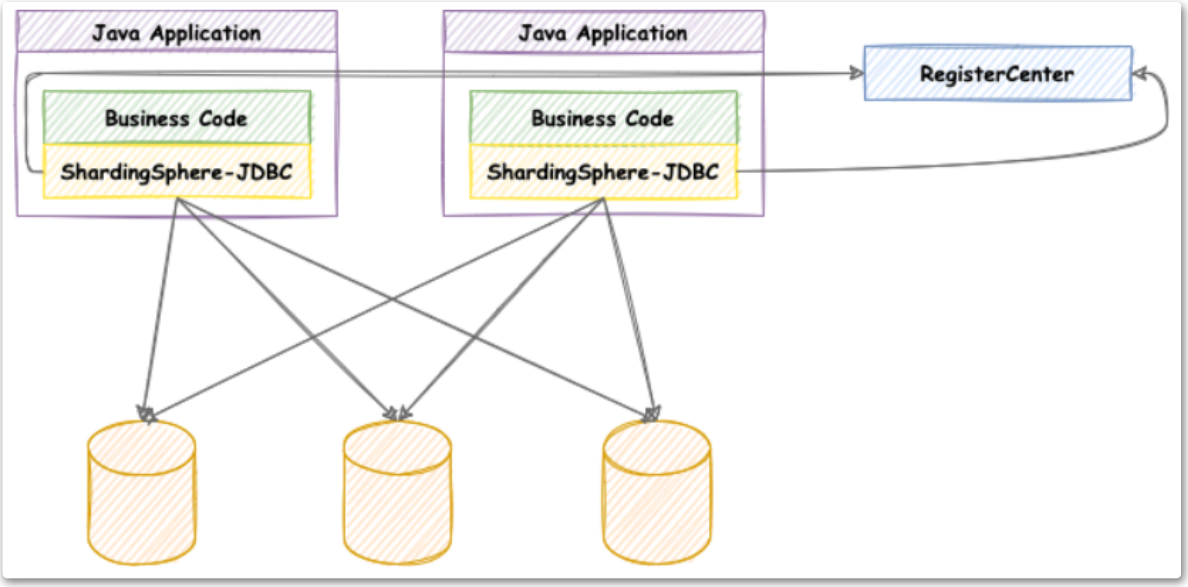
一、ShardingSphere产品介绍

ShardingSphere是一款起源于当当网内部的应用框架。2015年在当当网内部诞生，最初就叫ShardingJDBC。2016年的时候，由其中一个主要的开发人员张亮，带入到京东数科，组件团队继续开发。在国内历经了当当网、电信翼支付、京东数科等多家大型互联网企业的考验，在2017年开始开源。并逐渐由原本只关注于关系型数据库增强工具的ShardingJDBC升级成为一整套以数据分片为基础的数据生态圈，更名为ShardingSphere。到2020年4月，已经成为了Apache软件基金会的顶级项目。

ShardingSphere包含三个重要的产品，ShardingJDBC、ShardingProxy和ShardingSidecar。其中sidecar是针对service mesh定位的一个分库分表插件，目前在规划中。而我们今天学习的重点是ShardingSphere的JDBC和Proxy这两个组件。

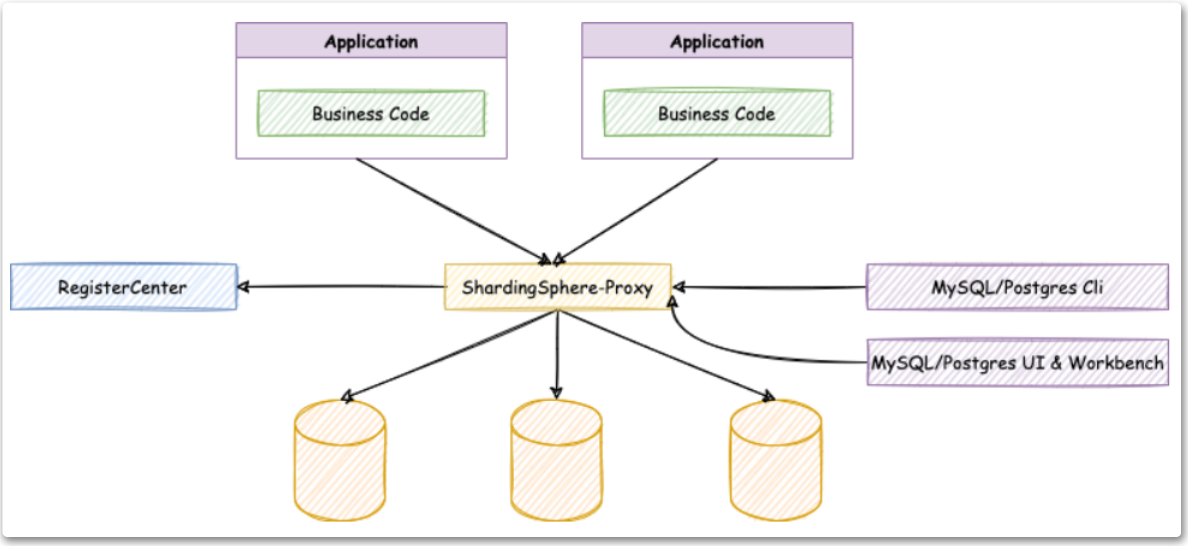
其中，ShardingJDBC是用来做客户端分库分表的产品，而ShardingProxy是用来做服务端分库分表的产品。这两者定位有什么区别呢？我们看下官方资料中给出的两个重要的图：

ShardingJDBC:



shardingJDBC定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

ShardingProxy



ShardingProxy定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。目前提供 MySQL 和 PostgreSQL 版本，它可以使用任何兼容 MySQL/PostgreSQL 协议的访问客户端。

