基于动作监听机制和角色权限控制的系统一瞥——以游戏作为一个嵌入式视角

梁子，软件学院，S0083，3120305385

日期：Tue Jun 15 10:20:36 2021

目录

1 绪论

1.1 写在最前面的话

1. 本报告的绪论内容完全不过是一个设想，没有在本次实验中实现。笔者只是基于该场景进行一个基本的展示。
2. 本报告中所有代码均基于 python 编写，python 不算是一门严肃的工业编程语言，但对快速实现和验证一些思路很有裨益。诸如“豆瓣”等网站后台或“骑马与砍杀”等著名游戏，都是基于 python 实现的。
3. 本文的界面太简陋了，基于 PYQT 编写。后来，笔者学会了前端，发现前端写界面简单太多了，同时也更灵活。

1.2 让同一款游戏“千人千面”：在一款游戏内部进行分析和推荐的可行性

关于游戏的灵活性，一直都是游戏行业广为关注的主题。为了让游戏更仿真、世界观更大，无数游戏厂商在游戏里铺陈其世界观，同时又为了防止游戏过于复杂而简化使用。这种 trade off 在游戏界广为传承，并且，最近新出的各类游戏，都在一定程度上以“灵活”、“自适应”作为其卖点。

比如，旨在构造“全新武侠世界”的手游《楚留香》曾经宣称构造了一个完整的武林世界，每个人都可以在里面找到自己的位置。而在实际体验上，这种说法不过是简单地将 RPG、养成、社交、偷菜、采集等各种类型的游戏杂糅在了一起罢了。再有另一些游戏，通过在剧情里设计不同的选择，构造一棵巨大的剧情树，而这一类的剧情树不过是实现基于规则定义好的剧情罢了。当符号主义和联结主义有效结合时，似乎这种局面才能进行改变。而本文，则是单纯考虑将推荐应用在游戏里。* 面对变得越来越复杂、越来越难以全然了解的游戏世界，如何通过推荐的方式，让玩家沉浸在游戏中一种或若干种的功能玩法里面，当观察到玩家对其失去兴趣后，再通过推荐让玩家更多地了解游戏里的其他世界。这样的一个功能，目前还没有游戏涉及到。那么它是可行吗？*

1.3 基本技术需求

实现这样的一种功能，大概需要如下几种技术：

1. 精细的数据采集。应当对每一个用户的数据进行精细的采集，除了数据库中常见的等级、充了多少钱、玩了多久等粗粒度的信息之外，更应该是：它玩哪些功能的次数最多？在哪些界面上驻留的时间最多？在聊天里谈论什么东西最多？本报告主要是对采集技术的分析。
2. 自定义的技术。应当让玩家根据自己喜欢的功能，自定义自己的前端。同时应当开放平台，供一些极客设计特殊的页面。所有的这一切本质上是对 javascript 对象的处理，所以，基于 javascript 对象设计良好的编辑器也是必须的。
3. 推荐技术。
4. 游戏本身。

1.4 本实验报告仅仅迈出了最微小的一步

本文只对第一点进行讨论，绝不可能设计出来一个游戏。本文的目的不在于设计这样一个游戏，在于了解这一套机制，以至于有一天需要的时候可以做到。当然，课程上的这些功能都是较为基本的。

2 一个玩具级 RBAC 库的实现

RBAC 源于对系统中不同角色进行体系化管理的实践，因而，RBAC 的重点在于角色。角色充当了用户请求端和响应端的桥梁，因而可以使得权限控制、对象化重载成为可能。一个完善的 RBAC，本身就是一个系统。粗略来看，主要可以分为景点的 RBAC 模块，以及在此基础上的对 RBAC 模块本身的权限控制管理，以及一些自定义权限的融入。

首先对经典的 RBAC 模块进行介绍。

2.1 经典 RBAC

2.1.1 user，role，action

经典 RBAC 中，主要包含三个层面的对象，分别是用户、角色和动作。其中用户和角色可以理解为数据对象，而动作可以理解为规则对象。在本库中，前二者基于字符串进行定义，动作则是一个指向对象用户的特定方法的指针。

