🡪

GraphQL有内置数据类型：

|  |  |
| --- | --- |
| 名字 | 描述 |
| ID | 代表唯一的ID识别符号 |
| String | 代表字符串 |
| Int | 代表整型数据类型 |
| Float | 代表浮点数数据类型 |
| Boolean | 代表布尔值数据类型 |

🡪

GraphQL定义Schema，在Schema中定义额外的数据类型，并且定义服务器端所支持的查询类型。

例如：

定义额外的数据类型：

type product {

id: ID!,

name: String!,

category: String!,

price: Float!

}

上述定义了名为product的自定义数据类型。其中有id, name, category, price数据域。

id数据域对应类型为内置数据类型ID,感叹号! 表示该域为必须项。

name数据域对应类型为内置数据类型String, 感叹号! 表示该域为必须项。

type supplier {

id: ID!,

name: String!,

city: String!,

products: [ID]

}

上述定义了名为supplier的自定义数据类型。其中有id, name, city, products数据域。

id数据域对应类型为内置数据类型ID,! 表示该域为必须项。

name数据域对应类型为内置数据类型String,! 表示该域为必须项。

city数据域对应类型为内置数据类型String,! 表示该域为必须项。

products数据域对应类型为数组，数组中的元素类型为ID.

type Query {

products: [product],

suppliers: [supplier]

}

定义了Query类型，表示GraphQL支持什么类型的查询。上述表示GraphQL支持两种查询类型。products查询将返回一个数组，数组元素为product类型。Suppliers查询将返回一个数组, 数组元素为supplier类型。

🡪

Resolver用于定义查询数据逻辑，并且返回。

每一个resolver是一个方法，每个resolver应付一种查询。在方法中定义查询数据的逻辑，并且返回在schema中规定该查询返回的结果格式。

例如：

var data = require (“../../restData”) ()

module.exports = {

products: () => data.products, //规定名为products的resolver，应付schema中对应的products查询。返回schema中规定的返回格式[product]

suppliers: () => data.suppliers

}

🡪

使用GraphQL查询语句向GraphQL服务器查询，返回需要的数据。

每个查询语句需要表明查询类型，该查询类型需要在schema中type Query预先定义。并且在查询语句中申明，需要返回哪些域,并且申明返回域的顺序便为这些域在返回数据中的顺序.

例如：

query { //query表明该语句是查询，而非修改。

suppliers: { //suppliers表明查询的类型为suppliers

id, //id,name,city,products,表明此查询想要获取的域。并且返回数据的域的排队顺序便为id, name,

city,

products

}

}

🡪

如果GraphQL查询语句中，所查找的某个域对应的自定义的复杂类型。那么还需要在查询语句中返回该复杂类型中的某些域。

例如Schema中定义查询：

type product {

id: ID!,

name: String!,

category: String!,

price: Float!

}

type Query {

products: [product],

suppliers: [supplier]

}

情况1:

当查询语句为suppliers类型时候，将会返回supplier数组，如果supplier定义为：

type supplier {

id: ID!,

name: String!,

city: String!,

products: [ID]

}

由于supplier中所有域为GraphQL检查内置类型，则查询语句为：

query {

suppliers: {

id,

name,

city,

products

}

}

情况2：

当查询语句为suppliers类型时候，将会返回supplier数组，如果supplier定义为：

type supplier {

id: ID!,

name: String!,

city: String!,

products: [product]

}

由于supplier中域products的类型为product复杂类型。所以在查询语句中还要指定返回product的具体哪些域：

query {

suppliers{

id,

name,

city,

products {

name

}

}

}

🡪

GraphQL查询可以带有参数。参数的添加主要应用于两个方面，第一个是Query中带有参数，第二个是返回的数据域中带有参数。

***第一，为Query指定参数***：

为了让GraphQL支持带有参数的Query，我们需要在Schema定义的查询中定义该查询，并且该查询中定义该参数。在resolver中定义该查询接收参数以后怎么获取对象。在未来的GraphQL查询语句中指定该参数的值。

**在Schema中**：

type Query {  
 products: [product],  
 product(id: ID!): product, //定义查询，并且以()指明该参数以及参数类型，返回值  
 suppliers: [supplier],  
 supplier(id: ID!): supplier //定义查询，并且以()指明该参数以及参数类型，返回值  
}

