

# 用状态机原理进行软件设计

池元武

展讯通信(上海)有限公司，PLD，上海

**摘要：**本文描述状态机基础理论，以及运用状态机原理进行软件设计和实现的方法。

**关键词：**有限状态机 层次状态机 面向对象分析 行为继承

## 参考文献

[1] Miro Samek, Ph.D

《Practical Statecharts in C/C++ Quantum Programming for Embedded Systems》

[2] OpenFans

<http://www.openfans.net/viewArticle.html?id=289>

## 缩略语

名称	描述
FSM	Finite State Machine
HSM	Hierarchical State Machine
OOP	Object Oriented Programming
UML	Unified Modeling Language
LSP	Liskov Substitution Principle
PoC	push to talk over cellular



# 目 录

第 1 章 引言.....	1-1
第 2 章 FSM 概念 .....	2-1
2.1 FSM 定义 .....	2-1
2.2 FSM 要素 .....	2-1
2.2.1 State (状态) .....	2-1
2.2.2 Guard (条件) .....	2-1
2.2.3 Event (事件) .....	2-1
2.2.4 Action (动作) .....	2-1
2.2.5 Transition (迁移) .....	2-2
2.3 FSM 图示 .....	2-2
第 3 章 FSM 设计方法 .....	3-1
3.1 C Parser(注释分析程序) .....	3-1
3.2 Calc(计算器)程序举例.....	3-2
第 4 章 HSM 概念.....	4-1
4.1 programming-by-difference ( 按照差异编程 ) .....	4-1
4.2 HSM 图示.....	4-1
4.3 HSM 分析和 OOP 分析.....	4-2
4.3.1 state inheritance and class inheritance ( 状态层次和类层次 ) .....	4-2
4.3.2 Entry/Exit Actions and Construction/Destruction(进入/退出状态和构造/析构类) .....	4-3
4.3.3 programming-by-difference ( 按照差异编程 ) .....	4-3
4.3.4 abstraction(抽象) .....	4-4
第 5 章 HSM 设计方法.....	5-1
5.1 继续进行 Calc 设计 .....	5-1
5.2 继承关系是否合理 .....	5-3
5.2.1 Transition 迁移执行顺序 .....	5-4
第 6 章 HSM 在实际工程的应用 .....	6-6
6.1 PoC Audio Player.....	6-6
6.2 PoC Call Control.....	6-7
第 7 章 FSM 实现 .....	7-1
7.1 nested switch statement ( 嵌套 switch ) .....	7-1
7.2 state table(状态表).....	7-2

7.3 Function Address As State (用函数指针作为状态) .....	7-3
7.4 QFSM frame (QFSM 框架) .....	7-5
<b>第 8 章 HSM 实现.....</b>	<b>8-1</b>
<b>第 9 章 附录.....</b>	<b>1</b>

# 图目录

图 2-1 Keyboard FSM in UML format 1 .....	2-2
图 2-2 Keyboard FSM in UML format 2 .....	2-3
图 3-1 C comment parser (CParser) FSM.....	3-2
图 3-2 Basic Calc Example.....	3-3
图 3-3 Basic Calc FSM.....	3-4
图 3-4 Basic Calc FSM add “Result” state .....	3-5
图 3-5 Basic Calc FSM add “Result” state and Cancel event.....	3-6
图 3-6 Simple Calc HSM.....	3-7
图 4-2 HSM conception .....	4-2
图 5-1 substate of operandX .....	5-1
图 5-2 Full Calc HSM.....	5-3
图 5-3 State Tree of Calc .....	5-5
图 6-1 HSM of audio Player .....	6-7
图 6-2 HSM of Call Control .....	6-8



# 第1章 引言

20 多年以前，David Harel 创造了状态机理论来描述复杂的交互系统。随后，状态机理论赢得了广泛的接受，并且被引入到许多软件系统中，最突出的是被引入到 UML 中作为其一个组成部分。

不过，状态机理论的发展却很缓慢。在众多原因中，状态机只是做为编程的实现工具而不是设计工具是一个最重要的原因。

本文的重点就在于，怎样利用状态机原理进行程序设计。本文会先给出普通的、一个平面上的 FSM（有限状态机）的概念和实例，并指出其中的一些缺点，然后引出本文的重点 HSM（层次状态机）的概念和设计方法。为了使本文既可以作为设计方法的参考，又可以作为实现方法的参考，本文会给出 FSM 和 HSM 的 C 语言实现。





## 第2章 FSM 概念

### 2.1 FSM 定义

总的来说，有限状态机系统，是指在不同阶段会呈现出不同的运行状态的系统，这些状态是有限的、不重叠的。这样的系统在某一时刻一定会处于其所有状态中的一个状态，此时它接收一部分允许的输入，产生一部分可能的响应，并且迁移到一部分可能的状态。

### 2.2 FSM 要素

#### 2.2.1 State (状态)

State(“状态”)，就是一个系统在其生命周期中某一时刻的运行情况，此时，系统会执行一些动作，或者等待一些外部输入。

#### 2.2.2 Guard (条件)

状态机对外部消息进行响应的时候，除了需要判断当前的状态，还要判断跟这个状态相关的一些条件是否成立。这种判断称为guard(“条件”)。guard通过允许或者禁止某些操作来影响状态机的行为。

#### 2.2.3 Event (事件)

Event(“事件”)，就是在一定的时间和空间上发生的对系统有意义的事情。

#### 2.2.4 Action (动作)

当一个Event被状态机系统分发的时候，状态机用Action(“动作”)来进行响应，比如修改一下变量的值、进行输入输出、产生另外一个Event或者迁移到另外一个状态等等。

### 2.2.5 Transition (迁移)

从一个状态切换到另一个状态被称为Transition ( “ 迁移 ” )。引起状态迁移的事件被称为triggering event ( “ 触发事件 ” ),或者被简称为trigger ( “ 触发 ” )。

## 2.3 FSM 图示

图2-1用UML描述了一个键盘输入模型 ( 这个是个假想的模型, 只是为了用来说明FSM图的要素 )。

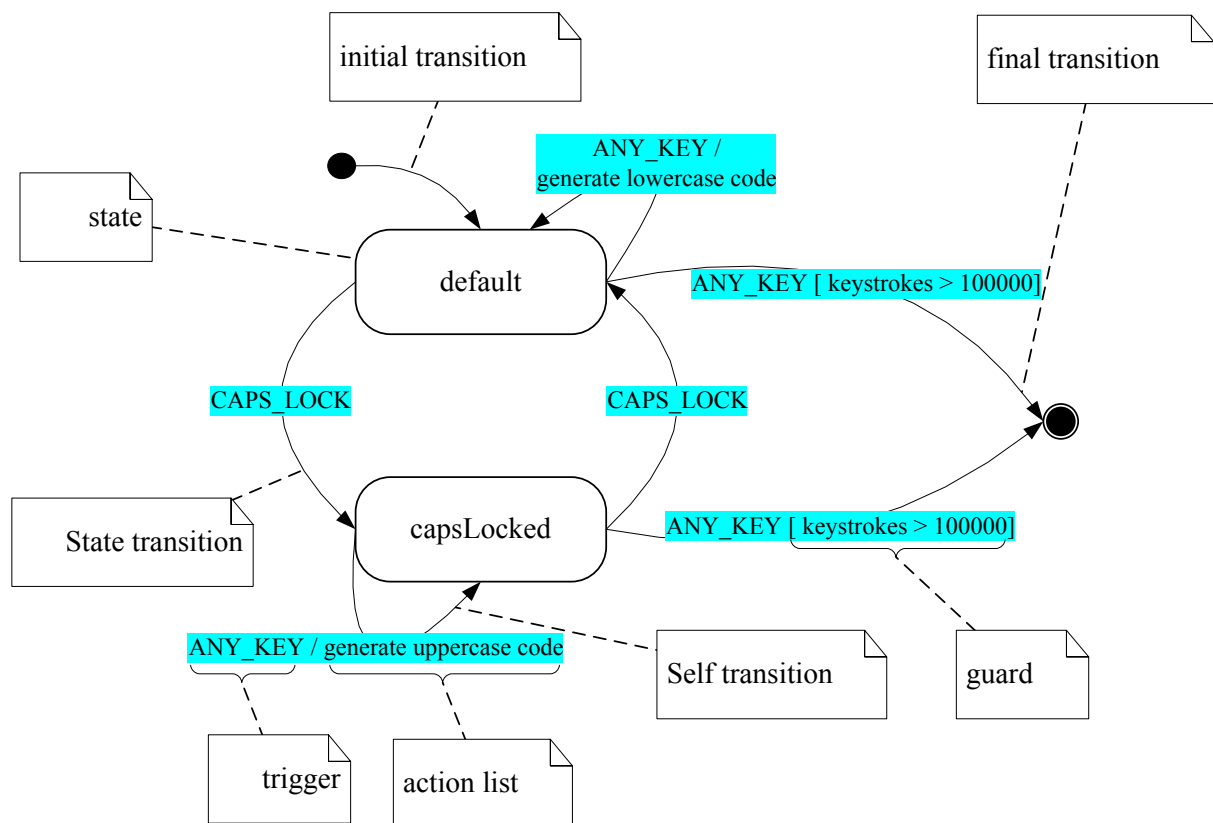


图 2-1 Keyboard FSM in UML format 1

State用圆角矩形来表示, 矩形中会写上state的名字。

Transition用箭头来表示，箭头上会写上Event的名字，可选的后面可以加上“ [guard] ”或者“ /action ”。

说明，Initial Transition(“初始状态迁移”)用实心的圆形和箭头来表示。Initial Transition就是FSM实例启动的时候的最初状态。每个状态机都应该有这么一个initial transition，其箭头上不应该有Event，因为这个迁移不是由事件触发的，但是箭头上可以写action，比如初始化一些变量或者初始化硬件等等。

