第3章 游戏场景的转换----状态模式(State)

　　3.1 游戏场景

Unity3D是使用场景(Scene)作为游戏运行时的环境.开始制作游戏时,开发者会将游戏需要的素材(3D模型,游戏对象)放到一个场景中,然后编写对应的程序代码,之后只要单击Play按钮,就可以开始运行游戏

笔者过去开发游戏时使用的游戏引擎(Game Engine)或开发框架(SDK, Framework),多数也都存在"场景"的概念,例如

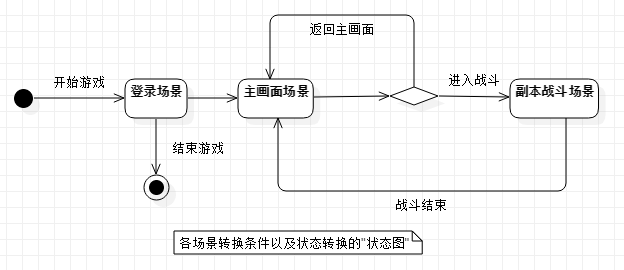
　　早期Java Phone的J2ME开发SDK中使用的Canvas类

　　Android的Java开发SDK中使用的Activity类

　　iOS上2D游戏开发工具Cocos2D中使用的CCScene类

虽然各种工具不见得都使用场景(Scene)这个名词,但在实际上,一样可使用相同的方式来呈现.而上面所列的各个类,都可以被拿来作为游戏实现中"场景"转换的目标

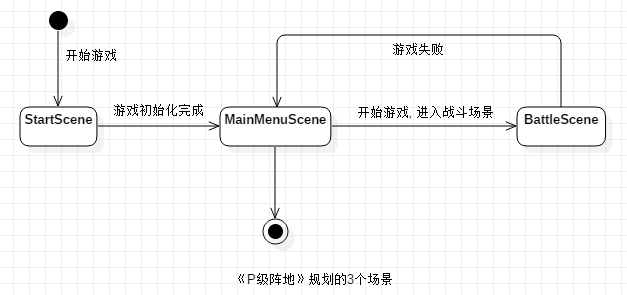
　　　　3.1.1 场景的转换



切分场景的好处

将游戏中不同的功能分类在不同的场景中来执行,除了可以将游戏功能执行时需要的环境明确分类之外,"重复使用"也是使用场景转换的好处之一

本书范例场景的规划



　　　　3.1.2 游戏场景可能的实现方式

Easy Way

1. **public** **class** SceneManager {
2. **private** string m\_state = "开始";
4. // 改换场景
5. **public** **void** ChangeScene(string StateName) {
6. m\_state = StateName;
8. **switch** (m\_state) {
9. **case** "菜单":
10. Application.LoadLevel("MainMenuScene");
11. **break**;
12. **case** "主场景":
13. Application.LoadLevel("GameScene");
14. **break**;
15. }
16. }
18. // 更新
19. **public** **void** Update() {
20. **switch** (m\_state) {
21. **case** "开始":
22. // ...
23. **break**;
24. **case** "菜单":
25. // ...
26. **break**;
27. **case** "主场景":
28. // ...
29. **break**;
30. }
31. }
32. }

缺点

　　只要增加一个状态,则所有switch(m\_state)的程序代码都需要增加对应的程序代码

　　与每一个状态有关的对象, 都必须在SceneManager类中被保留,当这些对象被多个状态共享时,可能会产生混淆,不太容易识别是由哪个状态设置的,造成游戏程序调试上的困难

　　每一个状态可能使用不同的类对象,容易造成StageManager类过度依赖其他类,让SceneManager类不容易移植到其他项目中

为了避免出现上述缺点,修正的目标会希望使用一个"场景类"来负责维护一个场景,让与此场景相关的程序代码和对象能整合在一起.这个负责维护的"场景类",其主要工作如下:

　　场景初始化

　　场景结束后,负责清除资源

　　定时更新游戏逻辑单元

　　转换到其他场景

　　其他与该场景有关的游戏实现

由于在范例程序中我们规划了3个场景,所以会产生对应的3个"场景类",但如何让这3个"场景类"相互合作,彼此转换呢?我们可以使用GoF的状态模式(State)来解决这些问题

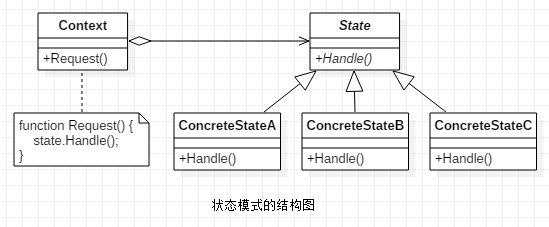
　　3.2 状态模式(State)

　　　　3.2.1 状态模式(State)的定义

状态模式(State), 在GoF中的解释是:

"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"

　　　　3.2.2 状态模式(State)的说明



参与者的说明如下:

　　Context(状态拥有者)

　　　　是一个具有"状态"属性的类, 可以制定相关的接口, 让外界能够得知状态的改变或通过操作让状态改变

　　　　有状态属性的类, 例如: 游戏角色有潜行, 攻击, 施法等状态; 好友上线, 脱机, 忙碌等状态; GoF使用TCP联网为例, 有已连接, 等待连接, 断线等状态. 这些类中会有一个ConcreteState[X]子类的对象为其成员, 用来代表当前的状态

　　State(状态接口类)

　　　　 制定状态的接口, 负责规范Context(状态拥有者)在特定状态下要表现的行为

　　ConcreteState(具体状态的类)

　　　　继承自State(状态接口类)

　　　　实现Context(状态拥有者)在特定状态下该有行为.例如, 实现角色在潜行状态时该有的行动变缓, 3D模型变半透明, 不能被敌方角色察觉等行为

　　　　3.2.3 状态模式(State)的实现范例

2. **public** **class** Context {
3. State m\_state = null;
5. **public** **void** Request(**int** Value) {
6. m\_State.Handle(Value);
7. }
9. **public** **void** SetState(State theState) {
10. Debug.Log("Context.SetState:" + theState);
11. m\_state = theState;
12. }
13. }
15. **public** abstract **class** State {
16. **protected** Context m\_Context = null;
18. **public** State(Context theContext) {
19. m\_Context = theContext;
20. }
22. **public** abstract **void** Handle(**int** Value);
23. }
25. **public** **class** ConcreteStateA: State {
26. **public** ConcreteStateA(Context theContext): base(theContext) {
27. }
29. **public** override **void** Handle(**int** Value) {
30. Debug.Log("ConcreteStateA.Handle");
31. **if** (Value > 10) {
32. m\_Context.SetState(**new** ConcreteStateB(m\_Context));
33. }
34. }
35. }
37. **public** **class** ConcreteStateB: State {
38. **public** ConcreteStateB(Context theContext): base(theContext) {
39. }
41. **public** override **void** Handle(**int** Value) {
42. Debug.Log("ConcreteStateB.Handle");
43. **if** (Value > 20) {
44. m\_Context.SetState(**new** ConcreteStateC(m\_Context));
45. }
46. }
47. }
49. **public** **class** ConcreteStateC: State {
50. **public** ConcreteStateC(Context theContext): base(theContext) {
51. }
53. **public** override **void** Handle(**int** Value) {
54. Debug.Log("ConcreteStateC.Handle");
55. **if** (Value > 30) {
56. m\_Context.SetState(**new** ConcreteStateA(m\_Context));
57. }
58. }
59. }
61. **void** UnitTest() {
62. Context theContext = **new** Context();
63. theContext.SetState(**new** ConcreteStateA(theContext));
64. theContext.Request(5);
65. theContext.Request(15);
66. theContext.Request(25);
67. theContext.Request(35);
68. }
70. Context.SetState:DesignPattern\_State.ConcreteStateA
71. ConcreteStateA.Handle
72. ConcreteStateA.Handle
73. Context.SetState:DesignPattern\_State.ConcreteStateB
74. Context.SetState:DesignPattern\_State.ConcreteStateC
75. Context.SetState:DesignPattern\_State.ConcreteStateA

Context类中提供了一个SetState方法,让外界能够设置Context对象当前的状态,而所谓的"外界", 也可以是由另一个State状态来调用.所以实现上,状态的转换可以有下列两种方式:

　　交由Context类本身,按条件在各状态之间转换

　　产生Context类对象时, 马上指定初始状态给Context对象,而在后续执行过程中的状态转换则交由State对象负责,Context对象不再介入

笔者在实现时,大部分情况下会选择第2种方式,原因在于:

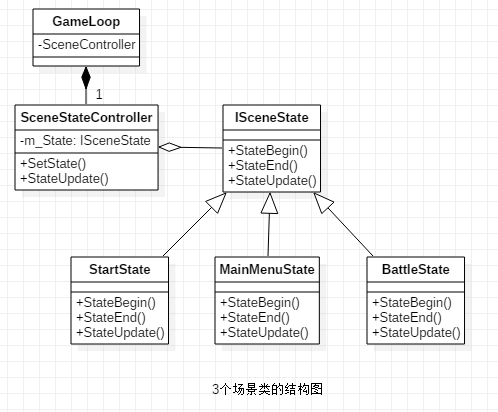
　　状态对象本身比较清楚"在什么条件下, 可以让Context对象转移到下一个State状态".所以在每个ConcreteState类的程序代码中,可以看到"状态转换条件"的判断, 以及设置哪一个ConcreteState对象成为新的状态

　　每个ConcreteState状态都可以保持自己的属性值,作为状态转换或展现状态行为的依据,不会与其他的ConcreteState状态混用,在维护时比较容易理解

　　因为判断条件及状态属性都被转换到ConcreteState类中,故而可缩减Context类的大小

　　3.3 使用状态模式(State)实现游戏场景的转换

　　　　3.3.1 SceneState的实现



参与者如下:

　　ISceneState: 场景类的接口, 定义《P级阵地》种场景转换和执行时需要调用的方法

　　StartState, MainMenuState, BattleState: 分别对应范例中的开始场景(StartScene), 主画面场景(MainMenuScene)及战斗场景(BattleScene),作为这些场景执行时的操作类

　　SceneStateController: 场景状态的拥有者(Context), 保持当前游戏场景状态, 并作为与GameLoop类互动的接口. 除此之外, 也是执行"Unity3D场景转换"的地方

　　GameLoop:: 游戏主循环类作为Unity3D与《P级阵地》的互动接口,包含了初始化游戏和定期调用更新操作

　　　　3.3.2 实现说明

1. **public** **class** ISceneState {
2. **private** string m\_StateName = "ISceneState";
3. **public** string StateName {
4. get { **return** m\_StateName; }
5. set { m\_StateName = value; }
6. }
8. **protected** SceneStateController m\_Controller = null;
10. **public** ISceneState(SceneStateController Controller) {
11. m\_Controller = Controller;
12. }
14. **public** **virtual** **void** StateBegin() { }
16. **public** **virtual** **void** StateEnd() { }
18. **public** **virtual** **void** StateUpdate() { }
20. **public** override string ToString() {
21. **return** string.Format("[I\_SceneState: StateName = {0}]", StateName);
22. }
23. }
25. **public** **class** StartScene: ISceneState {
26. **public** StartScene(SceneStateController Controller): base(Controller) {
27. **this**.StateName = "StartState";
28. }
30. **public** override **void** StateBegin() {
32. }
34. **public** override **void** StateUpdate() {
35. m\_Controller.SetState(**new** MainMenuState(m\_Controller), "MainMenuScene");
36. }
37. }
39. **public** **class** MainMenuState: ISceneState {
40. **public** MainMenuState(SceneStateController Controller): base(Controller) {
41. **this**.StateName = "MainMenuState";
42. }
44. **public** override **void** StateBegin() {
45. Button tmpBtn = UITool.GetUIComponent<Button>("StartGameBtn");
46. **if** (tempBtn != null) {
47. tmpBtn.onClick.AddListener(()=>OnStartGameBtnClick(tmpBtn));
48. }
50. **public** **void** OnStartGameClick(Button theButton) {
51. m\_Controller.SetState(**new** BattleState(m\_Controller), "BattleScene");
52. }
53. }
55. **public** **class** BattleScene: ISceneState {
56. **public** BattleScene(SceneStateController Controller): base(Controller) {
57. **this**.StateName = "BattleState";
58. }
60. **public** overrride **void** StateBegin() {
61. PBaseDefenseGame.Instance.Initial();
62. }
64. **public** ovrride **void** StateEnd() {
65. PBaseDefenseGame.Instance.Update();
66. }
68. **public** ovrride **void** StateUpdate() {
69. InputProcess();
70. PBaseDefenseGame.Instance.Update();
71. **if** (PBaseDefenseGame.Instance.ThisGameIsOver()) {
72. m\_Controller.SetState(**new** MainMenuState(m\_Controller), "MainMenuScene");
73. }
74. }
76. **private** **void** InputProcess() {
77. //...
78. }
79. }
81. **public** **class** SceneStateController {
82. **private** ISceneState m\_State;
83. **private** **bool** m\_bRunBegin = **false**;
85. **public** SceneStateController() { }
87. **public** **void** SetState(ISceneState State, string LoadSceneName) {
88. m\_bRunBegin = **false**;
90. LoadScene(LoadSceneName);
92. **if** (m\_State != null) {
93. m\_State.StateEnd();
94. }
96. m\_State = State;
97. }
99. **private** **void** LoadScene(string LoadSceneName) {
100. **if** (LoadSceneName == null || LoadSceneName.Length == 0) {
101. **return**;
102. }
104. Application.LoadLevel(LoadSceneName);
105. }
107. **public** **void** StateUpdate() {
108. **if** (Application.isLoadingLevel) {
109. **return**;
110. }
112. **if** (m\_state != null && m\_bRunBegin == **false**) {
113. m\_State.StateBegin();
114. m\_bRunBegin = **true**;
115. }
117. **if** (m\_State != null) {
118. m\_State.StateUpdate();
119. }
120. }
121. }
123. **public** **class** GameLoop: MonoBehavior {
124. SceneStateController m\_SceneStateController = **new** SceneStateController();
126. **void** Awake() {
127. GameObject.DontDestroyOnLoad(**this**.gameObject);
128. UnityEngine.Random.seed = (**int**)DateTime.Now.Ticks;
129. }
131. **void** Start() {
132. m\_SceneStateController.SetState(**new** StartState(m\_SceneStateController), "");
133. }
135. **void** Update() {
136. m\_SceneStateController.StateUpdate();
137. }
138. }
139. 复制代码

　　　　3.3.3 使用状态模式(State)的优点

使用状态模式(State)来实现游戏场景转换,有下列优点:

　　减少错误的发生并降低维护难度

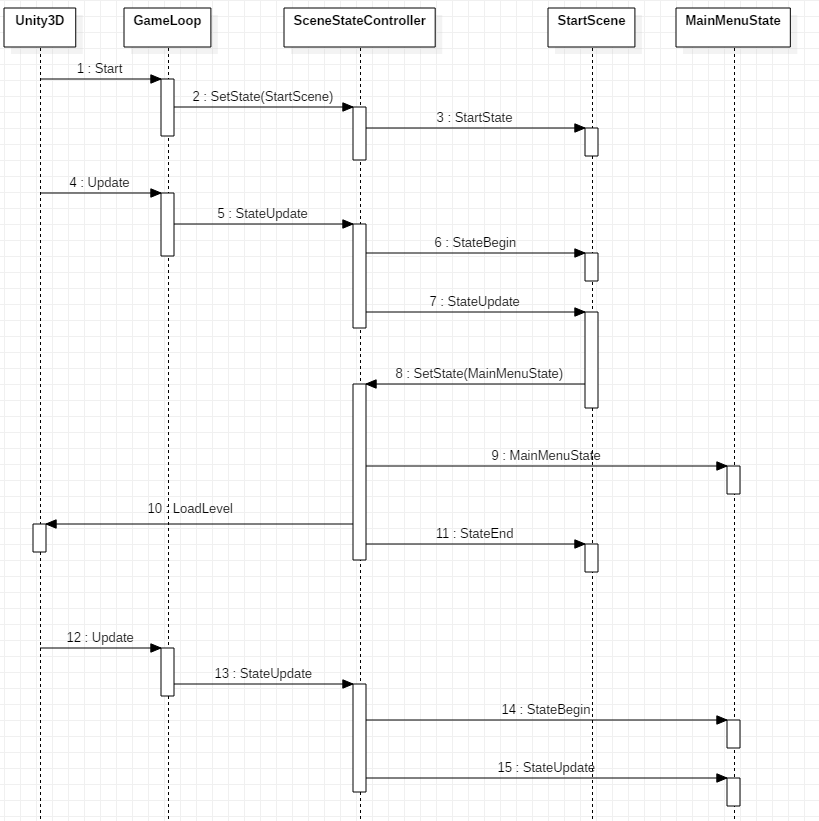
不再使用switch(m\_state)来判断当前的状态,这样可以减少新增游戏状态时,因未能检查到所有switch(m\_state)程序代码而造成的错误

　　状态执行环境单一化

与每一个状态有关的对象及操作都被实现在一个场景状态类下,对程序设计师来说,这样可以清楚地了解每一个状态执行时所需要的对象及配合的类

　　项目之间可以共享场景

　　　　3.3.4 游戏执行流程及场景转换说明



　　3.4 状态模式(State)面对变化时

随着项目开发进度进入中后期,游戏企划可能会提出新的系统功能来增加游戏内容.这些提案可能是增加小游戏关卡,提供查看角色信息图鉴,玩家排行等功能.当程序人员在分析这些新增的系统需求后,如果觉得无法在现有的场景(Scene)下实现,就必须使用新的场景来完成.而在现有的架构下,程序人员只需要完成下列几项工作:

　　在Unity3D编辑模式下新增场景

　　加入一个新的场景状态类对应到新的场景,并在其中实现相关功能

　　决定要从哪个现有场景转换到新的场景

　　决定新的场景结束后要转换到哪一个场景

上述流程,就程序代码的修改而言,只会新增一个程序文件(.cs)用来实现新的场景状态类,并修改一个现有的游戏状态,让游戏能按照需求转换到新的场景状态.除此之外,不需要修改其他任何的程序代码

　　3.5 结论

在本章中,我们利用状态模式(State)实现了游戏场景的切换,这种做法并非全然都是优点,但与传统的switch(state\_code)相比,已经算是更好的设计.

状态模式(State)的优缺点

使用状态模式(State)可以清楚地了解某个场景状态执行时所需要配合使用的类对象,并且减少因新增状态而需要大量修改现有程序代码的维护成本

《P级阵地》只规划了3个场景来完成整个游戏,算是"产出较少状态类"的应用.但如果状态模式(State)是应用在大量状态的系统时,就会遇到"产生过多状态类"的情况,此时会伴随着类爆炸的问题,这算是一个缺点.不过与传统使用switch(state\_code)的实现方式相比,使用状态模式(State)对于项目后续的长期维护效益上,仍然具有优势

与其他模式(Pattern)的合作

在《P级阵地》的BattleScene类实现中,分别调用了PBaseDefenseGame类的不同方法,此时的PBaseDefenseGame使用的是"单例模式(Singleton)",这是一种让BattleScene类方法中的程序代码,可以取得唯一对象的方式.而PBaseDefenseGame也使用了"外观模式(Facade)"来整合PBaseDefenseGame内部的复杂系统,因此BattleScene类不必了解太多关于PBaseDefenseGame内部的实现方式.

