API设计与实现

主讲人：陈长兵

目 录

CONTENTS

8 API实现-基于GraphQL

- GraphQL基础
- GraphQL框架
- 基于Apollo的GraphQL实战
- GraphQL java实战

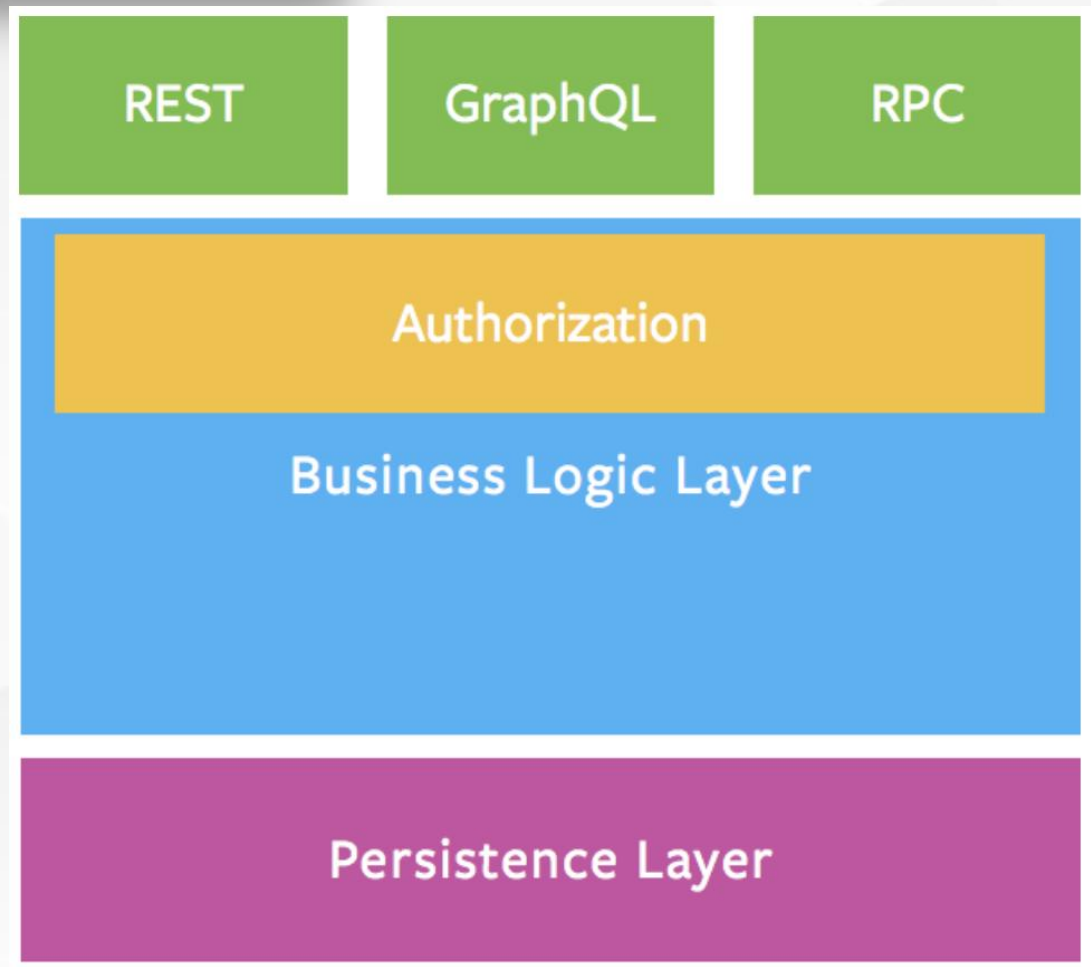
8.1 GraphQL基础

什么是GraphQL

- Facebook开发的一种数据查询语言，于2015年公开发布
- 官方描述
  - GraphQL 既是一种用于 API 的查询语言，也是一个满足你数据查询的运行时。
  - GraphQL 对你的 API 中的数据提供了一套易于理解的完整描述，使得客户端能够准确地获得它需要的数据，而且没有任何冗余，
  - 让 API 更容易地随着时间推移而演进，还能用于构建强大的开发者工具。
- 官网：https://graphql.org
- 中文网：https://graphql.cn

