# **Graphql的世界**

## **graphql概述**

### graphql是什么？

GraphQL是Facebook2015年开源的数据查询规范。

现今的绝大多数Web Service都是RESTful的，

也就是说，client和server的主要沟通模式还是靠client根据自己的需要向server的若干个endpoint (url)发起请求。

由于功能的日渐丰富，对Web Application的要求变得复杂，REST的一些问题逐渐暴露，人们开始思考如何应对这些问题。

GraphQL便是具有代表性的一种。GraphQL这个名字，Graph + Query Language，就表明了它的设计初衷是想要用类似图的方式表示数据：即不像在REST中，数据被各个API 所分割，而是有关联和层次结构的被组织在一起。

**直接的说：**

GraphQL是**API查询语言**，类似于数据库中的SQL。

不同的是sql的查询的是数据库，而GraphQL查询的是数据源，这个数据源可以是HTTP接口、数据库查询集合、静态json文件、另外一个api的数据源，特别的灵活。

### **graphql官网**

<http://graphql.cn/code/>

<http://graphql.cn/graphql-js/> graphql之js使用

### graphql作用？

GraphQL作为RESTful的一种辅助工具，为解决restfulAPI的一些问题而诞生。

### **RESTful API缺点**

1. 前端和后端对于接口的****控制权****是交叉冲突的，往往一方改动不算，前端改动一个字段，连带着后端也需要改动，反之亦是。
2. 前端对于真正用到的字段是没有直观映像的，仅仅通过url地址，无法预测也无法回忆返回的字段数目和字段是否有效，接口返回50个字段，但却只用5个字段，造成****字段冗余，扩展性差，单个RESTful接口返回数据越来越臃肿****。
3. ****API聚合问题****，某个前端展现，实际****需要调用多个独立的RESTful API才能获取到足够的数据。****
4. ****前后端字段频繁改动，导致类型不一致****，错误的数据类型可能会导致网站出错
5. 尤其是在业务多变的场景中，很难在保证工程质量的同时快速满足业务需求

### RESTful **API缺点案例**

#### 案例1

比如获取用户信息/users/:id，最初可能只有id、昵称，但随着需求的变化，用户所包含的字段可能会越来越多，年龄、性别、头像、经验、等级，等等等等。而具体到某个前端页面，可能只需要其中一小部分数据，这样就会****增加网络传输量****，前端获取了大量不必要的数据。

#### **案例2**

比如一个文章详情页，最初可能只需要文章内容，那么前端就调用/articles/:aid获取到文章内容来展现就行了

但随着需求的演进，产品可能会希望加上作者信息（昵称、头像等），这时前端又需要在获取文章详情后，根据其中的作者id字段继续获取作者相关的信息，/user/:uid

然后，需求又变化了，产品希望在加上这篇文章的评论，这时前端需要继续调用/comment/:aid来拉取评论列表

#### **案例3**

在App上遇到的问题

对于Web前端而言，由于ajax技术的存在，这种的请求数据方式，也就开发上稍微麻烦些，并不会造成太大的问题；

但对于App来说，渲染的方式不同，必须要拉取的全部的数据之后，才能绘制界面，就会导致这个界面必须要等到所有RESTful接口的返回数据都拿到，才能进行绘制。

## graphql**基本**示例

### 安装express-graphql模块

cnpm i express express-graphql graphql -S

### **服务端的简单**代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### **运行后的基本演示**



### **前端通过fetch访问**

fetch('http://localhost:4000/graphql/', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({ query: "{ hello }" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### 返回结果



## **graphql**基本类型

### 基本说明

大多数情况下，你所需要做的只是使用 GraphQL schema language 指定你的 API 需要的类型，然后作为参数传给 buildSchema 函数。

GraphQL schema language 支持的标量类型有 String、Int、Float、Boolean 和 ID，因此你可以在传给 buildSchema 的 schema 中直接使用这些类型。

默认情况下，每个类型都是可以为空的 —— 意味着所有的标量类型都可以返回 null。使用感叹号可以标记一个类型不可为空，如 String! 表示非空字符串。

如果是列表类型，使用方括号将对应类型包起来，如 [Int] 就表示一个整数列表。

这些类型都直接映射 JavaScript，所以你可以直接返回原本包含这些类型的原生 JavaScript 对象。

### **示例**

下面是一个展示如何使用这些基本类型的示例：

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL schema language 构建一个 schema

var schema = buildSchema(`

type Query {

w18071: String

w18072: Float!

w18073: [Int]

}

`);

// root 将会提供每个 API 入口端点的解析函数

var root = {

w18071: () => {

return Math.random() < 0.5 ? '天气不错' : '天气一般般了';

},

w18072: () => {

return Math.random();

},

w18073: () => {

return [1, 2, 3];

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### **前端通过fetch访问**

fetch('http://localhost:4000/graphql/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

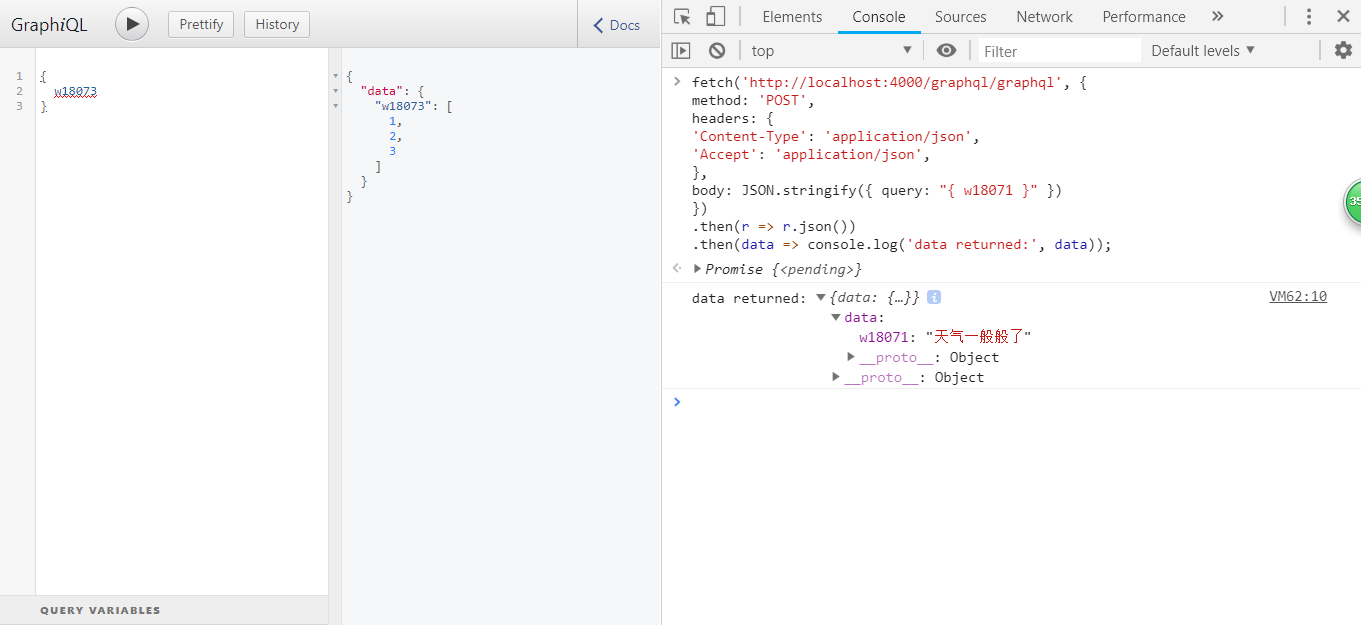
body: JSON.stringify({ query: "{ w18071}" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### **运行结果**



## graphql返回对象

如果要返回一个对象，那么我们先要定义一个我们要返回的对象类型才行。

### 后端代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type User{

name: String!

sex: String

age: Int

hobby: [String]

}

type Query {

users:[User]

}

`);

// root 规定了顶层的 API 入口端点

var root = {

users: function () {

var users = [

{

name: 'zs',

sex: 'man',

age: 28,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

{

name: 'ls',

sex: 'woman',

age: 22,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

];

return users;

},

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

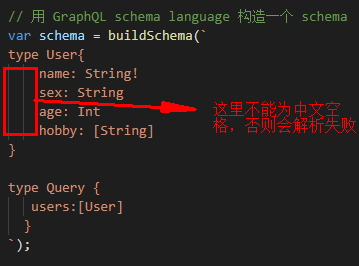
graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

注意：schema里面的空格一定不能是中文空格，否则解析时会报错的



### 前端代码

fetch('http://localhost:4000/graphql/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

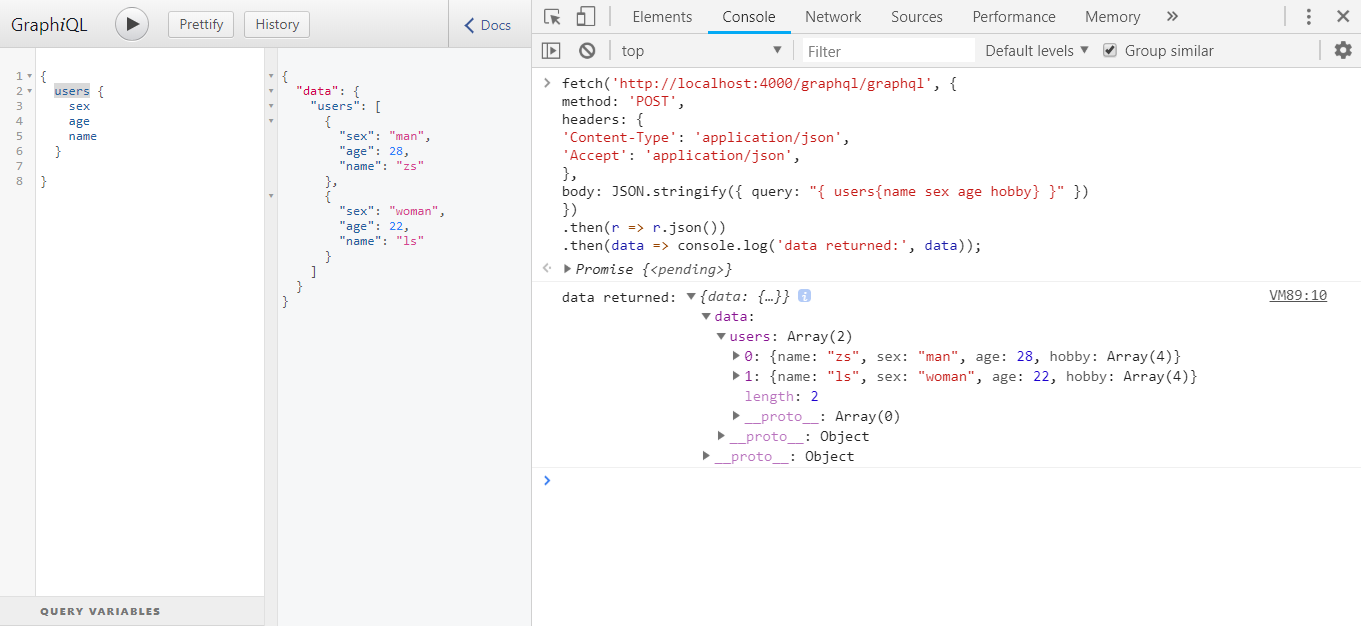
body: JSON.stringify({ query: "{ users{name sex age hobby} }" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### **运行结果**



在获取数据时，你想获取那些数据，就可以只传入那些相关的字段。

### 返回class实例

每次返回时都去单独的定义一个对象容易出错，这里我们直接返回一个class的实例，实现一样的效果。

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type User{

name: String!

sex: String

age: Int

hobby: [String]

}

type Query {

users:[User]

}

`);

class User {

constructor(name, sex, age, hobby) {

this.name = name;

this.sex = sex;

this.age = age;

this.hobby = hobby;

}

}

// root 规定了顶层的 API 入口端点

var root = {

users: function () {

return [new User("zs", "男", 22, ["小说", "电影"])];

},

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

**效果**



## **graphql**传递参数

就像 REST API 一样，可以在 GraphQL API 中，在前端发起请求时传入参数，在后端schema language 中定义参数，并自动进行类型检查。每一个参数必须有名字和数据类型。

### 后端示例

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type Query {

w18074(p1: Int!, p2: Int): [Int]

}

`);

// root 为每个端点入口 API 提供一个解析器

var root = {

w18074: function ({ p1, p2 }) { //这里通过es6的解构赋值实现参数的获取

var output = [p1\*10,p2\*100];

return output;

}

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 前端fetch请求

fetch('http://localhost:4000/graphql/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

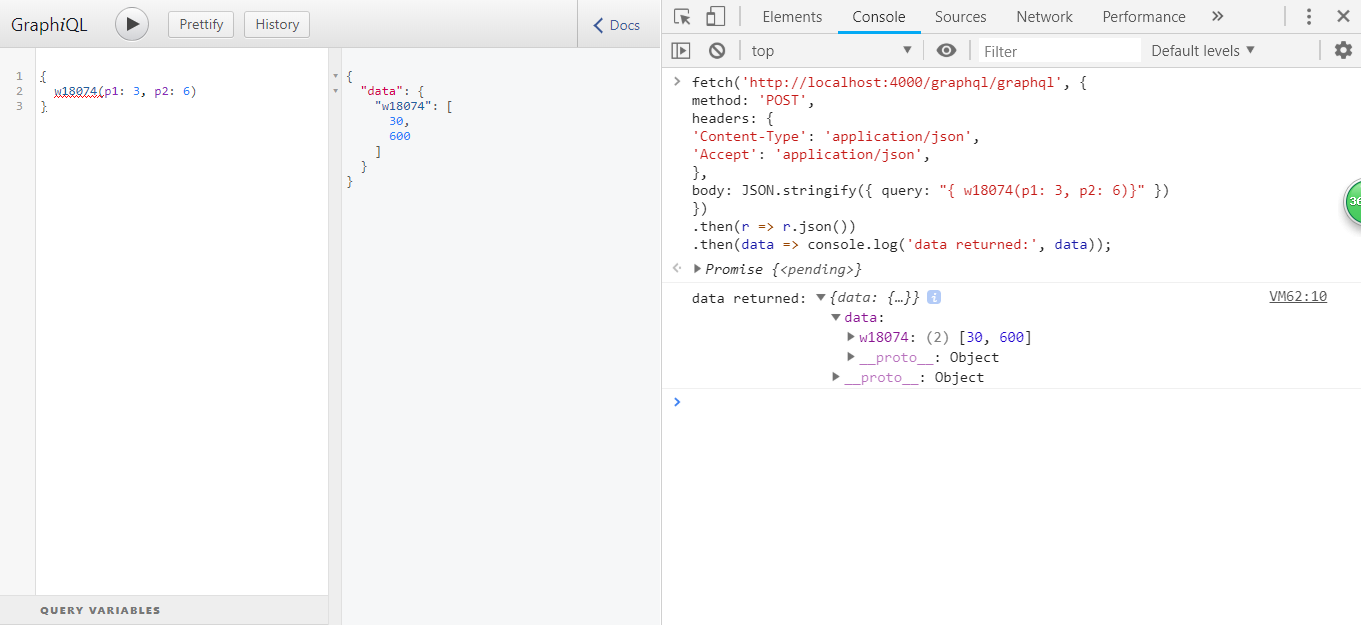
body: JSON.stringify({ query: "{ w18074(p1: 3, p2: 6)}" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### 请求结果



