

官网地址

1 GraphQL简介

1.1 来源

Google在2012年开始在移动应用上使用GraphQL，并且2015年将GraphQL开源供大家使用。

1.2 官网介绍

1.3 体验一下

在学习之前呢，我们先使用GraphiQL的体验一下GraphQL所带来的强大功能。

1.4 总结

通过这一小节，想必大家对GraphQL是什么、以及能做什么也有一定的了解了，如果对RESEful Api有一定了解的话，那么对于体会GraphQL对功能就会更加深刻。

2 pwzos

pwzos里面存放了这套教程所有的文档、实例代码等相关资源...

3 起步

这套教程采用C#语言，在学习基础知识阶段使用最简单的控制台程序编写练习案例。当然GraphQL最终的使用还是要作为服务器，但是使用控制台学习知识点是完全足够的，我这样中的目的是要将学习依赖项减少到最低，让学习门槛较低。教程后期会采用Asp.Net Core框架来提升整套教程的难度，并将在控制台程序阶段所学习的所有知识点串连起来。最终让大家熟练掌握GraphQL.Net。

- GraphType：GraphQL中的类型，包含GraphQL核心规范中定义的类型和自定义的类型
- Query：GraphQL所提供的操作类型之一，用于数据查询
- Mutation：GraphQL所提供的操作类型之一，用于数据修改
- Subscription：GraphQL所提供的操作类型之一，用于信息推送
- Schema：GraphQL整体的结构定义

3.1 最简单的控制台实现

3.1.1 依赖包

- Install-Package GraphQL

3.1.2 项目地址

GraphQL_3_1

3.1.3 Entity

- model属于C#的模型，他是GraphQL与C#之间桥梁。C#方法通过model来输入/输出数据，GraphQL引擎使用model与Type之间进行转换

```
1    /// <summary>
2    /// 人
```

```

3      /// </summary>
4      public class Persons
5      {
6          /// <summary>
7          /// 编号
8          /// </summary>
9          public int Id { get; set; }
10         /// <summary>
11         /// 姓名
12         /// </summary>
13         public string Name { get; set; }
14         /// <summary>
15         /// 年龄
16         /// </summary>
17         public int Age { get; set; }
18         /// <summary>
19         /// 地址
20         /// </summary>
21         public string Addr { get; set; }
22     }

```

3.1.4 type

- type是GraphQL中的类型，在GraphQL操作中，都是围绕他展开的。在创建的时候和Model关联起来

```

1      public class PersonsType:ObjectGraphType<Persons>
2      {
3          public PersonsType()
4          {
5              Name = "persons";
6              Description = "人";
7              Field(a => a.Id).Description("编号");
8              Field(a => a.Name).Description("姓名");
9              Field(a => a.Age).Description("年龄");
10             Field(a => a.Addr).Description("地址");
11         }
12     }

```

3.1.5 Query

- GraphQL的操作之一，通过这种方式，可以实现GraphQL类型的访问

```

1      public class RootQueries:ObjectGraphType
2      {
3          public RootQueries()
4          {
5              Name = "rootQueries";
6              Description = "根查询";
7              Field<PersonsType>("queryPerson", resolve: context => new Person
8          }
9      }

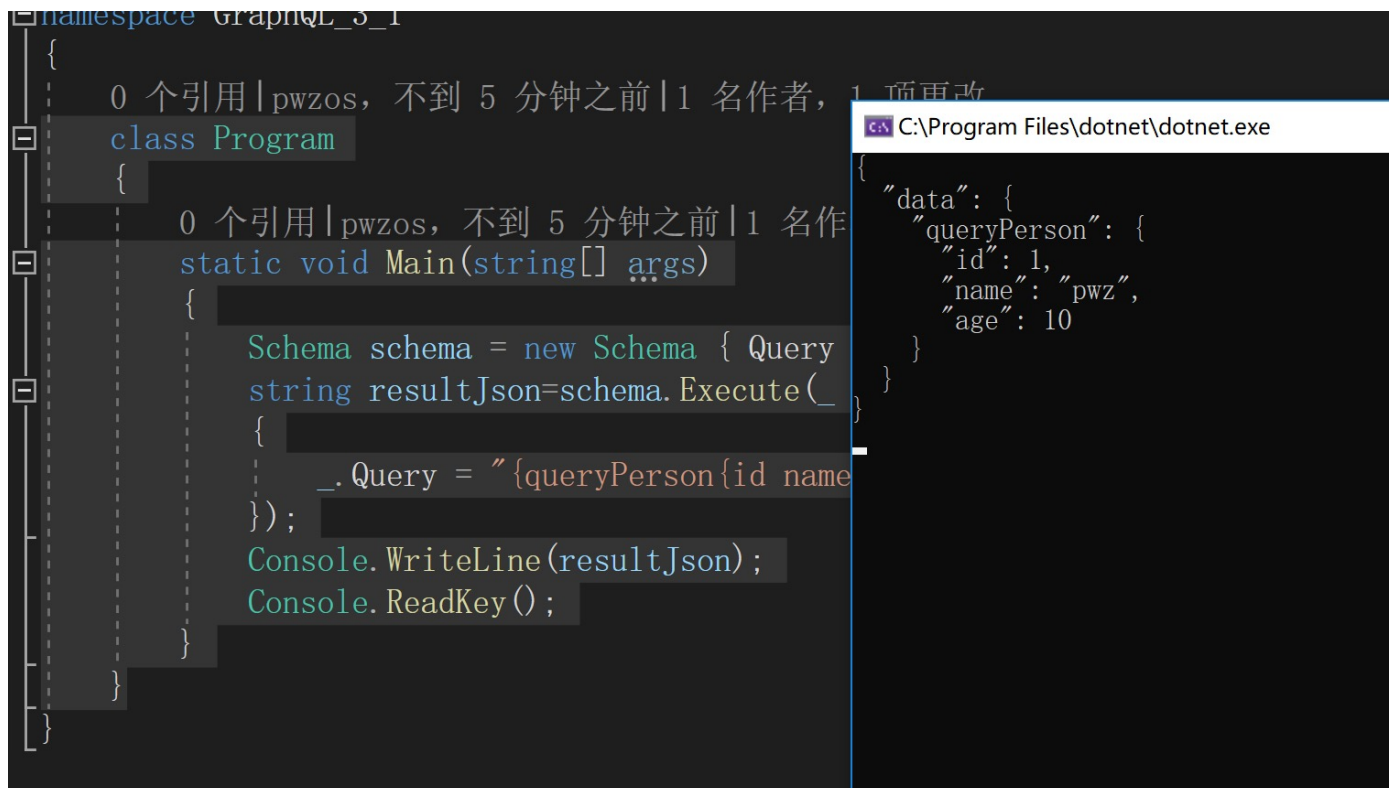
```

