java note

java note1 基础

java语言概述

历史

- 是SUN(Stanford University Network，斯坦福大学网络公司) 1995年推出的一门高级编程语言。
- 重要的版本变更:2014年，发布JDK8.0，是继JDK5.0以来变化最大的版本

术语

- Java SE(Java Standard Edition)标准版
- Java EE(Java Enterprise Edition)企业版
- Java ME(Java Micro Edition)小型版
- JVM (Java Virtual Machine)Java虚拟机
- GC(Garbage Collection)垃圾收集机制
- JDK(Java Development Kit)Java开发工具包
- JRE(Java Runtime Environment)Java运行环境
- JDK = JRE + 开发工具集（例如Javac编译工具等）
- JRE = JVM + Java SE标准类库

应用:

- 企业级应用,Android平台应用,大数据平台开发

特性

- 面向对象:两个基本概念：类、对象,三大特性：封装、继承、多态
- 健壮性:吸收了C/C++语言的优点，但去掉了其影响程序健壮性的部分（如指针、内存的申请与释放等）,提供了一个相对安全的内存管理和访问机制
- 跨平台性:JVM实现一次编写,到处运行

步骤

- 编写:1. 将Java代码编写到扩展名为.java的文件中。
- 编译:编译通过javac命令对该java文件进行编译,生成class文件。如 `javac HelloWorld.java`
- 运行:通过java命令对生成的class文件进行运行。如 `java HelloWorld`

第一个HelloWorld程序

```
//HelloWorld.java源文件
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println(“Hello World!”);
    }
}
```

注意:

- 在一个Java源文件中可以声明多个class,但最多只有一个类声明为public的,且该类的类名就是源文件名.
- 程序的入口就是main方法,格式是固定的,即 `public static void main(String[] args)`
- 编译后会生成一个或多个class文件,运行的时候运行与源文件名相同的.
- 输出语句 `System.out.println()` 和 `System.out.print()`
- 单行注释: `//` 多行注释 `/* */` 文档注释 `/** */` (可以被javadoc解析,有@Description,@author,@date,@param,@return@version等参数)

变量和运算符

关键字和保留字

- 关键字:被Java语言赋予了特殊含义，用做专门用途的字符串（单词）关键字中所有字母都为小写
- 保留字:现有Java版本尚未使用，但以后版本可能会作为关键字使用。

标识符

- 标识符：对各种变量、方法和类等要素命名时使用的字符序列

合法标识符规则：

- 由26个英文字母大小写，0-9 ，_或 $ 组成
- 数字不可以开头。
- 不可以使用关键字和保留字，但能包含关键字和保留字。
- Java中严格区分大小写，长度无限制。
- 标识符不能包含空格。

Java中的名称命名规范：

- 包名：多单词组成时所有字母都小写：xxxyyyzzz
- 类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz
- 变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz
- 常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

变量

- 变量是程序中最基本的存储单元。包含变量类型、变量名和存储的值

注意:

- Java中每个变量必须先声明，后使用
- 变量的作用域：其定义所在的一对{ }内,变量只有在其作用域内才有效
- 同一个作用域内，不能定义重名的变量
- 声明变量,如 `int var;`
- 变量赋值,如 `var = 10;`
- 声明和赋值变量,如 `int var = 10;`

分类:

- 基本数据类型:
  - 数值型
    - 整数类型:byte(1字节),short(2字节),int(4字节),long(8字节)
    - 浮点类型:float(4字节),double(8字节)
  - 字符型:char(2字节)
  - 布尔类型:boolean
- 引用数据类型
  - 类
  - 接口
  - 数组

数值型

- 声明long型常量须后加‘l’或‘L’
- Java程序中变量通常声明为int型，除非不足以表示较大的数，才使用long
- 声明float型常量，须后加‘f’或‘F’。
- Java 的浮点型常量默认为double型

字符类型

- Java中的所有字符都使用Unicode编码，故一个字符可以存储一个字母，一个汉字，或其他书面语的一个字符。
- 字符型变量的三种表现形式：
  - 字符常量是用单引号(‘ ’)括起来的单个字符,如： `char ch = 'a'`
  - Java中还允许使用转义字符‘\’来将其后的字符转变为特殊字符型常量。 `char ch = '\n'`
  - 直接使用 Unicode 值来表示字符型常量：’\uXXXX’。其中，XXXX代表一个十六进制整数。如： `\u000a` 表示 `\n` 。
- 了解:ASCII码一共规定了128个字符的编码,Unicode将世界上所有的符号都纳入其中,UTF-8 是在互联网上使用最广的一种 Unicode 的实现方式。

布尔类型

- 只有true,false两个值,用于条件判断

类型转换

自动类型转换:

- 数的范围只能从小到大转换:byte,char,short->int->long->float->long
- byte,char,short之间不能相互转化,只能转化为int
- boolean类型不能与其他数据类型运算
- 整型常量默认为int,浮点常量默认为double型

强制类型转换:

- 使用强转符()截断操作,可能有精度损失

字符串类型String(类)

- 定义:String str = “abc”
- 当把任何基本数据类型的值和字符串(String)进行连接运算时(+)，基本数据类型的值将自动转化为字符串(String)类型。

进制

对于整数，有四种表示方式：

- 二进制(binary)：0,1,以0b或0B开头
- 十进制(decimal)：0-9
- 八进制(octal)：0-7,以数字0开头表示
- 十六进制(hex)：0-9及A-F,以0x或0X开头表示

运算符

算数运算符

`+ - * / % ++ --`
注意:

- /返回的结果是int型,除非其中一个运算数是double
- mod的正负号与被模数相同
- 自增或自减不改变数据类型

赋值运算符

`= += -= *= /= %=`

- 同样不会改变数据类型

比较运算符

`== != < > <= >= instanceof`

- 结果是布尔类型
- instanceof用于判断一个对象是否为一个类的实例,如 `boolean result = obj instanceof Class`

逻辑运算符

`& && | || ! ^`

- &&和||会短路检测,&和|不会

位运算符

`<< >> >>> & | ^ ~`

- num1 ^ num2 ^ num2 = num1

三元运算符

`<条件> ? <返回值1> : <返回值2>`

- 可以改写为if-else

运算符的优先级

- 用()即可

流程控制

顺序结构

从上到下进行

条件结构

- if-else

```
if(<条件表达式>) {

}
else if {

}
...
else {

}
```

- switch-case

```
switch(<表达式>) {
  case 常量1: ... break;
  case 常量2: ... break;
  ...
  default: ... break;
}
```

- 表达式中是byte,short,int,char,枚举类型,字符串

循环结构

- for

```
for (<初始化>; <循环条件>; <迭代条件>) {
  <循环体>
}
```

- while

```
<初始化>
while (<循环条件>) {
  <循环体>
  <迭代条件>
}
```

- do-while

```
<初始化>
do {
  <循环体>
  <迭代条件>
} while (循环条件)
```

特殊关键字：break(跳出当前循环),continue(跳出当前循环次数)

- 这些关键字后面不能声明语句
- 在循环前可以添加标签,如 `label:` ,用break <标签>(或continue <标签>)可以跳到添加标签的循环

输入

从键盘获取不同类型的数值

1. 导包 `import java.util.Scanner;`
2. 实例化 `Scanner scanner = new Scanner(System.in);`

3. 调用相关方法 `int num = scanner.nextint();`

- 其他方法:next(接收字符串) nextDouble(接受浮点数) nextBoolean(接受布尔型true或false)
- 输入数据类型与要求的类型不匹配时,报InputMismatchException异常

数组

- 数组(Array)，是多个相同类型数据按一定顺序排列的集合，并使用一个名字命名，并通过编号的方式对这些数据进行统一管理。

一维数组

声明: `type[] var`

初始化:

- 静态初始化(初始化与赋值同时进行): `new int[]{1001, 1002, 1003, 1004};`
- 动态初始化(初始化与赋值分开进行): `new int[4]`
- 初始化后,长度就确定了
- 使用 `var1 = var2` 指向的是用一个地址
- 类型判断: `int arr[] = {1,2,3,4}`
- 默认初始化值:
  - 整型:0
  - 浮点型:0.0
  - 字符型:ASCII编号为0的字符
  - 布尔型:false
  - 引用数据类型:null

调用:使用角标 `var[index]`
获取长度:使用length属性 `var.length`

内存解析:

- 栈(局部变量)
- 堆(new)
- 方法区
  - 常量池(字符串等)
  - 静态域(static)

二维数组

- 对于二维数组的理解，我们可以看成是一维数组array1又作为另一个一维数组array2的元素而存在。

声明与初始化:

- 静态初始化 `int[][] arr = new int [][]{{1,2,3},{4,5},{6,7,8}}`
- 动态初始化 `int[][] arr = new int[3][3]` 或 `int[][] arr = new int[3][]`
- 外层元素初始化为地址或null

调用: `nums[2][3]`

十大排序算法

- 选择排序
  - 直接选择排序、堆排序
- 交换排序
  - 冒泡排序、快速排序
- 插入排序
  - 直接插入排序、折半插入排序、Shell排序
- 归并排序
- 桶式排序
- 基数排序

Arrays工具类

如:

1. boolean equals(int[] a,int[] b) 判断两个数组是否相等。
2. String toString(int[] a) 输出数组信息。
3. void fill(int[] a,int val) 将指定值填充到数组之中。
4. void sort(int[] a) 对数组进行排序。
5. int binarySearch(int[] a,int key) 对排序后的数组进行二分法检索指定的值。

数组常见异常

- 数组脚标越界异常(ArrayIndexOutOfBoundsException)
- 空指针异常(NullPointerException)

java note2 面向对象

面向过程(POP:Procedure Oriented Programming)与面向对象(OOP:Object Oriented Programming):
二者都是一种思想，面向对象是相对于面向过程而言的。面向过程，强调的是功能行为，以函数为最小单位，考虑怎么做。面向对象，将功能封装进对象，强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。

面向对象的三大特征:

- 封装 (Encapsulation)
- 继承 (Inheritance)
- 多态 (Polymorphism)

Java类以及类的成员

- 类:类是对一类事物的描述，是抽象的、概念上的定义
- 对象对象是实际存在的该类事物的每个个体，因而也称为实例(instance)

类的成员

设计一个类:

- 属性:成员变量
- 方法:成员方法

实例化

```
Class instance = new Class()
```

- 匿名对象: `new Class()`

调用

```
instance.attribute = ...
instance.method(args)
```

属性

```
权限修饰符 数据类型 属性名 = 默认值;
```

- 属性的默认初始化值与数组中的相同

方法

```
权限修饰符 返回值类型 方法名（形参列表）{
    方法体
}
```

- 可以调用当前类的属性和方法
- 方法中不可以再定义方法
- 形参列表中不可以有默认值(区别于C++)

方法重载:

- 在同一个类中，允许存在一个以上的同名方法，只要它们的参数个数或者参数类型不同即可。

可变个数的形参:

- 允许直接定义能和多个实参相匹配的形参。格式:方法名(数据类型…参数名),
  如: `public static void test(int a, String...strs)`
- 可变参数：方法参数部分指定类型的参数个数是可变多个：0个，1个或多个
- 可变个数形参的方法与同名的方法之间，彼此构成重载
- 可变参数方法的使用与方法参数部分使用数组是一致的,如上例相等于 `public static void test(int a, String[] strs)`
- 方法的参数部分有可变形参，需要放在形参声明的最后
- 在一个方法的形参位置，最多只能声明一个可变个数形参

方法参数的值传递机制(值传递):

- 对于基本数据类型:将数据值传递给形参
- 对于引用数据类型:将地址值传递给形参

递归:一个方法体内调用它自身。

封装性

封装性:隐藏对象内部的复杂性，只对外公开简单的接口。便于外界调用.

