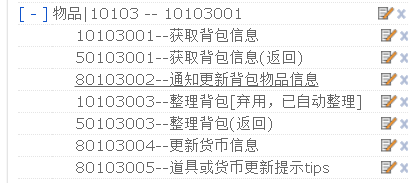
# 客户端通信的消息说明

## 消息号说明

消息号页面：<http://192.168.18.43:4024/develop/>

消息号规则：

1. 高5位是功能号，最低3位是功能下的每个接口顺序编号，如下图



1. 功能号最高位1表示客户端发过来的请求；最高位5表示对应请求的服务端应答消息；
2. 功能号最高位8表示服务端的主动推送消息
3. 功能号从10100开始按顺序递增下来，如下图



1. Struct是所有消息共用的公共消息结构

## 消息页面使用说明

### 添加Struct公共消息结构

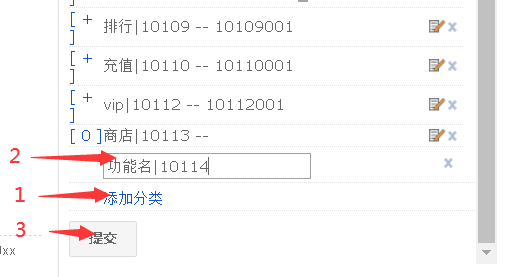


提交后，点击“生成并提交消息号”即可生成对应的.proto后缀的消息文件

如：ProtoPublic.proto

### 添加功能消息

#### 添加功能号



第二步要注意：格式固定为：功能名 加上 “|” 加上 功能号，

如：功能名|10114

#### 添加消息号



提交后，点击“生成并提交消息号”即可生成对应功能号的.proto后缀的消息文件

如：ProtoClient\_10112.proto

# 用IntelliJ IDEA维护项目代码(GIT)

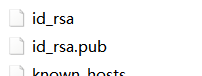
1. 把检出代码的ssh密钥id\_rsa和公钥id\_rsa.pub放在windows用户路径.ssh上：

注意：密钥文件名必须是id\_rsa

如用户名是Administrator

则.ssh路径是：

C:\Users\Administrator\.ssh



2. IDEA内存优化(秒开的快感!!)

因机器本身的配置而配置：

\IntelliJ IDEA8\bin\idea.exe.vmoptions

// (根据你的配置变大!!)

