状态模式

游戏设计模式 / Design Patterns Revisited

忏悔时间：我有些越界，将太多的东西打包到了这章中。 它表面上关于状态模式[GoF]，但我无法只讨论它和游戏，而不涉及更加基础的有限状态机（FSMs）。 但是一旦讲了那个，我发现也想要介绍层次状态机和下推自动机。

有很多要讲，我会尽可能简短，这里的示例代码留下了一些你需要自己填补的细节。 我希望它们仍然足够清晰，能让你获取一份全景图。

如果你从来没有听说过状态机，不要难过。 虽然在AI和编译器程序方面很出名，但它在其他编程圈就没那么知名了。 我认为应该有更多人知道它，所以在这里我将其运用在不同的问题上。

> 这些状态机术语来自人工智能的早期时代。 在五十年代到六十年代，很多AI研究关注于语言处理。 很多现在用于分析程序语言的技术在当时是发明出来分析人类语言的。

感同身受

假设我们在完成一个卷轴平台游戏。 现在的工作是实现玩家在游戏世界中操作的女英雄。这就意味着她需要对玩家的输入做出响应。按B键她应该跳跃。简单实现如下：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

看到漏洞了吗？

没有东西阻止“空中跳跃”——当角色在空中时狂按B，她就会浮空。 简单的修复方法是给`Heroine`增加`isJumping_`布尔字段，追踪它跳跃的状态。然后这样做：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_)
    {
      isJumping_ = true;
      // 跳跃……
```

```
    }
  }
}
```

> 这里也应该有在英雄接触到地面时将isJumping_设回false的代码。 我在这里为了简明没有写。

接下来，当玩家按下下方向键时，如果角色在地上，我们想要她卧倒，而松开按键时站起来：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // 如果没在跳跃，就跳起来……
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

这次看到漏洞了吗？

通过这个代码，玩家可以：

1. 按下键卧倒。
2. 按B从卧倒状态跳起。
3. 在空中放开下键。

英雄跳一半贴图变成了站立时的贴图。是时候增加另一个标识了……

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // 跳跃……
    }
  }
  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，如果玩家在跳跃途中按下下方向键，英雄能够做跳斩攻击就太酷了：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // 跳跃……
    }
  }
  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_)
    {
      // 站立……
    }
  }
}
```

