Object Oriented Design/Programming and Design Patterns

deepwaterooo

April 23, 2018

Contents

1 Object Oriented Design/Programming and Design Patterns

1.1 设计模式的几个原则

- http://www.cnblogs.com/zhaoqingqing/p/4288454.html
- 原则 1：单一职责
- 原则 2：里氏替换原则（子类扩展但不改变父类功能）

- 原则 3：依赖倒置原则
- 原则 4：接口隔离原则
- 原则 5：迪米特法则（最少知道原则）
- 原则 6：开闭原则

1.1.1 原则 1：单一职责原则

- 说到单一职责原则，很多人都会不屑一顾。
- 因为它太简单了，稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。
- 在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。
- 而避免出现这一问题的方法便是遵循单一职责原则。
- 虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。
- 为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责被分化成了更细的职责。

1. 用一个类描述动物呼吸这个场景

```
class Animal {
    public void breathe(string animal) {
        Debug.Log(animal + " 呼吸空气");
    }
}
public class Client {
    Animal animal = new Animal();
    void Start() {
        animal.breathe(" 牛");
        animal.breathe(" 羊");
        animal.breathe(" 猪");
    }
}
//运行结果:
//牛呼吸空气
//羊呼吸空气
//猪呼吸空气
```

2. 当需求变动

    - 程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。
    - 修改时如果遵循单一职责原则，需要将 Animal 类细分为陆生动物类 Terrestrial，水生动物 Aquatic，代码如下：

```
class Terrestrial {
    public void breathe(String animal) {
        Debug.Log(animal + " 呼吸空气");
    }
}
class Aquatic {
    public void breathe(String animal) {
        Debug.Log(animal + " 呼吸水");
    }
}
public class Client {
    public static void main(String[] args) {
```

```
        Terrestrial terrestrial = new Terrestrial();
        terrestrial.breathe(" 牛");
        terrestrial.breathe(" 羊");
        terrestrial.breathe(" 猪");
        Aquatic aquatic = new Aquatic();
        aquatic.breathe(" 鱼");
    }
}
//运行结果:
//牛呼吸空气
//羊呼吸空气
//猪呼吸空气
//鱼呼吸水
```

3. 改动量小的方法

   - 我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。
   - 而直接修改类 Animal 来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下:

```
class Animal {
    public void breathe(String animal) {
        if (" 鱼" == animal) {
            Debug.Log((animal + " 呼吸水"));
        }
        else {
            Debug.Log((animal + " 呼吸空气"));
        }
    }
}
public class Client {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.breathe(" 牛");
        animal.breathe(" 羊");
        animal.breathe(" 猪");
        animal.breathe(" 鱼");
    }
}
```

4. 隐患

   - 可以看到，这种修改方式要简单的多。
   - 但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，
   - 则又需要修改 Animal 类的 breathe 方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，
   - 也许某一天你会发现程序运行的结果变为“牛呼吸水”了。
   - 这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。

5. 另一种修改方式

```
class Animal {
    public void breathe(String animal) {
        Debug.Log(animal + " 呼吸空气");
    }
    public void breathe2(String animal) {
        Debug.Log(animal + " 呼吸水");
    }
}
public class Client {
    public static void main(String[] args) {
```

```
        Animal animal = new Animal();
        animal.breathe(" 牛");
        animal.breathe(" 羊");
        animal.breathe(" 猪");
        animal.breathe2(" 鱼");
    }
}
```

- 可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，
- 但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，
- 那么在实际编程中，采用哪一中呢?
- 其实这真的比较难说，需要根据实际情况来确定。
- 我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则。

6. 遵循单一职责原的优点有

- 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
- 提高类的可读性，提高系统的可维护性；
- 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。
- 需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

1.1.2 原则 2：里氏替换原则

1. 名字的由来

- 肯定有不少人跟我刚看到这项原则的时候一样，对这个原则的名字充满疑惑。
- 其实原因就是这项原则最早是在 1988 年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。
- 简单来说的话，就是当我们使用继承时，遵循里氏替换原则。

2. 定义

- 注：类 B 继承类 A 时，除添加新的方法完成新增功外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。
- 继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，
- 虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，
- 就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。
- 继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。
- 比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，
- 则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，
- 所有涉及到子类的功能都有可能会产生故障。

3. 继承的风险

- 那就让我们一起看看继承的风险，如下：

```
class A {
    public int func1(int a, int b) {
        return a - b;
    }
}
```

```
public class Client {
    void Start() {
        A a = new A();
        Debug.Log("100-50=" + a.func1(100, 50));
        Debug.Log("100-80=" + a.func1(100, 80));
    }
}
// 运行结果
// 100-50=50
// 100-80=20
```

4. 需求变动

- 后来，我们需要增加一个新的功能：完成两数相加，然后再与 100 求和，由类 B 来负责。
- 即类 B 需要完成两个功能：
- 两数相减。
- 两数相加，然后再加 100。
- 由于类 A 已经实现了第一个功能，所以类 B 继承类 A 后，只需要再完成第二个功能就可以了，代码如下

```
class B:A {
    public int func1(int a, int b) {
        return a + b;
    }
    public int func2(int a, int b) {
        return func1(a, b) + 100;
    }
}
public class Client {
    private void Start() {
        B b = new B();
        Debug.Log("100-50=" + b.func1(100, 50));
        Debug.Log("100-80=" + b.func1(100, 80));
        Debug.Log("100+20+100=" + b.func2(100, 20));
    }
}
// 类 B 运行结果
// 100-50=150
// 100-80=180
// 100+20+100=220
```

5. 影响了正常的功能

- 我们发现原本运行正常的相减功能发生了错误。
- 原因就是类 B 在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类 B 重写后的方法，造成原本运行正常的功能出现了错误。
- 在本例中，引用基类 A 完成的功能，换成子类 B 之后，发生了异常。
- 在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，
- 但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。
- 如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。

6. 里氏替换原则通俗的来讲就是

- 子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下 4 层含义：
- 1. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
- 2. 子类中可以增加自己特有的方法。

- 3. 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
- 4. 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。
- 看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。
- 所以大家都会产生这样的疑问，假如我非要不遵循里氏替换原则会有什么后果?
- 后果就是：你写的代码出问题的几率将会大大增加。

1.1.3 原则 3：依赖倒置原则

1. 定义

- 高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。
- 以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。
- 抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

2. 依赖倒置原则核心思想

- 依赖倒置原则的核心思想是面向接口编程，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。

3. 情景举例

- 场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

```
class Book {
    public String getContent() {
        return " 很久很久以前有一个阿拉伯的故事.....";
    }
}
class Mother {
    public void narrate(Book book) {
        Debug.Log(" 妈妈开始讲故事");
        Debug.Log(book.getContent());
    }
}
public class Client {
    void Start() {
        Mother mother = new Mother();
        mother.narrate(new Book());
    }
}
// 运行结果:
// 妈妈开始讲故事
// 很久很久以前有一个阿拉伯的故事.....
```

4. 需求变动

- 运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

```
class Newspaper {
    public String getContent() {
        return " 林书豪 38+7 领导尼克斯击败湖人.....";
    }
}
```

- 这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改 Mother 才能读。
- 假如以后需求换成杂志呢？换成网页呢?
- 还要不断地修改 Mother，这显然不是好的设计。

- 原因就是 Mother 与 Book 之间的耦合性太高了，必须降低他们之间的耦合度才行。

5. 抽象的接口

- 我们引入一个抽象的接口 IReader。
- 读物，只要是带字的都属于读物：

```
interface IReader {
    String getContent();
}
```

- Mother 类与接口 IReader 发生依赖关系，而 Book 和 Newspaper 都属于读物的范畴，
- 他们各自都去实现 IReader 接口，这样就符合依赖倒置原则了，代码修改为：

```
interface IReader {
    String getContent();
}
class Newspaper : IReader {
    public String getContent() {
        return " 林书豪 17+9 助尼克斯击败老鹰……";
    }
}
class Book : IReader {
    public String getContent() {
        return " 很久很久以前有一个阿拉伯的故事……";
    }
}
class Mother {
    public void narrate(IReader reader) {
        Debug.Log(" 妈妈开始讲故事");
        Debug.Log(reader.getContent());
    }
}
public class Client {
    public static void main(String[] args) {
        Mother mother = new Mother();
        mother.narrate(new Book());
        mother.narrate(new Newspaper());
    }
}
// 运行结果
// 妈妈开始讲故事
// 很久很久以前有一个阿拉伯的故事……
// 妈妈开始讲故事
// 林书豪 17+9 助尼克斯击败老鹰……
```

这样修改后，无论以后怎样扩展 Client 类，都不需要再修改 Mother 类了。

- 这只是一个简单的例子，实际情况中，代表高层模块的 Mother 类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。
- 所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。
- 采用依赖倒置原则给多人并行开发带来了极大的便利，
- 比如上例中，原本 Mother 类与 Book 类直接耦合时，Mother 类必须等 Book 类编码完成后才可以进行编码，因为 Mother 类依赖于 Book 类。
- 修改后的程序则可以同时开工，互不影响，因为 Mother 与 Book 类一点关系也没有。
- 参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。
- 现在很流行的 TDD 开发模式就是依赖倒置原则最成功的应用。

6. 在实际编程中，我们一般需要做到如下 3 点

- 1. 低层模块尽量都要有抽象类或接口，或者两者都有。

- 2. 变量的声明类型尽量是抽象类或接口。使用继承时遵循里氏替换原则。
- 3. 依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

1.1.4 原则 4：接口隔离原则

1. 定义

- 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。
- 将臃肿的接口 I 拆分为独立的几个接口，类 A 和类 C 分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。
- 举例来说明接口隔离原则：

2. 未遵循接口隔离原则的设计

- 类图 1
- 这个图的意思是：类 A 依赖接口 I 中的方法 1、方法 2、方法 3，类 B 是对类 A 依赖的实现。
- 类 C 依赖接口 I 中的方法 1、方法 4、方法 5，类 D 是对类 C 依赖的实现。
- 对于类 B 和类 D 来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口 I，所以也必须要实现这些用不到的方法。

