

## Sharding-JDBC介绍

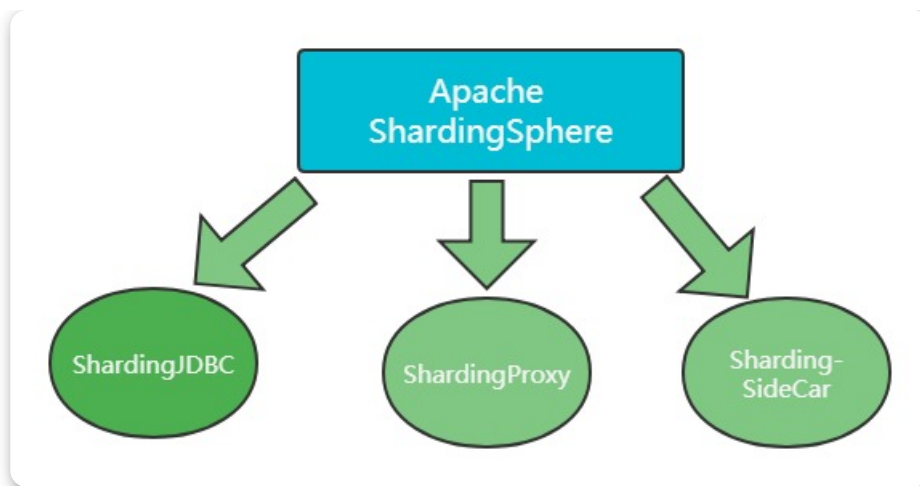
---



### 背景

随着通信技术的革新，全新领域的应用层出不穷，数据存量随着应用的探索不断增加，数据的存储和计算模式无时无刻不面临着创新。面向交易、大数据、关联分析、物联网等场景越来越细分，单一数据库再也无法适用于所有的应用场景。与此同时，场景内部也愈加细化，相似场景使用不同数据库已成为常态。由此可见，数据库碎片化的趋势已经不可逆转。

### ShardingJDBC是什么



Sharding-JDBC是Apache ShardingSphere生态圈中一款开源的分布式数据库第三方组件。ShardingSphere由它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（规划中）这3款相互独立的产品组成。它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，适用于Java同构、异构语言、容器、云原生等各种多样化的应用场景。

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 框架的使用。

**适用于任何基于Java的ORM框架**，如：JPA, Hibernate, Mybatis, Spring JDBC Template或直接使用JDBC。

**基于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和PostgreSQL。

## 主要功能

### ① 数据分片

- 分库
- 分表
- 读写分离
- 分片策略
- 分布式主键

### ② 分布式事务

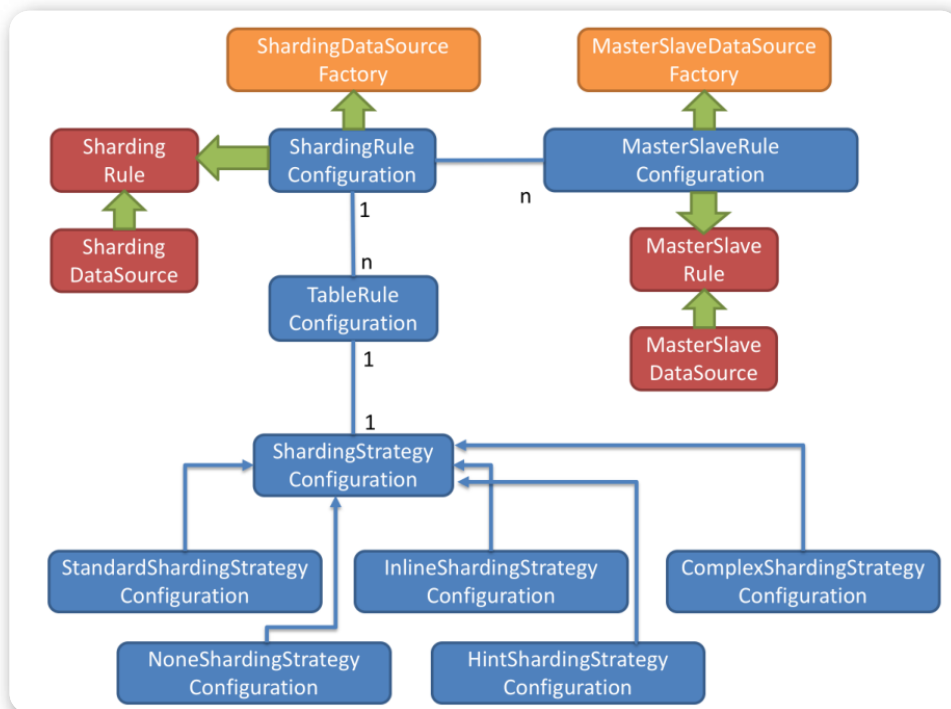
- 标准化的事务接口

XA强一致性事务  
柔性事务

### 3 数据库治理

配置动态化  
编排和治理  
数据脱敏  
可视化链路追踪

## 内部结构



- 1 图中黄色部分表示的是Sharding-JDBC的入口API，采用工厂方法的形式提供。目前有  
ShardingDataSourceFactory支持分库分表，读写分离操作  
MasterSlaveDataSourceFactory支持读写分离操作
- 2 图中蓝色部分表示的是Sharding-JDBC的配置对象，提供灵活多变的配置方式。  
TableRuleConfiguration,它包含分片配置规则  
MasterSlaveRuleConfiguration,它包含的是读写分离的配置规则  
ShardingRuleConfuguration, 主入口，它包含多个  
TableRuleConfiguration, 也可以包含多个  
MasterSlaveRuleConfiguration
- 3 图中红色部分表示的是内部对象，由Sharding-JDBC内部使用，应用开发者无需关注。

Shardingjdbc通过ShardingRuleConfuguration和MasterSlaveRuleConfiguration生成真正的规则对象，最终生成我们要使用的Datasource。

### Sharding-JDBC初始化流程：

根据配置信息生成configuration对象

通过Factory将configuration对象转化成Rule对象

通过Factory将Rule对象与DataSource对象进行封装

使用shardingjdbc进行分库分表操作

## Sharding-JDBC使用过程

### ① 引入maven依赖

```
<dependency>

  <groupId>org.apache.shardingsphere</groupId>
  <
    <artifactId>sharding-jdbc-
core</artifactId>
    <version>4.1.1</version>
  </dependency>
```

### ② 规则配置

Sharding-JDBC可以通过Java，YAML，Spring命名空间和Spring Boot Starter四种方式配置，开发者可根据场景选择适合的配置方式。

### ③ 创建DataSource

通过ShardingDataSourceFactory工厂和规则配置对象获取ShardingDataSource，然后即可通过DataSource选择使用原生JDBC开发，或者使用JPA, MyBatis等ORM工具。

```
DataSource dataSource =  
ShardingDataSourceFactory.createDataSource(data  
SourceMap,  
shardingRuleConfig, props);
```

## 实时效果反馈

### 1. 使用Sharding-JDBC时必须引入哪个包

- A** sharding-jdbc-spring
- B** sharding-jdbc-core
- C** sharding-jdbc-core-spring
- D** sharding-jdbc-spring-boot-starter

## 答案

1=>B

## 数据分片剖析实战



## 核心概念

- **真实表**  
数据库中真实存在的物理表。例如b\_order0、b\_order1
- **逻辑表**  
在分片之后，同一类表结构的名称（总称）。例如b\_order。

- **数据节点**

在分片之后，由数据源和数据表组成。例如ds0.b\_order1

- **绑定表**

指的是分片规则一致的关系表（主表、子表），例如b\_order和b\_order\_item，均按照order\_id分片，则此两个表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，可以提升关联查询效率。

b\_order:b\_order0,b\_order1

b\_order\_item: b\_order\_item0,b\_order\_item1

没有配置绑定关系，采用笛卡尔积关联查询4条sql语句

```
select * from b_order0 o left join
b_order_item0 i on o.order_id = i.order_id
where o.order_id in(10,11)
select * from b_order0 o left join
b_order_item1 i on o.order_id = i.order_id
where o.order_id in(10,11)
select * from b_order1 o left join
b_order_item0 i on o.order_id = i.order_id
where o.order_id in(10,11)
select * from b_order1 o left join
b_order_item1 i on o.order_id = i.order_id
where o.order_id in(10,11)
```