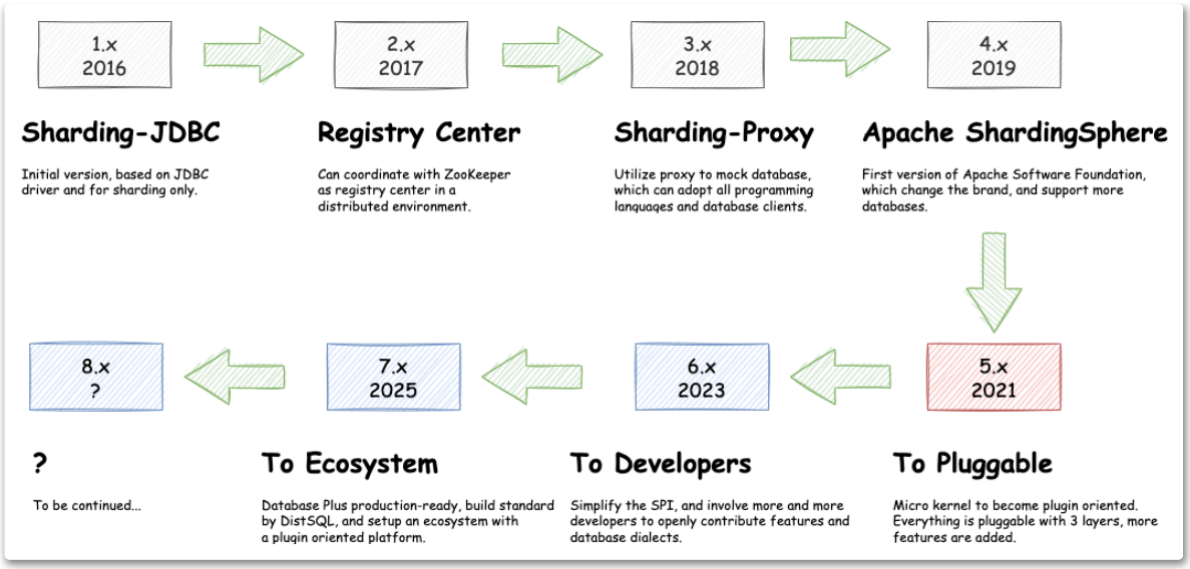
那这两种方式有什么区别呢?

	Sharding-JDBC	Sharding-Proxy
数据库	任意	MySQL/PostgreSQL
连接消耗数	高	低
异构语言	仅java	任意
性能	损耗低	损耗略高
无中心化	是	否
静态入口	无	有

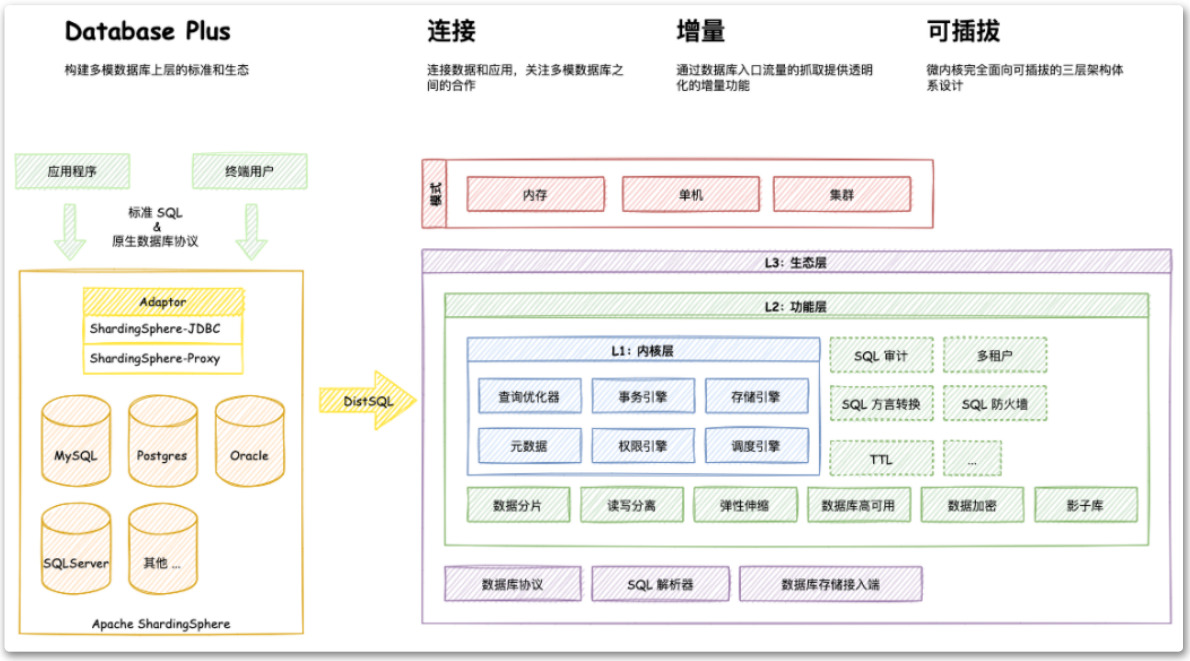
很显然，ShardingJDBC只是客户端的一个工具包，可以理解为一个特殊的JDBC驱动包，所有分库分表逻辑均由业务方自己控制，所以他的功能相对灵活，支持的数据库也非常多，但是对业务侵入大，需要业务方自己定制所有的分库分表逻辑。而ShardingProxy是一个独立部署的服务，对业务方无侵入，业务方可以像用一个普通的MySQL服务一样进行数据交互，基本上感觉不到后端分库分表逻辑的存在，但是这也意味着功能会比较固定，能够支持的数据库也比较少。这两者各有优劣。

二、ShardingSphere生态定位

对于ShardingSphere，大家用得多的一般是他的4.x版本，这也是目前最具影响力的一个系列版本。但是，ShardingSphere在2021年底，发布了5.x版本的第一个发布版，这也标志着ShardingSphere的产品定位进入了一个新的阶段。官网上也重点标识了一下ShardingSphere的发展路线:



其实从4.x版本升级到5.x版本，ShardingSphere做了很多功能增强，但是其核心功能并没有太大的变化。更大的区别其实是体现在产品定位上，在4.x版本中，ShardingSphere是定位为一个数据库中间件，而到了5.x版本，ShardingSphere给自己的定位成了DatabasePlus，旨在构建多模数据库上层的标准和生态，从而更接近于Sphere(生态)的定位。



