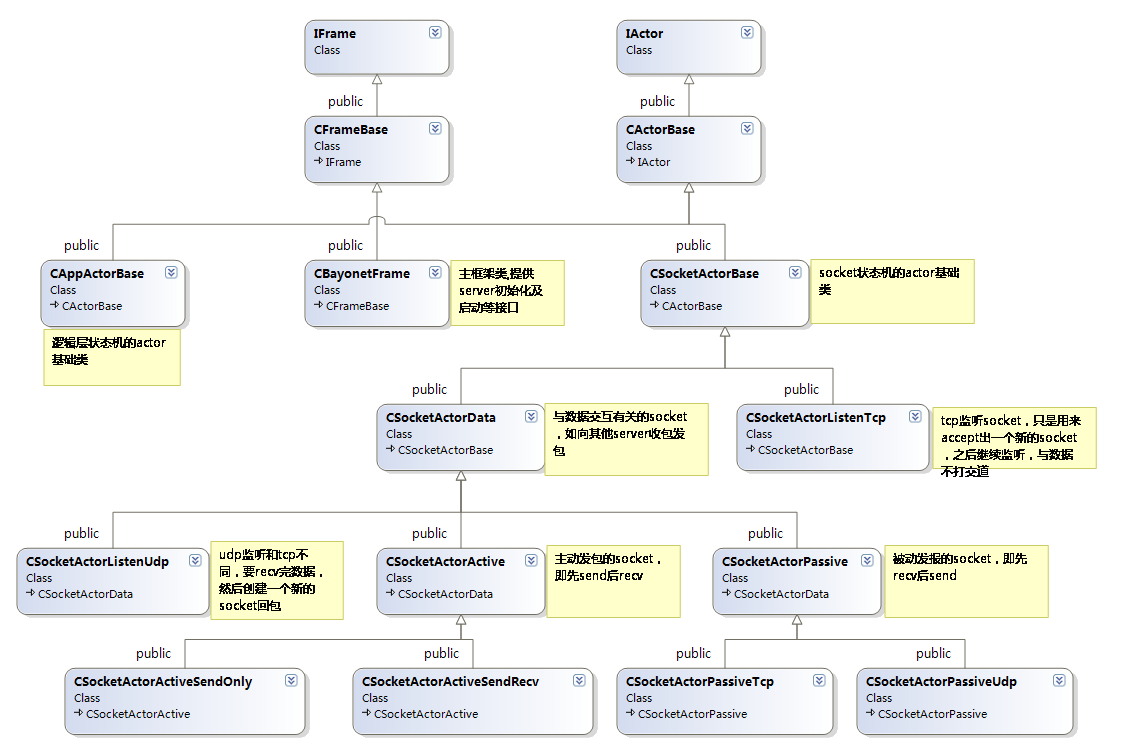
接着上一篇文章: [有限状态机的C++实现(1)-epoll状态机](http://www.vimer.cn/2011/01/%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E6%9C%BA%E7%9A%84c%E5%AE%9E%E7%8E%B01-epoll%E7%8A%B6%E6%80%81%E6%9C%BA.html)，我们今天来介绍更复杂和深入的部分。

为什么会在标题中提到[bayonet](http://code.google.com/p/bayonet/" \t "_blank)这个开源项目呢？笔者本人一直想要写一套架构优美、功能完善的异步server框架，也看过很多朋友、同事实现的版本，虽然功能上基本能满足需求，但是架构上我却始终觉得是有瑕疵的，直到后来和同事讨论，发现可以让一个客户端请求的到来作为一个session，而之后的每一次与其他server的交互都可以看作是一次状态转化，才感觉架构比较合理了。

简单来说即，一个session从开始到介绍会经历两种状态机的变化：

* 1.业务逻辑层面的状态变化，例如先验证登录态，再验证权限，再获取用户资料
* 2.每一个与其他server交互的socket自身的状态变化，如recv、send、等，而socket的状态变化会触发逻辑层的状态变化。

按照这种思路，目前的代码开发已经完成了70%，即可以正常的进行一个session的开始和结束，主要还缺一些细节的代码，比如超时的检测及超时之后的处理，健全的统计之类。好了，我们来用vs看一下代码的整体类图（图压缩比较严重，请单击后查看）：

[](http://www.vimer.cn/wp-content/uploads/2011/02/bayonet_code.png)