又是检查漏洞的时间了。找到了吗？

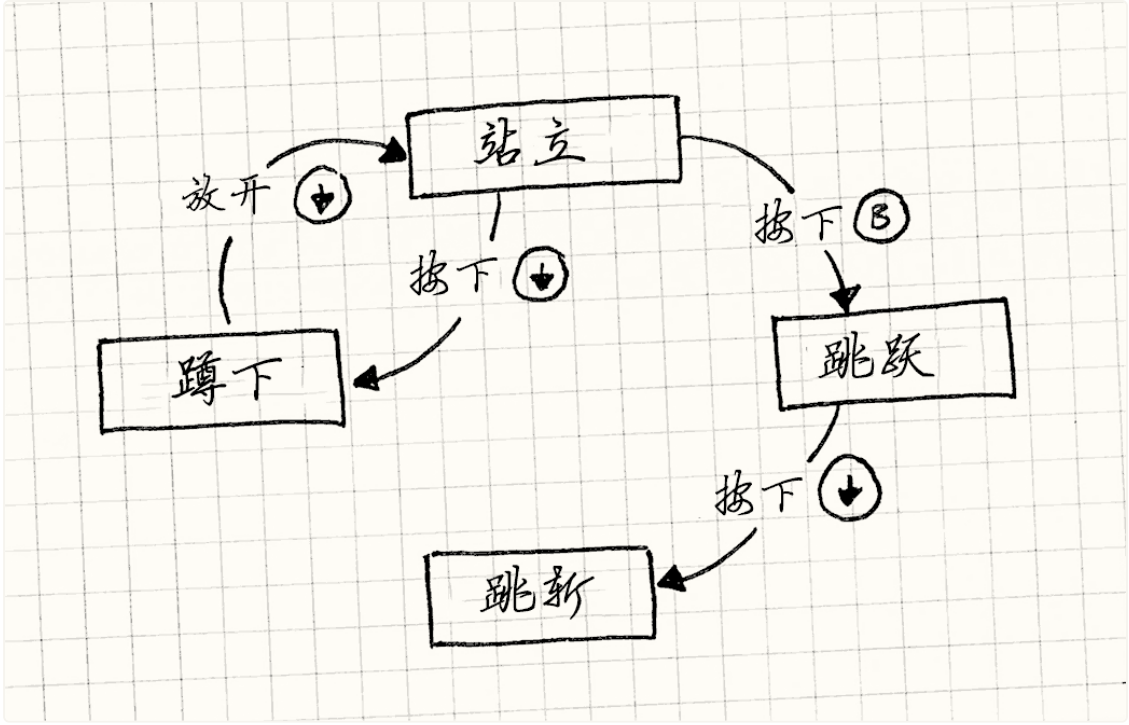
跳跃时我们检查了字段，防止了空气跳，但是速降时没有。又是另一个字段……

我们的实现方法很明显有错。 每次我们改动代码时，就破坏些东西。 我们需要增加更多动作——行走 都还没有加入呢——但以这种做法，完成之前就会造成一堆漏洞。

> 那些你崇拜的、看上去永远能写出完美代码的程序员并不是超人。 相反，他们有**哪种**代码易于出错的直觉，然后避开。
>
> 复杂分支和可变状态——随时间改变的字段——是两种易错代码，上面的例子覆盖了两者。

有限状态机前来救援

在经历了上面的挫败之后，把桌子扫空，只留下纸笔，我们开始画流程图。 你给英雄每件能做的事情都画了一个盒子：站立，跳跃，俯卧，跳斩。 当角色在能响应按键的状态时，你从那个盒子画出一个箭头，标记上按键，然后连接到她变到的状态。



祝贺，你刚刚建好了一个有限状态机。 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。 FSMs是其中最简单的成员。

要点是：

- 你拥有状态机所有可能状态的集合。 在我们的例子中，是站立，跳跃，俯卧和速降。
- 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状态。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状态机。 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移，每个转移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移相匹配，机器转换为所指的状态。

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

这就是核心部分的全部了：状态，输入，和转移。 你可以用一张流程图把它画出来。不幸的是，编译器不认识流程图， 所以我们如何实现一个？ GoF的状态模式是一个方法——我们会谈到的——但先从简单的开始。

> 对FSMs我最喜欢的类比是那种老式文字冒险游戏，比如Zork。 你有个由屋子组成的世界，屋子彼此通过出口相连。你输入像“去北方”的导航指令探索屋子。
>
> 这其实就是状态机：每个屋子都是一个状态。 你现在在的屋子是当前状态。每个屋子的出口是它的转移。 导航指令是输入。

枚举和分支

`Heroine`类的问题在于它不合法地捆绑了一堆布尔量： `isJumping_`和`isDucking_`不会同时为真。 但有些标识同时只能有一个是`true`，这提示你真正需要的其实是`enum`（枚举）。

在这个例子中的`enum`就是FSM的状态的集合，所以让我们这样定义它：

```
enum State
{
  STATE_STANDING,
  STATE_JUMPING,
  STATE_DUCKING,
  STATE_DIVING
};
```

不需要一堆标识，Heroine只有一个state_状态。 这里我们同时改变了分支顺序。在前面的代码中，我们先判断输入，然后 判断状态。 这让处理某个按键的代码集中到了一处，但处理某个状态的代码分散到了各处。 我们想让处理状态的代码聚在一起，所以先对状态做分支。这样的话：

```
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;
  }
}
```

这看起来很普通，但是比起前面的代码是个很大的进步。 我们仍有条件分支，但简化了状态变化，将它变成了字段。 处理同一状态的所有代码都聚到了一起。 这是实现状态机最简单的方法，在某些情况下，这也不错。

重要的是，英雄不再会处于**不合法**状态。 使用布尔标识，很多可能存在的值的组合是不合法的。 通过enum，每个值都是合法的。

但是，你的问题也许超过了这个解法的能力范围。 假设我们想增加一个动作动作，英雄可以俯卧一段时间充能，之后释放一次特殊攻击。 当她俯卧时，我们需要追踪充能的持续时间。

我们为Heroine添加了chargeTime_字段，记录充能的时间长度。 假设我们已经有一个每帧都会调用的update()方法。在那里，我们添加：

```
void Heroine::update()
{
  if (state_ == STATE_DUCKING)
  {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      superBomb();
    }
  }
}
```

> 如果你猜这就是更新方法模式，恭喜你答对了！

我们需要在她开始俯卧的时候重置计时器，所以我们修改`handleInput()`：

```
void Heroine::handleInput(Input input)
{
  switch (state_)
  {
    case STATE_STANDING:
      if (input == PRESS_DOWN)
      {
        state_ = STATE_DUCKING;
        chargeTime_ = 0;
        setGraphics(IMAGE_DUCK);
      }
      // 处理其他输入……
      break;

      // 其他状态……
  }
}
```