封装性体现:

- 将数据声明为私有的(private)，再提供公共的（public）方法:getXxx()和setXxx()实现对该属性的操作
- 将类方法中常用的方法抽象为工具方法,声明为私有的

权限修饰符:

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	√	×	×	×
缺省	√	×	×	×
protected	√	√	√	×
public	√	√	√	√

- 4种权限修饰符都可以修饰类的内部结构：属性,方法,构造器,内部类
- 修饰类只能用public和private

构造器(构造方法)

创建类:new + 构造器,相当于C++的构造函数

作用:

- 创建对象
- 给对象进行初始化

构造器的特征:

- 它具有与类相同的名称
- 它不声明返回值类型。（与声明为void不同）
- 不能被static、final、synchronized、abstract、native修饰，不能有return语句返回值

种类:

- 隐式无参构造器（系统默认提供）
- 显式定义一个或多个构造器（无参、有参）

注意:

- Java语言中，每个类都至少有一个构造器
- 默认构造器的修饰符与所属类的修饰符一致
- 一旦显式定义了构造器，则系统不再提供默认构造器
- 父类的构造器不可被子类继承

JavaBean

JavaBean是指符合以下标准的Java类:

- 类是公共的
- 有一个无参的公共的构造器
- 有属性，且有对应的get、set方法

this关键字

this指的是当前的对象或正在构造的对象

- 调用属性,方法:
  - 调用当前对象的属性

    - 调用当前对象的方法
    - 我们可以用this来区分属性和局部变量,如 `this.name = name`
- 调用构造器:
    - 如在另一个构造器中调用 `this()` 可以调用本类中的无参构造器
    - 此时 `this(形参列表)` 应放在首行
- 直接调用当前对象

package关键字

package语句作为Java源文件的第一条语句，指明该文件中定义的类所在的包。

- 格式为: `package 顶层包名.子包名`
- MVC设计模式:MVC是常用的设计模式之一，将整个程序分为三个层次：视图模型层，控制器层，与数据模型层。
- JDK中主要的包介绍
    - java.lang 包含一些Java语言的核心类，如String、Math、Integer、 System和Thread，提供常用功能
    - java.net 与网络相关的的类和接口
    - java.io 提供多种输入/输出功能的类
    - java.util 包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。
    - java.text 包含了一些java格式化相关的类
    - java.sql 包含了java进行JDBC数据库编程的相关类/接口

import关键字

为使用定义在不同包中的Java类，需用import语句来引入指定包层次下所需要的类或全部类(*)

- 格式： `import 包名. 类名;`
- 如果导入的类或接口是java.lang包下的，或者是当前包下的，则可以省略此import语句。
- 如果在代码中使用不同包下的同名的类。那么就需要使用类的全类名的方式指明调用的是哪个类。
- 如果已经导入java.a包下的类。那么如果需要使用a包的子包下的类的话，仍然需要导入。
- import static组合的使用：调用指定类或接口下的静态的属性或方法

继承性和多态性

继承性

继承:多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要继承那个类即可。子类继承了父类，就继承了父类的方法和属性。

语法规则:class Subclass extends SuperClass{ }

作用：

- 继承的出现减少了代码冗余，提高了代码的复用性。
- 继承的出现，更有利于功能的扩展。
- 继承的出现让类与类之间产生了关系，提供了多态的前提.

注意:

- Java只支持单继承和多层继承，不允许多重继承
- 如果我们没有显式地声明一个类地父类的话,则该类继承java.lang.Object类

方法的重写

在子类中可以根据需要对从父类中继承来的方法进行改造，也称为方法的重置、覆盖。在程序执行时，子类的方法将覆盖父类的方法。

要求：

- 子类重写的方法必须和父类被重写的方法具有相同的方法名称、参数列表
- 子类重写的方法的返回值类型不能大于父类被重写的方法的返回值类型
- 子类重写的方法使用的访问权限不能小于父类被重写的方法的访问权限,子类不能重写父类中声明为private权限的方法
- 子类方法抛出的异常不能大于父类被重写方法的异常
- 子类与父类中同名同参数的方法必须同时声明为非static的(即为重写)，或者同时声明为static的（不是重写）。因为static方法是属于类的，子类无法覆盖父类的方法。

super关键字

super表示调用直接父类父类或间接父类

- 访问父类中定义的属性
- 调用父类中定义的成员方法
  - 当子父类出现同名成员时(重写时)，可以用super表明调用的是父类中的成员
- 在子类构造器中调用父类的构造器 `super(形参列表)`
  - 子类中所有的构造器默认都会访问父类中空参数的构造器super()
  - 当父类中没有空参数的构造器时，子类的构造器必须通过this(参数列表)或者super(参数列表)语句指定调用本类或者父类中相应的构造器，且必须放在构造器的首行(因为在构造子类之前,先要构造父类)

子类对象实例化过程

当我们通过子类的构造器创建子类对象,我们一定或间接地调用了父类的构造器,直到调用到Object的空参构造器为止

注意:

- 虽然子类构造器调用了父类的构造器,但是只创建了一个对象

多态性

父类的引用指向子类的对象: `ParentClass p = new ChildClass()`

编译时类型与运行时类型:编译时的类型由声明该变量的类型决定,运行时实际由赋予该变量的对象决定,编译看左边,运行看右边.编译时类型和运行时类型不一致，就出现了对象的多态性(Polymorphism)实现通用性

虚拟方法调用:子类中定义了与父类同名同参数的方法(重写)，在多态情况下，此时父类的方法称为虚拟方法，因为父类根据赋给它的不同子类对象，动态调用属于子类的该方法。如上例ParentClass和ChildClass都定义了method(),则 `p.method()` 执行的是ChildClass类的method().这样的方法调用在编译期是无法确定的(动态绑定)。

注意:

- 多态性仅对于方法而言,成员变量不具备多态性
- 内存中实际加载了子类的属性和方法,但由于变量被声明为父类类型,此时父类不能调用子类的方法和属性

向下转型:使用强制类型转换符 `ChildClass c = (ChildClass)p` 可以让c调用子类的成员变量

instanceof关键字

为了避免在向下转型时,出现ClassCastException异常,使用 `x instanceof A` 可以判断x是否为A的对象

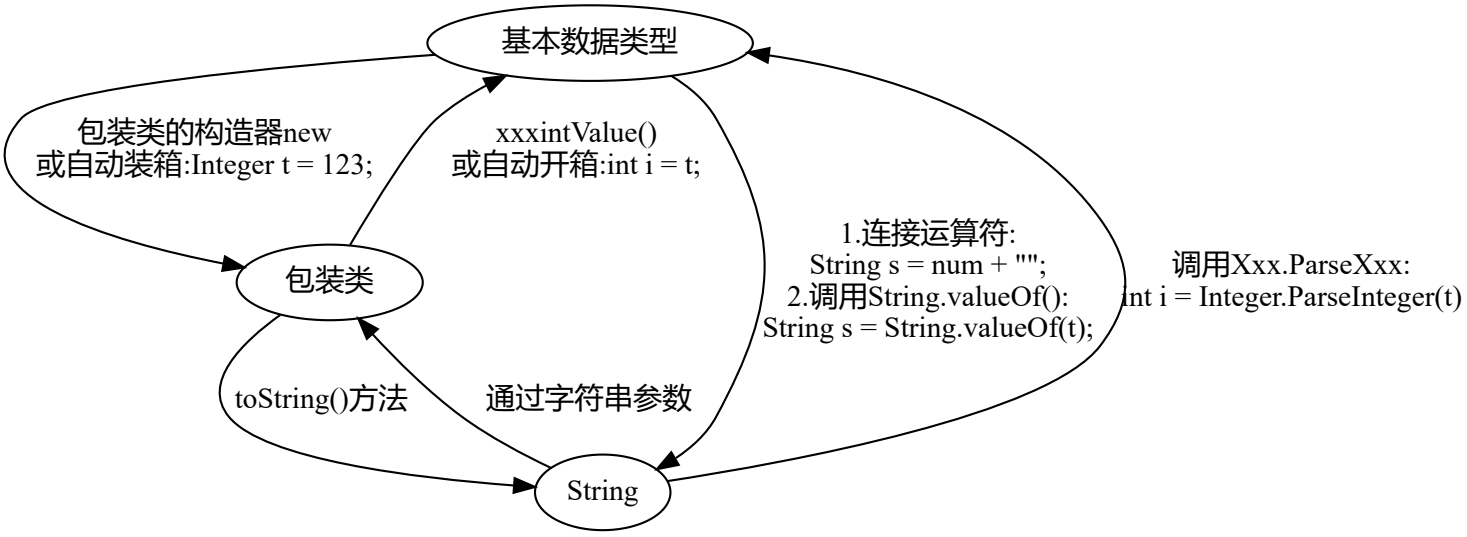
Object类的使用

- Object只声明了一个空参构造器
- Object里没有属性
- 方法:
  - equals(Object obj)在没重写之前和==的作用相同,而String,Date,File,包装类重写了该方法,判断两个对象的属性是否相同 (==是运算符,用于基本数据类型时判断数值是否相等,用于引用数据类型时判断地址值是否相等)
  - toString()当输出一个对象引用时,默认调用对象的同toString方法,重写的该方法可以返回对象的内容(类型+[内容])

包装类(Wrapper)

针对八种基本数据类型定义相应的引用类型—包装类（封装类）:

基本数据类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character



- JDK5.0新特性:自动装箱与自动开箱

面向对象的其他

static关键字

用于修饰属性:静态变量,多个相同类的对象共享同一个静态变量.

