GraphQL

介绍

GraphQL是Facebook开发的一种数据查询语言，并于2015年公开发布。它是REST API的替代品。GraphQL既是一-种用于 API的查询语言也是一-个满足你数据查询的运行时。GraphQL 对你的API中的数据提供了一套易于理解的完整描述，使得客户端能够准确地获得它需要的数据，而且没有任何冗余，也让API更容易地随着时间推移而演进。

特点

1. 请求需要的数据，不多不少

   例如：account中有name, age, sex, department等。可以只取得需要的字段。

2. 获取多个资源，只用一个请求描述所有可能类型的系统。便于维护，根据需求平滑演进，添加或者隐藏字段。

GraphQL与restful的区别

restful: Representational State Transfer表属性状态转移。本质上就是用定义uri,通过api接口来取得资源。通用系统架构,不受语言限制。

GraphQL与restful的对比

restful一个接口只能返回一个资源，graphql 一次可以获取多个资源。restful用不同的url来区分资源，graphq|用类型区分资源。GraphQL API比REST API具有更多的基础结构，有更强大的客户端（如Relay）可以自动处理批处理，缓存和其他功能。但是，不需要复杂的客户端即可调用GraphQL服务器。

使用GraphQL和express

1. 创建一个GraphQL文件夹
2. 使用命令行或终端跳转到该目录，输入 `npm init -y` 创建 `package.json`
3. 安装依赖包，终端输入 `npm install express graphql express-graphql -S`
4. 在目录下创建一个HelloWorld.js文件并输入以下代码

```
1.  //引入模块
2.  const express = require('express');
3.  const {buildSchema} = require('graphql');
```

```
const grapqlHTTP = require('express-graphql');
// 定义schema，查询和类型
const schema = buildSchema(`
 type Query {
 hello: String
 }
`)
// 定义查询对应的处理器
const root = {
 hello: () => {
 return 'hello world';
 }
}
//实例化express
const app = express();
//访问到地址/graphql，交给grapqlHTTP来处理，其中第一个是我们定义好的schema，第二个是处理器函数，而`graphiql: true` 表示启用调试界面
app.use('/graphql', grapqlHTTP({
 schema: schema,
 rootValue: root,
 graphiql: true
}))

app.listen(3000);
```

在上述代码中，其中 `grapqlHTTP` 使用来结合 `express` 和 `graphql` 的一个中间件。

`const {buildSchema} = require('graphql');` 从 `graphql` 引入的 `buildSchema` 方法是用来构建schema，定义查询语句和类型的。在这个 `schema` 中定义了一个查询语句，里面定义了一个字段 `hello` ，返回值类型是 `String` 。

`root` 中定义的是查询语句的处理器，上面定义的查询语句吗，在这里定义一个对应的处理器函数 `hello` ，并且返回值类型需要是上面定义好的 `String` 类型。

1. 在终端输入 `node Helloword.js` 启动服务，浏览器打开 `http://localhost:3000/graphql` 就能看到graphql的调试界面，界面分三个区域，左边的区域是查询语句区域，中间的是查询结果，右边区域显示的是你能够查询到的数据和使用的方法。

> *你也可以安装nodemon，用nodemon代替node命令启动服务，这样每次修改代码之后服务就会自动启动，*
>
> 1. *安装 `npm install nodemon -g`*
> 2. *启动 `nodemon HelloWorld.js`*

1. 在查询区域，你可以查询任何你定义好的字段，如

```
query{
 hello
}
//输出结果
 {
  "data": {
  "hello": "hello world"
  }
  }
}
```

参数类型和参数传递

基本参数类型

1. 基本类型：String，Int，Float，Boolean和ID。可以在shema声明的时候直接使用。

2. [类型]代表数组，例如：[Int]代表整型数组

参数传递

1. 和js传递参数一样，小括号内定义形参，但是注意:参数需要定义类型。

2. !(叹号)代表参数不能为空。

```
 type Query{
  rollDice(numDice:Int!,numSide: Int):[Int]
  //定义一个rollDice的方法，其中有两个参数，一个是numDice，类型为Int，且不为空，第二个是numSide，类型是Int，函数返回值是一个Int类型的数组
 }
```

