[1. 赠送用户3天的VIP（类似的还有赠送5天，7天等）(作者miller) 2](#_Toc388279508)

[2. 把某个功能点亮到百分之百(作者miller) 2](#_Toc388279509)

[3. 什么样的功能可以放到配置表里？(作者miller) 3](#_Toc388279510)

[4. 通用打怪流程接口(作者rudy) 4](#_Toc388279511)

[5. GM增加游戏道具(作者lemon) 7](#_Toc388279512)

[6. 如何更方便测试BUFF功能(作者lemon) 8](#_Toc388279513)

[7. 外网数据导入内网 (作者lemon) 8](#_Toc388279514)

[8. 一键登入 (作者lemon) 10](#_Toc388279515)

[9. 全局属性 (作者lemon) 10](#_Toc388279516)

[10. Switch服务器设计 (作者genry) 11](#_Toc388279517)

[11. 战斗条件调整(作者jimmylieu) 15](#_Toc388279518)

[12. 米币物品自动转免费物品(作者jimmylieu) 16](#_Toc388279519)

[13. 邮箱系统设计(作者vergilxu) 16](#_Toc388279520)

[14. 十二轮的类实现说明(作者 kaven) 23](#_Toc388279521)

[15. 赛尔号超进化汇总设计说明(作者 bell) 26](#_Toc388279522)

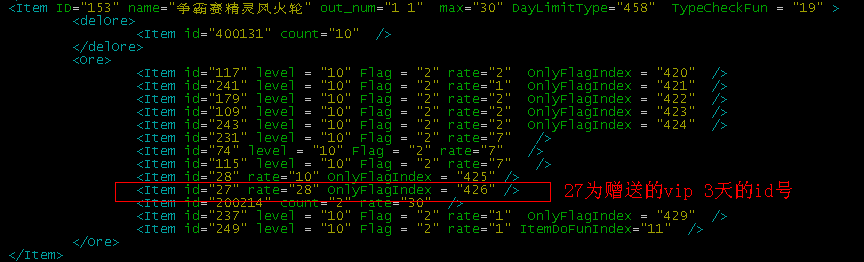
[16. 赛尔号老系统扩展案例—全能刻印系统后台开发(作者 toby) 27](#_Toc388279523)

[17. 赛尔号PVP对战流程(作者 krischien) 34](#_Toc388279524)

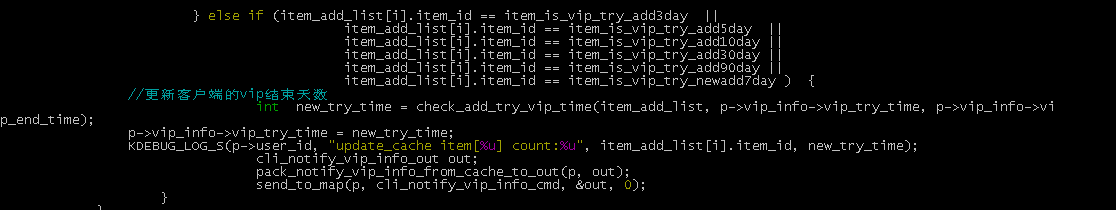
[18. 花仙PVE游戏开发思想(作者 kevinzhang) 35](#_Toc388279525)

# 赠送用户3天的VIP（类似的还有赠送5天，7天等）(作者miller)

解决思路：把赠送VIP的3天在item表中分配一个ID号，在配置表和前端的协议上把其视为物品处理，这样在配置表的时候就可以跟其他普通物品一样配置在配置表中：



在增加VIP 3天的时候，也是调用增加物品的接口，这就需要在增加物品的统一接口中特殊处理VIP 3天的逻辑，类似如下：



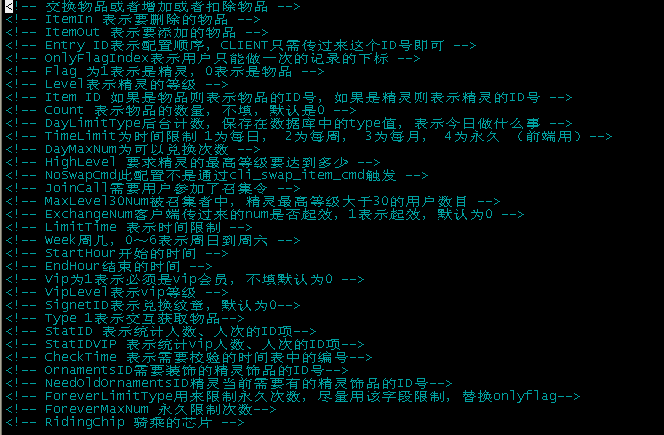
在后台开发中，很多东西虽然不是一个真正的物品，但可以给他一个ID号，当成一个物品处理，在加物品的底层屏蔽这种差异性，这样给后续的运营活动中增加类似的物品会带来比较大的便利性。

# 把某个功能点亮到百分之百(作者miller)

解决思路：可以把每点亮一点，看成给用户送一个物品，这个物品不需要在背包里显示出来。这样对于点亮的功能可以不需要开发新的功能，只要按给用户送物品的思路去做就可以了，如果用户的物品达到一百个，就认为是百分百点亮。

有些功能我们可以变换思路，看看能否能够使用现有的系统，如果能够使用现有的系统即减少了开发量，同时也提高了质量，你写的代码越少意味着你出错的可能性也越小。

# 什么样的功能可以放到配置表里？(作者miller)



以上是一个配置表里可以配置的字段的说明。

<Entry ID = “1”>

<ItemOut>

<Item ID = “400013” Count=”1” >

</ItemOut>

</Entry>

以上是增加一个物品的功能，如果在此基础上要限制每天的输出，可以这样配置：

<Entry ID=”1” DayLimitType=”100” DayMaxLimit=”1”>

<ItemOut>

<Item ID=”400013” Count=”1”>

</ItemOut>

</Entry>

以上DayLimitType表示每天限制的type值，DayMaxLimit表示每天限制的次数，如果有类似的每周，每月限制也可以通过类似的配置完成。如果要统计获取物品的人数和人次，可以加StatID字段，如下：

<Entry ID=”1” DayLimitType=”1” DayMaxLimit=”1” StatID=” 0x10030616”>

<ItemOut>

<Item ID=”400013” Count=”1”>

</ItemOut>

</Entry>

如果在某种情况下才能获取这个物品，这个某种情况可能根据不同的需求有不同的实现，这时可以在配置表中增加钩子函数，配置如下：

<Entry ID=”1” TypeCheckFun=”1” DayLimitType=”1” DayMaxLimit=”1” StatID=” 0x10030616”>

<ItemOut>

<Item ID=”400013” Count=”1”>

</ItemOut>

</Entry>

TypeCheckFun的配置就是我们实现的钩子函数的下标，在送物品的时候需要调用这个函数，只有这个函数返回为真的情况下，才送出物品。这种情况下需要写代码，但他也明显的好处，这时写代码只需要关心校验这部分代码，只要保证校验这部分代码正确即可，同样的可以达到减少开发量和提生代码质量的目的.

