

## 一、ShardingJDBC内核工作原理解读

- 0、配置管控
- 1、SQL Parser: SQL解析引擎
- 2、SQL Router- SQL 路由引擎
- 3、SQL Rewriter : SQL 优化引擎
- 4、SQL Executor : SQL执行引擎
- 5、Result Merger: 结果归并

## 二、ShardingJDBC重点源码解读

## 三、理解ShardingSphere核心的SPI扩展机制

- 1、从主键生成策略入手
- 2、尝试扩展分片算法

## 五、章节总结

# ShardingJDBC核心源码以及内核解析

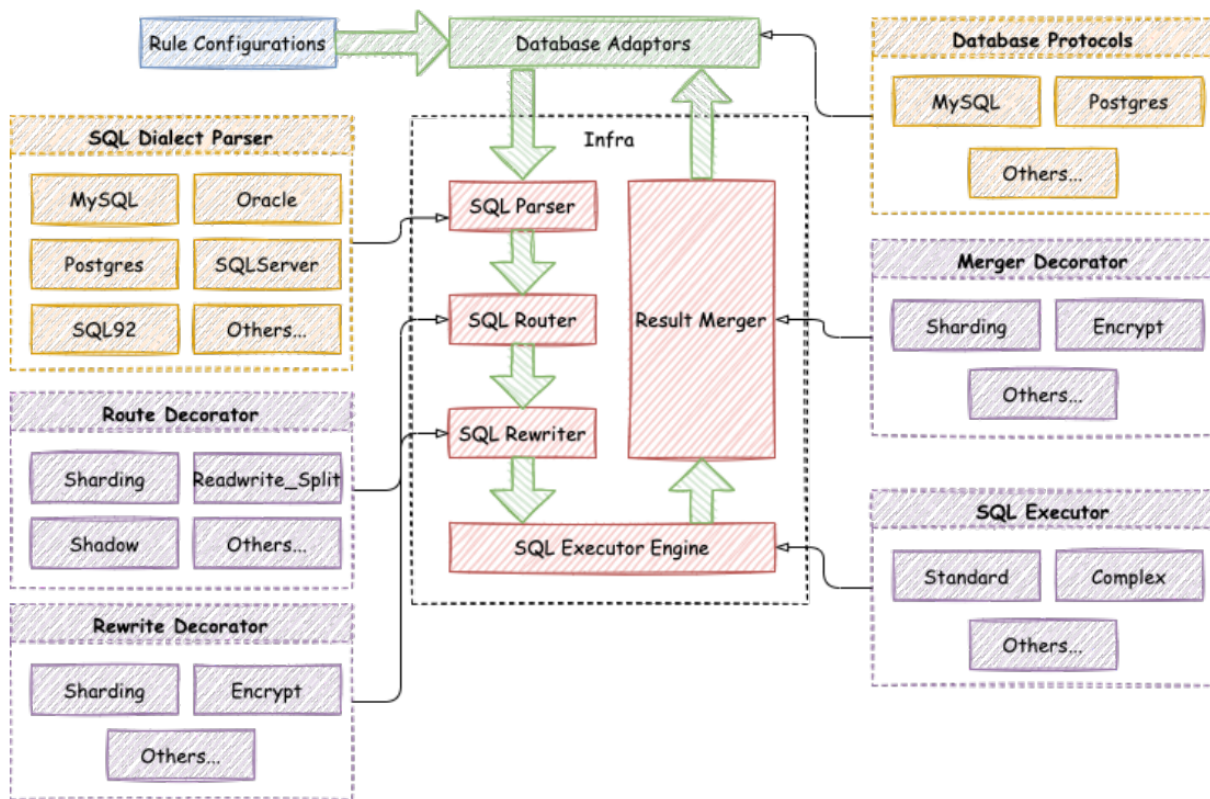
-- 楼兰

通过上一章节，我们已经对ShardingJDBC的各种使用方式有了一些了解，这一章节我们自然要更进一步，深入ShardingJDBC的内核。但是内核其实是非常复杂的，决定开始深入之前，最好带上明确的目的，否则只会是一头雾水。这次我们的主要问题就从实际使用出发，重点了解一下上一章节中用到的各种各样花里胡哨的配置，是怎么从ShardingJDBC的内核中扩展出来的。以及我们应该要如何自行扩展各种组件。

## 一、ShardingJDBC内核工作原理解读

---

在深入之前，你需要了解ShardingJDBC整体上是怎样工作的。ShardingSphere的整体架构图是这样的：



可以看到，ShardingSphere的工作整体就分为这几个步骤：

## 0、配置管控

在进入ShardingSphere的内核之前，ShardingSphere做了大量的配置信息管控。不光是将应用的配置信息进行解析，同时ShardingSphere还支持将这些配置信息放到第三方的注册中心，从而可以实现应用层的水平扩展。

对于使用ShardingJDBC开发来说，或许这不是一个多起眼的功能。因为应用完全可以自己管理配置，或者自行接入Nacos这样的配置中心。但是如果运维人员使用ShardingProxy的话，这样的配置管控功能就非常有作用了。后面分享ShardingProxy时，我们再深入理解。

