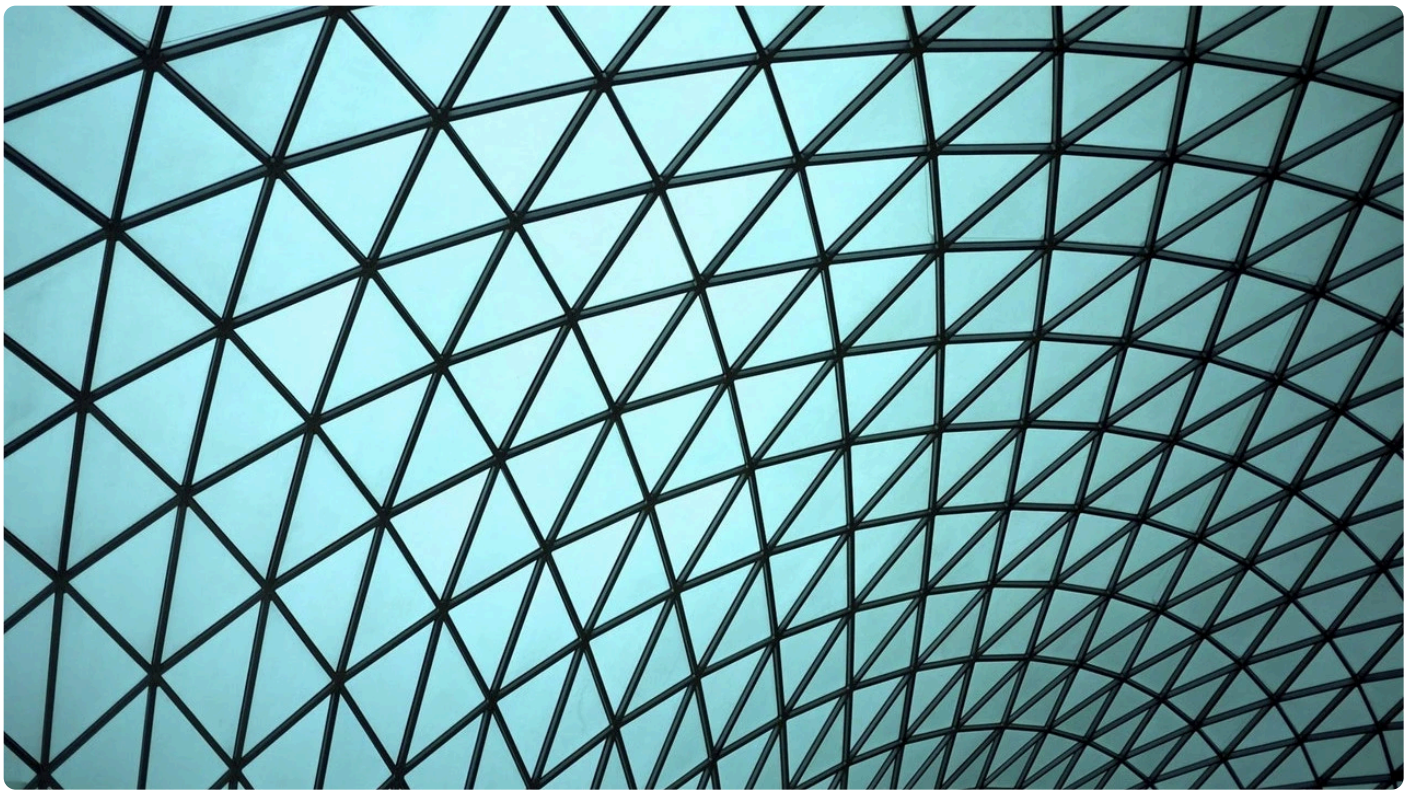
09 | 理论六：为什么基于接口而非实现编程？有必要为每个类都定义接口吗？

王争 · 设计模式之美



在上一节课中，我们讲了接口和抽象类，以及各种编程语言是如何支持、实现这两个语法概念的。今天，我们继续讲一个跟“接口”相关的知识点：基于接口而非实现编程。这个原则非常重要，是一种非常有效的提高代码质量的手段，在平时的开发中特别经常被用到。

