LightHouseIT

# 5.2 SQL 基础

- 数据定义
- 数据操作
- 表的连接

# 目录

# SQL 命令

- 以下列出最常见的 SQL 语言命令:
  - **数据库**的操作: CREATE DATABASE、DROP DATABASE 等;
  - **表**的操作（数据定义）: CREATE TABLE、DROP TABLE、ALTER TABLE、TRUNCATE TABLE 等;
  - **增删改查**操作（数据操作）: INSERT INTO、DELETE、UPDATE、SELECT 等;
  - 除此之外，还有一些特殊运算: 结果的交集、并集、差集; 两个数据表的连接等。
- 对于本课程没有介绍的命令，可以查看 PostgreSQL 的官方教程:
  https://www.postgresql.org/docs/14/index.html

# 基本语法

- **标识符命名**和 Java 类似：可以使用**大小写字母**、**数字**、**下划线**或**美元符号**。SQL 标准**不推荐**使用美元符号。

- SQL 中，关键字和标识符都不区分大小写：select 与 SELECT 相同，student 与 Student 相同。
  - 我们建议将所有关键字都**大写**，所有标识符都**小写**以示区分。
  - 标识符建议书写成小写**蛇形**：blog_user。但不同的 DBMS 可能有不同标准。
  - 所有数据库名称建议统一书写为单数（user），或统一书写为复数（users）。

- 每句 SQL 语句[クエリ]（Query）必须以分号 “;” 结尾。

# 注释

- 使用 “**--**” 符号添加单行注释:

```
-- 这是单行注释
```

- 使用 “**/***” 和 “***/**” 添加多行注释:

```
/*
 * 这是多行注释
 */
```

# 创建数据库

- **CREATE DATABASE** 语句用于创建新数据库。
- 句法：

```
CREATE DATABASE name_of_database;
```

- 以下语句创建一个名为 “hello” 的数据库：

```
CREATE DATABASE hello;
```

```
postgres=# CREATE DATABASE hello;
CREATE DATABASE
postgres=#
```

# 删除数据库

- **DROP DATABASE** 语句用于删除已有数据库。
- 句法:

```
DROP DATABASE name_of_database;
```

- 以下语句删除 “hello” 数据库:

```
DROP DATABASE hello;
```

**Note**

删除操作无法撤销，请三思而后行!

# 创建表

- **CREATE TABLE** 语句用于在数据库中创建新表。
- 句法:

```
1 CREATE TABLE table_name (
2   column1 type1,
3   column2 type2,
4   column3 type3
5 );
```

- 其中:
  - table_name 指定**表的名称**;
  - column1、column2、column3 指定表中的**列的名称**;
  - type1、 type2、type3 指定每一列的**数据类型**。

# 数据类型

- PostgreSQL 中提供了大量可用的**数据类型**，这里只介绍最常见的一些。全部可用类型可以参考：
  https://www.postgresql.org/docs/14/datatype.html

# 数字类型

| 名称 | 描述 |
| --- | --- |
| SMALLINT | 小范围整数 |
| INTEGER / INT | 整数 |
| BIGINT | 大范围整数 |
| DECIMAL / NUMERIC | 高精度小数 |
| REAL / DOUBLE PRECISION | 低精度小数 |
| SMALLSERIAL | *自增*的小范围整数 |
| SERIAL | 自增整数 |
| BIGSERIAL | 自增的大范围整数 |

# 字符串类型

| 名称 | 描述 |
| --- | --- |
| VARCHAR(*n*) | 有限制的变长字符串 |
| CHAR(*n*) | 有限制的定长字符串 |
| TEXT | 无限制的字符串 |

**Note**

字符串值要用**单引号** “'” 括起。

# 其它类型

| 名称 | 描述 |
| --- | --- |
| BOOLEAN | 布尔类型（true、false、unknown） |
| ENUM('a', 'b', 'c') | 枚举类型 |
| MONEY | 货币类型 |
| TIMESTAMP | 日期事件类型 |
| DATE | 日期类型 |
| TIME | 时间类型 |