在刚刚的目录下创建一个叫做 `baseType.js` 的文件

```
const express = require('express');
const {buildSchema} = require('graphql');
const grapqlHTTP = require('express-graphql');
// 定义schema，查询和类型
const schema = buildSchema(`
 type Query {
 getClassMates(classNo: Int!): [String]
 }
```

```
`)
// 定义查询对应的处理器
const root = {
 getClassMates({ classNo}) {
 const obj = {
 31: ['张三', '李四', '王五'],
 61: ['张大三', '李大四', '王大五']
 }
 return obj[classNo];
 }
}

const app = express();

app.use('/graphql', grapqlHTTP({
 schema: schema,
 rootValue: root,
 graphiql: true
}))

// 公开文件夹，供用户访问静态资源
app.use(express.static('public'))

app.listen(3000);
```

用node命令运行baseType.js

```
node baseType.js
```

在浏览器的调试界面调试

```
//查询
query{
 getClassMates(classNo: 31)
}

//返回结果
{
 "data": {
 "getClassMates": [
 "张三",
 "李四",
 "王五"
```

```
  ]
  }
 }
//如果你不输入参数，就会报如下错误
 {
  "errors": [
  {
  "message": "Syntax Error: Expected Name, found )",
  "locations": [
  {
  "line": 2,
  "column": 17
  }
  ]
  }
  ]
 }
```

自定义参数类型

GraphQL允许用户自定义参数类型，通常用来描述要获取的资源的属性。你可以通过type来声明你自己的类型，例如定义 `Account` 类型。自定义参数中可以是多个类型的结合，也可以是函数类型。当然，处理器定义的时候，你也需要是对应的函数类型。修改 `baseType.js` 如下

```
// 定义schema，查询和类型
const schema = buildSchema(`
  type Account {
  name: String
  age: Int
  sex: String
  department: String
  salary(city: String): Int
  }
  type Query {
  getClassMates(classNo: Int!): [String]
  account(username: String): Account
  }
`)
// 定义查询对应的处理器
const root = {
  getClassMates({ classNo}) {
```

```
  const obj = {
  31: ['张三', '李四', '王五'],
  61: ['张大三', '李大四', '王大五']
  }
  return obj[classNo];
  },
  account({ username}) {
  const name = username;
  const sex = 'man';
  const age = 18;
  const department = '开发部';
  const salary = ({city}) => {
  if(city === "北京" || city == "上海" || city == "广州" || city == "深圳") {
  return 10000;
  }
  return 3000;
  }
  return {
  name,
  sex,
  age,
  department,
  salary
  }
  }
 }
```

你可以在调试界面输入查询试试：

```
query{
 account(username: "张三"){
 name
 sex
 age
 salary(city: "北京")
 }
}
//输出结果
{
 "data": {
 "account": {
 "name": "张三",
 "sex": "man"
```

```
    "sex": "man",
    "age": 18,
    "salary": 10000
    }
    }
}
```

GraphQL clients

在客户端中请求 `GraphQL` 接口查询数据。拿上一节定义好的接口 `account` 来说。

在客户端中，首先需要定义查询语句和查询的参数：

```
const query = `
query Account($username: String, $city: String) {
  account(username: $username) {
  name
  age
  sex
  salary(city: $city)
  }
}`
//定义好查询的参数
const variables = {username: '李大四', city: '深圳'}
```

之后我们使用fetch来请求数据，其中body需要设置两个参数，一个就是 `query` ，他的值就是上面定义好的查询语句，还有一个是参数 `variables` ，他的值是我们上面定义好的 `variables`

```
fetch('/graphql', {
  method: "POST",
  headers: {
  'Content-Type': 'application/json',
  'Accept': 'application/json'
  },
  body: JSON.stringify({
  query: query,
  variables: variables
  })
}).then(res => res.json())
.then(data => {
  console.log(data);
})
```

