模块分类

参考资料

使用

数据结构

文件关系

模块设计

-----------------------------------

模块分类

实现模块

基础服务：集群模块 消息处理模块 数据库模块

gateserver: 网关模块 http模块 websocket模块

authserver: 网关模块 断线重连模块 排队认证模块

loginserver: 登录模块

hallserver: game模块

locatorserver: locator模块

gameserver: game模块

chatserver: chat模块

dbserver: 数据库模块

待实现模块

AOI模块

日志染色

热更新

宕机重启 redis存放在线用户的rpc请求

限流降级

redis mysql事务

垂直分布

auth 认证、注册、排队

login 登录

hall 平台 (认证、第三方、支付、邮件)

area (mmo)

room (副本/moba/rpg)

desk (team) 同服组队，或者跨服组队(desk服务独立于room实现)

locator 辅助启动、负载均衡gameserver

chat 聊天

注意：

redis无法存放int数据

pkill skynet;rm -f console.log;touch console.log;./start.sh 1;cat console.log;tail -f console.log

-----------------------------------

参考资料

skynet.stm 单节点内服务间共享配置数据

https://blog.csdn.net/lzb991435344/article/details/77182125

https://github.com/cloudwu/skynet/wiki/ShareData

RedisMQ发布与订阅

https://blog.csdn.net/u011499747/article/details/51232981

redis事务和watch使用

https://www.cnblogs.com/liuchuanfeng/p/7190654.html

redis订阅与watch相关例子

local function watching()

local w = redis.watch(conf)

w:subscribe "foo"

w:psubscribe "hello.\*"

while true do

print("Watch", w:message())

end

end

skynet.Coroutine

skynet.coroutine.thread(co) ，它返回两个值，第一个是该 co 是由哪个 skynet thread 间接调用的。

如果 co 就是一个 skynet thread ，那么这个值和 coroutine.running() 一致，且第二个返回值为 true ，否则第二个返回值为 false 。

https://github.com/cloudwu/skynet/wiki/Coroutine

skynet.cluster

cluster.reload(config) 加载配置

cluster.open(current\_conf.nodename) 监听节点

cluster.call("db", ".simpledb", "GET", "a") 请求

cluster.send(nodename, service, ...) 单向推送，可能会有数据丢失风险而发送者不知道数据丢失了

https://github.com/cloudwu/skynet/wiki/Cluster

skynet.queue临界区 保护一段代码不被同时运行、按队列顺序执行

https://github.com/cloudwu/skynet/wiki/CriticalSection

skynet.profile

统计每一个消息处理的时间 然后针对时间较长的消息做优化

http://blog.sina.com.cn/s/blog\_7f6c94b60102wiup.html

skynet.sharedata.corelib

相关文件config\_db config\_helper

https://github.com/cloudwu/skynet/wiki/ShareData

lua垃圾回收collectgarbage

https://blog.csdn.net/ecidevilin/article/details/53326411

guid 随机数设计

/dev/urandom

https://www.cnblogs.com/Flychown/p/6868520.html

lua重新加载文件

package.loaded[pathprefix .. path] = nil

https://blog.csdn.net/themagickeyjianan/article/details/70676172

skynet websocket实现 集成到skynet目录下

https://github.com/sctangqiang/skynetpatch

skynet相关

同一节点=同一进程

进程内有很多服务，通过sharedata共享数据，cluster共享配置文件就是这么做的，如果需要跨节点，需要自行同步处理

sproto相关

第一种 sproto.new？ host2？ 参考 https://blog.csdn.net/woxiaohahaa/article/details/78046050

proto.c2s = sprotoparser.parse [[ ]]

proto.s2c = sprotoparser.parse [[ ]]

client:

local host = sproto.new(proto.s2c):host "package"

local request = host:attach(sproto.new(proto.c2s))

local host2 = sproto.new(proto.c2s):host "package"

local req\_type, name, arg, func = host2:dispatch(str)

server:

host = sproto.new(proto.c2s):host "package"

request = host:attach(sproto.new(proto.c2s))

host2 = sproto.new(proto.s2c):host "package"

host:dispatch(str)

第二种

local f = assert(io.open(name .. ".s2c.sproto"))

local t = f:read "a"

var.host = sproto.parse(t):host "package"

local f = assert(io.open(name .. ".c2s.sproto"))

local t = f:read "a"

var.request = var.host:attach(sproto.parse(t))

local t, session\_id, resp, err = var.host:dispatch(msg)

-----------------------------------

使用

下载

https://github.com/zhangshiqian1214/skynet-server

修改：

1 顶层makefile添加lsocket库支持

2 skynet makefile添加 skynet\_crypt httppack add支持

lualib-src/lua-crypt.c lualib-src/lua-httppack.c lualib-src/add.c

安装

skynet: yum install dos2unix; yum install libcurl-dev libcurl-devel

make socket 安装客户端lsocket

问题： 安装 Redis 执行 make #error "Newer version of jemalloc required"

