第3章

数据库操作

学习目标：

- ❖ 了解什么是 SQL。
- ❖ 掌握数据库的基本操作。
- ❖ 掌握模式的使用。
- ❖ 掌握数据表的基本操作。
- ❖ 掌握数据插入、修改、删除、查询等操作。

数据库操作是学习任何数据库管理系统都必不可少的部分，贯穿于数据的创建、查询、更新和维护等多个阶段。有效的数据库操作不仅能确保数据的准确性和一致性，还能提高系统的性能和用户的满意度。本章将围绕数据库操作进行详细讲解。

3.1 SQL 简介

SQL 是 Structured Query Language(结构化查询语言)的缩写，它是用于管理关系数据库系统的标准化语言，可以用于执行各种任务，包括数据定义、数据操纵、数据查询和数据控制。

SQL 由 4 个主要部分组成，具体说明如下。

(1) 数据定义语言(Data Definition Language，DDL)。

DDL 用于管理数据库的结构，包括创建、修改和删除数据库对象(如表、索引、视图等)。常见的 DDL 命令包括 CREATE、ALTER 和 DROP 等。

(2) 数据操纵语言(Data Manipulation Language，DML)。

DML 用于在数据库中执行操作，包括插入、更新和删除数据。常见的 DML 命令包括 INSERT、UPDATE 和 DELETE 等。

(3) 数据查询语言(Data Query Language，DQL)。

DQL 用于从数据库中检索数据。它的主要命令是 SELECT，它允许用户指定要检索的列以及检索条件。DQL 用于执行诸如数据查询和报告生成等任务。

(4) 数据控制语言(Data Control Language,DCL)。

DCL 用于控制数据库访问权限和安全性,包括授权用户访问数据库、撤销访问权限以及管理数据库用户。常见的 DCL 命令包括 GRANT 和 REVOKE 等。

SQL 是一种强大而灵活的语言,被广泛用于各种数据库系统,如 MySQL、PostgreSQL、Oracle、Microsoft SQL Server、openGauss 等。掌握 SQL 是进行数据库管理和开发工作的基本要求之一。

3.2 数据库的基本操作

3.2.1 数据库的定义

数据库(Database)是按照数据结构来组织、存储和管理数据的仓库。它们存在于计算机系统中,旨在方便用户高效地存取信息。数据库的主要目的是帮助用户存储和查询数据信息。数据库的使用范围非常广泛,包括但不限于商业、科研、教育和工程等领域。简单来说,可以把数据库理解成一个电子化的文件柜,用于存储电子文件,用户可以对文件中的数据进行新增、修改、删除等操作。

总的来说,数据库是一个用于存储、管理和操作数据的系统,它为应用程序提供了一个结构化的数据存储和管理环境,帮助用户有效地管理和利用数据。

3.2.2 创建数据库

在 openGauss 中,创建数据库可以通过 Navicat 可视化工具直接创建数据库,也可以通过 SQL 语句的形式创建数据库。在使用 Navicat 创建数据库时,选中默认的数据库 postgres,右击,在右键菜单中选择“新建数据库”就可以创建一个新的数据库,包括创建模式、创建数据表也是同样的方式。

为了让读者更好地学习 SQL 语法,本书全部采用 SQL 语句的形式来对数据库进行操作,所以首先需要打开命令列界面。可以选中 postgres 数据库,右击选择“命令列界面”打开该界面,如图 3-1 所示。

创建数据库的基本命令是使用 CREATE DATABASE 语句,其语法格式如下。

```
CREATE DATABASE 数据库名;
```

例如,要创建一个名为 schooldb 的数据库,可以执行如下命令。

```
CREATE DATABASE schooldb;
```

在 Navicat 工具中,输入对应的 SQL 命令,输入完成后,按 Enter 键,便可以自动执行 SQL 语句,如图 3-2 所示。

在创建数据库时,还可以指定一些其他选项,如所有者、字符集、校对规则等。例如,创建一个具有特定所有者和 UTF-8 字符集的数据库,可以执行如下命令。

```
CREATE DATABASE schooldb
OWNER testuser
ENCODING 'UTF8'
```

