课堂主题

Sharding JDBC架构和核心概念、Sharding JDBC安装和核心组件、Sharding JDBC分片策略和读写分离

课堂目标

理解Sharding JDBC架构和核心概念（数据分片、SQL、分片策略、分片算法、配置）

能够在项目中引入Sharding JDBC开源组件

了解Sharding JDBC核心组件（解析引擎、路由引擎、改写引擎、执行引擎、归并引擎）

掌握Sharding JDBC分片策略

掌握Sharding JDBC读写分离

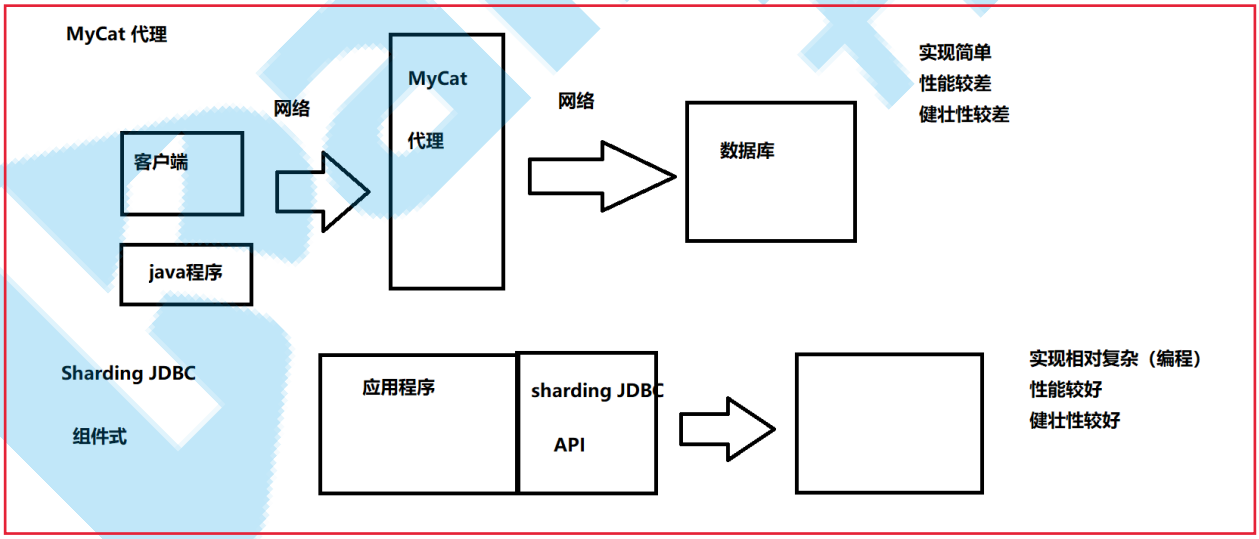
能够在项目中配置数据分片、读写分离、广播表、绑定表

什么是Sharding JDBC

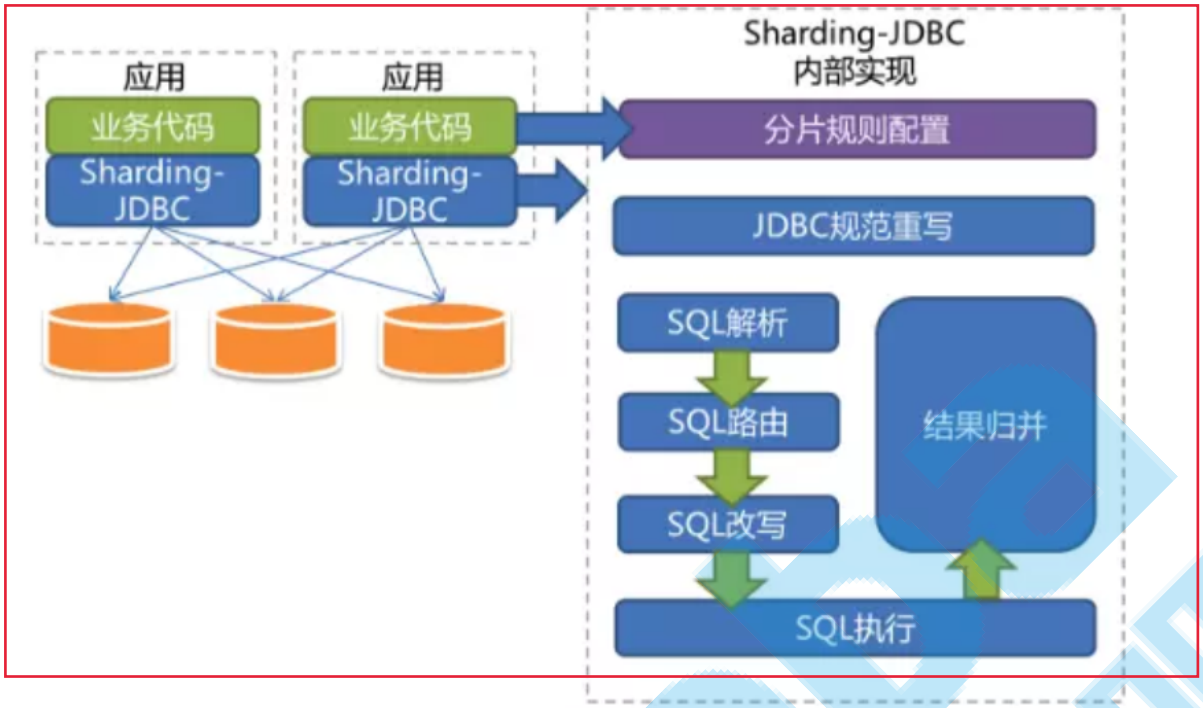
官方网站：http://shardingsphere.apache.org/index_zh.html

Apache ShardingSphere(Incubator) 是一套开源的分布式数据库中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（规划中）这3款相互独立，却又能够混合部署配合使用的产品组成。

