SQL 讲义

中文版

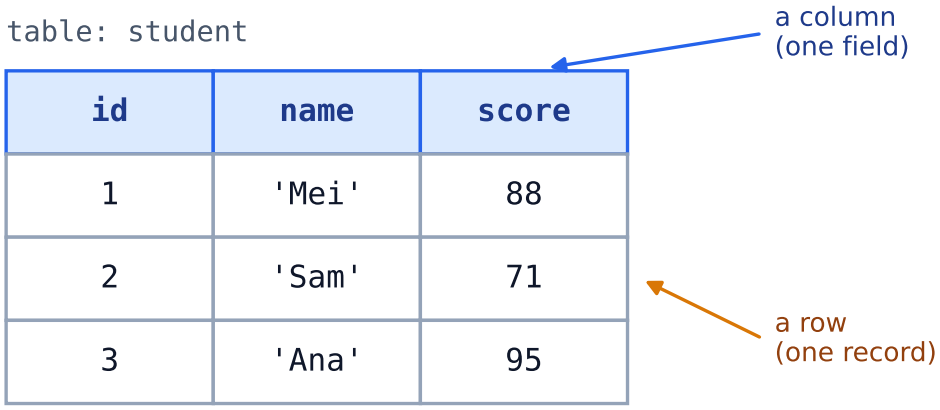
Contents

1 查询基础

1.1 SELECT 与 FROM

数据库 (database) 把数据存放在表 (table) 里。查询 (query) 用 SELECT 读取数据。SELECT * 返回每一列;FROM 指明是哪张表。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT * FROM student;
```



表存储记录: 每一行是一条记录, 每一列是一个字段

- 每一行 (row) 是一条记录 (record)。按习惯,SQL 关键字写成大写。
- 一条语句以分号 ; 结尾。

1.2 选择列

把你想要的列 (column) 列出来, 用逗号分隔。用 AS 在结果里给列重命名 (一个别名 (alias))。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT name, score AS mark FROM student;
```

选中的列也可以是一个计算——配合 AS 给新列命名:

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT name, score + 5 AS bonus FROM student;
```

- SELECT DISTINCT col 会去掉结果中的重复值。

1.3 用 WHERE 筛选

WHERE 只保留符合条件 (condition) 的行。用 =、<>(不等于)、<、>、<=、>= 比较。文本要放在单引号里。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71), (3, 'Ana', 95);
SELECT name, score FROM student WHERE score >= 80;
```

- SQL 按这个顺序运行各部分:FROM → WHERE → SELECT。

常见错误

- 文本值要用单引号:WHERE name = 'Ann'; 列名不加引号。
- SELECT * 返回所有列——只选你需要的那些列。
- WHERE 用来筛选行; 它写在 FROM 之后。

2 筛选与逻辑

2.1 AND、OR、NOT

用 AND、OR、NOT 组合条件。AND 需要各边都为真;OR 需要任意一边为真;NOT 把一个条件取反。用括号来设定优先级 (precedence)。

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
↪  (1,'Mei','11A',88),(2,'Sam','11B',71),(3,'Ana','11A',95);
SELECT name FROM student WHERE form = '11A' AND score >= 90;
```

2.2 LIKE、IN 与 BETWEEN

LIKE 匹配一个文本模式 (pattern):% 代表任意文本,_ 代表一个字符——它们是通配符 (wildcard)。

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
↪  (1,'Mei','11A',88),(2,'Sam','11B',71),(3,'Ana','11A',95);
SELECT name FROM student WHERE name LIKE 'M%';        -- starts with M
```

模式	匹配
'M%'	以 M 开头
'%a'	以 a 结尾
'%an%'	包含"an"
'M_i'	M, 接恰好一个字符, 再接 i

IN 匹配一个值的集合 (set);BETWEEN 匹配一个范围 (range)(两端都包含)。

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
↪  (1,'Mei','11A',88),(2,'Sam','11B',71),(3,'Ana','11A',95);
SELECT name, score FROM student
WHERE form IN ('11A', '11C') AND score BETWEEN 80 AND 100;
```

2.3 NULL 空值

NULL 表示缺失值 (missing value)——这个单元格里什么都没存。NULL 不是 0, 也不是空字符串。用 = 或 <> 的比较永远匹配不到它; 要用 IS NULL 或 IS NOT NULL 来判断。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', NULL), (3, 'Ana',
↪  95);
SELECT name FROM student WHERE score IS NULL;
```

```sql
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', NULL), (3, 'Ana',
↪  95);
SELECT name, score FROM student WHERE score IS NOT NULL;
```

