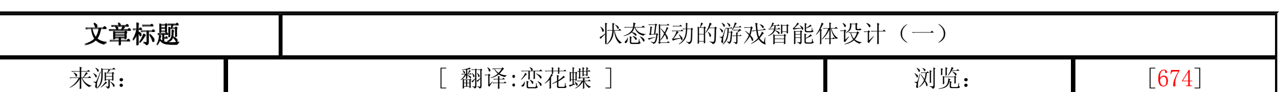
主 页 | 初学者入门 | 项目管理 | 创意思维 | 程序技术 | 美术设计 | 音乐音效 | 移动开发 | 数据库设计

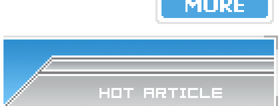


- 基于分布式对象的网游程序结构设计(5)
- 基于分布式对象的网游程序结构设计(4)
- 基于分布式对象的网游程序结构设计(3)
- 基于分布式对象的网游程序结构设计(2)
- 基于分布式对象的网游程序结构设计(1)
- 未来游戏设计的十大技术挑战--2
- 未来游戏设计的十大技术挑战--1
- Open Inventor－Coin3D开发环境

MORE

- 游戏音乐制作案例之《战火 红色警戒》音效制作揭秘
- 英雄连Online 原画
- 游戏音乐制作案例之《乱武天下》
- 游戏音乐制作案例之《诛仙》
- 《鹿鼎记》最新原画
- MIDP2.1规范的新特性
- 3D游戏编程入门经典(6)
- Introduction to 3d game engine design using directx 9 and c#(10)

MORE



您的位置： 开发技术

文章标题	状态驱动的游戏智能体设计（一）		
来源：	［ 翻译:恋花蝶 ］	浏览：	[674]

Finite state machines, or FSMs as they are usually referred to, have for many years been the AI coder’s instrument of choice to imbue a game agent with the illusion of intelligence. You will find FSMs of one kind or another in just about every game to hit the shelves since the early days of video games, and despite the increasing popularity of more esoteric agent architectures, they are going to be around for a long time to come. Here are just some of the reasons why:
就像经常听到的，有限状态机（简写为FSM）早就被AI程序员用来实现游戏智能体以体现智能感。你会发现在FSM几乎是所有视频游戏的基础构架，不管正在出现和流行的越来越深奥的智能体架构，FSM在长久的未来仍然有武之地。这里是一些为什么FSM如此强劲的原因：
They are quick and simple to code.There are many ways of programming a finite state machine and almost all of them are reasonably simple to implement. You’ll see several alternatives described in this article together with the pros and cons of using them.
可以快速简单地编写代码。实现有限状态机有多种途径，而且都可以简单实现。在本文你就能看到多种描述和赞成或者反对使用它们的理由。
They are easy to debug.Because a game agent’s behavior is broken down into easily manageable chunks, if an agent starts acting strangely, it can be debugged by adding tracer code to each state. In this way, the AI programmer can easily follow the sequence of events that precedes the buggy behavior and take action accordingly.
易于调试。因为一个游戏智能体的行为由一个易于管理的代码段来实现，如果一个智能出现奇怪的行为，可以通过为每一个状态增加Tracer来调试。这样能够容易地跟踪事件序列，就可以针对之前的怪异行为修改代码了。
They have little computational overhead.Finite state machines use hardly any precious processor time because they essentially follow hard- coded rules. There is no real “thinking” involved beyond the if-this-then-that sort of thought process.
需要付出一点计算代价。有限状态机几乎不使用宝贵的处理器时间，因为他们本质上跟硬编码是一样的。在那种“如果这样就那样”的思考处理中根本不存在真正的“思考”。
They are intuitive.It’s human nature to think about things as being in one state or another and we often refer to ourselves as being in such and such a state. How many times have you “got yourself into a state” or found yourself in “the right state of mind”? Humans don’t really work like finite state machines of course, but sometimes we find it useful to think of our behavior in this way. Similarly, it is fairly easy to break down a game agent’s behavior into a number of states and to create the rules required for manipulating them. For the same reason, finite state machines also make it easy for you to discuss the design of your AI with non-programmers (with game producers and level designers for example), providing improved communication and exchange of ideas.

