Boost MSM Basic

MSM stand for Meta State Machine

用例代码

- 使用GMock GTest框架，通过UT方式展现Boost MSM的功能
- 每一个UT Case文件包含一个Boost MSM状态机，使用GTest框架，构建多个Case，用于演示同一状态机的不同特征。
- 为了减少重复代码和简化状态机的代码，用例中使用的State，Action，Event，Guard都定义在Includes目录下，并用namespace隔开，比如

```
namespace Event
{
struct Event {};
} // namespace Event
```

- 每一个关键函数都使用TraceLib的Trace函数，打印对应的字符串。在UT Case中使用gMock对Trace函数打桩，在Case中检验Trace函数的输入值，用于演示状态机运行时的函数调用流程。

用例代码目录结构

```
├── Basic                                  // 包含主要用例
│   ├── AnonymousTransitTest.cpp
│   ├── ...Test.cpp
│   ├── CMakeLists.txt
│   ├── Includes                           // 用例所使用到的公共部分
│   │   ├── Action.hpp
│   │   ├── Event.hpp
│   │   ├── Guard.hpp
│   │   └── State.hpp
│   └── TracerLib                          // Trace的接口，Mocker用于UT测试
│       ├── CMakeLists.txt
│       ├── TracerMocker.cpp
│       └── TracerMocker.hpp
└── Doc                                    // 文档
    └── Boost State Machine.md
```

Functor

- 重载了（）operator的Class或struct
- Fuctor主要用于代替C语言中的回调函数
  - 更易于从用
  - 每个Functor实例可以有自己的状态
  - 更易于编译器优化代码可执行速度（可将Functor优化成inline函数）
- Boost::MSM 推荐使用Functor用于定义Guard和Action

```
class ShorterThan {
    public:
        explicit ShorterThan(int maxLength) : length(maxLength) {}
        bool operator() (const string& str) const {
            return str.length() < length;
        }
    private:
        const int length;
};
ShorterThan shorterThan5{5}, shorterThan9{9};
```

状态机

- A state machine is a concrete model describing the behavior of a system. It is composed of a finite number of states and transitions.
- Boost MSM状态机的实现主要基于UML的状态机的定义
  包含： State, Event, Transition, Action, Guard
- Boost MSM official link
  https://www.boost.org/doc/libs/1_73_0/libs/msm/doc/HTML/index.html
- 代码约定

```
#include <boost/msm/front/state_machine_def.hpp> // boost MSM header file
namespace msm = boost::msm; // 使用msm命名空间，简化代码
```

状态

- 用于描述系统在某一特定时间内的状态
- 一个状态可以包含以个部分：
  - 状态名 - 状态名称
  - Entry Action - 进入状态时执行的动作
  - Exit Action - 离开状态下时执行的操作
  - Defer Event - 在该状态下未处理的事件列表，而是被推迟并排队等待另一个状态的对象处理

Boost MSM 定义状态

- Must be derived from boost::msm::front::state<>

```
struct InitState : public msm::front::state<>
{
    // Entry action
    template <class Event, class Fsm>
    void on_entry(Event const &, Fsm &) const
    { ... }
    template <class Fsm>
    // 争对Inner消息的on_entry函数，在状态机收到Inner消息并进入到InitState时被调用。
    // 可以在此实现对应某特定消息的处理动作
    void on_entry(Event::Inner const &, Fsm &) const
    { ... }
    // Exit action
    template <class Event, class Fsm>
    void on_exit(Event const &, Fsm &) const
    { ... }
};
```

消息（事件）

- 在某一特定时间，发生可对系统产生影响的事件
- 消息可以有参数
- 消息被系统接收后，消息被消耗（consume）
- 比如对于键盘来说，用户每按下一个键，键盘就接收到一个消息，按键值为该消息的参数。当键盘接收消息后，用户需从新按键，才能再次触发按键消息。
- 在某一特定State下，消息可以被Defer，再推出该状态后，该消息被自动触发。

Boost MSM 定义消息

- Simple structure, no special base class is required

```
struct KeyPressed {
    KeyPressed(int key) : key{key} {}; // Constructor isn't mandatory
    int key;
};

struct PowerOff {}; // Empty Structure is Okay
```

Transition, Guard, Action

- 系统在接收到消息后，从当前状态转变到另一个状态。（源状态和目标状态可为同一状态）
- Transition发生前，系统调用Guard，用于决定是否发生状态转移。
  - 比如汽车发动前，检查刹车状态
- Guard检查通过后，系统退出当前状态，调用当前状态的exit方法。
  - 比如退出驻车状态，如果双跳灯开启的话，关闭双跳灯
- 系统调用Action方法用以响应收到的消息。
  - 比如启动发动机，响应启动消息
- 系统完成消息响应的Action后，进入目标状态，调用目标状态的entry方法。
  - 比如进入行车状态，打开行车灯