总而言之，为了增加这个充能攻击，我们需要修改两个方法， 添加一个`chargeTime_`字段到`Heroine`，哪怕它只在俯卧时有意义。 我们更喜欢的是让所有相关的代码和数据都待在同一个地方。GoF完成了这个。

状态模式

对于那些思维模式深深沉浸在面向对象的人，每个条件分支都是使用动态分配的机会（在C++中叫做虚方法调用）。 我觉得那就太过于复杂化了。有时候一个`if`就能满足你的需要了。

> 这里有个历史遗留问题。 原先的面向对象传教徒，比如写《设计模式》的GoF和写《重构》的Martin Fowler都使用Smalltalk。 那里，`ifThen:`只是个由你在一定情况下使用的方法，该方法在`true`和`false`对象中以不同的方式实现。

但是在我们的例子中，面向对象确实是一个更好的方案。 这带领我们走向状态模式。GoF这样描述状态模式：

> 允许一个对象在其内部状态发生变化时改变自己的行为，该对象看起来好像修改了它的类型

这可没太多帮助。我们的`switch`也完成了这一点。 它们描述的东西应用在英雄的身上实际是：

一个状态接口

首先，我们为状态定义接口。 状态相关的行为——之前用switch的每一处——都成为了接口中的虚方法。 在我们的例子中，那是handleInput()和update()：

```
class HeroineState
{
public:
  virtual ~HeroineState() {}
  virtual void handleInput(Heroine& heroine, Input input) {}
  virtual void update(Heroine& heroine) {}
};
```

为每个状态写个类

对于每个状态，我们定义一个类实现接口。它的方法定义了英雄在状态的行为。 换言之，从之前的switch中取出每个case，将它们移动到状态类中。举个例子：

```
class DuckingState : public HeroineState
{
public:
  DuckingState()
  : chargeTime_(0)
  {}

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN)
    {
      // 改回站立状态……
      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类中。 这很好——那部分数据只在这个状态有用，现在我们的对象模型显式反映了这一点。

状态委托

接下来，向Heroine添加指向当前状态的指针，放弃庞大的switch，转向状态委托：

```
class Heroine
{
public:
  virtual void handleInput(Input input)
  {
    state_->handleInput(*this, input);
  }

  virtual void update()
  {
    state_->update(*this);
  }
```

```
  // 其他方法……
private:
  HeroineState* state_;
};
```

为了“改变状态”，我们只需要将state_声明指向不同的HeroineState对象。 这就是状态模式的全部了。

> 这看上去有些像策略[GoF]模式和类型对象[→]模式。 在三者中，你都有一个主对象委托给下属。区别在于**意图**。
>
> - 在策略模式中，目标是解耦主类和它的部分行为。
> - 在类型对象中，目标是通过**共享**一个对相同类型对象的引用，让一**系列**对象行为相近。
> - 在状态模式中，目标是让主对象通过**改变**委托的对象，来**改变**它的行为。

状态对象在哪里？

我这里掩掩藏了一些细节。为了改变状态，我们需要声明state_指向新的状态， 但那个新状态又是从哪里来呢？ 在enum实现中，这都不用过脑子——enum实际上就像数字一样。 但是现在状态是类了，意味着我们需要指向实例。通常这有两种方案：

静态状态

如果状态对象没有其他数据字段， 那么它存储的唯一数据就是指向虚方法表的指针，用来调用它的方法。 在这种情况下，没理由产生多个实例。毕竟每个实例都完全一样。

> 如果你的状态没有字段，只有**一个**虚方法，你可以再简化这个模式。 将每个状态**类**替换成状态**函数**——只是一个普通的顶层函数。 然后，主类中的state_字段变成一个简单的函数指针。

在那种情况下，你可以用一个静态实例。 哪怕你有一堆FSM同时在同一状态上运行，它们也能指向同一实例，因为状态没有与状态机相关的部分。

> 这是享元[GoF]模式。

在哪里放置静态实例取决于你。找一个合理的地方。 没什么特殊的理由，在这里我将它放在状态基类中。

```
class HeroineState
{
public:
  static StandingState standing;
  static DuckingState ducking;
  static JumpingState jumping;
  static DivingState diving;

