分布式关系型数据库 DRDS

用户指南

用户指南

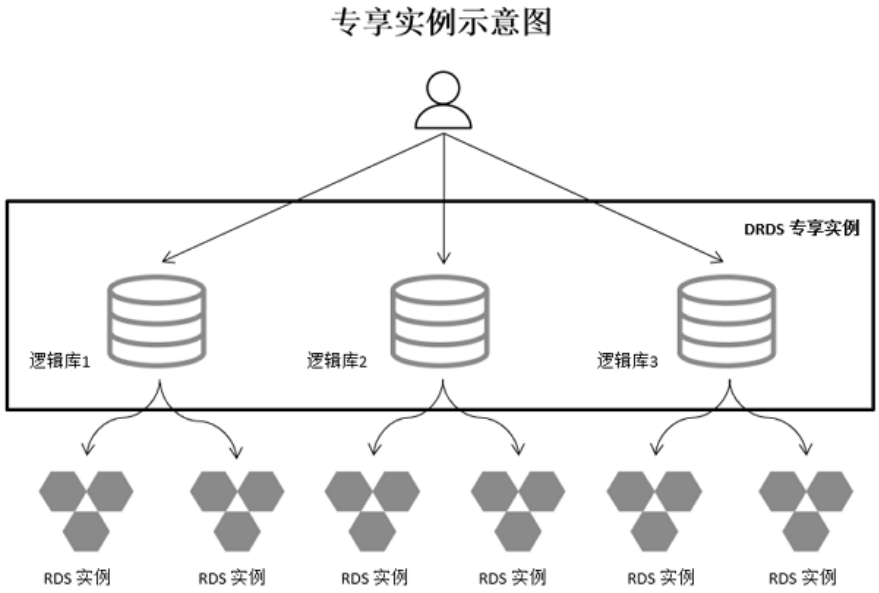
实例管理

DRDS 实例介绍

DRDS 实例在物理上是由多个 DRDS Server 和底层存储组成的分布式集群。 DRDS 的库是逻辑概念，只包含元信息，由底层存储的物理库存储具体数据。目前售卖的DRDS 实例为**专享实例**。

基本原理

- **专享实例** 用户独享 DRDS 的物理资源。



说明： 已有的共享实例上的 DRDS 的逻辑库元信息可以随时自助迁移到指定专享实例，无需重新导入数据即可快速完成从共享实例到专享实例的迁移。

创建实例

创建 DRDS 实例的步骤如下：

登录阿里云官网，单击右上角控制台按钮。

在左侧菜单栏选择**分布式关系型数据库**，进入 DRDS 控制台。

在 DRDS 控制台页面，单击右上角**创建实例**。

在**创建实例**页面，选择付费方式及规格等选项，点击**立即购买**，完成购买流程。

购买完成后，可以在 DRDS 控制台单击左侧**实例列表**，查看新购的实例。

选项说明：

- 付费方式：**包年包月**是预先付费，以月为单位计费；**按量付费**是后付费，按使用量计费。
- 地域、可用区：通过选择**地域**和**可用区**，可以配置实例所在的物理位置。
- 实例类型：实例类型标识了是共享或者专享。
- 网络类型：选择网络类型，是**经典网络**还是**专有网络**。
- 规格：**实例规格**用于选择实例物理规格 CPU 核数和内存大小。

注意：

- 包年包月和按量付费稍有不同，包年包月需要选择购买时长，并完成支付流程。
- 选项网络类型时，购买过相应地域和可用区内的专有网络和虚拟交换机，才能配置专有网络。
- 由于 DRDS 不包含 RDS，在选择网络类型时务必与要使用地 RDS 保持一致。选择可用区

时，推荐与使用地 RDS 匹配，否则有毫秒级的网络延迟。

实例变配

DRDS 实例是由 DRDS Server 和底层存储两部分组成的分布式集群。DRDS Server 承担了 SQL 路由、数据合并、聚合等功能，底层存储主要提供数据存储功能。通过实例变配，您可以变更 DRDS 集群的节点个数，在业务繁忙时进行升配，承载更多的业务流量，在业务空闲时进行降配，避免资源浪费。

实例升降配仅适用于专享实例：

- 按量付费实例：支持动态升降配。
- 包年包月实例：支持升配，不支持降配。

变更配置操作为：

1. 进入实例列表页面，找到要进行变更的实例，在右侧单击**变更配置**，进入实例变配页面。
2. 在变配页面中，选择变配规格，单击**确定变更**。

耐心等待几分钟，之后可以在实例列表中查看变更后的实例配置。

注意：

- 实例变配需要开通 DRDS 的 RAM 角色用户调用 RDS 的接口。
- 降配会导致应用与 DRDS 连接中断，在短时间内产生闪断。如果应用具备重连能力可以自动恢复。
- 使用长连接时，升配后新的节点不能有效的接收流量，尽量重启应用。

升级实例版本

您可以在控制台自主升级 DRDS 实例，及时将 DRDS 更新到最新版本，快速体验 DRDS 的新特性。DRDS 版本请参考 DRDS 版本说明。

升级注意事项

升级前请在推荐版本实例上进行全面验证，避免兼容性问题。验证方法如下：

先升级测试用的 DRDS 实例。

然后在自己的 ECS 上部署一个项目的代码，将数据库连接到已经升级到最新版本的测试 DRDS 实例，进行回归测试。

如果没有严重的兼容性异常，则可以升级生产用的 DRDS 实例。

升级过程中请不要进行其它操作，如建库、平滑扩容等。

升级过程会有闪断和少量报错，因此请在业务低峰期执行。

升级操作

针对某个实例有新版本推荐的时候，实例右侧操作选项中会出现升级按钮。

在**实例列表**页面，找到要进行升级的实例，在右侧单击**升级**按钮。请仔细阅读控制台升级说明，确认是否需要升级。

在升级确认页面，单击**确定**按钮，实例开始升级。等待约1分钟，将会显示升级成功。

如果出现实例升级失败，请提交工单由技术人员协助排查。

回滚操作

升级新版后如果发现业务受到影响，或者因为其他原因需要回滚版本，您可以在升级后24小时内回滚。

如果已经超过回滚时效，但仍需回滚，请提交工单。由于工单反馈存在时间差，因此为了保证您的业务稳定，请务必在升级前做好验证工作，尽量避免回滚。

注意：回滚实例时不能变更实例配置。

外网访问

DRDS 实例会默认分配一个内网地址，应用和 MySQL 客户端可以在阿里云 ECS 上连接 DRDS 实例的数据库。如果需要从非阿里云网络访问 DRDS，则需要为 DRDS 实例申请外网地址。当外网地址不需要时，请及时释放外网地址。

申请外网地址

前提条件

- 共享实例无法申请外网地址；
- 申请外网地址前请确保实例下所有 DRDS 数据库已经设置了 IP 白名单，限制除应用外的 IP 访问。

操作步骤

为 DRDS 实例申请外网地址的步骤如下。

登录 DRDS 控制台，进入对应的 DRDS 实例详情页面。

在**连接信息**栏，点击**申请外网地址**按钮，如下图所示。

在弹出的确认框中点击**确定**按钮，生成外网地址。

释放外网地址

释放 DRDS 实例外网地址的步骤如下。

登录 DRDS 控制台，进入对应 DRDS 实例详情页面。

在连接信息处，点击**释放外网地址**按钮，如下图所示。

在弹出的确认框中点击**确定**按钮，释放外网地址。

创建克隆实例

DRDS 提供了实例克隆的功能，方便您快速复制线上环境。实例克隆功能可以用于预发环境创建、压测及问题排查等。如果当前库中数据被误删，也可以利用实例克隆功能快速进行数据恢复。

创建克隆实例时，可以根据指定的 DRDS 实例和数据库，选择相应的时间点进行快速克隆。克隆的内容包括底层 RDS 实例（如备份设置、数据、网络类型），以及 DRDS 中的逻辑库（如分库数量等）。

注意：目前实例克隆功能暂未全面开放，如有需求请**提交工单**申请开通。申请内容请注明：开通实例克隆功能。

克隆详情说明

克隆内容：

- **DRDS**: 实例地域、可用区、数据库类型、版本、网络类型、连接池信息、逻辑库和原实例保持一致；
- **RDS**: 实例地域、可用区、数据库类型、版本、网络类型、白名单、数据、阈值报警设置、备份设置、SQL 审计设置和原实例保持一致。

实例规格：

- 克隆的新 DRDS 实例统一为最小规格，即 4C4G；
- 克隆的新 RDS 实例规格和原 RDS 实例保持一致；
- 新的 DRDS 实例和 RDS 实例均为按量计费。

RDS 主备实例克隆规则：

- 如果 RDS 主实例下挂载了只读实例和灾备实例，克隆时只克隆该主实例，不克隆其下的只读实例和灾备实例。

克隆影响：

- 克隆操作对线上实例无影响；
- 克隆完成后，新的 DRDS 和 RDS 实例即和普通实例没有区别。

克隆失败申请退款：

- 如果创建克隆实例失败，请**提交工单**申请退款，申请内容注明：
    - 实例克隆失败，申请退款；

- 实例 ID：xxxx。

前提条件

进行实例克隆前，请确保 DRDS 和 RDS 实例满足以下前提条件，否则克隆操作将会失败。

DRDS：

- DRDS 实例必须为 5.1.28-1320920 及以上版本，如何查看实例版本请参考 DRDS 版本说明；
- DRDS 实例必须为专享实例。

RDS：

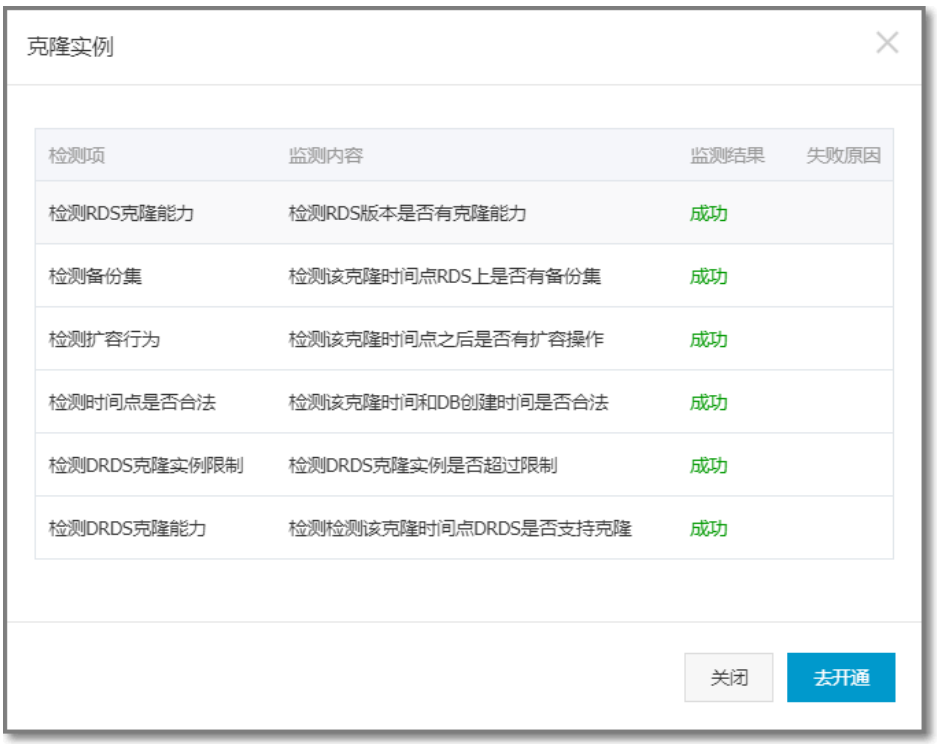
- RDS 状态为运行中且没有被锁定；
- 当前没有迁移任务；
- 已开启数据备份和日志备份；
- RDS 没有创建高权限账号；
- RDS 实例不是 5.7 版本。

操作步骤

在 DRDS 控制台左侧菜单栏选择**实例管理**，单击要操作的实例名称，进入实例基本信息页。

在**实例基本信息**页，单击右上角的**克隆实例**按钮。

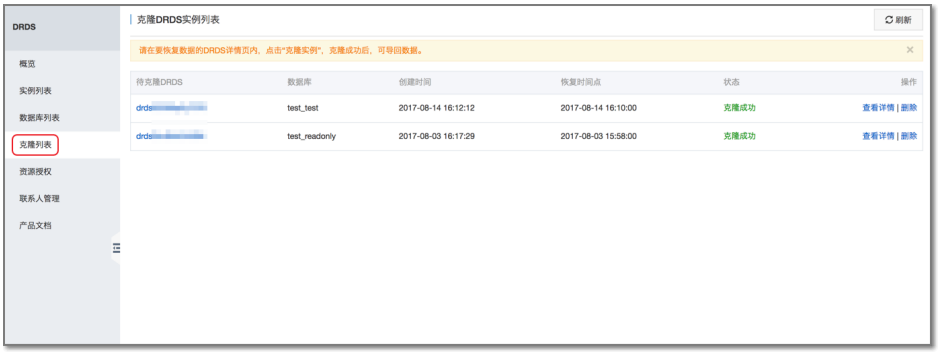
在**克隆实例**对话框，输入克隆任务名称，选择需要克隆的库，以及需要恢复的时间点，单击**预检查**。DRDS 会对实例的克隆能力进行检测，确保实例满足克隆条件。



预检测通过后，单击**去开通**。

在订单确认页面，确认需要开通的实例信息，单击**去开通**开始实例克隆。页面提示开通成功后，可以回到克隆列表查看克隆任务的状态。

克隆任务运行时间根据数据量大小会有所不同，请耐心等待。



释放实例

释放实例前，请确保已经删除了该实例下的所有数据库。

在实例列表中找到要释放的实例，单击右侧的**释放**选项。

在释放 DRDS 实例对话框单击**确定**。

注意：虽然实例释放操作本身不会主动删除 RDS 上的数据，但是实例释放后不可恢复，请谨慎操作。

实例监控

本文介绍 DRDS 的性能监控功能，如何分析性能指标并根据指标排查 DRDS 性能问题。

查看监控信息

在 DRDS 控制台的实例列表页，单击需要操作的实例名称。

在实例**基本信息**页，在左侧菜单栏选择**监控信息**，进入监控详情页。

实例监控主要分为**资源监控**和 DRDS **引擎监控**。引擎层面的监控指标又可以分成 DRDS 实例级别和 DRDS 库级别两个维度的监控，当某些引擎类的监控指标出现异常的时候，可以直接查看各个数据库的监控指标，从而定位到有性能问题的数据库。下文针对这两类监控下的各指标进行详细说明。

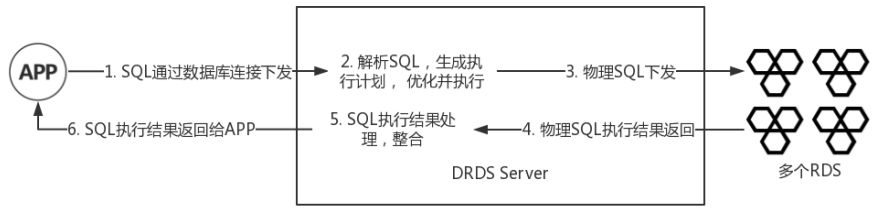
监控指标技术说明

监控项	监控项分类	含义	数据采集周期	数据保留时长	说明
CPU 利用率	资源监控	DRDS 服务节点的 CPU 利用率的平均值	5分钟	3天	
网络输入流量	资源监控	DRDS 服务节点的网络输入流量的总和	5分钟	3天	RDS 返回数据到 DRDS，会产生网络输入流量
网络输出流量	资源监控	DRDS 服务节点的网络输出流量的总和	5分钟	3天	DRDS 发送物理 SQL 到 RDS，DRDS 返回数据到应用，均会产生网络输出流量

逻辑 QPS	引擎监控	DRDS 服务节点每秒处理的 SQL 语句数目的总和	5分钟	7天	
物理 QPS	引擎监控	DRDS 服务节点每秒发送到 RDS 的 SQL 操作数总和	5分钟	7天	一条逻辑 SQL 可能会拆分成多条物理 SQL
逻辑 RT	引擎监控	DRDS 对于每条 SQL 的平均响应时间	5分钟	7天	如果逻辑 SQL 会变成物理 SQL 下发，那么此条 SQL 的逻辑 RT 会包含物理 SQL 的 RT
物理 RT	引擎监控	DRDS 发送到 RDS 的 SQL 的平均响应时间	5分钟	7天	
连接数	引擎监控	应用到 DRDS 的连接总数	5分钟	7天	不包括 DRDS 到 RDS 的连接
活跃线程数	引擎监控	DRDS 用来执行 SQL 的线程数	5分钟	7天	

监控指标原理

在分析监控指标之前，需要对 SQL 语句在 DRDS 上的执行流程进行了解。



在整个 SQL 的执行链路中，第2到第4步的执行状况都会在 DRDS 的各个监控指标上有所体现。

第2步：SQL 解析、优化、执行，主要消耗的是 CPU 资源。越是复杂的 SQL（结构复杂或者超长），消耗的 CPU 资源就越大。通过 TRACE 指令跟踪 SQL 的执行过程，可以看到一条 SQL 在优化阶段的耗时。这部分的耗时越高，表示 CPU 资源消耗的就越高。

第3步：物理 SQL 下发和执行，主要消耗的是 IO 资源，可以通过逻辑、物理的 QPS 和 RT 等指标分析出这一部分的执行状况。例如，如果物理 QPS 很低同时 物理 RT 很高，表示当前 RDS 处理 SQL 很慢，需要关注 RDS 的性能。

第5步：SQL 执行结果处理、整合，这部分操作主要是用来对物理 SQL 的执行结果进行转换。多数情况下，此类转换只会进行一些 SQL 元信息的转换，资源消耗很小。但是当出现 heap sort 等执行步骤的时候，则会消耗非常高的 CPU 资源。关于如何确定 SQL 在此阶段的消耗，可以参考排查 DRDS 慢 SQL文档中关于 TRACE 指令的说明。

预防性能问题

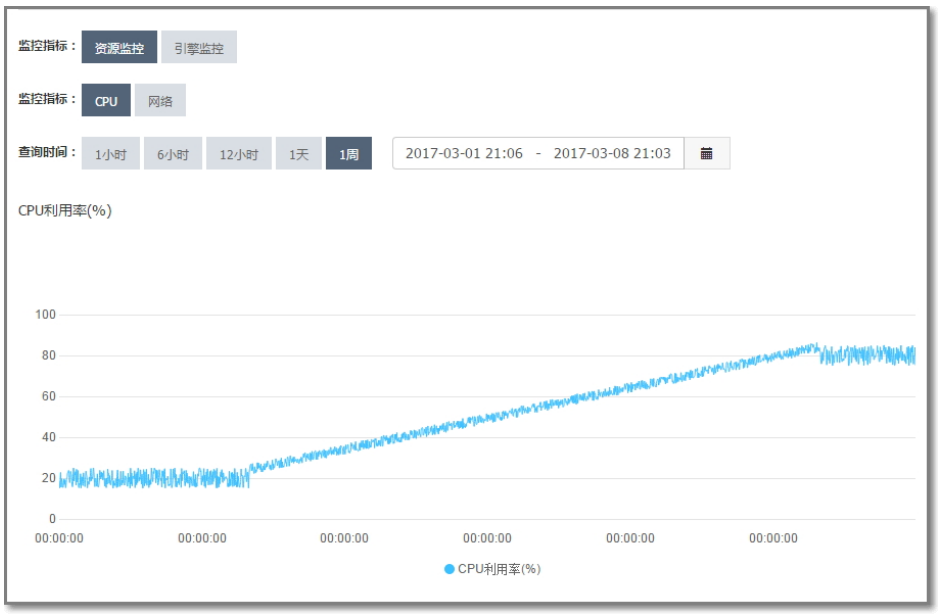
对于常见的数据库性能问题，结合性能监控指标，可以得到比较有效的处理方法。

示例一：性能监控指标随着业务流量的变化而变化

性能指标往往会随着当前系统业务量的波动而波动。以下是常见的两种情况：

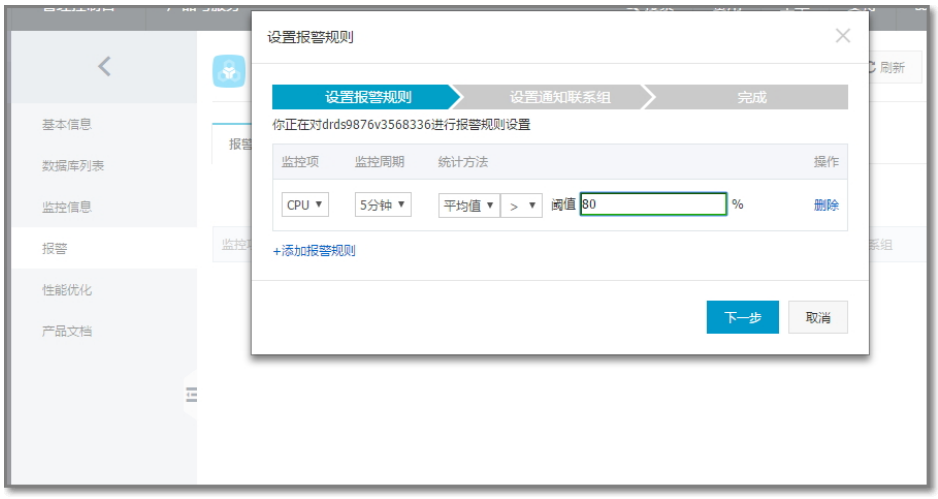
某应用在每天早上9点会有抢购活动，所以这个时间点整个系统的业务流量会有一个明显的增长。从监控数据上来看，DRDS 的 CPU 利用率从9点开始从20%增长到80%左右，整个高峰持续10分钟左右。

某应用业务一直在增长，系统的业务流量也随着一直增长，直到稳定到一个水平线。DRDS 的 CPU 监控数据也基本上反应了这一变化。



在 DRDS 的压力随着业务变化而变化的时候，应该密切关注监控指标的变化。如果超过预设的阈值，则**应该通过进行 DRDS 升配缓解性能压力。**

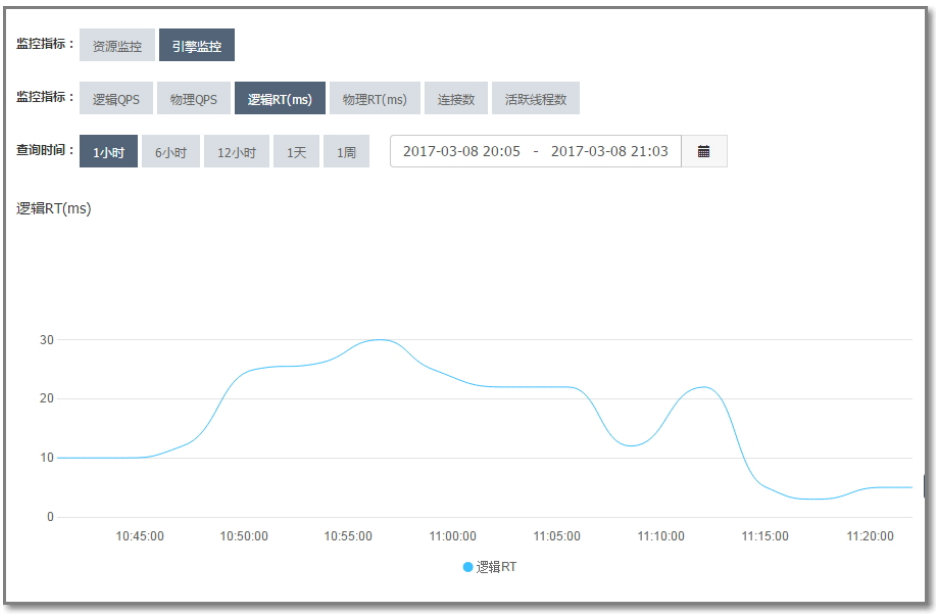
在 DRDS 控制台可以针对实例设置报警规则，当 CPU 的平均值超过预设的阈值的时候，系统会方发送短信到对应的联系人。CPU 的阈值可以根据实际情况设定，推荐设置成80%。



示例二：观察逻辑 RT 和物理 RT 的差值

逻辑 RT 指的是 DRDS 从收到逻辑 SQL 到返回数据给应用的响应时间，物理 RT 指的是 DRDS 从发出物理 SQL 到 RDS 至收到 RDS 返回数据的时间。

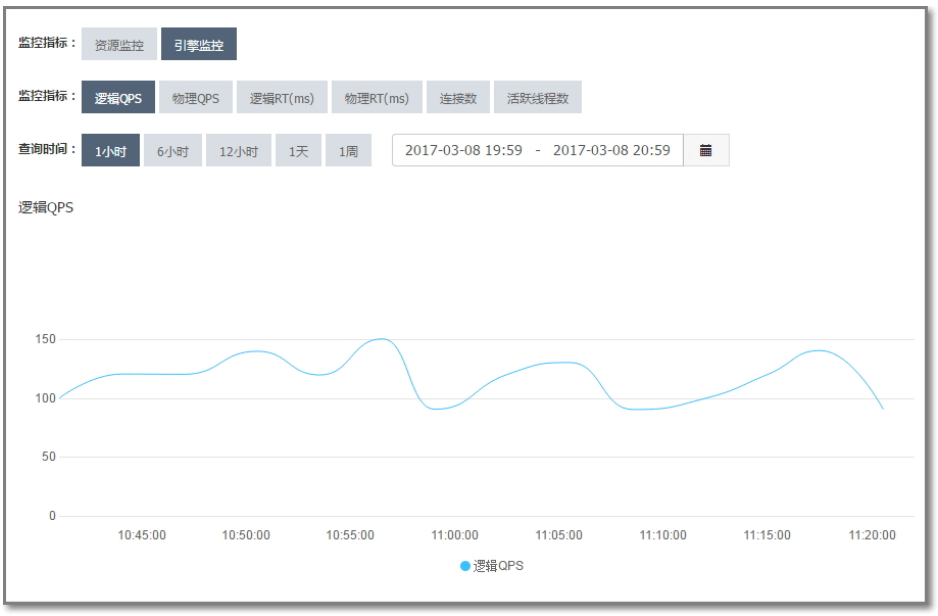
如果一条逻辑 SQL 被拆分成了一条或者多条物理 SQL 那么逻辑 RT 会大于等于物理 RT。理想情况下，DRDS 只会对于 RDS 返回过来的数据进行少量的操作，所以一般情况下逻辑 RT 只会略高于物理 RT 一点。但是某些情况下，物理的 SQL 执行很快，而逻辑 SQL 处理时间很久，则逻辑 RT 和物理 RT 会分别出现如下的走向：

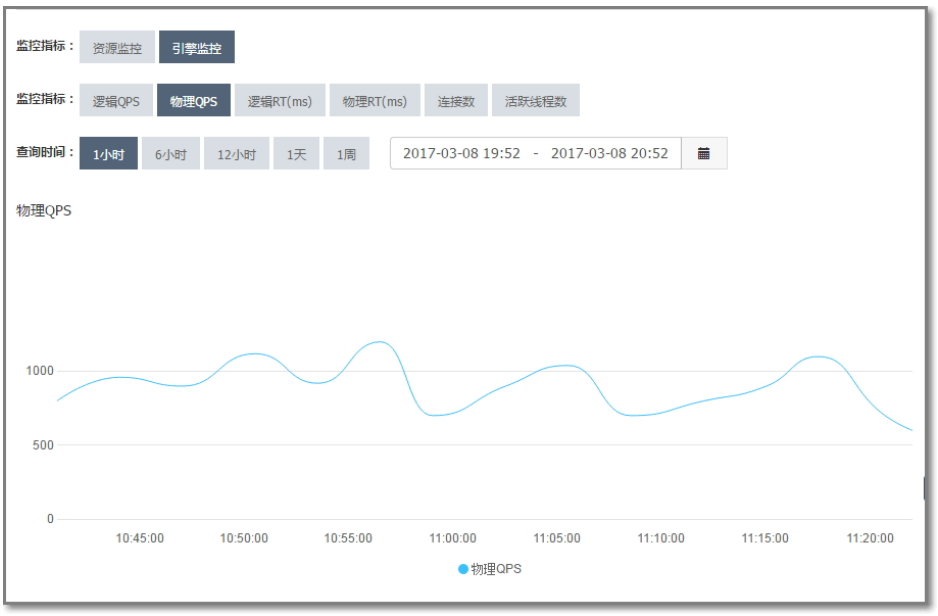


从上面两个监控图可以看到，逻辑 RT 和物理 RT 的变化趋势大致一样，逻辑 RT 在10毫秒到20毫秒范围内波动，物理 RT 在2毫秒到5毫秒范围内波动。这说明 DRDS 层面已经有较大的压力。这种情况可以通过升级 DRDS 配置来解决性能问题。相反，如果逻辑 RT 和 物理 RT 都很高，那么可以通过升级 RDS 的配置或者在 RDS 层面优化 SQL 来解决性能问题。

示例三：观察逻辑 QPS 和物理 QPS 的差值

从监控数据上来看，逻辑 QPS 和物理 QPS 的趋势相同，但是两者有差值比较大，且成一定的比例。





根据监控指标显示，逻辑 QPS 在80至150这个范围内波动，物理 QPS 在700至1200这个范围内波动。

造成这种现象的原因是 DRDS 会根据逻辑 SQL 生成物理 SQL，逻辑 SQL 和物理 SQL 的数量比例并不是1：1的关系。例如有一个 DRDS 逻辑表，建表语句如下：

```
CREATE TABLE drds_user
(id int,
name varchar(30))
dbpartition by hash(id);
```

当查询条件带上分库键id，DRDS 会将这条逻辑 SQL 直接下推到 RDS 去执行，从执行计划上可知，物理 SQL 的数目是1条：

```
mysql> explain select name from drds_user where id = 1;
```

```
+--------------------------+--------------------------------------------------------------------------+--------+
| GROUP_NAME | SQL | PARAMS |
+--------------------------+--------------------------------------------------------------------------+--------+
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`id` = 1) | {} |
+--------------------------+--------------------------------------------------------------------------+--------+
```