- `WHERE score = NULL` 一行都不会返回——连 Sam 那一行也不会。
- NULL 参与算术仍是 NULL: 对 Sam 来说 `score + 5` 还是 NULL。

常见错误

- 判断空值用 `IS NULL`, 绝不能用 `= NULL`。
- 在 `LIKE` 中,`%` 匹配任意文本,`_` 匹配一个字符:`'A%'` 表示“以 A 开头”。
- `IN (1, 2, 3)` 比一串 `OR` 更简洁;`BETWEEN a AND b` 两端都包含。

3 排序与限制

3.1 ORDER BY 与 LIMIT

ORDER BY 对结果行排序 (sort)。加 DESC 表示降序 (descending)(从高到低); 默认是升序 (ascending)(从低到高)。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1,'Mei',88),(2,'Sam',71),(3,'Ana',95);
SELECT name, score FROM student ORDER BY score DESC;
```

3.1.1 按多列排序

列出多个列。第一列出现平局 (tie) 时, 由下一列来决定先后。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1,'Mei',88),(2,'Sam',88),(3,'Ana',95);
SELECT name, score FROM student ORDER BY score DESC, name ASC;
```

3.1.2 LIMIT(前 N 名)

LIMIT n 只保留前 n 行——配合 ORDER BY 就能得到一个前 N 名 (top-N) 列表。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1,'Mei',88),(2,'Sam',71),(3,'Ana',95);
SELECT name, score FROM student ORDER BY score DESC LIMIT 2;
```

- ORDER BY 也可以按别名或计算排序:ORDER BY avg_score DESC。
- 文本按字母顺序排序:ORDER BY name 从 A 到 Z。

常见错误

- ORDER BY 默认升序; 要降序加 DESC。
- ORDER BY 写在靠后的位置, 在 WHERE 和 GROUP BY 之后。
- LIMIT 限制返回的行数, 但要在排序之后才起作用。

4 聚合与分组

4.1 聚合函数

聚合函数 (aggregate function) 把许多行变成一个汇总 (summary) 值。用 `ROUND(x, 2)` 把平均值整理一下。

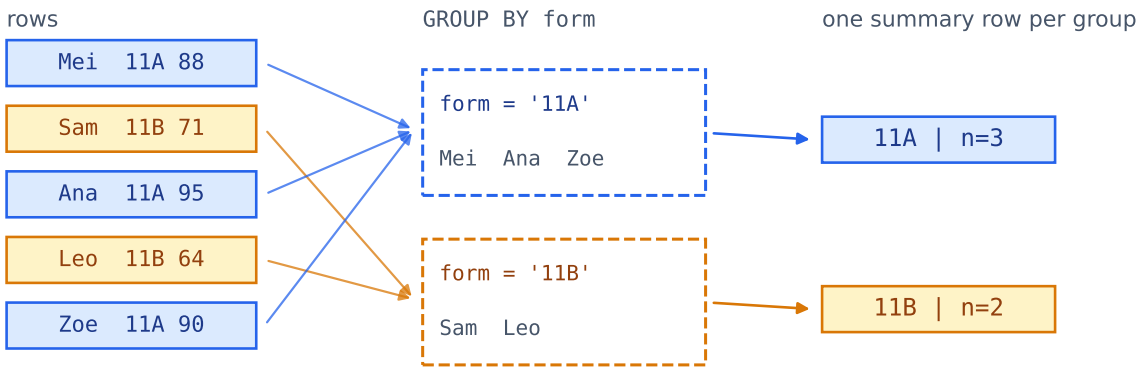
函数	结果
`COUNT(*)`	有多少行
`SUM(col)`	总和
`AVG(col)`	平均值
`MIN(col)` / `MAX(col)`	最小值 / 最大值

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
↪  (1,'Mei','11A',88),(2,'Sam','11B',71),(3,'Ana','11A',95);
SELECT COUNT(*), ROUND(AVG(score), 2), MAX(score) FROM student;
```