### 修改前端代码

使用 $p1和 $p2作为 GraphQL 中的变量，我们无需在客户端对它们进行转义。

通过基础类型和参数传递，你可以定义任意你"能够"在 REST API 中定义的内容。

var p1 = 3;

var p2 = 6;

var query = `query Test($p1: Int!, $p2: Int) {

w18074(p1: $p1, p2: $p2)

}`;

fetch('/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query,

variables: { p1, p2 },

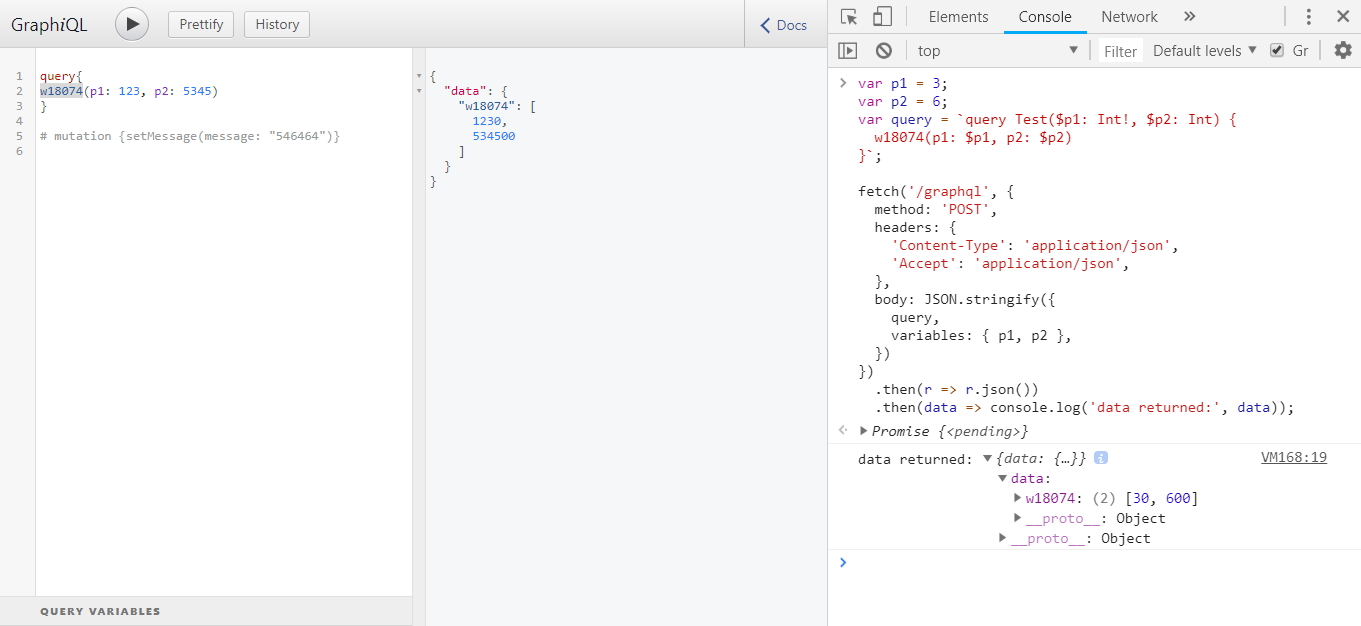
})

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

**效果**



注意：test只是我随便起得个名字，你可以修改，甚至不写名字都可以，如下

var query = `query($p1: Int!, $p2: Int) {

w18074(p1: $p1, p2: $p2)

}`;

## graphql**修改**数据

在GraphQL 中，对于修改数据的入口端点不再是Query而是 Mutation。

### 后端示例代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type Mutation {

setMessage(message: String): String

}

type Query {

getMessage: String

}

`);

var fakeDatabase = {};

var root = {

setMessage: function ({message}) {

fakeDatabase.message = message;

return message;

},

getMessage: function () {

return fakeDatabase.message;

}

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 前端代码一

var query=`

mutation{

setMessage(message:"234567")

}

`

fetch('/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query:query

}

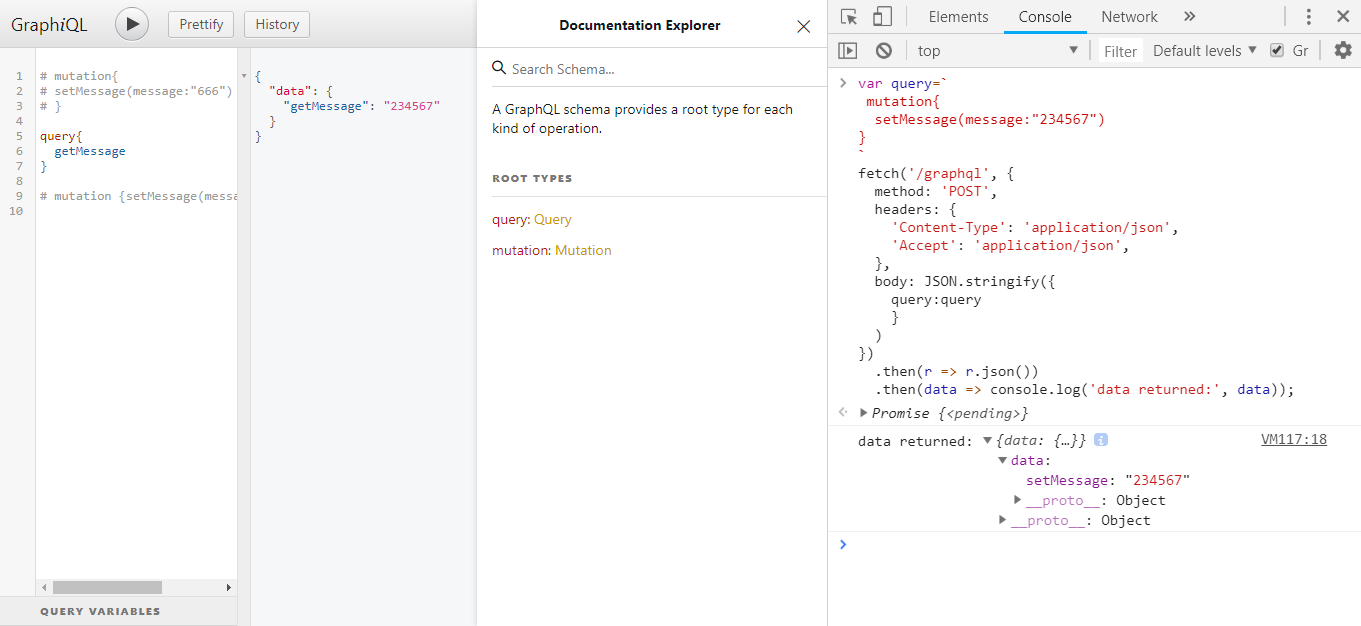
)

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### **效果一**



### **前端代码二**

var message = '测试信息';

var query = `mutation SetMessage($message: String) {

setMessage(message: $message)

}`;

fetch('/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query,

variables: {

message

}

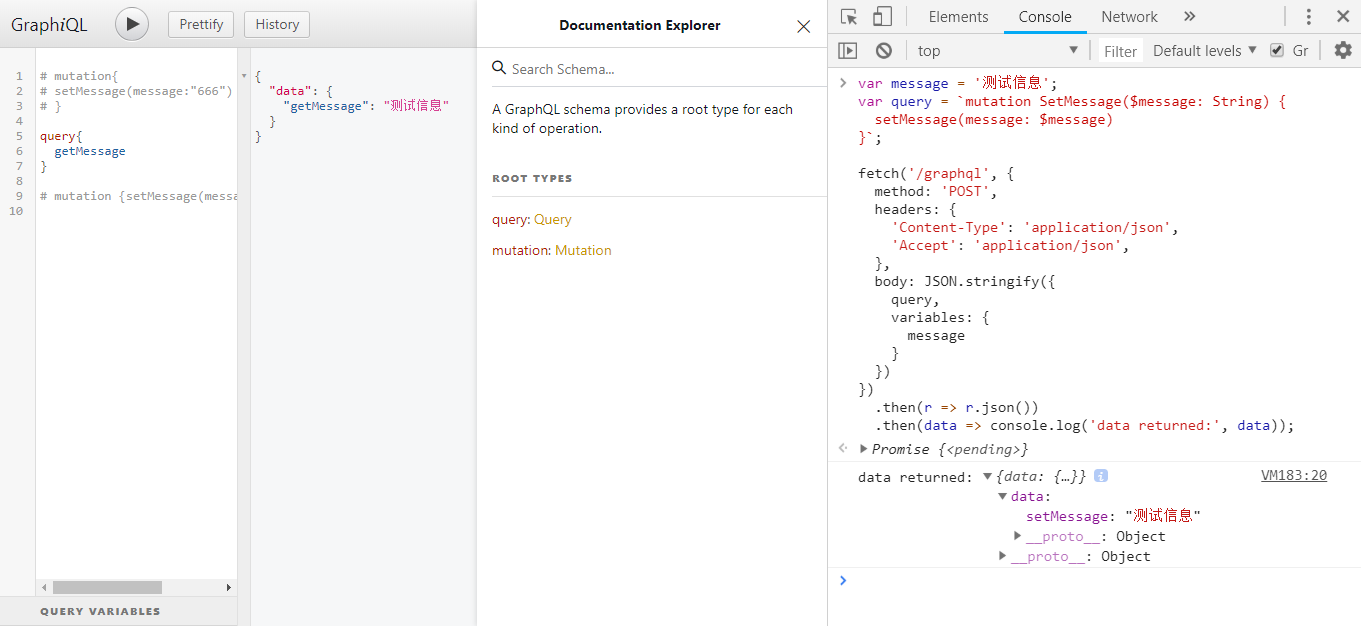
})

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### 效果**二**



## graphql**添加**数据

添加数据也是使用mutation类型，如下

### 后端代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type User{

name: String!

sex: String

}

type Query {

getusers:[User]

}

type Mutation{

adduser(name:String,sex:String):[User]

}

`);

var users = [

{

name: 'zs',

sex: 'man',

},

{

name: 'ls',

sex: 'woman',

},

];

// root 规定了顶层的 API 入口端点

var root = {

getusers: function () {

return users;

},

adduser:function({ name, sex }){

users.push({name,sex})

return users;

}

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### **前端代码**

var name = 'zl';

var sex = '女';

var query = `mutation Adduser($name: String,$sex: String,) {

adduser(name:$name,sex:$sex){

name sex

}

}`;

fetch('/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query,

variables: {

name,

sex

}

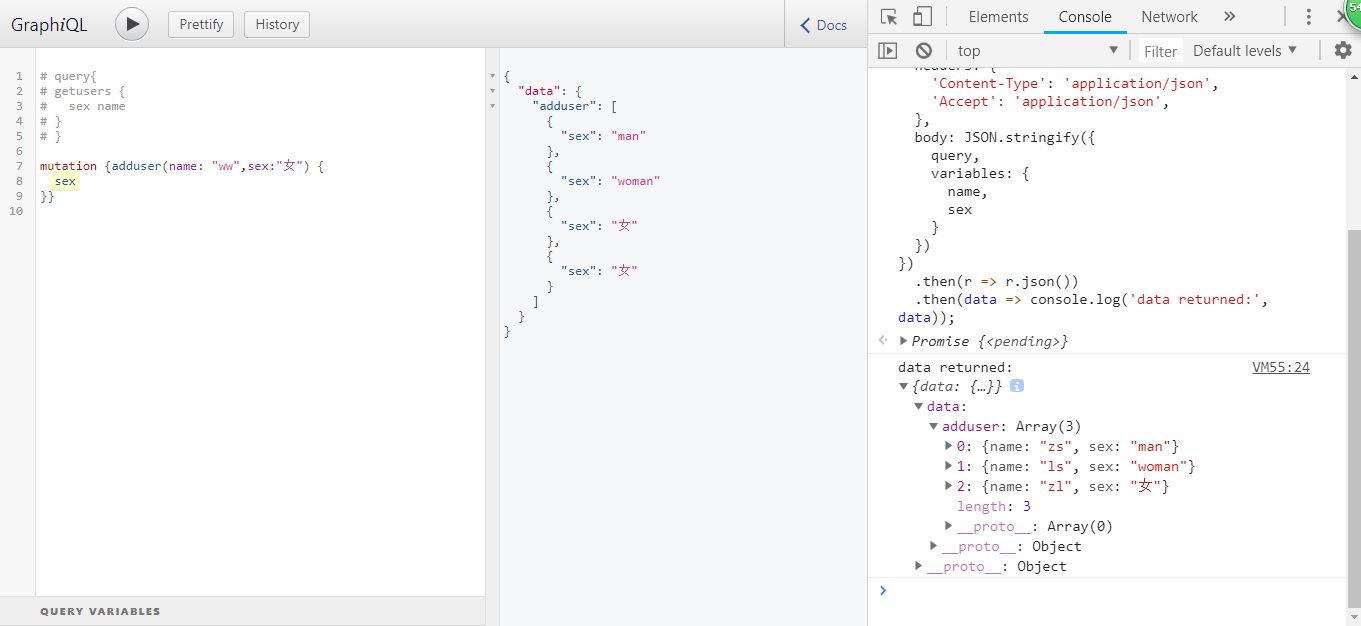
})

})

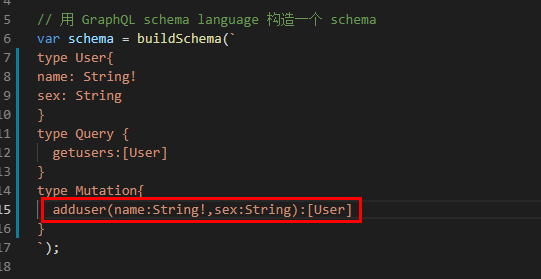
.then(r => r.json())

.then(data => console.log('data returned:', data));

### **运行效果**



如果后端设置了参数为必填，那么传参时也需要写必填，类似于如下操作





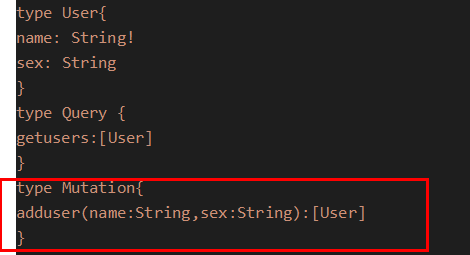
## **graphql**定义输入类型【为了让前端直接传一个对象参数过来】

### 概述

在有些时候前端会执行大批量的数据添加，比如对于一些ERP等系统，企业员工完善自己的信息时可能会填写许多的数据（姓名，年龄，性别，入职时间，地址，工作岗位...）

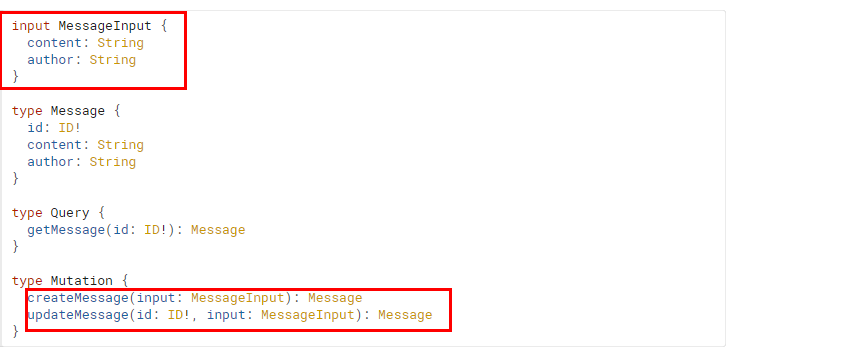
这个时候，如果我们直接批量的往后端发，然后后端就需要在mutation中的相关方法中写许多的参数，这是非常不科学的。