当查询没有带上分库键，DRDS 会将逻辑 SQL 拆分成多条条物理 SQL。从下面的执行计划可知，物理 SQL 的数目是8条。

```
mysql> explain select name from drds_user where name = 'LiLei';
+----------- --------------+----------------------------------------------------------------------------------+--------+
| GROUP_NAME | SQL | PARAMS |
+--------------------------+----------------------------------------------------------------------------------+--------+
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
| SANGUAN_BSQT_0001_RDS | select `drds_user`.`name` from `drds_user` where (`drds_user`.`name` = 'LiLei') | {} |
+--------------------------+----------------------------------------------------------------------------------+--------+
8 rows in set (0.06 sec)
```

逻辑/物理 QPS 指的是在单位时间内处理的逻辑/物理 SQL 语句的总数。当整个系统的多数 SQL 都带上了拆分条件，逻辑 QPS 和 物理 QPS 的比值应该是接近于 1：1。逻辑 QPS 和 物理 QPS 的差值过于大则意味着当前应用有很多 SQL 语句没有带拆分条件，应该排查应用的 SQL 语句，提高性能。

DRDS 账号和权限系统

DRDS 账号和权限系统的用法跟 MySQL 一致，支持 GRANT、REVOKE、SHOW GRANTS、CREATE USER、DROP USER、SET PASSWORD 等语句，目前支持库级和表级权限的授予，全局级别和列级别权限暂时不支持。

MySQL 账号和权限系统的详细信息请参考 MySQL 官方文档。

注意：DRDS 里通过 CREATE USER 创建出来的账号只存在于 DRDS，跟 RDS 没有任何关系，也不会同步到后端的 RDS 中去。

账号

用户名和主机名的组合 username@'host' 确定一个账号。用户名一样但是主机名不一样则代表不同的账号。例如 lily@30.9.73.96 和 lily@30.9.73.100 是两个完全不同的账号，这两个账号的密码和权限都可能不一样。

在 DRDS 控制台创建完数据库之后，系统会自动在该数据库下创建两个系统账号：管理员账号和只读账号。这

两个账号是系统内置的，不能删除，不能修改其权限。

- 管理员账号的名字跟数据库名一致，比如数据库名是 easydb，管理员账号的名字就叫 easydb。
- 只读账号的名字是数据库名加上 _RO 后缀，比如数据库名是 easydb，只读账号的名字就叫 easydb_RO。

假设有两个数据库 dreamdb、andordb，根据上面的规则可知，库 dreamdb 下面包含管理员账号 dreamdb，只读账号 dreamdb_RO；库 andordb 下面包含管理员账号 andordb，只读账号 andordb_RO。

账号规则

- 管理员账号具有所有权限；
- 只有管理员账号具有创建账号和授权功能；其它账号只能由管理员账号创建，其权限只能由管理员账号授予；
- 管理员账号是跟数据库绑定的，没有其它数据库的权限，连接的时候只能连接自己对应的那个数据库，不能授予其它数据库的权限给某个账号。例如 easydb 这个管理员账号只能连接 easydb 数据库，只能授予 easydb 数据库权限或者 easydb 数据库中表的权限给某个账号；
- 只读账号只具有 SELECT 权限。

用户名规则

- 大小写不敏感；
- 长度必须大于等于4个字符，小于等于20个字符；
- 必须以字母开头；
- 字符可以包括大写字母、小写字母、数字。

密码规则

- 长度必须大于等于6个字符，小于等于20个字符；
- 字符可以包括大写字母、小写字母、数字、特殊字符（@#$%^&+=）。

HOST 匹配规则

- HOST 必须是纯 IP 地址，可以包含 _ 和 % 通配符（ _ 代表一个字符，% 代表0个或多个字符）。含有通配符的 HOST 需要加上单引号，例如 lily@’30.9.%.%’，david@’%’；
- 假设系统中有两个用户都符合当前欲登录的用户，则以最长前缀匹配（不包含通配符的最长 IP 段）的那个用户为准。例如系统有两个用户 david@'30.9.12_.234' 和 david@'30.9.1%.234'，在主机 30.9.127.234 上面登录 david，则使用的是 david@'30.9.12_.234'这个用户。
- 开启 VPC 时，主机的 IP 地址会发生变化。**为避免账号和权限系统中的配置无效，请将 HOST 配置为‘%’ 来匹配任意 IP。**

权限

权限级别支持情况

- 全局层级（暂不支持）
- 数据库层级（支持）
- 表层级（支持）
- 列层级（暂不支持）
- 子程序层级（暂不支持）

权限项

目前支持和表相关联的8个基本权限项：CREATE、DROP、ALTER、INDEX、INSERT、DELETE、UPDATE、SELECT。

- TRUNCATE 操作需要有表上的 DROP 权限；
- REPLACE 操作需要有表上的 INSERT 和 DELETE 权限；
- CREATE INDEX 和 DROP INDEX 操作需要有表上的 INDEX 权限；
- CREATE SEQUENCE 需要有数据库级的创建表（CREATE）权限；
- DROP SEQUENCE 需要有数据库级的删除表（DROP）权限；
- ALTER SEQUENCE 需要有数据库级的更改表（ALTER）权限；
- INSERT ON DUPLICATE UPDATE 语句需要有表上的 INSERT 和 UPDATE 权限。

权限规则

- 权限是绑定到账号（username@’ host’ ），不是绑定到用户名（username）；
- 授权的时候会判断表是否存在，不存在则报错；
- 数据库权限级别从高到低依次是：全局级别权限（暂不支持）、数据库级别权限、表级别权限、列级别权限。高级别权限的授予会覆盖低级别权限，移除高级别权限的同时也会移除低级别权限；
- 不支持 USAGE 授权。

1. 创建账号（CREATE USER）语句

语法规则：

```
CREATE USER user_specification [, user_specification] ...
user_specification: user [ auth_option ]
auth_option: IDENTIFIED BY 'auth#string'
```

例如：

创建一个名为 lily，只能从 30.9.73.96 登录的账号，密码为123456。

```
CREATE USER lily@30.9.73.96 IDENTIFIED BY '123456';
```

创建一个名为 david，可以从任意主机登录的账号，密码为空。

```
CREATE USER david@'%';
```

2. 删除账号（DROP USER）语句

语法规则：

```
DROP USER user [, user] ...
```

例如：

移除账号 lily@30.9.73.96。

```
DROP USER lily@30.9.73.96;
```

3. 修改账号密码（SET PASSWORD）语句

语法规则：

```
    SET PASSWORD FOR user = password_option

password_option: {
PASSWORD('auth_string')
}
```

例如：

修改账号 lily@30.9.73.96 的密码为123456。

```
SET PASSWORD FOR lily@30.9.73.96 = PASSWORD('123456')
```

4. 授权权限（GRANT）语句

语法规则：

```
GRANT
priv_type[, priv_type] ...
ON priv_level
TO user_specification [, user_specification] ...
[WITH GRANT OPTION]

priv_level: {
| db_name.*
| db_name.tbl_name
| tbl_name
}

user_specification:
user [ auth_option ]
```

```
auth_option: {
IDENTIFIED BY 'auth#string'
}
```

注意：GRANT 语句里面的账号如果不存在，同时又没有提供 IDENTIFIED BY 信息，则报账号不存在异常；如果提供了 IDENTIFIED BY 信息，则会创建该账号同时授权。

例如：

在数据库 easydb 下面，创建一个用户名为 david，可以在任意主机登录，具有 easydb 数据库所有权限的账号。

方法1：先创建账号，再授权。

```
CREATE USER david@'%' IDENTIFIED BY 'your#password';
GRANT ALL PRIVILEGES ON easydb.* to david@'%';
```

方法2：一条语句完成创建账号和授权两个操作。

```
GRANT ALL PRIVILEGES ON easydb.* to david@'%' IDENTIFIED BY 'your#password';
```

在数据库 easydb 下面，创建一个用户名为 hanson，可以在任意主机登录，具有 easydb.employees 表所有权限的账号。

```
GRANT ALL PRIVILEGES ON easydb.employees to hanson@'%' IDENTIFIED BY 'your#password';
```

在数据库 easydb 下面，创建一个用户名为 hanson，只能在 192.168.3.10 登录，具有 easydb.emp 表的 INSERT 和 SELECT 权限的账号。

```
GRANT INSERT,SELECT ON easydb.emp to hanson@'192.168.3.10' IDENTIFIED BY 'your#password';
```

在数据库 easydb 下面创建一个只读账号 actro，可以在任意主机登录。

```
GRANT SELECT ON easydb.* to actro@'%' IDENTIFIED BY 'your#password';
```

5. 回收权限（REVOKE）语句

语法规则：

删除账号的权限

删除账号在某个权限级别下的权限项，具体权限级别由 priv_level 指定。

```
REVOKE
priv_type
```

```
[, priv_type] ...
ON priv_level
FROM user [, user] ...
```

删除账号的所有权限

删除账号在系统内（数据库级别和表级别的）的所有权限项。

```
REVOKE ALL PRIVILEGES, GRANT OPTION
FROM user [, user] ...
```

例如：

删除 hanson@’%’ 在 easydb.emp 表的 CREATE、DROP、INDEX 权限。

```
REVOKE CREATE,DROP,INDEX ON easydb.emp FROM hanson@'%';
```

删掉账号 lily@30.9.73.96 的所有权限。

```
REVOKE ALL PRIVILEGES,GRANT OPTION FROM lily@30.9.73.96;
```

注意：为了兼容 MySQL，需同时写上 GRANT OPTION。

6. 查看授权（SHOW GRANTS）语句

语法规则：

```
SHOW GRANTS[ FOR user@host];
```

查询所有的授权

```
SHOW GRANTS
```

查询针对某个账号的授权情况

```
SHOW GRANTS FOR user@host;
```

数据库管理

数据库管理概述

概念上，DRDS 数据库类似 MySQL 和 Oracle 中的数据库。在使用上，DRDS 提供一个完整的数据库使用接口；在物理存储上，DRDS 底层存储由一个或者多个 RDS 实例组成，一个 DRDS 数据库由底层 RDS 提供的数据库组成。

针对单个 DRDS 实例，数据库管理包括创建数据库、查看数据库详情、配置读写分离、数据表管理、分库管理、白名单设置、数据导入、只读实例管理、平滑扩容、查看监控信息等功能。

创建数据库

虽然在数据库使用上，DRDS 提供和 MySQL 相同的使用体验，但是 DRDS 创建数据库的流程有所不同，需要在控制台上完成。具体操作请参考**创建数据库**文档。

查看数据库信息

DRDS 控制台提供查看数据库具体信息、删除数据库、重置密码、只读用户管理等功能。

具体操作如下：

1. 在 DRDS 控制台左侧菜单栏单击**数据库列表**。

   在**数据库列表**页选择需要查看的数据库，单击名称进入**基本信息**页。

在此页面可以查看数据库名称、状态、工作模式等信息。

其中，命令行链接地址是指登录到数据库的链接地址。由于 DRDS 完全兼容 MySQL 协议，通过 MySQL 客户端可以操作 DRDS 数据库。将**命令行链接地址**复制粘贴到操作系统终端中，输入密码即可登录使用 DRDS 数据库。

注意：

- 一些旧版 MySQL 客户端对用户名长度有限制，不能大于16个字符。 DRDS 使用的库名和用户名相同，建库时库名超过16个字符，则会报错。
- 由于使用 MySQL 客户端，在使用 HINT 时，在命令行上务必添加 -c 参数。 DRDS 的 HINT 是通过注释实现(类似/*...*/)。如果没有加 -c 参数，会丢失注释，从而导致 DRDS 的 HINT 丢失。

只读用户管理

每个 DRDS 数据库可以创建一个只读用户，在 DRDS 数据库中该只读用户只能执行 SELECT、SET、SHOW 等读操作不能执行写操作，以便进行权限限制。

创建只读用户的步骤如下：

在数据库**基本信息**页，单击基本信息一栏右上角的**只读用户管理**按钮。

输入用户密码，完成创建只读用户。只读用户的用户名是固定的，不可更改。

创建成功后，再次单击**只读用户管理**按钮，可以对只读用户进行重置密码和删除只读用户操作。

说明：版本号高于 5.1.26-1100253 的 DRDS 支持更完善的账户体系功能，具体请参考 DRDS 账号和权限系统文档。

设置读写分离

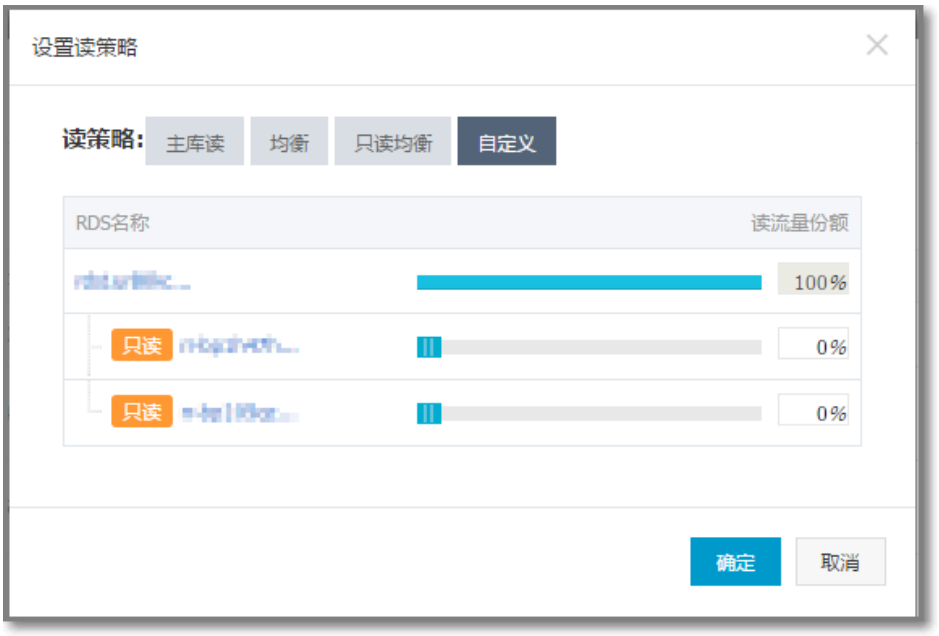
本文介绍如何设置读写分离。读写分离的原理介绍请参考文档 DRDS 读写分离。

操作步骤如下：

进入数据库基本信息页面，单击左侧菜单栏的**读写分离**。

注意：如果 RDS 没有建立只读实例或者只读实例数量不足，需要先添加只读实例。在 RDS 列表右侧单击**添加只读实例**，系统会自动跳转到 RDS 控制台，请按控制台引导完成只读实例添加。完成操作后，请在 RDS 控制台双击浏览器后退箭头，回到 DRDS 控制台的读写分离页面。

在对应的实例右侧单击**设置读策略**，在对话框选择读策略并配置比例，单击**确定**。



注意：

读写比例在容量管理页面是以 RDS 实例为单位设置的。如果一个 DRDS 数据库含有多个 RDS 实例，则需要针对每个 RDS 实例设置读写比例。

RDS 只读实例过期或者需要释放时，需要在 DRDS 控制台中把读权重置为0，否则流量会继续走到原有只读实例，导致执行失败。

读写分离以 DRDS 数据库为基本单位，如果同一个 RDS 只读实例在多个 DRDS 数据库上使用，需要在每个 DRDS 数据库上都将其权重设置为0。

设置完成后，可以连接 DRDS，执行 SHOW NODE 指令查看实际读流量分布。

通过 DRDS HINT 可以指定在主实例或只读实例执行 SQL，具体可以参考 DRDS 自定义 HINT 文档。

查看监控信息

DRDS 提供了数据库的 QPS、应用到 DRDS 的连接数、活跃线程三个指标在一定时间段内的监控信息。通过监控信息，可以查看 DRDS 上数据库的运行情况。具体可以参考 DRDS 监控文档。

查看分库信息

在分库管理页面可以查看 RDS 上的分库分表信息，以及各个物理分库名和所在的 RDS，也可以查看每个分库的连接属性和表信息。

操作如下：

在数据库基本信息页，单击左侧菜单栏的**分库管理**。分库列表页面列出了当前的 DRDS 数据库中各个分库与 RDS 实例的对应关系。

单击**数据库名**进入分库信息页，可以看到该分库下有哪些分表，以及 DRDS 连接该分库时使用的连接参数。

数据表管理

在数据表管理页面，可以查看表结构、设置全表扫描、删除表。

如果是建表，请通过支持 MySQL 协议的第三方客户端进行操作，具体请参考快速开始。

设置全表扫描

在执行带有 WHERE 条件的 UPDATE、DELETE、SELECT 语句时，如果 SQL 语句中没有使用拆分键，或者虽然指定了拆分键但是范围太广，会导致 SQL 语句被分发到所有分库上执行，即全表扫描。执行结果会在 DRDS 中进行合并。

在全表扫描时，通过聚合函数，DRDS 支持对全表的统计汇总。目前支持的聚合函数有 COUNT、MAX、MIN、SUM， 另外还支持 LIKE、ORDER BY 与 LIMIT 语法。

注意：全表扫描响应较慢，在高并发业务场景中应尽量避免使用。

全表扫描功能默认是关闭的，需要配置后才能使用。具体配置方式如下：

在 DRDS 控制台左侧菜单栏选择**实例列表**，并单击具体实例名称进入实例基本信息页面。

单击左侧菜单栏的**数据库列表**，选择具体数据库进入数据库基本信息页。

单击左侧菜单栏的**数据表管理**。

单击对应数据表栏内**高级设置**，打开全表扫描。

设置白名单

DRDS 提供了访问控制功能，只有通过白名单规则的 IP 才可以访问。

在数据库基本信息页面，单击左侧菜单栏的**白名单设置**。在白名单设置中，可以查看和设置能够访问该 DRDS 数据库的 IP 地址。

单击**手动修改**按钮，在白名单设置页面，输入允许访问的服务器 IP 地址。

IP 地址支持全 IP、 CIDR 模式和 * 号占位符模式。当配置为 *.*.*.* 或空时，为不设置 IP 访问限制。

数据导入

当需要使用 DRDS 且要保留历史数据时，需要从其它数据库把数据导入到 DRDS 中。DRDS 支持从 RDS 或 MySQL 数据库中将数据导入到 DRDS 数据库。

例如，已有数据存储在 RDS 数据库中，需要迁移到 DRDS，虽然 DRDS 底层存储是 RDS 数据库，但是 DRDS 分库分表的存储方式和 RDS 存储方式不同，就需要把数据从 RDS 导入到 DRDS 中。

- 当待导入的数据量比较小时，如总数据量少于500万条，可以使用 mysqldump 等工具导出数据，再使用 MySQL source 命令将数据导入到 DRDS 数据库里。具体示例请参考**数据导入与导出**文档。
- 当数据量较大时，可以通过 DRDS 控制台导入数据。

从 DRDS 控制台导入数据的步骤如下：

在 DRDS 控制台左侧菜单栏选择**实例列表**，并在列表中单击需要操作的实例数据库名称进入数据库基本信息页。

在基本信息页右上角单击**数据导入**。

执行导入。数据导入是通过阿里云 DTS 服务实现，具体操作可以参考**数据导入操作**。在选择目标库时，实例类型选择 DRDS。

注意：通过 DTS 数据迁移到 DRDS，不支持结构导入，请先在 DRDS 库上建表。

导入主键冲突处理。

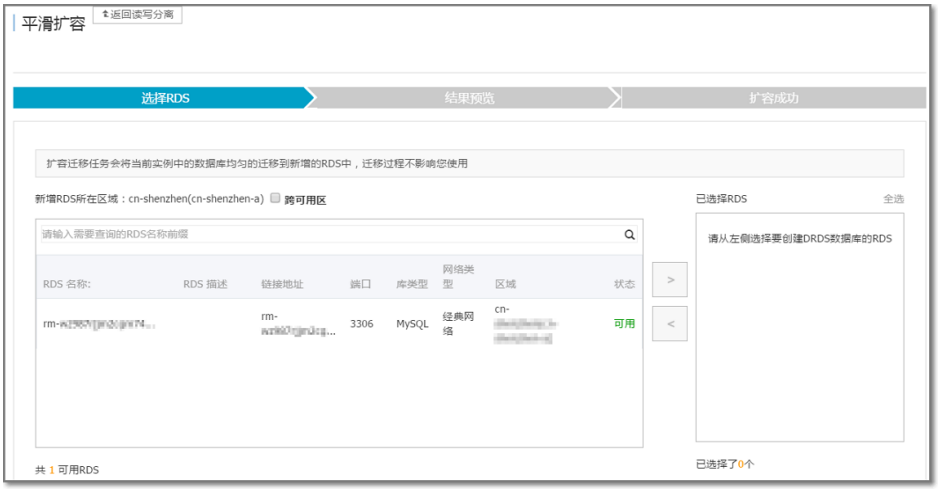
通过 DTS 向 DRDS 导入数据时，由于记录含有主键 ID，会导致 DRDS Sequence 无法变更，产生导入数据主键 ID 值领先于 DRDS Sequence 当前值，从而引起主键冲突的情况。为了避免出错，需要修改 Sequence 起始值，方法可参考 Sequence 修改。

DRDS 平滑扩容

平滑扩容流程分为配置>迁移>切换>清理四个步骤。平滑扩容基本原理请参考**平滑扩容介绍**。

步骤一：新增 RDS 节点

在数据库管理页面右上角，单击**平滑扩容**按钮，进入新增页面。



在 RDS 实例选择页面，添加 RDS 实例，单击**下一步**进入预览页面。

注意：

- 如果需要新增5个或5个以上 RDS 实例，需要事先提工单，以防后端迁移资源不足造成迁移不成功。
- 对于高权限 RDS 实例需要输入高权限账号和密码。

在预览页面中，可以看到迁移到新添加 RDS 上的分库。控制台默认会平均分配分库到新添加的 RDS 实例上。也可以手动向新增的 RDS 实例上添加或删除分库。

单击**开始扩容**按钮，提交平滑扩容任务，此时任务会异步执行。

在数据库管理页面的右上角能够看到任务执行的状态，直到完成。

步骤二：迁移

新增 RDS 后，需要对分库进行迁移。迁移任务不会变更原有数据库中数据，不会影响业务，只是把待迁移库中的数据同步到新增的 RDS 上。在切换前，可以通过回滚，放弃本次平滑扩容操作。

说明：

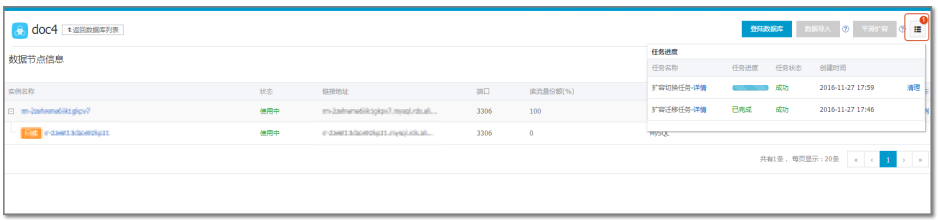
- 在执行切换前，本次扩容还没有对原数据库中数据产生实质影响，因此在切换前都可以通过回滚来放弃本次扩容。
- 扩容期间需要停止清理源 RDS 的 Binlog 文件 ，可能会导致磁盘空间不足，请务必在源 RDS 实例上预留充足的磁盘空间。一般百分之三十以上为宜。如空间实在无法保证，可以提交工单来扩容 RDS 存储空间。
- 源 RDS 实例扩容过程中会有读压力，请尽量在源 RDS 低负载时操作。
- 扩容期间请勿在控制台提交 DDL 任务或通过直接连接 DRDS 执行 DDL 语句，否则会导致扩容任务失败。
- 扩容需要保证源库中所有表具有主键，如果没有需要事先添加好。

历史数据和增量数据迁移完成后，迁移任务进度会达到100%，此时可以进行切换或者回滚，放弃本次扩容。

步骤三：切换

切换任务会将读写流量切换到新增的 RDS 实例上，整个过程会在3~5分钟内完成。在切换过程中，除了会有一到两次闪断，服务不受影响。**请在业务低谷期执行切换。**

在任务列表中，单击**切换**按钮并**确认**。切换过程会生成一个切换任务，并在任务进度中显示。



切换完成后，在**任务进度**中会显示**清理**按钮，表示切换任务已经完成。

步骤四：清理

切换完成后，单击**清理**按钮并确认。此步骤将删除原 RDS 上被迁移的分库。清理任务也是一个异步任务，可以在**任务进度**中查看执行状态。

清理任务完成后，整个平滑扩容过程结束。新增 RDS 实例会成为 DRDS 对应逻辑库新的存储节点。

目前平滑扩容是通过移库的方式来实现扩容。如果扩容到一定程度，出现一个分库超出了单个 RDS 容量，无法进一步平滑扩容时，可以提交工单，申请增加分库数目并扩容。这时会对数据重新进行 HASH 计算，重新分配。

注意：

- **清理任务会删除本次扩容后不再使用的数据库，可以考虑备份后再执行此操作。**
- **清理操作对数据库有一定压力，请在业务低谷期执行。**

拆分变更工具

拆分变更工具是一套可以对表的拆分方式进行变更的工具组合，该工具组合中包含单表转分表、分表转单表和分表转分表等三种变更工具。当某张表因业务数据增长、业务有变更等需要调整拆分方式时，可使用该工具对其作出灵活的变更调整。

原理

拆分变更是通过将旧表数据复制到一张新建的表完成的。如下图所示，数据的复制过程通过一个转换任务完成，转换任务包括全量复制、增量同步、全量校验、等待切换和任务过期等五个阶段。

- 全量复制即将旧表中已有数据全量复制到新表中的过程。
- 增量同步即将全量复制开始后旧表产生的增量数据同步到新表中的过程。
- 全量校验即将新表数据与旧表进行比对，确保两边数据一致的过程。
- 当全量校验完成后新表即处于可用状态，业务上可选择合适的时间点切换到新表。
- 等待切换阶段会持续3天时间，超过3天后转换任务进入过期阶段，过期后，增量会停止同步，若旧表仍有更新或写入，这部分增量将不会同步。

数据复制时仅对旧表产生一些读压力，不会有锁表操作，不影响业务的正常使用。

拆分变更工具使用教程

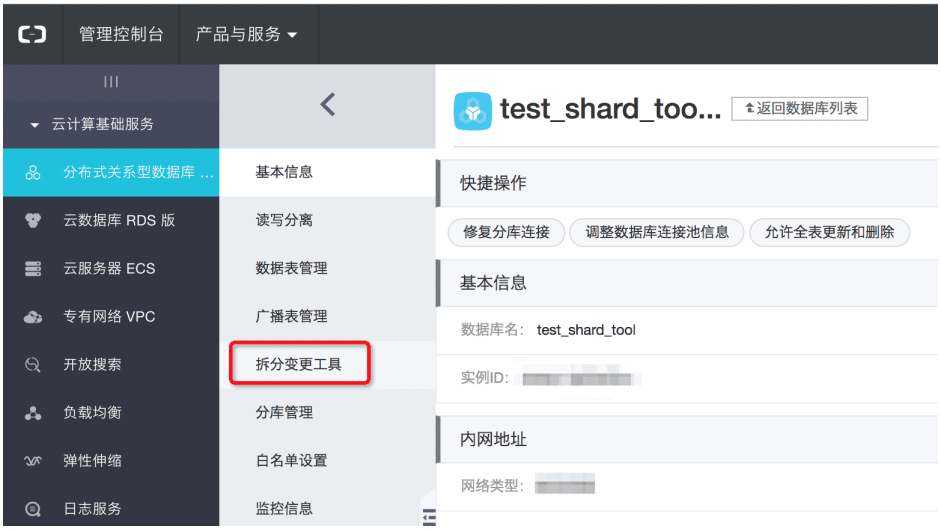
本节以单表转分表为例，讲述如何在DRDS控制台进行操作。

准备新表

在建立转换任务前，需要先准备好新表，除拆分方式和表名称外，新表需要与旧表结构相同，如何建表请参考这里。

建立转换任务

在DRDS控制台选择相应数据库的管理页面，进入“拆分变更工具”菜单，选择“单表转分表”一项。



第一步是选择旧表，本示例中该页面会列出库中所有的单表。

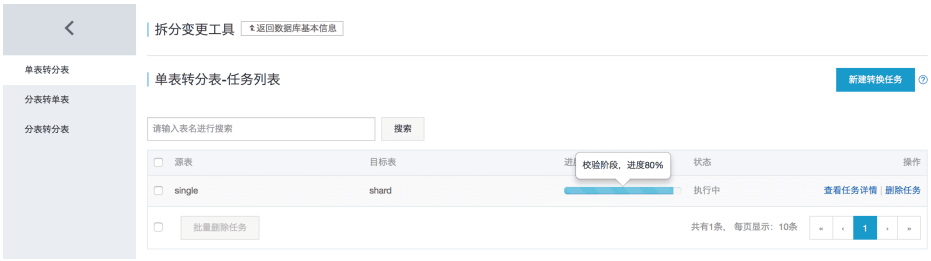
第二步是选择新表，本示例中该页面会列出库中所有分表。

第三步是任务预检，即检测新表是否符合转换要求，具体要求可查看后面的【注意事项】。

第四步是任务预览，即确认旧表和新表是待转换的表。

查看转换任务进度

转换任务建立后会出现在“单表转分表”任务列表页面，该页面会显示任务目前的进度和状态。



另外，可通过点击“查看任务详情”获取更详细的任务信息。

注意事项

使用范围

- 拆分变更工具需要先准备新表，且新表与旧表需在同一个数据库内。

转换任务预检项目

- 转换任务要求新表除名称和拆分方式外，其他结构与旧表保持完全一致，即新表中列的名字、数据类型、长度、非空约束等需要与旧表相同。
- 为保证数据复制的一致性，新表要求数据为空。在新建转换任务时，若新表中有数据，转换任务不会清空新表数据，但会给出任务无法创建的提示。
- 若转换任务类型为“分表到单表”或“分表到分表”，新建转换任务时，会对旧表和新表是否有主键和唯一键进行检测，当旧表或新表中有主键或唯一键时，转换任务可以进行，但因

复制过程中涉及数据的重新分布，会有主键或唯一键冲突的风险，所以请根据自己的数据特征评估是否可以正常转换。

其他

- 转换期间旧表可正常使用，当转换任务的状态为“完成”时即表示进入等待切换阶段，请在任务有效期（即3天）内完成切换。
- 切换后可删除转换任务，需要注意的是，删除转换任务不会删除旧表或表里的数据，也不会删除已经同步的数据。
- 另外，转换任务运行期间需避免对旧表或新表做结构变更。

