

Command 模式

Design Pattern - Command

张子阳

www.tracefact.net

jimmy_dev@163.com

引言

提起 Command 模式，我想没有什么比遥控器的例子更能说明问题了，本文将通过它来一步步实现 GOF 的 Command 模式。

我们先看下这个遥控器程序的需求：假如我们需要为家里的电器设计一个远程遥控器，通过这个控制器，我们可以控制电器(诸如灯、风扇、空调等)的开关。我们的控制器上有一系列的按钮，分别对应家中的某个电器，当我们在遥控器上按下“On”时，电器打开；当我们按下“Off”时，电器关闭。

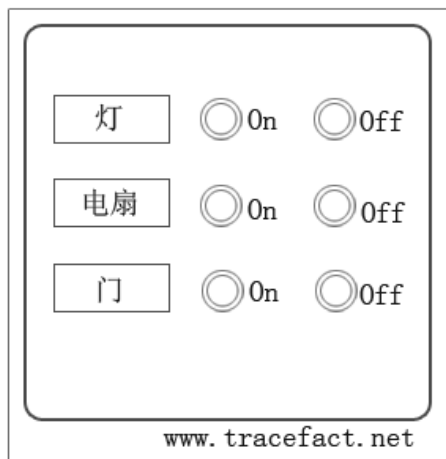
好了，让我们开始 Command 模式之旅吧。

HardCoding 的实现方式

控制器的实现

一般来说，考虑问题通常有两种方式：从最复杂的情况考虑，也就是尽可能的深谋远虑，设计之初就考虑到程序的可维护性、扩展性；还有就是从最简单的情况考虑，不考虑扩展，客户要求什么，我们就做个什么，至于以后有什么新的需求，等以后再说。当然这两种方式各有优劣，本文我们从最简单的情况开始考虑。

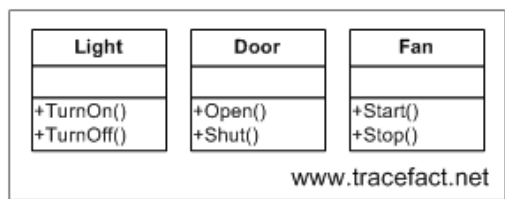
我们假设控制器只能控制三个电器，分别是：灯、电扇、门(你就当是电子门好了^^)。那么我们的控制器应该有三组，共六个按钮，每一组按钮分别有“On”，“Off”按钮。同时，我们规定，第一组按钮对应灯，第二组按钮对应电扇，第三组则对应门，那么控制器应该就像这样：



类的设计

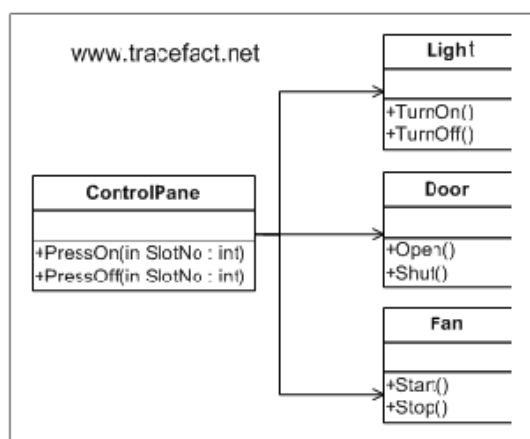
好了，控制器大致是这么个样子了，那么 灯、电扇、门又是什么样子呢？如果你看过前面

几节的模式，你可能会以为此时又要为它们创建一个基类或者接口，然后供它们继承或实现。现在让我们先看看我们想要控制的电器是什么样子的：



很抱歉，你遗憾地发现，它们的接口完全不同，我们没有办法对它们进行抽象，但是因为我们此刻仅考虑客户最原始的需求（最简单的情况），那么我们大可以直接将它们复合到 遥控器 (ControlPanel) 中

NOTE: 关于接口，有狭义的含义：就是一个声明为 interface 的类型。还有一个广义的含义：就是对象暴露给外界的方法、属性，所以一个抽象类也可以称作一个接口。这里，说它们的接口不同，意思是说：这三个电器暴露给外界的方法完全不同。



注意到，PressOn 方法，它代表着某一个按键被按下，并接受一个 int 类型的参数：SlotNo，它代表是第几个键被按下。显然，SlotNo 的取值为 0 到 2。对于 PressOff 则是完全相同的设计。