当然我们还可以加其他的功能到配置表里面，有时候我们需要思考一下，某个需求如果是多次出现，这个时候我们就需要考虑是否通过配置表来相对统一的解决这个问题。为了应对我们快节奏的开发和高质量的要求，在做案子的时候，需要我们能够分辨出需求当中哪些是固定的，哪些是变化的，变化的需求能不能通过配表解决。

# 通用打怪流程接口(作者rudy)

背景：热血的战斗是自动战斗，其流程为client-->online-->battle-->online-->client，而对游戏而言，对战模块是游戏的核心模块，策划喜欢在对战上面玩些小花样。

分析：对战的一般实现流程为与前端约定好cmd-id，后台注册一个处理函数，在处理函数里面实现处理逻辑，具体包括：

1. 判断对战是否可行。
2. 调用接口，向battle发包，开始对战。注册一个回调处理函数。
3. 在回调处理函数里面处理battle回包。

目前可以想到的所有的对战都需要走以上步骤，因此可以将以上步骤封装起来，再针对不同的需求，增加对应处理逻辑即可。

方案：要点在于在client的战斗发包使用同一个cmd-id，再增加一个enum字段，后台通过这个enum字段来选择不同的处理逻辑。后台的结构为：

BattleVsMonsterDirectlyCmdProcessor{

public:

int proc\_pkg\_from\_client(player\_t\* player, const char\* body, int bodylen);

int proc\_pkg\_from\_serv(player\_t\* player, const char\* body, int bodylen);

}

为针对cmd-id注册的处理类。

class BattleVsMonsterReason

{

public:

virtual int check\_if\_can\_fight(

player\_t\* player, onlineproto::cs\_battle\_vs\_monster\_directly& info) = 0;

virtual int aft\_fight(

player\_t\* player, onlineproto::sc\_battle\_vs\_monster\_directly& info) = 0;

}

为针对不同enum值的处理类的父类，子类通过继承父类，实现两个接口，并在

BattleVsMonsterDirectlyCmdProcessor的构造函数里面针对enum值进行注册，便可实现对应的业务需求。其中check\_if\_can\_fight为对战前调用以判断对战能否开始（是否有某种道具等等），aft\_fight为对战后进行一些逻辑处理。

效果：

1. 无需处理战斗的封包和回包，减少了开发量，可以少记一些与封包和回包相关的api~
2. 可以通过修改BattleVsMonsterReason的接口，以及

BattleVsMonsterDirectlyCmdProcessor的发包，回包流程来支持更多的操作，具体例子见下。

c,可以在BattleVsMonsterReason中添加一些函数，方便流程的实现，由于是在父类中实现，这些函数会被所有子类拥有，具体例子见下。

经过一段时间后，BattleVsMonsterReason类变成了这个样子：

class BattleVsMonsterReason

{

public:

virtual int check\_if\_can\_fight(player\_t\* player, onlineproto::cs\_battle\_vs\_monster\_directly& info) = 0;

virtual int aft\_fight(player\_t\* player, onlineproto::sc\_battle\_vs\_monster\_directly& info) = 0;

//对于某些特殊需求，在对战开始前改变一下monster的状态，如果实现这个接口，需要同时实现is-modify-monster接口并返回true

//应该会有更好一点的实现方式……

virtual monster\_t modify\_monster(player\_t\* player, const monster\_t\* monster){

monster\_t tmp;

return tmp;

}

virtual bool is\_modify\_monster()

{

return false;

}

//用于选择战斗所要使用的阵型（比如使用特定的system-tactic）

virtual tactic\_type\_t get\_tactic\_type()

{

return kTacticNormalBegin;

}

protected:

bool check\_is\_win(onlineproto::sc\_battle\_vs\_monster\_directly& info);

uint32\_t get\_monsterid(onlineproto::cs\_battle\_vs\_monster\_directly& info);

uint32\_t get\_monsterid\_ret(onlineproto::sc\_battle\_vs\_monster\_directly& info);

inline uint32\_t get\_total\_damage(onlineproto::sc\_battle\_vs\_monster\_directly& info)

{

return info.battle\_info().atk\_total\_damage();

}

//加奖励，并写在战斗回包中

uint32\_t add\_item\_to\_prize(player\_t\* player, uint32\_t item\_id,

uint32\_t count, onlineproto::sc\_battle\_vs\_monster\_directly& info);

};

如上，对于策划提出的新的需求，可以通过妥善的修改父类来支持，并且如果修改得当，系统以后就可以支持类似需求了。

一个使用此接口的例子如下：

//判断是否可对战

int BattleVsDragonRoarEgg::check\_if\_can\_fight(

player\_t\* player, onlineproto::cs\_battle\_vs\_monster\_directly& info)

{

uint32\_t err = BattleUtils::check\_can\_pve(player);

if (err) {

return err;

}

if (get\_monsterid(info) != tDragonRoarEggMonsterId) {

return cli\_err\_battle\_vs\_monster\_err;

}

if (AttrUtils::get\_attr\_value(player, kAttrDragonRoarMonsterCount) < 10) {

return cli\_err\_lack\_usable\_item;

}

int ret = GamesProcessorUtils::check\_count(player, kDailyDragonRoarGameCount,

kDailyDragonRoarBuyGameCount, tDragonRoarEggGameCount);

if (ret) {

return ret;

}

return 0;

}

//对战以后奖励

int BattleVsDragonRoarEgg::aft\_fight(

player\_t \*player, onlineproto::sc\_battle\_vs\_monster\_directly& info)

{

AttrUtils::set\_single\_attr\_value(player, kAttrDragonRoarMonsterCount, 0);

AttrUtils::add\_attr\_value(player, kDailyDragonRoarGameCount, 1);

if (check\_is\_win(info)) {

AttrUtils::add\_attr\_value(player, kAttrDragonRoarGoldEgg, 2);

Utils::write\_msglog\_new(player->userid, "活动", "活动触及/龙之咆哮", "一次性成功

获得2个蛋");

} else {

AttrUtils::add\_attr\_value(player, kAttrDragonRoarGoldEgg, 1);

Utils::write\_msglog\_new(player->userid, "活动", "活动触及/龙之咆哮", "一次性成功

获得1个蛋");

}

return 0;

}

总结：打怪的流程的特殊性在于，第一，策划喜欢改，加各种东西进去。第二，所有数据在online封装好，数据的结构不会随着需求而变化（虽然数据本身的数值计算方式可能会变，这也就是为什么会有modify-monster接口）。因此，可以使用注册具体处理流程的方式来进行封装，简化跨服调用的开发（在BattleVsMonsterReason中是不需要关心任何与异步回调相关的内容的）。而在实际开发中发现完全满足第二点的需求很少，一种可能的方式是通过增加数据的冗余度来实现。关于这个，暂时没有实际做过。

# GM增加游戏道具(作者lemon)

以往，我们通过改数据库来修改数据，现在我们可以通过GM命令来修改，而且不需要重新登录，更加方便。



常用的命令如：

@additem 添加物品

@addpet 添加精灵

@addbean 添加精灵豆

@setattr 设置属性值

@getattr 获取属性值

@addtitle 获取称号

@cleartask 重置任务

实现原理是，截获聊天内容，如果带有@前缀，则当做GM命令来处理，解析命令和参数然后执行。

当然外网是不开放的，可以通过bench.conf来配置，默认不配置表示不开启。

是不是和shell差不多。

# 如何更方便测试BUFF功能(作者lemon)



战斗逻辑比较复杂，如何看数值是否正确;一个buff是否添加上;效果是不是策划预期的;这个闪避值有点高，后台处理对不对啊;需要battle服务器把一些具体数值打包发给前端，这样策划和测试都很省心，数值一目了然。

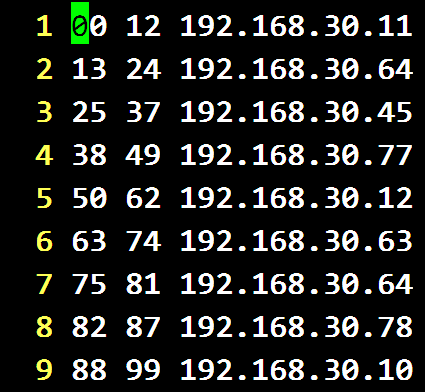
# 外网数据导入内网 (作者lemon)

将外网的真实数据导入到内网，对外网调试问题、策划决定数值有很大的帮助。<http://192.168.30.30/cplan/dump_user/dump.php>

