浅谈面向对象编程 – 文章

1. OOP简介

面向对象编程（object-oriented programming）以下统一简称为OOP。世界上第一个OOP语言叫Simula，诞生于20世纪60年代，是它引入了对象、类、继承、虚过程等等这些概念。当时还没有“object-oriented”这个术语，这个术语是由第二个OOP语言Smalltalk的发明者Alan Kay提出来的，Smalltalk是“纯OO”的语言，在Smalltalk中一切皆对象：class、primitive type、code block（相当于匿名函数）等全是对象，对象行为的执行是通过向对象发送消息实现的，它没有命令式编程（imperative programming）中if、while这种语法结构，这些控制结构是通过向Boolean类型对象传递带有code block的消息实现的，Smalltalk是OOP语言中的代表，它影响了许多后来的OO语言，像Objective-C、Ruby、Java。 OOP作为一种思想，并不是由某个人发明出来的，各路OO大师都有自己的观点，所以对于到底什么是OOP，并没有一致的、权威的定义。本文所表达的OOP来自 Smalltalk + 自己肤浅的理解。

2. 对象的组成

2.1 协议 和 实现

- 对象（object） 表示一个由 状态（私有的） 和 操作（公开的）组成的单元
- 消息（message） 表示发送给一个对象让它执行某个操作的请求。一个对象能够响应的消息的集合叫做它的接口（interface）或协议（protocol），外界与对象进行交互应当只能通过这个对象的接口

> 消息代表一个对象能够响应什么操作，操作具体如何执行则是由方法（method）表示的。对象收到消息后决定调用哪个方法来进行处理，方法属于内部实现，也应是私有的。

Java

```
Object i = 1 + 2;     // 向1这个对象发送“加”消息，消息参数是2
String str = i.toString();     // 向i这个对象发送toString消息，此时并不知道什么方法会被调用
```

Smalltalk中，类通过定义protocol description来表示本类的实例可以响应哪些消息，方法则是单独定义，而在C++、Java这些语言中，没有这个区分，对于C++可以把protocol description理解成头文件里的函数声明，把方法理解成源文件中的函数定义。
对于C#/Java，可以把protocol description理解成接口中的方法列表，方法理解成实现类中的方法。

2.2 状态及处理过程的隐藏

> 对象的状态是私有的，只能由方法操作，方法是行为的具体实现，也是私有的，方法的调用是对象收

> 到消息后由该对象自己进行的，这样对象的状态处理细节是完全隐藏的，这种特征就是“封装”。

汽车 与 封装
驾驶手动档汽车时不用直接去操作它引擎、变速齿轮，而是通过 变速杆、离合/制动/加速 踏板 这些接口，如果你不了解汽车的话，应该不知道变速杆和离合器是干什么用的，这其实是因为手动档汽车只是对引擎做了很浅的一层封装，某些接口其实暴露了其内部的实现 -> 汽油机，以至于在与汽车这个对象交互时需要注意一些规则比如松离合器要慢、换档前要踩离合器等等，以保证这个对象能正常工作，这就增加了使用者和这个对象间的耦合度，假设汽油机做了一些改进或者说引擎换成了电动机，那驾驶人的操作习惯就要作一些调整。

自动档汽车就封装得更好，只保留了 制动/加速 踏板，变速杆也被封装成了几个抽象档位，内部细节被隐藏了，对外耦合就小了。

2.2.1 隐藏的实现

JavaScript中可以通过闭包；Ruby中实例变量本身就是隐藏的，外部无法访问；C++/C#/Java可通过private关键字；C虽然语法上不支持，但程序员可以通过命名约定实现。

3. OOP的解耦利器 - 多态（subtype polymorphism）

多态从字面上讲是指“不同的对象以不同的方法响应相同的消息”，与过程式编程（procedural programming）不同，在OOP中对象是基本单元，函数存在于对象中，外部需要某个操作时向对象发送消息，对象来决定调用哪个函数，这样就将行为的实现者和行为的请求者解耦了。