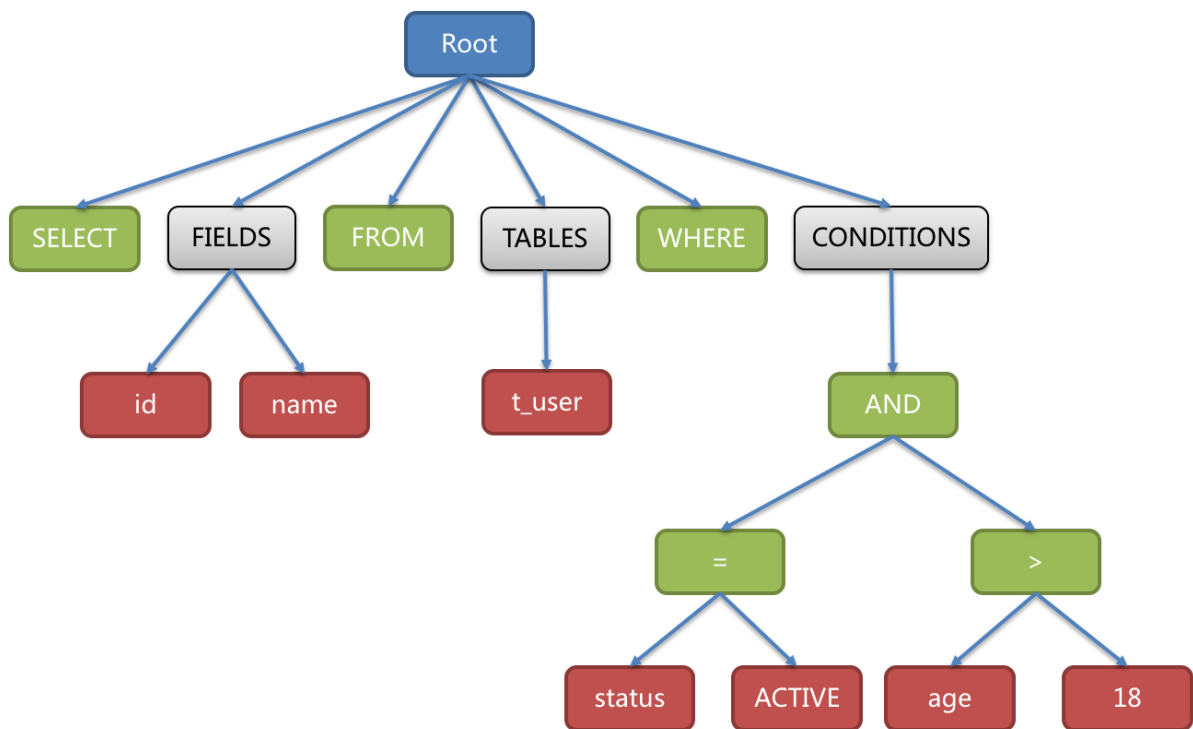
## 1、SQL Parser: SQL解析引擎

解析过程分为词法解析和语法解析。词法解析器用于将SQL拆解为不可再分的原子符号，称为Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。再使用语法解析器将SQL转换为抽象语法树(简称AST， Abstract Syntax Tree)。

例如对下面一条SQL语句：

```
SELECT id, name FROM t_user WHERE status = 'ACTIVE' AND age > 18
```

会被解析成下面这样一颗树：



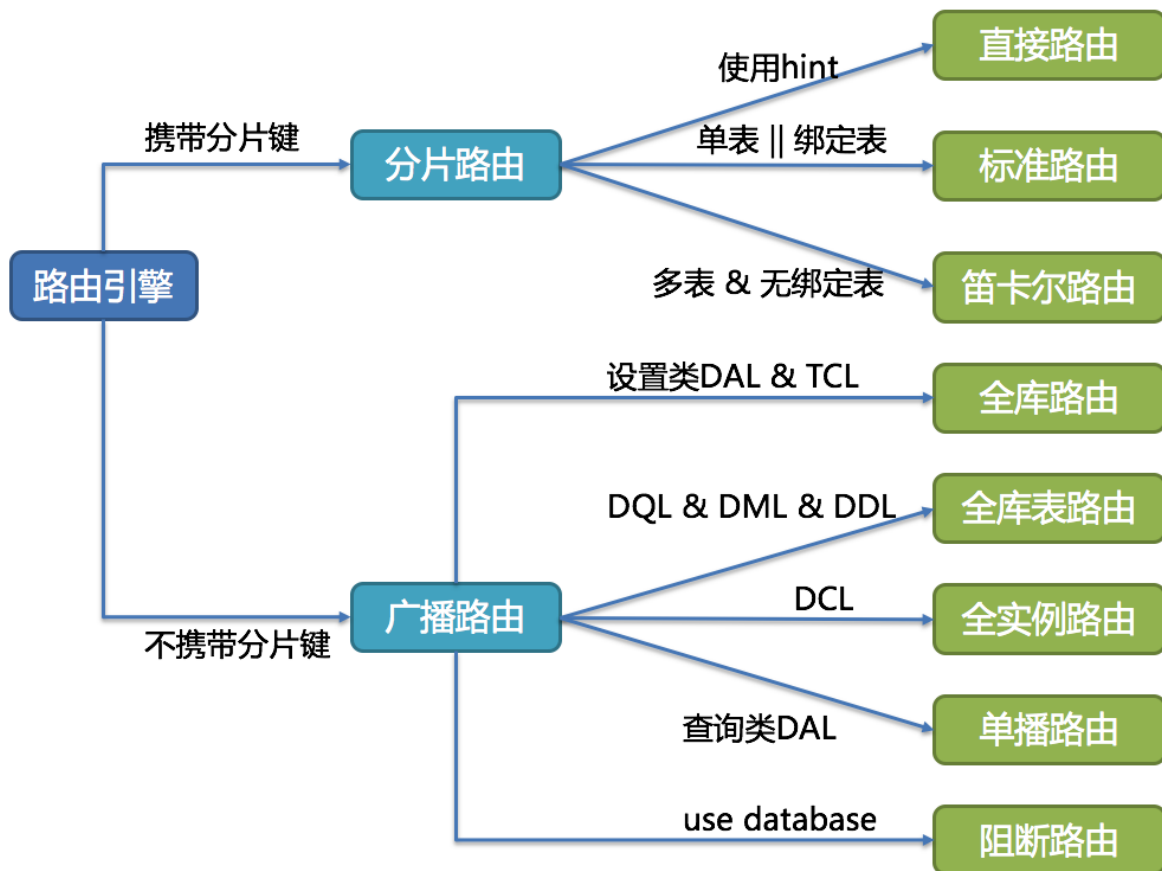
SQL解析是整个分库分表产品的核心，其性能和兼容性是最重要的衡量指标。ShardingSphere在1.4.x之前采用的是性能较快的Druid作为SQL解析器。1.5.x版本后，采用自研的SQL解析器，针对分库分表场景，采取对SQL半理解的方式，提高SQL解析的性能和兼容性。然后从3.0.x版本后，开始使用ANTLR作为SQL解析引擎。这是个开源的SQL解析引擎，很多开源产品都使用他来解析SQL。比如像Druid、Flink、Hive、RocketMQ、Elasticsearch等等。有兴趣的同学可以自行了解一下。ShardingSphere在使用ANTLR时，还增加了一些AST的缓存功能。针对ANTLR4的特性，官网建议尽量采用PreparedStatement的预编译方式来提高SQL执行的性能。

