

## 第13讲 | 谈谈接口和抽象类有什么区别？

2018-06-02 杨晓峰



### 第13讲 | 谈谈接口和抽象类有什么区别？

朗读人：黄洲君 11'06" | 5.09M

Java 是非常典型的面向对象语言，曾经有一段时间，程序员整天把面向对象、设计模式挂在嘴边。虽然如今大家对这方面已经不再那么狂热，但是不可否认，掌握面向对象设计原则和技巧，是保证高质量代码的基础之一。

面向对象提供的基本机制，对于提高开发、沟通等各方面效率至关重要。考察面向对象也是面试中的常见一环，下面我来聊聊面向对象设计基础。

今天我要问你的问题是，**谈谈接口和抽象类有什么区别？**

### 典型回答

接口和抽象类是 Java 面向对象设计的两个基础机制。


接口是对行为的抽象，它是抽象方法的集合，利用接口可以达到 API 定义和实现分离的目的。接口，不能实例化；不能包含任何非常量成员，任何 field 都是隐含着 public static final 的意

义；同时，没有非静态方法实现，也就是说要么是抽象方法，要么是静态方法。Java 标准类库中，定义了非常多的接口，比如 `java.util.List`。

抽象类是不能实例化的类，用 `abstract` 关键字修饰 `class`，其目的主要是代码重用。除了不能实例化，形式上和一般的 Java 类并没有太大区别，可以有一个或者多个抽象方法，也可以没有抽象方法。抽象类大多用于抽取相关 Java 类的共用方法实现或者是共同成员变量，然后通过继承的方式达到代码复用的目的。Java 标准库中，比如 `collection` 框架，很多通用部分就被抽取成为抽象类，例如 `java.util.AbstractList`。

Java 类实现 `interface` 使用 `implements` 关键词，继承 `abstract class` 则是使用 `extends` 关键词，我们可以参考 Java 标准库中的 `ArrayList`。

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    //...
}
```

 复制代码

## 考点分析

这是个非常高频的 Java 面向对象基础问题，看起来非常简单的问题，如果面试官稍微深入一些，你会发现很多有意思的地方，可以从不同角度全面地考察你对基本机制的理解和掌握。比如：

- 对于 Java 的基本元素的语法是否理解准确。能否定义出语法基本正确的接口、抽象类或者相关继承实现，涉及重载（Overload）、重写（Override）更是有各种不同的题目。
- 在软件设计开发中妥善地使用接口和抽象类。你至少知道典型应用场景，掌握基础类库重要接口的使用；掌握设计方法，能够在 review 代码的时候看出明显的不利于未来维护的设计。
- 掌握 Java 语言特性演进。现在非常多的框架已经是基于 Java 8，并逐渐支持更新版本，掌握相关语法，理解设计目的是很有必要的。

## 知识扩展

我会从接口、抽象类的一些实践，以及语言变化方面去阐述一些扩展知识点。

Java 相比于其他面向对象语言，如 C++，设计上有一些基本区别，比如 Java 不支持多继承。这种限制，在规范了代码实现的同时，也产生了一些局限性，影响着程序设计结构。Java 类可以实现多个接口，因为接口是抽象方法的集合，所以这是声明性的，但不能通过扩展多个抽象类来重用逻辑。

在一些情况下存在特定场景，需要抽象出与具体实现、实例化无关的通用逻辑，或者纯调用关系的逻辑，但是使用传统的抽象类会陷入到单继承的窘境。以往常见的做法是，实现由静态方法组成的工具类（Utils），比如 `java.util.Collections`。

设想，为接口添加任何抽象方法，相应的所有实现了这个接口的类，也必须实现新增方法，否则会出现编译错误。对于抽象类，如果我们添加非抽象方法，其子类只会享受到能力扩展，而不用担心编译出问题。

接口的职责也不仅仅限于抽象方法的集合，其实有各种不同的实践。有一类没有任何方法的接口，通常叫作 Marker Interface，顾名思义，它的目的就是为了声明某些东西，比如我们熟知的 `Cloneable`、`Serializable` 等。这种用法，也存在于业界其他的 Java 产品代码中。

从表面看，这似乎和 Annotation 异曲同工，也确实如此，它的好处是简单直接。对于 Annotation，因为可以指定参数和值，在表达能力上要更强大一些，所以更多人选择使用 Annotation。


Java 8 增加了函数式编程的支持，所以又增加了一类定义，即所谓 functional interface，简单说就是只有一个抽象方法的接口，通常建议使用 `@FunctionalInterface` Annotation 来标记。Lambda 表达式本身可以看作是一类 functional interface，某种程度上这和面向对象可以算是两码事。我们熟知的 `Runnable`、`Callable` 之类，都是 functional interface，这里不再多介绍了，有兴趣你可以参考：<https://www.oreilly.com/learning/java-8-functional-interfaces>。

还有一点可能让人感到意外，严格说，Java 8 以后，接口也是可以有方法实现的！

从 Java 8 开始，interface 增加了对 default method 的支持。Java 9 以后，甚至可以定义 private default method。Default method 提供了一种二进制兼容的扩展已有接口的办法。比如，我们熟知的 `java.util.Collection`，它是 collection 体系的 root interface，在 Java 8 中添加了一系列 default method，主要是增加 Lambda、Stream 相关的功能。我在专栏前面提到的类似 `Collections` 之类的工具类，很多方法都适合作为 default method 实现在基础接口里面。

你可以参考下面代码片段：

