15,051,064 名会员

第362

CODE PROJECT For those who code

文章 问答 论坛 东西 休息室 ?

Search for articles, questions,

手表

C++中的状态机设计

大卫·拉弗尼尔

2019 年 2 月 21 日 警察

评价我: 4.91/5 (135 票)

一种紧凑的 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

下载源 (StateMachine) - 22.5 KB

下载源 (StateMachineCompact) - 4.5 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）搜索 C++ 状态机时在 Google 上排名第一。这篇文章写于 15 多年前，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。

本文提供了一个具有更新功能的新实现，但代价是增加了一点代码。我还将更正原始实现中的错误，因为如果您的存储空间非常紧张，它仍然是一个可行的解决方案。这两种设计都是表格驱动的，适用于任何平台、嵌入式或 PC，以及任何 C++ 编译器。

为什么是另一种状态机设计？当然，到目前为止，有一个可以使用的现有实现，对吗？也许。有时，我会尝试一个新的状态机，但由于某种原因，我发现它不适合我的需求。对我来说，问题通常归结为以下一项或多项：

1. **太大**——最终的实现需要太多的代码空间来证明在嵌入式平台上是合理的
2. **太复杂**——模板过多或需要采用完整的框架才能使用状态机
3. **无编译器支持**– 依赖于编译器不支持的新 C++ 语言功能
4. **高学习曲线**- 有些需要在这方面付出很多努力
5. 难懂的**语法**——非直观的状态机表达式，难以诊断编译器错误
6. **外部库**——依赖于增加大小或平台不支持的外部库
7. **功能太多**——不需要完全符合 UML，因此超出了手头问题的基本需求
8. **没有事件数据**——不能将唯一的事件数据发送到状态函数
9. **中央事件处理程序**——单个事件处理函数和一个`switch`语句处理每个类的事件
10. **无类型安全**– 需要根据枚举手动类型转换事件数据
11. **有限的转换规则**- 不支持事件被忽略和不能发生的事件转换
12. **没有线程安全**——代码不是线程安全的

不要误会我的意思，有些实现非常令人印象深刻并且适用于许多不同的项目。每一种设计都有一定的权衡，这个也不例外。只有您可以决定这是否满足您的需求。我将尝试通过本文和示例代码尽快引导您。该状态机具有以下特点：

1. **紧凑**–`StateMachine`该类不是模板 – 在 Windows 上只有 448 字节的代码。模板很少使用。
2. **转换表**——转换表精确控制状态转换行为。
3. **事件**——每个事件都是一个简单的公共实例成员函数，带有任何参数类型。
4. **状态动作**——如果需要，每个状态动作都是一个单独的实例成员函数，具有单个、唯一的事件数据参数。
5. **Guards/entry/exit action** – 状态机可以选择为每个状态使用保护条件和单独的进入/退出动作函数。
6. **状态机继承**- 支持从基本状态机类继承状态。

7. **状态函数继承**——支持在派生类中覆盖状态函数。
8. **宏**– 可选的多行宏支持通过自动化代码“机器”来简化使用。
9. **类型安全**- 编译时检查及早发现错误。运行时检查其他情况。
10. **线程安全**——添加软件锁使代码线程安全很容易。

这种状态机设计并不是要实现完整的 UML 功能集。它也不是分层状态机 (HSM)。相反，它的目标是相对紧凑、便携且易于使用的传统有限状态机 (FSM)，具有足够的独特功能来解决许多不同的问题。

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。

有关 此状态机的 C 语言实现，请参阅C 语言中的状态机设计一文。

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。对于电机控制类，这两个事件或函数被视为外部事件。然而，对于使用我们代码的客户来说，这些只是类中的普通函数。

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。特定状态机类的每个实例都可以在构造期间设置初始状态。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 显示了电机控制类的状态转换。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。）

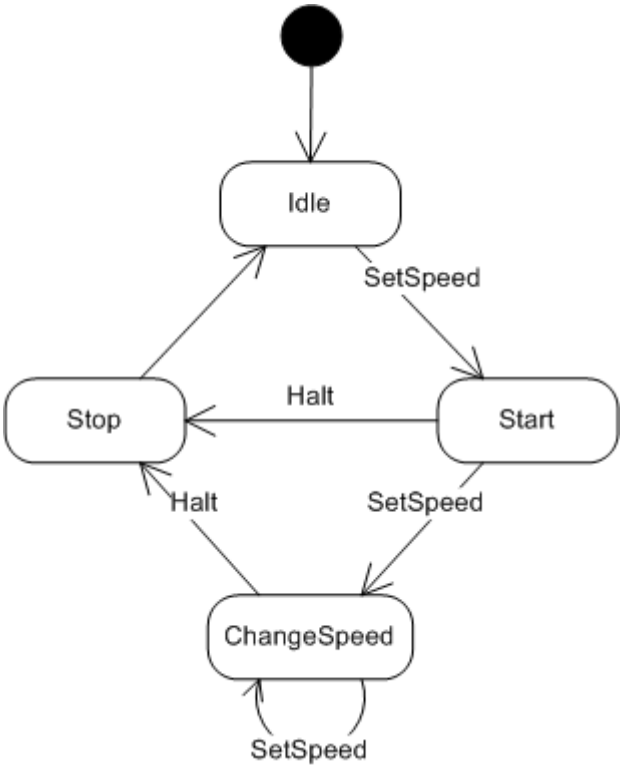

图 *1*：电机状态图

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。`Event`s 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对象的函数调用。这些函数是`public`从外部或从状态机对象外部的代码调用的。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，这又归结为对类`public`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`state`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对象的其他线程。请参阅源代码函数`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。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操作符 new 在堆上创建，以便状态机可以在使用后将其删除。此外，对于我们的特定实现，事件数据必须从`EventData` 基类继承。这为状态机引擎提供了一个公共基类，用于删除所有事件数据。

