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            xxxxxxxxxx
            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,"servert
ype":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
---