3.1.6 Console

- 访问GraphQL

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Schema schema = new Schema { Query = new RootQueries() };
6          string resultJson=schema.Execute(_ =>
7          {
8              _.Query = "{queryPerson{id name age}}";
9          });
10         Console.WriteLine(resultJson);
11         Console.ReadKey();
12     }
13 }
```

3.1.7 结果



The screenshot shows a Visual Studio editor with a C# file named `GraphQL_3_1.cs`. The code defines a `Program` class with a `Main` method that creates a `Schema` and executes a GraphQL query. The query is `{queryPerson{id name age}}`. The output is displayed in the console window as a JSON object.

```
namespace GraphQL_3_1
{
    class Program
    {
        static void Main(string[] args)
        {
            Schema schema = new Schema { Query = new RootQueries() };
            string resultJson=schema.Execute(_ =>
            {
                _.Query = "{queryPerson{id name age}}";
            });
            Console.WriteLine(resultJson);
            Console.ReadKey();
        }
    }
}
```

```
{
  "data": {
    "queryPerson": {
      "id": 1,
      "name": "pwz",
      "age": 10
    }
  }
}
```

4 Queries

GraphQL规范中定义的一种操作类型，在GraphQL中查询能够正确执行查询，必须要有一个根查询对象，

值得注意：query是并行执行的

4.1 查询举例

- 完整的查询

```
1
2 query{
3     queryPerson
4     {
5         id
6         name
7         age
8     }
9 }
10
```

- 简写形式

```
1
2     queryPerson
3     {
4         id
5         name
6         age
7     }
8
```

- 指定查询名

通过在query关键字后面指定关键字，可以为特定查询命名，如果当前请求只有一个操作，那么这个被命令的查询会被自动执行；如果有多个，可以选择指定查询查询名执行。

```
1 query queryStudent{
2     student{
3         id
4         name
5     }
6 }
```

5 Schema Types

5.1 标量（规范提供）

通常一个GraphQL对象由名字和字段（字段可能也是一个对象）组成，但是在有些情况下有些字段不得不在GraphQL规范中提供一些标量，我们自己可以在这些标量的基础进行扩展

GraphQL	GraphQL.Net	.Net
String	StringGraphType	String
Int	IntGraphType	int long

GraphQL	GraphQL.Net	.Net
Fload	FloatGraphType	double
Boolean	BooleanGraphType	bool
ID	IdGraphType	int、long、string

值得注意：**ID**是使用**Guid**生成，并且需要序列化为字符串类型，当你发送到服务器时也必须是字符串

5.2 标量（GraphQL.Net提供）

GraphQL.Net提供了如下额外的标量

GraphQL	GraphQL.NET	.NET
Date	DateGraphType	DateTime
DateTime	DateTimeGraphType	DateTime
DateTimeOffset	DateTimeOffsetGraphType	DateTimeOffset
Seconds	TimeSpanSecondsGraphType	TimeSpan
Milliseconds	TimeSpanMillisecondsGraphType	TimeSpan

列表中的数据项支持所有的标量类型

GraphQL	GraphQL.Net	.Net
[String]	ListGraphType	List<String>

5.3 对象

对象是由标量和其他对象组成

5.3.1 GraphQL

```
1 type Students
2 {
3     id:ID
4     name:String
5     age:Int
6 }
```

5.3.2 GraphQL .Net

```
1 /// <summary>
2 /// GraphQL中的学生类型
3 /// </summary>
```

```

4      public class StudentsType : ObjectGraphType<Students>
5      {
6          public StudentsType()
7          {
8              Name = "StudentsType";//类型名称
9              Description = "这是学生类型";//类型描述
10             #region 学生类型下的字段
11             Field(a => a.Id).Description("学生编号");
12             Field(a => a.Name).Description("学生姓名");
13             Field(a => a.Age).Description("学生年龄");
14             #endregion
15         }
16     }

```

5.3.3 .Net

```

1      /// <summary>
2      /// 学生
3      /// </summary>
4      public class Students
5      {
6          /// <summary>
7          /// 学生编号
8          /// </summary>
9          public int Id { get; set; }
10         /// <summary>
11         /// 年龄
12         /// </summary>
13         public int Age { get; set; }
14         /// <summary>
15         /// 姓名
16         /// </summary>
17         public string Name { get; set; }
18     }

```

5.4 枚举

5.4.1 GraphQL

```

1  enum Season
2  {
3      SPRING
4      SUMMER
5      AUTUMN
6      WINTER
7  }

```

5.4.2 GraphQL .Net

```

1  /// <summary>
2  /// 季节
3  /// </summary>
4  public class SeasonType:EnumerationGraphType
5  {
6      public SeasonType()
7      {
8          Name = "季节枚举";
9          Description = "季节的枚举";
10         AddValue("SPRING", "春", 0);
11         AddValue("SUMMER", "夏", 1);
12         AddValue("AUTUMN", "秋", 2);
13         AddValue("WINTER", "冬", 3);
14     }
15 }
16

```

or