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

-Xms2048m

-Xmx2048m

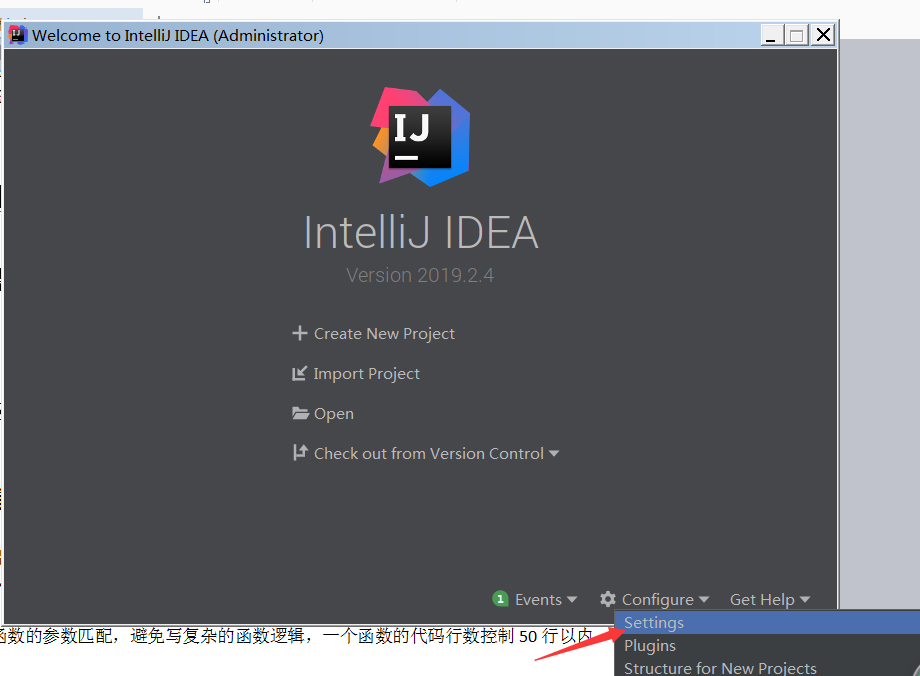
-XX:MaxPermSize=512m

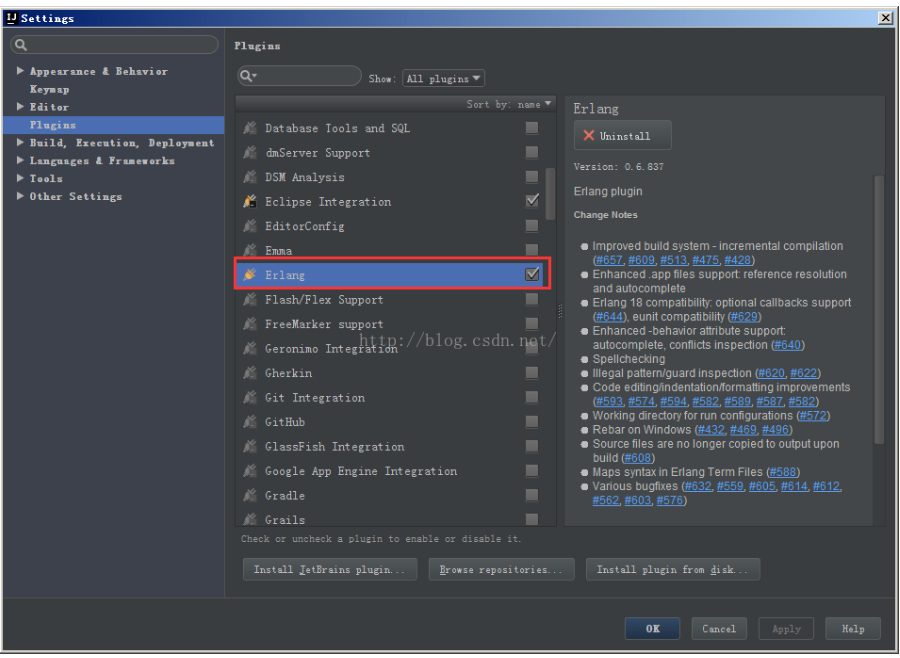
-ea

-server

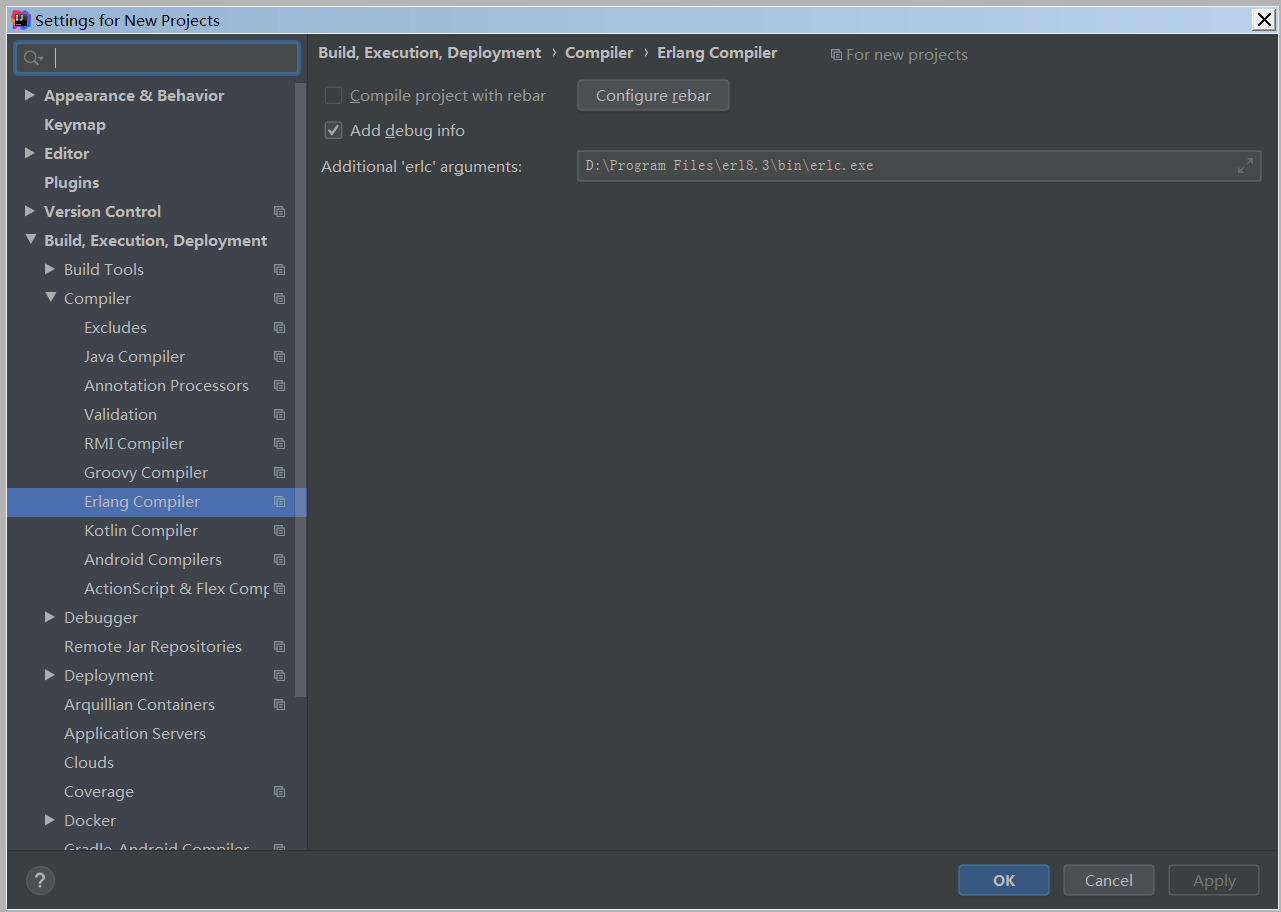
-Dsun.awt.keepWorkingSetOnMinimize=true

## 安装并设置erlang插件

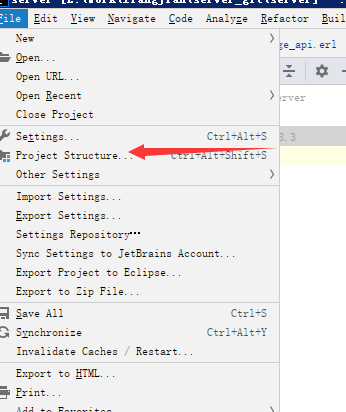


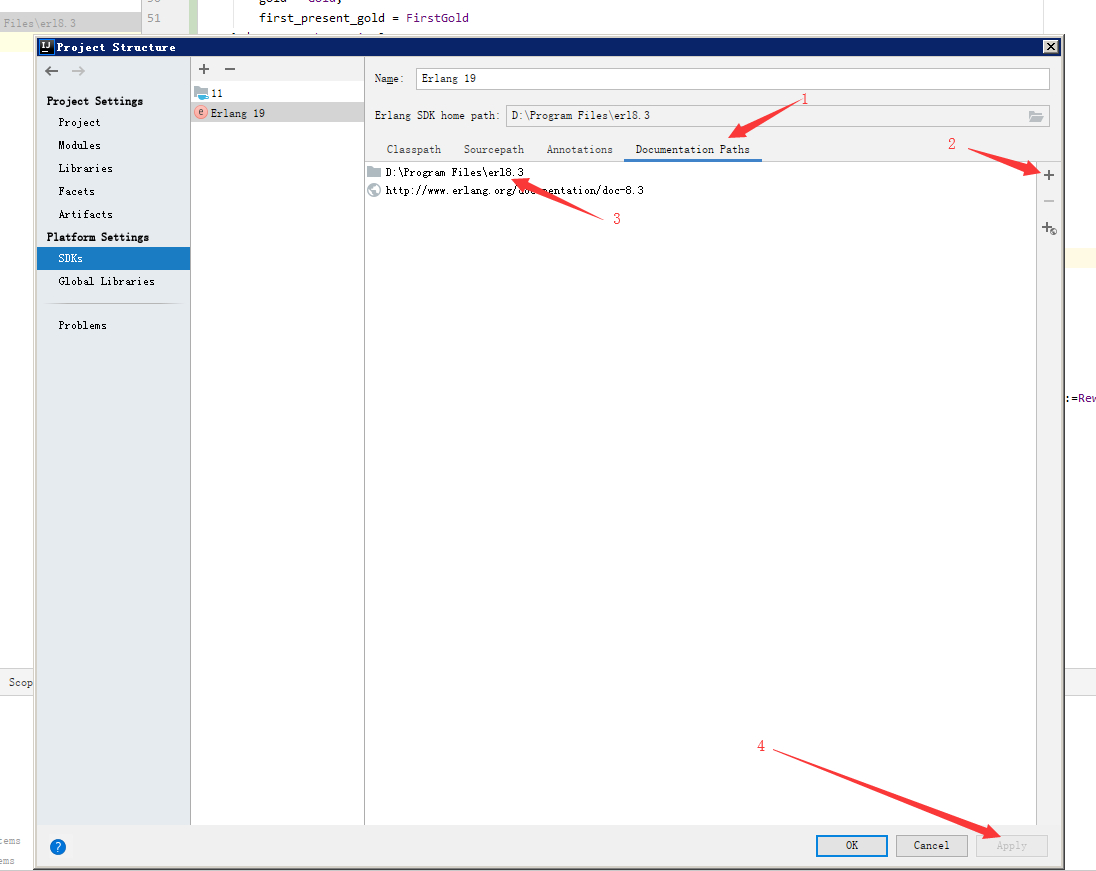


设置erlc程序所在的路径

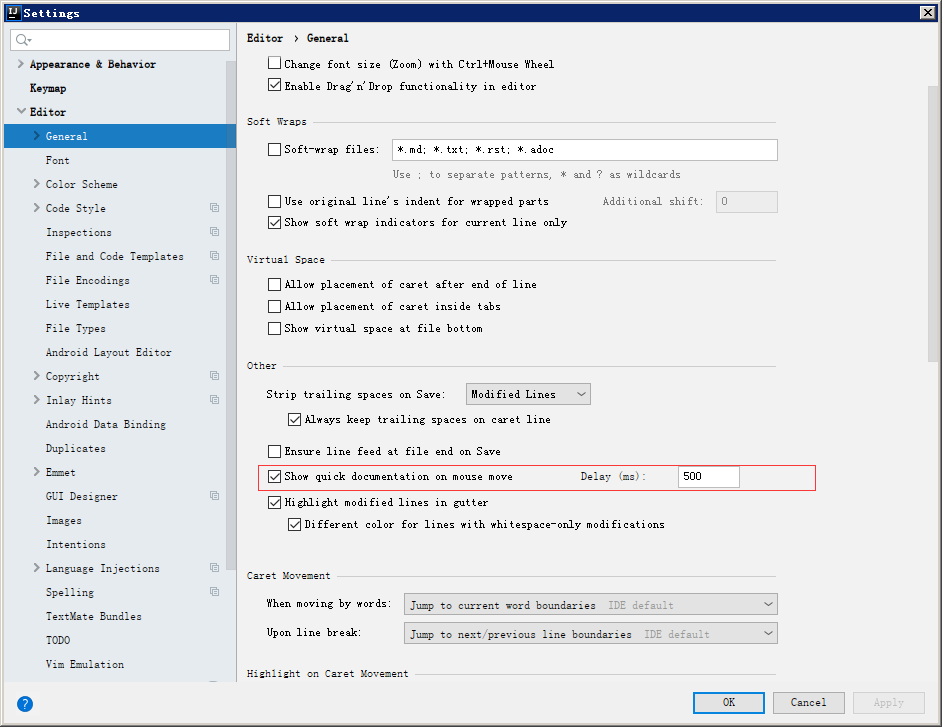


设置erlang api函数的文档提示：

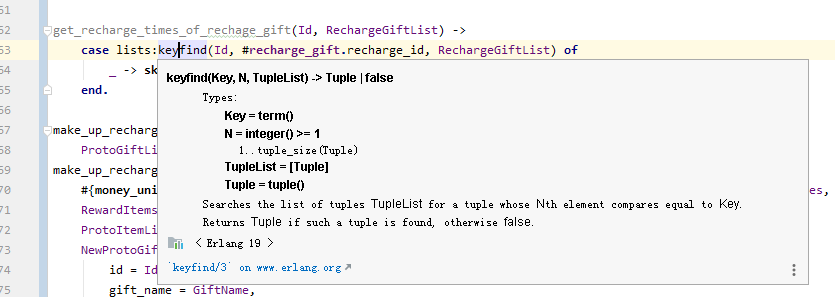




设置鼠标放在上面时延迟多少秒提示注释内容：

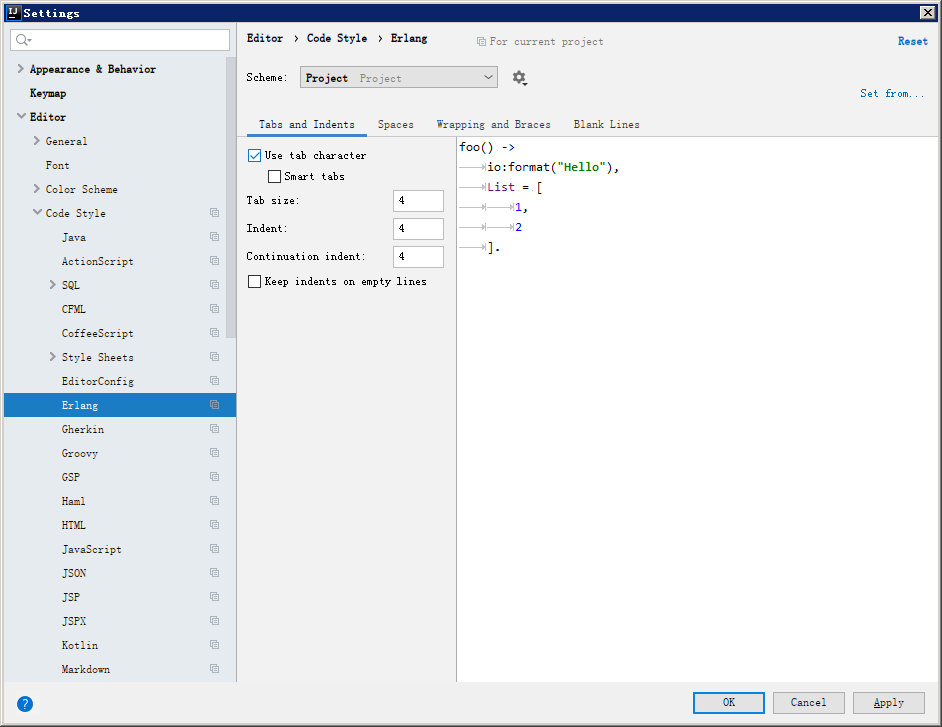


文档提示设置好的效果如下：



### 设置代码缩进风格

File -> Settings -> Editor -> Erlang

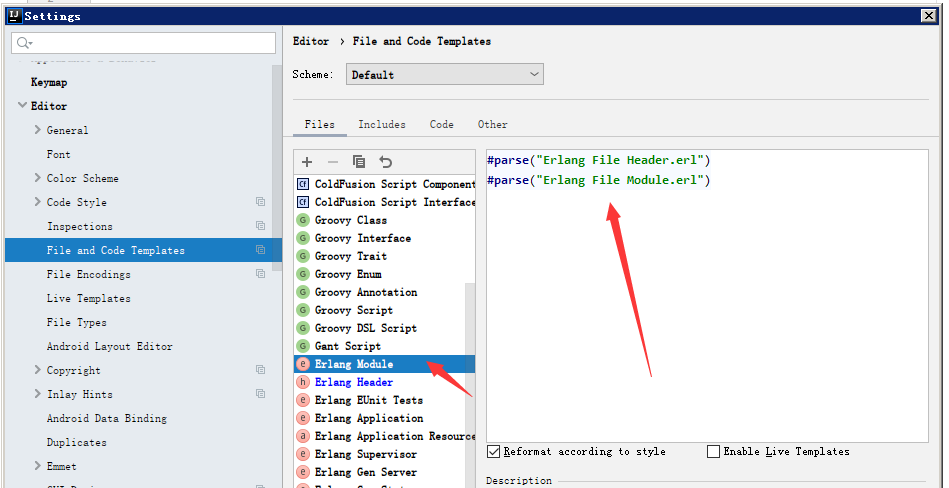


### 设置erl和hrl的文件头格式

File -> Settings -> Editor -> File and Code Template

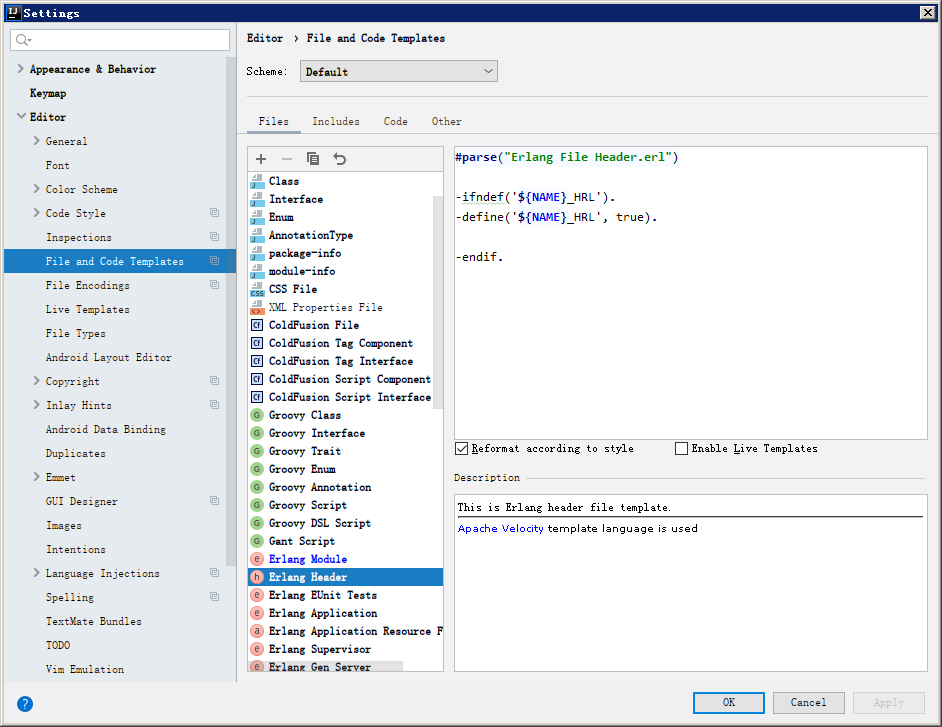
erl模块文件的文件头格式(Erlang Module)

**#parse**(**"Erlang File Header.erl"**)  
**#parse**(**"Erlang File Module.erl"**)



hrl头文件的文件头格式(Erlang Header)