```
public interface Collection<E> extends Iterable<E> {  
    /**  
     * Returns a sequential Stream with this collection as its source  
     * ...  
     */  
    default Stream<E> stream() {  
  
        return StreamSupport.stream(spliterator(), false);  
    }  
}
```

 复制代码

```
        return someApp.getSomeComponent().getSomeComponent();
    }
}
```

## 面向对象设计

谈到面向对象，很多人就会想起设计模式，那些是非常经典的问题和设计方法的总结。我今天来夯实一下基础，先来聊聊面向对象设计的基本方面。

我们一定要清楚面向对象的基本要素：封装、继承、多态。

封装的目的是隐藏事务内部的实现细节，以便提高安全性和简化编程。封装提供了合理的边界，避免外部调用者接触到内部的细节。我们在日常开发中，因为无意间暴露了细节导致的难缠 bug 太多了，比如多线程环境暴露内部状态，导致的并发修改问题。从另外一个角度看，封装这种隐藏，也提供了简化的界面，避免太多无意义的细节浪费调用者的精力。


继承是代码复用的基础机制，类似于我们对于马、白马、黑马的归纳总结。但要注意，继承可以看作是非常紧耦合的一种关系，父类代码修改，子类行为也会变动。在实践中，过度滥用继承，可能会起到反效果。

多态，你可能立即会想到重写（override）和重载（overload）、向上转型。简单说，重写是父子类中相同名字和参数的方法，不同的实现；重载则是相同名字的方法，但是不同的参数，本质上这些方法签名是不一样的，为了更好说明，请参考下面的样例代码：

```
public int doSomething() {
    return 0;
}

// 输入参数不同，意味着方法签名不同，重载的体现
public int doSomething(List<String> str) {
    return 0;
}

