见招拆招：ShardingJDBC分库分表实战指南

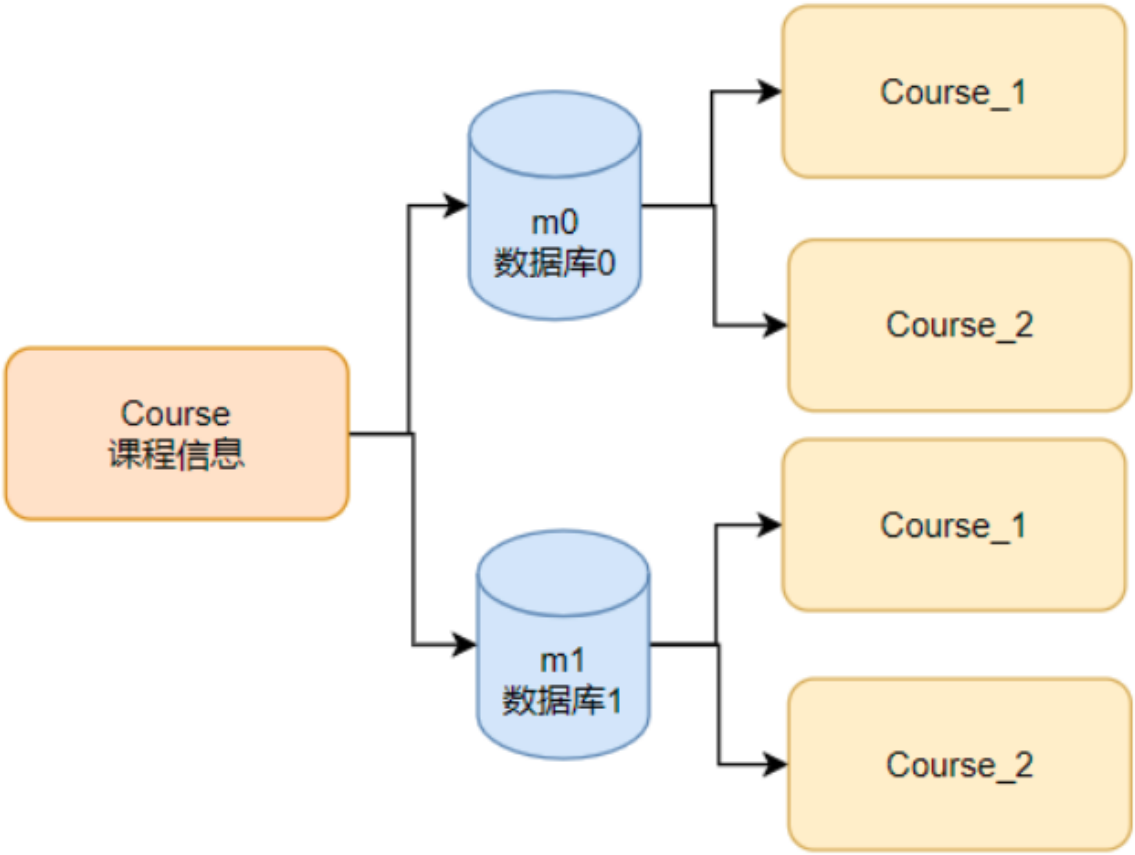
-- 楼兰

这一章将由浅入深，带你快速学会ShardingJDBC的各种神奇功能。

一、第一个分库分表的案例

为了让你对分库分表这个事情有一个直观的理解，我将快速带你实现一个简单的分库分表案例，为后续深入理解ShardingJDBC打下基础。

我们预备要将一批课程信息分别拆分到两个库，四个表中。



开发之前，需要提前准备一个MySQL数据库，并在其中创建Course表。Course表的建表语句如下：

```
CREATE TABLE course  (
  `cid` bigint(0) NOT NULL,
  `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;
```