3. 示例代码

- 对类图不熟悉的可以参照程序代码来理解，代码如下：

```
//接口
interface I {
    void method1();
    void method2();
    void method3();
    void method4();
    void method5();
}

class A {
    public void depend1(I i) {
        i.method1();
    }
    public void depend2(I i) {
        i.method2();
    }
    public void depend3(I i) {
        i.method3();
    }
}

class B : I {
    public void method1() {
        Debug.Log(" 类 B 实现接口 I 的方法 1");
    }
    public void method2() {
        Debug.Log(" 类 B 实现接口 I 的方法 2");
    }
    public void method3() {
        Debug.Log(" 类 B 实现接口 I 的方法 3");
    }
    //对于类 B 来说，method4 和 method5 不是必需的，但是由于接口 A 中有这两个方法，
    //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。
    public void method4() { }
    public void method5() { }
}
```

```
37
38  class C {
39      public void depend1(I i) {
40          i.method1();
41      }
42      public void depend2(I i) {
43          i.method4();
44      }
45      public void depend3(I i) {
46          i.method5();
47      }
48  }
49
50  class D : I {
51      public void method1() {
52          Debug.Log(" 类 D 实现接口 I 的方法 1");
53      }
54      //对于类 D 来说，method2 和 method3 不是必需的，但是由于接口 A 中有这两个方法，
55      //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。
56      public void method2() { }
57      public void method3() { }
58      public void method4() {
59          Debug.Log(" 类 D 实现接口 I 的方法 4");
60      }
61      public void method5() {
62          Debug.Log(" 类 D 实现接口 I 的方法 5");
63      }
64  }
65
66  public class Client {
67      void Start() {
68          A a = new A();
69          a.depend1(new B());
70          a.depend2(new B());
71          a.depend3(new B());
72          C c = new C();
73          c.depend1(new D()));
74          c.depend2(new D());
75          c.depend3(new D());
76      }
77  }
```

- 可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。
- 如果将这个设计修改为符合接口隔离原则，就必须对接口 I 进行拆分。

4. 遵循接口隔离原则的设计

- 在这里我们将原有的接口 I 拆分为三个接口，拆分后的设计如图 2 所示：
- 类图 2

5. 示例代码

- 照例贴出程序的代码，供不熟悉类图的朋友参考：

```
1  interface I1 {
2      void method1();
3  }
4
5  interface I2 {
6      void method2();
```

```
    void method3();
}

interface I3 {
    void method4();
    void method5();
}

class A {
    public void depend1(I1 i) {
        i.method1();
    }
    public void depend2(I2 i) {
        i.method2();
    }
    public void depend3(I2 i) {
        i.method3();
    }
}

class B : I1, I2 {
    public void method1() {
        Debug.Log(" 类 B 实现接口 I1 的方法 1");
    }
    public void method2() {
        Debug.Log(" 类 B 实现接口 I2 的方法 2");
    }
    public void method3() {
        Debug.Log(" 类 B 实现接口 I2 的方法 3");
    }
}

class C {
    public void depend1(I1 i) {
        i.method1();
    }
    public void depend2(I3 i) {
        i.method4();
    }
    public void depend3(I3 i) {
        i.method5();
    }
}

class D : I1, I3 {
    public void method1() {
        Debug.Log(" 类 D 实现接口 I1 的方法 1");
    }
    public void method4() {
        Debug.Log(" 类 D 实现接口 I3 的方法 4");
    }
    public void method5() {
        Debug.Log(" 类 D 实现接口 I3 的方法 5");
    }
}
```

- 接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。
- 也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去

调用。

- 本文例子中，将一个庞大的接口变更为 3 个专用的接口所采用的就是接口隔离原则。
- 在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。
- 接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
- 说到这里，很多人会觉的接口隔离原则跟之前的单一职责原则很相似，其实不然。
- 其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。
- 其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；
- 而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

6. 注意几点

- 采用接口隔离原则对接口进行约束时，要注意以下几点：
- 1. 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
- 2. 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
- 3. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。
- 运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。
-

1.1.5 原则 5：迪米特法则

1. 定义

- 一个对象应该对其他对象保持最少的了解
- 类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。
- 因此，尽量降低类与类之间的耦合。
- 自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。
- 无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。
- 低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

2. 最少知道原则

- 迪米特法则又叫最少知道原则，最早是在 1987 年由美国 Northeastern University 的 Ian Holland 提出。
- 通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的 public 方法，不对外泄漏任何信息。
- 迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：
- 每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。
- 耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，
- 而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

3. 违反迪米特法则的设计

- 举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工 ID。
- 先来看一下违反迪米特法则的设计。

```
//总公司员工
class Employee {
    private String id;
    public void setId(String id) {
        this.id = id;
    }
    public String getId() {
        return id;
    }
}

//分公司员工
class SubEmployee {
    private String id;
    public void setId(String id) {
        this.id = id;
    }
    public String getId() {
        return id;
    }
}

class SubCompanyManager {
    public List<SubEmployee> getAllEmployee() {
        List<SubEmployee> list = new List<SubEmployee>();
        for (int i = 0; i < 100; i++) {
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个 ID
            emp.setId(" 分公司" + i);
            list.Add(emp);
        }
        return list;
    }
}

class CompanyManager {
    public List<Employee> getAllEmployee() {
        List<Employee> list = new List<Employee>();
        for (int i = 0; i < 30; i++) {
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个 ID
            emp.setId(" 总公司" + i);
            list.Add(emp);
        }
        return list;
    }
    public void printAllEmployee(SubCompanyManager sub) {
        List<SubEmployee> list1 = sub.getAllEmployee();
        foreach (SubEmployee e in list1) {
            Debug.Log(e.getId());
        }
        List<Employee> list2 = this.getAllEmployee();
        foreach (Employee e in list2) {
            Debug.Log(e.getId());
        }
    }
}

```

```
public class Client {
    void Start() {
        CompanyManager e = new CompanyManager();
        e.printAllEmployee(new SubCompanyManager());
    }
}
```

- 现在这个设计的主要问题出在 CompanyManager 中，根据迪米特法则，只与直接的朋友发生通信，
- 而 SubEmployee 类并不是 CompanyManager 类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，
- 与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。

4. 修改后的代码

- 按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下:

```
class SubCompanyManager {
    public List<SubEmployee> getAllEmployee() {
        List<SubEmployee> list = new List<SubEmployee>();
        for (int i = 0; i < 100; i++) {
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个 ID
            emp.setId(" 分公司" + i);
            list.Add(emp);
        }
        return list;
    }
    public void printEmployee() {
        List<SubEmployee> list = this.getAllEmployee();
        foreach (SubEmployee e in list) {
            Debug.Log(e.getId());
        }
    }
}

class CompanyManager {
    public List<Employee> getAllEmployee() {
        List<Employee> list = new List<Employee>();
        for (int i = 0; i < 30; i++) {
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个 ID
            emp.setId(" 总公司" + i);
            list.Add(emp);
        }
        return list;
    }
    public void printAllEmployee(SubCompanyManager sub) {
        sub.printEmployee();
        List<Employee> list2 = this.getAllEmployee();
        foreach (Employee e in list2) {
            Debug.Log(e.getId());
        }
    }
}
```

5. 主要的变化

- 修改后，为分公司增加了打印人员 ID 的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

6. 总结

- 迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。
- 但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，
- 总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。
- 过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。
- 所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

1.1.6 原则 6：开闭原则

1. 定义

   - 一个软件实体如类、模块和函数应该对扩展开放，对修改关闭
   - 在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，
   - 可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。
   - 因此，当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。
   - 闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，
   - 它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。
   - 以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉的他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。
   - 在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。
   - 其实，我们遵循设计模式前面 5 大原则，以及使用 23 种设计模式的目的就是遵循开闭原则。

2. 如何遵守

   - 也就是说，只要我们对前面 5 项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，
   - 前面 5 项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；
   - 如果前面 5 项原则遵守的不好，则说明开闭原则遵守的不好。
   - 其实，开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。
   - 因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。
   - 而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。
   - 当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

1.1.7 如何去遵守这六个原则

- 对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。
- 任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。
- 对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。
- 如果大家对这六项原则的理解跟我有所不同，欢迎指正

1.2 Unity 中各种设计模式的实践与运用

- https://blog.csdn.net/u010019717/article/details/41172783
- https://www.ctolib.com/Unity-Design-Pattern.html
- 23 design patterns.

Creational Patterns	1. Abstract Factory 抽象工厂	创建几个相似的类的一个实例
	2. Builder 生成器	分离对象构造与它的表示
	3. Factory Method 工厂方法	创建几个派生类的一个实例
	4. Prototype 原型	要复制或克隆一个完全初始化的实例
	5. Singleton 单件	一个类只能运行一个实例可以存在
Structural Patterns	6. Adapter 适配器	不同的类的接口相匹配
	7. Bridge 桥接	从其实现分离对象接口
	8. Composite 复合	简单和复合对象的树形结构
	9 .Decorator 装饰者	动态添加到对象的责任
	10. Facade 外观	一个表示整个子系统的单个类
	11. Flyweight 享元	细粒度的实例用于高效共享
	12. Proxy 代理服务器	一个表示另一个对象的对象
Behavioral Patterns	13. Chain of Resp. 职责链模式	一连串的对象之间传递请求的一种方式
	14. Command 命令	将命令请求封装为一个对象
	15. Interpreter 解释器	方法包含程序中的语言元素
	16. Iterator 迭代器	按顺序访问集合中的元素
	17. Mediator 中介者	定义简化的类之间的通信
	18. Memento 备忘录	捕获和还原对象的内部状态
	19. Observer 观察者	一种方式通知到类数目的变化
	20. State 状态	在其状态改变时，改变一个对象的行为
	21. Strategy 策略	封装在类的内部算法
	22. Template Method 模板方法	推迟算法到子类的确切步骤
	23. Visitor 访问者	对一类没有改变定义新的操作

1.2.1 Gang of Four Patterns in Unity （23 种 GOF 设计模式的 Unity 实现）

1. Behavioral Patterns 行为型模式 Command Pattern 命令模式
   - State Pattern 状态模式
   - Observer Pattern 观察者模式
   - Chain of Responsibility Pattern 责任链模式
   - Mediator Pattern 中介者模式
   - Interpreter Pattern 解释器模式
   - Iterator Pattern 迭代器模式
   - Memento Pattern 备忘录模式
   - Strategy Pattern 策略模式
   - Template Method Pattern 模板方法模式
   - Visitor Pattern 访问者模式
2. Structural Patterns 结构型模式
   - Adapter Pattern 适配器模式
   - Bridge Pattern 桥接模式
   - Composite Pattern 组合模式
   - Decorator Pattern 装饰模式
   - Facade Pattern 外观模式
   - Flyweight Pattern 享元模式
   - Proxy Pattern 代理模式

3. Creational Patterns 创建型模式
    - Prototype Pattern 原型模式
    - Singleton Pattern 单例模式
    - Abstract Factory Pattern 抽象工厂模式
    - Builder Pattern 建造者模式
    - Factory Method Pattern 工厂方法模式

1.2.2 Game Programming Patterns in Unity （《游戏编程模式》的 Unity 实现）

- Subclass Sandbox Pattern 子类沙盒模式
- Type Object Pattern 类型对象模式
- Component Pattern 组件模式
- Event Queue Pattern 事件队列模式
- Game Loop Pattern 游戏循环模式
- Service Locator Pattern 服务定位器模式
- Data Locality Pattern 数据局部性模式
- Dirty Flag Pattern 脏标记模式
- Object Pool Pattern 对象池模式