**#parse**(**"Erlang File Header.erl"**)  
  
-ifndef('**${NAME}**\_HRL').  
-define('**${NAME}**\_HRL', true).  
   
-endif.



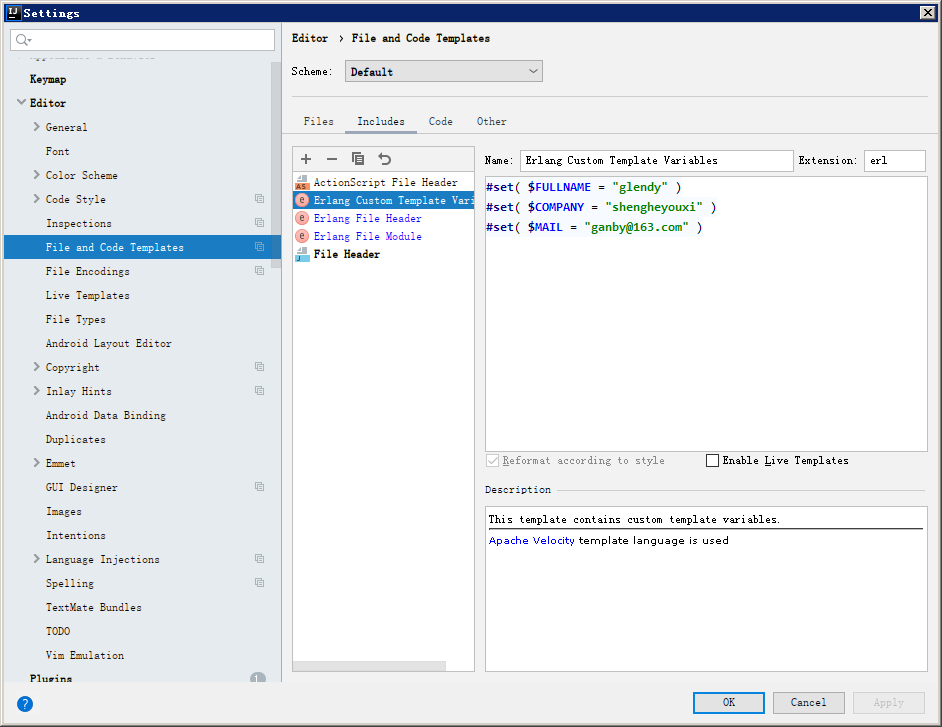
自定义个人信息变量(Erlang Custom Template Variables)：

FULLNAME 个人全称

COMPANY 公司名称

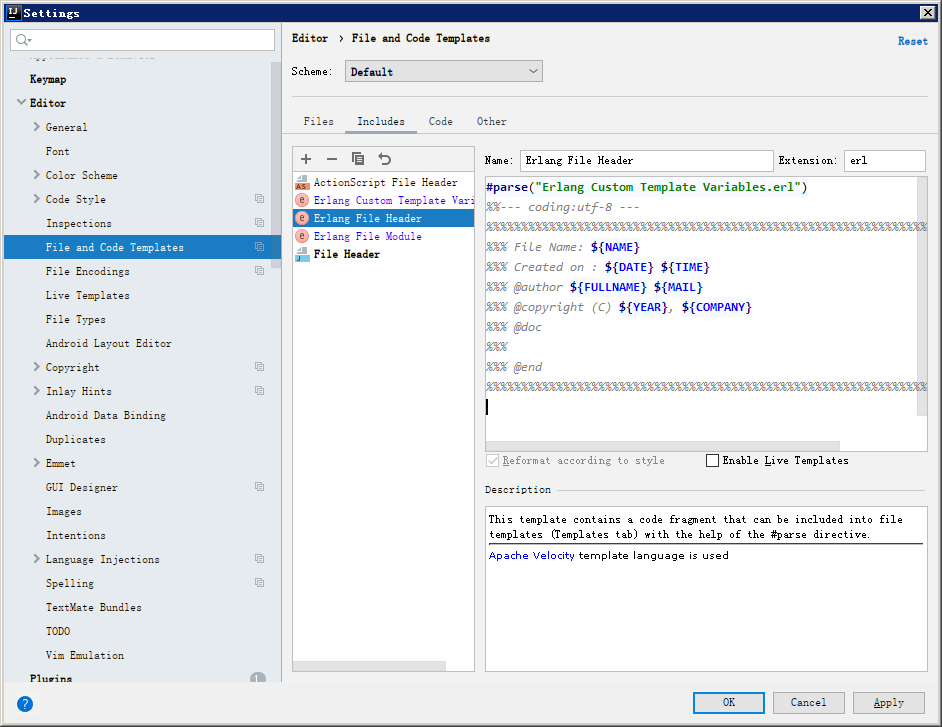
MAIL 个人邮箱

**#set**( **$FULLNAME** = **"名字拼音"** )  
**#set**( **$COMPANY** = **"shengheyouxi"** )  
**#set**( **$MAIL** = **"个人邮箱"** )



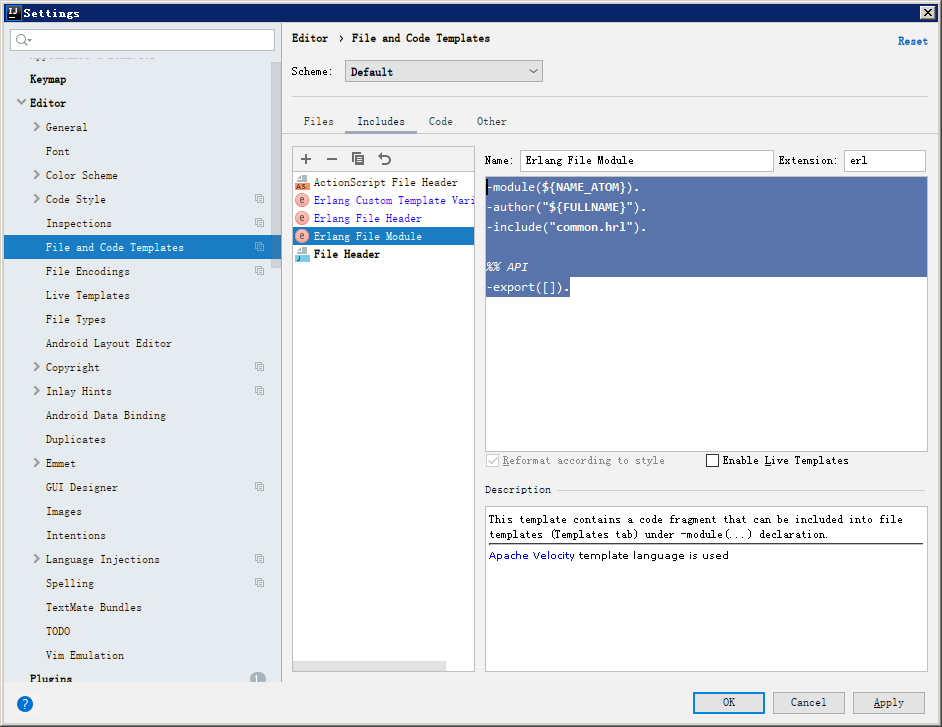
文件头格式变量定义(Erlang File Header)

**#parse**(**"Erlang Custom Template Variables.erl"**)  
%%--- coding:utf-8 ---  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% File Name: **${NAME}**%%% Created on : **${DATE} ${TIME}**%%% @author **${FULLNAME} ${MAIL}**%%% @copyright (C) **${YEAR}**, **${COMPANY}**%%% @doc  
%%%  
%%% @end  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