1、快速搭建基础JDBC应用

接下来我们使用最常见的SpringBoot+MyBatis-plus快速搭建一个可以访问数据库的简单应用，以这个应用作为后续分库分表的基础。

step1: 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>
```

step2: 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> {
}
```

step3: 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);
    }
}
```

step4: 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://192.168.65.212:3306/test?serverTimezone=UTC
spring.datasource.druid.username=root
spring.datasource.druid.password=root
```

step5: 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ShardingJDBC快速实现分库分表

接下来，在之前的简单案例基础上，快速使用ShardingSphere实现Course表的分库分表功能。体验一下ShardingSphere是如何让分库分表这个事情变简单的。

step1: 在pom.xml中引入ShardingSphere

```
<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>
        <!-- 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>
        <!-- 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>
  <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>
    </dependencies>
```

> 注：ShardingSphere目前最新版本5.5.0提供的官方依赖项是
>
> ```
> <dependency>
>     <groupId>org.apache.shardingsphere</groupId>
>     <artifactId>shardingsphere-jdbc</artifactId>
>     <version>${latest.release.version}</version>
> </dependency>
> ```
>
> 这里并没有使用官方的这个依赖项，而是采用的SpringBoot集成ShardingSphere的一个依赖项，依赖的是5.2.1版本。原因有两个：
>
> 一是目前最新版本ShardingSphere与SpringBoot的集成，版本相对较慢，这在业务功能上倒没有造成很直接的影响，但是会造成对开发工作的支持没有那么明显。比如IDEA中对SpringBoot的配置文件没有提示。
>
> 二是经过我的测试，官方提供的这个依赖项，目前从5.2往上的版本，还有一些问题，尤其跟其他组件集成时，经常会出现不兼容的情况。
>
> 所以，为了教学方便，这次并没有追求目前最新的5.5.0版本，而是采用5.2.1版本。

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。表结构与course表是一致的。

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SpringBoot的配置文件application.properties，在里面添加以下配置信息。再次执行addCourse方法，添加课程信息，数据就会自动完成分库分表。

```
# 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://192.168.65.212:3306/shardingdb1?
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://192.168.65.212:3306/shardingdb2?
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}
#MOD分库策略
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_$->{cid%2+1}
#这种算法如果cid是严格递增的，就可以将数据均匀分到四个片。但是雪花算法并不是严格递增的。
#如果需要做到均匀分片，修改算法同时，还要修改雪花算法。把SNOWFLAKE换成MYSNOWFLAKE
#spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.props.algorithm-
expression=course_$->{((cid+1)%4).intdiv(2)+1}
```

> 配置信息有点复杂，刚开始看不懂没关系，后面会详细解读配置过程。
>
> 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。这个表达式中，$->{}部分为动态部分，大括号内的就是Groovy语句。
>
> 在Groovy语法中，两个点表示一个数据组的起点和终点。m$->${0..1}表示m0和m1两个字符串集合。course_$->{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。

接下来主要观察addcourse方法的执行结果。可以看到，十条课程信息，cid字段自动生成了一些ID，并且数据按照cid的奇偶，拆分到了m0.course_1和m1.course_2 两张表中。

并且在日志里可以看到实际的执行情况

```
[ INFO] ShardingSphere-SQL             :Logic SQL: INSERT INTO course  ( cname, user_id, cstatus )  VALUES (  ?, ?, ?  )
[ INFO] ShardingSphere-SQL             :SQLStatement: MySQLInsertStatement(super=InsertStatement(super=AbstractSQLStatement(para
[ INFO] ShardingSphere-SQL             :Actual SQL: m1 ::: INSERT INTO course_2  ( cname, user_id, cstatus , cid)  VALUES (?, ?,
Course{cid=null, cname='java', userId=1001, cstatus='1'}
[ INFO] ShardingSphere-SQL             :Logic SQL: INSERT INTO course  ( cname, user_id, cstatus )  VALUES (  ?, ?, ?  )
[ INFO] ShardingSphere-SQL             :SQLStatement: MySQLInsertStatement(super=InsertStatement(super=AbstractSQLStatement(para
[ INFO] ShardingSphere-SQL             :Actual SQL: m0 ::: INSERT INTO course_1  ( cname, user_id, cstatus , cid)  VALUES (?, ?,
Course{cid=null, cname='java', userId=1001, cstatus='1'}
```