配置了绑定关系，只需要查2条sql语句

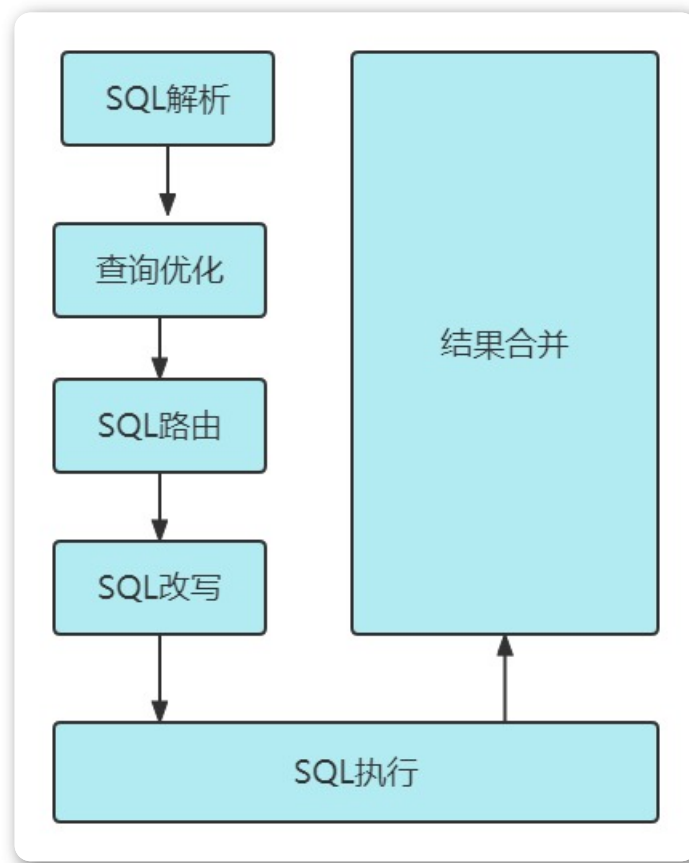
```
select * from b_order0 o left join
b_order_item0 i on o.order_id = i.order_id
where o.order_id in(10,11)
select * from b_order1 o left join
b_order_item1 i on o.order_id = i.order_id
where o.order_id in(10,11)
```

- **广播表**

在使用中，有些表没必要做分片，例如字典表、省份信息等，因为他们数据量不大，而且这种表可

能需要与海量数据的表进行关联查询。广播表会在不同的数据节点上进行存储，存储的表结构和数据完全相同。

## 数据分片流程解析



**SQL解析**分为词法解析和语法解析。先通过词法解析器将SQL拆分为一个个不可再分的单词。再使用语法解析器对SQL进行理解，并最终提炼出解析上下文。

Sharding-JDBC采用不同的解析器对SQL进行解析，解析器类型如下：

Mysql解析器

Oracle解析器

SQLSERVER解析器

PostgreSql解析器

默认解析器 sql-92标准

## 查询优化

负责合并和优化分片条件，如OR等。

## SQL路由

根据解析上下文匹配用户配置的分片策略，并生成路由路径。目前支持分片路由和广播路由。

## SQL改写

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。

## SQL执行

通过多线程执行器异步执行SQL。

## 结果归并

将多个执行结果集归并以便于通过统一的JDBC接口输出。结果归并包括流式归并、内存归并和使用装饰者模式的追加归并这几种方式。

## SQL使用规范

兼容全部常用的路由至单数据节点的SQL；路由至多数据节点的SQL由于场景复杂，分为稳定支持、实验性支持和不支持这三种情况

### ① 稳定支持

全面支持 DQL、DML、DDL、DCL、TCL 和常用 DAL。支持分页、去重、排序、分组、聚合、表关联等复杂查询。

#### 常规查询

- SELECT 主语句

```
SELECT select_expr [, select_expr ...]  
FROM table_reference [, table_reference  
...]  
[WHERE predicates]  
[GROUP BY {col_name | position} [ASC |  
DESC], ...]  
[ORDER BY {col_name | position} [ASC |  
DESC], ...]  
[LIMIT {[offset,] row_count | row_count  
OFFSET offset}]
```

- select\_expr



```
* |
[DISTINCT] COLUMN_NAME [AS] [alias] |
(MAX | MIN | SUM | AVG)(COLUMN_NAME |
alias) [AS] [alias] |
COUNT(* | COLUMN_NAME | alias) [AS]
[alias]
```

- table\_reference

```
tbl_name [AS] alias] [index_hint_list]
| table_reference ([INNER] | {LEFT|RIGHT}
[OUTER]) JOIN table_factor [JOIN ON
conditional_expr | USING (column_list)]
```

## 子查询

- 子查询和外层查询同时指定分片键，且分片键的值保持一致时，由内核提供稳定支持。

```
SELECT * FROM (SELECT * FROM t_order WHERE
order_id = 1) o WHERE o.order_id = 1;
```

## 2 实验性支持

实验性支持特指使用 Federation 执行引擎提供支持。该引擎处于快速开发中，用户虽基本可用，但仍需大量优化，是实验性产品。

- 子查询

子查询和外层查询未同时指定分片键，或分片键的值不一致时，由 Federation 执行引擎提供支持，例如：

```
SELECT * FROM (SELECT * FROM t_order) o;
SELECT * FROM (SELECT * FROM t_order) o
WHERE o.order_id = 1;
SELECT * FROM (SELECT * FROM t_order WHERE
order_id = 1) o;
SELECT * FROM (SELECT * FROM t_order WHERE
order_id = 1) o WHERE o.order_id = 2;
```

- 跨库关联查询

当关联查询中的多个表分布在不同的数据库实例上时，由 Federation 执行引擎提供支持。假设 t\_order 和 t\_order\_item 是多数据节点的分片表，并且未配置绑定表规则，t\_user 和 t\_user\_role 是分布在不同的数据库实例上的单表，那么 Federation 执行引擎能够支持如下常用的关联查询。

```
SELECT * FROM t_order o INNER JOIN
t_order_item i ON o.order_id = i.order_id
WHERE o.order_id = 1;
SELECT * FROM t_order o INNER JOIN t_user u
ON o.user_id = u.user_id WHERE o.user_id =
1;
SELECT * FROM t_order o LEFT JOIN
t_user_role r ON o.user_id = r.user_id WHERE
o.user_id = 1;
SELECT * FROM t_order_item i LEFT JOIN
t_user u ON i.user_id = u.user_id WHERE
i.user_id = 1;
SELECT * FROM t_order_item i RIGHT JOIN
t_user_role r ON i.user_id = r.user_id WHERE
i.user_id = 1;
SELECT * FROM t_user u RIGHT JOIN
t_user_role r ON u.user_id = r.user_id WHERE
u.user_id = 1;
```

- ③ 不支持

以下 CASE WHEN 语句不支持：

CASE WHEN 中包含子查询

CASE WHEN 中使用逻辑表名（请使用表别名）

不支持的 SQL	原因	解决方案
INSERT INTO tbl_name (col1, col2, ...) SELECT * FROM tbl_name WHERE col3 = ?	SELECT 子句不支持 * 和内置 分布式主键生成器	无
REPLACE INTO tbl_name (col1, col2, ...) SELECT * FROM tbl_name WHERE col3 = ?	SELECT 子句不支持 * 和内置 分布式主键生成器	无
SELECT MAX(tbl_name.col1) FROM tbl_name	查询列是函数表达式时，查询 列前不能使用表名	使用 表别名

## 行表达式 (Inline)



Inline是可以简化数据节点和分片算法配置信息，主要是解决配置简化、配置一体化。

### ① inline 表达式说明

`${begin..end}` 表示范围区间，`${[unit1, unit2, unitX]}` 表示枚举值。inline 表达式中连续多个 `${...}` 表达式，整个 inline 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，例如正式表 inline 表达式如下：

```
dbtbl_${['online', 'offline']}_${1..3}
dbtbl_${->{['online','offline']}_>{1..3}}
```

最终会解析为 dbtbl\_online\_1, dbtbl\_online\_2, dbtbl\_online\_3, dbtbl\_offline\_1, dbtbl\_offline\_2和 dbtbl\_offline\_3 这 6 张表。

## ② 数据节点配置

字符串中使用 `${}` 来嵌入 groovy 代码，下面的表达式中 `data_source_` 是字符串前缀，`id % 2 + 1` 是 groovy 代码。

```
data_source_${id % 2 + 1}
```

结果为：data\_source\_1、data\_source\_2

## ③ db0,db1, 每个库下面都有order0,order1两张表，行表达式配置如下

```
db${0..1}.order${0..1}
```

db0下面有order0,order1两张表，db1下面有 order2,order3,order4四张表，配置如下

```
db0.order${0..1},db1.order${2..4}
```