8.1 GraphQL基础

什么是GraphQL



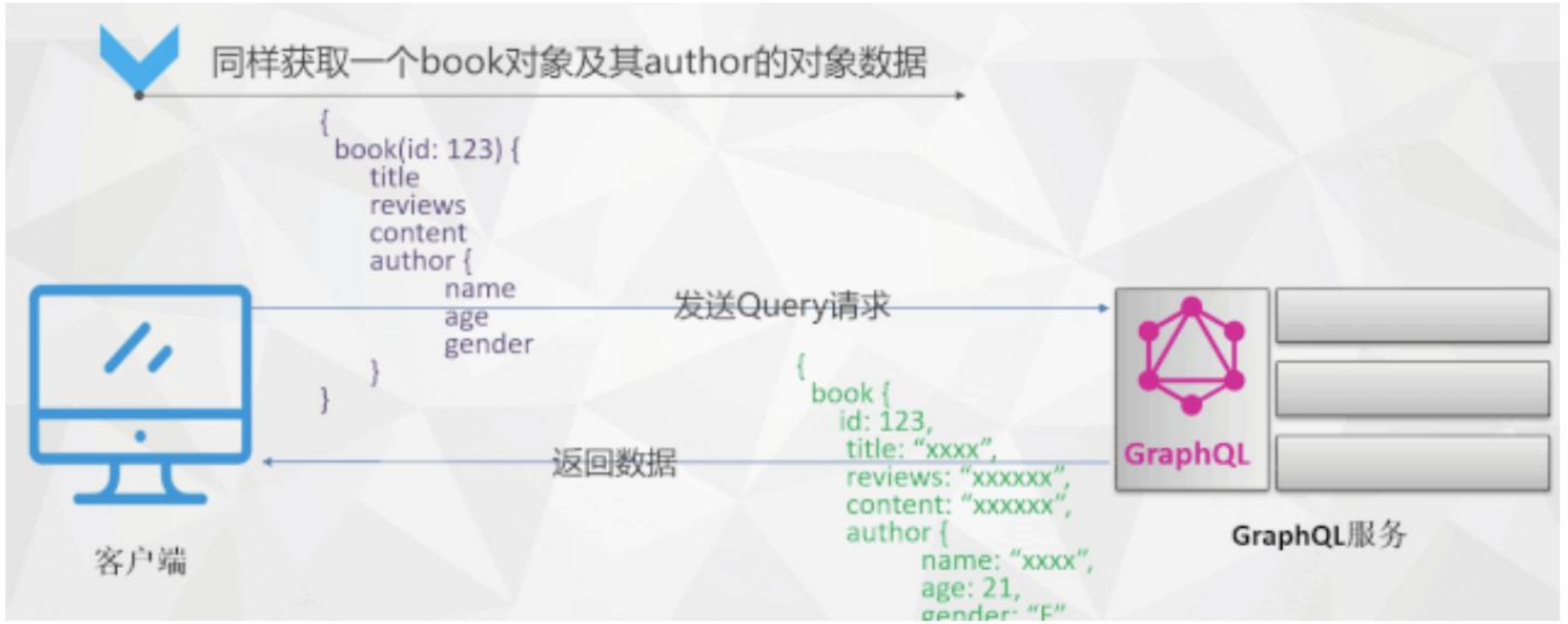
8.1 GraphQL基础

什么是GraphQL

REST API获取数据

8.1 GraphQL基础

什么是GraphQL



GraphQL获取数据

8.1 GraphQL基础

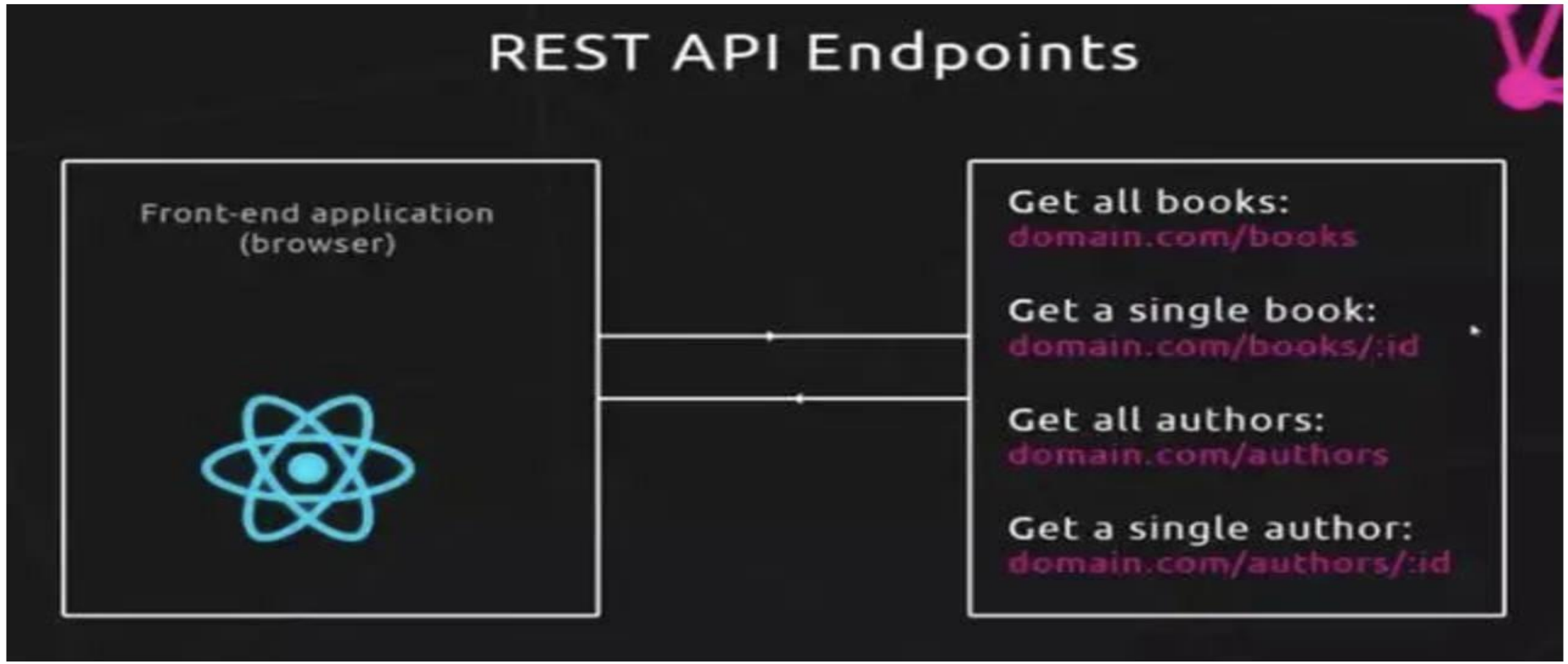
什么是GraphQL

- REST是面向资源，不同资源拥有不同的endpoint
- GraphQL是面向数据，每个GraphQL 服务其实对外只提供一个用于调用内部接口的端点，所有的请求都访问这个暴露出来的唯一端点。