两种实现方式的对比



Sharding JDBC架构

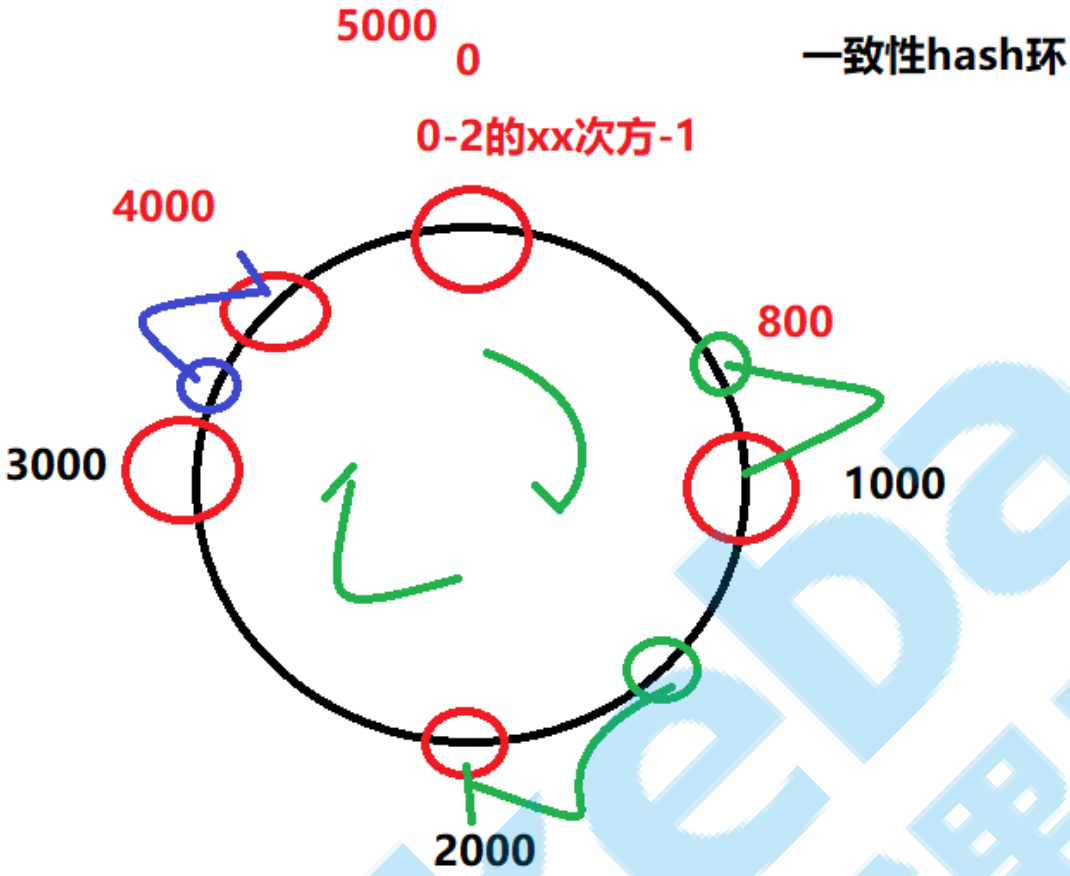


Sharding JDBC核心概念

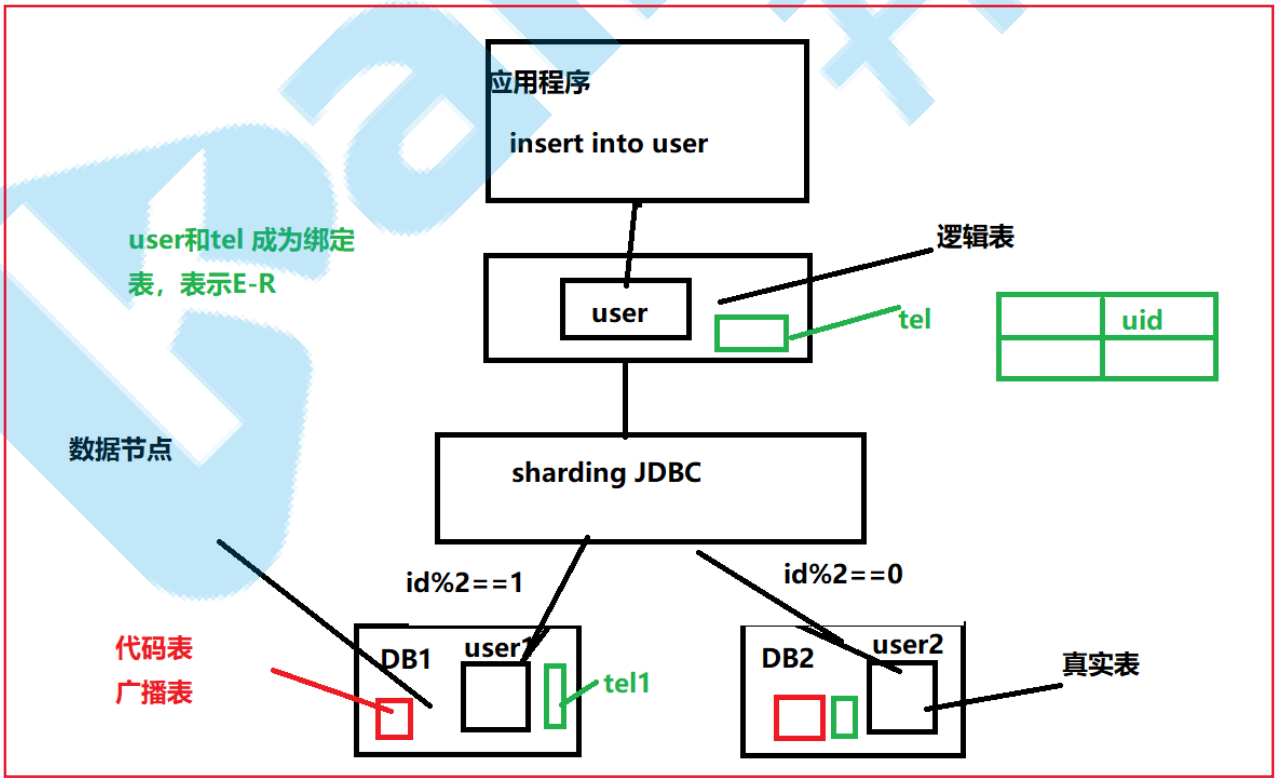
数据分片

数据分片分为垂直分片和水平分片。

一致性hash环



SQL



逻辑表

水平拆分的数据库（表）的相同逻辑和数据结构表的总称。

真实表

在分片的数据库中真实存在的物理表。

数据节点

数据分片的最小单元。由数据源名称和数据表组成。

绑定表

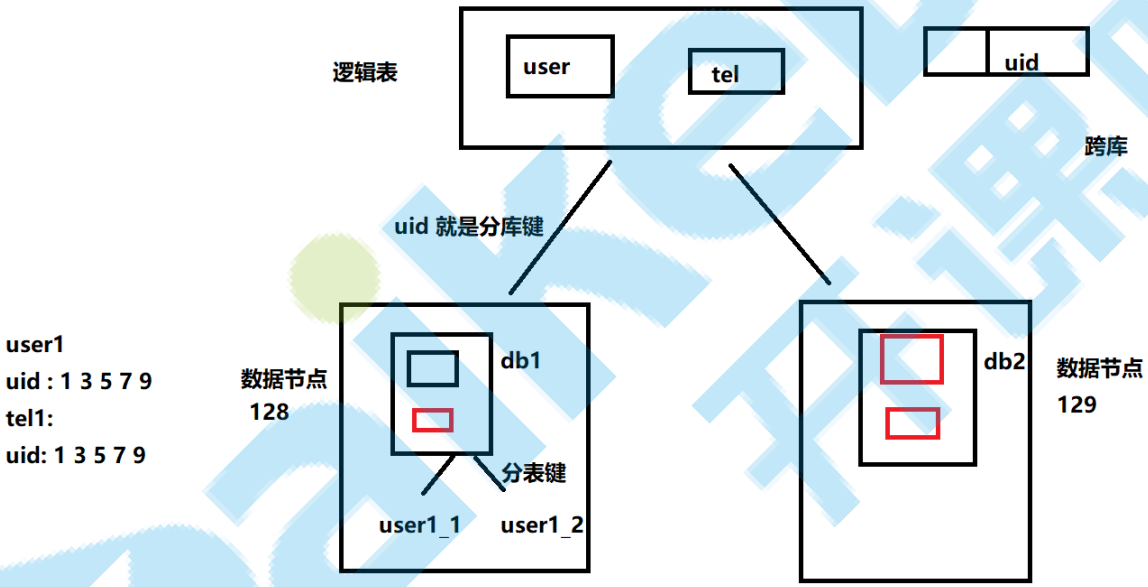
指分片规则一致的主表和子表。例如：`t_order`表和`t_order_item`表，均按照`order_id`分片，则此两张表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。