  // 其他代码……
};
```

每个静态字段都是游戏状态类的一个实例。为了让英雄跳跃，站立状态会这样做：

```
if (input == PRESS_B)
{
  heroine.state_ = &HeroineState::jumping;
```

```
  heroine.setGraphics(IMAGE_JUMP);
}
```

实例化状态

有时没那么容易。静态状态对俯卧状态不起作用。 它有一个`chargeTime_`字段，与正在俯卧的英雄特定相关。 在游戏中，如果只有一个英雄，那也行，但是如果要添加双人合作，同时在屏幕上有两个英雄，就有麻烦了。

在那种情况下，转换时需要创建状态对象。 这需要每个FSM拥有自己的状态实例。如果我们分配新状态， 那意味着我们需要释放当前的状态。 在这里要小心，由于触发变化的代码是当前状态中的方法，需要删除`this`，因此需要小心从事。

相反，我们允许`HeroineState`中的`handleInput()`返回一个新状态。 如果它那么做了，`Heroine`会删除旧的，然后换成新的，就像这样：

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL)
  {
    delete state_;
    state_ = state;
  }
}
```

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

```
HeroineState* StandingState::handleInput(Heroine& heroine,
                                         Input input)
{
  if (input == PRESS_DOWN)
  {
    // 其他代码……
    return new DuckingState();
  }

  // 保持这个状态
  return NULL;
}
```

如果可以，我倾向于使用静态状态，因为它们不会在状态转换时消耗太多的内存和CPU。但是，对于更多状态的事物，需要耗费一些精力来实现。

> 当你为状态动态分配内存时，你也许会担心碎片。 对象池[3]模式可以帮上忙。

入口行为和出口行为

状态模式的目标是将状态的行为和数据封装到单一类中。 我们完成了一部分，但是还有一些未了之事。

当英雄改变状态时，我们也改变她的贴图。 现在，那部分代码在她转换前的状态中。 当她从俯卧转为站立，俯卧状态修改了她的贴图：

```
HeroineState* DuckingState::handleInput(Heroine& heroine,
                                        Input input)
```

```
{
  if (input == RELEASE_DOWN)
  {
    heroine.setGraphics(IMAGE_STAND);
    return new StandingState();
  }

  // 其他代码……
}
```

我们想做的是，每个状态控制自己的贴图。这可以通过给状态一个入口行为来实现：

```
class StandingState : public HeroineState
{
public:
  virtual void enter(Heroine& heroine)
  {
    heroine.setGraphics(IMAGE_STAND);
  }

  // 其他代码……
};
```

在Heroine中，我们将处理状态改变的代码移动到新状态上调用：

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL)
  {
    delete state_;
    state_ = state;

    // 调用新状态的入口行为
    state_->enter(*this);
  }
}
```

这让我们将俯卧代码简化为：

```
HeroineState* DuckingState::handleInput(Heroine& heroine,
                                        Input input)
{
  if (input == RELEASE_DOWN)
  {
    return new StandingState();
  }

  // 其他代码……
}
```

它做的所有事情就是转换到站立状态，站立状态控制贴图。 现在我们的状态真正地封装了。 关于入口行为的好事就是，当你进入状态时，不必关心你是从哪个状态转换来的。

大多数真正的状态图都有转为同一状态的多个转移。 举个例子，英雄在跳跃或跳斩后进入站立状态。 这意味着我们在转换发生的最后重复相同的代码。 入口行为很好地解决了这一点。

我们能，当然，扩展并支持出口行为。 这是在我们离开现有状态，转换到新状态之前调用的方法。

有什么收获？

我花了这么长时间向您推销FSMs，现在我们来捋一捋。 我到现在讲的都是真的，FSM能很好地解决一些问题。但它们最大的优点也是它们最大的缺点。

状态机通过使用有约束的结构来理清杂乱的代码。 你只需一个固定状态的集合，单一的当前状态，和一些硬编码的转换。

> 一个有限状态机甚至不是**图灵完全的**。 自动理论用一系列抽象模型描述计算，每种都比之前的复杂。 **图灵机**是其中最具有表现力的模型之一。
>
> “图灵完全”意味着一个系统（通常是编程语言）足以在内部实现一个图灵机， 也就意味着，在某种程度上，所有的图灵完全具有同样的表现力。 FSMs不够灵活，并不在其中。

如果你需要为更复杂的东西使用状态机，比如游戏AI，你会撞到这个模型的限制上。 感谢上天，我们的前辈找到了一些方法来避免这些限制。我会在这一章的最后简单地浏览一下它们。

并发状态机

我们决定赋予英雄拿枪的能力。 当她拿着枪的时候，她还是能做她之前的任何事情：跑动，跳跃，跳斩，等等。 但是她在做这些的同时也要能开火。

如果我们执着于FSM，我们需要翻倍现有状态。 对于每个现有状态，我们需要另一个她持枪状态：站立，持枪站立，跳跃，持枪跳跃， 你知道我的意思了吧。

多加几种武器，状态就会指数爆炸。 不但增加了大量的状态，也增加了大量的冗余： 持枪和不持枪的状态是完全一样的，只是多了一点负责射击的代码。

问题在于我们将两种状态绑定到了一个状态机上——她做的和她携带的。 为了处理所有可能的组合，我们需要为每一对组合写一个状态。 修复方法很明显：使用两个单独的状态机。

> 如果她在做什么有**n**个状态，而她携带了什么有**m**个状态，要塞到一个状态机中，我们需要**n × m**个状态。使用两个状态机，就只有**n + m**个。

我们保留之前记录她在做什么的状态机，不用管它。 然后定义她携带了什么的单独状态机。 `Heroine`将会有两个“状态”引用，每个对应一个状态机，就像这样：