符合直觉。人类天生就以当前处以这种或者哪种状态来思考事情，所以我们常常听到我们自己处于什么状态的说法。多少次你“让你自己进入状态”或者发现你自己处于“正确的精神状态”？尽管人类并非真的像有限状态机那样工作，但通常我们发现这样有利于我们思考我们的行为。同样地，这易于通过一系列的状态和创造操作规则来实现一个游戏智能体的行为。基于同样的原因，有限状态机能让你和非程序员（如游戏策划和关卡设计师等）更好地进行关于你的AI设计的讨论，改进交流和交换观点。
They are flexible.A game agent’s finite state machine can easily be adjusted and tweaked by the programmer

- [电子书下载]游戏设计 — 原理与实践
- [电子书下载]网络游戏开发
- 游戏设计全过程
- [电子书下载]游戏设计技术
- [电子书下载]游戏设计理论
- CS游戏人物模型制作教程
- CG人物插画基本流程
- [转贴]MAX高级人头教程

MORE

to provide the behavior required by the game designer. It’s also a simple matter to expand the scope of an agent’s behavior by adding new states and rules. In addition, as your AI skills grow you’ll find that finite state machines provide a solid backbone with which you can combine other techniques such as fuzzy logic or neural networks.
可伸缩性。游戏智能体的有限状态机易于调整，能够很容易让程序员实现游戏设计师需要的行为，也易于通过增加新的状态和规则来扩展智能体的行为。此外，随着你AI技术的增进，你将发现有限状态机提供坚实的基础，让你能够把模糊逻辑和神经网络之类的技术组合到游戏中。

What Exactly Is a Finite State Machine?
有限状态机定义
Historically, a finite state machine is a rigidly formalized device used by mathematicians to solve problems. The most famous finite state machine is probably Alan Turing’s hypothetical device: the Turing machine, which he wrote about in his 1936 paper, “On Computable Numbers.” This was a machine presaging modern-day programmable computers that could perform any logical operation by reading, writing, and erasing symbols on an infinitely long strip of tape. Fortunately, as AI programmers, we can forgo the formal mathematical definition of a finite state machine; a descriptive one will suffice:
从历史观点上来说，有限状态机是一种严格的公式化的被数学家用以解决难题的一种策略。最著名的有限状态机可能是阿兰·图灵在1936年发表的论文《On Computable Numbers》上写下的的猜想——图灵机。这是现代计算机的雏形，能够通过在无限长的磁带上进行读、写和擦除符号来实现所有逻辑操作。幸运的是，作为AI程序员，我们能够对公式化的数学定义不加理会，如下描述已经足够：
A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.
有限状态机是一种策略或者一种策略模型，它由有限的一系列状态构成，在任一给定时刻，可以通过输入操作作出从一种状态到另一种状态的转换或者产生输出或者发生动作。有限状态机在任一时刻都只能够处于一种状态中。

The idea behind a finite state machine, therefore, is to decompose an object’s behavior into easily manageable “chunks” or states. The light switch on your wall, for example, is a very simple finite state machine. It has two states: on and off. Transitions between states are made by the input of your finger. By flicking the switch up it makes the transition from off to on, and by flicking the switch down it makes the transition from on to off. There is no output or action associated with the off state (unless you consider the bulb being off as an action), but when it is in the on state electricity is allowed to flow through the switch and light up your room via the filament in a lightbulb. See Figure 2.1.
因此，有限状态机背后的思想就是把一个对象的行为分解为易于管理的“块”或者状态。例如墙上的电灯开关，就是一种非常简单的有限状态机。它有两个状态：开与关。两个状态通过你指头产生的输入来切换。把开关扳上，它就从关的状态转换到开的状态，把开关扳下，它就从开的状态转换到关的状态。在关的状态没有任何输出或者动作（除非你装灯泡被关掉视为一种动作），但当在开的状态下时电流通过开关并且通过灯泡里的灯丝照亮房间。如图2.1

