**10个简单步骤，完全理解SQL**

**1、 SQL 是一种声明式语言**

首先要把这个概念记在脑中：“声明”。 SQL 语言是为计算机声明了一个你想从原始数据中获得什么样的结果的一个范例，而不是告诉计算机如何能够得到结果。这是不是很棒？

（译者注：简单地说，SQL 语言声明的是结果集的属性，计算机会根据 SQL 所声明的内容来从数据库中挑选出符合声明的数据，而不是像传统编程思维去指示计算机如何操作。）

MySQL



|  |  |
| --- | --- |
| 1 | SELECT first\_name, last\_name FROM employees WHERE salary > 100000 |

上面的例子很容易理解，我们不关心这些雇员记录从哪里来，我们所需要的只是那些高薪者的数据（译者注： salary>100000 ）。

**我们从哪儿学习到这些？**

如果 SQL 语言这么简单，那么是什么让人们“闻 SQL 色变”？主要的原因是：我们潜意识中的是按照命令式编程的思维方式思考问题的。就好像这样：“电脑，先执行这一步，再执行那一步，但是在那之前先检查一下是否满足条件 A 和条件 B ”。例如，用变量传参、使用循环语句、迭代、调用函数等等，都是这种命令式编程的思维惯式。

**2、 SQL 的语法并不按照语法顺序执行**

SQL 语句有一个让大部分人都感到困惑的特性，就是：SQL 语句的执行顺序跟其语句的语法顺序并不一致。SQL 语句的语法顺序是：

* SELECT[DISTINCT]
* FROM
* WHERE
* GROUP BY
* HAVING
* UNION
* ORDER BY

为了方便理解，上面并没有把所有的 SQL 语法结构都列出来，但是已经足以说明 SQL 语句的语法顺序和其执行顺序完全不一样，就以上述语句为例，其执行顺序为：

* FROM
* WHERE
* GROUP BY
* HAVING
* SELECT
* DISTINCT
* UNION
* ORDER BY

关于 SQL 语句的执行顺序，有三个值得我们注意的地方：

1、 FROM 才是 SQL 语句执行的第一步，并非 SELECT 。数据库在执行 SQL 语句的第一步是将数据从硬盘加载到数据缓冲区中，以便对这些数据进行操作。（译者注：原文为“The first thing that happens is loading data from the disk into memory, in order to operate on such data.”，但是并非如此，以 Oracle 等常用数据库为例，数据是从硬盘中抽取到数据缓冲区中进行操作。）

2、 SELECT 是在大部分语句执行了之后才执行的，严格的说是在 FROM 和 GROUP BY 之后执行的。理解这一点是非常重要的，这就是你不能在 WHERE 中使用在 SELECT 中设定别名的字段作为判断条件的原因。

MySQL



|  |  |
| --- | --- |
| 1  2  3 | SELECT A.x + A.y AS z  FROM A  WHERE z = 10 -- z 在此处不可用，因为SELECT是最后执行的语句！ |

如果你想重用别名z，你有两个选择。要么就重新写一遍 z 所代表的表达式：

MySQL



|  |  |
| --- | --- |
| 1  2  3 | SELECT A.x + A.y AS z  FROM A  WHERE (A.x + A.y) = 10 |

…或者求助于衍生表、通用数据表达式或者视图，以避免别名重用。请看下文中的例子。

3、 无论在语法上还是在执行顺序上， UNION 总是排在在 ORDER BY 之前。很多人认为每个 UNION 段都能使用 ORDER BY 排序，但是根据 SQL 语言标准和各个数据库 SQL 的执行差异来看，这并不是真的。尽管某些数据库允许 SQL 语句对子查询（subqueries）或者派生表（derived tables）进行排序，但是这并不说明这个排序在 UNION 操作过后仍保持排序后的顺序。