Final Transition(“最终状态迁移”)用圆圈实心圆和箭头表示，其意义是FSM实例生命周期的结束。

Internal Transition(“内部状态迁移”)用从自己出发并指向自己的箭头表示，其意义是事件只引起内部动作，不引起状态迁移。

另外，也可以用另一种 UML 的画法。在这种画法中用圆角矩形框加一条横线表示 state，矩形框内横线上面写明 state 名字，横线下面写 internal-transition，如图 2-2。

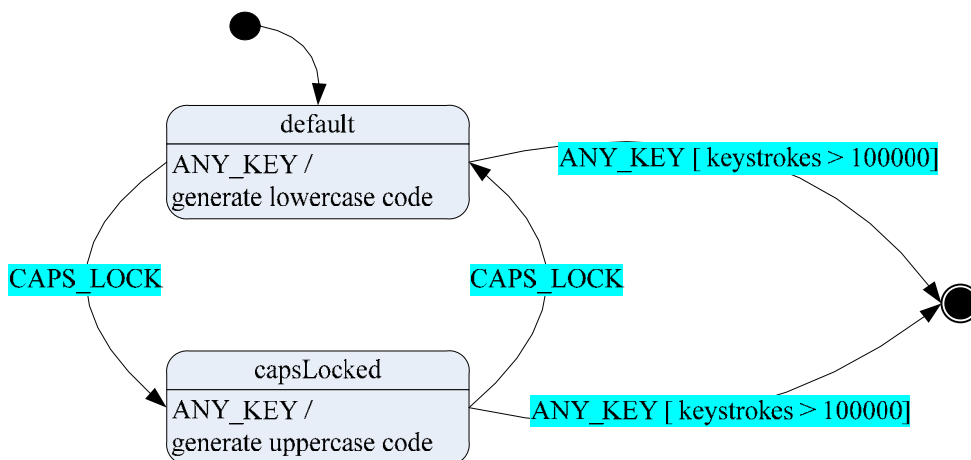


图 2-2 Keyboard FSM in UML format 2



## 第3章 FSM 设计方法

下面举 2 个例子来讲解一下 FSM 的设计方法。

### 3.1 C Parser(注释分析程序)

假如需要设计一个统计.C 文件中注释字符的个数的程序，这个程序需要分析出一个.C 文件中 C 风格的注释字符 (/\*...\*/) 的个数。“/\*...\*/”之外的字符统一认为是 code (不考虑“//”作为注释的情况)。

分析如下：

(1) 首先，我们可以假设一段输入字符，比如：

```
a = b * c / d; /* calc expression is b*c/d. */
```

(2) 因为 Event 比较清晰，可以先确定下来：

CHAR，表示字符。即除了斜杠和星号之外的所有字符。

SLASH，表示斜杠。

STAR，表示星号。

(3) 确定 Initial Transition：

确定初始状态为 code 态。(或者增加一个 idle 态，根据第一字符再判断。不过这个例子中不需要，因为不用计算 code 的个数。)

(4)确定所有的 state 和 transition。

在 code 态,如果输入是 CHAR 或者 STAR ,仍然处于 code 状态。当输入 SLASH 以后，则认为这个 SLASH 有可能是注释的开始，进入新的状态叫 slash。

在 slash 状态，检查后面的输入是否是 STAR，如果是 STAR 说明这个是注释的开始部分，转入 comment 状态，如果是 CHAR 或者 SLASH，则认为前面输入的 SLASH 和当前输入的字符仍然是 code，返回 code 态。

同样，comment 态的思路跟 code 态类似。star 态的思路跟 slash 态类似。

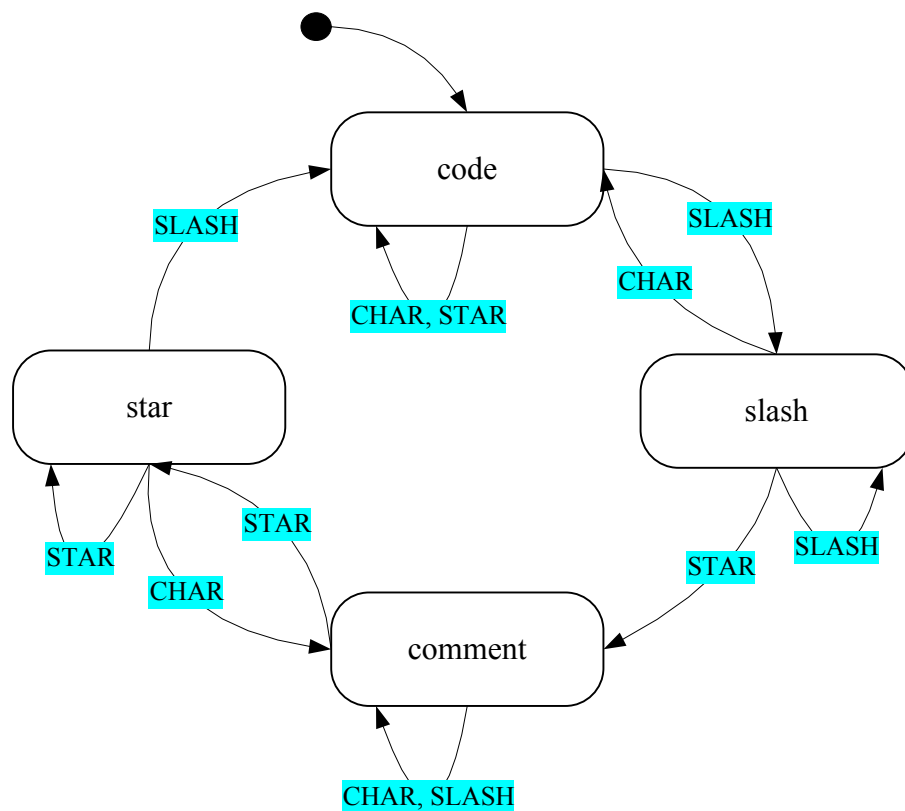


图 3-1 C comment parser (CParser) FSM

### 3.2 Calc(计算器)程序举例

假如需要设计一个四则运算计算器程序，实现“operand1 operator operand2 '='”，operand1 可以是整数或者小数或者是上一次的运算结果，operand2 是整数或者小数，操作符是“+ - \* /”。另外还要实现辅助键的功能，C 键清零，CE (Clear Entry) 键取消用户上一次的运算。界面如图 3-2。



图 3-2 Basic Calc Example

即：

1 expression ::= operand1 operator operand2 '='

2 operand1 ::= expression | ['+' | '-'] number

3 operand2 ::= ['+' | '-'] number

4 number ::= {'0' | '1' | ... '9'}\* ['.' {'0' | '1' | ... '9'}\*]

5 operator ::= '+' | '-' | '\*' | '/'

分析如下：

(1) 首先，考虑实现最基础的“operand1 operator operand2 '='”，比如实现表达式“1+2=”。因为状态可以很快确定下来，所以这个例子先考虑状态，再考虑事件。状态包括：

operand1：输入操作数 1

OpEntered：输入操作符

operand2：输入操作数 2

## (2) 确定 Initial Transition

初始状态为 operand1 状态。

## (3) 确定所有的 Event 和 Transition

从 operand1 状态开始分析, 这个状态的功能是要保证用户输入有效的操作数。很明显这个状态需要一些子状态来达到这个目的, 但是现在不考虑。从这个状态离开的触发条件是 operator (+ - \* /), 状态机进入 opEntered 状态, 在这个状态会等待用户输入 operand2。当用户按下数字 (0~9) 或者小数点, 则进入 operand2 状态。operand2 跟 operand1 类似。当用户按下“=”, 计算器就会运算并显示计算结果, 并回到 operand1 状态, 等待用户下一次输入。如图 3-3 示。