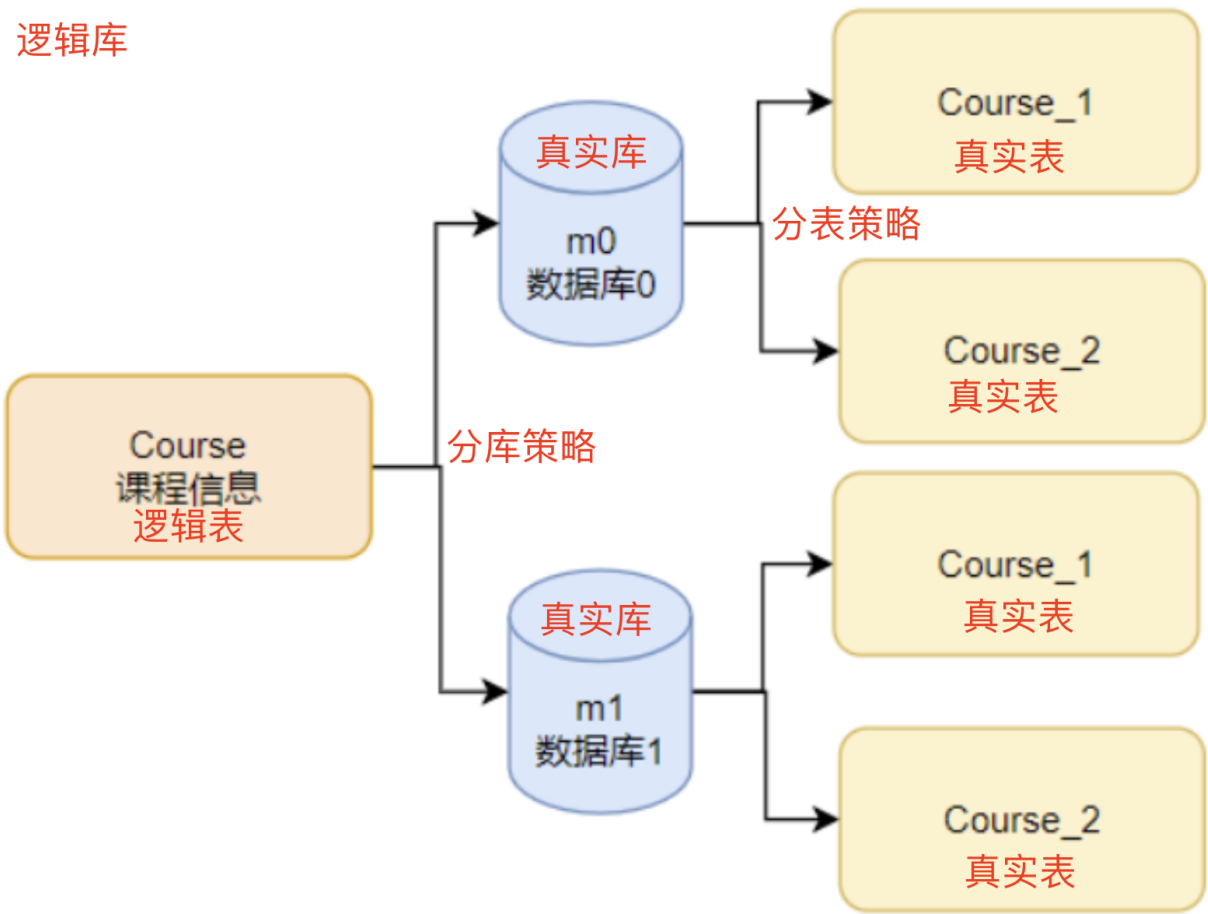
> 在这个示例中，十条course信息只能平均分配到两个表中，而无法均匀分到四张表中。如果希望改变这个结果，让course信息分到四个表中，需要从数据和算法两个方面思考。
>
> 如果cid是连续增长的，那么，将分片算法course_db_alg的计算表达式修改为 `course_$->{((cid+1)%4).intdiv(2)+1}`。理论上就可以了。
>
> 但是，事实上，snowflake雪花算法生成的ID并不是连续的，所以在这个示例中，依然是无法将数据分到四张表的。具体原因，会在后续详细分析分布式ID的课程中介绍。

