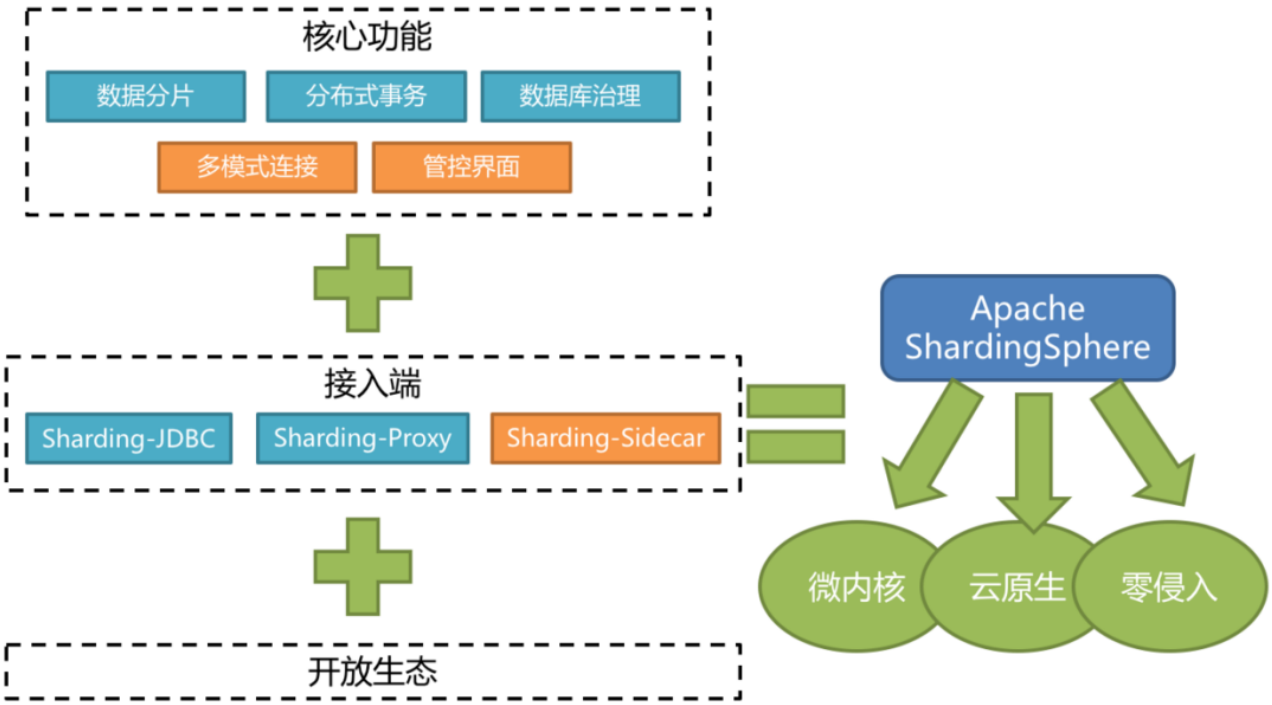
1. 概述

ShardingSphere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 这3款相互独立的产品组成。他们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、云原生等各种多样化的应用场景。

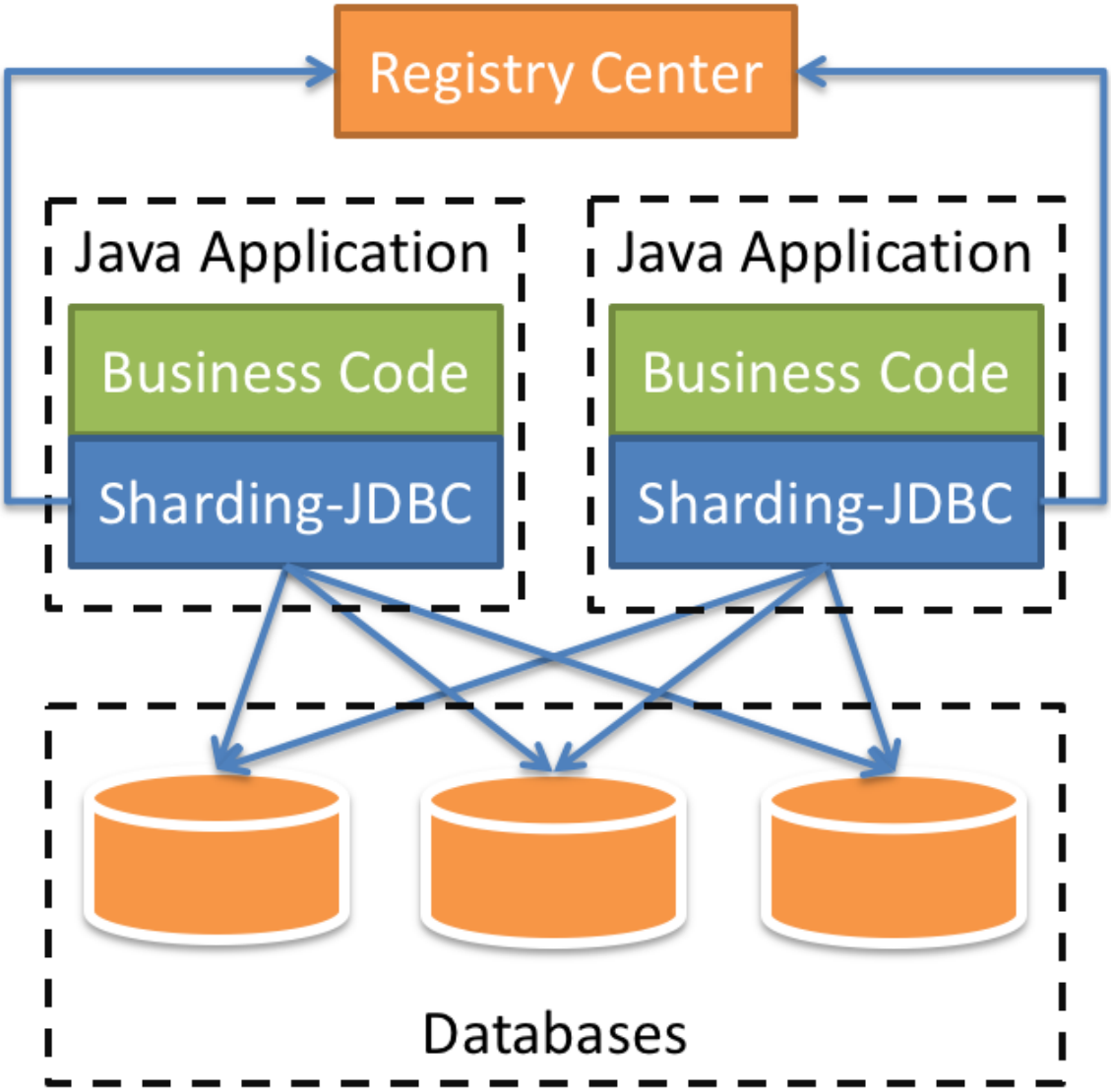
ShardingSphere定位为关系型数据库中间件，旨在充分合理地在分布式的场景下利用关系型数据库的计算和存储能力。



