目录

[1 Java语言 2](#_Toc45808131)

[1.1语言基础 2](#_Toc45808132)

[1.1 基础语法 2](#_Toc45808133)

[1.2 面向对象 2](#_Toc45808134)

[1.3接口 3](#_Toc45808135)

[1.4容器 4](#_Toc45808136)

[1.5 异常 4](#_Toc45808137)

[1.6 泛型 5](#_Toc45808138)

[1.7 反射 6](#_Toc45808139)

[1.8 注解 6](#_Toc45808140)

[1.9 I/O 6](#_Toc45808141)

[1.10集合 6](#_Toc45808142)

[1.11枚举 12](#_Toc45808143)

# 1 Java语言

## 1.1语言基础

### 1.1 基础语法

### 1.2 面向对象

区别：

**面向过程** ：**面向过程性能比面向对象高。** 因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，**面向过程没有面向对象易维护、易复用、易扩展。**

**面向对象** ：**面向对象易维护、易复用、易扩展。** 因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，**面向对象性能比面向过程低**。

**封装**

封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。就好像我们看不到挂在墙上的空调的内部的零件信息（也就是属性），但是可以通过遥控器（方法）来控制空调。

**继承**

不同类型的对象，相互之间经常有一定数量的共同点。例如，小明同学、小红同学、小李同学，都共享学生的特性（班级、学号等）。同时，每一个对象还定义了额外的特性使得他们与众不同。例如小明的数学比较好，小红的性格惹人喜爱；小李的力气比较大。继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间 ，提高我们的开发效率。

关于继承如下 3 点请记住：

1子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。

2子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

3子类可以用自己的方式实现父类的方法。（以后介绍）。

**多态**

多态，顾名思义，表示一个对象具有多种的状态。具体表现为父类的引用指向子类的实例。

多态的特点:

* 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
* 对象类型不可变，引用类型可变；
* 方法具有多态性，属性不具有多态性；
* 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
* 多态不能调用“只在子类存在但在父类不存在”的方法；
* 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法

### 1.3接口

接口相当于特殊的抽象类，定义方式/组成部分与抽象类类似

没有构造方法，不能创建对象

只能定义公开的静态常量（public static final）和公开的抽象方法（public abstract）

**使用方法**：用实现类去实现接口（implements）,必须覆盖所有抽象方法，然后用实现类定义对象。

**与抽象类异同**：

相同：

可编译成字节码文件

不能创建对象

可以作为引用类型

具备Object类总所定义的方法

不同：

接口所有属性都是公开的静态常量public static final

接口所有方法都是抽象方法 public abstract

接口没有构造方法，动态代码块，静态代码块

**应用**：Java为单继承，当父类方法种类无法满足子类需求时，可实现接口扩充子类能力

规范：1任何类在实现接口时，必须实现接口中所有的抽象方法，否则此类为抽象类

2实现接口的抽象方法时，访问修饰符必须是public

**常量接口**：将多个常用的表示状态或固定值的变量，以静态常量的形式定义在接口中统一管理

**接口回调**：现有接口的使用者，后有接口的实现者。

### 1.4容器

### 1.5 异常

在 Java 中，所有的异常都有一个共同的祖先 java.lang 包中的 Throwable 类。Throwable： 有两个重要的子类：Exception（异常） 和 Error（错误） ，二者都是 Java 异常处理的重要子类，各自都包含大量子类。

**Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题**。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

**Exception（异常）:是程序本身可以处理的异常**。Exception 类有一个重要的子类 RuntimeException。RuntimeException 异常由 Java 虚拟机抛出。NullPointerException（要访问的变量没有引用任何对象时，抛出该异常）、ArithmeticException（算术运算异常，一个整数除以 0 时，抛出该异常）和 ArrayIndexOutOfBoundsException （下标越界异常）。

**注意：异常和错误的区别：异常能被程序本身处理，错误是无法处理。**

**try-catch-finally**

try 块： 用于捕获异常。其后可接零个或多个 catch 块，如果没有 catch 块，则必须跟一个 finally 块。

catch 块： 用于处理 try 捕获到的异常。

finally 块： 无论是否捕获或处理异常，finally 块里的语句都会被执行。当在 try 块或 catch 块中遇到 return 语句时，finally 语句块将在方法返回之前被执行**。**

