C中的状态机设计

大卫·拉弗尼尔

2021 年 5 月 1 日 警察

评价我: 4.99/5 (47 票)

紧凑的 C 有限状态机 (FSM) 实现，易于在嵌入式和基于 PC 的系统上使用

本文基于“C++ 中的状态机设计”一文中提出的想法，提供了另一种 C 语言状态机实现。该设计适用于任何平台，嵌入式或 PC，具有任何 C 编译器。

下载源 - 55.3 KB

介绍

2000 年，我为 C/C++ 用户杂志 (RIP)写了一篇题为“ C++ 中的状态机设计”的文章。有趣的是，那篇旧文章仍然可用，并且（在撰写本文时）是搜索 C++ 状态机时在 Google 上排名第一的文章。这篇文章是 15 年前写的，但我继续在许多项目中使用基本思想。它结构紧凑，易于理解，并且在大多数情况下，具有足够的功能来完成我需要的功能。

有时，C 是完成这项工作的正确工具。本文基于“ C++ 中的状态机设计”一文中提出的思想，提供了另一种 C 语言状态机实现。该设计适用于任何平台，嵌入式或 PC，具有任何 C 编译器。该状态机具有以下特点：

- **C 语言**——用**C 语言**编写的状态机
- **紧凑**- 消耗最少的资源
- **对象**- 支持单个状态机类型的多个实例化
- **转换表**——转换表精确控制状态转换行为
- **事件**——每个事件都是一个带有任何参数类型的简单函数
- **状态动作**——如果需要，每个状态动作都是一个单独的函数，带有一个唯一的事件数据参数
- **Guards/entry/exit 动作**——可选的状态机可以为每个状态使用警戒条件和单独的进入/退出动作函数
- **宏**——可选的多行宏支持通过自动化代码“机器”来简化使用
- **错误检查**——编译时和运行时检查及早发现错误
- **线程安全**——添加软件锁使代码线程安全很容易

这篇文章不是关于软件状态机的最佳设计分解实践的教程。我将专注于状态机代码和简单的示例，它们的复杂性刚好足以帮助理解功能和用法。

背景

大多数程序员都掌握的一种常见设计技术是古老的有限状态机 (FSM)。设计人员使用这种编程结构将复杂问题分解为可管理的状态和状态转换。有无数种方法可以实现状态机。

一个`switch`语句提供一个最简单的实现和状态机的最常见的版本。在这里，`switch`语句中的每个 case 都变成了一个状态，实现如下：

C++ 复制代码

```
switch (currentState) {
   case ST_IDLE:
       // do something in the idle state
       break;

    case ST_STOP:
       // do something in the stop state
       break;

    // etc...
}
```

这种方法当然适用于解决许多不同的设计问题。然而，当在事件驱动的多线程项目中使用时，这种形式的状态机可能非常有限。

第一个问题围绕控制哪些状态转换是有效的，哪些是无效的。没有办法强制执行状态转换规则。任何时候都允许任何过渡，这不是特别可取的。对于大多数设计，只有少数过渡模式是有效的。理想情况下，软件设计应该强制执行这些预定义的状态序列并防止不需要的转换。尝试将数据发送到特定状态时会出现另一个问题。由于整个状态机位于单个函数中，因此向任何给定状态发送额外数据证明是困难的。最后，这些设计很少适用于多线程系统。设计人员必须确保从单个控制线程调用状态机。

为什么要使用状态机？

使用状态机实现代码是解决复杂工程问题的一种非常方便的设计技术。状态机将设计分解为一系列步骤，或者在状态机行话中称为状态。每个状态执行一些狭义的任务。另一方面，事件是刺激，它导致状态机在状态之间移动或转换。

举一个简单的例子，我将在整篇文章中使用它，假设我们正在设计电机控制软件。我们想要启动和停止电机，以及改变电机的速度。足够简单。暴露给客户端软件的电机控制事件如下：

1. **设置速度**- 设置电机以特定速度运行
2. **停止**- 停止电机

这些事件提供了以任何所需速度启动电机的能力，这也意味着改变已经移动的电机的速度。或者我们可以完全停止电机。对于电机控制模块，这两个事件或功能被视为外部事件。然而，对于使用我们代码的客户来说，这些只是普通的函数。

这些事件不是状态机状态。处理这两个事件所需的步骤是不同的。在这种情况下，状态是：

1. **空闲**— 电机未旋转但处于静止状态
    - 没做什么
2. **启动**— 从完全停止状态启动电机
    - 打开电机电源
    - 设置电机速度
3. **改变速度**— 调整已经移动的电机的速度
    - 改变电机速度
4. **停止**— 停止移动的电机
    - 关闭电机电源
    - 进入空闲状态

可以看出，将电机控制分解为谨慎的状态，而不是具有单一功能，我们可以更轻松地管理如何操作电机的规则。

每个状态机都有“当前状态”的概念。这是状态机当前所处的状态。在任何给定时刻，状态机只能处于单一状态。特定状态机实例的每个实例都可以在定义时设置初始状态。但是，该初始状态不会在对象创建期间执行。只有发送到状态机的事件才会导致状态函数执行。

为了以图形方式说明状态和事件，我们使用状态图。下面的图 1 显示了电机控制模块的状态转换。方框表示状态，连接箭头表示事件转换。列出事件名称的箭头是外部事件，而未修饰的线被视为内部事件。（我将在本文后面介绍内部事件和外部事件之间的差异。）