状态模式(State)的其他应用方式:

　　角色AI: 使用状态模式(State)来控制角色在不同状态下的AI行为

　　游戏服务器连线状态: 网络游戏的客户端,需要处理与游戏服务器的连线状态,一般包含开始连线,连线中,断线等状态,而在不同的状态下,会有不同的封包信息处理方式,需要分别实现

　　关卡进行状态: 如果是通关型游戏,进入关卡时通常会分成不同的阶段,包含加载数据,显示关卡信息,倒数通知开始,关卡进行,关卡结束和分数计算,这些不同的阶段可以使用不同的状态类来负责实现

第4章 游戏主要类----外观模式(Facade)

　　4.1 游戏子功能的整合

一款游戏要能顺利运行,必须同时由内部数个不同的子系统一起合作完成.在这些子系统中,有些是在早期游戏分析时规划出来的,有些则是实现过程中,将相同功能重构整合之后才完成的.以《P级阵地》为例,它是由下列游戏系统所组成:

　　游戏事件系统(GameEventSystem)

　　兵营系统(CampSystem)

　　关卡系统(StageSystem)

　　角色管理系统(CharacterSystem)

　　行动力系统(APSystem)

　　成就系统(AchivementSystem)

这些系统在游戏运行时会彼此使用对方的功能,并且通知相关信息或传送玩家的指令.另外,有些子系统必须在游戏开始运行前,按照一定的步骤将它们初始化并设置参数,或者游戏在完成一个关卡时,也要按照一定的流程替它们释放资源

可以理解的是, 上面这些子系统的沟通及初始化过程都发生在"内部"会比较恰当,因为对于外界或客户端来说,大可不必去了解它们之间的相关运行过程.如果客户端了解太多系统内部的沟通方式及流程,那么对于客户端来说,就必须与每一个游戏系统绑定,并且调用每一个游戏系统的功能.这样的做法对于客户端来说并不是一件好事,因为客户端可能只是单纯地想使用某一项游戏功能而已,但它却必须经过一连串的子系统调用之后才能使用,对于客户端来说,压力太大,并且让客户端与每个子系统都产生了依赖性,增加了游戏系统与客户端的耦合度

如果要在我们的游戏范例中举一个例子,那么上一章所提到的"战斗状态类(BattleScene)"就是一个必须使用到的游戏系统功能的客户端

根据上一章的说明,战斗状态类(BattleState)主要负责游戏战斗的运行,而《P级阵地》在进行一场战斗时,需要大部分的子系统一起合作完成.在实现时,可以先把这些子系统及相关的执行流程全都放在BattleState类之中一起完成

1. **public** **class** BattleState: ISceneState {
2. **private** GameEventSystem m\_GameEventSystem = null;
3. **private** CampSystem m\_CampSystem = null;
4. **private** StageSystem m\_StageSystem = null;
5. **private** CharacterSystem m\_CharacterSystem = null;
6. **private** APSystem m\_ApSystem = null;
7. **private** AchivementSystem m\_AchievementSystem =null;
9. **public** GameState(SceneStateController Controller): base(Controller) {
10. **this**.StateName = "GameState";
11. InitGameSystem();
12. }
14. **private** **void** InitGameSystem() {
15. m\_GameEventySystem = **new** GameEventSystem();
16. ...
17. }
19. **private** **void** UpdateGameSystem() {
20. m\_GameEventSystem.Update();
21. ...
22. }
23. }

虽然这样的实现方式很简单,但就如本章一开始所说明的,让战斗状态类(BattleState)整个客户端去负责调用所有与游戏玩法相关的系统功能,是不好的实现方式,原因是:

　　从让事情单一化(单一职责原则)这一点来看,BattleScene类负责的是游戏在"战斗状态"下的功能执行及状态切换,所以不应该负责游戏子系统的初始化,执行操作及相关的整合工作

　　以"可重用性"来看,这种设计方式会使得BattleState类不容易转换给其他项目使用,因为BattleState类与太多特定的子系统类产生关联,必须将它们删除才能转换给其他项目,因此丧失可重用性

综合上述两个原因,将这些子系统从BattleState类中移出,整合在单一类之下,会是比较好的做法.所以,在《P级阵地》中应用了外观模式(Facade)来整合这些子系统,使它们成为单一界面并提供外界使用

　　4.2 外观模式(Facade)

其实,外观模式(Facade)是在生活中最容易碰到的模式.当我们能够利用简单的行为来操作一个复杂的系统时,当下所使用的接口,就是以外观模式(Facade)来定义的高级接口

　　　　4.2.1 外观模式(Facade)的定义

外观模式(Facade)在GoF的解释是:

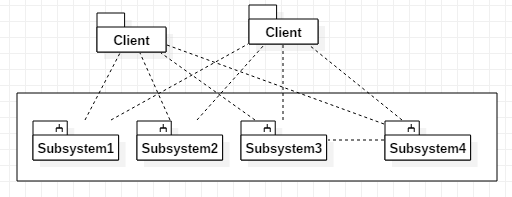
"为子系统定义一组统一的接口,这个高级的接口会让子系统更容易被使用"

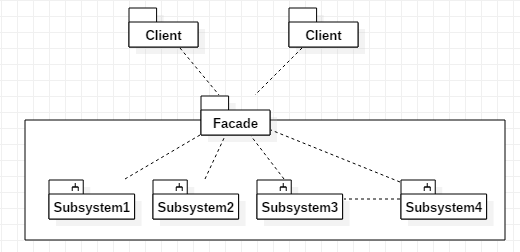
以驾驶汽车为例,当驾驶者能够开着一辆汽车在路上行走,汽车内部还必须由许多的子系统一起配合才能完成汽车行走这项功能,这些子系统包含引擎系统,传动系统,悬吊系统,车身骨架系统,电装系统等.但对于客户端(驾驶者)而言,并不需要了解这些子系统是如何协调工作的,驾驶者只需要通过高级接口(方向盘,踏盘,仪表盘)就可以轻易操控汽车

以微波炉为例,微波炉内部包含了电源供应系统,微博加热系统,冷却系统,外装防护等.当我们想要使用微波炉加热食物时,只需要使用微波炉上面的面板调整火力和时间,按下启动键后,微波炉的子系统就会立即交互合作将食物加热

所以,外观模式(Facade)的重点在于,它能将系统内部的互动细节隐藏起来,并提供一个简单方便的接口.之后客户端只需要通过这个接口,就可以操作一个复杂系统并让它们顺利运行

　　　　4.2.2 外观模式(Facade)的说明





参与者的说明如下:  
　　client(客户端,用户)

从原本需要操作多个子系统的情况,改为只需要面对一个整合后的界面

　　subSystem(子系统)

原本会由不同的客户端(非同一系统相关)来操作,改为只会由内部系统之间交互使用

　　Facade(统一对外的界面)

整合所有子系统的接口及功能,并提供高级界面(或接口)供客户端使用

接收客户端的信息后,将信息传送给负责的子系统

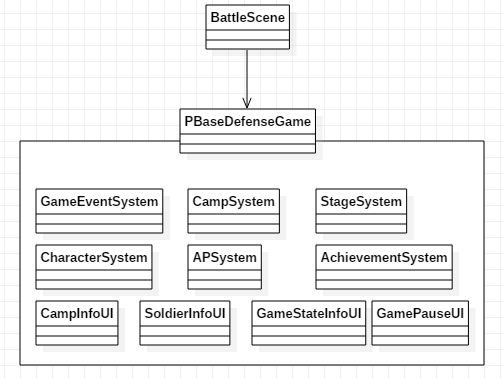
　　　　4.2.3 外观模式(Facade)的实现说明

从之前提到的一些实例来看,驾驶座位前的方向盘,仪表板,以及微波炉上的面板,都是制造商提供给用户使用的Facade界面

　　4.3 使用外观模式(Facade)实现游戏主程序

　　　　4.3.1 游戏主程序架构设计

PBaseGameDefenseGame就是"整合所有子系统,并提供高级界面的外观模式类"



参与者说明如下:

　　GameEventSystem, CampSystem......: 分别为游戏的子系统,每个系统负责各自应该实现的功能并提供接口

　　PBaseDefenseGame: 包含了和游戏相关的子系统对象, 并提供了界面让客户端使用

　　BattleScene: 战斗状态类, 即是《P级阵地》中与PBaseDefenseGame互动的客户端之一

　　　　4.3.2 实现说明

1. PBaseDefenseGame.cs
3. **public** **class** PBaeDefenseGame {
4. ...
5. **private** GameEventSystem m\_GameEventSystem =null;
6. ...
7. }
9. **public** **void** Initinal() {
10. ...
11. m\_GameEventSystem = **new** GameEventSystem(**this**);
12. ...
13. }
15. **public** **void** Update() {
16. ...
17. m\_GameEventSystem.Update();
18. ...
19. }
21. BattleState.cs
23. **public** **class** BattleState: ISceneState {
24. **public** override **void** StateBegin() {
25. PBaseDefenseGame.Instance.Initinal();
26. }
28. **public** override **void** StateEnd() {
29. PBaseDefenseGame.Instance.Release();
30. }
32. **public** override **void** StateUpdate() {
33. ...
34. PBaseDefenseGame.Instance.Update();
35. ...
36. **if** (PBaseDefenseGame.Instance.ThisGameIsOver()) {
37. m\_Controller.SetState(**new** MainMenuState(m\_Controller), "MainMenuState");
38. }
39. }
40. }

　　　　4.3.3 使用外观模式(Facade)的优点

将游戏相关的系统整合在一个类下,并提供单一操作界面供客户端使用,与当初所有功能都直接实现在BattleScene类中的方式相比,具有以下几项优点

　　使用外观模式(Facade)可将战斗状态类BattleState单一化,让该类只负责游戏在"战斗状态"下的功能执行及状态切换,不用负责串接各个游戏系统的初始化和功能调用

　　使用外观模式(Facade)使得战斗状态类BattleScene减少了不必要的类引用及功能整合,因此增加了BattleState类被重复使用的机会

除了上述优点之外,外观模式(Facade)如果应用得当,还具有下列优点:

节省时间

　　Unity3D本身提供了不少系统的Facade接口,例如物理引擎,渲染系统,动作系统,粒子系统等.

易于分工开发

　　对于一个既庞大又复杂的子系统而言,若应用外观模式(Facade), 即可成为另一个Facade接口.所以,在工作的分工配合上,开发者只需要了解对方负责系统的Facade接口类,不必深入了解其中的运行方式

增加系统的安全性

　　隔离客户端对子系统的接触,除了能减少耦合度之外,安全性也是重点之一

　　　　4.3.4 实现外观模式(Facade)时的注意事项

由于将所有子系统集中在Facade接口类中,最终会导致Facade接口类过于庞大且难以维护,当发生这种情况时,可以重构Facade接口类,将功能相近的子系统进行整合,以减少内部系统的依赖性,或是整合其他设计模式来减少Facade接口类过度膨胀

　　4.4 外观模式(Facade)面对变化时

随着开发需求的变更,任何游戏子系统的修改及更换,都被限制在PBaseDefenseGame这个Facade接口类内.所以, 当有新的系统需要增加时,也只会影响PBaseDefenseGame类的定义及增加对外开放的方法,这样就能使项目的变动范围减到最小

　　4.5 结论

将复杂的子系统沟通交给单一的一个类负责,并提供单一界面给客户端使用,使客户减少对系统的耦合度是外观模式(Facade)的优点.

与其他模式(Pattern)的合作

　　在《P级阵地》中,PBaseDefenseGame类使用单例模式(Singleton)来产生唯一的类对象,内部子系统之间则使用中介者模式(Mediator)作为互相沟通的方式,而游戏事件系统(GameEventSystem)是观察者(Observer)的实现,主要的目的就是减少PBaseDefenseGame类接口过于庞大而加入的设计

其他应用方式

　　网络引擎: 网络通信是一项复杂的工作,通常包含连线管理系统,信息事件系统,网络数据封包管理系统等, 所以一般会用外观模式(Facade)将上述子系统整合为一个系统

　　数据库引擎: 在游戏服务器的实现中,可以将与"关系数据库"(MySQL, MSSQL等)相关的操作, 以一种较为高级的接口隔离, 这个接口可以将数据库系统中所需的连线,数据表修改,新增,删除,更新,查询等的操作加以封装,让不是很了解关系数据库原理的设计人员也能使用

第5章 获取游戏服务的唯一对象----单例模式(Singleton)

　　5.1 游戏实现中的唯一对象

生活中的许多物品都是唯一的,地球是唯一的,太阳是唯一的等.软件设计上也会有唯一的对象的的需求,例如:服务器端的程序只能连接到一个数据库,只能有一个日志产生器,有些世界也是一样的,同时间只能有一个关卡正在进行,只能连线到一台游戏服务器,只能同时操作一个橘色等

　　5.2 单例模式(Singleton)

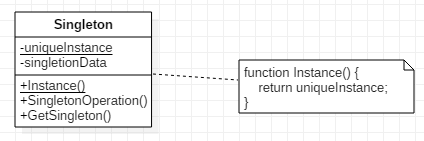
　　　　5.2.1 单例模式(Singleton)的定义

单例模式(Singleton)在GoF中的定义是:

"确认类只有一个对象,并提供一个全局的方法来获取这个对象"

单例模式(Singleton)在实现时,需要程序设计语言的支持.只要具有静态类属性,静态类方法和重新定义类建造者存取层级.3项语句功能的程序设计语言,就可以实现出单例模式(Singleton)

　　　　5.2.2 单例模式(Singleton)的说明



参与者如下:

　　能产生唯一对象的类,并且提供"全局方法"让外界可以方便获取唯一的对象

　　通常会把唯一的类对象设置为"静态类属性"

　　习惯上会使用Instance作为全局静态方法的名称,通过这个静态函数可能获取"静态类属性"

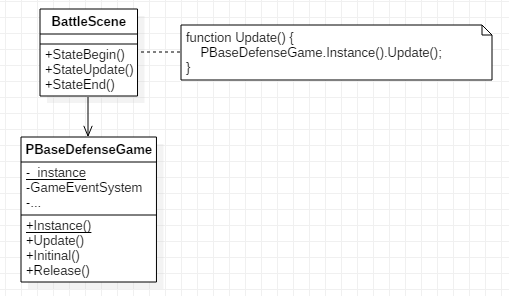
　　　　5.2.3 单例模式(Singleton)的实现范例

1. **public** **class** Singleton {
2. **public** string Name { get; set; }
4. **private** **static** Singleton \_instance;
6. **public** **static** Singleton Instance {
7. get {
8. **if** (\_instance == null) {
9. Debug.Log("产生Singleton");
10. \_instance = **new** Singleton();
11. }
12. **return** \_instance;
13. }
14. }
16. **private** Single() { }
17. }
19. **void** UnitTest() {
20. Singleton.Instance.Name = "Hello";
21. Singleton.Instance.Name = "World";
22. Debug.Log(Singleton.Instance.Name);
23. }

　　5.3 使用单例模式(Singleton)获取唯一的游戏服务对象

　　　　5.3.1 游戏服务类的单例模式实现

在《P级阵地》中,因为PBaseDefenseGame类包含了游戏大部分的功能和操作,因此希望只产生一个对象,并提供方便的方法来取用PBaseDefenseGame功能,所以将该类运用单例模式



参与者说明:

　　PBaseDefenseGame

　　　　游戏主程序,内部包含了类型为PBaseDefenseGame的静态成员属性\_instance,作为该类唯一的对象

　　　　提供使用C# getter实现的静态成员方法Instance,用它来获取唯一的静态成员属性\_instance

　　BattleScene

　　　　PBaseDefenseGame类的客户端,使用PBaseDefenseGame.Instance来获取唯一的对象

　　　　5.3.2 实现说明

1. PBaseDefenseGame.cs
3. **public** **class** PBaseDefenseGame {
4. **private** **static** PBaseDefenseGame \_instance;
6. **public** **static** PBaseDefenseGame Instance {
7. get {
8. **if** (\_instance == null) {
9. \_instance = **new** PBaseDefenseGame();
10. }
11. **return** \_instance;
12. }
13. }
14. ...
15. **private** PBaseDefenseGame() { }
16. }
18. BattleState.cs
20. **public** **class** BattleScene: ISceneState {
21. ...
22. pubic override **void** StateBegin() {
23. PBaseDefenseGame.Instance.Initinal();
24. }
25. ...
26. }
28. SoldierClickScript.cs
30. **public** **class** SoldierOnClick: MonoBehavior {
31. ...
32. **public** **void** OnClick() {
33. PBaseDefenseGame.Instance.ShowSoldierInfo(Solder);
34. }
35. }

　　　　5.3.3 使用单例模式(Singleton)后的比较

　　　　5.3.4 反对使用单例模式(Singleton)的原因

　　5.4 少用单例模式(Singleton)时如何方便地引用到单一对象

让类具有技术功能来限制对象数量

1. ClassWithCounter.cs
3. **public** **class** ClassWithCounter {
4. **protected** **static** **int** m\_ObjCounter = 0;
5. **protected** **bool** m\_bEnable = **false**;
7. **public** ClassWithCounter() {
8. m\_ObjCounter++;
9. m\_bEnable = (m\_ObjCounter == 1) ? **true** : **false**;
11. **if** (m\_bEnalbe == **false**) {
12. Debug.LogError("当前对象数[" + m\_ObjCounter + "]超过1个!!");
13. }
15. **public** **void** Operator() {
16. **if** (m\_bEnable == **false**) {
17. **return**;
18. }
19. Debug.Log("可以执行");
20. }
21. }
23. SingletonTest.cs
25. **void** UnitTest\_ClassWithCounter() {
26. ClassWithCounter pObj = **new** ClassWithCounter();
27. pObj1.Operator();
29. ClassWithCounter pObj2 = **new** ClassWithCounter();
30. pObj2.Operator();
32. pObj1.Operator();
33. }

设置成为类的引用,让对象可以被取用

某个类的功能被大量使用时,可以将这个类对象设置为其他类中的成员,方便直接引用这些类.而这种实现方法是"依赖性注入"的方式之一,可以让被引用的对象不必通过参数传递的方式,就能被类的其他方法引用.按照设置的方式又可以分为"分别设置"和"指定类静态成员"两种

　　分别设置

在《P级阵地》中,PBaseDefenseGame是最常被引用的.虽然已经运用了单例模式(Singleton),但笔者还是以此来示范如何通过设置它成为其他类引用的方式,来减少对单例模式的使用

1. **public** **class** PBaseDefenseGame {
2. **public** **void** Initinal() {
3. m\_GameEventSystem = **new** GameEventSystem(**this**);
4. }
5. }
7. **public** abstract **class** IGameSystem {
8. **protected** PBaseDefenseGame m\_PBDGame = null;
9. **public** IGameSystem(PBaseDefenseGame PBDGame) {
10. m\_PBDGame = PBDGame;
11. }
12. }
14. **public** **class** CampSystem: IGameSystem {
16. **public** CampSystem(PBaseDefenseGame PBDGame): base(PBDGame) {
17. Initialize();
18. }
20. **public** **void** ShowCaptiveCamp() {
21. m\_PBDGame.ShowGameMsg("获得俘兵营");
22. }
23. }

　　指定类的静态成员

**A类的功能若需要使用到B类的方法,并且A类在产生其对象时具有下列几种情况:**

**1. 产生对象的位置不确定**

**2. 有多个地方可以产生对象**

**3. 生成的位置无法引用到**

**4. 有众多子类**

**当满足上述情况之一时,可以直接将B类对象设置为A类中的"静态成员属性", 让该类的对象都可以直接使用**

1. // PBaseDefenseGame.cs
3. **public** **class** PBaseDefenseGame {
4. **public** **void** Initinal() {
5. m\_StageSystem = **new** StageSystem(**this**);
6. // 注入其他系统
7. EnemyAI.SetStageSystem(m\_StageSystem);
8. }
9. }

**举例来说,敌方单位AI类(EnemyAI), 在运行时需要使用关卡系统(StageSystem)的信息,但EnemyAI对象产生的位置是在敌方单位建造者(EnemyBuilder)之下:**

1. EnemyBuilder.cs
3. **public** **class** EnemyBuilder: ICharacterBuilder {
4. **public** override **void** AddAI() {
5. EnemyAI theAI = **new** EnemyAI(m\_BuildParam.NewCharacter, m\_BuildParam.AttackPosition);
6. m\_BuildParam.NewCharacter.SetAI(theAI);
7. }
8. }

**按照"最少知识原则(LKP)",会希望敌方单位的建造者(EnemyBuilder)减少对其他无关类的引用.因此,在产生敌方单位AI(EnemyAI)对象时,敌方单位建造者(EnemyBuilder)无法将关卡系统(StageSystem)对象设置给敌方单位AI,这是属于上述"生成的位置无法引用到"的情况.所以,可以在敌方单位AI(EnemyAI)类中,提供一个静态成员属性和静态方法,让关卡系统(StageSystem)对象产生的当下,就设置给敌方单位AI(EnemyAI)类:**

1. **public** **class** EnemyAI: ICharacterAI {
2. **private** **static** StageSystem m\_StageSystem = null;
4. **public** **static** **void** SetStageSystem(StageSystem StageSystem) {
5. m\_StageSystem = StageSystem;
6. }
8. **public** ovrride **bool** CanAttackHeart() {
9. m\_StageSystem.LoseHeart();
10. **return** **true**;
11. }
12. }

　　使用类的静态方法

每当增加一个类名称就等同于又少了一个可以使用的全局名称,但如果是在类下增加"静态方法"就不会减少可使用的全局名称数量,而且还能马上增加这个静态类方法的"可视性"----就是全局都可以引用这个静态类方法.如果在项目开发时,不存在限制全局引用的规则,或者已经没有更好的设计方法时,使用"类静态方法"来获取某一系统功能的接口,应该就是最佳的方式了.它有着单例模式(Singleton)的第二个特性:方便获取对象

举例来说,在《P级阵地》中,有一个静态类PBDFactory就是按照这个概念去设计的.由于它在《P级阵地》中负责的是所有资源的产生,所以将其定义为"全局引用的类"并不违反这个游戏项目的设计原则.它的每一个静态方法都负责返回一个"资源生成工厂接口",注意,是"接口",所以在以后的系统维护更新中,是可以按照需求的改变来替换子类而不影响其他客户端:

1. **public** **static** **class** PBDFactory {
2. **private** **static** IAssetFactory m\_AssetFactory = null;
4. **public** **static** IAssetFactory GetAssetFactory() {
5. **if** (m\_AssetFactory == null) {
6. **if** (m\_bLoadFromResource) {
7. m\_AssetFactory = **new** ResourceAssetFactory();
8. } **else** {
9. m\_AssetFactory = **new** RemoteAssetFactory();
10. }
11. }
12. **return** m\_AssetFactory;
13. }
14. }

　　5.5 结论

单例模式(Singleton)的优点是: 可以限制对象的产生数量;提供方便获取唯一对象的方法.单例模式(Singleton)的缺点是容易造成设计思考不周和过度使用的问题,但并不是要求设计者完全不使用这个模式,而是应该在仔细设计和特定的前提之下,适当地采用单例模式(Singleton)

