基于devrest的graphql拓展实践

课题介绍

本课题是在devrest的基础上，结合graphql为api数据查询提供的灵活的查询，复用原来SQLAlchemy定义的模型，支持原devrest 分页、过滤、搜索、子关系查询、排序、关系展开、黑白名单等功能，提供多种数据聚合，并实现Mysql、Mongo、RestAPI 接入抽象层的示例。

方案思路

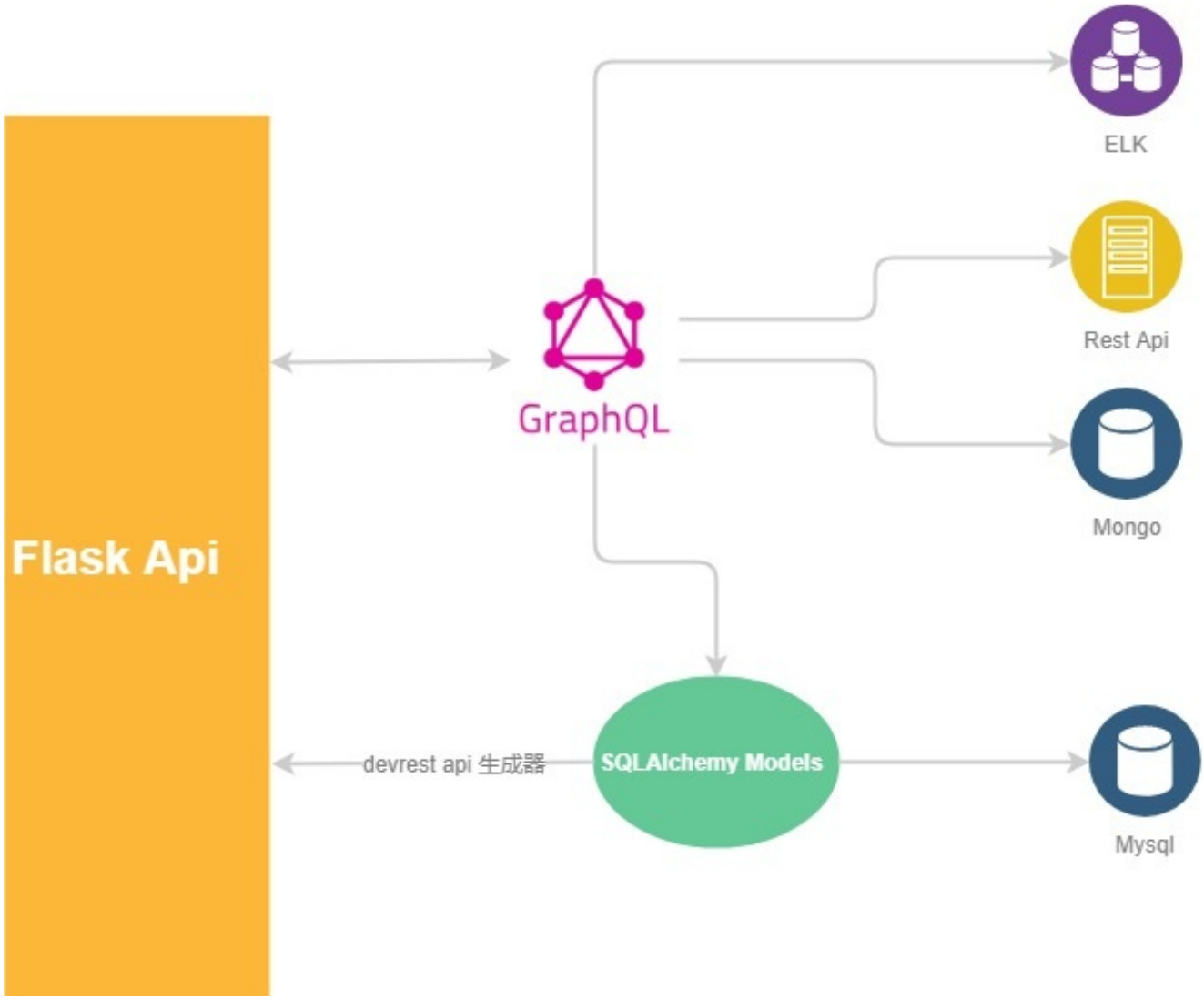
1. 为在devrest做二次开发、并保持devrest功能正常运行的基础上，考虑复用sqlalchemy的model，并在此基础上进行扩张，在数据库orm层的基础上进行封装，将多种数据库模型类转化为graphql类型，并提供多种数据库的相互转换以及用户自定义(兼具便利性以及灵活性)。