工作原理如下：

1、192.168.30.30开通了和各台mysql连接访问的权限。

2、根据米米号查找一张路由表找到对应的mysql服务器地址。



3、连接到mysql，show tables找到各个表，然后对每个表进行扫描select \* from xxx where userid = xx导出到文件中。

userid=$1

db\_no=`expr $userid % 100 | xargs -I^ printf '%02d' ^`

table\_no=`expr $userid / 100 % 100 | xargs -I^ printf '%02d' ^`

table\_no1=`expr $userid / 100 % 100 / 10`

#查找路由表

host=`get\_host $userid`

#找到库中的各个表

tables=`mysql -u$MYSQL\_USER -p$MYSQL\_PASSWD -h$host -BNe "

SHOW TABLES FROM game\_db\_$db\_no LIKE '%table\_$table\_no'"`

tables1=`mysql -u$MYSQL\_USER -p$MYSQL\_PASSWD -h$host -BNe "

SHOW TABLES FROM game\_db\_$db\_no LIKE '%table\_$table\_no1'"`

#建个临时目录存放数据

mkdir dump123

#遍历各个表

for table in $tables $tables1

do

mysql -u$MYSQL\_USER -p$MYSQL\_PASSWD -h$host --default-character-set=utf8 -BNe "

SELECT \* FROM game\_db\_$db\_no.$table WHERE userid = $userid" > dump123/game\_db\_$db\_no.$table

#sql是要在内网mysql执行的sql

#先删除原来数据，再从文件中导入

sql=$sql"DELETE FROM game\_db\_$db\_no.$table WHERE userid = $userid;"

sql=$sql"LOAD DATA INFILE '/tmp/dump123/game\_db\_$db\_no.$table' INTO TABLE game\_db\_$db\_no.$table;

"

done

4、将导出的数据通过sshpass和scp自动的传到内网。

sshpass -pxxxxx scp -q -P22000 -r dump123 svc@10.1.1.182:/tmp

5、在内网通过mysql LOAD DATA语句导入到数据库中。

echo $sql | mysql -uxxx -pxxx -h10.1.1.182 --default-character-set=utf8

注意：由于没有从账号平台导入密码，所以只能通过**一键登录**来登录导入的账号。

# 一键登入 (作者lemon)

一键登录方便后台登录用户的账号进行数据查看，了解bug的一些现象，使用外网真实账号进行测试。

原理：通过session登录

1. 后台通过0xA122协议向账号平台获取登录session。
2. 将登录session填入到url中，告诉前端是代签登录，无需密码，sid是session，uid是要代签的米米号，fromGameId是从哪个游戏代签过来。

<http://jl.61.com/?sid=7d78ed5a508fb0f79f50fd175b973ad4&uid=19999999&fromGameId=16>

1. 后台开放一个session登录的接口，前端将url中的参数填写到接口中，传给后台。
2. 后台接收到参数后去账号平台验证，账号平台会保存(uid, gameid) => session的映射，我们将uid, gameid, session传给账号平台验证，验证通过后则认为用户是合法用户，允许他进入online。

# 全局属性 (作者lemon)

全局属性一般是全服需要计数的属性，保存在一个不分库分表的数据库中。

CREATE TABLE IF NOT EXISTS other\_db.global\_attr\_table

(

type INT UNSIGNED NOT NULL DEFAULT 0 PRIMARY KEY COMMENT '类型',

subtype INT UNSIGNED NOT NULL DEFAULT 0 COMMENT '子类型',

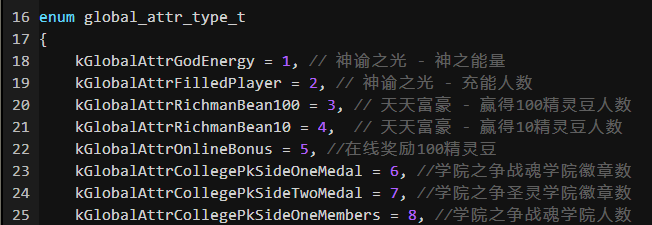
value INT UNSIGNED NOT NULL DEFAULT 0 COMMENT '数值',

PRIMARY KEY (type, subtype)

) ENGINE = INNODB CHARSET = UTF8;

Type一般用户活动，subtype一般按照日期来定义或者是活动中的一些子项。

代码中会建立一系列枚举值



我们定义了一些通过的函数来操作这些全局属性

class GlobalAttrUtils

{

public:

// 向一个全局属性添加值

static inline int add\_attr(player\_t\* player, global\_attr\_type\_t type, uint32\_t subtype, uint32\_t value)

+--- 15 lines: {--------------------------------------------------------------------------------------------

// 获取全局属性的解析，解析到GlobalAttrSet对象中

static inline int parse\_get\_attr\_ack(const char\* body, int bodylen,

GlobalAttrSet& attrs)

+--- 20 lines: {--------------------------------------------------------------------------------------------

// 获取一个全局属性并等待返回

static inline int get\_attr\_wait(player\_t\* player,

uint32\_t type, uint32\_t subtype)

+--- 12 lines: {--------------------------------------------------------------------------------------------

// 获取一系列全局属性并等待返回

static inline int get\_attrs\_wait(player\_t\* player,

const std::vector<uint32\_t>& type\_list,

const std::vector<uint32\_t>& subtype\_list)

+--- 22 lines: {--------------------------------------------------------------------------------------------

};

# Switch服务器设计 (作者genry)

**简介**

Switch在游戏服务器主要有三大作用，包括维护服务器进程的状态、维护用户的在线状态、转发消息。在游戏中，Switch起着至关重要的作用，主要与Login、Online、Vip、DB等服务器相连。根据连接不同的服务器，执行不同的功能。

约定：适用公司项目，且是单服。

**目的**

本篇根据不同服务器的功能需求，对相关的协议进行了设计。通过本篇希望读者对Switch整体有明确的了解。

**设计要点**

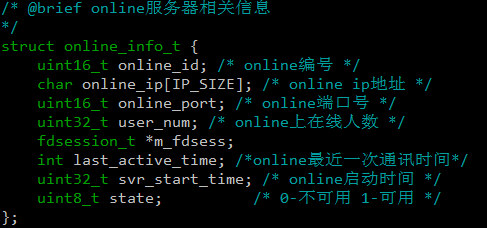
**服务器结构**

服务器结构如图1所示，其中与各个模块的交互如下：

1. Login：包含拉取所有服务器、拉取推荐服务器、拉取所有角色（多角色游戏）、获取用户简单信息
2. Onine：online服务器注册、消息转发、系统踢人、用户上、下线变更状态
3. Vip:充值结果
4. DB:离线物品、离线消息的保存

**协议设计**

**数据结构设计**



结构体online\_info\_t定义了online的信息。

通过map来管理，当online启动起来会设置相关的信息，当用户选择了online\_id后，会更新user\_num。

/\*以online\_id为索引，存每个online信息\*/

typedef std::map<uint16\_t, online\_info\_t> OnlineInfoMap;

OnlineInfoMap online\_info;

通过以下信息管理用户列表，主要用于转发全服消息

/\*以online\_id为索引，存每个online的在线用户list\*/

typedef std::set<userid\_t> online\_user\_set\_t;

std::map<uint16\_t, online\_user\_set\_t> online\_user\_info;

如果想查用户信息在哪条线上，可以用以下结构体

/\*以user\_id为索引，存用户信息\*/

typedef std::map<uint32\_t/\*user\_id\*/, user\_info\_t> user\_map\_t;

user\_map\_t user\_map;