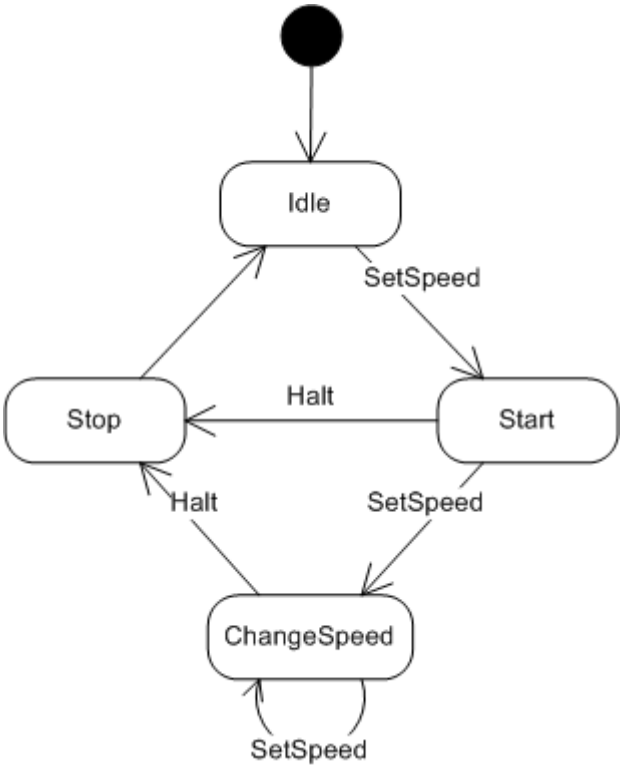

图 *1*：电机状态图

如您所见，当事件发生时，发生的状态转换取决于状态机的当前状态。`SetSpeed`例如，当一个事件发生时，电机处于该`Idle`状态，它会转换到该`Start`状态。但是，`SetSpeed`在当前状态`Start`转换时生成的相同事件将电机转换为该`ChangeSpeed`状态。您还可以看到并非所有状态转换都是有效的。例如，电机不能`ChangeSpeed`在`Idle`没有首先经过`Stop`状态的情况下从到转换。

简而言之，使用状态机捕获并强制执行复杂的交互，否则这些交互可能难以传达和实施。

内部和外部事件

正如我之前提到的，事件是导致状态机在状态之间转换的刺激。例如，按下按钮可能是一个事件。事件可以分为两类：外部和内部。在最基本的层面上，外部事件是对状态机模块的函数调用。这些函数是公共的，可以从外部调用，也可以从状态机对象的外部代码调用。系统内的任何线程或任务都可以生成外部事件。如果外部事件函数调用导致发生状态转换，则状态将在调用者的控制线程内同步执行。另一方面，内部事件在状态执行期间由状态机本身自行生成。

一个典型的场景包括生成一个外部事件，这又归结为对模块公共接口的函数调用。根据正在生成的事件和状态机的当前状态，执行查找以确定是否需要转换。如果是，则状态机转换到新状态并执行该状态的代码。在状态函数结束时，执行检查以确定是否生成了内部事件。如果是，则执行另一个转换并且新状态有机会执行。这个过程一直持续到状态机不再产生内部事件，此时原始外部事件函数调用返回。外部事件和所有内部事件（如果有）在调用者的控制线程内执行。

一旦外部事件开始执行状态机，如果使用锁，则在外部事件和所有内部事件完成执行之前，它不能被另一个外部事件中断。这个运行到完成模型为状态转换提供了一个多线程安全的环境。可以在状态机引擎中使用信号量或互斥锁来阻止可能试图同时访问同一状态机实例的其他线程。请参阅源代码函数`_SM_ExternalEvent()`注释以了解锁的去向。

事件数据

生成事件时，它可以选择性地附加事件数据以供状态函数在执行期间使用。事件数据是指向任何内置或用户定义数据类型的单个`const`或非`const` 指针。

一旦状态完成执行，事件数据就被认为用完了，必须删除。因此，发送到状态机的任何事件数据都必须通过`SM_XAlloc()`. 状态机引擎使用 自动释放分配的事件数据`SM_XFree()`。

状态转换

当生成外部事件时，将执行查找以确定状态转换动作过程。事件有三种可能的结果：新状态、事件被忽略或无法发生。新状态会导致转换到允许执行的新状态。也可以转换到现有状态，这意味着重新执行当前状态。对于被忽略的事件，不执行任何状态。但是，事件数据（如果有）将被删除。最后一种可能性，不可能发生，保留用于给定状态机的当前状态事件无效的情况。如果发生这种情况，则软件会出错。

在此实现中，不需要内部事件来执行验证转换查找。假设状态转换是有效的。您可以检查有效的内部和外部事件转换，但在实践中，这只会占用更多的存储空间并产生繁忙的工作，而收益甚微。验证转换的真正需要在于异步的外部事件，在这些事件中，客户端可能会导致事件在不适当的时间发生。状态机一旦执行，就不能被中断。它受私有实现的控制，因此不需要转换检查。这使设计人员可以通过内部事件自由更改状态，而无需更新转换表。

状态机模块

状态机源代码包含在*StateMachine.c*和*StateMachine.h*文件中。下面的代码显示了部分标题。该`StateMachine`标题包含各种预处理多宏来缓解执行的状态机。