2. 再为通过转化的graphql类型定义schema所需要query。并在此基础上丰富查询接口参数(从用户的角度看：最好和devrest的查询方式保持一致)，根据查询的方式分为：1.单向查询(只需把model参数映射到graphql的自省模块中)2.列表查询(处理外键的关系以及Field与node相互结合的方式)

3. 定义三种resolve处理函数，当请求经过flask API后，便是通过在view层定义的excutor执行graphql树状模型来找到所定义的resolve函数并进行数据处理并返回给graphql抽象层。

4. graphql抽象层匹配请求所需的字段过滤并返回用户真实需要的字段。

方案设计图



目录结构及其作用

```
├── auth        ------------------------------ 权限控制
│   ├── authconnection.py
│   └── permisssion.py
├── elk         ------------------------------ elasticsearch
│   ├── __init__.py
│   └── search.py
├── __init__.py
├── models      ------------------------------ graphql生成模块
│   ├── base.py         ---------------------- 基类
│   ├── connection.py   ---------------------- 自定义游标分页
│   ├── converter.py    ---------------------- 类型转化
│   ├── dataloader.py   ----------------------  批量加载
│   ├── __init__.py
│   ├── registry.py      ---------------------  注册类
│   ├── rest.py          ---------------------  第三方调用
│   └── utils.py         ---------------------  工具类
├── mutations.py
├── object.py           ----------------------  自定义graphql对象
├── query.py            ----------------------  生成查询query
```

```
    ├── schema.py            --------------------  生成schenma
    └── utils.py             --------------------   工具类
```

基本功能点

- 支持复用sqlalchemy模型，并提供mongo模型API
- 支持原devrest分页、过滤、搜索、子关系查询、排序、黑白名单等功能
- graphql支持多数据源聚合(mysql,mongo,Rest APi、ELK)等功能
- 简单易用，使用理念与devrest保持一致，并提供用户自定义

Todos

- auth(功能点)
- dataloader(bug点)

快速入门

使用setup.py进行安装

```
git checkout graphql
git clone ssh://git@git-sa.nie.netease.com:32200/devteam/devrest.git
cd devrest && pip install -r requirements_prod.txt && python setup.py install
```

example

model定义

- 定义sqlalchemy | mongoengine的model，再使用create_query_schema函数，将model转化的graphql传递进去，绑定路由到app，即可实现graphql灵活的查询功能。
- 若需要调用第三方公共接口或者多种数据库中的聚合，提供graphql对象自定义，覆盖之前生成api的graphql对象，方便用户自行处理更加实际复杂的需求。

代码地址
[用例涵盖用例涵盖模型之间多种关联关系(sqlalchemy ｜ mongoengine)]

schema定义

- 生成graphql schemaAPI，database为sqlalchemy模型定义，mongodatabase为mongoengine模型定义，connect_schema_gen为graphql中root schema生成函数，总起graphql入口。

```
from devrest.graphqldev.schema import connect_schema_gen
import database
import mongodatabase
schema = connect_schema_gen(mongodatabase)
```

- 用户自定义graphql对象：

```
class GithubCall(ObjectType):
    class Meta:
        interfaces = (RestCall,)
```

```
    github = String()
    github_status = String()

class Users(ObjectType):
    class meta:
        model = User
    hello = Field(String)
    name = Field(String)
    rest_call = List(GithubCall, name=String())
    def resolve_hello(self, args, context, info):
        return "hello"
    def resolve_name(self, args, context, info):
        return "liuhang"
    def resolve_rest_call(self, args, context, info):
        if args:
            name = args.get("name")
            import requests
            github_res = requests.get(
                'https://api.github.com/users/%s' % name
            )
            rest_uri = 'https://api.github.com/users/%s' % name
            description = "third call"
            github = yaml.safe_load(github_res.text)
            github_status = github_res.status_code
            githubcall = GithubCall(rest_uri=rest_uri, description=descript
ion,
                                    github=github, github_status=github_sta
tus)
            result = [githubcall]
            return result
        return None
GRAPQLDICT = {
    "graphql_object": [Users]
}

schema = connect_schema_gen(models, **GRAPQLDICT)
webapp.add_url_rule('/graphql', view_func=GraphQLView.as_view('graphql',
                    schema=schema, graphiql=True))
```

