## 里式替换（LSP）跟多态有何区别？哪些代码违背了LSP？

在上两节课中，我们学习了 SOLID 原则中的单一职责原则和开闭原则，这两个原则都比较重要，想要灵活应用也比较难，需要你在实践中多加练习、多加体会。今天，我们再来学习 SOLID 中的“L”对应的原则：里式替换原则。

整体上来讲，这个设计原则是比较简单、容易理解和掌握的。今天我主要通过几个反例，带你看看，哪些代码是违反里式替换原则的？我们该如何将它们改造成满足里式替换原则？除此之外，这条原则从定义上看起来，跟我们之前讲过的“多态”有点类似。所以，我今天也会讲一下，它跟多态的区别。

### 如何理解“里式替换原则”？

里式替换原则的英文翻译是：Liskov Substitution Principle，缩写为 LSP。这个原则最早是在 1986 年由 Barbara Liskov 提出，他是这么描述这条原则的：

If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program。

在 1996 年，Robert Martin 在他的 SOLID 原则中，重新描述了这个原则，英文原话是这样的：

Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it。

我们综合两者的描述，将这条原则用中文描述出来，是这样的：子类对象（object of subtype/derived class）能够替换程序（program）中父类对象（object of base/parent class）出现的任何地方，并且保证原来程序的逻辑行为（behavior）不变及正确性不被破坏。

这么说还是比较抽象，我们通过一个例子来解释一下。如下代码中，父类 Transporter 使用 org.apache.http 库中的 HttpClient 类来传输网络数据。子类 SecurityTransporter 继承父类 Transporter，增加了额外的功能，支持传输 appId 和 appToken 安全认证信息。

public class Transporter {

private HttpClient httpClient;

public Transporter(HttpClient httpClient) {

this.httpClient = httpClient;

}

public Response sendRequest(Request request) {

// ...use httpClient to send request

}

}

public class SecurityTransporter extends Transporter {

private String appId;

private String appToken;

public SecurityTransporter(HttpClient httpClient, String appId, String appToken) {

super(httpClient);

this.appId = appId;

this.appToken = appToken;

}

@Override

public Response sendRequest(Request request) {

if (StringUtils.isNotBlank(appId) && StringUtils.isNotBlank(appToken)) {

request.addPayload("app-id", appId);

request.addPayload("app-token", appToken);

}

return super.sendRequest(request);

}

}

public class Demo {

public void demoFunction(Transporter transporter) {

Reuqest request = new Request();

//...省略设置request中数据值的代码...

Response response = transporter.sendRequest(request);

//...省略其他逻辑...

}

}

// 里式替换原则

Demo demo = new Demo();

demo.demofunction(new SecurityTransporter(/\*省略参数\*/););

在上面的代码中，子类 SecurityTransporter 的设计完全符合里式替换原则，可以替换父类出现的任何位置，并且原来代码的逻辑行为不变且正确性也没有被破坏。

不过，你可能会有这样的疑问，刚刚的代码设计不就是简单利用了面向对象的多态特性吗？多态和里式替换原则说的是不是一回事呢？从刚刚的例子和定义描述来看，里式替换原则跟多态看起来确实有点类似，但实际上它们完全是两回事。为什么这么说呢？

我们还是通过刚才这个例子来解释一下。不过，我们需要对 SecurityTransporter 类中 sendRequest() 函数稍加改造一下。改造前，如果 appId 或者 appToken 没有设置，我们就不做校验；改造后，如果 appId 或者 appToken 没有设置，则直接抛出 NoAuthorizationRuntimeException 未授权异常。改造前后的代码对比如下所示：

// 改造前：

public class SecurityTransporter extends Transporter {

//...省略其他代码..

@Override

public Response sendRequest(Request request) {

if (StringUtils.isNotBlank(appId) && StringUtils.isNotBlank(appToken)) {

request.addPayload("app-id", appId);

request.addPayload("app-token", appToken);

}

return super.sendRequest(request);

}

}

// 改造后：

public class SecurityTransporter extends Transporter {

//...省略其他代码..

@Override

public Response sendRequest(Request request) {

if (StringUtils.isBlank(appId) || StringUtils.isBlank(appToken)) {

throw new NoAuthorizationRuntimeException(...);

}

request.addPayload("app-id", appId);

request.addPayload("app-token", appToken);

return super.sendRequest(request);

}

}

在改造之后的代码中，如果传递进 demoFunction() 函数的是父类 Transporter 对象，那 demoFunction() 函数并不会有异常抛出，但如果传递给 demoFunction() 函数的是子类 SecurityTransporter 对象，那 demoFunction() 有可能会有异常抛出。尽管代码中抛出的是运行时异常（Runtime Exception），我们可以不在代码中显式地捕获处理，但子类替换父类传递进 demoFunction 函数之后，整个程序的逻辑行为有了改变。