当拥有了用户、角色、动作三者时，就可以定义彼此之间的关系矩阵，并得到相关的哈希表进行快速的查询。该部分的代码如下所示：

```
self.user_roles_dict={}
  for index, user in enumerate(self.user_list):
      self.user_roles_dict[user]=[]
      index_role_list=self.user_role_matrix[index]
      for jndex, ele in enumerate(index_role_list):
    if ele-0==0.:
        continue
    else:
        self.user_roles_dict[user].append(self.role_list[jndex])

  print(">>>Begin to construct map from role to actions.")
  self.action_roles_dict={}
  for index, action in enumerate(self.action_list):
      self.action_roles_dict[action]=[]

      index_roles_list=(self.role_action_matrix.T)[index]
      for jndex,ele in enumerate(index_roles_list):
    if ele-0==0.:
        continue
    else:
        self.action_roles_dict[action].append((self.role_list[jndex]))
```

从中可以看出，通过输入 user 与 role 的关联矩阵、role 与 action 的关联矩阵，RBAC 管理器会维护从 user 到 action 的查询表，从而正确快速地完成查询。

那么，user 与 role 的关系，role 与 action 的关系，应当通过何种方式定义呢？

最简单的方式是通过手工输入，但这种方式效率过低，并且，当面对较大的 user、role、action 数目时，难以维护和管理。面对这种问题，基于规则的自动生成、基于数据库和配置文件进行管理都是必不可少的解决方案，后面将会对其进行介绍。

2.1.2 增删改查

此处的增删改查都是面向 RBAC 数据而言的。当然，关于这些操作本身的权限问题，将在对 RBAC 的管理中进行介绍。增删改查的操作，本质上是改变了 RBAC 所维护的三大对象的列表，同时将两个稀疏矩阵进行修改，最终在此基础上重新加载映射 hash 表，以等待查询。此处将部分代码作为示例展示：

```
def addUser(self,user,user_roles_list=None, user_roles_dict=None):
    """add ‘user‘ to RBAC system, with user-role-list, if not, use user-roles-dict"""

    self.user_list.append(user)
    user_role_lss=self.array2lists(self.user_role_matrix)
    if user_roles_list is not None:
  user_role_lss.append(user_role_lss)
    else:
  user_role_lss.append(user_roles_dict[user])
    self.user_role_matrix=np.array(user_role_lss)

    self.reloadForMap()

def addAction(self, action,cannot_action_list, action_roles_list=None, action_roles_dict=
    None):
    """add ‘action‘ to RBAC system, with action-roles-list, if not, use action-roles-dict"""


    self.action_list.append(action)
    self.cannot_action_list.append(cannot_action)

    action_role_lss=self.array2lists(self.role_action_matrix.T)
    if action_roles_list is not None:
  action_role_lss.append(action_role_lss)
    else:
  action_role_lss.append(action_roles_dict[action])
    self.role_action_matrix=np.array(action_role_lss).T
    self.reloadForMap()

def addRole(self, role, role_users_list, role_action_list):
    """add ‘role‘ to RBAC system, with role-users-list, if not, use role-action-dict"""

    ## add role
    self.role_list.append(role)

    ## update user role matrix
    user_role_lss=self.array2lists(self.user_role_matrix)
    user_role_lss.append(role_users_list)
    self.user_role_matrix=np.array(user_role_lss)

    # updat role action matrix
    self.role_action_matrix=np.array(self.array2lists(self.role_action_matrix).append(
        role_action_list))

    self.reloadForMap()
```

```
def removeUser(self,user):
    """remove ‘user‘ from RBAC system"""

    index=self.user_list.index(user)

    self.user_list.remove(user)
    self.user_role_matrix=np.delete(self.user_role_matrix,index,axis=0)
    self.reloadForMap()

def removeAction(self,action):
    """remove ‘action‘ from RBAC system"""

    index=self.action_list.index(user)

    self.action_list.remove(user)
    self.role_action_matrix=np.delete(self.role_action_matrix,index,axis=1)
    self.reloadForMap()

def removeRole(self, role):
    """remove ‘role‘ from RBAC system"""

    index=self.role_list.index(role)

    self.role_list.remove(role)
    self.user_role_matrix=np.delete(self.user_role_matrix,index,axis=1)
    self.role_action_matrix=np.delete(self.role_action_matrix,index,axis=0)
    self.reloadForMap()
```

