

有限状态机

有限状态机

有限状态机是一种用来进行对象行为建模的工具，其作用主要是描述对象在它的生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。在计算机科学中，有限状态机被广泛用于建模应用行为、硬件电路系统设计、软件工程，编译器、网络协议、和计算与语言的研究。比如下图非常有名的TCP协议状态机。

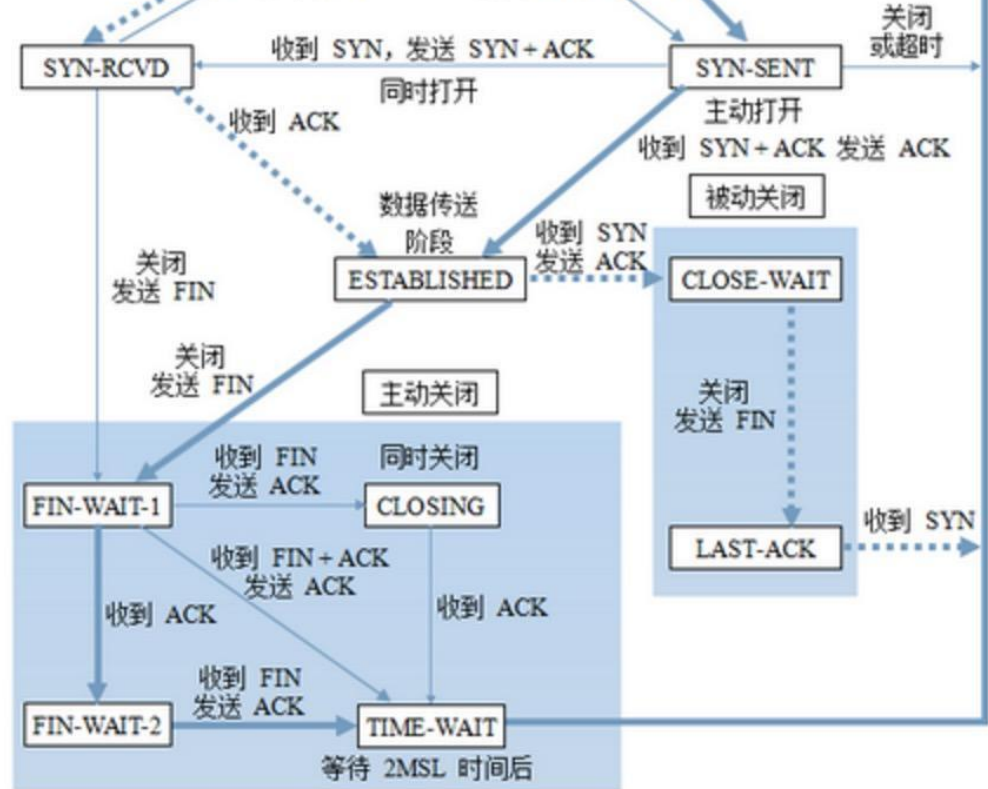


图 TCP 的有限状态机

其实我们在编程时实现相关业务逻辑时经常需要处理各种事件和状态切换，写各种 switch/case 和 if/else，所以我们其实可能一直都在跟有限状态机打交道，只是可能没有意识到。在处理一些业务逻辑比较复杂的需求时，可以先看看是否适合用一个有限状态机来描述，如果可以把业务模型抽象成一个有限状态机，那么代码就会逻辑特别清晰，结构特别规整。

下面我们就来聊聊所谓的状态机，以及它如何在代码中实现。

1、状态机的要素

状态机可归纳为4个要素，即现态、条件、动作、次态。“现态”和“条件”是因，“动作”和“次态”是果。详解如下：

①现态：是指当前所处的状态。

②条件：又称为“事件”。当一个条件被满足，将会触发一个动作，或者执行一次状态的迁移。

③动作：条件满足后执行的动作。动作执行完毕后，可以迁移到新的状态，也可以仍旧保持原状态。动作不是必需的，当条件满足后，也可以不执行任何动作，直接迁移到新状态。

④次态：条件满足后要迁往的新状态。“次态”是相对于“现态”而言的，“次态”一旦被激活，就转变成新的“现态”了。

我们可以用状态表表示了整个过程，如下图所示。

stateB	。 。 。	。 。 。	。 。 。
stateC	。 。 。	不存在	。 。 。

状态表

这里需要注意的两个问题：

1、避免把某个“程序动作”当作是一种“状态”来处理。那么如何区分“动作”和“状态”？“动作”是不稳定的，即使没有条件的触发，“动作”一旦执行完毕就结束了；而“状态”是相对稳定的，如果没有外部条件的触发，一个状态会一直持续下去。