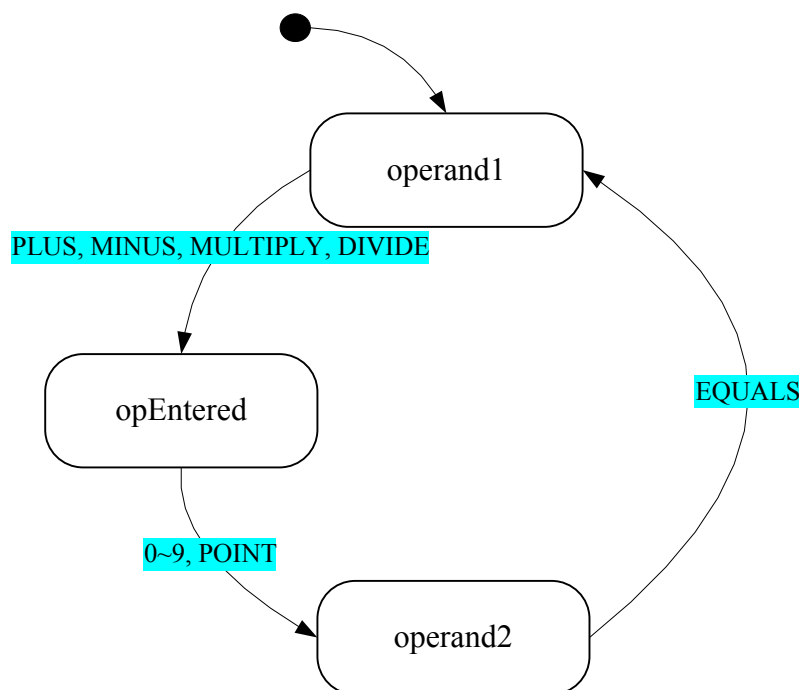


图 3-3 Basic Calc FSM

(4) 进一步分析。图3-3有个比较大的问题。当用户按下“=”以后, 状态机不能直接切换到operand1, 因为这将擦除显示结果。需要添加另外一个状态



“result”，在这个状态计算器会显示计算结果。在result状态，用户可能会有3种操作：

- 1) 用户按下了operator键，同时把第一次运算的结果作为operand1。
- 2) 用户按下Cancel键，重新开始一个计算
- 3) 用户按下数字键或者小数点，直接进入operand1状态。

图3-4显示了这种情况。另外，在此图中，还把信号进行了合并，把“+ - \* /”合并为一个信号，称作OPER。这个技巧很有用，它会简化设计。

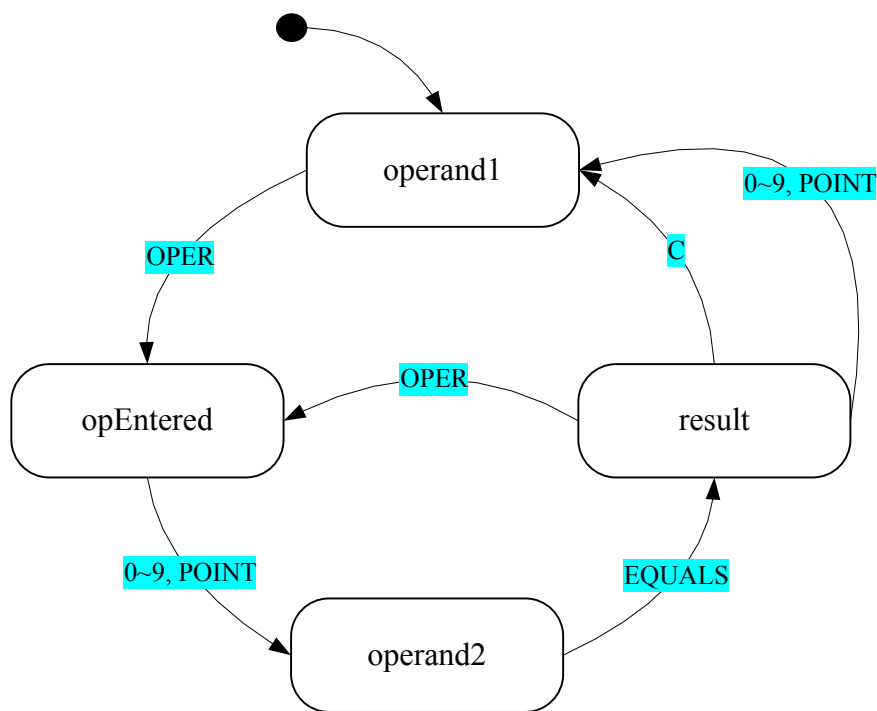


图 3-4 Basic Calc FSM add “Result” state

(5) 在图3-4中，Cancel键只会在result状态中处理。可是实际情况是，用户希望在任何情况都能处理Cancel键。如图3-5。

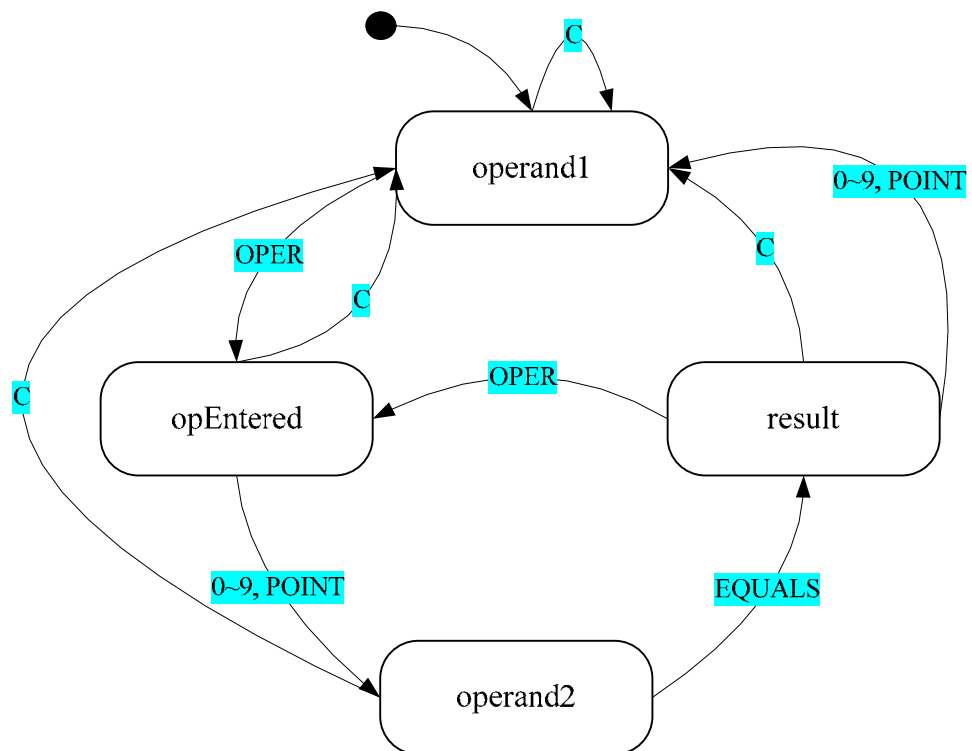


图 3-5 Basic Calc FSM add “Result” state and Cancel event

(6) 可是在图3-5中，有个比较不好的事情就是：所有状态都要处理Cancel键，状态机的设计和实现都很繁琐。有没有办法做到统一处理，使这种情况下的系统设计更简单呢？

有。那就是HSM，层次状态机，它也属于有限状态机的范畴，但是它跟普通的FSM的有很大的区别，那就是，普通的FSM的状态都是一个平面上的，而HSM的状态都是有层次关系的。如图3-6所示。