注意：并非所有的数据库对 SQL 语句使用相同的解析方式。如 MySQL、PostgreSQL和 SQLite 中就不会按照上面第二点中所说的方式执行。

**我们学到了什么？**

既然并不是所有的数据库都按照上述方式执行 SQL 预计，那我们的收获是什么？我们的收获是永远要记得： SQL 语句的语法顺序和其执行顺序并不一致，这样我们就能避免一般性的错误。如果你能记住 SQL 语句语法顺序和执行顺序的差异，你就能很容易的理解一些很常见的 SQL 问题。

当然，如果一种语言被设计成语法顺序直接反应其语句的执行顺序，那么这种语言对程序员是十分友好的，这种编程语言层面的设计理念已经被微软应用到了 LINQ 语言中。

**3、 SQL 语言的核心是对表的引用（table references）**

由于 SQL 语句语法顺序和执行顺序的不同，很多同学会认为SELECT 中的字段信息是 SQL 语句的核心。其实真正的核心在于对表的引用。

根据 SQL 标准，FROM 语句被定义为：

MySQL



|  |  |
| --- | --- |
| 1 | <from clause> ::= FROM <table reference> [ { <comma> <table reference> }... ] |

FROM 语句的“输出”是一张联合表，来自于所有引用的表在某一维度上的联合。我们们慢慢来分析：

MySQL



|  |  |
| --- | --- |
| 1 | FROM a, b |

上面这句 FROM 语句的输出是一张联合表，联合了表 a 和表 b 。如果 a 表有三个字段， b 表有 5 个字段，那么这个“输出表”就有 8 （ =5+3）个字段。

这个联合表里的数据是 a\*b，即 a 和 b 的笛卡尔积。换句话说，也就是 a 表中的每一条数据都要跟 b 表中的每一条数据配对。如果 a 表有3 条数据， b 表有 5 条数据，那么联合表就会有 15 （ =5\*3）条数据。

FROM 输出的结果被 WHERE 语句筛选后要经过 GROUP BY 语句处理，从而形成新的输出结果。我们后面还会再讨论这方面问题。

如果我们从集合论（关系代数）的角度来看，一张数据库的表就是一组数据元的关系，而每个 SQL 语句会改变一种或数种关系，从而产生出新的数据元的关系（即产生新的表）。

**我们学到了什么？**

思考问题的时候从表的角度来思考问题提，这样很容易理解数据如何在 SQL 语句的“流水线”上进行了什么样的变动。

**4、 灵活引用表能使 SQL 语句变得更强大**

灵活引用表能使 SQL 语句变得更强大。一个简单的例子就是 JOIN 的使用。严格的说 JOIN 语句并非是 SELECT 中的一部分，而是一种特殊的表引用语句。 SQL 语言标准中表的连接定义如下：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4 | <table reference> ::=      <table name>    | <derived table>    | <joined table> |

就拿之前的例子来说：

MySQL



|  |  |
| --- | --- |
| 1 | FROM a, b |

a 可能输如下表的连接：

MySQL



|  |  |
| --- | --- |
| 1 | a1 JOIN a2 ON a1.id = a2.id |

将它放到之前的例子中就变成了：

MySQL



|  |  |
| --- | --- |
| 1 | FROM a1 JOIN a2 ON a1.id = a2.id, b |

尽管将一个连接表用逗号跟另一张表联合在一起并不是常用作法，但是你的确可以这么做。结果就是，最终输出的表就有了 a1+a2+b 个字段了。

（译者注：原文这里用词为 degree ，译为维度。如果把一张表视图化，我们可以想象每一张表都是由横纵两个维度组成的，横向维度即我们所说的字段或者列，英文为columns；纵向维度即代表了每条数据，英文为 record ，根据上下文，作者这里所指的应该是字段数。）

在 SQL 语句中派生表的应用甚至比表连接更加强大，下面我们就要讲到表连接。

**我们学到了什么？**

思考问题时，要从表引用的角度出发，这样就很容易理解数据是怎样被 SQL 语句处理的，并且能够帮助你理解那些复杂的表引用是做什么的。

更重要的是，要理解 JOIN 是构建连接表的关键词，并不是 SELECT 语句的一部分。有一些数据库允许在 INSERT 、 UPDATE 、 DELETE 中使用 JOIN 。

