gStore 夏季课程作业

Python v3.7.6 flask v1.1.2 gunicorn v10.0.4 guniflask v0.7.2 Test with Postman
Code With Pycharm

Made with Markdown Write with Typora Author 郭一涵 ID 0906140129

作业目标

利用提供的金融数据集（见本文档最后位置的超链接），搭建以 `gStore` 为管理平台的小型金融知识图谱；并实现**任务一**、**任务二**、**任务三**的功能，返回正确的结果。(建议将gstore作为知识图谱存储介质，自己搭建应用平台完成实验)

平台设计

由于平常一直在组内单人负责一个大模块，我习惯于结合Flask和Gunicorn使用Guniflask来搭建一个RESTful风格的后台API服务器并提供大量服务。因此，本次平台设计依旧以Guniflask来作为框架，RESTful作为接口的规范。相应的依赖版本已经在本次报告开始就以标签的形式附上。相应在Github上的地址为：Gstore

文件结构

```
├── Dockerfile
├── bin
│   ├── app-config.sh
│   └── manage
├── conf
│   ├── app-env.sh
│   ├── gstore.py
│   └── gunicorn.py
├── docker-compose.yml
├── gstore
│   ├── __init__.py
│   ├── api
```

```
|   |   ├── __init__.py
|   |   ├── api.py
|   |   └── gstore.py
|   ├── app.py
|   ├── config
|   |   ├── __init__.py
|   |   └── security_config.py
|   └── examples
|       ├── __init__.py
|       └── hello_world.py
├── requirements
|   ├── app.txt
|   └── test.txt
└── tests
    └── __init__.py
```

整体文件结构如图所示，其中需要解释的地方有：

- `conf/gstore.py`:用于记录本平台的一些基本设置，这里主要记录了`GSTORE_URL`、`GSTORE_SECRET`和`GSTORE_KEY`三个系统级设置项
- `conf/gunicorn.py`：用于设置平台监听的端口，这里设置为了8000
- `gstore/api/api.py`：相应功能接口的实现文件，所有的接口都是在这里实现的
- `gstore/api/gstore.py`：用于将对`gstore`的请求封装，作为一个类直接调用
- `requirements/app.txt`:用来记录相应的依赖

接口解释

由于所有的接口都在类Gstore中实现，这里主要介绍一下相应的接口和对应的调用，详细的设计将会在下边每一道题中做详细的解释

Class Gstore

这个类下的所有接口都注册了根路径为：`/api/gstore`

Func _ _ init _ _()

这个函数主要在类实例化的时候顺达加载上一个实例化的`GstoreAPI`，方便后边直接调用来对学校的`gstore`云进行`http`请求

Func directRoutes_response()

这个函数主要用来响应采取 `POST` 方法到 `/api/gstore/directRoutes` 路径的事件，第一题的答案就是从这里获得

- **Func _ _ query _ direct _ routes _ _()**

这个函数主要是用来实现 `directRoutes` 功能

- **Func penetratingQuery_response()**

这个函数主要用来响应 `POST` 到 `/api/gstore/penetratingQuery` 路径的事件，第二题的答案就是从这里获得

- **Func _ _ get _ pre _ entity _ _()**

这个函数主要是用来获得给定公司向前连续n跳的所有公司

- **Func _ _ get_next_entity_ _()**

这个函数主要是用来获得给定公司向前连续n跳的所有公司

- **Func ringRoutes_response()**

这个函数主要用来响应 `POST` 到 `/api/gstore/ringRoutes` 路径的事件，第三题的答案就是从这里获得

- **Func _ _ check _ ring _ routes _ _()**

这个函数主要是用来实现 `ringRoutes` 功能

- **Func _ _ check _ include _ _()**

这个函数主要是检查给定的公司是否在给定的公司链条中，是的话返回相应的链条

- **Func _ _ analysis _ _()**

这个函数主要是用来将 `gstore` 返回的结果整理为文字形式的链条，方便理解，如：江苏金盛实业投资有限公司->上海天发投资有限公司

- **Class GstoreAPI**

这个类主要用于封装对 `gstore` 云平台的请求

- **Func exec()**

这个函数主要用来对外提供封装好了的请求服务，这样的话就能够将 `gstore` 平台视为一个数据库了

- **Func _ _ exec _ sparql _ _()**

这个函数主要用来真实执行 `exec` 的逻辑

- **平台运行管理**

```
cd gstore
bash bin/manage start/stop/debug
```

2-hop 关联路径

利用构建好的知识图谱，编写sparql语句查询两个公司之间的关联路径（2-hop）。例如输入公司“招商局轮船股份有限公司”和“招商银行股份有限公司”，得到这两家公司之间的所有路径。