二、理解分库分表的核心概念

看完热闹之后，接下来我们来看看门道。 分析一下ShardingSphere是如何完成分库分表这个事情的。

1、ShardingSphere分库分表的核心概念

在刚才的示例中，我们实际上操作了哪些数据呢?

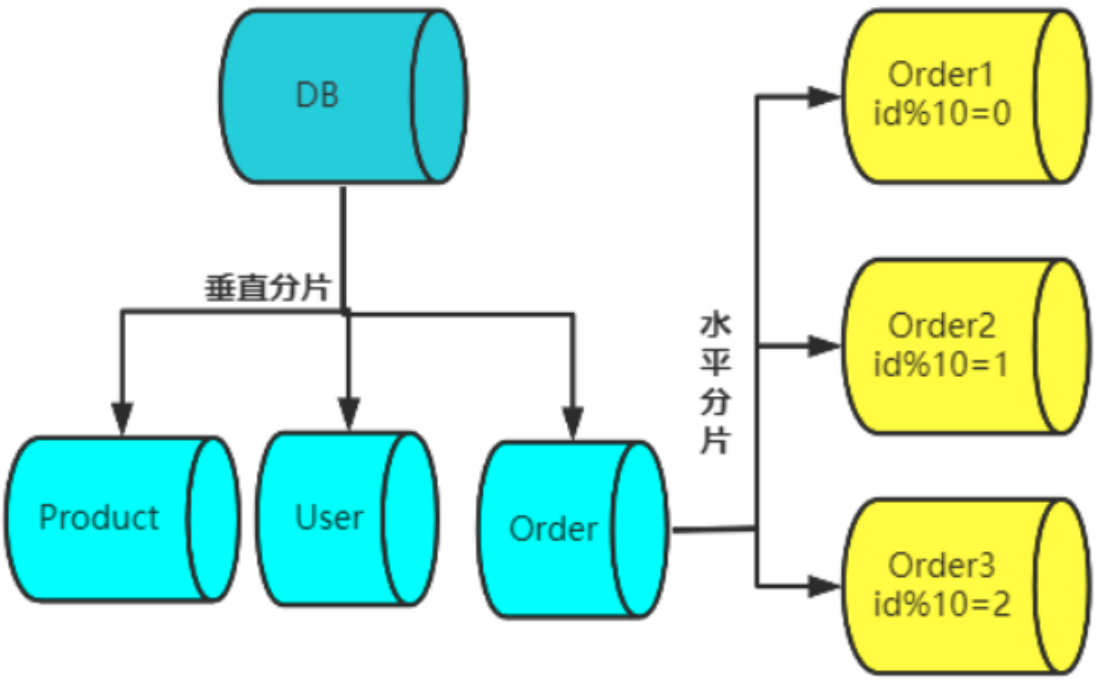


- **虚拟库**： ShardingSphere的核心就是提供一个具备分库分表功能的虚拟库，他是一个ShardingSphereDatasource实例。应用程序只需要像操作单数据源一样访问这个ShardingSphereDatasource即可。示例中，MyBatis框架并没有特殊指定DataSource，就是使用的ShardingSphere的DataSource数据源。
- **真实库**： 实际保存数据的数据库。这些数据库都被包含在ShardingSphereDatasource实例当中，由ShardingSphere决定未来需要使用哪个真实库。示例中，m0和m1就是两个真实库。
- **逻辑表**： 应用程序直接操作的逻辑表。 示例中操作的course表就是一个逻辑表，并不需要在数据库中真实存在。
- **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。示例中course_1和course_2就是真实表。
- **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。因此，在做分库分表时，通常都会独立出一个生成分布式ID的主键生成算法。示例中使用的SNOWFLAKE雪花算法就是一种很常见的主键生成算法。
- **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。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。示例当中对cid字段取模，就是一种简单的分片算法。 如果ShardingSphere匹配不到合适的分片策略，那就只能进行全分片路由，这是效率最差的一种实现方式。

强烈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。

2、垂直分片和水平分片

这也是设计分库分表方案时经常会提到的概念，所以也一并做一下梳理。当我们设计分库分表方案时，通常有两种拆分数据的维度。一是按照业务划分的维度，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。这种方案称为垂直分库。二是按照数据分布的维度，将原本存在同一张表当中的数据，拆分到多张子表当中。每个子表只存储一部分数据。这样可以介绍每一张表的数据量，提升查询效率。这种方案称为水平分表。



通常我们讲的分库分表，主要是指水平分片，因为这样才能减少数据量，从根本上解决数据量过大带来的存储和查询的问题。但是，这也并不意味着垂直分片方案就不重要。

三、ShardingJDBC常见数据分片策略实战

