当代数据库数据库管理系统 Homework2:寻宝游戏(二)

10185501409 池欣宁

写在前面

报告可能比较长，可以通过打开目录进行索引

实验环境

1. Postgresql
2. Postgresql可视化： Navicat Premium
3. Flask 框架:SQLAlchemy/pyscog
4. Flask APSschedule
5. Python 3.8
6. venv 虚拟环境与python包版本管理 venv 对应的包各版本如下：

```
(venv)  chixinning@mymac  ~/Desktop/HuntSQL  source venv/bin/activate
(venv)  chixinning@mymac  ~/Desktop/HuntSQL  virtualenv venv
created virtual environment CPython3.8.6.final.0-64 in 650ms
  creator CPython3Posix(dest=/Users/chixinning/Desktop/HuntSQL/venv, clear=False
, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle
, via=copy, app_data_dir=/Users/chixinning/Library/Application Support/virtualen
v)
    added seed packages: APScheduler==3.6.3, Flask==1.1.2, Flask_APScheduler==1.
11.0, Flask_SQLAlchemy==2.4.4, Jinja2==2.11.2, MarkupSafe==1.1.1, SQLAlchemy==1.
3.20, Werkzeug==1.0.1, click==7.1.2, itsdangerous==1.1.0, pip==20.2.3, pip==20.2
.4, psycopg2==2.8.6, python_dateutil==2.8.1, pytz==2020.4, setuptools==50.3.0, s
etuptools==50.3.2, six==1.15.0, tzlocal==2.1, wheel==0.35.1
  activators BashActivator,CShellActivator,FishActivator,PowerShellActivator,Pyt
honActivator,XonshActivator
```

7. pytest 测试与coverage

支持的游戏场景

homework2所描述的场景

1. 买卖宝物
2. 宝物分为两类：一类为**工具，它决定持有玩家的工作能力**；一类为**配饰，它决定持有玩家的运气**。
3. 寻宝与劳动。
4. 每个宝物都有一个自己的名字（尽量不重复）。每位玩家能够**佩戴**的宝物是有限的（比如一个玩家只能佩戴一**个工具和两个配饰**）。多余的宝物被放在存储箱中，不起作用，但可以拿到市场出售。
5. 挂牌限制：在市场上**挂牌**的宝物必须在存储箱中并仍然在存储箱中，直到宝物被卖出。挂牌的宝物可以被**收回**，并以新的价格重新挂牌。当存储箱装不下时，运气或工作能力值最低的宝物将被系统自动回收。

与homework1描述的场景相同不再赘述。

额外实现的更符合游戏完整逻辑的功能

7. 玩家初次进入游戏时的注册操作，同时记录玩家登陆密码
8. 玩家登陆进入游戏
9. 玩家忘记密码时，回看个人密码。
10. 玩家查看个人主页(基本信息)
11. 玩家查看个人存储箱/个人出售列表/个人穿戴列表
12. 玩家售卖物品支持改价，当有treasure被某玩家挂牌出售时，支持玩家直接修改该宝物价格，而不是通过先回收再重新出售

数据库设计

需求分析：了解用户需求，确定软件的基本功能

用户需求

用户：游戏玩家

用户属性：

1. 一定数量的金币
2. 随身携带：一个工具类宝物和两个运气类宝物，随身携带的宝物**决定**着当天玩家在自动寻宝的时间段过程中获得的收益。
3. 存储箱：一定数量的宝物(分位工具类)
4. 宝物：
    1. 类型：宝物功能用途
    2. 等级：宝物等级决定了用户某种事件的结果。

用户操作：

Type1: 系统自动帮助玩家进行的,即以游戏中的"天"为单位。**

1. 寻宝(afschedule定期巡航)： details:寻宝根据玩家佩戴的饰品进行，使用randint在level处保证随机性，对于这一个可多人在线的游戏而言，系统后台的宝物数据库的个数/种类对每个登陆游戏玩家是不透明的，但对每个玩家而言，系统后台的宝物数据库为不可更改的同一宝物数据库。**但是，玩家不可能寻到与已有宝物库里相同的宝物。**【与homework1相同】
2. 打工：(afschedule定期巡航) 根据佩戴的物品的labour属性帮助玩家打工

> 说明：为了使寻宝和打工操作更符合真实的游戏逻辑，默认flask 框架所构建的web应用app.py是全天候运行，即游戏后台服务器一直处于后台状态，所以系统自动帮助玩家进行的活动，默认只要flask run，即遍历players列表的所有玩家，自动帮助玩家进行打工和寻宝操作。**【无论玩家是否真实登陆游戏查看或进行相关操作】**同时，调整Config文件中的巡航执行时间，防止过快既不符合实际应用，也防止可能的数据污染。

Type2: 系统自动帮助玩家进行的,即系统帮助玩家的"自动"丢弃机制。

- 回收 当玩家**进行寻宝**前，判断玩家存储箱的大小，如果存储箱的大小超过maxlen，则自动回收level最低的宝物，这里working宝物和fortune宝物一起被排序。 当玩家想在市场上确定购买宝物时，系统自动帮助玩家进行不必要的宝物的回收操作。 同时，当玩家想将宝物从市场上回收时，也有此操作。 总结：

当玩家进行所有涉及到存储箱数量改变的操作时，均应该使用此操作。

Type3: 玩家自主进行的操作(opearation)