2.1.3 形成配置结果：存储与加载

当系统比较简单时，直接将配置信息写入代码中即可解决问题。当系统变得复杂后，将配置信息保存下来，并且下次使用时可以自动加载，就变得十分重要了。当系统变得更加复杂，三大对象的数量级使得单个文件都变得臃肿之后，数据的实现才有必要。本文面对的是中间复杂度的情形。在这种情形下，作者认为，*如果 user、role、action 的数量非得要经由数据库进行管理，那么说明这里没有进行足够的抽象。* 这是因为：

维护 user、role、action 三者的对应关系是需要人工设计的，即使是基于生成，也是人工设计规则。因此，这样的一种 0-1 矩阵必须具备直白的人类可以理解的复杂度。举例而言，在一个教学管理系统里，学生们的 ID 均对应者同样的身份——学生。在进行注册时，RBAC 可以通过两种思路实现：

1. 在后台设计一种规则。使得所有的这样一类 ID 都映射到一个名为 stu 的 role 上；
2. 在后端设计一种规则，将这些 ID 按照其特征（如本科生、硕士生，男的，女的）映射到 user 上，然后将每一个 user 映射到 x 个 role 上，每一个 role 背后代表了一类角色。

   第一种设计比较通用，也比较简单，符合扁平化设计的规则。本文主要为第二种思路服务，因为通过这种方式，可以实现现实场景无关因素和 RBAC 权限控制的解耦，后续的 RBAC 控制和维护都变得简单了。本文的场景将佐证这一观点。

   下面是一个简单的 JSON 文件的示例：

```
{"users": ["id001", "id002", "id003"],
  "roles": ["admin", "lv3", "lv2"],
  "actions": ["money10000", "moneyPlus1", "automaticPlus1"],
  "cannot_actions": ["say_cannot", "say_cannot", "say_cannot"],
  "user_role_matrix": [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]],
  "role_action_matrix": [[1.0, 1.0, 1.0], [0.0, 1.0, 1.0], [0.0, 0.0, 1.0]],
  "user_roles_dict": {
    "id001": ["admin"],
    "id002": ["lv3"],
    "id003": ["lv2"]},
  "action_roles_dict":
  {"money10000": ["admin"],
    "moneyPlus1": ["admin", "lv3"],
    "automaticPlus1": ["admin", "lv3", "lv2"]}}
```

2.1.4 与数据库的交互

与数据的交互，包括从业务数据库中抽取 user，以及 RBAC 本身数据的存储。这一部分有待加入。

2.2 META-RBAC：对 RBAC 进行管理

本小节以谈论课程中提到的的使用 RBAC 管理 RBAC 的讨论。用 RBAC 管理 RBAC 当然是可以的！但问题是，用来进行管理的 RBAC，是不是同样也需要被别的 RBAC 管理？在这个问题上，笔者的观点是，越上层的机制，越简单，越不容易变动，所以，再复杂的系统，两三层也完全足够了。这和自然界食物链的长度一般不超过 5 是相似的。

2.3 为用户提供一定的权限管理自由度——全新的 RBAC 实现

时间原因，这部分没有完成。在最后的大作业里将会出现这一部分。

2.3.1 简述权限管理自由度

2.3.2 实现与 RBAC 库的融合

3 事件管理和监听触发机制的自我实现

3.1 基本原理

监听触发模式和事件管理机制在 JAVA、Javascript 等开发型语言中具有极其广泛的定义和应用，但每种语言的实现存在一些不同。在以对象为核心的语言中，listerner 作为一个对象形式的属性被加入到另一个实例化的对象里，该对象常常具有一个初始化为空的监听器列表，而后，当某一个方法被调用时，该方法会遍历基于监听器基类遍历列表中所有的监听器，以发送事件。而在另外一些面向对象语言里，为了防止在高并发背景下事件监听和发送的一些额外问题，会使用一个特殊的实例化对象——事件管理器去管理这些监听触发的过程。

由于使用管理器可以实现监听者和监听对象在封装上的解耦，同时可以更好地应对海量监听信息的情形，本文基于事件管理器的方法进行使用。python 标准不存在这套机制，故此处自我封装实现之。

