(Container target,String name) 显示target容器中的指定名字卡片

**package** AWT;

**import** java.awt.BorderLayout;

**import** java.awt.Button;

**import** java.awt.FlowLayout;

**import** java.awt.Frame;

**import** java.awt.Panel;

**import** java.awt.ScrollPane;

**import** java.awt.TextField;

**public** **class** TestPanal {

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

Frame w=**new** Frame("Test");

w.setBounds(500,500,500,500);

w.add(**new** Button("男"),BorderLayout.***SOUTH***);

w.add(**new** Button("北"),BorderLayout.***NORTH***);

w.add(**new** Button("西"),BorderLayout.***WEST***);

w.add(**new** Button("动"),BorderLayout.***EAST***);

w.add(**new** Button("中"),BorderLayout.***CENTER***);

w.setVisible(**true**);

Frame f=**new** Frame("test");

f.setLayout(**new** FlowLayout(FlowLayout.***LEFT***,20,5));

**for**(**int** i=0;i<10;i++){

f.add(**new** Button("按钮"+i));

f.setVisible(**true**);

}

f.pack();//组件和容器保持一至

Panel p=**new** Panel();

p.add(**new** TextField(20));

p.add(**new** Button("提交"));

f.add(p);

f.setBounds(0,0,300,300);

f.setVisible(**true**);

Frame c=**new** Frame("test");

ScrollPane d=**new** ScrollPane(ScrollPane.***SCROLLBARS\_ALWAYS***);

d.add(**new** TextField(20),BorderLayout.***SOUTH***);

d.add(**new** Button("提交"),BorderLayout.***NORTH***);

c.add(d);

c.setBounds(0,0,300,300);

c.setVisible(**true**);

}

}

**四、awt常用组件**

Awt组件需要调用平台的图形界面来创建和平台一致的对等体，所以awt只能使用所以平台都支持的组件，所以awt只提供了一些常用组件

1）button按钮，接受点击的操作

2）canvas用于绘图的画布

3）checkbox复选框（可以实现单选框功能）

4）checkboxgroup将多个checkbox组件组成一个组，只有一个能被选中）

5）choice下拉组件

6）frame窗口

7）label标签类，用于放置提示性 文本

8）list列表框组件，可以添加多项条目

9）panel面板，不能单独存在

10）scrollbar滑动条组件

11）scrollpane带水平和垂直滚动条

12)textarea多行文本域

13）texrfield

2、对话框

Windows类的子类，也是一个容器，属于一个特殊的组件，可以独立存在的顶层窗口，用法和普通对话框一样。有两点需要注意

1）通常依赖于其他窗口

2）对话框有非模式，和模式两种窗口。

FileDialog提供了两个比较常用的方法，获取被打开和保存文件的路径

**public** **class** TextCol {

Frame f=**new** Frame("说");

Button ok=**new** Button("发送");

CheckboxGroup bg=**new** CheckboxGroup();

Checkbox male=**new** Checkbox("男",bg,**true**);

Checkbox female=**new** Checkbox("女",bg,**false**);

Checkbox married=**new** Checkbox("是否已婚",**false**);

Choice colorChooseer=**new** Choice();

List colorlist=**new** List(3,**true**);

TextArea ta=**new** TextArea(8,20);

TextField contrnt=**new** TextField();

**public** **void** init(){

Panel checkPanel=**new** Panel();

checkPanel.add(male);

checkPanel.add(female);

colorChooseer.add("红色");

colorChooseer.add("绿色");

colorChooseer.add("蓝色");

checkPanel.add(married);

checkPanel.add(colorChooseer);

colorlist.add("红色");

colorlist.add("绿色");

colorlist.add("蓝色");

Panel bottom=**new** Panel();

bottom.add(contrnt);

bottom.add(ok);

f.add(bottom,BorderLayout.***SOUTH***);

Box topLeft=Box.*createVerticalBox*();

JScrollPane tajsp=**new** JScrollPane(ta);

topLeft.add(tajsp);

topLeft.add(checkPanel);

Box top=Box.*createHorizontalBox*();

top.add(topLeft);

top.add(colorlist);

f.add(top);

f.pack();

f.setVisible(**true**);;

}

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

TextCol c=**new** TextCol();

c.init();

}

}

**public** **class** Testfl {

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

Testfl c=**new** Testfl();

c.init();

}

Frame f=**new** Frame();

FileDialog d1=**new** FileDialog(f,"请选择需要打开的文件",FileDialog.***LOAD***);

FileDialog d2=**new** FileDialog(f,"请选择需要保存的文件",FileDialog.***SAVE***);

Button b1=**new** Button("打开文件");

Button b2=**new** Button("关闭文件");

**private** **void** init() {

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

b1.addActionListener(

**new** ActionListener(){

**public** **void** actionPerformed(ActionEvent e){

d1.setVisible(**true**);

System.***out***.println(d1.getDirectory()+":"+d1.getFile());

}

}

);

b2.addActionListener(

**new** ActionListener(){

**public** **void** actionPerformed(ActionEvent e){

d2.setVisible(**true**);

System.***out***.println(d1.getDirectory()+":"+d1.getFile());

}

}

);

f.add(b1);

f.add(b2,BorderLayout.***SOUTH***);

f.pack();

f.setVisible(**true**);

}

}

**五、事件处理**

1、事件模型的流程



2、事件和事件监听器

事件机制设计三要素:事件源，事件，事件监听器

事件源很容易创建，new一个组件，这个组件就是事件源，事件是有系统产生的程序需要关系，事件监听器是整个事件处理的核心。

事件和事件监听器是对应的

事件有分高级事件和低级事件

低级事件

低级事件是基于特定动作的事件。如鼠标的进入、点击、拖放等动作的鼠标事件，当组件得到焦点、失去焦点时触发的焦点事件。

1）ComponentEvent：组件事件，当组件尺寸放生变化、位置发生移动、显示/隐藏状态发生变化时触发该事件。

2）ContainerEvent：容器事件，当容器里发生添加组件、删除组件时触发该事件。

3）WindowEvent：窗口事件，当窗口的状态发生改变时触发（打开、最大化、最小化、关闭）。

4）FocusEvent：焦点事件，当组件得到焦点或失去焦点时触发该事件。

5）KeyEvent：键盘事件，对键按下、松开、单击时触发此事件。

6）MouseEvent鼠标事件，当鼠标进行单击、按下、松开、移动等动作时触发该事件。

7)PaintEvent组件绘制事件，是一个特殊的事件类型，当GUI组件调用update/paint方法来呈现自身时触发该事件。

高级事件

高级事件是基于语意的事件，他可以不和特定的动作相关联而依赖于触发事件的类。比如：在TextField中按下Enter键会触发ActionEvent事件，选中项目列表中的某一条会触发ItemEvent事件。

（1）、ActionEvent：动作事件，当按钮、菜单项被单击，TextField中按下Enter时触发。

（2）、AdjustmentEvent：调节事件，在滑动条上移动滑块时触发。

（3）、ItemEvent：选项事件，当用户选中某项，或取消选中某项时触发。

（4）、TextEvent：文本事件，当文本框、文本域的文本发生改变时触发。

|  |  |  |
| --- | --- | --- |
| 事件 | 监听器接口 | 处理器及触发时机 |
| ActionEvent | ActionListener | actionPerform：按钮、文本框、菜单项被单击时 |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged:滑块位置发生变化时触发事件 |
| ContainerEvent | ContainerListener | componetAdded：向容器中添加组件时触发 |
| componentRemoved：从容器中删除组件时触发 |
| FocusEvent | FcousListener | focusGained：组件得到焦点时触发 |
| focusLost：组件失去焦点时触发 |
| ComponentEvent | ComponentListener | componentHidden：组件被隐藏时触发 |
| componentMoved：组件位置发生改变时触发 |
| componetResized：组件大小发生变化时触发 |
| componentshown：组件被显示时触发 |
| KeyEvent | KeyListener | keyPressed：按下某个键时触发 |
| keyReleased：松开某个键时触发 |
| keyTyped：单击某个键时触发 |
| MouseEvent | MouseListener | mouseClicked：在某个组件上单击鼠标触发 |
|  | mouseEntered：鼠标进入某个组件时触发 |
| mouseExited：鼠标离开某个组件时触发 |
| mousePressed：在某个组件上按下鼠标时触发 |
| mouseReleased：在某个组件上松开鼠标时触发 |
| TextEvent | TextListener | textValueChanged：当文本组件的文本内容发生改变时触发 |
| ItemEvent | ItemListener | ItemStateChanged：某项被选中或取消时触发。 |
| WindowEvent | WindowListener | windowActivated：窗口被激活时触发 |
| windowclosed：窗口关闭时触发 |
| WindowClosing：当点击右上角“X”按钮时触发 |
| windowDeactivated：窗口失去激活时触发 |
| windowIconified：窗口最小化时触发 |
| windowOpened：窗口被打开时触发 |
| windowDeiconified：窗口被恢复时触发 |

3、事件适配器

事件适配器是监听接口的空实现。

当创建事件监听器时，可以通过集成事件适配器的方式来实现。多方法的监听器接口都有一个对应的事件适配器，但只有一个方法的接口除外。

|  |  |
| --- | --- |
| 监听器接口 | 事件适配器 |
| ContainerListener | ContainerAdapter |
| FocusListener | FocusAdapter |
| ComponetListener | ComponetAdapter |
| KeyListener | KeyAdapter |
| MouseListener | MouseAdapter |
| WindowListener | WindowAdapter |

**六、AWT菜单**

分为:菜单条，菜单，菜单项。

MenuBar:菜单条，菜单的容器

Menu：菜单，菜单项的容器，又是菜单项的子类，可以作为菜单项使用

Menuitem菜单项

PopupMenu上下文菜单组件

CheckboxMenuitem复选框菜单组件

MenuShortcut菜单快捷键组件



**public** **class** lei {

**private** Frame f=**new** Frame();

**private** MenuBar mb=**new** MenuBar();

Menu file=**new** Menu("菜单");

Menu edit=**new** Menu("编辑");

MenuItem xj=**new** MenuItem("新建");

MenuItem bc=**new** MenuItem("保存");

MenuItem tc=**new** MenuItem("退出",**new** MenuShortcut(KeyEvent.***VK\_X***));

CheckboxMenuItem aw=**new** CheckboxMenuItem("自动换行");

MenuItem copy=**new** MenuItem("复制");

MenuItem paste=**new** MenuItem("粘贴");

Menu format=**new** Menu("格式");

MenuItem comm=**new** MenuItem("注释",**new** MenuShortcut(KeyEvent.***VK\_SLASH***,**true**));

MenuItem canece=**new** MenuItem("取消注释");

**private** TextArea ta=**new** TextArea(6,40);

**public** **void** init(){

ActionListener myMenuListener=**new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

String cmd=e.getActionCommand();

ta.append("单击"+cmd+"菜单"+"\n");

**if**(cmd.equals("退出")){

System.*exit*(0);;

}

}

};

tc.addActionListener(myMenuListener);

file.add(xj);

file.add(bc);

file.add(tc);

edit.add(aw);

edit.addSeparator();//分隔符

edit.add(copy);

edit.add(paste);

format.add(comm);

format.add(canece);

edit.add(**new** MenuItem("-"));

edit.add(format);

mb.add(file);

mb.add(edit);

f.setMenuBar(mb);

f.addWindowListener(

**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

ta.append("用户试图关闭窗口\n");

System.*exit*(0);

}

}

);

f.add(ta);

f.pack();

f.setVisible(**true**);

}

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

lei a=**new** lei();

a.init();

}

}

**public** **class** eess {

Frame f=**new** Frame();

TextArea ta=**new** TextArea(6,40);

**public** **void** into(){

f.addWindowListener(**new** MyD());

f.add(ta);

f.pack();

f.setVisible(**true**);}

**public** eess() {

// **TODO** Auto-generated constructor stub

}

**class** MyD **extends** WindowAdapter{

**public** **void** windowClosing(WindowEvent e) {

ta.append("用户试图关闭窗口");

System.*exit*(0);

}

}

}

**public** **class** TestWindowss {

**public** TestWindowss() {

// **TODO** Auto-generated constructor stub

}

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

TestWindowss c=**new** TestWindowss();

c.into();

