关系型数据库 with SQL & Hasura with GraphQL& Apollo

陈宇阳　chen-yy20@mails.tsinghua.edu.cn

> 视频、课件以及作业：https://cloud.tsinghua.edu.cn/d/a8845002de1b4458b93c/
>
> 仅视频 B站：关系型数据库　　Hasura GraphQL & Apollo

关系型数据库

> 关系型数据库，是指采用了**关系模型**来组织数据的数据库，其以行和列的形式存储数据，以便于用户理解，关系型数据库这一系列的行和列被称为**表**，一组表组成了数据库。用户通过查询来检索数据库中的数据，而查询是一个用于限定数据库中某些区域的执行代码。

关系模型

数据库用以储存大量的信息，如何高效地储存信息?

如何清晰地体现信息之间的关系?

如何方便高效地实现数据的增删改查?

这就需要科学合理地设计出储存信息的数据结构。

关系模型由美国IBM公司的E.F.Codd 于1970年在《Communication of the ACM》上发表的里程碑式论文 A relational model of data for large shared data banks 中被系统而严格地提出。

关系数据结构

关系模型的基本数据结构就是**二维表**，也称为**关系**。关系模型中，现实世界的**实体**和实体间的各种**联系**都通过**关系**来表示。

元组是一个表的一行，**属性**是一个表的一列。（记录为行，字段为列）

关系数据库是表的集合，即**关系的集合**。表是一个实体集，一行代表了一个实体，它由共同表示一个的有关联的若干**属性**的值所构成。

- 例·StudentDB·:

  Class (**classNo**, className, institute, grade) ;

  Student (**StudentNo**, studentName, sex, birthday, nation, classNo) ;

  Course (**courseNo**, courseName, creditHour, *priorCourse*);

  Score (**studentNo**, **courseNo**, **term**, score);

主键（Primary Key）

关系型数据库中的一条记录中有若干个属性，若其中某一个**属性组**能**唯一**标识一条记录，该属性组就可以成为一个主键 。主键又称作主关键字。

> 可以推知，每个元组所对应的主键值是unique的。
>
> 一个元组一旦插入表中，主键最好不要再作修改，因为主键是用来唯一地定位元组的，修改了主键就会造成乱七八糟的影响。

> 推荐使用与**业务完全无关**的字符串作为主键，一般把这个属性命名为`id`。全局唯一GUID类型变量可以很好地成为id，GUID算法通过网卡MAC地址、时间戳和随机数保证任意计算机在任意时间生成的字符串都是不同的。大部分编程语言都内置了GUID算法，可以自动生成主键。

外键（Foreign Key）

如果公共属性在一个关系中是主键，那么这个**公共属性**被称为另一个关系的外键。由此可见，外键表示了两个关系之间的相关联系。以另一个关系的外键作主关键字的表被称为主表，具有此外键的表被称为主表的从表。外键又称作外关键字。

> 注：在关系模型中，我们一般讲主码（和外码），它们与主键（和外键）有琐碎的区别。需要完全解释主码和外码，需要引入笛卡尔积等数学工具，在此不多赘述。

关系完整性约束条件

- 若属性集A是关系r的**主键**，则A不能取null
- 若主键由若干个属性的集合构成，要求构成主键的每一个属性的值都不能为null
- 若属性F是关系r的**外键**，它与关系s的主键Ks相对应，则对于关系r中的每一个元组在属性F上的取值要么为空值null，要么等于关系s中某个元组的主键值。
- 引用
  - 外键=> 多对一联系的属性引用
  - 关系表=> 多对多联系的属性引用
  - 关系内部属性间的引用
- 接上StudentDB例：
- 例·StudentDB·：

  Class (**classNo**, className, institute, grade) ;

  Student (**StudentNo**, studentName, sex, birthday, nation, classNo) ;

  Course (**courseNo**, courseName, creditHour, *priorCourse*);

  Score (**studentNo**, **courseNo**, **term**, score);
  - 关系Student和Class之间存在多对一的“归属”联系（一个班由多个学生组成，一个学生只属于一个班），通过外键classNo实现联系。
  - 关系Student与Course之间存在多对多的“选修”联系。
  - 关系Score的主键是{studentNo，courseNo，term}，显然同一个学生在同一个学期不允许修读同一门课程多次。
  - studentNo，courseNo都是关系Score的外键，**分别**实现了关系Student、Course之间相互的多对一联系，也就间接实现了多对多联系。
  - 关系Course的外键priorCourse**参照**本关系的主键courseNo
- 用户自定义完整性：用户自己设计的constraint

