华东师范大学数据科学与工程学院实验报告

课程名称：当代数据管理系统	**年级**：2019 级	**上机实践成绩**：
指导教师：周烜	**姓名**：龚敬洋	
上机实践名称：寻宝游戏（MongoDB）	**学号**：10195501436	**上机时间日期**：2021/10/8
上机实践编号：	**组号**：	**上机实践时间**：10:00

Contents

1 实验要求

考虑以下游戏场景：

每个游戏玩家都有一定数量的金币、宝物。有一个市场供玩家们买卖宝物。玩家可以将宝物放到市场上挂牌，自己确定价格。其他玩家支付足够的金币，可购买宝物。

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。

每位玩家每天可以通过寻宝获得一件宝物，宝物的价值由玩家的运气决定。每位玩家每天可以通过劳动赚取金币，赚得多少由玩家的工作能力决定。（游戏中的一天可以是现实中的 1 分钟、5 分钟、10 分钟，自主设定。）

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。

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。

假设游戏永不停止而玩家的最终目的是获得最好的宝物。

请根据以上场景构建一个假想的 Web 游戏，可供多人在线上玩耍。后台的数据库使用 MongoDB。对游戏玩家提供以下几种操作：**寻宝（可以自动每天一次）**、**赚钱（可以自动每天一次）**、**佩戴宝物**、**浏览市场**、**买宝物**、**挂牌宝物**、**收回宝物**。

2 实验过程

2.1 网站构架

本次寻宝游戏网站的整体构架如下图所示 (Figure1)。由于游戏平台通常会由多个不同的模块构成，且需要不断的迭代和集成，因此我们采用了**前后端分离**的架构来设计网站。整个网站分为四个部分，分别为**前端维生服务器**、**前端页面**、**后端请求服务器**及**服务器数据库**。

对于前端维生服务器，我们使用了基于 NodeJS 的经典 Web 服务器维持框架 Express。这一框架提供了一个快速的 Web 应用搭建流程，我们只需要直接将前端页面框架生成的相关资源统一放在相应的位置，Express 就会帮我们自动托管之后的服务器维持事务。

对于前端页面，我们使用了经典的 Vue 3 框架来进行搭建。这一框架提供了一整套完善的 UI 及前后端交互流程，可以十分清晰的梳理出各个模块之间的继承及通信关系，方便后续维护。对于 UI 样式，我们使用了基于 Vue 3 的 Quasar Framework 2 框架，这一框架支持流式数据加载及响应式的交互访问，可以吸引用户访问并使用该平台。

由于需要应对各种不同的情况，后端服务器由多个模块联合构成。首先，我们使用了 Flask 作为容器实现框架，并通过蓝图（Blueprint）功能将接口分摊至五个子接口集合上以实现业务隔离。随后，由于我们使用了前后端分离架构，自然会涉及到跨域问题。因此我们在请求处理接口上加上了一层 CORS 包装器。最后，为了应对高并发请求，我们使用了 Gunicorn 网关容器对全应用进行了封装，并对 Flask 开启了多线程支持。这样在面对高并发请求时系统可以较为均衡的分摊整个负载。

本次实验使用了 MongoDB 作为服务器存储数据库。作为一个文档型数据库，其类 Json 格式的数据管理模型更贴近 Web 交互时的数据格式，这使得我们在后续在设计接口数据协议时更为方便。

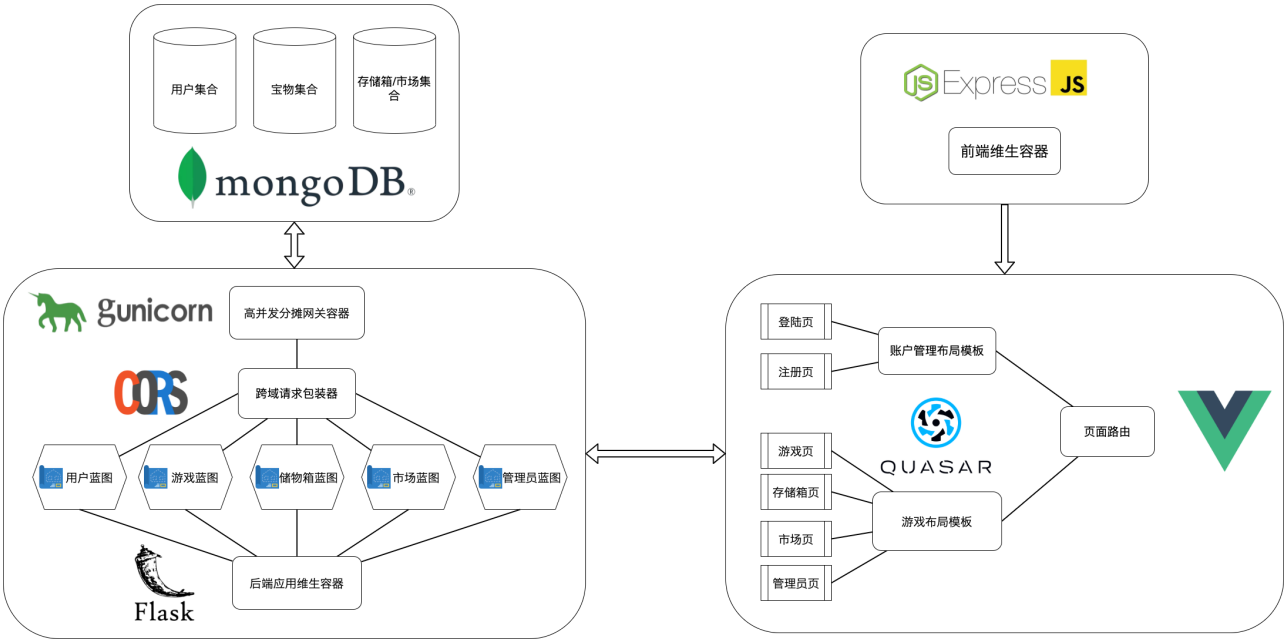

Figure1: 寻宝游戏网站构架

2.2 数据库设计

2.2.1 内嵌存储 v.s 归一化存储

对于当前应用，我们主要需要存储以下几种数据：**系统中可用的宝物及其价值、用户的各种基本信息、用户正在佩戴的物品、用户当前拥有的物品、市场上正在出售的宝物**。

对于文档型数据库，一个最直接的想法便是将所有的用户数据全部存储在同一个集合中。因此，对于当前应用，一个可能的数据库设计如下图所示 (Figure2)。其中，用户已佩戴的物品及储物箱中的物品被以数组的形式嵌入用户集合中。这样做的好处是在每次用户访问其拥有物品时，我们始终能够以 $\mathcal{O}(1)$ 的代价（获取到用户文档之后）完成对用户物品查询。然而，当用户想要变更物品的状态时，系统就需要遍历整个数组以找到对应的物品，此时其时间开销便会变得极为糟糕，在最坏情况下甚至能达到 $\mathcal{O}(n)$（获取到用户文档之后），且索引功能很难帮助优化这一查询代价。此外，这一存储方式会带来大量的数据冗余，且由于宝物的全部信息均被存储在了用户表中，当系统要对宝物信息进行更新时，就需要对所有集合中的所有玩家的每一条宝物数据进行遍历，当用户规模较大时，这一代价将变得十分巨大，且此时可能出现数据不一致的问题，破坏了数据库的 ACID 原则。