DRDS DDL 语法

DDL 简介

DRDS DDL 的建表语句跟 MySQL DDL 的建表语句类似，并在 MySQL 建表语法的基础上进行了扩充：创建拆分表时，DRDS 需要明确指定分库分表的拆分方式，增加了 drds_partition_options 拆分选项，包括 DBPARTITION BY、TBPARTITION BY、TBPARTITIONS、BROADCAST。

目前有三种方式执行 DDL 语句：

使用 MySQL 命令行客户端（比如 MySQL 命令行、Navicat、MySQL Workbench）。

使用 DMS（Data Management Service）的命令窗口执行（我们提供的 Web 界面版的 MySQL 客户端，在 DRDS 控制台点击登录数据库按钮登录）。

在程序中连接 DRDS，调用 DDL 语句执行。

创建表

创建表的语法和方式

本文主要介绍使用 DDL 语句进行建表的语法、子句和参数，以及基本方式。

注意：DRDS 目前不支持使用 DDL 语句直接建库，请登录 DRDS 控制台进行创建。具体操作指南请参考创建 DRDS 数据库。

语法：

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [drds_partition_options]
    [partition_options]

drds_partition_options:
    DBPARTITION BY
        HASH([column])
        [TBPARTITION BY
            { HASH(column)
            | {MM|DD|WEEK|MMDD}(column)}
          [TBPARTITIONS num]
        ]
```

分库分表的子句和参数：

DBPARTITION BY hash(partition_key)：指定分库键和分库算法，不支持按照时间分库；

TBPARTITION BY { HASH(column) | {MM|DD|WEEK|MMDD}(column)（可选）：默认与 DBPARTITION BY 相同，指定物理表使用什么方式映射数据；

TBPARTITIONS num（可选）：每个库上的物理表数目（默认为1），如不分表，就不需要指定该字段。

分库分表的基本方式及补充：

- 单库单表（不拆分）
- 只分库不分表
- 既分库又分表
- 使用主键作为拆分字段
- 广播表
- 其他 MySQL 建表属性

单库单表

本文介绍如何创建单库单表，以及如何在建单库单表时进行指定。

建一张单库单表，不做任何拆分。

```
CREATE TABLE single_tbl(
id int,
name varchar(30),
primary key(id)
);
```

查看逻辑表的节点拓扑，可以看出只在 0 库创建了一张单库单表的逻辑表。

```
mysql> show topology from single_tbl;
+------+--------------------------------------------------------------------+------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+--------------------------------------------------------------------+------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | single_tbl |
+------+--------------------------------------------------------------------+------------+
1 row in set (0.01 sec)
```

指定

单库单表建表的时候也可以指定（select_statement），拆分表则不支持指定。

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
[(create_definition,...)]
[table_options]
[partition_options]
select_statement
```

示例：建一张单库单表 single_tbl2，数据来自表 single_tbl，不做任何拆分。

```
CREATE TABLE single_tbl2(
id int,
name varchar(30),
primary key(id)
) select * from single_tbl;
```

分库不分表

本文介绍如何在建表时只分库不分表。

假设已经建好的分库数为 8，建一张表，只分库不分表，分库方式为根据 id 列哈希。

```
CREATE TABLE multi_db_single_tbl(
id int,
name varchar(30),
primary key(id)
) dbpartition by hash(id);
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 1 张分表，既只做了分库。

```
mysql> show topology from multi_db_single_tbl;
+------+--------------------------------------------------------------------+--------------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+--------------------------------------------------------------------+--------------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_single_tbl |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_single_tbl |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_single_tbl |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_single_tbl |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_single_tbl |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_single_tbl |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_single_tbl |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_single_tbl |
+------+--------------------------------------------------------------------+--------------------+
8 rows in set (0.01 sec)
```

分库分表

本文介绍如何使用不同的拆分方式进行分库分表：

- 使用哈希函数做拆分
- 使用双字段哈希函数做拆分
- 使用日期做拆分

以下示例均假设已经建好的分库数为 8。

使用哈希函数做拆分

建一张表，既分库又分表，每个库含有 3 张物理表，分库拆分方式为按照 id 列进行哈希，分表拆分方式为按照 bid 列进行哈希（先根据 id 列的值进行哈希运算，将表中数据分布在多个子库中，每个子库中的数据再根据 bid 列值的哈希运算结果分布在3个物理表中）。

```
CREATE TABLE multi_db_multi_tbl(
id int auto_increment,
bid int,
name varchar(30),
primary key(id)
) dbpartition by hash(id) tbpartition by hash(bid) tbpartitions 3;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 3 张分表。

```
mysql> show topology from multi_db_multi_tbl;
+------+--------------------------------------------------------------------+----------------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+--------------------------------------------------------------------+----------------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_00 |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_01 |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | multi_db_multi_tbl_02 |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_03 |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_04 |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | multi_db_multi_tbl_05 |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_06 |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_07 |
| 8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0002_RDS | multi_db_multi_tbl_08 |
| 9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_09 |
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_10 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0003_RDS | multi_db_multi_tbl_11 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_12 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_13 |
| 14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0004_RDS | multi_db_multi_tbl_14 |
| 15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_15 |
| 16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_16 |
| 17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0005_RDS | multi_db_multi_tbl_17 |
| 18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_18 |
| 19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_19 |
| 20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0006_RDS | multi_db_multi_tbl_20 |
| 21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_21 |
| 22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_22 |
| 23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | multi_db_multi_tbl_23 |
+------+--------------------------------------------------------------------+----------------------+
24 rows in set (0.01 sec)
```

查看该逻辑表的拆分规则，可以看出分库分表的拆分拆分方式均为哈希，分库的拆分键为 id，分表的拆分键为 bid。

```
mysql> show rule from multi_db_multi_tbl;
+------+--------------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+--------------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| 0 | multi_db_multi_tbl | 0 | id | hash | 8 | bid | hash | 3 |
+------+--------------------+-----------+-----------------+--------------------+--------------------+-----------------
```

```
+--------------------+--------------------+
1 row in set (0.01 sec)
```

使用双字段哈希函数做拆分

使用要求

拆分键的类型必须是字符类型或数字类型。

路由方式

根据任一拆分键后 N 位计算哈希值，以哈希方式完成路由计算。N 为函数第三个参数。例如：RANGE_HASH(COL1, COL2, N)，计算时会优先选择 COL1 ，截取其后 N 位进行计算。 COL1 不存在时按 COL2 计算。

适用场景

适合于需要有两个拆分键，并且仅使用其中一个拆分键值进行查询时的场景。例如，假设用户的 DRDS 里已经分了 8 个物理库, 现业务有如下的场景：

i. 一个业务想按买家 ID 和订单 ID 对订单表进行分库；
ii. 查询时条件仅有买家 ID 或订单 ID。

此时可使用以下 DDL 对订单表进行构建：

```
 create table test_order_tb (
id int,
seller_id varchar(30) DEFAULT NULL,
order_id varchar(30) DEFAULT NULL,
buyer_id varchar(30) DEFAULT NULL,
create_time datetime DEFAULT NULL,
primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by RANGE_HASH(buyer_id, order_id, 10) tbpartition by RANGE_HASH(buyer_id, order_id, 10) tbpartitions 3;
```

注意事项

- 两个拆分键皆不能修改。
- 插入数据时如果发现两个拆分键指向不同的分库或分表时，插入会失败。

使用日期做拆分

除了可以使用哈希函数做拆分算法，您还可以使用日期函数 MM/DD/WEEK/MMDD 来作为分表的拆分算法，具体参照以下五个示例：

建一张表，既分库又分表，分库方式为根据 userId 列哈希，分表方式为根据 actionDate 列，按照一周七天来拆分（WEEK(actionDate)计算的是DAY_OF_WEEK）。

比如 actionDate 列的值是 2017-02-27，这天是星期一，WEEK(actionDate)算出的值是2，该条记录就会被存储到 2（2 % 7 = 2）这张分表（位于某个分库，具体的表名是 user_log_2）；比如 actionDate 列的值是 2017-02-26，这天是星期天，WEEK(actionDate)算出的值是1，该条记录就会被存储到 1（1 % 7 = 1）这张分表（位于某个分库，具体的表名是 user_log_1）。

```
CREATE TABLE user_log(
userId int,
name varchar(30),
operation varchar(30),
actionDate DATE
) dbpartition by hash(userId) tbpartition by WEEK(actionDate) tbpartitions 7;
```

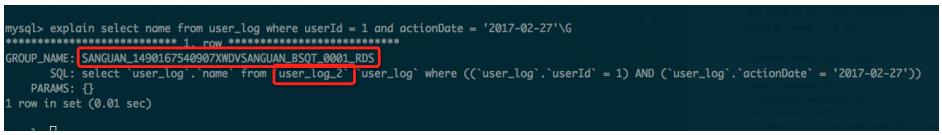
查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 7 张分表（一周7天）。以下示例中返回的结果较长，用...做了省略处理。

```
mysql> show topology from user_log;
+------+--------------------------------------------------------------------+------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+--------------------------------------------------------------------+------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_0 |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_1 |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_2 |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_3 |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_4 |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_5 |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log_6 |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_0 |
| 8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_1 |
| 9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_2 |
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_3 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_4 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_5 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log_6 |
...
| 49 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_0 |
| 50 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_1 |
| 51 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_2 |
| 52 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_3 |
| 53 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_4 |
| 54 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_5 |
| 55 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log_6 |
+------+--------------------------------------------------------------------+------------+
56 rows in set (0.01 sec)
```

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 userId，分表的拆分方式为按照时间函数 WEEK 进行拆分，分表的拆分键为 actionDate。

```
mysql> show rule from user_log;
+------+-----------+----------+-----------------+--------------------+--------------------+-----------------
-+--------------------+--------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT
| TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+-----------+----------+-----------------+--------------------+--------------------+-----------------
-+--------------------+--------------------+
| 0 | user_log | 0 | userId | hash | 8 | actionDate | week | 7 |
+------+-----------+----------+-----------------+--------------------+--------------------+-----------------
-+--------------------+--------------------+
1 row in set (0.00 sec)
```

查看给定分库键和分表键参数时， SQL 被路由到哪个物理分库和该物理分库下的哪张物理表。



建一张表，既分库又分表，分库方式为根据 userId 列哈希，分表方式为根据 actionDate 列，按照一年 12 个月进行拆分（MM(actionDate)计算的是MONTH_OF_YEAR）。

比如 actionDate 列的值是 2017-02-27，MM(actionDate)算出的值是02，该条记录就会被存储到02（02 % 12 = 02）这张分表（位于某个分库，具体的表名是 user_log_02）；比如 actionDate 列的值是 2016-12-27，MM(actionDate)算出的值是 12，该条记录就会被存储到00（12 % 12 = 00）这张分表（位于某个分库，具体的表名是 user_log_00）。

```
CREATE TABLE user_log2(
userId int,
name varchar(30),
operation varchar(30),
actionDate DATE
) dbpartition by hash(userId) tbpartition by MM(actionDate) tbpartitions 12;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 12 张分表（1 年有 12 个月）。由于返回结果较长，这里用...做了省略处理。

```
mysql> show topology from user_log2;
+------+-----------------------------------------------------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+-----------------------------------------------------------------+--------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_00 |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_01 |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_02 |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_03 |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_04 |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_05 |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_06 |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_07 |
| 8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_08 |
| 9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_09 |
```

```
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_10 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log2_11 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_00 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_01 |
| 14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_02 |
| 15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_03 |
| 16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_04 |
| 17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_05 |
| 18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_06 |
| 19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_07 |
| 20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_08 |
| 21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_09 |
| 22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_10 |
| 23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0001_RDS | user_log2_11 |
...
| 84 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_00 |
| 85 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_01 |
| 86 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_02 |
| 87 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_03 |
| 88 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_04 |
| 89 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_05 |
| 90 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_06 |
| 91 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_07 |
| 92 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_08 |
| 93 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_09 |
| 94 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_10 |
| 95 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log2_11 |
+------+-----------------------------------------------------------------+--------------+
96 rows in set (0.02 sec)
```

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 userId，分表的拆分方式为按照时间函数 MM 进行拆分，分表的拆分键为 actionDate。

```
mysql> show rule from user_log2;
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| 0 | user_log2 | 0 | userId | hash | 8 | actionDate | mm | 12 |
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
1 row in set (0.00 sec)
```

建一张表，既分库又分表，分库方式为根据 userId 列哈希，分表方式为按照一个月 31 天进行拆分（函数DD(actionDate)计算的是DAY_OF_MONTH）。

比如 actionDate 列的值是 2017-02-27，DD(actionDate)算出的值是 27，该条记录就会被存储到 27（27 % 31 = 27）这张分表（位于某个分库，具体的表名是 user_log_27）。

```
CREATE TABLE user_log3(
userId int,
name varchar(30),
operation varchar(30),
actionDate DATE
) dbpartition by hash(userId) tbpartition by DD(actionDate) tbpartitions 31;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 31 张分表（按每个月有31天处理）。由于返回的结果较长，这里用...做了省略处理。

```
mysql> show topology from user_log3;
+------+--------------------------------------------------------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+--------------------------------------------------------------------+--------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_00 |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_01 |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_02 |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_03 |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_04 |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_05 |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_06 |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_07 |
| 8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_08 |
| 9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_09 |
| 10 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_10 |
| 11 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_11 |
| 12 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_12 |
| 13 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_13 |
| 14 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_14 |
| 15 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_15 |
| 16 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_16 |
| 17 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_17 |
| 18 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_18 |
| 19 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_19 |
| 20 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_20 |
| 21 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_21 |
| 22 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_22 |
| 23 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_23 |
| 24 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_24 |
| 25 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_25 |
| 26 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_26 |
| 27 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_27 |
| 28 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_28 |
| 29 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_29 |
| 30 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log3_30 |
...
| 237 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_20 |
| 238 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_21 |
| 239 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_22 |
| 240 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_23 |
| 241 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_24 |
| 242 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_25 |
| 243 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_26 |
| 244 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_27 |
| 245 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_28 |
```

```
| 246 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_29 |
| 247 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log3_30 |
+------+-----------------------------------------------------------------+--------------+
248 rows in set (0.01 sec)
```

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 userId，分表的拆分方式为按照时间函数 DD 进行拆分，分表的拆分键为 actionDate。

```
mysql> show rule from user_log3;
+------+------------+-----------+-----------------+--------------------+-------------------+-----------------
+--------------------+-------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+-------------------+-----------------
+--------------------+-------------------+
| 0 | user_log3 | 0 | userId | hash | 8 | actionDate | dd | 31 |
+------+------------+-----------+-----------------+--------------------+-------------------+-----------------
+--------------------+-------------------+
1 row in set (0.01 sec)
```

建一张表，既分库又分表，分库方式为根据 userId 列哈希，分表方式为按照一年 365 天进行拆分，路由到 365 张物理表（MMDD(actionDate) tbpartitions 365计算的是DAY_OF_YEAR % 365）。

比如 actionDate 列的值是 2017-02-27，MMDD(actionDate)算出的值是 58，该条记录就会被存储到 58 这张分表（位于某个分库，具体的表名是user_log_58）。

```
CREATE TABLE user_log4(
userId int,
name varchar(30),
operation varchar(30),
actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 365;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 365 张分表（按每年有 365 天处理）。由于返回的结果较长，这里用...做了省略处理。

```
mysql> show topology from user_log4;
+------+-----------------------------------------------------------------+---------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+-----------------------------------------------------------------+---------------+
...
| 2896 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_341
|
| 2897 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_342
|
| 2898 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_343
|
| 2899 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_344
```

```
|
| 2900 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_345
|
| 2901 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_346
|
| 2902 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_347
|
| 2903 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_348
|
| 2904 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_349
|
| 2905 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_350
|
| 2906 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_351
|
| 2907 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_352
|
| 2908 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_353
|
| 2909 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_354
|
| 2910 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_355
|
| 2911 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_356
|
| 2912 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_357
|
| 2913 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_358
|
| 2914 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_359
|
| 2915 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_360
|
| 2916 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_361
|
| 2917 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_362
|
| 2918 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_363
|
| 2919 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log4_364
|
+------+-------------------------------------------------------------+---------------+
2920 rows in set (0.07 sec)
```

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 userId，分表的拆分方式为按照时间函数 MMDD 进行拆分，分表的拆分键为 actionDate。

```
mysql> show rule from user_log4;
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+-----------------+--------------------+--------------------+-----------------
+--------------------+--------------------+
| 0 | user_log4 | 0 | userId | hash | 8 | actionDate | mmdd | 365 |
```

```
+------+------------+-----------+-----------------+--------------------+-------------------+-----------------
+--------------------+-------------------+
1 row in set (0.02 sec)
```

建一张表，既分库又分表，分库方式为根据 userId 列哈希，分表方式为按照一年 365 天进行拆分，路由到 10 张物理表（ MMDD(actionDate) tbpartitions 10计算的是DAY_OF_YEAR % 10 ）。

```
CREATE TABLE user_log5(
userId int,
name varchar(30),
operation varchar(30),
actionDate DATE
) dbpartition by hash(userId) tbpartition by MMDD(actionDate) tbpartitions 10;
```

查看该逻辑表的节点拓扑，可以看出在每个分库都创建了 10 张分表（按照一年 365 天进行拆分，路由到 10 张物理表）。由于返回的结果较长，这里用...做了省略处理。

```
mysql> show topology from user_log5;
+------+-------------------------------------------------------------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+-------------------------------------------------------------------------+--------------+
| 0 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_00 |
| 1 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_01 |
| 2 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_02 |
| 3 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_03 |
| 4 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_04 |
| 5 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_05 |
| 6 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_06 |
| 7 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_07 |
| 8 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_08 |
| 9 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0000_RDS | user_log5_09 |
...
| 70 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_00 |
| 71 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_01 |
| 72 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_02 |
| 73 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_03 |
| 74 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_04 |
| 75 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_05 |
| 76 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_06 |
| 77 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_07 |
| 78 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_08 |
| 79 | SANGUAN_TEST_123_1488766060743ACTJSANGUAN_TEST_123_WVVP_0007_RDS | user_log5_09 |
+------+-------------------------------------------------------------------------+--------------+
80 rows in set (0.02 sec)
```

查看该逻辑表的拆分规则，可以看出分库的拆分方式为哈希，分库的拆分键为 userId，分表的拆分方式为按照时间函数 MMDD 进行拆分，路由到10张物理表，分表的拆分键为 actionDate。

```
mysql> show rule from user_log5;
+------+------------+-----------+-----------------+--------------------+-------------------+-----------------
-+---------------------+--------------------+
```

```
| ID | TABLE_NAME | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT
| TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+------------+-----------+------------------+---------------------+-------------------+-----------------
-+---------------------+--------------------+
| 0 | user_log5 | 0 | userId | hash | 8 | actionDate | mmdd | 10 |
+------+------------+-----------+------------------+---------------------+-------------------+-----------------
-+---------------------+--------------------+
1 row in set (0.01 sec)
```

使用主键作为拆分字段

当拆分算法不指定任何拆分字段时，系统默认使用主键作为拆分字段。以下示例将介绍如何使用主键当分库和分表键。

使用主键当分库键

```
CREATE TABLE prmkey_tbl(
id int,
name varchar(30),
primary key(id)
) dbpartition by hash();
```

使用主键当分库分表键

```
CREATE TABLE prmkey_multi_tbl(
id int,
name varchar(30),
primary key(id)
) dbpartition by hash() tbpartition by hash() tbpartitions 3;
```

广播表

子句BROADCAST用来指定创建广播表。广播表是指将这个表复制到每个分库上，在分库上通过同步机制实现数据一致，有秒级延迟。这样做的好处是可以将 JOIN 操作下推到底层的 RDS（MySQL），来避免跨库 JOIN。SQL 优化方法 详细讲述了如何使用广播表来做 SQL 优化。

```
CREATE TABLE brd_tbl(
  id int,
  name varchar(30),
  primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 BROADCAST;
```

其他 MySQL 建表属性

您在分库分表的同时还可以指定其他的 MySQL 建表属性，例如：

```
CREATE TABLE multi_db_multi_tbl(
id int,
name varchar(30),
primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(id) tbpartition by hash(id) tbpartitions 3;
```

修改表

修改表（ALTER TABLE）

语法：

```
ALTER [ONLINE|OFFLINE] [IGNORE] TABLE tbl_name
    [alter_specification [, alter_specification] ...]
    [partition_options]
```

日常 DDL 操作，如增加列、增加索引、修改数据定义，可以使用 DRDS DDL 语句方式进行。详细语法请参考 MySQL 修改表语法。

注意：如果修改的是拆分表，则不允许修改拆分字段。

增加列：

```
ALTER TABLE user_log
  ADD COLUMN idcard varchar(30);
```

增加索引：

```
ALTER TABLE user_log
  ADD INDEX idcard_idx (idcard);
```

删除索引：

```
ALTER TABLE user_log
  DROP INDEX idcard_idx;
```

修改字段：

```
ALTER TABLE user_log
  MODIFY COLUMN idcard varchar(40);
```

删除表

删除表（DROP TABLE）

语法：

```
DROP [TEMPORARY] TABLE [IF EXISTS]
tbl_name [, tbl_name] ...
[RESTRICT | CASCADE]
```

通过 DRDS 删除表与删除 MySQL 表没有区别，系统会自动处理相关物理表的删除操作，详细语法请参考 MySQL 删除表语法参考。

删除表 user_log

```
DROP TABLE user_log;
```

DDL 常见问题处理

建表的时候执行出错怎么办？

DRDS DDL 的执行是一个分布式处理过程，出错可能导致各个分片表结构不一致，所以需要进行手动清理。

步骤如下：

DRDS 会提供基本的错误描述信息，比如语法错误等。如果错误信息太长，则会提示用户调用 SHOW WARNINGS 的 SQL 命令来查看每个分库执行失败的原因。

可以通过 SHOW TOPOLOGY 命令来查看物理表的拓扑结构。SHOW TOPOLOGY的具体用法请参考 DRDS 控制指令。

```
SHOW TOPOLOGY FROM multi_db_multi_tbl;
+------+-----------------+----------------------+
| ID   | GROUP_NAME      | TABLE_NAME           |
+------+-----------------+----------------------+
|    0 | corona_qatest_0 | multi_db_multi_tbl_00 |
|    1 | corona_qatest_0 | multi_db_multi_tbl_01 |
|    2 | corona_qatest_0 | multi_db_multi_tbl_02 |
|    3 | corona_qatest_1 | multi_db_multi_tbl_03 |
|    4 | corona_qatest_1 | multi_db_multi_tbl_04 |
|    5 | corona_qatest_1 | multi_db_multi_tbl_05 |
|    6 | corona_qatest_2 | multi_db_multi_tbl_06 |
|    7 | corona_qatest_2 | multi_db_multi_tbl_07 |
|    8 | corona_qatest_2 | multi_db_multi_tbl_08 |
|    9 | corona_qatest_3 | multi_db_multi_tbl_09 |
|   10 | corona_qatest_3 | multi_db_multi_tbl_10 |
|   11 | corona_qatest_3 | multi_db_multi_tbl_11 |
+------+-----------------+----------------------+
12 rows in set (0.21 sec)
```

使用 CHECK TABLE tablename指令来查看逻辑表是否创建成功。比如下面的例子展示了 multi_db_multi_tbl 的某个物理分表没有创建成功。

```
mysql> check table multi_db_multi_tbl;
+-----------------------------------------------+-------+----------+-------------------------------------------------------------------------+
| TABLE                                         | OP    | MSG_TYPE | MSG_TEXT                                                                |
+-----------------------------------------------+-------+----------+-------------------------------------------------------------------------+
| andor_mysql_qatest. multi_db_multi_tbl | check | Error    | Table 'corona_qatest_0. multi_db_multi_tbl_02' doesn't exist |
+-----------------------------------------------+-------+----------+-------------------------------------------------------------------------+
1 row in set (0.16 sec)
```

使用幂等的方式继续执行建表操作或删表操作，会创建或删除剩余的物理表。

```
CREATE TABLE IF NOT EXISTS table1
(id int, name varchar(30), primary key(id))
dbpartition by hash(id);
DROP TABLE IF EXISTS table1;
```

建索引失败、加列失败怎么办？

建索引失败、加列失败的处理方法跟上面建表失败的处理类似。具体请参考 DDL 异常处理文档。

DRDS DDL 拆分函数

DDL 拆分函数概述

DRDS DDL 拆分函数对分库分表的支持情况

DRDS 是一个支持既分库又分表的数据库服务。目前 DRDS 分库函数与分表函数的支持情况如下：

拆分函数	描述	是否支持用于分库	是否支持用于分表
HASH	简单取模	是	是
UNI_HASH	简单取模	是	是
RIGHT_SHIFT	数值向右移	是	是
RANGE_HASH	双拆分列哈希	是	是
MM	按月份哈希	否	是
DD	按日期哈希	否	是
WEEK	按周哈希	否	是
MMDD	按月日哈希	否	是
YYYYMM	按年月哈希	是	是
YYYYWEEK	按年周哈希	是	是
YYYYDD	按年日哈希	是	是
YYYYMM_OPT	按年月哈希，改进型	是	是

YYYYWEEK_OPT	按年周哈希，改进型	是	是
YYYYDD_OPT	按年日哈希，改进型	是	是

DRDS 分库分表拆分方式的注意点

在 DRDS 中，一张逻辑表的**拆分方式是由拆分函数（包括分片数目与路由算法）与拆分键（包括拆分键的 MySQL 数据类型）共同定义。**

只有当 DRDS 的分库函数与分表函数相同并且分库键与分表键也相同时，才会被认为分库与分表都使用了共同的拆分方式。这样的方式可以让 DRDS 可以根据拆分键的值唯一定位到一个物理分库与一张物理分表。

当一张逻辑表的分库拆分方式与分表拆分方式不一致时，若 SQL 查询没有同时带上分库条件与分表条件，则 DRDS 在查询过程会产生全分库扫描或全分表扫描的操作。

DRDS DDL 拆分函数的数据类型支持情况

DRDS 的拆分函数对各数据类型对支持情况有所不同，下表显示了 DRDS 拆分函数对各种数据类型的支持情况（√ 表示支持，× 表示不支持）：

拆分函数	BIGINT	INT	MEDIUMINT	SMALLINT	TINYINT	VARCHAR	CHAR	DATE	DATETIME	TIMESTAMP	其它类型
HASH	√	√	√	√	√	√	√	×	×	×	×
UNI_HASH	√	√	√	√	√	√	√	×	×	×	×
RANGE_HASH	√	√	√	√	√	√	√	×	×	×	×
RIGHT_SHIFT	√	√	√	√	√	×	×	×	×	×	×
MM	×	×	×	×	×	×	×	√	√	√	×
DD	×	×	×	×	×	×	×	√	√	√	×
WEEK	×	×	×	×	×	×	×	√	√	√	×
MMDD	×	×	×	×	×	×	×	√	√	√	×
YYYYMM	×	×	×	×	×	×	×	√	√	√	×
YYYYWEEK	×	×	×	×	×	×	×	√	√	√	×
YYYYDD	×	×	×	×	×	×	×	√	√	√	×
YYYYMM_OPT	×	×	×	×	×	×	×	√	√	√	×
YYYYWEEK_OPT	×	×	×	×	×	×	×	√	√	√	×
YYYYDD_OPT	×	×	×	×	×	×	×	√	√	√	×

DRDS 的 DDL 拆分函数的语法说明

DRDS 兼容 MySQL 的 DDL 表操作语法，并添加了drds_partition_options的分库分表关键字如下：

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
(create_definition,...)
[table_options]
```

```
[drds_partition_options]
[partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
[(create_definition,...)]
[table_options]
[drds_partition_options]
[partition_options]
select_statement

drds_partition_options:
DBPARTITION BY
{ {HASH|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}([column])}
[TBPARTITION BY
{ {HASH|MM|DD|WEEK|MMDD|YYYYMM|YYYYWEEK|YYYYDD|YYYYMM_OPT|YYYYWEEK_OPT|YYYYDD_OPT}(column)}
[TBPARTITIONS num]
]
```

拆分函数使用说明

HASH

使用要求

拆分键的数据类型必须是整数类型或字符串类型。

该拆分函数对 DRDS 版本无要求，默认支持。

路由方式

使用 HASH 分库但不使用同一个拆分键进行 HASH 分表时，根据分库键的键值直接按分库数取余。如果键值是字符串，则字符串会被计算成哈希值再进行计算，完成路由计算，例如： HASH（‘8’）等价于 8 % D（D 是分库数目）。

分库和分表都使用同一个拆分键进行 HASH 时，根据拆分键的键值按总的分表数取余。例如，有 2 个分库，每个分库 4 张分表，那么 0 库上保存分表 0~3，1 库上保存分表 4~7。某个键值为 15，15 % (2 * 4) = 7，所以 15 被分到 1 库的表 7 上。

使用场景

适合于需要按用户 ID 或订单 ID 进行分库的场景。

适合于拆分键是字符串类型的场景。

使用示例

假设用户需要对 ID 列按 HASH 函数进行分库不分表，则应该使用如下的建表 DDL：

```
create table test_hash_tb (
id int,
name varchar(30) DEFAULT NULL,
create_time datetime DEFAULT NULL,
primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by HASH(ID);
```

注意事项

- 哈希算法是简单取模，要求拆分列的值的自身分布均衡才能保证哈希均衡。

UNI_HASH

使用要求

拆分键的数据类型必须是整数类型或字符串类型。

DRDS 实例的版本必须是 5.1.28-1508068 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

使用 UNI_HASH 分库时，根据分库键的键值直接按分库数取余。如果键值是字符串，则字符串会被计算成哈希值再进行计算，完成路由计算，例如： HASH（‘8’）等价于 8 % D（D 是分库数目）。

