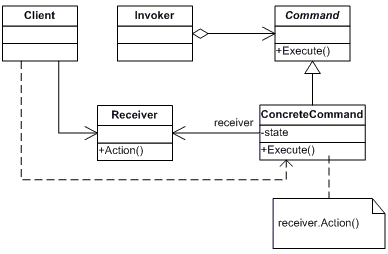
# **二、命令模式的介绍**

## **2.1 命令模式的定义**

命令模式属于对象的行为型模式。命令模式是把一个操作或者行为抽象为一个对象中，通过对命令的抽象化来使得发出命令的责任和执行命令的责任分隔开。命令模式的实现可以提供命令的撤销和恢复功能。

## **2.2 命令模式的结构**

　　既然，命令模式是实现把发出命令的责任和执行命令的责任分割开，然而中间必须有某个对象来帮助发出命令者来传达命令，使得执行命令的接收者可以收到命令并执行命令。例如，开学了，院领导说计算机学院要进行军训，计算机学院的学生要跑1000米，院领导的话也就相当于一个命令，他不可能直接传达给到学生，他必须让教官来发出命令，并监督学生执行该命令。在这个场景中，发出命令的责任是属于学院领导，院领导充当与命令发出者的角色，执行命令的责任是属于学生，学生充当于命令接收者的角色，而教官就充当于命令的发出者或命令请求者的角色，然而命令模式的精髓就在于把每个命令抽象为对象。从而命令模式的结构如下图所示：



从命令模式的结构图可以看出，它涉及到五个角色，它们分别是：

* 客户角色：发出一个具体的命令并确定其接受者。
* 命令角色：声明了一个给所有具体命令类实现的抽象接口
* 具体命令角色：定义了一个接受者和行为的弱耦合，负责调用接受者的相应方法。
* 请求者角色：负责调用命令对象执行命令。
* 接受者角色：负责具体行为的执行。

## **2.3 命令模式的实现**

　　现在，让我们以上面的军训的例子来实现一个命令模式，在实现之前，可以参考下命令模式的结构图来分析下实现过程。

　　军训场景中，具体的命令即是学生跑1000米，这里学生是命令的接收者，教官是命令的请求者，院领导是命令的发出者，即客户端角色。要实现命令模式，则必须需要一个抽象命令角色来声明约定，这里以抽象类来来表示。命令的传达流程是：

　　命令的发出者必须知道具体的命令、接受者和传达命令的请求者，对应于程序也就是在客户端角色中需要实例化三个角色的实例对象了。

　　命令的请求者负责调用命令对象的方法来保证命令的执行，对应于程序也就是请求者对象需要有命令对象的成员，并在请求者对象的方法内执行命令。

　　具体命令就是跑1000米，这自然属于学生的责任，所以是具体命令角色的成员方法，而抽象命令类定义这个命令的抽象接口。

　　有了上面的分析之后，具体命令模式的实现代码如下所示：

// 教官，负责调用命令对象执行请求

public class Invoke

{

public Command \_command;

public Invoke(Command command)

{

this.\_command = command;

}

public void ExecuteCommand()

{

\_command.Action();

}

}

// 命令抽象类

public abstract class Command

{

// 命令应该知道接收者是谁，所以有Receiver这个成员变量

protected Receiver \_receiver;

public Command(Receiver receiver)

{

this.\_receiver = receiver;

}

// 命令执行方法

public abstract void Action();

}

//

public class ConcreteCommand :Command

{

public ConcreteCommand(Receiver receiver)

: base(receiver)

{

}

public override void Action()

{

// 调用接收的方法，因为执行命令的是学生

\_receiver.Run1000Meters();

}

}

// 命令接收者——学生

public class Receiver

{

public void Run1000Meters()

{

Console.WriteLine("跑1000米");

}

}

// 院领导

class Program

{

static void Main(string[] args)

{

// 初始化Receiver、Invoke和Command

Receiver r = new Receiver();

Command c = new ConcreteCommand(r);

Invoke i = new Invoke(c);

// 院领导发出命令

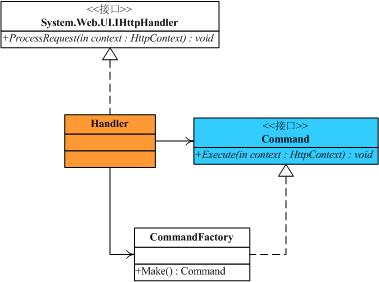
i.ExecuteCommand();