在《P级阵地》中,只有少数地方引用到单例类PBaseDefenseGame,而引用点可以视为单例模式(Singleton)优点的呈现

其他应用方式

　　网络在线游戏的客户端,可以使用单例模式(Singleton)来限制连接数,以预防误用而产生过多连接,避免服务器端因此失败

　　日志工具是比较不受项目类型影响的功能之一,所以可以设计为跨项目共享使用,此外,日志工具大多使用在调试或重要信息的输出上,而单例模式(Singleton)能让程序设计师方便快速地获取日志工具,所以是个不错的设计方式

第6章 游戏内各系统的整合----中介者模式(Mediator)

　　6.1 游戏系统之间的沟通

回顾单一职责原则(SRP)强调的是,将系统功能细分,封装,让每一个类都能各司其职,负责系统中的某一功能.因此,一个分析设计良好的软件或游戏,都是由一群子功能或子系统一起组合起来运行的

1. **public** **class** CampInfoUI {
2. CampSystem m\_CampSystem;
4. **public** **void** TrainSoldier(**int** SoldierID) {
5. m\_CampSystem.TrainSoldier(SoldierID);
6. }
7. }
9. **public** **class** CampSystem {
10. APSystem m\_ApSystem;
11. CharacterSystem m\_CharacterSystem;
12. }
14. **public** **class** APSystem {
15. GameStateInfoUI m\_StateInfoUI;
16. }
18. **public** **class** GameStateInfoUI {
19. APSystem m\_ApSystem;
20. }
22. **public** **class** CharacterSystem {
24. }

从上面的程序代码可以看出,所有系统在实现上都必须引用其他系统的对象,而这些被引用的对象都必须在功能执行前设置好,或者在调用方法时通过参数传入.但这些方法都会增加系统之间的依赖程度,也与最少知识原则(LKP)有所抵触

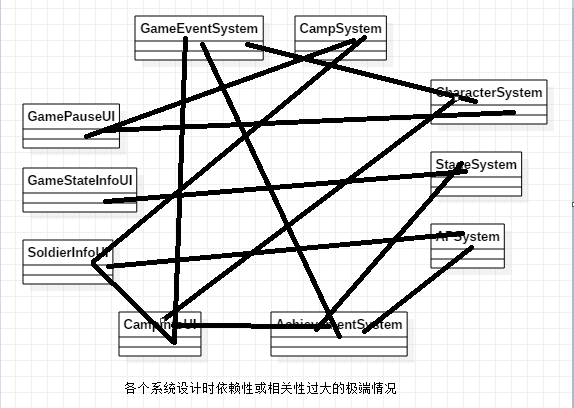
系统切分越细,则意味着系统之间的沟通越复杂,如果系统内部持续存在这样的连接,就会产生以下缺点:

　　1. 单一系统引入太多其他系统的功能,不利于单一系统的转换和维护

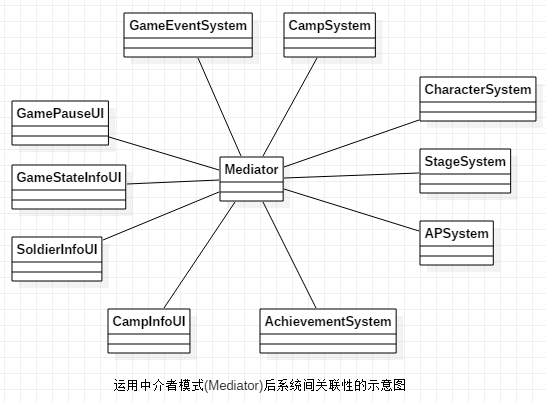
　　2. 单一系统被过多的系统所依赖,不利于接口的更改,容易牵一发而动全身

　　3. 因为需要提供给其他系统操作,系统的接口可能会过于庞大,不容易维护

要解决上述问题,可以使用中介者模式(Mediator)的设计方式



中介者模式(Mediator)简单解释的话,比较类似于中央管理的概念.建立一个信息集中的中心,任何子系统要与它的子系统沟通时,都必须先将请求交给中央单位,再由中央单位分派给对应的子系统.这种交给中央单位统一分配的方式,在物流业中已证明是最有效率的方式



　　6.2 中介者模式(Mediator)

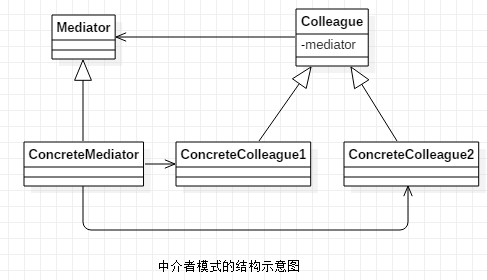
刚开始学习中介者模式(Mediator)时,会觉得为什么要如此麻烦,让两个功能直接调用就好了.但随着经验的累积,接触过许多项目,并且想要跨项目转换某个功能时就会知道,减少类之间的耦合度是一项很重要的设计原则.中介者模式(Mediator)在内部系统的整合上,扮演着重要的角色

　　　　6.2.1 中介者模式(Mediator)的定义

中介者模式(Mediator)在GoF中的说明是:

"定义一个接口用来封装一群对象的互动行为.中介者通过移除对象之间的引用,来减少它们之间的耦合度,并且能改变它们之间的互动独立性"

　　　　6.2.2 中介者模式(Mediator)的说明



参与者的说明如下:

　　Colleague(同事接口)

　　　　拥有一个Mediator属性成员,可以通过它来调用中介者的功能

　　ConcreateColleagueX(同事接口实现类)

　　　　实现Colleague界面的类,对于单一实现类而言,只会依赖一个Mediator接口

　　Mediator(中介者接口),ConcreteMediator(中介者接口实现类)

　　　　由Mediator定义让Colleague类操作的接口

　　　　ConcreteMediator实现类中包含所有ConcreteColleague的对象引用

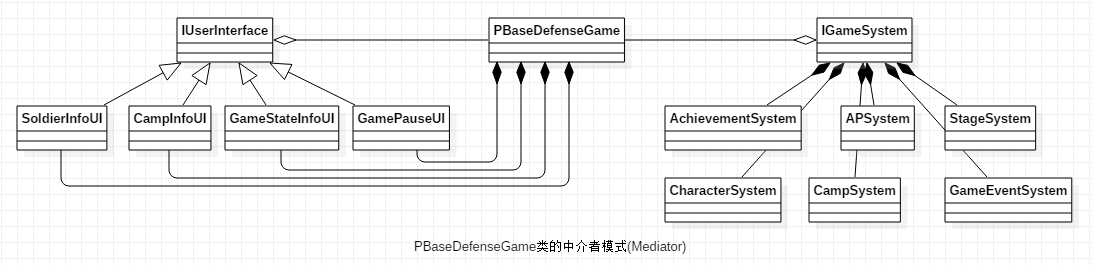
　　　　ConcreteMediator类之间的互动会在ConcreteMediator中发生

　　　　6.2.3 中介者模式(Mediator)的实现范例

1. Colleague.cs
3. **public** abstract **class** Colleague {
4. **protected** Mediator m\_Mediator = null;
6. **public** Colleague(Mediator theMediator) {
7. m\_Mediator = theMediator;
8. }
10. **public** abstract **void** Request(string Message);
11. }
13. ConcreteColleague1.cs
15. **public** **class** ConcreteColleague1: Colleague {
16. **public** ConcreteColleague1(Mediator theMediator): base(theMediator) {
18. }
20. **public** **void** Action() {
21. m\_Mediator.SendMessage(**this**, "Colleague1发除通知");
22. }
24. **public** override **void** Request(string Message) {
25. Debug.Log("ConcreteColleague1.Request: " + Message);
26. }
27. }
29. **public** **class** ConcreteColleague2: Colleague {
30. **public** ConcreteColleague2(Mediator theMediator): base(theMediator) {
32. }
34. **public** **void** Action() {
35. m\_Mediator.SendMessage(**this**, "Colleague2发除通知");
36. }
38. **public** override **void** Request(string Message) {
39. Debug.Log("ConcreteColleague2.Request: " + Message);
40. }
41. }
43. **public** abstract **class** Mediator {
44. **public** abstract **void** SendMesasge(Colleague theColleague, string Message);
45. }
47. **public** **class** ConcreteMediator: Mediator {
48. ConcreteColleague1 m\_Colleague1 = null;
49. ConcreteColleague2 m\_Colleague2 = null;
51. **public** **void** SetColleague1(ConcreateColleague1 theColleague) {
52. m\_Colleague1 = theColleague;
53. }
55. **public** **void** SetColleague2(ConcreteColleague2 theColleague) {
56. m\_Colleague2 = theColleague;
57. }
59. **public** override **void** SendMessage(Colleague theColleague, string Message) {
60. **if** (m\_Colleague1 == theColleague) {
61. m\_Colleage2.Request(Message);
62. }
64. **if** (m\_Colleague2 == theColleague) {
65. m\_Colleage1.Request(Message);
66. }
67. }
68. }
70. MediatorTest.cs
72. **void** UnitTest() {
73. ConcreteMediator pMediator = **new** ConcreateMediator();
75. ConcreateColleague1 pColleague1 = **new** ConcreateColleague1(pMediator);
76. ConcreateColleague2 pColleague2 = **new** ConcreteColleague2(pMediator);
78. pMediator.SetColleague1(pColleague1);
79. pMediator.SetColleague2(pColleague2);
81. pColleague1.Action();
82. pColleague2.Action();
83. }
85. ConcreteColleague2.Request:Colleague1发出通知
86. ConcreteColleague1.Request:Colleague2发出通知

　　6.3 中介者模式(Mediator)作为系统之间的沟通接口

　　　　6.3.1 使用中介者模式(Mediator)的系统架构



参与者的说明如下:

　　PBaseDefenseGame:

担任中介者角色,定义相关的操作界面给所有游戏系统与玩家界面来使用,并包含这些游戏系统和玩家界面的对象,同时负责相关的初始化流程

　　IGameSystem:

游戏系统的共同父类,包含一个指向PBaseDefenseGame对象的类成员,在其下的子类都能通过这个成员向PBaseDefenseGame发出需求

　　GameEventSystem, CampSystem,...:

负责游戏内的系统实现,这些系统之间不会互相引用及操作,必须通过PBaseDefenseGame来完成

　　IUserInterface:

玩家界面的共同父类,包含一个指向PBaseDefenseGame对象的类成员,在其下的子类都能通过这个成员向PBaseDefenseGame发出需求

　　SoldierInfoUI, CampInfoUI,...:

负责各玩家界面的实现,这些玩家界面与游戏系统之间不会互相引用及操作,必须通过PBaseDefenseGame来完成

　　　　6.3.2 实现说明

1. PBaseDefenseGame.cs
3. **public** **class** PBaseDefenseGame {
4. **private** GameEventSystem m\_GameEventSystem = null;
5. ...
7. **private** CampInfoUI m\_CampInfoUI = null;
8. ...
10. **public** **void** Initinal() {
11. m\_GameEventSyste = **new** GameEventSystem(**this**);
12. ...
14. m\_CampInfoUI = **new** CampInfoUI(**this**);
15. ...
17. EnemyAI.SetStageSystem(m\_StageSystem);
18. }
20. **public** **void** UpgrateSoldier() {
21. **if** (m\_CharacterSystem != null) {
22. m\_CharacterSystem.UpgrateSoldier();
23. }
24. }
26. ...
28. **public** **void** ShowCampInfo(ICamp camp) {
29. m\_CampInfoUI.ShowInfo(Camp);
30. m\_SoldierInfoUI.Hide();
31. }
32. ...
33. }

为了能够更灵活地处理游戏系统之间的沟通,《P级阵地》也实现了观察者模式(Observer),游戏事件系统(GameEventSystem)即观察者模式(Observer)的类.通过它能减少PBaseDefenseGame中增加接口方法,并且让信息的通知更有效率.

1. PBaseDefenseGame.cs
3. **public** **void** RegisterGameEvent(ENUM\_GameEvent emGameEvent, IGameEventObserver Observer) {
4. m\_GameEventSystem.RegisterObserver(emGameEvent, Observer);
5. }
7. **public** **void** NotifyGameEvent(ENUM\_GameEvent emGameEvent, System.Object Param) {
8. m\_GameEventySystem.NotifySubject(emGameEvent, Param);
9. }
11. IGameSystem.cs
13. **public** abstract **class** IGameSystem {
14. **protected** PBaseDefenseGame m\_PBDGame = null;
15. **public** IGameSystem(PBaseDefenseGame PBDGame) {
16. m\_PBDGame = PBDGame;
17. }
18. ...
19. }
21. IUserInterface.cs
23. **public** abstract **class** IUserInterface {
24. **protected** PBaseDefenseGame m\_PBDGame = null;
25. **public** IUserInterface(PBaseDefenseGame PBDGame) {
26. m\_PBDGame = PBDGame;
27. }
28. ...
29. }
31. StageSystem.cs
33. **public** **class** StageSystem: IGameSystem {
34. **public** StageSystem(PBaseDefenseGame PBDGame): base(PBDGame) {
35. Initialize();
36. }
37. ...
38. }

　　　　6.3.3 使用中介者模式(Mediator)的优点

不会引入太多其他的系统

从上面《P级阵地》的实现来看,每一个游戏系统和玩家界面除了会引用与本身功能相关的类外,无论是对外的信息获取还是信息的传递,都只通过PBaseDefenseGame类对象来完成.这使得每一个游戏系统,玩家界面对外的依赖度缩小到只有一个类(PBaseDefenseGame)

系统被依赖的程度也降低

每一个游戏系统或玩家界面,也只在PBaseDefenseGame类的方法中调用.所以,当游戏系统或玩家界面有所更动时,受影响的也仅仅局限于PBaseDefenseGame类,因此可以减少系统维护的难度

　　　　6.3.4 实现中介者模式(Mediator)时的注意事项

由于PBaseDefenseGame类担任中介者(Mediator)的角色,再加上各个游戏系统和玩家界面都必须通过它来进行信息交换即沟通,所以要注意的是,PBaseDefenseGame类会因为担任过多中介者的角色而容易出现"操作接口爆炸"的情况.因此,在实现上,我们可以搭配其他设计模式来避免发生这种情况.在前面的说明中,我们提及的游戏事件系统(GameEventSystem),其作用就是用来提供更好的信息传递方式,以减轻PBaseDefenseGame类的负担

　　6.4 中介者模式(Mediator)面对变化时

如何应对变化

当游戏系统或玩家界面需要新增功能,且该功能需要由外界提供信息才能完成时,可以先在PBaseDefenseGame类中增加获取信息的方法,之后再通过PBaseDefenseGame类来获取信息完成新的功能.这样一来,项目的修改可以保持在两个类或最多3个类的更改,而不会影响任何类的"依赖性"

如何面对新增

当需要新增加游戏系统或玩家界面时,只要是继承自IGameSystem或IUserInterface的游戏系统和玩家界面,都可以直接加入PBaseDefenseGame的类成员中,并通过现有的接口进行实现或增加功能.这时候项目更改的幅度,可能只是新增一个程序文件和修改一个PBaseDefenseGame类而已,不太容易影响到其他系统或接口

　　6.5 结论

与其他模式(Pattern)的合作

PBaseDefenseGame类在《P级阵地》中,除了是中介者模式(Mediator)中的中介者(Mediator)之外,也是外观模式(Facade)中对外系统整合接口的主要类,并且还运用单例模式(Singleton)来产生唯一的类对象

此外,为了降低PBaseDefenseGame类有接口过大的问题,其子系统"游戏事件系统"(GameEventSystem)专门运用观察者模式(Observer)来解决游戏系统之间,对于信息的产生和通知的需求,减少这些信息和通知的方法充满在PBaseDefenseGame类之中

**在进行分析设计时,集合多种设计模式是良好设计常见的方式,如何将所学设计模式融合并适当地运用,才是设计模式之道**

其他应用方式

网络引擎

　　连线管理系统与网络数据封包管理系统之间,如何可以通过中介者模式(Mediator)进行沟通,那么就能轻松地针对连线管理系统抽换所使用的通信方式(TCP或UDP)

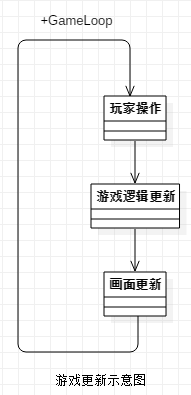
数据库引擎

　　内部可以分成数个子系统, 有专门负责数据库连接的功能和产生数据库操作数据的功能,两个子功能之间的沟通可以通过中介者模式(Mediator)来进行,让两者之间不相互依赖,方便抽换另一个子系统

第7章 游戏的主循环----Game Loop

　　7.1 GameLoop由此开始

游戏开发时特有的设计模式----游戏循环



　　7.2 怎么实现游戏循环(Game Loop)

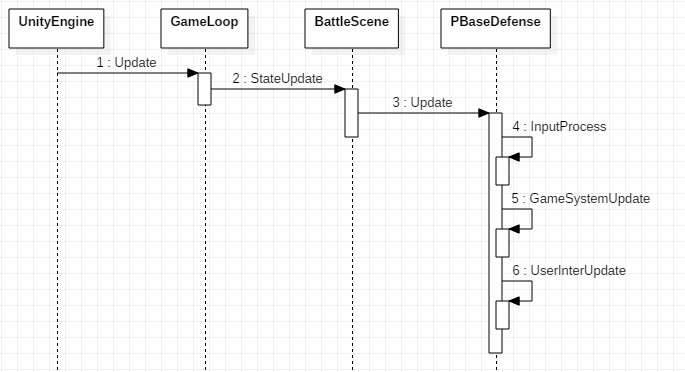
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　7.3 在Unity3D中实现游戏循环

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　7.4 P级阵地的游戏循环

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



　　7.5 结论

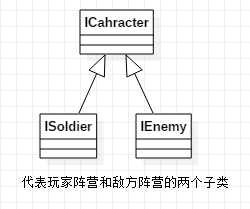
每一款游戏在实现时,都会有专用于这款游戏的"玩家操作"和"游戏逻辑更新"这两项特殊需求.因此,在PBaseDefenseGame类内实现"游戏循环"是比较好的设计方式,这样可以提高PBaseDefenseGame类整个移植道其他项目的可能性

虽然《P级阵地》中大部分的游戏功能和用户界面类,都采用"不"继承MonoBehaviour的方式来运行,但对于会出现在场景中的每个游戏3D角色上,还是会搭配使用组件(继承MonoBehaviour),所以每一个脚本组件还是会按照Unity3D引擎的流程去操作每一个游戏对象(GameObject)

第8章 角色系统的设计分析

　　8.1 游戏角色的架构

　　8.2 角色类的规划



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

第9章 角色与武器的实现----桥接模式(Bridge)

　　9.1 角色与武器的关系

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

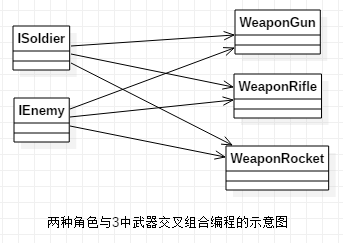
将两种角色与3种武器交叉组合,然后以上述方式实现,会存在以下两个缺点:

　　1. 每个继承自ICharacter角色接口的类,在重新定义Attack方式时,都必须针对每一种武器来实现(显示特效和播放音效),或者进行额外的公式计算.所以当要新增角色类时,也要在新的子类种重复编写相同的程序代码

　　2. 当要新增武器类型时,所有角色子类种的Attack方法,都必须修改,针对新的武器类型编写新的对应程序代码.这样会增加维护的难度,使得武器类型不容易增加

一般来说,上述的情况可以视为两个类群组交互使用所引发的问题

GoF的设计模式中,桥接模式(Bridge)可以用来解决上述实现方式的缺点



　　9.2 桥接模式(Bridge)

笔者认为,在GoF的23种设计模式中,桥接模式是最好应用但也是最难理解的,尤其是它的定义不长,其中关键的"抽象与实现分离(Decouple an abstraction from its implementation)",常让设计师花费许多时间,才能慢慢了解它背后所代表的原则

　　　　9.2.1 桥接模式(Bridge)的定义

桥接模式(Bridge),在GoF中的解释是:

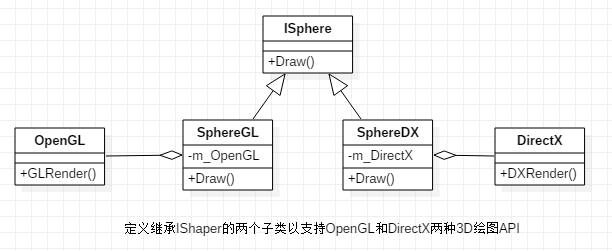
"将抽象与实现分离,使二者可以独立地变化"

多数人会以为这是"只依赖接口而不依赖实现"原则的另外一个解释:

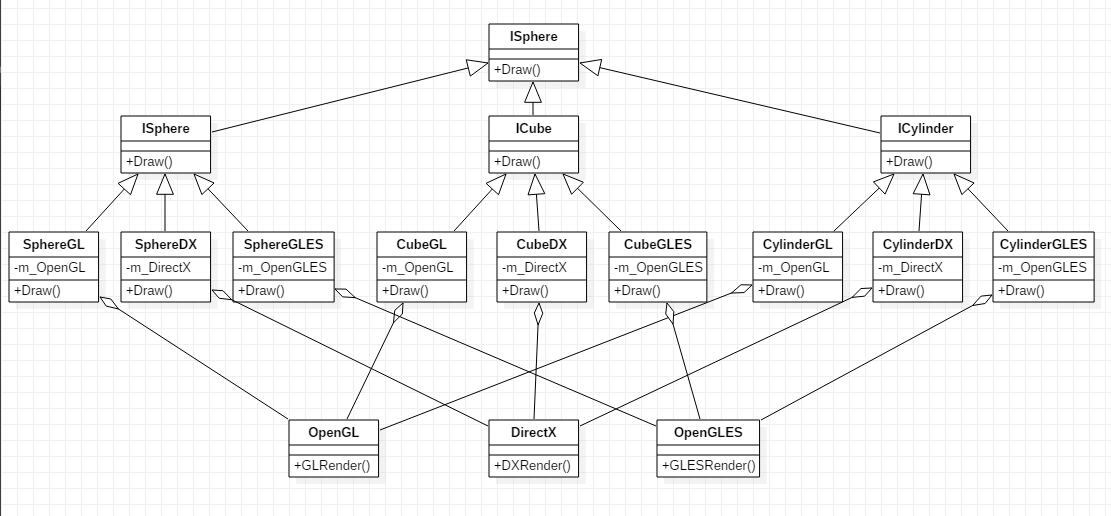
"定义一个接口类,然后将实现的部分在子类中完成"



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

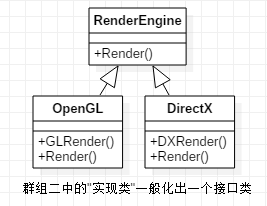


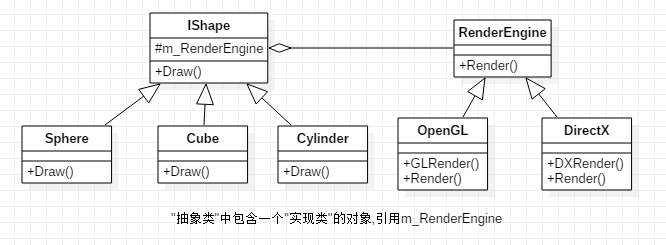
　　　　9.2.2 桥接模式(Bridge)的说明

如果要避免被限制在只能以"继承实现"来完成功能实现,可考虑使用桥接模式(Bridge).从先前的例子中可以看出,基本上这是两个类组群之间,关系呈现"交叉组合汇编"的情况:

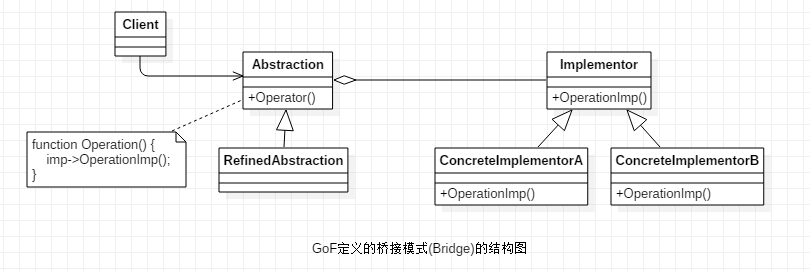
群组一的"抽象类"指的是将对象或功能经"抽象"之后所定义出来的类接口,并通过子类继承的方式产生多个不同的对象或功能.例如上述的"形状"类,其用途是用来描述一个有"形状"的对象应该具备的功能和操作方式.所以,这个群组只负责增加"抽象类",不负责实现"接口定义的功能

群组二的"实现类"指的是这些类可以用来实现"抽象类"中所定义的功能.例如上述例子中的OpenGL引擎类和DirectX引擎类,它们可以用来实现"形状"类中所定义的"绘出"功能,能将形状绘制道屏幕上.所以,这个群组只负责增加"实现类"





继承"抽象类"的子类需要实现功能时,只要通过"实现类"的对象引用m\_RenderEngine来调用实现功能即可.这样一来,就真正让"抽象与实现分离",也就是"抽象不与实现绑定",让"球体"或者"立方体"这种抽象概念的类,不再通过产生不同子类的方式去完成特定的"实现方式"(OpenGL或DirectX),将"抽象类群组"与"实现类群组"彻底分开



参与者的说明如下:

Abstraction(抽象体接口)

　　拥有指向Implementor的对象引用

　　定义抽象功能的接口,也可作为子类调用实现功能的接口

RefinedAbstraction(抽象体实现,扩充)

　　继承抽象体并调用Implementor完成实现功能

　　扩充抽象体的接口,增加额外的功能

Implementor(实现体接口)

　　定义实现功能的接口,提供给Abstraction(抽象体)使用

　　接口功能可以只有单一的功能,真正的选择则再由Abstraction(抽象体)的需求加以组合应用

ConcreteImplementA/B(实现体)

　　实际完成实现体接口上所定义的方法

　　　　9.2.3 桥接模式(Bridge)的实现范例

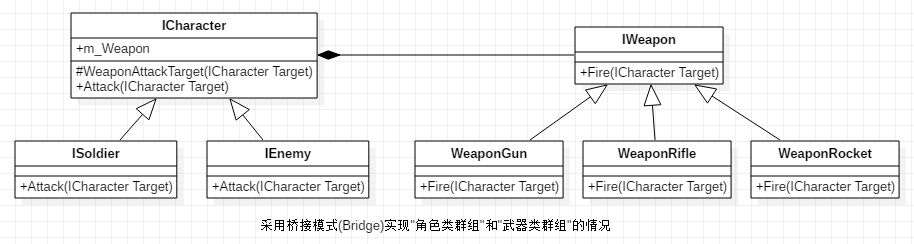
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　9.3 使用桥接模式(Bridge)实现角色与武器接口

定义哪个群组类是"抽象类",哪个又是"实现类"并不容易.不过,如果从两个类群组的交叉合作开始分析,那么对于桥接模式(Bridge)的运用就不会那么困难了

　　　　9.3.1 角色与武器接口设计

桥接模式(Bridge)除了能够应用在"抽象与实现"的分离之外,还可以应用在: "当两个群组因为功能上的需求,想要连接合作,但又希望两组类可以各自发展不受彼此影响时"



参与者的说明如下:

　　ICharacter: 角色的抽象接口拥有一个IWeapon对象引用,并且在接口中声明了一个武器攻击目标WeaponAttackTarget()方法让子类可以调用,同时要求继承的子类必须在Attack()中重新实现攻击目标的功能

　　ISoldier,IEnemy: 双方阵营单位,实现攻击目标Attack()时,只需要调用父类的WeaponAttackTarget()方法,就可以使用当前装备的武器攻击对手

　　IWeapon: 武器接口,定义游戏中对于武器的操作和使用方法

　　WeaponGun, WeaponRifle, WeaponRocket: 游戏中可以使用的3种武器类型的实现

　　　　9.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　　　9.3.3 使用桥接模式(Bridge)的优点

运用桥接模式(Bridge)后的ICharacter(角色接口)就是群组一"抽象类",它定义了"攻击目标“功能,但真正实现"攻击目标"功能的类,则是群组二IWeapon(武器接口)"实现类".对于ICharacter及其继承类都不必理会IWeapon群组的变化,尤其是游戏开发后期可能增加的武器类型.二对于ICharacter来说,它面对的至于IWeapon这个接口类,相对地,IWeapon类群组也不必理会角色类群组内的新增或修改,让两个群组之间的耦合度降到最低

　　　　9.3.4 实现桥接模式(Bridge)的注意事项

　　9.4 桥接模式(Bridge)面对变化时

应用了桥接模式(Bridge)的角色与武器系统,在后续的游戏系统设计上,增加了不少的弹性和灵活度.当需要新增武器类型时,继承IWeapon类并重新实现抽象方法后,就可让角色系统装备使用

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　9.5 结论

桥接模式(Bridge)可以将两个群组有效地分离,让两个群组彼此互相不受影响.这两个群组可以是"抽象定义"与"功能实现",也可以是两个需要交叉合作后才能完成某项任务的类

与其他模式(Pattern)的合作

《P级阵地》将使用建造者模式(Builder)负责产生游戏中的游戏角色对象,当角色产生时会设置需要装备的武器,而设置武器的操作则是由角色接口中的方法来完成

其他应用方式

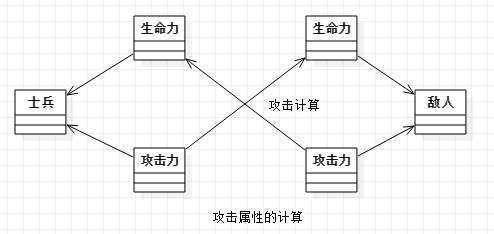
两组类群组需要搭配使用的实现方式,常见于游戏设计中,例如:

游戏角色可以驾驶不同的行动载具,如汽车,飞机,水上摩托......

奇幻类型游戏的角色可以施展法术,除了多样的角色之外,"法术”本身也是另一个复杂的系统,火系法术,冰系法术,......,远程法术,近战法术,补血法术......,想额外加上使用限制的话,就必须使用桥接模式(Bridge)让角色与法术类群组妥善结合

第10章 角色的属性----策略模式(Strategy)

　　10.1 角色属性的计算需求



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

在这3个操作方法中,都针对不同的角色类型进行了相对应的属性计算,但这样的实现方式有以下缺点:

　　每个方法都针对角色类型进行属性计算,所以这3个方法依赖角色类型,当新增角色类型时,必须修改这3个方法,因此会增加维护的难度

　　同一类型的计算规则分散在角色类Character中,不易阅读和了解

对于这些因角色不同而有差异的计算公式,该如何重新设计才能解决上述问题呢?GoF的策略模式为我们提供了解答

　　10.2 策略模式(Strategy)

因条件的不同而需要有所选择时,刚入门的程序设计师会使用If else或多组的if elseif else 来完成需求,或者使用switch case 语句来完成.当然,这是因为入门的程序书籍大多是这样建议的,而且也是最快完成实现的方式.对于小型项目或快速开发验证用的项目而言,或许可以使用比较快速的条件判断方式来实现.但若遇到具有规模或产品化(需要长期维护)项目时,最好还是选择策略模式来完,因为这将有利于项目的维护

　　　　10.2.1 策略模式(Strategy)的定义

GoF对策略模式的解释是  
"定义一组算法,并封装每个算法,让它们可以彼此交换使用.策略模式让这些算法在客户端使用它们时能更加独立"

就"策略"一词来看,有当发生“某情况”时要做出什么“反应”的含义.从生活中可以举出许多在相同的环境下针对不同条件,要进行不同计算方式的例子:

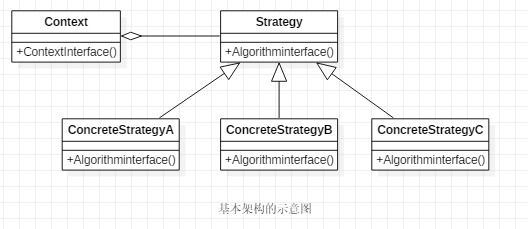
　　当购买商品满399时,要加送100元折价券

　　当购买商品满699时,要加送200元折价券

　　...

在策略模式中,这些不同的计算方式就是所谓的"算法",而这些算法中的每一个都应该独立出来,将计算细节加以封装隐藏,并让它们成为一个算法类群组.客户端只需要根据情况来选择对应的算法类即可,至于计算方式及规则,客户端不需要去理会

　　　　10.2.2 策略模式(Strategy)的说明



参与者的说明如下:

　　Strategy(策略接口类)　　提供"策略客户端“可以使用的方法

　　ConcreteStrategyA~ConcreteStrategyC(策略实现类)　　不同算法的实现

　　Context(策略客户端)　　拥有一个Strategy类的对象引用,并通过对象引用获取想要的计算结果

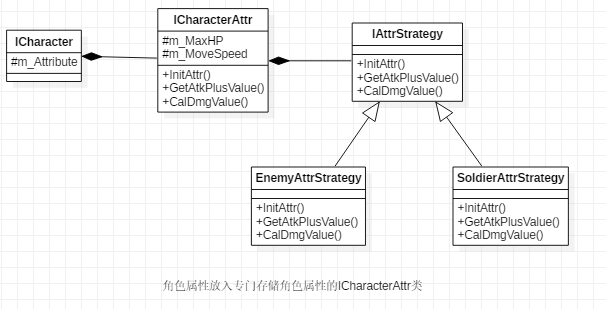
　　　　10.2.3 策略模式(Strategy)的实现范例

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　10.3 使用策略模式(Strategy)实现攻击计算

许多人在想到要应用策略模式时,常常会遇到不知从何切入的情况.究其原因,通常是不知道如何在不使用if else或switch case语句的情况下,将这些计算策略配对调用.其实,有时候处理方式是必须利用重构方法或搭配其他的设计模式来完成的,也就是先利用重构方法或搭配其他的设计模式将这些条件判断语句从程序代码中删除,再将策略模式加入到项目的设计方案中.否则,最常见的策略模式应用方式,还是会在if else或 switch case语句中调用对应的策略类对象

　　　　10.3.1 攻击流程的实现



参与者的说明如下

　　ICharacterAttr　　声明游戏内使用的角色属性,访问方法和声明攻击流程中所需要的方法,并拥有一个IAttrStrategy对象,通过该对象来调用真正的计算公式

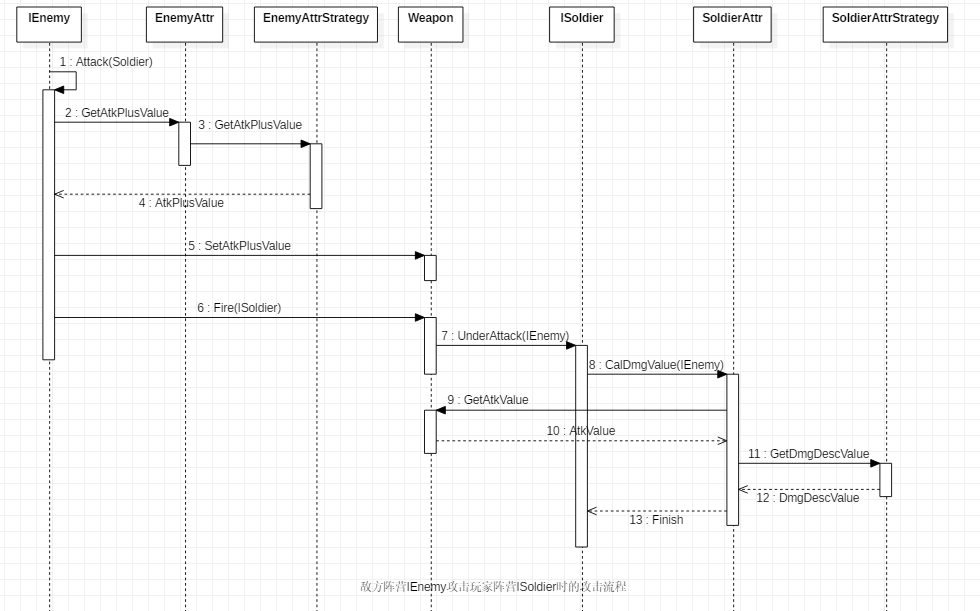
　　IAttrStrategy　　声明角色属性计算的接口方法,用来把ICharacterAttr与计算方法分离,让ICharacterAttr可轻易地更换计算策略

　　EnemyAttrStrategy　　实现敌方阵营单位在攻击流程中所需要的各项公式的计算

　　SoldierAttrStrategy　　实现玩家阵营单位在攻击流程中所需要的各项公式的计算

　　　　10.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



　　　　10.3.3 使用策略模式(Strategy)的优点

让角色属性变得好维护

对于改进后的角色类ICharacter来说,将角色属性有关的属性以专属类ICharacterAttr来取代,可以使以后角色属性变动时,不会影响到角色类ICharacter.此外,随着游戏需求的复杂化,加入更多的角色属性是可预期的,所以让角色属性集中在同一个类下管理,将有助于后续游戏项目的维护,也可以减少角色类ICharacter的更改及降低复杂度

不必再针对角色类型编写程序代码

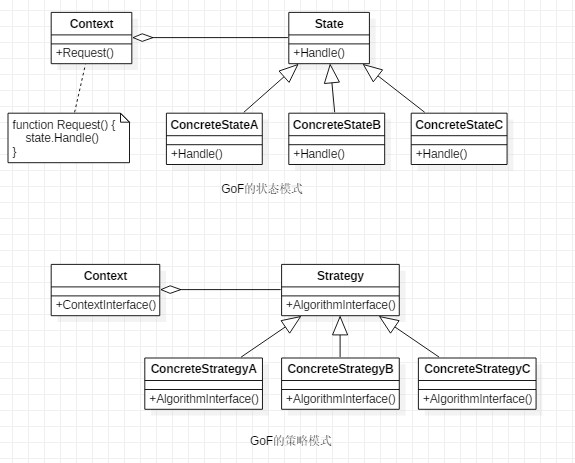
通过ICharacterAttr与其子类的分工,将双方阵营的属性放置于不同的类中.对于角色类ICharacter而言,使用ICharacterAttr的对象引用,完全不用考虑将使用哪一个子类对象,避免了使用switch case语句的编写方式及后续可能产生的维护问题.当有新的阵营类产生时,角色类ICharacter并不需要有任何改动

计算公式的替换更为方便

在游戏开发的过程中,属性计算公式是最常变换的.运用策略模式后的ICharacterAttr,更容易替换公式,除了可以保留原来的计算公式外,还可以让所有公式同时并存,并且能自由切换.

　　　　10.3.4 实现策略模式(Strategy)的注意事项

与状态模式的差别



如果读者仔细分析状态模式与策略模式的类结构图,可能会发现两者看起来非常相似

两者都被GoF归类在行为模式分类下,都是由一个Context类来维护对象引用,并借此调用提供功能的方法.就笔者过去的实践经验,对于这两种模式,可归类出下面几点差异,供读者作为以后选择时的引用依据:

　　State是在一群状态中进行切换,状态之间有对应的和连接的关系; Strategy则是由一群没有任何关系的类所组成,不知彼此的存在

　　State受限于状态机的切换规则,在设计初期就会定义所有可能状态,就算后期追加也需要和现有的状态有所关联,而不是想加入就加入;Strategy是由封装计算算法而形成的一种设计模式,算法之间不存在任何依赖关系,有新增的算法就可以马上加入或替换

　　10.4 策略模式(Strategy)面对变化时

　　10.5 结论

将复杂的计算公式从客户端独立出来成为一个群组,之后客户端可以按情况来决定使用的计算公式策略,既提高了系统应用的灵活程度,也强化了系统中对所有计算策略的维护方式.让后续开发人员很容易找出相关计算公式的差异,同时修改点也会缩小到计算公式本身,也不会影响到使用的客户端

与其他模式的合作

在第15章中,《P级阵地》将使用建造者模式负责产生游戏中的角色对象.当角色产生时,会需要设置该角色要使用的"角色属性",这部分将有各阵营的建造者来完成.若策略模式搭配其他设计模式一起应用的话,就可以不必使用if else或switch case 来选择要使用的策略类

其他应用方式

有些角色扮演型游戏的属性系统,会使用转换计算的方式来获取角色最终要使用的属性.例如:玩家看到角色界面上只会显示体力,力量,敏捷......,但实际在运用攻击计算时,这些属性会被再转换为生命力,攻击力,闪避率.......而之所以会这样设计的原因在于,该游戏有"职业"的设置,对于不同的"职业",在计算转换时会有不同的转换方式,利用策略模式将这些转换公式独立出来是比较好的

游戏角色操作载具时,会引用角色当前对该类型载具的累积时间,并将之转换为"操控性",操控性越好,就越能控制该载具,而获取操控性的计算公式,也可以利用策略模式将其独立出来

网络在线型游戏往往需要玩家注册账号,注册账号有多种方式,例如OpenID,自建账号,随机产生等.通过策略模式可以将不同账号的注册方式独立为不同的登录策略.这样做,除了可以强化项目的维护,也可以方便转换到不同的游戏项目上,增加重复利用的价值

第11章 攻击特效与击中反应----模板方法模式(Template Method)

　　11.1 武器的攻击流程

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　11.2 模板方法模式(Template Method)

程序代码中的"流程",有时候不太容易观察出来,尤其是当原有的程序代码还没有经过适当重构.有个很好的判断技巧,如果程序设计师发现更新一段程序代码之后,还有另一段程序代码也使用相同的"演算流程",且实现的内容不太一样,那么这两段程序代码就可以用模板方法模式加以重写

　　　　11.2.1 模板方法模式(Template Method)的定义

GoF对于模板方法模式的定义是:

"在一个操作方法中定义算法的流程,其中某些步骤由子类完成.模板方法模式让子类在不变更原有算法流程的情况下,还能够重新定义其中的步骤"

从上述的定义来看,模板方法模式包含以下两个概念:

　　1. 定义一个算法的流程，即是很明确地定义算法的每一个步骤,并写在父类的方法中,而每一个步骤都可以是一个方法的调用

　　2. 某些步骤由子类完成,为什么父类不自己完成,却要由子类去实现呢?