为了让你理解透彻，并真正掌握这条原则如何应用，今天，我会结合一个有关图片存储的实战案例来讲解。除此之外，这条原则还很容易被过度应用，比如为每一个实现类都定义对应的接口。针对这类问题，在今天的讲解中，我也会告诉你如何来做权衡，怎样恰到好处地应用这条原则。

话不多说，让我们正式开始今天的学习吧！

如何解读原则中的“接口”二字？

“基于接口而非实现编程”这条原则的英文描述是：“Program to an interface, not an implementation”。我们理解这条原则的时候，千万不要一开始就与具体的编程语言挂钩，局限在编程语言的“接口”语法中（比如 Java 中的 interface 接口语法）。这条原则最早出现于 1994 年 GoF 的《设计模式》这本书，它先于很多编程语言而诞生（比如 Java 语言），是一条比较抽象、泛化的设计思想。

实际上，理解这条原则的关键，就是理解其中的“接口”两个字。还记得我们上一节课讲的“接口”的定义吗？从本质上来看，“接口”就是一组“协议”或者“约定”，是功能提供者提供给使用者的一个“功能列表”。“接口”在不同的应用场景下会有不同的解读，比如服务端与客户端之间的“接口”，类库提供的“接口”，甚至是一组通信的协议都可以叫作“接口”。刚刚对“接口”的理解，都比较偏上层、偏抽象，与实际的写代码离得有点远。如果落实到具体的编码，“基于接口而非实现编程”这条原则中的“接口”，可以理解为编程语言中的接口或者抽象类。

前面我们提到，这条原则能非常有效地提高代码质量，之所以这么说，那是因为，应用这条原则，可以将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低耦合性，提高扩展性。

实际上，“基于接口而非实现编程”这条原则的另一个表述方式，是“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。在软件开发中，最大的挑战之一就是需求的不断变化，这也是考验代码设计好坏的一个标准。**越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计，不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能够在不破坏原有代码设计的情况下灵活应对。**而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。

如何将这条原则应用到实战中？

对于这条原则，我们结合一个具体的实战案例来进一步讲解一下。

假设我们的系统中有很多涉及图片处理和存储的业务逻辑。图片经过处理之后被上传到阿里云上。为了代码复用，我们封装了图片存储相关的代码逻辑，提供了一个统一的 AliyunImageStore 类，供整个系统来使用。具体的代码实现如下所示：

复制代码

```
public class AliyunImageStore {
  //...省略属性、构造函数等...

  public void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket代码逻辑...
    // ...失败会抛出异常..
  }

  public String generateAccessToken() {
    // ...根据accesskey/secrectkey等生成access token
  }

  public String uploadToAliyun(Image image, String bucketName, String accessToken
    //...上传图片到阿里云...
    //...返回图片存储在阿里云上的地址(url) ...
  }

  public Image downloadFromAliyun(String url, String accessToken) {
    //...从阿里云下载图片...
  }
}

// AliyunImageStore类的使用举例
public class ImageProcessingJob {
  private static final String BUCKET_NAME = "ai_images_bucket";
  //...省略其他无关代码...

  public void process() {
    Image image = ...; //处理图片，并封装为Image对象
    AliyunImageStore imageStore = new AliyunImageStore(/*省略参数*/);
    imageStore.createBucketIfNotExisting(BUCKET_NAME);
    String accessToken = imageStore.generateAccessToken();
    imagestore.uploadToAliyun(image, BUCKET_NAME, accessToken);
  }

}
```

整个上传流程包含三个步骤：创建 bucket（你可以简单理解为存储目录）、生成 access token 访问凭证、携带 access token 上传图片到指定的 bucket 中。代码实现非常简单，类中的几个方法定义得都很干净，用起来也很清晰，乍看起来没有太大问题，完全能满足我们将图片存储在阿里云的业务需求。

不过，软件开发中唯一不变的就是变化。过了一段时间后，我们自建了私有云，不再将图片存储到阿里云了，而是将图片存储到自建私有云上。为了满足这样一个需求的变化，我们该如何修改代码呢？