代码实现

```
namespace Command {

    // 定义灯
    public class Light {
        public void TurnOn() {
            Console.WriteLine("The light is turned on.");
        }
        public void TurnOff() {
            Console.WriteLine("The light is turned off.");
        }
    }
}
```

```
}

// 定义风扇
public class Fan {
    public void Start() {
        Console.WriteLine("The fan is starting.");
    }
    public void Stop() {
        Console.WriteLine("The fan is stopping.");
    }
}

// 定义门
public class Door {
    public void Open() {
        Console.WriteLine("The door is open for you.");
    }
    public void Shut() {
        Console.WriteLine("The door is closed for safety");
    }
}

// 定义遥控器
public class ControlPanel {
    private Light light;
    private Fan fan;
    private Door door;

    public ControlPanel(Light light, Fan fan, Door door) {
        this.light = light;
        this.fan = fan;
        this.door = door;
    }

    // 点击 On 按钮时的操作。slotNo, 第几个按钮被按
    public void PressOn(int slotNo){
        switch (slotNo) {
            case 0:
                light.TurnOn();
                break;
            case 1:
                fan.Start();
                break;
            case 2:
                door.Open();
        }
    }
}
```

```

        break;
    }
}

// 点击 Off 按钮时的操作。
public void PressOff(int slotNo) {
    switch (slotNo) {
        case 0:
            light.TurnOff();
            break;
        case 1:
            fan.Stop();
            break;
        case 2:
            door.Shut();
            break;
    }
}
}

class Program {
    static void Main(string[] args) {
        Light light = new Light();
        Fan fan = new Fan();
        Door door = new Door();

        ControlPanel panel = new ControlPanel(light, fan, door);

        panel.PressOn(0);    // 按第一个 On 按钮，灯被打开了
        panel.PressOn(2);    // 按第二个 On 按钮，门被打开了
        panel.PressOff(2);   // 按第二个 Off 按钮，门被关闭了

    }
}
}

```

输出为：

```

The light is turned on.
The door is open for you.
The door is closed for safety

```

存在问题

这个解决方案虽然能解决当前的问题，但是几乎没有任何扩展性可言。或者说，被调用者

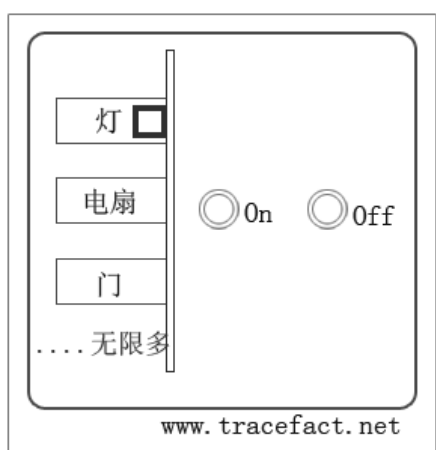
(Receiver: 灯、电扇、门)与它们的调用者(Invoker: 遥控器)是紧耦合的。遥控器不仅需要确切地知道它能控制哪些电器, 并且需要知道这些电器由哪些方法可供调用。

- 如果我们需要调换一下按钮所控制的电器的次序, 比如说我们需要让按钮 1 不再控制灯, 而是控制门, 那么我们需要修改 PressOn 和 PressOff 方法中的 Switch 语句。
- 如果我们需要给遥控器多添一个按钮, 以使它多控制一个电器, 那么遥控器的字段、构造函数、PressOn、PressOff 方法都要修改。
- 如果我们不给遥控器多添按钮, 但是要求它可以控制 10 个或者电器, 换言之, 就是我们可以动态分配某个按钮控制哪个电器, 这样的设计看上去简直无法完成。

HardCoding 的另一实现

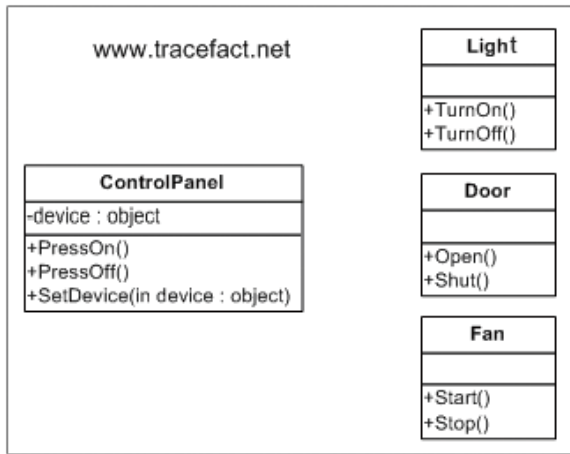
新设计方案

在考虑新的方案以前, 我们先回顾前面的设计, 第三个问题似乎暗示着我们的遥控器不够好, 思考一下, 我们发现可以这样设计遥控器:



对比一下, 我们看到可以通过左侧可以上下活动的阀门来控制当前遥控器控制的是哪个电器 (按照图中当前显示, 控制的是灯), 在选定了阀门后, 我们可以再通过 On, Off 按钮来对电器进行控制。此时, 我们需要多添一个方法, 通过它来控制阀门 (进而选择想要控制的电器)。我们管这个方法叫做 SetDevice()。那么我们的设计变成下图所示:

NOTE: 在图中, 以及现实世界中, 阀门所能控制的电器数总是有限的, 但在程序中, 可以是无限的, 就看你有多少个诸如 light 的电器类了



注意到几点变化：

- 因为我们假设遥控器可以控制的电器是无限多的，所以这里不能指定具体电器类型，因为在 C# 中所有类型均继承自 Object，我们将 SetDevice() 方法接受的参数设置成为 Object。
- ControlPanel 不知道它将控制哪个类，所以图中 ControlPanel 和 Light、Door、Fan 没有联系。
- PressOn() 和 PressOff() 方法不再需要参数，因为很明显，只有一组 On 和 Off 按钮。

代码实现

```

namespace Command {

    public class Light { // 略 }
    public class Fan { // 略 }
    public class Door { // 略 }

    // 定义遥控器
    public class ControlPanel {
        private Object device;

        // 点击 On 按钮时的操作。
        public void PressOn() {
            Light light = device as Light;
            if (light != null) light.TurnOn();

            Fan fan = device as Fan;
            if (fan != null) fan.Start();

            Door door = device as Door;
            if (door != null) door.Open();
        }
    }
}
  
```

```

    }

    // 点击 of 按钮时的操作。
    public void PressOff() {
        Light light = device as Light;
        if (light != null) light.TurnOff();

        Fan fan = device as Fan;
        if (fan != null) fan.Stop();

        Door door = device as Door;
        if (door != null) door.Shut();
    }

    // 设置阀门控制哪个电器
    public void SetDevice(Object device) {
        this.device = device;
    }
}

class Program {
    static void Main(string[] args) {

        Light light = new Light();
        Fan fan = new Fan();

        ControlPanel panel = new ControlPanel();

        panel.SetDevice(light);    // 设置阀门控制灯
        panel.PressOn();           // 打开灯
        panel.PressOff();          // 关闭灯

        panel.SetDevice(fan);      // 设置阀门控制电扇
        panel.PressOn();           // 打开门
    }
}
}

```

存在问题

我们首先可以看到，这个方案似乎解决了第一种设计的大多数问题，除了一点点瑕疵：

- 尽管我们可以控制任意多的设备，但是我们每添加一个可以控制的设备，仍需要修改 `PressOn()` 和 `PressOff()` 方法。

- 在 `PressOn()` 和 `PressOff()` 方法中，需要对所有可能控制的电器进行类型转换，无疑效率低下。

封装调用

问题分析

我们的处境似乎一筹莫展，想不到更好的办法来解决。这时候，让我们先回头再观察一下 `ControlPanel` 的 `PressOn()` 和 `PressOff()` 代码。

```
// 点击 On 按钮时的操作。
public void PressOn() {
    Light light = device as Light;
    if (light != null) light.TurnOn();

    Fan fan = device as Fan;
    if (fan != null) fan.Start();

    Door door = device as Door;
    if (door != null) door.Open();
}
```

我们发现 `PressOn()` 和 `PressOff()` 方法在每次添加新设备时需要作修改，而实际上改变的是对对象方法的调用，因为不管有多少个 `if` 语句，只会调用其中某个不为 `null` 的对象的一个方法。然后我们再回顾一下 OO 的思想，Encapsulate what varies(封装变化)。我们想是不是应该有什么办法将这变化的这部分(方法的调用)封装起来呢？

在考虑如何封装之前，我们假设已经有一个类，把它封装起来了，我们管这个类叫做 `Command`，那么这个类该如何使用呢？

我们先考虑一下它的构成，因为它要封装各个对象的方法，所以，它应该暴露出一个方法，这个方法既可以代表 `light.TurnOn()`，也可以代表 `fan.Start()`，还可以代表 `door.Open()`，让我们给这个方法起个名字，叫做 `Execute()`。