## 2、SQL Router- SQL 路由引擎

根据解析上下文匹配数据库和表的分片策略，并生成路由路径。对于携带分片键的SQL，根据分片键的不同可以划分为单片路由（分片键的操作符是等号）、多片路由（分片键的操作符是 IN）和范围路由（分片键的操作符是 BETWEEN）。不携带分片键的SQL则采用广播路由。

分片策略通常可以采用由数据库内置或由用户方配置。数据库内置的方案较为简单，内置的分片策略大致可分为尾数取模、哈希、范围、标签、时间等。由用户方配置的分片策略则更加灵活，可以根据使用方需求定制复合分片策略。

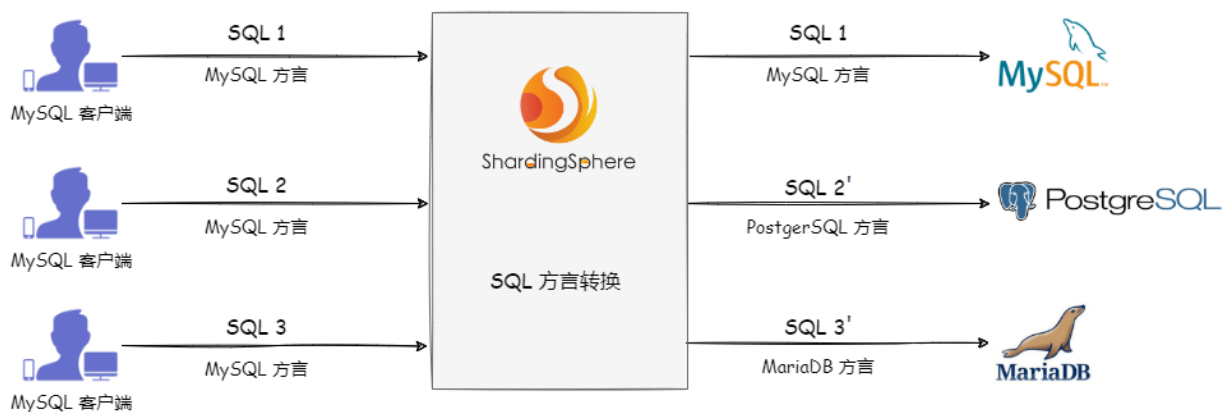
实际使用时，应尽量使用分片路由，明确路由策略。因为广播路由影响过大，不利于集群管理及扩展。



- 全库表路由：对于不带分片键的DQL、DML以及DDL语句，会遍历所有的库表，逐一执行。例如 `select * from course` 或者 `select * from course where ustatus='1'`(不带分片键)
- 全库路由：对数据库的操作都会遍历所有真实库。例如 `set autocommit=0`
- 全实例路由：对于DCL语句，每个数据库实例只执行一次，例如 `CREATE USER customer@127.0.0.1 identified BY '123';`
- 单播路由：仅需要从任意库中获取数据即可。例如 `DESCRIBE course`
- 阻断路由：屏蔽SQL对数据库的操作。例如 `USE coursedb`。就不会在真实库中执行，因为针对虚拟表操作，不需要切换数据库。

### 3、SQL Rewriter : SQL 优化引擎

首先，在数据方言方面。Apache ShardingSphere 提供了 SQL 方言翻译的能力，能否实现数据库方言之间的自动转换。例如，用户可以使用 MySQL 客户端连接 ShardingSphere 并发送基于 MySQL 方言的 SQL，ShardingSphere 能自动识别用户协议与存储节点类型自动完成 SQL 方言转换，访问 PostgreSQL 等异构存储节点。