**在resolver中**：

我们已经在Schema中定义Query product, supplier将必须带有参数id. 在resolver中，该参数将放在args对象当中。然后根据Schema中规定需要返回的数据格式，返回对应类型的内容。

module.exports = {  
 …….  
 product: (args) => data.products.find(p => p.id === parseInt(args.id)),  
 supplier: (args) => {  
 const result = data.suppliers.find(s => s.id === parseInt(args.id));  
 if(result){  
 return {  
 ...result,  
 products: () => result.products.map(id =>data.products.find(p => p.id === ***Number***(id))) //此处定义一个箭头匿名函数，当需要获取products时候便会调用  
 }  
 }  
 }

…..  
}

**在GraphQL查询语句当中：**

query {

supplier(id:1){

id,

name,

city,

products{

name

}

}

}

在GraphQL查询语句中，在查询类型supplier后加入(参数名：参数值)填入所需的参数。

***第二，为域添加参数***：

当我们对某个类型的域添加参数后，所有返回该Query类型的结果中的该域都可以应用该参数来过滤结果。

例如：

**在Schema中：**

在Schema当中定义域接收参数，products(nameFilter: String= “”)表示该域接受一个类型为String名为nameFilter的参数。从而所有返回结果为supplier类型的结果中，都可以使用该参数对products域返回的结果进行过滤。并且由于没有!修饰，所以该参数nameFilter为非必要，在没有传入该参数的时候默认使用””作为其参数值。

type supplier {  
 id: ID!,  
 name: String!,  
 city: String!,  
 products(nameFilter: String = ""): [product]  
}

type Query {  
 products: [product],  
 product(id: ID!): product,  
 suppliers: [supplier],  
 supplier(id: ID!): supplier  
}

**在Resolver中：**

对于supplier查询, 它将返回supplier类型。从而参数nameFilter可用于过滤其返回的supplier结果中的products域。在Schema中添加参数nameFilter以后，将会在products对应的方法中，获取该参数。

对于suppliers查询， 它将返回[suppliers]类型。每个元素都是supplier.所以该参数也可以用于过于suppliers查询结果中，每个元素supplier的products域的过滤。

supplier: ({id}) => {  
 const result = data.suppliers.find(s => s.id === parseInt(id));  
 if(result){  
 return {  
 ...result,  
 products: ({nameFilter}) => mapIdsToProducts(result, nameFilter)  
 }  
 }  
}

suppliers: () => data.suppliers.map(s => ({  
 ...s,  
 products: ({nameFilter}) => mapIdsToProducts(s, nameFilter)  
})),

**在GraphQL查询语句当中：**

query {

supplier(id:1){ //此处传入Query传入的参数

id,

name,

city,

products(nameFilter: "ak"){ //此处传入products域传入的参数

name

}

}

}

query {

suppliers{

id,

name,

city,

products(nameFilter: "g"){ //此处传入products域传入的参数

name

}

}

}

🡪

使用GraphQL改变数据(Mutation)

首先在schema中定义mutation:

type product {  
 id: ID!,  
 name: String!,  
 category: String!,  
 price: Float!  
}  
  
type supplier {  
 id: ID!,  
 name: String!,  
 city: String!,  
 products(nameFilter: String = ""): [product]  
}  
  
type Query {  
 products: [product],  
 product(id: ID!): product,  
 suppliers: [supplier],  
 supplier(id: ID!): supplier  
}  
  
input productInput{ //使用input关键字自定义名为productInput的input类型数据。在该数据中包含可选参数id类型为ID, 必需参数name类型为String, 必需参数category类型为String, 必需参数price类型为Int.  
 id:ID,  
 name: String!,  
 category: String!,  
 price: Int!  
}  
  
type Mutation{ //使用Mutation关键字自定义名为storeProduct以及storeSupplier的mutation。名为storeProduct的mutation需要接受自定义input参数类型productInput, 更新保存以后返回product类型数据。名为storeSupplier的mutation需要接受4个内置参数类型，id类型为ID, name类型为String, city类型为String, products类型为Int数组。更新保存以后返回supplier类型数据。  
 storeProduct(product: productInput): product  
 storeSupplier(id:ID, name: String!, city: String!, products: [Int]): supplier  
}