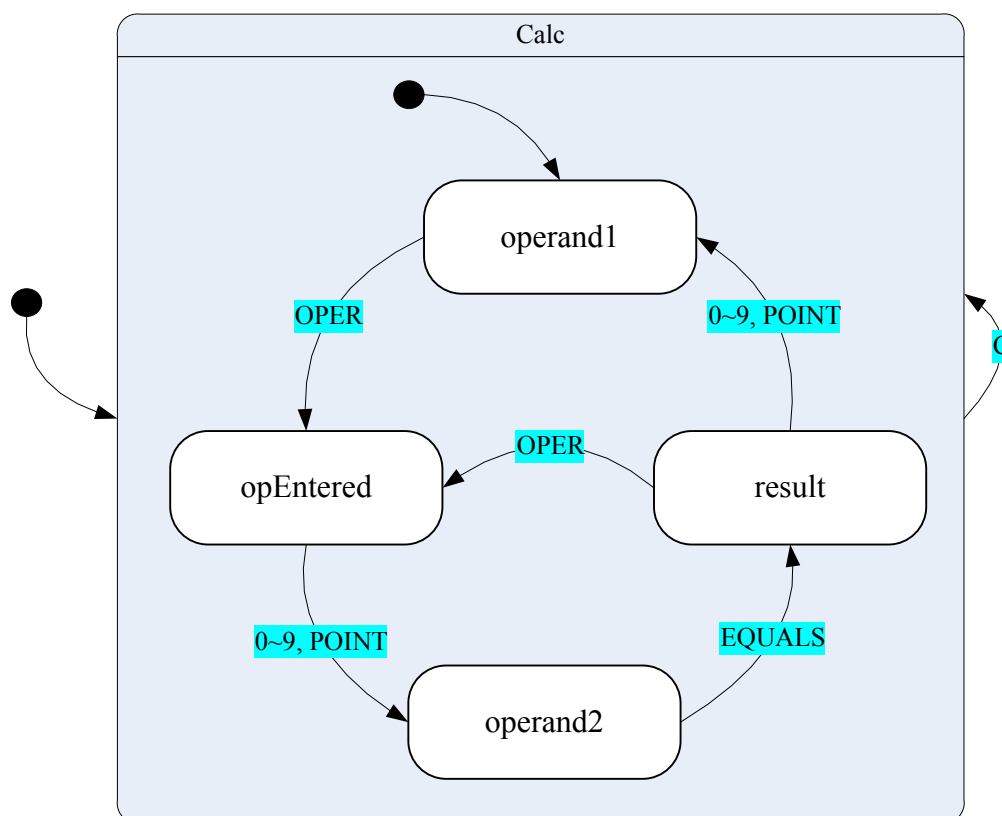


图 3-6 Simple Calc HSM

这样设计被简化了很多。HSM 的相关概念会在下面的章节中进行详细介绍。



## 第4章 HSM 概念

### 4.1 programming-by-difference (按照差异编程)

HSM 概念中有个一个很基础的概念,称为( programming-by-difference ) “按照差异编程”。让我们从这个概念开始。

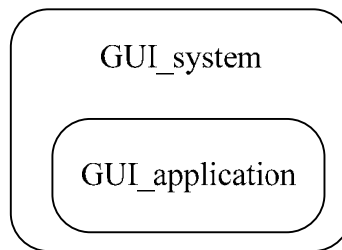


图 4-1 GUI system

举个例子,考虑一下 GUI 系统(比如 MS-Windows 的 GUI 系统或者手机上的 GUI 系统)。GUI 系统收到消息以后,首先把这个消息分发给应用(比如 MS-Windows 会调用一个函数,而手机上一般会调用应用窗口的消息处理函数),如果应用不处理这个消息,这个消息则会返回给系统,由系统进行默认处理。这样就建立了一个事件处理的层次关系。在层次关系中处于下层的“应用”,有机会处理所有消息,这样“应用”就能够自定义自己的行为。另外,所有没处理的消息都会返回到更高的一层(比如 MS-Windows 系统处理或者 GUI 默认处理),在这里它们会被按照标准的方法被处理,展现出 GUI 的标准 look and feel。这个例子就是“按照差异编程”的例子,因为“应用”只需要对和系统标准行为有差异的地方进行编程即可。

后来 David Harel(也就是创造 FSM 的那个人)把这种思想引入到状态机系统中,提出了 HSM(层次状态机)的概念。

### 4.2 HSM 图示

HSM 如图 4-2 A)或者 4-2 B)所示。它的图示方法跟图 2-1, 2-2 一样。只不过状态可以嵌套,外层的状态称为 superstate(父状态)。内层的状态称为 substate(子状态)。

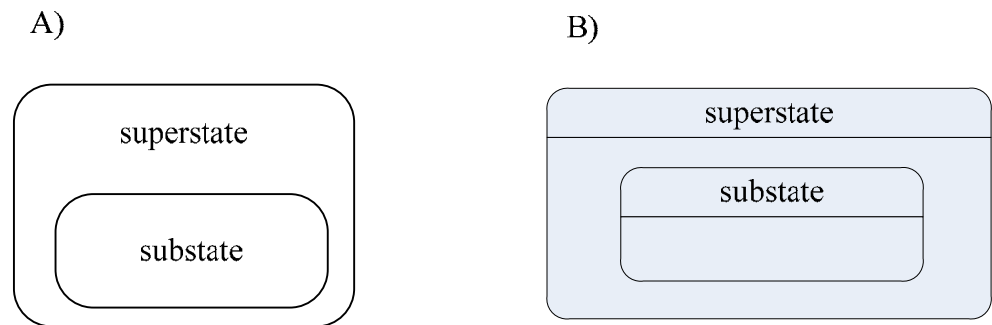


图 4-2 HSM conception

这个状态机的分发 Event 的流程是：首先 Event 交给 substate 处理，如果 substate 不处理，状态机会自动的把这个 Event 交给他的 superstate 处理。（这点很强大，请记住）

### 4.3 HSM 分析和 OOP 分析

如果你了解面向对象分析方法，你可以发现 HSM 分析和 OOP 分析有很多类似的地方。

比如 OOP 被认为采用了 3 个最基础的概念：抽象（abstraction），继承（inheritance）和多态（polymorphism）。层次状态机也支持这 3 种概念。请看下面描述：

#### 4.3.1 state inheritance and class inheritance（状态层次和类层次）

OOP 的基石之一是 class inheritance（“类继承”），它允许你基于一个旧的类定义一个新的类，使类能够按照层次关系组织起来。而状态的层次关系实际上也是一种继承方式，本文称其为 behavioral inheritance（“行为继承”）。

要理解行为继承怎么工作的，请参考图 4-2，假设 substate 不处理任何事件，那么所有的事件都由 superstate 处理，即 substate 的行为跟 superstate 完全一致，也就是 substate 继承了 superstate 的行为。这就是行为继承。

如果子状态需要跟父状态产生差异的话，和类继承相似，子状态可以通过添加新的状态迁移或者覆盖其父状态的状态迁移来进行。这样层次状态机也就实现了多态。

在典型的类继承中，子类比父类更具体，父类比子类更概括，这种情况在行为继承中也是类似的。举个例子，考虑如下图，假定在进入 failed 状态的时候会响起警铃，在退出的时候会关闭警铃。而现在，系统进入 substate 也就是 unsafe 态，同样警铃也会响，因为进入了 unsafe 状态也就进入了 failed 状态。行为继承是对 is-in-a-state (“ 在某个状态 ”) 中的抽象，与此相应的，类继承是 is-a-kind-of (“ 是某一种 ”) 的抽象。

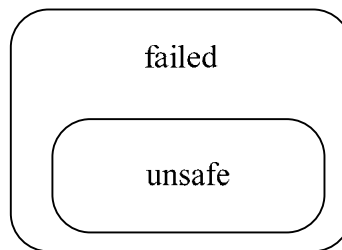


图 4-3 Example of HSM

#### 4.3.2 Entry/Exit Actions and Construction/Destruction (进入/退出状态和构造/析构类)

在前面的例子中，进入一个状态的时候，Entry 的操作会被自动执行，退出状态的时候 Exit 操作会被自动的执行。这两个动作和类的 Construction、Destruction 方法类似，都是按照预先定义好的顺序执行的：Construction 是从其最高的父类开始，一直到子类，Destruction 则相反；Entry 动作是从最顶端的状态开始的，一直到子状态，Exit 动作则相反。

#### 4.3.3 programming-by-difference (按照差异编程)

类继承是被广泛采用的按照差异编程的方法。这种编程方法的本质是“复用”：一个子类只需要定义它和父类的差别。行为继承也是出于同样的考虑。子状态只需要定义和其父状态不同的地方，否则就复用其父状态的行为。

#### 4.3.4 abstraction (抽象)

用层次状态机另一个和 OOP 类似的地方是：抽象。它使开发者能侧重于系统的一部分，而不必面对一个复杂系统的所有方面。HSM 是一种用来隐藏内部细节的理想机制，因为开发者很容易的 zoom in 或者 zoom out 来显示或者隐藏嵌套的状态，只关注同一层次上的状态。尽管抽象并不能减少系统的总的复杂程度，但是它可以使你减少在某一时刻要处理的细节。



