35 | 编程范式游记（6）- 面向对象编程

2018-1-30 陈皓

前面我们谈了函数式编程，函数式编程总结起来就是把一些功能或逻辑代码通过函数拼装方式来组织的玩法。这其中涉及最多的是函数，也就是编程中的代码逻辑。但我们知道，代码中还是需要处理数据的，这些就是所谓的“状态”，函数式编程需要我们写出无状态的代码。

而这天下并不存在没有状态没有数据的代码，如果函数式编程不处理状态这些东西，那么，状态会放在什么地方呢？总是需要一个地方放这些数据的。

对于状态和数据的处理，我们有必要提一下“面向对象编程”（Object-oriented programming，OOP）这个编程范式了。我们知道，**面向对象的编程有三大特性：封装、继承和多态**。

面向对象编程是一种具有对象概念的程序编程范型，同时也是一种程序开发的抽象方针，它可能包含数据、属性、代码与方法。对象则指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的可重用性、灵活性和可扩展性，对象里的程序可以访问及修改对象相关联的数据。在面向对象编程里，计算机程序会被设计成彼此相关的对象。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对计算机下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。

现在，几乎所有的主流语言都支持面向对象，比如：Common Lisp、Python、C++、Objective-C、Smalltalk、Delphi、Java、Swift、C#、Perl、Ruby与PHP等。

说起面向对象，就不得不提由Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides合作出版的《设计模式：可复用面向对象软件的基础》（Design Patterns -

Elements of Reusable Object-Oriented Software）一书，在此书中共收录了23种设计模式。

这本书的23个经典的设计模式，基本上就是说了两个面向对象的核心理念：

"Program to an ‘interface’, not an ‘implementation’."

使用者不需要知道数据类型、结构、算法的细节。

使用者不需要知道实现细节，只需要知道提供的接口。

利于抽象、封装、动态绑定、多态。

符合面向对象的特质和理念。

"Favor ‘object composition’ over ‘class inheritance’."