- 静态变量随着类的加载而加载,可以通过 类.静态变量 调用
- 静态变量的加载早于对象的创建,只加载一次,存在于方法区的静态域中
- 类中的常量(final)通常也修饰为静态的

用于修饰方法:静态方法

- 静态方法随着类的加载而加载,可以通过 类.静态方法 调用
- 静态方法中,只能调用静态的属性和方法,不能调用this,super关键字,因为并没有创建类
- 工具类中的方法通常声明为静态的

单例设计模式

采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

饿汉式:

```
class Singleton {
// 1.私有化构造器
  private Singleton() {

  }
// 2.内部提供一个当前类的实例
// 4.此实例也必须静态化
  private static Singleton single = new Singleton();
// 3.提供公共的静态的方法，返回当前类的对象
  public static Singleton getInstance() {
    return single;
  }
}
```

懒汉式:

```
class Singleton {
  private Singleton() {

  }

  private static Singleton single = null;

  public static Singleton getInstance() {
    if(single == null) {
      single = new Singleton();
    }
  return single;
  }
}
```

- 饿汉式对象的加载时间过长
- 懒汉式线程不安全

main方法

- public:Java虚拟机需要调用该方法作为程序的入口
- static:使用该方法不必造一个对象
- void:没有返回值
- String[] args:接受String数组参数

代码块

对Java类或对象进行初始化

- 静态代码块:
  - 静态代码块随着类的加载而加载，且只执行一次。
  - 若有多个静态的代码块，那么按照从上到下的顺序依次执行。
  - 静态代码块的执行要先于非静态代码块。
- 非静态代码块:
  - 每次创建对象的时候，都会执行一次。且先于构造器执行。
  - 若有多个非静态的代码块，那么按照从上到下的顺序依次执行。

总结：程序中成员变量赋值的执行顺序

1. 声明变量的默认初始化
2. 由上到下依次执行显示初始化或代码块
3. 构造器中的初始化
4. 对象.属性或对象.方法赋值

final关键字

- 修饰类:不能被继承
- 修饰方法:不能被重写
- 修饰变量:变成常量
  - 修饰属性:可以显式,在代码块,在构造器中赋值
  - 修饰局部变量:不能修改值

abstract关键字

- 修饰类:抽象类,该类不能实例化
  - 该类中要有构造器便于子类实例化
- 修饰方法:抽象方法,只有方法的声明,没有方法体
  - 包含抽象方法的类一定是抽象类
  - 子类需要重写父类所有的抽象类,或子类也是一个抽象类

匿名类:

假如Person是一个抽象类且eat()是Preson中的所有抽象方法,以下可以定义一个匿名类.

```
Person p = new Person(){
    @Override
    public void eat() {
      System.out.println("吃东西");
};
```

模板方法设计模式

接口

接口使用interface定义,定义了相同的行为特征;类通过实现(implements)接口的方式使用

接口中能定义:

- 全局变量(public static final),但是在代码中可以省略这些关键字

- 抽象方法(public abstract)

注意:

- 接口中没有构造器
- 接口与接口之间可以多继承
- 接口中也体现了多态性
- 也可以定义匿名实现类
- Java8在接口中还可以定义静态方法和默认(default)方法,接口中的静态方法只能接口使用,默认方法可以通过实现类的对象调用
- 如果实现类重写了接口中的默认方法,则调用的是重写后的方法;若实现类的父类声明了和接口中同名同参数的方法,那么在子类没有重写的情况下,默认调用父类的方法(类优先)
- 调用接口中被重写的方法: `接口名.super.方法`

```
// 接口的多态性
interface USB{
  void start();
  void stop();
}

class Flash implements USB{
  @Override
  public void start() {
    System.out.println("U盘开启工作");
  }

  @Override
  public void stop() {
    System.out.println("U盘结束工作");
  }
}

class Computer{
  public void transferData(USB usb){//USB usb = new Flash();
    usb.start();
    //具体传输数据的细节
    usb.stop();
  }
}
```

代理设计模式,工厂设计模式

内部类

在Java中，允许一个类的定义位于另一个类的内部，前者称为内部类，后者称为外部类。

分类:

- 成员内部类(static和非static两种)
- 局部内部类

注意:

- 内部类可以直接使用外部类的所有成员，包括私有的数据
- 调用外部类 `外部类.this.成员`

成员内部类:

- 与外部类不同,成员内部类还可以声明为private或protected的

- 非static的成员内部类中的成员不能声明为static的，只有在外部类或static的成员内部类中才可声明static成员。

局部内部类:

- 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类
- 但是它的对象可以通过外部方法的返回值返回使用，返回值类型只能是局部内部类的父类或父接口类型
- 当局部内部类使用外部方法的局部变量时,此局部变量应是final的

匿名内部类:

```
new 父类构造器(实参列表) implements 实现接口 {
    //匿名内部类的类体部分
}
```

java note3 异常 枚举 注解 泛型

异常

异常及其分类

程序执行中发生的不正常情况称为“异常”

分类:

- Error:Java虚拟机无法解决的严重问题。如：JVM系统内部错误、资源耗尽等严重情况。比如：StackOverflowError和OOM(OutOfMemoryError)。一般不编写针对性的代码进行处理。
- Exception: 其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。如空指针访问,试图读取不存在的文件,网络连接中断,数组角标越界等
  - 编译时异常(checked)
  - 运行时异常(unchecked):RunTimeException

常见的异常

编译时异常：

- IOException
  - FileNotFoundException
- ClassNotFoundException

运行时异常：

- NullPointerException
- IndexOutOfBoundsException
  - ArrayIndexOutOfBoundsException
  - StringIndexOutOfBoundsException
- ClassCastException
- NumberFormatException
- InputMismatchException
- ArithmeticException
- …

异常处理机制1 try-catch-finally

```
try{
    //可能出现异常的代码
} catch (异常类型1 e) {
    //处理异常的方式1
} catch (异常类型2 e) {
    //处理异常的方式2
}
...
finally {
    //一定会执行的代码
}
```

注意:

- catch中的异常类型如果满足子父类关系，则要求子类一定声明在父类的上面。
- 常用的异常对象处理的方式:1.String getMessage() 2.e.printStackTrace()
- 使用try-catch-finally处理编译时异常,使得程序在编译时就不再报错,但是运行时仍可能报错,相当于我们将一个编译时可能出现的异常，延迟到运行时出现。
- 由于运行时异常比较常见，所以通常不针对运行时异常处理,针对于编译时异常才一定要考虑异常处理。

关于finally:

- finally是可选的
- finally中声明的是一定会被执行的代码,即使catch中又出现异常,try中有return语句,catch中有return语句等情况
- 像数据库连接、输入输出流、网络编程Socket等资源，JVM是不能自动的回收的，我们需要自己手动的进行资源的释放。此时的资源释放，就需要声明在finally中。

异常处理机制2 throws+异常类

`throws 异常类` 写在方法的声明处,一旦当方法体执行时,出现异常,就将异常抛给了方法的调用者.异常代码后续的代码，将不再执行!

- 如果父类中被重写的方法没有throws方式处理异常，则子类重写的方法也不能使用throws,必须使用try-catch-finally方式处理.如果父类throws一个异常类,那么子类throws的异常类不能比父类抛出的大
- 执行的方法a中，先后又调用了另外的几个方法，且这几个方法是递进关系执行的。我们建议这几个方法使用throws的方式进行处理,而执行的方法a可以考虑使用try-catch-finally方式进行处理。

手动抛出异常

- 自动生成异常
- 手动抛出异常 `throw 异常类`

用户自定义异常类

```
public class MyException extends Exception{  //继承Exception或RunTimeException
    static final long serialVersionUID = -7034897193246939L;  //id

    public MyException(){  //空参构造器
    }

    public MyException(String msg){
    super(msg);
    }
}
```

枚举类

当需要定义一组常量时，建议使用枚举类

自定义枚举类

1. 对象如果有实例变量，应该声明为private final，并在构造器中初始化
2. 私有化类的构造器
3. 在类的内部创建枚举类的实例，声明为：public static final

```
class Season{
    private final String SEASONNAME;//季节的名称
    private final String SEASONDESC;//季节的描述

    private Season(String seasonName,String seasonDesc){
        this.SEASONNAME = seasonName;
        this.SEASONDESC = seasonDesc;
    }

    public static final Season SPRING = new Season("春天", "春暖花开");
    public static final Season SUMMER = new Season("夏天", "夏日炎炎");
    public static final Season AUTUMN = new Season("秋天", "秋高气爽");
    public static final Season WINTER = new Season("冬天", "白雪皑皑");

    //按需求提供get()方法,toString()方法等
}

Season summer = Season.SUMMER;
```

enum定义枚举类

1. 提供当前枚举类的对象(必须声明在第一行)，多个对象之间用",“隔开，末尾对象”;"结束
2. 声明Season对象的属性:private final修饰
3. 私有化类的构造器,并给对象属性赋值

```
enum Season1 {
    SPRING("春天","春暖花开"),
    SUMMER("夏天","夏日炎炎"),
    AUTUMN("秋天","秋高气爽"),
    WINTER("冬天","冰天雪地");

    private final String seasonName;
    private final String seasonDesc;

    private Season1(String seasonName,String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    //按需求提供get()方法,toString()方法等
}
```

- 使用 enum 定义的枚举类默认继承了 java.lang.Enum类，因此不能再继承其他类
- 列出的实例系统会自动添加 public static final 修饰,不能再添加修饰
- JDK 1.5 中可以在 switch 中使用Enum定义的枚举类的对象作为表达式, case 子句可以直接使用枚举值的名字, 无需添加枚举类作为限定。

- 枚举类可以实现一个或多个接口,若每个枚举值在调用实现的接口方法呈现相同的行为方式，则只要统一实现该方法即可。若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式,则可以让每个枚举值分别来实现该方法

方法:

- String toString():返回枚举类对象的名称
- values():返回所有的枚举类对象构成的数组
- valueOf(String objName):返回枚举类中对象名是objName的对象,如果没有objName的枚举类对象，则抛IllegalArgumentException异常

注解(JDK5.0)

注解(Annotation)其实就是代码里的特殊标记, 这些标记可以在编译, 类加载或运行时被读取, 并执行相应的处理。

生成文档相关的注解

- @author 标明开发该类模块的作者，多个作者之间使用,分割
- @version 标明该类模块的版本
- @see 参考转向，也就是相关主题
- @since 从哪个版本开始增加的
- @param 对方法中某参数的说明，如果没有参数就不能写
- @return 对方法返回值的说明，如果方法的返回值类型是void就不能写
- @exception 对方法可能抛出的异常进行说明 ，如果方法没有用throws显式抛出的异常就不能写
- 其中@param @return 和 @exception 这三个标记都是只用于方法的。
  - @param的格式要求：@param 形参名 形参类型 形参说明
  - @return 的格式要求：@return 返回值类型 返回值说明
  - @exception的格式要求：@exception 异常类型 异常说明
  - @param和@exception可以并列多个

在编译时进行格式检查的注解(JDK内置的三个基本注解)