### 1.6 泛型

泛型：ArrayList <String> strList = new ArrayList<>()；

非泛型：

ArrayList list = new ArrayList();

list.add(“abc”); list.add(100);

遍历时 Object o = list.get(i);

使用时强转为String类型 String str = （String）o;

会引发ClassCastException异常，编译时没有错，运行时报错。

从JDK 5.0开始，Java引入“参数化类型（parameterized type）”的概念，这种参数化类型称为“泛型（Generic）”。

泛型是将数据类型参数化，即在编写代码时将数据类型定义成参数，这些类型参数在使用之前再进行指明具体类型。

泛型提高了代码的重用性，使得程序更加灵活、安全和简洁。【使用泛型可以避免方法重载需要写多个同名的方法】

在JDK 5.0之前，为了实现参数类型的任意化，都是通过Object类型来处理。但这种处理方式所带来的缺点是需要进行强制类型转换，

此种强制类型转换不仅使代码臃肿，而且要求程序员必须对实际所使用的参数类型已知的情况下才能进行，否则容易引起ClassCastException异常。

从JDK 5.0开始，Java增加对泛型的支持。使用泛型之后就不会出现上述问题。泛型的好处是在程序编译期会对类型进行检查，

捕捉类型不匹配错误，以免引起ClassCastException异常；而且泛型不需要进行强制转换，数据类型都是自动转换的。

泛型经常使用在类、接口和方法的定义中，分别称为泛型类、泛型接口和泛型方法。泛型类是引用类型，在内存堆中。

**泛型的优点：1编译期间检查类型 2 无须数据类型转换 3提高代码复用性**

泛型类：

1在没有指定具体数据类型时，按Object类型操作

2类型参数只能是类类型，不能时基本数据类型

3不同类类型参数构造的泛型类实质上是同一种类

类型擦除：泛型信息只存在于编译期间，在运行时会将泛型信息擦除。

### 1.7 反射

### 1.8 注解

### 1.9 I/O

### 1.10集合

<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/basic/Arrays,CollectionsCommonMethods.md>

#### Collections：

Collections 工具类常用方法:

* 排序
* 查找,替换操作
* 同步控制(不推荐，需要线程安全的集合类型时请考虑使用 JUC 包下的并发集合)
* **void** **reverse**(**List** list)//反转
* **void** **shuffle**(**List** list)//随机排序
* **void** **sort**(**List** list)//按自然排序的升序排序
* **void** **sort**(**List** list, **Comparator** c)//定制排序，由Comparator控制排序逻辑
* **void** **swap**(**List** list, **int** i , **int** j)//交换两个索引位置的元素
* **void** **rotate**(**List** list, **int** distance)//旋转。当distance为正数时，将list后distance个元素整体移到前面。当distance为负数时，将 list的前distance个元素整体移到后面。

排序：

**ArrayList**<**Integer**> arrayList = **new** **ArrayList**<**Integer**>();

arrayList.add(-1);

arrayList.add(3);

arrayList.add(3);

arrayList.add(-5);

arrayList.add(7);

arrayList.add(4);

arrayList.add(-9);

arrayList.add(-7);

**System**.out.println("原始数组:");

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

// void reverse(List list)：反转

**Collections**.reverse(arrayList);

**System**.out.println("Collections.reverse(arrayList):");

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

**Collections**.rotate(arrayList, 4);

**System**.out.println("Collections.rotate(arrayList, 4):");

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

// void sort(List list),按自然排序的升序排序

**Collections**.sort(arrayList);

**System**.out.println("Collections.sort(arrayList):");

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

// void shuffle(List list),随机排序

**Collections**.shuffle(arrayList);

**System**.out.println("Collections.shuffle(arrayList):");

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

// void swap(List list, int i , int j),交换两个索引位置的元素

**Collections**.swap(arrayList, 2, 5);

**System**.out.println("Collections.swap(arrayList, 2, 5):");

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

// 定制排序的用法

**Collections**.sort(arrayList, **new** **Comparator**<**Integer**>() {

@Override

**public** **int** **compare**(**Integer** o1, **Integer** o2) {

**return** o2.compareTo(o1);

}

});

