# 前言：

v0.0.1 2015-04-10 誉小痕（shawhen2012@hotmail.com）

v0.0.2 2015-04-12 誉小痕（shawhen2012@hotmail.com）

v0.0.3 2015-04-23 誉小痕（shawhen2012@hotmail.com）

v0.0.4 2015-05-02 誉小痕（shawhen2012@hotmail.com）

changelog请查看： <http://bbs.kbengine.org/forum.php?mod=viewthread&tid=247&extra=page%3D1>

新增内容以 ***\*\*New in version x.x.x\*\**** 标示

友情提醒：打开Word的导航标题查看，因为我不会在文档内对标题进行缩进

基于kbengine 0.4.20 （兼容0.5.0）

（现在和这份文档一起的有一个kbengine主干活动图，建议先看看那个图，参照图中的流程然后对照本文档理解流程的实现细节。活动图可在changelog帖子中找到下载地址）

MMOG服务端是一种高品质的工程项目，品读开源的kbe是一种乐趣。本文档我带童鞋们一起领略一下。囿于我知识面和经验方面所限，文中所述之处难免有错误存在，还请读童鞋们睁大慧眼，如果你发现错误，可以 [电邮至shawhen2012@hotmail.com](mailto:电邮至shawhen2012@hotmail.com)。（因为我个人懒散或者时间仓促的关系，这个文档的排版有点小乱。。。）

其他牛逼哄哄的前言就不说了。

从理论上来讲，我们阅读一份源代码，首先应该基于现有的文档从整体上把握项目的架构之后再庖丁解牛一般地细分阅读，不过在我写这个文档的现在，我暂时没发现这样的文档，所以我就按照我自己的阅读顺序从而编排这个文档的内容。

***\*\*NEW in version 0.0.3\*\****

# 客户端概要：

在过去0.0.2中我们对kbengine的服务端进行了大致流程的了解，接下来为了更加全面地了解到kbengine的整个框架流程（从理论上来讲不解读客户端我们也能完完整整地解读完服务端，不过我们还是看一看，以免管窥蠡测之嫌）。本文档采用Ogre的Demo，主干活动图现已更新到v0.0.2，包含ogre demo的活动图。

你可能担心自己不会Ogre，会看不懂这个文档，其实不用太担心，我也不会，我之所以选Ogre，是因为Unity，cocos2d-x，Ogre这些我一个都不会 ;-( ，但Ogre的demo对于我们解读kbengine来说最clean，所以我选了这个来分析。

## main函数：

### 初始化kbengine引擎：

找到客户端的main函数(app/OgreApplication.cpp)，可以看到主要就是装载了一下kbengine.dll，然后执行了kbe\_init，我们跟进kbe\_init（kbengine\_dll/main.cpp）：

bool kbe\_init()

{

g\_componentID = genUUID64();

g\_componentType = CLIENT\_TYPE;

//pserverconfig = new ServerConfig;

pconfig = new Config;

if(!loadConfig())

{

ERROR\_MSG("loadConfig() is error!\n");

return false;

}

DebugHelper::initialize(g\_componentType);

INFO\_MSG( "-----------------------------------------------------------------------------------------\n\n\n");

#ifdef USE\_OPENSSL

if(KBEngine::KBEKey::getSingletonPtr() == NULL)

KBEngine::KBEKey kbekey(KBEngine::Resmgr::getSingleton().matchPath("key/") + "kbengine\_public.key", "");

#endif

if(g\_pScript == NULL)

g\_pScript = new KBEngine::script::Script();

if(g\_pDispatcher == NULL)

g\_pDispatcher = new Network::EventDispatcher();

if(g\_pNetworkInterface == NULL)

{

g\_pNetworkInterface = new Network::NetworkInterface(g\_pDispatcher,

0, 0, "", 0, 0,

0, "", 0, 0);

}

if(!installPyScript(\*g\_pScript, g\_componentType))

{

ERROR\_MSG("installPyScript() is error!\n");

return false;

}

g\_pApp = new ClientApp(\*g\_pDispatcher, \*g\_pNetworkInterface, g\_componentType, g\_componentID);

g\_pApp->setScript(g\_pScript);

START\_MSG(COMPONENT\_NAME\_EX(g\_componentType), g\_componentID);

if(!g\_pApp->initialize())

{

ERROR\_MSG("app::initialize is error!\n");

g\_pApp->finalise();

Py\_DECREF(g\_pApp);

g\_pApp = NULL;

uninstallPyScript(\*g\_pScript);

SAFE\_RELEASE(g\_pNetworkInterface);

SAFE\_RELEASE(g\_pScript);

SAFE\_RELEASE(g\_pDispatcher);

SAFE\_RELEASE(pserverconfig);

SAFE\_RELEASE(pconfig);

return false;

}

if(g\_pTelnetServer == NULL)

{

g\_pTelnetServer = new TelnetServer(g\_pDispatcher, g\_pNetworkInterface);

g\_pTelnetServer->pScript(g\_pScript);

if(!g\_pTelnetServer->start(pconfig->telnet\_passwd,

pconfig->telnet\_deflayer,

pconfig->telnet\_port))

{

ERROR\_MSG("app::initialize: TelnetServer is error!\n");

return false;

}

}

else

{

g\_pTelnetServer->pScript(g\_pScript);

}

if(g\_pThreadPool == NULL)

{

g\_pThreadPool = new thread::ThreadPool();

if(!g\_pThreadPool->isInitialize())

{

g\_pThreadPool->createThreadPool(1, 1, 4);

}

else

{

ERROR\_MSG("g\_threadPool.isInitialize() is error!\n");

return false;

}

}

INFO\_MSG(fmt::format("---- {} is running ----\n", COMPONENT\_NAME\_EX(g\_componentType)));

PyEval\_ReleaseThread(PyThreadState\_Get());

KBEConcurrency::setMainThreadIdleCallbacks(&releaseLock, &acquireLock);

g\_pThreadPool->addTask(new KBEMainTask());

return true;

}

有没有感觉很熟悉？是的，跟v0.0.2中分析得到的服务端初始化流程大同小异，我们从installPyScript开始看。

#### 脚本引擎安装（暂未完成）：

要读懂kbe的脚本引擎模块需要对python这种脚本语言有所了解，特别是python嵌入c的各种api，如果你还没准备好建议你去这里过一遍：[python3 c api手册](https://docs.python.org/3/c-api/)。

看看installPyScript的代码（client\_lib/kbemain.h）：

inline bool installPyScript(KBEngine::script::Script& **script**, COMPONENT\_TYPE **componentType**)

{

if(Resmgr::getSingleton().respaths().*size*() <= 0 ||

Resmgr::getSingleton().getPyUserResPath().*size*() == 0 ||

Resmgr::getSingleton().getPySysResPath().*size*() == 0 ||

Resmgr::getSingleton().getPyUserScriptsPath().*size*() == 0)

{

ERROR\_MSG("EntityApp::installPyScript: KBE\_RES\_PATH is error!\n");

return false;

}

std::*wstring* **user\_scripts\_path** = L"";

wchar\_t\* **tbuf** = KBEngine::strutil::char2wchar(const\_cast<char\*>(Resmgr::getSingleton().getPyUserScriptsPath().*c\_str*()));

if(**tbuf** != NULL)

{

**user\_scripts\_path** += **tbuf**;

free(**tbuf**);

}

else

{

ERROR\_MSG("EntityApp::installPyScript: KBE\_RES\_PATH error[char2wchar]!\n");

return false;

}

std::*wstring* **pyPaths** = **user\_scripts\_path** + L"common;";

**pyPaths** += **user\_scripts\_path** + L"data;";

**pyPaths** += **user\_scripts\_path** + L"user\_type;";

if(**componentType** == CLIENT\_TYPE)

**pyPaths** += **user\_scripts\_path** + L"client;";

else

**pyPaths** += **user\_scripts\_path** + L"bots;";

std::*string* **kbe\_res\_path** = Resmgr::getSingleton().getPySysResPath();

**kbe\_res\_path** += "scripts/common";

**tbuf** = KBEngine::strutil::char2wchar(const\_cast<char\*>(**kbe\_res\_path**.*c\_str*()));

bool **ret** = **script**.install(**tbuf**, **pyPaths**, "KBEngine", **componentType**);

free(**tbuf**);

EntityDef::installScript(**script**.getModule());

client::Entity::installScript(**script**.getModule());

Entities<client::Entity>::installScript(NULL);

EntityGarbages<client::Entity>::installScript(NULL);

return **ret**;

}

基本流程可以理解为先调用Script（pyscript/script.h/.cpp）的install接口，我们跟进看一下：

设置python解释器的“家”路径->初始化->获取\_\_main\_\_模块->添加一个叫”KBEngine”的模块（到sys.modules）->为KBEngine模块添加一个叫做component（值为安装脚本引擎的组件的类型字符串，比如loginapp/client……）的变量->为”KBEngine”模块添加一个叫做genUUID64的方法->安装几个dll模块（socket/hashlib/select/ctypes/elementtree/unicodedata/pyexpat）->初始化”KBEngine”模块->将KBEngine模块对象添加为\_\_main\_\_模块的属性->安装”Math”模块->安装“KBExtra”模块。

#### ClientApp，计时器添加：

我们跟进app的initialize接口，在initializeBegin中看到（lib/client\_lib/clientapp.cpp）：

bool ClientApp::initializeBegin()

{

gameTimer\_ = this->**dispatcher**().addTimer(1000000 / g\_kbeConfig.gameUpdateHertz(), this,

reinterpret\_cast<void \*>(TIMEOUT\_GAME\_TICK));

ProfileVal::setWarningPeriod(stampsPerSecond() / g\_kbeConfig.gameUpdateHertz());

Network::g\_extReceiveWindowBytesOverflow = 0;

Network::g\_intReceiveWindowBytesOverflow = 0;

Network::g\_intReceiveWindowMessagesOverflow = 0;

Network::g\_extReceiveWindowMessagesOverflow = 0;

Network::g\_receiveWindowMessagesOverflowCritical = 0;

return true;

}

即添加了一个计时器到事件派发器。由v0.0.2中得出的结论我们知道，计时器是在事件派发器的processTimers中处理的，所以我们留下了一个问题，事件派发器是在哪进行事件分发的。

我们再看看计时器的处理，如下（lib/client\_lib/clientapp.cpp）：

void ClientApp::handleTimeout(TimerHandle, void \* **arg**)

{

switch (reinterpret\_cast<uintptr>(**arg**))

{

case TIMEOUT\_GAME\_TICK:

{

handleGameTick();

}

default:

break;

}

}

我们跟到handleGameTick：

void ClientApp::handleGameTick()

{

g\_kbetime++;

threadPool\_.onMainThreadTick();

networkInterface().processChannels(KBEngine::Network::MessageHandlers::pMainMessageHandlers);

tickSend();

switch(state\_)

{

case C\_STATE\_INIT:

state\_ = C\_STATE\_PLAY;

break;

case C\_STATE\_INITLOGINAPP\_CHANNEL:

state\_ = C\_STATE\_PLAY;

break;

case C\_STATE\_LOGIN:

state\_ = C\_STATE\_PLAY;

if(!ClientObjectBase::login())

{

WARNING\_MSG("ClientApp::handleGameTick: login is failed!\n");

return;

}

break;

case C\_STATE\_LOGIN\_GATEWAY\_CHANNEL:

{

state\_ = C\_STATE\_PLAY;

bool **ret** = updateChannel(false, "", "", "", 0);

if(**ret**)

{

// 先握手然后等helloCB之后再进行登录

Network::Bundle\* **pBundle** = Network::Bundle::ObjPool().createObject();

(\***pBundle**).newMessage(BaseappInterface::hello);

(\***pBundle**) << KBEVersion::versionString();