其中核心的理念就是图中的连接、增量、可拔插。一方面未来会支持更多的数据库，甚至不光是MySQL、PostGreSQL这些关系型数据库，还包括了像RocksDB，Redis这一类非关系型的数据库。又一方面会拓展ShardingSphere的数据库功能属性，让用户可以完全基于ShardingSphere来构建上层应用，而其他的数据库只是作为ShardingSphere的可选功能支持。另一方面形成 微内核+三层可拔插扩展 的模型(图中的L1,L2,L3三层内核模型)，让开发者可以在ShardingSphere的内核基础上，做更灵活的功能拓展，可以像搭积木一样定制属于自己的独特系统。

虽然从目前来看，ShardingSphere离他自己构建的这个宏伟蓝图还非常遥远，但是从他逐渐清晰的功能定位可以看出，未来可期。而这也确确实实的带来了github上关注度增长。相比MyCat、DBLE等其他产品，未来更有吸引力。

由于ShardingSphere5.X版本还只提出短短几个月的时间，所以接下来的实战部分，我们依然会选用更为稳定的4.X版本。在课程最后会跟大家再来分享一下5.X版本的一些新特性。

三、ShardingJDBC实战

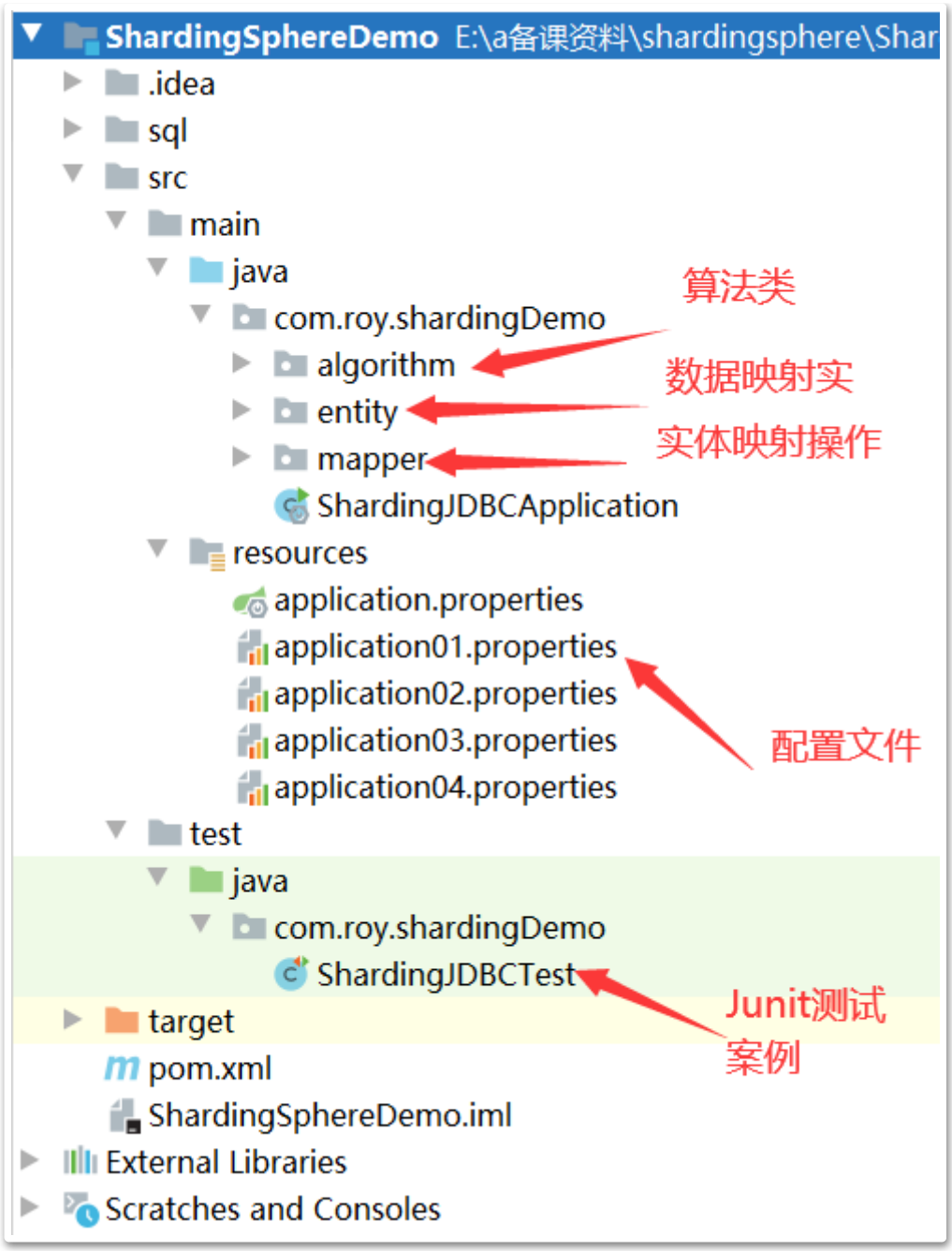
ShardingJDBC是整个ShardingSphere最早也是最为核心的一个功能模块，他的主要功能就是数据分片和读写分离，通过ShardingJDBC，应用可以透明的使用JDBC访问已经分库分表、读写分离的多个数据源，而不用关心数据源的数量以及数据如何分布。

1、核心概念：

- 逻辑表：水平拆分的数据库的相同逻辑和数据结构表的总称
- 真实表：在分片的数据库中真实存在的物理表。
- 数据节点：数据分片的最小单元。由数据源名称和数据表组成
- 绑定表：分片规则一致的主表和子表。
- 广播表：也叫公共表，指素有的分片数据源中都存在的表，表结构和表中的数据在每个数据库中都完全一致。例如字典表。
- 分片键：用于分片的数据库字段，是将数据库(表)进行水平拆分的关键字段。SQL中若没有分片字段，将会执行全路由，性能会很差。
- 分片算法：通过分片算法将数据进行分片，支持通过=、BETWEEN和IN分片。分片算法需要由应用开发者自行实现，可实现的灵活度非常高。
- 分片策略：真正用于进行分片操作的是分片键+分片算法，也就是分片策略。在ShardingJDBC中一般采用基于Groovy表达式的inline分片策略，通过一个包含分片键的算法表达式来制定分片策略，如t_user_$->{u_id%8}标识根据u_id模8，分成8张表，表名称为t_user_0到t_user_7。

