之前公司的同事写了一个基于epoll的网络服务器，其中涉及到socket状态的转化(如等待接收，接收中，接收完成等)，以及socket之间的转化（如验证完ip权限之后，验证完登录态），可见是一个多层次的状态机。  
但是在原来的实现中却并没有使用状态模式，导致整个逻辑非常复杂，状态之间的跳转也很难把握。本系列的文章将会通过状态模式来重构整套代码。

状态机模式本身这里就不做详细介绍了，读者可以google一下，笔者在仔细对比过《设计模式之禅》，《研磨设计模式》以及游戏中NPC状态机的实现之后，抽象了如下的一套接口.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | */\*=============================================================================*  *# Author: dantezhu - http://www.vimer.cn*  *# Email: zny2008@gmail.com*  *# FileName: interfaces.h*  *# Description: 公共接口*  *# Version: 1.0*  *# LastChange: 2011-01-19 23:24:33*  *# History:*  *=============================================================================\*/*  #ifndef \_INTERFACES\_H\_  #define \_INTERFACES\_H\_  #include <iostream>  #include <map>  using namespace std;    class IFsm;    class IActor  {  public:  virtual ~IActor() {}  virtual int AttachFsmMgr(map<int, IFsm\*> \*mapFsmMgr)=0;  virtual int ChangeState(int destState)=0;  };      class IFsm  {  public:  virtual ~IFsm() {}  */\*\**  *\* @brief 在进入这个状态的时候，obj需要做的事情*  *\**  *\* @param obj*  *\**  *\* @return 0 succ*  *\* else fail*  *\*/*  virtual int Init(IActor\* obj)=0;    */\*\**  *\* @brief 执行这个状态该做的事情*  *\**  *\* @param obj*  *\**  *\* @return 应该进入的另一个状态*  *\* 0 结束本次Process执行，不进入其他状态*  *\* <0 结束整个请求（obj需要被后续删除）*  *\* else 其他状态（可以返回自己，但是会造成循环，有点危险）*  *\**  *\*/*  virtual int Process(IActor\* obj)=0;    */\*\**  *\* @brief 退出这个状态时需要做的事情*  *\**  *\* @param obj*  *\**  *\* @return 0 succ*  *\* else fail*  *\*/*  virtual int Fini(IActor\* obj)=0;  };    #endif |

IActor是整个状态转化所依附的实体，比如socket可以是recv状态，也可是send状态，那么socket就是一个IActor；而IFsm是状态类的抽象，其中的Init，Process，Fini几个函数都只有一个参数即IActor\* obj，IFsm的具体实现类会根据情况来调用obj不同的函数（action）。  
所以这里有两个问题:

* 1.代码中可以看出IFsm是无状态的，即IFsm本身不会存储obj的任何数据，那么为什么不把IFsm的几个函数都定义成static类型呢？这里笔者思索了很久，最终的结论是由于语言的问题，如果是在python中，那么是可以直接传递类名来作为参数的，而在C++中，只能传递一个类的实例在作为参数，这就是原因。
* 2.既然Process注释中说明返回的是下一个状态，那为什么不直接返回一个IFsm指针，而是返回数字呢，这里会在下面的代码中回答

定义了状态之后，我们还缺少一个状态管理器：

|  |  |
| --- | --- |
| 1 | map<int, IFsm\* > m\_mapFsmMgr; |

这个是定义在后面会介绍的world里的，之所以有这个管理器

* 1.是因为C++不同于python，可以直接传递类名
* 2.通过int来做映射能够更好的实现配置化

所以在状态机启动之前，一般需要有如下代码:

|  |  |
| --- | --- |
| 1  2  3 | m\_mapFsmMgr[1]=new CWaitSendFsm();  m\_mapFsmMgr[2]=new CSendingFsm();  m\_mapFsmMgr[3]=new CSendOverFsm(); |

那这里还有一个异议，即有的书上推荐将状态指针定义成CFsmMgr的static变量，当进行状态转化的时候，直接用这个指针即可。但是因为配置化以及底层逻辑与应用逻辑分离的原因，这里个人不建议用如下方式:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class CFsmMgr  {  public:  CFsmMgr () {}  virtual ~CFsmMgr () {}    public:  static IFsm\* WaitSendFsmObj;  static IFsm\* SendingFsmObj;  static IFsm\* SendOverFsmObj;  };  IFsm\* CFsmMgr::WaitSendFsmObj = new CWaitSendFsm();  IFsm\* CFsmMgr::SendingFsmObj = new CSendingFsm();  IFsm\* CFsmMgr::SendOverFsmObj = new CSendOverFsm(); |

