**corotinue框架使用文档**

**bbwang，2015/07/29**

目录

[1.本框架的目的 2](#_Toc425926837)

[2.框架内部的功能 2](#_Toc425926838)

[3.使用方法 2](#_Toc425926839)

[4.注意事项 5](#_Toc425926840)

# 1.本框架的目的

本框架在imo\_net和net\_manager的基础上进行再次封装，借助libco实现rpc的能力，便于开发。任何事物有好有坏，坏处就是付出性能代价。比老框架性能有所下降，差别不是很大。

# 2.框架内部的功能

1.session管理，默认管理serversession，如若要管理clientsession，需要在客户端心跳逻辑中添加。

2.组的概念。所有的session都挂在组里面。

3.session和组的选取支持不同的策略。

4.单次请求性能统计。

5.针对不同协议的请求进行处理，BinaryTranslater，CmdLineTranslater，ProtocTranslater(workchat使用，cmd在包头的第三个int位置)。

6.支持tcp和udp协议。

7.默认的数据包最大为1m，如若超包大小或是非法的包，会拒掉并断连接。

8.创建具体的msg，需要注册工厂，由工厂实施创建。

9.发包可指定包的大小，默认4500字节，如果需要更多空间，在结果的msg中指定(SetMaxSize())。

# 3.使用方法

按照如下流程可以使用本框架搭建一个服务：(基本与老框架风格一致)

**加载配置 --> 初始化资源(Option,NetManager,工厂) --> 初始化连接 -->loop event --> 资源回收(Option,NetManager,工厂) --> 退出进程**

注:

服务本身需要绑定一个唯一id

对于同种服务的连接，需要option，这些服务共用此option

对于同种服务的连接，他们之间的msg需要由工厂创建，这些服务共用此工厂

所有的连接由SessionManager建立

所有的业务处理在处理器中进行，不再需要在session中处理，因此本框架需要根据不同种类的服务实现对应的处理器类(Processor，单例)

简单的一个例子说明:

#include "libco\_wrap.h"

#include "msg.h"

#include "session\_manager.h"

#include "client\_processor.h"

#include "server\_processor.h"

#include "translater.h"

#include "option.h"

#include "factories.h"

string CLIENT\_SESSION;

string SERVER\_SESSION;

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

{

// =====================

// 加载配置

// =====================

Config config;

LoadCfg(config);

// =====================

// 初始化资源

// =====================

// 1. option

Option \*cli\_option\_ = new Option;

cli\_option\_->service\_name = config.sys\_name;

cli\_option\_->translater = BinaryTranslater::Instance();

cli\_option\_->processor = ClientProcessor::Instance();

Option \*srv\_option\_ = new Option;

srv\_option\_>service\_name = config.srv\_service\_name;

srv\_option\_>translater = BinaryTranslater::Instance();

srv\_option\_>processor = UdpClientProcessor::Instance();

// 2. net\_manager

Net\_Manager \* net\_manager\_ = new Net\_Manager;

// 3. 注册工厂

CLIENT\_SESSION = config.sys\_name;

SERVER\_SERVICE = config.srv\_service\_name;

MsgFactory \*cli\_factory\_ = new CliMsgFactory;

MsgFactory \*srv\_factory\_ = new SrvMsgFactory;

GlobalFactoryRegister(CLIENT\_SESSION.c\_str(), cli\_factory\_);

GlobalFactoryRegister(SERVER\_SERVICE.c\_str(), srv\_factory\_);

// =====================

// 初始化网络连接

// =====================

if (0 == config.sys\_id)

{

LOG(ERROR)("config have not specify system id. ");

return -1;

}

// todo: start net thread, handle event

if (0 != net\_manager\_->start())

{

LOG(ERROR)("[start] net\_manager\_ start failed.");

return -1;

}

SessionManager::Instance().net\_manager(net\_manager\_);

// 指定本服务的id，用于向服务端报告

SessionManager::Instance().BindId(config.sys\_id);

Server\_Info info;

info.server\_ip = config.sys\_ip;

info.server\_port = config.sys\_port;

info.conn\_time\_out = config.sys\_timeout;

info.heart\_beat\_interval = config.sys\_ttl;

info.max\_buf\_size = config.sys\_buf\_size;

info.reconnect\_interval = 2;

info.service\_name = config.sys\_name;

info.time\_out = config.sys\_timeout;