}

Frame f=**new** Frame();

TextArea ta=**new** TextArea(6,40);

**public** **void** into(){

f.addWindowListener(**new** MListener());

f.add(ta);

f.pack();

f.setVisible(**true**);

}

**class** MListener **implements** WindowListener{

@Override

**public** **void** windowActivated(WindowEvent e) {

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

ta.append("窗口被激活\n");

}

@Override

**public** **void** windowClosed(WindowEvent e) {

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

ta.append("窗口已被关闭\n");

}

@Override

**public** **void** windowClosing(WindowEvent e) {

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

ta.append("用户试图关闭窗口");

System.*exit*(0);

}

@Override

**public** **void** windowDeactivated(WindowEvent e) {

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

ta.append("窗口失去焦点\n");

}

@Override

**public** **void** windowDeiconified(WindowEvent e) {

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

ta.append("窗口被恢复\n");

}

@Override

**public** **void** windowIconified(WindowEvent e) {

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

ta.append("窗口被最小化\n");

}

@Override

**public** **void** windowOpened(WindowEvent e) {

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

ta.append("打开窗口\n");

}

}

}

**Swing**

**一、Swing概述**

使用swing比awt更优秀，因为swing是一种轻量级组件，完全由java实现，不再依赖于本地平台的图形界面，所以在所有平台运行时保持相同的运行效果，独立于平台的swing被称为轻量级组件。

优势：提供大量图形编程组件，不依赖于本地平台因此不会产生与平台相关的bug不同平台运行效果一致

**二、Swing基本组件**

Swing为几乎所有Awt组件提供了对应实现，在awt前加j就变成了swing

**三、Swing组件分类**

1、顶层容器jframe、japplet、jdialog，jwindow

2、中间容器jpanel、jscrollpane、jmenubar

3、基本组件jbuttonjcombobox、jlist、jmenu、jmenuitem灯

4、显示组件（不可编辑信息）jlabel

显示组件（可编辑）jtextfield、jtextarea

**四、Swing组件的特别之处**

1、可以为swing组件设置提示信息，使用setToolTipText（）为组件设置对用户有用的提示信息

2、对许多组件可以使用图标修使，inagation类图标

3、支持可插拔的外观风格，每个组件都对应一个组件ui类，完成绘图，事件处理，决定尺寸大小灯功能，组件ui对象依赖当前使用的plaf使用umanagersetlookandfell（）可以改变图形界面的外观风格使用UIManager.setLookAndFell()，可以改变图形界面的外观风格。如：JButtton组件有一个对应的ButtonUI类作为代理。UI代理类类名规则总是去掉“J”，加上后缀UI。UI代理类通常是一个抽象的基类，不同的plaf会有不同的UI代理类实现。Swing类库中包括了多套不同的UI代理：javax.swing.plaf.metal.MetalLookAndFeel，com.sun.java.swing.plaf.windows.WindowsLookAndFeel, com.sun.java.swing.plaf.motif.MotifLookAndFeel, com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel

4、支持设置边框

Swing组件可以设置一个或多个边框。

为了给Swing组件设置边框，可以调用JCompont提供的setBorder（Border b），其中Border是Swing的一个接口，用来代表组件的边框，该接口有数量众多的实现类，BevelBorder、LineBorder、EmptyBorder、EtchedBorder、TitledBorder、MatteBorder、CompoundBorder，这些类都提供了相应的构造器，用于创建Border对象。Swing提供了静态工厂类，该类提供了大量的静态工厂方法，用于创建Border实例。

**package** swing;

**import** java.awt.BorderLayout;

**import** java.awt.Color;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.InputEvent;

**import** javax.swing.\*;

**import** javax.swing.border.Border;

**public** **class** Testll {

JFrame f=**new** JFrame();

ImageIcon icon=**new** ImageIcon("");

JButton ok=**new** JButton("发送",icon);

JRadioButton male=**new** JRadioButton("男",**true**);

JRadioButton female=**new** JRadioButton("女",**true**);

ButtonGroup bg=**new** ButtonGroup();

JCheckBox married=**new** JCheckBox("是否已婚",**false**);

String[]colors=**new** String[]{"红色","绿色","蓝色"};

JComboBox<?> colorChooser=**new** JComboBox<>(colors);

JList<?> colorList=**new** JList<>(colors);

JTextArea ta=**new** JTextArea(8,30);

JTextField content=**new** JTextField(40);

JMenu file=**new** JMenu("文件");

JMenu edit=**new** JMenu("编辑");

JMenuItem newItem=**new** JMenuItem("新建");

JMenuItem saveItem=**new** JMenuItem("保存");

JMenuItem exitItem=**new** JMenuItem("退出");

JCheckBoxMenuItem autoWrap=**new** JCheckBoxMenuItem("自动换行(W)");

JMenuItem copyItem = **new** JMenuItem("复制");

JMenuItem pastetem = **new** JMenuItem("粘贴");

JMenuBar tiao=**new** JMenuBar();

JMenu format = **new** JMenu("格式");

JMenuItem commentItem = **new** JMenuItem("注释");

JMenuItem cancelItem = **new** JMenuItem("取消注释");

JPopupMenu pop = **new** JPopupMenu();

ButtonGroup fg = **new** ButtonGroup();

JRadioButtonMenuItem metalItem=**new** JRadioButtonMenuItem("metal风格",**true**);

JRadioButtonMenuItem windowsItem=**new** JRadioButtonMenuItem("windows风格");

JRadioButtonMenuItem motifItem=**new** JRadioButtonMenuItem("motif风格");

JRadioButtonMenuItem nimbusItem=**new** JRadioButtonMenuItem("nimbus风格");

**public** **void** into(){

JPanel checkPanel=**new** JPanel();

JLabel j=**new** JLabel("性别:");

checkPanel.add(j);

bg.add(male);

bg.add(female);

checkPanel.add(male);

checkPanel.add(female);

married.setToolTipText("必选项");

checkPanel.add(married);;

checkPanel.add(colorChooser);

JPanel bottom=**new** JPanel();

bottom.add(content);

ok.addActionListener(

**new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

StringBuffer sf = **new** StringBuffer();

sf.append("您已经选择：");

**if** (male.isSelected()) {

sf.append("男性");

} **else** **if** (female.isSelected()) {

sf.append("女性");

}

sf.append("\n");

sf.append("您已经选择：");

**if** (married.isSelected()) {

sf.append("已婚");

} **else** {

sf.append("未婚");

}

sf.append("\n");

sf.append("您已经选择：");

sf.append(colorChooser.getSelectedItem());

sf.append("\n");

sf.append("您已经发送：");

sf.append(content.getText());

ta.append(sf.toString());

}

}

);

bottom.add(ok);

/\*Border bb = BorderFactory.createBevelBorder(BevelBorder.LOWERED,

Color.red,Color.GREEN,Color.BLUE,Color.GRAY);\*/

Border bl = BorderFactory.*createLineBorder*(Color.***orange***,1);

bottom.setBorder(bl);

f.add(bottom,BorderLayout.***SOUTH***);

Box topLeft=Box.*createVerticalBox*();

JScrollPane tajsp=**new** JScrollPane(ta);

topLeft.add(tajsp);

topLeft.add(checkPanel);

Box top =Box.*createHorizontalBox*();

top.add(topLeft);

top.add(colorList);

f.add(top);

newItem.setAccelerator(KeyStroke.*getKeyStroke*('N',InputEvent.***CTRL\_MASK***));

newItem.addActionListener(**new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent arg0) {

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

}

});

file.add(newItem);

file.add(saveItem);

file.add(exitItem);

edit.add(autoWrap);

edit.addSeparator();

edit.add(copyItem);

edit.add(pastetem);

format.add(commentItem);

format.add(cancelItem);

edit.add(format);

tiao.add(file);

tiao.add(edit);

fg.add(metalItem);

fg.add(windowsItem);

fg.add(motifItem);

fg.add(nimbusItem);

pop.add(metalItem);

pop.add(windowsItem);

pop.add(motifItem);

pop.add(nimbusItem);

ActionListener flavorListener = **new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

**if** (e.getActionCommand().equals("metal风格")) {

changeFlavor(1);

} **else** **if** (e.getActionCommand().equals("windows风格")) {

changeFlavor(2);

} **else** **if** (e.getActionCommand().equals("motif风格")) {

changeFlavor(3);

} **else** **if** (e.getActionCommand().equals("nimbus风格")) {

changeFlavor(4);

}

}

};

metalItem.addActionListener(flavorListener);

windowsItem.addActionListener(flavorListener);

motifItem.addActionListener(flavorListener);

nimbusItem.addActionListener(flavorListener);

ta.setComponentPopupMenu(pop);

f.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

f.pack();

f.setVisible(**true**);

f.setJMenuBar(tiao);

}

**public** **void** changeFlavor(**int** flavor){

**switch**(flavor) {

**case** 1:

**try** {

UIManager.*setLookAndFeel*("javax.swing.plaf.metal.MetalLookAndFeel");

} **catch** (ClassNotFoundException | InstantiationException

| IllegalAccessException | UnsupportedLookAndFeelException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**break**;

**case** 2:

**try** {

UIManager.*setLookAndFeel*("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");

} **catch** (ClassNotFoundException | InstantiationException

| IllegalAccessException | UnsupportedLookAndFeelException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**break**;

**case** 3:

**try** {

UIManager.*setLookAndFeel*("com.sun.java.swing.plaf.motif.MotifLookAndFeel");

} **catch** (ClassNotFoundException | InstantiationException

| IllegalAccessException | UnsupportedLookAndFeelException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**break**;

**case** 4:

**try** {

UIManager.*setLookAndFeel*("com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");

} **catch** (ClassNotFoundException | InstantiationException

| IllegalAccessException | UnsupportedLookAndFeelException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**break**;

}

SwingUtilities.*updateComponentTreeUI*(f.getContentPane());

SwingUtilities.*updateComponentTreeUI*(tiao);

SwingUtilities.*updateComponentTreeUI*(pop);

}

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

**new** Testll().into();

}

}

**public** **class** HelloApplet **extends** JApplet {

**public** **void** init(){

setLayout(**new** FlowLayout());

**this**.add(**new** Button("提交"));

**this**.add(**new** Button("保存"));

System.***out***.println("init方法被调用");

}

**public** **void** start(){

System.***out***.println("start方法被调用");

}

**public** **void** stop(){

System.***out***.println("stop方法被调用");

}

**public** **void** destroy(){

System.***out***.println("destory方法被调用");

}

**public** **void** paint(Graphics g) {

**super**.paint(g);

g.drawLine(3, 3, 200, 300);

System.***out***.println("paint方法被调用");

}

}

**五、Applet基础知识**

Applet是java的小程序，java应用程序的一种，通常需要嵌入到HTML中由客户端浏览器下载执行。

Applet不能访问本地磁盘，不能运行客户端程序，不能打开主机上的socket与外界通讯。

开发Applet基本操作

开发Applet需要继承java.swing.JApplet类，与前期学习的java程序不同，Applet类无序提供main方法。Applet是由浏览器执行的，Applet提供以下几个方法：

init：由浏览器调用，对Applet进行初始化。当Applet成功装载后，java第一次运行该Applet时，会调用此方法。通常情况下，我们不在Appplet的构造器中初试化信息，而是在init中初始化信息。

start：由浏览器调用，启动该Applet的执行，在init后被执行。如果在浏览该页面的过程中离开了该页面。当用户再次回到该页面时，重新调用start方法，不会执行init。

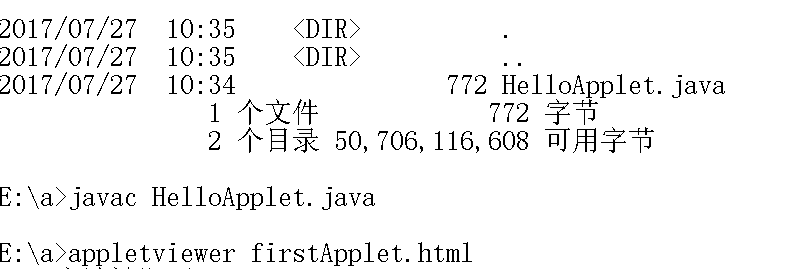
stop：由浏览器调用，用于停止该Applet的执行，与start方法相对。当用户离开此页面时，执行此方法。

destroy：由浏览器调用，用于销毁Applet。与init方法相对，在stop方法后执行。通常情况下，如果用户离开了Applet所在的网页或正常退出浏览器，先调用stop后，调用destory。

paint：该方法继承java.awt.Container，用于将内容绘制到屏幕上，该方法的执行时间由JVM决定，与普通容器执行时间相同，当该Applet大小发生变化，显示或隐藏状态发生变化时，该方法被执行。