C++ 复制代码

```
class EventData
{
public:
    virtual ~EventData() {}
};
```

状态机实现现在有一个构建选项，可以消除在堆上创建外部事件数据的要求。有关详细信息，请参阅**外部事件无堆数据**部分。

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。它在类的`private`实现的控制之下，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。

状态机类

创建自己的状态机时需要两个基类：`StateMachine` 和`EventData`. 一个类继承自`StateMachine` 以获得必要的机制来支持状态转换和事件处理。该`StateMachine` 头还包含各种预处理多宏安心的实施状态机（在本文后面介绍）。要将唯一数据发送到状态函数，该结构必须从`EventData` 基类继承。

状态机源代码包含在*StateMachine.cpp*和*StateMachine.h*文件中（请参阅随附的*StateMachine.zip*）。下面的代码显示了类声明。

C++ 缩小▲ 复制代码

```
class StateMachine
{
public:
    enum { EVENT_IGNORED = 0xFE, CANNOT_HAPPEN };

    StateMachine(BYTE maxStates, BYTE initialState = 0);
    virtual ~StateMachine() {}

    BYTE GetCurrentState() { return m_currentState; }

protected:
    void ExternalEvent(BYTE newState, const EventData* pData = NULL);
    void InternalEvent(BYTE newState, const EventData* pData = NULL);

private:
    const BYTE MAX_STATES;
    BYTE m_currentState;
    BYTE m_newState;
```

```
    BOOL m_eventGenerated;
    const EventData* m_pEventData;

    virtual const StateMapRow* GetStateMap() = 0;
    virtual const StateMapRowEx* GetStateMapEx() = 0;

    void SetCurrentState(BYTE newState) { m_currentState = newState; }

    void StateEngine(void);
    void StateEngine(const StateMapRow* const pStateMap);
    void StateEngine(const StateMapRowEx* const pStateMapEx);
};
```

StateMachine 是用于处理事件和状态转换的基类。该接口包含在四个函数中：

C++ 复制代码

```
void ExternalEvent(BYTE newState, const EventData* pData = NULL);
void InternalEvent(BYTE newState, const EventData* pData = NULL);
virtual const StateMapRow* GetStateMap() = 0;
virtual const StateMapRowEx* GetStateMapEx() = 0;
```

ExternalEvent()使用新状态和指向EventData对象的指针（如果有）作为参数为状态机生成外部事件。该 InternalEvent() 函数使用相同的参数集生成内部事件。

的GetStateMap() 和GetStateMapEx() 函数返回的数组StateMapRow 或StateMapRowEx 实例将由当所述状态引擎适当进行检索。继承类必须返回具有这些函数之一的数组。如果状态机只有状态函数，GetStateMap()则使用。如果需要警卫/进入/退出功能，则GetStateMapEx() i使用 s。另一个未使用的版本必须返回NULL。但是，提供了多行宏来为我们实现这些功能，我将在稍后演示。

电机示例

Motor 和MotorNM 类是如何使用StateMachine. MotorNM （无宏）完全匹配Motor设计而不依赖宏。这允许查看所有宏扩展代码以便于理解。然而，一旦达到速度，我发现宏通过隐藏所需的源机器极大地简化了使用。

MotorNM 下面显示的类声明不包含宏：

C++ 缩小▲ 复制代码

```
class MotorNMData : public EventData
{
public:
    INT speed;
};

// Motor class with no macros
class MotorNM : public StateMachine
{
public:
    MotorNM();

    // External events taken by this state machine
    void SetSpeed(MotorNMData* data);
    void Halt();

private:
    INT m_currentSpeed;

    // 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
    };

    // Define the state machine state functions with event data type
    void ST_Idle(const NoEventData*);
    StateAction<MotorNM, NoEventData, &MotorNM::ST_Idle> Idle;

    void ST_Stop(const NoEventData*);
    StateAction<MotorNM, NoEventData, &MotorNM::ST_Stop> Stop;

    void ST_Start(const MotorNMData*);
    StateAction<MotorNM, MotorNMData, &MotorNM::ST_Start> Start;

    void ST_ChangeSpeed(const MotorNMData*);
    StateAction<MotorNM, MotorNMData, &MotorNM::ST_ChangeSpeed> ChangeSpeed;

    // State map to define state object order. Each state map entry defines a
    // state object.
private:
    virtual const StateMapRowEx* GetStateMapEx() { return NULL; }
    virtual const StateMapRow* GetStateMap() {
        static const StateMapRow STATE_MAP[] = {
            &Idle,
            &Stop,
            &Start,
            &ChangeSpeed,
        };
        C_ASSERT((sizeof(STATE_MAP)/sizeof(StateMapRow)) == ST_MAX_STATES);
        return &STATE_MAP[0]; }
};
```

本Motor 类使用宏进行比较：

C++ 缩小▲ 复制代码

```
class MotorData : public EventData
{
public:
    INT speed;
};

// Motor class using macro support
class Motor : public StateMachine
{
public:
    Motor();

    // External events taken by this state machine
    void SetSpeed(MotorData* data);
    void Halt();

private:
    INT m_currentSpeed;

    // 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
    };

    // Define the state machine state functions with event data type
    STATE_DECLARE(Motor,     Idle,            NoEventData)
    STATE_DECLARE(Motor,     Stop,            NoEventData)
```

```
    STATE_DECLARE(Motor,     Start,           MotorData)
    STATE_DECLARE(Motor,     ChangeSpeed,     MotorData)

    // State map to define state object order. Each state map entry defines a
    // state object.
    BEGIN_STATE_MAP
        STATE_MAP_ENTRY(&Idle)
        STATE_MAP_ENTRY(&Stop)
        STATE_MAP_ENTRY(&Start)
        STATE_MAP_ENTRY(&ChangeSpeed)
    END_STATE_MAP
};
```