广播表

指所有的分片数据源中都存在的表，表结构和表中的数据在每个数据库中均完全一致。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。

分片策略

包含分片键和分片算法。分片键是用于分片的数据库字段，是将数据库(表)水平拆分的关键字段。



分片算法

精确分片算法、范围分片算法、复合分片算法、Hint分片算法

精确分片算法（PreciseShardingAlgorithm）用于处理使用单一键作为分片键的=与IN进行分片的场景。需要配合StandardShardingStrategy使用。

```
public class DemoTableShardingAlgorithm implements
PreciseShardingAlgorithm<Long> {
@Override
public String doSharding(Collection<String> collection,
PreciseShardingValue<Long> preciseShardingValue) {
    for (String each : collection) {
        if
(each.endsWith(Long.parseLong(preciseShardingValue.getValue().toString()) %
2+"")) {
            return each;
        }
    }
    throw new IllegalArgumentException();
}

}
```

范围分片算法（RangeShardingAlgorithm）用于处理使用单一键作为分片键的BETWEEN AND进行分片的场景。需要配合StandardShardingStrategy使用。

```
public class MyRangeShardingAlgorithm implements RangeShardingAlgorithm<Long> {
    @Override
    public Collection<String> doSharding(Collection<String> collection,
RangeShardingValue<Long> rangeShardingValue) {
        log.info("Range collection:" + JSON.toJSONString(collection) +
",rangeShardingValue:" + JSON.toJSONString(rangeShardingValue));
        Collection<String> collect = new ArrayList<>();
        Range<Long> valueRange = rangeShardingValue.getValueRange();
        for (Long i = valueRange.lowerEndpoint(); i <=
valueRange.upperEndpoint(); i++) {
            for (String each : collection) {
                if (each.endsWith(i % collection.size() + "")) {
                    collect.add(each);
                }
            }
        }
        return collect;
    }
}
```

复合分片算法（ComplexKeysShardingAlgorithm）用于处理使用多键作为分片键进行分片的场景，包含多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。需要配合ComplexShardingStrategy使用。

```
public class MyComplexShardingAlgorithm implements ComplexKeysShardingAlgorithm
{
    @Override
    public Collection<String> doSharding(Collection<String> collection,
Collection<ShardingValue> shardingValues) {
        log.info("collection:" + JSON.toJSONString(collection) +
",shardingValues:" + JSON.toJSONString(shardingValues));

        Collection<Long> orderIdValues = getShardingValue(shardingValues,
"order_id");
```

```
        Collection<Long> userIdValues = getShardingValue(shardingValues,
"user_id");
        List<String> shardingSuffix = new ArrayList<>();
        /**例如：根据user_id + order_id 双分片键来进行分表*/
        //Set<List<Integer>> valueResult = Sets.cartesianProduct(userIdValues,
orderIdValues);
        for (Long userIdVal : userIdValues) {
            for (Long orderIdVal : orderIdValues) {
                String suffix = userIdVal % 2 + "_" + orderIdVal % 2;
                collection.forEach(x -> {
                    if (x.endsWith(suffix)) {
                        shardingSuffix.add(x);
                    }
                });
            }
        }

        return shardingSuffix;
    }

    private Collection<Long> getShardingValue(Collection<ShardingValue>
shardingValues, final String key) {
        Collection<Long> valueSet = new ArrayList<>();
        Iterator<ShardingValue> iterator = shardingValues.iterator();
        while (iterator.hasNext()) {
            ShardingValue next = iterator.next();
            if (next instanceof ListShardingValue) {
                ListShardingValue value = (ListShardingValue) next;
                /**例如：根据user_id + order_id 双分片键来进行分表*/
                if (value.getColumnName().equals(key)) {
                    return value.getValues();
                }
            }
        }
        return valueSet;
    }
}
```

Hint分片算法（HintShardingAlgorithm）用于处理使用Hint行分片的场景。需要配合HintShardingStrategy使用。

分片策略

标准分片策略、复合分片策略、行表达式分片策略、Hint分片策略

标准分片策略（StandardShardingStrategy）提供对SQL语句中的=, IN和BETWEEN AND的分片操作支持。StandardShardingStrategy只支持单分片键，提供PreciseShardingAlgorithm和RangeShardingAlgorithm两个分片算法。PreciseShardingAlgorithm是必选的，用于处理=和IN的分片。RangeShardingAlgorithm是可选的，用于处理BETWEEN AND分片，如果不配置RangeShardingAlgorithm，SQL中的BETWEEN AND将按照全库路由处理。

复合分片策略（ComplexShardingStrategy）提供对SQL语句中的=, IN和BETWEEN AND的分片操作支持。ComplexShardingStrategy支持多分片键，由于多分片键之间的关系复杂，因此并未进行过多的封装，而是直接将分片键值组合以及分片操作符透传至分片算法，完全由应用开发者实现，提供最大的灵活度。

行表达式分片策略（InlineShardingStrategy）使用Groovy的表达式，提供对SQL语句中的=和IN的分片操作支持，只支持单分片键。对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的Java代码开发。

行表达式的使用非常直观，只需要在配置中使用 `${ expression }` 或 `$->{ expression }` 标识行表达式即可。 目前支持数据节点和分片算法这两个部分的配置。行表达式的内容使用的是Groovy的语法，Groovy能够支持的所有操作，行表达式均能够支持。

Hint分片策略（HintShardingStrategy）通过Hint而非SQL解析的方式分片的策略。对于分片字段非SQL决定，而由其他外置条件决定的场景，可使用SQL Hint灵活的注入分片字段。

配置

分片规则：分片规则配置的总入口。包含数据源配置、表配置、绑定表配置以及读写分离配置等。

数据源配置：真实数据源列表，结合数据库连接池使用

表配置:逻辑表名称、数据节点与分表规则的配置。

数据节点配置:用于配置逻辑表与真实表的映射关系。可分为均匀分布和自定义分布两种形式

- 数据源分片策略

对应于DatabaseShardingStrategy。用于配置数据被分配的目标数据源。

- 表分片策略

对应于TableShardingStrategy。用于配置数据被分配的目标表，该目标表存在与该数据的目标数据源内。故表分片策略是依赖与数据源分片策略的结果的。

自增主键生成策略,通过在客户端生成自增主键替换以数据库原生自增主键的方式，做到分布式主键无重复。(UUID和雪花算法)

