



下载APP



## 22 | Liskov替换原则：用了继承，子类就设计对了吗？

2020-07-17 郑晔

软件设计之美

[进入课程 >](#)**讲述：郑晔**

时长 14:03 大小 12.87M



你好！我是郑晔。

上一讲，我们讲了开放封闭原则，想要让系统符合开放封闭原则，最重要的就是我们要构建起相应的扩展模型，所以，我们要面向接口编程。

而大部分的面向接口编程要依赖于继承实现，虽然我们在前面的课程中说过，继承的灵活性不如封装和多态，但在大部分面向对象程序设计语言中，继承却是构建一个对象体系的重要组成部分。



理论上，在定义了接口之后，我们就可以把继承这个接口的类完美地嵌入到我们设计好的体系之中。然而，用了继承，子类就一定设计对了吗？事情可能并没有这么简单。

新的类虽然在语法上声明了一个接口，形成了一个继承关系，但我们要想让这个子类真正地扮演起这个接口的角色，还需要有一个好的继承指导原则。

所以，这一讲，我们就来看看可以把继承体系设计好的设计原则：Liskov 替换法则。

## Liskov 替换原则

2008 年，图灵奖授予 Barbara Liskov，表彰她在程序设计语言和系统设计方法方面的卓越工作。她在设计领域影响最深远的就是以她名字命名的 Liskov 替换原则（Liskov substitution principle，简称 LSP）。

1988 年，Barbara Liskov 在描述如何定义子类型时写下这样一段话：

这里需要如下替换性质：若每个类型 S 的对象 o1，都存在一个类型 T 的对象 o2，使得在所有针对 T 编程的程序 P 中，用 o1 替换 o2 后，程序 P 行为保持不变，则 S 是 T 的子类型。


用通俗的讲法来说，意思就是，子类型（subtype）必须能够替换其父类型（base type）。

这句话看似简单，但是违反这个原则，后果是很严重的，比如，父类型规定接口不能抛出异常，而子类型抛出了异常，就会导致程序运行的失败。

虽然很好理解，但你可能会有个疑问，我的子类型不都是继承自父类型，咋就能违反 LSP 呢？这个 LSP 是不是有点多此一举呢？

我们来看个例子，有不少的人经常写出类似下面这样的代码：

```
1 void handle(final Handler handler) {  
2     if (handler instanceof ReportHandler) {  
3         // 生成报告  
4         ((ReportHandler)handler).report();  
    }
```

 复制代码

```
5     return;
6 }
7
8 if (handler instanceof NotificationHandler) {
9     // 发送通知
10    ((NotificationHandler)handler).sendNotification();
11 }
12 ...
13
```

根据上一讲的内容，这段代码显然是违反了 OCP 的。另外，在这个例子里面，虽然我们定义了一个父类型 Handler，但在这段代码的处理中，是通过运行时类型识别（Run-Time Type Identification，简称 RTTI），也就是这里的 instanceof，知道子类型是什么的，然后去做相应的业务处理。

但是，ReportHandler 和 NotificationHandler 虽然都是 Handler 的子类，但**它们没有统一的处理接口**，所以，它们之间并不存在一个可以替换的关系，这段代码也是违反 LSP 的。这里我们就得到了一个经验法则，**如果你发现了任何做运行时类型识别的代码，很有可能已经破坏了 LSP。**

## 基于行为的 IS-A

如果你去阅读关于 LSP 的资料，很有可能会遇到一个有趣的问题，也就是长方形正方形问题。在我们对于几何通常的理解中，正方形是一种特殊的长方形。所以，我们可能会写出这样的代码：

```
1 class Rectangle {
2     private int height;
3     private int width;
4
5     // 设置长度
6     public void setHeight(int height) {
7         this.height = height;
8     }
9
10    // 设置宽度
11    public void setWidth(int width) {
12        this.width = width;
13    }
14
15    //
16    public void area() {
```

[复制代码](#)

```
17     return this.height * this.width;
18 }
19 }
20
21 class Square extends Rectangle {
22     // 设置边长
23     public void setSide(int side) {
24         this.setHeight(side);
25         this.setWidth(side);
26     }
27 }
28
29 @Override
30 public void setHeight(int height) {
31     this.setSide(height);
32 }
33
34 @Override
35 public void setWidth(int width) {
36     this.setSide(width);
37 }
38 }
```

