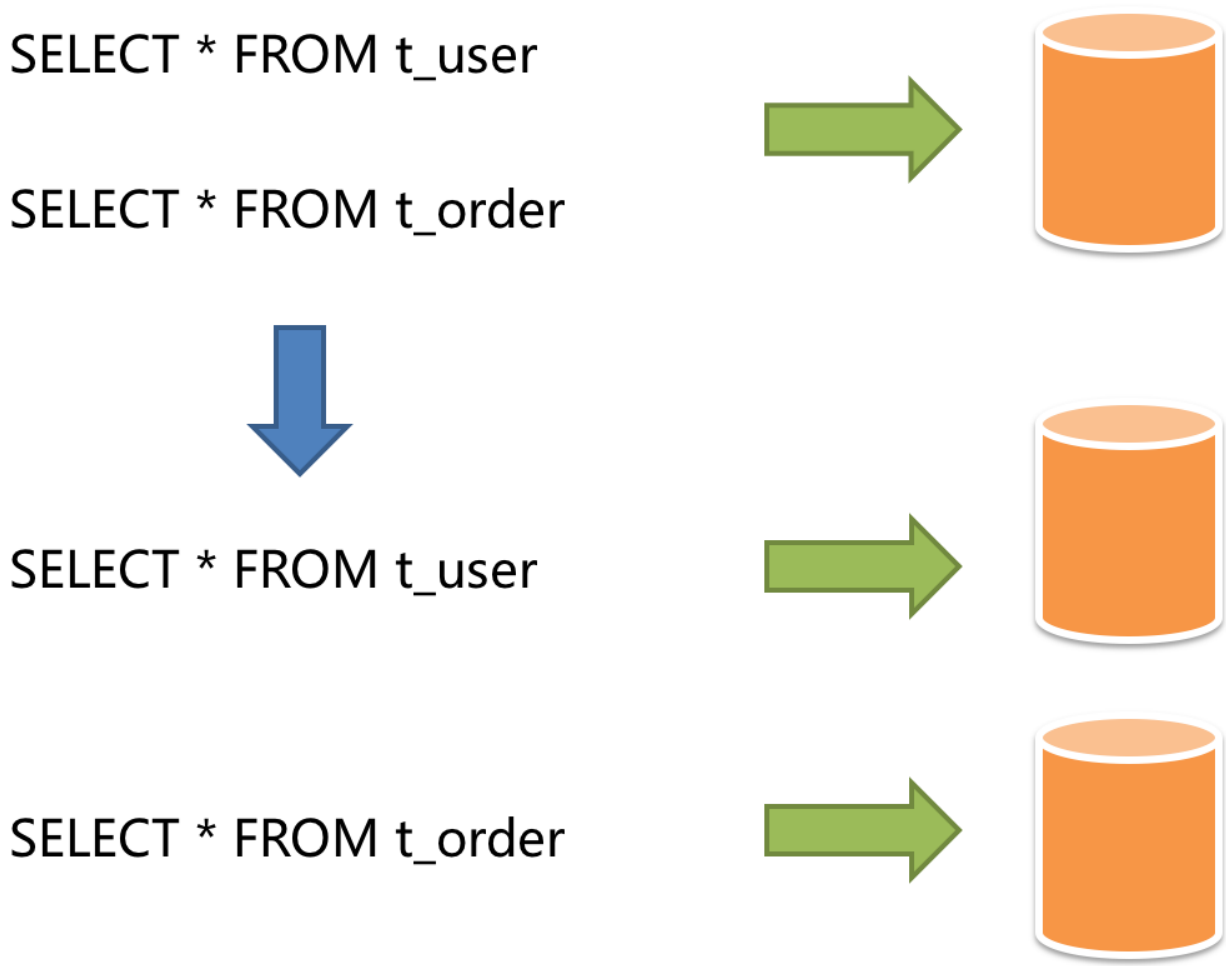
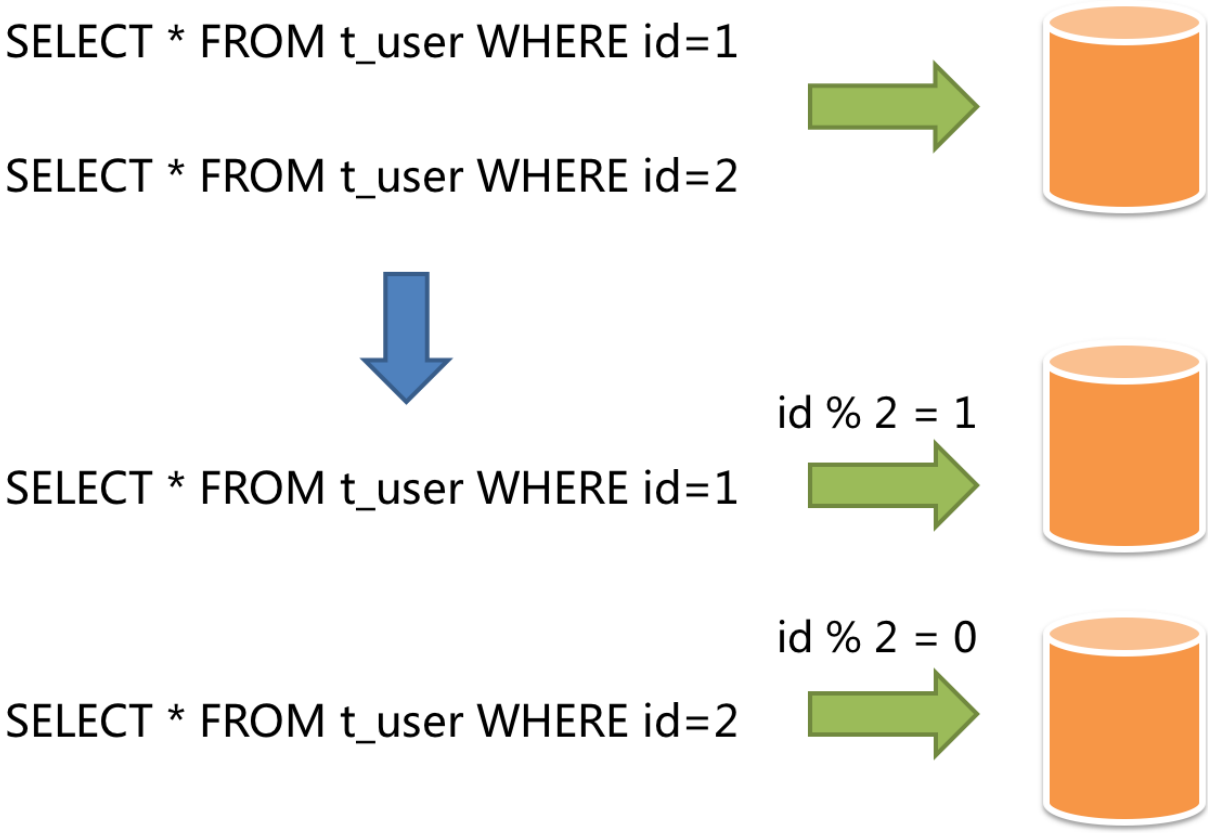
4-Sharding-JDBC分库分表

