从零开始--多人麻将游戏开发

序言

记得大概是17年年底开始吧，第一次萌生了写些棋牌游戏开发入门和模板化的文章，然而由于工作和个人等方面的原因，一直拖拖拖，时光荏苒，今年总算开始了...

作为一个棋牌游戏开发的多年资深从业者，从游戏客户端做起，转战服务端，再到作为PM主导开发多款麻将、扑克等等棋牌和休闲游戏，亲身深度开发接触的麻将不下几十款。在当前的棋牌游戏实际开发过程中，往往浪费了大量的人力物力，做了太多重复的工作，从中小初创公司到成熟的上市企业，大多如此。棋牌游戏，特别是麻将扑克类有非常多的共通点，很多时候只需要我们稍微转换下思路，跳脱下思维，经常能达到事半功倍的效果。

本文麻将设计将尽量抽象化模块化麻将类游戏开发， 一方面方便小白用户最快时间内从零开始学会麻将后端开发，另一方面最大化封装设计模块，在读者学会该款麻将后，后续最为快速比如一周甚至几天内出一款全新的地方麻将代码，大大缩短麻将的开发和测试周期。

通篇的写作思路大体在于引导读者读懂后端麻将开发，在讲解麻将开发设计思路的同时，将新项目入门、理解、跟踪、调试的技巧融杂其中，最大化接近开发者进入企业中实际学习项目的方式。

出于项目开发周期和工作效率考虑，麻将后端基于python版本进行实现，希望读者有一定的python编码基础（需要能看懂普通python语言代码，熟悉python语法规则），数据库基于mysql进行存储。限于作者水平所限，读者发现有可优化改进之处，欢迎随时指正~

一、游戏简介

1. 游戏学习目标

1）掌握如何做一款麻将

自2000年Python第一个稳定的2.7版本发布以来，Python"简单、明确、优雅"设计哲学的根本出发点就决定了Python这门语言的编程易用性和工作高效性。特别是近5年来，在游戏行业中传统的编程语言c++开发及维护成本极高的劣势逐渐凸显，同样的业务需求实现，可能传统编程语言c++程序猿需要1周，而一个普通的Python程序猿可能三天就能完成，同时后续隐藏的bug还会少很多。于是在休闲、棋牌游戏领域，Python这门高效的脚本语言开始逐渐流行起来。

作为棋牌游戏中复杂度最高的游戏，麻将游戏开发游戏逻辑复杂多变，据不完全统计，全国地方麻将的种类已经超过上千种，不同的地方麻将差异巨大，比如有的没有"万"，有的没有"饼"，有的存在"亮倒"等操作，胡牌的类型更是千变万化，一些胡牌类型甚至有些匪夷所思(如绿一色：即手牌全是绿色的牌，一种东北地方麻将的大胡胡法)。

这门课程一个大的目的是希望大家通过十天的学习，能够掌握如何做一款国标或地方麻将，理解棋牌游戏开发详细流程及实现方式，让有意向的学生学会游戏后端开发技能，具备找到游戏后端开发的专业能力。

2）学会企业中如何快速理解、修改一个新项目

大家入门之后，进到具体的某一个公司之后，通常就职的是一个初级岗位。具体在公司入职之后，一般第一天装开发环境，第二天开始扔给员工一个项目，让他熟悉代码，然后开始分配一些小任务，例如修改bug，添加一些小功能等。通常给新员工的新项目理解周期不会太长，大多1-2周；此时经常会有很多新员工会觉得鸭梨山大，难以应对。

在此希望这门游戏课程的学习能够教会大家如何快速理解一个新项目，同时可以迅速添加实现一些新的业务逻辑需求功能。

2. 演示一局麻将运行

1）登录、大厅、房间坐桌、游戏、胡牌简单演示

大厅界面

牌局界面


12.34万
过1次
剩余20张
x2
08
听牌
左
x2
初级场 底分50
12.34万

系统提示：


请游戏结束后再退出房间！
6520
二人麻将
08
剩余45张
初级场 底分:500
6520

邮件：

2）麻将后端代码快速概览

game server: echecs

web service: echecs_web_services

client: tpmj

3. web开发、游戏开发、App开发比较

在wiki上对Server的分类：

Typical servers are database servers, file servers, mail servers, print servers, web servers, game servers, and application servers.

其他几种Server我们都比较清楚了，跟unix差不多同时诞生。接下来我们主要针对web servers, game servers, and application servers进行简单介绍。

1）web server

典型例子是淘宝。

特点：所有流程均由客户端发起，客户端发个请求，服务端返回个响应。而且，根据客户端访问的服务不同，客户端可以向不同的具体服务端节点发起请求。

2）game server

典型例子是王者荣耀。

特点：有一个肉眼能感觉到的连接握手的过程，建立连接后，流程有可能是服务端发起（比如给你展示周边玩家），也有可能是客户端发起（比如你移动了一下）。

同时，如果你手边有抓包工具，可以看到，如果你选中了某个玩家，在该玩家的头像框消失之前，一直是同一个场景服务器在跟你通信。

3) app server

典型例子是QQ。

特点：介于Web Server和Game Server之间，看着像一个web服务器，但是又有游戏服务器的特点。

4）共同点

都是为客户端提供多种服务

都需要连接会话的概念

服务端的每台物理机服务多个客户端

都具有分布式结构

5) 不同点

a. 会话的存在形式:

这一点是web服务端与游戏服务端最本质的区别。

web服务端的客户端与客户端之间交互非常有限，因此，服务端可以将会话保存在外部存储服务，比如一些缓存中间件、文件系统中间件，然后等再用到的时候再拿出来就可以了。

而游戏服务端的客户端与客户端之间交互非常频繁，比如，同场景的其他玩家会不停做不规律移动，战斗时一个技能就会对复数个玩家造成影响。

这时如果将会话状态保存在外部，会造成频繁的状态存取，严重影响服务器吞吐量。因此对于游戏服务端来说，会话通常保存在进程内。

b. 交互频率与数据流向:

web服务端的频率低，而且数据的流动是由客户端驱动的，流向通常是客户端请求了，服务端才返回。

而游戏服务端的频率高，数据的流动一部分由客户端驱动，一部分由服务端驱动。流向除了服务端对客户端请求的响应，还有服务端的主动推送。

c. 通信协议基础:

web通信的基础在应用层是http协议。

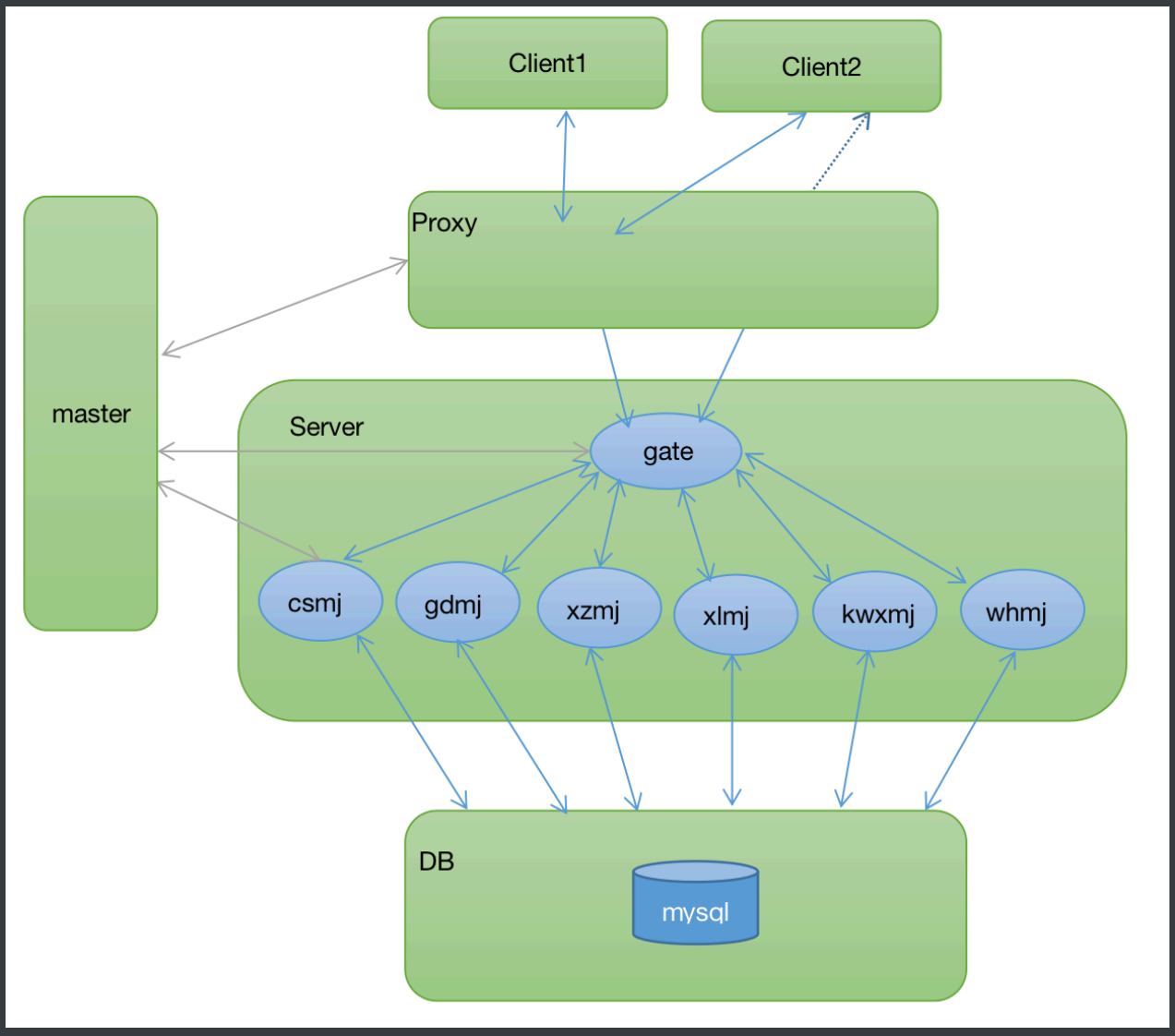
游戏通常会实现私有的序列化协议，可以简单理解为应用层定义协议包结构平铺成字节流或者是串行序列化字节流。如果要支持一定程度的协议版本兼容，会用二进制json或者protobuf来实现协议序列化，但是通信协议本身是没有「基础」可言的，纯私有化协议，不具普适性，也没有必要定义成一种专门的协议。

4. firefly游戏框架介绍

1）框架架构及通信流程

Firefly是免费、开源、稳定、快速扩展、能 “热更新”的分布式游戏服务器端框架，采用Python编写，基于Twisted框架开发。

在麻将游戏中，应用firefly框架后的游戏总体架构图如下：



client： 客户端，即玩家用户，游戏中客户端和服务端之间的连接是长连接，客户端和服务端的proxy节点进行连接；

proxy：服务端的代理节点，其主要任务是负责消息打包和解包，加解密，然后将合法的消息转发向后端节点。proxy节点地址对公网开发(客户端通过域名和端口连接)；

gate: 服务端的消息分发节点（门户节点）。该节点根据消息请求id将不同的消息分发到不同的子节点中进行处理，如长沙麻将分发到长沙麻将的游戏节点处理，广东麻将分发到广东麻将节点处理，该类型节点一般不对公网开放；

游戏节点，各类型的游戏，如csmj(长沙麻将)，gdmj(广东麻将), xzmj(血战)等。此类节点为游戏主逻辑节点。

master：firefly框架中的管理节点，它负责管理所有的proxy，gate，游戏节点等，主要管理节点的加入和退出，不负责具体业务逻辑；

DB模块：游戏中所有涉及到数据库的部分，各节点皆有可能操作；

二、麻将产品需求及流程介绍

1. 麻将产品需求文档

1) 麻将术语

a. 名词术语

牌子：

序数牌：一万，二万，三万，…，九万，一筒，二筒，…，九筒，一条 ---- 九条

字牌：字牌又分为风牌和箭牌

风牌： 东南西北

箭牌：中发白

花牌：春夏秋冬梅兰竹菊

幺牌：1、9、字牌，统称幺牌

刻子：三张相同的牌

箭刻：中发白的刻子

风刻：东南西北的刻子

明刻：碰的牌为明刻

暗刻： 自己摸上来的为暗刻

幺九刻：1、9或字牌的刻子成为幺九刻

对子：

普通对子：两张相同的牌

将牌：将规定牌型胡牌时必须具备的单独组合的对子

顺子：一般为三张同花色序数相连的牌，如一万，二万，三万

手牌：一般标准数为十三张。包括摆亮在门前的刻子、杠；未亮明的手牌为立牌

庄家、闲家：每局中一人为庄家，其他的未闲家

盘：每次从起牌到胡牌或流局为一盘

轮：行牌一周为一轮

牌墩：2个叠在一起的麻将

牌墙：2人各自在门前码的墩牌

b. 动作术语

吃牌：

正常吃牌：上家打出一张牌后，本家打出两张牌，与上家的牌组成一个顺子为吃牌。吃牌只能吃上家的。

胡牌吃牌：任意玩家打出一张牌，本家打出两张牌，与玩家的牌组成一个顺子并且能胡牌，胡牌时吃牌不限于上家。

碰牌：任一家打出牌后，报碰牌者把自己的对子取出，加在一起组成一副刻子摆亮在立牌前。

杠牌：

明杠：上家打牌之后，如果你手上有三张跟那被打出的牌一样的，就可以“杠牌”，这种杠牌叫做“明杠”。

暗杠：如果是手内摸有四张相同的牌，取出杠牌，则叫做“暗杠”。

加杠/补杠：如果是已经碰牌了，却又再摸入一张相同的牌，也可以叫杠牌—这种杠牌叫做“加杠”

补张：杠牌后，需要补张，即从牌墙的末尾端摸一张牌放入手牌中

补花：摸到花牌时，将花牌放一旁，再进行补张

听牌：一盘中玩家只差所需的一张牌技能胡牌的状态

胡牌：摸到符合规定的牌型条件。最终形成指定牌型，如四个顺子或刻子组合加一对将牌

自摸：所胡的牌为自己正常莫得，即称之为自摸胡，简称自摸

点炮：所胡的牌为别人打出来的牌，成为点炮

流局：摸完所有牌后都无人胡牌即称为流局。如出现流局庄家继续坐庄

2) 基本规则

此处以二人麻将基本规则为例：

a. 游戏人数：2人

b. 牌数

共72张牌，包括：

万字牌，一万至九万，各四张，共36张

风字牌，东南西北各4张，共16张

箭牌，中发白各4张，共12张

花牌：春夏秋冬梅兰竹菊各一张

c. 定庄

第一局庄家：匹配房随机庄家；好友房房主坐庄

第二盘起，上盘谁胡牌，下盘谁坐庄

流局则庄家继续坐庄

抢杠胡，被抢杠的玩家下局当庄(输家当庄)