这段代码看上去一切都很好，然而，它却是有点问题的，因为它在下面这个测试里会失败：

[复制代码](#)

```
1 Rectangle rect = new Square();
2 rect.setHeight(4); // 设置长度
3 rect.setWidth(5);  // 设置宽度
4 assertThat(rect.area(), is(20)); // 对结果进行断言
```

如果想保证断言（assert）的正确性，Rectangle 和 Square 二者在这里是不能互相替换的。使用 Rectangle 的代码必须知道自己使用的到底是 Rectangle 还是 Square。

出现这个问题的原因就在于，我们构建模型时，会理所当然地把我们的直觉中的模型直接映射到代码模型上。在我们直觉中，正方形确实是一种长方形。

在我们设计的这个对象体系中，边长是可以调整的。然而，在几何的体系里面，长方形的边长是不能随意改变的，设置好了就是设置好了。换句话说，两个体系内，“长方形”的行为是不一致的。所以，在这个对象体系中，正方形边长即使可以调整，但正方形并不是一个长方形，也就是说，它们之间不满足 IS-A 关系。



你可能听说过继承要符合 IS-A 的关系，也就是说，**如果 A 是 B 的子类，就需要满足 A 是一个 B (A is a B)**。但你有没有想过，凭什么 A 是一个 B 呢？判断依据从何而来呢？

你应该知道，这种判定显然不能依靠直觉。其实，从前面的分析中，你也能看出一些端倪来，**IS-A 的判定是基于行为的**，只有行为相同，才能说是满足 IS-A 的关系。

这个道理说起来很简单，但在实际的工作中，我们时常就会走上歧途。我给你举个例子，我要做一个图片制作的网站，创作者可以在上面创作自己的内容，还可以发布自己创作的一些素材在网站上销售。显然，这个网站要提供一个销售的能力，那这个可以销售的素材算不算商品呢？

如果站在销售的角度看，它确实是一个商品，我们需要给它定价，需要让它支持后续的购买行为等等。从行为上看，素材也确实是商品，但它又与创作相关，我们需要知道它的作者是谁，需要知道它所应用的不同创作阶段等等，这些行为又与商品完全无关。

其实，在我们分析问题的时候，答案就已经呼之欲出了。这里的“素材”就不是一个“素材”，前面讲 SRP 的时候，我们已经做过类似的分析了，虽然我们在讨论的时候，用的是一个词“素材”，但创作者和销售却是两个不同的领域。

所以，如果我们把“素材”做一个拆分，这个问题就迎刃而解了。一个是“创作者素材”，一个是“可销售素材”，显然，“可销售素材”是一种商品，而“创作者素材”不是。

这是一种常见的概念混淆。产品经理在描述一个需求时，可能并不会注意到这是两个不同领域的概念，而程序员如果不好好分析一下，在概念上就会走偏，后续的问题将无穷无尽。

所以，IS-A 这个关系理解起来并不难，但在实际工作中，当它和其他一些问题混在一起的时候，它就不像看起来那么简单了。

到这里，你应该对 LSP 原则有了一些理解，**要满足 LSP，首先这个对象体系要有一个统一的接口，而不能各行其是，其次，子类要满足 IS-A 的关系。**

有了对 LSP 的理解，你再用它去衡量一些设计，就会发现一些问题。比如，程序员们最常用的数据结构 List，很多人都习惯地把它当做接口传来传去。在绝大多数场景下，使用它的目的只是为了传递一些数据，也就是为了从中读取数据，但 List 接口本身一般都有写的方法。

所以，尽管你的目的是读，但还是有人不小心写了，就会导致一些奇怪的问题。Google 的 Guava 库提供了一个 ImmutableList，在概念上做了改进。但为了配合现有的各种程序，它不得不继承自 List 接口，实际上，根本的问题并没有得到完全的解决。

还有一类常见的违反 LSP 的问题，就是继承数据结构。比如，我要实现包含多个学生的类，结果声明成：

[复制代码](#)

```
1 class Students extends ArrayList<Student> {  
2     ...  
3 }
```