如下：



这时候我们就可以使用输入类型了，所谓的输入类型就是定义一个类型，然后把这一系列的数据当中其类型字段进行定义。（类似与es6的class）

定义了输入类型后，我们在后端就可以直接使用输入类型来代替所有参数，如下：



需要注意的是：

输入类型的字段不能是其他对象类型，只能是基础标量类型、列表类型或者其他输入类型。

### 后端代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL schema language 构建 schema

var schema = buildSchema(`

input MessageInput {

id:Int!

name: String

sex: String

}

type Message {

id:Int!

name: String

sex: String

}

type Query {

getMessage: [Message]

}

type Mutation {

createMessage(inputobj: MessageInput):String

}

`);

// 映射 username 到 content

var student = [{ id: 1, name: "zs", sex: "男" }];

var root = {

getMessage: function () {

return student;

},

createMessage: function ({ inputobj }) {

student.push(inputobj);

return "创建成功";

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000, () => {

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

});

### **前端**代码

var obj={

id:4,

name:"zl",

sex:"女"

}

var query = `mutation CreateMessage($inputobj:MessageInput) {

createMessage(inputobj:$inputobj)

}`;

fetch('/graphql', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query,

variables: obj

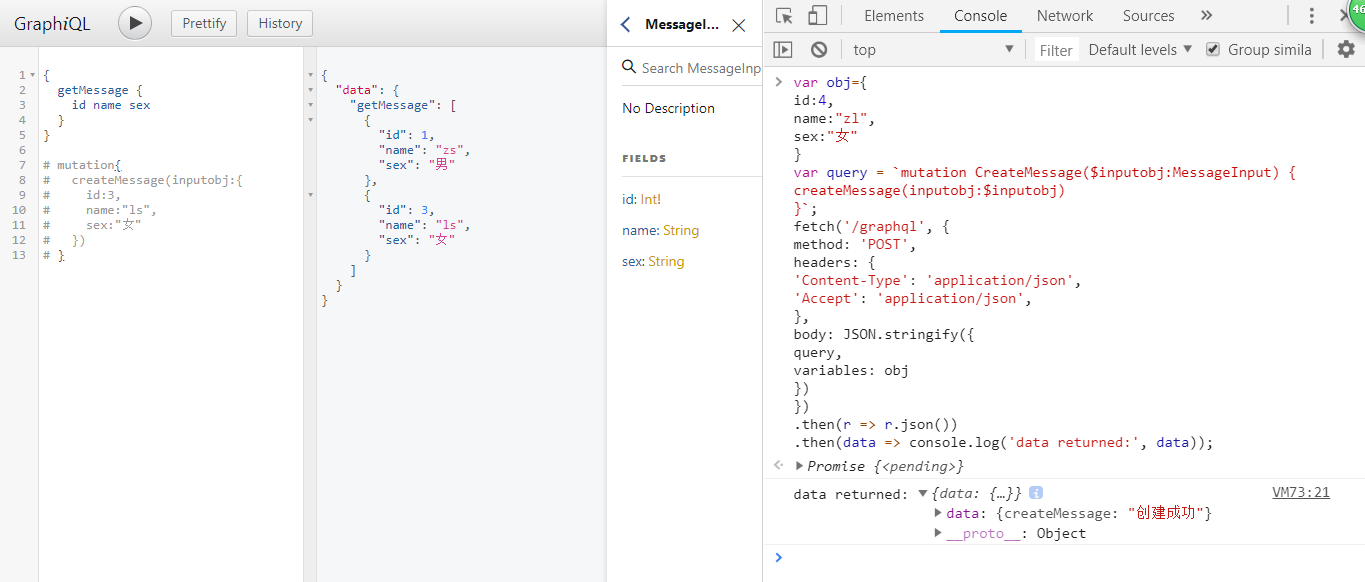
})

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### **效果截图**



## **获取schema细节**

GraphQL是可自省的，也就是说你可以通过查询一个GraphQL知道它自己的schema细节。

### 查询所有类型细节

查询\_\_schema以列出所有该schema中定义的类型，并获取每一个的细节：

**query** {

**\_\_schema** {

**types** {

**name**

kind

description

fields {

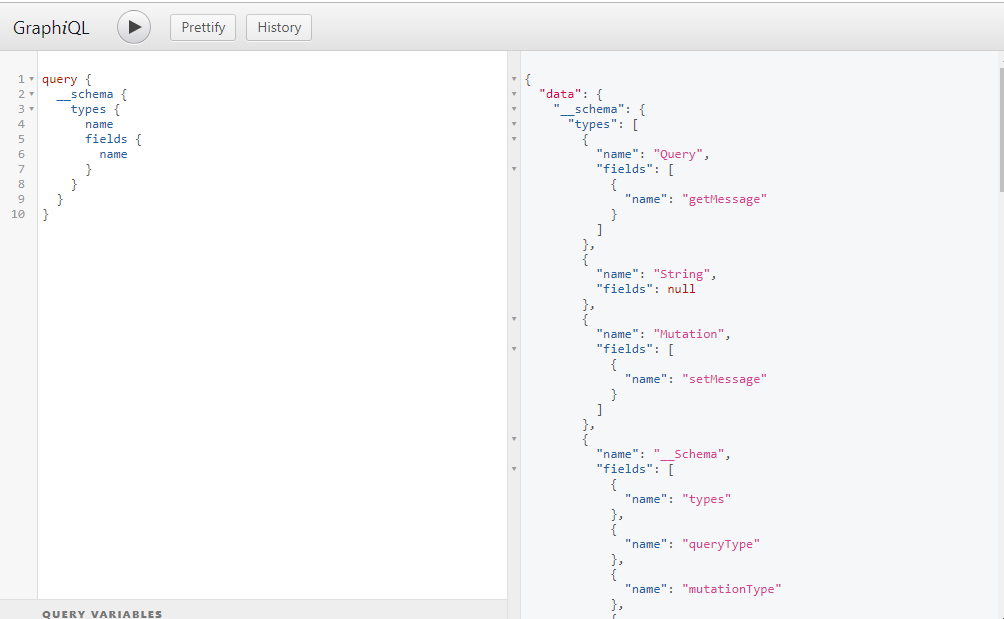
**name**

}

}

}

}



### 查询任意类型细节

查询\_\_type以获取任意类型的细节：

query {

\_\_type(name: "Query") {

name

kind

description

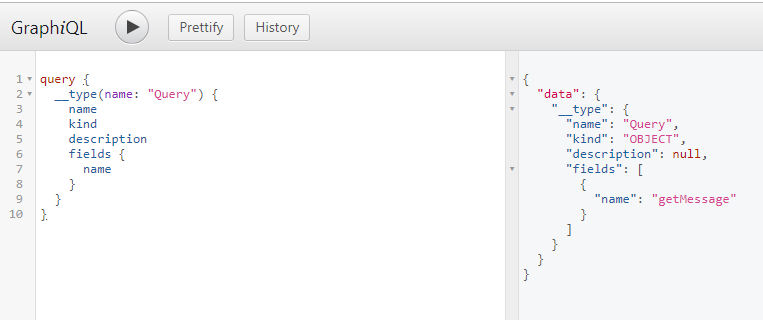
fields {

name

}

}

}



### 提示

提示：自省查询可能是你在GraphQL中唯一的GET请求。不管是query还是mutation，如果你要传递请求体，GraphQL请求方式都应该是POST

## **组合查询**

我们可以类似于REST那样发起多个同时发起多个请求

### 后端代码（定义多个请求）

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type Mutation {

setMessage(message: String): String

}

type Query {

getMessage: String

getName: String

getList: [Int]

}

`);

var fakeDatabase = {message:"nihao"};

var root = {

setMessage: function ({ message }) {

fakeDatabase.message = message;

return message;

},

getMessage: function () {

return fakeDatabase.message;

},

getName: function () {

return "张三"

},

getList:function(){

return [1,2,3,4]

}

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 前端代码（组合查询多个接口）

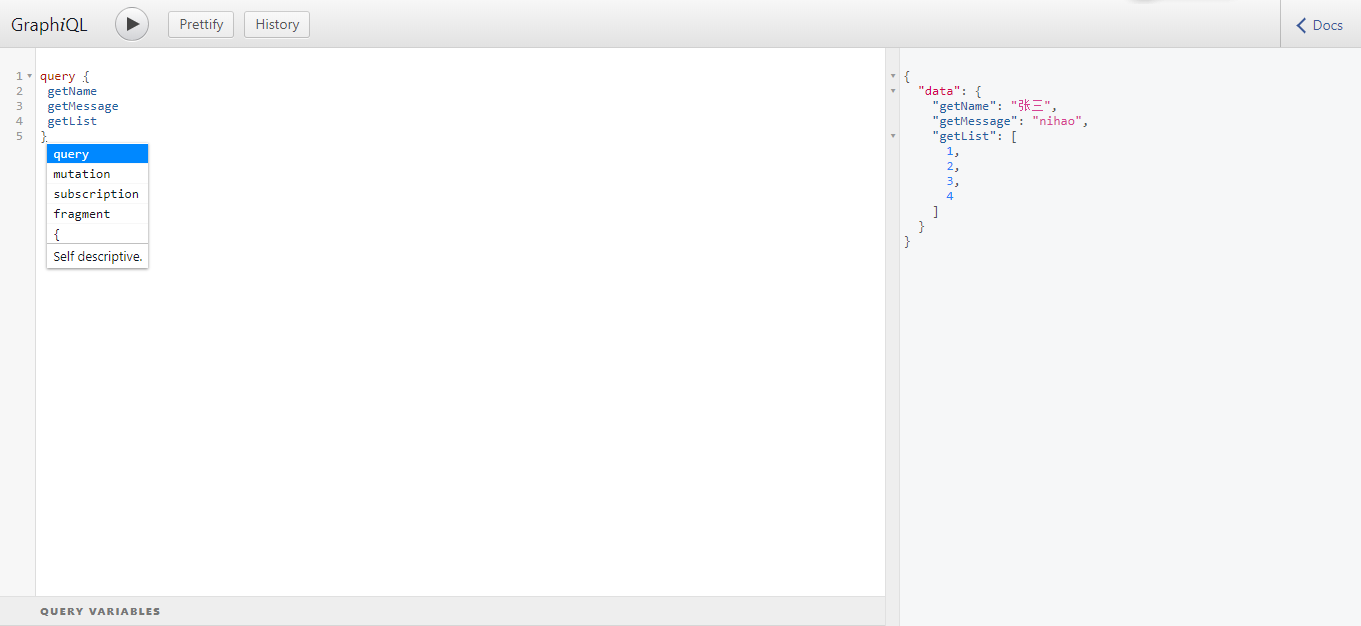
query {

getName

getMessage

getList

}

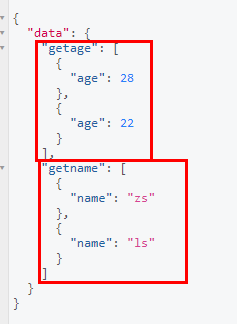


## **[查询别名](http://www.siyuweb.com/mysql/3259.html" \l "%E6%9F%A5%E8%AF%A2%E5%88%AB%E5%90%8D)**

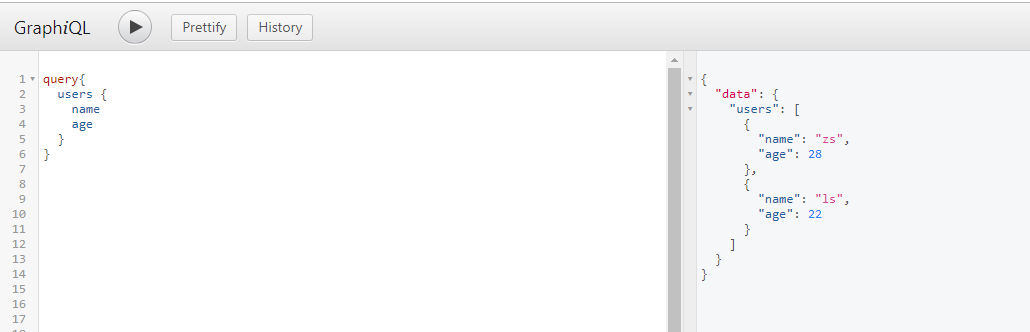
### 背景

当后端返回的是一个对象数组时，每个对象中有很多字段，比如name,sex,age等，

此时我可能想要得到其中由name组成的集合，由age组成的集合，如下：



但正常情况下我们获取的是这样的，name与age是在一个对象中的。



### **后端代码**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type User{

name: String!

sex: String

age: Int

hobby: [String]

}

type Query {

users:[User]

}

`);

// root 规定了顶层的 API 入口端点

var root = {

users: function () {

var users = [

{

name: 'zs',

sex: 'man',

age: 28,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

{

name: 'ls',

sex: 'woman',

age: 22,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

];

return users;

},

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### **前端**代码

此时我们就可以使用查询别名了，如下

query {

getage: users {

age

}

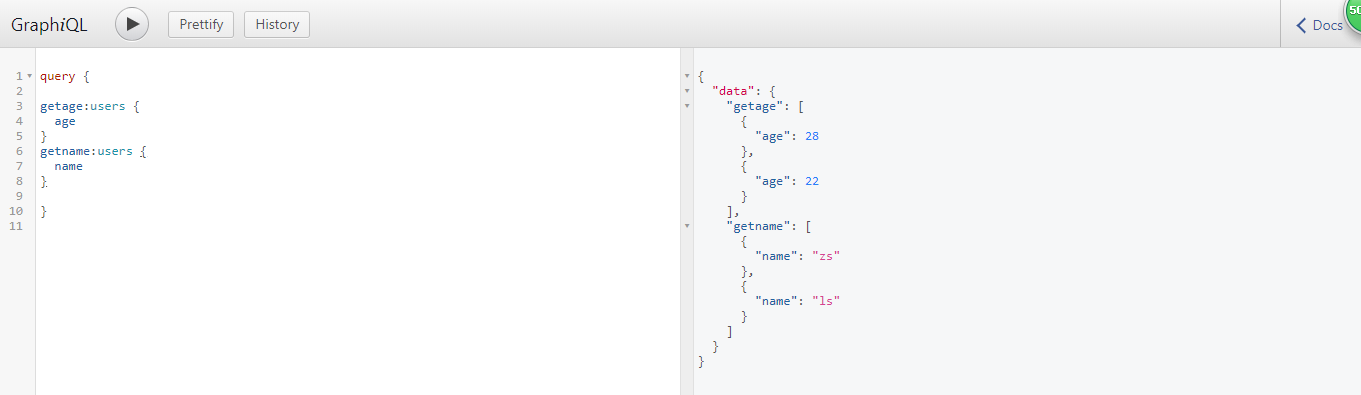
getname: users {

name

}

}

### **效果**



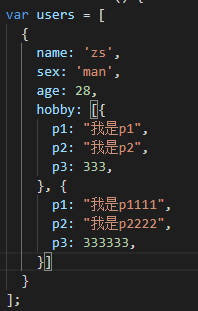
这里要注意了，查询别名不能一样，否则还是会被合并的。

## [嵌套](http://www.siyuweb.com/mysql/3259.html" \l "%E5%B5%8C%E5%A5%97%E6%9F%A5%E8%AF%A2)**[查询](http://www.siyuweb.com/mysql/3259.html" \l "%E5%B5%8C%E5%A5%97%E6%9F%A5%E8%AF%A2)**