## 第5章 HSM 设计方法

下面我们开始用行为继承的方法进行 HSM 设计。

### 5.1 继续进行 Calc 设计

有了上一章的经验，我们可以把第三章中没有实现完毕的 Calc HSM 继续设计下去了：

(7) 考虑增加小数点的支持。考虑 operand1 或者 operand2 有可能为 0，或者整数，或者小数。也就是 operandX 的子状态如图 5-1：

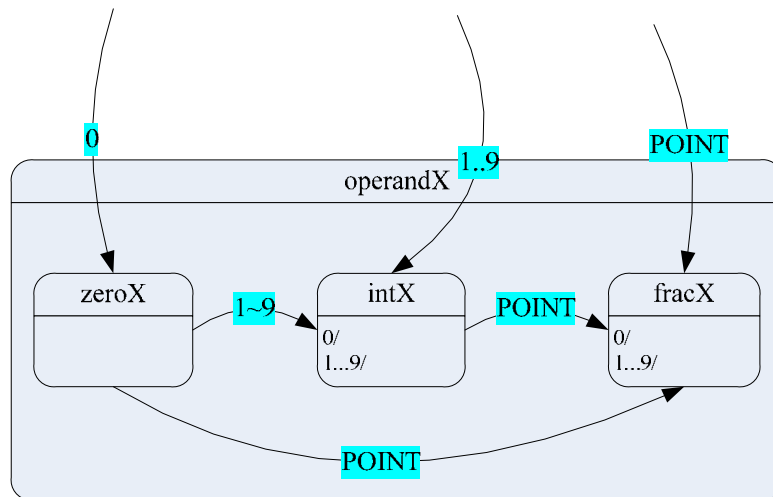


图 5-1 substate of operandX

(8) 考虑 CE (Cancel Entry) 键功能和负数，完成完整的 Calc HSM 设计，如图 5-2：

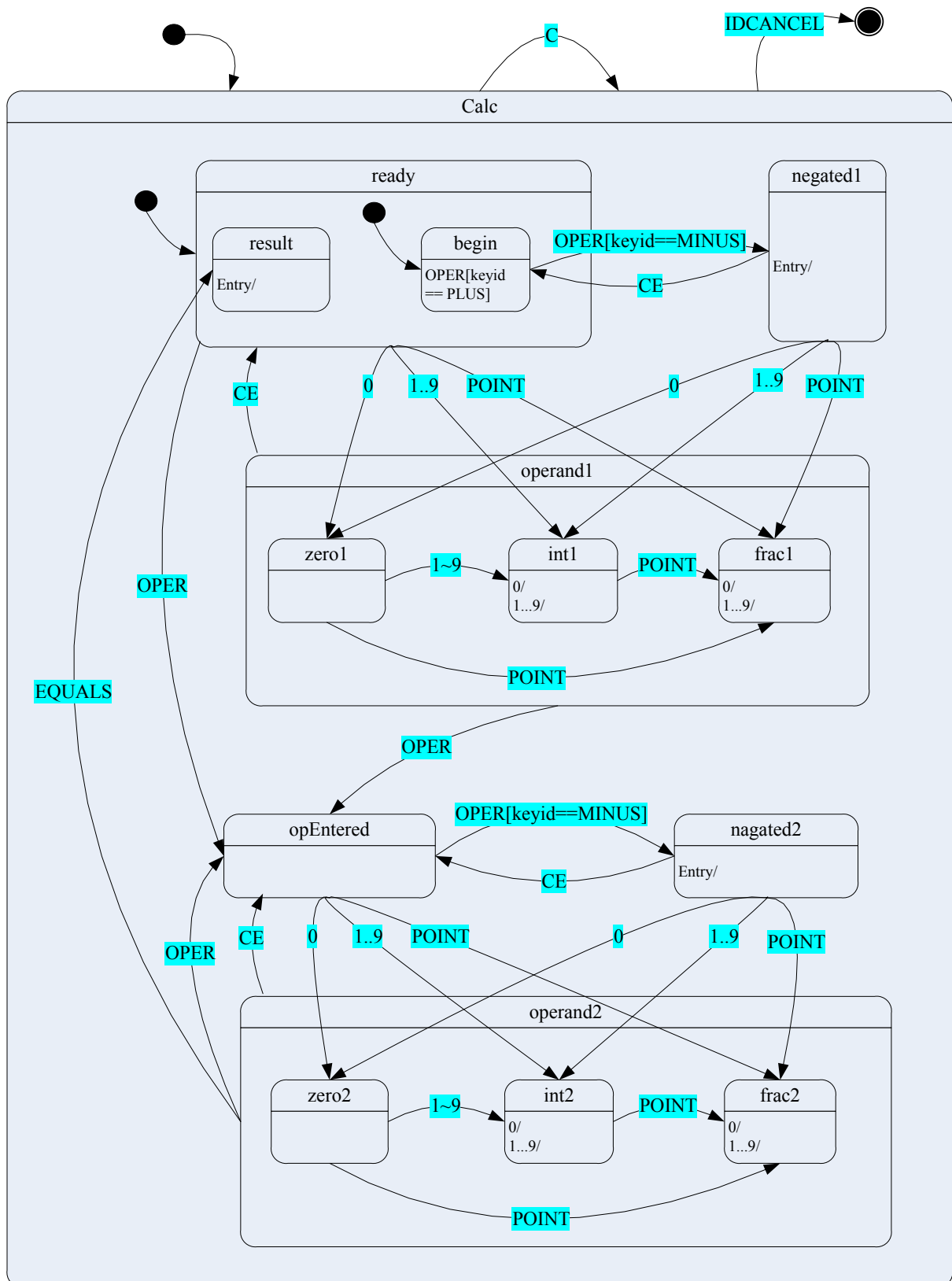


图 5-2 Full Calc HSM

在阅读图 5-2 的时候，你可以通过 zoom in、zoom out 的方法来对每一个 level 的 HSM 进行抽象。

通过前面的阅读，大家可能发现，行为继承的核心在于合理的抽象出父状态和子状态。下面进行讲解。

## 5.2 继承关系是否合理

怎样判断子状态和父状态的继承关系是否合理呢，自己的设计中是否有一些不合理的继承关系呢？

可以采用类继承里常用的“里氏替代法则”(Liskov Substitution Principle, LSP)来进行判断。LSP 法则要求：在所有使用父类的场合，其子类能够完全替代父类。

怎样理解 LSP 法则呢？“在使用父类的场合”，可以理解为父类的实例相关的方法。“换成其子类也能同样正确运行”，也就是父类提供出来的方法，如果用子类的方法替代的话，子类和父类的运行结果是一样的，即，在父类提供出来的方法上，子类必须和父类保持一致。LSP 法则在 HSM 中的解释是，父状态对某个 Event 响应的 Action 和 Transition,其子类必须与其保持一致。

举一个通俗一点的例子：比如有一个组对外宣称，我们组只要外面喊某一个口令，我们组所有人都会向下走。而如果实际结果不是这样，外面喊一个口令，大家都朝下走，除了一个人朝上走。那么这种继承是不符合 LSP 法则的。如果要修改这种状况，要么把不听话的人从这个组中拿掉，所有听话的人组成新的组，要么修改组的对外宣称。

按照 LSP 法则设计出来的 HSM，能够保证你建立正确的状态继承关系，并且有效的进行抽象。只有所有的子状态和其父状态一致，这样的抽象才是有意义的。否则，如果违反了 LSP 法则的话，zoom out 出来然后忽略其子状态，就有可能产生错误。

可以用另外一个方法来解释 LSP 法则。1988 年，B. Meyer 提出了 Design by Contract (契约式设计) 理论。DBC 从形式化方法中借鉴了一套确保对象行为和自身状态的方法。

- 1) 每个方法调用之前，该方法应该校验传入参数的正确性，只有正确才能执行该方法，否则认为调用方违反契约，不予执行。这称为前置条件(Pre-condition)。
- 2) 一旦通过前置条件的校验，方法必须执行，并且必须确保执行结果符合契约，这称之为后置条件(Post-condition)。
- 3) 对象本身有一套对自身状态进行校验的检查条件，以确保该对象的本质不发生改变，这称之为不变式(Invariant)。

以上是单个对象的约束条件。为了满足 LSP，当存在继承关系时，子类中方法的前置条件必须与父类中被覆盖的方法的前置条件相同或者更宽松；而子类中方法的后置条件必须与父类中被覆盖的方法的后置条件相同或者更为严格。

在子类重载父类方法的时候要小心，尽量别违反 LSP 法则。

此时，回过头来再看一下图 5-2，研究一下是否有不符合 LSP 法则的地方？实际上是有的，begin 态对 OPER[key==MINUS]的处理跟其父状态 ready 态对 OPER 的处理是有冲突的，不符合 LSP 原则。

### 5.2.1 Transition 迁移执行顺序

和 entry/exit 操作绑定在一起的 HSM 迁移，很明显的比普通的 FSM 复杂。由于继承关系的存在，实际上嵌套的各个状态构成了一个树形的结构。比如把图 5-2 中的 state 按照层次关系抽象出来，如图 5-3。