4.1 分库分表概念

垂直拆分



水平拆分



逻辑表

水平拆分的数据库（表）的相同逻辑和数据结构表的总称。例：订单数据根据主键尾数拆分为10张表，分别是`t_order_0`到`t_order_9`，他们的逻辑表名为`t_order`。

```
sharding.jdbc.config.sharding:
    tables:                                          分片表定义
      t_order:                                       逻辑表
            actualDataNodes: ds$->{0..1}.t_order$->{0..1}        真实表  数据节点
            databaseStrategy:                        数据源分片策略
              inline:                                分片策略
                    shardingColumn: user_id          分片键
                    algorithmInlineExpression: ds$->{user_id%2}   分片算法
            tableStrategy:                           表分片策略
              inline:
                    shardingColumn: order_id
                    algorithmInlineExpression: t_order$->{order_id%2}
```

配置示例

```yaml
shardingRule:
  tables:
    t_order:
      actualDataNodes: ds${0..1}.t_order${0..1}
      databaseStrategy:
        inline:
          shardingColumn: user_id
          algorithmInlineExpression: ds${user_id % 2}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmInlineExpression: t_order${order_id % 2}
```

真实表

在分片的数据库中真实存在的物理表。即上个示例中的`t_order_0`到`t_order_9`。

数据节点

数据分片的最小单元。由数据源名称和数据表组成，例：`ds_0.t_order_0`。

可分为均匀分布和自定义分布两种形式。

- 均匀分布

指数据表在每个数据源内呈现均匀分布的态势，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order0
  └── t_order1
```

那么数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order0, db1.t_order1
```