2、测试项目介绍

测试项目参见配套的ShardingDemo项。首先我们对测试项目的结构做下简单的梳理：



> 注：1、引入MyBatisPlus依赖，简化JDBC操作，这样我们就不需要在代码中写SQL语句了。
>
> 2、entity中的实体对象就对应数据库中的表结构。而mapper中的接口则对应JDBC操作。
>
> 3、所有操作均使用JUnit的测试案例执行。 后续所有测试操作都会配合application.properties中的配置以及JUnit测试案例进行。
>
> 4、关于ShardingSphere版本，由于目前最新的5.0版本还在孵化当中，所以我们使用已发布的4.1.1版本来进行学习。

3、快速实战

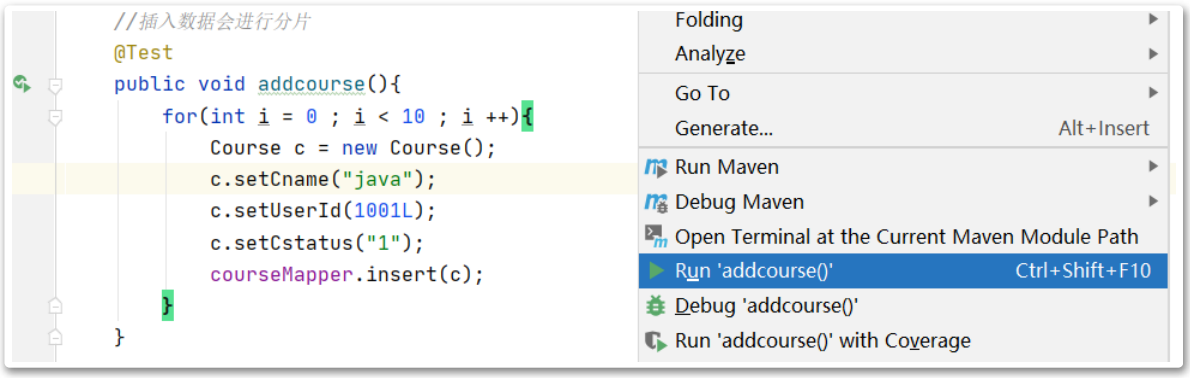
我们先运行一个简单的实例，来看下ShardingJDBC是如何工作的。

在application.properties配置文件中写入application01.properties文件的内容：

```
#垂直分表策略
# 配置真实数据源
spring.shardingsphere.datasource.names=m1

# 配置第 1 个数据源
spring.shardingsphere.datasource.m1.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m1.url=jdbc:mysql://localhost:3306/coursedb?serverTimezone=GMT%2B8
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root

# 指定表的分布情况 配置表在哪个数据库里，表名是什么。水平分表，分两个表：m1.course_1,m1.course_2
spring.shardingsphere.sharding.tables.course.actual-data-nodes=m1.course_$->{1..2}

# 指定表的主键生成策略
spring.shardingsphere.sharding.tables.course.key-generator.column=cid
spring.shardingsphere.sharding.tables.course.key-generator.type=SNOWFLAKE
#雪花算法的一个可选参数
spring.shardingsphere.sharding.tables.course.key-generator.props.worker.id=1

#使用自定义的主键生成策略
#spring.shardingsphere.sharding.tables.course.key-generator.type=MYKEY
#spring.shardingsphere.sharding.tables.course.key-generator.props.mykey-offset=88

#指定分片策略 约定cid值为偶数添加到course_1表。如果是奇数添加到course_2表。
# 选定计算的字段
spring.shardingsphere.sharding.tables.course.table-strategy.inline.sharding-column= cid
# 根据计算的字段算出对应的表名。
spring.shardingsphere.sharding.tables.course.table-strategy.inline.algorithm-expression=course_$->{cid%2+1}

# 打开sql日志输出。
spring.shardingsphere.props.sql.show=true

spring.main.allow-bean-definition-overriding=true
```

> 1、首先定义一个数据源m1，并对m1进行实际的JDBC参数配置
>
> 2、spring.shardingsphere.sharding.tables.course开头的一系列属性即定义了一个名为course的逻辑表。
>
> actual-data-nodes属性即定义course逻辑表的实际数据分布情况，他分布在m1.course_1和m1.course_2两个表。
>
> key-generator属性配置了他的主键列以及主键生成策略。ShardingJDBC默认提供了UUID和SNOWFLAKE两种分布式主键生成策略。
>
> table-strategy属性即配置他的分库分表策略。分片键为cid属性。分片算法为course_$->{cid%2+1}，表示按照cid模2+1的结果，然后加上前面的course_ 部分作为前缀就是他的实际表结果。注意，这个表达式计算出来的结果需要能够与实际数据分布中的一种情况对应上，否则就会报错。
>
> sql.show属性表示要在日志中打印实际SQL
>
> 3、coursedb的表结构见示例中sql文件夹中的sql语句。