我们需要重新设计实现一个存储图片到私有云的 PrivateImageStore 类，并用它替换掉项目中所有的 AliyunImageStore 类对象。这样的修改听起来并不复杂，只是简单替换而已，对整个代码的改动并不大。不过，我们经常说，“细节是魔鬼”。这句话在软件开发中特别适用。实际上，刚刚的设计实现方式，就隐藏了很多容易出问题的“魔鬼细节”，我们一块来看看都有哪些。

新的 PrivateImageStore 类需要设计实现哪些方法，才能在尽量最小化代码修改的情况下，替换掉 AliyunImageStore 类呢？这就要求我们必须将 AliyunImageStore 类中所定义的所有 public 方法，在 PrivateImageStore 类中都逐一定义并重新实现一遍。而这样做就会存在一些问题，我总结了下面两点。

首先，AliyunImageStore 类中有些函数命名暴露了实现细节，比如，uploadToAliyun() 和 downloadFromAliyun()。如果开发这个功能的同事没有接口意识、抽象思维，那这种暴露实现细节的命名方式就不足为奇了，毕竟最初我们只考虑将图片存储在阿里云上。而我们把这种包含“aliyun”字眼的方法，照抄到 PrivateImageStore 类中，显然是不合适的。如果我们在新类中重新命名 uploadToAliyun()、downloadFromAliyun() 这些方法，那就意味着，我们要修改项目中所有使用到这两个方法的代码，代码修改量可能就会很大。

其次，将图片存储到阿里云的流程，跟存储到私有云的流程，可能并不是完全一致的。比如，阿里云的图片上传和下载的过程中，需要生产 access token，而私有云不需要 access token。一方面，AliyunImageStore 中定义的 generateAccessToken() 方法不能照抄到 PrivateImageStore 中；另一方面，我们在使用 AliyunImageStore 上传、下载图片的时候，代码中用到了 generateAccessToken() 方法，如果要改为私有云的上传下载流程，这些代码都需要做调整。

那这两个问题该如何解决呢？解决这个问题的根本方法就是，在编写代码的时候，要遵从“基于接口而非实现编程”的原则，具体来讲，我们需要做到下面这 3 点。

1. 函数的命名不能暴露任何实现细节。比如，前面提到的 uploadToAliyun() 就不符合要求，应该改为去掉 aliyun 这样的字眼，改为更加抽象的命名方式，比如：upload()。
2. 封装具体的实现细节。比如，跟阿里云相关的特殊上传（或下载）流程不应该暴露给调用者。我们对上传（或下载）流程进行封装，对外提供一个包裹所有上传（或下载）细节的方法，给调用者使用。
3. 为实现类定义抽象的接口。具体的实现类都依赖统一的接口定义，遵从一致的上传功能协议。使用者依赖接口，而不是具体的实现类来编程。

我们按照这个思路，把代码重构一下。重构后的代码如下所示：

复制代码

```
public interface ImageStore {
  String upload(Image image, String bucketName);
  Image download(String url);
}

public class AliyunImageStore implements ImageStore {
  //...省略属性、构造函数等...

  public String upload(Image image, String bucketName) {
    createBucketIfNotExisting(bucketName);
    String accessToken = generateAccessToken();
    //...上传图片到阿里云...
    //...返回图片在阿里云上的地址(url)...
  }

  public Image download(String url) {
    String accessToken = generateAccessToken();
    //...从阿里云下载图片...
  }

  private void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket...
    // ...失败会抛出异常..
  }

  private String generateAccessToken() {
    // ...根据accesskey/secrectkey等生成access token
  }
}

// 上传下载流程改变：私有云不需要支持access token
```

```
public class PrivateImageStore implements ImageStore  {
  public String upload(Image image, String bucketName) {
    createBucketIfNotExisting(bucketName);
    //...上传图片到私有云...
    //...返回图片的url...
  }

  public Image download(String url) {
    //...从私有云下载图片...
  }

  private void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket...
    // ...失败会抛出异常..
  }
}

// ImageStore的使用举例
public class ImageProcessingJob {
  private static final String BUCKET_NAME = "ai_images_bucket";
  //...省略其他无关代码...

  public void process() {
    Image image = ...;//处理图片，并封装为Image对象
    ImageStore imageStore = new PrivateImageStore(...);
    imagestore.upload(image, BUCKET_NAME);
  }
}
```