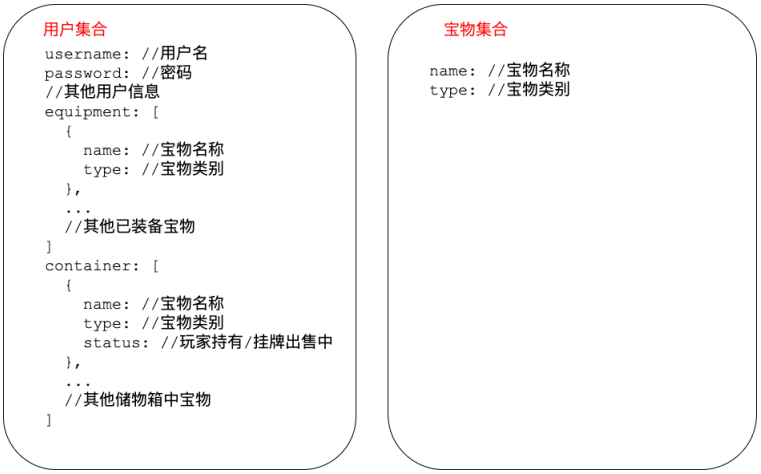

Figure2: 内嵌式数据库设计

在数据库设计中，我们通常还会尽可能的让同一数据在所有集合中尽可能只保留一份以减少数据冗余，即所谓的**存储归一化**（**Storage Normalization**）。在这种模式下，数据库可以以下图所示的方式设计 (Figure3)。这样做的好处是极大的减少了数据冗余，且从灾备的角度来看，即使发生了数据丢失，由于宝物和用户的信息是单独存储的，其数据损失的概率也相对较小。此外，当宝物状态需要发生改变时，我们只需要移动其唯一标识即可进行更改，数据移动的开销较小。然而，由于宝物与用户的信息发生了分离，对于用户的每一件物品，我们都需要至少访问两个集合才能获取到所有需要的数据，而访问不同的集合对于 MongoDB 而言开销是巨大的，因此对于本应用来说仍然不是一个合适的选择。

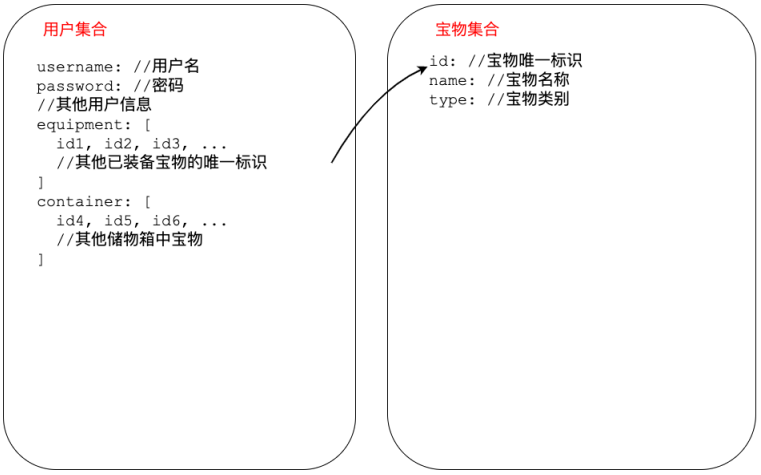

Figure3: 归一化数据库设计

2.2.2 缓存型存储

为了解决上面两种方式的问题，对于本应用，我们采用了一种称为**缓存型存储**（**或存储反归一化，Storage Denormalization**）的思想来设计本应用的数据库结构。与直接存储不同的是，缓存型存储是先将数据进行归一化，随后再将数据以最适合访问的方式进行冗余缓存，这样既保证了

数据修改时的数据一致性，又使得数据能以较高的效率被访问。

本应用的数据库集合设计如下图所示 (Figure4)。其中，用户集合包含了用户的全部基本信息，而宝物集合则维护了当前系统中能够被提供的全部宝物信息。持有物品集合中存储了所有玩家所拥有的物品及其相关状态，其在本应用中既用作玩家存储箱信息的维护，也用作市场上物品信息的维护。此外，根据实际的查询需求，我们对宝物集合的 **gain** 键建立了索引，对持有物品集合的 **owner** 和 **status** 两个键分别建立了索引。

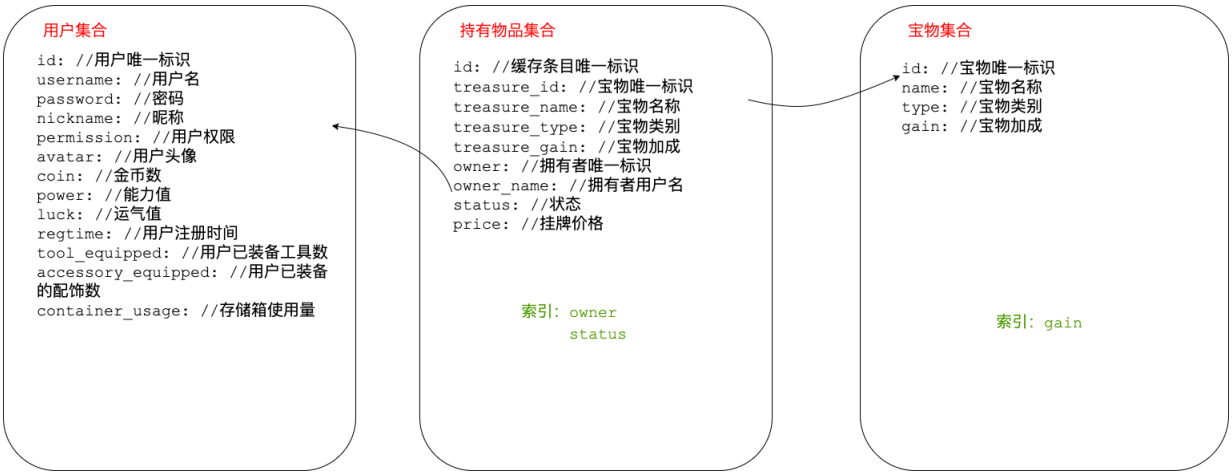

Figure4: 本应用数据库设计

更具体的，每个集合中各个键的数据类型定义如下表所示 (Table1.1 to Table1.3)。这里需要注意的是，对于用户的头像，我们使用了一个对象标识符类型将其指向一个外部的位置。由于 MongoDB 采用了 BSON 作为其文档存储实现，其最大单个文档的大小限制为 4MB。而若要将用户头像这样的二进制数据直接存储在单个文档中，则很容易超过这一限制导致无法存储。为了解决这一问题，MongoDB 提供了一个名为 **GridFS** 的存储方案。通过将二进制文件拆分为多个**小块（Chunk，通常为 256KB/个）**，我们便可以将图片等媒体数据存储在数据库中。因此事实上该应用一共有 5 个集合，额外的两个集合（分别名为 **fs.chunks** 和 **fs.files**）分别用于存储大文件的二进制数据及其块索引。