## 实时效果反馈

2. 下列ShardingJdbc**不支持**的sql语句是哪个？

**A** SELECT \* FROM (SELECT \* FROM t\_order) o WHERE o.order\_id = 1;

**B** SELECT USER\_id,USER\_NAME, (CASE WHEN SEX=1 THEN '男' WHEN SEX=2 THEN '女' END) AS SEX FROM SYS\_USER

**C** SELECT \* FROM t\_order o INNER JOIN t\_order\_item i ON o.order\_id = i.order\_id WHERE o.order\_id = 1;

**D** SELECT \* FROM t\_order\_item i RIGHT JOIN t\_user\_role r ON i.user\_id = r.user\_id WHERE i.user\_id = 1;

答案

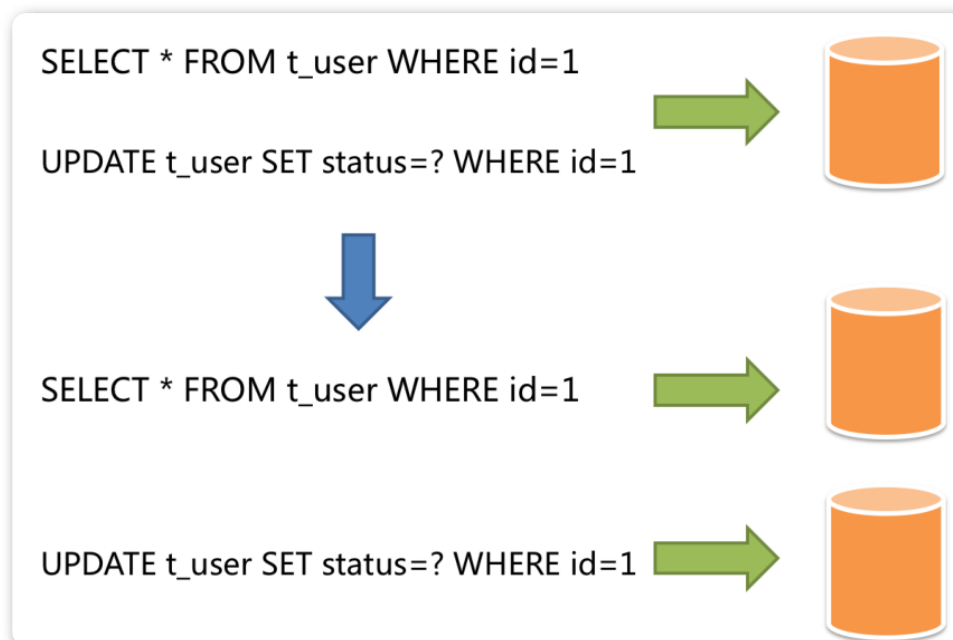
2=>B

## 读写分离剖析实战



### 读写分离概念

读写分离是通过主从的配置方式，将查询请求均匀分散到多个数据副本，进一步的提升系统的处理能力。



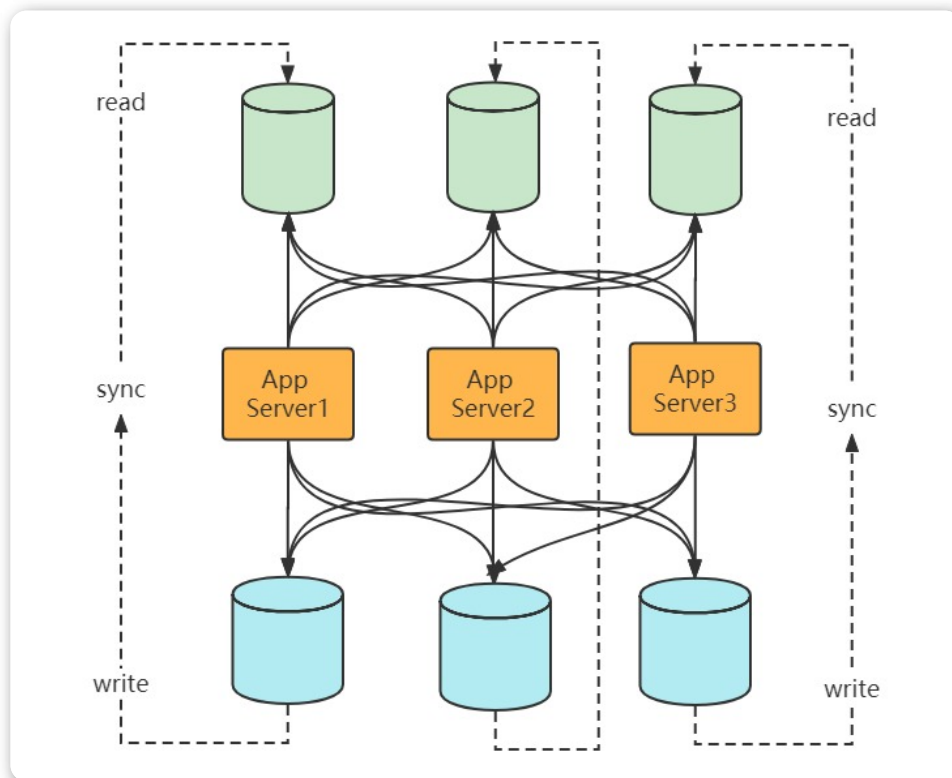
主库：添加、更新以及删除数据操作所使用的数据库

从库：查询数据操作所使用的数据库

主从架构：读写分离，目的是高可用、读写扩展。主从库内容相同，根据SQL语义进行路由。

分库分表架构：数据分片，目的读写扩展、存储扩容，库和表内容不同，根据分片配置进行路由。

将水平分片和读写分离联合使用，能够更加有效的提升系统性能，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。



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，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。

## 读写分离配置

按照一主一从的方式，先配置主库，打开主库/etc/my.cnf文件

```
For advice on how to change settings please see
#
http://dev.mysql.com/doc/refman/5.7/en/server-configuration-defaults.html
```

```
[mysqld]
```

```
log_bin=mysql_bin_log
skip-name-resolve
character_set_server=utf8

#
# Remove leading # and set to the amount of RAM
for the most important data
# cache in MySQL. Start at 70% of total RAM for
dedicated server, else 10%.
# innodb_buffer_pool_size = 128M
#
# Remove leading # to turn on a very important
data integrity option: logging
# changes to the binary log between backups.
# log_bin

server-id=10
#
# Remove leading # to set options mainly useful
for reporting servers.
# The server defaults are faster for
transactions and fast SELECTs.
# Adjust sizes as needed, experiment to find
the optimal values.
# join_buffer_size = 128M
# sort_buffer_size = 2M
# read_rnd_buffer_size = 2M
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

# Disabling symbolic-links is recommended to
prevent assorted security risks
```

```
symbolic-links=0
```

```
log-error=/var/log/mysqld.log
```

```
pid-file=/var/run/mysqld/mysqld.pid
```

```
binlog-do-db=sxt_his
```

```
binlog-ignore-db=performance_schema
```

```
binlog-ignore-db=information_schema
```

打开从库my.cnf文件

```
# For advice on how to change settings please  
see
```

```
#
```

```
http://dev.mysql.com/doc/refman/5.7/en/server-  
configuration-defaults.html
```

```
[mysqld]
```

```
log_bin=mysql_bin_log
```

```
server-id=11
```

```
skip-name-resolve
```

```
character_set_server=utf8
```

```
#
```

```
# Remove leading # and set to the amount of RAM  
for the most important data
```

```
# cache in MySQL. Start at 70% of total RAM for  
dedicated server, else 10%.
```

```
# innodb_buffer_pool_size = 128M
```

```
#
```

```
# Remove leading # to turn on a very important  
data integrity option: logging
```



```
# changes to the binary log between backups.
# log_bin
#
# Remove leading # to set options mainly useful
for reporting servers.
# The server defaults are faster for
transactions and fast SELECTs.
# Adjust sizes as needed, experiment to find
the optimal values.
# join_buffer_size = 128M
# sort_buffer_size = 2M
# read_rnd_buffer_size = 2M
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

default_time_zone=+8:00

# Disabling symbolic-links is recommended to
prevent assorted security risks
symbolic-links=0

log-error=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
```

主从设置，首先进入主库mysql，执行以下命令查看主库状态

```
show master status;
```

执行后效果如下：

File	Position
Binlog_Do_DB	Binlog_Ignore_DB
+-----+-----+-----	
--+	
mysql_bin_log.000017	378990
performance_schema,information_schema,sys	

进入从库MySQL，执行以下命令

```
change master to
master_host='masterIp',master_port=3306,master_
user='root',master_password='password',master_l
og_file='mysql_bin_log.000017',master_log_pos=3
78990;
```

完成后，启动从库

```
start slave;
```

执行如下命令查看从库状态

```
mysql> show slave status \G
***** 1. row
*****

Slave_IO_State: waiting for
master to send event

Master_Host: 10.13.181.208
Master_User: root
Master_Port: 3306
Connect_Retry: 60
Master_Log_File:
mysql_bin_log.000017
Read_Master_Log_Pos: 378990
```

