20 数据汇总优化查询方案设计

更新时间：2020-05-11 09:27:46

> 没有智慧的头脑，就象没有腊烛的灯笼。——列夫·托尔斯泰

数据汇总优化和数据统计的概念非常类似，但是，它们却是两类完全不同的业务需求。数据汇总优化的目的是优化，实现是汇总，是针对大数据量查询缓慢而提出的解决方案。理解任何的方案设计，都是建立在实际案例的基础之上，接下来，我们就一起探讨下优化查询中的数据汇总。

1 认识数据汇总

单独去说数据汇总的概念比较简单，但是可能听完之后还是“云里雾里”。下面，我先去说两个实际在项目中遇到的场景，提出它们存在的问题。之后，分析怎样用数据汇总解决问题。最后，讨论数据汇总存在的优缺点。

1.1 需要数据汇总的两个场景

这里我将要介绍的两个场景都来自于广告系统，也都是我在日常开发中遇到的场景（实际的会稍微复杂一些，我做了一些精简，不过，核心是思想）。据此，仔细分析你的业务系统，你会发现其中存在着大量的相似性。

其实，不只是广告系统，任何业务系统都几乎会有自己的报表模块（业务）。报表模块用于报告、分析当前业务系统的表现，对于广告系统来说，就是经典的：点（广告点击）、展（广告展示）、消（广告消费）。下面，我给出一张示例表的创建语句，建表并向其中插入一些数据。

```
-- daily_creative_stat 表：每一天的创意信息
CREATE TABLE IF NOT EXISTS `daily_creative_stat` (
  `user_id` bigint(20) NOT NULL COMMENT '关联创意所属用户',
  `plan_id` bigint(20) NOT NULL COMMENT '关联推广计划 id',
  `unit_id` bigint(20) NOT NULL COMMENT '关联推广单元 id',
  `creative_id` bigint(20) NOT NULL COMMENT '关联创意 id',
  `click` bigint(20) NOT NULL COMMENT '点击次数',
  `display` bigint(20) NOT NULL COMMENT '展现次数',
  `cost` bigint(20) NOT NULL COMMENT '消费金额, 单位为分',
  `date` date NOT NULL COMMENT '数据日期'
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='创意报表信息';

-- fake 一些数据插入 daily_creative_stat 表
INSERT INTO daily_creative_stat VALUES(10001, 10, 20, 30, 108, 100, 890, '2019-12-01');
INSERT INTO daily_creative_stat VALUES(10002, 11, 21, 31, 188, 221, 760, '2019-12-01');
INSERT INTO daily_creative_stat VALUES(10003, 12, 24, 34, 121, 345, 310, '2019-12-01');
INSERT INTO daily_creative_stat VALUES(10001, 11, 26, 32, 298, 786, 407, '2019-12-01');
INSERT INTO daily_creative_stat VALUES(10001, 10, 22, 30, 211, 112, 351, '2019-12-01');
INSERT INTO daily_creative_stat VALUES(10002, 11, 21, 35, 212, 238, 765, '2019-12-02');
INSERT INTO daily_creative_stat VALUES(10001, 10, 20, 37, 345, 765, 329, '2019-12-02');
INSERT INTO daily_creative_stat VALUES(10002, 11, 21, 30, 777, 119, 342, '2019-12-02');
INSERT INTO daily_creative_stat VALUES(10001, 10, 20, 39, 209, 110, 667, '2019-12-02');
INSERT INTO daily_creative_stat VALUES(10003, 12, 24, 34, 287, 114, 665, '2019-12-02');
INSERT INTO daily_creative_stat VALUES(10001, 10, 20, 30, 231, 187, 498, '2019-12-03');
INSERT INTO daily_creative_stat VALUES(10001, 10, 20, 30, 421, 231, 238, '2019-12-03');
INSERT INTO daily_creative_stat VALUES(10003, 12, 24, 34, 877, 879, 108, '2019-12-03');
INSERT INTO daily_creative_stat VALUES(10002, 11, 27, 33, 996, 900, 211, '2019-12-03');
INSERT INTO daily_creative_stat VALUES(10002, 11, 26, 32, 251, 243, 520, '2019-12-03');
```