### 基本说明

当我们返回的数据是一个对象数组，然后每个对象元素中有的属性的值又是对象数组，这时我们就可以使用嵌套查询了，

如下：



### **后端代码**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type Obj{

p1: String!

p2: String

p3: Int

}

type User{

name: String!

sex: String

age: Int

hobby: [Obj]

}

type Query {

users:[User]

hello:String

}

`);

// root 规定了顶层的 API 入口端点

var root = {

users: function () {

var users = [

{

name: 'zs',

sex: 'man',

age: 28,

hobby: [{

p1: "我是p1",

p2: "我是p2",

p3: 333,

}, {

p1: "我是p1111",

p2: "我是p2222",

p3: 333333,

}]

}

];

return users;

},

hello() {

return "你好啊";

}

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 效果



### 前端查询

query {

users {

name

sex

age

hobby {

p2

p3

}

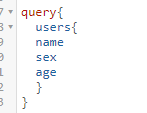
}

}

## [分片](http://www.siyuweb.com/mysql/3259.html" \l "%E5%88%86%E7%89%87)**查询**

### 说明

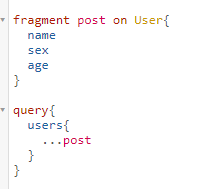
当我们返回的数据是一个对象或者对象数组时，我们必须在他的后面加上一个括号，里面给上需要的字段，每次查询都需要这样写，查询较多时就比较麻烦。



这时我们就可以采取分片查询了

如下：其中User是我在后端定义的要返回的类型，我根据他分片了个post出来，

下面我要查询相关字段时直接使用post就可以了（这里使用了es6的展开运算符）



### 后端代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 用 GraphQL schema language 构造一个 schema

var schema = buildSchema(`

type User{

name: String!

sex: String

age: Int

hobby: [String]

}

type Query {

users:[User]

}

`);

// root 规定了顶层的 API 入口端点

var root = {

users: function () {

var users = [

{

name: 'zs',

sex: 'man',

age: 28,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

{

name: 'ls',

sex: 'woman',

age: 22,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

},

];

return users;

},

}

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 前端代码

fragment post on User{

name

sex

age

}

query{

users{

...post

}

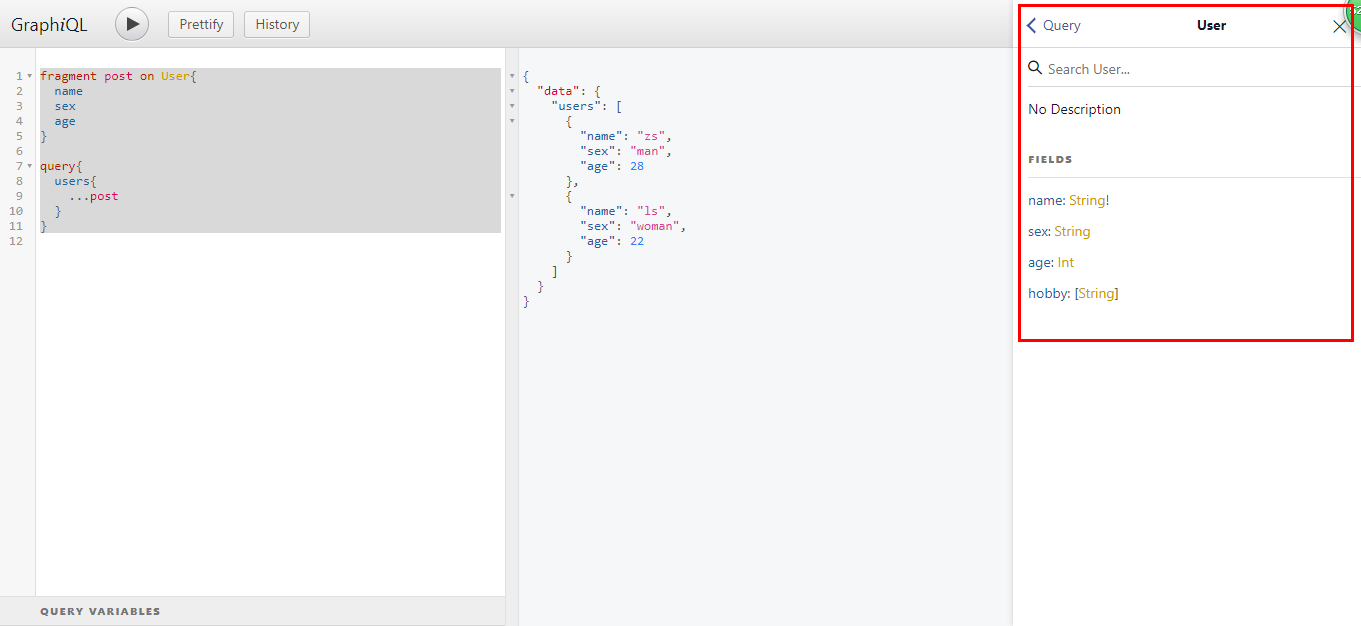
}

### 效果



## 自动生成的接口文档

在我们的页面中，我们可以获取查询的相关文档描述，就在最右边



## 实现完整的流程【mysql->nodejs->前端】

### nodejs中的回调问题-->无法直接返回数据

我们已经知道nodejs操作数据库是在回调函数中进行的，是无法直接返回操作结果.

**代码如下所示**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

var connection = getDBcon();

var sql1 = 'SELECT \* FROM zs\_2';

//像数据库查询

connection.query(sql1, function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

return;

}

console.log(JSON.stringify(result)); //将查询结果返回到前端

connection.end(); //操作完成后记得加上这句关闭数据库链接

return JSON.stringify(result);

});

return 'Hello world!';

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 解决方法：可以通过异步等待+promise成功返回数据

对于上面这个问题，我们就可以使用es8的异步等待+promise来实现。

#### 回忆promise的用法

所谓Promise ，简单说就是一个容器，里面保存着某个未来才回结束的事件(通常是一个异步操作）的结果。从语法上说，Promise是一个对象，它可以获取异步操作的消息。 并返回一个含有此消息的promise对象

###### Promise对象的两个特点

1. 对象的状态不受外界影响。

Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

1. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。

Promise 对象的状态改变，只有两种可能：从进行中变为已成功和从进行中变为已失败。只要这两种情况发生，状态就不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。如果改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

###### **promise**好处

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

###### **promise**缺点

1. 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
2. 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
3. 当处于进行中状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

###### promise**对象**中的then 方法

ES6规定，Promise对象是一个构造函数，用来生成Promise实例。Promise 实例生成以后，可以用then 方法分别指定成功状态和失败状态的回调函数。

**then方法可以接受两个回调函数作为参数：**

第一个回调函数是Promise 对象的状态变为resolved时调用，(成功的定型)

第二个回调函数是Promise 对象的状态变为rejected时调用。

其中，第二个函数是可选的，不一定要提供，这两个函数都接受Promise 对象传出的值作为参数。

###### promise对象中的catch方法

正常情况then方法会接受两个参数，分别表示成功的处理函数和失败的处理函数。

但是很多时候我们可能含有多个then方法，这时如果要每个then里面都写上失败处理函数就太麻烦了，这时我们就可以使用catch方法。

我们在最末尾加上catch方法，那么他就可以捕获前面任何地方产生的错误，如下：



**注意：**

一旦在catch前的某个then方法中写了第二个参数用于捕获错误，那么当前错误就会到此停止，后面就是正确的流程了。

如果在后面的流程中没产生其他错误，那么catch方法也不会再执行，如果出错了，那么catch仍然会继续执行。

#### **回**忆async await**【异**步等待】的用法

###### 异步等待的基本概念

1. async/await是es8的规范，async 后面一般跟一个函数，也就是async一般跟在function前。
2. await必须放在aysnc里面的函数体范围内
3. 在async修饰的函数里面，一定会返回一个promise对象，如果你没返回一个promise对象的话，默认会返回一个值为undefined的promise对象
4. await后面如果是同步执行代码，那么会等待其执行完成，如果是异步代码则只有返回一个primise对象时，才会进行等待的。

###### 异步等待的基本实例

**实例一： await后面函数执行的同步代码会等待**



var m=(async function () {

var d = new Date();

var n = await (function () {

while (new Date() - d < 5000);

return 8890;

})()

return n;

})()

**实例二： await后面的函数返回一个promise对象，会等待**



**注意：**

其实在一些时候让await等待的函数返回一个promise对象是非常有用的，因为这样你就可以确保让我们的操作一定会先执行完成。

### 具体实现

#### 后端代码

**数据库连接代码getDBcon.js**

//此模块作用为获取数据库的链接

var mysql = require('mysql'); //导入mysql模块

var defaultDBObj={

host: '127.0.0.1', //数据库地址

user: 'root', //数据库用户名

password: '123456A', //数据库密码

database: 'zs' //数据库名

}

function getDBcon(DBobj=defaultDBObj ){

var connection = mysql.createConnection(DBobj);

connection.connect(); //启动链接

return connection;

}

module.exports=getDBcon;

**构建graphql服务的文件 serve.js**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

//定义了一个去执行数据库操作的函数,同时接受一个对象，会把操作结果赋值到这个对象上

var fn = function (resobj) {

return new Promise(function (resovle, reject) {

var connection = getDBcon();

var sql1 = 'SELECT \* FROM zs\_2';

//像数据库查询

connection.query(sql1, function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

return;

}

resobj.resdata = JSON.stringify(result);

connection.end(); //操作完成后关闭数据库链接

resovle(JSON.stringify(result)); //执行此代码表示数据库操作完成

});

});

}

//定义一个立即执行的函数

var resData = async function () {

var obj = { resdata: "" }//准备接受数据库操作结果的对象

await fn(obj); // fn会返回一个promise对象，会等待，所以能成功

return obj.resdata; //等待数据库操作完成后，将数据赋值给obj.resdata，并返回

}();

return resData; //直接返回一个promise对象时，graphql已经给你处理过了，可以直接返回

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

#### 前端代码

fetch('http://localhost:4000/graphql/', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

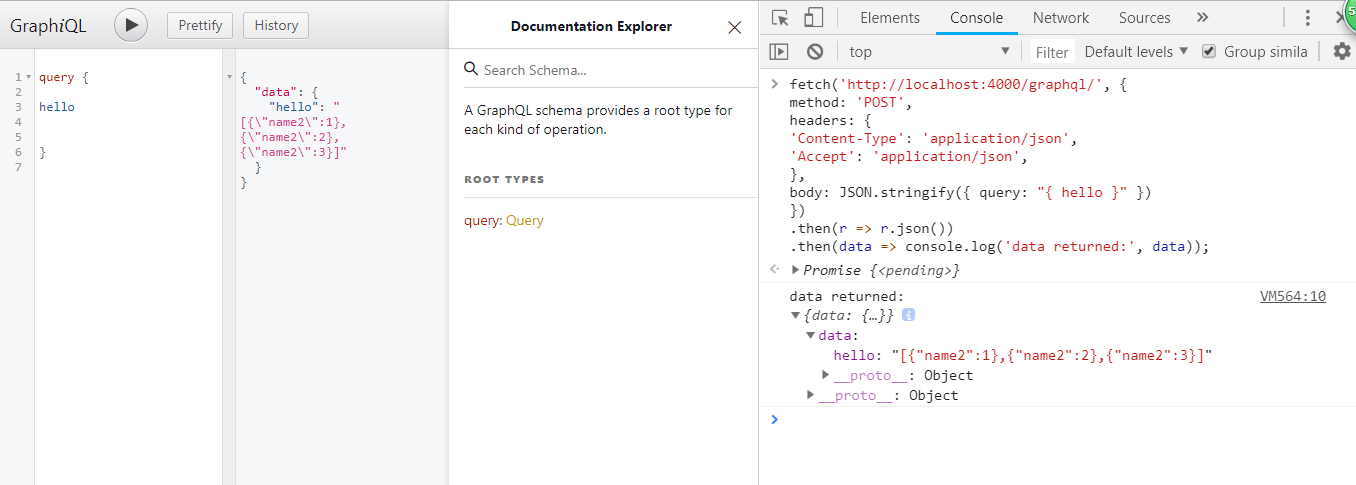
body: JSON.stringify({ query: "{ hello }" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

#### 效果



#### **只用promise也能实现**

由于只使用promise也可以实现同步，所以这里我们可以直接使用promise对象。

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

//定义了一个去执行数据库操作的函数,同时接受一个对象，会把操作结果赋值到这个对象上

var fn = function () {

return new Promise(function (resovle, reject) {

var connection = getDBcon();

var sql1 = 'SELECT \* FROM zs\_2';

//像数据库查询

connection.query(sql1, function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

return;

}

connection.end(); //操作完成后关闭数据库链接

resovle(JSON.stringify(result)); //执行此代码表示数据库操作完成

});

});

}

//定义一个立即执行的函数

return fn();

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

## 在**graphql**下跨域

还是和以前一样的代码，复制过来就能用了

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

else

next();

});

## 在graphql下验证token

也是和以前一样的方式，我们可以在两个地方进行验证

### 地方一：在跨域时进行验证token

这里简单的模拟了下token



**后端代码**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

var app = express();

var bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

else {

//可以在这里实现token验证

console.log("有请求进入了后端", req.body)

var obj = req.body;

if (obj.token != "123456") {

res.send(JSON.stringify({ state: "token无效，请重新登录" }))

}

else next(); //token通过进入下一个流程

}

});

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

**前端代码**



fetch('http://localhost:4000/graphql/', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

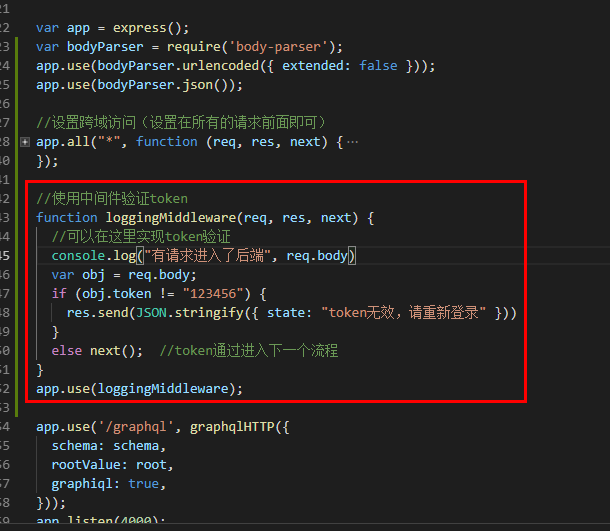
body: JSON.stringify({ query: "{ hello }", token: "1234567" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### 地方二：使用中间件进行验证token



**后端代码【前端等地方不做任何更改】**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

var app = express();

var bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

else

next();

});

//使用中间件验证token

function loggingMiddleware(req, res, next) {

//可以在这里实现token验证

console.log("有请求进入了后端", req.body)

var obj = req.body;

if (obj.token != "123456") {

res.send(JSON.stringify({ state: "token无效，请重新登录" }))

}

else next(); //token通过进入下一个流程

}

app.use(loggingMiddleware);