在上述命令中,创建了一个名为 schooldb 的数据库,指定 testuser 为所有者,设置编码

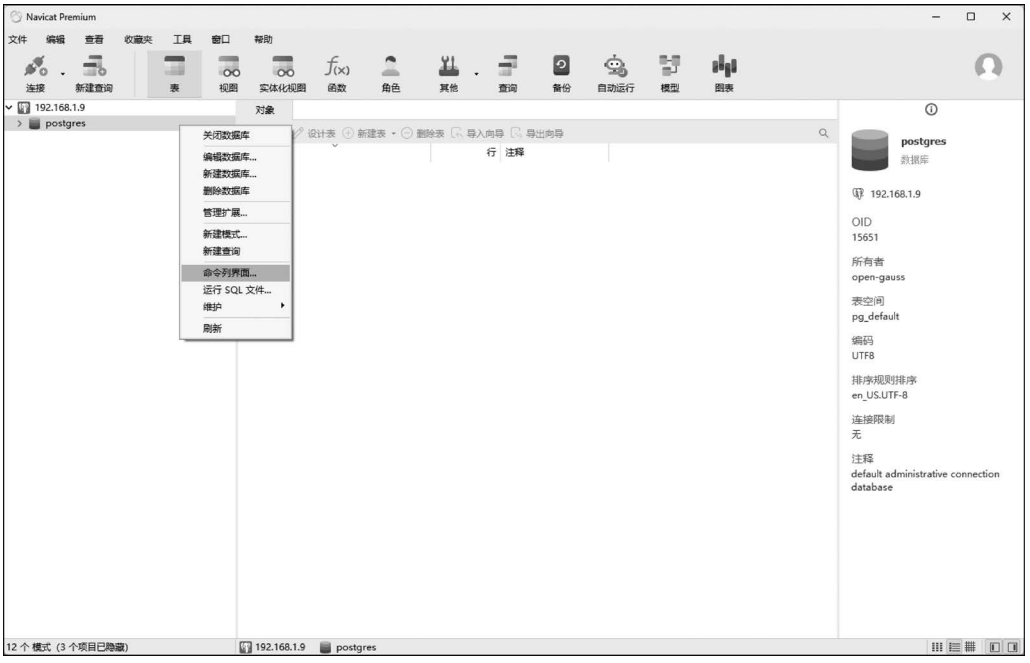

图 3-1　打开命令列界面

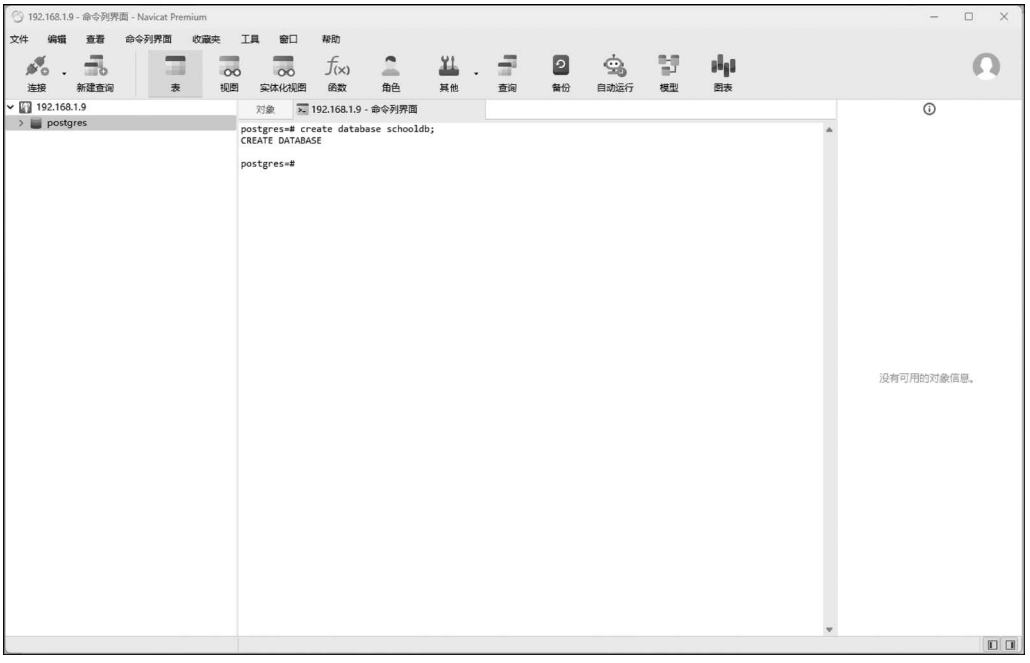

图 3-2　创建数据库

为 UTF-8。

数据库创建完成后，在 Navicat 中刷新后才可以看到，此时选中 postgres 数据库，右击，在右键菜单中选择“刷新”即可，如图 3-3 所示。

还可以通过 SQL 语句的形式来查看数据库是否创建成功，具体命令如下。

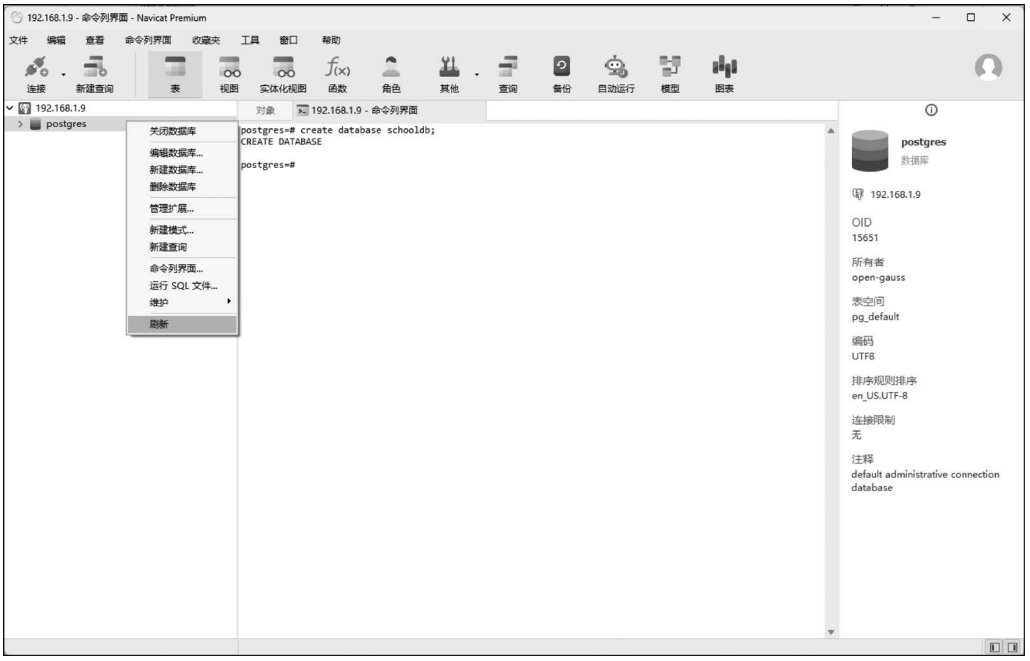

图 3-3　刷新数据库

```
1.    SELECT datname FROM pg_database;
```

在上述语法格式中，pg_database 这个系统目录表用于存储数据库的信息，通过查询该表中的信息便可以查询到所有数据库。

为了更好地看出执行效果，在 Navicat 工具中执行查看数据库的 SQL 语句，如图 3-4 所示。

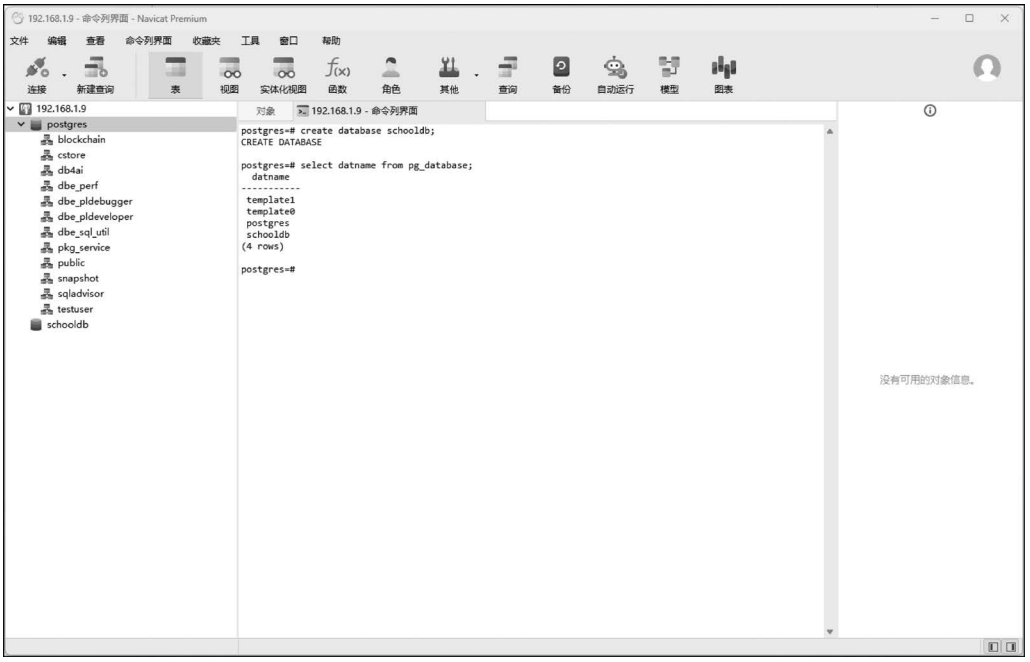

图 3-4　查看数据库

从图 3-4 中可以看出，pg_database 表是包括 schooldb 数据库信息的，说明数据库创建成功了。

3.2.3 修改数据库

在 openGauss 中，修改数据库通常指的是更改数据库的配置或属性，而不是修改数据库中的数据。这些修改可以包括改变数据库的所有者、重命名数据库、调整数据库的编码设置等。以下是进行一些常见数据库修改操作的介绍。

