# 一.Framework

分布式程序成为当前互联网开发的主要开发模式，掌握跨平台多线程的分布式程序，成为必备技能。接下来完善跨平台统一网络库的使用说明。设计的基础工具:pthread,socket,标准C++等。

完善库分两步：第一步，模仿poco完善功能；第二步，添加C++11的线程版本。

服务器端，内部数据格式类似于：

#ApiName=onRecvDeferQuotation#instID=Ag(T+D)#sZipBuff=TA3LQJQAAAKRCANQuA==#

|  |  |  |  |
| --- | --- | --- | --- |
|  | POCO | U-framwork |  |
| 字符串 |  |  |  |
| 时间日期 |  |  |  |
| 日志 |  |  |  |
| 内存池 |  |  |  |
| File读写 |  |  |  |
| XML读写 |  |  |  |
| Json读写 |  |  |  |
| Zip文件操作 |  |  |  |
| 文本编码和转换 |  |  |  |
| 数据库访问(sqlite,mysql,odbc) |  |  |  |
| 进程管理 |  |  |  |
| 进程通信库 |  |  |  |
| 多线程框架（线程池，活动对象，工作队列） |  |  |  |
| 网络数据流编解码 |  |  |  |

## 1.基础库

### 字符串

### 时间处理

CGessDate gDate;

CGessTime gTime;

### 日志

### 怎么使用日志库

### 定时器

### 加入定时器

1. **class** MainCPMgr : **public** CProtocolCpMgr
2. {
3. **private**:
5. **class** CIfkTimer : **public** CGessTimer
6. {
7. **public**:
8. CIfkTimer() {}
9. **virtual** ~CIfkTimer() {}
10. **void** Bind(MainCPMgr\* p)
11. {
12. m\_pParent = p;
13. }
14. //虚函数，超时
15. **int** TimeOut(**const** string& ulKey, unsigned **long**& ulTmSpan)
16. {
17. **if** (0 != m\_pParent)
18. **return** m\_pParent->OnIfkTimeout(ulKey);
19. **return** -1;
20. }
21. //虚函数，取消定时器
22. **void** TimerCanceled(**const** string& ulKey)
23. {
24. }
25. **private**:
26. MainCPMgr\* m\_pParent;
27. };
29. **public**:
30. **int** OnIfkTimeout(**const** string& ulKey);
32. //定时器管理类
33. CGessTimerMgrIf \* m\_pGessTimerMgr;
34. CIfkTimer m\_oIfQTTimer;
35. }

#### 初始化定时器：

m\_oIfQTTimer.Bind(this);

//初始化定时器管理器/启动定时器

m\_pGessTimerMgr = CGessTimerMgrImp::Instance();

m\_pGessTimerMgr->Init(2);

#### 创建定时器：

m\_pGessTimerMgr->CreateTimer(&m\_oIfTDTimer, 8, "tradeRecord");

m\_pGessTimerMgr->CreateTimer(&m\_oIfQTTimer, 5, "quotation");

启动定时器:

m\_pGessTimerMgr->Start();

#### 在超时函数处理业务:

int MainCPMgr::OnIfkTimeout(const string& ulKey)

{

if (ulKey == "quotation")

{

CRLog(E\_APPINFO, "%s", "[行情定时器]");

}

else if (ulKey == "tradeRecord")

{

CRLog(E\_APPINFO, "%s", "[交易定时器]");

}

else if (ulKey == "Strategy")

{

CRLog(E\_APPINFO, "%s", "[策略定时器]");

}

return 0;//返回-1，定时器执行一次就删除，返回0，多次运行

}

#### 停止定时器：

void MainCPMgr::Stop()

{

//m\_pGessTimerMgr->DestroyTimer(&m\_oIfTDTimer, "tradeRecord");

m\_pGessTimerMgr->DestroyTimer(&m\_oIfQTTimer, "quotation");

m\_pGessTimerMgr->Stop();

}

### 文件读写

File

Xml

Json

Zip

### 文件编码和转换

数据库访问

线程池

对象池

### 线程

### 订阅器

### 监控机制

1. 怎么监视网络数据（子线程业务）

### 大容量

### 怎么增加队列（子线程功能块）

### 低延时

### 分类路由

### 订阅与通知机制(reactor模式)

### 分布式

自定义协议，协议的编解码，压缩，数据的汇集与分发路由

Paxco协议

### 高并发

跨平台库epoll模式，iocp模式（参考nginx,asio,nanomsg等等）

### 生产者/消费者框架

## 2.业务库

### 2.1数据定义

#### CSamplerPacket

业务:属于FramworkApSampler模块，自定义协议，行情数据类定义：代码信息，行情定义，K线数据定义，历史数据定义，tick数据定义，资讯数据定义.

具体业务为:登陆,注册，订阅，心跳，发送实时，历史数据，退出等。

**技术**：二进制读写。CSamplerPacket内部定义了一个缓冲块CMsgBlock(默认1024字节,长度字段默认为4字节)，CMsgBlock自带访问接口。

#### CTradePacket:public CPairPacket

#### CIpcPacket:public CPairPacket

#### CPacketLineReq:public CPacketRouteable

#### CIpcPacket:public CPairPacket

#### CBroadcastPacket:public CPairPacket

### 2.2 调用API

2.2.1同步API

2.2.2异步API

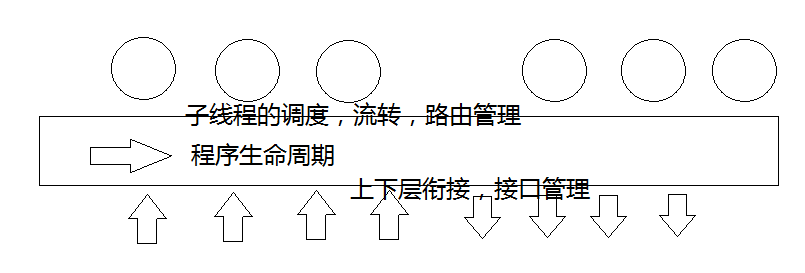
### 2.3协议定义与编解码

#### 2.3.1交易数据定义，同步API定义，协议定义和编解码

#### 2.3.2行情数据定义，异步API定义，协议定义和编解码

## 3.线程程序的开发

### 3.1主线程的构建



#### 3.1.1引入库

1. //引入库平台差异定义
2. //引入基础库（pthread,STL，不同平台网络API，soecket的引入）
3. #include "Gess.h"
4. //但是一般不需要明显引入，基本会被业务头文件包含进去，只需要包含业务引入，如下：
5. #include "Logger.h"
6. #include "WorkThread.h"
7. #include "ConfigImpl.h"

#### 3.1.2主程序不管理网络事件(用了第三方的网络库)

1. class MainCPMgr
2. {
3. }

#### 3.2.3主程序管理网络事件(主要使用模式)

网络程序一种是网络事件，一种接收数据。因而有4个网络事件接口，有一个报文转发接口，需要继承实现。

1. #include "BroadcastPacket.h" //广播数据定义与编解码
2. #include "IpcPacket.h" //自路由数据定义与编解码
3. **class** MainCPMgr : **public** CProtocolCpMgr
4. {
5. **virtual** **int** OnAccept(**const** unsigned **long**& ulKey,**const** string& sLocalIp, **int** nLocalPort, **const** string& sPeerIp, **int** nPeerPort) override;
6. **virtual** **int** OnConnect(**const** unsigned **long**& ulKey,**const** string& sLocalIp, **int** nLocalPort, **const** string& sPeerIp, **int** nPeerPort,**int** nFlag) override;
7. **virtual** **int** OnLogin( **const** unsigned **long**& ulKey,**const** string& sLocalIp, **int** nLocalPort, **const** string& sPeerIp, **int** nPeerPort,**int** nFlag) override;
8. **virtual** **int** OnClose( **const** unsigned **long**& ulKey,**const** string& sLocalIp, **int** nLocalPort, **const** string& sPeerIp, **int** nPeerPort) override;
9. //网络底层回调，收到数据，可以向应用层转发
10. **virtual** **int** Forward(CPacket &GessPacket,**const** unsigned **long**& ulKey) override;
11. }

#### 3.2.3主线程管理网络事件接口，自转发

#### 3.2.4主线程的数据业务路由与分发

#### 3.2.5主线程分发器，提供给业务(子线程)订阅

### 3.2子线程的构建

#### 3.2子线程队列的构建，入列，出列，消息通知机制

1. 头文件
2. //异步网络接口
3. //子线程
4. **class** CServiceHandler :**public** CConnectPointAsyn, **public** CWorkThread
5. {
6. **public**:
7. CServiceHandler(**void**);
8. ~CServiceHandler(**void**);
10. **int** Init(CConfig\* pConfig);
11. **int** Start();
12. **void** Stop();
13. **void** Finish();
14. **int** OnRecvPacket(CPacket &GessPacket){**return** 0;}
15. **int** SendPacket(CPacket &pkt);
16. **void** Bind(CConnectPointManager\* pCpMgr,**const** unsigned **long**& ulKey);
18. **private**:
19. //定义成员函数指针
20. **typedef** **int** (CServiceHandler::\*MFP\_PktHandleApi)(CSamplerPacket& pkt);
21. //报文命令字与报文处理成员函数映射结构
22. **typedef** **struct** tagPktCmd2Api
23. {
24. string sApiName; //报文ApiName或交易代码
25. MFP\_PktHandleApi pMemberFunc; //报文处理函数指针
26. } PktCmd2Api;
27. //报文命令字与报文处理成员函数映射表
28. **static** PktCmd2Api m\_PktCmd2Api[];
29. **int** RunPacketHandleApi(CSamplerPacket& pkt);
31. CCvgCpMgr\* m\_pCvgCpMgr;
32. unsigned **long** m\_ulKey;
33. CConfig\* m\_pCfg;
34. string m\_sNodeID;
36. std::deque<CSamplerPacket> m\_deqService;
37. CCondMutex m\_deqCondMutex;
39. **int** ThreadEntry();
40. **int** End();
42. **int** OnRecvLoginRsp(CSamplerPacket& pkt);
43. **int** OnRecvSubscripRsp(CSamplerPacket& pkt);
44. **int** OnRecvLogoutRsp(CSamplerPacket& pkt);
45. **int** OnRecvUnSubscripRsp(CSamplerPacket& pkt);
47. **int** OnRecvLogin(CSamplerPacket& pkt);
48. **int** OnRecvSubscrip(CSamplerPacket& pkt);
49. **int** OnRecvLogout(CSamplerPacket& pkt);
50. **int** OnRecvUnSubscrip(CSamplerPacket& pkt);
51. **int** OnRecvHello(CSamplerPacket& pkt);
52. };