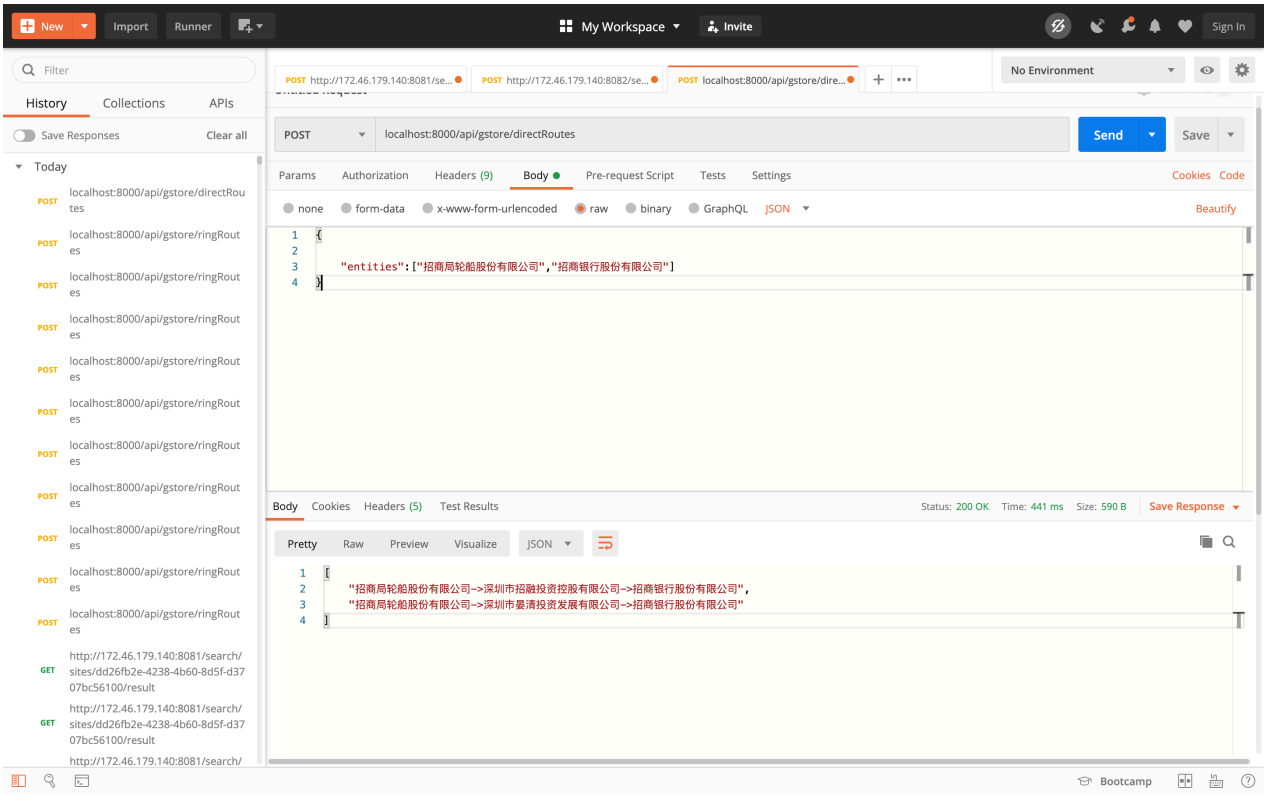
这个很简单，构造好相应的请求就好了，在构造的时候多写一层就没问题了

```
    def __query_direct_routes__(self, entities: List[AnyStr], hop: int) ->
Optional[List[Dict]]:
        """
        用于查询两两实体之间的关联路径,先直接写死两跳
        @param entities: 实体列表，在这里是公司的名称
        @param hop:关联的跳数
        @return: 用于展示的List，每一个词典都是两两之间的关系
        """
        final_result = []
        # 便利任意两个（有序）
        for each_combine in list(itertools.permutations(entities, 2)):
            # 构造相应的sql
            sql = "select  * where { <file:///F:/d2r-server-
0.7/holder8.nt#holder_copy/" + \
                  each_combine[0] + ">" + \
                  "?p ?o . ?o ?q  " + \
                  "<file:///F:/d2r-server-0.7/holder8.nt#holder_copy/" +
each_combine[1] + "> }"
            response = self.gstore_api.exec(db="jinrong", sparql=sql)
            print(response)
            if response:
                tmp_result = {
                    "head": each_combine[0],
                    "tail": each_combine[1],
```

```
                "bindings": response['data']['results']['bindings'],
                "index": response['data']['head']['vars']
            }
            final_result.append(tmp_result)
    return final_result
```