继承需要给子类暴露一些父类的设计和实现细节。

父类实现的改变会造成子类也需要改变。

我们以为继承主要是为了代码重用，但实际上在子类中需要重新实现很多父类的方法。

继承更多的应该是为了多态。

示例一：拼装对象

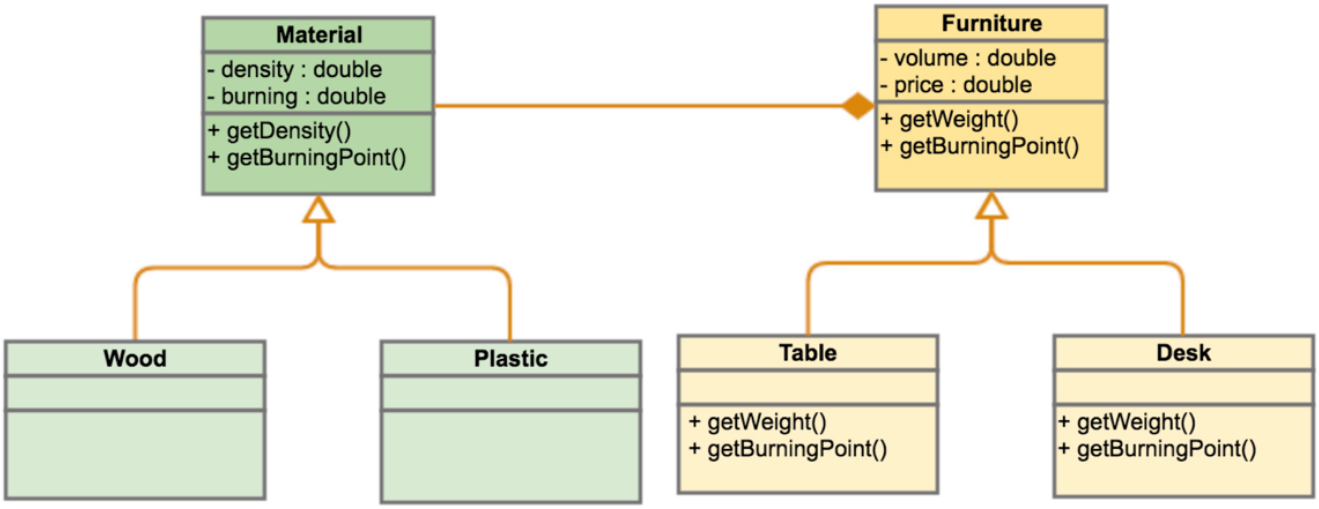
好，我们先来看一个示例，假设我们有如下的描述：

四个物体：木头桌子、木头椅子、塑料桌子、塑料椅子

四个属性：燃点、密度、价格、重量

那么，我们怎么用面向对象的方式来设计我们的类呢？

参看下图：



图的左边是“材质类” Material。其属性有燃点和密度。

图的右边是“家具类” Furniture。其属性有价格和体积。

在Furniture中耦合了Material。而具体的Material是Wood还是Plastic，这在构造对象的时候注入到Furniture里就好了。

这样，在家具类中，通过材料的密度属性和家具的体积属性就可以计算出重量属性。

这样设计的优点显而易见，它能和现实世界相对应起来，而且，材料类是可以重用的。这个模式也表现了面向对象的拼装数据的另一个精髓——喜欢组合，而不是继承。这个模式在设计模式里叫“桥接（Bridge）模式”。

和函数式编程来比较，函数式强调动词，而面向对象强调名词，面向对象更多的关注接口间的关系，而通过多态来适配不同的具体实现。

示例二：拼装功能

再来看一个示例。我们的需求是：处理电商系统中的订单，处理订单有一个关键的动作就是计算订单的价格。有的订单需要打折，有的则不打折。

在进行面向对象编程时，假设我们用Java语言，我们需要先写一个接口——`BillingStrategy`，其中一个方法就是`GetActPrice(double rawPrice)`，输入一个原始的价格，输出一个根据相应的策略计算出来的价格。

```
interface BillingStrategy {
    public double GetActPrice(double rawPrice);
}
```

这个接口很简单，只是对接口的抽象，而与实现无关。现在我们需要对这个接口进行实现。

```
// Normal billing strategy (unchanged price)
class NormalStrategy implements BillingStrategy {
    @Override
    public double GetActPrice(double rawPrice) {
        return rawPrice;
    }
}

// Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {
    @Override
    public double GetActPrice(double rawPrice) {
        return rawPrice * 0.5;
    }
}
```

上面的代码实现了两个策略，一个是不打折的：`NormalStrategy`，一个是打了5折的：`HappyHourStrategy`。

于是，我们先封装订单项 `OrderItem`，其包含了每个商品的原始价格和数量，以及计算价格的策略。

```
class OrderItem {
    public String Name;
    public double Price;
    public int Quantity;
    public BillingStrategy Strategy;
    public OrderItem(String name, double price, int quantity, BillingStrategy strategy) {
        this.Name = name;
        this.Price = price;
        this.Quantity = quantity;
        this.Strategy = strategy;
```

```
    }
}
```

然后，在我们的订单类——`Order` 中封装了 `OrderItem` 的列表，即商品列表。并在操作订单添加购买商品时，加入一个计算价格的 `BillingStrategy`。

```
class Order {
    private List<OrderItem> orderItems = new ArrayList<OrderItem>();
    private BillingStrategy strategy = new NormalStrategy();

    public void Add(String name, double price, int quantity, BillingStrategy strategy) {
        orderItems.add(new OrderItem(name, price, quantity, strategy));
    }

    // Payment of bill
    public void PayBill() {
        double sum = 0;
        for (OrderItem item : orderItems) {

            actPrice = item.Strategy.GetActPrice(item.price * item.quantity);
            sum += actPrice;

            System.out.println("%s -- %f(%d) - %f",
                        item.name, item.price, item.quantity, actPrice);
        }
        System.out.println("Total due: " + sum);
    }
}
```