1.1. ShardingSphere-JDBC

Sharding-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。

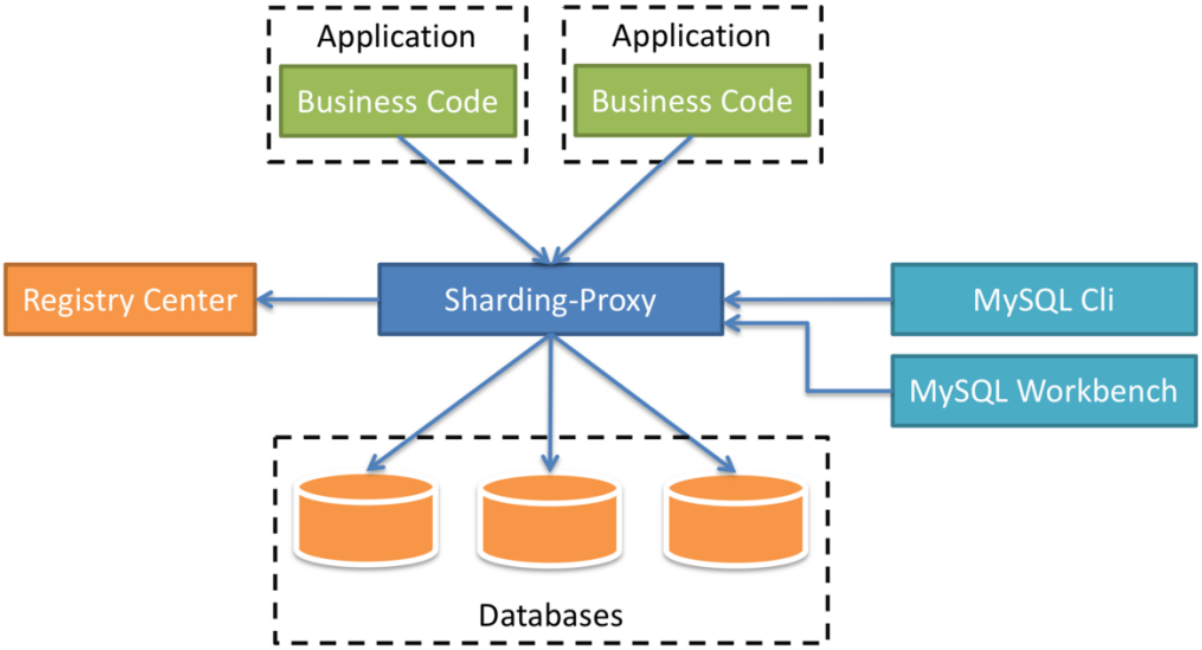
- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC。
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, Druid, HikariCP 等。
- 支持任意实现JDBC规范的数据库。目前支持 MySQL，Oracle，SQLServer，PostgreSQL 以及任何遵循 SQL92 标准的数据库。



1.2. ShardingSphere-Proxy

Sharding-Proxy 定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。目前提供 MySQL 和 PostgreSQL 版本，它可以使用任何兼容 MySQL/PostgreSQL 协议的访问客户端(如：MySQL Command Client, MySQL Workbench, Navicat 等)操作数据，对 DBA 更加友好。