3.2 事件管理器的设计

事件管理器是本机制的核心，因而首先介绍。事件管理器的核心是管理事件，因此，该对象一直维护一个事件-监听器的哈希表，该字典以每一个事件类型为其键，以指向该事件需要触发的监听器函数对象的指针的列表为键所对应值，当接受到一个事件时，管理器就会触发该事件对应的所有的监听器。在这个过程中，传入的事件可以非常多，因此，管理器还需要维护一个事件队列，以实现间隔触发。该部分整体的代码实现如下，可以发现，实现的代码并不优雅，尤其是传入监听器信息时：

```
class myEventManager(object):
    def __init__(self):
  self.__is_run=0
  self.__event_queue=Queue()
  self.__thread=Thread(target=self.__run)
  self.__event_listerner_dict={}

    def __run(self):
  self.is_run=1
  while True:
      try:
    print("开始处理新的数据....")
    event=self.__event_queue.get(block=True,timeout=5)
    self.__event_process(event)
      except Empty:
    print("没有数据，等待中......")
    pass

    def start(self):
  self.__is_run=1
  print("事件管理器已启动")
  self.__thread.start()

    def stop(self):
  self.__is_run=0
  self.__thread.join()


    def __event_process(self,event):
  if event.type_ in self.__event_listerner_dict.keys():
      for listerner in self.__event_listerner_dict[event.type_]:
    listerner(event)
  print("事件处理完成")

    def addEventListerner(self,type_,listerner):
  if type_ not in self.__event_listerner_dict.keys():
      self.__event_listerner_dict[type_]=[]
  self.__event_listerner_dict[type_].append(listerner)
```

```
  print("监听器添加成功")

    def removeEventListerner(self,type_,listerner):
  if type_ in self.__event_listerner_dict.keys():
      if len(self.__event_listerner_dict[type_])>1:
    self.__event_listerner_dict[type_].remove(listerner)
      else:
    del self.__event_listerner_dict[type_]


    def sendEvent(self,event):
  print("事件发送成功")
  self.__event_queue.put(event)
```

3.3 监听器和触发事件的设计

监听器和触发事件的设计相比事件管理较为简单。唯一的区别是，在诸如 JAVA 这种强制面向对象语言中，函数需要通过类进行定义。而此处，监听器的设计则单单通过函数定义了。项目中的一个例子为：

```
def ListernerQiangHua1times(event1times):
    action=controller.myRBAC.return_actions(event1times.datadict["data"].getUser(),
        successQiangHua1times)
    action(event1times.datadict["data"])
```

可以发现，这样的一个监听器以定义的数据对象作为输入，使用数据字典中的数据进行处理。由于此处需要将 RBAC 与 listerner 进行结合，所以实践并非直接被执行，而是经过一个查询之后，在进行执行。经由这种方式，当没有权限时，针对该动作的对应处理动作就会被调用。

触发事件的设计稍微复杂一点。一方面，触发事件要通过事件类型同监听器进行绑定，另一方面，触发事件中要包含进行处理的数据。可以将这个特性抽象出来获得一个有关于事件的基类：

```
class Event:
    def __init__(self,type_=None,datadict=None):
  self.type_=type_
  self.datadict=datadict
```

之后，基于这个事件类去实例化对象就好了，下面的例子又定义了新的子类，这是为了将每一个事件同其 user 挂钩，因为有时监听器需要回过头来对对象进行修改，通过这种方式，可以实现回调。

```
class EventQianghua1Times(Event):
    def __init__(self,user):
  super(EventQianghua1Times, self).__init__("强化一次",{"data":user})
```

4 以简陋到极致的一个界面为例，展露游戏中 RBAC 应用的嵌入式视角

前面更多地都是对工具的利用，下面以一个例子的形式将作业进行展现。

4.1 基础的场景介绍

此处设想的是这样一个场景：在某一个游戏下，存在着这样一个装备强化功能。对于装备强化功能，游戏方支持“强化一次”、“强化十次”和“强化一百次”。如果一个玩家等级在 10 级以下，则只能使用强化一次，十级以上将解锁强化十次，而如果这位玩家充了 VIP 会员，则他可以使用强化 100 次的功能。本次作业就是在如此简单寻常的场景下进行的。

1. 动作：分别包括强化一次、强化十次、强化一百次三个函数。当没有权限时，默认不执行并 alert 没有权限的弹窗。
2. 角色：小于十级、大于等于十级、VIP。
3. 用户：此处的用户进行了一定的抽象。正常而言，user 需要是游戏玩家，而一个游戏里面玩家的数量是非常多的，因此，我们基于对每个 player 进行简单的属性提取，以得到 user。在此处，user 是一个二元组拼接的结果，也就是将数据库中的玩家等级（level）和玩家是否是 VIP（is_VIP）两个变量做字符串形式的拼接。假设在该游戏里等级一共有 100 级，VIP 只有一种，这样得到的映射矩阵其元素数为 100*2*3=600 个。如果不进行抽象，但是一个映射矩阵就需要上千万个元素不止了。