56. CPP文件
58. #include "ServiceHandler.h"
59. #include "CvgCpMgr.h"//主线程管理器

62. //报文对应的处理成员函数配置表
63. CServiceHandler::PktCmd2Api CServiceHandler::m\_PktCmd2Api[] =
64. {
65. //请求ApiName 请求处理函数指针
66. {"80000001", &CServiceHandler::OnRecvLoginRsp},
67. {"80000002", &CServiceHandler::OnRecvLogoutRsp},
68. {"80000003", &CServiceHandler::OnRecvSubscripRsp},
69. {"80000004", &CServiceHandler::OnRecvUnSubscripRsp},
70. {"80000005", 0},//&CServiceHandler::OnRecvHelloRsp}
72. {"00000001", &CServiceHandler::OnRecvLogin},
73. {"00000002", &CServiceHandler::OnRecvLogout},
74. {"00000003", &CServiceHandler::OnRecvSubscrip},
75. {"00000004", &CServiceHandler::OnRecvUnSubscrip},
76. {"00000005", &CServiceHandler::OnRecvHello}
77. };
79. CServiceHandler::CServiceHandler(**void**)
80. :m\_pCvgCpMgr(0)
81. ,m\_pCfg(0)
82. {
83. }
85. CServiceHandler::~CServiceHandler(**void**)
86. {
87. }
89. //初始化配置，创建线程池
90. **int** CServiceHandler::Init(CConfig\* pCfg)
91. {
92. assert(0 != pCfg);
93. **if** (0 == pCfg)
94. **return** -1;
95. m\_pCfg = pCfg;
96. **return** 0;
97. }
99. //启动调度线程及工作线程池
100. **int** CServiceHandler::Start()
101. {
102. //启动调度线程
103. BeginThread();
104. **return** 0;
105. }
107. //停止工作线程池及调度线程
108. **void** CServiceHandler::Stop()
109. {
110. //停止调度线程
111. CRLog(E\_APPINFO,"%s","Stop ServiceHandler Thread");
112. EndThread();
113. }
115. //清理资源
116. **void** CServiceHandler::Finish()
117. {
118. m\_deqService.clear();
119. }
121. **void** CServiceHandler::Bind(CConnectPointManager\* pCpMgr,**const** unsigned **long**& ulKey)
122. {
123. m\_ulKey = ulKey;
124. m\_pCvgCpMgr = **dynamic\_cast**<CCvgCpMgr\*>(pCpMgr);
125. }

128. **int** CServiceHandler::SendPacket(CPacket &pkt)
129. {
130. **try**
131. {
132. CSamplerPacket & pktService = **dynamic\_cast**<CSamplerPacket&>(pkt);
134. m\_deqCondMutex.Lock();
135. m\_deqService.push\_back(pktService);//**向队列添加数据**
136. m\_deqCondMutex.Signal(); //**队列发送信号**
137. m\_deqCondMutex.Unlock();
138. **return** 0;
139. }
140. **catch**(std::bad\_cast)
141. {
142. CRLog(E\_ERROR,"%s","packet error!");
143. **return** -1;
144. }
145. **catch**(std::exception e)
146. {
147. CRLog(E\_ERROR,"exception:%s!",e.what());
148. **return** -1;
149. }
150. **catch**(...)
151. {
152. CRLog(E\_ERROR,"%s","Unknown exception!");
153. **return** -1;
154. }
155. }
157. **int** CServiceHandler::ThreadEntry()
158. {
159. **try**
160. {
161. **while**(!m\_bEndThread)
162. {
163. m\_deqCondMutex.Lock();
164. **while**(m\_deqService.empty() && !m\_bEndThread)
165. m\_deqCondMutex.Wait(); //**队列等待信号**
167. **if** (m\_bEndThread)
168. {
169. m\_deqCondMutex.Unlock();
170. **break**;
171. }
173. CSamplerPacket pkt = m\_deqService.front();
174. m\_deqService.pop\_front();
175. m\_deqCondMutex.Unlock();
177. **try**
178. {
179. RunPacketHandleApi(pkt);
180. }
181. **catch**(...)
182. {
183. CRLog(E\_CRITICAL,"%s","Unknown exception");
184. }
185. }
186. CRLog(E\_APPINFO,"%s","RiskHandler Thread exit!");
187. **return** 0;
188. }
189. **catch**(std::exception e)
190. {
191. CRLog(E\_ERROR,"exception:%s!",e.what());
192. **return** -1;
193. }
194. **catch**(...)
195. {
196. CRLog(E\_ERROR,"%s","Unknown exception!");
197. **return** -1;
198. }
199. }
201. **int** CServiceHandler::End()
202. {
203. m\_deqCondMutex.Lock();
204. m\_deqCondMutex.Signal();
205. m\_deqCondMutex.Unlock();
207. CRLog(E\_APPINFO,"%s","ServiceHanlder thread wait end");
208. Wait();
209. **return** 0;
210. }
212. //匹配报文处理成员函数并进行调用处理
213. **int** CServiceHandler::RunPacketHandleApi(CSamplerPacket& pkt)
214. {
215. std::string sCmdID;
216. **try**
217. {
218. sCmdID = pkt.GetCmdID();
220. **int** nSize = **sizeof**(m\_PktCmd2Api)/**sizeof**(PktCmd2Api);
221. **for** ( **int** i = 0 ; i < nSize ; i++ )
222. {
223. **if** ( m\_PktCmd2Api[i].sApiName == sCmdID )
224. {
225. **if** (m\_PktCmd2Api[i].pMemberFunc == 0)
226. **break**;
228. **return** (**this**->\*(m\_PktCmd2Api[i].pMemberFunc))(pkt);
229. }
230. }
232. CRLog(E\_ERROR,"Unknown packet! sCmdID= %s" ,sCmdID.c\_str());
233. **return** -1;
234. }
235. **catch**(std::exception e)
236. {
237. CRLog(E\_CRITICAL,"exception:%s,Handle Packet:%s", e.what(),sCmdID.c\_str());
238. **return** -1;
239. }
240. **catch**(...)
241. {
242. CRLog(E\_CRITICAL,"Unknown exception,Handle Packet:%s",sCmdID.c\_str());
243. **return** -1;
244. }
245. }
247. **int** CServiceHandler::OnRecvLoginRsp(CSamplerPacket& pkt)
248. {
249. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
251. unsigned **int** uiSeqNo = 0;
252. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
254. unsigned **int** uiNodeID = 0;
255. msg.GetField(MSG\_NODE\_ID,uiNodeID);
256. m\_sNodeID = ToHexString<unsigned **int**>(uiNodeID);
258. unsigned **int** uiRst = 0;
259. msg.GetField(MSG\_LOGIN\_RESULT,uiRst);
260. CRLog(E\_PROINFO,"OnRecvLoginRsp SeqNo:%u, NodeID:%u, Result:%u",uiSeqNo, uiNodeID, uiRst);
262. **return** 0;
263. }
265. **int** CServiceHandler::OnRecvLogoutRsp(CSamplerPacket& pkt)
266. {
267. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
269. unsigned **int** uiSeqNo = 0;
270. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
272. unsigned **int** uiNodeID = 0;
273. msg.GetField(MSG\_NODE\_ID,uiNodeID);
274. m\_sNodeID = ToHexString<unsigned **int**>(uiNodeID);
276. unsigned **int** uiRst = 0;
277. msg.GetField(MSG\_LOGIN\_RESULT,uiRst);
278. CRLog(E\_PROINFO,"OnRecvLogoutRsp SeqNo:%u, NodeID:%u, Result:%u",uiSeqNo, uiNodeID, uiRst);
280. **return** 0;
281. }
283. **int** CServiceHandler::OnRecvSubscripRsp(CSamplerPacket& pkt)
284. {
285. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
287. unsigned **int** uiSeqNo = 0;
288. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
290. unsigned **int** uiNodeID = 0;
291. msg.GetField(MSG\_NODE\_ID,uiNodeID);
293. unsigned **int** uiRst = 0;
294. msg.GetField(MSG\_SUBSCRIP\_RESULT,uiRst);
295. CRLog(E\_PROINFO,"OnRecvSubscripRsp SeqNo:%u, NodeID:%u, Result:%u",uiSeqNo, uiNodeID, uiRst);

298. **return** 0;
299. }
301. **int** CServiceHandler::OnRecvUnSubscripRsp(CSamplerPacket& pkt)
302. {
303. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
305. unsigned **int** uiSeqNo = 0;
306. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
308. unsigned **int** uiNodeID = 0;
309. msg.GetField(MSG\_NODE\_ID,uiNodeID);
311. unsigned **int** uiRst = 0;
312. msg.GetField(MSG\_SUBSCRIP\_RESULT,uiRst);
313. CRLog(E\_PROINFO,"OnRecvUnSubscripRsp SeqNo:%u, NodeID:%u, Result:%u",uiSeqNo, uiNodeID, uiRst);
315. **return** 0;
316. }
318. **int** CServiceHandler::OnRecvLogin(CSamplerPacket& pkt)
319. {
320. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
322. string sUserID;
323. msg.GetField(MSG\_LOGIN\_ID,sUserID);
325. string sPwd;
326. msg.GetField(MSG\_LOGIN\_PWD,sPwd);
328. unsigned **short** usEnc = 0;
329. msg.GetField(MSG\_LOGIN\_PWD\_ENC,usEnc);
331. unsigned **int** uiSeqNo = 0;
332. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
334. unsigned **int** uiNodeID = 0;
335. msg.GetField(MSG\_NODE\_ID,uiNodeID);
337. CRLog(E\_PROINFO,"OnRecvLogin SeqNo:%u, NodeID:%u, UserID:%s, Pwd:%s, Enc:%d",uiSeqNo, uiNodeID, sUserID.c\_str(),sPwd.c\_str(),usEnc);

340. CMessageImpl oMsgRsp;
341. oMsgRsp.SetField(MSG\_SEQ\_ID,uiSeqNo);
342. oMsgRsp.SetField(MSG\_NODE\_ID,uiNodeID);
344. unsigned **int** nRst = 0;
345. oMsgRsp.SetField(MSG\_LOGIN\_RESULT,nRst);
347. CSamplerPacket oPktRsp(oMsgRsp,YL\_LOGIN\_RSP);
348. **if** (0 != m\_pCvgCpMgr)
349. **return** m\_pCvgCpMgr->Forward(oPktRsp,m\_ulKey);
351. **return** 0;
352. }

355. **int** CServiceHandler::OnRecvLogout(CSamplerPacket& pkt)
356. {
357. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
359. string sUserID;
360. msg.GetField(MSG\_LOGIN\_ID,sUserID);
362. string sPwd;
363. msg.GetField(MSG\_LOGIN\_PWD,sPwd);
365. unsigned **short** usEnc = 0;
366. msg.GetField(MSG\_LOGIN\_PWD\_ENC,usEnc);
368. unsigned **int** uiSeqNo = 0;
369. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
371. unsigned **int** uiNodeID = 0;
372. msg.GetField(MSG\_NODE\_ID,uiNodeID);
374. CRLog(E\_PROINFO,"OnRecvLogout SeqNo:%u, NodeID:%u, UserID:%s, Pwd:%s, Enc:%d",uiSeqNo, uiNodeID, sUserID.c\_str(),sPwd.c\_str(),usEnc);

377. CMessageImpl oMsgRsp;
378. oMsgRsp.SetField(MSG\_SEQ\_ID,uiSeqNo);
379. oMsgRsp.SetField(MSG\_NODE\_ID,uiNodeID);
381. unsigned **int** nRst = 0;
382. oMsgRsp.SetField(MSG\_LOGIN\_RESULT,nRst);
384. CSamplerPacket oPktRsp(oMsgRsp,YL\_LOGOUT\_RSP);
385. **if** (0 != m\_pCvgCpMgr)
386. **return** m\_pCvgCpMgr->Forward(oPktRsp,m\_ulKey);
388. **return** 0;
389. }
391. **int** CServiceHandler::OnRecvSubscrip(CSamplerPacket& pkt)
392. {
393. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
395. **char** cSubscripType='\*';
396. vector<unsigned **int**> vecMarkeType;
397. string sSubItems;
398. unsigned **int** uiSeqNo = 0;
399. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
401. unsigned **int** uiNodeID = 0;
402. msg.GetField(MSG\_NODE\_ID,uiNodeID);
403. msg.GetField(MSG\_SUBSCRIP\_TYPE,sSubItems);
404. cSubscripType=sSubItems.at(0);
405. msg.GetBinaryField(MSG\_SUBSCRIP\_RECS,sSubItems);

