

角色技能 与 Strategy

Design Pattern - Strategy

张子阳

www.tracefact.net

jimmy_dev@163.com

引言

看过一些设计模式方面的书籍和文章，虽然很正式，很权威，（也觉得有那么一点刻板），总是觉得让人不那么好靠近。于是，我思考着像写故事一样来写下自己对设计模式的理解。我们将以一款奇幻角色扮演游戏(D&D)为蓝本，通过游戏中的模块创建或者功能实现来展示 GOF 的设计模式。当然，这不是一款真正意义上的游戏，只是为了了解设计模式，所以，我会尽可能的使游戏简单。废话不多说了，我们 Start off 吧。

继承及其问题

在开始我们的游戏之旅之前，我们需要定义玩家可以选择的角色。我们首先想到了四个角色职业：野蛮人(Barbarian)、佣兵(Soldier)、圣骑士(Paladin)、法师(Wizard)。

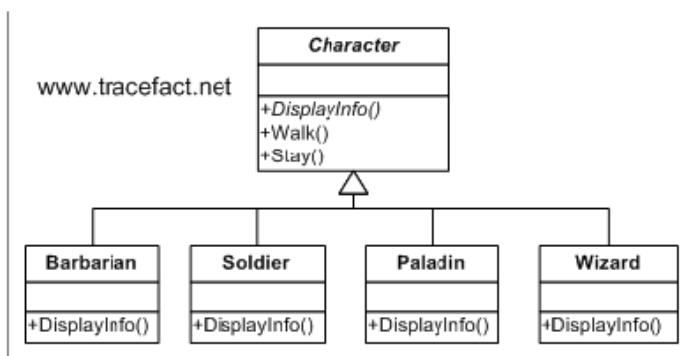
按照 OO 的思想，我们需要先定义一个抽象类作为基类，然后供这四个职业继承，以实现代码的重用。在此之前，我来分析一下角色拥有的能力(方法)：

- DisplayInfo(): 显示角色的基本信息。（比如圣骑士：追求至善的热情、维护法律的意志、击退邪恶的力量 -- 这就是圣骑士的三件武器 ... ）
- Walk(): 让角色行走。
- Stay(): 让角色站立。

一般来说，设计时会遵循这样的原则：

1. 对于所有继承类都有，但是每个继承类的实现各不相同的方法，我们在基类中只给出定义，不给出实现，而在继承类中予以实现。换言之，就是在基类中定义一个抽象方法。
2. 对于所有继承类都有，并且每个继承类的实现完全相同的方法，我们直接在基类中实现它，而由子类去继承，以实现代码重用。

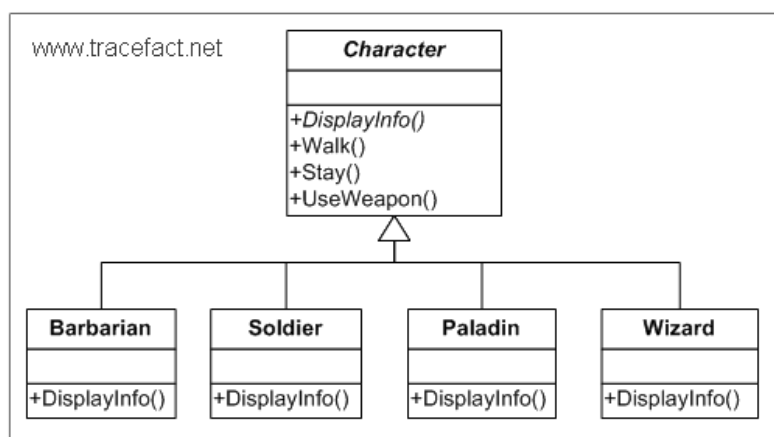
很显然，每个角色都拥有 DisplayInfo()、Walk() 和 Stay() 的能力。其中，DisplayInfo() 对于每个角色都不同；而 Walk() 和 Stay() 对每个角色都相同。于是，我们构建基类 Character，实现了这样的设计：



在基类中实现的问题

到目前为止，我们的程序仅实现了四个角色样子各不相同，并且都能行走和站立。为了让角色更丰富一些，现在我们让角色可以装配武器，所以，我们需要新添一个方法，我们给它命名为 `UseWeapon()`，它的实现效果是角色手中拿起一把剑。我们首先想到的是可以将 `UseWeapon()` 放到基类中，这样可以实现代码的重用。

于是，我们的设计变成下图：



这样看上去很不错，我们利用了面向对象四大思想(抽象、封装、继承、多态)中的继承。可是好景不长，没过几天，我们觉得这样的角色设置有些单调，个性不鲜明，我们想要对游戏规则做如下修改：