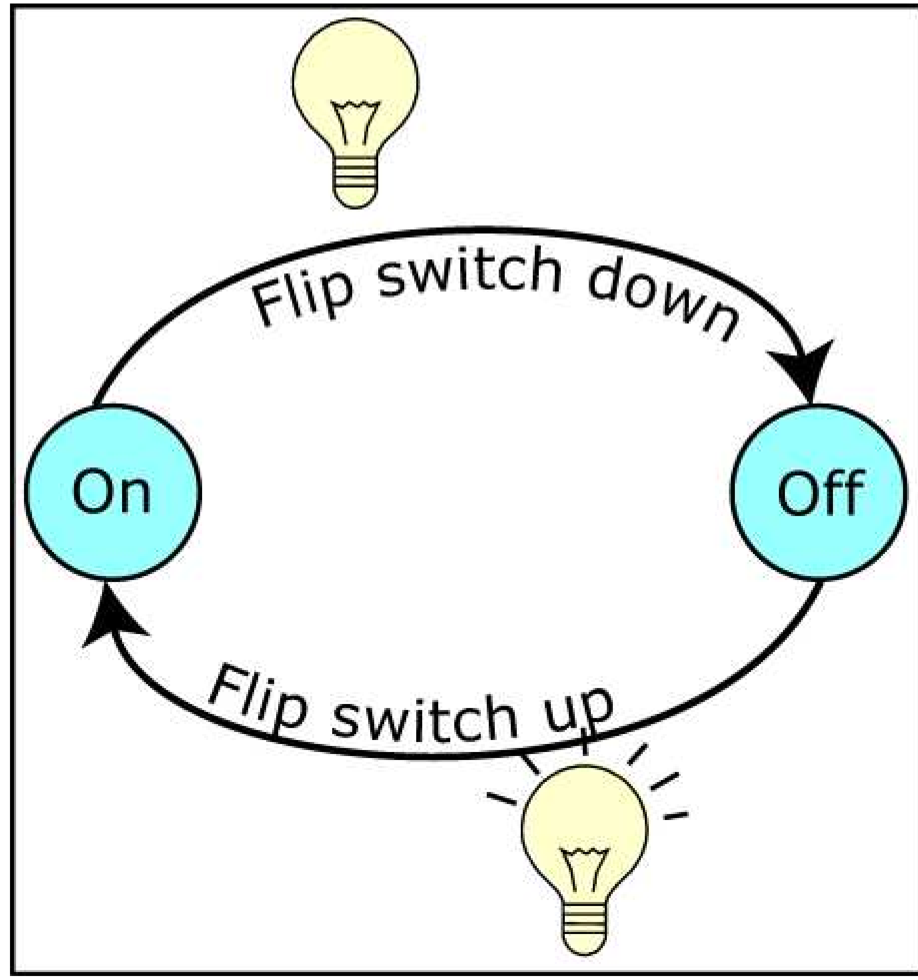


Figure 2.1. A light switch is a finite state machine. (Note that the switches are reversed in Europe and many other parts of the world.)
图2.1 开关是一种有限状态机。（注意：这种开关在欧洲和其它很多国家仍有存在。）
Of course, the behavior of a game agent is usually much more complex than a lightbulb (thank goodness!). Here are some examples of how finite state machines have been used in games.
当然，游戏智能体的行为往往比灯泡要复杂得多。下文是一些在游戏中使用有限状态机的的例子。
本文由恋花蝶最初发表于http://blog.csdn.net/lanphaday，欢迎转载，但必须保持全文完整，也必须包含本声明。
译者并示取得中文版的翻译授权，翻译本文只是出于研究和学习目的。任何人不得在未经同意的情况下将英文版和中文版用于商业行为，转载本文产生的法律和道德责任由转载者承担，与译者无关。
• The ghosts’ behavior in Pac-Man is implemented as a finite state machine. There is one Evade state, which is the same for all ghosts, and then each ghost has its own Chase state, the actions of which are implemented differently for each ghost. The input of the player eating one of the power pills is the condition for the transition from Chase to Evade. The input of a timer running down is the condition for the transition from Evade to Chase.
• Pac-Mac里的精灵的行为用有限状态机实现。所有的精灵都有一种Evade（逃避）状态，它们的实现都是一样的；但每一个精灵都一个Chase（追踪）状态，它的实现各不相同。