// return 类型不一样，编译不能通过
public short doSomething() {
    return 0;
}
```

 复制代码

这里你可以思考一个小问题，方法名称和参数一致，但是返回值不同，这种情况在 Java 代码中算是有效的重载吗？答案是不是的，编译都会出错的。


进行面向对象编程，掌握基本的设计原则是必须的，我今天介绍最通用的部分，也就是所谓的 S.O.L.I.D 原则。

- 单一职责 (Single Responsibility) ， 类或者对象最好是只有单一职责， 在程序设计中如果发现某个类承担着多种义务， 可以考虑进行拆分。
- 开关原则 (Open-Close, Open for extension, close for modification) ， 设计要对扩展开放， 对修改关闭。换句话说， 程序设计应保证平滑的扩展性， 尽量避免因为新增同类功能而修改已有实现， 这样可以少产出些回归 (regression) 问题。
- 里氏替换 (Liskov Substitution) ， 这是面向对象的基本要素之一， 进行继承关系抽象时， 凡是可以父类或者基类的地方， 都可以用子类替换。
- 接口分离 (Interface Segregation) ， 我们在进行类和接口设计时， 如果在一个接口里定义了太多方法， 其子类很可能面临两难， 就是只有部分方法对它是有意義的， 这就破坏了程序的内聚性。  
对于这种情况， 可以通过拆分成功能单一的多个接口， 将行为进行解耦。在未来维护中， 如果某个接口设计有变， 不会对使用其他接口的子类构成影响。
- 依赖反转 (Dependency Inversion) ， 实体应该依赖于抽象而不是实现。也就是说高层次模块， 不应该依赖于低层次模块， 而是应该基于抽象。实践这一原则是保证产品代码之间适当耦合度的法宝。

## OOP 原则实践中的取舍


值得注意的是， 现代语言的发展， 很多时候并不是完全遵守前面的原则的， 比如， Java 10 中引入了本地方法类型推断和 var 类型。按照， 里氏替换原则， 我们通常这样定义变量：

```
List<String> list = new ArrayList<>();
```

 复制代码


如果使用 var 类型， 可以简化为

```
var list = new ArrayList<String>();
```

 复制代码

但是， list 实际会被推断为 “ArrayList < String >”


```
ArrayList<String> list = new ArrayList<String>();
```

 复制代码

理论上，这种语法上的便利，其实是增强了程序对实现的依赖，但是微小的类型泄漏却带来了书写的便利和代码可读性的提高，所以，实践中我们还是要按照得失利弊进行选择，而不是一味得遵循原则。

## OOP 原则在面试题目中的分析


我在以往面试中发现，即使是有多年编程经验的工程师，也还没有真正掌握面向对象设计的基本原则，如开关原则（Open-Close）。看看下面这段代码，改编自朋友圈盛传的某伟大公司产品代码，你觉得可以利用面向对象设计原则如何改进？

 复制代码

```
public class VIPCenter {  
    void serviceVIP(T extend User user>) {  
        if (user instanceof SlumDogVIP) {  
            // 穷 X VIP，活动抢的那种  
            // do something  
        } else if (user instanceof RealVIP) {  
            // do something  
        }  
        // ...  
    }  
}
```

这段代码的一个问题是，业务逻辑集中在一起，当出现新的用户类型时，比如，大数据发现了我们是肥羊，需要去收获一下，这就需要直接去修改服务方法代码实现，这可能会意外影响不相关的某个用户类型逻辑。

利用开关原则，我们可以尝试改造为下面的代码：

 复制代码

```
public class VIPCenter {  
    private Map<User.TYPE, ServiceProvider> providers;  
    void serviceVIP(T extend User user) {  
        providers.get(user.getType()).service(user);  
    }  
}  
  