- 1、野蛮人用斧。
- 2、法师不用武器。

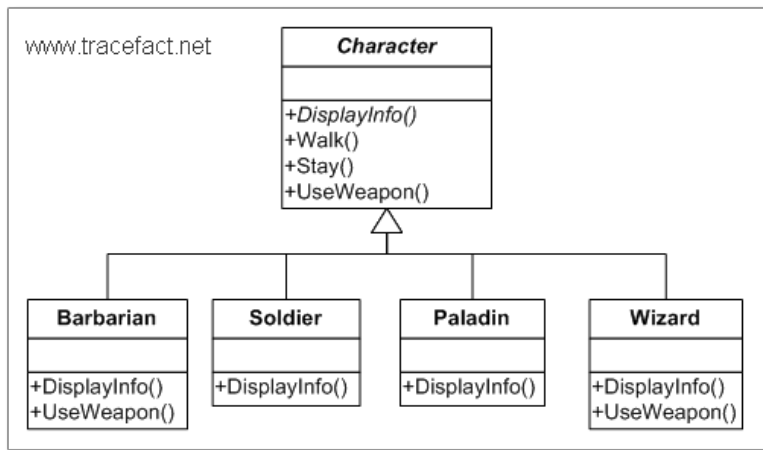
这样，便引出了在本例中使用继承的问题：我们发现法师、野蛮人都可以用剑，而这不符合我们定义的游戏规则。

我们将上面的问题抽象化，得到的结论是：**给基类添加实体方法，使得不应该拥有此方法的子类也拥有了此方法，也使得所有子类方法拥有了完全一样的实现。**

覆盖基类方法的问题

这个问题似乎很好解决，既然野蛮人和法师不同，那我们只需要让野蛮人和法师覆盖基类的 `UseWeapon()` 方法就可以了，与此同时，我们将基类方法声明为 `virtual` 虚拟方法，并给出实现。这个实现，可以视为角色的默认实现(默认角色用剑)。

这一次，我们的设计变成了下面这样：



这样看上去似乎很好的解决了这个问题，直到有一天，我们又有了新的需求：

- 1、 需要新添角色 战士(Warrior)，他也使用斧。
- 2、 需要新添角色 牧师(Cleric)，他也不使用武器。

同时也带来了新的问题：

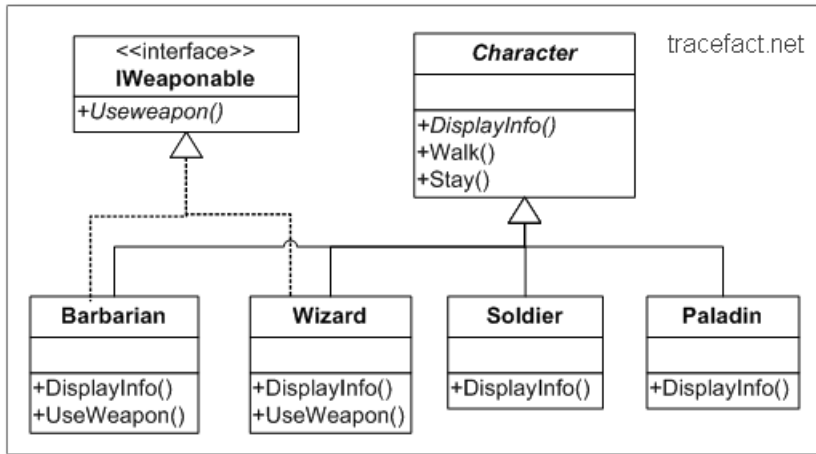
- 1、 我们没有实现代码重用，对于 野蛮人 和 战士，他们对 UseWeapon() 的实现是相同的（都用斧），但我们不得不将完全一样的代码在每个子类中都写一遍。
- 2、 如果这个方法的实现需要经常改动，我们需要反复修改所有相关子类中的方法。
- 3、 牧师 和 法师都不使用武器，但是他们都继承了 UseWeapon() 方法，即便是用一个什么都不做的(空的)UseWeapon() 方法覆盖基类方法，他们仍会暴露出 UseWeapon() 的能力（可以从他们的实例中访问此方法）。

接口及其问题

我们发现继承并不是那么好用，尤其是使用继承 问题 3 似乎难以解决，于是我们改变思路，这一次，我们使用接口来实现。

我们定义一个 IWeaponable 接口，然后从基类中去除 UseWeapon() 方法，然后对于可以使用武器的子类，实现这个接口；对于不可以使用武器的子类(牧师、法师)，不去实现这个接口。

