4.3 声明式编程

::: details INFO
译者：kam1u，Mancuoj

来源：4.3 Declarative Programming
:::

除了数据流，数据值通常存储在称为数据库的大型存储库中。数据库有一个数据存储区，其中包含数据值以及检索和转换这些值的接口。存储在数据库中的每个值都称为记录（record）。具有相似结构的记录会被分组到表中。使用查询语言中的查询语句可以检索和转换记录。到目前为止，最常用的查询语言是结构化查询语言 (Structured Query Language)，简称 SQL（发音为“sequel”）。

> 译者注：“SQL”的发音为“ess-cue-ell”或“sequel”。后者的发音，即“sequel”，现在越来越流行。

SQL 是一种声明式编程语言的例子。SQL 语句不直接描述计算过程，而是描述一些计算的预期结果。数据库系统的查询解释器负责设计和执行计算过程以产生这样结果。

这种交互方式与 Python 或 Scheme 的过程式编程范式有很大的区别。在 Python 中，计算过程由程序员直接描述。声明式语言抽象了过程细节，而是将重点放在结果的形式上。

4.3.1 表

SQL 语言是标准化的，但大多数数据库系统实现了一些具有专有功能的自定义变体。在本文中，我们将使用一个 Sqlite 实现的小的 SQL 子集。您可以通过 下载 Sqlite 或使用 在线 SQL 解释器 来跟随学习。

一张表（table），也称为关系（relation），具有固定数量的命名列和类型列。表的每一行代表一个数据记录，并且每个列都有一个值。例如，一个城市表可能有两个包含数字值的列：经度和纬度，以及一个包含字符串的城市名列。每一行都将通过其经度值和纬度值来表示一个城市的位置。

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

可以使用 SQL 语言中的 `select` 语句创建一个单行表，其中行值用逗号分隔，列名跟在关键字 **"as"** 后面。所有的 SQL 语句都以分号结尾。

```
sqlite> select 38 as latitude, 122 as longitude, "Berkeley" as name;
38|122|Berkeley
```

第二行是输出结果，其中每一行都有若干列，用垂直线分隔。

通过 `union` 语句可以将两个表的行合并在一起，创建一个多行表。创建的表会使用左表的列名。且一行内的空格不会影响结果。

```
sqlite> select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...> select 42,             71,              "Cambridge"        union
   ...> select 45,             93,              "Minneapolis";
38|122|Berkeley
42|71|Cambridge
45|93|Minneapolis
```

使用 `create table` 语句可以给表命名。虽然这个语句也可以用来创建空表，但我们将重点关注给已由 `select` 语句定义的现有表命名的形式。

```
sqlite> create table cities as
   ...>    select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...>    select 42,             71,              "Cambridge"        union
   ...>    select 45,             93,              "Minneapolis";
```

一旦表被命名，就可以在 `select` 语句的 `from` 子句中使用该名称。使用特殊的 `select *` 形式可以显示表的所有列。

```
sqlite> select * from cities;
38|122|Berkeley
42|71|Cambridge
45|93|Minneapolis
```

4.3.2 Select 语句

`select` 语句可以通过列出单行的值或更常见的通过在 `from` 子句中使用现有表进行投影来定义一个新表：

```
select [column name] from [existing table name]
```

> 译者注：如果我们只希望返回某些列的数据，而不是所有列的数据，我们可以用 `SELECT 列1, 列2, 列3 FROM ...`，让结果集仅包含指定列。这种操作称为 **投影查询**。

生成的表的列由逗号分隔的表达式列表描述，每个表达式在现有输入表的每一行上都会被计算。

例如，我们可以创建一个双列表，通过描述每个城市相对于伯克利向北或向南的距离来描述它。每一度纬度向北测量 60 海里。

```
sqlite> select name, 60*abs(latitude-38) from cities;
Berkeley|0
Cambridge|240
Minneapolis|420
```