```
class Heroine
{
  // 其他代码……

private:
  HeroineState* state_;
  HeroineState* equipment_;
};
```

> 为了便于说明，她的装备也使用了状态模式。 在实践中，由于装备只有两个状态，一个布尔标识就够了。

当英雄把输入委托给了状态，两个状态都需要委托：

```
void Heroine::handleInput(Input input)
{
  state_->handleInput(*this, input);
  equipment_->handleInput(*this, input);
}
```

> 功能更完备的系统也许能让状态机**销毁**输入，这样其他状态机就不会收到了。 这能阻止两个状态机响应同一输入。

每个状态机之后都能响应输入，发生行为，独立于其它机器改变状态。 当两个状态集合几乎没有联系的时候，它工作得不错。

在实践中，你会发现状态有时需要交互。 举个例子，也许她在跳跃时不能开火，或者她在持枪时不能跳斩攻击。 为了完成这个，你也许会在状态的代码中做一些粗糙的`if`测试其他状态来协同， 这不是最优雅的解决方案，但这可以搞定工作。

分层状态机

再充实一下英雄的行为，她可能会有更多相似的状态。 举个例子，她也许有站立、行走、奔跑和滑铲状态。在这些状态中，按B跳，按下蹲。

如果使用简单的状态机实现，我们在每个状态中的都重复了代码。 如果我们能够实现一次，在多个状态间重用就好了。

如果这是面向对象的代码而不是状态机的，在状态间分享代码的方式是通过继承。 我们可以为“在地面上”定义一个类处理跳跃和速降。 站立、行走、奔跑和滑铲都从它继承，然后增加各自的附加行为。

> 它的影响有好有坏。 继承是一种有力的代码重用工具，但也在两块代码间建立了非常强的耦合。 这是重锤，所以请小心使用。

你会发现，这是个被称为分层状态机的通用结构。 状态可以有父状态（这让它变为子状态）。 当一个事件进来，如果子状态没有处理，它就会交给链上的父状态。 换言之，它像重载的继承方法那样运作。

事实上，如果我们使用状态模式实现FSM，我们可以使用继承来实现层次。 定义一个基类作为父状态：

```
class OnGroundState : public HeroineState
{
public:
  virtual void handleInput(Heroine& heroine, Input input)
  {
    if (input == PRESS_B)
    {
      // 跳跃……
    }
    else if (input == PRESS_DOWN)
    {
      // 俯卧……
    }
  }
};
```

每个子状态继承它：