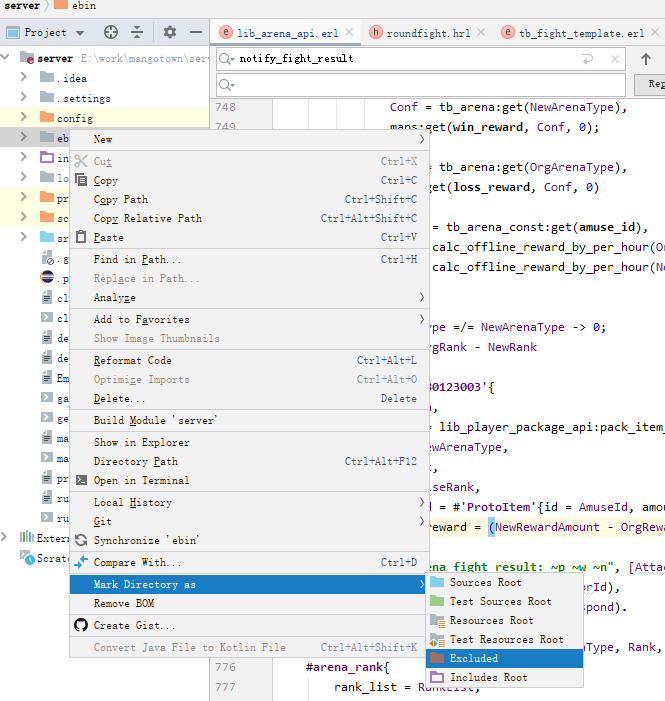
Erl文件的模块预定义(Erlang File Module)

-module(**${NAME\_ATOM}**).  
-author("**${FULLNAME}**").  
-include("common.hrl").  
  
%% API  
-export([]).



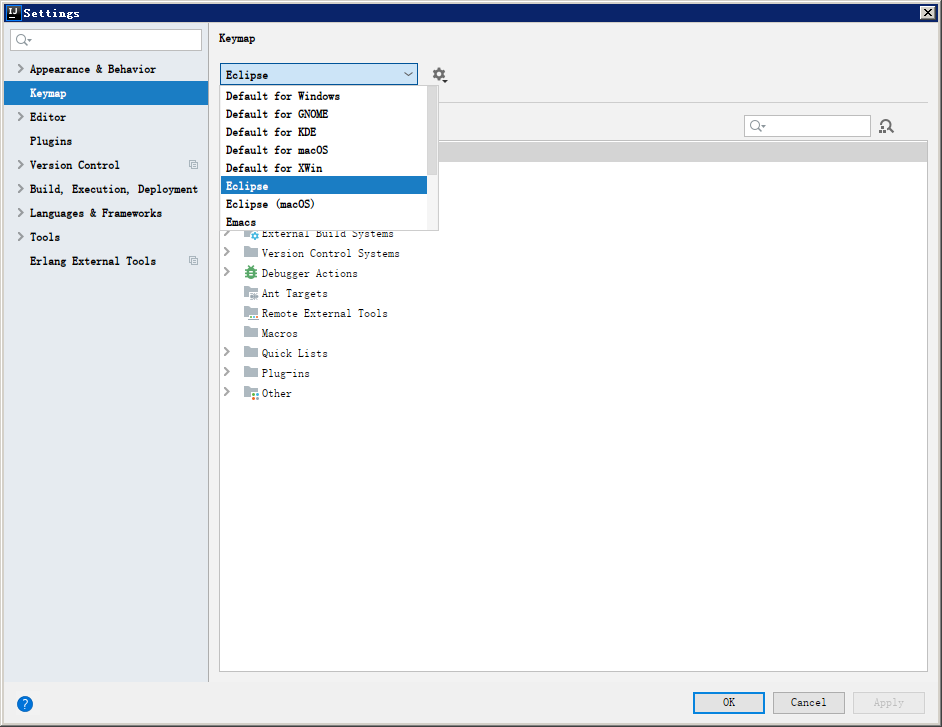
### 把ebin等非代码目录排除，加快索引建立

如下排除ebin目录的设置，其他config, proto, scripts等目录同样设置



### 设置使用eclipse的快捷键方案

File -> Settings -> KeyMap -> 选择Eclipse



Alt + / 代码提示

Ctrl + ← 代码跳转后退

Ctrl + → 代码跳转前进

Ctrl + Shift + R 文件快速查找定位

Ctrl + H 全工程查找字符串

Ctrl + G 查找函数引用的地方

Alt + Shift + R 全文重命名变量

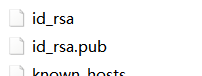
## 检出项目代码

1. 把检出代码的ssh密钥id\_rsa和公钥id\_rsa.pub放在windows用户路径.ssh上：  
   注意：密钥文件名必须是id\_rsa

如用户名是Administrator

则.ssh路径是：

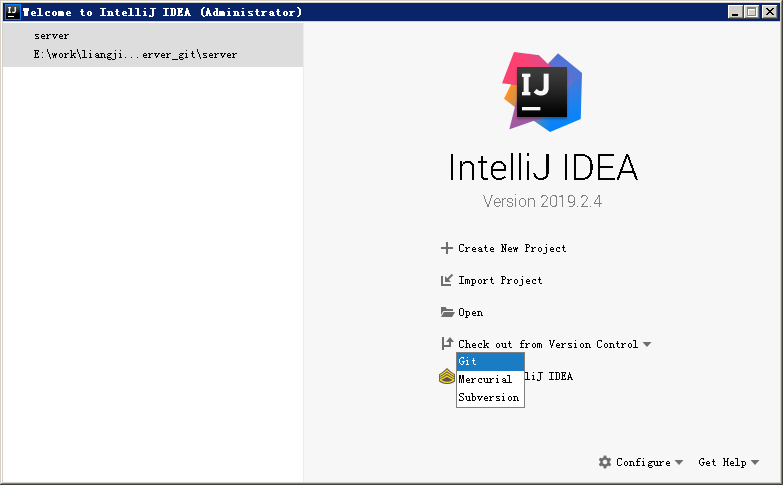
C:\Users\Administrator\.ssh

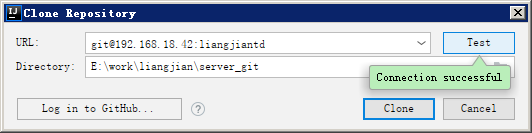


1. 用git命令检出代码：

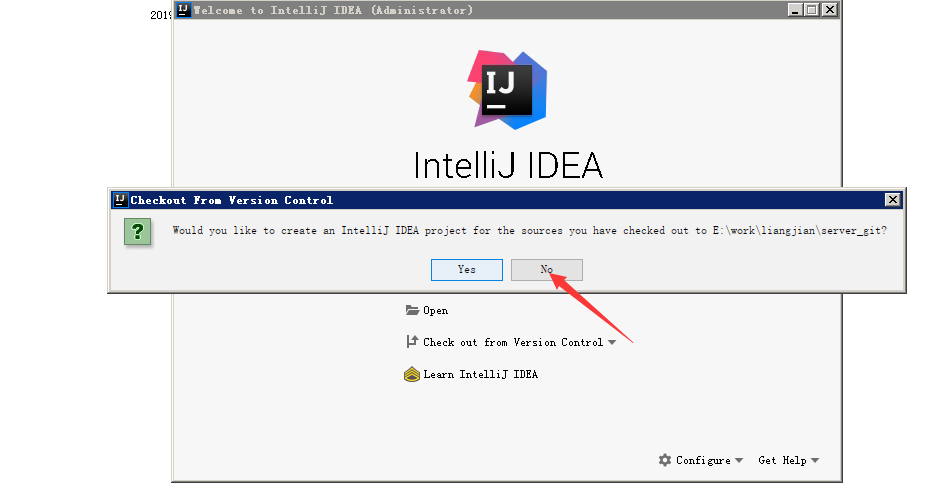
git clone [git@192.168.18.42:liangjiantd](mailto:git@192.168.18.42:liangjiantd)

Import工程:

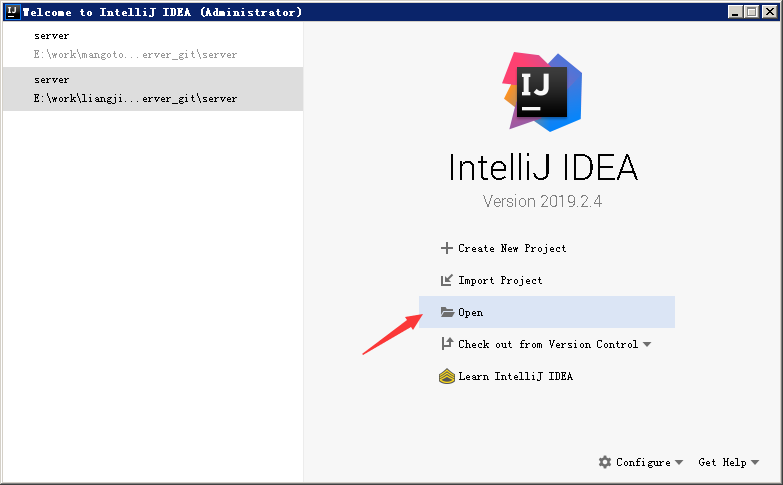


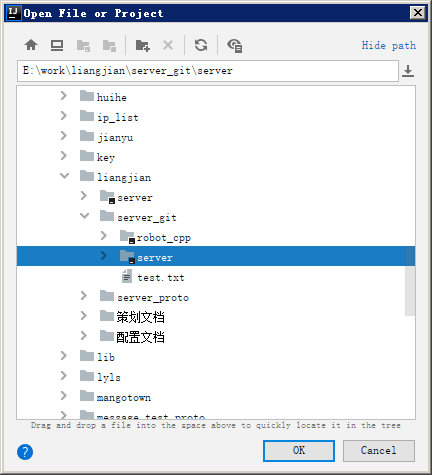


点击“Clone”导出项目代码：

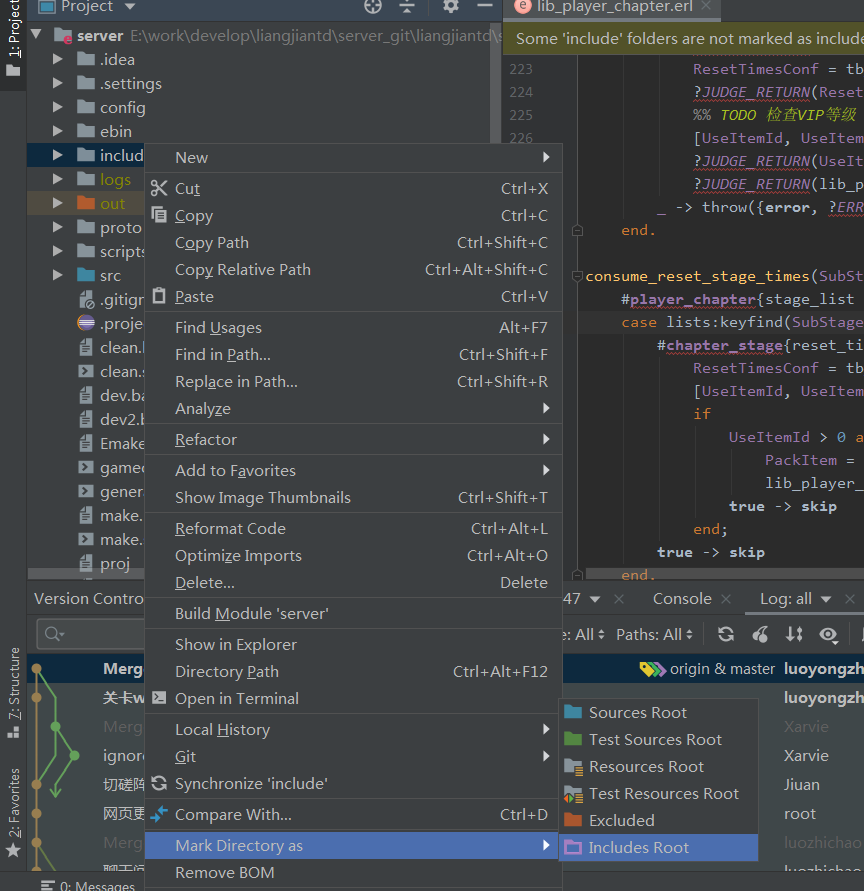


导出后弹窗提示点“No”,不直接创建工程





把include目录标记为include路径：



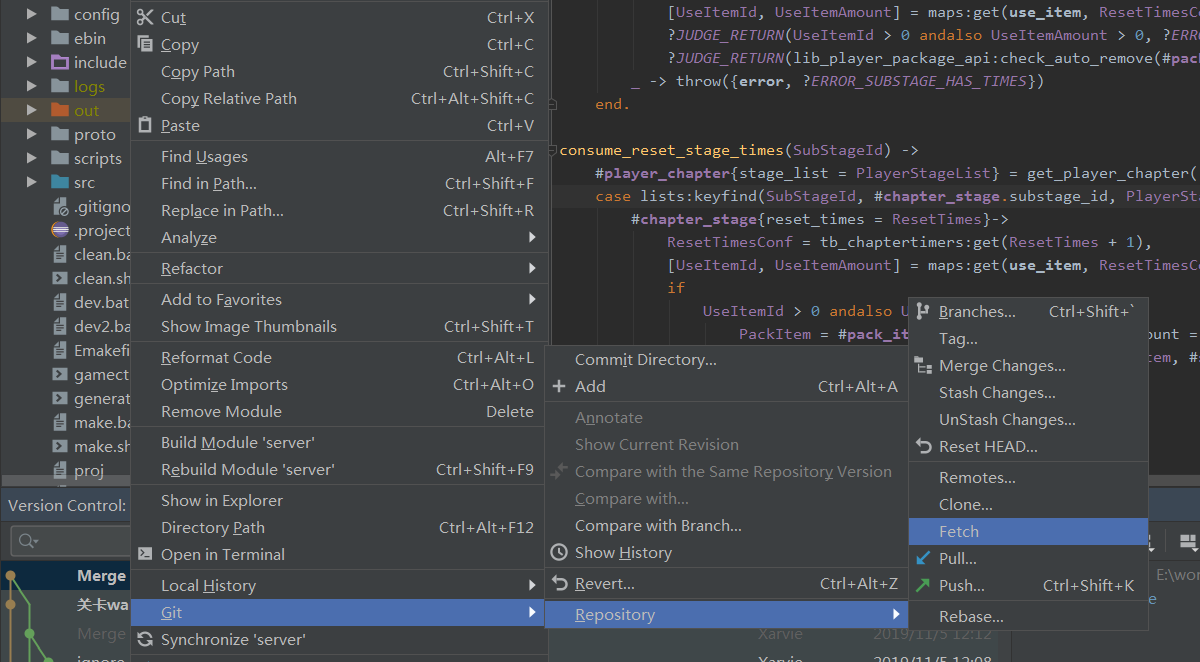
## 项目代码更新

一般步骤：

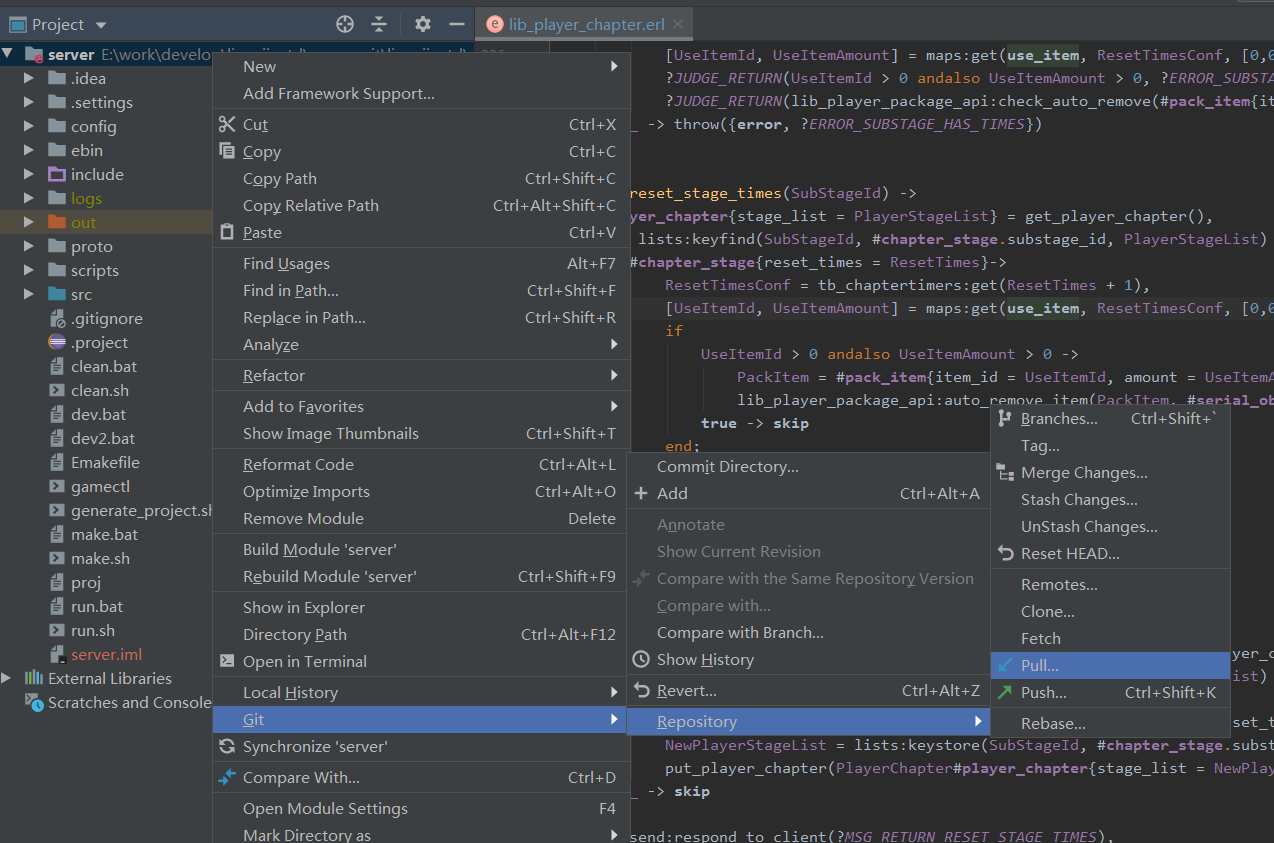
先更新 fetch"远程服务器 --> 本地服务器"，

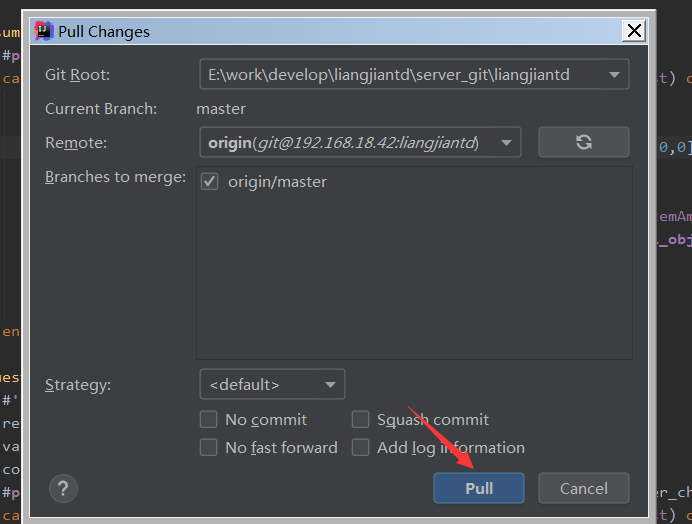
再进行 pull更新 " 本地服务器 --> 本地代码"