**5、 SQL 语句中推荐使用表连接**

我们先看看刚刚这句话：

MySQL



|  |  |
| --- | --- |
| 1 | FROM a, b |

高级 SQL 程序员也许学会给你忠告：尽量不要使用逗号来代替 JOIN 进行表的连接，这样会提高你的 SQL 语句的可读性，并且可以避免一些错误。

利用逗号来简化 SQL 语句有时候会造成思维上的混乱，想一下下面的语句：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5 | FROM a, b, c, d, e, f, g, h  WHERE a.a1 = b.bx  AND a.a2 = c.c1  AND d.d1 = b.bc  -- etc... |

我们不难看出使用 JOIN 语句的好处在于：

* 安全。 JOIN 和要连接的表离得非常近，这样就能避免错误。
* 更多连接的方式，JOIN 语句能去区分出来外连接和内连接等。

**我们学到了什么？**

记着要尽量使用 JOIN 进行表的连接，永远不要在 FROM 后面使用逗号连接表。

**6、 SQL 语句中不同的连接操作**

SQL 语句中，表连接的方式从根本上分为五种：

* EQUI JOIN
* SEMI JOIN
* ANTI JOIN
* CROSS JOIN
* DIVISION

**EQUI JOIN**

这是一种最普通的 JOIN 操作，它包含两种连接方式：

* INNER JOIN（或者是 JOIN ）
* OUTER JOIN（包括： LEFT 、 RIGHT、 FULL OUTER JOIN）

用例子最容易说明其中区别：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | -- This table reference contains authors and their books.  -- There is one record for each book and its author.  -- authors without books are NOT included  author JOIN book ON author.id = book.author\_id    -- This table reference contains authors and their books  -- There is one record for each book and its author.  -- ... OR there is an "empty" record for authors without books  -- ("empty" meaning that all book columns are NULL)  author LEFT OUTER JOIN book ON author.id = book.author\_id |

**SEMI JOIN**

这种连接关系在 SQL 中有两种表现方式：使用 IN，或者使用 EXISTS。“ SEMI ”在拉丁文中是“半”的意思。这种连接方式是只连接目标表的一部分。这是什么意思呢？再想一下上面关于作者和书名的连接。我们想象一下这样的情况：我们不需要作者 / 书名这样的组合，只是需要那些在书名表中的书的作者信息。那我们就能这么写：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | -- Using IN  FROM author  WHERE author.id IN (SELECT book.author\_id FROM book)    -- Using EXISTS  FROM author  WHERE EXISTS (SELECT 1 FROM book WHERE book.author\_id = author.id) |

尽管没有严格的规定说明你何时应该使用 IN ，何时应该使用 EXISTS ，但是这些事情你还是应该知道的：

* IN比 EXISTS 的可读性更好
* EXISTS 比IN 的表达性更好（更适合复杂的语句）
* 二者之间性能没有差异（但对于某些数据库来说性能差异会非常大）

因为使用 INNER JOIN 也能得到书名表中书所对应的作者信息，所以很多初学者机会认为可以通过 DISTINCT 进行去重，然后将 SEMI JOIN 语句写成这样：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4 | -- Find only those authors who also have books  SELECT DISTINCT first\_name, last\_name  FROM author  JOIN book ON author.id = book.author\_id |

这是一种很糟糕的写法，原因如下：

* SQL 语句性能低下：因为去重操作（ DISTINCT ）需要数据库重复从硬盘中读取数据到内存中。（译者注： DISTINCT 的确是一种很耗费资源的操作，但是每种数据库对于 DISTINCT 的操作方式可能不同）。
* 这么写并非完全正确：尽管也许现在这么写不会出现问题，但是随着 SQL 语句变得越来越复杂，你想要去重得到正确的结果就变得十分困难。