4.2 GROUP BY 与 HAVING

`GROUP BY` 为每个组 (group) 生成一个汇总行。`HAVING` 用来筛选这些组——它像 `WHERE`，但在分组之后才运行。

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
↪  (1,'Mei','11A',88),(2,'Sam','11B',71),(3,'Ana','11A',95);
SELECT form, COUNT(*) AS n, ROUND(AVG(score), 2) AS avg_score
FROM student
GROUP BY form
HAVING COUNT(*) > 1;
```



GROUP BY 把行按值收进各自的组；每组产出一行汇总

无论你按什么顺序书写，SQL 总是按同一个固定顺序运行查询的各个子句：

步骤	子句
1	FROM(以及 JOIN)
2	WHERE——筛选行
3	GROUP BY——分组
4	HAVING——筛选组
5	SELECT——计算输出列
6	ORDER BY, 然后 LIMIT

常见错误

- 除非普通列出现在 GROUP BY 中, 否则不能和聚合函数并列选择。
- 用 WHERE 筛选行 (分组前), 用 HAVING 筛选组 (分组后)。
- COUNT(*) 统计行数;COUNT(col) 会跳过该列的 NULL。

5 连接

5.1 键与关系

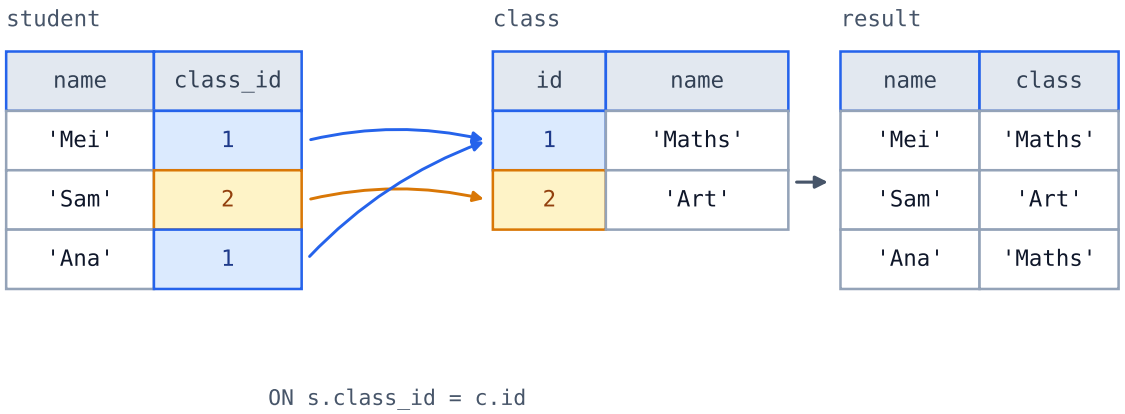
主键 (primary key) 在一张表里唯一地标识每一行。一张表里的外键 (foreign key) 指向另一张表的主键——这就在它们之间建立了关系 (relationship)。

```
CREATE TABLE class (id INTEGER, name TEXT);
CREATE TABLE student (id INTEGER, name TEXT, class_id INTEGER);
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2);
SELECT * FROM student;
```

5.2 INNER JOIN

连接 (join) 把两张表的行组合起来。`INNER JOIN ... ON ...` 只保留键匹配的行。用短别名 (alias)(`s`、`c`) 让查询更易读。

```
CREATE TABLE class (id INTEGER, name TEXT);
CREATE TABLE student (id INTEGER, name TEXT, class_id INTEGER);
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2);
SELECT s.name, c.name AS class
FROM student s INNER JOIN class c ON s.class_id = c.id;
```



INNER JOIN 用外键匹配主键，把匹配上的行组合起来

5.3 连接与分组

先连接，再用 `GROUP BY` 对连接后的行做汇总。