Sharding JDBC对多数据库的支持

Sharding JDBC安装

引入Maven依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-core</artifactId>
    <version>3.0.0</version>
</dependency>
```

调用API编程实现，最新的是shardingJDBC4.0RC

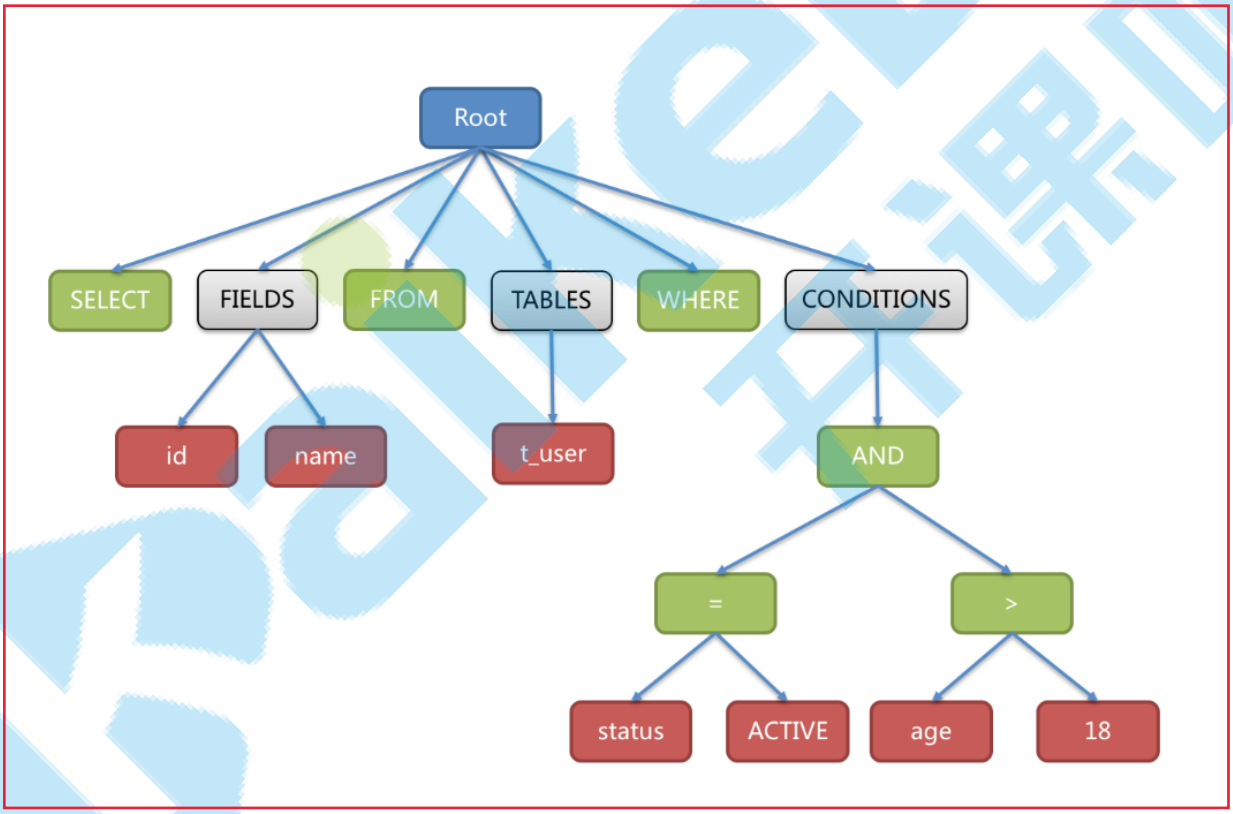
Sharding JDBC核心组件

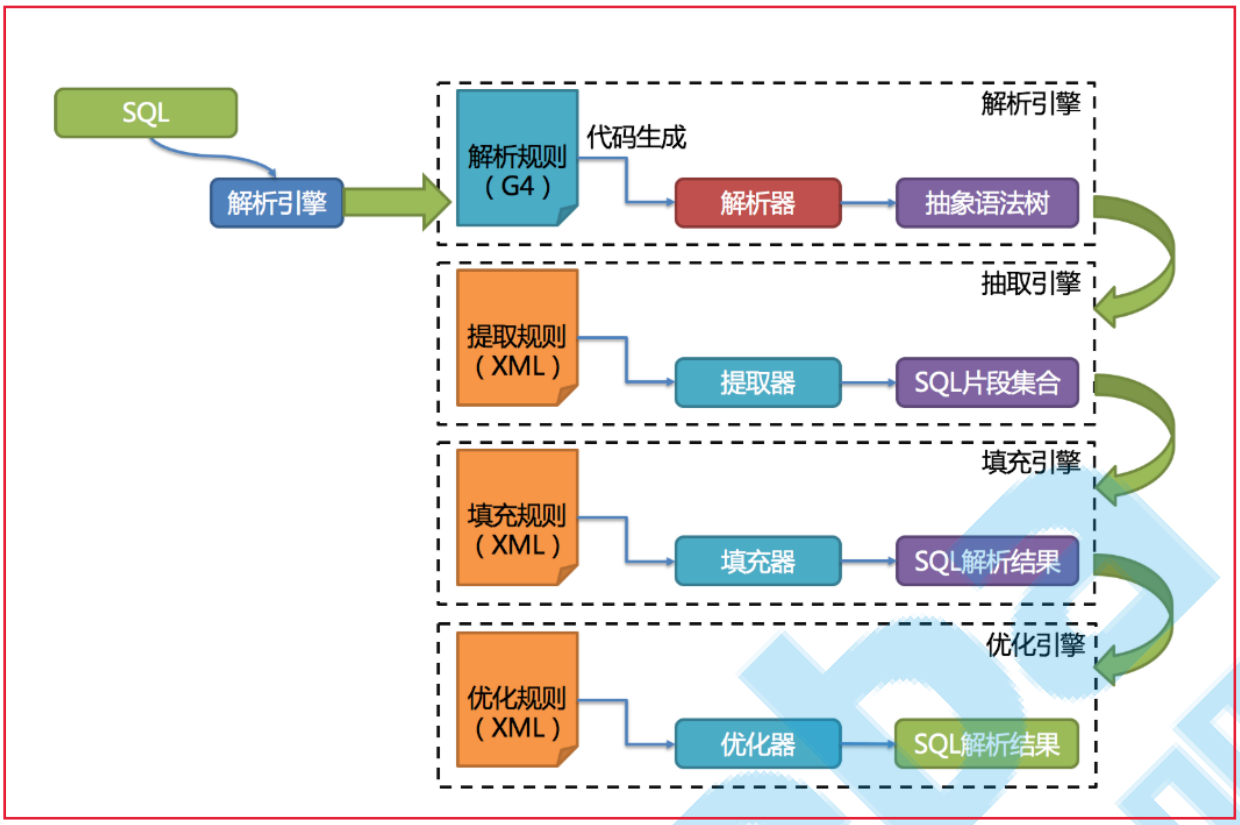
解析引擎

词法解析

语法解析

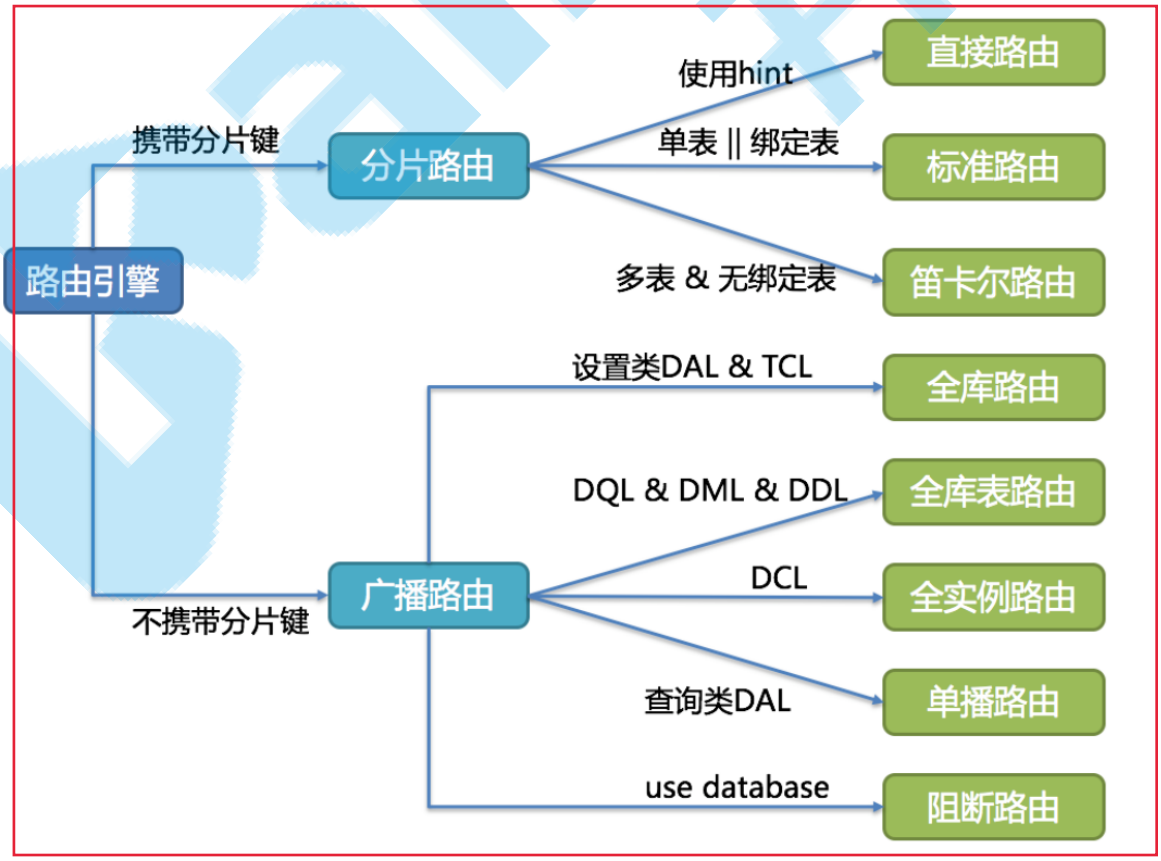