- @Override: 限定重写父类方法,
- @Deprecated: 用于表示所修饰的元素(类, 方法等)已过时,通常是因为所修饰的结构危险或存在更好的选择
- @SuppressWarnings: 抑制编译器警告,如 `@SuppressWarnings("unused")`

实现替代配置文件功能的注解

通常在框架中使用

自定义注解

参照@SuppressWarnings定义

- 注解声明为：@interface
- 内部定义成员，通常使用value表示
- 可以指定成员的默认值，使用default定义

```
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE}) //{}内为参数
public @interface MyAnnotation {
    String value() default "hello";  //成员变量以无参数方法的形式来声明
}
```

- 如果自定义注解没有成员，表明是一个标识作用。
- 如果注解有成员，在使用注解时，需要指明成员的值。
- 自定义注解必须配上注解的信息处理流程(使用**反射**,且注解的生命周期要声明为RUNTIME)才有意义。
- 自定义注解通过都会指明两个元注解：Retention、Target

元注解

对注解进行解释说明的注解

- Retention：指定所修饰的 Annotation 的生命周期：SOURCE\CLASS（默认行为）\RUNTIME.只有声明为RUNTIME生命周期的注解，才能通过反射获取。
- Target:用于指定被修饰的 Annotation 能用于修饰哪些程序元素:TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE
- Documented:表示所修饰的注解在被javadoc解析时，保留下来。
- Inherited:被它修饰的 Annotation 将具有继承性。

JDK8新特性

可重复注解：

```
@Repeatable(MyAnnotations.class)//在MyAnnotation上声明@Repeatable，成员值为MyAnnotations.class
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
public @interface MyAnnotation {
    String value() default "hello";
}

// MyAnnotations元注解与MyAnnotations相同
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
public @interface MyAnnotations {
    MyAnnotation[] value();//注解数组
}

//使用
//jdk 8之前的写法：
//@MyAnnotations({@MyAnnotation(value="hi"),@MyAnnotation(value="hi")})
@MyAnnotation(value="hi")
@MyAnnotation(value="abc")
```

类型注解：

- ElementType.TYPE_PARAMETER 表示该注解能写在类型变量的声明语句中（如：泛型声明）。
- ElementType.TYPE_USE 表示该注解能写在使用类型的任何语句中。

泛型(JDK5)

泛型就是允许在定义类、接口时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型,这个类型参数将在使用时确定.目的是使类型统一一致.

注意:

- 实例化后，操作原来泛型位置的结构必须与指定的泛型类型一致。
- 泛型的类型必须是类。需要用到基本数据类型的位置，拿包装类替换
- 如果实例化时，没有指明泛型的类型。默认类型为Object类型

指定泛型

例子如: `Map<String, Integer> map = new HashMap<String,Integer>();`

- 泛型数组 `E[] elements = (E[])new Object[capacity];`
- jdk7新特性：类型推断 `Map<String, Integer> map = new HashMap<>();`

自定义泛型

泛型类

- 泛型类的构造函数不能加上泛型
- 泛型不同的引用不能相互赋值
- 异常类不能是泛型的
- 父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：

```
class A<T1, T2> {}

class B<T2, T3> extends A <Integer, T2> {} //T1指定了泛型,T2保留了泛型,T3是自定义的泛型
```

泛型方法

```
public class DAO {
    public <E> E get(int id, E e) {
    ...
    }
}
```

- 自定义了泛型的方法才叫泛型方法,而不是使用了类的泛型的方法
- 在静态方法中不能使用类的泛型,因为泛型在创建对象的时候才能确定类型
- 但是泛型方法可以使用自定义的泛型

泛型的继承性

- 如果类A是类B的父类，G<A>和G<B>二者不具备子父类关系，二者是并列关系。
  - 二者共同的父类是G<?>(?是通配符)
  - 但A是B的父类

通配符泛型的方法:

- 写入：对于List<?>不能向其内部添加数据(除了null),因为不确定类型
- 读取：允许读取数据，读取的数据类型为Object。

注意：

- 不能用在泛型方法声明上，返回值类型前面<>不能使用?，如public static void test(ArrayList list){}
- 不能用在泛型类的声明上，如class GenericTypeClass<?>{}
- 不能用于创建对象，如new ArrayList<?>()

有限制条件的通配符的使用：

- G<? extends A> 只允许泛型为A或A的子类调用
- G<? super A> 只允许泛型为A或A的父类调用

java note 4 常见类

String

特点

- 被final修饰,不可被继承
- 实现了Serializable接口,表示字符串是支持序列化的
- 实现了Comparable接口：表示String可以比较大小
- 内部定义了final char[] value用于存储字符串数据
- String具有不可变性:当对字符串进行修改时，需要重新在常量池中指定新的内存区域

初始化

- `String s = "XXX"` 即字面量定义
  - 此时字符串值声明在常量池中
  - 字符串常量池中不会存储相同内容的字符串
- `String s = new String()`
  - 参数可以是字符串"XXX",或另一个String类,或char[]数组
  - 此时地址值是new出的String类的地址值

注意:

- 只要操作中有一个是变量(而不是常量)，结果就在堆中
- 如果调用intern()方法，返回值在常量池中

常用方法

- int length()：返回字符串的长度
- char charAt(int index)：返回某索引处的字符
- boolean isEmpty()：判断是否是空字符串
- String toLowerCase()：将所有字符转换为小写
- String toUpperCase()：将所有字符转换为大写
- String trim()：返回字符串的副本，忽略前导空白和尾部空白
- boolean equals(Object obj)：比较字符串的内容是否相同
- boolean equalsIgnoreCase(String anotherString)：与equals方法类似，忽略大小写
- String concat(String str)：将指定字符串连接到此字符串的结尾，等价于用“+”
- int compareTo(String anotherString)：比较两个字符串的大小
- String substring(int beginIndex)：返回一个新的字符串，它是此字符串的从beginIndex开始截取到最后的一个子字符串。
- String substring(int beginIndex, int endIndex) ：返回一个新字符串，它是此字符串从beginIndex开始截取到endIndex(不包含)的一个子字符串。
- String replace(char oldChar, char newChar)：用 newChar 替换此字符串中出现的所有 oldChar
- String replace(CharSequence target, CharSequence replacement)：使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。
- String replaceAll(String regex, String replacement)：使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
- String replaceFirst(String regex, String replacement)：使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。
- boolean matches(String regex)：告知此字符串是否匹配给定的正则表达式。

- String[] split(String regex)：根据给定正则表达式的匹配拆分此字符串。
- String[] split(String regex, int limit)：根据匹配给定的正则表达式来拆分此字符串，最多不超过limit个，如果超过了，剩下的全部都放到最后一个元素中。
- boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束
- boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始
- boolean startsWith(String prefix, int toffset)：测试此字符串从指定索引开始的子字符串是否以指定前缀开始
- boolean contains(CharSequence s)：当且仅当此字符串包含指定的 char 值序列时，返回 true
- int indexOf(String str)：返回指定子字符串在此字符串中第一次出现处的索引
- int indexOf(String str, int fromIndex)：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
- int lastIndexOf(String str)：返回指定子字符串在此字符串中最右边出现处的索引
- int lastIndexOf(String str, int fromIndex)：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索

转换

与基本数据类型，包装类的转换

- String->基本数据类型、包装类：调用包装类的静态方法：parseXxx(str)
- 基本数据类型、包装类->String:调用String重载的valueOf(xxx)

与char[]的转换

- String->char[]:调用String的toCharArray()
- char[]->String:调用String的构造器

与byte[]的转换

- 编码：String->byte[]:调用String的getBytes()
- 解码：byte[]->String:调用String的构造器
- 参数可加字符编码类型如"gbk"等

Stringbuffer和StringBuilder

- String:不可变的字符序列,效率低
- StringBuffer:可变的字符序列；线程安全的
- StringBuilder:可变的字符序列；jdk5.0新增，线程不安全的，效率比StringBuffer高

比较:

- 底层都是使用char[]存储,但StringBuffer和StringBuilder底层创建了一个默认长度是16的数组
- StringBuffer和StringBuilder如果要添加的数据超出数组长度，则数组扩容为原来容量的2倍 + 2，同时将原有数组中的元素复制到新的数组中。

方法:

- String的方法
- StringBuilder(int capacity)
- StringBuffer append(xxx)：字符串拼接
- StringBuffer delete(int start,i nt end)：删除指定位置的内容
- StringBuffer replace(int start, int end, String str)：把[start,end)位置替换为str
- StringBuffer insert(int offset, CharSequence s)：在指定位置插入xxx
- StringBuffer reverse()：把当前字符序列逆转
- void setCharAt(int n ,char ch): 修改索引为n位置的字符

日期时间

(java.util.)Date

- Date()：创建一个对应当前时间的Date对象
- Date(long date): 创建指定毫秒数的Date对象
- toString():显示当前的年、月、日、时、分、秒、时区
- long getTime():获取当前Date对象对应的毫秒数。（时间戳）

(java.sql.)Date

数据库中的日期类,继承于java.util.Date

- java.sql.Date(long date)
- 调用Date的getTime()方法做参数可以将Date转换为java.sql.Date

时间

- long System.currentTimeMillis()返回当前时间距离1970-01-01的毫秒数

(java.text.)SimpleDateFormat

实例化

- SimpleDateFormat(String pattern):填入格式,如"YYYY-MM-dd hh:mm:ss"

格式化

- String format(Date date):将日期转化为特定格式的字符串

解析

- Date parse(String date):将格式化字符串转化为日期
- 如果date的格式不正确,会抛出ParseException异常

(java.util.)Calendar

实例化

- GregorianCalendar Calendar.getInstance();
- Calendar是一个抽象类,不能直接使用构造器,其静态方法getInstance()返回的是其子类GregorianCalendar的对象

方法

- int get(Calendar.DAY_OF_MONTH):取得日历对象的日期是那个月的第几天
- int set(Calendar.DAY_OF_MONTH, 22):修改成当月的第几天
- void add(Calendar.DAY_OF_MONTH, -3):增删天数
- Date getTime():日历类->日期类
- void setTime(Date):日期类->日历类
- 获取月份时,0是一月,1是一月,以此类推;获取星期时,1是星期日,2是星期一,以此类推

(java.time.)LocalDate LocalTime LocalDateTime(JDK8新增)

- LocalDateTime LocalDateTime.now():返回当前时间

- LocalDateTime LocalDateTime.of(year, month, day, hour, minute, second):设定指定的时间
- int getDayOfMonth():getXxx()返回指定的属性
- LocalDateTime withDayOfMonth(int):withXxx(int)修改指定属性
- LocalDateTime plusMonths(int):plusXxx(int),minusXxx(int)修改指定属性
- 特性:不可变性

Instant(JDK8新增)