8.1 GraphQL基础

什么是GraphQL

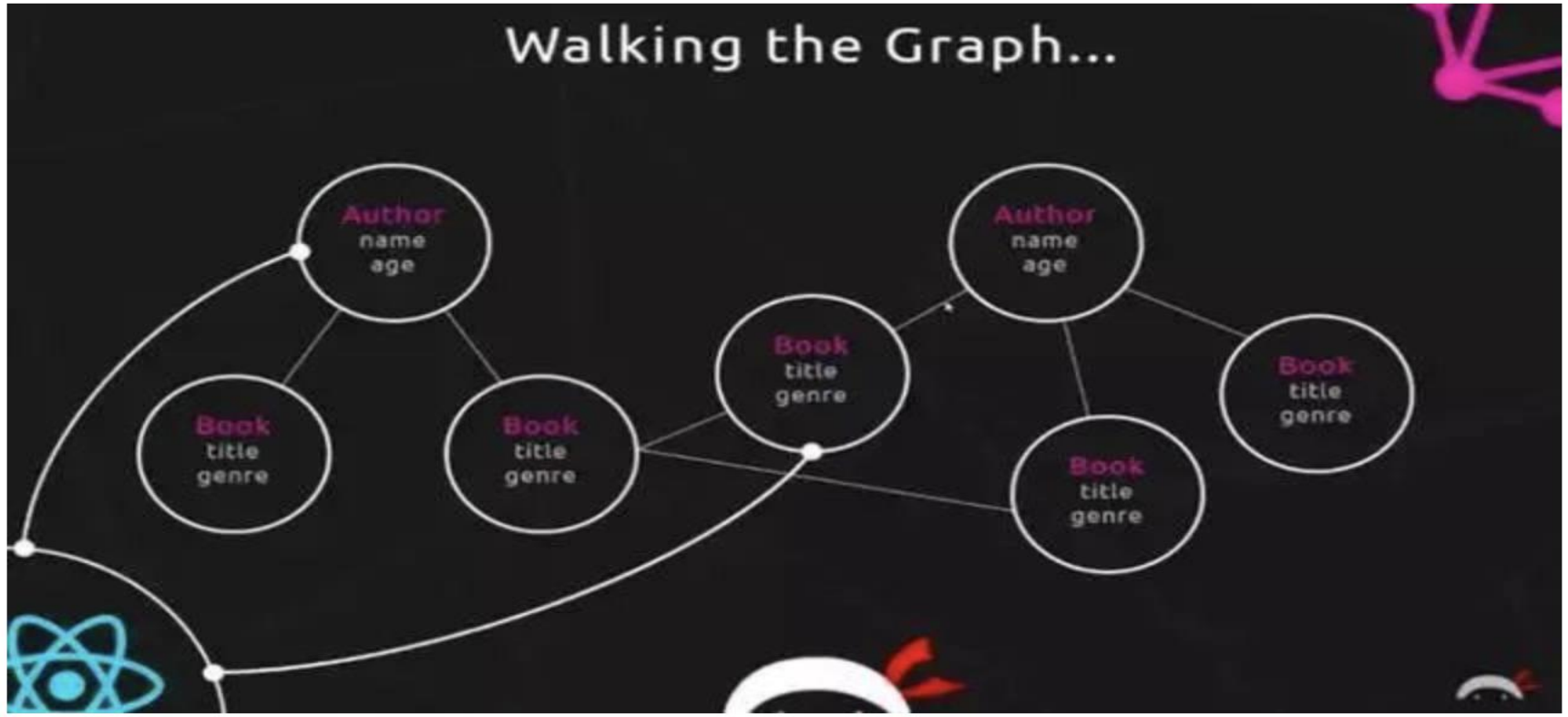
◆ REST API



8.1 GraphQL基础

什么是GraphQL

◆ GraphQL：图数据库模式的数据查询



8.1 GraphQL基础

GraphQL特点

- 请求并得到所要的数据
- 在单个请求中获取许多资源
- 描述类型的可能性
- 功能强大的开发人员工具

8.1 GraphQL基础

GraphQL背景

- 移动端需要更高效、更精准的数据加载
- 后端统一支撑各种不同的前端框架和平台
- 快速应对业务变化需求，加快产品快速开迭代
- 前端工程化的背景下，前端对数据的掌控自由度增长。

8.1 GraphQL基础-语言规范

字段Fields

- GraphQL就是请求对象上的特定字段

```
{
  hero {
    name
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

8.1 GraphQL基础-语言规范

字段Fields

- 可交互性
- 次级选择

```
{
  hero {
    name
    # 查询可以有备注!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
```

8.1 GraphQL基础-语言规范

参数Arguments

◆ 每个字段和嵌套对象都能有自己的一组参数

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

8.1 GraphQL基础-语言规范

别名Aliases

◆ 重命名结果中的字段

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

8.1 GraphQL基础-语言规范

片段Fragments

◆ 可复用单元，在需要它们的地方引入

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
}
```

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Han Solo"
        },
        {
          "name": "R2-D2"
        }
      ]
    },
    "rightComparison": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
```

8.1 GraphQL基础-语言规范

片段Fragments

◆ 片段内可以使用查询或变更中声明的变量

```
query HeroComparison($first: Int = 3) {
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  friendsConnection(first: $first) {
    totalCount
    edges {
      node {
        name
      }
    }
  }
}
```

VARIABLES

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "friendsConnection": {
        "totalCount": 4,
        "edges": [
          {
            "node": {
              "name": "Han Solo"
            }
          },
          {
            "node": {
              "name": "Leia Organa"
            }
          },
          {
            "node": {
              "name": "C-3PO"
            }
          }
        ]
      }
```

8.1 GraphQL基础-语言规范

操作Operation

◆ 带操作类型和操作名称的查询

```
query HeroNameAndFriends {
  hero {
    name
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
```

8.1 GraphQL基础-语言规范

操作Operation

- 操作类型
  - 查询query：获取数据的基本查询
  - 变更mutation：支持对数据的增删改等操作
  - 订阅subscription：用于监听数据变动、并靠websocket等协议推送变动的消息方式

8.1 GraphQL基础-语言规范

操作Operation

- 操作名称
  - 在有多操作的文档中是必需的
  - 建议使用，对于调试和服务端日志记录非常有用

8.1 GraphQL基础-语言规范

变量Variables

- 变量定义
  - $episode: Episode
  - 变量前缀必须为$，后跟其类型
  - 可选或者必要的，在类型后加!

8.1 GraphQL基础-语言规范

变量Variables

- 变量默认值
  - $episode: Episode = "JEDI"
  - 在类型后将默认值赋给变量
  - 在调用操作时，指定变量值，则会覆盖默认值

8.1 GraphQL基础-语言规范

指令Directives

◆ 如何动态改变查询结构？

```
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

8.1 GraphQL基础-语言规范

指令Directives

- GraphQL核心规范包含两个指令
  - @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
- 服务端实现可以定义新指令来添加新的特性

8.1 GraphQL基础-语言规范

变更Mutions

◆ 任何导致写入的操作都应该显式通过变更来发送

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

VARIABLES

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

8.1 GraphQL基础-语言规范

变更Mutions

- 变更中的多个字段
  - 查询字段时，是并行执行
  - 变更字段时，是线性执行，一个接着一个

8.1 GraphQL基础-语言规范

内联片段Inline Fragments

◆ 请求具体类型上的字段

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}
```

VARIABLES

```
{
  "ep": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "primaryFunction": "Astromech"
    }
  }
}
```

8.1 GraphQL基础-语言规范

内联片段Inline Fragments

◆ 元字段Meta fields

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
    }
    ... on Droid {
      name
    }
    ... on Starship {
      name
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo"
      },
      {
        "__typename": "Human",
        "name": "Leia Organa"
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1"
      }
    ]
```

8.1 GraphQL基础-语言规范

类型系统Type System

- GraphQL是强类型语言，通过类型系统定义多种数据类型
- 类型系统帮助定义schema，客户端与服务端交互的契约
- 常用的数据类型
  - 标量类型
  - 对象类型
  - 枚举类型
  - 查询/变更类型

8.1 GraphQL基础-语言规范

标量类型Scalar

- 默认自带标量类型
  - Int：有符号32位整数
  - Float：有符号双精度浮点值
  - String：UTF-8字符序列
  - Boolean：true 或 false
  - ID：表示一个唯一标识符，序列化方式与String一样

8.1 GraphQL基础-语言规范

标量类型Scalar

- 大部分GraphQL服务实现中，都有自定义标量类型的方式
  - 比如：定义Date类型，scalar Date
  - 实现重点：序列化、反序列化和验证

8.1 GraphQL基础-语言规范

对象类型Object

- 定义对象的格式及其构成的字段
  - 对象类型的名称
  - 字段及字段类型
  - 非空，数组

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

8.1 GraphQL基础-语言规范

对象类型Object

- 对象类型的每个字段都可以有零个或多个参数
  - 参数必须要有名字
  - 参数可能是必选或可选。如果可选，可以定义默认值
  - 非空，数组

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

8.1 GraphQL基础-语言规范

查询和变更类型

- Schema的两个特殊类型
  - query和mutation
  - 每个GraphQL服务都有一个querye类型，可能有一个mutation类型
  - 除了定义每个GraphQL查询的入口，跟常规对象类型没有差异

8.1 GraphQL基础-语言规范

枚举类型Enumeration

- 是一种特殊的标量，在特殊的可选值集合内取值

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
```