```

1  /// <summary>
2  /// 季节
3  /// </summary>
4  public class SeasonType:EnumerationGraphType<Season>
5  {
6      public SeasonType()
7      {
8          Name = "季节枚举";
9          Description = "季节的枚举";
10         AddValue(Season.SPRING.ToString(), "春", 0);
11         AddValue(Season.SUMMER.ToString(), "夏", 1);
12         AddValue(Season.AUTUMN.ToString(), "秋", 2);
13         AddValue(Season.WINTER.ToString(), "冬", 3);
14     }
15 }

```

5.4.3 .Net

```

1  /// <summary>
2  /// 季节
3  /// </summary>
4  public enum Season
5  {
6      SPRING=0,
7      SUMMER=1,
8      AUTUMN=2,
9      WINTER=3
10 }

```

5.5 非空修饰符和集合

通过使用"!"修饰符来对GraphQL中所支持对类型进行修饰，那么这个类型的值就被限制为不允许为空。


```

1 type Students{
2     id:ID!
3     name:String!
4     age:Int!
5 }

```

当我们在一个类型后面使用不允许为空的修饰符后，那么这个类型所返回的值就不允许为null，如果为null，那么返回null。当我们参数后面使用不允许为空的修饰符后，如果这个参数为null，无论这个参数是在字符串还是在变量中，那么返回null。

5.5.1 如下是GraphQL列表的验证

- [String!] 列表项不允许为null，列表允许为null
- [String]! 列表项允许为null，列表不允许为null
- [String!]! 列表项和列表都不允许为null

6 参数

查询、修改和推送在很多情况下都需要使用参数，那么怎么传递参数？后台怎么获取参数呢？其实是很简单的。

6.1 项目地址

[GraphQL_6_1](#)

6.2 Query

```

1 public class RootQueries:ObjectGraphType
2 {
3     public RootQueries()
4     {
5         Name = "rootQueries";
6         Description = "根查询";
7         Field<PersonsType>("queryPerson",
8             arguments: new QueryArguments(new QueryArgument<StringGraphType>
9             {
10                 Name = "name",
11                 Resolve = context =>
12                 {
13                     string name = context.GetArgument<string>("name");
14                     Console.WriteLine($"我的名字是: {name}");
15                     return new Persons { Id = 1, Name = "pwz", Age = 10, Add
16                 }
17             }
18     }
19 }

```

6.3 Console

```

1 class Program
2 {

```

```

3      static void Main(string[] args)
4      {
5          #region 第一种传参数的方式
6          /*
7              Schema schema = new Schema { Query = new RootQueries() };
8              string resultJson=schema.Execute(_ =>
9              {
10                  _.Query = "{queryPerson(name:\"pwz\"){id name age}}";
11              });
12              Console.WriteLine(resultJson);
13              Console.ReadKey();
14          */
15          #endregion
16          #region 第二中传参数的方式
17          string variableJson = "{name:'pwz'}";
18          Inputs inputs = variableJson.ToInputs();
19          Schema schema = new Schema { Query = new RootQueries() };
20          string resultJson = schema.Execute(_ =>
21          {
22              _.Query = "query rootQueries($name:String){queryPerson(name:
23                  _.Inputs = inputs;
24              });
25              Console.WriteLine(resultJson);
26              Console.ReadKey();
27          #endregion
28      }
29  }

```

6.4 结果

The screenshot shows a code editor with the following code:

```

{
    _.Query = "{queryPerson(name:\"pwz\"){id name age}}";
});
Console.WriteLine(resultJson);
Console.ReadKey();
*/
#endregion
#region 第二中传参数的方式
string variableJson = "{name:'pwz'}";
Inputs inputs = variableJson.ToInputs();
Schema schema = new Schema { Query = new RootQueries() };
string resultJson = schema.Execute(_ =>
{
    _.Query = "query rootQueries($name:String){queryPerson(name:$name){id name age}}";
    _.Inputs = inputs;
});
Console.WriteLine(resultJson);
Console.ReadKey();
#endregion
}

```

The console window (C:\Program Files\dotnet\dotnet.exe) shows the output:

```

我的名字是: pwz
{
  "data": {
    "queryPerson": {
      "id": 1,
      "name": "pwz",
      "age": 10
    }
  }
}

```

7 别名

对于同一个类型的操作，如果查询内容不同的话，我们可以使用别名对该操作类型进行重命名，具体形式如下
(myQueryStudent*就是queryStudent查询字段的别名)

```
1  {
2      myQueryStudent1:queryStudent
3      {
4          id
5          name
6          age
7      }
8      myQueryStudent2:queryStudent
9      {
10         id
11         name
12         age
13     }
14 }
```

8 片段

通过片段，我们能够构造一个查询字段，以实现这个片段能够被复用。这个字段里面包含了我们所需要的一些查询。具体格式如下：

```
1  query
2  {
3      queryStudent
4      {
5          ...student
6      }
7  }
8  fragment student on StudentsType
9  {
10     id
11     name
12 }
```

9 Variables

通过"\$"来定义变量，变量是一个json对象，\$id:Int是定义变量的名称和类型，d:\$id是使用变量具体格式如下：

json

```
1  query QueryStudentById($id:Int!){
2      queryStudentById(id:$id)
3      {
```

```

4         id
5         name
6         addr
7     }
8 }
9
10
11 variables
12
13 {
14     id:1
15 }

```

C#