408. **size\_t** nDataLen = sSubItems.length();
409. **const** unsigned **int** \* puiData = (**const** unsigned **int** \*)sSubItems.data();
410. **for**(**int** i = 0;i < nDataLen/**sizeof**(unsigned **int**);i++)
411. {
412. vecMarkeType.push\_back(ntohl(\*puiData));
413. puiData ++;
414. }
416. CRLog(E\_PROINFO,"OnRecvSubscrip SeqNo:%u, NodeID:%u ",uiSeqNo, uiNodeID);
418. ///订阅时增加订阅请求
419. CMemData::Instance()->GetSubscriberTbl().AddSubscripItem(uiNodeID,cSubscripType,vecMarkeType);
421. CMessageImpl oMsgRsp;
422. oMsgRsp.SetField(MSG\_SEQ\_ID,uiSeqNo);
423. oMsgRsp.SetField(MSG\_NODE\_ID,uiNodeID);
425. unsigned **int** nRst = 0;
426. oMsgRsp.SetField(MSG\_SUBSCRIP\_RESULT,nRst);
428. CSamplerPacket oPktRsp(oMsgRsp,YL\_SUBSCRIP\_RSP);
429. **if** (0 != m\_pCvgCpMgr)
430. **return** m\_pCvgCpMgr->Forward(oPktRsp,m\_ulKey);
432. **return** 0;
433. }

436. **int** CServiceHandler::OnRecvUnSubscrip(CSamplerPacket& pkt)
437. {
438. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
440. vector<unsigned **int**> vecMarkeType;
441. string sSubItems;
442. unsigned **int** uiSeqNo = 0;
443. msg.GetField(MSG\_SEQ\_ID,uiSeqNo);
445. unsigned **int** uiNodeID = 0;
446. msg.GetField(MSG\_NODE\_ID,uiNodeID);
447. msg.GetBinaryField(MSG\_SUBSCRIP\_RECS,sSubItems);

450. **size\_t** nDataLen = sSubItems.length();
451. **const** unsigned **int** \* puiData = (**const** unsigned **int** \*)sSubItems.data();
452. **for**(**int** i = 0;i < nDataLen/**sizeof**(unsigned **int**);i++)
453. {
454. vecMarkeType.push\_back(ntohl(\*puiData));
455. puiData ++;
456. }
458. CRLog(E\_PROINFO,"OnRecvUnSubscrip SeqNo:%u, NodeID:%u ",uiSeqNo, uiNodeID);
460. ///订阅时增加订阅请求
461. CMemData::Instance()->GetSubscriberTbl().CancelSubscriber(uiNodeID);
462. CMessageImpl oMsgRsp;
463. oMsgRsp.SetField(MSG\_SEQ\_ID,uiSeqNo);
464. oMsgRsp.SetField(MSG\_NODE\_ID,uiNodeID);

467. //MemDb handle
468. unsigned **int** nRst = 0;
469. oMsgRsp.SetField(MSG\_SUBSCRIP\_RESULT,nRst);
471. CSamplerPacket oPktRsp(oMsgRsp,YL\_UNSUBSCRIP\_RSP);
472. **if** (0 != m\_pCvgCpMgr)
473. **return** m\_pCvgCpMgr->Forward(oPktRsp,m\_ulKey);
475. **return** 0;
476. }
478. **int** CServiceHandler::OnRecvHello(CSamplerPacket& pkt)
479. {
480. CMessageImpl& msg = **dynamic\_cast**<CMessageImpl&>(pkt.GetMsg());
482. string sHelloCtn;
483. msg.GetBinaryField(MSG\_HELLO\_CONTENT,sHelloCtn);
485. string sNodeID;
486. CConfig \*pGlobalCfg = m\_pCfg->GetCfgGlobal();
487. **if** (0 != pGlobalCfg)
488. {
489. pGlobalCfg->GetProperty("node\_id",sNodeID);
490. }
491. sNodeID = "/" + sNodeID;
492. sHelloCtn.append(sNodeID);
493. msg.SetBinaryField(MSG\_HELLO\_CONTENT,sHelloCtn);
495. **if** (0 != m\_pCvgCpMgr)
496. **return** m\_pCvgCpMgr->Forward(pkt,m\_ulKey);
498. **return** 0;
499. }

# 二.Theron（Actor框架）★★★★★★

简介

Matlab使用了Theron的并发框架库，提升了4.5倍

http://www.theron-library.com/index.php?t=story&p=43

## Actor模型

## [为什么Actor模型是高并发事务的终极解决方案](http://www.baidu.com/link?url=6rD_KM-tz4Rid0m7-_n1UDa4C_UoZL4O8SbEA64-eJtT65KMkOMMmK9ggTEqLF49)

## 整体脉络

## 库结构分析

## 代码分析

# 三.Poco笔记

**Foundation模块。**

Platform.h ：定义跨平台操作系统。

Foundation.h : 选中平台os

# 四.QT并发编程

1.QThread

1.1例子

Void Func()

{

…

}

在使用的地方

QThread thread;

Thread.start(&Func);

1.2

2.QThreadPool和QRunnable

Class classname : public QRunnable

{

void run()

{

}

setParent()

{

}

Private:

void \*parent;

}

QThreadPool::Instance().start(QRunnable)

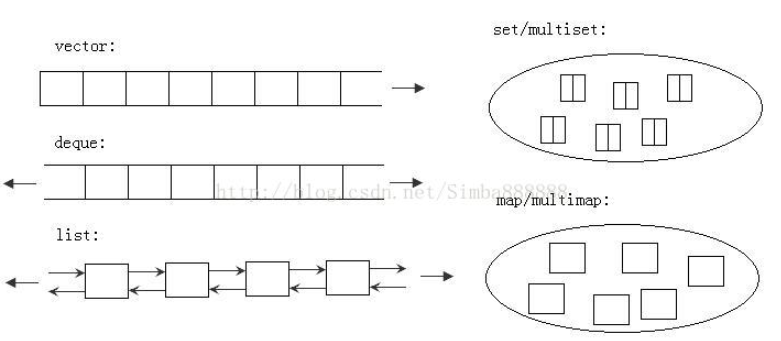
# 五.STL源码学习记录

## STL六大组件

### 容器(container)

#### 概念:

（七种基本容器vector，list，Map , deque,set,multimap,multiset）



STL提供的Queue和Stack，虽然看似容器，其实只能算是一种容器配接器，因为 它们的底部完全借助Deque，所有操作有底层的Deque供应。

stack是一种先进后出（FILO）的数据结构，它只有一个出口。deque是双向开口的数据结构，所以SGI STL便以deque作为缺省情况下的stack底部结构，封闭其头端开口。stack没有迭代器，所以除了顶部元素，无法存取其它元素，即不能遍历stack。

stack的成员函数都是针对其顶部元素进行操作：

修改:push()，pop()

访问函数 ：top()。

queue是一种先进先出（FIFO）的数据结构，它有两个出口。queue也是以deque作为底部结构，封闭其底端的出口和前端的入口。queue，只有顶端（两端）的元素能被外部使用，所以queue也没有迭代器，不提供遍历功能。

queue的成员函数有：

访问函数：front()，back()

修改函数：push()，pop()。

可以看到stack和queue的成员函数以及特性都是针对其数据结构来的，所以深入理解其内部结构，不易与deque众多的成员函数混淆。当然stack和queue也可以list为底层结构实现。

#### 自定义容器:

### 算法(algorithm)

#### 概念:

#### 自定义算法

### 迭代器(iterator)

#### Iterator基本要素

**由于iterator在STL体系内的核心，iterators和算法都是根据iterator的条件而撰写，需要满足一些实现需求（理念上就是以iterator concepts为一份清单，代表【应提供的机能】，对预算法而言则表示【假设已经存在的机能】）：**

##### 1.注意iterator应该是constant或mutable。两者间的区别在于其reference type和pointer type是T &和T \* ，或是 const T & 和 const T \* .

##### 2.必须确定iterator\_traits能够正确作用于你的iterator身上。

##### 3.尽可能多的提供较多的iterator操作行为，除非对效率有所影响。

**除了需求，还有一些必要组成:**

##### 4.iterator具有5种分类,的五个iterator concepts：input iterator,output iterator,forward iterator,biderectional iterator 和Random access iterator. 这5中分类决定了5种Iterator, 其中普通指针是Random access Iterator这个concept的一个modal。STL给了这5种Iterator一个tag用于标识这五种类型。

##### 5.Concepts里面包含一些associated types(相关型别)，是形成iterator条件的重要组成：

Value Type(数值型别),Difference Type（差距型别），Reference Type和Pointer Type,Iterator Tags.

##### 6.STL为了方便取得iterator的特性类型，统一上诉因素，定义了类型萃取,泛化版如下：

template <class iterator>

struct iterator\_traits

｛

typedef typename iterator::iterator\_category iterator\_category;

typedef typaname iterator::value\_type value\_type;

typedef typename iterator::diference\_type diference\_type;

typedef typename iterator::reference reference;

typedef typename iterator::pointer pointer;

｝

#### 实作样例:

每个STL容器都拥有嵌套的迭代器，此外，STL还包含一些独立运作的迭代器类，其中大部分是迭代器适配器。

标准C++规定，所有的迭代器类都得继承自迭代器基类，以满足iterator\_traits所要求的各种嵌套型别。继承自那五种iterator，在标准C++里面，想定义一个新的output iterator,你需要继承 iterator<output\_iterator\_tag,void,void,void,void>，或者定义一个新的Forward iterator,你应该继承自: iterator<forward\_iterator\_tag,T>;

类似这样：

template <class T>

class my\_forward\_iterator:public std::iterator<forward\_iterator\_tag,T>

{

//内部通过继承，自动获得五个typedefs

};

不继承iterator,同样可以做相同的事情，只需要嵌套五个型别，虽然罗嗦，但是很清晰：

template <class T>

class my\_forward\_iterator

{

public:

//型别的名字固定为红色

typedef forward\_iterator\_tag iterator\_category;

typedef T value\_type;

typedef ptrdiff\_t diference\_type;

typedef T & reference;

typedef T \* pointer;

}

#### 容器迭代器:

struct input\_iterator\_tag {};

struct output\_iterator\_tag {};

struct forward\_iterator\_tag : public input\_iterator\_tag {};

struct bidirectional\_iterator\_tag : public forward\_iterator\_tag {};

struct random\_access\_iterator\_tag : public bidirectional\_iterator\_tag {};

这是五种迭代器类别的标志，可以看出，这些结构只是标志，没有任何内容。从语义上讲，它基本可以使用一个长变量来表示。但是种表达方式有它的好处，在定义普通迭代器的时候，我们只需要在迭代器内部typedef其中一个类型，那么将来就可以通过类型萃取器，来得到这个迭代器的类型信息（进而可以做相应的优化）。

（数据和算法分离，迭代器起衔接的作用）容器内部定义，重载！=，==，\*，->,++,--等操作符。迭代器是为了在算法与容器之间隔开，可以让两者独立发展。这样势必要求迭代器需要像算法展现统一的接口, 即使我们尽量希望向外部展现一个统一的容器接口（迭代器），我们仍然需要区分对待，我们在迭代器上区分对待，就可以充分发挥各种容器的个性特点。算法也可以根据相应的不同，优化对待不同的容器。这就是出现不同的类型容器的原因。

|  |  |  |
| --- | --- | --- |
| **Iterator Category** | **Ability** | **Providers** |
| **Input iterator** | Reads forward | istream |
| **Output iterator** | Writes forward | ostream, inserter |
| **Forward iterator** | Reads and writes forward |  |
| **Bidirectional iterator**  双向迭代器，forward基础上加上了-- | Reads and writes forward and backward | list, set, multiset, map, multimap |
| **Random access iterator** | Reads and writes with random access | vector, deque string, array |

