Go语言 有限状态机FSM

原创

Rust编程学院 于 2014-04-20 09:54:04 发布 5100 收藏 2
文章标签： go语言 go golang

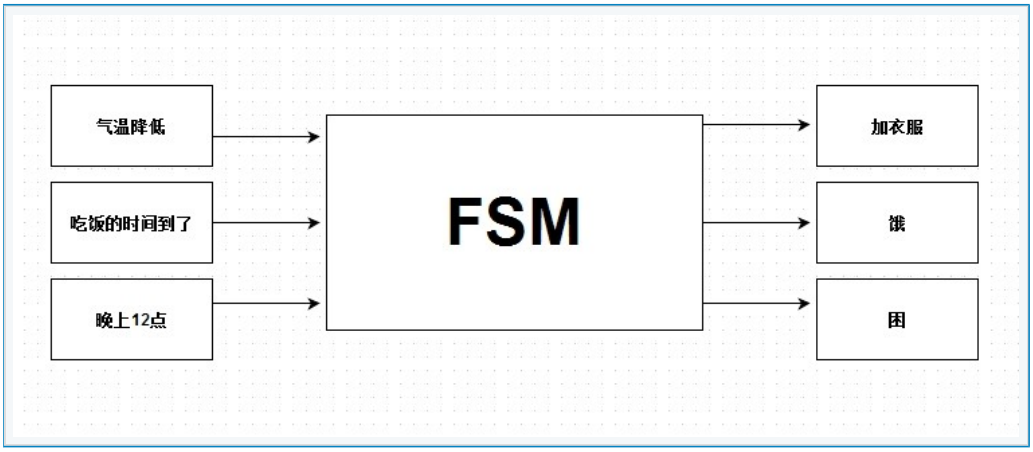
有限状态机又简称FSM(Finite-State Machine的首字母缩写)。这个在离散数学里学过了，它是计算机领域中被广泛使用的数学概念。是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。编译原理学得好的童鞋应该对FSM不陌生，因为编译器就用了FMS来做词法扫描时的状态转移。

FSM的概念在网上一搜可以搜一大堆出来，但估计您也看不大明白。本文将以不一样的方式来讲述FSM的概念以及实现。

现实生活中，状态是随处可见的，并且通过不同的状态来做不同的事。比如冷了加衣服；饿了吃饭；困了睡觉等。这里的冷了、饿了、困了是三种不同的状态，并且根据这三个状态的转变驱动了不同行为的产生(加衣服、吃饭和睡觉)。

FSM是什么

所谓有限状态机，就是由有限个状态组成的机器。再看上面举到的例子：人就是一部机器，能感知三种状态(冷、饿、困)。由于气温降低所以人会觉得冷；由于到了吃饭的时间所以觉得饿；由于晚上12点所以觉得困。状态的产生以及改变都是由某种条件的成立而出现的。不考虑FSM的内部结构时，它就像是一个黑箱子，如下图：



左边是输入一系列条件，FSM通过判定，然后输出结果。

FSM的处理流程

上图FSM屏蔽了判定的过程，事实上FSM是由有限多个状态组成的，每个状态相当于FSM的一个部件。比如要判断一个整数是否偶数，其实只需要判断这个整数的最低位是否为0就行了，代码如下：