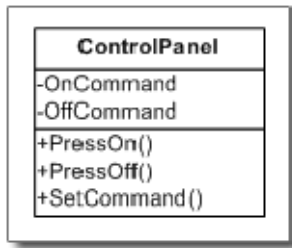
好了，现在我们有 `Command` 类，还有一个万金油的 `Execute()` 方法，现在，我们修改 `PressOn()` 方法，让它通过这个 `Command` 类来控制电器(调用各个类的方法)。

```
// 点击 On 按钮时的操作。
public void PressOn() {
    command.Execute();
}
```

哇，是不是有点简单的过分了！？但就是这么简单，可我们还是发现了两个问题：

1. Command 应该能知道它调用的是哪个电器类的哪个方法，这暗示我们 Command 类应该保存对于具体电器类的一个引用。
2. 我们的 ControlPanel 应该有两个 Command，一个 Command 对应于所有开启的操作(我们管它叫 onCommand)，一个 Command 对应所有关闭的操作(我们管它叫 offCommand)。

同时，我们的 SetDevice(object)方法，也应该改成 SetCommand(onCommand, offCommand)。好了，现在让我们看看新版 ControlPanel 的全景图吧。



Command 类型的实现

显然，我们应该能看出：onCommand 实体变量(instance variable)和 offCommand 变量属于 Command 类型，同时，上面我们已经讨论过 Command 类应该具有一个 Execute()方法，除此以外，它还需要可以保存对各个对象的引用，通过 Execute()方法可以调用其引用的对象的方法。

那么我们按照这个思路，来看下开灯这一操作(调用 light 对象的 TurnOn()方法)的 Command 对象应该是什么样的：

```
public class LightOnCommand{
    Light light;
    public Command(Light light){
        this.light = light;
    }

    public void Execute(){
        light.TurnOn();
    }
}
```

再看下开电扇(调用 fan 对象的 Start()方法)的 Command 对象应该是什么样的：

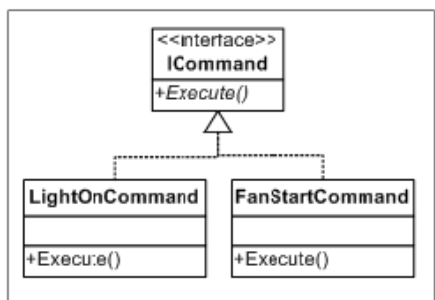
```
public class FanStartCommand{
    Fan fan;
    public Command(Fan fan){
        this.fan = fan;
    }
}
```

```

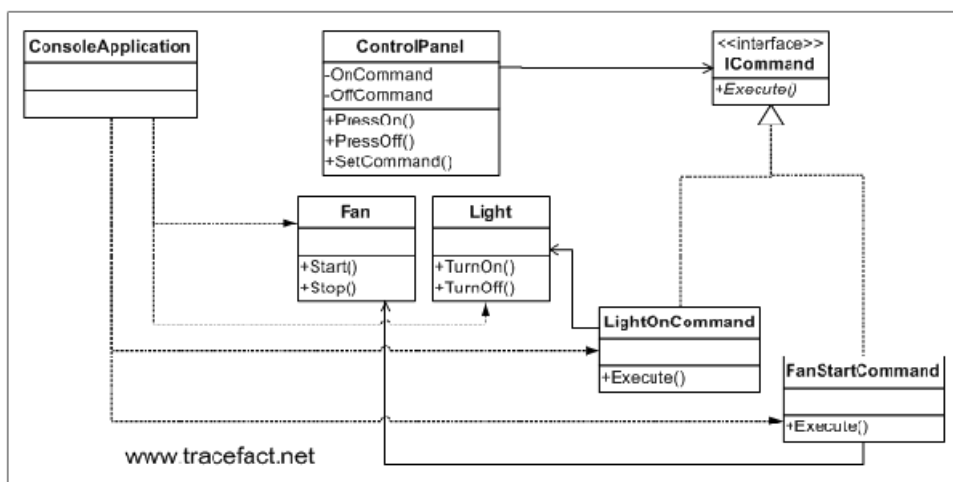
public void Execute(){
    fan.Start();
}
}