右键工程，依次选择Git -> Repository -> Fetch



右键工程，依次选择Git -> Repository -> Pull：





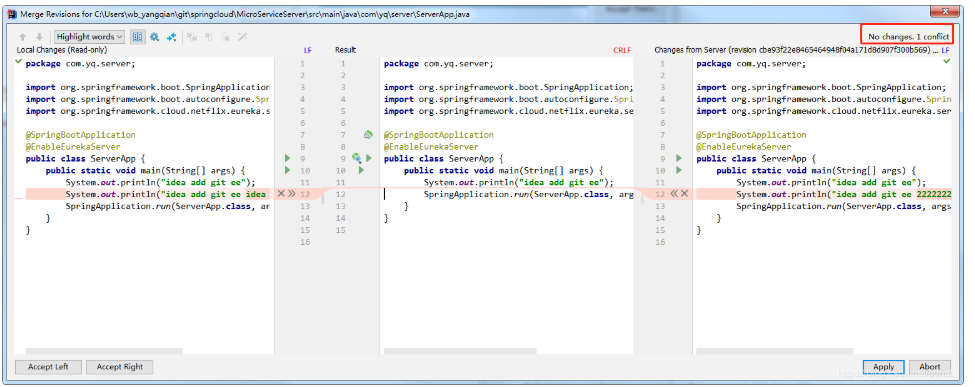
## 解决代码冲突

先从远程仓库更新项目代码，再把本地代码commit提交到本地仓库，更新 fetch"远程服务器 --> 本地服务器"，

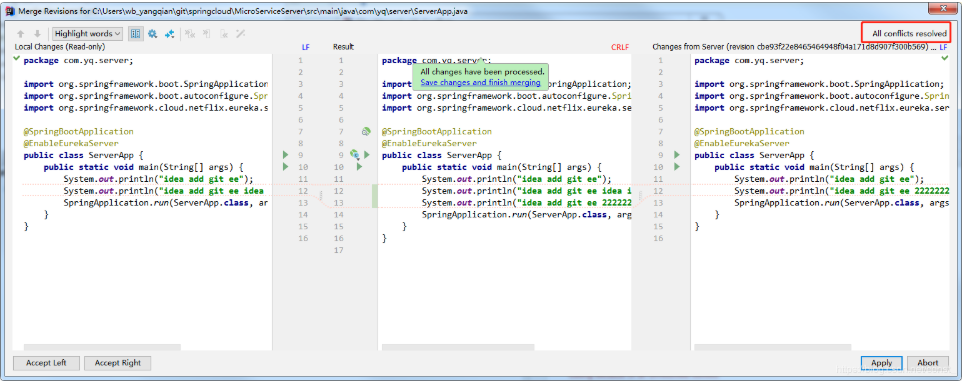
进行 pull更新 " 本地服务器 --> 本地代码"

pull时会弹窗提示产生冲突，此时选择“merge”来合并代码：

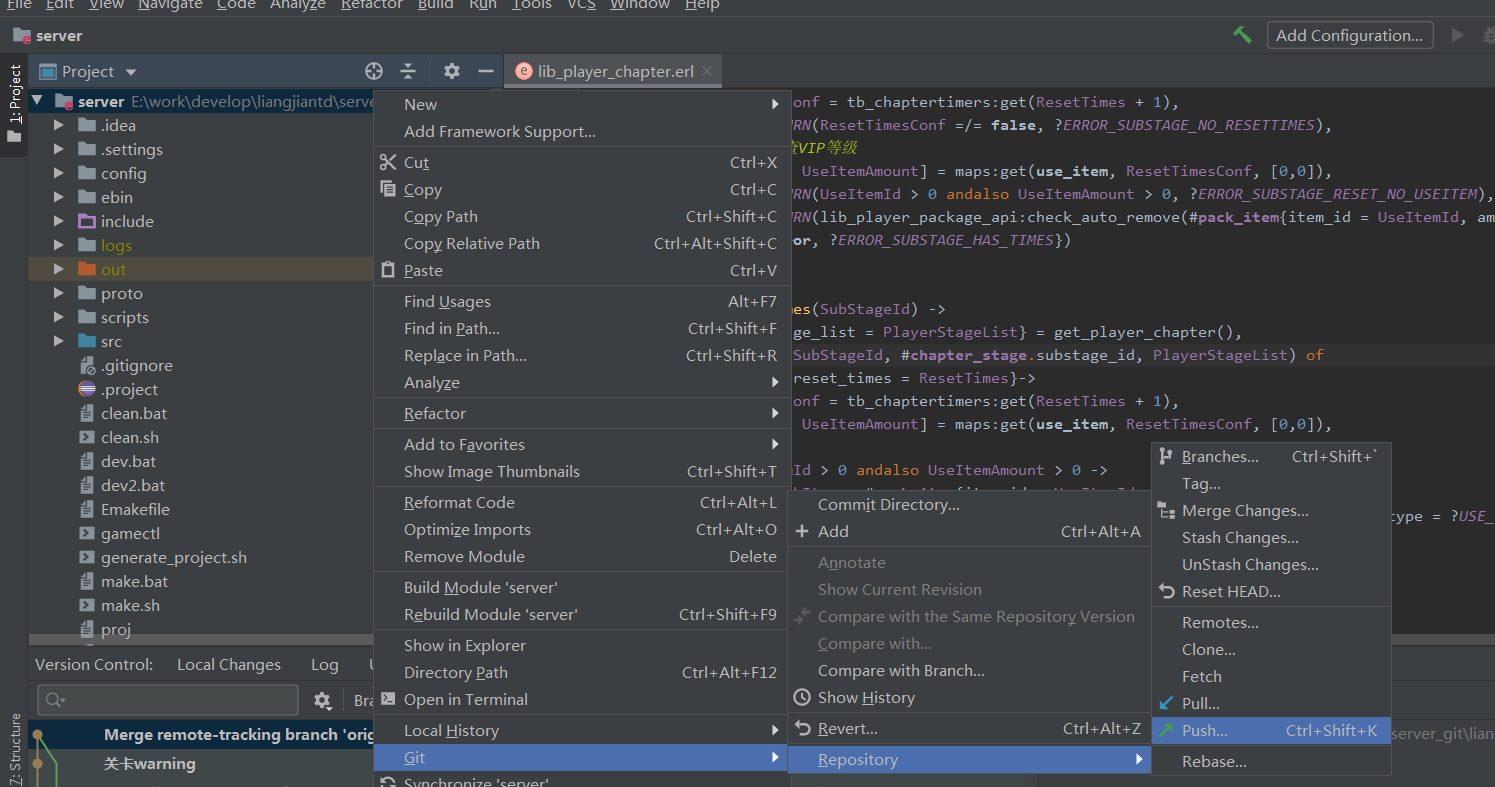
合并代码：左边窗口是自己的本地代码，右边窗口的服务器最新的代码



合并完成后会有绿色窗口提示全部冲突已解决（没有提示的仔细检查有遗漏的地方），点击Apply按钮，标记冲突解决：



Push提交代码到远程仓库：



## 提交代码到服务器

其实小伙伴就记住以上四个步骤，严格按照这四步顺序来执行，失败了不能继续往下一步走：

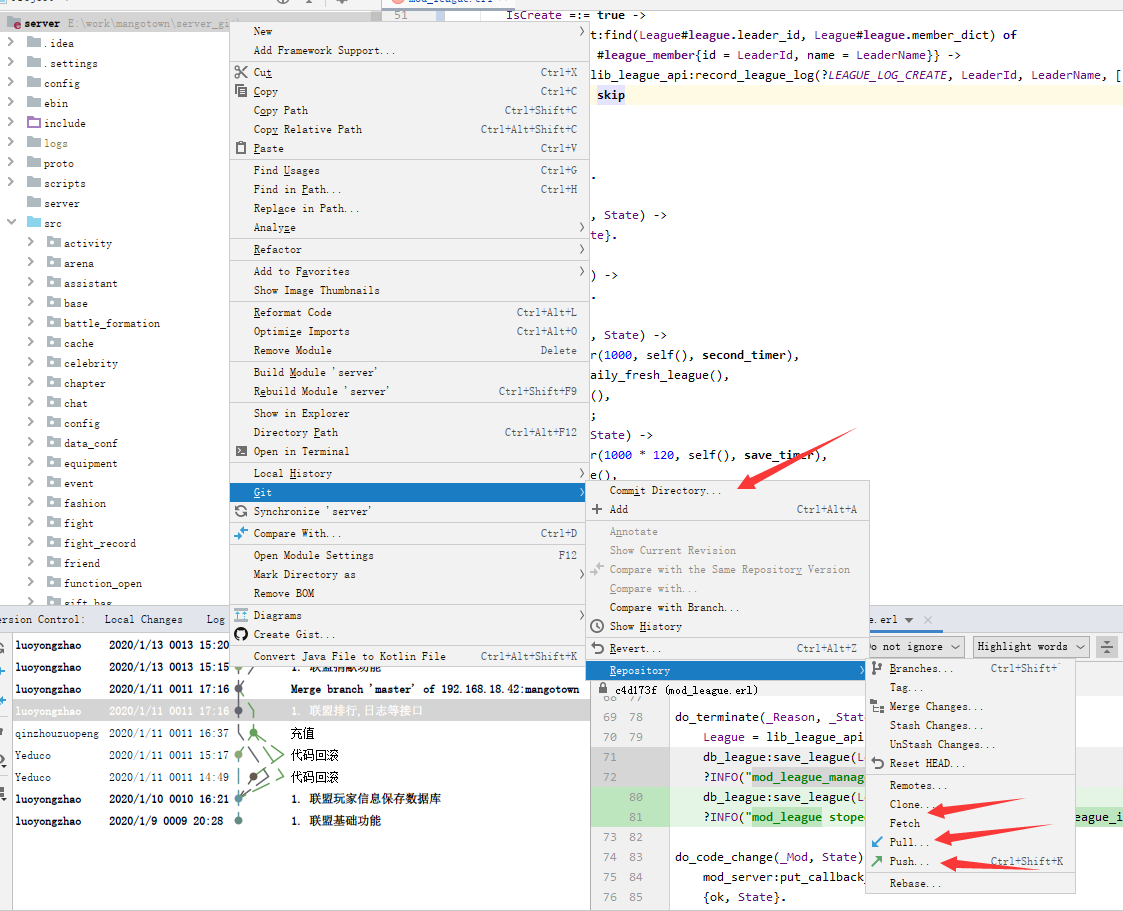
1. Commit : " 本地代码 --> 本地服务器"

2. 更新 fetch"远程服务器 --> 本地服务器"，

3. 进行 pull更新 " 本地服务器 --> 本地代码"