键名	数据类型	含义
username	字符串（String）	玩家用户名
password	字符串（String）	玩家密码
nickname	字符串（String）	玩家昵称
permission	整数（Integer）	权限/用户组
avatar	对象标识符（ObjectID）	玩家头像
coin	整数（Integer）	持有金币数
power	整数（Integer）	玩家当前能力值
luck	整数（Integer）	玩家当前运气值
regtime	时间（DateTime）	玩家注册时间
tool_equipped	整数（Integer）	玩家已装备工具数
accessory_equipped	整数（Integer）	玩家已装备配饰数
container_usage	整数（Integer）	玩家存储箱已用空间

Table1.1: 用户集合键数据类型定义

键名	数据类型	含义
treasure_id	对象标识符（ObjectID）	宝物唯一标识符
treasure_name	字符串（String）	宝物名称
treasure_type	整数（Integer）	宝物类型
treasure_gain	整数（Integer）	宝物加成
owner	对象标识符（ObjectID）	宝物拥有者
owner_name	字符串（String）	宝物拥有者用户名
status	整数（Integer）	宝物状态
price	整数（Integer）	宝物价格

Table1.2: 持有物品集合键数据类型定义

键名	数据类型	含义
name	字符串（String）	宝物名称
type	整数（Integer）	宝物类型
gain	整数（Integer）	宝物加成

Table1.3: 宝物集合键数据类型定义

作为归一化集合，当数据进行更新时，用户集合和宝物集合拥有最高的更新优先级。持有物品集合作为本应用的缓存集合，包含了玩家可能频繁访问的全部物品数据。因此无论玩家是在访问自己的装备物品、存储箱还是市场上正在出售的物品时，我们都只需要访问持有物品这一个集合，极大地提高了数据查询的效率。

此外，为了保证数据一致性，在每一次应用启动时，系统都会对缓存集合中的数据进行检查。由于缓存集合中记录了其他集合相关条目的唯一标识符，因此这一同步是可行的。当管理员需要进行某些更新（例如对某件宝物的属性进行调整时），系统也会先更新宝物和用户集合，再将数据同步至持有物品集合中。

可以注意到，这里我们没有单独为市场设计一个缓存集合，这是由于用户查询市场上的物品就等价于查询所有玩家中 status=3 的物品，而我们已经对持有物品集合中的 status 键建立了索引，这一查询的效率是极高的，因此无需再另设一个集合专门对市场上的物品进行缓存。此外，若再设一个缓存集合，则需要花费更高的代价去解决数据不一致的问题，得不偿失，可见这样做是并不合适的。

2.2.3 ODM 与数据库交互实现

MongoDB 提供了对 Python 的原生访问接口模块 **pymongo**。通过这一模块，我们可以使用和命令行中类似的类 JSON 方式对数据库中的对象进行 CRUD（Create-Retrieve-Update-Delete）操作。然而，由于其没有与 Python 中对象的直接映射关系，我们无法快速获知当前操作对数据库及 Python 对象进行了怎样的数据变化，这样很容易使得操作逻辑变得不可控（尤其是考虑到文档型数据库中可以动态加入键值的特性）。

受到 MySQL 等关系型数据库中 ORM 概念的启发，对于对象型编程语言与数据库间的交互，一个极好的办法便是实用所谓的**对象-文档映射（ODM，Object Document Mapping）**模型。通过将数据库中的对象映射为 Python 中的类，我们便可以直接使用 Python 中的类操作方法对数据库中的对象进行操作。由于每一个数据库文档中的键都与对象中的一个变量有一一对应关系，我们便可以随时掌握当前操作的内容及其可能的行为，这样就使得整个系统更易于维护。

幸运的是，对于 MongoDB，Python 中已经存在了一款十分完善的 ODM 实现模块 **MongoEngine**。对于本应用，我们直接使用专门针对 Flask 框架封装的 Flask-MongoEngine 来实现对应的 ODM 模型。对于上述提出的集合结构，其 ODM 模型申明代码如下：

```
class User(db.Document):
    username = db.StringField(required=True)
    password = db.StringField(required=True)
    nickname = db.StringField()
    permission = db.IntField(default=1)  # 1 - Player; 2 - Admin
    avatar = db.FileField()
    coin = db.IntField(default=10)
    power = db.IntField(default=1)
    luck = db.IntField(default=1)
    regtime = db.DateTimeField(default=datetime.now())
    tool_equipped = db.IntField(default=0)
    accessory_equipped = db.IntField(default=0)
    container_usage = db.IntField(default=0)
class Treasure(db.Document):
    name = db.StringField(required=True)
    type = db.IntField(required=True)  # 1 - Tools; 2 - Accessories
    gain = db.IntField()
    meta = {
        'indexes': [
            'gain'
        ]
    }
class Container(db.Document):
    treasure_id = db.ObjectIdField(required=True)
    treasure_name = db.StringField()
    treasure_type = db.IntField()
    treasure_gain = db.IntField()
    owner = db.ObjectIdField(required=True)
    owner_name = db.StringField()
    status = db.IntField()  # 1 - Equipped; 2 - In inventory; 3 - On sale
    price = db.IntField()  # Only exists when status = 3
    meta = {
        'indexes': [
            'owner',
            'status'
        ]
    }
```

在该种映射模型下，数据库的 CRUD 操作变得异常简单。下面以用户表的一次 CRUD 操作代码为例：

```
user_create = User(username=username, password=password)  # Create
user_item = User.objects(username=username).first()  # Retrieve
user_item.nickname = abc  # Update
user_item.delete()  # Delete
user_create.save()  # Save
```

2.3 业务实现

2.3.1 账户管理

对于玩家来说，一款游戏的账户管理系统主要涉及用户注册及用户登录两个功能，因此我们分别设计两个接口**/register** 及**/login** 来完成这一交互逻辑。

首先我们来设计注册接口。当用户发起注册请求时，我们首先从请求中拿到用户所要注册的用户名和密码，随后使用 MongoEngine 提供的查询语句查询数据库中用户名是否已经存在。若用户名不存在，则向数据库的 User 文档集合中插入一条新的用户文档，否则则报错。

为了方便后续维护，我们约定/register 接口的返回状态码如下：

状态码	定义
0	业务未完成
1	注册成功
2	用户名已存在
3	用户名或密码为空
999	请求体格式错误

对于登录接口，我们首先从 request 请求体中获得用户名及密码，随后使用 MongoEngine 查询数据库中用户名符合的第一条记录（由于在注册时对数据库中是否存在重名用户进行了检查，因此这里可以保证获得的第一条记录是整个数据库中唯一符合条件的记录）。得到记录后，我们只需要对其密码进行比对，并将判断结果返回给用户（前端）即可。