形成语法树





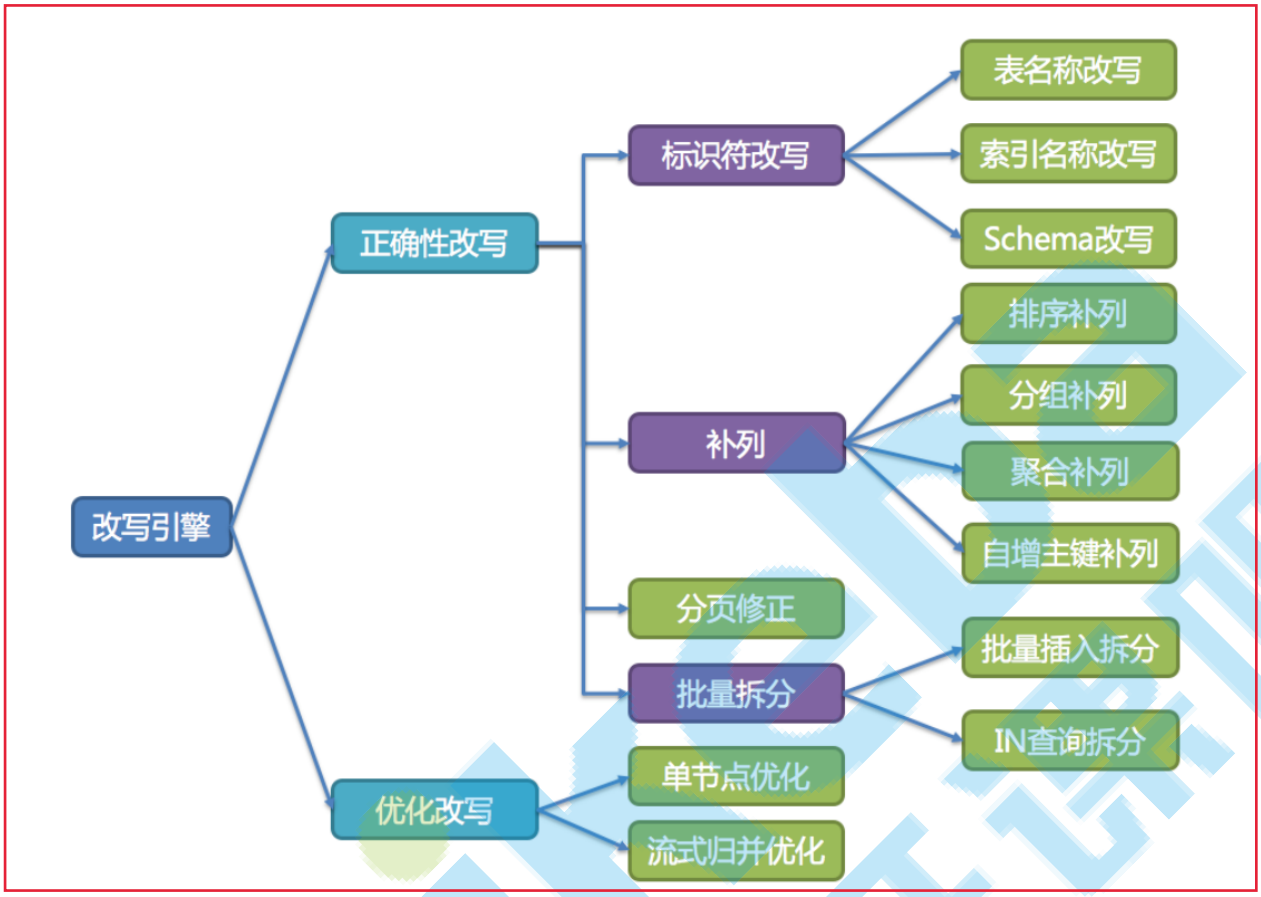
路由引擎

标准路由是ShardingSphere最为推荐使用的分片方式，它的适用范围是不包含关联查询或仅包含绑定表之间关联查询的SQL。 当分片运算符是等于号时，路由结果将落入单库（表），当分片运算符是BETWEEN或IN时，则路由结果不一定落入唯一的库（表），因此一条逻辑SQL最终可能被拆分为多条用于执行的真实SQL。



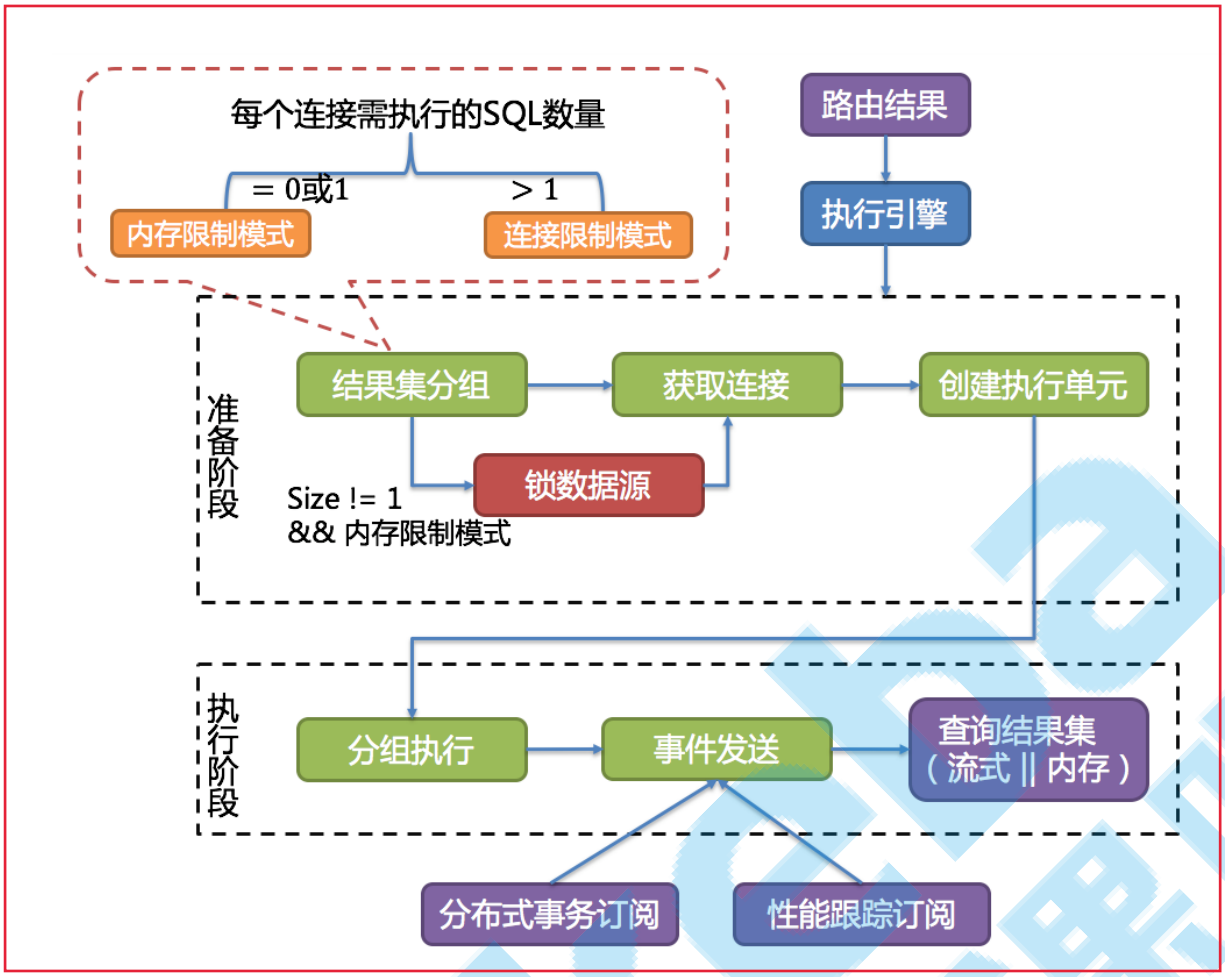
改写引擎

工程师面向逻辑库与逻辑表书写的SQL，并不能够直接在真实的数据库中执行，SQL改写用于将逻辑SQL改写为在真实数据库中可以正确执行的SQL。 它包括正确性改写和优化改写两部分。