1. 玩家浏览自己的个人基本信息(个人主页)
2. 玩家浏览自己的存储箱/穿戴列表/市场/个人出售列表
3. 玩家在市场挂牌售卖宝物
4. 玩家将宝物从市场上回收
5. 玩家售卖物品支持改价，当有treasure被某玩家挂牌出售时，支持玩家直接修改该宝物价格，而不是通过先回收再重新出售
6. 玩家初次进入游戏时的注册操作，同时记录玩家登陆密码
7. 玩家登陆进入游戏
8. 玩家忘记密码时，回看个人密码。

概念模型设计：确定数据库需要ER设计图

从上述用户需求可得

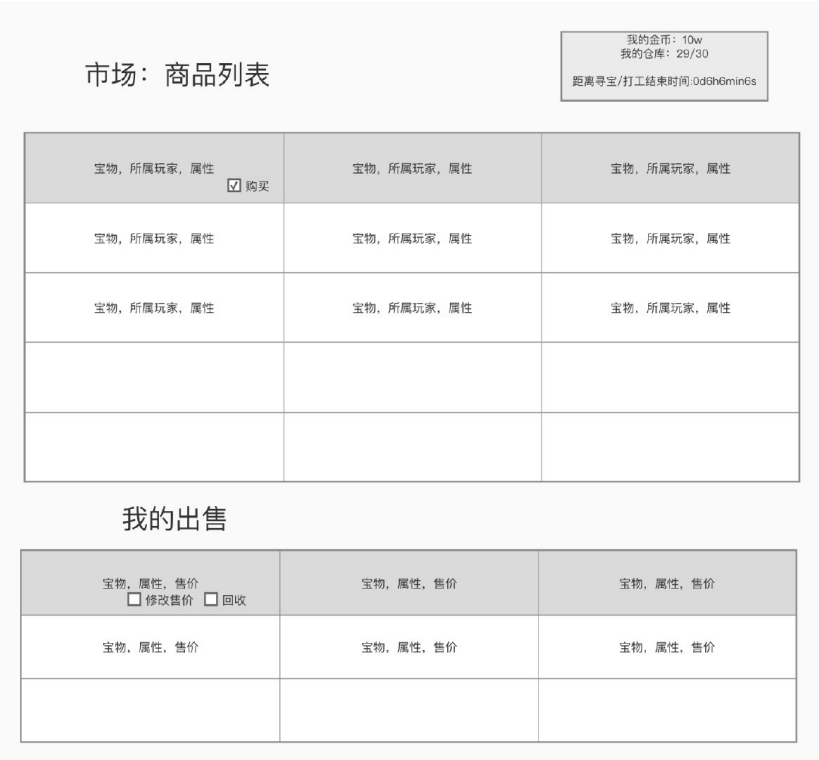
1. 宝物(treasure)
2. 宝物库(Box)
3. 用户(player)
4. 市场(market)
5. 穿戴(Takeaway)#与Nosql设计的显著区别。

模拟UI

模拟用户操作而模拟用户访问频率，得到初始的DB design Schema，通过**模拟sql查询过程**对数据库结构进行设计与调整。

用户界面UI

市场列表UI



模拟UI与考虑游戏版本可扩展性所得设计表的结论,按照访问频率从高到低排列

正常玩家的操作：

1.Most Frequent:

用户登陆，即进入用户主界面，所以Player这个表格是用户访问最多的文档集，由于用户访问的频繁行，所以要尽可能多的涉及用户有关的最常用信息嵌入到player这个的entity中，便于用户需要时夺得及时的访问，以及减少数据库其他文档集的频繁访问，以增强用户体验。

2.查看仓库

用户点击主页面的仓库按钮，即可查看仓库box与更改穿戴takeaway等。

3.与市场有关的交易操作

通常而言，用户一般在浏览完自身信息和存储库信息后，才会去选择去市场进行相关交易，以增强自身能力，防止存储箱溢出等。

【Additional】：游戏版本可扩展性考虑(Nosql和SQL的显著不同点)

在Nosql数据库中，对于玩家box/takewawy等可变长(即很经常的涉及到该类型数据库的增改删)，对于MongoDB提供的嵌套document/array类型数据，都使该游戏有很灵活的扩展性。

对于此种**多值属性**，在SQL中，选择但列表是一个很好的选择；即使在本次游戏设定中玩家takeaway最多有一个working类型的宝物和两个fortune类型的宝物，对于直接嵌入到玩家Table也是一个比较好的选择，**(因为玩家获胜的最终目的是通过改变穿戴获得最好的宝物)**，但是这不能提供很好的扩展性，如当游戏后续版本升级后，玩家takeaway的最大容量上升，所以玩家穿戴类型我在设计时选择了新增单个表。