同样的，这里我们约定/login 接口的返回状态码如下：

状态码	定义
0	业务未完成
1	登录成功
2	密码错误
3	用户名不存在
999	请求体格式错误

对于账户管理页面的前端设计，我们使用了 Layout+Page 的模式对其进行了样式统一，并使界面尽可能的保持简洁。其效果如下图所示 (Figure5 to Figure6)。

Figure5: 登陆界面

Figure6: 注册界面

2.3.2 Session 与用户组

由于后续的请求中大量涉及到用户验证，若每次都需要在请求体中加入用户名和密码，则后续操作将会变得十分复杂，且由于用户信息始终在端与端之间传输，会造成极大的安全隐患。因此这里我们使用**会话（Session）**技术来保持用户的登录状态。同时，为了保证请求安全性，我们还需要对 Session 进行加密。

Flask 原生提供了对 Session 的支持，我们可以直接使用键值对的方式对一个应用中的 Session 进行操作。对于本应用，我们对除登录注册外的所有接口都设置了 **Session 验证**，当检测到用户发来的请求头中没有 Session 信息的话，则会直接返回 100 状态码告诉用户无权访问。

在实际场景下，有时我们需要对游戏中的内容进行更新操作（如添加新的可用宝物），为了方便这一操作，我们将其引入前端的交互界面中。然而一旦将修改全局数据库的操作暴露在公开接口中，我们就需要开始考虑操作的权限验证问题，否则就有可能产生安全问题。得益于 Session 用户验证机制，我们可以通过设置用户组来对用户的访问权限进行限制。

对于当前应用，我们进行如下的权限组约定：

权限码	权限组
0	未注册用户
1	普通玩家
2	管理员用户

可以看到，在前面的数据库结构设计中，我们在用户集合中设置了一个 permission 键用于标识用户所在的组。当用户注册完成后，该键默认被设置为 1，且不可通过接口请求的方式进行更改。为了方便统一管理，我们将所有的管理员操作专门放到一个**/admin** 蓝图中。该蓝图中的所有接口均进行了用户组认证。当用户发起请求时，系统会首先检查请求头 Session 中包含的用户信息，若用户的权限组高于 2，则继续处理用户所请求的操作，否则直接返回用户码 2 告诉用户无权修改服务器。

对于不同的用户组，前端界面的呈现也进行了一定的区分 (Figure7 to Figure8)。其中当用户为管理员时，菜单中会多出一栏管理员界面可供用户进行操作，这与常规游戏中的设计也基本吻合。

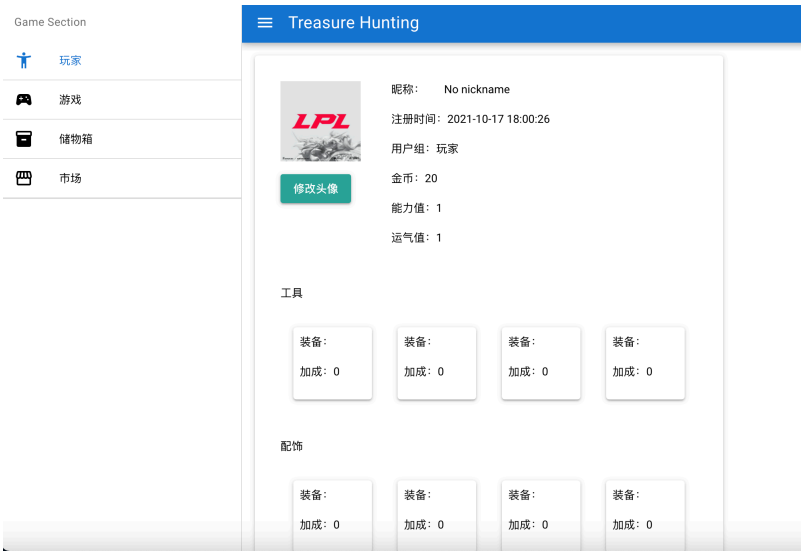

Figure7: 普通用户的游戏菜单

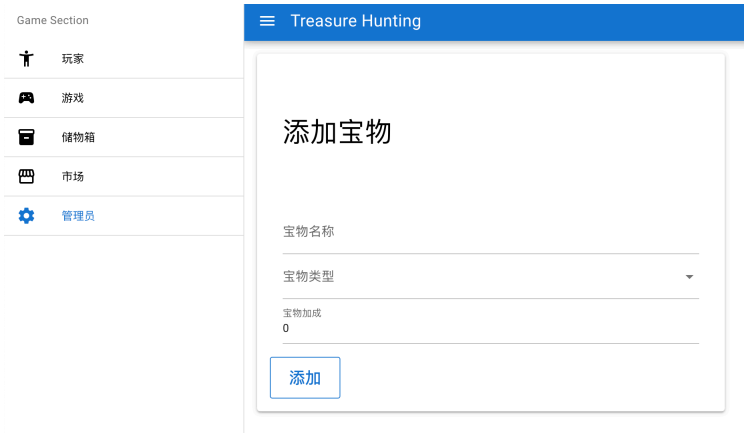

Figure8: 管理员用户的游戏菜单和管理界面

2.3.3 用户信息管理

对于当前应用，用户信息管理主要包括**用户信息查询、修改头像、修改昵称**及**用户已佩戴装备查询**四个操作，我们将其封装在**/user** 蓝图中，接口分别为**/info**、**/set_avatar**、**/get_avatar**、**/set_nickname** 及**/equipment**。

用户信息查询及昵称修改的操作较为简单。我们只需要直接使用 CRUD 操作对数据库进行查询和更新即可，实现代码如下：

```
@user_bp.route("/info", methods=['GET'])
def get_info():
    status = 0
    info_dict = {}
    if session.get('user') is not None:
        user_id_str = session.get('user')
        user_id = ObjectId(user_id_str)
        user_obj = User.objects(id=user_id).first()
        if user_obj.nickname is not None:
            info_dict['nickname'] = user_obj.nickname
        else:
            info_dict['nickname'] = ""
        if user_obj.regtime is not None:
            info_dict['regtime'] = user_obj.regtime.timestamp()
        if user_obj.permission is not None:
            info_dict['permission'] = user_obj.permission
        else:
            info_dict['permission'] = 1
        if user_obj.coin is not None:
            info_dict['coin'] = user_obj.coin
        else:
            info_dict['coin'] = 0
        if user_obj.power is not None:
            info_dict['power'] = user_obj.power
        else:
            info_dict['power'] = 0
        if user_obj.luck is not None:
            info_dict['luck'] = user_obj.luck
```

```
        status = 1
    else:
        status = 100
    return json.dumps({'status': status, 'info': info_dict})
@user_bp.route("/set_nickname", methods=['POST'])
def set_nickname():
    req_str = request.get_data(as_text=True)
    try:
        status = 0
        if session.get('user') is not None:
            req_dict = json.loads(req_str)
            nickname = req_dict['nickname']
            user_id_str = session.get('user')
            user_id = ObjectId(user_id_str)
            user_obj = User.objects(id=user_id).first()
            user_obj.nickname = nickname
            user_obj.save()
            status = 1
        else:
            status = 100
        return json.dumps({'status': status})
    except Exception as e:
        print(e)
        print("Error when phasing request json string!")
        return json.dumps({'status': 999})
```