```
CREATE TABLE class (id INTEGER, name TEXT);
CREATE TABLE student (id INTEGER, name TEXT, class_id INTEGER);
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
```

```sql
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2), (3, 'Ana', 1);
SELECT c.name AS class, COUNT(*) AS n
FROM student s INNER JOIN class c ON s.class_id = c.id
GROUP BY c.name;
```

5.4 LEFT JOIN 左连接

INNER JOIN 只保留匹配上的行。LEFT JOIN(左连接) 保留左边 (第一张) 表的**每一行**; 没有匹配时, 右表的列返回 NULL。

```sql
CREATE TABLE class (id INTEGER, name TEXT);
CREATE TABLE student (id INTEGER, name TEXT, class_id INTEGER);
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', NULL);
SELECT s.name, c.name AS class
FROM student s LEFT JOIN class c ON s.class_id = c.id;
```

- Sam 没有班级, 但这一行仍然出现——class 是 NULL。
- 只想找没匹配上的行: 再加 WHERE c.id IS NULL。

常见错误

- 没有 ON 条件的连接会把每一行和每一行配对 (交叉连接)。
- 让外键对上主键:ON orders.customer_id = customers.id。
- INNER JOIN 会丢掉在另一侧没有匹配的行; 想保留它们就用 LEFT JOIN。

6 修改数据

6.1 INSERT

INSERT INTO ... VALUES ... 添加新的行 (row)。先写出列名, 再按相同顺序给出值。(下面每个代码块都以 SELECT 结尾, 这样你能看到结果。)

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student (id, name, score) VALUES (1, 'Mei', 88);
INSERT INTO student VALUES (2, 'Sam', 71);
SELECT * FROM student;
```

- INSERT INTO t VALUES (...), (...); 一条语句插入多行。

6.2 UPDATE

UPDATE ... SET ... WHERE ... 改变已有的行。**一定**要加 WHERE, 否则每一行都会被改。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
UPDATE student SET score = 75 WHERE name = 'Sam';
SELECT * FROM student;
```

6.3 DELETE

DELETE FROM ... WHERE ... 删除行。没有 WHERE 时, 它会清空整张表 (table)。

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
DELETE FROM student WHERE score < 80;
SELECT * FROM student;
```

常见错误

- **没有** WHERE 的 UPDATE 和 DELETE 会改动每一行——一定要加 WHERE。
- 在 INSERT 中, 值要在顺序和类型上与列表一一对应。
- 危险的 DELETE 先用同样的 WHERE 写成 SELECT 试一下。

7 定义表

7.1 CREATE TABLE 与数据类型

CREATE TABLE 定义一张表的表结构 (schema): 列名以及它们的数据类型 (data type)。SQLite 主要的类型是 INTEGER、TEXT 和 REAL(带小数的数)。

```
CREATE TABLE student (id INTEGER, name TEXT, score REAL);
INSERT INTO student VALUES (1, 'Mei', 88.5);
SELECT * FROM student;
```

7.2 键与约束

约束 (constraint) 是对某一列的规则:PRIMARY KEY(唯一的 id)、NOT NULL(必须有值)、UNIQUE, 以及 DEFAULT(默认值)。

```
CREATE TABLE student (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    score INTEGER DEFAULT 0
);
INSERT INTO student (id, name) VALUES (1, 'Mei');
SELECT * FROM student;
```

用 REFERENCES 声明外键——它记录这一列指向另一张表的主键:

```
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    class_id INTEGER REFERENCES class(id)
);
INSERT INTO class VALUES (1, 'Maths');
INSERT INTO student VALUES (1, 'Mei', 1);
SELECT s.name, c.name AS class
FROM student s INNER JOIN class c ON s.class_id = c.id;
```

- 完整写法是单独一行的 FOREIGN KEY (class_id) REFERENCES class(id)。

7.3 ALTER TABLE

ALTER TABLE ... ADD COLUMN ... 改变一张已存在的表的表结构。DEFAULT 会把新列在旧行里填上值。

```
CREATE TABLE student (id INTEGER, name TEXT);
INSERT INTO student VALUES (1, 'Mei');
ALTER TABLE student ADD COLUMN score INTEGER DEFAULT 0;
SELECT * FROM student;
```