8.1 GraphQL基础-语言规范

类型修饰符

- 非空!
  - 对象类型的字段
  - 变量

```
type Character {
  name: String!
  appearsIn: [Episode]!
}
```

```
query DroidById($id: ID!) {
  droid(id: $id) {
    name
  }
}
```

VARIABLES

```
{
  "id": null
}
```

```
{
  "errors": [
    {
      "message": "Variable \"$id\" of non-null type \"ID!\" must not be nul
      "locations": [
        {
          "line": 1,
          "column": 17
        }
      ]
    }
```

8.1 GraphQL基础-语言规范

类型修饰符

- 数组[]
  - 数组本身可以为空，但其不能有任何控制的成员
  - 不为空的数组

```
myField: null // 有效
myField: [] // 有效
myField: ['a', 'b'] // 有效
myField: ['a', null, 'b'] // 错误
```

8.1 GraphQL基础-语言规范

接口Interfaces

◆接口是抽象类型，包含某些字段。

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}
```

8.1 GraphQL基础-语言规范

接口Interfaces

◆对象类型必须包含这些字段

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

8.1 GraphQL基础-语言规范

接口Interfaces

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    primaryFunction
  }
}
```

VARIABLES

```
{
  "ep": "JEDI"
}
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"primaryFunction\" on type \"Cl
      "locations": [
        {
          "line": 4,
          "column": 5
        }
      ]
    }
  ]
}
```

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
  }
}
```

VARIABLES

```
{
  "ep": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "primaryFunction": "Astromech"
    }
  }
}
```

8.1 GraphQL基础-语言规范

联合类型Union Type

- 将多个具体对象类型组合在一起
- 不能使用接口或其他联合类型创造一个联合类型

```
union SearchResult = Human | Droid | Starship
```

8.1 GraphQL基础-语言规范

联合类型Union Type

◆查询时，需要使用内联片段

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo",
        "height": 1.8
      },
      {
        "__typename": "Human",
        "name": "Leia Organa",
        "height": 1.5
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1",
        "length": 9.2
      }
    ]
```

8.1 GraphQL基础-语言规范

联合类型Union Type

◆使用同一个接口，可以单独指定接口的字段，避免重复相同的字段

```
{
  search(text: "an") {
    __typename
    ... on Character {
      name
    }
    ... on Human {
      height
    }
    ... on Droid {
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

8.1 GraphQL基础-语言规范

输入类型Input Type

- 除了枚举或标量值外，如何传递复杂对象？
- 定义方式与常规对象一样，除了关键词使用input

```
input ReviewInput {
  stars: Int!
  commentary: String
}
```

8.1 GraphQL基础-语言规范

输入类型Input Type

- 除了枚举或标量值外，如何传递复杂对象？
- 定义方式与常规对象一样，除了关键词使用input

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

VARIABLES

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

8.1 GraphQL基础-语言规范

验证

- 通过使用类型系统，可以预判一个查询是否有效。
- 在创建查询时就有效地通知开发者，而不用依赖运行时检查。

```
{
  hero {
    ...NameAndAppearancesAndFriends
  }
}

fragment NameAndAppearancesAndFriends on Characte
  name
  appearsIn
  friends {
    ...NameAndAppearancesAndFriends
  }
}
```

```
{
  "errors": [
    {
      "message": "Cannot spread fragment \"NameAndAppearancesAndFriends\" wi
      "locations": [
        {
          "line": 11,
          "column": 5
        }
      ]
    }
  ]
}
```

8.1 GraphQL基础-语言规范

执行

- 一个 GraphQL 查询在被验证后，GraphQL 服务器会将之执行，并返回与请求的结构相对应的结果，该结果通常会是 JSON 的格式。
- 每个类型的每个字段都由一个resolver函数支持，该函数会由GraphQL服务开发人员提供
- 当一个字段被执行时，相应的resolver就被调用产生一个值
- 如果字段产生标量值，则执行完成；如果一个字段产生一个对象，则该查询将继续执行对象对应字段的解析器，直到生成标量值

8.1 GraphQL基础-语言规范

解析器

◆ 类型系统定义

```
type Query {
  human(id: ID!): Human
}

type Human {
  name: String
  appearsIn: [Episode]
  starships: [Starship]
}

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

type Starship {
  name: String
}
```

8.1 GraphQL基础-语言规范

解析器

◆ GraphQL查询

```
{
  human(id: 1002) {
    name
    appearsIn
    starships {
      name
    }
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Han Solo",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "starships": [
        {
          "name": "Millenium Falcon"
        },
        {
          "name": "Imperial shuttle"
        }
      ]
    }
  }
}
```

8.1 GraphQL基础-语言规范

解析器

- GraphQL API的入口
  - ROOT类型或Query类型的解析器

```
Query: {
  human(obj, args, context, info) {
    return context.db.loadHumanByID(args.id).then(
      userData => new Human(userData)
    )
  }
}
```

8.1 GraphQL基础-语言规范

简单解析器

- 获取对象的字段字段
  - obj是上层解析器生成的Human对象
  - 一般GraphQL库会有默认解析器，如果一个字段没有自定义解析器时，则会从上层对象中读取和返回与这个字段同名的属性

```
Human: {
  name(obj, args, context, info) {
    return obj.name
  }
}
```

8.1 GraphQL基础-语言规范

列表解析器

- Human对象starshipIDs只保存id信息
  - 需要遍历每个id，进一步查询并组装Starship对象

```
Human: {
  starships(obj, args, context, info) {
    return obj.starshipIDs.map(
      id => context.db.loadStarshipByID(id).then(
        shipData => new Starship(shipData)
      )
    )
  }
}
```

8.1 GraphQL基础-语言规范

组装结果

- 每个字段被解析后，结果被放置到Map结构中
  - 字段名字作为key，解析器返回的值作为value
  - 组装顺序，从查询字段的底部子节点开始，直到根Query类型的起始节点

8.1 GraphQL基础-语言规范

内省Introspection

- 通过内省机制获取元数据信息
  - GraphQL Schema提供哪些查询？
  - 查询有些类型？
  - 指定类型有些字段？

8.1 GraphQL基础-语言规范

内省Introspection

◆ 查询有些类型？