d. 摸牌

起手摸牌：游戏一开始，庄家可得到14张牌，闲家13张，庄家先出

局内摸牌：玩家打出一张牌后，无人响应在，则下家从牌堆的起始处摸一张牌

补张：杠牌或摸到花牌后，从牌堆的末尾处摸一张牌

e. 补花

局内补花：当玩家摸到花牌时，展示花牌，并从手牌中移出，然后进行补张，如下一张也是花牌，重复此动作，直到摸到非花牌

起手摸牌阶段补花：从庄家开始依次补花，即庄家先补，如果补上来的牌也是花牌，需要等其他玩家都补过以后，庄家才可以补，其他玩家动作和庄家一样

f. 吃碰杠

吃牌和碰牌后，玩家需要打出一张牌

杠牌中的暗杠、补杠，玩家摸到后不强制杠牌，玩家可以选择过，在以后的回合开始后选择杠。

杠牌的玩家需要从牌堆末尾摸一张牌，再打出一张牌

g. 听牌

胡牌提示：玩家再打出一张牌即进入听牌状态时，给玩家的一个提示；玩家选中要打出的牌时，提示玩家能胡哪几张牌及其剩余的张数及番型。

报听：告诉对手自己已经进入听牌状态，界面上有展示，报听的玩家不能展示自己的手牌，摸啥打啥

天听：摸完牌，庄家打出一张牌时即报听、闲家摸到第一张牌时即报听且报听后必须打出第一张摸来的牌，为天听

选择听牌后，将对方玩家的手牌展示出来使该听牌玩家可以看到，此时未听牌的玩家不能看到对方手牌

h. 胡牌

在第几场中6番起胡，中级场中10番起胡，高级场中12番起胡

3) 特殊规则

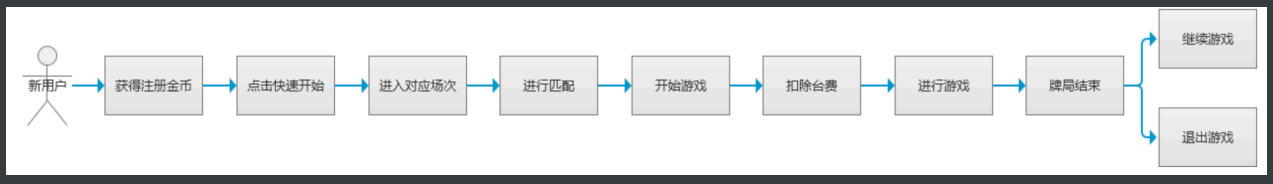
过胡加倍：

听牌的玩家在触发胡牌时，可以选择过胡，过胡的玩家胡牌时，结算的分数翻倍；

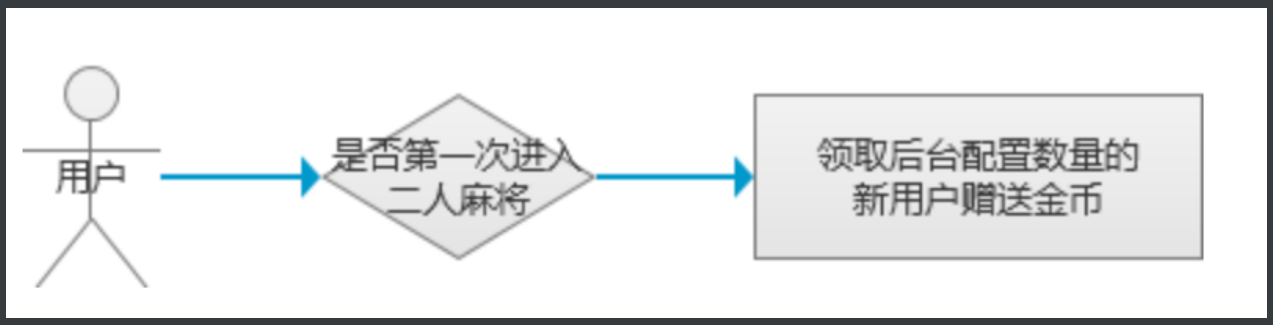
结算时，得分为2的n次幂，即过胡一次为2的1次幂，过胡2次，则为2的2次幂。

2. 麻将游戏流程

1) 新用户游戏流程

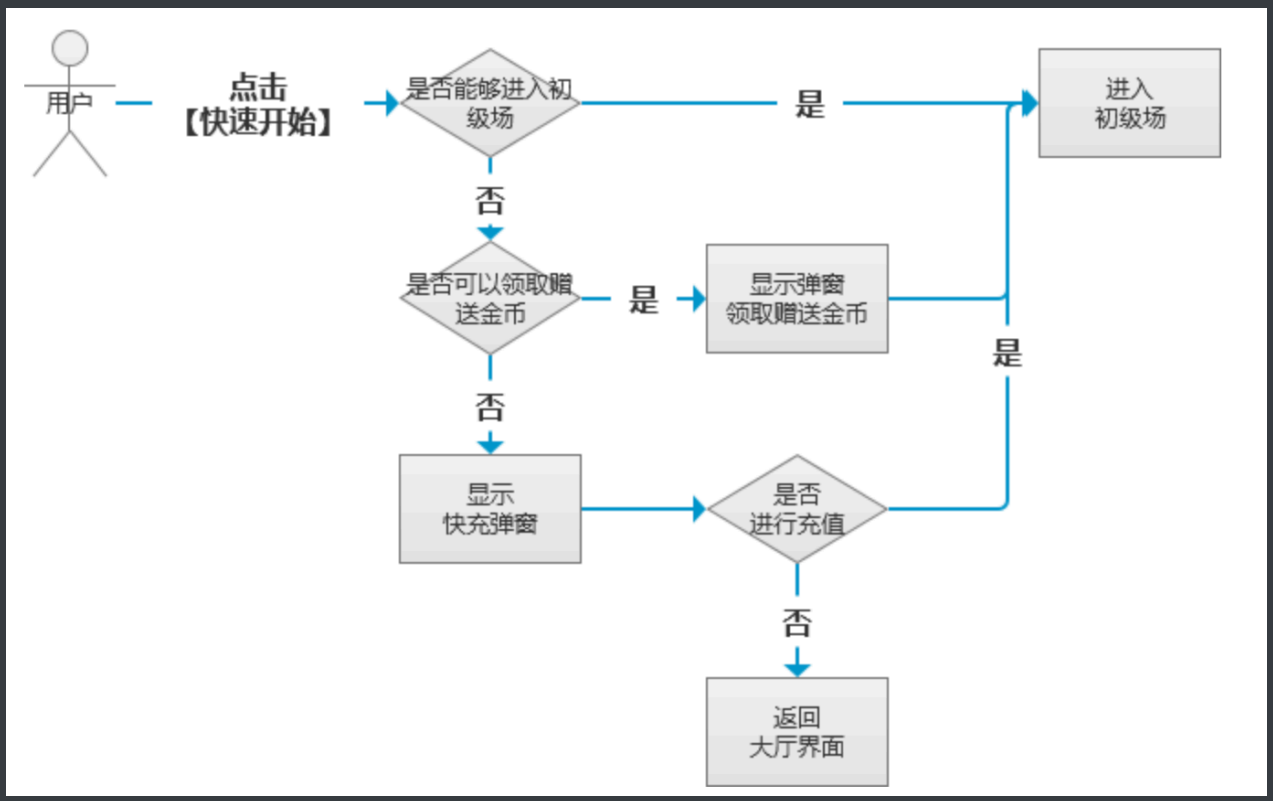


2) 新用户领取注册金币流程

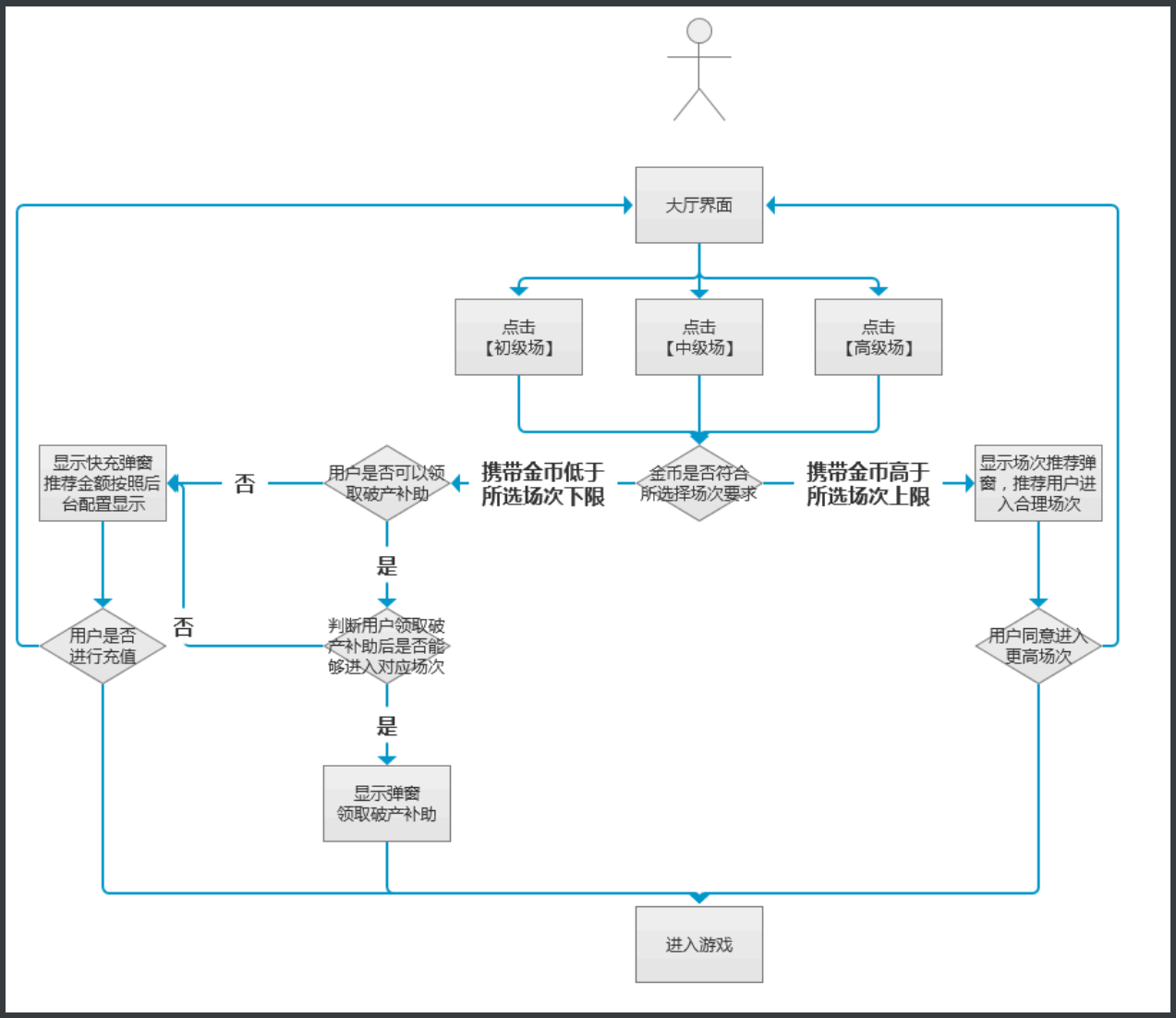


3) 用户领取破产补助流程

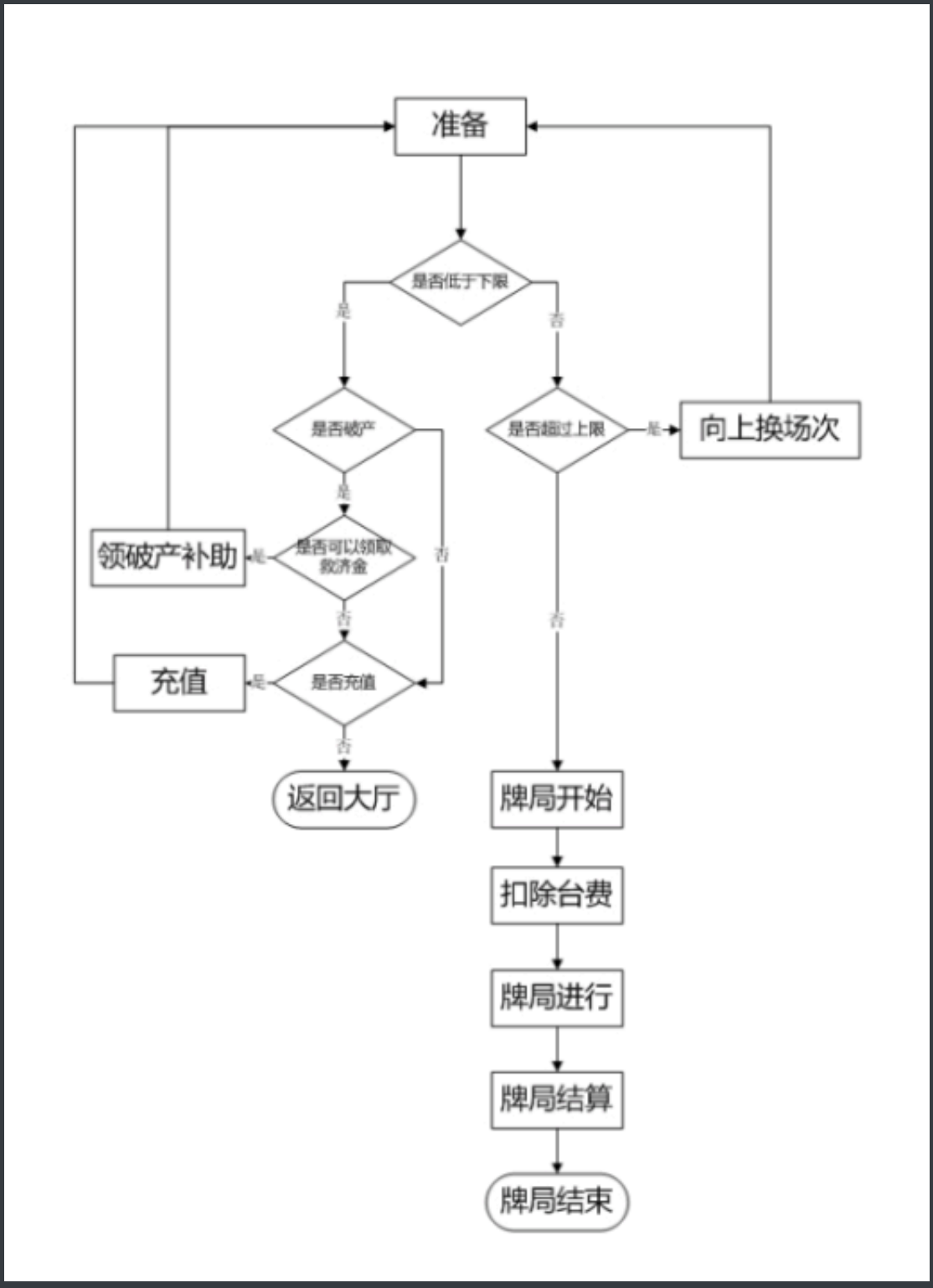
4) 快速开始流程



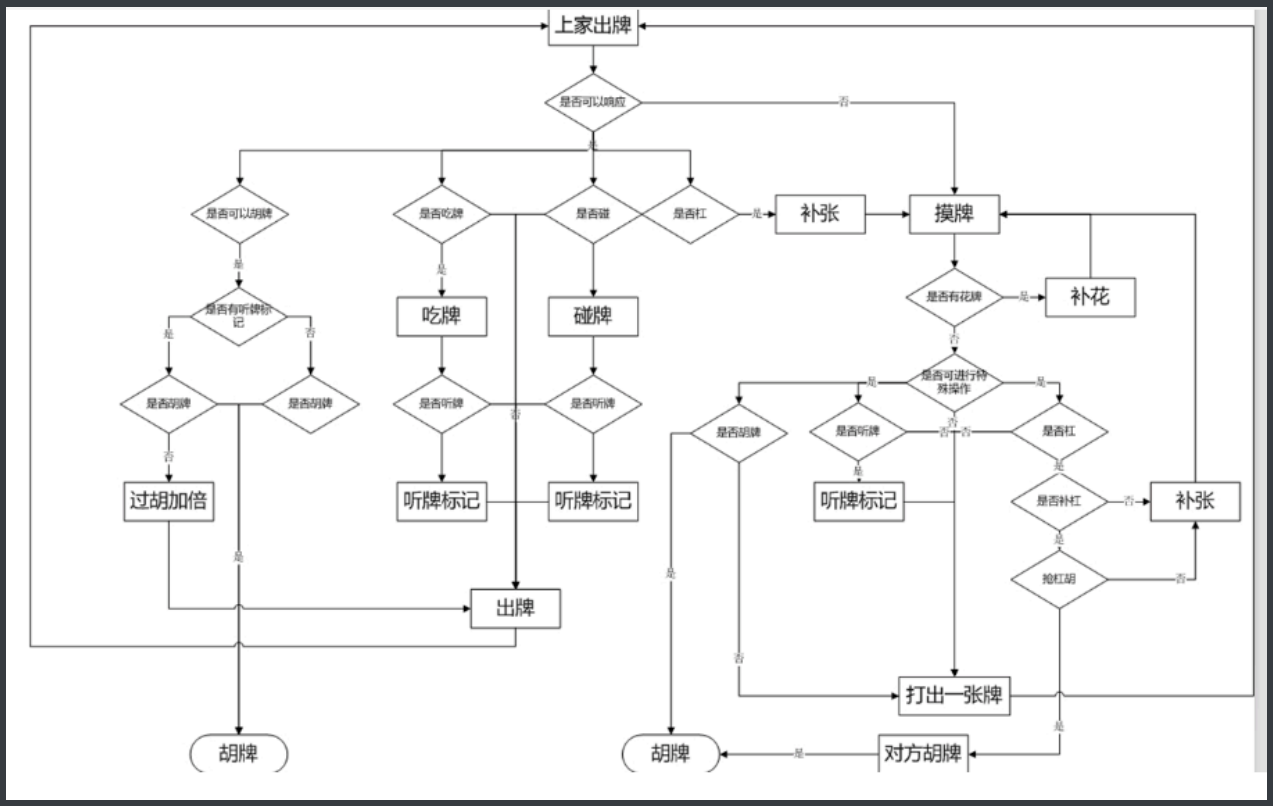
5) 初级场、中级场、高级场逻辑流程


大厅界面
点击
【初级场】
点击
【中级场】
点击
【高级场】
金币是否符合
所选择场次要求
携带金币低于
所选场次下限
携带金币高于
所选场次上限
用户是否可以领
取破产补助
否
显示快充弹窗
推荐金额按照后
台配置显示
是
判断用户领取破
产补助后是否能
够进入对应场次
否
用户是否
进行充值
是
显示弹窗
领取破产补助
显示场次推荐弹
窗，推荐用户进
入合理场次
用户同意进入
更高场次
进入游戏

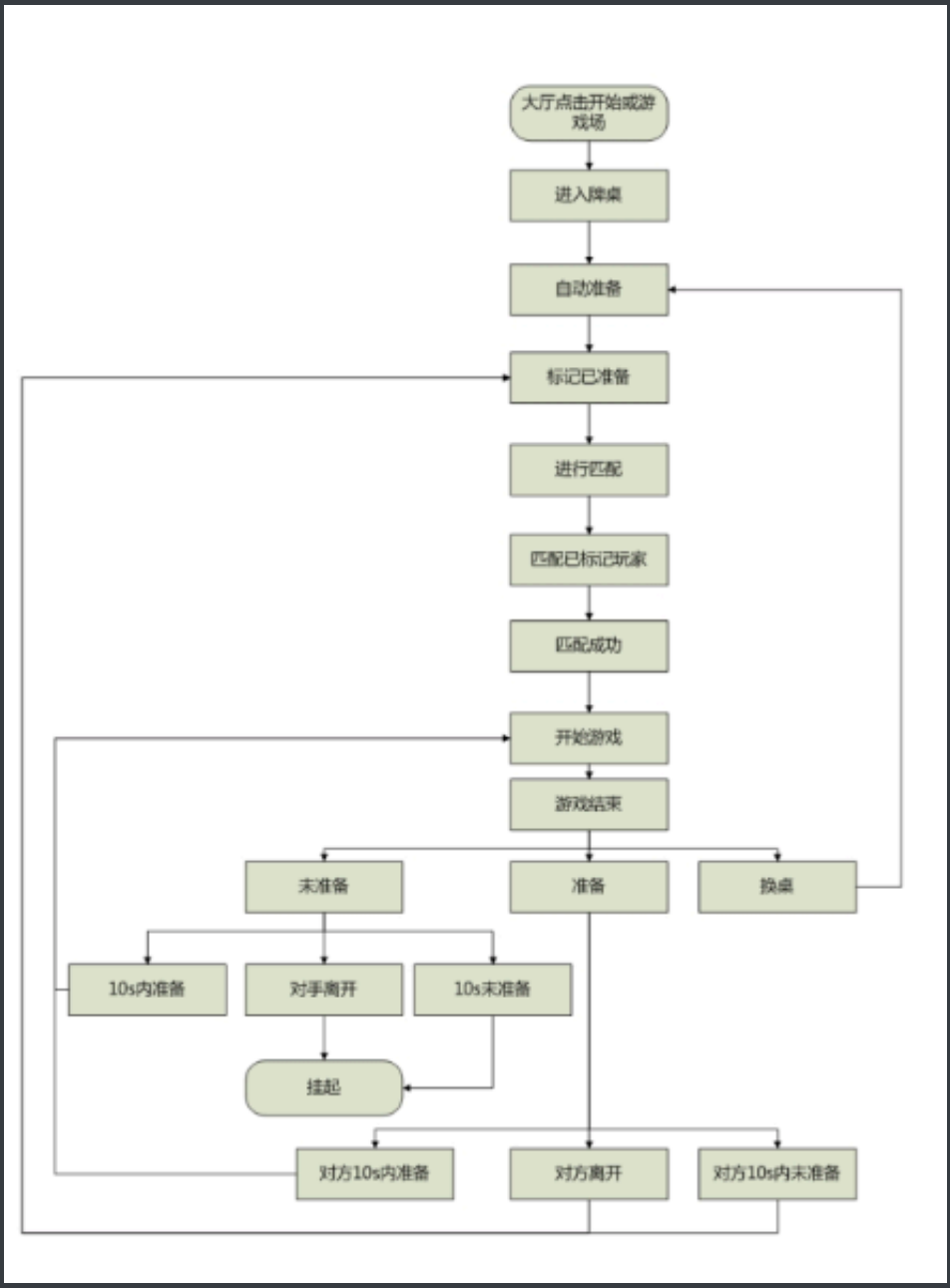
6) 金币流向流程图


准备
是否低于下限
是
否
是否破产
是否超过上限
是
向上换场次
是
领破产补助
是否可以领取救济金
是
否
否
充值
是
是否充值
否
返回大厅
牌局开始
扣除台费
牌局进行
牌局结算
牌局结束

7) 牌局进行流程图



8）牌局结束后再匹配流程图



三、游戏加解密及通信

1. 日志系统

在具体的游戏开发过程中，我们经常需要输出一些辅助信息，方便以后可能的运维跟踪及数据统计；此时传统的print函数一方面过于简陋，同时又灵活度不够，而Python自带的logging模块本身功能也相对比较原始，在实际的上线运维过程中，很多时候我们希望日志的一些warning和error信息能够及时上报出来，方便运维人员监控，快速发展异常及潜在的安全隐患。

在该项目中重新封装了日志输出的logger模块，该封装的日志模块主要使用了以下几种策略：

a. 按天自动进行日志切分，比如2019-07-24分一个日志文件，07-25再分一个文件

b. 同时根据文件大小做二次自动切分，日志文件每到30M则自动切分成一个新文件

c. warning和error的错误日志会定向到单独的文件，方便运维监控及预警

d. debug级别日志会同时print一份，以便开发人员自身单元测试时更快速的根据日志定位问题

（share/espoirlog.py）

```
# coding=utf-8

"""

"""

__all__ = [
    'getLogger',
    'DEBUG',
    'INFO',
    'WARN',
    'ERROR',
    'FATAL']

import time
import logging
import logging.handlers
import os
from logging import getLogger, INFO, WARN, DEBUG, ERROR, FATAL, WARNING,
CRITICAL
from firefly.server.globalobject import GlobalObject

LOG_FILE_MAX_BYTES = 31457280
LOG_FILE_BACKUP_COUNT = 1000
LOG_LEVEL = logging.DEBUG

MODULE_NAME = GlobalObject().json_config.get("name", '')
LOG_DIR = GlobalObject().json_config.get("log_dir", './logs')
if not os.path.exists(LOG_DIR):
    os.system("mkdir -p %s" % LOG_DIR)

# DATE_FORMAT = time.strftime('%Y-%m-%d', time.localtime(time.time()))

FORMAT = '[%(asctime)s]-%(levelname)-8s<%(name)s> {%(filename)s:%(lineno)s}
-> %(message)s'
formatter = logging.Formatter(FORMAT)
```

```
def get_normal_log(module_name, date_format):
    file_name = '{0}/{1}_{2}.log'.format(LOG_DIR, module_name, date_format)
    normal_handler = logging.handlers.RotatingFileHandler(file_name,
maxBytes=LOG_FILE_MAX_BYTES,backupCount=LOG_FILE_BACKUP_COUNT)
    normal_handler.setFormatter(formatter)
    normal_log = getLogger(module_name)
    normal_log.setLevel(LOG_LEVEL)
    normal_log.addHandler(normal_handler)
    return normal_log


def get_error_log(module_name, date_format):
    file_name = '{0}/ERROR_{1}_{2}.log'.format(LOG_DIR, module_name,
date_format)
    error_handler = logging.handlers.RotatingFileHandler(file_name,
maxBytes=LOG_FILE_MAX_BYTES,backupCount=LOG_FILE_BACKUP_COUNT)
    error_handler.setFormatter(formatter)
    error_log = getLogger(module_name+'_error')
    error_log.setLevel(LOG_LEVEL)
    error_log.addHandler(error_handler)
    return error_log

class EspoirLog(object):

    def __init__(self, module_name):
        self._normal = None
        self._error = None
        self.name = module_name
        self.last_date = time.strftime('%Y-%m-%d',
time.localtime(time.time()))   # 存放上一次打印日志的时间(字符串)

    @property
    def normal_log(self):
        cur_date = time.strftime('%Y-%m-%d', time.localtime(time.time()))
        if not self._normal or self.last_date != cur_date:
            self.last_date = cur_date
            self._normal = get_normal_log(self.name, self.last_date)
        return self._normal

    @property
    def error_log(self):
        cur_date = time.strftime('%Y-%m-%d', time.localtime(time.time()))
        if not self._error or self.last_date != cur_date:
```

```
            self.last_date = cur_date
            self._error = get_error_log(self.name, self.last_date)
        return self._error

    def set_name(self, name):
        self.name = name

    def setLevel(self, level):
        self.normal_log.setLevel(level)

    def _backup_print(self, msg, *args, **kwargs):
        if args:
            msg = "{0}/{1}".format(msg, str(args))
        if kwargs:
            msg = "{0}/{1}".format(msg, str(kwargs))
        print(msg)

    def debug(self, msg, *args, **kwargs):
        if self.normal_log.isEnabledFor(DEBUG):
            self.normal_log._log(DEBUG, msg, args, **kwargs)
            self._backup_print(msg, args, kwargs)

    def info(self, msg, *args, **kwargs):
        if self.normal_log.isEnabledFor(INFO):
            self.normal_log._log(INFO, msg, args, **kwargs)
            self._backup_print(msg, args, kwargs)

    def warning(self, msg, *args, **kwargs):
        if self.normal_log.isEnabledFor(WARN):
            self.normal_log._log(WARNING, msg, args, **kwargs)

    def warn(self, msg, *args, **kwargs):
        if self.normal_log.isEnabledFor(WARN):
            self.normal_log._log(WARN, msg, args, **kwargs)

    def error(self, msg, *args, **kwargs):
        if self.error_log.isEnabledFor(ERROR):
            self.normal_log._log(ERROR, msg, args, **kwargs)
            self.error_log._log(ERROR, msg, args, **kwargs)
        print(msg, args, kwargs)

    def critical(self, msg, *args, **kwargs):
        if self.error_log.isEnabledFor(CRITICAL):
            self.normal_log._log(CRITICAL, msg, args, **kwargs)
            self.error_log._log(CRITICAL, msg, args, **kwargs)
```

```
        def fatal(self, msg, *args, **kwargs):
            if self.error_log.isEnabledFor(FATAL):
                self.normal_log._log(FATAL, msg, args, **kwargs)
                self.error_log._log(FATAL, msg, args, **kwargs)


    logger = EspoirLog(MODULE_NAME)
```

2. 带json串解析

因为json串便捷及易于理解等特性，很多时候我们会用json文件作为配置文件，在文件配置较少各配置命名规范时，不会出现什么问题，但当有些时候命名较为随意或者配置项太多经常出现重名时，此时配置文件的理解容易让人费解或者产生歧义，添加注释不失为一种很好的解决方法。

Python中三方库json或ujson都不支持带注释的json串解析，故我们自己封装了一套带注释（“//”）的json串解析库。大家以后在企业工作中可以直接copy过去。（下面代码中兼容了Python3和Python2两个版本）

（share/espoirjson.py）

```
    # coding=utf-8

    import json
    import re
    import sys


    class TrimNote(object):
        """
        创建一个TrimNote类，用于处理从文件中读出的字符串
        """
        def __init__(self, instr):
            self.instr = instr

        # 删除“//”标志后的注释
        def rmCmt(self):
            qtCnt = cmtPos = slashPos = 0
            rearLine = self.instr
            # rearline: 前一个“//”之后的字符串，
            # 双引号里的“//”不是注释标志，所以遇到这种情况，仍需继续查找后续的“//”
            while rearLine.find('//') >= 0: # 查找“//”
                slashPos = rearLine.find('//')
                cmtPos += slashPos
```

```
                # print 'slashPos: ' + str(slashPos)
                headLine = rearLine[:slashPos]
                while headLine.find('"') >= 0: # 查找“//”前的双引号
                    qtPos = headLine.find('"')
                    if not self.isEscapeOpr(headLine[:qtPos]): # 如果双引号没有被
转义
                        qtCnt += 1 # 双引号的数量加1
                    headLine = headLine[qtPos+1:]
                    # print qtCnt
                if qtCnt % 2 == 0: # 如果双引号的数量为偶数，则说明“//”是注释标志
                    # print self.instr[:cmtPos]
                    return self.instr[:cmtPos]
                rearLine = rearLine[slashPos+2:]
                # print rearLine
                cmtPos += 2
            # print self.instr
            return self.instr

    # 判断是否为转义字符
    def isEscapeOpr(self, instr):
        if len(instr) <= 0:
            return False
        cnt = 0
        while instr[-1] == '\\':
            cnt += 1
            instr = instr[:-1]
        if cnt % 2 == 1:
            return True
        else:
            return False


class ParseJson(object):
    """
    解析带有//注释的json文件
    """
    @classmethod
    def loads(cls, json_path):
        try:
            if 3 == sys.version_info.major:
                srcJson = open(json_path, 'r', encoding='UTF-8')
            else:
                srcJson = open(json_path, 'r')
        except Exception as e:
            print('cannot open ' + json_path)
```

```
                quit()

            dstJsonStr = ''
            for line in srcJson.readlines():
                if not re.match(r'\s*//', line) and not re.match(r'\s*\n',
    line):
                    xline = TrimNote(line)
                    dstJsonStr += xline.rmCmt()

            dstJson = {}
            try:
                dstJson = json.loads(dstJsonStr)
                return dstJson
            except:
                print(json_path + ' is not a valid json file')
                quit()


    if __name__ == "__main__":
        print(ParseJson.loads("config2.json"))
```

3. 协议打包解包

前文讲到游戏中通信一般都采用纯私有化协议。在服务端和客户端之间发送数据时，根据指定的消息格式对消息进行打包，接收数据时进行数据解包。

消息的打包解包使用Python中的struct模块，struct模块的常见的格式符意义如下：

Format	C Type	Python	字节数
x	pad byte	no value	1
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer or long	4

l	long	integer	4
L	unsigned long	long	4
q	long long	long	8
Q	unsigned long long	long	8
f	float	float	4
d	double	float	8
s	char[]	string	1
p	char[]	string	1
P	void *	long	

在该麻将游戏中，消息格式使用的是

（proxy/protocol.py）

```
class DataProtocol:

    def __init__(self):
        self.handfrt = "iii"   # 第一个i表示消息长度，第二个i表示消息ID，第三个i代表协议版本号
```

消息打包：

先对消息正文内容加密，然后按照指定格式开始打包，核心代码如下：

```
def _pack(self, data, command_id):
    """
    打包消息，  用于传输
    :param data:  传输数据
    :param command_id:  消息ID
    :return: string
    """
    data = self.aes_ins.encode(data)
    data = "%s" % data
    print "pack data=", len(data), [data]
    length = data.__len__() + self._get_head_len()
    head = struct.pack(self.handfrt, length, command_id, self.version)
    return str(head + data)
```

消息解包：

先解出消息头，然后解密消息正文，核心代码如下：

```
    def _unpack(self, pack_data):
      """
      消息解包
      :param pack_data: 待解包的数据内容
      :return: dict, like {"result":True/False, "command": int(command id),
    "data": string}
      """
        head_len = self._get_head_len()
        if head_len > len(pack_data):
            return None

        data_head = pack_data[0:head_len]
        print "unpack data_head=", [data_head]
        list_head = struct.unpack(self.handfrt, data_head)
        print "unpack list_head:", list_head
        data = pack_data[head_len:]
        result = self.aes_ins.decode_aes(data)
        # result = data
        if not result:
            result = {}
        return {'result': True, 'command': list_head[1], 'data': result}
```

4. 消息加密和解密

在游戏产业中，经常容易受到各类非法玩家或者黑客的抓包，然后伪造消息破坏游戏的正常运行，因此一般都会对传输的内容进行加密。该游戏中使用的是AES进行加密。

游戏中针对于AES加解密封装了一个统一的模块，该模块具备以下特点：

a. 允许调用者选择是否进行AES加密或解密；

b. 秘钥长度必须为16字节的整数倍，否则会报错；

c. AES加密时会有一个对消息内容进行长度补位的操作，比如消息长度为57，则会在消息最后补为7个字符，补位的字符为7对应的字符；

d. AES解密时先对消息解密，解密后的明文需要根据加密时的补位规则将最后面的补位字符去掉；

部分代码（proxy/encodeutil.py）如下：

```
class AesEncoder:
    __metaclass__ = Singleton

    NONE_TYPE = 0
    AES_TYPE = 1

    def __init__(self, password='@ZYHD#GDMJ!112233!love**foreverX',
encode_type=0):
        self.aes_obj = AES.new(password, mode=AES.MODE_CFB)
        self.encode_type = encode_type

    def encode(self, msg):
        if self.encode_type == self.NONE_TYPE:
            return msg
        elif self.encode_type == self.AES_TYPE:
            return self.encode_aes(msg)

    def encode_aes(self, msg):
        return self.aes_obj.encrypt(pad(msg))
```

5. 启动解析

实际企业项目开发过程中，那么如何快速入手一个新项目呢?

这里给大家推荐一种方法和思路，从以下三个方面入手:

1）入口

首先找准程序入口，就像一堆零乱的绳子堆放在一起，我们现在要把绳子理顺，首先要做的是找到绳子的一端绳头。在项目中找绳头一般可以从项目的启动文件（一般在根目录下，名字带main或者start等）开始进行跟进，选择一处关键入口处（即所有消息都要经过的函数，方便运维跟踪和调试）

2）消息走向

找到绳头之后，选择一条项目消息进行跟踪，顺着该消息的请求处理流程（从路由转发到数据操作...）完整跟一遍，忽略不相干的模块及函数细节，从消息走向中理解代码的大体架构。

3）出口

顺着消息走向往下捋后，找到往客户端返回消息的地方。一般项目中消息接口会有很多个，选择其中一个进行跟踪，添加各类日志信息，确定消息的返回格式，使用客户端进行测试；

在该麻将游戏中，启动文件有start_mastersingle.py, start_proxy_1.py, start_gate_1.py, start_game_1.py中，这四个文件内容很类似。

start_mastersingle.py

该文件中会调用start_master.py文件

```
# coding=utf-8

import os


if __name__ == "__main__":
    command = "python " + os.getcwd() + "/" + "start_master.py single
master"
    os.system(command)
```

start_master.py，该文件中在Master对象中会调用appmain.py文件

```
# coding=utf8

import os
if os.name!='nt' and os.name!='posix':
    from twisted.internet import epollreactor
    epollreactor.install()

if __name__=="__main__":
    from firefly.master.master import Master
    master = Master()
    master.config('config.json','appmain.py')
    master.start()
```

start_proxy_1.py：

```
if __name__ == "__main__":
    command = "python " + os.getcwd() + "/" + "appmain.py proxy_1
config.json"
    os.system(command)
```

start_gate_1.py, start_game_1.py的启动和start_proxy_1类似，只有一个参数不一样，即proxy_1换成gate_1和game_1.