=> make MALLOC=libc

openssl/crypto.h: No such file or directory

=> yum install openssl-devel

“uuid/uuid.h: No such file or directory

=> yum install libuuid-devel

测试cluster：1 ./run\_test1.sh ./run\_test.sh

运行

1 只启动gate login hall db服务，redis没有其他服务

单服务启动

./run\_redis.sh

./run\_gate.sh

./run\_login.sh

./run\_hall.sh

./run\_db.sh

单服务停止

pkill -u `whoami` -xf "./skynet/skynet ./config/config\_xpnn \_\_default\_\_"

停止game服务

pkill -u `whoami` -xf "./skynet/skynet ./config/config\_xpnn21 \_\_default\_\_"

2 启动客户端

./client.sh 1

控制台输入0 正常退出客户端

控制台ctrl+c 断线(socket直接断开)

重启所有服务

./restart.sh

-----------------------------------

数据结构

1 内存数据

proto\_map[protos] =

table: 0028E168 {

[module] => "player"

[type] => 1

[name] => "get\_weixinpay\_info"

[fullname] => "player.get\_weixinpay\_info"

[id] => 774

[is\_agent] => true

[desc] => "获取微信支付信息"

[response] => "player.WeixinPayInfo"

}

gate\_mgr.connections

//gate.connect建立连接时设置

uuid.seed()

local c = {}

c.fd = fd

c.ip = string.match(ip, "([%d.]+):(%d+)")

c.session = uuid()

//hall.agent.login时设置

c.player\_id = player\_id

c.hall\_agentnode = hall

c.hall\_agentaddr = hall.agent地址 没体现出作用

c.auth\_ok = true

//area.get\_role/create\_role时设置

c.role\_id = role\_id

//area.enter\_area时设置 xxxxxxxxxxx

c.agentnode = xpnn1

c.agentver = xpnn1 ver

c.game\_id = 101

//room.agent.login时设置

c.agentaddr = room.agent地址 用于desk调用其它服务的媒介

//area.enter\_room时设置

c.roomaddr = roomaddr

client\_msg.get\_context

ctx ={

//socket\_msg.data(socket\_msg.c对象) => client\_msg.dispatch时设置

ctx.gate = cluster\_monitor.get\_current\_nodename()

ctx.watchdog = skynet.self()

ctx.is\_websocket = gate\_mgr.is\_websocket()

ctx.fd = c.fd

ctx.ip = c.ip

ctx.session = c.session

//hall.agent.login的下个请求时设置

ctx.player\_id = c.player\_id

//area.get\_role/create\_role的下个请求时设置

ctx.role\_id = c.role\_id

ctx.game\_id = c.game\_id

}

2 redis格式数据

db0

//节点启动时设置

cluster\_nodes

locator: {"intranetip":"127.0.0.1","extranetip":"127.0.0.1","use\_intranet":1,"ver":151,"servertype":6,"nodeport":9006,"serverid":6,"nodename":"locator","is\_online":1}

incr\_player\_id player\_id自增值

incr\_role\_id role\_id自增值

db1

player\_online:1 {

//hall.agent.login时设置

session=xxx

state=1 玩家位置状态标志

player\_id=3

fd=19

watchdog=12

ip=127.0.0.1

gate=gate1

hall\_agentnode=hall

//断线、login.signin\_account时设置

offline = 0

//area.get\_role/create\_role/exit\_area时设置

role\_id=xxx

//area.enter\_area时设置

agentnode=xpnn

agentver= xpnn .ver

server\_id=xxx

game\_id=xxx

//room.agent.login时设置

agentaddr=15 room.agent地址

//room.enter\_room时设置

roomaddr=xxx

}

cache\_info:player\_id:pack\_id 缓存返回客户端数据

proto\_id: {"ec":0,"data":{"player":{"nickname":"止痛药也是我的错","create\_time":1536652735,"sex":1,"head\_id":1,"head\_url":"","player\_id":1,"gold":100}},"proto\_id":2305}

auth\_info 缓存认证到登录的验证数据

player\_id: {"waitnum":0,"login\_addr":17,"pid":1,"secret":"x \r ubM","subid":1,"waitsecond":0}

login\_addr sub\_id //登录地址 登录号码牌

secret //login.signin\_account 账号方式登录密码解析

pid // pid=-1表示断线重连情况

db2 仅用于locator分配gameserver节点

fdnum\_room:server\_id@room\_id: fdnum : room\_addr 记录所有server\_id上各个room\_id的各个room的fdnum

fdnum\_server:game\_id fdnum : server\_id 记录所有game\_id下各个server\_id的fdnum

readystart\_server:game\_id {server\_id} 记录未启动的server\_id

readystop\_server: {server\_id} 记录因为人数扣为0，等待被关闭的server\_id

db3

//room.login时设置

player\_info:player\_id {head\_url:xx sex=1 nickname=xixi create\_time=xxxxxx player\_id=3 head\_id=1 gold=100}