Quake-style bots are implemented as finite state machines. They have states such as FindArmor, FindHealth, SeekCover, and RunAway. Even the weapons in Quake implement their own mini finite state machines. For example, a rocket may implement states such asMove, TouchObject, and Die.
• Quake系列的机器人以有限状态机实现。它们FindArmor（找装备）、FindHealth（找补血）、SeekCover（找掩护）和RunAway（逃跑）等多种状态。甚至Quake里实现的武器都带有小型有限状态机，例如一个火箭炮实现的状态就有Move（移动）、TouchObject（触到物体）和Die（死亡）等几种状态。
• Players in sports simulations such as the soccer game FIFA2002 are implemented as state machines. They have states such as Strike, Dribble, ChaseBall, and MarkPlayer. In addition, the teams themselves are often implemented as FSMs and can have states such as KickOff, Defend, or WalkOutOnField.
• FIFA2002之类的运动模拟游戏里的运动员是用状态机实现的，它们有Strike（踢出）、Dribble（带球）、ChaseBall

（逐球）和MarkPlayer（盯人）等状态。此外，整个球队通常也是用FSM实现的，有KickOff（发球）、Defend（防守）和WalkOutOnField（不知道怎么翻译，请足球达人告知一下）。

• The NPCs (non-player characters) in RTSs (real-time strategy games) such as Warcraft make use of finite state machines. They have states such as MoveToPosition, Patrol, andFollowPath.

• RTS（实时策略游戏）（例如Warcraft）中的NPC（非玩家角色）也利用有限状态机。它们的状态有MoveToPosition（移动到某地）、Patrol（巡逻）和FollowPath（跟随）等。

Implementing a Finite State Machine

有限状态机实现

There are a number of ways of implementing finite state machines. A naive approach is to use a series of if-then statements or the slightly tidier mechanism of a switch statement. Using a switch with an enumerated type to represent the states looks something like this:

实现有限状态机有许多方式。一个直观的做法就是使用一系列的if-then语句或者稍显整洁的switch语句。使用switch的实现看起来就像这里的代码：

```
enum StateType{state_RunAway, state_Patrol, state_Attack};
void Agent::UpdateState(StateType CurrentState)
{
switch(CurrentState)
{
case state_RunAway:
EvadeEnemy();
if (Safe())
{
ChangeState(state_Patrol);
}
break;
case state_Patrol:
FollowPatrolPath();
if (Threatened())
{
if (StrongerThanEnemy())
{
ChangeState(state_Attack);
}
else
{
ChangeState(state_RunAway);
}
}
break;
case state_Attack:
if (WeakerThanEnemy())
{
ChangeState(state_RunAway);
}
else
{
BashEnemyOverHead();
}
break;
}//end switch
}
```

Although at first glance this approach seems reasonable, when applied practically to anything more complicated than the simplest of game objects, the switch/if-then solution becomes a monster lurking in the shadows waiting to pounce. As more states and conditions are added, this sort of structure ends up looking like spaghetti very quickly, making the program flow difficult to understand and creating a debugging nightmare. In addition, it’s inflexible and difficult to extend beyond the scope of its original design, should that be desirable… and as we all know, it most often is. Unless you are designing a state machine to implement very simple behavior (or you are a genius), you will almost certainly find yourself first tweaking the agent to cope with unplanned-for circumstances before honing the behavior to get the results you thought you were going to get when you first planned out the state machine!

尽管咋一看这个方案还可以，但只要将其应用到比最简单的游戏对象稍为复杂的实际情况下，这个switch/if-then方案就变成了蛰伏在阴影下的怪物——随时都可能突袭你一下。随着大量的状态和条件的增加，那种结构很快就会变得像意大利面条