  - 通过定义user的graphql对象，只需将模型放入meta类中，如若存在需求，可定义column覆盖之前的Field，但是同时需要提供相应的resolve处理函数(由于Field对应一个resolve解析,否则无效)，如上图中的 user对象，自定义的hello Field。
  - 通过接口的方式继承rest定义第三方服务，rest中已经添加了rest_uri和description相关必要字段，然后将服务的graphql对象加入需要的对象中，如上图中的user类中。
  - 通过引入devrest.graphqldev中utils下的PageArg类和get方法即可实现参数的分页以及过滤等功能(e.g:resourse_list = PageArgs(args).get(model).all())即可得到查询的数据,arg即查询参数。

- 在**connect_schema_gem**函数中加上**GRAPQLDICT**即可实现用户定制(参数名暂时写死了，后面会做调整)。

app定义

```
from flask import Flask, render_template
from flask_graphql import GraphQLView
from database import db_session
from schema import schema
app = Flask(__name__)
app.add_url_rule('/graphql', view_func=GraphQLView.as_view('graphql', schema=sch
ema, graphiql=True, context={'session': db_session}))
```

运行app

```
 > python app.py
Running on http://127.0.0.1:5000/
```

功能点效果展示

模型节点查询

在嵌套关联的模型中对各个模型节点的查询，支持分页，过滤等功能(1.采用connection+edge+node2.采用List均可)

connection+edge+node查询

```
输入
   {
  allUser(Num:4,Page:1){
    id
    email
    name
    phone
    groups(first:4){
      edges{
        node{
          code
          description
          project
            Project{
            costCode
            description
          }
          permissions(last:4){
            edges{
              node{
                name
                description
              }
            }
          }
        }
      }
    }
  }
}
```

输出


```
{
  "data": {
    "allUser": [
      {
        "id": "1",
        "email": "xkcn1748@corp.netease.com",
        "name": "liuhang",
        "phone": "18665589783",
        "groups": {
          "edges": [
            {
              "node": {
                "code": "ma38.dev",
                "description": "",
                "fullname": "Castle Doombad程序",
                "project": "ma38",
                "Project": {
                  "costCode": "D056200205",
                  "description": "反派视觉策略塔防",
                  "developPlace": "广州"
                },
                "permissions": {
                  "edges": [
                    {
                      "node": {
                        "id": "13",
                        "name": "用户账号编辑、发布",
                        "description": "用户控制列表编辑、发布"
                      }
                    },
                    {
                      "node": {
                        "id": "15",
                        "name": "查看用户控制列表",
                        "description": "查看用户控制列表信息"
                      }
                    },
                    {
                      "node": {
                        "id": "16",
                        "name": "内部服务器列表编辑、发布",
                        "description": "管理内部服务器列表"
                      }
                    },
                    {
                      "node": {
                        "id": "17",
                        "name": "内部服务器列表查看",
                        "description": "查看内部服务器列表条目"
                      }
                    }
                  ]
                }
```

```
              }
            },
```

list查询

```
输入
{
  allUser(Num:4,Page:1){
    id
    email
    no
    name
    groupslist(Num:3,Page:2){
      id
      description
      code
      name
      userslist(Num:3,Page:3){
        id
        email
        no
        name
      }
    }
  }
}
输出
{
  "data": {
"allUser": [
  {
    "id": "1",
    "email": "xkcn1748@corp.netease.com",
    "no": "N1748",
    "name": "liuhang",
    "groupslist": [
      {
        "id": "4",
        "description": "",
        "code": "f4.dev",
        "name": "程序",
        "userslist": [
          {
            "id": "7",
            "email": "gzzhoutianya@corp.netease.com",
            "no": "G5631",
            "name": "liuhang"
          },
          {
            "id": "8",
            "email": "hzzhangpeng2013@corp.netease.com",
            "no": "H2837",
            "name": "liuhang"
          },
```

```
          {
            "id": "9",
            "email": "gzpengxin@corp.netease.com",
            "no": "G5989",
            "name": "liuhang"
          }
        ]
      },
```

