

人物构建 与 **Abstract Factory**

Design Pattern - Abstract Factory

张子阳

www.tracefact.net

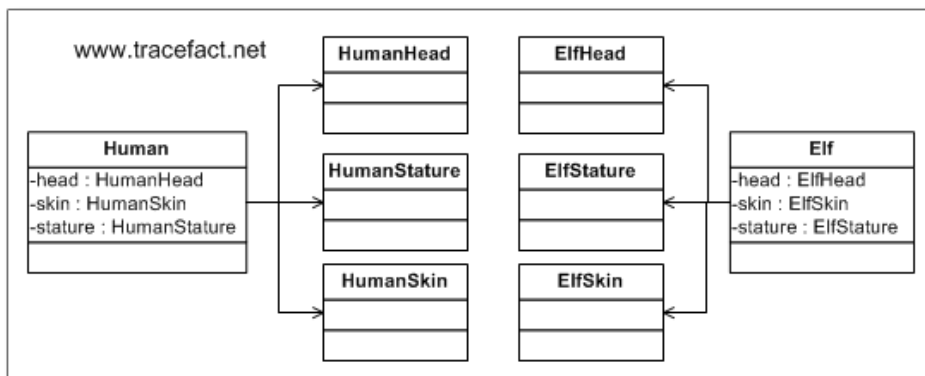
jimmy_dev@163.com

引言

在前一节，我们介绍了 Strategy 模式，并使用此模式实现了一个根据角色的职业来分配技能的范例(实际也就是动态地为类分配方法)。作为一款奇幻 RPG，有了职业，我们还应当可以为角色选择种族，比如说：人类(Human)、精灵(Elf)、矮人(Dwarf)、兽人(Orc)等等。而这四个种族又有着截然不同的外形，精灵皮肤灰白、有着长长的耳朵、没有体毛和胡须；矮人的皮肤与人类近似，但是身材矮小、通常留着浓密的胡子；兽人则有着绿色的皮肤和高大的身躯，并且面目丑陋。本文将讨论如何使用 GOF 的 Abstract Factory 抽象工厂来实现这样的角色外形设计。