```

这样显然是不行的，它没有解决任何的问题，因为 FanStartCommand 和 LightOnCommand 是不同的类型，而我们的 ControlPanel 要求对于所有打开的操作应该只接受一个类型的 Command 的。但是经过我们上面的讨论，我们已经知道所有的 Command 都有一个 Execute() 方法，我们何不定义一个接口来解决这个问题呢？



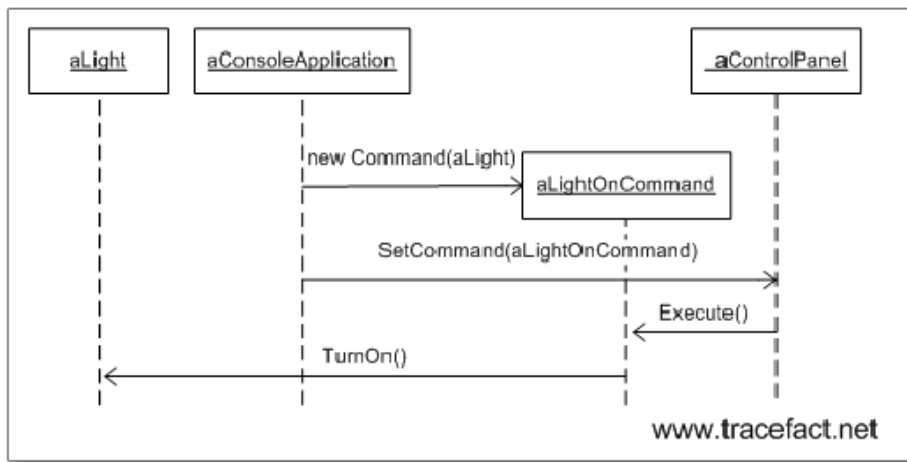
OK，现在我们已经完成了全部的设计，让我们先看一下最终的 UML 图，再进行代码实现吧(简单起见，只加入了灯和电扇)。



我们先看下这张图说明了什么，以及发生的顺序：

1. ConsoleApplication，也就是我们的应用程序，它创建电器 Fan、Light 对象，以及 LightOnCommand 和 FanStartCommand。
2. LightOnCommand、FanStartCommand 实现了 ICommand 接口，它保存着对于 Fan 和 Light 的引用，并通过 Execute() 调用 Fan 和 Light 的方法。
3. ControlPanel 复合了 Command 对象，通过调用 Command 的 Execute() 方法，间接调用了 Light 的 TurnOn() 方法或者是 Fan 的 Stop() 方法。

它们之间的时序图是这样的：



可以看出：通过引入 Command 对象，ControlPanel 对于它实际调用的对象 Fan 或者 Light 是一无所知的，它只知道当 On 按下的时候就调用 onCommand 的 Execute() 方法；当 Off 按下的时候就调用 offCommand 的 Execute() 方法。Light 和 Fan 当然更不知道谁在调用它。通过这种方式，我们实现了 调用者(Invoker, 遥控器 ControlPanel) 和 被调用者(Receiver, 电扇 Fan 等) 的解耦。如果将来我们需要对这个 ControlPanel 进行扩展，只需要再添加一个实现了 ICommand 接口的对象就可以了，对于 ControlPanel 无需做任何修改。

代码实现

```

namespace Command {

    // 定义空调，用于测试给遥控器添新控制类型
    public class AirCondition {
        public void Start() {
            Console.WriteLine("The AirCondition is turned on.");
        }
        public void SetTemperature(int i) {
            Console.WriteLine("The temperature is set to " + i);
        }
        public void Stop() {
            Console.WriteLine("The AirCondition is turned off.");
        }
    }

    // 定义 Command 接口
    public interface ICommand {
        void Execute();
    }

    // 定义开空调命令
    public class AirOnCommand : ICommand {
        AirCondition airCondition;
    }
}
  
```

```

    public AirOnCommand(AirCondition airCondition) {
        this.airCondition = airCondition;
    }
    public void Execute() { //注意, 你可以在 Execute()中添加多个方法
        airCondition.Start();
        airCondition.SetTemperature(16);
    }
}

// 定义关空调命令
public class AirOffCommand : ICommand {
    AirCondition airCondition;
    public AirOffCommand(AirCondition airCondition) {
        this.airCondition = airCondition;
    }
    public void Execute() {
        airCondition.Stop();
    }
}

// 定义遥控器
public class ControlPanel {
    private ICommand onCommand;
    private ICommand offCommand;

    public void PressOn() {
        onCommand.Execute();
    }

    public void PressOff() {
        offCommand.Execute();
    }

    public void SetCommand(ICommand onCommand, ICommand offCommand) {
        this.onCommand = onCommand;
        this.offCommand = offCommand;
    }
}

class Program {
    static void Main(string[] args) {

        // 创建遥控器对象
        ControlPanel panel = new ControlPanel();