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。该 SetSpeed()和Halt() public功能被认为是外部事件进入Motor 状态机。SetSpeed()获取包含电机速度的事件数据。该数据结构将在状态处理完成后被删除，因此必须在进行函数调用之前继承EventData 并创建该结构operator new 。

当Motor 被创建的类，它的初始状态是Idle。第一次调用SetSpeed() 将状态机转换为Start电机最初启动的状态。随后的SetSpeed()事件转换到ChangeSpeed状态，在该状态中，已经在移动的电机的速度被调整。所述Halt() 事件转换到Stop，其中，状态执行期间，产生以过渡回空闲状态的内部事件。

创建一个新的状态机需要几个基本的高级步骤：

1. 从StateMachine 基类继承
2. 创建一个States 枚举，每个状态函数一个条目
3. 使用STATE 宏创建状态函数
4. 可选择使用GUARD,ENTRY 和EXIT 宏为每个状态创建保护/进入/退出功能
5. 使用STATE_MAP 宏创建一个状态映射查找表
6. 使用TRANSITION_MAP宏为每个外部事件创建一个转换图查找表

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。在这个实现中，所有状态函数都必须遵守这个状态函数签名，如下所示：

C++ 复制代码

```
void <class>::<func>(const EventData*)
```

<class>并且<func>不是模板参数，而只是分别用于特定类和函数名称的占位符。例如，您可以选择诸如 void Motor::ST_Start(const MotorData*). 这里重要的是该函数不返回任何数据（具有void 返回类型）并且它具有一个类型的输入参数EventData*（或其派生类）。

state用STATE_DECLARE 宏声明函数。宏参数是状态机类名、状态函数名和事件数据类型。

C++ 复制代码

```
STATE_DECLARE(Motor,     Idle,            NoEventData)
STATE_DECLARE(Motor,     Stop,            NoEventData)
STATE_DECLARE(Motor,     Start,           MotorData)
STATE_DECLARE(Motor,     ChangeSpeed,     MotorData)
```

扩展上面的宏产生：

C++ 复制代码

```
void ST_Idle(const NoEventData*);
StateAction<Motor, NoEventData, &Motor::ST_Idle> Idle;

void ST_Stop(const NoEventData*);
StateAction<Motor, NoEventData, &Motor::ST_Stop> Stop;

void ST_Start(const MotorData*);
StateAction<MotorNM, MotorData, &Motor::ST_Start> Start;
```

```
void ST_ChangeSpeed(const MotorData*);
StateAction<Motor, MotorData, &Motor::ST_ChangeSpeed> ChangeSpeed;
```

请注意，多行宏在每个状态函数名称前添加了“ST_”。三个字符被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(Motor, Idle, NoEventData)`实际状态函数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

一个后`state`声明函数时，定义`state`与功能实现`STATE_DEFINE` 宏。参数是状态机类名、状态函数名和事件数据类型。实现状态行为的代码位于`state`函数内部。注意，任何`state`功能代码都可以调用 `InternalEvent()`切换到另一种状态。Guard/entry/exit 函数不能调用，`InternalEvent()` 否则会导致运行时错误。

C++ 复制代码

```
STATE_DEFINE(Motor, Stop, NoEventData)
{
    cout << "Motor::ST_Stop" << endl;
    m_currentSpeed = 0;

    // perform the stop motor processing here
    // transition to Idle via an internal event
    InternalEvent(ST_IDLE);
}
```

扩展宏产生这个状态函数定义。

C++ 复制代码

```
void Motor::ST_Stop(const NoEventData* data)
{
    cout << "Motor::ST_Stop" << endl;
    m_currentSpeed = 0;

    // perform the stop motor processing here
    // transition to Idle via an internal event
    InternalEvent(ST_IDLE);
}
```

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。

C++ 复制代码

```
enum States
{
    ST_IDLE,
    ST_STOP,
    ST_START,
    ST_CHANGE_SPEED,
    ST_MAX_STATES
};
```

枚举顺序与状态映射中提供的顺序匹配是很重要的。这样，状态枚举与特定的状态函数调用相关联。`EVENT_IGNORED`和`CANNOT_HAPPEN` 是与这些状态枚举结合使用的另外两个常量。`EVENT_IGNORED` 告诉状态引擎不要执行任何状态，只是返回并且什么都不做。`CANNOT_HAPPEN` 告诉状态引擎故障。这种异常的灾难性故障情况永远不会发生。

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m_currentState变量映射到特定的状态函数。例如，如果m_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
    STATE_MAP_ENTRY(&Idle)
    STATE_MAP_ENTRY(&Stop)
    STATE_MAP_ENTRY(&Start)
    STATE_MAP_ENTRY(&ChangeSpeed)
END_STATE_MAP
```

完成的状态映射只是实现了基类中定义的纯virtual函数。现在基类可以通过这个调用获取所有对象。宏展开后的代码如下所示：GetStateMap()StateMachine StateMachine StateMapRow

C++ 复制代码