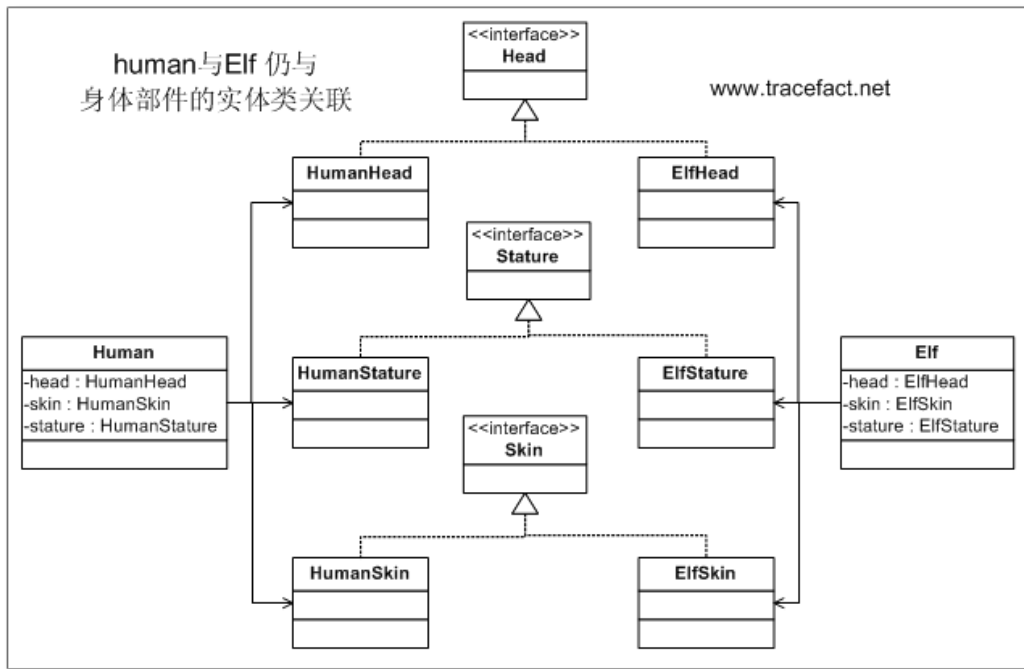
面向实现的方式

简单起见，我们假设角色身体由三部分构成，分别是：头(Head)、身材(Stature)、皮肤(Skin)。那么对于人类的构造，我们的第一反应自然而然地想到：它应当由 HumanHead、HumanStature、HumanSkin 三个类复合而成。对于精灵也是类似的设计，于是，我们实现了下面这样的设计(本文将仅以人类和精灵为例介绍)：