# 表的例子

- 下面的示例创建一个名为 “student” 的表，该表包含 3 列：id、name 和 score：

```
1 CREATE TABLE student (
2   id    SERIAL,
3   name  VARCHAR(255),
4   score SMALLINT
5 );
```

- 其中：
  - id 的类型为整数，并且会自增。
  - name 的类型为字符串，限制长度为 255。
  - score 的类型为小整数。

# 约束

- **约束**[制約]用于指定表中数据的某些限制要求。在创建表时，我们可以在每一个**数据类型之后**书写某些约束。
- 句法：

```
1 CREATE TABLE table_name (
2   column1 type1 constraints1,
3   column2 type2 constraints2,
4   column3 type3 constraints3
5 );
```

- 其中， constraints1、constraints2、constraints3 指定每一列的约束。

# 约束

- 约束限制了可以插入表的数据的特点，这样可以确保表格中数据的准确性和可靠性。如果想要插入不满足约束条件的数据，则该插入命令将产生错误并终止。

- SQL 中常见约束如下：
  - NOT NULL：确保列不能具有 NULL 值（不能为空）。
  - UNIQUE：确保列中的所有值都不同。
  - **PRIMARY KEY**：主键，等于 NOT NULL + UNIQUE。SQL 可以通过主键唯一确定一行数据。
  - **FOREIGN KEY**：外键。可以连接到其它表的主键上。
  - DEFAULT：设置默认值。
  - INDEX：设置用于检索数据的列。使用该列进行检索速度将更快。

- 你可以为一列添加多个约束，使用**空格**分隔。

# 约束的例子

- 和刚刚的例子相似的 student 表格：

```sql
CREATE TABLE student (
  id       SERIAL          PRIMARY KEY,
  name     VARCHAR(255)    NOT NULL,
  score    SMALLINT        DEFAULT 0
);
```

- 其中：
  - id 的类型为主键，唯一确定且不为空。
  - name 的内容不能为空。
  - score 的默认值为 0。

# 修改表

- **ALTER TABLE** 语句用于添加、删除或修改表中的列，或改变它们的约束。

- 语法：
  - 添加列：

```
ALTER TABLE table_name ADD column_name type;
```

  - 删除列：

```
ALTER TABLE table_name DROP COLUMN column_name;
```

接次页

## 接前页

- 语法：

  - 修改某列数据类型：

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE type;
```

  - 为某列添加 NOT NULL 约束：

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

# 删除表

- **DROP TABLE** 语句用于删除数据库中的现有表。
- 句法：

```
DROP TABLE table_name;
```

- 以下语句删除 “student” 表：

```
DROP TABLE student;
```

- 以下语句可以仅在表存在时删除它：

```
DROP TABLE IF EXISTS student;
```


*Question and answer*

# 目录

# 插入数据

- **INSERT INTO** 语句用于在表中插入新记录。
- 句法:

```
INSERT INTO table_name VALUES (data1, data2, ...);
```

- 使用逗号 “,” 隔开每个记录，可以一次性插入**多行记录**:

```
1 INSERT INTO student
2 VALUES (0, 'Alice'   , 90),
3        (1, 'Bob'     , 40),
4        (2, 'Charlie', 70);
```

**Note**

数据的书写顺序必须与定义时的顺序相同。

# 插入指定列的数据

- 也可以指定要插入**哪些列**的数据。没有被指定的列将自动被设为 NULL（或 DEFAULT 设定的默认值）：

```
1 INSERT INTO table_name (column1, column2)
2 VALUES (data1_column1, data1_column2),
3        (data2_column1, data2_column2);
```

**Tips**

此时，column1、column2 等的顺序可以和定义时不同。

# 插入的例子

- 以下命令将一行记录插入了表格，其中 name 设为 “Dave”:

```
INSERT INTO student (id, name) VALUES (3, 'Dave');
```

- 结果:

| id | name | score |
|---|---|---|
| 0 | Alice | 90 |
| 1 | Bob | 40 |
| 2 | Charlie | 70 |
| 3 | Dave | 0 |

- 可以看到:
  - id 字段和 name 字段被设为了指定的值。
  - score 字段被自动设为了默认值。

Try
01011
11010
01011
insert.sql

# 查询数据

- **SELECT** 语句用于从数据库中选择某些记录查询。
- 返回的数据也以表的形式呈现，被称为**结果集**[結果セット]。
- 句法：