其中user\_info\_t 包含online\_id等用户其它信息，按需定义。

通过封装相关的类来操作以上数据结构，就建立了Switch的核心。

**拉取服务器列表**

1. 分页拉去所有服务器

输入： 起始online号

输出: online信息结构体列表，信息包含online id，online ip、port，及状态

1. 拉去推荐服务器

输入：无

输出：online信息结构体列表，同（1）

**拉取角色列表**

1. 多角色

输入：用户userid

输出：用户简单信息列表，包括用户id、等级、角色信息等。

1. 单角色

输入：用户userid

输出：用户简单信息，包括用户id、等级、角色信息等。

**Online服务器注册**

1. 注册

输入:online id、online ip、online port、svr\_start\_time、用户列表

输出:无

包含online重启和重载时的注册，所有用户列表是需要的

1. 心跳

输入：无

输出：无

通过定时器每隔一定时间来检查online的存活，通过心跳来设置online信息中的last\_active\_time（最后活动时间）。如果发现online没有及时上报状态，说明online服务器已经挂掉，则会更新online\_info，并报警通知。

**用户状态**

1. 上线

输入：online id、客户端fd

输出：无

设置user\_map 和online\_user\_info，插入，注意处理好重复登录踢自己另一端下线

1. 下线

输入：无

输出：无

设置user\_map 和online\_user\_info，删除。

**踢人**

由客服平台踢用户下线。

输入：封号标志、封号原因

输出：封号标志、封号原因

设置用户封停标志和原因，并发向指定online\_id，由具体的online处理通知用户下线

**消息转发**

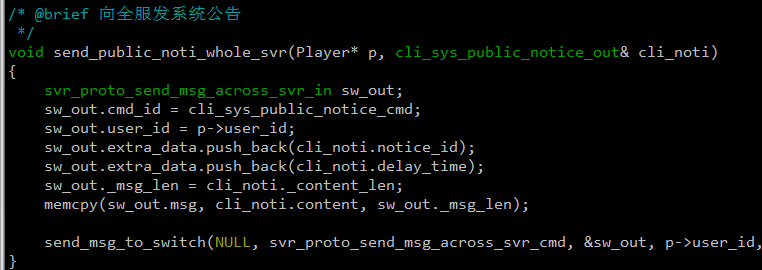
可以通过Swtich转发online上的协议，如公告、世界聊天、好友邮件等

输入:cmd\_id、userid、extra\_data变长参数、msg 变长消息体

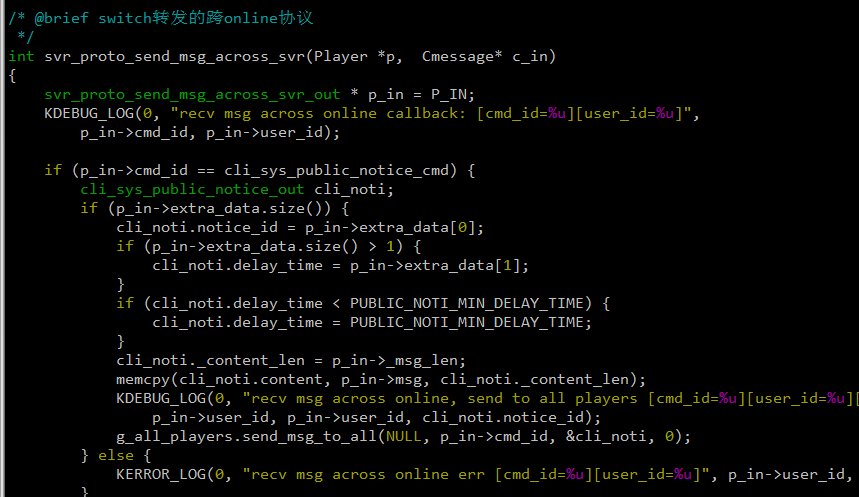
输出: cmd\_id、userid、extra\_data变长参数、msg 变长消息体

举例如全服公告：

打包公告cmd,userid,及其参数notice\_id和delay\_time及公告内容发到switch



当其它online收到swtich转发的消息后进行消息发送



**Vip充值**

从充值serv发过来，用于通知用户充值后逻辑处理，设置 DB相关的充值信息，以及对用户充值优惠如赠送东西等的相关处理。分为在线和非在线。设计时，最好所有相关逻辑都放到online上处理，如赠送变为主动领取，以控制对背包的检查，不要把逻辑写在switch或者DB。下例根据需求来定

输入：充值总金额、本次增加、原vip等级、现vip等级、玩家等级、角色信息等

输出：无

**注意点**

1. Switch目前都是单点，有条件允许情况下，可以配两个，一个挂掉，连接另一个即可
2. 由于Switch主要负责用户在线情况，所以其它需求，如打全服boss等需要另设服务器处理。
3. 功能独立，协议尽量少，降低维护成本

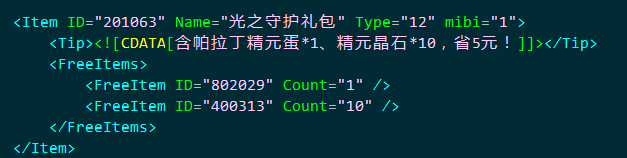
# 战斗条件调整(作者jimmylieu)

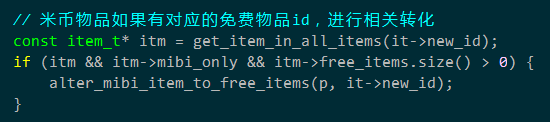
在约瑟传说中，经常需要根据某些条件来触发一场战斗，如果玩家战斗失败，则需要重新达到条件。目前的做法分为2种：第一种，在战斗开始前（即：check\_fight\_mon\_illegal\_xx），把条件置回初始状态，然后再进入战斗，这种做法的弊端在于，有的时候判断完，却没有顺利进入战斗，就有可能浪费玩家达到的条件，比如加载卡住。第二种，在战斗结束分出胜负以后操作，这样玩家如果玩家恶意掉线，又没有特殊处理（特殊处理在这里：offline\_btl\_done），就会导致玩家可以反复刷战斗。所以，现在新添加一个处理函数（deal\_after\_load\_res\_success\_xx，它的定义、位置都和 check\_fight\_mon\_illegal\_xx 类似），该函数在玩家战斗加载成功后触发。如果状态初始化放在这里，就可以保证玩家不会因为战斗资源加载卡住浪费战斗条件，又可以防止玩家恶意掉线来刷战斗。例子：战斗编号为 919 的挑战草莓猫。

# 米币物品自动转免费物品(作者jimmylieu)

matching.xml 是约瑟配置平台商品id和游戏内物品id对应关系的表，以前2个物品转换需要配置处理函数，在函数中手动转换，现添加了一个新函数，可以自动转换。详细如下：

首先在item.xml加入freeitems标签，然后在后买成功后做相关处理：





# 邮箱系统设计(作者vergilxu)

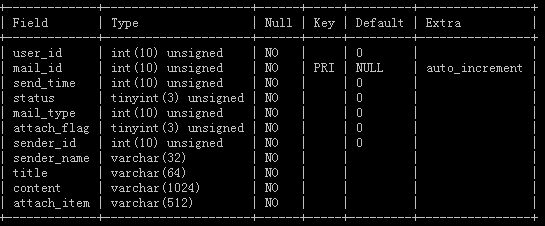
## 第一期策划需求

1. 要求实现系统邮件的发送、接收功能。不实现玩家之间发送邮件功能。
2. 要求邮件中能携带附件，附件可以是物品或精灵。
3. 要求实现阅读邮件和领取附件功能。
4. 要求实现删除邮件功能。