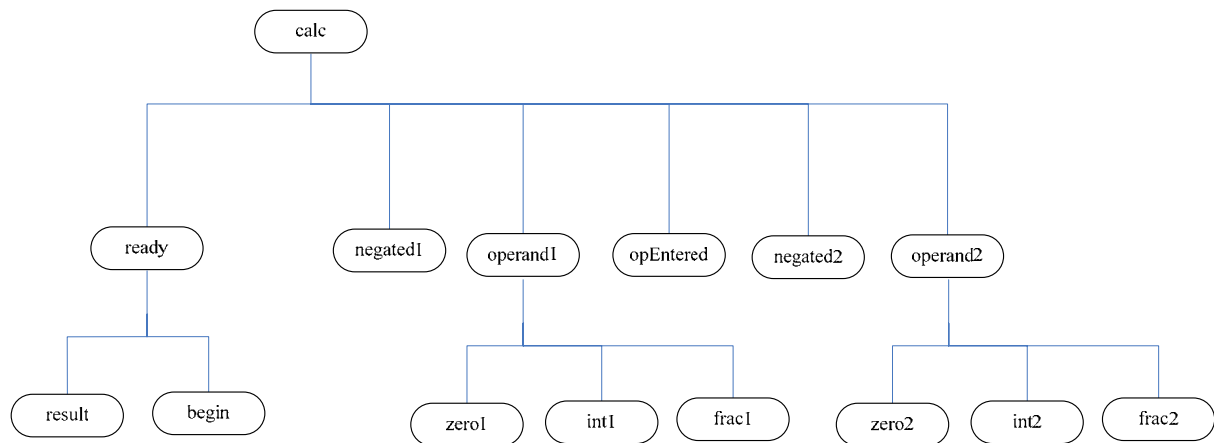


图 5-3 State Tree of Calc

状态迁移的实际执行操作是：确定原状态和目的状态的最小公共父状态(least common ancestor) LCA，执行原状态到 LCA（不含）之间状态 exit 操作，执行 transition 相关的 action，执行 LCA（不含）到目的状态之间的状态的 entry 操作。

比如从 int2 态到 result 态，执行的操作是：int2 态 exit，operand2 态 exit，执行状态变量赋值，ready 态 entry，result 态 entry。

## 第6章 HSM 在实际工程的应用

下面会给出几个在 PoC 项目中用到的比较典型的 HSM 应用作为例子，供大家参考。例子阅读的重点在于怎样抽象父状态。

### 6.1 PoC Audio Player

假如需要设计一个audio player模块，要求其功能是对从网络传过来的流数据进行播放。其流程是，当收到播放请求以后，开始对数据进行消抖（方法是在收到网络传过来的数据以后，延时一段时间再播放），然后创建audiohandle（同步函数），调用底层audio播放函数（异步函数），等播放完毕会收到播放完毕的消息，然后从buffer中读取数据继续播放，直到用户停止播放。

另外，需要考虑一个问题，就是实现设置audio参数的功能，用户可以在播放前设置参数，audio player模块需要把这些参数保存，而在play过程中用户修改参数的话，需要保存参数并且实时的控制底层audio模块修改播放的参数。

下图描述了这套流程：（图中的“R:”表示Receive，“S:”表示Send）

- (1) Initial transition到idle态。
- (2) 在idle态，收到POC\_AUDIO\_START\_PLAY\_REQ以后，会进入dejitter状态。
- (3) dejitter状态的作用是对网络数据进行消抖，在进入这个状态的时候会启动timer，等timer超时会创建audio handle，会把一段buffer传给下层audio模块并开始播放。然后进入StartPlay态。等这段数据播放完毕会收到POC\_AUDIO\_PLAY\_OK\_IND，然后检查dejitter buffer中是否还有数据需要播放（这是一个guard（“条件”）），如果没有则进入PlayDone态。如果有则是internal-transition（“内部迁移”），继续进行播放，状态不变。
- (4) 在PlayDone态，当收到POC\_AUDIO\_START\_PLAY\_REQ以后，又会进入StartPlay态进行播放。



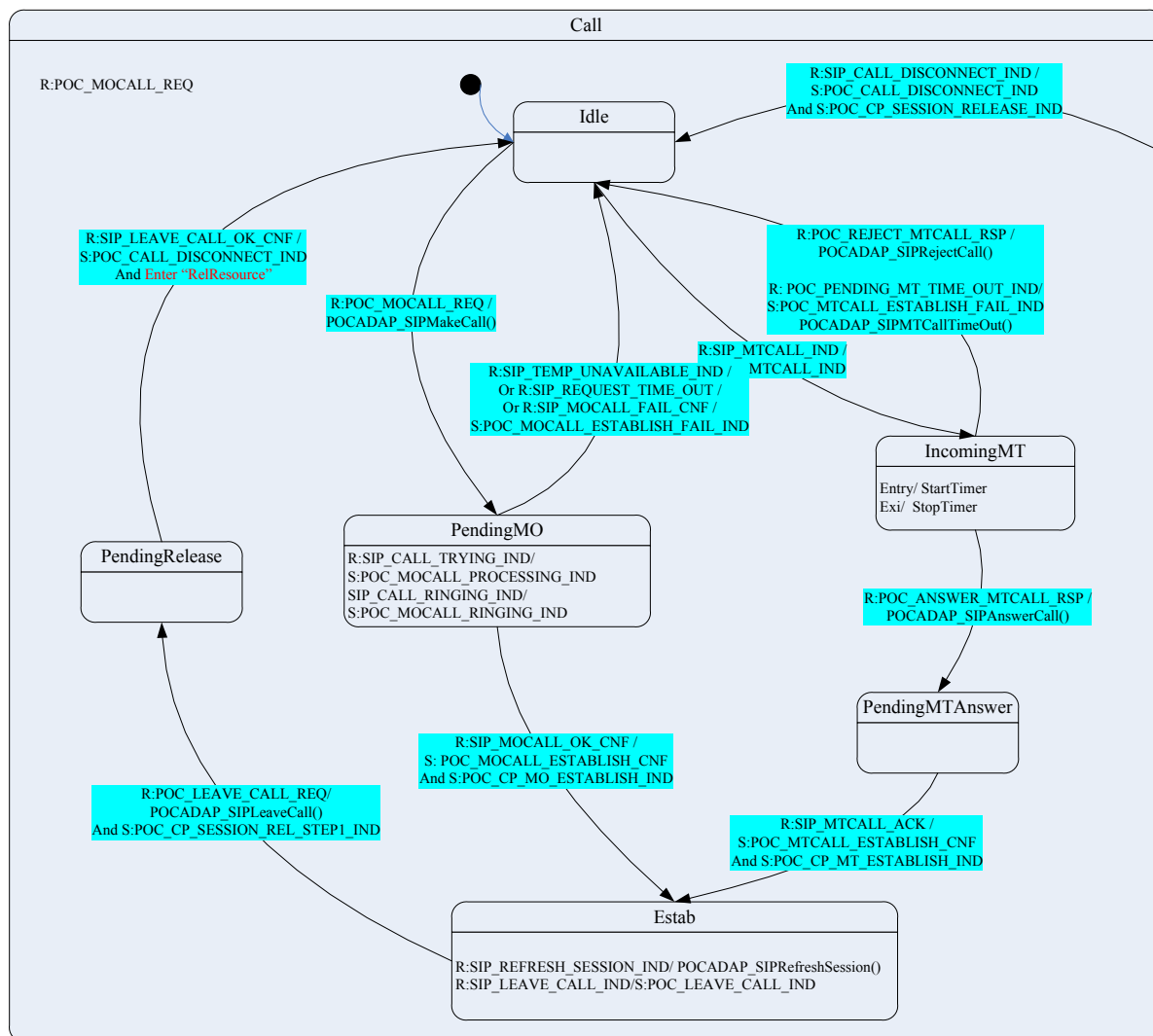


图 6-2 HSM of Call Control

可以关注的几个地方：

(1) 对POC\_MOCALL\_REQ（发起通话请求）的处理。因为上层应用有可能在任何时候都发送这个请求过来，而本HSM在一些情况下是不允许这个请求的（比如PendingRelease的时候），这时候就可以把出错处理统一交给其父状态即最外层的Call状态处理，其action是返回错误码给上层即可。