//area.get\_role/create\_role时设置

role\_info:role\_id {head\_url:xx sex=1 nickname=xixi create\_time=xxxxxx player\_id=3 head\_id=1 gold=100 role\_id=4 game\_id=101}

db4 仅用于chat模块

fd\_hall {fd} 进入大厅

fd\_game:game\_id {fd} 进入area

fd\_server:server\_id {fd} 进入area

fd\_roomtype:room\_id {fd} 进入room

fd\_roomaddr:server\_id@room\_addr {fd} 进入room

-----------------------------------

文件关系

2 调用关系

cluster\_monitor => cluster\_monitord => cluster\_mgr

3 文件关系

集群相关

cluster\_monitor cluster\_monitord cluster\_mgr connector share\_memory redis\_mq

消息处理相关

preload: server\_define service\_define module\_define proto\_map

lualib: sproto\_helper dispatcher(=>watchdog) requester context service\_base service/\*

logic: gate\_msg client\_msg socket\_msg

service目录相关

dbserver master\_db master\_db\_svc db\_mgr(db\_config mysql\_config redis\_config) db\_module db\_define db\_helper(db\_node节点 svc服务)

gameserver/xpnn room

hallserver hall hall\_impl hall\_ctrl hall\_logic\_svc hall\_logic hall\_db agent config\_db agent\_ctrl player\_ctrl

loginserver auth auth\_impl auth\_ctrl auth\_logic\_svc auth\_logic auth\_db webclient

gateserver watchdog wswatchdog socket\_msg gate\_msg gate\_mgr client\_msg(被socket\_msg调用) gate wsgate不存在 logic/gate相关

-----------------------------------

数据库模块

配置 db\_config mysql\_config redis\_config

db\_module 动态加载\_db文件（例如logic/hall/hall\_db.lua）

db\_mgr 提供redis mysql句柄

master\_db 创建、管理master\_db\_svc的地址

master\_db\_svc 分发消息给db\_module处理并返回结果，服务提供者

db\_helper 被logic调用，服务消费者

启动流程

dbserver/main => 创建master\_db => master\_db.start => 创建多个master\_db\_svc => master\_db\_svc.start => db\_mgr.init 构造mysql、redis句柄

(db\_config[svc\_name] => conf => mysql\_config[conf.mysql\_id] redis\_config[conf.redis\_id] => 句柄)

使用流程

logic => db\_helper.call/send => master\_db\_svc别名(bname + id => db\_config[dbname] 根据conf的get\_svc和service\_name) =>

(nodename+svc+method+id) => context.rpc => master\_db\_svc.dispatch => 调用db\_module(cmd => modname funcName => db\_module[modname][funcName]) =>

数据返回logic

-----------------------------------

集群模块

配置 cluster\_config redis\_config

redis\_mq 监听节点的加入、退出

share\_memory 共享cluster配置数据

connector 集群节点在线监控

cluster\_monitor 入口服务

1 管理cluseter\_nodes、 current\_node

2 订阅节点：self => cluster\_mgr 向cluster\_mgr注册connect服务，添加subscribe\_nodes[nodename] = callback

3 调用cluster\_mgr start、open

cluster\_mgr

1 redis msg redis\_mq订阅者接收回调

add\_cluster\_node

cache\_cluster\_conf conf => 内存

reload\_cluster\_conf 内存 => cluster.reload

reset\_connectors current\_conf => 向cluster\_monitor发送connect请求（为了注册monitor，目前没有用）

2 cluster\_mgr

3 subscribe 设置订阅地址？

subscribe\_monitor addr =>

unsubscribe\_monitor addr =>

4 cluster memory

cache\_cluster\_conf conf => share\_memory["cluster\_nodes"] 更新share\_memory内存对象，如果是当前配置同时更新share\_memory["current\_nodename"]

remove\_cluster\_conf

5 cluster redis

load\_conf\_from\_redis redis遍历cluster\_conf => 更新版本号，使当前conf版本+1 + cache\_cluster\_conf更新内存数据

cache\_conf\_to\_redis cluster\_conf => redis, redis使用conf.nodename索引

remove\_conf\_from\_redis

6 skynet.cluster conf

reload\_cluster\_conf 从内存中获取所有cluster\_conf，调用cluster.reload

7 connector mgr 版本更新、新结点添加connect操作，根据内存cluster\_node判断停止下线结点的connect

reset\_connectors 连接成功调check，连接失败调disconnect\_callback，连接断开会继续调用connect ?

