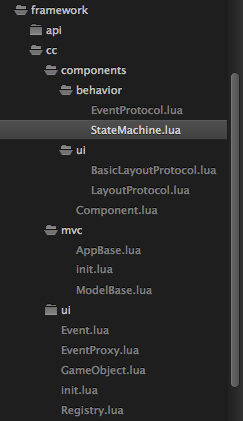
**状态机源码位置**

[](http://www.tairan.com/usr/uploads/2014/02/content.png)  
  
注：写这篇文章的时候，笔者所用的是quick-cocos2d-x 2.2.1rc版本。

**quick状态机**

状态机的设计，目的就是为了避免大量状态的判断带来的复杂性，消除庞大的条件分支语句，因为大量的分支判断会使得程序难以修改和扩展。但quick状态机的设计又不同设计模式的状态模式，TA没有将各个状态单独划分成单独的状态类，相反根据js、lua语言的特点，特别设计了写法，使用起来也比较方便。  
  
quick框架中的状态机，是根据[javascript-state-machine](https://github.com/jakesgordon/javascript-state-machine)重新设计改写而成，同时

*sample/statemachine*

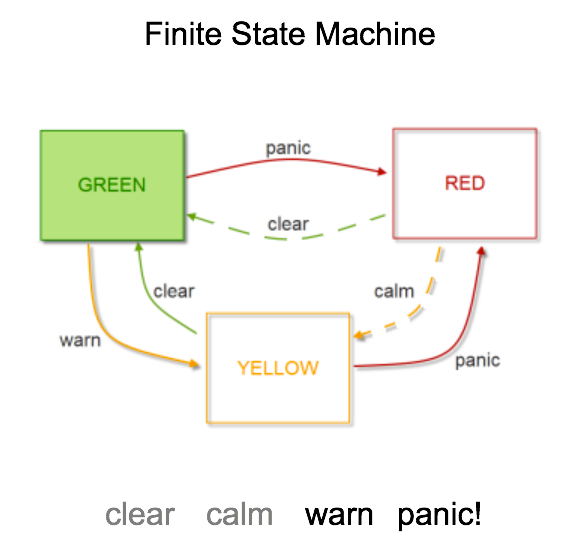
范例也是根据js版demo改写而来。该js库现在是

*2.2.0*

版本。基于js版的README.md，结合廖大的lua版重构，我针对状态机的使用做了点说明，如果有不对的地方，感谢指出:)。  
  
推荐大家在理解的时候结合

*sample/statemachine*

范例进行理解，注意player设置成竖屏模式，demo里面的按钮在横屏模式下看不见。

**sample图示**  
[](http://www.tairan.com/usr/uploads/2014/02/statemachine.png)

**2. 用法**

**创建一个状态机**

local fsm = StateMachine.new()

-- （注：和demo不同的是，demo采用组件形式完成的初始化）

fsm:setupState({

initial = "green",

events = {

{name = "warn", from = "green", to = "yellow"},

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "yellow"},

{name = "clear", from = "yellow", to = "green" },

}

})

之后我们就可以通过

* fsm:doEvent("start")-从"none"状态转换到"green"状态
* fsm:doEvent("warn")-从"green"状态转换到"yellow"状态
* fsm:doEvent("panic")-从"green"状态转换到"red"状态
* fsm:doEvent("calm")-从"red"状态转换到"yellow"状态
* fsm:doEvent("clear")-从"yellow"状态转换到"green"状态

同时，

* fsm:isReady()-返回状态机是否就绪
* fsm:getState()-返回当前状态
* fsm:isState(state)-判断当前状态是否是参数state状态
* fsm:canDoEvent(eventName)-当前状态如果能完成eventName对应的event的状态转换，则返回true
* fsm:cannotDoEvent(eventName)-当前状态如果不能完成eventName对应的event的状态转换，则返回true
* fsm:isFinishedState()-当前状态如果是最终状态，则返回true
* fsm:doEventForce(name, ...)-强制对当前状态进行转换

**单一事件的多重from和to状态**

如果一个事件允许我们从多个状态(

*from*

)转换到同一个状态(

*to*

), 我们可以通过用一个集合来构建

*from*

状态。如下面的 "

*rest"*

事件。但是，如果一个事件允许我们从多个状态(

*from*

)转换到对应的不同的状态(

*to*

)，那么我们必须将该事件分开写，如下面的"

*eat*

"事件。

local fsm = StateMachine.new()

fsm:setupState({

initial = "hungry",

events = {

{name = "eat", from = "hungry", to = "satisfied"},

{name = "eat", from = "satisfied", to = "full"},

{name = "eat", from = "full", to = "sick" },

{name = "rest", from = {"hungry", "satisfied", "full", "sick"}, to = "hungry"},

}

})