列描述是一个语言中的表达式，它与 Python 具有许多相似之处：中缀运算符，如 + 和 %，内置函数，如 `abs` 和 `round`，以及用于描述计算顺序的括号。这些表达式中的名称，如上面的 `latitude`，在进行投影时求值为该行的列值。

可选地，每个表达式都可以跟随关键字 `as` 和列名。当整个表被命名时，通常有助于为每个列赋予一个名称，以便将来可以在选择语句中引用它。由简单名称描述的列会自动命名。

```
sqlite> create table distances as
   ...>   select name, 60*abs(latitude-38) as distance from cities;
sqlite> select distance/5, name from distances;
0|Berkeley
48|Cambridge
84|Minneapolis
```

Where 子句：`select` 语句还可以包括一个带有过滤表达式的 `where` 子句。该表达式会对被投射的行进行过滤。只有当过滤表达式的计算结果为真时，才会使用该行来产生结果表中的一行。

```
sqlite> create table cold as
   ...>   select name from cities where latitude > 43;
sqlite> select name, "is cold!" from cold;
Minneapolis|is cold!
```

Order 子句：`select` 语句还可以对结果表达排序。`Order` 子句包含一个对于每个未过滤的行进行计算的排序表达式。该表达式的计算结果用作结果表的排序标准。

```
sqlite> select distance, name from distances order by -distance;
84|Minneapolis
48|Cambridge
0|Berkeley
```

这些特性的组合允许 `select` 语句表达将输入表投影到相关输出表的广泛范围。

4.3.3 连接

数据库通常包含多个表，查询可能需要包含在不同表中的信息来计算所需结果。例如，我们可能有第二个表描述不同城市的每日平均最高温度。

```
sqlite> create table temps as
   ...>   select "Berkeley" as city, 68 as temp union
   ...>   select "Chicago"  , 59         union
   ...>   select "Minneapolis"     , 55;
```

通过将多个表连接成一个表来组合数据，这是数据库系统中的基本操作。连接（join）有许多密切相关的方法，但在本文中我们只关注其中一种。当表进行连接时，结果表中会包含输入表中每个行的组合的新行。如果两个表进行连接，左表有 m 行，右表有 n 行，则连接表将有 $m \cdot n$。连接在 SQL 中通过在 `select` 语句的 `from` 子句中用逗号分隔表名来表示。

```
sqlite> select * from cities, temps;
38|122|Berkeley|Berkeley|68
38|122|Berkeley|Chicago|59
38|122|Berkeley|Minneapolis|55
42|71|Cambridge|Berkeley|68
42|71|Cambridge|Chicago|59
42|71|Cambridge|Minneapolis|55
45|93|Minneapolis|Berkeley|68
45|93|Minneapolis|Chicago|59
45|93|Minneapolis|Minneapolis|55
```

连接通常伴随着一个 `where` 子句，用于表达两个表之间的关系。例如，如果我们想将数据收集到一个表中，使纬度和温度相关联，我们会从连接中选择那些在每个表中都提到相同城市的行。在 `cities` 表中，城市名称存储在一个名为 `name` 的列中。而在 `temps` 表中，城市名称存储在一个名为 `city` 的列中。`where` 子句可以选择 `join` 表中这些值相等的行。在 SQL 中，使用单个 `=` 符号来进行数值相等的测试。

```
sqlite> select name, latitude, temp from cities, temps where name = city;
Berkeley|38|68
Minneapolis|45|55
```

表可能具有重叠的列名，因此我们需要一种通过表名来消除歧义的方法。一个表也可以与自身连接，因此我们需要一种区分表的方法。为此，SQL 允许我们在 from 子句中使用关键字 `as` 给表命名别名，并使用点表达式引用特定表中的列。以下 `select` 语句计算不同城市对之间的温度差异。`where` 子句中的字母顺序约束确保每个对只出现一次在结果中。

```
sqlite> select a.city, b.city, a.temp - b.temp
   ...>        from temps as a, temps as b where a.city < b.city;
Berkeley|Chicago|10
Berkeley|Minneapolis|15
Chicago|Minneapolis|5
```