Raw schema：1st DB design--Table Structure

具体涉及文件在`~/action/init_.py`文件下

Players					
pid	pname	passwd	money	fortune	working

Treasures			
tid	tname	type	level

box		
bid	pid	tid

Takeaway							
sid	pid	twname	tf1name	tf2name	twlevel	tf1level	tf2level

market			
mid	pid	tid	price

Final DDL在实际编写过程中最终的数据库设计展示

这里只做简单真实和简单介绍，逻辑索引选择的合理性见下一部分设计细节。

Players					
pname	pid	passwd	money	fortune	working

Treasures			
tid	tname	type	level

box		
bid	pid	tid

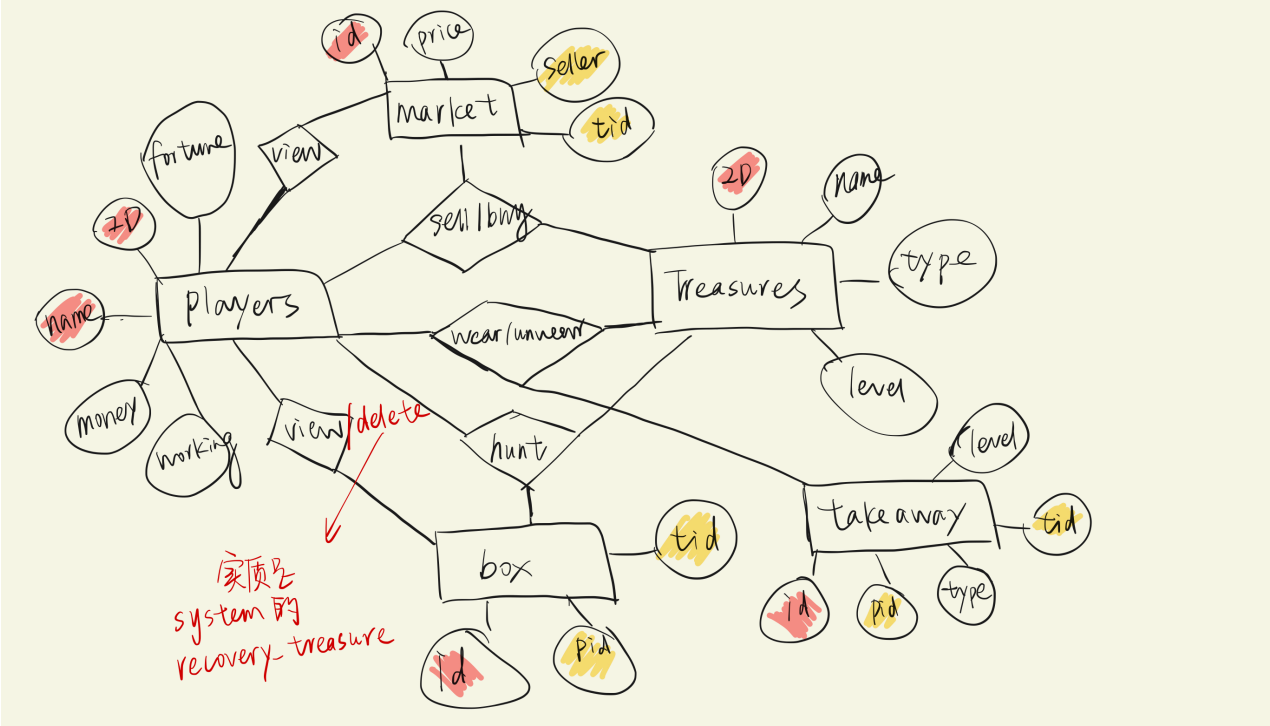
Takeaway					
aid	pid	tname	tid	ttlevel	type

market			
mid	sid(pid)	tid	price

主键
外键
主键+外键

具体建表代码在`~/action/init_db.py`文件下

ER图



players

```
CREATE TABLE "public"."players" (
  "pname" varchar(64) COLLATE "pg_catalog"."default" NOT NULL,
  "pid" int4 NOT NULL DEFAULT nextval('players_pid_seq'::regclass),
  "passwd" varchar(64) COLLATE "pg_catalog"."default" NOT NULL,
  "money" int4,
  "fortune" int4,
  "working" int4,
  CONSTRAINT "players_pkey" PRIMARY KEY ("pid", "pname"),
  CONSTRAINT "players_pname_key" UNIQUE ("pname"),
  CONSTRAINT "players_pid_key" UNIQUE ("pid"),
  CONSTRAINT "players_money_check" CHECK (money > 0)
)
;
```

Player(Entitiy) Table 的属性内容如下：

- pid: string 类型，为玩家id，uniqueKey, primaryKey;
- pname： string 类型，为玩家id，uniqueKey, primaryKey;用户名和密码都不可为空，且用户名是唯一索引；

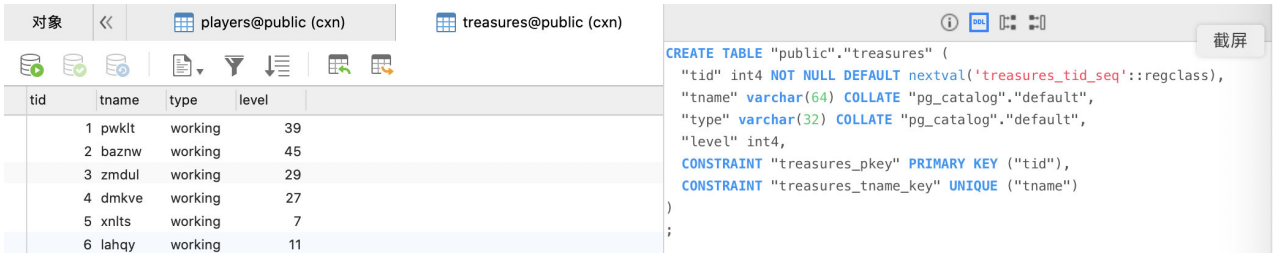
  player表的主键为：(pname,pid); 这两个都是经常出现在选择条件中的属性，耗时太长的查询所涉及的属性，且经常使用`pname`属性作为用户访问数据库的接口，绝大多数函数是通过pname去locate到pid，pid用于该表暴露在外与其它表做join
- money：int 类型，该玩家当前拥有的金币数量
- Passwd：str类型，为玩家的登陆密码