执行引擎

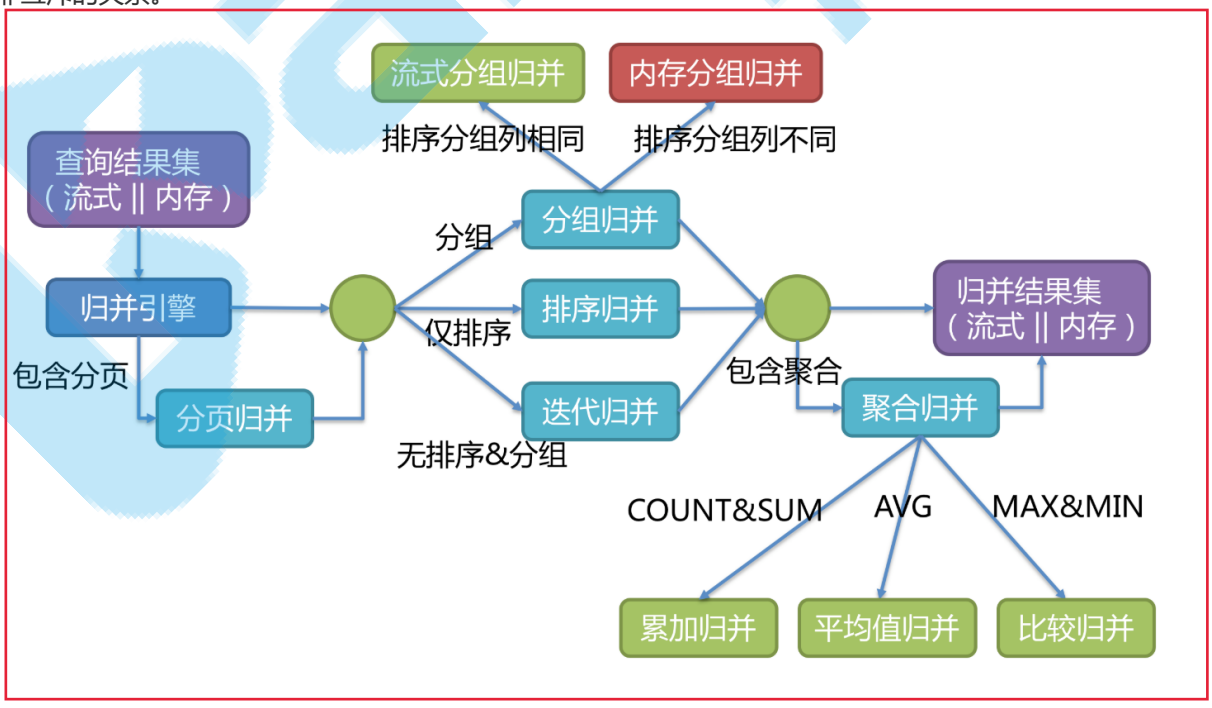
ShardingSphere采用一套自动化的执行引擎，负责将路由和改写完成之后的真实SQL安全且高效发送到底层数据源执行。执行引擎的目标是自动化的平衡资源控制与执行效率。执行引擎分为准备和执行两个阶段。

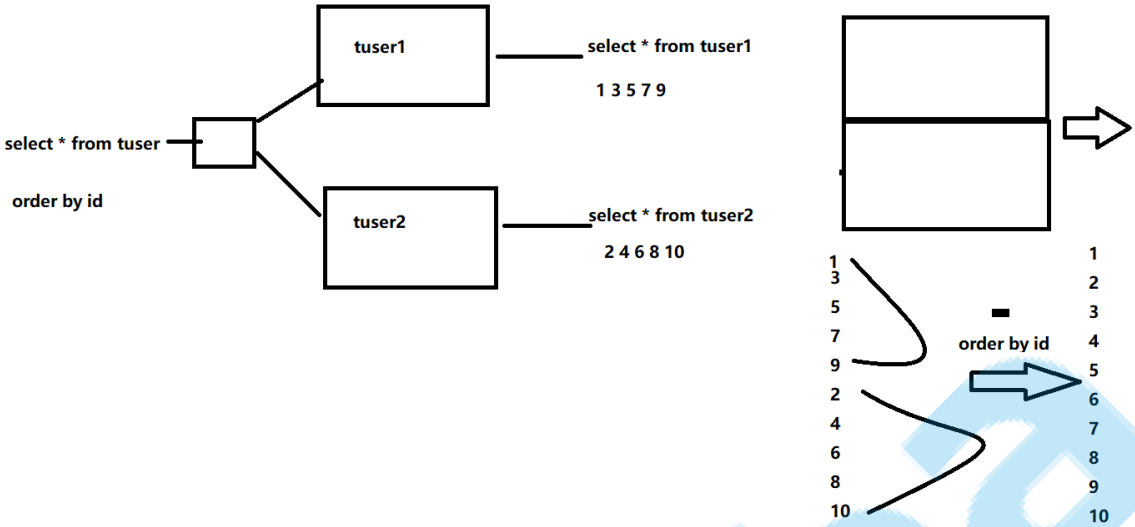


归并引擎

将从各个数据节点获取的多数据结果集，组合成为一个结果集并正确的返回至请求客户端，称为结果归并。

ShardingSphere支持的结果归并从功能上分为遍历、排序、分组、分页和聚合5种类型，它们是组合而非互斥的关系。





测试Demo

Java调用sharding JDBC

表结构