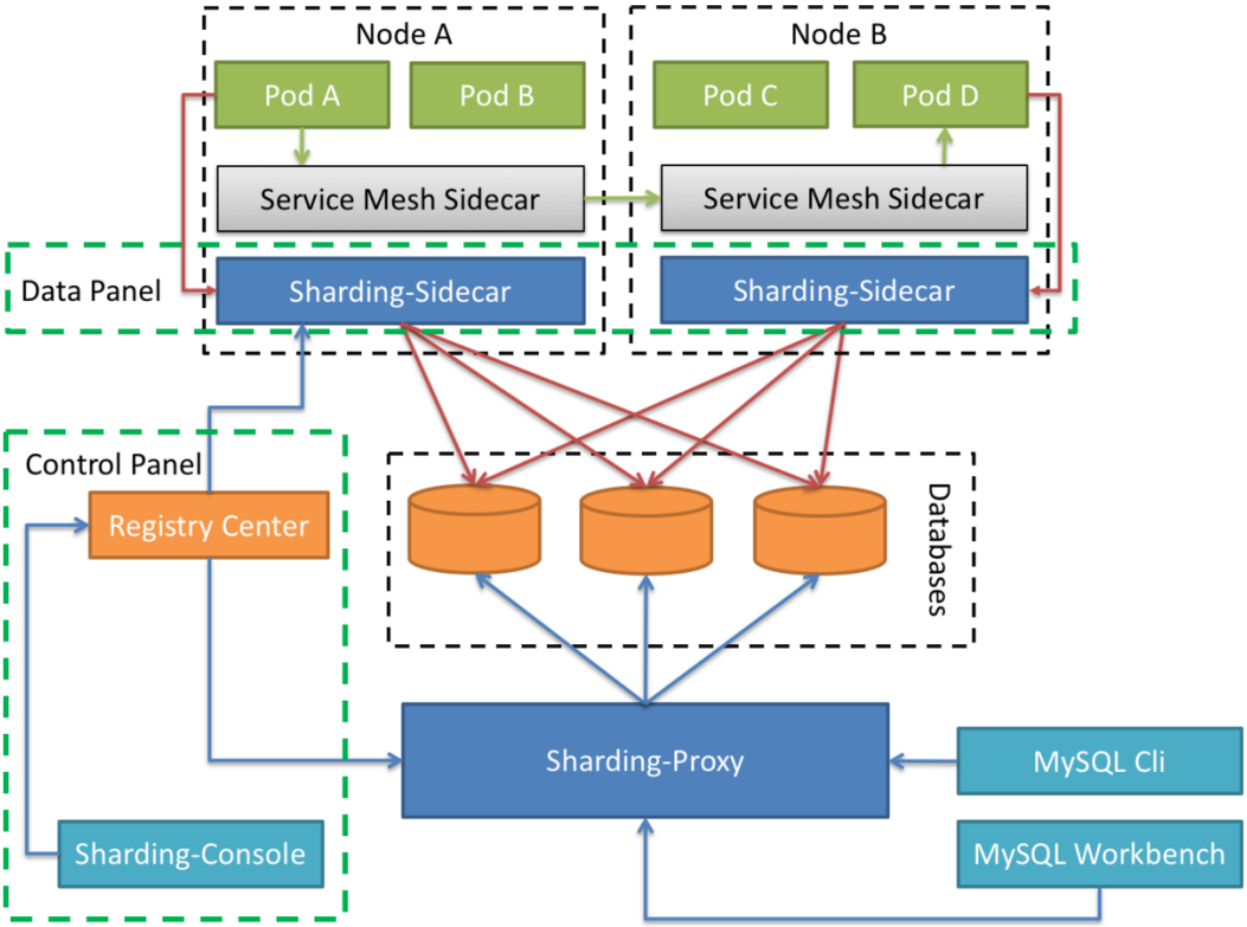
- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用。
- 适用于任何兼容 MySQL/PostgreSQL 协议的的客户端。



1.3. ShardingSphere-Sidecar

Sharding-Sidecar 定位为 Kubernetes 的云原生数据库代理，以 Sidecar 的形式代理所有对数据库的访问。通过无中心、零侵入的方案提供与数据库交互的的啮合层，即 Database Mesh，又可称数据库网格。

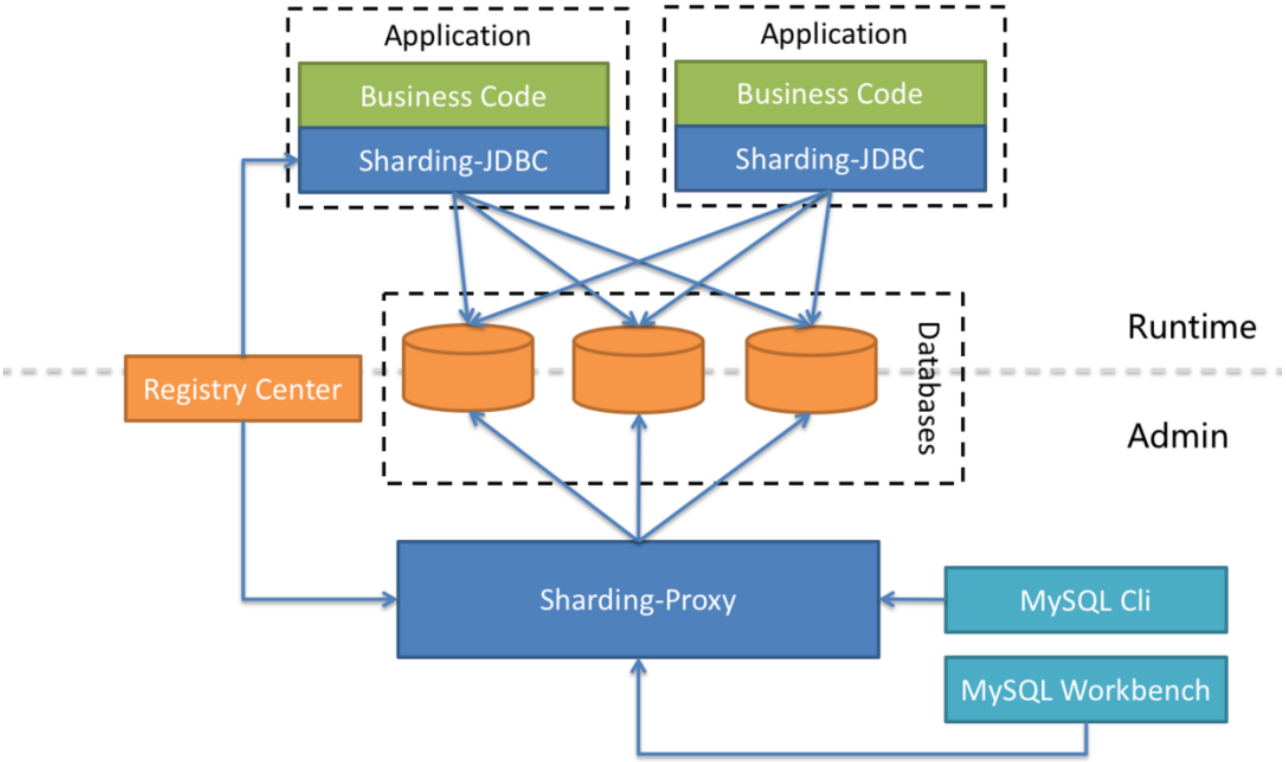
Database Mesh 的关注重点在于如何将分布式的数据访问应用与数据库有机串联起来，它更加关注的是交互，是将杂乱无章的应用与数据库之间的交互有效的梳理。使用 Database Mesh，访问数据库的应用和数据库终将形成一个巨大的网格体系，应用和数据库只需在网格体系中对号入座即可，它们都是被啮合层所治理的对象。