最终，我们在 `PayBill()` 函数中，把整个订单的价格明细和总价打印出来。

在上面这个示例中，可以看到，我把定价策略和订单处理的流程分开了。这么做的好处是，我们可以随时给不同的商品注入不同的价格计算策略，这样一来就有很高的灵活度了。剩下的事就交给我们的运营人员来配置不同的商品使用什么样的价格计算策略了。

注意：现实社会中，订单价格计算会比这个事复杂得多，比如：有会员价，有打折卡，还有商品的打包价等，而且还可以叠加不同的策略（叠加策略用前面说的函数式的pipeline或decorator就可以实现）。我们这里只是为了说明面向对象编程范式，所以故意简单化了。

其实，这个设计模式叫——策略模式。我认为，这是设计模式中最为经典的模式了，其充分体现了面向对象编程的方式。

示例三：资源管理

先看一段代码：

```
mutex m;

void foo() {
    m.lock();
    Func();
    if ( ! everythingOk() ) return;
    ...
    ...
    m.unlock();
}
```

可以看到，上面这段代码是有问题的，原因是：那个 `if` 语句返回时没有把锁给unlock掉，这会导致锁没有被释放。如果我们要把代码写对，需要在return前unlock一下。

```
mutex m;

void foo() {
    m.lock();
    Func();
    if ( ! everythingOk() ) {
        m.unlock();
        return;
    }
    ...
    ...
    m.unlock();
}
```

但是，在所有的函数退出的地方都要加上 `m.unlock();` 语句，这会让我们很难维护代码。于是可以使用面向对象的编程模式，我们先设计一个代理类。

```
class lock_guard {
        private:
                mutex &_m;
        public:
                lock_guard(mutex &m):_m(m) { _m.lock(); }
                ~lock_guard() { _m.unlock(); }
};
```

然后，我们的代码就可以这样写了：

```
mutex m;

void foo() {
        lock_guard guard(m);
        Func();
        if ( ! everythingOk() ) {
                return;
        }
        ...
        ...
}
```

这个技术叫RAII（Resource Acquisition Is Initialization，资源获取就是初始化），是C++中的一个利用了面向对象的技术。这个设计模式叫“代理模式”。我们可以把一些控制资源分配和释放的逻辑交给这些代理类，然后，只需要关注业务逻辑代码了。而且，在我们的业务逻辑代码中，减少了这些和业务逻辑不相关的程序控制的代码。

从上面的代码中，我们可以看到下面几个面向对象的事情。

我们使用接口抽象了具体的实现类。

然后其它类耦合的是接口而不是实现类。这就是多态，其增加了程序的可扩展性。

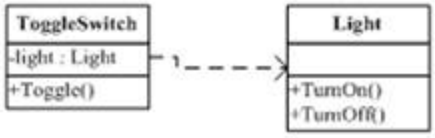
因为这就是接口编程，所谓接口也就是一种“协议”，就像HTTP协议一样。浏览器和后端的程序都依赖于这一种协议，而不是具体实现（如果是依赖具体实现，那么浏览器就要依赖后端的编程语言或中间件了，这就太恶心了）。于是，浏览器和后端的程序就完全解除依赖关系，而去依赖于一个标准的协议。