2、状态划分时漏掉一些状态，导致跳转逻辑不完整。

所以维护上述一张状态表就非常必要，而且有意义了。从表中可以直观看出那些状态直接存在跳转路径，那些状态直接不存在。如果不存在，就把对应的单元格置灰。每次写代码之前先把表格填写好，并且对置灰的部分重点review，看看是否有“漏态”，然后才是写代码。QA拿到这张表格之后，写测试用例也是手到擒来。

2、状态机在object-C的代码实现。

我在开发百度地图导航过程页以及百度CarLife的反控手机识别中都用到了有些状态机编程，下面我结合个人的经验，给大家分享一个iOS程序中实现有限状态机的写法。

先回顾一下上面那个状态表，其中状态变迁时执行的动作，可能是由一系列的元动作组成，并且通常都是跟现态和次态强相关的，所以，我把状态表做一个改进，如下所示：

次态 初态	stateA	stateB	stateC
stateA	触发条件：event1 执行动作： FSM_FUN(stateA, stateA)	触发条件：event2 执行动作： FSM_FUN(stateA, stateB)	触发条件：event3 执行动作： FSM_FUN(stateA, stateB)
stateB	。 。 。	。 。 。	。 。 。
stateC	。 。 。	不存在	。 。 。

其中：FSM_FUN(stateA, stateB) 就表示，从状态stateA跳转到stateB时要执行的所有元动作的有序集。

```

// declare the access function and define enum values
#define DECLARE_ENUM(EnumType, ENUM_DEF) \
typedef enum EnumType { \
ENUM_DEF(ENUM_VALUE) \
}EnumType; \
NSString *NSStringFrom##EnumType(EnumType value); \
EnumType EnumType##FromNSString(NSString *string); \

// Define Functions
#define DEFINE_ENUM(EnumType, ENUM_DEF) \
NSString *NSStringFrom##EnumType(EnumType value) \
{ \
switch(value) \
{ \
ENUM_DEF(ENUM_CASE) \
default: return @""; \
} \
} \
EnumType EnumType##FromNSString(NSString *string) \
{ \
ENUM_DEF(ENUM_STRCMP) \
return (EnumType)0; \
}

//状态枚举
#define _EN_FSMState_Type(XX) \
XX(stateA, = 0x01) \
XX(stateB, = 0x02)\
XX(stateC, = 0x03)
DECLARE_ENUM(_EN_FSMState_Type, _EN_FSMState_Type)

//事件枚举
#define _EN_FSMEvent_Type(XX) \
XX(event1, = 0x01) \
XX(event2, = 0x02)\
XX(event3, = 0x03)
DECLARE_ENUM(_EN_FSMEvent_Type, _EN_FSMEvent_Type)

```

宏定义

这里没有啥特殊的，主要是借助宏定义，比较巧妙的实现枚举值到字符串的转换，比如枚举值stateA，能自动生成字符串@"stateA",用于后面的状态函数名拼接。这是一个通用的枚举值自动转字符串的解决方案，参考的链接（<http://stackoverflow.com/a/202511>）

2、Model类定义

iOS开发都会采用MVC架构或者相关变种，但是状态的维护都会实现在Model中。这里定义了一个简单的TestModel，它有一个成员变量state，保存着当前的状态。

```

* 条件事件触发处理
*
* @param event 事件来源
*/
-(void)onHandleEvent:(EN_FSMEvent_Type) event;

/**
 * 状态跳转时执行的元动作1
 */
-(void)testAction1;

/**
 * 状态跳转时执行的元动作2
 */
-(void)testAction2;

@end