更多的关于滥用 DISTINCT 的危害可以参考这篇博文

（<http://blog.jooq.org/2013/07/30/10-common-mistakes-java-developers-make-when-writing-sql/>）。

**ANTI JOIN**

这种连接的关系跟 SEMI JOIN 刚好相反。在 IN 或者 EXISTS 前加一个 NOT 关键字就能使用这种连接。举个例子来说，我们列出书名表里没有书的作者：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | -- Using IN  FROM author  WHERE author.id NOT IN (SELECT book.author\_id FROM book)    -- Using EXISTS  FROM author  WHERE NOT EXISTS (SELECT 1 FROM book WHERE book.author\_id = author.id) |

关于性能、可读性、表达性等特性也完全可以参考 SEMI JOIN。

这篇博文介绍了在使用 NOT IN 时遇到 NULL 应该怎么办，因为有一点背离本篇主题，就不详细介绍，有兴趣的同学可以读一下

（<http://blog.jooq.org/2012/01/27/sql-incompatibilities-not-in-and-null-values/>）。

**CROSS JOIN**

这个连接过程就是两个连接的表的乘积：即将第一张表的每一条数据分别对应第二张表的每条数据。我们之前见过，这就是逗号在 FROM 语句中的用法。在实际的应用中，很少有地方能用到 CROSS JOIN，但是一旦用上了，你就可以用这样的 SQL语句表达：

MySQL



|  |  |
| --- | --- |
| 1  2 | -- Combine every author with every book  author CROSS JOIN book |

**DIVISION**

DIVISION 的确是一个怪胎。简而言之，如果 JOIN 是一个乘法运算，那么 DIVISION 就是 JOIN 的逆过程。DIVISION 的关系很难用 SQL 表达出来，介于这是一个新手指南，解释 DIVISION 已经超出了我们的目的。但是有兴趣的同学还是可以来看看这三篇文章

（<http://blog.jooq.org/2012/03/30/advanced-sql-relational-division-in-jooq/>）

（[http://en.wikipedia.org/wiki/Relational\_algebra#Division](http://blog.jobbole.com/55086/#Division)）

（<https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division/>）。

**推荐阅读 →\_→ 《[画图解释SQL联合语句](http://blog.jobbole.com/40443/" \t "_blank)》**

**我们学到了什么？**

学到了很多！让我们在脑海中再回想一下。 SQL 是对表的引用， JOIN 则是一种引用表的复杂方式。但是 SQL 语言的表达方式和实际我们所需要的逻辑关系之间是有区别的，并非所有的逻辑关系都能找到对应的 JOIN 操作，所以这就要我们在平时多积累和学习关系逻辑，这样你就能在以后编写 SQL 语句中选择适当的 JOIN 操作了。

**7、 SQL 中如同变量的派生表**

在这之前，我们学习到过 SQL 是一种声明性的语言，并且 SQL 语句中不能包含变量。但是你能写出类似于变量的语句，这些就叫做派生表：

说白了，所谓的派生表就是在括号之中的子查询：

MySQL



|  |  |
| --- | --- |
| 1  2 | -- A derived table  FROM (SELECT \* FROM author) |

需要注意的是有些时候我们可以给派生表定义一个相关名（即我们所说的别名）。

MySQL



|  |  |
| --- | --- |
| 1  2 | -- A derived table with an alias  FROM (SELECT \* FROM author) a |

派生表可以有效的避免由于 SQL 逻辑而产生的问题。举例来说：如果你想重用一个用 SELECT 和 WHERE 语句查询出的结果，这样写就可以（以 Oracle 为例）：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | -- Get authors' first and last names, and their age in days  SELECT first\_name, last\_name, age  FROM (    SELECT first\_name, last\_name, current\_date - date\_of\_birth age    FROM author  )  -- If the age is greater than 10000 days  WHERE age > 10000 |

