C语言高效有限状态机(FSM)详细设计说明书

个人使用

180210V1.0

**编 写： 潘小飞 日期： 180310**

**校 对： 日期：**

**审 核： 日期：**

**批 准： 日期：**

目录

[C语言高效有限状态机(FSM)详细设计说明书 i](#_Toc508469219)

[目录 i](#_Toc508469220)

[文档说明 ii](#_Toc508469221)

[修改记录 ii](#_Toc508469222)

[1 引言 - 1 -](#_Toc508469223)

[1.1 编写目的 - 1 -](#_Toc508469224)

[1.2 项目背景 - 2 -](#_Toc508469225)

[1.3 定义 - 2 -](#_Toc508469226)

[1.4 参考资料 - 2 -](#_Toc508469227)

[2 总体设计 - 3 -](#_Toc508469228)

[2.1 需求概述 - 3 -](#_Toc508469229)

[2.2 软件结构 - 3 -](#_Toc508469230)

[2.3 设计资料 - 3 -](#_Toc508469231)

[3 程序描述 - 4 -](#_Toc508469232)

[3.1 通用数据类型定义 - 4 -](#_Toc508469233)

[3.2 功能描述 - 4 -](#_Toc508469234)

[3.3 状态机建立 - 5 -](#_Toc508469235)

[3.4 示例DEMO - 8 -](#_Toc508469236)

[3.5 总结 - 8 -](#_Toc508469237)

文档说明

1. 文档普通内容字体使用 宋体 五号。
2. 一级标题使用2号字体。
3. 二级及其他标题使用3号字体。

修改记录

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **修订人** | **修订日期** | **修订描述** |
| V1.0 | 潘小飞 | 180210 | 初次生成当前模板 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. 引言
   1. 编写目的

很多情况下，使用状态机跳转的思想进行编程可以把复杂问题进行最大限度的简化。摘录一些有关有限状态机的说明，如下。

有限状态机，（[英语](https://baike.baidu.com/item/%E8%8B%B1%E8%AF%AD)：Finite-state machine, FSM），又称[有限状态自动机](https://baike.baidu.com/item/%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E8%87%AA%E5%8A%A8%E6%9C%BA" \t "_blank)，简称状态机，是表示有限个[状态](https://baike.baidu.com/item/%E7%8A%B6%E6%80%81" \t "_blank)以及在这些状态之间的转移和动作等行为的[数学模型](https://baike.baidu.com/item/%E6%95%B0%E5%AD%A6%E6%A8%A1%E5%9E%8B)。

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。

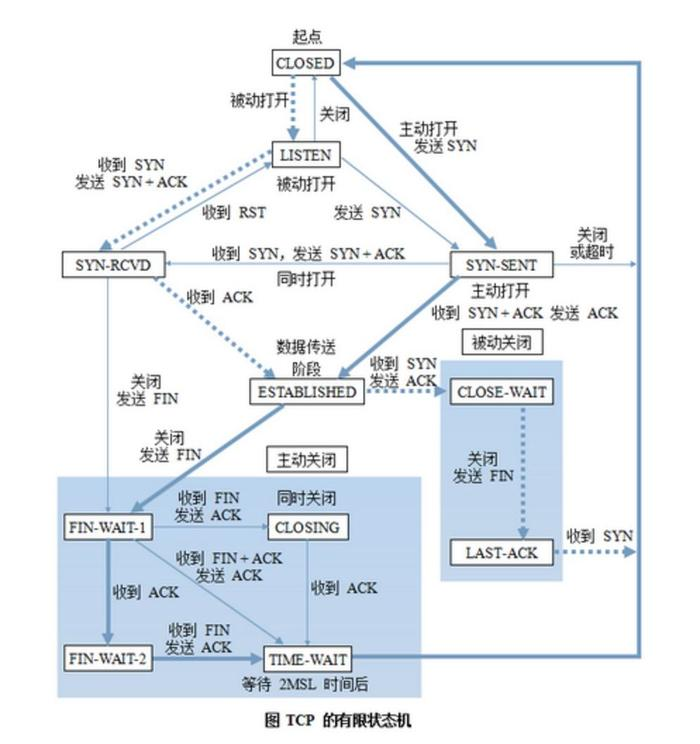


图1.1 TCP的有限状态机

在C语言开发环境下，进行状态机的设计常见的是switch-case/if-else这样的方式。为了找出一些更加简单的方法，以及更加高效的方式，对当前C语言开发环境下设计一套状态机开发宏。