```
DROP TABLE IF EXISTS `t_order_0`;
CREATE TABLE `t_order_0` (
  `oid` int(11) NOT NULL,
  `uid` int(11) DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`oid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of t_order_0
-- ----------------------------
```

1、pom.xml

```
        <dependency>
            <groupId>io.shardingsphere</groupId>
            <artifactId>sharding-jdbc-core</artifactId>
            <version>3.0.0</version>
        </dependency>

        <!-- mysql 数据库驱动. -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>
        <!-- 数据源 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.0.26</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.6</version>
        </dependency>
```

2、sharding.java

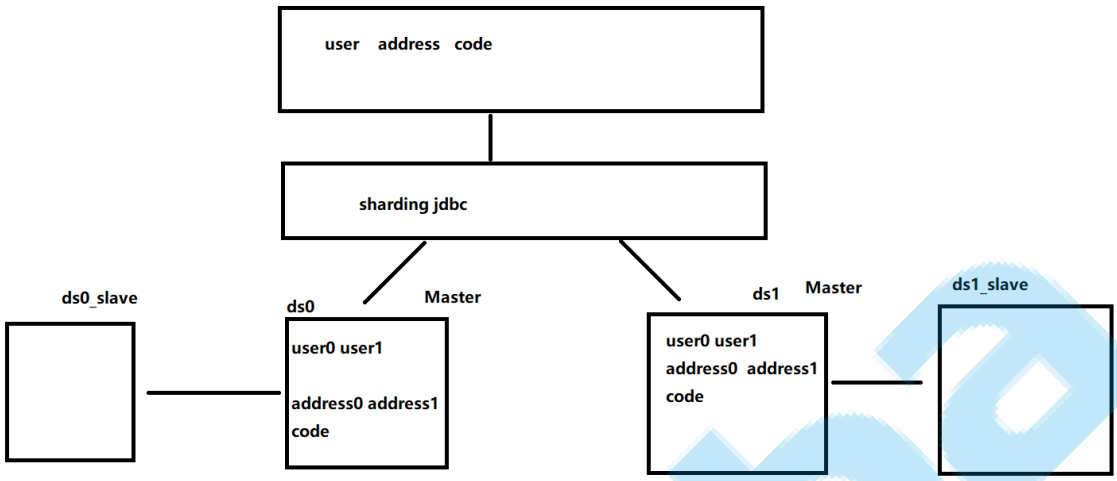
```
Map<String, DataSource> map=new HashMap<>();
        map.put("kkb_ds_0",
createDataSource("root","root","jdbc:mysql://127.0.0.1:3307/kkb_ds_0"));
        map.put("kkb_ds_1",
createDataSource("root","root","jdbc:mysql://127.0.0.1:3308/kkb_ds_1"));

        ShardingRuleConfiguration config=new ShardingRuleConfiguration();

        // 配置Order表规则
        TableRuleConfiguration orderTableRuleConfig = new
TableRuleConfiguration();
        orderTableRuleConfig.setLogicTable("t_order");//设置逻辑表.

orderTableRuleConfig.setActualDataNodes("kkb_ds_${0..1}.t_order_${0..1}");//设置
实际数据节点.
        orderTableRuleConfig.setKeyGeneratorColumnName("oid");//设置主键列名称.

