PYTHON+MYSQL 分库分表实战

作者：HH QQ: 275258836

2016-3-6

运维生存时间
http://www.ttlsa.com/

目录

0.前言

0.1 关于我

从事 JAVA、PHP、Python 开发工作。Oracle 数据库认证专家，且与 MySQL 数据库结缘至今。

0.2 感谢

感谢我的父母：陈明、宝月，他们一直以来支持我选择的行业。并给予我支持和鼓励。
感谢我的女友：学霞，帮我料理着生活上的琐事，并包容着我生活上的恶习。
感谢 ttlsa.com：给与了我一个分享的平台。
感谢凉白开：是他在后面对我细心的指导，以至于我能坚持的写着文章。

0.3 关于本站

网站名称并没有特别的意思，运维时常接触到 TTL 以及职业本身便是 SA（system administraotr），连起来便得到了 ttlsa.com。域名有了，名称也就这么来了－运维生存时间。

0.4 发展历程

日期	历程
2010 年	8 月 4 日：购买域名
2011 年	2 月 13 日：发布本站第一篇文章，团队陆续发布上百篇技术文章
2012 年	本站放空的一年，偶尔更新文章
2013 年	6 月 22 日：重启 TTLSA 团队，用心经营本站。 将网站从国内空间迁移至 linode，并运行了将近一年时间，之后迁移至阿里云 2 个月
2014 年	7 月 7 日：迎来了本站的首次赞助，kaopuyun.com 免费提供国内云服务器。 12 月 14:Alexa 排名首次进入 10 万以内，排名 97901
2015 年	迎来了各大 APM 合作伙伴：云智慧、ONEAPM、听云等
2016 年	开启崭新的一年

本站用心更新，用户量也不断创新高，感谢作者们对 TTLSA 的无私奉献.

0.5 本站原创系列

当然，《Python + MySQL 分库分表-小实例》只是我们系列之中的一份，我们还有更多系列教程，截至 2016 年 3 月 06 日，我们一共有 7 个原创系列教程，如下：

- 《Python + MySQL 分库分表-小实例》

作者：HH

文章列表：http://www.ttlsa.com/mysql/python-mysql-split-db-tb-train/

- 《ZABBIX 教程从入门到精通》

  作者：凉白开

  文章列表：http://www.ttlsa.com/zabbix/follow-ttlsa-to-study-zabbix/

- 《NGINX 教程从入门到精通》

  作者：漠北、凉白开

  文章地址：http://www.ttlsa.com/nginx/nginx-tutorial-from-entry-to-the-master-ttlsa/

- 《Mongodb 实战教程》

  作者：漠北

  文章地址：http://www.ttlsa.com/mongodb/mongodb-study/

- 《Mongodb 官方监控：MMS》

  作者：漠北

  文章地址：http://www.ttlsa.com/mms/follow-me-to-use-mongodb-mms-services/

- 《MySQL 管理工具利器：MySQL Utilities》

  作者：漠北

  文章地址：http://www.ttlsa.com/mysql/mysql-manager-tools-mysql-utilities-tutorial/

- 《TTLSA 带你学习 Thinkphp》

  作者：tonyty163

  文章地址：http://www.ttlsa.com/php/follow-ttlsa-to-study-thinkphp/

0.6 业务合作

一切运维有关业务可与我们联系洽谈

0.7 联系我们

联系方式	
E-mail	support@ttlsa.com
QQ	1715812369
QQ 群	6690706、39514058(已满)、168085569、415230207(新)、48158813（招聘专用）

0.8 捐助

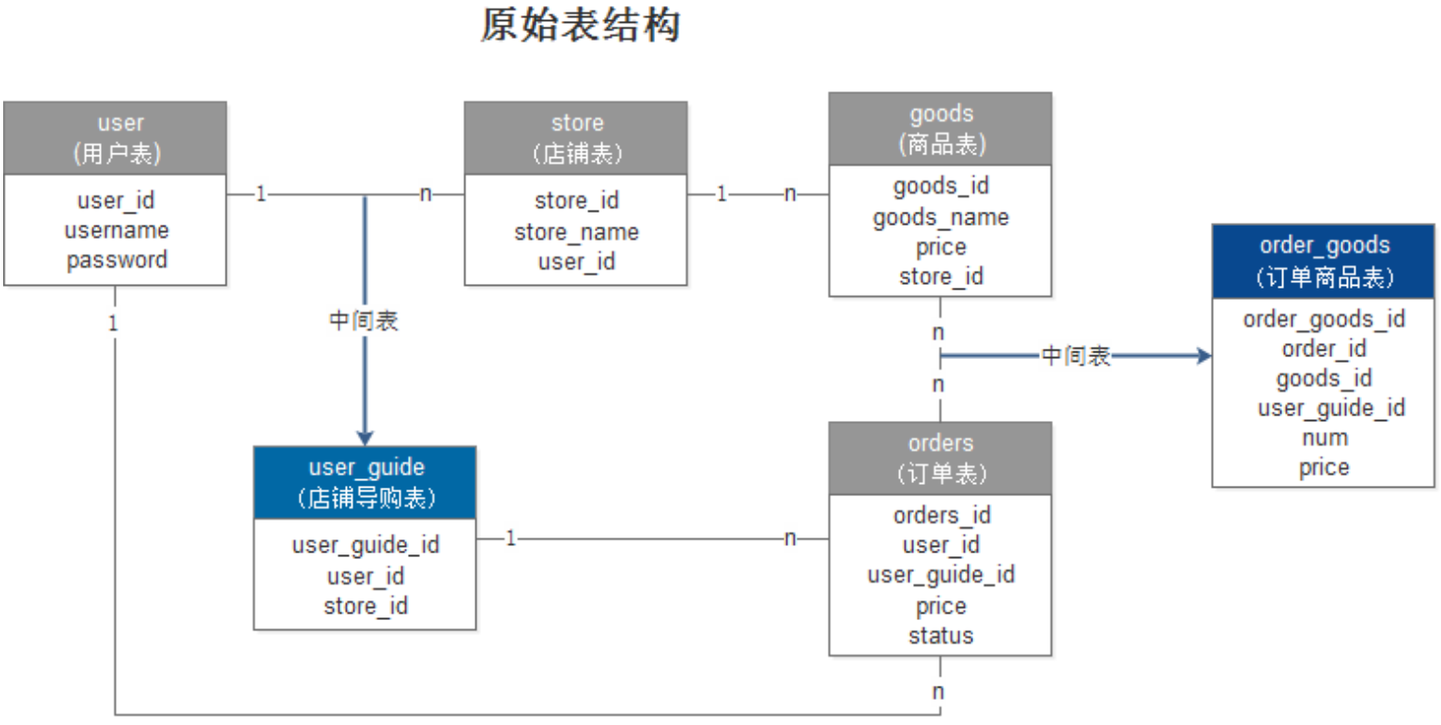
如果你喜欢我们的内容，也想支持我们，那么捐助我们吧！有钱，任性，那就打赏吧

1. MySQL 分库分表基础表介绍

1.1. 表基本模型结构

这里我们模拟一个商城的基本的表结。此结构由（用户、门店、导购、门店商品、订单、订单对应的商品）。其中，导购也是一个用户，门店是只属于一个店主的，同时店主本身也是一个导购也是一个普通用户。

结构图：



构造数据脚本:MySQL 分库分表(1)-脚本

1.2. 对业务场景进行模拟

- **场景 1：购买者下订单。**

1、从 session 中获得客户 ID。

2、可以通过时间戳等拼凑一个订单 ID（在创建表的时候为了方便我用自增的，在以下我们一直就吧订单 ID 看成不是自增的，是用程序生成的）。

3、从商品的店铺能获得到导购 ID（获取规则我们这边认为是随机）。

4、可以从商品中计算出订单价格。

最终就能拼凑出下单的 INSERT SQL 语句(这边我就不真正写插入语句了)

```
SET autocommit=0;
START TRANSACTION;
-- 创建订单语句
INSERT INTO orders VALUES(订单 ID, 导购 ID, 购买用户 ID, 订单价格, 订单状态);
-- 创建商品订单语句
INSERT INTO order_goods VALUES(NULL, 订单 ID, 商品 ID, 商品价格, 商品数量);
-- 可以给添加多个商品到订单中
......
COMMIT;
SET autocommit=1;
```

以上就是一个客户下单时要操作的，订单 ID（订单号）是程序生成的，订单 ID（订单号）是程序生成的，重要的事要说三遍。

- **情况 2：购买者浏览订单**

当用户查看订单列表的时候可以通过分页一次性获得自己的订单列表。

```
-- 每一页 10 行(这边顺便展示一下单数据量大时优化后的 sql 语句)
-- 查找用户 ID 为 100 的订单
SELECT l_o.orders_id,
  o.user_guide_id,
  o.user_id,
  o.price,
  og.price
FROM (
  SELECT orders_id
  FROM orders
  WHERE user_id = 100
  LIMIT 0, 10
) AS l_o
  LEFT JOIN orders AS o ON l_o.orders_id = o.orders_id
  LEFT JOIN order_goods AS og ON l_o.orders_id = og.orders_id;
```

- **情况 3：导购查看订单**

```
-- 每个导购也可以查看他销售了多少的订单
-- 查找导购 ID 为 1 的销售情况
SELECT o.orders_id,
  o.user_guide_id,
  o.user_id,
  o.price,
  og.price
FROM orders AS o
  LEFT JOIN order_goods AS og ON o.orders_id = og.orders_id
WHERE o.orders_id IN(
  SELECT orders_id
  FROM (
    SELECT orders_id
    FROM orders
    WHERE user_guide_id=1
    LIMIT 0, 10
  ) AS tmp
);
```

- **情况 4：导购修改订单**

```
-- 这边我们修改订单金额就好，修改 ID 为 1000 的订单
UPDATE orders SET price = '10000' WHERE orders_id=1000;
```

- **情况 5：店主为店铺添加商品**

1、我们可以根据操作的用户获得店铺名

```
-- 添加商品伪 SQL
INSERT INTO goods VALUES(NULL, 商品名, 商品价格, 店铺名);
```

2. MySQL 分库分表创建新表结构

2.1. 前言

在互联网时代大家都知道数据量是爆炸试的增加，从之前的表结构设计来看，我们很容易的知道商品表(goods)、订单表(orders)、订单商品表(order_goods)这几张表的数据量将会爆炸试的增加。

因此，在数据量达到一定程度就算是建了索引，查询使用了索引，查询、修改速度也是会降下来的。为了额能较好的克服这样的问题，我们不得不重新整理并对大数据的表进行表切分。

2.2. 分表介绍

当下有静态分表和动态分表两种：

- **静态分表：**事先估算出表能达到的量，然后根据每一个表需要存多少数据直接算出需要创建表的数量。如：1 亿数据每一个表 100W 条数据那就要建 100 张表，然后通过一定的 hash 算法计算每一条数据存放在那张表。其实就有点像是使用 partition table 一样。静态分表有一个毙命就是当分的那么多表还不满足时，需要再扩展难度和成本就会很高。
- **动态分表：**同样也是对大数据量的表进行拆分，他可以避免静态分表带来的后遗症。当然也需要在设计上多一些东西（这往往是我们能接受的）。