　　　　定义算法的流程中,某些步骤需要由执行时"当下的环境"来决定

　　　　定义算法时,针对每一个步骤都提供了预设的解决方案,但有时候会出现"更好的解决方法",此时就需要让这个更好的解决方案,能够在原有的架构中被使用

以下提供几个例子跟大家说明:

以面包的配方和制作方法为例,大概是这样写的:

食材: A1.xxx, A2.xxx, A3.xxxx ... B1.yyy, B2.yyy

步骤:

　　1. 将材料A1~A5混合在一起搅拌至光滑

　　2. 至于密闭空间醒面30~50分钟

　　3. 分成5等份, 整形滚圆再静置越10~20分钟

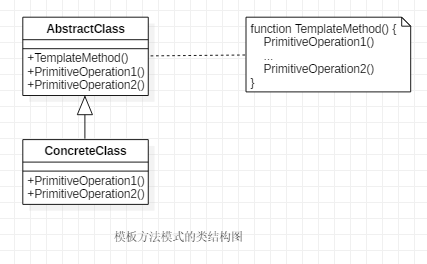
　　4. 包入B1~B3内馅, 整形成长条形状

　　5. 置于密闭空间做二次发酵,约30~50分钟

　　6. 烤培:越热180°,进炉降温165°,烘烤15~20分钟至表面上色即可

如果将面包配方和制作方法看成是"算法的流程",那么其中的1~6就是每一个步骤,而且每一个步骤遵循着一定的先后顺序.做过面包的读者应该可以了解,面包要好吃,发酵的时间长度是关键,而温度,湿度等都会影响发酵所需的时间.所以,上述制作面包的步骤中,第2,3,5项是需要实现面包的人按照当天的环境情况来决定发酵的时间,这也是为什么食谱上常出现xx~xx分钟,而不是明确告诉你一定要多少分钟,也就是GoF定义中所提示的"定义算法的流程中,某些步骤需要由执行时'当下的环境'来决定"

　　　　11.2.2 模板方法模式(Template Method)的说明



参与者的说明如下:

　　AbstractClass(算法定义类)

　　　　定义算法架构的类

　　　　可以在某个操作方法(Template Method)中定义完整的流程

　　　　定义流程中会调用到方法(PrimitiveOperation), 这些方法将由子类重新实现

　　ConcreteClass(算法步骤的实现类)

　　　　重新实现父类中定义的方法,并可按照子类的执行情况反应步骤实际的内容

　　　　11.2.3 模板方法模式(Template Method)的实现范例

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

public abstract class AbstractClass {

public void TemplateMethod() {

PrimitiveOperation1();

PrimitiveOperation2();

}

protected abstract void PrimitiveOperation1();

protected abstract void PrimitiveOperation2();

}

public class ConcreteClassA: AbstractClass {

protected override void PrimitiveOperation1() {

Debug.Log("ConcreteClassA.PrimitiveOperation1");

}

protected override void PrimitiveOperation2() {

Debug.Log("ConcreateClassA.PrimitiveOperation2");

}

}

public class ConcreteClassB: AbstractClass {

protected override void PrimitiveOperation2() {

Debug.Log("ConcreteClassB.PrimitiveOperation1");

}

protected override void PrimitiveOperation2() {

Debug.Log("ConcreteClassB.PrimitiveOperation2");

}

}

void UnitTest() {

AbstractClass theClass = new ConcreteClassA();

theClass.TemplateMethod();

theClass = new ConcreteClassB();

theClass.TemplateMethod();

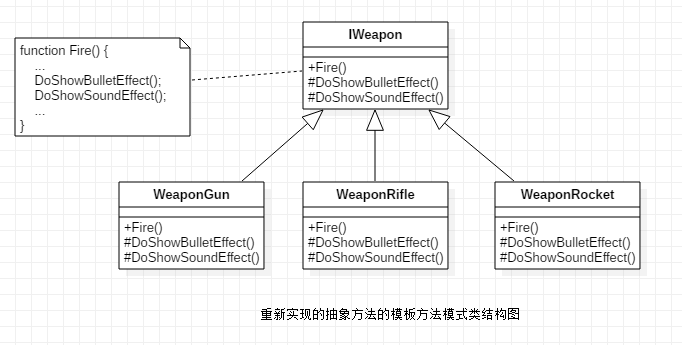
}

[复制代码](javascript:void(0);)

　　11.3 使用模板方法模式实现攻击与击中流程

很难找出程序代码中相同的演算流程,是程序设计放弃使用模板方法模式的原因之一;另一种常见的情况是,有时这些演算流程中会有一些小变化,也是因为这些小变化导致程序设计放弃使用模板方法模式.而那个小变化可能是,A流程中有一个if判断语句用以决定是否执行某项功能,但在B流程中却没有这个if判断语句.当笔者在遇到这种情况时,会连同这个if判断语句一起设置为步骤的一部分,只是重构后的B类(B流程)不去重新定义这一步骤所调用的方法

　　　　11.3.1 攻击与击中流程的实现



参与者的说明如下:

　　IWeapon　　在攻击目标Fire方法中定义流程, 也就是要执行的各个步骤,并将这些步骤声明为抽象方法

　　WeaponGun, WeponRifle, WeaponRocket　　实现IWeapon类中需要重新实现的抽象方法

　　　　11.3.2 实现说明

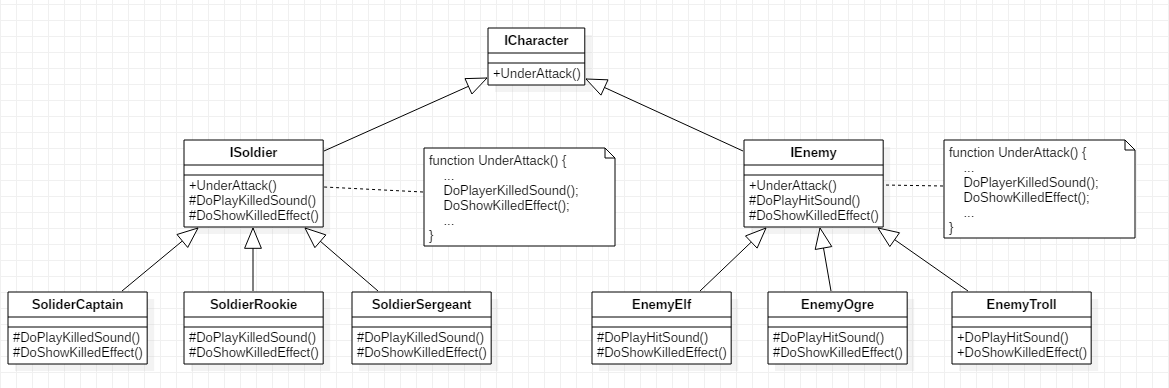
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　　　11.3.3 运用模板方法模式(Template Method)的优点

在IWeapon类中,将"攻击目标Fire方法"重新修改后,攻击目标的"算法"只被编写一次,需要变化的部分,则由实现的子类负责,这样一来,原本需要在子类中"重复实现算法"的缺点就不会再出现了

　　　　11.3.4 修改击中流程的实现

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



　　11.4 模板方法模式(Template Method)面对变化时

　　11.5 结论

运用模板方法模式的优点是,将可能重复出现的"算法流程",从子类提升到父类中,减少重复的发生,并且也开放子类参与算法中各个步骤的执行或优化.但如果"算法流程"开放太多的步骤,并要求子类必须全部重新实现的话,反而会造成实现的困难,也不容易维护

其他应用方式

　　奇幻类角色扮演游戏,对于游戏角色要施展一个法术时,会有许多特定的检查条件,如魔力是否足够,是否还在冷却时间内,对象是否在法术施展范围内等.如果这些检查条件会按照施展法术的类型而有所不同,那么就可以使用模板方法模式将检查流程固定下来,真正检查的功能则交给各法术子类去实现.另外,一个法术的施展流程和击中计算也可以如同本章范例一样,将流程固定下来,细节交给各法术子类去实现

　　在线游戏的角色登录,也可以使用模板方法模式将登录流程固定下来,例如: 显示登录画面,选择登录方法,输入账号密码,向Server请求登录等,然后让登录功能的子类去重新实现其中的步骤,另外,也可以实现不同的登录流程样板来对应不同的登录方式

第12章 角色AI----状态模式(State)

　　12.1 角色的AI

　　12.2 状态模式(State)

　　12.3 使用状态模式(State)实现角色AI

　　　　12.3.1 角色AI的实现

　　　　12.3.2 实现说明

　　　　12.3.3 使用状态模式(State)的优点

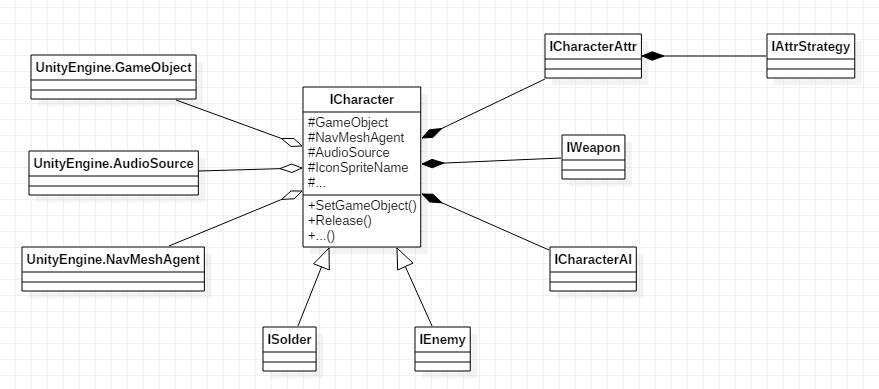
　　　　12.3.4 角色AI执行流程

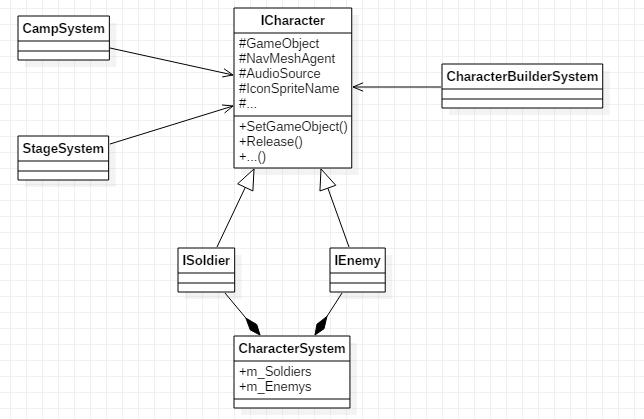
　　12.4 状态模式(State)面对变化时

　　12.5 结论

第13章 角色系统

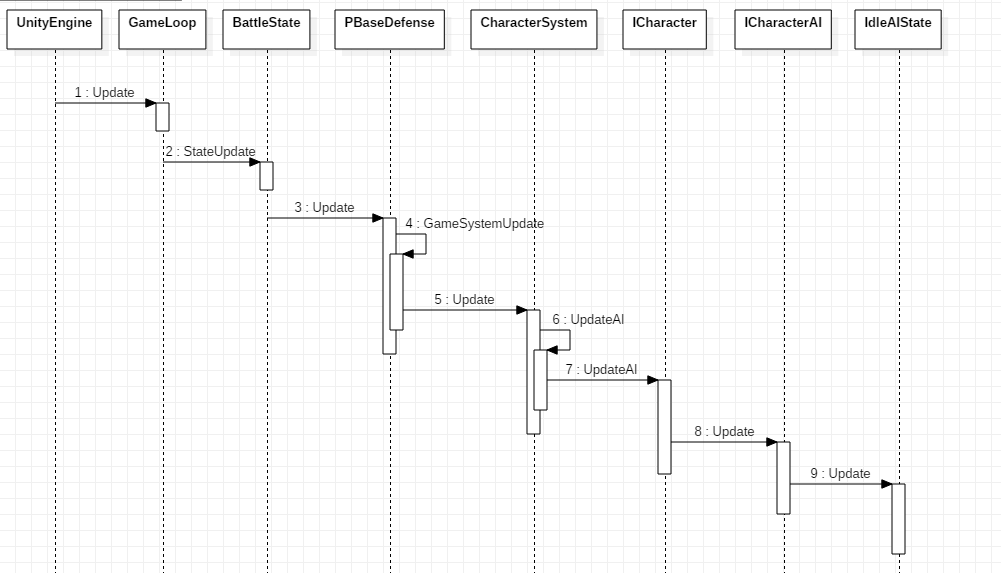
　　13.1 角色类





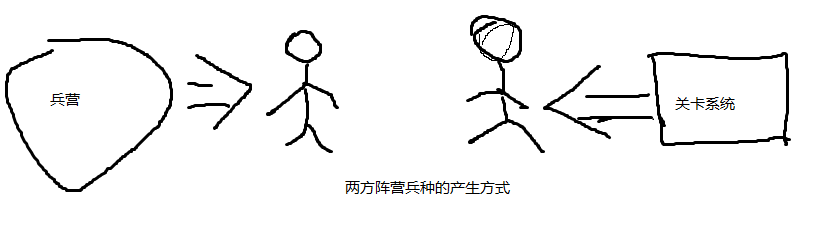
　　13.2 游戏角色管理系统

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



第14章 游戏角色的产生----工厂方法模式(Factory Method)

　　14.1 产生角色



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

在两个类中,共声明了6个方法来产生不同的角色对象.在实践中,声明功能相似过高的方法会有不易管理的问题,而且这一次实现的6个方法中,每个角色对象的组装流程重复性太高.此外,将产生相同类群组对象的实现,分散在不同的游戏功能下不易管理和维护.所以,是否可以将这些方法都集合在一个类下实现,并且以更灵活的方式来决定产生对象的类呢?GoF的工厂方法模式为上述问题提供了答案

　　14.2 工厂方法模式(Factory Method)

提到工厂,大多数人的概念可能是可以大量生产东西的地方,并且是以有组织,有规则的方式来生产东西.它会有多条生产线,每一条生产线都有特殊的配置,专门用来生产特定的东西.没错,工厂方法模式就是用来搭建专门生产软件对象的地方,而且这样的软件工厂,也能针对特定的类配置特定的组装流程,来满足客户端的要求

　　　　14.2.1 工厂方法模式(Factory Method)的定义

GoF对工厂方法模式的解释是:

"定义一个可以产生对象的接口,但是让子类决定要产生哪一个类的对象.工厂方法模式让类的实例化程序延迟到子类中实施"

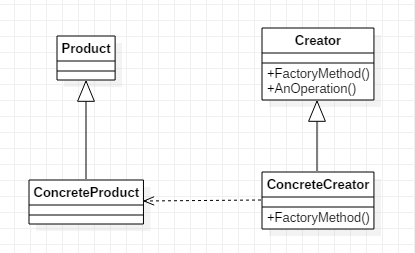
工厂方法模式就是将类产生对象的流程集合管理的模式.集合管理带来的好处是:

　　1. 能针对对象产生的流程指定规则

　　2. 减少客户端参与对象生成的过程,尤其是对于那种类对象生产过程过于复杂的,如果让客户端操作对象的组装过程,将使得客户端与该类的耦合度(即依赖度)过高,不利于后续的项目维护

工厂方法模式是先定义一个产生对象的接口,之后让它的子类去决定产生哪一种对象,这有助于将庞大的类群组进行分类.

　　　　14.2.2 工厂方法模式(Factory Method)的说明



参与者的说明如下:

　　Product(产品类)　　定义产品类的操作接口,而这个产品将由工厂产生

　　ConcreteProduct(产品实现)　　实现产品功能的类,可以不只定义一个产品实现类,这些产品实现类的对象都会由ConcreteCreator(工厂实现类)产生

　　Creator(工厂类)　　定义能产生Product(产品类)的方法: FactoryMethod

　　ConcreteCreator(工厂实现类)　　实现FactoryMethod, 并产生指定的ConcreteProduct(产品实现)

　　　　14.2.3 工厂方法模式(Factory Method)的实现范例

在实现工厂方法模式的选择上并非是固定的,而是按照程序语言设计的特性来决定有多少种实现方式.因为C#支持泛型程序设计,所以有4种实现方式

第一种方式: 有子类产生

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

第二种方式: 在FactoryMethod增加参数

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

第三种方式: Creator泛型类

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

第四种方式: FactoryMethod泛型方法

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　14.3 使用工厂方法模式(Factory Method)产生角色对象

当类的对象产生时,若出现下列情况:

　　需要复杂的流程

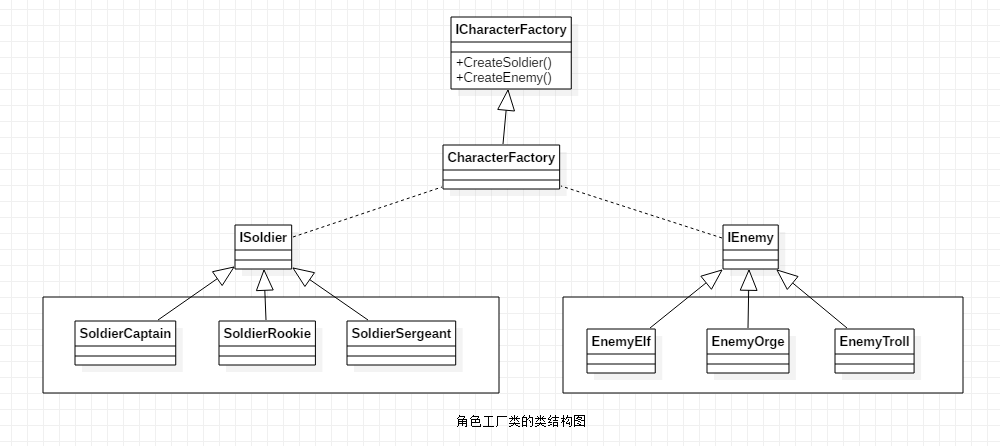
　　需要加载外部资源,如从网络,存储设备,数据库

　　有对象上限

　　可重复使用

建议使用工厂方法模式来实现一个工厂类,而这个工厂类内还可以搭配其他的设计模式,让对象的产生与管理更有效率

　　　　14.3.1 角色工厂类



参与者说明如下:

　　ICharacterFactory:　　负责产生角色类ICharacter的工厂接口,并提供两个工厂方法来产生不同阵营的角色对象: CharacterSoldier负责产生玩家阵营的角色对象;CharacterEnemy负责产生敌方阵营的角色对象

　　CharacterFactory:　　继承并实现ICharacter工厂接口的类,其中实现的工厂方法是实际产生对象的地方

　　ISoldier, SoldierCaption......:　　由工厂类产生的"产品", 在《P级阵地》中为玩家角色

　　IEnemy, EnemyElf......:　　由工厂类产生的另一项"产品",在《P级阵地》中为敌方角色

　　　　14.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

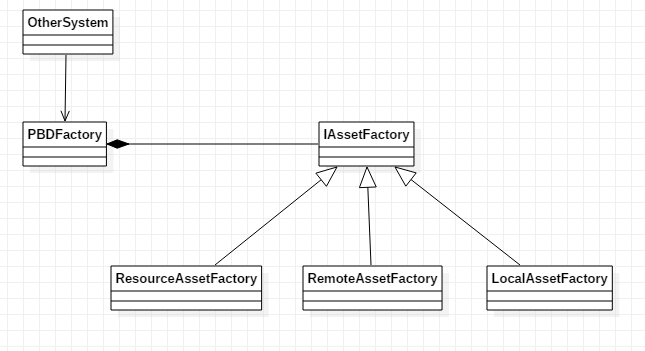
　　　　14.3.3 使用工厂方法模式(Factory Method)的优点

角色工厂类CharacterFactory将"角色类群组"产生对象的实现,都整合到两个工厂方法下,并将有关的程序从客户端删除,同时降低了客户端与"角色产生过程"的耦合度(或成为依赖度).此外,角色生成后的后续设置功能(给武器,设置属性,设置AI等),也都在同一个地方实现,让开发人员能快速了解类之间的关联性及设置的先后顺序

　　　　14.3.4 工厂方法模式(Factory Method)的实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　14.4 工厂方法模式(Factory Method)面对变化时



　　14.5 结论

工厂方法模式的优点是,将类群组对象的产生流程整合于同一个类下实现,并提供唯一的工厂方法,让项目内的"对象产生流程"更加独立.不过,当类群组过多时,无论使用哪种方式,都会出现工厂子类爆量或switch case 语句过长的问题,这是美中不足的地方

与其他模式的合作

　　角色工厂(CharacterFactory)中,产生不同阵营的角色时,会搭配建造者模式(Builder)的需求,将需要的参数设置给各角色的建造者

　　本地资源加载工厂(ResourceAssetFactory)若同时要求系统性能的优化,可使用代理者模式来优化本地加载性能

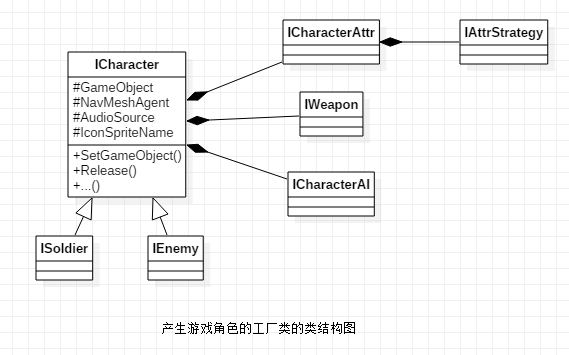
　　属性产生工厂(AttrFactory)可使用享元模式(Flyweight)来减少重复对象的产生