1.4. 混合架构

ShardingSphere-JDBC 采用无中心化架构，适用于 Java 开发的高性能的轻量级 OLTP 应用；ShardingSphere-Proxy 提供静态入口以及异构语言的支持，适用于 OLAP 应用以及对分片数据库进行管理和运维的场景。

Apache ShardingSphere 是多接入端共同组成的生态圈。 通过混合使用 ShardingSphere-JDBC 和 ShardingSphere-Proxy，并采用同一注册中心统一配置分片策略，能够灵活的搭建适用于各种场景的应用系统，使得架构师更加自由的调整适合与当前业务的最佳系统架构。



2. 概念 & 功能

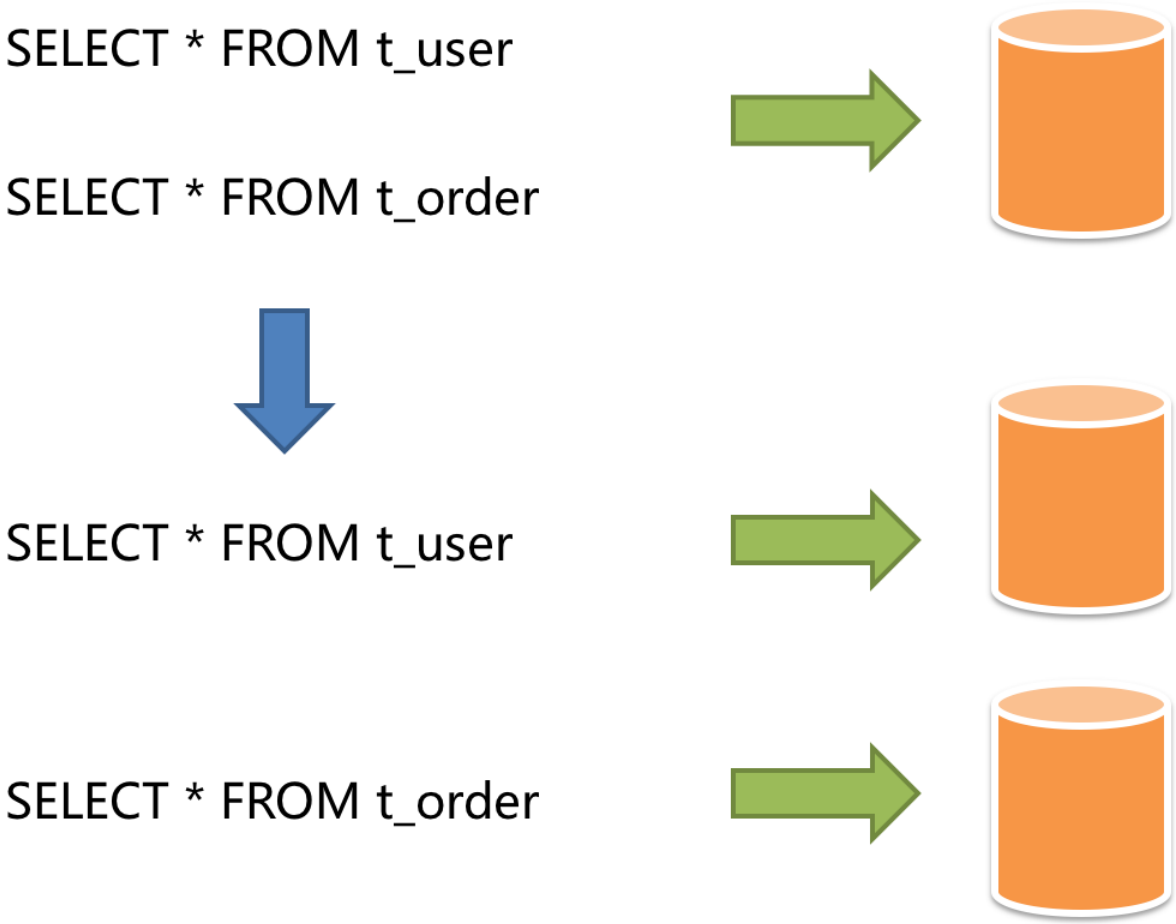
2.1. 数据分片

从性能方面来说，由于关系型数据库大多采用B+树类型的索引，在数据量超过阈值的情况下，索引深度的增加也将使得磁盘访问的IO次数增加，进而导致查询性能的下降；同时，高并发访问请求也使得集中式数据库成为系统的最大瓶颈。

从运维成本方面考虑，当一个数据库实例中的数据达到阈值以上，对于DBA的运维压力就会增大。数据备份和恢复的时间成本都将随着数据量的大小而愈发不可控。一般来讲，单一数据库实例的数据的阈值在1TB之内，是比较合理的范围。