如果使用了分表的设计的数据库在一些查询上面会变的复杂一些。

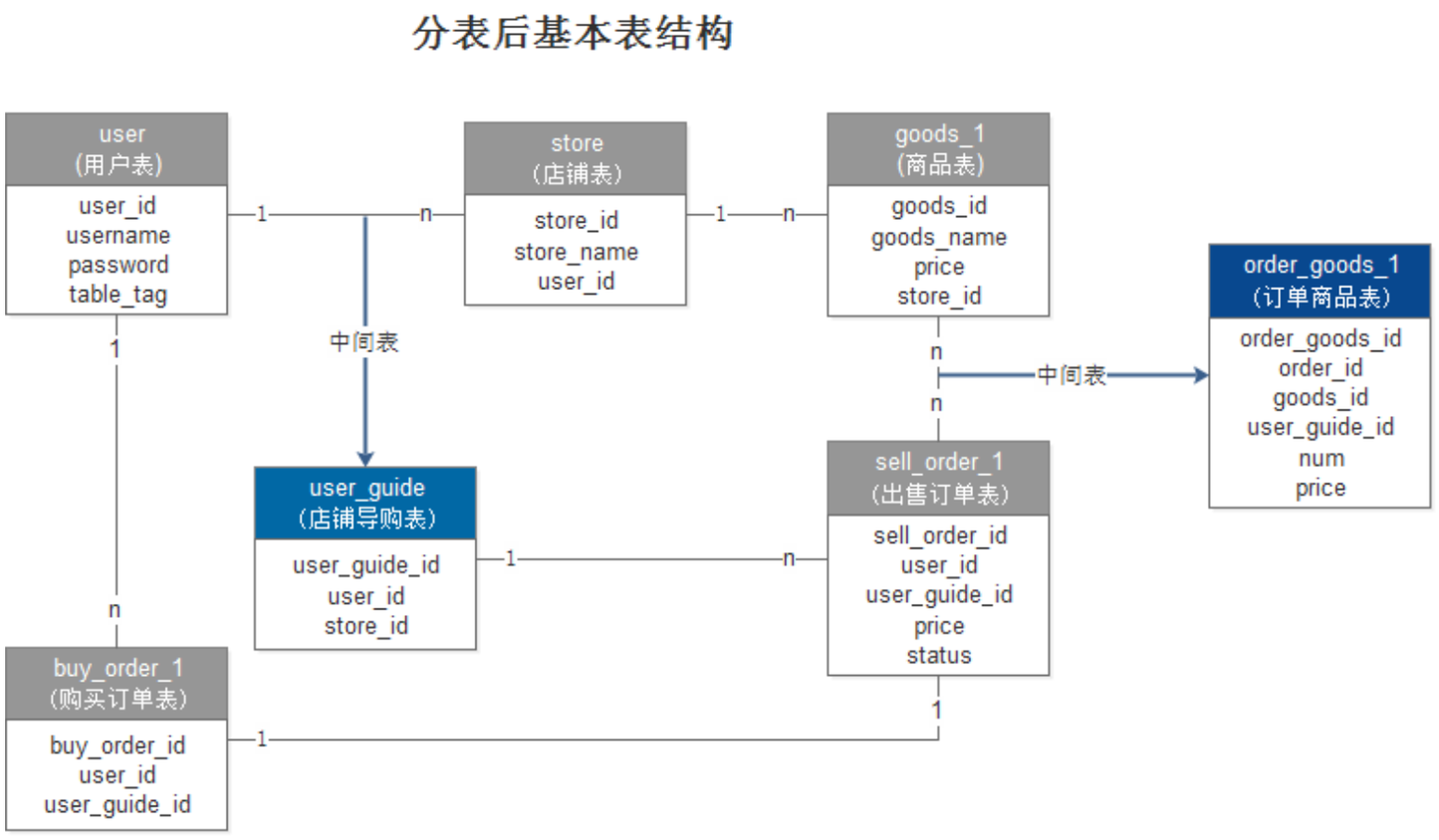
2.3. 我的选择

本着要让之后让表能更好的扩展，并能达到手工指定数据到自己想要的表，为了以后能自动化分表操作，我选择了动态分表。

2.4. 业务分解

由于在我们的业务中每一个导购除了能卖东西，还能买东西，因此在逻辑上就分为两张表：出售订单表、购买订单表。

业务分解后表结构图如下：



2.5. 我们潜规则

我们是按 user 表中的每一个用户去指定他的订单数据要在哪个表。

由于按用户分表后会涉及到是然购买者方便查询，还是让销售者方便查询的问题。我们这里选择的是让销售者查询方便来分表，因为销售者的查询和对订单的修改会频繁一些。因此，我们将出售订单表中存放着比较完整的订单信息，而在购买订单表中存放的是出售订单表的 ID 作为关联。

我们出购买订单表 ID 和售订单表 ID 保持一致。

小提示：你也可以在购买订单表中添加一些冗余字段为了更好的查询，但是建议冗余字段不要是业务上是可变的。

2.6. 业务分解后数据迁移到新表

```
-- 创建出售订单表-sell_order_1
CREATE TABLE sell_order_1 LIKE orders;
-- 修改出售订单表 ID 字段名
ALTER TABLE sell_order_1
  CHANGE orders_id sell_order_id INT unsigned NOT NULL AUTO_INCREMENT
  COMMENT '出售订单 ID';
-- 修改商品订单表的订单 ID 名为 sell_order_id
ALTER TABLE order_goods
  CHANGE orders_id sell_order_id INT unsigned NOT NULL
  COMMENT '出售订单 ID';
-- 将 orders 表数据转移到 sell_order_1 表
INSERT INTO sell_order_1
SELECT * FROM orders;

-- 迁移商品表到 goods_1
CREATE TABLE goods_1 LIKE goods;
-- 插入 goods_1 表数据
```

```
INSERT INTO goods_1
SELECT * FROM goods;

-- 迁移订单商品表到 order_goods_1
CREATE TABLE order_goods_1 LIKE order_goods;
-- 插入 order_goods_1
INSERT INTO order_goods_1
SELECT * FROM order_goods;

-- 创建购买订单表
CREATE TABLE buy_order_1(
  buy_order_id BIGINT unsigned NOT NULL COMMENT '出售订单 ID 与出售订单相等',
  user_id INT unsigned DEFAULT NULL COMMENT '下单用户 ID',
  user_guide_id INT unsigned DEFAULT NULL COMMENT '导购 ID',
  PRIMARY KEY(buy_order_id),
  KEY idx$buy_order_1$user_id(user_id),
  KEY idx$buy_order_1user_guide_id(user_guide_id)
);
-- 买订单表导入数据
INSERT INTO buy_order_1
SELECT sell_order_id,
  user_id,
  user_guide_id
FROM sell_order_1;

-- user 表增加指定表标识字段
ALTER TABLE user
  ADD table_flag TINYINT NOT NULL DEFAULT 1
  COMMENT '分表标识';
```

3. MySQL 分库分表使用 Snowflake 全局 ID 生成器

3.1. 前言

由于考虑到以后要动态切分数据，防止将不同表切分数据到同一个表中时出现主键相等的冲突情况，这里我们使用一个全局 ID 生存器。重要的是他是自增的。

这边我使用 Snowflake 的 python 实现版（pysnowflake）。当然你也可以使用 java 实现版.

具体详细信息：http://pysnowflake.readthedocs.org/en/latest/

3.2. Snowflake 的使用

安装 requests

```
pip install requests
```

安装 pysnowflake

```
pip install pysnowflake
```

启动 pysnowflake 服务

```
snowflake_start_server \
  --address=192.168.137.11 \
  --port=30001 \
  --dc=1 \
  --worker=1 \
  --log_file_prefix=/tmp/pysnowflask.log
```

这里解释一下参数意思（可以通过--help 来获取）：

--address： 本机的 IP 地址默认 localhost

--dc： 数据中心唯一标识符默认为 0

--worker： 工作者唯一标识符默认为 0

--log_file_prefix： 日志文件所在位置

使用示例（这边引用官网的）

```
# 导入 pysnowflake 客户端
>>> import snowflake.client

# 链接服务端并初始化一个 pysnowflake 客户端
>>> host = '192.168.137.11'
>>> port = 30001
>>> snowflake.client.setup(host, port)
# 生成一个全局唯一的 ID（在 MySQL 中可以用 BIGINT UNSIGNED 对应）
>>> snowflake.client.get_guid()
3631957913783762945
# 查看当前状态
>>> snowflake.client.get_stats()
{
  'dc': 1,
  'worker': 1,
  'timestamp': 1454126885629, # current timestamp for this worker
  'last_timestamp': 1454126890928, # the last timestamp that generated ID on
  'sequence': 1, # the sequence number for last timestamp
  'sequence_overload': 1, # the number of times that the sequence is overflow
```

```
  'errors': 1, # the number of times that clock went backward
}
```

3.3. 数据整理重建 ID

重建 ID 是一个很庞大的工程，首先要很了解表的结构。不然，如果少更新了某个表的一列都会导致数据的不一致。

当然，如果你的表中有很强的外键以及设置了级联那更新一个主键会更新其他相关联的外键。这里我还是不建议去依赖外键级联更新来投机取巧毕竟如果有数据库的设计在项目的里程碑中经过了 n 次变化，也不能肯定设置的外键一定是级联更新的。

在这边我强烈建议重建 ID 时候讲 MySQL 中的检查外键的参数设置为 0。

```
SET FOREIGN_KEY_CHECKS=0;
```

小提示：其实理论上我们是没有必要重建 ID 的因为原来的 ID 已经是唯一的了而且是整型，他兼容 BIGINT。但是这里我还是做了重建，主要是因为以后的数据一致。并且如果有些人的 ID 不是整型的，而是有一定含义的那时候也肯定需要做 ID 的重建。

修改相关表 ID 的数据类型为 BIGINT

```
-- 修改商品表 goods_id 字段
ALTER TABLE goods_1
  MODIFY COLUMN goods_id BIGINT UNSIGNED NOT NULL
  COMMENT '商品 ID';

-- 修改出售订单表 goods_id 字段
ALTER TABLE sell_order_1
  MODIFY COLUMN sell_order_id BIGINT UNSIGNED NOT NULL
  COMMENT '出售订单 ID';

-- 修改购买订单表 buy_order_id 字段
ALTER TABLE buy_order_1
  MODIFY COLUMN buy_order_id BIGINT UNSIGNED NOT NULL
  COMMENT '出售订单 ID 与出售订单相等';

-- 修改订单商品表 order_goods_id、orders_id、goods_id 字段
ALTER TABLE order_goods_1
  MODIFY COLUMN order_goods_id BIGINT UNSIGNED NOT NULL
  COMMENT '订单商品表 ID';
ALTER TABLE order_goods_1
  MODIFY COLUMN sell_order_id BIGINT UNSIGNED NOT NULL
  COMMENT '订单 ID';
ALTER TABLE order_goods_1
  MODIFY COLUMN goods_id BIGINT UNSIGNED NOT NULL
  COMMENT '商品 ID';
```

3.4. 使用 python 重建 ID

使用的 python 模块：

模块名	版本	备注
pysnowflake	0.1.3	全局 ID 生成器
mysql_connector_python	2.1.3	mysql python API

这边只展示主程序：完整的程序在附件中都有

```
if __name__=='__main__':
  # 设置默认的数据库链接参数
  db_config = {
```

```
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
}
# 设置 snowflake 链接默认参数
snowflake_config = {
    'host': '192.168.137.11',
    'port': 30001
}

rebuild = Rebuild()
# 设置数据库配置
rebuild.set_db_config(db_config)
# 设置 snowflak 配置
rebuild.set_snowflake_config(snowflake_config)
# 链接配置 snowflak
rebuild.setup_snowflake()

# 生成数据库链接和
rebuild.get_conn_cursor()

################################################################################
## 修改商品 ID
################################################################################
# 获得商品的游标
goods_sql = '''
    SELECT goods_id FROM goods
'''
goods_iter = rebuild.execute_select_sql([goods_sql])
# 根据获得的商品 ID 更新商品表(goods)和订单商品表(order_goods)的商品 ID
for goods in goods_iter:
    for (goods_id, ) in goods:
        rebuild.update_table_id('goods', 'goods_id', goods_id)
        rebuild.update_table_id('order_goods', 'goods_id', goods_id, rebuild.get_current_guid())
    rebuild.commit()

################################################################################
## 修改订单 ID, 这边我们规定出售订单 ID 和购买订单 ID 相等
################################################################################
# 获得订单的游标
orders_sql = '''
    SELECT sell_order_id FROM sell_order_1
'''
sell_order_iter = rebuild.execute_select_sql([orders_sql])
# 根据出售订单修改 出售订单(sell_order_1)、购买订单(buy_order_1)、订单商品(order_goods)的出售订单 ID
for sell_order_1 in sell_order_iter:
    for (sell_order_id, ) in sell_order_1:
        rebuild.update_table_id('sell_order_1', 'sell_order_id', sell_order_id)
        rebuild.update_table_id('buy_order_1', 'buy_order_id', sell_order_id, rebuild.get_current_guid())
        rebuild.update_table_id('order_goods', 'sell_order_id', sell_order_id, rebuild.get_current_guid())
```

```
    rebuild.commit()

  ##############################################################################
  ## 修改订单商品表 ID
  ##############################################################################
  # 获得订单商品的游标
  order_goods_sql = '''
    SELECT order_goods_id FROM order_goods
  '''
  order_goods_iter = rebuild.execute_select_sql([order_goods_sql])
  for order_goods in order_goods_iter:
    for (order_goods_id, ) in order_goods:
      rebuild.update_table_id('order_goods', 'order_goods_id', order_goods_id)
    rebuild.commit()
  # 关闭游标
  rebuild.close_cursor('select')
  rebuild.close_cursor('dml')
  # 关闭连接
  rebuild.close_conn()
```

完整的 python 程序：rebuild_id.py

3.5. 执行程序

```
python rebuild_id.py
```

3.6. 最后查看表的结果

```
SELECT * FROM goods LIMIT 0, 1;
+--------------------+------------+---------+----------+
| goods_id           | goods_name | price   | store_id |
+--------------------+------------+---------+----------+
| 3791337987775664129 | goods1     | 9369.00 |        1 |
+--------------------+------------+---------+----------+
SELECT * FROM sell_order_1 LIMIT 0, 1;
+--------------------+---------------+---------+---------+--------+
| sell_order_id      | user_guide_id | user_id | price   | status |
+--------------------+---------------+---------+---------+--------+
| 3791337998693437441 |             1 |      10 | 5320.00 |      1 |
+--------------------+---------------+---------+---------+--------+
SELECT * FROM buy_order_1 LIMIT 0, 1;
+--------------------+---------+---------------+
| buy_order_id       | user_id | user_guide_id |
+--------------------+---------+---------------+
| 3791337998693437441 |      10 |             1 |
+--------------------+---------+---------------+
SELECT * FROM order_goods LIMIT 0, 1;
+-------------------+-------------------+-------------------+--------------+---------+------+
| order_goods_id    | sell_order_id     | goods_id          | user_guide_id | price   | num  |
+-------------------+-------------------+-------------------+--------------+---------+------+
| 3792076554839789569 | 3792076377064214529 | 3792076372429508609 |            1 | 9744.00 |    2 |
```

```
+--------------------+--------------------+--------------------+--------------+---------+------+
```