1. 重命名数据库

要重命名一个数据库，可以使用 ALTER DATABASE 命令配合 RENAME TO 选项。例如，将 schooldb 重命名为 newschooldb，可以使用如下命令。

```
1.    ALTER DATABASE schooldb RENAME TO newschooldb;
```

注意，在进行重命名操作时，应确保没有用户连接到该数据库。

2. 更改数据库所有者

如果想要更改数据库的所有者，同样可以使用 ALTER DATABASE 命令配合 OWNER TO 选项。例如，要将数据库 schooldb 的所有者更改为 testuser1（testuser1 必须存在于数据库中），可以使用如下命令。

```
1.    ALTER DATABASE schooldb OWNER TO testuser1;
```

需要注意的是，在执行操作时，应确保数据库没有被其他用户使用，特别是在执行诸如重命名数据库这类操作时。修改数据库的某些属性，如编码，可能需要更高权限的用户或数据库管理员权限。

3.2.4 删除数据库

在 openGauss 数据库系统中，删除数据库是一个不可逆操作，它会永久移除数据库及其包含的所有数据。因此，在执行删除操作之前，要确保已经备份了所有重要数据，并确认不再需要这个数据库。下面是删除数据库的步骤。

在删除数据库时，可以使用 DROP DATABASE 命令，其语法格式如下。

```
1.    DROP DATABASE database_name;
```

在上述语法中，database_name 是指要删除的数据库名称。假如想要删除名为 schooldb 的数据库，可以使用如下语句。

```
1.    DROP DATABASE schooldb;
```

需要注意的是，在执行 DROP DATABASE 命令之前，应确保没有任何用户连接到该数据库。openGauss 不允许在数据库正在被访问时将其删除。如果有用户连接到数据库，需要先关闭连接让这些用户断开连接。只有数据库的所有者或具有相应权限的用户才能删除数据库。考虑到删除数据库是不可逆的，务必三思而后行。

3.3 模式

openGauss 数据库基于 PostgreSQL 发展而来的，因此很多概念和操作与 PostgreSQL 相似，包括对模式（Schema）的支持和使用。在 openGauss 数据库中，模式的概念和作用与

PostgreSQL 中的相同，用作数据库内部对象的逻辑分组，便于管理和组织数据。

1. 模式分类

（1）public 模式（默认）：openGauss 在新创建的数据库中默认包含一个名为 public 的模式。这意味着，如果没有指定模式来创建数据库对象（如表、视图等），这些对象将默认被创建在 public 模式下。这种默认设置使得初次使用数据库的用户可以直接创建和管理数据，不需要额外的配置步骤。

（2）自定义模式：尽管 public 模式对于简单应用来说可能足够使用，但在更复杂或更具有组织性的数据库设计中，创建自定义模式是一种常见做法。自定义模式可以在逻辑上分组数据库对象，提高数据的管理效率和安全性。

2. 模式的作用

（1）充当命名空间：模式在数据库内部充当命名空间的角色，允许在不同的模式下创建名称相同的对象，有助于避免名称冲突。

（2）管理和隔离数据：通过使用不同的模式，可以更好地管理和隔离数据。例如，为不同的项目、应用或团队创建不同的模式，并对它们应用特定的权限设置。这有助于保护数据，避免未授权访问，并使数据库结构更加清晰。

（3）权限控制：自定义模式还可以更细致地控制数据库对象的访问权限，通过为特定模式或模式中的对象分配权限，来限制用户对数据的操作，确保数据安全。

3. 创建模式

创建模式的基本 SQL 语法与 PostgreSQL 非常相似，其语法格式如下。

```
1.    CREATE SCHEMA schema_name;
```

其中，schema_name 是模式的名称。例如，要想创建一个模式名为 my_schema，可以使用如下命令。

```
1.    CREATE SCHEMA my_schema;
```

需要注意的是，创建模式要切换到指定的数据库下进行操作，这里切换到 schooldb 数据库下进行操作，双击 schooldb 数据库打开该数据库，然后右击打开“命令列界面”，在该数据库中进行命令操作，如图 3-5 所示。

在创建模式时，还可以指定所有者，其语法格式如下。

```
1.    CREATE SCHEMA schema_name AUTHORIZATION owner_name;
```

例如，指定模式的所有者为 testuser，可以使用如下命令。

```
1.    CREATE SCHEMA my_schema AUTHORIZATION testuser;
```

此外，还可以为用户设置默认的搜索模式，这样在引用数据库对象时就不需要每次都指定模式名称了，后续操作更加方便，其语法格式如下。

```
1.    ALTER USER user_name SET search_path TO schema_name;
```

为了后续使用方便，这里就为 testuser 指定默认的搜索模式为 my_schema，具体命令如下。

```
1.    ALTER USER testuser SET search_path TO my_schema;
```

需要注意的是，在 Navicat 中，无论是创建数据库、模式、数据表，创建完成后都需要刷新一下才能够在界面中显示，否则是不显示的。

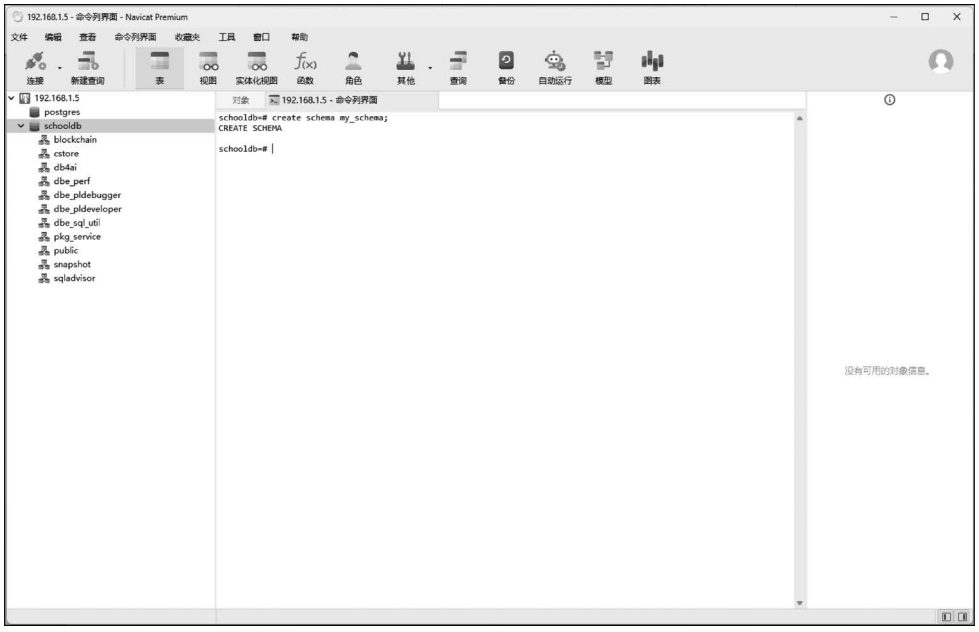

图 3-5　创建模型

3.4　数据类型

在数据库系统中，数据类型是一个重要概念，它定义了存储在数据库中的数据可以采取的形式和操作。每个数据库管理系统都提供了一组数据类型，用于优化存储、检索和操作数据。以下是一些最常见的数据类型及其详细说明。

