# 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](mailto: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

基于kbengine 0.4.20

（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。

# 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

intKBENGINE\_MAIN(intargc, char\* argv[])

{

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

returnkbeMainT<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(intargc, char\* argv[]); \

intmain(intargc, 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); \

intretcode = -1; \

THREAD\_TRY\_EXECUTION; \

retcode = kbeMain(argc, argv); \

THREAD\_HANDLE\_CRASH; \

returnretcode; \

} \

稍微整理一下之后main函数看起来很像是这个样子：

intkbeMain(intargc, char\* argv[]);

intmain(intargc, char\* argv[])

{

loadConfig();

g\_componentID = genUUID64();

parseMainCommandArgs(argc, argv);

chardumpname[MAX\_BUF] = {0};

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

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

intretcode = -1;

THREAD\_TRY\_EXECUTION;

retcode = kbeMain(argc, argv);

THREAD\_HANDLE\_CRASH;

return (retcode);

}

intkbeMain(intargc, 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）

inlinevoidloadConfig()

{

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：

#defineg\_kbeSrvConfigServerConfig::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"

#defineDEFINE\_IN\_INTERFACE

#defineLOGINAPP

#include"loginapp\_interface.h"

xxx\_interface.h中，有这样的代码：

#ifdefined(DEFINE\_IN\_INTERFACE)

#undefKBE\_LOGINAPP\_INTERFACE\_H

#endif

#ifndefKBE\_LOGINAPP\_INTERFACE\_H

#defineKBE\_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

#defineNETWORK\_INTERFACE\_DECLARE\_BEGIN(INAME) \

namespace INAME \

{ \

externNetwork::MessageHandlersmessageHandlers; \

#else

#defineNETWORK\_INTERFACE\_DECLARE\_BEGIN(INAME) \

namespace INAME \

{ \

Network::MessageHandlersmessageHandlers; \

#endif

#defineNETWORK\_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 = newNetwork::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)

{

newResmgr();

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。

ServerConfigg\_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：

intretcode = -1; \

THREAD\_TRY\_EXECUTION; \

retcode = kbeMain(argc, argv); \

THREAD\_HANDLE\_CRASH; \

returnretcode;

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();

returnkbeMainT<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：

boolLoginapp::run()

{

returnServerApp::run();

}

* machine组件的run接口：

boolMachine::run()

{

boolret = true;

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

{

threadPool\_.onMainThreadTick();

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

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

KBEngine::sleep(100);

};

returnret;

}

* baseapp组件的run接口：

boolBaseapp::run()

{

returnEntityApp<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::EventDispatcherdispatcher;

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：

intEventDispatcher::processOnce(boolshouldIdle)

{

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){

returnthis->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：

intEventDispatcher::processNetwork(boolshouldIdle)

{

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

returnpPoller\_->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。

intEpollPoller::processPendingEvents(doublemaxWait)

{

constintMAX\_EVENTS = 10;

struct epoll\_event events[ MAX\_EVENTS ];

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

#if ENABLE\_WATCHERS

g\_idleProfile.start();

#else

uint64startTime = timestamp();

#endif

KBEConcurrency::onStartMainThreadIdling();

intnfds = 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 (inti = 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);

}

}

}

returnnfds;

}

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：

boolEventPoller::triggerRead(intfd)

{

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

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

{

returnfalse;

}

iter->second->handleInputNotification(fd);

returntrue;

}

#### 网络可写事件：

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

boolEventPoller::triggerWrite(intfd)

{

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

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

{

returnfalse;

}

iter->second->handleOutputNotification(fd);

returntrue;

}

#### 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::NetworkInterfacenetworkInterface(&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,

int32extlisteningPort\_min, int32extlisteningPort\_max, constchar \* extlisteningInterface,

uint32extrbuffer, uint32extwbuffer,

int32intlisteningPort, constchar \* intlisteningInterface,

uint32intrbuffer, uint32intwbuffer):

extEndpoint\_(),

intEndpoint\_(),

channelMap\_(),

pDispatcher\_(pDispatcher),

pExtensionData\_(NULL),

pExtListenerReceiver\_(NULL),

pIntListenerReceiver\_(NULL),

pDelayedChannels\_(newDelayedChannels()),

pChannelTimeOutHandler\_(NULL),

pChannelDeregisterHandler\_(NULL),

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

numExtChannels\_(0)

{

if(isExternal())

{

pExtListenerReceiver\_ = newListenerReceiver(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\_ = newListenerReceiver(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：

boolNetworkInterface::recreateListeningSocket(constchar\* pEndPointName, uint16listeningPort\_min, uint16listeningPort\_max,

constchar \* listeningInterface, EndPoint\* pEP, ListenerReceiver\* pLR, uint32rbuffer,

uint32wbuffer)

{

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

if (pEP->good())

{

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

pEP->close();

}

Addressaddress;

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));

returnfalse;

}

/\*

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：

intListenerReceiver::handleInputNotification(intfd)

{

inttickcount = 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();

boolret = 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。

boolChannel::initialize(NetworkInterface&networkInterface,

constEndPoint \* pEndPoint,

Traitstraits,

ProtocolTypept,

PacketFilterPtrpFilter,

ChannelIDid)

{

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\_ = newTCPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

}

else

{

pPacketReceiver\_ = newTCPPacketReceiver(\*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\_ = newUDPPacketReceiver(\*pEndPoint\_, \*pNetworkInterface\_);

}

}

else

{

pPacketReceiver\_ = newUDPPacketReceiver(\*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);

returntrue;

}

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：

intPacketReceiver::handleInputNotification(intfd)

{

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

{

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

{

/\* pass \*/;

}

}

return 0;

}

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

boolTCPPacketReceiver::processRecv(boolexpectingPacket)