4.2 界面设与示例

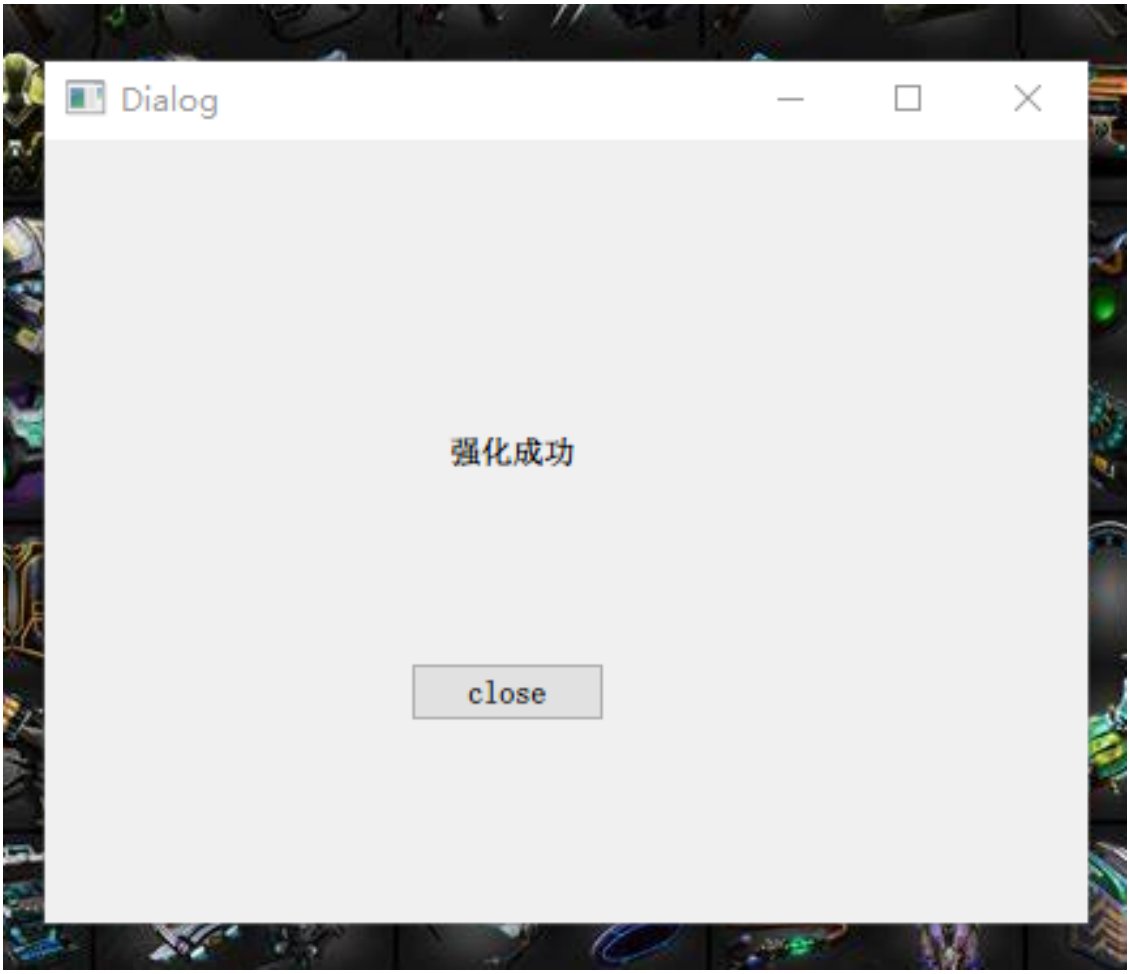
本文基于 pyqt5 自己撰写了一个简陋的界面，也是笔者第一次学习写界面。随后学了前端，这个界面如果使用前端，估计只需要十分钟，大作业将使用 BS 模式实现。

