ShardingJDBC分库分表实战指南

-- 楼兰

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。发展至今，已经成为了业界分库分表最成熟的产品。

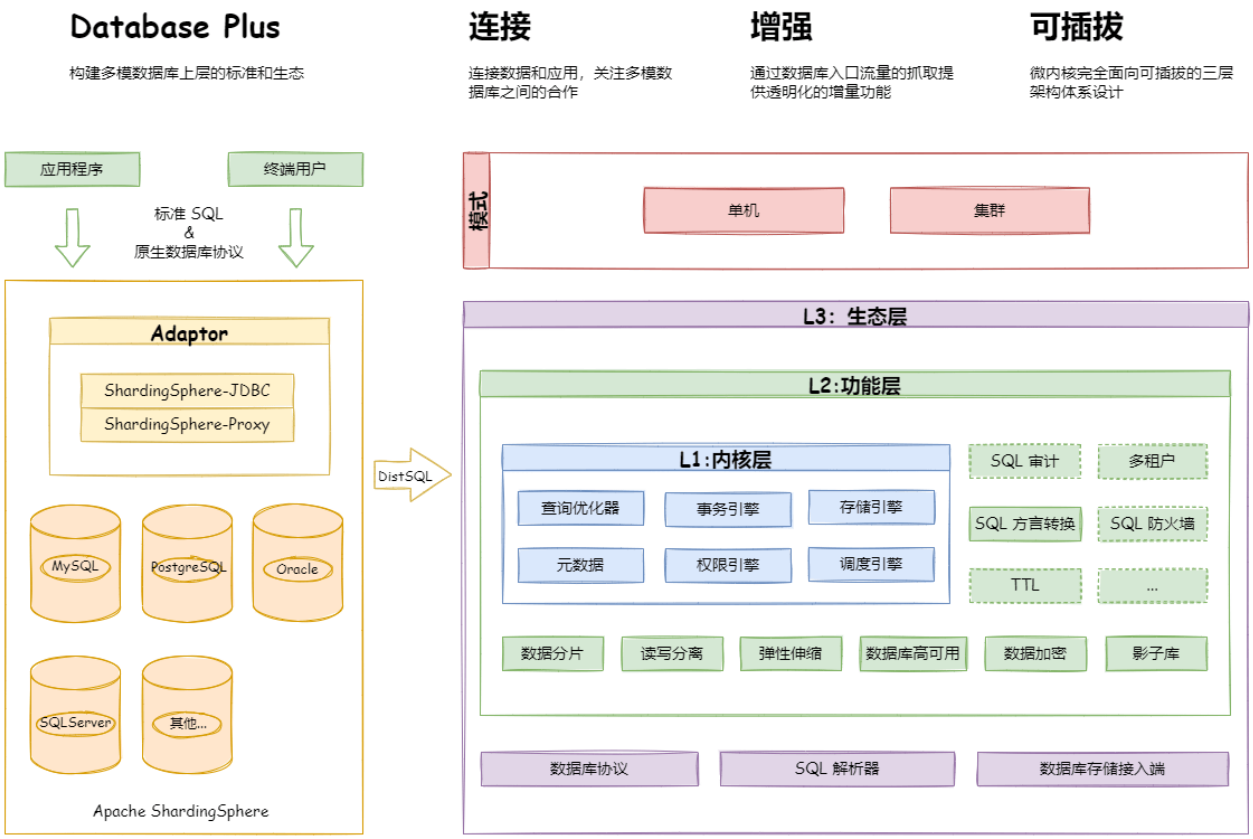
ShardingSphere这个词可以分为两个部分，其中Sharding就是我们之前介绍过的数据分片。从官网介绍上就能看到，他的核心功能就是可以将任意数据库组合，转换成为一个分布式的数据库，提供整体的数据库集群服务。只是给自己的定位并不是Database，而是Database plus。其实这个意思是他自己并不做数据存储，而是对其他数据库产品进行整合。整个ShardingSphere其实就是围绕数据分片这个核心功能发展起来的。后面的Sphere是生态的意思。这意味着ShardingSphere不是一个单独的框架或者产品，而是一个由多个框架以及产品构成的一个完整的技术生态。目前ShardingSphere中比较成型的产品主要包含核心的ShardingJDBC以及ShardingProxy两个产品，以及一个用于数据迁移的子项目ElasticJob，另外还包含围绕云原生设计的一系列未太成型的产品。

ShardingSphere经过这么多年的发展，已经不仅仅只是用来做分库分表，而是形成了一个围绕分库分表核心的技术生态。他的核心功能已经包括了数据分片、分布式事务、读写分离、高可用、数据迁移、联邦查询、数据加密、影子库、DistSQL庞大的技术体系。

另外需要注意的是ShardingSphere的版本演进路线。

当前最新的版本是5.x。可以看到，对于当前版本，ShardingSphere的核心是可插拔。其核心设计哲学就是连接、增强以及可拔插。这是官网对于其整个设计哲学的核心描述。

我们接下来会使用5.2.1版本的ShardingSphere。由于当前版本还比较新，企业中使用得还相对比较少，所以，在学习之前还要再强调一下，要带着研究的心态学习！



你现在当然不需要去了解各个细节，但是你应该要理解ShardingSphere是希望演进成一个重要的分库分表的功能核心。这个功能核心是构建在现有的数据库产品之上的，同时他可以支持大量的可插拔的上层应用扩展。这也意味着，在后面学习ShardingSphere时，你一定需要花更多心思去理解如何对ShardingSphere的功能进行扩展，而不能仅仅是学会如何使用ShardingSphere已经提供的功能。

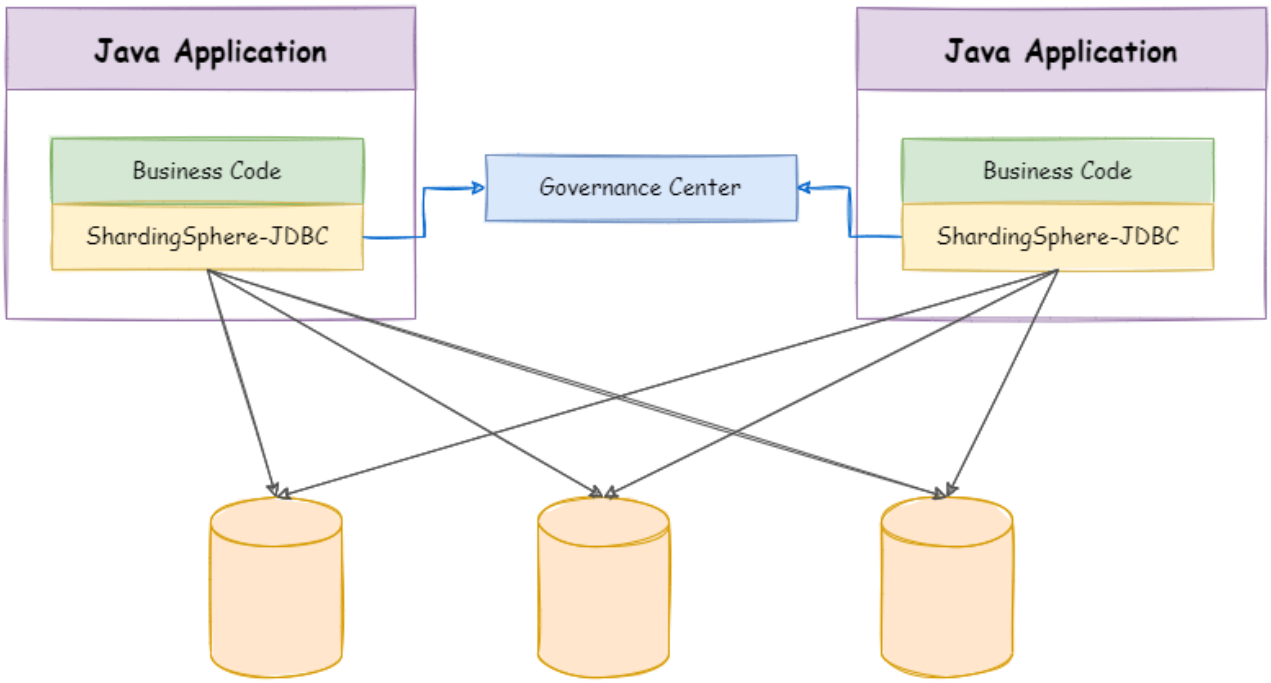
二、客户端分库分表与服务端分库分表

ShardingSphere最为核心的产品有两个：一个是ShardingJDBC，这是一个进行客户端分库分表的框架。另一个是ShardingProxy，这是一个进行服务端分库分表的产品。他们代表了两种不同的分库分表的实现思路。

1、ShardingJDBC客户端分库分表

ShardingSphere-JDBC 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。

- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，PostgreSQL，Oracle，SQLServer 以及任何可使用 JDBC 访问的数据库。