- fortune/working都是值玩家属性值，符合UI设计时**实时**显示在右上角的逻辑。

treasures

treasure文档集，相当于系统默认的宝物库，对于每个初次注册登陆的玩家，为保证游戏起始的公平性，宝物库相同，【可以模拟玩家“出生”的偶然性，模拟不同玩家有不同质量的宝物库~游戏后续升级的🍑之一】。

宝物库，只记录宝物的最基本信息，即宝物名称，注意，这里将宝物名称作为唯一的宝物索引不同的宝物。我自行编写`db_gener.py`脚本使用random函数随机生成的宝物名称和初始化宝物库的插入代码



- tame:str宝物名称
- Level:宝物等级
- Type:working 即为工具类宝物，fortune即为玩具类宝物。

markets

```
CREATE TABLE "public"."markets" (
  "mid" int4,
  "sid" int4 NOT NULL,
  "tid" int4 NOT NULL,
  "price" int4,
  CONSTRAINT "markets_pkey" PRIMARY KEY ("sid", "tid"),
  CONSTRAINT "markets_sid_fkey" FOREIGN KEY ("sid") REFERENCES "public"."players" ("pid") ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT "markets_tid_fkey" FOREIGN KEY ("tid") REFERENCES "public"."treasures" ("tid") ON DELETE NO ACTION ON UPDATE NO ACTION
)
;
```

当玩家准备在市场上售出时，向markets 集合插入一条信息，如上图所示。

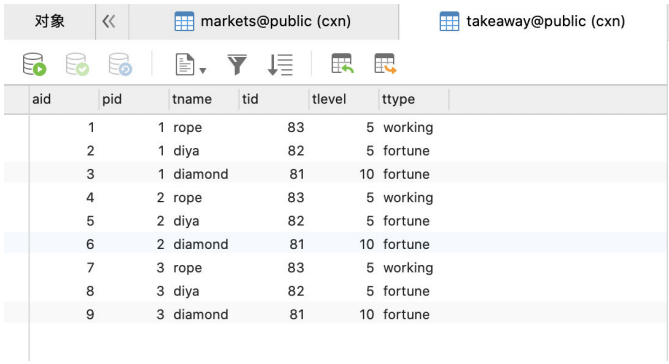
- Tid:int类型 引用treasures的tid,主键
- Price: int类型，记录宝物出售的价格，功能支持宝物售卖者进行改价。
- Sid:int类型，记录出售者的名称。更符合正常游戏逻辑的设计

  因为我在初始化宝物库，即treasure collection的时候，只向其中插入了80个document，且所有玩家在第一次寻宝(hunt)之前，自身的幸运等级和工作能力等级被初始化相同，所以，有很高的概率在游戏的初始阶段，不同玩家寻到的宝物是有大比例重叠的。所以需要在市场信息中记录出售玩家以**区分同名宝物**，玩家在后续的购买功能时，需要以(treasurename,sellername)来唯一指定购买的

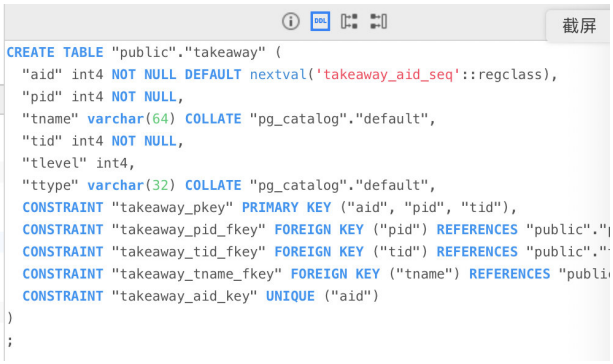
是哪一款宝物。

- level:int类型，记录该宝物的等级。更符合正常游戏逻辑的设计。
- Mid：通过市场mid作为市场销售信息的唯一索引，更利于回归和标示。

takeaway



aid	pid	tname	tid	tlevel	ttype
1	1	rope	83	5	working
2	1	diya	82	5	fortune
3	1	diamond	81	10	fortune
4	2	rope	83	5	working
5	2	diya	82	5	fortune
6	2	diamond	81	10	fortune
7	3	rope	83	5	working
8	3	diya	82	5	fortune
9	3	diamond	81	10	fortune




```
CREATE TABLE "public"."takeaway" (
  "aid" int4 NOT NULL DEFAULT nextval('takeaway_aid_seq'::regclass),
  "pid" int4 NOT NULL,
  "tname" varchar(64) COLLATE "pg_catalog"."default",
  "tid" int4 NOT NULL,
  "tlevel" int4,
  "ttype" varchar(32) COLLATE "pg_catalog"."default",
  CONSTRAINT "takeaway_pkey" PRIMARY KEY ("aid", "pid", "tid"),
  CONSTRAINT "takeaway_pid_fkey" FOREIGN KEY ("pid") REFERENCES "public"."
  CONSTRAINT "takeaway_tid_fkey" FOREIGN KEY ("tid") REFERENCES "public"."
  CONSTRAINT "takeaway_tname_fkey" FOREIGN KEY ("tname") REFERENCES "publi
  CONSTRAINT "takeaway_aid_key" UNIQUE ("aid")
)
;
```