4. Push : "本地服务器 --> 远程服务器"

快捷键 ：Ctrl + K



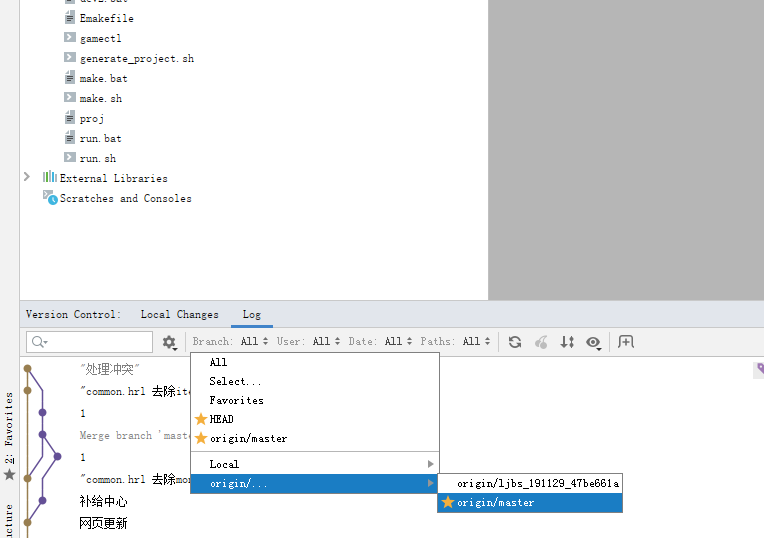
## 合并不同分支代码

示例，把主版本某些提交合并到版署分支：

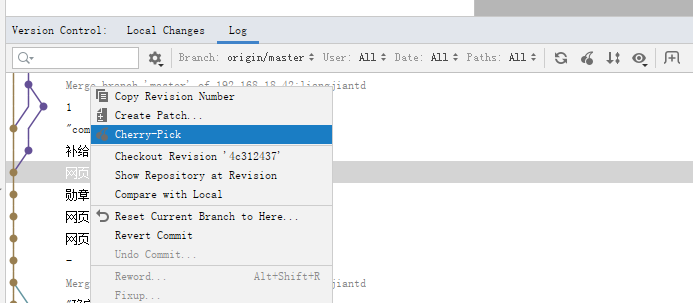
先检出版署分支代码：

git clone -b ljbs\_191129\_47be661a [git@192.168.18.42:liangjiantd](mailto:git@192.168.18.42:liangjiantd)

打开工程后，选择：log -> origin -> origin/master (主干版本)



选择要合并的提交日志，选择”Cherry-Pick”：



解决完冲突后，选择”commit”提交, 然后编译通过测试没问题后，push到版署远程分支上；

# 游戏架构说明

## 重点：规范约束

### 性能热点

1. 存储性能由快到慢如下：

进程状态 > 进程字典 > ets > 数据库存取

1. 尽量用函数的参数匹配，避免写复杂的函数逻辑，一个函数的代码行数控制50行以内
2. 涉及到多人交互的功能启动一个mod服务进程来处理相关的逻辑，使用异步消息通信cast，避免使用同步通信call，避免使用rpc操作;
3. 即多进程通信应该使用async\_apply或async\_status异步通信
4. 在dict与maps之间，maps的处理性能更好更快

### 流水表字段规范

每张流水必须包含字段：

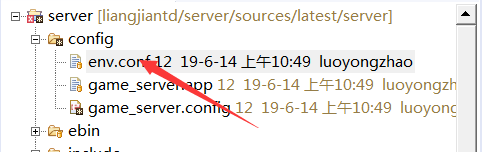
time 流水记录创建时间戳, 整数类型

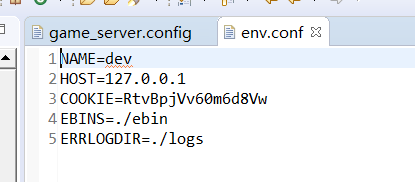
agent 渠道编号，整数类型

market 包号，整数类型

server\_flag 玩家创号时的服务器标识, #player\_base.server\_flag

### 内网运行时读取的启动节点名配置





NAME正式时填单服的目录名

HOSTS正式时填单服的内网IP

COOKIE跨服和单服需要相连的，必须是相同的cookie

### 数据包格式

#### 客户端请求包格式

32位长度 + 32位Crc校验码 + 32位自增序号 + 32位消息号 + 32位整数预留 + 二进制序列的消息体

其中，长度 = （32位Crc校验码 + 32位自增序号 + 32位消息号 + 32位整数预留 + 二进制序列的消息体）字节数之和

#### 返回包格式

32位长度 + 32位整数预留+ 32位消息号 + 32位错误码 + 二进制序列的消息体

其中，长度 = （32位整数预留+ 32位消息号 + 32位错误码 + 二进制序列的消息体）字节数之和

## 进程分布



说明：

mod\_accept是负责收发玩家socket数据的进程

mod\_unpack是负责把收到的二进制包分包并解码的进程

mod\_player是玩家进程，每个玩家一个单独的玩家进程

mod\_chat是负责全服玩家遍历发消息用的进程，管理聊天的频道子进程

mod\_channel是聊天频道进程，每种频道单独一个进程，联盟频道按联盟分为单独的频道进程

mod\_backactivity是后台运营活动管理进程，负责读取后台运营活动配置，通知玩家

mod\_player\_manage是玩家缓存管理进程，以及在线玩家管理广播进程

mod\_cluster是跨服节点间连接维护以及节点信息同步的进程

mod\_cluster\_backgroup是后台跨服分组配置信息读取进程

mod\_cluster\_gamegroup是每个玩法的跨服分组信息转发进程

## 功能模块

### lib\_counter——全服全世界唯一ID分配模块

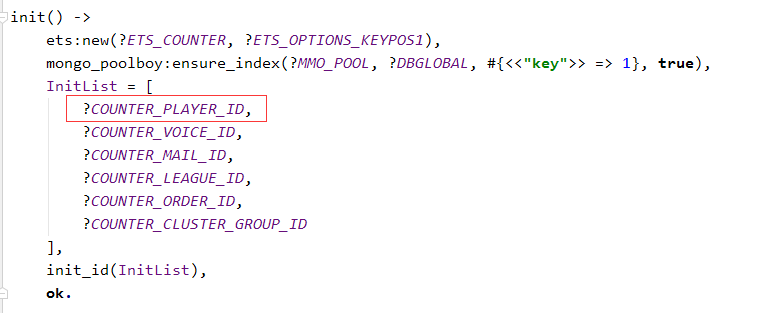
#### 新增新的ID接口

以player\_id为例：

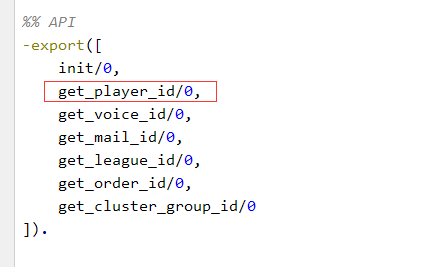
1. 定义宏名

-define(COUNTER\_PLAYER\_ID, counter\_player\_id).

1. 添加到初始化函数lib\_counter:init()里



1. 添加自增唯一id的获取接口



get\_player\_id() ->

get\_global\_id(?COUNTER\_PLAYER\_ID).

### 背包模块

#### lib\_player\_pack——玩家其他功能使用背包的接口

##### 使用缓存减少背包操作的接口使用示例（不涉及异步的建议使用此方法）

%% 生成一个缓存key

lib\_player\_pack::generate\_pack\_cache\_key(),

%% 判断是否有足够数量扣除道具

Ret1 = lib\_player\_pack::validate\_remove\_item(PackItemList1, true),

?JUDGE\_RETURN(Ret1 =:= 0, Ret1),

%% 判断是否有足够空格插入道具

Ret2 = lib\_player\_pack::validate\_insert\_item(PackItemList2, true),

?JUDGE\_RETURN(Ret2 =:= 0, Ret2),

...

%% 判断是否有足够数量扣除道具或资源

Ret2K = lib\_player\_pack::validate\_remove\_item(PackItemListN, true),

?JUDGE\_RETURN(Ret2K =:= 0, Ret2K),

%% 判断是否有足够空格插入道具或资源

Ret2N = lib\_player\_pack::validate\_insert\_item(PackItemListN, true),

?JUDGE\_RETURN(Ret2N =:= 0, Ret2N),

%% 执行以上所有validate接口需要的插入和删除道具或资源操作，更新玩家数据

lib\_player\_pack::cache\_update\_package(SerialObj, ItemTips, MoneyTips)

##### 不使用缓存的接口使用示例

%% 判断是否有足够数量扣除道具

Ret1 = validate\_remove\_item(PackItemList1, false),

?JUDGE\_RETURN(Ret1 =:= 0, Ret1),

%% 判断是否有足够空格插入道具

Ret2 = validate\_insert\_item(PackItemList2, false),

?JUDGE\_RETURN(Ret2 =:= 0, Ret2),

%% 实际删除道具或资源的操作，更新玩家数据

auto\_remove\_item(PackItemList1, SerialObj, ItemTips, MoneyTips),

%% 实际插入道具或资源的操作，更新玩家数据

auto\_insert\_item(PackItemList2, SerialObj, ItemTips, MoneyTips)

#### lib\_pack\_api——背包道具操作接口

#### lib\_money\_api——背包的数值资源操作接口

#### lib\_item\_api——pack\_item的公共操作接口，与玩家数据无关

serialize\_proto\_item(#pack\_item{}) 返回#’ProtoItem’{}

unserialize\_proto\_item(#’ProtoItem’{}) 返回#pack\_item{}

serialize\_proto\_item\_list([#pack\_item{}]) 返回[#’ProtoItem’{}]

### lib\_mail\_api——邮件模块

在单服上下文里发送系统邮件的使用示例：

MailInfo0 = lib\_mail\_api:create\_sys\_mail(邮件附件的流水类型),

MailInfo1 = lib\_mail\_api:set\_title\_content(MailInfo0, 邮件配置id, [邮件内容参数列表]),

MailInfo2 = lib\_mail\_api:add\_item(MailInfo1, 道具id, 数量),

lib\_mail\_api:send(MailInfo2, 接收者角色id).

