# 3．WestWorldWithWoman项目

在前一个项目WestWorld项目的基础上，本文章要创建第二个项目——WestWorldWithWoman。这个项目中West World拥有了另一个居民：艾尔莎（Elsa），她是矿工的妻子。艾尔莎现在还不需要做很多；她主要是全神贯注地清洁小木屋并且倒空她的膀胱（她喝太多的咖啡了）。

程序截图如下：



**图1 程序截图**

## 使State基类可重用

从程序设计的角度出发，为了能很容易地进行扩展，必须创建一个分离的State基类，使每个角色类型从它继承状态。由于我们需要引入另一个居民艾尔莎，为了让状态的使用更方便，我们可以采用泛型类的方法。

public abstract class State<T>

{

 // 当进入一个状态时执行此代码

 public abstract void Enter(T entity);

 // 当状态正常更新时执行此代码

 public abstract void Execute(T entity);

 // 当状态退出时执行此代码

 public abstract void Exit(T entity);

}

对一个具体的类的声明（使用EnterMineAndDigForNugget矿工状态作为一个例子），现在看起来像这样：

public class EnterMineAndDigForNugget : State<Miner>

{

 // 代码省略

}

这样可以使得今后的工作变得更为容易。

## 全局状态和状态翻转（State Blip）

当设计一个有限状态机时，你往往会因为在每一个状态中复制代码而不胜其烦。例如，在Maxis公司的游戏《模拟人生》（The Sims）中，Sim可能会感到本能的迫切要求，不得不去洗手间方便。这种急切的需求会发生在Sim的任何状态或任何可能的时间。假设当前的设计，是为了把这类行为赋予挖金矿工，复制条件的逻辑将会被加进他的每一个状态，或者，放置进Miner.Update方法中。虽然后面的解决方法是可接受的，但最好创建一个全局状态，这样每次FSM更新时就会被调用。那样，所有用于FSM的逻辑被包含在状态中，而不在拥有FSM的智能体类中。

为了实现一个全局状态，需要一个额外的成员变量：

State<Miner> \_globalState;

除了全局行为之外，有时让智能体退出一个状态时能返回到前一个状态会带来方便，我们称这种行为为**状态翻转（State Blip）**。例如，正如在Sims中，你可能会坚持你的智能体可以在任何时候去到洗手间，但要确保他总能返回先前的状态。为了赋赋予FSM这种功能，必须保持前一个状态的纪录，从而使状态翻转可以回到前一个状态。这非常容易做到，因为所需要做的就是在Miner.ChangeState方法中增加另一个成员变量和一些附加的逻辑。

那么到现在，为了实现这些附加的成分，Miner类已经获得两个额外的成员变量和一个附加的方法，代码如下：

class Miner : public BaseGameEntity

{

 private State<Miner> \_currentState;

 private State <Miner> \_previous State

 private State<Miner> \_globaistate ;

 public void ChangeStata (State<Miner> newState){}

public void RevertToPreviousState (){}

}

## 创建一个StateMachine类

通过把所有与状态有关的数据和方法封装到一个StateMachine类中，可以使得设计更为简洁。这种方式下一个智能体可以拥有一个StateMachine类的实例，并且委托它管理当前状态、全局状态和前一个状态。

下面是State Machine类的代码：

public class StateMachine<T>

{

 T \_owner; // 包含当前状态机的对象

 State<T> \_currentState; // 当前状态

 State<T> \_previousState; // 前一个状态

 State<T> \_globalState; // 全局状态，在Update方法中进行更新

 // 获取三个状态

 public State<T> CurrentState { get { return \_currentState; } }

 public State<T> GlobalState { get { return \_globalState; } }

 public State<T> PreviousState { get { return \_previousState; } }

 public StateMachine(T owner)

 {

 \_owner = owner;

 \_currentState = null;

 \_previousState = null;

 \_globalState = null;

 }

 // 使用以下方法初始化FSM

 public void SetCurrentState(State<T> s) { \_currentState = s; }

 public void SetGlobalState(State<T> s) { \_globalState = s; }

 public void SetPreviousState(State<T> s) { \_previousState = s; }

 // 更新FSM

 public void Update()

 {

 // 如果存在全局状态，则执行它的Execute方法

 if (\_globalState != null)

 \_globalState.Execute(\_owner);

 // 如果存在当前状态，则执行它的Execute方法

 if (\_currentState != null)

 \_currentState.Execute(\_owner);

 }

 // 切换到一个新状态

 public void ChangeState(State<T> pNewState)

 {

 // 保存前一个状态的记录

 \_previousState = \_currentState;

 // 调用当前状态的Exit方法

 \_currentState.Exit(\_owner);

 // 切换到新状态

 \_currentState = pNewState;

 // 调用新状态的Enter方法

 \_currentState.Enter(\_owner);

 }

 // 将状态返回到前一个状态

 public void RevertToPreviousState()

 {

 ChangeState(\_previousState);

