游戏开发过程中，各种游戏状态的切换无处不在。但很多时候，简单粗暴的if else加标志位的方式并不能很地道地解决状态复杂变换的问题，这时，就可以运用到状态模式以及状态机来高效地完成任务。状态模式与状态机，因为他们关联紧密，常常放在一起讨论和运用。而本文将对他们在游戏开发中的使用，进行一些探讨。

PS:

这篇文章起源于《Game Programming Patterns》第二章第六节。

这是一篇略长的文章，约5200余字，将分析游戏开发过程中状态模式与有限状态机的运用，已经非常了解相关内容的高端选手请略读。

文中使用C++承载讲解内容，文章末尾也提供了Unity&C#版本的代码实现。

一、文章的短版本与思维导图

还是国际惯例，先放出这篇文章的短版本——所涉及知识点的一张思维导图，再开始正文。大家若是疲于阅读文章正文，直接看这张图，也是可以Get到本文的主要知识点的大概。

二、引例

假如我们现在正在开发一款横版游戏。当前的任务是实现玩家用按键操纵女英雄。当按下向上方向键的时候，女英雄应该跳跃。那么我们可以这样实现:

void Heroine::handleInput(Input input)

{

if (input == PRESS\_UP)

{

yVelocity\_ = JUMP\_VELOCITY;

setGraphics(IMAGE\_JUMP);

}

}

OK，实现是实现了，但是一堆BUG。比如，我们没有防止主角“在空中跳跃“，当主角跳起来后持续按向上键，会导致她一直飘在空中。简单地修复方法可以是：添加一个 isJumping布尔值变量。当主角跳起来后，就把该变量设置为True.只有当该变量为False时，才让主角跳跃，代码如下：

void Heroine::handleInput(Input input)

{

if (input == PRESS\_UP)

{

if (!isJumping\_)

{

isJumping\_ = true;

// Jump...

}

}

}

接下来，我们想实现主角的闪避动作。当主角站在地面上的时候，如果玩家按下向下方向键，则下蹲躲避，如果松开此键，则起立。代码如下：

void Heroine::handleInput(Input input)

{

if (input == PRESS\_UP)

{

// Jump if not jumping...

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

setGraphics(IMAGE\_DUCK);

}

}

else if (input == RELEASE\_DOWN)

{

setGraphics(IMAGE\_STAND);

}

}

 找找看， 这次bug又在哪里？

使用这段代码，玩家可以：按向下键下蹲，按向上键则从下蹲状态跳起，英雄会在跳跃的半路上变成站立图片…….是时候增加另一个标识了……

void Heroine::handleInput(Input input)

{

if (input == PRESS\_UP)

{

if (!isJumping\_ && !isDucking\_)

{

// Jump...

}

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

isDucking\_ = true;

setGraphics(IMAGE\_DUCK);

}

}

else if (input == RELEASE\_DOWN)

{

if (isDucking\_)

{

isDucking\_ = false;

setGraphics(IMAGE\_STAND);

}

}

}

下面再加一点功能，如果玩家在跳跃途中按了下方向键，英雄能够做下斩攻击就太炫酷了。其代码实现如下：

void Heroine::handleInput(Input input)

{

if (input == PRESS\_UP)

{

if (!isJumping\_ && !isDucking\_)

{

// Jump...

}

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

isDucking\_ = true;

setGraphics(IMAGE\_DUCK);

}

else

{

isJumping\_ = false;

setGraphics(IMAGE\_DIVE);

}

}

else if (input == RELEASE\_DOWN)

{

if (isDucking\_)

{

// Stand...

}

}

}

BUG又出现了，这次发现了没？

目前在下斩的时候，按跳跃键居然可以继续向上跳， OK，要解决它又是另一个字段……

很明显，我们采用的这种if else加标志位的做法并不好用。每次我们添加一些功能的时候，都会不经意地破坏已有代码的功能。而且，我们还没有添加“行走”的状态，加了之后问题恐怕更多。

这一幕是不是有些似曾相识？我想各位同学在踏入游戏开发领域的早期，多少会碰到过一些类似的情况，反正我是碰到过。其实，在这种情况下，状态机是可以帮上我们忙的。

三、使用有限状态机

让我们画一个流程图。目前的状态有，站立，跳跃，下蹲，下斩。得到的状态图示大致如下：

OK，我们成功创建了一个有限状态机(Finite-state machine,FSM)。它来自计算机科学的分支自动理论，那里有很多著名的数据结构，包括著名的图灵机。状态机是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。有限状态机是其中最简单的成员。（本文限于篇幅，更多状态机暂不讨论，在文章末尾进阶阅读中，列举了分层状态机Hierarchical State Machines与下推自动机Push down Automata的参考资料，有需要的朋友们可以阅读）