SQL 中组合表的两种方法，`join` 和 `union`，为语言提供了很大的表达能力。

4.3.4 SQL 解释器

为了创建一个解释器来处理我们到目前为止介绍的 SQL 子集，我们需要创建一个表示表的结构、一个能够将文本语句解析成可执行代码的解析器以及一个用于执行解析后语句的求值器。SQL 解释器示例包含了所有这些组件，提供了一个简单但功能齐全的声明性语言解释器的演示。

在这个实现中，每个表都有自己的类，并且表中的每一行都由其所属表的一个实例来表示。每行具有与表中每列对应的一个属性，而表是一个行的序列。

表的类使用 Python 标准库中 `collections` 包中的 namedtuple 函数创建，该函数返回一个新的元组子类，为元组中的每个元素命名。

考虑下面重复了的上一节中的 `cities` 表：

```
sqlite> create table cities as
   ...>    select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...>    select 42,             71,               "Cambridge"        union
   ...>    select 45,             93,               "Minneapolis";
```

下面的 Python 语句构建了这个表的一个表示：

```
>>> from collections import namedtuple
>>> CitiesRow = namedtuple("Row", ["latitude", "longitude", "name"])
>>> cities = [CitiesRow(38, 122, "Berkeley"),
              CitiesRow(42,  71, "Cambridge"),
              CitiesRow(43,  93, "Minneapolis")]
```

使用序列操作可以解释 `select` 语句的结果。考虑上一节中重复的距离表：

```
sqlite> create table distances as
   ...>   select name, 60*abs(latitude-38) as distance from cities;
sqlite> select distance/5, name from distances;
0|Berkeley
48|Cambridge
84|Minneapolis
```

一个 `select` 语句的结果可以使用序列操作进行解释。考虑下面重复了的上一节中的 `distances` 表：

```
>>> DistancesRow = namedtuple("Row", ["name", "distance"])
>>> def select(cities_row):
        latitude, longitude, name = cities_row
        return DistancesRow(name, 60*abs(latitude-38))
>>> distances = list(map(select, cities))
>>> for row in distances:
        print(row)
Row(name='Berkeley', distance=0)
Row(name='Cambridge', distance=240)
Row(name='Minneapolis', distance=300)
```

SQL 解释器的设计将这种方法推广到了更一般的场景，一个 `select` 语句被表示为 `Select` 类的一个实例，该实例由 `select` 语句中的子句构建而成。

```
>>> class Select:
        """select [columns] from [tables] where [condition] order by [order]."""
        def __init__(self, columns, tables, condition, order):
            self.columns = columns
            self.tables = tables
            self.condition = condition
            self.order = order
            self.make_row = create_make_row(self.columns)
        def execute(self, env):
            """Join, filter, sort, and map rows from tables to columns."""
            from_rows = join(self.tables, env)
            filtered_rows = filter(self.filter, from_rows)
            ordered_rows = self.sort(filtered_rows)
            return map(self.make_row, ordered_rows)
        def filter(self, row):
            if self.condition:
                return eval(self.condition, row)
            else:
                return True
        def sort(self, rows):
            if self.order:
                return sorted(rows, key=lambda r: eval(self.order, r))
            else:
                return rows
```

`execute` 方法连接了输入的表，过滤和排序结果中的行，然后将一个叫做 `make_row` 的函数映射到这些结果行上。`make_row` 函数是在 `Select` 构造函数中通过对 `create_make_row` 进行调用创建的，它是一个高阶函数，负责创建结果表的新类并定义如何将输入行投射到输出行。（一个带有更多错误处理和特殊情况处理的版本出现在 sql 中）。

```
>>> def create_make_row(description):
        """Return a function from an input environment (dict) to an output row.
        description -- a comma-separated list of [expression] as [column name]
        """
        columns = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
            expressions.append(expression)
            names.append(name)
        row = namedtuple("Row", names)
        return lambda env: row(*[eval(e, env) for e in expressions])
```