在客户端中使用

首先我们在上一节创建的baseType.js文件中添加一些代码，讲public文件夹设置成静态资源供我们访问：

```
app.use('/graphql', grapqlHTTP({
 schema: schema,
 rootValue: root,
 graphiql: true
}))
//添加如下代码
// 公开文件夹，供用户访问静态资源
app.use(express.static('public'))

app.listen(3000);
```

然后在 `GraphQL` 的目录下创建 `puclic` 文件夹，在 `puclic` 文件夹下面创建 `index.html` 文件，并输入如下代码

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <meta http-equiv="X-UA-Compatible" content="ie=edge">
 <title>Document</title>
</head>
<body>
 <button onclick="getData()">获取数据</button>
</body>
<script>
 function getData() {
 const query = `
 query Account($username: String, $city: String) {
 account(username: $username) {
 name
 age
 sex
 salary(city: $city)
 }
 }
 `

 const variables = {username: '李大四', city: '深圳'}
```

```
26.
27.   fetch('/graphql', {
28.   method: "POST",
29.   headers: {
30.   'Content-Type': 'application/json',
31.   'Accept': 'application/json'
32.   },
33.   body: JSON.stringify({
34.   query: query,
35.   variables: variables
36.   })
37.   }).then(res => res.json())
38.   .then(data => {
39.   console.log(data);
40.   })
41.   }
42.  </script>
43.  </html>
```

在这个 `html` 文件中，我们定义了一个按钮，按钮的点击事件是 `getData` ,点击之后向本地服务器请求数据，并在控制台打印。

在浏览器输入 `http://localhost:3000/index.html` ，点击 `获取数据` ,然后在控制台就可以看到返回的数据了。

使用Mutation修改数据

如果有一个更改数据的API接口，例如将数据插入数据库或更改数据库中已有的数据，则应将此接口设为一个 `Mutation` 而不是 `Query` 。定义 `mutation` ：

```
1.   type Mutation {setMessage(message: String): String }
```

具有映射到数据库创建或更新操作的突变通常很方便，例如 `setMessage` ，返回与服务器存储的相同的东西。这样，如果您修改服务器上的数据，则客户端可以了解这些修改。根解析器可以处理变异和查询，因此实现此架构的根可以简单地是：

```
1.  var fakeDatabase = {};
2.  var root = {
3.   setMessage: ({
4.   message
5.   }) => {
6.   fakeDatabase.message = message;
7.   return message;
```

```
  },
  getMessage: () => {
  return fakeDatabase.message;
  }
};
```

在许多情况下，你会发现许多不同的 `mutation` 都接受相同的输入参数。一个常见的示例是，在数据库中创建对象和更新数据库中的对象通常采用相同的参数。为了简化架构，可以使用“输入类型”，方法是使用 `input` 关键字而不是 `type` 关键字。

```
input MessageInput {
 content: String author: String
}
type Message {
 id: ID!content: String author: String
}
type Query {
 getMessage(id: ID!): Message
}
type Mutation {
 createMessage(input: MessageInput): Message updateMessage(id: ID!, input: MessageInput): Message
}
```

在项目目录下创建 `mutation.js`

```
const express = require('express');
const {
 buildSchema
} = require('graphql');
const grapqlHTTP = require('express-graphql');
// 定义schema，查询和类型，mutation
const schema = buildSchema(`
 input AccountInput {
 name: String
 age: Int
 sex: String
 department: String
  }
 type Account {
 name: String
 age: Int
 sex: String
 department: String
```

```
  department: String
  }
  type Mutation {
  createAccount(input: AccountInput): Account
  updateAccount(id: ID!, input: AccountInput): Account
  }
  type Query {
  accounts: [Account]
  }
  `
)

const fakeDb = {};