Boost MSM Transition Table

- Boost MSM使用Transition Table描述Transition

```
using msm::front::Row;

// Transition table
struct transition_table : public boost::mpl::vector<
    // |Start State      |Event         |Next State       |Action               |Guard
    Row<State::InitState, Event::Event, State::NextState>
    Row<State::NextState, Event::Stop,  State::EndState, Action::OnEventStop, Guard::EventStopGuard>

    // Row的参数个数可以是3，即Action和Guard为none如：
    // Row<State::InitState, Event::Event, State::NextState>
    // Row的参数个数也可以是4，即Guard为none
    // Row<State::InitState, Event::Stop, State::EndState, Action::OnEventStop>
>{};
```

- 通过transition table，可以确定State的Id（即current_state()返回的值），其确定的方法是:先从Start State这一列开始由上往下（由0开始）为每一个State标index（InitState为0， NextState为1），当遍历完Start State后。同样规则遍历Next State这一列（EndState为2）

Boost MSM Guard

```
struct GeneralGuard
{
    // Operator() with four parameters
    template <class Event, class Fsm, class SourceState, class TargetState>
    bool operator()(Event const &e, Fsm &, SourceState &, TargetState &)
    {
    }
    // 根据C++ Template的特性，也可以定义 () operator的specialized版本，用于响应特定类型的Event
    template <class Fsm, class SourceState, class TargetState>
    bool operator()(Event::Stop const &e, Fsm &, SourceState &, TargetState &)
    {
        return e.readyToStop;
    }
    // 同样也可以定义对应特定状态，或特定状态机的 Operator() 操作符
};
```

Boost MSM Action

```
struct Action
{
    // Parameter is same as Guard, but without return value
    template <class Event, class Fsm, class SourceState, class TargetState>
    void operator()(Event const &, Fsm &fsm, SourceState &, TargetState &) const
    {
    }
};
```

和Guard一样，也可以定义operator（）的Specialized版本

Boost MSM StateMachine

```
struct StateMachine_ : public msm::front::state_machine_def<StateMachine_>
{
    // 当调用状态机对象的start()函数启动状态机时被调用
    template <class Event, class Fsm>
    void on_entry(Event const&, Fsm&) const
    {}
    // 当调用状态机对象的stop()函数停止状态机时被调用
    template <class Event, class Fsm>
    void on_exit(Event const&, Fsm&) const
    {}
    // Set initial state
    typedef State::InitState initial_state;
    // Transition table
    struct transition_table : public boost::mpl::vector<
        Row  <State::InitState, Event::Stop,     State::EndState,   Action::OnEventStop>
    >{};
};
// Pick a back-end
typedef msm::back::state_machine<StateMachine_> StateMachine;
```

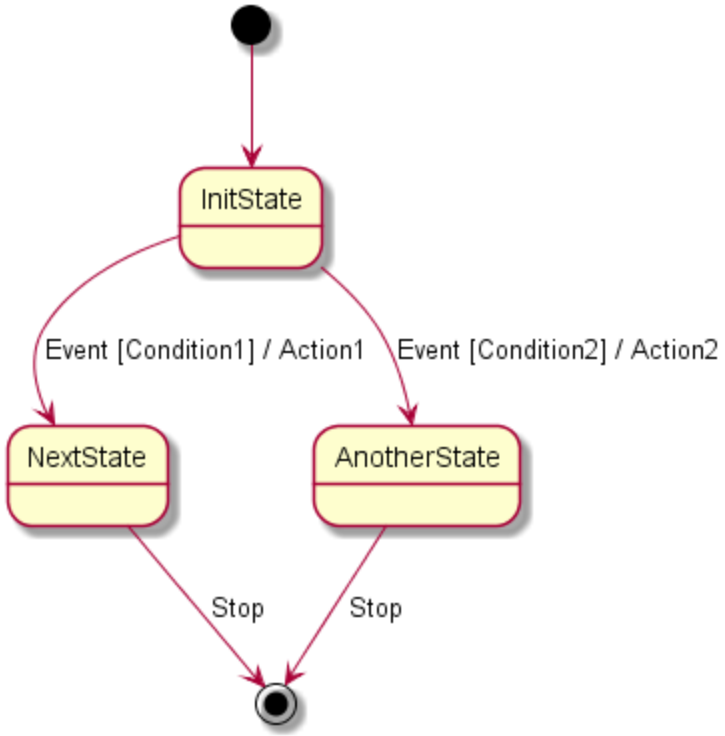
Boost MSM 基本操作

```
typedef msm::back::state_machine<StateMachine_> StateMachine;
StateMachine machine;

machine.start(); // 启动状态机。 还有一重载版本start(event); Region
                 // 如果状态机的on_entry有响应event的specialized的版本，则调用该Specialized版本
machine.process_event(KeyPressed{60});
machine.current_state(); // 得到当前状态值，注返回值为int*类型，为一组当前所处的类型（如果状态机
                         // 中有多个Orthogonal Region，则数组有多个状态值）
                         // 状态值有Transition Table中State由上至下的顺序决定其值。
machine.stop(); // 停止状态机。 还有一重载版本stop(event)
```

