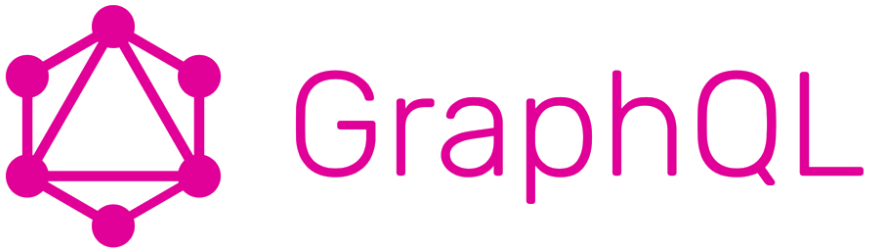
https://ithelp.ithome.com.tw/articles/10204863

GraphQL 入门： 深度解析 Field Resolver 的参数: (parent, args, context)



在 Resolver field 时，预设传进来的参数无疑是非常强大的帮手，但要搞懂它并不容易。

我自己在刚开始学习时，因为 JS 不会强制规定参数名称，所以每个教学文章使用的命名皆不尽相同，搞得明明一样的东西你要兜一大圈才知道他们是一样的。因此这边我会使用与**语意**较为接近的命名方法，分别为

1. parent 为此 field 上一层的资料。
2. args 为 client side 的 query 对此项 fied 传进来的参数。
3. context 类似于全局变量的概念，在单一 query 内所有 field 都可享用的资源，通常会放些 user 数据或 ORM 方法。

开始前可以参考 [Prisma] 部落格里面一张超赞的解说图：

1. Parent 让你找到前一世 (层)

首先是 `parent`，最直白的意义就是上一层的数据，至于什么是上一层呢？只需要看你的 field 是属于哪个 Object Type 底下，你的 parent 就是该 Object Type 的数据，而如果是最上层的 Query field，我通常会把 `parent` 命名为 `root`，而 `root` 的值除非有特别设定不然都会是 `null`。

Parent 范例

- Schema

```
type User {
  id
  name
  age
  friends
}

type Query {
  me: User
}
```

- Resolver

```
const resolvers = {
  Query: {
    me: (root, args, context) => ({
      // 这里 root, args 都是 null
      id: 1,
```

```
      name: 'Fong',
      age: 23,
      posts: [2, 3]
    })
  },
  User: {
    id: (parent, args, context) => parent.id,
    name: (parent, args, context) => parent.name,
    age: (parent, args, context) => parent.age,
    posts: (parent, args, context) => {
      /* db operation to get posts by userId */
    }
  }
};
```

以上 Resover 中 `User` 的三个 field resolver function 的 `parent` 值都是一样的。

`parent` 的妙用在于如果你需要特别处理某些 field ，假设是要对 age 添加额外数据 (ex: 搭配参数计算、依用户权力调整值) 等等，就可以使用 `parent.age` 取得原值再做计算 ；又假设你是想要得到其他同样也是 `User` Type 的 `posts` 数据，但 db 捞出的原始资料与 schema 不合或甚至根本没这项值，就可以用 `parent.id` 取得 user id 再去 db 捞相关的贴文 (post) 资料出来。

但对于初学者需注意的是，**并不是 Schema 有定义的 field 就一定会出现在 `parent` 的里面，重点是要看你 如何实作 Resolver**。

之前说过 `Query` 为整个 Schema 的 entry point 。所以如果是因为前端 query 到 `me` 这个 field 而进入 `User` 的 field resolver ，那里面的 `parent` 就得要看 `me` 的 Field Resolver 传了什么东西回去。

不同的 entry point 可能会造成 `parent` 值的不同，所以应该透过良好的规范来避免不确定的 `parent` 值。

Parent 在 Relational Database 的注意事项

如果是使用 Relational Database (关系数据库) 的朋友需要注意， Field Resolver 只能实作一层的
Resolver，没有 Nested Field Resolver 这件事，因此也不会有 `parent.parent` 的存在 ！因此也不会有以下的程序出现：

```
const resolver = {
  User: {
    posts: {
      title: () => 'Can Nested FieldResolver Works?'
    }
  }
};
```

所以当你想新增的 Object Type (ex: `Post` Type) 在 Database Table (ex: post table) 有 Foriegn Key (ex: post.authorId) 时，最好放在 Foriegn Key (ex: `authorId`) 指向的 Object Type (ex: `User` Type) 的第一层。

EX: 你今天想要新增贴文 post 功能，而 post 的数据会存 `authorId` 来指向 user ，这时候如果你不知道为何没把它放在第一层如下：

```
type Post { ... }



type Blog {

  "贴文"

  posts: [Post]

  "贴文数"

  postCount: Int

  "观看数"

  viewCount: Int

}



type User {

  ...

  blog: Blog

}
```

加上如果你的 Resolver 如下方程序这样做，可能就会拿不到贴文:

```
const resolver = {
  Blog: {
    posts: (parent, args, context) {
      // 此时的 parent 就是 blog 而非 user 数据，因此会没有 userId 可以取得
    }
  }
}
```

可以看到这时候 `Blog.posts` 就无法依靠 user id 来取得贴文 (post)，所以要不一开始就放第一层，要不就在设计贴文的 post table 时就将 blog id 也加进 Foreign Key.