对于 daily_creative_stat 表，我们需要重点关注 click、display 和 cost 三个数据项，它们也就是构成报表的核心。而 user_id、plan_id、unit_id 以及 creative_id 只是关于创意的一些层级附属信息。那么，如果我想要得到这些数据：

- 用户 10001 的不同 plan_id 的分类汇总数据
- 用户 10001 的不同 plan_id、unit_id 的分类汇总数据
- 最近一周的用户数据

这些需求并不难解决，我们只需要对应着各种条件做 GROUP BY 就可以了。但是，报表数据通常都会很多，对如此大量的“原始数据” 做聚合计算不仅会降低 MySQL 服务器的性能，对我们的服务也会有很大的延迟。所以，数据汇总应运而生：

> 按照预先约定的查询需求，提前将数据进行分类聚合（按照条件进行 GROUP BY 和 SUM），并把聚合结果存储在辅助性的汇总表中。而数据汇总的最终目的是降低整体的数据量、减小数据查询难度和计算延迟。

为报表性的数据表构造汇总表是数据汇总最常见的一种场景。下面，考虑第二种场景：广告系统中的创意物料想要支持视频文件，就必须要提供文件服务器支持文件的上传、下载和存储。那么，也就需要一张文件表来记录相关信息。建表语句如下：

```
CREATE TABLE IF NOT EXISTS `file_info` (
  `name` varchar(64) NOT NULL COMMENT '文件名',
  `pro_line` varchar(64) NOT NULL COMMENT '产品线',
  `size` bigint(20) unsigned NOT NULL COMMENT '文件大小',
  PRIMARY KEY (`name`,`pro_line`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='文件信息';
```

这张表非常简单，每一条记录存储了产品线（可以理解是广告主，例如：宝马、大众）中视频物料的文件名和文件大小。但是，如果我想要知道每一个产品线上传文件的总量、删除总量、上传次数和删除次数等等这样的数据，就需要去创建汇总表解决这个需求了。即对应于每一次 file_info 表的插入、更新和删除，汇总表也必须做出相应的改动。

1.2 数据汇总的优缺点

数据汇总会给我们日常的工作需求带来便利性，但是它也是一把双刃剑，缺点也非常明显。下面，我将总结数据汇总的优缺点：

数据汇总的优点

- 减少了原始数据量，使数据计算的难度和复杂度大幅降低
- 数据的分类聚合让业务更加清晰，可读性好
- 辅助汇总表也是对原始数据的备份，提高系统数据安全性

数据汇总的缺点

- 汇总表数据来自于原始数据，所以，是冗余存储，占用额外的存储空间
- 汇总表数据需要与原始数据完全匹配，需要增加校验机制，增加了工作量

2 报表型数据汇总

如今的数据处理大致可以分为两类：OLTP（联机事务处理）和 OLAP（联机分析处理）。OLTP 是传统关系型数据库的主要应用，而我们这里所说的报表型应用则属于 OLAP，它偏向于复杂的分析操作、侧重决策支持，并且提供直观易懂的查询结果。下面，我先去讲解 OLAP 的相关知识，再去讲解 OLAP 的优化，主要目的也是想让知识点更加的系统化。

2.1 认识 OLAP

对于 OLAP，我们先不要去管它的定义，先去搞清楚两个核心概念：

- **维度**：是描述与业务主题相关的一组属性，单个属性或属性集合可以构成一个维
- **指标**：起到数据度量的作用，是数据的实际意义，即描述数据“是什么”

那么，针对于 daily_creative_stat 表来说，指标列就是 `click`、`display` 和 `cost`，其他的则属于维度列。简单来说，OLAP 就是根据维度列的组合对指标列做聚合计算（SUM）。所以，我们只需要把“对维度列的组合”搞清楚也就基本上搞清楚了 OLAP。

这里其实有一个误区，维度列的组合并不是排列组合，而是选择性组合，即挑选想要的维度列。例如，对于 daily_creative_stat 表来说，我可以这样做 OLAP 查询：