(\***pBundle**) << KBEVersion::scriptVersionString();

if(Network::g\_channelExternalEncryptType == 1)

{

pBlowfishFilter\_ = new Network::BlowfishFilter();

(\***pBundle**).appendBlob(pBlowfishFilter\_->key());

pServerChannel\_->pFilter(NULL);

}

else

{

std::*string* **key** = "";

(\***pBundle**).appendBlob(**key**);

}

pServerChannel\_->pEndPoint()->send(**pBundle**);

Network::Bundle::ObjPool().reclaimObject(**pBundle**);

// ret = ClientObjectBase::loginGateWay();

}

}

break;

case C\_STATE\_LOGIN\_GATEWAY:

state\_ = C\_STATE\_PLAY;

if(!ClientObjectBase::loginGateWay())

{

WARNING\_MSG("ClientApp::handleGameTick: loginGateWay is failed!\n");

return;

}

break;

case C\_STATE\_PLAY:

break;

default:

KBE\_ASSERT(false);

break;

};

}

可以看到是根据不同的状态进行不同的动作的一个状态机。

#### 事件派发器工作：

如下构造了一个全局线程池：

**g\_pThreadPool** = new thread::ThreadPool();

添加一个任务：

**g\_pThreadPool**->addTask(new KBEMainTask());

我们跟一跟KBEMainTask，看到其process接口（client/kbengine\_dll/main.cpp）：

virtual bool process()

{

while(**g\_pApp** && !**g\_break**)

{

**g\_pLock** = new KBEngine::script::PyThreadStateLock;

if(**targetID** >= 0)

{

**g\_pApp**->setTargetID(**targetID**);

**targetID** = -1;

}

**g\_inProcess** = true;

**g\_pApp**->processOnce(true);

**g\_inProcess** = false;

SAFE\_RELEASE(**g\_pLock**);

}

**g\_break** = true;

return false;

}

跟进g\_pApp->processOnce()接口看看（lib/client\_lib/clientapp.cpp）：

int ClientApp::processOnce(bool **shouldIdle**)

{

return dispatcher\_.processOnce(**shouldIdle**);

}

不难发现这就是我们在v0.0.2中提到过的事件派发器的工作入口，由此也解决了我们上面的疑问，计时器是在一个单独的线程中进行工作的（这也可以说是客户端和服务端的区别使然，GUI客户端的主线程几乎总是UI线程，如果不专门开辟新线程进行i/o之类的事件派发，那UI线程就被阻塞了）。

### 客户端OgreApplication app：

#### 构造OgreApplication：

// Create application object

OgreApplication\* **app** = new OgreApplication();

跟进OgreApplication的构造函数：

kbe\_registerEventHandle(this);

这个函数是kbengine\_dll提供的一个接口，跟进去看的话也很简单，就不贴代码了，基本就是注册一个事件handle（流程是山路十八弯，应用程序调用dll接口，dll内转给全局的g\_pApp，然后再转给eventHandler\_，现在我们不得不假设我们还不知道这个事件handle到底作甚的），这里和Timer一样的手法，通过handle从而给handler提供处理接口

#### 运行app：

此后main函数的流程转移到app->go()，代码如下（app/OgreApplication.cpp）：

void OgreApplication::go(void)

{

if (!setup())

return;

while(!mShutDown)

{

mRoot->renderOneFrame();

// 通知系统分派消息

Ogre::WindowEventUtilities::messagePump();

if(!**g\_hasEvent**)

*Sleep*(1);

}

// clean up

destroyScene();

}

跟进setup：

bool OgreApplication::setup(void)

{

#ifdef \_DEBUG

mRoot = new Ogre::*Root*("plugins\_d.cfg");

#else

mRoot = new Ogre::*Root*();

#endif

setupResources();

if (!configure()) return false;

// Set default mipmap level (NB some APIs ignore this)

Ogre::TextureManager::getSingleton().setDefaultNumMipmaps(5);

// Create any resource listeners (for loading screens)

createResourceListener();

// Load resources

loadResources();

createFrameListener();

mWindow->setDeactivateOnFocusChange(false);

changeSpace(new SpaceLogin(mRoot, mWindow, mInputManager, mTrayMgr));

return true;

};

可以看到changeSpace到了SpaceLogin，跟一下OgreApplication::changeSpace，可以看到它还调用了新的Space的setup接口。

##### 登陆页面：

上面说到changeSpace的时候都会触发新页面的setup，我们看看SpaceLogin的setup（client/ogre/app/space.cpp）:

bool Space::setup()

{

setupResources();

// Create the scene

createScene();

return true;

}

跟到SpaceLogin的createScene接口，从气氛上可以感受出就是设置页面的控件之类的（我看不懂Ogre的代码）。

此后的代码流程几乎全是event-driven了，所以我们得找到一个事件的入口点。

###### 登陆 ClientApp::login：

概读一下SpaceLogin（app/space\_login.h/space\_login.cpp）可以看到在buttonHit中，当点击“fast login”之后便触发kbe\_login接口，代码如下（kbengine\_dll/main.cpp）：

bool kbe\_login(const char\* **accountName**, const char\* **passwd**, const char\* **datas**, KBEngine::uint32 **datasize**,

const char\* **ip**, KBEngine::uint32 **port**)

{

if(**ip** == NULL || **port** == 0)

{

**ip** = **pconfig**->ip();

**port** = **pconfig**->port();

}

kbe\_lock();

std::*string* **sdatas**;

**sdatas**.*assign*((const char\*)**datas**, **datasize**);

bool **ret** = **g\_pApp**->login(**accountName**, **passwd**, **sdatas**, **ip**, **port**);

kbe\_unlock();

return **ret**;

}

可以看到实际上负责实现登陆到kbengine服务端接口的是一个kbengine\_dll内ClientApp的全局实例的login接口（lib/client\_lib/clientapp.cpp）：

bool ClientApp::login(std::*string* **accountName**, std::*string* **passwd**, std::*string* **datas**,

std::*string* **ip**, KBEngine::uint32 **port**)

{

connectedGateway\_ = false;

if(canReset\_)

reset();

clientDatas\_ = **datas**;

bool **ret** = updateChannel(true, **accountName**, **passwd**, **ip**, **port**);

if(**ret**)

{

// 先握手然后等helloCB之后再进行登录

Network::Bundle\* **pBundle** = Network::Bundle::ObjPool().createObject();

(\***pBundle**).newMessage(LoginappInterface::hello);

(\***pBundle**) << KBEVersion::versionString();

(\***pBundle**) << KBEVersion::scriptVersionString();

if(Network::g\_channelExternalEncryptType == 1)

{

pBlowfishFilter\_ = new Network::BlowfishFilter();

(\***pBundle**).appendBlob(pBlowfishFilter\_->key());

}

else

{

std::*string* **key** = "";

(\***pBundle**).appendBlob(**key**);

}

pServerChannel\_->pEndPoint()->send(**pBundle**);

Network::Bundle::ObjPool().reclaimObject(**pBundle**);

//ret = ClientObjectBase::login();

}

return **ret**;

}

这个接口调用了一下updateChannel，我们跟进去看看：

bool ClientApp::updateChannel(bool **loginapp**, std::*string* **accountName**, std::*string* **passwd**,

std::*string* **ip**, KBEngine::uint32 **port**)

{

if(pServerChannel\_->pEndPoint())

{

pServerChannel\_->stopSend();

networkInterface().dispatcher().deregisterReadFileDescriptor(\*pServerChannel\_->pEndPoint());

networkInterface().deregisterChannel(pServerChannel\_);

}

bool **ret** = **loginapp** ? (initLoginappChannel(**accountName**, **passwd**, **ip**, **port**) != NULL) : (initBaseappChannel() != NULL);

if(**ret**)

{

if(pTCPPacketReceiver\_)

pTCPPacketReceiver\_->pEndPoint(pServerChannel\_->pEndPoint());

else

pTCPPacketReceiver\_ = new Network::TCPPacketReceiver(\*pServerChannel\_->pEndPoint(), networkInterface());

if(pTCPPacketSender\_)

pTCPPacketSender\_->pEndPoint(pServerChannel\_->pEndPoint());

else

pTCPPacketSender\_ = new Network::TCPPacketSender(\*pServerChannel\_->pEndPoint(), networkInterface());

pServerChannel\_->pPacketSender(pTCPPacketSender\_);

networkInterface().registerChannel(pServerChannel\_);

networkInterface().dispatcher().registerReadFileDescriptor(\*pServerChannel\_->pEndPoint(), pTCPPacketReceiver\_);

}

return **ret**;

}

比较核心的地方在于：

bool **ret** = **loginapp** ? (initLoginappChannel(**accountName**, **passwd**, **ip**, **port**) != NULL) : (initBaseappChannel() != NULL);

ClientApp::login接口内将loginapp置为true，所以会调用initLoginappChannel，我们跟看看：

Network::Channel\* ClientObjectBase::initLoginappChannel(std::*string* **accountName**, std::*string* **passwd**, std::*string* **ip**, KBEngine::uint32 **port**)

{

Network::EndPoint\* **pEndpoint** = Network::EndPoint::ObjPool().createObject();

**pEndpoint**->socket(*SOCK\_STREAM*);

if (!**pEndpoint**->good())

{

ERROR\_MSG("ClientObjectBase::initLoginappChannel: couldn't create a socket\n");

Network::EndPoint::ObjPool().reclaimObject(**pEndpoint**);

return NULL;

}

u\_int32\_t **address**;

Network::Address::string2ip(**ip**.*c\_str*(), **address**);

if(**pEndpoint**->connect(*htons*(**port**), **address**) == -1)

{

ERROR\_MSG(fmt::format("ClientObjectBase::initLoginappChannel: connect server is error({})!\n",

kbe\_strerror()));

Network::EndPoint::ObjPool().reclaimObject(**pEndpoint**);

return NULL;

}

Network::Address **addr**(**ip**.*c\_str*(), **port**);

**pEndpoint**->addr(**addr**);

pServerChannel\_->pEndPoint(**pEndpoint**);

**pEndpoint**->setnonblocking(true);

**pEndpoint**->setnodelay(true);

password\_ = **passwd**;

name\_ = **accountName**;

lastSentActiveTickTime\_ = timestamp();

return pServerChannel\_;

}

基本可以理解为将pServerChannel\_置于与服务端的LoginApp端点通信。

既然已经到了这，我们就顺便来看看initBaseappChannel：

Network::Channel\* ClientObjectBase::initBaseappChannel()

{

Network::EndPoint\* **pEndpoint** = Network::EndPoint::ObjPool().createObject();

**pEndpoint**->socket(*SOCK\_STREAM*);

if (!**pEndpoint**->good())

{

ERROR\_MSG("ClientObjectBase::initBaseappChannel: couldn't create a socket\n");

Network::EndPoint::ObjPool().reclaimObject(**pEndpoint**);

return NULL;

}

u\_int32\_t **address**;

Network::Address::string2ip(ip\_.c\_str(), **address**);

if(**pEndpoint**->connect(*htons*(port\_), **address**) == -1)

{

ERROR\_MSG(fmt::format("ClientObjectBase::initBaseappChannel: connect server is error({})!\n",

kbe\_strerror()));

Network::EndPoint::ObjPool().reclaimObject(**pEndpoint**);

return NULL;

}

Network::Address **addr**(ip\_.c\_str(), port\_);

**pEndpoint**->**addr**(**addr**);

pServerChannel\_->pEndPoint(**pEndpoint**);

**pEndpoint**->setnonblocking(true);

**pEndpoint**->setnodelay(true);

connectedGateway\_ = true;

lastSentActiveTickTime\_ = timestamp();

lastSentUpdateDataTime\_ = timestamp();

return pServerChannel\_;