- ALTER TABLE student RENAME TO pupil; 给整张表改名。

- `DROP TABLE student;` 彻底删除这张表——结构**和**数据。

常见错误

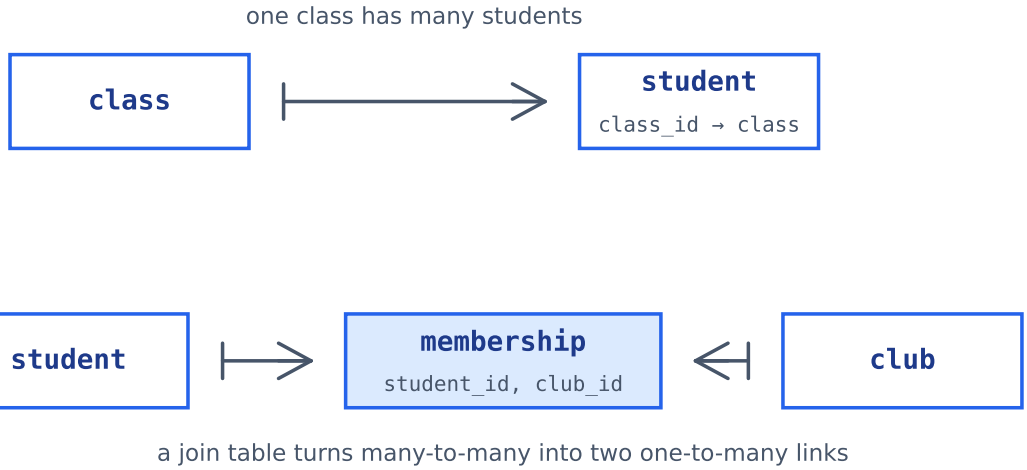
- 每一列都要有数据类型 (如 `INTEGER`、`TEXT`)。
- 主键 (PRIMARY KEY) 必须唯一, 且不能为 NULL。
- 外键的值必须在它所指向的表中存在。

8 数据库设计

8.1 关系与 ER 图

表通过关系 (relationship) 相互连接。一对多 (one-to-many) 最常见: 一个班有许多学生。多对多 (many-to-many) 需要中间有一张连接表 (join table)。实体关系图 (entity-relationship diagram,ER 图) 把每个实体 (entity) 画成一个方框, 把每个关系画成一条线。

关系	例子
一对一	一个人和他的护照
一对多	一个班和它的学生
多对多	学生和社团



”多”的一侧携带外键; 多对多关系需要一张连接表

连接表里每个链接占一行。这里 Mei 参加了两个社团, 而 Chess 社有两名成员:

```
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE club (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE membership (student_id INTEGER, club_id INTEGER);
INSERT INTO student VALUES (1, 'Mei'), (2, 'Sam');
INSERT INTO club VALUES (1, 'Chess'), (2, 'Art');
INSERT INTO membership VALUES (1, 1), (1, 2), (2, 1);
SELECT s.name, c.name AS club
FROM membership m
INNER JOIN student s ON m.student_id = s.id
INNER JOIN club c ON m.club_id = c.id;
```

8.2 范式化

范式化 (normalisation) 把表组织好, 以避免冗余 (redundancy)(同样的数据重复) 以及它带来的更新错误。前三个范式 (normal form):

- **1NF**: 每个单元格只放一个原子 (atomic) 值——单元格里不放列表。
- **2NF**: 没有列只依赖复合主键 (composite key) 的一部分。
- **3NF**: 没有列依赖另一个非键列。

未范式化 (差)	已范式化 (更好)
`student(name, club1, club2)`	`student(name)` + `membership(student, club)`

常见错误

- 把重复的组拆到单独的表里 (规范化), 而不是用许多相似的列。
- 每张表只描述**一种**事物。
- 用外键把表连接起来, 外键指向另一张表的主键。