　　其他应用方式

就如同本章的重点,如果系统实现人员想要将对象的产生及相关的初始化工作集中在一个地方完成,那么都可以使用工厂方法模式来完成,换句话说,就是工厂方法模式的应用层面非常广泛

第15章 角色的组装----建造者模式(Builder)

　　15.1 角色功能的组装



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　15.2 建造者模式(Builder)

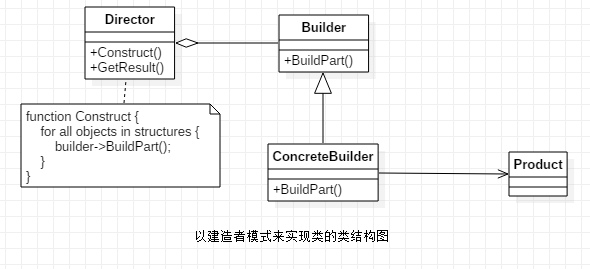
工厂类是将生产对象的地点全部集中到一个地点来管理,但是如何在生产对象的过程中,能够更有效率并且更具弹性,则需要搭配其他的设计模式.建造者模式就是常用来搭配使用的模式之一

　　　　15.2.1 建造者模式(Builder)的定义

在GoF中对建造者模式的定义是:

"将一个复杂对象的构建流程与它的对象表现分离出来,让相同的构建流程可以产生不同的对象行为表现"

　　　　15.2.2 建造者模式(Builder)的说明



参与者的说明如下:

　　Director(建造指示者)

　　　　负责对象构建时的"流程分析安排"

　　　　在Constructor方法中,会明确定义对象组装的流程,即调用Builder接口方法的顺序

　　Builder(功能实现者接口)

　　　　定义不同的操作方法将"功能分开来实现"

　　　　其中的每一个方法都是用来提供给某复杂对象的一部分功能,或是提供设置规则

　　ConcreteBuilder(功能实现者)

　　　　Builder的具体实现,实现产出功能的类

　　　　不同的ConcreteBuilder(功能实现者)可以产出不同的功能,用来实现不同对象的行为表现和功能

　　Product(产品)

　　　　代表最终完成的复杂对象,必须提供方法让Builder类可以将各部位功能设置给它

　　　　15.2.3 建造者模式(Builder)的实现范例

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

public class Product {

private List<string> m\_Part = new List<string.();

public Product() { }

public void AddPart(string Part) {

m\_Part.Add(part);

}

public void ShowProduct() {

Debug.Log(Part);

}

}

public class Director {

private Product m\_Product;

public Director() { }

public void Construct(Builder theBuilder) {

m\_Product = new Product();

theBuilder.BuildPart1(m\_Product);

theBuilder.BuildPart2(m\_Product);

}

public Product GetResult() {

return m\_Product;

}

}

public abstract class Builder {

public abstract void BuildPart1(Product theProduct);

public abstract void BuildPart2(Product theProduct);

}

public class ConcreteBuilderA: Builder {

public override void BuildPart1(Product theProduct) {

theProduct.AddPart("ConcreteBuilderA\_Part1");

}

public override void BuildPart2(Product theProduct) {

theProduct.AddPart("ConcreteBuilderA\_Part2");

}

}

public class ConcreteBuilderB: Builder {

public override void BuildPart1(Product theProduct) {

theProduct.AddPart("ConcreteBuilderB\_Part1");

}

public override void BuildPart2(Product theProduct) {

theProduct.AddPart("ConcreteBuilderB\_Part2");

}

}

void UnitTest() {

Director theDirector = new Director();

Product theProduct = null;

theDirector.Construct(new ConcreteBuilderA());

theProduct = theDirector.GetResult();

theProduct.ShowProduct();

theDirector.Construct(new ConcreteBuilderB());

theProduct = theDirector.GetResult();

theProduct.ShowProduct();

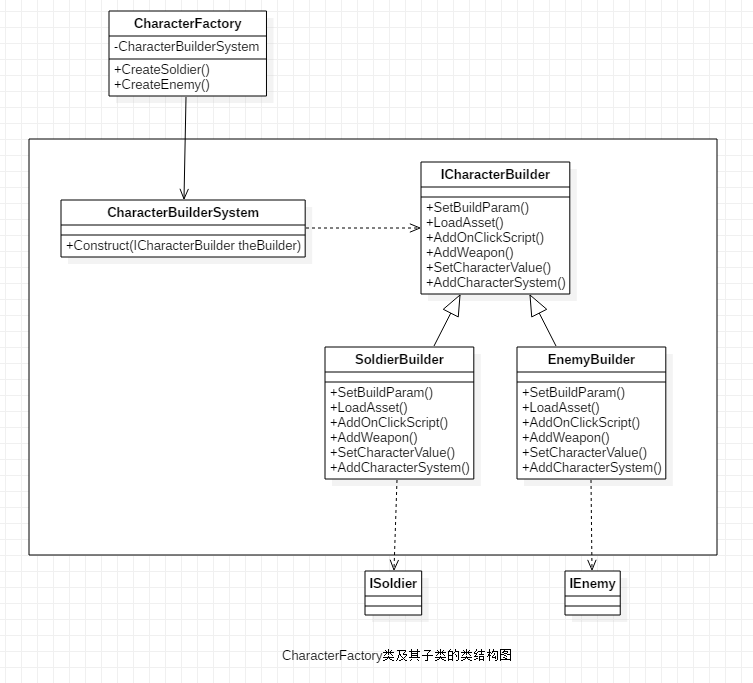
}

[复制代码](javascript:void(0);)

　　15.3 使用建造者模式(Builder)组装角色的各项功能

角色的组装算是游戏实现上最复杂的功能之一.每款游戏遇到这个部分时,都要针对程序代码不断地重构,调整,修正,防呆......, 原因是"角色"是游戏的卖点之一.

　　　　15.3.1 角色功能的组装



参与者的说明如下:

　　CharacterBuilderSystem: 角色建造者系统负责《P级阵地》中双方角色构建时的装配流程.它是一个"IGameSystem游戏系统",因为角色构建完成后,还需要通知其他游戏系统,所以将其加入已经具有中介者模式(Mediator)的PBaseDefenseGame类中,方便与其他游戏功能沟通

　　ICharacterBuilder: 定义游戏角色功能的组装方法,包含3D模型,武器,属性,AI等功能

　　SoldierBuilder: 负责玩家阵营角色功能的产生并设置给玩家角色

　　EnemyBuilder: 负责地方阵营角色功能的产生并设置给敌方角色

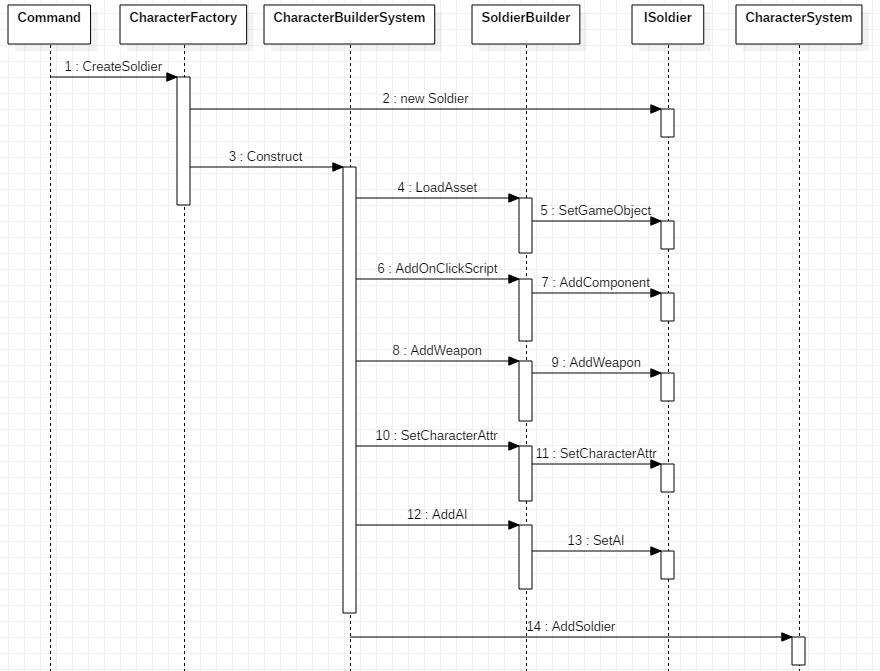
　　　　15.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　　　15.3.3 使用建造者模式(Builder)的优点

在重构后的角色工厂中,之简单负责角色的"产生",而复杂的功能组装工作则交由新增加的角色建造者系统来完成.运用建造者模式的角色建造者系统,将角色功能的"组装流程"给独立出来,并以明确的方法调用来实现,这有助于程序代码的阅读和维护.而各个角色的功能装备任务,也交由不同的类来实现,并使用接口方法操作,将系统之间的耦合度(即依赖度)降低.所以当实现系统有任何变化时,也可以使用替换实现类的方式来应对

　　　　15.3.4 角色建造者的执行流程



　　15.4 建造者模式(Builder)面对变化时

　　15.5 结论

建造者模式的优点是,能将复杂对象的"产生流程"与"功能实现"拆分后, 让系统调整和维护变得更容易.此外,在不需要更新实现者的情况下,调整产生流程的顺序就能完成装备线的更改,这也是建造者模式的另一优点

与其他模式的合作

　　建造者模式在实现过程中, 大多利用《P级阵地》的工厂类获取所需的功能组件,而这两种生成模式的相互配合,也是章范例的重点之一

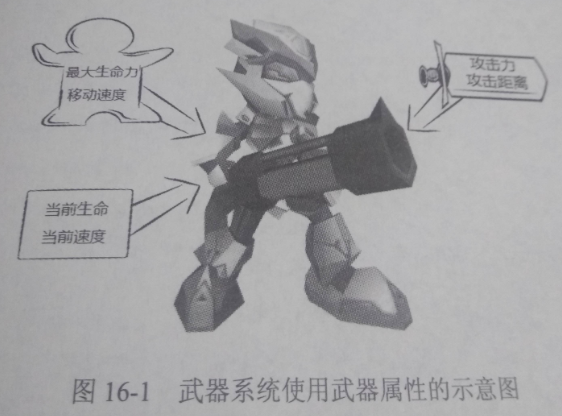
其他应用方式

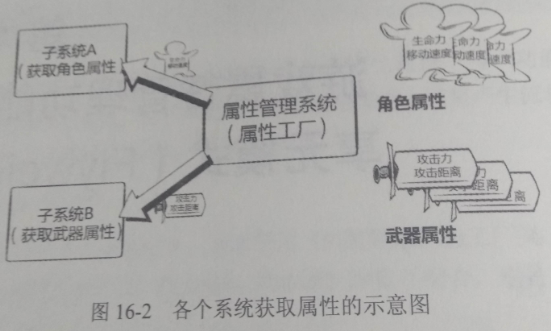
　　在奇幻类型的角色扮演游戏中,设计者为了增加法术系统的声光效果,在施展法术时,大多会分成不同的段落来呈现法术特效.例如发射箭的法术吟唱特效,发射时的特效,法术在进行时的特效,击中对手时的特效,对手被打中时的特效,最后消失时的特效.有时为了执行性能的考虑,会在施展法术时,就将所有特效全部准备完成.这个时候就可以利用建造者模式将所有特效组装完成

　　游戏的用户界面就如同一般的网页或App,有时也会有复杂的版面配置和信息显示.利用建造者模式可以将界面的呈现,分成不同的区域或内容来实现,让界面也可以有"功能装组"的应用方式

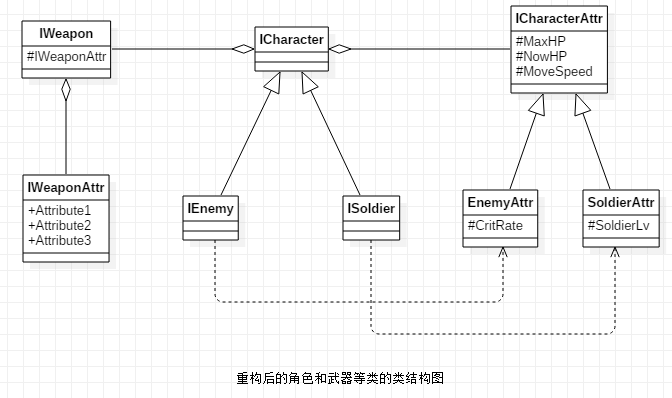
第16章 游戏属性管理功能----享元模式(Flyweight)

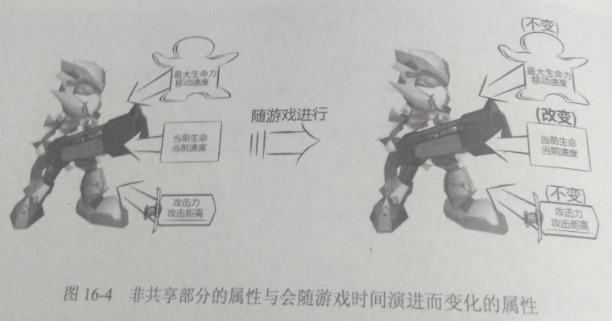
　　16.1 游戏属性的管理





https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code





　　16.2 享元模式(Flyweight)

享元模式是用来解决"大量且重复的对象"的管理问题,尤其是程序设计师最常忽略的"虽然小但却大量重复的对象". 随着计算机设备的升级,程序设计师渐渐遗忘了在内存受限制环境下,对每一个字节都很计较的程序编写方式.但近几年来,由于移动设备App的兴起,有大小限制的内存环境又成为设计师必须考虑的设计条件之一,善用享元模式(Flyweight)可以解决大部分对象共享的问题

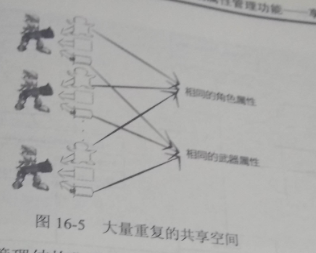
　　　　16.2.1 享元模式(Flyweight)的定义

GoF中享元模式(Flyweight)的定义是:

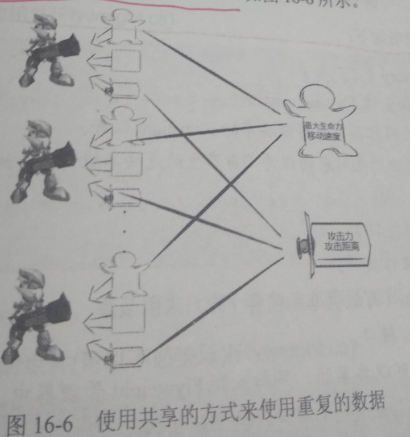
"使用共享的方式,让一大群小规模对象能更有效地运行"

定义中的两个重点: "共享"与"一大群小规模的对象"

首先,"一大群小规模对象"指的是: 虽然有时候类的组成很简单,可能只有几个类型为int的类成员,但如果这些类成员的属性是相同而且可以共享的,那么当系统产生了一大群类的对象时,这些重复的部分就都是浪费的,因为它们只需要存在一份即可



而"共享"指的是使用"管理结构"来设计信息的存取方式,让可以被共享的信息,只需要产生一份对象,而这个对象能够被引用到其他对象中

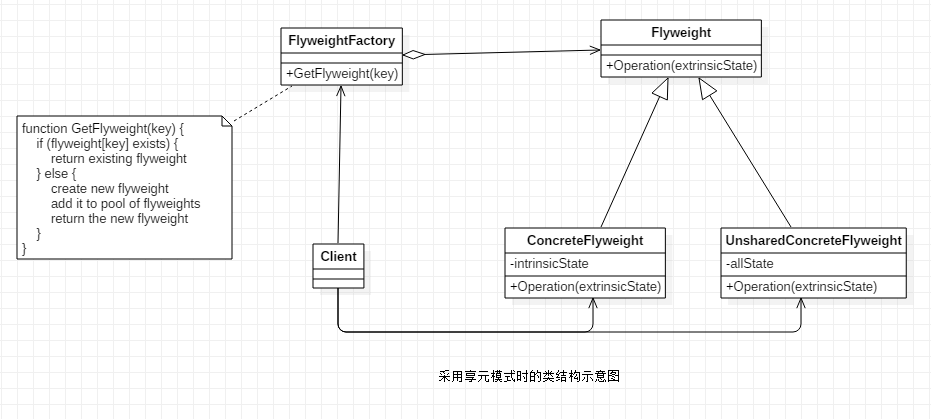


但必须注意的是,既然可以被多个对象"共享",那么对于共享对象的"修改"就必须加以限制,因为被多个对象共享之后,任何更改共享对象中的属性,都可能导致其他引用对象的错误

因此在设计上,对象中那些"只能读取而不能写入"的共享部分被称为"内在(intrinsic)状态",就如前一节中提到的最大生命力(MaxHP),移动速度(MoveSpeed),攻击力(Atk),攻击距离(Range)这些值.而对象中"不能被共享"的部分,如当前的生命力(NowHP),等级(LV),暴击率(CritRate)等,这些属性会随着游戏运行的过程而变化,则称为"外在(extrinsic)状态".

享元模式提供的解决方案是: 产生对象时,将能够共享的"内在(intrinsic)状态"加以管理,并且将属于各对象能自由更改的"外部(extrinsic)状态”也一起设置给新产生的对象中

　　　　16.2.2 享元模式(Flyweight)的说明



GoF参与者的说明如下:

　　FlyweightFactory(工厂类)

　　　　负责产生和管理Flyweight的组件

　　　　内部通常使用容器类来存储共享的Flyweight组件

　　　　提供工厂方法产生对应的组件,当产生的是共享组件时,就加入到Flyweight管理容器内

　　Flyweight (组件接口)

　　　　定义组件的操作接口

　　ConcreteFlyweight (可以共享的组件)

　　　　实现Flyweight接口

　　　　产生的组件是可以共享的,并加入到Flyweight管理器中

　　UnsharedConcreteFlyweight (不可以共享的组件)

　　　　实现Flyweight接口,也可以选择不继承自Flyweight接口

　　　　可以定义为单独的组件,不包含任何共享资源

　　　　也可以将一些共享组件定义为类的成员,称为内部状态;并另外定义其他不被共享的成员,作为外部状态使用

　　　　16.2.3 享元模式(Flyweight)的实现范例

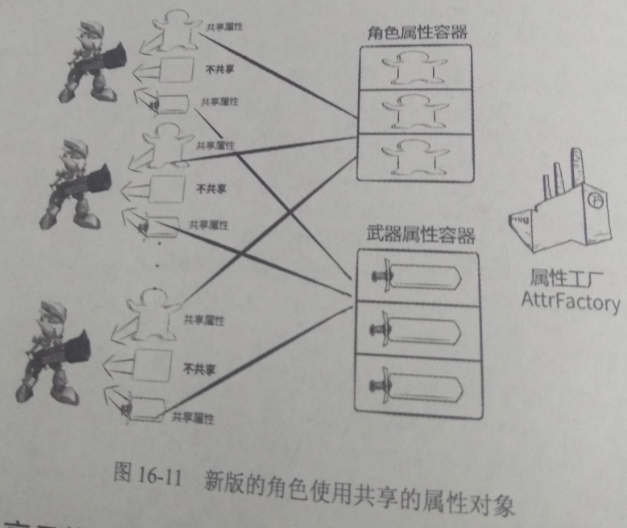
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　16.3 使用享元模式(Flyweight)实现游戏

　　　　16.3.1 SceneState的实现

　　　　16.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



　　　　16.3.3 使用享元模式(Flyweight)的优点

新版运用享元模式的属性工厂AttrFactory, 将属性设置集以更简短的格式呈现,免去了使用switch case的一长串语句,方便企划人员阅读和设置.此外,因为共享属性的部分(BaseAttr),每一个编号对应的属性对象,在整个游戏执行中只会产生一份,不像旧方法那样会产生重复的对象而增加内存的负担,对于游戏性能有所提升

　　　　16.3.4 享元模式(Flyweight)的实现说明

就笔者的经验来说,享元模式在游戏开发领域中,最常被应用到的地方就是属性系统.每一款游戏无论规模大小,都需要属性系统协助调整游戏平衡,如角色等级属性,装备属性,武器属性,宠物属性,道具属性等,而每一种属性设置数据又可能多达上百或上千之多,当这些属性设置都成为对象并存在游戏之中时,即符合了享元模式定义中所说的"一大群小规模对象",每一项属性可能只包含3,4个字段,也可能包含多达数十个字段,若不采用"共享”的方式管理,很容易造成系统的问题和实现上的困难,应用上也会产生相关的问题.