}

可以看到这个接口内将pServerChannel\_连接到了新的端点，地址是ip\_和port\_，由于这个变量名使得我们看不出这到底是个啥的地址（代码里面也没注释出），所以我们对这个接口的探索就到此为止了。

###### 发送 LoginappInterface::hello：

我们继续回到app的login接口，根据刚才的理解得知，就是先将pServerChannel\_和LoginApp连接（就是接口传入的ip/port），然后构造一个LoginappInterface::hello的Bundle通过pServerChannel\_发送出去。之后依靠事件派发器分发网络读事件（processNetwork），再由TcpPacketReceiver接收数据并缓存到相应的Channel。再由Channel进行数据的处理（processTimers），根据数据包的id调用相应的messageHandler，messageHandler根据编译时构造的消息-处理映射调用相应的处理接口。（你晕了吗!?整理完这里我在火炬2刷了好几张图才算清醒一点）

###### 等待 ClientInterface::onHelloCB：

LoginappInterface::hello的响应包的id是ClientInterface::onHelloCB（这里可以配合服务端代码发现，在后面更详细的服务端解读里面会提及，现在靠这个名字应该就能感知到了），而onHello的处理接口是ClientApp::onHelloCB（其实是ClientObjectbase::onHelloCB），我们来两斤代码压压惊（lib/client\_lib/clientobjectbase.cpp）：

void ClientObjectBase::onHelloCB(Network::Channel\* **pChannel**, MemoryStream& **s**)

{

std::*string* **verInfo**;

**s** >> **verInfo**;

std::*string* **scriptVerInfo**;

**s** >> **scriptVerInfo**;

std::*string* **protocolMD5**;

**s** >> **protocolMD5**;

std::*string* **entityDefMD5**;

**s** >> **entityDefMD5**;

COMPONENT\_TYPE **ctype**;

**s** >> **ctype**;

INFO\_MSG(fmt::format("ClientObjectBase::onHelloCB: verInfo={}, scriptVerInfo={}, protocolMD5={}, entityDefMD5={}, addr:{}\n",

**verInfo**, **scriptVerInfo**, **protocolMD5**, **entityDefMD5**, **pChannel**->c\_str()));

onHelloCB\_(**pChannel**, **verInfo**, **scriptVerInfo**, **protocolMD5**, **entityDefMD5**, **ctype**);

}

可以看到就是把MemoryStream反序列化了，把参数转给onHelloCB\_（注意ctype变量），因为ClientApp重载了这个接口，所以我们去ClientApp::onHelloCB\_看看：

void ClientApp::onHelloCB\_(Network::Channel\* **pChannel**, const std::*string*& **verInfo**,

const std::*string*& **scriptVerInfo**, const std::*string*& **protocolMD5**, const std::*string*& **entityDefMD5**,

COMPONENT\_TYPE **componentType**)

{

if(Network::g\_channelExternalEncryptType == 1)

{

pServerChannel\_->pFilter(pBlowfishFilter\_);

pBlowfishFilter\_ = NULL;

}

if(**componentType** == LOGINAPP\_TYPE)

{

state\_ = C\_STATE\_LOGIN;

}

else

{

state\_ = C\_STATE\_LOGIN\_GATEWAY;

}

}

大意基本为根据返回onHelloCB消息的组件类型置state\_为C\_STATE\_LOGIN或C\_STATE\_LOGIN\_GATEWAY。

在异步/事件驱动模型里面，因为这个代码编排被打乱之后，会造成很多困扰，例如上面的发送hello和等待onHelloCB，本来是很自然的顺序流程，却不得不拆成两个函数，并且还要依靠一些状态变量来维系。解决这个问题有人用协程，有人用状态机，有人用队列……理论上来讲协程是最自然的解决手段（如果你还没用过这个，赶快去google），不过C++的协程至今为止都还比较薄弱，不像go/python之流内置协程语义，所以在C++工业工程里面用协程的几近于无。kbe选择了用状态机。

设置完状态之后等待计时器触发handleTimeout。

###### 触发 ClientApp::handleTimeout：

###### 发送 LoginappInterface::login：

根据前面的分析可知，此时state\_被置为C\_STATE\_LOGIN，可以看到这种状态下将会调用ClientObjectBase::login（lib/client\_lib/clientobjectbase.cpp）：

bool ClientObjectBase::login()

{

Network::Bundle\* **pBundle** = Network::Bundle::ObjPool().createObject();

// 提交账号密码请求登录

(\***pBundle**).newMessage(LoginappInterface::login);

(\***pBundle**) << typeClient\_;

(\***pBundle**).appendBlob(clientDatas\_);

(\***pBundle**) << name\_;

(\***pBundle**) << password\_;

(\***pBundle**) << EntityDef::md5().getDigestStr();

pServerChannel\_->send(**pBundle**);

connectedGateway\_ = false;

return true;

}

###### 等待 ClientInterface::onLoginSuccessfully：

即发送LoginappInterface::login数据包（前面是LoginappInterface::hello，就是说直接调用ClientApp::login是发送hello，然后等待onHelloCB，然后再调用ClientObjectBase::login，当然这一切都是kbe自动的，跟应用没关系），然后等待onLoginSuccessfully/Failed（如果你是现在是凭直觉发现这一点的话，那你和我一样是幸运的，在后面的版本中会解读的），如下：

void ClientApp::onLoginSuccessfully(Network::Channel \* **pChannel**, MemoryStream& **s**)

{

ClientObjectBase::onLoginSuccessfully(**pChannel**, **s**);

Config::getSingleton().writeAccountName(name\_.c\_str());

state\_ = C\_STATE\_LOGIN\_GATEWAY\_CHANNEL;

}

我们跟进ClientObjectBase::onLoginSuccessfully（lib/client\_lib/clientobjectbase.cpp）：

void ClientObjectBase::onLoginSuccessfully(Network::Channel \* **pChannel**, MemoryStream& **s**)

{

std::*string* **accountName**;

**s** >> **accountName**;

**s** >> ip\_;

**s** >> port\_;

**s**.readBlob(serverDatas\_);

connectedGateway\_ = false;

INFO\_MSG(fmt::format("ClientObjectBase::onLoginSuccessfully: {} addr={}:{}!\n", name\_, ip\_, port\_));

EventData\_LoginSuccess **eventdata**;

eventHandler\_.fire(&**eventdata**);

gatewayIP\_ = ip\_;

gateWayPort\_ = port\_;

}

此时此刻我们终于搞清了ip\_/port\_是个甚，他们就是登陆后返回的所谓的网关（就是一个Baseapp）地址。

将状态置为C\_STATE\_LOGIN\_GATEWAY\_CHANNEL后等待handleTimeout

###### 触发 ClientApp::handleTimeout：

此时状态置为C\_STATE\_LOGIN\_GATEWAY\_CHANNEL，会执行这一段：

case C\_STATE\_LOGIN\_GATEWAY\_CHANNEL:

{

state\_ = C\_STATE\_PLAY;

bool **ret** = updateChannel(false, "", "", "", 0);

if(**ret**)

{

// 先握手然后等helloCB之后再进行登录

Network::Bundle\* **pBundle** = Network::Bundle::ObjPool().createObject();

(\***pBundle**).newMessage(BaseappInterface::hello);

(\***pBundle**) << KBEVersion::versionString();

(\***pBundle**) << KBEVersion::scriptVersionString();

if(Network::g\_channelExternalEncryptType == 1)

{

pBlowfishFilter\_ = new Network::BlowfishFilter();

(\***pBundle**).appendBlob(pBlowfishFilter\_->key());

pServerChannel\_->pFilter(NULL);

}

else

{

std::*string* **key** = "";

(\***pBundle**).appendBlob(**key**);

}

pServerChannel\_->pEndPoint()->send(**pBundle**);

Network::Bundle::ObjPool().reclaimObject(**pBundle**);

// ret = ClientObjectBase::loginGateWay();

}

}

break;

updateChannel中loginapp此次调用被置为false，所以将pServerChannel\_连接到ip\_/port\_（登陆返回的网关地址）端点。

然后再次发送hello消息，不过此时发送的是BaseappInterface::hello，然后等待ClientInterface::onHelloCB（是的，跟LoginappInterface::hello返回一样的消息的id），只不过他们的ComponentType值是不一样的。

###### 再次等待 ClientInterface::onHelloCB：

和iii中一样的流程，只是这次状态state\_被置为C\_STATE\_LOGIN\_GATEWAY，然后等待ClientApp::handleTimeout。

###### 触发 ClientApp::handleTimeout：

此时状态state\_被置为C\_STATE\_LOGIN\_GATEWAY，所以执行下面的代码：

case C\_STATE\_LOGIN\_GATEWAY:

state\_ = C\_STATE\_PLAY;

if(!ClientObjectBase::loginGateWay())

{

WARNING\_MSG("ClientApp::handleGameTick: loginGateWay is failed!\n");

return;

}

break;

我们跟进ClientObjectBase::loginGateWay（lib/client\_lib/clientobjectbase.cpp）：

bool ClientObjectBase::loginGateWay()

{

// 请求登录网关, 能走到这里来一定是连接了网关

connectedGateway\_ = true;

Network::Bundle\* **pBundle** = Network::Bundle::ObjPool().createObject();

(\***pBundle**).newMessage(BaseappInterface::loginGateway);

(\***pBundle**) << name\_;

(\***pBundle**) << password\_;

pServerChannel\_->send(**pBundle**);

return true;

}

可知是发送BaseappInterface::loginGateway数据包给网关，然后等待ClientInterface::onCreatedProxies/ClientInterface::onLoginGatewayFailed（我们暂时凭小宇宙感知得出这个结论）。

###### 等待 ClientInterface::onCreatedProxies：

代码如下（lib/client\_lib/clientobjectbase.cpp）：

void ClientObjectBase::onCreatedProxies(Network::Channel \* **pChannel**, uint64 **rndUUID**, ENTITY\_ID **eid**, std::*string*& **entityType**)

{

client::Entity\* **entity** = pEntities\_->*find*(**eid**);

if(**entity** != NULL)

{

WARNING\_MSG(fmt::format("ClientObject::onCreatedProxies({}): rndUUID={} eid={} entityType={}. has exist!\n",

name\_, **rndUUID**, **eid**, **entityType**));

return;

}

if(entityID\_ == 0)

{

EventData\_LoginGatewaySuccess **eventdata**;

eventHandler\_.fire(&**eventdata**);

}

BUFFEREDMESSAGE::*iterator* **iter** = bufferedCreateEntityMessage\_.*find*(**eid**);

bool **hasBufferedMessage** = (**iter** != bufferedCreateEntityMessage\_.*end*());

// 能走到这里来一定是连接了网关

connectedGateway\_ = true;

entityID\_ = **eid**;

rndUUID\_ = **rndUUID**;

INFO\_MSG(fmt::format("ClientObject::onCreatedProxies({}): rndUUID={} eid={} entityType={}!\n",

name\_, **rndUUID**, **eid**, **entityType**));

// 设置entity的baseMailbox

EntityMailbox\* **mailbox** = new EntityMailbox(EntityDef::findScriptModule(**entityType**.c\_str()),

NULL, appID(), **eid**, MAILBOX\_TYPE\_BASE);

client::Entity\* **pEntity** = createEntity(**entityType**.c\_str(), NULL, !**hasBufferedMessage**, **eid**, true, **mailbox**, NULL);

KBE\_ASSERT(**pEntity** != NULL);

if(**hasBufferedMessage**)

{

// 先更新属性再初始化脚本

this->onUpdatePropertys(**pChannel**, \***iter**->second.*get*());

bufferedCreateEntityMessage\_.*erase*(**iter**);

**pEntity**->initializeEntity(NULL);

SCRIPT\_ERROR\_CHECK();

}

}

值得注意的是此时并未再改变任何状态，我们跟进这一段代码：

if(entityID\_ == 0)