需要我们注意的是：在有些数据库，以及 SQL ： 1990 标准中，派生表被归为下一级——通用表语句（ common table experssion）。这就允许你在一个 SELECT 语句中对派生表多次重用。上面的例子就（几乎）等价于下面的语句：

MySQL



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | WITH a AS (    SELECT first\_name, last\_name, current\_date - date\_of\_birth age    FROM author  )  SELECT \*  FROM a  WHERE age > 10000 |

当然了，你也可以给“ a ”创建一个单独的视图，这样你就可以在更广泛的范围内重用这个派生表了。更多信息可以阅读下面的文章（[http://en.wikipedia.org/wiki/View\_%28SQL%29](http://en.wikipedia.org/wiki/View_(SQL))）。

**我们学到了什么？**

我们反复强调，大体上来说 SQL 语句就是对表的引用，而并非对字段的引用。要好好利用这一点，不要害怕使用派生表或者其他更复杂的语句。

**8、 SQL 语句中 GROUP BY 是对表的引用进行的操作**

让我们再回想一下之前的 FROM 语句：

MySQL



|  |  |
| --- | --- |
| 1 | FROM a, b |

现在，我们将 GROUP BY 应用到上面的语句中：

MySQL



|  |  |
| --- | --- |
| 1 | GROUP BY A.x, A.y, B.z |

上面语句的结果就是产生出了一个包含三个字段的新的表的引用。我们来仔细理解一下这句话：当你应用 GROUP BY 的时候， SELECT 后没有使用聚合函数的列，都要出现在 GROUP BY 后面。（译者注：原文大意为“当你是用 GROUP BY 的时候，你能够对其进行下一级逻辑操作的列会减少，包括在 SELECT 中的列”）。

* 需要注意的是：其他字段能够使用聚合函数：

MySQL



|  |  |
| --- | --- |
| 1  2  3 | SELECT A.x, A.y, SUM(A.z)  FROM A  GROUP BY A.x, A.y |

* 还有一点值得留意的是： MySQL 并不坚持这个标准，这的确是令人很困惑的地方。（译者注：这并不是说 MySQL 没有 GROUP BY 的功能）但是不要被 MySQL 所迷惑。 GROUP BY 改变了对表引用的方式。你可以像这样既在 SELECT 中引用某一字段，也在 GROUP BY 中对其进行分组。

**我们学到了什么？**

GROUP BY，再次强调一次，是在表的引用上进行了操作，将其转换为一种新的引用方式。

**9、 SQL 语句中的 SELECT 实质上是对关系的映射**

我个人比较喜欢“映射”这个词，尤其是把它用在关系代数上。（译者注：原文用词为 projection ，该词有两层含义，第一种含义是预测、规划、设计，第二种意思是投射、映射，经过反复推敲，我觉得这里用映射能够更直观的表达出 SELECT 的作用）。一旦你建立起来了表的引用，经过修改、变形，你能够一步一步的将其映射到另一个模型中。 SELECT 语句就像一个“投影仪”，我们可以将其理解成一个将源表中的数据按照一定的逻辑转换成目标表数据的函数。

通过 SELECT语句，你能对每一个字段进行操作，通过复杂的表达式生成所需要的数据。

SELECT 语句有很多特殊的规则，至少你应该熟悉以下几条：

1. 你仅能够使用那些能通过表引用而得来的字段；
2. 如果你有 GROUP BY 语句，你只能够使用 GROUP BY 语句后面的字段或者聚合函数；
3. 当你的语句中没有 GROUP BY 的时候，可以使用开窗函数代替聚合函数；
4. 当你的语句中没有 GROUP BY 的时候，你不能同时使用聚合函数和其它函数；
5. 有一些方法可以将普通函数封装在聚合函数中；
6. ……