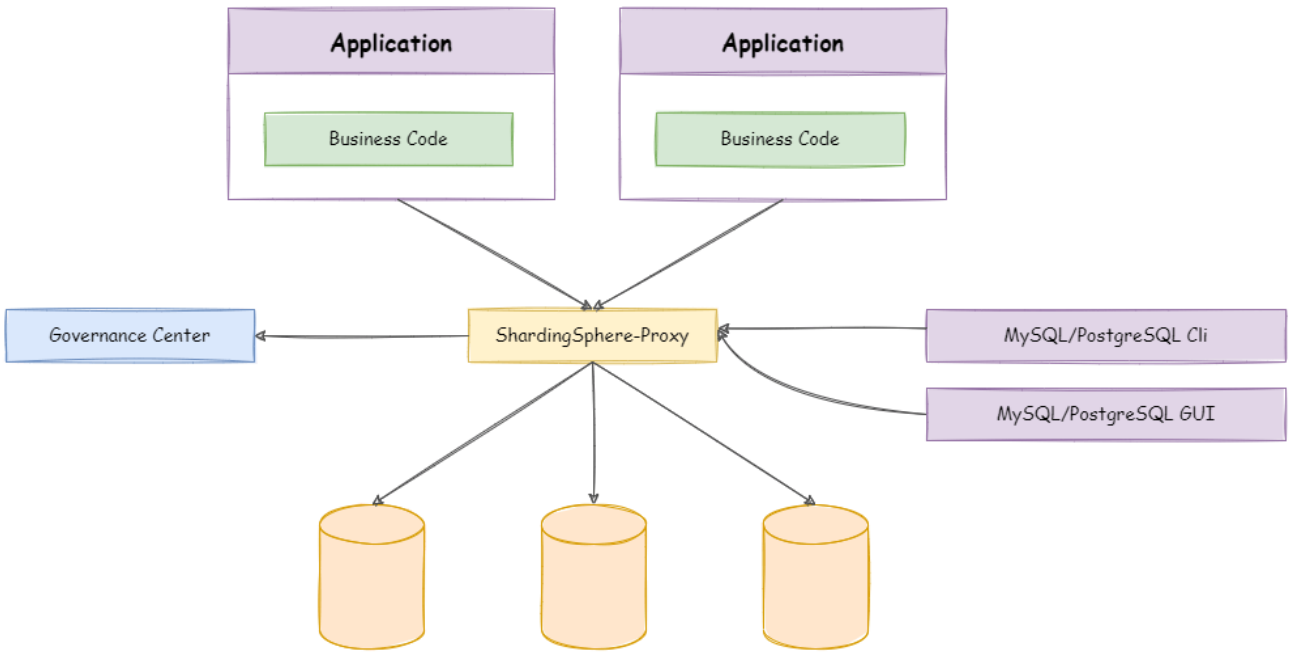


这是ShardingSphere最初的产品形态。

2、ShardingProxy服务端分库分表

ShardingSphere-Proxy 定位为透明化的数据库代理端，通过实现数据库二进制协议，对异构语言提供支持。 目前提供 MySQL 和 PostgreSQL 协议，透明化数据库操作，对 DBA 更加友好。

- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用；
- 兼容 MariaDB 等基于 MySQL 协议的数据库，以及 openGauss 等基于 PostgreSQL 协议的数据库；
- 适用于任何兼容 MySQL/PostgreSQL 协议的的客户端，如：MySQL Command Client, MySQL Workbench, Navicat 等。



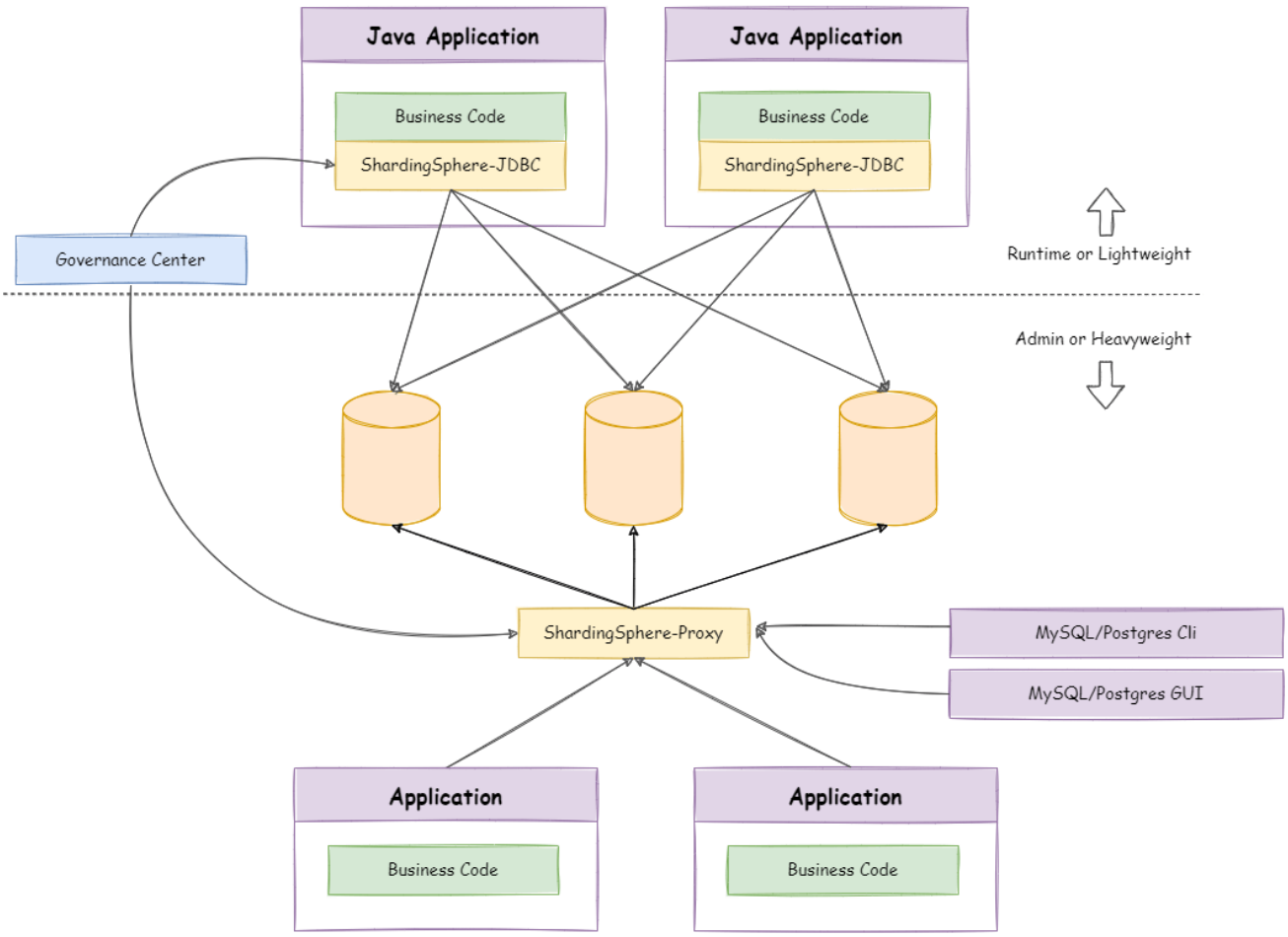
3、ShardingSphere混合部署架构

这两个产品都各有优势。ShardingJDBC跟客户端在一起，使用更加灵活。而ShardingProxy是一个独立部署的服务，所以他的功能相对就比较固定。他们的整体区别如下：

	ShardingSphere-JDBC	ShardingSphere-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

另外，在产品图中，Governance Center也是其中重要的部分。他的作用有点类似于微服务架构中的配置中心，可以使用第三方服务统一管理分库分表的配置信息，当前建议使用的第三方服务是Zookeeper，同时也支持Nacos，Etcd等其他第三方产品。

由于ShardingJDBC和ShardingProxy都支持通过Governance Center，将配置信息交个第三方服务管理，因此，也就自然支持了通过Governance Center进行整合的混合部署架构。

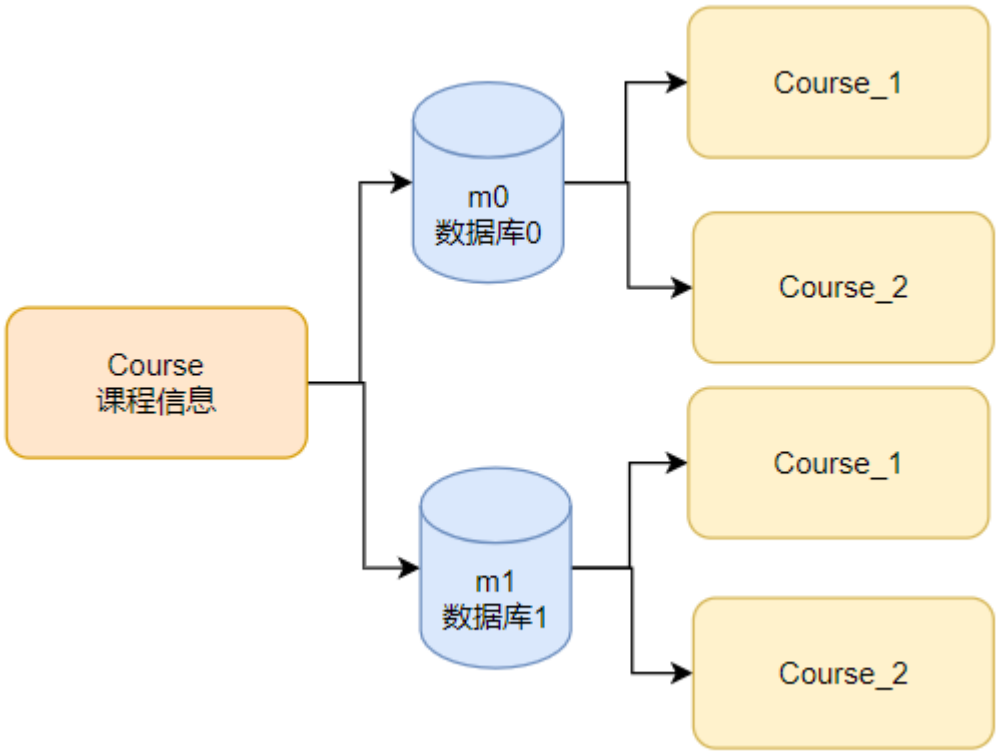


二、快速上手ShardingJDBC

接下来我们用ShardingJDBC快速实现一个简单的分库分表示例，了解一下ShardingJDBC是怎么工作的。

> 课程配套资料会给你全部的测试代码，但是我还是建议你自己动手搭建一次。这个搭建过程也不会很复杂。

我们预备将一批课程信息分别拆分到两个库中的两个表里。



1、搭建基础开发环境

接下来我们使用最常用的SpringBoot+MyBatis+MyBatis-plus快速搭建一个可以访问数据的简单应用，以这个应用作为我们分库分表的基础。

step1: 在数据库中创建course表，建表语句如下：

```
CREATE TABLE course  (
  `cid` bigint(0) NOT NULL AUTO_INCREMENT,
  `cname` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `user_id` bigint(0) NOT NULL,
  `cstatus` varchar(10) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  PRIMARY KEY (`cid`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

step2: 搭建一个Maven项目，在pom.xml中加入依赖，其中就包含访问数据库最为简单的几个组件。