主界面为：

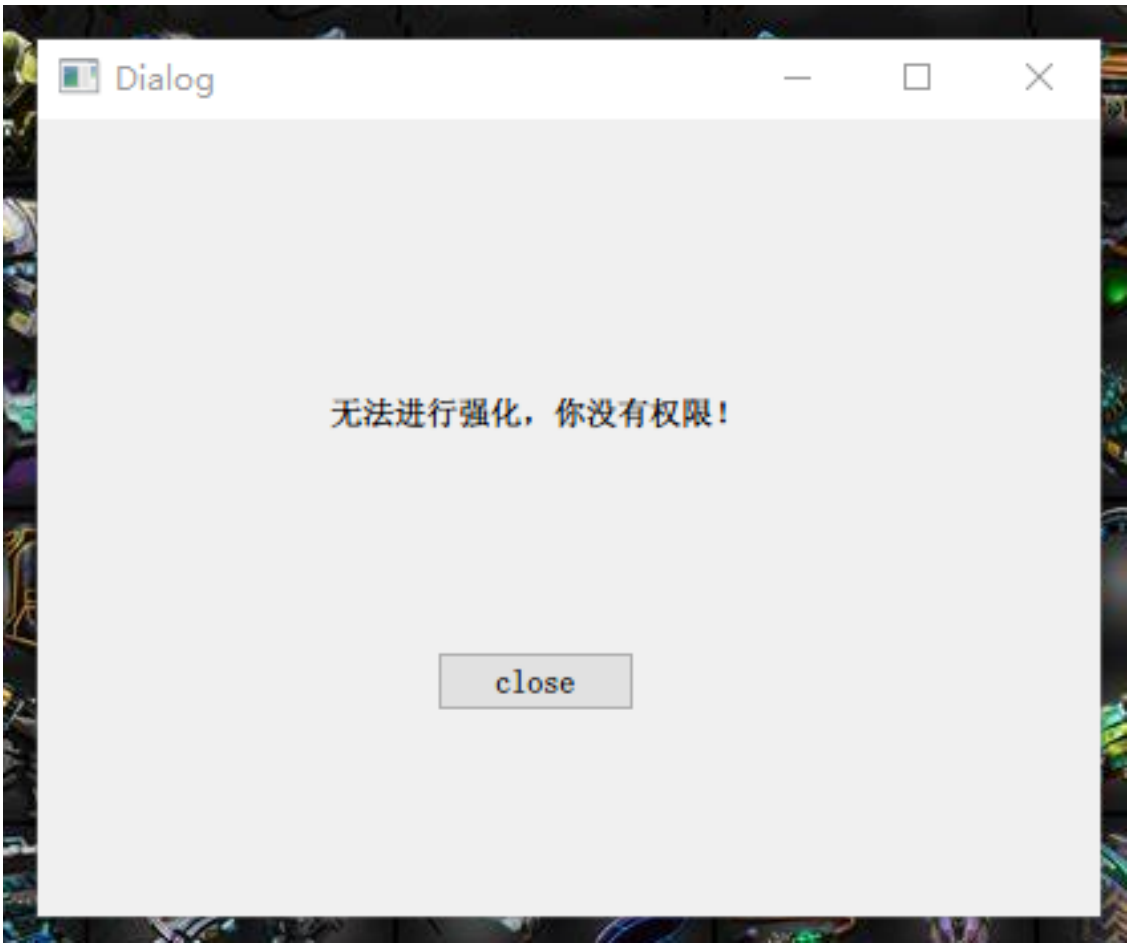
返回

强化一次　强化十次　强化100次

如果有权限，界面为：



如果没有权限，截面为：



举例，如果我们定义一个玩家，等级 3 且没有 VIP，那么他只能够进行强化一次的操作。这时后台会获取以下日志：

```
没有数据，等待中.....
开始处理新的数据....
事件发送成功
{'1_0': ['level_under_10'], '1_1': ['level_under_10', 'vip'], '2_0': ['level_under_10'], '2
    _1': ['level_under_10', 'vip'], '3_0': ['level_under_10'], '3_1': ['level_under_10', '
    vip'], '4_0': ['level_under_10'], '4_1': ['level_under_10', 'vip'], '5_0': ['
    level_under_10'], '5_1': ['level_under_10', 'vip'], '6_0': ['level_under_10'], '6_1': ['
    level_under_10', 'vip'], '7_0': ['level_under_10'], '7_1': ['level_under_10', 'vip'], '8
    _0': ['level_under_10'], '8_1': ['level_under_10', 'vip'], '9_0': ['level_under_10'], '9
    _1': ['level_under_10', 'vip'], '10_0': [], '10_1': ['vip'], '11_0': [], '11_1': ['vip'
    ], '12_0': [], '12_1': ['vip'], '13_0': [], '13_1': ['vip'], '14_0': [], '14_1': ['vip'
    ], '15_0': [], '15_1': ['vip'], '16_0': [], '16_1': ['vip'], '17_0': [], '17_1': ['vip'
    ], '18_0': [], '18_1': ['vip'], '19_0': [], '19_1': ['vip'], '20_0': ['level_up_20'], '
    20_1': ['level_up_20', 'vip'], '21_0': ['level_up_20'], '21_1': ['level_up_20', 'vip'],
    '22_0': ['level_up_20'], '22_1': ['level_up_20', 'vip'], '23_0': ['level_up_20'], '23_1'
    : ['level_up_20', 'vip'], '24_0': ['level_up_20'], '24_1': ['level_up_20', 'vip'], '25_0
    ': ['level_up_20'], '25_1': ['level_up_20', 'vip'], '26_0': ['level_up_20'], '26_1': ['
    level_up_20', 'vip'], '27_0': ['level_up_20'], '27_1': ['level_up_20', 'vip'], '28_0': [
    'level_up_20'], '28_1': ['level_up_20', 'vip'], '29_0': ['level_up_20'], '29_1': ['
```

```
    level_up_20', 'vip'], '30_0': ['level_up_20'], '30_1': ['level_up_20', 'vip'], '31_0': [
    'level_up_20'], '31_1': ['level_up_20', 'vip'], '32_0': ['level_up_20'], '32_1': ['
    level_up_20', 'vip'], '33_0': ['level_up_20'], '33_1': ['level_up_20', 'vip'], '34_0': [
    'level_up_20'], '34_1': ['level_up_20', 'vip'], '35_0': ['level_up_20'], '35_1': ['
    level_up_20', 'vip'], '36_0': ['level_up_20'], '36_1': ['level_up_20', 'vip'], '37_0': [
    'level_up_20'], '37_1': ['level_up_20', 'vip'], '38_0': ['level_up_20'], '38_1': ['
    level_up_20', 'vip'], '39_0': ['level_up_20'], '39_1': ['level_up_20', 'vip'], '40_0': [