有限状态机FSM的要点是：

拥有一组状态，并且可以在这组状态之间进行切换。在我们的例子中，是站立，跳跃，蹲下和跳斩。

状态机同时只能在一个状态。英雄不可能同时处于跳跃和站立。事实上，防止这点是使用FSM的理由之一。

一连串的输入或事件被发送给机器。在我们的例子中，就是按键按下和松开。

每个状态都有一系列的转换，转换与输入和另一状态相关。当输入进来，如果它与当前状态的某个转换匹配，机器转为转换所指的状态。

举个例子，在站立状态时，按下下键转换为俯卧状态。在跳跃时按下下键转换为跳斩。如果输入在当前状态没有定义转换，输入就被忽视。

目前而言，游戏编程中状态机的实现方式，有两种可以选择：

用枚举配合switch case语句。

用多态与虚函数（也就是状态模式）。

下面让我们用代码来实现。不妨先从简单的方式开始,用枚举与switch case语句实现。

四、用枚举配合switch case实现状态机

我们知道，上文中实现的女英雄类Heroine有一些布尔类型的成员变量：isJumping\_和isDucking，但是这两个变量永远不可能同时为True。

OK，这边可以提供一个小经验：当你有一系列的标记成员变量，而它们只能有且仅有一个为True时，定义成枚举(enum)其实更加适合。

在这个例子当中，我们的FSM的每一个状态可以用一个枚举来表示，所以，让我们定义以下枚举:

enum State

{

STATE\_STANDING,

STATE\_JUMPING,

STATE\_DUCKING,

STATE\_DIVING

};

好了，无需一堆flags了， Heroine类只需一个state成员就可以胜任。在前面的代码中，我们先判断输入事件，然后才是状态。那种风格的代码可以让我们集中处理与按键相关的逻辑，但是，它也让每一种状态的处理代码变得很乱。我们想把它们放在一起来处理，因此，我们先对状态做分支switch处理。代码如下:

void Heroine::handleInput(Input input)

{

switch (state\_)

{

case STATE\_STANDING:

if (input == PRESS\_B)

{

state\_ = STATE\_JUMPING;

yVelocity\_ = JUMP\_VELOCITY;

setGraphics(IMAGE\_JUMP);

}

else if (input == PRESS\_DOWN)

{

state\_ = STATE\_DUCKING;

setGraphics(IMAGE\_DUCK);

}

break;

case STATE\_JUMPING:

if (input == PRESS\_DOWN)

{

state\_ = STATE\_DIVING;

setGraphics(IMAGE\_DIVE);

}

break;

case STATE\_DUCKING:

if (input == RELEASE\_DOWN)

{

state\_ = STATE\_STANDING;

setGraphics(IMAGE\_STAND);

}

break;

}

}

现在的代码看起来比之前的代码地道了一点。我们简化了状态的处理，将所有处理单个状态的代码都集中在了一起。这样做是实现状态机的最简单方式，而且在特定情况下，这就是最佳的解决方案。

我们的问题可能也会超过此方案能解决的范围。比如，我们想在主角下蹲躲避的时候“蓄能”，然后等蓄满能量之后可以释放出一个特殊的技能。那么，当主角处理躲避状态的时候，我们需要添加一个变量来记录蓄能时间。

我们可以添加一个chargeTime成员来记录主角蓄能的时间长短。假设，我们已经有一个update方法了，并且这个方法会在每一帧被调用。那么，我们可以使用其来记录蓄能的时间，就像这样:

void Heroine::update( )

{

if (state\_ == STATE\_DUCKING)

{

chargeTime\_++;

if (chargeTime\_ > MAX\_CHARGE)

{

superBomb( );

}

}

}

我们需要在主角躲避的时候重置这个蓄能时间，所以，我们还需要修改handleInput方法：

void Heroine::handleInput(Input input)

{

switch (state\_)

{

case STATE\_STANDING:

if (input == PRESS\_DOWN)

{

state\_ = STATE\_DUCKING;

chargeTime\_ = 0;

setGraphics(IMAGE\_DUCK);

}

// Handle other inputs...

break;

// Other states...