建议：如果在生产上有使用到 snowflake 请务必要弄一个高可用防止单点故障，具体策略看你们自己定啦。

4. MySQL 分库分表单库分表和迁移数据

4.1. 前奏

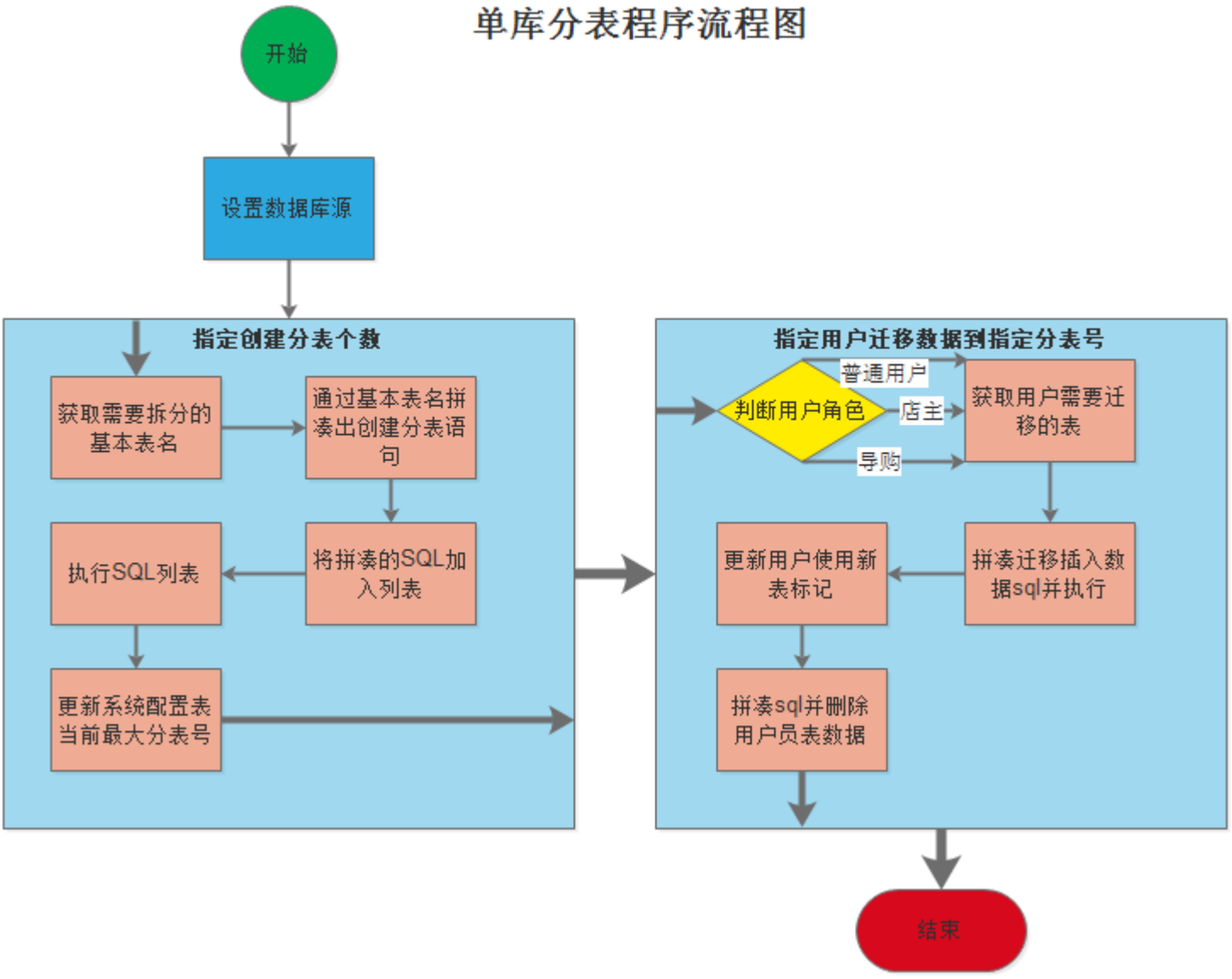
因为在分表的时候我们需要知道我们分的是第几个表，所以我们先需要初始化我们的分表号

```
-- 创建一个系统信息表为了记录下当前最大的分表号
DROP TABLE system_setting;
CREATE TABLE system_setting(
  system_setting_id INT unsigned NOT NULL AUTO_INCREMENT COMMENT '系统设置表 ID',
  name VARCHAR(45) NOT NULL COMMENT '系统设置项目名',
  value VARCHAR(45) NOT NULL COMMENT '系统设置值',
  PRIMARY KEY(system_setting_id)
);

-- 初始化当前最大分表号
INSERT INTO system_setting VALUES(NULL, 'max_sharding_table_num', 1);

-- 指定需要有哪些表需要分，为了下面分表时进行锁表
INSERT INTO system_setting VALUES(NULL, 'sharding_table', 'sell_order');
INSERT INTO system_setting VALUES(NULL, 'sharding_table', 'buy_order');
INSERT INTO system_setting VALUES(NULL, 'sharding_table', 'goods');
INSERT INTO system_setting VALUES(NULL, 'sharding_table', 'order_goods');

-- 需要分表的表是通过什么字段来分表的
INSERT INTO system_setting VALUES(NULL, 'sharding_sell_order_by', 'user_guide_id');
INSERT INTO system_setting VALUES(NULL, 'sharding_buy_order_by', 'user_id');
INSERT INTO system_setting VALUES(NULL, 'sharding_goods_by', 'store_id');
INSERT INTO system_setting VALUES(NULL, 'sharding_order_goods_by', 'user_guide_id');

-- 普通用户需要分那张表
INSERT INTO system_setting VALUES(NULL, 'normal_user_sharding', 'buy_order');
-- 导购需要分的表
INSERT INTO system_setting VALUES(NULL, 'user_guide_sharding', 'buy_order');
INSERT INTO system_setting VALUES(NULL, 'user_guide_sharding', 'sell_order');
INSERT INTO system_setting VALUES(NULL, 'user_guide_sharding', 'order_goods');
-- 店主需要分哪些表
INSERT INTO system_setting VALUES(NULL, 'store_owner_sharding', 'buy_order');
INSERT INTO system_setting VALUES(NULL, 'store_owner_sharding', 'sell_order');
INSERT INTO system_setting VALUES(NULL, 'store_owner_sharding', 'order_goods');
INSERT INTO system_setting VALUES(NULL, 'store_owner_sharding', 'goods');
```

4.2. 我们的目标

1、我们的目标是实现能手动指定创建多少张表，并且能手动指定哪些用户到哪些表中。
2、最终能通过一定的算法，自动化这些分表迁移数据的工作。
在这里我们来实现 '1' 手动指定，以下可能比较枯燥都是代码了，一般只要看主逻辑有一个思路，代码自己玩转它 ^_^

4.3. 程序流程图



4.4. 代码主逻辑展示

在附件中有完整的代码

```
if __name__=='__main__':
  # 设置默认的数据库链接参数
  db_config = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }

  sharding = ShardingTable()
  # 设置数据库配置
  sharding.set_db_config(db_config)
  # 初始化游标
  sharding.get_conn_cursor()
  # 提供需要分表的个数，创建分表
```

```
  sharding.create_tables(9)
  # 指定用户迁移数据到指定表
  sharding.move_data('username1', 2)
  sharding.move_data('username6', 6)
  sharding.move_data('username66', 9)
```

上面我们指定新分 9 个表，并且迁移 'username1-店主'、'username6-导购'、'username66-普通用户' 的数据到指定分表

完整的 python 代码：sharding_table.py 、 MySQL 分库分表(4)-脚本

4.5. 执行程序

```
python rebuild_id.py
```

4.6. 查看迁移后的数据

```
-- 查看迁移后的数据-购买订单表
SELECT * FROM buy_order_2 LIMIT 0, 1;
SELECT * FROM buy_order_6 LIMIT 0, 1;
SELECT * FROM buy_order_9 LIMIT 0, 1;
-- 查看迁移后的数据-商品表
SELECT * FROM goods_2 LIMIT 0, 1;
SELECT * FROM goods_6 LIMIT 0, 1;
SELECT * FROM goods_9 LIMIT 0, 1;
-- 查看迁移后的数据-出售订单表
SELECT * FROM sell_order_2 LIMIT 0, 1;
SELECT * FROM sell_order_6 LIMIT 0, 1;
SELECT * FROM sell_order_9 LIMIT 0, 1;
-- 查看迁移后的数据-订单商品表
SELECT * FROM order_goods_2 LIMIT 0, 1;
SELECT * FROM order_goods_6 LIMIT 0, 1;
SELECT * FROM order_goods_9 LIMIT 0, 1;
-- 查看用户数据分布在哪个分表 table_flag
SELECT * FROM user WHERE user_id IN(1, 6, 66);
```

5. MySQL 分库分表分表后数据的查询

5.1. 前言

在分表完之后显然对于数据的查询会变的比较的复杂，特别是在表的关联方面，在有些情况下根本就不能使用 JOIN。

其实个人是比较鼓励将那些大的 JOIN SQL 拆分成几个小的 SQL 来查询数据。这样虽然总体的效率可能会稍稍下降（如果使用了连接池完全可以忽略），但是查询的语句变简单了，使得后续的维护带来的方便。同时也能带来比较便利的扩展。你可以感受一下有一个 100 行的 SQL 语句给你维护，和给你 10 个 10 行并且每一块都有很好的注释的 SQL 去维护，去帮助调优。你愿意选哪个。不管你们信不信，反正我是选第二种，而且第二种可以很好的理解业务。

上面说到要拆分 JOIN，我的意思不是将每个语句都拆分。我的准则是 O(n) 次的查询。忌讳那种查出数据后通过程序循环查出结果再去数据库中查询，也就是需要 O(n*M)这种。 瞬间感觉方法论很重要有木有 ^_^。

5.2. 模拟场景

- **场景 1：购买者下订单**

1、在浏览商品的时候能获得商品的 门店 ID 和 商品 ID，至于导购 ID 这里我们能以随机的形式得到（需要根据业务来确定如何获取导购 ID）

2、通过导购 ID 获得导购的用户信息从而得到导购的数据应该放在那张分表。

3、将下单数据存入出售者的分表，和购买者的分表。

下面展示的是伪代码(因为只用 SQL 不好展示具体业务逻辑)，其实是自己比较懒不想写 Python 了。^_^

```
-- 获得导购分表信息，和所在门店
SELECT u.table_flag AS guide_flag,
  ug.store_id AS store_id
FROM user AS u, user_guide AS ug
WHERE u.user_id = ug.user_id
  AND user_guide_id = 导购 ID;

SET autocommit=0;
START TRANSACTION;
-- 创建销售订单 sell_order_2 通过程序拼凑出来的
INSERT INTO sell_order_2
VALUES(order_SnowflakeID, 导购 ID, 购买者 ID, 订单总额, 订单状态);
-- 记录此订单有哪些商品
INSERT INTO order_goods_2
VALUES(order_goods_SnowflakeID, order_SnowflakeID, 商品 ID, 商品价格, 商品个数);
-- 记录购买订单表  buy_order_6 购买者所在的分表，上面的是出售者所在的分表别弄混了
-- 购买者订单 ID 和 出售者订单 ID 是一样的
INSERT INTO buy_order_6
VALUES(order_SnowflakeID, 用户 ID, 导购 ID)

COMMIT;
SET autocommit=1;
```

- **情况 2：购买者浏览订单**

浏览购买者订单就是比较麻烦的，因为购买者订单信息和商品信息不是在同一分表中。

1、分页查找出购买者的订单列表。

2、将订单信息返回给浏览器后，使用 ajax 获取每个订单的商品。

```
-- 获得用户的分表信息 user_id = 66
SELECT table_flag FROM user WHERE user_id=66;
```

```
+------------+
| table_flag |
+------------+
|          9 |
+------------+
-- 获取用户订单，这些信息值直接先返回给浏览器的
SELECT * FROM buy_order_9 WHERE user_id=66 LIMIT 0, 1;
+--------------------+---------+---------------+
| buy_order_id       | user_id | user_guide_id |
+--------------------+---------+---------------+
| 3792111966815784961 |      66 |             1 |
+--------------------+---------+---------------+
-- 获取 user_guide_id=1 用户的分表信息
SELECT u.table_flag AS guide_flag
FROM user AS u, user_guide AS ug
WHERE u.user_id = ug.user_id
  AND user_guide_id = 1;
+------------+
| guide_flag |
+------------+
|          2 |
+------------+
-- 浏览器通过 ajax 获取商品信息进行展现
SELECT *
FROM order_goods_2
WHERE sell_order_id = 3792111966815784961
  AND user_guide_id = 1;
+--------------------+--------------------+--------------------+---------------+---------+------+
| order_goods_id     | sell_order_id      | goods_id           | user_guide_id | price   | num  |
+--------------------+--------------------+--------------------+---------------+---------+------+
| 3792112143781859329 | 3792111966815784961 | 3792111950445416449 |             1 | 3100.00 |    2 |
| 3792112160789762049 | 3792111966815784961 | 3792111951305248769 |             1 | 5810.00 |    1 |
+--------------------+--------------------+--------------------+---------------+---------+------+
```

从上面的试验我们可以看到原本在 '分库分表(1)--基础表介绍' 中的关联查询就能获得出订单的数据现在需要被拆为多个部分来查询(是不可避免的，这样做也未必不是好事)。

这里说一下我们为什么要使用 ajax 来获取并展现 '订单商品' 的数据：

1、我们不知道 '购买订单' 的导购的分表是哪一个，因此我们需要便利查询出的每一条 '购买订单'，如果有 10 个订单就需要便利 10 次去获取对应导购是哪个分表。