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 注意

由于token的验证时异步的，所以我们最好把token的验证结果作为一个promise对象放回

#### **token的封装如下**

注意：此处使用了es6的promise语法与class语法

/\*\*

\* 此class的作用为生成token与解析token

\*/

var jwt = require('jsonwebtoken')

class Token {

constructor () {}

/\*\*

\* 创建token

\* @title {Stiring} 创建token的主题信息，一般可以取当前用户的用户名

\*/

createToken (title) {

var content = {content: title}; // 要生成token的主题信息

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

var token = jwt.sign(content, secretOrPrivateKey, {

expiresIn: 3600\*24 // 一天后过期

})

// console.log('token ：' + token)

return token

}

/\*\*

\* 解析token

\* @token {Stiring} 要解析的token

\*/

analysisToken (token) {

return new Promise(function (resolve) {

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

jwt.verify(token, secretOrPrivateKey, function (err, decode) {

var obj;

if (err) { // 当token过期，或这是一个伪造的token，或这是无效的token时会触发此逻辑

obj = {

state: false,

result: err.name

}

} else {

obj = {

state: true,

result: decode.content

}

}

resolve(obj);

})

})

}

}

module.exports = Token

#### Token生成与校验函数的调用

//创建token

var token=new Token().createToken(\_this.email);

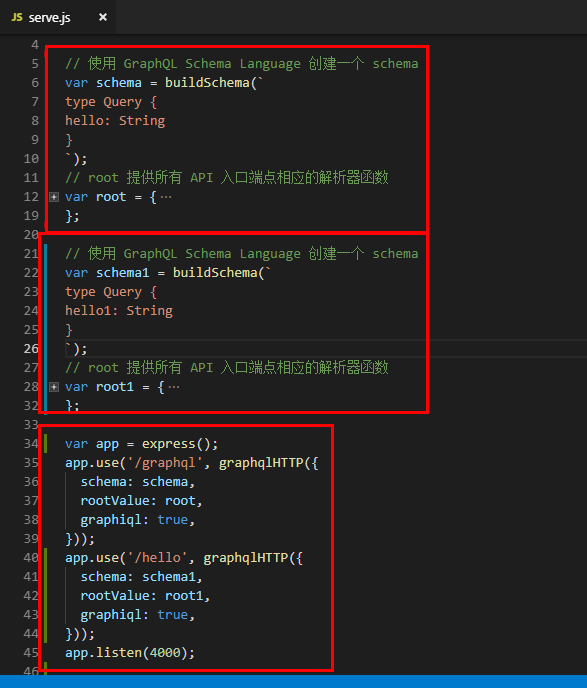
// 开始验证token

var checkResult = token.analysisToken(this.token)

#### **在graphql中的使用**



## 在后端可以构建多个graphql服务路径



**代码**

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

hello1: () => {

return 'Hello world11111111!';

},

};

// 使用 GraphQL Schema Language 创建一个 schema

var schema1 = buildSchema(`

type Query {

hello1: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root1 = {

hello1: () => {

return 'Hello world11111111!';

},

};

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.use('/hello', graphqlHTTP({

schema: schema1,

rootValue: root1,

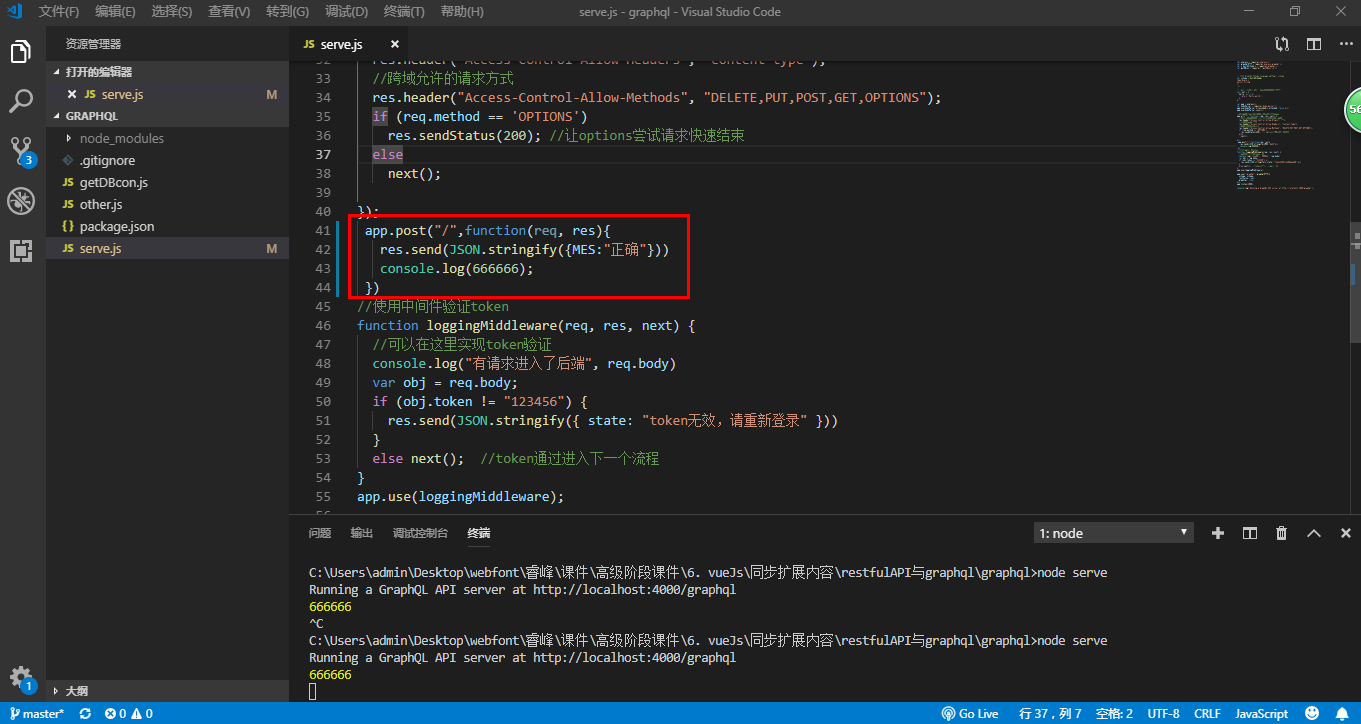
graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

## **也可以在后端直接创建restful API接口**



### 后端代码

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

var getDBcon = require("./getDBcon.js");

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

var app = express();

var bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

else

next();

});

app.post("/",function(req, res){

res.send(JSON.stringify({MES:"正确"}))

console.log(666666);

})

//使用中间件验证token

function loggingMiddleware(req, res, next) {

//可以在这里实现token验证

console.log("有请求进入了后端", req.body)

var obj = req.body;

if (obj.token != "123456") {

res.send(JSON.stringify({ state: "token无效，请重新登录" }))

}

else next(); //token通过进入下一个流程

}

app.use(loggingMiddleware);

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

### 前端代码

fetch('http://localhost:4000/', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

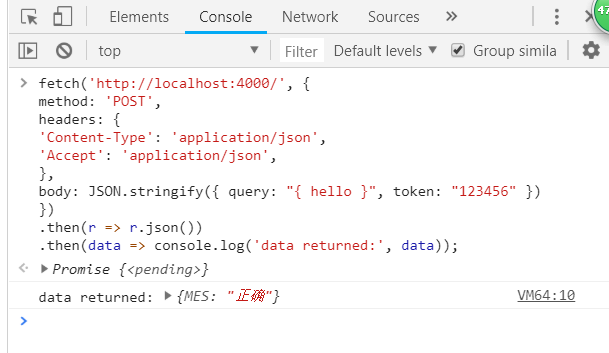
body: JSON.stringify({ query: "{ hello }", token: "123456" })

})

.then(r => r.json())

.then(data => console.log('data returned:', data));

### 运行效果



# 作业一

## 作业说明

请使用mysql+nodejs+graphql+fetch实现一个前后交互流程完整的客户管理系统。

## 描述

前端具有登录页面，注册页面，登录成功后会进入用户查询页面，在此页面可以进行查询当前组成的所有用户。

## 前端要求

1. 所有页面使用vue进行构建，由于页面内容比较简单，所以每个页面可以作为一个组件
2. 创建一个布局组件，使用slot接收登录和组成组件，布局组件默认只显示登录或注册组件中的一个。
3. 登录成功后的页面跳转通过动态组件实现切换（component + is）
4. 前端请求使用fetch进行发送

## 后端要求

1. 登录注册和查询所有用户的后端接口全部使用graphql实现，
2. 用户数据存储在mysql数据库中。
3. 后端实现跨域

# 作业二

在作业一的基础上对后端实现做调整

**后端要求**

1. 用户数据存储在mysql数据中。
2. 后端实现跨域
3. 把登录注册接口改为Restful接口实现（不需要token验证），其他使用graphql接口实现（需要token验证）

# 作业一参考实现【纯graphql】

## **后端实现**

所有的的接口都使用graphql接口实现

## 第一步： 创建数据库，创建用户表

1. 在mysql中创建数据库：graphql
2. 在graphql数据库中创建用户表：users

/\*

Navicat MySQL Data Transfer

Source Server : W1807

Source Server Version : 50051

Source Host : localhost:3306

Source Database : graphql

Target Server Type : MYSQL

Target Server Version : 50051

File Encoding : 65001

Date: 2019-04-14 19:25:10

\*/

SET FOREIGN\_KEY\_CHECKS=0;

-- ----------------------------

-- Table structure for users

-- ----------------------------

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

`id` int(11) NOT NULL auto\_increment,

`username` varchar(50) NOT NULL,

`password` varchar(50) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

-- ----------------------------

-- Records of users

-- ----------------------------

## 第二步： 构建graphql服务器

1. 安装如下模块，生成package.json文件

{

"dependencies": {

"body-parser": "^1.18.3",

"express": "^4.16.4",

"express-graphql": "^0.7.1",

"graphql": "^14.2.1",

"jsonwebtoken": "^8.5.1",

"mysql": "^2.16.0"

}

}

1. 使用nodejs链接数据库

getDBcon.js

//此模块作用为获取数据库的链接

var mysql = require('mysql'); //导入mysql模块

var defaultDBObj={

host: '127.0.0.1', //数据库地址

user: 'root', //数据库用户名

password: '123456A', //数据库密码

database: 'graphql' //数据库名

}

function getDBcon(DBobj=defaultDBObj ){

var connection = mysql.createConnection(DBobj);

connection.connect(); //启动链接

return connection;

}

module.exports=getDBcon;

1. 使用nodejs搭建graphql服务器
2. 安装与引入模块
3. 创建schema【接口列表】
4. 创建root对象【接口实现】
5. 使用schema与root构建graphql服务器

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

// 使用root与schema构建一个graphql服务器

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

## 第三步： 给服务器设置跨域访问

在app创建后，添加如下代码即可

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

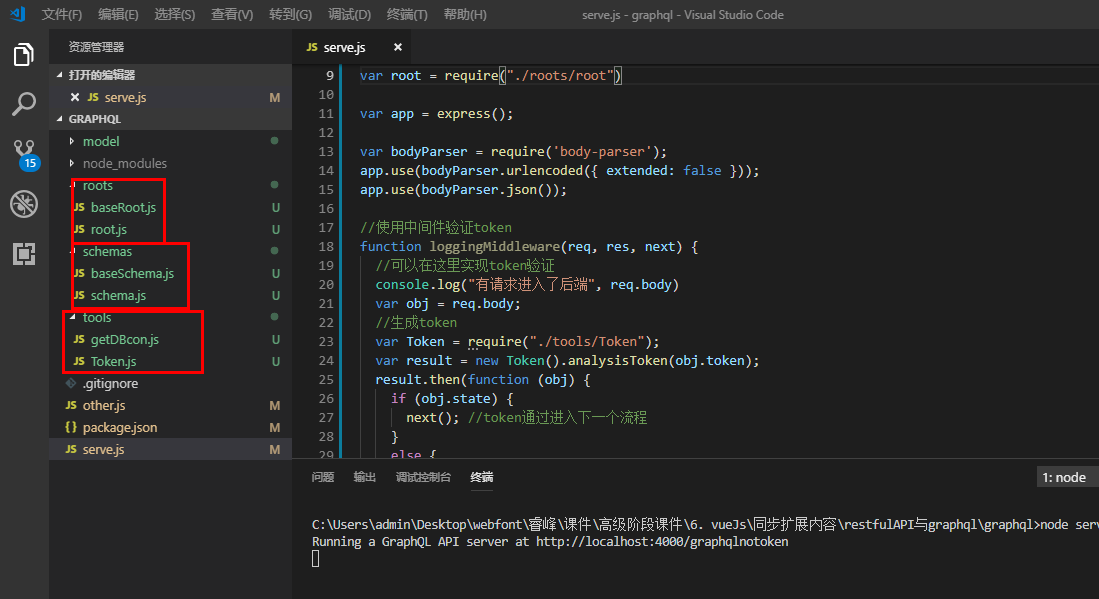
else {

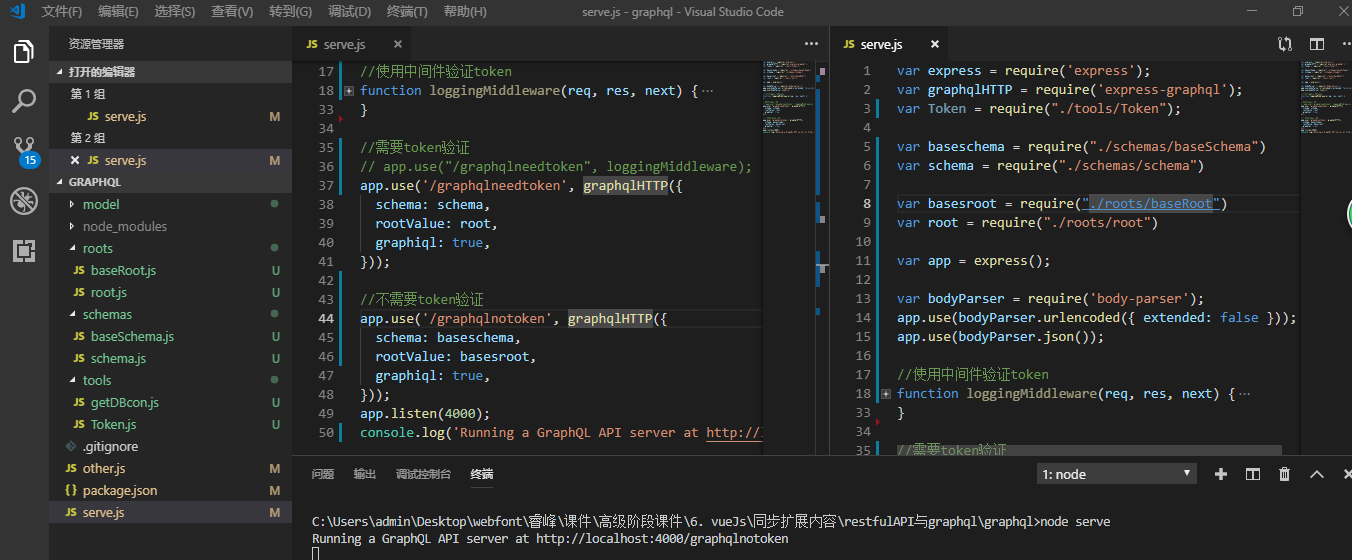
next();

}

});

## **第四步： 将schema，root对象，实现分文件写**





## **第四步： 完成schema**

**baseschema.js**

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

input Userobj{

username:String

password:String

}

type LoginsreturnObj{

token:String

result:String

}

type Query {

hello: String

login(user:Userobj): LoginsreturnObj

}

type Mutation{

registry(user:Userobj): String

}

`);

module.exports=schema;

**schema.js**

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Usermes{

id:Int

username:String

}

type QuallRobj{

result:[Usermes]

token:String

}

type Query {

queryall: QuallRobj

}

`);

module.exports = schema;

## **第六步： 完善root.js**

**baseRoot.js**

var User=require("../model/User")

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

registry: new User().registry,

login:new User().login,

};

module.exports=root;

**root.js**

var User=require("../model/User")

// root 提供所有 API 入口端点相应的解析器函数

var root = {

queryall: new User().queryAll,

};

module.exports = root;

## **第五步： 使用graphql构建业务接口实现类User.js**

构建接口时应该按照业务实体进行划分，然后使用es6的语法创建class.