}

}

总之，为了添加蓄能攻击，我们不得不修改两个方法，并且添加一个 chargeTime成员给主角，尽管这个成员变量只有在主角处于躲避状态的时候才有效。我们其实真正想要的是把所有这些与状态相关的数据和代码封装起来。

接下来，我们正式介绍四人帮设计模式中的状态模式来解决这个问题。

五、用状态模式实现状态机

5.1 状态模式概述

对于沉浸于面向对象思维方式的同学来说，每一个条件分支都可以用动态调度来解决（也就是虚函数和多态来解决）。但是，如果你不分青红皂白每次都这样做，可能就会简单的问题复杂化。其实有时候，一个简单的if语句就足够了。

四人帮对于状态模式是这么描述的：

“Allow an object to alter its behavior whenits internal state changes. The object will appear to change its class.

允许对象在当内部状态改变时改变其行为，就好像此对象改变了自己的类一样。”

其实，状态模式主要解决的就是当控制一个对象状态转换的条件表达式过于复杂的情况，它把状态的判断逻辑转移到表示不同的一系列类当中，可以把复杂的逻辑判断简单化。

状态模式的实现要点，主要有三点：

为状态定义一个接口。

为每个状态定义一个类。

恰当地进行状态委托。

下面将分别进行概述。

5.2 步骤一、为状态定义一个接口

首先，我们为状态定义一个接口。每一个与状态相关的行为都定义成虚函数。对于上文的例子而言，就是handleInput和update函数。

class HeroineState

{

public:

virtual ~HeroineState( ) {}

virtual void handleInput(Heroine& heroine, Input input) {}

virtual void update(Heroine& heroine) {}

};

5.3 步骤二、为每个状态定义一个类

对于每一个状态，我们定义了一个类并继承至此状态接口。它覆盖的方法定义主角对应此状态的行为。换句话说，把之前的switch语句里面的每个case语句中的内容放置到它们对应的状态类里面去。比如：

class DuckingState : public HeroineState

{

public:

DuckingState( )

:chargeTime\_(0)

{ }

virtual void handleInput(Heroine& heroine, Input input) {

if (input == RELEASE\_DOWN)

{

// Change to standing state...

heroine.setGraphics(IMAGE\_STAND);

}

}

virtual void update(Heroine& heroine) {

chargeTime\_++;

if (chargeTime\_ > MAX\_CHARGE)

{

heroine.superBomb( );

}

}

private:

int chargeTime\_;

};

注意我们也将chargeTime\_移出了Heroine，放到了DuckingState（下蹲状态）类中。 因为这部分数据只在这个状态有用，这样符合设计的原则。

5.4 步骤三、恰当地进行状态委托

接下来，我们在主角类Heroine中定义一个指针变量，让它指向当前的状态。放弃之前巨大的switch，然后让它去调用状态接口的虚函数，最终这些虚方法就会动态地调用具体子状态的相应函数了：

class Heroine

{

public:

virtual void handleInput(Input input)

{

state\_->handleInput(\*this, input);

}

virtual void update( )

{

state\_->update(\*this);

}

// Other methods...

private:

HeroineState\* state\_;

};

而为了“改变状态”，我们只需要将state\_声明指向不同的HeroineState对象。至此，经过为状态定义一个接口，为每个状态定义一个类以及进行状态委托，经历这三步，就是的状态模式的实现思路了。

六、状态对象的存放位置探讨

这里忽略了一些细节。为了修改一个状态，我们需要给state指针赋值为一个新的状态，但是这个新的状态对象要从哪里来呢？我们的之前的枚举方法是一些数字定义。但是，现在我们的状态是类，我们需要获取这些类的实例。

通常来说，有两种实现存放的思路：

静态状态。初始化时把所有可能的状态都new好，状态切换时通过赋值改变当前的状态。

实例化状态。每次切换状态时动态new出新的状态。

下面分别进行介绍。

6.1 方法一：静态状态

如果一个状态对象没有任何数据成员，那么它的惟一数据成员便是虚表指针了。那样的话，我们就没有必要创建此状态的多个实例了，因为它们的每一个实例都是相等的。

如果你的状态类没有任何数据成员，并且它只有一个函数方法在里面。那么我们还可以进一步简化此模式。我们可以通过一个状态函数来替换状态类。这样的话，我们的state变量只需要变成一个状态函数指针就可以了。在此情况下，我们可以定义一个静态实例。即使你有一系列的FSM在同时运转，所有的状态机都同时指向这一个惟一的实例。

在哪里放置静态实例取决于你的喜好。如果没有任何特殊原因的话，我们可以把它放置到基类状态类中：

class HeroineState

{

public:

static StandingState standing;//站立状态

static DuckingState ducking;//下蹲状态

static JumpingState jumping;//跳跃状态

static DivingState diving;//下斩状态

// Other code...