不难发现，所有的启动文件都会调用到appmain.py文件，那我们接下来看看appmain文件到底干了些什么：

```
# coding=utf8

import os

if os.name != 'nt' and os.name != 'posix':
    from twisted.internet import epollreactor
    epollreactor.install()      #

import json, sys
from firefly.server.server import FFServer

if __name__ == "__main__":
    args = sys.argv
    servername = None
    config = None
    if len(args) > 2:
        servername = args[1]
        config = json.load(open(args[2], 'r'))
    else:
        raise ValueError
    dbconf = config.get('db')
    memconf = config.get('memcached')
    sersconf = config.get('servers', {})
    masterconf = config.get('master', {})
    serconfig = sersconf.get(servername)
    ser = FFServer()       # 实例化服务
    ser.config(serconfig, servername=servername, dbconfig=dbconf,
memconfig=memconf, masterconf=masterconf)     # 传输配置
    ser.start()
```

其中FFServer为关键类，firefly的服务启动类，接下来我们再跳进FFServer的config函数

```
def config(self, config, servername=None, dbconfig=None,
                memconfig=None, masterconf=None):
        '''配置服务器
        '''
    # 此处省略了部分代码
    if netport:
        self.netfactory = LiberateFactory()
        netservice = services.CommandService("netservice")
        self.netfactory.addServiceChannel(netservice)
        reactor.listenTCP(netport,self.netfactory)
```

```
        if webport:
            self.webroot = vhost.NameVirtualHost()
            GlobalObject().webroot = self.webroot
            reactor.listenTCP(webport, DelaySite(self.webroot))

        if rootport:
            self.root = PBRoot()
            rootservice = services.Service("rootservice")
            self.root.addServiceChannel(rootservice)
            reactor.listenTCP(rootport, BilateralFactory(self.root))
        # 此处省略部分代码
```

可以看到在config中有调用reactor.listenTCP(...), 该部分根据配置的地址信息监听相应的端口，如rootport（rpc端口），webport(http web接口)， netport(websocket接口)， 其中reactor是twisted框架里面一个很重要的对象，它为我们实现了循环。

6. 配置文件

接下来我们看下后端核心的配置信息，firefly服务的配置文件内容如下：

```
{
    "master":{
         "rootport":10010,
         "webport":10009,
         "log":"logs/masterlog.log"
    },
    "servers":{
          "proxy_1":{
                "port": 10000,
                "webport": 10001,
                "name": "proxy_1",
                "app": "proxy.start_up",
                "remoteport":[
                  {"rootport": 11001, "rootname": "gate_1",
"is_available":1},
                  {"rootport": 11003, "rootname": "gate_2"}
                ]
          },
          "gate_1": {
                "rootport":11001,
                "webport":11002,
                "name":"gate_1",
                "app":"gate.start_up"
          },
```

```
            "gate_2": {
                  "rootport":11003,
                  "webport":11004,
                  "name":"gate_2",
                  "app":"gate.start_up"
            },
            "room_1":{
                  "rootport": 12001,
                  "webport":12002,
                  "name":"room_1",
                  "app":"game.start_up",
                  "remoteport":[
                    {"rootport": 11001, "rootname": "gate_1"},
                    {"rootport": 11003, "rootname": "gate_2"}
                  ]
            },
            "room_2":{
                  "rootport": 12003,
                  "webport":12004,
                  "name":"room_2",
                  "app":"game.start_up",
                  "remoteport":[
                    {"rootport": 11001, "rootname": "gate_1"},
                    {"rootport": 11003, "rootname": "gate_2"}
                  ]
            }
      }
}
```

此配置文件的解析内置在firefly框架代码中，该json文件不能存在注释，否则会解析出错。其中关键几个配置参数含义如下：

rootport: 供rpc调用的接口

webport: http web监听的接口

port: socket连接接口（此处和firefly框架中解析的netport名字不一样，框架中的netport在该游戏中被屏蔽）

name: 服务节点的名字，一般用一个字符串标识

remoteport: 该节点需要连接的远程节点，用一个列表进行存储

app: 启动程序的相对路径，比如game.start_up，则firefly框架中实际会执行的是import game.start_up

log: 日志存放目录

7. 消息走向

后端程序启动后，当收到客户端发送过来的消息请求后，在后端游戏框架中消息是如何走向呢?

在第一章介绍firefly流程架构图时说到，客户端client请求首先是先到proxy节点，然后再经由gate类型节点，最后再到game节点。那么接下来我们具体看下到底是如何在这几个节点间进行的通信的。

7.1 proxy节点

前面从启动文件start_xxx跟踪到firefly的FFServer类的config方法，在config方法中有以下代码：

```
    def config(self, config, servername=None, dbconfig=None,
               memconfig=None, masterconf=None):
        '''配置服务器
        '''
        ...
        app = config.get('app')#入口模块名称
        ...
        if app:
            __import__(app)
```

即在启动时根据配置文件中app参数import相应模块，在proxy节点的配置中，app的配置内容为“proxy.start_up”，那么我们来看看proxy/start_up.py文件干了些什么：

```
    import controller
    import remoteservice
```

该文件中只做了一件事，即导入controller.py和remoteservice文件。

7.1.1 控制模块controller.py

当我们在公司接触到一个全新的项目时，这里向大家提供一个快速理解项目的小技巧 ---- 碰到一个全新的模块文件时：

1）首先看的是这个模块大体干了些什么，先忽略实现的细节，不看函数内部实现详情!

2）对模块内大体功能有了大概认识后，再对感兴趣需要用到的部分进行深入跟踪；

3) 理解函数时，一样先不要看细节，先关注函数的作用，函数的输入参数，函数的返回值，有必要的情况下再去看函数每一行的实现；

在浏览controller代码时，有以下几句代码

```
websocket_factory = CustomWebProxyFactory()
port = GlobalObject().json_config.get("port")
reactor.listenTCP(port, websocket_factory)
```

该部分代码作用是proxy节点启动对websocket端口的监听，同时绑定端口收到的消息给websocket_factory实例对象。跳进去该类定义即到了proxy/protocol.py中，

消息入口

protocol.py主要干了以下几件事：

```
class DataProtocol:    # 数据打包及解包（前文提及）
class ProxyCommandService(CommandService):  # 消息分发类
class WebSocketProtocol(WebSocketServerProtocol):   # websocket消息协议类
class CustomWebProxyFactory(WebSocketServerFactory): # websocketserver处理类

proxy_service = ProxyCommandService("WebProxyService")  # 消息分发实例

def ProxyServiceHandle(target):   # 消息注册的装饰器定义
```

根据上文reactor监听时绑定的CustomWebProxyFactory类对象，我们以此为入口去理解，该类继承的twisted的WebSocketServerFactory，它是twisted框架中用来处理websocket客户端消息的server类，在游戏项目中CustomWebProxyFactory子类有一个成员protocol，protocol中实例化了WebsocketProtocol类。那具体WebsocketProtocol类是何时调用的呢？如果大家持续的跟踪WebSocketServerProtocol类，跟踪若干层之后肯定可以找到，但这里我教大家一种跟踪更简单的办法：

在WebSocketProtocol方法的构造函数处添加调试代码raise Exception()

```
class WebSocketProtocol(WebSocketServerProtocol):

    def __init__(self):
        WebSocketServerProtocol.__init__(self)
        self.time_out_task = None
        self.connect_timeout = 30 * 60
        raise Exception()     # 方便跟踪调用者，一种调试方法
```

然后在客户端开始正常登录，此时会发现proxy节点进程有异常抛出，堆栈信息如下：

2019-08-08 18:08:00+0800 [proxy.protocol.CustomWebProxyFactory] Unhandled Error Traceback (most recent call last): File "/Library/Python/2.7/site-packages/twisted/python/log.py", line 86, in callWithContext return context.call({ILogContext: newCtx}, func, *args, **kw) File "/Library/Python/2.7/site-packages/twisted/python/context.py", line 122, in callWithContext return self.currentContext().callWithContext(ctx, func, *args, **kw) File "/Library/Python/2.7/site-packages/twisted/python/context.py", line 85, in callWithContext return func(*args,**kw) File "/Library/Python/2.7/site-packages/twisted/internet/selectreactor.py", line 149, in doReadOrWrite why = getattr(selectable, method)() --- --- File "/Library/Python/2.7/site-packages/twisted/internet/tcp.py", line 1427, in doRead self.buildAddr(addr)) File "/Library/Python/2.7/site-packages/twisted/internet/protocol.py", line 140, in buildProtocol p = self.protocol() File "/data/source/echecs/proxy/protocol.py", line 100, in **init** raise Exception() exceptions.Exception:

可以看到在twisted底层会调用到我们自定义的WebSocketProtocol初始化方法。那我们下一步回归到该类定义，理解的时候，同样先忽略细节，只看主要的方法，其主要方法如下：

```
    def add_new_time_task(self):    # 添加定时任务（超时连接控制）
        ...

    def close_time_task(self):  # 关闭定时任务
        ...

    def onConnect(self, request):
        ...

    def onOpen(self):
        ...

    def onMessage(self, data, *args):
        ...

    def onClose(self, *args):
        ...

    def send_data_for_me(self, data, commandID):
        # 部分请求已经通过推送返回，此时不再在请求消息的结果中再次推送
        ...
```

从方法名不难猜测是onConnect()为连接建立时会调用的方法，onMessage为接收消息的处理方法。同样的，如果想去验证自己的推理逻辑，可以在相应的函数内部打印日志或者raise Exception，在我们第一次快速理解项目的过程中，我们不需要每一个函数都弄懂具体是什么意思，该类中初始时我们着重关注onMessage方法，其主体实现如下：

```
    def onMessage(self, data, *args):
        print u"onMessage:", [data]
        ret = self.factory._datapack._unpack(data)
        print u"ret=", ret
        if ret:
            self.add_new_time_task()
            d = proxy_service.callTarget(ret["command"], self, ret["data"])
            if d:
                d.addCallback(self.send_data_for_me, ret["command"])
```

该部分主要做了三件事：一是根据指定协议对消息进行解包，解包过程参考上文；二是添加刷新连接超时断开任务（add_new_time_task）；最后是消息分发调用对应的处理函数（callTarget），调用方式为异步调用，执行完后回调send_data_for_me方法。

callTarget方法

proxy_service对象为ProxyCommandService类的实例，该类重载了其父类CommandService的callTargetSingle方法，跟中进行会发现游戏中的callTarget方法最终执行时会调用到该处，该函数是根据消息ID，然后调用之前注册的具体的方法，在该类中寻找方法时消息ID固定死了写的是0，这是因为在proxy节点中是不需要针对不同的消息ID进行不同的处理逻辑，因为proxy节点的职责是负责消息加解包、加解密及消息转发，不涉及到具体的业务逻辑。

```
    class ProxyCommandService(CommandService):
        def callTargetSingle(self, targetKey, *args, **kw):
            """

            :param targetKey:
            :param args:
            :param kw:
            :return:
            """
            self._lock.acquire()
            try:
                target = self.getTarget(0)
                if not target:
                    return None
                if targetKey not in self.unDisplay:
                    pass
                defer_data = target(targetKey, *args, **kw)
                if not defer_data:
                    return None
                if isinstance(defer_data, defer.Deferred):
                    return defer_data
                d = defer.Deferred()
```

```
                d.callback(defer_data)
            finally:
                self._lock.release()
            return d
```

全局搜索可以找到ProxyCommandService的使用地方，其中定义了一个ProxyServiceHandle, 该函数作为装饰器在controller.py文件中加载了forwading_0方法定义上，也即消息初始注册（在import controller时执行）。

换句话说，上文的call_target方法最终将执行到forwarding_0方法内。此处我们便可以将forwarding_0方法作为该游戏框架中真正的消息入口（当然前文的onMessage方法也可以）。

forwarding_0方法

```
@ProxyServiceHandle
def forwarding_0(keyname, _conn, data):
    """
    选择一个
    :param keyname:
    :param _conn:
    :param data:
    :return:
    """
    session_id = "%s,%d" % (server_name, _conn.transport.sessionno)
    gate_name = get_gate_name(_conn.transport.sessionno)
    print "aaaaaaaaaaaaaaaaaaaaaAA:", data
    data = convert_to_json(data)
    logger.debug(u"forwarding_0: %s, %s", str(keyname), str(data))
    defered = forward_to_gate(gate_name, keyname, session_id, data)

    def on_time_out(a, b):
        logger.warn("%s is time out!" % gate_name)
        for node in GlobalObject().json_config["remoteport"]:
            if node["rootname"] == gate_name:
                node["is_available"] = False
            return
forward_to_gate(get_gate_name(_conn.transport.sessionno), keyname,
session_id, data)

    defered.addTimeout(200, reactor, on_time_out)
    return defered
```

该方法实际也只做了三件事：一是根据一定的策略选择消息下一步要转发的gate节点名字（get_gate_name）；二是将消息转发往选定的gate节点（forward_to_gate）；三是添加无法连接或超时连接控制（on_time_out）。

注意此处有一个实现分布式节点维护的细节，即当连接下一个节点超时时，更新下一个节点在当前节点的状态is_availalbe为False，这样下一次消息分发时将不会再往下一个gate节点进行消息转发。

函数最后将返回一个延迟结果对象（rpc调用gate节点函数的返回），然后代码将执行到上文callTargetSingle中使用的send_data_for_me函数。

消息出口

send_data_for_me方法很简单，第一步对返回值结果进行打包（按照游戏内约定私有化协议），第二步调用sendMessage方法推送消息给到游戏客户端。此处该函数既可以当做整个消息框架的消息出口（因为所有服务端对来自客户端请求的回应结果都将经过该函数）。

7.1.2 消息推送模块remoteservice.py

该文件内容主要用于服务端向客户端主动进行的消息推送，从文件中方法名中可见一二：

```
    @RemoteServiceHandle()
    def push_object(msgid, msg, send_list):
        print "push_object:", msgid, msg, send_list
        if isinstance(msg, dict):
            msg = ujson.dumps(msg)
        data = websocket_factory._datapack._pack(msg, msgid)
        websocket_factory.push_object(send_list, data, True)
```

其中@RemoteServiceHandle装饰器在后文中会详细讲到，其主要主要是代表着其修饰的方法可以用其他节点进行RPC调用。推送方法中同样只做了两件事：一对推送内容进行打包，二是调用定义的websocket协议示例的推送方法（push_object）将消息发往客户端。

7.2 gate节点

gate节点功能相对比较单一，其主要负责两块功能：一是处理游戏登录消息，二是根据消息ID转发相应消息到具体的游戏节点。

gate文件夹下两个文件: controller.py和start_up.py，作用和proxy节点内大体相同。controller.py文件内只有三个函数：

forwarding_game: gate节点的消息处理主函数。

```
    def forwarding_game(key, session_id, data):
        """
```

```
        消息转发给游戏节点
        :param key: 消息id
        :param session_id:
        :param data: json串，里面必须包含userid
        :return:
        """
        logger.debug(u"forwarding_game:%s", str([key, session_id, data]))
        user_id = data.get("user_id", -1)
        passwd = data.get("passwd", -1)
        if -1 == user_id and key not in [USER_OFFLINE]:
            print "forwarding_game user_id =", user_id
            return

        if USER_LOGIN == key:
            logger.debug(u"forwarding_game2:%s", str([key, session_id, data]))
            return process_login(user_id, passwd, session_id, data)

        if session_id in ROUTE_CACHE.keys():
            room_name = ROUTE_CACHE[session_id]["room"]
            data.update({"gate_name": GATE_NAME})
            return GlobalObject().root.callChildByName(room_name,
    "forwarding_game", key, session_id, data)
        # 节点转发关系缓存不存在时
        route_info = route_ins.get_route(user_id, session_id=session_id)
        print "route_info:", route_info
        if 200 == route_info.get('code'):
            data.update({"gate_name": GATE_NAME})
            room_name = route_info['info'].get("room")
            return GlobalObject().root.callChildByName(room_name,
    "forwarding_game", key, session_id, data)
        else:
            return
```

该函数主要做了以下几件事：

a. 屏蔽参数不合法的消息

b. 处理登录消息（调用process_login函数）

c. 维护更新游戏节点和gate节点对应关系缓存

d. 转发消息值room_xx节点（缓存已存在的向原room节点发，不存在的则请求web服务，根据返回值确定要转发的room节点）

process_login:登录消息处理（用户登录状态更新，同时踢掉该用户之前在其他设备登录的连接），其中实际用户登录消息的处理过程在另一个web服务中；

push_object: 接收远程节点RPC调用的和推送有关的消息，然后向proxy节点进行消息推送转发；

7.3 game节点

game节点内容比较复杂，其启动方式和proxy节点、gate节点一样，start_up.py为启动模块，controller为消息分发模块，其具体的分发逻辑在下一章将会详细阐述。

8. 消息ID定义

框架中不同消息通信是通过消息ID进行区分，消息ID为常量，会统一定义到share/messageid.py中，消息类型分为两大类，客户端请求消息及游戏推送消息，定义常量时常量范围分别在不同区间。

```
HEART_BEAT = 10000
USER_LOGIN = 100002                 # 登录
USER_RECONNECT = 100010             # 断线重连
USER_READY = 100100                 # 玩家准备
CREATE_FRIEND_DESK = 100101         # 创建好友房
JOIN_FRIEND_DESK = 100102           # 加入好友房
USER_EXIT_DESK = 100103             # 玩家退出桌子
JOIN_MATCH_DESK = 100104            # 快速加入匹配桌
JOIN_MATCH_DESK_BY_TYPE = 100105    # 加入指定匹配场
DISSOLVE_FRIEND_DESK = 100110       # 解散房间
DISSOLVE_DESK_ANSWER = 100111       # 解散房间应答
DISSOLVE_MATCH_DESK = 100112       # 解散匹配房间

USER_SET_CONFIG = 100120            # 自定义配置
USER_OFFLINE = 100130               # 玩家断线

USER_ACT = 100140                   # 玩家叫牌

USER_TEST_ACT = 100999              # 测试接口

PUSH_CALL_CARD = 101001         # 推送玩家叫牌
PUSH_DRAW_CARD = 101002         # 推送玩家摸牌
PUSH_GAME_OVER = 101003         # 推送游戏结束w
PUSH_GEN_BANK  = 101004         # 推送定庄信息
PUSH_DEAL_CARD = 101005         # 推送发牌信息
PUSH_GAME_SETTLE = 101006       # 推送游戏结算
PUSH_GAME_BU_HUA = 101007       # 推送游戏补花
PUSH_GAME_DEAL_BU_HUA = 101008 # 推送发牌补花
PUSH_PLAYER_CALL_CARD_RES = 101009        # 推送玩家叫牌响应
```

```
PUSH_USER_POINT = 101100                # 推送玩家点数发生变化
PUSH_USER_OTHER_LOGIN = 101101          # 推送玩家在其他地方登录
PUSH_DESK_DISSOLVE_RESULT = 101102      # 推送桌子解散结果
PUSH_DESK_DISSOLVE = 101103             # 推送有玩家请求解散桌子
PUSH_DESK_DISSOLVE_ANSWER = 101104      # 推送玩家对解散桌子的响应
PUSH_USER_EXIT = 101105                 # 推送玩家退出桌子
PUSH_USER_READY = 101106                # 推送玩家准备/取消准备
PUSH_USER_JOIN_DESK = 101107            # 推送玩家加入房间
PUSH_USER_SET_CONFIG = 101108           # 推送玩家更改了配置
PUSH_USER_RECONNECT = 101109            # 推送玩家断线重连
PUSH_USER_STATUS = 101110               # 推送玩家连接状态
PUSH_MATCH_DESK_DISSOLVE_RESULT = 101111      # 推送匹配桌子解散结果
PUSH_OTHER_PLAYER_CALL_CARD = 101112      # 推送匹配桌子解散结果
```

9. 游戏大厅功能实现

游戏后端框架中，我们会尝试把尽量多的消息请求通过http服务器进行处理，这是因为http服务器相对而言无论是在高可用，还是处理高并发负载均衡热更等方面会比需要长连接的游戏服务器代价小很多。

目前在该麻将开发中，登录、注册、查询匹配房、大厅等模块实现都采用http服务器对消息进行处理。

四、房间、桌子、座位、轮次、局数等

1. 房间、桌子逻辑接口实现

在棋牌类游戏中，通常会有会有以下概念：

房间：类似于线下房间的概念，为多数同类型游戏用户聚集地。在该麻将游戏中，一般使用单一进程实现。

桌子：等同于线下桌子的作用，特定数目的用户在一张桌子上进行游戏。同一张桌子一局内只能同时进行一种类型游戏；房间和桌子的概念本游戏框架中和具体的游戏已经封装抽象分离开。

座位：每个桌子有若干个座位，具体座位数根据游戏配置决定，一般按照逆时针方向进行游戏。

轮次：棋牌游戏中经常会出现多局游戏统一结算的情况，此时我们把多局游戏称为一轮。

局数：每一把游戏即为一局。

框架中为了使框架尽可能复用，层次更加明晰，选择了将桌子的概念分为了两种，一种是房间中桌子，一种是游戏内部的桌子。

房间中桌子类比于线下的桌子，在该桌子上每轮游戏可以进行不同游戏，如各类麻将和扑克，此桌子和具体游戏内容没有任何关系；

游戏中的桌子是一个相对虚化的概念，用来在具体麻将游戏中和外面的房间桌子进行一对一的对照关系，主要是方便游戏内部理解。

1.1 快速匹配

棋牌类游戏中一般匹配场会提供一个快速加入匹配场的功能，减少用户操作复杂度，提升用户体验。

我们需要理清快速匹配功能在游戏中是如何实现的，怎么去快速入手呢?

第一步，我们可以借助ide的全文搜索功能，从消息ID定义可以知道快速加入匹配场的消息ID为：

JOIN_MATCH_DESK = 100104 # 快速加入匹配桌

此时通过搜索不难发现在game/room/handlers/match_desk/join_desk.py中，在该文件的加入匹配场的执行方法execute中主要做了三件事：

1）参数验证：JoinMatchDeskValidator 参数校验，该校验方式借用的wtform进行参数校验。详细校验机制下文会详细描述。

2) 加入桌子主逻辑，调用apply_match_desk函数，其主要伪代码逻辑如下：

```
1. 寻找该房间内人数未满的桌子
2. 带权重随机选择可以加入一张桌子（权重值和桌子内当前人数有关，人数越多权重值越高，随机到的概率越大）
3. 如果随机到0人的桌子，此时需要创建一张新桌子；
4. 执行玩家坐桌逻辑（user_sit函数），玩家加入桌子时是随机选择座位
```

其中带权随机算法weight_choice实现如下（列表中元素数值越大，随机到的概率越大）：

```
    def weight_choice(weight):
        """
        :param weight:[], list对应的权重序列
        :return:int,  选取的值在原列表里的索引
        """
        t = random.randint(0, sum(weight) - 1)
        for i, val in enumerate(weight):
            t -= val
            if t < 0:
                return i
```

3）通知：通知web服务有玩家加入桌子（因为房间人数信息由web服务echecs_web_services记录），同时通知桌子上的其他玩家有新玩家进入

1.2 参数验证

上文在跟踪快速加入匹配场功能时涉及到参数验证模块，下面借由加入匹配场的参数验证阐述该框架中参数验证逻辑。

```
    from wtforms import fields, validators

    from game.room.validators.basevalidator import BaseValidator
    from game.room.models.user_manager import UserManager
    from game.room.models.roomdesk_manager import desk_mgr
    from share import errorcode


    class JoinMatchDeskValidator(BaseValidator):
        user_id = fields.IntegerField("user_id")
        session_id = fields.StringField("session_id")

        def validate_user_id(self, field):
            if UserManager().get_user_by_id(self.user_id.data):
                raise validators.ValidationError(errorcode.USER_IN_OTHER_DESK)

        def validate_session_id(self, field):
            if not self.session_id.data:
                raise validators.ValidationError(errorcode.SESSION_NOT_EXIST)
```

通过使用wtform进行参数验证，该接口对user_id, session_id两个参数进行校验，user_id为int，如果需要对user_id参数进行深入验证，比如此接口中验证该user_id对应的用户是否已坐桌，此时使用的方法为定义一个新函数，以validate_xxx（属性名），验证结果不合法时抛出指定类型异常（raise xxx）

验证器的父类BaseValidator 基本代码逻辑如下：

```
class BaseValidator(form.Form):
    """
    参数校验器
    """
    def __init__(self, handler, obj=None, **kwargs):
        self.handler = handler
        super(BaseValidator, self).__init__(handler.params, obj, **kwargs)

    def process(self, form_data=None, obj=None, data=None, **kwargs):
        if form_data is not None and not hasattr(form_data, "getlist"):
            form_data = InputWrapper(form_data)
        super(BaseValidator, self).process(form_data, obj, **kwargs)
        if not self.validate():
            error_code = self.errors.values()[0][0]
            raise ValidatorError(error_code)
```

process函数中调用父类form.Form的validate方法，其方法中不难发现会遍历验证器的所有以validate_开头的验证方。

```
def validate(self):
    """
    Validates the form by calling `validate` on each field, passing any
    extra `Form.validate_<fieldname>` validators to the field validator.
    """
    extra = {}
    for name in self._fields:
        inline = getattr(self.__class__, 'validate_%s' % name, None)
        if inline is not None:
            extra[name] = [inline]

    return super(Form, self).validate(extra)
```

当出现错误是，raise抛出的异常在上层 (game/room/handlers/basehandler.py)中进行捕获，然后封装后向用户返回参数错误信息。

```
    @defer.inlineCallbacks
    def execute_event(self):
        try:
            result = yield self.execute()
            defer.returnValue(self.success_response(result))
        except ValidatorError as e:
            defer.returnValue(self.error_response(e.error_code))
```

1.3 游戏消息分发

上文中提到的basehandler.py文件为游戏节点的核心消息分发处理类，在收到gate节点转发过来的消息后，在execute_event函数中根据注册的消息进行消息分发，调用相应的处理函数。

其中defer.inlineCallbacks装饰器twisted中用来和yield关键字配合使用，将异步的代码结构编程同步的。在BaseHandler类中同时还对消息的返回（包括成功和失败）进行了封装，将其变成约定好的json格式，函数为类方法success_response和error_response. 另外封装了根据session或user_id获取当前用户对象的方法。

在上文快速加入匹配场的消息类JoinMatchDeskHandler(BaseHandler) 中可以看到对该类加了一个类装饰器RegisterEvent，该装饰器实现将对应消息（此处为JOIN_MATCH_DESK）进行注册，方便后续收到消息时进行分发，其主要代码如下（basehandler.py文件中）：

```
    class RegisterEvent(object):
        """
        事件管理器，将命令码与handler进行绑定
        """
        events = dict()
        def __init__(self, command_id):
            self.command_id = command_id

        def __call__(self, handler):
            self.events[self.command_id] = handler
            return handler
```

那么BaseHandler类的执行消息分发的方法execute_event又是何时被调用的呢?