最后，我们需要定义 `join` 函数来创建输入行。给定一个包含现有表（即行的列表）的 `env` 字典，每个表用它们的名称作为键，`join` 函数使用 `itertools` 包中的 product 函数将输入表中的所有行组合在一起。然后，它会映射一个叫做 `make_env` 的函数到连接的行上，这个函数会将每个行组合转换成一个字典，以便可以用来计算表达式。（一个带有更多错误处理和特殊情况处理的版本出现在 sql 中）。

```
>>> from itertools import product
>>> def join(tables, env):
        """Return an iterator over dictionaries from names to values in a row.
        tables -- a comma-separate sequences of table names
        env    -- a dictionary from global names to tables
        """
        names = tables.split(", ")
        joined_rows = product(*[env[name] for name in names])
        return map(lambda rows: make_env(rows, names), joined_rows)
>>> def make_env(rows, names):
        """Create an environment of names bound to values."""
        env = dict(zip(names, rows))
        for row in rows:
            for name in row._fields:
                env[name] = getattr(row, name)
        return env
```

在上面的代码中，`row._fields` 表示包含该行的表的 `column_name._fields` 属性之所以存在，是因为 `row` 的类型是一个 `namedtuple` 类。

我们的解释器已经足够完整，可以执行 `select` 语句了。例如，我们可以计算从伯克利到其他所有城市的纬度距离，并按它们的经度排序。

```
>>> env = {"cities": cities}
>>> select = Select("name, 60*abs(latitude-38) as distance",
                    "cities", "name != 'Berkeley'", "-longitude")
>>> for row in select.execute(env):
        print(row)
Row(name='Minneapolis', distance=300)
Row(name='Cambridge', distance=240)
```

上面的例子等价于下面的 SQL 语句：

```
sqlite> select name, 60*abs(latitude-38) as distance
   ...>        from cities where name != "Berkeley" order by -longitude;
Minneapolis|420
Cambridge|240
```

我们还可以将这个结果表存储在环境中，并将其与城市表结合起来，检索每个城市的经度。

```
>>> env["distances"] = list(select.execute(env))
>>> joined = Select("cities.name as name, distance, longitude", "cities,
distances",
                    "cities.name == distances.name", None)
>>> for row in joined.execute(env):
        print(row)
Row(name='Cambridge', distance=240, longitude=71)
Row(name='Minneapolis', distance=300, longitude=93)
```

上面的例子等价于下面的 SQL 语句：

```
sqlite> select cities.name as name, distance, longitude
   ...>        from cities, distances where cities.name = distances.name;
Cambridge|240|71
Minneapolis|420|93
```

完整的 sql 示例程序还包含了一个简单的 `select` 语句解析器，以及用于 `create table` 和 `union` 的 `execute` 方法。该解释器可以正确执行到目前为止所包含的所有 SQL 语句。虽然这个简单的解释器只实现了结构化查询语言（Structured Query Language）的很小一部分，但它的结构展示了序列处理操作和查询语言之间的关系。

查询计划。声明性语言描述结果的形式，但没有明确地描述如何计算结果。该解释器总是连接、过滤、排序，然后投影输入行以计算结果行。然而，可能存在计算相同结果更有效的方法，查询解释器可以在其中进行选择。选择高效的过程来计算查询结果是数据库系统的核心特性。

例如，考虑上面的最终 `select` 语句。与其先计算 `cities` 和 `distances` 的连接，然后对结果进行过滤，不如首先通过对 `name` 列对两个表进行排序，然后在排好序的表中线性地遍历，只连接具有相同名称的行。当表很大时，从查询计划选择中获得的效率提升可能是巨大的。

4.3.5 递归 select 语句

`Select` 语句可以选择性包含一个 `with` 子句，用于生成和命名计算最终结果所需的附加表。不包括 `union` 的 `select` 语句的完整语法如下：