\_connect\_func current\_conf => 向cluster\_monitor发送connect请求

\_connect\_callback conf => 向订阅地址发送monitor\_node\_change请求

\_disconnect\_callback conf => 向订阅地址发送monitor\_node\_change请求 + 更新内存cluster\_conf

8 init

start 当前节点加载其它节点配置

1 设置redis配置，当前cluster配置

2 构造redis\_mq，订阅cluster\_mgr.消息，启动订阅监听

3 load\_conf\_from\_redis redis => 内存

4 reload\_cluster\_conf 内存 => cluster.reload

5 reset\_connectors current\_conf => 向cluster\_monitor发送connect请求（没有用到）

open 当前节点启动通知其它节点

1 cluster.open当前node

2 cache\_conf\_to\_redis cluster\_conf => redis

3 发布cluster\_mgr.add\_cluster\_node消息

4 cluster启动gate服务 => 启动gateserver监听当前cluster的nodeport(重要)

-----------------------------------

消息处理模块

module\_define 处理客户端消息 client\_msg.dispatch

proto\_map 定义proto相关

sproto\_helper

1 register\_protos sproto.loader按照proto\_map.PROTO\_FILES.id加载spb文件

2 打包：pack sp:encode("Package", header) + sp:encode(proto.request, data) => sproto.pack(binary) => binary

解包： 方式1：unpack sproto.unpack(msg, sz)+ sp:decode("Package", binary) => header+ 未解密data =>

sp:decode(proto.response, content) => 解密后data => header, result

方式2：unpack\_data 测试websocket 未解密data => sp:decode(proto.response, content) => header, result

requester 调用skynet和cluster发送消息

1 call send 同节点消息发送 service + cmd + args => service.dispatcher.dispatch\_service\_msg => service.impl

2 rpc\_call rpc\_send 不同节点rpc消息发送 node + service + cmd + args => service.dispatcher.dispatch\_service\_msg => service.impl

3 send\_client\_msg 发送消息给客户端 ctx + proto + header + data => ctx.gateNode.watchdog.send\_client\_msg(ctx.fd, buffer)

context 调用request发送消息服务端、客户端

service\_base 导出的module被dispatcher调用(如service.modules.auth = auth\_impl)

通过继承，实现service\_base.modules[modname][funcName] => 方法

dispatcher 被service\_base调用，处理client消息、其它service方法并返回,是接收方的入口函数

dispatch\_client\_msg service\_base调用,处理客户端消息，使用sproto\_helper解包，调用response\_client\_msg返回消息。

proto => funcName=service\_base.modules[proto.module][proto.name] => response\_client\_msg

response\_client\_msg 内部调用，返回消息给client ctx.gateNode.watchdog.send\_client\_msg(ctx.fd, buffer)

dispatch\_service\_msg service\_base调用，根据service.func请求并返回处理结果

调用过程（重要!!）：logic => context => requester => service\_base => dispatcher

method => modname, funcname => funcName=service\_base.modules[proto.module][proto.name] => ret

-----------------------------------

http模块

webclient 客户端

旧模块(长连接??)

启动流程

1 创建watchdog服务 => 创建http服务

2 watchdog.start => http.cmd.open => socketdriver.listen

建立连接

http.msg.open => wathdog.http.open(无用)

接收数据

http.msg.data => watchdog.post/watchdog.get 直接返回data => 数据处理(待修改) => http.msg.data返回

新模块 短连接

httpServer

httpServerAgent

-----------------------------------

websocket模块（skynet/service目录下有agent）

skynet/service/wsclient websocket测试客户端

skynet/service/wsgate对象

local connection = {} -- connection[fd] = { fd , client, agent , ip, mode }

local forwarding = {} -- forwarding[agent] = connection

skynet/service/wswatchdog对象

local agent = {} -- agent[fd] = agent

启动流程

1 wsmain => 创建wswatchdog => 创建wsgate => wsgateserver.start

2 wsmain.start => wswatchdog.start => wsgate.handler.open 设置conf

连接流程

wsgate.handler.connect => wswatchdog.socket.open => 创建agent+agent.start => wsgate.forward => wsgate构造对象+gateserver.openclient

接收返回数据

wsgate.handler.message => agent.dispatch.client => socket.write 返回数据

退出流程

wsgate.handler.message => agent.quit => wswatchdog.close => wsgate.kick+gateserver.closeclient + agent.disconnect =>

wsgate.handler.disconnect+wsgate.unforward+gateserver.closesocket => wswatchdog.socket.close

-----------------------------------

网关模块(socket和websocket,没有agent)

gate\_mgr 管理[fd]=socket\_msg.c对象

init => 创建gate服务

add\_connection fd => 添加客户端fd

socket\_msg 用于接收client消息

oepn fd+ip => 创建socket\_msg.c对象

data fd+msg => client\_msg.dispatch

gate\_msg 被逻辑调用返回给client

start conf => 启动gate.open

monitor\_node\_change conf.nodename => cluster\_monitor.callback ??

send\_client\_msg 1 ws解包发送给gate 2 非ws直接发送给gate fd+buffer => gate.send\_buffer

//退出

第一种 玩家请求退出 hall\_logic.cast\_logout => gate\_msg.logout

第二种 玩家断线超时退出 heartbeat\_ctrl.do\_kick\_work => gate\_msg.player\_leave

第三种 玩家重复登录被踢出 hall\_logic.cast\_login => gate\_msg.player\_leave

第四种 玩家断线登出(不清除玩家数据) heartbeat\_ctrl.do\_offline\_work/socket\_msg.close/error => gate\_msg.close