Conflict Transition

- 当响应同一消息时，根据不同条件进行不同处理

```
// Transition table
struct transition_table : public boost::mpl::vector<
    Row<InitState, Event, NextState,    Action1, Condition1>,
    Row<InitState, Event, AnotherState, Action2, Condition2>
>{};
```



注意：Boost MSM匹配transition的顺序是由下向上的。上例的匹配顺序为：

- 先运行Condition2，如满足，运行Action2，状态迁移到AnotherState
- 如Condition2验证失败，运行Condition1，如满足，运行Action1，状态迁移到NextState
- 如Condition1验证失败，则不发生状态迁移

Anonymous Transitions

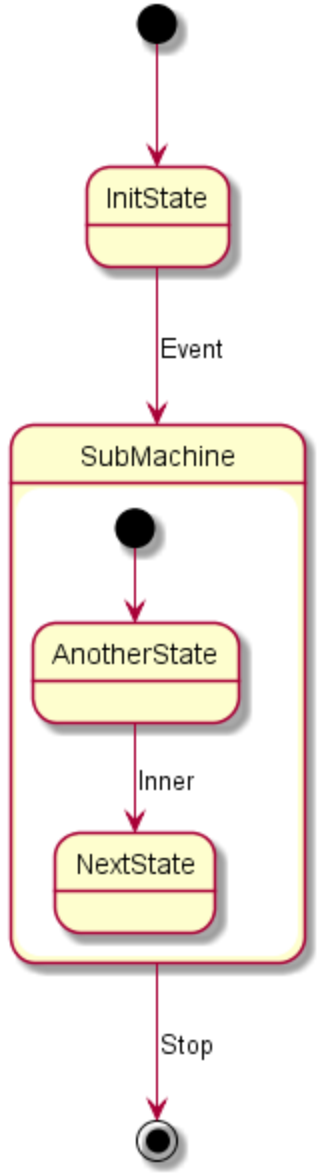
- 在没有事件触发的情况下，当一个State完成后自动迁移到下一状态。
  - 状态机启动后进入初始状态
  - 可以通过设置Guard，实现条件判断，状态机可以迁徙到不同目标状态
- 在上例中将触发消息用boost::msm::front::none替代，就将普通State Transit转化为Anonymous Transit

```
struct transition_table : public boost::mpl::vector<
    Row<InitState, msm::front::none, NextState,     Action1, Condition1>,
    Row<InitState, msm::front::none, AnotherState,  Action2, Condition2>
>{};
```

- 上述Transit Table中的Action和Guard也可删除或为none。则状态自动迁移，匹配规则顺序同样自下而上。

Submachine

```
struct StateMachine_ : public msm::front::state_machine_def<StateMachine_>
{
    // Define SubMachine
    struct SubMachine_ : public msm::front::state_machine_def<SubMachine_>
    {
        typedef State::AnotherState initial_state;
        struct transition_table : public boost::mpl::vector<
            Row<State::AnotherState, Event::Inner, State::NextState>,
            Row<State::NextState,    Event::Stop,  State::EndState>
        >{};
    };
    typedef msm::back::state_machine<SubMachine_> SubMachine;
    typedef State::InitState initial_state;
    struct transition_table : public boost::mpl::vector<
        Row<State::InitState, Event::Event, SubMachine>
    >{};
};
typedef msm::back::state_machine<StateMachine_> StateMachine;
```