- Instant Instant.now():获取本初子午线对应的标准时间
- OffsetDateTime atOffset(ZoneOffset.ofHours(8)):添加时间的偏移量
- long .toEpochMilli():获取自1970年1月1日0时0分0秒（UTC）开始的毫秒数
- Instant Instant.ofEpochMilli(long):通过给定的毫秒数，获取Instant实例

DateTimeFormatter(JDK8新增)

实例化

- DateTimeFormatter DateTimeFormatter.ISO_LOCAL_DATE_TIME:使用预定义的标准格式
- DateTimeFormatter DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG):使用本地化相关格式
- DateTimeFormatter DateTimeFormatter.ofPattern(“yyyy-MM-dd hh:mm:ss”):使用自定义格式

格式化

- String format(LocalDateTime):日期->字符串

解析:

- TemporalAccessor formatter.parse(String):字符串->日期

比较器

实现了Comparable或Comparator接口即可以使用比较运算符或sort方法比较大小

Comparable

自然排序

- 重写compareTo(obj)的规则：如果当前对象this大于形参对象obj，则返回正整数;小于，返回负整数;等于，返回零。

```
//按商品的价格从小到大排序,价格相等则按名称从小到大排序
@Override
public int compareTo(Object o) {
    if(o instanceof Goods){
        Goods goods = (Goods)o;
        if (!this.price.equals(goods.price)) {
            return this.price.compareTo(good.price);
        }else{
           return this.name.compareTo(goods.name);
        }
    }
    throw new RuntimeException("传入的数据类型不一致！");
}
```

Comparator

定制排序:当元素的类型没有实现Comparable接口而又不方便修改代码，或者实现的Comparable接口的排序规则不适合当前的操作，那么可以考虑使用Comparator的对象来排序.规则与Comparable相同

```
Arrays.sort(arr, new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        ...
    }
});
```

其他

System

- void exit(int status):退出程序。其中status的值为0代表正常退出，非零代表异常退出。使用该方法可以在图形界面编程中实现程序的退出功能
- void gc()：请求系统进行垃圾回收。至于系统是否立刻回收，则取决于系统中垃圾回收算法的实现以及系统执行时的情况。
- String getProperty(String key)：获得系统中的属性。系统中常见的属性如：
  - java.version
  - java.home
  - os.name
  - os.version
  - user.home
  - user.name
  - user.dir

Math

- abs 绝对值
- acos,asin,atan,cos,sin,tan 三角函数
- sqrt 平方根
- pow(double a,double b) a的b次幂
- log 自然对数
- exp e为底指数
- max(double a,double b) 最大值
- min(double a,double b) 最小站
- random() 返回0.0到1.0的随机数
- long round(double a) double型数据a转换为long型（四舍五入）
- toDegrees(double angrad) 弧度—>角度
- toRadians(double angdeg) 角度—>弧度

(java.math.)BigInteger BigDecimal

BigInteger可以表示不可变的任意精度的整数

- BigInteger(String val)：根据字符串构建BigInteger对象
- BigInteger abs()：绝对值
- BigInteger add(BigInteger val)：加
- BigInteger subtract(BigInteger val)：减
- BigInteger multiply(BigInteger val)：乘
- BigInteger divide(BigInteger val)：除,整数相除只保留整数部分。

- BigInteger remainder(BigInteger val)：取模
- BigInteger[] divideAndRemainder(BigInteger val)：返回相除的结果和余数
- BigInteger pow(int exponent)：幂运算

BigDecimal类支持不可变的、任意精度的有符号十进制定点数

- BigDecimal(double val)
- BigDecimal(String val)
- BigDecimal add(BigDecimal augend)
- BigDecimal subtract(BigDecimal subtrahend)
- BigDecimal multiply(BigDecimal multiplicand)
- BigDecimal divide(BigDecimal divisor, int scale, int roundingMode): scale指返回的精度, roundingMode指截断的模式,如BigDecimal.ROUND_HALF_UP等

java note 5 集合

面向对象语言对事物的体现都是以对象的形式，为了方便对多个对象的操作，就要对对象进行存储。另一方面，使用Array存储对象方面具有一些弊端，而Java 集合就像一种容器，可以动态地把多个对象的引用放入容器中。Java 集合类可以用于存储数量不等的多个对象，还可用于保存具有映射关系的关联数组。

框架:

- Collection接口：单列数据，定义了存取一组对象的方法的集合
  - List：元素有序、可重复的集合(动态数组)
    - ArrayList
    - LinkedList
    - Vector
  - Set：元素无序、不可重复的集合
    - HashSet
      - LinkedHashSet
    - TreeSet
- Map接口：双列数据，保存具有映射关系"key-value对"的集合
  - HashMap
    - LinkedHashMap
  - TreeMap
  - Hashtable、
    - Properties

Collection接口

Collection方法

- 增
  - add(Object obj)
  - addAll(Collection coll)
- 查
  - int size()
  - boolean isEmpty()
  - boolean contains(Object obj)：通过equals方法判断是否相同

    - boolean containsAll(Collection c)：也是调用元素的equals方法来比较的,判断集合中是否包含c中的全部数据
    - boolean equals(Object obj):判断两个集合是否相等
    - hashCode()获取集合对象的哈希值
- 删
    - void clear()
    - boolean remove(Object obj) ： 通过元素的equals方法判断是否是要删除的那个元素。只会删除找到的第一个元素
    - boolean removeAll(Collection coll)：取当前集合的差集
- 改
    - boolean retainAll(Collection c)：把交集的结果存在当前集合中
- 转换
    - Object[] toArray()
- 遍历
    - Iterator iterator()：返回迭代器对象，用于集合遍历

将数组转换为集合: `<Serializable> List<Serializable> Arrays.asList(Serializable... a)`

Iterator接口

集合元素的遍历操作，使用迭代器Iterator接口

方法:

- boolean hasNext():判断接下来是否有元素
- Object next():指针移动到下一元素并返回
- void remove():删除集合中迭代器所指向的元素

```
while(iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

注意:

- 集合对象每次调用iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前
- 不能删除同一个位置的元素两次

foreach循环(JDK5新增)

用于遍历集合、数组,底层仍是用迭代器实现

```
//内部仍然调用的是迭代器。
for(Object obj : collection) {
    System.out.println(obj);
}
```

List子接口

- ArrayList：List接口的主要实现类；线程不安全的，效率高；底层使用Object[] elementData存储
- LinkedList：对于频繁的插入、删除操作，使用此类效率比ArrayList高；底层使用双向链表存储
- Vector：作为List接口的古老实现类；线程安全的，效率低；底层使用Object[] elementData存储

ArrayList

- 底层创建了长度是10的Object[]数组elementData

- 如果添加导致底层elementData数组容量不够，则扩容。默认情况下，扩容为原来的容量的1.5倍，同时需要将原有数组中的数据复制到新的数组中。
- 建议开发中使用带参的构造器
- JDK8中第一次调用add()时，底层才创建了长度为10的数组，并将数据添加到elementData中

LinkedList

- 底层使用双向链表实现
- 内部声明了Node类型的first和last属性，默认值为null

Vector

略

List新增方法

- void add(int index, Object ele):在index位置插入ele元素
- boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来
- Object get(int index):获取指定index位置的元素
- int indexOf(Object obj):返回obj在集合中首次出现的位置
- int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
- Object remove(int index):移除指定index位置的元素，并返回此元素
- Object set(int index, Object ele):设置指定index位置的元素为ele,并返回被替换的元素
- List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的子集合

Set子接口

- HashSet：作为Set接口的主要实现类；线程不安全的；可以存储null值
- LinkedHashSet：作为HashSet的子类；遍历其内部数据时，可以按照添加的顺序遍历。对于频繁的遍历操作，LinkedHashSet效率高于HashSet.
- TreeSet：可以按照添加对象的指定属性，进行排序。

Set特性

- 无序性：不等于随机性。存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。
- 不可重复性：保证添加的元素按照equals()判断时，不能返回true.即：相同的元素只能添加一个。

以HashSet为例说明:向HashSet中添加元素a,首先调用元素a所在类的hashCode()方法，计算元素a的哈希值，然后此哈希值接着通过与运算计算出在HashSet底层数组中的存放位置（即为：索引位置），判断数组此位置上是否已经有元素：

- 如果此位置上没有其他元素，则元素a添加成功。
- 如果此位置上有其他元素b(或以链表形式存在的多个元素），则比较元素a与元素b的hash值：
  - 如果hash值不相同，则元素a添加成功。
  - 如果hash值相同，进而需要调用元素a所在类的equals()方法：
    - equals()返回true,元素a添加失败
    - equals()返回false,则元素a添加成功。

向Set中添加的数据，其所在的类一定要重写hashCode()和equals(),且要求相等的对象必须具有相等的散列码.对象中用equals() 方法比较的 Field，都应该用来计算 hashCode 值。

HashSet

底层使用HashMap实现,数组+链表的结构

- 元素a 与已经存在指定索引位置上数据以链表的方式存储。
  - jdk 7 :元素a放到数组中，指向原来的元素。
  - jdk 8 :原来的元素指向元素a

LinkedHashSet

底层使用LinkedHashMap实现,在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据。

TreeSet

底层使用TreeMap实现,红黑树结构.向TreeSet中添加的数据，要求是相同类的对象。因为TreeSet通过一个类的compareTo()或compare()方法排序.且比较两个对象是否相同的标准为比较器返回0,而不再使用equals().

Set无新增方法

Map接口

Map中的key是无序的、不可重复的，value是可重复的

- HashMap:作为Map的主要实现类；线程不安全的，效率高；可以存储null的key和value
  - LinkedHashMap:保证在遍历map元素时，可以按照添加的顺序实现遍历。对于频繁的遍历操作，此类执行效率高于HashMap。
- TreeMap:会对添加的key-value对进行排序，实现排序遍历。
- Hashtable:作为古老的实现类；线程安全的，效率低；不能存储null的key和value
  - Properties:常用来处理配置文件。key和value都是String类型

Map方法

- 增/改
  - Object put(Object key,Object value)：将指定key-value添加(或修改)到当前map对象中
  - void putAll(Map m):将m中的所有key-value对存放到当前map中
- 删
  - Object remove(Object key)：移除指定key的key-value对，并返回value
  - void clear()：清空当前map中的所有数据
- 查
  - Object get(Object key)：获取指定key对应的value
  - boolean containsKey(Object key)：是否包含指定的key
  - boolean containsValue(Object value)：是否包含指定的value
  - int size()：返回map中key-value对的个数
  - boolean isEmpty()：判断当前map是否为空
  - boolean equals(Object obj)：判断当前map和参数对象obj是否相等
- 遍历
  - Set keySet()：返回所有key构成的Set集合
  - Collection values()：返回所有value构成的Collection集合
  - Set entrySet()：返回所有key-value对构成的Set集合,entrySet集合中的元素都是Map.Entry
    - entry.getKey():返回entry的key
    - entry.getValue():返回entry的value

HashMap

jdk7:

在实例化以后，底层创建了长度是16的一维数组Entry[] table(或其他2的幂次值)。map.put(key, value):与set.add(value)方法类似,不同的地方是:计算哈希值的时候使用的是key;对key相同的情况进行的是value替换.
在添加元素当超出临界值(且要存放的位置非空)时，默认的扩容方式是扩容为原来容量的2倍，并将原有的数据经过重新计算哈希值复制过来。

jdk8:

- 实例化后没有立即创建一个长度为16的数组,在加入第一个元素时创建
- 底层的数组是：Node[], 而非Entry[]
- 增加元素时将原来的元素指向新增的元素;而不是让新增的元素放入数组中,新的元素指向原来的元素
- 当数组的某一个索引位置上的元素以链表形式存在的数据个数>8且当前数组的长度>64时，此时此索引位置上的所数据改为使用红黑树存储。但当当前数组的长度<=64时,先扩容.

一些常量:

- DEFAULT_INITIAL_CAPACITY : HashMap的默认容量:16
- DEFAULT_LOAD_FACTOR：HashMap的默认加载因子：0.75
- threshold：扩容的临界值，容量 * 填充因子：16 * 0.75 = 12
  - 当达到扩容临界值但新增元素所在位置还没有元素时,还不必扩容
- TREEIFY_THRESHOLD：Bucket中链表长度大于该默认值，转化为红黑树:8
- MIN_TREEIFY_CAPACITY：桶中的Node被树化时最小的hash表容量:64

LinkedHashMap

在原有的HashMap底层结构基础上，添加了一对指针，指向前一个和后一个元素。

TreeMap

和TreeSet类似,向TreeMap中添加的数据，要求key是相同类的对象;TreeMap通过key的compareTo()或compare()方法排序.

Properties

常用来处理配置文件