C++ 缩小▲ 复制代码

```
enum { EVENT_IGNORED = 0xFE, CANNOT_HAPPEN = 0xFF };

typedef void NoEventData;

// State machine constant data
typedef struct
{
    const CHAR* name;
    const BYTE maxStates;
    const struct SM_StateStruct* stateMap;
    const struct SM_StateStructEx* stateMapEx;
} SM_StateMachineConst;

// State machine instance data
typedef struct
{
    const CHAR* name;
    void* pInstance;
    BYTE newState;
    BYTE currentState;
    BOOL eventGenerated;
    void* pEventData;
} SM_StateMachine;

// Generic state function signatures
typedef void (*SM_StateFunc)(SM_StateMachine* self, void* pEventData);
typedef BOOL (*SM_GuardFunc)(SM_StateMachine* self, void* pEventData);
typedef void (*SM_EntryFunc)(SM_StateMachine* self, void* pEventData);
typedef void (*SM_ExitFunc)(SM_StateMachine* self);

typedef struct SM_StateStruct
{
    SM_StateFunc pStateFunc;
} SM_StateStruct;

typedef struct SM_StateStructEx
{
    SM_StateFunc pStateFunc;
    SM_GuardFunc pGuardFunc;
    SM_EntryFunc pEntryFunc;
    SM_ExitFunc pExitFunc;
} SM_StateStructEx;

// Public functions
#define SM_Event(_smName_, _eventFunc_, _eventData_) \
    _eventFunc_(&_smName_##Obj, _eventData_)

// Protected functions
#define SM_InternalEvent(_newState_, _eventData_) \
```

```
    _SM_InternalEvent(self, _newState_, _eventData_)
#define SM_GetInstance(_instance_) \
    (_instance_*)(self->pInstance);

// Private functions
void _SM_ExternalEvent(SM_StateMachine* self,
     const SM_StateMachineConst* selfConst, BYTE newState, void* pEventData);
void _SM_InternalEvent(SM_StateMachine* self, BYTE newState, void* pEventData);
void _SM_StateEngine(SM_StateMachine* self, const SM_StateMachineConst* selfConst);
void _SM_StateEngineEx(SM_StateMachine* self, const SM_StateMachineConst* selfConst);

#define SM_DECLARE(_smName_) \
    extern SM_StateMachine _smName_##Obj;

#define SM_DEFINE(_smName_, _instance_) \
    SM_StateMachine _smName_##Obj = { #_smName_, _instance_, \
        0, 0, 0, 0 };

#define EVENT_DECLARE(_eventFunc_, _eventData_) \
    void _eventFunc_(SM_StateMachine* self, _eventData_* pEventData);

#define EVENT_DEFINE(_eventFunc_, _eventData_) \
    void _eventFunc_(SM_StateMachine* self, _eventData_* pEventData)

#define STATE_DECLARE(_stateFunc_, _eventData_) \
    static void ST_##_stateFunc_(SM_StateMachine* self, _eventData_* pEventData);

#define STATE_DEFINE(_stateFunc_, _eventData_) \
    static void ST_##_stateFunc_(SM_StateMachine* self, _eventData_* pEventData)
```

该SM_Event()宏用于生成外部事件，而SM_InternalEvent()在状态函数执行期间生成内部事件。SM_GetInstance()获取指向当前状态机对象的指针。

SM_DECLARE 和SM_DEFINE用于创建状态机实例。EVENT_DECLARE并EVENT_DEFINE创建外部事件函数。最后，STATE_DECLARE与STATE_DEFINE创建国家职能。

电机示例

Motor 实现我们假设的电机控制状态机，客户可以在其中以特定速度启动电机，然后停止电机。所述Motor插头接口如下所示：

C++ 复制代码

```
#include "StateMachine.h"

// Motor object structure
typedef struct
{
    INT currentSpeed;
} Motor;

// Event data structure
typedef struct
{
    INT speed;
} MotorData;

// State machine event functions
EVENT_DECLARE(MTR_SetSpeed, MotorData)
EVENT_DECLARE(MTR_Halt, NoEventData)
```

在Motor源文件通过隐藏所需的状态机机械使用宏来简化使用。

C++ 缩小▲ 复制代码

```
// State enumeration order must match the order of state
// method entries in the state map
enum States
{
    ST_IDLE,
    ST_STOP,
    ST_START,
    ST_CHANGE_SPEED,
    ST_MAX_STATES
};

// State machine state functions
STATE_DECLARE(Idle, NoEventData)
STATE_DECLARE(Stop, NoEventData)
STATE_DECLARE(Start, MotorData)
STATE_DECLARE(ChangeSpeed, MotorData)

// State map to define state function order
BEGIN_STATE_MAP(Motor)
    STATE_MAP_ENTRY(ST_Idle)
    STATE_MAP_ENTRY(ST_Stop)
    STATE_MAP_ENTRY(ST_Start)
    STATE_MAP_ENTRY(ST_ChangeSpeed)
END_STATE_MAP(Motor)

// Set motor speed external event
EVENT_DEFINE(MTR_SetSpeed, MotorData)
{
    // Given the SetSpeed event, transition to a new state based upon
    // the current state of the state machine
    BEGIN_TRANSITION_MAP                        // - Current State -
        TRANSITION_MAP_ENTRY(ST_START)          // ST_Idle
        TRANSITION_MAP_ENTRY(CANNOT_HAPPEN)     // ST_Stop
        TRANSITION_MAP_ENTRY(ST_CHANGE_SPEED)   // ST_Start
        TRANSITION_MAP_ENTRY(ST_CHANGE_SPEED)   // ST_ChangeSpeed
    END_TRANSITION_MAP(Motor, pEventData)
}

// Halt motor external event
EVENT_DEFINE(MTR_Halt, NoEventData)
{
    // Given the Halt event, transition to a new state based upon
    // the current state of the state machine
    BEGIN_TRANSITION_MAP                        // - Current State -
        TRANSITION_MAP_ENTRY(EVENT_IGNORED)     // ST_Idle
        TRANSITION_MAP_ENTRY(CANNOT_HAPPEN)     // ST_Stop
        TRANSITION_MAP_ENTRY(ST_STOP)           // ST_Start
        TRANSITION_MAP_ENTRY(ST_STOP)           // ST_ChangeSpeed
    END_TRANSITION_MAP(Motor, pEventData)
}
```