其实原先我打算去写一个n跳的，但是给忘了了

- **实验结果**



N-hop 穿透查询

可以根据指定层数获得对应层级的股东，例如：输入“招商局轮船股份有限公司””和层数3，就会把“招商局轮船股份有限公司”所对应公司所有三层以内的公司找出来。

这个由于不太确定，我把双向的都写了，同时，为了方便，我直接设置为了可以随意设定相应的层级，具体的实现方法很简单，就是一个for循环直接循环出来相应的sparql中的hop数量

```
@post_route('/penetratingQuery')
def penetratingQuery_response(self):
    """
    用来实现多层股权的穿透式查询
```

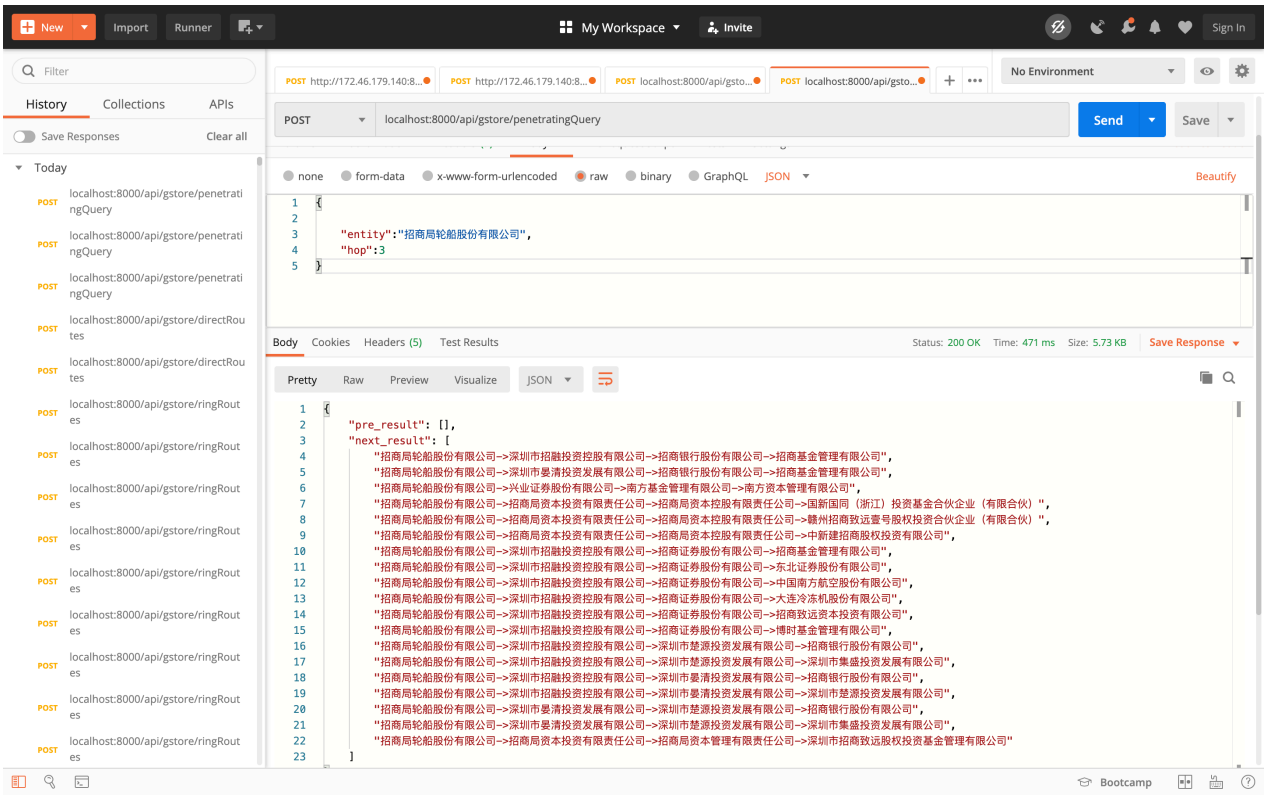
```
        @return:
        """
        data = request.get_json()
        entity = data['entity']
        hop = data['hop']
        result = dict(
            pre_result=self.__get_pre_entity__(entity, hop),
            next_result=self.__get_next_entity__(entity, hop)
        )
        return json.dumps(result)


    def __get_pre_entity__(self, entity: AnyStr, hop: int) -> List[AnyStr]:
        """
        获得上n跳的所有实体
        @param entity: 本级实体
        @param hop: 跳数
        @return: 上一级的所有实体
        """
        candidate = ["?a", "?b", "?c", "?d", "?e", "?f", "?g", "?h", "?i", "?
j", "?k", "?l", "?m", "?n"]
        raw_node = []
        for each_hop in range(hop):
            s = candidate[2 * each_hop]
            p = candidate[2 * each_hop + 1]
            o = candidate[2 * each_hop + 2]
            raw_node += [s, p, o, "."]
        # 构造相应的sparql句子
        # 首先要去掉最后两个，因为分别是entity和相应的"."
        raw_node = raw_node[:-2]
        raw_sql = " ".join(raw_node)
        sql = "select  * where { " + raw_sql + " <file:///F:/d2r-server-
0.7/holder8.nt#holder_copy/" + entity + ">  }"
        # sql = "select  * where { ?p ?o <file:///F:/d2r-server-
0.7/holder8.nt#holder_copy/" + entity + ">  }"
        response = self.gstore_api.exec(db="jinrong", sparql=sql)
