基于HSM的微赛事状态管理方案

1. **状态模型**

微赛事的不同阶段可以抽象为不同状态，以状态机管理这些状态，进而控制赛事的生命周期。

以斗士锦标赛为例，此赛事综合性强，由积分赛和淘汰赛组成，先以积分赛海选前N强，再通过淘汰赛决出冠军。将赛事阶段抽象为状态模型，如图1-1所示：

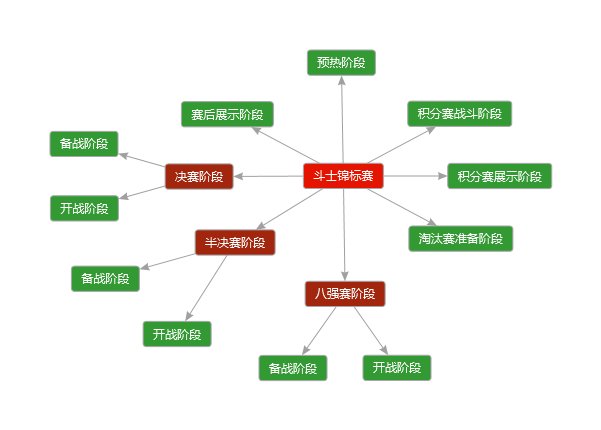


图1-1 斗士锦标赛状态模型

斗士锦标赛具有状态机，统一管理所有赛事状态。八强赛、半决赛和决赛又分为备战和开战阶段，可以抽象为状态机。而备战和开战阶段，以及其余阶段为最小单位，于是抽象为状态。

1. **状态树**

以状态机是状态为设计思想，使状态机之间易于聚合嵌套。状态衍生状态机，而状态机既能够管理状态，也能够管理状态机。倘若以状态机管理状态机，则形成层次状态机。将状态机和状态视作节点，层次状态机就是多叉树，可以称为状态树。

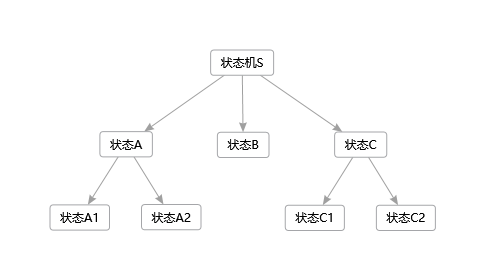


图2-1 状态树

凡是涉及状态树的序列化与反序列化，都采用深度优先的遍历方式。如图2-1，状态树的遍历顺序为S-A-A1-A2-B-C-C1-C2。

自根节点向最底层遍历有效节点，这些有效节点的状态编号形成有效节点路径，又称为状态编号向量。如图2-1，若状态树的最底层有效节点是状态A2，则有效节点路径是A-A2。