3.4.1　数值类型

openGauss 的数值类型又可细分为整数类型和任意精度类型两种，接下来分别针对这两种类型进行详细讲解。

（1）整数类型，如表 3-1 所示。

表 3-1　整数类型

名　称	描　述	存储空间	范　围
TINYINT	微整数，别名为 INT1	1B	0～255
SMALLINT	小范围整数，别名为 INT2	2B	－32 768～＋32 767
INTEGER	常用的整数，别名为 INT4	4B	－2 147 483 648～＋2 147 483 647
BINARY_INTEGER	常用的整数 INTEGER 的别名	4B	－2 147 483 648～＋2 147 483 647
BIGINT	大范围的整数，别名为 INT8	8B	－9 223 372 036 854 775 808～＋9 223 372 036 854 775 807
int16	16B 的大范围整数，目前不支持用户用于建表等使用	16B	－170 141 183 460 469 231 731 687 303 715 884 105 728～＋170 141 183 460 469 231 731 687 303 715 884 105 727

（2）任意精度类型，如表 3-2 所示。

表 3-2　任意精度类型

名　称	描　述	存储空间	范　围
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。 p 为总位数，s 为小数位数	用户声明精度。每 4 位（十进制位）占用 2B，然后在整个数据上加上 8B 的额外开销	未指定精度的情况下，小数点前最大 131 072 位，小数点后最大 16 383 位
NUMBER[(p[,s])]	NUMERIC 类型的别名	用户声明精度。每 4 位（十进制位）占用 2B，然后在整个数据上加上 8B 的额外开销	未指定精度的情况下，小数点前最大 131 072 位，小数点后最大 16 383 位

在整数类型中，常用的类型称为 INTEGER，即整数。因为该类型提供了在范围、存储空间、性能之间的一个最佳平衡，一般只有取值范围确定不超过小范围整数的情况下，才会使用小范围整数类型，而只有在 INTEGER 常用整数的范围不够时，与整数类型相比，任意精度类型需要更大的一个存储空间，其存储效率、运算效率以及压缩比效果都要差一些，那么在进行数值类型定义时，只有当整数类型不满足需求时，才选择任意精度类型。

3.4.2　字符类型

字符类型是用来存储文本数据的一种数据类型，对于处理和存储各种文本信息至关重要。在 openGauss 中存储文本数据时，要关注几个常用的字符数据类型存储文本数据时的特点和用途。下面是 openGauss 数据库中常见的字符类型及其描述，如表 3-3 所示。

表 3-3　字符类型

名　称	描　述	存储空间
CHAR(n) CHARACTER(n) NCHAR(n)	定长字符串，不足补空格。n 是指字节长度，如不带精度 n，默认精度为 1	最大为 10MB
VARCHAR(n) CHARACTER VARYING(n)	变长字符串。PG 兼容模式下，n 是字符长度。其他兼容模式下，n 是指字节长度	最大为 10MB
VARCHAR2(n)	变长字符串。是 VARCHAR(n)类型的别名。n 是指字节长度	最大为 10MB
NVARCHAR2(n)	变长字符串。n 是指字符长度	最大为 10MB
NVARCHAR(n)	变长字符串。是 NVARCHAR2(n)类型的别名。n 是指字符长度	最大为 10MB
TEXT	变长字符串	最大为 1GB－1，但还需要考虑到列描述头信息的大小，以及列所在元组的大小限制（也小于 1GB－1），因此 TEXT 类型最大大小可能小于 1GB－1
CLOB	文本大对象。是 TEXT 类型的别名	最大为 1GB－1，但还需要考虑到列描述头信息的大小，以及列所在元组的大小限制（也小于 1GB－1），因此 CLOB 类型最大大小可能小于 1GB－1

3.4.3 日期和时间类型

在数据库中，日期和时间类型是用于存储与日期和时间相关的数据的数据类型，这些类型对于记录事件发生的时间、用户行为的时间戳、历史数据的保存等场景至关重要。openGauss 数据库提供了多种日期和时间类型，以满足不同的应用需求，如表 3-4 所示。

表 3-4 日期和时间类型

名 称	描 述	存储空间
DATE	日期和时间	4B
TIME[(p)][WITHOUT TIME ZONE]	只用于一日内时间。 p 表示小数点后的精度，取值范围为 0～6	8B
TIME[(p)] [WITH TIME ZONE]	只用于一日内时间，带时区。 p 表示小数点后的精度，取值范围为 0～6	12B
TIMESTAMP[(p)] [WITHOUT TIME ZONE]	日期和时间 p 表示小数点后的精度，取值范围为 0～6	8B
TIMESTAMP[(p)] [WITH TIME ZONE]	日期和时间，带时区。TIMESTAMP 的别名为 TIMESTAMPTZ。 p 表示小数点后的精度，取值范围为 0～6	8B
SMALLDATETIME	日期和时间，不带时区。 精确到分钟，秒位大于或等于 30 秒时进一位	8B
INTERVAL DAY (l) TO SECOND (p)	时间间隔，X 天 X 小时 X 分 X 秒。 l：天数的精度，取值范围为 0～6。兼容性考虑，目前未实现具体功能。 p：秒数的精度，取值范围为 0～6。小数末尾的零不显示	16B
INTERVAL[FIELDS][(p)]	时间间隔。 FIELDS：可以是 YEAR、MONTH、DAY、HOUR、MINUTE、SECOND、DAY TO HOUR、DAY TO MINUTE、DAY TO SECOND、HOUR TO MINUTE、HOUR TO SECOND、MINUTE TO SECOND。 p：秒数的精度，取值范围为 0～6，且 FIELDS 为 SECOND、DAY TO SECOND、HOUR TO SECOND 或 MINUTE TO SECOND 时，p 才有效。小数末尾的零不显示	12B
reltime	相对时间间隔。格式为 X years X mons X days XX:XX:XX。 采用儒略历计时，规定一年为 365.25 天，一个月为 30 天，计算输入值对应的相对时间间隔，输出采用 POSTGRES 格式	4B
abstime	日期和时间。格式为 YYYY-MM-DD hh:mm:ss＋timezone 取值范围为 1901-12-13 20:45:53 GMT～2038-01-18 23:59:59 GMT，精度为秒	4B