}

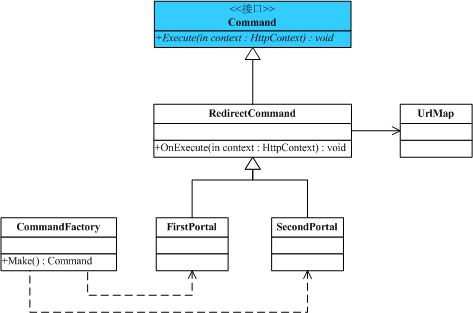
}

# **三、.NET中命令模式的应用(引用TerryLee)**

 在ASP.NET的MVC模式中，有一种叫Front Controller的模式，它分为Handler和Command树两个部分，Handler处理所有公共的逻辑，接收HTTP Post或Get请求以及相关的参数并根据输入的参数选择正确的命令对象，然后将控制权传递到Command对象，由其完成后面的操作，这里面其实就是用到了Command模式。



 　　　　Front Controller 的处理程序部分结构图



　　　　Front Controller的命令部分结构图

Handler 类负责处理各个 Web 请求，并将确定正确的 Command 对象这一职责委派给 CommandFactory 类。当 CommandFactory 返回 Command 对象后，Handler 将调用 Command 上的 Execute 方法来执行请求。具体的实现如下

// Handler类

public class Handler : IHttpHandler

{

public void ProcessRequest(HttpContext context)

{

Command command = CommandFactory.Make(context.Request.Params);

command.Execute(context);

}

public bool IsReusable

{

get

{

return true;

}

}

}

Command接口：

/// <summary>

/// Command

/// </summary>

public interface Command

{

void Execute(HttpContext context);

}

CommandFactory类：

/// <summary>

/// CommandFactory

/// </summary>

public class CommandFactory

{

public static Command Make(NameValueCollection parms)

{

string requestParm = parms["requestParm"];

Command command = null;

//根据输入参数得到不同的Command对象

switch (requestParm)

{

case "1":

command = new FirstPortal();

break;

case "2":

command = new SecondPortal();

break;

default:

command = new FirstPortal();

break;

}

return command;

}

}

RedirectCommand类：

public abstract class RedirectCommand : Command

{

//获得Web.Config中定义的key和url键值对，UrlMap类详见下载包中的代码

private UrlMap map = UrlMap.SoleInstance;

protected abstract void OnExecute(HttpContext context);

public void Execute(HttpContext context)

{

OnExecute(context);

//根据key和url键值对提交到具体处理的页面

string url = String.Format("{0}?{1}", map.Map[context.Request.Url.AbsolutePath], context.Request.Url.Query);

context.Server.Transfer(url);

}

}

FirstPortal类：

public class FirstPortal : RedirectCommand

{

protected override void OnExecute(HttpContext context)

{

//在输入参数中加入项portalId以便页面处理

context.Items["portalId"] = "1";

}

}

SecondPortal类：

public class SecondPortal : RedirectCommand

{

protected override void OnExecute(HttpContext context)

{

context.Items["portalId"] = "2";

}

}

# **四、命令模式的适用场景**

 　　在下面的情况下可以考虑使用命令模式：

1. 系统需要支持命令的撤销（undo）。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用undo方法吧命令所产生的效果撤销掉。命令对象还可以提供redo方法，以供客户端在需要时，再重新实现命令效果。
2. 系统需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命周期。意思为：原来请求的发出者可能已经不存在了，而命令对象本身可能仍是活动的。这时命令的接受者可以在本地，也可以在网络的另一个地址。命令对象可以串行地传送到接受者上去。
3. 如果一个系统要将系统中所有的数据消息更新到日志里，以便在系统崩溃时，可以根据日志里读回所有数据的更新命令，重新调用方法来一条一条地执行这些命令，从而恢复系统在崩溃前所做的数据更新。
4. 系统需要使用命令模式作为“CallBack(回调)”在面向对象系统中的替代。Callback即是先将一个方法注册上，然后再以后调用该方法。

# **五、命令模式的优缺点**

 　　命令模式使得命令发出的一个和接收的一方实现低耦合，从而有以下的优点：

* 命令模式使得新的命令很容易被加入到系统里。
* 可以设计一个命令队列来实现对请求的Undo和Redo操作。
* 可以较容易地将命令写入日志。
* 可以把命令对象聚合在一起，合成为合成命令。合成命令式合成模式的应用。

　　命令模式的缺点：

* 使用命令模式可能会导致系统有过多的具体命令类。这会使得命令模式在这样的系统里变得不实际。

# **六、总结**

 　　命令模式的实现要点在于把某个具体的命令抽象化为具体的命令类，并通过加入命令请求者角色来实现将命令发送者对命令执行者的依赖分割开，在上面军训的例子中，如果不使用命令模式的话，则命令的发送者将对命令接收者是强耦合的关系，实现代码如下：

// 院领导

class Program

{

static void Main(string[] args)

{

// 行为的请求者和行为的实现者之间呈现一种紧耦合关系

Receiver r = new Receiver();

r.Run1000Meters();

}

}

public class Receiver

{

// 操作

public void Run1000Meters()

{

Console.WriteLine("跑1000米");

}

}

　　到这里，本章的内容就介绍结束了