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、INLINE简单分片

INLINE简单分片主要用来处理针对分片建的=和 in 这样的查询操作。在这些操作中，可以拿到分片键的精确值。例如对下面这样的操作：

```
    /**
     * 针对分片键进行精确查询，都可以使用表达式控制
     */
    @Test
    public void queryCourse() {
        QueryWrapper<Course> wrapper = new QueryWrapper<Course>();
        wrapper.eq("cid",924770131651854337L);
//        wrapper.in("cid",901136075815124993L,901136075903205377L,901136075966119937L,5L);
        //带上排序条件不影响分片逻辑
//        wrapper.orderByDesc("user_id");
        List<Course> courses = courseMapper.selectList(wrapper);
        courses.forEach(course -> System.out.println(course));
    }
```

对于这样的操作，拿到分片键的精确值后，都可以通过表达式计算出可能的真实库以及真实表。而ShardingJDBC就会将逻辑SQL转化成对应的真实SQL，并路由到真实库中去执行。

这里有几个有趣的问题：

第一个是，ShardingJDBC关注的是过滤数据的关键查询条件中是否包含了分片键，而并只是简单关注附加的条件。 例如在SQL语句后面加上order by user_id，并不会影响ShardingJDBC的处理过程。而如果查询条件中不包含分片建，那么ShardingJDBC就只能根据actual-nodes，到所有的真实表和真实库中查询。这也就是全分片路由。

对于全分片路由，ShardingJDBC做了一定的优化。比如通过Union将同一库的多条语句结合起来，这样可以减少与数据库的交互次数。

```
[ INFO] ShardingSphere-SQL             :Logic SQL: SELECT  cid,cname,user_id,cstatus  FROM course
[ INFO] 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
[ INFO] 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
```

但是，在真实项目中，这种全分片路由是一定要尽力避免的。因为在真实项目中，你要面对的，就不是示例中少数的几个分片了，通常都是几十个甚至上百个分片。在这样大数据情况下，进行全分片路由，效率是非常低的。