```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
```

```
                <version>2.2.1.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>

            <!-- mybatisplus依赖 -->
            <dependency>
                <groupId>com.baomidou</groupId>
                <artifactId>mybatis-plus-boot-starter</artifactId>
                <version>3.0.5</version>
            </dependency>
            <dependency>
                <groupId>com.alibaba</groupId>
                <artifactId>druid-spring-boot-starter</artifactId>
                <version>1.1.20</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>
        <!-- 数据源连接池 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
            <version>1.1.20</version>
        </dependency>
        <!-- mysql连接驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <!-- mybatisplus依赖 -->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.4.3.3</version>
        </dependency>
    </dependencies>
```

step3: 使用MyBatis-plus的方式，直接声明Entity和Mapper，映射数据库中的course表。

```
public class Course {
    private Long cid;

    private String cname;
    private Long userId;
    private String cstatus;

    //省略。getter ... setter ....
}
```

```
public interface CourseMapper extends BaseMapper<Course> {
}
```

step4: 增加SpringBoot启动类，扫描mapper接口。

```
@SpringBootApplication
@MapperScan("com.roy.jdbcdemo.mapper")
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class,args);
    }
}
```

step5: 在springboot的配置文件application.properties中增加数据库配置。

```
spring.datasource.druid.db-type=mysql
spring.datasource.druid.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.druid.url=jdbc:mysql://localhost:3306/coursedb?serverTimezone=UTC
spring.datasource.druid.username=root
spring.datasource.druid.password=root
```

step6: 做一个单元测试，简单的把course课程信息插入到数据库，以及从数据库中进行查询。

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class JDBCTest {
    @Resource
    private CourseMapper courseMapper;
    @Test
    public void addcourse() {
        for (int i = 0; i < 10; i++) {
            Course c = new Course();
            c.setCname("java");
            c.setUserId(1001L);
            c.setCstatus("1");
            courseMapper.insert(c);
            //insert into course values ....
            System.out.println(c);
        }

    }
```

```
    @Test
    public void queryCourse() {
        QueryWrapper<Course> wrapper = new QueryWrapper<Course>();
        wrapper.eq("cid",1L);
        List<Course> courses = courseMapper.selectList(wrapper);
        courses.forEach(course -> System.out.println(course));
    }
}
```

OK，完成了！接下来执行单元测试，就可以完成与数据库的交互了。很简单，对吧？他将作为我们后续深入学习的基础。

2、引入ShardingSphere分库分表

接下来，我们将在这个简单案例上使用ShardingSphere快速Course表的分库分表功能。

step1:调整pom.xml中的依赖，引入ShardingSphere。

ShardingSphere的实现机制和我们之前章节中使用DynamicDataSource框架实现读写分离很类似，也是在底层注入一个带有分库分表功能的DataSource数据源。因此，在调整依赖时，需要注意不要直接使用druid-sprint-boot-starter依赖了。因为这个依赖会在Spring容器中注入一个DataSource，这样再要使用ShardingSphere注入DataSource就会产生冲突了。

```
<dependencies>
        <!-- shardingJDBC核心依赖 -->
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-jdbc-core-spring-boot-starter</artifactId>
            <version>5.2.1</version>
            <exclusions>
                <exclusion>
                    <artifactId>snakeyaml</artifactId>
                    <groupId>org.yaml</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <!-- 坑爹的版本冲突 -->
        <dependency>
            <groupId>org.yaml</groupId>
            <artifactId>snakeyaml</artifactId>
            <version>1.33</version>
        </dependency>
        <!-- SpringBoot依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <exclusions>
                <exclusion>
                    <artifactId>snakeyaml</artifactId>
                    <groupId>org.yaml</groupId>
                </exclusion>
            </exclusions>

        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>
        <!-- 数据源连接池 -->
        <!--注意不要用这个依赖，他会创建数据源，跟上面ShardingJDBC的SpringBoot集成依赖有冲突 -->
        <!--        <dependency>-->
        <!--            <groupId>com.alibaba</groupId>-->
        <!--            <artifactId>druid-spring-boot-starter</artifactId>-->
        <!--            <version>1.1.20</version>-->
        <!--        </dependency>-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.20</version>
        </dependency>
        <!-- mysql连接驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <!-- mybatisplus依赖 -->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.4.3.3</version>
        </dependency>
    </dependencies>
```

step2: 在对应数据库里创建分片表。

按照我们之前的设计，去对应的数据库中自行创建course_1和course_2表。但是这里要注意，在创建分片表时，cid字段就不要用自增长了。因为数据分到四个表后，每个表都自增长，就没办法保证cid字段的唯一性了。

step3: 增加ShardingJDBC的分库分表配置

然后，好玩的事情来了。应用层代码不需要做任何的修改，直接修改application.properties里的配置就可以完成我们之前设计的分库分表的目标。

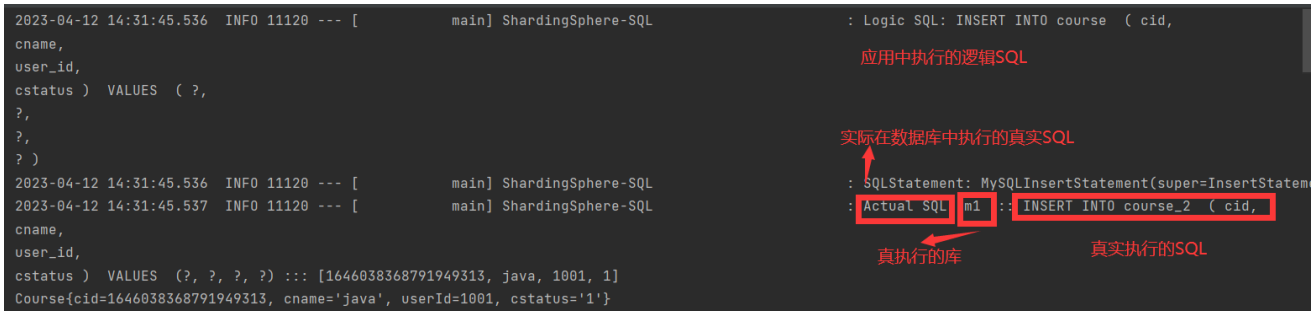
```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# ----------------数据源配置
# 指定对应的真实库
spring.shardingsphere.datasource.names=m0,m1
# 配置真实库
spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root
```

```
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/coursedb2?
serverTimezone=UTC
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root
#------------------------分布式序列算法配置
# 雪花算法，生成Long类型主键。
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.type=SNOWFLAKE
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.props.worker.id=1
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.course.key-generate-strategy.column=cid
spring.shardingsphere.rules.sharding.tables.course.key-generate-strategy.key-generator-
name=alg_snowflake
#-----------------------配置实际分片节点
spring.shardingsphere.rules.sharding.tables.course.actual-data-nodes=m$->{0..1}.course_$->
{1..2}
#-----------------------配置分库策略，按cid取模
spring.shardingsphere.rules.sharding.tables.course.database-strategy.standard.sharding-
column=cid
spring.shardingsphere.rules.sharding.tables.course.database-strategy.standard.sharding-
algorithm-name=course_db_alg

spring.shardingsphere.rules.sharding.sharding-algorithms.course_db_alg.type=MOD
spring.shardingsphere.rules.sharding.sharding-algorithms.course_db_alg.props.sharding-
count=2
#给course表指定分表策略  standard-按单一分片键进行精确或范围分片
spring.shardingsphere.rules.sharding.tables.course.table-strategy.standard.sharding-
column=cid
spring.shardingsphere.rules.sharding.tables.course.table-strategy.standard.sharding-
algorithm-name=course_tbl_alg
# 分表策略-INLINE：按单一分片键分表
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.props.algorithm-
expression=course_$->{cid%2+1}
```

> 配置过程，刚开始看会有点复杂，但是对应之前的设计图不难对应上。而且后面也会详细来解读配置过程。
>
> 这里主要需要理解一下的是配置中用到的Groovy表达式。 比如 m$->${0..1}.course_$->{1..2} 和 course_$->{cid%2+1} 。这是ShardingSphere支持的Groovy表达式，在后面会大量接触到这样的表达式。这个表达式中，$->{}部分为动态部分，大括号内的就是Groovy语句。 两个点，表示一个数据组的起点和终点。m$->${0..1}表示m0和m1两个字符串集合。course_$->{1..2}表示course_1和course_2集合。 course_$->{cid%2+1} 表示根据cid的值进行计算，计算的结果再拼凑上course_前缀。

接下来再次执行addcourse的单元测试，就能看到，十条课程信息被按照cid的奇偶，拆分到了m0.course_1和m1.course_2两张表中。

在日志里也能看到实际的执行情况。



> 这个示例中，course信息只能平均分到两个表中，而无法均匀分到四个表中。这其实是根据cid进行计算的结果。而将course_tbl_alg的计算表达式改成 course_$->{((cid+1)%4).intdiv(2)+1} 后，理论上，如果cid是连续递增的，就可以将数据均匀分到四个表里。但是snowflake雪花算法生成的ID并不是连续的，所以有时候还是无法分到四个表。

补充：尽量不要使用批量插入。

从这里可以看到，ShardingSphere实际上是将我们的SQL语句通过cid路由到了某一个分片上执行。那么可不可以把一个SQL语句路由到多个分片上执行呢?