- 查询不同用户的数据汇总情况，即根据 user_id 的组合
- 查询不同用户的不同推广计划的数据汇总情况，即根据 user_id 和 plan_id 的组合
- …

查询操作是比较简单的（这也是 OLAP 的特性之一），我们只需要对需要组合的维度列做 `GROUP BY` 就可以。唯一的问题是当报表的数据量偏大时，需要花很多时间，消耗 MySQL 很多计算性能。由此，我们可以利用汇总表来优化 OLAP 的查询性能。

2.2 优化 OLAP

其实，根据前文对报表数据查询的劣势，应该可以想到我们需要创建哪些汇总表。但是，创建多少汇总表、哪些维度列组合的汇总表等等，总会有一些依据。总结如下：

- 根据需求确定需要哪些汇总表，不要盲目地一次性创建很多
- 维度列的组合不要有太大的冗余，特别是数据量级之间的规模相差不是很大的情况
- 汇总表最好包含日期（或时间）列

按照周期（例如一个周、一个月、一个季度等等）去查看数据是再正常不过的事情了，所以，汇总表包含日期通常也是个硬需求。那么，对于维度列组合的冗余指的是什么意思呢？看一看下面的例子：

```
-- 查看不同用户、不同推广计划的指标数据情况
mysql> SELECT user_id, plan_id, SUM(click), SUM(display), SUM(cost) FROM daily_creative_stat GROUP BY user_id, plan_id;
+---------+---------+------------+--------------+-----------+
| user_id | plan_id | SUM(click) | SUM(display) | SUM(cost) |
+---------+---------+------------+--------------+-----------+
|   10001 |      10 |       1525 |         1505 |      2973 |
|   10001 |      11 |        298 |          786 |       407 |
|   10002 |      11 |       2424 |         1721 |      2598 |
|   10003 |      12 |       1285 |         1338 |      1083 |
+---------+---------+------------+--------------+-----------+
4 rows in set (0.00 sec)

-- 查看不同用户的指标数据情况
mysql> SELECT user_id, SUM(click), SUM(display), SUM(cost) FROM daily_creative_stat GROUP BY user_id;
+---------+------------+--------------+-----------+
| user_id | SUM(click) | SUM(display) | SUM(cost) |
+---------+------------+--------------+-----------+
|   10001 |       1823 |         2291 |      3380 |
|   10002 |       2424 |         1721 |      2598 |
|   10003 |       1285 |         1338 |      1083 |
+---------+------------+--------------+-----------+
3 rows in set (0.00 sec)
```

如果我们确实有以上的两个查询需求，我们只需要创建一张数据汇总表即可，即 “不同用户、不同推广计划的数据汇总”，建表 SQL 如下：

```
-- 用户推广计划汇总表，即你想汇总哪些维度和指标列，表中就应该包含那些列
CREATE TABLE IF NOT EXISTS `user_plan_stat` (
  `user_id` bigint(20) NOT NULL COMMENT '关联创意所属用户',
  `plan_id` bigint(20) NOT NULL COMMENT '关联推广计划 id',
  `click` bigint(20) NOT NULL COMMENT '点击次数',
  `display` bigint(20) NOT NULL COMMENT '展现次数',
  `cost` bigint(20) NOT NULL COMMENT '消费金额, 单位为分'
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户推广计划汇总表';
```

那么，为什么不再去创建一张 “用户汇总表” 呢？这是因为 user_plan_stat 表的维度已经分的 “足够细” 了，而且表中的数据量也是偏少的。如果想要查询用户汇总数据，直接查询 user_plan_stat 表就可以。没有必要再去创建一张汇总表，造成数据冗余的同时，也增加了维护成本。