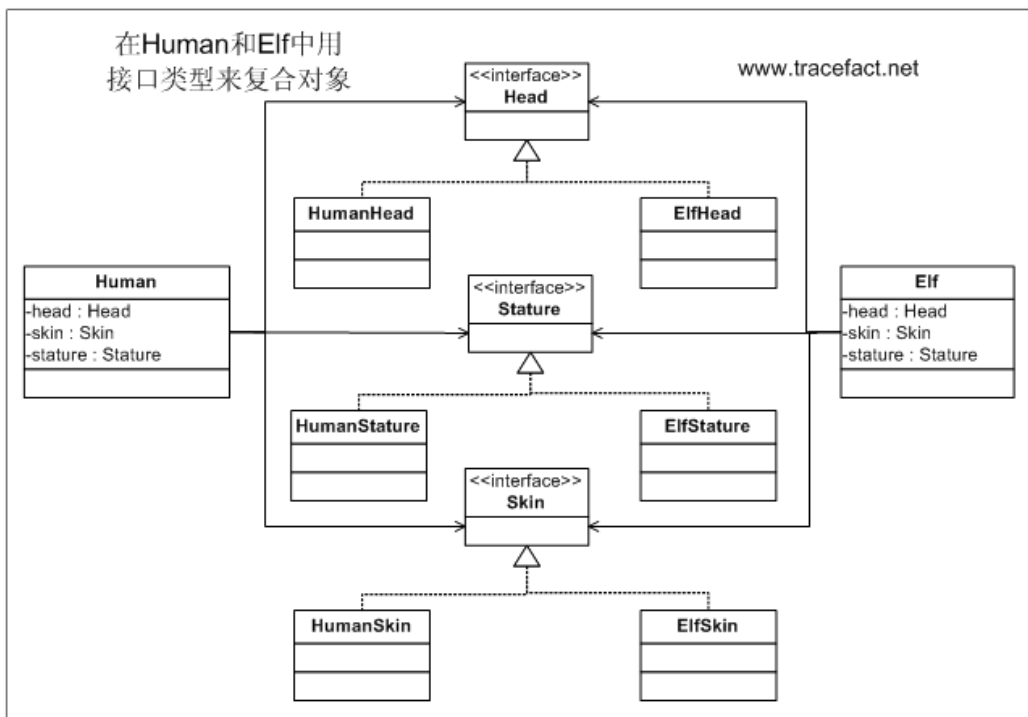


抽象组成身体的实体类

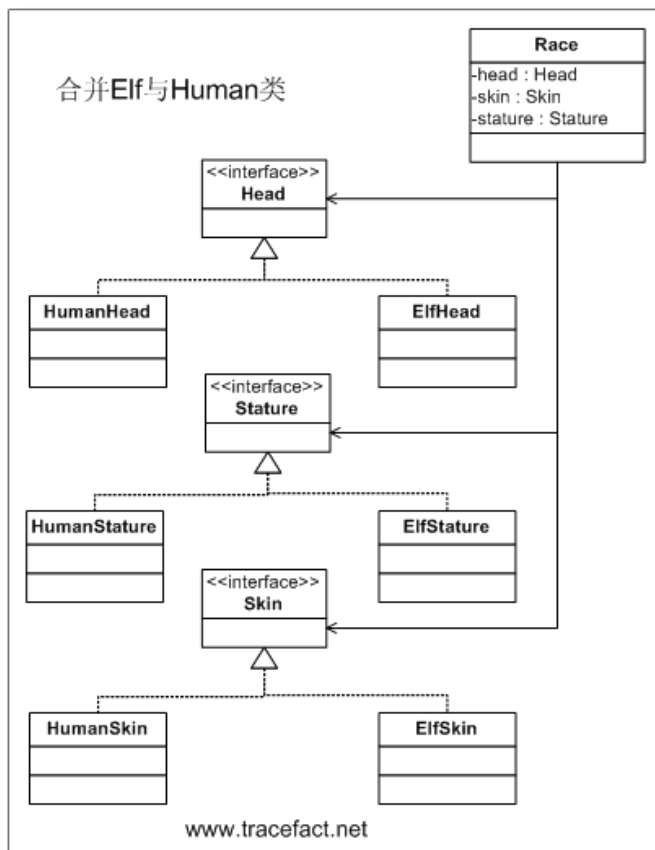
我们发现这样做，每个角色与他的身体部件是牢牢绑定在一起的，每创建一个角色，我们都需要为它先行创建所有其复合的类(组成身体的实体类(Concret Class)，比如 HumanHead)。按照面向对象的思想，我们想到应该对这一过程进行封装，将创建角色部件这件事委派给其他的类来完成。观察上图，我们发现尽管角色不同，但它们都是由三个部分构成，所以，我们所能想到的实现这一过程的第一步，就是对组成身体的实体类进行抽象，我们定义三个接口：Head、Stature、Skin，代表身体的三个部分，并且让 Human 和 Elf 的实体类去实现这个接口：



观察上图,我们发现尽管定义了接口,但是如果角色 Human 和 Elf 仍然与接口的实体类关联,那么效果与面向实现完全相同。现在,是时候对设计做些改动,我们让 Human 和 Elf 与接口关联,而不是与接口的实现关联(OO 思想:面向接口编程,而不是面向实现编程)。这一次,我们的设计变成下图:



一眼望去,我们发现的第一个问题就是: Human 与 Elf 惊人地相似,再仔细看看,我们注意到它们除了名字不同其余的完全一样。我们不禁思考:有必要把两个完全一样的类起两个名字分别保存么?答案是没有必要,我们将它们合并成一个类,起名叫 Race,设计再次变成下面这样:



创建工厂类

看到这里，我们可能会想：现在结构似乎已经很完善了，我们定义了接口来解决问题，也没有为不同的角色创建多个不同的类，而只要在 Race 的构造函数中为代表身体部件的变量赋不同的值，就可以创建不同种族的角色。好的，那么来看看如果要创建一个 Human 代码需要如何编写：

```

Head head = new HumanHead();
Stature stature = new HumanStature();
Skin skin = new HumanSkin();
Race human = new Race(head, stature, skin);
  
```

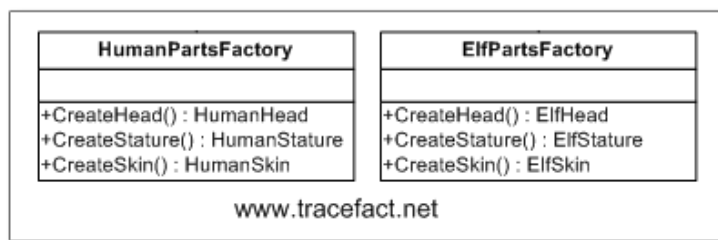
而 Race 的构造函数是这样的：

```

public Race(head, stature, skin){
    this.head = head;
    this.stature = stature;
    this.skin = skin;
}
  
```

我们看到，仅仅创建一个类这样似乎太麻烦了，而且身体的部分类是角色的组成部分，为什么它们要先于角色创建呢？

这时候，我们想到如果有一个类可以专门负责创建身体部件这件事，当我们想要创建角色的时候，将这个类传递给 Race 的构造函数就可以了。我们管创建 Human 身体组成部分的类称作：HumanPartsFactory，创建 Elf 身体部分的类称作 ElfPartsFacotry。那么它们应该是这样的：