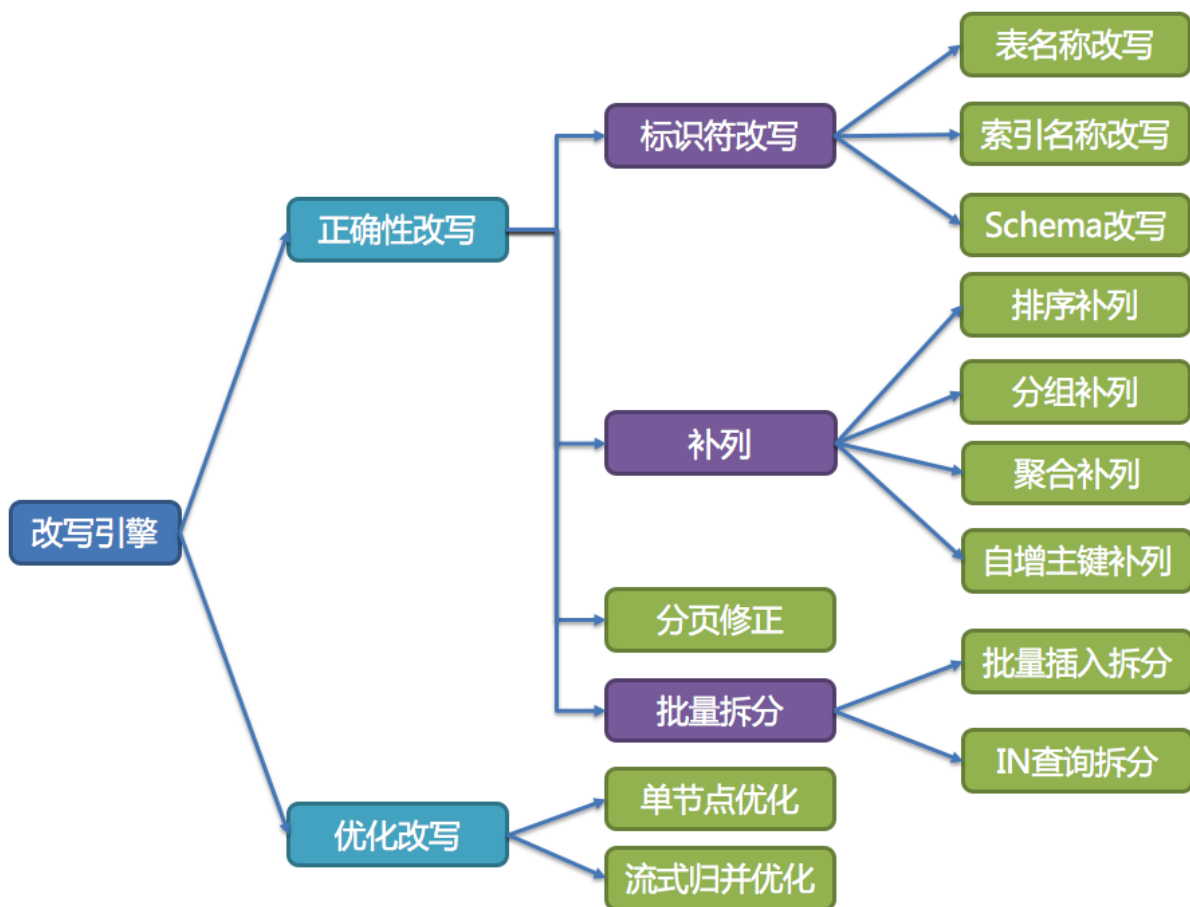
接下来，用户只需要面向逻辑库和逻辑表来写SQL，最终由ShardigSphere的改写引擎将SQL改写为在真实数据库中可以正确执行的语句。SQL改写分为正确性改写和优化改写。