1.3 单例模式（Unity3D/C#）

1.3.1 【单例概述】

- 定义：单例，顾名思义，单个实例，即应用单例模式的类有且只有一个实例对象，并提供一个全局访问点来共其他类与对象访问。
- （1）有且只有一个实例对象。
- 有实例对象说明该类不是抽象类；只有一个实例对象表示不能随时随地的 new 一个该类的对象出来（这不是对开发者的约定，而是代码层面上的约定，即如果你这样做了，编辑器会提示你错误），即该类的构造函数是 private 级别，只能在类的内部构建实例对象；
- （2）提供全局访问点使其他类与对象访问。
- 这说明应用单例模式的类在提供实例访问方法时，该方法应该是静态的。
- 有了如上的分析，就可以很轻松的创建单例类了，代码并不是很复杂。为什么要分成“C# 中的单例模式”和“Unity 中的伪单例模式”，前者指 C# 语言本身，后者指在 Unity 的 Mono 框架下。还有一点要说的，下面所有所有的例子都只是说明单例如何创建，不是说类中就这么点东西。。。- -

1.3.2 【C# 中的单例模式】

- 在单例模式中，根据应用情况的不同，也有着不同的实现方式。先把统一的访问方式写出来，这种访问方式应用于该命题下述的所有模式。

```
namespace CSharpTest {
    class Program {
        static void Main(string[] args) {
            Singleton s = Singleton.GetInstance();
        }
    }
}
```

1. 饿汉模式

- 意思是，在该类装载时构建类的单例。（类的装载可以粗浅的理解为发生在程序启动时，Main 之前）也就是说，这个单例跟你什么时候用，是否要用无关，只要运行程序，这个单例就存在了。这样做的坏处是如果程序初始化时要载入的资源过多时显然这种方式又提高了加载的负担，其次如果没有使用到的话也浪费了内存。

```
namespace CSharpTest {
    public class Singleton {
        // 私有化构造函数，使得外部无法构造类的实例
        private Singleton() { }

        // 定义实例对象时便创建实例
        private static Singleton _instance = new Singleton();

        // 提供全局访问点
        public static Singleton GetInstance() {
            return _instance;
        }
    }
}
```

2. 懒汉模式

- 意思是，在该类的单例被使用时构造类的单例。它相比饿汉模式更加灵活，所以应用更为广泛。

(a) 基本模式（单线程模式）

- 在单线程中，只需做如下定义：

```
namespace CSharpTest {
    public class Singleton {
        //私有化构造函数 使得外部无法构造类的实例
        private Singleton() { }

        //定义一个空的单例对象
        private static Singleton _instance;

        //提供全局访问点
        public static Singleton GetInstance() {
            //第一次访问时会创建实例
            if (_instance == null)
                _instance = new Singleton();
            return _instance;
        }
    }
}
```

(b) 多线程模式

- 在多线程的程序中，构造单例的方式要发生什么变化呢？我们来依据单线程模式构造单例的代码来分析：全局访问方法内提供了如下的判断条件：

```
if (_instance == null)
    _instance = new Singleton();
```

- 这会引发什么问题？以两个线程情况为例。当两个线程运行到这里时，可能线程 1 刚经过判断还没创建实例时，线程 2 就也已经通过判断要创建实例了，这会造成两个线程都创建了实例，这就违背了我们单例模式的初衷。所以我们要对其进行“加锁”，进行争用条件判断。即谁先来的谁先访问，我访问的时候你不许访问，我访问完了你再访问。
- 实现思路如下：
- 这里使用一个辅助对象（必须是引用类型）充当锁，当多个线程同时访问 GetInstance 方法时，第一个进来的锁定该对象，这时，其他线程遇到锁时会挂起等待，当这个线程执行完锁定代码块时解锁，这时第二个进来的线程再锁，解锁之后第三个进来的线程再锁。。。依次类推。这样就避免了多线程访问同一对象时会引发的风险。此例中的风险便是创建多个实例。

```
namespace CSharpTest {
    public class Singleton {

        //私有化构造函数 使得外部无法构造类的实例
        private Singleton() { }

        //定义一个空的单例对象
        private static Singleton _instance;

        //辅助对象
        private static object obj = new object();

        //提供全局访问点
        public static Singleton GetInstance() {
            //加锁，此时其他线程挂起，等待上锁的那个线程执行完事
            lock (obj) {
                //第一次访问时会创建实例
                if (_instance == null)
                    _instance = new Singleton();
            }
            //运行完代码块就解锁了，其他线程此时可以进入
            return _instance;
        }
    }
}
```

这边对线程做个小说明。通常我们学习编程基础时都是单线程模式。当我们开启第二条线程时，两条线程的运行是各自独立，处理各自的逻辑，他们基本上是同时运行的。可能上述例子会有个疑问，为什么可以同时通过判断而不能同时加锁呢？这涉及到两个问题。判断与锁的区别多线程的执行顺序。

- （1）判断与锁的区别。
- 判断中，只要满足条件即可执行相应的代码块，并无其他限制；锁是当一个访问者进入锁的代码块之后马上加锁，其他访问者只能等前一个访问者出来后才能进去，当然，无论谁进去都会马上加锁。
- （2）多线程的执行顺序。
- 这里做两个合理的猜想。一是多个线程各自独立，只是执行的快慢有微小差别，这种速度差别能使一个线程刚通过判断语句还没创建实例时，另外的线程也通过了判断语句；二是多个线程的确是同时过来的，但是在锁之前会出现顺序之分可能是底层的处理机制，因为每个线程都是有自己的标识的，当遇到琐时线程管理器会自动为多个线程分配优先顺序，保证他们有序申请锁定。

(c) 优化多线程模式

- 多线程模式主要解决的问题是当单例未创建时，多个线程同时访问 GetInstance 方法造成单例的多次创建。但现在的解决方案显然是有问题的。首先，单例没创建时，多线程是否会同时访问我们是不清楚的；在单例已经创建时，我们再去访问 GetInstance 方法时其实只需判断 $instance$ --。
- 在分析中很显然提出了解决的办法，加个判断就好了 ~

```
namespace CSharpTest {
    public class Singleton {

        //私有化构造函数 使得外部无法构造类的实例
        private Singleton() { }

        //定义一个空的单例对象
        private static Singleton _instance;

        //辅助对象
        private static object obj = new object();

        //提供全局访问点
        public static Singleton GetInstance() {

            //后续再访问时只要判断实例是否为 null 就行了
```

```
            //不为 null 直接返回 _instance
            //只有未创建时才会启动锁的功能
            if (_instance == null) {

                //加锁，此时其他线程挂起，等待上锁的那个线程执行完事
                lock (obj)
                {
                    //第一次访问时会创建实例
                    if (_instance == null)
                        _instance = new Singleton();
                }
                //运行完代码块就解锁了，其他线程此时可以进入
            }
            return _instance;
        }
    }
}
```

1.3.3 【Unity 中的伪单例模式】

- 以下模式的前提都是单场景。下述的单例模式都是伪单例模式。
- Unity 实际是脚本编程，基于 Mono 框架，类默认继承自 MonoBehavior 可以直接附加到物体上作为组件，组件所在的物体就是这个脚本类的对象，它提供了一种除了 new 之外新的对象构建方式。
- 将脚本类应用于单例模式通常是想应用例如 Update、Start 等 Message 方法，或者应用其组件化的特性在编辑器中设置脚本的成员等等。基本套路是脚本指定给物体上，获取单例使用 FindObjectOfType 方法，这也解释了为什么只能单场景使用，因为场景中的物体会随着场景变更而销毁，而脚本依附在物体上面也会被销毁。

1. 基本模式

```
using UnityEngine;
public class Singleton : MonoBehaviour {

    //不写也无妨，创建继承自 MonoBehavior 的类使不允许的
    //虽然不会报错而是产生警告，但仍不可直接 new
    //因为其作为组件来使用，继承关系如下
    //Object->Component->Behaviour->MonoBehaviour->Singleton
    private Singleton() { }
    private static Singleton _instance;

    public static Singleton GetInstance() {
        if (_instance == null) {
            Debug.Log("Create singleton...");
            _instance = GameObject.FindObjectOfType<Singleton>();
        }
        return _instance;
    }
}
```

2. 复用模式

- 可能有多个类都需要应用单例模式，它们用于处理不同的逻辑块。为每个类都写一个提供单例的创建方式显然太低效率了，那就直接写个泛型来剥离出创建单例的代码吧！

(a) 静态类

- 用来创建实例的 SingletonStatic 类：

```
using UnityEngine;

// FindObjectOfType 方法的泛型参数必须继承自 Object 类，所以这里对 T 要进行约束
```

```
public static class SingletonStatic<T> where T : MonoBehaviour {
    private static T _instance;
    public static T GetInstance() {
        if (_instance == null) {
            Debug.Log("Create " + typeof(T).ToString() + " singleton...");
            _instance = GameObject.FindObjectOfType<T>();
            if (_instance == null)
                Debug.LogError("Class of " + typeof(T).ToString() + " not found!")
        }
        return _instance;
    }
}
```

- 需要应用单例模式的两个类，SingletonClass1 类和 SingletonClass2 类：

```
using UnityEngine;
public class SingletonClass1 : MonoBehaviour {
    private SingletonClass1() { }
    public int myInt = 2;
}

using UnityEngine;
public class SingletonClass2 : MonoBehaviour {
    private SingletonClass2() { }
    public int myInt = 5;
}
```

- 用来访问单例的测试类，TestClass 类：

```
public class TestClass : MonoBehaviour {
void Awake ()  {
    SingletonClass1 s1 = SingletonStatic<SingletonClass1>.GetInstance();
    SingletonClass2 s2 = SingletonStatic<SingletonClass2>.GetInstance();
    Debug.Log(s1.myInt);
    Debug.Log(s2.myInt);
    Debug.Log(s1.myInt);
    Debug.Log(s2.myInt);
}
}
```

- 除了静态类，将这三个脚本分别指定给不同的对象，运行查看 Console 面板：
- 这里写图片描述
- 可以看到两个类的单例都实例了一次。很多人会有疑问，应用泛型会不会导致另外一个类型创建实例时会覆盖掉之前类型的实例，经过这样的测试我们发现这样的担忧完全是不必要的。

(b) 继承抽象类

- 继承抽象类的原理其实与静态类比较相似，这里直接给出父类，应用单例模式类，以及测试类的代码。
- 父类 SingletonBase 类：

```
using UnityEngine;

public abstract class SingletonBase<T> : MonoBehaviour where T : MonoBehaviour {
    private static T _instance;
    public static T GetInstance() {
        if (_instance == null) {
            Debug.Log("Create " + typeof(T).ToString() + " singleton...");
            _instance = GameObject.FindObjectOfType<T>();
            if (_instance == null)
                Debug.LogError("Class of " + typeof(T).ToString() + " not found!")
        }
        return _instance;
    }
}
```