这些方法都是Applet生命周期的方法，只需要按要求定义这些方法即可，不需要显示调用这些方法。

使用applet浏览applet



**public** **class** Helel **extends** JApplet {

**private** **static** **final** **long** ***serialVersionUID*** = 8473658060729616732L;

**public** **void** init(){

setLayout(**new** FlowLayout());

**this**.add(**new** Button("提交"));

**this**.add(**new** Button("保存"));

}

**public** **void** start(){

System.***out***.println("startb被调用");

}

**public** **void** stop(){

System.***out***.println("stop被调用");

}

**public** **void** destroy(){

System.***out***.println("destroy被调用");

}

**public** **void** paint(Graphics g){

**super**.paint(g);

g.drawLine(3, 3,200, 200);

System.***out***.println("paint被调用");

}

}

**IO**

**一、I/O（输入/输出）**：

输入：程序读取外部数据（磁盘、光盘、优盘等存储设备的数据）、用户录入的数据。输出：程序将数据记录到磁盘、光盘、优盘等存储设备上。

Java对IO支持，通过java.io、java.nio包下的类和接口来支持。

Java对象的序列化，通过序列化，我们可以把内存中的java对象转化成二进制字节流，把java对象存储到磁盘，或在网络上传输。

**二、File类**

File类可以操作文件和目录。File类能新建、删除、重命名文件和目录，但不能访问文件内容。访问文件内容，需要使用输入/输出流。

1、访问文件和目录

File类可以使用文件路径字符串创建File实例，其中文件路径字符串即可以是绝对路径，也可以是相对路径。

2、File类的常用方法：

（1）访问文件名的相关方法：

String getName():返回File对象所表示的文件名或路径名。

String getPath():返回File对象对应的路径。

File getAbsoluteFile():返回此File对象所对应的绝对路径所对应的File对象。

String getAbsolutePath():返回File对象所对应的绝对路径名。

String getParent():返回File对象对应的父目录名。

boolean rename（File newname）：重命名此file对象对应的文件或目录。如果重命名成功返回true，否则返回false。

（2）文件检测的相关方法

boolean exits()：判读File对象对应的文件或目录是否存在。

boolean canWrite():判读File对象对应的文件和目录是否可写。

boolean canRead():判读File对象对应的文件和目录是否可读。

boolean isFile():判断File对象所对应的是否是文件，不是目录。

boolean isDirectory():判读File对象所对应的是否是目录，不是文件。

boolean isAbsolute():判读File对象所对应的文件和目录是否是绝对路径。

（3）获取常规文件信息

long lastModified():返回文件的最后修改时间。

long length():返回文件内容的长度。

（4）文件操作的相关方法

boolean createNewFile():当此File对象所对应的文件不存在，该方法将新建一个该File对象所指定的新文件。

boolean delete():删除File对象所对应的文件和路径。

static File createTempFile(String prefix，String suffix)：在默认临时文件目录中创建一个临时的空文件，使用给定前缀、系统生成的随机数和给定的后缀作为文件名。prefix至少三个字节。Suffix可以为null，在null的情况下，系统默认后缀.tmp。

static File createTempFile(String prefix，String suffix,File directory)：在directory所指定的文件目录中创建一个临时的空文件，使用给定前缀、系统生成的随机数和给定的后缀作为文件名。prefix至少三个字节。Suffix可以为null，在null的情况下，系统默认后缀.tmp。

void deleteOnExit():指定当java虚拟机退出时，删除File对象所对应的文件和目录。

（5）目录操作的相关方法

boolean mkdir():试图创建一个File对象所对应的目录。调用该方法时File对象必须对应一个路径，而不是一个文件。

String[] list():列出File对象的所有字文件名和路径名。返回String数组。

File[] listFiles():列出File对象的所有子文件和路径，返回File数组。

static File() listRoots():返回系统所有的根路径。

**三、理解java的IO流**

1、流的分类

（1）按流向分类：输入流、输出流

输入流：InputStream和Reader作为基类。从中读取数据，到应用程序中。

输出流：OutputStream和Writer作为基类。从应用程序的内存中，写出数据。

（2）按操作的基本单元：字节流和字符流

字节流最小操作单元为8位字节，字符最小操作单元16位字符。InputStream和OutputStream是字节流，Reader和Writer是字符流。

（3）节点流和处理流

节点流：可以向IO设备读写数据的流。又称为低级流。

处理流：对以存在的流进行封装和连接，通过封装后的流进行数据的复写功能。称为高级流。

2、流的概念模型

输入流：把输入设备抽象成水管，其中依次排列类似水滴的数据单元。有一个隐式的指针记录从哪里开始读取数据，读取的下一个单元。

输出流：数据流向相反，把水滴放到输出流的水管中。

3、字节流和字符流

InputStream和Reader

（1）InputStream和Reader是所有输入流的基类，抽象类。是所有输入流的模板，它们的方法所有输入流都可以使用。

在InputStream里有如下三个常用方法：

int read():从输入流读取单个字节，返回所读取的字节数据。字节数据可以直接转化为int类型。

int read(byte[] b):从输入流中读取做多b.length个字节的数据，并将其存储在字节数据b中，并返回实际读取的字节数。

int read(byte[] b,int offset,int len):从输入流中读取最多len字节的数据，并将其存储在数组b中，放入b中时，从offset位置开始。并返回实际读取的字节数。

在Reader里有如下三个常用方法：

int read():从输入流读取单个字符，返回所读取的字符数据。字符数据可以直接转化为int类型。

int read(char[] b):从输入流中读取做多个b.length字符的数据，并将其存储在字符数据b中，并返回实际读取的字符数。

int read(char [] b,int offset,int len):从输入流中读取最多len字符的数据，并将其存储在数组b中，放入b中时，从offset位置开始。并返回实际读取的字符数。

（2）FileInputStream和FileReader，继承了InputStream和Reader抽象类，可以直接操作文件。

OutputStream和Writer

（1）OutputStream和Writer用法也非常类似，操作模式相同，操作数据的基本单元不同。

常用函数：

1）void write(int c):将指定的字节/字符输出到输出流中。

2）void write(byte[]/char[] buf):将字节数组或字符数组中的数据输入到输出流中。

3）void write(byte[]/char[] buf,inf off,int len):将字节数组或字符数组中从off位置开始，长度为len的字节或字符输入到输出流中。

Writer里还包含以下两个函数：

1）void write(String str):将str字符串里包含的字符输入到指定输出流中。

2）void write(String str,int off,int len)：将str字符串里从off位置开始，长度为len的字符输入到输出流中。

（2）FileOutputStream和FileWriter

输入/输出流体系

3、处理流的用法

使用处理流来包装节点流，程序通过处理流进行输入、输出，节点流与底层IO交互。

4、输入、输出流体系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | FileInputStream | FileQutputStream | FileReader | FileWriter |
| 访问数组 | ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| 访问管道 | PipedInputStream | PipedOutputStream | PipedReader | PipedWriter |
| 访问字符串 |  |  | StringReader | StringWriter |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 |  |  | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream |  |  |
| 抽象基类 | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 |  | PrintStream |  | PrintWriter |
| 推回输入 | PushbackInputStream |  | PushbackReader |  |
| 特殊流 | DataInputStream | DataOutputStream |  |  |

还有一些不在IO包下的类，AudioInputStream、CipherInputStream、DeflaterInputStream、ZipInputStream等，是用于访问音频、加密/解密，压缩/解压缩等具有特殊功能的工具。

**四、重定向标准输入/输出**

System.in，System.out是java的标准输入、输出。实际上就是键盘输入、屏幕输出。

System类提供了三个重定向的方法：

static void setErr（PrintStream err）：重定向错误标准输出流。

static void setIn（InputStream in）：重定向标准输入流。

static void setOut（PrintStream out）：重定向标准输出流。

**public** **class** Rtt {//重定向标准输入

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

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

FileInputStream fs = **null**;

**try** {

fs = **new** FileInputStream("src/class13/RedirectIn.java");

System.*setIn*(fs);

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

**while** (sc.hasNext()) {

System.***out***.println(sc.next());

}

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally** {

**try** {

fs.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**public** **static** **void** main(String[] args) {//重定向标准输出

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

PrintStream ps = **null**;

File file = **new** File("D:\\test\\out3.txt");

**if** (!file.exists()) {

**try** {

file.createNewFile();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

FileOutputStream fos = **null**;

**try** {

fos = **new** FileOutputStream(file.getAbsolutePath());

ps = **new** PrintStream(fos);

System.*setOut*(ps);

System.***out***.println("春夜喜雨");

System.***out***.println("好雨知时节，当春乃发生。");

System.***out***.println("随风潜入夜，润物细无声。");

System.***out***.println("野径云俱黑，江船火独明。");

System.***out***.println("晓看红湿处，花重锦官城。");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally** {

ps.close();

}

}

}

**public** **class** Rtt {

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

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

File file1 = **new** File("D:\\test");

System.***out***.println(file1.exists());

/\*File file2 = new File(".");

System.out.println(file2.getName());

System.out.println(file2.getParent());

System.out.println(file2.getAbsolutePath());

try {

File temFile = File.createTempFile("aaab", ".txt",file2);

} catch (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

File newFile = new File(System.currentTimeMillis() + "");

System.out.println(newFile.exists());

try {

boolean fileExit = newFile.createNewFile();

System.out.println(fileExit);

} catch (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}\*/

**boolean** dirExit = file1.mkdir();

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

/\* String[] fileList = file2.list();

for (String fileName:fileList) {

System.out.println(fileName);

}

File[] roots = File.listRoots();

for (File root:roots) {

System.out.println(root);

}\*/

}

}

**public** **class** Rtt {

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

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

FileInputStream fs = **null**;

**try** {

fs =

**new** FileInputStream("src/class13/TestFileInputStream.java");

**byte**[] bf = **new** **byte**[1024];

**int** hasRead = 0;

**try** {

**while**((hasRead = fs.read(bf)) > 0) {

System.***out***.println(**new** String(bf,0,hasRead));

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally** {

**try** {

fs.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

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

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

FileInputStream fs = **null**;

**try** {

fs =

**new** FileInputStream("src/class13/TestFileOutStream.java");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

FileOutputStream fo = **null**;

**try** {

fo = **new** FileOutputStream("D:\\test\\out.txt");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**byte**[] bf = **new** **byte**[1024];

**int** hasRead = 0;

**try** {

**while**((hasRead = fs.read(bf)) > 0) {

fo.write(bf, 0, hasRead);

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally**{

**try** {

fs.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

fo.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

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

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

FileReader fr = **null**;

**try** {

fr = **new** FileReader("src/class13/TestFileReader.java");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**char**[] cb = **new** **char**[32];

**int** hasRead = 0;

**try** {

**while**((hasRead = fr.read(cb)) > 0) {

System.***out***.println(**new** String(cb,0,hasRead));

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally** {

**try** {

fr.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

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

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

FileWriter fw = **null**;

File file = **new** File("D:\\test\\out1.txt");

**if** (file.exists()) {

file.delete();

}

**if** (!file.exists()) {

**try** {

file.createNewFile();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**try** {

fw = **new** FileWriter(file.getAbsolutePath());

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

fw.write("春夜喜雨\n");

fw.write("好雨知时节，当春乃发生。\n");

fw.write("随风潜入夜，润物细无声。\n");

fw.write("野径云俱黑，江船火独明。\n");

fw.write("晓看红湿处，花重锦官城。\n");

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally** {

**try** {

fw.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

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

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

PrintStream ps = **null**;

File file = **new** File("D:\\test\\out2.txt");

**if** (!file.exists()) {

**try** {

file.createNewFile();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

FileOutputStream fos = **null**;