分库和分表都使用同一个拆分键进行 UNI_HASH 时，仍然保证先根据分库键键值按分库数取余(HASH不是)，再均匀散布到该分库的各个分表上。

使用场景

适合于需要按用户 ID 或订单 ID 进行分库的场景。

适合于拆分键是整数或字符串类型的场景。

两张逻辑表需要根据同一个拆分键进行分库，两张表的分表数不同，又经常会按该拆分键进行 JOIN 的场景。

与 HASH 的比较

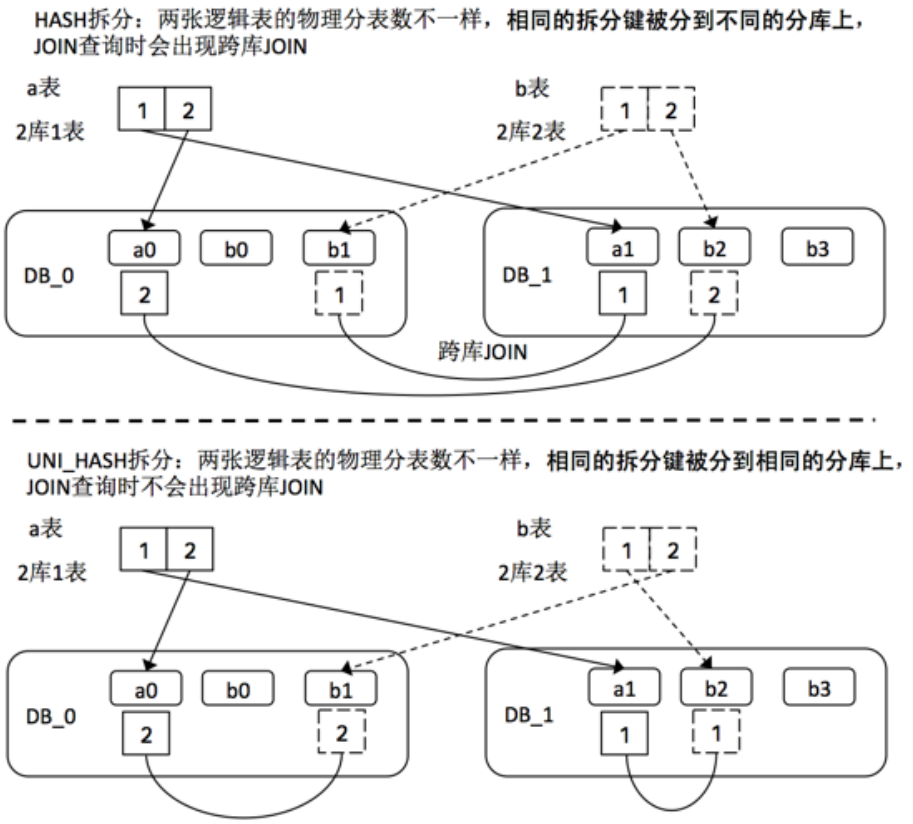
在使用 UNI_HASH 分库但不分表时，UNI_HASH 和 HASH 的路由方式一样，都是根据分库键的键值按分库数取余。

分库和分表都使用同一个拆分键进行 HASH 时，随着分表数的变化，同一个键值分到的分库不是固定的。

分库和分表都使用同一个拆分键进行 UNI_HASH 时，无论分表数是多少，同一个键值总是分到相同的分库。

如果两张逻辑表需要根据同一个拆分键进行分库分表，但分表数不同，那么当两张表按该拆分键进行 JOIN 时，如果使用 HASH 会出现跨库 JOIN，而 UNI_HASH 不会有跨库 JOIN。

例如，假设用户有 2 个分库，有 2 张逻辑表：a 表每库 1 张分表，b表每库 2 张分表。下图展示了分别使用 HASH 和 UNI_HASH 进行拆分后，a 表和 b 表进行 JOIN 的情景：



使用示例

假设用户需要对 ID 列按 UNI_HASH 函数进行分库分表，每库 4 表，则应该执行以下建表 DDL 语句：

```
create table test_hash_tb (
id int,
name varchar(30) DEFAULT NULL,
create_time datetime DEFAULT NULL,
primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by UNI_HASH(ID)
tbpartition by UNI_HASH(ID) tbpartitions 4;
```

注意事项

UNI_HASH 算法是简单取模，要求拆分列的值的自身分布均衡才能保证哈希均衡。

RIGHT_SHIFT

使用要求

拆分键的类型必须整数类型。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的键值（键值必须是整数）有符号地向右移二进制移指定的位数（位数由用户通过 DDL 指定），然后将得到的整数值按分库（表）数目取余。

使用场景

当拆分键的大部分的键值的低位部分区分度比较低而高位部分区分度比较高时，则适用于通过此拆分函数提高散列结果的均匀度。

例如：有 4 个拆分键的键值，分别为 12340000、12350000、12460000 与 1233000， 这 4 个值低 4 位部分都是 0000，直接通过取余散列效果比较差，但如是通过 RIGHT_SHIFT（shardKey, 4）将拆分键的值进行向右移 4 位，则变成了 1234、1235、1246 与 1233，这样的散列效果就变得比较好。

使用示例

假设想将 ID 作为拆分键，并且想将 ID 的值向右移二进制 4 位的值作为哈希值，则可以如下建表：

```
create table test_hash_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by RIGHT_SHIFT(id, 4)
tbpartition by RIGHT_SHIFT(id, 4) tbpartitions 2;
```

注意事项

移位的数目不能超过整数类型所占有的位数目。

RANGE_HASH

使用要求

拆分键的类型必须是字符类型或数字类型。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据任一拆分键后 N 位计算哈希值，然后再按分库数去取余，完成路由计算。N 为函数第三个参数。

例如：RANGE_HASH(COL1, COL2, N) ，计算时会优先选择 COL1，截取其后 N 位进行计算。 COL1 不存在时找 COL2。

适用场景

适合于需要有两个拆分键，并且查询时仅有其中一个拆分键值的场景。

使用示例

例如，假设用户的 DRDS 里已经分了 8 个物理库，现业务有如下的场景：

一个业务想按买家 ID 和订单 ID 对订单表进行分库；查询时条件仅有买家 ID 或订单 ID。

此时可使用以下 DDL 对订单表进行构建：

```
create table test_order_tb (
    id int,
    buyer_id varchar(30) DEFAULT NULL,
    order_id varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8
    dbpartition by RANGE_HASH(buyer_id,order_id, 10)
    tbpartition by RANGE_HASH (buyer_id,order_id, 10) tbpartitions 3;
```

注意事项

- 两个拆分键皆不能修改。
- 插入数据时如果发现两个拆分键指向不同的分库或分表时，插入会失败。

MM

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

只能作为分表函数使用，但不能作为分库函数。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值的月份数进行取余运算并得到分表下标。

使用场景

MM 适用于按月份数进行分表，分表的表名就是月份数。

使用示例

假设先按 id 对用户进行分库，再需要对 create_time 列按月进行分表，并且每个月能够对应一张物理表，则应该使用如下的建表 DDL：

```
create table test_mm_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(id)
tbpartition by MM(create_time) tbpartitions 12;
```

注意事项

按 MM 进行分表，由于一年的月份只有 12 个月，所以各分库的分表数不能超过 12 张分表。

DD

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

只能作为分表函数使用，但不能作为分库函数。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值所对应的当月的日期进行取余运算并得到分表下标。

使用场景

DD 适用于按一个月的天数（即日期）进行分表，分表的表名的下标就是一个月中的第某天，一个月最多有31天。

使用示例

假设先按 id 对用户进行分库，再需要对 create_time 列按日进行分表，并要求每天都对应一张物理表，则应该使用如下的建表 DDL：

```
create table test_dd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(id)
tbpartition by DD(create_time) tbpartitions 31;
```

注意事项

由于一个月最多是 31 天，当按 DD 进行分表时，所以各个分库的分表数目不能超过 31 张。

WEEK

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

只能作为分表函数使用，但不能作为分库函数。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值所对应的的一周之中的日期进行取余运算并得到分表下标。

使用场景

WEEK 适用于按周数的日期目进行分表，分表的表名的下标分别对应一周中的各个日期（星期一到星期天）。

使用示例

假设先按 id 对用户进行分库，再需要对 create_time 列按周进行分表，并且每周 7 天（星期一到星期天）各对应一张物理表，则应该使用如下的建表 DDL：

```
create table test_week_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(name)
tbpartition by WEEK(create_time) tbpartitions 7;
```

注意事项

由于一周共有 7 天，当按 WEEK 进行分表时，所以各分库的分表数不能超过 7 张。

MMDD

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

只能作为分表函数使用，但不能作为分库函数。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值所对应的日期在一年中对应的天数，然后进行取余运算并得到分表下标。

使用场景

MMDD 适用于按一年的天数（即一年中日期）进行分表，分表的表名的下标就是一年中的第某天，一年最多 366 天。

使用示例

假设先按 id 对用户进行分库，然后需要对 create_time 列按日期（时间的月份与日期）进行建表，并且一年中每一天的日期都能对应一张物理表，则应该使用如下的建表 DDL：

```
create table test_mmdd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by HASH(name)
tbpartition by MMDD(create_time) tbpartitions 365;
```

注意事项

由于一年最多只有 366 天，当按 MMDD 进行分表时，所以各个分库的分表数目不能超过 366 张分表。

YYYYMM

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据拆分键的时间值的年份与月份进行计算哈希值，然后再按分库数去取余，完成路由计算。

例如：YYYYMM('2012-12-31 12:12:12') 等价于 (2012 * 12 + 12) % D, (D 是分库数目)。

使用场景

适合于需要按年份与月份进行分库的场景，建议该函数会与 tbpartition YYYYMM(ShardKey) 联合使用。

例如，假设用户的 DRDS 里已经分了 8 个物理库，现业务有如下的场景：

1. 一个业务想按年月进行分库；
2. 要求是同一个月的数据能落在同一张分表，并且两年以内的每个月都单独对应一张分表；
3. 查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

那么， 用户这时就可以使用 YYYYMM 的分库函数进行解决：业务要求两年以内的每个月都对应一张分表（就是一个月一张表），由于一年有 12 个月，所以至少需要创建 24 个物理分表才能满足用户的场景，而用户的 DRDS 有 8 个分库，所以每个分库应该建 3 张物理分表。因此，与用户业务场景应该对应的 DDL 应该是：

```
create table test_yyyymm_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYMM(create_time)
tbpartition by YYYYMM(create_time) tbpartitions 3;
```

注意事项

YYYYMM 不支持对于每一个年月都独立对应一张分表，YYYYMM 的分库分表必须固定分表数目。

当月份经过一个轮回（如 2013-03 是 2012-03 的一个轮回）后，同一个月份就有可能被路由到同一个分库分表，视实际的分表数目而定。

YYYYWEEK

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值的年份与一年的周数进行计算哈希值，然后再按分库数去取余，完成路由计算。

例如：YYYYWEEK('2012-12-31 12:12:12') 等价于 （计算出"2012-12-31"是 2013 年第 1 周，即 2013*52 + 1 ） % D，(D 是分库数目)。

使用场景

适合于需要按年份与一年的周数进行分库的场景，建议该函数会与 tbpartition YYYYWEEK(ShardKey) 同联合使用。

例如，假设用户的 DRDS 里已经分了 8 个物理库, 现业务有如下的场景：

一个业务想按年周进行分库；

要求是同一周的数据都能落在同一张分表，并且两年以内的每个周都单独对应一张分表；

查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

那么， 用户这时就可以使用 YYYYWEEK 的分库函数进行解决：业务要求两年以内的每个周都对应一张分表（就是一个周一张表），由于一年有近 53 个周(四舍五入)，所以两年至少需要创建 106 个物理分表才能满足用户的场景，而用户的 DRDS 有 8 个分库，所以每个分库应该建 14 张物理分表（14 * 8 = 112 > 106， 分表数最好是分库数的整数倍）。因此，与用户业务场景应该对应的 DDL 应该是：

```
create table test_yyyymm_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYWEEK(create_time)
```

```
tbpartition by YYYYWEEK(create_time) tbpartitions 14;
```

注意事项

YYYYWEEK 不支持对于每一个年周都独立对应一张分表，YYYYWEEK 的分库分表必须固定分表数目。

当周数经过一个轮回（如 2013 第 1 周是 2012 第一周的一个轮回）后，相同周数有可能被路由到同一个分库分表，视实际的分表数目而定。

YYYYDD

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

路由方式

根据分库键的时间值的年份与一年的天数进行计算哈希值，然后再按分库数去取余，完成路由计算。

例如：YYYYDD('2012-12-31 12:12:12') 等价于 （计算出"2012-12-31"是 2012 年第 365 天，即 2012*366 + 365 ） % D。D 是分库数目。

使用场景

适用于需要按年份与一年的天数进行分库的场景。建议该函数与 tbpartition YYYYDD(ShardKey) 联合使用。

例如，假设用户的 DRDS 里已经分了 8 个物理库，现业务有如下的场景：

一个业务想按年天进行分库；

要求是同一周的数据都能落在同一张分表，并且两年以内的每个天都能单独对应一张分表；

3. 查询时带上分库分表键后能直接将查询落在某个物理分库的某个物理分表。

这时就可以使用 YYYYDD 的分库函数进行解决。业务要求两年以内的每天都对应一张分表（就是一天一张表），由于一年有最多 366 天，所以两年至少需要创建 732 个物理分表才能满足用户的场景。用户的 DRDS 有 8 个分库，所以每个分库应该建 92 张物理分表（732 / 8 = 91.5，取整为 92， 分表数最好是分库数的整数倍）。因此，与用户业务场景应该对应的 DDL 应该是：

```
create table test_yyyydd_tb (
    id int,
    name varchar(30) DEFAULT NULL,
    create_time datetime DEFAULT NULL,
    primary key(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
dbpartition by YYYYDD(create_time)
tbpartition by YYYYDD(create_time) tbpartitions 92;
```

注意事项

YYYYDD 不支持对于每一个年天都独立对应一张分表，YYYYDD 的分库分表必须固定分表数目。

当日期经过一个轮回（如 2013-03-01 是 2012-03-01 的一个轮回）后，同一个日期有可能被路由到同一个分库分表，视实际的分表数目而定。

YYYYMM_OPT

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

用户数据的时间的年份与月份是随时间自然增长的，而不是随机。

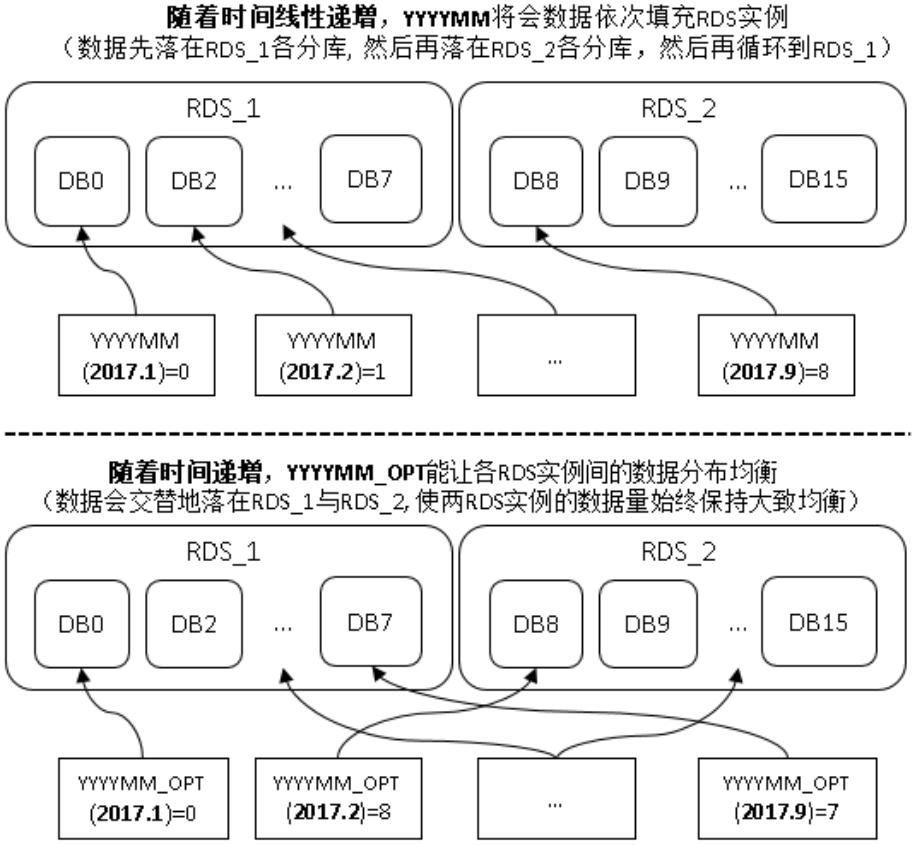
DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

优化点

相对于 YYYYMM，YYYYMM_OPT **随着时间线递增能够保持数据在各个 RDS 实例之间的均衡分布。**

例如，假设用户在 DRDS 实例上挂了 2 个 RDS 实例，共 16 个分库，通常 DB0~DB7 的分库在一个 RDS 实例上，DB8~DB15 的分布在另外的 RDS 实例上。

那么，下图就是分别使用 YYYYMM 与 YYYYMM_OPT 进行分库时，它们的映射结果（只分库不分表）的对比：



YYYYMM_OPT 的各个 RDS 实例之间的数据均衡性，有助于充分利用各个 RDS 实例的性能。

关于 YYYYMM 与 YYYYMM_OPT 的选择：

如果业务数据的时间按顺序逐渐增大的，并且各个时间点的数据量相差不大，那么适合用 YYYYMM_OPT 实现 RDS 实例之间的数据均衡；

如果业务数据的时间会比较跳跃，数据的时间点出现比较随机，那么适合用 YYYYMM。

路由方式

根据分库键的时间值的年份与月份进行计算哈希值，然后再将哈希值按分库数去取余，完成路由计算。

分库分表键的哈希计算过程会根据 DRDS 之下的 RDS 实例的数目适当考虑 RDS 实例间数据的均衡性。

使用场景

业务需要按年月进行分库分表。

您希望 DRDS 的各个 RDS 实例的数据量保持相对均衡。

拆分键的时间呈顺序递增（即不是随机的）并且各个月份的数据量相对平均的前提下（例如，每个月的流水日志，它随着月份不断增大，数据不会集中在同一个 RDS 上）。

注意事项

YYYYMM_OPT 不支持对于每一个年月都独立对应一张分表，YYYYMM_OPT 的分库分表必须固定分表数目。

当月份经过一个轮回（如 2013-03 是 2012-03 的一个轮回）后，同一个月份就有可能被路由到同一个分库分表，视实际的分表数目而定。

YYYYWEEK_OPT

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

优化点

相对于 YYYYWEEK，YYYYWEEK_OPT **随着时间线递增能够保持数据在各个 RDS 实例之间的均衡分布**，效果与 YYYYMM_OPT 类似。

YYYYWEEK_OPT 的各个 RDS 实例之间的数据均衡性，有助于充分利用各个 RDS 实例的性能。

关于 YYYYWEEK 与 YYYYWEEK_OPT 的选择：

如果业务数据的时间按顺序逐渐增大的，并且各个时间点的数据量相差不大，那么适合用 YYYYWEEK_OPT 实现 RDS 实例之间的数据均衡；

如果业务数据的时间会比较跳跃，数据的时间点出现比较随机，那么适合用 YYYYWEEK。

路由方式

根据分库键的时间值的年份与一年的周数进行计算哈希值，然后再将哈希值按分库数去取余，完成路由计算。

分库分表键的哈希计算过程会根据 DRDS 之下的 RDS 实例的数目适当考虑 RDS 数据的均衡性。

使用场景

业务需要按年周进行分库分表。

希望 DRDS 的各个 RDS 实例的数据量保持相对均衡。

拆分键的时间呈顺序递增（即不是随机的）并且各个周的数据量相对平均的前提下（例如，每个周的流水日志，它随着周数不断增大，数据不会集中在同一个 RDS 上）。

注意事项

YYYYWEEK_OPT 不支持对于每一个年周都独立对应一张分表，YYYYWEEK_OPT 的分库分表必须固定分表数目

当周数经过一个轮回（如 2013 第 1 周是 2012 第一周的一个轮回）后，相同周数有可能被路由到同一个分库分表，视实际的分表数目而定。

YYYYDD_OPT

使用要求

拆分键的类型必须是 DATE / DATETIME / TIMESTAMP 其中之一。

DRDS 实例的版本必须是 5.1.28-1320920 及其以上的版本。DRDS 版本说明请参考文档版本说明。

优化点

相对于 YYYYDD，YYYYDD_OPT **随着时间线递增能够保持数据在各个 RDS 实例之间的均衡分布**，效果与 YYYYMM_OPT 类似。

YYYYDD_OPT 的各个 RDS 实例之间的数据均衡性，有助于充分利用各个 RDS 实例的性能。

YYYYDD 与 YYYYDD_OPT 该如何选择：

如果业务数据的时间按顺序逐渐增大的，并且各个时间点的数据量相差不大，那么适合用 YYYYDD_OPT 实现 RDS 实例之间的数据均衡；

如果业务数据的时间会比较跳跃，数据的时间点出现比较随机，那么适合用 YYYYDD。

路由方式

根据分库键的时间值的年份与一年的天数进行计算哈希值，然后再将哈希值按分库数去取余，完成路由计算。

分库分表键的哈希计算过程会根据 DRDS 之下的 RDS 实例的数目适当考虑 RDS 数据的均衡性。

使用场景

业务需要按年日进行分库分表。

希望 DRDS 的各个 RDS 实例的数据量保持相对均衡。

拆分键的时间呈顺序递增（即不是随机的）并且每天的数据量相对平均的前提下(例如，每天的流水日志，它随着日的期不断增大，数据不会集中在同一个 RDS 上)。

注意事项

YYYYDD_OPT 不支持对于每一个年天都独立对应一张分表，YYYYDD_OPT 的分库分表必须固定分表数目。

当日期经过一个轮回（如 2013-03-01 是 2012-03-01 的一个轮回）后，同一个日期有可能被路由到同一个分库分表，视实际的分表数目而定。

访问控制

主子账号介绍

本节主要介绍阿里云的访问控制服务（RAM）的基本概念以及 RAM 在 DRDS 中的应用场景。

RAM 介绍

RAM 是阿里云提供的用户身份管理与访问控制服务。使用 RAM 可以创建、管理用户账号（比如员工、系统或应用程序），并控制这些用户账号对云账户名下资源的操作权限。当企业或者组织存在多用户协同操作资源时，RAM 可以避免云账号秘钥大范围传播，按需分配最小权限，从而降低企业信息安全管理风险。详细信息请参考 RAM 产品文档。

RAM 基本概念

云账户

云账户是阿里云资源归属、资源使用计量计费的基本主体。使用阿里云服务，首先需要注册一个云账户。云账户为其名下所拥有的资源付费，并对其名下所有资源拥有完全权限。

RAM 用户

在一个云账户下创建的 RAM 用户(可以对应企业内的员工、系统或应用程序)。RAM 用户不拥有资源，没有独立的计量计费，这些用户由所属云账户统一控制和付费。RAM 用户只有在获得云账户的授权后才能登录控制台或使用 API 操作云账户下的资源。

身份凭证(Credential)

用于证明用户真实身份的凭据，通常是指登录密码或访问密钥(AccessKey)。

- 登录名/密码(Password)：用于登陆阿里云控制台进行资源操作，以便查看订单、账单或购买资源。
- 访问密钥(AccessKey)：用于构造一个 API 请求(或者使用云服务 SDK)并操作资源。

角色(Role)

是一种虚拟身份或影子账号，表示某种操作权限的虚拟概念，有独立的身份 ID，但是没有独立的登录密码和 AccessKey。子账号可以扮演角色，扮演角色时拥有该角色自身的权限。

资源(Resource)

代表用户可访问的云资源，比如该用户所拥有的 DRDS 实例，或者某个 DRDS 实例下面的某个数据库等。

权限(Permission)

允许或拒绝一个用户对某种资源执行资源管控或者使用类操作。

授权策略(Policy)

用于定义权限规则，比如允许用户读取或者写入某些资源。

角色扮演

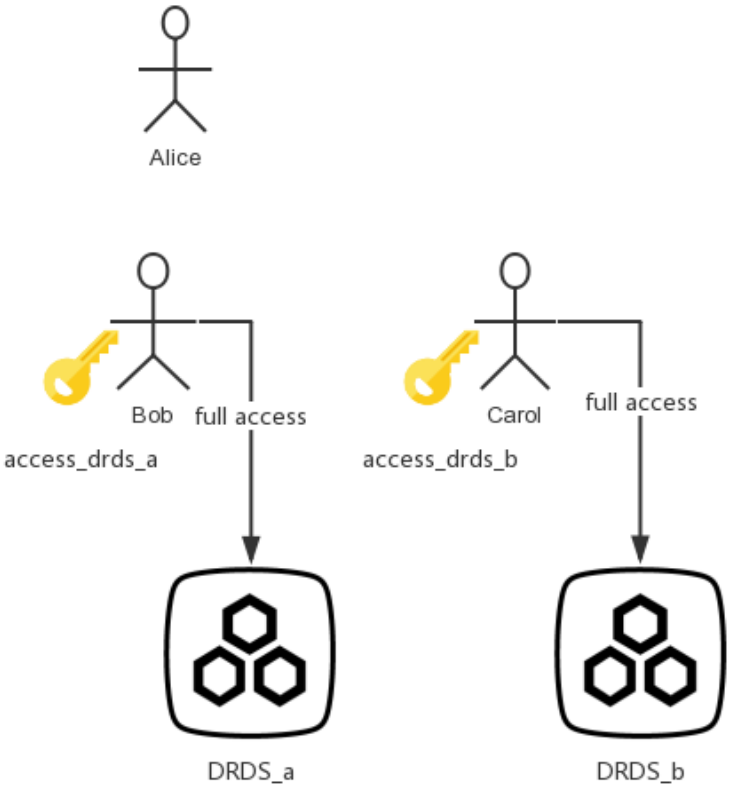
子账号和角色可以类比为某个个人和其身份的关系。某人在公司的角色是员工，在家里的角色是父亲，在不同的场景扮演不同的角色，但是还是同一个人。在扮演不同的角色的时候也就拥有对应角色的权限。单独的员工或者父亲概念并不能作为一个操作的实体，只有有人扮演了之后才是一个完整的概念。同一个角色也可以被多个不同的个人同时扮演。完成角色扮演之后，就自动拥有该角色的所有权限。

DRDS 的 RAM 应用场景示例

假设某阿里云用户 Alice 拥有两个 DRDS 个实例，DRDS_a 和 DRDS_b。 Alice 对这两个实例都拥有完全的权限。

1. 为了避免阿里云账号的 AccessKey 泄露导致安全风险，Alice 使用 RAM 创建了两个子账号 Bob 和 Carol。
2. Alice 建立了两个授权策略，access_drds_a 和 access_drds_b，分别表示 DRDS_a 和 DRDS_b 的读写权限。
3. Alice 在控制台分别对 Bob 和 Carol 进行授权，使 Bob 对 DRDS_a 拥有读写权限，Carol 对 DRDS_b 拥有读写权限。

Bob 和 Carol 都拥有独立的 AccessKey，这样万一泄露也只会影响其中一个 DRDS 实例，而且 Alice 可以很方便的在控制台取消泄露用户的权限。
DRDS RAM 应用场景示意图



使用 RAM 的准备工作

DRDS 的部分操作会调用 RDS 的 Open API，因此在使用 RAM 之前，需要先激活 DRDS 访问 RDS 服务的授权，创建一个供 DRDS 访问 RDS 的服务角色。

您可以通过控制台操作或者调用 RAM Open API 的方式来完成授权激活。

在控制台上激活 RAM 授权

在 DRDS 控制台左侧菜单选择**资源授权**，单击页面上的**激活授权**按钮。

激活授权的同时， DRDS 会在 RAM 控制台上创建一个供 DRDS 使用的 RAM 角色（role），用于访问该账户下的 RDS，同时授权 DRDS 访问 RDS 的 Open API。**请不要删除这个 RAM 角色。**

通过 RAM 的 Open API 激活 RAM 授权

如果在某些特定环境下无法访问 DRDS 控制台，也可以通过调用 RAM Open API 创建 RAM 服务角色。角色创建完成则 DRDS 访问 RDS 的授权也被激活。

创建 RAM 角色

CreateRole 接口可以创建 RAM 角色，具体用法请参考 RAM 文档。假设要创建的 RAM 角色名是 Jack，该接口的参数如下：

参数名	参数值
Action	CreateRole
RoleName	Jack
AssumeRolePolicyDocument	格式见下文

AssumeRolePolicyDocument 的格式：

```
{
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "drds.aliyuncs.com"
        ]
      }
    }
  ],
  "Version": "1"
```

```
}
```

如果使用 RAM Java SDK，Demo 如下：

```
String rolePolicyDoc = "{\"Statement\":[{\"Action\": \"sts:AssumeRole\", \"Effect\":
\"Allow\",\"Principal\":{\"Service\":[\"drds.aliyuncs.com\"]}}],\"Version\": \"1\"}";
String roleName = "Jack";
CreateRoleRequest request = new CreateRoleRequest();
request.setAssumeRolePolicyDocument(rolePolicyDoc);
request.setRoleName(roleName);
client.getAcsResponse(request);
// 给角色授予策略
AttachPolicyToRoleRequest attachRequest = new AttachPolicyToRoleRequest();
attachRequest.setPolicyType("System");
attachRequest.setPolicyName("AliyunDRDSRolePolicy");
attachRequest.setRoleName("AliyunDRDSDefaultRole");
client.getAcsResponse(attachRequest);
```

查看角色

如果想验证上一步创建的角色是否成功，可以调用GetRole 接口查看角色，使用方式可以参见 RAM 接口文档。假设要获取刚刚创建的 RAM 角色 Jack，参数是：

参数名	参数值
Action	GetRole
RoleName	Jack

如果使用 RAM Java SDK，Demo 如下：

```
String roleName = "AliyunDRDSDefaultRole";
GetRoleRequest request = new GetRoleRequest();
request.setRoleName(roleName);
GetRoleResponse resp = client.getAcsResponse(request);
GetRoleResponse.Role role = resp.getRole();
```

DRDS 控制台使用 RAM

本文介绍如何在 DRDS 中使用 RAM 的账号体系及权限策略进行资源和权限控制。

目前，DRDS 控制台使用 RAM 有如下限制:

- RAM 子帐户删除数据库与删除只读账户需要开启 MFA 多因素认证（具体请参见下文）；
- RAM 子帐户没有修改 DRDS 数据库密码的权限。

DRDS 控制台使用 RAM

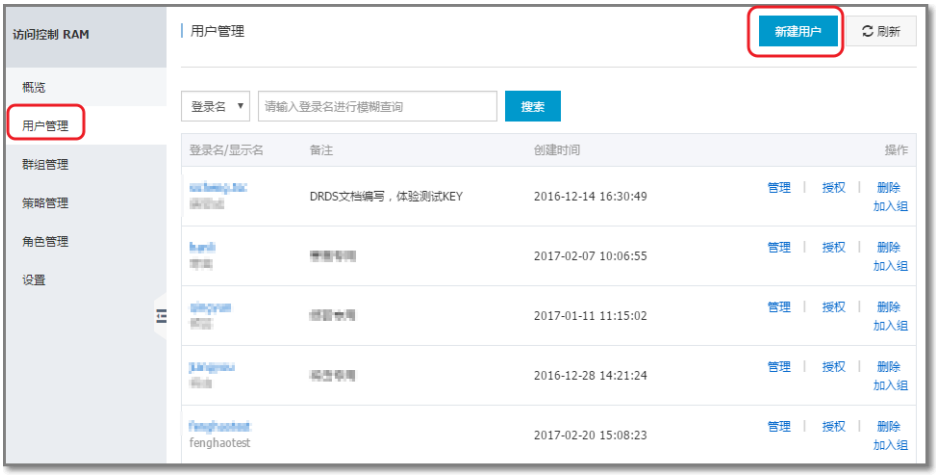
在 DRDS 控制台使用 RAM 需要在 RAM 控制台进行以下操作：

- 创建 RAM 子账户；
- 创建授权策略；
- 为 RAM 子账户授权。

注意： 在 DRDS 使用 RAM 账号系统前，请确保已经激活 DRDS 对 RDS 的访问授权，创建了供 DRDS 使用的 RAM 角色（role），具体请参考使用 RAM 的准备工作文档。

创建 RAM 子账户

登录 RAM 控制台，根据控制台引导创建 RAM 子帐户。



RAM 子帐户创建成功后，就可以为子帐户授予相应的资源权限。RAM 中的权限由策略来实现，所以需要先创建(修改)策略。

创建策略

在 RAM 控制台左侧菜单栏选择**策略管理**，单击页面右上角的**新建授权策略**，根据引导完成策略创建。

说明：

- RAM 控制台提供了AliyunDRDSReadOnlyAccess（表示只读操作的权限合集）和AliyunDRDSFullAccess（表示所有操作的权限合集）两个策略，在授权时可以选择授予子帐户这两个策略。
- 如果需要更加灵活的授权策略，也可以选择空白模板，自定义具体的 DRDS 授权策略。在 RAM 服务中，使用授权策略语言来描述一个授权策略，具体的语法请参考 Policy 语法结构。 目前 DRDS 支持的授权策略参见 DRDS 支持的资源授权。

自定义策略示例

赋予某个子帐户对应的主账户所拥有的 DRDS 控制台操作权限：

```
{
  "Version": "1",
  "Statement": [
   {
      "Action": "drds:*",
      "Resource": "*",
      "Effect": "Allow"
   },
   {
      "Action": "ram:PassRole",
      "Resource": "*",
      "Effect": "Allow"
   }
  ]
}
```

指定用户只可以访问杭州可用区下所有 DRDS 的权限：

```
{
"Version": "1",
"Statement": [
 // 1234指的是付费账户的 uid，登陆阿里云主菜单在账号管理-安全设置菜单可见
 {
    "Action": "drds:*",
    "Resource": "acs:drds:cn-hangzhou:1234:instance/*",
    "Effect": "Allow"
 },
 //注意，需要确保策略中存在下面这一段，以保证 RAM 功能的正常使用。
 {
    "Action": "ram:PassRole",
    "Resource": "*",
    "Effect": "Allow"
 }
]
}
```

指定用户无法访问特定的某个实例。被授予该策略的 RAM 子帐户可以访问除drds******hb4之外的所有 DRDS 实例。：

```
{
"Version": "1",
"Statement": [
 {
    "Action": "drds:*",
    "Resource": "acs:drds:*:1234:instance/*",
    "Effect": "Allow"
 },
 {
    "Action": "drds:DescribeDrdsInstance",
    "Resource": "acs:drds:*:1234:instance/drds******hb4",
    "Effect": "Deny"
 },
 //注意，需要确保策略中存在下面这一段，以保证 RAM 功能的正常使用。
 {
    "Action": "ram:PassRole",
    "Resource": "*",
    "Effect": "Allow"
 }
]
}
```

为子账户授权

创建好策略后，就可以对某个子账户进行授权，赋予该子账户特定的 DRDS 权限。

1. 在 RAM 控制台左侧菜单栏选择**用户管理**。
2. 在需要操作的子账户对应一行，单击右侧的**授权**按钮。
3. 选择相应的策略进行授权。

完成以上步骤后，就可以用 RAM 子账户登录 DRDS 并进行相关操作了。

主子账号操作区别

在删除数据库或者数据库只读账户时，主子账户存在以下区别：

使用主账户删除：需要验证主账户的手机号码，即主账户输入 DRDS 控制台发送的短信验证码以后才能删除。

使用子账户删除：不需要与手机号码绑定，只需要在 RAM 控制台为子帐户开启多因素认证。

开启多因素认证操作如下：

1. 在 RAM 控制台左侧菜单栏选择**用户管理**。

2. 单击需要操作的子账户名称进入**用户详情**页。
3. 在**多因素认证设备**一栏下，单击**启用虚拟 MFA 设备**。

开启成功后，子账户登陆阿里云控制台都需要使用智能设备的应用“身份宝”进行认证。身份宝安装请参考**身份宝**。

DRDS 支持的资源授权

DRDS 目前支持可以授权的资源如下。具体语法请参考**阿里云文档**。用$开头的参数为用户自己填写的参数。

Action	鉴权规则	描述
CreateDrdsInstance	acs:drds:$regionid:$accountid:instance/*	创建实例
DescribeDrdsInstanceList	acs:drds:$*:$accountid:instance/*	查看实例列表（此处 “regionid” 需设置为 “*” 号 ）
UpgradeDrdsInstance	acs:drds:$regionid:$accountid:instance/$instanceid	实例变配
RemoveDRDSInstance	acs:drds:$regionid:$accountid:instance/$instanceid	释放实例
DescribeDrdsInstance	acs:drds:$regionid:$accountid:instance/$instanceid	查看实例详情
VersionChanage	acs:drds:$regionid:$accountid:instance/$instanceid	升级回滚 DRDS 实例版本
CreateInternetAddress	acs:drds:$regionid:$accountid:instance/$instanceid	创建 DRDS 实例的公网地址
ReleaseInternetAddress	acs:drds:$regionid:$accountid:instance/$instanceid	释放 DRDS 实例的公网地址
CloneInfo	acs:drds:$regionid:$accountid:instance/$instanceid	克隆实例
CloneTaskList	acs:drds:$regionid:$accountid:instance/*	查看克隆列表
CreateDrdsDB	acs:drds:$regionid:$accountid:instance/$instanceid/db/*	创建 DRDS 数据库
DescribeDrdsDbList	acs:drds:$regionid:$accountid:instance/$instanceid/db/*	查看 DRDS 实例的数据库列表
DescribeDrdsDb	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	查看 DRDS 数据库详情
DeleteDrdsDb	acs:drds:$regionid:$accounti	删除 DRDS 数据库

	d:instance/$instanceid/db/$dbname	
ModifyReadWriteWeight	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	修改读策略
DescribeLogicTableList	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	查看 DRDS 数据库中数据表列表
ExecuteDDL	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	在 DRDS 控制台执行 DDL 语句
ModifyDrdsIpWhiteList	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	修改 DRDS 数据库的 IP 白名单
DrdsDataImport	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	数据导入
DrdsSmoothExpand	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	平滑扩容
CreateReadOnlyAccount	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	创建只读账户
ModifyReadOnlyAccountPassword	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	修改只读账户密码
RemoveReadOnlyAccount	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	删除只读账户
DescribeAlarmContacts	acs:drds:$regionid:$accountid:contacts/*	查看报警联系人列表
AddAlarmContacts	acs:drds:$regionid:$accountid:contacts/*	添加报警联系人
ModifyAlarmContacts	acs:drds:$regionid:$accountid:contacts/*	修改报警联系人
RemoveAlarmContacts	acs:drds:$regionid:$accountid:contacts/*	删除报警联系人
DescribeAlarmGroup	acs:drds:$regionid:$accountid:contacts/*	查看报警联系人分组列表
AddAlarmGroup	acs:drds:$regionid:$accountid:contacts/*	添加报警联系人分组
ModifyAlarmGroup	acs:drds:$regionid:$accountid:contacts/*	修改报警联系人分组
RemoveAlarmGroup	acs:drds:$regionid:$accountid:contacts/*	删除报警联系人分组

DescribeInstanceMonitor	acs:drds:$regionid:$accountid:instance/$instanceid	查看实例监控信息
DescribeAlarmRule	acs:drds:$regionid:$accountid:instance/$instanceid	查看报警规则列表
CreateAlarmRule	acs:drds:$regionid:$accountid:instance/$instanceid	创建报警规则
ModifyAlarmRule	acs:drds:$regionid:$accountid:instance/$instanceid	修改报警规则
RemoveAlarmRule	acs:drds:$regionid:$accountid:instance/$instanceid	删除报警规则
DescribeAlarmHistory	acs:drds:$regionid:$accountid:instance/$instanceid	查看报警历史
DescribeSlowSql	acs:drds:$regionid:$accountid:instance/$instanceid	查看 DRDS 慢 SQL
DrdsShardTool	acs:drds:$regionid:$accountid:instance/$instanceid/db/$dbname	使用拆分变更工具

注意：

1. 创建报警规则权限依赖查询联系人分组权限。
2. 修改报警规则权限依赖查询联系人分组权限。

DRDS 支持 RAM 的区域

目前 DRDS 已经开通 RAM 服务的地域以及对应的 regionId 如下表所示。在配置 RAM 策略时，$regionId需要被替换成表中的 regionId。

regionId	区域名
cn-hangzhou	华东1
cn-shenzhen	华南1
cn-shanghai	华东2
cn-qingdao	华北1
cn-beijing	华北2

DRDS 自定义控制指令

自定义控制指令简介

DRDS 提供了一系列辅助 SQL 指令帮助用户方便使用 DRDS。

主要包括以下几大类：

- 帮助语句
- 查看规则和节点拓扑类语句
- SQL 调优类语句
- 统计信息查询类语句
- SHOW PROCESSLIST 指令和 KILL 指令
- DRDS SEQUENCE 相关语句

帮助语句

SHOW HELP 语句

展示 DRDS 所有辅助 SQL 指令及其说明。

```
mysql> show help;
+---------------------------------------+-----------------------------------------------------+------------------------
-------------------+
| STATEMENT | DESCRIPTION | EXAMPLE |
+---------------------------------------+-----------------------------------------------------+------------------------
-------------------+
| show rule | Report all table rule | |
| show rule from TABLE | Report table rule | show rule from user |
| show full rule from TABLE | Report table full rule | show full rule from user |
| show topology from TABLE | Report table physical topology | show topology from user |
| show partitions from TABLE | Report table dbPartition or tbPartition columns | show partitions from user |
| show broadcasts | Report all broadcast tables | |
| show datasources | Report all partition db threadPool info | |
| show node | Report master/slave read status | |
| show slow | Report top 100 slow sql | |
| show physical_slow | Report top 100 physical slow sql | |
| clear slow | Clear slow data | |
| trace SQL | Start trace sql, use show trace to print profiling data | trace select count(*) from user; show trace |
| show trace | Report sql execute profiling info | |
| explain SQL | Report sql plan info | explain select count(*) from user |
| explain detail SQL | Report sql detail plan info | explain detail select count(*) from user |
| explain execute SQL | Report sql on physical db plan info | explain execute select count(*) from user |
| show sequences | Report all sequences status | |
| create sequence NAME [start with COUNT] | Create sequence | create sequence test start with 0 |
```

```
| alter sequence NAME [start with COUNT] | Alter sequence | alter sequence test start with 100000 |
| drop sequence NAME | Drop sequence | drop sequence test |
+---------------------------------------+-----------------------------------------------------+-------------------------
--------------------+
20 rows in set (0.00 sec)
```

查看规则和节点拓扑类语句

查看规则和节点拓扑类语句如下：

- SHOW RULE [FROM tablename] 语句
- SHOW FULL RULE [FROM tablename] 语句
- SHOW TOPOLOGY FROM tablename 语句
- SHOW PARTITIONS FROM tablename 语句
- SHOW BROADCASTS 语句
- SHOW DATASOURCES 语句
- SHOW NODE 语句

1. SHOW RULE [FROM tablename]

使用说明：

- show rule：查看数据库下每一个逻辑表的拆分情况；
- show rule from tablename：查看数据库下指定逻辑表的拆分情况。

重要列详解：

- **BROADCAST**：是否为广播表（0：否，1：是）；
- **DB_PARTITION_KEY**：分库的拆分键，没有分库的话，值为空；
- **DB_PARTITION_POLICY**：分库的拆分策略，取值包括哈希或 YYYYMM、YYYYDD、YYYYWEEK 等日期策略；
- **DB_PARTITION_COUNT**：分库数；
- **TB_PARTITION_KEY**：分表的拆分键，没有分表的话，值为空；
- **TB_PARTITION_POLICY**：分表的拆分策略，取值包括哈希或 MM、DD、MMDD、WEEK 等日期策略；
- **TB_PARTITION_COUNT**：分表数。

```
mysql> show rule;
+------+-------------+----------+----------------+------------------+------------------+----------------+------
---------------+------------------+
| ID   | TABLE_NAME   | BROADCAST | DB_PARTITION_KEY | DB_PARTITION_POLICY | DB_PARTITION_COUNT |
TB_PARTITION_KEY | TB_PARTITION_POLICY | TB_PARTITION_COUNT |
+------+-------------+----------+----------------+------------------+------------------+----------------+------
```

```
--------------+--------------------+
|   0 | dept_manager |        0 |              | NULL             | 1              |              | NULL             | 1              |
|   1 | emp          |        0 | emp_no       | hash             | 8              | id           | hash             | 2              |
|   2 | example      |        0 | shard_key    | hash             | 8              |              | NULL             | 1              |
+------+--------------+----------+-----------------+--------------------+--------------------+-----------------+------
--------------+--------------------+
3 rows in set (0.01 sec)
```

2. SHOW FULL RULE [FROM tablename]

查看数据库下逻辑表的拆分规则，比 SHOW RULE 指令展示的信息更加详细。

重要列详解：

- **BROADCAST**：是否为广播表（0：否，1：是）；
- **JOIN_GROUP**：保留字段，暂时无意义。
- **ALLOW_FULL_TABLE_SCAN**：分库分表在没有指定分表键值的情况下是否允许查询数据，如果配置为 true，此时需要扫描每一个物理表来查找出符合条件的数据，简称为全表扫描；
- **DB_NAME_PATTERN**：DB_NAME_PATTERN 中 {} 之间的 0 为占位符，执行 SQL 时会被 DB_RULES_STR 计算出的值替代，并保持位数。比如，DB_NAME_PATTERN 的值为 SEQ_{0000}_RDS，DB_RULES_STR 的值为 [1,2,3,4]，则会产生4个 DB_NAME，分别为 SEQ_0001_RDS、SEQ_0002_RDS、SEQ_0003_RDS、SEQ_0004_RDS；
- **DB_RULES_STR**：具体的分库规则；
- **TB_NAME_PATTERN**：TB_NAME_PATTERN 中 {} 之间的 0 为占位符，执行 SQL 时会被 TB_RULES_STR 计算出的值替代，并保持位数。比如，TB_NAME_PATTERN 的值为 table_{00}，TB_RULES_STR 的值为 [1,2,3,4,5,6,7,8]，则会产生8张表，分别为 table_01、table_02、table_03、table_04、table_05、table_06、table_07、table_08；
- **TB_RULES_STR**：分表规则；
- **PARTITION_KEYS**：分库和分表键集合，对于既分库又分表的情形，分库键在前，分表键在后；
- **DEFAULT_DB_INDEX**：单库单表存放的分库。

```
mysql> show full rule;
+------+--------------+-----------+------------+----------------------+-----------------------------------------------
+-------------------------------------------------+----------------+-----------------------------------+--------------
-+-------------------------------------------------+
| ID   | TABLE_NAME   | BROADCAST | JOIN_GROUP | ALLOW_FULL_TABLE_SCAN | DB_NAME_PATTERN
| DB_RULES_STR                                    | TB_NAME_PATTERN | TB_RULES_STR                      | PARTITION_KEYS |
DEFAULT_DB_INDEX                                |
+------+--------------+-----------+------------+----------------------+-----------------------------------------------
+-------------------------------------------------+----------------+-----------------------------------+--------------
-+-------------------------------------------------+
|   0 | dept_manager |        0 | NULL       |                    0 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS   | NULL                                              | dept_manager
| NULL                            | NULL           | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |
|   1 | emp          |        0 | NULL       |                    1 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_{0000}_RDS |
((#emp_no,1,8#).longValue().abs() % 8)              | emp_{0}         | ((#id,1,2#).longValue().abs() % 2) | emp_no      id |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |
|   2 | example      |        0 | NULL       |                    1 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_{0000}_RDS
| ((#shard_key,1,8#).longValue().abs() % 8).intdiv(1) | example         | NULL                              | shard_key      |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |
```

```
+------+-------------+----------+-----------+---------------------+-------------------------------------------------
+-----------------------------------------------------+----------------+-----------------------------------+--------------
-+-----------------------------------------------+
3 rows in set (0.01 sec)
```

3. SHOW TOPOLOGY FROM tablename

查看指定逻辑表的拓扑分布，展示该逻辑表保存在哪些分库中，每个分库下包含哪些分表。

```
mysql> show topology from emp;
+------+-----------------------------------------------+------------+
| ID   | GROUP_NAME                                    | TABLE_NAME |
+------+-----------------------------------------------+------------+
|   0 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS | emp_0     |
|   1 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS | emp_1     |
|   2 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS | emp_0     |
|   3 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS | emp_1     |
|   4 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS | emp_0     |
|   5 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS | emp_1     |
|   6 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS | emp_0     |
|   7 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS | emp_1     |
|   8 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS | emp_0     |
|   9 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS | emp_1     |
|  10 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS | emp_0     |
|  11 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS | emp_1     |
|  12 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS | emp_0     |
|  13 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS | emp_1     |
|  14 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS | emp_0     |
|  15 | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS | emp_1     |
+------+-----------------------------------------------+------------+
16 rows in set (0.01 sec)
```

4. SHOW PARTITIONS FROM tablename

查看分库分表键集合，分库键和分表键之间用逗号分割。如果最终结果有两个值，说明是既分库又分表的情形，第一个是分库键，第二个是分表键。如果结果只有一个值，说明是分库不分表的情形，该值是分库键。

```
mysql> show partitions from emp;
+-----------+
| KEYS      |
+-----------+
| emp_no,id |
+-----------+
1 row in set (0.00 sec)
```

5. SHOW BROADCASTS

查看广播表列表。

```
mysql> show broadcasts;
+------+------------+
| ID   | TABLE_NAME |
```

```
+------+------------+
|    0 | brd2       |
|    1 | brd_tbl    |
+------+------------+
2 rows in set (0.01 sec)
```

6. SHOW DATASOURCES

查看底层存储信息，包含数据库名、数据库分组名、连接信息、用户名、底层存储类型、读写权重、连接池信息等。

重要列详解：

- **SCHEMA**：数据库名；
- **GROUP**：数据库分组名。分组的目标是管理多组数据完全相同的数据库，比如通过 RDS（MySQL）进行数据复制后的主备数据库。主要用来解决读写分离、主备切换的问题；
- **URL**：底层 RDS（MySQL）的连接信息；
- **TYPE**：底层存储类型，目前只支持 RDS（MySQL)；
- **READ_WEIGHT**：读权重。在主实例的读压力比较大的时候，可以通过 DRDS 读写分离功能将读流量进行分流，减轻 RDS 主实例的压力。DRDS 会自动识别读写流量，引导写流量进入 RDS 主实例，读流量则按配置的权重流向所有 RDS 实例；
- **WRITE_WEIGHT**：写权重。解释见上。

```
mysql> show datasources;
+------+--------------------------+---------------------------------------------+-------------------------------------
----------+---------------------------------------------------------------------------------+-----------+-------+------+--
----+------+-------------+----------+-------------+--------------+--------------------------------------------+-------
------+-------------+
| ID   | SCHEMA                   | NAME                                        | GROUP                                  | URL
| USER       | TYPE  | INIT | MIN  | MAX  | IDLE_TIMEOUT | MAX_WAIT | ACTIVE_COUNT | POOLING_COUNT | ATOM
| READ_WEIGHT | WRITE_WEIGHT |
+------+--------------------------+---------------------------------------------+-------------------------------------
----------+---------------------------------------------------------------------------------+-----------+-------+------+--
----+------+-------------+----------+-------------+--------------+--------------------------------------------+-------
------+-------------+
|    0 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0000_iiab_1 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0000 | jnkinsea0 | mysql | 0    | 24   |
72   | 15           | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0000_iiab | 10          | 10           |
|    1 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab_2 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0001 | jnkinsea0 | mysql | 0    | 24   |
72   | 15           | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0001_iiab | 10          | 10           |
|    2 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab_3 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0002 | jnkinsea0 | mysql | 0    | 24   |
72   | 15           | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0002_iiab | 10          | 10           |
|    3 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab_4 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0003 | jnkinsea0 | mysql | 0    | 24   |
72   | 15           | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0003_iiab | 10          | 10           |
|    4 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab_5 |
```

```
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0004 | jnkinsea0 | mysql | 0    | 24   |
72   | 15          | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0004_iiab | 10          | 10          |
|    5 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab_6 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0005 | jnkinsea0 | mysql | 0    | 24   |
72   | 15          | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0005_iiab | 10          | 10          |
|    6 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab_7 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0006 | jnkinsea0 | mysql | 0    | 24   |
72   | 15          | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0006_iiab | 10          | 10          |
|    7 | seq_test_1487767780814rgkk | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab_8 |
SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS |
jdbc:mysql://rds1ur80kcv8g3t6p3ol.mysql.rds.aliyuncs.com:3306/seq_test_wnjg_0007 | jnkinsea0 | mysql | 0    | 24   |
72   | 15          | 5000     | 0           | 1            | rds1ur80kcv8g3t6p3ol_seq_test_wnjg_0007_iiab | 10          | 10          |
+------+----------------------------+----------------------------------------------+-----------------------------------
-----------+-------------------------------------------------------------------------------------+-----------+-------+------+--
----+------+-------------+----------+-------------+--------------+-----------------------------------------------+-------
------+--------------+
8 rows in set (0.01 sec)
```

7. SHOW NODE

查看物理库的读写次数（历史累计数据）、读写权重（历史累计数据）。

重要列详解：

- **MASTER_READ_COUNT**：RDS 主实例处理的只读查询次数（历史累计数据）；
- **SLAVE_READ_COUNT**：RDS 备实例处理的只读查询次数（历史累计数据）；
- **MASTER_READ_PERCENT**：RDS 主实例处理的只读查询占比（注意该列显示的是累计的实际数据占比，并不是用户配置的百分比）；
- **SLAVE_READ_PERCENT**：RDS 备实例处理的只读查询占比（注意该列显示的是累计的实际数据占比，并不是用户配置的百分比）。

注意：

- 事务中的只读查询会被发送到 RDS 主实例；
- 由于MASTER_READ_PERCENT，SLAVE_READ_PERCENT这两列代表的是历史累计数据，更改读写权重的配比后，这几个数值并不能立即反应最新的读写权重配比，需累计一段比较长的时间才行。

```
mysql> show node;
+------+----------------------------------------------+------------------+-----------------+---------------------+----
----------------+
| ID   | NAME                                         | MASTER_READ_COUNT | SLAVE_READ_COUNT | MASTER_READ_PERCENT
| SLAVE_READ_PERCENT |
+------+----------------------------------------------+------------------+-----------------+---------------------+----
----------------+
| 0    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0000_RDS |              12 |               0 | 100%                | 0%
|
| 1    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0001_RDS |               0 |               0 | 0%                  | 0%
|
| 2    | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0002_RDS |               0 |               0 | 0%                  | 0%
```

```
|
| 3   | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0003_RDS |           0 |           0 | 0%            | 0%
|
| 4   | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0004_RDS |           0 |           0 | 0%            | 0%
|
| 5   | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0005_RDS |           0 |           0 | 0%            | 0%
|
| 6   | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0006_RDS |           0 |           0 | 0%            | 0%
|
| 7   | SEQ_TEST_1487767780814RGKKSEQ_TEST_WNJG_0007_RDS |           0 |           0 | 0%            | 0%
|
+------+--------------------------------------------------+-------------------+-----------------+--------------------+----
----------------+
8 rows in set (0.01 sec)
```

SQL 调优类语句

SQL 调优类语句如下：

- SHOW [FULL] SLOW [WHERE expr] [limit expr] 语句
- SHOW [FULL] PHYSICAL_SLOW [WHERE expr] [limit expr] 语句
- CLEAR SLOW 语句
- EXPLAIN SQL 语句
- EXPLAIN DETAIL SQL 语句
- EXPLAIN EXECUTE SQL 语句
- TRACE SQL 和 SHOW TRACE 语句
- CHECK TABLE tablename 语句
- SHOW TABLE STATUS 语句

1. SHOW [FULL] SLOW [WHERE expr] [limit expr]

执行时间超过 1 秒的 SQL 语句是慢 SQL，逻辑慢 SQL 是指应用发送到 DRDS 的慢 SQL。如何排查慢 SQL 可以参考文档 排查 DRDS 慢 SQL。

- SHOW SLOW: 查看自 DRDS 启动或者上次执行CLEAR SLOW以来最慢的 100 条逻辑慢 SQL（注意，这里记录的是最慢的 100 个，缓存在 DRDS 系统中，当实例重启或者执行 CLEAR SLOW 时会丢失）；
- SHOW FULL SLOW: 查看实例启动以来记录的所有逻辑慢 SQL（持久化到 DRDS 的内置数据库中）。该记录数有一个上限（具体数值跟购买的实例规格相关），DRDS 会滚动删除比较老的慢 SQL 语句。实例的规格如果是 4C4G 的话，最多记录 10000 条慢 SQL 语句（包括逻辑慢 SQL 和 物理慢 SQL）；实例的规格如果是 8C8G 的话，最多记录 20000 条慢 SQL 语句（包括逻辑慢 SQL 和 物理慢 SQL），其它规格依此类推。

重要列详解：

- **HOST**：来源 IP；
- **START_TIME**：执行开始时间；
- **EXECUTE_TIME**：执行时间；
- **AFFECT_ROW**：对于 DML 语句是影响行数；对于查询语句是返回的记录数。

```
mysql> show slow where execute_time > 1000 limit 1;
+-----------+---------------------+--------------+------------+-----------+
| HOST      | START_TIME          | EXECUTE_TIME | AFFECT_ROW | SQL       |
+-----------+---------------------+--------------+------------+-----------+
| 127.0.0.1 | 2016-03-16 13:02:57 |         2785 |          7 | show rule |
+-----------+---------------------+--------------+------------+-----------+
1 row in set (0.02 sec)
```

2. SHOW [FULL] PHYSICAL_SLOW [WHERE expr] [limit expr]

执行时间超过1秒的 SQL 语句是慢 SQL，物理慢 SQL 是指 DRDS 发送到 RDS 的慢 SQL。如何排查慢 SQL 可以参考文档 **排查 DRDS 慢 SQL**。

- SHOW PHYSICAL_SLOW: 查看自 DRDS 启动或者上次执行CLEAR SLOW以来最慢的 100 条物理慢 SQL（注意，这里记录的是最慢的 100 个，缓存在 DRDS 系统中，当实例重启或者执行 CLEAR SLOW 时会丢失）；
- SHOW FULL PHYSICAL_SLOW: 查看实例启动以来记录的所有物理慢 SQL（持久化到 DRDS 的内置数据库中）。该记录数有一个上限（具体数值跟购买的实例规格相关），DRDS 会滚动删除比较老的慢 SQL 语句。实例的规格如果是 4C4G 的话，最多记录 10000 条慢 SQL 语句（包括逻辑慢 SQL 和物理慢 SQL）；实例的规格如果是 8C8G 的话，最多记录 20000 条慢 SQL 语句（包括逻辑慢 SQL 和 物理慢 SQL），其它规格依此类推。

重要列详解：

- **GROUP_NAME**：数据库分组；
- **START_TIME**：执行开始时间；
- **EXECUTE_TIME**：执行时间；
- **AFFECT_ROW**：对于 DML 语句是影响行数；对于查询语句是返回的记录数。

```
mysql> show physical_slow;
+----------------+----------------------------------+---------------------+--------------+------------------+-------------
-----------+------------------------+------------+-----------------+
| GROUP_NAME     | DBKEY_NAME                       | START_TIME          | EXECUTE_TIME | SQL_EXECUTE_TIME |
GETLOCK_CONNECTION_TIME | CREATE_CONNECTION_TIME | AFFECT_ROW | SQL             |
+----------------+----------------------------------+---------------------+--------------+------------------+-------------
-----------+------------------------+------------+-----------------+
| TDDL5_00_GROUP | db218249098_sqa_zmf_tddl5_00_3309 | 2016-03-16 13:05:38 |         1057 |             1011 |
0 |                      0 |          1 | select sleep(1) |
+----------------+----------------------------------+---------------------+--------------+------------------+-------------
-----------+------------------------+------------+-----------------+
1 row in set (0.01 sec)
```

3. CLEAR SLOW

清空自 DRDS 启动或者上次执行CLEAR SLOW以来最慢的 100 条逻辑慢 SQL 和 最慢的 100 条物理慢 SQL。注意：SHOW SLOW 和 SHOW PHYSICAL_SLOW展示的是最慢的 100 个 SQL，如果长时间未执行CLEAR SLOW，可能都是非常老的 SQL 了，一般执行过 SQL 优化之后，建议都执行下CLEAR SLOW，等待系统运行一段时间，再查看下慢 SQL 的优化效果。

```
mysql> clear slow;
Query OK, 0 rows affected (0.00 sec)
```