现在的设计变成了这样：



使用接口所产生的新问题远比它解决的问题多，我们首先看下它解决了什么问题：

- 牧师、法师 不再具有使用武器的能力，它们的实例也不会暴露出 UseWeapon() 方法。

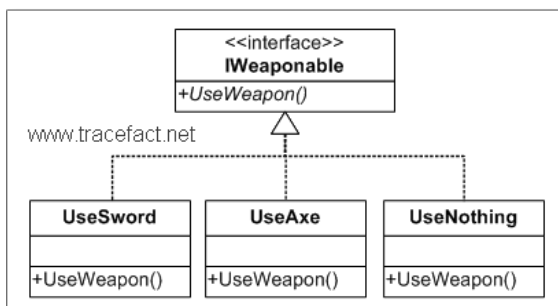
再看它产生了哪些问题：

1. 因为接口只是一个契约，而不包含实现，于是将有大量的子类需要实现此接口。
2. 代码没有重用，所有用剑、用斧的角色，其 UseWeapon() 的实现方式都相同。如果将来需要修改，所有的相关子类都需要改动。

封装行为

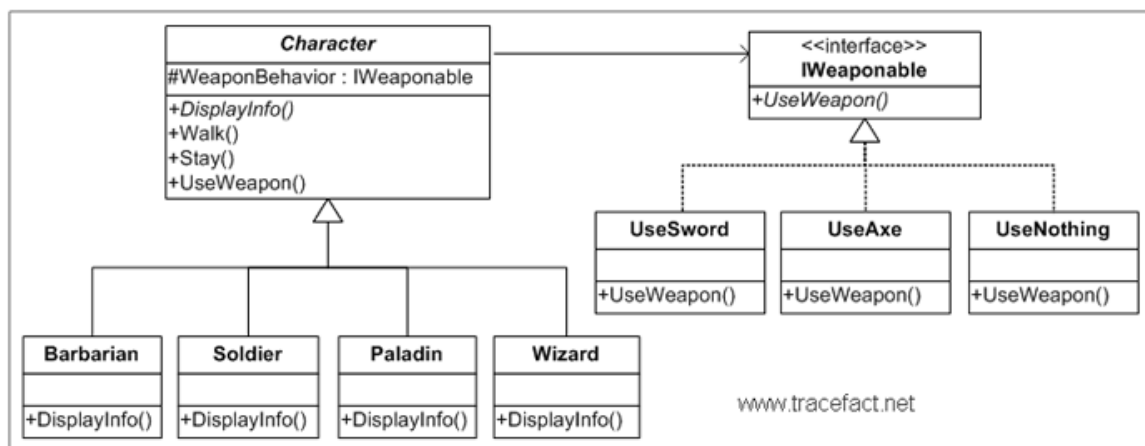
到目前位置，我们在进行这个角色设计的时候，不管是使用继承还是使用接口，UseWeapon() 方法要么是在基类中实现，要么是在子类中实现，我们实际上都是在面向实现编程。OO 的一个原则是面向接口编程。面向实现编程的一个主要问题就是不够灵活，以本例而言，所有的角色，要么用剑，要么用斧，要么什么都不用。如果我想让同一个角色先用斧，再用剑，也就是动态地给他分配武器，是无法实现的。

OO 有一个原则称作“Encapsulate what varies”，也就是俗称的“封装变化”。在我们当前的情况中，UseWeapon() 这个行为是不断变化的，那么我们就应该想办法将它封装起来。这一次，我们依然要使用接口，但是实现此接口的类不再是我们定义的角色基类或者子类，而是专用于 UseWeapon() 这个行为的类。如下图所示：



可以看出：我们将对接口实现分放到了它自己的继承体系中，而不是放到我们的角色类中。每一个实现此接口的类完成一个特定的对 UseWeapon() 方法的实现。比如说，UseSword 类的 UseWeapon() 实现是 拿起一把剑。UseAxe 类的 UseWeapon 实现是 拿起一把斧。而 UseNothing 的实现是什么都不做，仅仅由角色发一句抱怨：I can't use any weapon。

现在我们要做的，就是将这个方法体系 与 我们的角色体系结合起来，具体如何做呢？就是在我们角色的基类中声明一个 IWeaponable 类型的变量，把它复合进去。如下图所示：



注意到 Character 类中仍有一个 UseWeapon() 方法，但是这个方法与之前的 UseWeapon() 方法不同：

1. 它不是用来给子类去覆盖的。
2. 我们通过 Character 的 UseWeapon() 方法实际去调用 IWeaponable 接口的 UseWeapon() 方法(实际上调用了其实体类的 UseWeapon() 方法)。

所以，它的代码大致是这样的：

```
public void UseWeapon(){
    WeaponBehavior.UseWeapon();
}
```