第五种 玩家认证失败断开连接(认证服) auth\_logic.auth\_secret => gate\_msg.close\_fd

player\_leave 玩家完全退出 hall\_logic.logout\_account =>

第一步 fd.unset\_fd 清除redis fd

第二步 update\_player\_online 更新redis player\_online

第三步 remove\_cache\_info 清除redis cache\_info

第四步 gate\_msg.logout

logout 玩家完全退出

第一步 heartbeat\_ctrl.del\_playerId 停止心跳计算

第二步 gate\_mgr.close\_connection 清空socket\_msg.c对象

第三步 gate.kick 关闭socket

close\_fd 关闭socket连接 fd => gate.kick

close 更新断线状态 player\_id => player\_online.offline=on

//设置socket\_msg.c对象

login\_ok //hall.agent.login

set\_agent //room.agent.login

set\_c\_context //断线重登 恢复socket\_msg.c和redis数据,关闭src\_fd

enter\_area/exit\_area

set\_role\_id

set\_room\_addr

login\_desk/logout\_desk

client\_msg 被socket\_msg调用

get\_context 内部调用 设置context内容

client\_msg.send 内部调用，返回消息给gate header+data => gate.send\_buffer

client\_msg.dispatch header => proto => node+service => context.rpc\_call.dispatch\_client\_msg

gateserver socket连接模块

gate socket连接管理模块

open 设置watchdog

connect fd+addr => socket\_msg.open

message fd+msg+sz => watchdog.socket => socket\_msg.data 接收处理msg

cmd.send\_buffer fd+buffer => gateserver.send\_buffer 发送消息

disconnect fd => socket\_msg.close

error fd+msg => socket\_msg.error

kick fd => gateserver.closeclient 关闭socket连接

watchdog

1 cmd消息

agent\_ctrl => watchdog.login\_ok/set\_agent

desk\_ctrl => watchdog.login\_desk/logout\_desk

hall\_logic => watchdog.kick\_player

dispatcher.response\_client\_msg => watchdog.send\_client\_msg(从这里开始都是复用接口) => gate\_msg => gate.send\_buffer =>

snax.gateServer.send\_buffer => socketdriver.send(fd,buffer) 发送消息给client(接口1)

2 socket消息

gate.socket => watchdog.socket => socket\_msg => client\_msg.dispatch 接收client消息

启动流程

gateserver/main =>

第一步 创建watchdog => gate\_mgr.init => 创建gate

第二步 watchdog.start => gate\_msg.start => gate.handler.open 设置conf

连接流程

gate.handler.connect => watchdog.socket.open => socket\_msg.open + gate\_mgr.add\_connection

接收返回数据

gate.handler.message => watchdog.socket.data => socket\_msg.data => client\_msg.dispatch =>

客户端入口 context.rpc\_call.dispatch\_client\_msg(ctx, msg协议包) => requester.rpc\_call(cluster.call) => command\_base.dispatch =>

service\_base.command.dispatch\_client\_msg => dispatcher.dispatch\_client\_msg =>

service.module[modulename] => impl(必须return) =>

服务端入口 context.rpc\_call.xxx(msg参数表) => requester.rpc\_call => command\_base.dispatch => service.command.xxx =>

c ctrl 返回数据 => dispatcher.response\_client\_msg 返回数据给客户端

退出流程

gate.handler.disconnect+gateserver.closesocket => watchdog.socket.close => socket\_msg.close => gate\_mgr.close\_connection

待解决：

1 多台gate 登录 退出

-----------------------------------

断线重连模块

断线、心跳、延迟关系和处理

http://blog.oraycn.com/ESFramework\_07.aspx

数据结构

定时器检查 player\_id {updatetime state} state=0正常(<3\*5) state=1断线(>3\*5) state=2可重连期(<30\*60\*3) M=3 N=5 T=30\*60

proto添加pid

客户端缓存request数据包

redis格式数据

player\_online添加offline

cache\_info player\_id {pid {json respnse/push}} pid < 100为进入场景等预留包，使用重复pid值只更不删除 缓存100-150条不断更新重用。

游戏中

1 客户端缓存request包，收到后清除 pid对应

2 服务端缓存每玩家后50条pid和对应的response/push包(不能是最大pid判断)

3 客户端定时发送心跳，服务端心跳处理：更新player\_id {updatetime state} 返回数据包 3s发一次

4 服务器 gate 接收更新心跳 返回心跳 处理心跳

正常登出逻辑

登出服务端 清空cache\_info 清空plyaer\_id{updatetime state} 关闭sock

断线检测处理

1 networkException(需要double check) socket\_msg.close/error

2 延迟、丢包(客户端一定时间没收到心跳返回包)

=> 客户端低频率尝试重连5次

3 服务端心跳处理，返回包。一定时间没收到心跳触发定时器 offline=1

4 服务端收到networkException(需要double check) close 触发定时器 offline=1

=> 断线处理

关闭socket连接;定时器超时，执行正常登出逻辑

重连成功

=> auth.login player\_online offline+state=>重复登录剔除？？？ offline =>断线重连 state => 重新登录