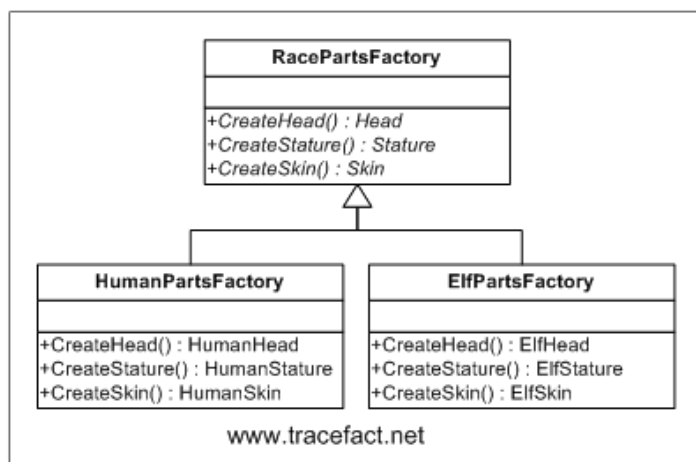
现在，我们再来创建一个 Human，代码变成了这样：

```
HumanPartsFactory humanFactory = new HumanPartsFactory();
Race Human = new Race(humanFactory);
```

相应的，我们的构造函数也需要改一改：

```
public Race(HumanPartsFactory humanFacotry){
    head = humanFactory.CreateHead();
    stature = humanFactory.CreateStature();
    skin = humanFactory.CreateSkin();
}
```

一切似乎都很好，直到我们需要创建一个 Elf 的时候... 我们发现 Race 的构造函数只能接受一个 HumanPartsFactory 类型的参数，为了传递 ElfPartsFactory，我们将不得不再添加一个接受 ElfPartsFactory 类型的构造函数。这样显然不好，这一次，有了之前的经验，我们知道我们可以通过同样的方法来解决。



看到这里，你是否能够体会到一些“面向接口”编程的意味？注意到 RacePartsFactory，它内部的方法返回的都是接口类型，而其实体子类的方法返回的都是接口的实体类。如果我们之

前不声明那看似无用的接口，这里是无法实现的。

现在，我们再次修改 Race 的构造函数：

```
public Race(RacePartsFactory raceFacotry){  
    head = raceFacotry.CreateHead();  
    stature = raceFacotry.CreateStature();  
    skin = raceFacotry.CreateSkin();  
}
```

当我们需要一个 Human 的时候：

```
RacePartsFactory humanFactory = new HumanPartsFactory();  
Race human = new Race(humanFactory);
```

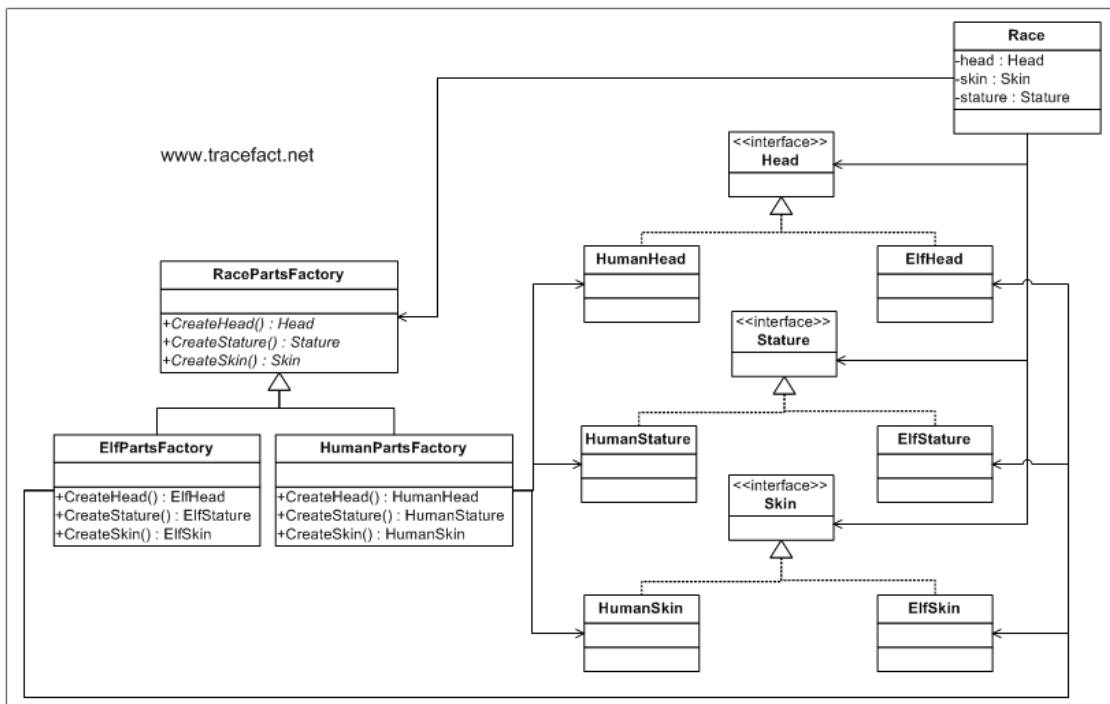
当我们需要一个 Elf 的时候：

```
RacePartsFactory elfFactory = new ElfPartsFactory();  
Race elf = new Race(elfFactory);
```