        // 配置Order表规则：配置分库 + 分表策略(这个也可以在ShardingRuleConfiguration进
行统一设置)
        orderTableRuleConfig.setDatabaseShardingStrategyConfig(new
InlineShardingStrategyConfiguration("uid", "kkb_ds_${uid % 2}"));
        orderTableRuleConfig.setTableShardingStrategyConfig(new
InlineShardingStrategyConfiguration("oid", "t_order_${oid % 2}"));
        config.getTableRuleConfigs().add(orderTableRuleConfig);

        try {
            DataSource ds=ShardingDataSourceFactory.createDataSource(map,
config, new HashMap(), new Properties());

            for(int i=1;i<=10;i++) {
                String sql="insert into t_order(uid,name) values(?,?)";
                execute(ds,sql,i,i+"aaa");
            }
            System.out.println("数据插入完成。。。");

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

项目案例

Springboot2+mybatisplus+shardingJDBC3 实现数据分片、读写分离、广播表、主键自增、绑定表

user address E-R

code 全局表



表结构

```
--用户表
CREATE TABLE `user_0` (
  `uid` int(11) NOT NULL DEFAULT '0',
  `name` varchar(255) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`uid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
--地址表
CREATE TABLE `address_0` (
  `aid` int(11) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `uid` int(11) DEFAULT NULL,
  PRIMARY KEY (`aid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
--字典表
CREATE TABLE `code` (
  `id` bigint(20) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

分库分表

```
# 数据源 ds0,ds1
sharding.jdbc.datasource.names=ds0,ds1
# 第一个数据库
sharding.jdbc.datasource.ds0.type=com.zaxxer.hikari.HikariDataSource
sharding.jdbc.datasource.ds0.driver-class-name=com.mysql.jdbc.Driver
sharding.jdbc.datasource.ds0.jdbc-url=jdbc:mysql://192.168.56.101:3306/ds0?
characterEncoding=utf-8
sharding.jdbc.datasource.ds0.username=root
sharding.jdbc.datasource.ds0.password=root
```

```
# 第二个数据库
sharding.jdbc.datasource.ds1.type=com.zaxxer.hikari.HikariDataSource
sharding.jdbc.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
sharding.jdbc.datasource.ds1.jdbc-url=jdbc:mysql://192.168.56.102:3306/ds1?
characterEncoding=utf-8
sharding.jdbc.datasource.ds1.username=root
sharding.jdbc.datasource.ds1.password=root


# 水平拆分的数据库（表） 配置分库 + 分表策略 行表达式分片策略
# 分库策略     user表和address表都用uid作为分库键
sharding.jdbc.config.sharding.default-database-strategy.inline.sharding-
column=uid
sharding.jdbc.config.sharding.default-database-strategy.inline.algorithm-
expression=ds$->{uid % 2}

# 分表策略 其中user为逻辑表 分表主要取决于age行
sharding.jdbc.config.sharding.tables.user.actual-data-nodes=ds$->{0..1}.user_$->
{0..1}
sharding.jdbc.config.sharding.tables.user.table-strategy.inline.sharding-
column=age

#分表策略
sharding.jdbc.config.sharding.tables.address.actual-data-nodes=ds$->
{0..1}.address_$->{0..1}
sharding.jdbc.config.sharding.tables.address.table-strategy.inline.sharding-
column=aid


# 分片算法表达式
sharding.jdbc.config.sharding.tables.user.table-strategy.inline.algorithm-
expression=user_$->{age % 2}

sharding.jdbc.config.sharding.tables.address.table-strategy.inline.algorithm-
expression=address_$->{aid % 2}

#广播表配置
sharding.jdbc.config.sharding.broadcast-tables=code
# 主键 SNOWFLAKE 18位数 如果是分布式还要进行一个设置 防止主键重复
sharding.jdbc.config.sharding.tables.code.key-generator-column-name=id

#绑定表
sharding.jdbc.config.sharding.binding-tables=user,address

# 打印执行的数据库以及语句
sharding.jdbc.config.sharding.props.sql.show=true
spring.main.allow-bean-definition-overriding=true
```

分库分表+读写分离

ds0和ds0-slave先做好主从复制

ds0和ds1不做主从复制

```
# 数据源 ds0,ds1,ds0-slave    ds0,ds1做分库分表    ds0,ds0-slave 做主从读写分离
sharding.jdbc.datasource.names=ds0,ds1,ds0-slave
```

```
# 第一个数据库
sharding.jdbc.datasource.ds0.type=com.zaxxer.hikari.HikariDataSource
sharding.jdbc.datasource.ds0.driver-class-name=com.mysql.jdbc.Driver
sharding.jdbc.datasource.ds0.jdbc-url=jdbc:mysql://192.168.56.101:3306/ds0?
characterEncoding=utf-8
sharding.jdbc.datasource.ds0.username=root
sharding.jdbc.datasource.ds0.password=root

# 第二个数据库
sharding.jdbc.datasource.ds1.type=com.zaxxer.hikari.HikariDataSource
sharding.jdbc.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
sharding.jdbc.datasource.ds1.jdbc-url=jdbc:mysql://192.168.56.102:3306/ds1?
characterEncoding=utf-8
sharding.jdbc.datasource.ds1.username=root
sharding.jdbc.datasource.ds1.password=root

# 第一个数据库的从库
sharding.jdbc.datasource.ds0-slave.type=com.zaxxer.hikari.HikariDataSource
sharding.jdbc.datasource.ds0-slave.driver-class-name=com.mysql.jdbc.Driver
sharding.jdbc.datasource.ds0-slave.jdbc-
url=jdbc:mysql://192.168.56.103:3306/ds0?characterEncoding=utf-8
sharding.jdbc.datasource.ds0-slave.username=root
sharding.jdbc.datasource.ds0-slave.password=root


# 水平拆分的数据库（表） 配置分库 + 分表策略 行表达式分片策略
# 分库策略     user表和address表都用uid作为分库键
sharding.jdbc.config.sharding.default-database-strategy.inline.sharding-
column=uid
sharding.jdbc.config.sharding.default-database-strategy.inline.algorithm-
expression=ds$->{uid % 2}

# 分表策略 其中user为逻辑表 分表主要取决于age行
sharding.jdbc.config.sharding.tables.user.actual-data-nodes=ds$->{0..1}.user_$->
{0..1}
sharding.jdbc.config.sharding.tables.user.table-strategy.inline.sharding-
column=age

#分表策略
sharding.jdbc.config.sharding.tables.address.actual-data-nodes=ds$->
{0..1}.address_$->{0..1}
sharding.jdbc.config.sharding.tables.address.table-strategy.inline.sharding-
column=aid


# 分片算法表达式
sharding.jdbc.config.sharding.tables.user.table-strategy.inline.algorithm-
expression=user_$->{age % 2}

sharding.jdbc.config.sharding.tables.address.table-strategy.inline.algorithm-
expression=address_$->{aid % 2}

#广播表配置
sharding.jdbc.config.sharding.broadcast-tables=code
# 主键 SNOWFLAKE 18位数 如果是分布式还要进行一个设置 防止主键重复
sharding.jdbc.config.sharding.tables.code.key-generator-column-name=id

#绑定表
```

```
sharding.jdbc.config.sharding.binding-tables=user,address

#主库
sharding.jdbc.config.sharding.master-slave-rules.ds0.master-data-source-name=ds0
#从库
sharding.jdbc.config.sharding.master-slave-rules.ds0.slave-data-source-
names=ds0-slave

# 打印执行的数据库以及语句
sharding.jdbc.config.sharding.props.sql.show=true
spring.main.allow-bean-definition-overriding=true
```