1. **状态机**

状态衍生多功能状态，多功能状态派生状态机。如图3-1所示：

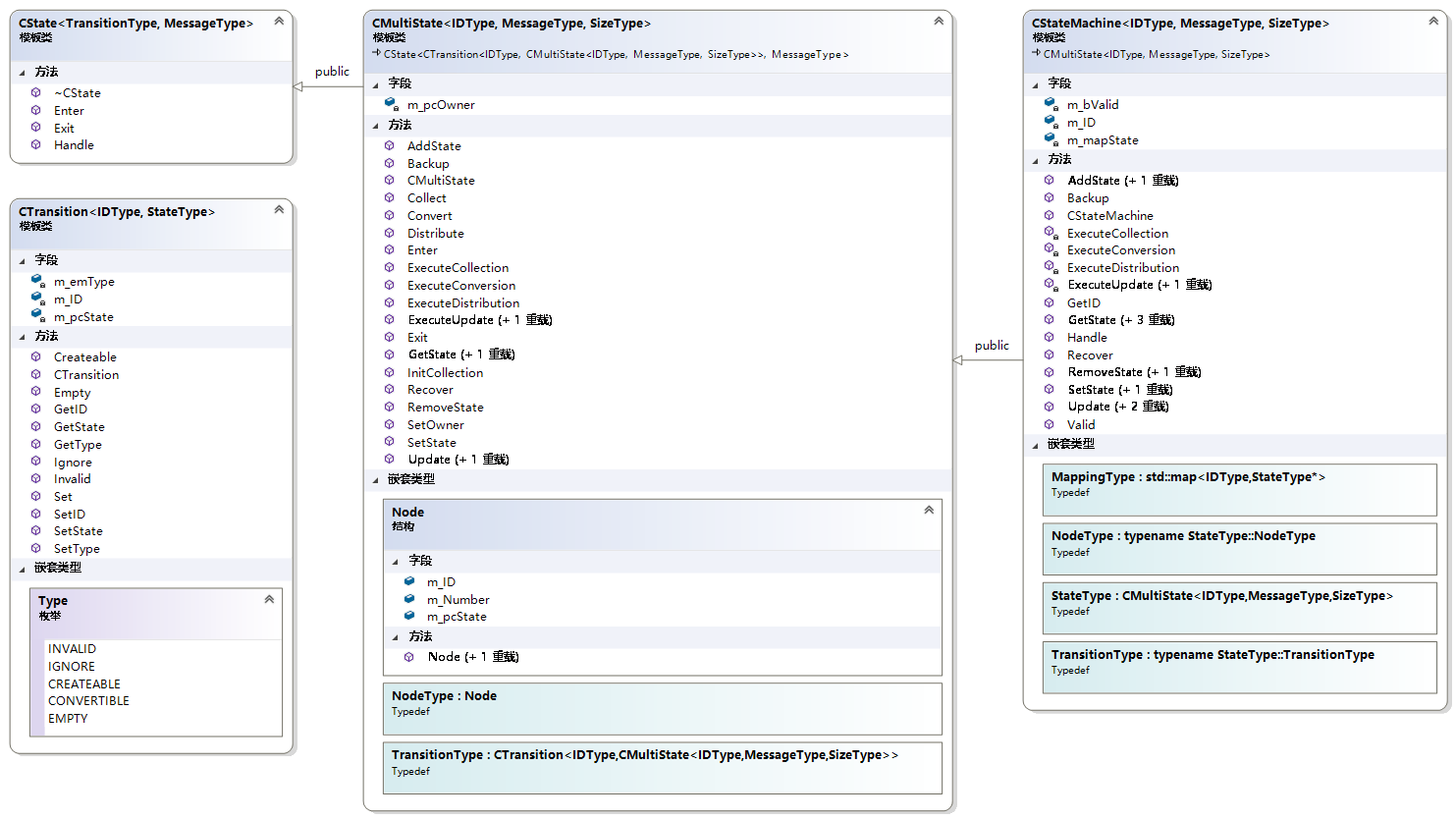


图3-1 状态机UML类图

状态机并未直接继承状态，而是继承由状态派生的多功能状态。其实，多功能状态才是状态机管理的状态。多功能状态为层次状态机定义接口，没有状态机为管理状态而定义的数据结构。状态机类模板说明如表3-1所示：

表3-1 状态机类模板说明

|  |  |
| --- | --- |
| **类模板** | **说明** |
| CState | 状态抽象类模板，定义处理事件的接口。 |
| CMultiState | 多功能状态抽象类模板，作为状态树的叶子节点。 |
| CStateMachine | 状态机类模板，作为状态树的非叶子节点。 |
| CTransition | 过渡类模板。过渡是抽象事物，当处理输入和转换状态之时，在不同状态之间传递信息。 |

**3.1 构建与销毁**

状态机提供三种构建方式和两种销毁方式，以确保构建与销毁的灵活性。

**3.1.1 构建方式**

状态机的三种构建方式分别是外部单独添加方式、外部批量添加方式和内部动态添加方式。内部添加方式指状态向所属状态机添加状态，除此之外都称为外部添加方式。

外部单独添加方式既可以向位于指定路径的状态机添加单状态，也可以先添加单状态至所属状态机，再添加所属状态机至另一状态机。外部单独添加方式对应于函数AddState，其调用形式如下：

|  |
| --- |
| // 添加指定状态至当前状态机  stateMachine.AddState(TYPE\_A, new A(this));  // 添加指定状态至指定状态机  std::vector<id\_type> ids(1, TYPE\_A);  stateMachine.AddState(TYPE\_A1, new A1(this), ids);  stateMachine.AddState(TYPE\_A2, new A2(this), ids); |

外部批量添加方式是先序列化状态树，生成节点向量并交予状态机，再反序列化状态树，分发节点以构建状态机。外部批量添加方式对应于函数Distribute，其调用形式如下：

|  |
| --- |
| // 生成节点向量  nodes.reserve(8);  nodes.push\_back(node\_type(0, NULL, 3));  nodes.push\_back(node\_type(TYPE\_A, new A(this), 2));  nodes.push\_back(node\_type(TYPE\_A1, new A1(this), 0));  nodes.push\_back(node\_type(TYPE\_A2, new A2(this), 0));  nodes.push\_back(node\_type(TYPE\_B, new B(this), 0));  nodes.push\_back(node\_type(TYPE\_C, new C(this), 2));  nodes.push\_back(node\_type(TYPE\_C1, new C1(this), 0));  nodes.push\_back(node\_type(TYPE\_C2, new C2(this), 0));  // 分发节点以构建状态机  stateMachine.Distribute(nodes); |

内部动态添加方式指在执行处理输入函数Handle，或者执行强制转换函数Convert之时，由前一状态创建后一状态，以过渡实例的形式，将后一状态交予状态机。处理输入函数Handle的调用形式如下：

|  |
| --- |
| transition\_type C1::Handle(message\_type message)  {  // 若未达到转换条件，返回忽略状态的过渡实例  if (message < TYPE\_C1)  return transition\_type(transition\_type::IGNORE);  // 若达到转换条件，返回创建状态的过渡实例，创建并指示状态机转换状态  return transition\_type(transition\_type::CREATEABLE, TYPE\_C2, new C2(owner));  } |

