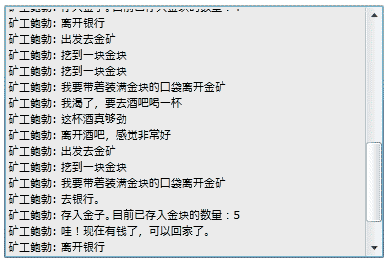
# 2．West World项目

作为一个关于使用限状态机创建一个智能体的实际例子，我们将着眼于一个游戏的环境，是智能体居住的的一个古老的西部风格的开采金矿的小镇，称作West World。最初可能只有一个居民（一个挖金矿工，叫做矿工鲍勃Bob）。但是在后面他的妻子将出会现。因为West World是作为一个简单的基于文本的控制台应用实现的，所以你将不得不想象遍地的风滚草，叽叽嘎嘎的矿井支柱，时有荒漠旧灰尘吹进你的眼睛。任何状态的改变或者状态动作的输出将作为文文传送到控制台窗口。使用这种只有普通文本的方法是因为它能将有限状态机的机制演示清楚而不会由于更复杂的环境而增加编码混乱。

在WestWorld中有4个位置：一个金矿，1个银行使鲍勃可以在那存放他找到的天然金块，一个酒吧间使他可以解除干渴，还有家——使他在疲劳后可以睡觉。他准确地向哪走，他到达后要干什么，这都由Bob当前的状态决定。他将依赖于变量如口渴、疲劳和在金矿下面他找到了多少金子来改变他的状态。

在我们钻研源码前，检查一下如下West World实例执行程序的输出信息，如下图所示：



**图1 West World程序的输出信息**

在游戏的输出中，每次你看到矿工鲍勃改变位置即表示他在改变状态。所有其他的事件都是发生在状态中的动作。下面解释一下演示的编码结构。

## BaseGameEntity类

West World的所有居民是从基类BaseGameEntity继承来的。这是一个简单的带有一个用于存储ID号码的私有变量的类。它也指定一个抽象函数Update，必须由所有的子类执行。Update是一个函数，在每个更新步骤它都要被调用，并且被子类用来更新它们的状态机以及在每个更新步骤中必须更新的任何其他数据。

BaseGameEntity类的代码如下：

public abstract class BaseGameEntity

{

int \_id; // 每个实体都必须拥有一个唯一的表示数字

// 下一个实体的ID。每实例化一个BaseGameEntity时都会更新这个变量。

static int m\_iNextValidID;

/// <summary>

/// 获取实体的ID

/// </summary>

public int ID { get { return \_id; } }

public BaseGameEntity(int id)

{

SetID(id);

}

public abstract void Update();

private void SetID(int val)

{

\_id = val;

m\_iNextValidID = \_id + 1;

}

}

在游戏中每一个实体都有一个唯一的标识符是非常重要的。因此在实例化时，传递给构造函数的ID在SetID方法中进行测试来确保ID是唯一的。本例中，实体将使用一个枚举值作为它们唯一的标识符。这些可以在Global.cs文件中作为ent\_Miner\_Bob和ent\_Elsa被找到。

Global.cs文件中包含了程序用到的枚举和全局变量，代码如下：

public enum location\_type

{

shack, // 窝棚

goldmine, // 金矿

bank, // 银行

saloon // 酒吧

};

public enum ent\_type

{

ent\_Miner\_Bob, // 矿工鲍勃

ent\_Elsa // 艾尔莎

};

/// <summary>

/// 包含全局变量的静态类

/// </summary>

public static class Global

{

// 富裕程度，超过这个值矿工会回家休息。

public const int ComfortLevel = 5;

// 矿工能携带的最大金块数量

public const int MaxNuggets = 3;

// 口渴上限，超过这个值矿工就会感到口渴

public const int ThirstLevel = 5;

// 疲劳上限，超过这个值矿工就会感到疲劳，需要睡觉进行恢复

public const int TirednessThreshold = 5;

// 输出的信息

public static string OutputMessage;

// 根据实体的编号获取它的名称

public static string GetNameOfEntity(int id)

{

switch (id)

{

case (int)ent\_type.ent\_Miner\_Bob:

return "矿工鲍勃";

case (int)ent\_type.ent\_Elsa:

return "艾尔莎";

default:

return "UNKNOWN!";

}

}

}

## Miner类

Miner类是从BaseGameEntitv类中继承的，并且包含着代表矿工拥自的的各种各样的特性数据成员，例如它的财富、它的疲劳程度、它的位置，等等。类似于本章的前面介绍的troll例子。一个Miner除了拥有一个State类的实例，还有一个方法用于改变某个状态。