        if response:
            routes = {
                "head": None,
                "tail": entity,
                "bindings": response['data']['results']['bindings'],
                "index": response['data']['head']['vars']
            }
            result = __analysis__(routes)
        return result

    def __get_next_entity__(self, entity: AnyStr, hop: int) -> List[AnyStr]:
```

```
        """
        获得下n跳的所有实体
        @param entity:本级实体
        @param hop:相应的跳数
        @return: 下一级的所有实体
        """
        candidate = ["?a", "?b", "?c", "?d", "?e", "?f", "?g", "?h", "?i", "?
j", "?k", "?l", "?m", "?n"]
        raw_node = []
        for each_hop in range(hop):
            s = candidate[2 * each_hop]
            p = candidate[2 * each_hop + 1]
            o = candidate[2 * each_hop + 2]
            raw_node += [s, p, o, "."]
        # 构造相应的sparql句子
        # 去掉最开始的头
        del raw_node[0]
        del raw_node[-1]
        raw_sql = " ".join(raw_node)
        sql = "select  * where { "   " <file:///F:/d2r-server-
0.7/holder8.nt#holder_copy/" + entity + "> " + raw_sql + "  }"
        # sql = "select  * where { ?p ?o <file:///F:/d2r-server-
0.7/holder8.nt#holder_copy/" + entity + ">  }"
        response = self.gstore_api.exec(db="jinrong", sparql=sql)
        if response:
            routes = {
                "head": entity,
                "tail": None,
                "bindings": response['data']['results']['bindings'],
                "index": response['data']['head']['vars']
            }
            result = __analysis__(routes)
        return result
```