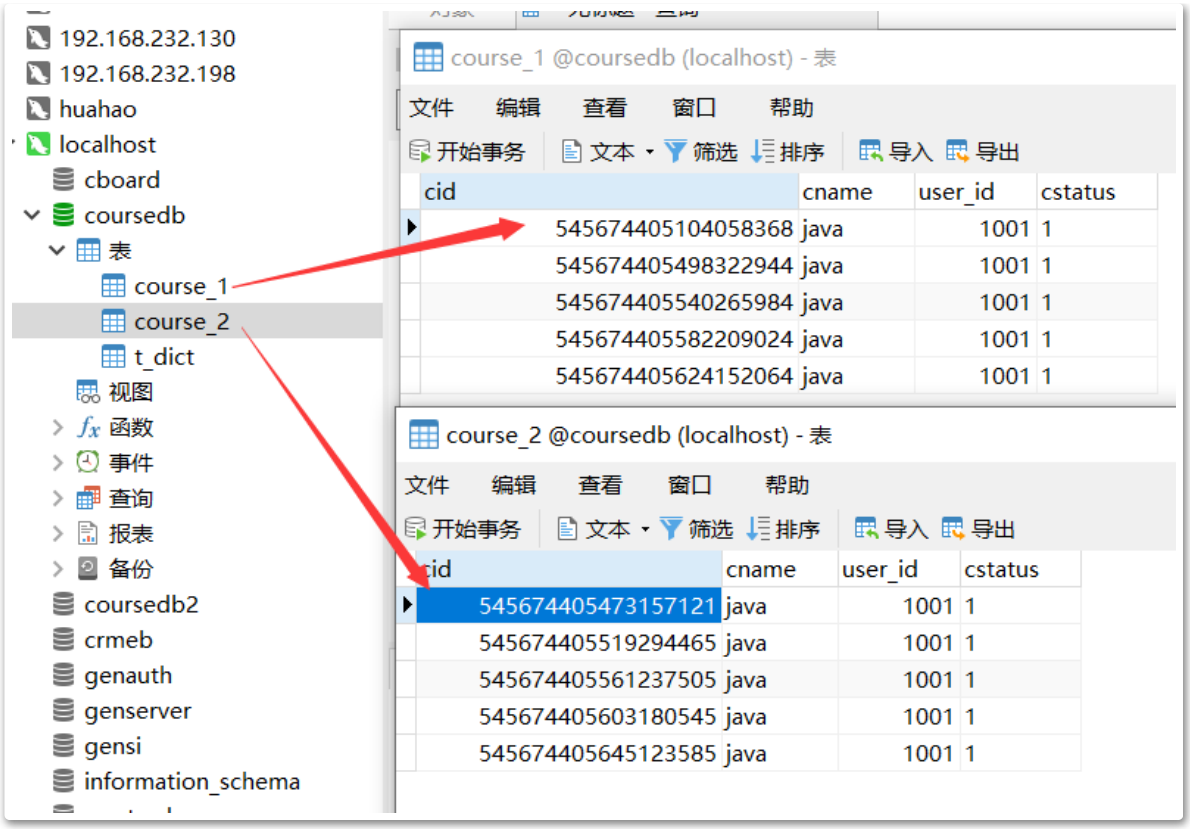
然后我们执行测试案例中的addcourse案例。



执行后，我们可以在控制台看到很多条这样的日志：

```
.......
2020-12-15 18:35:16.426  INFO 22412 --- [           main] ShardingSphere-SQL                       : Logic SQL: INSERT INTO course  ( cname,
user_id,
cstatus )  VALUES  ( ?,
?,
? )
2020-12-15 18:35:16.427  INFO 22412 --- [           main] ShardingSphere-SQL                       : SQLStatement: InsertStatementContext(super=CommonSQLStatementContext(sqlStatement=org.apache.shardingsphere.sql.parser.sql.statement.dml.InsertStatement@1cbc5693, tablesContext=org.apache.shardingsphere.sql.parser.binder.segment.table.TablesContext@124d26ba), tablesContext=org.apache.shardingsphere.sql.parser.binder.segment.table.TablesContext@124d26ba, columnNames=[cname, user_id, cstatus], insertValueContexts=[InsertValueContext(parametersCount=3, valueExpressions=[ParameterMarkerExpressionSegment(startIndex=59, stopIndex=59, parameterMarkerIndex=0), ParameterMarkerExpressionSegment(startIndex=62, stopIndex=62, parameterMarkerIndex=1), ParameterMarkerExpressionSegment(startIndex=65, stopIndex=65, parameterMarkerIndex=2), DerivedParameterMarkerExpressionSegment(super=ParameterMarkerExpressionSegment(startIndex=0, stopIndex=0, parameterMarkerIndex=3))], parameters=[java, 1001, 1])], generatedKeyContext=Optional[GeneratedKeyContext(columnName=cid, generated=true, generatedValues=[545674405561237505])])
2020-12-15 18:35:16.427  INFO 22412 --- [           main] ShardingSphere-SQL                       : Actual SQL: m1 ::: INSERT INTO course_2  ( cname,
user_id,
cstatus , cid)  VALUES  (?, ?, ?, ?) ::: [java, 1001, 1, 545674405561237505]
.....
```

从这个日志中我们可以看到，程序中执行的Logic SQL经过ShardingJDBC处理后，被转换成了Actual SQL往数据库里执行。执行的结果可以在MySQL中看到，course_1和course_2两个表中各插入了五条消息。这就是ShardingJDBC帮我们进行的数据库的分库分表操作。



然后，其他的几个配置文件依次对应了其他几种分库分表策略，我们可以一一演示一下。

- application02.properties: 分库分表示例配置。内置分片算法示例， inline、standard、complex、hint。广播表配置示例。
- application03.properties: 绑定表示例配置
- application04.properties: 读写分离示例配置

> 要注意理解在读写分离策略中，ShardingJDBC只能帮我们把读写操作分发到不同的数据库上，而数据库之间的数据同步，还是需要由MySQL主从集群来完成。

4、ShardingJDBC的分片算法

ShardingJDBC的整个实战完成后，可以看到，整个分库分表的核心就是在于配置的分片算法。我们的这些实战都是使用的inline分片算法，即提供一个分片键和一个分片表达式来制定分片算法。这种方式配置简单，功能灵活，是分库分表最佳的配置方式，并且对于绝大多数的分库分片场景来说，都已经非常好用了。但是，如果针对一些更为复杂的分片策略，例如多分片键、按范围分片等场景，inline分片算法就有点力不从心了。所以，我们还需要学习下ShardingSphere提供的其他几种分片策略。

ShardingSphere目前提供了一共五种分片策略：

- NoneShardingStrategy

  不分片。这种严格来说不算是一种分片策略了。只是ShardingSphere也提供了这么一个配置。

- InlineShardingStrategy

  最常用的分片方式

  - 配置参数： inline.shardingColumn 分片键；inline.algorithmExpression 分片表达式
  - 实现方式： 按照分片表达式来进行分片。

- StandardShardingStrategy

  只支持单分片键的标准分片策略。

  - 配置参数：standard.sharding-column 分片键；standard.precise-algorithm-class-name 精确分片算法类名；standard.range-algorithm-class-name 范围分片算法类名
  - 实现方式：

    shardingColumn指定分片算法。

    preciseAlgorithmClassName 指向一个实现了io.shardingsphere.api.algorithm.sharding.standard.PreciseShardingAlgorithm接口的java类名，提供按照 = 或者 IN 逻辑的精确分片 示例：`com.roy.shardingDemo.algorithm.MyPreciseShardingAlgorithm`

    rangeAlgorithmClassName 指向一个实现了io.shardingsphere.api.algorithm.sharding.standard.RangeShardingAlgorithm接口的java类名，提供按照Between 条件进行的范围分片。示例：`com.roy.shardingDemo.algorithm.MyRangeShardingAlgorithm`
  - 说明：