```
SELECT column1, column2, ... FROM table_name;
```

- 其中，column1、column2 是我们想要获取的列的名称。

# 查询的例子

- 以下语句查询表中的 name 列以及 score 列：

```
SELECT name, score FROM student;
```

```
hello=# SELECT name, score FROM student;
  name   | score
---------+-------
 Alice   |    90
 Bob     |    40
 Charlie |    70
 Dave    |     0
```

**Tips**

你也可以使用常见的运算符，比如：

```
SELECT name, score * 2 - 10 FROM student;
```

**Try**

select.sql

# 查询所有列

- 使用 “*” 可以选择所有列:

```
SELECT * FROM student;
```

```
hello=# SELECT * FROM student;
 id |   name    | score
----+-----------+-------
  0 | Alice     |    90
  1 | Bob       |    40
  2 | Charlie   |    70
  3 | Dave      |     0
```

# WHERE 子句

- 有时，我们仅需查询表中**满足特定条件**的行的数据。此时就要使用 **WHERE** 子句。
- 句法：

```
1 SELECT column1, column2, ... FROM table_name
2 WHERE constraints;
```

- 其中，在 constraints 处我们可以指定查询时，**一行记录中的某些字段**需要满足的条件。

# 比较运算符

- 最基本的条件就是对数据的**比较运算**:

| 运算符 | 描述 |
| --- | --- |
| = | 等于 |
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |
| != / <> | 不等于 |

# WHERE 的例子

- 以下命令将选出表格中 score 大于 60 的记录:

```
1 SELECT * FROM student
2 WHERE score > 60;
```

```
hello=# SELECT * FROM student WHERE score > 60;
 id |   name    | score
----+-----------+-------
  0 | Alice     |    90
  2 | Charlie   |    70
```

# NULL 的比较

- 判断某个字段是否为 NULL，只能使用 **IS NULL**（为 NULL）或 **IS NOT NULL**（不为 NULL）运算符，**不能使用比较运算符**。
- 比如，以下语句选择 name 不为空的记录：

```
1 SELECT * FROM student
2 WHERE name IS NOT NULL;
```

# LIKE 运算符

- **LIKE** 运算符在字段中选取满足特定条件的**字符串**。
- 通常将以下两个通配符与 LIKE 运算符结合使用（还有很多其他通配符）：
  - “**%**” 符号表示 0 个，1 个或多个任意字符。
  - “**_**”（下划线）表示 1 个任意字符。
- 比如，以下语句将选取表中 name 以 “A” 开头的记录。

```
1 SELECT * FROM student
2 WHERE name LIKE 'A%';
```

# IN 运算符

- **IN** 运算符指定多个值，只要字段是其中任意一个就满足条件。
- 以下语句选择分数为 50、60 或 70 的记录：

```
1 SELECT * FROM student
2 WHERE score IN (50, 60, 70);
```

# BETWEEN 运算符

- **BETWEEN**（AND）运算符指定一个范围，在该范围内的字段满足条件。值可以是数字、文本或日期。
- 以下语句选择 score 在 30 到 50 之间的记录。

```
1 SELECT * FROM student
2 WHERE score BETWEEN 30 AND 50;
```

# 逻辑运算符

- 上述运算符还可以和 **AND** （并且）、**OR** （或者）或 **NOT** （不是）运算符结合使用。
- 以下语句选择 “name 以 e 结尾并且分数大于 50” 的记录:

```
1 SELECT * FROM student
2 WHERE name LIKE '%e'
3 AND score > 50;
```


*Question and answer*

# ORDER BY 子句

- **ORDER BY** 子句用于按升序或降序对结果集进行排序，你可以选择按照哪（些）列数据作为排序标准。

- 默认情况下，ORDER BY 关键字对记录进行**升序**排序。要按降序对记录进行排序，使用 **DESC** 关键字。

- 以下语句对选择到的记录按 score 降序排序:

```
1 SELECT * FROM student
2 ORDER BY score DESC;
```

```
hello=# SELECT * FROM student
hello-# ORDER BY score DESC;
 id |   name    | score
----+-----------+-------
  0 | Alice     |    90
  2 | Charlie   |    70
  1 | Bob       |    40
  3 | Dave      |     0
```

# DISTINCT 关键字

- 在表内部，一列可能包含许多重复值。
- 使用 **DISTINCT** 关键字，可以去除查询结果中的重复数据：

```
SELECT DISTINCT score FROM student;
```

# LIMIT 与 OFFSET 关键字

- 使用 **LIMIT** 关键字，可以只查询前几行记录。
- 以下语句只会查询前 3 行记录：

```
SELECT * FROM student LIMIT 3;
```

- 使用 OFFSET 关键字，可以跳过前几行查询记录。
- 以下语句查询从第 5 行开始的 3 行记录：

```
SELECT * FROM student OFFSET 4 LIMIT 3;
```

# 重命名输出列

- 可以使用 **AS** 关键字重命名输出列。
- 在选择的列名后书写 “AS 新名称”，可以使 SQL 输出查询结果时使用新名称显示：

```
1 SELECT name AS student_name, score AS test_score
2 FROM student;
```

```
hello=# SELECT name AS student_name, score AS test_score FROM student;
 student_name | test_score
--------------+------------
 Alice        |         90
 Bob          |         40
 Charlie      |         70
 Dave         |          0
```

# 聚集函数

- 你可以使用**聚集函数**[集計関数]通过结果集中所有数据计算某些结果:
  - **MIN() / MAX()** 函数计算某一列的最小值 / 最大值。
  - **COUNT()** 函数统计有几行记录。
  - **AVG()** 函数计算一列的平均值。
  - **SUM()** 函数计算一列的和。
- 以下语句计算 score 的最大值、最小值和平均值。

```
SELECT MAX(score), MIN(score), AVG(score) FROM student;
```

# GROUP BY

- **GROUP BY** 语句将具有相同值的记录分组为摘要行，例如 “每个国家 / 地区的客户数量”。
- GROUP BY 语句通常与聚集函数（COUNT、MAX、MIN、SUM、AVG 等）一起使用，以计算每组数据的一些特性。
- 以下语句把记录按国家分组，并算出每个国家的平均工资：

```
1 SELECT country, AVG(salary)
2 FROM  employee
3 GROUP BY country;
```

Try
01011
11010
01011
group.sql

# HAVING 子句

- **HAVING** 子句可以让我们筛选分组后的各组数据。
- HAVING 子句必须**放置于 GROUP BY 子句后面**。
- 以下语句把记录按国家分组，并输出平均工资大于 80 的国家：

```sql
SELECT country FROM  employee
GROUP BY country
HAVING AVG(salary) > 80;
```

# 修改数据

- **UPDATE**（SET）语句用于修改表中的现有记录。
- 可以使用 **WHERE** 子句选择修改哪些记录。
- 以下语句修改将 name 为 "Bob" 的 score 修改为新数据：

```
UPDATE student SET score = 100 WHERE name = 'Bob';
```

```
hello=# SELECT * FROM student;
 id |  name   | score
----+---------+-------
  0 | Alice   |    90
  2 | Charlie |    70
  3 | Dave    |     0
  1 | Bob     |   100
```

# 删除数据

- **DELETE** 语句用于删除表中的现有记录。
- 可以使用 **WHERE** 子句选择删除哪些记录:

```
DELETE FROM student WHERE name = 'Bob';
```

- 可以使用 **TRUNCATE** 删除表中的所有行。注意表本身不会被删除:

```
TRUNCATE TABLE student;
```

# 组合查询

- 我们可以用一些运算符组合多个 SELECT 语句获得的查询结果:
  - **UNION** 语句可以求两个结果集的**并集**[和集合];
  - **INTERSECT** 语句可以求两个结果集的**交集**[積集合];
  - **EXCEPT** 语句可以求某个结果集对另一个的**差集**[差集合]。

- 以下语句查询两个数据表中都有的 name:

```
1 SELECT name FROM student
2 INTERSECT
3 SELECT name FROM employee;
```


*Question and answer*


LightHouseIT

# 连接

- **JOIN** 子句使我们通过某些方式**连接**[結合]起两个表。这让我们可以通过别的表的数据来辅助查询。

- SQL 提供 5 种连接类型:
  - **交叉连接**[交差結合]: 返回左右表记录的所有 “组合”;
  - **内连接**[内部結合]: 从左右表记录的 “组合” 中挑选合适的记录;
  - **左外连接**[外部結合]: 使用右表的记录 “补充” 左表;
  - **右外连接**: 使用左表的记录 “补充” 右表;
  - **全外连接**: 左右外连接的并集。

# 例子中表的准备

- 我们将以这两张表为例讲解不同的连接类型:

pokemon

| id | name | type_id |
|---|---|---|
| 37 | "Vulpix" | 10 |
| 46 | "Paras" | 7 |
| 133 | "Eevee" | 1 |

type

| id | name | super_id |
|---|---|---|
| 2 | "Fight" | 1 |
| 7 | "Bug" | 12 |
| 10 | "Fire" | 7 |
| 11 | "Water" | 10 |

Try
joins.sql

# 交叉连接

- **交叉连接**计算两张表的**笛卡尔积**，也就是所有可能的 “记录的组合”。
- 要计算交叉连接，使用 **CROSS JOIN** 运算符或其简写 “**,**”。
- 以下语句将计算 pokemon 表和 type 表的笛卡尔积:

```
SELECT * FROM pokemon, type;
```

# 交叉连接的结果

- 可以看到，计算出的表列举出了所有可能的 pokemon 和 type 的组合：

```
hello=# SELECT * FROM pokemon, type;
 id  |  name   | type_id | id | name  | super_id
-----+---------+---------+----+-------+----------
  37 | Vulpix  |      10 |  2 | Fight |        1
  37 | Vulpix  |      10 |  7 | Bug   |       12
  37 | Vulpix  |      10 | 10 | Fire  |        7
  37 | Vulpix  |      10 | 11 | Water |       10
  46 | Paras   |       7 |  2 | Fight |        1
  46 | Paras   |       7 |  7 | Bug   |       12
  46 | Paras   |       7 | 10 | Fire  |        7
  46 | Paras   |       7 | 11 | Water |       10
 133 | Eevee   |       1 |  2 | Fight |        1
 133 | Eevee   |       1 |  7 | Bug   |       12
 133 | Eevee   |       1 | 10 | Fire  |        7
 133 | Eevee   |       1 | 11 | Water |       10
```

# 内连接

- **内连接**同样会产生记录的组合，但是只有满足特定条件的组合才会被选中。
- 要计算内连接，使用 **INNER JOIN** 运算符或其简写 **JOIN**。
- 以下语句选择出所有记录组合中 pokemon 的 type_id 和 type 的 id 相同的记录（注意我们是如何回避同名列问题的）：

```sql
SELECT *
FROM pokemon JOIN type
ON pokemon.type_id = type.id;
```

# 内连接的结果

- 可以看到，只有 pokemon 的 type_id 和 type 的 id 相吻合的记录才会被选中：

```
hello=# SELECT *
hello-# FROM pokemon JOIN type
hello-# ON pokemon.type_id = type.id;
 id |  name   | type_id | id | name | super_id
----+---------+---------+----+------+----------
 37 | Vulpix  |      10 | 10 | Fire |        7
 46 | Paras   |       7 |  7 | Bug  |       12
```

- 以下语句通过 type 表查询出了 pokemon 的 type 的名字：

```
hello=# SELECT pokemon.name AS pokemon, type.name as type
hello-# FROM pokemon JOIN type
hello-# ON pokemon.type_id = type.id;
 pokemon | type
---------+------
 Vulpix  | Fire
 Paras   | Bug
```

# 左外连接

- **左（外）连接**查找一些满足条件的右表中的记录 “扩充” 左表中的记录。
- 如果对于一个左表记录，没有满足条件的右表记录与之对应，“扩充” 的字段会被设置为 NULL。
- 要计算左外连接，使用 **LEFT OUTER JOIN** 运算符或其简写 **LEFT JOIN**。
- 以下语句通过 type 表 “扩充” pokemon 表：