**创建User实体类，包含注册，登录，查询方法。**

class User{

//注册接口实现，rest接口会调用他

registry(){

}

//登录接口实现，rest接口会调用他

login(){

}

//查询接口实现，graphql接口会调用他

queryAll(){

}

}

（注意：如果实体中的每个方法都非常复杂，那么我们可以把每个方法单独的定义为一个文件）

**User.js**

//数据库连接对象实现

var DBcon = require("../tools/getDBcon");

class User {

//注册接口实现

registry({ user }) {

return new Promise(function (resovle, reject) {

var connection = DBcon();

var sql1 = 'insert into users(username,password) values(?,?)';

//向数据库注册

connection.query(sql1, [user.username, user.password], function (err, result) {

if (err) {

console.log('[insert ERROR] - ', err.message);

reject("注册失败" + err.message)

}

else if (result.affectedRows <= 0) {

reject("注册失败，数据未进入到数据库")

}

else {

resovle("注册成功，数据成功添加到数据库")

}

connection.end(); //操作完成后关闭数据库链接

});

});

}

//登录接口实现

login({ user }) {

return new Promise(function (resovle, reject) {

var connection = DBcon();

var sql1 = 'select \* from users where username=? and password=?';

//向数据库注册

connection.query(sql1, [user.username, user.password], function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

resovle({

result: "登录失败",

token: "",

})

}

else if (result.length <= 0) {

resovle({

result: "登录失败,用户名与密码不匹配",

token: ""

})

}

else {

//生成token

var Token = require("../tools/Token");

var token = new Token().createToken(user.username);

//组装返回对象

resovle({

result: "登录成功",

token: token

})

}

connection.end(); //操作完成后关闭数据库链接

});

});

}

//查询接口实现

queryAll() {

return new Promise(function (resovle, reject) {

var connection = DBcon();

var sql1 = 'select id,username from users';

//向数据库注册

connection.query(sql1, function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

reject(null)

}

else if (result.length <= 0) {

console.log('无任何用户数据');

reject(null)

}

else {

//生成token

var Token = require("../tools/Token");

var token = new Token().createToken("test");

//组装返回对象

var obj = {

result: result,

token: token

}

resovle(obj); //执行此代码表示数据库操作完成

}

connection.end(); //操作完成后关闭数据库链接

});

});

}

}

module.exports = User;

## **第七步： 创建token.js文件**

创建token.js文件，实现对token的验证和解析

/\*\*

\* 此class的作用为生成token与解析token

\*/

var jwt = require('jsonwebtoken')

class Token {

/\*\*

\* 创建token

\* @title {Stiring} 创建token的主题信息，一般可以取当前用户的用户名

\*/

createToken(title) {

var content = { content: title }; // 要生成token的主题信息

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

var token = jwt.sign(content, secretOrPrivateKey, {

expiresIn: 3600 \* 24 // 一天后过期

})

// console.log('token ：' + token)

return token

}

/\*\*

\* 解析token

\* @token {Stiring} 要解析的token

\*/

analysisToken(token) {

return new Promise(function (resolve) {

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

jwt.verify(token, secretOrPrivateKey, function (err, decode) {

var obj;

if (err) { // 当token过期，或这是一个伪造的token，或这是无效的token时会触发此逻辑

obj = {

state: false,

result: err.name

}

} else {

obj = {

state: true,

result: decode.content

}

}

resolve(obj);

})

})

}

}

module.exports = Token

Token生成与校验函数的调用

//创建token

var token=new Token().createToken(\_this.email);

// 开始验证token

var checkResult = new Token().analysisToken(this.token)

## **第八步： 在serve文件中实现token验证**

这里使用中间件形式来进行token的验证

var express = require('express');

var graphqlHTTP = require('express-graphql');

var Token = require("./tools/Token");

var baseschema = require("./schemas/baseSchema")

var schema = require("./schemas/schema")

var basesroot = require("./roots/baseRoot")

var root = require("./roots/root")

var app = express();

var bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

//使用中间件验证token

function loggingMiddleware(req, res, next) {

//可以在这里实现token验证

console.log("有请求进入了后端", req.body)

var obj = req.body;

//生成token

var Token = require("./tools/Token");

var result = new Token().analysisToken(obj.token);

result.then(function (obj) {

if (obj.state) {

next(); //token通过进入下一个流程

}

else {

res.send(JSON.stringify({ state: "token无效，请重新登录" }))

}

})

}

//需要token验证

// app.use("/graphqlneedtoken", loggingMiddleware);

app.use('/graphqlneedtoken', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

//不需要token验证

app.use('/graphqlnotoken', graphqlHTTP({

schema: baseschema,

rootValue: basesroot,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphqlnotoken');

## **前端实现**

## 第一步：准备fetch请求代码

### 注册代码

fetch('http://localhost:4000/graphqlnotoken', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query: `mutation { registry(user:{username:"zs66",password:"123456"})}`

})

})

.then(r => r.json())

.then(data => console.log('data returned:', data));



### 登录代码

fetch('http://localhost:4000/graphqlnotoken', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({

query: `query { login(user:{username:"zs",password:"123456"}){result,token}}`

})

})

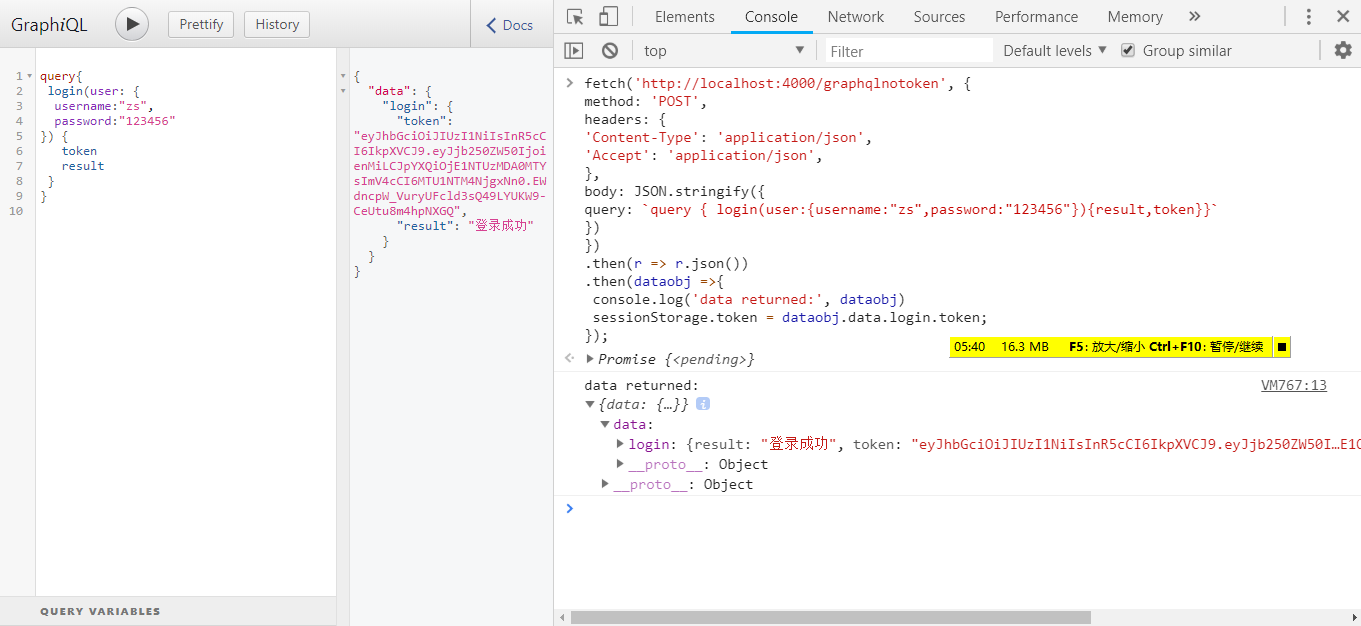
.then(r => r.json())

.then(dataobj =>{

console.log('data returned:', dataobj)

sessionStorage.token = dataobj.data.login.token;

});



注意：登录成功后需要把后端返回来的token保存起来放到sessionStorage中。

### 请求所有的用户数据代码

这里需要带上登录成功后返回的token

var parobj = JSON.stringify({ query: `query{queryall{token result{id username}}}`, token: sessionStorage.token })

fetch("http://localhost:4000/graphqlneedtoken", {

method: "post",

headers: {

"Accept": "application/json",

"Content-Type": "application/json"

},

body: parobj,

})

.then(function (data) {

return data.json()

})

.then(function (data) {

console.log(data);

})



# 作业二参考实现【Rest+graphql混合】

## **后端实现**

这里登录注册等不需要验证token的接口使用Rest来实现，其他需要验证token的接口使用graphql接口实现

## 第一步： 创建数据库，创建用户表

1. 在mysql中创建数据库：graphql
2. 在graphql数据库中创建用户表：users

/\*

Navicat MySQL Data Transfer

Source Server : W1807

Source Server Version : 50051

Source Host : localhost:3306

Source Database : graphql

Target Server Type : MYSQL

Target Server Version : 50051

File Encoding : 65001

Date: 2019-04-14 19:25:10

\*/

SET FOREIGN\_KEY\_CHECKS=0;

-- ----------------------------

-- Table structure for users

-- ----------------------------

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

`id` int(11) NOT NULL auto\_increment,

`username` varchar(50) NOT NULL,

`password` varchar(50) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

-- ----------------------------

-- Records of users

-- ----------------------------

## 第二步： 构建graphql服务器

1. 安装如下模块，生成package.json文件

{

"dependencies": {

"body-parser": "^1.18.3",

"express": "^4.16.4",

"express-graphql": "^0.7.1",

"graphql": "^14.2.1",

"jsonwebtoken": "^8.5.1",

"mysql": "^2.16.0"

}

}

1. 使用nodejs链接数据库

getDBcon.js

//此模块作用为获取数据库的链接

var mysql = require('mysql'); //导入mysql模块

var defaultDBObj={

host: '127.0.0.1', //数据库地址

user: 'root', //数据库用户名

password: '123456A', //数据库密码

database: 'graphql' //数据库名

}

function getDBcon(DBobj=defaultDBObj ){

var connection = mysql.createConnection(DBobj);

connection.connect(); //启动链接

return connection;

}

module.exports=getDBcon;

1. 使用nodejs搭建graphql服务器
2. 安装与引入模块
3. 创建schema【接口列表】
4. 创建root对象【接口实现】
5. 使用schema与root构建graphql服务器

var express = require('express');

var graphqlHTTP = require('express-graphql');

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type Query {

hello: String

}

`);

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: () => {

return 'Hello world!';

},

};

// 使用root与schema构建一个graphql服务器

var app = express();

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

## 第三步： 给服务器设置跨域访问

在app创建后，添加如下代码即可

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

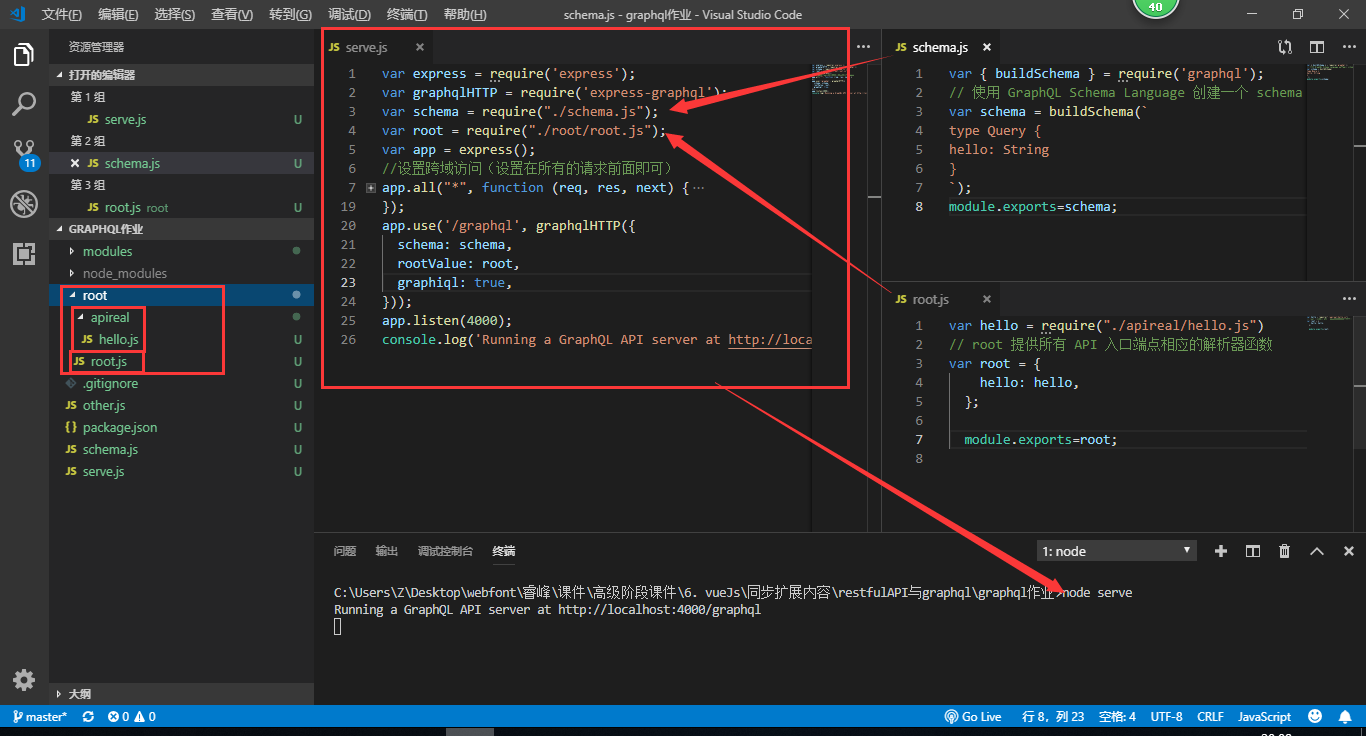
else {

next();

}

});

## **第四步： 将schema，root对象，实现分文件写**



## **第四步： 完成schema内容**

var { buildSchema } = require('graphql');

// 使用 GraphQL Schema Language 创建一个 schema

var schema = buildSchema(`

type UserObj{

id:Int

username:String

}

type QueryALLSuccessObj{

result:[UserObj]

token:String

}

type Query {

hello: String

queryAll:QueryALLSuccessObj

}

`);

module.exports = schema;

## **第五步： 使用graphql构建业务接口实现类User.js**

构建接口时应该按照业务实体进行划分，然后使用es6的语法创建class.