3.4.4 布尔类型

在数据库系统中，布尔类型用于存储真或假的值，通常用于表示二元状态、条件判断或启用/禁用状态等。布尔类型在逻辑运算和流程控制中非常有用，能够有效地简化数据表示和增强可读性，如表 3-5 所示。

表 3-5 布尔类型

名 称	描 述	存储空间	取 值
BOOLEAN	布尔类型	1B	true：真 false：假 null：未知(unknown)

3.5 数据表的基本操作

3.5.1 数据表的定义

数据表是由表名、表中的字段和表中的记录三部分组成。设计数据表结构的过程就是定义表名、确定表中包含的字段以及各字段的属性(字段名、字段类型、字段长度等)。这个过程是数据库设计中非常重要的一步，因为它直接影响到数据的存储、检索和管理效率，以及数据的完整性和一致性。

- 表名：表名是数据表的唯一标识符，用于在数据库中引用该表。一个好的表名应该简明扼要地描述表所存储的数据内容或用途。
- 字段：字段是数据表中的列，用于存储表中的数据。每个字段都有一个字段名，用于标识该字段，以及一个数据类型，用于定义该字段可以存储的数据类型。常见的数据类型包括整型、字符型、日期时间型等。此外，还可以定义字段的长度、精度、默认值等属性。
- 记录：记录是数据表中的行，每一行存储了一组相关的数据，每个字段对应一列数据。记录的数量取决于表中实际存储的数据量。

通过设计良好的数据表结构，可以有效地组织和存储数据，提高数据的检索效率，并确保数据的完整性和一致性。在设计过程中，需要考虑到数据的需求和业务逻辑，合理地选择字段和数据类型，设计出符合实际应用场景的数据表结构。

3.5.2 创建数据表

在 openGauss 数据库中，创建数据表是数据库管理的基础操作之一，用于存储结构化数据。数据表由列(字段)和行(记录)组成，每列有其特定的数据类型和约束条件。可以使用 CREATE TABLE 语句来创建一个新的数据表，定义表的名称、列及其数据类型和可能的约束条件(如主键、唯一性约束、非空约束等)。

数据表需要在指定模式中创建，如果未指定模式则自动进入默认的 public 模式中，创建数据表的语法格式如下。

```
1.    CREATE TABLE schema_name.table_name (
2.        column1 datatype,
3.        column2 datatype,
4.        ...
5.    );
```

上述语法格式中，schema_name 表示模式名称，table_name 表示数据表名称，column1 表示列名，datatype 表示数据类型。

由于3.3节中已经指定了testuser用户默认搜索模式为my_schema，因此在创建表时，默认就会在my_schema模式下创建，可以省略my_schema。

接下来在schooldb数据库my_schema模式中，创建三个数据表，分别为学生表students、课程表courses和选课表enrollments。

1. 创建学生表students

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    major VARCHAR(100),
    email VARCHAR(255)
);
```

上述语句中，student_id作为主键，用于唯一标识每位学生，SERIAL关键字用于自动生成递增的整数，name、age、major和email分别存储学生的姓名、年龄、专业、电子邮箱。

2. 创建课程表courses

```
CREATE TABLE courses (
    course_id SERIAL PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL,
    credits INT
);
```

上述语句中，course_id是课程的主键，每门课程有一个唯一的标识，course_name和credits分别存储课程的名称和学分。

3. 创建选课表enrollments

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    grade INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

上述语句中，student_id和course_id是外键，分别引用students表和courses表的主键，表示哪位学生选了哪门课，grade存储学生在该课程中的成绩。

数据表创建完成后，在Navicat中刷新模式就可以看到新创建的数据表，同时也可以通过SQL语句进行查验，具体语句如下。

```
SELECT * FROM information_schema.tables
WHERE table_schema = 'my_schema' AND table_name = 'students';
```

上述语句中，数据表information_schema.tables是一个标准的信息模式视图，包含数据库中所有表的信息，因此可以通过该表进行查询。

3.5.3 修改数据表

在openGauss中，修改数据表通常涉及添加、删除或修改列，以及修改表的约束等操作。通过ALTER TABLE语句可以修改数据表，以下是一些基本的SQL命令，用于执行

这些常见的修改操作。

1. 添加列

添加新列到现有表中。例如,向 students 表添加一个名为 birthdate 的日期类型列,可以使用如下命令。

```
ALTER TABLE students ADD COLUMN birthdate DATE;
```

2. 删除列

删除表中的列。例如,从 students 表中删除 birthdate 列,可以使用如下命令。

```
ALTER TABLE students DROP COLUMN birthdate;
```

3. 修改列的数据类型

如果需要修改列的数据类型,例如将 students 表中的 email 列的数据类型从 VARCHAR(255)修改为 VARCHAR(100),可以使用如下命令。

```
ALTER TABLE students ALTER COLUMN email TYPE VARCHAR(100);
```

4. 重命名列

列的名称也可以更改。例如,将 students 表中的 student_id 列重命名为 stu_id,可以使用如下命令。

```
ALTER TABLE students RENAME COLUMN student_id TO stu_id;
```

5. 添加约束

向表中添加约束,如唯一约束、检查约束(CHECK)或外键约束。例如,为 email 列添加唯一约束,确保表中不会有重复的电子邮件地址,可以使用如下命令。

```
ALTER TABLE students ADD CONSTRAINT email_unique UNIQUE (email);
```

6. 删除约束

也可以删除已有的约束。例如,删除上面添加的 email_unique 约束,可以使用如下命令。

```
ALTER TABLE students DROP CONSTRAINT email_unique;
```

3.5.4 删除数据表

删除数据表在数据库管理中是一个常见的操作,用于移除整个表及其包含的所有数据。在 openGauss 中,删除数据表可以通过执行 DROP TABLE 语句来完成。注意,这是一个不可逆的操作。

1. 删除数据表

要删除一个名为 students 的数据表,可以使用如下命令。

```
DROP TABLE students;
```

2. 删除多个数据表

如果想一次性删除多个表,可以在同一 DROP TABLE 语句中列出所有想要删除的表名,各表名之间用半角逗号分隔。例如,删除 students 和 courses 两个表,可以使用如下命令。

```
DROP TABLE students, courses;
```

3. 使用 IF EXISTS 语法

为了避免在尝试删除不存在的表时出错，可以在 DROP TABLE 语句中使用 IF EXISTS 语法。这样，如果指定的表不存在，openGauss 将不会报错，而是生成一条警告消息。这对于脚本编写和自动化任务特别有用。使用 IF EXISTS 删除 students 表，可以使用如下命令。