不同类型的迭代器可以参考和“各种iterator的能力”:

http://blog.sina.com.cn/s/blog\_93b45b0f01014o9d.html

#### 其他迭代器（大都为iterator adapter）

除了普通迭代器，C++标准模板库还定义了几种特殊的迭代器，分别是插入迭代器、流迭代器、反向迭代器和移动迭代器，定义在<iterator>头文件中。

|  |  |  |  |
| --- | --- | --- | --- |
| Iterator classes |  | 辅助函数 | 意义 |
| Insert iterators  (属于output iterator类型迭代器) | Front\_insert\_iterator  Back\_insert\_iterator  Insert\_iterator | Front\_inserter  Back\_inserter  inserter |  |
| Stream iterators | Istream\_iterator  Ostream\_iterator  Istreambuf\_iterator  Ostreambuf\_iterator | I（属于input iterator类型迭代器）  O （属于output iterator类型迭代器） |  |
| Reverse iterators |  | Bidirectional iterator | 能够在区域上逆向移动 |
| Raw\_storage\_iterators |  | Output iterator |  |

主要介绍三种插入迭代器（back\_inserter,inserter,front\_inserter）的区别，首先，什么是插入迭代器？插入迭代器是指被绑定在一个容器上，可用来向容器插入元素的迭代器。   
back\_inserter：创建一个使用push\_back的迭代器   
inserter：此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器。**元素将被插入到给定迭代器所表示的元素之前。**   
front\_inserter：创建一个使用push\_front的迭代器（**元素总是插入到容器第一个元素之前**）   
由于list容器类型是双向链表，支持push\_front和push\_back操作，因此选择list类型来试验这三个迭代器。

list<int> lst = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

list<int> lst2 ={10}, lst3={10},lst4={10};

copy(lst.cbegin(), lst.cend(), back\_inserter(lst2));

//lst2包含10,1,2,3,4,5,6,7,8,9

copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));

//lst3包含1,2,3,4,5,6,7,8,9,10

copy(lst.cbegin(), lst.cend(), front\_inserter(lst4));

//lst4包含9,8,7,6,5,4,3,2,1,10

Copy(lst.cbegin(), lst.cend(), ostream\_iterator(lst4))

由示例便可轻易看出这三者的区别了。

一般使用back\_inserter，inserter，front\_inserter等模版参数，来返回对应的迭代器，来规避对应迭代器的强类型。

copy(vecSrc.begin(), vecSrc.end(), back\_insert\_iterator<list<int> >(vecDest));

// copy(vecSrc.begin(), vecSrc.end(), back\_inserter(vecDest)); // 效果一样

#### 自定义迭代器

1.通过继承自基类获得型别声明。

2.清清楚楚提供嵌套的typedefs ，这个是主流而且比较简单的做法。

#### 使用迭代器

使用容器嵌套的迭代器，使用自定义迭代器，通过iterator adapter获得迭代器配合泛型算法使用。

#### 各种Iterator的能力

##### 3.1 Output iteator

具有只写性质，适用于单回（single-pass）算法，如：copy。

Input iterator的需求条件：

（1）可以复制和赋值： Type(iter), copy constructor

(2) 可以写值：\*iter = val，但是不能读：val = \*iter，val = iter->member;

(3) 可以累加：++iter, iter++。

注意：不支持 ==， ！=, 唯一不支持的iterator。

类似Input iterator，同一时间不能有两个不同的Output iterator指向同一个区间内的不同地点。移动中的指针进行写入，一旦写完遍继续移动。

下面以copy算法来讲述：

template <typename InputIterator, typename OutputIterator>

OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)

{

for (  ; first != last; ++result, ++first) {

\*result = \*first;  
}

return result;  
}

copy输入区间需要只读性质，输出区间需要只写性质，是与输入输出都有关的单回算法。

Output Iterator 所形成的区间都是以单独一个Iterator再加上一个计量值来决定的。如copy: 输出区间以result开头，再以[first, last）的元素个数作为

区间的长度。

Output iterator :有 insert\_iterator, front\_insert\_iterator, back\_insert\_iterator和 ostream\_iterator。

ostream\_iterator iter; \*iter = val会使val格式化输出到ostream身上。

template <class T>

class ostrem\_iterator {

private:

ostream \*os;

const char \*string;

public:

ostrem\_iterator(ostream &s, const char \*c = 0) : os(&s), string(c) {}

ostream\_iterator(const ostream\_iterator &i) : os(i.os), string(i.string) {}

ostream\_iterator& operator=(const ostream\_iterator &i)

{

os = i.os;

string = i.string;

return \*this;  
}

ostream\_iterator<T>& operator=(const T &value)

{

\*os << value;

if (string)  {

\*os << string;  
}

return \*this;  
}

//实现os << value; 替换为 \*iter = val;使用proxy class, 而且是其自身\*this,也可以使用其它class但是没理由这么做。

ostream\_iterator<T>& operator\*() { return \*this; }

ostream\_iterator<T>& operator++() { return \*this; }

ostream\_iterator<T>& operator++(int) { return \*this; }  
};

##### 3.2 Input Iterator

具有只读性质，适用于单回（single-pass）算法，如：copy。

限制条件：

（1）Input iterator用来指向某对象，但不需要提供任何更改该对象的方法, 就是只读的。

你可以\*或->，但是不能对结果赋值，即：\*iter; iter->member可以，\*iter = val（要求写）不一定可以。

（2）Input iterator可以累加，但是不可以递减。

可以++iter; iter++, 这是Input iterator唯一需要的运算形式。 不可以--iter; iter--。

（3）Input iterator支持有限的比较形式，可以测试iter1 = iter2； iter1 ！= iter2； 但是不可以比较谁在谁前面。

（4）Input iterator 是 single pass的，你有且只能遍历range(beg, end)一次。不能重复遍历。

（5）Input iterator可以copy,支持copy constructor: TYPE（iter）。

特点：

Input iterator 只能读元素一次，如果使用iterator的副本和原始迭代器共同读，可能读取不同的值。典型的情况是从键盘读取输入，你不能读取两次。

Input iterator 就像从终端机或网络连线读取数据一样的读取输入值：你可以要求下一个值，但一旦如此，前一个值就消失无踪了。使用Input iterator class

istream\_iterator,可以简单的从input stream读取数据。

使用Input iterator的算法：

此算法必须是single pass（单回）的算法，只需要遍历区间一次读取值，不需要更改值的。常见有：

find, find\_if, equal, partial\_sum, random\_sample, set\_intersection等等。

##### 3.3. Forward Iterators

缘起：

（1） input iterator具只读性， output iterator 具只写性。意味着： 需要读取并更改的某一区间的算法，无法单就这些concept来运作。

可以利用input iterator写出某种查找算法，但是无法写出查找并置换的算法。

（2） 运用input iterator和output iterator的算法，只能是single pass算法，使得算法的复杂度O(N)。

(3)任何时刻，区间内只能有一个有效的input iterator或output iterator。使得算法在同一时刻只能对单一元素做动作。对于行为取决于两个或多个不同元素

之间的关系的算法无能为力。

而采用Fordward iterator就不会有上述限制。一个Forward iterator model必须是input iterator model和output iterator model。

因此，可以写出一种算法在同一区间内做读取动作和更新动作。

例如：replace()

template <class ForwardIterator, class T>

void replace(ForwardIterator first, ForwardIterator last,

  const T &old\_value, const T &new\_value)

{

for ( ; first != last; ++first) {

if (\*first == old\_value) {

\*first = new\_valude;  
}  
}  
}

replace 是single pass算法，一次只作用一个元素，由于需要在同一个iterator上进行读取和更改，所以用Forward\_iterato。

Forward iterator也适用于在同一区间使用一个以上的iterator，是multipass的。

//查找连续两个元素具有相同值

template <class ForwardIterator>

ForwardIterator adjacent\_find(ForwardIterator first, ForwardIterator last)

{

if (first == last) {

return last;  
}

ForwardIterator next = first;

while (++next != last) {

if (\*first == \*next) {

return first;  
}

first = next;  
}

return last;  
｝

\*first = \*next;使用了两个Iterator所以用Forward\_iterator。

所以相比于Input iterator和Output iterator， forward\_iterator提供了iter1 = iter2,因为它可以使用不同的Iteator。

相关容器： forward\_list<>, unordered containes。

##### 3.4. Bidirectional Iterator

类似于Forward Iterator,Bidirection iterator允许multipass，也可以是const或mutable。

支持双向移动，因此相对于Forward\_iterator支持--iter和iter--，如遍历单项链表用Forward Iterator，遍历双向链表用Bidirectiorn iterator。

通常是由于需要反向移动而选择Bidirectional iterator, 即指定某元素之后需要寻找先前的元素。

如：

template <class BidrectionalIteartaor, class OutputIterator>

OutputIterator reverse\_copy(BidirctionalIterator first, Bidirctional last, OutputIterator result)

{

while (first ！= last) {

--last;

\*result = \*last;

++result;  
}

return result;

}

需要--last;

相关容器： class list<>, association containers。

##### 3.5 Random Access Iterators

提供了算术运算能力：iter + n, iter -n, iter[n], iter - iter2, iter < iter2，这些都是特有的。

排序算法需要用到这个iterator,需要对比并交换相隔甚远的元素，而非只是相邻元素。

考虑到Random Access Iterator主要是考虑到复杂度：

例如：advance(), Forward Iterator是O(N), 而随机Iterator为O(1);

所以：Random Access iterator正真独特的性质为：在固定时间随机访问任意元素。

相关容器： array, vector, deque, string, wstring, C-style arrays(pointers)。

### 适配器（adapter）

#### 定义

适配器[Adapter] 是 [ 能够将某接口转化为另外一个接口]的东西。

#### 概念:

适配器在STL里面是用来辅助其他STL组件工作的。

##### 容器适配器

##### 迭代器适配器

Iterator adapter是一种有用的iterator，提供不同的接口给既有的STL组件。STL定义了一些iterator adapter，包括front\_insert\_iterator,back\_insert\_iterator,reverse\_iterator

##### 仿函数适配器

#### 自定义适配器：

### 空间分配器(allocator)

#### 概念：

**二级分配器。大于128bytes，小于128bytes，则找一个16个元素的数组，分表保存着指向8的整数倍的大小空间的内存地址，大小逐个变大，为：8bytes，16bytes，24bytes,32bytes …… ,128bytes**

#### 自定义：

### 仿函数(functor)

#### 概念:

重载operation()

#### 自定义：

# 六.Libgo（并发编程库）★

**编译不通过，暂时不研究，可参考:Theron , cpp-actor-framework.**

主页: <https://github.com/yyzybb537/libgo>

介绍:libgo是一个使用C++11编写的协作式调度的stackful协程库, 同时也是一个强大易用的并行编程库.

使用libgo编写多线程程序，即可以像golang、erlang这些并发语言一样开发迅速且逻辑简洁，又有C++原生的性能优势，鱼和熊掌从此可以兼得。

libgo有以下特点：

1.提供golang一般功能强大协程，基于corontine编写代码，可以以同步的方式编写简单的代码，同时获得异步的性能，

2.支持海量协程, 创建100万个协程只需使用4.5GB物理内存.(真实值, 而不是刻意压缩stack得到的测试值)

3.支持多线程调度协程, 提供高效的负载均衡策略和协程同步机制, 很容易编写高效的多线程程序.

4.调度线程数支持动态伸缩, 不再有调度慢协程导致头部阻塞效应的问题.

5.使用hook技术让链接进程序的同步的第三方库变为异步调用，大大提升其性能。再也不用担心某些DB官方不提供异步driver了，比如hiredis、mysqlclient这种客户端驱动可以直接使用，并且可以得到不输于异步driver的性能。

6.动态链接和全静态链接均支持，便于使用C++11的用户静态链接生成可执行文件并部署至低版本的linux系统上。

7.提供Channel, 协程锁(co\_mutex), 协程读写锁(co\_rwmutex), 定时器等特性, 帮助用户更加容易地编写程序.