企业级开发中，大多数的查询需求可能是：某一个周、一个月的汇总数据，此时，我们可以使用 MySQL 提供的 WEEK、MONTH 等等函数实现。例如：

```
-- 查询每周的用户数据
mysql> SELECT user_id, WEEK(date), SUM(click), SUM(display), SUM(cost) FROM daily_creative_stat GROUP BY user_id, WEEK(date);
+---------+------------+------------+--------------+-----------+
| user_id | WEEK(date) | SUM(click) | SUM(display) | SUM(cost) |
+---------+------------+------------+--------------+-----------+
|   10001 |         48 |       1823 |         2291 |      3380 |
|   10002 |         48 |       2424 |         1721 |      2598 |
|   10003 |         48 |       1285 |         1338 |      1083 |
+---------+------------+------------+--------------+-----------+
3 rows in set (0.01 sec)

-- 查询12月的用户数据
mysql> SELECT user_id, MONTH(date), SUM(click), SUM(display), SUM(cost) FROM daily_creative_stat WHERE MONTH(date) = 12 GROUP BY user
_id, MONTH(date);
+---------+-------------+------------+--------------+-----------+
| user_id | MONTH(date) | SUM(click) | SUM(display) | SUM(cost) |
+---------+-------------+------------+--------------+-----------+
|   10001 |          12 |       1823 |         2291 |      3380 |
|   10002 |          12 |       2424 |         1721 |      2598 |
|   10003 |          12 |       1285 |         1338 |      1083 |
+---------+-------------+------------+--------------+-----------+
3 rows in set (0.00 sec)
```

此时，就可以去创建周、月汇总表，并把每个周、每个月的汇总数据插入其中。以后再去查询类似的数据时，只需要 SELECT 汇总表，而不需要 GROUP BY 原始报表。以此，也就实现了对 OLAP 查询报表的优化。

3 触发器实现数据汇总

很多介绍 MySQL 的书中说到：不建议使用触发器，主要是增加程序的复杂度，后期维护困难。但是，凡事无绝对，对于我在上文中讲解的“数据汇总第二种情况”来说，触发器具有天然的优势。接下来，我先去详细的讲解下触发器的概念和使用方法，再去使用触发器解决我们的需求。

3.1 初识触发器

首先来说，触发器只会用在特定的场合，它是写在数据库中的一段 SQL 脚本，当数据库表记录发生变化时（插入、删除或更新操作），触发（这也是触发器名称的由来）一条或多条 SQL 语句实现多张表的数据同步。且触发器最大的特点是：触发事件的操作和触发器里的 SQL 语句是一个事务操作，具有原子性，即要么全部执行，要么都不执行。

说到这里，你肯定会想，我用代码也可以实现呀，而且用代码可以更加灵活的控制，当然也包括调试。事实确实如此，所以，当你选择使用触发器时，一定是业务逻辑比较简单，且涉及的表比较少（最好不超过3个）。

3.2 创建触发器

创建触发器的语法是比较复杂的，如下所示：

```
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

下面，我来讲解下其中各个参数或关键字的含义：

- **trigger_name**：触发器的名称
- **trigger_time** 触发器执行的时机
    - **BEFORE**：事件发生之前
    - **AFTER**：事件发生之后
- **trigger_event**：触发事件
    - **INSERT**：插入数据时
    - **UPDATE**：更新数据时
    - **DELETE**：删除数据时
- **tbl_name**：触发器关联的表名称，可以使用 `数据库.表名` 来明确指定
- **FOR EACH ROW**：标识任何一条表记录上的操作满足触发事件都会触发该触发器
- **trigger_order**：用于定义多个触发器的执行顺序
    - **FOLLOWS**：在 … 之后
    - **PRECEDES**：在 … 之前
- **trigger_body**：触发器的主体，即触发器具体要做的事情

理解语法往往都不会很难，但是一去写发现有难度了。所以，我来写一个示例，帮你更好的理解触发器的语法。如下所示：

```
-- 设置变量 worker_salary 的值为 0
mysql> SET @worker_salary = 0;
Query OK, 0 rows affected (0.01 sec)

-- 创建触发器 worker_salary_sum， 在 worker 表插入数据发生之前（针对于每一行），让 worker_salary 增加插入记录的 salary 值
mysql> CREATE TRIGGER worker_salary_sum BEFORE INSERT ON `imooc_mysql`.`worker` FOR EACH ROW SET @worker_salary = @worker_s
alary + NEW.salary;
Query OK, 0 rows affected (0.05 sec)