```

```

AirCondition airCondition = new AirCondition();           //创建空调对象

// 创建 Command 对象, 传递空调对象
ICommand onCommand = new AirOnCommand(airCondition);
ICommand offCommand = new AirOffCommand(airCondition);

// 设置遥控器的 Command
panel.SetCommand(onCommand, offCommand);

panel.PressOn();           //按下 On 按钮, 开空调, 温度调到 16 度
panel.PressOff();         //按下 Off 按钮, 关空调

}
}
}

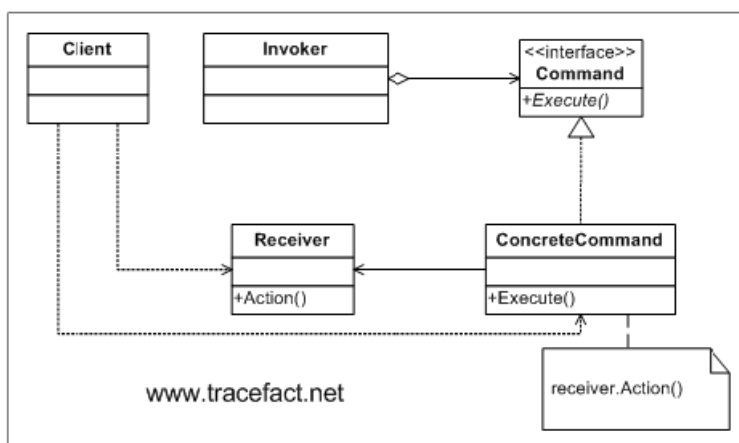
```

Command 模式

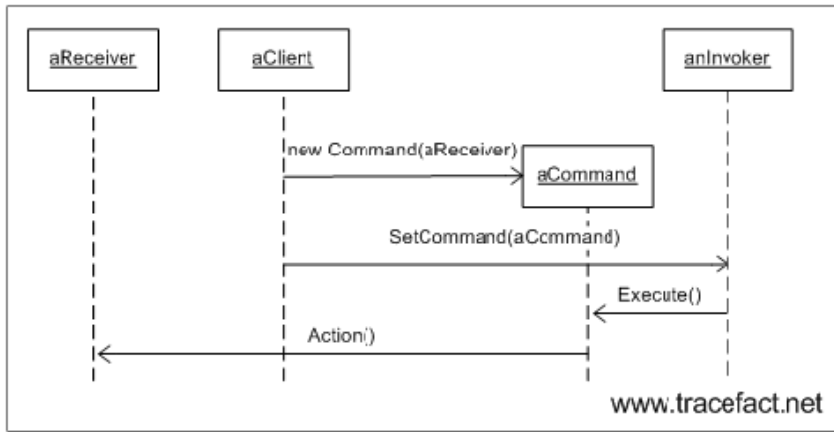
实际上，我们上面做的这一切，实现了另一个设计模式：Command 模式。现在又到了给出官方定义的时候了。每次到了这部分我就不知道该怎么写了，写的人太多了，资料也太多了，我相信你看到这里对 Command 模式已经比较清楚了，所以我还是一如既往地从简吧。

Command 模式的正式定义：**将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。**

它的 静态图 是这样的：



它的 时序图 是这样的：



可以和我们前面的图对比一下，对于这两个图，除了改了个名字外基本没变，我就不再说明了，也留给你一点思考的空间。

总结

本文简单地介绍了 GOF 的 Command 模式，我们通过一个简单的范例 家电遥控器 实现了这一模式。

我们首先了解了不使用此模式的 HardCoding 方式的实现方法，讨论了它的缺点；然后又换了另一种改进了的实现方法，再次讨论了它的不足。然后，我们通过将对象的调用封装到一个 Command 对象中的方式，巧妙地完成了设计。最后，我们给出了 Command 模式的正式定义。

本文仅仅简要介绍了 Command 模式，它的高级应用：取消操作 (Undo)、事务支持 (Transaction)、队列请求 (Queuing Request) 以后有时间了会再写文章。

希望这篇文章能对你有所帮助！