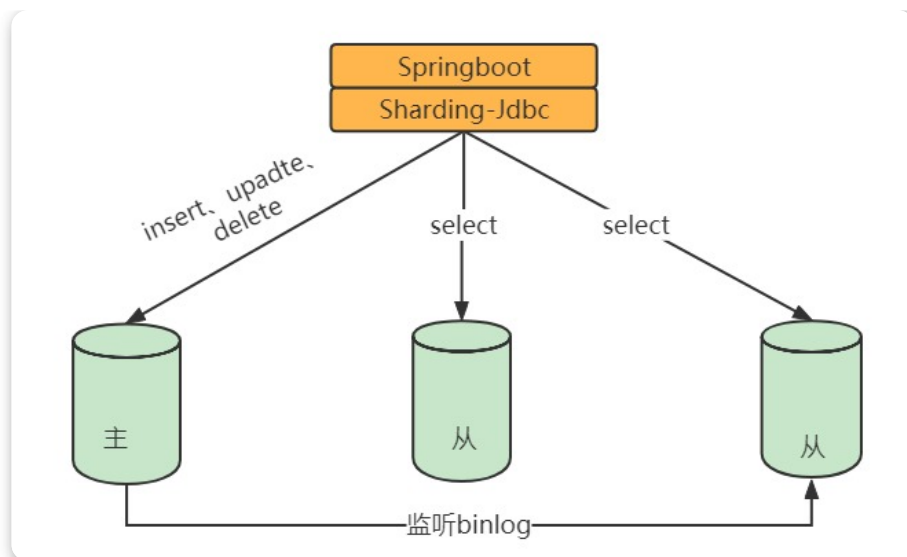
Relay\_Log\_File: 10-13-53-213-relay-bin.000004  
Relay\_Log\_Pos: 14008  
Relay\_Master\_Log\_File: mysql\_bin\_log.000017  
Slave\_IO\_Running: Yes  
Slave\_SQL\_Running: Yes  
Replicate\_Do\_DB:  
Replicate\_Ignore\_DB:  
Replicate\_Do\_Table:  
Replicate\_Ignore\_Table:  
Replicate\_Wild\_Do\_Table:  
Replicate\_Wild\_Ignore\_Table:  
Last\_Errno: 0  
Last\_Error:  
Skip\_Counter: 0  
Exec\_Master\_Log\_Pos: 378990  
Relay\_Log\_Space: 14222  
Until\_Condition: None  
Until\_Log\_File:  
Until\_Log\_Pos: 0  
Master\_SSL\_Allowed: No  
Master\_SSL\_CA\_File:  
Master\_SSL\_CA\_Path:  
Master\_SSL\_Cert:  
Master\_SSL\_Cipher:  
Master\_SSL\_Key:  
Seconds\_Behind\_Master: 0  
Master\_SSL\_Verify\_Server\_Cert: No  
Last\_IO\_Errno: 0  
Last\_IO\_Error:  
Last\_SQL\_Errno: 0

```
        Last_SQL_Error:
Replicate_Ignore_Server_Ids:
        Master_Server_Id: 10
                Master_UUID: 90889a22-2288-
11ec-8134-525400671b87
        Master_Info_File:
/var/lib/mysql/master.info
                SQL_Delay: 0
        SQL_Remaining_Delay: NULL
Slave_SQL_Running_State: slave has read
all relay log; waiting for more updates
        Master_Retry_Count: 86400
                Master_Bind:
        Last_IO_Error_Timestamp:
        Last_SQL_Error_Timestamp:
        Master_SSL_Crl:
        Master_SSL_Crlpath:
        Retrieved_Gtid_Set:
        Executed_Gtid_Set:
                Auto_Position: 0
        Replicate_Rewrite_DB:
                Channel_Name:
        Master_TLS_Version:
1 row in set (0.00 sec)
```

## 读写分离方案剖析

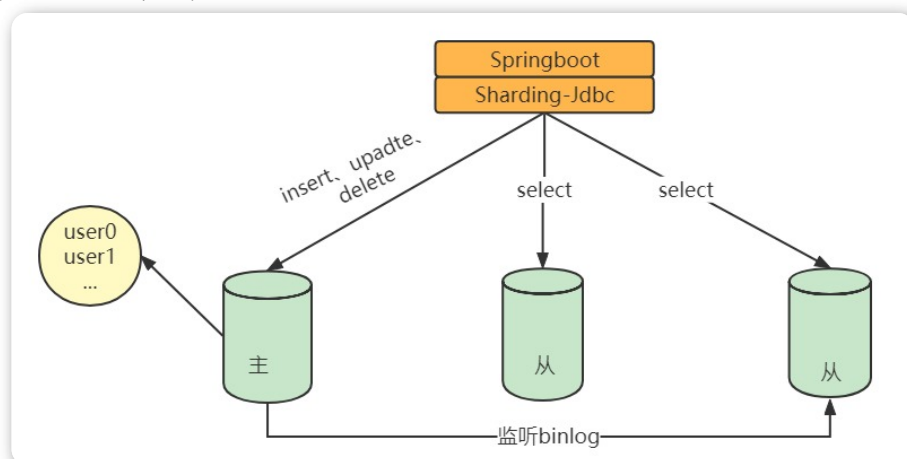
### ① 分库+读写分离

在数据量不多的情况下，我们可以将数据库进行读写分离，以应对高并发的需求，通过水平扩展从库，来缓解查询的压力。



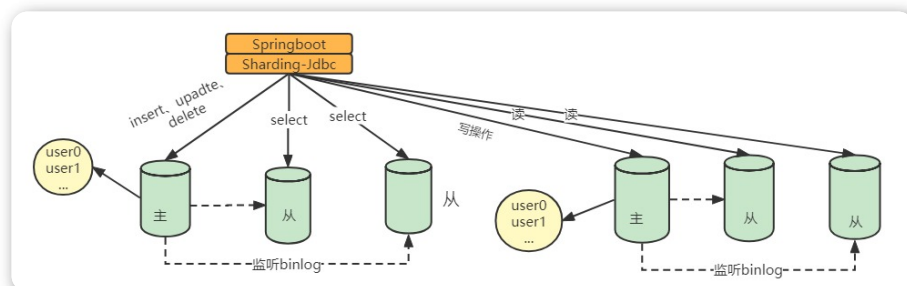
## 2 分表+读写分离

在数据量达到500万的时候，这时数据量预估千万级别，我们可以将数据进行分表存储。



## 3 分库分表+读写分离

在数据量继续扩大，这时可以考虑分库分表，将数据存储在不同数据库的不同表中



透明化读写分离所带来的影响，让使用方尽量像使用一个数据库一样使用主从数据库集群，是ShardingSphere读写分离模块的主要设计目标。

## 核心功能

提供一主多从的读写分离配置。仅支持单主库，可以支持独立使用，也可以配合分库分表使用

独立使用读写分离，支持SQL透传。不需要SQL改写流程

同一线程且同一数据库连接内，能保证数据一致性。如果有写入操作，后续的读操作均从主库读取。

基于Hint的强制主库路由。可以强制路由走主库查询实时数据，避免主从同步数据延迟。

## 不支持项

主库和从库的数据同步

主库和从库的数据同步延迟

主库双写或多写

跨主库和从库之间的事务的数据不一致。建议在主从架构中，事务中的读写均用主库操作。

## 实时效果反馈

3. 下列哪个不属于读写分离负载均衡策略？

**A** 随机

**B** 权重

**C** 轮询

**D** ip-hash

## 答案

3=>D

## 强制路由剖析实战

---



在一些应用场景中，分片条件并不存在于SQL，而存在于外部业务逻辑。因此需要提供一种通过在外业务代码中指定路由配置的一种方式，在ShardingSphere中叫做Hint。如果使用hint强制路由，那么sql将无视原有的分片逻辑，直接将路由到指定的数据节点上操作。

## 应用场景

Hint使用场景：

数据分片操作，如果分片键没有在SQL或数据表中，而是在业务逻辑代码中；

读写分离操作，如果强制在主库进行某些数据操作；

## 应用过程

- 编写分库或分表路由策略，实现HintShardingAlgorithm接口

```

public class MyHintShardingAlgorithm implements
HintShardingAlgorithm<Integer> {
    @Override
    public Collection<String>
doSharding(Collection<String> collection,
HintShardingValue<Integer> hintShardingValue) {
        //添加分库或分表路由逻辑
    }
}

```

- 在配置文件指定分库或分表策略

### #强制路由库

```

spring.shardingsphere.sharding.tables.t_user.d
atabase-strategy.hint.algorithm-class-
name=com.itbaizhan.hint.MyHintShardingAlgorithm

```

### #强制路由库和表

```

spring.shardingsphere.sharding.tables.b_order.d
atabase-strategy.hint.algorithm-class-
name=com.itbaizhan.hint.MyHintShardingAlgorithm
spring.shardingsphere.sharding.tables.b_order.t
able-strategy.hint.algorithm-class-
name=com.itbaizhan.hint.MyHintShardingAlgorithm
spring.shardingsphere.sharding.tables.b_order.a
ctual-data-nodes=ds$->{0..1}.b_order$->{0..1}

```

- 在代码执行查询前使用HintManager指定执行策略值



```

@Test//路由库和表
public void test(){
    HintManager hintManager =
HintManager.getInstance();

    hintManager.addDatabaseShardingValue("b_order"
,1);

    hintManager.addTableShardingValue("b_order",1)
;
    List<Order> list =
orderRepository.findAll();
    hintManager.close();
    list.forEach(o -> {
        System.out.println(o.getOrderID()+"
"+o.getUserId()+" "+o.getOrderPrice());
    });
}

```

在读写分离结构中，为了避免主从同步数据延迟及时获取刚添加或更新的数据，可以采用强制路由走主库查询实时数据，使用 `hintManager.setMasterRouteOnly` 设置主库路由即可。