Miner的Update方法是非常简单的的，在调用当前状态的Execute方法之前它只是增加了\_thirst的值。具体代码如下：

namespace WestWorld1

{

public class Miner : BaseGameEntity

{

private State \_currentState; // 矿工的当前状态

private location\_type \_location; // 矿工当前所处的位置

int \_goldCarried; // 矿工挖到的金块数量

int \_moneyInBank; // 矿工在银行中的存款

private int \_thirst; // 矿工的口渴程度，这个值越高越口渴

private int \_fatigue; // 矿工的疲劳程度，这个值越高越疲劳

/// <summary>

/// 创建一个矿工对象

/// </summary>

public Miner(int id)

: base(id)

{

\_location = location\_type.shack;

\_goldCarried = 0;

\_moneyInBank = 0;

\_thirst = 0;

\_fatigue = 0;

\_currentState = GoHomeAndSleepTilRested.Instance();

}

/// <summary>

/// 获取矿工所处的位置

/// </summary>

public location\_type Location

{

get { return \_location; }

}

/// <summary>

/// 获取或设置矿工拥有的金块

/// </summary>

public int GoldCarried

{

get { return \_goldCarried; }

set { \_goldCarried = value; }

}

/// <summary>

/// 装金子的口袋是否已满？

/// </summary>

public bool PocketsFull

{

get { return \_goldCarried >= Global.MaxNuggets; }

}

/// <summary>

/// 获取或设置矿工在银行中的存款

/// </summary>

public int Wealth

{

get { return \_moneyInBank; }

set { \_moneyInBank = value; }

}

/// <summary>

/// 矿工是否已经非常口渴

/// </summary>

public bool Thirsty

{

get

{

if (\_thirst >= Global.ThirstLevel) { return true; }

return false;

}

}

/// <summary>

/// 矿工是否已经非常疲劳

/// </summary>

public bool Fatigued

{

get

{

if (\_fatigue > Global.TirednessThreshold) { return true; }

return false;

}

}

public override void Update()

{

\_thirst += 1;

if (\_currentState!=null)

{

\_currentState.Execute(this);

}

}

// 将当前状态修改为新的状态。

// 首先调用当前状态的Exit()方法，然后将新状态赋予\_currentState，

// 最后调用新状态的Enter()方法。

public void ChangeState(State newState)

{

\_currentState.Exit(this);

\_currentState = newState;

\_currentState.Enter(this);

}

/// <summary>

/// 改变矿工所处的位置

/// </summary>

public void ChangeLocation(location\_type loc)

{

\_location = loc;

}

/// <summary>

/// 添加矿工所携带的金块数量

/// </summary>

public void AddToGoldCarried(int val)

{

\_goldCarried += val;

if (\_goldCarried < 0) \_goldCarried = 0;

}

/// <summary>

/// 添加矿工的财富值

/// </summary>

public void AddToWealth(int val)

{

\_moneyInBank += val;

if (\_moneyInBank < 0)

\_moneyInBank = 0;

}

/// <summary>

/// 让矿工到酒吧买酒解渴，他的财富会减2，但口渴程度变成0

/// </summary>

public void BuyAndDrinkAWhiskey()

{

\_thirst = 0;

\_moneyInBank -= 2;

}

/// <summary>

/// 减少疲劳度

/// </summary>

public void DecreaseFatigue()

{

\_fatigue -= 1;

}

/// <summary>

/// 增加疲劳度

/// </summary>

public void IncreaseFatigue()

{

\_fatigue += 1;