SQL

SQL是结构化查询语言的缩写，是访问和处理关系型数据库的计算机标准语言。无论使用什么编程语言编写程序，只要是涉及到了关系型数据库的操作，必须通过SQL来完成。

> 你可能还听说过NoSQL数据库，也就是非SQL的数据库，包括MongoDB、Cassandra、Dynamo等等，它们都不是关系数据库。有很多人鼓吹现代Web程序已经无需关系数据库了，只需要使用NoSQL就可以。但事实上，SQL数据库从始至终从未被取代过。回顾一下NoSQL的发展历程：
>
> - 1970: NoSQL = We have no SQL
> - 1980: NoSQL = Know SQL
> - 2000: NoSQL = No SQL!
> - 2005: NoSQL = Not only SQL

> - 2013: NoSQL = No, SQL!
>
> 今天，SQL数据库仍然承担了各种应用程序的核心数据存储，而NoSQL数据库作为SQL数据库的补充，两者不再是二选一的问题，而是主从关系。所以，无论使用哪种编程语言，无论是Web开发、游戏开发还是手机开发，掌握SQL，是所有软件开发人员所必须的。

```sql
# 查询全体学生的学号和姓名
SELECT stu_no, stu_name
FROM Student;

# 查选修电电的学生的学号和成绩，并按分数降序排列
SELECT stu_no, grade
FROM StudentCourse
WHERE course_name='电电'
ORDER BY grade DESC;
# 在SQL中不区分大小写
```

SQL是数据库操作的基础，但软件部在日常开发的时候并不会直接使用SQL语句，而是使用GraphQL。SQL并不是培训重点，且学会基本的使用并不困难，同学们稍加自学相信就能拿下。

数据库设计

给定一个应用环境，我们要怎样才能构造出最优的数据库模式，使之能有效地储存数据，满足各种用户的应用需求?

在数据库基本结构确定下来后并储存了一定量的数据以后，想再改变数据库的结构就会变得很困难。因此，数据库的设计是极其关键的步骤。

概念模型

找出数据库使用场景中的所有**实体**和实体之间的**联系**，建立**实体-联系模型。**

- **实体**（Entity）：客观存在并可以互相区别的事物，事物的属性可以通过一些参数来表征，并拥有一个可供唯一辨识的属性。
- **联系**（Relationship）：实体内部和实体之间都会存在联系，联系即属性之间的相互关联，它们可以用一些参数来表征。

  联系之间可以是一对一的，可以是一对多的，也可以是多对多的。

例：1位校长“管理”1所学校 n位老师“工作在”1所学校 n位老师“任课在”n门课程

实体之间还存在“包含”的联系，即一个类是另一个类的子类，作为子类，除了拥有基类的所有属性外，还拥有一些额外的属性。

优化全局概念模型

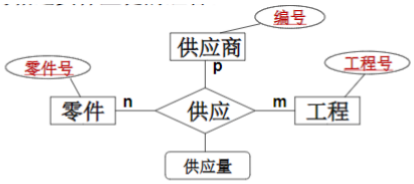
- 尽可能进行相关实体类型的合并，以减少实体类型的个数。
- 尽可能消除实体中的冗余属性。
- 尽可能消除实体间的冗余联系类型。

E-R模型向关系模型转换

如何将实体和实体间的联系转换为关系模式，又要如何确定这些关系模式的属性和键？规则如下：

1. 每种实体的类型转换为一个关系模式（关系表），实体的属性就是关系的属性，实体的键就是关系的键。键使用下划线标出。
2. 若实体之间的联系是1：n（包括1：1），在“n”端关系模式中加入“1”端实体的主键，作为其外键，“联系”本身的属性也加入“n”端关系模式。

3. 若实体的联系是m：n的多对多联系，则将联系转换为一个关系模式。“联系”多端实体的主键以及“联系”本身的属性转换为该关系的属性。该关系的主键为各实体主键的组合。



供应（ 编号，零件号，工程号，供应量 ）

4. ”isa“联系不需要作单独转换。

关系模式的规范化处理

异常

糟糕的关系模式会导致什么问题？

- 冗余储存：信息被重复储存，浪费大量储存空间
- 更新异常：重复信息的一个副本被修改，所有副本都需要进行重复的修改。因此更新数据时，系统要付出极大代价维护数据库的完整性，否则会面临数据不一致的危险。
- 插入异常：只有当一些信息事先已经存放在数据库中时，另外一些信息才能存入数据库中。
- 删除异常：删除某些信息时可能丢失其它信息。