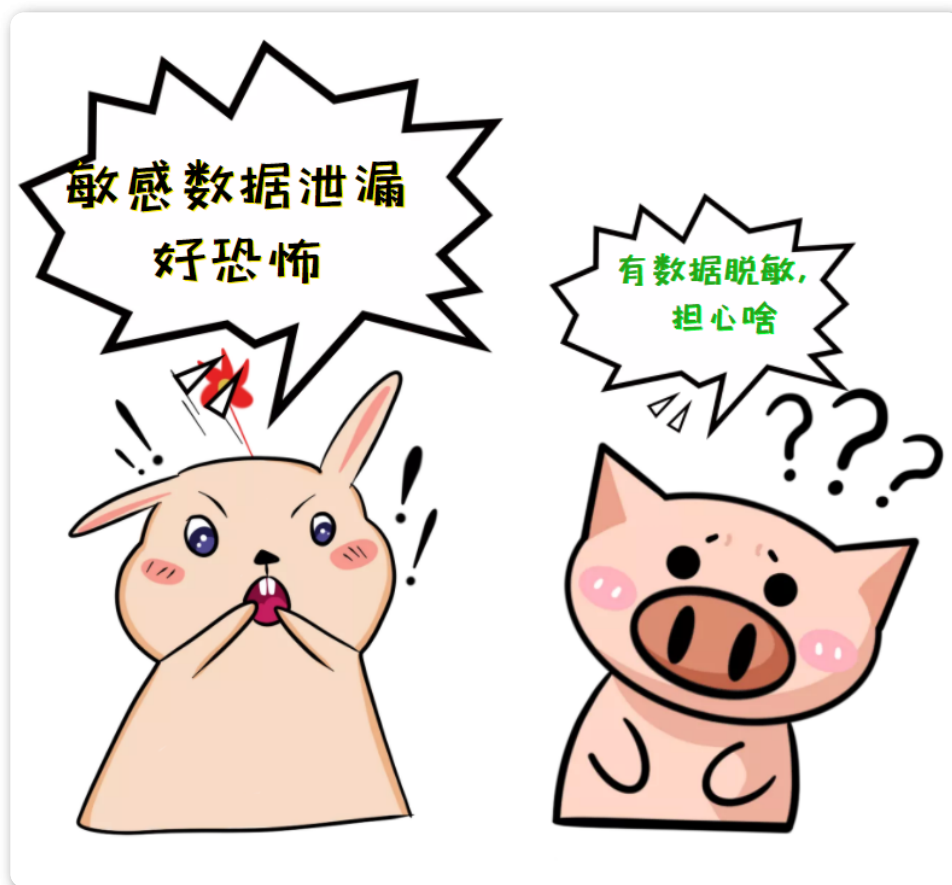
## 实时效果反馈

4. 自定义强制路由策略需要实现以下哪个接口

- ☒ A ComplexKeyShardingAlgorithm
- ☐ B PreciseShardingAlgorithm
- ☐ C HintShardingAlgorithm
- ☐ D RangeShardingAlgorithm

**答案**

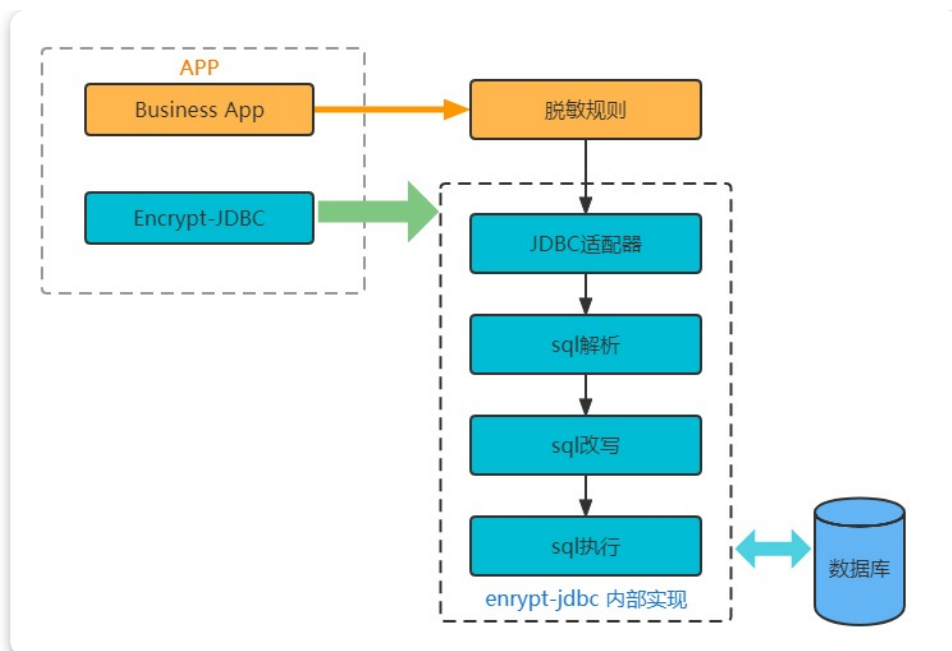
## 数据脱敏剖析实战



数据脱敏是指对某些敏感信息通过脱敏规则进行数据的变形，实现敏感隐私数据的可靠保护。涉及客户安全数据或者一些商业性敏感数据，如身份证号、手机号、卡号、客户号等个人信息按照规定，都需要进行数据脱敏。数据脱敏模块属于ShardingSphere分布式治理这一核心功能下的子功能模块功能。

- 在更新操作时，它通过对用户输入的SQL进行解析，并依据用户提供的脱敏配置对SQL进行改写，从而实现对原文数据进行加密，并将密文数据存储到底层数据库。
- 在查询数据时，它又从数据库中取出密文数据并对其解密，最终将解密后的原始数据返回给用户。

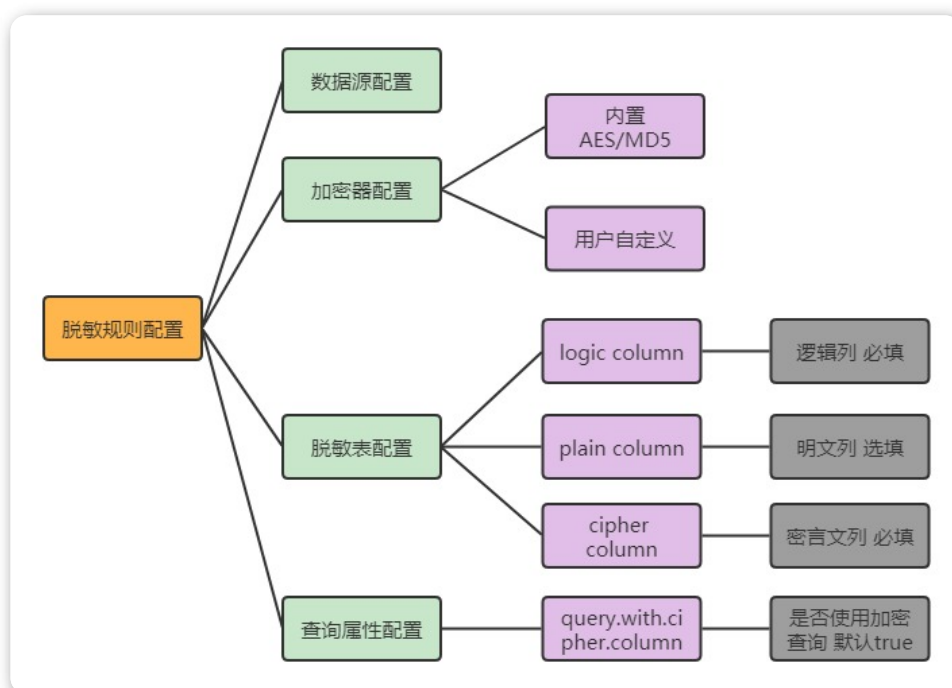
### 整体架构



Encrypt-JDBC将用户发起的SQL进行拦截，并通过SQL语法解析器进行解析、理解SQL行为，再依据用户传入的脱敏规则，找出需要脱敏的字段和所使用的加解密器对目标字段进行加解密处理后，再与底层数据库进行交互。