8.支持协程局部变量(CLS), 并且完全覆盖TLS的所有使用场景(详见教程代码sample13\_cls.cpp).

从近两年的用户反馈情况看，有很多用户都是已经有了一个异步非阻塞模型的项目(可能是基于epoll、libuv或asio等网络库)，然后需要访问MySQL这类没有提供异步Driver的DB. 常规的连接池+线程池的方案在高并发场景下的开销十分昂贵(每个连接对应一个线程才能达到最佳性能, 几千个指令周期的线程上下文切换消耗+过多的活跃线程会导致OS的调度能力急剧下降), 让许多用户难以接受.

鉴于此种情况, 想要使用libgo解决非阻塞模型中阻塞操作的问题，也是完全不必重构现有代码的, 全新的libgo3.0为此场景量身打造了三大利器, 可以无侵入地解决这个问题：运行环境隔离又可以便捷交互的多调度器(详见教程代码sample1\_go.cpp)，替代传统线程池方案的libgo协程池(详见教程代码sample10\_co\_pool.cpp)，连接池(详见教程代码sample11\_connection\_pool.cpp)

tutorial目录下有很多教程代码，内含详细的使用说明，让用户可以循序渐进的学习libgo库的使用方法。

# 七.boost

## asio

# 八.nanomsg

关键字(C语言，IOCP)

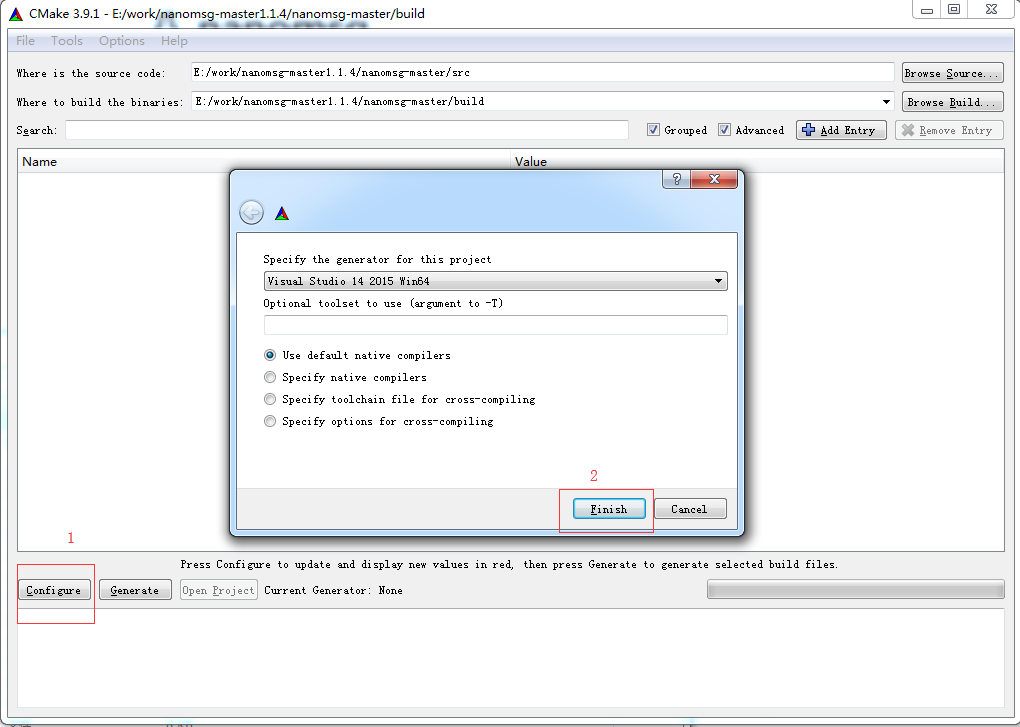
ZeroMQ的下一代作品，也由C++转为C语言开发，window平台提供IOCP支持，

消息队列库，主页” https://nanomsg.org/

下载代码: <https://github.com/nanomsg/nanomsg>

解压到本地: E:\work\nanomsg-master1.1.4\nanomsg-master

打开Cmake3.9:设置好路径，生成VS工程。



# 九.Facebook - wangle

（https://github.com/facebook/wangle）

**Wangle**

我们的 OLAP 引擎基本上是一个用于存储和查询多维数据的分布式数据库。该引擎使用 Wangle 作为一个应用服务器的基础框架。所有的逻辑被分层堆放在 Wangle 的处理函数中，链式地放到一个管道中或者从一个管道中取出。它与我们的 Proxygen HTTP 服务器通信，以处理数据查询和节点之间的通信。

它使用网状的服务器来共享相同的（对称的）二进制可执行文件，所以没有主从模式之分。每个节点既是主服务器也是从服务器，它们之间使用一套自定义的二进制数据协议来传递数据或者消息。

我们需要的库中唯一缺少的是用于完成协助调度存储器和引擎里的查询任务的 fibers。另外，虽然目前 Folly 和 Wangle 的开发者提供了实验性的版本，但是这还不能用于产品。

James的OLAP引擎本质上是一个分布式的数据库，用于存储和查询多维数据。该引擎以Wangle作为应用服务器的基础。所有的逻辑都分解到了Wangle handler中，可以链到一起构成一条流水线。每个服务器都是对称的，是相同的二进制可执行程序，没有主从之分。每个服务器节点可以通过定制的二进制协议交换数据或消息。

# 十.Facebook - folly

**Folly** （https://github.com/facebook/folly/）

这是一个不错的C++库，有很多高性能的类。James在其引擎中用到了fbvector、fbstring等类，与std::vector和std::string相比，它们性能更好。另外像Future，原子化的无锁数据结构也经常用到。

Folly的容器性能很好（可以参阅：https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md）。

另外，阅读Folly的代码对C++程序员成长也有很大帮助。

这是一个很棒的 C++ 库，它包含很多高性能的类以供使用。我在我们的引擎中使用它们的 fbvector 和 fbstring，因为它们分别提供了比 std::vector 和 std::string 更好的性能。我们还使用了它们的 futures 和原子无锁数据结构。

Facebook 针对动态增长分配做了一个很聪明的举动来避免平方阶增长（在数学上可以简单地证明并解释为什么平方阶增长是不好的）。他们的容器使用 1.5x 而不是 2x 来增加内存尺寸，从而提高性能。

## String类

string类的设计和优化，主要体现在两个方面：

1. 内存模型

2. 常用方法的优化

# 十一. Facebook - Proxygen

Proxygen（https://github.com/facebook/proxygen）

Proxygen是Facebook开发的一个异步HTTP服务器。James使用Proxygen作为HTTP服务器，将JSON数据插入到OLAP引擎，或从引擎获得JSON数据。只需要一天，就能创建一个调用引擎的高性能服务器。

James将其与Python Tornado服务器做了对比，在一个EC2实例上，使用200个HTTP连接，C++/Proxygen每秒可以处理1 990 130个请求，Python/Tornado每秒可以处理41 329个请求。

# 十二. NoahGameFrame ★★★★★★★

## 作者主页:

<https://github.com/ketoo/>

<https://github.com/ArkGame>

## 敏捷服务器开发解决方案，主要思想是模块化，插件化，分层设计，事件驱动。

评价:代码通俗移动，写的文档也是很清晰，可以研究下。

涉及领域：

插件构建

Libevent

Actor模型的场景使用

## Dependencies

* libevent 2.0.22
* easylogging++ 9.80
* mysql++ 3.2.2
* google protobuf 3.6
* redis-cpp-client
* Theron 6.00.01

代码地址: <https://gitee.com/hamasm/cpp-actor-framework>

教程:

<http://bbs.noahframe.com/forum.php?mod=forumdisplay&fid=39&page=1>

<http://bbs.noahframe.com/forum.php?mod=viewthread&tid=8>

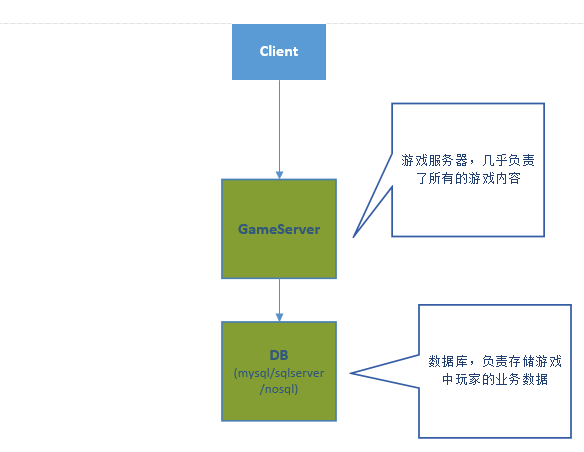
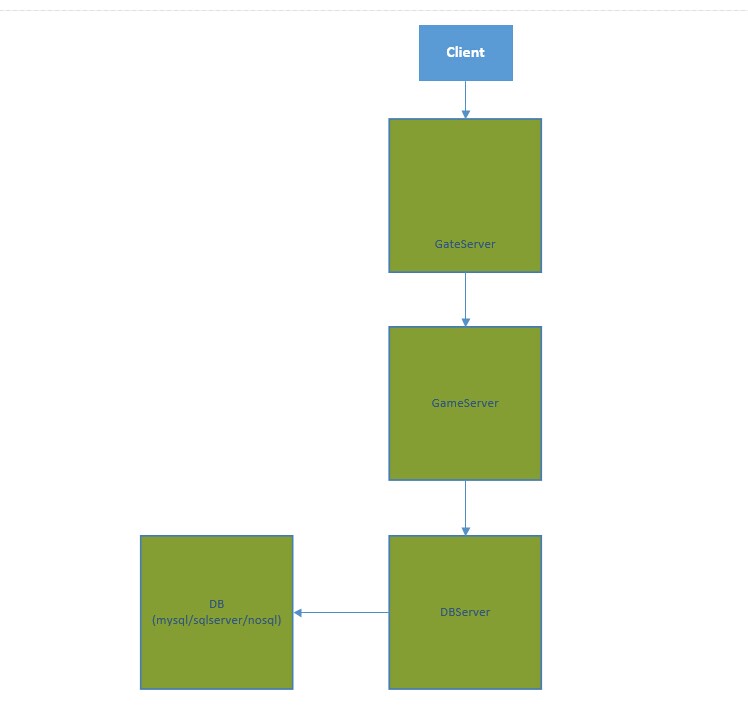
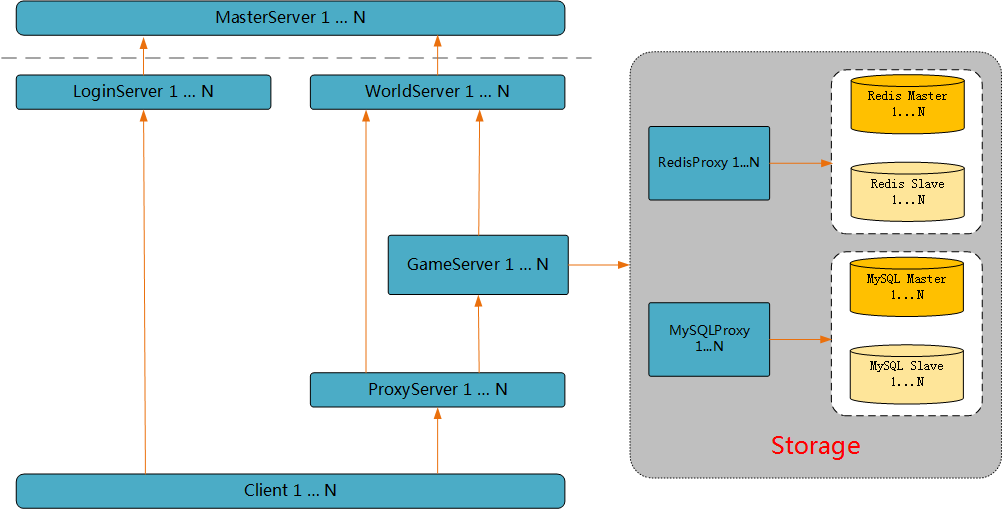
（面向接口，面向数据编程写得很好）: 抽象业务