**创建User实体类，包含注册，登录，查询方法。**

class User{

//注册接口实现，rest接口会调用他

registry(){

}

//登录接口实现，rest接口会调用他

login(){

}

//查询接口实现，graphql接口会调用他

queryAll(){

}

}

（注意：如果实体中的每个方法都非常复杂，那么我们可以把每个方法单独的定义为一个文件）

**User.js**

//数据库连接对象实现

var DBcon = require("../../modules/getDBcon");

class User {

//注册接口实现

registry(user, res) {

var connection = DBcon();

var sql1 = 'insert into users(username,password) values(?,?)';

//向数据库注册

connection.query(sql1, [user.username, user.password], function (err, result) {

if (err) {

console.log('[insert ERROR] - ', err.message);

res.send(JSON.stringify({

state: "注册失败",

msg: err.message,

}))

}

else if (result.affectedRows <= 0) {

res.send(JSON.stringify({

state: "注册失败",

msg: "数据未进入到数据库",

}))

}

else {

res.send(JSON.stringify({

state: "注册成功",

msg: "数据成功添加到数据库",

}))

}

connection.end(); //操作完成后关闭数据库链接

});

}

//登录接口实现

login(user, res) {

var connection = DBcon();

var sql1 = 'select \* from users where username=? and password=?';

//向数据库注册

connection.query(sql1, [user.username, user.password], function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

res.send(JSON.stringify({

state: "登录失败",

msg: err.message,

}))

}

else if (result.length <= 0) {

res.send(JSON.stringify({

state: "登录失败",

msg: "用户名与密码不匹配",

}))

}

else {

//生成token

var Token = require("../../modules/Token");

var token = new Token().createToken(user.username);

//组装返回对象

res.send(JSON.stringify({

state: "登录成功",

token: token

}))

}

connection.end(); //操作完成后关闭数据库链接

});

}

//查询接口实现

queryAll() {

return new Promise(function (resovle, reject) {

var connection = DBcon();

var sql1 = 'select id,username from users';

//向数据库注册

connection.query(sql1, function (err, result) {

if (err) {

console.log('[SELECT ERROR] - ', err.message);

reject(null)

}

else if (result.length <= 0) {

console.log('无任何用户数据');

reject(null)

}

else {

//生成token

var Token = require("../../modules/Token");

var token = new Token().createToken("test");

//组装返回对象

var obj = {

result: result,

token: token

}

resovle(obj); //执行此代码表示数据库操作完成

}

connection.end(); //操作完成后关闭数据库链接

});

});

}

}

module.exports = User;

## **第六步： 完善root.js**

//hello接口实现

var hello = require("./apireal/hello.js")

//用户业务实现

var User = require("./apireal/User.js");

var user = new User();

// root 提供所有 API 入口端点相应的解析器函数

var root = {

hello: hello,

queryAll: user.queryAll,

};

module.exports = root;

## **第七步： 创建token.js文件**

创建token.js文件，实现对token的验证和解析

/\*\*

\* 此class的作用为生成token与解析token

\*/

var jwt = require('jsonwebtoken')

class Token {

/\*\*

\* 创建token

\* @title {Stiring} 创建token的主题信息，一般可以取当前用户的用户名

\*/

createToken(title) {

var content = { content: title }; // 要生成token的主题信息

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

var token = jwt.sign(content, secretOrPrivateKey, {

expiresIn: 3600 \* 24 // 一天后过期

})

// console.log('token ：' + token)

return token

}

/\*\*

\* 解析token

\* @token {Stiring} 要解析的token

\*/

analysisToken(token) {

return new Promise(function (resolve) {

var secretOrPrivateKey = 'www.blogzl.com' // 这是加密的key（密钥或私钥）

jwt.verify(token, secretOrPrivateKey, function (err, decode) {

var obj;

if (err) { // 当token过期，或这是一个伪造的token，或这是无效的token时会触发此逻辑

obj = {

state: false,

result: err.name

}

} else {

obj = {

state: true,

result: decode.content

}

}

resolve(obj);

})

})

}

}

module.exports = Token

Token生成与校验函数的调用

//创建token

var token=new Token().createToken(\_this.email);

// 开始验证token

var checkResult = new Token().analysisToken(this.token)

## **第八步： 在serve文件中实现token验证**

这里使用中间件形式来进行token的验证

var express = require('express');

var graphqlHTTP = require('express-graphql');

var schema = require("./schema.js");

var root = require("./root/root.js");

var app = express();

// 使用bodyParser中间件处理post请求参数

var bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

//设置跨域访问（设置在所有的请求前面即可）

app.all("\*", function (req, res, next) {

//设置允许跨域的域名，\*代表允许任意域名跨域

res.header("Access-Control-Allow-Origin", "\*");

//允许的header类型

res.header("Access-Control-Allow-Headers", "content-type");

//跨域允许的请求方式

res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");

if (req.method == 'OPTIONS')

res.sendStatus(200); //让options尝试请求快速结束

else {

next();

}

});

//注册使用rest接口

app.post("/registry", function (req, res) {

var User = require("./root/apireal/User");

var user = new User();

user.registry(req.body, res)

})

//登录使用rest接口

app.post("/login", function (req, res) {

var User = require("./root/apireal/User");

var user = new User();

user.login(req.body, res)

})

//其他请求全部使用graphql接口，都需要使用中间件验证token

//一旦全部开启token验证后，我们就无法使用相应的浏览器调试graphql接口的工具了，

//对于这点我们可以使用使用rest接口控制一个数据库标识 变量来决定

function loggingMiddleware(req, res, next) {

//可以在这里实现token验证

console.log("有请求进入了后端", req.body)

var obj = req.body;

//获取token并进行验证

var Token = require("./modules/Token.js");

var checkResult = new Token().analysisToken(obj.token)

checkResult.then(function (checkobj) {

// token验证通过

if (checkobj.state) {

next(); //token通过进入下一个流程

}

else {

res.send(JSON.stringify({ state: "token无效，请重新登录" }))

}

})

}

app.use('/graphql', loggingMiddleware);

app.use('/graphql', graphqlHTTP({

schema: schema,

rootValue: root,

graphiql: true,

}));

app.listen(4000);

console.log('Running a GraphQL API server at http://localhost:4000/graphql');

## **前端实现**

## 第一步：准备fetch请求代码

### 注册代码

fetch('http://localhost:4000/registry', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Accept': 'application/json',

},

body: JSON.stringify({username:"张三6",password:"123456"})

})

.then(r => r.json())

.then(data => console.log('data returned:', data));



### 登录代码

登录成功后需要把后端返回来的token保存起来放到sessionStorage中。

var parobj = JSON.stringify({username:"张三6",password:"123456"});

fetch("http://localhost:4000/login", {

method: "post",

headers: {

"Accept": "application/json",

"Content-Type": "application/json"

},

body: parobj,

})

.then(function (data) {

return data.json()

})

.then(function (data) {

sessionStorage.token=data.token;

console.log(data);

})



### 请求所有的用户数据代码

这里需要带上登录成功后返回的token

var parobj = JSON.stringify({query:"{queryAll{result{id username}}}",token:sessionStorage.token});

fetch("http://localhost:4000/graphql", {

method: "post",

headers: {

"Accept": "application/json",

"Content-Type": "application/json"

},

body: parobj,

})

.then(function (data) {

return data.json()

})

.then(function (data) {

console.log(data);

})



## 第二步：使用vue构建登录，注册，查询三个页面

请求尝试自己实现。。。

## 第三步：实现登录，注册，查询三个功能

请求尝试自己实现。。。

# graphql进阶使用

## **添加接口文档---方式一 使用#注释**

### 给类型名/方法名加上注释

只需要在方法名/类型名的的上一行添加注释代码即可（必须紧紧挨着，不能隔行）

# ====================schema：类似路由文件=================

#读操作

type Query {

#查询用户信息

user(input:User): String

#查询商品信息

books\_getbooks:String

}

# 写操作

type Mutation {

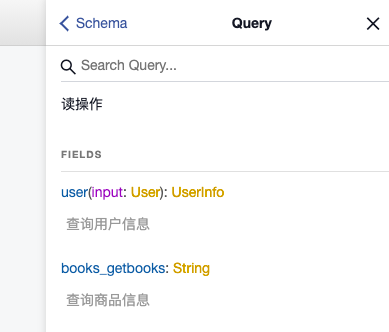
#设置用户信息

setUser: String

#设置商品信息

books\_setBooks:String

}



### 给请求参数或者返回字段名加上注释

# 用户输入的相关参数

input User{

#姓名

name: String!

#性别

sex: String

#年龄

age: Int

#爱好

hobby: [String]

}

# 返回的用户信息

type UserInfo{

#姓名

name: String!

#性别

sex: String

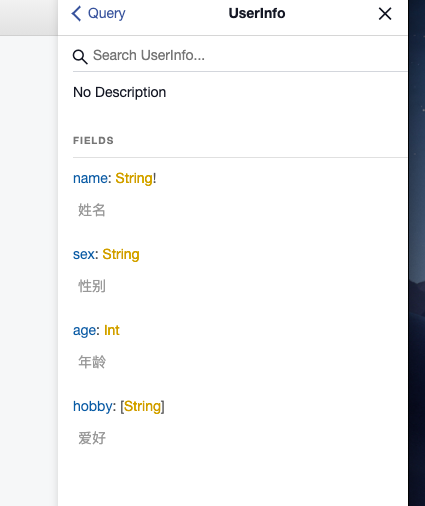
#年龄

age: Int

#爱好

hobby: [String]

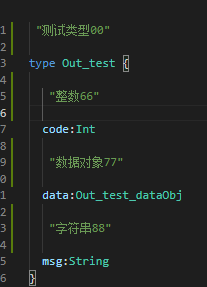
}



## **添加接口文档---方式二 使用** "注释"

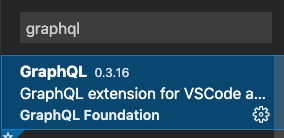
这种 "注释"写法和上面那种差不多，唯一的好处是：字段和注释不是紧挨着，隔有空行也可以生效 ，使用#号注释的则不行

如：

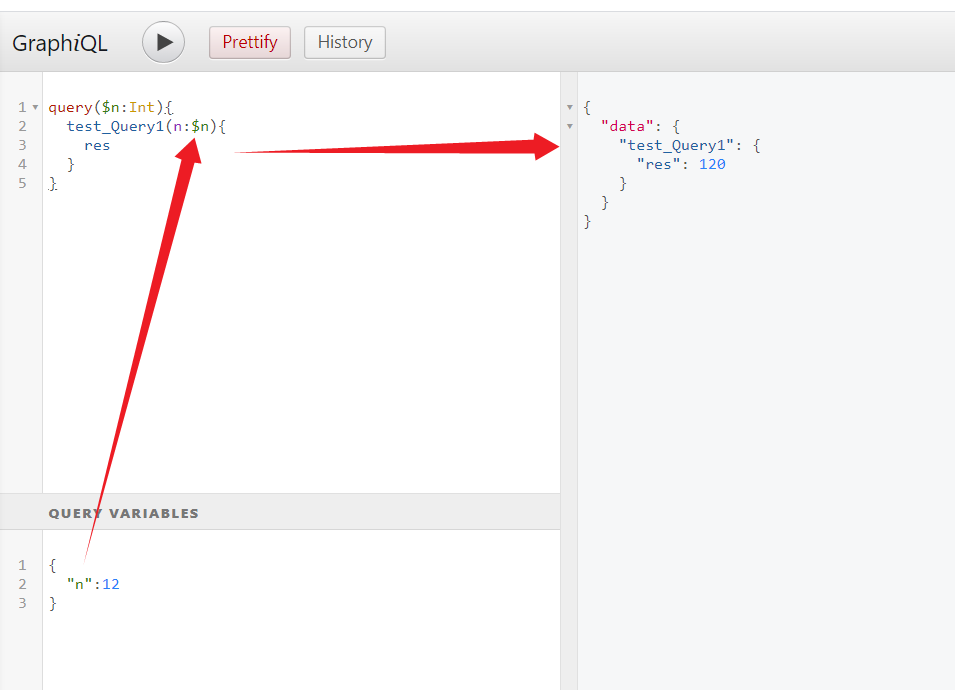
 

## **安装vscode插件对代码进行颜色高亮**

graphql



## 在使用graphql自带的调试器时，也可以直接传参的，写法和通过前端的代码调用传参方式差不多



## Directives 指令

### 指令可以改变 GraphQL 的执行行为，比如有条件地包含或跳过一个字段。



GraphQL 中目前有两个指令：

@include(if: Boolean) 仅在参数为 true 时，包含此字段。

@skip(if: Boolean) 如果参数为 true，跳过此字段。

// 请求

query myQuery($login:String = "kikajack",$showID:Boolean!){

user(login:$login) {

login

id @include(if: $showID)

}

}

// 参数

{

"login":"kikajack",

"showID": true

}

### **内联片段也可用于将指令应用于一组字段。如果省略了TypeCondition，则认为内联片段与封闭上下文的类型相同。**

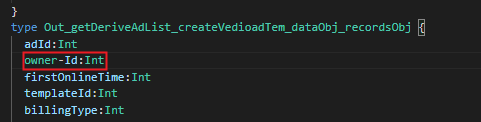


## graphql下如何排查bug

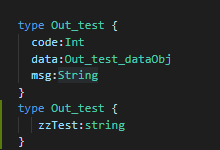
1. Node端写好graphql接口后，先用grapgqli调通
2. 然后把查询结果赋值到postman,长久保存一个记录
3. 然后观察请求来时，node端的日志，然后进行排查

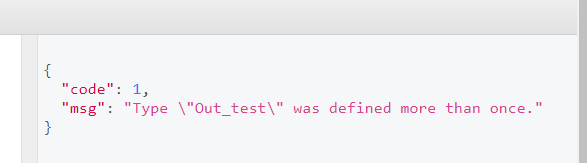
# Graphql要注意的一些问题

## Graphql中字段不能是中杠线分割的单词



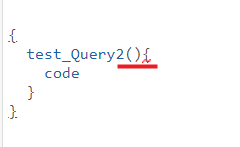
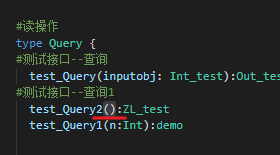
## Graphql中，类型定义多次时会报错





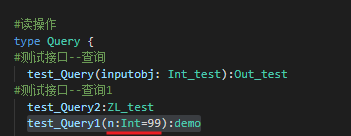
## Graphql schemal字段不接收参数时，不能写空括号吗，否则会报错