### 正确性改写

在包含分表的场景中，需要将分表配置中的逻辑表名称改写为路由之后所获取的真实表名称。仅分库则不需要表名称的改写。除此之外，还包括补列和分页信息修正等内容。

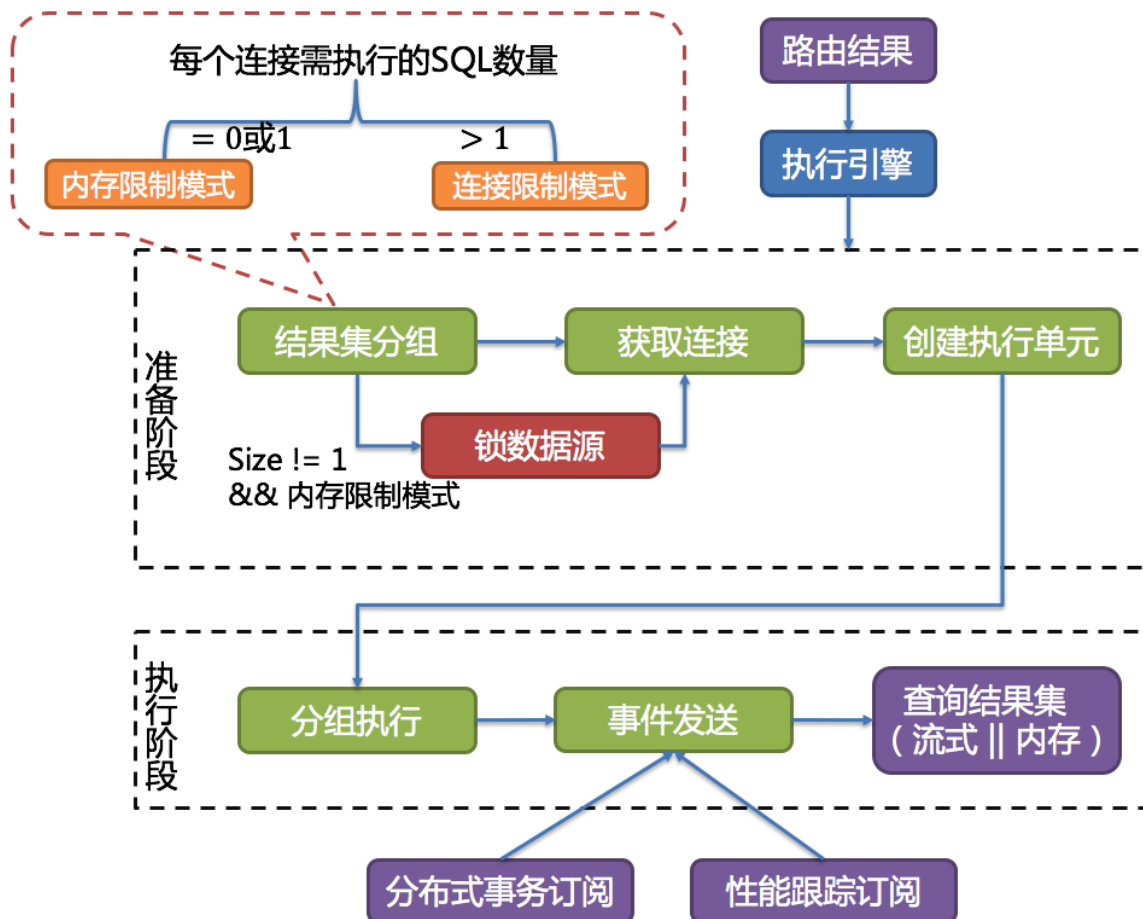
### 优化改写

优化改写的目的是在不影响查询正确性的情况下，对性能进行提升的有效手段。它分为单节点优化和流式归并优化。比如我们之前提到，在当前版本下，对一个库的多次查询，会通过UNION 合并成一个大的SQL，这也是一种优化改写。



## 4、SQL Executor：SQL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。它不是简单地将 SQL 通过 JDBC 直接发送至数据源执行；也并非直接将执行请求放入线程池去并发执行。它更关注平衡数据源连接创建以及内存占用所产生的消耗，以及最大限度地合理利用并发等问题。执行引擎的目标是自动化的平衡资源控制与执行效率。



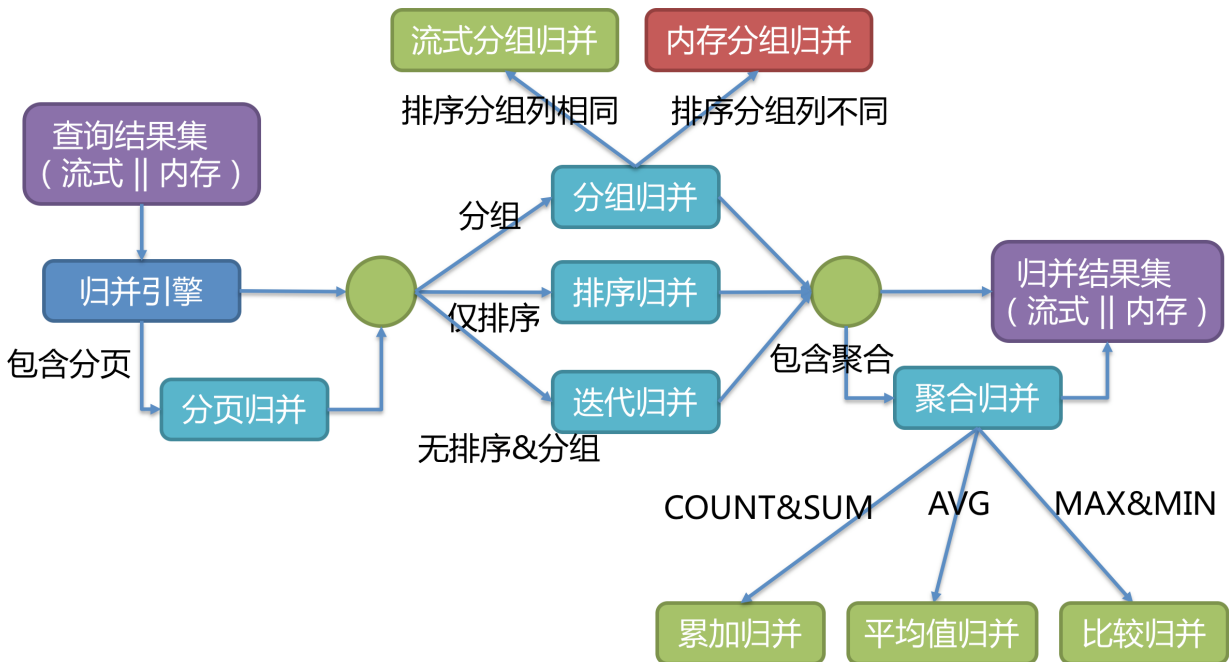
这里主要是理解内存限制模式和连接限制模式。简单理解，

- 内存限制模式只需要保持一个JDBC连接，单线程即可完成某一个真实库的所有数据访问。
- 连接限制模式就需要保持多个JDBC连接，也就需要多线程并发完成某一个真实库的所有数据访问。