**3.1.2 销毁方式**

状态机与状态是聚合关系，在销毁状态机之前，先销毁状态，防止资源泄漏。状态机支持两种销毁方式，分别是单独移除方式和批量收集方式。

单独移除方式可以通过函数RemoveState移除并释放非当前状态，也可以通过调用函数GetState获取并释放状态实例。函数RemoveState的调用形式如下：

|  |
| --- |
| // 移除指定状态机的指定状态  std::vector<id\_type> ids(1, TYPE\_A);  delete stateMachine.RemoveState(TYPE\_A1, ids);  delete stateMachine.RemoveState(TYPE\_A2, ids);  // 移除当前状态机的指定状态  delete stateMachine.RemoveState(TYPE\_A); |

批量收集方式先收集所有状态实例，再依次释放状态实例，对应于函数Collect，其调用形式如下：

|  |
| --- |
| // 收集状态实例  std::vector<state\_machine\_type::NodeType> nodes;  stateMachine.Collect(nodes);  // 释放状态实例  for (size\_t index = 0; index < nodes.size(); ++index)  delete nodes[index].m\_pcState; |

**3.2 备份与还原**

在程序运行过程之中，难免会重启进程，或者遇到进程崩溃。为应对这些情况，状态机提供状态的备份与还原方案。

对于不同应用场景，以状态机的角度而言，状态模型不同仍然可以结构化，而状态的数据模型不同却无法结构化，状态机不应限制状态的实现。因此仅备份与还原状态树的有效节点路径，而状态的自定义数据由使用者自行备份与还原。

备份与还原函数以有效节点路径为参数，其调用形式如下：

|  |
| --- |
| // 备份状态机的有效状态  std::vector<id\_type> ids;  stateMachine.Backup(ids);  // 还原状态机至指定状态  stateMachine.Recover(ids); |

特别注意，除非在还原之时，能够确保有效状态存在，否则不应采用内部动态添加方式构建状态机。

**3.3 状态转换**

状态转换分为三个步骤，首先向状态机指定转换目标，再由状态机控制转换流程，最后状态完成转换行为。此设计思想使状态之间未直接关联，能够降低状态之间的耦合性。

从指定转换目标的角度看，既可以由状态机指定目标，也可以由状态指定目标。将状态机指定目标状态视为外部转换，而状态指定目标状态视为内部转换。

**3.3.1 外部转换**

外部转换原来只有设置状态方式，后来为赛事定制功能，支持调整当前阶段，而设计顺序转换方式。设置状态方式对应于函数SetState。

顺序转换方式是按照一定顺序，深度优先遍历状态树，持续转换状态至目标状态。若无目标状态，则持续转换状态，直到最后状态退出，从此状态机成为无效状态。状态机提供两种顺序转换方法。第一种适用于状态编号有序，依次转换状态至目标状态。第二种在序列化状态树的基础之上，指定节点访问序列，此方法更通用，可以多次访问同一节点。顺序转换方式对应于函数Update，其调用形式如下：

|  |
| --- |
| // 按照顺序转换状态至指定状态  //stateMachine.Update(TYPE\_C);  // 按照序列转换状态至指定状态  nodes.reserve(8);  nodes.push\_back(node\_type(0, NULL, 3));  nodes.push\_back(node\_type(TYPE\_A, new A(this), 2));  nodes.push\_back(node\_type(TYPE\_A1, new A1(this), 0));  nodes.push\_back(node\_type(TYPE\_A2, new A2(this), 0));  nodes.push\_back(node\_type(TYPE\_B, new B(this), 0));  nodes.push\_back(node\_type(TYPE\_C, new C(this), 2));  nodes.push\_back(node\_type(TYPE\_C1, new C1(this), 0));  nodes.push\_back(node\_type(TYPE\_C2, new C2(this), 0));  stateMachine.Update(TYPE\_C, nodes); |

**3.3.2 内部转换**

内部转换分为输入驱动和强制转换两种方式。输入驱动是状态机的核心思想，而强制转换用以弥补输入驱动的不足，使状态转换更灵活，不受输入时机限制。

当状态机发生输入事件时，传给当前状态处理，返回过渡实例作为处理结果。倘若过渡实例持有后续状态，则向状态机添加后续状态。而状态机先触发当前状态的退出事件，再触发后续状态的进入事件。此过程称为输入驱动，对应于函数Handle，其一般定义形式如下：

|  |
| --- |
| transition\_type C1::Handle(message\_type message)  {  // 若未达到转换条件，返回忽略状态的过渡实例  if (message < TYPE\_C1)  return transition\_type(transition\_type::IGNORE);  // 若达到转换条件，返回转换状态的过渡实例，指示状态机转换状态  return transition\_type(transition\_type::CONVERTIBLE, TYPE\_C2);  } |

强制转换方式是在非输入驱动之时，由当前状态访问所属状态机，主动转换为后续状态。强制转换方式对应于函数Convert，此方式仅限于同一状态机的状态转换，跨状态机的强制转换由涉及的状态机自行实现。