    'level_up_20'], '40_1': ['level_up_20', 'vip'], '41_0': ['level_up_20'], '41_1': ['
    level_up_20', 'vip'], '42_0': ['level_up_20'], '42_1': ['level_up_20', 'vip'], '43_0': [
    'level_up_20'], '43_1': ['level_up_20', 'vip'], '44_0': ['level_up_20'], '44_1': ['
    level_up_20', 'vip'], '45_0': ['level_up_20'], '45_1': ['level_up_20', 'vip'], '46_0': [
    'level_up_20'], '46_1': ['level_up_20', 'vip'], '47_0': ['level_up_20'], '47_1': ['
    level_up_20', 'vip'], '48_0': ['level_up_20'], '48_1': ['level_up_20', 'vip'], '49_0': [
    'level_up_20'], '49_1': ['level_up_20', 'vip'], '50_0': ['level_up_20'], '50_1': ['
    level_up_20', 'vip'], '51_0': ['level_up_20'], '51_1': ['level_up_20', 'vip'], '52_0': [
    'level_up_20'], '52_1': ['level_up_20', 'vip'], '53_0': ['level_up_20'], '53_1': ['
    level_up_20', 'vip'], '54_0': ['level_up_20'], '54_1': ['level_up_20', 'vip'], '55_0': [
    'level_up_20'], '55_1': ['level_up_20', 'vip'], '56_0': ['level_up_20'], '56_1': ['
    level_up_20', 'vip'], '57_0': ['level_up_20'], '57_1': ['level_up_20', 'vip'], '58_0': [
    'level_up_20'], '58_1': ['level_up_20', 'vip'], '59_0': ['level_up_20'], '59_1': ['
    level_up_20', 'vip'], '60_0': ['level_up_20'], '60_1': ['level_up_20', 'vip'], '61_0': [
    'level_up_20'], '61_1': ['level_up_20', 'vip'], '62_0': ['level_up_20'], '62_1': ['
    level_up_20', 'vip'], '63_0': ['level_up_20'], '63_1': ['level_up_20', 'vip'], '64_0': [
    'level_up_20'], '64_1': ['level_up_20', 'vip'], '65_0': ['level_up_20'], '65_1': ['
    level_up_20', 'vip'], '66_0': ['level_up_20'], '66_1': ['level_up_20', 'vip'], '67_0': [
    'level_up_20'], '67_1': ['level_up_20', 'vip'], '68_0': ['level_up_20'], '68_1': ['
    level_up_20', 'vip'], '69_0': ['level_up_20'], '69_1': ['level_up_20', 'vip'], '70_0': [
    'level_up_20'], '70_1': ['level_up_20', 'vip'], '71_0': ['level_up_20'], '71_1': ['
    level_up_20', 'vip'], '72_0': ['level_up_20'], '72_1': ['level_up_20', 'vip'], '73_0': [
    'level_up_20'], '73_1': ['level_up_20', 'vip'], '74_0': ['level_up_20'], '74_1': ['
    level_up_20', 'vip'], '75_0': ['level_up_20'], '75_1': ['level_up_20', 'vip'], '76_0': [