```
{
  __schema {
    types {
      name
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query"
        },
        {
          "name": "String"
```

8.1 GraphQL基础-语言规范

内省Introspection

◆ 查询有哪些查询？

```
{
  __schema {
    queryType {
      name
    }
  }
}
```

```
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      }
    }
  }
}
```

8.1 GraphQL基础-语言规范

内省Introspection

- 查询特定类型的基本信息？
  - kind字段描述对象类型，是_KeyKind枚举类型，有8种取值
  - SCALAR，OBJECT，INTERFACE，UNION，ENUM
  - INPUT_OBJECT，LIST，NON_NULL

```
{
  __type(name: "Droid") {
    name
    kind
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "kind": "OBJECT"
    }
  }
}
```

8.1 GraphQL基础-语言规范

内省Introspection

◆ 查询特定类型的详细信息？

- 如果kind是NON_NULL（包装类型），可通过ofType查看内部元素的类型

```
__type(name: "Droid") {
  name
  fields {
    name
    type {
      name
      kind
      ofType {
        name
        kind
      }
    }
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null,
            "kind": "NON_NULL",
            "ofType": {
              "name": "ID",
              "kind": "SCALAR"
            }
          }
        },
        {
          "name": "name",
```

8.1 GraphQL基础-语言规范

内省Introspection

◆ 查询文档信息？

```
{
  __type(name: "Droid") {
    name
    description
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "description": "An autonomous mechanical charac
    }
  }
}
```

8.1 GraphQL基础

关于Graphs的思考

- 一切皆是图
  - 使用GraphQL，可以将所有的业务建模为图

8.1 GraphQL基础

关于Graphs的思考

- 共同语言
  - 命名是构建直观接口中一个困难但重要的部分

8.1 GraphQL基础

关于Graphs的思考

- 业务逻辑层
  - 业务逻辑层应该作为执行业务域规则的唯一正确来源

8.1 GraphQL基础

通过HTTP提供服务

- GraphQL一般使用HTTP作为客户端服务器之间的通信协议
- 网络请求管道
  - 大多数现代Web框架使用管道模型，通过一组中间组件（过滤器或插件）递归传递请求
  - 当请求流经管道时，可以被检查、转换、修改或是响应并终止
  - GraphQL一般放在所有身份验证中间组件之后
- 一般使用单个URL入口，通常使用/graphql

8.1 GraphQL基础

通过HTTP提供服务

- 使用GET请求
  - query，variables，operationName参数

```
{
  me {
    name
  }
}
```

```
http://myapi/graphql?query={me{name}}
```

8.1 GraphQL基础

通过HTTP提供服务

- 使用POST请求
  - contenttype使用application/json