};

每一个静态成员变量都是对应状态类的一个实例。如果我们想让主角跳跃，那么站立状态的可以这样实现：

if (input == PRESS\_UP)

{

heroine.state\_ = &HeroineState::jumping;

heroine.setGraphics(IMAGE\_JUMP);

}

6.2 方式二：实例化状态

刚刚讲到的基于静态状态的方式有一定的局限性。比如说，一个静态状态就不能胜任上面例子中我们提到的躲避状态的实现。因为躲避状态有一个 chargeTime成员变量，而chargeTime成员变量是专属于每个主角类的躲避状态的。

也就是说，如果我们的游戏里面只有一个主角，那么定义一个静态类没啥问题。但是，如果我们想加入多个玩家，那么此方法就行不通了。

所以，当有多个玩家时，我们不得不在状态切换的时候动态地创建一个躲避状态实例。

这样，我们的FSM就拥有了它自己的实例。当然，如果我们又动态分配了一个新的状态实例，我们需要负责清理老的状态实例。我们这里必须要相当小心，因为当前状态修改的函数是处在当前状态里面，我们需要小心地处理删除的顺序。

另外，我们也会在handleInput方法里面可选地返回一个新的状态。当这个状态返回的

时候，主角将会删除老的状态并切换到这个新的状态，如下所示:

void Heroine::handleInput(Input input)

{

HeroineState\* state = state\_->handleInput(\*this, input);

if (state != NULL)

{

delete state\_;

state\_ = state;

}

}

这样，我们直到从之前的状态返回，才需要删除它。 现在，站立状态可以通过创建一个新实例转换为俯卧状态：

HeroineState\* StandingState::handleInput(Heroine& heroine,

Input input)

{

if (input == PRESS\_DOWN)

{

// Other code...

return new DuckingState( );

}

//Stay in this state.

return NULL;

}

如果可以，推荐使用静态状态，因为它们不会在状态转换时消耗太多的内存和CPU。但是，对于更多的状态，实例化状态的方式将是不错的选择。

七、实践：Unity中C#版本状态模式实现

除了文章中讲解用到的C++版本的代码，我也使用C#在Unity中实现了几个版本的状态模式，如下图，包含一个Stucture（状态模式的框架）和四个Example（四种不同使用状态模式的实例）。

上文引例中的示例，也就是女英雄在站立，跳跃，俯卧，下斩几个状态之间切换的问题，在example4中进行了Unity版本的实现。链接：

https://github.com/QianMo/Unity-Design-Pattern/tree/master/Assets/Behavioral%20Patterns/State%20Pattern

运行场景，按键盘上的上下方向键，可以在console窗口中看到对应的状态，有点游戏发展史早期文字冒险游戏的感觉。运行截图：

由于篇幅原因，代码就不贴在这边了。具体可以到我的Github的repo "Unity-Design-Pattern”中看到。这个repo目前已经在Unity中实现了《设计模式:可复用面向对象软件的基础》一书中提出的23种设计模式。且每种模式都包含对应的结构实现、应用示例以及图示介绍。有感兴趣的朋友可以关注一下。链接：

https://github.com/QianMo/Unity-Design-Pattern

八、本文知识点总结

本文涉及知识点总结如下：

1. 在游戏开发过程中，涉及到复杂的状态切换时，可以运用状态模式以及状态机来高效地完成任务。

2. 有限状态机的实现方式，有两种可以选择：

用枚举配合switch case语句。

用多态与虚函数（即状态模式）。

3. 状态模式的经典定义：允许对象在当内部状态改变时改变其行为，就好像此对象改变了自己的类一样。

4. 对状态模式的理解：状态模式用来解决当控制一个对象状态转换的条件表达式过于复杂的情况，它把状态的判断逻辑转移到表示不同的一系列类当中，可以把复杂的逻辑判断简单化。

5. 状态模式的实现分为三个要点：

为状态定义一个接口。

为每个状态定义一个类。

恰当地进行状态委托。

6. 通常来说，状态模式中状态对象的存放有两种实现存放的思路：

静态状态。初始化时把所有可能的状态都new好，状态切换时通过赋值改变当前的状态。

实例化状态。每次切换状态时动态new出新的状态。

---------------------

作者：浅墨\_毛星云

来源：CSDN

原文：https://blog.csdn.net/poem\_qianmo/article/details/52824776

版权声明：本文为博主原创文章，转载请附上博文链接！