// 定义查询对应的处理器
const root = {
  accounts() {
  var arr = [];
  for (const key in fakeDb) {
  arr.push(fakeDb[key])
  }
  return arr;
  },
  createAccount({
  input
  }) {
  // 相当于数据库的保存
  fakeDb[input.name] = input;
  // 返回保存结果
  return fakeDb[input.name];
  },

  updateAccount({
  id,
  input
  }) {
  // 相当于数据库的更新
  const updatedAccount = Object.assign({}, fakeDb[id], input);
  fakeDb[id] = updatedAccount;
  // 返回保存结果
  return updatedAccount;
```

```
  }
}

const app = express();

app.use('/graphql', grapqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true
}))

app.listen(3000);
```

重新启动项目 `node mutation.js` 。在调试界面，可以调用 `createAccount` 插入数据，使用 `updateAccount` 修改数据。

```
//插入两条数据
mutation {
  createAccount(input:{
  name: "zhangsan",
  age: 16,
  sex: "nan",
  department:"工信部"
  }) {
  name
  age
  sex
  department
  }
}

mutation {
  createAccount(input:{
  name: "lisi",
  age: 25,
  sex: "nan",
  department:"工信部"
  }) {
  name
  age
  sex
  department
```

```
  }
}

//查询
query{
 accounts {
 name
 age
 sex
 department
 }
}
//结果
{
 "data": {
 "accounts": [
 {
 "name": "zhangsan",
 "age": 16,
 "sex": "nan",
 "department": "工信部"
 },
 {
 "name": "lisi",
 "age": 25,
 "sex": "nan",
 "department": "工信部"
 }
 ]
 }
}
```

身份认证和中间件

要将中间件与GraphQL解析器一起使用，只需像使用普通Express应用程序一样使用中间件即可。

```
const app = express();

const middleware = (req, res, next) => {
 if(req.url.indexOf('/graphql') !== -1 && req.headers.cookie.indexOf('auth') === -1) {
 res.send(JSON.stringify({
        "你没有权限访问这个接口"
```

```
    error: '您没有权限访问这个接口'
  }));
  return;
  }
  next();
 }

 app.use(middleware);
```

假设我们希望服务器记录每个请求的IP地址，并且还希望编写一个API，该API返回调用方的IP地址。我们可以使用中间件来实现前者，而可以通过 `request` 在解析器中访问对象来实现后者。这是实现此目的的服务器代码：

```
var express = require('express');
var graphqlHTTP = require('express-graphql');
var {buildSchema} = require('graphql');
var schema = buildSchema(` type Query { ip: String } `);
const loggingMiddleware = (req, res, next) => {
 console.log('ip:', req.ip);
 next();
}
var root = {
 ip: function(args, request) {
  return request.ip;
 }
};
var app = express();
app.use(loggingMiddleware);
app.use('/graphql', graphqlHTTP({
 schema: schema,
 rootValue: root,
 graphiql: true,
}));
app.listen(4000);
```

Constructing Types

使用 `buildSchema` 方式构造出来的查询类，当你的查询出错的时候，不能够精确的显示出错误的位置，而 `Constructing Types` 构造方式定义的查询能够解决这个问题，并且做到相同的查询效果，并且易于维护。

对于很多应用程序，可以在应用程序启动时定义固定的 `shcema` ，然后使用GraphQL架构语言进行定义。在某些情况下，以编程方式构造模式是很有用的。可以使用 `GraphQLSchema` 构造函数执行此操作。使

用 `GraphQLSchema` 构造函数创建模式时，而不是仅使用模式语言来定义 `Query` 和 `Mutation` 类型，而是将它们创建为单独的对象类型。

例如，假设我们正在构建一个简单的API，该API可让您根据ID为一些硬编码用户获取用户数据。使用 `buildSchema` 我们可以编写具有以下内容的服务器

buildSchema 方式定义的查询接口

```
// 定义schema，查询和类型
const schema = buildSchema(`
 type Account {
 name: String
 age: Int
 sex: String
 department: String
 }
 type Query {
 account(username: String): Account
 }
`)