* 1. 项目背景

暂无项目。

目前提供使用例程DEMO。

* 1. 定义

SD:state define状态定义

SADC:state action declaration状态动作声明

SADF:state action define状态动作定义

SMDC:state machine declaration状态机声明

SMDF:state machine define状态机定义

SMR:state machine run驱动状态机运行

* 1. 参考资料

1.很久以前在网上看到过的一个说明资料-- C语言最优状态机规范，目前好象删了：



1. 总体设计
   1. 需求概述

简化C语言编写状态机的过程，同时提高程序运行效率。

* 1. 软件结构



图2.1 状态机设计

说明：

1.所有状态都有枚举表示；

2.数组中存储状态操作函数；

3.枚举中存储的状态与状态操作函数数组下标对应；

4.状态操作函数根据当前状态输出到记录结构体确定下一状态；

* 1. 设计资料



1. 程序描述
   1. 通用数据类型定义

模块内部定义了以下通用数据类型。

表3.1 通用数据类型

|  |  |
| --- | --- |
| 名称 | 含义 |
| typedef unsigned char sta; | 状态机状态，目前最多可以有127个状态，对于一般项目应用来说已足够 |
| typedef void (\*staAct)(void \*hStaRec); | 状态操作函数 |
|  |  |
|  |  |

表3.2 状态及状态操作函数定义

|  |  |
| --- | --- |
| 名称 | 含义 |
| sm\_type##\_sta\_##sm\_staX | 状态机状态通用形式，其中  sm\_type为状态机名称  sm\_staX为状态机某个状态  \_sta\_为状态的识别关键字  示例，如MySm\_sta\_init |
| sm\_type##\_act\_##sm\_staX | 状态机状态操作函数通用形式，  sm\_type为状态机名称  sm\_staX为状态机某个状态  \_act\_为状态操作函数的识别关键字  示例，如MySm\_act\_init |
|  |  |
|  |  |

* 1. 功能描述

为实现C代码程序能够简单编程。有3个种类宏需要使用。

表3.3 功能宏定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 单词 | 含义 | 其他 |
| SD | state define | 状态定义 | 为X\_MACRO扩展提供方便的内部宏，对于用户来说不可见 |
| SADC | state action declaration | 状态动作声明 | 为X\_MACRO扩展提供方便的内部宏，对于用户来说不可见 |
| SADF | state action define | 状态动作定义 | 为X\_MACRO扩展提供方便的内部宏，对于用户来说不可见 |
|  |  |  |  |
| STA\_LIST | state list | 状态列表定义 | 此宏不在sm.h文件中定义，需用户根据不同项目而进行设计  不同的状态机需要定义为不同的x\_STA\_LIST，如A\_STA\_LIST,B\_STA\_LIST |
|  |  |  |  |
| SMDC | state machine declaration | 状态机声明 | 当前宏为用户声明状态机宏；  此宏主要是让用户声明一个记录状态宏，  typedef strcut StaRec  {  sta next;  ……  } \*hStaRec, StaRec; |
| SMDF | state machine define | 状态机定义 | 当前宏为用户定义状态机宏，内部有所有状态对应状态函数函数进行匹配的机制 |
| SMR | state machine run | 驱动状态机 | 当前宏为用户对状态机给入时钟功能宏，可以让状态机正常工作 |
|  |  |  |  |
|  |  |  |  |

注意：

对于用户来说可见宏就只有3个，SMDC/ SMDF/ SMR。但是对状态机进行声明和定义之后只进行了状态和状态操作函数的匹配，并没有把状态函数实体定义出来。实际的状态操作函数不能忘记进行定义。