**try** {

fos = **new** FileOutputStream(file.getAbsolutePath());

ps = **new** PrintStream(fos);

ps.println("春夜喜雨");

ps.println("好雨知时节，当春乃发生。");

ps.println("随风潜入夜，润物细无声。");

ps.println("野径云俱黑，江船火独明。");

ps.println("晓看红湿处，花重锦官城。");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally** {

ps.close();

}

}

**五、RandomAcessFile**

RandomAcessFile是java输入输出中功能最丰富的文件内容访问类，既可以向文件写内容，也可以读数据，可以跳转到文件任意位置读写数据。

可以不从文件开始点读写。

两个方法操作文件指针：

long getFilePointer(f):返回文件记录指针的当前位置

void seek（long pos）：将文件指针定位到参数pos位置

两个构造器：

RandomAccessFile（File file/String name，String mode）：创建从中读取或向其写入的随机存储文件流。mode 参数指定以何种方式打开文件。其取值范围：r（read），rw（文件不存在时创建文件），rws（要求对文件内容或元数据的每个更新都要同步写到基础存储设备），rwd（同rws）

public class AppendContent {

public static void main(String[] args) {

// TODO Auto-generated method stub

RandomAccessFile raf = null;

try {raf = new RandomAccessFile("D:\\test\\out3.txt","rw");

raf.seek(raf.length());

Calendar c = Calendar.getInstance();

StringBuffer b = new StringBuffer();

b.append(c.get(c.YEAR));

b.append("年");

b.append(c.get(c.MONTH)+1);

b.append("月");

b.append(c.get(c.DATE));

b.append("日");

raf.write(b.toString().getBytes());

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally {try {raf.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}}

public class TestRandomAccessFile {

public static void main(String[] args) {

// TODO Auto-generated method stub

RandomAccessFile raf = null;

try {raf = new RandomAccessFile("src/class13/TestRandomAccessFile.java","r");

System.out.println("指针的位置："+raf.getFilePointer());

raf.seek(300);

System.out.println("指针的位置："+raf.getFilePointer());

byte[] bf = new byte[1024];

int hasRead = 0;

while ((hasRead = raf.read(bf)) > 0) {

System.out.println(new String(bf,0,hasRead));

}

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally {try {raf.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}}

**六、对象序列化的含义和意义**

对象序列化的目的是将对象保存到磁盘中，或在网络上进行传输

让某个对象支持序列化，必须让他们的类是可序列化的。类实现Serializable、Externallizable接口，即可实现序列化。

Java的很多类都实现了Serializable接口，实现该接口无序实现任何方法。它只是表明该类时可以序列化的。RMI（Remote Method Invoke，即远程函数调用）的参数、返回值、web应用中需要保存到HttpSession或Servlet的对象。

1、使用对象类实现序列化

（1）程序序列化对象，两个步骤：

创建对象输出流ObjectOutputStream。

使用对象writeObject(new 对象)。

（2）程序反序列化的步骤：

创建对象输入流ObjectInputStream。

调用对象的readObject函数，读取流中的对象。对象强制转换为真实类型。

2、对象引用的序列化

如果某个类的属性类型是引用类型，那么引用类必须是可可序列化的，否则拥有该类型的属性类是不可序列化的。

3、自定义序列化

在一些特定的场景下，如果某个类里包含的属性信息是敏感信息，如银行卡密码，不希望被序列化；或某个属性的类不可序列。因此不希望对该属性递归序列化。可以通过在属性前加上transient关键字，可以指定java序列化时，无需理会该属性。

public class Person implements Serializable {

private String name;

private transient int age;

public Person(String name,int age) {

this.name = name;

this.age = age; }

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 class Class implements Serializable{

private String name;

private Person p;

public String getName() {

return name; }

public void setName(String name) {

this.name = name; }

public Person getP() {

return p; }

public void setP(Person p) {

this.p = p; } }

public class TestWriteObject {

public static void main(String[] args) {

// TODO Auto-generated method stub

ObjectOutputStream oos = null;

File file = new File("D:\\test\\person.txt");

if (!file.exists()) {try {file.createNewFile();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();}}

try {oos = new ObjectOutputStream(new FileOutputStream(file.getAbsolutePath()));

oos.writeObject(new Person("孙悟空",500));

} catch (FileNotFoundException e) {// TODO Auto-generated catch block

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

// TODO Auto-generated catch block

e.printStackTrace();

}finally {try {oos.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();}}}}

public class TestReadObject {

public static void main(String[] args) {

// TODO Auto-generated method stub

ObjectInputStream ois = null;

try {ois = new ObjectInputStream(new FileInputStream("D:\\test\\person.txt"));

Person p = (Person)ois.readObject();

System.out.println(p.getName()+":"+p.getAge());

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} finally {try {ois.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();}}}}

**七、NIO**

从jdk1.4开始，java提供了一系列的改进的输入、输出的新功能，这些新功能统称为新IO（New IO）。新增的许多处理输入、输出的类，都放到了java.nio包及其子包下。

1、java新IO概述

新IO和传统IO有相同的目的，都是用于输入和输出，但是新IO采用内存映射文件的方式来处理输入输出。新IO将文件或文件的一段区域映射到内存文件中，这样就可以像访问内存一样来访问文件了，输入输出更快。

Java中与新IO相关的包如下：

Java.nio包：主要包含了各种与Buffer相关的类。

Java.nio.channels包：总要包括了与Channel和Selector相关的类。

Java.nio.charset包：主要包含了和字符集相关的类。

Java.nio.channels.spi包：主要包含了与Channel相关服务提供者编程接口。

Java.nio.charset.spi包：包含与字符集相关的服务提供者编程接口。

Channel（通道）和Buffer（缓存）是新IO中两个核心对象。Channel是对传统的输入和输出系统的模拟，在新IO系统中所有的数据都需要通过通道传输。Channel与传统InputStream和OutputStream最大区别，它有一个map（）方法，通过该方法可以直接将一块数据映射到内存中。如果说传统的输入、输出系统是面向流处理，则新的IO则是面向块的处理。

Buffer可以理解为容器，发送到channel中的所有对象都必须先放到Buffer中，而从Channel中读取数据也必须先放到Buffer。

新IO还提供了用于将Unicode字符串映射成字节序列以及逆映射操作的charset类。同时还提供selector类用于实现非阻塞的输入、输出。

2、使用Buffer

Buffer是一个数组，可以保存多个类型的相同的数据。Buffer是一个抽象类，其最长用的子类是ByteBuffer。其他基本数据类型（boolean除外）都有相应的（1）Buffer类：

CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。都没有构造器，通过static XxxBuffer allocate（int capasity）：创建一个容量为capasity的XxxBuffer对象。

使用最多的是ByteBuffer和CharBuffer。其中，ByteBuffer有一个子类MappedByteBuffer，它用于标识Channel将磁盘文件的部分或全部内容映射到内存中后得到的结果。通常MappedByteBuffer对象由Channel的map方法返回。

（2）Buffer有三个重要的概念：容量（capacity）、界限（limit）、位置（position）。

容量（capacity）：缓冲区的容量表示该Buffer的最大数据容量。创建后就不能改变，缓冲区的容量不能为负值。

界限（limit）：标识第一个不应该被读出或写入的缓冲区的位置。位于limit后的数据既不能读也不能写。

位置（position）：用于指明下一个可以被读出或写入的缓冲区的位置。当刚刚建一个Buffer对象时，其position是0；如果从Channel读取了2个数据到Buffer中，则position为2。指向Buffer中第3个位置。

mark：标记。

0<=mark<=position<=limit<=capacity

（3）Buffer的主要作用就是装入数据，然后输出数据；开始时Buffer的position为0，limit为capacity，程序通过put（）方法向Buffer中放入数据，或者从Channel中获取数据，每放入一些数据，Buffer的position后移一些位置。当Buffer装入数据结束后，调用Buffer的flip()方法，该方法将将limit设置成position。并将position设为0，这就是Buffer的读写指针又移动到了开始位置。也就是说，Buffer调用了flip方法后，Buffer为输出数据做好了准备。

Buffer输出数据结束后，Buffer调用clear（）方法，把position位置设为0，将limit设为capacity。这样为再次向Buffer中装入数据做好准备。

（4）Buffer常用的函数：

int capacity():返回Buffer的capacity的大小。

boolean hasRemaining()：判读position和limit之间是否还有元素可供处理。

int limit():返回limit的位置。

Buffer limit(int newli):重新设置limit的值，并返回一个具有新的limit的缓冲区的对象。

Buffer mark()：设置Buffer中mark的位置，它只能在0和position之间。

int position():返回Buffer中position的值。

Buffer position(int newPs)：设置position的值，返回设置后的Buffer对象。

int remaining():返回当前位置和limit之间的元素个数。

Buffer reset():将position转到mark所在的位置。

Buffer rewind():将position设置为0，取消设置的mark。

两个重要的方法：

put()和get()，用于向Buffer中放入数据和从Buffer中取出数据。

public class BufferTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

CharBuffer buff = CharBuffer.allocate(8);

buff.put('a');

buff.put('b');

buff.put('c');

System.out.println("capacity:"+buff.capacity());

System.out.println("limit:"+buff.limit());

System.out.println("position:"+buff.position());

buff.flip();

System.out.println("flip()后，capacity:"+buff.capacity());

System.out.println("flip()后，limit:"+buff.limit());

System.out.println("flip()后，position:"+buff.position());

System.out.println("第一个元素："+buff.get());

System.out.println("取出第一个元素后，capacity:"+buff.capacity());

System.out.println("取出第一个元素后，limit:"+buff.limit());

System.out.println("取出第一个元素后，position:"+buff.position());

buff.clear();

System.out.println("clear()后，capacity:"+buff.capacity());

System.out.println("clear()后，limit:"+buff.limit());

System.out.println("clear()后，position:"+buff.position());

System.out.println("clear()后，position:"+buff.get(2));

} }

3、使用Channel

（1）Channel与传统的流对象的两个主要的区别：

Channel可以将文件的全部或部分直接映射成Buffer。

程序不直接和Channel交互，Channel和Buffer进行数据交互。

（2）Java为Chnanel接口提供了DatagramChannel、FileChannel、SelectableChanel、ServerSocketChannel、SocketChannel等实现类。

所有的Channel都不应该通过构造器来创建对象，而是通过传统的InputStream、OutputStream的getChannel（）方法来返回对应的Channel。

（3）Channel三个方法map()、read()、write()：

map():用于将Channel对应的部分或全部数据映射成ByteBuffer。

read():从Buffer中读取数据。

write():向Buffer中写入数据。

public class RandomFileChannelTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

File f = new File("a.txt");

try {RandomAccessFile raf = new RandomAccessFile(f,"rw");

FileChannel randomChannel = raf.getChannel();

MappedByteBuffer buffer = randomChannel.map(FileChannel.MapMode.READ\_ONLY, 0, f.length());

randomChannel.position(f.length());

randomChannel.write(buffer);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

public class FileChannelTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

File f = new File("src/class13/FileChannelTest.java");

try {

FileChannel inChannel = new FileInputStream(f).getChannel();

FileChannel outChannel = new FileOutputStream("a.txt").getChannel();

MappedByteBuffer buffer =inChannel.map(FileChannel.MapMode.READ\_ONLY, 0, f.length());

outChannel.write(buffer);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

4、字符集和Charset

（1）编码和解码。

通常，把明文的字符序列转化成计算机理解的二进制称为编码。

把二进制序列转化为能看懂的明文字符串称为解码。

Java1.4提供了Charset来处理字节序列和字符序列之间的转换关系。该类包含了用于创建解码器和编码器的方法，还提供了获取Charset所支持字符集的方法，availableCharsets（）静态方法，获取当前jdk所支持的所有字符集。

（2）一旦知道了字符集别名后，就可以调用Charset的forName（）方法来创建对应的Charset对象。

Charset cs = Charset.forName(“GBK”);

获得Charset对象后，就可以通过该对象的newDecoder()、newEncoder()，分别返回CharsetDecoder和CharsetEncoder对象。分别对应decode（）和encode（）。

public class CharsetTransform {

public static void main(String[] args) {

// TODO Auto-generated method stub

Charset cn = Charset.forName("GBK");

CharsetDecoder decoder = cn.newDecoder();

CharsetEncoder encoder = cn.newEncoder();