2、获得分表完之后还需要通过每个分表去关联 '订单商品' 获得商品信息。

3、获得到以上信息或需要整合成一个列表返回给浏览器。

通过上面一次性把说有数据返回给浏览器的方法，会影响到用户体验，让用户觉得很慢的感觉。并且需要写复杂的逻辑，难以维护。

我们将查询时间放大，一个查是 1s 如果有 10 个订单 一次性完成就可能需要 11s 以上的时间才返回给浏览器。如果先将查询的订单返回给浏览器。看上去就只需要 1s 就吧数据返回给浏览器了。

- **情况 3：导购查看订单**

导购也是一个普通用户，因此一登陆系统就知道 导购 ID 和 用户 ID

```
-- 获得导购的分表信息 user_id = 6, user_guide_id = 5
SELECT table_flag FROM user WHERE user_id=6;
+------------+
| table_flag |
+------------+
```

```
|           6 |
+-------------+
-- 查询订单信息
SELECT * FROM sell_order_6 WHERE user_guide_id = 5 LIMIT 0, 3;
+--------------------+---------------+---------+---------+--------+
| sell_order_id      | user_guide_id | user_id | price   | status |
+--------------------+---------------+---------+---------+--------+
| 3792112033412943873 |             5 |      10 | 5197.00 |      1 |
| 3792112033429721089 |             5 |      10 | 6826.00 |      1 |
| 3792112033446498305 |             5 |      10 | 5765.00 |      1 |
+--------------------+---------------+---------+---------+--------+
-- 查询订单商品信息
SELECT * FROM order_goods_6
WHERE sell_order_id IN(
  3792112033412943873,
  3792112033429721089,
  3792112033446498305
);
+--------------------+--------------------+--------------------+---------------+---------+------+
| order_goods_id     | sell_order_id      | goods_id           | user_guide_id | price   | num  |
+--------------------+--------------------+--------------------+---------------+---------+------+
| 3792112273532653569 | 3792112033412943873 | 3792111951800176641 |             5 | 7826.00 |    1 |
| 3792112292964864001 | 3792112033412943873 | 3792111952559345665 |             5 | 3057.00 |    2 |
| 3792112273545236481 | 3792112033429721089 | 3792111952660008961 |             5 | 8540.00 |    1 |
| 3792112292981641217 | 3792112033429721089 | 3792111951863091201 |             5 | 8545.00 |    1 |
| 3792112273566208001 | 3792112033446498305 | 3792111952110555137 |             5 | 8383.00 |    2 |
| 3792112292998418433 | 3792112033446498305 | 3792111952966193153 |             5 | 3282.00 |    2 |
+--------------------+--------------------+--------------------+---------------+---------+------+
```

- **情况 4：导购修改订单**

```
-- 修改订单价格
UPDATE sell_order_6 SET price = 1000.00 WHERE sell_order_id = 3792112033412943873;
```

- **情况 5：店主为店铺添加商品**

  添加商品只有店铺的店主有权限。然而店主也是一个普通用户。

```
-- 获得店主的分表信息 user_id = 1
SELECT table_flag FROM user WHERE user_id=1;
+------------+
| table_flag |
+------------+
|          2 |
+------------+
-- 店主添加商品
INSERT INTO goods_2 VALUES(SnowflakeID, 商品名称, 商品价格, 门店 ID);
```

6. MySQL 分库分表分库准备

6.1. 前言

随着业务的发展单库中的分表的数量越来越多，使用在单库上存放过多的表这样是不合理的。因此，我们就需要考虑将数据根据数据库进行拆分。

一般 mysql 不建议表的数量超过 1000 个。当然，这不能一概而论，还需要根据你的数据量，和硬件来确定然后根据自己的服务器调整几个 mysql '%open%' 参数，从而来确定你的库应该不超过几张表性能能在可接受范围内。

6.2. 分库思路

在分库前我们需要确定一下我们应该如何去分库：

1、我们是根据用户 ID 来进行分库，和分表的思路一样。

2、我们需要在用户表中标记一下用户的数据是在哪个库。

3、在系统设置表中应该记录下当前最大分库数量。

4、在系统设置表中应该记录现在所有分库的库名。

5、在系统设置表中应该记录每个分库的数据库连接描述符信息。

6.3. 分库规则

我们以 '数字' 为分库标识最终分库的名称如：test_1、test_2、test_3 ...

在新增加库的时候，我们在新库中创建的表的数量是在系统设置表中的最大分表数。如在系统设置表中 name='max_sharding_table_num' 的 value='10'，这时我们会初始化每个分表的个数为 10 个。

6.4. 数据迁移

和分表一样我们应该很清楚哪些表是需要进行分库，我们需要分库的表有 buy_order_n、goods_n、sell_order_n、order_goods_n。

我们应该将之前的数据的库名进行统一。如之前 test 库的数据要先迁移到 test_1 上

提醒：数据迁移慎重，不是说迁移就迁移的。其实也可以不用迁移的，如果不迁移之后的自动分库的代码就需要做多一点的判断。这为了统一我就做了迁移。

6.5. 数据迁移 SQL

```
-- 创建新库
CREATE DATABASE test_1;
use test;
-- 拼出需要创建的表
SELECT CONCAT('CREATE TABLE test_1.',
  TABLE_NAME,
  ' LIKE ',
  TABLE_SCHEMA, '.', TABLE_NAME, ';'
)
FROM information_schema.tables
WHERE TABLE_SCHEMA = 'test';
```

```
-- 创建表这边我们不迁移公用的表:user、store、user_guide、system_setting
CREATE TABLE test_1.buy_order_1 LIKE test.buy_order_1;
CREATE TABLE test_1.buy_order_10 LIKE test.buy_order_10;
CREATE TABLE test_1.buy_order_2 LIKE test.buy_order_2;
CREATE TABLE test_1.buy_order_3 LIKE test.buy_order_3;
CREATE TABLE test_1.buy_order_4 LIKE test.buy_order_4;
CREATE TABLE test_1.buy_order_5 LIKE test.buy_order_5;
CREATE TABLE test_1.buy_order_6 LIKE test.buy_order_6;
CREATE TABLE test_1.buy_order_7 LIKE test.buy_order_7;
CREATE TABLE test_1.buy_order_8 LIKE test.buy_order_8;
CREATE TABLE test_1.buy_order_9 LIKE test.buy_order_9;
CREATE TABLE test_1.goods_1 LIKE test.goods_1;
CREATE TABLE test_1.goods_10 LIKE test.goods_10;
CREATE TABLE test_1.goods_2 LIKE test.goods_2;
CREATE TABLE test_1.goods_3 LIKE test.goods_3;
CREATE TABLE test_1.goods_4 LIKE test.goods_4;
CREATE TABLE test_1.goods_5 LIKE test.goods_5;
CREATE TABLE test_1.goods_6 LIKE test.goods_6;
CREATE TABLE test_1.goods_7 LIKE test.goods_7;
CREATE TABLE test_1.goods_8 LIKE test.goods_8;
CREATE TABLE test_1.goods_9 LIKE test.goods_9;
CREATE TABLE test_1.order_goods_1 LIKE test.order_goods_1;
CREATE TABLE test_1.order_goods_10 LIKE test.order_goods_10;
CREATE TABLE test_1.order_goods_2 LIKE test.order_goods_2;
CREATE TABLE test_1.order_goods_3 LIKE test.order_goods_3;
CREATE TABLE test_1.order_goods_4 LIKE test.order_goods_4;
CREATE TABLE test_1.order_goods_5 LIKE test.order_goods_5;
CREATE TABLE test_1.order_goods_6 LIKE test.order_goods_6;
CREATE TABLE test_1.order_goods_7 LIKE test.order_goods_7;
CREATE TABLE test_1.order_goods_8 LIKE test.order_goods_8;
CREATE TABLE test_1.order_goods_9 LIKE test.order_goods_9;
CREATE TABLE test_1.sell_order_1 LIKE test.sell_order_1;
CREATE TABLE test_1.sell_order_10 LIKE test.sell_order_10;
CREATE TABLE test_1.sell_order_2 LIKE test.sell_order_2;
CREATE TABLE test_1.sell_order_3 LIKE test.sell_order_3;
CREATE TABLE test_1.sell_order_4 LIKE test.sell_order_4;
CREATE TABLE test_1.sell_order_5 LIKE test.sell_order_5;
CREATE TABLE test_1.sell_order_6 LIKE test.sell_order_6;
CREATE TABLE test_1.sell_order_7 LIKE test.sell_order_7;
CREATE TABLE test_1.sell_order_8 LIKE test.sell_order_8;
CREATE TABLE test_1.sell_order_9 LIKE test.sell_order_9;
-- 生成插入表的数据
SELECT CONCAT('INSERT INTO ',
  TABLE_SCHEMA, '.', TABLE_NAME,
  ' SELECT * FROM test', '.', TABLE_NAME, ';'
)
FROM information_schema.tables
WHERE TABLE_SCHEMA = 'test_1';
-- 插入数据
INSERT INTO test_1.buy_order_1 SELECT * FROM test.buy_order_1;
INSERT INTO test_1.buy_order_10 SELECT * FROM test.buy_order_10;
INSERT INTO test_1.buy_order_2 SELECT * FROM test.buy_order_2;
```

```
INSERT INTO test_1.buy_order_3 SELECT * FROM test.buy_order_3;
INSERT INTO test_1.buy_order_4 SELECT * FROM test.buy_order_4;
INSERT INTO test_1.buy_order_5 SELECT * FROM test.buy_order_5;
INSERT INTO test_1.buy_order_6 SELECT * FROM test.buy_order_6;
INSERT INTO test_1.buy_order_7 SELECT * FROM test.buy_order_7;
INSERT INTO test_1.buy_order_8 SELECT * FROM test.buy_order_8;
INSERT INTO test_1.buy_order_9 SELECT * FROM test.buy_order_9;
INSERT INTO test_1.goods_1 SELECT * FROM test.goods_1;
INSERT INTO test_1.goods_10 SELECT * FROM test.goods_10;
INSERT INTO test_1.goods_2 SELECT * FROM test.goods_2;
INSERT INTO test_1.goods_3 SELECT * FROM test.goods_3;
INSERT INTO test_1.goods_4 SELECT * FROM test.goods_4;
INSERT INTO test_1.goods_5 SELECT * FROM test.goods_5;
INSERT INTO test_1.goods_6 SELECT * FROM test.goods_6;
INSERT INTO test_1.goods_7 SELECT * FROM test.goods_7;
INSERT INTO test_1.goods_8 SELECT * FROM test.goods_8;
INSERT INTO test_1.goods_9 SELECT * FROM test.goods_9;
INSERT INTO test_1.order_goods_1 SELECT * FROM test.order_goods_1;
INSERT INTO test_1.order_goods_10 SELECT * FROM test.order_goods_10;
INSERT INTO test_1.order_goods_2 SELECT * FROM test.order_goods_2;
INSERT INTO test_1.order_goods_3 SELECT * FROM test.order_goods_3;
INSERT INTO test_1.order_goods_4 SELECT * FROM test.order_goods_4;
INSERT INTO test_1.order_goods_5 SELECT * FROM test.order_goods_5;
INSERT INTO test_1.order_goods_6 SELECT * FROM test.order_goods_6;
INSERT INTO test_1.order_goods_7 SELECT * FROM test.order_goods_7;
INSERT INTO test_1.order_goods_8 SELECT * FROM test.order_goods_8;
INSERT INTO test_1.order_goods_9 SELECT * FROM test.order_goods_9;
INSERT INTO test_1.sell_order_1 SELECT * FROM test.sell_order_1;
INSERT INTO test_1.sell_order_10 SELECT * FROM test.sell_order_10;
INSERT INTO test_1.sell_order_2 SELECT * FROM test.sell_order_2;
INSERT INTO test_1.sell_order_3 SELECT * FROM test.sell_order_3;
INSERT INTO test_1.sell_order_4 SELECT * FROM test.sell_order_4;
INSERT INTO test_1.sell_order_5 SELECT * FROM test.sell_order_5;
INSERT INTO test_1.sell_order_6 SELECT * FROM test.sell_order_6;
INSERT INTO test_1.sell_order_7 SELECT * FROM test.sell_order_7;
INSERT INTO test_1.sell_order_8 SELECT * FROM test.sell_order_8;
INSERT INTO test_1.sell_order_9 SELECT * FROM test.sell_order_9;

-- 向系统表中添加当前最大分库数量
INSERT INTO test.system_setting
VALUES(NULL, 'max_sharding_database_num', 1);
-- 向系统表中添加分库名前缀
INSERT INTO test.system_setting
VALUES(NULL, 'sharding_database_prefix', 'test');
-- 向系统表中添加当前有哪些分库
INSERT INTO test.system_setting
VALUES(NULL, 'sharding_database', 'test_1');
-- 修改系统表字段类 value 型为 varchar(120)
ALTER TABLE test.system_setting
MODIFY `value` varchar(120) NOT NULL COMMENT '系统设置值';
-- 向系统表添加响应数据库链接描述符
INSERT INTO test.system_setting
```

```
VALUES(NULL, 'test_1', '{"user":"root","password":"root","host":"127.0.0.1","port":3306,"database":"test_1"}');

-- 初始化用户所在库为 test_1
ALTER TABLE user
ADD db_name VARCHAR(45) NOT NULL DEFAULT 'test_1'
COMMENT '用户数据所在数据库名';
```