一样，使用程序难以理解，并成为调试梦魇。此外，它不可伸缩并且难以在最初的设计范围之外进行扩展，然而我们都知道，它极为常见。除非你用状态机实现非常简单的行为（或者你是一个天才），否则当你第一次策划状态机的时候，在你“磨合”取得的结果和你希望取得的结果之前，你几乎肯定会发现你的智能体无法应用未考虑到的环境。
Additionally, as an AI coder, you will often require that a state perform a specific action (or actions) when it’s initially entered or when the state is exited. For example, when an agent enters the state RunAway you may want it to wave its arms in the air and scream “Arghhhhhhh!” When it finally escapes and changes state to Patrol, you may want it to emit a sigh, wipe its forehead, and say “Phew!” These are actions that only occur when the RunAway state is entered or exited and not during the usual update step. Consequently, this additional functionality must ideally be built into your state machine architecture. To do this within the framework of a switch or if-then architecture would be accompanied by lots of teeth grinding and waves of nausea, and produce very ugly code indeed.
此外，作为一名AI程序员，你经常需要在某一状态实现一种特殊行为（或者一系列行为），比如在进入或者离开某一状态的时候。例如当一个智能体进行RunAway（逃跑）状态时你希望它把武器抛向空中并尖叫一声“Arghhhhhh（啊）！”当它成功逃脱并转换到Patrol（巡逻）状态，你可能想让它喘口气、擦擦额头然后说一声“Phew（呸）！”这些行为都只在进入和离开RunAway（逃跑）状态时才会发生，而不是整个普通的update（更新）阶段都会出现。因此这些额外的功能必须被完美地集成到你的状态机架构里。在switch或者if-then架构里实现这些将让人难以忍受，产生的代码将非常丑陋，不忍卒读。

State Transition Tables
状态转换表
A better mechanism for organizing states and affecting state transitions is a state transition table. This is just what it says it is: a table of conditions and the states those conditions lead to. Table 2.1 shows an example of the mapping for the states and conditions shown in the previous example.
一个能够更好地组织和进行状态转换的机制是状态转换表。顾名思义这就是一个包含条件和条件导致的状态的表。表2.1是前文代码的状态和条件影射表：
Table 2.1. A simple state transition table
表2.1 简单状态转换表

Current State	Condition	State Transition
Runaway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened AND StrongerThanEnemy	Attack
Patrol	Threatened AND WeakerThanEnemy	RunAway

This table can be queried by an agent at regular intervals, enabling it to make any necessary state transitions based on the stimulus it receives from the game environment. Each state can be modeled as a separate object or function existing external to the agent, providing a clean and flexible architecture. One that is much less prone to spaghettification than the if-then/switch approach discussed in the previous section.
智能体隔一定时间查询这个表格，以使得它能够基于从游戏环境接收到的消息来进行必须的状态转换。每一个状态都能够实现为彼此分离的与智能体不耦合的对象或函数，以提供清晰和可伸缩的架构。这一设计不再那么容易像前文讨论的if-then/switch架构那样容易成为意大利面条。
Someone once told me a vivid and silly visualization can help people to understand an abstract concept. Let’s see if it works…
曾有人告诉我一个明晰而无聊的可视物能帮助人们理解抽象的理论，让我们来看看当它工作时……
Imagine a robot kitten. It’s shiny yet cute, has wire for whiskers and a slot in its stomach where cartridges — analogous to its states — can be plugged in. Each of these cartridges is programmed with logic, enabling the kitten to perform a specific set of actions. Each set of actions encodes a different behavior; for example, play_with_string, eat_fish, or poo_on_carpet. Without a cartridge stuffed inside its belly the kitten is an inanimate metallic sculpture, only able to sit there and look cute… in a Metal Mickey kind of way.

想像存在一个机器猫，它闪闪发亮但是非常可爱，有着金属丝制作的胡须并且在胃部有一个插槽——可以依照状态插入可插入模组。每一个可插入模组都是一段逻辑程序，使得小猫能够实现一系列指定的动作。每一系列动作都编码为不同的行为，如play_with_string、eat_fish和poo_on_carpet。如果没有在小猫的胃部插上可插入模组，它就是一件死物——坐在那里，看起来蛮可爱。
The kitten is very dexterous and has the ability to autonomously exchange its cartridge for another if instructed to do so. By providing the rules that dictate when a cartridge should be switched, it’s possible to string together sequences of cartridge insertions permitting the creation of all sorts of interesting and complicated behavior. These rules are programmed onto a tiny chip situated inside the kitten’s head, which is analogous to the state transition table we discussed earlier. The chip communicates with the kitten’s