除此之外，很多人在定义接口的时候，希望通过实现类来反推接口的定义。先把实现类写好，然后看实现类中有哪些方法，照抄到接口定义中。如果按照这种思考方式，就有可能导致接口定义不够抽象，依赖具体的实现。这样的接口设计就没有意义了。不过，如果你觉得这种思考方式更加顺畅，那也没问题，只是将实现类的方法搬移到接口定义中的时候，要有选择性的搬移，不要将跟具体实现相关的方法搬移到接口中，比如 AliyunImageStore 中的 generateAccessToken() 方法。

总结一下，我们在做软件开发的时候，一定要有抽象意识、封装意识、接口意识。在定义接口的时候，不要暴露任何实现细节。接口的定义只表明做什么，而不是怎么做。而且，在设计接口的时候，我们要多思考一下，这样的接口设计是否足够通用，是否能够做到在替换具体的接口实现的时候，不需要任何接口定义的改动。

是否需要为每个类定义接口?

看了刚刚的讲解，你可能会有这样的疑问：为了满足这条原则，我是不是需要给每个实现类都定义对应的接口呢？在开发的时候，是不是任何代码都要只依赖接口，完全不依赖实现编程呢?

做任何事情都要讲求一个“度”，过度使用这条原则，非得给每个类都定义接口，接口满天飞，也会导致不必要的开发负担。至于什么时候，该为某个类定义接口，实现基于接口的编程，什么时候不需要定义接口，直接使用实现类编程，我们做权衡的根本依据，还是要回归到设计原则诞生的初衷上来。只要搞清楚了这条原则是为了解决什么样的问题而产生的，你就会发现，很多之前模棱两可的问题，都会变得豁然开朗。

前面我们也提到，这条原则的设计初衷是，将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低代码间的耦合性，提高代码的扩展性。

从这个设计初衷上来看，如果在我们的业务场景中，某个功能只有一种实现方式，未来也不可能被其他实现方式替换，那我们就没有必要为其设计接口，也没有必要基于接口编程，直接使用实现类就可以了。

除此之外，越是不稳定的系统，我们越是要在代码的扩展性、维护性上下功夫。相反，如果某个系统特别稳定，在开发完之后，基本上不需要做维护，那我们就没有必要为其扩展性，投入不必要的开发时间。

重点回顾

今天的内容到此就讲完了。我们来一块总结回顾一下，你需要掌握的重点内容。

1.“基于接口而非实现编程”，这条原则的另一个表述方式，是“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。我们在做软件开发的时候，一定要有抽象意识、封装意识、接口意识。越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性、扩展性、可维护性。

2. 我们在定义接口的时候，一方面，命名要足够通用，不能包含跟具体实现相关的字眼；另一方面，与特定实现有关的方法不要定义在接口中。

3.“基于接口而非实现编程”这条原则，不仅仅可以指导非常细节的编程开发，还能指导更加上层的架构设计、系统设计等。比如，服务端与客户端之间的“接口”设计、类库的“接口”设计。

课堂讨论

在今天举的代码例子中，尽管我们通过接口来隔离了两个具体的实现。但是，在项目中很多地方，我们都是通过下面第 8 行的方式来使用接口的。这就会产生一个问题，那就是，如果我们要替换图片存储方式，还是需要修改很多类似第 8 行那样的代码。这样的设计还是不够完美，对此，你有更好的实现思路吗?

复制代码

```
1  // ImageStore的使用举例
2  public class ImageProcessingJob {
3    private static final String BUCKET_NAME = "ai_images_bucket";
4    //...省略其他无关代码...
5
6    public void process() {
7      Image image = ...;//处理图片，并封装为Image对象
8      ImageStore imageStore = new PrivateImageStore(/*省略构造函数*/);
9      imagestore.upload(image, BUCKET_NAME);
10   }
```