// // 定义查询对应的处理器
const root = {
 account({ username }) {
 const name = username;
 const sex = 'man';
 const age = 18;
 const department = '开发部';
 return {
 name,
 sex,
 age,
 department
 }
 }
}
const app = express();
app.use('/graphql', grapqlHTTP({
 schema: schema,
 rootValue: root,
 graphiql: true
```

```
34. }))
```

使用Constructing Types 定义接口

Constructing Types 方式使用使用 `graphql` 提供的构造函数来定义类型，查询语句等。

```
1.  var AccountType = new graphql.GraphQLObjectType({
2.   name: 'Account',
3.   fields: {
4.   name: { type: graphql.GraphQLString },
5.   age: { type: graphql.GraphQLInt },
6.   sex: { type: graphql.GraphQLString },
7.   department: { type: graphql.GraphQLString }
8.   }
9.  });
10.
11. var queryType = new graphql.GraphQLObjectType({
12.  name: 'Query',
13.  fields: {
14.  account: {
15.  type: AccountType,
16.  // `args` describes the arguments that the `user` query accepts
17.  args: {
18.  username: { type: graphql.GraphQLString }
19.  },
20.  resolve: function (_, { username }) {
21.  const name = username;
22.  const sex = 'man';
23.  const age = 18;
24.  const department = '开发部';
25.  return {
26.  name,
27.  sex,
28.  age,
29.  department
30.  }
31.  }
32.  }
33.  }
34. });
35. var schema = new graphql.GraphQLSchema({ query: queryType });
36. const app = express();
37. app.use('/graphql', graphlHTTP({
```

```
app.use('/graphql', grapqlHTTP({
  schema: schema,
  graphiql: true
}))
```

`graphql/type` 模块负责定义GraphQL类型和架构，可以从 `graphql/type` 模块导入，也可以从根 `graphql` 模块导入。

```
// ES6
import{ GraphQLSchema }from'graphql';
// CommonJS
var{ GraphQLSchema }= require('graphql');
```

操作数据库

以下实例使用 `mysql` 作为例子：

安装 `mysql` 模块

```
npm install mysql -S
```

连接数据库

```
var pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'dashen'
});
```

查询对应的处理器中插入mysql操作即可

```
const express = require('express');
const {buildSchema} = require('graphql');
const grapqlHTTP = require('express-graphql');
const mysql = require('mysql');
// https://www.npmjs.com/package/mysql
var pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'dashen'
});

```

```
14.
15. // 定义schema，查询和类型，mutation
16. const schema = buildSchema(`
17.  input AccountInput {
18.  name: String
19.  age: Int
20.  sex: String
21.  department: String
22.  }
23.  type Account {
24.  name: String
25.  age: Int
26.  sex: String
27.  department: String
28.  }
29.  type Mutation {
30.  createAccount(input: AccountInput): Account
31.  deleteAccount(id: ID!): Boolean
32.  updateAccount(id: ID!, input: AccountInput): Account
33.  }
34.  type Query {
35.  accounts: [Account]
36.  }
37. `);
38.
39. // 定义查询对应的处理器
40. const root = {
41.  accounts() {
42.  return new Promise((resolve, reject)=>{
43.  pool.query('select name, age, sex, department from account', (err, results)=> {
44.  if(err) {
45.  console.log('出错了' + err.message);
46.  return;
47.  }
48.  const arr = [];
49.  for(let i=0;i<results.length;i++) {
50.  arr.push({
51.  name: results[i].name,
52.  sex: results[i].sex,
53.  age: results[i].age,
54.  department: results[i].department
```