- 自定义分布

指数据表呈现有特定规则的分布，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order2
  ├── t_order3
  └── t_order4
```

那么数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order2, db1.t_order3, db1.t_order4
```

分片策略配置

对于分片策略存有数据源分片策略和表分片策略两种维度。

- 数据源分片策略

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

- 表分片策略

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

两种策略的API完全相同。

```yaml
shardingRule:
  tables:
    t_order:
      actualDataNodes: ds$->{0..1}.t_order$->{0..1}
      databaseStrategy:
        inline:
          shardingColumn: user_id
          algorithmInlineExpression: ds$->{user_id % 2}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmInlineExpression: t_order$->{order_id % 2}

  defaultDataSourceName: ds_0
  defaultDatabaseStrategy:
    inline:
```

```
      shardingColumn: user_id
      algorithmExpression: ms_ds${user_id % 2}
  defaultTableStrategy:
    none:
  defaultKeyGenerator:
    type: SNOWFLAKE
```

分片策略

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

- 标准分片策略

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

- 复合分片策略

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

- 行表达式分片策略

对应InlineShardingStrategy。使用Groovy的表达式，提供对SQL语句中的=和IN的分片操作支持，只支持单分片键。对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的Java代码开发，如: `t_user_$->{u_id % 8}` 表示t_user表根据u_id模8，而分成8张表，表名称为 `t_user_0` 到 `t_user_7`。行表达式语法见下一节"行表达式语法说明"。

- Hint分片策略

对应HintShardingStrategy。通过Hint而非SQL解析的方式分片的策略。

- 不分片策略

对应NoneShardingStrategy。不分片的策略。

```
```

```
  defaultDatabaseShardingStrategy: # 默认库级别sharidng规则,对应代码中ShardingStrategy接口的实现类,目前支持none,inline,hint,complex,standard五种配置注意此处默认配置仅可以配置五个中的一个
  # 规则配置同样适合表sharding配置,同样是在这些算法中选择
    none: # 不配置任何规则,SQL会被发给所有节点去执行,这个规则没有子项目可以配置
    inline: # 行表达式分片
      shardingColumn: test_id # 分片列名称
      algorithmExpression: master_test_${test_id % 2} # 分片表达式,根据指定的表达式计算得到需要路由到的数据源名称 需要是合法的groovy表达式,示例配置中,取余为0则语句路由到master_test_0,取余为1则路由到master_test_1
    hint: #基于标记的sharding分片
      shardingAlgorithm: # 需要是HintShardingAlgorithm接口的实现,目前代码中,有为测试目的实现的OrderDatabaseHintShardingAlgorithm可参考
    complex: # 支持多列的shariding,目前无生产可用实现
      shardingColumns: # 逗号切割的列
      shardingAlgorithm: # ComplexKeysShardingAlgorithm接口的实现类
    standard: # 单列sharidng算法,需要配合对应的preciseShardingAlgorithm,rangeShardingAlgorithm接口的实现使用
      shardingColumn: # 列名,允许单列
      preciseShardingAlgorithm: # preciseShardingAlgorithm接口的实现类
      rangeShardingAlgorithm: # rangeShardingAlgorithm接口的实现类
```

行表达式语法说明

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

- `${begin..end}`表示范围区间
- `${[unit1, unit2, unit_x]}`表示枚举值
- 行表达式中如果出现连续多个`${ expression }`或`$->{ expression }`表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

spring boot 分片示例

配置

```yaml
sharding:
  jdbc:
    datasource:
      names: ds0,ds1
      ds0:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/db1?
useUnicode=true&characterEncoding=utf-8&serverTimezone=UTC
        username: mike
        password: Mike666!
        maxPoolSize: 50
        minPoolSize: 1
      ds1:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/db2?
useUnicode=true&characterEncoding=utf-8&serverTimezone=UTC
        username: mike
        password: Mike666!
        maxPoolSize: 50
        minPoolSize: 1
    config:
      sharding:
        tables:
           t_order:
             actual-data-nodes: ds$->{0..1}.t_order$->{0..1}
             database-strategy:
               standard:
                 sharding-column: order_time
                 preciseAlgorithmClassName:
com.study.mike.sharding.jdbc.sharding.OrderTimePreciseShardingAlgorithm
             #inline:
               #sharding-column: customer_id
               #algorithm-expression: ds$->{customer_id % 2}
             table-strategy:
               inline:
                 sharding-column: order_id
                 algorithm-expression: t_order$->{order_id % 2}
        default-data-source-name: ds0
      props:
        sql.show: true
```