## 设计思路

一、创建邮件数据库表项

根据策划需求确定邮件数据库表项内容如下



其中：

User\_id表示接收邮件用户的米米号；

Mail\_id表示这封邮件的id。由数据库自动自增生成。

Send\_time表示邮件发送时间。

Status表示邮件状态，是否被阅读过。

Mail\_type表示邮件类型，是系统邮件还是其他类型邮件（为后续可能需要的玩家之间互相发送邮件的功能预留）。

Attach\_flag表示是否携带附件。

Sender\_id表示邮件发送人的米米号。

Sender\_name表示邮件发送人的昵称。

Title表示邮件标题

Content表示邮件内容。

Attach\_item表示邮件附件内容。

一、邮件发送功能

由于不需要实现玩家与玩家之间发送邮件，因此无需定义与前段交互协议。邮件的发送接口放在online服务器。每次需要发送系统邮件的时候，则根据策划要求在online上实现一个邮件内容的封装函数然后调用邮件发送接口。将协议发送至DB，再由DB将邮件内容写入数据库。

邮件发送协议内容：



协议中尽可能减少输入的参数，在封装系统邮件中只需要指定sender\_id，系统本身会根据id自动生成对应的sender\_name。由于目前只需要实现系统邮件功能。发送系统邮件的sender\_name都为NPC的名称，因此就在代码中指定sender\_id和sender\_name的对应关系。比如，sender\_id=10，则sender\_name就为“船长辛迪”。

附件为一个变长数组，最对允许携带5个附件。但协议制定过程为防止后续策划需要增加附件因此提前设置附件上限为10 个。系统可以判断附件数组是否为空来自动设置attach\_flag字段。

协议有online发送DB。返回包协议中携带了邮件信息但实际上并没有使用。

关于附件信息的处理：

由于数据库中只能存放整型数据和字符串。而输入的附件内容是一个结构体的变长数组，

结构体内容如下：



因此在将邮件存入数据库的过程中，就存在将附件结构体中3个整型数拼接成一个字符串的工作。最终形成的字符串格式为id,count,flag;id,count,flag;…。

二、邮件接收功能

用户在客户端打开邮箱时，前端会向后台拉取用户所有的邮件信息。



为了防止用户邮箱中邮件数量过多，导致回包的时候数据内容太大。因此限定每次拉取最多100封邮件。



拉取到的邮件信息。

整个拉取过程为同步过程，前端发协议给online，online则发协议给DB，收到DB回包之后，online才会发送返回包给前端。

三、阅读邮件内容

前端根据用户选择，将具体邮件mail\_id和用户米米号发送给后端，后端根据mail\_id在数据库中取出邮件内容返回给前端。



返回包具体内容



在从数据库中读取邮件附件内容时，也存在将字符串转化成结构体数组的过程。

整个过程也是同步过程，类似拉取。

四、删除邮件

用户可以选择删除一封或者是多封邮件。但是如果邮件有附件且没有领取，则邮件不能被删除。前端将用户米米号和mail\_id变长数组发送给后端。后端通知数据库直接删除数据库字段。

删除协议内容：



## 第一期邮箱实现存在的问题

每次策划需要发送新邮件的时候，都需要后台新增邮件封装函数，并且在用户登录的时候调用，将邮件发送到用户的邮箱中。这样就需要后台专门发在线补丁或者等到周四发新版本才能使新邮件生效。同时也会因为新邮件而使online中的邮件代码越来越多，增加维护成本。

基于如上考虑，决定对邮箱系统进行二次开发。

## 第二期策划需求

1. 能够通过客服平台直接给玩家发送邮件。
2. 能够给单一玩家发送邮件，也能给多个玩家或者全服玩家发送邮件。
3. 继承第一期邮箱中的所有功能不变。

4、给单一或者多个玩家发送邮件时，如果玩家在线，则在邮箱图标上会有新邮件提示。

## 设计思路

1. 针对单一或者多个玩家发送邮件，由于需要有在线提醒功能，因此需要switch给客服提供接口来根据米米号判断玩家是否在线，在哪一个online上，然后才能向该online发送新邮件通知。
2. 针对全服玩家的邮件，考虑到每个数据库都有大量的玩家存在。如果采用和单一用户发邮件相同的方式。则DB在一个时间点会需要写入大量的邮件信息，如果发生在玩家登陆的高峰期，必然极大加重DB负载，影响玩家正常游戏。

因此只能先记录需要发送的全服邮件内容，然后在用户登录的时候触发将邮件发送至用户数据库中。

提供给客服平台的邮件发送接口

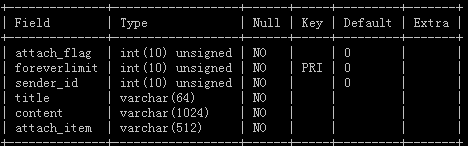
协议内容：



Switch根据flag字段判断是指定用户邮件还是全服邮件。如果是指定对象，则直接将邮件发送至DB，由DB将邮件写入用户数据库。

如果是全服邮件，则switch通知DB将邮件内容写入public\_info数据库中，同时根据当前时间戳生成一个foreverlimit值，用户在登录的时候拉取public\_info中所有foreverlimit值，然后判断是否为0，如果为不为0，则表示该值对应的邮件用户已经接收过。如果不为0，则表示该值对应的邮件用户没有接收。此时再次向DB拉取该值对应的邮件具体内容，然后通过第一期实现的邮件发送接口，将该邮件发送至用户的DB中。

Public\_info表项内容：



用户登录时online向DB拉取public\_info中foreverlimit值协议：



用户根据foreverlimit值拉取对应邮件具体内容的协议：



## 第二期开发后仍遗留问题

目前邮箱没有定时删除过期邮件的功能。因此随着时间推移用户如果不主动删除邮件的话，邮箱中邮件会越来越多。最终影响用户阅读新邮件。因此需要实现一个定时删除过期邮件的功能。

但是由于目前只实现了系统邮件功能，而策划发送系统邮件的频率为一周一封左右。因此要发满100封邮件需要2年多的时间。而大部分用户都不会长期存着过期邮件不删除。通过线上表现来看，大多数玩家都会主动删除邮件，因此这个需求并不是十分紧迫。可以后期在时间允许的情况下，再做进一步优化。

# 十二轮的类实现说明(作者 kaven)

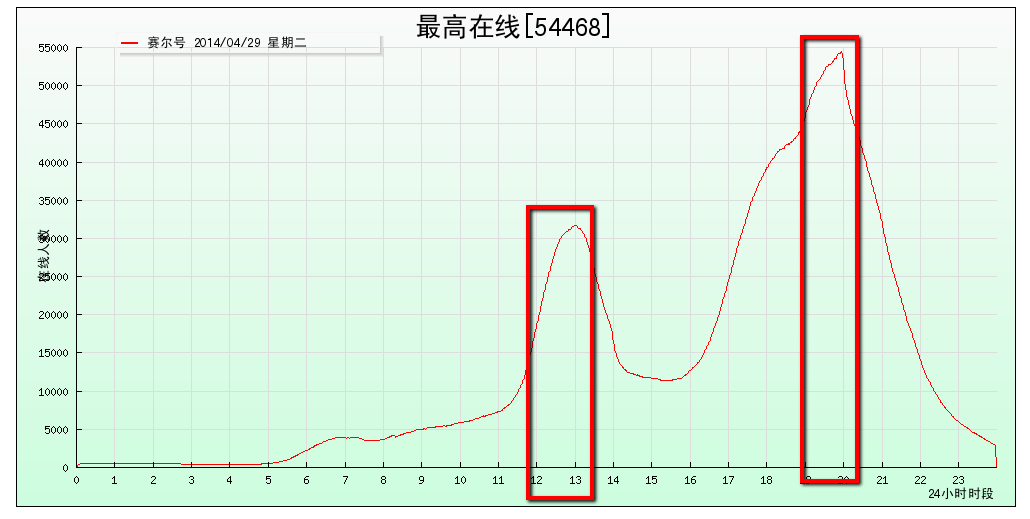
**说明**

12轮是赛尔号中非常受欢迎的常规活动，每个星期都至少会有一个类似的活动。12轮的基本玩法就是一共有12个Boss，每5分钟可以对战一个Boss，对战完成后，无论胜败都可以获得丰富的奖励，包括升级道具、超进化道具、精元、常用道具等。