每个类的用处已经在途中简单说明了，这里就不再赘述，我们重点来看一下用这个框架来实现一个逻辑server时需要做哪些事情。  
[svr2](http://code.google.com/p/bayonet/source/browse/#svn%2Ftrunk%2Fsrc%2Fsvr2)目录下的main.cpp即实现了一个最简单的server，我们按部分来看其实现:  
**1.逻辑层状态的定义**

|  |  |
| --- | --- |
| 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 | class CAppFsmLogic1 : public CAppFsmBase  {  public:  virtual ~CAppFsmLogic1 () {}  virtual int HandleEntry(CActionInfoSet \*pActionInfoSet, CAppActorBase\* pAppActor)  {  static CActionFirst actionFirst;  StActionInfoParam param;  param.id = 1;  param.ip = "127.0.0.1";  param.port = 100;  param.protoType = PROTO\_TYPE\_UDP;  param.pAction = &actionFirst;  param.actionType = ACTIONTYPE\_SENDONLY;  param.timeout\_ms = 1000;    CActionInfo \* pActionInfo = new CActionInfo();  pActionInfo->Init(param);  pActionInfo->SetAppActor(pAppActor);  pActionInfoSet->Add(pActionInfo);  return 0;  }  virtual int HandleProcess(CActionInfoSet \*pActionInfoSet, CAppActorBase\* pAppActor)  {  trace\_log("HandleProcess");  set<CActionInfo\*> &setAction = pActionInfoSet->GetActionSet();  for(set<CActionInfo\*>::iterator it = setAction.begin(); it != setAction.end(); ++it)  {  trace\_log("error no:%d",(\*it)->GetErrno());  }  return APP\_FSM\_RSP;//代表要回复客户端啦  }  virtual int HandleExit(CActionInfoSet \*pActionInfoSet, CAppActorBase\* pAppActor)  {  return 0;  }  }; |

CAppFsmLogic1是一个逻辑层的状态:

* 1.HandleEntry代表当这个状态第一次进入的时候要做的事情，其函数中创建了一个向其他server发包的action(CActionFirst的定义我们在后面介绍)。
* 2.HandleProcess代表当这个状态的所有action都完成时需要做的事情，return APP\_FSM\_RSP;代表向客户端回包

看到这里大家应该很奇怪，对于每一个socket，判断收包长度以及受到包之后的解包在哪里完成呢？所以我们还需要定义action:  
**2.action的定义**

|  |  |
| --- | --- |
| 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 | #define APP\_FSM\_LOGIC1 2000    class CAppFsmLogic1;  class CActionFirst : public IAction  {  public:  // 为发送打包  int HandleEncodeSendBuf(  IActor\* pSocketActor,  IActor\* pAppActor,  string & strSendBuf,  int &len)  {  trace\_log("send");  strSendBuf="woainizhende111111";  len = strSendBuf.size();  return 0;  }    // 回应包完整性检查  int HandleInput(  IActor\* pSocketActor,  IActor\* pAppActor,  const char \*buf,  int len)  {  return len;  }    // 回应包解析  int HandleDecodeRecvBuf(  IActor\* pSocketActor,  IActor\* pAppActor,  const char \*buf,  int len)  {  CAppActorBase \* app\_actor = new CAppActorBase();  app\_actor->AttachFrame(pSocketActor->GetFrame());  app\_actor->AttachCommu(pSocketActor);  app\_actor->ChangeState(APP\_FSM\_LOGIC1);  trace\_log("listen tcp HandleDecodeRecvBuf");  return 0;  }  }; |

每个函数的意义已经在代码中说明了，可以看出在HandleDecodeRecvBuf中创建逻辑层的actor: app\_actor，并ChangeState为APP\_FSM\_LOGIC1。  
最后就是main函数的实现了:  
**3.main函数实现**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main(int argc, const char \*argv[])  {  CBayonetFrame srv;  StFrameParam param;  param.ip="0.0.0.0";  param.port = 10001;  param.bKeepcnt= true;  //param.protoType = PROTO\_TYPE\_UDP;  param.protoType = PROTO\_TYPE\_TCP;  param.pAction = new CActionFirst();    srv.Init(param);  srv.RegFsm(APP\_FSM\_LOGIC1,new CAppFsmLogic1());  srv.Process();  return 0;  } |

注释的部分是可以随时切换TCP还是UDP的。  
当然作为一个server来说，这里还是太过简单了，比如信号的处理等都没有加上，但是笔者认为那是业务代码需要做的逻辑，所以并没有放到框架中。

OK，整个项目的结构就是这个样子了。  
但是也不得不说点扫兴的话，由于笔者最近有另外一个项目需要投入大量的精力，所以该项目的更新可能会被延缓，这是我所不愿意看到的，所以很希望有志同道合的朋友能够加入到这个项目的开发中来，一起把这个事情做出来。

按照我当初的想法，压力测试框架[fuload](http://code.google.com/p/fuload/)已经就绪了，等到bayonet完成，我们就用fuload来测试一下bayonet的性能究竟如何。

最后，附上bayonet的项目地址:  
<http://code.google.com/p/bayonet/>  
十分欢迎大家感兴趣的朋友与我联系。