## 介绍:

最早为客户端设计，后来随着时代的变化，而为自己又转为服务器开发，故在吸收了众多引擎的优点后（包含Ogre的插件模式&模块化管理机制，Bigworld的数据管理&配置机制，类似MYGUI的接口层次设计），经过多年演化和实践，变成了一套游戏开发解决方案。方案中包含开源的服务器架构，网络库（站在libevent的肩膀上），和unity3d的demo源码。现在NF已经在多个公司的多个项目中使用，其中包含知名产品 《全民无双》。

项目为开源的分布式服务器解决方案，其中包含了网络库，actor库，以及数据驱动等新技术，能大幅提升开发效率节省开发周期以及提高程序的稳定性。

关键词  
  
NoahGameFrame/NoahFrame/NF  
集群/负载均衡/分布式  
网关服务器 GateServer 心跳 多线程/线程池 开源网络框架/模型  
一致性hash算法/ConsistentHash  
游戏开发中的设计模式/数据结构  
Socket Nagle/粘包/开源游戏服务器/ Game Server

一说到游戏服务器，其实大部分人第一认知，估计就是网络库，在很多人眼里，网络库几乎作为服务器坚实技术的代表，但我这里想提醒的是，这些人最后几乎都走了造轮子的不归路，因为网络库虽然是组成服务器软件最重要的基础库之一，但是同时也应该是项目成熟期接触最少的库，唯一却可替代。  
  
第一阶段：最原始的服务器架构  
单纯来说很简单，大部分服务器在处理业务逻辑的时候，还会直接处理socketid相关的内容，出BUG让人防不胜防。大概如下图所示：  
   
  
优点：  
开发简单，新成员可以快速理解，无复杂集群状态管理。  
缺点：  
几乎所有业务均在一起，一处业务出问题宕机，则会影响所有业务；同时不能扩容也不能应对运营不定时带来的流量压力。  
  
第二阶段：支持分线，代理的架构  
目前还是有很多游戏服务器属于此类型，对于做手机游戏，小游戏或者创业团队来说，这类方案基本是优选（特别是对于没有历史积累的团队来说--比如说拿unity直接开发服务器，服务器能跑就行了。。。至于负载，质量，则是游戏先能做完，或者能上线再说吧）  
  
   
优点：  
1：增加了多线程/进程机制，可以有效的通过部署应对不定期的流量压力；  
2：增加了网关，有效加强了对于外网非法数据的屏蔽能力，降低了GameServer处理这些数据的消耗，不再直接连接数据库，加快了game处理业务的速度。  
缺点：增加了新成员对于架构的理解，以及也需要写一部分异步代码（数据库方面），会加入各种转发消息的功能。  
  
第三阶段：支持集群，actor，拆分子结构等方案的架构（NF解决方案）  
   
  
优点：  
1：增加了可动态扩容的多线程/进程/Actor机制，可以有效的通过动态扩容应对大平台带来的流量压力（理论上有机制，就无限扩容）；  
2：拆分了各种功能划分为单独的服务器，可以有效降低单点故障，加强了集群的可持续性和系统的可用性，避免了单个系统故障可能引起的”雪崩效应“；  
3：集群内部服务器，均增加了一致性算法，有效的保障了数据的一致性，和协议路由准确性；  
4：大量的actor应用，有效的避免了http，mysql等同步等待的业务堵塞主业务的情况，加强了整个架构的灵活性；  
5：大量的基础库插件，大大的提升了开发效率和系统的成型率；  
缺点：架构过于复杂，需要一定水平的成员才能驾驭，否则会很容易发生隐藏的非必现事故；  
  


那现在来说一下NF的服务器架构的一些基础内容。  
**1：区服管理相关**  
在整体架构上，NF设计为可分区分服也可全区全服部署。如果是全区全服部署，则相当于一个大区扩容接纳所有玩家（在worldserver下面扩容gameserver）；如果是分区分服部署，则worldserver下属为一个单服，许多worldserver构成的单服一起注册在master构成整个游戏世界。  
  
**2：服务器管理机制**  
（1）masterserver  
masterserver管理所有的loginserver和所有的worldserver，区服的状态和列表，可由worldserver主动发消息更新（所有的loginserver和worldserver一启动就要向master注册），它是整个额架构中的老大，中心节点。同时，次节点可以支持配置多台服务器以避免单点，而且这些无需额外写代码；  
  
  
（2）worldserver  
worldserver管理所有此服下面所有的gameserver和proxyserver，并保存他们的状态（所有gameserver和proxyserver一启动就需要连接到worldserver注册）。同时，次节点可以支持配置多台服务器以避免单点，而且这些无需额外写代码；  
  
  
（3）loginserver  
loginserver主要用于向用户展示区服列表和处理登录逻辑，loginserver一启动就需要像masterserver注册。同时，次节点可以支持配置多台服务器以避免单点，而且这些无需额外写代码；  
  
  
（4）gameserver一启动就要向worldserver注册并纳入管理，如果worldserver广播下来的服务器列表中有其他worldserver，还需要主动去连接其他的worldserver（几乎所有的server都会有这样的逻辑，扩容用）；同时，gameserver接受proxyserver的连接；gameserver内部自带actor，因此架构中不再继续使用DBServer这样的代理数据库服务器；  
  
  
（5）proxyserver在架构中主要用与代理client数据请求的作用以及核对消息的作用，一启动就需连接到worldserver并接受管理，并从worldserver获取到gameserver（同一个服）列表然后主动连接到所有的gameserver；proxyserver会主动汇报自己的状态和开放端口给worldserver。同时，次节点可以支持配置多台服务器以避免单点，而且这些无需额外写代码；  
  
**3：插件和模块机制**  
下过代码观察的同学就肯定知道，所有服务器中，可执行文件都是一样的，都是一个NFPluginLoader\_d这样的执行文件，他启动的时候会自动检测Plugin.xml目录下面的所有插件(plugin)，然后加载所有的插件，开始帧循环；而在每个插件中，又有大量的各种程序员们根据功能的归类划分出来的模块(module)，插件又会初始化这些模块，然后启动帧循环。插件Plugin作为模块Module的容器/承载着，本身提供了他所承载的Module创造，初始化的任务；所有的Module并行完成所有的业务逻辑，Module与Module可以互相调用接口，为对方提供服务。  
  
**4：actor机制**  
actor机制主要用户解决同步堵塞访问的瓶颈，比如数据库访问，比如http请求，比如长时间运算等问题；在NF中，大量使用了actor技术，后续还有计划，让每个scene都运行一个actor，这样来提升gameserver的承载能力；  
  
  
**5：直链nosql数据库机制**  
从架构图上可以看出，gameserver直连了数据库，并没有像传统架构那样使用一层数据库代理，是因为基于目前nosql越来越优秀，夸服同步服务需要也越来越旺盛，需要直接内存数据库以开发更复杂的业务。同时，在直连数据库的情况下，又提供actor服，解决了同步存储/加载数据的堵塞问题，使用NF即可支持同步/异步访问数据库；同时，协程技术也支持同步代码异步性能来支持此特性，让程序员不用担心io堵塞问题写同步代码，大大提升了效率；

# 十三. ARK

初略编译一次，不过，暂时不研究

项目主页：<https://github.com/ArkGame>

作者主页 : <https://www.cppfans.org/2171.html>

https://arkgame.net/

# 十四. Cpp-actor-framework

代码很炫，也可编译通过，功力深厚，但作者没有留下文档，暂时不做研究。

文档地址:

<https://actor-framework.readthedocs.io/en/latest/Introduction.html#actor-model>

代码地址: https://gitee.com/hamasm/cpp-actor-framework

<https://github.com/HAM-2015/CPP-Actor-framework>

本机代码: E:\work\hamasm-cpp-actor-framework-master\cpp-actor-framework

支持系统 windows、x86\_x64/armv7\_v8-linux、armv7\_v8-android;

支持编译器 VC2013/VC2015，GCC4.8+;

依赖于boost 1.5x;

并发逻辑控制框架(Actor Model)，适用于复杂业务逻辑.

## 基础

### 基础API

Windows下是Fiber, POSIX的等价物是setcontext, getcontext, makecontext and swapcontext

### 纤程概念

Widnows 是提供了用户级线程的，类似 coroutine 需要用户主动是切换。这在单线程程序中非常有用。线程调度模块只负责提供堆栈，环境的保存。不负责分配时间片等。

自己实现 coroutine 并不难，但能用操作系统提供的可以得到更多的便利。Windows 中把这种用户级线程叫做 Fiber，纤维的意思。比较通用的译名是纤程。

我们可以把一个 thread 转换成一个 fiber ，用到的 API 是 ConvertThreadToFiber。其实用的更多的是CreateFiber，它可以创建一个纤程，但并不切换过去运行。

被创建出来的 Fiber 会有一个上下文的地址被返回，用于以后的切换操作。我们可以用 SwitchToFiber 来切换。这是唯一用于 Fiber 释放操作权的途径。SwitchToFiber 必须显式的指定切换的目标，所以 Fiber 调度的工作需要我们自己写代码来实现。

GetCurrentFiber 和 GetFiberData 这两个函数都很有用，一个用来取到运行环境，一个用来取得创建参数，这两个函数都是用 inline 函数的形式提供在 .h 文件中的。

见云风日志: <https://blog.codingnow.com/2005/10/fiber.html>

# 十五. [zsummerX](https://github.com/zsummer/zsummerX) ★★★★★

作者:tencent游戏开发人员

zsummerX is a cross-platform C++ high performance lightweight network library. via IOCP/EPOLL/SELECT.

**zsummerX是一个跨win/mac/linux/ios/android平台的高性能轻量级网络库, 采用C++11标准实现.**

本机代码: E:\work\zsummerX-master\zsummerX-master

编译工具: VS2015

知识点:

纯手工打造的IOCP，EPOLL，由C++11编写的网络库。

**BREEZE以该库为基础，重点可以学习下breeze库**

# 十六. [breeze](https://github.com/zsummer/breeze)★★★★★★★★

可以编译，可以研究下。

zsummerX为此库的基础库，同一作者，一个C++的轻量级的分布式服务器引擎, 架构思想为一切皆service.

特点:

●实现docker和service的核心代码约2000行, 简洁朴素.

●使用C++11标准, 充分发挥shared\_ptr,functional, lambda应有的作用, 结合C++异常和模板,在满足高性能与可靠性的前提下把易用性做到了接近lua这样的脚本语言的水平.

●对开发者透明的分布式方案, 对业务开发者而言并不需要关心分布式本身, 但通过配置可以很灵活的调整服务器负载. (架构思想上可以参考skynet,bigworld,gce,nf等服务器引擎).

●网络库使用zsummerX, 支持全平台,win/linux/mac/android/ios, 可以直接嵌入到移动平台的客户端中, 打通移动客户端到服务端的通讯, zsummerX的特性该服务器引擎全部支持(本来就是为了breeze写的).

●协议序列化采用IDL工具生成的方案, 采用的开源库为proto4z. 通过xml配置文件生成原生的C++,lua,C#数据结构代码, 支持二进制协议和http协议, 并可以生成操作mysql的序列化/反序列化代码. 配合zsummerX可以很方便的在上述语言中混合通讯.