外部事件

MTR_SetSpeed 并且MTR_Halt被认为是进入Motor状态机的外部事件。MTR_SetSpeed 获取一个指向MotorData事件数据的指针，其中包含电机速度。该数据结构将SM_XFree()在状态处理完成后被释放，因此必须SM_XAlloc()在进行函数调用之前创建它。

状态枚举

每个状态函数都必须有一个与之关联的枚举。这些枚举用于存储状态机的当前状态。在 中Motor，States提供了这些枚举，稍后用于索引转换映射和状态映射查找表。

状态函数

状态函数实现每个状态——每个状态机状态一个状态函数。STATE_DECLARE 用于声明状态函数接口并STATE_DEFINE 定义实现。

C++ 缩小▲ 复制代码

```
// State machine sits here when motor is not running
STATE_DEFINE(Idle, NoEventData)
{
    printf("%s ST_Idle\n", self->name);
}

// Stop the motor
STATE_DEFINE(Stop, NoEventData)
{
    // Get pointer to the instance data and update currentSpeed
    Motor* pInstance = SM_GetInstance(Motor);
    pInstance->currentSpeed = 0;

    // Perform the stop motor processing here
    printf("%s ST_Stop: %d\n", self->name, pInstance->currentSpeed);

    // Transition to ST_Idle via an internal event
    SM_InternalEvent(ST_IDLE, NULL);
}

// Start the motor going
STATE_DEFINE(Start, MotorData)
{
    ASSERT_TRUE(pEventData);

    // Get pointer to the instance data and update currentSpeed
    Motor* pInstance = SM_GetInstance(Motor);
    pInstance->currentSpeed = pEventData->speed;

    // Set initial motor speed processing here
    printf("%s ST_Start: %d\n", self->name, pInstance->currentSpeed);
}

// Changes the motor speed once the motor is moving
STATE_DEFINE(ChangeSpeed, MotorData)
{
    ASSERT_TRUE(pEventData);

    // Get pointer to the instance data and update currentSpeed
    Motor* pInstance = SM_GetInstance(Motor);
    pInstance->currentSpeed = pEventData->speed;

    // Perform the change motor speed here
    printf("%s ST_ChangeSpeed: %d\n", self->name, pInstance->currentSpeed);
}
```

STATE_DECLARE并STATE_DEFINE使用两个参数。第一个参数是状态函数名称。第二个参数是事件数据类型。如果不需要事件数据，请使用NoEventData. 宏也可用于创建守卫、退出和进入操作，本文稍后将对此进行解释。

该SM_GetInstance()宏获取状态机对象的实例。宏的参数是状态机名称。

在这个实现中，所有状态机函数都必须遵守这些签名，如下所示：

C++ 复制代码

```
// Generic state function signatures
typedef void (*SM_StateFunc)(SM_StateMachine* self, void* pEventData);
typedef BOOL (*SM_GuardFunc)(SM_StateMachine* self, void* pEventData);
typedef void (*SM_EntryFunc)(SM_StateMachine* self, void* pEventData);
typedef void (*SM_ExitFunc)(SM_StateMachine* self);
```

每个都SM_StateFunc 接受一个指向SM_StateMachine对象和事件数据的指针。如果NoEventData 使用，则 pEventData 参数将为NULL。否则，pEventData参数是 中指定的类型STATE_DEFINE。

在Motor的Start状态函数中，STATE_DEFINE(Start, MotorData) 宏扩展为：

C++ 复制代码

```
void ST_Start(SM_StateMachine* self, MotorData* pEventData)
```

请注意，每个状态函数都有self 和pEventData 参数。self 是指向状态机对象的指针，pEventData 是事件数据。另请注意，宏ST_在状态名称前加上“ ”以创建函数ST_Start()。

类似地，Stop 状态函数STATE_DEFINE(Stop, NoEventData)扩展为：

C++ 复制代码

```
void ST_Stop(SM_StateMachine* self, void* pEventData)
```

Stop 不接受事件数据，所以pEventData 参数是void*.

三个字符会自动添加到宏中的每个状态/守卫/进入/退出功能。例如，如果使用STATE_DEFINE(Idle, NoEventData)实际状态函数名称声明一个函数被调用ST_Idle()。

1. ST_ - 状态函数前置字符
2. GD_ - 保护功能前置字符
3. EN_ - 输入函数前置字符
4. EX_ - 退出函数前置字符

SM_GuardFunc 和SM_Entry functiontypedef也接受事件数据。SM_ExitFunc 独特之处在于不允许事件数据。

州地图

状态机引擎通过使用状态图知道要调用哪个状态函数。状态图将currentState变量映射到特定的状态函数。例如，如果currentState 是2，则将调用第三个状态映射函数指针条目（从零开始计数）。状态映射表是使用这三个宏创建的：