在关系数据库中,takeaway我选择了作为嵌套array直接插入player的document中，而在这次sql设计中，我选择将它单独作为一个table,以支持游戏的高可扩展性。

box

```
CREATE TABLE "public"."box" (
  "bid" int4,
  "pid" int4 NOT NULL,
  "tid" int4 NOT NULL,
  CONSTRAINT "box_pkey" PRIMARY KEY ("pid", "tid"),
  CONSTRAINT "box_pid_fkey" FOREIGN KEY ("pid") REFERENCES "public"."players" ("pid") ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT "box_tid_fkey" FOREIGN KEY ("tid") REFERENCES "public"."treasures" ("tid") ON DELETE NO ACTION ON UPDATE NO ACTION
)
;
```

box跟takeaway本质上有一定的对称性，但在信息存储的冗余程度上有一定的不同。

raw schema与final DDL的对比 && 设计细节

问题1:为什么选择(pname,pid)作为主属性而不是(pid,pname)?

我在一开始设计数据的时候，犯了以为(pid,pname)和(pname,pid)的索引顺序对于数据库的查询没有影响，但这样是错误的，在raw schema设计时，我很自然的把pid作为第一列，但当再次复盘时发现，如果将pid作为顺序较为前的索引对于从用户输入用户名的角度考虑的逻辑是很不合理的。

因为作为检索最常用的`username`属性，用户在进行游戏时，为了用户体验，只需要输入用户名，而`pid`一般是由`player=conn.execute(select([players]).where(players.c.pname == username)).fetchone()`查询获得，而在我所编写的函数中，无一例外，这条语句都成为执行的第一步，所以说是频率最为频繁的。虽然是以(pname,pid)都作为聚簇索引，但是元组的顺序很大的影响到了查询次序。

而treasures中只需要以`tid`作为主属性，因为在绝大多数情况下，是以tid作为后台表与表之间的连接属性的。

问题2:为什么没有将玩家的穿戴(takeaway)进行嵌套进player Table?

我在一开始设计的时候，对于玩家的穿戴放在哪里纠结了很久。在raw schema的最初设计时，准备直接将它嵌套进去作为player的属性。可以嵌套的理由是“对于玩家穿戴的操作，才涉及到玩家"takeaway"这个key所对应的value的更新，但由于takeaway的上限容量更小，所以并不是过多冗余。”

但是，经过后面游戏具体设计实现发现，如果我直接嵌套，对于游戏版本迭代的升级效果和可扩展性有了非常大的限制，如果游戏在后续版本更迭时满足working类型的宝物也可以带两个时，整个表的结构都需要很大的更改。所以我选择新建一个takeaway的表格。

问题3:为什么box中没有将(bid,pid,tid)作为primaryKey而是选择了(bid, pid)作为primaryKey?

自动寻宝功能//穿戴，脱下宝物功能//买卖功能等，即所有游戏的核心功能，都围绕着玩家box的更改进行。所以在box库中的检索速度是很重要的，`bid`属性作为box实体的标示,在很多玩家各新增宝物时，通过bid可以很快的insert，也很符合数据库操作的逻辑。

在真实的应用场景中，玩家自主查看自己的宝物库是很频繁的，但对于在tid属性上也建立索引则没有那么必要，因为对于宝物库来说，最常做的操作是以pid作为groupby的属性，进一步讨论，box表只有3个column,即使当后续游戏扩展后有很多新的column(假设)，以tid作为primarykey也是不合适的，不应该在较少column的情况下建立过多的索引。

问题4:为什么与box在某种意义上对称的takeawayTable要存储冗余(重复)信息?

我在rawschema时，打算一个玩家作为takeaway的一行数据，后来在实际操作中，发现我这样不仅编写代码麻烦，也违背了我把takeaway单独作为一个table的初衷。所以，在FinalDDL中，我更改了takeaway的设计方式。

虽然实质上，box和takeaway的表格所存储的信息概念是相同的，本质上都是玩家宝物不同意义上的集合，但我在这里对于这两个表格存储的冗余信息的选择不同。

对于box我只选择了存储(pid,tid)作为Columns,而takeaway我则是选择了逆范式化，存储了部分冗余信息。

游戏玩家的根本目的在于使自己的working ability和fortune ability尽可能的高，只有在这两个属性提高的情况下，才有可能寻到更好的宝物获得更多的钱，获取游戏的胜利。所以，模拟玩家的角度，他访问自身working ability和fortune ability的频率是非常高的，即可能和UI一样就在游戏主页上实时显示。玩家穿戴和他宝物库里的宝物数量大小相差是非常远的，所以对于box不适合的冗余，在takeaway这个相对很小的数据量的冗余是很有必要的，尽可能加快速度，增强用户体验。

问题5:为什么不单独区分working treasure 和fortune treasure?