    'level_up_20'], '76_1': ['level_up_20', 'vip'], '77_0': ['level_up_20'], '77_1': ['
    level_up_20', 'vip'], '78_0': ['level_up_20'], '78_1': ['level_up_20', 'vip'], '79_0': [
    'level_up_20'], '79_1': ['level_up_20', 'vip'], '80_0': ['level_up_20'], '80_1': ['
    level_up_20', 'vip'], '81_0': ['level_up_20'], '81_1': ['level_up_20', 'vip'], '82_0': [
    'level_up_20'], '82_1': ['level_up_20', 'vip'], '83_0': ['level_up_20'], '83_1': ['
    level_up_20', 'vip'], '84_0': ['level_up_20'], '84_1': ['level_up_20', 'vip'], '85_0': [
    'level_up_20'], '85_1': ['level_up_20', 'vip'], '86_0': ['level_up_20'], '86_1': ['
    level_up_20', 'vip'], '87_0': ['level_up_20'], '87_1': ['level_up_20', 'vip'], '88_0': [
    'level_up_20'], '88_1': ['level_up_20', 'vip'], '89_0': ['level_up_20'], '89_1': ['
    level_up_20', 'vip'], '90_0': ['level_up_20'], '90_1': ['level_up_20', 'vip'], '91_0': [
    'level_up_20'], '91_1': ['level_up_20', 'vip'], '92_0': ['level_up_20'], '92_1': ['
    level_up_20', 'vip'], '93_0': ['level_up_20'], '93_1': ['level_up_20', 'vip'], '94_0': [
    'level_up_20'], '94_1': ['level_up_20', 'vip'], '95_0': ['level_up_20'], '95_1': ['
    level_up_20', 'vip'], '96_0': ['level_up_20'], '96_1': ['level_up_20', 'vip'], '97_0': [
    'level_up_20'], '97_1': ['level_up_20', 'vip'], '98_0': ['level_up_20'], '98_1': ['
    level_up_20', 'vip'], '99_0': ['level_up_20'], '99_1': ['level_up_20', 'vip'], '100_0':
    ['level_up_20'], '100_1': ['level_up_20', 'vip']}
{<function successQiangHua1times at 0x000002129D7BA1F0>: ['level_under_10', 'level_up_20', '
    vip'], <function successQiangHua10times at 0x00000212A4E34D30>: ['level_up_20', 'vip'],
```

```
    <function successQiangHua100times at 0x00000212A4E34DC0>: ['vip']}
================inference================
We allow [3_0] becuase of roles [{'level_under_10'}] for action [<function
    successQiangHua1times at 0x000002129D7BA1F0>]
========END-inference================
update! to: 13
事件处理完成
```

日志中详细记录了是哪一个 user 使用哪一个 role 完成或者失败了哪一个 action。

但是，这里的问题是，此处的记录没有把 player 的粒度刻画出来，因此是不够精确的。一个比较好的方式是将 RBAC 与推荐相关的 log 记录分离，二者均在 listerner 里被记录，因为在监听器中，每一个 player 的信息被默认为是已知的。

5 附录

源代码网址：https://gitee.com/18842378119/course1-zhuang-bei-qiang-hua

环境：

```
# python 3.x
pip install pyqt5 numpy
```