CharBuffer cbuff = CharBuffer.allocate(8);

cbuff.put('孙');

cbuff.put('悟');

cbuff.put('空');

cbuff.flip();

try {

ByteBuffer bbuff = encoder.encode(cbuff);

for (int i = 0; i < bbuff.capacity();i++){

System.out.println(bbuff.get(i));

}

CharBuffer cb = decoder.decode(bbuff);

System.out.println(cb);

} catch (CharacterCodingException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

public class CharsetTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

SortedMap<String,Charset> map = Charset.availableCharsets();

for (String al: map.keySet()) {

System.out.println(map.get(al));}}}

5、文件锁

使用文件可以有效的阻止多个进程并发修改同一个文件。

在NIO中，java提供了FileLock来支持文件锁定功能。在FileChannel中提供了lock和tryLock两个方法，可以获取文件锁FileLock对象。其中tryLock是非阻塞的。

lock（）和tryLock（）锁定文件的全部内容。

lock(long position,long size,boolean shared):对文件从position开始，长度为size的内容加锁。这个方法是阻塞式的。

tryLock(long position,long size,boolean shared): 对文件从position开始，长度为size的内容加锁。这个方法是非阻塞式的。

Shared为true时，是一个共享锁，允许多个进程读这个文件。当为false时，是一个排他锁，将锁定对文件的读写。处理完文件后通过FileLock的release

()方法释放文件锁。

public class FileLockTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

FileChannel channel = new FileOutputStream("a.txt").getChannel();

FileLock lock =channel.tryLock();

Thread.sleep(10000);

lock.release();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

**八、NIO2**

1、Path、Paths和Files

public class PathTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

Path path = Paths.get(".");

System.out.println("path里包含的路径数量："+path.getNameCount());

System.out.println("path绝对路径："+path.toAbsolutePath());

System.out.println("path绝对路径的根路径："+path.toAbsolutePath().getRoot());

System.out.println("path绝对路径包含的路径数量："+path.toAbsolutePath().getNameCount());

Path path2 = Paths.get("D:", "test","test");

System.out.println(path2);

}}

2、使用FileVisitor遍历文件和目录

（1）Files类提供了两个方法来遍历文件和子目录：

walkFileTree（Path start，FileVisitor<? super Path> visitor）:遍历start路径下所有的文件和子目录。

walkFileTree（Path start，Set<FileVisitOption> options,int maxDepth, FileVisitor<? super Path> visitor）: maxDepth最多遍历的深度。

FileVisitor代表文件的访问器，walkFileTree方法会自动遍历start路径下的所有文件和目录，遍历文件和子目录都会触发FileVisitor中相应的方法。

（2）FileVisitor定义了如下4个方法：

postVisitDirectory（T dir，IOException exc）：访问子目录后触发此方法。

preVisitDirectory（T dir，BasicFileAttributes atts）：访问目录前触发此方法。

visitFile（T file，BasicFileAttributes atts）：访问file文件时触发此方法。

visitFileFailed（T file，IOException exc）：访问文件失败后，触发此方法。

上面4个方法都会返回FileVisitResult，它是一个枚举类，代表访问之后的后续行为:

CONTINUE：代表“继续访问”的后续行为。

SKIP\_SIBLINGS:代表“继续访问”的后续行为，不访问该文件或目录的兄弟文件或目录。

SKIP\_SUBTREE: 代表“继续访问”的后续行为，不访问该文件或目录的子目录树。

TERMINATE：代表“终止访问”的后续行为。

public class FileVositorTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

Files.walkFileTree(Paths.get("E:", "soft","myEclipe","MyEclipse 10","workspace","JavaLau","src"),

new SimpleFileVisitor<Path>(){

public FileVisitResult visitFile(Path file,BasicFileAttributes attrs){

System.out.println("正在访问"+file+"文件");

if (file.endsWith("FileVositorTest.java")) {

System.out.println("已经找到目标文件");

return FileVisitResult.TERMINATE;

}

return FileVisitResult.CONTINUE;

}

public FileVisitResult preVisitDirectory(Path dir,BasicFileAttributes attrs){//访问路径

System.out.println("正在访问"+dir+"路径");

return FileVisitResult.CONTINUE;

}

}

);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

public class FilesTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

Files.copy(Paths.get("src/class13/FilesTest.java"), new FileOutputStream("s.txt"));

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

System.out.println("FilesTest.java是否为隐藏文件："+Files.isHidden(Paths.get("src/class13/FilesTest.java")));

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

List<String> lines = Files.readAllLines(Paths.get("src/class13/FilesTest.java"), Charset.forName("GBK"));

System.out.println(lines);

System.out.println(Files.size(Paths.get("src/class13/FilesTest.java")));

List<String> list = new ArrayList<>();

list.add("八一建军节");

list.add("八一建军节");

Files.write(Paths.get("list.txt"), list, Charset.forName("GBK"));

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

3、watchservice监控文件变化

path类提供了一个方法来监听文件系统的变化

register(watchservic watcher,watchevent.kind<?>…events)监听path代表的目录下的文件的变化，event指定要监听的事件

一旦使用了register方法完成注册后，就可以调用WatchService的如下三个方法来获取被监听目录的文件变化事件。

（1）poll():获取下一个WatchKey。

（2）poll(long timeout,TimeUnit unit):尝试等待timeout事件去获取一个WatchKey。

（3）take():获取下一个WatchKey，如果没有WatchKey发生就一直等待。

如果需要一直监听，应该选择take(),如果只需要在指定的事件端内监听，则可以考虑使用poll()。

public class WatchServiceTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

WatchService watchService = FileSystems.getDefault().newWatchService();

Paths.get("C:/1/2/").register(watchService, StandardWatchEventKinds.ENTRY\_CREATE,StandardWatchEventKinds.ENTRY\_MODIFY,StandardWatchEventKinds.ENTRY\_DELETE);

while(true){

try {

WatchKey key = watchService.take();

for (WatchEvent<?> event:key.pollEvents()) {

System.out.println(event.context() + "文件发生了"+event.kind()+"事件");}

boolean valid = key.reset();

if (!valid) {

System.out.println("key重设失败");

break;}

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();}}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

4、访问文件的属性

Java.nio.file.attribute包下提供了大量的工具类，通过这些工具类，可以比较简单的读取、修改文件的属性。

XxxAttributeView:代表某种文件属性的视图。

XxxAttributes:代表某种文件属性的集合，通过XxxAttributeView获得XxxAttributes。

**多线程**

**一、线程的概述**

1、进程和线程

运行的程序就变成了进程。进程具有一定的独立功能。是系统进行资源分配和调度的独立单位

三个特征：

独立性：进程是系统中独立存在的实体，拥有独立资源，私有地址空间，不能访问其他进程的地址空间。

动态性：程序时静态指令的集合，进程是动态的指令集。进程有生命周期和运行状态。

并发性：多个进程可以在单个处理器上并发执行，相互不影响。

（1）即使我们普通的电脑也支持多进程并发执行。

（2）cpu某个时间点，只处理一个任务。cpu按照时间片轮训任务。

（3）多线程扩充了多进程的概念。使得一个进程中可以处理多个任务。线程被称为了轻量级的进程，线程在程序中是独立的、并发执行。

（4）线程是进程的组成部分。一个进程可以有多个线程，一个线程必须存在于一个进程中。一个进程中的线程共享进程的资源。

（5）线程之间的关系，共享变量和环境，协同工作。相互独立，资源争抢。

（6）线程是由进程调度和管理的。

（7）一个程序运行后至少有一个进程，一个进程至少包含一个线程。

2、多线程的优势

（1）多线程并发程度高。

（2）多线程共享内存资源，极大的提高了系统的运行效率。

（3）多线程有更好的性能，一个进程中的线程共享代码段和数据。利用共享数据，可以进行线程间的通讯。

（4）java内置多线程功能的支持，简化了多线程编程。

**二**

1、继承Tread类创建线程

（1）定义Thread的子类，并重写该类的run方法。run方法体就是我们需要线程完成的任务。

（2）创建子类的实例，即创建线程的对象。

（3）用线程对象的start方法，启动线程。

2、示例

**public** **class** Fristthrad **extends** Thread{

**private** **int** i;

**public** **void** run(){

**for**(;i<100;i++){

System.***out***.println(**this**.getName()+"："+i);

}

}

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

**for**(**int** i=0;i<100;i++){

System.***out***.println(Thread.*currentThread*().getName()+":"+i);

**if**(i==20){

**new** Fristthrad().start();

Fristthrad fr= **new** Fristthrad();

fr.setName("xxx");

fr.start();

}

}

}

}

**public** **class** wqw **implements** Runnable {

**private** **int** i;

**public** **void** run() {

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

**for** (;i < 100;i++) {

System.***out***.println(Thread.*currentThread*().getName()+":"+i);

}

}

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

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

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println(Thread.*currentThread*().getName()+":"+i);

**if** (i == 20) {

wqw st = **new** wqw();

**new** Thread(st,"st1").start();

**new** Thread(st,"st2").start();

}

}}

}

3、实现Runnable接口创建线程类

Runnable接口创建多线程的步骤：

（1）定义Runnable接口的实现类，实现run方法。

（2）创建Runnable实现类的实例。该实例作为构建Thread类对象的参数。

（3）调用线程对象的start方法启动线程。

Runnable作为线程对象的target（要启动的线程对象）。

I是连续的，实例属性共享，target共享了。

4、两种方式创建线程的对比

|  |  |  |
| --- | --- | --- |
|  | Thread | Runnable |
| 是否允许再继承类 | 否 | 是 |
| 复杂度 | 简单：this获得当前线程 | 复杂thread.currntthread获得当前线程 |
| 共享属性 | 不能实现 | 可以，多线程处理一份任务 |

**三、线程的生命周期**

线程的声明周期：新建、就绪、运行、阻塞、死亡

1、新建：new，创建一个线程。和创建其他对象一样。jvm分配内存，初始化成员变量。此时没有线程的任何特征

2、就绪：start函数执行时线程处于就绪状态。jvm为其创建方法调用栈和程序计数器。线程还没有运行。运行开始取决于jvm的线程调度器

注意启动线程使用start。不要直接调用run方法。

当我们使用start方法启动线程时，系统会把run方法当作现成的执行体。直接调用run方法，run立刻执行，并且run返回前，其他线程无法并发执行

3、运行和阻塞

运行状态：处于就绪状态的线程获得了cpu的使用权，开始执行run方法的线程执行体。则该线程处于运行状态。

阻塞状态：（1）调用sleep方法，主动放弃cpu使用权（2）调用了一个阻塞式的io方法在io返回结果前线程被阻塞（3）试图获得同步监视器，但该同步监视器正在被其他线程调用（4）线程等待通知（5）线程被挂起，suspend（）

4、线程重新进入就绪状态（1）调用sleep方法后，经过了指定时间。

（2）阻塞io方法，返回结果（3）获得同步监视器（4）收到通知（5）结束挂起状态resume（）

5、死亡

（1）以下三种情况之一，线程死亡：

run执行完毕，线程正常结束。

Exception或error。

stop（）结束线程。容易产生死锁，不推荐使用。

（2）主线程结束，其他线程不受影响。

（3）isAlive方法，测试线程是否处于活动状态。

**public** **class** Fristthrad **extends** Thread{

**public** Fristthrad(String name){

**super**(name);

}

**public** **void** run(){

**for**(**int** i=0;i<100;i++){

System.***out***.println(**this**.getName()+"：:"+i);

}

}

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

**new** Fristthrad("new Thread").start();

**for**(**int** i=0;i<100;i++){

**if**(i==20){

Fristthrad j=**new** Fristthrad("jj Thrad");

j.start();

**try** {

j.join();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

}

**四、控制线程**

1、join函数让一个线程等待另一个线程执行完成

2、后台线程

在后台运行，后台线程的任务为其他线程提供服务。有守护线程，精灵线程等。jvm的垃圾回收线程是一个典型的后台线程。如果所有前台线程都死亡了，后台线程也会死亡。

3、线程睡眠

如果想让当前正在执行的线程暂停，进入阻塞状态。可以使用sleep方法

4、改变线程的优先级

优先级高执行机会多，优先级低执行机会少。线程的默认优先级由创建它的父线程相同。

Thread提供setPriority（int var）设置线程优先级，var取值范围1-10

getPriority（）获得线程的优先级。var的三个常用静态常量MIN\_PRIORITY:1

NORM\_ PRIORITY:5,MAX\_PRIORITY:10。

public class DaemonTread extends Thread{

public DaemonTread(String name) {

super(name);}

public void run() {

for (int i =0; i < 10000;i++) {

System.out.println(this.getName()+":"+i);