```
{
  "query": "...",
  "operationName": "...",
  "variables": { "myVariable": "someValue", ... }
}
```

8.1 GraphQL基础

通过HTTP提供服务

- 响应
  - 以JSON格式返回

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

8.1 GraphQL基础-开发原则

完整原则

- 单一图
  - 公司内应当只有一个统一图，而不是多个团队分别创建多个图

8.1 GraphQL基础-开发原则

完整原则

- 联合实现
  - 虽然只有一个图，但该图应该由多个团队联合实现

8.1 GraphQL基础-开发原则

完整原则

- 追踪在注册表中的Schema
  - 注册和追踪图时应当有一个单一的事实来源

8.1 GraphQL基础-开发原则

敏捷原则

- 抽象、面向需求的Schema
  - Schema 应当作为抽象层以隐藏服务实现细节并为消费者提供灵活性

8.1 GraphQL基础-开发原则

敏捷原则

- 使用敏捷方法进行 Schema 开发
  - Schema 应当根据实际需求增量构建，并随着时间的推移平滑演进。

8.1 GraphQL基础-开发原则

敏捷原则

- 迭代地提高性能
  - 性能管理应当是一个连续的、数据驱动的过程，可以平滑地适应不断变化的查询负载和服务实现。

8.1 GraphQL基础-开发原则

敏捷原则

- 使用图的元数据为开发人员提供支持
  - 开发人员应当在整个开发过程中对图充分了解。

8.1 GraphQL基础-开发原则

操作原则

- 访问和需求控制
  - 基于每个客户端授予对图的访问权限，并管理客户端可以访问的内容和方式。

8.1 GraphQL基础-开发原则

操作原则

- 结构化日志
  - 捕获所有图操作的结构化日志，并以之为主要工具了解图的使用情况。

8.1 GraphQL基础-开发原则

操作原则

- 将 GraphQL 层从服务层分离
  - 采用分层架构将数据图功能分解为单独的层，而不是融入到每个服务中。

8.2 GraphQL框架

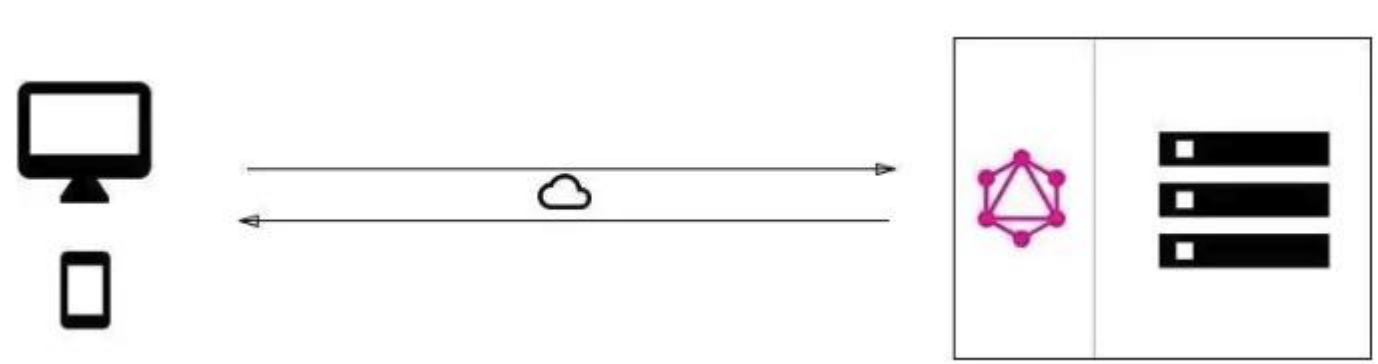
GraphQL Server

- GraphQL 是描述 GraphQL 服务器行为的规范。
  - 它是一套关于如何处理请求和响应的指南，如支持的协议，服务器可以接受的数据格式，服务器返回的响应格式等。
  - 客户端对GraphQL的请求server称为Query。
- GraphQL Server如何接入现有系统中？

8.2 GraphQL框架

接入架构

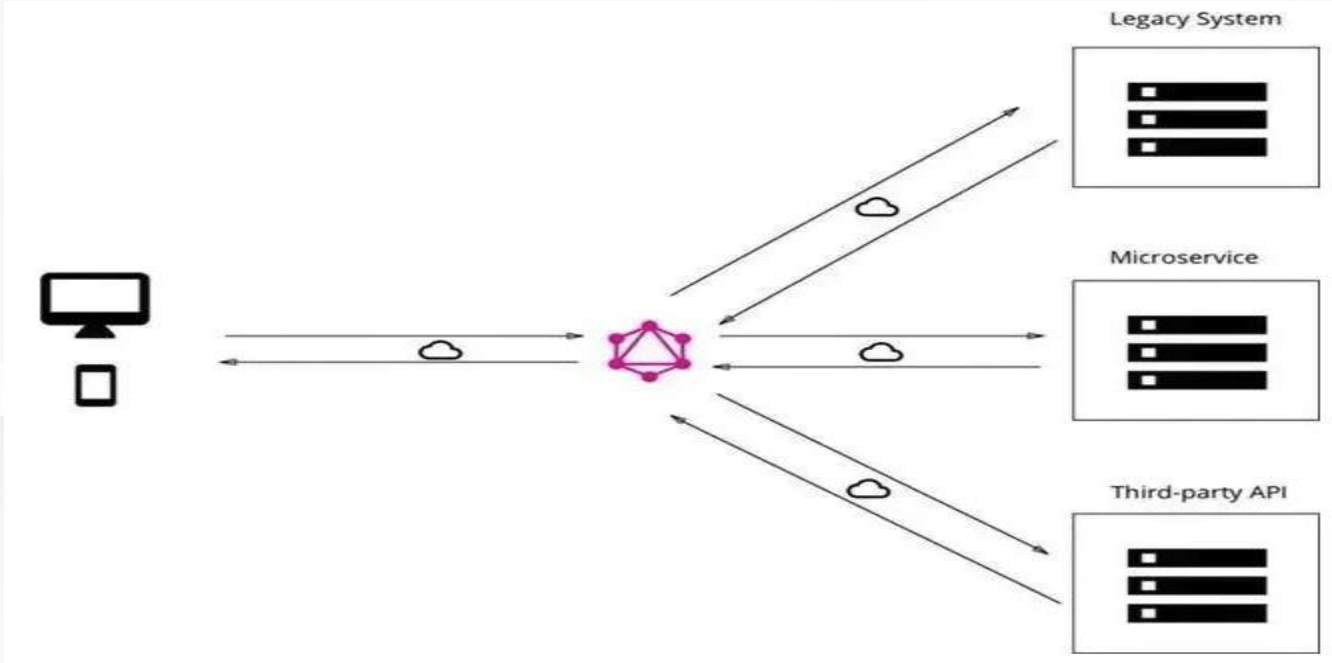
- GraphQL如何接入现有系统中？
  - 直接数据库的接入



8.2 GraphQL框架

接入架构

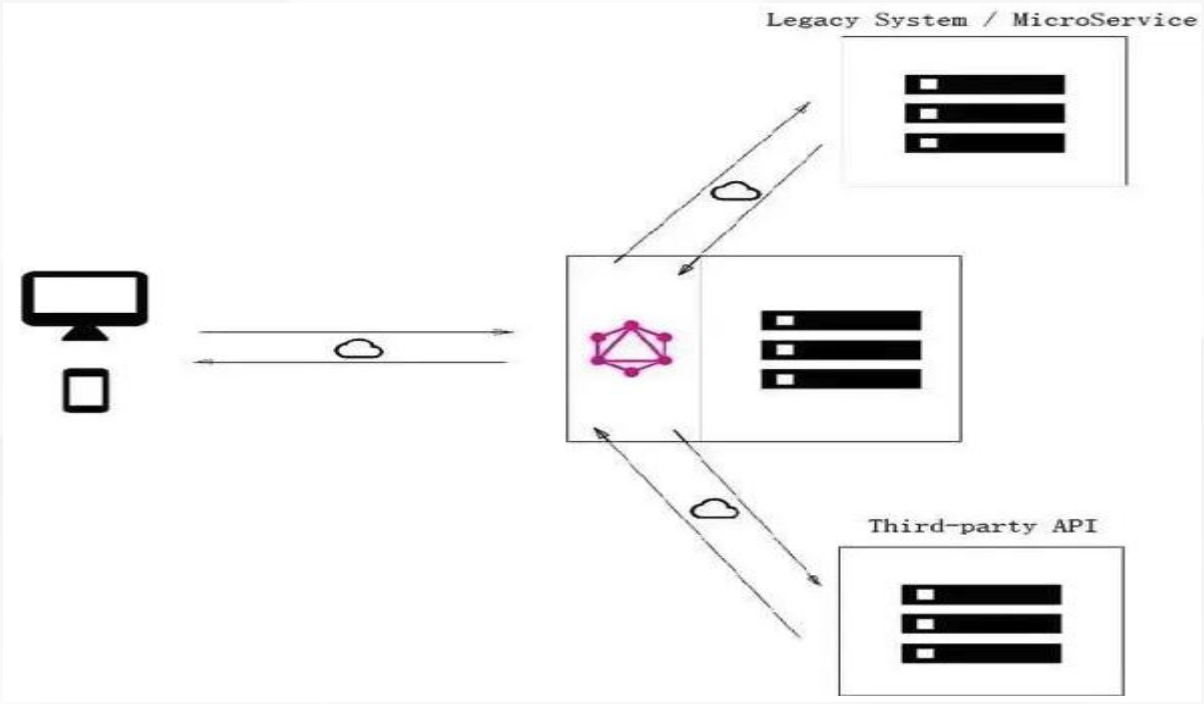
- GraphQL如何接入现有系统中？
  - 集成现有服务的GraphQL层



8.2 GraphQL框架

接入架构

- GraphQL如何接入现有系统中？
  - 混合方式



8.2 GraphQL框架

应用程序组件

- 应用程序组件包含两部分
  - 服务器端组件
  - 客户端组件

8.2 GraphQL框架

应用程序组件

- 服务器组件，GraphQL Server是最重要的核心组件
  - Schema，是GraphQL Server实现的关键，描述连接到它的客户端可用的功能
  - Query，处理客户端应用程序的请求，从数据库或旧API中检索数据