第二个是，ShardingJDBC其实只负责改写以及路由SQL，至于有没有数据，他就无法关心了。例如，在之前的示例中，我们提出了将分表规则改写为`course_$->{((cid+1)%4).intdiv(2)+1}`，就能在cid连续递增的情况下，保证数据均匀分布。那么在此时，你就可以尝试去修改一下分表规则，然后查询cid in (1L,2L,3L,4L)，然后去分析一下ShardingJDBC会如何执行。

2、STANDARD标准分片

应用当中我们对于主键信息通常不只是进行精确查询，还需要进行范围查询。这时就需要一种能够同时支持精确查询和范围查询的算法出厂了。这就是STANDARD标准分片。 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，如果不修改分片算法，直接执行。由于ShardingSphere无法根据配置的表达式计算出可能的分片情况，在执行时就会抛出一个异常

```
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 false, inline sharding algorithm can not tackle with range query
### 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 algorithm can not tackle with range query
; uncategorized SQLException; SQL state [0A000]; error code [30001]; Unsupported SQL operation: Since the property of
 `allow-range-query-with-inline-sharding` is false, inline sharding algorithm can not tackle with range query; nested exception
  is java.sql.SQLException: Unsupported SQL operation: Since the property of `allow-range-query-with-inline-sharding` is false,
  inline sharding algorithm can not tackle with range query
```

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。只不过，很显然，这种范围查询要匹配的精确值太多了，不可能通过一个简单的表达式来处理。那要怎么解决呢？记住这个问题，在后续章节会带你解决。

3、COMPLEX_INLINE复杂分片

除了针对单个分片键的查询，我们还有可能需要针对多个属性进行组合查询。例如

```
    @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);
    }
```

简单执行一下，当然是可以执行的。

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

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。

例如，我要进行下面这样的查询，包含对user_id的范围查询。

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

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，要求查询的范围必须包括1001L这个值。

如果连这个简单的规则都无法满足，那么这个SQL语句明显不可能有数据。对于这样的SQL，当然是希望他不要去数据库里执行了。因为这样明显是浪费性能。那么这样的需求要怎么实现呢?

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.MyComplexAlgorithm
```

这时，再去执行查询方法，就会得到这样的异常信息。

```
org.springframework.jdbc.UncategorizedSQLException:
### Error querying database.  Cause: java.sql.SQLException: Can not support operation `error range query param` with sharding table `course`.
### 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 IN (?,?) AND user_id BETWEEN ? AND ?)
### Cause: java.sql.SQLException: Can not support operation `error range query param` with sharding table `course`.
; uncategorized SQLException; SQL state [0A000]; error code [20040]; Can not support operation `error range query param` with sharding table `course`.;
```

在我们当前设定的业务场景下，这其实不是出问题了，而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。

> 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

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、常用分片算法总结

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。

四、ShardingJDBC数据加密功能实战

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。最典型的，比如对用户的密码，通常都是需要加密存储的。使用ShardingSphere就可以用应用无感知的方式，快速实现数据加密。并且可以灵活切换多种内置的加密算法。

下面新建一张user用户表，来实现下数据加密的功能：

```
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;
```

> 这个实例表后续还可以用来测试字符串型主键的主键生成以及数据分片等功能，因此，建议在shardingdb1和shardingdb2两个数据库中也同样创建user_1和user_2两个分片表。

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> {
}
```

SpringBoot中配置逻辑表user的加密算法

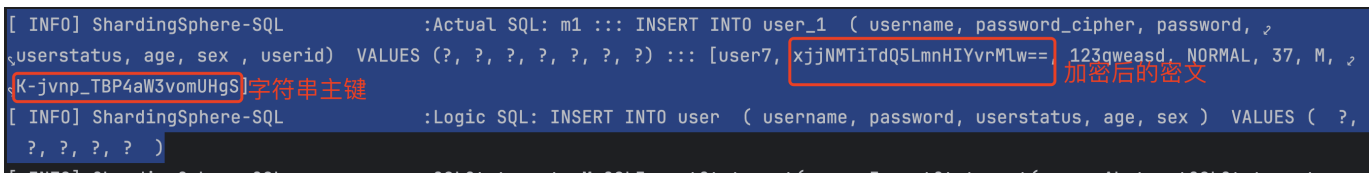
```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# -----------------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://192.168.65.212:3306/shardingdb1?
serverTimezone=Asia/Shanghai
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://192.168.65.212:3306/shardingdb2?
serverTimezone=Asia/Shanghai
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
```