```

1     class Program
2     {
3         static void Main(string[] args)
4         {
5             string variableJosnStr = "{id:2}"; //初始化
6             Inputs inputs = variableJosnStr.ToInputs(); //调用扩展方法将其转换为I
7             var schema = new Schema { Query = new QueryStudents() }; //定义一个
8             var json = schema.Execute(_ => { _.Query = ("query queryStudents
9             System.Console.WriteLine(json);
10            System.Console.ReadKey();
11        }
12    }

```

10 指令

GraphQL核心规范中包含了两个指令（@include(if:Boolean)、@skip(if:Boolean)），指令中是查询字段或者片段。这些指令能够在服务器支持的任何查询中生效。

- @include(if:Boolean) 只有当参数为true时才会查询指令所包含的字段
- @skip(if:Boolean) 当参数为true是，会直接跳过指令所包含的字段

10.1 json

```

1 query QueryStudent($isQueryName:Boolean!)
2 {
3     queryStudent
4     {
5         id
6         name @include(if:$isQueryName) //如果name是一个对象，那么可以在指令后面加J
7     }
8 }
9 variables
10 {
11

```

```
12 |         isQueryName:false
    |     }
    | }
```

10.2 C#

```
1 |     class Program
2 |     {
3 |         static void Main(string[] args)
4 |         {
5 |             string variableJosnStr = "{isQueryName:false}";
6 |             Inputs inputs = variableJosnStr.ToInputs();
7 |             var schema = new Schema { Query = new QueryStudents() };//定义一个Schema
8 |             var json = schema.Execute(_ => { _.Query = ("query QueryStudent(
9 |             System.Console.WriteLine(json);
10 |            System.Console.ReadKey();
11 |        }
12 |    }
```

11 Mutations

与查询类型，要正确创建一个修改操作类型，那么必须要有一个修改根，并且只能有一个。在GraphQL.Net中，一个修改操作能够修改指定数据并且返回一个结果。

值得注意：修改操作是串行执行的

修改操作需要使用mutation关键字，与查询类似，当只有一个请求操作的时候，可以省略操作名称。

11.1 json

```
1 | mutation ($student:StudentInput!)
2 | {
3 |     addStudent(student:$student)
4 |     {
5 |         id
6 |         name
7 |         addr
8 |     }
9 | }
10 |
11 | variables
12 | {
13 |     student:
14 |     {
15 |         "id":1,
16 |         "name":"xiaowang",
17 |         "age":18,
18 |         "addr":"四川省成都市"
19 |     }
20 | }
```

值得注意：如果请求操作的过程中，如果需要的参数是一个自定义的对象类型，那么这个对象必须是继承自InputObjectGraphType

11.2 StudentInput

```
1 public class StudentInput: InputObjectGraphType<Students>
2 {
3     public StudentInput()
4     {
5         Name = "StudentInput";
6         Description = "传入的学生模型";
7         #region 学生类型下的字段
8         Field(a => a.Id).Description("学生编号");
9         Field(a => a.Name).Description("学生姓名");
10        Field(a => a.Age).Description("学生年龄");
11        Field(a => a.Addr).Description("学生地址");
12        #endregion
13    }
14 }
```

11.3 MutationStudents

```
1 public class MutationStudents: ObjectGraphType
2 {
3     private IList<Students> students = new List<Students>();
4     public MutationStudents()
5     {
6         Name = "mutationStudents";
7         Description = "学生的修改操作";
8         Field<StudentsType>("addStudent", arguments: new QueryArguments(
9             {
10                var student = context.GetArgument<Students>("student");
11                students.Add(student);
12                return student;
13            }));
14    }
15 }
```

11.4 Console

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         string variableJosnStr = "{student:{\"id\":1,\"name\":\"xiaowang\"}}";
6         Inputs inputs = variableJosnStr.ToInputs();
7         var schema = new Schema { Query = new QueryStudents(), Mutation = new MutationStudents() };
8         var json = schema.Execute(_ => { _.Query = ("mutation ($student: {id: 1, name: \"xiaowang\"}) { addStudent(student: $student) { id name age addr } }");
9         System.Console.WriteLine(json);
```

```

10         System.Console.ReadKey();
11     }
12 }
13

```

11.5 总结mutation

mutation和query在执行方式上是不同，query是parallel（串行）的形式，而mutation是serially（并行）的执行。

12 Interfaces

GraphQL接口是一个抽象类型，它里面定义了一些字段，如果一个类型实现了这个接口，那么这个类型也必须要包含这些字段;另外这里所指的接口实现并且不是C#传统方式上的实现，在GraphQL.Net中需要的实现方式是，在实现类中使用Interface()就可以表示在当前类实现了这个GraphQL的接口类型。

12.1 Persons

```

1    /// <summary>
2    /// 人类
3    /// </summary>
4    public class Persons
5    {
6        /// <summary>
7        /// 编号
8        /// </summary>
9        public int Id { get; set; }
10       /// <summary>
11       /// 姓名
12       /// </summary>
13       public string Name { get; set; }
14       /// <summary>
15       /// 地址
16       /// </summary>
17       public string Addr { get; set; }
18   }

```

12.2 PersonsInterfaceType 将Persons定义为接口类型

```

1    public class PersonsType:InterfaceGraphType<Persons>
2    {
3        public PersonsType(ChinesesType chinesesType,EnglishesType englishes
4        {
5            Name = "personsType";
6            Description = "人类的基本类";
7            Field(a => a.Id).Description("人类编号");
8            Field(a => a.Name).Description("姓名");
9            Field(a => a.Addr).Description("人类地址");

```

```

10     }
11 }

```

12.3 PersonsQueryType

```

1  public class PersonsQueryType:ObjectGraphType
2  {
3      public PersonsQueryType()
4      {
5          Name = "personsQuery";
6          Description = "人类接口类型的查询类型";
7          Field<PersonsType>("queryPerson", resolve: context => new Chines
8      }
9  }