1 结束定时器，offline=0

2 ctx sock\_msg替换fd

第一种

重发请求，根据pid判断是否处理

重构socket\_msg.c offline = 1

重传请求 存在cache就重传

第二种

重新请求登录逻辑，执行负重传请求

push人物属性、场景数据

push场景数据

启动流程：

heartbeat\_ctrl.init => cal\_offline\_player 定时检测在线、心跳超时检测处理、断线超时检测处理

接收心跳流程

client\_msg.dispatch (protoid==0x0002 心跳包) 直接返回客户端心跳 + heartbeat\_ctrl.reset\_updatetime 更新心跳时间

断线重连

client\_msg.dispatch (protoid= login.三种登录方式) online.offline = on =>

第一步 cache\_login.set\_c\_context => 更新context对象 + heartbeat\_ctrl.reset\_updatetime

第二步 dispatcher.\_dispatch\_client\_msg 根据-pid 查找redis.cache\_info直接返回给客户端，pid是正数则缓存redis.cache\_info

待做：没有缓存push包

-----------------------------------

排队认证模块

数据结构

auth\_logic fd\_map[fd]={challenge secret} 用于auth的几次交互临时变量

auth\_ctrl

wait\_num 队列当前剩余人数：认证成功wait\_num++, 定时器10s wait\_num = wait\_num - 1000 or 0 or 不排队(队列不拥堵/断线重连)

=> wait\_second 当前需要等待时间 = wait\_num/1000 \* 10 or 不排队

客户端 sleep(wait\_second) => login.signin\_account

redis格式数据

auth\_info playerId : {"waitnum":0,"login\_addr":17,"pid":1,"secret":"x \r ubM","subid":1,"waitsecond":0}

启动流程(排队流程部分)

auth\_ctrl.init => cal\_wait\_num 定时扣减排队人数

排队流程

auth\_logic.set\_auth\_info(wait\_num player\_id) => login\_addr+subid+waitnum+waitsecond 缓存redis.auth\_info 并推送给客户端

auth\_ctrl.login\_account/weixin\_login/visitor\_login 认证成功wait\_num++

认证流程

第一步 auth\_logic.auth\_secret (step=1,ckey) => 返回客户端challenge skey 同时缓存fd\_map[fd]={challenge secret}

第二步 auth\_logic.auth\_secret (step=2,chmac) => 返回ec=success/认证失败断开连接

第三步(账号用户认证) auth\_logic.login\_account (account,password) password通过secret解密校验 => set\_auth\_info

第三步(账号用户注册) auth\_logic.register\_account => 注册... => auth\_logic.login\_account

第三步(微信用户认证) auth\_logic.weixin\_login => set\_auth\_info

第三步(游客认证) auth\_logic.visitor\_login => set\_auth\_info

待解决：

1 login\_addr和subid设置

-----------------------------------

locator模块

对象关系(重要!!)

game\_id 一个game可以启动多个server来实现，但是一个server只运行一个game。

一个game对应多个room\_id，一个room\_id只对应一个game\_id。

server\_id 一个server上可以启动多个room\_id，一个room\_id可以在多个server上启动。

room\_id

redis格式数据

fdnum\_room:server\_id@room\_id: fdnum : room\_addr 记录所有server\_id上各个room\_id的各个room的fdnum

fdnum\_server:game\_id fdnum : server\_id 记录所有game\_id下各个server\_id的fdnum

readystart\_server:game\_id {server\_id} 记录未启动的server\_id

readystop\_server: {server\_id} 记录因为人数扣为0，等待被关闭的server\_id

配置 locator\_server\_config game\_room\_config(game\_id => module\_name)

启动流程 locator\_ctrl.init =>

locator\_ctrl.set\_unstart\_server\_map 构造未启动server列表 添加到redis.readystart\_server

非locator启动的gameserver

area\_ctrl.init => locator\_ctrl.register\_start\_info(server\_id) => 移除对应server\_id的redis.readystart\_server

登录流程

locator\_ctrl.route\_sid(game\_id) =>

第一步 locator\_ctrl.check\_stop\_server(server\_id) => locator\_ctrl.stop\_server(server\_id) 停止一台game\_id的fd为0的server,

移除对应server\_id的redis.readystop\_server 添加到redis.readystart\_server

第二步 locator\_ctrl.get\_server\_id =>

第一步 room.get\_max\_server => server\_id+fd\_num 获取fd数最多的server\_id

第二步 若server\_id的fd满了，=> register\_start\_server(game\_id) 获取待启动的server\_id

更新redis.fdnum\_server、redis.readystart\_server 并启动server\_id

相关登录流程

enter\_area => 更新redis.fdnum\_server fdnum +1

exit\_area => 更新redis.fdnum\_server fdnum -1 若fdnum=0 且 game\_id的server已启动数>1 => 添加到redis.readystop\_server

enter\_room/exit\_room => 更新redis.fdnum\_room fdnum +/- 1

旧流程 已废弃【

启动流程：locator\_ctrl.init =>

a locator\_ctrl.set\_game\_list(构造unstart\_server)

b locator\_ctrl.gameserver\_manager(game\_id+room\_type) => db.room.get\_room\_list(预留一台人数未满服务器，多则stop\_game.sh，少则run\_xpnn.sh)