Abstract Factory 设计模式

上面做的这些，使我们又完成了一个设计模式：Abstract Factory。它的正式定义是这样的：提供一个接口用于创建一系列相互关联或者相互依赖的对象，而不需要指定它们的实体类。

下面是本例中 Abstract Factory 模式的最终图：



代码实现和测试

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

namespace AbstractFactory {

    // 定义构成身体部分的接口
    public interface IHead { string name { get; } }
    public interface IStature { string name { get; } }
    public interface ISkin { string name { get; } }

    // 组成 Human 的类
    public class HumanHead : IHead { public string name { get { return "Human Head"; } } }
    public class HumanStature : IStature { public string name { get { return "Human Stature"; } } }
    public class HumanSkin : ISkin { public string name { get { return "Human Skin"; } } }

    // 组成 Elf 的类
    public class ElfHead : IHead { public string name { get { return "Elf Head"; } } }
    public class ElfStature : IStature { public string name { get { return "Elf Stature"; } } }
    public class ElfSkin : ISkin { public string name { get { return "Elf Skin"; } } }

    // 定义工厂接口
    public interface IRacePartsFactory {
        IHead CreateHead();
        ISkin CreateSkin();
        IStature CreateStature();
    }

    // 定义 Human 身体的工厂类
    public class HumanPartsFactory : IRacePartsFactory {
        public IHead CreateHead() {
            return new HumanHead();
        }
        public IStature CreateStature() {
            return new HumanStature();
        }
        public ISkin CreateSkin() {
            return new HumanSkin();
        }
    }
}
```

```

// 定义 Elf 身体的工厂类
public class ElfPartsFactory : IRacePartsFactory {
    public IHead CreateHead() {
        return new ElfHead();
    }
    public IStature CreateStature() {
        return new ElfStature();
    }
    public ISkin CreateSkin() {
        return new ElfSkin();
    }
}

// 定义 Race 类
public class Race {
    public IHead Head; // 做示范用, 所以没有构建属性
    public IStature Stature;
    public ISkin Skin;

    public Race(IRacePartsFactory raceFactory) {
        Head = raceFactory.CreateHead();
        Stature = raceFactory.CreateStature();
        Skin = raceFactory.CreateSkin();
    }
}

class Program {
    static void Main(string[] args) {
        // 创建一个 精灵(Elf)
        Race Elf = new Race(new HumanPartsFactory());
        Console.WriteLine(Elf.Head.name);
        Console.WriteLine(Elf.Stature.name);
        Console.WriteLine(Elf.Skin.name);
    }
}
}

```

总结

本文中我们一步步学习了 Abstract Factory 抽象工厂模式的实现。

我首先介绍了我们奇幻 RPG 所面临的一个问题：我们需要创建形态各异的角色。随后，我们通过面向实现的方式来完成了这一过程，并讨论了它的不足。随后，我们先通过接口的使用对种

族进行了抽象。接着，我们将创建角色组成部分分类的过程进行了封装，将这一过程委派给了工厂类。最后，我们又对工厂类进行了抽象，最终实现了 Abstract Factory 工厂模式。

希望本文能给你带来帮助。