C++ 复制代码

```
BEGIN_STATE_MAP
STATE_MAP_ENTRY
END_STATE_MAP
```

BEGIN_STATE_MAP 启动状态映射序列。每个STATE_MAP_ENTRY 都有一个状态函数名称参数。END_STATE_MAP 终止地图。状态图Motor 如下所示：

C++ 复制代码

```
BEGIN_STATE_MAP(Motor)
    STATE_MAP_ENTRY(ST_Idle)
    STATE_MAP_ENTRY(ST_Stop)
    STATE_MAP_ENTRY(ST_Start)
    STATE_MAP_ENTRY(ST_ChangeSpeed)
END_STATE_MAP
```

或者，守卫/进入/退出功能需要使用_EX宏的（扩展）版本。

C++ 复制代码

```
BEGIN_STATE_MAP_EX
STATE_MAP_ENTRY_EX or STATE_MAP_ENTRY_ALL_EX
END_STATE_MAP_EX
```

该STATE_MAP_ENTRY_ALL_EX 宏有四个参数，依次用于状态动作、保护条件、进入动作和退出动作。状态动作是强制性的，但其他动作是可选的。如果状态没有动作，则0用于参数。如果一个状态没有任何守卫/进入/退出选项，STATE_MAP_ENTRY_EX 宏会将所有未使用的选项默认为 0。下面的宏片段是本文后面介绍的高级示例。

C++ 复制代码

```
// State map to define state function order
BEGIN_STATE_MAP_EX(CentrifugeTest)
    STATE_MAP_ENTRY_ALL_EX(ST_Idle, 0, EN_Idle, 0)
    STATE_MAP_ENTRY_EX(ST_Completed)
    STATE_MAP_ENTRY_EX(ST_Failed)
    STATE_MAP_ENTRY_ALL_EX(ST_StartTest, GD_StartTest, 0, 0)
    STATE_MAP_ENTRY_EX(ST_Acceleration)
    STATE_MAP_ENTRY_ALL_EX(ST_WaitForAcceleration, 0, 0, EX_WaitForAcceleration)
    STATE_MAP_ENTRY_EX(ST_Deceleration)
    STATE_MAP_ENTRY_ALL_EX(ST_WaitForDeceleration, 0, 0, EX_WaitForDeceleration)
END_STATE_MAP_EX(CentrifugeTest)
```

不要忘记添加预谋字符（ST_，GD_，EN_或EX_每个功能）。

状态机对象

在 C++ 中，对象是语言不可或缺的一部分。使用 C，您必须更加努力地工作才能完成类似的行为。此 C 语言状态机支持多个状态机对象（或实例），而不是具有单个静态状态机实现。

所述SM_StateMachine 数据结构存储状态机实例的数据; 每个状态机实例一个对象。的SM_StateMachineConst 数据结构存储的常量数据; 每个状态机类型一个常量对象。

状态机是使用SM_DEFINE 宏定义的。第一个参数是状态机名称。第二个参数是指向用户定义状态机结构的指针，或者NULL 如果没有用户对象。

C++ 复制代码

```
#define SM_DEFINE(_smName_, _instance_) \
    SM_StateMachine _smName_##Obj = { #_smName_, _instance_, \
        0, 0, 0, 0 };
```

在这个例子中，状态机名称是Motor，并且创建了两个对象和两个状态机。

C++ 复制代码

```
// Define motor objects
static Motor motorObj1;
static Motor motorObj2;

// Define two public Motor state machine instances
SM_DEFINE(Motor1SM, &motorObj1)
SM_DEFINE(Motor2SM, &motorObj2)
```

每个电机对象独立于其他对象处理状态执行。该Motor 结构用于存储状态机实例特定的数据。在状态函数中，用于在运行时SM_GetInstance()获取指向Motor 对象的指针。

C++ 复制代码

```
// Get pointer to the instance data and update currentSpeed
Motor* pInstance = SM_GetInstance(Motor);
pInstance->currentSpeed = pEventData->speed;
```

过渡图

最后要注意的细节是状态转换规则。状态机如何知道应该发生什么转换？答案是过渡图。转换映射是将currentState 变量映射到状态枚举常量的查找表。每个外部事件函数都有一个使用三个宏创建的转换映射表：

C++ 复制代码

```
BEGIN_TRANSITION_MAP
TRANSITION_MAP_ENTRY
END_TRANSITION_MAP
```

的MTR_Halt 在事件函数Motor 定义的过渡地图为：

C++ 复制代码

```
// Halt motor external event
EVENT_DEFINE(MTR_Halt, NoEventData)
{
    // Given the Halt event, transition to a new state based upon
    // the current state of the state machine
    BEGIN_TRANSITION_MAP                        // - Current State -
        TRANSITION_MAP_ENTRY(EVENT_IGNORED)     // ST_Idle
        TRANSITION_MAP_ENTRY(CANNOT_HAPPEN)     // ST_Stop
        TRANSITION_MAP_ENTRY(ST_STOP)           // ST_Start
        TRANSITION_MAP_ENTRY(ST_STOP)           // ST_ChangeSpeed
    END_TRANSITION_MAP(Motor, pEventData)
}
```

BEGIN_TRANSITION_MAP开始地图。后面的每一个都TRANSITION_MAP_ENTRY表示状态机应该根据当前状态做什么。每个转换映射表中的条目数必须与状态函数数完全匹配。在我们的示例中，我们有四个状态函数，因此我们需要四个转换映射条目。每个条目的位置与状态映射中定义的状态函数的顺序相匹配。因此，MTR_Halt函数中的第一个条目表示EVENT_IGNORED 如下所示：