}}

public static void main(String[] args) {

// TODO Auto-generated method stub

DaemonTread dt = new DaemonTread("守护线程");

dt.setDaemon(true);

dt.start();

for (int i = 0;i < 10;i++) {

System.out.println(Thread.currentThread().getName()+"\*"+i);

}}}

public static void main(String[] args) {

// TODO Auto-generated method stub

Thread.currentThread().setPriority(6);

for (int i = 0; i < 30; i++) {

if (i == 10) {

TestPriority low = new TestPriority("低优先级线程");

System.out.println(low.getName()+"默认优先级："+low.getPriority());

low.setPriority(MIN\_PRIORITY);

low.start();

}

if (i == 20) {

TestPriority hight = new TestPriority("高优先级线程");

System.out.println(hight.getName()+"默认优先级："+hight.getPriority());

hight.setPriority(MAX\_PRIORITY);

hight.start();

}}}}

public class TestSleep {

public static void main(String[] args) {

// TODO Auto-generated method stub

for (int i = 0; i < 10; i++) {

System.out.println(System.currentTimeMillis());

try {

Thread.sleep(3000);

System.out.println(System.currentTimeMillis());

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}}

五、线程同步

1、线程安全问题

多线程从一个账户扣款，步骤：

（1）录入扣款金额（2）系统判断账户余额是否大于扣款金额（3）如果大于则扣款成功否则失败

定义3个类，账户类，线程类，测试类

2、同步代码块

上例中，问题的原因是run 方法不具有同步安全性，两个线程同时操作了庄户对象。为了解决此问题，java引入了同步监视器。使用同步监视器的通用方法就是同步代码块。操作如下

synchronized（obj）{} obj同步监视器

3、同步方法与同步代码块类似，使用synchronized修饰方法，不需要同步监视器，可以很方便的把类变成一个线程安全类

4、同步锁

使用更灵活

**public** **class** ACCOUNT {

**private** String accountNo;

**private** **double** balance;

**public** ACCOUNT(){

}

**public** ACCOUNT(String accountNo,**double** balance){

**this**.accountNo=accountNo;

**this**.balance=balance;

}

**public** String getAccountNo() {

**return** accountNo;

}

**public** **void** setAccountNo(String accountNo) {

**this**.accountNo = accountNo;

}

**public** **double** getBalance() {

**return** balance;

}

**public** **void** setBalance(**double** balance) {

**this**.balance = balance;

}

/\*public synchronized void take(String name,double takeMoney){//方法2

if(this.getBalance()>takeMoney){

setBalance(this.getBalance()-takeMoney);

System.out.println(""+name+"扣款成功，扣款金额="+takeMoney);

System.out.println(""+name+"扣款成功，扣款后余额="+this.getBalance());

}else{

System.out.println(name+"扣款失败，余额不足");

}

}\*/

/\*private final ReentrantLock lock=new ReentrantLock();

public void take(String name,double takeMoney){//方法3

lock.lock();try{

if(this.getBalance()>takeMoney){

setBalance(this.getBalance()-takeMoney);

System.out.println(""+name+"扣款成功，扣款金额="+takeMoney);

System.out.println(""+name+"扣款成功，扣款后余额="+this.getBalance());

}else{

System.out.println(name+"扣款失败，余额不足");

}}finally{

lock.tryLock();

}

}\*/

}

**public** **class** TakeThread **extends** Thread{

**private** ACCOUNT account;//自定义的类变量，可以调用这个类的方法，就是这个累的对象，引用变量传值会了🙃

**private** **double** takeMoney;

**public** TakeThread(String name,ACCOUNT account,**double** takeMoney){

**super**(name);

**this**.account=account;

**this**.takeMoney=takeMoney;

}

**public** ACCOUNT getAccount() {

**return** account;

}

**public** **void** setAccount(ACCOUNT account) {

**this**.account = account;

}

**public** **double** getTakeMoney() {

**return** takeMoney;

}

**public** **void** setTakeMoney(**double** takeMoney) {

**this**.takeMoney = takeMoney;

}

**public** **void** run(){

**synchronized**(account){

**if**(**this**.account.getBalance()>**this**.takeMoney){

account.setBalance(**this**.account.getBalance()-**this**.takeMoney);

System.***out***.println(""+**this**.getName()+"扣款成功，扣款金额="+**this**.takeMoney);

System.***out***.println(""+**this**.getName()+"扣款成功，扣款后余额="+**this**.getAccount().getBalance());

}**else**{

System.***out***.println(**this**.getName()+"扣款失败，余额不足");

}

}

}

/\*public void run(){//方法2

account.take(getName(), takeMoney);

}\*/

}

**public** **class** TestTake {

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

ACCOUNT ac=**new** ACCOUNT("123456",10000);

TakeThread th1=**new** TakeThread("房贷",ac,6000);

TakeThread th2=**new** TakeThread("车贷",ac,6000);

th1.start();

th2.start();

}

}

六、线程通讯

银行家算法之类的，pv操作

1、线程协调运行

需求：系统中由两个线程，分别代表存款和取款者，不断存取，不容许连续去两次

为了实现通讯，可以使用Object类的wait、notify、notifyAll方法。

七、线程池

系统启动一个新线程的成本是比较高的，因为涉及到和操作系统的交互。线程池可以提高性能，特别是对存在时间短的线程，更应该考虑线程池。

线程池在启动时创建大量的线程。程序讲一个runnable对象提交给线程池。线程池就会启动一条线程执行该对象的run方法，run结束后，线程不会消亡，而是回到线程池中成为空闲状态。等待执行下一个runnable对象的run方法

Exectors工厂类产生线程池：

（1）newCachedThreadPool（）创建一个具有缓存功能的线程池，系统根据需要创建线程，这些被创建的线程被缓存到线程池中

（2）newFixedThreadPool（int n）创建有固定个数线程的线程池

（3）newSingleThreadExecutor():创建只有一个单线程的线程池。

（4）newScheduledThreadPool(int n):创建具有指定线程个数的线程池，它可以指定延迟后执行任务。

（5）newSingleThreadScheduledExecutor():创建只有一个调线程的线程池，它可以在指定延迟后执行线程任务。

前三个方法返回ExecutorService线程池对象，可以执行runnable对象所代表的线程。后两个方法ScheduleExecutorService线程

使用线程池执行线程任务的步骤

（1）、通过Executors类静态工厂方法创建一个ExecutorService对象。

（2）、创建Runnable实现类的实例。作为线程执行任务。

（3）、调用ExecutorService对象的submit方法来提交Runnable实现类的实例。

（4）、当不想提交任务时，调用ExecutorService对象的shutdown方法关闭线程池。

八、线程相关类

1、ThreadLocal类

ThreadLocal类主要是针对共享资源读操作的情况。为每个使用共享变量的线程提供一个变量值的副本。每一个线程都可以独立改变自己的副本，而不会和其他线程副本冲突。从线程的角度看，线程完全拥有该变量。

ThreadLocal类的用法：

（1）T get():返回此线程局部变量中当前线程副本中的值。

（2）remove():删除此线程局部变量中当前线程的值。

（3）set(T value):设置此线程局部变量中当前线程副本中的值。

2、包装线程不安全的集合

（1）集合ArrayList、HashMap、HashSet都是线程不安全的。多线程向集合中放入元素时，可能会破坏元素的完整性。

（2）Collections类提供的静态方法，把这些集合包装成线程安全的集合。

网络编程

一、网络编程的基础知识

1、网络基础

计算机网路的定义：不不同地理位置的计算机和外部设备，通过通信线路互联起来，从而实现信息传递，共享硬件、软件、信息资源。

（1）、万维网，www，world wide web，全球信息网。

（2）、种类：按距离划分，广域网、城域网、局域网。

（3）、OSI（Open System Inerconnection）开放互联参考模型。七层模型：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

（4）、TCP/IP协议，四层模型：应用层、传输层、网络层、物理+数据链路层。

2、IP地址和端口

IP地址用于标识网络中的一个通讯实体。如计算机、打印机、存储等。

IP地址被分成了A、B、C、D、E，每个类别的网络标识和主机标识各有规则：

A：10.0.0.0-10.255.255.255

B: 172.16.0.0-172.31.255.255

C: 192.168.0.0-192.168.255.255

IP地址用于唯一标识网络上的通讯实体，但一个通讯实体可以有多个通讯程序提供网络服务，此时还需要端口进行区分。

端口是一个16位的整数，用于标识数据交给那个通讯程序处理。因此，端口是应用程序与外界进行交流的出入口。在同一台机器上两个程序不能使用同一个端口。端口的范围0-65535。三类：

（1）、公认端口：从0-1023，一些特定服务的端口。

（2）、注册端口：从1024-49151，应用程序通常会使用的端口。

（3）、动态或私有端口：从49152-65535，程序动态分配的端口。

二、java的基本网络支持

java对网络的支持放在java.net包下，该包下的url和urlconnection等类提供了以编程方式访问web服务的功能。urldecoder和urlencoder则提供了普通字符串和application/x-www-form-urlencoder MIME字符串相互转换的静态方法

1、使用inetaddress

java提供了inetaddress类代表ip地址，他有两个子类inet4address和inet6address分别代表ip4和ip6

没有构造器

方法：两个静态方法获取实例对象

getByName（String host）：根据主机获取对应的inetaddress对象

getByAddress（byte addr）：根据ip地址获取inetaddress

对象

inetaddress

实例的方法获取对应ip和主机名

getCanonicalHostName（）获取ip地址的全限定域名

getHostAddress（）返回inetaddress实例对应的ip地址字符串

getHostName（）获取ip地址的主机名

isReachable（）用于测试是否到达一个地址

**public** **class** TsetInetAddress {

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

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

**try** {

InetAddress ip=InetAddress.*getByName*("www.baidu.com");

System.***out***.println("是否可以到达百度"+ip.isReachable(2000));

System.***out***.println("百度ip是"+ip.getHostAddress());

InetAddress local=InetAddress.*getByAddress*(**new** **byte**[]{127,0,0,1});

System.***out***.println("本机是否可达"+local.isReachable(2000));

System.***out***.println(""+local.getCanonicalHostName());

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

2、使用URLDecoder和URLEncoder

URLDecoder和URLEncoder用于完成普通字符串和特殊字符串。

URLDecoder类，decode（Strings，String enc）静态方法，实现特殊字符串转化为普通字符串。

URLEncoder类，encode（Strings ，String enc ）将普通字符串转化为特殊字符串

3、使用URL和URLConnection

URL（uniform resource locator）统一资源定位器，互联网资源的指针。url由协议名，主机，资源组成

URL（Strings）s是一个url的字符串。

常用方法

getFile（）获取url的资源名

getHost（）主机

gtPath（）路径

getPort（）端口号

getProtocol（）协议名‘

openConnection（）返回一个urlconnection对象，表示url所引用的远程对象链接

openStream（）打开此url的链接返回一个用于读取该url资源的Inputstream

三、基于tcp协议的网络编程

1、TCP协议的基础

IP协议，internet Protocal，简称IP协议。

两台计算机能够进行通讯，就必须使用相同的语言或协议，IP协议就是此通用协议。IP协议能够保证计算机发送和接收分组数据。

TCP协议，（Transmission Control Protocol）传输控制协议。是一种面向连接的、可靠的、基于字节流的传输层通信协议，当两台计算机发送、接收数据前，该协议负责建立连接，连接是用于发送和接收数据的虚拟链路。

三次握手、重发机制。

虽然IP和TCP协议功能不尽相同，也可以分开使用。但只有二者联合使用，才能保证Internet在复杂的环境下正常工作。

2、使用ServerSocket创建TCP服务器端

两端，服务器端、客户端。

java中提供了能够接收其他通讯实体连接请求类：ServerSocket。

accept():监听来自客户端的请求。如果收到来自客户端的Socket连接请求，该方法返回一个与客户端Socket对应的Socket。

构造器：

ServerSocket（int port）：指定的端口port来创建一个ServerSocket对象。

ServerSocket（int port，int backlog）：指定的端口port和连接队列长度来创建一个ServerSocket对象。

ServerSocket（int port，int backlog，InetAddress localaddr）：指定的端口port和连接队列长度、指定本机的IP，来创建一个ServerSocket对象。