Submachine直接定义在State Machine内，Boost MSM会自动将消息派送到Submachine内。

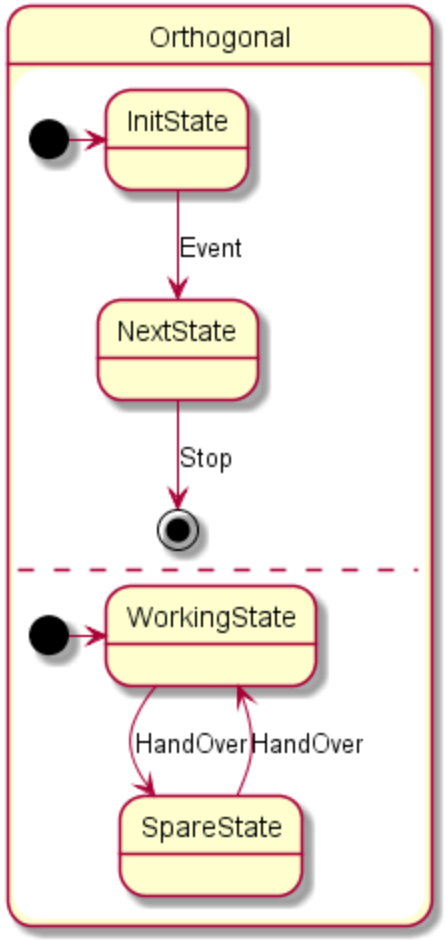
```
StateMachine sut;
sut.start();
sut.process_event(Event::Event{});
sut.process_event(Event::Inner{});
```

Orthogonal Region

- 有时我们需要定义在一个状态机里面定义两个正交的子状态机。比如一个子状态机专门用于处理业务，而另一个状态机专门维护当前的状态。

```
struct StateMachine_ : public msm::front::state_machine_def<StateMachine_>
{
    // Set initial state and define two Orthogonal region
    typedef boost::mpl::vector<State::InitState, State::WorkingState> initial_state;
    // Transition table
    struct transition_table : public boost::mpl::vector<
        //|Start                |Event           |Next
        Row<State::InitState,    Event::Event,    State::NextState>,
        Row<State::NextState,    Event::Stop,     State::EndState>,
        Row<State::WorkingState, Event::HandOver, State::SpareState>,
        Row<State::SpareState,   Event::HandOver, State::WorkingState>
    >{};
};
```

- 注意两个子状态不要有公共的消息，不然位于前面的Transition会被后面的Transition掩盖掉。
- 定义有几个Region，由typedef boost::mpl::vector<...> initial_state;模板中的初始状态个数决定



Defer Event

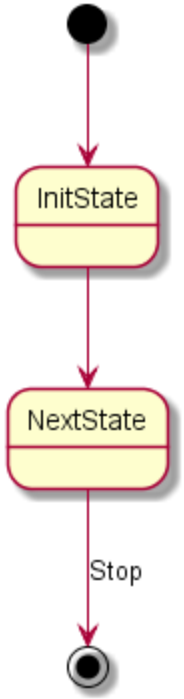
```
struct StateMachine_ : public msm::front::state_machine_def<StateMachine_>
{
    // Make the Current StateMachine support Defer Message
    typedef int activate_deferred_events;
    typedef State::InitState initial_state;
    // Transition table
    struct transition_table : public boost::mpl::vector<
        //|Start              |Event         |Next               |Action              |Guard
        Row<State::InitState, Event::Event, State::NextState, Action::Action1,   Guard::Condition1>,
        Row<State::InitState, Event::Stop,  msm::front::none, msm::front::Defer, Guard::Condition1>,
        Row<State::NextState, Event::Stop,  State::EndState,  Action::Action2,   Guard::Condition2>
    >{};
};
// Pick a back-end
typedef msm::back::state_machine<StateMachine_> StateMachine;

StateMachine sut{};
sut.start();
sut.process_event(Event::Stop{}); // Event is deferred at first
sut.process_event(Event::Event{});
```

- 通过Guard可以决定，是否在当前状态下保存收到的Event
- 状态机可以保存多条消息

Anonymous State

- 某一个State不响应任何消息，自动转移到下一状态，该状态为Anonymous State。

```
struct StateMachine_ : public msm::front::state_machine_def<StateMachine_>
{
    typedef State::InitState initial_state;
    // Transition table
    struct transition_table : public boost::mpl::vector<
        // |Start              |Event              |Next
        Row<State::InitState, msm::front::none, State::NextState>,
        Row<State::NextState, Event::Stop,      State::EndState>
    >{};
};
// Pick a back-end
typedef msm::back::state_machine<StateMachine_> StateMachine;
```



- Row<State::InitState, msm::front::none, State::NextState>将InitState设置为Anonymous State
- Anonymous State的优先级最高。