```
DROP TABLE IF EXISTS students;
```

需要注意的是，在执行 DROP TABLE 操作时，任何依赖于该表的对象（如视图、存储过程、外键约束等）都可能会受到影响。应确保在删除表之前了解这些依赖关系，并相应地管理它们。同时，在生产环境中，通常建议使用 IF EXISTS 先检查表是否存在，以避免因表不存在而导致的错误。

3.6 约束

在 openGauss 数据库中，约束是用于限制表中的数据，新行或者更新的行必须满足这些约束才能成功插入或更新，以确保数据的完整性、准确性和可靠性。约束可以在创建表时规定，或者在表创建之后规定。接下来将针对约束进行详细讲解。

3.6.1 非空约束（NOT NULL）

NOT NULL 表示非空约束，用于指定列不能存储 NULL 值。在创建表时如果不指定该约束，则默认值为 NULL，即允许列插入空值。这里的 NULL 与没有数据不同，它代表着未知的数据。

例如，前面创建的 students 表，name 字段设置了非空约束，不接受空值。

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    major VARCHAR(100),
    email VARCHAR(255)
 );
```

如果此时给 students 插入数据，当 name 字段插入空值时，数据库就会返回报错信息。

3.6.2 唯一约束（UNIQUE）

UNIQUE 表示唯一约束，用于确保某列或某组列的值是唯一的，但允许有 NULL 值。

例如，在 students 表中，可以为 email 字段设置 UNIQUE 约束，此时不能添加两条相同 email 的记录，否则会报错。

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    major VARCHAR(100),
    email VARCHAR(255) UNIQUE
);
```

3.6.3 主键约束(PRIMARY KEY)

PRIMARY KEY 为主键,是数据表中每一条记录的唯一标识。主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。主键是非空约束和唯一约束的组合。一个表只能声明一个主键。

例如,students 表中,student_id 为主键,唯一标识数据库表中的每一行数据。

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    major VARCHAR(100),
    email VARCHAR(255)
);
```

在上述命令中,还将 SERIAL 和 PRIMARY KEY 结合使用,SERIAL 是一个伪数据类型,它创建了一个自增的整数列。在一个列上定义 SERIAL 类型时,openGauss 会为这个列创建一个序列(sequence),并在插入新记录时自动从该序列中获取下一个值。此外,SERIAL 还隐式地包括 NOT NULL 约束,因为自增列不能包含 NULL 值。

3.6.4 外键约束(FOREIGN KEY)

FOREIGN KEY 即外键约束,外键约束用于保证一个表中的数据匹配另一个表中的值的参照完整性。外键是表中的一列,其值必须在另一个表的主键列中有对应的值。

例如,前面创建的 enrollments 表,其中,student_id 就是外键,与 students 表中的 student_id 对应。

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    grade INT ,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

3.6.5 检查约束(CHECK)

CHECK 约束声明一个布尔表达式,检查约束保证列中的值符合一定条件。例如,可以确保年龄列的值大于 0。

例如,在 students 表中,对 age 字段新增检查约束,确保插入的年龄都大于 0。

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT CHECK (age > 0),
    major VARCHAR(100),
    email VARCHAR(255)
);
```

3.7 数据操作

3.7.1 数据插入

数据插入是数据库管理中的一个基本操作，用于向数据库表中添加新的数据行。数据插入通过 INSERT INTO 语句完成，该语句允许指定要插入数据的表和列，以及要插入的数据值。

1. 基本语法

```
INSERT INTO table_name (column1, column2, column3, …)
VALUES (value1, value2, value3, …);
```

上述语法中，table_name 表示要插入数据的目标表名。(column1，column2，column3，…)是表中要插入数据的列名，列名之间用半角逗号分隔。列名是可选的，但如果指定了列名，VALUES 中的值必须与列对应且顺序一致。VALUES (value1，value2，value3，…)是表中要插入指定列的数据值。每个值与前面指定的列一一对应。如果列名列表省略，则默认为向表的所有列插入数据，这时 VALUES 中的值顺序必须与表中列的定义顺序一致。

接下来针对数据的操作部分，均以 students 表为例，该表包含 student_id(学生 ID，自动增长)、name(姓名)、age(年龄)、major(专业)、email(电子邮件)字段。

2. 插入一条数据

```
INSERT INTO students (name, age, major, email)
VALUES ('John Doe', 20, 'Computer Science', 'john.doe@gmail.com');
```

3. 插入多条数据

在许多数据库系统中，可以一次性插入多行数据，方法是在 VALUES 子句中提供多组值，每组值之间用逗号分隔，这里一次性插入三行数据，具体语句如下。

```
INSERT INTO students (name, age, major, email)
VALUES
('Jane Smith', 22, 'Mathematics', 'jane.smith@gmail.com'),
('Alex Johnson', 20, 'Physics', 'alex.johnson@gmail.com'),
('Maria Garcia', 21, 'Chemistry', 'maria.garcia@gmail.com');
```

为了方便后续多表查询数据的使用，这里分别为 courses 表和 enrollments 表提前插入数据。首先为 courses 表插入数据，可以使用如下语句。

```
INSERT INTO courses (course_name, credits)
VALUES
('Introduction to Computer Science', 4),
('Advanced Mathematics', 3),
('General Physics', 4),
('Organic Chemistry', 3);
```

接下来为 enrollments 表插入数据。为 enrollments 表插入数据时，需要知道关联的学生 ID 和课程 ID。这里，students 表中 John Doe 的 student_id 是 1，Jane Smith 的 student_id 是 2。courses 表中，Introduction to Computer Science 的 course_id 是 1，Advanced Mathematics 的 course_id 是 2，插入数据如下。

```
INSERT INTO enrollments (student_id, course_id, grade)
VALUES
(1, 1, 62),
(1, 2, 93),
(2, 1, 83),
(2, 2, 73);
```

3.7.2 数据修改

在 openGauss 数据库中，修改数据是通过 UPDATE 语句来实现的，这与其他关系数据库管理系统的操作方法一样。UPDATE 语句允许更新表中已存在记录的值。以下是 UPDATE 语句的基本用法和一个示例，演示如何在 openGauss 数据库中修改数据。

1. 基本语法

```
UPDATE table_name
SET column1 = value1, column2 = value2, …
WHERE condition;
```

上述语法中，table_name 表示要更新记录的目标表名。SET 子句指定了要更新的列和它们应该被赋予的新值。可以同时更新一个或多个列，列之间用逗号分隔。WHERE 子句指定了哪些记录应该被更新。如果省略 WHERE 子句，表中的所有记录都将被更新，这可能导致不希望的结果，因此使用 WHERE 子句是一个好习惯。

2. 更新单个记录

假设要想更新一位名为“John Doe”的学生的专业（major）为“Data Science”，可以使用如下语句。

```
UPDATE students
SET major = 'Data Science'
WHERE name = 'John Doe';
```

3. 更新多个记录

如果想将所有 20 岁学生的年龄增加 1 岁，可以使用如下语句。

```
UPDATE students
SET age = age + 1
WHERE age = 20;
```

上述命令，会查找 students 表中所有年龄为 20 岁的学生，并将他们的年龄增加 1。

3.7.3 数据删除

在 openGauss 数据库中，删除数据是通过 DELETE 语句来实现的。DELETE 语句允许从表中删除满足特定条件的行。如果没有指定条件，则可以删除表中的所有行，但这种操作需要谨慎进行，因为它会删除表中的所有数据。以下是 DELETE 语句的基本语法和示例，展示如何在 openGauss 数据库中删除数据。

1. 基本语法