单元测试案例

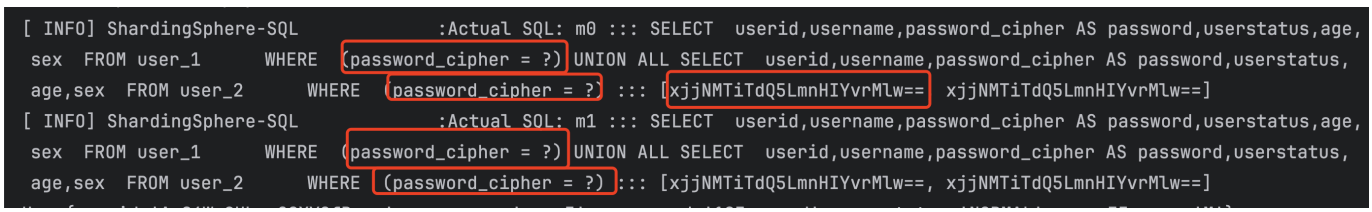
```
    @Test
    public void addUser(){
        for (int i = 0; i < 10; i++) {
            User user = new User();
//            user.setUserid();
            user.setUsername("user"+i);
            user.setPassword("123qweasd");
            user.setUserstatus("NORMAL");
            user.setAge(30+i);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

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

在插入时，就会在password_cipher字段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

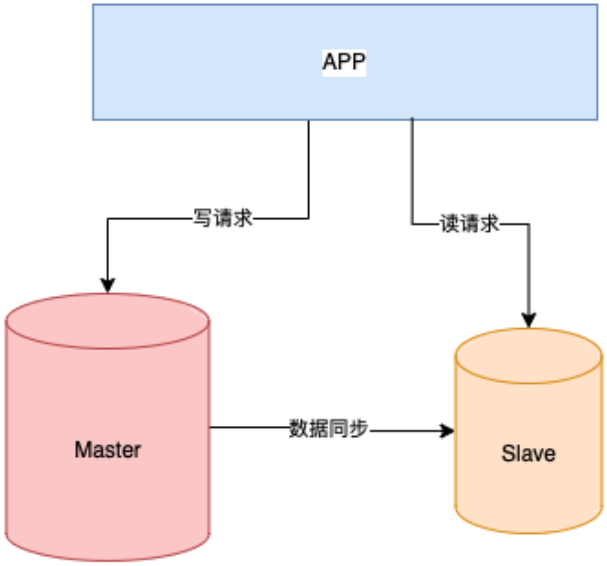
```
    @Test
    public void queryUser() {
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("password","123qweasd");
        List<User> users = userMapper.selectList(queryWrapper);
        for(User user : users){
            System.out.println(user);
        }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



关于各种不同的加密算法，可以自己尝试一下。

五、基于ShardingJDBC实现读写分离

读写分离方案是应用中常用的一种保护数据库的方案。



通过将读写请求分发到不同的数据库，从而减少主库的客户端请求。

读写分离方案通常需要分两个层面配合解决。在数据层面，需要将Master的数据实时同步到slave。这一部分通常是通过一些第三方的工具去执行。例如Canal框架，或者MySQL自己提供的主从同步方案等。

而在应用层面，要做的就是将读请求和写请求分发到不同的数据库中。这本质也是一种数据路由的功能。用ShardingSphere来实现就非常简单。只需要配置一个readwrite-splitting的分片规则即可。