这是一种非常直觉的设计，只要一继承 ArrayList，添加、获取的方法就都有了。但从我们前面讲的内容上来看，这显然是不好的，因为 Students 不是一个 ArrayList，不能满足 IS-A 关系。这种做法想做的就是实现继承，而我们在前面讲继承的时候，就说过这种做法的问题。

你会发现，LSP 的关注点让人把注意力放到父类上，而一旦子类成了重点，我们必须小心谨慎。在前面讲继承的时候，我们说过，关心子类是一种实现继承的表现，而实现继承是我们要努力摒弃的，接口继承才是我们的努力方向，而做好接口继承，显然会更符合 LSP。

## 更广泛的 LSP

如果理解了 LSP，你会发现，它不仅适用于类级别的设计，还适用于更广泛的接口设计。比如，我们在开发中经常会遇到系统集成的问题，有不同的厂商都要通过 REST 接口把他们的统计信息上报到你的系统中，但是，有一个大厂上报的消息格式没法遵循你定义的格式，因为他的系统改动起来难度比较大。你该怎么办呢？

也许，专门为大厂设计一个特定接口是最简单的想法，但是，一旦开了这个口子，后面的各种集成接口都要为这个大厂开发一份特殊的，而且，如果未来再有其他大厂也提出要求，你要不要为它们也设计特殊接口呢？事实上，很多项目功能不多，但接口特别多，就是因为在这种决策的时候开了口子。**请记住，公开接口是最宝贵的资源，千万不能随意添加。**

如果我们用 LSP 的角度看这个问题，通用接口就是一个父类接口，而不同厂商的内容就相当于一个个子类。让厂商面对特定接口，系统将变得无法维护。后期随着人员变动，接口只会更加膨胀，到最后，没有人说清楚每个接口到底是做什么的。

好，那我们决定采用统一的接口，可是不同的消息格式该怎么处理呢？首先，我们需要区分出不同的厂商，办法有很多，无论是通过 REST 的路径，还是 HTTP 头的方式，我们可以得到一个标识符。然后呢？

很容易想到的做法就是写出一个 if 语句来，像下面这样：

```
1 if (identifier.equals("SUPER_VENDOR")) {  
2     ...  
3 }
```

[复制代码](#)

但是，千万要遏制自己写 if 的念头，一旦开了这个头，后续的代码也将变得难以维护。我们可以做的是，提供一个解析器的接口，根据标识符找到一个对应的解析器，像下面这样：

```
1 RequestParser parser = parsers.get(identifier);  
2 if (parser != null) {  
3     return parser.parse(request);  
4 }
```

[复制代码](#)

这样一来，即便有其他厂商再因为某些奇怪的原因要求有特定的格式，我们要做的只是提供一个新的接口实现。这样一来，所有代码的行为就保持了一致性，核心的代码结构也保持了稳定。