为了系统丢弃时考虑。

系统自动帮助玩家回收宝物，需要将working treasure和fortune treasure作为同等权的宝物丢弃，所以在这里，我没有将working treasure 和fortune treasure分开，考虑到玩家可佩戴的不同种类的宝物的最大值和系统可寻宝宝物库的个数，可以赋不同权，在这里等权的情况下，单独区分working treasure和fortune treasure没有太大的意义。

对比NoSQL 和SQL DB 设计的不同

在MongoDB实现中，由于嵌套数组的高扩展型，对于schema的结构调整比较累活，适合于具有不确定需求的数据。

在Postgresql实现中，建表的过程本身就有很多要求，对于什么属性设置何种限制就有要求。SQL是精确的。它最适合于具有精确标准的定义明确的项目。

最终实现玩家操作时的SQL查询语句

单表查询

1. 获取id:`select pid from players where pname='testname'`
2. 获取属性:`select working from players where pname='username'`
3. 获取玩家所有宝物：`select * from box group by pid`
4. 获取玩家所有穿戴`select * from takeaway group by pid`

多表查询

(我在实际应用中是分步，所以很多多表查询被拆成了单表查询)

1. 获取玩家拥有的该宝物属性`select level from players,treasures,box where players.pid=box.pid and box.tid=treasures.tid`
2. 获取玩家在销售列表:`select markets.tid from markets,players where markets.sid=players.pid`
3. 获取玩家工具类穿戴`select takeaway.tid,takeaway.level from takeaway,players where pname=username and takeaway.type=working`

程序设计与游戏最终支持的功能展示

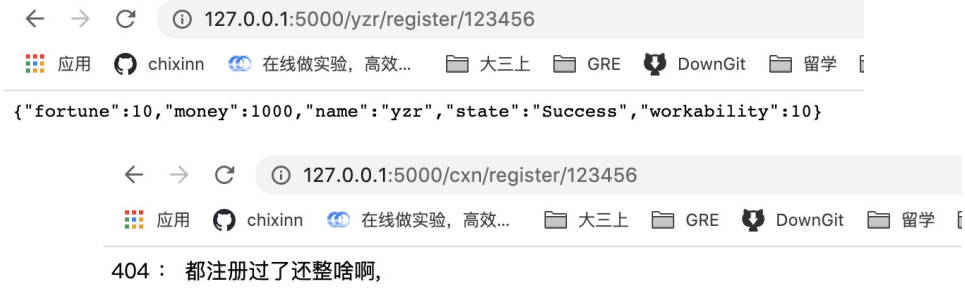
为了保证说明文档的简介性，只保留实现思路，不保留代码。

后端

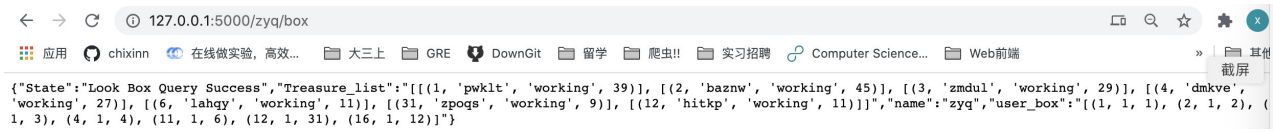
register

用户注册，不同玩家不能拥有相同的用户名，以"name"作为unique索引，避免重复。

注册初始化时，默认玩家working_level=fortube_level=10,也是什么都不佩戴时的level。

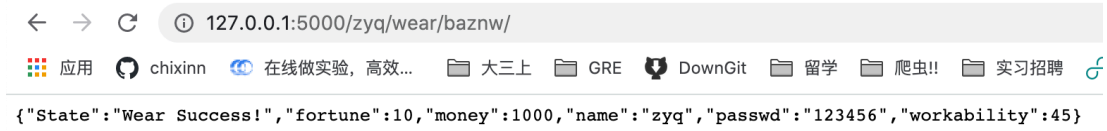


Look_box



Look_takeaway

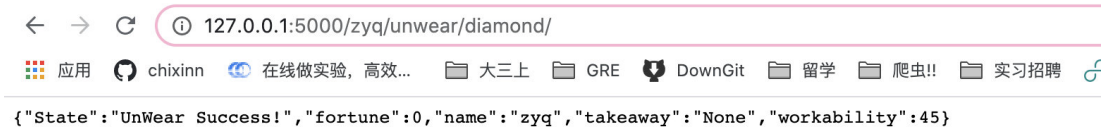
wear



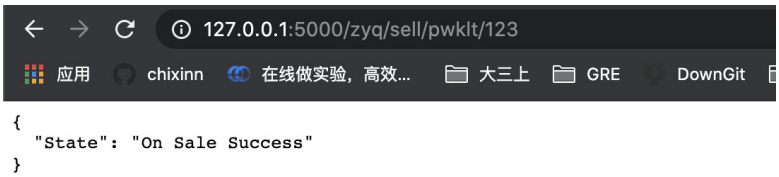
可以看到本来人物working_ability为10，穿working类成功后显示workability为45,fortune还是10

unwear

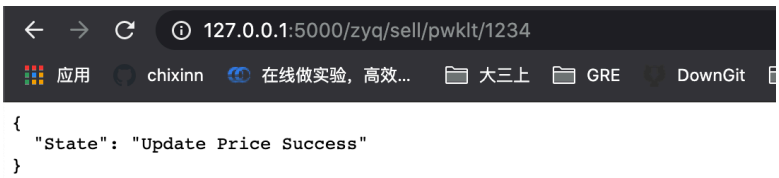
可以看到把唯一的fortune属性脱掉以后，fortune变成了0



sell：首次出售



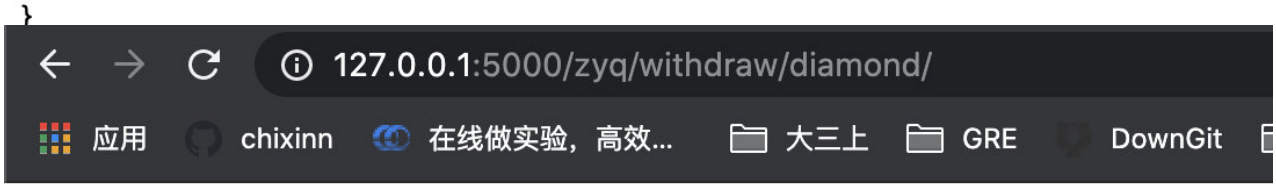
sell:改价



withdraw

用户将某宝物挂牌回收。


```
{
  "State": "Success!",
  "sell_list": "[['item_id: 1', 'seller_id: 3', 'price:81']]"
}
```


```
{
  "State": "WithDraw Item Success"
}
```



```
{
  "State": "Success!",
  "sell_list": "[]"
}
```