{

Channel\* pChannel = getChannel();

KBE\_ASSERT(pChannel != NULL);

if(pChannel->isCondemn())

{

returnfalse;

}

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

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

if (len< 0)

{

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

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

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

{

onGetError(pChannel);

returnfalse;

}

returnrstate == PacketReceiver::RECV\_STATE\_CONTINUE;

}

elseif(len == 0) // 客户端正常退出

{

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

onGetError(pChannel);

returnfalse;

}

Reasonret = this->processPacket(pChannel, pReceiveWindow);

if(ret != REASON\_SUCCESS)

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

returntrue;

}

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

ReasonPacketReceiver::processPacket(Channel\* pChannel, Packet \* pPacket)

{

if (pChannel != NULL)

{

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

if (pChannel->pFilter())

{

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

}

}

returnthis->processFilteredPacket(pChannel, pPacket);

}

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

ReasonTCPPacketReceiver::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);

}

returnREASON\_SUCCESS;

}

我们跟进addReceiveWindow(channel.cpp)：

voidChannel::addReceiveWindow(Packet\* pPacket)

{

bufferedReceives\_.push\_back(pPacket);

uint32size = (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)：

voidChannel::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::iteratorpacketIter = 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();

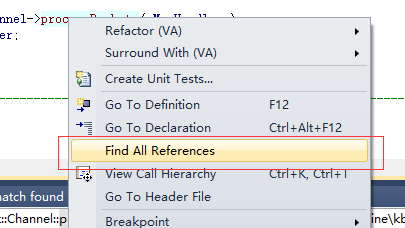
}

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)：



voidNetworkInterface::processChannels(KBEngine::Network::MessageHandlers\* pMsgHandlers)

{

ChannelMap::iteratoriter = channelMap\_.begin();

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

{

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

if(pChannel->isDestroyed())

{

++iter;

}

elseif(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）

voidLoginapp::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）

voidLoginapp::handleTimeout(TimerHandlehandle, void \* arg)

{

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

{

caseTIMEOUT\_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<classTIME\_STAMP>

voidTimersT<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<classTIME\_STAMP>

intTimersT<TIME\_STAMP>::process(TimeStampnow)

{

intnumFired = 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

{

deletepTime;

KBE\_ASSERT(numCancelled\_> 0 );

--numCancelled\_;

}

}

pProcessingNode\_ = NULL;

lastProcessTime\_ = now;

returnnumFired;

}

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。。。）

voidEventDispatcher::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）

boolLoginapp::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));

returntrue;

}

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

intEventDispatcher::processOnce(boolshouldIdle)

{

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){

returnthis->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：

## machine

## loginapp

### 消息与handler映射的建立：

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

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"

#defineDEFINE\_IN\_INTERFACE

#defineLOGINAPP

#include"loginapp\_interface.h"

namespaceKBEngine{

namespaceLoginappInterface{

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

}

}

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：

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，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：

namespaceLoginappInterface {

extern Network::MessageHandlers messageHandlers;

定义：

namespaceLoginappInterface {

Network::MessageHandlers messageHandlers;

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：

#defineLOGINAPP\_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：

classimportClientMessagesLoginappMessagehandler0 : public Network::MessageHandler

{

public:

virtualvoidhandle(Network::Channel\* pChannel, KBEngine::MemoryStream&s);

};

定义：

voidimportClientMessagesLoginappMessagehandler0::handle(Network::Channel\* pChannel, KBEngine::MemoryStream&s)

{

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

}

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：

#defineNETWORK\_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。

声明：

externconstimportClientMessagesLoginappMessagehandler0&importClientMessages;

定义：

importClientMessagesLoginappMessagehandler0\* pimportClientMessages =static\_cast<importClientMessagesLoginappMessagehandler0\*>(messageHandlers.*add*("Loginapp::importClientMessages",new importClientMessagesArgs0, NETWORK\_FIXED\_MESSAGE, newimportClientMessagesLoginappMessagehandler0);

constimportClientMessagesLoginappMessagehandler0&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：

classimportClientMessagesArgs0 : public Network::MessageArgs

{

public:

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

~importClientMessagesArgs0() {}

staticvoidstaticAddToBundle(Network::Bundle&s)

{

}

staticvoidstaticAddToStream(MemoryStream&s)

{

}

virtual int32 dataSize(void)

{

return 0;

}

virtualvoidaddToStream(MemoryStream&s)

{

}

virtualvoidcreateFromStream(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。

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）

classimportClientMessagesLoginappMessagehandler0 : public Network::MessageHandler

{

public:

virtualvoidhandle(Network::Channel\* pChannel, KBEngine::MemoryStream&s);

};

externconstimportClientMessagesLoginappMessagehandler0&importClientMessages;

classimportClientMessagesArgs0 : public Network::MessageArgs

{

public:

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

~importClientMessagesArgs0() {}

staticvoidstaticAddToBundle(Network::Bundle&s)

{

}

staticvoidstaticAddToStream(MemoryStream&s)

{

}

virtual int32 dataSize(void)

{

return 0;

}

virtualvoidaddToStream(MemoryStream&s)

{

}

virtualvoidcreateFromStream(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）

voidimportClientMessagesLoginappMessagehandler0::handle(Network::Channel\* pChannel, KBEngine::MemoryStream&s)

{

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

}

importClientMessagesLoginappMessagehandler0\* pimportClientMessages

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

newimportClientMessagesArgs0,

NETWORK\_FIXED\_MESSAGE,

newimportClientMessagesLoginappMessagehandler0);

constimportClientMessagesLoginappMessagehandler0&importClientMessages = \*pimportClientMessages;

## dbmgr

## caseapp

## baseappmgr

## cellapp

## cellappmgr

# 脚本逻辑层：

# 各种工具：