比如在MySQL中，是可以在一个Insert语句里批量插入多条数据的。像这样

```
insert into course values (1,'java',1001,1),(2,'java',1001,1);
```

但是这样的SQL语句如果交给ShardingSphere去执行，那么这个语句就会造成困惑。到底应该往course_1还是course_2里插入？对于这种没有办法正确执行的SQL语句，ShardingSphere就会抛出异常。`Insert statement does not support sharding table routing to multiple data nodes.` 这个也是很多人在使用ShardingSphere时经常会遇到的问题。在这个示例中，可以将配置信息中 “配置分库策略，按cid取模” 的那一端配置中注释掉，也就是不生成cid，这样与批量插入的效果是一样的。也能模拟这种情况。

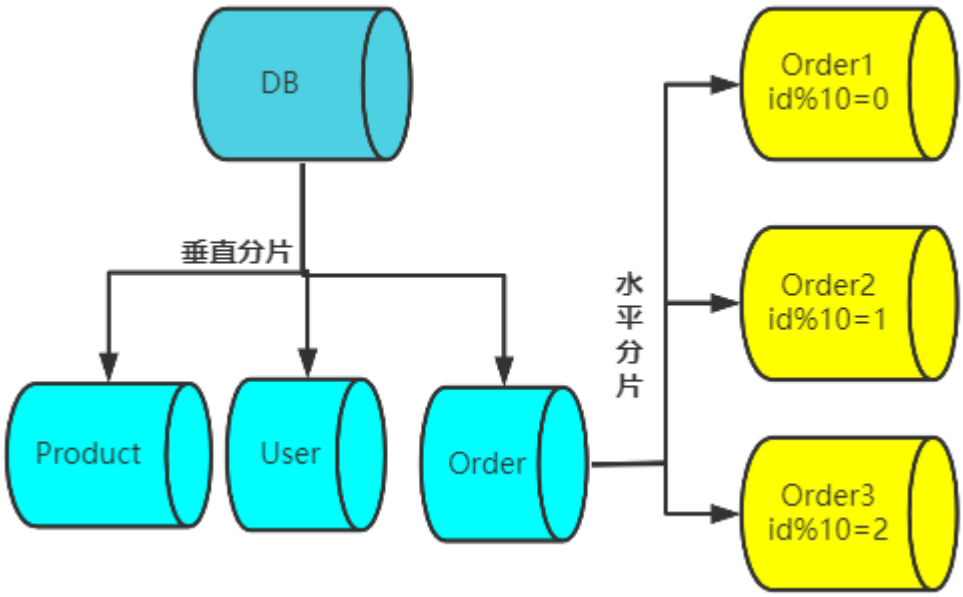
这个示例也说明了，ShardingSphere并不是支持所有的SQL语句。

三、理解ShardingSphere的核心概念

从这个简单示例中，我们可以接触到分库分表很多核心的概念。这些概念都是后面进行更复杂的分库分表时，需要大量运用的重要工具。

1、垂直分片与水平分片

这是设计分库分表方案时经常会提到的概念。 其中垂直分片表示按照业务的纬度，将不同的表拆分到不同的库当中。这样可以减少每个数据库的数据量以及客户端的连接数，提高查询效率。而水平分表表示按照数据的纬度，将原本存在同一张表中的数据，拆分到多张子表当中。每个子表只存储一份的数据。这样可以将数据量分散到多张表当中，减少每一张表的数据量，提升查询效率。



2、ShardingSphere实现分库分表的核心概念

接下来我们依次解析一下刚才示例中配置的一些重要的概念，可以对照一下之前的配置信息进行验证。

1. 虚拟库： ShardingSphere的核心就是提供一个具备分库分表功能的虚拟库，他是一个ShardingSphereDatasource实例。应用程序只需要像操作单数据源一样访问这个ShardingSphereDatasource即可。
2. 真实库： 实际保存数据的数据库。这些数据库都被包含在ShardingSphereDatasource实例当中，由ShardingSphere决定未来需要使用哪个真实库。
3. 逻辑表： 应用程序直接操作的逻辑表。
4. 真实表： 实际保存数据的表。这些真实表与逻辑表表名不需要一致，但是需要有相同的表结构，可以分布在不同的真实库中。应用可以维护一个逻辑表与真实表的对应关系，所有的真实表默认也会映射成为ShardingSphere的虚拟表。
5. 分布式主键生成算法： 给逻辑表生成唯一主键。由于逻辑表的数据是分布在多个真实表当中的，所有，单表的索引就无法保证逻辑表的ID唯一性。ShardingSphere集成了几种常见的基于单机生成的分布式主键生成器。比如SNOWFLAKE，COSID_SNOWFLAKE雪花算法可以生成单调递增的long类型的数字主键，还有UUID，NANOID可以生成字符串类型的主键。当然，ShardingSphere也支持应用自行扩展主键生成算法。比如基于Redis，Zookeeper等第三方服务，自行生成主键。
6. 分片策略： 表示逻辑表要如何分配到真实库和真实表当中，分为分库策略和分表策略两个部分。分片策略由分片键和分片算法组成。分片键是进行数据水平拆分的关键字段。如果没有分片键，ShardingSphere将只能进行全路由，SQL执行的性能会非常差。分片算法则表示根据分片键如何寻找对应的真实库和真实表。简单的分片策略可以使用Groovy表达式直接配置，当然，ShardingSphere也支持自行扩展更为复杂的分片算法。

   示例当中给course表分配配置了分库策略course_db_alg ，和分表策略 course_tbl_alg。其中course_db_alg是使用的ShardingSphere内置的简单算法MOD取模。如果对于字符串类型的主键，也提供了HASH_MOD进行计算。这两个算法都需要配置一个参数sharding-count分片数量。这种内置的算法虽然简单，但是不太灵活。 因为对2取模的结果只能是0和1，而对于course表来说，他的真实表是course_1和course_2，后缀需要在取模的结果上加1，这种计算就没法通过简单的取模算法实现了，所以需要通过Groovy表达式进行定制。

建议你先仔细停下来总结抽象一下这些概念。虽然他们可能并不是你未来进行分库分表时都需要实现的部分，但是，通过这些抽象的概念才能构建出一个完整的分库分表策略。并且，ShardingSphere中还有很多其他类似的概念，包括读写分离库、广播表、绑定表、影子库等等很多其他的概念，这些都需要在后面自己去抽象总结。

四、ShardingJDBC深入实战

理解这些基础概念之后，我们就继续深入更多的分库分表场景。下面的过程会通过一系列的问题来给你解释ShardingSphere最常用的分片策略。这个过程强烈建议你自己动手试试。因为不管你之前熟不熟悉ShardingSphere，你都需要一步步回顾总结一下分库分表场景下需要解决的是哪些稀奇古怪的问题。分库分表的问题非常非常多，你需要的是学会思考，而不是API。

1、简单INLINE分片算法

我们之前配置的简单分库分表策略已经可以根据自动生成的cid，将数据插入到不同的真实库当中。那么当然也支持按照cid进行数据查询。

```
    @Test
    public void queryCourse() {
        QueryWrapper<Course> wrapper = new QueryWrapper<Course>();
//        wrapper.eq("cid",851198095084486657L);
        wrapper.in("cid",851198095084486657L,851198095139012609L,851198095180955649L,4L);
        List<Course> courses = courseMapper.selectList(wrapper);
        courses.forEach(course -> System.out.println(course));
    }
```

像= 和 in 这样的操作，可以拿到cid的精确值，所以都可以直接通过表达式计算出可能的真实库以及真实表，ShardingSphere就会将逻辑SQL转去查询对应的真实库和真实表。这些查询的策略，只要配置了sql-show参数，都会打印在日志当中。

这里有几个比较有趣的问题：

第一个是，如果不使用分片键cid进行查询，那么SQL语句就只能根据actual-nodes到所有的真实库和真实表里查询。而这时ShardingSphere是怎么执行的呢？例如，如果直接执行select * from course，执行情况是这样的：

```
2023-04-12 15:55:02.958  INFO 12448 --- [           main] ShardingSphere-SQL
        : Logic SQL: SELECT  cid,cname,user_id,cstatus  FROM course

2023-04-12 15:55:02.958  INFO 12448 --- [           main] ShardingSphere-SQL
        : Actual SQL: m0 ::: SELECT  cid,cname,user_id,cstatus  FROM course_1 UNION ALL
SELECT  cid,cname,user_id,cstatus  FROM course_2
2023-04-12 15:55:02.958  INFO 12448 --- [           main] ShardingSphere-SQL
        : Actual SQL: m1 ::: SELECT  cid,cname,user_id,cstatus  FROM course_1 UNION ALL
SELECT  cid,cname,user_id,cstatus  FROM course_2
```