```
class DuckingState : public OnGroundState
{
public:
  virtual void handleInput(Heroine& heroine, Input input)
  {
    if (input == RELEASE_DOWN)
    {
      // 站起……
    }
    else
    {
      // 没有处理输入，返回上一层
      OnGroundState::handleInput(heroine, input);
    }
  }
};
```

这当然不是唯一的实现层次的方法。 如果你没有使用GoF的状态模式，这可能不会有用。 相反，你可以显式的使用状态栈而不是单一状态来表示当前状态的父状态链。

栈顶的状态是当前状态，在他下面是它的直接父状态， 然后是那个父状态的父状态，以此类推。 当你需要状态的特定行为，你从栈的顶端开始， 然后向下寻找，直到某一个状态处理了它。（如果到底也没找到，就无视它。）

下推自动机

还有一种有限状态机的扩展也用了状态栈。 容易混淆的是，这里的栈表示的是完全不同的事物，被用于解决不同的问题。

要解决的问题是有限状态机没有任何历史的概念。 你记得正在什么状态中，但是不记得曾在什么状态。 没有简单的办法重回上一状态。

举个例子：早先，我们让无畏英雄武装到了牙齿。 当她开火时，我们需要新状态播放开火动画，发射子弹，产生视觉效果。 所以我们拼凑了一个`FiringState`，不管现在是什么状态，都能在按下开火按钮时跳转为这个状态。

> 这个行为在多个状态间重复，也许是用层次状态机重用代码的好地方。

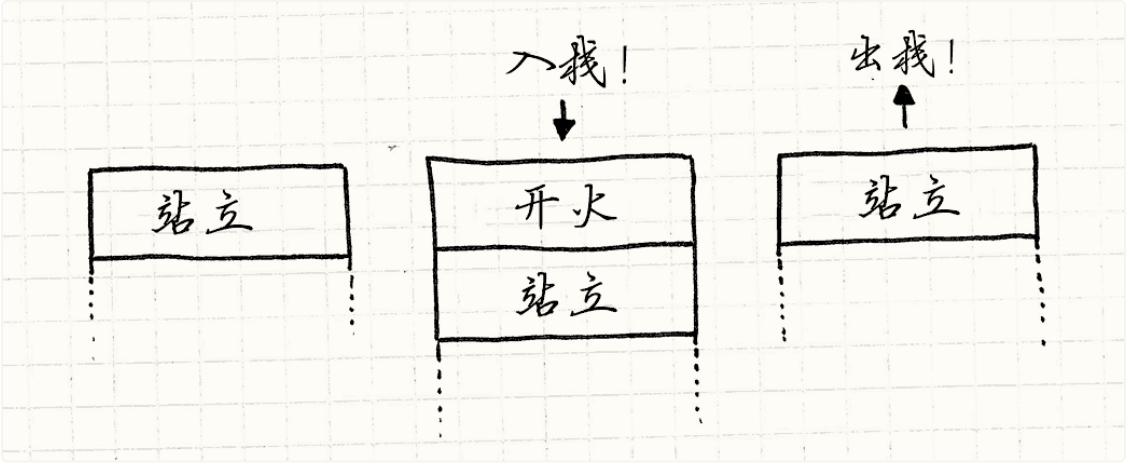
问题在于她射击后转换到的状态。 她可以在站立、奔跑、跳跃、跳斩时射击。 当射击结束，应该转换为她之前的状态。

如果我们固执于纯粹的FSM，我们就已经忘了她之前所处的状态。 为了追踪之前的状态，我们定义了很多几乎完全一样的类——站立开火，跑步开火，跳跃开火，诸如此类—— 每个都有硬编码的转换，用来回到之前的状态。

我们真正想要的是，它会存储开火前所处的状态，之后能回想起来。 自动理论又一次能帮上忙了，相关的数据结构被称为下推自动机。

有限状态机有一个指向状态的指针，下推自动机有一栈指针。 在FSM中，新状态代替了之前的那个状态。 下推自动机不仅能完成那个，还能给你两个额外操作：

1. 你可以将新状态压入栈中。“当前的”状态总是在栈顶，所以你能转到新状态。 但它让之前的状态待在栈中而不是销毁它。
2. 你可以弹出最上面的状态。这个状态会被销毁，它下面的状态成为新状态。



这正是我们开火时需要的。我们创建单一的开火状态。 当开火按钮在其他状态按下时，我们压入开火状态。 当开火动画结束，我们弹出开火状态，然后下推自动机自动转回之前的状态。

所以它们有多有用呢？

即使状态机有这些常见的扩展，它们还是很受限制。 这让今日游戏AI移向了更加激动人心的领域，比如行为树和规划系统 。 如果你关注复杂AI，这一整章只是为了勾起你的食欲。 你需要阅读其他书来满足你的欲望。

这不意味着有限状态机，下推自动机，和其他简单的系统没有用。 它们是特定问题的好工具。有限状态机在以下情况有用：

- 你有个实体，它的行为基于一些内在状态。
- 状态可以被严格地分割为相对较少的不相干项目。
- 实体响应一系列输入或事件。

在游戏中，状态机因在AI中使用而闻名，但是它也常用于其他领域， 比如处理玩家输入，导航菜单界面，分析文字，网络协议以及其他异步行为。

← 上一章 ≡ 首页 下一章 →

© 2009-2015 Robert Nystrom