- 应用单例模式的类 SingleClass1 类：

```
using UnityEngine;
public class SingletonClass1 : SingletonBase<SingletonClass1> {
    private SingletonClass1() { }
}

// 测试类 TestClass 类：
using UnityEngine;
public class TestClass : MonoBehaviour {
    void Awake () {
        SingletonClass1 s1 = SingletonClass1.GetInstance();
    }
}
```

3. 为什么称为伪单例

- 假设应用单例模式的类（脚本）的名称为 SingletonClass：

(a)（一）根本问题

- 无法避免脚本挂在多个物体上，因为 SingletonClass 会继承 MonoBehavior 类。虽然我们在任何时候访问 SingletonClass 对象都是同一个，但是这不代表场景中这个对象是唯一的。说白了就是当脚本挂在物体上时已经是个实例了，FindObjectOfType 方法只是去找到其中一个实例，并不是在创造独一无二。每个实例都会执行 Monobehaviour 中的 Message 方法（Start、Update 这些）。
- 总结是当你同样的脚本挂在两个物体上的时候这个脚本类的对象就不唯一了，且没有方法阻止脚本挂在物体上除非不继承 MonoBehaviour 类。那既然不需要 MonoBehaviour 类，何不写成标准 C# 中的真单例模式呢？

(b)（二）衍生问题

- 前面说到这样的伪单例只适合单场景，其实使用 Object 类的静态方法 DontDestroyOnLoad 方法可以将对象加载到内存中，只有整个程序结束的时候才会被清除。但这样做又会引发新的问题。这里做个演示，将“继承抽象类”例子中的代码修改至如下所示：

```
using UnityEngine;

public abstract class SingletonBase<T> : MonoBehaviour where T : MonoBehaviour {
    private static T _instance;

    public static T GetInstance() {
        if (_instance == null) {
            Debug.Log("Create " + typeof(T).ToString() + " singleton...");
            _instance = GameObject.FindObjectOfType<T>();
            //创建完实例后使其不会因场景切换被销毁
            Object.DontDestroyOnLoad(_instance);
            if (_instance == null)
                Debug.LogError("Class of " + typeof(T).ToString() + " not found!")
        }
        return _instance;
    }
}
```

- 新建一个场景，两个场景都添加个按钮，点击按钮能来回切换场景。这里单例的物体名称为 SingletonClass1，测试类所在物体叫 TestClass，该场景为 Scene1，新创建场景为 Scene2,。现在我们从 Scene1 运行，点击按钮切换到 Scene2，再点击按钮切换回 Scene，资源面板显示如图所示：
- 这里写图片描述
- 出现了两个实例！这是因为我们将其加载到内存中时它已经不属于场景本身了，而场景初始化的时候会创建预制的资源，这就导致了我们再次回到场景时，出现了两个 SingletonClass1。这进一步的违背单例模式的初衷。

4. 为什么 C# 中没写“复用模式”？

- 上面介绍过，Unity 中脚本挂在物体上，我们构建所谓的单例是找到这个物体，并不是创建对象的方式；而 C# 中都是用 new 关键字创建对象的形式来构造单例。假设我们构造了这样一个泛型类，来看看具体的写法：

```
namespace CSharpTest {
    public class Singleton<T> where T : new() {

        //私有化构造函数 使得外部无法构造类的实例
        private Singleton() { }

        //定义实例对象时便创建实例
        private static T _instance;

        //提供全局访问点
        public static T GetInstance() {
            if (_instance == null)
                return new T();
            return _instance;
        }
    }
}
```

- 与最开始例子给出的代码的不同之处，除了所有的类型都写成了泛型 T 以为，还有很关键的一点，我们对 T 的类型进行了约束，约束 T 类型必须含有 public 级别的构造函数。问题就在于这里。我们为了应用单例模式的类不被随意创建，会将其构造函数设为 private 级，这就造成了冲突，导致需要应用单例模式的类无法作为该泛型类的的类型参数。

1.3.4 【为什么要用单例】

- 比如我们玩游戏，游戏的目标是把三个任务都完成就可以通关。在游戏内部机制中，应该是没完成一个任务就通知管理器该任务已经完成，此时游戏管理器就是一个单例，这样当游戏管理器检测到三个任务都完成时才会通知玩家游戏通关。如果每个任务都创建了一个游戏管理器，那么这个游戏是不可能通关的——每个游戏管理器中只有一个任务被完成的记录啊！这些任务记录并没有集中到同一个游戏管理器中。此时是一定要使用单例模式的。

1.3.5 【总结】

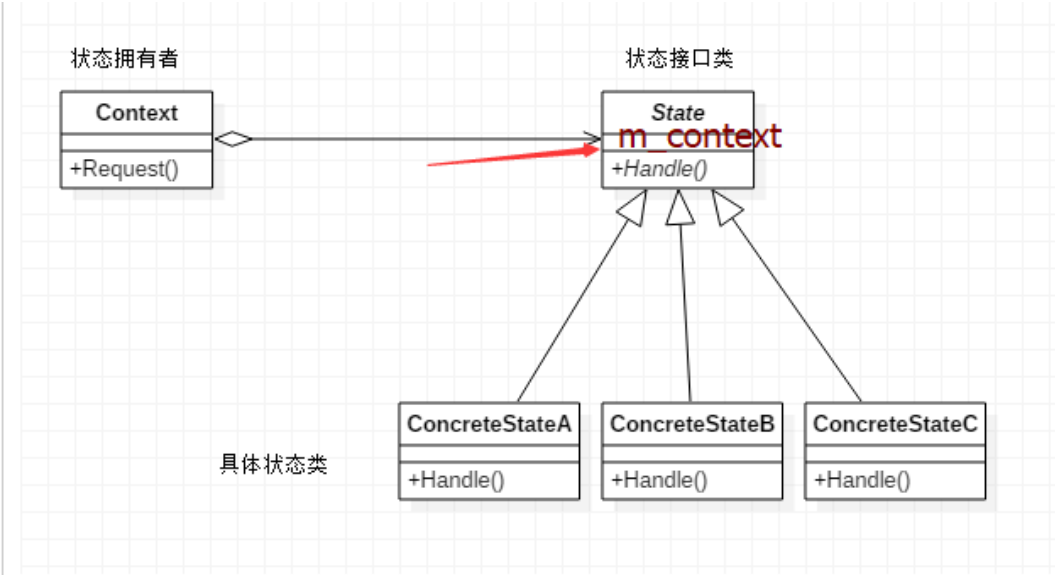
- 举了很多例子，单例的性质已经很清晰了。但现在还有的疑问可能是，既然基继承自 Monobehaviour 写的单例是伪单例，为什么还要一一列举出来呢？存在必有道理。MonoBehaviour 的确提供了极大地便利，很多开发者在 Unity 中都会用这样的伪单例形式，的确可以这样用，而且对于某些需求，这样做会极大的提高开发效率，但是要跟真正的单例模式区分开，不是说死记硬背一种设计模式，而是掌握其核心思想，更加安全高效的开发才是最重要的。

1.4 Unity 设计模式游戏完美开发的学习总结

- https://blog.csdn.net/mcaisw/article/details/74762182

1.4.1 State 模式

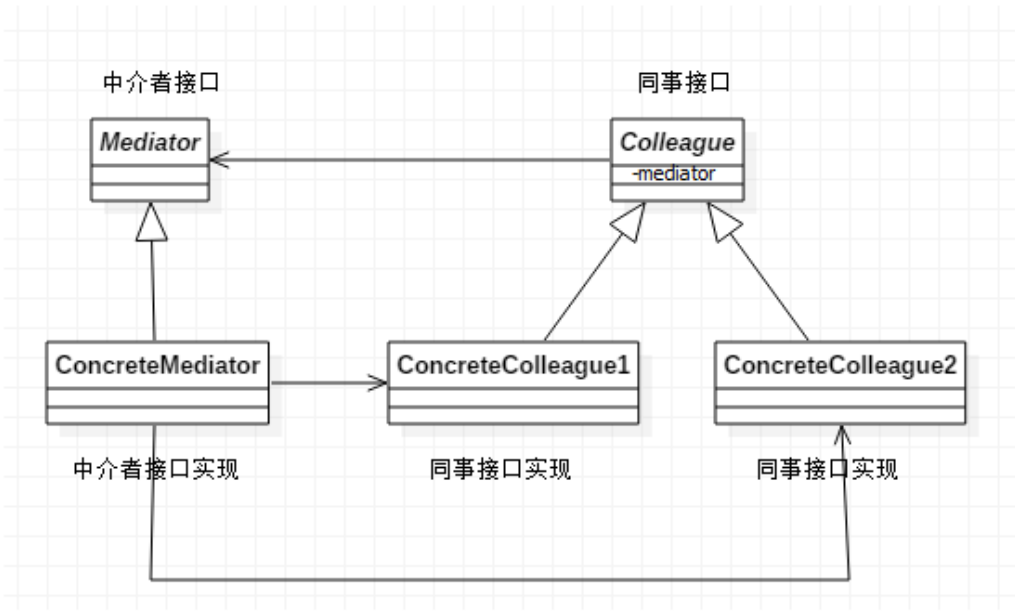
- GOF 的解释”：让一个对象的行为随着内部状态的改变而变化，而该对象也像是换了类一样”。
- 书中用 State 模式实现了场景的转换。
- 这是 State 模式的结构图。




```
// 状态接口类
public abstract class State {
    // 关联的状态拥有者
    protected Context m_Context = null;
    // 构造函数
    // <param name="theContext"> 状态拥有者 </param>
    public State(Context theContext) {
        // 构造函数接收 Context 类的对象，根据这本书的绘图习惯，State 那一栏的下面会写 m_Cont
        m_Context = theContext;
    }
    // Handle 抽象方法
    // <param name="Value"> 参数值 </param>
    public abstract void Handle(int Value);
}
```

- 我在原结构图上添加了 $m_{context}$，是因为这本书的后面，在一个类的构造函数里，接收另一个类的对象，然后给这个类的对象赋值，都会在其结构图上，在这个类的下面会写另外那个类的对象，这个例子中，就是 $m_{context}$
- 我这样区别这个问题，是因为，后面的 Mediator 中介者模式，也在一个接口类的构造函数里添加一个对象作为参数。在结构图上，Collegue 下面有-mediator。
- 注意：Context 不是一个接口类，就是说，它不是一个类的集合，后面的中介者模式，Mediator 是一个接口，先看看 Mediator 模式的结构图。