  - Resolver，定义将GraphQL操作转换为数据的说明

8.2 GraphQL框架

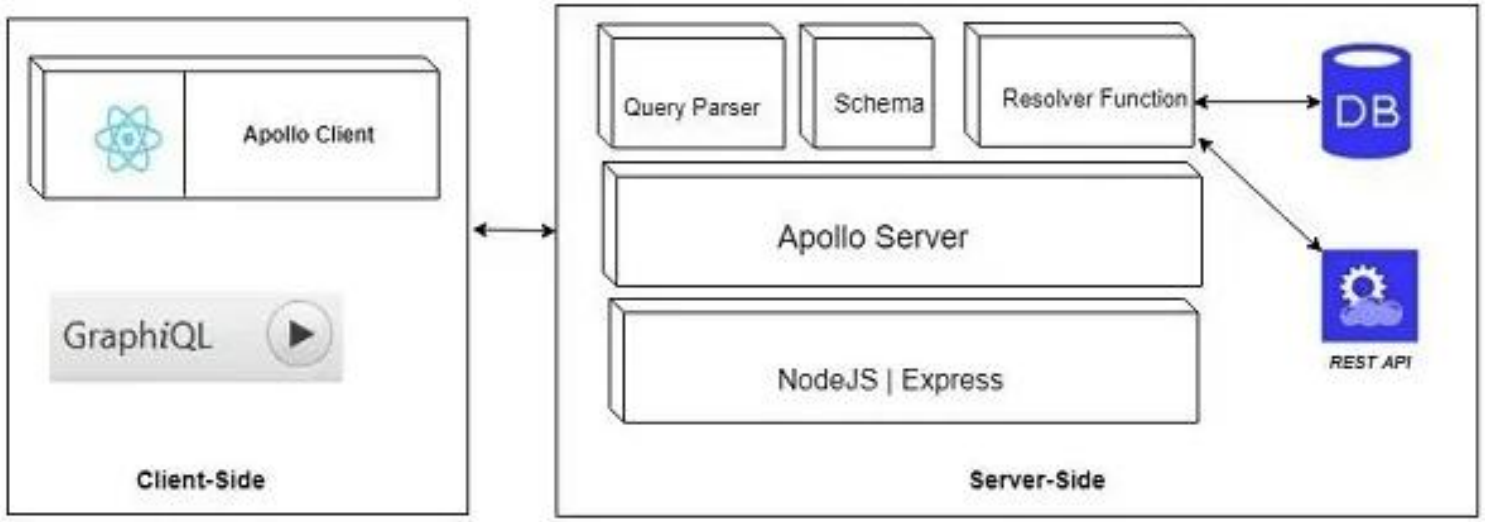
应用程序组件

- 客户端组件
  - GraphiQL，基于浏览器的界面，用于编辑和测试GraphQL查询和变更
  - ApolloClient，构建GraphQL客户端应用程序的工具

8.2 GraphQL框架

应用程序组件

◆基于Apollo的客户端-服务器架构



8.2 GraphQL框架

GraphQL生态

◆ GraphQL是一门语言规范，本身不提供具体实现，在实际开发中需要选择适合的第三方实现。目前有很多种语言已支持GraphQL的实现。

JavaScript	Go	PHP	Java / Kotlin	C# / .NET	Python
Swift / Objective-C	Rust	Ruby	Elixir	Scala	Flutter
Clojure	Haskell	C / C++	Elm	OCaml / Reason	Erlang
Julia	R	Groovy	Perl	D	Ballerina

8.2 GraphQL框架

GraphQL生态

- 社区驱动项目分四个层面
  - 规范层面：GraphQL本身核心部分，与语言无关，描述的是GraphQL的标准行为。由GraphQL工作组维护，非常稳定，每一到两年发布一个新版本
  - 实现层面：很多编程语言对GraphQL规范的实现
  - 应用层面：后端Server、前端Client、数据库ORM等方面的实现
  - 工具链层面：包含lint、编译、测试等方便的实现

8.2 GraphQL框架

GraphQL生态

- GraphQL.js
  - 官方针对JavaScript语言提供的一个GraphQL规范参考实现

8.2 GraphQL框架

GraphQL生态

- Express GraphQL
  - 官方基于Express Web服务器提供的Graph API server实现参考

8.2 GraphQL框架

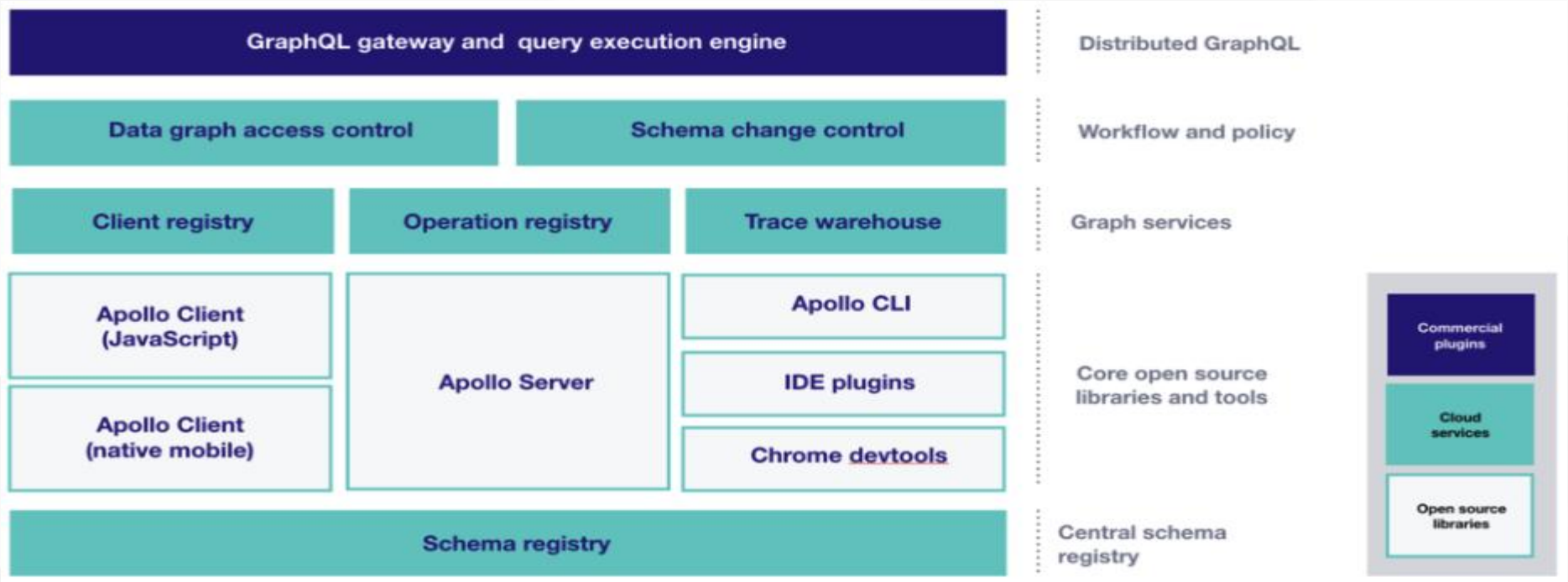
GraphQL生态

- GraphiQL
  - 官方基于浏览器提供的GraphQL IDE

8.2 GraphQL框架

GraphQL生态

- Apollo Server & Client
  - Apollo公司提供的一套GraphQL开发框架和工具，包括Server端、Client端等



8.2 GraphQL框架

GraphQL生态

- Graphql-java
  - 社区针对Java语言提供的一个GraphQL规范实现

8.2 GraphQL框架

GraphQL生态

- Graphql-go
  - 社区针对Go语言提供的一个GraphQL规范实现

8.2 GraphQL框架

GraphQL生态

- Graphene
  - 社区针对python语言提供的一个GraphQL规范实现

8.3 基于Apollo的GraphQL实战

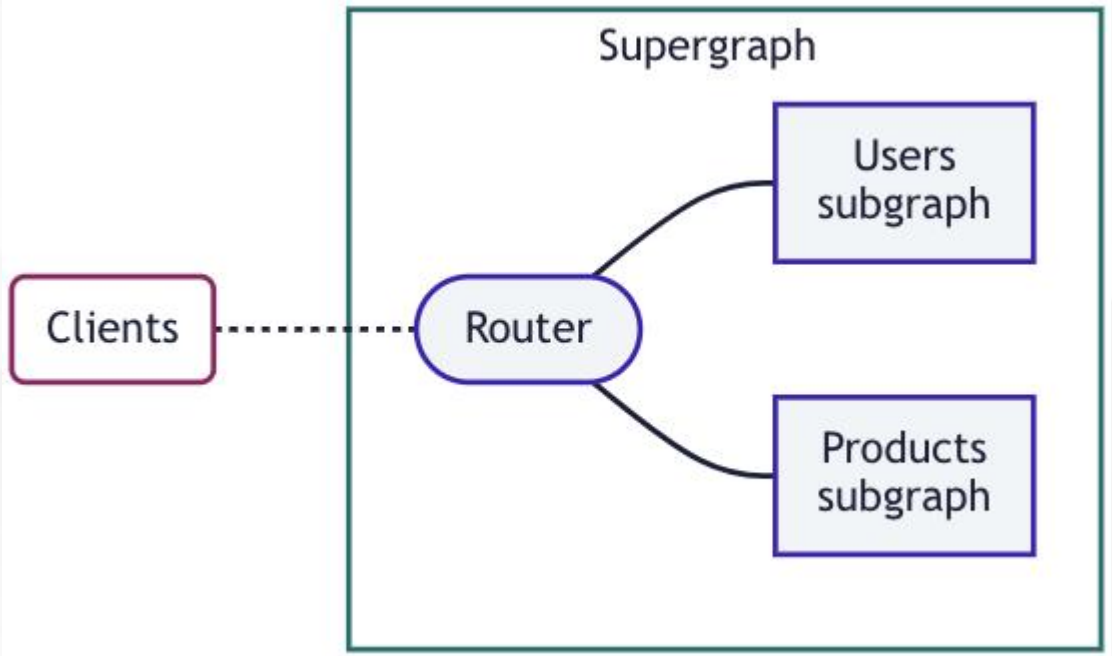
Apollo介绍

- Apollo是一个用于构建超级图的平台，它是连接到应用程序客户端的所有数据、服务和功能的统一网络。
- 官网：https://www.apollographql.com/

8.3 基于Apollo的GraphQL实战

Apollo Federation

- Apollo联邦是一种构建超图的功能强大的开放架构，它可以整合多个GraphQL API。



8.3 基于Apollo的GraphQL实战

Apollo Federation

- 工作机制
  - subgraph：单独的GraphQL API