$GOPATH/src/fsm_test

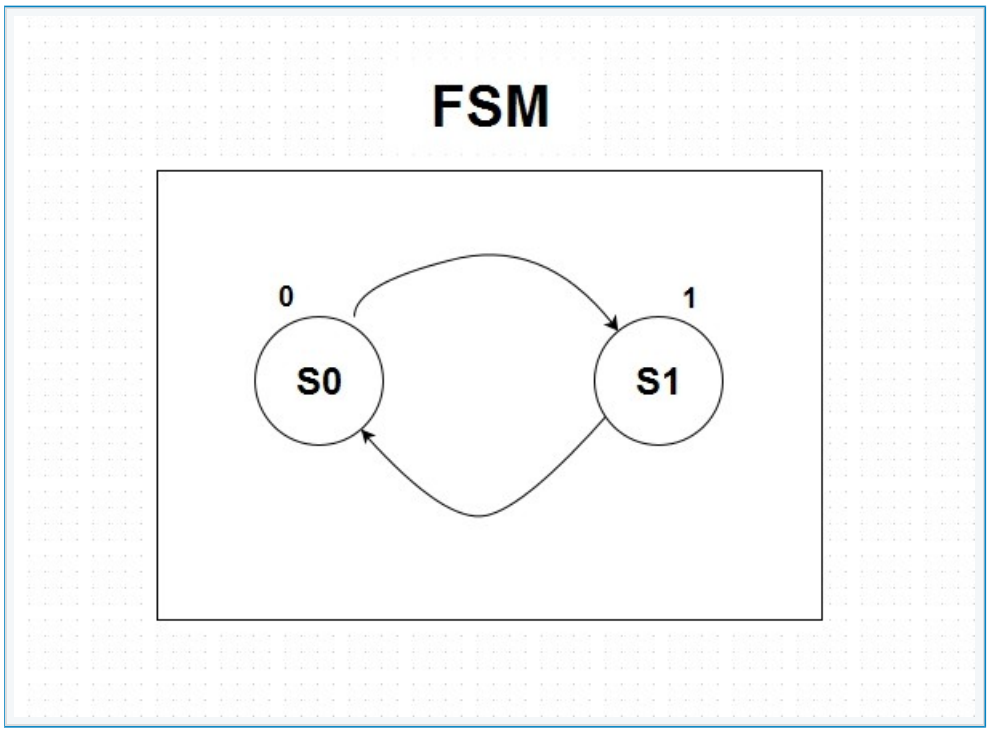
----main.go

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 func IsEven(num int) bool {
08     if num&0x1 == 0x0 {
09         return true
10     }
11
12     return false
13 }
14
15 func main() {
```

```
    fmt.Printf("%d is even? %t\n", 4, IsEven(4))
    fmt.Printf("%d is even? %t\n", 5, IsEven(5))
}
```

```
$ cd $GOPATH/src/fsm_test
$ go build
$ ./fsm_test
4 is even? true
5 is even? false
```

对数字5来说，它的二进制表示为0101。二进制只能为0或1，所以二进制的字符集合为：{0, 1}，对应到FSM来说，就是有2种状态，分别为S0和S1。如果用FSM来处理，它总是从左边读取(当然也可以把FSM反过来)，也就是从0101最左边那位开始输入：首先输入左边第一位0，停留在S0状态，然后输入第二位1，转到S1状态，再输入第三位0，则又回到S0状态，最后输入是后一位1则又回到S1状态。如下图所示：



上图忽略了一个很重要的细节，就是0和1是怎么输入的。状态S0和状态S1是FSM里的2个小部件，它们分别关联了0和1(也可以说是特定的输入语句)，所以只能通过FSM来输入。当FSM接收到0时，它就交给S0去处理，这时S0就变成当前状态，然后对S0输入1，S0则将它交给S1去处理，这时S1就变成当前状态。如此这般，FSM持有有限多个状态，它可以接收输入并执行状态转移(比如将最初的0交给S0去处理)。状态S0和状态S1也是如此。

但是为什么最开始FSM接收输入的0后会交给S0去处理呢？这是因为FSM的默认状态是S0。就像是有一台电视机，它总是有默认的频道的，您一打开电视机就可以看到影像，即使是满屏的雪花点。而且可以在按下电视机的开关前预先调整频道，之后也可以调整频道。

如何用程序建模

FSM持有有限多个状态集合，有当前状态、默认状态、接收的外部数据等。并且FSM有一系列的行为：启动FSM、退出FSM以及状态转移等。State(状态)也会有一系列的行为：进入状态，转移状态等。并且State还有Action行为，比如电视机当前频道正在播放西游记，切换频道后就变成了播放封神榜，原理上是一样的。代码定义如下：

```
package main

// 接口
type IFSMState interface {
    Enter()
    Exit()
    CheckTransition()
}

// State父struct
type FSMState struct{}

// 进入状态
func (this *FSMState) Enter() {
    //
}

// 退出状态
func (this *FSMState) Exit() {
    //
}
```

```
22
23  // 状态转移检测
24  func (this *FSMState) CheckTransition() {
25      //
26  }
27
28  type FSM struct {
29      // 持有状态集合
30      states map[string]IFSMState
31      // 当前状态
32      current_state IFSMState
33      // 默认状态
34      default_state IFSMState
35      // 外部输入数据
36      input_data interface{}
37  }
38
39  // 初始化FSM
40  func (this *FSM) Init() {
41      //
42  }
43
44  // 添加状态到FSM
45  func (this *FSM) AddState(key string, state IFSMState) {
46      //
47  }
48
49  // 设置默认的State
50  func (this *FSM) SetDefaultState(state IFSMState) {
51      //
52  }
53
54  // 转移状态
55  func (this *FSM) TransitionState() {
56      //
57  }
58
59  // 设置输入数据
60  func (this *FSM) SetInputData(inputData interface{}) {
61      //
62  }
63
64  // 重置
65  func (this *FSM) Reset() {
66      //
67  }
68
69  func main() {
70  }
```

以上代码只是初略的定义。我们知道FSM不是直接去选择某种状态，而是根据输入条件来选择的。所以可以定义一张输入语句和状态的映射表，本文仅仅简单实现。

NPC例子

游戏中一个玩家可以携带宠物，那么这个 宠物(NPC)就可以看作是FSM。比如这个宠物在每天8点钟开始工作(挣金币)，中午12点钟开始打坐练功。8点钟和12点钟就是对这个FSM的输入语句，对应的状态则是开始工作和开始打坐练功。代码实现如下：