```
DELETE FROM table_name
WHERE condition;
```

上述语法中，table_name 是要删除记录的目标表名，WHERE 子句指定了哪些记录应

该被删除。如果省略 WHERE 子句,表中的所有记录都将被删除。

2. 删除特定记录

如果要删除一名特定学生的记录,例如,删除名为“John Doe”的学生,可以使用如下语句。

```
DELETE FROM students
WHERE name = 'John Doe';
```

3. 条件删除多条记录

如果要基于某个条件删除多条记录,例如,删除所有 Data Science 专业的学生,可以使用如下语句。

```
DELETE FROM students
WHERE major = 'Data Science';
```

4. 删除所有记录

要删除表中的所有记录,可以省略 WHERE 子句,使用如下命令。

```
DELETE FROM students;
```

除此之外,还可以使用 TRUNCATE 语句来删除数据,相比 DELETE 语句来说,TRUNCATE 执行效率更高,因为它不会逐行删除数据,而是直接移除所有数据并重置表的状态,但这也意味着它不能与 WHERE 子句一起使用,具体语句如下。

```
TRUNCATE TABLE students ;
```

需要注意的是,如果存在主外键关联时,是不能使用 TRUNCATE 语句的。同时,如果数据表很大,那么删除操作可能会很慢,并且会锁定表直到操作完成,这可能影响到应用程序的性能。

3.8 数据查询

3.8.1 单表查询

单表查询是数据库操作中最基本且最频繁的操作之一,它允许从数据库的单个表中检索数据。在 openGauss 和其他数据库中,这种查询通常通过 SELECT 语句实现。SELECT 语句可以灵活地指定要检索的列、数据筛选条件、排序方法等。以下是单表查询的基本语法和一些示例。

1. 基本语法

```
SELECT column1, column2, …
FROM table_name
```

上述语法中,SELECT 后面表示想从表中检索的列名,使用星号(*)可以选择所有列。FROM 子句指定了查询将要访问的表名。

2. 查询特定列

获取所有学生的姓名和年龄,可以使用如下命令。

```
SELECT name, age
FROM students;
```

3. 查询所有列

获取所有学生的所有信息，可以使用如下命令。

```
SELECT *
FROM students;
```

单表查询是学习 SQL 的基础，通过掌握这些基本操作，可以有效地从数据库中检索出所需的信息。

3.8.2 条件查询

条件查询用于从数据库中检索满足特定条件的记录。这种查询通过在 SELECT 语句中使用 WHERE 子句实现，通过筛选条件来限制查询结果集中的行。条件查询在数据库操作中非常重要，因为它们使得数据检索变得更加灵活和强大。以下是条件查询的基本语法和一些示例。

1. 基本语法

```
SELECT column1, column2, …
FROM table_name
WHERE condition;
```

上述语法中，SELECT 指定了要从表中检索的列。FROM 子句指定了查询将要访问的表名。WHERE 子句用于指定筛选条件，只有满足这些条件的行才会被包含在结果集中。

在 WHERE 子句中，可以使用多种条件表达式来筛选数据，包括但不限于以下几种。

- 比较操作符：＝、＞、＜、＞＝、＜＝、＜＞(不等于)。
- 范围条件：BETWEEN … AND …用于匹配一个范围内的值。
- 列表条件：IN (list_of_values)用于匹配指定列表中的任意值。
- 模糊匹配：LIKE 用于基于模式匹配筛选字符串值。
- NULL 值检查：IS NULL 或 IS NOT NULL 用于筛选 NULL 值或非 NULL 值的列。

2. 基本的条件查询

假设想要查询 students 表中所有专业为 Computer Science 的学生信息，可以使用如下语句。

```
SELECT * FROM students
WHERE major = 'Computer Science';
```

上述语句会返回 students 表中所有 major 列值为“Computer Science”的行。

3. 使用比较运算符

假设想要查询年龄大于 20 岁的学生，可以使用“＞”运算符，具体语句如下。

```
SELECT * FROM students
WHERE age > 20;
```

4. 组合多个条件

可以使用 AND、OR 逻辑运算符来组合多个条件。例如，查询专业为 Computer Science 且年龄大于 20 岁的学生，可以使用如下语句。

```
SELECT * FROM students
WHERE major = 'Computer Science' AND age > 20;
```

5. 使用 IN 操作符

如果想查询属于多个指定专业的学生，可以使用 IN 操作符，具体语句如下。

```
SELECT * FROM students
WHERE major IN ('Computer Science', 'Chemistry');
```

上述语句将返回专业为 Computer Science 或 Chemistry 的所有学生。

6. 使用 LIKE 操作符进行模式匹配

假设想找出所有名字以“J”开头的学生，可以使用 LIKE 操作符，具体语句如下。

```
SELECT * FROM students
WHERE name LIKE 'J%';
```

上述语句中，%是一个通配符，代表任意字符出现任意次数。

7. 使用 BETWEEN 操作符查询范围

如果想要查询年龄为 18～22 岁的学生，可以使用 BETWEEN 操作符，具体语句如下。

```
SELECT * FROM students
WHERE age BETWEEN 18 AND 22;
```

上述语句将返回年龄为 18～22 岁(包括 18 岁和 22 岁)的学生。

3.8.3 多表查询

多表查询是数据库操作中一个重要的部分，它允许从两个或多个表中基于某种关联条件来检索数据。在 SQL 中，这通常是通过 JOIN 语句来实现的。JOIN 语句能够合并来自不同表的行，提供了一种强大的方式来查询跨多个表的关系数据。

基本的 JOIN 类型如下。

- INNER JOIN(内连接)：只返回两个表中匹配的行。
- LEFT JOIN(左连接)：返回左表中的所有行，即使右表中没有匹配的行。
- RIGHT JOIN(右连接)：返回右表中的所有行，即使左表中没有匹配的行。
- FULL JOIN(全连接)：返回左表和右表中的所有行，无论另一边是否有匹配。

1. 内连接

内连接返回两个表中匹配的记录。如果某条记录在连接的另一表中没有对应的匹配，那么这条记录就不会出现在查询结果中。

示例：查询选修了课程的学生及其课程信息。

```
SELECT s.name AS StudentName, c.course_name AS CourseName
FROM enrollments e
INNER JOIN students s ON e.student_id = s.student_id
INNER JOIN courses c ON e.course_id = c.course_id;
```

查询结果：

```
 studentname  |             coursename
--------------+------------------------------------
 John Doe     | Introduction to Computer Science
 John Doe     | Advanced Mathematics
 Jane Smith   | Introduction to Computer Science
 Jane Smith   | Advanced Mathematics
(4 rows)
```

2. 左连接

左连接返回左表(FROM 子句中指定的表)的所有记录,即使在右表中没有匹配的记录。如果右表中没有匹配,则右表的列将返回 NULL。

示例:查询所有学生及其可能选修的课程信息。