(2) 任何状态收到下层报上来的出错消息，比如SIP\_CALL\_DISCONNECT\_IND，都要返回idle态。这个event就可以交给其父状态处理。



## 第7章 FSM 实现

下面介绍一下 FSM 的几种典型的 C 语言实现方法。以 CParser FSM 图 3-1 为例。阅读代码的重点请放在 dispatch 函数和 state transition 函数和粗体的代码上( 其他的辅助代码已经删除 )。Initial Transition 虽然是必须的但没做重点描述。另外, 下面的例子中有些代码有 CLASS 或者 SUBCLASS 关键字, 因为这部分代码是仿照 C++写的, 把这 2 个关键字理解为 C 中的 STRUCTURE 即可。

### 7.1 nested switch statement ( 嵌套 switch )

#### Cparser1.h

```
enum Signal {                                /* enumeration for CParser signals */
    CHAR_SIG, STAR_SIG, SLASH_SIG
};
enum State {                                /* enumeration for CParser states */
    CODE, SLASH, COMMENT, STAR
};

struct CParser1 {
    enum State state__;                      /* the scalar state-variable */
    long commentCtr__;                      /* comment character counter */
    /* ... */                               /* other CParser1 attributes */
};

#define CParser1Tran(me_, target_) ((me_)->state__ = target_)
```

#### Cparser1.c

```
void CParser1Dispatch(CParser1 *me, unsigned const sig) {
    switch (me->state__) {
    case CODE:
        switch (sig) {
        case SLASH_SIG:
            CParser1Tran(me, SLASH);          /* transition to "slash" */
            break;
        }
        break;
    case SLASH:
        switch (sig) {
        case STAR_SIG:
            me->commentCtr__ += 2;             /* SLASH-STAR count as comment */
            CParser1Tran(me, COMMENT);        /* transition to "comment" */
            break;
        case CHAR_SIG:
            CParser1Tran(me, CODE);           /* go back to "code" */
            break;
        }
        break;
    case COMMENT:
        switch (sig) {
```

```

        case STAR_SIG:
            CParser1Tran(me, STAR);          /* transition to "star" */
            break;
        case CHAR_SIG:
        case SLASH_SIG:
            ++me->commentCtr__;              /* count the comment char */
            break;
    }
    break;
case STAR:
    switch (sig) {
        case STAR_SIG:
            ++me->commentCtr__;              /* count STAR as comment */
            break;
        case SLASH_SIG:
            me->commentCtr__ += 2;           /* count STAR-SLASH as comment */
            CParser1Tran(me, CODE);         /* transition to "code" */
            break;
        case CHAR_SIG:
            me->commentCtr__ += 2;           /* count STAR-? as comment */
            CParser1Tran(me, COMMENT);      /* go back to "comment" */
            break;
    }
    break;
}
}
}

```

分析：这个是最简单、最常用的 FSM 实现方法。通过嵌套的 2 层 switch case 实现。外层 switch case 判断状态，内层 switch case 判断消息。状态采用枚举量。

优点：结构简单，便于理解。

缺点：代码冗长。

结论：这种代码结构感觉比较冗长，不建议在复杂的状态机中使用。

## 7.2 state table(状态表)

### Cparser2.h

```

CLASS (Tran)
    Action action;
    unsigned nextState;
METHODS
END_CLASS

enum State {                      /* enumeration for CParser states */
    CODE, SLASH, COMMENT, STAR, MAX_STATE
};

```

### Cparser2.c

```

void StateTableDispatch(StateTable *me, unsigned const sig) {
    register Tran const *t = me->table__ +

```

```

        me->state__ *me->nSignals__ + sig;
    (t->action)(me);
    me->state__ = t->nextState;
}
void StateTableDoNothing(StateTable *me) {}

static Tran const locTable[MAX_STATE][MAX_SIG] = {
    {{StateTableDoNothing, CODE}},
    {{StateTableDoNothing, CODE}},
    {{StateTableDoNothing, SLASH}},
    {{StateTableDoNothing, CODE}},
    {{(Action)CParserA2, COMMENT}},
    {{StateTableDoNothing, CODE}},
    {{(Action)CParserA1, COMMENT}},
    {{StateTableDoNothing, STAR}},
    {{(Action)CParserA1, COMMENT}},
    {{(Action)CParserA2, COMMENT}},
    {{(Action)CParserA1, STAR}},
    {{(Action)CParserA2, CODE}}
};

void CParserA1(CParser2 *me) { me->commentCtr__ += 1; }
void CParserA2(CParser2 *me) { me->commentCtr__ += 2; }

```

分析：这种 FSM 实现方法对 nested switch statement 进行了改良。根据 state、event 做成了 2 维表格，表格中的项就表示 action 和 transition。状态采用枚举量。

优点：结构简单，便于理解。代码比较简练。效率最高。

缺点：代码结构不是太好。

结论：这是一种常用的实现方法。如果没有更好的选择，可以使用。

### 7.3 Function Address As State (用函数指针作为状态)

#### Cparser4.h

```

typedef void (*State) /* return value */
              /* pointer-to-member name */
              (Fsm *me, /* class the function pointer is a member of */
               unsigned const sig); /* the rest of the argument list */

struct Fsm {
    State state_; /* protected current state */
};

#define FsmDispatch(me_, sig_) ((*me_)->state_)((me_), sig_)
#define TRAN(target_) (((Fsm *)me)->state_ = (State)(target_))

```

如果要支持 Entry 和 Exit 方法，可以改为：

```

#define TRAN(target_) do{                                     \
    *((Fsm *)me)->state_)((me), EXIT_SIG);                 \
    ((Fsm *)me)->state = (State)(target_);                 \
    *((Fsm *)me)->state_)((me), ENTRY_SIG);                 \
}while(0)

Cparser4.c

void CParser4code(CParser4 *me, unsigned const sig) {
    switch (sig) {
        case SLASH_SIG:
            TRAN(CParser4slash); /* transition to "slash" */
            break;
    }
}

void CParser4slash(CParser4 *me, unsigned const sig) {
    switch (sig) {
        case STAR_SIG:
            me->commentCtr__ += 2; /* SLASH-STAR chars count as comment */
            TRAN(CParser4comment); /* transition to "comment" */
            break;
        case CHAR_SIG:
            TRAN(CParser4code); /* go back to "code" */
            break;
    }
}

void CParser4comment(CParser4 *me, unsigned const sig) {
    static const void (*lookup[])(CParser4 *) = {
        CParser4commentOnCHAR,
        CParser4commentOnSTAR,
        CParser4commentOnSLASH
    };
    REQUIRE(sig <= SLASH_SIG); /* signal must be in range */
    (*lookup[sig])(me); /* rapidly dispatch without 'switch' */
}

void CParser4star(CParser4 *me, unsigned const sig) {
    switch (sig) {
        case STAR_SIG:
            ++me->commentCtr__; /* count '*' as comment character */
            break;
        case CHAR_SIG:
            me->commentCtr__ += 2; /* count STAR-? as comment */
            TRAN(CParser4comment); /* go back to "comment" */
            break;
        case SLASH_SIG:
            me->commentCtr__ += 2; /* count STAR-SLASH as comment */
            TRAN(CParser4code); /* transition to "code" */
            break;
    }
}

```

分析：这种 FSM 实现方法用函数地址代替 state 值（请看“State”的声明），比如 CParser4slash() 函数 就用 CParser4slash 函数的地址来作为 slash

状态的值。当状态机分发 event 的时候，FsmDispatch() 函数会直接调用当前 state 对应的处理函数。

优点：代码框架好，比较直观。可以方便的增加 entry/exit 操作。

结论：效率较高（比 nested switch statement 高，比 state table 效率低）。代码框架好。

## 7.4 QFSM frame (QFSM 框架)

### qfsm.h

```
SUBCLASS(QFsm, Object) /* Quantum Finite State Machine base class */
    QFsmState state__; /* the active state */
VTABLE(QFsm, Object)

#define QFsmDispatch(me_, e_) (*(me_)->state__)((me_), (e_))

#define QFSM_TRAN(target_) \
    (((QFsm *)me)->state__ = (QFsmState)(target_))
```

如果支持Entry和Exit方法，可以改为：

```
QFSM_TRAN(me, target_) do{\
{\
    QEvent e; \
    e.sig = EXIT_SIG; \
    (*(me)->state__)((me), (e_)); \
    ((QFsm *)me)->state__ = (QFsmState)(target_); \
    e.sig = ENTRY_SIG; \
    (*(me)->state__)((me), (e_)); \
}while(0)
```

### Cparser5.h

```
#include "qfsm.h"
enum Event { /* enumeration for CParser events */
    CHAR_SIG = Q_USER_SIG, STAR_SIG, SLASH_SIG
};

SUBCLASS(CParser5, QFsm)
    long commentCtr__; /* comment character counter */
METHODS
    CParser5 *CParser5Ctor(CParser5 *me); /* Ctor */
    void CParser5initial(CParser5 *me, QEvent const *e);
    void CParser5code(CParser5 *me, QEvent const *e);
    void CParser5slash(CParser5 *me, QEvent const *e);
    void CParser5comment(CParser5 *me, QEvent const *e);
    void CParser5star(CParser5 *me, QEvent const *e);
    #define CParser5GetCommentCtr(me_) ((me_)->commentCtr__)
END_CLASS
```