- **实验结果**



环形持股查询

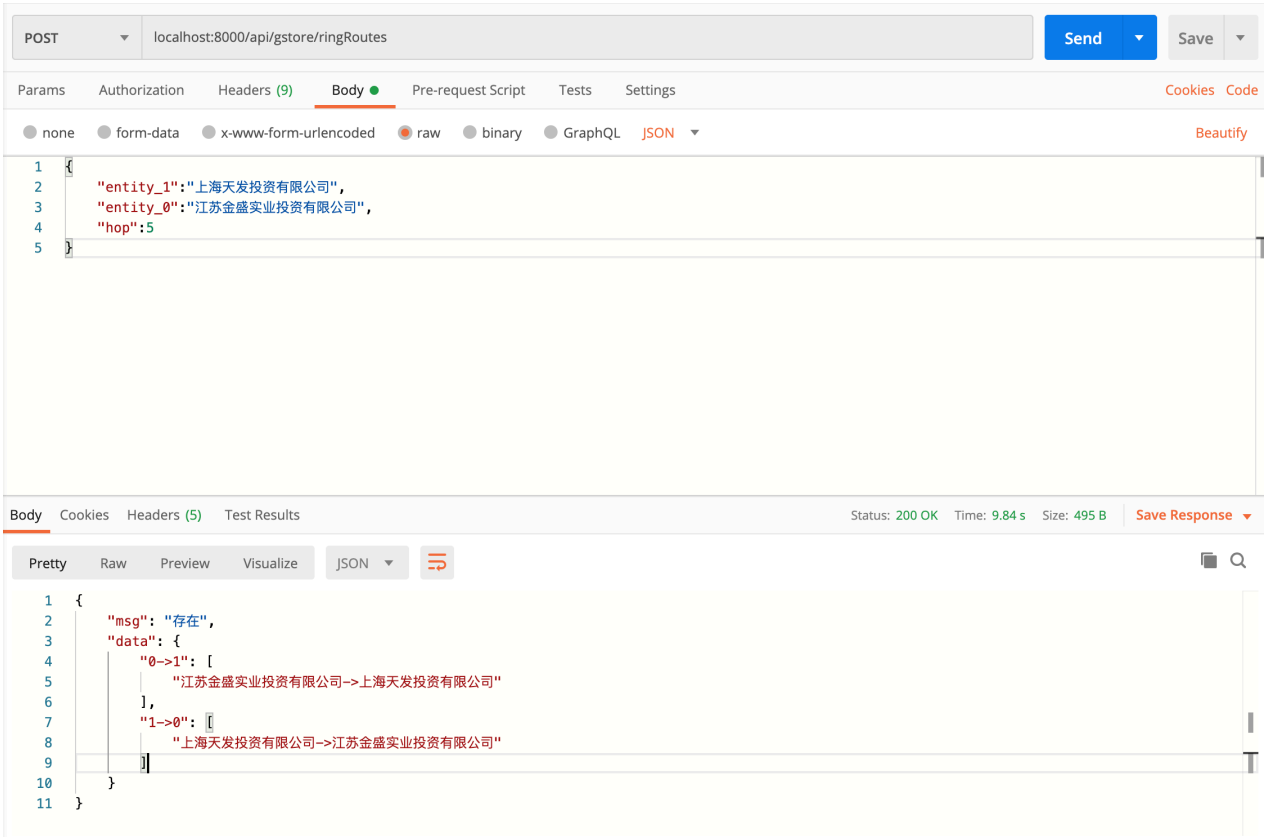
判断两家公司是否存在环形持股现象，环形持股是指两家公司彼此持有对方的股份。例如：输入“A”和“C”，判断两家公司是否存在环形持股。

我在设计这个的时候考虑到sparql语言很难直接对任意跳数直接查询，因此我设计成为了在给定最大跳数下判断是否存在环形持股，如实验结果所示，hop=5指最大5跳之内判断。整个的实现思路是这样的，首先获得两个实体每一个最大n跳的后续，之后在对方后续中查找，看是否自己存在，若存在截取相应的chain并且返回，方法很粗暴，下边是部分函数

```
@post_route('/ringRoutes')
def ringRoutes_response(self):
    """
    用来查询两家公司是否存在环形持股现象
    @return:
    """
    data = request.get_json()
    entity_0 = data['entity_0']
    entity_1 = data['entity_1']
    hop = data['hop']
    result = self.__check_ring_routes__(entity_0, entity_1, hop)
    return json.dumps(result)
```

```
    def __check_ring_routes__(self, entity_0: AnyStr, entity_1: AnyStr, hop:
int) -> Dict:
        """
        用来检验两家公司是否存在环形持股
        @param entity_0: 第一家公司
        @param entity_1: 第二家公司
        @param hop:指定的环大小（单侧最长）
        @return: True: 存在, False: 不存在
        """
        # 根据群里说的不超过五跳，直接获得五跳之内的，然后做查询就好了
        entity_0_next = self.__get_next_entity__(entity_0, hop)
        entity_1_next = self.__get_next_entity__(entity_1, hop)
        chain0_1 = __check_include__(entity_0, entity_1_next)
        chain1_0 = __check_include__(entity_1, entity_0_next)
        if chain0_1 == [] or chain1_0 == []:
            return {"msg": "Do not exists"}
        # 然后对链路进行处理
        # 从entity0开始的链路
        print(chain0_1)
        print('\n\n')
        print(chain1_0)
        chain0 = list(set([x[:x.index(entity_1) + len(entity_1)] for x in
chain1_0]))
        chain1 = list(set([x[:x.index(entity_0) + len(entity_0)] for x in
chain0_1]))
        return {
            "msg": "存在",
            "data": {
                "0->1": chain0,
                "1->0": chain1
            }
        }
```

- **实验结果**




```
{
    "msg": "存在",
    "data": {
        "0->1": [
            "江苏金盛实业投资有限公司->上海天发投资有限公司"
        ],
        "1->0": [
            "上海天发投资有限公司->江苏金盛实业投资有限公司"
        ]
    }
}
```