异常的根源是关系模式中的数据依赖存在不好的性质。

范式

范式（Normal Form）是用来判断关系模式好坏的标准。基于函数依赖理论，范式可以分成四个范式，越高级别的范式越严格，高级别范式包含低级别范式。

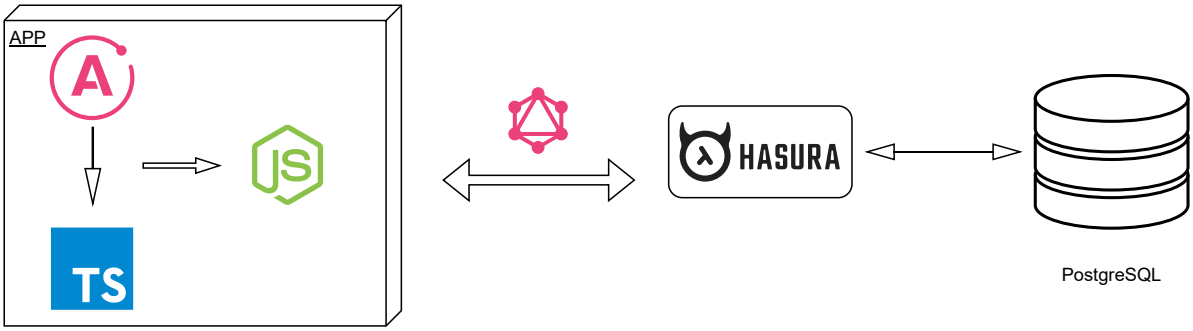
> 前置知识：
>
> **候选码：** 若关系中的某一属性组的值能唯一地标识一个元组，而其子集不能，则称该属性组为候选码。若一个关系中有多个候选码，则选定其中一个为主码。
>
> **主属性：** 所有候选码的属性称为主属性。不包含在任何候选码中的属性称为非主属性或非码属性。
>
> **函数依赖**： 设R(U)是属性集U上的关系模式，X、Y是U的子集。若对于R(U)的任意一个可能的关系r，r中不可能存在两个元组在X上的属性值相等，而在Y上的属性值不等，则称Y函数依赖于X或X函数确定Y。
>
> **完全函数依赖**： 设R(U)是属性集U上的关系模式，X、Y是U的子集。如果Y函数依赖于X，且对于X的任何一个真子集X’，都有Y不函数依赖于X’，则称Y对X完全函数依赖。记作：如果Y函数依赖于X，但Y不完全函数依赖于X，则称Y对X部分函数依赖。

- 1NF： 关系中的属性都不可再分。
- 2NF： 符合1NF，且每一个非主属性完全函数依赖于任何一个候选码。
- 3NF： 符合2NF，且任何非主属性不依赖于其它非主属性。
- BCNF：符合3NF，且每一个决定因素中都包含候选码。

关于范式的进一步解释和理解，感兴趣的同学可以进一步深入查询。

Hasura with GraphQL

> 数据库如何与网页前端进行连接，并互相进行通信，响应请求呢？目前EESAST的网站拉取数据的方案是编写查询相关的 GraphQL 语句，Apollo 会生成相应的 TS 代码，经编译后运行在 Node.js 上，使用 GraphQL 的形式与 Hasura 进行交互，由 Hasura 访问 PostgreSQL 数据库存取数据。



GraphQL

SQL 较为古老，对于数据量大且关系复杂的查询不够直观，GraphQL 则是一个更为直观的查询语言。

在 JS / json 中，我们使用类似的结构来表示对象

```
[
  {
    "username": "Zhang San",
    "gender": "male"
  },
  {
    "username": "Nagisa",
    "gender": "female"
  }
]
```

我们可以使用极为相似的 GraphQL 语句来查询

```
query {
  user {
    username
  }
}
```

得到相应的 json 结果

```
{
  "data": {
    "user": [
      {
        "username": "Zhang San"
      },
      {
        "username": "Nagisa"
      }
    ]
  }
}
```

查询语句和查询结果的显示都与json格式类似，方便了我们对查询结果的调用。实际上，我们写GraphQL查询语句时并不会直接干写，而是借用hasura 的GraphQL Engine来生成GraphQL语句。