对于头像的修改，我们需要使用前文中提到的 GridFS 模型对数据库进行操作。幸运的是，MongoEngine 已经为我们封装好了统一的访问接口，因此我们同样可以使用常规的数据库操作将图像的二进制数据存储在数据库中。这里需要注意的是，由于 MongoEngine 提供的文件存储接口传入的是文件标识符，因此我们需要首先将图像存入文件系统中，再将它们以文件的方式读取到 MongoDB 的集合中。

```
@user_bp.route("/set_avatar", methods=['POST'])
def set_avatar():
    status = 0
    if session.get('user') is not None:
        user_id_str = session.get('user')
        avatar_img = request.files.get("avatar")
        save_dir = 'data/' + session.get('user') + '/'
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)
        save_path = os.path.join(save_dir, avatar_img.filename)
        avatar_img.save(save_path)
        avatar_f = open(save_path, 'rb')
        user_id = ObjectId(user_id_str)
        user_obj = User.objects(id=user_id).first()
        if user_obj.avatar.read() is not None:
            user_obj.avatar.replace(avatar_f)
        else:
            user_obj.avatar.put(avatar_f)
        user_obj.save()
        status = 1
    else:
```

```
        status = 100
    return json.dumps({'status': status})
@user_bp.route("/get_avatar", methods=['GET'])
def get_avatar():
    default_avatar_path = 'data/avatar_default.png'
    if session.get('user') is not None:
        user_id_str = session.get('user')
        user_id = ObjectId(user_id_str)
        user_obj = User.objects(id=user_id).first()
        buffer_tmp = user_obj.avatar.read()
        if buffer_tmp is not None:
            avatar_buffer = buffer_tmp
        else:
            avatar_f = open(default_avatar_path, 'rb')
            avatar_buffer = avatar_f.read()
    else:
        avatar_f = open(default_avatar_path, 'rb')
        avatar_buffer = avatar_f.read()
    return send_file(io.BytesIO(avatar_buffer), attachment_filename='avatar.
        jpg')
```

由于图像数据通常较大，因此我们在请求用户信息时使用了多步请求的方式。/info 接口返回用户的文本信息，而/get_avatar 接口专门传输用户的头像。对于未设置头像的用户，我们只需返回一张系统默认的头像即可 (Figure9)。

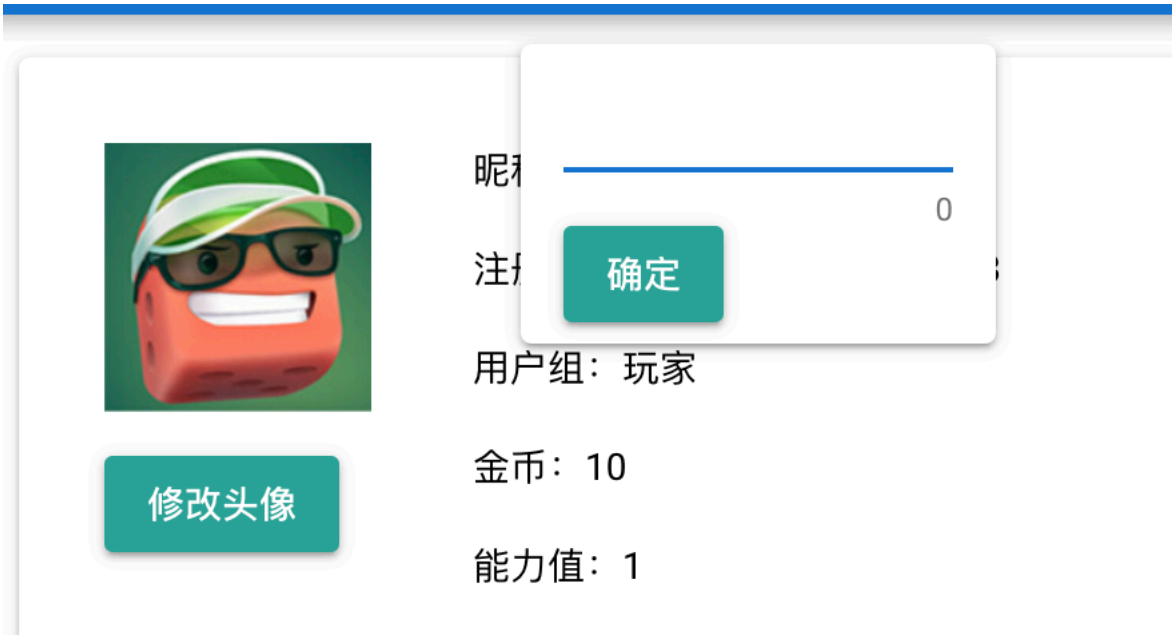

Figure9: 用户头像及昵称修改界面

2.3.4 宝物管理

由于我们使用了统一的集合来对玩家拥有的宝物进行记录，因此宝物管理的相关操作十分简单。这里我们分别设计了**/list**、**/wear**、**/takeoff** 三个接口分别用于**列出玩家拥有的宝物**、**佩戴宝物**、**脱下宝物**三种操作，并将其统一封装在 **container** 蓝图中。

从数据库操作视角来看，列出玩家拥有的宝物等价于在数据库持有物品集合中查询 **owner=用户 ID** 的所有条目，由于前面我们已经对该集合的 owner 键建立了索引，因此这一查询操作是十分高效的（$\mathcal{O}(\lg n)$ 时间开销）。

对于佩戴宝物操作，我们首先从 session 中取得用户 ID，从请求体中取得用户所要佩戴宝物的记录条目 ID，随后分别对玩家的相关键及条目的状态进行修改即可。这里我们限制一个玩家最多只能同时装备 4 件工具和 4 件配饰。因此当检测到玩家已经装备了上限数量的宝物时，我们就取消装备操作并告知用户已不能再装备更多的宝物。

脱下宝物的操作与佩戴宝物正好相反，我们首先判断玩家是否已经佩戴该物品，随后同样分别对玩家的相关键及条目的状态进行修改即可。

/wear 接口的返回状态码定义如下：

状态码	定义
0	业务未完成
1	装备佩戴成功
2	对应装备数量已达到上限
100	用户未登录
999	请求体格式错误

/takeoff 接口的返回状态码定义如下：

状态码	定义
0	业务未完成
1	装备脱下成功
2	当前未佩戴该物品
100	用户未登录
999	请求体格式错误

对于前端设计，我们采用了卡片式的方式呈现用户所持有的物品。此外，根据物品当前的状态，我们分别显示对应的可选操作按钮，使得整个交互逻辑较为清晰。呈现效果如下图所示(Figure10)。