```
Properties pros = new Properties();
fis = new FileInputStream("xxx.properties");
pros.load(fis);
String name = pros.getProperty("name");
String password = pros.getProperty("password");
System.out.println("name = " + name + ", password = " + password);
//关闭: fis.close();
```

配置文件中的内容:(中文需要用ASCII码存储)

```
name=jack
password=1234
```

Colletions

操作Collection,Map的工具类

Collections方法

- reverse(List)：反转 List 中元素的顺序
- shuffle(List)：对 List 集合元素进行随机排序
- sort(List)：根据元素的自然顺序对指定 List 集合元素按升序排序
- sort(List, Comparator)：根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
- swap(List, int, int)：将指定 list 集合中的 i 处元素和 j 处元素进行交换
- Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素
- Object max(Collection, Comparator)：根据 Comparator 指定的顺序，返回给定集合中的最大元素
- Object min(Collection)
- Object min(Collection, Comparator)
- int frequency(Collection, Object)：返回指定集合中指定元素的出现次数
- void copy(List dest, List src)：将src中的内容复制到dest中
  - copy前要保证dest的长度与src的长度相同 `List dest = Arrays.asList(new Object[list.size()]);`
- boolean replaceAll(List list, Object oldVal, Object newVal)：使用新值替换 List 对象的所有旧值
- Collection synchronizedCollection(Collection c): 将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题

java note 6 多线程 IO流

多线程

程序、进程、线程

- 程序:一段静态的代码
- 进程:正在运行的一个程序
- 线程:一个程序内部的一条执行路径,JVM中每个线程有自己的栈和程序计数器,多个线程共享堆和方法区

并行与并发:

- 并行：多个CPU同时执行多个任务
- 并发：一个CPU(采用时间片)同时执行多个任务

守护线程:

- 守护线程是用来服务用户线程的，通过在start()方法前调用
- thread.setDaemon(true)可以把一个用户线程变成一个守护线程。
- Java垃圾回收就是一个典型的守护线程。
- 若JVM中都是守护线程，当前JVM将退出。

线程的创建和使用

方式1:继承Thread类

- 创建继承于Thread的类
- 重写Thread类的run(),将此线程执行的操作声明在run()中
- 创建Thread类的子类的对象
- 通过此对象调用start()

```
class MyThread extends Thread {
    @Override
    public void run() {
        ...
    }
}

MyThread t1 = new MyThread();
MyThread t2 = new MyThread();
t1.start();
t2.start();
```

注意:

- 当调用start()时:1.启动当前线程2.调用当前线程的run().不能直接调用run(),因为这样不能创建多线程
- 一个对象只能启动一个线程,否则抛出IllegalThreadStateException异常;启动多线程需要多个线程对象

方式2:实现Runnable接口

- 创建一个实现了Runnable接口的类
- 实现run()方法
- 创建实现类的对象
- 将此对象作为参数传递到Thread类的构造器中,创建Thread类的对象
- 通过Thread类的对象调用start()

```
class MThread implements Runnable{
    @Override
    public void run() {
        ...
    }
}

MThread mThread = new MThread();
Thread t1 = new Thread(mThread);
Thread t2 = new Thread(mThread);
t1.start();
t2.start();
```

注意:

- 开发中,优先选择实现Runnable接口的方式,原因:1. 没有类单继承性的局限性 2. 实现的方式更适合来处理多个线程有共享数据的情况。
- Thread也实现了Runnable接口

方式3:实现Callable接口(JDK5.0新增)

- 创建一个实现Callable的实现类
- 实现call方法
- 创建Callable接口实现类的对象
- 将此Callable接口实现类的对象作为参数传递到FutureTask构造器中，创建FutureTask的对象
- 将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()
- FutureTask的对象get()方法的返回值即为call()的返回值。

```
class MyThread implements Callable{
    @Override
    public Object call() throws Exception {
        ...
        return res;
    }
}

MyThread myThread = new myThread();
FutureTask futureTask = new FutureTask(myThread);
new Thread(futureTask).start();
try {
    Object res = futureTask.get();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

- 使用该方法可以有返回值,可以throws异常,且支持泛型

方法四:创建线程池(JDK5.0新增)

提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用,也便于线程管理。

```
//1. 提供指定线程数量的线程池
ExecutorService service = Executors.newFixedThreadPool(10);

//实际上返回的是ThreadPoolExecutor类
ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;

//可以设置线程池的属性,如:
//service1.setCorePoolSize(15);//核心池的大小
//service1.setKeepAliveTime(10);//线程没有任务时最多保持多长时间后会终止

//2.执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象
service.execute(new myThread());//用于Runnable实现类
//service.submit(Callable callable);//用于Callable实现类

//3.关闭连接池
service.shutdown();
```

Thread中的其他API

- currentThread():静态方法，返回执行当前代码的线程
- getName():获取当前线程的名字(或在创建对象时传入String参数)
- setName():设置当前线程的名字
- yield():释放当前cpu的执行权
- join():在线程a中调用线程b的join(),此时线程a就进入阻塞状态，直到线程b完全执行完以后，线程a才结束阻塞状态。
- sleep(long millitime):让当前线程“睡眠”指定的millitime毫秒。在指定的millitime毫秒时间内，当前线程是阻塞状态
- isAlive():判断当前线程是否激活/存活

线程的优先级

Thread中关于优先级的常量:

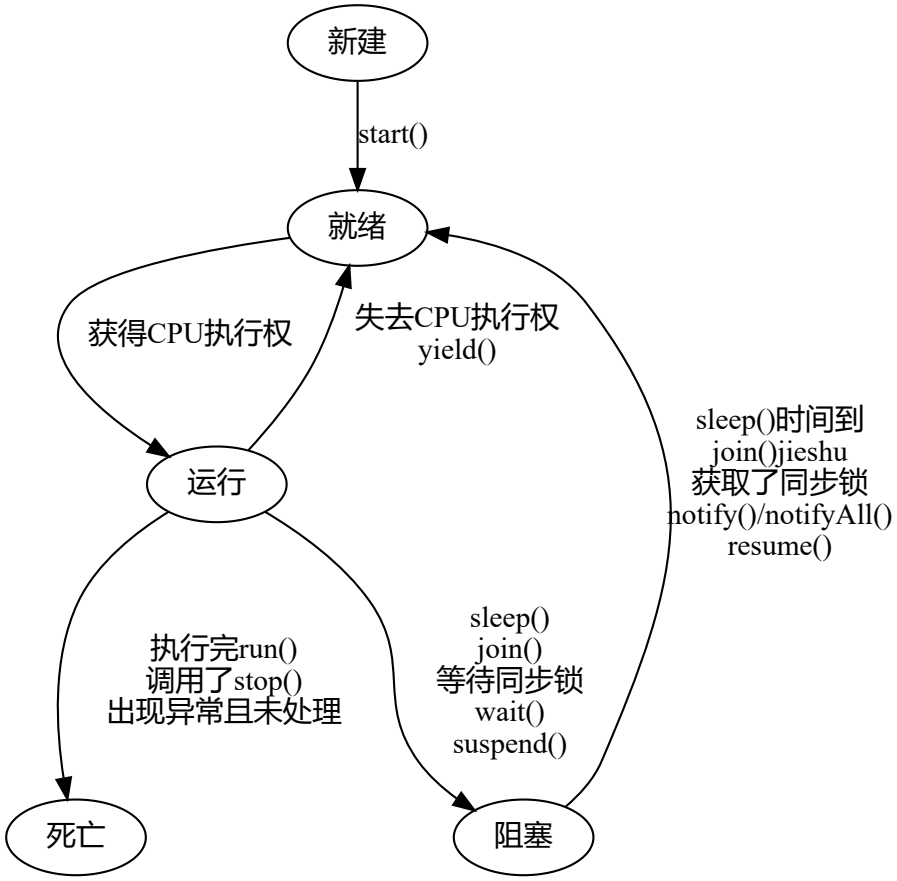
- MAX_PRIORITY：10

- MIN_PRIORITY：1
- NORM_PRIORITY：5(默认优先级)

方法:

- getPriority():获取线程的优先级
- setPriority(int p):设置线程的优先级
- 注意:高优先级的线程要抢占低优先级线程cpu的执行权,高优先级的线程较大概率的情况被执行,但并不意味着只有当高优先级的线程执行完以后,低优先级的线程才执行.

线程的生命周期



线程同步

线程安全问题:一个线程操作共享数据的过程中,另一个线程也参与了进来,导致数据计算错误.解决方法:当一个线程在操作共享数据的时候,其他线程不能参与进来,直到这个线程操作完成

同步机制解决线程安全问题:

方式一:同步代码块

```
synchronized (锁) {
   //需要被同步的代码
}
```

- 任何一个类的对象,都可以充当锁
- 多个线程必须要共用同一把锁
- 可以考虑this关键字或 `当前类名.class` 作为锁

方法二:同步方法

在方法前加上synchronized关键字

- 同步的普通方法相当于加上当前对象(this)作为锁
- 同步的静态方法相当于加上当前类作为锁

方法三:ReentrantLock类锁(JDK5新增)

- ReentrantLock实例化
- 使用lock()方法加锁
- 使用unlock()方法解锁

```
ReentrantLock lock = new ReentrantLock();
lock.lock()
//需要被同步的代码
lock.unlock()
```

同步的方式，解决了线程的安全问题。但是操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低.