C++ 复制代码

```
TRANSITION_MAP_ENTRY (EVENT_IGNORED)    // ST_Idle
```

这被解释为“如果在当前状态为空闲状态时发生 Halt 事件，则忽略该事件。”

同样，地图中的第三个条目是：

复制代码

```
TRANSITION_MAP_ENTRY (ST_STOP)          // ST_Start
```

这表示“如果在当前状态为 Start 时发生 Halt 事件，则转换到状态 Stop”。

END_TRANSITION_MAP 终止地图。此宏的第一个参数是状态机名称。第二个参数是事件数据。

在C_ASSERT()宏内使用END_TRANSITION_MAP。如果状态机状态数与转换映射条目数不匹配，则会生成编译时错误。

新状态机步骤

创建一个新的状态机需要几个基本的高级步骤：

1. 创建一个States 枚举，每个状态函数一个条目
2. 定义状态函数
3. 定义事件函数
4. 使用STATE_MAP宏创建一个状态映射查找表
5. 使用TRANSITION_MAP 宏为每个外部事件函数创建一个转换图查找表

状态引擎

状态引擎根据生成的事件执行状态函数。转换图是SM_StateStruct由currentState 变量索引的实例数组。当_SM_StateEngine()函数执行时，它会在SM_StateStruct 数组中查找正确的状态函数。在状态函数有机会执行后，它会释放事件数据（如果有），然后再检查是否通过SM_InternalEvent().

C++ 缩小▲ 复制代码

```
// The state engine executes the state machine states
void _SM_StateEngine(SM_StateMachine* self, SM_StateMachineConst* selfConst)
{
    void* pDataTemp = NULL;
```

```
    ASSERT_TRUE(self);
    ASSERT_TRUE(selfConst);

    // While events are being generated keep executing states
    while (self->eventGenerated)
    {
        // Error check that the new state is valid before proceeding
        ASSERT_TRUE(self->newState < selfConst->maxStates);

        // Get the pointers from the state map
        SM_StateFunc state = selfConst->stateMap[self->newState].pStateFunc;

        // Copy of event data pointer
        pDataTemp = self->pEventData;

        // Event data used up, reset the pointer
        self->pEventData = NULL;

        // Event used up, reset the flag
        self->eventGenerated = FALSE;

        // Switch to the new current state
        self->currentState = self->newState;

        // Execute the state action passing in event data
        ASSERT_TRUE(state != NULL);
        state(self, pDataTemp);

        // If event data was used, then delete it
        if (pDataTemp)
        {
            SM_XFree(pDataTemp);
            pDataTemp = NULL;
        }
    }
}
```

守卫、进入、状态和退出动作的状态引擎逻辑由以下序列表示。该`_SM_StateEngine()`引擎实现了仅＃1和＃以下5。扩展`_SM_StateEngineEx()`引擎使用整个逻辑序列。

1. 评估状态转换表。如果`EVENT_IGNORED`，则忽略该事件并且不执行转换。如果`CANNOT_HAPPEN`，则软件出现故障。否则，继续下一步。
2. 如果定义了保护条件，则执行保护条件功能。如果保护条件返回`FALSE`，则忽略状态转换并且不调用状态函数。如果守卫返回`TRUE`，或者没有守卫条件存在，状态函数将被执行。
3. 如果转换到新状态并且为当前状态定义了退出操作，则调用当前状态退出操作函数。
4. 如果转换到新状态并且为新状态定义了进入动作，则调用新状态进入动作函数。
5. 为新状态调用状态动作函数。新状态现在是当前状态。

生成事件

在这一点上，我们有一个工作状态机。让我们看看如何为它生成事件。通过使用 动态创建事件数据结构`SM_XAlloc()`，分配结构成员变量，并使用`SM_Event()`宏调用外部事件函数来生成外部事件。以下代码片段显示了如何进行同步调用。

C++ 复制代码

```
MotorData* data;

// Create event data
data = SM_XAlloc(sizeof(MotorData));
data->speed = 100;

// Call MTR_SetSpeed event function to start motor
SM_Event(Motor1SM, MTR_SetSpeed, data);
```

第`SM_Event()`一个参数是状态机名称。第二个参数是要调用的事件函数。第三个参数是事件数据，`NULL` 如果没有数据。

要从状态函数中生成内部事件，请调用`SM_InternalEvent()`。如果目标不接受事件数据，则最后一个参数是`NULL`。否则，使用创建事件数据`SM_XAlloc()`。

C++ 复制代码

```
SM_InternalEvent(ST_IDLE, NULL);
```

在上面的例子中，一旦状态函数完成执行，状态机就会转换到`ST_Idle`状态。另一方面，如果需要将事件数据发送到目标状态，则需要在堆上创建数据结构并作为参数传入。

C++ 复制代码

```
MotorData* data;
data = SM_XAlloc(sizeof(MotorData));
data->speed = 100;
SM_InternalEvent(ST_CHANGE_SPEED, data);
```

无堆使用

所有状态机事件数据都必须动态创建。但是，在某些系统上，使用堆是不可取的。包含的`x_allocator`模块是一个固定的块内存分配器，可消除堆使用。`USE_SM_ALLOCATOR` 在*StateMachine.c* 中定义以使用固定块分配器。有关信息，请参阅下面的**参考资料**部分`x_allocator`。