全局搜索时不难发现调用地点在game/controller.py中的forwarding_game方法，该方法首先判断参数中是否有user_id，或者是否是用户断线的消息，屏蔽掉game节点本身不进行处理的消息；接着才进行真正的消息分发：

```
    @RemoteServiceHandle()
    def forwarding_game(key, session_id, data):
        """
```

```
        消息转发给游戏节点
        :param key: 消息id
        :param session_id:
        :param data: json串, 里面必须包含userid
        :return:
        """
        logger.debug(u"forwarding_game:%s", str([key, session_id, data]))
        user_id = data.get("user_id", -1)
        if -1 == user_id and key not in [USER_OFFLINE]:
            return

        class_obj = RegisterEvent.events.get(key)     # 获取注册时注册的类
        if not class_obj:
            return BaseHandler.error_response(COMMAND_NOT_FOUND)
        data.update({"session_id": session_id})
        event_ins = class_obj(data, key, session_id)
        return event_ins.execute_event()
```

1.4 RemoteServiceHandle和rootserviceHandle

上文中forwarding_game函数定义时加了一个装饰器@RemoteServiceHandler, 该装饰器的主要代码如下:

```
    from firefly.server.globalobject import GlobalObject


    class RemoteServiceHandle:
        """
        重新定义原remoteservicehandler,使其可以对同一个函数/类进行多重绑定
        eg.
            @RemoteServiceHandle("gate_1")
            @RemoteServiceHandle("gate_2")
            def forwarding_gate():
                pass
        """

        def __init__(self):
            pass

        def __call__(self, target):
            for remote_node in GlobalObject().json_config["remoteport"]:
                GlobalObject().remote[remote_node["rootname"]]\
                    ._reference._service.mapTarget(target)
```

```
            return target
```

在游戏firefly中有一个remoteservicehandle装饰器，在麻将游戏中，为了实现节点间分布式通信，故重新封装了RemoteServiceHandlele, 重新封装的新类可以结合配置文件config.json实现gate节点和game节点多对多的关系。

那么RemoteServiceHandler的作用是什么呢?

它的作用是当我们在另外一个游戏节点希望远程RPC调用当前节点的某个方法时，此时便在该方法的定义处加上该装饰器。游戏中当gate节点调用game节点方法、gate节点调用proxy节点方法时使用。远程节点调用该方法时一般配套使用GlobalObject().root.callChildByName(...)

同时在游戏中还有另一个装饰器@rootservicehandle，在游戏中用来在proxy节点调用gate节点、game节点调用gate节点方法时使用，其配套调用方式为GlobalObject().remote[node_name].callRemote(...)

1.5 代码结构

game节点文件目录下内容主要分为两块：mahjong和room。

mahjong: 麻将游戏目录。里面包括麻将内部所有游戏逻辑实现。

constants: 游戏内常量定义；

controls: 游戏内部管理控制类目录；

models: 游戏内部各种逻辑实现；

room：和具体游戏内部无关的房间桌子逻辑处理。

handlers：各类用户消息处理类；

models: 桌子及用户管理类

validators: 各消息接口参数验证类

common_define.py: 房间层面常量定义

notifybridge.py: 房间通知游戏内部统一接口

bridgecontroller.py: 房间和游戏交互通信接口

controller.py: game节点消息分发控制类

push_service.py: 推送消息接口

session_gate_rel.py: session和gate节点关系缓存类

start_up.py: 启动文件

1.6 退出房间

游戏中退出房间也即离开桌子接口，我们需要找到其实现代码，那怎么找呢?

这里提供一种快速查找的方法：之前有看到过消息ID定义（在messageids.py中），在其中可以找到玩家退出桌子的消息 USER_EXIT_DESK = 100103，此时用USER_EXIT_DESK进行全局搜索可以发现在user_exit.py文件中（game/room/handlers/）。该文件即玩家退出桌子请求的实际处理程序。

```
    @RegisterEvent(USER_EXIT_DESK)
    class UserExitHandler(BaseHandler):

        def execute(self, *args, **kwargs):
            """
            玩家请求退出桌子
            :param args:
            :param kwargs:
            :return:
            """
            validator = UserExitValidator(handler=self)
            data = validator.user.to_dict()
            validator.desk.notify_desk_some_user(PUSH_USER_EXIT, data,
    [validator.user.user_id])
            session_gate_ins.del_rel(validator.session_id.data)
            validator.desk.user_exit(validator.user.user_id)
            UserManager().exit_user(validator.user.user_id)

            return {"code": 200}
```

其主要内容如下：

a. 参数验证

b. 通知桌子上其他用户有用户退出

c. 删除session和gate对应关系（因为用户退出后再进入可能是进入不用的game节点）

d. 清理用户管理类中用户残留信息（有两块，一块是桌子中，一块是用户管理类中）

2. 游戏开始前相关操作

2.1 准备

和离桌同样的搜索方法（根据message_ids文件全局搜索）可以定位到用户游戏准备的操作逻辑实现位于game/room/handlers/user_ready.py下，其主要执行步骤有：

1. 参数验证是否合法
2. 通知用户准备成功
3. 通知桌子上的其他玩家有新玩家准备
4. 检查是否当前桌子上所有玩家都已准备好，如果都准备好了，则游戏自动开始
5. 如果游戏正式开始，此时需要做两件事：一是通知web服务当前桌游戏一开始，二是调用游戏内部桌子游戏开始逻辑（开始一局游戏， 即desk.start_game()）

2.4 准备

其核心代码实现如下：

```
@RegisterEvent(USER_READY)
class UserReadyHandler(BaseHandler):

    def execute(self, *args, **kwargs):
        """
        用户准备/取消准备
        :param args:
        :param kwargs:
        :return:
        """
        validator = UserReadyValidator(handler=self)

        if 1 == validator.ready.data:
            validator.user.set_status(UserStatus.READY)
        else:
            validator.user.set_status(UserStatus.UNREADY)

        data = {"user_id": validator.user.user_id, "nick":
validator.user.nick_name,
                "ready": validator.ready.data, "seat_id":
validator.user.seat_id}
        validator.desk.notify_desk_some_user(PUSH_USER_READY, data,
[validator.user.user_id])

        response_data = {"ready": validator.ready.data}
        validator.desk.notify_player(validator.user.seat_id, USER_READY,
response_data)
```

```
            # 当所有玩家都准备好时，默认触发游戏开始
            all_ready = 1
            for u in validator.desk.users:
                if not u or u.status == UserStatus.UNREADY:
                    all_ready = 0
                    break
            if all_ready:
                # 通知web服务器记录是否开始
                user_ids = []
                for i in validator.desk.users:
                    user_ids.append(i.user_id)
                ret = notify_web_server_match_room_start_game(user_ids,
                                                      validator.session_id.data,

    room_name=GlobalObject().json_config.get("name", ''),

    room_type=validator.desk.room_type)
                if ret.get("ret") == 0:
                    validator.desk.start_game()
                else:
                    raise Exception("start game error ret=%s" % ret)

            return {"need_push": 0}
```

其中desk.start_game()方法desk对象来自于房间管理器（可跟踪发现room_mgr），桌子类的定义位于models/room_desk.py中，其start_game在设定桌子状态为游戏中后，即调用和游戏内部通信的接口文件notifybridge.py, 最终调用到游戏内部。

2.2 取消准备

该游戏中，取消准备和准备调用的同一个接口，只是接口的参数不一样（ready字段0为取消准备，1位准备完成），用户在游戏未开始之前可以根据自己的意愿调整是否取消准备或开始准备。

3. 断线重连

3.1 断线

用户在网络断线时，虽然客户端不会主动向服务端发送网络断线的消息，但因为是长连接，当客户端网络出现问题时，此时服务端是能够自动检测到用户断线，然后触发后续相关的断线逻辑。

用户断线后服务端首先发现的地点在websocket连接的地方，即proxy，具体跟踪有两种方式可以实现目的：一是直接去proxy节点文件夹下找连接断开时的逻辑，然后验证猜测；二是和上面一样根据断线的消息ID全文检索（USER_OFFLINE）不难发现在proxy/protocol.py中的onClose()函数中有调用到。

```
    def onClose(self, *args):
        print u"onClose:", args
        self.close_time_task()
        self.add_new_time_task()
        d = proxy_service.callTarget(USER_OFFLINE, self, {})
        self.factory.onClose(self)
```

WebsocketProtocol类的onClose()触发时机是客户端和服务端连接断开时，此时会向后端的gate节点发送用户连接断开的消息，gate节点收到消息后，会继续将消息转发向game节点。最后game节点的用户断线处理逻辑位于（可按前序方法搜索）game/room/handlers/user_offline.py中，其主要做了几件事：

```
1. 参数验证是否合法
2. 删除session中和gate的关系缓存数据
3. 通知桌子上其他玩家有特定玩家断线
4. 如果游戏未开始，此时相当于执行玩家退出操作（desk.user_exit，
UsserManager().exit_user）
   该游戏中用户断线对游戏内部无影响
```

其部分代码如下：

```
@RegisterEvent(USER_OFFLINE)
class UserOfflineHandler(BaseHandler):

    def execute(self, *args, **kwargs):
        """
        玩家断线
        :param args:
        :param kwargs:
        :return:
        """
        validator = UserOfflineValidator(handler=self)
        validator.user.set_offline()
        session_gate_ins.del_rel(validator.session_id.data)
        data = {"user_id": validator.user.user_id, "nick":
validator.user.nick_name, "is_offline": validator.user.is_offline}
        validator.desk.notify_desk_some_user(PUSH_USER_STATUS, data,
[validator.user.user_id])
        if validator.desk.status != DeskStatus.PLAYING and
validator.desk.desk_type == DeskType.MATCH_DESK:
            print "game.status != playing!"
            validator.desk.user_exit(validator.user.user_id)
            UserManager().exit_user(validator.user.user_id)
```

```
        return {"need_push": 0}
```

3.2 断线重连

断线重连一般是游戏中极度容易出错的地方，其逻辑一般相对复杂。这一块我们将着重进行跟踪，在game节点中其具体处理方法入口（搜索可得知）在game/room/handlers/user_reconnect.py中

```
@RegisterEvent(USER_RECONNECT)
class UserReconnectHandler(BaseHandler):
    def execute(self, *args, **kwargs):
        """
        玩家断线重连
        :param args:
        :param kwargs:
        :return:
        """
        validator = UserReconnectValidator(handler=self)
        UserManager().add_user(validator.user.user_id,
validator.desk.desk_id, self.session_id)
        session_gate_ins.update_rel(self.session_id, self.gate_name)
        validator.user.is_online = 1

        data = {"user_id": validator.user.user_id, "nick":
validator.user.nick_name}
        validator.desk.notify_desk(PUSH_USER_RECONNECT, data)

        game_data =
validator.desk.get_reconnect_info(validator.user.seat_id)
        user_info = validator.desk.get_users_info()
        end_time = game_data["wait_task"].get("end_time", 0)
        if end_time:
            game_data["wait_task"]["end_time"] = end_time - time.time()
        else:
            game_data["wait_task"]["end_time"] = end_time
        validator.desk.notify_player(validator.user.seat_id,
                                     USER_RECONNECT,
                                     {"user_info": user_info, "game_data":
game_data})
        if game_data.get('wait_task', None):
            pass
            # 每个玩家同时只可能有一个定时任务待执行
            # validator.desk.notify_player(validator.user.seat_id,
game_data["wait_task"]["command_id"], act_info)
```

```
        return {"need_push": 0}
```

其主要逻辑如下：

1. 验证参数是否合法
2. 管理类中添加用户信息，同时更新session--gate缓存
3. 通知桌子上其他玩家该玩家回来了
4. 获取游戏中当前时刻的实时信息（游戏各阶段信息内容不一样）
5. 获取用户当前信息（用户id，昵称，游戏点数，状态，座位号）
6. 获取哪些任务正处于倒计时状态
7. 返回该重连玩家断线后恢复场景需要的各类信息

其中第四个阶段获取游戏中当前时刻信息desk.get_reconnect_info，持续跟踪进去会跳转游戏逻辑内部，在game/mahjong/controls/gamemanager.py的get_reconnect_desk_info函数，该函数中根据桌子和座位号获取游戏现场信息（desk类的get_all_info_of_player方法）

```
    def get_all_info_of_player(self, seat_id):
        """
        获取指定玩家在当局游戏中的所有信息
        :param seat_id:
        :return:
        """
        return {
            "player_info": self.game_data.get_all_player_info(seat_id),
            "desk_info": self.game_data.get_desk_info(),
            "wait_task": self.game_data.get_wait_task(seat_id)
        }
```

player_info存放游戏中所有玩家信息，内容会比较复杂，包括当前桌子上所有玩家的手牌信息，非本人信息部分手牌需要进行内容隐藏（比如不能让该玩家看到其他玩家有什么牌，只能看到出过什么牌）

desk_info存放的是该桌当局的信息，如桌子号，桌子类型，游戏状态，庄家是谁，还剩多少张牌

wait_task存放当前有哪些任务处于倒计时的状态，比如出牌倒计时等，此处实现设计到定时任务管理模块，后文中将详细阐述

五、麻将通用胡牌算法

麻将游戏中，胡牌逻辑是其中变化最多、最复杂的功能之一。不同的麻将胡牌类型胡法千奇百怪，限于篇幅，此处游戏中我们只讨论一些通用麻将胡法的实现，具体各地方麻将的特殊胡法实现暂不赘述。

具体使用场景有两种：一是在玩家进行某个操作后（如出牌）判断是否可以听牌；二是是否可以胡牌（包括自摸和放炮|点炮胡）；

1. 胡牌接口的封装

胡牌和听牌的接口都封装在了game/mahjong/models/hutype目录下，该目录下typemanager.py为胡牌接口文件，也即所有调用胡牌模块的调用入口，胡牌的具体业务和计算逻辑都封装在该文件中。

胡牌接口类的核心定义如下：

```
class HuTypeManager(object):
    """
    胡牌类型管理
    """

    def __init__(self, game_config, card_analyse):
        super(HuTypeManager, self).__init__()

        self.card_analyse = card_analyse          # 牌值分析器
        self.fan_info = game_config.hu_fan_info
        self.mutex_list = game_config.mutex_list
        self.type_info = game_config.used_hu_types  # 特殊胡牌牌型信息，key存放基础胡牌类型， value存放特殊胡牌类型
        self.base_hu_types = self.type_info.keys()
        self.special_hu_types = []
```

在麻将游戏中，胡牌相关接口的调用频次非常高，玩家每抓一张牌或出一张牌都要判断是否有玩家可以听牌或胡牌。为了尽可能提高游戏性能，经过综合分析，我们发现胡牌的算法可以分为两个层次：

基本胡牌算法：一些不能继续细分成更小的胡法的类型，比如4 * 3+1 * 2的平胡（屁胡），七小对，十三幺，清一色等等

特殊胡牌算法：指基于基本胡牌类型上的又额外满足一定规则的胡法，如碰碰胡（基于平胡），豪华七小对（基于七小对）等

基本胡法一般相对数目不会特别多，针对每一种基本胡法类型需要有一个单独的算法；而特殊胡牌类型的判断出于效率考虑，一般我们是首先判断是否符合其对应的基本胡牌类型，如果不符合，那自然也不可能满足特殊胡牌类型，在计算出满足特定基本胡法的前提下，此时我们就只需要判断基于这些基本胡法类型的基础上衍生出来的特殊胡法。比如判断出某副手牌可以七小对，但不满足平胡，此时就没有必要去计算是否满足碰碰胡（因为平胡都不是，就一定不是碰碰胡）。

在麻将的基本胡牌类型和特殊胡牌类型中，有些胡牌类型是可以共存的，比如手牌可以满足七小对的同时又是清一色，算番型时两种胡法番数是相加或相乘，而有的胡法之间是包含关系，在算番型时是互斥的，比如七小对和豪华七小对，算番时就只算豪华七小对的番数。

1.1 听牌接口

听牌接口判断主要逻辑如下：

1. 首先判断手牌张数是否合法，如果%3 ！=2 ，此时不可能听牌，一般麻将判断听牌时手牌和牌堆前碰的牌总数目为14张
2. 将手牌和自己牌堆前的牌（吃、碰、杠的牌）联合起来
3. 因为听牌判断逻辑中间会更改手牌，所以复制一份手牌再判断
4. 循环判断打出某一张手牌后剩下的牌是否可以听
5. 如果可以听牌，再计算可以胡牌的类型；
6. 去除胡牌类型中互斥的胡牌类型（如七小对和豪华七小对就只保留豪华七小对）
7. 计算可胡牌类型的番型

其实现代码如下：

```
    def check_ting_result(self, hand_card):
        """
        检查是否可以听牌
        :param hand_card:
        :return: {出牌1：{胡的牌:{"fan":胡牌基本类型番数, "type_list":[胡牌类型]},
    ...}
        """
        if 2 != len(hand_card.hand_card_vals) % 3:
            return {}
        # 组合联合手牌
        hand_card.union_hand_card()
        ret = {}
        temp_card_vals = list(set(hand_card.hand_card_vals))
        for c in temp_card_vals:
            cards = hand_card.hand_card_vals
            cards.append(LAI_ZI)
            cards.remove(c)
            ting_infos = self.card_analyse.get_can_ting_info_by_val(cards)
            if ting_infos:
                ret[c] = {}
                hand_card.del_hand_card_by_val(c)
                for bt, lst in ting_infos.items():
                    for tc in lst:
                        hand_card.add_hand_card_by_vals(card_vals=[tc])
                        special_hu_list = self.check_special_hu(hand_card,
    [bt])
                        print "special_hu_list = ", special_hu_list
                        can_hu_list = copy.deepcopy(special_hu_list)
```

```
                    can_hu_list.append(bt)
                    print "can_hu_list = ", can_hu_list
                    mutexed_list = self.remove_mutex_type(can_hu_list)
                    print "mutexed_list = ", mutexed_list
                    total_fan = self.get_fan_hu_type_list(mutexed_list)
                    ret[c][tc] = {"fan": total_fan, "type_list":
mutexed_list}
                    hand_card.del_hand_card_by_val(tc)
            hand_card.add_hand_card_by_vals(card_vals=[c])
    return ret
```

该接口类的成员变量 self.card_analyse 为牌值分析器，该类的实例中封装了吃、碰、杠、平胡等基本胡判断方法。

1.2 胡牌接口

在游戏模块中，所有的麻将牌在计算时都采用16进制的整数进行表示，如假使一万用0x01表示，则二万用0x02表示，一条用0x11表示，0x12代表二条。

胡牌接口的主要逻辑如下：

1. 如果是点炮胡判断，则将可能点炮的牌加到手牌中
2. 检查手牌是否符合基本胡牌类型
3. 如果可以胡某些基本胡牌类型，则再判断可以胡哪些特殊胡法
4. 点炮胡时再从手牌中将点炮牌去掉
5. 去除发生互斥的胡牌类型

其代码实现为：

```
def check_hu_result(self, hand_card, pao_card_val=BLACK):
    """
    检查胡牌结果
    :param hand_card: 待检查的手牌信息，手牌对象
    :param pao_card_val: 点炮的牌
    :return: 返回可胡牌类型
    """
    if pao_card_val:
        hand_card.add_hand_card_by_vals(card_vals=[pao_card_val])
    base_type_list = self.check_base_hu(hand_card)
    can_hu_type_list = copy.deepcopy(base_type_list)
    if base_type_list:
        special_type_list = self.check_special_hu(hand_card,
base_type_list)
```

```
            can_hu_type_list.extend(special_type_list)
        if pao_card_val:
            hand_card.del_hand_card_by_val(pao_card_val)
        # 去除互斥胡法
        return self.remove_mutex_type(can_hu_type_list)
```

基本胡法检查和特殊胡法的检查类似，主要逻辑都为：

> 遍历基本胡牌类型，HU_TYPE_DICT字典存储了各胡牌类型对应的实例化对象，胡牌的判断都是通过具体类型对象的is_this_type函数。

其部分代码如下：

```
    def check_base_hu(self, hand_card):
        """
        检查是否可胡基本牌型
        :param hand_card: hand_card 对象，手牌数要求为 %3=2张
        return: base_type_list: []
        """
        base_type_list = []
        for i in self.base_hu_types:
            if HU_TYPE_DICT.get(i).is_this_type(hand_card, self.card_analyse):
                base_type_list.append(i)
        return base_type_list
```

2. 平胡算法实现

平胡的算法判断调用封装在game/mahjong/models/utils/cardanalyse.py文件中，该文件如上文所说承担牌值分析的作用，这里我们现主要跟踪平胡算法的接口，函数名为get_can_pi_hu_info_by_val，跳进去会到同utils目录下的pihu_anylyse.py中，该文件将屁胡（平胡）判断封装在了PiHuAnalyse类中。

该类是一个公用算法类，和具体的游戏逻辑没有直接依赖关系，可以作为一个三方库使用，此处使用的平胡算法是有考虑到存在癞子的情况，癞子使用常量数字表示。

在理解一个比较复杂的类时，这边教大家一种快速理解的思路方法，开始时我们不要纠结于细节，不要关注与每个函数的具体代码实现，我们只看该类的方法定义，同时首先看的是该类的公有方法，而私有方法（一般以__开头的）先跳过。该类中公有方法接口定义如下：

```
    def ke(self, card_list):
        """
        判断列表中是否相同的牌
        :param card_list: 牌值列表
```

```
        """

    def shun(self, card_list):
        """
        判断列表中是否为顺子
        :param card_list: 列表，前三个元素值按大小升序排列
        """

    def hu(self, js, ls):
        """
        实际判胡
        :param js: 列表，  癞子列表
        :param ls: 列表，  普通牌列表
        :return: 返回[[顺子/刻子1], [顺子/刻子2], ...]
        """

    def get_hu_detail(self, card_list):
        """
        胡牌判断
        :param card_list: 牌值列表
        :return: 返回所有胡牌详细信息[胡法1, 胡法2, ...]
                胡法1 = [[将牌], [[顺子/刻子1], [顺子/刻子2], ...]]]
        """

    def trans_non_num_card_to_gap(self, card_list):
        """
        将非数字牌（风,箭,字,花）值转换成 间断的，避免误判为顺子
        :param card_list:
        :return:
        """

    def trans_non_num_card_from_gap(self, card_list):
        """
        将间断的非数字牌（风,箭,字,花）值转换成原始值
        :param card_list:
        :return:
        """

    def trans_non_num_card_from_gap_plus(self, card_list):
        """
        将间断的非数字牌（风,箭,字,花）值转换成原始值
        :param card_list:
        :return:
        """
```

```
    def get_shun_liang_tou_card(self, card_1, card_2):
        """
        获取顺子两头的牌，card_1=card_2+1
        """

    def get_lai_zi_card(self, hu_detail):
        """
        从胡牌中获取用癞子替代的牌
        :param hu_detail: 返回所有胡牌详细信息[胡法1，胡法2，...]
                胡法1 = [[将牌]，[[顺子/刻子1]，[顺子/刻子2]，...]]]
        :return:
        """

    def get_ting_cards(self, card_list):
        """
        获取听牌
        :param card_list:
        :return: []
        """
```

平胡算法的关键入口在get_hu_detail函数，该函数进行胡牌判断，然后返回可以胡牌的类型，其主要逻辑如下：

1．检查传参的手牌中是否有花，如果有花，则不可能是平胡，直接返回；
2．一般麻将平胡时手牌数一定是%3==2 ；
3．防止在计算顺子时把连着的风牌和箭牌算为顺子，需要对风牌和箭牌数值进行转换；
4．算法核心：
  a．先计算将牌的组合（此种思路相较先计算顺子或刻子性能更好），需要考虑有癞子的情况；
    （如果任意两张牌相等或者两张牌的数值之和大于癞子数值（1000）则为1组将牌组合情况）
  b．去除将牌后，求出剩余的癞子和普通牌
  c．判断剩余的牌是否可以组成4个顺子或刻子：
    1）如果剩余的牌总共3张：有两个癞子时一定可以组成顺子或刻子，有一个癞子时如果剩余两张牌相同或者数值连着或者数值相差1，此时也可以组成顺子或刻子；
    2）如果剩余的牌大于3张，则算出一个顺子或刻子后再递归求剩下的顺子和刻子；
      （因为待计算的牌都是按照牌的数值升序排列，如果想要胡牌，则所有的牌都必须参与到某个顺子或刻子中，所以在递归判断时我们取第一张牌看其能否组成顺子或刻子即可）
      判断是否组成顺子（假使判断该牌数值为x）：
       判断该牌的后面数值紧挨两张x+1，x+2的是否在手牌中存在；
       或判断x-1，x+1是否存在
       或x-1，x+1，x+2，x-2中是否存在一张，剩下一张以癞子替代
      判断是否能组成刻子：
       牌值x的张数+癞子数是否>=3

