vJass 系列教程 8

面向对象编程（三）接口（interface）、继承与多态

Aeris ▸ NJU ▸ 2009/2/3

vJass 系列教程 8

面向对象编程（三）接口（interface）、继承与多态

内容提要

本节我们终于要接触到面向对象最强大，也是最复杂的特性——继承和多态了。继承和多态是面向对象基本特性之一， vJass 因为受到种种制约，未能实现完全的面向对象特性，但是它实现的部分对于一般地图已经足够了。

如果读者之前没有接触过相关内容，那么这一章的内容会相对比较艰深。如果之前学习过一些面向对象的语言（C++、Java 等），会比较容易。

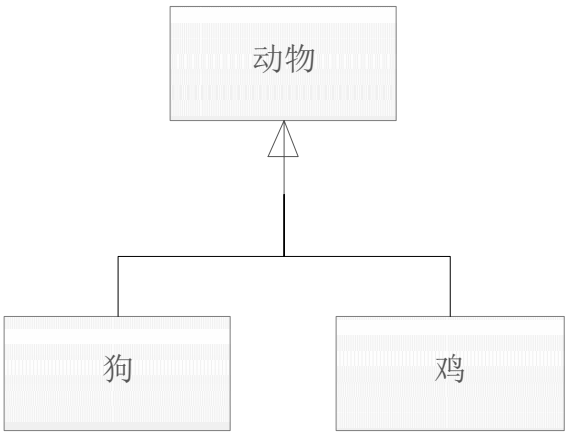
继承和多态概述

继承和多态到底是描述什么的？我们从一些简单的例子开始。

继承

我们知道，一只狗**是一个**动物，一只鸡也**是一个**动物，这是因为它们都具有动物的共同特征。在魔兽里，一个人族的步兵**是一个**单位，一个兽族的狼骑也**是一个**单位。这也是因为他们都具有单位的共同特征。那么，“狗”和“鸡”这两个概念与“动物”这个概念是什么关系？“步兵”和“狼骑”这两个概念和“单位”这个概念是什么关系？用普通的语言来描述，就是“‘狗’和‘鸡’各是一种特殊的‘动物’”，“‘步兵’和‘狼骑’各是一种特殊的‘单位’”。用面向对象的术语表述，它们之间是**泛化**（Generalization）关系（或者叫**一般化**关系，从“狗”和“鸡”到“动物”，从“步兵”和“狼骑”到“单位”），用图示[1]可以表示如下：

[1] 如果读者懂得软件工程的相关知识，会知道这种图叫 UML 图。



从“狗”到“动物”的箭头表示：“狗”和“动物”之间是**泛化**关系。就是说，“狗”是一种特殊的“动物”。“鸡”和“动物”之间的关系也是一样。

前面提到，之所以说“狗”是一种特殊的“动物”，是因为**狗具有动物应该具有的所有公共特征**。（当然，狗除了具有动物的公共特征之外，还有自己的独特特征[比如叫声是“汪汪”]）因此我们可以说，“狗”这个概念**继承**了“动物”这个概念。**“继承”是“泛化”的另一种表述方式**。

“狗”继承了“动物”，那么“动物”就叫做**基类**或者**父类**，“狗”就叫**派生类**或者**子类**。同样的，可以说“单位”是父类或者基类，“步兵”是子类或者派生类。

接口（Interface）

如果我们考察“会叫的动物”，而狗继承了“会叫的动物”，狗就也一定能叫。如果魔兽的单位有生命值，那么步兵继承了单位，步兵就也有生命值。从这些事实我们可以看出，基类或者父类规定了子类或者派生类“应该具有什么功能”。换句话说，基类或者父类规定了子类或者派生类所必须具有的“**接口（Interface）**”。“接口”的含义是，这个对象和外界打交道，只能依靠这个“窗口”所规定的功能来进行。

通常，基类都不是具体概念（“动物”和“单位”），因此又叫“**抽象基类**”。vJass 里，抽象基类就是**接口**。派生类会继承这个接口，由于接口中只是定义抽象的方法（功能），需要在派生类中具体实现，因此继承一个接口又叫**实现**一个接口。

多态

前面说过，“动物”这个接口规定了“叫”这个函数，那么“狗”就必须实现这个函数（汪汪汪……），同样，“鸡”也必须实现这个函数（喔喔喔……），它们都是动物，都会叫，但是具体叫法却不同，这就是**多态**行为。

多态的妙处是，有了多态的帮助，我们可以写一个函数，它负责处理一般的“动物”。如果我们想让它叫，那么直接调用“动物”的“叫”函数，而不必考虑这个动物具体是狗还是鸡，抑或是其他动物。它都会得到正确的结果。

三言两语总结：如果“狗”是一种“动物”，那么说“狗”**继承**了“动物”。如果动物规定了必须能叫，那么这个规定就是动物的**接口**。狗和鸡都要继承（实现）这个接口，但是它们的叫法不同，同是动物，却有不同的叫法，称为“**多态**”。