{

EventData\_LoginGatewaySuccess **eventdata**;

eventHandler\_.fire(&**eventdata**);

}

跟进eventHandler\_.fire（lib/client\_lib/event.cpp）：

void EventHandler::fire(const EventData\* **lpEventData**)

{

EVENT\_HANDLES::*iterator* **iter** = eventHandles\_.*begin*();

for(; **iter** != eventHandles\_.*end*(); ++**iter**)

{

(\***iter**)->kbengine\_onEvent(**lpEventData**);

}

}

###### 缓存kbe事件数据

联想前面构造OgreApplication时注册的event handle可知此接口之作用，我们跟进OgreApplication::kbengine\_onEvent（client/ogre/app/OgreApplication.cpp）：

void OgreApplication::kbengine\_onEvent(const KBEngine::EventData\* **lpEventData**)

{

KBEngine::EventData\* **peventdata** = KBEngine::copyKBEngineEvent(**lpEventData**);

if(**peventdata**)

{

boost::mutex::*scoped\_lock* **lock**(**g\_spaceMutex**);

events\_.*push*(std::tr1::*shared\_ptr*<const KBEngine::EventData>(**peventdata**));

**g\_hasEvent** = true;

}

}

可知这个接口就是将一个EventData推入到events\_队列，那问题又来了，何时处理events\_呢？

###### 分派kbe事件数据队列

不管是通过搜索通读，我们找到如下代码（client/ogre/OgreApplication.cpp）：

bool OgreApplication::frameRenderingQueued(const Ogre::FrameEvent& **evt**)

{

if(**g\_space** == NULL)

{

return BaseApplication::frameRenderingQueued(**evt**);

}

while(true)

{

**g\_spaceMutex**.**lock**();

if(events\_.*empty*())

{

**g\_spaceMutex**.unlock();

break;

}

std::tr1::*shared\_ptr*<const KBEngine::EventData> **pEventData** = events\_.*front*();

events\_.*pop*();

**g\_spaceMutex**.unlock();

KBEngine::EventID **id** = **pEventData**->**id**;

if(**id** == CLIENT\_EVENT\_SERVER\_CLOSED)

{

//OgreApplication::getSingleton().changeSpace(new SpaceAvatarSelect(mRoot, mWindow, mInputManager, mTrayMgr));

//break;

}

// 如果需要在本线程访问脚本层则需要锁住引擎

if(**id** == CLIENT\_EVENT\_SCRIPT)

{

kbe\_lock();

}

**g\_space**->kbengine\_onEvent(**pEventData**.*get*());

if(**id** == CLIENT\_EVENT\_SCRIPT)

{

kbe\_unlock();

}

**g\_hasEvent** = false;

}

if(!**g\_space**->frameRenderingQueued(**evt**))

return true;

return BaseApplication::frameRenderingQueued(**evt**);

}

###### 处理kbe事件数据

我们看到，在这个接口中（据说是ogre渲染帧的中间过程调用）对events\_进行了处理，将所有的EventData提交给g\_space->kbengine\_onEvent接口，我们跟进Space::kbengine\_onEvent（在Space中这是一个空体的函数，以SpaceLogin为例）：

void SpaceLogin::kbengine\_onEvent(const KBEngine::EventData\* **lpEventData**)

{

switch(**lpEventData**->id)

{

case CLIENT\_EVENT\_LOGIN\_SUCCESS:

break;

case CLIENT\_EVENT\_LOGIN\_FAILED:

{

const KBEngine::EventData\_LoginFailed\* **info** = static\_cast<const KBEngine::EventData\_LoginFailed\*>(**lpEventData**);

char **str**[256];

if(**info**->failedcode == 20)

{

sprintf(**str**, "server is starting, please wait!");

}

else

{

sprintf(**str**, "SpaceLogin::kbengine\_onEvent(): login is failed(code=%u)!", **info**->failedcode);

}

*MessageBox*( NULL, **str**, "warning!", *MB\_OK*);

}

break;

case CLIENT\_EVENT\_LOGIN\_GATEWAY\_SUCCESS:

OgreApplication::getSingleton().changeSpace(new SpaceAvatarSelect(mRoot, mWindow, mInputManager, mTrayMgr));

break;

case CLIENT\_EVENT\_LOGIN\_GATEWAY\_FAILED:

{

const KBEngine::EventData\_LoginGatewayFailed\* **info** = static\_cast<const KBEngine::EventData\_LoginGatewayFailed\*>(**lpEventData**);

char **str**[256];

sprintf(**str**, "SpaceLogin::kbengine\_onEvent(): loginGateway is failed(code=%u)!", **info**->failedcode);

*MessageBox*( NULL, **str**, "warning!", *MB\_OK*);

}

break;

case CLIENT\_EVENT\_VERSION\_NOT\_MATCH:

{

const KBEngine::EventData\_VersionNotMatch\* **info** = static\_cast<const KBEngine::EventData\_VersionNotMatch\*>(**lpEventData**);

char **str**[256];

sprintf(**str**, "SpaceLogin::kbengine\_onEvent(): verInfo=%s not match(server:%s)", **info**->verInfo.*c\_str*(), **info**->serVerInfo.*c\_str*());

*MessageBox*( NULL, **str**, "error!", *MB\_OK*);

}

break;

case CLIENT\_EVENT\_SCRIPT\_VERSION\_NOT\_MATCH:

{

const KBEngine::EventData\_ScriptVersionNotMatch\* **info** = static\_cast<const KBEngine::EventData\_ScriptVersionNotMatch\*>(**lpEventData**);

char **str**[256];

sprintf(**str**, "SpaceLogin::kbengine\_onEvent(): scriptVerInfo=%s not match(server:%s)", **info**->verInfo.*c\_str*(), **info**->serVerInfo.*c\_str*());

*MessageBox*( NULL, **str**, "error!", *MB\_OK*);

}

break;

case CLIENT\_EVENT\_LAST\_ACCOUNT\_INFO:

{

const KBEngine::EventData\_LastAccountInfo\* **info** = static\_cast<const KBEngine::EventData\_LastAccountInfo\*>(**lpEventData**);

**g\_accountName** = **info**->name;

if(mTrayMgr)

{

OgreBites::ParamsPanel\* **pannel** = ((OgreBites::ParamsPanel\*)mTrayMgr->getWidget("accountName"));

if(**pannel**)

{

**pannel**->setParamValue(0, **g\_accountName**);

}

}

}

break;

default:

break;

};

}

在ClientObjectBase::onCreatedProxies中，EventData\_LoginGatewaySuccess的id即为CLIENT\_EVENT\_LOGIN\_GATEWAY\_SUCCESS，故此时将执行：

OgreApplication::getSingleton().changeSpace(new SpaceAvatarSelect(mRoot, mWindow, mInputManager, mTrayMgr));

即Space切换到SpaceAvatarSelect。

至此，我们已能了解到kbengine客户端与服务端交互流程的整个轮廓。

##### 精灵选择页面：

# 服务端概要：

从已有的文档可知（我得假设你已经大致看完了kbe官网的现有文档），kbe由几个组件共同协作，所以我们先看看组件们：

各个组件被设计为独立的app，使用网络通信进行协作。C++程序自然是从main函数开始。

## main函数的戏法：

看起来似乎所有的组件都有一个这样的宏(KBENGINE\_MAIN)来包裹main函数

int KBENGINE\_MAIN(int argc, char\* argv[])

{

ENGINE\_COMPONENT\_INFO& info = g\_kbeSrvConfig.getXXX();

return kbeMainT<XXX>(argc, argv, YYY, info.externalPorts\_min,

info.externalPorts\_max, info.externalInterface, 0, info.internalInterface);

}

这个宏展开是这样子：

kbeMain(int argc, char\* argv[]); \

int main(int argc, char\* argv[]) \

{ \

loadConfig(); \

g\_componentID = genUUID64(); \

parseMainCommandArgs(argc, argv); \

char dumpname[MAX\_BUF] = {0}; \

kbe\_snprintf(dumpname, MAX\_BUF, "%"PRAppID, g\_componentID); \

KBEngine::exception::installCrashHandler(1, dumpname); \

int retcode = -1; \

THREAD\_TRY\_EXECUTION; \

retcode = kbeMain(argc, argv); \

THREAD\_HANDLE\_CRASH; \

return retcode; \

} \

稍微整理一下之后main函数看起来很像是这个样子：

int kbeMain(int argc, char\* argv[]);

int main(int argc, char\* argv[])

{

loadConfig();

g\_componentID = genUUID64();

parseMainCommandArgs(argc, argv);

char dumpname[MAX\_BUF] = {0};

kbe\_snprintf(dumpname, MAX\_BUF, "%"PRAppID, g\_componentID);

KBEngine::*exception*::installCrashHandler(1, dumpname);

int retcode = -1;

THREAD\_TRY\_EXECUTION;

retcode = kbeMain(argc, argv);

THREAD\_HANDLE\_CRASH;

return (retcode);

}

int kbeMain(int argc, char\* argv[])

{

ENGINE\_COMPONENT\_INFO& info = g\_kbeSrvConfig.getXXX();

return kbeMainT<XXX>(argc, argv, YYY, info.externalPorts\_min, info.externalPorts\_max, info.externalInterface, 0, info.internalInterface);

}

嗯。。。基本可以理解为每个组件的main函数流程都是一样的，只是在特化kbeMainT时所给参数不一样。

### 载入配置文件：

我们跟着main函数的loadConfig进去看看（kbemain.h）

inline void loadConfig()

{

Resmgr::getSingleton().initialize();

// "../../res/server/kbengine\_defs.xml"

g\_kbeSrvConfig.loadConfig("server/kbengine\_defs.xml");

// "../../../assets/res/server/kbengine.xml"

g\_kbeSrvConfig.loadConfig("server/kbengine.xml");

}

在serverconfig.h中可以看到这样的代码：

#define g\_kbeSrvConfig ServerConfig::getSingleton()

#### Resmgr和ServerConfig单例的实现：

Resmgr和ServerConfig，这两个类都是被搞成单例了的（kbe的单例是用的OGRE的单利，不算太严格的单例，线程安全，编译时阻断都无法满足，我们先不细究），

Resmgr是资源管理器，在resmgr.h/.cpp中声明和定义，在FixedMessages类(fixed\_messages.h/.cpp)的构造函数中被new出来。（有点小隐晦，我是调试了一下才跟到的。。。）

过程是这样，在各个组件的xxx\_interface.cpp中，有这样的代码：（摘自loginapp）

#include "loginapp\_interface.h"

#define DEFINE\_IN\_INTERFACE

#define LOGINAPP

#include "loginapp\_interface.h"

xxx\_interface.h中，有这样的代码：

#if defined(DEFINE\_IN\_INTERFACE)

#undef KBE\_LOGINAPP\_INTERFACE\_H

#endif

#ifndef KBE\_LOGINAPP\_INTERFACE\_H

#define KBE\_LOGINAPP\_INTERFACE\_H

大意可以理解为在xxx\_interface.cpp中通过在包含xxx\_interface.h前后定义DEFINE\_IN\_INTERFACE和LOGINAPP宏，使得xxx\_interface.h被包含了两次（但产生的代码确实不同的），从而对xxx\_interface.h内的一些变量实现了声明（第一次）和定义（第二次）。

在xxx\_interface.h中有这样一句：

NETWORK\_INTERFACE\_DECLARE\_BEGIN(LoginappInterface)

展开就是：

// 定义接口域名称

#ifndef DEFINE\_IN\_INTERFACE

#define NETWORK\_INTERFACE\_DECLARE\_BEGIN(INAME) \

namespace INAME \

{ \

extern Network::MessageHandlers messageHandlers; \

#else

#define NETWORK\_INTERFACE\_DECLARE\_BEGIN(INAME) \

namespace INAME \

{ \

Network::MessageHandlers messageHandlers; \

#endif

#define NETWORK\_INTERFACE\_DECLARE\_END() }

在第一次包含xxx\_interface.h的时候就是extern Network….这样的外部全局变量引用声明，第二次包含的时候就是Network::M…..这样的全局变量的定义了。