```
private:
    virtual const StateMapRowEx* GetStateMapEx() { return NULL; }
    virtual const StateMapRow* GetStateMap() {
        static const StateMapRow STATE_MAP[] = {
            &Idle,
            &Stop,
            &Start,
            &ChangeSpeed,
        };
        C_ASSERT((sizeof(STATE_MAP)/sizeof(StateMapRow)) == ST_MAX_STATES);
        return &STATE_MAP[0]; }
```

注意C_ASSERT 宏。它针对具有错误条目数的状态映射提供编译时保护。Visual Studio 给出错误“ error C2118: negative subscript”。您的编译器可能会给出不同的错误消息。

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。如果状态没有任何保护/进入/退出选项，则STATE_MAP_ENTRY_EX 宏将所有未使用的选项默认为0. 下面的宏片段用于本文后面介绍的高级示例。

C++ 复制代码

```
BEGIN_STATE_MAP_EX
    STATE_MAP_ENTRY_ALL_EX(&Idle, 0, &EntryIdle, 0)
    STATE_MAP_ENTRY_EX(&Completed)
    STATE_MAP_ENTRY_EX(&Failed)
    STATE_MAP_ENTRY_ALL_EX(&StartTest, &GuardStartTest, 0, 0)
    STATE_MAP_ENTRY_EX(&Acceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForAcceleration, 0, 0, &ExitWaitForAcceleration)
    STATE_MAP_ENTRY_EX(&Deceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForDeceleration, 0, 0, &ExitWaitForDeceleration)
END_STATE_MAP_EX
```

转换图中的每个条目都是一个StateMapRow:

C++ 复制代码

```
struct StateMapRow
{
    const StateBase* const State;
};
```

该StateBase 指针拥有纯正virtual的叫接口StateEngine()。

C++ 复制代码

```
class StateBase
{
public:
    virtual void InvokeStateAction(StateMachine* sm, const EventData* data) const = 0;
};
```

该StateAction 派生自StateBase 其唯一的责任是落实InvokeStateAction()和铸StateMachine 和 EventData 指向正确的派生类的类型，然后调用状态成员函数。因此，调用每个状态函数的状态引擎开销是一个虚函数调用，一static_cast<> 加一dynamic_cast<>。

C++ 复制代码

```
template <class SM, class Data, void (SM::*Func)(const Data*)>
class StateAction : public StateBase
{
public:
    virtual void InvokeStateAction(StateMachine* sm, const EventData* data) const
    {
        // Downcast the state machine and event data to the correct derived type
        SM* derivedSM = static_cast<SM*>(sm);

        // Dynamic cast the data to the correct derived type
        const Data* derivedData = dynamic_cast<const Data*>(data);
        ASSERT_TRUE(derivedData != NULL);

        // Call the state function
        (derivedSM->*Func)(derivedData);
    }
};
```

的模板参数StateAction<>是状态机类 (SM)、事件数据类型 (Data) 和指向状态函数的成员函数指针 (Func)。

GuardCondition<>,EntryAction<>并且 ExitAction<>类也存在并且它们的作用是相同的——类型转换状态机和事件数据然后调用动作成员函数。模板参数存在细微的变化。在GuardCondition<> 类Func 模板参数稍有变化，并返回BOOL。ExitAction<>没有Data 模板参数。

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m_currentState 变量映射到状态枚举常量的查找表。每个外部事件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Halt()在事件函数Motor 定义的过渡地图为:

C++ 复制代码

```
void Motor::Halt()
{
    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_CHANGE_SPEED
    END_TRANSITION_MAP(NULL)
}
```

的宏扩展代码`Halt()` 如下。再次注意`C_ASSERT`提供编译时保护的宏，以防止转换映射条目的数量不正确。

C++ 复制代码

```
void Motor::Halt()
{
    static const BYTE TRANSITIONS[] = {
        EVENT_IGNORED,                  // ST_IDLE
        CANNOT_HAPPEN,                  // ST_STOP
        ST_STOP,                        // ST_START
        ST_STOP,                        // ST_CHANGE_SPEED
    };
    ExternalEvent(TRANSITIONS[GetCurrentState()], NULL);
    C_ASSERT((sizeof(TRANSITIONS)/sizeof(BYTE)) == ST_MAX_STATES);
}
```

`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，`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：

C++ 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。此结束宏的参数是事件数据（如果有）。`Halt()`没有事件数据所以参数是`NULL`，但`ChangeSpeed()`有数据所以它在这里传递。

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`StateMapRow`由`m_currentState` 变量索引的实例数组。当`StateEngine()`函数执行时，它通过调用or来查找一个`StateMapRow` or`StateMapRowEx` 数组：`GetStateMap()GetStateMapEx()`

C++ 复制代码

```
void StateMachine::StateEngine(void)
{
    const StateMapRow* pStateMap = GetStateMap();
    if (pStateMap != NULL)
        StateEngine(pStateMap);
    else
    {
        const StateMapRowEx* pStateMapEx = GetStateMapEx();
        if (pStateMapEx != NULL)
```

```
            StateEngine(pStateMapEx);
        else
            ASSERT();
    }
}
```

`StateMapRow` 使用新的状态值索引到表中，状态函数通过调用来执行`InvokeStateAction()`：

C++ 复制代码

```
const StateBase* state = pStateMap[m_newState].State;
state->InvokeStateAction(this, pDataTemp);
```

在状态函数有机会执行后，它会删除事件数据（如果有），然后再检查是否生成了任何内部事件。一个完整的状态引擎功能如下所示。另一个重载状态引擎函数（请参阅附加的源代码）使用`StateMapRowEx` 包含附加保护/进入/退出功能的表来处理状态机。

C++ 缩小▲ 复制代码

```
void StateMachine::StateEngine(const StateMapRow* const pStateMap)
{
    const EventData* pDataTemp = NULL;