3、使用Socket进行通信

客户端可以使用Socket构造器来连接到指定服务器，构造器如下：

Socket（InetAddress/String remoteAddress,int port）：创建连接到远程主机、远程端口的socket对象。

Socket（InetAddress/String remoteAddress,int port，InetAddress loAdd,int locpot）：创建连接到远程主机、远程端口的socket对象。指定本机的ip和端口。

Socket s = new Socket(“127.0.0.1”,端口)。连接到服务器端后，客户端和服务器端就产生了一对相互连通的socket。

产生一对socket后，就可以通过socket进行通讯，socket通过两个方法获得输入流和输出流：

InputStream getInputStream();

OutputStream getOutputStream();

4、加入多线程

聊天室例子

常用字符接口

**package** class15.talk;

**public** **interface** Constant {

//特殊字符标识的长度

**int** ***LEN*** = 2;

//标识公聊信息

String ***PUBLIC\_TALK*** = "$}";

//标识私聊信息

String ***PRIVATE\_TALK*** = "@(";

//用户注册信息

String ***USER\_NAME\_REGISTE*** = "!\*";

//字符串拆分标志

String ***SPIT\_SIGN*** = "#";

//登录成功标志

String ***LOGIN\_SUCCESS*** = "1";

//注册失败标志

String ***NAME\_REGISTER\_FAIL*** = "-1";

}

map类

**package** class15.talk;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** EMAP<K,V> **extends** HashMap<K,V>{

/\*\*

\* 根据value值删除map中项

\* **@param** value

\*/

**public** **void** removeByValue(Object value) {

**for** (Object key:keySet()) {

**if** (get(key) == value) {

remove(key);

**break**;

}

}

}

/\*\*

\* 获取map的value Set集合

\* **@return**

\*/

**public** Set<V> valueSet(){

Set<V> result = **new** HashSet<V>();//hashset只能存放不重复的值

**for** (K key : keySet()) {

result.add(get(key));

}

**return** result;

}

/\*\*

\* 通过value获得map的key值

\* **@param** value

\* **@return**

\*/

**public** K getKeyByValue(V value){

**for** (K key:keySet()) {

**if** (get(key).equals(value)

&& get(key) == value) {

**return** key;

}

}

**return** **null**;

}

**public** V put(K key,V value){//根据v判断添加map值

**for** (V val:valueSet()) {

**if** (val.equals(value)) {

**throw** **new** RuntimeException("EMAP实例中不允许有重复的value");

}

}

**return** **super**.put(key, value);

}

}

客户端

**package** class15.talk;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintStream;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**import** javax.swing.JOptionPane;

**public** **class** MyClient {

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

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

Socket socket = **null**;

BufferedReader br = **null**;

BufferedReader brServer = **null**;

PrintStream ps = **null**;

**try** { socket = **new** Socket("127.0.0.1",30000);

ps = **new** PrintStream(socket.getOutputStream());

brServer =**new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));

String tip = "";

**while**(**true**) {

String userName = JOptionPane.*showInputDialog*(tip+"输入用户名");

ps.println(Constant.USER\_NAME\_REGISTE+userName+Constant.USER\_NAME\_REGISTE);

String result = brServer.readLine();

**if** (Constant.NAME\_REGISTER\_FAIL.equals(result)) {

tip = "用户名重复，请重新";

**continue**;

}

**if** (Constant.LOGIN\_SUCCESS.equals(result)) {

**break**;

}

}

**new** Thread(**new** ClientThread(brServer)).start();

String line = **null**;

br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

**while**((line = br.readLine()) != **null**) {

**if** (line.startsWith("/")&&line.indexOf(":")>0) {

line = line.substring(1);

line = Constant.PRIVATE\_TALK+

line.split(":")[0]+Constant.SPIT\_SIGN+

line.split(":")[1]

+Constant.PRIVATE\_TALK;

ps.println(line);

} **else** {

ps.println(Constant.PUBLIC\_TALK+line+Constant.PUBLIC\_TALK);

}

}

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally**{

**try** {

br.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

socket.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

brServer.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

ps.close();

}

}

}

客户端线程

**package** class15.talk;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**public** **class** ClientThread **implements** Runnable {

BufferedReader br = **null**;

**public** ClientThread(BufferedReader br) {

**this**.br = br;

}

@Override

**public** **void** run() {

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

String content = **null**;

**try** {

**while** ((content = br.readLine()) != **null**) {

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

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

服务器

**package** class15.talk;

**import** java.io.IOException;

**import** java.io.PrintStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.util.ArrayList;

**public** **class** MyServer {

//将用户名和打印流按键值对存放

**public** **static** EMAP<String,PrintStream> *clients* = **new** EMAP<>();

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

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

**try** {

ServerSocket ss = **new** ServerSocket(30000);

**while** (**true**) {//一直监听客户端请求

Socket s = ss.accept();

//启动线程

**new** Thread(**new** ServerThread(s)).start();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

服务器线程

**package** class15.talk;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintStream;

**import** java.net.Socket;

**public** **class** ServerThread **implements** Runnable {

Socket s = **null**;

BufferedReader br = **null**;

**public** ServerThread(Socket s){//线程构造器

**this**.s = s;

**try** {

br = **new** BufferedReader(**new** InputStreamReader(s.getInputStream()));//br存放socket流信息

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

/\* (non-Javadoc)

\* @see java.lang.Runnable#run()

\*/

@Override

**public** **void** run() {

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

String content = **null**;

PrintStream ps = **null**;

**try** {

ps = **new** PrintStream(s.getOutputStream());

**while** ((content = readFromClient()) != **null**) {

//用户注册信息判断

**if** (content.startsWith(Constant.USER\_NAME\_REGISTE) &&

content.endsWith(Constant.USER\_NAME\_REGISTE)) {

String userName = getRealMsg(content);

//判断用户名是否重复

**if** (MyServer.clients.containsKey(userName)) {

ps.println(Constant.NAME\_REGISTER\_FAIL);//返回注册失败标志

} **else** {

ps.println(Constant.LOGIN\_SUCCESS);//返回注册成功标志

MyServer.clients.put(userName, ps);//将用户和对应的流保存在map中

}

} **else** **if** (content.startsWith(Constant.PRIVATE\_TALK) &&

content.endsWith(Constant.PRIVATE\_TALK))//私聊

{

String userAndMsg = getRealMsg(content);

String user = userAndMsg.split(Constant.SPIT\_SIGN)[0];

String msg = userAndMsg.split(Constant.SPIT\_SIGN)[1];

MyServer.clients.get(user).println(

MyServer.clients.getKeyByValue(ps)+":"+msg);

} **else** {//公聊

String msg = getRealMsg(content);

**for** (PrintStream cps:MyServer.clients.valueSet()) {

cps.println(MyServer.clients.getKeyByValue(ps)+":"+msg);

}

}

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

MyServer.clients.removeByValue(ps);//删除出错的流对应的map值

} **finally**{

**try** {

s.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

br.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

ps.close();

}

}

**private** String readFromClient(){

**try** {

**return** br.readLine();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** **null**;

}

**private** String getRealMsg(String line){

**return** line.substring(Constant.LEN,line.length() - Constant.LEN);

}

}

四、UDP编程

UDP协议是一种不可靠的网络协议，它在通讯实例的两端各建立一个Socket，但是没有虚拟链路的概念，两端的Socket知识发送、接收数据包的对象。Java提供了DatagramSocket对象作为UDP协议的Socket。

1、UDP协议基础

UDP协议英文全称USER Datagram Protocal，用户数据包协议。优势：传输速度快。适用于实时性要求较高的场景，如：网络游戏、视频会议。

建立在IP（网络层）层上，UDP和TCP都是传输层的协议。

UDP传输效率高、可靠性低，传输的大小能达到64K。

2、适用DatagramSocket发送、接收数据

DatagramSocket的作用就是发送和接收数据包，java适用DatagramPacket来代表数据包。

DatagramSocket的构造器

DatagramSocket():创建实例，绑定到本机的默认IP，随机端口。

DatagramSocket(int port):创建实例，绑定到本机的默认IP，指定端口。

DatagramSocket(int port,InetAddress iaddr):创建实例，绑定到本机的指定IP，指定端口。

创建DatagramSocket实例后，就可以调用数据发送和接收数据的方法：

receive(DatagramPacket p):接收数据。

send(DatagramPacket p):发送数据。

DatagramPacket构造器

DatagramPacket(byte[] buf,int length):构造实例，用来接收长度为length的数据包。

DatagramPacket(byte[] buf,int length,InetAddress address, int port):构造实例，用来将长度为length的数据包，发送到指定主机、指定端口。

DatagramPacket(byte[] buf,int offset,int length):构造实例，用来接收长度为length的数据包，在缓冲区中指定了偏移量。

DatagramPacket(byte[] buf, int offset ，int length,InetAddress address, int port):构造实例，用来将长度为length偏移量为offset的数据包，发送到指定主机、指定端口。

DatagramPacket(byte[] buf, int offset ，int length,SocketAddress address):构造实例，用来将长度为length偏移量为offset的数据包，发送到指定主机、指定端口。

DatagramPacket(byte[] buf，int length,SocketAddress address):构造实例，用来将长度为length的数据包，发送到指定主机、指定端口。

程序可以通过调用DatagramPacket的方法获得发送者的ip和端口：

getAddress():返回某台机器的ip地址，数据包可以发送到该ip的机器，或者从该机器接收数据。

getPort():返回某台主机的端口号，当前数据包将要发送到该主机或者从该主机收到的。

getSocketAddress()。

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.SocketException;

public class UdpServer {

private static final int PORT = 30000;

private static final int DATA\_LEN = 4096;

byte[] inBuff = new byte[DATA\_LEN];

private DatagramPacket inPacket = new DatagramPacket(inBuff,inBuff.length);

private DatagramPacket outPacket = null;

String[] books = new String[]{"三国演义","水浒传","西游记","红楼梦"};

public void init() {

DatagramSocket socket = null;

try {

socket = new DatagramSocket(PORT);

for (int i = 0; i < 1000; i++) {

socket.receive(inPacket);

System.out.println(new String(inBuff,0,inPacket.getLength()));

byte[] sendData = books[i%4].getBytes();

outPacket = new DatagramPacket(sendData,sendData.length,

inPacket.getSocketAddress());

socket.send(outPacket);}

} catch (SocketException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} finally {

socket.close(); }}

public static void main(String[] args) {

// TODO Auto-generated method stub

new UdpServer().init();

}}

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;

import java.util.Scanner;

public class UdpClient {

private static final int DEST\_PORT = 30000;

private static final String DEST\_IP = "127.0.0.1";

private static final int DATA\_LEN = 4096;

byte[] inBuff = new byte[DATA\_LEN];

private DatagramPacket inPacket = new DatagramPacket(inBuff,inBuff.length);

private DatagramPacket outPacket = null;

public void init(){

DatagramSocket socket = null;

try {

socket = new DatagramSocket();

outPacket = new DatagramPacket(new byte[0],0,InetAddress.getByName(DEST\_IP),DEST\_PORT);

Scanner scan = new Scanner(System.in);

while(scan.hasNextLine()) {

byte[] buff = scan.nextLine().getBytes();

outPacket.setData(buff);

socket.send(outPacket);

socket.receive(inPacket);

System.out.println(new String(inBuff,0,inPacket.getLength())); }

} catch (SocketException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (UnknownHostException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();} finally{

socket.close();

}}

public static void main(String[] args) {

// TODO Auto-generated method stub

new UdpClient().init();

}

}

反射机制

java.long.flect包下的接口、类包含class，method，fileld、constructor、array等，这些类分别代表了，类，方法、成员变量，构造器，数组。java程序可以使用这些类动态的获得某个类或对象运行时的信息，并可以动态的创建java对象，动态的调用java方法访问并修改指定对象的成员变量