而 WeaponBehavior 在使用之前是应该被赋值的，我们在子类的构造函数中去做这件事：

```
public Barbarian(){
    WeaponBehavior = new UseAxe();
}
```

同时，因为各种 UseWeapon 的方法都被封装在了 WeaponBehavior 中，我们可以动态地改变它，我们给 Character 基类再添加一个更换武器的方法：

```
public void ChangeWeapon(IWeaponable newWeapon){
    WeaponBehavior = newWeapon;
}
```

Strategy 模式

实际上，我们上面所做的一切，就完成了 Strategy 模式，现在让我们看看 Strategy 模式的官方定义：

Strategy 模式定义了一系列的算法，将它们每一个进行封装，并使它们可以相互交换。Strategy 模式使得算法不依赖于使用它的客户端。

代码实现与测试

我们已经完成了设计，现在通过代码来实现整个过程，简单起见，我们只创建两个角色和三种武器技能(包含一个不能使用武器的实现)，必要的说明会放在注释中。

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Strategy {

    #region 封装 UseWeapon() 行为

    // 定义使用 武器接口
    public interface IWeaponable {
        void UseWeapon();
    }

    // 使用 剑 的类
    public class UseSword : IWeaponable {
        public void UseWeapon() {
            Console.WriteLine("Action: Sword Armed. There is a sharp sword in my hands
now.");
        }
    }

    // 使用 斧 的类
    public class UseAxe : IWeaponable {
        public void UseWeapon() {
            Console.WriteLine("Action: Axe Armed. There is a heavy axe in my hands
now.");
        }
    }

    // 不能使用武器的类
    public class UseNothing : IWeaponable {
```

```

        public void UseWeapon() {
            Console.WriteLine("Speak: I can't use any weapon.");
        }
    }

#endregion

#region 定义角色类

// 角色基类
public abstract class Character {

    protected IWeaponable WeaponBehavior;    // 通过此接口调用实际的 UseWeapon
方法。

    // 使用武器，通过接口来调用方法
    public void UseWeapon() {
        WeaponBehavior.UseWeapon();
    }

    // 动态地给角色更换武器
    public void ChangeWeapon(IWeaponable newWeapon){
        Console.WriteLine("Haha, I've got a new equip.");
        WeaponBehavior = newWeapon;
    }

    public void Walk() {
        Console.WriteLine("I'm start to walk ...");
    }

    public void Stop() {
        Console.WriteLine("I'm stopped.");
    }

    public abstract void DisplayInfo();    // 显示角色信息
}

// 定义野蛮人
public class Barbarian : Character {

    public Barbarian() {
        // 初始化继承自基类的 WeaponBehavior 变量
        WeaponBehavior = new UseAxe(); // 野蛮人用斧
    }
}

```

```

    }

    public override void DisplayInfo() {
        Console.WriteLine("Display: I'm a Barbarian from northeast.");
    }
}

// 定义圣骑士
public class Paladin : Character {

    public Paladin() {
        WeaponBehavior = new UseSword();
    }

    public override void DisplayInfo() {
        Console.WriteLine("Display: I'm a paladin ready to sacrifice.");
    }
}

// 定义法师
public class Wizard : Character {

    public Wizard() {
        WeaponBehavior = new UseNothing();
    }

    public override void DisplayInfo() {
        Console.WriteLine("Display: I'm a Wizard using powerful magic.");
    }
}
#endregion

// 测试程序
class Program {
    static void Main(string[] args) {
        Character barbarian = new Barbarian(); // 默认情况下野蛮人用斧
        barbarian.DisplayInfo();
        barbarian.Walk();
        barbarian.Stop();
        barbarian.UseWeapon();

        barbarian.ChangeWeapon(new UseSword()); // 现在也可以使用剑
        barbarian.UseWeapon();
    }
}

```

```
        barbarian.ChangeWeapon(new UseNothing()); // 也可以让他什么都用不了，当然一  
不会这样:)  
        barbarian.UseWeapon();  
    }  
}  
}
```

值得注意的地方是：Strategy 模式没有解决我们之前提到的问题 3。法师不应该暴露出 UseWeapon() 的能力 (**NOTE:** 如果在 VS2005 下使用 C# 语言，当你在法师后面按下点号的时候，智能提示上应该找不到 UseWeapon() 方法) 而不是提供一个什么都不做的 UseWeapon() 方法。

总结

在本文中，我们通过一个实现奇幻角色扮演游戏 (RPG) 的技能设计演示了设计模式中的 Strategy 模式。

我们首先以继承的方式来实现，然后分析了继承可能引起的问题；随后又使用接口实现了一遍，分析了使用接口会带来问题。

最后，我们通过封装行为的 Strategy 模式完成了整个设计，并给出了它的定义。

希望这篇文章能对你有所帮助！