buy


```
{
  "State": "Transaction Success!"
}
```


再次查看市场时，该宝物已从市场上删除。


```
{
  "State": "Success!",
  "sell_list": "[]"
}
```

逻辑程序编写核心点(与后台数据库如何设计无关):

着重以unwear函数和buy函数为例:

unwear

```
#脱下
def unwear(username,treasure):
    # 边界检查1/2:确定玩家宝物名称输入正确
    player=conn.execute(select([players]).where(players.c.pname ==
username)).fetchone()
    if player==None:
        return '404: No Such User Error'

treasure_tounwear=conn.execute(select([treasures]).where(treasures.c
.tname == treasure)).fetchone()
    if treasure_tounwear == None:
        return make_response(jsonify({"State":"Treasure found in
treasures Error"}))
    # return str(treasure_tounwear)
    #获取要佩戴宝物的玩家
    pid=player["pid"]
    #获取该玩家的随身携带

current_takeaway=conn.execute(select([takeaway]).where(takeaway.c.pi
d == pid)).fetchall()
    #再获取要脱下的宝物类型,id和类型
    treasure_tounwear_type=treasure_tounwear["type"]
    treasure_tounwear_id=treasure_tounwear["tid"]
    treasure_tounwear_level=treasure_tounwear["level"]
    #遍历current_takeaway 判断该宝物是不是真的在戴:
    #通过SQL语句也可以获得该逻辑
    current_list=[]
    for i in current_takeaway:
        current_list.append(i[2])
    if treasure not in current_list:
         return make_response(jsonify({"State":"Treasure found in
takeawway Error"}))
    # 然后就可以脱了，被脱下的宝物要从takeawayTable中删除，然后要更新回box,
    # 相应的人物属性也要更新:
    if treasure_tounwear_type=='working':
        # 为了防止有重复的 还是要设置删除功能 这里要继续debug
```

```
        # conn.execute(takeaway.delete().where(takeaway.c.pid==pid
and takeaway.c.tname==treasure))
        #注意这个bug！！！
        drop_aid=session.query(takeaway).filter(and_(takeaway.c.tid
== treasure_tounwear_id,takeaway.c.pid == pid)).one()[0]
        # drop_aid =
conn.execute(select([takeaway]).where(takeaway.c.tid ==
treasure_tounwear_id and takeaway.c.pid == pid)).fetchone()[0]
        # return
str(drop_aid)+'************'+str(pid)+'*********'+str(treasure_tounw
ear_id )
        conn.execute(takeaway.delete().where(takeaway.c.aid ==
drop_aid))#终于多重条件删掉了5555
        #box UPdate前要检查size
        size=session.query(box).filter(and_(box.c.pid ==
pid)).count()
        if size > MAXSTROAGE:
            recovery_treasure(pid)

conn.execute(box.insert().values(pid=pid,tid=treasure_tounwear_id))#
与bid无瓜

conn.execute(players.update().where(players.c.pid==pid).values(worki
ng=10))#不能为0
        player=conn.execute(select([players]).where(players.c.pname
== username)).fetchone()

current_takeaway=conn.execute(select([takeaway]).where(takeaway.c.pi
d == pid)).fetchone()
        # return '4-4'
        return make_response(jsonify({"State":"UnWear
Success!","name": username,"fortune": player['fortune'],
"workability": player['working'],"takeaway":str(current_takeaway)}))

    if (treasure_tounwear_type=='fortune'):
        # conn.execute(takeaway.delete().where(takeaway.c.pid==pid
and takeaway.c.tname==treasure))
        drop_aid=session.query(takeaway).filter(and_(takeaway.c.tid
== treasure_tounwear_id,takeaway.c.pid == pid)).one()[0]
```

```
        # drop_aid =
conn.execute(select([takeaway]).where(takeaway.c.tid ==
treasure_tounwear_id and takeaway.c.pid == pid)).fetchone()[0]
        conn.execute(takeaway.delete().where(takeaway.c.aid ==
drop_aid))#终于多重条件删掉了5555
        size=session.query(box).filter(and_(box.c.pid ==
pid)).count()
        if size > MAXSTROAGE:
            recovery_treasure(pid)

conn.execute(box.insert().values(pid=pid,tid=treasure_tounwear_id))#
与bid无瓜
        current_fortune_level=player['fortune']

conn.execute(players.update().where(players.c.pid==pid).values(fortu
ne=current_fortune_level-treasure_tounwear_level))
        player=conn.execute(select([players]).where(players.c.pname
== username)).fetchone()

current_takeaway=conn.execute(select([takeaway]).where(takeaway.c.pi
d == pid)).fetchone()
        # return '4-4'
        return make_response(jsonify({"State":"UnWear
Success!","name": username,"fortune": player['fortune'],
"workability": player['working'],"takeaway":str(current_takeaway)}))
```