接口和继承的语法

定义接口和继承

我们来看看上面的概念在 vJass 里具体是怎么做的。仍然以动物为例子。

我们知道，动物这个抽象概念规定了所有动物必须具有哪些特征，因此它是一个抽象基类或者接口。定义接口的语法如下：

```
interface 接口名
    数据定义

    成员函数声明
endinterface
```

注意，成员函数只是“声明”而已，不写内容和 endmethod 关键字。

例如，我们规定，动物必须会叫，因此定义动物接口如下：

```
interface Animal
    method cry takes nothing returns nothing
endinterface
```

这个 cry 方法没有实现，就一句声明而已。因为它是一个抽象概念。

继承的语法如下：

```
struct 派生结构名 extends 接口名（一个结构只能继承一个接口）
    实现（重写，Override）接口规定的所有方法

    其他方法和数据定义
endstruct
```

例如，狗这个结构的定义如下：

```
struct Dog extends Animal
    // 必须有cry方法，而且参数和返回值必须和接口中定义的一致
    method cry takes nothing returns nothing
        call BJDebugMsg("汪汪汪……")
    endmethod
endstruct
```

鸡定义如下：

```
struct Chicken extends Animal
    method cry takes nothing returns nothing
        call BJDebugMsg("喔喔喔……")
    endmethod
endstruct
```

多级继承

一个继承了接口的结构还可以再被其他结构所继承，而接口里规定的方法也可以被重写，当然，也可以选择不重写，直接继承父类的。例如：

```
// 母鸡
struct Hen extends Chicken
    // 可以对Animal中定义的cry方法再次重写
    method cry takes nothing returns nothing
        call BJDebugMsg("咯咯咯……")
    endmethod
endstruct

// 公鸡
struct Rooster extends Chicken
    // 也可以不重写cry方法，直接继承Chicken类里面的
endstruct
```

虽然在公鸡这个结构里我们没有写任何方法，但是因为公鸡继承了“鸡”这个结构，所以公鸡也是有 cry 方法的，运行公鸡的 cry 方法，其行为和“鸡”这个结构的 cry 方法是一样的。

子结构也可以添加新的方法，不过不能重写父类中定义的、但是不是从接口里继承的方法。假如，母鸡这个结构中定义了“下蛋”这个方法，“下蛋”这个方法并没有在接口里定义，所以如果再写一个结构继承母鸡这个结构，结构中就不能定义“下蛋”这个方法了。

构造和析构

实现接口的结构可以正常定义构造函数和析构函数，和一般结构没有区别。

销毁一个继承了接口的结构实例，不必对这个实例本身操作，直接对接口操作就可以了。这意思是说，可以这样销毁一个实例：

```
local Animal a = Dog.create()
call a.destroy()
```

a 是 Animal 类型的变量，直接销毁 a 就可以了，会正确调用 Dog 或者 Chicken 的析构函数。这是因为，**析构函数也是多态的**。

对于多级继承的情况，则略微复杂一点。在这种情况下，子类是“包含”父类的，子类的数据分为两块，一块是从父类继承来的，另一块才是自己的。初始化子类**之前**，vJass 编译器会自动去初始化继承来的父类部分；而销毁子类**之后**，编译器同样会自动去销毁父类部分。

多态实例

前面我们定义了接口和两个子类，那么多态体现在哪里呢？请看下面这段代码：

```
function MakeItCry takes Animal a returns nothing
    // 调用Animal的cry方法
    call a.cry()
```

```
endfunction
```

如果这段代码执行，会打印出什么信息来呢？汪汪还是喔喔？

答案是：谁都不知道会打印出什么，只有等到运行时传过来一个具体参数才知道这次能打印出什么。

例如这样：

```
local Animal a = Dog.create()
call MakeItCry(a)
```

会打印出汪汪。

```
local Animal a = Chicken.create()
call MakeItCry(a)
```

会打印出喔喔。

每次都是调用的 Animal 的 cry 方法，却有不同的行为，这就是多态。

多态的应用

从上面例子可以看出，多态的核心作用在于，使用一个基类对象的引用（一个基类类型的变量）可以操作任意的派生类对象。它有以下几种常见用途。

实现回调（Callback）

什么是回调？大家对这种代码都不陌生：

```
function GroupAction takes nothing returns nothing
    // ....
endfunction

call ForGroup(g, function GroupAction)
```

这就是回调的例子。简而言之，回调就是你调用一个函数，那个函数又回过头来调用你自己提供的一个函数参数。

ForGroup 函数的定义如下：

```
native ForGroup takes group whichGroup, code callback returns nothing
```

第 2 个参数就是回调参数。

如何用多态来实现回调？请看下面的例子：

```
interface Callback
    method onEvent takes nothing returns nothing
endinterface

function SomeFunction takes Callback cb returns nothing
    call cb.onEvent()
    call cb.destroy()
endfunction
```

当调用 SomeFunction 这个函数的时候，必须传一个 Callback 对象给它（当然要自己写结构来实现这个接口并且实例化），然后这个函数内部就会自动去回调你自己定义的结构中的 onEvent 函数了。代码如下：

```
struct MyCallback extends Callback
    method onEvent takes nothing returns nothing
        call BJDebugMsg("MyCallback called")
    endmethod
endstruct

call SomeFunction(MyCallback.create())
```