线程安全的懒汉式单例模式

```
class Singleton {
    private Singleton() {

    }

    private static Singleton single = null;

    public static Singleton getInstance() {
        if (single == null) {  //如果对象已经被创建,直接return,防止线程等待
            synchronized (Singleton.class()) {  //加锁保证线程安全
                if(single == null) {
                    single = new Singleton();
                }
            }
        }
        return single;
    }
}
```

死锁

不同的线程分别占用对方需要的锁不放弃，都在等待对方放弃自己需要的锁，就形成了线程的死锁

锁的释放方式:

- 同步代码块或方法执行结束
- 遇到return,break结束同步代码
- 遇到未处理的异常
- wait()方法

不会释放锁:

- yield()方法
- sleep()方法

线程通信

- wait():使当前线程进入阻塞状态，并释放锁。
- notify():唤醒wait中的一个线程。如果有多个线程wait，唤醒优先级高的那个。
- notifyAll():唤醒所有wait中的线程。

```
synchronized (lock) {
    lock.notify();
    ...
    try {
        lock.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- wait()，notify()，notifyAll()三个方法必须使用在同步代码块或同步方法中。
- 三个方法的调用者必须是同步代码块或同步方法中的同步监视器,否则会出现IllegalMonitorStateException异常
- 三个方法定义在java.lang.Object类中.

生产者消费者问题

IO流

File类

File类声明在java.io包下,File类的一个对象，代表一个文件或一个文件目录(文件夹).File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法,但并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，需要使用IO流来完成。

方法:

- 实例化:
  - File(String filePath)
  - File(String parentPath,String childPath)
  - File(File parentFile,String childPath)
    - 以上的路径可以是相对路径或绝对路径;分隔符在windows下用\\表示,在unix下为/,分隔符可用File.separator代替
- 查询:
  - String getAbsolutePath()：获取绝对路径
  - String getPath()： 获取路径
  - String getName()： 获取名称
  - String getParent()：获取上层文件目录路径。查询不到则返回null
  - long length()： 获取文件长度（即：字节数）,不能获取目录的长度。
  - long lastModified()： 获取最后一次修改的时间戳
  - String[] list()： 获取指定目录下的所有文件或者文件目录的名称数组,用于文件夹
  - File[] listFiles()： 获取指定目录下的所有文件或者文件目录的File数组,用于文件夹
  - boolean isDirectory()：返回是否是文件目录
  - boolean isFile()： 返回是否是文件
  - boolean exists()： 返回是否存在
  - boolean canRead()： 返回是否可读
  - boolean canWrite()： 返回是否可写
  - boolean isHidden()： 返回是否隐藏

- 移动
  - boolean renameTo(File dest):把文件重命名(移动)为指定的文件路径,返回是否成功.如 `file1.renameTo(file2)` 要想保证成功,需要file1在硬盘中是存在的，且file2不能在硬盘中存在。
- 创建
  - boolean createNewFile()： 创建文件。若文件存在，则不创建，返回false
  - boolean mkdir()： 创建文件目录。若目录存在则不创建了。如果此文件目录的上层目录不存在，也不创建
  - boolean mkdirs()： 创建文件目录。若目录存在则不创建了。如果上层文件目录不存在，一并创建
- 删除
  - boolean delete()：删除文件或者文件夹,注意被删除的文件目录下不能有子目录或文件,且Java中删除的东西不走回收站

流

Java程序中，对于数据的输入/输出操作以“流(stream)” 的方式进行。

流的分类:

- 操作数据单位
  - 字节流
  - 字符流:用来处理文本文件,不能使用字符流来处理图片等字节数据
- 数据的流向
  - 输入流
  - 输出流
- 流的角色
  - 节点流:直接从数据源或目的地读写数据
  - 处理流:“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。

流的体系结构:

抽象基类	节点流（或文件流）	缓冲流（处理流的一种）
InputStream	FileInputStream (read(byte[] buffer))	BufferedInputStream (read(byte[] buffer))
OutputStream	FileOutputStream (write(byte[] buffer,0,len)	BufferedOutputStream (write(byte[] buffer,0,len) / flush()
Reader	FileReader (read(char[] cbuf))	BufferedReader (read(char[] cbuf) / readLine())
Writer	FileWriter (write(char[] cbuf,0,len)	BufferedWriter (write(char[] cbuf,0,len) / flush()

输入流:FileReader

从文件中读出数据到内存中

```
FileReader fr = null;
try {
    //1.File类的实例化
    File file = new File("hello.txt");

    //2.FileReader流的实例化
    fr = new FileReader(file);

    int data;
    //3.读/写的操作
    while((data = fr.read()) != -1){
        System.out.print((char)data);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //4.资源的关闭
    if(fr != null){
        try {
            fr.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

jdk7新特性:try-with-resources自动关闭资源

```
File file = new File("./solution/tmp.txt");
try(FileReader reader = new FileReader(file);) {
    int ch;
    while ((ch = reader.read()) != -1) {
        System.out.print((char) ch);
    }
} catch (IOException e) {
    e.getStackTrace();
}
```

- 读入的文件一定要存在，否则就会报FileNotFoundException

方法:

- int read():返回读入的一个字符。如果达到文件末尾，返回-1
- int read(char[] cbuf):将读入的字符写入cbuf中,返回每次读入cbuf数组中的字符的个数。如果达到文件末尾，返回-1
- int read(char[] cbuf, int off, int len):指定放入数组的起始位置和长度
- void close() throws IOException:关闭此输入流并释放与该流关联的所有系统资源。

输出流:FileWriter

从内存中写出数据到硬盘的文件里。

- File对应的硬盘中的文件如果不存在，在输出的过程中，会自动创建此文件。
- File对应的硬盘中的文件如果存在：默认情况会对原有文件进行覆盖,指定append参数为true后,不会对原有文件覆盖，而是在原有文件基础上添加内容

方法:

- void write(int c);
- void write(char[] cbuf);

- void write(char[] buff, int off, int len);
- void close();

字节流:FileInputStream/FileOutputStream

与FileReader/FileWriter类似,不同在于输入/输出的是字节流,用byte[]存储

缓存流:BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter

缓冲流作用：内部提供了一个缓冲区,加快流的读取、写入的速度

使用:构造器参数中传入字节流即可,如 `BufferedInputStream(FileInputStream)` ;方法调用基本一样

注意:

- 关闭外层流的同时，内层流也会自动的进行关闭。关于内层流的关闭，我们可以省略.
- 缓存流的输出流有flush()方法刷新缓存区
- BufferedReader提供String readLine()方法读取文本文件的一行(不包含换行符)

转换流:InputStreamReader/OutputStreamWriter

- InputStreamReader(InputStream in)：将InputStream转换为Reader,输入字节流转换为输入字符流
- OutputStreamWriter(OutputStream out)：将Writer转换为OutputStream,输出字符流转换为输出字节流
- 可以加上第二个字符串参数指明字符集类型

标准输入输出流:System.in/System.out

- InputStream System.in:从键盘中读入字节流
- PrintStream System.out:输出到控制台

打印流:PrintStream/PrintWriter

实现将基本数据类型的数据格式转化为字符串输出

- PrintStream(OutputStream, true):创建打印字节流,设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出缓冲区)
  - 提供多种print(),println()方法
- System.setOut(PrintStream)把标准输出流(控制台输出)改成文件

数据流:DataInputStream/DataOutputStream

用于读取或写出基本数据类型的变量或字符串

- DataOutputStream(OutputStream)
  - 提供了多种输出基本数据类型的方法,如writeInt(),writeUTF()等
- DataInputStream(InputStream)
  - 提供了多种读取基本数据类型的方法,将write改为read即可,注意读取的类型要与原来输出的类型一致

对象流:ObjectInputStream/ObjectOutputStream

可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来。

- 序列化过程：对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流,从而将java对象保存到磁盘中或通过网络传输出去
- 反序列化：将磁盘文件中的对象还原为内存中的一个java对象

```
oos = new ObjectOutputStream(new FileOutputStream("object.dat"));
oos.writeObject(new String("我爱北京天安门"));
oos.flush();//刷新
ois = new ObjectInputStream(new FileInputStream("object.dat"));
Object obj = ois.readObject();
String str = (String) obj;
```

对象流要求对象是是可序列化的，可序列化的类需要满足相应的要求。

```
public class Person implements Serializable{  //需要实现Serializable接口
    public static final long serialVersionUID = 475463534532L;  //提供序列版本id
    ...//如果有其他属性,需要保证其内部所有属性也必须是可序列化的
    //补充：ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量
}
```

随机存取文件流:RandomAccessFile

- RandomAccessFile直接继承于java.lang.Object类，实现了DataInput和DataOutput接口
- RandomAccessFile既可以作为一个输入流，又可以作为一个输出流
- 如果RandomAccessFile作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建。
- 如果写出到的文件存在，则会对原有文件内容进行覆盖。（从头覆盖）

方法

- RandomAccessFile(String File, String mode);
  - mode:“r”: 以只读方式打开;“rw”：打开以便读取和写入
- long getFilePointer()：获取文件记录指针的当前位置
- void seek(long pos)：将文件记录指针定位到pos位置
- read/write方法与其他流类似

NIO.2

Non-Blocking IO 非阻塞式IO

jdk 7.0 时，引入了 Path、Paths、Files三个类。此三个类声明在：java.nio.file包下。Path可以看做是java.io.File类的升级版本。也可以表示文件或文件目录，与平台无关.

