13 谈谈接口和抽象类有什么区别？-极客时间

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
{
//...
}
```

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);
     }
  }
```

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

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，凡是可以用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义的，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<>();
```

如果使用 var 类型，可以简化为

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

但是，list 实际会被推断为“ArrayList < String >”

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

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的原则，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？

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

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：

```
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 somthing
  }
}
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。

上一页 下一页