## 脱敏规则

脱敏配置主要分为四部分：数据源配置，加密器配置，脱敏表配置以及查询属性配置，其详情如下图所示：



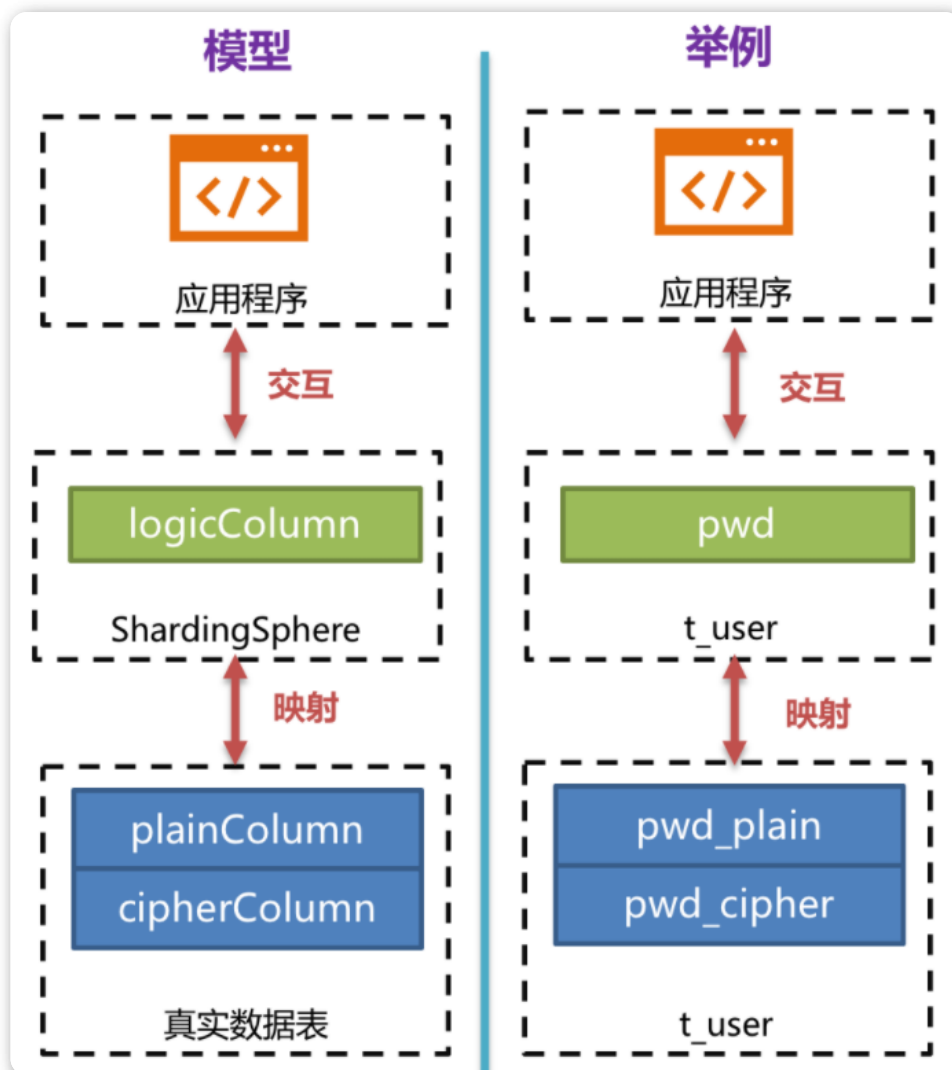
- **数据源配置**：指数据源配置。
- **加密算法配置**：指使用什么加密算法进行加解密。目前ShardingSphere内置了三种加解密算法：AES，MD5 和 RC4。用户还可以通过实现ShardingSphere提供的接口，自行实现一套加解密算

法。

- **加密表配置**：用于告诉ShardingSphere数据表里哪个列用于存储密文数据（cipherColumn）、哪个列用于存储明文数据（plainColumn）以及用户想使用哪个列进行SQL编写（logicColumn）。

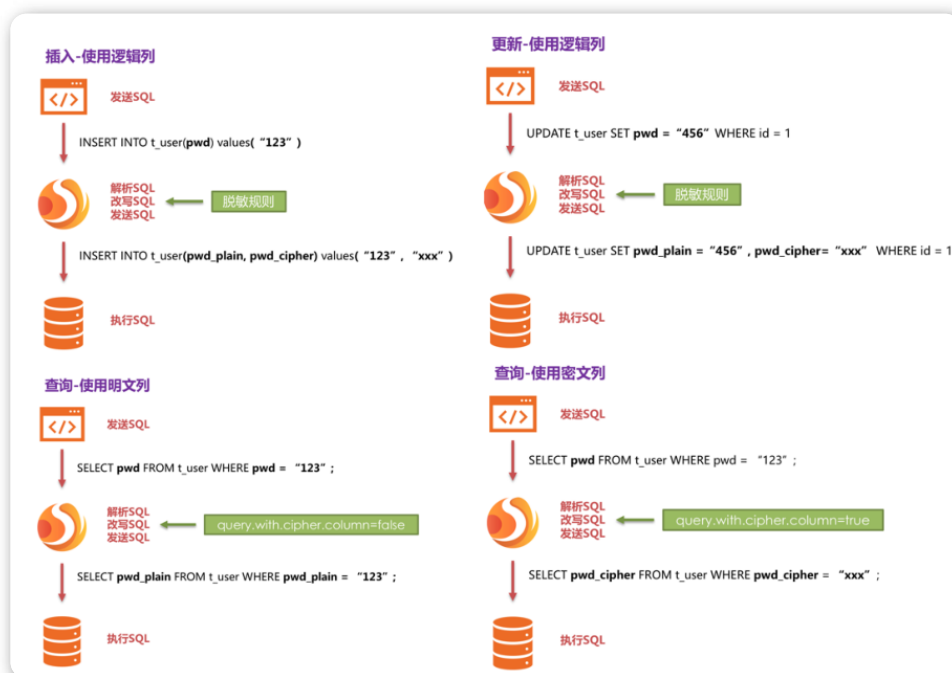
## 脱敏处理流程

举例说明，假如数据库里有一张表叫做 t\_user，这张表里有两个字段 pwd\_plain，用于存放明文数据、pwd\_cipher，用于存放密文数据，同时定义 logicColumn 为 pwd。那么，用户在编写 SQL 时应该面向 logicColumn 进行编写，即 INSERT INTO t\_user SET pwd = '123'。Apache ShardingSphere 接收到该SQL，通过用户提供的加密配置，发现 pwd是 logicColumn，于是便对逻辑列及其对应的明文数据进行加密处理。**Apache ShardingSphere 将面向用户的逻辑列与面向底层数据库的明文列和密文列进行了列名以及数据的加密映射转换。** 如下图所示：



即依据用户提供的加密规则，将用户 SQL 与底层数据表结构割裂开来，使得用户的 SQL 编写不再依赖于真实的数据库表结构。而用户与底层数据库之间的衔接、映射、转换交由 Apache ShardingSphere 进行处理。

下方图片展示了使用加密模块进行增删改查时，其中的处理流程和转换逻辑，如下图所示，



## 加密策略

ShardingSphere提供了两种加密策略用于数据脱敏，该两种策略分别对应ShardingSphere的两种加解密的接口，即Encryptor和QueryAssistedEncryptor。

一方面，Apache ShardingSphere 为用户提供了内置的加解密实现类，用户只需进行配置即可使用；另一方面，为了满足用户不同场景的需求，我们还开放了相关加解密接口，用户可依据这两种类型的接口提供具体实现类。再进行简单配置，即可让 Apache ShardingSphere 调用用户自定义的加解密方案进行数据加密。

## 实时效果反馈

### 5.shardingJdbc数据脱敏流程不包含哪个步骤

**A** sql解析

B sql改写

C 执行sql

D sql路由

答案

5=>D

## 分布式事务剖析实战



## 分布式事务理论

- CAP (强一致性), 对于共享数据系统, 最多只能同时拥有CAP其中的两个, 任意两个都有其适应的场景。

选项	描述
C (consistence)	所有的节点上的数据时刻保持同步
A (availiblity)	每个请求都能接受到一个响应, 无论响应成功或失败
P (partition tolerance)	系统应该能持续提供服务, 即使系统内部有消息丢失

- BASE (最终一致性), BASE 是指基本可用 (Basically Available)、软状态 (Soft State)、最终一致性 (Eventual Consistency)。它的核心思想是即使无法做到强一致性 (CAP 就是强一致

性)，但应用可以采用适合的方式达到最终一致性。

## 分布式事务模式

了解了分布式事务中的强一致性和最终一致性理论，下面介绍几种常见的分布式事务的解决方案。

- 2PC模式（强一致性）

2PC是Two-Phase Commit缩写，即两阶段提交，就是将事务的提交过程分为两个阶段来进行处理。事务的发起者称协调者，事务的执行者称参与者。协调者统一协调参与者执行。

阶段1：准备阶段，协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。各参与者执行事务操作，但不提交事务，将 undo 和 redo 信息记入事务日志中。如参与者执行成功，给协调者反馈 yes；如执行失败，给协调者反馈 no。

阶段2：提交阶段，如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(rollback)消息；否则，发送提交(commit)消息。

2PC 方案实现起来简单，实际项目中使用比较少，主要因为以下问题：

性能问题：所有参与者在事务提交阶段处于同步阻塞状态，占用系统资源，容易导致性能瓶颈。

可靠性问题：如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定状态。

数据一致性问题：在阶段 2 中，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

- 3PC模式（强一致性）

3PC (三阶段提交)，是两阶段提交的改进版本，与两阶段提交不同的是，引入超时机制。同时在协调者和参与者中都引入超时机制。三阶段提交将两阶段的准备阶段拆分为 2 个阶段，插入了一个preCommit 阶段，解决了原先在两阶段提交中，参与者在准备之后，由于协调者或参与者发生崩溃或错误，而导致参与者无



法知晓处于长时间等待的问题。如果在指定的时间内协调者没有收到参与者的消息则默认失败。

阶段1: canCommit, 协调者向参与者发送 commit 请求, 参与者如果可以提交就返回 yes 响应, 否则返回 no 响应。

阶段2: preCommit, 协调者根据阶段 1 canCommit 参与者的反应情况执行预提交事务或中断事务操作。

参与者均反馈 yes: 协调者向所有参与者发出 preCommit 请求, 参与者收到

preCommit 请求后, 执行事务操作, 但不提交; 将 undo 和 redo 信息记入事务日志

中; 各参与者向协调者反馈 ack 响应或 no 响应, 并等待最终指令。任何一个参与者反馈 no或等待超时: 协调者向所有参与者发出 abort 请求, 无论收到协调者发出的 abort 请求, 或者在等待协调者请求过程中出现超时, 参与者均会中断事务。