1.4.2 Mediator 模式



Mediator

- 书中原图是有 mediator 的。
- 和 state 的结构图很像，从图上看，区别就是 Mediator 是作为一个接口的，即是抽象的，它有一个子类来实现它内部的功能。
- Colleague 类，很像 State 模式里的 State 类，从结构图的连线关系来看，State 模式看起来真的是“状态模式”，State 类只是作为三个子类的接口，供客户端 Context 在这三个子类中进行挑选，切换。这里 Mediator 可不是客户端，仔细看结构图，Mediator 是个接口，因为它有实现它的子类。从 Mediator 的结构图连线来看，它真的是“中介模式”，Mediator 的子类，把其他需要沟通的类全部包含引用了，这些类的沟通都发生在 Mediator 的子类中。看看 Mediator 模式的代码：

```
// 实现 Colleague 的类 1
public class ConcreateColleague1 : Colleague {
    public ConcreateColleague1( Mediator theMediator) : base(theMediator) {}
    //执行动作
    public void Action() {
        //执行后需要通知其他 Colleague
        m_Mediator.SendMessage(this,"Colleage1 发出通知");
    }
    //Mediator 通知请求
    public override void Request(string Message) {
        Debug.Log("ConcreateColleague1.Request:"+Message);
    }
}

// 实现 Colleague 的类 2
public class ConcreateColleague2 : Colleague {
    public ConcreateColleague2( Mediator theMediator) : base(theMediator) {}
    // 執行動作
    public void Action() {
        //执行后需要通知其他 Colleague
        m_Mediator.SendMessage(this,"Colleage2 发出通知");
    }
    //Mediator 通知请求
    public override void Request(string Message) {
        Debug.Log("ConcreateColleague2.Request:"+Message);
```

```
    }
}

// 实现 Mediator 接口，并集合管理 Colleague 对象
public class ConcreteMediator : Mediator {
    ConcreateColleague1 m_Colleague1 = null;
    ConcreateColleague2 m_Colleague2 = null;

    public void SetColleageu1( ConcreateColleague1 theColleague ) {
        m_Colleague1 = theColleague;
    }

    public void SetColleageu2( ConcreateColleague2 theColleague ) {
        m_Colleague2 = theColleague;
    }

    // 收到来自 Colleague 的通知请求
    public override void SendMessage(Colleague theColleague,string Message) {
        // 收到 Colleague1 通知 Colleague2
        if( m_Colleague1 == theColleague)
        m_Colleague2.Request( Message);

        // 收到 Colleague2 通知 Colleague1
        if( m_Colleague2 == theColleague)
        m_Colleague1.Request( Message);
    }
}
```

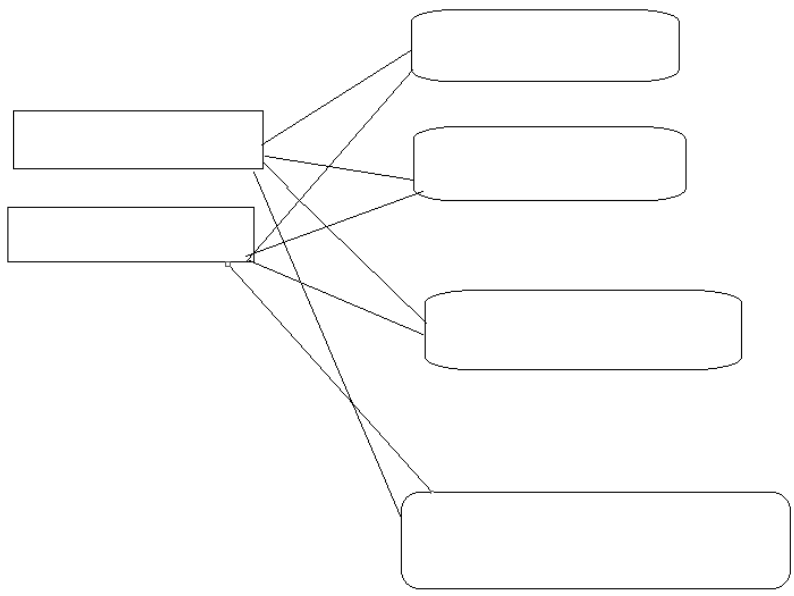
- 看代码，不难发现，Colleague1 和 Colleague2 这两个类之间并没有包含引用，它们的沟通和交流都是在 ConcreteMediator 类中进行的，这里和 State 模式又有区别了，看看 State 模式的代码：

```
// 具体状态 A
public class ConcreteStateA : State {
    // 构造函数
    // <param name="theContext"> 状态拥有者 </param>
    public ConcreteStateA(Context theContext) : base(theContext) {
    }
    // Handle
    // <param name="Value"> 参数值 </param>
    public override void Handle(int Value) {
        Debug.Log("ConcreteStateA.Handle");
        //根据条件切换到状态 B
        if (Value > 10) {
            m_Context.SetState(new ConcreteStateB(m_Context));
        }
    }
}
public class ConcreteStateB:State {
    // 构造函数
    // <param name="theContext"></param>
    public ConcreteStateB(Context theContext) : base(theContext) {
    }
    // Handle
    // <param name="Value"> 参数值 </param>
    public override void Handle(int Value) {
        Debug.Log("ConcreteStateB.Handle");
        //根据条件切换到状态 C
        if (Value > 20) {
            m_Context.SetState(new ConcreteStateC(m_Context));
```

```
        }
    }
}
public class ConcreteStateC:State {
    // 构造函数
    // <param name="theContext"></param>
    public ConcreteStateC(Context theContext) : base(theContext) {
    }
    // Handle
    // <param name="Value"> 参数值 </param>
    public override void Handle(int Value) {
        Debug.Log("ConcreteStateC.Handle");
        //根据条件切换到状态 A
        if (Value > 30) {
            m_Context.SetState(new ConcreteStateA(m_Context));
        }
    }
}
```

- 这里的区别也是由每个模式的特点来决定的，在 State 模式里，客户端的方法和操作，贯穿着 State 类的三个子类，$m_{Context}$.SetState(); 让我们感受下。。。。。。这好像是线性贯穿的。
- 我好像感受到什么了，一个东西，或者叫 object，对外要呈现不同的状态，它要贯穿游走在这几个状态，线性贯穿有木有，可以用状态模式来设计。
- Mediator 中介者模式，比较好理解，把需要沟通的类，用 Mediator 接口的子类（实现类）全部包含，让它们在里面沟通，它们自己的类里并不互相包含引用。而 Mediator 的对象是在这些需要沟通的子类的接口类在构造函数里就生成赋值的，也就是说，Mediator 的对象也是贯穿在这些个需要沟通的子类里的，中介者嘛，必须和每个需要沟通的类打交道。
- 总结下，贯穿，游离，沟通，切换，有这些动作的对象，都会在它要植入的对象的类的构造函数中进行生成和引用。

1.4.3 桥接模式



- 左边的两个方框代表两个类，右面的带圆角的方框代表另外的四个类，左边的两个类都要用到右边那四个类的方法，就是说，左边的都要和右边的进行连线，左边和右边的那一列彼此不连线。这样的连线方式，就可以用桥接模式。

- 把左右两列的类分别抽象化成两个接口，比如玩家类这个接口，有敌人类和士兵类，炸弹，枪，火箭筒各类武器抽象成一个武器类

```
// 武器类群组（武器接口）
public abstract class IWeapon {
    // 武器的拥有者
    protected ICharacter m_WeaponOwner = null;
    // 攻击目标
    public abstract void Fire(ICharacter theTarget);
    // 显示射击特效
    protected void ShowShootEffect() {
    }
    // 显示子弹特效
    protected void ShowBulletEffect(Vector3 TargetPosition,float LineWidth,float DisplayTi
    }
    // 播放音效
    protected void ShowSoundEffect(string ClipName) {
    }
}

// 角色接口
public abstract class ICharacter {
    // ...
    // 使用的武器
    private IWeapon m_Weapon = null;
    protected void WeaponAttackTarget(ICharacter Target) {
        m_Weapon.Fire(Target);
    }
    // 攻击目标
    public abstract void Attack(ICharacter Target);
    // 被其他角色攻击
    public abstract void UnderAttack(ICharacter Attacker);
    // ...
}
```

- 和 state 模式，中介者模式不同，桥接模式，武器和玩家这两个接口之间互相包含引用，彼此的构造函数并没有带对方接口类的对象作为参数。这是我看出来状态模式，中介者模式，桥接模式之间的区别。

1.4.4 再总结下

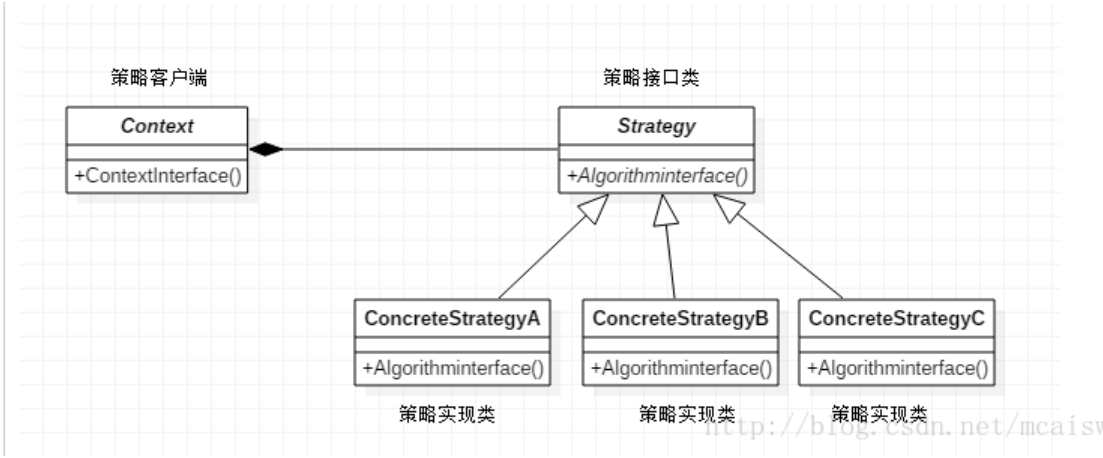
- State 模式，一个客户端，多个状态类，把状态类抽象化成一个接口，供客户端进行调用，此时，状态的接口在其构造函数内，要把客户端的对象作为参数，并对客户端对象进行赋值。而多个状态类的内部有实现状态切换的方法，供客户端进行调用。

- 中介者模式，把需要沟通的多个对象抽象成一个接口，中介者也是一个需要实现的接口类，中介者在多个需要沟通的对象实例化时就生成，就是说，多个对象的接口的构造函数是以中介者类为参数的，然后给中介者对象赋值。在中介者的实现类内部，包含所有需要沟通的类，在中介者的实现类内部，让这些需要沟通的类自己们互相交流。

- 桥接模式，最简单，抽象出两个大接口，比如图形渲染器例子，在一个接口里只有一个绘制图形方法，在需要调用绘制方法的类里定义一个 set 方法，设置使用哪一个绘制引擎进行绘制。