●日志库使用log4z.

●breeze的单组服务器设计容量在实际项目中, 轻度游戏10~100万同时在线, 有大量交互的模块和丰富的外围系统的中度/重度游戏1~10万+在线(瓶颈在于交互模块的单例service的负载能力, 通过对业务的把握尽量拆分更多的单例service, 转移负载到负载均衡的动态service上可以有效提升和突破这个上限), 数据库注册容量<=1000万(大于这个容量要考虑对数据库进行sharding, 垂直sharding直接根据业务进行分割, 添加更多的数据库service类型和配置即可, 而单表大于1000万需要水平sharding, 这个对dbhelper进行扩展后可以对业务层透明,但暂无需求 并没有去实现. , 另外需要注意减小不需要装载的service的驻留资源, 超规模使用资源不好挥霍的.).

# 十七. **Nana跨平台GUI库(C++11)**

使用简单，代码量少，基于C++11.

可以参考下这边文章: <https://blog.csdn.net/jinhao/article/details/7076810/>

# 十八. Mongoose(HttpServer库)

Cesanta开源的一个可嵌入的web server和可嵌入的网络库，提供httpServer库，支持很多的平台，支持epoll和select两种网络并发模型，也支持websoket

本机代码: E:\work\mongoose-master\mongoose-master

C++实现客户端与服务器的通信(一)

<https://blog.csdn.net/wwww1244/article/details/80973376>

利用Mongoose搭建一个http服务器

# 十九. [C++ actor framework](http://actor-framework.org)

<http://actor-framework.org/#components>

<https://github.com/actor-framework/actor-framework>

# 二十.玩转MYSQL

### 1.net start mysql 服务无法启动

安装完Mysql安装包后，net start mysql 启动不了，需要做：却说 mysql服务无法启动，服务没有报告任何错误。后来查看好像是my.ini跟data两个文件的问题所以

### 2.安装目录下，新建一个my.ini,内容如下:

**[mysql]**

**#设置mysql客户端默认字符集**

**default-character-set=utf8**

**[mysqld]**

**#设置3306端口**

**port = 3306**

**#设置mysql的安装目录**

**basedir=c:\mysql-8.0.12-winx64**

**#设置mysql数据库的数据的存放目录**

**datadir=c:\mysql-8.0.12-winx64\data**

**#允许最大连接数**

**max\_connections=200**

**#服务端使用的字符集默认为8比特编码的latin1字符集**

**character-set-server=utf8**

**#创建新表时将使用的默认存储引擎**

**default-storage-engine=INNODB**

### 3重点来了

**此处注意，我们无需自行创建 data 文件夹，待初始化会自行创建**

**进入命令行（管理员身份）：（进入 MySQL安装的 bin 目录下）  
1、mysqld --initialize（检查目录下存在 data 文件夹，删除即可）**

**2、net start mysql**

**然后就可以看到服务mysql启动成功.**

### 4找到你独一无二的初始登录密码：

在data目录下找到唯一一个数据库创建的日志文件，后缀名是.err  
用notepad++ 打开它，第二行最末尾就是你的初始密码，如下格式:

2018-05-30T03:06:28.382535Z 5 [Note] [MY-010454] [Server] A temporary password is generated for root@localhost: E2r;\*\*IpwwQQ

注意：末尾的 E2r;\*\*IpwwQQ 即为初始密码

若果不行，可以删掉data目录，执行mysqld –initialize –console，即可以在控制台看到这个唯一的密码。

### 5.接下来在命令行输入以下命令进行root用户登录：

mysql -u root -p

回车后，输入第4步你找到的初始密码，区分大小写，输完直接回车，出现以下提示则登录成功：

Welcome to the MySQL monitor

### 6.初始密码是不是很反人类？修改命令：

ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql\_native\_password BY '123456';

### 7.执行最后一个刷新MySQL系统权限的命令：

FLUSH PRIVILEGES;

总结:

新建表: Create table tb\_message(id varchar(20) , messgae varchar(20));

删除表: Drop table tb\_message;

清空表数据: Truncate table tb\_message;

插入数据: insert into tb\_message(id,message) values(“1”,”hello”);

修改数据: update tb\_message set message=”byebye”where id = “1”

8.mysql server对外提供服务端口 3306 ，10060

9.客户端安装了odbc驱动后，需要mysql server创建用户

Crate user ‘client’@’10.0.128.25’identified by ’MyPass@123’;

Grant all privileges on info.\* to ‘client’@’10.0.128.25’;

# 二十一. µWebSockets

代码地址: <https://github.com/uNetworking/uWebSockets>

最强websoickets框架

低层为libuv,OPENSSL.,ZLIB等第三方库。

# 二十二.WEBSOCKET++

代码地址: <https://github.com/zaphoyd/websocketpp>

优点：只有一个头文件，方便集成进应用。网络请求使用boost::asio实现（Windows上是IOCP完成端口）

主页: <https://www.zaphoyd.com/websocketpp/manual/building-program-websocket>

## 1．准备工作

### 1.1下载

直接在CTPServer工程里面加入头文件，见测试文件”test\_websocket\_client”,”test\_websocket\_server”.

## 2.

# 二十三.Casablanca

代码地址:

<https://github.com/microsoft/cpprestsdk>

主页:

<https://archive.codeplex.com/?p=casablanca>

# 二十四.tulipindicators

<https://github.com/TulipCharts/tulipindicators>

# 二十五.交易引擎.

参考最基础的交易引擎，自己编写模拟交易所。

主页: <https://github.com/sirak92/TradeMatchingEngine>

资料不齐全，项目作废

# 二十六.策略交易平台:

主页: <https://gitee.com/oyzhiquan/kernal>

本机代码: E:\work\oyzhiquan-kernal-master

经过下载编译，核心功能可以编译过，demo也可以编译通过，但是依赖库不全(E:\work\fastdb-master\fastdb-master)，主要是数据库和模块创建相关。

有价值的东西:

1.核心库的组织，策略系统是怎么组织的。

2.写策略是使用lua，怎么把策略转化为C++代码的。

测试工程: Direct + 这个代码的策略部分 = ？？？？

# 二十七 FastDB

# 二十八.Redis的使用

学习例子:

# 二十九.Nginx的学习

# 三十.game\_service\_system

<https://github.com/limingfan2016/game_service_system>

base lib, connect lib, db operator lib, develop frame, and game engine, game frame game service system! 从0开始开发 基础库（配置文件读写、日志、多线程、多进程、锁、对象引用计数、内存池、免锁消息队列、免锁数据缓冲区、进程信号、共享内存、定时器等等基础功能组件），网络库（socket、TCP、UDP、epoll机制、连接自动收发消息等等），数据库操作库（mysql，redis、memcache API 封装可直接调用），开发框架库（消息调度处理、自动连接管理、服务开发、游戏框架、服务间消息收发、消息通信等等），消息中间件服务（不同网络节点间自动传递收发消息）等多个功能组件、服务，最后完成一套完整的服务器引擎，基于该框架引擎可开发任意的网络服务。 主体架构：N网关+N服务+N数据库代理+内存DB（Redis、MemCache）+Mysql数据库，基于该架构可建立集群，稳定高效的处理大规模、高并发消息。

# 三十一.miniblink（未挖）

<https://github.com/weolar/miniblink49>

可嵌入式的超级小的浏览器内核，用来代替CEF和electronic

# 三十四 react（未挖）

React 是一个用于构建用户界面的 JAVASCRIPT 库。

React主要用于构建UI，很多人认为 React 是 MVC 中的 V（视图）。

React 起源于 Facebook 的内部项目，用来架设 Instagram 的网站，并于 2013 年 5 月开源。

React 拥有较高的性能，代码逻辑非常简单，越来越多的人已开始关注和使用它

# 三十五。VUE.JS（未挖）

三十六 .libcur

https://github.com/curl/curl

E:\work\curl-master\curl-master

# 思想学习-- 面向对象转向面向接口/面向数据编程

面向对象固然也有他的缺点：

1：相比面向过程来说，更难以理解，低水平的程序员开发的代码，耦合可能更高；

2：过度的封装也可能导致难以维护，新成员接受维护难度几何级加大；

3：一些简单的业务，使用面向过程开发反而更快速；

因此很多事情，不是偏左或者偏右就是好，组合取优才是硬道理。

于是在面向对象的基础上，就又有了面向接口编程(PO->OO->IO)。

面向接口从更深层次的反应了业务开发的本质，它轻松的融入了面向对象和面向过程的优点，更深层次的再次封装了对象个层次的综合业务需求为各种对外业务接口，彰显了业务中业务员对于系统的理解程度，从此编程不再是偏向技术，而是更偏向业务分析。它增强了对象与对象之间，模块与模块的之间的低耦合性，是软件系统更容易维护、扩展。

这里我说的面向接口编程，并非语言机制上的面向接口“编程”(JAVA/C#的interface，C++的抽象类)，我是在说“面向接口”编程，而不是面向接口“编程”。

在面向接口编程的概念中，一切业务均接口。接口抽象出业务员分析的需求(业务分析)，内部可以让程序员用各种面向对象机制来实现(抽象继承多态)。

## 【C++泛型编程】基于策略(Policy)的类设计

基于策略(Policy)的类设计是将templates和多重继承组合起来，这样可以产生程序库中的“设计元素”。Policies由templates和多重继承组成。一个class如果使用了policies,就称其为host class,那是一个拥有多个template参数的class template,每一个参数代表一个policy.host class所有的机能都来自policies,运作起来就像是一个聚合了数个policies的容器。

### 什么是Policy?

它是用来定义一个class或class template的接口，该接口由下面项目的之一或 全部组成：内隐型别定义(如typedef int \* pointer)、成员函数和成员变量。Policies有点类似设计模式中的Strategy,只不过policies更注重编译器。而Strategy更注重运行期。

例子1：定义一个ploicy 用来产生对象,Creator policy是一个带有型别为T的class template,它必须提供一个名为Create()的函数给外界使用，此函数不接受参数，返回一个指向T的指针。下面是三种产生对象的方法即policy：

1. Policy 1:
2. template<class T>
3. struct OpNewCreator
4. {
5. static T \* Create()
6. {
7. return new T;
8. }
9. };
11. Policy 2:
12. template<class T>
13. struct MallocCreator
14. {
15. static T \* Create()
16. {
17. void\* buf=std::malloc(sizeof(T));
18. if(!buf) return 0;
19. else
20. return new(buf) T;
21. }
22. };
24. Policy 3:
26. template<class T>
27. struct PrototypeCreator
28. {

31. PrototypeCreator(T\* pObj=0):pPrototype\_(pObj)
32. {}
33. T \* Create()
34. {
35. return ppPrototype\_? pPrototype\_->Clone():0;

38. }
39. T \*GetPrototype(){ return pPrototype\_;}
40. void setPrototype(T\* pObj){pPrototype\_=pObj;}
41. private:
42. T \* pPrototype\_;
43. };

上面三个policy称为policy classes.这些东西并不是用来单独使用，主要是用于继承或被内含于其他的类。  
Creator policy并没有规定Create()必须是static 还是virtual,只要求class必须定义一个Create().你可以定义多个policy class,但是必须遵循统一的接口。

### 2.利用前面的Creator Policy,怎么样设计一个类？

我们可以用复合或继承的方式使用前面定义的三个类之一。

例如：

1. template<class CreationPolicy>
2. class WidgetManager:public CreationPolicy
3. { };

如果一个类采用了一个或多个policy class,就称其为host或host classes.上面定义的WidgetManager采用了一个policy,称其为host class.Hosts主要负责把上面定义的policies的结构和行为组成一个风复杂的结构和行为。  
  