## 5、Result Merger：结果归并

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。





其中重点是理解流式归并与内存归并：

- 流式归并是指每一次从结果集中获取到的数据，都能够通过逐条获取的方式返回正确的单条数据，它与数据库原生的返回结果集的方式最为契合。遍历、排序以及流式分组都属于流式归并的一种。通常内存限制模式就可以使用流式归并，比较适合OLTP场景。
- 内存归并则是需要将结果集的所有数据都遍历并存储在内存中，再通过统一的分组、排序以及聚合等计算之后，再将其封装成为逐条访问的数据结果集返回。。通常连接限制模式就可以使用内存归并，比较适合OLAP场景。

## 二、ShardingJDBC重点源码解读

这些引擎都比较抽象，最终还是需要深入到源码当中才能理解他们的庐山真面目。

如何调试ShardingJDBC的源码呢？之前分析过，ShardingJDBC的本质其实是生成一个带有分库分表功能的ShardingSphereDataSource。他就是我们经常使用的DataSource接口的实现类。因此，我们也可以尝试构建一个JDBC的应用，作为代码调试的起点。

```

public class ShardingJDBCDemo {
    public static void main(String[] args) throws SQLException {
        //=====一、配置数据库
        Map<String, DataSource> dataSourceMap = new HashMap<>(2); //为两个数据库的
        datasource
        // 配置第一个数据源
        HikariDataSource dataSource0 = new HikariDataSource();
        dataSource0.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource0.setJdbcUrl("jdbc:mysql://localhost:3306/coursedb?
        serverTimezone=GMT%2B8&useSSL=false");
        dataSource0.setUsername("root");
        dataSource0.setPassword("root");
        dataSourceMap.put("m0", dataSource0);
        // 配置第二个数据源
    }
}
  
```

```

HikariDataSource dataSource1 = new HikariDataSource();
dataSource1.setDriverClassName("com.mysql.cj.jdbc.Driver");
dataSource1.setJdbcUrl("jdbc:mysql://localhost:3306/coursedb2?
serverTimezone=GMT%2B8&useSSL=false");
dataSource1.setUsername("root");
dataSource1.setPassword("root");
dataSourceMap.put("m1", dataSource1);
//=====二、配置分库分表策略
ShardingRuleConfiguration shardingRuleConfig = createRuleConfig();
//三、配置属性值
Properties properties = new Properties();
//打开日志输出 4.x版本是sql.show, 5.x版本变成了sql-show
properties.setProperty("sql-show", "true");
//K1 创建ShardingSphere的数据源 ShardingDataSource
DataSource dataSource =
ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,Collections.singleton
(shardingRuleConfig), properties);

//-----测试部分-----//
ShardingJDBCDemo test = new ShardingJDBCDemo();
//建表
//      test.droptable(dataSource);
//      test.createtable(dataSource);
//插入数据
//      test.addcourse(dataSource);
//K1 调试的起点 查询数据
test.querycourse(dataSource);
}

private static ShardingRuleConfiguration createRuleConfig(){
    ShardingRuleConfiguration result = new ShardingRuleConfiguration();
    //spring.shardingsphere.rules.sharding.tables.course.actual-data-nodes=m$->
{0..1}.course_$->{1..2}
    ShardingTableRuleConfiguration courseTableRuleConfig = new
ShardingTableRuleConfiguration("course",
        "m$->{0..1}.course_$->{1..2}");
    //spring.shardingsphere.rules.sharding.key-
generators.alg_snowflake.type=SNOWFLAKE
    //spring.shardingsphere.rules.sharding.key-
generators.alg_snowflake.props.worker.id=1
    Properties snowflakeprop = new Properties();
    snowflakeprop.setProperty("worker.id", "123");
    result.getKeyGenerators().put("alg_snowflake", new
AlgorithmConfiguration("SNOWFLAKE", snowflakeprop));
    //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
    courseTableRuleConfig.setKeyGenerateStrategy(new
KeyGenerateStrategyConfiguration("cid", "alg_snowflake"));
    //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
        courseTableRuleConfig.setDatabaseShardingStrategy(new
StandardShardingStrategyConfiguration("cid","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
        Properties modProp = new Properties();
        modProp.put("sharding-count",2);
        result.getShardingAlgorithms().put("course_db_alg",new
AlgorithmConfiguration("MOD",modProp));
        //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
        courseTableRuleConfig.setTableShardingStrategy(new
StandardShardingStrategyConfiguration("cid","course_tbl_alg"));
        //spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.type=INLINE
        //spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.props.algorithm-expression=course_${cid%2+1}
        Properties inlineProp = new Properties();
        inlineProp.setProperty("algorithm-expression", "course_${cid%2+1}");
        result.getShardingAlgorithms().put("course_tbl_alg",new
AlgorithmConfiguration("INLINE",inlineProp));

        result.getTables().add(courseTableRuleConfig);
        return result;
    }

    //添加10条课程记录
    public void addcourse(DataSource dataSource) throws SQLException {
        for (int i = 1; i < 10; i++) {
            long orderId = executeAndGetGeneratedKey(dataSource, "INSERT INTO course
(cname, user_id, cstatus) VALUES ('java'," + i + ", '1')");
            System.out.println("添加课程成功, 课程ID: " + orderId);
        }
    }

    public void querycourse(DataSource dataSource) throws SQLException {
        Connection conn = null;
        try {
            //ShardingConnection
            conn = dataSource.getConnection();
            //ShardingStatement
            Statement statement = conn.createStatement();
            String sql = "SELECT cid,cname,user_id,cstatus from course where
cid=851198093910081536";
            //ShardingResultSet
            ResultSet result = statement.executeQuery(sql);