登录流程：client\_msg.dispatch => locator\_ctrl.route\_sid => locator\_ctrl.gameserver\_manager返回人数最少的server\_id =>

返回server\_name给client\_msg.dispatch

监控创建、销毁：

1 roomid => 所有serverid和对应playernum

2 如果有playernum=0保留1台，多余销毁。如果都不是0则添加一台。

玩家进入game：分配playernum数量最少那台，如果playernum满需要通知locator监控动态创建。

玩家进入：locator.route\_sid => 获取servername返回

】

问题：

1 pkill 关闭 room，redis没有删除导致locator默认已经存在room

2 考虑min\_enter金币因素

-----------------------------------

登录模块

redis格式数据

plyaer\_info:3 {head\_url:xx sex=1 nickname=xixi create\_time=xxxxxx player\_id=3 head\_id=1 gold=100}

player\_online:3 {session:xxx state=1 player\_id=3 fd=19 watchdog=12 ip=127.0.0.1 gate=gate1 agentnode=xpnn agentaddr=15}

chat模块fd相关

login.login\_ctrl

logout\_account

signin\_account

weixin\_account

visitor\_login

启动流程

loginserver/main => 创建login => login\_ctrl.init => 创建多个login\_logic\_svc => login\_logic.init(空操作)

hallserver/main => 创建hall => hall\_ctrl.init => 创建多个hall\_logic\_sv 创建多个agent(预创建) => agent\_ctrl.init

登录流程

gate.handler.connect => watchdog.socket.data => socket\_msg.data => client\_msg.dispatch => service\_base.dispatch\_client\_msg =>

dispatcher.dispatch\_client\_msg => login\_impl => login\_ctrl.signin\_account(login、weixin、vistor) =>

第一步，login\_logic\_svc => login\_logic => db 验证账号密码

第二步, login\_ctrl.cast\_login => hall.cast\_login => hall\_ctrl => hall\_logic\_svc => hall\_logic.cast\_login =>

db(设置、更新redis.plyaer\_online redis.fd相关 基础数据)+ hall.agent.login => 登录大厅

第一步，agent\_ctrl.on\_login => watchdog.login\_ok => gate\_msg 设置socket\_msg.c对象(player\_id agentnode)

第二步，player\_ctrl.on\_login => db.player.plyaer\_info(mysql=>redis不存在则添加到redis)

第三步, chat.set\_chat\_ctx(ctx) 设置聊天模块支持

=> dispatcher.response\_client\_msg

重复登录剔除1号流程

2号窗口登录流程至hall\_logic.cast\_login => gate\_msg.player\_leave(参考网关模块)

登出流程 (参考网关模块)

1 重复登录、已经登录过大厅情况 进入、退出hall redis数据更新 playerOnline

1.1 重复登录playerOnline.state=1时：session相同,登入大厅逻辑不处理；session不同，剔除旧连接。

1.2 进入过房间playerOnline存在：

若agentnode(gamenode)版本号没更新，执行hall.agent\_ctrl.login更新redis.playerOnline的roomId；

若版本更新，则删除redis.playerOnline信息，等enter\_room请求时重新设置上诉redis信息。

待解决：

agentnode gatenode没有用到

cluster\_monitor.callback

-----------------------------------

game模块

xpnn\_const.GAME\_STATE = {

ready\_begin = 0, 创建队伍时

qiang\_banker = 1, 开始游戏时 => 改标签名

bet = 2,

open\_card = 3,

game\_end = 4,

gaming = 5, 开始游戏时

}

SEAT\_STATE = {

null = 0x00,

unready = 0x01,

ready = 0x02, 组队时

gaming = 0x04, 游戏开始时

offline = 0x08,

exit = 0x10,

game\_end = 0x12, 游戏结束时

}

配置 room\_id => game\_room\_config.game\_id => game\_config.game\_type => game\_type\_config

redis格式数据

player\_online:3 {session:xxx state=1 room\_id= 10101 player\_id=3 fd=19 watchdog=12 ip=127.0.0.1 gate=gate1 agentnode=xpnn agentaddr=15}

两种消息：

1 cd事件 需要注册cdtype才能接收到开始、结束通知

2 广播/单发消息 队伍中都能接收

cd\_ctrl

register\_callback 添加cdtype的callback

register\_listener 添加监听cdtype的player\_id

add\_cd 添加cdtype的一个事件(id标识,入参seconds,args)并下发给监听cdtype的所有客户端 =>

定时结束触发 => handle\_cd\_result 执行cd\_type对应的callback，并把id,args传入 =>

执行结束cd\_ctrl.del\_cd

del\_cd 销毁事件id,并下发给监听cdtype的所有客户端

on\_login(ctx) 下发该用户监听的所有事件(更新结束事件)