离心机测试示例

该`CentrifugeTest` 示例显示了如何使用保护、进入和退出操作创建扩展状态机。状态图如下所示：

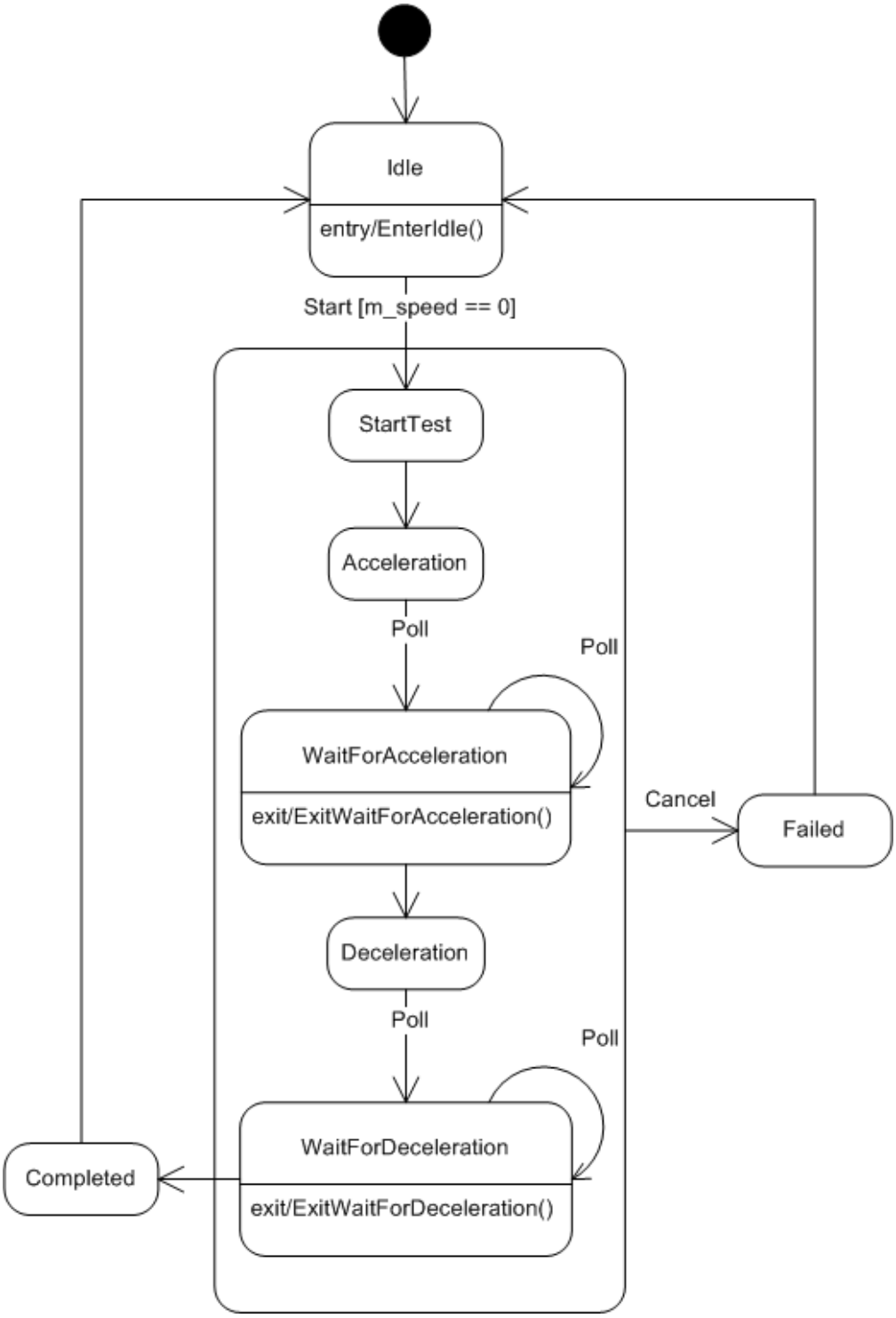

图 2：*CentrifugeTest* 状态图

甲`CentrifgeTest` 对象和状态机被创建。这里唯一的区别是状态机是一个单例，这意味着对象是`private`并且只能`CentrifugeTest` 创建一个实例。这与`Motor` 允许多个实例的状态机不同。

C++ 复制代码

```
// CentrifugeTest object structure
typedef struct
{
    INT speed;
    BOOL pollActive;
} CentrifugeTest;

// Define private instance of motor state machine
CentrifugeTest centrifugeTestObj;
SM_DEFINE(CentrifugeTestSM, &centrifugeTestObj)
```

扩展状态机使用ENTRY_DECLARE,GUARD_DECLARE和EXIT_DECLARE 宏。

C++ 缩小▲ 复制代码

```
// State enumeration order must match the order of state
// method entries in the state map
enum States
{
    ST_IDLE,
    ST_COMPLETED,
    ST_FAILED,
    ST_START_TEST,
    ST_ACCELERATION,
    ST_WAIT_FOR_ACCELERATION,
    ST_DECELERATION,
    ST_WAIT_FOR_DECELERATION,
    ST_MAX_STATES
};

// State machine state functions
STATE_DECLARE(Idle, NoEventData)
ENTRY_DECLARE(Idle, NoEventData)
STATE_DECLARE(Completed, NoEventData)
STATE_DECLARE(Failed, NoEventData)
STATE_DECLARE(StartTest, NoEventData)
GUARD_DECLARE(StartTest, NoEventData)
STATE_DECLARE(Acceleration, NoEventData)
STATE_DECLARE(WaitForAcceleration, NoEventData)
EXIT_DECLARE(WaitForAcceleration)
STATE_DECLARE(Deceleration, NoEventData)
STATE_DECLARE(WaitForDeceleration, NoEventData)
EXIT_DECLARE(WaitForDeceleration)

// State map to define state function order
BEGIN_STATE_MAP_EX(CentrifugeTest)
    STATE_MAP_ENTRY_ALL_EX(ST_Idle, 0, EN_Idle, 0)
    STATE_MAP_ENTRY_EX(ST_Completed)
    STATE_MAP_ENTRY_EX(ST_Failed)
    STATE_MAP_ENTRY_ALL_EX(ST_StartTest, GD_StartTest, 0, 0)
    STATE_MAP_ENTRY_EX(ST_Acceleration)
    STATE_MAP_ENTRY_ALL_EX(ST_WaitForAcceleration, 0, 0, EX_WaitForAcceleration)
    STATE_MAP_ENTRY_EX(ST_Deceleration)
    STATE_MAP_ENTRY_ALL_EX(ST_WaitForDeceleration, 0, 0, EX_WaitForDeceleration)
END_STATE_MAP_EX(CentrifugeTest)
```