}

}

}

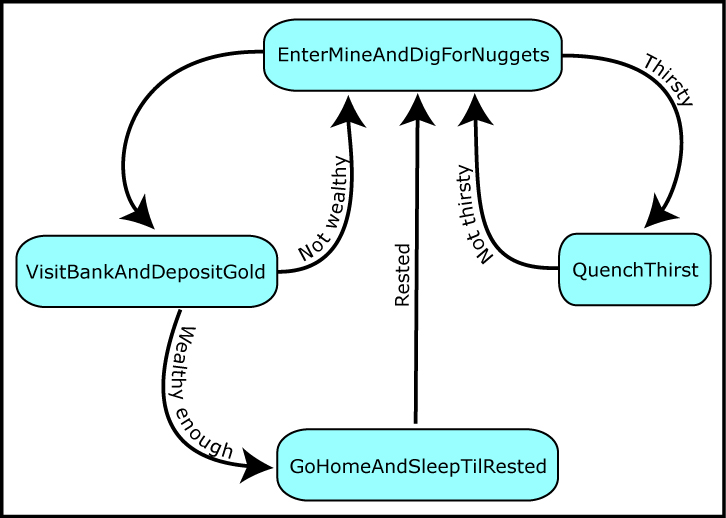
现在你已经了解Miner类是如何操作的，下面看看一个矿工可能处于的每一种状态。

## Miner的状态

矿工可能会进入4种状态中的一种。这里给出那些状态的名字，随后是动作的描述以及在这些状态中发生的状态变换。

* **EnterMineAndDigForNugget**：如果矿工没有在金矿里，他将位置改变到金矿。如果已经在金矿里了，他会挖掘金块。当他的口袋装满了，鲍勃会将状态改变到**VisitBankAndDepositGold**，如果挖掘时他发现自己很渴，他会停下来并且改变状态到**QuenchThirst**。
* **VisitBankAndDepositGold**：在这个状态里，矿工将会走到银行并且存储他携带的所有金块。之后如果他认为他自己已经足够有钱了，他将会改变状态到**GoHomeAndSleepTiIRested**。否则他会改变状态到**EnterMineAndDigForNugget**。
* **GoHomeAndSleepTiIRested**：在这个状态里，矿工将会回到他的小木屋睡觉直到他的疲劳等级下降到一个可以接受的等级之下。然后他会改变状态到**EnterMineAndDigForNugget**。
* **QuenchThirst**：如果在任何时候矿工感到口渴，他改变他的状态并且造访酒吧为自己买一杯威士忌。当解除了他的口渴问题，他改变状态到**EnterMineAndDigForNugget**。

通过这样的文本描述很难领会状态逻辑的流动，此时拿起笔和纸来为你的游戏智能体画一个状态转换图，这通常是很有帮助的。图2显示了一个金矿工人的状态转换图。方框代表着单独的状态，它们之间的线是可用的变换。



**图2 矿工鲍勃的状态转换图**

这样的一张图可视性更好，并且可以非常容易地发现任何逻辑流上的错误。

## 状态设计模式

前面已经简述了这个设计模式。每一个游戏的智能体的状态是作为一个唯一的类实现的，并且每个智能体拥有一个当前状态的实例。智能体也实现一个ChangeState方法，无论何时需要状态变换时可以被调用来进行状态变换。决定任何状态变换的逻辑包含在每个State类中，所有的状态类是从一个抽象的基类继承来的。

前面提到过，每个状态有相关联的进入和退出动作常常是很好的。这允许程序员编写只在状态的进入或退出时执行一次的逻辑，并且大大的增加了一个FSM的灵活性。让我们看一下这个加强的State基类：

namespace WestWorld1

{

public abstract class State

{

// 当进入一个状态时执行此代码

public abstract void Enter(Miner miner);

// 当状态正常更新时执行此代码

public abstract void Execute(Miner miner);

// 当状态退出时执行此代码

public abstract void Exit(Miner miner);

}

}

这些增加的方法只在矿工改变状态时被调用。当一个状态变换发生时，Miner的ChangeState方法首先调用当前状态的Exit方法，然后它分配一个新的状态给当前的状态，并且以调用新状态（现在已经是当前状态了）的Enter方法结束。下面是Miner的ChangeState方法的代码：

public void ChangeState(State newState)

{

\_currentState.Exit(this);

\_currentState = newState;

\_currentState.Enter(this);

}

注意一个矿工Miner是如何将this指针传递给每个状态的，这使得状态可以访问Miner的任何相关的数据。

一个矿工可以访问的4个可能的状态中的每一个都是从State类继承来的，并提供具体的类：EnterMineAndDigForNugget，VisitBankAndDepositGold，GoHomeAndSleepTiIRested和QuenchThirst。Miner类的\_currentState可以指向这些状态中的任何一个。当Miner中的Update方法被调用，它接着调用当前活动状态的Execute方法，以this指针作为参数。

每个具体的状态被简化为1个Singleton对象。这是为了确保每一个状态只有一个实例，这是智能体共享的。使用Singleton使得设计更为有效，因为它消除了在状态每次改变时分配和释放内存的需要。如果你有很多的智能体共享一个复杂的FSM，和（或）你正在为只有有限的资源的机器进行开发时，这是非常重要的。

**注意**：笔者喜欢将Singleton用于状态，但是它有一个缺点。因为它们在客户之间是共享的，Singleton状态无法使用自身局部的、智能体专用的数据。例如，如果一个智能体使用一个状态，当进入状态应该移动它到一个任意的位置，这个位置不能被存储在状态当中（因为对于每个使用这个状态的智能体来说，位置可能是不同的），而是不得不在外部某处存储并且状态要经由智能体才能访问。如果你的智能体访问的仅仅是一两个数据，这还不是个问题，但是如果发现你设计的状态要重复地访问大量的外部数据，就应该考虑放弃Singleton设计模式，并且写一些代码行来管理分配和释放状态内存。