    // While events are being generated keep executing states
    while (m_eventGenerated)
    {
        // Error check that the new state is valid before proceeding
        ASSERT_TRUE(m_newState < MAX_STATES);

        // Get the pointer from the state map
        const StateBase* state = pStateMap[m_newState].State;

        // Copy of event data pointer
        pDataTemp = m_pEventData;

        // Event data used up, reset the pointer
        m_pEventData = NULL;

        // Event used up, reset the flag
        m_eventGenerated = FALSE;

        // Switch to the new current state
        SetCurrentState(m_newState);

        // Execute the state action passing in event data
        ASSERT_TRUE(state != NULL);
        state->InvokeStateAction(this, pDataTemp);

        // If event data was used, then delete it
        if (pDataTemp)
        {
             delete 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`StateMapRow`引擎实现了仅＃1和＃以下5。扩展`StateMapRowEx`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，则执行保护条件函数。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new在堆上创建事件数据结构，分配结构成员变量，并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 = new MotorData();
data->speed = 50;
motor.SetSpeed(data);
```

要从状态函数中生成内部事件，请调用`InternalEvent()`。如果目标不接受事件数据，则仅使用您要转换到的状态调用该函数：

C++ 复制代码

```
InternalEvent(ST_IDLE);
```

在上面的例子中，一旦状态函数完成执行，状态机将转换到空闲状态。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 = new MotorData();
data->speed = 100;
InternalEvent(ST_CHANGE_SPEED, data);
```

外部事件无堆数据

状态机具有`EXTERNAL_EVENT_NO_HEAP_DATA`可更改`ExternalEvent()`. 定义时，只需在堆栈上传入数据，而不是在堆上创建外部事件数据，如下所示。此选项使调用者不必记住动态创建事件数据结构。

C++ 复制代码

```
MotorData data;
data.speed = 100;
motor.SetSpeed(&data);
```

该选项不会影响内部事件。`InternalEvent()`数据（如果有）仍必须在堆上创建。

C++ 复制代码

```
MotorData* data = new MotorData();
data->speed = 100;
InternalEvent(ST_CHANGE_SPEED, data);
```

隐藏和消除堆使用

该`SetSpeed()`函数采用`MotorData` 客户端必须在堆上创建的参数。或者，该类可以对调用者隐藏堆使用情况。更改就像`MotorData` 在`SetSpeed()`函数中创建实例一样简单。这样，调用者不需要创建动态实例：

C++ 复制代码

```
void Motor::SetSpeed(INT speed)
{
    MotorData* data = new MotorData;
    pData->speed = speed;

    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_CHANGE_SPEED
    END_TRANSITION_MAP(data)
}
```

或者，使用EXTERNAL_EVENT_NO_HEAP_DATA构建选项，调用者不需要在堆上创建外部事件数据。

在某些系统上，使用堆是不可取的。对于这些系统，我xallocator 在附加的源代码中包含了固定块分配器。它是可选的，但在使用时它会从静态内存或先前回收的堆内存中创建内存块。基类XALLOCATOR 中的单个宏EventData为所有EventData 类和派生类提供固定块分配。

C++ 复制代码

```
#include "xallocator.h"
class EventData
{
public:
    virtual ~EventData() {}
    XALLOCATOR
};
```

有关 xallocator 的更多信息，请参阅文章“**用快速固定块内存分配器替换 malloc/free** ”。

状态机继承

继承状态允许公共状态驻留在基类中，以便与继承的类共享。StateMachine 以最少的努力支持状态机继承。我会用一个例子来说明。

某些系统具有内置的自检模式，其中软件执行一系列测试步骤以确定系统是否正常运行。在这个例子中，每个自检必须处理常见的状态Idle，Completed和Failed状态。使用继承，基类包含共享状态。类SelfTest和CentrifugeTest 演示概念。状态图如下所示。