internal functions to retrieve the information necessary to process the rules (such as how hungry Kitty is or how playful it’s feeling). As a result, the state transition chip can be programmed with rules like:
IF Kitty_Hungry AND NOT Kitty_Playful SWITCH_CARTRIDGE eat_fish
All the rules in the table are tested each time step and instructions are sent to Kitty to switch cartridges accordingly.
这个小猫非常灵巧，并且能够根据指令自动地更换可插入模组。通过给定什么时候该更换可插入模组的规则，它能利用连贯地插入一系列的可插入模组创造所有有趣而复杂的行为。这些与前文讨论的状态转换表相类似的规则被编成程序写入到一个极小的芯片，放置在小猫的头部。芯片与小猫的内部功能通信，以获得处理规则时需要的信息（如Kitty有多饿和它感觉到的好玩度是多少）。状态转换芯片可以用如下的方式编写规则：
IF Kitty_Hungry AND NOT Kitty_Playful SWITCH_CARTRIDGE eat_fish
在每一个时间片都检测表中的所有的规则，从而给Kitty发送指令以切换弹可插入模组。
This type of architecture is very flexible, making it easy to expand the kitten’s repertoire by adding new cartridges. Each time a new cartridge is added, the owner is only required to take a screwdriver to the kitten’s head in order to remove and reprogram the state transition rule chip. It is not necessary to interfere with any other internal circuitry.
这种架构有良好的伸缩性，通过增加新的可插入模组就可以轻易扩展小猫的指令表。每一次增加新的可插入模组，只需要用起子打开小猫的头壳，重编程状态转换规则芯片即可，不需要与其它内部电路打交道。
Embedded Rules
规则内嵌
An alternative approach is to embed the rules for the state transitions within the states themselves. Applying this concept to Robo-Kitty, the state transition chip can be dispensed with and the rules moved directly into the cartridges. For instance, the cartridge for play_with_string can monitor the kitty’s level of hunger and instruct it to switch cartridges for the eat_fish cartridge when it senses hunger rising. In turn the eat_fish cartridge can monitor the kitten’s bowel and instruct it to switch to the poo_on_carpet cartridge when it senses poo levels are running dangerously high.

另一可选的方法是把状态转换规则内嵌到状态本身当中。在对机器猫应用这个概念后，可以去除状态转换芯片，直接将规则内置到可插入模组里。例如play_with_string可插入模组能够监视小猫的饥饿度并适时地命令它切换到eat_fish可插入模组。同样地，eat_fish可插入模组能够监视小猫是否已经吃饱，并在感觉到迫切的排泄感时命令它切换到poo_on_carpet可插入模组。
Although each cartridge may be aware of the existence of any of the other cartridges, each is a self-contained unit and not reliant on any external logic to decide whether or not it should allow itself to be swapped for an alternative. As a consequence, it’s a straightforward matter to add states or even to swap the whole set of cartridges for a completely new set (maybe ones that make little Kitty behave like a raptor). There’s no need to take a screwdriver to the kitten’s head, only to a few of the cartridges themselves.
尽管每一个可插入模组需要知道其它模组的存在，但它们都是自包含的，无论是否要其它模组来替换自己，都并不需要任何外部逻辑来帮助决策。从而可以推论出其中的简明关系，甚至可以用全新的模组集合替换已有的集合（可能这会使用小猫的行为像鸟类一样）。使用这个方案不再需要用起子打开小猫的头部，只要改变可插入模组即可。
Let’s take a look at how this approach is implemented within the context of a video game. Just like Kitty’s cartridges, states are encapsulated as objects and contain the logic required to facilitate state transitions. In addition, all state objects share a common interface: a pure virtual class named State. Here’s a version that provides a simple interface:
现在来看看在视频游戏中如何实现这一方案。像刚才讨论的小猫的可插入模组，可以封装到一个对象里，其中包含辅助状态转换的的逻辑。另外，所有的状态共享一个通用接口：一个命名为State的纯虚类。这里有个接口简单的实现：

```
Class State
{
public:

virtual void Execute (Troll* troll) = 0;
};
```