    其中精确分片算法是必须提供的，而范围分片算法则是可选的。

- ComplexShardingStrategy

  支持多分片键的复杂分片策略。

  - 配置参数：complex.sharding-columns 分片键(多个);complex.algorithm-class-name 分片算法实现类。
  - 实现方式：

    shardingColumn指定多个分片列。

algorithmClassName指向一个实现了org.apache.shardingsphere.api.sharding.complex.ComplexKeysShardingAlgorithm接口的java类名。提供按照多个分片列进行综合分片的算法。

示例：

```
com.roy.shardingDemo.algorithm.MyComplexKeysShardingAlgorithm
```

- HintShardingStrategy

  不需要分片键的强制分片策略。这个分片策略，简单来理解就是说，他的分片键不再跟SQL语句相关联，而是用程序另行指定。对于一些复杂的情况，例如 select count(*) from (select userid from t_user where userid in (1,3,5,7,9)) 这样的SQL语句，就没法通过SQL语句来指定一个分片键。这个时候就可以通过程序，给他另行执行一个分片键，例如在按userid奇偶分片的策略下，可以指定1作为分片键，然后自行指定他的分片策略。

  - 配置参数：hint.algorithm-class-name 分片算法实现类。
  - 实现方式：

    algorithmClassName指向一个实现了org.apache.shardingsphere.api.sharding.hint.HintShardingAlgorithm接口的java类名。 示例：

    ```
    com.roy.shardingDemo.algorithm.MyHintShardingAlgorithm
    ```

    在这个算法类中，同样是需要分片键的。而分片键的指定是通过HintManager.addDatabaseShardingValue方法(分库)和HintManager.addTableShardingValue(分表)来指定。

    使用时要注意，这个分片键是线程隔离的，只在当前线程有效，所以通常建议使用之后立即关闭，或者用try资源方式打开。

> 而Hint分片策略并没有完全按照SQL解析树来构建分片策略，是绕开了SQL解析的，所有对某些比较复杂的语句，Hint分片策略性能有可能会比较好(情况太多了，无法一一分析)。
>
> 但是要注意，Hint强制路由在使用时有非常多的限制：

```sql
-- 不支持UNION
SELECT * FROM t_order1 UNION SELECT * FROM t_order2
INSERT INTO tbl_name (col1, col2, …) SELECT col1, col2, … FROM tbl_name WHERE col3 = ?

-- 不支持多层子查询
SELECT COUNT(*) FROM (SELECT * FROM t_order o WHERE o.id IN (SELECT id FROM t_order WHERE status = ?))

-- 不支持函数计算。ShardingSphere只能通过SQL字面提取用于分片的值
SELECT * FROM t_order WHERE to_date(create_time, 'yyyy-mm-dd') = '2019-01-01';
```

> 示例详见application02.properties配置。
>
> 从这里也能看出，即便有了ShardingSphere框架，分库分表后对于SQL语句的支持依然是非常脆弱的。

5、ShardingSphere的SQL使用限制

参见官网文档： https://shardingsphere.apache.org/document/current/cn/features/sharding/use-norms/sql/ 文档中详细列出了非常多ShardingSphere目前版本支持和不支持的SQL类型。这些东西要经常关注。

支持的SQL

SQL	必要条件
SELECT * FROM tbl_name	
SELECT * FROM tbl_name WHERE (col1 = ? or col2 = ?) and col3 = ?	
SELECT * FROM tbl_name WHERE col1 = ? ORDER BY col2 DESC LIMIT ?	
SELECT COUNT(*), SUM(col1), MIN(col1), MAX(col1), AVG(col1) FROM tbl_name WHERE col1 = ?	
SELECT COUNT(col1) FROM tbl_name WHERE col2 = ? GROUP BY col1 ORDER BY col3 DESC LIMIT ?, ?	
INSERT INTO tbl_name (col1, col2,...) VALUES (?, ?, ...)	
INSERT INTO tbl_name VALUES (?, ?,....)	
INSERT INTO tbl_name (col1, col2, ...) VALUES (?, ?, ...), (?, ?, ...)	

SQL	必要条件
INSERT INTO tbl_name (col1, col2, ...) SELECT col1, col2, ... FROM tbl_name WHERE col3 = ?	INSERT表和SELECT表必须为相同表或绑定表
REPLACE INTO tbl_name (col1, col2, ...) SELECT col1, col2, ... FROM tbl_name WHERE col3 = ?	REPLACE表和SELECT表必须为相同表或绑定表
UPDATE tbl_name SET col1 = ? WHERE col2 = ?	
DELETE FROM tbl_name WHERE col1 = ?	
CREATE TABLE tbl_name (col1 int, ...)	
ALTER TABLE tbl_name ADD col1 varchar(10)	
DROP TABLE tbl_name	
TRUNCATE TABLE tbl_name	
CREATE INDEX idx_name ON tbl_name	
DROP INDEX idx_name ON tbl_name	
DROP INDEX idx_name	
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?	
SELECT COUNT(DISTINCT col1) FROM tbl_name	
SELECT subquery_alias.col1 FROM (select tbl_name.col1 from tbl_name where tbl_name.col2=?) subquery_alias	

不支持的SQL

SQL	不支持原因
INSERT INTO tbl_name (col1, col2, ...) VALUES(1+2, ?, ...)	VALUES语句不支持运算表达式
INSERT INTO tbl_name (col1, col2, ...) SELECT * FROM tbl_name WHERE col3 = ?	SELECT子句暂不支持使用*号简写及内置的分布式主键生成器
REPLACE INTO tbl_name (col1, col2, ...) SELECT * FROM tbl_name WHERE col3 = ?	SELECT子句暂不支持使用*号简写及内置的分布式主键生成器
SELECT * FROM tbl_name1 UNION SELECT * FROM tbl_name2	UNION
SELECT * FROM tbl_name1 UNION ALL SELECT * FROM tbl_name2	UNION ALL