```

Model定义

3、实现Model类的一个category,

里面主要定义和实现了状态跳转时要执行的一些动作。

```

@implementation TestModel(FSM)

#pragma mark- stateA 开始的状态跳转
/**
 * stateA跳转到stateA之前要执行的动作序列
 */
- (void)FSM_FUN(stateA, stateA)
{
    //执行元动作1
    [self testAction1];
}

/**
 * stateA跳转到stateB之前要执行的动作序列
 */
- (void)FSM_FUN(stateA, stateB)
{
    //执行元动作1
    [self testAction1];
    //执行元动作2
    [self testAction2];
}

#pragma mark- stateB 开始的状态跳转
- (void)FSM_FUN(stateB, stateA)
{
    //...
}

- (void)FSM_FUN(stateB, stateC)
{
    //...
}

#pragma mark- stateC 开始的状态跳转

```

4、重新Model的setState方法，使得在设置状态时能自动去执行状态跳转时需要执行的动作。

```

/**
 * 根据字符串生成SEL
 * -(void)FSM_stateA_stateB函数会在Model+FSM.m中实现
 *
 */
SEL sel = NSSelectorFromString(FSMFuntionName);

/**
 * 判断是否支持该Selector, 如果支持则执行
 */
if([self respondsToSelector:sel])
{
    NSMethodSignature* signature = [[self class] instanceMethodSignatureForSelector:sel];
    NSInvocation* invocation = [NSInvocation invocationWithMethodSignature: signature];
    [invocation setTarget: self];
    [invocation setSelector:sel];
    [invocation invoke];
}
else
{
    NSLog(@"not respondsToSelector %@",FSMFuntionName);
}

//跳转到新的状态
_state = newstate;
}

```

5、处理事件输入，实现状态跳转逻辑。

这里有两种写法，一种是在状态中判断事件：

```

        /**
        * 这里仅需做状态设置即可，因为我们重写了Set方法，
        * 在setState中会自动运行状态跳转要执行的动作序列FSM_stateA_stateB
        */
        self.state = stateB;
    }
    break;
case event2:
{
    self.state = stateC;
}
    break;
case event3:
{

}
    break;
default:
    break;
}
}
    break;
case stateB:
{

}
    break;
case stateC:
{

}
    break;
default:
    break;
}
}
}

```

状态中判断事件

一种是事件中判断状态：


```

        * 这里只需做状态设置即可，因为我们重写了set方法，
        * 在setState中会自动运行状态跳转要执行的动作序列FSM_stateA_stateB
        */
        self.state = stateB;
    }
    break;
case stateB:
{
    self.state = stateC;
}
    break;
case stateC:
{
}
    break;
default:
    break;
}
}
    break;
case event2:
{
}
    break;
case event3:
{
}
    break;
default:
    break;
}
}
}

```

事件中判断状态

思考与讨论：

状态跳转逻辑的两种写法，实现的功能和效果完全相同，孰优孰劣，欢迎留言探讨。

本人观点：一般业务场景来说，状态的数量是确定的且数目较少，不同状态下需要处理的事件也不一样。而触发的事件数量则比较多，采用上面第二种方式在事件中判断状态也有利于把里面一层的switch/case剥离出来当成单独的函数，做一些代码模块结构的优化，故推荐使用第二种方式，事件中判断状态。

优化后的状态变化函数代码，如下图。

```

        break;
    case event3:
    {
        [self onHandleEvent3];
    }
    break;

    default:
    break;
}
}
-(void)onHandleEvent1
{
    switch (_state) {
    case stateA:
    {
        /**
         * 这里仅需做状态设置即可，因为我们重写了Set方法，
         * 在setState中会自动运行状态跳转要执行的动作序列FSM_stateA_stateB
         */
        self.state = stateB;
    }
    break;
    case stateB:
    {
        self.state = stateC;
    }
    break;
    case stateC:
    {
    }
    break;

    default:
    break;
    }
}
}


```

优化后

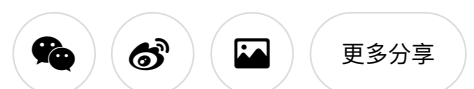
欢迎探讨

本文主要介绍了一下什么是有限状态机，然后通过一个具体的代码示例介绍了一些本人在状态机编程上的经验和理解，欢迎各位进行交流指正。

谢谢大家的宝贵时间。

 编程那些事 (/nb/2306944)

[举报文章](#) © 著作权归作者所有



更多分享

(<http://cwb.assets.jianshu.i>