7. MySQL 分库分表 python 实现分库

7.1. 理清思路

现在我们分为两大个步骤：

1、创建分库，并在分库中创建分表。

2、能指定用户的数据到特定的库和表。

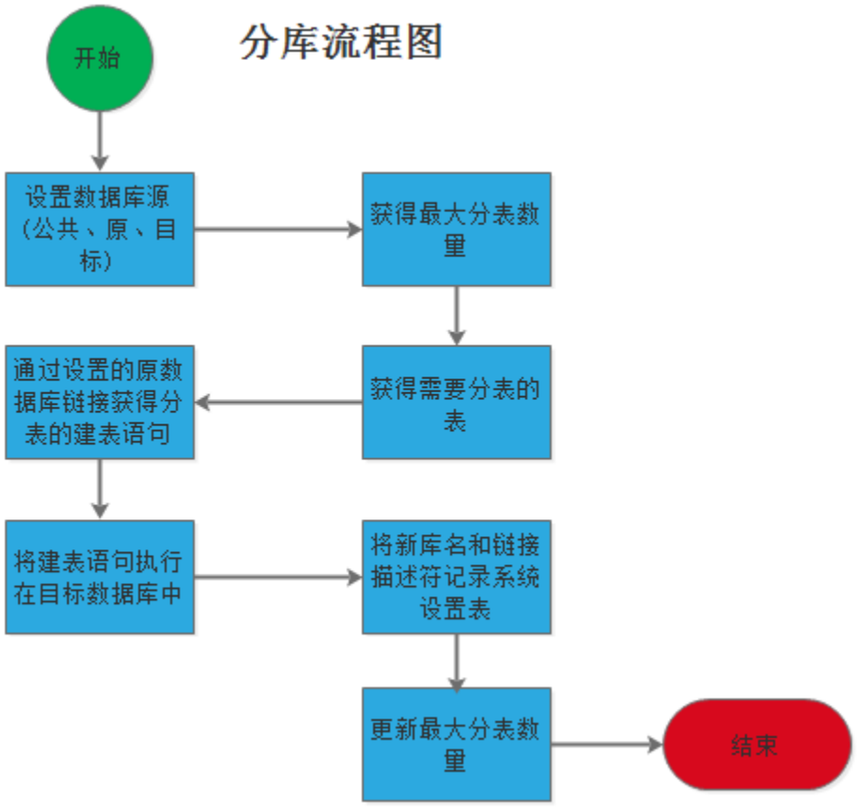
现在我们有两个数据库了：

1、test 库：里面存放了公共访问的数据表，因此在 python 我们需要有一个公共数据源。

2、test_1 分库：里面存放的是需要分表的表和数据，因此我们需要一个用户原数据所在的数据源。

3、test_n 分库：此库是用户的数据需要迁移到其他库的库，因此我们需要一个数据迁移的目录库数据源。

7.2. 分库流程图



7.3. 主执行过程

```
if __name__=='__main__':
  # 设置默认的数据库链接参数
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
```

```
# 配置用户数据所在数据库源
db_config_from = {
  'user'    : 'root',
  'password': 'root',
  'host'    : '127.0.0.1',
  'port'    : 3306,
  'database': 'test_1'
}
# 配置用户数据迁移目标数据目录
db_config_from = {
  'user'    : 'root',
  'password': 'root',
  'host'    : '127.0.0.1',
  'port'    : 3306,
}

sharding = ShardingDatabase()
# 设置公共数据库配置
sharding.get_conn_cursor(db_config_common, 'common')
# 设置用户原数据数据库配置
sharding.get_conn_cursor(db_config_from, 'from')
# 设置用户目标数据库配置
sharding.get_conn_cursor(db_config_to, 'to')

# 创建分库
db_config_to.pop('database')
sharding.create_db(db_config_to)

# 向分库中创建分表
max_num = sharding.get_max_sharding_table_num()
sharding.create_tables(begin = 1, offset = max_num, force=True)
```

7.4. 执行分库程序

```
python sharding_database.py
python sharding_database.py
```

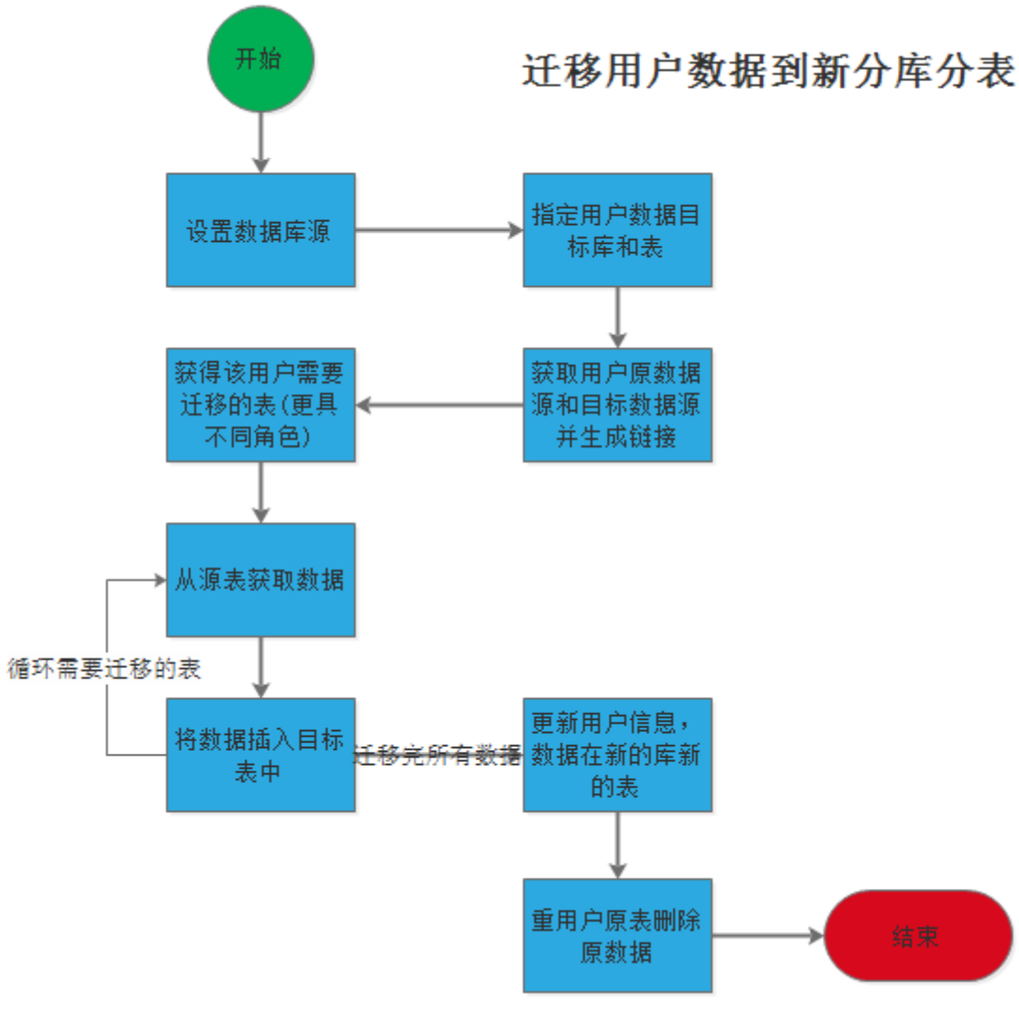
7.5. 执行后结果

```
SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| test               |
| test_1             |
| test_2             |
| test_3             |
+--------------------+
SELECT * FROM test.system_setting;
+-----------------+----------------------+-------------------------------------------------------------------------------------------------+
| system_setting_id | name                 | value                                                                                           |
```

```
+----------------+-------------------------+-----------------------------------------------------------------------------------------+
|             18 | max_sharding_database_num | 3                                                                                       |
|             19 | sharding_database         | test_1                                                                                  |
|             20 | test_1                    | {'user':'root','password':'root','host':'127.0.0.1','port':3306,'database':'test_1'}    |
|             21 | sharding_database_prefix  | test                                                                                    |
|             38 | harding_database          | test_2                                                                                  |
|             39 | test_2                    | {"port": 3306, "host": "127.0.0.1", "password": "root", "user": "root", "database": "test_2"} |
|             40 | harding_database          | test_3                                                                                  |
|             41 | test_3                    | {"port": 3306, "host": "127.0.0.1", "password": "root", "user": "root", "database": "test_3"} |
+----------------+-------------------------+-----------------------------------------------------------------------------------------+
USE test_2
SHOW TABLES;
+------------------+
| Tables_in_test_2 |
+------------------+
| buy_order_1      |
| buy_order_10     |
| buy_order_2      |
| buy_order_3      |
| buy_order_4      |
| buy_order_5      |
...
```

7.6. python 迁移用户数据到指定的分库分表

流程图：



7.7.主程序

```
if __name__=='__main__':
  # 设置公共库配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }

  sharding = ShardingDatabase()
  # 设置公共数据库配置
  sharding.get_conn_cursor(db_config_common, 'common')
  # 指定用户数据到 哪个库 哪个表，如：用户 username3 数据迁移到 test_3 库 10 号表
  sharding.move_data('username3', 'test_3', 10)
  sharding.move_data('username7', 'test_2', 3)
  sharding.move_data('username55', 'test_2', 6)
```

上面程序展示了将三位用户的数据迁移到指定的分库和分表中:

1、用户：username3 -> 库：test_3 -> 表：*_10

2、用户：username7 -> 库：test_2 -> 表：*_3

3、用户：username55 -> 库：test_2 -> 表：*_6

分库分表迁移数据 python 程序：sharding_database.py

7.8.迁移后结果展示

```
SELECT * FROM user;
+---------+-------------+-------------+------------+---------+
| user_id | username    | password    | table_flag | db_name |
+---------+-------------+-------------+------------+---------+
|       3 | username3   | password3   |         10 | test_3  |
|       7 | username7   | password7   |          3 | test_2  |
|      55 | username55  | password55  |          6 | test_2  |
...

USE test_3
SELECT * FROM sell_order_10 LIMIT 0, 1;
+--------------------+---------------+---------+---------+--------+
| sell_order_id      | user_guide_id | user_id | price   | status |
+--------------------+---------------+---------+---------+--------+
| 3792112071144902657 |             7 |      10 | 9720.00 |      1 |
+--------------------+---------------+---------+---------+--------+
SELECT * FROM buy_order_10 LIMIT 0, 1;
+--------------------+---------+---------------+
| buy_order_id       | user_id | user_guide_id |
+--------------------+---------+---------------+
| 3792111974680104961 |       3 |             1 |
+--------------------+---------+---------------+
SELECT * FROM goods_10 LIMIT 0, 1;
```

```
+--------------------+------------+--------+----------+
| goods_id           | goods_name | price  | store_id |
+--------------------+------------+--------+----------+
| 3792111953670836225 | goods1     | 370.00 |        3 |
+--------------------+------------+--------+----------+
SELECT * FROM order_goods_10 LIMIT 0, 1;
+--------------------+--------------------+--------------------+---------------+--------+------+
| order_goods_id     | sell_order_id      | goods_id           | user_guide_id | price  | num  |
+--------------------+--------------------+--------------------+---------------+--------+------+
| 3792112350317776897 | 3792112071144902657 | 3792111953670836225 |             7 | 370.00 |    1 |
+--------------------+--------------------+--------------------+---------------+--------+------+

USE test_2
SELECT * FROM sell_order_3 LIMIT 0, 1;
+--------------------+---------------+---------+---------+--------+
| sell_order_id      | user_guide_id | user_id | price   | status |
+--------------------+---------------+---------+---------+--------+
| 3792112052236980225 |             6 |      10 | 7790.00 |      1 |
+--------------------+---------------+---------+---------+--------+
SELECT * FROM buy_order_3 LIMIT 0, 1;
+--------------------+---------+---------------+
| buy_order_id       | user_id | user_guide_id |
+--------------------+---------+---------------+
| 3792111974399086593 |       7 |             1 |
+--------------------+---------+---------------+
SELECT * FROM order_goods_3 LIMIT 0, 1;
+--------------------+--------------------+--------------------+---------------+---------+------+
| order_goods_id     | sell_order_id      | goods_id           | user_guide_id | price   | num  |
+--------------------+--------------------+--------------------+---------------+---------+------+
| 3792112312489349121 | 3792112052236980225 | 3792111952869724161 |             6 | 6368.00 |    2 |
+--------------------+--------------------+--------------------+---------------+---------+------+

USE test_2
SELECT * FROM buy_order_3 LIMIT 0, 1;
+--------------------+---------+---------------+
| buy_order_id       | user_id | user_guide_id |
+--------------------+---------+---------------+
| 3792111974399086593 |       7 |             1 |
+--------------------+---------+---------------+
```

8. MySQL 分库分表分库后的查询

8.1. 前言

这边我们以使用 python 程序要展示一下再分库分表后，我们需要如何对数据库进行操作。

8.2. python 操作数据库

我们这边还是沿用之前的那 5 中:

- 场景 1：购买者下订单

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 客户下订单
# Author : HH
# Date   : 2016-02-08

import sys
import mysql.connector
import json
import snowflake.client

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟用户：username77 购买 store2 中的 goods27 和 goods69 商品
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  # 设置 snowflake 链接默认参数
  snowflake_config = {
    'host': '192.168.137.11',
    'port': 30001
  }
  # 配置 snowflake
  snowflake.client.setup(**snowflake_config)
  # 获得公共数据库的链接和游标
  conn_common = mysql.connector.connect(**db_config_common)
  cursor_select_common = conn_common.cursor(buffered=True)
  cursor_dml_common = conn_common.cursor(buffered=True)
```