这就是面向对象的编程范式的精髓！同样也是IoC/DIP（控制反转/依赖倒置）的本质。

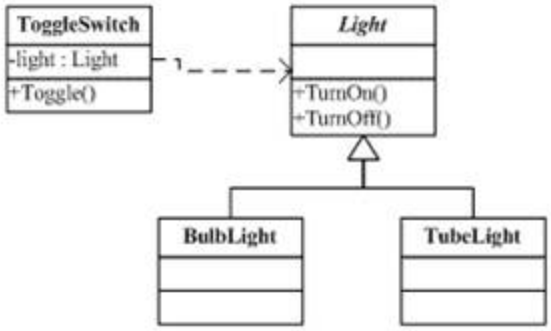
IoC 控制反转

关于IoC的的概念提出来已经很多年了，其被用于一种面向对象的设计。我在这里再简单地回顾一下这个概念。我先谈技术，再说管理。

话说，我们有一个开关要控制一个灯的开和关这两个动作，最常见也是最没有技术含量的实现会是这个样子：

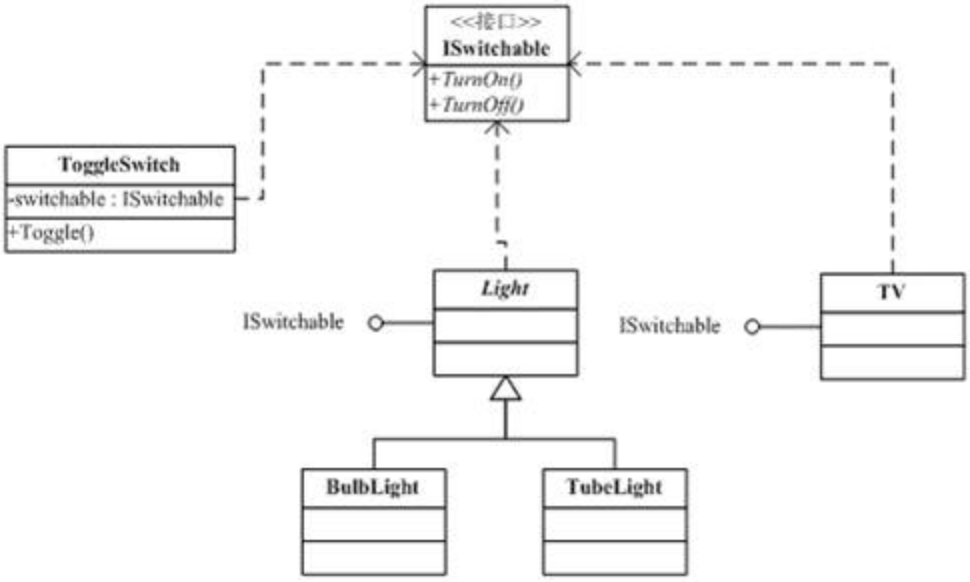


然后，有一天，我们发现需要对灯泡扩展一下，于是做了个抽象类：



但是，如果有一天，我们发现这个开关可能还要控制别的不单单是灯泡的东西，就会发现这个开关耦合了灯泡这种类别，非常不利于扩展，于是反转控制出现了。

就像现实世界一样，造开关的工厂根本不关心要控制的东西是什么，它只做一个开关应该做好的事，就是把电接通，把电断开（不管是手动的，还是声控的，还是光控，还是遥控的）。而我们造的各种各样的灯泡（不管是日光灯、白炽灯）的工厂也不关心你用什么样的开关，反正我只管把灯的电源接口给做出来。然后，开关厂和电灯厂依赖于一个标准的通电和断电的接口。于是产生了IoC控制反转，如下图：



所谓控制反转的意思是，开关从以前设备的专用开关，转变到了控制电源的开关，而以前的设备要反过来依赖于开关厂声明的电源连接接口。只要符合开关厂定义的电源连接的接口，这个开关可以控制所有符合这个电源连接接口的设备。也就是说，开关从依赖设备这种情况，变成了设备反过来依赖于开关所定义的接口。