在设置了事件`

*events*

`之后，我们可以通过下面两个方法来完成状态转换。

* fsm:doEvent("eat")
* fsm:doEvent("rest")

`

*rest*

` 事件的目的状态永远是 `

*hungry*

` 状态，而 `

*eat*

` 事件的目的状态取决于当前所处的状态。

注意1：如果事件可以从任何当前状态开始进行转换，那么我们可以用一个通配符`\*`来替代`from`状态。如`rest`事件,我们可以写成`{name = "rest", from = "\*", to = "hungry"}`。  
  
注意2：上面例子的`rest`事件可以拆分写成4个，如下：

{name = "rest", from = "hungry", to = "hungry"},

{name = "rest", from = "satisfied", to = "hungry"},

{name = "rest", from = "full", to = "hungry"},

{name = "rest", from = "sick", to = "hungry"}

**回调**

quick的状态机支持4种***特定事件***类型的回调:

* onbeforeEVNET- 在特定事件EVENT开始前被激活
* onleaveSTATE - 在离开旧状态STATE时被激活
* onenterSTATE - 在进入新状态STATE时被激活
* onafterEVENT - 在特定事件EVENT结束后被激活

注解：编码时候，EVENT/STATE应该被替换为特定的名字

为了便利起见，

* onenterSTATE可以简写为 onSTATE
* onafterEVENT可以简写为 onEVENT

所以假如要使用简写的话，为了避免`onSTATE`和`onEVENT`的STATE/EVENT被替换成具体的名字后名字相同引起问题，`to`状态和`name`名字尽量不要相同。比如

-- 角色开火

{name = "fire", from = "idle", to = "fire"}

--假如使用简写

--onSTATE --- onfire

--onEVENT --- onfire，回调会引起歧义。

--如果不使用简写

--则onenterSTATE --- onenterfire

--onafterEVENT --- onafterfire

另外，我们可以使用5种通用型的回调来捕获所有事件和状态的变化:

* `onbeforeevent`- 在任何事件开始前被激活
* `onleavestate` - 在离开任何状态时被激活
* `onenterstate` - 在进入任何状态时被激活
* `onafterevent` - 在任何事件结束后被激活
* `onchangestate` - 当状态发生改变的时候被激活

>注解：这里是任何事件、状态, 小写的event、state不能用具体的事件、状态名字替换。

**回调参数**

所有的回调都以`

*event*

`为参数，该event为表结构，包含了

* name 事件名字
* from 事件表示的起始状态
* to 事件表示的目的状态
* args 额外的参数，用来传递用户自定义的一些变量值

local fsm = StateMachine.new()

fsm = fsm:setupState({

initial = "green",

events = {

{name = "warn", from = "green", to = "yellow"},

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "yellow"},

{name = "clear", from = "yellow", to = "green" },

},

callbacks = {

onbeforestart = function(event) print("[FSM] STARTING UP") end,

onstart = function(event) print("[FSM] READY") end,

onbeforewarn = function(event) print("[FSM] START EVENT: warn!") end,

onbeforepanic = function(event) print("[FSM] START EVENT: panic!") end,

onbeforecalm = function(event) print("[FSM] START EVENT: calm!") end,

onbeforeclear = function(event) print("[FSM] START EVENT: clear!") end,

onwarn = function(event) print("[FSM] FINISH EVENT: warn!") end,

})

fsm:doEvent("warn", "some msg")

如上例子，`

*fsm:doEvent("warn", "some msg")*

` 中的 `

*some msg*

` 作为额外的参数字段 `

*args*

` 结合 `

*name*

`  `

*from*

`  `

*to*

`被添加到`

*event*

`，此时

event = {

name = "warn",

from = "green",

to = "yellow",

args = "some msg"

}

而 `

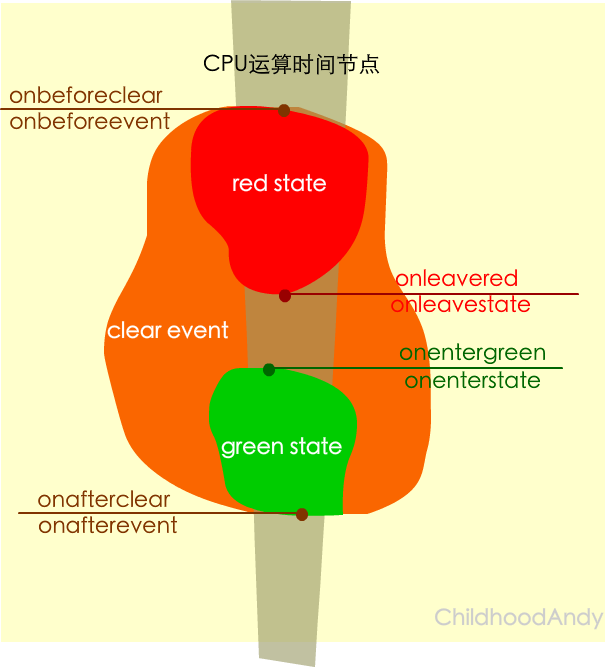
*event*

` 表正是回调函数的参数。

**回调顺序**

用

*{name = "clear", from = "red", to = "green"}*

举例，我画个示意图来说明[](http://www.tairan.com/usr/uploads/2014/02/%E5%9B%9E%E8%B0%83.png)