```

```

        while (result.next()) {
            System.out.println("result:" + result.getLong("cid"));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        if (null != conn) {
            conn.close();
        }
    }
}

private void execute(final DataSource dataSource, final String sql) throws
SQLException {
    try (
        Connection conn = dataSource.getConnection();
        Statement statement = conn.createStatement()) {
        statement.execute(sql);
    }
}

private long executeAndGetGeneratedKey(final DataSource dataSource, final String
sql) throws SQLException {
    long result = -1;
    try (
        Connection conn = dataSource.getConnection();
        Statement statement = conn.createStatement()) {
        statement.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
        ResultSet resultSet = statement.getGeneratedKeys();
        if (resultSet.next()) {
            result = resultSet.getLong(1);
        }
    }
    return result;
}

/**
 * -----表初始化-----
 */
public void droptable(DataSource dataSource) throws SQLException {
    execute(dataSource, "DROP TABLE IF EXISTS course_1");
    execute(dataSource, "DROP TABLE IF EXISTS course_2");
}

public void createtable(DataSource dataSource) throws SQLException {
    execute(dataSource, "CREATE TABLE course_1 (cid BIGINT(20) PRIMARY KEY,cname
VARCHAR(50) NOT NULL,user_id BIGINT(20) NOT NULL,cstatus varchar(10) NOT NULL);");
    execute(dataSource, "CREATE TABLE course_2 (cid BIGINT(20) PRIMARY KEY,cname
VARCHAR(50) NOT NULL,user_id BIGINT(20) NOT NULL,cstatus varchar(10) NOT NULL);");
}
}

```

前面一大段代码都不用太过关注，无法就是将之前采用配置文件指定的规则换成了Java实现而已。重点从K1处调试即可。

## 三、理解ShardingSphere核心的SPI扩展机制

### 1、从主键生成策略入手

源码中有非常多值得学习研究的部分。按照之前的思路，我们重点是理解配置的各种策略是如何从ShardingSphere中扩展出来的。以主键生成策略为例，抽取重点源码解读一下：

```
package org.apache.shardingsphere.sharding.factory;

/**
 * Key generate algorithm factory.
 */
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public final class KeyGenerateAlgorithmFactory {

    //加载所有的主键生成策略
    static {
        ShardingSphereServiceLoader.register(KeyGenerateAlgorithm.class);
    }

    /**
     * 根据配置的主键生成策略，获取一个主键生成算法
     * 例如: spring.shardingsphere.rules.sharding.key-
     * generators.usercourse_keygen.type=SNOWFLAKE
     */
    public static KeyGenerateAlgorithm newInstance(final AlgorithmConfiguration
keyGenerateAlgorithmConfig) {
        return
ShardingSphereAlgorithmFactory.createAlgorithm(keyGenerateAlgorithmConfig,
KeyGenerateAlgorithm.class);
    }

    /**
     * 判断是否包含配置的算法
     */
    public static boolean contains(final String keyGenerateAlgorithmType) {
        return TypedSPIRegistry.findRegisteredService(KeyGenerateAlgorithm.class,
keyGenerateAlgorithmType).isPresent();
    }
}
```

先来看主键生成策略是如何加载的：

```
ShardingSphereServiceLoader.register(KeyGenerateAlgorithm.class);
```

```
public final class ShardingSphereServiceLoader {

    //线程安全Map，缓存所有主键生成器
    private static final Map<Class<?>, Collection<Object>> SERVICES = new
ConcurrentHashMap<>();
```

```

public static void register(final Class<?> serviceInterface) {
    if (!SERVICES.containsKey(serviceInterface)) {
        SERVICES.put(serviceInterface, load(serviceInterface));
    }
}
//使用java的SPI机制加载接口的所有实现类
private static <T> Collection<Object> load(final Class<T> serviceInterface) {
    Collection<Object> result = new LinkedList<>();
    for (T each : ServiceLoader.load(serviceInterface)) {
        result.add(each);
    }
    return result;
}
}

```

其中看到了最熟悉不过的ServiceLoader.load方法，这就是Java提供的SPI扩展机制。这意味着，我们自己也可以写一个KeyGenerateAlgorithm实现类，然后只需要通过SPI的方式让ShardingSphere加载进去就行。

然后我们看看KeyGenerateAlgorithm有哪些实现类。比如说随便找一个比较眼熟的NanoIdKeyGenerateAlgorithm。他的源码很简单

```

public final class NanoIdKeyGenerateAlgorithm implements KeyGenerateAlgorithm {
    private Properties props;

    public NanoIdKeyGenerateAlgorithm() {
    }

    public void init(Properties props) {
        this.props = props;
    }

    public String generateKey() {
        return NanoIdUtils.randomNanoId(ThreadLocalRandom.current(),
NanoIdUtils.DEFAULT_ALPHABET, 21);
    }

    public String getType() {
        return "NANOID";
    }

    @Generated
    public Properties getProps() {
        return this.props;
    }
}