在Network::MessageHandles的构造函数(message\_handler.cpp)中，有：

MessageHandlers::MessageHandlers():

msgHandlers\_(),

msgID\_(1),

exposedMessages\_()

{

g\_fm = Network::FixedMessages::getSingletonPtr();

if(g\_fm == NULL)

g\_fm = new Network::FixedMessages;

Network::FixedMessages::getSingleton().loadConfig("server/messages\_fixed.xml");

messageHandlers().push\_back(this);

}

至此，Network::FixedMessages类被有机会实例化，构造函数中有：

FixedMessages::FixedMessages():

\_infomap(),

\_loaded(false)

{

new Resmgr();

Resmgr::getSingleton().initialize();

}

ServerConfig在各个组件（比如loginapp在loginapp.cpp）的类的定义文件中实例化。

ServerConfig g\_serverConfig;

KBE\_SINGLETON\_INIT(Loginapp);

我们再次回到loadConfig，里面的函数小跟一下就能读明白了，我们继续跟进主干流程。

### 生成组件的随机id：

下面的语句给组件生成一个随机id

g\_componentID = genUUID64();

### 解析main函数参数：

下面的语句解析主函数的参数：（比如设定指定的组件id，以及gus，我也还没了解到gus搞啥用的。。。不影响我们阅读整体流程，不细究）

parseMainCommandArgs(argc, argv);

### crash处理：

下面的语句进行crash处理：（不影响我们阅读整体流程，不细究）

char dumpname[MAX\_BUF] = {0}; \

kbe\_snprintf(dumpname, MAX\_BUF, "%"PRAppID, g\_componentID); \

KBEngine::exception::installCrashHandler(1, dumpname);

### 转向kbeMain：

下面的语句就是一个标准的main函数转向：

int retcode = -1; \

THREAD\_TRY\_EXECUTION; \

retcode = kbeMain(argc, argv); \

THREAD\_HANDLE\_CRASH; \

return retcode;

在kbemain.h中可以看到KBENGINE\_MAIN针对不同的平台有不同的定义。。。其实就是非win32平台没有crash处理。

## kbeMain：

kbeMain是在各个组件的main.cpp中定义的：（摘自loginapp）

{

ENGINE\_COMPONENT\_INFO& info = g\_kbeSrvConfig.getLoginApp();

return kbeMainT<Loginapp>(argc, argv, LOGINAPP\_TYPE, info.externalPorts\_min,

info.externalPorts\_max, info.externalInterface, 0, info.internalInterface);

}

### 转向kbeMainT：

第一句就是获取到各个组件相关的信息，然后流程再次转向，到了kbeMainT函数。

## kbeMainT的戏法：

kbeMainT是一个模板函数，根据各个组件的主类的不同，产生不同的代码（。。。我是不是有点过了）

前面的一些语句我们暂且不看（后面会需要看的），有设置环境变量的，设置密钥对的。可以看到kbeMainT的流程以各个组件的实例的run接口而终止。

### 组件实例的run接口：

大致看了几个组件的run函数，有的是调用ServerApp的run接口，有的是直接调用dispatcher的processXXX接口，总的来讲，就是依靠事件派发器来进行工作了。所以我们有必要去看看kbe的事件派发器了。

* loginapp组件的run接口：

bool Loginapp::run()

{

return ServerApp::run();

}

* machine组件的run接口：

bool Machine::run()

{

bool ret = true;

while(!this->dispatcher().hasBreakProcessing())

{

threadPool\_.onMainThreadTick();

this->dispatcher().processOnce(false);

networkInterface().processChannels(&MachineInterface::messageHandlers);

KBEngine::sleep(100);

};

return ret;

}

* baseapp组件的run接口：

bool Baseapp::run()

{

return EntityApp<Base>::run();

}

实际上EntityApp也是继承自ServerApp

### 事件派发器：

在kbeMainT中可以看到：

Network::EventDispatcher dispatcher;

这里构造了事件派发器，我们得看看它所谓的process到底干了些什么。

#### 事件派发器的工作：

在event\_dispatcher.cpp可以看到：

int EventDispatcher::processOnce(bool shouldIdle)

{

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING)

breakProcessing\_ = EVENT\_DISPATCHER\_STATUS\_RUNNING;

this->processTasks();

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING){

this->processTimers();

}

this->processStats();

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING){

return this->processNetwork(shouldIdle);

}

return 0;

}

这里我们有必要明白一些常识，对于从select，poll，epoll，iocp，kqueue的api，到boost的asio，ace，libevent，libev的库，这些网络i/o复用模型都是为了让我们监听nic上的事件，然后提交给相应的handler处理。他们除了工作方式导致的性能和编码方式有区别外，还有回调的时机的区别，iocp是对于资源的指定动作完成后回调，其他的unix族（kqueue是bsd的）接口都是资源对于指定动作准备好时回调。

所以我们要解读这个事件派发器得找到两个重点，如何产生事件，如何交付到应用处理。

EventDispatcher的processTasks可以稍微跟一下，发现它是处理所有“任务”的一个接口，但什么是任务，我们还不知道，不继续跟了。

processTimers， processStats也暂时不跟了。。。

#### 事件派发器的网络事件：

我们跟到processNetwork里面去：

int EventDispatcher::processNetwork(bool shouldIdle)

{

double maxWait = shouldIdle ? this->calculateWait() : 0.0;

return pPoller\_->processPendingEvents(maxWait);

}

这个pPoller是一个EventPoller的实例。EventPoller是一个抽象类(event\_poller.h/.cpp)，它目前有两个派生子类，SelectorPoller(poller\_select.h/.cpp)和EpollPoller(poller\_epoll.h/.cpp)，顾名思义，它们分别是利用select和epoll系统api进行异步i/o工作的。（在win32上面如果使用iocp的话性能应该是可以和epoll匹敌的，不过由于iocp和epoll工作方式不一样，我估计这是kbe里面没有在win32上使用iocp的原因，如果要将这两个工作方式抽象为一种，工作量估计比kbe本身小不了多少，如果是我的话，我会直接使用asio或者libevent，不过kbe作者为啥没用，可能就像redis的作者的解释一样，虽然我觉得那是很操蛋的解释，使用现有库的好处是显而易见的，如果碰巧像我这种对asio或者libevent有经验的，那这里kbe的网络底层我可以一掠而过，我也可以把更多的精力放在这个项目本身要解决的问题上。继续发牢骚可能）

select和epoll的工作方式一致，所以我们任选一个阅读都行，我倾向于使用epoll。

int EpollPoller::processPendingEvents(double maxWait)

{

const int MAX\_EVENTS = 10;

struct epoll\_event events[ MAX\_EVENTS ];

int maxWaitInMilliseconds = int(ceil(maxWait \* 1000));

#if ENABLE\_WATCHERS

g\_idleProfile.start();

#else

uint64 startTime = timestamp();

#endif

KBEConcurrency::onStartMainThreadIdling();

int nfds = epoll\_wait(epfd\_, events, MAX\_EVENTS, maxWaitInMilliseconds);

KBEConcurrency::onEndMainThreadIdling();

#if ENABLE\_WATCHERS

g\_idleProfile.stop();

spareTime\_ += g\_idleProfile.lastTime\_;

#else

spareTime\_ += timestamp() - startTime;

#endif

for (int i = 0; i < nfds; ++i)

{

if (events[i].events & (EPOLLERR|EPOLLHUP))

{

this->triggerError(events[i].data.fd);

}

else

{

if (events[i].events & EPOLLIN)

{

this->triggerRead(events[i].data.fd);

}

if (events[i].events & EPOLLOUT)

{

this->triggerWrite(events[i].data.fd);

}

}

}

return nfds;

}

大意就是对Poller内注册的文件描述符进行事件等待，然后对事件进行区分之后触发读(triggerRead)或者写(triggerWrite)的接口。我们跟一下这两个接口：

#### 网络可读事件：

bool EventPoller::triggerRead(int fd)

{

FDReadHandlers::iterator iter = fdReadHandlers\_.find(fd);

if (iter == fdReadHandlers\_.end())

{

return false;

}

iter->second->handleInputNotification(fd);

return true;

}

#### 网络可写事件：

//-------------------------------------------------------------------------------------

bool EventPoller::triggerWrite(int fd)

{

FDWriteHandlers::iterator iter = fdWriteHandlers\_.find(fd);

if (iter == fdWriteHandlers\_.end())

{

return false;

}

iter->second->handleOutputNotification(fd);

return true;

}

#### 事件处理：

可以看到就是对注册的文件描述符查找相应的输入输出处理接口（这里也指导了我们下一步的阅读方向，找到注册文件描述符的地方）。至此我们找到了事件如何产生。（其实我一直不习惯poll流的模型，我比较喜欢iocp的模型，虽然理论上来讲poll会给予应用层更多的变化点。打个不太形象的比喻，你让poll或者iocp给你准备三辆车。你给poll说完，poll稍后会告诉你：老爷，车备好了，在车库里，但具体有几辆我也不清楚，您自个去看看。你给iocp三把钥匙，iocp稍后会告诉你：老爷，三辆车准备好了，就停在门外。）

为了找到注册文件描述符和事件处理接口的流程，我们再次回到kbeMainT。映入眼帘的是这行代码：

Network::NetworkInterface networkInterface(&dispatcher,

extlisteningPort\_min, extlisteningPort\_max, extlisteningInterface,

channelCommon.extReadBufferSize, channelCommon.extWriteBufferSize,

(intlisteningPort != -1) ? htons(intlisteningPort) : -1, intlisteningInterface,

channelCommon.intReadBufferSize, channelCommon.intWriteBufferSize);

Network::NetworkInterface的构造函数：

NetworkInterface::NetworkInterface(Network::EventDispatcher \* pDispatcher,

int32 extlisteningPort\_min, int32 extlisteningPort\_max, const char \* extlisteningInterface,

uint32 extrbuffer, uint32 extwbuffer,

int32 intlisteningPort, const char \* intlisteningInterface,

uint32 intrbuffer, uint32 intwbuffer):

extEndpoint\_(),

intEndpoint\_(),

channelMap\_(),

pDispatcher\_(pDispatcher),

pExtensionData\_(NULL),

pExtListenerReceiver\_(NULL),

pIntListenerReceiver\_(NULL),

pDelayedChannels\_(new DelayedChannels()),

pChannelTimeOutHandler\_(NULL),

pChannelDeregisterHandler\_(NULL),

isExternal\_(extlisteningPort\_min != -1),

numExtChannels\_(0)

{

if(isExternal())

{

pExtListenerReceiver\_ = new ListenerReceiver(extEndpoint\_, Channel::EXTERNAL, \*this);

this->recreateListeningSocket("EXTERNAL", htons(extlisteningPort\_min), htons(extlisteningPort\_max),

extlisteningInterface, &extEndpoint\_, pExtListenerReceiver\_, extrbuffer, extwbuffer);

// 如果配置了对外端口范围， 如果范围过小这里extEndpoint\_可能没有端口可用了

if(extlisteningPort\_min != -1)

{

KBE\_ASSERT(extEndpoint\_.good() && "Channel::EXTERNAL: no available port, "

"please check for kbengine\_defs.xml!\n");

}

}

if(intlisteningPort != -1)

{

pIntListenerReceiver\_ = new ListenerReceiver(intEndpoint\_, Channel::INTERNAL, \*this);

this->recreateListeningSocket("INTERNAL", intlisteningPort, intlisteningPort,

intlisteningInterface, &intEndpoint\_, pIntListenerReceiver\_, intrbuffer, intwbuffer);