```

12.4 Chineses

```

1  /// <summary>
2  /// 中国人
3  /// </summary>
4  public class Chineses : Persons
5  {
6      /// <summary>
7      /// 吃中国饭
8      /// </summary>
9      public string CNEat { get; set; }
10     /// <summary>
11     /// 说中国话
12     /// </summary>
13     public string CNSay { get; set; }
14 }

```

12.5 Chineses

```

1  public class ChinesesType:ObjectGraphType<Chineses>
2  {
3      public ChinesesType()
4      {
5          Name = "中国人类型";
6          Description = "中国人的类型";
7          Field(a => a.Id).Description("中国人的身份证编号");
8          Field(a => a.Name).Description("中国人的姓名");
9          Field(a => a.Addr).Description("中国的的地址信息");
10         Field(a => a.CNEat).Description("中国人吃的中餐");
11         Field(a => a.CNSay).Description("中国人说的中国话");
12
13         Interface<PersonsType>(); //实现PersonsType这个接口类型
14         IsTypeOf = obj => obj is Chineses;
15     }
16 }

```


12.6 Console

```
1      class Program
2      {
3          static void Main(string[] args)
4          {
5              var schema = new Schema { Query = new PersonsQueryType() };
6              schema.RegisterType<ChinesesType>(); //这里需要注册接口实现类型, 具体实现
7              var json = schema.Execute(_ => {
8                  _ .Query = "query{queryPerson{id name addr}}";
9              });
10             Console.WriteLine(json);
11             Console.ReadKey();
12         }
13     }
```

12.7 Register Type

当一个Schema被创建, 它作为"root"类型 (Query、Mutation、Subscription), 并且和GraphQL中定义的类型一起对外公开。通常当我使用接口类型的时候, Schema不会公开根类型或者子类型, 因为这些具体的类型并没有在类型图中公开, 因此Schema并不知道有它们的存在。为了解决这个问题, Schema提供了RegisterType<>方法来解决这个问题, 它告诉Schema有这个特定的类型, 并且在初始化Schema的时候把这个类型加入的接口类型的PossibleTypes的集合中。

12.7.1 Console

```
1      class Program
2      {
3          static void Main(string[] args)
4          {
5              var schema = new Schema { Query = new PersonsQueryType() };
6              schema.RegisterType<ChinesesType>(); //这里需要注册接口实现类型, 具体实现
7              var json = schema.Execute(_ => {
8                  _ .Query = "query{queryPerson{id name addr}}";
9              });
10             Console.WriteLine(json);
11             Console.ReadKey();
12         }
13     }
```

12.8 IsTypeOf

IsTypeOf可以帮助你执行期间解析GraphQL类型的具体实现类, 比如, 我们有一个GraphQL接口类型PersonsInterfaceType, 这个接口类型有两个实现类, 分别是ChinesesType、EnglisesType。在这种情况下, 当我们使用接口类型作为返回类型的时, 我们需要GraphQL引擎知道当前返回的具体是哪个类型。IsTypeOf是一个委托, 它定义为将当前对象作为参数传递给IsTypeOf的委托方法, 并且这个方法的返回值为Boolean类型。

12.8.1 Chineses

```

1 public class ChinesesType:ObjectGraphType<Chineses>
2 {
3     public ChinesesType()
4     {
5         Name = "中国人类型";
6         Description = "中国人的类型";
7         Field(a => a.Id).Description("中国人的身份证编号");
8         Field(a => a.Name).Description("中国人的姓名");
9         Field(a => a.Addr).Description("中国的的地址信息");
10        Field(a => a.CNEat).Description("中国人吃的中餐");
11        Field(a => a.CNSay).Description("中国人说的中国话");
12
13        Interface<PersonsType>(); //实现PersonsType这个接口类型
14        IsTypeOf = obj => obj is Chineses;
15    }
16 }

```

注意：ObjectGraph<***>默认实现了IsTypeOf

12.9 ResolveType

对于接口类型，为了实现判断实现类的类型，除了使用IsTypeOf外，在接口类型或者内敛类型中也可以使用ResolveType，它们两者最主要的不同点是ResolveType必须要罗列所有的实现类。如果你需要新加一个实现类，那么也需要来修改这里。如果使用了ResolveType的方式，那么IsTypeOf将会被直接忽略掉

```

1 public class PersonsType:InterfaceGraphType<Persons>
2 {
3     public PersonsType()
4     {
5         Name = "personsType";
6         Description = "人类的基本类";
7         Field(a => a.Id).Description("人类编号");
8         Field(a => a.Name).Description("姓名");
9         Field(a => a.Addr).Description("人类地址");
10        ResolveType = obj => {
11
12            if (obj is Chineses)
13            {
14                return (from t in PossibleTypes where t is ChinesesType
15                }
16            else if(obj is Englishes)
17            {
18                return (from t in PossibleTypes where t is EnglishesType
19                }
20            throw new Exception("没有找到指定的解析类型");
21        };
22    }
23 }

```

13 union

联合类型是将多个类型联合在一起来使用，具体使用方式如下

13.1 Cat

```
1    /// <summary>
2    /// 猫
3    /// </summary>
4    public class Cat
5    {
6        public string MiaoMiao { get; set; }
7    }
```

13.2 CatType

```
1    public class CatType:ObjectGraphType<Cat>
2    {
3        public CatType()
4        {
5            Name = "catType";
6            Description = "cat的类型";
7            Field(a => a.MiaoMiao).Description("猫的叫");
8        }
9    }
```

13.3 Dog

```
1    /// <summary>
2    /// 狗
3    /// </summary>
4    public class Dog
5    {
6        /// <summary>
7        /// HoHo
8        /// </summary>
9        public string HoHo { get; set; }
10    }
```

13.4 DogType

```
1    public class DogType:ObjectGraphType<Dog>
2    {
3        public DogType()
4        {
5            Name = "dogType";
6            Description = "dog 的类型";
7            Field(a => a.HoHo).Description("dog的叫声");
8        }
9    }
```