Figure10: 玩家存储箱界面

2.3.5 游戏行为

在该游戏中，两个主要的游戏行为分别为寻宝和工作，我们分别设计了**/seek** 和**/work** 接口用于这两种操作，并将其统一封装在 **game** 蓝图中。

根据游戏规则可知，玩家每天可以分别进行一次寻宝和工作操作。要实现这一点，我们便需要维护一个世界时间的全局变量。幸运的是，Flask 中提供了一个 **current_app** 变量可供我们直接使用，该变量会在当前激活的程序实例存在时始终保持在内存中，因此我们可以通过其维护当前的世界时间。在这里我们定义游戏内的一天对应真实世界中的 5 分钟。为了通知玩家最新的游戏时刻，我们使用了 **WebSocket** 技术将客户端与服务端建立了实时通讯，并每隔 10 秒向所有当前连接的客户端广播一次最新的世界时间。

玩家每次寻宝所能获得的宝物与玩家当前的运气值相关，为此我们需要设计一套合理的宝物随机选取算法。本项目中使用的宝物选取算法如下图所示 (Figure11)。我们首先以玩家当前的运气值为均值，方差为 3 的尺度得到一个正态分布，并从这个正态分布中生成一个随机数。随后，我们查询系统中所拥有的所有宝物价值，并选择出价值最接近的宝物集合。最后我们再使用均匀分布从这个集合中随机选取一个宝物作为本次寻宝所获得的最终宝物。由于我们对宝物的 **gain** 键进行了索引，因此这一查询的效率是极高的。

得到选择的宝物后，我们更新玩家集合及持有物品集合，将新的宝物放到玩家的储物箱中。同时，若玩家已存放的宝物超过了储物箱上限，我们就需要从中选取价值最低的宝物进行淘汰。在这里我们规定存储箱的存储上限为 20，因此当玩家集合的 **containe_usage** 键计数大于 20 时，我们便从玩家的所有存储箱物品中选择一个价值最低的进行淘汰。

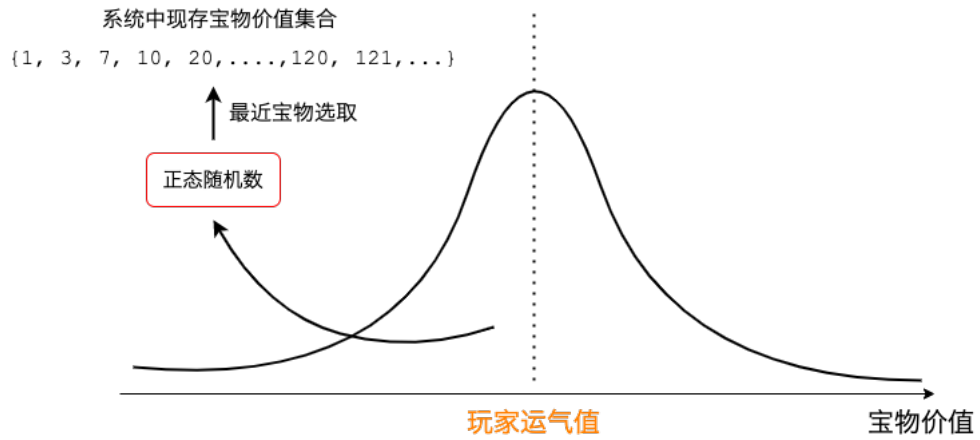

Figure11: 宝物选取算法

寻宝操作的相关实现代码如下：

```
@game_bp.route("/seek", methods=['GET'])
def seek_treasure():
    status = 0
    treasure = {}
    if session.get('user') is not None:
        user_id_str = session.get('user')
        user_id = ObjectId(user_id_str)
        user_obj = User.objects(id=user_id).first()
        user_luck = user_obj.luck
        value_decide = int(np.random.normal(user_luck, 3))
        if value_decide < 1:
            value_decide = 1
        closest_value_obj = Treasure.objects(gain__lte=value_decide).order_by
            ('-gain').first()
        available_objs = Treasure.objects(gain=closest_value_obj.gain).all()
        chosen_index = np.random.randint(0, len(available_objs))
        chosen_obj = available_objs[chosen_index]
        container_item = Container(treasure_id=chosen_obj.id, treasure_name=
            chosen_obj.name,
                                   treasure_type=chosen_obj.type,
                                       treasure_gain=chosen_obj.gain, owner=
                                       user_id,
                                   owner_name=user_obj.username, status=2)
        user_obj.container_usage += 1
        container_item.save()
        user_obj.save()
        if user_obj.container_usage > 20:
            recycle_obj = Container.objects(owner=user_id).order_by('gain').
                first()
            recycle_obj.delete()
            user_obj.container_usage -= 1
            user_obj.save()
        treasure['name'] = chosen_obj.name
        treasure['type'] = chosen_obj.type
```

```
        treasure['gain'] = chosen_obj.gain
        status = 1
    else:
        status = 100
    return json.dumps({'status': status, 'treasure': treasure})
```

工作操作的实现相对较为简单。根据游戏规则，玩家工作所获得的金币数跟玩家的能力值相关。因此我们以玩家能力值作为中心，5 为方差的尺度作出一个正态分布，并从中随机决定一个玩家本次工作获得的金币数。得到决定的金币数后，我们将其加至用户集合的 **coin** 键上，并将结果返回给用户。实现代码如下：

```
@game_bp.route('/work', methods=['GET'])
def gain_money():
    status = 0
    coin = 0
    if session.get('user') is not None:
        user_id_str = session.get('user')
        user_id = ObjectId(user_id_str)
        user_obj = User.objects(id=user_id).first()
        user_power = user_obj.power
        gain_decide = int(np.random.normal(user_power, 5))
        if gain_decide < 1:
            gain_decide = 1
        coin = gain_decide
        user_obj.coin += coin
        user_obj.save()
        status = 1
    else:
        status = 100
    return json.dumps({'status': status, 'coin': coin})
```

2.3.6 市场交易

最后我们来完成市场交易的业务实现。在 **container** 蓝图中，我们提供了**/sell** 和**/abortsell** 接口用于宝物的挂牌及取消挂牌操作；在 **market** 蓝图中，我们提供了**/list** 和**/purchase** 两个接口分别用于浏览市场上正在出售的宝物及购买选择的宝物。

由于我们直接使用了持有物品集合中的 **status** 状态用于指示每个物品的状态，因此挂牌出售的实现逻辑是十分简单的。具体的，我们首先判断当前物品的交易合法性，随后只需要将 status 键值改为 3 并将价格写入 price 键即可。对于查询操作，我们只需要选取持有物品集合中所有 **status=3** 的物品返回给用户即可。由于我们对 **status** 键建立了索引，且一个物品总共只有 3 种状态，因此这一查询操作是十分高效的。