在之前4.x版本下，这种情况会拆分成四个SQL，查询四次。而当前版本下，会将每一个真实库里的语句通过UNION合并成一个大SQL，一起进行查询。为什么要这样呢？这其实是一个很大的优化。因为如果需要对一个真实库进行多个SQL查询，那么就需要通过多线程进行并发查询，这种情况下，如果要进行后续的结果归并，比如sum,max这样的结果归并，那就只能将所有的结果都合并到一个大内存，再进行归并。这种方式称为内存归并。

这种方式是比较消耗内存的。而如果合并成了一个大的SQL，对一个真实库只要进行一次SQL查询，这样就可以通过一个线程进行查询。在进行结果归并时，就可以拿一条数据归并一次。这种方式称为流式归并。相比内存归并可以极大的节约内存。

第二个是，在使用in进行查询时，有可能计算出属于多个不同的分片。在4.x版本当中，如果出现了这种情况，由于ShardingSphere无法确定in算出来的分片有多少个，所以遇到这种情况，他就不再去计算in中所有的分片结果了，直接改为全路由分片。这样计算比较简单，但是查询的效率肯定不好。而在当前版本下，则优化了这个问题。比如示例当中in操作的cid有奇数也有偶数，新版本就能够准确的计算出m0.course_1和m1.course_2两个分片。

另外还有第三个，在之前示例中，我们提到了一种改进的分片算法规则，可以将十条course数据分布到四个表当中，但是插入数据时，由于雪花算法的不连续性，很大可能还是只能插入到两张表里。那么到底是不是分片算法不对呢？你可以在这里确认一下。比如查询cid in (1L,2L,3L,4L,5L)，然后看一下他在ShardingSphere中是怎么执行的。

2、STANDARD标准分片算法

应用当中我们可能对于主键信息不只是进行精确查询，还需要进行范围查询。例如：

```
    @Test
    public void queryCourseRange(){
        //select * from course where cid between xxx and xxx
        QueryWrapper<Course> wrapper = new QueryWrapper<>();
        wrapper.between("cid",799020475735871489L,799020475802980353L);
        List<Course> courses = courseMapper.selectList(wrapper);
        courses.forEach(course -> System.out.println(course));
    }
```

这时，如果直接执行，那么由于ShardingSphere无法根据配置的表达式计算出可能的分片值，所以执行时他就会抛出一个异常。

```
org.springframework.jdbc.UncategorizedSQLException:
### Error querying database.  Cause: java.sql.SQLException: Unsupported SQL operation: Since the property of `allow-range-query-with-inline-sharding` is f
### The error may exist in com/roy/shardingDemo/mapper/CourseMapper.java (best guess)
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: SELECT  cid,cname,user_id,cstatus  FROM course     WHERE (cid BETWEEN ? AND ?)
### Cause: java.sql.SQLException: Unsupported SQL operation: Since the property of `allow-range-query-with-inline-sharding` is false, inline sharding algo
; uncategorized SQLException; SQL state [0A000]; error code [30001]; Unsupported SQL operation: Since the property of `allow-range-query-with-inline-shard
```

报错信息明确提到需要添加一个allow-range-query-with-inline-sharding参数。这时，就需要给course_tbl_alg算法添加这个参数。

```
# 允许在inline策略中使用范围查询。
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.props.allow-range-query-with-inline-sharding=true
```

加上这个参数后，就可以进行查询了。但是这样就可以了吗？观察一下Actual SQL的执行方式，你会发现这时SQL还是按照全路由的方式执行的。之前一直强调过，这是效率最低的一种执行方式。那么有没有办法通过查询时的范围下限和范围上限自己计算出一个目标真实库和真实表呢？当然是支持的。记住这个问题，在后续章节会带你解决。

3、COMPLEX_INLINE复杂分片算法

现在既然可以根据cid进行查询，那么还可以增加其他的查询条件吗？像这样：

```
    @Test
    public void queryCourseComplexSimple(){
        QueryWrapper<Course> wrapper = new QueryWrapper<Course>();
        wrapper.orderByDesc("user_id");
        wrapper.in("cid",851198095084486657L,851198095139012609L);
        wrapper.eq("user_id",1001L);
        List<Course> course = courseMapper.selectList(wrapper);
        //select * from couse where cid in (xxx) and user_id =xxx
        System.out.println(course);
    }
```

执行一下，这当然是可以的。但是有一个小问题，user_id查询条件只能参与数据查询，但是并不能参与到分片算法当中。例如在我们的示例当中，所有的user_id都是1001L，这其实是数据一个非常明显的分片规律。如果user_id的查询条件不是1001L，那这时其实不需要到数据库中去查，我们也能知道是不会有结果的。有没有办法让user_id也参与到分片算法当中呢?

当然是可以的， 不过STANDARD策略就不够用了。这时候就需要引入COMPLEX_INLINE策略。注释掉之前给course表配置的分表策略，重新分配一个新的分表策略：

```
#给course表指定分表策略  complex-按多个分片键进行组合分片
spring.shardingsphere.rules.sharding.tables.course.table-strategy.complex.sharding-
columns=cid,user_id
spring.shardingsphere.rules.sharding.tables.course.table-strategy.complex.sharding-
algorithm-name=course_tbl_alg
# 分表策略-COMPLEX：按多个分片键组合分表
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.type=COMPLEX_INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.props.algorithm-
expression=course_$->{(cid+user_id+1)%2+1}
```

在这个配置当中，就可以使用cid和user_id两个字段联合确定真实表。例如在查询时，将user_id条件设定为1002L，此时不管cid传什么值，就总是会路由到错误的表当中，查不出数据了。

4、CLASS_BASED自定义分片算法

到这，你可能会觉得这也有点太low了吧，这跟写个查不出数据的SQL语句好像没什么区别。这样查不出结果的实现方式，也完全体现不出分片策略的作用啊。还记得我们之前提到的STANDARD策略进行范围查询的问题吗？我们不妨设定一个稍微有一点实际意义的场景。

我们测试数据中的user_id都是固定的1001L，那么接下来我就可以希望在对user_id进行范围查询时，能够提前判断一些不合理的查询条件。而具体的判断规则，比如在对user_id进行between范围查询时，要求查询的下限不能超过查询上限，并且查询的范围必须包括1001L这个值。如果不满足这样的规则，那么就希望这个SQL语句就不要去数据库执行了。因为明显是不可能有数据的，还非要去数据库查一次，明显是浪费性能。

虽然对于COMPLEX_INLINE策略，也支持添加allow-range-query-with-inline-sharding参数让他能够支持分片键的范围查询，但是这时这种复杂的分片策略就明显不能再用一个简单的表达式来忽悠了。

这就需要一个Java类来实现这样的规则。这个算法类也不用自己瞎设计，只要实现ShardingSphere提供的ComplexKeysShardingAlgorithm接口就行了。

```
public class MyComplexAlgorithm implements ComplexKeysShardingAlgorithm<Long> {
```

```
    private static final String SHARING_COLUMNS_KEY = "sharding-columns";

    private Properties props;
    //保留配置的分片键。在当前算法中其实是没有用的。
    private Collection<String> shardingColumns;

    @Override
    public void init(Properties props) {
        this.props = props;
        this.shardingColumns = getShardingColumns(props);
    }

    /**
     * 实现自定义分片算法
     * @param availableTargetNames 在actual-nodes中配置了的所有数据分片
     * @param shardingValue 组合分片键
     * @return 目标分片
     */
    @Override
    public Collection<String> doSharding(Collection<String> availableTargetNames,
ComplexKeysShardingValue<Long> shardingValue) {
        //select * from cid where cid in (xxx,xxx,xxx) and user_id between {lowerEndpoint}
and {upperEndpoint};
        Collection<Long> cidCol =
shardingValue.getColumnNameAndShardingValuesMap().get("cid");
        Range<Long> userIdRange =
shardingValue.getColumnNameAndRangeValuesMap().get("user_id");
        //拿到user_id的查询范围
        Long lowerEndpoint = userIdRange.lowerEndpoint();
        Long upperEndpoint = userIdRange.upperEndpoint();
        //如果下限 》= 上限
        if(lowerEndpoint >= upperEndpoint){
            //抛出异常，终止去数据库查询的操作
            throw new UnsupportedShardingOperationException("empty record
query","course");
            //如果查询范围明显不包含1001
        }else if(upperEndpoint<1001L || lowerEndpoint>1001L){
            //抛出异常，终止去数据库查询的操作
            throw new UnsupportedShardingOperationException("error range query
param","course");
//            return result;
        }else{
            List<String> result = new ArrayList<>();
            //user_id范围包含了1001后，就按照cid的奇偶分片
            String logicTableName = shardingValue.getLogicTableName();//操作的逻辑表 course
            for (Long cidVal : cidCol) {
                String targetTable = logicTableName+"_"+(cidVal%2+1);
                if(availableTargetNames.contains(targetTable)){
                    result.add(targetTable);
                }
            }

            return result;
```

```
        }
    }

    private Collection<String> getShardingColumns(final Properties props) {
        String shardingColumns = props.getProperty(SHARING_COLUMNS_KEY, "");
        return shardingColumns.isEmpty() ? Collections.emptyList() :
Arrays.asList(shardingColumns.split(","));
    }

    public void setProps(Properties props) {
        this.props = props;
    }
    @Override
    public Properties getProps() {
        return this.props;
    }
}
```