　　16.4 享元模式(Flyweight)面对变化时

　　16.5 结论

将有可能散步在程序代码中,零碎的游戏属性对象进行统一管理,是享元模式应用在游戏开发领域带来的好处之一

与其他模式合作

每一个阵营的角色建造函数,在设置武器属性和角色属性时,都会通过属性工厂来获取属性,而这些属性则是使用享元模式产生的,在游戏的执行过程中只会存在一份

其他应用方式

在射击游戏中,画面上出现的子弹或导弹,大多会使用"对象"的方式来代表.而为了让游戏系统能够有效地产生和管理这些对象,导弹对象,可使用享元模式来建立子弹对象池,让其他游戏对象也使用对象池内的子弹,减少因为重复处理产生子弹对象,删除子弹对象所导致的性能损失

第17章 Unity3D的界面设计----组合模式(Composite)

　　17.1 玩家界面设计

　　17.2 组合模式(Composite)

　　　　17.2.1 组合模式(Composite)的定义

　　　　17.2.2 组合模式(Composite)的说明

　　　　17.2.3 组合模式(Composite)的实现范例

　　　　17.2.4 分了两个子类但是要使用同一个操作界面

　　17.3 Unity3D游戏对象的分层式管理功能

　　　　17.3.1 游戏对象的分层管理

　　　　17.3.2 正确有效地获取UI的游戏对象

　　　　17.3.3 游戏用户界面的实现

　　　　17.3.4 兵营界面的实现

　　17.4 结论

第18章 兵营系统及兵营信息显示

　　18.1 兵营系统

　　18.2 兵营系统的组成

　　18.3 初始兵营系统

　　18.4 兵营信息的显示流程

第19章 兵营训练单位----命令模式(Command)

　　19.1 兵营界面上的命令

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　19.2 命令模式(Command)

在本节使用软件开发作为范例说明设计模式之前,我们先举个较为生活化的例子来说明命令模式.例如,在餐厅用餐就是命令模式的一种表现,当餐厅的前台服务人员接收到客人的点餐之后,就会将餐点内容记载在点餐单(命令)上,这张点餐单(命令)就会随着其他客人的点餐单(命令)一起排入厨房的"待做"列表(命令管理器)内.厨房内的厨师(功能提供者)根据先到先做的原则,将点餐单(命令)上的内容一个个制作(执行)出来.当然,如果餐厅不计较的话,那么等待很久的客人,也可以选择不继续等待(取消命令),改去其他餐厅用餐

　　　　19.2.1 命令模式(Command)的定义

GoF对于命令模式(Command)的定义如下:

"将请求封装成为对象,让你可以将客户端的不同请求参数化,并配合队列,记录,复原等方法来执行请求的操作."

上述定义可以简单分成两部分来看待:

　　请求的封装

　　请求的操作

请求的封装

所谓的请求,简单来说就是某个客户端组件,想要调用执行某种功能,而这个某种功能是被实现在某个类中.

请求的操作

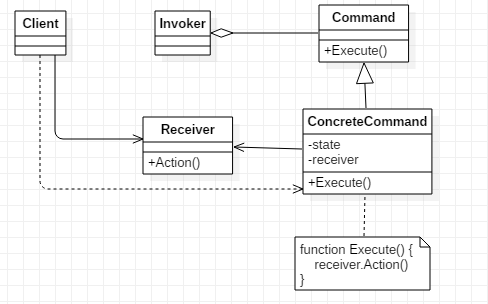
当请求可以被封装成一个对象时,那么这个请求对象就可以被操作,例如:

　　存储: 可以将请求对象放入一个数据结构中进行排序,排对,搬移,删除,暂缓执行等操作

　　记录: 当某一个请求对象被执行后,可以先不删除,将其移入"已执行"数据容器内,通过查看"已执行"数据容器的内容,就可以知道系统过去执行命令的流程和轨迹

　　复原: 延续上一项记录功能,若系统针对每项请i去命令实现了"反向"操作时,可以将已执行的请求复原,这在大部分的文字编辑软件和绘图编辑软件中是很常见的

　　　　19.2.2 命令模式(Command)的说明



GoF参与者的说明如下:

　　Command(命令界面): 定义命令封装后具备的操作界面

　　ConcreteCommand(命令实现): 实现命令封装和界面,会包含每一个命令的参数和Receiver(功能执行者)

　　Receiver(功能执行者): 被封装在ConcreteCommand(命令实现)类中,真正执行功能的类对象

　　Client(客户端/命令发起者): 产生命令的客户端,可以视情况设置命令给Receiver(功能执行者)

　　Invoker(命令管理者): 命令对象的管理容器或是管理类,并负责要求每个Command执行其功能

　　　　19.2.3 命令模式(Command)的实现范例

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

上述范例看似颇为简单,也正因为如此,让命令模式在实现上的弹性非常大,也出现许多变化的形式.在实际分析时,可以着重在"命令对象"的"操作行为"加以分析:

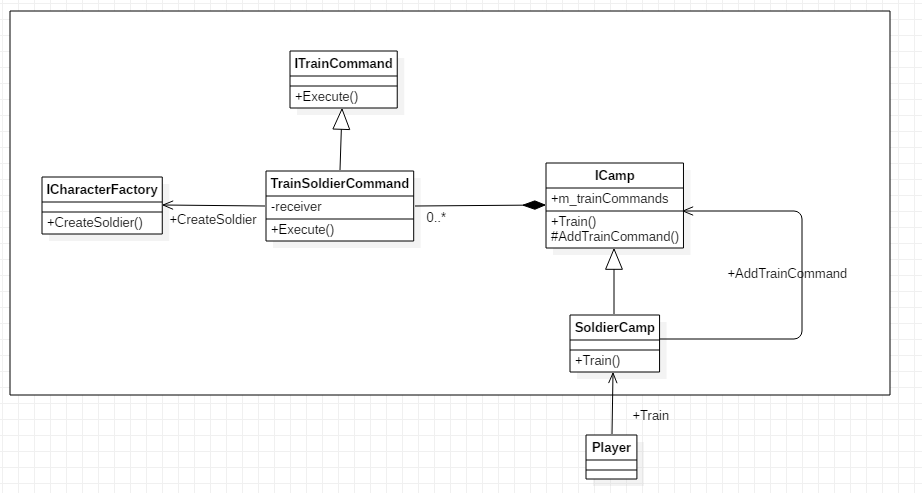
　　如果希望让"命令对象"能包含最多可能的执行方法数量,那么就加强在命令类群组的设计分析.以餐厅点餐的例子来看,就是要思考,是否将餐点与饮料的点餐单合并为一张

　　如果希望能让命令可以任意地执行和撤销,那么就需要着重在命令管理者(Invoker)的设计实现上.以餐厅点餐的例子来看,就是要思考这些点餐单是要用人工管理还是要使用计算机系统来辅助管理

　　此外,如果让命令具备任意撤销或不执行的功能,那么系统对于命令的"反向操作"的定义也必须加以实现,或者将反向操作的执行参数,也一并封装在命令类中

　　19.3 使用命令模式(Command)实现兵营训练角色

　　　　19.3.1 训练命令的实现



参与者的说明如下:

　　ITrainCommand: 训练命令界面, 定义了《P级阵地》中训练一个作战单位应有的命令格式和执行方法

　　TrainSoldierCommand: 封装训练玩家角色的命令,将要训练角色的参数定义为成员,并在执行时调用"功能执行类"去执行指定的命令

　　ICharacterFactory: 角色工厂,实际产生角色单位的"功能执行类"

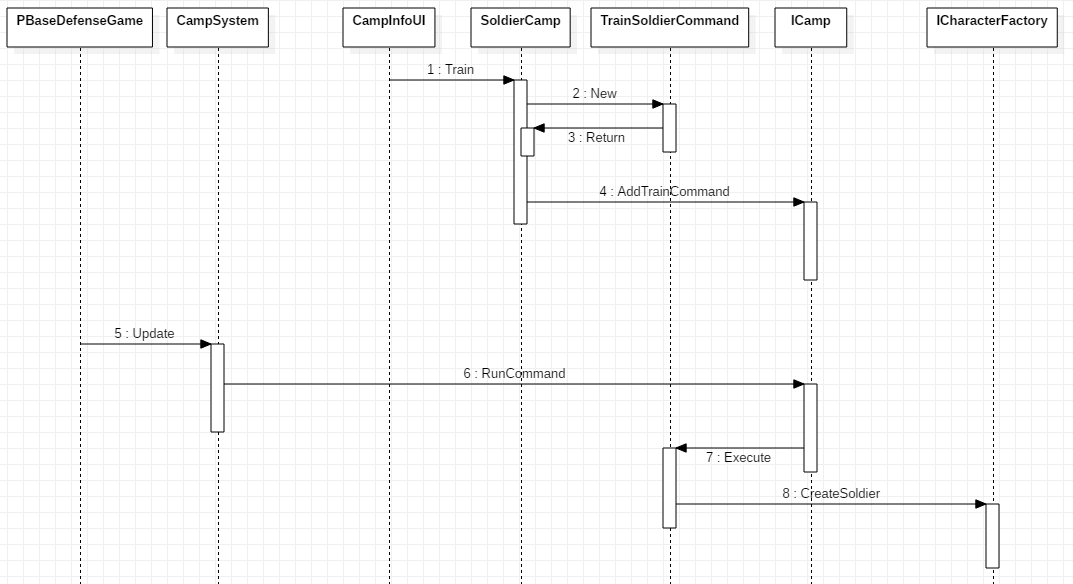
　　ICamp: 兵营界面,包含"管理训练作战单位的命令"的功能,即担任Invoker(命令管理者)的角色,使用泛型来暂存所有的训练命令,并且使用相关的操作方法来添加,删除训练命令

　　SoldierCamp: Soldier兵营界面,负责玩家角色的作战单位训练.当收到训练命令时,会产生命令对象,并按照当前兵营的状态来设置命令对象的参数,最后使用ICamp类提供的界面,将命令加入管理器内

　　　　19.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　　　19.3.3 执行流程



　　　　19.3.4 实现命令模式(Command)时的注意事项

命令模式实现上的选择

《P级阵地》的兵营界面上,除了与训练单位有关的两个命令(训练,取消训练)之外,另外还有两个与升级有关的命令按钮(兵营升级,武器升级).但针对这两个界面命令,《P级阵地》并没有运用命令模式来实现

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

不运用命令模式的主要原因在于:

　　类过多

如果游戏的每一个功能请求都运用命令模式,那么就有能会出现类过多的问题.每一个命令都将产生一个类来负责封装,大量的类会造成项目不易维护

　　请求对象并不需要被管理

指的是兵营升级和武器升级两个命令,在执行上并没有任何延迟或需要被暂存的需求,也就是当请求发出时,功能就要被立即执行.因此,在实现上,只要通过界面类提供的方法(ICamp.LevelUp,ICamp.WeaponLevelUp)来执行即可,让功能的实现类(SoldierCamp)与客户端(CampInfoUI)分离,就可以达成这些功能的设计目标了

因此,在《P级阵地》中,选择实现命令模式(Command)的标准在于:

**”当请求被对象化后,对于请求对象是否有'管理'上的需求.如果有,则以命令模式实现"**

需要实现大量的请求命令时

一个中小型规模的多人在线游戏,Server与Client之间的请求命令可能多达上千个,若每一个请求命令都需产生类的话,那么就真的会发生"类过多"的问题.为了避免这样的问题发生,可以改用下列的方式来实现

　　1. 使用注册回调函数

同样将所有的命令以管理容器组织起来,并针对每一个命令,注册一个回调函数,并将功能执行者(Receiver)改为一个函数/方法,而非对象.最后,将多个相同功能的回调函数以一个类封装在一起

　　2. 使用泛型程序设计

将命令界面以泛型方式来设计,将功能执行者(Receiver)定义为泛型类,命令执行时调用泛型类中的固定方法.但以这种方式实现时,限制会比较大,必须限定每个命令可以封装的参数数量,而且封装参数的名称比较不直观,也就是将参数以Param1, Param2的方式命名

因为固定调用功能执行者(Receiver)中的某一个方法,所以方法名称会固定,比较不容易与实际功能联想

话虽如此,但如果系统中的每个命令都很"单纯"时,使用泛型程序设计可以省去重复定义类或回调函数的麻烦

　　19.4 命令模式(Command)面对变化时

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　19.5 结论

命令模式的优点是,将请求命令封装成对象后,对于命令的执行,可加上额外的操作和参数化.但因为命令模式的应用广泛,在分析时需要针对系统需求加以分析,以避免产生过多的命令类

其他应用方式

实现网络在线型游戏时,对于Client/Server间数据封包的传递,大多数会使用命令模式来实现.但对于数据封包命令的管理,可能不会实现撤销操作,一般比较侧重于执行和记录上.而"记录"则是网络在线型游戏的另一个重点,通过记录,可以分析玩家与游戏服务器之间的互动,了解玩家在操作游戏时的行为,另外也有防黑客预警的作用

第20章 关卡设计----责任链模式(Chain of Responsibility)

　　20.1 关卡设计

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

定期更新Update方法中,判断当前是否需要产生新的关卡,如果需要,则先调用产生关卡CreateStage方法.而关卡是否结束,则是直接判断当前敌方阵营的角色数量,如果为0,代表关卡结束可以进入下一个关卡.确认开始下一个关卡CheckNextStage方法中,会先判断当前得分来判断是否可以进入下一个关卡

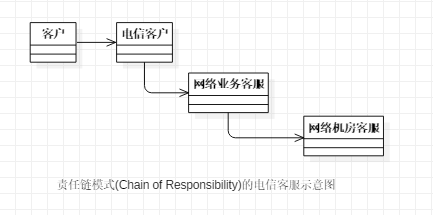
仔细分析两个与关卡产生有关的方法: CreateStage, CheckNextStage, 其中都按照当前关卡(m\_NowStageLv)的值,来决定接下来要产生哪些地方角色以及是否切换到下一个关卡.上述的程序代码只产生了3个关卡而已,但《P级阵地》的目标是希望能设置数十个以上的关卡让玩家挑战,所以若以上述的写法来设计的话,程序代码将变得非常冗长,而且弹性不足,无法让企划人员快速设置和调整,所以我们需要使用新的设计来重新编写程序

重构的目标是,希望能将关卡数据使用类加以封装.而封装的信息包含: 要出场的敌方角色的设置,通关条件,下一关的记录等.也就是让每一关都是一个对象并加以管理.而关卡系统则是在这群对象中寻找"条件符合"的关卡,让玩家进入挑战.等到关卡完成后,再进入到下一个条件符合的关卡

　　20.2 责任链模式(Chain of Responsibility)

当有问题需要解决,而且可以解决问题的人还不止一个时,就有很多方式可以得到想要的答案.例如可以将问题同时交给可以解决问题的人,请他们都回答,但这个方式比较浪费资源,也会造成浪费,也有可能回答的人有等级之分,不适合太简单和太复杂的问题.另一个方式就是将可以回答问题的人,按照等级或专业一个个串接起来.责任链模式(Chain of Responsibility)就是提供了一个可以将这些回答问题的人,一个个链接起来的设计方法

　　　　20.2.1 责任链模式(Chain of Responsibility)的定义



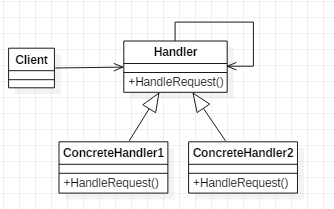
在运用责任链模式解决问题时,只要将下列的几个重点列出来分别实现,即可满足模式的基本要求

　　可以解决请求的接收者对象:　　这些类对象能够了解"请求"信息的内容,并判断本身能否解决

　　接收者对象间的串接:　　利用一个串接机制,将每一个可能可以解决问题的接收者对象给串接起来,对于被串接的接收者对象来说,当本身无法解决这个问题时,就利用这个串接机制,让请求能不断地传递下去;或使用其他管理方式,让接收者对象得以链接

　　请求自动转移传递:　　发出请求后,请求会自动往下转移传递,过程之中,发送者不需特别转换接口

　　　　20.2.2 责任链模式(Chain of Responsibility)的说明



GoF参与者的说明如下:

　　Handler(请求接收者接口):

定义可以处理客户端请求事项的接口

可包含“可链接下一个同样能处理请求"的对象引用

　　ConcreteHandler1, ConcreteHandler2(实现请求接收者接口)

实现请求处理接口,并判断对象本身是否能处理这次的请求

不能完成请求的话,交由后继者(下一个)来处理

　　Client (请求发送者)

将请求发送给第一个接收者对象,并等待请求的回复

　　　　20.2.3 责任链模式(Chain of Responsibility)的实现范例

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　20.3 使用责任链模式(Chain of Responsibility)实现关卡系统

游戏中的关卡都是一关关的串接,完成了这一关之后就进入下一关,所以在实现上使用责任链模式来串接每一个关卡是非常合适的.但是对于每一个关卡的通关判断规则,则要按照各个游戏的需求来设计.

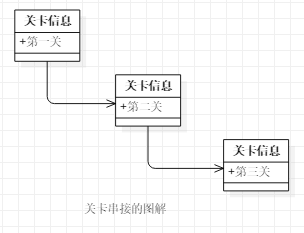
　　　　20.3.1 关卡系统的设计

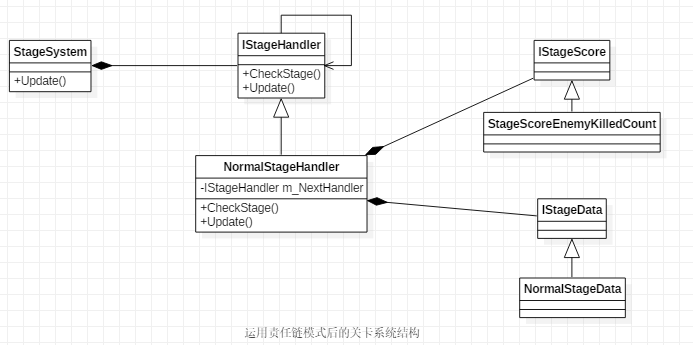
对于《P级阵地》关卡系统的修改需求上,关卡可能需要的信息包含要出场的敌方角色设置,通关条件及连接下一关卡对象的引用,封装成一个"接收者类",并增加能够判断通关与否的方法,作为是否前进到下一关的判断依据.

每个关卡对象都会判断"当前的游戏状态"是否符合关卡的"通关条件"

如果符合通关条件,则将关卡通关与否的判断交由下一个关卡对象判断,直到有一个关卡对象负责接下来的"关卡开启"工作

如果不符合,则将"当前关卡"维持在这一个关卡对象上,继续让现在的关卡对象负责"关卡开启"工作





参与者说明如下:

　　IStageHandler:　　定义可以处理"过关判断"和"关卡开启"的接口,也包含指向下一个关卡对象的应用

　　NormalStageHandler:　　实现关卡接口, 负责"常规"关卡的开启和过关条件判断

　　IStageScore:　　定义判断通关与否的操作接口

　　StageScoreEnemyKilledCount:　　使用当前的"击杀敌方角色数",作为通关与否的判断

　　IStageData:　　定义关卡内容的操作接口,在《P级阵地》中,关卡内容指的是:

　　　　这一关会出现攻击玩家阵营的敌方角色数据之设置

　　　　关卡开启

　　　　关卡是否结束的判断

　　NormalStageData:　　实现"常规"关卡内容,实际将产生的的敌方角色放入战场上攻击玩家阵营,以及实现判断关卡是否结束的方法

此外,IStageScore和IStageData这两个类也是应用策略模式(Strategy)的类,让关卡系统在"过关判断"和"产生敌方单位"这两个设计需求上,能更具备灵活性,不限制只有一种玩法

　　　　20.3.2 实现说明

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

"关卡内容"一般指的就是玩家要挑战的项目,这些项目可能是出现3个敌方角色,让玩家击退;也可能是出现3个道具让玩家可以去搜索获取;或是设计特殊任务关卡让玩家去完成.而这些设置内容都会放进IStageData的子类中,并且通过Game Loop更新机制,让关卡内容可以顺利产生给玩家挑战

修正的关键点在于

1. 定期更新方法中,将切换关卡的判断交给一群关卡对象串接起来的链表来负责,所以需要切换关卡时,询问关卡对象链表,就可以获取当前可以进行的关卡

2. 在初始化关卡系统时,将所有关卡的数据一次设置完成,包含关卡要出现的敌方角色等级,数量,武器等级,过关的判断及连接的下一关

　　　　20.3.3 使用责任链模式(Chain of Responsibility)的优点

将旧方法中的CreateStage,CheckNextStage两个方法的内容,使用关卡对象来代替,这样一来,原本可能出现的冗长式写法就获得了改善.并且将关卡内容(IStageData),过关条件(IStageScore)类化,可使得《P级阵地》中关卡的类型有多种形式的组合.而关卡设计的数据将来也可以搭配"企划工具"来设置,增加关卡设计人员的调整灵活度

　　　　20.3.4 实现责任链模式(Chain of Responsibility)时的注意事项

不用从头判断

使用泛型容器来管理关卡对象

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

public class StageSystem: IGameSystem {

private List<IStageHandler> m\_StageHandlers;

private int m\_NowStageLv = 1;

...