垂直分片

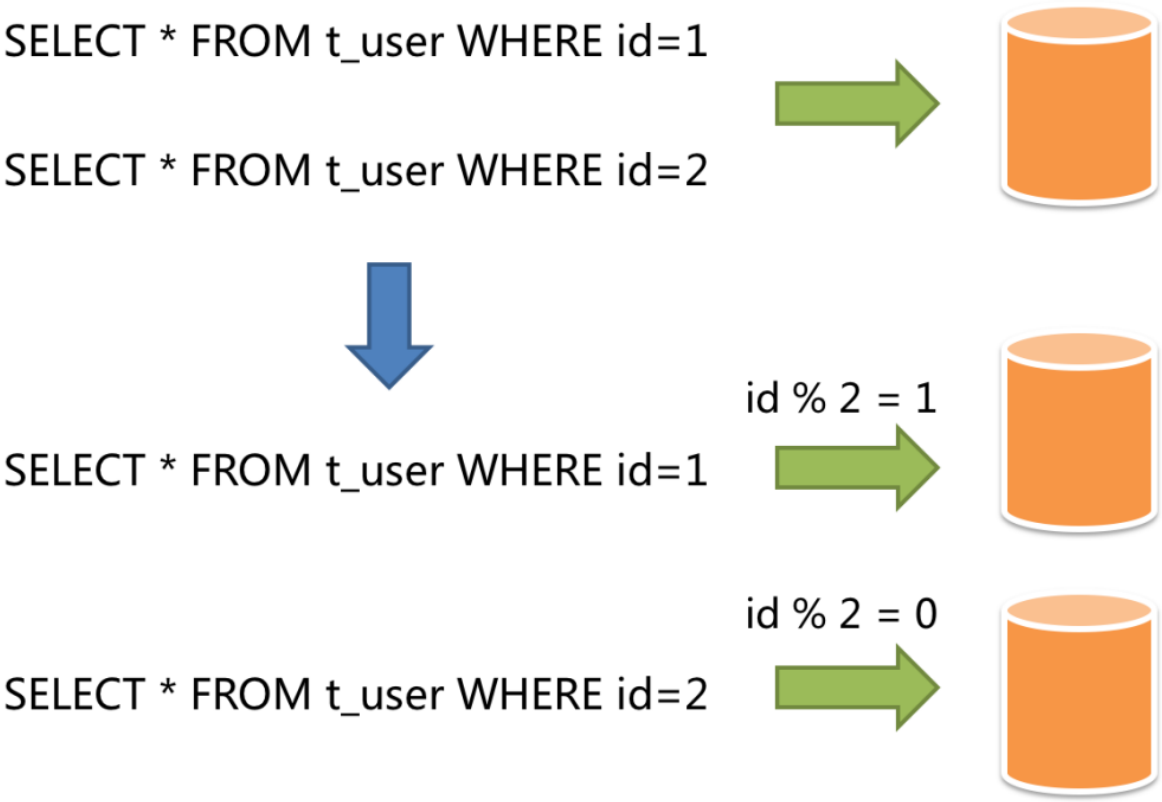
按照业务拆分的方式称为垂直分片，又称为纵向拆分，它的核心理念是专库专用。在拆分之前，一个数据库由多个数据表构成，每个表对应着不同的业务。而拆分之后，则是按照业务将表进行归类，分布到不同的数据库中，从而将压力分散至不同的数据库。下图展示了根据业务需要，将用户表和订单表垂直分片到不同的数据库的方案。



垂直分片往往需要对架构和设计进行调整。通常来讲，是来不及应对互联网业务需求快速变化的；而且，它也并无法真正的解决单点瓶颈。垂直拆分可以缓解数据量和访问量带来的问题，但无法根治。如果垂直拆分之后，表中的数据量依然超过单节点所能承载的阈值，则需要水平分片来进一步处理。

水平分片

水平分片又称为横向拆分。 相对于垂直分片，它不再将数据根据业务逻辑分类，而是通过某个字段（或某几个字段），根据某种规则将数据分散至多个库或表中，每个分片仅包含数据的一部分。 例如：根据主键分片，偶数主键的记录放入0库（或表），奇数主键的记录放入1库（或表），如下图所示。



水平分片从理论上突破了单机数据量处理的瓶颈，并且扩展相对自由，是分库分表的标准解决方案。

目标

尽量透明化分库分表所带来的影响，让使用方尽量像使用一个数据库一样使用水平分片之后的数据库集群，是 Apache ShardingSphere 数据分片模块的主要设计目标。

2.1.1. 核心概念

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，例如：ds_0.t_order_0。

分片键

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。例：将订单表中的订单主键的尾数取模分片，则订单主键为分片字段。

SQL 中如果无分片字段，将执行全路由，性能较差。

除了对单分片字段的支持，Apache ShardingSphere 也支持根据多个字段进行分片。

分片算法

通过分片算法将数据分片，支持通过=、>=、<=、>、<、BETWEEN和IN分片。分片算法需要应用方开发者自行实现，可实现的灵活度非常高。

分片策略

包含分片键和分片算法，由于分片算法的独立性，将其独立抽离。真正可用于分片操作的是分片键 + 分片算法，也就是分片策略。目前提供 5 种分片策略。

行表达式

使用表达式可以简化配置，只需要在配置中使用 `${ expression }` 或 `$->{ expression }` 标识行表达式即可 `${begin..end}` 表示范围区间

`${[unit1, unit2, unit_x]}` 表示枚举值

行表达式中如果出现连续多个 `${ expression }` 或 `$->{ expression }` 表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

例如，{1..3} 最终会被解析为 online_table1, online_table2, online_table3, offline_table1, offline_table2, offline_table3