}

KBE\_ASSERT(good() && "NetworkInterface::NetworkInterface: no available port, "

"please check for kbengine\_defs.xml!\n");

pDelayedChannels\_->init(this->dispatcher(), this);

}

在recreateListeningSocket接口的代码中：

bool NetworkInterface::recreateListeningSocket(const char\* pEndPointName, uint16 listeningPort\_min, uint16 listeningPort\_max,

const char \* listeningInterface, EndPoint\* pEP, ListenerReceiver\* pLR, uint32 rbuffer,

uint32 wbuffer)

{

KBE\_ASSERT(listeningInterface && pEP && pLR);

if (pEP->good())

{

this->dispatcher().deregisterReadFileDescriptor( \*pEP );

pEP->close();

}

Address address;

address.ip = 0;

address.port = 0;

pEP->socket(SOCK\_STREAM);

if (!pEP->good())

{

ERROR\_MSG(fmt::format("NetworkInterface::recreateListeningSocket({}): couldn't create a socket\n",

pEndPointName));

return false;

}

/\*

pEP->setreuseaddr(true);

\*/

this->dispatcher().registerReadFileDescriptor(\*pEP, pLR);

找到了事件派发器注册文件描述符的地方，而注册的事件处理接口也就是这个ListenerReceiver(listener\_receiver.h/.cpp)。

跟到ListenerReceiver的handleInputNotification接口：

int ListenerReceiver::handleInputNotification(int fd)

{

int tickcount = 0;

while(tickcount ++ < 256)

{

EndPoint\* pNewEndPoint = endpoint\_.accept();

if(pNewEndPoint == NULL){

if(tickcount == 1)

{

WARNING\_MSG(fmt::format("PacketReceiver::handleInputNotification: accept endpoint({}) {}!\n",

fd, kbe\_strerror()));

this->dispatcher().errorReporter().reportException(

REASON\_GENERAL\_NETWORK);

}

break;

}

else

{

Channel\* pChannel = Network::Channel::ObjPool().createObject();

bool ret = pChannel->initialize(networkInterface\_, pNewEndPoint, traits\_);

if(!ret)

{

ERROR\_MSG(fmt::format("ListenerReceiver::handleInputNotification: initialize({}) is failed!\n",

pChannel->c\_str()));

pChannel->destroy();

Network::Channel::ObjPool().reclaimObject(pChannel);

return 0;

}

if(!networkInterface\_.registerChannel(pChannel))

{

ERROR\_MSG(fmt::format("ListenerReceiver::handleInputNotification: registerChannel({}) is failed!\n",

pChannel->c\_str()));

pChannel->destroy();

Network::Channel::ObjPool().reclaimObject(pChannel);

}

}

}

return 0;

}

如果你手工撸过epoll就会知道在监听套接口上如果有可读事件，则代表着有新的连接进来，

通常我们就是使用accept来接收这个新连接，然后注册到epoll上，但是kbe在这个ListenerReceiver中不是这么做的，它只是关心监听套接口的可读事件，然后将新的连接封装到它所谓的一个Channel中去了。（kbe用一个EndPoint来表征一个socket终端）。所以我们再跟进这个Channel看看。

bool Channel::initialize(NetworkInterface & networkInterface,

const EndPoint \* pEndPoint,

Traits traits,

ProtocolType pt,

PacketFilterPtr pFilter,

ChannelID id)

{

id\_ = id;

protocoltype\_ = pt;

traits\_ = traits;

pFilter\_ = pFilter;

pNetworkInterface\_ = &networkInterface;

this->pEndPoint(pEndPoint);

KBE\_ASSERT(pNetworkInterface\_ != NULL);

KBE\_ASSERT(pEndPoint\_ != NULL);

if(protocoltype\_ == PROTOCOL\_TCP)

{

if(pPacketReceiver\_)

{

if(pPacketReceiver\_->type() == PacketReceiver::UDP\_PACKET\_RECEIVER)

{

SAFE\_RELEASE(pPacketReceiver\_);

pPacketReceiver\_ = new TCPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

}

else

{

pPacketReceiver\_ = new TCPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

KBE\_ASSERT(pPacketReceiver\_->type() == PacketReceiver::TCP\_PACKET\_RECEIVER);

// UDP不需要注册描述符

pNetworkInterface\_->dispatcher().registerReadFileDescriptor(\*pEndPoint\_, pPacketReceiver\_);

// 需要发送数据时再注册

// pPacketSender\_ = new TCPPacketSender(\*pEndPoint\_, \*pNetworkInterface\_);

// pNetworkInterface\_->dispatcher().registerWriteFileDescriptor(\*pEndPoint\_, pPacketSender\_);

}

else

{

if(pPacketReceiver\_)

{

if(pPacketReceiver\_->type() == PacketReceiver::TCP\_PACKET\_RECEIVER)

{

SAFE\_RELEASE(pPacketReceiver\_);

pPacketReceiver\_ = new UDPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

}

else

{

pPacketReceiver\_ = new UDPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

KBE\_ASSERT(pPacketReceiver\_->type() == PacketReceiver::UDP\_PACKET\_RECEIVER);

}

pPacketReceiver\_->pEndPoint(pEndPoint\_);

if(pPacketSender\_)

pPacketSender\_->pEndPoint(pEndPoint\_);

startInactivityDetection((traits\_ == INTERNAL) ? g\_channelInternalTimeout :

g\_channelExternalTimeout,

(traits\_ == INTERNAL) ? g\_channelInternalTimeout / 2.f:

g\_channelExternalTimeout / 2.f);

return true;

}

可以看到在这个initialize接口中，新的EndPoint还是注册到了事件派发器中，只是处理的方式变为了PacketReceiver（虽然实现了UDP和TCP两种PacketReceiver，不过目前似乎只有TCPPacketReceiver被用到了）。

看看PacketReceiver的handleInputNotification：

int PacketReceiver::handleInputNotification(int fd)

{

if (this->processRecv(/\*expectingPacket:\*/true))

{

while (this->processRecv(/\*expectingPacket:\*/false))

{

/\* pass \*/;

}

}

return 0;

}

我们跟进processRecv(tcp\_packet\_receiver.cpp)：

bool TCPPacketReceiver::processRecv(bool expectingPacket)

{

Channel\* pChannel = getChannel();

KBE\_ASSERT(pChannel != NULL);

if(pChannel->isCondemn())

{

return false;

}

TCPPacket\* pReceiveWindow = TCPPacket::ObjPool().createObject();

int len = pReceiveWindow->recvFromEndPoint(\*pEndpoint\_);

if (len < 0)

{

TCPPacket::ObjPool().reclaimObject(pReceiveWindow);

PacketReceiver::RecvState rstate = this->checkSocketErrors(len, expectingPacket);

if(rstate == PacketReceiver::RECV\_STATE\_INTERRUPT)

{

onGetError(pChannel);

return false;

}

return rstate == PacketReceiver::RECV\_STATE\_CONTINUE;

}

else if(len == 0) // 客户端正常退出

{

TCPPacket::ObjPool().reclaimObject(pReceiveWindow);

onGetError(pChannel);

return false;

}

Reason ret = this->processPacket(pChannel, pReceiveWindow);

if(ret != REASON\_SUCCESS)

this->dispatcher().errorReporter().reportException(ret, pEndpoint\_->addr());

return true;

}

不难发现正常情况会调用processPacket，我们跟进(packet\_receiver.cpp)：

Reason PacketReceiver::processPacket(Channel\* pChannel, Packet \* pPacket)

{

if (pChannel != NULL)

{

pChannel->onPacketReceived(pPacket->length());

if (pChannel->pFilter())

{

return pChannel->pFilter()->recv(pChannel, \*this, pPacket);

}

}

return this->processFilteredPacket(pChannel, pPacket);

}

在没有filter的情况，流程会转向processFilteredPacket(tcp\_packet\_receiver.cpp)：

Reason TCPPacketReceiver::processFilteredPacket(Channel\* pChannel, Packet \* pPacket)

{

// 如果为None， 则可能是被过滤器过滤掉了(过滤器正在按照自己的规则组包解密)

if(pPacket)

{

pChannel->addReceiveWindow(pPacket);

}

return REASON\_SUCCESS;

}

我们跟进addReceiveWindow(channel.cpp)：

void Channel::addReceiveWindow(Packet\* pPacket)

{

bufferedReceives\_.push\_back(pPacket);

uint32 size = (uint32)bufferedReceives\_.size();

if(Network::g\_receiveWindowMessagesOverflowCritical > 0 && size > Network::g\_receiveWindowMessagesOverflowCritical)

{

if(this->isExternal())

{

if(Network::g\_extReceiveWindowMessagesOverflow > 0 &&

size > Network::g\_extReceiveWindowMessagesOverflow)

{

ERROR\_MSG(fmt::format("Channel::addReceiveWindow[{:p}]: external channel({}), receive window has overflowed({} > {}), Try adjusting the kbengine\_defs.xml->receiveWindowOverflow.\n",

(void\*)this, this->c\_str(), size, Network::g\_extReceiveWindowMessagesOverflow));

this->condemn();

}

else

{

WARNING\_MSG(fmt::format("Channel::addReceiveWindow[{:p}]: external channel({}), receive window has overflowed({} > {}).\n",

(void\*)this, this->c\_str(), size, Network::g\_receiveWindowMessagesOverflowCritical));

}

}

else

{

if(Network::g\_intReceiveWindowMessagesOverflow > 0 &&

size > Network::g\_intReceiveWindowMessagesOverflow)

{

WARNING\_MSG(fmt::format("Channel::addReceiveWindow[{:p}]: internal channel({}), receive window has overflowed({} > {}).\n",

(void\*)this, this->c\_str(), size, Network::g\_intReceiveWindowMessagesOverflow));

}

}

}

}

可以看到，正常情况下，一个包（客户端与服务端的一个有效链接上的负载）接收之后先被放到Channel的bufferedReceives队列。于是我们还需要找到何处处理这个包。

#### 处理接收数据：

我们可以在Channel的processPackets接口内找到处理bufferedReceives(channel.cpp)：

void Channel::processPackets(KBEngine::Network::MessageHandlers\* pMsgHandlers)

{

lastTickBytesReceived\_ = 0;

lastTickBytesSent\_ = 0;

if(pMsgHandlers\_ != NULL)

{

pMsgHandlers = pMsgHandlers\_;

}

if (this->isDestroyed())

{

ERROR\_MSG(fmt::format("Channel::processPackets({}): channel[{:p}] is destroyed.\n",

this->c\_str(), (void\*)this));

return;

}

if(this->isCondemn())

{

ERROR\_MSG(fmt::format("Channel::processPackets({}): channel[{:p}] is condemn.\n",

this->c\_str(), (void\*)this));

//this->destroy();

return;

}

if(pPacketReader\_ == NULL)

{

handshake();

}

try

{

BufferedReceives::iterator packetIter = bufferedReceives\_.begin();

for(; packetIter != bufferedReceives\_.end(); ++packetIter)

{

Packet\* pPacket = (\*packetIter);

pPacketReader\_->processMessages(pMsgHandlers, pPacket);

RECLAIM\_PACKET(pPacket->isTCPPacket(), pPacket);

}

}catch(MemoryStreamException &)

{

Network::MessageHandler\* pMsgHandler = pMsgHandlers->find(pPacketReader\_->currMsgID());

WARNING\_MSG(fmt::format("Channel::processPackets({}): packet invalid. currMsg=(name={}, id={}, len={}), currMsgLen={}\n",

this->c\_str()

, (pMsgHandler == NULL ? "unknown" : pMsgHandler->name)

, pPacketReader\_->currMsgID()

, (pMsgHandler == NULL ? -1 : pMsgHandler->msgLen)