```
54.   department: results[i].department,
55.   })
56.   }
57.   resolve(arr);
58.   })
59.   })
60.   },
61.   createAccount({ input }) {
62.   const data = {
63.   name: input.name,
64.   sex: input.sex,
65.   age: input.age,
66.   department: input.department
67.   }
68.   return new Promise((resolve, reject)=>{
69.   pool.query('insert into account set ?', data, (err) => {
70.   if (err) {
71.   console.log('出错了' + err.message);
72.   return;
73.   }
74.   // 返回保存结果
75.   resolve(data);
76.   })
77.   })
78.   },
79.
80.   updateAccount({ id, input }) {
81.   const data = input
82.   return new Promise((resolve, reject) => {
83.   pool.query('update account set ? where name = ?', [data, id], (err) => {
84.   if (err) {
85.   console.log('出错了' + err.message);
86.   return;
87.   }
88.   // 返回保存结果
89.   resolve(data);
90.   })
91.   })
92.   },
93.   deleteAccount({id}) {
94.   return new Promise((resolve, reject)=>{
```

```
  pool.query('delete from account where name = ?', [id], (err)=>{
  if(err) {
  console.log('出错了' + err.message);
  reject(false);
  return;
  }
  resolve(true);
  })
  })
  }
 }

 const app = express();

 app.use('/graphql', grapqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true
 }))

 app.listen(3000);
```

GraphQL案例

引子：https://www.cnblogs.com/fuhuixiang/p/7479276.html

一、搭建环境

1、安装

```
yarn add express @babel/cli @babel/core @babel/node @babel/preset-env
```

2、搭建express环境

```
scripts: {
 "dev": "nodemon index.js --exec @babel/node --presets @babel/preset-env",
}
```

3、创建Express服务

4、用json-server 搭建服务

```
scripts: {
 "json-server": "json-server --watch db.json --port 3300"
}
```

```
}
```

5、用Express 创建 GraphQL HTTP 服务

```
yarn add graphql express-graphql
```

在index.js里导入： `import graphqlHTTP from 'express-graphql'`

创建schema.js, 导入： `import schema from './schema'`

```
app.use('/graphql', graphqlHTTP({
 schema,
 graphiql: true
}))
```

6、定义GraphQL 的 Schema: 查询

查看graphql 文档

7、查询操作(Query): 定义内容列表查询

8、定义单个内容的查询：查询参数的使用

9、查询评论内容

10、修改操作（mutation）：创建

11、更改

13、删除

index.js

```
import express from 'express'
import graphqlHTTP from 'express-graphql'
import schema from './schema'
import {
 graphql
} from 'graphql'
const app = express()

app.use('/graphql', graphqlHTTP({
 schema,
 graphiql: true
}))

app.get('/abc', (req, res) => {
 var query = `{
```

```
  subject(id: "2") {
  id,
  title,
  genres,
  rating,
  theater {
  id,
  name
  }
  }
  }`
  graphql(schema, query).then(result => {
  res.send(result)
  });
})

app.listen(8080)
```

shema.js

```

import {
  graphql,
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLList,
  GraphQLFloat,
  GraphQLNonNull,
  GraphQLInt
} from 'graphql';

import axios from 'axios'

const theatersType = new GraphQLObjectType({
  name: 'Theaters',
  fields: {
  id: {
  type: GraphQLString
  },
  name: {
  type: GraphQLString
  }
```

```
  }
  }
 })

 const commentsType = new GraphQLObjectType({
  name: 'Comments',
  fields: {
  id: {
  type: GraphQLString
  },
  theater: {
  type: theatersType,
  resolve(obj) {
  return axios.get(`${ API_BASE }/theaters/${ obj.theater }`)
  }
  },
  content: {
  type: GraphQLString
  }
  }
 })

 const subjectsType = new GraphQLObjectType({
  name: 'Subjects',
  fields: {
  id: {
  type: GraphQLString
  },
  title: {
  type: GraphQLString
  },
  genres: {
  type: GraphQLString
  },
  rating: {
  type: GraphQLFloat
  },
  theater: {
  type: theatersType,
  resolve(obj) {
  return axios.get(`${ API_BASE }/theaters/${ obj.theater }`)
```