 }

}

一个智能体所需要的做的全部事情就是去拥有一个StateMachine类的实例，并且为了得到完全的FSM功能，实现一个方法来更新状态机。

改进的Miner类如下所示：

public class Miner : BaseGameEntity

{

 private StateMachine<Miner> \_stateMachine;// 当前智能体所拥有的有限状态机对象

// 代码省略…

 public Miner(int id)

 : base(id)

 {

 // 代码省略…

 // 建立有限状态机

 \_stateMachine = new StateMachine<Miner>(this);

 \_stateMachine.SetCurrentState(GoHomeAndSleepTilRested.Instance());

 }

 /// <summary>

 /// 获取有限状态机

 /// </summary>

 public StateMachine<Miner> FSM { get { return \_stateMachine; } }

 public override void Update()

 {

 \_thirst += 1;

 \_stateMachine.Update();

}

// 以下代码省略…

}

## 艾尔莎实体和艾尔莎的状态

艾尔莎智能体的MinersWife类的代码非常简单：

public class MinersWife : BaseGameEntity

{

 private StateMachine<MinersWife> \_stateMachine;

 private location\_type \_location;

 public location\_type Location { get { return \_location; } }

 public StateMachine<MinersWife> FSM { get { return \_stateMachine; } }

 public MinersWife(int id)

 : base(id)

 {

 \_location = location\_type.shack;

 \_stateMachine = new StateMachine<MinersWife>(this);

 \_stateMachine.SetCurrentState(DoHouseWork.Instance());

 \_stateMachine.SetGlobalState(WifesGlobalState.Instance());

 }

 public override void Update()

 {

 \_stateMachine.Update();

 }

 public void ChangeLocation(location\_type loc)

 {

 \_location = loc;

 }

}

当一个StateMachine类被实例化时，注意当前状态和全局状态时如何被设置的。

Elsa的状态转换图如图2显示。



**图2 Elsa的状态转换图。全局状态没有显示在图中，因为它的逻辑是有效地实现在任何一个状态中并且从不改变的**

艾尔莎所拥有的的状态类的代码如下：

//------------------------------------------------------------------------

// Elsa的全局状态，包含上厕所的逻辑

//------------------------------------------------------------------------

public class WifesGlobalState : State<MinersWife>

{

 private WifesGlobalState() { }

 private static WifesGlobalState \_wifesGlobalState;

 // 实现Singleton设计模式

 public static WifesGlobalState Instance()

 {

 if (\_wifesGlobalState == null)

 \_wifesGlobalState = new WifesGlobalState();

 return \_wifesGlobalState;

 }

 public override void Enter(MinersWife wife) { }

 public override void Execute(MinersWife wife)

 {

 // 有十分之一的概率上厕所

 if (Global.random.NextDouble() < 0.1)

 {

 wife.FSM.ChangeState(VisitBathroom.Instance());

 }

 }

 public override void Exit(MinersWife wife) { }

}

//------------------------------------------------------------------------

// Elsa做家务的状态

//------------------------------------------------------------------------

public class DoHouseWork : State<MinersWife>

{

 private DoHouseWork() { }

 private static DoHouseWork \_doHouseWork;

 // 实现Singleton设计模式

 public static DoHouseWork Instance()

 {

 if (\_doHouseWork == null)

 \_doHouseWork = new DoHouseWork();

 return \_doHouseWork;

 }

 public override void Enter(MinersWife wife) { }

 // 艾尔莎各有三分之一的概率进行拖地板、洗碗或整理床铺

 public override void Execute(MinersWife wife)

 {

 switch ((int)Global.random.Next(3))

 {

 case 0:

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 拖地板";

 break;

 case 1:

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 洗碗";

 break;

 case 2:

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 整理床铺";

 break;

 }

 }

 public override void Exit(MinersWife wife) { }

}

//------------------------------------------------------------------------

// Elsa上厕所的状态

//------------------------------------------------------------------------

public class VisitBathroom : State<MinersWife>

{

 private VisitBathroom() { }

 private static VisitBathroom \_visitBathroom;

 // 实现Singleton设计模式

 public static VisitBathroom Instance()

 {

 if (\_visitBathroom == null)

 \_visitBathroom = new VisitBathroom();

 return \_visitBathroom;

 }

 public override void Enter(MinersWife wife)

 {

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 需要上厕所";

 }

 public override void Execute(MinersWife wife)

 {

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 啊哈！现在放松了！";

 wife.FSM.RevertToPreviousState();

 }

 public override void Exit(MinersWife wife)

 {

 Global.OutputMessage += "\n" + Global.GetNameOfEntity(wife.ID) + ": 离开厕所";

 }

}

注意看VisitBathroom状态是如何作为一个翻转状态（即，它总是返回到前面的状态）实现的。同样要注意一个全局状态WifesGlobalState也被定义了，它包含了对Elsa去厕所需要的逻辑。这个逻辑之所以包含在一个全局状态中是因为Elsa可能在任何状态、任何时候感到有这种自然的需求。