4. EXPLAIN SQL

查看指定 SQL 在 DRDS 层面的执行计划，注意这条 SQL 不会实际执行。关于该指令的更多案例请参考文档 SQL 优化基本概念、SQL 优化方法 。

示例：

查看select * from doctest这条 SQL 的执行计划（doctest 这张表是按照 id 列进行分库的）。从执行计划可以看出该 SQL 会下推到每个分库，然后将执行结果聚合。

```
mysql> explain select * from doctest;
+------------------------------------------------+-------------------------------------+--------+
| GROUP_NAME                                     | SQL                                 | PARAMS |
+------------------------------------------------+-------------------------------------+--------+
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0000_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0001_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0002_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0003_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0004_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0005_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0006_RDS | select `doctest`.`id` from `doctest` | {}     |
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0007_RDS | select `doctest`.`id` from `doctest` | {}     |
+------------------------------------------------+-------------------------------------+--------+
8 rows in set (0.00 sec)
```

查看select * from doctest where id = 1这条 SQL 的执行计划（doctest 这张表是按照 id 列进行分库的）。从执行计划可以看出该 SQL 会根据拆分键（id）计算出具体的分库，将 SQL 直接下推到该分库，然后执行结果聚合。

```
mysql> explain select * from doctest where id = 1;
+------------------------------------------------+-----------------------------------------------------------------------+--------+
| GROUP_NAME                                     | SQL                                                                   | PARAMS |
+------------------------------------------------+-----------------------------------------------------------------------+--------+
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0001_RDS | select `doctest`.`id` from `doctest` where (`doctest`.`id` = 1) | {}     |
+------------------------------------------------+-----------------------------------------------------------------------+--------+
1 row in set (0.01 sec)
```

5. EXPLAIN DETAIL SQL

查看指定 SQL 在 DRDS 层面的执行计划。注意这条 SQL 不会实际执行。关于该指令的更多案例请参考文档 SQL 优化基本概念、SQL 优化方法 。

```
mysql> explain detail select * from doctest where id = 1;
+---------------------------------------------+---------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+--
------+
| GROUP_NAME                                  | SQL
| PARAMS |
+---------------------------------------------+---------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+--
------+
| DOCTEST_1488704345426RCUPDOCTEST_CAET_0001_RDS | Query from doctest as doctest
    keyFilter:doctest.id = 1
    queryConcurrency:SEQUENTIAL
    columns:[doctest.id]
    tableName:doctest
    executeOn:DOCTEST_1488704345426RCUPDOCTEST_CAET_0001_RDS
 | NULL   |
+---------------------------------------------+---------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+--
------+
1 row in set (0.02 sec)
```

6. EXPLAIN EXECUTE SQL

查看底层存储的执行计划，等同于 MYSQL 的 EXPLAIN 语句。关于该指令的更多案例请参考文档 SQL 优化基本概念、SQL 优化方法 。

```
mysql> explain execute select * from tddl_mgr_log limit 1;
+----+-------------+--------------+------+---------------+------+---------+------+------+-------+
| id | select_type | table        | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+--------------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | tddl_mgr_log | ALL  | NULL          | NULL | NULL    | NULL |    1 | NULL  |
+----+-------------+--------------+------+---------------+------+---------+------+------+-------+
1 row in set (0.07 sec)
```

7. TRACE SQL 和 SHOW TRACE

查看具体 SQL 的执行情况。TRACE [SQL] 和 SHOW TRACE 要结合使用。注意 TRACE SQL 和 EXPLAIN SQL 的区别在于 TRACE SQL 会实际执行该语句。 关于该指令的更多案例请参考文档 SQL 优化基本概念、SQL 优化方法 。

例如查看 select 1这条语句的执行情况。

```
mysql> trace select 1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.03 sec)
mysql> show trace;
+------+----------+---------------+---------------------------------+--------------+-------------------------+------
+-----------+--------+
```

```
| ID   | TYPE     | GROUP_NAME     | DBKEY_NAME                      | TIME_COST(MS) | CONNECTION_TIME_COST(MS) |
ROWS | STATEMENT | PARAMS |
+------+----------+----------------+---------------------------------+---------------+--------------------------+------
+-----------+--------+
|    0 | Optimize | DRDS           | DRDS                            | 3             | 0.00                     |    0 | select 1  | NULL   |
|    1 | Query    | TDDL5_00_GROUP | db218249098_sqa_zmf_tddl5_00_3309 | 7             | 0.15                     |    1 | select
1  | NULL   |
+------+----------+----------------+---------------------------------+---------------+--------------------------+------
+-----------+--------+
2 rows in set (0.01 sec)
```

8. CHECK TABLE tablename

对数据表进行检查。主要用于 DDL 建表失败的情形，应用案例请参考文档 DDL 常见问题处理 。

- 对于拆分表，检查底层物理分表是否有缺失的情况，底层的物理分表的列和索引是否是一致；
- 对于单库单表，检查表是否存在。

```
mysql>  check table tddl_mgr_log;
+------------------------+-------+----------+----------+
| TABLE                  | OP    | MSG_TYPE | MSG_TEXT |
+------------------------+-------+----------+----------+
| TDDL5_APP.tddl_mgr_log | check | status   | OK       |
+------------------------+-------+----------+----------+
1 row in set (0.56 sec)
mysql> check table tddl_mg;
+-------------------+-------+----------+---------------------------------------+
| TABLE             | OP    | MSG_TYPE | MSG_TEXT                              |
+-------------------+-------+----------+---------------------------------------+
| TDDL5_APP.tddl_mg | check | Error    | Table 'tddl5_00.tddl_mg' doesn't exist |
+-------------------+-------+----------+---------------------------------------+
1 row in set (0.02 sec)
```

9. SHOW TABLE STATUS [LIKE 'pattern' | WHERE expr]

获取表的信息，该指令聚合了底层各个物理分表的数据。

重要列详解：

- **NAME**：表名称；
- **ENGINE**：表的存储引擎；
- **VERSION**：表的存储引擎的版本；
- **ROW_FORMAT**：行格式，主要是 Dynamic、Fixed、Compressed 这三种格式。动态（Dynamic）行的行长度可变，例如 VARCHAR 或 BLOB 类型字段；固定（Fixed）行是指行长度不变，例如 CHAR 和 INTEGER 类型字段；
- **ROWS**：表中的行数；
- **AVG_ROW_LENGTH**：平均每行包括的字节数；
- **DATA_LENGTH**：整个表的数据量（单位：字节）；
- **MAX_DATA_LENGTH**：表可以容纳的最大数据量；
- **INDEX_LENGTH**：索引占用磁盘的空间大小 ；
- **CREATE_TIME**：表的创建时间；

- **UPDATE_TIME**：表的最近更新时间；
- **COLLATION**：表的默认字符集和字符排序规则；
- **CREATE_OPTIONS**：指表创建时的其他所有选项。

```
mysql> show table status like 'multi_db_multi_tbl';
+--------------------+--------+---------+------------+------+----------------+-------------+-----------------+-------------
--+-----------+-----------------+-----------------------+-------------+-----------+-----------------+----------+----------
------+---------+
| NAME               | ENGINE | VERSION | ROW_FORMAT | ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
MAX_DATA_LENGTH | INDEX_LENGTH | DATA_FREE | AUTO_INCREMENT | CREATE_TIME           | UPDATE_TIME |
CHECK_TIME | COLLATION       | CHECKSUM | CREATE_OPTIONS | COMMENT |
+--------------------+--------+---------+------------+------+----------------+-------------+-----------------+-------------
--+-----------+-----------------+-----------------------+-------------+-----------+-----------------+----------+----------
------+---------+
| multi_db_multi_tbl | InnoDB |      10 | Compact    |    2 |          16384 |       16384 |               0 |        16384 |         0 |
100000 | 2017-03-27 17:43:57.0 | NULL        | NULL       | utf8_general_ci | NULL     |                |         |
+--------------------+--------+---------+------------+------+----------------+-------------+-----------------+-------------
--+-----------+-----------------+-----------------------+-------------+-----------+-----------------+----------+----------
------+---------+
1 row in set (0.03 sec)
```

和 DRDS 的 SCAN HINT 结合，还可以查看每个物理分表的数据量。具体请参考 HINT 语法。

```
mysql> /!TDDL:SCAN='multi_db_multi_tbl'*/show table status like 'multi_db_multi_tbl';
+----------------------+--------+---------+------------+------+----------------+-------------+-----------------+----------
----+-----------+----------------+---------------------+------------+-----------+-----------------+----------+----------
------+---------+--------------+
| Name                 | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length |
Index_length | Data_free | Auto_increment | Create_time         | Update_time | Check_time | Collation       | Checksum |
Create_options | Comment | Block_format |
+----------------------+--------+---------+------------+------+----------------+-------------+-----------------+----------
----+-----------+----------------+---------------------+------------+-----------+-----------------+----------+----------
------+---------+--------------+
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    1 |          16384 |       16384 |               0 |        16384 |         0 |
2 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_1 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
| multi_db_multi_tbl_0 | InnoDB |      10 | Compact    |    0 |              0 |       16384 |               0 |        16384 |         0 |
1 | 2017-03-27 17:43:57 | NULL        | NULL       | utf8_general_ci |     NULL |                |         | Original     |
```

```
| multi_db_multi_tbl_1 | InnoDB |     10 | Compact    |    0 |          0 |      16384 |          0 |      16384 |       0 |
1 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
| multi_db_multi_tbl_0 | InnoDB |     10 | Compact    |    0 |          0 |      16384 |          0 |      16384 |       0 |
1 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
| multi_db_multi_tbl_1 | InnoDB |     10 | Compact    |    0 |          0 |      16384 |          0 |      16384 |       0 |
1 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
| multi_db_multi_tbl_0 | InnoDB |     10 | Compact    |    0 |          0 |      16384 |          0 |      16384 |       0 |
1 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
| multi_db_multi_tbl_1 | InnoDB |     10 | Compact    |    0 |          0 |      16384 |          0 |      16384 |       0 |
1 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
| multi_db_multi_tbl_0 | InnoDB |     10 | Compact    |    1 |      16384 |      16384 |          0 |      16384 |       0 |
3 | 2017-03-27 17:43:57 | NULL       | NULL       | utf8_general_ci |    NULL |            |       | Original    |
+----------------------+--------+---------+------------+------+---------------+------------+----------------+----------
----+-----------+----------------+--------------------+------------+------------+----------------+----------+----------
------+---------+--------------+
16 rows in set (0.04 sec)
```

统计信息查询类语句

统计信息查询类语句

DRDS 提供以下语句用于查询实时统计信息。

SHOW [FULL] STATS

SHOW DB STATUS

SHOW FULL DB STATUS

SHOW [FULL] STATS

查看整体的统计信息，这些信息都是瞬时值。注意不同版本的 DRDS SHOW FULL STATS的结果是有区别的。

重要列说明：

- **QPS**：应用到 DRDS 的 QPS，通常称为逻辑 QPS；
- **RDS_QPS**：DRDS 到 RDS 的 QPS，通常称为物理 QPS；
- **ERROR_PER_SECOND**：每秒的错误数，包含 SQL 语法错误，主键冲突，系统错误，连通性错误等各类错误总和；
- **VIOLATION_PER_SECOND**：每秒的主键或者唯一键冲突；
- **MERGE_QUERY_PER_SECCOND**：通过分库分表，从多表中进行的查询；

- **ACTIVE_CONNECTIONS**：正在使用的连接；
- **CONNECTION_CREATE_PER_SECCOND**：每秒创建的连接数；
- **RT(MS)**：应用到 DRDS 的响应时间，通常称为逻辑 RT（响应时间）；
- **RDS_RT(MS)**：DRDS 到 RDS/MySQL 的响应时间，通常称为物理 RT；
- **NET_IN(KB/S)**：DRDS 收到的网络流量；
- **NET_OUT(KB/S)**：DRDS 输出的网络流量；
- **THREAD_RUNNING**：正在运行的线程数；
- **HINT_USED_PER_SECOND**：每秒带 HINT 的查询的数量；
- **HINT_USED_COUNT**：启动到现在带 HINT 的查询总量；
- **AGGREGATE_QUERY_PER_SECCOND**：每秒聚合查询的频次；
- **AGGREGATE_QUERY_COUNT**：聚合查询总数（历史累计数据）；
- **TEMP_TABLE_CREATE_PER_SECCOND**：每秒创建的临时表的数量；
- **TEMP_TABLE_CREATE_COUNT**：启动到现在创建的临时表总数量；
- **MULTI_DB_JOIN_PER_SECCOND**：每秒跨库 JOIN 的数量；
- **MULTI_DB_JOIN_COUNT**：启动到现在跨库 JOIN 的总量。

示例：

```
mysql> show stats;
+------+---------+----------+-----------------+-----------------+----------------------+------------------+--------
+------------+-------------+--------------+---------------+
| QPS | RDS_QPS | SLOW_QPS | PHYSICAL_SLOW_QPS | ERROR_PER_SECOND | MERGE_QUERY_PER_SECOND |
ACTIVE_CONNECTIONS | RT(MS) | RDS_RT(MS) | NET_IN(KB/S) | NET_OUT(KB/S) | THREAD_RUNNING |
+------+---------+----------+-----------------+-----------------+----------------------+------------------+--------
+------------+-------------+--------------+---------------+
| 1.77 | 1.68 | 0.03 | 0.03 | 0.02 | 0.00 | 7 | 157.13 | 51.14 | 134.49 | 1.48 | 1 |
+------+---------+----------+-----------------+-----------------+----------------------+------------------+--------
+------------+-------------+--------------+---------------+
1 row in set (0.01 sec)

mysql> show full stats;
+------+---------+----------+-----------------+-----------------+--------------------+----------------------+------
-------------+---------------------------+--------+-----------+-------------+--------------+---------------+-------
--------------+---------------+-------------------------+---------------------+------------------------------+------
------------------+-----------------------+--------------------+-------+---------+------------+-----------+
| QPS | RDS_QPS | SLOW_QPS | PHYSICAL_SLOW_QPS | ERROR_PER_SECOND | VIOLATION_PER_SECOND |
MERGE_QUERY_PER_SECOND | ACTIVE_CONNECTIONS | CONNECTION_CREATE_PER_SECOND | RT(MS) |
RDS_RT(MS) | NET_IN(KB/S) | NET_OUT(KB/S) | THREAD_RUNNING | HINT_USED_PER_SECOND | HINT_USED_COUNT
| AGGREGATE_QUERY_PER_SECOND | AGGREGATE_QUERY_COUNT | TEMP_TABLE_CREATE_PER_SECOND |
TEMP_TABLE_CREATE_COUNT | MULTI_DB_JOIN_PER_SECOND | MULTI_DB_JOIN_COUNT | CPU | FREEMEM |
FULLGCCOUNT | FULLGCTIME |
+------+---------+----------+-----------------+-----------------+--------------------+----------------------+------
-------------+---------------------------+--------+-----------+-------------+--------------+---------------+-------
--------------+---------------+-------------------------+---------------------+------------------------------+------
------------------+-----------------------+--------------------+-------+---------+------------+-----------+
| 1.63 | 1.68 | 0.03 | 0.03 | 0.02 | 0.00 | 0.00 | 6 | 0.01 | 157.13 | 51.14 | 134.33 | 1.21 | 1 | 0.00 | 54 | 0.00 | 663 | 0.00 |
512 | 0.00 | 516 | 0.09% | 6.96% | 76446 | 21326906 |
+------+---------+----------+-----------------+-----------------+--------------------+----------------------+------
-------------+---------------------------+--------+-----------+-------------+--------------+---------------+-------
--------------+---------------+-------------------------+---------------------+------------------------------+------
------------------+-----------------------+--------------------+-------+---------+------------+-----------+
```

```
1 row in set (0.01 sec)
```

SHOW DB STATUS

用于查看物理库容量/性能信息，所有返回值为实时信息。 容量信息通过 MySQL 系统表获得，与真实容量情况可能有差异。

重要列说明：

- **NAME：** 代表一个 DRDS DB，此处显示的是 DRDS 内部标记，与 DRDS DB 名称不同；
- **CONNECTION_STRING：** 分库的连接信息；
- **PHYSICAL_DB：**分库名称，TOTAL 行代表一个 DRDS DB 中所有分库容量的总和；
- **SIZE_IN_MB：** 分库中数据占用的空间，单位为 MB；
- **RATIO：** 单个分库数据量在当前 DRDS DB 总数据量中的占比；
- **THREAD_RUNNING：** 物理数据库实例当前正在执行的线程情况，含义与 MySQL SHOW GLOBAL STATUS 指令返回值的含义相同，详情请参考 MySQL 文档。

示例：

```
mysql> show db status;
+------+--------------------------+-------------------+------------------+------------+--------+----------------+
| ID | NAME | CONNECTION_STRING | PHYSICAL_DB | SIZE_IN_MB | RATIO | THREAD_RUNNING |
+------+--------------------------+-------------------+------------------+------------+--------+----------------+
| 1 | drds_db_1516187088365daui | 100.100.64.1:59077 | TOTAL | 13.109375 | 100% | 3 |
| 2 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | 1.578125 | 12.04% | |
| 3 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | 1.4375 | 10.97% | |
| 4 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | 1.4375 | 10.97% | |
| 5 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | 1.4375 | 10.97% | |
| 6 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | 1.734375 | 13.23% | |
| 7 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | 1.734375 | 13.23% | |
| 8 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | 2.015625 | 15.38% | |
| 9 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | 1.734375 | 13.23% | |
+------+--------------------------+-------------------+------------------+------------+--------+----------------+
```

SHOW FULL DB STATUS [LIKE {tablename}]

用于查看物理库表容量和性能信息，所有返回值为实时信息。 容量信息通过 MySQL 系统表获得，与真实容量情况可能有差异。
重要列说明：

- **NAME：** 代表一个 DRDS DB。此处显示的是 DRDS 内部标记，与 DRDS DB 名称不同；
- **CONNECTION_STRING：** 分库的连接信息；
- **PHYSICAL_DB：**分库名称，TOTAL 行代表经过 LIKE 关键字筛选后得到的分库容量的总和。如果没有 LIKE，则为全部分库容量的总和；
- **PHYSICAL_TABLE：**分表名称，TOTAL 行代表经过 LIKE 关键字筛选后得到的分表容量的总和。如果没有LIKE，则为全部分表容量的总和；
- **SIZE_IN_MB：** 分表中数据占用的空间，单位为 MB；

- **RATIO:** 单个分表数据量在当前筛选出的分表总数据量中的占比；
- **THREAD_RUNNING：** 物理数据库实例当前正在执行的线程情况，含义与 MySQL SHOW GLOBAL STATUS 指令返回值的含义相同。详情请参考 MySQL 文档。

示例：

```
mysql> show full db status like hash_tb;
+------+--------------------------+-------------------+------------------+---------------+-----------+--------+----------+
| ID | NAME | CONNECTION_STRING | PHYSICAL_DB | PHYSICAL_TABLE | SIZE_IN_MB | RATIO | THREAD_RUNNING |
+------+--------------------------+-------------------+------------------+---------------+-----------+--------+----------+
| 1 | drds_db_1516187088365daui | 100.100.64.1:59077 | TOTAL | | 19.875 | 100% | 3 |
| 2 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | TOTAL | 3.03125 | 15.25% | |
| 3 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | hash_tb_00 | 1.515625 | 7.63% | |
| 4 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0000 | hash_tb_01 | 1.515625 | 7.63% | |
| 5 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | TOTAL | 2.0 | 10.06% | |
| 6 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | hash_tb_02 | 1.515625 | 7.63% | |
| 7 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0001 | hash_tb_03 | 0.484375 | 2.44% | |
| 8 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | TOTAL | 3.03125 | 15.25% | |
| 9 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | hash_tb_04 | 1.515625 | 7.63% | |
| 10 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0002 | hash_tb_05 | 1.515625 | 7.63% | |
| 11 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | TOTAL | 1.953125 | 9.83% | |
| 12 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | hash_tb_06 | 1.515625 | 7.63% | |
| 13 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0003 | hash_tb_07 | 0.4375 | 2.2% | |
| 14 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | TOTAL | 3.03125 | 15.25% | |
| 15 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | hash_tb_08 | 1.515625 | 7.63% | |
| 16 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0004 | hash_tb_09 | 1.515625 | 7.63% | |
| 17 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | TOTAL | 1.921875 | 9.67% | |
| 18 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | hash_tb_11 | 1.515625 | 7.63% | |
| 19 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0005 | hash_tb_10 | 0.40625 | 2.04% | |
| 20 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | TOTAL | 3.03125 | 15.25% | |
| 21 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | hash_tb_12 | 1.515625 | 7.63% | |
| 22 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0006 | hash_tb_13 | 1.515625 | 7.63% | |
| 23 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | TOTAL | 1.875 | 9.43% | |
| 24 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | hash_tb_14 | 1.515625 | 7.63% | |
| 25 | drds_db_1516187088365daui | 100.100.64.1:59077 | drds_db_xzip_0007 | hash_tb_15 | 0.359375 | 1.81% | |
+------+--------------------------+-------------------+------------------+---------------+-----------+--------+----------+
```

SHOW PROCESSLIST 指令与 KILL 指令

功能版本说明

1. 当 DRDS 版本号小于 5.1.28-1408022 时，DRDS 仅支持物理连接的 SHOW PROCESSLIST 与 KILL 功能，请参考老版本文档老版本 SHOW PROCESSLIST 指令与 KILL 指令。

2. 当 DRDS 版本号大于等于 5.1.28-1408022 时，DRDS 支持逻辑连接与物理连接的 SHOW PROCESSLIST 与 KILL 功能，请继续阅读此文档。

获取 DRDS 版本号，DRDS 自助升级的方法以及更多的版本介绍请参考版本说明文档。

SHOW PROCESSLIST 指令

DRDS 中，可以使用 SHOW PROCESSLIST 指令查看 DRDS 中的连接与正在执行的 SQL 等信息。

语法：

```
SHOW [FULL] PROCESSLIST
```

示例：

```
mysql> SHOW PROCESSLIST\G
     ID: 1971050
   USER: admin
   HOST: 111.111.111.111:4303
     DB: drds_test
COMMAND: Query
   TIME: 0
  STATE:
   INFO: show processlist
1 row in set (0.01 sec)
```

结果集各字段含义：

- ID：连接的 ID，为一个 Long 型数字。
- USER：建立此连接所使用的用户名。
- HOST：建立此连接的机器的 IP 与端口。
- DB：此连接所访问的库名。
- COMMAND，目前有两种取值：
    - Query，代表当前连接正在执行 SQL 语句；
    - Sleep，代表当前连接正处于空闲状态。
- TIME， 连接处于当前状态持续的时间：
    - 当 COMMAND 为 Query 时，代表当此连接上正在执行的 SQL 已经执行的时间；
    - 当 COMMAND 为 Sleep 时，代表当此连接空闲的时间。
- STATE：目前无意义，恒为空值。
- INFO：
    - 当 COMMAND 为 Query 时，为此连接上正在执行的 SQL 的内容。当不带 FULL 参数时，最多返回正在执行的 SQL 的前 30 个字符。当带 FULL 参数时，最多返回正在执行的 SQL 的前1000个字符；
    - 当 COMMAND 为其他值时，无意义，为空值。

SHOW PHYSICAL_PROCESSLIST 指令

DRDS 中，可以使用 SHOW PHYSICAL_PROCESSLIST 指令查看底层所有 MySQL/RDS 上正在执行的 SQL 信息。

语法：

```
SHOW [FULL] PHYSICAL_PROCESSLIST
```

当 SQL 比较长的时候，SHOW PHYSICAL_PROCESSLIST 会截断，这时可以使用 SHOW FULL PHYSICAL_PROCESSLIST 获取完整 SQL。

返回结果中每一列的含义与 MySQL 的 SHOW PROCESSLIST 指令等价，请参考 SHOW PROCESSLIST Syntax。

注意：与 MySQL 不同，DRDS 返回的物理连接的 ID 列为一个字符串，并非一个数字。

示例：

```
mysql> SHOW PHYSICAL_PROCESSLIST\G
*************************** 1. row ***************************
          ID: 0-0-521414
        USER: tddl5
          DB: tddl5_00
     COMMAND: Query
        TIME: 0
       STATE: init
        INFO: show processlist
*************************** 2. row ***************************
          ID: 0-0-521570
        USER: tddl5
          DB: tddl5_00
     COMMAND: Query
        TIME: 0
       STATE: User sleep
        INFO: /*DRDS /88.88.88.88/b67a0e4d8800000/ */ select sleep(1000)
2 rows in set (0.01 sec)
```

KILL 指令

KILL 指令用于终止一个正在执行的 SQL。

DRDS 使用 DRDS 在 MySQL/RDS 上创建的用户名连接 MySQL/RDS，所以一般直接连接 MySQL/RDS 是没有权限对 DRDS 发起的请求进行 KILL 操作的。

如果需要终止一个 DRDS 上正在执行的 SQL，需要使用 MySQL 命令行、DMS 等工具连接 DRDS，在 DRDS 上执行 KILL 指令。

语法：

```
KILL PROCESS_ID | 'PHYSICAL_PROCESS_ID' | 'ALL'
```

有三种用法：

终止一个特定的逻辑 SQL： KILL PROCESS_ID

PROCESS_ID 为 SHOW [FULL] PROCESSLIST 指令返回的 ID 列。

DRDS 中，KILL PROCESS_ID 指令会将此连接正在执行的逻辑 SQL 与物理 SQL 均终止掉，并断开此连接。

DRDS 不支持 KILL QUERY 指令。

终止一个特定的物理 SQL： KILL 'PHYSICAL_PROCESS_ID'

其中的 PHYSICAL_PROCESS_ID 来自SHOW PHYSICAL_PROCESS_ID指令返回的 ID 列。

注意：由于 PHYSICAL_PROCESS_ID 列为一个字符串，并非一个数字，因此 KILL 指令中，PHYSICAL_PROCESS_ID 需要使用单引号括起来。

示例：

```
mysql> KILL '0-0-521570';
Query OK, 0 rows affected (0.01 sec)
```

终止当前库上所有通过 DRDS 执行的物理 SQL： KILL 'ALL'

当底层 MySQL/RDS 因为一些 SQL 导致压力非常大的时候，可以使用 KILL 'ALL' 指令终止当前 DRDS 库上所有正在执行的物理 SQL。

符合以下条件的物理 PROCESS 会被KILL 'ALL'指令终止：

- 该 PROCESS 的 User 是 DRDS 在 MySQL/RDS 上所创建的用户名；
- 该 PROCESS 正在执行查询，也即 COMMAND 为 Query。

老版本 SHOW PROCESSLIST 指令与 KILL 指令

功能版本说明

1. 当 DRDS 版本号小于 5.1.28-1408022 时，DRDS 仅支持物理连接的 SHOW PROCESSLIST 与 KILL 功能，请继续阅读此文档。
2. 当 DRDS 版本号大于等于 5.1.28-1408022 时，DRDS 支持逻辑连接与物理连接的 SHOW

PROCESSLIST 与 KILL 功能，请参考文档 SHOW PROCESSLIST 指令与 KILL 指令。
获取 DRDS 版本号，DRDS 自助升级的方法以及更多的版本介绍请参考版本说明文档。

SHOW PROCESSLIST 指令

DRDS 中，可以使用SHOW PROCESSLIST指令查看底层所有 MySQL/RDS 上正在执行的 SQL 信息。

语法：

```
SHOW [FULL] PROCESSLIST
```

当 SQL 比较长的时候，SHOW PROCESSLIST会截断，这时可以使用SHOW FULL PROCESSLIST获取完整 SQL。

返回结果中每一列的含义与 MySQL 的SHOW PROCESSLIST指令等价，请参考 SHOW PROCESSLIST Syntax 。

示例：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
          ID: 0-0-521414
        USER: tddl5
          DB: tddl5_00
     COMMAND: Query
        TIME: 0
       STATE: init
        INFO: show processlist
   ROWS_SENT: NULL
ROWS_EXAMINED: NULL
   ROWS_READ: NULL
*************************** 2. row ***************************
          ID: 0-0-521570
        USER: tddl5
          DB: tddl5_00
     COMMAND: Query
        TIME: 0
       STATE: User sleep
        INFO: /*DRDS /88.88.88.88/b67a0e4d8800000/ */ select sleep(1000)
   ROWS_SENT: NULL
ROWS_EXAMINED: NULL
   ROWS_READ: NULL
2 rows in set (0.01 sec)
```

KILL 指令

KILL 指令用于终止一个正在执行的 SQL。

DRDS 使用 DRDS 在 MySQL/RDS 上创建的用户名连接 MySQL/RDS，所以一般直接连接 MySQL/RDS 是没有权限对 DRDS 发起的请求进行 KILL 操作的。

如果需要终止一个 DRDS 上正在执行的 SQL，需要使用 MySQL 命令行、DMS 等工具连接 DRDS，在 DRDS 上执行 KILL 指令。

语法：

```
KILL 'PROCESS_ID' | 'ALL'
```

有两种用法：

终止一个特定的 SQL： KILL 'PROCESS_ID'

其中的 PROCESS_ID 来自SHOW PROCESSLIST指令返回的 ID 列。

注意：与 MySQL 不同，DRDS 返回的 ID 列为一个字符串，并非一个数字，因此 KILL 指令中，PROCESS_ID 需要使用单引号括起来。

示例：