1.4.5 策略模式

- 策略模式，比较像桥接模式，只不过策略模式是客户端根据需求来调用不同的方法。



- 桥接模式 Context 那一栏里是个抽象接口。

1.5 Unity 中常用的几种设计模式

- https://blog.csdn.net/swj524152416/article/details/52931422
- 23 种设计模式，实在是太多了，而且其中有一些看着还貌似差不多，让人很费解，好不容易理解了每一种设计模式的含义，并且看了一堆伪代码之后，高高兴兴的合上了书本去玩几把 LOL，赢了几把之后脑袋里关于那 23 种设计模式的概念就剩下 80% 了，然后接下来的每日工作中，基本写代码的时候也用不到啊！老板催着让你做功能，你就还哪里记得去使用设计模式啊，就开始乱写吧，日复一日，23 种设计模式基本就和你拜拜了，再见了。
- 其实呢，在游戏开发中，我们能够在 Unity 中看见的，和我们经常使用的也就是那么几种，在其他软件设计中，同样也就是经常用这么几种，那些“备忘录”模式，“责任链”模式等等，基本上不用，下面我们就说说我们常用的这几种吧：“单例模式”，“观察者模式”，“迭代器模式”，“访问者模式”。（顺便插一句，那些各种工厂模式，我就不多说了，都是很容易理解的。）

1.5.1 单例模式

- 概念很简单，保证一个类仅有一个实例，并提供一个访问它的全局访问点。我就提供两段代码就好了，在游戏当中，有两种类，一种是不继承 MonoBehavior，另外一种是继承它的，首先看不继承的，

```
Class Singleton {
    Static MySingleton;                          // 单件对象，全局唯一的。

    Static Instance() {
        if(MySingleton == null)
            MySingleton = new MySingleton();
        return MySingleton;
    }       // 对外暴露接口
}

// 下面来看继承自 MonoBehavior 的类，
Class Singleton : MonoBehavior {
    Static MySingleton;
    Static Instance() {
        return MySingleton;
    }
    void Awake() {
        MySingleton = this;
    }
}
```

- 直接在游戏开发中这么使用就可以了。

1.5.2 观察者模式

- 概念：它将对象与对象之间创建一种依赖关系，当其中一个对象发生变化时，它会将这个变化通知给与其创建关系的对象中，实现自动化的通知更新。
- 在游戏开发中，比如 UI 上有一个下拉的 List，我选择了其中的每一项，都会导致 UI 界面的变化，比如我选择“强化”，对应出现强化装备的界面；我选择“镶嵌”，就会出现镶嵌的界面。
- 伪代码如下：

```
Class Subject {
    // 对本目标绑定一个观察者 Attach( Observer );
    // 解除一个观察者的绑定   DeleteAttach( Observer );
    // 本目标发生改变了，通知所有的观察者，但没有传递改动了什么
    Notity() {
        For ( ...遍历整个ObserverList ...) { pObserver ->Update(); }
    }
    // 对观察者暴露的接口，让观察者可获得本类有什么变动       GetState();
}
// 观察者/监听者类
Class Observer {
    // 暴露给对象目标类的函数，当监听的对象发生了变动，则它会调用本函数通知观察者
    Void Update () {
        pSubject ->GetState();
// 获取监听对象发生了什么变化
        TODO: DisposeFun();
// 根据状态不同，给予不同的处理
    }
}
```

- 这个很好理解了吧！

1.5.3 迭代器模式

- 我们就拿 C# 中的迭代器为例直接说明即可，既能了解了迭代器模式的概念，有了解了 C# 中迭代器是如何实现的。
- 迭代器模式是设计模式中行为模式 (behavioral pattern) 的一个例子，他是一种简化对象间通讯的模式，也是一种非常容易理解和使用的模式。简单来说，迭代器模式使得你能够获取到序列中的所有元素而不用关心是其类型是 array，list，linked list 或者是其他什么序列结构。这一点使得能够非常高效的构建数据处理通道 (data pipeline)–即数据能够进入处理通道，进行一系列的变换，或者过滤，然后得到结果。事实上，这正是 LINQ 的核心模式。
- 在.NET 中，迭代器模式被 IEnumerator 和 IEnumerable 及其对应的泛型接口所封装。如果一个类实现了 IEnumerable 接口，那么就能够被迭代；调用 GetEnumerator 方法将返回 IEnumerator 接口的实现，它就是迭代器本身。迭代器类似数据库中的游标，他是数据序列中的一个位置记录。迭代器只能向前移动，同一数据序列中可以有多个迭代器同时对数据进行操作。
- 在 C#1 中已经内建了对迭代器的支持，那就是 foreach 语句。使得能够进行比 for 循环语句更直接和简单的对集合的迭代，编译器会将 foreach 编译来调用 GetEnumerator 和 MoveNext 方法以及 Current 属性，如果对象实现了 IDisposable 接口，在迭代完成之后会释放迭代器。但是在 C#1 中，实现一个迭代器是相对来说有点繁琐的操作。C#2 使得这一工作变得大为简单，节省了实现迭代器的不少工作。

```
public System.Collections.IEnumerator GetEnumerator() {
    for (int i = 0; i < 10; i++) {
        yield return i;
    }
}
static void Main() {
    ListClass listClass1 = new ListClass();
    foreach (int i in listClass1) {
        System.Console.Write(i + " ");
```

```
    }    // Output: 0 1 2 3 4 5 6 7 8 9}
}
```

1.5.4 访问者模式

- 当我们希望对一个结构对象添加一个功能时，我们能够在不影响结构的前提下，定义一个新的对其元素的操作。
- 例如场景管理器中管理的场景节点，是非常繁多的，而且种类不一，例如有 Ogre 中的 Root, Irrchit 中就把摄象机，灯光，Mesh，公告版，声音都做为一种场景节点，每个节点类型是不同的，虽然大家都有共通的 Paint(),Hide() 等方法，但方法的实现形式是不同的，当我们外界调用时需要统一接口，那么我们很可能需要需要这样的代码

```
Hide (Object) {
    if (Object == Mesh) HideMesh();
    if (Object == Light) HideLight();
    // ...
}
```

- 此时若我们需要增加一个 Object 新的类型对象，我们就不得不对该函数进行修正。而我们可以这样做，让 Mesh,Light 他们都继承于 Object, 他们都实现一个函数 Hide(), 那么就变成

```
Mesh::Hide( Visitor ) {
    Visitor.Hide(Mesh);
}
Light::Hide(Visitor ) {
    Visitor.Hide(Light);
}
```

- 意思就是说，Mesh 的隐藏可能涉及到 3 个步骤，Light 的隐藏可能涉及到 10 个步骤，这样就可以在每个自己的 Visitor 中去实现每个元素的隐藏功能，这样就把很多跟元素类本身没用的代码转移到了 Visitor 中去了。
- 每个元素类可以对应于一个或者多个 Visitor 类。比如我们去银行柜台办业务，一般情况下会开几个个人业务柜台的，你去其中任何一个柜台办理都是可以的。我们的访问者模式可以很好付诸在这个场景中：对于银行柜台来说，他们是不用变化的，就是说今天和明天提供个人业务的柜台是不需要有变化的。而我们作为访问者，今天来银行可能是取消费流水，明天来银行可能是去办理手机银行业务，这些是我们访问者的操作，一直是在变化的。
- 伪代码如下：

```
//  访问者基类
Class Visitor {
    Virtual VisitElement( A ){ ... };
^^I// 访问的每个对象都要写这样一个方法
    Virtual VisitElement( B ){ ... };
}

// 访问者实例 A
Class VisitorA {
    VisitElement( A ){ ... };
    // 实际的处理函数
    VisitElement( B ){ ... };
    // 实际的处理函数
}

// 访问者实例 B
Class VisitorB {
    VisitElement( A ){ ... };
^^I// 实际的处理函数
    VisitElement( B ){ ... };
```

```
^^I// 实际的处理函数
}

// 被访问者基类
Class Element {
    Virtual Accept( Visitor );
^^I// 接受访问者
}

// 被访问者实例 A
Class ElementA {
    Accecpt( Visitor v ){ v-> VisitElement(this); };
^^I// 调用注册到访问者中的处理函数
}

// 被访问者实例 B
Class ElementB {
    Accecpt( Visitor v ){ v-> VisitElement(this); };
    // 调用注册到访问者中的处理函数
}
```

1.6 设计模式之观察模式

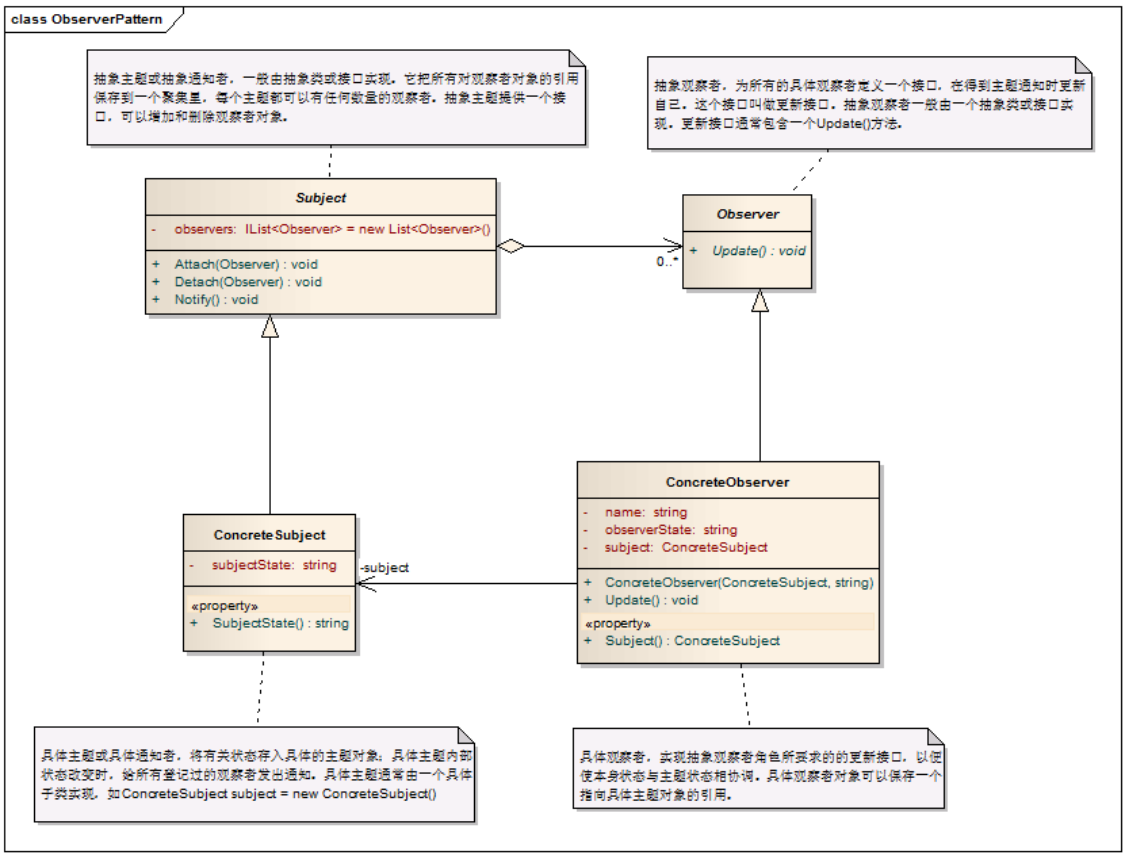
- http://www.newbieol.com/information/1766.html
- 有时被称作发布/订阅模式，观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这就是观察模式。那么今天的 Unity3d 教程我们就来讲讲它。
- 在 Unity3d 中，有时被称作发布/订阅模式，观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。下面我们的 Unity3d 教程就来讲讲它：
- 解决的问题
- 将一个系统分割成一个一些类相互协作的类有一个不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。观察者就是解决这类的耦合关系的。