**System**.out.println("定制排序后：");

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

查找、替换操作

**int** **binarySearch**(**List** list, **Object** key)//对List进行二分查找，返回索引，注意List必须是有序的

**int** **max**(**Collection** coll)//根据元素的自然顺序，返回最大的元素。 类比int min(Collection coll)

**int** **max**(**Collection** coll, **Comparator** c)//根据定制排序，返回最大元素，排序规则由Comparatator类控制。类比int min(Collection coll, Comparator c)

**void** **fill**(**List** list, **Object** obj)//用指定的元素代替指定list中的所有元素。

**int** **frequency**(**Collection** c, **Object** o)//统计元素出现次数

**int** **indexOfSubList**(**List** list, **List** target)//统计target在list中第一次出现的索引，找不到则返回-1，类比int lastIndexOfSubList(List source, list target).

**boolean** **replaceAll**(**List** list, **Object** oldVal, **Object** newVal), 用新元素替换旧元素

**ArrayList**<**Integer**> arrayList = **new** **ArrayList**<**Integer**>();

arrayList.add(-1);

arrayList.add(3);

arrayList.add(3);

arrayList.add(-5);

arrayList.add(7);

arrayList.add(4);

arrayList.add(-9);

arrayList.add(-7);

**ArrayList**<**Integer**> arrayList2 = **new** **ArrayList**<**Integer**>();

arrayList2.add(-3);

arrayList2.add(-5);

arrayList2.add(7);

**System**.out.println("原始数组:");

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

**System**.out.println("Collections.max(arrayList):");

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

**System**.out.println("Collections.min(arrayList):");

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

**System**.out.println("Collections.replaceAll(arrayList, 3, -3):");

**Collections**.replaceAll(arrayList, 3, -3);

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

**System**.out.println("Collections.frequency(arrayList, -3):");

**System**.out.println(**Collections**.frequency(arrayList, -3));

**System**.out.println("Collections.indexOfSubList(arrayList, arrayList2):");

**System**.out.println(**Collections**.indexOfSubList(arrayList, arrayList2));

**System**.out.println("Collections.binarySearch(arrayList, 7):");

// 对List进行二分查找，返回索引，List必须是有序的

**Collections**.sort(arrayList);

**System**.out.println(**Collections**.binarySearch(arrayList, 7));

#### Arrays类的常见操作

* 排序 : sort()
* 查找 : binarySearch()
* 比较: equals()
* 填充 : fill()
* 转列表: asList()
* 转字符串 : toString()
* 复制: copyOf()

排序 : sort()

// \*\*\*\*\*\*\*\*\*\*\*\*\*排序 sort\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**int** a[] = { 1, 3, 2, 7, 6, 5, 4, 9 };

// sort(int[] a)方法按照数字顺序排列指定的数组。

**Arrays**.sort(a);

**System**.out.println("Arrays.sort(a):");

**for** (**int** i : a) {

**System**.out.print(i);

}

// 换行

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

// sort(int[] a,int fromIndex,int toIndex)按升序排列数组的指定范围

**int** b[] = { 1, 3, 2, 7, 6, 5, 4, 9 };

**Arrays**.sort(b, 2, 6);

**System**.out.println("Arrays.sort(b, 2, 6):");

**for** (**int** i : b) {

**System**.out.print(i);

}

// 换行

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

**int** c[] = { 1, 3, 2, 7, 6, 5, 4, 9 };

// parallelSort(int[] a) 按照数字顺序排列指定的数组(并行的)。同sort方法一样也有按范围的排序。

// parallelSort将数组分成子数组，并行排序，然后合并，适用于数组长度很大时。

**Arrays**.parallelSort(c);

**System**.out.println("Arrays.parallelSort(c)：");

**for** (**int** i : c) {

**System**.out.print(i);

}

// 换行

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

// parallelSort给字符数组排序，sort也可以

**char** d[] = { 'a', 'f', 'b', 'c', 'e', 'A', 'C', 'B' };

**Arrays**.parallelSort(d);

**System**.out.println("Arrays.parallelSort(d)：");

**for** (**char** d2 : d) {

**System**.out.print(d2);

}

// 换行

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

查找 : binarySearch()