13.5 CatAndDog

```
1      class CatAndDog:UnionGraphType
2      {
3          public CatAndDog()
4          {
5              Name = "catAndDog";
6              Description = "catAndDog的联合类型";
7              Type<CatType>();
8              Type<DogType>();
9          }
10     }
```

13.6 CatAndDogQueryType

```
1      public class CatAndDogQueryType:ObjectGraphType
2      {
3          public CatAndDogQueryType()
4          {
5              Name = "catAndDogQuery";
6              Description = "cat和dog的联合类型查询类型";
7              //前端可以传递参数来区分返回的类型
8              Field<CatAndDog>("queryCatAndDog", resolve: context => new Cat {
9          }
10     }
```

13.7 Console

```
1      class Program
2      {
3          static void Main(string[] args)
4          {
5              var schema = new Schema { Query = new CatAndDogQueryType()};
6              var json = schema.Execute(_ => {
7                  _.Query = "query { queryCatAndDog { ...on catType{ miaoMiao
8                  }));
9                  Console.WriteLine(json);
10                 Console.ReadKey();
11             }
12     }
```

14 Subscriptions

Subscriptions通过使用IObservable\<T>来实现, 另外我们需要提供一个能够支持Subscription

Subscriptions使用Subscription关键字。与Query、Mutation类似，当只有一个请求操作当时，GraphQL.Net实现Subscription需要使用Reactive框架，这是一个事件订阅/发布模型，有兴趣的发布、订阅是Subscription的关键词。

14.1 依赖包

14.1.1 必须装的两个包

Install-Package GraphQL.Server.Transports.AspNetCore

Install-Package GraphQL.Server.Transports.WebSockets

14.1.2 以下是三选一

Install-Package GraphQL.Server.Ui.GraphiQL

Install-Package GraphQL.Server.Ui.Playground

Install-Package GraphQL.Server.Ui.Voyager

14.2 Message

```
1  /// <summary>
2  /// 消息
3  /// </summary>
4  public class Message
5  {
6      /// <summary>
7      /// 发送者的id
8      /// </summary>
9      public String UserId { get; set; }
10     /// <summary>
11     /// 发送者的姓名
12     /// </summary>
13     public string UserName { get; set; }
14     /// <summary>
15     /// 内容
16     /// </summary>
17     public string Content { get; set; }
18     /// <summary>
19     /// 创建
20     /// </summary>
21     public DateTime CreateTime { get; set; }
22 }
```

14.3 MessageType

```
1  public class MessageType:ObjectGraphType<Message>
2  {
3      public MessageType()
4      {
5          Name = "messageType";
```

```

6         Description = "消息类型";
7         Field(m => m.UserId).Description("发送者id");
8         Field(m => m.UserName).Description("发送者姓名");
9         Field(m => m.Content).Description("内容");
10        Field(m => m.CreateTime).Description("创建时间");
11    }
12 }
13

```

14.4 MessageInputType

```

1    public class MessageInputType:InputObjectGraphType<Message>
2    {
3        public MessageInputType()
4        {
5            Name = "messageInputType";
6            Description = "消息输入类型";
7            Field(m=>m.UserId).Description("发送者id");
8            Field(m => m.UserName).Description("发送者姓名");
9            Field(m => m.Content).Description("内容");
10           Field(m => m.CreateTime).Description("创建时间");
11        }
12    }

```

14.5 IMessageService

```

1    public interface IMessageService
2    {
3        void AddError(Exception ex);
4        Message AddMessage(Message message);
5        IObservable<Message> Messages();
6    }

```

14.6 MessageService

```

1    public class MessageService : IMessageService
2    {
3        //ReplaySubject可以将之前发布的所有消息都发送给订阅者
4        private readonly ISubject<Message> _eventStream = new ReplaySubject<
5        public void AddError(Exception ex)
6        {
7            _eventStream.OnError(ex);
8        }
9        /// <summary>
10       /// 发布消息
11       /// </summary>
12       /// <param name="message"></param>
13       /// <returns></returns>
14       public Message AddMessage(Message message)
15       {

```

```

16         _eventStream.OnNext(message);
17         return message;
18     }
19     /// <summary>
20     /// 获取消息
21     /// </summary>
22     /// <returns></returns>
23     public IObservable<Message> Messages()
24     {
25         return _eventStream.AsObservable();
26     }
27 }

```

14.7 MutationMessage

在这里面发布消息

```

1     public class MutationMessage:ObjectGraphType
2     {
3         public MutationMessage(IMessageService messageService)
4         {
5             Name = "mutationMessage";
6             Description = "message的修改操作";
7             Field<MessageType>("addMessage",
8                 arguments: new QueryArguments(new QueryArgument<MessageInput>
9                     resolve: context =>
10                    {
11                        Message messageInput = context.GetArgument<Message>("mes
12                        messageService.AddMessage(messageInput);
13                        return messageInput;
14                    });
15         }
16     }

```

14.8 SubscriptionMessage

```

1     public class SubscriptionMessage:ObjectGraphType
2     {
3         private readonly IMessageService messageService;
4         public SubscriptionMessage(IMessageService messageService)
5         {
6             this.messageService = messageService;
7             Name = "messageSubscription";
8             Description = "message的subscription";
9             AddField(new EventStreamFieldType()
10            {
11                Name = "receiveMessage",
12                Description = "消息的订阅者",
13                Arguments = new QueryArguments(new QueryArgument<StringGraph
14                Type=typeof(MessageType),
15                Resolver = new FuncFieldResolver<Message>(SendMessage),
16                Subscriber = new EventStreamResolver<Message>(FilterSubscrib