注意：多态不是实现回调的唯一办法，还有更优雅，更实用的办法在后面的章节中会讲述。

实现通用事件系统

如果我想实现一个通用的光环系统（或者法球系统等，法球系统我自己已经实现了一个），经过观察，发现不同光环的代码大部分都相似，仅仅是核心代码有一些不同。借助于多态的启示，我们可以利用这种机制来封装那些不同的代码（和回调的例子类似，这些不同的核心代码就是回调函数）。具体例子就不细说了。

当然，多态并不是实现通用事件系统的唯一方法，后面有更好的。

扩展一个结构

这是 vJass 的另一个语法，它和多态是**没有关系**的。但是它和继承/多态一样，也使用同一个关键字 extends，读者应该特别注意和继承的区别。

所谓扩展，是指根据现有结构创建一个新的结构，新结构**自动具有**原有结构的所有数据和函数，然后，你可以给新结构添加新的函数，以达到代码复用的目的。语法如下：

```
struct 新结构名 extends 基础结构名
    // 可以定义新的方法，但是不能重写基础结构中有的方法
    // ...
endstruct
```

读者可能会问，这和“多级继承”有什么区别呢？？语法完全一样！回忆一下多级继承的内容：

一个继承了接口的结构还可以再被其他结构所继承，而接口里规定的方法也可以被重写，当然，也可以选择不重写，直接继承父类的。例如：

```
// 母鸡
struct Hen extends Chicken
    // 可以对Animal中定义的cry方法再次重写
    method cry takes nothing returns nothing
        call BJDebugMsg("咯咯咯……")
    endmethod
endstruct
```

```
// 公鸡
struct Rooster extends Chicken
    // 也可以不重写cry方法，直接继承Chicken类里面的
endstruct
```

奥妙就是：这个“基础结构”并没有继承接口，这个“扩展”的过程没有多态的成分。

一个扩展结构的例子，是 vJass 作者写的，这个例子也是一个系统（表格系统），给大家参考下。

其中“method operator [] takes $type$ key returns integer”这种语法是操作符重载，在下一章会讲到。

```
library Table initializer init

//*************************************************************
//* Table object
//* ------------
//*
//*   set t=Table.create() - instanceates a new table object
//*   call t.destroy()     - destroys it
//*   t[1234567]           - Get value for key 1234567
//*                          (zero if not assigned previously)
//*   set t[12341]=32      - Assigning it.
//*   call t.flush(12341)  - Flushes the stored value, so it
//*                          doesn't use any more memory
//*   t.exists(32)         - Was key 32 assigned? Notice
//*                          that flush() unassigns values.
//*   call t.reset()       - Flushes the whole contents of the
//*                          Table.
//*
//*   call t.destroy()     - Does reset() and also recycles the id.
//*
//*   If you use HandleTable instead of Table, it is the same
//* but it uses handles as keys, the same with StringTable.
//*
//*************************************************************

//===========================================================
    globals
        private gamecache vexgc
    endglobals

    private struct VexorianBaseTable
        method reset takes nothing returns nothing
            call FlushStoredMission(vexgc,"VexorianTableSystem:MissionKey"
+ I2S(this))
        endmethod

        private method onDestroy takes nothing returns nothing
            call FlushStoredMission(vexgc,"VexorianTableSystem:MissionKey"
+ I2S(this))
        endmethod
    endstruct
```

```
    //Hey: Don't instanciate other people's textmacros that you are not
supposed to, thanks.
    //! textmacro VexorianTableMake takes name, type, key
    struct $name$ extends VexorianBaseTable

        method operator [] takes $type$ key returns integer
            return GetStoredInteger(vexgc,"VexorianTableSystem:MissionKey"
+ I2S(this), $key$)
        endmethod

        method operator []= takes $type$ key, integer value returns nothing
            call StoreInteger(vexgc,"VexorianTableSystem:MissionKey" +
I2S(this),$key$, value)
        endmethod

        method flush takes $type$ key returns nothing
            call FlushStoredInteger(vexgc,"VexorianTableSystem:MissionKey"
+ I2S(this), $key$)
        endmethod

        method exists takes $type$ key returns boolean
            return HaveStoredInteger(vexgc,"VexorianTableSystem:MissionKey"
+ I2S(this), $key$)
        endmethod
    endstruct
    //! endtextmacro

    //! runtextmacro VexorianTableMake("Table","integer","I2S(key)")
    //! runtextmacro VexorianTableMake("RealTable","real","R2S(key)")
    //! runtextmacro VexorianTableMake("StringTable","string","key")
    //! runtextmacro
VexorianTableMake("HandleTable","handle","I2S(H2I(key))")

    //===========================================================
    // initialize it all.
    //
    private function init takes nothing returns nothing
       call FlushGameCache(InitGameCache("table.w3v"))
        set vexgc = InitGameCache("table.w3v")
    endfunction
endlibrary
```