* 1. 状态机建立

状态机的建立很简单，对于用户来说只要4个步骤就可以建立一个简单的状态机。最为关键的两个步骤是1.创建状态LIST和3.定义状态操作函数。

表3.4 状态机创建过程

|  |  |
| --- | --- |
| 内容项 | 详细说明 |
| 1．建立STA\_LIST | 用户需根据需求建立不同状态机的状态LIST，如下LIST  /\*  #define STA\_LIST(\_) \  \_(sm\_type, count) \  \_(sm\_type, done)  \*/  不同的状态机对应的LIST应取不同的名字； |
| 示例 | **#define** **MY\_SM\_LIST**(\_) \  \_(mySm, upCnt) \  \_(mySm, downCnt) \  \_(mySm, done) |
| 2．使用SMDC/SMDF声明和定义状态机 | 使用SMDC/SMDF对状态机进行声明和定义；  有两个注意事项：   1. 宏参数中的输入的状态机类型需要与STA\_LIST中定义的状态机名称一致 2. SMDC定义状态记录结构体中第一个参数需要是sta next;   其他，SMDC最好在h文件中使用，SMDF一定要在c文件中使用； |
| 示例 | **SMDC**(*mySm*, **MY\_SM\_LIST**)  {  sta next;  sta last;  int16 cnt;  };  **SMDF**(*mySm*, **MY\_SM\_LIST**); |
|  |  |
| 3．定义状态操作函数 | 状态操作函数需要按sm\_type##\_act\_##sm\_staX(void \*hStaRec)的形式进行定义；  除了STA\_LIST中各状态对应状态操作函数，还必须对定义两个默认的状态操作函数sm\_type##\_act\_init(void \*hStaRec)/ sm\_type##\_act\_default(void \*hStaRec)； |
| 示例 | **void** **mySm\_act\_init**(**void** \*smRec)  {  hmySmRec hRec = (hmySmRec)smRec;  hRec->last = *mySm\_sta\_init*;  hRec->cnt = 0;  hRec->next = *mySm\_sta\_upCnt*;  }  **void** **mySm\_act\_upCnt**(**void** \*smRec)  {  hmySmRec hRec = (hmySmRec)smRec;  hRec->last = *mySm\_sta\_upCnt*;  hRec->cnt++;  **if** (hRec->cnt >= 4)  {  hRec->next = *mySm\_sta\_downCnt*;  }  **else**  {  hRec->next = *mySm\_sta\_upCnt*;  }  }  **void** **mySm\_act\_downCnt**(**void** \*smRec)  {  hmySmRec hRec = (hmySmRec)smRec;  hRec->last = *mySm\_sta\_downCnt*;  hRec->cnt--;  **if** (hRec->cnt < 3)  {  hRec->next = *mySm\_sta\_done*;  }  **else**  {  hRec->next = *mySm\_sta\_downCnt*;  }  }  **void** **mySm\_act\_done**(**void** \*smRec)  {  hmySmRec hRec = (hmySmRec)smRec;  hRec->last = *mySm\_sta\_done*;  hRec->cnt--;  hRec->next = *mySm\_sta\_default*;  }  **void** **mySm\_act\_default**(**void** \*smRec)  {  hmySmRec hRec = (hmySmRec)smRec;  hRec->last = *mySm\_sta\_default*;  hRec->cnt = 0;  hRec->next = *mySm\_sta\_default*;  } |
|  |  |
| 4．SMR给入时钟驱动状态机 | 使用SMR直接驱动状态机即可 |
| 示例 | **for** (i = 0; i < 20; i++)  {  **SMR**(*mySm*);  } |
|  |  |
|  |  |

* 1. 示例DEMO





* 1. 总结

1. 应用方法

在拥有方法的基础上是如果快速使用起来，前期状态机都是手写，目前使用脚本生成的方式增加开发速度。

* 1. Python脚本代码生成

使用脚本可以大大简化开发过程，让自己的开发速度和效率上一个台阶。让一些重复性的工作给工具辅助完成。

* + 1. 脚本代码

脚本本身的代码量并不大，使用规则也很简单。Python一般都是指令的方式。

当前脚本指令为:python sm.py stateMachineName stateList

脚本至少需要3个参数，

sm.py是当前脚本名称

状态机名称stateMachineName根据需要选取

stateList是多个参数，目前配置状态机时至少需要给定1个状态



* + 1. 脚本使用示例

脚本参数：

python sm.py mysm examp next

会生成名为mysm的状态机，除去init/default两个状态，还有examp/next两个状态。