## 总结时刻

今天，我们讲了 Liskov 替换原则，其主要意思是说子类型必须能够替换其父类型。

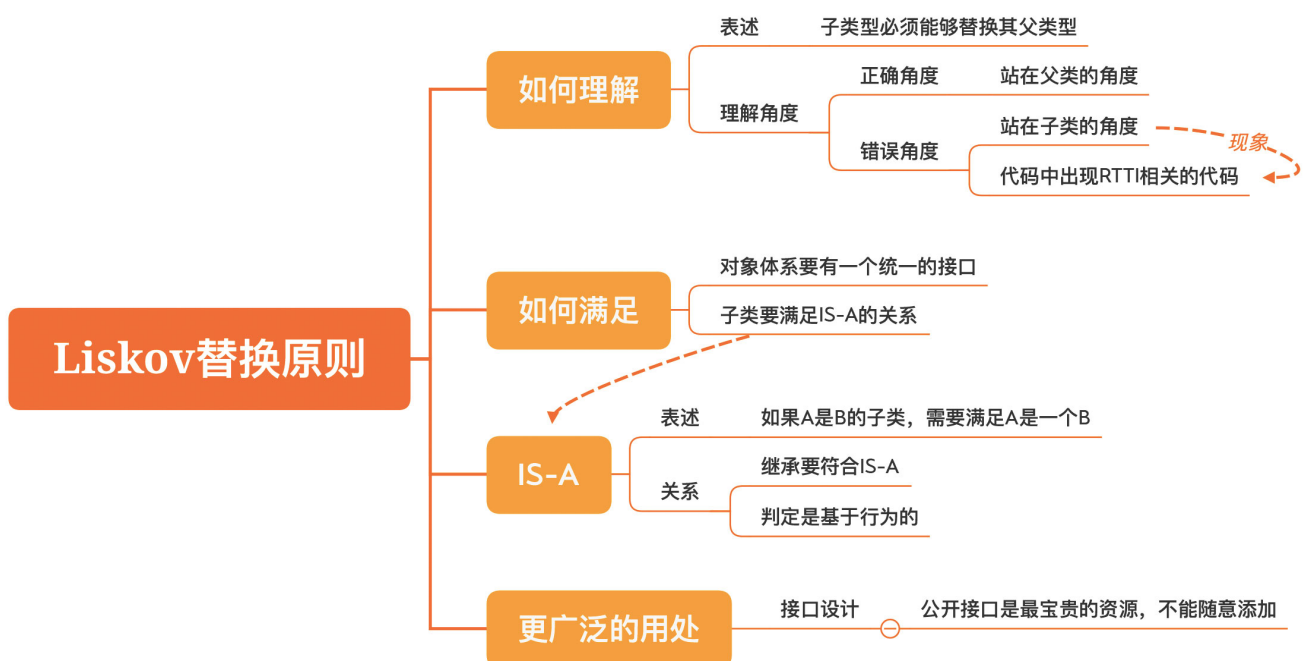
理解 LSP，我们需要站在父类的角度去看，而站在子类的角度，常常是破坏 LSP 的做法，一个值得警惕的现象是，代码中出现 RTTI 相关的代码。

继承需要满足 IS-A 的关系，但 IS-A 的关键在于行为上的一致性，而不能单纯凭日常的概念或直觉去理解。

LSP 不仅仅可以用在类关系的设计上，我们还可以把它用在更广泛的接口设计中。任何接口都是宝贵的，在设计时，都要精心考量。

这一讲，你可以看到 LSP 的根基在于继承，但显然接口继承才是重点。那我们该如何设计接口呢？我们下一讲来讨论。

如果今天的内容你只能记住一件事，那请记住：**用父类的角度去思考，设计行为一致的子类。**





## 思考题

在今天的內容中，我們提到了长方形正方形問題，我只分析了這個做法有問題的地方，現在我把解決這個問題的機會留給你，請你來動動腦，歡迎在留言區寫下你的解決方案。

感謝閱讀，如果你覺得這一講的內容對你有幫助的話，也歡迎把它分享給你的朋友。

提建議

更多課程推薦

# 設計模式之美

前 Google 工程師手把手教你寫高質量代碼

王爭

前 Google 工程師

《數據結構與算法之美》專欄作者



漲價倒計時 🕒

限時秒殺 **¥149**，7月31日漲價至 **¥299**

© 版權歸極客邦科技所有，未經許可不得傳播售賣。頁面已增加防盜追蹤，如有侵權極客邦將依法追究其法律責任。

上一篇 21 | 開放封閉原則：不改代碼怎麼寫新功能？

下一篇 23 | 接口隔離原則：接口里的方法，你都得到嗎？

## 精选留言 (10)

写留言



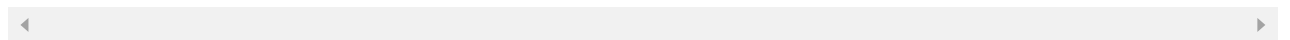
赵冲

2020-07-17

那从父类的角度来考虑的话，应该是定义一个几何图形的接口，接口有计算面积的方法。然后长方形、正方形、圆形、三角形.....都实现这个接口，然后各自实现计算面积的方法。各自有自己特别的关键属性，根据属性计算各自面积:长\*宽、边长<sup>2</sup>、 $\pi r^2$ 、(底长\*高)/2、.....