分布式主键

在分片规则配置模块可配置每个表的主键生成策略，**默认使用雪花算法（snowflake）生成 64bit 的长整型数据。**

雪花算法是由 Twitter 公布的分布式主键生成算法，它能够保证不同进程主键的不重复性，以及相同进程主键的有序性。

实现原理

在同一个进程中，它首先是通过时间位保证不重复，如果时间相同则是通过序列位保证。同时由于时间位是单调递增的，且各个服务器如果大体做了时间同步，那么生成的主键在分布式环境可以认为是总体有序的，这就保证了对索引字段的插入的高效性。例如 MySQL 的 Innodb 存储引擎的主键。

使用雪花算法生成的主键，二进制表示形式包含 4 部分，从高位到低位分表为：1bit 符号位、41bit 时间戳位、10bit 工作进程位以及 12bit 序列号位。

- 符号位(1bit)

预留的符号位，恒为零。

- 时间戳位(41bit)

41 位的时间戳可以容纳的毫秒数是 2 的 41 次幂，一年所使用的毫秒数是：365 * 24 * 60 * 60 * 1000。通过计算可知：结果约等于 69.73 年。Apache ShardingSphere的雪花算法的时间纪元从2016年11月1日零点开始，可以使用到2086年，相信能满足绝大部分系统的要求。

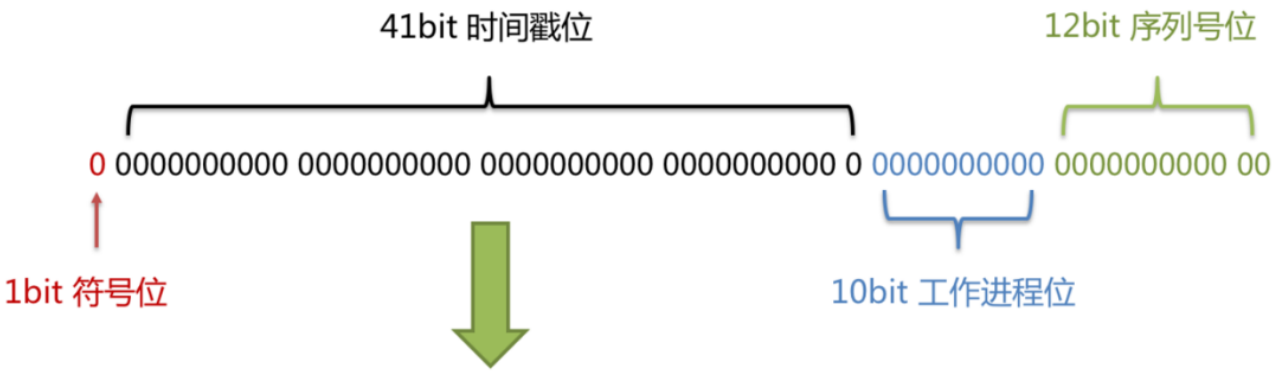
- 工作进程位(10bit)

该标志在 Java 进程内是唯一的，如果是分布式应用部署应保证每个工作进程的 id 是不同的。该值默认为 0，可通过属性设置。

- 序列号位(12bit)

该序列是用来在同一个毫秒内生成不同的 ID。如果在这个毫秒内生成的数量超过 4096 (2的12次幂)，那么生成器会等待到下个毫秒继续生成。

雪花算法主键的详细结构见下图：



时间范围： 2^{41} / (365 * 24 * 60 * 60 * 1000L) = 69.73年
工作进程数量： 2^{10} = 1024
生成不碰撞序列的TPS： 2^{12} * 1000 = 409.6万