一些更复杂的规则多到足够写出另一篇文章了。比如：为何你不能在一个没有 GROUP BY 的 SELECT 语句中同时使用普通函数和聚合函数？（上面的第 4 条）

原因如下：

1. 凭直觉，这种做法从逻辑上就讲不通。
2. 如果直觉不能够说服你，那么语法肯定能。 SQL : 1999 标准引入了 GROUPING SETS，SQL： 2003 标准引入了 group sets : GROUP BY() 。无论什么时候，只要你的语句中出现了聚合函数，而且并没有明确的 GROUP BY 语句，这时一个不明确的、空的 GROUPING SET 就会被应用到这段 SQL 中。因此，原始的逻辑顺序的规则就被打破了，映射（即 SELECT ）关系首先会影响到逻辑关系，其次就是语法关系。（译者注：这段话原文就比较艰涩，可以简单理解如下：在既有聚合函数又有普通函数的 SQL 语句中，如果没有 GROUP BY 进行分组，SQL 语句默认视整张表为一个分组，当聚合函数对某一字段进行聚合统计的时候，引用的表中的每一条 record 就失去了意义，全部的数据都聚合为一个统计值，你此时对每一条 record 使用其它函数是没有意义的）。

糊涂了？是的，我也是。我们再回过头来看点浅显的东西吧。

**我们学到了什么？**

SELECT 语句可能是 SQL 语句中最难的部分了，尽管他看上去很简单。其他语句的作用其实就是对表的不同形式的引用。而 SELECT 语句则把这些引用整合在了一起，通过逻辑规则将源表映射到目标表，而且这个过程是可逆的，我们可以清楚的知道目标表的数据是怎么来的。

想要学习好 SQL 语言，就要在使用 SELECT 语句之前弄懂其他的语句，虽然 SELECT 是语法结构中的第一个关键词，但它应该是我们最后一个掌握的。

**10、 SQL 语句中的几个简单的关键词： DISTINCT ， UNION ， ORDER BY 和 OFFSET**

在学习完复杂的 SELECT 豫剧之后，我们再来看点简单的东西：

* 集合运算（ DISTINCT 和 UNION ）
* 排序运算（ ORDER BY，OFFSET…FETCH）

**集合运算（ set operation）：**

集合运算主要操作在于集合上，事实上指的就是对表的一种操作。从概念上来说，他们很好理解：

* DISTINCT 在映射之后对数据进行去重
* UNION 将两个子查询拼接起来并去重
* UNION ALL 将两个子查询拼接起来但不去重
* EXCEPT 将第二个字查询中的结果从第一个子查询中去掉
* INTERSECT 保留两个子查询中都有的结果并去重

**排序运算（ ordering operation）：**

排序运算跟逻辑关系无关。这是一个 SQL 特有的功能。排序运算不仅在 SQL 语句的最后，而且在 SQL 语句运行的过程中也是最后执行的。使用 ORDER BY 和 OFFSET…FETCH 是保证数据能够按照顺序排列的最有效的方式。其他所有的排序方式都有一定随机性，尽管它们得到的排序结果是可重现的。

OFFSET…SET是一个没有统一确定语法的语句，不同的数据库有不同的表达方式，如 MySQL 和 PostgreSQL 的 LIMIT…OFFSET、SQL Server 和 Sybase 的 TOP…START AT 等。具体关于 OFFSET..FETCH 的不同语法可以参考这篇文章

（<http://www.jooq.org/doc/3.1/manual/sql-building/sql-statements/select-statement/limit-clause/>）。

**让我们在工作中尽情的使用 SQL！**

正如其他语言一样，想要学好 SQL 语言就要大量的练习。上面的 10 个简单的步骤能够帮助你对你每天所写的 SQL 语句有更好的理解。另一方面来讲，从平时常见的错误中也能积累到很多经验。下面的两篇文章就是介绍一些 JAVA 和其他开发者所犯的一些常见的 SQL 错误：

* [10 Common Mistakes Java Developers Make when Writing SQL](http://blog.jooq.org/2013/07/30/10-common-mistakes-java-developers-make-when-writing-sql/)
* [10 More Common Mistakes Java Developers Make when Writing SQL](http://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql/)