算法实现：

```
import java.util.Calendar;
import java.util.Collection;
import java.util.Date;
import java.util.Iterator;

import io.shardingsphere.api.algorithm.sharding.PreciseShardingValue;
import
io.shardingsphere.api.algorithm.sharding.standard.PreciseShardingAlgorithm
;

public class OrderTimePreciseShardingAlgorithm implements
PreciseShardingAlgorithm<Date>{

    Date[] dateRanges = new Date[2];

    {
        Calendar cal = Calendar.getInstance();
        cal.set(2019, 1, 1, 0, 0, 0);
        dateRanges[0] = cal.getTime();
        cal.set(2020, 1, 1, 0, 0, 0);
        dateRanges[1] = cal.getTime();
    }

    @Override
    public String doSharding(Collection<String> availableTargetNames,
PreciseShardingValue<Date> shardingValue) {

        Date d = shardingValue.getValue();
        Iterator<String> ite = availableTargetNames.iterator();
        String target = null;
        for(Date s : dateRanges) {
            target = ite.next();
            if(d.before(s)) {
                break;
            }
        }
        return target;
    }
```

```
}
```

建表SQL

```
CREATE TABLE t_order0 (
    order_id    BIGINT PRIMARY KEY,
    order_time  DATETIME,
    customer_id BIGINT,
    order_amount    DECIMAL(8,2)
);

CREATE TABLE t_order1 (
    order_id    BIGINT PRIMARY KEY,
    order_time  DATETIME,
    customer_id BIGINT,
    order_amount    DECIMAL(8,2)
);
```

默认配置（yaml格式）

```
shardingRule: # sharding的配置
  # 配置主要分两类,一类是对整个sharding规则所有表生效的默认配置,一个是sharing具体某张表时候的配置
  # 首先说默认配置
  masterSlaveRules: # 在shardingRule中也可以配置shardingRule,对分片生效,具体内容与全局masterSlaveRule一致,但语法为:
    master_test_0:
      masterDataSourceName: master_ds_0
      slaveDataSourceNames:
        - slave_ds_0
    master_test_1:
      masterDataSourceName: master_ds_1
      slaveDataSourceNames:
        - slave_ds_1
        - slave_ds_1_slave2
  defaultDataSourceName: master_test_0 # 这里的数据源允许是dataSources的配置项目或者masterSlaveRules配置的名称,配置为masterSlaveRule的话相当于就是配置读写分离了
```

```
  broadcastTables: # 广播表 这里配置的表列表,对于发生的所有数据变更,都会不经sharidng处理,而是直接发送到所有数据节点,注意此处为列表,每个项目为一个表名称
    - broad_1
    - broad_2
  bindingTables: # 绑定表,也就是实际上哪些配置的sharidng表规则需要实际生效的列表,配置为yaml列表,并且允许单个条目中以逗号切割,所配置表必须已经配置为逻辑表
    - sharding_t1
    - sharding_t2,sharding_t3
  defaultDatabaseShardingStrategy: # 默认库级别sharidng规则,对应代码中ShardingStrategy接口的实现类,目前支持none,inline,hint,complex,standard五种配置注意此处默认配置仅可以配置五个中的一个
  # 规则配置同样适合表sharding配置,同样是在这些算法中选择
    none: # 不配置任何规则,SQL会被发给所有节点去执行,这个规则没有子项目可以配置
    inline: # 行表达式分片
      shardingColumn: test_id # 分片列名称
      algorithmExpression: master_test_${test_id % 2} # 分片表达式,根据指定的表达式计算得到需要路由到的数据源名称 需要是合法的groovy表达式,示例配置中,取余为0则语句路由到master_test_0,取余为1则路由到master_test_1
    hint: #基于标记的sharding分片
      shardingAlgorithm: # 需要是HintShardingAlgorithm接口的实现,目前代码中,仅有为测试目的实现的OrderDatabaseHintShardingAlgorithm,没有生产环境可用的实现
    complex: # 支持多列的shariding,目前无生产可用实现
      shardingColumns: # 逗号切割的列
      shardingAlgorithm: # ComplexKeysShardingAlgorithm接口的实现类
    standard: # 单列sharidng算法,需要配合对应的preciseShardingAlgorithm,rangeShardingAlgorithm接口的实现使用,目前无生产可用实现
      shardingColumn: # 列名,允许单列
      preciseShardingAlgorithm: # preciseShardingAlgorithm接口的实现类
      rangeShardingAlgorithm: # rangeShardingAlgorithm接口的实现类
  defaultTableStrategy: #配置参考defaultDatabaseShardingStrategy,区别在于,inline算法的配置中,algorithmExpression的配置算法结果需要是实际的物理表名称,而非数据源名称
  defaultKeyGenerator: #默认的主键生成算法 如果没有设置,默认为SNOWFLAKE算法
    column: # 自增键对应的列名称
    type: #自增键的类型,主要用于调用内置的主键生成算法有三个可用值:SNOWFLAKE(时间戳+worker id+自增id),UUID(java.util.UUID类生成的随机UUID),LEAF,其中Snowflake算法与UUID算法已经实现,LEAF目前(2018-01-14)尚未实现
    props:
      # 定制算法需要设置的参数,比如SNOWFLAKE算法的worker.id与max.tolerate.time.difference.milliseconds
```