对于购买操作，我们首先检查买家是否能够负担所要购买的物品（用户文档中 **coin** 键值是否大于卖家所提出的价格），若能够负担，则直接将对应的持有物品文档 **owner** 键值由卖家改为买家并分别更新买家和卖家的金币数即可。同样的，在购买行为完成后，我们同样需要判断买家的存储箱中已有的物品是否大于存储上限，并需要进行相应的自动回收操作。

/purchase 接口的返回状态码定义如下：

状态码	定义
0	业务未完成
1	装备佩戴成功
2	当前用户无法负担所要购买物品的价格
3	请求物品不在出售状态
100	用户未登录
999	请求体格式错误

在市场交易的前端实现中，我们对玩家的定价进行了数据验证，只有当价格为大于 0 的整数时才会发送请求至后端，保证了边界情况不会发生。此外，在浏览市场时，对于自己出售的物品，系统将自动隐去购买按钮，使得整个界面的交互逻辑更为清晰 (Figure12 to Figure13)

Figure12: 出售物品界面

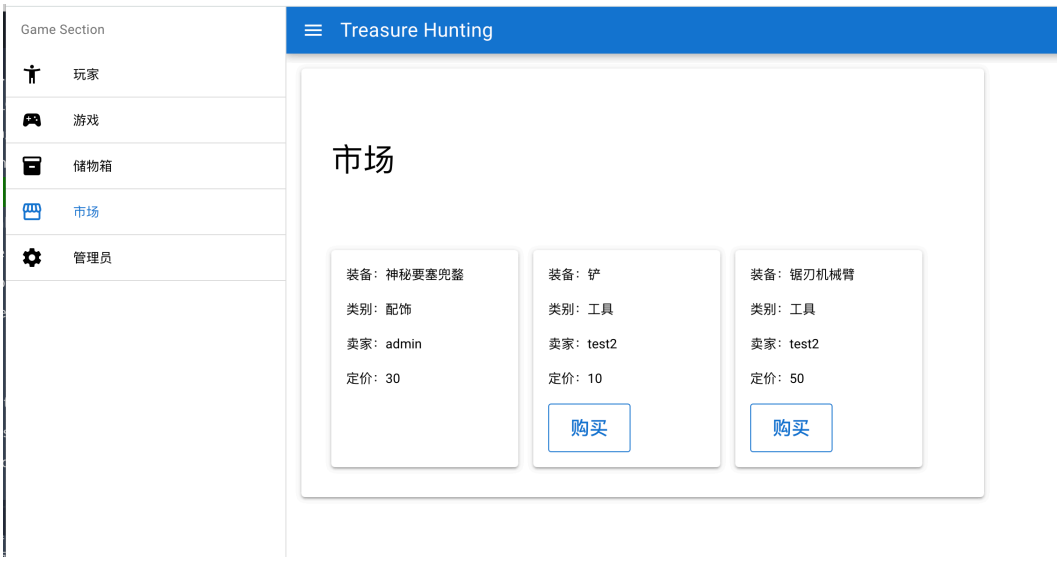

Figure13: 市场界面

2.4 高并发处理

对于一个游戏应用，我们可能会同时收到非常多来自不同玩家的请求。若使用 Flask 的原生 WGSI 服务，则很快便会达到性能瓶颈并导致大量的拒绝请求。为解决这一问题，我们引入了 **Gunicorn** 网关调度器对当前的 Flask 应用进行封装，并使用多线程策略来提高应用所能承担的并发数。(Figure14)

```
gongjingyang@GONGGONGJOHNs-Macbook backend % gunicorn -c gun.py run:app
[2021-10-18 12:05:55 +0800] [2738] [INFO] Starting gunicorn 20.1.0
[2021-10-18 12:05:55 +0800] [2738] [INFO] Listening at: http://0.0.0.0:5000 (273
8)
[2021-10-18 12:05:55 +0800] [2738] [INFO] Using worker: gevent
[2021-10-18 12:05:55 +0800] [2740] [INFO] Booting worker with pid: 2740
[2021-10-18 12:05:55 +0800] [2741] [INFO] Booting worker with pid: 2741
[2021-10-18 12:05:56 +0800] [2742] [INFO] Booting worker with pid: 2742
[2021-10-18 12:05:56 +0800] [2743] [INFO] Booting worker with pid: 2743
[2021-10-18 12:05:56 +0800] [2744] [INFO] Booting worker with pid: 2744
[2021-10-18 12:05:56 +0800] [2745] [INFO] Booting worker with pid: 2745
[2021-10-18 12:05:56 +0800] [2746] [INFO] Booting worker with pid: 2746
[2021-10-18 12:05:56 +0800] [2747] [INFO] Booting worker with pid: 2747
[2021-10-18 12:05:56 +0800] [2748] [INFO] Booting worker with pid: 2748
[2021-10-18 12:05:56 +0800] [2749] [INFO] Booting worker with pid: 2749
[2021-10-18 12:05:56 +0800] [2750] [INFO] Booting worker with pid: 2750
[2021-10-18 12:05:56 +0800] [2751] [INFO] Booting worker with pid: 2751
[2021-10-18 12:05:56 +0800] [2752] [INFO] Booting worker with pid: 2752
[2021-10-18 12:05:56 +0800] [2753] [INFO] Booting worker with pid: 2753
[2021-10-18 12:05:56 +0800] [2754] [INFO] Booting worker with pid: 2754
[2021-10-18 12:05:56 +0800] [2755] [INFO] Booting worker with pid: 2755
[2021-10-18 12:05:56 +0800] [2756] [INFO] Booting worker with pid: 2756
```

Figure14: Gunicorn 启动日志

2.5 接口测试

为了保证应用接口的可用性，下面我们来对后端接口进行相应的测试。由于我们采用了前后端分离的构架设计系统，且应用中的绝大多数请求均涉及到了 Cookies 存储，为了更全面的反映应用在真实数据交互场景下的表现，这里我们首先使用 Postman 进行接口测试。

以'/user/login' 接口为例，测试请求体及测试响应结果如下图所示 (Figure15.1 to Figure15.3)