```
# 获得用户:username77 的基本信息
select_sql = '''
  SELECT u.user_id,
    u.table_flag,
    u.db_name,
    ss.value
  FROM user AS u
    LEFT JOIN system_setting AS ss ON u.db_name = ss.name
  WHERE username = 'username77'
'''
cursor_select_common.execute(select_sql)
buy_user_id, buy_table_flag, buy_db_name, buy_db_config_json \
  = cursor_select_common.fetchone()
# 获得购买者的链接和游标
conn_buy = mysql.connector.connect(**json.loads(buy_db_config_json))
cursor_select_buy = conn_buy.cursor(buffered=True)
cursor_dml_buy = conn_buy.cursor(buffered=True)
# 通过店铺名称获得导购以及导购所对应的用户所使用的数据库链接描述符
select_sql = '''
  SELECT s.user_id,
    ug.user_guide_id,
    u.table_flag,
    u.db_name,
    ss.value AS db_config_json
  FROM store AS s
    LEFT JOIN user AS u USING(user_id)
    LEFT JOIN user_guide AS ug USING(user_id)
    LEFT JOIN system_setting AS ss ON ss.name = u.db_name
  WHERE s.user_id = 2
'''
cursor_select_common.execute(select_sql)
sell_user_id, user_guide_id, sell_table_flag, sell_dbname, \
sell_db_config_json = cursor_select_common.fetchone()

# 获得出售者的数据库链接描述符以及游标
conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
cursor_select_sell = conn_sell.cursor(buffered=True)
cursor_dml_sell = conn_sell.cursor(buffered=True)
# 成订单 ID
order_id = snowflake.client.get_guid()
# 获得商品信息并生成商品订单信息。
select_goods_sql = '''
  SELECT goods_id,
    price
  FROM {goods_table}
  WHERE goods_id IN(3794292584748158977, 3794292585729626113)
'''.format(goods_table = 'goods_' + str(sell_table_flag))
cursor_select_sell.execute(select_goods_sql)
# 订单价格
order_price = 0
for goods_id, price in cursor_select_sell:
  order_price += price
```

```
    # 生成订单商品信息
    insert_order_goods_sql = '''
      INSERT INTO {table_name}
      VALUES({guid}, {order_id}, {goods_id}, {user_guide_id},
        {price}, 1)
    '''.format(table_name = 'order_goods_' + str(sell_table_flag),
               guid = snowflake.client.get_guid(),
               order_id = order_id,
               goods_id = goods_id,
               user_guide_id = user_guide_id,
               price = price)
    cursor_dml_sell.execute(insert_order_goods_sql)
  # 生成订单记录
  insert_order_sql = '''
    INSERT INTO {order_table}
    VALUES({order_id}, {user_guide_id}, {user_id},
      {price}, 0)
  '''.format(order_table = 'sell_order_' + str(sell_table_flag),
             order_id = order_id,
             user_guide_id = user_guide_id,
             user_id = buy_user_id,
             price = order_price)
  cursor_dml_sell.execute(insert_order_sql)
  # 生成购买者订单记录
  insert_order_sql = '''
    INSERT INTO {order_buy_table}
    VALUES({order_id}, {user_id}, {user_guide_id})
  '''.format(order_buy_table = 'buy_order_' + str(buy_table_flag),
             order_id = order_id,
             user_id = buy_user_id,
             user_guide_id = user_guide_id)
  cursor_dml_buy.execute(insert_order_sql)

  # 提交事物
  conn_buy.commit()
  conn_sell.commit()
  # 关闭有标链接
  cursor_select_common.close()
  cursor_select_buy.close()
  cursor_select_sell.close()
  cursor_dml_common.close()
  cursor_dml_buy.close()
  cursor_dml_sell.close()

  conn_common.close()
  conn_buy.close()
  conn_sell.close()
```

- 场景 2：购买者浏览订单

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 客户下订单
```

```
# Author : HH
# Date   : 2016-02-08

import sys
import mysql.connector
import json

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟用户：username34 订单查询分页为每页一笔订单
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  conn_common = mysql.connector.connect(**db_config_common)
  cursor_select_common = conn_common.cursor(buffered=True)
  # 获得用户:username34 的基本信息
  select_sql = '''
    SELECT u.user_id,
      u.table_flag,
      u.db_name,
      ss.value
    FROM user AS u
      LEFT JOIN system_setting AS ss ON u.db_name = ss.name
    WHERE username = 'username77'
  '''
  cursor_select_common.execute(select_sql)
  buy_user_id, buy_table_flag, buy_db_name, buy_db_config_json \
    = cursor_select_common.fetchone()
  # 获得购买者的链接和游标
  conn_buy = mysql.connector.connect(**json.loads(buy_db_config_json))
  cursor_select_buy = conn_buy.cursor(buffered=True)
  # 获得购买者的一笔订单, 直接在后台获取数据传到前台
  select_buy_order_sql = '''
    SELECT buy_order_id,
      user_id,
      user_guide_id
    FROM {buy_order_table}
    WHERE user_id = 34
    LIMIT 0, 1
  '''.format(buy_order_table = 'buy_order_' + str(buy_table_flag))
  cursor_select_buy.execute(select_buy_order_sql)
  buy_order_id, buy_user_id, user_guide_id = cursor_select_buy.fetchone()
  # 使用打印来模拟现实在前台
```

```
  print 'buy order info: ', buy_order_id, buy_user_id, user_guide_id

  # 通过 user_guide_id 获得出售者用户信息以及其数据所在的库和表(需要通过 ajax 来实现)
  sell_info_sql = '''
    SELECT u.user_id,
      ug.user_guide_id,
      u.table_flag,
      u.db_name,
      ss.value AS db_config_json
    FROM user_guide AS ug
      LEFT JOIN user AS u USING(user_id)
      LEFT JOIN system_setting AS ss ON ss.name = u.db_name
    WHERE ug.user_guide_id = {user_guide_id}
  '''.format(user_guide_id = user_guide_id)
  cursor_select_common.execute(sell_info_sql)
  sell_user_id, user_guide_id, sell_table_flag, sell_dbname, \
  sell_db_config_json = cursor_select_common.fetchone()

  # 获得出售者的数据库链接描述符以及游标(需要通过 ajax 来实现)
  conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
  cursor_select_sell = conn_sell.cursor(buffered=True)
  # 获得订单商品(需要通过 ajax 来实现)
  order_goods_sql = '''
    SELECT *
    FROM {order_goods_table}
    WHERE sell_order_id = {buy_order_id}
  '''.format(order_goods_table = 'order_goods_' + str(sell_table_flag),
           buy_order_id = buy_order_id)
  cursor_select_sell.execute(order_goods_sql)
  order_goods = cursor_select_sell.fetchall()
  #使用打印来模拟 ajax 获取数据显示在前台
  for order_good in order_goods:
    print 'order good info: ', order_good

  # 关闭有标链接
  cursor_select_common.close()
  cursor_select_buy.close()
  cursor_select_sell.close()

  conn_common.close()
  conn_buy.close()
  conn_sell.close()
```

- 情况 3：导购查看订单

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 导购下订单
# Author : HH
# Date   : 2016-02-09

import sys
import mysql.connector
```

```
import json

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟导购：6 查询订单的情况
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  conn_common = mysql.connector.connect(**db_config_common)
  cursor_select_common = conn_common.cursor(buffered=True)
  # 获得导购:6 的基本信息
  sell_info_sql = '''
    SELECT u.user_id,
      ug.user_guide_id,
      u.table_flag,
      u.db_name,
      ss.value AS db_config_json
    FROM user_guide AS ug
      LEFT JOIN user AS u USING(user_id)
      LEFT JOIN system_setting AS ss ON ss.name = u.db_name
    WHERE ug.user_guide_id = 6
  '''
  cursor_select_common.execute(sell_info_sql)
  sell_user_id, user_guide_id, sell_table_flag, sell_db_name, \
  sell_db_config_json = cursor_select_common.fetchone()
  # 获得出售者的链接和游标
  conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
  cursor_select_sell = conn_sell.cursor(buffered=True)
  # 获得者的一笔订单以及订单商品
  select_sell_order_sql = '''
    SELECT *
    FROM (
      SELECT sell_order_id
      FROM {sell_order_table}
      WHERE user_guide_id = {user_guide_id}
      LIMIT 0, 1
    ) AS tmp_order
      LEFT JOIN {sell_order_table} USING(sell_order_id)
      LEFT JOIN {order_goods_table} USING(sell_order_id)
  '''.format(sell_order_table = 'sell_order_' + str(sell_table_flag),
             user_guide_id = user_guide_id,
             order_goods_table = 'order_goods_' + str(sell_table_flag))
  cursor_select_sell.execute(select_sell_order_sql)
```

```
  # 使用打印来模拟现实在前台显示订单详情
  for sell_order in cursor_select_sell:
    print sell_order

  # 关闭有标链接
  cursor_select_common.close()
  cursor_select_sell.close()

  conn_common.close()
  conn_sell.close()
```

- 情况 4：导购修改订单

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 导购修改订单信息
# Author : HH
# Date   : 2016-02-13

import sys
import mysql.connector
import json

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟修改导购 ID：6，订单 id：3794292705695109121 的订单
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  conn_common = mysql.connector.connect(**db_config_common)
  cursor_select_common = conn_common.cursor(buffered=True)
  # 获得导购:6 的基本信息
  sell_info_sql = '''
    SELECT u.user_id,
      ug.user_guide_id,
      u.table_flag,
      u.db_name,
      ss.value AS db_config_json
    FROM user_guide AS ug
      LEFT JOIN user AS u USING(user_id)
      LEFT JOIN system_setting AS ss ON ss.name = u.db_name
    WHERE ug.user_guide_id = 6
  '''
  cursor_select_common.execute(sell_info_sql)
```

```
  sell_user_id, user_guide_id, sell_table_flag, sell_db_name, \
  sell_db_config_json = cursor_select_common.fetchone()
  # 获得出售者的链接和游标
  conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
  cursor_dml_sell = conn_sell.cursor(buffered=True)
  # 修改订单 3794292705695109121 的价格
  update_sell_order_sql = '''
    UPDATE sell_order_{table_flag}
    SET price = {price}
    WHERE sell_order_id = 3794292705695109121
  '''.format(table_flag = sell_table_flag,
             price = 5320.00)

  cursor_dml_sell.execute(update_sell_order_sql)
  conn_sell.commit()

  # 关闭有标链接
  cursor_select_common.close()
  cursor_dml_sell.close()

  conn_common.close()
  conn_sell.close()
```

- 情况 5：店主为店铺添加商品

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 店主添加商品
# Author : HH
# Date   : 2016-02-13

import sys
import mysql.connector
import json
import snowflake.client

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟修改导购 ID：7 为商店添加商品
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  # 设置 snowflake 链接默认参数
  snowflake_config = {
```

```
    'host': '192.168.137.11',
    'port': 30001
}
# 配置 snowflake
snowflake.client.setup(**snowflake_config)
# 获得公共数据库链接和游标
conn_common = mysql.connector.connect(**db_config_common)
cursor_select_common = conn_common.cursor(buffered=True)
# 获得导购:7 的基本信息
sell_info_sql = '''
  SELECT u.user_id,
    ug.user_guide_id,
    u.table_flag,
    u.db_name,
    ss.value AS db_config_json,
    ug.store_id AS store_id
  FROM user_guide AS ug
    LEFT JOIN user AS u USING(user_id)
    LEFT JOIN system_setting AS ss ON ss.name = u.db_name
  WHERE ug.user_guide_id = 7
'''
cursor_select_common.execute(sell_info_sql)
sell_user_id, user_guide_id, sell_table_flag, sell_db_name, \
sell_db_config_json, sell_store_id = cursor_select_common.fetchone()
# 获得出售者的链接和游标
conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
cursor_dml_sell = conn_sell.cursor(buffered=True)
# 修改订单 3794292705695109121 的价格
insert_goods_sql = '''
  INSERT INTO goods_{table_flag}
  VALUES({gid}, 'goods101', 5320.00, {store_id})
'''.format(gid = snowflake.client.get_guid(),
           table_flag = sell_table_flag,
           store_id = sell_store_id)

cursor_dml_sell.execute(insert_goods_sql)
conn_sell.commit()

# 关闭有标链接
cursor_select_common.close()
cursor_dml_sell.close()

conn_common.close()
conn_sell.close()
```

以上就是在分库完之后的一些操作。具体如何查询还是需要和业务相结合的，万事离不开业务嘛。

python 脚本下载：mysql 分库分表 8

9. MySQL 分库分表-多实例 INSERT 的困扰

9.1. 存在问题

分库分表是完成了，细心的朋友可能会发现。我们这边存在一个问题就是本来应该在一起的事务，现在因为分库事务将被分成了两个。如果第一个事务完成了提交，这时候应用程序或服务器发生了问题导致第二个没有提交。这样就有问题了。

例如：在之前文章的示例中导购的销售订单的事务已经提交了，而这时由于某些原因导致购买者的订单没有提交。这时候就会出现导购能看得到订单而消费者却看不到。

9.2. 解决办法

1、直接使用 MySQL 的 XA(分布式事物)。

2、从业务上去解决。

使用 MySQL 的 XA 前需要知道的

在同一个实例中不能有相同的 XID(分布式事务 ID)出现。使用 Python 在同一实例创建多个相同的 XA 会报错"The XID already exists"。

9.3. 使用 MySQL 的 XA

由于之前为了方便只在一个实例中演示了分库分表的操作。如果使用多个 XA 会报错的。

这边我只使(用伪代码)的形式来说明使用 XA。如果有想知道 Python 如何使用 XA 可以常考如下链接 http://stackoverflow.com/questions/18485895/python-distributed-transactions-like-java-transaction-api-jta。