interface ServiceProvider{  
    void service(T extend User user) ;  
}  
  
class SlumDogVIPServiceProvider implements ServiceProvider{  
  
    void service(T extend User user){
```

```
        // do something
    }
}

class RealVIPServiceProvider implements ServiceProvider{
    void service(T extend User user) {
        // do something
    }
}
```

上面的示例，将不同对象分类的服务方法进行抽象，把业务逻辑的紧耦合关系拆开，实现代码的隔离保证了方便的扩展。

今天我对 Java 面向对象技术进行了梳理，对比了抽象类和接口，分析了 Java 语言在接口层面的演进和相应程序设计实现，最后回顾并实践了面向对象设计的基本原则，希望对你有所帮助。

### 一课一练

关于接口和抽象类的区别，你做到心中有数了吗？给你布置一个思考题，思考一下自己的产品代码，有没有什么地方违反了基本设计原则？那些一改就崩的代码，是否遵循了开关原则？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



# Java 核心技术 36 讲

—— Oracle 首席工程师  
带你修炼 Java 内功 ——

杨晓峰 Oracle 首席工程师





精选留言



公号-Java大后端  
接口vs抽象类vs类

👍 36

- 1 支持多重继承：接口支持；抽象类不支持；类不支持；
- 2 支持抽象函数：接口语义上支持；抽象类支持；类不支持；
- 3 允许函数实现：接口不允许；抽象类支持；类允许；
- 4 允许实例化：接口不允许；抽象类不允许；类允许；
- 5 允许部分函数实现：接口不允许；抽象类允许；类不允许。
- 6 定义的内容：接口中只能包括public函数以及public static final常量；抽象类与类均无任何限制。
- 7 使用时机：当想要支持多重继承，或是为了定义一种类型请使用接口；当打算提供带有部分实现的“模板”类，而将一些功能需要延迟实现请使用抽象类；当你打算提供完整的具体实现请使用类。

在实际项目开发过程，一方面是业务需求频繁，需要满足开闭原则，也就是小到一个模块，大到一个架构都需要有好的可扩展性；另外一方面软件往往是团队协作开发的过程；由于团队成员水平参差不齐，这方面的坑不少。可以通过前期做好设计评审、code review等手段去提升代码质量。

2018-06-02



ωǒ→无悔  
最后一个例子就是策略模式加工厂模式

👍 16

2018-06-02



Woong  
class SlumDogVIPServiceProvider和RealVIPServiceProvider缺少implements.

👍 7

2018-06-02

作者回复

汗，手敲搞出这种低级错误，非常感谢指出

2018-06-03



张立春  
我理解继承的根本目的是为了多态而不是为了复用，如果仅为了复用那就采用松耦合的组合。

👍 5

2018-06-02



Seven4X

👍 3



对象和抽象类是is a 的关系，对象和接口是 like this的关系。

从接口的命名一般是able ability后缀表示一种能力。

比如大家都是程序员，如果拥有了编写java程序的能力，就可以说你是一个java程序员，同时你还如果你还会编写Go，也可以说你是一个go程序员。

程序员就是对象实例，java编程能力就是接口，如果把java编程能力定义为抽象类，那还要实现Go能力接口才能同时具有两种语言能力的程序员，这表现的便是一个以java编程为主的程序员，同时具有编写go的能力。

如果把java能力和go能力都定义为接口，表现的是：这是一个程序员他同时具有编写java和Go的能力。

2018-06-16



j.c.

👍 3

同种类型行为是通过不同方法调用还是不同实现类，哪个好点？

2018-06-02



qpm

👍 3

hi，老师早上好。我是一家游戏公司的程序员，由于项目非常紧，所以很多技术写的代码，都非常乱。我们的战斗系统中，之前的开发模式是在战斗逻辑里面嵌入并修改一些代码，以达到新技能的开发。这就是典型的以修改来达到需求。通过重构之后，技能的逻辑通过扩展的方式开发出来，可以通过继承技能的抽象类，来完成技能的开发。现在我们这部分的模块从最难处理变成最容易开发的了。

2018-06-02

### 作者回复

听说过电信代码里有26个if-else.....然后还有俩是重复的，少的时候无所谓，多了就是坑