2.1.2. 使用规范

下面列出已明确可支持的SQL种类以及已明确不支持的SQL种类，尽量让使用者避免踩坑。

支持项

路由至单数据节点

- 100%全兼容（目前仅MySQL，其他数据库完善中）

路由至多数据节点

- 全面支持DML、DDL、DCL、TCL和部分DAL。支持分页、去重、排序、分组、聚合、关联查询（不支持跨库关联）。

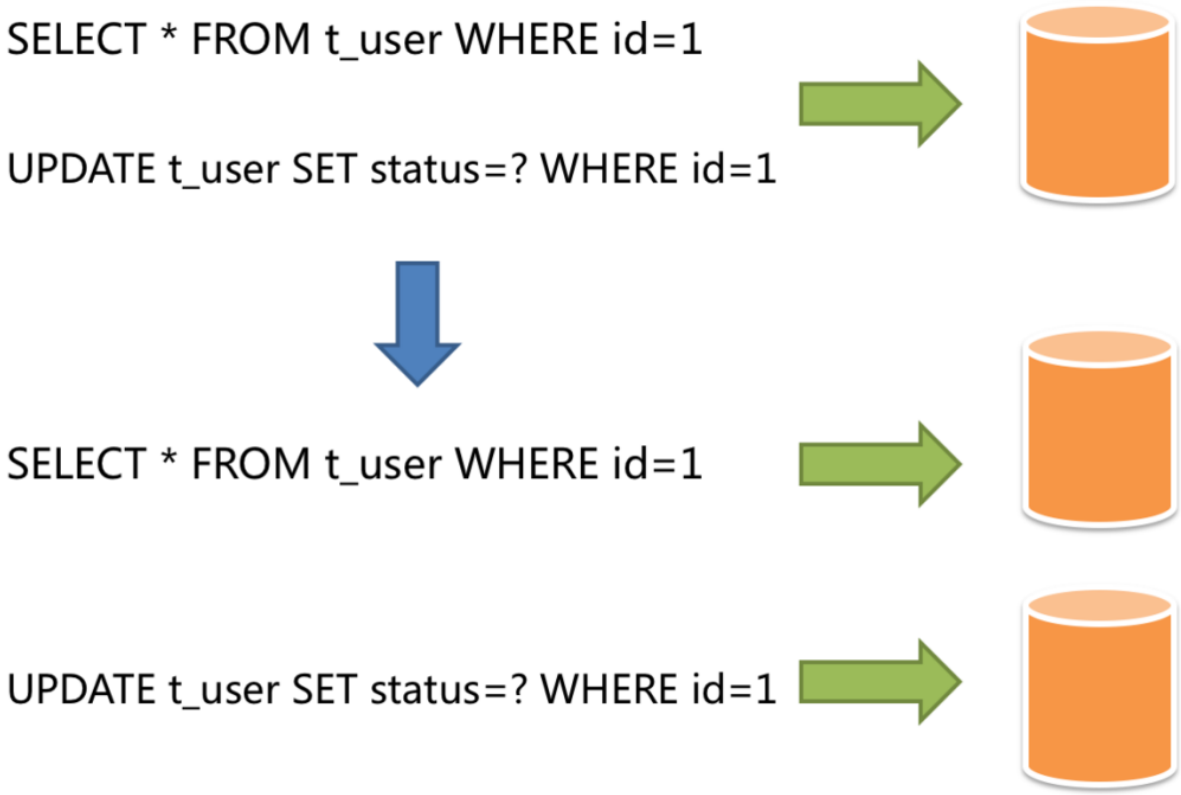
不支持项

路由至多数据节点

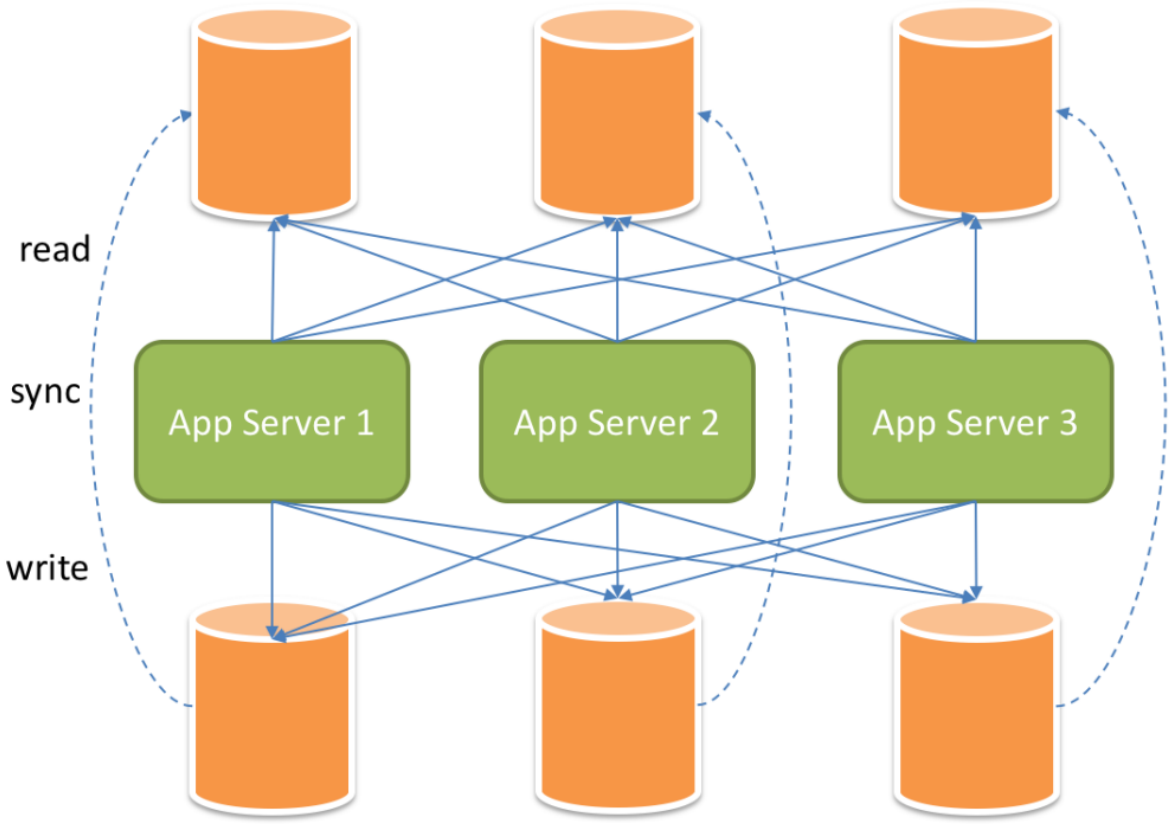
- 不支持CASE WHEN、HAVING、UNION (ALL)，有限支持子查询。

https://shardingsphere.apache.org/document/current/cn/features/sharding/use-norms/sql/

2.2. 读写分离



读写分离虽然可以提升系统的吞吐量和可用性，但同时也带来了数据不一致的问题。 这包括多个主库之间的数据一致性，以及主库与从库之间的数据一致性的问题。 并且，读写分离也带来了与数据分片同样的问题，它同样会使得应用开发和运维人员对数据库的操作和运维变得更加复杂。 下图展现了将分库分表与读写分离一同使用时，应用程序与数据库集群之间的复杂拓扑关系。



3. 示例：水平分库分片

引入maven依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-core</artifactId>
    <version>${sharding-sphere.version}</version>
</dependency>

```

或者

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

```

话不多说，上pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="
http://maven.apache.org/POM/4.0.0
<project xmlns="
http://www.w3.org/2001/XMLSchema-instance
<project xmlns="
http://maven.apache.org/POM/4.0.0
<project xmlns="
https://maven.apache.org/xsd/maven-4.0.0.xsd
<project xmlns="
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
```