安装

参照官方文档，我们使用 docker 进行安装。首先修改官方提供的 `docker-compose.yaml`

```yaml
version: "3.6"
services:
  postgres:
    image: postgres:latest
    container_name: postgres
    restart: always
    volumes:
      - ~/data/postgres:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: postgrespassword
  graphql-engine:
    image: hasura/graphql-engine:latest
    container_name: hasura
    ports:
      - "23333:8080"
    depends_on:
      - "postgres"
    restart: always
    environment:
      HASURA_GRAPHQL_DATABASE_URL: postgres://postgres:postgrespassword@postgres:5432/postgres
      HASURA_GRAPHQL_ENABLE_CONSOLE: "true" # set to "false" to disable console
      HASURA_GRAPHQL_ENABLED_LOG_TYPES: startup, http-log, webhook-log, websocket-log, query-log
      ## uncomment next line to set an admin secret
      # HASURA_GRAPHQL_ADMIN_SECRET: myadminsecretkey
```

注意其中需要修改的几处地方：

- 容器端口绑定宿主机端口是否会冲突
- PostgreSQL 的密码是否需要设置
- Hasura 的密码是否需要设置（默认不设置）

在修改好后的 `docker-compose.yaml` 所在路径使用如下命令拉取镜像并启动容器

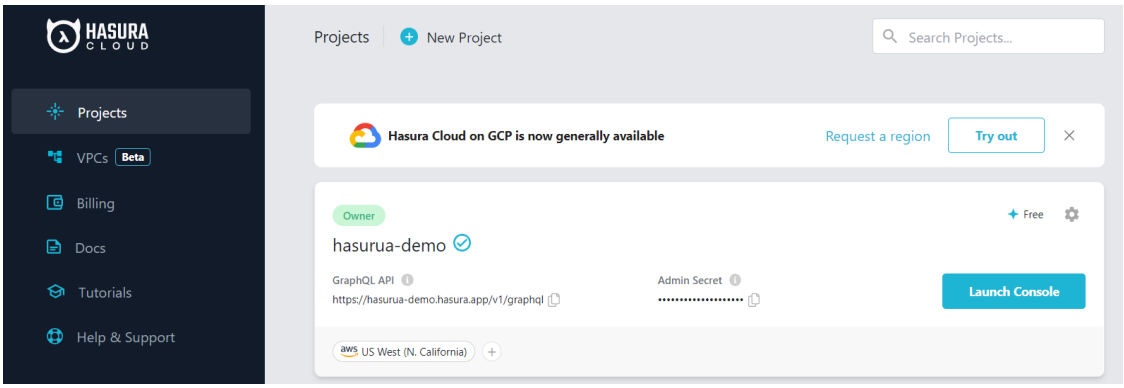
```
docker-compose up -d
```

启动后，我们使用 `docker ps -a` 可以查看容器状态（类似下方所示）

```
CONTAINER ID IMAGE                ... CREATED STATUS PORTS
     ...
097f58433a2b hasura/graphql-engine ... 1m ago  Up 1m  0.0.0.0:23333->8080/tcp
     ...
b0b1aac0508d postgres             ... 1m ago  Up 1m  0.0.0.0:5432->5432/tcp
      ...
```

现在访问 `localhost:23333/console` 可以进入 Hasura 的控制界面

> 未学习docker的同学可以先使用Projects - Hasura Cloud来运行hasura console进行实例的尝试。



Hasura实例

> 具体基本操作敬请观看录屏，更多操作参考Hasura GraphQL Engine Documentation | Hasura GraphQL Docs

- 创建Table
- 插入数据
- 设置外键
- 使用GraphQL查询、修改数据
- 添加关系
- 权限设置

> 目前 Hasura 提供的权限设置自由度较小，如果要设置非常复杂的权限，需要通过添加数据表的`view`再给其设置权限等方法来实现。具体细节可以参看Multiple column + row permissions for the same role

Apollo

我们这里说的 Apollo 主要指的是 Apollo Client (React)，实现了拉取并管理数据的功能。Apollo 本身会根据配置好的服务端的数据，将本地写好的`GraphQL`语句编译为`typescript`接口。

目前的前端项目中，我们在`src/api`文件夹下编写`*.graphql`文件，即填写在 Hasura 中测试过得到需要结果的`query`或`mutation`语句。之后使用 Apollo 的`codegen`编译为相应的接口，供前端页面拉取数据使用。

我们直接以当前EESAST网站的前端web仓库作为例子，向大家展示Apollo如何连接数据库与react页面。

apollo通过`yarn codegen`把`contest.graphql`中的GraphQL语句转化为`type.ts`中的各种`interface`,注意这个`type.ts`是自动生成的，请不要直接修改它，而是通过修改GraphQL来修改`interface`。

我们在`react`页面中import`type.ts`中的`interface`，使用apollo的`useQuery`和`useMutation`语句即可拉取数据到前端。

> 使用`./src/api/contest.graphql`和`./src/api/type.ts`
> 和`./src/pages/ContestSite/JoinPage.tsx`作为例子，详见录屏。