这样的例子在生活中太多见了，比如说：

> 钱就是一个很好的例子。以前大家都是“以物易物”，所以，在各种物品之前都需要相应的“交易策略”，比如：一头羊换2袋米，一袋米换一斤猪后腿肉……这种换算太复杂了。于是，“钱”就出来了，所谓“钱”，其实就是一种交易协议，所有的商品都依赖这个协议，而不用再互相依赖了。于是整个世界的运作就简单了很多。
>
> 在交易的过程中，卖家向买家卖东西，一手交钱一手交货，所以，基本上来说卖家和买家必需强耦合（必需见面）。这个时候，银行出来做担保，买家把钱先垫到银行，银行让卖家发货，买家验货后，银行再把钱打给卖家。这就是反转控制。买卖双方把对对方的直接依赖和控制，反转到了让对方来依赖一个标准的交易模型的接口。股票交易也是一样的，证交所就是买卖双方的标准交易模型接口。
>
> 上面这个例子，可能还不明显，再举一个例子。海尔公司作为一个电器制商需要把自己的商品分销到全国各地，但是发现，不同的分销渠道有不同的玩法，于是派出了各种销售代表玩不同的玩法。随着渠道越来越多，发现，每增加一个渠道就要新增一批人和一个新的流程，严重耦合并依赖各渠道商的玩法。

实在受不了了，于是制定业务标准，开发分销信息化系统，只有符合这个标准的渠道商才能成为海尔的分销商，让各个渠道商反过来依赖自己标准。反转了控制，倒置了依赖。

这个思维方式其实还深远地影响了很多东西，比如我们的系统架构。

> 云计算平台中有很多的云产品线。一些底层服务的开发团队只管开发底层的技术，然后什么也不管了，就交给上层的开发人员。上层开发人员在底层团队开发出来的产品上面开发各种管理这个底层资源的东西，比如：生产底层资源的业务，底层资源的控制台，底层资源的监控系统。

然而，随着接入的资源越来越多，上层为各个云资源控制生产，开发控制台和监控的团队，完全干不过来了。这个时候依赖倒置和反转控制又可以解决问题了。为了有统一体验，各个云产品线需要遵从一定的协议或规范来开发。比如，每个云产品团队需要按照标准定义相关资源的生命周期管理，提供控制台，接入整体监控系统，通过标准的协议开发控制系统。

> 集中式处理电子商务订单的流程。各个垂直业务线都需要通过这个平台来处理自己的交易业务，但是垂直业务线上的个性化需求太多。于是，这个技术平台开始发现，对来自各个业务方的需求应接不暇，各种变态需求严重干扰系统，各种技术决策越来越不好做，导致需求排期排不过来。

这个时候，也可以使用依赖倒置和反转控制的思想来解决问题：开发一个插件模型、工作流引擎和Pub/Sub系统，让业务方的个性化需求支持以插件的方式插入订单流程中。业务方自己的数据存在自己的库中，业务逻辑也不要侵入系统，并可以使用工作流引擎或Pub/Sub的协议标准来自己定义工作流的各个步骤（甚至把工作流引擎的各个步骤的decider交给各个业务方自行处理）。

让各个业务方来依赖于标准插件和工作流接口，反转控制，让它们来控制系统，依赖倒置，让它们来依赖标准。

上面这些我想说什么？我想说的是：

> 我们每天都在标准化和定制化中纠结。我们痛苦于哪些应该是平台要做的，哪些应该要甩出去的。
>
> 这里面会出现大量的与业务无关的软件或中间件，包括协议、数据、接口……
>
> 通过面向对象的这些方式，我们可以通过抽象来解耦，通过中间件来解耦，这样可以降低软件的复杂度。