12轮游戏截图

查看赛尔号的在线，可以看到在最高在线总是在这个12轮活动的第12轮结束时开始下降。



赛尔在线中的12轮活动效果

以前一开始的时候一个12轮是中午和晚上都有，但只能参加一次，即中午打了第1轮后，晚上的第1轮不能打，两个时段是互补的，后来（约2013年第四季度）改成两个时间段都可以参加，而且都可以获得奖励。

**实现介绍**

在很长的一段时间12轮的实现都是通过面向过程的代码实现，而且是直接在已有的函数中插入新的代码。这种实现导致一个函数里的代码迅速膨胀，一个处理对战后的操作的函数已经达到数千行；由于在一个函数里，每次调用这个函数时（这个函数还是常用函数，进而导致了许多不必要的判断）都要全部判断一次。

这样的实现导致了下面的一些问题：一、难以阅读。由于函数冗长，相关的实现相隔很远，有时还出现同样的操作进行多次；二、难以维护。火爆的活动有时候应玩家或运营的要求，需要复用一次，每次复用都需要作一些小的变动，零散的实现让维护的人苦不堪言。三、还有一个不明显但影响很深的问题：多个活动用了同一个字段储存。这个问题并不是必现的，查找到这个问题无论对测试还是后台，都花了巨量的时间。

为了解决这种难以维护的问题，一开始计划设计用配表的方式实现，后来发现必要性不强，因而最终采用类来实现。

12轮活动中变动的主要是输出规则，规则的变动情况非常复杂，配表的功能不容易控制，而且对策划的变动适应性不强，同时实现这一功能也需要耗费较大的人力资源，所以就暂时放弃了这个实现。目前（2014.05.04使用的实现）的实现也应用了时间限制配置表、Boss配置表、奖励输出配置表。

类的实现主要是将基本的流程在基类中实现，每一个新的活动都继承这个基类，并重载输出规则函数，实现活动策划案中的要求。

**实现设计**

NoRegionFightInterface

无地图对战接口

BasicNoMap12Round

12轮基本类

BasicNoMap12RoundEx

扩展的12轮基本类

继承

继承

包含对战前对战后的接口，在12轮中可以控制是否已经对战过、是否还不到对战时间

实现对战前后的处理，包括判断对战的合法性和对战后的数据处理、奖励输出，奖励输出调用虚函数，需要子类实现

扩展后的12轮基本类，实现一天可以参加两次的要求

具体类

12轮活动实现

继承

具体的实现类，主要就是初始化相关数据、实现奖励输出逻辑

这里的实现继承了无地图对战的虚类（由aubrey设计的），简化了对战前后的操作，还把相关的代码放在一起方便维护。

BasicNoMap12Round是最初的实现，后来策划变动了基本的玩法（即两个时间段要求都能参加），于是新加了一层继承，重写了需要变动的函数，从而实现平滑地切换到新流程里。

具体的实现细节参考文档《doc-用无地图实现12轮的方法.docx》。

**进一步设计**

目前12轮的实现有较大的改进，但仍有些不足的地方：每个12轮的实现需要变动的地方仍然比较多，涉及了多个配表、多个代码文件，维护仍不太方便，后续是不是可以将这些都放在一起？用配表实现？

使用自动生成工具生成？

# 赛尔号超进化汇总设计说明(作者 bell)

**策划需求**

超进化指消耗某(几)种特定道具将一种精灵变为另一种精灵

**设计思路**

1. 原实现方式：

因为需求简单（扣用户物品后修改精灵id），原实现也相对简单，既通过一个协议函数完成操作，然而随着策划不断提出超进化需求，超进化的经历不断增加，导致该协议函数及其回调函数变成一个几百行的大函数，而增加了维护难度。

1. 汇总实现设计思路

汇总主要解决两个问题：

1. 通过配表来统一管理超进化功能，简化新增超进化精灵的工作量。
2. 兼容以往种类繁多的唯一性限制以及通过注册函数的形式进行特殊处理。

最终结构如下：

配置文件

super\_evolve.xml

Load

全局数据

std::map<uint32\_t, super\_evolve\_handler\_t>

std::map<uint32\_t, super\_evolve\_t>

超进化协议

super\_evolve\_handler\_t

super\_evolve\_t

通过读super\_evolve.xml 配表使用配表对应id为key生成两个全局的map，其中super\_evolve\_t 存放需要的道具，各种唯一性标志，源精灵id，超进化(目标)精灵id等超进化所需的信息，super\_evolve\_handler\_t则是指向自定义函数的指针，在客户端发出请求协议进行超进化时，函数根据super\_evolve\_t的信息做校验和设置相应的标记位，并在消耗道具之前和进行回调函数超进化之前调用（如果有的话）自定义的函数做特殊处理。

通过这种方式实现了超进化的汇总

**存在的问题和思考**

尽管对兼容旧的多种唯一性标记做了一些努力并实现了这一需求，但是由于策划和测试最终坚持要求采用另一种方案，即跟据外网数据生成新的统一的标记位，而最终白费了功夫。然而跑数据也并不是一了百了的做法，当复用旧的超进化案子时，就需要将代码中旧的标记位替换为新的标记位。

造成这些痛苦的原因是系统设计之初没有对用户唯一性输出做统一处理，后期策划提出的用户唯一需求便只能在输出逻辑的外围来判断和设置。而外围的判断条件多种多样同时对限制标记的命名也因程序员习惯不同而十分混乱，当某种道具或精灵需要重新输出时，搜索这些标记则变成了十分痛苦的过程。为解决这一问题，修改旧的输出接口显然是费时费力且容易出错的，最简单的办法是统一新的唯一限制的方式和在该方式上统一命名规范。例如，赛尔号目前通过bitbuf来表示用户的某种唯一性操作记录，而我们可以约定所有的唯一道具输出都使用bitbuf来限定，bitbuf的命名中都使用带相关id的描述来命名，例如：限制道具山岳石(1700771)输出的标记名称为bitbuf\_item\_ 1700771\_gotten.

# 赛尔号老系统扩展案例—全能刻印系统后台开发(作者 toby)

## 旧刻印功能与全能刻印简介

1. **旧刻印功能说明**



赛尔刻印是精灵装备的一种，用来强化精灵属性或者战斗技能特效，分为技能刻印和能力刻印两种。在每只精灵身上有三个刻印孔位置，分别对应能力刻印、技能刻印、通用刻印激活孔，可以安装或卸下对应类型的刻印，安装后的刻印会对精灵有相应的效果加成。

1. **全能刻印需求**



在旧刻印的基础上增加一种新的全能刻印，兼容游戏中现有的刻印属性和操作流程。刻印功能与前两种不同，并增加刻印等级、品质、刻印宝石、刻印宝石孔、宝石特效等新属性概念，增加刻印绑定、强化、重铸、分解、宝石合成、宝石镶嵌等操作

## 扩展步骤

### 简单估算是否“大案子”

通过对游戏功能的了解，和源码mintmark搜索，可以确认是个需要改动online、dbsvr 、

battle和客服平台等物品辅助接口的多服务变更需求，并且新增的操作模式和宝石逻辑也有较大开发量，可以初步定性为“大功能”开发。

### 旧刻印功能模块、存储结构与配表结构整理

* **功能模块，交互协议整理**

**功能模块**

|  |  |
| --- | --- |
| 1 玩家操作 | 安装，卸载 |
| 2 数量变更 | 活动输出，客服平台添加，后台脚本增加  游戏内没有扣除逻辑，只能通过sql语句删除 |
| 3 属性相关 | 战斗能力值加成显示、特效显示  战斗能力值加成实现、刻印特效功能实现 |