1.6.1 模式中的角色

- 抽象主题 (Subject)：它把所有观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。
- 具体主题 (ConcreteSubject)：将有关状态存入具体观察者对象; 在具体主题内部状态改变时，给所有登记过的观察者发出通知。
- 抽象观察者 (Observer)：为所有的具体观察者定义一个接口，在得到主题通知时更新自己。
- 具体观察者 (ConcreteObserver)：实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题状态协调。

1.6.2 模式解读

- 观察者模式的类图



- Unity3d 教程：设计模式之观察模式
- 观察者模式的代码

```
// 抽象主题类
public abstract class Subject {
    private IList observers = new List();
^^I// 增加观察者
    public void Attach(Observer observer) {
        observers.Add(observer);
    }
^^I// 移除观察者
    public void Detach(Observer observer) {
        observers.Remove(observer);
    }
^^I// 向观察者（们）发出通知
    public void Notify() {
        foreach (Observer o in observers) {
            o.Update();
        }
    }
}

// 抽象观察者类，为所有具体观察者定义一个接口，在得到通知时更新自己
public abstract class Observer {
    public abstract void Update();
}

// 具体观察者或具体通知者，将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记
public class ConcreteSubject : Subject {
    private string subjectState;
^^I// 具体观察者的状态
```

```
    public string SubjectState {
        get { return subjectState; }
        set { subjectState = value; }
    }
}

// 具体观察者，实现抽象观察者角色所要求的更新接口，已是本身状态与主题状态相协调
public class ConcreteObserver : Observer {
    private string observerState;
    private string name;
    private ConcreteSubject subject;

^^I// 具体观察者用一个具体主题来实现
    public ConcreteSubject Subject {
        get { return subject; }
        set { subject = value; }
    }
    public ConcreteObserver(ConcreteSubject subject, string name) {
        this.subject = subject;
        this.name = name;
    }
^^I// 实现抽象观察者中的更新操作
    public override void Update() {
        observerState = subject.SubjectState;
        Console.WriteLine("The observer's state of {0} is {1}", name, observerState);
    }
}
```

- 客户端代码

```
class Program {
    static void Main(string[] args) {
^^I// 具体主题角色通常用具体自来来实现
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject, "Observer A"));
        subject.Attach(new ConcreteObserver(subject, "Observer B"));
        subject.Attach(new ConcreteObserver(subject, "Observer C"));
        subject.SubjectState = "Ready";
        subject.Notify();
        Console.Read();
    }
}
```

- 运行结果
- Unity3d 教程：设计模式之观察模式

1.6.3 模式总结

1. 优点 观察者模式解除了主题和具体观察者的耦合，让耦合的双方都依赖于抽象，而不是依赖具体。从而使得各自的变化都不会影响另一边的变化。
2. 缺点
   - 依赖关系并未完全解除，抽象通知者依旧依赖抽象的观察者。
3. 适用场景
   - 当一个对象的改变需要给变其它对象时，而且它不知道具体有多少个对象有待改变时。
   - 一个抽象某型有两个方面，当其中一个方面依赖于另一个方面，这时用观察者模式可以将这两者封装在独立的对象中使它们各自独立地改变和复用。
   -

1.7 C# 设计模式学习笔记之建造者模式（Builder）

- http://www.58kaifa.com/article/177
- 建造者模式（Builder）概述：
- 在构造一个对象的时候，往往有很多复杂的过程和次序，例如建造一个机器人，那要先造头，还是要先造身子，这就关乎到一个制造工序的问题。其实建造者模式 (Builder) 和工厂模式很接近的，但是建造模式提供了一个更加细粒度的对象的建造过程。
- 定义：
- 建造者模式 (Builder) 将复杂的结构与其表示分离，使得同样的构建过程可以创建不同的表示。
- 原型模式应用：
- 在软件系统中，有时候面临一个复杂对象的创建工作，该对象通常由各个部分子对象用一定的算法构成，或者按一定的步骤组合而成；这些算法和步骤是稳定的，而构成这个对象的子对象却经常由于需求的变化而不断变化。
- 假设要组装一台电脑，它的组装过程基本是不变的，都可以由主板、CPU、内存等按照某个稳定方式组合而成。然而主板、cpu、内存等零件本身是可能多变的。将内存等这种易变的零件与电脑的其他部件分离，实现解耦合，则可以轻松实现电脑不断升级。
- 建造模式结构图：
- 建造模式参与者：
- Builder（抽象建造者）：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。
- ConcreteBuilder（建造者）：实现 Builder 接口，针对不同的商业逻辑，具体化负责对象的各部分的创建，在建造过程完成后，提供产品的实例。
- Director（导演类）：调用具体建造者来创建复杂对象的各个部分，在导演类中不急具体产品的信息，只复杂保证对象各个部分完整创建或按某种顺序创建。
- Product（产品类）：
- 表示被构造的复杂对象。ConcreteBuilder 创建该产品的内部表示并定义它的装配过程
- 包含定义组成部件的类，包括将这些部件装配成最终产品的接口
- 在建造者模式中，Director 规定了创建一个对象所需要的步骤和次序，Builder 则提供了一些列完成这些步骤的方法，ConcreteBuilder 给出了这些方法的具体实现，是对象的直接创建者。
- 建造者模式结构实现：

```
// 产品类
public class Product {
    private List<string> _parts = new List<string>();
    public void Add(string part) {
        _parts.Add(part);
    }
    public void Show() {
        Console.WriteLine("Product Parts");
        foreach (string part in _parts) {
            Console.WriteLine(part);
        }
    }
}

// 抽象建造者类
public abstract class Builder {
    public abstract void BuildPartA();
```

```
    public abstract void BuildPartB();
    public abstract Product GetResult();
}
// 建造者 1
public class ConcreteBuilder1 : Builder {
    private Product _product = new Product();
    public override void BuildPartA() {
        _product.Add("PartA");
    }
    public override void BuildPartB() {
        _product.Add("PartB");
    }
    public override Product GetResult() {
        return _product;
    }
}
// 建造者 2
public class ConcreteBuilder2 : Builder {
    private Product _product = new Product();
    public override void BuildPartA() {
        _product.Add("Partx");
    }
    public override void BuildPartB() {
        _product.Add("PartY");
    }
    public override Product GetResult() {
        return _product;
    }

}
// 导演类
public class Director {
    public void Construct(Builder builder) {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}
// 客户端
class Client {
    static void Main(string[] args) {
        Director director = new Director();
        Builder b1 = new ConcreteBuilder1();
        Builder b2 = new ConcreteBuilder2();

        director.Construct(b1);
        Product p1 = b1.GetResult();
        p1.Show();

        director.Construct(b2);
        Product p2 = b2.GetResult();
        p2.Show();

    }
}
```

- 建造者模式实例
- 建造小人，要求：小人必须包括，头，身体，手和脚。现在系统要包括的分为胖子和瘦子：

```
// 产品类
```

```
class Product {
    private List<string> _part = new List<string>();
    public void Add(string part) {
        _part.Add(part);
    }
    public void Show() {
        foreach (string part in _part) {
            Console.WriteLine(part);
        }
    }
}
// 抽象建造者
public abstract class Builder {
    public abstract void BuildHead();
    public abstract void BuildBody();
    public abstract void BuildHand();
    public abstract void BuildFeet();
    public abstract Product GetResult();
}
// 胖子建造
public class FatPBuilder : Builder {
    private Product _product;
    public override void BuildHead() {
        _product.Add(" 胖子头");
    }
    public override void BuildBody() {
        _product.Add(" 胖子身体");
    }
    public override void BuildHand() {
        _product.Add(" 胖子手");
    }
    public override void BuildFeet() {
        _product.Add(" 胖子脚");
    }
    public override Product GetResult() {
        return _product;
    }
}
// 瘦子建造
public class ThinPBuilder : Builder {
    private Product _product;
    public override void BuildHead() {
        _product.Add(" 瘦子头");
    }
    public override void BuildBody() {
        _product.Add(" 瘦子身体");
    }
    public override void BuildHand() {
        _product.Add(" 瘦子手");
    }
    public override void BuildFeet() {
        _product.Add(" 瘦子脚");
    }
    public override Product GetResult() {
        return _product;
    }
}
// 导演类
```

```
public class Director {
    public void Construct(Builder builder) {
        builder.BuildHead();
        builder.BuildBody();
        builder.BuildHand();
        builder.BuildFeet();
    }
}
// 客户端类
public class Client {
    static void Main(string[] args) {
        Director _director = new Director();
        Builder b1 = new FatPBuilder();
        Builder b2 = new ThinPBuilder();
        _director.Construct(b1);
        Product p1 = b1.GetResult();
        p1.Show();
    }
}
```