2018-06-03



小情绪

👍 2

杨老师，对于开头的：接口中没有非静态方法实现，也就是说要么是抽象方法，要么是静态方法。这句话我有疑问，java.util.List中default 方法不就是非静态方法的实现吗？还是我理解有误？

2018-06-03

### 作者回复

前面就是个举例的回答，用来后面分析的，新版Java不准确了

2018-06-05



雷霹雳的爸爸

👍 2

问题本身就是典型热身题，但是SOLID则是一块试金石，曾经真的以为是试金石，很多老江湖好像都不知道有这串缩写这么回事，单一职责一说就都是顾名思义，纯粹的同义反复，用自己解释自己跟没说差不多吧，里氏替换背后的契约设计基本规则就更甭说了，实际项目中能不违反的几乎凤毛麟角吧，除非对接口调用后就真的没有约束条件，连spring这货都崇尚什么都封装成非受控异常，可见这个点到底有多么不受人待见，最小知识原则也基本上就是任人践踏吧，要不全变函数接口也就都没必要反复换个名字了吧，依赖倒置还好，好歹有IoC帮衬着，但是helper和各种静态的tool被无数人个性的反复的造，同时拿着依赖查找的实用

性来堵DIP的嘴，也就剩OCP，再违反就太不像话了吧，没关系最大的伤害就是无视啊，帅气的蓝精灵命名法的那一串类，依次加个方法还是客气的，直接加个version字段，把if...else请回来，总之这玩意一说我就陷入到吐槽情绪，但问题吐的是solid的槽，还是吐的不把solid当回事的槽，我自己也分不清了，这个问题真的好让人纠结

2018-06-02



bamboo

👍 2

老师最后举的例子应该就是把简单工厂模式修改为工厂方法模式。原来的违反了开闭选择，工厂方法模式刚好弥补了这个问题，倒是相应的系统中的类个数也成对的增加。设计模式没有最优的，只是特定场景下我们选择相对优良的模式来优化我们的逻辑。不知道是否正确，望老师指点迷津，谢谢老师。😊

2018-06-02

作者回复

嗯，也要避免过度设计，这个只是举例

2018-06-03



clz1341521

👍 1

目前项目中很多代码工具类都是直接面向实现，才用static 方法。如日期类型转换，string操作等。这类方法有什么好的改进意见？

2018-08-05



Seven.Lin

👍 1

确实是的，目前还是许多程序员用面向对象语言书写面向过程的代码。实际工作中应该从业务流抽离出技术流，把类似if-else的处理封装，即使只是把代码抽离为方法；对于代码结构类似的（如，检查参数->业务处理->保存），应该更高一级的抽象，分离出业务模板（在JAVA表现就是一个接口或者抽象类）；在每一次编程之前考虑其扩展性和松耦合度，避免过度依赖导致后期难以维护。

2018-07-30



卡斯瓦德

👍 1

其实interface的default可以通过结合抽象父类来实现吧，抽象父类实现接口方法，但是因为抽象所以不能实例化，而其子类拥有重写权，可以做到default的效果

2018-07-14

作者回复

不错，只是：

抽象类不能多继承；

default method不会打破现有代码兼容性，lambda需要靠它来无缝增强collection之类API

2018-07-14



zc

👍 1

“这里你可以思考一个小问题，方法名称和参数一致，但是返回值不同，这种情况在Java代码中算是有效的重载吗？答案是不是的，编译都会出错的。”

编译出错是因为重载必须参数不一样，重载与返回值无关。感觉这里的表述有点问题.....

示例代码既不属于重写（同名同参同返回）也不属于重载（同名不同参）的范畴。

2018-07-12



000

抽象方法不实现，就是为了给子类用的吗？

2018-07-12

👍 1



仙道

两个接口里有一个同名方法，然后一个类实现了这两个接口，这怎么办呢

2018-07-02

👍 1



夏洛克的救赎

但是，list 实际会被推断为 “ArrayList < String >”

```
ArrayList<String> list = new ArrayList<String>();
```

不是很理解

2018-06-17

作者回复

一般我们会写成：

```
list<String> list=new ArrayList.....;
```

里氏代换，可以参考oo基础那章

2018-06-20

👍 1



张健

没解释多态

方法重载这里，如果用jdk javac编译就会过的

2018-06-14

👍 1



云学

其实有了函数式编程，绝大部分设计模式是多余的，记住，类的数量不要泛滥！！

2018-06-13

👍 1



Yao

问个问题，jdk8 default 是否应该使用？？？

2018-06-12

👍 1

作者回复

看你的需求啊，比如代码需不需要兼容老的jdk版本；有没有需要用的逻辑

2018-06-12