-- 校验 worker_salary 的值是否是 0
mysql> SELECT @worker_salary;
+----------------+
| @worker_salary |
+----------------+
|              0 |
+----------------+
1 row in set (0.00 sec)

-- 向 worker 表中插入一行数据，指定 salary 的值是 3200
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('C', 'app', 3200, 0);
Query OK, 1 row affected (0.03 sec)

-- 校验 worker_salary 的值是否会变成 3200
mysql> SELECT @worker_salary;
+----------------+
| @worker_salary |
+----------------+
|           3200 |
+----------------+
1 row in set (0.00 sec)
```

以上的创建、执行流程并不难理解，只是需要注意我在创建触发器时使用的 NEW 关键字。其实，MySQL 提供了 OLD 和 NEW 这两个关键字对触发器进行扩展，且它们不区分大小写。关于它们，你需要知道：

- 触发事件是 INSERT 时，只有 NEW 关键字可用，它代表的是将要（BEFORE）或已经（AFTER）插入的数据记录
- 触发事件是 DELETE 时，只有 OLD 关键字可用，它代表的是将要（BEFORE）或已经（AFTER）删除的数据记录
- 触发事件是 UPDATE 时，OLD 是该行数据更新之前的副本，NEW 是更新之后的副本

3.3 查看触发器

查看触发器可以使用 `SHOW TRIGGERS` 命令，它会打印在当前数据库中的触发器信息。如果想要限定数据库，可以加上 `FROM schema_name` 子句。如下所示：

```
-- 查看 imooc_mysql 库中的触发器信息
mysql> SHOW TRIGGERS FROM imooc_mysql\G
*************************** 1. row ***************************
             Trigger: worker_salary_sum
               Event: INSERT
               Table: worker
           Statement: SET @worker_salary = @worker_salary + NEW.salary
              Timing: BEFORE
             Created: 2019-12-15 23:58:32.99
            sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
             Definer: root@localhost
character_set_client: utf8
collation_connection: utf8_general_ci
  Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

当然，系统中的触发器信息一定存储在系统库中（这也是经验之谈），如果还记得之前对 information_schema 库的介绍就可以知道，它存储在 TRIGGERS 表中。我们可以直接查询这张表查看触发器信息：

```
-- 查看 imooc_mysql 库中定义的触发器的详细信息
mysql> SELECT * FROM information_schema.TRIGGERS WHERE TRIGGER_SCHEMA = 'imooc_mysql'\G
*************************** 1. row ***************************
           TRIGGER_CATALOG: def
            TRIGGER_SCHEMA: imooc_mysql
              TRIGGER_NAME: worker_salary_sum
        EVENT_MANIPULATION: INSERT
      EVENT_OBJECT_CATALOG: def
       EVENT_OBJECT_SCHEMA: imooc_mysql
        EVENT_OBJECT_TABLE: worker
              ACTION_ORDER: 1
          ACTION_CONDITION: NULL
          ACTION_STATEMENT: SET @worker_salary = @worker_salary + NEW.salary
        ACTION_ORIENTATION: ROW
             ACTION_TIMING: BEFORE
ACTION_REFERENCE_OLD_TABLE: NULL
ACTION_REFERENCE_NEW_TABLE: NULL
  ACTION_REFERENCE_OLD_ROW: OLD
  ACTION_REFERENCE_NEW_ROW: NEW
                  CREATED: 2019-12-15 23:58:32.99
                 SQL_MODE: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_
ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
                  DEFINER: root@localhost
      CHARACTER_SET_CLIENT: utf8
      COLLATION_CONNECTION: utf8_general_ci
        DATABASE_COLLATION: latin1_swedish_ci
1 row in set (0.01 sec)
```

3.4 删除触发器

在 MySQL 中做删除操作通常都会使用到 DROP 语句，对于触发器的删除来说，也不会是个例外。下面，我来给出删除触发器的语法和演示示例（需要注意，删除触发器需要有关联触发器的表的 Trigger 特权）：

```
-- 删除触发器的语法
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name

-- 删除 imooc_mysql 库中的 worker_salary_sum 触发器
mysql> DROP TRIGGER IF EXISTS imooc_mysql.worker_salary_sum;
Query OK, 0 rows affected (0.01 sec)
```