- 汽车生产

```
// 车辆产品类
public class Vehicle {
    private string _vehicleType;
    private Dictionary<string, string> _parts = new Dictionary<string, string>();

    public Vehicle(string vehicleType) {
        this._vehicleType = vehicleType;
    }

    public string this[string key] {
        get { return _parts[key]; }
        set { _parts[key] = value; }
    }

    public void Show() {
        Console.WriteLine("\n----------------------");
        Console.WriteLine("Vehicle Type:{}",_vehicleType);
        Console.WriteLine("Frame:{0}",_parts["frame"]);
        Console.WriteLine("Engine:{0}", _parts["engine"]);
        Console.WriteLine("#Wheels{0}",_parts["wheels"]);
        Console.WriteLine("#Doors:{0}",_parts["doors"]);
    }
}

// 汽车制作抽象类
public abstract class VehicleBuilder {
    protected Vehicle vehicle;
    public Vehicle Vechicle {
        get { return Vechicle; }
    }
    public abstract void BuildFrame();
    public abstract void BuildEngine();
    public abstract void BuildWheels();
    public abstract void BuildDoors();
}

// 摩托车制作类
```

```
38 public class MotorCycleBuilder : VehicleBuilder {
39     public MotorCycleBuilder() {
40         vehicle = new Vehicle("MotorCycle");
41     }
42     public override void BuildFrame() {
43         vehicle["Frame"] = "MotorCycle Frame";
44     }
45     public override void BuildEngine() {
46         vehicle["engine"] = "500 cc";
47     }
48     public override void BuildWheels() {
49         vehicle["wheels"] = "2";
50     }
51     public override void BuildDoors() {
52         vehicle["doors"] = "0";
53     }
54 }
55
56 // 踏板车类制作类
57 public class ScooterBuilder : VehicleBuilder {
58     public ScooterBuilder() {
59         vehicle = new Vehicle("Scooter");
60     }
61     public override void BuildFrame() {
62         vehicle["Frame"] = "Scooter Frame";
63     }
64     public override void BuildEngine() {
65         vehicle["engine"] = "50 cc";
66     }
67     public override void BuildWheels() {
68         vehicle["wheels"] = "2";
69     }
70     public override void BuildDoors() {
71         vehicle["doors"] = "0";
72     }
73 }
74
75 // 商店类
76 public class Shop {
77     public void Construct(VehicleBuilder vb) {
78         vb.BuildFrame();
79         vb.BuildEngine();
80         vb.BuildWheels();
81         vb.BuildDoors();
82     }
83 }
84
85 // 客户端类
86 class Client {
87     static void Main(string[] args) {
88         VehicleBuilder builder;
89         Shop shop = new Shop();
90
91         builder = new ScooterBuilder();
92         shop.Construct(builder);
93         builder.Vechicle.Show();
94
95         builder = new MotorCycleBuilder();
```

```
96          shop.Construct(builder);
97          builder.Vechicle.Show();
98      }
99  }
```

建造者模式应用分析

- 需要生成的产品对象有复杂的内部结构
- 需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序
- 在对象创建过程会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不易得到
- 建造者模式特点：
- 建造者模式的使用使得产品的内部表对象可以独立地变化。使用建造者模式可以使客户不必知道产品内部组成的细节
- 每一个 builder 都相对独立，而与其他 builder 无关
- 可使对构造过程更加精细控制
- 将构建代码和表示代码分开
- 建造者模式的缺点在于难于应付分步步骤构造算法的需求变动。
- ConcreteBuilder），用来实现抽象制造类中的操作，并进行现象内容的操作。（可定义多个，继承抽象制造者（Builder）类）
- Director），用来实现操作的流程。
- ConcreteBuilder），在 ConcreteBuilder 中我们根据 Builder 里定义操作来进行组装，在这里我们可以在操作中定义我们 CPU 使用什么型号，主板用什么型号，内存用的时多打内存等等。

1.8 C# 设计模式学习笔记之工厂设计模式

- http://www.58kaifa.com/article/171
- 模式概述：
- 工厂方法（Factory Method）模式又成为工厂模式，属于类的创建型模式。在工厂方法模式中，父类负责定义创建对象的公共接口，子类负责生产具体的对象，这样做的目的是将类的实例化操作延迟到子类中完成，即由子类决定究竟应该实例哪个类。
- 定义：
- 工厂方法模式定义一个永远创建对象的接口，让子类决定实例化哪一个类。工厂方法模式是以一个类的实例化延迟到其子类。
- Factory Method 模式用于在不指定待创建对象的具体类的情况下创建对象。
- Factory Method 模式的主要意图是隐藏对象创建的负责性。Client 通常不指定要创建的具体类，Client 将面向接口或抽象类进行编码，让 Factory 类负责创建具体的类型。通常 Factory 类有一个返回抽象类或接口的静态方法。Client 通常提供某种信息让 Factory 类使用提供的信息来确定创建并返回哪个子类。
- 将创建子类的责任抽象出来的好处是允许 Client 完成无需考虑依赖类是如何创建。这遵守依赖倒置原则(DIP)。Factory Method 模式另外一个好处是把负责对象创建的代码集中起来，如果需要修改对象生产方式，可以轻松定位并跟新，而不会影响到依赖它的代码。
- 在面向对象编程中，一般方法是用一个 new 操作符产生一个对象的实例。但是在一些情况下，用 new 操作符直接生产对象会带来一些问题。首先，要使用 new 运算符创建一个对象必须清楚所要创建的对象的类信息，包括类名、构造函数等，而又时并不现实。其次许多类型的对象创建需要一系列的步骤，可能需要计算或取得对系那个的初始设置，选择生产那个对象实例，或在需要的对象之前必须生产一些辅助功能的对象。在这些情况下，新对象的创建就是一个过程，而不是一个简单的操作。
- 工厂方法模式结构：

- 工厂模式参与者：
- Product: 抽象的产品角色，定义工厂方法所创建的对象接口
- ConcreteProduct: 具体 Product 角色，实现 Product 接口
- Factory：抽象的工厂角色，声明工厂方法，返回一个 Product 类型的对象
- Factory 可以定义一个工厂方法的默认实现，返回一个默认的 ConcreteProduct 对象。可以调用工厂方法创建一个 Product 对象。
- ConcreteFactory: 具体的工厂角色，创建具体 Product 的子工厂，重写工厂方法以返回一个 ConcreteProduct 实例。
- 工厂方法模式结构实现：

```
// 工厂模式结构实现
// 抽象产品类
public class Product {
}
// 具体产品类 A
public class ConcreteProductA : Product {
}
// 具体产品类 B
public class ConcreteProductB : Product {
}
// 抽象工厂类
public abstract class Factory {
    public abstract Product CreateProduct();
}
// 具体工厂 A
public class ConcreteFactoryA : Factory {
    public override Product CreateProduct() {
        return new ConcreteProductA();
    }
}
// 具体工厂 B
public class ConcreteFactoryB : Factory {
    public override Product CreateProduct() {
        return new ConcreteProductB();
        throw new NotImplementedException();
    }
}
///客户端类
public class Client {
    static void Main(string[] args) {
        Factory[] factories = new Factory[2];
        factories[0] = new ConcreteFactoryA();
        factories[1] = new ConcreteFactoryB();

        foreach (Factory factory in factories) {
            Product product = factory.CreateProduct();
            Console.WriteLine("Created{0}",product.GetType().Name);
        }
    }
}
//  工厂模式应用 1: 扩展刷卡处理
//  在简单工厂模式应用中我们谢了刷卡处理的应用，但是突然公司要添加新的卡种，
//  就可以很使用工厂模式来做，只要增加卡的处理类和生产卡处理类的工厂。

//  抽象产品类
```

```
abstract class BankCardHandle {
    public abstract void HandleProcess();
}
// 具体产品类 VISA，继承抽象产品类
class VisaHandle : BankCardHandle {
    public override void HandleProcess() {
        Console.WriteLine("Visa 卡处理中..");
    }
}
// 具体产品类 Master，继承抽象产品类
class MasterCardHandle : BankCardHandle {
    public override void HandleProcess() {
        Console.WriteLine("Master 卡处理中..");
    }
}
// 抽象工厂类
public abstract class HandleFactory {
    public abstract BankCardHandle CreateBankCardHandle();
}
// 具体工厂类，继承抽象工厂类
public class VisaFactory:HandleFactory {
    public override BankCardHandle CreateBackCardHandle() {
        return new VisaHandle();
    }
}
// 具体工厂类，继承抽象工厂类
public class MasterFactory:HandleFactory {
    public override BankCardHandle CreateBackCarHandle() {
        return new BankCardHandle();
    }
}
// 客户端调用
class Client {
    public static void Main(string[] args) {
        //实例化工厂
        HandleFactory visaFacotry = new VisaFactory();
        HandleFactory masterFactory = new MasterFactory();
        //创建卡类
        BankCardHandle vf = visaFacotry.CreateBackCarHandle();
        BankCardHandle mf = masterFactory.CreateBackCarHandle();
        //新添加的卡种
        HandleFactory unionFactory = new UnionPayCardFactory();
        BankCardHandle nf = unionFactory.CreateBackCarHandle();
    }
}
// 新添加的卡的特性
public class UnionPayCardHandle : BankCardHandle {
    public override void HandleProcess() {
        Console.WriteLine(" 银联卡处理中..");
    }
}
// 新添加的卡的工厂
public class UnionPayCardFactory : HandleFactory {
    public override BankCardHandle CreateBackCarHandle() {
        return new UnionPayCardHandle();
    }
}
```

- 工厂模式应用：

- 扩展刷卡应用

```
// KFC 生产:
// 应用实例: KFC 生产模式
///抽象产品类
public abstract class KFCFood {
    public abstract void Display();
}
// 具体产品类 鸡腿
public class Chicken : KFCFood {
    public override void Display() {
        Console.WriteLine(" 鸡腿 +1");
    }
}
// 具体产品类鸡翅
public class Wings : KFCFood {
    public override void Display() {
        Console.WriteLine(" 鸡翅 +1");
    }
}
// 抽象工厂类
public abstract/interface class KFCFactory {
    public abstract KFCFood CreateFood();

}
// 具体工厂类 鸡腿工厂
public class ChikenFactory:KFCFactory {
    public override KFCFood CreateFood() {
        return new Chicken();
    }
}
// 具体工厂类 鸡翅工厂
public class WingsFactory:KFCFactory {
    public override KFCFood CreateFood() {
        return new Wings();
    }
}
// 客户端类
public class Client {
    public static void Main(string[] args) {
        //创建鸡腿和鸡翅工厂
        KFCFactory chichen = new KFCFactory();
        KFCFactory wings = new KFCFactory();
        //生产鸡腿
        KFCFood food1  = chichen.CreateFood();
        food1.Display();

        KFCFood food2  = chichen.CreateFood();
        food1.Display();
        //生产鸡翅
        KFCFood food3 = wings.CreateFood();
        food3.Display();
    }
}
```

- 工厂模式方法应用分析:
- 1、工厂模式使用情形:
- 当一个类不知道它所必须创建的对象的类信息的时候

- 当一个类希望由它来指定它所创建的对象的时候
- 当类将创建对象的职责委托给多个辅助子类中的某一个，希望将哪一个辅助之类时代理者这以信息局部化的时候
- 2、工厂模式特点
- 使用工厂方法在一个类的内部创建对象通常比直接创建对象更灵活
- 工厂方法模式通过面向对象的手法，将所要创建的具体对象的创建工作延迟到子类，从而提供了一种扩展的策略，较好的解决了紧耦合的关系
- 工厂方法模式遵守依赖倒置原则 (DIP);
- 总结:
- 相对于简单工厂模式而言，工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体工厂类上。在工厂模式中，子工厂与产品往往具有平行的的等级结构，他们之间一一对应。
- 就上一节玩具工厂来说，我们知道有一台机器可以设定多个模式，那么可以多买几台机器，每台机器的设定不一样，比如，这一是生产小熊用的，另一台是生产汽车用的，当我们需要那种玩具的时候就用相对的机器进行生产。可以工厂模式就是多个简单工厂模式的综合。