```
    <groupId>com.cjs.example</groupId>
    <artifactId>sharding-jdbc-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>sharding-jdbc-demo</name>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <!--<dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>sharding-jdbc-core</artifactId>
            <version>4.1.1</version>
        </dependency>-->
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
            <version>4.1.1</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.22</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

OrderEntiry.java

```
package com.cjs.example.sharding.entity;
import lombok.Data;
import javax.persistence.*;
import java.io.Serializable;

@Entity
```

```
@Table(name = "t_order")
public class OrderEntity implements Serializable {

    @Id
    @Column(name = "order_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY) private Long orderId; private Integer userId; private Integer status = 1;
}

```

OrderRepository.java

```
package com.cjs.example.sharding.repository;
import com.cjs.example.sharding.entity.OrderEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<OrderEntity, Long> {
}

```

OrderService.java

```
package com.cjs.example.sharding.service;
import com.cjs.example.sharding.entity.OrderEntity;
import com.cjs.example.sharding.repository.OrderRepository;
import org.springframework.stereotype.Service;
import javax.annotation.Resource;

@Service
public class OrderService {

    @Resource private OrderRepository orderRepository; public void save(OrderEntity entity) {
        orderRepository.save(entity);
    }

}

```

OrderController.java

```
package com.cjs.example.sharding.controller;
import com.cjs.example.sharding.entity.OrderEntity;
import com.cjs.example.sharding.service.OrderService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController; /

@RequestMapping("/order")
public class OrderController {

    @Autowired private OrderService orderService;

    @GetMapping("/save") public String save(@RequestParam("userId") Integer userId) {
        OrderEntity entity = new OrderEntity();
        entity.setUserId(userId);
        orderService.save(entity); return "ok";
    }
}

```

启动类

```
package com.cjs.example.sharding;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration;
import javax.annotation.Resource;
import javax.sql.DataSource;
/** *
http://shardingsphere.apache.org/index.html
/** *
```

```
https://shardingsphere.apache.org/document/legacy/4.x/document/en/manual/sharding-jdbc/
/** *
http://shardingsphere.apache.org/index_zh.html
/** *
 */
@SpringBootApplication(exclude = JtaAutoConfiguration.class)
public class ShardingJdbcDemoApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(ShardingJdbcDemoApplication.class, args);
    }

    @Resource
    private DataSource dataSource;

    @Override
    public void run(String... args) throws Exception {
        System.out.println(dataSource);
    }
}

```

最最重要的是 application.properties

```
#
https://shardingsphere.apache.org/document/legacy/4.x/document/en/manual/sharding-jdbc/
#

# 配置真实数据源
spring.shardingsphere.datasource.names=ds0,ds1

# 配置第 1 个数据源
spring.shardingsphere.datasource.ds0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds0.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds0.url=jdbc:mysql://localhost:3306/ds0
spring.shardingsphere.datasource.ds0.username=root
spring.shardingsphere.datasource.ds0.password=123456

```

```
# 配置第 2 个数据源
spring.shardingsphere.datasource.ds1.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds1.url=jdbc:mysql://localhost:3306/ds1
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=123456

# 配置 t_order 表规则
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds$->{0..1}.t_order_$->{0..1}
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.sharding-column=order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.algorithm-expression=t_order_$->{order_id % 2}
spring.shardingsphere.sharding.tables.t_order.key-generator.type=SNOWFLAKE
spring.shardingsphere.sharding.tables.t_order.key-generator.column=order_id
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.sharding-column=user_id
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.algorithm-expression=ds$->{user_id % 2}

spring.shardingsphere.props.sql.show=true

```

通过访问http://localhost:8080/order/save?userId=xxx想数据库中插入数据，结果确实如预期的那样

```
mysql> select * from ds0.t_order_0;
+--------------------+----------+--------+
| order_id           | user_id  | status |
+--------------------+----------+--------+
| 480818987119149056 | 108584624 |      1 |
+--------------------+----------+--------+
1 row in set (0.00 sec)

mysql> select * from ds0.t_order_1;
+--------------------+----------+--------+
| order_id           | user_id  | status |
+--------------------+----------+--------+
| 480819017687236609 | 108582664 |      1 |
+--------------------+----------+--------+
1 row in set (0.00 sec)

mysql> select * from ds1.t_order_0;
+--------------------+----------+--------+
| order_id           | user_id  | status |
+--------------------+----------+--------+
| 480819055951872000 | 108584511 |      1 |
+--------------------+----------+--------+
1 row in set (0.00 sec)

mysql> select * from ds1.t_order_1;
+--------------------+----------+--------+
| order_id           | user_id  | status |
+--------------------+----------+--------+
| 480819080257863681 | 108586049 |      1 |
+--------------------+----------+--------+
1 row in set (0.00 sec)
```

4. 写在最后

配置入口类：

org.apache.shardingsphere.shardingjdbc.spring.boot.SpringBootConfiguration

文档在这里：

> https://shardingsphere.apache.org https://shardingsphere.apache.org/document/legacy/4.x/document/en/manual/sharding-jdbc http://shardingsphere.apache.org/elasticjob/

虽然 ShardingSphere-JDBC (Sharding-JDBC) 提供了很多功能，但是最常用的还是分库分表、读写分离，通常是一起用

> https://shardingsphere.apache.org/document/legacy/4.x/document/en/manual/sharding-jdbc/configuration/config-spring-boot/

分库分表以后，编写SQL时有诸多限制，很多之前在单库单表上的操作就不能用了，而且每次查询必须带上分片键，不然的话全表扫描

如果非要分表的话，不妨先考虑一下将数据存到ElasticSearch中，查询直接走ES。或者先走ES，然后通过主键再去查MySQL。