展开 ∨

作者回复: 嗯，这个解决方案的味道不错。



6



Demon.Lee

2020-07-21

思考题，想了好几天，总觉得没那么简单，定义一个Shape接口（里面一个计算面积的area方法）就完了吗？那老师例子里面的set方法改变属性值怎么破，我反复刷留言，还是没有我想要答案，只能Google了，然后我就释怀了🤔：<http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document>

展开 ∨



2



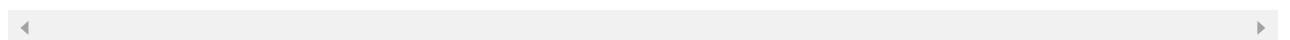
Being

2020-07-18

全篇一直在强调行为，我想这也是思考题的突破口。长宽是数据，而Rectangle并没有将行为抽象出来，导致Rectangle和Square不能成为IS-A的关系，我们只要把求面积的行为放在Rectangle下，子类分别去实现面积的方法就好了。

展开 ∨

作者回复: “把求面积的行为放在Rectangle下，子类分别去实现面积的方法”，可以解决这个问题吗？



1

2



桃子-夏勇杰

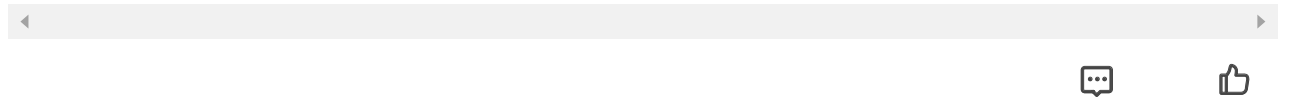
2020-07-29

这个设计原则看着非常简单，提出者居然能获得图灵奖，可见这个设计原则的价值非常

大。郑老师，这个设计原则的价值到底有多大呢？

展开 ∨

作者回复: LSP告诉我们什么样的继承是对的，而继承使用范围太广了。



**三生**

2020-07-22

所有的形状都有求面积的方式，但是计算方式都不同，这行为应该是“正常的”，但是设置长和宽的行为不正确，因为长方体有宽和高，正方形只有宽或高，这里只能抽象出计算面积这个方法。

比如企鹅和麻雀，我们认为所有的鸟都会飞，但企鹅不会飞，而他却具有了飞的行为， ...

展开 ∨



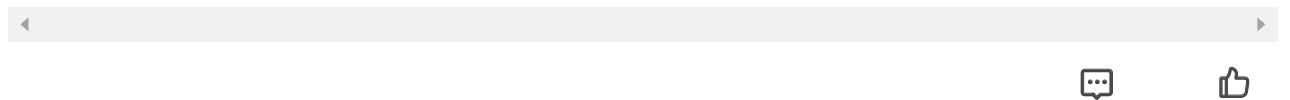
**张珈毓**

2020-07-20

在正方形类中重写长方形的计算面积接口

展开 ∨

作者回复: 但测试依然无法通过。

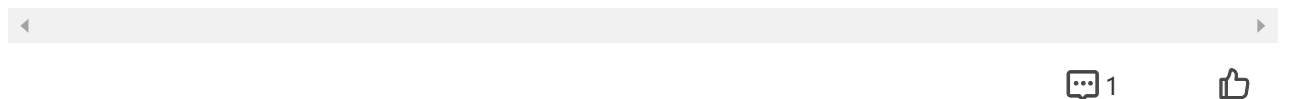


**monalisali**

2020-07-20

RequestParser 中还是免不了用多个 if 来判断 identifier，从而返回特定的子类吧

作者回复: 不一定，可以通过一个Map实现。



**zchq88**

2020-07-20

是不是可以吧Square做父类，然后Rectangle作为子类来设计，父类只有一个设置边长，子类增加设置长宽的接口，如果没有设置长宽默认使用边长的值。这样是不是符合替换原

则？



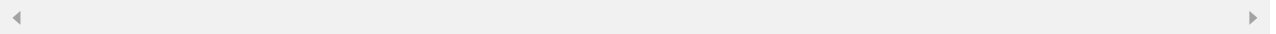
Jxin

2020-07-17

如果业务场景合适，约束功能也不失为一个解决办法。让宽高不可变，初始化时就必须赋值。这样就能符合现实中的特性。自然也没有长宽赋不同值的麻烦。

展开 ✓

作者回复: setter 确实是一个有杀伤力的东西，但回避 setter并不是在解决我们提出的问题。



阳仔

2020-07-17

Liskov替换的意思是子类型能够替换父类型，且在继承体系中保持接口的一致  
长方形与正方形计算面积的行为接口是一样的，但是定义长方形和正方形的接口是不一样的，所以这两个行为可以分别抽离出来

展开 ✓

作者回复: 长方形和正方形接口不一样，这是一个点。