举个例子，汽车、飞机、轮船这些交通工具，虽然它们的动力原理、操作方式都不一样，但它们都具有一些相同的接口：加速、获取速度，现要实现一个测速操作，可以测试任何交通工具。

Java示例：

```
// 交通工具
interface IVehicle{
    // 加速
    public void speedUp();
    // 获取当前速度
    public int getCurrentSpeed();
}// 汽车
class Automobile implements IVehicle{
    private int currentSpeed = 0;
    public void speedUp(){
        System.out.println("汽车正在加速。。。");
        ++this.currentSpeed;
    }
    public int getCurrentSpeed(){
        return this.currentSpeed;
```

```
    }
}// 飞机
class Aeroplane implements IVehicle{
    private int currentSpeed = 0;
    public void speedUp(){
        System.out.println("飞机正在加速。。。");
        this.currentSpeed += 3;
    }
    public int getCurrentSpeed(){
        return this.currentSpeed;
    }
}
public class Main{
    public static void main(String[] args){
        accelerationTest(new Automobile(), 5);
        accelerationTest(new Aeroplane(), 5);
    }
    /**
     * 测试某个交通工具的加速度     *     * @param vehicle     交通工具     * @param
duration    加速多长时间     */
    public static void accelerationTest(IVehicle vehicle, int duration){
        for(int start = 0, end = start + duration; start speedUp();
        }
        System.out.printf("%s在%d内将速度提升到%dn", vehicle.getClass().getName(),
duration, vehicle.getCurrentSpeed());
    }
}
```

在上面代码中，accelerationTest虽然是交通工具的使用者，但却完全不受具体交通工具的影响，如果新添加一个Ship（船），accelerationTest一点都不用修改，因为accelerationTest和具体的交通工具都遵循了IVehicle这个协议，这样accelerationTest就知道：不管你具体是什么交通工具，反正都能够响应协议里的消息，到底调用什么方法来响应这些消息则交给了协议的实现者（即具体的交通工具）（这个从实现角度说的话应该是交给了编译器/解释器），而不是让accelerationTest根据具体的交通工具自己选择调用哪个函数（过程式的思维）。

3.1 多态的实现

- duck-typing（鸭子类型）。因为动态语言中没有静态类型检查，所以能够做到“只要会呱呱叫的，就可以算是鸭子”，比如用JavaScript代码继续上面的示例： Java
  ```
  var duck = new Duck();    // 创建一个鸭子对象
  duck.speedUp = function(){ /*略*/ };
  duck.getCurrentSpeed = function(){ /*略*/ };
  // 这样就可以把一个鸭子对象当作“交通工具”测试其加速度
  ```

accelerationTest(duck, 5);