阶段3: do Commit

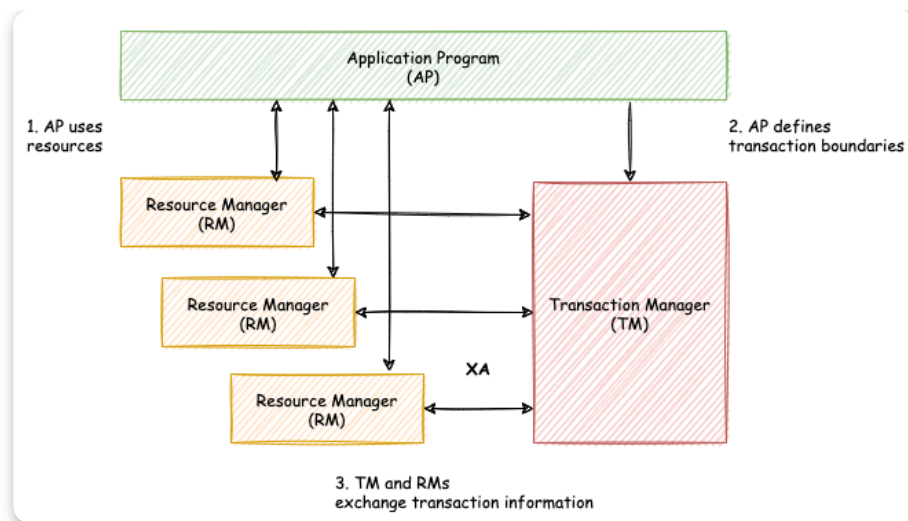
该阶段进行真正的事务提交, 根据阶段 2 preCommit反馈的结果完成事务提交或中断操作。

相比2PC模式, 3PC模式降低了阻塞范围, 在等待超时后协调者或参与者会中断事务。避免了协调者单点问题, 阶段 3 中协调者出现问题时 (比如网络中断等), 参与者会继续提交事务。

- XA (强一致性)

XA是由X/Open组织提出的分布式事务的规范, 是基于两阶段提交协议。XA规范主要定义了全局事务管理器(TM)和局部资源管理器(RM)之间的接口。目前主流的关系型数据库产品都是实现了XA接口。





XA之所以需要引入事务管理器，是因为在分布式系统中，从理论上讲两台机器理论上无法达到一致的状态，需要引入一个单点进行协调。由全局事务管理器管理和协调的事务，可以跨越多个资源（数据库）和进程。

事务管理器用来保证所有的事务参与者都完成了准备工作(第一阶段)。如果事务管理器收到所有参与者都准备好的消息，就会通知所有的事务都可以提交了（第二阶段）。MySQL 在这个XA事务中扮演的是参与者的角色，而不是事务管理器。

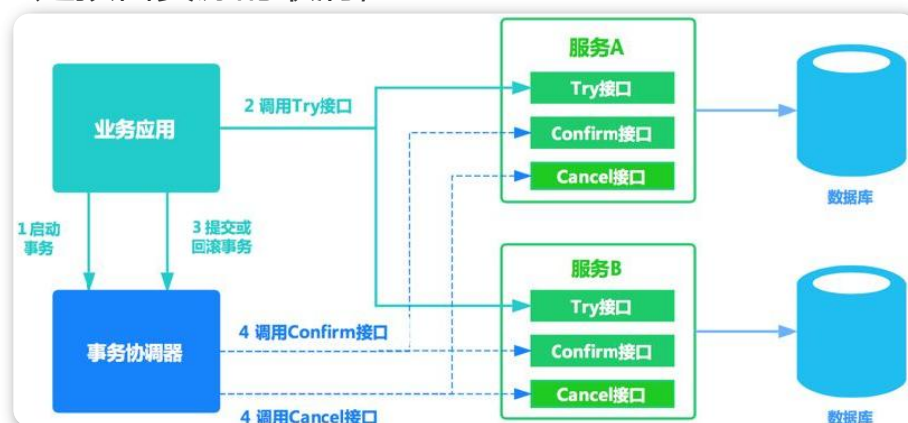
- TCC模式（最终一致性）

TCC (Try-Confirm-Cancel) 的概念，最早是由 Pat Helland 于 2007 年发表的一篇名为《Lifebeyond Distributed Transactions:an Apostate's Opinion》的论文提出。TCC 是服务化的两阶段编程模型，其 Try、Confirm、Cancel 3 个方法均由业务编码实现：

Try 操作作为一阶段，负责资源的检查和预留；

Confirm 操作作为二阶段提交操作，执行真正的业务；

Cancel 是预留资源的取消；



TCC 模式相比于 XA，解决了如下几个缺点：

解决了协调者单点，由业务应用发起并完成这个业务活动。业务活动管理器可以变成多点，引入集群。

同步阻塞：引入超时机制，超时后进行补偿，并且不会锁定整个资源，将资源转换为业务逻辑形式，粒度变小。

数据一致性，有了补偿机制之后，由业务活动管理器控制一致性。

## Sharding-JDBC整合XA数据源

XAShardingSphereTransactionManager为Apache ShardingSphere 的分布式事务的 XA 实现类。它主要负责对多数据源进行管理和适配，并且将相应事务的开启、提交和回滚操作委托给具体的 XA 事务管理器。ShardingSphere整合XA事务时，分离了XA事务管理和连接池管理，这样接入XA时，可以做到对业务的零侵入。

ShardingSphere支持以下功能：

支持数据分片后的跨库XA事务；

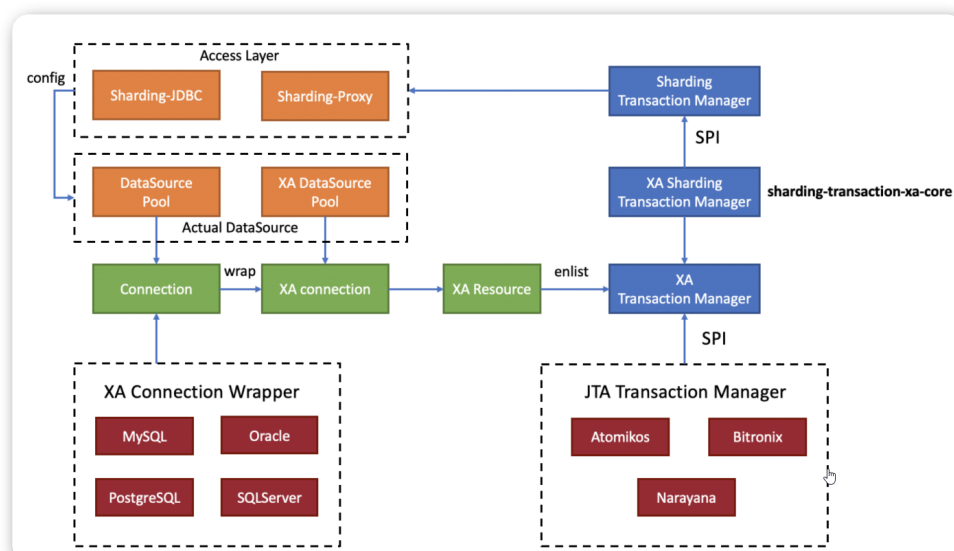
两阶段提交保证操作的原子性和数据的强一致性；

服务宕机重启后，提交/回滚中的事务可自动恢复；

SPI机制整合主流的XA事务管理器，默认Atomikos；

同时支持XA和非XA的连接池；

提供spring-boot和namespace的接入端。



- 开启全局事务  
XAShardingSphereTransactionManager将调用具体的 XA 事务管理器开启 XA 全局事务，以 XID 的形式进行标记。
- 执行真实分片SQL  
XAShardingSphereTransactionManager将数据库连接所对应的 XAResource注册到当前 XA 事务中之后，事务管理器会在此阶段发送XAResource.start命令至数据库。数据库在收到 XAResource.end命令之前的所有 SQL 操作，会被标记为 XA 事务。

```
XAResource1.start          ## Enlist阶段执行
statement.execute("sql1");  ## 模拟执行一个分片
SQL1
statement.execute("sql2");  ## 模拟执行一个分片
SQL2
XAResource1.end            ## 提交阶段执行
```

示例中的sql1和sql2将会被标记为 XA 事务。

- 提交或回滚事务  
XAShardingSphereTransactionManager在接收到接入端的提交命令后，会委托实际的 XA 事务管理进行提交动作，事务管理器将收集到的当前线程中所有注册的 XAResource，并发送 XAResource.end指令，用以标记此 XA 事务边界。接着会依次发送prepare指令，收集所有参与 XAResource 投票。若所有 XAResource 的反馈结果均为正确，则调用commit指令进行最终提交；若有任意 XAResource 的反馈结果不正确，则调用rollback指令进行回滚。在事务管理器发出提交指令后，任何 XAResource 产生的异常都会通过恢复日志进行重试，以保证提交阶段的操作原子性，和数据强一致性。

```
XAResource1.prepare      ## ack: yes
XAResource2.prepare      ## ack: yes
XAResource1.commit
XAResource2.commit

XAResource1.prepare      ## ack: yes
XAResource2.prepare      ## ack: no
XAResource1.rollback
XAResource2.rollback
```