**协议整理，协议是逻辑功能扩展的基本入口，漏考虑一条都会出现bug或者突发**

|  |  |
| --- | --- |
| as-online协议 | 通过游戏内操作需求+online头文件关键词搜索确定 |
| 2359 | 替换精灵刻印 |
| 2360 | 拉取刻印列表 |
| 2122 | 激活通用刻印孔 |

|  |  |
| --- | --- |
| online-btl协议 | 通过刻印功能分析+搜索关键词确认 |
| btl\_fight\_xx系列 | 在打包精灵信息中加入了3个刻印id字段，btl\_svr解包后与本地配表结合，完成属性和特效加成 |
| gbtl\_fight\_xx系列 | 在打包精灵信息中加入了3个刻印id字段，btl\_svr解包后与本地配表结合，完成属性和特效加成 |

|  |  |
| --- | --- |
| online-db协议 | 通过db头文件关键词搜索确认 |
| 0x1319 | 增加刻印 |
| 0x121a | 拉取用户刻印信息(客服) |
| 0x1320 | 安装或卸载刻印 |
| 0x1221 | 获取未使用的刻印 |
| 0x840b | 获取同一id的刻印数量 |
| 0x8429 | 增加刻印（客服） |
| 0x8436 | 替换刻印（一个活动单独用的，非通用接口） |

* **配表**

对可能有刻印的配表进行检查

grep -i "mintmark\|刻印" \*.xml | awk -F: '{print $1}' | sort –u

|  |  |
| --- | --- |
| 独立配表 | mintmark.xml |
| 刻印ID和属性配置，eg:  <MintMark ID="10174" Type="0" Effect="1" Arg="0 0 0 0 0 55" Level="100" Max="99" Des="神魔之印-体力" EffectDes="体力+55"/>  <MintMark ID="20001" Type="1" Effect="103" Arg="1" Level="100" MoveID="10001" Max="99" Des="撞击刻印" EffectDes="先制+1" /> |
| 关联配表 | exchange.xml  no\_wait\_bonus.xml  tasks.xml  top\_fight\_exchange.xml |
| 刻印都是作为输出子项，可以配置在这些配表中 |

* **存储结构**

存储结构整理，先找出基本信息结构，主要方向是解析表结构，内存刻印结构，输出结

构和db表结构，分别在Online battle db服务中确认。

|  |  |
| --- | --- |
| 解析配表结构 | mintmark\_info\_t attribute\_mintmarks[max\_mintmark\_num];(online battle共用)  mintmark\_info\_t skill\_mintmarks[max\_mintmark\_num];(online battle 共用) |
| 刻印信息结构 | mintmark\_info\_t (online battle 共用) |
| 输出结构 | exchange\_mark\_out\_t  mintmark\_arr\_out\_t  db\_add\_mintmark\_t |
| db表结构 | PP\_00-99.t\_pp\_mintmark\_00-99 (show tables like '%mintmark%';) |

### 需求拆分&工作周期评估

**完成现有功能代码与存储结构整理后，与新需求文档对比**，可以发现刻印与as-操作方

面的旧协议不多，新旧功能的主要耦合部分是战斗效果加成和输入输出数值计算方面，另外还有两个基本概念冲突。

一是旧刻印是以刻印id做唯一标识，同id刻印属性相同，在db中没有做唯一区分，接口也都以刻印id做key，新全能刻印是随机强化属性，每个刻印都有自己的特定值，必须另外生成唯一key来操作。

二是需求案中要兼容旧刻印孔的安放逻辑，而旧刻印孔字段，记录的是刻印id，由于冲突一这种方式不能用于新的全能刻印，需要加一层能区分新旧刻印的映射来实现兼容。

完整需求拆分为三个部分，完成配表和存储设计后三人同时开发，预估时间2周完成（非独立开发，2周内还有其他小案子任务）。

|  |
| --- |
| 需求模块划分 |
| 1刻印操作逻辑和战斗特效实现（包括所有新旧耦合功能） |
| 2宝石玩法实现（与1的唯一交互点是宝石镶嵌） |
| 3 db接口与配表解析实现，辅助平台修改 |

### 开发流程

#### 源码库建立分支&搭建独立登陆环境

这种多服务修改一般会影响登陆、对战和其他同事的开发案，所以需要建立分支进行开

发。我们的版本库管理软件统一用的是svn，建立分支方法，以online为例：

svn copy http://svn.taomee.com/seer/server/online/trunk

http://svn.taomee.com/seer/server/online/trunk\_20140504 -m “cut

online:online\_20140504”

独立登陆环境后台比较容易实现，主要server用分支版本新配置一套即可。

#### 配表结构扩展

**刻印配表设计方案有两种备选，一种是新增刻印配表，独立解析。一种在旧配表新增字段，在原解析结构中加入新类型。实际选择第二种方案**，但因为新旧刻印除了id和type外，属性字段名称全部不同，解析结构还是独立存放，只在刻印配表load时调用了新全能刻印的解析函数，存放到全局结构中。

int load\_mintmark\_config(xmlNodePtr cur)

{

…

if (load\_all\_function\_mintmark\_config(cur) == -1) {

return -1;

}

return 0;

}

/\*! all\_function刻印解析结构 \*/

std::map<uint32\_t, mintmark\_info\_t> g\_all\_function\_mintmarks;

宝石配表可以独立实现，包含宝石升级，分解，失败补偿，宝石特效等属性，新增gems.xml配表。泰坦之灵，宝石，宝石灵等物品操作需求同普通物品，所以在物品表中新增两类，不再新建配表。

所有配表都在online和btl同步修改，db不变。

#### 存储结构扩展

**存储结构扩展主要依据有两点，一是按照文案需求，在t\_pp\_mintmark表中加入了全**

**能刻印的属性记录字段，二是为了解决前面的唯一id问题，加入了获得时间唯一化生成处理**。

mysql> desc t\_pp\_mintmark\_99;

+-------------+------------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+-------------+------------------+------+-----+---------+----------------+

| id | int(10) unsigned | NO | PRI | NULL | auto\_increment |

| userid | int(10) unsigned | NO | MUL | NULL | |

| mintmarkid | int(10) unsigned | NO | | NULL | |

| catchtime | int(10) unsigned | NO | | 0 | |

| type | int(10) unsigned | NO | | 0 | |

// 刻印获得时间，新全能刻印插入时会做唯一化处理

| obtain\_tm | int(10) unsigned | NO | MUL | NULL | |

// 新增字段

| bind\_monid | int(10) unsigned | NO | | NULL | |

| bind\_moveid | int(10) unsigned | NO | | NULL | |

| atk\_adj | int(10) unsigned | NO | | NULL | |

| def\_adj | int(10) unsigned | NO | | NULL | |

| sp\_atk\_adj | int(10) unsigned | NO | | NULL | |

| sp\_def\_adj | int(10) unsigned | NO | | NULL | |

| spd\_adj | int(10) unsigned | NO | | NULL | |

| hp\_adj | int(10) unsigned | NO | | NULL | |

| gem\_id | int(10) unsigned | NO | | NULL | |

+-------------+------------------+------+-----+---------+----------------+

唯一化处理方法：

* 在用户信息表t\_pp\_user\_info里面记录一个上次刻印获得时间戳last\_mintmark\_time，每新增一个全能刻印，判断当前时间是否不大于last\_mintmark\_time，是的话将obtain\_tm设置为last\_mintmark\_time+1插入，否则设置为当前时间。
* 为了操作和开发方便，新全能刻印采用了全缓存的做法，旧刻印还是保留以前的存放拉取方法不变。通过db接口增加的刻印（测试工具、客服平台、后台脚本等），由于我们的dbsvr没有主动同步online缓存的机制，所以采用延后加入方案，用db加新全能刻印时设置一个MAGIC\_TIME，在用户下一次登陆时才生成唯一id。