`with [tables] select [columns] from [names] where [condition] order by [order]`

我们已经演示了 [columns] 和 [names] 的允许值。[condition] 和 [order] 是可以对输入行进行求值的表达式。[tables] 部分是一个逗号分隔的列表，其中每个元素都是以下格式的表描述：

`[table name]([column names]) as ([select statement])`

任何 `select` 语句都可以用来描述 [tables] 中的一个表。

例如，下面的 `with` 子句声明了一个包含城市及其所在州的表 `states`。`select` 语句计算出同一州内的城市对。

```
sqlite> with
   ...>   states(city, state) as (
   ...>     select "Berkeley",  "California"    union
   ...>     select "Boston",    "Massachusetts" union
   ...>     select "Cambridge", "Massachusetts" union
   ...>     select "Chicago",   "Illinois"      union
   ...>     select "Pasadena",  "California"
   ...>   )
   ...> select a.city, b.city, a.state from states as a, states as b
   ...>        where a.state = b.state and a.city < b.city;
Berkeley|Pasadena|California
Boston|Cambridge|Massachusetts
```

在 `with` 子句中定义的表可能具有一个单独的递归情况，该情况将输出行定义为其他输出行的组合。例如，下面的 `with` 子句定义了从 5 到 15 的整数表，并选择其中的奇数值并对其进行平方。

```
sqlite> with
   ...>   ints(n) as (
   ...>     select 5 union
   ...>     select n+1 from ints where n < 15
   ...>   )
   ...> select n, n*n from ints where n % 2 = 1;
5|25
7|49
9|81
11|121
13|169
15|225
```

多个表可以在 `with` 子句中用逗号分隔定义。下面的示例从一个整数表、它们的平方和两个平方和之和中计算所有毕达哥拉斯三元组。毕达哥拉斯三元组由整数 a、b 和 c 组成，满足 $a^2+b^2=c^2$ 。

```
sqlite> with
   ...>   ints(n) as (
   ...>     select 1 union select n+1 from ints where n < 20
   ...>   ),
   ...>   squares(x, xx) as (
   ...>     select n, n*n from ints
   ...>   ),
   ...>   sum_of_squares(a, b, sum) as (
   ...>     select a.x, b.x, a.xx + b.xx
   ...>            from squares as a, squares as b where a.x < b.x
   ...>   )
   ...> select a, b, x from squares, sum_of_squares where sum = xx;
3|4|5
6|8|10
5|12|13
9|12|15
8|15|17
12|16|20
```

设计递归查询涉及确保每个输入行中都有足够的信息来计算结果行。例如，为了计算斐波那契数列，输入行不仅需要当前元素，还需要前一个元素才能计算下一个元素。

```
sqlite> with
   ...>   fib(previous, current) as (
   ...>     select 0, 1 union
   ...>     select current, previous+current from fib
   ...>     where current <= 100
   ...>   )
   ...> select previous from fib;
0
1
1
2
3
5
8
13
21
34
55
89
```

这些示例演示了递归是一种强大的组合手段，即使在声明性语言中也是如此。

构建字符串：在 SQL 中，可以使用 `||` 运算符将两个字符串连接成一个更长的字符串。

```
sqlite> with wall(n) as (
  ....>   select 99 union select 98 union select 97
  ....> )
  ....> select n || " bottles" from wall;
99 bottles
98 bottles
97 bottles
```

这个特性可以用于通过连接短语来构造句子。例如，构建英语句子的一种方式是将主语名词短语、动词和宾语名词短语连接起来。

```
sqlite> create table nouns as
  ....>   select "the dog" as phrase union
  ....>   select "the cat"           union
  ....>   select "the bird";
sqlite> select subject.phrase || " chased " || object.phrase
  ....>        from nouns as subject, nouns as object
  ....>        where subject.phrase != object.phrase;
the bird chased the cat
the bird chased the dog
the cat chased the bird
the cat chased the dog
the dog chased the bird
the dog chased the cat
```