在核心的dosharding方法当中，就可以按照我们之前的规则进行判断。不满足规则，直接抛出UnsupportedShardingOperationException异常，就可以组织ShardingSphere把SQL分配到真实数据库中执行。

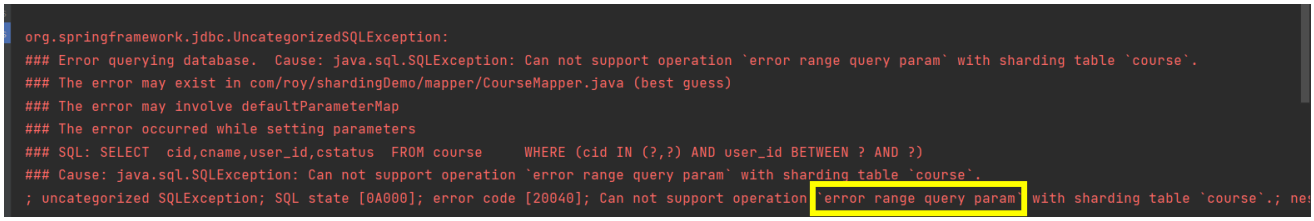
接下来，还是需要增加策略配置，让course表按照这个规则进行分片。

```
# 使用CLASS_BASED分片算法- 不用配置SPI扩展文件
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.type=CLASS_BASED
# 指定策略 STANDARD|COMPLEX|HINT
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.props.strategy=COMPLEX
# 指定算法实现类。这个类必须是指定的策略对应的算法接口的实现类。 STANDARD->
StandardShardingAlgorithm;COMPLEX->ComplexKeysShardingAlgorithm;HINT ->
HintShardingAlgorithm
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.props.algorithmClassName=com.roy.shardingDemo.algorithm.MyComple
xAlgorithm
```

这时我们再执行这样的查询：

```
    @Test
    public void queryCourdeComplex(){
        QueryWrapper<Course> wrapper = new QueryWrapper<Course>();
        wrapper.in("cid",799020475735871489L,799020475802980353L);
        wrapper.between("user_id",3L,8L);
        // wrapper.between("user_id",3L,3L);
        List<Course> course = courseMapper.selectList(wrapper);
        System.out.println(course);
    }
```

就会抛出异常。但在当前我们设定的业务场景下，这其实是对数据库性能的保护。



这里抛出的是SQLException，因为ShardingSphere实际上是在模拟成一个独立的虚拟数据库，在这个虚拟数据库中执行出现的异常，也都作为SQL异常抛出来。

另外，如果你尝试一下wrapper.between("user_id",8L,3L);这样的情况，那么ShardingSphere会抛出另外一个异常，这说明ShardingSphere对这种情况是已经有内部判断的。你当然不需要记住这些细枝末节的东西，但是如果你真的需要对这些异常做处理的时候，这些细节就能起到很多作用了。

> STANDARD策略要如何实现这样的自定义复杂分片算法呢？你可以自己尝试出来了吗？

5、HINT_INLINE强制分片算法

接下来我们把查询场景再进一步，需要查询所有cid为奇数的课程信息。这要怎么查呢？按照MyBatis-plus的机制，你应该很快能想到在CourseMapper中实现一个自定义的SQL语句就行了。

```
public interface CourseMapper extends BaseMapper<Course> {

    @Select("select * from course where MOD(cid,2)=1")
    List<Long> unsupportSql();
}
```

OK，拿过去试试。

```
    @Test
    public void unsupportTest(){
        //select * from course where mod(cid,2)=1
        List<Long> res = courseMapper.unsupportSql();
        res.forEach(System.out::println);
    }
```

执行结果当然是没有问题。但是你会发现，分片的问题又出来了。在我们当前的这个场景下，course的信息就是按照cid的奇偶分片的，所以自然是希望只去查某一个真实表就可以了。这种基于虚拟列的查询语句，对于ShardingSphere来说实际上是一块难啃的骨头。因为他很难解析出你是按照cid分片键进行查询的，并且不知道怎么组织对应的策略去进行分库分表。所以他的做法只能又是性能最低的全路由查询。

> 实际上ShardingSphere无法正常解析的语句还有很多。基本上用上分库分表后，你的应用就应该要和各种多表关联查询、多层嵌套子查询、distinct查询等各种复杂查询分手了。

这个cid的奇偶关系并不能通过SQL语句正常体现出来，这时，就需要用上ShardingSphere提供的另外一种分片算法HINT强制路由。HINT强制路由可以用一种与SQL无关的方式进行分库分表。

注释掉之前给course表分配的分表算法，重新分配一个HINT_INLINE类型的分表算法

```
#给course表指定分表策略  hint-与SQL无关的方式进行分片
spring.shardingsphere.rules.sharding.tables.course.table-strategy.hint.sharding-algorithm-name=course_tbl_alg
# 分表策略-HINT：用于SQL无关的方式分表，使用value关键字。
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.type=HINT_INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.props.algorithm-expression=course_$->{value}
```

然后，在应用进行查询时，使用HintManager给HINT策略指定value的值。

```
    @Test
    public void queryCourseByHint(){
        //强制只查course_1表
        HintManager hintManager = HintManager.getInstance();
        // 强制查course_1表
        hintManager.addTableShardingValue("course","1");
        //select * from course;
        List<Course> courses = courseMapper.selectList(null);
        courses.forEach(course -> System.out.println(course));
        //线程安全，所有用完要注意关闭。
        hintManager.close();
        //hintManager关闭的主要作用是清除ThreadLocal，释放内存。HintManager实现了AutoCloseable接口，所以建议使用try-resource的方式，用完自动关闭。
        //try(HintManager hintManager = HintManager.getInstance()){ xxxx }
    }
```

这样就可以让SQL语句只查询course_1表，在当前场景下，也就相当于是实现了只查cid为奇数的需求。

6、常用策略总结

在之前的示例中就介绍了ShardingSphere提供的MOD、HASH-MOD这样的简单内置分片策略，standard、complex、hint三种典型的分片策略以及CLASS_BASED这种扩展分片策略的方法。为什么要有这么多的分片策略，其实就是以为分库分表面临的业务场景其实是很复杂的。即便是ShardingSphere，也无法真的像MySQL、Oracle这样的数据库产品一样，完美的兼容所有的SQL语句。因此，一旦开始决定用分库分表，那么后续业务中的每一个SQL语句就都需要结合分片策略进行思考，不能像操作真正数据库那样随心所欲了。

五、其他常见的分片策略

最复杂的数据分片策略理解清楚了之后，相信你对于ShardingSphere的那些核心概念已经有所掌握了。接下来再介绍另外几种比较常见的分片策略，就能够简单很多了。

1、分片审计

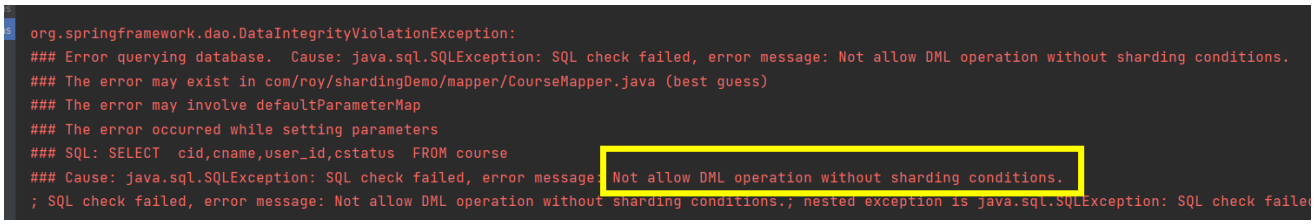
分片审计功能是针对数据库分片场景下对执行的 SQL 语句进行审计操作。分片审计既可以进行拦截操作，拦截系统配置的非法 SQL 语句，也可以是对 SQL 语句进行统计操作。

目前ShardingSphere内置的分片审计算法只有一个，DML_SHARDING_CONDITIONS。他的功能是要求对逻辑表查询时，必须带上分片键。

例如在之前的示例中，给course表配置一个分片审计策略

```
# 分片审计规则： SQL查询必须带上分片键
spring.shardingsphere.rules.sharding.tables.course.audit-strategy.auditor-
names[0]=course_auditor
spring.shardingsphere.rules.sharding.tables.course.audit-strategy.allow-hint-disable=true

spring.shardingsphere.rules.sharding.auditors.course_auditor.type=DML_SHARDING_CONDITIONS
```

这样，再次执行之前HINT策略的示例，就会报错。



当前这个策略看起来好像用处不是很大。但是，别忘了ShardingSphere可插拔的设计。这是一个扩展点，可以自行扩展出很多有用的功能的。

> 具体要如何扩展，等下个章节带你看懂官方文档了，你就知道了的。

2、数据加密

ShardingSphere内置了多种加密算法，可以用来快速对关键数据进行加密。

新创建一张user用户表，来测试下数据加密的功能。

```
CREATE TABLE user  (
  `userid` varchar(64) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `username` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `password` varchar(64) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `password_cipher` varchar(64) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `userstatus` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `age` int(0) DEFAULT NULL,
  `sex` varchar(2) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL COMMENT 'F or
M',
  PRIMARY KEY (`userid`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

> 1、在这个示例中会增加测试字符串型分片键的分片功能。
>
> 2、在coursedb和coursedb2两个数据库中分别创建user_1和user_2两种分片表。

创建对应的数据实体。