## Singleton设计模式

你常常会发现SingIeton（单例模式）非常有用，因为它确保了一个对象只能实例化一次，和（或）它是全局可访问的。例如，在游戏设计中设计的是包含许多不同实体类型的环境（玩家、怪物、抛射体，植物区域等等），常常需要存在一个“管理者”对象来完成创建，删除，和管理这些对象的工作。具有这个对象（一个Singleton） 的一个实例是必要的，对它可以进行全局访问是方便的，因为许多其他的对象会需要访问它。

Singleton模式确保了这两个性质。有许多实现一个singleton的方法。下面的代码中采取的方法是使用一个静态的方法Instance，它返回指向这个类的一个静态实例。

public class EnterMineAndDigForNugget : State

{

// 构造函数设置为私有，外部代码不能直接对这个类进行实例化

private EnterMineAndDigForNugget() { }

// 声明一个私用的静态类变量

private static EnterMineAndDigForNugget \_enterMineAndDigForNugget;

// 实现Singleton设计模式

public static EnterMineAndDigForNugget Instance()

{

if (\_enterMineAndDigForNugget == null)

\_enterMineAndDigForNugget = new EnterMineAndDigForNugget();

return \_enterMineAndDigForNugget;

}

// 以下代码省略

……………

}

注意：如果Singleton对你来说是一个新的概念，并且你决定搜索互联网来寻找更多的信息，你会发现它们激起了许多关于面向对象的软件设计的争论。是的，程序员热衷于争论这个问题，并且没有什么是比讨论全局变量或者伪装成全局变量的对象，例如Singlcton更能挑起一个争论。无论在什么情况下，在不能损害设计的前提下，只要是它们提供了方便就使用它们。尽管如此，还是建议你读一读支持的或反对的论点，并得出你自己的结论。

下面通过仔细阅读一种矿工状态的完整代码，我们看看每一件事是如何适配在一起的。

## EnterMineAndDigForNugget状态

在这个状态矿工应该改变位置到金矿里面。一旦在金矿中，他应该挖掘金子直到把口袋装满，这时他应该改变状态到**VisitBankAndDepositNugget**。如果存挖掘过程中矿工觉得口渴，他应该改变状态到**QuenchThirst**。

因为具体的状态仅仅实现了定义在抽象基类State中的函数声明，所以它的代码是非常简单的：

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

// 矿工去金矿采金块。

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

public class EnterMineAndDigForNugget : State

{

// 构造函数设置为私有，外部代码不能直接对这个类进行实例化

private EnterMineAndDigForNugget() { }

// 声明一个私用的静态类变量

private static EnterMineAndDigForNugget \_enterMineAndDigForNugget;

// 实现Singleton设计模式

public static EnterMineAndDigForNugget Instance()

{

if (\_enterMineAndDigForNugget == null)

\_enterMineAndDigForNugget = new EnterMineAndDigForNugget();

return \_enterMineAndDigForNugget;

}

public override void Enter(Miner miner)

{

// 如果矿工不在金矿中，则必须将他所处的位置设置为金矿

if (miner.Location != location\_type.goldmine)

{

Global.OutputMessage += "\n" + Global.GetNameOfEntity(miner.ID) + ": 出发去金矿";

miner.ChangeLocation(location\_type.goldmine);

}

}

public override void Execute(Miner miner)

{

// 矿工会一直采金块直到携带的金块数量超过MaxNuggets。

// 如果他渴了，会离开金矿去酒吧喝一杯威士忌。

miner.AddToGoldCarried(1);

miner.IncreaseFatigue();

Global.OutputMessage += "\n" + Global.GetNameOfEntity(miner.ID) + ": 挖到一块金块";

// 如果挖到足够的金矿，则将它们存入银行

if (miner.PocketsFull)

{

miner.ChangeState(VisitBankAndDepositGold.Instance());

}

// 如果渴了，则去酒吧

if (miner.Thirsty)

{

miner.ChangeState(QuenchThirst.Instance());

}

}

public override void Exit(Miner miner)

{

Global.OutputMessage += "\n" + Global.GetNameOfEntity(miner.ID) + ": 我要带着装满金块的口袋离开金矿";

}

}

其余三个状态**VisitBankAndDepositNugget**、**GoHomeAndSleepTilRested**和**QuenchThirst**不再赘述，你可以自己看源代码。

现在你已经了解了怎样使用状态设计模式来创建一个非常灵活的机制用于状态驱动的智能体。按照需求来增加一个额外的状态是非常简单的。1个非常复杂的设计可以由几个独立的小的状态机集成，这将是非常实用的。例如，第一人称射击游戏（FPS）像Unreal2的状态机趋向于大的和复杂的。当设计一个这种类型的人工智能游戏时，你会发现更可取的是依据几个更小的代表功能性的如“defend the flag”或“explore map”的状态机来思考，当适当的时候它可以被选择成加入或去除。状态设计模式可以轻松实现这些情况。