分布式主键

传统数据库软件开发中，主键自动生成技术是基本需求。而各个数据库对于该需求也提供了相应的支持，比如MySQL的自增键，Oracle的自增序列等。 数据分片后，不同数据节点生成全局唯一主键是非常棘手的问题。同一个逻辑表内的不同实际表之间的自增键由于无法互相感知而产生重复主键。 虽然可通过约束自增主键初始值和步长的方式避免碰撞，但需引入额外的运维规则，使解决方案缺乏完整性和可扩展性。

目前有许多第三方解决方案可以完美解决这个问题，如UUID等依靠特定算法自生成不重复键，或者通过引入主键生成服务等。为了方便用户使用、满足不同用户不同使用场景的需求，ShardingSphere不仅提供了内置的分布式主键生成器，例如UUID、SNOWFLAKE、LEAF(进行中)，还抽离出分布式主键生成器的接口，方便用户自行实现自定义的自增主键生成器。

内置的主键生成器

UUID

采用UUID.randomUUID()的方式产生分布式主键。

SNOWFLAKE

ShardingSphere提供灵活的配置分布式主键生成策略方式。 在分片规则配置模块可配置每个表的主键生成策略，默认使用雪花算法（snowflake）生成64bit的长整型数据。

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

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

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

- 符号位(1bit)

预留的符号位，恒为零。

- 时间戳位(41bit)

41位的时间戳可以容纳的毫秒数是2的41次幂，一年所使用的毫秒数是：`365 * 24 * 60 * 60 * 1000`。通过计算可知：

```
Math.pow(2, 41) / (365 * 24 * 60 * 60 * 1000L);
```

结果约等于69.73年。ShardingSphere的雪花算法的时间纪元从2016年11月1日零点开始，可以使用到2086年，相信能满足绝大部分系统的要求。

- 工作进程位(10bit)

该标志在Java进程内是唯一的，如果是分布式应用部署应保证每个工作进程的id是不同的。该值默认为0，可通过调用静态方法`DefaultKeyGenerator.setWorkerId()`设置。

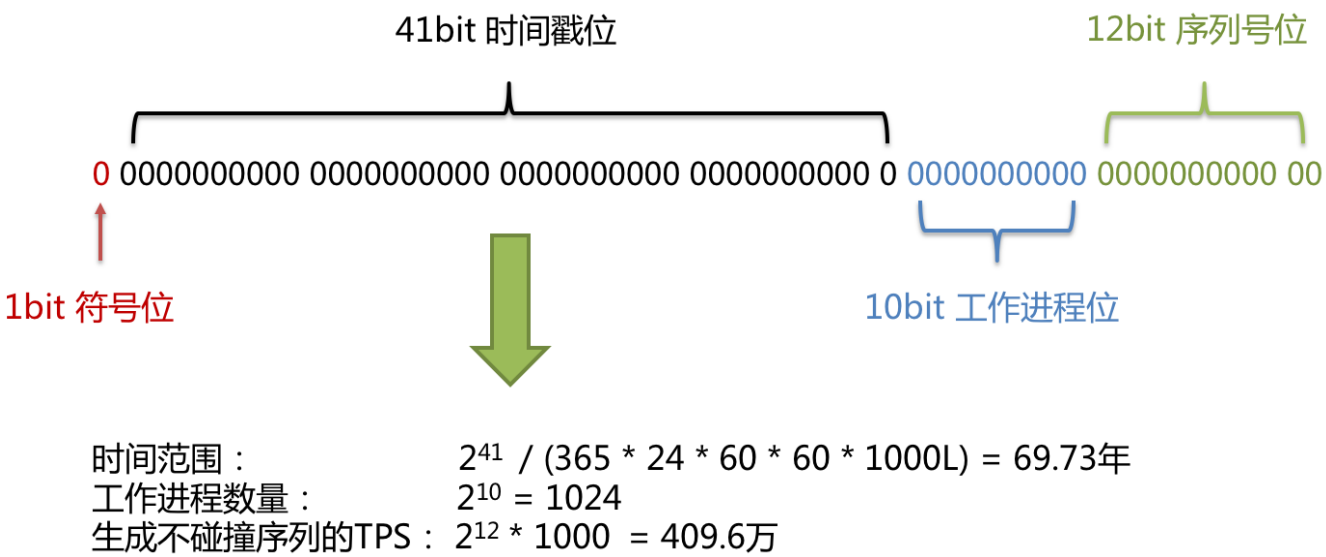
- 序列号位(12bit)