, pPacketReader\_->currMsgLen()));

pPacketReader\_->currMsgID(0);

pPacketReader\_->currMsgLen(0);

condemn();

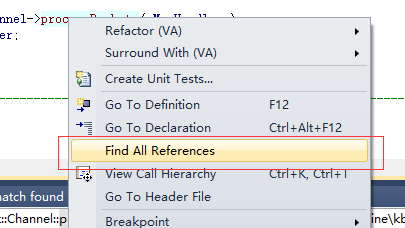
}

bufferedReceives\_.clear();

}

到这里我们又要弄清楚两个问题，这个接口何时被谁调用，调用又做了些什么，为了联通整个流程，我们还是先弄清楚这个接口在哪被谁调用。

通过vs的“Find All References”功能我顺利地找到了这个接口被调用的地方(network\_interface.cpp)：



void NetworkInterface::processChannels(KBEngine::Network::MessageHandlers\* pMsgHandlers)

{

ChannelMap::iterator iter = channelMap\_.begin();

for(; iter != channelMap\_.end(); )

{

Network::Channel\* pChannel = iter->second;

if(pChannel->isDestroyed())

{

++iter;

}

else if(pChannel->isCondemn())

{

++iter;

deregisterChannel(pChannel);

pChannel->destroy();

Network::Channel::ObjPool().reclaimObject(pChannel);

}

else

{

pChannel->processPackets(pMsgHandlers);

++iter;

}

}

}

同理，我们得找到processChannels被调用的地方。（几乎在每个继承自ServerApp的XXXApp的handleCheckStatusTick接口内都发现了类似下面的代码）：（loginapp.cpp）

void Loginapp::handleCheckStatusTick()

{

threadPool\_.onMainThreadTick();

networkInterface().processChannels(&LoginappInterface::messageHandlers);

pendingLoginMgr\_.process();

pendingCreateMgr\_.process();

}

#### 组件的定时器超时处理：

在XXXApp的handleTimeout的接口内我们找到了下面的代码：（loginapp.cpp）

void Loginapp::handleTimeout(TimerHandle handle, void \* arg)

{

switch (reinterpret\_cast<uintptr>(arg))

{

case TIMEOUT\_CHECK\_STATUS:

this->handleCheckStatusTick();

return;

default:

break;

}

ServerApp::handleTimeout(handle, arg);

}

根据ServerApp的父类TimerHandler的handleTimeout接口顺利地找到了下面的代码：（timer.inl）

template <class TIME\_STAMP>

void TimersT< TIME\_STAMP >::Time::triggerTimer()

{

if (!this->isCancelled())

{

state\_ = TIME\_EXECUTING;

pHandler\_->handleTimeout( TimerHandle( this ), pUserData\_ );

if ((interval\_ == 0) && !this->isCancelled())

{

this->cancel();

}

}

if (!this->isCancelled())

{

time\_ += interval\_;

state\_ = TIME\_PENDING;

}

}

找到调用triggerTimer的地方：（timer.inl）

template <class TIME\_STAMP>

int TimersT< TIME\_STAMP >::process(TimeStamp now)

{

int numFired = 0;

while ((!timeQueue\_.empty()) && (

timeQueue\_.top()->time() <= now ||

timeQueue\_.top()->isCancelled()))

{

Time \* pTime = pProcessingNode\_ = timeQueue\_.top();

timeQueue\_.pop();

if (!pTime->isCancelled())

{

++numFired;

pTime->triggerTimer();

}

if (!pTime->isCancelled())

{

timeQueue\_.push( pTime );

}

else

{

delete pTime;

KBE\_ASSERT( numCancelled\_ > 0 );

--numCancelled\_;

}

}

pProcessingNode\_ = NULL;

lastProcessTime\_ = now;

return numFired;

}

找到调用TimerHandler的process接口的地方：（event\_dispatcher.cpp，是不是感觉这个文件很熟悉。。。）

void EventDispatcher::processTimers()

{

numTimerCalls\_ += pTimers\_->process(timestamp());

}

***\*\*NEW in version 0.0.2\*\*：***

为什么EventDispatcher::processTimer内会调用XXXapp的handleTimeout接口，这个要在我们前面漏掉的一个接口(XXXapp::initialize)里面找寻一下。（我发现这一点的思考过程是这样的：既然超时后之后被调用的接口，那肯定是在此之前注册了计时器，可以看到EventDispatcher::addTimer接口就是向EventDispatcher::pTimers\_内注册计时器的接口，于是乎在XXXapp的initialize/initializeBegin/initializeEnd中看到这个接口的调用，这里摘自loginapp.cpp）：（kbe的Timer/Time也有的可写，不过我们现在先不深究）

bool Loginapp::initializeEnd()

{

// 添加一个timer， 每秒检查一些状态

loopCheckTimerHandle\_ = this->dispatcher().addTimer(1000000 / 50, this,

reinterpret\_cast<void \*>(TIMEOUT\_CHECK\_STATUS));

return true;

}

最终终于找到了我们失散多年的整体流程：（event\_dispatcher.cpp）

int EventDispatcher::processOnce(bool shouldIdle)

{

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING)

breakProcessing\_ = EVENT\_DISPATCHER\_STATUS\_RUNNING;

this->processTasks();

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING){

this->processTimers();

}

this->processStats();

if(breakProcessing\_ != EVENT\_DISPATCHER\_STATUS\_BREAK\_PROCESSING){

return this->processNetwork(shouldIdle);

}

return 0;

}

其实上面的这个过程有点小艰辛，花了好几个小时才完工。。。其实看到Channel::addReceiveWindow就应该想到这里的，因为这里的bufferedReceives的push\_back操作没有加锁（stl的vector是非线程安全的），所以他应该是和EventDispatcher::processNetwork接口在同一个线程的同步调用流程上，如此一来就只可能是EventDispatcher::processTasks，EventDispatcher::processTimers，EventDispatcher::processStats中的一个了。

我们接着回到之前的包处理的分支流程（就是Channel的processPackets接口）。

在packet\_reader.cpp内我们可以找到PacketReader::processMesages接口（代码太长，不贴了）。最终我们可以看到消息对应的handler的处理接口调用：

pMsgHandler->handle(pChannel\_, \*pFragmentStream\_);

建立消息对应的handler的映射在XXXapp\_interface.h中（各种宏，看的头都大了，留到后面各个组件的单独消息分析中再说， ***注：version 0.0.2中以loginapp进行了解读***）

至此，我们可以得出一个kbengine底层的大致流程的轮廓。

其实kbe里面还有很多“轮子”可以拿出来细说，像是单例，定时器，内存池，线程池，以及用来通讯的序列化和反序列化（MemoryStream）机制。。。。。。

总得来讲，大体因为底层网络（select，epoll）的异步回调机制，使得整个流程都有点小乱的感觉。“轮子”有点多，不利于利用现有的知识。一个高性能的c/c++项目几乎总会用到一些语言的 奇淫技巧，不过对于分布式的项目，既然我们已经做好了不追求单点高性能的心理准备，就不应该再为了追求性能而损失可读性。在单点性能可接受的程度上提高代码的可读性，提高整体的水平扩展能力，才是分布式项目的正道。

***\*\*NEW in version 0.0.2\*\****

负载消息与处理handler映射的建立：

在0.0.1中我们分析到了processMessages这里会调用对应的消息handler，但这个映射的建立过程我们还没分析，这次我们就一探究竟。（直接放在loginapp中叙述）

## 组件：

***\*\*NEW in version 0.0.4\*\****

### 小谈单例：

有人可能注意到了，kbe里面好多单例。。。Resmgr、ServerConfig、FixedMessages。。。

其实kbe的单例还算靠谱的（kbe的单例是需要显示构造一次的，虽然不是线程安全的，不过如果多次调用构造函数将会触发断言），还有一种实现的单例，是在getInstance的时候每次检查，然后再构造，使用静态阅读的方式真的很难找到构造时机（我绝对没说cocos2d-x是这么搞的）。

单例这个东西用一点可能很优雅，用多了真的很烦，我个人建议能不用的地方就不要用。我所了解的并且自己尝试过的化解方式有：

1. 使用单例hub，用一个单例来汇流其他也可能需要用的单例。比如在kbe中其实可以将ServerConfig/Resmgr对于组件App，FixedMessages对于EventDispatcher等使用对象组合（object composition），而不是都使用单例进行对象聚合（object aggregation）。
2. 把可能需要用到的单例使用参数传递。这种需要比较高的前期设计功底，不然后面为了将一个参数传到需要的地方可能要传很多层，这会增加代码理解上的复杂度，不过如果设计地比较“扁平”的话，这种情况可以得到比较好的缓解。

#### 单例在哪构造：

我们再来看看kbe中的那些单例都在哪被构造了。

##### ServerConfig：

ServerConfig涉及到服务端每个组件的各种配置选项，比如数据库访问（后面想到或者遇到再补充。。。嗯，是的）。它的构造在 组件名.cpp 中，比如loginapp就在loginapp.cpp，machine就在machine.cpp中，loginapp的如下（server/loginapp/loginapp.cpp）：

ServerConfig g\_serverConfig;

KBE\_SINGLETON\_INIT(Loginapp);

它的初始化（配置）工作主要由loadConfig接口完成，如下（lib/server/kbemain.h）：

inline void loadConfig()

{

Resmgr::getSingleton().initialize();

// "../../res/server/kbengine\_defs.xml"

g\_kbeSrvConfig.loadConfig("server/kbengine\_defs.xml");

// "../../../assets/res/server/kbengine.xml"

g\_kbeSrvConfig.loadConfig("server/kbengine.xml");

}

##### Resmgr：

Resmgr负责管理kbe的所有资源管理，比如资源路径，环境变量。Resmgr的构造地方如下（lib/network/fixed\_messages.cpp）：

FixedMessages::FixedMessages():

\_infomap(),

\_loaded(false)

{

new Resmgr();

Resmgr::getSingleton().initialize();

}

我们可以理解为FixedMessages构造的时候Resmgr就构造了。

Resmgr的初始化（配置）工作主要由initialize接口完成，代码如上。

##### FixedMessages：

FixedMessages存储所有固定消息（有显示制定id的消息，当然，这并不表示非固定消息就没有id，也是有的，只是不是显示制定的，可以参看loginapp的“消息与handle映射的建立”一节）。

它的构造地方如下（lib/network/message\_handler.cpp）：

MessageHandlers::MessageHandlers():

msgHandlers\_(),

msgID\_(1),

exposedMessages\_()

{

g\_fm = Network::FixedMessages::getSingletonPtr();

if(g\_fm == NULL)

g\_fm = new Network::FixedMessages;

Network::FixedMessages::getSingleton().loadConfig("server/messages\_fixed.xml");

messageHandlers().*push\_back*(this);

}

意即MessageHandlers构造的时候，如果它还没构造，那就构造，如果构造了，那就没啥事（至于MessageHandlers是在哪构造的请参看loginapp的“消息与handle映射的建立”一节）。

它的初始化（配置）是由loadConfig接口来完成的，代码见上。

### machine

### loginapp

#### 消息与handler映射的建立：

***\*\*NEW in version 0.0.2\*\****

##### 两次包含xxx\_interface.h，实现声明和定义：

每个app组件的接口定义都在xxxapp\_interface.cpp中开始，代码如下：

#include "loginapp\_interface.h"

#define DEFINE\_IN\_INTERFACE

#define LOGINAPP

#include "loginapp\_interface.h"

namespace KBEngine{

namespace LoginappInterface{

//-------------------------------------------------------------------------------------

}

}

所有的戏法都是通过包含loginapp\_interface.h前后定义了DEFINE\_IN\_INTERFACE和LOGINAPP来完成的。第一次的包含就是各种变量，类的声明（当然也有一些类是声明类时使用类inline函数定义完成了，比如MESSAGE\_ARGS0/1/2......）。