```
1 SELECT *
2 FROM pokemon LEFT JOIN type
3 ON pokemon.type_id = type.id;
```

# 左外连接的结果

- 可以看到，有对应 type 的记录右侧增加了对应字段，而没有对应 type 的变为了 NULL（之所以叫左外连接是因为这种变化保留了所有左表的记录）：

```
hello=# SELECT *
hello-# FROM pokemon LEFT JOIN type
hello-# ON pokemon.type_id = type.id;
 id  |   name   | type_id | id | name | super_id
-----+----------+---------+----+------+----------
  37 | Vulpix   |      10 | 10 | Fire |        7
  46 | Paras    |       7 |  7 | Bug  |       12
 133 | Eevee    |       1 |    |      |
```

# 右外连接

- **右（外）连接**与左外连接类似，只是左右互换。
- 要计算左外连接，使用 **RIGHT OUTER JOIN** 运算符或其简写 **RIGHT JOIN**。
- 以下语句通过 pokemon 表 “扩充” type 表：

```
1 SELECT *
2 FROM pokemon RIGHT JOIN type
3 ON pokemon.type_id = type.id;
```

# 右外连接的结果

- 可以看到，右表中的记录都被保留，有些被“扩充”：

```
hello=# SELECT *
hello-# FROM pokemon RIGHT JOIN type
hello-# ON pokemon.type_id = type.id;
 id |   name   | type_id | id | name   | super_id
----+----------+---------+----+--------+----------
 37 | Vulpix   |      10 | 10 | Fire   |        7
 46 | Paras    |       7 |  7 | Bug    |       12
    |          |         | 11 | Water  |       10
    |          |         |  2 | Fight  |        1
```

# 全外连接

- **全外连接**就是左右外连接结果的并集。换句话说，左右表中所有记录都会被保留，而其中有对应记录的组合会互相补充。

- 要计算左外连接，使用 **FULL OUTER JOIN** 运算符或其简写 **FULL JOIN**。

- 以下语句让 pokemon 表和 type 表相互补充：

```
1 SELECT *
2 FROM pokemon FULL JOIN type
3 ON pokemon.type_id = type.id;
```

# 全外连接的结果

- 可以看到，左右表中的所有记录都能在结果中找到：

```
hello=# SELECT *
hello-# FROM pokemon FULL JOIN type
hello-# ON pokemon.type_id = type.id;
 id  |   name   | type_id | id | name   | super_id
-----+----------+---------+----+--------+----------
  37 | Vulpix   |      10 | 10 | Fire   |        7
  46 | Paras    |       7 |  7 | Bug    |       12
 133 | Eevee    |       1 |    |        |
     |          |         | 11 | Water  |       10
     |          |         |  2 | Fight  |        1
(5 行)
```

# 自连接

- 我们也可以把一个表和它**自己连接**[自己結合]。比如，以下语句使用左连接，查找每个 type 的 super_id 对应的 type（注意我们是如何回避同名表问题的）：

```
1 SELECT *
2 FROM type AS a LEFT JOIN type AS b
3 ON a.super_id = b.id;
```

```
hello=# SELECT *
hello-# FROM type AS a LEFT JOIN type AS b
hello-# ON a.super_id = b.id;
 id | name   | super_id | id | name | super_id
----+--------+----------+----+------+----------
  2 | Fight  |        1 |    |      |
  7 | Bug    |       12 |    |      |
 10 | Fire   |        7 |  7 | Bug  |       12
 11 | Water  |       10 | 10 | Fire |        7
```


*Question and answer*

# 总结

## Sum Up

1. SQL 基本语法。

2. 数据定义语法：
   ① 数据库的创建和操作；
   ② 表的创建、数据类型、约束条件。

3. 数据操作语法：
   ① 选择数据：WHERE、ORDER BY、GROUP BY；
   ② 插入、删除和修改数据的方法。

4. 表连接的语法和适用场景。

LightHouseIT

# THANK YOU!