6. 常见的分布式事务解决方案有？

- ☒ A 2PC、3PC
- ☐ B XA
- ☐ C 使用MQ
- ☐ D 以上都对

**答案**

6=>D

### Sharding-JDBC分布式事务实战

ShardingSphere整合了XA、为分布式事务控制提供了极大的便利，我们可以在应用程序编程时，采用以下统一模式进行使用。

- 引入Maven依赖

```
//XA模式
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-transaction-xa-core</artifactId>
</dependency>
```

- JAVA编码方式设置事务类型

```
TransactionTypeHolder.set(TransactionType.XA);
```

- 参数配置

ShardingSphere默认的XA事务管理器为Atomikos，通过在项目的classpath中添加jta.properties来定制化Atomikos配置项。具体的配置规则如下：

```
#指定是否启动磁盘日志，默认为true。在生产环境下一定要保证为true，否则数据的完整性无法保证
com.atomikos.icatch.enable_logging=true
#JTA/XA资源是否应该自动注册
com.atomikos.icatch.automatic_resource_registration=true
#JTA事务的默认超时时间，默认为10000ms
com.atomikos.icatch.default_jta_timeout=10000
#事务的最大超时时间，默认为300000ms。这表示事务超时时间由
#UserTransaction.setTimeout()较大者决定。4.x版本之后，指定为0的话则表示不设置超时时间
com.atomikos.icatch.max_timeout=300000
```

#指定在两阶段提交时，是否使用不同的线程(意味着并行)。  
3.7版本之后默认为false，更早的版本默认为true。如果  
为false，则提交将按照事务中访问资源的顺序进行。  
com.atomikos.icatch.threaded\_2pc=false  
#指定最多可以同时运行的事务数量，默认值为50，负数表示  
没有数量限制。在调用  
#UserTransaction.begin()方法时，可能会抛出一个  
“Max number of active  
transactionsreached”异常信息，表示超出最大事务数限制  
com.atomikos.icatch.max\_actives=50  
#是否支持subtransaction，默认为true  
com.atomikos.icatch.allow\_subtransactions=true  
#指定在可能的情况下，是否应该join子事务  
(subtransactions)，默认值为true。如果设置为  
false，对于有关联的不同subtransactions，不会调用  
XAResource.start(TM\_JOIN)  
com.atomikos.icatch.serial\_jta\_transactions=  
true  
#指定JVM关闭时是否强制(force)关闭事务管理器，默认为  
false  
com.atomikos.icatch.force\_shutdown\_on\_vm\_exit=false  
#在正常关闭(no-force)的情况下，应该等待事务执行完成的  
时间，默认为Long.MAX\_VALUE  
com.atomikos.icatch.default\_max\_wait\_time\_on  
\_shutdown=9223372036854775807  
===== 日志记录配置=====

#事务日志目录，默认为./。  
com.atomikos.icatch.log\_base\_dir=.

```
#事务日志文件前缀，默认为tmlog。事务日志存储在文件中，文件名包含一个数字后缀，日志文件以.log为扩展名，如tmlog1.log。遇到checkpoint时，新的事务日志文件会被创建，数字增加。
com.atomikos.icatch.log_base_name=tmlog
#指定两次checkpoint的时间间隔，默认为500
com.atomikos.icatch.checkpoint_interval=500
=====日志恢复配置=====
#指定在多长时间后可以清空无法恢复的事务日志(orphaned)，默认86400000ms
com.atomikos.icatch.forget_orphaned_log_entries_delay=86400000
#指定两次恢复扫描之间的延迟时间。默认值为与
#com.atomikos.icatch.default_jta_timeout相同
com.atomikos.icatch.recovery_delay=${com.atomikos.icatch.default_jta_timeout}
#提交失败时，再抛出一个异常之前，最多可以重试几次，默认值为5
com.atomikos.icatch.oltp_max_retries=5
#提交失败时，每次重试的时间间隔，默认10000ms
com.atomikos.icatch.oltp_retry_interval=10000
```

## 编排治理剖析

### 配置中心

配置集中化：越来越多的运行时实例，使得散落的配置难于管理，配置不同步导致的问题十分严重。将配置集中于配置中心，可以更加有效进行管理。

配置动态化：配置修改后的分发，是配置中心可以提供的另一个重要能力。它可支持数据源、表与分片及读写分离策略的动态切换。



## 配置中心数据结构

配置中心在定义的命名空间的config下，以YAML格式存储，包括数据源，数据分片，读写分离、Properties配置，可通过修改节点来实现对于配置的动态管理。

```
config
  └─authentication #
Sharding-Proxy权限配置
  └─props # 属性
配置
  └─schema #
Schema配置
  └─ └─sharding_db
# SchemaName配置
  └─ └─ └─datasource
# 数据源配置
  └─ └─ └─rule
# 数据分片规则配置
  └─ └─masterslave_db
# SchemaName配置
  └─ └─ └─datasource
# 数据源配置
  └─ └─ └─rule
# 读写分离规则
```

### config/authentication

```
password: root
username: root
```

### config/sharding/props

```
sql.show: true
```



config/schema/schemeName/datasource

多个数据库连接池的集合，不同数据库连接池属性自适应（例如：DBCP, C3P0, Druid, HikariCP）。

```
ds_0:
dataSourceClassName:
com.zaxxer.hikari.HikariDataSource
  properties:
    url: jdbc:mysql://127.0.0.1:3306/itbaizhan1?
serverTimezone=UTC&useSSL=false
    password: root
    username: root
    maxPoolSize: 50
    minPoolSize: 1
ds_1:
dataSourceClassName:
com.zaxxer.hikari.HikariDataSource
  properties:
    url: jdbc:mysql://127.0.0.1:3306/itbaizhan2?
serverTimezone=UTC&useSSL=false
    password: root
    username: root
    maxPoolSize: 50
    minPoolSize: 1
```

config/schema/sharding\_db/rule

数据分片配置，包括数据分片配置。

```
tables:
  t_order:
```

```
    actualDataNodes: ds_-$->{0..1}.t_order_-$->
{0..1}
    databaseStrategy:
      inline:
        shardingColumn: user_id
        algorithmExpression: ds_-$->{user_id %
2}
    keyGenerator:
      column: order_id
    logicTable: t_order
    tableStrategy:
      inline:
        shardingColumn: order_id
        algorithmExpression: t_order_-$->
{order_id % 2}
    t_order_item:
      actualDataNodes: ds_-$->
{0..1}.t_order_item_-$->{0..1}
      databaseStrategy:
        inline:
          shardingColumn: user_id
          algorithmExpression: ds_-$->{user_id %
2}
      keyGenerator:
        column: order_item_id
      logicTable: t_order_item
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order_item_-$->
{order_id % 2}
    bindingTables:
```

```
- t_order,t_order_item  
broadcastTables:  
- t_config  
defaultDataSourceName: ds_0  
masterSlaveRules: {}
```

config/schema/masterslave/rule

读写分离独立使用时使用该配置。

```
name: ds_ms  
masterDataSourceName: ds_master  
slaveDataSourceNames:  
- ds_slave0  
- ds_slave1  
loadBalanceAlgorithmType: ROUND_ROBIN
```

动态生效

在注册中心上修改、删除、新增相关配置，会动态推送到生产环境并立即生效

## 注册中心

相对于配置中心管理配置数据，注册中心存放运行时的动态/临时状态数据，比如可用的proxy的实例，需要禁用或熔断的datasource实例。

通过注册中心，可以提供熔断数据库访问程序对数据库的访问和禁用从库的访问的编排治理能力。治理仍然有大量未完成的功能（比如流控等）。

注册中心数据结构

注册中心在定义的命名空间的state下，创建数据库访问对象运行节点，用于区分不同数据库访问实例。包括instances和datasources节点。

```
instances
├─your_instance_ip_a@-@your_instance_pid_x
├─your_instance_ip_b@-@your_instance_pid_y
├─....
datasources
├─ds0
├─ds1
├─....
```

state/instances

数据库访问对象运行实例信息，子节点是当前运行实例的标识。运行实例标识由运行服务器的IP地址和PID构成。运行实例标识均为临时节点，当实例上线时注册，下线时自动清理。注册中心监控这些节点的变化来治理运行中实例对数据库的访问等。

state/datasources

可以控制读写分离，可动态添加删除以及禁用。

熔断实例

可在IP地址@-@PID节点写入DISABLED（忽略大小写）表示禁用该实例，删除DISABLED表示启用。

禁用从库

在读写分离场景下，可在数据源名称子节点中写入DISABLED表示禁用从库数据源，删除DISABLED或节点表示启用。

支持的配置中心和注册中心

ShardingSphere在数据库治理模块使用SPI方式载入数据到配置中心/注册中心，进行实例熔断和数据库禁用。目前，ShardingSphere内部支持Zookeeper和Etcd这种常用的配置中心/注册中心。此外，您可以使用其他第三方配置中心/注册中心，例如

Apollo、Nacos等，并通过SPI的方式注入到ShardingSphere，从而使用该配置中心/注册中心，实现数据库治理功能。