```

```

17         });
18
19     }
20     /// <summary>
21     /// 为消息订阅者过滤消息
22     /// </summary>
23     /// <param name="context"></param>
24     /// <returns></returns>
25     private IObservable<Message> FilterSubscriberMessage(ResolveEventStr
26     {
27         string userName = context.GetArgument<string>("userName");
28         //为注册者进行消息过滤
29         if (!string.IsNullOrEmpty(userName) && userName.Contains("pwz"))
30         {
31             return messageService.Messages().Where(m => m.UserName.Contai
32         }
33         else
34         {
35             return messageService.Messages();
36         }
37     }
38     /// <summary>
39     /// 向消息订阅者发送消息
40     /// </summary>
41     /// <param name="context"></param>
42     /// <returns></returns>
43     private Message SendMessage(ResolveFieldContext context)
44     {
45         return context.Source as Message;
46     }
47 }
48

```

14.9 QueryMessage

需要有一个query，否则会报错

```

1     public class QueryMessage : ObjectGraphType
2     {
3         public QueryMessage()
4         {
5             Name = "queryMessage";
6             Description = "查询message";
7         }
8     }

```

14.10 Startup

这里是配置的核型部分，对GraphQL的类型和Schema以及其他类进行依赖注册、配置GraphQL服务器、WebScokt和浏览器GraphQL开发工具。


```

1 public class Startup
2 {
3     public Startup(IConfiguration configuration)
4     {
5         Configuration = configuration;
6     }
7
8     public IConfiguration Configuration { get; }
9
10    // This method gets called by the runtime. Use this method to add se
11    public void ConfigureServices(IServiceCollection services)
12    {
13        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.V
14        //依赖注入
15        services.AddSingleton<IMessageService, MessageService>();
16        services.AddSingleton<MessageType>();
17        services.AddSingleton<MessageInputType>();
18        services.AddSingleton<QueryMessage>();
19        services.AddSingleton<MutationMessage>();
20        services.AddSingleton<SubscriptionMessage>();
21        services.AddSingleton<MessageSchema>();
22        services.AddGraphQL(options =>
23        {
24            options.EnableMetrics = true;
25            options.ExposeExceptions = true;
26        })
27            .AddWebSockets()
28            .AddDataLoader();
29
30    }
31
32    // This method gets called by the runtime. Use this method to config
33    public void Configure(IApplicationBuilder app, IHostingEnvironment e
34    {
35        if (env.IsDevelopment())
36        {
37            app.UseDeveloperExceptionPage();
38        }
39        // this is required for websockets support
40        app.UseWebSockets();
41        // use websocket middleware for ChatSchema at path /graphql
42        app.UseGraphQLWebSockets<MessageSchema>("/graphql");
43        // use HTTP middleware for ChatSchema at path /graphql
44        app.UseGraphQL<MessageSchema>("/graphql");
45        // use graphql-playground middleware at default url /ui/playgrou
46        app.UseGraphQLPlayground(new GraphQLPlaygroundOptions());
47    }
48 }

```

15 Query Validation

当一个查询被执行的时候，会有一些验证规则被执行，这些都是默认执行。通过使用validationRules属性我们能够自定义一些验证规则或者益处一个已经存在的验证规则。

```
1 schema.Execute(_ =>
2 {
3     _.Query = "...";
4     _.ValidationRules =
5         new[]
6         {
7             new RequiresAuthValidationRule()
8         }
9     .Concat(DocumentValidator.CoreRules());
10 });
```

16 Query Organization

应为GraphQL只允许有一个查询根对象，这就可能知道我们的根查询对象随着时间的推移越来越臃肿，介

16.1 ChineseQuery

```
1 public class ChineseQuery:ObjectGraphType
2 {
3     public ChineseQuery()
4     {
5         Name = "chineseQuery";
6         Description = "中国人相关的查询";
7         Field<ChineseType>("getChinese", resolve: context => new Chinese
8     }
9 }
```

16.2 EnglishQuery

```
1 public class EnglishQuery:ObjectGraphType
2 {
3     public EnglishQuery()
4     {
5         Name = "englishQuery";
6         Description = "there are queries of english";
7         Field<EnglishType>("getEnglish", resolve: context => new English
8     }
9 }
10 }
```

16.3 GroupQueries

注意这里resolve的返回只设置一个空对象即可

```

1      public class GroupQueries:ObjectGraphType
2      {
3          public GroupQueries()
4          {
5              Name = "groupQueries";
6              Description = "查询组";
7              Field<ChineseQuery>("chineseQuery", resolve: context => new { })
8              Field<EnglishQuery>("englishQuery", resolve: context => new { })
9          }
10     }

```

16.4 MySchema

```

1      public class MySchema:Schema
2      {
3          public MySchema()
4          {
5              Query = new GroupQueries();
6          }
7      }

```

17 User Context

当前请求上下文信息，可以根据这个信息进行一些身份认证和业务逻辑的处理，我们可以在字段解析或者验证规则的时候都可以获取到这个上下文值。

17.1 初始化值

```

1      services.AddGraphQL(options =>
2      {
3          options.EnableMetrics = true;
4          options.ExposeExceptions = true;
5      })
6      .AddUserContextBuilder(httpContext=> {
7          var url = httpContext.Request.Path;
8          //这里做一些业务逻辑的处理，然后返回合适的对象
9          return new Chinese { Id = -1, Name = "平娃子", CNEat = "川菜" };
10     });
11

```

17.2 获取值

```

1      Field<ChineseType>("getChinese",
2          resolve: context =>
3          {
4              var userContext = context.UserContext as Chinese;
5              return new Chinese { Id = 1, Name = "平娃子", CNEat = "川菜" };
6          });