}

　　20.4 责任链模式(Chain of Responsibility)面对变化时

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

public abstract class IStageHandler {

...

public abstract int LoseHeart();

}

public class NormalStageHandler: IStageHandler {

...

public override int LoseHeard() {

return 1;

}

}

public class StageSystem: IGameSystem {

public void LoseHeader() {

m\_NowHeader -= m\_NowStageHandler.LoseHeart();

m\_PBDGame.ShowHeart(m\_NowHeart);

}

}

public class BossStageHandler: NormalStageHandler {

public BossStageHandler(IStageScore StageScore, IStageData StageData): base(StateScore, StageData) { }

public override int LoseHeader() {

return StageSystem.MAX\_HEART;

}

StageData = new NormalStageData(3f, GetSpawnPosition(), AttackPosition);

StageData.AddStageData(ENUM\_Enemy.Ogre,ENUM\_Weapon.Rocket, 3);

StageScore = new StageScoreEnemyKilledCount(13, this);

NewStage = NewStage.SetNextHandler(new BossStageHandler(StageScore, StageData));

[复制代码](javascript:void(0);)

　　20.5 结论

责任链模式让一群信息接收者能够一起被串联起来管理,让信息判断上能有一致的操作接口,不必因为不同的接收者而必须执行"类转换操作",并且让所有的信息接收者都有机会可以判断是否提供服务或将需求移往下一个信息接收者,在后续的系统维护上,也可以轻易地增加接收者类

与其他模式的合作

在通关判断上,可以配合策略模式,让通关的规则具有其他的变化形式,而不只是单纯地击退所有进攻的敌方角色

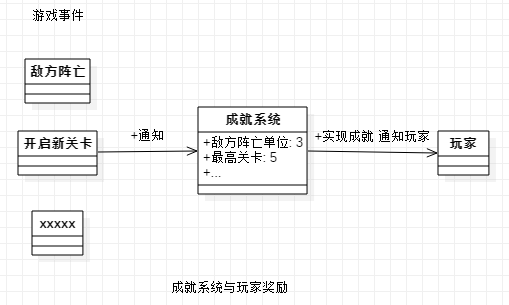
第21章 成就系统----观察者模式(Observer)

　　21.1 成就系统

成就系统(AchievementSystem),是早期单机游戏就出现的一种系统。

成就系统(AchievementSystem)中的项目,都会和游戏本身有关,并且在玩家游玩的过程中,就能顺便收集,或是反复进行某种操作就能实现目标.一般可以先将成就项目分门别类

实现上,会先定义"游戏事件",如敌方角色阵亡,玩家角色阵亡,玩家角色升级等.当游戏进行过程中,有任何"游戏事件"被触发时,系统就要通知对应的"成就项目",进行累积或条件判断,如果达到,则完成"成就项目"并通知玩家或直接给予奖励



一个简单的设计方式是,我们可以把通知成就系统的程序代码加入到"成就事件触发"的方法中.

事件触发后,调用成就系统(AchievementSystem)中的NotifyGameEvent方法,并将触发的游戏事件及触发时的敌方角色传入.上述范例中,使用枚举(ENUM)的方式来定义"游戏事件",并将事件从参数行传入,而不是针对每一个游戏事件定义特定的调用方法,这样组可以避免成就系统定义过多的接口方法.而成就系统的NotifyGameEvent方法,可根据参数传入的"游戏事件"参数,来决定后续的处理流程

因为"游戏事件"非常多,所以在NotifyGameEvent方法中判断emGameEvent的参属性,再分别调用对应的私有成员方法

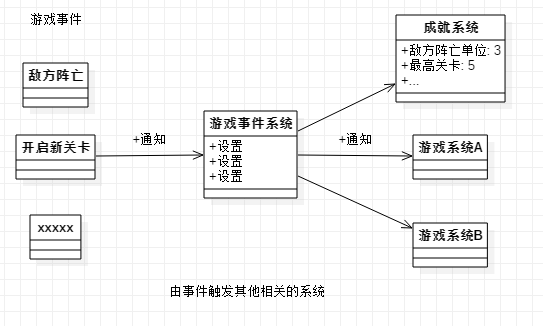
如果让成就系统(AchievementSystem)负责每一个游戏事件的方法,并针对每一个单独的游戏事件,去进行"成就项目的累积或判断", 会让成就系统的扩充被限制在每个游戏事件处理方法中,当以后需要针对某一个游戏事件增加成就项目时,就必须通过修改原有"游戏事件处理方法"中的程序代码才能达成.

此外,"游戏事件"发生时可能不是只有成就系统会被影响,其他系统也可能需要追踪相关的游戏事件,因此,如果都是在"游戏事件"的触发点进行每个系统调用的话,那么触发点的程序代码将会变得非常复杂

所以要将"游戏事件"与"成就系统"分开,让成就系统仅关注于某些游戏事件的发生,而游戏事件的发生,也不只是提供给成就系统使用.这样的设计才是适当的设计

要如何完成这样的设计呢?如果能将"游戏事件的产生与通知"独立成为一个系统,并且让其他系统能通过"订阅"或"关注"的方式,来追踪游戏事件系统发生的事.也就是,当游戏事件系统发生事件时,会负责去通知所有"订阅"了游戏事件的系统,此时被通知的系统,再根据自己的系统逻辑自行决定后续的处理操作.如果能按照上述流程来进行设计,就是一个极为适当的设计.上述的流程,其实就是观察者模式(Observer)所要表达的内容

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code



　　21.2 观察者模式(Observer)

观察者模式(Observer)与命令模式(Command)是很相似的模式,两者都是希望"事件发生"与"功能执行"之间不要有太多的依赖性,不过,还是可以按照系统的使用需求,分析出应该运用哪个模式

　　　　21.2.1 观察者模式(Observer)的定义

GoF对观察者模式(Observer)的定义为:

"在对象之间定义一个一对多的连接方法,当一个对象变换状态时,其他关联的对象都会自动收到通知"

社交网站就是最佳的观察者模式(Observer)实现范例,当我们在社交网站上,与另一位用户成为好友,加入一个粉丝团或关注另一位用户的状态,那么当这些好友,粉丝团,用户有任何的新的动态或状态变动时,就会在我们动态页面上"主动"看到这些对象更新的情况,而不必到每一位好友或粉丝团中查看

在早期社交网站还没有广泛流行之前,说明观察者模式(Observer)常以"报社-订户"来做说明:多为订户向报社"订阅(Subscribe)"了一份报纸,而报社针对昨天的新闻整理编辑之后,在今天一早进行"发布(Publish)"的工作,接着送报生会主动按照订阅的信息,将每份报纸送到订户手上

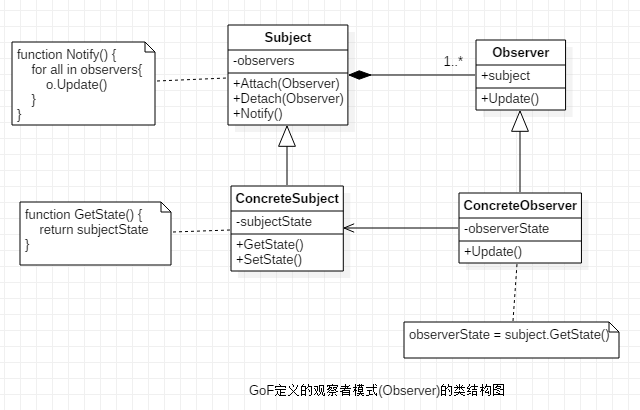
在上面的案例中,都存在"主题目标"与其他"订阅者/关注者"之间的关系(一对多),当主题变化时,就会通过之前建立的"关系",将更动态的信息传递给"订阅者/关注者".因此,实现上可分为以下几点:

　　主题者,订阅者的角色

　　如何建立订阅者与主题者的关系

　　主题者发布信息时,如何通知所有订阅者

　　　　21.2.2 观察者模式(Observer)的说明



GoF参与者的说明如下:

　　Subject(主题接口)

定义主题的接口

让观察者通过接口方法,来订阅,解除订阅主题,这些观察者在主题内部可使用泛型容器加以管理

在主题更新时,通知所有观察者

　　ConcreteSubject(主题实现)

实现主题接口

设置主题的内容及更新,当主题变化时,使用父类的通知方法告知所有的观察者

　　Observer(观察者接口)

定义观察者的接口

提供更新通知方法,让主题可以通知更新

　　ConcreteObserver(观察者实现)

实现观察者接口

针对主题的更新,按需求向主题获取更新状态

　　　　21.2.3 观察者模式(Observer)的实现范例

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

信息的推与拉

主题(Subject)改变时,改变的内容要如何让观察者(Observer)得知,运行方式可分为推(Push)信息与拉(Pull)信息两种模式:

　　推信息:

主题(Subject)将变动的内容主动"推"给观察者(Observer).一般会在调用观察者(Observer)的通知(Update)方法时,同时将更新的内容当成参数传给观察者(Observer).例如传统的报社,杂志社的模式,每一次的发行都会将所有的内容一次发送给订阅者,所有的订阅者接到的信息都是一致的,然后订阅者再从获取需要的信息来进行处理:

优点:　　所有的内容都一次传给观察者,省去观察者再向主题查询的操作,主题类也不需要定义太多的查询方式供观察者来查询

缺点:　　如果推送的内容过多,容易使观察者收到不必要的信息或造成查询上的困难,不当的信息设置也可能造成系统性能的降低

　　拉信息:

主题内容变动时,只是先通知观察者当前内容有变动,而观察者则是按照系统需求,再向主题查询(拉)所需的信息

优点:　　主题只通知当前内容有更新,再由观察者自己去获取所需的信息,因为观察者自己更知道需要哪些信息,所以太会去获取不必要的信息

缺点:　　因为观察者需要向主题查询更新的内容,所以主题必须提供查询方式,这样一来,就容易造成主题类的接口方法过多

而再实现设计上,必须根据系统所需要的最佳情况来判断,是要使用"推信息"还是"拉信息"的方式

　　21.3 使用观察者模式(Observer)实现成就系统

重构成就系统,可按照下面的步骤来进行:

1. 实现游戏事件系统(GameEventSystem);

2. 完成各个游戏事件的主题及其观察者

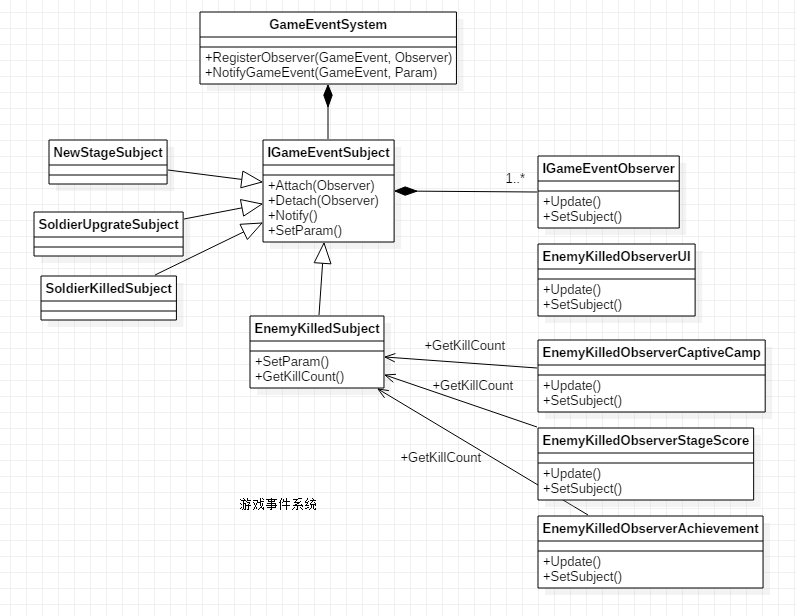
3. 实现成就系统(AchievementSystem)及订阅游戏事件

4. 重构游戏事件触发点

　　　　21.3.1 成就系统的新架构

对于解决《P级阵地》成就系统的需求,首先应该完成的是游戏事件系统.在游戏事件系统中,会将每个游戏事件当成主题,让其他系统可针对感兴趣的游戏事件进行"订阅".当游戏事件被触发时,游戏事件系统会去通知所有的系统,再让各个系统针对所需要的信息进行查询

而成就系统将是游戏事件系统的一个订阅者/观察者.它将针对成就项目所需要的游戏事件进行订阅的操作,等到游戏事件系统发布游戏事件时,成就系统再去获取所需的信息来累积成就项目或判断成就项目是否达到



参与者的说明如下:

GameEventSystem: 游戏事件系统,用来管理游戏中发生的事件,针对每一个游戏事件产生一个"游戏事件主题(Subject)",并提供接口方法让其他系统能订阅指定的游戏事件

IGameEventSubject: 游戏事件主题接口,负责定义《P级阵地》中"游戏事件"内容的接口,并延伸出下列的游戏事件主题:

　　EnemyKilledSubject: 敌方角色阵亡

　　SoldierKilledSubject: 玩家角色阵亡

　　SoldierUpgateSubject: 玩家角色升级

　　NewStageSubject: 新关卡

IGameEventObserver: 游戏事件观察者接口,负责《P级阵地》中游戏事件触发时被通知的操作接口

EnemyKilledObserver 观察者们: 订阅"敌方角色阵亡“主题(EnemyKilledSubject)的观察者类,共有:

　　EnemyKilledObserverUI: 将敌方角色阵亡信息显示在界面上

　　EnemyKilledObserverStageScore: 将敌方角色阵亡信息提供给关卡系统(StageSystem)

　　EnemyKilledObserverAchievement: 将敌方角色提供给成就系统(AchievementSystem)

同一个游戏事件可以提供给不同的系统一起订阅,并能同时接收到更新信息

　　　　21.3.2 实现说明

实现游戏事件系统

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

完成各个游戏事件主题及其观察者

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

1. 敌人角色阵亡

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

2. 玩家角色阵亡

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

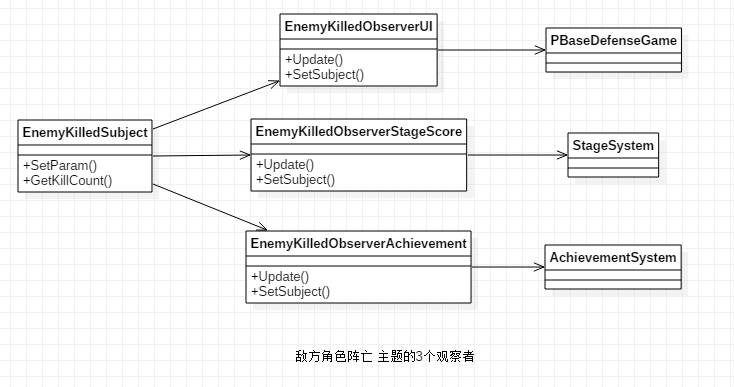
3. 玩家角色升级

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

4. 进入新关卡

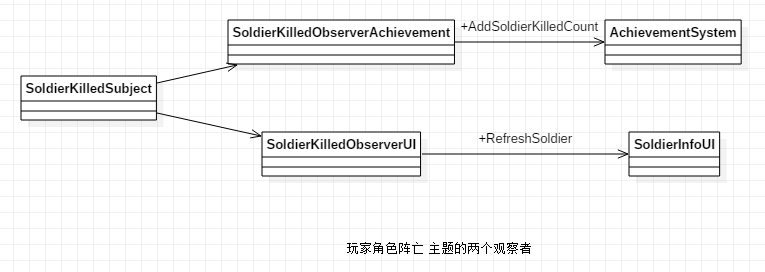
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

1. 敌方角色阵亡 主题的观察者



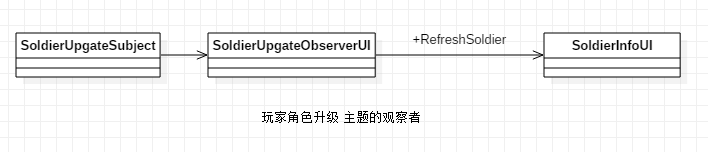
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

2. 玩家角色阵亡 主题的观察者



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

3. 玩家角色升级 主题的观察者



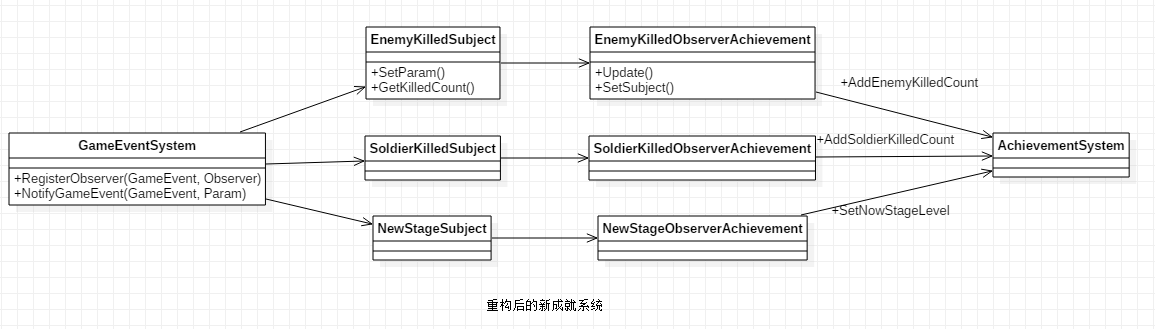
https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

4. 进入新关卡     主题的观察者



https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

到了这个阶段,游戏事件系统算是构建完成,让我们回到本章开始讨论的成就系统.配合游戏事件系统的订阅机制,新的成就系统被重构为:只记录相关的成就事项,并提供相关的接口方法,让与成就事项相关的观察者们(Observer)使用



实现成就系统及订阅游戏事件

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

重构游戏事件触发点

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　　　21.3.3 使用观察者模式(Observer)的优点

成就系统以游戏事件为基础,记录每个游戏事件发生的次数及时间点,作为成就项目的判断依据.但是当同一游戏事件被触发后,可能不只是只有一个成就系统会被触发,系统中也可能存在着其他系统需要使用同一个游戏事件.因此,加入了以观察者模式为基础的游戏事件系统,就可以有效地解除游戏事件的发生与有关的系统功能调用之间的绑定,这样在游戏事件发生时,不必理会后续的处理工作,而是交给游戏事件主题负责调用观察者/订阅者.此外,也能同时调用多个系统同时处理这个事件引发的后续操作

　　　　21.3.4 实现观察者模式(Observer)时的注意事项

双向与单向信息通知

社交网页上的"粉丝团"比较像是观察者模式:当粉丝团发布了一则新的动态后,所有订阅的用户都可以看到新增的动态,而用户与用户之间则是同时扮演"主题(Subject)"与”观察者(Observer)"的角色,除了同时收到其他好友的动态信息,当自己有任何的动态消息时,也会同时广播给好友们(观察者)

类过多的问题

游戏事件,游戏事件主题 会随着项目的开发不断地增加,于此同时,这些主题的观察者的数量也会随之上升.从当前的《P级阵地》内容来看,已经产生了7个游戏事件观察者类(IGameEventObserver), 所以不难想象,在大型项目可能会产生非常多的观察者类(IGameEventObserver).当然,在某些情况下类过多,反而是个缺点.因此,如果想要减少类的产生,可以考虑向游戏的主题注册时间,不要使用"类对象"而是使用"回调函数",之后再将功能相似的"回调函数"以同一个类来管理,就能减少过多类的问题.这一部分的解决方式与解决大量请求命令时的想法是一样的

比较命令模式与观察者模式

这两个模式都是着重在于将"发生"与"执行"这两个操作消除耦合(或减少依赖性)的模式,当观察者模式中的主题只存在一个观察者时,就非常像是命令模式的基本架构,但还是有一些差异可以分辨出两个模式应用的时机:

　　命令模式:　　该模式的另一个重点是"命令的管理",应用的系统对于发出的命令有新增,删除,记录,排序,撤销等等的需求

　　观察者模式:　　对于"观察者/订阅者"可进行管理,意思是观察者可以在系统运行时间决定订阅或退订等操作,让"执行者(观察者/订阅者)"可以被管理

所以,两者在应用上还是有明确的目标.当然,如果有需要将两个模式整合应用并非不可能,像是让命令模式(Command)的执行者可以动态新增,删除;或是让观察者模式(Observer)的"每一次发布"都可以被管理等等.而这也是本书所要呈现的重点----模式之间的交互合作,会产生更大的效果

　　21.4 观察者模式(Observer)面对变化时

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　21.5 结论

观察者模式的设计原理是,新设置一个主题,让这个主题发布时可同时通知关系这个主题的观察者/订阅者,并且主题不必理会观察者/订阅者接下来会执行哪些操作.观察者模式的主要功能和优点,就是将主题发生和功能执行这两个操作解除绑定----即消除依赖性,而且对于"执行者(观察者/订阅者)"来说,还是可以动态决定是否要执行后续的功能

观察者模式(Observer)的缺点是可能造成过多的观察者类.不过利用注册"回调函数"来取代"注册类对象"可有效减少类的产生

其他应用方式

在游戏场景中,设计者通常会摆放一些所谓的"事件触发点",这些事件触发点会在玩家角色进入时,触发对应的游戏功能,例如突然会出现一群怪物NPC来攻击玩家角色;或是进入剧情模式演出一段游戏故事剧情等等.而且游戏通常不会限制一个事件触发点只能执行一个操作,因此实现时可以将每一个事件触发点当成一个"主题",而每一个要执行的功能,都成为"观察者",当事件被触动发布时,所有的观察者都能立即反应