buy

```
def buy(username, treasure,seller):
    #边界检查1:确定买家和卖家都在
    buyer=conn.execute(select([players]).where(players.c.pname ==
username)).fetchone()
    seller=conn.execute(select([players]).where(players.c.pname ==
seller)).fetchone()
    if buyer==None:
        return jsonify({"State":"Buyer Not found ERR"})
    seller_id=seller['pid']
    if seller==None:
```

```
        return jsonify({"State":"Seller Not found ERR"})
    buyer_id=buyer['pid']
     #边界检查2:确定宝物名称输入正确

treasure_to_buy=conn.execute(select([treasures]).where(treasures.c.t
name == treasure)).fetchone()
    if treasure_to_buy==None:
        return jsonify({"State":"Trea Not found ERR"})
    treasure_to_buy_tid=treasure_to_buy['tid']
    #边界检查3:市场检查
    treasure_to_buy=session.query(markets).filter(and_(markets.c.tid
== treasure_to_buy_tid ,markets.c.sid==seller_id)).one()
    drop_mid=treasure_to_buy[0]
    if treasure_to_buy ==None:
        return jsonify({"State":"Trea Not in market found ERR"})
    seller_box_id=conn.execute(select([box]).where(box.c.pid ==
seller_id and box.c.tid==treasure_to_buy_tid)).fetchone()[0]
    treasure_to_buy_price=treasure_to_buy[3]
    seller_money=seller['money']+treasure_to_buy_price
    buyer_money=buyer['money']-treasure_to_buy_price
    #边界检查4:判断买家是否可以买的起
    if buyer_money<0:
        return '<h1>你买不起</h1>爬'
       #边界检查5:buyer箱子/Takeaway检查，一个玩家对于一种宝物来说持有的数量
最多是1
    try:

session.query(box).filter(and_(box.c.pid==buyer_id,box.c.tid==treasu
re_to_buy_tid)).one()

session.query(takeaway).filter(and_(takeaway.c.pid==buyer_id,takeawa
y.c.tid==treasure_to_buy_tid))
        # return str(res)
        return jsonify({"State":"Transaction Failure! due to Already
Have One"})
    except:
    #删seller箱子
        conn.execute(box.delete().where(box.c.bid == seller_box_id))
  #边界检查6:增buyer箱子(recovery_treasure)，增之前要看有没有满
        size=session.query(box).filter(and_(box.c.pid ==
buyer_id)).count()
        if size > MAXSTROAGE:
```

```
            recovery_treasure(buyer_id)
        try:
  #
conn.execute(box.insert().values(pid=pid,tid=treasure_tounwear_id))#
与bid无瓜
    #更新双方的账户
        #东西放入买家账户
        #钱走出买家
        #钱走入买家
        # 该市场记录还需要删除

conn.execute(box.insert().values(pid=buyer_id,tid=treasure_to_buy[2]
))

conn.execute(players.update().where(players.c.pid==buyer_id).values(
money=buyer_money))

conn.execute(players.update().where(players.c.pid==seller_id).values
(money=seller_money))
        #该市场记录还需要删除
            conn.execute(markets.delete().where(markets.c.mid==
drop_mid))

            return jsonify({"State":"Transaction Success!"})
        except:
           #边界检查7:Code Error Cases;
            return jsonify({"State":"Transaction Failure!"})
```

Other details:

- hunt(寻宝)函数要保证玩家不能寻到所持有过的一样的宝物。
- 设计到玩家box更改操作，只要是使box_size增加的函数，都要检查box容量上限，即使调用自动自动丢弃函数。所以，有可能前后两次update的操作。
- 挂牌出售时，支持玩家改价的功能很重要，否则对于market来说会显示两条玩家信息
- 每次访问数据库进行更改的时候，都要做边界检查返回值，不同的查询语句conn.execute()和session.query()的细节不一样，要注意，以防止出错。

- **自动寻宝/打工的整体逻辑**借鉴了这个代码：APS Scheduler自动巡航
- 注意两个条件查询的代码细节BUG!!!
- MySQL的auto_increment属性只能用在主键上，所以bid/mid/aid/pid/tid都必须是主键。
- Pytest有严格的Json借口，我在编写的时候没有特别遵守，可能导致出错。

总结

这次的数据库作业实践，让我第一次完整的体会到了web应用的构建流程，并以此应用为驱动，获得了很多web应用开发的知识。使刚刚结束的计算机网络课程中的http协议，前端、后端、数据设计、测试等，有了脱离于纸的理解与认识，同时对python语言本身也得到了熟悉，也逐渐学会去看一些报错信息，何用log打印的形式进行Debug,如最近一次报错信息：`NoneType' object is not subscriptable`我之前一直以为是我选择的数据结构不能索引的问题，仔细排查以后，才发现是索引的空对象。有很多DEBUG的心得已经写在了代码里，在这里不再赘述。

在上一次实验中，由于NoSQL数据库的可扩展和灵活性，我有很多的时间全部花在了前端的了解和学习设计，重点放在实现了后端到前端的传值连接上。

但是，在这次试验中，我更多的将重点放在**SQL表的结构的设计上，并着重细细体会了表的结构的影响，以及SQL数据库设计理念中的DB和Code Developer尽可能低的耦合度上。**

通过两次作业，初次体会到了SQL型数据库和NoSQL型数据库从访问的角度对于数据库设计要求的不同不过，纸上得来终觉浅，绝知此事要躬行，需要更多的实验才可以有更好的体会，这次实验使我收获颇丰，获益匪浅。