  - 由一组subgraph构成一个supergraph
  - 客户端通过查询supergraph的router server，可以在一个请求中获取所有subgraph中的数据
  - 在Router端，可以将所有subgraph的schema进行组合

8.3 基于Apollo的GraphQL实战

Apollo Federation

- 模块技术实现
  - router模块：Apollo Router，或者apollo/gateway扩展的Apollo Server
  - subgraph模块：apollo/subgraph扩折的Apollo Server

8.3 基于Apollo的GraphQL实战

Apollo Federation

- 优势&特性
  - 实现统一图的设计理念
  - 提供一种单体系统的分解机制
  - 提供一种增量迭代的开发机制
  - 分解关注点
  - 实现托管联邦

8.3 基于Apollo的GraphQL实战

Apollo Server

- 开源的、符合规范的GraphQL服务器
  - 可与任何GraphQL客户端兼容，包括Apollo客户端
  - 构建生产级别的GraphQL API
  - 支持REST API、微服务、数据库类型的数据源

8.3 基于Apollo的GraphQL实战

Apollo客户端

- Apollo Client React
  - 针对JavaScript语言的全面状态管理库
  - 它使您能够使用GraphQL管理本地和远程数据
  - 使用它获取、缓存和修改应用程序数据，同时自动更新UI。
- Apollo iOS
  - 用于本地客户端应用程序的开源GraphQL客户端。基于Swift编写
- Apollo Kotlin
  - 提供支持从GraphQL查询生成Kotlin和Java模型

8.3 基于Apollo的GraphQL实战

案例分析与实现

8.4 GraphQL java实战

GraphQL Java介绍

- 是GraphQL规范的Java原生实现
- GraphQL Java可以看成是一个引擎层，关注GraphQL的执行查询。
- 在工程应用实践时，可结合使用graphql-java-spring 或 Spring for GraphQL，通过Spring Boot在HTTP暴露API

8.4 GraphQL java实战

GraphQL Java介绍

- GraphQL Java执行引擎
  - TypeDefinitionRegistry，schema文件的解析
  - RuntimeWiring，用于注册Datafetchers实例
  - GraphQLSchema，用于整合TypeDefinitionRegistry和RuntimeWiring
  - GraphQL是执行引擎的主入口，基于GraphQLSchema来构建
  - Datafetchers，是执行查询时，用于获取一个字段的数据

8.4 GraphQL java实战

GraphQL Java介绍

- Spring for GraphQL应用框架
  - 与GraphQL Java是同一作者，已贡献给Spring官方；
  - 基于GraphQL Java的Spring应用框架，目标是成为基于Spring的所有GraphQL应用的基石；
  - 传输协议支持HTTP、WebSocket和RSocket，其中WebSocket主要用于支持GraphQL subscription操作；
  - GraphQL控制器，通过@QueryMapping、@MutationMapping、@SubscriptionMapping注解来标识根操作。

8.4 GraphQL java实战

案例分析与实现

QA