其关键代码如下：

```
    def hu(self, js, ls):
        """
        实际判胡
        :param js: 列表，  癞子列表
        :param ls: 列表，  普通牌列表
        :return: 返回[[顺子/刻子1], [顺子/刻子2], ...]
        """
        # global cut
        # cut += 1
        jlen = len(js)
        llen = len(ls)
        rst = [[] for _ in xrange(6)]
        if llen == 0:
            return [[js]]
        if jlen + llen == 3:
            if jlen >= 2 or \
                    (jlen == 1 and (self.ke(ls) or abs(ls[0] - ls[1]) <= 2)) or
\
                    (jlen == 0 and (self.ke(ls) or self.shun(ls))):
                return [[js + ls]]
            else:
                return []
        elif jlen + llen > 3:
            s1, s2 = self._index_card(ls, ls[0] - 1), self._index_card(ls,
ls[0] - 2)
            if llen >= 3 and self.ke(ls[:3]):
                rst[0] = map(lambda x: [ls[:3]] + x, self.hu(js, ls[3:]))
            if s1 != -1 and s2 != -1:
                tmp = [ls[0], ls[s1], ls[s2]]
                rst[1] = map(lambda x: [tmp] + x, self.hu(js, self._delete(ls,
tmp)))
            if jlen > 0 and llen >= 2 and self.ke(ls[:2]):
                tmp = [js[0]] + ls[:2]
                rst[2] = map(lambda x: [tmp] + x, self.hu(js[1:], ls[2:]))
            if jlen > 0 and s1 != -1:
                tmp = [js[0], ls[0], ls[s1]]
                rst[3] = map(lambda x: [tmp] + x, self.hu(js[1:],
self._delete(ls, [ls[0], ls[s1]])))
            if jlen > 0 and s2 != -1:
                tmp = [js[0], ls[0], ls[s2]]
                rst[4] = map(lambda x: [tmp] + x, self.hu(js[1:],
self._delete(ls, [ls[0], ls[s2]])))
```

```
            if jlen > 1:
                tmp = [js[0], js[1], ls[0]]
                rst[5] = map(lambda x: [tmp] + x, self.hu(js[2:], ls[1:]))
            return reduce(lambda x, y: x + y, rst)

    def get_hu_detail(self, card_list):
        """
        胡牌判断
        :param card_list: 牌值列表
        :return: 返回所有胡牌详细信息[胡法1, 胡法2, ...]
                胡法1 = [[将牌], [[顺子/刻子1], [顺子/刻子2], ...]]
        """
        temp_card_list = copy.deepcopy(card_list)
        rst = []
        for c in temp_card_list:
            if CardType.HUA == Card.cal_card_type(c):
                # 如果牌中有花,则不能为屁胡
                return rst

        num_len = len(card_list)
        if num_len % 3 == 2:
            card_list.sort(key=lambda x: -x)
            card_list = self.trans_non_num_card_to_gap(card_list)
            jiang = []
            tmpset = set()

            # 首先求出所有的将牌组合
            for i in xrange(num_len-1):
                for j in xrange(i + 1, num_len):
                    if card_list[i] == card_list[j] or card_list[i] +
    card_list[j] > LAI_ZI:
                        if card_list[i] * LAI_ZI + card_list[j] not in tmpset:
                            jiang.append([card_list[i], card_list[j]])
                            tmpset.add(card_list[i] * LAI_ZI + card_list[j])
            for it in jiang:
                tmp = self._delete(card_list, it)
                jp = filter(lambda x: x >= LAI_ZI, tmp)
                lp = filter(lambda x: x < LAI_ZI, tmp)
                rst += map(lambda x: [it, x], self.hu(jp, lp))
            rst = self.trans_non_num_card_from_gap_plus(rst)
        return rst
```

在计算平胡的听牌接口处理逻辑如下：

判断是否能听平胡，也即打出一张牌后剩下的牌是否缺一张特殊的牌即可胡牌。
在算法具体计算时，同样需要考虑到癞子的情况。
听牌算法在程序中的实现策略是：
　将手牌中的一张牌替换为癞子，如果此时能够胡牌，则只需要计算在胡牌的组合中癞子被当做的牌，这些被当做的牌就是可以听的牌；

判断可听牌时癞子被当做的牌需要考虑几种情况：

a. 癞子做将牌时；

b. 癞子做顺子牌的两头

c. 癞子做顺子牌的中间

d. 癞子做刻字牌

关键代码如下：

```
    def get_lai_zi_card(self, hu_detail):
        """
        从胡牌中获取用癞子替代的牌
        :param hu_detail: 返回所有胡牌详细信息[胡法1, 胡法2, ...]
                胡法1 = [[将牌], [[顺子/刻子1], [顺子/刻子2], ...]]]
        :return:
        """
        ting_cards = []
        for card_list in hu_detail:
            jiang_cards = card_list[0]

            if LAI_ZI < jiang_cards[0] + jiang_cards[1]:
                 ting_cards.append(jiang_cards[0] + jiang_cards[1] - LAI_ZI)
            for cards in card_list[1]:
                if 3 == len(cards) and LAI_ZI == cards[0]:
                    if cards[1] == cards[2]+1:

     ting_cards.extend(self.get_shun_liang_tou_card(cards[1], cards[2]))
                    elif cards[1] == cards[2]+2:
                        ting_cards.append(cards[1]-1)
                    elif cards[1]==cards[2]:
                        # 此组合为刻子
                        ting_cards.append(cards[1])
        return list(set(ting_cards))

    def get_ting_cards(self, card_list):
```

```
        """

        :param card_list:
        :return: []
        """
        return self.get_lai_zi_card(self.get_hu_detail(card_list))
```

3. 七小对算法实现

七小对算法的判断调用同样封装在game/mahjong/models/utils/cardanalyse.py文件中，其中get_can_qi_dui_by_val方法将具体跳转到实际的七小对判胡类QiDuiAnalyse（game/mahjong/models/utils/qidui_analyse.py)。

七小对的判胡方法相对比较简单，在考虑癞子的情况下，其主要思路如下：

```
1. 如果手牌不是14张，则一定不能胡七小对牌型；
2. 计算出癞子数
3. 新申请一个列表（深拷贝手牌），遍历列表，如果第i个元素和第i-1个元素相同，则列表第i个元素数值置为0；
   这样一轮遍历下来，即可统计出总共有几对；
   接下来考虑癞子存在的情况，当上一步统计出来的对子数+癞子数目>=7时，此时则一定可以胡七小对
```

下文是其初步代码实现：

```
    def get_hu_detail(self, card_list):
        """
        七对判断
        :param card_list: 牌值列表
        :return: 返回所有胡牌信息[[对子1], [对子2], ...]]]
        """
        temp_card_list = copy.deepcopy(card_list)
        if 14 == len(temp_card_list):
            lai_zi_num = sum(temp_card_list)/LAI_ZI

            temp = [temp_card_list[0]]
            for i in xrange(1, 14):
                if LAI_ZI != temp_card_list[i] and temp_card_list[i] == temp[i-1]:
                    temp.append(0)
                else:
                    temp.append(temp_card_list[i])
            dui_num = 0
```

```
            for j in temp:
                if 0 == j:
                    dui_num += 1
            if 7 <= dui_num + lai_zi_num:
                hu_info = []
                for k in xrange(1, 14):
                    if 0 == temp[k]:
                        hu_info.append([temp[k-1], temp[k-1]])
                    elif 0 != temp[k-1] and LAI_ZI != temp[k-1]:
                        hu_info.append([temp[k-1], LAI_ZI])
                for _ in xrange(7-len(hu_info)):
                    hu_info.append([LAI_ZI, LAI_ZI])
                return hu_info
        return []
```

获取癞子被当做的牌，此时需要考虑两种情况（get_lai_zi_card函数）：

a. 某个对子中有一个癞子时，此时癞子被当做对子中另一张牌；

b. 某个对子中有两个癞子即为癞子对，此时可听任何一张牌；

4.十三幺算法实现

十三幺是一种特殊的玩法，胡牌时十四张牌为：一万，九万，一条，九条，一饼，九饼，东南西北中发白，十三张牌，然后再加一张这十三张牌中的任何一张即可胡牌，为一种超大胡胡法。

十三幺的判断算法相对比较巧妙和简单，主要分为两步：

a. 将十三幺的牌变成集合 减去 手牌集合， 差集即为胡十三幺缺的牌；

b. 上一步缺的牌数是否小于癞子数，如果小于，则可以胡十三幺；

核心代码为：

```
    def get_hu_detail(self, card_list):
        """
        十三幺判断
        :param card_list: 牌值列表[card1, ..., laizi1, ...]
        :return: 返回所有胡牌信息[card1, ..., laizi1, ...]
        """
        temp_card_list = copy.deepcopy(card_list)
        if 14 == len(temp_card_list):
            lai_zi_num = sum(temp_card_list) / LAI_ZI
            temp = list(set(SHI_SAN_YAO) - set(temp_card_list))
            if lai_zi_num >= len(temp):
                return temp_card_list
        return []
```

听牌接口中获取癞子被当做的牌接口需要考虑两种情况：

a. 只有一个癞子时，听的牌为十三幺减去手牌集合后的差集；

b. 有多个癞子时或差集为空时（十三张牌都不缺），听的牌为十三张；

关键代码如下：

```
    def get_lai_zi_card(self, card_list):
        """
        从胡牌中获取用癞子替代的牌
        :param card_list: [card1, ..., laizi1, ...]
        :return:
        """
        ting_cards = []
        temp = list(set(SHI_SAN_YAO) - set(card_list))
        lai_zi_num = sum(card_list) / LAI_ZI
        if temp and 1 == lai_zi_num:
            ting_cards.extend(temp)
        else:
            ting_cards.extend(SHI_SAN_YAO)
        return ting_cards
```

5. 麻将特殊胡牌抽象

不同麻将的特殊胡法非常多，此处我们只简述三种特殊胡法的实现，其他特殊胡法实现可以参照示例的实现过程。

查找特定胡牌的代码在什么地方有一种快速的方法，找到胡牌类型的常量定义（游戏中的常量定义在game/mahjong/constants/）。比如杠上开花GANG_SHANG_KAI_HUA，然后全局搜索发现在typemanager.py中具体使用到，其为字典HU_TYPE_DICT的一个value值，再跟踪跳跃进value的实例对象中即可发现刚上开花的具体实现。

5.1 杠上开花

根据上面的方法可以找到杠上开花在game/mahjong/models/hutype/eight/gangshangkaihua.py中，所有的特殊胡牌类型都会继承与基本胡牌类型BaseType类，通过在子类中覆盖基类方法 is_this_type 来判断是否符合某种特殊的胡法。

先说下杠上开花胡法的定义：它是指在杠牌之后补抓的牌正好可以胡牌（不包括补花）。其功能关键点即两处：

a. 最后一个动作为玩家杠牌（可能为补杠、明杠、暗杠）

b. 可以胡牌

在判断特定胡牌时，默认的前提条件是一定可以胡相应的基本胡，所以此处不需要再判断是否符合某种基本胡玩法。其核心代码如下：

```
    def is_this_type(self, hand_card, card_analyse):
        if len(hand_card.game_data.act_record_list)< 1:
            return False
        record = hand_card.game_data.act_record_list[-1]
        # 倒数第一个动作为杠
        return record.act_type in [Act.BU_GANG, Act.DIAN_GANG, Act.AN_GANG]
```

一局游戏中的动作信息都会存放在全局对象game_data的act_record_list成员中。game_data的实现位于game/mahjong/gamedata.py里，该文件主要封装了游戏过程中所要用到的全局游戏信息，主要两个类：

RoundData： 一轮游戏中的数据信息，如玩家数，一轮为几局，当前是第几局，当前几份情况等；

GameData: 当局游戏中的全局数据，包括桌子编号，游戏配置，玩家对象，胡牌信息，出牌信息，结算信息等等；

判胡方法传参的一个参数为手牌对象hand_card，该类定义在game/mahjong/models/card/hand_card.py中，HandCard类主要存储手牌相关数据，大体如下：

```
    class HandCard(object):
        """
        手牌类不关注任何游戏逻辑，只负责存储部分数据,定义数据格式
        """
```

```
    def __init__(self, seat_id, game_data):
        self.game_data = game_data
        self.seat_id = seat_id
        self.hand_card_info = {}        # 手牌信息（不包括吃碰的牌）
        self.union_card_info = {}       # 全部手牌信息（包括吃碰）
        self.init_card_info()

        self.out_card_vals = []         # 已出的牌
        self.chi_card_vals = []         # 吃的牌
        self.peng_card_vals = []        # 碰的牌
        self.dian_gang_card_vals = []   # 点杠的牌
        self.bu_gang_card_vals = []     # 补杠的牌
        self.an_gang_card_vals = []     # 暗杠的牌
        self.ting_info_records = []     # 听牌信息记录
        self.i_can_see_cards = []       # 所有我可以见到的牌

        self.hua_card_vals = []         # 花牌
        self.last_card_val = BLACK      # 最后一张牌
        self.jiang_card_val = BLACK     # 将牌
        self.lou_hu_card_vals = []      # 当圈漏胡的牌

        self.dian_gang_source = {}     # 点杠的来源， {card_val: seat_id}
        self.shun_zi_info = None
        self.ke_zi_info = None
        self.hu_source_seat_id = -1   # 胡牌来源位置
        self.hand_card_for_settle_show = [[],[]]  # 记录吃碰杠顺序 用于结束游戏显示[[][][][手牌][胡牌]]
        self.guo_hu_num = 0      # 过胡次数
        self.bu_hua_num = 0      # 补花次数
        self.is_ting = 0            # 是否处于听牌状态
        self.is_tian_ting = 0       # 是否 天听，用于牌型判定
        self.is_ren_hu = 0          # 是否 人胡，用于牌型判定
        self.is_tian_hu = 0         # 是否 天胡，用于牌型判定
        self.is_di_hu = 0           # 是否 地胡，用于牌型判定
        self.hu_card_val = 0        # 胡的牌
        self.zi_mo = 0              # 1:表示自摸，0:反之
        self.is_qiang_gang = 0      # 1:表示抢杠，0:反之
        self.qiang_gang_hu_seat_id = -1    # 1:表示抢杠胡作为ID，-1:表示没有
        self.drewed_card_lst = []       # 从开局到最后所有摸过的牌
```

所以判断是否符合杠上开花只需要直接验证倒数第一个动作是否为杠即可，而杠一般为三种：明杠、暗杠、补杠。

5.2 九连宝灯

九连宝灯胡法定义：由一种花色序数牌按1112345678999组成的特定牌型，不能吃、碰、杠牌，见同花色任何一张序数牌即成胡牌。

九连宝灯的具体实现参照上文的搜索方法（根据胡法类型常量定义进行全局搜索）可定位到game/mahjong/models/hutype/eighty_eight/jiulianbaodeng.py中，大体判断逻辑如下：

```
    def is_this_type(self, hand_card, card_analyse):
        # 不是清一色则返 False
        union_card = hand_card.union_card_info
        # 4,5,6 至少有一张
        if union_card[CardType.WAN][0] != 14:
            return False
        if union_card[CardType.WAN][1] != 3:
            return False
        if union_card[CardType.WAN][9] != 3:
            return False

        for i, count in enumerate(union_card[CardType.WAN]):
            if i in [0, 1, 9]:
                continue
            if count < 1:
                return False
        return True
```

九连宝灯的判断需要几个部分：

a. 判断是否为清一色（14张手牌）；

b. 判断该花色的1和9是否大于等于3张；

c. 判断该花色的2-8是否大于等于1张；

在该游戏中，完整的手牌信息都存放在union_card_info（包括吃碰杠的牌），union_card_info的存储结构如下：

{

CardType.WAN: [万牌总张数，一万张数，二万张数，...， 九万张数]

CardType.BING: [饼牌总张数，一饼张数，二饼张数，...， 九饼张数]

CardType.TIAO: [条牌总张数，一条张数，二条张数，...， 九条张数]

CardType.FENG: [风牌总张数，东风张数，南风张数，西风张数， 北风张数]

CardType.JIAN: [箭牌总张数，“中”张数，“发”张数，“白”张数]

CardType.HUA: [花牌总张数，“春”张数，“夏”张数，...，“菊”张数]

}

故判断时结合union_card_info的结构特点来判断会简化胡法判断的大量逻辑。

5.3 碰碰胡

碰碰胡的玩法是由4副刻子（或杠）、将牌组成的胡牌，也即4 *x + 1 *2。

碰碰胡是一种特殊的玩法，如果要胡碰碰胡，则一定需要先满足基本胡中的平胡。同上文一样，搜索跟踪到碰碰胡具体实现在game/mahjong/models/hutype/six/pengpenghu.py中，碰碰胡判断需要考虑的逻辑有：

a. 检查碰的牌中的刻子；

b. 检查补杠、暗杠、明杠（点杠）中的刻子；

c. 校验刻子数是否为4

在实际的程序中，我们可以先判断手牌中是否有吃的牌，如果有吃的牌，则不可能满足碰碰胡，之后再去判断是否满足4个刻子的条件。其核心代码如下：

```
    def is_this_type(self, hand_card, card_analyse):
        chi_cards = hand_card.chi_card_vals
        if chi_cards:
            return False
        j, s, k = card_analyse.get_jiang_ke_shun(hand_card.hand_card_vals)
        if hand_card.peng_card_vals:
            for peng_group in hand_card.chi_card_vals:
                k.append(peng_group)
        if hand_card.bu_gang_card_vals:
            for group in hand_card.bu_gang_card_vals:
                k.append([group[0], group[0], group[0]])
        if hand_card.an_gang_card_vals:
            for group in hand_card.an_gang_card_vals:
                k.append([group[0], group[0], group[0]])
        if hand_card.dian_gang_card_vals:
            for group in hand_card.dian_gang_card_vals:
                k.append([group[0], group[0], group[0]])

        return len(k) == 4
```

hutype目录下，当前各特殊玩法的位置划分是根据胡牌结算时该类型的番数来确定，比如九连宝灯为88番，所以放在eighty_eight目录下。

如果需要添加其他胡法，建议遵守相同规则。

六、麻将用户各类游戏动作实现

在当前游戏架构中， 我们对游戏中所有需要用户操作的行为进行了抽象，归类为用户行为，所有用户行为处理逻辑都放在game/mahjong/models/playeract/目录下。

用户操作的行为有暗杠、补杠、吃、出、点杠、点炮胡、过、碰、测试接口、自摸等。

针对于所有的玩家游戏操作，我们封装了一个基类BasePlayerAct，其主要抽象了玩家游戏操作需要用到的各类成员信息引用及方法，如game_data, desk_id，players, game_config等。

1. 吃动作实现

麻将里吃牌操作一般是针对于顺子而言，比如针对于万字牌，手牌有一万，三万，该玩家上家打出一张二万，此时该玩家便可以选择是否进行吃操作。

一般吃牌都只允许吃上家的牌，吃牌时所需要进行的操作不同的地方麻将理论上可能需要做的事情不一样，所以在具体的设计过程中我们尽可能的进行了微操作|步骤封装。其代码位于playeract目录下的chi.py文件中，吃动作的类定义如下：

```
class Chi(BasePlayerAct):
    def __init__(self, game_data):
        super(Chi, self).__init__(game_data=game_data)
        self.step_handlers = {
            "param_check": self.param_check,                # 参数验证
            "clear_other_act": self.clear_other_act,        # 清除该玩家其他
动作
            "set_data": self.set_data,                      # 设置相应数据
            "record": self.record,                          # 记录玩家动作
            "notify_other_player": self.notify_other_player,# 通知其他玩家
            "clear_lou_hu": self.clear_lou_hu,              # 清除漏胡数据
            "after_chi": self.after_chi                     # 吃动作之后，通
知玩家叫牌
        }

        self.used_cards = []        # 吃牌使用的牌
        self.chi_group = []         # 吃牌的组合
        self.seat_id = -1           # 执行出牌的玩家位置
```

成员变量step_handlers存放的是吃操作所需要做的所有步骤，key为步骤的名称，value为具体步骤对应执行的函数。吃操作需要进行的步骤如下：

参数验证（param_check）：验证玩家吃操作传递过来的参数是否合法；

清除该玩家其他操作（clear_other_act)：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；

设置数据（set_data）：修改吃操作需要修改的全局数据；

记录玩家动作（record)：记录玩家操作，即记下玩家吃操作的操作记录

通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家

清除漏胡数据（clear_lou_hu）：玩家操作后，需要清除漏胡相关的信息

吃操作之后的流程（after_chi）：吃操作之后要进行的操作

当有一些其他特殊的吃操作流程时，我们可以对其中总的一部分步骤进行进一步细化，比如吃操作之后的流程可以进一步细化。

这些步骤的具体执行控制在execute函数中

```
    def execute(self, act_params={}):
        """
        执行吃牌
        :param act_params:
        :return:
        """
        logger.debug(u"吃牌: %s", str(act_params))
        print "CHI EXECUTE self.game_config.player_act_step=",
    self.game_config.player_act_step
        print "CHI_step=", self.game_config.player_act_step.get(Act.CHI)
        for step in self.game_config.player_act_step.get(Act.CHI):
            for name, cfg in step.items():
                ret = self.step_handlers.get(name)(act_params=act_params,
    config_params=cfg)
                if not ret:
                    logger.error("step:%s", step)
                    return
        return 1
```

这些玩家操作的执行先后顺序，以及需要执行哪些操作都在配置文件中可以进行配置。execute函数主体就是根据配置文件按照先后顺序执行相应的各个步骤。

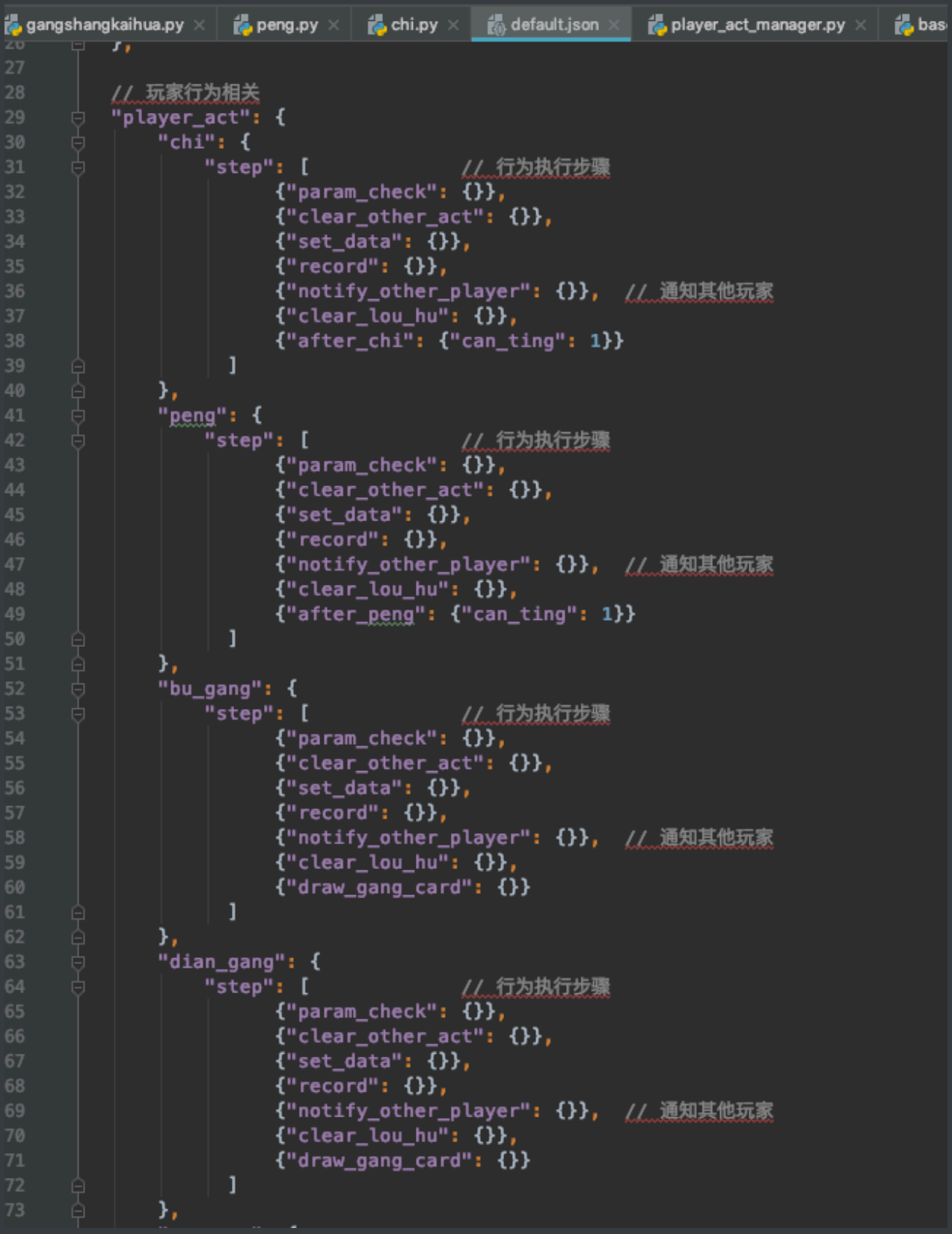
其中设置数据（set_data）部分吃的操作需要做的事有：

1. 删除手牌中用来吃的牌
2. 设定玩家的下一个操作为出牌
3. 碰牌时，移除被吃牌玩家牌墩前已出的牌中该被吃牌；
4. 全局可见的已出牌中添加拿出来吃的两张牌

吃牌逻辑的最后（after_peng）需要根据配置文件做一些收尾操作:

1. 判断游戏中是否允许听牌操作
2. 如果允许听牌，此时开始检查该玩家当前是否可以听，如果可以听，则通知该玩家可以听牌；如果不行，则通知玩家进行下一步操作(出牌)
3. 如果游戏设置中没有听牌操作，则直接通知玩家进行下一步操作

游戏内配置文件位于game_setting/ 目录下，default.json便为默认的游戏内配置信息，其内部玩家行为相关的部分配置如下图：



2. 碰动作实现

麻将里的碰牌操作一般指当有玩家打出一张牌时，某玩家拿出两张相同的手牌，三张牌一起组成一个刻子摆在该玩家的牌堆前。如A玩家打出一万，B玩家拿出手上的两张一万，此时便可以组成一个刻子。碰牌和吃不一样，碰大多没有位置要求，同时其优先级要高于吃操作，即如果A玩家可以吃某一张牌，B玩家同时选择了碰，此时只执行B玩家的操作。 碰操作的执行步骤大体和吃操作类似：

参数验证（param_check）：验证玩家碰操作传递过来的参数是否合法；

清除该玩家其他操作（clear_other_act)：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；

设置数据（set_data）：修改碰操作需要修改的全局数据；

记录玩家动作（record)：记录玩家操作，即记下玩家碰操作的操作记录

通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家

清除漏胡数据（clear_lou_hu）：玩家操作后，需要清除漏胡相关的信息

碰操作之后的流程（after_chi）：碰操作之后要进行的操作

其中设置数据(set_data)部分碰的操作需要做的事有：

1. 删除手牌中用来碰的牌
2. 设定玩家的下一个操作为出牌
3. 碰牌时，移除被碰牌玩家牌墩前已出的牌中该被碰牌；
4. 全局可见的已出牌中添加拿出来碰的两张牌