```
# 1.获取买家和卖家数据库连接描述符
buy_user_conn_info = get_conn_info()
sell_user_conn_info = get_conn_info()

# 比较 得到的两个数据库连接描述符
if buy_user_conn_info == sell_user_conn_info:
  # 如果相等使用同只创建一个数据量连接进行数据库操作
  conn = mysql.connector.connect(**sell_user_conn_info)
  # 通过 pysnowflask 获得 XID, order_id。XID 和 order_id 是同一个
  guid = snowflake.client.get_guid()
  # 获得游标
  cur = conn.cursor();
  # 开始一个 XA 事务
  cur.execute("XA START '{xid}'".format(xid = guid));
  # 生成出售者订单信息
  cur.execute("INSERT INTO test_x.sell_order_x VALUES(xx)");
  cur.execute("INSERT INTO test_x.order_goods_x VALUES(xx)");
  cur.execute("INSERT INTO test_x.order_goods_x VALUES(xx)");
  # 生成购买者订单信息
  cur.execute("INSERT INTO test_y.buy_order_y VALUES(xx)");
  # 完成数据操作
  cur.execute("XA END '{xid}'".format(xid = guid));
  # 要准备提交了
  cur.execute("XA PREPARE '{xid}'".format(xid = guid));
  # 提交事务
  cur.execute("XA COMMIT '{xid}'".format(xid = guid));
```

```
else:
  # 如果连接描述符不相等就需要在不实例中使用 XID 相等的 XA 事务
  conn_sell = mysql.connector.connect(**sell_user_conn_info)
  conn_buy = mysql.connector.connect(**buy_user_conn_info)
  # 获得游标
  cur_sell = conn_sell.cursor();
  cur_buy = conn_buy.cursor();
  try:
    # 通过 pysnowflask 获得 XID, order_id。XID 和 order_id 是同一个
    cur_sell.execute("XA START '{xid}'".format(xid = guid));
    cur_buy.execute("XA START '{xid}'".format(xid = guid));
    # 生成出售者订单信息
    cur_sell.execute("INSERT INTO test_x.sell_order_x VALUES(xx)");
    cur_sell.execute("INSERT INTO test_x.order_goods_x VALUES(xx)");
    cur_sell.execute("INSERT INTO test_x.order_goods_x VALUES(xx)");
    # 生成购买者订单信息
    cur_buy.execute("INSERT INTO test_y.buy_order_y VALUES(xx)");
    # 要准备提交了
    cur_sell.execute("XA PREPARE '{xid}'".format(xid = guid));
    cur_buy.execute("XA PREPARE '{xid}'".format(xid = guid));
    # 提交事务
    cur_sell.execute("XA COMMIT '{xid}'".format(xid = guid));
    cur_buy.execute("XA COMMIT '{xid}'".format(xid = guid));
  except:
    # 发生错误回滚 XA 事务
    cur_sell.execute("XA ROLLBACK '{xid}'".format(xid = guid));
    cur_buy.execute("XA ROLLBACK '{xid}'".format(xid = guid));
  finally:
    cur_sell.close()
    cur_buy.close()
    conn_sell.disconnect()
    conn_buy.disconnect()
```

需要说明的是如果使用了 XA 事务在性能上肯定会比没有使用事务来的差。

- **MySQL XA 事务额外注意事项**

在使用 MySQL 5.6 版本之前(包括 5.6)使用分布式事务的时候如果还没有 COMMIT 的事务是不会记录到 binlog 中的。因此如果在 COMMIT 之前如果发生了 MySQL 挂掉，一定要将之前还没有提交的 XA 事务给回滚了。如果提交了由于没有些 binlog 导致从库就同步不到这次事务，从而导致了主从不一致的情况。而在 MySQL5.7 中解决了这一个问题。

9.4. 从业务上去解决

从业务上去解决说白了就是能用语言来掩饰会遇到的 BUG。然后，让那些做产品的能够接受，并且最后能有一些友好的补救措施。当然，这种方式不能影响主要业务。

- **问题分析**

我们现在担心的是。当用户下单的时候发生了只有出售者的用户订单是写入数据库的，而购买订单没有写入到数据库。从表现来看就是出售者能看到订单信息，而购买者就看不到。并且这样的事一般只会在应用程序或服务器出问题容易发生，平常都是能正常的。

认真想一下，购买者没看到订单，那就不能够付款。不能够付款，说明订单的状态始终是未付款状态。也就是成了一个僵尸单。买家没有经济上的损失，卖家也没有货物上的损失。那这种情况是不是双方都能接受的嘞？这就需要和产品沟通了。

当然，我们不能放着这些僵尸单不管的。我们需要定期的去监控这些僵尸单的存在。如果检测到了僵尸单应该做出相应的处理，这时候又要和产品沟通了，是恢复该僵尸单让购买者能看到并给相关提示呢？还是该僵尸单给干掉，或者其他更友好的办法。

- **对僵尸单的监控**

很容易想到可以通过查询数据库来过滤订单僵尸订单，这其实是一个办法。还有另外一个办法，就是在提交订单事物完成和会写相关的日志。我们只要去分析日志就好了。

- **这边我以日志的形式进行讲解：**

1、当卖家订单提交是将信息记入到文件（主要是记录订单 ID）。

2、当买家订单提交是将信息记录到另一个文件（主要是记录订单 ID）。

3、用程序分析两个文件取出订单是否产生差集。并记录下差集的订单 ID。这个差集的订单 ID 有僵尸单的嫌疑，

4、当再寻找僵尸单的时候先对上次可以的僵尸单进行处理，如果没有在购买者的日志中那说明那就是僵尸单了。然后再接着进行分析对比。

需要注意的是要把控好监控的时间间隔，不能太短，比如 1s。这时购买者订单可能还没提交。这样就会错误判断为僵尸单。

10. MySQL 分库分表-弃强妥最提高性能

10.1. 回顾

之前我们介绍了使用分布式事务（XA）处理用户下订单，对 MySQL 有所了解的都知道 XA 其实是在非用不可的情况下才用的，因为它实在是影响性能。当然，其实迫使我们使用 XA 的原因也是因为我们的设计决定的（其实使用 XA 这种分库分表的方案基本能满足那些一定需要强一致性的需求）。

之前我们的设计是为了方便卖家的，所以完整的订单信息只保存在卖家方，而在买家方我们只保存着完整订单的引用。这样做是因为业务需要强一致性，迫使我们使用 XA，但我们又希望让数据写磁盘少一点，让插入快一点。

10.2. 我们想要的

无论在买家还是卖家在操作订单数据的时候都能方便。为了满足这样的情况，我们只能做到数据冗余了。就是买家有一个完整的数据卖家也有一份完整的数据。这样操作起来就能在单机上进行，这样是最方便的。

10.3. 业务最终一致性

其实，往往再深入分析业务，可以发现其实业务上并非一定需要强一致性。我们的目的只要买家已经下的订单能完整让卖家看到，卖家多久能看到其实并不是很关心。因为如果卖家没看到订单也不会对订单进行操作，也不会发货给买家，这样卖家是不会有损失的。而买家就算是付了款，发现买家没发货，也可以退款。这样一来买家也没有金钱的损失。所以我们这边能使用最终一致性来解决问题。

10.4. 为什么要最终一致性

说白了就是为了提高性能，因为我们现在需要买家和卖家都有完整的订单数据，为了让买家下单时不用跨库、跨实例或跨节点执行 XA 事务。这样我们保证买家的订单数据先入库后能马上返回成功信息。而不用关心卖家的数据是否入库，只是提醒卖家有订单信息要入库了。

10.5. kafka 配合完成最终一致性

要完成最终一致性这种业务，我最先想到的就是使用消息队列了。而在消息队列中我最熟悉的只能是 kafka 了，我使用 kafka 的原因是他能高可用，还能将消息持久化。

下面我们就来演示使用 kafka 来完成最终一致性

- 重构买家订单表(buy_order_x)结构：

```
-- 在每个分库为 buy_order_x 添加 price 列
ALTER TABLE buy_order_1 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_2 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_3 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_4 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_5 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_6 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_7 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_8 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_9 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
ALTER TABLE buy_order_10 ADD `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格';
```

```
-- 在每个分库为 buy_order_x 添加 status 列
ALTER TABLE buy_order_1 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_2 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_3 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_4 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_5 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_6 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_7 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_8 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_9 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
ALTER TABLE buy_order_10 ADD `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态';
-- 最终表结构
CREATE TABLE `buy_order_1` (
  `buy_order_id` bigint(20) unsigned NOT NULL COMMENT '出售订单 ID 与出售订单相等',
  `user_id` int(10) unsigned DEFAULT NULL COMMENT '下单用户 ID',
  `user_guide_id` int(10) unsigned DEFAULT NULL COMMENT '导购 ID',
  `price` decimal(11,2) DEFAULT NULL COMMENT '订单价格',
  `status` tinyint(3) unsigned DEFAULT NULL COMMENT '订单状态',
  PRIMARY KEY (`buy_order_id`),
  KEY `idx$buy_order_1$user_id` (`user_id`),
  KEY `idx$buy_order_1user_guide_id` (`user_guide_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- 为买家创建订单商品表:

```
-- 在每一个库都进行添加多个 buy_order_goods_x 表
-- 下面表 buy_order_goods_x 中的 x 代表的是 1,2,3,4,5,6,7,8,9,10
CREATE TABLE `buy_order_goods_x` (
  `order_goods_id` bigint(20) unsigned NOT NULL COMMENT '订单商品表 ID',
  `user_id` int(10) unsigned DEFAULT NULL COMMENT '下单用户 ID',
  `sell_order_id` bigint(20) unsigned NOT NULL COMMENT '订单 ID',
  `goods_id` bigint(20) unsigned NOT NULL COMMENT '商品 ID',
  `user_guide_id` int(10) unsigned DEFAULT NULL COMMENT '导购 ID',
  `price` decimal(11,2) DEFAULT NULL COMMENT '商品购买价格价格',
  `num` tinyint(3) unsigned DEFAULT NULL COMMENT '商品数量',
  PRIMARY KEY (`order_goods_id`),
  KEY `idx$order_goods$orders_id` (`sell_order_id`),
  KEY `idx$order_goods$user_id` (`user_id`),
  KEY `idx$order_goods$goods_id` (`goods_id`),
  KEY `idx$order_goods$user_guide_id` (`user_guide_id`)
);
-- 将 buy_order_goods 加入系统配置中表明该表是分表。
INSERT INTO system_setting VALUES(NULL, 'sharding_table', 'buy_order_goods');
-- 设置 buy_order_goods 分表的关键字
INSERT INTO system_setting VALUES(NULL, 'sharding_buy_order_goods_by', 'user_id');
-- 指定每种角色需要访问的表
INSERT INTO system_setting VALUES(NULL, 'normal_user_sharding', 'buy_order_goods');
INSERT INTO system_setting VALUES(NULL, 'user_guide_sharding', 'buy_order_goods');
INSERT INTO system_setting VALUES(NULL, 'store_owner_sharding', 'buy_order_goods');
```

这边我不关心 kafka 是如何搭建的，就直接使用了。要想知道如何搭建 kafka 请点击一下网址：http://www.linuxidc.com/Linux/2014-07/104470.htm

- 买家下订单演示:

注意这边 python 代码使用的是 simplejson 因此需要安装。安装包：simplejson-master

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 客户下订单+kafka 订单推送
# Author : HH
# Date   : 2016-02-27

import sys
import mysql.connector
import time
import simplejson as json
import snowflake.client
from pykafka import KafkaClient

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  这边模拟用户：username77 购买 store2 中的 goods27 和 goods69 商品
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  # 设置 snowflake 链接默认参数
  snowflake_config = {
    'host': '192.168.137.11',
    'port': 30001
  }
  # 配置 snowflake
  snowflake.client.setup(**snowflake_config)
  # 配置连接 kafka
  client = KafkaClient(hosts="192.168.137.11:9092, \
                          192.168.137.11:9093, \
                          192.168.137.11:9094")
  topic = client.topics['shop']
  # 创建生产者
  producer = topic.get_producer(delivery_reports=False)

  # 获得公共数据库的链接和游标
  conn_common = mysql.connector.connect(**db_config_common)
  cursor_select_common = conn_common.cursor(buffered=True)
  # 获得用户:username77 的基本信息
  select_sql = '''
    SELECT u.user_id,
      u.table_flag,
      u.db_name,
```