```
001 package main
002
003 import (
004     "fmt"
005 )
006
007 // 接口
008 type IFSMState interface {
009     Enter()
010     Exit()
011     CheckTransition(hour int) bool
012     Hour() int
013 }
014
015 // State父struct
016 type FSMState struct{}
017
018 // 进入状态
019 func (this *FSMState) Enter() {
020     //
021 }
022
023 // 退出状态
024 func (this *FSMState) Exit() {
025     //
026 }
027
028 // 状态转移检测
029 func (this *FSMState) CheckTransition(hour int) {
030     //
```

```
}

// 打坐
type ZazenState struct {
    hour int
    FSMState
}

func NewZazenState() *ZazenState {
    return &ZazenState{hour: 8}
}

func (this *ZazenState) Enter() {
    fmt.Println("ZazenState: 开始打坐")
}

func (this *ZazenState) Exit() {
    fmt.Println("ZazenState: 退出打坐")
}

func (this *ZazenState) Hour() int {
    return this.hour
}

// 状态转移检测
func (this *ZazenState) CheckTransition(hour int) bool {
    if hour == this.hour {
        return true
    }

    return false
}

// 工作
type WorkerState struct {
    hour int
    FSMState
}

func NewWorkerState() *WorkerState {
    return &WorkerState{hour: 12}
}

func (this *WorkerState) Enter() {
    fmt.Println("WorkerState: 开始工作")
}

func (this *WorkerState) Exit() {
    fmt.Println("WorkerState: 退出工作")
}

func (this *WorkerState) Hour() int {
    return this.hour
}

// 状态转移检测
func (this *WorkerState) CheckTransition(hour int) bool {
    if hour == this.hour {
        return true
    }

    return false
}

type FSM struct {
    // 持有状态集合
    states map[string]IFSMState
    // 当前状态
    current_state IFSMState
    // 默认状态
    default_state IFSMState
    // 外部输入数据
    input_data int
    // 是否初始化
    inited     bool
}

// 初始化FSM
func (this *FSM) Init() {
    this.Reset()
}

// 添加状态到FSM
func (this *FSM) AddState(key string, state IFSMState) {
    if this.states == nil {
        this.states = make(map[string]IFSMState, 2)
    }
    this.states[key] = state
}

// 设置默认的State
func (this *FSM) SetDefaultState(state IFSMState) {
    this.default_state = state
```

```
}

// 转移状态
func (this *FSM) TransitionState() {
    nextState := this.default_state
    input_data := this.input_data
    if this.inited {
        for _, v := range this.states {
            if input_data == v.Hour() {
                nextState = v
                break
            }
        }
    }

    if ok := nextState.CheckTransition(this.input_data); ok {
        if this.current_state != nil {
            // 退出前一个状态
            this.current_state.Exit()
        }
        this.current_state = nextState
        this.inited = true
        nextState.Enter()
    }
}

// 设置输入数据
func (this *FSM) SetInputData(inputData int) {
    this.input_data = inputData
    this.TransitionState()
}

// 重置
func (this *FSM) Reset() {
    this.inited = false
}

func main() {
    zazenState := NewZazenState()
    workerState := NewWorkerState()
    fsm := new(FSM)
    fsm.AddState("ZazenState", zazenState)
    fsm.AddState("WorkerState", workerState)
    fsm.SetDefaultState(zazenState)
    fsm.Init()
    fsm.SetInputData(8)
    fsm.SetInputData(12)
    fsm.SetInputData(12)
    fsm.SetInputData(8)
    fsm.SetInputData(12)
}
```

```
$ cd $GOPATH/src/fsm_test
$ go build
$ ./fsm_test
ZazenState: 开始打坐
ZazenState: 退出打坐
WorkerState: 开始工作
WorkerState: 退出工作
WorkerState: 开始工作
WorkerState: 退出工作
ZazenState: 开始打坐
ZazenState: 退出打坐
WorkerState: 开始工作
```

关于对FSM的封装

FSM主要是处理感知外部数据而产生的状态转变，所以别打算去封装它。不同的条件，不同的状态以及不同的处理方式令FSM基本上不太可能去封装，至也多只是做一些语法上的包装罢了。

结束语

真实的场景中，这个NPC所做的工作可能会非常多。比如自动判断周边的环境，发现怪物就去打怪，没血了就自动补血，然后实在打不过就逃跑等等。上例中的SetInputData()就是用于模拟周边环境的数据对NPC的影响，更复杂的情况还在于NPC有时候执行的动作是不能被打断的(上例中的Exit()方法)，它只有在完成某个周期的行为才能被终止。这个很容易理解。比如NPC发送网络数据包的时候就不能轻易的被中断，那这个时候其实是可以实现同步原语，状态之间互相wait。

FSM被广泛用于游戏设计和其它各方面，的确是个比较重要的数学模型。