## Cparser5.c

```
void CParser5code(CParser5 *me, QEvent const *e) {
    switch (e->sig) {
        case SLASH_SIG:
            QFSM_TRAN(CParser5slash);          /* transition to "slash" */
            break;
    }
}

void CParser5slash(CParser5 *me, QEvent const *e) {
    switch (e->sig) {
        case STAR_SIG:
            me->commentCtr__ += 2;              /* SLASH-STAR count as comment */
            QFSM_TRAN(CParser5comment);         /* transition to "comment" */
            break;
        case CHAR_SIG:
            QFSM_TRAN(CParser5code);            /* go back to "code" */
            break;
    }
}

void CParser5comment(CParser5 *me, QEvent const *e) {
    switch (e->sig) {
        case STAR_SIG:
            QFSM_TRAN(CParser5star);           /* transition to "star" */
            break;
        case CHAR_SIG:
        case SLASH_SIG:
            ++me->commentCtr__;                 /* count the comment character */
            break;
    }
}

void CParser5star(CParser5 *me, QEvent const *e) {
    switch (e->sig) {
        case STAR_SIG:
            ++me->commentCtr__;                 /* count '*' as comment character */
            break;
        case CHAR_SIG:
            me->commentCtr__ += 2;              /* count STAR-? as comment */
            QFSM_TRAN(CParser5comment);         /* go back to "comment" */
            break;
        case SLASH_SIG:
            me->commentCtr__ += 2;              /* count STAR-SLASH as comment */
            QFSM_TRAN(CParser5code);            /* transition to "code" */
            break;
    }
}
```

分析：QFSM 实现方法对第三种方法（Function Address As State 方法）做了进一步改良，并且把 fsm 的 framework 作为公共的函数库提出来，包括 qfsm.h 和 qfsm.c 等其他辅助文件。（其作者是 Miro Samek, Ph.D.）

优点：代码非常整齐、直观。可以方便的 FSM UML 图转化成 code。可以方便的增加 entry/exit 操作。

结论：效率较高，代码整齐，使用方便，强烈推荐使用。





## 第8章 HSM 实现

以 Calc HSM 图 5-2 为例进行讲解。因为 QHSM 的代码框架和 QFSM 的相似，下面只给出一小部分代码，initial transition 也没有列出，完整代码请参考附件。

### qhsm.h

```
SUBCLASS(QHsm, Object)    /* Hierarchical State Machine base class */
    QState state__;        /* the active state */
    QState source__;       /* source state during a transition */
VTABLE(QHsm, Object)

    void QHsmDispatch(QHsm *me, QEvent const *e); /* take RTC step */

#define Q_TRAN(target_) if (1) { \
    static Tran t_; \
    QHsmTranStat_((QHsm *)me, &t_, (QState)(target_)); \
}

#define Q_TRAN_DYN(target_) \
    QHsmTran_((QHsm *)me, (QState)(target_))
```

### calc.h

```
#include "qhsm.h"
SUBCLASS(CalcEvt, QEvent)
    int keyId;                /* ID of the key depressed */
METHODS
END_CLASS

SUBCLASS(Calc, QHsm)
    HWND hWnd_;              /* the calculator window handle */
    BOOL isHandled_;
    char display_[40];
    char *ins_;
    double operand1_;
    double operand2_;
    int operator_;
METHODS
    Calc *CalcCtor(Calc *me);
    void CalcXtor(Calc *me);
    Calc *CalcInstance(int id); /* static Singleton accessor method */
    void Calc_initial(Calc *me, QEvent const *e);
    QSTATE Calc_calc(Calc *me, QEvent const *e);
    QSTATE Calc_ready(Calc *me, QEvent const *e);
    QSTATE Calc_result(Calc *me, QEvent const *e);
    QSTATE Calc_begin(Calc *me, QEvent const *e);
    QSTATE Calc_negated1(Calc *me, QEvent const *e);
    QSTATE Calc_operand1(Calc *me, QEvent const *e);
    QSTATE Calc_zero1(Calc *me, QEvent const *e);
    QSTATE Calc_int1(Calc *me, QEvent const *e);
    QSTATE Calc_frac1(Calc *me, QEvent const *e);
    QSTATE Calc_opEntered(Calc *me, QEvent const *e);
    QSTATE Calc_negated2(Calc *me, QEvent const *e);
    QSTATE Calc_operand2(Calc *me, QEvent const *e);
    QSTATE Calc_zero2(Calc *me, QEvent const *e);
    QSTATE Calc_int2(Calc *me, QEvent const *e);
    QSTATE Calc_frac2(Calc *me, QEvent const *e);
```

```
void CalcClear_(Calc *me);
void CalcInsert_(Calc *me, int keyId);
void CalcNegate_(Calc *me);
void CalcEval_(Calc *me);
void CalcDispState(Calc *me, char const *s);
END_CLASS
```

### calc.c

大体的思路跟QFSM类似，这里只举一个state的处理函数的例子。

```
QSTATE Calc_ready(Calc *me, QEvent const *e) {
    BOOLEAN is_handled = TRUE;

    switch (e->sig) {
    case Q_ENTRY_SIG:
        CalcDispState(me, "ready");
        break;
    case Q_INIT_SIG:
        Q_INIT(Calc_begin);
        break;
    case IDC_0:
        CalcClear_(me);
        Q_TRAN_DYN(Calc_zero1);
        break;
    case IDC_1_9:
        CalcClear_(me);
        CalcInsert_(me, ((CalcEvt *)e)->keyId);
        Q_TRAN_DYN(Calc_int1);
        break;
    case IDC_POINT:
        CalcClear_(me);
        CalcInsert_(me, IDC_0);
        CalcInsert_(me, ((CalcEvt *)e)->keyId);
        Q_TRAN_DYN(Calc_frac1);
        break;
    case IDC_OPER:
        sscanf(me->display_, "%lf", &me->operand1_);
        me->operator_ = ((CalcEvt *)e)->keyId;
        Q_TRAN_DYN(Calc_opEntered);
        break;
    default:
        is_handled = FALSE;
        break;
    }

    if (is_handled)
    {
        return 0;
    }
    else
    {
        return (QSTATE)Calc_calc;
    }
}
```

分析：QHSM是一套HSM的实现框架，它包括qhsm.h和qhsm.c文件等其他辅助文件。他的一些实现方法跟前面的QFSM frame类似，也是用函数指针代替

状态。不过 HSM 比 FSM 多了很多特性，以 ready 态的处理函数 (Calc\_ready) 为例。描述一下 QHSM 的框架和 HSM 的执行情况：

- 2) SUPERCLASS QHsm 中会记录 Calc HSM instance 的状态 (函数指针)。当收到一个消息，QF 框架会直接调用之。
- 3) 状态的层次关系的体现：某个状态的处理函数中，如果这条消息需要这个状态处理，它的返回值就是 0。如果本状态不处理，则返回其父状态的指针。比如 ready 态会处理 event IDC\_0，它处理完毕就返回 0，对于其他不处理的 event 则返回 Calc\_calc，也就是说其父状态就是 Calc 态。另外，QF 自己定义了一个最顶层的状态叫 QTop()。
- 4) QF 自己定义了几个系统消息，包括：

Q\_EMPTY\_SIG = 0, //空消息，用于检测状态之间的父子关系的，外部不能响应这个消息。

Q\_INIT\_SIG = 1, //定义initial transition的消息，看代码，比如 ready态的初始状态是begin，当state transition到ready态以后，它就会自动迁移到begin态。

Q\_ENTRY\_SIG = 2, //Entry消息

Q\_EXIT\_SIG = 3 //Exit 消息

- 5) Transition：当调用 Q\_TRAN\_DYN 进行状态迁移的时候，QF 会计算源状态和目标状态的公共父状态，会给源状态到公共父状态之间所有状态发送一个 Q\_EXIT\_SIG 消息，然后给公共父状态到目标状态之间的所有状态发一个 Q\_ENTRY\_SIG。

结论：前面用 UML 画的层次状态机用 QHSM 框架都能实现，而且把图翻译成代码也非常直观、方便。QHSM 框架结构好，效率高，可重入。强烈推荐使用。

讲解完毕，别忘了我们的重点是：用状态机原理进行软件“设计”。



## 第9章 附录

附件是前面提到的 CParser 和 Calc 的实现。



SampleCode.rar