```
@TableName("user")
public class User {
    private String userid;
    private String username;
    private String password;
    private String userstatus;
    private int age;
    private String sex;
 // getter ... setter ...
}
```

创建操作数据库的Mapper

```
public interface UserCourseInfoMapper extends BaseMapper<UserCourseInfo> {
}
```

加密配置

```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# ----------------数据源配置
# 指定对应的库
spring.shardingsphere.datasource.names=m0,m1

spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root

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/coursedb2?
serverTimezone=UTC
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root
#------------------------分布式序列算法配置
# 生成字符串类型分布式主键。
spring.shardingsphere.rules.sharding.key-generators.user_keygen.type=NANOID
#spring.shardingsphere.rules.sharding.key-generators.user_keygen.type=UUID
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.user.key-generate-strategy.column=userid
spring.shardingsphere.rules.sharding.tables.user.key-generate-strategy.key-generator-
name=user_keygen
#-----------------------配置实际分片节点
spring.shardingsphere.rules.sharding.tables.user.actual-data-nodes=m$->{0..1}.user_$->
{1..2}
# HASH_MOD分库
spring.shardingsphere.rules.sharding.tables.user.database-strategy.standard.sharding-
```

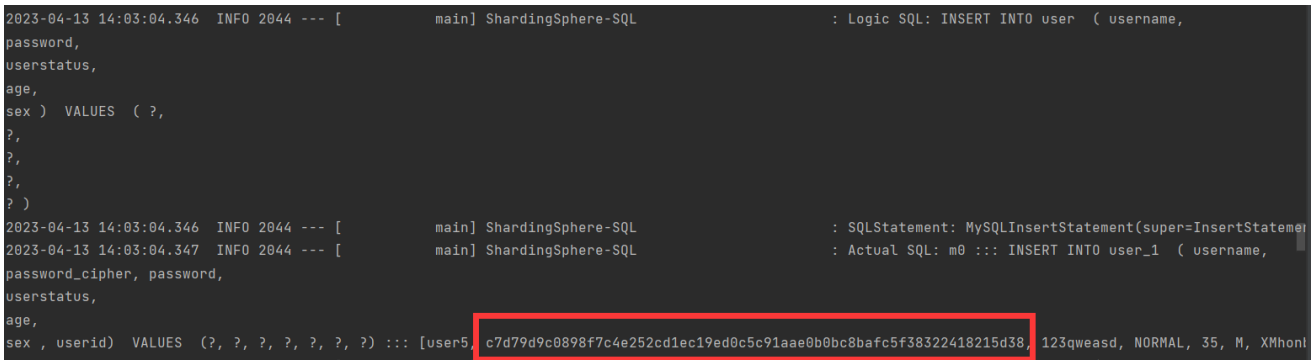
```
column=userid
spring.shardingsphere.rules.sharding.tables.user.database-strategy.standard.sharding-algorithm-name=user_db_alg

spring.shardingsphere.rules.sharding.sharding-algorithms.user_db_alg.type=HASH_MOD
spring.shardingsphere.rules.sharding.sharding-algorithms.user_db_alg.props.sharding-count=2
# HASH_MOD分表
spring.shardingsphere.rules.sharding.tables.user.table-strategy.standard.sharding-column=userid
spring.shardingsphere.rules.sharding.tables.user.table-strategy.standard.sharding-algorithm-name=user_tbl_alg

spring.shardingsphere.rules.sharding.sharding-algorithms.user_tbl_alg.type=INLINE
# 字符串类型要先hashcode转为long，再取模。但是Grovvy的 "xxx".hashcode%2 不知道为什么会产生 -1,0,1三种结果
#spring.shardingsphere.rules.sharding.sharding-algorithms.user_tbl_alg.props.algorithm-expression=user_$->{Math.abs(userid.hashCode()%2) +1}
# 用户信息分到四个表
spring.shardingsphere.rules.sharding.sharding-algorithms.user_tbl_alg.props.algorithm-expression=user_$->{Math.abs(userid.hashCode()%4).intdiv(2) +1}
# 数据加密:对password字段进行加密
# 存储明文的字段
spring.shardingsphere.rules.encrypt.tables.user.columns.password.plainColumn = password
# 存储密文的字段
spring.shardingsphere.rules.encrypt.tables.user.columns.password.cipherColumn = password_cipher
# 加密器
spring.shardingsphere.rules.encrypt.tables.user.columns.password.encryptorName = user_password_encry
# AES加密器
#spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.type=AES
#spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.props.aes-key-value=123456
# MD5加密器
#spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.type=MD5
# SM3加密器
spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.type=SM3
spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.props.sm3-salt=12345678

# sm4加密器
#spring.shardingsphere.rules.encrypt.encryptors.user_password_encry.type=SM4
```

测试案例

```
    @Test
    public void addUser(){
        for (int i = 0; i < 10; i++) {
            User user = new User();
//            user.setUserid();
            user.setUsername("user"+i);
            user.setPassword("123qweasd");
            user.setUserstatus("NORMAL");
```

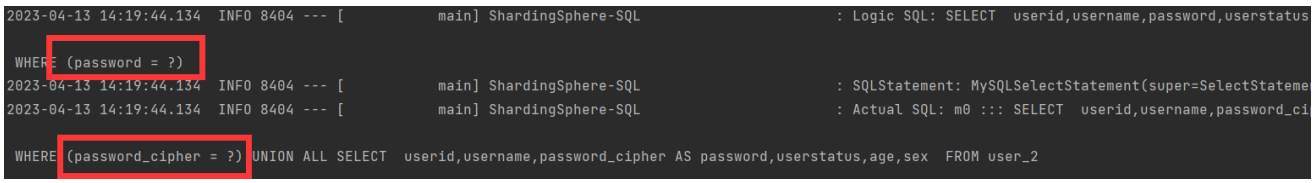
```
            user.setAge(30+i);
            user.setSex(i%2==0?"F":"M");

            userMapper.insert(user);
        }
    }
```

在插入时，就会在password_cipher中加入加密后的密文



接下来在查询时，可以主动传入password_cipher查询字段，按照密文进行查询。同时，针对password字段的查询，也会转化成为密文查询。查询案例

```
    @Test
    public void queryUser() {
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
//         queryWrapper.eq("userid","1644954727911317506");
        queryWrapper.eq("password","123qweasd");
//
queryWrapper.eq("password_cipher","c7d79d9c0898f7c4e252cd1ec19ed0c5c91aae0b0bc8bafc5f38322
418215d38");
        List<User> users = userMapper.selectList(queryWrapper);
        for(User user : users){
            System.out.println(user);
        }
    }
```



还有一些加密算法可以自己尝试一下。

3、读写分离

读写分离场景在之前章节已经带大家实现过。 在ShardingSphere中，实现读写分离也非常简单。只需要创建一个类型为readwrite-splitting的分片规则即可。我们就用之前建立的user表来快速配置一个读写分离示例。

```
# 打印SQL
spring.shardingsphere.props.sql-show = true

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

```
# ----------------数据源配置
# 指定对应的库
spring.shardingsphere.datasource.names=m0,m1

spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root

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/coursedb2?
serverTimezone=UTC
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root
#------------------------分布式序列算法配置
# 生成字符串类型分布式主键。
spring.shardingsphere.rules.sharding.key-generators.user_keygen.type=NANOID
#spring.shardingsphere.rules.sharding.key-generators.user_keygen.type=UUID
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.user.key-generate-strategy.column=userid
spring.shardingsphere.rules.sharding.tables.user.key-generate-strategy.key-generator-
name=user_keygen
#-----------------------配置读写分离
# 要配置成读写分离的虚拟库
spring.shardingsphere.rules.sharding.tables.user.actual-data-nodes=userdb.user
# 配置读写分离虚拟库 主库一个，从库多个
spring.shardingsphere.rules.readwrite-splitting.data-sources.userdb.static-strategy.write-
data-source-name=m0
spring.shardingsphere.rules.readwrite-splitting.data-sources.userdb.static-strategy.read-
data-source-names[0]=m1
# 指定负载均衡器
spring.shardingsphere.rules.readwrite-splitting.data-sources.userdb.load-balancer-
name=user_lb
# 配置负载均衡器
# 按操作轮训
spring.shardingsphere.rules.readwrite-splitting.load-balancers.user_lb.type=ROUND_ROBIN
# 按事务轮训
#spring.shardingsphere.rules.readwrite-splitting.load-
balancers.user_lb.type=TRANSACTION_ROUND_ROBIN
# 按操作随机
#spring.shardingsphere.rules.readwrite-splitting.load-balancers.user_lb.type=RANDOM
# 按事务随机
#spring.shardingsphere.rules.readwrite-splitting.load-
balancers.user_lb.type=TRANSACTION_RANDOM
# 读请求全部强制路由到主库
#spring.shardingsphere.rules.readwrite-splitting.load-balancers.user_lb.type=FIXED_PRIMARY
```

然后去执行对user表的插入和查询操作，从日志中就能体会到读写分离的实现效果。

> 这个配置看起来挺麻烦，但是思路和第一章实现读写分离是一样的，就是将多个真实库组合成一个虚拟库

4、广播表

广播表表指所有的分片数据源中都存在的表，表结构及其数据在每个数据库中均完全一致。适用于数据量不大且需要与海量数据的表进行关联查询的场景，例如：字典表。示例如下：

建表：

```
CREATE TABLE `coursedb`.`dict_1`  (
                                    `dictId` bigint(0) NOT NULL,
                                    `dictkey` varchar(32) CHARACTER SET utf8 COLLATE
utf8_general_ci DEFAULT NULL,
                                    `dictVal` varchar(32) CHARACTER SET utf8 COLLATE
utf8_general_ci DEFAULT NULL,
                                    PRIMARY KEY (`dictId`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

创建实体：