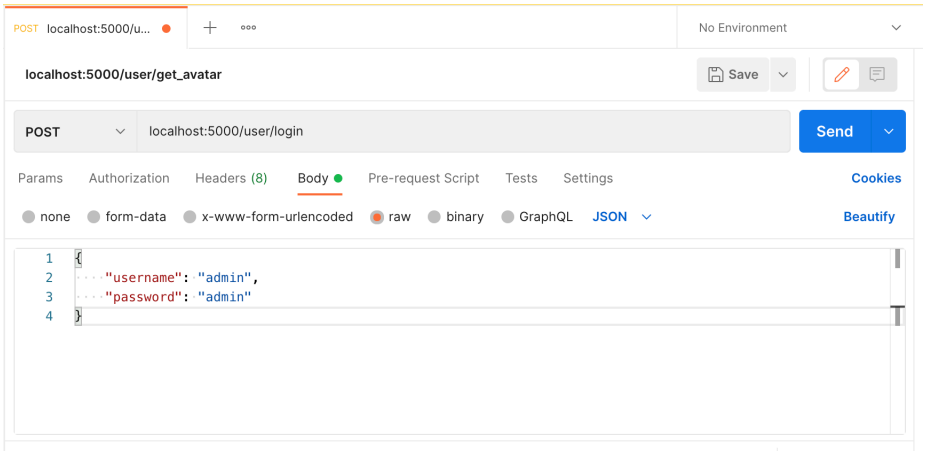

Figure15.1: 单次测试请求体

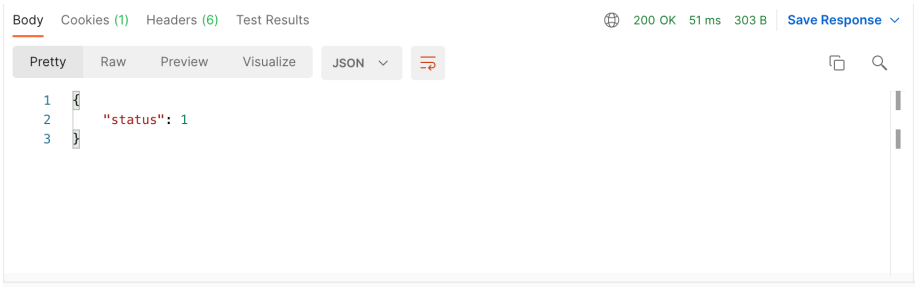

Figure15.2: 单次测试响应结果

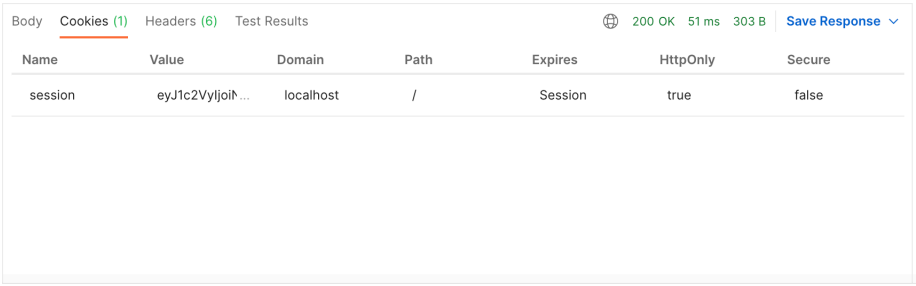

Figure15.3: 单次测试响应附带的 Session 内容

可以看到，系统正确返回了预期的结果，表明了接口的可用性。

进一步的，对于 Flask 应用本身，我们可以使用 Python 提供的 **unittest** 模块来对后端应用进行测试。这里我们模拟了一组完整的玩家操作请求序列来对应用中的全部接口进行流水线测试，测试代码如下：

```
class TestApp(unittest.TestCase):
    def setUp(self) -> None:
        app = create_app()
        app.config['TESTING'] = True
        self.app = app.test_client()

    def test_a_register(self):
```

```
8           response = self.app.post('/user/register', data=json.dumps({'username
                ': 'test6', 'password': 'test6'}))
9           status = json.loads(response.data)['status']
10          self.assertEqual(status, 1)
11
12      def test_b_login(self):
13          response = self.app.post('/user/login', data=json.dumps({'username':
                'admin', 'password': 'admin'}))
14          status = json.loads(response.data)['status']
15          self.assertEqual(status, 1)
16
17      def test_c_user_info(self):
18          response = self.app.get('/user/info')
19          status = json.loads(response.data)['status']
20          self.assertEqual(status, 1)
21
22      def test_d_set_nickname(self):
23          response = self.app.post('/user/set_nickname', data=json.dumps({'
                nickname': 'testName'}))
24          status = json.loads(response.data)['status']
25          self.assertEqual(status, 1)
26
27      def test_e_seek(self):
28          response = self.app.get('/game/seek')
29          data = json.loads(response.data)
30          status = data['status']
31          self.assertEqual(status, 1)
32
33      def test_f_work(self):
34          response = self.app.get('/game/work')
35          status = json.loads(response.data)['status']
36          self.assertEqual(status, 1)
37
38      def test_g_list_container(self):
39          response = self.app.get('/container/list')
40          data = json.loads(response.data)
41          status = data['status']
42          self.targetId = data['items'][0]['id']
43          self.assertEqual(status, 1)
44
45      def test_h_wear(self):
46          response = self.app.post('/container/wear', data=json.dumps({'id':
                self.targetId}))
47          status = json.loads(response.data)['status']
48          self.assertEqual(status, 1)
49
50      def test_i_equipment(self):
51          response = self.app.get('/user/equipment')
52          status = json.loads(response.data)['status']
53          self.assertEqual(status, 1)
54
55      def test_j_takeoff(self):
56          response = self.app.post('/container/takeoff', data=json.dumps({'id':
```

```
            self.targetId}))
        status = json.loads(response.data)['status']
        self.assertEqual(status, 1)

    def test_k_sell(self):
        response = self.app.post('/container/sell', data=json.dumps({'id':
            self.targetId, 'price': 10}))
        status = json.loads(response.data)['status']
        self.assertEqual(status, 1)

    def test_l_list_market(self):
        response = self.app.get('/market/list')
        status = json.loads(response.data)['status']
        self.assertEqual(status, 1)

    def test_m_abort_sell(self):
        response = self.app.post('/container/abortsell', data=json.dumps({'id
            ': self.targetId}))
        status = json.loads(response.data)['status']
        self.assertEqual(status, 1)
```

测试结果如下图所示 (Figure16)。可以看到，该应用中的所有接口均可以处理各种边界情况，且由于多线程支持及高并发框架的引入，服务器的吞吐量上限能达到一个很高的峰值。

```
gongjingyang@GONGGONGJOHNs-Macbook backend % python3.8 test_app.py
test_a_register (__main__.TestApp) ... ok
test_b_login (__main__.TestApp) ... ok
test_c_user_info (__main__.TestApp) ... ok
test_d_set_nickname (__main__.TestApp) ... ok
test_e_seek (__main__.TestApp) ... ok
test_f_work (__main__.TestApp) ... ok
test_g_list_container (__main__.TestApp) ... ok
test_h_wear (__main__.TestApp) ... ok
test_i_equipment (__main__.TestApp) ... ok
test_j_takeoff (__main__.TestApp) ... ok
test_k_sell (__main__.TestApp) ... ok
test_l_list_market (__main__.TestApp) ... ok
test_m_abort_sell (__main__.TestApp) ... ok

----------------------------------------------------------------------
Ran 13 tests in 0.289s

OK
```

Figure16: Unittest 单元测试结果