```

18 Error Handling

返回结果中存在一个error属性（在执行期间没有错误信息当时，这个属性不会被显示出来），里面会包含详细当错误信息和调用堆栈。

18.1 是否向前端展示错误堆栈信息

```
1 services.AddGraphQL(options =>
2 {
3     options.EnableMetrics = true;
4     options.ExposeExceptions = false; //不向前端展示详细的错误堆栈信息
5 })
6
```

18.2 自定义错误信息

这里有两种方式，一种是在不影响当前流程的情况下，将错误信息加入到error属性中，另外一种中断当前流程并将错误信息加入到error属性中

```
1 Field<ChineseType>("getChinese",
2     resolve: context =>
3     {
4         throw new ExecutionError("出现了错误");
5         //context.Errors.Add(new ExecutionError("出现了错误信息"))
6         return new Chinese { Id = 1, Name = "平娃子", CNEat = "川菜" };
7     });
8
```

18.3 前端展示的错误信息

```
1 "errors": [
2   {
3     "message": "GraphQL.ExecutionError: 出现了错误\r\n    at QueryOrganizatio
4     "locations": [
5       {
6         "line": 3,
7         "column": 5
8       }
9     ],
10    "path": [
11      "chineseQuery",
12      "getChinese"
13    ]
14  }
15]
```

值得注意：我们可以自定义一些额外到错误处理程序和日志记录

19 Dependency Injection

使用Asp.NetCore自带的

20 DataBases

使用GraphQL与ORM框架结合在一起来使用不做详细介绍，有兴趣的同学可以查看[DataBase](#)

21 Protection Against Malicious Queries

因为GraphQL允许嵌套查询，那么在这种情况下就很有可能引起GraphQL服务器拒绝服务攻击。为来缓解在GraphQL.Validation.Complexity中提供了一个ComplexityConfiguration类，这个类里面就

```
1 public class ComplexityConfiguration
2 {
3     public ComplexityConfiguration();
4
5     public int? MaxDepth { get; set; } //查询深度
6     public int? MaxComplexity { get; set; } //查询成本
7     public double? FieldImpact { get; set; } //查询实体（字段数量）
8 }
```

```
1 services.AddGraphQL(options =>
2 {
3     options.EnableMetrics = true;
4     options.ExposeExceptions = false;
5     options.ComplexityConfiguration = new ComplexityConfiguratio
6 })
```

21.1 举个例子计算一下

MaxComplexity和FieldImpact是对每个查询返回的实体（或者数据库中的单元格）数量的预估，因此

```
1 {
2   Product {
3     Title
4     ...X
5   }
6 }
7
8 fragment X on Product {
9   Location(first: 3) {
10     lat
11     long
12   }
13 }
```

1. 查询深度为2。第一次查询是返回一个Product的列表，第二次查询是获取这个列表的前三个。
2. 假设每次查询只返回2个实体，查询影响就为2
3. 查询成本：2 Products returned + 2 * (1 * Title per Product) + 2 * [(3 * Locations) + (3 * lat entries) + (3 * long entries)] = 22

注意；设置MaxComplexity时我们一般使用所有查询计算的最大值。“同样查询实体数我们根据返回实体数的最大值来设置”（这句话是我根据文档中所描述的意思进行的个人猜测）。

22 Object/Field Metadata

IGraphType和FieldType实现了IProvideMetadata接口，它允许我们将任意的信息添加到field或

```
1 public interface IProvideMetadata
2 {
3     IDictionary<string, object> Metadata { get; }
4     TType GetMetadata<TType>(string key, TType defaultValue = default(TType));
5     bool HasMetadata(string key);
6 }
```

```
1 public class MyGraphType : ObjectGraphType
2 {
3     public MyGraphType()
4     {
5         Metadata["rule"] = "value";
6     }
7 }
```

23 Field Middleware

通过写一个中间件，可以实现在字段解析期间做一些额外的操作。

23.1 InstrumentFieldsMiddleware

```
1 class InstrumentFieldsMiddleware
2 {
3     public Task<object> Resolve(ResolveFieldContext context, FieldMiddleware
4     {
5         var metadata = new Dictionary<string, object>
6         {
7             {"typeName", context.ParentType.Name},
8             {"fieldName", context.FieldName},
9             {"path", context.Path}
10        };
11    }
```

```

11         Console.WriteLine("hello world");
12         using (context.Metrics.Subject("field", context.FieldName, meta)
13         {
14             return next(context);
15         }
16     }
17 }

```

23.2 Console

```

1     class Program
2     {
3         static void Main(string[] args)
4         {
5             Schema schema = new Schema() { Query = new PersonQuery() };
6             String json = schema.Execute(_ =>
7             {
8                 _.Query = "{queryPerson{id name age}}";
9                 _.FieldMiddleware.Use(next =>
10                 {
11                     return context =>
12                     {
13                         return new InstrumentFieldsMiddleware().Resolve(cont
14                     };
15                 });
16             });
17             Console.WriteLine(json);
18             Console.ReadKey();
19         }
20     }

```

24 Metrics

对查询进行性能分析时使用

```

1         var start = DateTime.UtcNow;
2         var executor = new DocumentExecutor();
3         ExecutionResult result =executor.ExecuteAsync( _ =>
4         {
5             _.Schema = schema;
6             _.Query = "query{queryPerson{id name age}}";
7             _.EnableMetrics = true;
8             _.FieldMiddleware.Use<InstrumentFieldsMiddleware>();
9         }).Result;
10        result.EnrichWithApolloTracing(start);
11        string serialStr = Newtonsoft.Json.JsonConvert.SerializeObject(r
12        Console.WriteLine(serialStr);
13        Console.ReadKey();

```

25 Authorization

我们可以设置一些验证规则用于查询之前对身份认真，比较详细对介绍可以查看[Authorization](#)项目。

26 GraphQL

GraphiQL是通过浏览器与GraphQL服务器交互的一个工具（除了GraphiQL外，还有playground、voyager等）