3.5 数据汇总案例实现

到目前为止，我们已经基本上掌握了触发器的思想和编写方法，那么，下面就使用触发器来解决数据汇总的第二个场景需求吧。等等，别着急去编写触发器，我们需要先去创建一张文件信息汇总表，创建语句如下：

```
-- 文件汇总信息表
CREATE TABLE IF NOT EXISTS `file_sum_info` (
  `pro_line` varchar(64) NOT NULL COMMENT '产品线',
  `upload_size` bigint(20) unsigned NOT NULL DEFAULT 0 COMMENT '上传文件总量',
  `delete_size` bigint(20) unsigned NOT NULL DEFAULT 0 COMMENT '删除文件总量',
  `upload_count` bigint(20) unsigned NOT NULL DEFAULT 0 COMMENT '上传次数',
  `delete_count` bigint(20) unsigned NOT NULL DEFAULT 0 COMMENT '删除次数',
  PRIMARY KEY (`pro_line`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='文件汇总信息表';
```

下面，我们需要创建三个触发器（插入、更新和删除）来关联 file_info 和 file_sum_info 表（需要注意，这两张表的更新过程是原子的，这是由触发器来保证的）。首先，创建插入触发器：

```
-- 修改分隔符为 $$
DELIMITER $$
-- 创建触发器 file_insert_trigger
CREATE TRIGGER file_insert_trigger
-- 在 file_info 表插入数据之后
AFTER INSERT ON file_info
-- 对于每一行记录都执行触发器
FOR EACH ROW
-- 触发器中存在多条语句，使用 BEGIN 和 END
BEGIN
 -- 如果 file_sum_info 表中不存在当前产品线，则插入新记录，否则，更新原记录
 INSERT INTO file_sum_info(pro_line, upload_size, upload_count) VALUES(NEW.pro_line, NEW.size, 1) ON DUPLICATE KEY
 UPDATE upload_size = upload_size + NEW.size, upload_count = upload_count + 1;
END $$
-- 将分隔符修改回分号
DELIMITER ;
```

file_insert_trigger 触发器的思想非常简单：file_info 插入数据之后“按需”插入或修改 file_sum_info 表的记录。那么，对于 file_info 表更新的情况，我们也需要一个更新触发器：

```
-- 修改分隔符为 $$
DELIMITER $$
-- 创建触发器 file_update_trigger
CREATE TRIGGER file_update_trigger
-- 在 file_info 表更新数据之后
AFTER UPDATE ON file_info
-- 对于每一行记录都执行触发器
FOR EACH ROW
BEGIN
 -- 更新 file_sum_info 表的记录（pro_line 需要匹配）
 UPDATE file_sum_info SET upload_size = upload_size + NEW.size - OLD.size WHERE pro_line = NEW.pro_line;
END $$
-- 将分隔符修改回分号
delimiter ;
```

最后，当 file_info 表删除记录时，file_sum_info 表也要做相应的更新，即创建一个删除触发器。如下所示：

```
-- 修改分隔符为 $$
DELIMITER $$
-- 创建触发器 file_delete_trigger
CREATE TRIGGER file_delete_trigger
-- 在 file_info 表删除数据之后
AFTER DELETE ON file_info
-- 对于每一行记录都执行触发器
FOR EACH ROW
BEGIN
 -- 更新 file_sum_info 表的记录（pro_line 需要匹配）
 UPDATE file_sum_info SET delete_size = delete_size + OLD.Size, delete_count = delete_count + 1 WHERE pro_line = OLD.pro_line;
END $$
-- 将分隔符修改回分号
delimiter ;
```