作为一种练习，使用递归本地表生成句子，例如，“追逐猫咪的狗也追逐了鸟”。

> 译者注：原文为 the dog that chased the cat that chased the bird also chased the bird

4.3.6 聚合和分组

到目前为止介绍的 `select` 语句可以连接、投影和操作单个行。此外，`select` 语句可以对多行执行聚合操作。聚合函数 `max`、`min`、`count` 和 `sum` 返回列中值的最大值、最小值、数量和总和。可以通过定义多个列将多个聚合函数应用于同一组行。只有 `where` 子句包含的列会在聚合中被考虑。

```
sqlite> create table animals as
  ....>   select "dog" as name, 4 as legs, 20 as weight union
  ....>   select "cat"        , 4        , 10           union
  ....>   select "ferret"     , 4        , 10           union
  ....>   select "t-rex"      , 2        , 12000        union
  ....>   select "penguin"    , 2        , 10           union
  ....>   select "bird"       , 2        , 6;
sqlite> select max(legs) from animals;
4
sqlite> select sum(weight) from animals;
12056
sqlite> select min(legs), max(weight) from animals where name <> "t-rex";
2|20
The distinct keyword ensures that no repeated values in a column are included in
the aggregation. Only two distinct values of legs appear in the animals table.
The special count(*) syntax counts the number of rows.

sqlite> select count(legs) from animals;
6
sqlite> select count(*) from animals;
6
sqlite> select count(distinct legs) from animals;
2
```

每个 `select` 语句都生成了一个只有单行的表。`select` 语句中的 `group by` 和 `having` 子句用于将行分成组，并仅选择部分组。`having` 子句或列描述中的任何聚合函数都将独立应用于每个组，而不是表中整个行集。

例如，要从该表中计算四足动物和二足动物的最大体重，以下第一条语句将狗和猫分为一组，将鸟作为另一组，然后进行分组。结果表明，二足动物的最大体重为 3（即鸟），四足动物的最大体重为 20（即狗）。第二个查询列出 `legs` 列中至少有两个不同名称的值。

```
sqlite> select legs, max(weight) from animals group by legs;
2|12000
4|20
sqlite> select weight from animals group by weight having count(*)>1;
10
```

在 `group by` 子句中可以出现多个列和完整表达式，并且将为每个产生的唯一值组形成分组。通常，用于分组的表达式也会出现在列描述中，以便可以轻松地识别哪些结果行来自哪个组。

```
sqlite> select max(name) from animals group by legs, weight order by name;
bird
dog
ferret
penguin
t-rex
```

```
sqlite> select max(name), legs, weight from animals group by legs, weight
  ....>   having max(weight) < 100;
bird|2|6
penguin|2|10
ferret|4|10
dog|4|20
sqlite> select count(*), weight/legs from animals group by weight/legs;
2|2
1|3
2|5
1|6000
```

`having` 子句可以包含与 `where` 子句相同的过滤条件，但也可以包括聚合函数的调用。为了获得最快的执行速度和最清晰的语言使用方式，应将基于其内容过滤单个行的条件放在 `where` 子句中，而只有在需要聚合条件时才应使用 `having` 子句（例如指定组的最小计数）。

使用 `group by` 子句时，列描述中可以包含不进行聚合的表达式。在某些情况下，SQL 解释器将选择对应于包括聚合的另一列的行的值。例如，以下语句给出了具有最大重量的动物的名称。

```
sqlite> select name, max(weight) from animals;
t-rex|12000
sqlite> select name, legs, max(weight) from animals group by legs;
t-rex|2|12000
dog|4|20
```

然而，每当对应于聚合的行不清楚时（例如，使用 `count` 进行聚合而不是 `max` ），所选择的值可能是任意的。为了最清晰和最可预测地使用语言，包括 `group by` 子句的 `select` 语句应至少包括一个聚合列，并且只包含非聚合列，如果它们的内容可以从聚合中预测。