```
mysql> KILL '0-0-521570';
Query OK, 0 rows affected (0.01 sec)
```

终止当前库上所有通过 DRDS 执行的 SQL： KILL 'ALL'

当底层 MySQL/RDS 因为一些 SQL 导致压力非常大的时候，可以使用KILL 'ALL'指令终止当前 DRDS 库上所有正在执行的 SQL。

符合以下条件的 PROCESS 会被KILL 'ALL'指令终止：

- 该 PROCESS 的 User 是 DRDS 在 MySQL/RDS 上所创建的用户名；
- 该 PROCESS 正在执行查询，即 COMMAND 为 Query。

注意：较低版本的 DRDS 实例不支持KILL 'ALL'功能，执行的时候会报错。可以将 DRDS 实例升级到最新版本，升级方法请参考升级实例版本。

DRDS 自定义 HINT

自定义 HINT 简介

DRDS 自定义 HINT 概要

HINT 作为一种 SQL 补充语法，在关系型数据库中扮演着非常重要的角色。它允许用户通过相关的语法影响 SQL 的执行方式，从对 SQL 进行特殊的优化。同样，DRDS 也提供了特殊的 HINT 语法。

例如，假设已知目标数据在某些分库的分表中，需要直接将 SQL 下发到该分库执行，就可以使用 DRDS 自定义 HINT 来完成。

```
/!TDDL:NODE IN('node_name', ...)*/SELECT * FROM table_name;
```

这个 SQL 语句中/!和*/之间的语句就是 DRDS 的自定义 HINT，即TDDL:node in('node_name', ...)，它指定了 SQL 语句在特定的 RDS 分库上执行。

注意：

> **DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。**
>
> **如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。**

DRDS 自定义 HINT 语法

基本语法：

```
/!TDDL:hint command*/
```

DRDS 自定义 HINT 是借助于 MySQL 注释实现的，也就是 DRDS 的自定义 HINT 语句位于/!与*/之间，并且必须以TDDL:开头。其中 hint command 是 DRDS 自定义 HINT 命令，与具体的操作相关。例如下面的 SQL 语句通过 DRDS 的自定义 HINT 展示每个分库的表名。

```
/!TDDL:SCAN*/SHOW TABLES;
```

该 SQL 语句中 /!TDDL:SCAN*/ 为 DRDS 自定义 HINT 部分，以TDDL:开头，SCAN 为 DRDS 自定义 HINT 命令。

DRDS 自定义 HINT 分类

根据操作类型的不同，DRDS 的自定义 HINT 主要可以分为以下几类：

- 读写分离
- 备库延迟切断
- 自定义 SQL 超时时间
- 指定分库执行 SQL
- 扫描全部分库分表

读写分离

DRDS 提供了一种针对应用层透明的读写分离实现。但是由于 RDS 主实例与只读实例之间数据的同步存在着毫秒级别的延迟，如果在主库中变更以后需要马上读取变更的数据，则需要保证将读取数据的 SQL 下发到主实例中。针对这种需求，DRDS 提供了读写分离自定义 HINT，指定将 SQL 下发到主实例或者只读实例。

语法

```
/!TDDL:MASTER|SLAVE*/
```

在该自定义 HINT 中可以指定 SQL 是在主实例上执行还是在只读实例上执行。对于/!TDDL:SLAVE*/这个自定义 HINT，如果一个主 RDS 实例存在多个只读实例，那么 DRDS 会根据所分配的权重随机选择一个只读实例执行 SQL 语句。

注意：

DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。

如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。

示例

指定 SQL 在主实例上执行：

```
/!TDDL:MASTER*/SELECT * FROM table_name;
```

在 SQL 语句前添加 /!TDDL:MASTER*/ 这个自定义 HINT 后，这条 SQL 将被下发到主实例上执行。

指定 SQL 在只读实例上执行：

```
/!TDDL:SLAVE*/SELECT * FROM table_name;
```

在 SQL 语句前添加 /!TDDL:SLAVE*/ 这个自定义 HINT 后，这条 SQL 将会根据所分配的权重被随机下发到某个只读实例上执行。

注意：

- 此读写分离自定义 HINT 仅仅针对非事务中的读 SQL 语句生效，如果 SQL 语句是写 SQL 或者 SQL 语句在事务中，那么还是会下发到 RDS 的主实例执行。
- DRDS 针对 /!TDDL:SLAVE*/ 自定义 HINT，会从只读实例中按照权重随机选取一个下发 SQL 语句执行。若只读实例不存在时，不会报错，而是选取主实例执行。

只读实例延迟切断

正常情况下，如果给 DRDS 数据库的 RDS 主实例配置了只读实例，并且给主实例和只读实例都设置了读流量，那么 DRDS 会根据读写比例将 SQL 下发到主实例或者是只读实例执行。但是如果主实例与只读实例的异步数据复制存在较大的延迟，将 SQL 下发到只读实例执行就会导致出错或者返回错误结果。

只读实例延时切断会根据主备复制最大延时时间判断将所执行的 SQL 下发到主实例还是只读实例。

语法

```
/!TDDL:SQL_DELAY_CUTOFF=time*/
```

在自定义 HINT 中指定 SQL_DELAY_CUTOFF 的值，当备库的 SQL_DELAY 值（MySQL 主备复制延迟）达到或超过 time 的值（单位秒）时，查询语句会被下发到主实例。

注意：

DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。

如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。

示例

指定主备复制延迟时间为 5 秒：

```
/!TDDL:SQL_DELAY_CUTOFF=5*/SELECT * FROM table_name;
```

在 SQL 语句指定了 SQL_DELAY_CUTOFF 的值为 5，当备库的 SQL_DELAY 值达到或超过5秒时，查询语句会下发到主实例执行。

配合其他自定义 HINT 使用：

```
/!TDDL:SLAVE AND SQL_DELAY_CUTOFF=5*/SELECT * FROM table_name;
```

备库延迟切断注释也可以配合其他注释使用，该 SQL 查询请求默认会被下发到只读实例，但是当出现主备复制延时达到或超过 5 秒时，会下发到主实例。

自定义 SQL 超时时间

在 DRDS 中，DRDS 节点与 RDS 的默认的 SQL 执行超时时间是 900 秒（可以调整），但是对于某些特定的慢 SQL，其执行时间可能超过了 900 秒 。针对这种慢 SQL，DRDS 提供了调整超时时间的自定义 HINT。通过这个自定义 HINT 可以任意调整 SQL 执行时长。

语法

DRDS 自定义 SQL 超时时间 HINT 的语法如下：

```
/!TDDL:SOCKET_TIMEOUT=time*/
```

其中，SOCKET_TIMEOUT的单位是毫秒。通过该 HINT 用户可以根据业务需要，自由调整 SQL 语句的超时时间。

注意：

DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。

如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释

形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。

示例

设置 SQL 超时时间为40秒：

```
/!TDDL:SOCKET_TIMEOUT=40000*/SELECT * FROM t_item;
```

注意：超时时间设置得越长，占用数据库资源的时间就会越长。如果同一时间长时间执行的 SQL 过多，可能消耗大量的数据库资源，从而导致无法正常使用数据库服务。所以，对于长时间执行的 SQL 语句，尽量对 SQL 语句进行优化。

指定分库执行 SQL

在使用 DRDS 的过程中，如果遇到某个 DRDS 不支持的 SQL 语句，可以通过 DRDS 提供的自定义 HINT，直接将 SQL 下发到一个或多个分库上去执行。此外如果需要单独查询某个分库或者已知分库的某个分表中的数据，也可以使用该自定义 HINT，直接将 SQL 语句下发到分库中执行。

语法

指定分库执行 SQL 自定义 HINT 有两种使用方式，即通过分片名指定 SQL 在分库上执行或者通过分库键值指定 SQL 在分库上执行。其中分片名是 DRDS 中分库的唯一标识，可以通过 SHOW NODE 控制指令得到。

注意：如果在目标表包含 Sequence 的 INSERT 语句上使用了指定分库的 HINT，那么 Sequence 将不生效。更多相关信息，请参考 Sequence 限制及注意事项。

通过分库名指定 SQL 在分库上执行

通过分库名指定 SQL 在分库上执行又分两种使用方式，分别是指定 SQL 在某个分库上执行和指定 SQL 在多个分库上执行。

指定 SQL 在某个分库上执行：

```
/!TDDL:NODE='node_name'*/
```

node_name 为分片名，通过这个 DRDS 自定义 HINT，就可以将 SQL 下发到node_name对应的分

库中执行。

指定 SQL 在多个分库上执行：

```
/!TDDL:NODE IN ('node_name'[,'node_name1','node_name2'])*/
```

使用 in 关键字指定多个分片名，将 SQL 下发到多个分库上执行，括号内分片名之间使用逗号分隔。

注意：使用该自定义 HINT 时，DRDS 会将 SQL 直接下发到分库上执行，所以在 SQL 语句中，表名必须是该分库中已经存在的表名。

通过分库键值指定 SQL 在分库上执行

```
/!TDDL:table_name.partition_key=value [and table_name1.partition_key=value1]*/
```

在这个 DRDS 自定义 HINT 中table_name为逻辑表名，该表是一张拆分表，partition_key是拆分键，value为指定的拆分键的值。在该自定义注释中，可以使用and关键字指定多个拆分表的拆分键。通过这个 DRDS 自定义 HINT，DRDS 会计算出 SQL 语句应该在哪些分库和分表上执行，进而将 SQL 语句下发到相应的分库。

注意：

DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。

如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。

示例

对于名为 drds_test 的 DRDS 数据库，SHOW NODE 的结果如下：

```
mysql> SHOW NODE\G
*************************** 1. row ******************
ID: 0
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS
MASTER_READ_COUNT: 212
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 2. row ******************
ID: 1
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0001_RDS
MASTER_READ_COUNT: 29
```

```
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 3. row ******************
ID: 2
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0002_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 4. row ******************
ID: 3
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0003_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 5. row ******************
ID: 4
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0004_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 6. row ******************
ID: 5
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0005_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 7. row ******************
ID: 6
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0006_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
*************************** 8. row ******************
ID: 7
NAME: DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0007_RDS
MASTER_READ_COUNT: 29
SLAVE_READ_COUNT: 0
MASTER_READ_PERCENT: 100%
SLAVE_READ_PERCENT: 0%
8 rows in set (0.02 sec)
```

可以看到每个分库都有 NAME 这个属性，这就是分库的分片名。每个分片名都唯一对应一个分库名，比如 DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0003_RDS这个分片名对应的分库名是 drds_test_vtla_0003。得到了分片名，就可以使用 DRDS 的自定义 HINT 指定分库执行 SQL 语句了。

指定 SQL 在第 0 个分库上执行：

```
/!TDDL:NODE='DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS'*/SELECT * FROM
table_name;
```

指定 SQL 在多个分库上执行：

```
/!TDDL:NODE
IN('DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS','DRDS_TEST_1473471355140LRPRDRD
S_TEST_VTLA_0006_RDS')*/SELECT * FROM table_name;
```

这条 SQL 语句将在 DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS，DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0006_RDS这两个分片上执行。

查看某个分库的执行计划：

```
/!TDDL:NODE='DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS'*/EXPLAIN SELECT * FROM
table_name;
```

这条 SQL 语句将会展示 SELECT 语句在分片 DRDS_TEST_1473471355140LRPRDRDS_TEST_VTLA_0000_RDS 中的执行计划。

通过键值指定 SQL 在分库上执行：

对于UPDATE语句，DRDS 不支持SET子句中的子查询，由于UPDATE语句在 DRDS 中必须指定拆分键，所以可以使用 DRDS 的自定义 HINT 将该语句直接下发到分库上执行。

比如有两张逻辑表，分别是 t1 和 t2，它们都是分库分表，建表语句如下：

```
  CREATE TABLE `t1` (
`id` bigint(20) NOT NULL,
`name` varchar(20) NOT NULL,
`val` varchar(20) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`name`)
tbpartitions 3

CREATE TABLE `t2` (
`id` bigint(20) NOT NULL,
`name` varchar(20) NOT NULL,
`val` varchar(20) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`name`)
tbpartitions 3
```

需要执行的语句是：

```
UPDATE t1 SET val=(SELECT val FROM t2 WHERE id=1) WHERE id=1;
```

这条语句直接在 DRDS 上执行会报不被支持的错误，但是可以给这条语句加上 DRDS 的自定义 HINT，再提交到 DRDS 执行。具体 SQL 语句如下：

```
/!TDDL:t1.id=1 and t2.id=1*/UPDATE t1 SET val=(SELECT val FROM t2 WHERE id=1) WHERE id=1;
```

这条语句会被下发到t1的id为 1 的分库上执行。通过 explain 命令可以看到执行这条 SQL 语句的执行计划：

```
  mysql> explain /!TDDL:t1.id=1 and t2.id=1*/UPDATE t1 SET val=(SELECT val FROM t2 WHERE id=1)
WHERE id=1\G
*************************** 1. row ***************************
GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
SQL: UPDATE `t1_2` AS `t1` SET `val` = (SELECT val FROM `t2_2` AS `t2` WHERE `id` = 1) WHERE `id` = 1
PARAMS: {}
*************************** 2. row ***************************
GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
SQL: UPDATE `t1_1` AS `t1` SET `val` = (SELECT val FROM `t2_1` AS `t2` WHERE `id` = 1) WHERE `id` = 1
PARAMS: {}
*************************** 3. row ***************************
GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
SQL: UPDATE `t1_0` AS `t1` SET `val` = (SELECT val FROM `t2_0` AS `t2` WHERE `id` = 1) WHERE `id` = 1
PARAMS: {}
3 rows in set (0.00 sec)
```

从explain命令的结果集可以看到，SQL 语句被改写成 3 条语句下发到分库上执行。还可以继续指定分表键值，将 SQL 执行范围缩小到一张分表：

```
  mysql> explain /!TDDL:t1.id=1 and t2.id=1 and t1.name='1'*/UPDATE t1 SET val=(SELECT val FROM t2
WHERE id=1) WHERE id=1\G
*************************** 1. row ***************************
GROUP_NAME: TEST_DRDS_1485327111630IXLWTEST_DRDS_IGHF_0001_RDS
SQL: UPDATE `t1_1` AS `t1` SET `val` = (SELECT val FROM `t2_1` AS `t2` WHERE `id` = 1) WHERE `id` = 1
PARAMS: {}
1 row in set (0.00 sec)
```

注意：使用该自定义注释需要保证两张表的分库和分表数量一致，否则 DRDS 计算出的两个键值对应的分库不一致，就会报错。

扫描全部分库分表

除了可以将 SQL 单独下发到一个或多个分库执行，DRDS 还提供了扫描全部分库与分表的自定义 HINT。通过这个自定义 HINT，您可以一次将 SQL 下发到每一个分库执行。比如通过这个自定义 HINT，可以查看某个分库上的所有分表。还可以通过这个自定义 HINT，查看某个逻辑表的每个分库的分表数据量。

语法

扫描全部分片的 DRDS 自定义 HINT 有两种使用方式，第一种方式是将 SQL 语句下发到每个分库执行，第二种方式是将 SQL 语句下发到每个分库上对某个逻辑表进行操作。

将 SQL 下发到全部分库上执行：

```
/!TDDL:SCAN*/
```

对某个逻辑表进行操作：

```
/!TDDL:SCAN='table_name'*/
```

其中table_name是 DRDS 数据库的某个逻辑表名。该自定义 HINT 是为分库分表提供的，请尽量确保table_name为分库分表。

注意：

DRDS 自定义 HINT 支持 /!TDDL:hint command*/ 和 /*TDDL:hint command*/ 两种格式。

如果使用 /*TDDL:hint command*/ 格式，在使用 MySQL 官方命令行客户端执行带有 DRDS 自定义 HINT 的 SQL 时，请在登录命令中加上 -c 参数。否则，由于 DRDS 自定义 HINT 是以 MySQL 注释形式使用的，该客户端会将注释语句删除后再发送到服务端执行，导致 DRDS 自定义 HINT 失效。具体请查看 MySQL 官方客户端命令。

示例

查看某个广播表每个分库中的数据量：

```
/!TDDL:SCAN*/SELECT COUNT(1) FROM table_name
```

SQL 语句中 table_name 是一个广播表，这条语句会将 SQL 语句下发到每个分库上执行，所以结果集会包含每个分库中 table_name 表的数据量。这条语句可以很方便的检查广播表数据是否正常。

只分库不分表的逻辑表扫描：

```
/!TDDL:SCAN*/SELECT COUNT(1) FROM table_name
```

这条语句将在每个分库中执行语句select count(1) from table_name，其中table_name为 DRDS 数据库中的逻辑表，在执行之前请确保每个分库中都有表名为table_name的分表（也就是table_name为一张只分库，不分表的逻辑表），否则会报找不到表的错误。

分库分表的逻辑表扫描：

```
/!TDDL:SCAN='table_name'*/SELECT COUNT(1) FROM table_name
```

DRDS 在执行这条语句时，首先会计算出逻辑表 table_name 的所有分库和分表，再生成对每个分库中，每个分表的 COUNT 语句。

查看所有分库的执行计划：

```
/!TDDL:SCAN='table_name'*/EXPLAIN SELECT * FROM table_name;
```

全局唯一数字序列 Sequence

DRDS Sequence 介绍

DRDS 全局唯一数字序列（64 位数字，对应 MySQL 中 Signed BIGINT 类型，以下简称为 Sequence）的主要目标是为了生成全局唯一和有序递增的数字序列，常用于主键列、唯一索引列等值的生成。

DRDS 中的 Sequence 主要有两类用法：

- 显式 Sequence，通过 Sequence DDL 语法创建和维护，可以独立使用；通过select seq.nextval;获取序列值，seq 是具体 Sequence 的名字；

  隐式 Sequence，在为主键定义 AUTO_INCREMENT 后，用于自动填充主键，由 DRDS 自动维护。

  注意：仅拆分表和广播表指定了 AUTO_INCREMENT 后，DRDS 才会创建隐式的 Sequence。非拆分表并不会，非拆分表的 AUTO_INCREMENT 的值由底层 RDS（MySQL）自己生成。

下文主要包含以下内容：

DRDS Sequence 类型与特性说明

- Group Sequence（GROUP，默认使用）
- Simple Sequence（SIMPLE）

不同类型 Sequence 使用场景及选择

DRDS Sequence 类型与特性说明

目前共支持 2 种 Sequence 类型：

类型（缩写）	全局唯一	连续	单调递增	同一连接内单调递增	非单点	数据类型	可读性
Group Sequence（GROUP）	是	否	否	是	是	所有整型	好
Simple Sequence（SIMPLE）	是	是	是	是	否	所有整型	好

概念解释：

- **连续：**如果本次取值为 n，下一次取值一定是 n + 1，则是连续的；如果下一次取值不能保证为 n + 1，则是非连续的；
- **单调递增：**如果本次取值为 n，下一次取值一定是一个比 n 大的数，则是单调递增的；
- **单点：**存在单点故障风险；

Group Sequence（GROUP，默认使用）

特性

全局唯一的 Sequence，产生的值是自然数序列，但是不保证连续和单调递增。如果未指定 Sequence 类型，DRDS 默认使用 Group Sequence。

- 优点：**全局唯一、不会产生单点问题、性能非常好。**
- 缺点：**产生的序列不连续、可能会有跳跃段；不会严格从起始值开始取值；不能循环。**

实现机理

采用多个节点产生值来保证高可用，每次取出一段值，如果该段值没有取完（比如连接断掉等情形），就会产生跳跃段。

Simple Sequence（SIMPLE）

特性

仅 Simple Sequence 支持自定义步长、最大值和循环/非循环利用。

- 优点：**全局唯一、连续、单调递增。**
- 缺点：**单点，性能较差，存在瓶颈，需要谨慎使用。**

实现机理

每产生一个值都要进行一次持久化操作。

使用场景

这两种 Sequence 都保证全局唯一，均可以应用在**主键列**和**唯一索引列**。

- 大部分场景下建议选用 **Group Sequence**；
- 如果业务强依赖连续的 Sequence 值，此时只能使用 Simple Sequence（注意 Simple Sequence 的性能问题）；

以创建一个起始值是 100000，步长为 1 的 Sequence 为例。

如果采用 **Simple Sequence**，则会严格产生100000、100001、100002、100003、100004、.....、200000、200001、200002、200003、.....这样的序列（全局唯一、连续、单调递增）。Simple Sequence 会保证持久化，即使发生单点问题，服务重启后依然会在断点继续产生 Sequence 值，中间不会产生跳跃段。Simple Sequence 的机理是每产生一个值都要进行一次持久化操作，因此性能并不是很好。

如果采用 **Group Sequence**，产生的序列有可能是200001、200002、200003、200004、100001、100002、100003、.....这样的序列。

注意：

- Group Sequence 起始值并不会严格从设定的参数（本例中是100000）开始，但是保证比该参数大。本例中是从 200001 开始取值的。
- Group Sequence 保证全局唯一，但是会有跳跃段。比如 Group Sequence 的某个节点挂掉，或者某个连接只取了一部分值，然后该连接被关闭了，都会产生跳跃段。该例中 200004 和 100001 之间产生了跳跃段。

Sequence 显式用法

本文介绍如何用 DDL 语句创建、修改、删除、查询 Sequence， 以及如何获取显式 Sequence 的值。

- 创建 Sequence
- 修改 Sequence
- 删除 Sequence
- 查询 Sequence
- 获取显式 Sequence 值
- 批量获取 Sequence 值

创建 Sequence

语法：

```
CREATE [ GROUP | SIMPLE ] SEQUENCE <name>
[ START WITH <numeric value> ] [ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ] [ CYCLE | NOCYCLE ]
```

参数说明：

参数	解释说明	适用范围
START WITH	Sequence 的起始值，若未指定，则默认值为1。	Simple Sequence、Group Sequence
INCREMENT BY	Sequence 每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1。	Simple Sequence
MAXVALUE	Sequence 允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值。	Simple Sequence
CYCLE 或 NOCYCLE	两者只能选择其一，代表当 Sequence 值增长到最大值后，是否允许继续循环（即仍从 START WITH 开始）使用 Sequence 值。若未指定，则默认值为 NOCYCLE。	Simple Sequence

注意：

- 如果未指定类型关键字，则默认类型为 Group Sequence；
- INCREMENT BY、MAXVALUE、CYCLE 或 NOCYCLE这三个参数对于 Group Sequence 是无意义的；
- Group Sequence 是非连续的。START WITH 参数对于 Group Sequence 仅具有指导意义，**Group Sequence 不会严格按照该参数作为起始值，但是保证起始值比该参数大。**

以下为创建不同类型 Sequence 的几个示例。

示例一：创建一个 Group Sequence。

方法一：

```
mysql> CREATE SEQUENCE seq1;
Query OK, 1 row affected (0.27 sec)
```

方法二：

```
mysql> CREATE GROUP SEQUENCE seq1;
Query OK, 1 row affected (0.27 sec)
```

示例二： 创建一个 Simple Sequence，起始值是 1000，步长为 2，最大值为 99999999999，增长到最大值后不继续循环。

```
mysql> CREATE SIMPLE SEQUENCE seq2 START WITH 1000 INCREMENT BY 2 MAXVALUE 99999999999 NOCYCLE;
Query OK, 1 row affected (0.03 sec)
```

修改 Sequence

DRDS 支持对 Sequence 的以下几个方面进行修改：

- 修改 Simple Sequence 的参数：起始值、步长、最大值、循环或非循环；
- 修改 Group Sequence 的参数：起始值；
- 不同类型 Sequence 间的转换。

语法：

```
ALTER SEQUENCE <name> [ CHANGE TO GROUP | SIMPLE ]
START WITH <numeric value> [ INCREMENT BY <numeric value> ]
[ MAXVALUE <numeric value> ] [ CYCLE | NOCYCLE ]
```

参数说明：

参数	解释说明	适用范围
START WITH	Sequence 的起始值，若未指定，则默认值为1。	Simple Sequence、Group Sequence
INCREMENT BY	Sequence 每次增长时的增量值（或称为间隔值或步长），若未指定，则默认值为1。	Simple Sequence
MAXVALUE	Sequence 允许的最大值，若未指定，则默认值为有符号长整型（Signed BIGINT）的最大值。	Simple Sequence

CYCLE 或 NOCYCLE	两者只能选择其一，代表当 Sequence 值增长到最大值后，是否允许继续循环（即仍从 START WITH 开始）使用 Sequence 值。若未指定，则默认值为 NOCYCLE。	Simple Sequence

注意：

- Group Sequence 是非连续的。START WITH 参数对于 Group Sequence 仅具有指导意义，**Group Sequence 不会严格按照该参数作为起始值，但是保证起始值比该参数大。**
- 对于 Simple Sequence，如果修改 Sequence 时指定了 START WITH，则会立即生效，下次取 Sequence 值时会从新的 START WITH 值开始。比如原先 Sequene 增长到 100 了，这时把 START WITH 值改成了 200，那么下一次获取的 Sequence 值就是从 200 开始的。
- 修改 START WITH 的参数值，需要仔细评估已经产生的 Sequence 值，以及生成新 Sequence 值的速度，防止产生冲突。如非必要，请谨慎修改 START WITH 参数值。

示例：

将 Simple Sequence seq2 的起始值改为 3000，步长改为 5，最大值改为 1000000，增长到最大值后改为继续循环。

```
mysql> ALTER SEQUENCE seq2 START WITH 3000 INCREMENT BY 5 MAXVALUE 1000000 CYCLE;
Query OK, 1 row affected (0.01 sec)
```

不同类型 Sequence 间的转换

- 通过ALTER SEQUENCE的CHANGE TO <sequence_type>子句实现；
- ALTER SEQUENCE 如果指定了 CHANGE TO子句，则强制必须加上 START WITH 参数，避免忘记指定起始值而造成取值时得到重复值；若没有 CHANGE TO（可选参数），则不强制。

示例：

将 Group Sequence 转换为 Simple Sequence。

```
mysql> ALTER SEQUENCE seq1 CHANGE TO SIMPLE START WITH 1000000;
Query OK, 1 row affected (0.02 sec)
```

删除 Sequence

语法：

```
DROP SEQUENCE <name>
```

示例：

```
mysql> DROP SEQUENCE seq3;
Query OK, 1 row affected (0.02 sec)
```

查询 Sequence

语法：

```
SHOW SEQUENCES
```

示例：

结果集中的 TYPE 列，显示的是 Sequence 类型的缩写。

```
mysql> SHOW SEQUENCES;
+------------+--------------------+-------------+------------+--------------------+-------+--------+
| NAME | VALUE | INCREMENT_BY | START_WITH | MAX_VALUE | CYCLE | TYPE |
+------------+--------------------+-------------+------------+--------------------+-------+--------+
| AUTO_SEQ_1 | 91820513 | 1 | 91820200 | 9223372036854775807 | N | SIMPLE |
| AUTO_SEQ_4 | 91820200 | 2 | 1000 | 9223372036854775807 | Y | SIMPLE |
| AUTO_SEQ_2 | 100000 | N/A | N/A | N/A | GROUP |
| AUTO_SEQ_3 | 200000 | N/A | N/A | N/A | GROUP |
+------------+--------------------+-------------+------------+--------------------+-------+--------+
4 rows in set (0.01 sec)
```

获取显式 Sequence 值

语法:

```
< sequence name >.NEXTVAL
```

示例：

```
SELECT sample_seq.nextVal FROM dual;
+--------------------+
| SAMPLE_SEQ.NEXTVAL |
+--------------------+
| 101001 |
+--------------------+
1 row in set (0.04 sec)
```

或者可以把这个 SAMPLE_SEQ.nextVal 当做一个值写入 SQL 中：

```
mysql> INSERT INTO some_users (name,address,gmt_create,gmt_modified,intro) VALUES
('sun',SAMPLE_SEQ.nextVal,now(),now(),'aa');
Query OK, 1 row affected (0.01 sec)
```

注意：如果建表时已经指定了 AUTO_INCREMENT 参数，insert 时不需要指定自增列，让 DRDS 自动维护。

批量获取 Sequence 值

语法:

```
SELECT < sequence name >.NEXTVAL FROM DUAL WHERE COUNT = < numeric value >
```

示例：

```
SELECT sample_seq.nextVal FROM dual WHERE count = 10;
+--------------------+
| SAMPLE_SEQ.NEXTVAL |
+--------------------+
| 101002 |
| 101003 |
| 101004 |
| 101005 |
| 101006|
| 101007 |
| 101008 |
| 101009 |
| 101010 |
| 101011 |
+--------------------+
10 row in set (0.04 sec)
```

Sequence 隐式用法

在为主键定义 AUTO_INCREMENT 后，Sequence 可以用于自动填充主键，由 DRDS 自动维护。

CREATE TABLE

扩展标准建表语法，增加了自增列的 Sequence 类型，如果未指定类型关键字，则默认类型为 GROUP。
DRDS 自动创建的跟表相关联的 Sequence 名称，都是以 AUTO_SEQ_ 为前缀，后面加上表名。

```
CREATE TABLE <name> (
<column> ... AUTO_INCREMENT [ BY GROUP | SIMPLE ],
<column definition>,
...
) ... AUTO_INCREMENT=<start value>
```

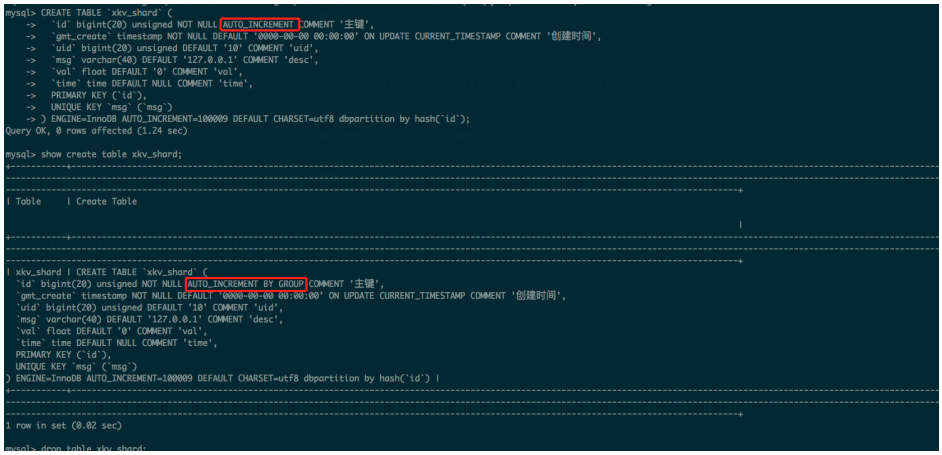
SHOW CREATE TABLE

当表为拆分表或者广播表时，显示自增列 Sequence 的类型。