// 客服添加刻印的占位值, 用于重新登录时刷新属性和获得时间

kf\_all\_function\_mintmark\_tmp\_time = 1000000000,

#### 交互接口扩展

**交互接口扩展是主要开发工作，设计依据同存储扩展，一是按新增案子需求，加入重**

**铸、强化、合成、分解、升级、宝石镶嵌等操作逻辑，并完成关联的db接口开发**（对全能刻印属性相关的查询设置接口全部用新协议实现，**对刻印数量、刻印操作方面的旧接口需要逐条修改**）。**二是对之前的刻印镶嵌逻辑进行扩展，这里通过沟通和统计确定刻印id的需求范围有限，于是采用了最直接的id划分处理，来解决前面遇到的刻印孔冲突问题**。

enum {

/\*! max\_mintmark\_num \*/

max\_mintmark\_num = 1000,

// 属性刻印起始ID

attribute\_mintmark\_startid = 10001,

// 技能刻印起始ID

skill\_mintmark\_startid = 20001,

// all\_function刻印起始ID

all\_function\_mintmark\_startid = 40001,

max\_mintmark\_id = all\_function\_mintmark\_startid + max\_mintmark\_num,

};

/\*\*

\* @brief 判断刻印孔中的记录，是否刻印获得时间

\*

\* @param mintmark\_id 刻印id或刻印获得时间

\*

\* @return

\*/inline bool is\_mintmark\_obtain\_time(uint32\_t mintmark\_id)

{

return (mintmark\_id > max\_mintmark\_id);

}

#### 代码梳理&联调

在完成文案开发的过程中，发现旧刻印不是做的独立功能设计，所以函数和定义比较散

乱，特效实现也是在战斗流程中的随机插入，没有统一框架。在完成大部分新功能开发时，对代码进行了简单梳理，把所有相关函数搬迁到mintmark.cpp文件中，枚举类型搬迁到服务共享目录下mintmark\_common.hpp头文件中，特效部分复制了技能特效的注册框架，刻印和宝石在online初始化时传入特效id和参数即可，没有走旧刻印的实现方法，也避免重新实现一套特效框架。

#### 提交测试&合并分支

完成联调和基本功能测试后进行合并分支操作，提交给测试和策划验收，svn合并步骤

比较多，按下面博客操作可以完成：

<http://www.sepcot.com/blog/2007/04/SVN-Merge-Branch-Trunk>

#### 辅助平台修改

辅助平台主要是内外网测试工具和客服平台修改，在全能刻印完整开发完成后，2周

内陆续完成。

## 遇到的问题小结

1. 最初是对新旧刻印都做了全缓存设计，一次性拉取，没有考虑到用户的刻印数量上限，发布到线上后部分用户出现无法登陆情况，经过检查是包长超出online和dbprxoy限制被丢弃，后面改成了分页拉取只缓存新刻印，问题解决。
2. 发布后出现过绑定精灵id不对，修复版本发布后还是没有刷新刻印绑定的精灵id，检查发现是旧刻印定义的monster\_id与新全能刻印定义的绑定记录字段bind\_monid引起混淆，错记录到monster\_id上，确认问题后修正。
3. 宝石特效这块存在硬编码逻辑，主要是宝石数量限定为1，绑定特效也对应为1，由于策划明确镶嵌宝石数量不是需求目标，所以这里短期内没有优化安排。

# 赛尔号PVP对战流程(作者 krischien)



# 花仙PVE游戏开发思想(作者 kevinzhang)

**花仙pve游戏是吸引玩家的主要玩法之一，几乎每周活动都会有一个（以上）PVE小游戏来吸引用户参与活动，通过活动可以获得本周的奖励（代币），代币积累可以兑换特殊的物品；**

PVE游戏实现的UML类结构如下图所示：



具体的实现：

**1.定义游戏管理器类SingleGameManager（**定义为：sSingleGameM**）**

该类用一个数据成员用来管理PVE游戏，实现了玩家与游戏的一个映射关系。

数据结构：map容器（定义为m\_single\_games）

其中key为米米号；value为游戏基类指针；

PS: 该容器添加元素，是在游戏开始时，new出游戏子类SingleGame1，并将其地址交给SingleGameBase的指针p\_game，再将其以 id号-p\_game键值对形式放入m\_single\_games中；

1. **定义游戏基类SingleGameBase**

定义该基类是为了使用c++的多态性质；主要定义 “游戏开始”，“游戏结束”这两个虚函数（我们约定“游戏开始”函数名为game\_start，“游戏结束”函数名为game\_finish）

PS: 其中1,2两步骤已经固定，新人开发时不需要重新实现，只需要根据每周的策划案的PVE游戏的数量来定义相关的子类，详见3步骤

1. **定义本周活动的PVE游戏子类,SingleGame1**

游戏子类中，主要是重定义基类中的game\_start，game\_finish，两个虚函数；

至于为何要用多态的方式，主要源于策划案的灵活多变；这里以一个例子来说明：

**比如**：

SingleGame1的PVE游戏由五个小关卡组成，策划要求是根据胜利这个条件对关卡依序解锁。那么在game\_start中，应该保存当前玩家的关卡（不妨定义uint32\_t m\_card\_id的数据成员，具体关卡数值是由前端给出，同时在游戏中根据胜利这个条件来更新当前待解锁的关卡m\_locked\_id；另：实质上game\_start是在游戏开始协议对应的协议处理函数中被调用的；game\_finish是在游戏结束时前端请求游戏结束协议对应的处理函数中被调用）

在game\_finish中，若满足以下两个条件：

1）游戏胜利；

2）当前所玩的关卡正好是待解锁的前一关，即（m\_card\_id + 1 == m\_locked\_id）

则待解锁的关卡被解锁，同时将m\_locked\_id更新并持久化到DB中。

由此可见，使用多态的思想，在PVE游戏中是比较实用；另：在game\_finish中进行游戏结算发奖励，那么策划设定的奖励策略更新灵活多变，这同时也说明，每个游戏子类中，需要重新定义自己的game\_finish；

Game\_finish函数的最后，都要统一调用SingleGameBase::game\_end(pPlayer);以detele掉SingleGame1，目的是释放内存

至此，PVE游戏的开发工作就完成了，玩家在进入游戏时，前端会请求“游戏开始”的协议，此时，根据玩家指针与前端传来游戏的id，并依据简单工厂的设计模式，来new出游戏的子类；并将子类的地址赋值给游戏基类的指针，目的是通过基类指针来实现多态的虚函数调用； 其中，游戏的id是开发时，后台与前端通过协商自行定义，不同的游戏子类id号不能相同。

在“游戏开始”协议中，调用游戏子类的game\_start

在“游戏结束”协议中，调用游戏子类的game\_finish

**附：**PVE游戏中，也引入了信号的思想；通过定义信号（player类中定义），并注册信号以及定义相应的处理函数，来实现一些清理内存的工作；比如当玩家进行非正常游戏流程操作时（比如游戏中途关闭浏览器，这时会析构掉该玩家的player对象）在player析构函数中，应该触发信号，调用相应的信号处理函数，delete掉SingleGame1，从而调用游戏基类**SingleGameBase**的析构函数，从而释放内存，避免玩家退出时，但游戏内对象还存在而引起的内存泄露；同时在**SingleGameBase**析构函数中触发另一个信号，（这个信号在游戏开始时注册的）在相应的处理函数中，将相应的元素从m\_single\_games中删除。