碰牌操作的其他流程逻辑和吃牌行为类似，此处不再赘述。

3. 明杠动作实现

明杠又称点杠，指某玩家打出一张手牌，其他玩家从手牌中拿出三张一样的牌选择进行杠操作，即玩家A打出一万，玩家B拿出三张一万就可以进行操作。

点杠操作执行的大体步骤如下（根据配置文件配置）：

1. 参数验证（param_check）：验证玩家明杠操作传递过来的参数是否合法；
2. 清除该玩家其他操作（clear_other_act)：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；
3. 设置数据（set_data）：修改明杠操作需要修改的全局数据；
4. 记录玩家动作（record）：记录玩家操作，即记下玩家明杠操作的操作记录
5. 通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家
6. 清除漏胡数据（clear_lou_hu）：玩家操作后，需要清除漏胡相关的信息
7. 杠后摸牌（draw_gang_card）：明杠操作后，需要从牌堆的反向侧（和正常抓牌的方向不一样）摸一张牌；

主要步骤的执行体和吃操作基本一致， 此处着重讲下最后一步杠后摸牌，其核心代码如下：

```
    def draw_gang_card(self, **kwargs):    # 杠后摸牌
        self.draw_card(self.seat_id, is_last=True)
        return 1
```

该部分会调用摸牌函数draw_card，该函数定义为：

```
    def draw_card(self, seat_id, card_num=1, is_last=False):
        """
        玩家摸牌
        :param seat_id: 玩家座位号
        :param card_num: 摸几张牌
        :param is_last: 正向摸牌还是反向摸牌
        :return:
        """
        from game.mahjong.models.systemact.system_act_manager import
    SystemActManager
        return SystemActManager.get_instance(self.desk_id).system_act(
            act_type=SystemActType.DRAW_CARD,
            act_params={"seat_id": seat_id, "card_num": card_num, "is_last":
    is_last})
```

摸牌操作为自动执行的，在麻将游戏中，我们把它归类为系统行为，其具体实现在下一章中会具体说到。

4. 暗杠动作实现

暗杠一般指到某玩家的回合时，该玩家选择拿四张相同的手牌进行杠，一般此时杠的牌不对其他玩家公开（增加游戏趣味性）。

暗杠操作执行的大体步骤如下（根据配置文件配置）：

```
1. 参数验证（param_check）：验证玩家暗杠操作传递过来的参数是否合法；
2. 清除该玩家其他操作（clear_other_act）：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；
3. 设置数据（set_data）：修改暗杠操作需要修改的全局数据；
4. 记录玩家动作（record）：记录玩家操作，即记下玩家暗杠操作的操作记录
5. 通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家
6. 清除漏胡数据（clear_lou_hu）：玩家操作后，需要清除漏胡相关的信息
7. 杠后摸牌（draw_gang_card）：暗杠操作后，需要从牌堆的反向侧（和正常抓牌的方向不一样）摸一张牌；
```

暗杠操作的具体步骤和明杠基本相同，需要注意的一点在于其他玩家时，暗杠操作需要对牌进行隐藏处理：

```
    def notify_other_player(self, **kwargs):  # 记录玩家动作
        act_info = {"seat_id": self.seat_id, "act_type": Act.AN_GANG,
    "card_list": [self.used_card]}
        self.notify_other_player_act_an_gang(self.seat_id,
                                             act_info=act_info,

     max_player_num=self.game_data.max_player_num)
```

其中notify_other_player_act_an_gang的执行内容如下：

```
    1. 遍历桌子上的玩家；
    2. 对于选择暗杠的玩家，推送其暗杠成功的回应；
    3. 对于非暗杠的玩家，推送该桌子上有某玩家暗杠的消息，此时暗杠的牌会被隐藏，此处会将牌的值设为0，客户端收到牌值为0的牌时，则只展示该牌的背面。
    4. 产生一个定时任务，等待玩家操作，超时则托管执行；
```

其函数定义为：

```
    def notify_other_player_act_an_gang(self, seat_id, act_info={},
    interval=1000, max_player_num=4):
        """
        通知其他玩家有人进行暗杠操作
        :param seat_id:
        :param act_info:  {"seat_id": 座位号, "act_type": Act.CHU, "card_list":
    [操作的牌]}
        :param interval:
        :return:
        """
        for i in xrange(max_player_num):
            is_ting = self.players[i].hand_card.is_ting
            if not i == int(seat_id) and not is_ting:
                tmp_act_info = copy.deepcopy(act_info)
                tmp_act_info["card_list"] = [0]
            else:
                tmp_act_info = act_info
            notify_single_user(self.desk_id, i, PUSH_OTHER_PLAYER_CALL_CARD,
    tmp_act_info)
            timer_manager_ins.add_timer(self.desk_id, i, interval,
    call_type=CallbackFuncType.FUNC_AUTO_PLAYER_ACT
```

```
                                    , call_params=act_info)
        return 1
```

5. 补杠功能实现

补杠一般指玩家之前已经碰过某张牌，然后自己又再一次抓到了这张牌，选择进行杠，这种杠法被称为补杠。点杠、暗杠、补杠都是杠，只是组成杠的四张牌中部分牌的来源不同。

补杠操作的大体程序步骤如下（根据配置文件配置）：

```
1. 参数验证（param_check）：验证玩家补杠操作传递过来的参数是否合法（包括是否符合补杠条件）；
2. 清除该玩家其他操作（clear_other_act）：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；
3. 设置数据（set_data）：修改补杠操作需要修改的全局数据；
4. 记录玩家动作（record）：记录玩家操作，即记下玩家补杠操作的操作记录
5. 通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家
6. 清除漏胡数据（clear_lou_hu）：玩家操作后，需要清除漏胡相关的信息
7. 杠后摸牌（draw_gang_card）：补杠操作后，需要从牌堆的反向侧（和正常抓牌的方向不一样）摸一张牌；
```

其中补杠中notify_other_player通知其他玩家模块，函数中会调用notify_other_player_act_executed，最终跳转到工具类game/mahjong/models/utils/notify_playeract.py，该文件类主要用途用来通知玩家，其中封装的三个通知玩家的方法略有区别。

```
    def notify_player(self, desk_id, seat_id, act_info={},
    interval=1000,t_type=TimerType.NORMAL, command_id=PUSH_CALL_CARD,
    code=200):
        """
        通知玩家进行操作，同时开始倒计时
        :param desk_id:  桌子ID
        :param seat_id:  执行动作的玩家位置
        :param act_info: 玩家可进行的操作信息{act_type:params, ...}
        :param interval: 倒计时时间
        :return:
        """

    def notify_some_player(self, desk_id, seat_id, act_info={}, interval=1000,
    max_player_num=4, exclude=[]):
        """
        推送给桌子上的某些用户，用于玩家执行动作后的推送
        :param desk_id: 桌子ID
        :param seat_id: 执行动作的玩家位置
```

```
        :param act_info: 详细动作参数 {act_type:[card_val]}
        :param interval: 延迟时间
        :param exclude: 可以排除推送的作为ID, [seat_id,1]
        :return:
        """

    def notify_some_player_ting(self, desk_id, seat_id, act_info={},
    interval=1000, max_player_num=4):
        """
        推送给桌子上的某些用户，用于玩家执行听牌后的推送
        :param desk_id: 桌子ID
        :param seat_id: 执行动作的玩家位置
        :param act_info: 详细动作参数 {act_type:[card_val]}
        :param interval: 延迟时间
        :return:
        """
```

补杠部分其他逻辑和暗杠类似。

6. 听牌功能

在很多麻将中，会存在一个听牌功能，玩家选择听牌之后，只差一张合适的牌即可胡牌，此时摸牌时，不允许更换手牌，如果不能胡牌，则必须摸啥打啥。

听牌动作的逻辑流程大体和其他动作类似，我们着重关注其中设置数据(set_data)和听牌之后逻辑(after_ting)。设置数据主要流程如下：

1. 记录听牌信息（听的牌）
2. 清空之前缓存的可听牌的缓存信息
3. 更改玩家的状态为听；
4. 如果有天听则进行天听判断（起始发牌后即听牌：两个判断：只摸过一张牌和没有吃碰）

听操作的收尾流程略有特别：

1.通过回调的方式执行出牌操作（听牌操作默认包含了出牌，此处使用回调的方式进行解耦，调用的CallbackManager类解决循环调用的问题）

7. 出牌操作

麻将中的出牌封装成了一个单独的动作，出牌的逻辑和其他动作相差较大，其大体步骤有（根据配置文件配置）：

1. 参数验证（param_check）：验证玩家手牌中是否有该张牌；
2. 清除该玩家其他操作（clear_other_act）：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；
3. 设置数据（set_data）：修改明杠操作需要修改的全局数据；
4. 记录玩家动作（record）：记录玩家操作，即记下玩家明杠操作的操作记录
5. 通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家
6. 检查出牌后其他玩家可进行的操作（check_chu_against）：该玩家打出一张牌后，其他玩家可能吃碰杠胡等等需要检查；

设置数据部分主体流程有：

1. 记录出过的牌
2. 手牌减去出的牌
3. 更新当局游戏最近一张出的牌
4. 更新当局游戏最近是谁出的牌
5. 更新当前牌局上正在操作的牌

检查出牌后其他玩家可以进行的操作，该部分思路如下：

1. 检查其他玩家可以进行的操作（具体会调用到SystemActManager类中，系统行为详细逻辑在下一章中会进行详述）
2. 如果有其他玩家可以对该牌进行操作，则分别通知这些玩家，同时产生相应的延时计时任务；再更新桌子上等下下一个说话者的信息
3. 如果其他玩家不能操作，则通知下家进行摸牌

其核心代码如下：

```
def check_chu_against(self, **kwargs):      # 检查出牌后是否有玩家可操作
    next_seat_id = self.get_next_seat_id(self.seat_id)
    self.check_against(self.seat_id, self.chu_cardval)

    # 通知玩家
    print "check_chu_against:", self.game_data.cur_players_to_act,
self.seat_id, next_seat_id
    if self.game_data.cur_players_to_act:
        for i in xrange(self.max_player_num):
            act_list = self.game_data.get_player_can_speaker(i)
            if act_list:
                # data = {
                #     "command_id": PUSH_CALL_CARD, "seat_id": i,
                #     "act": act_list, "card": self.chu_cardval,
"is_against": 1
```

```
                # }
                # notify_single_user(self.desk_id, self.seat_id,
PUSH_CALL_CARD, data)
                self.notify_player_to_act(i, act_info=act_list,
interval=self.get_act_wait_time(i))
        self.game_data.next_speaker_callback = {"type":
CallbackFuncType.FUNC_DRAW_CARD,
                                                "call_params": {"seat_id":
next_seat_id}}
        return 1
    else:
        # 没有玩家可以操作时通知下家摸牌
        return self.draw_card(next_seat_id)
```

8. “过”功能

“过”在游戏中被抽象成了一个操作，即放弃吃、碰、杠、胡、听等等；

过操作的大体流程如下（根据配置文件）：

```
1. 参数验证（param_check）：验证该玩家是否可以说话；
2. 设置数据（set_data）：更新相关状态信息；
3. 通知桌子进行下一步操作（next_act）：该局游戏进行下一步操作；
```

信息更新set_data定义如下：

```
def set_data(self, **kwargs):      # 设置关键参数
    # 针对过胡操作，如果玩家在听牌情况下选择过,则过户次数+1
    is_ting = self.players[self.seat_id].hand_card.is_ting
    ting_cards = self.players[self.seat_id].ting_info.keys()
    cur_op_card = self.game_data.cur_deal_card_val
    if is_ting and cur_op_card in ting_cards and
self.game_config.pass_hu_double:
        guo_hu_card = self.game_data.last_get_card_val[self.seat_id]
        self.players[self.seat_id].hand_card.guo_hu_num += 1
        act_info = {"seat_id": self.seat_id,
                    "act_type": Act.GUO_HU_DOUBLE,
                    "card_list": [guo_hu_card]}
        self.notify_other_player_act_executed(self.seat_id,
                                              act_info=act_info,

 max_player_num=self.game_data.max_player_num)
    return 1
```

其中需要额外考虑在过胡加倍的情况下的过操作，此时需要额外进行一些判断：

> 判断是否为听牌的操作中选择过，如果是同时游戏有过胡加倍玩法，则更新过胡次数，同时通知其他玩家；

过之后的下一步操作（next_act）实现如下：

```
def next_act(self, **kwargs):  # 通知桌子进行下一步操作
    c_type = self.game_data.next_speaker_callback.get('type')
    call_params = self.game_data.next_speaker_callback.get("call_params")
    print "guo, next_act:", self.game_data.next_speaker_callback
    self.game_data.next_speaker_callback = {}

 CallbackManager.get_instance(self.desk_id).execute(call_func_type=c_type, call_params=call_params)
```

9. 点胡实现

很多麻将中都存在胡其他玩家点炮的玩法，简称点胡。及玩家A打出一张牌X，玩家B手牌加上X可以凑成一种或几种胡牌牌型，此时玩家B即可选择胡牌。

点胡实现的大体步骤有（根据配置文件配置）：

> 1. 参数验证（param_check）：验证最近的操作是否其他玩家出牌；
> 2. 清除该玩家其他操作（clear_other_act）：比如玩家可以吃，碰，此时选择了吃操作，则将该玩家倒计时的任务清理掉；
> 3. 设置数据（set_data）：修改点胡操作需要修改的全局数据；
> 4. 记录玩家动作（record）：记录玩家操作，即记下玩家明杠操作的操作记录
> 5. 通知其他玩家（notify_other_player）：某玩家做出的操作需要通知其他玩家

其中重点关注更新状态信息（set_data）部分，该部分需要做的事为：

> 1. 记录放炮的来源玩家；
> 2. 将胡牌时手牌信息保存起来，以供后续结算使用；
> 3. 记录下胡的牌；
> 4. 计算胡牌结果，胡了哪些牌型，缓存胡牌信息（因为胡牌并不一定就游戏结束）
> 5. 如果游戏有人胡的玩法，则判断是否满足人胡；

胡牌计算会调用上一章说到的hutype下的胡牌判断接口。

10. 自摸动作实现

麻将中一般会支持自摸这种胡牌的模式，即玩家自己摸到一张牌后手牌正好可以满足若干种胡牌玩法。自摸大多情况下在计算时其他玩家都是输方，自摸的玩家为赢方。

自摸的大体流程和上面的点胡类似。在执行胡牌操作时一大区别在于更新数据信息时（set_data）需要检查是否满足天胡或者地胡，庄家起手发牌后自摸为天胡，闲家起手摸到第一张牌后胡牌为地胡。核心判断逻辑如下：

```
    是否天地胡牌
if len(self.players[self.seat_id].hand_card.drewed_card_lst) == 1 and
self._is_first_has_chi_peng():
    if self.seat_id == self.game_data.banker_seat_id:
        # 天胡
        self.players[self.seat_id].hand_card.is_tian_hu = 1
    else:
        # 地胡
        self.players[self.seat_id].hand_card.is_di_hu = 1
```

11. 玩家行为管理器

前面介绍了10种玩家行为，不同玩家行为之间是相互比较独立的，它们之间没有直接的依赖关系。而这些玩家行为的管理我们抽象出来了一个管理类PlayerActManager, 其位于game/mahjong/models/playeract/player_act_manager.py文件中。

根据之前说的方法，理解一个较复杂的类时，首先第一步我们不要去扣细节，只看类有哪些公用方法，PlayerActManager类为一个单例类，其公用接口方法如下：

```
    def cache_act(self, seat_id, act_type, act_params):
        """
        缓存动作,用于判断动作优先级
        :param seat_id:
        :param act_type:
        :param act_params:
        :return:
        """

    def player_act(self, seat_id, act_type, act_params={}):
        """
        玩家发送过来的操作只有在所有玩家的操作都达到后，才会真正决定是否执行
        :return:
        """
```

```
    def execute_player_act(self, seat_id, act_type, act_params):

    def act_chu(self, act_params={}):
        # 出操作处理

    def act_chi(self, act_params={}):
        # 吃操作处理

    def act_peng(self, act_params={}):
        # 碰操作处理

    def act_dian_gang(self, act_params={}):
        # 点杠操作处理

    def act_bu_gang(self, act_params={}):
        # 补杠操作处理

    def act_an_gang(self, act_params={}):
        # 暗杠操作处理

    def act_ting(self, act_params={}):
        # 听牌操作处理

    def act_dian_hu(self, act_params={}):
        # 点炮胡操作处理

    def act_zi_mo(self, act_params={}):
        # 自摸操作处理

    def act_guo(self, act_params={}):
        # 过操作处理

    def test_act(self, seat_id, act, card_list):

    def waite_answer(self, act_params={}):

    def get_cur_player_max_act(self):
        """
        获取当前所有玩家待进行的操作中最高优先级的动作， 当此动作到达,则不需要等待其他动作的回复
        :return:
        """
```

其中以“act_”开头的接口不难猜测出是具体的玩家行为处理函数。全局搜索PlayerActManager类的使用可以定位到在game/mahjong/controls/gamedesk.py中的handler_player_act函数，而handler_player_act函数又是在gamemanager.py中调用。该部分接口从房间和游戏内桌子中的桥接类BridgeController中调用而来，其负责处理游戏过程中的所有用户操作。

我们着重关注于该管理器中的入口函数playeract, 接口核心代码如下：

```
    def player_act(self, seat_id, act_type, act_params={}):
        """
        玩家发送过来的操作只有在所有玩家的操作都达到后，才会真正决定是否执行
        :return:
        """
        # 如果是 watie_answer直接执行
        if act_type == Act.WAITE_ANSWER:
            if not self.game_data.state_machine.is_waite_answer:

     self.game_data.state_machine.change_state(GameingStatus.REV_FIRST_ANSWER)
                self.handler[act_type]()
            return
        # 保存玩家回馈的动作参数变动
        self.game_data.add_player_acted(seat_id, act_type, act_params)
        print "player_act:", seat_id, act_type, act_params
        if act_type not in self.handler.keys() or not self.cache_act(seat_id,
    act_type, act_params):
            logger.debug(u"player_act error:act_type=%s", str(act_type))
            return
        if 0 == len(self.game_data.cur_players_to_act):
            # 所有玩家都已进行过操作，开始执行缓存中的操作，只执行优先级最高的
            max_priority_act = Act.GUO
            need_run_act = {"act_type": max_priority_act, "detail":{}}

                # {"act_type": Act.GUO, "detail":{seat_id: act_params, ...}}
            for tmp_seatid, info in self.game_data.cur_players_acted.items():
                acted_type = info.keys()[0]
                if info[acted_type] >= need_run_act["act_type"]:
                    need_run_act["act_type"] = info[acted_type]
                    need_run_act["detail"][tmp_seatid] = info[acted_type]
            self.game_data.cur_players_acted = {}

            # 钩子是否截获行为判断
            if not self.hook.hook(seat_id, act_type, act_params):
                return 2
            return self.handler[act_type](need_run_act["detail"])
        else:
```

```
            # 動作暫時緩存，等待其他玩家操作
            if act_type == self.get_cur_player_max_act():
                logger.debug(u"获得最高优先级动作，立刻出发执行效果！ 最高优先级动作
    =%s"%act_type)
                ret = self.handler[act_type]
    (self.game_data.cur_players_acted[seat_id][act_type])
            # 動作暫時緩存，等待其他玩家操作
            return 2
```

代码不少，先通过代码中的注释会发现其实逻辑并不复杂：

```
 1. 记录玩家已做过某操作；
 2. 如果所有玩家都已进行过操作，开始执行缓存中的操作，只执行优先级最高的（比如A玩家打出
了一张一万，此时B玩家选择了吃，C玩家选择了碰，D玩家选择了胡，则实际执行的是优先级最高的胡
操作）；
 3. 如果还有玩家未执行操作，此时再判断是否优先级最高的动作已经到达，如果到达，则不用管其
他玩家的操作，直接执行最高优先级的操作；
 4. 如果最高优先级操作未到达，则等待状态；
```

执行玩家操作有两种前提情况：一是所有玩家操作执行完，二是最高优先级操作到达，其执行代码为：

```
    self.handler[act_type](...)
```

即调用类的成员变量注册的函数

```
    self.handler = {
        Act.GUO: self.act_guo,
        Act.CHU: self.act_chu,
        Act.CHI: self.act_chi,
        Act.PENG: self.act_peng,
        Act.DIAN_GANG: self.act_dian_gang,
        Act.BU_GANG: self.act_bu_gang,
        Act.AN_GANG: self.act_an_gang,
        Act.TING: self.act_ting,
        Act.DIAN_HU: self.act_dian_hu,
        Act.ZI_MO: self.act_zi_mo,
        Act.WAITE_ANSWER: self.waite_answer,

    }
```

我们选择其中一个成员函数act_chi看下，其函数定义为：

```
    def act_chi(self, act_params={}):
        # 吃操作处理
        from game.mahjong.models.playeract.chi import Chi
        if not self.acts.get("chi", None):
            self.acts['chi'] = Chi(game_data=self.game_data)
        return self.acts["chi"].execute(act_params=act_params)
```

此处使用了在函数里面导入模块的方法，主要解决管理器和玩家动作模块之间循环引用的问题。吃操作的实际执行便在该处调用。

依次类推，其他碰、明杠、暗杠、补杠、点胡、自摸、听、过、出牌等动作实际调用时都在此处，即以act_前缀开头的方法。

该类中另一个关键接口execute_player_act只有一次调用，来源于game/mahjong/models/callbackmanager.py中的call_player_act函数。继续回溯跟踪可发现实际使用到的地方也只有一处，听牌动作ting.py里的after_ting函数：

```
    def after_ting(self, **kwargs):       # 玩家听牌后
        next_seat_id = self.get_next_seat_id(self.seat_id)
        params = {}
        params["seat_id"] = self.seat_id
        params["act_type"] = Act.CHU
        params["act_params"] = {"card": self.chu_card_val}
        CallbackManager.get_instance(self.desk_id).execute(
            call_func_type=CallbackFuncType.FUNC_PLAYER_ACT,
                                                        call_params=params)
```

听牌操作最后，需要自动打出用户选择的一张牌，此时相当于立即调用出牌操作，所以execute_player_act的功能也就出来了，其使用场景即：给一些需要理解执行某个用户操作的场景进行调用，比如听操作之后的出牌。

七、麻将各类系统行为实现

1. 系统行为管理器

麻将架构中，我们对所有的操作进行了抽象，除了刚才的用户行为类别之外，还有另外一种类别：系统行为。

系统行为的定义是指该行为操作由游戏内部自动执行，不需要玩家参与。该游戏中系统行为包括：检查玩家可以进行的操作、检查癞子、洗牌、发牌、摸牌、定庄、定癞子、结算、游戏结束。

系统行为和玩家|用户行为一样，都会有一个管理器进行统一管理,该管理器位于systemact/system_act_manager.py。 其主要方法声明如下：

```
    def try_settle_hu(self, hu_seat_id, is_zi_mo, hu_type_list, source=-1,
    hook_seat_id=-1):
        '''
        计算可能的胡牌结算番数
        :param hu_type_list: 胡牌的类型
        :param virtual_params: 虚拟未发生的操作参数，用来计算可能出现的胡牌番型
        :return:
        '''

    def system_act(self, act_type, act_params={}):

    def gen_bank(self, act_params):

    def deal_card(self, act_params):

    def draw_card(self, act_params):

    def check_against(self, act_params):

    def settle(self, act_params):

    def game_over(self, act_params):
```

从函数名字可以猜测出除了system_act函数外其他的函数都是某个系统行为的处理方法（其中try_settle_hu方法比较特别，后续再论），其中system_act方法根据系统行为的类型去具体调用相应的系统行为方法。

系统行为类型和对应函数的关系在self.handler成员变量中，定义如下：

```
    self.handler = {
        SystemActType.GEN_BANK: self.gen_bank,
        SystemActType.DEAL_CARD: self.deal_card,
        SystemActType.DRAW_CARD: self.draw_card,
        SystemActType.CHECK_AGAINST: self.check_against,
        SystemActType.SETTLE: self.settle,
        SystemActType.GAME_OVER: self.game_over
    }
```

该类为一个单例类，其中一大用途进行系统行为情景进行解耦，和玩家行为管理器一样使用from ... import ...解决循环引用的问题。

2. 系统行为基类

所有的系统行为都继承于同一个基类BaseSystemAct, 其抽象了各系统行为所需要用到的各类属性和方法。核心定义如下：

```
class BaseSystemAct(object):
    """基础麻将系统操作"""
    def __init__(self, game_data):
        self.game_data = game_data

    @property
    def desk_id(self):
        return self.game_data.desk_id

    @property
    def players(self):
        return self.game_data.players

    @property
    def max_player_num(self):
        return self.game_data.max_player_num

    @property
    def game_config(self):
        return self.game_data.game_config

    @property
    def card_dealer(self):
        return self.game_data.card_dealer

    @property
    def card_analyse(self):
        return self.game_data.card_analyse

    @property
    def banker_seat_id(self):
        return self.game_data.banker_seat_id

    def get_next_seat_id(self, seat_id):
        return self.game_data.get_next_seat_id(seat_id)
```

```
        def get_act_wait_time(self, seat_id, act_type=Act.CHU):
            pass
        def notify_player_to_act(self, seat_id, act_info={}, interval=1000):
            """
            通知玩家进行操作
            :param seat_id:
            :param act_info:  {act_type:params}
            :param interval:
            :return:
            """
            return notify_player_ins.notify_player(self.desk_id, seat_id,
    act_info=act_info, interval=interval)
```

其中get_act_wait_time为获取各玩家行为的操作等待时间，时间数值根据配置文件而来。

3. 检查玩家可以进行的操作

当某玩家出了一张牌后，需要检查其他玩家可以进行哪些操作，该部分我们抽象出为一个系统行为。其实现逻辑位于game/mahjong/models/systemact/check_against.py中。

首先我们关注CheckAgainst类定义，关键在self.check_handler:

```
    def __init__(self, game_data):
        super(CheckAgainst, self).__init__(game_data=game_data)
        self.check_handler = {
            CheckAgainType.CAN_CHI: self.can_chi,
            CheckAgainType.CAN_PENG: self.can_peng,
            CheckAgainType.CAN_DIAN_GANG: self.can_dian_gang,
            CheckAgainType.CAN_DIAN_PAO: self.can_dian_pao
        }
```