方法略

java note 7 网络编程

网络编程

网络编程中有两个主要的问题：

1. 如何准确地定位网络上一台或多台主机；定位主机上的特定的应用
2. 找到主机后如何可靠高效地进行数据传输

网络编程中的两个要素：

1. IP和端口号
2. 提供网络通信协议：TCP/IP参考模型（应用层、传输层、网络层、物理+数据链路层）

IP地址

Internet上的计算机（通信实体）的唯一标识

分类：

- IPv4(4个字节组成,如192.168.0.1)和IPv6
- 万维网和局域网

域名:如www.baidu.com

本地回路地址：127.0.0.1 对应着：localhost
在Java中用java.net.InetAddress类代表IP

- 实例化：
  - InetAddress InetAddress.getByName(String host)
  - InetAddress InetAddress.getLocalHost()
- 方法:
  - String getHostName()
  - String getHostAddress()

端口号

表示正在计算机上运行的网络进程

- 不同的进程有不同的端口号
- 范围：被规定为一个 16 位的整数 0~65535
- 端口号与IP地址的组合得出一个网络套接字：Socket
  - Socket(InetAddress address, int port)
  - InetAddress getInetAddress()
  - int getPort()

协议

服务端(TCP)

```
//1.创建服务器端的ServerSocket，指明自己的端口号
ServerSocket ss = new ServerSocket(8899);
//2.调用accept()表示接收来自于客户端的socket
Socket socket = ss.accept();
//3.获取输入流
InputStream is = socket.getInputStream();
//4.读取输入流中的数据
ByteArrayOutputStream baos = new ByteArrayOutputStream();
byte[] buffer = new byte[5];
int len;
while((len = is.read(buffer)) != -1){
    baos.write(buffer,0,len);
}
System.out.println(baos.toString());
//5.关闭资源
baos.close();
is.close();
socket.close();
ss.close();
```

客户端(TCP)

```
//1.创建Socket对象，指明服务器端的ip和端口号
InetAddress inet = InetAddress.getByName("127.0.0.1");
Socket socket = new Socket(inet,8899);
//2.获取一个输出流，用于输出数据
OutputStream os = socket.getOutputStream();
//3.写出数据的操作
os.write("你好，我是客户端".getBytes());
//4.资源的关闭
os.close();
socket.close();
```

接收端(UDP)

```
DatagramSocket socket = new DatagramSocket(9090);
byte[] buffer = new byte[100];
DatagramPacket packet = new DatagramPacket(buffer, 0, buffer.length);
socket.receive(packet);
System.out.println(new String(packet.getData(), 0, packet.getLength()));
socket.close();
```

发送端(UDP)

```
DatagramSocket socket = new DatagramSocket();
byte[] data = "UDP方式发送".getBytes();
InetAddress inet = InetAddress.getByName("127.0.0.1");
DatagramPacket packet = new DatagramPacket(data, 0, data.length, inet, 9090);
socket.send(packet);
socket.close();
```

URL

统一资源定位符,格式:协议 主机名 端口号 资源地址 (参数列表),如 `http://localhost:8080/examples/beauty.jpg?username=Tom`

```
URL url = new URL("http://localhost:8080/examples/beauty.jpg");
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
urlConnection.connect();
InputStream is = urlConnection.getInputStream();
//... 传输操作
urlConnection.disconnect();
//... 关闭资源
```

反射

反射(reflection)是被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。因为加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。

Class类

程序经过javac.exe命令以后，会生成一个或多个字节码文件(.class结尾)。接着使用java.exe命令对某个字节码文件进行解释运行,将某个字节码文件加载到内存中,此过程就称为类的加载。加载到内存中的类，就称为运行时类.此运行时类就是Class类的一个对象。

获取Class的实例的方式:

```
//方式一：调用运行时类的属性：.class
Class clazz1 = Person.class;

//方式二：通过运行时类的对象,调用getClass()
Person p1 = new Person();
Class clazz2 = p1.getClass();

//方式三：调用Class的静态方法：forName(String classPath) (常用)
Class clazz3 = Class.forName("package.Person");
//如clazz3 = Class.forName("java.lang.String");

//方式四：使用类的加载器：ClassLoader
ClassLoader classLoader = SomeClass.class.getClassLoader();
Class clazz4 = classLoader.loadClass("package.Person");
```

类的加载器:

```
//对于自定义类，使用系统类加载器进行加载
ClassLoader classLoader1 = SomeClass.class.getClassLoader();
//调用系统类加载器的getParent()：获取扩展类加载器
//负责将jar包装入工作库
ClassLoader classLoader2 = classLoader1.getParent();
//调用扩展类加载器的getParent()：无法获取引导类加载器
//引导类加载器主要负责加载java的核心类库，无法加载自定义类。
ClassLoader classLoader3 = classLoader2.getParent();//null
```

类,接口,数组,枚举类,注解,基本数据类型,void,Class都可以是Class的实例,万事万物皆对象:

- Class c1 = Object.class;
- Class c2 = Comparable.class;
- Class c3 = String[].class;
- Class c4 = int[][].class;
- Class c5 = ElementType.class;
- Class c6 = Override.class;
- Class c7 = int.class;

- Class c8 = void.class;
- Class c9 = Class.class;

运行时类的完整结构

- 属性
  - Field[] getFields():获取当前运行时类及其父类中声明为public访问权限的属性
  - Field[] getDeclaredFields():获取当前运行时类中声明的所有属性。（不包含父类中声明的属性）
    - int getModifiers();属性的修饰符
    - Class getType();属性类型
    - String getName();属性名
- 方法
- 构造器
- 父类(带泛型的)
- 实现的接口
- 所在的包
- 注解
- …

调用运行时类的指定结构

```
Class clazz = Person.class;

//创建对象,默认使用的是空参构造器
Person p = clazz.getDeclaredConstructor().newInstance();

//调用构造器(可以是私有的)创建对象
Constructor constructor = clazz.getConstructor(String.class, int.class);  //参数列表为参数的类型
constructor.setAccessible(true);  //获得调用私有结构的权限
Person p = (Person) constructor.newInstance("Tom", 12);

//调用属性
Field age = clazz.getDeclaredField("age");  //属性名
age.setAccessible(true);
age.set(p, 18);  //对象 属性值
int getAge = age.get(p);  //对象

//调用方法
Method show = clazz.getDeclaredMethod("getDescription", String.class);  //方法名 形参列表
show.setAccessible(true);
String description = getDescription.invoke(p, "China");  //对象(如果是静态方法,这里是对应的类) 参数列表
```

当需要使用动态性时,通过反射的方法调用构造器,属性和方法

动态代理 AOP

略

JDK8新特性

Lambda表达式

Lambda表达式的本质：函数式接口的实例

原来的写法:

```
new Comparator<Integer> comparator = new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
};
```

Lambda表达式写法: `(o1,o2) -> Integer.compare(o1,o2);`

- -> :lambda操作符 或 箭头操作符
- 左边：lambda形参列表 （接口中的抽象方法的形参列表）
- 右边：lambda体 （重写的抽象方法的方法体）

规则:

- 左边:
  - lambda形参列表的参数类型在可以类型推断的情况下可以省略(类型推断)
  - 如果lambda形参列表只有一个参数，()也可以省略
- 右边:
  - lambda体应该使用一对{}包裹,但如果lambda体只有一条执行语句（可能是return语句），省略这一对{}和return关键字

函数式接口

只声明了一个抽象方法的接口,可以在一个接口上使用 @FunctionalInterface 注解

java内置的4大核心函数式接口:

接口	抽象方法
消费型接口 Consumer	void accept(T t)
供给型接口 Supplier	T get()
函数型接口 Function<T,R>	R apply(T t)
断定型接口 Predicate	boolean test(T t)

方法引用 构造器引用

方法引用，本质上就是Lambda表达式

使用格式：类(或对象)::方法名

- 对象::非静态方法
- 类::静态方法
- 类::非静态方法

要求:

- 接口中的抽象方法的形参列表和返回值类型与方法引用的方法的形参列表和返回值类型相同（对应类型1和类型2）
- 当函数式接口方法的第一个参数是需要引用方法的调用者，并且第二个参数是需要引用方法的参数(或无参数)时(对应类型3)

```
//对象::实例方法
//Consumer中的void accept(T t)
//PrintStream中的void println(T t)

//Lambda表达式写法
Consumer<String> consumer = str -> System.out.println(str);

//方法引用写法
Consumer<String> consumer = System.out::println;

//调用
consumer.accept("beijing");

//类::静态方法
//Comparator中的int compare(T t1,T t2)
//Integer中的int compare(T t1,T t2)
Comparator<Integer> comparator = Integer::compare;

//类 :: 实例方法
//Comparator中的int comapre(T t1,T t2)
//String中的int t1.compareTo(t2)
Comparator<String> comparator = String::compareTo;
```

构造器引用

```
//返回对应对象
Supplier<Employee> supplier = Employee::new;
Employee employee = supplier.get();

//返回对应长度的数组
Function<Integer, int[]> function = int[]::new;
int[] arr = function.apply(10);
```

Stream API

使用Stream API 能对集合数据进行操作，类似于使用 SQL 执行的数据库查询。

- Stream关注的是对数据的运算，与CPU相关;集合关注的是数据的存储，与内存相关
- Stream 自己不存储元素,且Stream不会改变源对象,它们返回的是一个持有结果的新Stream
- Stream 执行流程:Stream的实例化,一系列的中间操作(过滤、映射、…),终止操作
- Stream 操作是延迟执行的,执行终止操作才会就执行中间操作链并产生结果,且之后不会再被使用

实例化

```
//通过集合
List<Employee> employees = ...;
//default Stream<E> stream() : 返回一个顺序流
Stream<Employee> stream = employees.stream();
//default Stream<E> parallelStream() : 返回一个并行流
Stream<Employee> parallelStream = employees.parallelStream();

//通过数组
//调用Arrays类的static <T> Stream<T> stream(T[] array): 返回一个流
IntStream stream = Arrays.stream(arr);
//自定义对象数组
Stream<Employee> stream1 = Arrays.stream(arr);

//通过Stream.of(T...values)
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);

//创建无限流
//迭代
//public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
//遍历前10个偶数
Stream.iterate(0, t -> t + 2).limit(10).forEach(System.out::println);
//生成
//public static<T> Stream<T> generate(Supplier<T> s)
//生成10个随机数
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

中间操作

- 筛选与切片
  - filter(Predicate p):从流中筛选出某些元素
  - limit(n):截断流，使元素不超过给定数量
  - skip(n):跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。
  - distinct():去重，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
- 映射
  - map(Function f):将元素转换成其他形式或提取信息，该函数会被应用到每个元素上，并将其映射成一个新的元素。
  - flatMap(Function f):将流中的每个值都换成另一个流，然后把所有流连接成一个流。
- 排序
  - sorted():自然排序
  - sorted(Comparator com):定制排序

终止操作

- 匹配与查找
  - allMatch(Predicate p)——检查是否匹配所有元素。
  - anyMatch(Predicate p)——检查是否至少匹配一个元素。
  - noneMatch(Predicate p)——检查是否没有匹配的元素。
  - Optional findFirst()返回第一个元素
  - Optional findAny()返回当前流中的任意元素
  - long count()返回流中元素的总个数
  - Optional max(Comparator c):返回流中最大元素
  - Optional min(Comparator c):返回流中最小值
  - forEach(Consumer c):内部迭代
- 归约
  - T reduce(T identity, BinaryOperator):可以将流中元素反复结合起来，得到一个值。返回 T
  - Optional reduce(BinaryOperator)
- 收集

- Collection collect(Collector c):将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法