总而言之，我们就是想通过一种标准来让业务更为规范。

小结

不过，我们也需要知道面向对象的优缺点。

优点

能和真实的世界交相辉映，符合人的直觉。

面向对象和数据库模型设计类型，更多地关注对象间的模型设计。

强调于“名词”而不是“动词”，更多地关注对象和对象间的接口。

根据业务的特征形成一个个高内聚的对象，有效地分离了抽象和具体实现，增强了可重用性和可扩展性。

拥有大量非常优秀的设计原则和设计模式。

S.O.L.I.D（单一功能、开闭原则、里氏替换、接口隔离以及依赖反转，是面向对象设计的五个基本原则）、IoC/DIP……

缺点

代码都需要附着在一个类上，从一侧面上说，其鼓励了类型。

代码需要通过对象来达到抽象的效果，导致了相当厚重的“代码粘合层”。

因为太多的封装以及对状态的鼓励，导致了大量不透明并在并发下出现很多问题。

还是好多人并不是喜欢面向对象，尤其是喜欢函数式和泛型那些人，似乎都是非常讨厌面向对象的。

通过对象来达到抽象结果，把代码分散在不同的类里面，然后，要让它们执行起来，就需要把这些类粘合起来。所以，它另外一方面鼓励相当厚重的代码黏合层（代码黏合层就是把代码黏合到这里面）。

在Java里有很多注入方式，像Spring那些注入，鼓励黏合，导致了大量的封装，完全不知道里面在干什么事情。而且封装屏蔽了细节，具体发生啥事你还不知道。这些都是面向对象不太好的地方。

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。**这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。**

精选留言 13

光明
1517272384

文章太好了，赞👍

连子
1523715032

领域建模啥时候讲讲？

---

fsj
1522942631

示例三中的代码感觉使用了RAII技术也很丑陋；控制反转挺难理解的，但是我觉得更难的是识别出哪里需要控制反转

作者回复 求不丑陋的例子

瀚海星尘
1563185073

IoC/DIP的思想真是酷啊！！！

拉欧
1558941111

面向对象编程，一是抽象思维，把数据和算法抽象成类和对象；二是标准化，接口即协议，所有的实现类都要满足定义的接口才可以作为依赖；三是封装，对象之间通过接口调用，互相之间不关心彼此的细节；总之，接口的定义是否清晰是面向对象设计的关键；
由于对象内封装了状态，所以在并发环境下天然存在问题

七月有风
1558315759

老师，您好，RAII 那段代码在C++中可以那样去实现，在JavaScript中有没有替代方案，最近就有这个问题，一直没找到很好的解决方案。请老师回复。

Join
1519825135

刚好今天代码里用到了IOC/DI,这时候看下文章太有感觉了，有补充了新的知识，谢谢皓子叔

Freezer
1517368279

耗子神，更新的太慢了，求加餐啊 

RZ_diversity
1517345473

看过后，对本科学的C + + 有了新的认识，需要反复领会其中的知识点

edisonhuang
1561336867

面向对象的编程有三大特性：封装、继承和多态
相比于函数式编程更强调动作，面向对象则更强调名词。它更符合我们人类的思考模式，将数据封装在对象之中
面向对象强调两点，一是面向接口而非面向实现编程。二是偏向使用组合而非继承。
通过面向对象的方法，实现控制反转个依赖导致。对于底层的服务更多的是实现标准的协议，同时将协议开放出来提供给其他系统订阅 。而基于协议需要做怎样的具体操作，则交给其他系统以及来决策

Valen
1559214851

@xilie 关于会员等级与多个业务特权对应的问题, 想了下可以用控制反转的方法. 就是抽出一个 等级<->特权 的标准出来, 向外提供 类似 `getPrivileges(level)`, `havePrivilege(privilege, level)` 的接口, 然后会员等级系统和业务系统都依赖于这个标准. 有变动的话就直接改这个标准就行

xilie
1556090991

有个面向对象的设计问题求解：

背景：1、有会员等级系统，等级级别分类需要由运营情况来定可伸缩，并不固定；2、好几个业务系统，跟进会员等级，给予不同的会员权利；

问题：如果会员等级系统只提供接口，业务系统使用，这样虽然会员等级系统很干净，可是

一旦会员等级系统的级别分类变化，各业务系统得对应开发。而如果换个方式，业务系统注入不同等级的会员权利，会员等级系统就很不干净，而且各业务系统注入的会员等级可能不一致，造成混乱。

不像开关，只有开和关，固定不变，会员等级系统内的等级级别会变化，怎么解决呢？

宋桓公

1534509466

我觉得面向对象最有用的是多态