该handler中存放检查的操作和对应的函数关系。具体检查哪些操作根据配置文件确定，配置文件在游戏配置目录下（即game_setting/），如下图：

```
// 系统行为相关
"system_act": {
    "check_against": {
        "check_list": [1, 2, 3, 4]                  // 需要检查的操作 CheckAgainType
    },
    "draw_card": {
        "step": [
            {"param_check": {}},
            {"has_enough_card": {}},
            {"set_data": {}},
            {"notify_drew_card": {}},
            {"check_after": {}}
        ],
        "check_list": [1, 2, 3, 4],                 // 需要检查的操作 CheckSelfActTypeCAN_1:可暗杠 2:可补杠 3:可自摸 4:可听
        "bu_hua": 1                                 // 摸牌时是否需要补花
    },
    "settle": {
        "gang_info": {"dian_gang": 1, "bu_gang": 1, "an_gang": 1},  //Act.DIAN_GANG: 1, Act.BU_GANG: 1, Act.AN_GANG: 1
        "fan_config": {         // 胡牌时结算番型增益
            "A": {},
            "B": {
                "zhuang": 1, "dian_gang": 1, "bu_gang": 1, "an_gang": 1
            },
            "C": {},
            "D": {}
        },
    "gen_zhuang_fan": 1             // 跟庄的番数，默认和屁胡基础番数一样
    }
}
}
```

需要检查的操作类型常量定义在game/mahjong/constants/gamedefine.py中，其定义为：

```python
class CheckAgainType(object):
    """某次出牌后，检查其他玩家可进行的操作配置项"""
    CAN_CHI = 1             # 可吃
    CAN_PENG = 2            # 可碰
    CAN_DIAN_PAO = 3        # 可点炮
    CAN_DIAN_GANG = 4       # 可点杠

    @classmethod
    def to_dict(cls):
        return {
            "can_chi": cls.CAN_CHI,
            "can_peng": cls.CAN_PENG,
            "can_dian_pao": cls.CAN_DIAN_PAO,
            "can_dian_gang": cls.CAN_DIAN_GANG
        }
```

检查的核心控制逻辑在execute和_check方法中，其主要逻辑有：

1. 遍历该桌子上所有玩家（排除出牌玩家）
2. 针对于具体的玩家，遍历检查配置文件允许的操作（此处为吃、碰、点杠、点炮）；
3. 如果某玩家可以操作，则注意需要再添加一个过选项；

该部分其他函数为处理检查是否可吃，可碰，可点杠，可点炮等，其中判断能否点杠最为复杂，此处选择点杠来进行剖析。

判断点杠的实现逻辑有：

1. 判断是否可以点杠时需要将判断的牌加到手牌中才能进行判断；
2. 判断是否可以点杠；
3. 去除刚才做点杠判断时添加的手牌；
4. 如果可以点杠，此时需要判断点杠会不会破坏听牌状态！！！
   a. 此时听牌状态会破坏手牌，故需要申请临时变量，做深拷贝
   b. 去除三张杠的牌，判断剩下的牌是否能听
   c. 如果能听，且听牌的结果包含原有听牌结果，则可以点杠；
   d. 如果不能听，或者听牌的结果不包含原有的听牌结果，则此时不能点杠；
   e. 如果可以点杠， 则将其加入某玩家接下来可以操作的动作缓存；

参考代码如下：

```
    def can_dian_gang(self, seat_id, card_val):
        self.players[seat_id].hand_card.add_hand_card_by_vals(card_vals=
    [card_val])
        gang_result =
    self.card_analyse.get_can_dian_gang_info_by_handcard(card_val,
    self.players[seat_id].hand_card)
        self.players[seat_id].hand_card.del_hand_card_by_val(card_val)
        if gang_result:
            if self.players[seat_id].ting_info:
                # 玩家处于听牌状态，判断点杠后会否破坏牌型
                temp_cards = self.players[seat_id].hand_card.hand_card_vals
                temp_cards.remove(card_val)
                temp_cards.remove(card_val)
                temp_cards.remove(card_val)
                temp_cards.append(LAI_ZI)
                ting_cards =
    self.card_analyse.get_can_ting_info_by_val(temp_cards)
                ret = []
                for k in ting_cards.keys():
                    ret.extend(ting_cards[k])
                if not
    set(self.players[seat_id].ting_info.keys()).issubset(set(ret)):
                    # 如果新的听牌结果不包含原有听牌结果，则不能点杠
                    print "seat=%s, card_val=%s, ting_cards=%s , can't ting!" %
    (seat_id, card_val, ting_cards)
                    return 0

            self.game_data.add_player_to_act(seat_id, Act.DIAN_GANG,
    act_params={})
        return 1
```

4. 发牌

发牌此处指初始时给桌子上的所有玩家发13张牌，初始发牌和后续玩家摸牌是不同的操作，庄家14张牌相当于发牌发了13张，然后庄家再摸了一张牌。

初始发牌的逻辑如下：

1. 给桌子上所有的玩家发13张牌；如果是测试环境，则发牌时需要考虑到优先使用测试的牌；
2. 通知桌子上的玩家，给各玩家发了牌（通知A玩家发了13张具体的牌，通知A玩家B玩家发了13张牌，但看不到具体是什么）
3. 如果游戏中需要补花，则遍历玩家的手牌，进行补花操作，通知桌子上玩家；
4. 添加超时回调任务，通知玩家进行下一步（摸牌）操作；

其中我们具体看一下补花的逻辑，是否有补花玩法取决于游戏配置，补花操作逻辑有：

1. 判断传过来的手牌中是否有花牌，没有花牌，则返回手牌本身；
2. 遍历手牌，如果碰到花牌的话，则几张花则从牌堆后补几张手牌；
3. 判断新补的手牌是否又有花，如果有，则继续补花；
4. 将补花数量缓存起来；
5. 通知桌子上各玩家，玩家自己有补牌的详细信息，其他玩家则只有数量，看不到具体的牌值；

其示例代码如下：

```
    def bu_hua(self, seat_id, card_list=[]):
        """检查是否补花"""
        hua_card = []
        tmp_card_list = copy.deepcopy(card_list)
        for c in tmp_card_list:
            if CardType.HUA == Card.cal_card_type(c):
                hua_card.append(c)
                card_list.remove(c)
        if not hua_card:
            return 0
        logger.debug(u"发牌补花: %d", len(hua_card))
        new_cards = []
        temp_hua = copy.deepcopy(hua_card)
        while temp_hua:
            cards = self.card_dealer.draw_cards(num=len(temp_hua),
    is_last=True)
            temp_hua = []
            for c in cards:
                if CardType.HUA == Card.cal_card_type(c):
```

```
                temp_hua.append(c)
                hua_card.append(c)
            else:
                new_cards.append(c)
    card_list.extend(new_cards)
    # 将补花数量添加进手牌中 hua_card_vals 中
    self.players[seat_id].hand_card.hua_card_vals.extend(hua_card)
    for p in self.players:
        if p.seat_id == seat_id:
            notify_single_user(self.desk_id, p.seat_id,
messageids.PUSH_GAME_DEAL_BU_HUA,
                               data={"seat_id": seat_id, "hua_card":
hua_card, "bu_cards": new_cards})
        else:
            notify_single_user(self.desk_id, p.seat_id,
messageids.PUSH_GAME_DEAL_BU_HUA,
                               data={"seat_id": seat_id, "hua_card":
hua_card, "bu_cards": [BLACK]*len(new_cards)})
    return 1
```

5. 摸牌

摸牌操作也被抽象成一个系统行为，摸牌包括吃碰杠后的摸牌，以及正常轮次玩家摸牌。

完成一次摸牌行为需要进行的步骤（根据配置文件得来）有：

1. 操作行为参数验证（param_check）
2. 判断牌堆是否有足够的牌可以摸(has_enough_card)
3. 设置数据(set_data)：更改摸牌操作需要更新的全局信息
4. 通知玩家摸到了牌(notify_drew_card)
5. 摸牌后检查该玩家可以做的操作(check_after)

其代码实现在game/mahjong/models/systemact/draw_card.py中，同样在成员变量step_handlers中存储了步骤名和对应的成员方法。

参数验证部分较为简单，主要判断座位号和摸牌的数目是否合法。

判断牌堆是否牌足够（has_enougn_card）需要考虑几个细节：

1. 不同游戏配置的情况下每一局保留的底牌（牌堆不能被摸的牌）数目不一样；
2. 牌不够的情况下将触发游戏结算和游戏结束流程；
3. 触发游戏结束时将不再进行摸牌剩余的流程；

核心代码如下：

```
    def has_enough_card(self, **kwargs):
        """
        判断牌堆中是否还有足够的牌，返回-1表示牌数不够，不进行下一步操作
        """
        if self.card_dealer.card_count < self.card_num +
    self.game_data.get_min_left_cards():
            logger.debug(u"摸牌牌数不够: %s", str([self.seat_id, self.card_num,
    self.is_last]))
            # 遊戲结算
            SystemActManager.get_instance(self.desk_id).system_act(
                act_type=SystemActType.SETTLE, act_params={"type_list":
    [SettleType.DRAW]})
            # 遊戲結束
            SystemActManager.get_instance(self.desk_id).system_act(
                act_type=SystemActType.GAME_OVER, act_params={})
            return -1
        return 1
```

通知桌子上有玩家摸牌时，需要注意，一般情况（过胡加倍除外）下通知桌子上非摸牌玩家时需要隐藏牌值信息。

摸牌操作的最后需要检查该玩家摸牌后是否可以进行其他操作，比如自摸胡牌、补杠、暗杠等，具体检查哪些操作由配置文件决定，配置文件和之前一样，位于game/game_setting/目录下；检查步骤的核心逻辑有：

1. 如果有补花存在,则不进行下一步的检查操作
2. 遍历待检查的操作配置，确认玩家自身可以做的操作；
3. 如果玩家可以有特殊操作，则：
   操作中添加“过”操作
   通知玩家进行操作
   记录下如果玩家没选择操作，则下一步干什么
4. 如果玩家没有特殊操作，则通知玩家出牌

核心代码如下：

```
    def check_after(self, **kwargs):
        """
        摸牌后检查玩家可进行的操作
        :return: []
        """
        if self.game_config.draw_card_bu_hua and self.bu_hua():
```

```
            # 如果有补花存在,则不进行下一步的检查操作
            return 1

        hand_card = self.players[self.seat_id].hand_card
        can_op_info = {}

        for check in self.game_config.draw_card_check_list:
            ret = self.check_handler.get(check)(hand_card)
            if ret:
                act_type = CheckNameTypeRel.REL.get(check)
                can_op_info[act_type] = ret
                self.game_data.add_player_to_act(self.seat_id, act_type,
act_params=ret)

        if
self.game_data.state_machine.players[self.seat_id].hand_card.is_ting:
            chu_card = self.game_data.last_get_card_val[self.seat_id]
        else:
            chu_card = self.players[self.seat_id].hand_card.hand_card_vals[-1]
        if self.game_data.cur_players_to_act.get(self.seat_id, {}):
            self.game_data.add_player_to_act(self.seat_id, Act.GUO, act_params=
{})
            # 根据不同动作获取不同操作时间
            can_op_act = Act.GUO
            if can_op_info.keys():
                can_op_act = can_op_info.keys()[0]
            # 非WAITE_ANSWER 情况加添加过动作
            if not
self.game_data.cur_players_to_act.get(self.seat_id).get(100):
                can_op_info[Act.GUO] = {}
            self.notify_player_to_act(self.seat_id, act_info=can_op_info,

 interval=self.get_act_wait_time(self.seat_id, act_type=can_op_act))
            self.game_data.next_speaker_callback = {"type":
CallbackFuncType.FUNC_NOTIFY_PLAYER_ACT, "call_params": {
                "seat_id": self.seat_id, "interval":
self.get_act_wait_time(self.seat_id, Act.CHU),
                "act_info": {Act.CHU: {"card": chu_card}}}}
        else:
            # 当前玩家没有任何操作

            self.notify_player_to_act(self.seat_id, act_info={Act.CHU: {"card":
chu_card}},

 interval=self.get_act_wait_time(self.seat_id))
```

```
            self.game_data.del_player_to_act(self.seat_id)
            self.game_data.add_player_to_act(self.seat_id, Act.CHU, act_params=
    {"card": chu_card})

        return 1
```

6. 定庄

在麻将游戏中，会有庄家的存在，游戏开始时，一般通过摇骰来确认庄家是谁，定庄的策略不同游戏不尽相同，此处我们把定庄抽象成一个特定的系统行为。

定庄的算法思路较为简单，主要分为：

```
1. 每轮次的第一局时，由于全新开始，不存在上一把的庄家，此时庄家为摇骰决定；
2. 如果不是初始时，此时又分两种情况：
  如果上一局是流局|荒庄，则当局进行随机定庄，摇骰子决定；
  如果上一局有玩家胡牌，又可以分为两种情况：
      上一局时通炮胡的情况，此时当局庄家由上一句中点炮的玩家担任
      上一句不是通炮胡的情况，庄家由上局中胡牌的玩家担任；
3. 清楚上一局残留的信息；
4. 记录庄家位置，开始洗牌
```

核心代码如下：

```
    def execute(self):
        """
        定庄
        :return:
        """
        logger.debug(u"定庄: %s", str([]))
        dice = []
        if -1 == self.game_data.banker_seat_id:
            # 初始时
            dice = self.get_random_dice()
            self.game_data.banker_seat_id = (dice[0]+dice[1]-1) %
    self.max_player_num
        else:
            if not self.game_data.hu_player_static:
                # 上局为流局/荒庄，则随机定庄
                dice = self.get_random_dice()
                self.game_data.banker_seat_id = (dice[0] + dice[1] - 1) %
    self.max_player_num
            else:
```

```
                source = -1
                for seat_id, params in self.game_data.hu_player_static.items():
                    self.game_data.banker_seat_id = seat_id
                    source = params.get("source", -1)
                if 1 < len(self.game_data.hu_player_static):
                    # 一炮多响，点炮的人做庄
                    if -1 == source:
                        logger.error("gen_banker error: %s",
    str(self.game_data.hu_player_static))
                        raise Exception()
                    self.game_data.banker_seat_id = source
        # 重置胡牌相关信息
        self.game_data.reset_hu_static()

        notify_all_desk_player(self.desk_id, messageids.PUSH_GEN_BANK,
                               data={"bank_seat_id":
    self.game_data.banker_seat_id, "dice": dice})
        # 清理上一局信息
        self.game_data.reset_game_data()
        # 洗牌
        self.card_dealer.shuffle_card()
        # 将庄家位置存入game_data
        self.game_data.last_chu_card_seat_id = self.game_data.banker_seat_id
```

7. 结算

结算操作逻辑较为复杂，该系统操作代码位于systemact/settle.py . 具体详情在下一章详细阐述。

8. 游戏结束

游戏结束同样被抽象成为单一的系统操作行为。触发游戏结束的条件不同游戏不同，一般有牌堆牌不够了，胡牌结束等等；

游戏结束需要处理当局游戏的收尾工作。内容主要包括：

```
    1. 相关状态重置；
    2. 通知当前桌子上的所有玩家游戏结束（包含最终结算的信息）
    3. 通知游戏外面的房间中的对应桌子当局游戏结束
    4. 清理其他数据（部分信息下一局定庄时再清理）；
```

主体逻辑位于systemact/game_over.py，核心代码如下：

```
    def execute(self):
```

```
        """
        游戏结束
        :return:
        """
        logger.debug(u"游戏结束: %s", str([]))
        self.game_data.game_status = GameStatus.OVER

        # 重置状态机
        self.game_data.state_machine.reset_data()
        #

        # 通知玩家游戏结束
        data = {"player_info": {}}

        for x in xrange(self.game_data.max_player_num):
            data["player_info"][x] = {}
            data["player_info"][x]["hand_card"] =
    self.game_data.players[x].hand_card.hand_card_for_settle_show

        notify_all_desk_player(self.desk_id, messageids.PUSH_GAME_OVER,
    data=data)
        notify_desk_game_over(self.desk_id)

        GlobalConfig().reset_test_data()
```

八、游戏结算及统计

1. 结算管理

麻将中结算逻辑变化繁多，麻将结算当前分为几种：胡牌结算（包括点胡结算、自摸结算）、跟庄结算、杠结算（点杠、暗杠、补杠结算）等。不同的结算触发时机不一样，有的是立即结算，有的是游戏结束时才进行结算。

结算逻辑位于game/mahjong/models/systemact/settle.py中，在system_act_manager.py中的settle方法里调用。调用时需要传参type_list, 即告诉结算接口是产生什么类型的结算，结算类型的常量定义在constants/gamedefine.py里的SettleType类型。

结算接口的主体逻辑如下：

```
1. 保存结算相关信息
2. 根据不同的结算类型调用对应的特殊结算实现接口
3. 通知桌子上玩家结算详情
```

核心代码有：

```
def execute(self, type_list=[SettleType.HU]):
    """
    执行结算
    :param params:  {settletype:[], settletype2:[,...], ...}
    :return:
    """
    logger.debug(u"执行结算: %s", str(type_list))
    for i in xrange(self.game_data.max_player_num):
        self.players[i].hand_card.handle_hand_card_for_settle_show()
        # 联合手牌,用于计算胡牌番型
        self.game_data.players[i].hand_card.union_hand_card()

    for t in type_list:
        if SettleType.GANG == t:
            self.settle_gang()
        elif SettleType.GEN_ZHUANG == t:
            self.settle_gen_zhuang()
        elif SettleType.HU == t:
            self.settle_hu()
        elif SettleType.DRAW == t:
            self.settle_draw()

    # 返回结算数据
    notify_settle_data(self.desk_id, self.game_data.settle_data)
```

不同类型的结算计算方式不太一样，下文将逐一阐述。

2. 跟庄结算

一些地方麻将会存在跟庄的玩法：初始发牌结束后，庄家打出第一张牌，此时如果剩下每个玩家都依次打出一张一样的牌，则产生跟庄。此时庄家将输给每位玩家游戏开始时约定好的跟庄番数。

跟庄玩法的结算触发场景大多为立即结算，最终同样会调用到上文的Settle类的execute方法中，具体再跳入实际的跟庄结算逻辑settle_gen_zhuang函数，该函数主要操作任务有：

```
1. 根据配置文件跟庄结算类型对应的番数计算跟庄的输赢，假使跟庄的番数是x，则：
  庄家：输的点数 =  （玩家数-1）* x
  每位闲家：赢的点数 = x
```

3. 杠结算

不论是明杠、补杠还是暗杠，结算时都归类为杠结算，杠结算有的麻将是杠后立即出发，有的是游戏结束时触发，视具体的游戏玩法而定。

杠结算最终会调用到Settle类中的settle_gang方法。其结算是计算公式不同类型杠略有不同。

```
1. 遍历所有的玩家，本别计算其输赢，计算结果缓存到game_data中；
2. 在计算输赢时：
  点杠：点杠一般只有放杠的玩家输，杠的玩家赢相应番数，其他玩家无影响；所以在计算和缓存时需要根据点杠来源来计算，同时结果中需要缓存点杠来源；
  补杠：补杠时赢家同样只有杠牌的玩家，但除此之外的其他玩家都为杠牌输家，即假定补杠的基本番数为x：
    则：  杠牌玩家：赢的番数=   （玩家数-1）* x
        其他玩家：输的番数= x
3. 暗杠：暗杠的输赢计算公式和补杠类似，不过很多麻将中暗杠的番数可能会大于补杠的番数；
```

杠结算的核心代码如下：

```
def settle_gang(self):
    for i in xrange(self.max_player_num):
        for val in self.players[i].hand_card.dian_gang_card_vals:
            tmp_points = [0 for j in xrange(self.max_player_num)]
            gang_fan = self.game_config.settle_gang_info.get(Act.DIAN_GANG)
            tmp_points[i] = gang_fan
            source = self.players[i].hand_card.dian_gang_source.get(val)
            tmp_points[source] = -gang_fan
            self.game_data.add_settle_info(SettleType.GANG,
seat_points=tmp_points,
                                           params={"type":
GangType.DIAN_GANG, "source": source})
        for _ in self.players[i].hand_card.bu_gang_card_vals:
            gang_fan = self.game_config.settle_gang_info.get(Act.BU_GANG)
            tmp_points = [-gang_fan for _ in xrange(self.max_player_num)]
            tmp_points[i] = gang_fan * (self.max_player_num-1)
            self.game_data.add_settle_info(SettleType.GANG,
seat_points=tmp_points,
```

```
                                                params={"type":
GangType.BU_GANG})
        for _ in self.players[i].hand_card.an_gang_card_vals:
            gang_fan = self.game_config.settle_gang_info.get(Act.AN_GANG)
            tmp_points = [-gang_fan for j in xrange(self.max_player_num)]
            tmp_points[i] = gang_fan * (self.max_player_num-1)
            self.game_data.add_settle_info(SettleType.GANG,
seat_points=tmp_points,
                                                params={"type":
GangType.AN_GANG})
```

4. 点胡结算

麻将中结算最复杂的逻辑在于胡牌结算。笔者参考几十款地方麻将几百种不同的胡牌类型番型计算，推导总结归纳出一个相对通用的番数计算公式。

```
某玩家胡牌的计算公式为：
y = (A(x+B)+C) * D
y 为 胡牌最终番数， x为基本胡法番数， 如碰碰胡清一色共4番，x=4
A = A1*A2*...，A为算最终番数相乘的因子积
B = B1+B2+...，B为算番时和基础番数相加的因子和
C = C1+C2+...，C为算最终番时相加的因子积
D = D1*D2*...，D为最外层最终番数相乘的因子积
```

不同的胡牌类型番数计算方法不一样，比如同时满足清一色和碰碰胡两种胡法，两种胡法的番数一般是相加，但也有可能是相乘，此时如果是庄家，则又有可能产生一个额外附加值，比如额外增加1番等等；在二人麻将中，有一种过胡加倍的玩法，每一次过胡后胡牌最后的番数都乘以2。

```
再如四人麻将中，假定A为胡牌玩家，其他玩家为B/C/D，庄家为A，A胡牌时是B玩家放炮。
A胡牌时牌面为“一万，一万，一万，一万，二万，二万，二万，二万，三万，三万，三万，三万，四
万，四万，四万，四万，五万，五万”，其中一万、二万、三万、四万为四个杠子；
此时胡牌类型满足：
  基本胡：屁胡
  特殊胡法：十八罗汉
            清一色
相应的游戏番数定义规则为：
  杠：计算时 一个杠番数+1， 四个杠则+4
  庄家：计算时 番数+1
  不同的特殊胡法间计算时是相乘的关系
  十八罗汉番型：64番
  清一色：4番

此时则B玩家输的番数为
```

```
屁胡1番*十八罗汉64番*清一色4番 + 庄家1番+ 杠4番（如果要计算杠）= 261番
对应的上面公式中    A：A1(十八罗汉)，A2(清一色)
                   C：C1(庄家)，C2(杠)
```

不同玩法对番数的影响最终归类于对A/B/C/D四个常数因子的配置；具体配置来源于配置文件game_setting/default.json，如：

```
        "settle": {
            "gang_info": {"dian_gang": 1, "bu_gang": 1, "an_gang": 1},  //Act.DIAN_GANG: 1, Act.BU_GANG: 1, Act.AN_GANG: 1
            "fan_config": {        // 胡牌时结算番型增益
                "A": {},
                "B": {
                    "zhuang": 1, "dian_gang": 1, "bu_gang": 1, "an_gang": 1
                },
                "C": {},
                "D": {}
            },
        "gen_zhuang_fan": 1              // 跟庄的番数，默认和屁胡基础番数一样
        }
    }
```

比如该配置文件中配置了B因子的番数影响："zhuang", "dian_gang","bu_gang","an_gang", B因子计算时子元素间是相加的关系，代入具体的计算公式最终对胡牌番数产生影响。

胡牌番数计算具体在Settle类的settle_hu函数中，我们快速看下该函数的首层代码：

```
    def settle_hu(self):
        """
        胡牌结算
        某玩家胡牌的计算公式为：
        y = (A(x+B)+C) * D
        y 为 胡牌最终番数， X为基本胡法番数， 如碰碰胡清一色共4番，x=4
        A = A1*A2*..., A为算最终番数相乘的因子积
        B = B1+B2+..., B为算番时和基础番数相加的因子和
        C = C1+C2+..., C为算最终番时相加的因子积
        :return:
        """
        for seat_id, params in self.game_data.hu_player_static.items():
            base_fan =
    self.game_data.hu_manager.get_fan_hu_type_list(params.get("type_list"))
            tmp_points = self.compute_tpm_fan(base_fan, seat_id,

     self.players[seat_id].hand_card.guo_hu_num,
                                                  self.game_config.base_bet)
            hand_card_for_settle_show =
    self.game_data.players[seat_id].hand_card.hand_card_for_settle_show
            self.game_data.add_settle_info(
                SettleType.HU,
                seat_points=tmp_points,
                params={"type_list": params.get("type_list"),
                        "hu_seat_id": seat_id,
                        "hu_fan_count": base_fan,
```

```
                    "guo_hu_count":
    self.game_data.players[seat_id].hand_card.guo_hu_num,
                    "source_seat_id": params.get('source_seat_id', -1),
                    "hand_card_for_settle_show": hand_card_for_settle_show
                    }
            )
```

不难发现，其大体分为几步走：

1. 遍历所有的胡牌结果（部分麻将中可能有多次胡牌），分别计算每次胡牌的番数；
2. 计算时，先根据胡牌类型（可能同时胡多种）获得基本番数；
3. 调用番数计算公式方法compute_tmp_fan求得最终番数；
4. 缓存结算计算结果信息；

我们继续跟进关注其中关键的公式计算方法compute_tpm_fan：

```
    def compute_tpm_fan(self, base_fan, hu_seat_id, guo_hu_count=0,
    base_bet=1):
        """
        计算二人麻将胡牌后点数输赢
        :param base_fan: 番数
        :param hu_seat_id: 胡牌玩家位置
        :param guo_hu_count: 过户次数
        :param base_bet: 本场次底分
        :return:
        """
        points = [0 for _ in xrange(self.max_player_num)]
        for i in xrange(self.max_player_num):
            last_fan = self.get_settle_hu_fan(base_fan, hu_seat_id, i)
            if hu_seat_id == i:
                points[i] = last_fan * base_bet * pow(2, guo_hu_count)
            else:
                points[i] = -last_fan * base_bet * pow(2, guo_hu_count)
        return points
```