SQL	不支持原因
SELECT SUM(DISTINCT col1), SUM(col1) FROM tbl_name	详见DISTINCT支持情况详细说明
SELECT * FROM tbl_name WHERE to_date(create_time, ‘yyyy-mm-dd’) = ?	会导致全路由
(SELECT * FROM tbl_name)	暂不支持加括号的查询
SELECT MAX(tbl_name.col1) FROM tbl_name	查询列是函数表达式时,查询列前不能使用表名;若查询表存在别名,则可使用表的别名

DISTINCT支持情况详细说明

支持的SQL

SQL
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?
SELECT DISTINCT col1 FROM tbl_name
SELECT DISTINCT col1, col2, col3 FROM tbl_name
SELECT DISTINCT col1 FROM tbl_name ORDER BY col1
SELECT DISTINCT col1 FROM tbl_name ORDER BY col2
SELECT DISTINCT(col1) FROM tbl_name
SELECT AVG(DISTINCT col1) FROM tbl_name
SELECT SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1
SELECT COUNT(DISTINCT col1 + col2) FROM tbl_name
SELECT COUNT(DISTINCT col1), SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1), col1 FROM tbl_name GROUP BY col1
SELECT col1, COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1

不支持的SQL

SQL	不支持原因
SELECT SUM(DISTINCT tbl_name.col1), SUM(tbl_name.col1) FROM tbl_name	查询列是函数表达式时,查询列前不能使用表名;若查询表存在别名,则可使用表的别名

6、分库分表带来的问题

1、分库分表，其实围绕的都是一个核心问题，就是单机数据库容量的问题。我们要了解，在面对这个问题时，解决方案是很多的，并不止分库分表这一种。但是ShardingSphere的这种分库分表，是希望在软件层面对硬件资源进行管理，从而便于对数据库的横向扩展，这无疑是成本很小的一种方式。

> 大家想想还有哪些比较好的解决方案？

2、一般情况下，如果单机数据库容量撑不住了，应先从缓存技术着手降低对数据库的访问压力。如果缓存使用过后，数据库访问量还是非常大，可以考虑数据库读写分离策略。如果数据库压力依然非常大，且业务数据持续增长无法估量，最后才考虑分库分表，单表拆分数据应控制在1000万以内。

当然，随着互联网技术的不断发展，处理海量数据的选择也越来越多。在实际进行系统设计时，最好是用MySQL数据库只用来存储关系性较强的热点数据，而对海量数据采取另外的一些分布式存储产品。例如PostGreSQL、VoltDB甚至HBase、Hive、ES等这些大数据组件来存储。

3、从上一部分ShardingJDBC的分片算法中我们可以看到，由于SQL语句的功能实在太多太全面了，所以分库分表后，对SQL语句的支持，其实是步步为艰的，稍不小心，就会造成SQL语句不支持、业务数据混乱等很多很多问题。所以，实际使用时，我们会建议这个分库分表，能不用就尽量不要用。

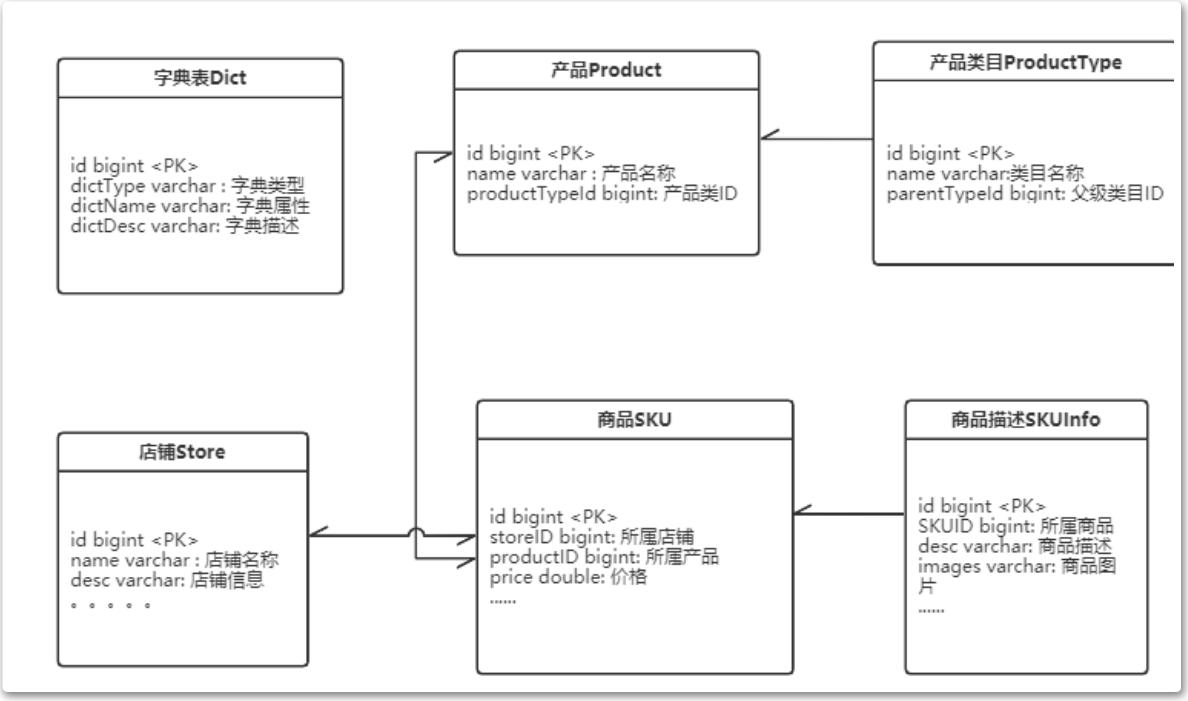
如果要使用优先在OLTP场景下使用，优先解决大量数据下的查询速度问题。而在OLAP场景中，通常涉及到非常多复杂的SQL，分库分表的限制就会更加明显。当然，这也是ShardingSphere以后改进的一个方向。

4、如果确定要使用分库分表，就应该在系统设计之初开始对业务数据的耦合程度和使用情况进行考量，尽量控制业务SQL语句的使用范围，将数据库往简单的增删改查的数据存储层方向进行弱化。并首先详细规划垂直拆分的策略，使数据层架构清晰明了。而至于水平拆分，会给后期带来非常非常多的数据问题，所以应该谨慎、谨慎再谨慎。一般也就在日志表、操作记录表等很少的一些边缘场景才偶尔用用。