// \*\*\*\*\*\*\*\*\*\*\*\*\*查找 binarySearch()\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**char**[] e = { 'a', 'f', 'b', 'c', 'e', 'A', 'C', 'B' };

// 排序后再进行二分查找，否则找不到

**Arrays**.sort(e);

**System**.out.println("Arrays.sort(e)" + **Arrays**.toString(e));

**System**.out.println("Arrays.binarySearch(e, 'c')：");

**int** s = **Arrays**.binarySearch(e, 'c');

**System**.out.println("字符c在数组的位置：" + s);

比较: equals()

// \*\*\*\*\*\*\*\*\*\*\*\*\*比较 equals\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**char**[] e = { 'a', 'f', 'b', 'c', 'e', 'A', 'C', 'B' };

**char**[] f = { 'a', 'f', 'b', 'c', 'e', 'A', 'C', 'B' };

/\*

\* 元素数量相同，并且相同位置的元素相同。 另外，如果两个数组引用都是null，则它们被认为是相等的 。

\*/

// 输出true

**System**.out.println("Arrays.equals(e, f):" + **Arrays**.equals(e, f));

填充 : fill()

// \*\*\*\*\*\*\*\*\*\*\*\*\*填充fill(批量初始化)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**int**[] g = { 1, 2, 3, 3, 3, 3, 6, 6, 6 };

// 数组中所有元素重新分配值

**Arrays**.fill(g, 3);

**System**.out.println("Arrays.fill(g, 3)：");

// 输出结果：333333333

**for** (**int** i : g) {

**System**.out.print(i);

}

// 换行

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

**int**[] h = { 1, 2, 3, 3, 3, 3, 6, 6, 6, };

// 数组中指定范围元素重新分配值

**Arrays**.fill(h, 0, 2, 9);

**System**.out.println("Arrays.fill(h, 0, 2, 9);：");

// 输出结果：993333666

**for** (**int** i : h) {

**System**.out.print(i);

}

转列表 asList()

// \*\*\*\*\*\*\*\*\*\*\*\*\*转列表 asList()\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

\* 返回由指定数组支持的固定大小的列表。

\* （将返回的列表更改为“写入数组”。）该方法作为基于数组和基于集合的API之间的桥梁，与Collection.toArray()相结合 。

\* 返回的列表是可序列化的，并实现RandomAccess 。

\* 此方法还提供了一种方便的方式来创建一个初始化为包含几个元素的固定大小的列表如下：

\*/

**List**<**String**> stooges = **Arrays**.asList("Larry", "Moe", "Curly");

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

转字符串 toString()

// \*\*\*\*\*\*\*\*\*\*\*\*\*转字符串 toString()\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

\* 返回指定数组的内容的字符串表示形式。

\*/

**char**[] k = { 'a', 'f', 'b', 'c', 'e', 'A', 'C', 'B' };

**System**.out.println(**Arrays**.toString(k));// [a, f, b, c, e, A, C, B]

复制 copyOf()

// \*\*\*\*\*\*\*\*\*\*\*\*\*复制 copy\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// copyOf 方法实现数组复制,h为数组，6为复制的长度

**int**[] h = { 1, 2, 3, 3, 3, 3, 6, 6, 6, };

**int** i[] = **Arrays**.copyOf(h, 6);

**System**.out.println("Arrays.copyOf(h, 6);：");

// 输出结果：123333

**for** (**int** j : i) {

**System**.out.print(j);

}

// 换行

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

// copyOfRange将指定数组的指定范围复制到新数组中

**int** j[] = **Arrays**.copyOfRange(h, 6, 11);

**System**.out.println("Arrays.copyOfRange(h, 6, 11)：");

// 输出结果66600(h数组只有9个元素这里是从索引6到索引11复制所以不足的就为0)

**for** (**int** j2 : j) {

**System**.out.print(j2);

}

// 换行

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

### 1.11枚举

1使用枚举会有更强的类型约束，编译器帮助检查入参类型，规避风险

2常用方法：

ordinal 序数

name 名称

valueOf 根据字符串取枚举对象

3 扩展：在枚举类中可以声明属性、构造函数和方法

常量必须在一开始定义，构造方法只能用private修饰

4 专用于枚举的集合类 EnumSet/EnumMap

5 使用枚举实现设计模式：单例模式/策略模式