在计算某玩家的输赢番数时，针对于单次胡牌赢家一般只有一个，而输家可能有多个，我们计算时先计算每一个输家输的番数，所有输家输的番数累加则为赢家赢得番数。跟进输家输的番数计算方法get_settle_hu_fan:

```
    def get_settle_hu_fan(self, base_fan, hu_seat_id, seat_id):
        a = self.compute_fan_A(hu_seat_id, seat_id)
        b = self.compute_fan_B(hu_seat_id, seat_id)
        c = self.compute_fan_C(hu_seat_id, seat_id)
        d = self.compute_fan_D(hu_seat_id, seat_id)
        return (a * (base_fan + b) + c) * d
```

该方法中调用了4个小方法分别计算上述公式中的A/B/C/D最终数值，求出ABCD后，代入公式可获得最终番数，我们选择其中一个因子A进行分析：

```
    def compute_fan_A(self, hu_seat_id, seat_id):
        fan = 1
        for t in
    self.game_config.settle_fan_config.get(HuSettleParamType.A).keys():
            fan = fan * self.settle_hu_handler.get(t)(hu_seat_id, seat_id,
    p_type=HuSettleParamType.A)
        return fan
```

在公式中A因子中的A1,A2, ...子元素之间的关系为相乘的关系，根据配置文件配置计算时注意基础番数指是1（fan=1），否则计算结果会为0。其中涉及到一个常量HuSettleParamType, 其定义（可全局搜索）在mahjong/constants/game_define.py中。

而B因子计算方法为累加（累加时基础值为0（fan=0））：

```
    def compute_fan_B(self, hu_seat_id, seat_id):
        fan = 0
        for t in
    self.game_config.settle_fan_config.get(HuSettleParamType.B).keys():
            fan = fan + self.settle_hu_handler.get(t)(hu_seat_id, seat_id,
    p_type=HuSettleParamType.B)
        return fan
```

其他C因子的计算和B因子计算方式一致，D因子和A因子计算方式相同。

在计算A,B,C,D四个因子的子元素时，需要判断这些子元素在当局游戏胡牌番数计算中，是否满足其条件，比如胡牌的玩家如果不是庄家，则不会触发庄家加成（假定庄家加成配置在B因子中）。具体的是否符合因子各子元素的条件都封装到了相应的方法中，对应的方法关系在Settle类的settle_hu_handler成员变量里，示例定义如下：

```
    self.settle_hu_handler = {
        SettleHuFan.ZHUANG: self.get_extra_fan_zhuang,
        SettleHuFan.DIAN_GANG: self.get_extra_fan_dian_gang,
```

```
        SettleHuFan.BU_GANG: self.get_extra_fan_bu_gang,
        SettleHuFan.AN_GANG: self.get_extra_fan_an_gang,
        SettleHuFan.HUA: self.get_extra_fan_hua,
        SettleHuFan.JIA_MA: self.get_extra_fan_jia_ma,
        SettleHuFan.JIA_PIAO: self.get_extra_fan_jia_piao,
        SettleHuFan.LIANG_DAO: self.get_extra_fan_liang_dao,
        SettleHuFan.GUO_HU: self.get_extra_fan_guo_hu,
        SettleHuFan.GANG_SHANG_PAO: self.get_extra_fan_gang_shang_pao,
        SettleHuFan.GANG_SHANG_KAI_HUA: self.get_extra_fan_gang_shang_kai_hua,
        SettleHuFan.QIANG_BU_GANG_HU: self.get_extra_fan_qiang_bu_gang_hu
    }
```

胡牌番数计算的核心部分便在于公式的中A,B,C,D四个因子的计算，settle_hu计算出番数后，缓存到当局游戏数据类game_data中，此时结算系统动作执行结束。

5. 自摸结算

自摸结算和点炮胡（点胡）只有一点不同：点胡结算时一般只有一个赢家一个输家，而自摸除了一个赢家外其他的都是输家。比如四人麻将，一人自摸，其他三位玩家都需要输给自摸的玩家同样的番数。

在程序代码的实现中，调用胡牌结算时，需要用到一个参数is_zi_mo，该参数值在game_data.hu_player_static中，最终调用Settle类的compute_last_fan函数（该函数供通用麻将使用，compute_tmp_fan为二人麻将定制方法）。

compute_last_fan中进行胡牌算番，分别计算自摸和点炮胡时各玩家输赢情况并返回。主要代码如下：

```
    def compute_last_fan(self, base_fan, hu_seat_id, is_zi_mo=0, source=-1):
        """
        进行加成后的最终番数
        某玩家胡牌的计算公式为：
        y = (A(x+B)+C) * D
        y 为 胡牌最终番数， X为基本胡法番数， 如碰碰胡清一色共4番，x=4
        A = A1*A2*..., A为算最终番数相乘的因子积
        B = B1+B2+..., B为算番时和基础番数相加的因子和
        C = C1+C2+..., C为算最终番时相加的因子积
        :param base_fan:
        :param hu_seat_id:
        :param is_zi_mo:
        :param source:
        :return: [...]
        """
        points = [0 for _ in xrange(self.max_player_num)]
        win_fan = 0
        if is_zi_mo:
```

```
            for i in xrange(self.max_player_num):
                if hu_seat_id == i:
                    continue
                last_fan = self.get_settle_hu_fan(base_fan, hu_seat_id, i)
                win_fan += win_fan
                points[i] = -last_fan
        else:
            last_fan = self.get_settle_hu_fan(base_fan, hu_seat_id, source)
            points[source] = -last_fan
            win_fan = last_fan

        points[hu_seat_id] = win_fan
        return points
```

6. 通知玩家

上文讲到结算计算完毕后，游戏中会通知外面的房间桌子游戏内进行了计算，即Settle类中调用了notify_settle_data方法，传递桌子号和之前缓存的结算信息。跟踪可发现在房间和游戏的桥接器bridgecontroller.py中的notify_settle_data里调用了房间中桌子管理类desk_mgr的notify_settle_data方法，再继续跳转到game/room/models/room_desk.py。

在房间中的桌子接口内对结算数据进行封装，比如添加用户个人信息，最终一起推送个桌子上的所有玩家。

```
    def notify_settle_data(self, data):
        """
        游戏通知桌子，给予结算数据
        :param data:
        :return:
        """
        # TODO 需要对结算数据进行封装，添加上用户个人信息
        data = data
        self.notify_desk(PUSH_GAME_SETTLE, data)
```

九、模拟真实用户机器人

传统的自动化测试一般是一些独立的http请求，按照约定好的参数和逻辑向服务端发送消息，然后根据消息的返回结果判断正确与否。 而游戏测试和传统的web服务测试相差很大，游戏中由于双端随时交互的存在，其中前后请求之间的交互状态信息极多，逻辑复杂，待测试可选择的分支走向随着业务逻辑的愈加复杂呈指数级增长。

另一方面，传统的压测基本都是针对于某一个具体的接口进行测试，这种测试通常只能测试接口的性能怎样，每秒能承受多少并发，响应延迟是多少，接口是否可以用。

要想获得比较好的测试效果，及时发现服务逻辑中的各种必现和偶现bug，我们需要找到一种新的测试思路和方法。经过大样本的游戏用户行为分析后，我们发现当用户规模达到一定程度后，用户的操作行为呈现一定程度的正太分布，比如一万名用户同时在线，可能其中6000人正在玩游戏，1500人在游戏大厅，100人正断线，100人退出，100人查看用户详情等等。

1. 设计思路

此时，我们便可以设计一套自动化交互测试程序，按照大体的概率分布最大可能模拟用户行为，因为游戏通信是长连接，也即测试时一个长连接模拟一个用户，用户在每个消息请求后的下一步操作根据具体的概率分布区进行选择，比如登录后，以60%的概率进入游戏大厅，20%的概率退出游戏，10%的概率断线，10%的概念断线重连后进入游戏。用户的多个行为之间时间间隔同样呈现一定的随机时间间隔。

这样随着测试模拟的用户越来越多，测试程序整体将越来越接近真实的海量用户行为，解决用户交互问题的同时又可以尽可能多的发现各种隐藏的疑难bug。

2. 机器人设计

机器人相关测试程序逻辑都在test目录下，核心代码在test/robot.py中。模拟游戏用户的机器人和服务端进行通信，此时我们需要遵循和服务端一样的通信协议，使用websocket进行通信连接，包括打包和解包的方式、加解密的方式等。

为了将测试模块更好的和其他部分进行解耦，我们将websocket通信和消息加解密模块在测试部分重新进行了封装。从代码中不难发现robot.py中的主类RobotClient继承于BaseTestClient。

2.1 基础通信及结构定义

BaseTestClient中封装了消息的打包（pack）和解包(unpack)，我们先看其类定义和成员变量：

```
class BaseTestClient(object):
    def __init__(self, config_file='robot.json'):
        self.all_config = EspoirJson.loads(config_file)
        self.response_seq = self.all_config.get("response_seq")
        self.encode_type = self.all_config.get("encode_type")
        ip = self.all_config.get('ip')
        port = self.all_config.get('port')
        address = (ip, port)
        websocket.enableTrace(True)
        self.client = websocket.WebSocketApp(
            url='ws://%s:%s' % address,
            on_message=self.on_message,
            on_open=self.on_open,
            on_close=self.on_close,
```

```
                on_error=self.on_error
            )

            self.fmt = 'iii'
            self.version = 0
            self.head_len = struct.calcsize(self.fmt)
            self.aes_encoder = AesEncoder(encode_type=self.encode_type)
            self.user_id = self.all_config.get("user_id")
            self.user_name = self.all_config.get("user_name")
```

self.client为封装的websocket客户端对象，里面定义了连接建立、断开、正常通信、出错等方法。该类为基类，这些封装的方法在其子类中将会被子类的同名方法覆盖，其中run方法为测试机器人的启动入口。

测试模块目录下robot_x.py为机器人测试启动文件，之所以弄多个，是为了方便在没有客户端的情况下使用机器人调试服务端游戏接口（如果使用多线程，则调试时看日志比较麻烦，效率不高）。这里以robot_1.py为例，其主体代码为：

```
from twisted.internet import reactor
from robot import RobotClient


if __name__ == '__main__':
    # input_thread.start()
    # reactor.suggestThreadPoolSize(100)
    for i in range(0, 1):
        user_id = 1
        robot = RobotClient("robot_1.json")
        # reactor.callInThread(robot.run)
        reactor.callInThread(robot.run)
    reactor.run()
```

为了方便，机器人启动直接使用了twisted框架中的reactor模块，借助于其callInThread方法将机器人放在线程中运行。机器人的主体类为RobotClient，位于robot.py中。

robot.py中有两个类定义，其中MyThread类为另一种测试程序启动方式，此处暂不论，我们先看主体类RobotClient，其类定义如下：

```
class RobotClient(BaseTestClient):
    def __init__(self, config_file):
        BaseTestClient.__init__(self, config_file=config_file)

        # super(RobotClient, self).__init__()
```

```
        self.response_seq = self.all_config.get("response_seq")
        self.sendhandler = {
            100002: self.send_100002,
            100010: self.send_100010,
            100100: self.send_100100,
            100101: self.send_100101,
            100102: self.send_100102,
            100104: self.send_100104,
            100110: self.send_100110,
            100111: self.send_100111,
            100103: self.send_100103,
            100120: self.send_100120,
            9999: self.send_9999
        }

    def on_open(self):
        print "on_open!!!"
        # self.send_data(100002, {"userid": self.user_id, "pass":
self.all_config.get("USER_PASSWORD")}, ws)
        global s_time
        s_time = time.time()
        self.send_100002(self.client)

    def on_message(self, message):
        ...
        self.receivehandler(self.client, command, data)

    def receivehandler(self, ws, key, params):
        ...
        return self.sendhandler.get(rconfig.get("msg")[random_index])(ws,
params)

    def send_data(self, commandID, data, ws):
        print u"*******发送数据*********: ", commandID, data, ws, "||"
        s = self.pack(data, commandID)
        print [s]
        ws.send(s, opcode=ABNF.OPCODE_BINARY)

    def send_100002(self, ws, params={}):
        '''  登录
        '''
        print "@" * 33, 'this is %s' % inspect.stack()[1][3]
        passwd = self.all_config.get("password")
        data = {"user_id": 1, "passwd": passwd}
        self.send_data(100002, data, ws)
```

```python
    def send_100010(self, ws, params={}):
        """断线重连"""
        data = {"user_id": self.user_id}
        self.send_data(100010, data, ws)

    def send_100100(self, ws, params={}):
        """
        玩家准备
        """
        data = {"user_id": self.user_id, "ready": 1}
        self.send_data(100100, data, ws)

    def send_100101(self, ws, params={}):
        """创建好友桌"""
        data = {"user_id": self.user_id}
        self.send_data(100101, data, ws)

    def send_100102(self, ws, params={}):
        """加入好友桌"""
        data = {"user_id": self.user_id, "desk_id": 100000}
        self.send_data(100102, data, ws)

    def send_100103(self, ws, params={}):
        """玩家退出桌子"""
        data = {"user_id": self.user_id}
        self.send_data(100103, data, ws)

    def send_100104(self, ws, params={}):
        """玩家加入匹配场"""
        data = {"user_id": self.user_id}
        self.send_data(100104, data, ws)

    def send_100110(self, ws, params={}):
        """解散桌子"""
        data = {"user_id": self.user_id}
        self.send_data(100110, data, ws)

    def send_100111(self, ws, params={}):
        """解散房间应答"""
        data = {"user_id": self.user_id, "agree": 1}
        self.send_data(100111, data, ws)

    def send_100120(self, ws, params={}):
        """玩家自定义配置"""
```

```
            data = {"user_id": self.user_id, "custom_config": {}}
            self.send_data(100120, data, ws)

        def send_9999(self, ws, params={}):
            pass
```

从类的成员方法名我们可以猜测send_***和具体的各类型消息有关（实际情况确实是用来模拟不同类型的消息请求，如100002代表登录），on_open为websocket连接建立时调用的方法，on_message为服务端推送过来的消息请求处理方法，也即消息处理入口。

2.2 通信配置和流程切换机制

我们再来看on_message函数的主体代码：

```
    def on_message(self, message):
        length = self.head_len
        unpackdata = self.unpack(message)  # 这一行的作用
        print "接收数据：", unpackdata
        if unpackdata is None:
            print '!'*20, 'unpack_data is None!'
            return
        command = unpackdata.get('command')
        rlength = unpackdata.get('length')
        data = unpackdata.get("data")
        n = random.randint(1, 3)

        self.receivehandler(self.client, command, data)
```

该接口对接收到的消息进行解包后，得出消息请求ID、消息包长度、消息正文。最后调用receiveHandler函数对具体的消息进行处理：

```
    def receivehandler(self, ws, key, params):
        if params and isinstance(params, str):
            params = ujson.loads(params)
        global responseConfig
        print u"接收处理:", key, [params]
        rconfig = self.response_seq.get(str(key), None)
        if not rconfig:
            logger.error("index(%s):Error: response config error! key=%s", 1,
    str(key))
            return

        if not rconfig.get("msg", None):
```

```
            # logger.warning("response has not receiver msg handler")F
            return
        # if key == 100002 and params.get("info"):
        #     return self.send_100801(ws, params)
        random_index = random_weighted_choice(rconfig.get("weight"))
        # print "@"*20, random_index, key, params, rconfig.get("msg")
        return self.sendhandler.get(rconfig.get("msg")[random_index])(ws,
    params)
```

receiveHandler函数体中首先读取配置文件，根据配置文件中配置得出针对于接收到的该指定消息ID，接下来可能要做的操作，最后调用带权随机后的操作消息对应的函数。

2.2.1 流程配置

机器人的配置文件为test/robot_x.json，其中包括请求的服务端地址和端口，消息通信是否加密，测试的用户账号信息，请求的流程执行顺序等，示例配置文件如下：

```
{
    "ip": "127.0.0.1",
    "port": 10000,

    //请求执行顺序
    "response_seq": {
        "100002": { "msg": [100104], "weight": [100]},       //登录
        "100010": { "msg": [9999], "weight": [100]},         //断线重连
        "100100": { "msg": [9999], "weight": [100]},         //玩家准备
        "100101": { "msg": [9999], "weight": [100]},         //创建好友房
        "100102": { "msg": [9999], "weight": [100]},         //加入好友房
        "100104": { "msg": [100103], "weight": [100]},       //加入好友房
        "100110": { "msg": [9999], "weight": [100]},         //解散房间
        "100111": { "msg": [9999], "weight": [100]},         //解散房间应答
        "100103": { "msg": [9999], "weight": [100]},         //玩家退出桌子
        "100120": { "msg": [9999], "weight": [100]},         //自定义配置
        "101001": { "msg": [9999], "weight": [100]},         //推送玩家叫牌
        "101002": { "msg": [9999], "weight": [100]},         //推送玩家摸牌
        "101003": { "msg": [9999], "weight": [100]},         //推送游戏结束
        "101004": { "msg": [9999], "weight": [100]},         //推送定庄信息
        "101005": { "msg": [9999], "weight": [100]},         //推送发牌信息
        "101006": { "msg": [9999], "weight": [100]},         //推送游戏结算
        "101100": { "msg": [9999], "weight": [100]},         //推送玩家点数发生变化
        "101101": { "msg": [9999], "weight": [100]},         //推送玩家在其他地方登录
        "101102": { "msg": [9999], "weight": [100]},         //推送桌子解散结果
        "101103": { "msg": [9999], "weight": [100]},         //推送有玩家请求解散桌子
        "101104": { "msg": [9999], "weight": [100]},         //推送玩家对解散桌子的响应
        "101105": { "msg": [9999], "weight": [100]},         //推送玩家退出桌子
        "101106": { "msg": [9999], "weight": [100]},         //推送玩家准备/取消准备
        "101107": { "msg": [9999], "weight": [100]},         //推送玩家加入房间
        "101108": { "msg": [9999], "weight": [100]},         //推送玩家更改了配置
        "101109": { "msg": [9999], "weight": [100]}          //推送玩家断线重连
    },
    "encode_type": 0,                //是否加密，0未加密，1加密
    "user_id": 1,
    "user_name": "test1",
    "password": "123456"
}
```

其中请求执行顺序配置解析为：key为消息ID，如“100002”意为收到100002登录消息请求时接下来要做的事配置为

{"msg": [100104], "weight": [100]}, 其中msg键值存放的是接下来可能要做的消息ID，weight键值存放的是msg对应的操作执行的概率权重。比如{"msg": [100104， 100101], "weight": [100，200]} 代表接下来做的操作有100/（100+200）的概率做100104操作，200/300的概率做100101的操作。上文中msg键值列表只有一个值的情况下，即代表接下来一定会执行100104操作。

2.2.2 带权随机

收到某个消息后，在进行接下来操作的选择时，我们根据配置文件中的带权概率进行随机，此时会用到一个带权随机算法，核心代码如下：

```
    def random_weighted_choice(weights):
        """
        带权随机
        :param weights: []， 里面存放随机的权重， 如[100,300,200]
        :return: int, range:[0, len(weights)) , 返回随机到的元素序号， 即随机到第几个
        """
        rnd = random.random() * sum(weights)
        for i, w in enumerate(weights):
            rnd -= w
            if rnd < 0:
                return i
```

其中run方法为测试机器人的启动入口。

该算法核心思想为随即一个[0, 权重值总和）范围内的整数，根据其具体数值分布区间判断其对应随机到列表的第几个元素，如：

weights = [100, 300, 200], 即在[0, 600) 中随机，随机后的数值在[0, 100)中对应随机到第0个元素，[100, 400)中对应随机到第1个元素，[400, 600)对应随机到第2个元素。 如果随机后的数值为387，则返回数值所在区间对应的序号1.

2.2.3 函数注册及消息分发

前文我们在看RobotClient类时，发现了很多send_xxx的方法，在类的成员变量里有一个成员self.sendhandler, 里面存放了消息ID和对应的函数执行关系。客户端收到服务端传递过来的消息请求后，带权随机到下一步消息操作，然后从self.sendhandler中获取到对应的函数执行方法。

如果后续要添加新的消息类型，则在定义相应的函数后，需要在self.sendhandler进行注册。为了最大可能复用代码，在所有的send_xxx中抽象出了send_data方法逻辑，所有最后实际向服务端发送数据的过程都封装在该函数中，核心代码如下：

```
    def send_data(self, commandID, data, ws):
        print u"*******发送数据*********: ", commandID, data, ws, "||"
        s = self.pack(data, commandID)
        ws.send(s, opcode=ABNF.OPCODE_BINARY)
```

注册的消息方法中，有一个消息比较特别，为9999消息，该消息ID代表休止符，即该消息不做任何操作，不再向服务端发送消息。

3. 麻将服务端客户端联调测试

实际开发过程中，很多时候我们再做后端接口测试时，不一定会有一个完整的客户端来供我们进行测试。为了更好地模拟交互过程，同时测试我们自己后端的服务接口，我们可以利用上文的模拟机器人进行接口测试。所需要做的只需要两步：

1. 更改配置文件（主要设定消息执行顺序）
2. 在接收消息的入口处添加消息日志，方便跟踪维护

下文将以加入房间消息接口测试为示例，阐述如何使用机器人对其进行测试。

3.1 配置文件修改

快速加入房间的消息ID为100104，该消息的前提条件是用户已登录，即在100002消息返回成功后，此时大体的消息流程为：

1. 客户端连接后首先发送登录消息请求100002；
2. 服务端收到请求处理结束后向客户端推送100002的处理结果；
3. 客户端收到100002登录消息返回成功的消息，然后发送加入房间请求（100104）；
4. 服务端收到100104请求，处理结束后推送100104请求到客户端；
5. 客户端收到服务端返回的100104处理结果，输出特定的日志调试信息；

消息的执行顺序我们可以在配置文件中进行配置，此处只需要修改"response_seq"键值的两小处：

```
"100002": { "msg": [100104], "weight": [100]},      //配置登录消息后的下一步请求为加入房间100104
...
"100104": { "msg": [9999], "weight": [100]},       //收到加入房间消息的响应后，执行9999消息
```

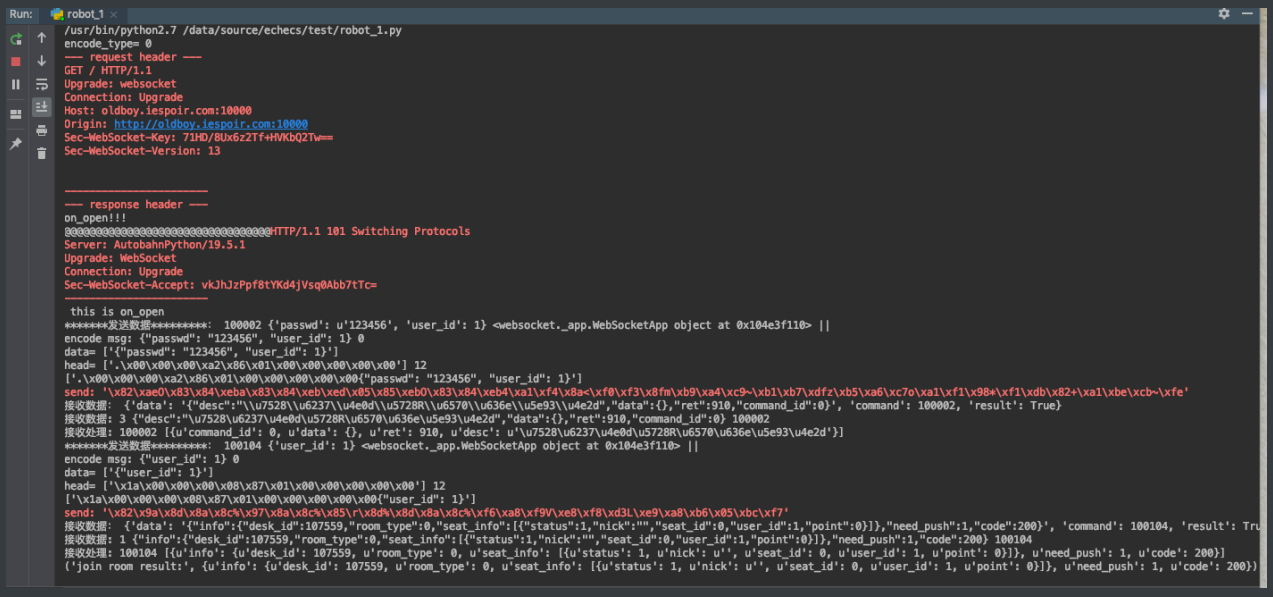
前文有说到在配置文件中当msg的键值只有一个时，即随机相当于无效，下一步一定会执行指定的操作，此时接口测试便是其应用场景之一。收到服务端传递回来的加入房间请求后，为了排除后续信息的干扰，此处我们把接下来的操作设置为9999，即停止操作，相当于客户端发送消息暂停，方便我们调试。

3.2 接口消息跟踪

如果我们要跟踪调试加入房间接口消息，我们可以在机器人接收服务端消息入口(robot/on_message())方法中将接收到的消息内容打印出来；同时在详细的消息处理函数中receivehandler中添加消息判断，如：

```
    if key == 100104 and params.get("info"):
        print("join room result:", params)
```

下图为测试联调时的部分日志输出结果：



从图中最后不难发现该测试中加入房间接口最后的返回结果，其状态码为200，同时有加入的房间详细信息，消息结果符合预期。

后记

自七月初草定撰写一本麻将开发的课程来，前后颠簸下，甚为惭愧，两月来总算出了第一个版本。麻将项目在棋牌和休闲游戏中属于复杂度最高的游戏类型之一，麻将玩法千变万化，虽然本文部分地方在阐述二人麻将，但实际整个麻将后端是多人麻将通用的，只需要做极少部分配置和代码的改变便可以适应一款四人五人等麻将开发的需求，有兴趣和需求的读者不妨尝试下。

时间因素的原因，文中可能有疏漏之处，还望大家海涵，请轻踩~ 后续生活中，个人将对麻将开发代码进行进一步的封装优化，可能会尝试基于一套新的游戏引擎框架进行开发，但主体的麻将逻辑将沿用本文中的设计思想，有机会再和一起探讨和分享。

谢谢各位读者百忙之中能阅读此文，愿您能有所得，有所获~ Best wishes to you ! ^_^

2019年09月18日 菩提