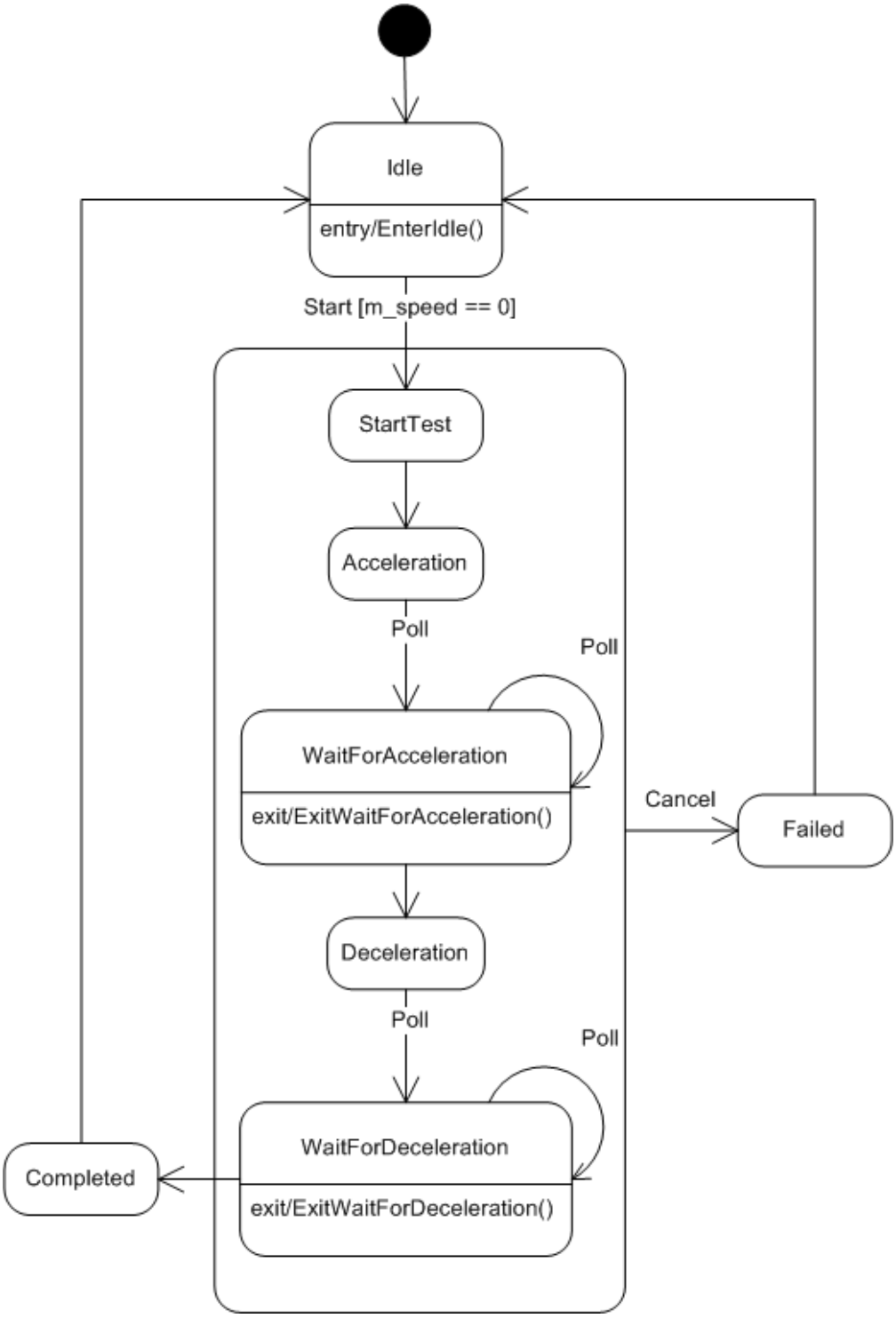

图 2：*CentrifugeTest* 状态图

`SelfTest` 定义以下状态：

0. `Idle`
1. `Completed`
2. `Failed`

`CentrifugeTest` 继承这些状态并创建新的状态：

3. 开始测试
4. 加速
5. 等待加速
6. 减速
7. 等待减速

所述`SeftTest` 基类定义的`States` 枚举和状态而不是状态图。只有层次结构中派生程度最高的状态机类才定义状态映射。

```
enum States
{
    ST_IDLE,
    ST_COMPLETED,
    ST_FAILED,
    ST_MAX_STATES
};

// Define the state machine states
STATE_DECLARE(SelfTest,     Idle,           NoEventData)
ENTRY_DECLARE(SelfTest,     EntryIdle,      NoEventData)
STATE_DECLARE(SelfTest,     Completed,      NoEventData)
STATE_DECLARE(SelfTest,     Failed,         NoEventData)
```

在CentrifugeTest 类定义低于其状态机。请注意“ ST_START_TEST = SelfTest::ST_MAX_STATES”枚举条目。派生类继续编号基类停止的状态是至关重要的。

C++ 缩小▲ 复制代码

```
enum States
{
    // Continue state numbering using the last SelfTest::States enum value
    ST_START_TEST = SelfTest::ST_MAX_STATES,
    ST_ACCELERATION,
    ST_WAIT_FOR_ACCELERATION,
    ST_DECELERATION,
    ST_WAIT_FOR_DECELERATION,
    ST_MAX_STATES
};

// Define the state machine state functions with event data type
STATE_DECLARE(CentrifugeTest,     Idle,                         NoEventData)
STATE_DECLARE(CentrifugeTest,     StartTest,                    NoEventData)
GUARD_DECLARE(CentrifugeTest,     GuardStartTest,               NoEventData)
STATE_DECLARE(CentrifugeTest,     Acceleration,                 NoEventData)
STATE_DECLARE(CentrifugeTest,     WaitForAcceleration,          NoEventData)
EXIT_DECLARE(CentrifugeTest,      ExitWaitForAcceleration)
STATE_DECLARE(CentrifugeTest,     Deceleration,                 NoEventData)
STATE_DECLARE(CentrifugeTest,     WaitForDeceleration,          NoEventData)
EXIT_DECLARE(CentrifugeTest,      ExitWaitForDeceleration)

// State map to define state object order. Each state map entry defines a
// state object.
BEGIN_STATE_MAP_EX
    STATE_MAP_ENTRY_ALL_EX(&Idle, 0, &EntryIdle, 0)
    STATE_MAP_ENTRY_EX(&Completed)
    STATE_MAP_ENTRY_EX(&Failed)
    STATE_MAP_ENTRY_ALL_EX(&StartTest, &GuardStartTest, 0, 0)
    STATE_MAP_ENTRY_EX(&Acceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForAcceleration, 0, 0, &ExitWaitForAcceleration)
    STATE_MAP_ENTRY_EX(&Deceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForDeceleration, 0, 0, &ExitWaitForDeceleration)
END_STATE_MAP_EX
```

状态映射包含层次结构中所有状态的条目，在本例中为SelfTest 和CentrifugeTest。请注意_EX扩展状态映射宏的使用，以便支持守卫/进入/退出功能。例如， 的保护条件StartState声明为：

C++ 复制代码

```
GUARD_DECLARE(CentrifugeTest, GuardStartTest, 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_DEFINE(CentrifugeTest, GuardStartTest, NoEventData)
{
    cout << "CentrifugeTest::GuardStartTest" << endl;
    if (m_speed == 0)
        return TRUE;    // Centrifuge stopped. OK to start test.
    else
        return FALSE;   // Centrifuge spinning. Can't start test.
}
```