```
SHOW CREATE TABLE <name>
```

示例：

建表时指定 AUTO_INCREMENT，但是没有指定 Sequence 类型关键字，则默认使用 Group Sequence。



ALTER TABLE

暂不支持通过 ALTER TABLE 来修改对应 Sequence 的类型，但可修改起始值。如果想要修改表中带的隐式 Sequence 的类型，需要通过 SHOW SEQUENCES 指令查找出 Sequence 的具体名称和类型，然后再用 ALTER SEQUENCE 指令去修改。

```
ALTER TABLE <name> ... AUTO_INCREMENT=<start value>
```

注意：使用 DRDS Sequence 后，请谨慎修改 AUTO_INCREMENT 的起始值（仔细评估已经产生的 Sequence 值，以及生成新 Sequence 值的速度，防止产生冲突）。

Sequence 限制及注意事项

限制与注意事项

- 转换 Sequence 类型时，必须指定 START WITH 起始值；

- 在 DRDS 非拆分模式库（即后端仅关联一个已有的 RDS 物理库）、或拆分模式库中仅有单表（即所有表都是单库单表，且无广播表）的场景下执行 INSERT 时， DRDS 会自动优化并直接下推语句，绕过优化器中分配 Sequence 值的部分。此时 INSERT INTO ... VALUES （seq.nextval, ...)这种用法不支持，建议使用后端 RDS/MySQL 自增列机制代替。
- 如果将指定分库的 Hint 用在 INSERT 语句上，比如 INSERT INTO ... VALUES ... 或 INSERT INTO ... SELECT ...，且目标表使用了 Sequence，则 DRDS 会绕过优化器直接下推语句，使 Sequence 不生效。目标表最终会使用后端 RDS/MySQL 表中的自增机制生成 id。
- 必须对同一个表采用一种统一的方式分配自增 id：或者依赖于 DRDS Sequence，或者依赖于后端 RDS/MySQL 表的自增列。避免两种机制混用，否则可能会造成 id 冲突（产生重复 id）的情况，且难于排查。

如何处理主键冲突

比如直接在 RDS 中写入了数据，而对应的主键值不是 DRDS 生成的 Sequence 值，那么后续让 DRDS 自动生成主键写入数据库，可能会和这些数据发生主键冲突，可以通过以下步骤解决问题：

通过 DRDS 指定 SQL 来查看当前已有 Sequence。**AUTO_SEQ_** 开头的 Sequence 是隐式 Sequence（创建表时加上 AUTO_INCREMENT 参数的表产生的 Sequence）：

```
mysql> SHOW SEQUENCES;
+--------------------+-------+-------------+-----------+-----------+-------+-------+
| NAME | VALUE | INCREMENT_BY | START_WITH | MAX_VALUE | CYCLE | TYPE |
+--------------------+-------+-------------+-----------+-----------+-------+-------+
| AUTO_SEQ_xkv_t_item | 0 | N/A | N/A | N/A | N/A | GROUP |
| AUTO_SEQ_xkv_shard | 0 | N/A | N/A | N/A | N/A | GROUP |
+--------------------+-------+-------------+-----------+-----------+-------+-------+
2 rows in set (0.04 sec)
```

比如 xkv_t_item 表有冲突，并且 xkv_t_item 表主键是 ID，那么从 DRDS 获取这个表最大主键值：

```
mysql> SELECT MAX(id) FROM xkv_t_item;
+-----------+
| max(id) |
+-----------+
| 8231 |
+-----------+
1 row in set (0.01 sec)
```

更新 DRDS Sequence 表中对应的值，这里更新成比 8231 要大的值，比如 9000，更新完成后，后续插入语句生成的自增主键将不再报错：

```
mysql> ALTER SEQUENCE AUTO_SEQ_xkv_t_item START WITH 9000;
Query OK, 1 row affected (0.01 sec)
```

数据导入导出

数据导入导出方式概述

DRDS 支持多种方式进行数据迁移，包括 MySQL 官方的 mysqldump、source 命令，阿里云的数据传输服务 DTS，数据集成（Data Integration），以及通过编写代码的方式进行数据迁移。

以下是几种方式的应用场景及对比：

方式	导入 DRDS	从 DRDS 导出	数据量	全量迁移	增量迁移	操作说明
DTS	RDS、MySQL 至 DRDS 数据迁移	不支持	大			说明
MySQL 官方命令	从 SQL 文本文件导入到 DRDS（source 命令）	从 DRDS 导出 SQL 文本文件 (mysqldump 命令)	小（低于 2000万）			说明
编程方式	任意数据库、文本文件导入到 DRDS	DRDS 数据导出到任意数据库、文本文件	大			说明
数据集成	MaxCompute（原 ODPS）导入到 DRDS	DRDS 数据导出到 MaxCompute（原 ODPS）	大			说明

使用 mysqldump 导入导出数据

DRDS 支持 MySQL 官方数据导出工具 mysqldump。本文围绕 DRDS 数据导入导出的几种常见场景对操作步

骤和注意事项进行说明。mysqldump 命令的详细说明请参考 MySQL 官方文档。

说明：mysqldump 适合小数据量(低于1000万)的离线导入导出。如果需要完成更大数据量或者实时的数据迁移任务，请参考阿里云提供的数据传输服务。

场景一：从 MySQL 导入到 DRDS

从 MySQL 导入数据到 DRDS，请按照以下步骤进行操作。

1.从 MySQL 中导出数据到文本文件。

输入以下命令，从 MySQL 中导出表结构和数据。假设导出文件为 dump.sql。

```
mysqldump -h ip -P port -u user -ppassword --default-character-set=char-set --net_buffer_length=10240 --no-create-db --skip-add-locks  --skip-tz-utc --set-charset  [--hex-blob] [--no-data] database [table1 table2 table3...] > dump.sql
```

参数说明如下，请根据实际情况输入：

参数名	说明	是否必选
ip	DRDS 实例的 IP。	
port	DRDS 实例的端口。	
user	DRDS 的用户名。	
password	DRDS 的密码，注意前面有个 -p，之间没有空格。	
char-set	指定的编码。	
--hex-blob	使用十六进制格式导出二进制字符串字段。如果有二进制数据就必须使用本选项。影响的字段类型包括 BINARY、VARBINARY、BLOB。	
--no-data	不导出数据。	
table	指定导出某个表。默认导出该数据库所有的表。	

2.修改建表语句。

从 MySQL 导出的数据文件包含每个表的建表语句。如果直接在在 DRDS 上执行这些建表语句，会在 DRDS 上建立一个单表。如果要对某个表进行分库分表，那么需要手工对建表语句进行修改，DRDS 建表语句的语法请参考 DDL 语句。

3.导入数据文件到 DRDS。可以通过两种方式导入数据文件到 DRDS。

注意：下面两个命令中 default-character-set 要设置成实际的数据编码。如果是 Windows 平台，source 命令指定的文件路径需要对分隔符转义。

- 通过 mysql -h ip -P port -u user --default-character-set=char-set 命令登录目标 DRDS，执行 source /yourpath/dump.sql 命令将数据导入到目标 DRDS。
- 直接通过 mysql -h ip -P port -u user --default-character-set=char-set< /yourpath/dump.sql 命令将数据导入到目标 DRDS。

第一种方式会把所有的步骤回显到屏幕上，速度略慢，但是可以观察导入过程。

注意： 导入的时候，由于某些 DRDS 和 MySQL 实现上的不同，可能会报错，错误信息类似：ERROR 1231 (HY000): [a29ef6461c00000][10.117.207.130:3306][**]Variable @saved_cs_client can't be set to the value of @@character_set_client。此类错误信息并不影响导入数据的正确性。

场景二：从一个 DRDS 导入到另一个 DRDS

假设您之前有一个测试环境的 DRDS，测试完毕以后，需要把测试过程中的一些表结构和数据导入到生产环境中的 DRDS 中，那么可以按照以下步骤进行操作。

从源 DRDS 中导出数据到文本文件。请参考场景一第 1 步。

导入数据文件到 DRDS。请参考场景一第 3 步。

手工创建 Sequence 对象。

mysqldump 并不会导出 DRDS 中的 Sequence 对象，所以如果在源 DRDS 中使用了 Sequence 对象，并且需要在目标 DRDS 中继续使用相同的 Sequence 对象，则需要手工在目标 DRDS 中创建同名的 Sequence 的对象。具体步骤如下：

a. 在源 DRDS 上执行 SHOW SEQUENCES，获取当前 DRDS 中的 Sequence 对象的状态。

b. 在目标 DRDS 数据库上通过 CREATE SEQUENCE 命令创建新的 Sequence 对象。

Sequence 命令详情请参考**全局唯一数字序列** 。

场景三：从 DRDS 导出数据到 MySQL

从 DRDS 导出数据到 MySQL，和在 DRDS 之间相互导入数据的过程类似，也分为以下几个步骤。

从源 DRDS 中导出表结构和数据。请参考场景一第 1 步。

手工修改拆分表的 DDL 语句。

DRDS 中拆分表的**建表语句**和 MySQL 并不兼容。为了后继导入到 MySQL 中，需手工修改导出的 SQL 文件，删除以下关键字：

- DBPARTITION BY hash(partition_key):
- TBPARTITION BY hash(partition_key):
- TBPARTITIONS N
- BROADCAST

例如一个拆分表语句导出如下：

```
CREATE TABLE multi_db_single_tbl
(id int,
name varchar(30),
primary key(id)) dbpartition by hash(id);
```

需修改成以下语句：

```
CREATE TABLE multi_db_single_tbl
(id int,
name varchar(30),
primary key(id));
```

3. 导入修改以后的文件。请参考场景一第 3 步。

使用程序进行大数据导入

本文介绍如何通过编写代码的方式，离线导入大数据量到 DRDS 数据库。

假设当前数据库有一个表需要导入到 DRDS 数据库中，数据量大致为814万，表结构如下。

```
CREATE TABLE `post` (
`postingType` int NOT NULL,
`id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
`acceptedAnswer` bigint(20) DEFAULT NULL,
`parentId` bigint(20) DEFAULT NULL,
`score` int DEFAULT NULL
`tags` varchar(128) DEFAULT NULL,
PRIMARY KEY (`id`)
);
```

导出源数据

数据库之间大数据量的迁移，建议把原始数据导出成一个文本文件，然后通过程序或者命令的方式导入到目标数据库。

对于上一节的 **post** 表，可以通过 SELECT INTO 语法将数据从 MySQL 导出到一个名为 stackoverflow.csv 的文件中。在 MySQL 客户端执行以下命令：

```
SELECT postingType,id,acceptedAnswer,parentId,score,tags
INTO OUTFILE '/tmp/stackoverflow.csv'
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM test_table;
```

在 DRDS 数据库上建表

由于导出的数据文件不包括表结构，所以需要手工在 DRDS 目标数据库上建立表，并且根据实际情况设置拆分键。

例如以下是按照 **id** 对 **post 表**进行分库。

```
CREATE TABLE `post` (
`postingType` int NOT NULL,
`id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
`acceptedAnswer` bigint(20) DEFAULT NULL,
`parentId` bigint(20) DEFAULT NULL,
`score` int DEFAULT NULL,
`tags` varchar(128) DEFAULT NULL,
PRIMARY KEY (`id`)
) DBPARTITION BY hash(id) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

导入数据到 DRDS 数据库

导出数据文件以后，可以通过代码的方式读取文件内容，然后导入到 DRDS 数据库中。为了提高效率，建议通过批量插入的方式。

以下是用 Java 写的一个 Demo。

测试场景：插入8143801条数据，耗时916秒，TPS 在9000左右。

测试客户端配置：i5、8G、SSD。

测试 DRDS 配置：4C4G。

```
public static void main(String[] args) throws IOException, URISyntaxException, ClassNotFoundException,
SQLException {

URL url = Main.class.getClassLoader().getResource("stackoverflow.csv");

File dataFile = new File(url.toURI());

String sql = "insert into post(postingType,id,acceptedAnswer,parentId,score,tags) values(?,?,?,?,?,?)";
```

```
int batchSize = 10000;

try (

Connection connection = getConnection("XXXXX.drds.aliyuncs.com", 3306, "XXXXX",
"XXXX",
"XXXX");
BufferedReader br = new BufferedReader(new FileReader(dataFile))) {
String line;
PreparedStatement st = connection.prepareStatement(sql);
long startTime = System.currentTimeMillis();
int batchCount = 0;
while ((line = br.readLine()) != null) {
String[] data = line.split(",");
st.setInt(1, Integer.valueOf(data[0]));
st.setInt(2, Integer.valueOf(data[1]));

st.setObject(3, "".equals(data[2]) ? null : Integer.valueOf(data[2]));
st.setObject(4, "".equals(data[3]) ? null : Integer.valueOf(data[3]));
st.setObject(5, "".equals(data[4]) ? null : Integer.valueOf(data[4]));
if (data.length >= 6) {
st.setObject(6, data[5]);
}
st.addBatch();
if (++batchCount % batchSize == 0) {
st.executeBatch();
System.out.println(String.format("insert %d record", batchCount));
}
}
if (batchCount % batchSize != 0) {
st.executeBatch();
}
long cost = System.currentTimeMillis() - startTime;

System.out.println(String.format("Take %d second，insert %d record, tps %d", cost/1000,batchCount,
batchCount/(cost/1000) ));

}

}

/**
* 获取数据库连接
*
* @param host 数据库地址
* @param port 端口
* @param database 数据库名称
* @param username 用户名
* @param password 密码
* @return
* @throws ClassNotFoundException
* @throws SQLException
*/
private static Connection getConnection(String host, int port, String database, String username, String password)
throws ClassNotFoundException, SQLException {
```

```
Class.forName("com.mysql.jdbc.Driver");
String url = String.format(
"jdbc:mysql://%s:%d/%s?autoReconnect=true&socketTimeout=600000&rewriteBatchedStatements=true", host,
port,
database);
Connection con = DriverManager.getConnection(url, username, password);
return con;
}
```

通过数据集成导入导出数据

本文介绍如何通过数据集成（Data Integration）在 DRDS 中进行数据导入和导出。

数据集成是阿里巴巴集团提供的数据同步平台。该平台具备可跨异构数据存储系统、可靠、安全、低成本、可弹性扩展等特点，可为20多种数据源提供不同网络环境下的离线(全量/增量)数据进出通道。详细的数据源类型列表请参考支持数据源类型。

使用数据集成，您可以在 DRDS 完成以下数据同步任务：

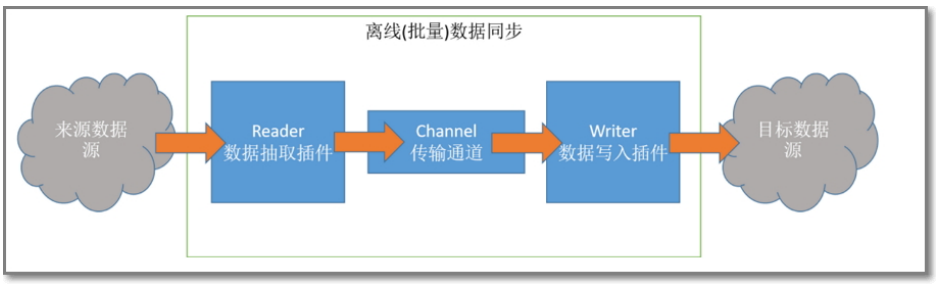
- 将 DRDS 的数据同步到到其他的数据源里，并将数据进行相应的处理；
- 将处理好的其他数据源数据同步到 DRDS。

本文包含以下内容：

流程概述

数据同步流程主要包含以下几个步骤：

- 第一步：数据源端新建表
- 第二步：添加数据源
- 第三步：向导模式或脚本模式配置同步任务
- 第四步：运行同步任务，检查目标端的数据质量



准备工作

使用数据集成在 DRDS 进行数据导入导出之前，请参考**开通阿里云主账号**和**准备 RAM 子账号**两个文档，完成以下准备工作：

1. 开通阿里云官网实名认证账号，创建好账号的访问秘钥，即 AccessKeys。
2. 开通 MaxCompute，这样会自动产生一个默认的 ODPS 的数据源，并使用主账号登录大数据开发套件。
3. 创建项目。您可以在项目中协作完成工作流，共同维护数据和任务等，因此使用大数据开发套件之前需要先创建一个项目。
4. 如果想通过子账号创建数据集成任务，可以赋予其相应的权限。

新添加数据源

下面以添加 DRDS 的数据源为例。

注意：只有项目管理员角色才能够新建数据源，其他角色的成员仅能查看数据源。

以项目管理员身份登录**数加管理控制台**。

在**项目列表**中对应项目的操作栏单击**进入工作区**。

进入顶部菜单栏中的**数据集成**页面，单击左侧导航栏中的**数据源**。

点击右上角的**新增数据源**，如下图所示：

在新增数据源弹出框中填写相关配置项，如下图所示：

针对 DRDS 数据源配置项的具体说明如下：

数据源名称：由英文字母、数字、下划线组成且需以字符或下划线开头，长度不超过 60 个字符 。

数据源描述：对数据源进行简单描述，不得超过 80 个字符 。

数据源类型：当前选择的数据源类型 DRDS。

网络类型：当前选择的网络类型。

JDBCUrl：JDBC 连接信息，格式为 jdbc://mysql://serverIP:Port/database 。

用户名/密码：对应的用户名和密码。

完成上述信息项的配置后，单击**测试连通性**。

测试连通性通过后，单击**确定**。

其他的数据源的配置请参考**数据源配置**。

通过数据集成导入数据

下文以通过数据集成的向导模式将 MaxCompute (原 ODPS)数据同步到 DRDS 为例。

在数据集成页面，新建同步任务。

- **向导模式**：向导模式是可视化界面配置同步任务， 一共涉及五步选择来源，选择目标，字段映射，通道控制，预览保存五个步骤。在每个不同的数据源之间，这几步的界面可能有不同的内容。向导模式可以转换成脚本模式。
- **脚本模式**：进入脚本界面你可以选择相应的模板，此模板包含了同步任务的主要参数，将相关的信息填写完整，但是脚本模式不能转化成向导模式。

选择数据来源。

选择 MaxCompute 数据源及源头表 mytest，数据浏览默认是收起的，选择后单击**下一步**：

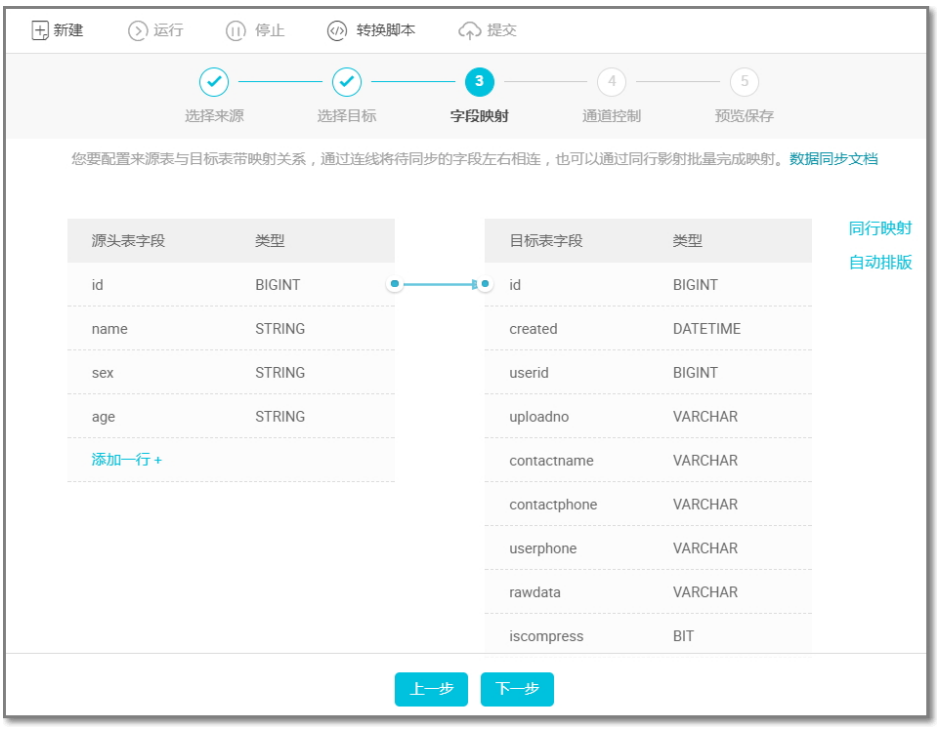
选择目标。

选择 DRDS 数据源及目标表 contact_infos，选择后单击**下一步**：

- preSql：执行数据同步任务之前率先执行的 SQL 语句。目前向导模式只允许执行一条 SQL 语句，脚本模式可以支持多条 SQL 语句，例如清除旧数据。
- postSql：执行数据同步任务之后执行的 SQL 语句。目前向导模式只允许执行一条 SQL 语句，脚本模式可以支持多条 SQL 语句，例如加上某一个时间戳。

选择字段的映射关系。

左侧“源头表字段”和右侧“目标表字段”为一一对应的关系，如下图所示。



在通道控制页面单击**下一步**，配置作业速率上限和脏数据检查规则。

- 作业速率上限：是指数据同步作业可能达到的最高速率，其最终实际速率受网络环境、数据库配置等的影响。
- 作业并发数：作业速率上限=作业并发数 * 单并发的传输速率。

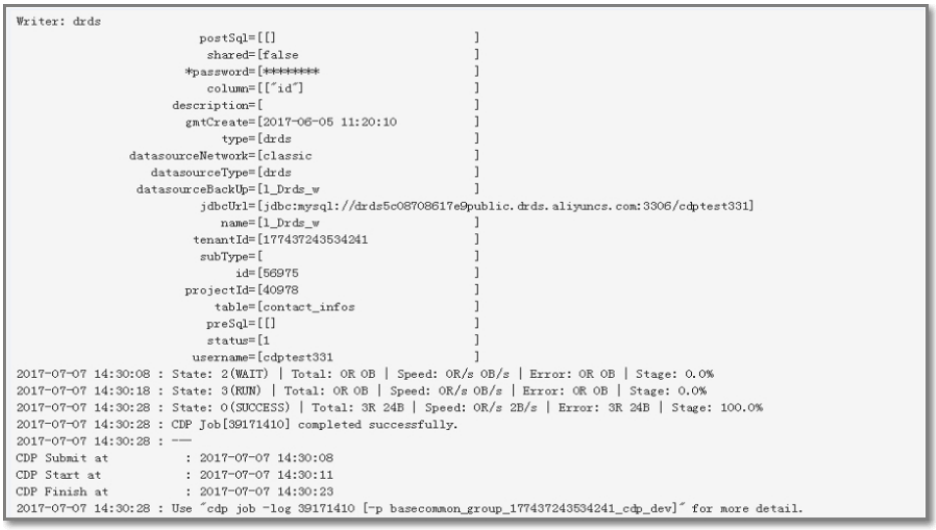
当作业速率上限已选定的情况下，可以根据以下原则选择并发数：

- 如果你的数据源是线上的业务库，建议您不要将并发数设置过大，以防对线上库造成影响；
- 如果您对数据同步速率特别在意，建议您选择最大作业速率上限和较大的作业并发数。

完成以上配置后，上下滚动鼠标可查看任务配置，确定无误后单击**保存**。

单击**运行任务**，直接运行同步任务结果。

您可以将同步任务提交到调度系统中，调度系统会按照配置属性从第二天开始自动定时执行。相关调度的配置请参考文档调度配置介绍。



脚本模式配置同步任务

```
{
  "type": "job",
  "version": "1.0",
  "configuration": {
    "reader": {
```

```
    "plugin": "odps",
    "parameter": {
      "datasource": "lzz_odps",//数据源的名称，建议都添加数据源后进行同步
      "table": "mytest",//数据来源的表名
      "partition": "",//分区信息
      "column": [
        "id"
      ]
    }
  },
  "writer": {
    "plugin": "drds",
    "parameter": {
      "datasource": "l_Drds_w",//数据源的名称，建议都添加数据源后进行同步
      "table": "contact_infos",//目的表名
      "preSql": [],//导入前准备语句
      "postSql": [],//导入后准备语句
      "column": [
        "id"
      ]
    }
  },
  "setting": {
    "speed": {
      "mbps": "1",//一个并发的速率上线是1MB/S
      "concurrent": "1"//并发的数目
    }
  }
 }
}
```

通过数据集成导出数据

下文以通过向导模式将 DRDS 数据同步到 MaxCompute 为例。

在数据集成页面，新建同步任务。

选择数据来源。

选择 DRDS 数据源及源头表 bit_type_test。数据浏览默认是收起的，选择后单击**下一步**，如下图所示：

- **过滤条件**：筛选条件，DrdsReader 根据指定的 column、table、where 条件拼接 SQL，并根据这个 SQL 进行数据抽取 。例如在做测试时，可以将 where 条件指定实际业务场景，往往会选择当天的数据进行同步，可以将 where 条件指定为 STRTODATE（‘${bdp.system.bizdate}’，‘%Y%m%d’）<= taday AND taday < DATEADD(STRTODATE（‘${bdp.system.bizdate}’，‘%Y%m%d’), interval 1 day) 。
- **切分键**：您可以将源数据表中某一列作为切分键，切分之后可进行并发数据同步。目前仅支持整型字段；建议使用主键或有索引的列作为切分键。

选择 MaxCompute 数据源及目标表 mytest，选择后单击**下一步**。

单击**下一步**，选择字段的映射关系。

左侧“源头表字段”和右侧“目标表字段”为一一对应的关系：

您也可以单击“添加一行”增加映射关系：

- 可以输入常量，输入的值需要使用英文单引号包括，如'abc'、'123'等；
- 可以配合调度参数使用，如'${bdp.system.bizdate}'等；
- 可以输入关系数据库支持的函数，如 now()、count(1)等；
- 如果您输入的值无法解析，则类型显示为'-'。

在通道控制页面单击**下一步**，配置作业速率上限和脏数据检查规则。

- 作业速率上限：是指数据同步作业可能达到的最高速率，其最终实际速率受网络环境、数据库配置等的影响。
- 作业并发数：作业速率上限=作业并发数 * 单并发的传输速率。

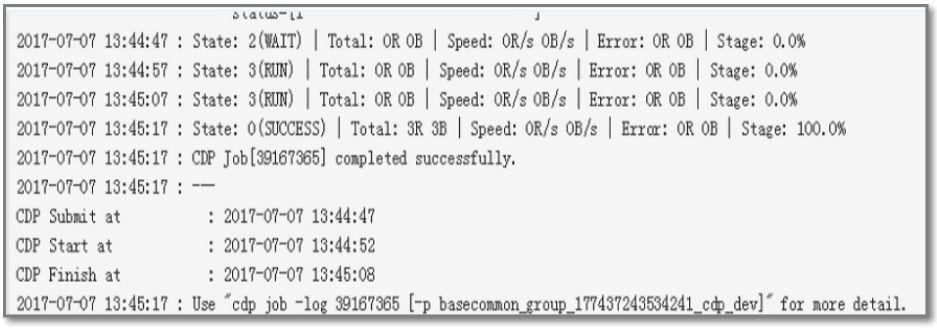
当作业速率上限已选定的情况下，可以按以下原则选择并发数：

- 如果你的数据源是线上的业务库，建议您不要将并发数设置过大，以防对线上库造成影响；
- 如果您对数据同步速率特别在意，建议您选择最大作业速率上限和较大的作业并发数。

完成以上配置后，上下滚动鼠标可查看任务配置。确认无误后单击**保存**。

单击**运行任务**直接运行同步任务结果。

您也可以将同步任务提交到调度系统中，调度系统会按照配置属性从第二天开始自动定时执行。相关调度的配置请参考文档**调度配置介绍**。



脚本模式配置同步任务

```
{
  "type": "job",
  "version": "1.0",
  "configuration": {
    "reader": {
      "plugin": "drds",
      "parameter": {
        "datasource": "l_Drds_w",//数据源的名称，建议都添加数据源后进行同步
        "table": "bit_type_test",/数据来源的表名
        "where": "",
        "splitPk": "col2",//切分键
        "column": [
```

```
        "idbit"
       ]
      }
    },
    "writer": {
      "plugin": "odps",
      "parameter": {
        "datasource": "lzz_odps",//数据源的名称，建议都添加数据源后进行同步
        "table": "mytest",
        "truncate": true,
        "partition": "",//分区信息
        "column": [
          "id"
        ]
      }
    },
    "setting": {
      "speed": {
        "mbps": "1",//作业速率上限
        "concurrent": "1"//并发数
      },
      "errorLimit": {
        "record": "234"//错误记录数
      }
    }
  }
}
```

其他同步任务配置详细信息请参考下面的文档：

- 各数据源 reader 的配置
- 各数据源 writer 的配置

数据运维

高危类 SQL 自动保护

为了防止误操作，DRDS 默认禁止全表删除和全表更新的操作。

下列语句默认会被禁止：

1. DELETE 语句不带 WHERE 条件或者 LIMIT 条件；
2. UPDATE 语句不带 WHERE 条件或者 LIMIT 条件。

如果确实需要执行这类操作，可以通过 HINT 来临时跳过这个限制
：HINT：/!TDDL:FORBID_EXECUTE_DML_ALL=false*/

示例

执行全表删除默认会被拦截：

```
mysql> delete from tt;
ERR-CODE: [TDDL-4620][ERR_FORBID_EXECUTE_DML_ALL] Forbid execute DELETE ALL or UPDATE ALL
sql. More: [http://middleware.alibaba-inc.com/faq/faqByFaqCode.html?faqCode=TDDL-4620]
```

加 HINT 则可执行成功：

```
mysql> /!TDDL:FORBID_EXECUTE_DML_ALL=false*/delete from tt;
Query OK, 10 row affected (0.21 sec)
```

执行全表更新默认会被拦截：

```
mysql> update tt set id = 1;
ERR-CODE: [TDDL-4620][ERR_FORBID_EXECUTE_DML_ALL] Forbid execute DELETE ALL or UPDATE ALL
sql. More: [http://middleware.alibaba-inc.com/faq/faqByFaqCode.html?faqCode=TDDL-4620]
```

加 HINT 则可执行成功：

```
mysql> / ! TDDL:FORBID_EXECUTE_DML_ALL=false*/update tt set id = 1;
Query OK, 10 row affected (0.21 sec)
```

DELETE 或者 UPDATE 语句中带有 WHERE 或者 LIMIT 条件，不会出现这个限制。

```
mysql> delete from tt where id = 1;
Query OK, 1 row affected (0.21 sec)
```