定义和调用都不能写空括号



## Graphql schemal参数可以给上默认值

定义默认值



不传时以默认值生效



传了时，以传的生效



## 对于 键为动态数字的 对象，graphql如何处理

如：键为1-31中的任何一个值，值类型为字符串，这时实际上就是说，让键为一个动态的值，虽然类型可以确定，

暂时我没找到比较好的处理方式，而是将其构造成了{key:”键”，val:”值”} 这种形式，这种情况下，类型就固定了，graphql就可以构造对应的类型对象了。

## 在input对象中的对象也必须是input类型

否则会报错

## Graphql中的其他类型

接口，枚举 ，联合类型，有时候会有点用

# Graphql在egg.js中的使用

## 安装模块

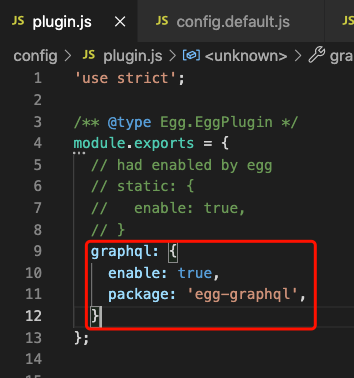
npm i egg-graphql -S

npm i zl-create-graphql-type -D

## 基本配置

## 在app目录下新建 graphql文件夹目录

## 开启插件



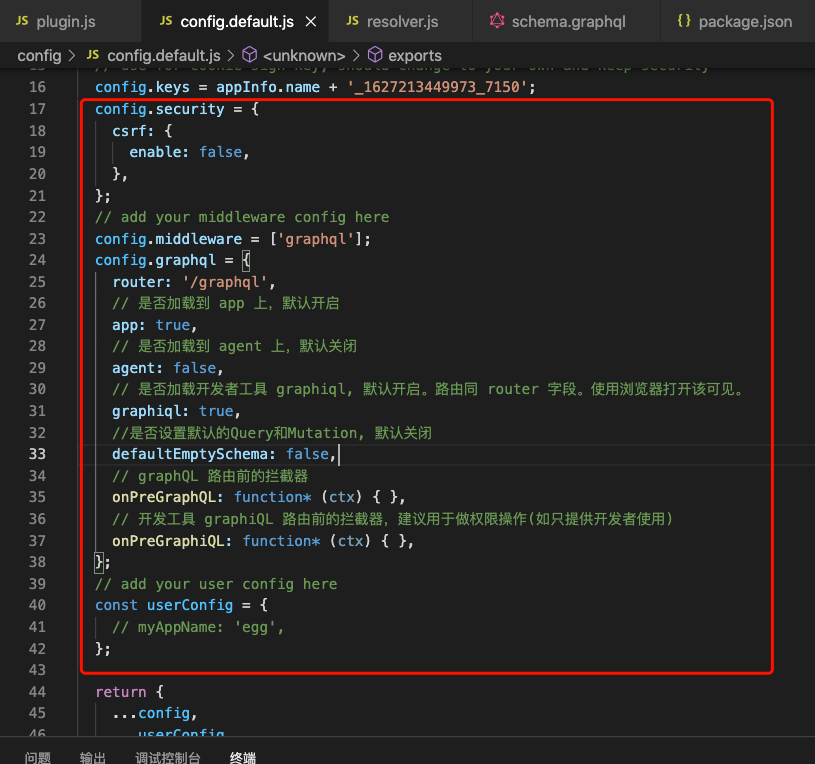
graphql: {

enable: true,

package: 'egg-graphql',

}

## 在配置文件中进行配置



config.security = {

csrf: {

enable: false,

},

};

// add your middleware config here

config.middleware = ['graphql'];

config.graphql = {

router: '/graphql',

// 是否加载到 app 上，默认开启

app: true,

// 是否加载到 agent 上，默认关闭

agent: false,

// 是否加载开发者工具 graphiql, 默认开启。路由同 router 字段。使用浏览器打开该可见。

graphiql: true,

//是否设置默认的Query和Mutation, 默认关闭

defaultEmptySchema: false,

// graphQL 路由前的拦截器

onPreGraphQL: function\* (ctx) { },

// 开发工具 graphiQL 路由前的拦截器，建议用于做权限操作(如只提供开发者使用)

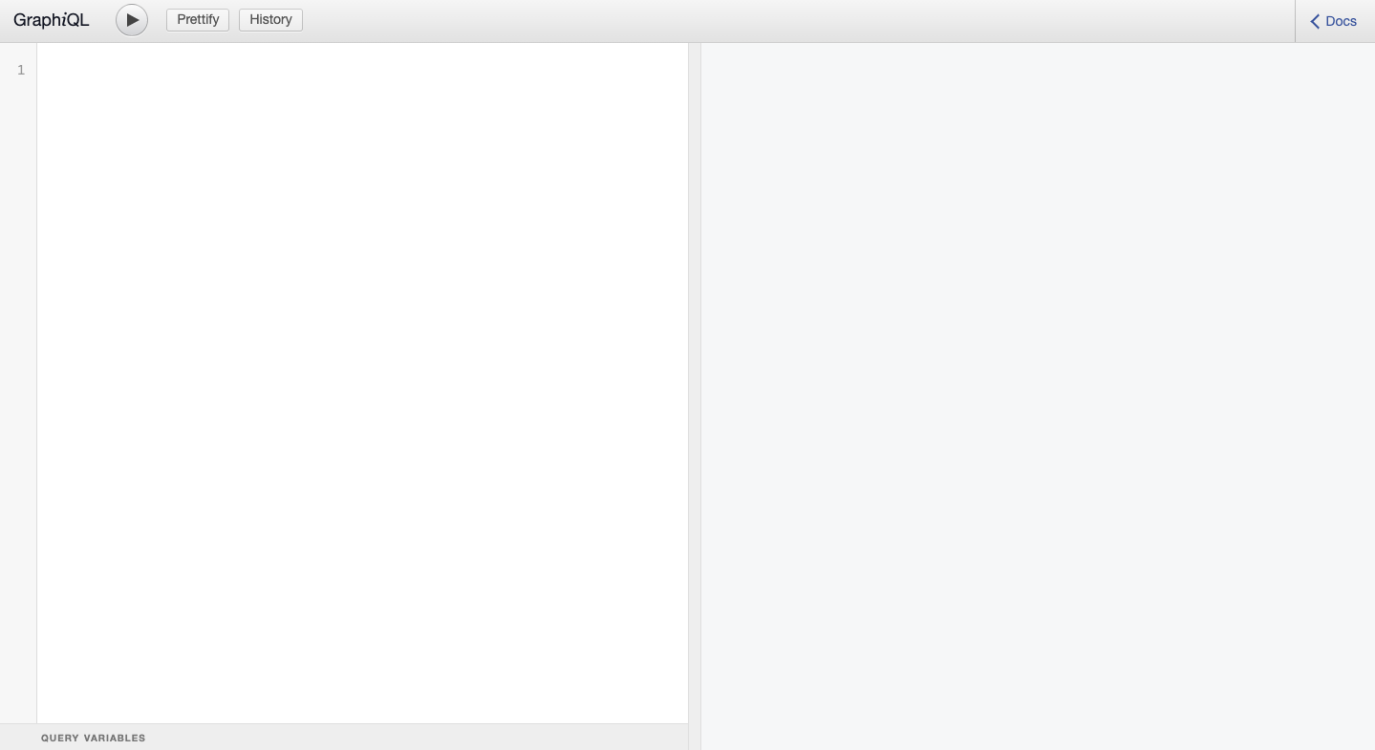
onPreGraphiQL: function\* (ctx) { },

};

配置完成之后，每个落到 /graphql的请求都会触发 GraphQL Schema 的查询。

## 启动服务，访问开发者工具

1. npm run dev
2. ip:端口/graphql



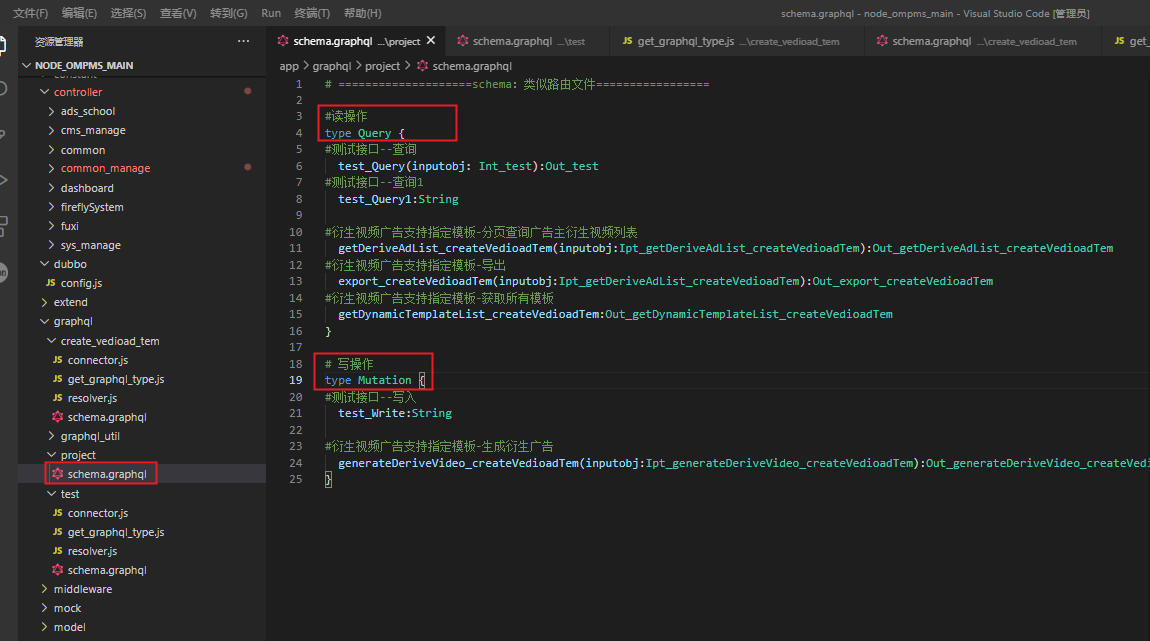
## 编写graphql代码

## **目录结构**

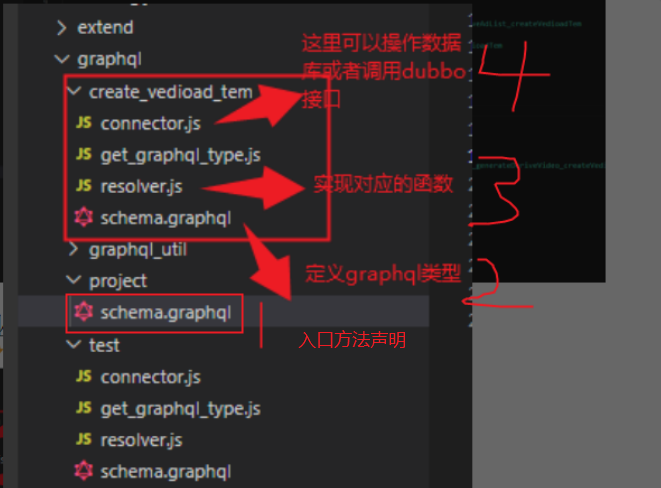


## **使用流程**

1. 先在 app\graphql\project\schema.graphql 下构建入口方法（类似于rest的路由函数）



1. 在对应的业务模块里面写下相关的文件



## **node处理前端传来的参数**

### 定义参数对象类型

使用input 关键字进行定义： node要接收的参数对象 （基本对象不需要定义）

# 用户输入的相关参数

input User{

#姓名

name: String!

#性别

sex: String

#年龄

age: Int

#爱好

hobby: [String]

}

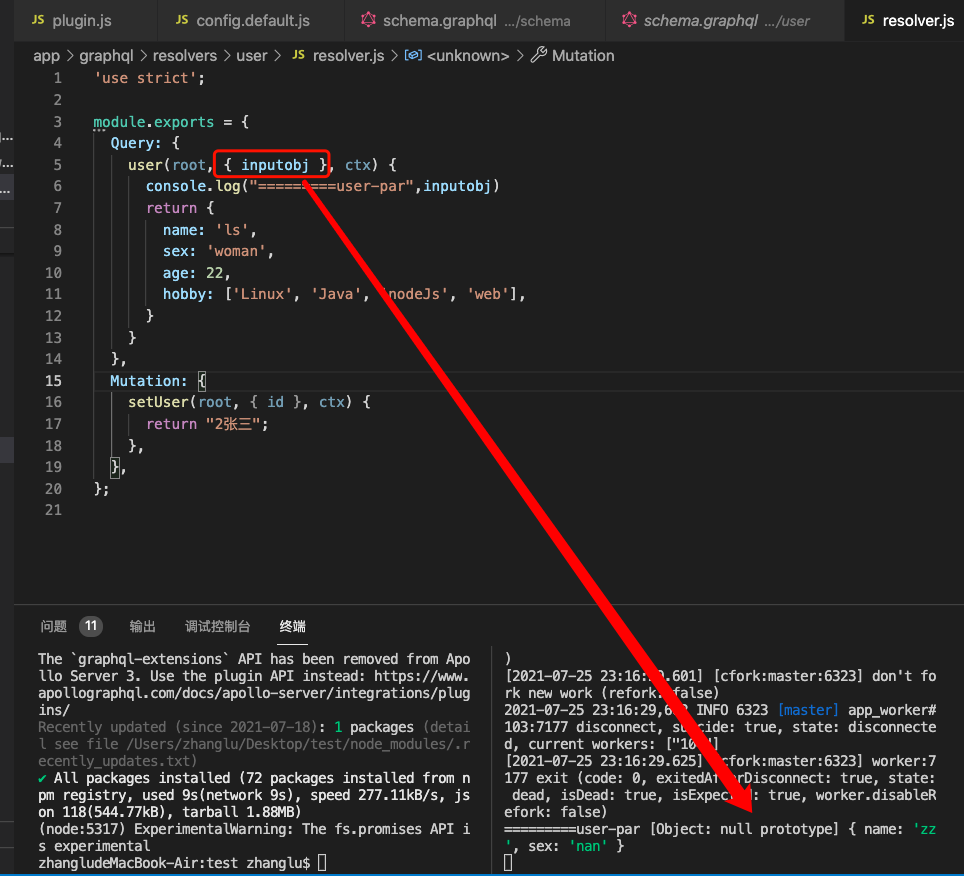
### 函数声明，定义形参类型

inputobj:User

#查询用户信息

user(inputobj:User): UserInfo

### 在resolver中获取参数



user(root, { inputobj }, ctx) {

console.log("=========user-par",inputobj)

return {

name: 'ls',

sex: 'woman',

age: 22,

hobby: ['Linux', 'Java', 'nodeJs', 'web'],

}

}

## **调用dubbo服务**

前端触发请求---》schemal--->resolver--->connector(这里发起dubbo服务调用+逻辑处理 或者 操作数据库（dataloader）)

## **一些注意事项**

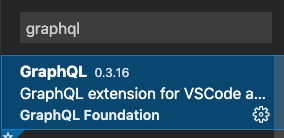
1. 项目中只能定义一个query和mutation,多了就会报错
2. 在基础配置是否设置默认的Query和Mutation, 需要设置为关闭（默认也是关闭，否则无法显式的设置query和mutation）

//是否设置默认的Query和Mutation, 默认关闭

defaultEmptySchema: false,

1. 安装vscode插件对代码进行颜色高亮

graphql



1. 很重要的一点：query和 mutation 中的方法和 我们的自定义类型，尽量添加文件名作为业务模块名\_作为前缀，以防名字冲突而报错
2. Graphql类型字段名不能是中杠线分割的

## 兼容rest接口权限

这里写了个路径进行转发，即前端可以配置任何的接口路径，只有格式为

/graphql/\*\*\* ，

都会被重定向到真正的处理函数上去





## 一些参考

npm: <https://www.npmjs.com/package/egg-graphql>

github:<https://github.com/eggjs/egg-graphql/>