7、分库分表方案设计实战

接下来，我们来给电商的商品管理模块设计一个分库分表的方案，来理解下分库分表应该如何落地。

一个典型的电商场景，商品管理模块大致的功能组件如下图：

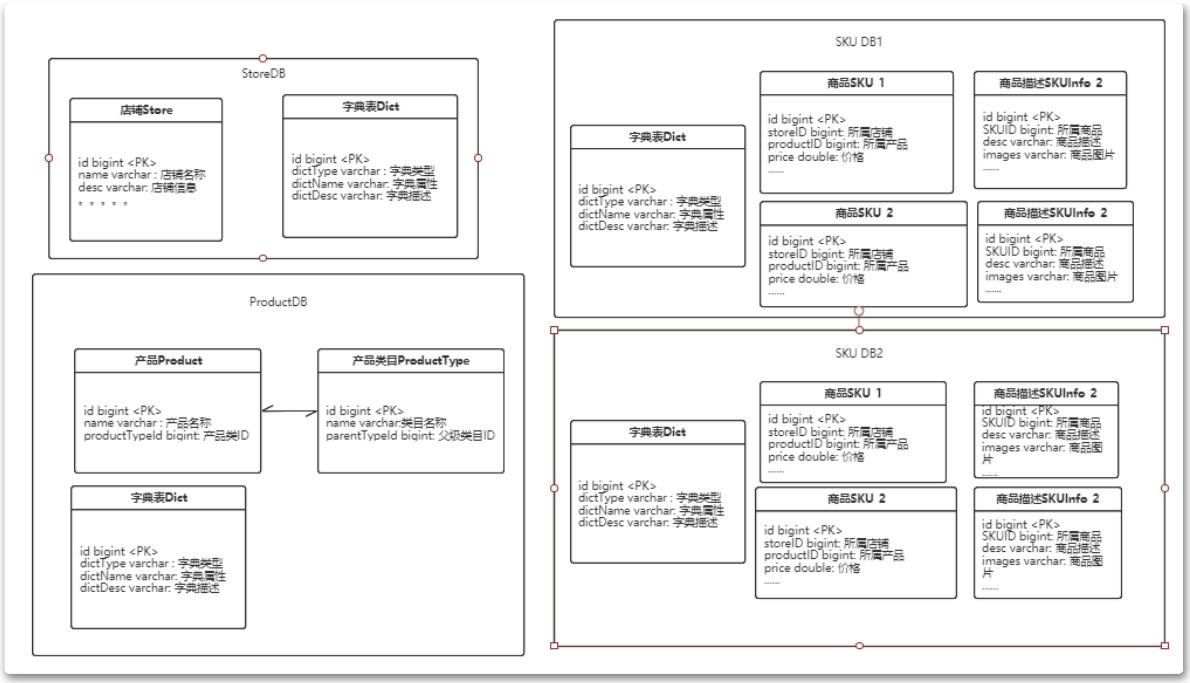


针对这个场景，考虑到商品信息会持续增长，越来越多的情况，要如何设计分库分表方案?

1、以业务为单位考虑对数据进行垂直分片，店铺、产品、商品三种业务数据垂直拆分成三个不同的库。字典表作为广播表冗余到三个不同的库中。

2、考虑数据增长情况，商品将会是以后增长最快的数据，店铺和产品的数据增速会逐渐降低。所以对商品表进行分片。

3、将关联性较强的商品信息表和商品补充信息表配置为绑定表。整体分库分表大致如下图：



思考：

1、对于两个库，每个库中两张表的分库分表策略，要如何定制分片策略将数据均匀分配到四个表中？application02.propertie

2、这种分库分表解决了哪些问题？支持的场景有哪些？不支持的场景有哪些？

四、分库分表与多数据源切换

从实战过程中看到，ShardingJDBC分库分表提供了对多个数据源的数据整合功能。因此，有很多同学就简单的认为ShardingJDBC就是用来做多个数据源的数据管理的。

比如在电商网站中可能会有这样的需求：根据用户所在城市不同，查询不同城市的商品数据。而在后台，这些不同城市的数据被分配在不同的数据库当中。很多同学想当然的就觉得需要使用ShardingJDBC来实现多数据源管理。

这种场景就是一个典型的多数据源切换的场景。但是我们仔细分析这样的场景，他跟分库分表其实并不太相同。ShardingJDBC固然可以使用Hint策略实现快速的数据库分库查询。例如前端传进来一个cityId字段，然后在后台查询数据时，将cityId设置到HintManager中，通过定制Hint策略，将后续的SQL操作分配到目标数据源

当中。

> 在课程的示例代码中，添加了一个使用AbstractDataSource实现数据源动态切换的示例。

但是，其实这种场景跟分库分表还是有差距的。分库分表关注的是数据的分布情况，而这种动态数据源切换的场景，只关注数据源分布，并不关心SQL语句是什么样的。所以一方面，这种场景下使用分库分表，需要给每个表都配置一个分库策略，会比较麻烦。另一方面，如果业务场景比较复杂，SQL语句也会多种多样。这些场景下，每写一个业务场景都需要考虑ShardingJDBC会如何转发SQL，如何改写SQL等等一系列的业务逻辑。比如Insert,Update,Delete这样的修改数据的SQL需要考虑是不是需要配置广播表，比如多表关联的复杂查询SQL需要时刻考虑是不是需要绑定表。

从架构层面考虑，数据库应该只是对业务的一个支撑工具，不应该影响到业务层的逻辑。而使用ShardingJDBC后，数据库层面针对分库分表的各种处理逻辑就被蔓延到了业务层，这种方式其实就是不太合适的。相反，使用示例中的数据源切换方式，数据库层对业务层的影响就被极大的压缩，只需要考虑数据源的问题，而不存在对SQL以及数据分布的影响。

所以，对于分库分表这种解决方案，一定需要结合你的实际业务场景，谨慎使用。比如ShardingSphere框架，如果不能全面把握好他的实现机制，就最好不要直接使用。在使用时，也尽量是针对最核心的数据，尽量控制影响范围。

> 有道云笔记分享链接：
>
> 文档：VIP02-ShardingJDBC分库分表实战与核心原理
> 链接：http://note.youdao.com/noteshare?id=b03df47cf349368c281f1fbdf1939316&sub=9CE839290B0D44A2B12E1B77EFD6F196