```

这个getType方法简单粗暴的返回了一个字符串NANOID，有没有觉得跟我们在配置文件当中配置的主键生成算法很像？

看到这，我们就可以自己做一个实验，尝试扩展一个自己的主键生成策略。

首先：自行实现一个算法类，实现KeyGenerateAlgorithm接口。

```
public class MyKeyGeneratorAlgorithm implements KeyGenerateAlgorithm {

    private AtomicLong atom = new AtomicLong(0);

    private Properties props;

    @Override
    public Comparable<?> generateKey() {
        LocalDateTime ldt = LocalDateTime.now();
        String timestamps = DateTimeFormatter.ofPattern("HHmmssSSS").format(ldt);
        return Long.parseLong(""+timestamps+atom.incrementAndGet());
    }

    @Override
    public Properties getProps() {
        return this.props;
    }

    public String getType() {
        return "MYKEY";
    }

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

然后：在classpath/META-INF/services目录下(这个目录是Java的SPI机制加载的固定目录)创建一个SPI的扩展文件，文件名就是接口的全类名。org.apache.shardingsphere.sharding.spi.KeyGenerateAlgorithm。文件中的内容，一行就记录一个接口的实现类全类名。就把我们自己写的这个类名写进去。  
com.roy.shardingDemo.algorithm.MyKeyGeneratorAlgorithm

最后：在我们之前做的生成主键的Demo中修改一下配置，将主键生成算法的类型配置成MYKEY

```
spring.shardingsphere.rules.sharding.key-generators.course_cid_alg.type=MYKEY
```

这样，在插入course表的时候，就可以按照我们自己的实现生成主键了。

通过这个简单的试验，你是不是可以自行梳理出ShardingSphere提供了哪些主键生成算法，以及这些算法要怎么配了？实际上，这也就对应了官方文档中不太看得懂这一段技术手册。

5.7.2 KeyGenerateAlgorithm

全限定类名

org.apache.shardingsphere.sharding.spi.KeyGenerateAlgorithm

定义

分布式主键生成算法

已知实现

配置标识	详细说明	全限定类名
SNOWFLAKE	基于雪花算法的分布式主键生成算法	org.apache.shardingsphere.sharding.algorithm.keygen.SnowflakeKeyGenerateAlgorithm
UUID	基于 UUID 的分布式主键生成算法	org.apache.shardingsphere.sharding.algorithm.keygen.UUIDKeyGenerateAlgorithm
NANOID	基于 NanoId 的分布式主键生成算法	org.apache.shardingsphere.sharding.algorithm.keygen.NanoIdKeyGenerateAlgorithm
COSID	基于 CosId 的分布式主键生成算法	org.apache.shardingsphere.sharding.algorithm.keygen.CosIdKeyGenerateAlgorithm
COSID_SNOWFLAKE	基于 CosId 的雪花算法分布式主键生成算法	org.apache.shardingsphere.sharding.cosid.algorithm.keygen.CosIdSnowflakeKeyGenerateAlgorithm

看懂了这一个部分了之后，官方文档中其他那些组件，你也就都可以入手自己梳理了。

接下来，如果你对主键生成策略比较熟悉，可以尝试对接一些第三方的主键生成框架。比如最简单的通过Redis的incr指令生成主键，或者像美团的leaf这一类主键生成框架。

另外，如果你有兴趣，建议你再通过CosIdKeyGenerateAlgorithm去深入理解一下CosID主键生成框架。这是ShardingSphere引入的一个开源的主键生成框架，据说性能非常高。官方地址：<https://cosid.ahoo.me/>。但是，据我的试验，当前版本下对CosID的segmentID模式集成好像还不是太好。有兴趣你可以自行尝试一下。

2、尝试扩展分片算法

还记得在上一章节中，我们在介绍COMPLEX复杂分片策略时，创建了一个MyComplexAlgorithm吗？之前我们是通过ShardingSphere提供的一个很别扭的CLASS\_BASED类型的分片算法配置进去的。实际上，我们也可以使用ShardingSphere提供的SPI机制配置进去。

首先，需要添加一个SPI的配置文件org.apache.shardingsphere.sharding.spi.ShardingAlgorithm，然后在文件中写入com.roy.shardingDemo.algorithm.MyComplexAlgorithm。就可以完成算法接入了。

然后，如果想要能够被配置文件识别，我们可以在MyComplexAlgorithm类中，增加实现getType方法。

```
public class MyComplexAlgorithm implements ComplexKeysShardingAlgorithm<Long> {  
    //.....  
    @Override  
    public String getType(){  
        return "MYCOMPLEX";  
    }  
}
```

示例中读取了一个属性sharding-columns，在示例中其实并没有用到。如果你需要增加有用的属性时，可以进行参照。

接下来，就可以在配置分库分表策略时，指定我们自己的这个实现类了。

```
spring.shardingsphere.rules.sharding.sharding-  
algorithms.course_tbl_alg.type=MYCOMPLEX
```

最后，愉快的尝试扩展其他组件吧。

## 五、章节总结

这一章节我们理解了一下ShardingSphere内核当中比较重要的几个机制。然后建立了一个示例，作为调试重点源码的入口。最后带着扩展主键生成算法的目的，从源码中推测出了ShardingSphere提供的SPI扩展点。整个过程虽然并没有研究太深，但是整个路线最大的目的是为了给你打开一算深入理解ShardingSphere的门。将应用和源码之间的隔阂打开。有兴趣的话，你可以更深入的研究了解ShardingSphere的底层，结合你自己的业务场景，去设计更复杂的分库分表策略，支撑数据量更大，更复杂的分库分表场景。

有道云笔记链接：<https://note.youdao.com/s/6e2NjaCd>