注意：之前的`

*onbeforeEVENT*

`,这里`EVENT`就被具体替换为`clear`，于是是 `

*onbeforeclear*

` ，而 `

*onbeforeevent*

` 类似的通用型则不用替换。

* onbeforeclear - clear事件执行前的回调
* onbeforeevent - 任何事件执行前的回调
* onleavered - 离开红色状态时的回调
* onleavestate - 离开任何状态时的回调
* onentergreen - 进入绿色状态时的回调
* onenterstate - 进入任何状态时的回调
* onafterclear - clear事件完成之后的回调
* onafterevent - 任何事件完成之后的回调

**3种影响事件响应的方式**

1. 在 `onbeforeEVENT` 方法中返回false来取消事件
2. 在 `onleaveSTATE` 方法中返回false来取消事件
3. 在 `onleaveSTATE` 方法中返回`ASYNC`来执行异步状态转换

**异步状态转换**

有时候，我们需要在状态转换的时候执行一些异步性代码来确保不会进入新状态直到代码执行完毕。  
举个例子来说，假如要从一个`menu`状态转换出来，或许我们想让TA淡出？滑出屏幕之外？总之执行完动画再进入`

*game*

`状态。  
  
我们可以在`onleavestate`或者`onleaveSTATE`方法里返回`StateMachine.ASYNC`,这时状态机会被挂起，直到我们使用了event的`transition()`方法。

...

onleavered = function(event)

self:log("[FSM] LEAVE STATE: red")

self:pending(event, 3)

self:performWithDelay(function()

self:pending(event, 2)

self:performWithDelay(function()

self:pending(event, 1)

self:performWithDelay(function()

self.pendingLabel\_:setString("")

event.transition()

end, 1)

end, 1)

end, 1)

return "async"

end,

...

提示：如果想取消异步事件，可以使用event的`cancel()`方法。

**初始化选项**

2.7.1 状态机的初始化选项一般根据我们游戏需求来决定，quick状态机提供了几个简单的选项。  
在默认情况下，如果你没指定 `initial` 状态，状态机会指定当前状态为 `none` 状态，所以需要定义一个能将 `none` 状态转换出去的事件。

local fsm = StateMachine.new()

fsm = fsm:setupState({

events = {

{name = "startup", from = "none", to = "green" },

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "green"}

}

})

echoInfo(fsm:getState()) -- "none"

fsm:doEvent("start")

echoInfo(fsm:getState()) -- "green"

2.7.2 如果我们特别指定了`initial`状态，那么状态机在初始化的时候会自动创建`startup`事件，并且被执行。

local fsm = StateMachine.new()

fsm = fsm:setupState({

initial = "green",

events = {

-- 当指定initial状态时，这个startup事件会被自动创建，所以可以不用写这一句 {name = "startup", from = "none", to = "green" },

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "green"}

}

})

echoInfo(fsm:getState()) -- "green"

2.7.3 我们也可以这样指定`initial`状态：

local fsm = StateMachine.new()

fsm = fsm:setupState({

initial = {state = "green", event = "init"},

events = {

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "yellow"}

}

})

echoInfo(fsm:getState()) -- "green"

2.7.4 如果我们想延缓初始化状态转换事件的执行，我们可以添加 `defer = true`

local fsm = StateMachine.new()

fsm = fsm:setupState({

initial = {state = "green", event = "init", defer = true},

events = {

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "green"}

}

})

echoInfo(fsm:getState()) -- "none"

fsm:doEvent("init")

echoInfo(fsm:getState()) -- "green"

**异常处理**

在默认情况下，如果我们尝试着执行一个当前状态不允许转换的事件，状态机会抛出异常。如果选择处理这个异常，我们可以定义一个错误事件处理。在quick中，发生异常的时候`

*StateMachine:onError\_(event, error, message)*

`会被调用。

local fsm = StateMachine.new()

fsm:setupState({

initial = "green",

events = {

{name = "warn", from = "green", to = "yellow"},

{name = "panic", from = "green", to = "red" },

{name = "calm", from = "red", to = "green"},

{name = "clear", from = "yellow", to = "green" },

}

})

fsm:doEvent("calm") -- fsm:onError\_会被调用,在当前green状态下不允许执行calm事件