我们看看loginapp\_interface.h中的代码：

##### 消息与handlers的存储

首先是这一句：

NETWORK\_INTERFACE\_DECLARE\_BEGIN(LoginappInterface)

此句展开的话声明和定义了Network::MessageHandlers messageHandlers（记住它们都在LoginappInterface命名空间内），展开宏之后的代码看起来像这样（是的，你的眼睛是好的，没有}闭合）：

声明：

namespace LoginappInterface {

extern Network::MessageHandlers messageHandlers;

定义：

namespace LoginappInterface {

Network::MessageHandlers messageHandlers;

##### 消息与handle建立映射

然后是这一句：

LOGINAPP\_MESSAGE\_DECLARE\_ARGS0(importClientMessages, NETWORK\_FIXED\_MESSAGE)

此句展开的话分明声明和定义了一个importClientMessagesLoginappMessagehandler0的类，这个类继承自Network::MessageHandler，这里就是实现了handle的虚函数接口；声明和定义了importClientMessagesLoginappMessagehandler0的一个名为importClientMessages的全局变量；声明和定义了importClientMessagesArgs0的类，这个类继承自Network::MessageArgs。我们一个个地分析一下：

首先展开下面的宏：

LOGINAPP\_MESSAGE\_DECLARE\_ARGS0(importClientMessages, NETWORK\_FIXED\_MESSAGE)

之后是这样：

#define LOGINAPP\_MESSAGE\_DECLARE\_ARGS0(NAME, MSG\_LENGTH) \

LOGINAPP\_MESSAGE\_HANDLER\_ARGS0(NAME) \

NETWORK\_MESSAGE\_DECLARE\_ARGS0(Loginapp, NAME, \

NAME##LoginappMessagehandler0, MSG\_LENGTH)

展开LOGINAPP\_MESSAGE\_HANDLER\_ARGS0(NAME)之后分别得到importClientMessagesLoginappMessagehandler0的声明和定义：

声明：

class importClientMessagesLoginappMessagehandler0 : public Network::MessageHandler

{

public:

virtual void handle(Network::Channel\* pChannel, KBEngine::MemoryStream& s);

};

定义：

void importClientMessagesLoginappMessagehandler0::handle(Network::Channel\* pChannel, KBEngine::MemoryStream& s)

{

KBEngine::Loginapp::getSingleton().importClientMessages(pChannel);

}

（handle/handler，傻傻分不清楚。。。这里的handle是xxxApp中真正用来处理这个消息的接口，而这里的handler提供一个中间层的作用，集中处理一些通用的工作，可以将耦合减少一点）

上面完成了相当于是importClientMessages消息的handler的声明和定义，下面则将这个类实例化之后添加到messageHandlers：

#define NETWORK\_MESSAGE\_DECLARE\_ARGS0(DOMAIN, NAME, MSGHANDLER, \

MSG\_LENGTH) \

NETWORK\_MESSAGE\_HANDLER(DOMAIN, NAME, MSGHANDLER, MSG\_LENGTH, 0)\

MESSAGE\_ARGS0(NAME) \

展开NETWORK\_MESSAGE\_HANDLER(DOMAIN, NAME, MSGHANDLER, MSG\_LENGTH, 0)之后得到importClientMessages的handler类（importClientMessagesLoginappMessagehandler0）的名为importClientMessages的全局变量（不过欣慰的是他们都在各自的XXXInterface命名空间内）。

声明：

extern const importClientMessagesLoginappMessagehandler0& importClientMessages;

定义：

importClientMessagesLoginappMessagehandler0\* pimportClientMessages = static\_cast<importClientMessagesLoginappMessagehandler0\*>(messageHandlers.*add*("Loginapp::importClientMessages",new importClientMessagesArgs0, NETWORK\_FIXED\_MESSAGE, new importClientMessagesLoginappMessagehandler0);

const importClientMessagesLoginappMessagehandler0& importClientMessages = \*pimportClientMessages;

下面的MESSAGE\_ARGS0(NAME)展开后对importClientMessagesArgs0进行了声明和定义（其他它声明的时候就已经完成了全部的定义），声明的时候就是个空语句：

声明兼定义：

class importClientMessagesArgs0 : public Network::MessageArgs

{

public:

importClientMessagesArgs0() :Network::MessageArgs() {}

~importClientMessagesArgs0() {}

static void staticAddToBundle(Network::Bundle& s)

{

}

static void staticAddToStream(MemoryStream& s)

{

}

virtual int32 dataSize(void)

{

return 0;

}

virtual void addToStream(MemoryStream& s)

{

}

virtual void createFromStream(MemoryStream& s)

{

}

};

唯一需要小注意一下的就是importClientMessagesArgs0的声明（兼定义）是和importClientMessagesLoginappMessagehandler0的实例的声明和定义是错开的，因为后者实例化添加到messageHandlers的时候需要new一个importClientMessagesArgs0的实例。

##### 流程的伪代码

稍微整理一下之后，使用LOGINAPP\_MESSAGE\_HANDLER\_ARGSn建立一个消息到handler的映射的代码很像是这样：

声明：（第一次包含loginapp\_interface.h产生的代码）

class importClientMessagesLoginappMessagehandler0 : public Network::MessageHandler

{

public:

virtual void handle(Network::Channel\* pChannel, KBEngine::MemoryStream& s);

};

extern const importClientMessagesLoginappMessagehandler0& importClientMessages;

class importClientMessagesArgs0 : public Network::MessageArgs

{

public:

importClientMessagesArgs0() :Network::MessageArgs() {}

~importClientMessagesArgs0() {}

static void staticAddToBundle(Network::Bundle& s)

{

}

static void staticAddToStream(MemoryStream& s)

{

}

virtual int32 dataSize(void)

{

return 0;

}

virtual void addToStream(MemoryStream& s)

{

}

virtual void createFromStream(MemoryStream& s)

{

}

};

定义：（定义DEFINE\_IN\_INTERFACE和LOGINAPP之后第二次包含loginapp\_interface.h产生的代码）

void importClientMessagesLoginappMessagehandler0::handle(Network::Channel\* pChannel, KBEngine::MemoryStream& s)

{

KBEngine::Loginapp::getSingleton().importClientMessages(pChannel);

}

importClientMessagesLoginappMessagehandler0\* pimportClientMessages

= static\_cast<importClientMessagesLoginappMessagehandler0\*>(messageHandlers.*add*("Loginapp::importClientMessages",

new importClientMessagesArgs0,

NETWORK\_FIXED\_MESSAGE,

new importClientMessagesLoginappMessagehandler0);

const importClientMessagesLoginappMessagehandler0& importClientMessages = \*pimportClientMessages;

***\*\*New in version 0.0.4\*\****

##### 消息id：固定消息与非固定消息

要接着v0.0.3的分析继续写，回过头来要看之前写的东西说实话自己都有点难以理解。。。不过出于幸运或者努力，总算是看懂了;-(，读源代码（感觉特别是C++）本来就不是件容易的事，所以读源代码一定要做好长期战斗的准备。

上面我们分析到了，其实一个消息，就是由这样一个宏来和它的handle建立链接的：

LOGINAPP\_MESSAGE\_DECLARE\_ARGS0(importClientMessages, NETWORK\_FIXED\_MESSAGE)

通过上面的分析，我们得知，实际上建立消息和handle映射，起到核心作用的接口是messageHandlers.add(xxx, xxxx)，所以我们跟进去看看（lib/network/message\_handler.cpp）：

MessageHandler\* MessageHandlers::add(std::*string* ihName, MessageArgs\* args,

int32 msgLen, MessageHandler\* msgHandler)

{

if(msgID\_ == 1)

{

//printf("\n------------------------------------------------------------------\n");

//printf("KBEMessage\_handlers begin:\n");

}

//bool isfixedMsg = false;

FixedMessages::MSGInfo\* msgInfo = FixedMessages::getSingleton().isFixed(ihName.*c\_str*());

if(msgInfo == NULL)

{

while(true)

{

if(FixedMessages::getSingleton().isFixed(msgID\_))

{

msgID\_++;

//isfixedMsg = true;

}

else

{

break;

}

};

msgHandler->msgID = msgID\_++;

}

else

{

msgHandler->msgID = msgInfo->msgid;

}

msgHandler->name = ihName;

msgHandler->pArgs = args;

msgHandler->msgLen = msgLen;

msgHandler->exposed = false;

msgHandler->pMessageHandlers = this;

msgHandler->onInstall();

msgHandlers\_[msgHandler->msgID] = msgHandler;

if(msgLen == NETWORK\_VARIABLE\_MESSAGE)

{

//printf("\tMessageHandlers::add(%d): name=%s, msgID=%d, size=Variable.\n",

// (int32)msgHandlers\_.size(), ihName.c\_str(), msgHandler->msgID);

}

else

{

if(msgLen == 0)

{

msgHandler->msgLen = args->dataSize();

if(msgHandler->type() == NETWORK\_MESSAGE\_TYPE\_ENTITY)

{

msgHandler->msgLen += sizeof(ENTITY\_ID);

}

}

//printf("\tMessageHandlers::add(%d): name=%s, msgID=%d, size=Fixed(%d).\n",

// (int32)msgHandlers\_.size(), ihName.c\_str(), msgHandler->msgID, msgHandler->msgLen);

}

//if(isfixedMsg)

// printf("\t\t!!!message is fixed.!!!\n");

return msgHandler;

}

大意可以理解为，首先看看消息名称是不是一个固定消息，我们跟进去看看（lib/network/fixed\_messages.cpp）：

FixedMessages::MSGInfo\* FixedMessages::isFixed(const char\* msgName)

{

MSGINFO\_MAP::*iterator* iter = \_infomap.*find*(msgName);

if(iter != \_infomap.*end*())

{

MSGInfo\* infos = &iter->second;

return infos;

}

return NULL;

}

//-------------------------------------------------------------------------------------

bool FixedMessages::isFixed(MessageID msgid)

{

MSGINFO\_MAP::*iterator* iter = \_infomap.*begin*();

while (iter != \_infomap.*end*())

{

FixedMessages::MSGInfo& infos = iter->second;

if(infos.msgid == msgid)

return true;

++iter;

}

return false;

}

###### 固定消息

通过通读FixedMessages（fixed\_message.h/.cpp）可以看到这个\_infomap是在loadConfig中建立的，这个\_infomap就是所谓的固定消息（fixed message）与其id的映射表。loadConfig就是检视server/messages\_fixed.xml，将其中的消息名称与其id关联建立这个映射表。我们继续接着看MessageHandlers::add接口。

###### 非固定消息

对于isFixed为假的消息（非固定消息），则为其生成一个id（随着调用add的次序依次递增），这个id是在MessageHandlers类中唯一的，而每个组件的MessageHandlers又是处于自己的命名空间内，所以当出现某个组件的非固定消息时，则会为其生成单一组件内唯一的id（但这个id并不是所有组件内唯一的）。于是可能出现这种情况，Loginapp::xxxx与Dbmgr::yyyy都是非固定消息，但他们却有着同样的消息id，此时若有其他组件发送其中任一消息给其他组件，接受消息的组件将无法识别到底是Loginapp::xxxx或者是Dbmgr::yyyy。当然，只要我们将非固定消息发送给所属的组件，则不会有问题（上例中任何组件将Loginapp::xxxx发送给loginapp都是不会出乱子的）。

#### loginapp概述：

Loginapp组件主要用来处理账户登录/注册的业务，嗯，是的。

### dbmgr

#### dbmgr概述：

dbmgr组件主要负责数据库相关的事务，比如：账户登录/注册事务；账户充值；

### baseapp

### baseappmgr

### cellapp

### cellappmgr

## 脚本逻辑层：

# 各种工具：