创建好触发器之后，我们来对 file_info 表做一些增删改查的工作（最好是先清空 file_info 和 file_sum_info 表的数据，以方便验证），验证下触发器是否好用。执行的流程以及注释信息如下所示：

```
-- 插入第一条记录
mysql> INSERT INTO file_info(`name`, `pro_line`, `size`) VALUES('宝马 X6', '分众', 1024);
Query OK, 1 row affected (0.02 sec)

-- 插入第二条记录
mysql> INSERT INTO file_info(`name`, `pro_line`, `size`) VALUES('奔驰 C200', '新潮', 512);
Query OK, 1 row affected (0.02 sec)

-- 插入第三条记录，与第一条记录属于同一个产品线
mysql> INSERT INTO file_info(`name`, `pro_line`, `size`) VALUES('奥迪 Q7', '分众', 2000);
Query OK, 1 row affected (0.01 sec)

-- 查看下 file_sum_info 表的数据（应该有两条记录）
-- 记录数、upload_size 和 upload_count 都符合预期
mysql> SELECT * FROM file_sum_info;
+----------+-------------+-------------+--------------+--------------+
| pro_line | upload_size | delete_size | upload_count | delete_count |
+----------+-------------+-------------+--------------+--------------+
| 分众     |        3024 |           0 |            2 |            0 |
| 新潮     |         512 |           0 |            1 |            0 |
+----------+-------------+-------------+--------------+--------------+
2 rows in set (0.00 sec)

-- 更新记录
mysql> UPDATE file_info SET size = 1000 WHERE name = '奔驰 C200' AND pro_line = '新潮';
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- 查看下 file_sum_info 表的数据
-- pro_line 为新潮的记录的 upload_size 符合预期
mysql> SELECT * FROM file_sum_info;
+----------+-------------+-------------+--------------+--------------+
| pro_line | upload_size | delete_size | upload_count | delete_count |
+----------+-------------+-------------+--------------+--------------+
| 分众     |        3024 |           0 |            2 |            0 |
| 新潮     |        1000 |           0 |            1 |            0 |
+----------+-------------+-------------+--------------+--------------+
2 rows in set (0.00 sec)

-- 删除数据
mysql> DELETE FROM file_info WHERE name = '奥迪 Q7' AND pro_line = '分众';
Query OK, 1 row affected (0.01 sec)

-- 查看下 file_sum_info 表的数据
-- pro_line 为分众的记录的 delete_size、delete_count 符合预期
mysql> SELECT * FROM file_sum_info;
+----------+-------------+-------------+--------------+--------------+
| pro_line | upload_size | delete_size | upload_count | delete_count |
+----------+-------------+-------------+--------------+--------------+
| 分众     |        3024 |        2000 |            2 |            1 |
| 新潮     |        1000 |           0 |            1 |            0 |
+----------+-------------+-------------+--------------+--------------+
2 rows in set (0.00 sec)
```

经过了以上的验证流程，可以确定我们的触发器是正确且可用的。而且可以看出，在“某些情况下”，触发器也是非常好用的。所以，多去想一想、查一查，看看你当前的工作使用触发器是否可行。

4 总结

数据汇总这类需求通常都是类似的，所以，熟练掌握它的思想和实现是一劳永逸的。我在这一节里提到的两个场景也是出自我在工作中遇到的需求，同时，解决方案我也已经做了详细的解读。当然，你可以对它们修改，以便适用于你遇到的场景，或者干脆你有更好的实现方法，我们也可以一起交流学习。

5 问题

对于 OLAP 型的数据汇总表，如果让你来做，你会怎样做数据汇总并存储呢？

对于 OLAP 型的数据汇总表，应该在什么时机做数据插入呢？

你使用过触发器吗？是怎样使用的呢？或者说，你觉得触发器适用于哪些场景呢？

也许你听说过，触发器的效率比较低，你知道这是为什么吗？

6 参考资料

《高性能 MySQL（第三版）》

MySQL 官方文档：CREATE TRIGGER Statement

MySQL 官方文档：Using Triggers

MySQL 官方文档：MySQL 5.7 FAQ: Triggers

MySQL 官方文档：SHOW TRIGGERS Statement

MySQL 官方文档：DROP TRIGGER Statement

MySQL 官方文档：The INFORMATION_SCHEMA TRIGGERS Table

}

19 听过存储过程，但是你会用吗？

21 经常听到分库分表，但是怎样分呢？