SessionManager::Instance().OpenAcceptor(info, cli\_option\_);

LOG(INFO)("[start] open\_serverservice");

const vector<Server\_Info> &list\_serv = config.srv\_infos;

for (vector<Server\_Info>::const\_iterator it=list\_serv.begin(); it!=list\_serv.end(); it++)

{

const Server\_Info &info = \*it;

SessionManager::Instance().ConnectServer(info, srv\_option\_);

LOG(INFO)("[start] connect status:%s succeed. max buf size:%u."

, info.print(), info.max\_buf\_size);

}

// 设置co的最大值，可以动态设定并更改

set\_max\_co\_num(config.sys\_max\_co\_num);

// =====================

// 进入事件主循环

// =====================

co\_run\_service(net\_manager\_);

// =====================

// 退出进程

// =====================

GlobalFactoryUnregister(CLIENT\_SESSION.c\_str(), cli\_factory\_);

GlobalFactoryUnregister(SERVER\_SERVICE.c\_str(), status\_factory\_);

return 0;

}

Class ClientProcessor : public Processor

{

public:

int ProcessData(const Param &param, Msg \*msg) const

{

switch (msg->GetCmd())

{

case CHANGE\_STATUS:

{

FrontMsg \*front\_msg = dynamic\_cast<FrontMsg \*>(msg);

// get result from server

Msg \* msg\_result = NULL;

// 如果是以hash方式访问后端，就要指定hash值

front\_msg->set\_hashid(uid);

int ret = GetResult(SERVER\_SERVICE, \*front\_msg, msg\_result);

// 此处务必要对ret做判断，你无法得知收到的msg\_result就是你想要的

if (0 != ret)

{

LOG(ERROR)("something error. err code:%d", ret);

return ret;

}

BackendMsg \*backend\_msg = dynamic\_cast<BackendMsg \*>(msg\_result);

// do something, then reply client

ret = Reply(param, \*front\_msg, \*backend\_msg);

}

break;

default:

LOG(ERROR)(" unsupported cmd:%u from client", msg->GetCmd());

break;

}

return 0;

}

};

# 4.注意事项

1.如果使用重定向，重定向要在初始化网络连接之前完成。否则在会有关闭连接的可能。(未知原因，待查)

2.性能时间统计不要在协程里面做(业务代码都是在协程里面)，否则会打日志crash。(未知原因，待查)

3.只有数据处理在协程中执行，其他的如：连接打开，连接关闭，连接异常等都不在协程中执行(在processor中执行的代码都是在协程之中)，框架内部会处理，无需在processor中捕获这些事件。

**如下代码可说明: co\_routine\_manager.cpp : 607**

switch ( ev->net\_event\_type )

{

case TYPE\_DATA:

{

Msg \*msg = CoroutineMgr::Instance().GetCurMsg();

// 此处对msg作判断，能进入到业务层的msg都是非空的，但不一定就是你想要的

// 可能是框架生成的SystemTimeoutMsg,SystemErrorMsg

if (!msg)

{

LOG(ERROR)("get cur co msg failed. handle:%u, coid:%u", ev->id, CoroutineMgr::Instance().GetCurCoId());

break;

}

cmd = msg->GetCmd();

LOG(DEBUG)("start task, name:%s, handle:%u, task id:%u, cmd:%u, remote\_addr:%s"

, psession\_param->service\_name, psession\_param->net\_id, co\_param->coid, msg->GetCmd(), remote\_addr.c\_str());

// 心跳

if (0 == cmd)

{

processor->KeepAlive(\*psession\_param, msg);

break;

}

processor->ProcessData(\*psession\_param, msg);

}break;

default:

{

LOG(WARN)("process error, unknown net\_event\_type:%d", ev->net\_event\_type);

}break;

}

4.协程框架内部的task id生成规则为32为整型(8bit固定id+24bit自增id)，固定id(libco\_src::service\_id,范围是[0-255],超出部分被截断)可由用户指定(set\_service\_id())，也可由框架随机生成。

5.框架的协程个数有限制，默认20000，用户可指定(set\_max\_co\_num()).