例如，针对之前建立的user表，我们可以快速配置一个读写分离的示例：

```
# 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。

六、广播表与绑定表实战

广播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 dict  (
  `dictId` bigint NOT NULL,
  `dictKey` varchar(32) NULL,
  `dictVal` varchar(32) NULL,
  PRIMARY KEY (`dictId`)
);
```

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。

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 NOT NULL,
  `userid` varchar(32) NULL,
  `courseid` bigint NULL,
  PRIMARY KEY (`infoid`)
);
```

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。

七、分片审计

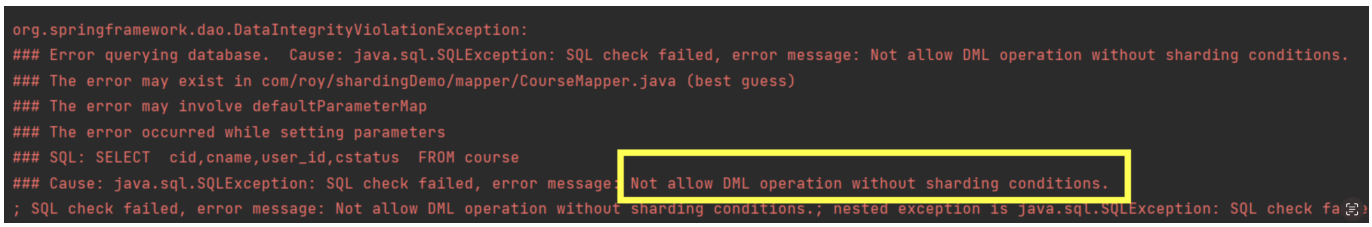
分片审计功能是针对数据库分片场景下对执行的 SQL 语句进行审计操作。分片审计既可以进行拦截操作，拦截系统配置的非法 SQL 语句，也可以是对 SQL 语句进行统计操作。

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。

八、章节总结

这一章节主要是见招拆招，在我给大家精心预设的各种业务场景下，实战体验了ShardingSphere大部分的核心功能。重点其实有两个。

一是ShardingJDBC提供的各种花里胡哨的功能。这些是开发过程中最直接有力的工具，所以，一定要自己多尝试，开始熟悉起来。这自然不必多说。但是要强调的是，分库分表的问题，五花八门，层出不穷。而且ShardingSphere目前版本正处在快速迭代的阶段，因此他本身的功能，尤其是与其他框架集成的功能变动会非常大。同样的配置，或许换换其中maven组件的版本就会冒出各种BUG。因此，建议你一定要尝试从头开始搭建整个示例项目。

二是关于分库分表的那些核心概念。这些概念不起眼，但是很重要。只有把这些抽象的概念理解透了，你才可能真正设计出一个可落地的分库分表方案。之前有很多学员，在工作过程中遇到了分库分表的问题，想要跟老师请教。但是，解释半天，老师都很难融入到他的业务场景中。这其中的问题，就是没有理解这些核心概念在项目中是如何落地的，因此对整个分库分表的方案描述不清。没有完整的方案设计，只关注某一个报错信息，那么自然无法形成有效的沟通。这还只是提问，沟通不清也问题不大。但是如果是面试，或者是在项目内跟别人分享，后果可想而知。所以，再次强调，这些核心概念很重要。

然后，在这一系列的实战过程中，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。更何况谁也搞不准后续的版本这些配置方式会怎么变。如果你之前接触过ShardingSphere4.x的版本，那么一定对这些头疼的配置大感头疼。那要怎么去理解这些配置呢？你最需要记住的，是ShardingSphere解决这些分库分表问题的思路。然后再结合后面的课程，理解了ShardingSphere的各种扩展机制，你才能真正把ShardingSpere用得随心所欲。

> 有道云笔记：【有道云笔记】二、见招拆招：ShardingJDBC分库分表实战指南.md
> https://note.youdao.com/s/FAXGvTcb