启动流程

gameserver/xpnn/main => 创建room => room\_ctrl.init =>

第一步， init\_desk\_pool => 创建多个desk => desk.init(空操作)+xpnn.init => desk\_ctrl.register\_callback+cd\_ctrl.register\_callback注册回调，

当desk\_ctrl和cd\_ctrl接口需要，调用xpnn内部接口

第二步， init\_agent\_pool => 创建多个agent => agent.init =>

第三步， init\_room => redis 添加room

大厅服务hall (server = SERVER.HALL, service = SERVICE.HALL)

get\_room\_inst\_list(c2s) => redis

get\_player\_online\_state(c2s) => redis

area服务

启动: area\_ctrl.init => 创建svc服务+根据game\_id创建对应room服务

enter\_area 进入area. redis(player\_online添加agentnode agentver,state=area fd相关添加) socket\_msg添加agentnode、agentver、game\_id

exit\_area enter\_area添加对象的删除,state=online + get\_role/create\_role添加对象的清除

get\_role 根据game\_id+player\_id => role\_id，如果存在则添加redis.player\_online.role\_id

create\_role 创建role，添加redis.role\_info，redis.player\_online.role\_id

enter\_room 设置socket\_msg.room\_addr、redis.player\_online.room\_addr

房间服务room ( server = SERVER.GAME, service = SERVICE.ROOM) agent对象用于xpnn逻辑调用代理

enter\_room(c2s)

第一步 area\_ctrl.enter\_room(roomtype) => get\_room\_addr 选择或动态创建room => room\_addr room\_id

第二步 room\_ctrl.enter\_room

第一步 agent.login =>

第一步 agent\_ctrl.on\_login =>

第一步 gate\_msg.set\_agent =>设置socket\_msg.c对象(agentaddr地址)

第二步 更新redis.player\_online agentaddr

第二步 player\_ctrl.on\_login(空操作)

第二步 更新redis.player\_online room\_id state roomaddr fd相关 设置socket\_msg.c对象(roomaddr)

exit\_room(c2s) room\_ctrl.exit\_room =>

第一步， desk.logout\_desk(执行过group\_request) => desk\_ctrl => xpnn\_ctrl

第二步， agent.logout => callback.logout => kill agent

第三步， 更新db在线人数(更新redis.room\_inst.player\_num)

group\_request(c2s) room\_ctrl.group\_request => 获取group\_id+分配desk服务 =>

第一步， desk设置对应player\_id的agent、ctx(内部数据)

第二步， xpnn\_ctrl.login\_desk =>

第一步，desk\_ctrl.get\_player\_info => agent(代理调用) => player\_ctrl.get\_player\_info 设置xpnn\_ctrl对象

第二步，cd\_ctrl.register\_listener 注册监听所有cdtype

第三步，cd\_ctrl.on\_login 接收登录前存在的事件

第四步，广播消息seat\_state\_event

第五步，can\_game\_start人数满足 => 推送ready\_begin事件 => 触发on\_ready\_begin => 单发消息game\_start\_event

第三步， gate\_msg.login\_desk 设置socket\_msg.deskaddr(client\_msg.dispatch作为service对象发送请求)

logout\_desk(c2s) room\_ctrl.logout\_desk => desk.logout\_desk =>

第一步， desk清空对应player\_id的agent、ctx(desk内部数据)

第二步， xpnn\_ctrl.logout\_desk

第一步，cd\_ctrl.unregister\_listener 解除监听

第二步，若不在游戏状态,广播消息seat\_state\_event

第三步， gate\_msg.logout\_desk 清除socket\_msg.deskaddr

game\_end\_event(s2c) on\_game\_end => 单发game\_end\_event(某些人游戏结束)

player\_disconnect

player\_reconnect

desk服务

1 仅游戏待开始的desk才能被加入

2 游戏结束后一定时间，剔除所有玩家room\_logic.logout\_desk + 广播消息exit\_desk\_event + 更新desk为未开始状态

问题

agentver版本更新研究

-----------------------------------

chat模块

数据结构

fd\_ctx[ctx.fd]= {fd = ctx.fd, watchdog = ctx.watchdog, gate = ctx.gate}

redis格式数据

fd\_hall {fd} 进入大厅

fd\_game:game\_id {fd} 进入area

fd\_server:server\_id {fd} 进入area

fd\_roomtype:room\_id {fd} 进入room

fd\_roomaddr:server\_id@room\_addr {fd} 进入room

登录流程

hall\_logic.login(ctx) => chat.chat\_ctrl.set\_chat\_ctx 设置fd相关，用于推送数据 + 更新redis.fd\_hall

area.enter\_area/exit\_area 更新redis.fd\_game + redis.fd\_server

room.enter\_room/exit\_room 更新redis.fd\_roomtype + redis.fd\_roomaddr

聊天流程

chat\_ctrl.chat\_req(type context) => chat\_logic.chat\_req(type 聊天范围 context内容) =>

第一步 get\_fds 根据type(type类型：hall game server room\_type room\_addr)读取redis.fd相关 => fds

第二步 send\_data => fds+ctx+reply => context.send\_client\_event 推送给所有client

-----------------------------------