虽然改造之后的代码仍然可以通过 Java 的多态语法，动态地用子类 SecurityTransporter 来替换父类 Transporter，也并不会导致程序编译或者运行报错。但是，从设计思路上来讲，SecurityTransporter 的设计是不符合里式替换原则的。

好了，我们稍微总结一下。虽然从定义描述和代码实现上来看，多态和里式替换有点类似，但它们关注的角度是不一样的。多态是面向对象编程的一大特性，也是面向对象编程语言的一种语法。它是一种代码实现的思路。而里式替换是一种设计原则，是用来指导继承关系中子类该如何设计的，子类的设计要保证在替换父类的时候，**不改变原有程序的逻辑以及不破坏原有程序的正确性**。

### 哪些代码明显违背了 LSP？

实际上，里式替换原则还有另外一个更加能落地、更有指导意义的描述，那就是“Design By Contract”，中文翻译就是“按照协议来设计”。

看起来比较抽象，我来进一步解读一下。子类在设计的时候，要遵守父类的行为约定（或者叫协议）。父类定义了函数的行为约定，那子类可以改变函数的内部实现逻辑，但不能改变函数原有的行为约定。这里的行为约定包括：函数声明要实现的功能；对输入、输出、异常的约定；甚至包括注释中所罗列的任何特殊说明。实际上，定义中父类和子类之间的关系，也可以替换成接口和实现类之间的关系。

为了更好地理解这句话，我举几个违反里式替换原则的例子来解释一下。

### 1、子类违背父类声明要实现的功能

父类中提供的 sortOrdersByAmount() 订单排序函数，是按照金额从小到大来给订单排序的，而子类重写这个 sortOrdersByAmount() 订单排序函数之后，是按照创建日期来给订单排序的。那子类的设计就违背里式替换原则。

### 2. 子类违背父类对输入、输出、异常的约定

在父类中，某个函数约定：运行出错的时候返回 null；获取数据为空的时候返回空集合（empty collection）。而子类重载函数之后，实现变了，运行出错返回异常（exception），获取不到数据返回 null。那子类的设计就违背里式替换原则。

在父类中，某个函数约定，输入数据可以是任意整数，但子类实现的时候，只允许输入数据是正整数，负数就抛出，也就是说，子类对输入的数据的校验比父类更加严格，那子类的设计就违背了里式替换原则。

在父类中，某个函数约定，只会抛出 ArgumentNullException 异常，那子类的设计实现中只允许抛出 ArgumentNullException 异常，任何其他异常的抛出，都会导致子类违背里式替换原则。

### 3. 子类违背父类注释中所罗列的任何特殊说明

父类中定义的 withdraw() 提现函数的注释是这么写的：“用户的提现金额不得超过账户余额……”，而子类重写 withdraw() 函数之后，针对 VIP 账号实现了透支提现的功能，也就是提现金额可以大于账户余额，那这个子类的设计也是不符合里式替换原则的。

以上便是三种典型的违背里式替换原则的情况。除此之外，判断子类的设计实现是否违背里式替换原则，还有一个小窍门，那就是拿父类的单元测试去验证子类的代码。如果某些单元测试运行失败，就有可能说明，子类的设计实现没有完全地遵守父类的约定，子类有可能违背了里式替换原则。

实际上，你有没有发现，里式替换这个原则是非常宽松的。一般情况下，我们写的代码都不怎么会违背它。所以，只要你能看懂我今天讲的这些，这个原则就不难掌握，也不难应用。

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。

里式替换原则是用来指导，继承关系中子类该如何设计的一个原则。理解里式替换原则，最核心的就是理解“design by contract，按照协议来设计”这几个字。父类定义了函数的“约定”（或者叫协议），那子类可以改变函数的内部实现逻辑，但不能改变函数原有的“约定”。这里的约定包括：函数声明要实现的功能；对输入、输出、异常的约定；甚至包括注释中所罗列的任何特殊说明。

理解这个原则，我们还要弄明白里式替换原则跟多态的区别。虽然从定义描述和代码实现上来看，多态和里式替换有点类似，但它们关注的角度是不一样的。多态是面向对象编程的一大特性，也是面向对象编程语言的一种语法。它是一种代码实现的思路。而里式替换是一种设计原则，用来指导继承关系中子类该如何设计，子类的设计要保证在替换父类的时候，不改变原有程序的逻辑及不破坏原有程序的正确性。

### 课堂讨论

把复杂的东西讲简单，把简单的东西讲深刻，都是比较难的事情。而里式替换原则存在的意义可以说不言自喻，非常简单明确，但是越是这种不言自喻的道理，越是难组织成文字或语言来描述，有点儿只可意会不可言传的意思，所以，今天的课堂讨论的话题是：请你有条理、有深度地讲一讲里式替换原则存在的意义。