一、类的加载机制和反射

系统可以在第一次使用某个类时加载该类，也可以采用预加载机制加载某个类。

1、jvm和类

（1）当Java的某个程序运行时，将会启动一个java虚拟机进程，不管程序多复杂，包含多少个线程，他们都处于java虚拟机中，当系统出现下列情况时jvm进程被终止

程序运行到最后，正常退出

遇到未捕获的异常或错误

程序运行到system.exit()或runtime.getruntime.exit（）

强制结束jvm

（2）java程序运行结束

2、类的加载

（1）当程序调用某个类时，如果该类在内存中不存在，则系统会通过加载、连接、初始化三个步骤对该类进行初始化。

（2）类的加载指的是将类的class文件读入到内存中，并为之创建一个java.lang.Class的对象。

（3）类的加载是由类的加载器完成，类的加载器由jvm提供，这些类的加载器是程序运行的基础，其中jvm提供的类加载器称为系统类加载器。我们可以通过继承ClassLoader基类创建自己的类加载器。

3、类的连接

当类加载完成后，系统为之生成了一个Class对象，接着将会进入连接阶段，连接阶段负责把类二级制数据合并到JRE中。大致分为三个阶段：

验证：检查被加载的类是否有正确的内部结构，并和其他类协调一致。

准备：为类的变量分配内存，并设置初试值。

解析：将类二级制数据中的符号引用替换成为直接引用。

4、类的初始化

（1）类的初试化阶段，主要是对类变量进行初试化。两种方式：声明时初始化。在类静态块中初试化。

（2）初始化的步骤

先加载并连接该类。

先初始化其直接父类。如果父类还有父类依次初始化，jvm总是先初试化java.lang.Object。

顺序执行初始化的语句。

5、类的初始化的时机

（1）程序中类、接口会被初始化的情况：

创建类的实例。new、反射、反序列化。

调用类的方法。

访问类或接口的变量。

使用反射机制来强制创建接口或类的java.lang.Class对象。Class.forName(“Person”);

初始化某个类的子类。

直接使用java.exe运行主类。

二、类的加载器

1、类的加载器简介

（1）一个类一旦别加载到内存中，同一个类就不会重复加载。

（2）jvm启动时，三种类加载器：

Bootstrap ClassLoader:根类加载器

又称为引导类加载器，加载java的核心类。

Extension ClassLoader：扩展类加载器

它负责加载JRE的扩展目录（%JAVA\_HOME%/jre/lib/ext）中的jar包中的类。

System ClassLoader：系统类加载器。

加载classpath选项，java.class.path系统属性，或classpath环境变量所指定的jar包和类路径。

2、类加载机制

Jvm类加载机制有三种：

全部加载，该类所依赖和引用的所有类。

父类委托，先让父类加载器加载该类。

缓存机制，所有被加载的类都被缓存，使用类时，先从缓存中查找。类如果被修改后需要重新加载类，重启jvm才会生效。

3、URLClassLoader类

（1）java为ClassLoader提供了一个URLClassLoader实现类，同时该类也是系统类加载器和扩展类加载器的父类。功能强大，可以从本地和远程主机上加载二进制文件class。

构造器：

URLClassLoader(URL[] urls):使用默认的父类加载器创建一个ClassLoader对象，该对象从urls所指定的系列路径来查询并加载类。

URLClassLoader(URL[] urls，ClassLoader parent):使用指定的父类加载器创建一个ClassLoader对象，该对象从urls所指定的系列路径来查询并加载类。

一旦获得了URLClassLoader对象后，就可以调用对象的loadClass（）方法加载指定类。

三、通过反射查看类的信息

Java程序中许多对象在运行时都会出现两种类型：编译时的类型和运行时的类型，Peron p = new Student();更为极端的情况，程序在运行时接收到外部传入一个对象，该对象的编译时类型是Object，但程序又需要调用该对象运行时类型的方法。

程序需要在运行时发现对象和类的真实信息，两种做法：

编译时和运行时都知道对象和类的真实信息，instanceof判断，强制转化为运行时类型的变量即可。

编译时无法知道该对象和类可能属于那些类，程序依靠运行时的信息来发现该对象和类的真实信息。需要使用反射机制。

1、获得Class对象

（1）类被加载到内存中，会生成一个Class对象，通过Class对象就能访问到jvm中的这个类。获得Class对象三种方法：

Class类forName（String s）静态方法，参数s包含类的全限定名称。

调用某个类的Class属性。Person.class，返回Person类对应Class对象。

调用某个对象的getClass()方法。Object类的方法。

一旦获得了Class后，就可以调用Class对象的方法获得该对象和类的真实信息。

2、从Class中获取信息

（1）从Class对象中获取其对应类的详细信息：

获取类的构造器

Constructor [] getConstructors():返回一个包含某些Constructor对象的数组，这些对象所反映此Class对象所表示的类的所有公共构造器方法。

Constructor<T> getConstructor(Class… parameterType):返回一个Constructor对象，它反映此Class对象所表示的类的指定公共构造方法。parameterType参数是Class对象的一个数组。这些Class对象按照声明顺序标识构造方法的形参类型。

Constructor [] getDeclaredConstructors():返回一个包含某些Constructor对象的数组，这些对象所反映此Class对象所表示的类的所有构造器方法。

获得类的方法

Method getMethod(String name,Class… parameterType)

返回Method对象，它反映此Class对象所表示的类或者接口的指定公共成员方法。name参数是一个String，所需要的方法简称，parameterType参数是按声明顺序标识该方法的形参。是一个Class对象的数组。

Method[] getMethod():返回一个包含某些Method对象的数组，这些对象反映此Class对象所表示的类或接口的公共成员方法。

获取类的成员变量

Field getField(String name):

返回一个Field对象，它反映此Class对象所表示的类或接口的指定公共成员变量。name指定所需字段的简称。

Field[] getField():

返回一个Field对象的数组，它反映此Class对象所表示的类或接口的所有公共成员变量。

获取类的修饰符、包、类名等基本信息。

int getModifiers():返回此类或接口以整数编码的java语言修饰符。

Package getPackage():获取此类的包。

String getName():以String的形式返回此Class对象所表示的实体名称。

判断该类是否为接口、枚举等类型。

isArray():判断此Class对象是否表示一个数组类。

isEnum():判断此Class对象是否表示一个枚举类。

isInterface():判断此Class对象是否表示一个接口。

四、使用反射生成对象并操作对象

1、创建对象

（1）通过反射生成对象有两种方式：

使用class对象的newInstance()方法来创建该class对象对应类的实例。Class对象对应的类必须有默认的构造器。

先使用Class对象获得Constructor对象，再调用构造器对象的newInstance()方法来创建Class对象对应类的实例。可以指定构造器。

2、调用方法

（1）获得Method对象后可以通过该对象调用它的方法：

Object invoke（Object obj，Obeject… args）:对带有指定参数的指定对象调用由此Method对象表示的方法。

public class BootstrapTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs();

for (int i = 0;i < urls.length;i++) {

System.out.println(urls[i].toExternalForm());

}

}

}

public class ClassLoaderPropTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

ClassLoader systemLoader = ClassLoader.getSystemClassLoader();

System.out.println("系统类加载器："+systemLoader);

try {

Enumeration<URL> em1 = systemLoader.getResources("");

while (em1.hasMoreElements()) {

System.out.println(em1.nextElement());

}

ClassLoader extensionLoader = systemLoader.getParent();

System.out.println("父类类加载器："+extensionLoader);

System.out.println("父类类加载器加载路径："+System.getProperty("java.ext.dirs"));

System.out.println("祖父类类加载器："+extensionLoader.getParent());

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

public class URLClassLoaderTest {

private static Connection conn;

public static Connection getConn(String url,String user,String password){

if (conn == null) {

try {

URL[] urls = {new URL("file:mysql-connector-java-5.1.5-bin.jar")};

URLClassLoader myClassLoader = new URLClassLoader(urls);

Driver diver =

(Driver)myClassLoader.loadClass("com.mysql.jdbc.Driver")

.newInstance();

Properties props = new Properties();

props.setProperty("user", user);

props.setProperty("password", password);

conn = diver.connect(url, props);

return conn;

} catch (MalformedURLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InstantiationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

} else {

return conn;

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Connection conn =

getConn("jdbc:mysql://localhost:3306/mysql","root","root");

System.out.println(conn);

}

}

public class CreateJFrame {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

Class<?> jframeCla = Class.forName("javax.swing.JFrame");

Constructor<?> ctor = jframeCla.getConstructor(String.class);

JFrame obj = (JFrame)ctor.newInstance("测试窗口");

System.out.println(obj);

obj.pack();

obj.setVisible(true);

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ReflectiveOperationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}}}

public class ClassTest {

private ClassTest(){

}

public ClassTest(String name){

System.out.println("执行有参数的构造器");

}

public void info(){

System.out.println("执行无参数的info方法");

}

public void info(String str){

System.out.println("执行有参数的info方法，其str参数值："+str);

}

class Inner{

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Class clazz = ClassTest.class;

Constructor[] ctors = clazz.getDeclaredConstructors();

System.out.println("ClassTest的全部构造器如下：");

for (Constructor c:ctors) {

System.out.println(c);

}

Constructor[] ctors1 = clazz.getConstructors();

System.out.println("ClassTest的全部公共构造器如下：");

for (Constructor c:ctors1) {

System.out.println(c);

}

Method[] mtds = clazz.getMethods();

System.out.println("ClassTest的全部公共方法如下：");

for (Method m:mtds) {

System.out.println(m);

}

try {

System.out.println("ClassTest的一个带有参数的info方法如下："

+clazz.getMethod("info", String.class));

} catch (NoSuchMethodException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

Class[] inners = clazz.getDeclaredClasses();

System.out.println("ClassTest的全部内部类如下：");

for (Class c:inners) {

System.out.println(c);

}

try {

Class inCla = Class.forName("class16.ClassTest$Inner");

System.out.println("Inner类的外部类："+inCla.getDeclaringClass());

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("ClassTest类的包："+clazz.getPackage());

System.out.println("ClassTest类的父类："+clazz.getSuperclass());

}

}

public class ExtendObjectPoolFactory {

private Map<String,Object> objectPool = new HashMap<String,Object>();

private Properties config = new Properties();

public void init(String fileName){

try {

FileInputStream fis = new FileInputStream(fileName);

config.load(fis);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

private Object createObject(String clazName){

Class<?> clazz = null;

try {

clazz = Class.forName(clazName);

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

return clazz.newInstance();

} catch (InstantiationException | IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

public void initPool() {

for (String name:config.stringPropertyNames()) {

if (!name.contains("%")) {

objectPool.put(name, createObject((String)config.get(name)));

}

}

}

public void initProperty(){

for (String name:config.stringPropertyNames()) {

if (name.contains("%")) {

String[] objAndProp = name.split("%");

Object target = objectPool.get(objAndProp[0]);

//a%title=Test Title setTitle(String ss)

String medName = "set"+objAndProp[1].substring(0,1).toUpperCase()

+objAndProp[1].substring(1);

Class<?> targetClass = target.getClass();

try {

Method mtd = targetClass.getMethod(medName, String.class);

try {

mtd.invoke(target, config.getProperty(name));

} catch (IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalArgumentException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InvocationTargetException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

} catch (NoSuchMethodException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

// TODO Auto-generated method stub

ExtendObjectPoolFactory epf = new ExtendObjectPoolFactory();

epf.init("extObj.txt");

epf.initPool();

epf.initProperty();

System.out.println(epf.objectPool.get("a"));

JFrame j = (JFrame)epf.objectPool.get("a");

j.pack();

j.setVisible(true);

}}

public class ObjectPoolFactory {

private Map<String, Object> objectPool = new HashMap<String,Object>();

private Object createObject(String name){

Class<?> cla = null;

try {

cla = Class.forName(name);

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

return cla.newInstance();

} catch (InstantiationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

public void initPool(String fileName) {

FileInputStream fis = null;

try {

fis = new FileInputStream(fileName);

Properties props = new Properties();

props.load(fis);

for (String name:props.stringPropertyNames()) {

objectPool.put(name, createObject(props.getProperty(name)));

}

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} finally {

try {

fis.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

ObjectPoolFactory pf = new ObjectPoolFactory();

pf.initPool("obj.txt");

System.out.println(pf.objectPool.get("a"));

System.out.println(pf.objectPool.get("b"));

}

}