```
  .then(response => response.data)
  }
  },
  comments: {
  type: new GraphQLList(commentsType),
  resolve(obj) {
  return axios.get(`${ API_BASE }/subjects/${ obj.id }/comments`)
  .then(response => response.data)
  }
  }
  }
 })

 const MutationRootType = new GraphQLObjectType({
  name: 'MutationRoot',
  fields: {
  create: {
  type: subjectsType,
  args: {
  title: {
  type: new GraphQLNonNull(GraphQLString)
  }
  },
  resolve(obj, args) {
  return axios.post(`${API_BASE}/subjects`, {
  ...args
  })
  .then(response => response)
  }
  },
  update: {
  type: subjectsType,
  args: {
  id: {
  type: new GraphQLNonNull(GraphQLString)
  },
  title: {
  type: GraphQLString
  }
  },
  resolve(obj, args) {
```

```
105.   return axios.patch(`${API_BASE}/subjects/${ args.id }`, {
106.   ...args
107.   })
108.   .then(response => response)
109.   }
110.   },
111.   delete: {
112.   type: subjectsType,
113.   args: {
114.   id: {
115.   type: new GraphQLNonNull(GraphQLString)
116.   }
117.   },
118.   resolve(obj, args) {
119.   return axios.delete(`${API_BASE}/subjects/${ args.id }`)
120.   .then(response => response)
121.   }
122.   }
123.   }
124. })
125.
126. const API_BASE = "http://localhost:3300"
127.
128. const schema = new GraphQLSchema({
129.   query: new GraphQLObjectType({
130.   name: 'RootQueryType',
131.   fields: {
132.   hello: {
133.   type: GraphQLString,
134.   resolve() {
135.   return 'world';
136.   }
137.   },
138.   subjects: {
139.   type: new GraphQLList(subjectsType),
140.   resolve() {
141.   return axios.get(`${ API_BASE }/subjects`)
142.   .then(response => response.data)
143.   }
144.   },
145.   subject: {
```

```
    type: subjectsType,
    args: {
    id: {
    type: GraphQLString
    }
    },
    resolve(obj, args, context) {
    return axios.get(`${ API_BASE }/subjects/${ args.id }`)
    .then(response => response.data)
    }
    },
    }
    }),
    mutation: MutationRootType
});
export default schema
```

db. json

```
{
 "subjects": [
  {
 "id": 1,
 "title": "欧洲攻略",
 "genres": "喜剧，动作，爱情",
 "rating": 3.8,
 "theater": 1
  },
  {
 "id": 2,
 "title": "精灵旅社3：疯狂假期",
 "genres": "喜剧，动画，奇幻",
 "rating": 7.2,
 "theater": 2
  }
  ],
 "theaters": [
  {
 "id": 1,
 "name": "CGV影城(清河店)"
  },
  {
```

```
  "id": 2,
  "name": "橙天嘉禾影城(上地店)"
  }
  ],
  "comments": [
  {
  "id": 1,
  "subject": 1,
  "theaters": 1,
  "content": "电影不错，影院环境太糟了"
  },
  {
  "id": 2,
  "subject": 2,
  "theaters": 2,
  "content": "满分好评"
  }
  ]
}
```

graphIQL 工具命令例子：

```
query getGreeting {
 greeting
}

query getPost {
 post(id: "1") {
 id,
 title,
 content,
 author {
 name
 }
 }
}

query getPost {
 posts {
 id,
 author {
 name
```

```
  },
  title,
  comments {
  author {
  name
  },
  cotent
  }
  }
}

mutation createPost {
  createPost(
  title: "hello ~",
  content: "大家好",
  author: "1"
  ) {
  id,
  title,
  content,
  author {
  name
  }
  }
}

mutation updatePost {
  updatePost(
  id: "3",
  title: "您好"
  ) {
  id,
  title,
  author {
  name
  }
  }
}

mutation deletePost {
  deletePost(id: "3") {
```

```
61.   deletePost(id: 3) {
62.     id
63.   }
64. }
```