基类外部事件函数

在使用状态机继承时，不仅仅是派生最多的类可以定义外部事件函数。基类和中间类也可以使用转换映射。父状态机不知道子类。可能存在一个或多个子类状态机分层级别，每个级别都添加更多状态。因此，最衍生状态机的父类使用部分转换图，即状态转换图只处理父类已知的状态。当前状态超出最大父状态枚举范围时生成的任何事件都必须由PARENT_TRANSITION 直接放置在转换图上方的宏处理，如下所示。

C++ 复制代码

```
void SelfTest::Cancel()
{
    PARENT_TRANSITION (ST_FAILED)

    BEGIN_TRANSITION_MAP                                // - Current State -
        TRANSITION_MAP_ENTRY (EVENT_IGNORED)            // ST_IDLE
        TRANSITION_MAP_ENTRY (CANNOT_HAPPEN)            // ST_COMPLETED
        TRANSITION_MAP_ENTRY (CANNOT_HAPPEN)            // ST_FAILED
    END_TRANSITION_MAP(NULL)
}
```

PARENT_TRANSITION 上面的例子读作"如果Cancel事件是在状态机不处于ST_IDLE,时生成ST_COMPLETE，或者ST_FAILED 然后转换到ST_FAILED 状态"。如果当前状态超出了当前状态机层次结构级别的已知范围，则转换映射之前的宏是包罗万象的。PARENT_TRANSITION 上面例子的扩展宏如下所示。

C++ 复制代码

```
if (GetCurrentState() >= ST_MAX_STATES &&
    GetCurrentState() < GetMaxStates()) {
     ExternalEvent(ST_FAILED);
     return; }
```

GetCurrentState()返回状态机当前状态。ST_MAX_STATES 是最大的父类State 枚举值。GetMaxStates()返回整个状态机的最大状态值，直到最派生的类。展开后的代码表明，如果当前状态高于最大父类状态值（即SelfTest::ST_MAX_STATES）但小于整个状态机最大状态值（即CentrifugeTest::ST_MAX_STATES），则转换到状态Failed。这样，如果在父部分过渡映射处理的范围之外，则统一处理过渡。否则，部分转换映射处理状态转换。EVENT_IGNORED 并且CANNOT_HAPPEN 也是有效的PARENT_TRANSITION论据。

假设父SelfTest::Cancel()类对于某些子类是不可接受的。只需覆盖或创建外部事件函数virtual 并在派生Cancel()函数中使用完整的转换图即可。

作为部分过渡映射的替代方案，父类可以在没有任何宏支持的情况下手动生成外部事件。要在父类层次结构级别内具有外部事件函数，请使用GetCurrentState()和ExternalEvent()。该SelfTest 班有一个外部事件功能：Cancel()。如以下代码所示，如果当前状态不是 Idle，状态机将转换为 Failed 状态。

C++ 复制代码

```
void SelfTest::Cancel()
{
    if (GetCurrentState() != ST_IDLE)
        ExternalEvent(ST_FAILED);
}
```

这里表达的状态机继承技术没有实现分层状态机 (HSM)。HSM 具有不同的语义和行为，其中包括分层事件处理模型。此处解释的此功能通过将公共状态分解为基类来提供代码重用，但状态机仍被视为传统的 FSM。

状态函数继承

状态函数可以在派生类中被覆盖。如果需要，派生类可以调用基本实现。在这种情况下，`SelfTest` 声明并定义一个空闲状态：

C++ 复制代码

```
STATE_DECLARE(SelfTest, Idle, NoEventData)

STATE_DEFINE(SelfTest,  Idle, NoEventData)
{
    cout << "SelfTest::ST_Idle" << endl;
}
```

`CentrifugeTest` 还声明并定义了具有相同事件数据类型的相同空闲状态。

C++ 复制代码

```
STATE_DECLARE(CentrifugeTest,  Idle, NoEventData)

STATE_DEFINE(CentrifugeTest,   Idle, NoEventData)
{
    cout << "CentrifugeTest::ST_Idle" << endl;

    // Call base class Idle state
    SelfTest::ST_Idle(data);
    StopPoll();
}
```

该`CentrifugeTest::ST_Idle()` 函数调用基本实现`SelfTest::ST_Idle()`。状态函数继承是一种强大的机制，允许层次结构中的每个级别处理相同的事件。

与覆盖状态函数的方式相同，派生类也可以覆盖保护/进入/退出函数。在覆盖中，您可以根据具体情况决定是否调用基本实现。

状态机紧凑类

如果`StateMachine` 由于`virtual`函数和类型转换导致实现太大或太慢，则可以使用紧凑版本，但不会对成员函数指针进行类型检查。压缩版本只有 68 字节（在 Windows 发行版上），而非压缩版本则为 448 字节。有关源文件，请参阅 *StateMachineCompact.zip*。

请注意，我们必须`reinterpret_cast<>`在`STATE_MAP_ENTRY` 宏中使用运算符将派生类成员函数指针转换为 `StateMachine` 成员函数指针。

C++ 复制代码

```
reinterpret_cast<StateFunc>(stateFunc)
```

由于`StateMachine`基类不知道派生类是什么，因此有必要执行此向上转换。因此，提供给的条目必须`STATE_MAP_ENTRY` 是继承类的真正成员函数，并且它们符合前面讨论的状态函数签名（参见状态函数部分）。否则，会发生不好的事情。

在大多数项目中，我不计算用于状态执行的 CPU 指令，并且一些额外的存储字节并不重要。我项目的状态机部分从来都不是瓶颈。所以我更喜欢非压缩版本的增强错误检查。

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` 该类可以在`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。