经验谈：公司一开始开发时其实并不太会用 Field Resolver，因此就傻傻的将有 Foreign Key 对应到的 Object Type 给塞到了深处，结果在开始了解 Field Resolver 的妙用后却有很多因为拿不到上一层需要的数据而难以实作。当然在 GraphQL 有一些比较成熟的 Design Pattern 如 Pagination 为了处理大量数据分页因此会将数据往里层封装，但除非你有很好的理解与适当的实作，不然不要轻易将数据往内层塞。

2. Args 处理参数一把罩

这里的 `args` 概念很简单，就是取得 query 或是 mutation 传进来的参数，如果是使用 JS 的朋友我会建议直接在第一行就 deconstruct (解构) 掉，变成：

```
(parent, { args1, args2 }, context)
```

如果只是一个个参数排下来还好，但当加入 Input Object Type 就很容易造成错误的 deconstruct 。 因为 Input Object Type 格式上是 Object 形式 ，
所以如果 Schema 如下

```
input AddPostInput {
  title: String!
  body: String
}

type Mutation {
  addPost: (input: AddPostInput!): Post
}
```

需注意 `input` 里面还有一层，所以 resolver 使用时需再多 deconstruct 一层 (我当初常常忘记多 deconstruct 一层导致 bug 一直爆出来而且又超难除错)

想知道更多参数如何命名的学问，可推荐参考 Github GraphQL API Explorer，可以从里面可以推论出很多 pattern ，如

1. 凡是 Query field 需要参数的，皆使用一个或多个 Scalar Type 或 Enum Type (都有实际值)。
2. 凡是 Mutation field 需要参数的，参数列一律只放一个 `input` 并有该 mutation 专属的 Input Objcet Type ，连回复的 Object Type 也是专属的。

3. Context 让数据/功能一脉相传！

接下来是今天的重头戏 `context` 。

很多人会想说在 GraphQL 怎么认证使用者 ？

答案是就通通放在 `context` 里 ！尽管去拿吧 ！

以 token-based authentication 来说， Client Side 发出一支 `login` mutation 来， Server side 会给 Client Side 一个 token (可参考 JWT token)，而下次 Client Side 发 query 或 mutation 来时就把 token 夹进 header 里，这时 Server Side 接收到并开始解析 token ，如果成功从中解析出 user 数据，就将 user 数据塞进 context ，确保让接下来一层层的 Field Resolver 是使用同一个 user 数据。

除了 token 解析成功的 user 数据， `context` 还可以放入一些机密的 secret 、环境变量等等，而这些资料可以传入 authorization 、 business logic 等 layer 使用，如官方教学的下图：

另外 `context` 还有另一个妙用：放入 ORM 或是其他 db operation 的 function 。如下：

```
const resolvers = {

  me: (parent, args, { user, UserModel }) => {

    return UserModel.findById(user.id);

  }

};
```

这样的好处是减少管理外部引入的 dependency ，此外 GraphQL 在做 cache 或是 batching 等效能提升时，能将 function 下去会相当有用! (之后会介绍）)

`context` 在 Apollo Server 的设定

可以参考 Apollo Server 官方教学

。

```
const typeDefs = gql`
```

```
  type Author {
    name
  }
`;

new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    const token = req.header['x-token'];
    const user = jwt.verify(token);
    return {
      me: user,
      userModel
      // ...others like db models, env, secret, ...
    };
  }
});
```

有了这一层，当我登入成功得到 token 后使用 `addPost` mutation (记得 token 要放 header) 进入 GraphQL resolver 时，我们可以看到 `context` 里面带有 `me` 这项 user 数据。

```
const resolvers = {
```

```
  Mutation: {
    addPost: (root, { input: { title, body } }, context) => {
      const { me, userModel } = context;
      return userModel.create({
        title,
        body,
        authorId: me.id
      });
    }
  }
};
```

可以发现其实 context 其实就是一个 middleware ，只是藉由参数传进 Apollo Server 会自动帮你设定好。

隐藏的第四人 info

但其实，还有一个低调的第四人 `info`，里面主要存取一些 GraphQL 的 AST 数据结构及执行料，但基本上完全不会用到，连 GraphQL 官方文件都没有说里面在搞什么鬼，因此真的不需费心。

有兴趣的可以看 Prisma 的这篇文：

https://www.prisma.io/blog/graphql-server-basics-demystifying-the-info-argument-in-graphql-resolvers-6f26249f613a/

原本今天想要直接实作一个 server ，但发现还有一些观念需要先建立，这样实作过程若有不懂的也可以往前找到解答。

Reference

- https://www.robinwieruch.de/graphql-apollo-server-tutorial/#apollo-server-authentication