接下来是Actor的基类实现:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | class CBaseActor : public IActor  {  public:  CBaseActor () {  m\_Fsm = NULL;  m\_mapFsmMgr = NULL;  }  virtual ~CBaseActor () {}    int AttachFsmMgr(map<int, IFsm\*> \* mapFsmMgr)  {  m\_mapFsmMgr = mapFsmMgr;  return 0;  }    int ChangeState(int destState)  {  if (m\_mapFsmMgr == NULL)  {  return -1;  }    if (0 == destState)  {  //此次处理结束  return 0;  }  else if (destState < 0)  {  //需要关闭整个请求  return destState;  }  IFsm \* destFsm = NULL;  destFsm = (\*m\_mapFsmMgr)[destState];  int state = doChangeFsm(destFsm);  return ChangeState(state);  }  private:  int doChangeFsm(IFsm\* destFsm)  {  if (destFsm == NULL)  {  return 0;  }    if (m\_Fsm != destFsm)  {  if (m\_Fsm != NULL)  {  m\_Fsm->Fini(this);  }  m\_Fsm = destFsm;  m\_Fsm->Init(this);  }  return m\_Fsm->Process(this);  }      protected:  IFsm\* m\_Fsm;  map<int, IFsm\*> \*m\_mapFsmMgr;  }; |

这个代码也是实现了Actor进行状态转化时的一些逻辑，对于状态转化的触发一般会有两种方式:

* 1.外界调用ChangeState，即由外界触发-如epoll出发的recv事件导致socket的状态转化成recving
* 2.状态在执行Process之后自行转化为其他状态-如recving状态在recv结束之后自动转化为recvover状态

所以基于第二点的原因，上面的代码中实现了递归调用的逻辑（这里在状态机的设计中需要注意环的出现）。  
下面是一个继承自Actor基类的socketActor，实现如下:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class CSocketActor : public CBaseActor  {  public:  CSocketActor () {}  virtual ~CSocketActor () {}    int HandleSend()  {  cout<<"sending"<<endl;  if (rand() % 5 == 0)  {  //代表发送完了  return 1;  }  return 0;  }  int HandleRecv()  {  cout<<"recving"<<endl;  return 0;  }  int HandleError()  {  cout<<"error"<<endl;  return 0;  }  int HandleTimeout()  {  cout<<"timeout"<<endl;  return 0;  }  }; |

再来看一下状态类的具体实现:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | //waitsend sending sendover waitrecv recving recvover waitclose closing closeover error timeout    class CWaitSendFsm : public IFsm  {  public:  CWaitSendFsm () {}  virtual ~CWaitSendFsm () {}  virtual int Init(IActor\* obj)  {  cout<<"Init WaitSend"<<endl;  return 0;  }  virtual int Process(IActor\* obj)  {  cout<<"Process WaitSend"<<endl;  return 0;  }  virtual int Fini(IActor\* obj)  {  cout<<"Fini WaitSend"<<endl;  return 0;  }  };  class CSendingFsm : public IFsm  {  public:  CSendingFsm () {}  virtual ~CSendingFsm () {}  virtual int Init(IActor\* obj)  {  cout<<"Init Sending"<<endl;  return 0;  }  virtual int Process(IActor\* obj)  {  cout<<"Process Sending"<<endl;  CSocketActor \* chirdObj = (CSocketActor\*) obj;  int ret = chirdObj->HandleSend();  if (ret == 1)  {  return 3;  }  return 0;  }  virtual int Fini(IActor\* obj)  {  cout<<"Fini Sending"<<endl;  return 0;  }  };  class CSendOverFsm : public IFsm  {  public:  CSendOverFsm () {}  virtual ~CSendOverFsm () {}  virtual int Init(IActor\* obj)  {  cout<<"Init SendOver"<<endl;  return 0;  }  virtual int Process(IActor\* obj)  {  cout<<"Process SendOver"<<endl;  return 0;  }  virtual int Fini(IActor\* obj)  {  cout<<"Fini SendOver"<<endl;  return 0;  }  }; |

OK，这样一切准备工作就完成，暂时先不把真正的epoll引入进来，我们来模拟一个epoll环境（比较假，没有考虑状态的顺序，大家能明白就行）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | class CWorld  {  public:  CWorld () {  srand(time(NULL));  }  virtual ~CWorld () {}    int Init()  {  int count = 10;    m\_mapFsmMgr[1]=new CWaitSendFsm();  m\_mapFsmMgr[2]=new CSendingFsm();  m\_mapFsmMgr[3]=new CSendOverFsm();    for (int i = 0; i < count; i++)  {  IActor \* actor = new CSocketActor();  m\_vecActors.push\_back(actor);    actor->AttachFsmMgr(&m\_mapFsmMgr);  actor->ChangeState(1);  }  return 0;  }  int Run()  {  while (true)  {  int state = 0;  int val = rand() % 5;  switch(val)  {  case 0:  case 1:  state = 1;  break;  case 2:  state = 2;  break;  default:  state = 3;  break;  }  foreach (m\_vecActors, it)  {  (\*it)->ChangeState(state);  }  sleep(1);  }  return 0;  }    private:  vector<IActor\*> m\_vecActors;  map<int, IFsm\* > m\_mapFsmMgr;  }; |

main函数如下:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include "fsm\_achieve.h"  int main(int argc, const char \*argv[])  {  CWorld world;  world.Init();  world.Run();  return 0;  } |

执行结果如下（部分）:

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Init WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

Process WaitSend

这样，一个一层的状态机就实现出来了~后面的文章，我们会面向应用来讨论两层状态机的实现。  
附:  
[代码下载](http://code.google.com/p/vimercode/source/browse/#svn%2Ftrunk%2Ffsm%2Fepoll_fsm)