请注意_EX扩展状态映射宏，因此支持守卫/进入/退出功能。每个守卫/进入/退出DECLARE 宏必须与DEFINE. 例如，StartTest 状态函数的保护条件声明为：

C++ 复制代码

```
GUARD_DECLARE(StartTest, NoEventData)
```

TRUE 如果要执行状态函数或FALSE 以其他方式执行，则保护条件函数返回。

C++ 复制代码

```
// Guard condition to determine whether StartTest state is executed.
GUARD_DEFINE(StartTest, NoEventData)
{
    printf("%s GD_StartTest\n", self->name);
    if (centrifugeTestObj.speed == 0)
        return TRUE;    // Centrifuge stopped. OK to start test.
    else
        return FALSE;   // Centrifuge spinning. Can't start test.
}
```

多线程安全

为了防止状态机在执行过程中被另一个线程抢占，`StateMachine` 模块可以在`_SM_ExternalEvent()`函数内使用锁。在允许执行外部事件之前，可以锁定信号量。当外部事件和所有内部事件都被处理后，软件锁被释放，允许另一个外部事件进入状态机实例。

如果应用程序是多线程的并且多线程能够访问单个状态机实例，则注释指示应该将锁定和解锁放在何处。请注意，每个 `StateMachine` 对象都应该有自己的软件锁实例。这可以防止单个实例锁定并防止所有其他`StateMachine` 对象执行。仅当一个`StateMachine` 实例被多个控制线程调用时才需要软件锁。如果不是，则不需要锁。

结论

使用这种方法而不是旧的`switch`语句样式来实现状态机似乎需要额外的努力。然而，回报是更健壮的设计，能够在整个多线程系统上统一使用。将每个状态放在自己的函数中比单个大`switch`语句更容易阅读，并允许将唯一的事件数据发送到每个状态。此外，验证状态转换通过消除不需要的状态转换引起的副作用来防止客户端误用。

这个 C 语言版本是我多年来在不同项目中使用的 C++ 实现的密切翻译。如果使用 C++，请考虑**参考**部分中的 C++ 实现。

参考

- C++ 中的状态机设计- David Lafreniere
- C 语言中的固定块分配器- David Lafreniere

历史

- 第二2日，2019：首次发布
- 2021 年 5 月1日：添加了 getter 功能并修复了编译器警告。更新了源代码。

执照

分享

关于作者

大卫·拉弗尼尔

美国

手表
该会员

我做专业软件工程师已经超过 20 年了。不编写代码时，我喜欢与家人共度时光，在南加州露营和骑摩托车。

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

First Prev Next

Not using dynamically alocated eventdata
Member 15302992 19-Aug-21 17:02

My vote of 5
David A. Gray 4-May-21 14:12

Why event functions?
Member 15062728 5-Feb-21 20:15

A question about sequential operation within a state.
Andre_E_S 29-Sep-20 1:38

A more flexible design?
rrotstein 7-Sep-20 6:55

Kudos and a question
ganjula 25-Aug-20 15:35

Re: Kudos and a question
David Lafreniere 27-Aug-20 6:57

My vote of 5
DickC17 25-Aug-20 4:08

Implementation of getSpeed function and lock/unlock motor
luc ruyven 6-Jul-20 16:03

Re: Implementation of getSpeed function and lock/unlock motor
David Lafreniere 14-Jul-20 22:43

My vote of 5
Gynar 3-Jul-20 14:28

Re: My vote of 5
David Lafreniere 5-Jul-20 23:53

variable "uname" was set but never used
Member 11645292 26-Feb-20 0:40

Re: variable "uname" was set but never used
David Lafreniere 16-Mar-20 4:16

Translation
WVS 18-Oct-19 9:58

Re: Translation
David Lafreniere 18-Oct-19 21:20

Re: Translation
WVS 19-Oct-19 10:23

I got compiler warnings
Marcel Wagner 8-Aug-19 7:00

Re: I got compiler warnings
David Lafreniere 24-Aug-19 21:39

Re: I got compiler warnings
Member 12478329 22-Dec-19 13:11

Re: I got compiler warnings
David Lafreniere 23-Dec-19 22:03

Re: I got compiler warnings
Robeeeeeeeee 29-Jun-21 18:19

Re: I got compiler warnings
David Lafreniere 29-Jun-21 20:26

Re: I got compiler warnings
Robeeeeeeeee 2-Jul-21 13:23

Excellent work!
Mike Hankey 18-Mar-19 23:07

Refresh 1 2 Next ▷

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.