接下来需要在Resolvers中添加resolver去处理这些mutation请求：

let nextId = 100;  
  
module.exports = {  
 storeProduct({product}){//处理schema中定义的storeProduct mutation.使用解构赋值接受类型为productInput到变量product当中  
 if(product.id == null){//当接受的product不存在id时候，为添加。放入data.products中并且增加id  
 product.id = nextId++;  
 data.products.push(product)  
 }  
 else{ //当接受的product存在id时候，为更新。替换data.products中对应的原本product对象  
 product = {...product, id: Number(product.id)};  
 data.products.map(p => p.id === product.id ? product: p)  
 }  
 return product; //最后根据mutation在schema中的要求返回一个product对象  
 },  
  
 storeSupplier(args){ //处理schema中定义的storeSupplier mutation.使用解构赋值将变量保存到args对象当中

const supp = {...args, id: Number(args.id)};  
 if (supp.id == null){ //当接受的supp不存在id时候，为添加。放入data.suppliers中并且增加id  
 supp.id = nextId++;  
 data.suppliers.push(supp)  
 }  
 else{ //当接受的supp存在id时候，为更新。替换data.suppliers中对应的原本supplier对象  
 data.suppliers = data.suppliers.map(s => s.id === supp.id ? supp : s);  
 }  
 let result = data.suppliers.find(s => s.id === supp,id);  
 if (result){ //最后根据mutation在schema中的要求返回一个supplier对象,并且对于其中products将为[product]符合schema中的定义  
 return {  
 ...result,  
 products: ({nameFilter}) => mapIdsToProducts(result, nameFilter)  
 }  
 }  
 }  
}

最后在GraphQL查询语句当中：

mutation{ //表明为mutation请求  
 storeProduct(product:{ //请求的mutation具体为storeProduct,并且传入参数product  
 id: 1,  
 name: "Green Kayak",  
 category: "WaterSports",  
 price: 290  
 }){ //根据schema, storeProduct mutation以后将返回product,所以需要获取product的哪些域  
 id,  
 name,  
 category,  
 price  
 }  
}

🡪

**GraphQL额外特性**

1.使用$符号申明变量，并用于查询当中。

例如：

query($id: ID!){ //在GraphQL查询中使用$id定义变量id,类型为必需的ID类型。从而在下面query语句中，可以使用$id放入  
 product(id: $id){  
 id, name, category, price  
 }  
}

与此同时，我们需要指定$id变量的值：

{

"id": 2

}

2.GraphQL允许我们通过”,”分隔，单次提交多个查询。

例如：

query{  
 product(id: 1){ //查询类型为product用于查询id为1的product  
 id, name, category, price  
 },  
 supplier(id: 1){ //查询类型为supplier用于查询id为1的supplier  
 id, name, city  
 }  
}

返回结果为：

{

"data": {

"product": { // key为product表示此为product查询结果

"id": "1",

"name": "Kayak",

"category": "Watersports",

"price": 275

}, //使用,进行分隔多个查询

"supplier": { // key为supplier表示此为supplier查询结果

"id": "1",

"name": "Surf Dudes",

"city": "San Jose"

}

}

}

如果我们想一次提交多个同类型的查询，我们还可以指定别名来在返回结果中加以区分。

例如：

query{

first: product(id: 1){ //指定别名为first,在返回结果中以first标识

id, name, category, price

},

second: product(id: 2){ //指定别名为second,在返回结果中以second标识

id, name, category, price

}

}

结果：

{

"data": {

"first": {

"id": "1",

"name": "Kayak",

"category": "Watersports",

"price": 275

},

"second": {

"id": "2",

"name": "Lifejacket",

"category": "Watersports",

"price": 48.95

}

}

}

3. 定义片段(Fragment)。GraphQL支持通过定义分片来统一指代某几个数据域。从而能在需要经常获取这几个域的查询中，不需要重复编写。

例如：

fragment coreFields on product{ //fragment关键字创建名为coreFields的fragment,并且on表示该fragment用于具体哪个数据类型上

id, name, category

}

query{  
 first: product(id: 1){  
 ...coreFields, //使用spread operator来应用fragment统一指定获取fragment所对应的id, name, category域。从而不需要在这两个查询中重复编写  
 price  
 },  
 second: product(id: 2){  
 ...coreFields  
 }  
}