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

本文深入探讨了“基于接口而非实现编程”的重要性以及如何在实际编码中应用这一原则。作者首先解释了“基于接口而非实现编程”原则的含义，强调了将接口和实现相分离，封装不稳定的实现，暴露稳定的接口的重要性。文章指出，这一原则可以降低代码的耦合性，提高代码的扩展性，使代码更加灵活应对未来的需求变化。通过具体的实战案例，文章展示了如何在代码中应用这一原则，以及如何通过重构代码来遵循“基于接口而非实现编程”的原则。作者还提到了“基于抽象而非实现编程”是这一原则的另一种表述方式，强调了抽象是提高代码扩展性、灵活性、可维护性最有效的手段之一。总的来说，本文通过解释原则的含义、重要性以及如何在实际编码中应用，帮助读者更好地理解了“基于接口而非实现编程”的概念。文章还强调了在设计接口时要遵循抽象意识、封装意识、接口意识，不暴露任何实现细节，以及在业务场景中根据实现的稳定性来决定是否需要定义接口。

全部留言 (306)

最新　精选

👽
2019-11-25

那现在的MVC代码，要求service先写接口，然后再写实现，有必要嘛?
说实话，我一直没看懂这种行为的意义何在。

作者回复: 确实意义不大

共 19 条评论>　16

bearlu
2019-11-22

老师，希望能把示例代码和问题代码也放到Github上。

作者回复: 👌 我抽空整理一下放上去
https://github.com/wangzheng0822

共 2 条评论>　6

大智
2020-11-23

思考题的话，结合spring的话我觉得应该是初始化一个存储处理类并在使用类中@Autowired即可。初始化哪个类取决于你给哪个存储类进行了初始化

作者回复: 嗯嗯

共 2 条评论>　5

tuyu
2020-05-09

老师, 我今天碰到一个问题, 如果我创建阿里云k8s, 那么参数是很多很多, 根据接口原则, 我不太清楚未来不同云平台创建k8s需要哪些参数, 那我应该怎么做

作者回复: 这个就没法抽象成接口了，只能每个不同的云平台不同处理了。我们也没法追求在替换云平台时，一点代码都不改。尽量少改动代码就可以了。

5

Geek_East
2020-05-21

想问个问题，泛型的主要目的是代码复用还是抽象呀?

作者回复: 代码复用

3

刘小辉
2020-11-27

第一：思考题我觉得可以用SPI!
第二：　抽象类的功能应该远远多于接口。但是定义抽象类的代价是比较高的。因为像java,C#这样的高级语言，是不允许多继承的所以，你在设计一个父类为抽象类的时候，一定得将这个类的子类所有的共同属性和方法都定义出来；但是接口可以不用这样。因为接口是一个方法的集合，一个类可以实现多个接口。所以，你的接口里面只有一个方法，还是两个方法，都是可以的。之后如果还有新的方法，我大不了再设计一个接口就是了。所以说，抽象类的设计必须谨慎，接口的设计很灵活。

作者回复: ??????

2

Geek_5a5d9a
2020-11-17

我的想法是对外提供构造函数，构造不同场景的上传, 例如:

```
public ImageProcessingJob aliyunProcessingJob() {
        // 设置类的基本属性
        // self.imageStore = new AliyunImageStore()
    }
```

```
public ImageProcessingJob privateProcessingJob() {
    // 设置类的属性
    // self.imageStore = new PrivateImageStore()
}
```

作者回复: ❓❓❓❓❓❓

2

李二木
2020-11-24

我们在做软件开发的时候，一定要有抽象意识、封装意识、接口意识。越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性、扩展性、可维护性。

作者回复: 说的没错

刘永超
2020-11-14

老师把行业“黑话”讲透彻了，捅破那层窗户纸，不用再纠结于一些基本概念。

作者回复: 哈哈，爱你

varotene
2020-01-06

Programming against interface 是不是也是抽象(abstraction)的一种手段？把原来具体的问题或者实现（aliyun或者私有云）抽象成任意的云，然后通过interface来予以描述。
至于什么时候用interface，什么时候不用，感觉也跟什么时候用抽象什么时候不用抽象是一个道理？因为进行抽象是需要成本的， 但不需要的时候，我们就可以略过，节省工程成本（YAGNI原则）。这么理解对吗？

作者回复: 理解的没问题~ 基于接口而非实现是一种抽象的思维