用户可以将WidgetManager实例化，传进一个用户指定的Policy,如下：  
typedef WidgetManager<OpNewCreator<Widget> > MyWidgetPtr;  
当一个MyWidgetPtr需要产生一个Widget对象时，它就调用它的policy子对象OpNewCreator<Widget>所提供的Create()函数。WidgetManager是用来选择生成策略(Creation policy)的(基于Policy设计类的宗旨)。

### 3.怎么样为host class 中的Policy classes的参数设定缺省值?

在前面可以看到，policy 类的template参数是非常累赘的。使用者每次都需要向Policy 类(如OpNewCreator)传递一个template参数，表示需要创建的对象。如果使用者操作的对象固定，就不需要每次都传递这个参数，可以采用默认的对象来实例化。  
  
**a.采用Template Template参数来设定缺省值**

1. template <template<class Created> class CreationPolicy>
2. class WidgetManager:public CreationPolicy<Widget>
3. {.......}

//Created 是CreationPolicy的参数。CreationPolicy 是WidgetManager的参数。上面的Widget已经写入程序库，使用时不需要再传一次参数给policy.可以采用如下方式使用：  
typedef WidgetManager<OpNewCreator> MyWidgetMgr;  
  
如果用户想要用WidgetManger的相同策略产生一个Gadget对象，如下：

1. template<template<class> class CreationPolicy>
2. class WidgetManager:public CreationPolicy<Widget>
3. {
4. void DoSomething()
5. {
6. Gadget \*pW=CreationPolicy<Gadget>.Create();
7. }
8. };

注：为常用的policy提供缺省的参数：

1. template <template<class> class CreationPolicy=OPNewCreator>
2. class WidgetManager...;

policies有丰富的型别信息及静态连接等特性，在编译期才把host class和policies结合在一起，所以是建立“设计元素”时的本质性东西。  
  
**b.利用Template成员函数来定义policy class。**  
  
       我们可以重新定义之前的policy为非template class,该函数内部定义一个template函数，如下：

1. struct OpNewCreator
2. {
3. template<class T>
4. static T \*Create()
5. {

8. return new T;
9. }
10. };

这样定义的policy,对旧的编译器比较兼容。这样的policy难以讨论，定义和运用。

### 4.Policy Classes的析构函数

大部分情况下，host class会以public方式继承某些policies，所以调用者可以将一个host class自动转为一个policy class,并delete 该指针。除非为policy class定义虚析构函数，否则delete一个执行policy class的指针，会产生不可预期的结果。然而如果为policy定一个虚析构函数，会影响policy的静态链接特性，也会影响执行效率。  
许多policies并没有数据成员，只纯粹提供一个接口。虚函数的加入会为对象的大学带来额外开销，所以虚函数应该尽量避免。  
 一个解决办法是host class 派生policy class时候，采用protected或private继承。但这样会失去丰富的policies特性。  
另一个有效的解法是定义一个protected的析构函数。由于是protected，只有派生而得的类才可以摧毁这个policy对象。这样外界就不可能delete一个指向policy class的指针。由于析构函数不是虚函数，不会带来大小或速度上的额外开销。

### 5.通过不完全具现化获取选择机能

C++的有趣特性，造就了policies的丰富特性。如果class template有一个成员函数没有用到，它就不会被编译器具体实现出来。编译器不会理会，甚至不进行语法检验。这样就可以让host class有机会并使用policy class的可选特性。如下：

1. template<template<class> class CreationPolicy>
2. class WidgetManage:public CreationPolicy<Widget>
3. {
4. void SwitchPrototype(Widget \*pNewPrototype)
5. {
6. CreationPolicy<Widget> &myPolicy=\*this;
7. delete myPolicy.GetPrototype();
8. myPolicy.SetPrototype(pNewPrototype);
9. }
10. };

如果你采用一个定义了SetPrototype()函数的Creator policy来实例化WidgetManager，就可以使用SwitchPrototype()函数。若采用不支持SetPrototype()函数的Creator policy来实例化WidgetManager，就会参数编译错误。如果你采用一个不支持SetPrototype()函数的Creator policy来实例化WidgetManager，如果使用在没有调用SwitchPrototype(),程序是合法的。

**6.用Policy class定制结构**

       templates的限制之一是你无法定制class的结构，只能定制其行为。然而基于policy 类的设计支持结构方面的定制。  
一般像如下定义：

1. template<class T>
2. class SmartPtrStorage
3. {
4. public:
5. typedef T \*PointerType;
6. typedef T& ReferenceType;
7. protected:
8. PointerType GetPointer(){ return pointee\_;}
9. void SetPointer(PointerType ptr){pointee\_=ptr;}
10. private:
11. PointerType pointee\_;

# ★[C++11模版元编程](https://www.cnblogs.com/qicosmos/p/4480460.html)

<http://www.cnblogs.com/qicosmos/p/4480460.html>

# ★[C++11模版元编程的应用](https://www.cnblogs.com/qicosmos/p/4772328.html)

http://www.cnblogs.com/qicosmos/p/4772328.html

# ★网络书籍资料

<https://github.com/ketoo/free-programming-books-zh_CN>

# ★[C++11的应用](https://www.cnblogs.com/qicosmos/p/4772328.html)

## Std::fuction

c++11中推出function是为了泛化函数对象，函数指针，引用函数，成员函数的指针，让我们可以按更统一的方式写出更加泛化的代码；推出bind是为了替换和增强之前标准库的bind1st和bind2st，让我们的用起来更方便！

std::function取代传统的定义虚函数的回调函数，提供了函数指针，成员函数只针，函数对象（lamda）的调用。。当然，他与bind和lambda一起工作，大大提高了回调的使用机会，常用于：信号/接受者的实现，GUI和业务逻辑的分离，以及再标准容器中存储不同的函数类型。

### Example

#include <functional>

#include <iostream>

struct Foo

{

Foo(int num) : num\_(num) {}

void print\_add(int i) const { [std::cout](http://en.cppreference.com/w/cpp/io/cout) << num\_+i << '**\n**'; }

int num\_;

};

void print\_num(int i)

{

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << i << '**\n**';

}

struct PrintNum

{

void operator()(int i) const

{

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << i << '**\n**';

}

};

int main()

{

// store a free function

std::function<void(int)> f\_display = print\_num;

f\_display(-9);

// store a lambda

std::function<void()> f\_display\_42 = []() { print\_num(42); };

f\_display\_42();

// store the result of a call to std::bind

std::function<void()> f\_display\_31337 = [std::bind](http://en.cppreference.com/w/cpp/utility/functional/bind)(print\_num, 31337);

f\_display\_31337();

// store a call to a member function

std::function<void(const Foo&, int)> f\_add\_display = &Foo::print\_add;

const Foo foo(314159);

f\_add\_display(foo, 1);

f\_add\_display(314159, 1);

// store a call to a data member accessor

std::function<int(Foo const&)> f\_num = &Foo::num\_;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "num\_: " << f\_num(foo) << '**\n**';

// store a call to a member function and object

using std::placeholders::\_1;

std::function<void(int)> f\_add\_display2 = [std::bind](http://en.cppreference.com/w/cpp/utility/functional/bind)( &Foo::print\_add, foo, \_1 );

f\_add\_display2(2);

// store a call to a member function and object ptr

std::function<void(int)> f\_add\_display3 = [std::bind](http://en.cppreference.com/w/cpp/utility/functional/bind)( &Foo::print\_add, &foo, \_1 );//placeholders\_1;

f\_add\_display3(3);

// store a call to a function object

std::function<void(int)> f\_display\_obj = PrintNum();

f\_display\_obj(18);

}

Output:

-9

42

31337

314160

314160

num\_: 314159

314161

314162

18

## std::bind

### 将函数、成员函数和闭包转成function函数对象

### 将多元(n>1)函数转成一元函数或者(n-1)元函数。

void H(int a);

//绑定全局函数

auto f11 = std::bind(H, std::placeholders::\_1);

auto的类型实际上是std::function<void(int)>

//绑定带参数的成员函数

std::function<void (char\*, int)> f = std::bind(&ReadHandler::ConnectPreProcess, this, std::placeholders::\_1, std::placeholders::\_1);

//三元函数转换成一元函数

int f(int, char, double);

// 绑定f()函数调用的第二个和第三个参数，

// 返回一个新的函数对象为ff，它只带有一个int类型的参数

auto ff = bind(f, \_1, ‘c’, 1.2);

int x = ff(7);

### bind简化和增强bind1st和bind2nd

其实bind简化和增强了之前标准库中bind1st和bind2nd，它完全可以替代bind1s和bind2st，并且能组合函数。我们知道bind1st和bind2nd将一个二元算子转换成一个一元算子。

//查找元素值大于10的元素的个数

int count = count\_if(coll.begin(), coll.end(), std::bind1st(less<int>(), 10));

//查找元素之小于10的元素

int count = count\_if(coll.begin(), coll.end(), std::bind2nd(less<int>(), 10));

本质上是对一个二元函数less<int>的调用，但是它却要分别用bind1st和bind2nd，而且还要想想到底是用bind1st还是bind2nd，用起来感觉不方便。现在用bind，就可以以统一的方式去实现了。

//查找元素值大于10的元素的个数

int count = count\_if(coll.begin(), coll.end(), bind(less<int>(), 10, \_1));

//查找元素之小于10的元素

int count = count\_if(coll.begin(), coll.end(), bind(less<int>(), \_1, 10));

这样我就不用关心到底是用bind1st还是bind2nd了，只要用bind就都搞定了。

### bind函数组合

bind的还有一个强大之处就是函数组合。假设我们要找出集合中大于5小于10的元素个数怎么做呢？我们需要一个逻辑与才能做到类似于：

std::bind(std::logical\_and<bool>(),\_1,\_2);

然后，我们需要另一个谓词来回答 \_1 是否大于5。

std::bind(std::greater<int>(),\_1,5);

然后，我们还需要另一个谓词来回答 \_1 是否小于等于10。

std::bind(std::less\_equal<int>(),\_1,10);

最后，我们需要把它们两个用逻辑与合起来，就象这样：

//查找集合中大于5小于10的元素个数

auto f = bind(std::logical\_and<bool>(), bind(std::greater<int>(),\_1,5), bind(std::less\_equal<int>(),\_1,10));

int count = count\_if(coll.begin(), coll.end(), f);

# ★C++代码资源

<https://github.com/fffaraz/awesome-cpp#standard-libraries>

<http://top.jobbole.com/14380/>

中文资源:

<https://github.com/ketoo/free-programming-books-zh_CN#rust>

<https://github.com/ketoo/free-programming-books-zh_CN#cc>

C++并发指南：

<https://github.com/forhappy/Cplusplus-Concurrency-In-Practice>

<https://chenxiaowei.gitbooks.io/cpp_concurrency_in_action/content/content/appendix_D/D.1-chinese.html>

在线编写API:

<https://legacy.gitbook.com/>

一个合格的程序员应该读过哪些书:

<http://justjavac.com/other/2012/05/15/qualified-programmer-should-read-what-books.html>

C++ STL 内存配置的设计思想与关键源码分析:

<http://blog.jobbole.com/109627/>

人人都能读懂的设计模式（1）：创建型模式

<http://blog.jobbole.com/111799/?utm_source=blog.jobbole.com&utm_medium=relatedPosts>

C++ Template 进阶指南

<https://github.com/wuye9036/CppTemplateTutorial#c-template-%E8%BF%9B%E9%98%B6%E6%8C%87%E5%8D%97>

设计模式代码

<https://gitee.com/micooz/DesignPattern>

<https://github.com/micooz/DesignPattern>