在跨服玩法中发系统邮件的接口：

lib\_mail\_api:send\_in\_cluster(MailOffline, ReceiverId, ServerFlag, ClusterType)

### lib\_broadcast\_api——全服聊天公告广播模块

在单服上下文里发送全服公告广播的使用示例：

ShoutMsg0 = lib\_broadcast\_api:create\_broadcast(广播号),

ShoutMsg1 = lib\_broadcast\_api:push\_broad\_arg(ShoutMsg0, 1), %% 广播参数1

ShoutMsg2 = lib\_broadcast\_api:push\_broad\_arg(ShoutMsg1, "test"), %% 广播参数2

lib\_broadcast\_api:send\_all\_broadcast(ShoutMsg2). %% 发送全服广播

%% 联盟公告广播，其中LeagueId为联盟id

lib\_broadcast\_api:send\_league\_broadcast(LeagueId, ShoutMsg)

%% 跨服公告广播，其中ServerFlag是对应跨服分组里的某个服务器标识

lib\_broadcast\_api:send\_cluster\_broadcast(ShoutMsg, ServerFlag)

### lib\_timer——时间处理模块

1. sleep(Ms) 休眠阻塞多少毫秒
2. unixtime() 获取当前时间戳
3. unixtime({{Y,M,D},{H,M,S}}) 返回指定日期的时间戳
4. to\_localtime(Tick) 把指定的时间戳转换为日期格式：{{年,月,日}, {时,分,秒}}
5. current\_zero\_unixtime() 返回当天的零晨的时间戳
6. next\_zero\_unixtime() 返回第二天的零晨时间戳

### lib\_types——类型转换模块

1. to\_integer(Val) 转为整数
2. to\_float(Val) 转为浮点数
3. to\_binary(Val) 转为二进制格式数据，  
    如果是中文，则会调用unicode:characters\_to\_binary(Val, utf8)转换
4. to\_atom(Val) 转为原子类型
5. to\_list(Val) 转为列表或字符串
6. int\_to\_long(High, Low) 把两个32位整数合并为一个64位整数
7. long\_to\_int(Val) 把一个64位整数分解为{高32位整数, 低32位整数}

### lib\_random——随机模块

1. rand(Val) 在[1~N]区间随机一个数
2. rand(Min, Max) 在[Min~Max]区间随机一个数
3. rand\_float(Min, Max) 在[Min~Max]区间随机一个浮点数，精度是四位小数
4. rand\_one(List) 从列表中随机抽取一个元素
5. random\_sort(List) 对列表进行随机排序

### lib\_config\_api——配置预处理模块

主要目的是预处理读取配置，按字段类型归类，减少功能的遍历配置项的数目，加快配置的读取判断

1. reload()  
    预读取配置处理接口

### Ets缓存处理模块

#### lib\_game\_cache——游戏的缓存接口模块

1. 从缓存ets中获取玩家指定功能的数据，不存在则从数据库中读取

lib\_game\_cache:get\_player\_cache(Ets, PlayerId)

1. 从Ets中获取联盟数据，只能读，不能修改

lib\_game\_cache:get\_league\_cache(LeagueId)

1. 从Ets中获取联盟成员的数据，只能读，不能修改

lib\_game\_cache:get\_league\_member\_cache(LeagueId, MemberId)

### lib\_player\_flag——玩家标识模块

#### 战力定时更新标识

1. 设置战力更新标识

lib\_player\_flag:set\_player\_force\_flag(1)

1. 检查更新战力的接口

lib\_player\_fight:check\_update\_player\_force()

#### 显示模型ProtoShowRole定时更新标识

1. 设置ProtoShowRole更新标识

lib\_player\_flag:set\_player\_show\_flag(1)

1. 检查更新ProtoShowRole的接口

lib\_player\_copy:check\_update\_player\_show()

1. ProtoShowRole有更新时其他功能的同步接口在此接口处添加：

lib\_player\_copy:listen\_update\_show\_role()

#### 排行定时更新标识

1. 设置等级排行的更新标识

lib\_player\_flag:set\_rank\_player\_level(1)

1. 设置战力排行的更新标识

lib\_player\_flag:set\_rank\_player\_force(1)

1. 检查更新同步排行信息的接口

lib\_player\_rank:check\_update\_rank()

### lib\_player\_copy——玩家机器人数据模块

此模块的数据用于构造玩家的战斗时需要显示和战斗需要的数据

### back\_activity——后台运营活动模块目录

已有累积充值的后台运营活动作为参考示例

### ban——后台封禁模块目录

目前支持封禁帐号，封禁IP，禁言（静默禁言，即禁言后只有玩家自己看到自己的聊天信息，其他玩家看不到）

### cluster——跨服分组模块目录

目前支持自动分组和后台手动分组，每个跨服玩法功能各自单独设置，根据cluster\_type对应不同的跨服玩法

## 启动流程



## 玩家登录流程

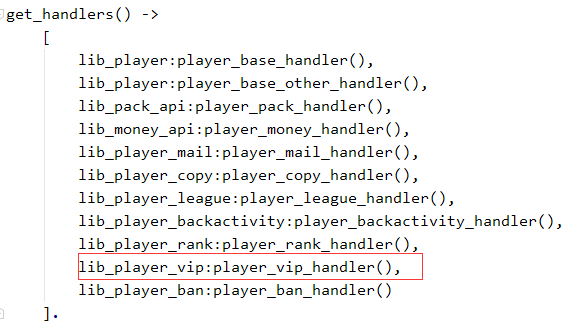
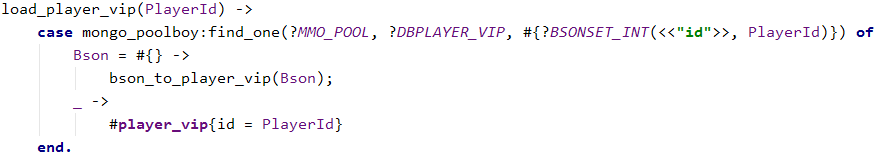
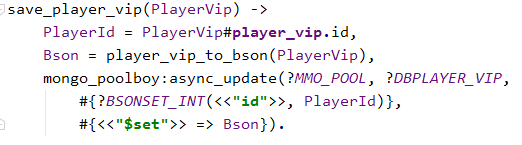
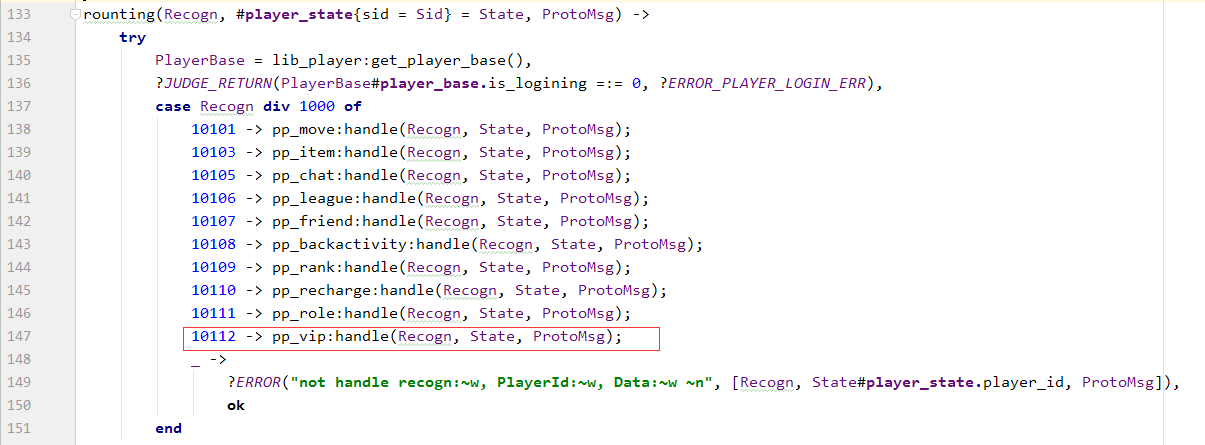


## 玩家离线流程



## 玩家添加新功能的基本接口定义

以vip为例：

1. 定义功能的基本玩家模块：lib\_player\_vip，此模块的基本接口
   1. put\_player\_vip(PlayerVip, IsSaveDb)  
      此接口必须调用：lib\_player:check\_update\_ets\_cache(IsSaveDb, ?ETS\_PLAYER\_VIP)  
      其中，?ETS\_PLAYER\_VIP是ets表的宏定义
   2. get\_player\_vip()
   3. ets\_init()  
      玩家对应的ets表初始化定义：  
      ets:new(?ETS\_PLAYER\_VIP, [?ETS\_KEY\_POS(#player\_vip.id) | ?ETS\_OPTIONS]).
   4. build\_index()  
      功能的mongo数据表的索引定义：  
      mongo\_poolboy:async\_ensure\_index(  
       ?MMO\_POOL,   
       ?DBPLAYER\_VIP,   
       #{?BSONSET\_INT(<<"id">>, 1)},   
       true)  
      其中，最后的参数是布尔型，true表示唯一索引，false表示普通的查找索引  
      必须所有的数据库查找条件都建立索引
   5. player\_vip\_handler()  
      玩家功能数据统一接口列表定义#player\_handler{}：  
      #player\_handler{  
       ets = ?ETS\_PLAYER\_VIP,  
       ets\_init\_func = fun lib\_player\_vip:ets\_init/0,  
       mongo\_index\_func = fun lib\_player\_vip:build\_index/0,  
       load\_func = fun db\_player\_vip:load\_player\_vip/1,  
       save\_func = fun db\_player\_vip:save\_player\_vip/1,  
       get\_func = fun lib\_player\_vip:get\_player\_vip/0,  
       put\_func = fun lib\_player\_vip:put\_player\_vip/2  
      }.
   6. 绑定handler接口，修改lib\_player\_handler:get\_handlers()：  
      
2. 定义玩家的数据库保存接口模块db\_player\_vip，此模块的基本接口：
   1. load\_player\_vip(PlayerId)  
      从数据库中加载数据生成：#player\_vip记录  
      
   2. save\_player\_vip(PlayerVip)  
      把玩家功能数据保存到数据库中  
      
3. 定义功能的客户端请求处理模块pp\_vip，此模块的接口格式：
   1. handle(Recogn, State, ProtoMsg)
   2. 在玩家消息分发路由的接口里添加处理入口：  
      mod\_player:rounting  
      
4. 如果该功能需要调用其他功能的监听触发接口，可以考虑定义\_listen结尾的模块，如：lib\_player\_vip\_listen  
   