第三方调用

第三方调用灵活添加到相关的模型输出

```
#输入请求体
{
  allUser(Num:4){
    email
    no
    restCall(name:"liuhang"){
      description
      restUri
      githubStatus
    }
  }
}
#输出
  {
  "data": {
   "allUser": [
      {
        "email": "xkcn1748@corp.netease.com",
        "no": "N1748",
        "restCall": [
          {
            "description": "third call",
            "restUri": "https://api.github.com/users/liuhang",
            "githubStatus": "200"
          }
        ]
      }...
```

mongoengine数据模型输出

3.1 mongoengine定义mongoengine模型

```
from datetime import datetime
from mongoengine import Document, EmbeddedDocument
from mongoengine.fields import (
    DateTimeField, EmbeddedDocumentField,
    ListField, ReferenceField, StringField,
)
class Department(Document):
    meta = {'collection': 'department'}
```

```
    name = StringField()
class Role(Document):
    meta = {'collection': 'role'}
    name = StringField()
class Task(EmbeddedDocument):
    name = StringField()
    deadline = DateTimeField(default=datetime.now)
class Employee(Document):
    meta = {'collection': 'employee'}
    name = StringField()
    hired_on = DateTimeField(default=datetime.now)
    department = ReferenceField(Department)
    roles = ListField(ReferenceField(Role))
    leader = ReferenceField('Employee')
    tasks = ListField(EmbeddedDocumentField(Task))
```

3.2将模型输入到生成的schema中

```
from devrest.graphqldev.schema import connect_schema_gen
import database
import mongodatabase
# from mutations import MyMutations
schema = connect_schema_gen(mongodatabase)
```

3.3查询效果

```
输入
{
  allEmployee{
    id
    hiredOn
    name
    department{
      id
      name
    }
    roles(first:1){
      edges{
        node{
          id
          name
        }
      }
    }
  }
}
输出
{
  "data": {
    "allEmployee": [
      {
        "id": "5b5c34881d41c83b71f400b9",
        "hiredOn": "2018-07-28 02:16:56.191825",
        "name": "Tracy",
```

```
        "department": {
          "id": "5b5c34881d41c83b71f400b6",
          "name": "Human Resources"
        },
        "roles": {
          "edges": [
            {
              "node": {
                "id": "5b5c34881d41c83b71f400b7",
                "name": "manager"
              }
            },
            {
              "node": {
                "id": "5b5c34881d41c83b71f400b8",
                "name": "engineer"
              }
            }
          ]
        }
      },
```

dataloader的使用效果

(在使用之前请参考文档dataloader原理再使用)

- 当关联节点采用batch_list节点的模型时，自动采用dataloader进行性能优化，解决request n+1 question
- 当前不能在connection节点上使用dataloader(bug未解决)
- 无法在使用dataloader的节点下使用参数进行查询(dataloader采用的原理是建一个队列，然后一个key对应一个value，在批量收集父模型节点的bug)

readme

readme**欢迎进行尝试使用和交流**

性能优化

- 列表多项查询将关联键sql同时批量执行(解决n+1 request question)：graphql存在 n+1 request 问题，因为父节点如果产生n条数据，若在基础上进行子查询，则存在n次子查询，严重的性能不足，采用dataloader即可解决这个问题，但是灵活度不高。
- 如果请求内容比较多，可以将请求体散列化，后端做好缓存与映射(暂时为未完成)
- 可以在resolve函数的时候，通过解析语法模型树AST获取当前输入的字段，节省不必要的花销

目前存在的问题

- dataloader无法做出边查询、边批量加载的性能提升
- elk目前还存在ｂｕｇ
- mongoengine的embeddeddocument暂时无法查询出来
-

总结

graphql是专为查询api而生的语言，强调一点，它可以是面对各种复杂场景用rest解决比较麻烦的时候的另一种方案，即插即用，并不会影响 像devrest等生成查询api的使用，graphql更是一种拓展和补充。