```
    ss.value
  FROM user AS u
    LEFT JOIN system_setting AS ss ON u.db_name = ss.name
  WHERE username = 'username77'
'''
cursor_select_common.execute(select_sql)
buy_user_id, buy_table_flag, buy_db_name, buy_db_config_json \
  = cursor_select_common.fetchone()
# 获得购买者的链接和游标
conn_buy = mysql.connector.connect(**json.loads(buy_db_config_json))
cursor_select_buy = conn_buy.cursor(buffered=True)
cursor_dml_buy = conn_buy.cursor(buffered=True)
# 通过店铺名称获得导购以及导购所对应的用户所使用的数据库链接描述符
select_sql = '''
  SELECT s.user_id,
    ug.user_guide_id,
    u.table_flag,
    u.db_name,
    ss.value AS db_config_json
  FROM store AS s
    LEFT JOIN user AS u USING(user_id)
    LEFT JOIN user_guide AS ug USING(user_id)
    LEFT JOIN system_setting AS ss ON ss.name = u.db_name
  WHERE s.user_id = 2
'''
cursor_select_common.execute(select_sql)
sell_user_id, user_guide_id, sell_table_flag, sell_dbname, \
sell_db_config_json = cursor_select_common.fetchone()

# 获得出售者的数据库链接描述符以及游标
conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
cursor_select_sell = conn_sell.cursor(buffered=True)
# 成订单 ID
order_id = snowflake.client.get_guid()
# 获得商品信息并生成商品订单信息。
select_goods_sql = '''
  SELECT goods_id,
    price
  FROM {goods_table}
  WHERE goods_id IN(3794292584748158977, 3794292585729626113)
'''.format(goods_table = 'goods_' + str(sell_table_flag))
cursor_select_sell.execute(select_goods_sql)
# 订单价格
order_price = 0
for goods_id, price in cursor_select_sell:
  # 生成订单货物 ID
  guid = snowflake.client.get_guid()
  order_price += price
  # 生成订单商品信息
  insert_order_goods_sql = '''
    INSERT INTO {table_name}
    VALUES({guid}, {user_id}, {order_id}, {goods_id}, {user_guide_id},
```

```
      {price}, 1)
  '''.format(table_name = 'buy_order_goods_' + str(buy_table_flag),
             guid = guid,
             user_id = buy_user_id,
             order_id = order_id,
             goods_id = goods_id,
             user_guide_id = user_guide_id,
             price = price)
  cursor_dml_buy.execute(insert_order_goods_sql)
  # 将订单商品信息放入队列中
  goods_dict = {
    'user_id'    : sell_user_id,
    'option_type' : 'insert',
    'option_table': 'order_goods',
    'option_obj'  : {
      'order_goods_id': guid,
      'sell_order_id' : order_id,
      'goods_id'     : goods_id,
      'user_guide_id' : user_guide_id,
      'price'        : price,
      'num'          : 1
    }
  }
  goods_json = json.dumps(goods_dict)
  producer.produce(goods_json)
# 生成订单记录
insert_order_sql = '''
  INSERT INTO {order_table}
  VALUES({order_id}, {user_id}, {user_guide_id},
    {price}, 0)
'''.format(order_table = 'buy_order_' + str(buy_table_flag),
           order_id = order_id,
           user_id = buy_user_id,
           user_guide_id = user_guide_id,
           price = order_price)
cursor_dml_buy.execute(insert_order_sql)
# 将订单信息放入队列中
order_dict = {
  'user_id'    : sell_user_id,
  'option_type' : 'insert',
  'option_table': 'sell_order',
  'option_obj'  : {
    'sell_order_id' : order_id,
    'user_guide_id' : user_guide_id,
    'user_id'      : buy_user_id,
    'price'        : order_price,
    'status'       : 0
  }
}
order_json = json.dumps(order_dict)
producer.produce(order_json)
producer.stop()
```

```
  # 提交事物
  conn_buy.commit()
  # 关闭有标链接
  cursor_select_common.close()
  cursor_select_buy.close()
  cursor_select_sell.close()
  cursor_dml_buy.close()

  conn_common.close()
  conn_buy.close()
  conn_sell.close()
```

- 生成卖家订单信息:

通过上面买家下订单，现在订单的信息在队列中。我们只需要将队列中的数据取出并保存就好了。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# Program: 客户下订单+kafka 订单推送
# Author : HH
# Date   : 2016-02-27

import sys
import mysql.connector
import time
import simplejson as json
from pykafka import KafkaClient

reload(sys)
sys.setdefaultencoding('utf-8')

if __name__ == '__main__':
  '''
  获取队列数据保存到数据库
  '''
  # 设置公共库连接配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }
  # 配置 snowflake
  snowflake.client.setup(**snowflake_config)
  # 配置连接 kafka
  client = KafkaClient(hosts="192.168.137.11:9092, \
                          192.168.137.11:9093, \
                          192.168.137.11:9094")
  topic = client.topics['shop']
  # 生成消费者
  balanced_consumer = topic.get_balanced_consumer(
    consumer_group='goods_group',
    auto_commit_enable=True,
```

```
    zookeeper_connect='localhost:2181'
)
# 获得公共数据库的链接和游标
conn_common = mysql.connector.connect(**db_config_common)
cursor_select_common = conn_common.cursor(buffered=True)

# 消费信息
for message in balanced_consumer:
  if message is not None:
    # 解析 json 为 dict
    info_dict = json.loads(message.value)
    select_sql = '''
      SELECT s.user_id,
        ug.user_guide_id,
        u.table_flag,
        u.db_name,
        ss.value AS db_config_json
      FROM store AS s
        LEFT JOIN user AS u USING(user_id)
        LEFT JOIN user_guide AS ug USING(user_id)
        LEFT JOIN system_setting AS ss ON ss.name = u.db_name
      WHERE s.user_id = {user_id}
    '''.format(user_id=info_dict['user_id'])
    cursor_select_common.execute(select_sql)
    sell_user_id, user_guide_id, sell_table_flag, sell_dbname, \
    sell_db_config_json = cursor_select_common.fetchone()

    # 获得出售者的数据库链接描述符以及游标
    conn_sell = mysql.connector.connect(**json.loads(sell_db_config_json))
    cursor_dml_sell = conn_sell.cursor(buffered=True)
    if info_dict['option_type'] == 'insert':
      # 构造 insert sql 语句
      if info_dict['option_table'] == 'order_goods':
        insert_sql = '''
          INSERT INTO {table_name}_{flag} VALUES(
            {order_goods_id},
            {sell_order_id},
            {goods_id},
            {user_guide_id},
            {price},
            {num}
          )
        '''.format(table_name = info_dict['option_table'],
            flag = sell_table_flag,
            **info_dict['option_obj']
        )
      elif info_dict['option_table'] == 'sell_order':
        insert_sql = '''
          INSERT INTO {table_name}_{flag} VALUES(
            {sell_order_id},
            {user_guide_id},
            {user_id},
```

```
            {price},
            {status}
          )
        '''.format(table_name = info_dict['option_table'],
            flag = sell_table_flag,
            **info_dict['option_obj']
        )
      # 执行 sql
      cursor_dml_sell.execute(insert_sql)
      conn_sell.commit()
      cursor_dml_sell.close()
      conn_sell.close()
    elif info_dict['option_type'] == 'update':
      pass
```

源码：order_create_script

11. MySQL 分库分表-缩容

11.1. 此缩容非彼缩容

无论是在 ORACLE、MSSQL 中都会存在着扩容、缩容的操作，并且这个技能基本是 DBA 所必备的。下面是本人的一点理解：

- **扩容：**数据在增长，在快达到磁盘或数据库的容量时，增加磁盘和或表空间的一种操作。
- **缩容：**在 delete、update、insert 操作平凡的表中会产生许多的磁盘碎片，这是后需要对碎片进行整理或者对表数据进行从新生成一次，从而达到减小容量的目的。

而现在要说的扩容和缩容是结业某种业务场景的（其实就是分库分表的数据迁移）：

- **扩容：**在预计访问会爆增之前。增加机器，并分库分表将数据进行迁移，让压力进行分散处理。从而能度过高频反问时期。
- **缩容：**在高频访问时期过去了，再将数据进行汇集。以至于能腾出机器，从而达到减少成本的一种做法。

扩容其实在之前分库分表的时候从操作过了。只是之前我们不知道那样就叫做扩容。

11.2. 缩容

- 下面我们演示将 test_3 的库数据进行迁移

```
if __name__=='__main__':
  # 设置公共库配置
  db_config_common = {
    'user'    : 'root',
    'password': 'root',
    'host'    : '127.0.0.1',
    'port'    : 3306,
    'database': 'test'
  }

  sharding = ShardingDatabase()
  # 设置公共数据库配置
  sharding.get_conn_cursor(db_config_common, 'common')
  # 获得数据存在 test_3 的用户
  select_sql = '''
    SELECT username FROM user where db_name = 'test_3'
  '''
  sharding.cursor_select_common.execute(select_sql)
  username_list = []
  for (username,) in sharding.cursor_select_common:
    username_list.append(username)
  for username in username_list:
    # 指定用户数据到 test_2 库 表 7
    sharding.move_data(username, 'test_2', 7)

  # 删除库 test_3
  drop_db_sql = '''
    DROP DATABASE test_3
  '''
  sharding.cursor_dml_common.execute(drop_db_sql)
```

源码：reduce_capacity

- 查看迁移后数据：

```
-- 库 test_3 已经被删除
show databases;
+--------------------+
| Database           |
+--------------------+
| test               |
| test_1             |
| test_2             |
+--------------------+

-- 在 test_2 库的用户数据
SELECT * FROM test.user WHERE db_name = 'test_2';
+---------+------------+------------+------------+---------+
| user_id | username   | password   | table_flag | db_name |
+---------+------------+------------+------------+---------+
|       3 | username3  | password3  |          7 | test_2  |
|       7 | username7  | password7  |          3 | test_2  |
|      55 | username55 | password55 |          6 | test_2  |
+---------+------------+------------+------------+---------+
SELECT * FROM buy_order_7 WHERE user_id = 3 LIMIT 0, 2;
+--------------------+---------+---------------+-------+--------+
| buy_order_id       | user_id | user_guide_id | price | status |
+--------------------+---------+---------------+-------+--------+
| 3794292612787081217 |       3 |             1 |  0.00 |      0 |
| 3794292612803858433 |       3 |             1 |  0.00 |      0 |
+--------------------+---------+---------------+-------+--------+
SELECT * FROM order_goods_7
WHERE sell_order_id = 3794292749739495425
LIMIT 0, 1;
+--------------------+--------------------+--------------------+--------------+--------+------+
| order_goods_id     | sell_order_id      | goods_id           | user_guide_id | price  | num  |
+--------------------+--------------------+--------------------+--------------+--------+------+
| 3794293053134475265 | 3794292749739495425 | 3794292588254597121 |           7 | 630.00 |    2 |
+--------------------+--------------------+--------------------+--------------+--------+------+
```

11.3. 总结

分库不是 DBA 一个人的事。它牵扯到的太多太多，一定要和产品和开发一起慢慢琢磨。

12. MySQL 分库分表(番外篇 1)-使用 kafka 记入日志

12.1. 前言

其实这篇文章说介绍的和分库分表没有很大的关系，主要是业务流程的完整性。

12.2. 实际情况

在开发过程中许多重要的操作都会记录日志的。如：谁什么时候下单了、下单成功还是失败等等。

往往这些比较重要的信息都会记录到数据库中，而且记录日志的数据库硬件会比正式线上的差很多，因为这种日志数据库目的比较单纯只是为了记录用户行为信息。

如果将这些日志信息也变成是事务在程序中的话会严重影响到生产上的性能。

12.3. 解决方案

从业务的角度上看，像这种日志信息并不是那么的重要，被延时插入也是可以接收的。因此我们这边就使用消息队列的方式来记录这些日志。一来可以将日志完整的记录到日志库中，二来能很好的减小对线上的性能。

这边具体如何实现就不多少了，在之前的文章中都有提到如何使用 kafka，这边就给大家一个思路，扩展一下思维。

总的来说消息队列是很有用的。大家可以更具自己的实际情况来选择是否是用消息队列。

13. MySQL 分库分表(番外篇 2)-使用 redis

13.1. 前言

其实在系统中我们可以看到，那些属于公共信息的库和表最总将会成为一个高频爆发的点，系统在那里也容易出现瓶颈。因此我们为了提高系统的并发性和性能。我们可以选择一些 NoSQL 数据库。

13.2. 我的推荐

NoSQL 数据库其实有很多：redis、memcache、mongodb 等等。这部我个人比较倾向于 redis 的使用。在说为什么之前我们先要搞清楚用途。我们主要是为了把数据放在内存以便于能提高性能。因此 redis 的性能会比 memcache 好。而且 redis 在易用性和管理上会比 mongodb 来的容易。这边能减少一些运维的成本，同时 redis 对于开发来说还是比较熟悉的。

13.3. 在那里使用 redis

对于访问比较频繁并且改动很少的地方我们就能将其数据放在 redis 中。比如用户信息，系统配置信息等等。反正就已给原则，方便自己有能带来可观的效果，都能试一试。

14. MySQL 分库分表(番外篇 3)-小表冗余

其实在商品中都应该有这相关的类别信息。往往我们在查看商品的时候都需要知道此商品是什么类别。因此每一条商品都会关联到相关的类别。因此，在系统中就会有一张类别表。稍微有一点经验的都知道像这种类别表的数据一般都不会很大。因此，在分库分表系统中其实可以将这些类别表都冗余在不用的库中。主要了是为了方便查询，并且提高查询速度。基本思想就是将查询放在单机上。

版本记录

版本	更新内容	日期	操作人
V1.0	初始化文档	2016 年 03 月 06 日	HH