Now imagine a Troll class that has member variables for attributes such as health, anger, stamina, etc., and an interface allowing a client to query and adjust those values. A Troll can be given the functionality of a finite state machine by adding a pointer to an instance of a derived object of the Stateclass, and a method permitting a client to change the instance the pointer is pointing to.
现在想象Troll类有一系列的数值成员变量：health、anger、stamina等，当然也有相应的接口以查询和设置这些变量值。Troll通过增加一个成员指针变量（指向State类派生类的实例）来增加有限状态机的功能，还提供一个改变指针指向的实例的方法。

```
class Troll
{
/* ATTRIBUTES OMITTED */

State* m_pCurrentState;
```

```
public:

/* INTERFACE TO ATTRIBUTES OMITTED */

void Update()
{
m_pCurrentState->Execute(this);
}

void ChangeState(const State* pNewState)
{
delete m_pCurrentState;
m_pCurrentState = pNewState;
}
};
```

When the Update method of a Troll is called, it in turn calls the Executemethod of the current state type with the this pointer. The current state may then use the Troll interface to query its owner, to adjust its owner's attributes, or to effect a state transition. In other words, how a Troll behaves when updated can be made completely dependent on the logic in its current state. This is best illustrated with an example, so let's create a couple of states to enable a troll to run away from enemies when it feels threatened and to sleep when it feels safe.
当调用Troll的Update方法时，它以this指针为参数调用当前状态类型的Excecute方法。当前状态可能使用Troll的接口查询它的拥有者，设置拥有者的属性或者产生一个状态转换。换句话说，Troll能依赖当前状态的逻辑作出完整行为。这里有最好的例子表达这一观点，让我们来完成两个状态实现以使得Troll能够在危险时逃跑，或者在安全时睡觉。

```
//----------------------------------State_Runaway
class State_RunAway : public State
{
public:

void Execute(Troll* troll)
{
if (troll->isSafe())
{
troll->ChangeState(new State_Sleep());
}
else
{
troll->MoveAwayFromEnemy();
}
}
};

//----------------------------------State_Sleep
class State_Sleep : public State
{
public:

void Execute(Troll* troll)
{
if (troll->isThreatened())
{
troll->ChangeState(new State_RunAway())
}

else
{
troll->Snore();
}
}
};
```

As you can see, when updated, a troll will behave differently depending on which of the states m_pCurrentState points to. Both states are encapsulated as objects and both provide the rules effecting

state transition. All very neat and tidy.
如你所见，当调用Update时，Troll的行为依赖m_pCurrentState指向的状态不同而有所不同。两者状态都封装到对象里，并且都提供了产生状态转换的规则。所有这一切都灵巧而整洁。
This architecture is known as the state design patternand provides an elegant way of implementing state-driven behavior. Although this is a departure from the mathematical formalization of an FSM, it is intuitive, simple to code, and easily extensible. It also makes it extremely easy to add enter and exit actions to each state; all you have to do is create Enter and Exit methods and adjust the agent’s ChangeState method accordingly. You’ll see the code that does exactly this very shortly.

这一架构即是有名的状态设计模式，它提供了雅致的状态驱动行为实现。尽管这有违FSM的数学形式，但它符合直觉、易于编码并且容易扩展。它同样也能够极其容易地增加进入和离开状态时的动作；你需要做只是实现 Enter和Exit方法并相应地改变ChangeState方法。你将可以看到完成这些所产生的代码真的非常短小。

本栏目登载此文出于传递信息之目的，如有任何的问题请及时和我们联系！

无任何评论！

请您注意：

- 尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规
- 尊重网上道德，遵守中华人民共和国的各项有关法律法规
- 承担一切因您的行为而直接或间接导致的民事或刑事法律责任
- 中国网游研发中心新闻留言板管理人员有权保留或删除其管辖留言中的任意内容
- 您在中国网游研发中心留言板发表的作品，中国网游研发中心有权在网站内转载或引用
- 参与本留言即表明您已经阅读并接受上述条款

发表评论：

昵　　称：

联系EMAIL：

GO

关于我们 - 免责声明 - 联络热线 - 申请链接 - 站点地图 - 网站帮助

Copyright © 2004-2007 盛趣信息技术（上海）有限公司 All rights reserved.
OGDEV.NET -- 网络游戏研发网 最佳分辨率 1024×768