```
SELECT s.name AS StudentName, c.course_name AS CourseName
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id
LEFT JOIN courses c ON e.course_id = c.course_id;
```

查询结果:

```
 studentname  |             coursename
--------------+------------------------------------
 John Doe     | Introduction to Computer Science
 John Doe     | Advanced Mathematics
 Jane Smith   | Introduction to Computer Science
 Jane Smith   | Advanced Mathematics
 Alex Johnson |
 Maria Garcia |
(6 rows)
```

上述查询中,确保了即使学生没有选修任何课程,也会被列出。

3. 右连接

右连接与左连接相反,它返回右表的所有记录,即使左表中没有匹配的记录。

示例:查询所有课程及其可能被哪些学生选修。

```
SELECT s.name AS StudentName, c.course_name AS CourseName
FROM students s
RIGHT JOIN enrollments e ON s.student_id = e.student_id
RIGHT JOIN courses c ON e.course_id = c.course_id;
```

查询结果:

```
 studentname  |             coursename
--------------+------------------------------------
 John Doe     | Introduction to Computer Science
 John Doe     | Advanced Mathematics
 Jane Smith   | Introduction to Computer Science
 Jane Smith   | Advanced Mathematics
              | General Physics
              | Organic Chemistry
(6 rows)
```

注意,大多数 SQL 数据库(包括 openGauss)优先支持 LEFT JOIN。RIGHT JOIN 可以通过改变表的顺序并使用 LEFT JOIN 来实现相同的效果。

4. 全连接

全连接返回左表和右表中所有的记录。当某条记录在其中一个表中没有匹配时,查询结果会用 NULL 来填充那个表的列。

示例:列出所有学生和所有课程,包括没有匹配的记录。

```
SELECT s.name AS StudentName, c.course_name AS CourseName
FROM students s
```

```
3.    FULL JOIN enrollments e ON s.student_id = e.student_id
4.    FULL JOIN courses c ON e.course_id = c.course_id;
```

查询结果：

```
1.    studentname  |            coursename
2.    -----------+-----------------------------------
3.    John Doe     | Introduction to Computer Science
4.    John Doe     | Advanced Mathematics
5.    Jane Smith   | Introduction to Computer Science
6.    Jane Smith   | Advanced Mathematics
7.    Alex Johnson |
8.    Maria Garcia |
9.                 | General Physics
10.                | Organic Chemistry
11.   (8 rows)
```

请注意，不是所有的数据库系统都支持 FULL JOIN。如果数据库不支持，可能需要通过 UNION 将 LEFT JOIN 和 RIGHT JOIN 的结果组合起来以模拟 FULL JOIN 的效果。

通过这 4 种类型的连接查询，可以灵活地从关联的表中检索出需要的信息。选择哪一种连接类型取决于具体需求，例如，想要的结果集是否应该包含没有匹配记录的行。

3.8.4 高级查询

高级查询在 SQL 中指的是超出基本 SELECT、INSERT、UPDATE 和 DELETE 操作的查询技巧和功能，用于解决更复杂的数据检索和处理需求。这些查询可能包括使用子查询、聚合函数、公用表表达式(Common Table Expression，CTE)、条件表达式等高级特性。

以下是基于之前定义的 students、courses 和 enrollments 表的一些高级查询示例。

1. 使用聚合函数统计信息

示例：统计每门课程的选修学生人数。

```
1.    SELECT c.course_name, COUNT(e.student_id) AS student_count
2.    FROM courses c
3.    LEFT JOIN enrollments e ON c.course_id = e.course_id
4.    GROUP BY c.course_name;
```

查询结果：

```
1.    course_name                  | student_count
2.    ----------------------------+---------------
3.    General Physics              |             0
4.    Advanced Mathematics         |             2
5.    Organic Chemistry            |             0
6.    Introduction to Computer Science |          2
```

上述查询中，显示了每门课程及其对应的选修学生人数。

2. 子查询

示例：查询选修课程数量最多的学生。

```
1.    SELECT s.name, s.email, COUNT(e.course_id) AS course_count
2.    FROM students s
3.    JOIN enrollments e ON s.student_id = e.student_id
4.    GROUP BY s.student_id
```

```
    HAVING COUNT(e.course_id) = (
     SELECT MAX(course_count) FROM (
      SELECT student_id, COUNT(course_id) AS course_count
      FROM enrollments
      GROUP BY student_id
     ) AS subquery
    );
```

查询结果：

```
    name       |           email           | course_count
    ---------+-----------------------+--------------
    John Doe   | john.doe@gmail.com        |            2
    Jane Smith | jane.smith@example.com    |            2
    (2 rows)
```

上述查询中，先计算每位学生选修的课程数量，然后通过子查询找出选修课程数量最多的学生。

3. 公用表表达式

示例：使用 CTE 查询每个专业的平均成绩。

```
    WITH average_grades AS (
      SELECT s.major, AVG(e.grade) AS avg_grade
      FROM students s
      JOIN enrollments e ON s.student_id = e.student_id
      GROUP BY s.major
    )
    SELECT major, avg_grade
    FROM average_grades
    WHERE avg_grade IS NOT NULL
    ORDER BY avg_grade DESC;
```

查询结果：

```
    major               |      avg_grade
    ----------------+----------------------
    Mathematics         | 78.0000000000000000
    Computer Science    | 77.5000000000000000
    (2 rows)
```

上述查询中，通过 WITH average_grades AS 定义了一个名为 average_grades 的 CTE，来查询学生的平均成绩，并按专业分类。这个查询的目的是计算每个专业学生的平均成绩。

4. 使用 CASE 语句处理条件逻辑

示例：根据成绩评级学生。

```
    SELECT s.name, e.course_id,
    CASE
      WHEN e.grade >= 90 THEN 'Excellent'
      WHEN e.grade >= 80 THEN 'Good'
      WHEN e.grade >= 60 THEN 'Average'
      ELSE 'Needs Improvement'
    END AS performance
    FROM enrollments e
    JOIN students s ON e.student_id = s.student_id;
```

查询结果：

```
        name     | course_id     | performance
 ----------+------------+--------------
 John Doe      |             1 | Average
 John Doe      |             2 | Excellent
 Jane Smith    |             1 | Good
 Jane Smith    |             2 | Average
 (4 rows)
```

上述查询中，评估了学生在每门课程的表现。

高级查询技术能够对数据进行深入分析，提取出有价值的信息。上述示例展示了如何利用 SQL 的高级特性来执行复杂的查询操作。在实际应用中，可以根据具体需求选择合适的查询方法。

小结

本章首先讲解了什么是 SQL，然后针对数据库的基本操作和数据表的基本操作进行讲解，最后讲解了数据的添加、更新、修改、删除、查询等核心内容，学习完本章就可以掌握数据库的基本操作了。

习题

1. 请简要说明如何创建数据库以及创建数据表。
2. 请简要说明在数据库中如何进行条件查询。