注释指示如果应用程序是多线程的，则应将锁定和解锁放在何处。如果使用锁，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。请注意，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。

有关使用此处提供的状态机的完整多线程示例，请参阅文章“**带有线程的 C++ 状态机**”。

备择方案

有时，您需要更强大的功能来使用事件和状态来捕获系统的行为。此处介绍的版本是传统 FSM 的变体。另一方面，真正的分层状态机 (HSM) 可以显着简化某些类型问题的解决方案。许多优秀的 HSM 项目已准备好供您探索。但有时您只需要一个简单的 FSM。

好处

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巨大的 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。

我已将此代码的变体用于自测引擎、手势识别库、用户界面向导和机器自动化以及其他项目。此实现为继承类提供了简单的使用。使用宏，它可以让您“转动曲柄”，而无需过多考虑状态引擎如何运行的基本机制。这让您有更多时间专注于更重要的事情，例如状态转换和状态函数实现的设计。

参考

- **C 语言中的状态机设计**- David Lafreniere
- **带线程的 C++ 状态机**- David Lafreniere
- **带有异步多播委托的 C++ 状态机**- David Lafreniere
- **用快速固定块内存分配器替换 malloc/free** - 作者 David Lafreniere
- **一个高效的 C++ 固定块内存分配器**- David Lafreniere

历史

- 23届三月2016
  - 首次发布
- 28个月2016
  - 更新附加的*StateMachine.zip*以帮助移植
- 3次月2016
  - 更正了图 1 中的错误
- 17个月2016
  - 更新了附加的*StateMachine.zip*源代码以包含 xallocator 错误修复
- 26个月2016
  - 在固定的预处理错误`STATE_DECLARE`内*StateMachine.zip*
  - 在固定宏错误`END_STATE_MAP`内*StateMachineCompact.zip*
  - 小修复使移植代码到其他平台更容易
  - 更新了附带的*StateMachine.zip*和*StateMachineCompact.zip*源代码并修复了错误
- 19个月2016
  - 添加了**参考**部分
- 16个月，2017年
  - 添加了`EXTERNAL_EVENT_NO_HEAP_DATA`构建选项

- 更新文章并附上*StateMachine.zip*源代码

- 17 个月，2017年
  - 添加了部分过渡贴图功能
  - 更新文章并附上 StateMachine.zip 源代码。
- 2019 年 2 月 20 日
  - 添加了对 C 语言状态机文章的引用。
  - 小文章更正。
  - 更新的源代码包含**xallocator.cpp 中的**次要修复。

执照

本文以及任何相关的源代码和文件均根据The Code Project Open License (CPOL)获得许可

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

第一 页上一页 下一页

HSM
Member 14905114 5-Aug-20 16:47

回复：HSM
David Lafreniere 27-Aug-20 7:03

它比状态设计模式好在哪里?
Manoj Sharma 2-Aug-20 22:57

Re: 它比状态设计模式好在哪里?
David Lafreniere 27-Aug-20 7:07

退出函数没有事件数据参数是否有原因?
会员 14661233 20-Nov-19 23:00

回复：退出函数没有事件数据参数是否有原因?
David Lafreniere 12-12-19 20:57

更易读的过渡图
andreasgass 14-May-19 16:25

状态到字符串宏
andreasgass 14-May-19 16:10

回复：状态到字符串宏
David Lafreniere 24-Aug-19 22:58

五颗星
Manuela Arcuri 15-Mar-19 17:26

回复：五颗星
David Lafreniere 21-Mar-19 10:32

给这个人打5分
Mike Hankey 8-Mar-19 22:21

回复：给这个人打5分
David Lafreniere 21-Mar-19 10:33

我的5票
Slobos 20-Dec-18 2:06

回复：我的5票
David Lafreniere 21-Feb-19 7:16

xallocator.h 中的错别字
Member 14032673 26-Oct-18 17:22

回复：xallocator.h 中的错字
David Lafreniere 18 年 11 月 22 日 20:19

内部事件上的新“NoEventData 或 EventData”对象
会员 13628461 20-Aug-18 23:23

回复：内部事件上的新“NoEventData 或 EventData”对象
David Lafreniere 23-Sep-18 23:05

回复：内部事件上的新“NoEventData 或 EventData”对象
David Lafreniere 21-Feb-19 7:16

为什么使用 static_cast 而不是 dynamic_cast?
AlexYIN 21-Jul-18 22:20

回复：为什么使用 static_cast 而不是 dynamic_cast?
David Lafreniere 18-Sep-18 1:38

嗨，

已经有一段时间了，但我相信原因是这样的：使用 static_cast<> 是因为它更快并且转换总是会成功，这意味着状态机的构造方式没有机会 static_cast<>可能会转换为错误的类型。因此，在这种情况下没有理由使用较慢的 dynamic_cast<>。另一方面，如果有人在事件数据上犯了错误，最好通过 dynamic_cast 上的断言来捕获错误。在这种情况下，dynamic_cast<> 通过在运行时捕获错误使代码更安全。

我希望这有帮助。

回复 · 电子邮件 · 查看主题

重新分配状态机时崩溃
Bùi Quang Minh **17-Jul-18 0:28**

回复：当我重新分配状态机时崩溃
加里安德里奥塔基斯 9-Nov-18 4:00

回复：当我重新分配状态机时崩溃
cmeyer42 16-Apr-19 19:25

刷新 1 2 3 4 下一页 ▷

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。

永久链接
广告
隐私
Cookie
使用条款

布局：固定 | 体液

文章版权所有 2016 年 David Lafreniere
所有其他版权 © CodeProject，

1999-2021 Web01 2.8.20210930.1