```
@TableName("dict")
public class Dict {
    private Long dictid;
    private String dictkey;
    private String dictval;

 // getter ... setter
}
```

创建mapper

```
public interface DictMapper extends BaseMapper<Dict> {
}
```

配置广播规则： 配置方式很简单。 直接配置broadcast-tables就可以了。

```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# ----------------数据源配置
# 指定对应的库
spring.shardingsphere.datasource.names=m0,m1

spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root
```

```
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/coursedb2?
serverTimezone=UTC
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root
#-----------------------分布式序列算法配置
# 生成字符串类型分布式主键。
spring.shardingsphere.rules.sharding.key-generators.dict_keygen.type=SNOWFLAKE
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.dict.key-generate-strategy.column=dictId
spring.shardingsphere.rules.sharding.tables.dict.key-generate-strategy.key-generator-
name=dict_keygen
#-----------------------配置读写分离
# 要配置成读写分离的虚拟库
#spring.shardingsphere.rules.sharding.tables.dict.actual-data-nodes=m$->{0..1}.dict_$->
{1..2}
spring.shardingsphere.rules.sharding.tables.dict.actual-data-nodes=m$->{0..1}.dict
# 指定广播表。广播表会忽略分表的逻辑，只往多个库的同一个表中插入数据。
spring.shardingsphere.rules.sharding.broadcast-tables=dict
```

测试示例

```
    @Test
    public void addDict() {
        Dict dict = new Dict();
        dict.setDictkey("F");
        dict.setDictval("女");
        dictMapper.insert(dict);

        Dict dict2 = new Dict();
        dict2.setDictkey("M");
        dict2.setDictval("男");
        dictMapper.insert(dict2);
    }
```

这样，对于dict字段表的操作就会被同时插入到两个库当中。

5、绑定表

绑定表指分片规则一致的一组分片表。使用绑定表进行多表关联查询时，必须使用分片键进行关联，否则会出现笛卡尔积关联或跨库关联，从而影响查询效率。

比如我们另外创建一张用户信息表，与用户表一起来演示这种情况

建表语句：`老规矩，自己进行分片`、

```
CREATE TABLE user_course_info  (
                                        `infoid` bigint(0) NOT NULL,
                                        `userid` varchar(64) CHARACTER SET utf8 COLLATE
utf8_general_ci DEFAULT NULL,
                                        `courseid` bigint(0) DEFAULT NULL,
                                        PRIMARY KEY (`infoid`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

接下来同样增加映射实体以及Mapper。这里就略过了。

然后配置分片规则：

```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# ----------------数据源配置
# 指定对应的库
spring.shardingsphere.datasource.names=m0

spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root
#------------------------分布式序列算法配置
# 生成字符串类型分布式主键。
spring.shardingsphere.rules.sharding.key-generators.usercourse_keygen.type=SNOWFLAKE
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.user_course_info.key-generate-
strategy.column=infoid
spring.shardingsphere.rules.sharding.tables.user_course_info.key-generate-strategy.key-
generator-name=usercourse_keygen
# ----------------------配置真实表分布
spring.shardingsphere.rules.sharding.tables.user.actual-data-nodes=m0.user_$->{1..2}
spring.shardingsphere.rules.sharding.tables.user_course_info.actual-data-
nodes=m0.user_course_info_$->{1..2}
# ----------------------配置分片
spring.shardingsphere.rules.sharding.tables.user.table-strategy.standard.sharding-
column=userid
spring.shardingsphere.rules.sharding.tables.user.table-strategy.standard.sharding-
algorithm-name=user_tbl_alg

spring.shardingsphere.rules.sharding.tables.user_course_info.table-
strategy.standard.sharding-column=userid
spring.shardingsphere.rules.sharding.tables.user_course_info.table-
strategy.standard.sharding-algorithm-name=usercourse_tbl_alg
# ----------------------配置分表策略
spring.shardingsphere.rules.sharding.sharding-algorithms.user_tbl_alg.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.user_tbl_alg.props.algorithm-
expression=user_$->{Math.abs(userid.hashCode()%4).intdiv(2) +1}
```

```
spring.shardingsphere.rules.sharding.sharding-algorithms.usercourse_tbl_alg.type=INLINE
spring.shardingsphere.rules.sharding.sharding-
algorithms.usercourse_tbl_alg.props.algorithm-expression=user_course_info_$->
{Math.abs(userid.hashCode()%4).intdiv(2) +1}
# 指定绑定表
spring.shardingsphere.rules.sharding.binding-tables[0]=user,user_course_info
```

然后把user表的数据都清空，重新插入一些有对应关系的用户和用户信息表。

```
@Test
    public void addUserCourseInfo(){
        for (int i = 0; i < 10; i++) {
            String userId = NanoIdUtils.randomNanoId();
            User user = new User();
            user.setUserid(userId);
            user.setUsername("user"+i);
            user.setPassword("123qweasd");
            user.setUserstatus("NORMAL");
            user.setAge(30+i);
            user.setSex(i%2==0?"F":"M");

            userMapper.insert(user);
            for (int j = 0; j < 5; j++) {
                UserCourseInfo userCourseInfo = new UserCourseInfo();
                userCourseInfo.setInfoid(System.currentTimeMillis()+j);
                userCourseInfo.setUserid(userId);
                userCourseInfo.setCourseid(10000+j);
                userCourseInfoMapper.insert(userCourseInfo);
            }
        }
    }
```

接下来按照用户ID进行一次关联查询。在UserCourseInfoMapper中配置SQL语句

```
public interface UserCourseInfoMapper extends BaseMapper<UserCourseInfo> {
    @Select("select uci.* from user_course_info uci ,user u where uci.userid = u.userid")
    List<UserCourseInfo> queryUserCourse();
}
```

查询案例：

```
    @Test
    public void queryUserCourseInfo(){
        List<UserCourseInfo> userCourseInfos = userCourseInfoMapper.queryUserCourse();
        for (UserCourseInfo userCourseInfo : userCourseInfos) {
            System.out.println(userCourseInfo);
        }
    }
```

在进行查询时，可以先把application.properties文件中最后一行，绑定表的配置注释掉。此时两张表的关联查询将要进行笛卡尔查询。

```
Actual SQL: m0 ::: select uci.* from user_course_info_1 uci ,user_1 u where uci.userid =
u.userid
Actual SQL: m0 ::: select uci.* from user_course_info_1 uci ,user_2 u where uci.userid =
u.userid
Actual SQL: m0 ::: select uci.* from user_course_info_2 uci ,user_1 u where uci.userid =
u.userid
Actual SQL: m0 ::: select uci.* from user_course_info_2 uci ,user_2 u where uci.userid =
u.userid
```

这种查询明显性能是非常低的，如果两张表的分片数更多，执行的SQL也会更多。而实际上，用户表和用户信息表，他们都是按照userid进行分片的，他们的分片规则是一致的。

这样，再把绑定关系的注释加上，此时查询，就会按照相同的userid分片进行查询。

```
Actual SQL: m0 ::: select uci.* from user_course_info_1 uci ,user_1 u where uci.userid =
u.userid
Actual SQL: m0 ::: select uci.* from user_course_info_2 uci ,user_2 u where uci.userid =
u.userid
```

在进行多表关联查询时，绑定表是一个非常重要的标准。

六、章节总结

这一章节带你大致体验了一下ShardingSphere大部分的核心功能。这些示例都很简单，你可能会觉得分库分表好像没有那么必要。但是请记住，我们的示例中只分了两个库，表最多也就分了两个。而在实际开发过程中，面对海量数据，同一个表往往要拆分成几十上百个分片。这时，像全路由查询、笛卡尔查询这些低效查询就会成为项目无法承受的痛。

另外还有一些平常用得不是太多的功能，比如 集中化配置、DistSQL，SQL Hint等就不再一一演示了。如果有兴趣，我还是建议你一定要去自己体验一下。因为在ShardingSphere中，这每一个机制都是一个很重要的扩展点。未来遇到真正棘手的问题，这些扩展点可能就是你解决问题的钥匙。ShardingSphere没有提供现成的功能，你可以自己扩展实现。但是如果你连这个方向都不清楚，那就有点可惜了。

然后，在之前的实战过程中，进行了很多的配置，甚至其中包含了很多莫名其妙的关键字。像COMPLEX，NANOID等等，还有props添加的很多莫名其妙的参数。甚至有很多配置，在IDEA中都是报红色的。**你有没有疑问，这些配置是怎么来的？**这么多的配置，如果光是靠记忆，是不可能记得住的。毕竟你不可能天天抱着官方文档开发。那要怎么去寻找这些配置呢？更何况谁也搞不准后续的版本这些配置方式会怎么变。从ShardingSphere 4.x版本到5.x版本，这些规则的配置方式做了非常大的调整。因此，你需要记住的是ShardingSphere解决这些分库分表问题的思路。

最后，还是强调一点，ShardingSphere提供的这些实现，都是一个重要的功能扩展点。不要尝试去记住他提供的这些具体的实现，而要积极尝试去理解每种机制背后的原理。在实际使用ShardingSphere分库分表时，只有能够灵活扩展的功能才是最好的功能。

> 有道云笔记链接： https://note.youdao.com/s/5Ae3bq0X