- dynamic-dispatch（动态分派）
  dynamic-dispatch是指在运行时去确定真正调用哪个函数，比如C++中的虚函数

即使是像C这种过程式的语言，也可利用函数指针实现多态：

Java

```
#include
#include
#include
struct vehicle;
/** * 交通工具虚函数表 */
struct vehicle_vtable{
    void (*speed_up)(struct vehicle *self);
    int (*get_current_speed)(struct vehicle *self);
};
/** * 交通工具 */
struct vehicle{
    int current_speed_;
    char name[100];
    struct vehicle_vtable methods;    // 虚函数表
};
// 汽车加速方法
void automobile_speed_up(struct vehicle *self){
    puts("汽车正在加速。。。");
    ++self->current_speed_;
}
int automobile_get_current_speed(struct vehicle *self){
    return self->current_speed_;
}// 汽车构造函数
struct vehicle *automobile_new(){
    struct vehicle *self = (struct vehicle *)calloc(sizeof(struct vehicle));
    self->methods.speed_up = automobile_speed_up;
    self->methods.get_current_speed = automobile_get_current_speed;
    strcpy(self->name, "汽车");
    return self;
}
void automobile_destory(struct vehicle *self){
    free(self);
}// 飞机加速方法
void aeroplane_speed_up(struct vehicle *self){
    puts("飞机正在加速。。。");
    self->current_speed_ += 3;
```

```
}
int aeroplane_get_current_speed(struct vehicle *self){
    return self->current_speed_;
}// 飞机构造函数
struct vehicle *aeroplane_new(){
    struct vehicle *self = (struct vehicle *)calloc(sizeof(struct vehicle));
    self->methods.speed_up = aeroplane_speed_up;
    self->methods.get_current_speed = aeroplane_get_current_speed;
    strcpy(self->name, "飞机");
    return self;
}
void aeroplane_destory(struct vehicle *self){
    free(self);
}
void acceleration_test(struct vehicle *vehicle, int duration){
    for(int i = 0; i methods.speed_up(vehicle);
    printf("%s在%d内将速度提升到%dn", vehicle->name, duration, vehicle-
>methods.get_current_speed(vehicle));
}
int main(){
    struct vehicle *automobile = automobile_new();
    struct vehicle *aeroplane = aeroplane_new();
    acceleration_test(automobile, 5);
    acceleration_test(aeroplane, 5);
    automobile_destory(automobile);
    aeroplane_destory(aeroplane);
    return 0;
}
```

4. 关于继承

个人认为OOP只有封装和多态，继承不属OOP的特性，它只是某些编程语言用来实现subtyping、多态和代码复用的一种手段。（subtyping和subclassing不是一回事，subtyping表达的是“可替换性”，subclassing就是指继承，某些语言只能用subclassing实现subtyping）

如果你看过一些OO的书，你会发现都会提到一个原则叫“组合优于继承”，组合表示的是一种包含关系，而继承是一种层级关系，为什么要用组合代替继承呢？

其实呢，继承也是包含关系，子类继承父类，也就包含了父类的状态，只不过这种包含关系是编译器帮你做了，还包含了方法的实现，所以能够响应父类能响应的消息，就实现了多态。那这一举两得，不是挺好吗？是挺好的，但不用继承照样能实现以上功能，而且继承有如下缺点：

- 大多数的语言都是单根继承，也就是只能继承一个父类，这样你就失去了一次使用继承的机会

- 继承的耦合度太高，子类和父类的关系在编译时被固定死了，不能动态切换，而组合关系可以在运行随意切换，一个例子就是装饰者模式 ，装饰者模式和继承类似实现的是一种“扩展”的目的，在装饰者模式中被扩展的对象是可选择的，而继承实现的这种扩展则是“死”的

> 什么时候用继承？

如果你想实现多态，正好又想复用实现，那么可以用继承。在使用继承前，建议想一下，使用它的目的是什么，如果仅仅为了实现多态就用接口，如果仅仅为了复用实现就用组合，继承虽然方便，但太重量级，又有限制，少用为妙。

4.1 为什么“正方形不是长方形”？

很多资料上都说继承表达的是一种“xx is a yy”的关系，我觉得这种说法不够准确，因为继承是用来实现多态的，是针对行为而言的，所以表达的是“xx 能够响应 yy 所能响应的消息”。

“正方形 继承 长方形”是一个经典的违背“里氏替换原则”的例子，常识上我们一般都认为“正方形是一种特殊的长方形”，貌似满足is-a的关系，可以用继承，但如果从接口考虑就不一样了：长方形能响应“操作长”、“操作宽”这两个消息，而正方形只能响应“操作边长”这一个消息，所以在OOP中我们要从接口上去考虑而不是简单的去判断是否具有“xx是一种特殊的yy”的关系。

5. 总结

OOP是用来在对象间统一协议，让对象间的交互只关心协议，而不关心实现，所以OOP只有封装和多态，至于其它的比如继承，只是语言层面提供的实用特性。

> 打赏支持作者写出更多好文章，谢谢！

前端 - JavaScript, HTML5, CSS

Java - 专注Java技术分享

Python - 专注Python技术分享