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

时钟回拨

服务器时钟回拨会导致产生重复序列，因此默认分布式主键生成器提供了一个最大容忍的时钟回拨毫秒数。 如果时钟回拨的时间超过最大容忍的毫秒数阈值，则程序报错；如果在可容忍的范围内，默认分布式主键生成器会等待时钟同步到最后一次主键生成的时间后再继续工作。 最大容忍的时钟回拨毫秒数的默认值为0，可通过调用静态方法`DefaultKeyGenerator.setMaxTolerateTimeDifferenceMilliseconds()`设置。

雪花算法主键的详细结构见下图。



LEAF

借鉴美团点评分布式ID生成系统Leaf，借助注册中心生成分布式自增主键。该内置分布式主键算法正在进行中。

配置示例

```
shardingRule:
  tables:
    t_order:
      actualDataNodes: ms_ds${0..1}.t_order${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order${order_id % 2}
      keyGeneratorColumnName: order_id
      #keyGeneratorClassName: xxx.xxx.XXclass
  #defaultKeyGenerator:
    #type: SNOWFLAKE
```

配置说明

```
  defaultKeyGenerator: #默认的主键生成算法 如果没有设置,默认为SNOWFLAKE算法
    column: # 自增键对应的列名称
    type: #自增键的类型,主要用于调用内置的主键生成算法有三个可用值:SNOWFLAKE(时间戳+worker id+自增id),UUID(java.util.UUID类生成的随机UUID),LEAF,其中Snowflake算法与UUID算法已经实现,LEAF目前(2018-01-14)尚未实现
    props:
      # 定制算法需要设置的参数,比如SNOWFLAKE算法的worker.id与max.tolerate.time.difference.milliseconds

  tables: #配置表sharding的主要位置
    sharding_t1:
      actualDataNodes: master_test_${0..1}.t_order${0..1} # sharidng 表对应的数据源以及物理名称,需要用表达式处理,表示表实际上在哪些数据源存在,配置示例中,意思是总共存在4个分片 master_test_0.t_order0,master_test_0.t_order1,master_test_1.t_order0,master_test_1.t_order1
      # 需要注意的是,必须保证设置databaseStrategy可以路由到唯一的dataSource,tableStrategy可以路由到dataSource中唯一的物理表上,否则可能导致错误:一个insert语句被插入到多个实际物理表中
      databaseStrategy: # 局部设置会覆盖全局设置,参考defaultDatabaseShardingStrategy
      tableStrategy: # 局部设置会覆盖全局设置,参考defaultTableStrategy
      keyGenerator: # 局部设置会覆盖全局设置,参考defaultKeyGenerator
```

```
      logicIndex: # 逻辑索引名称 由于Oracle,PG这种数据库中,索引与表共用命名空间,如果接受到drop index语句,执行之前,会通过这个名称配置的确定对应的实际物理表名称
```

dao SQL语句

```
@Insert("insert into t_order(order_time,customer_id) values(#{orderTime},#{customerId})")
void addOrder(Order o);
```

绑定表

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

```
SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);
```

在不配置绑定表关系时，假设分片键`order_id`将数值10路由至第0片，将数值11路由至第1片，那么路由后的SQL应该为4条，它们呈现为笛卡尔积：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);

SELECT i.* FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);
```

在配置绑定表关系后，路由的SQL应该为2条：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
WHERE o.order_id in (10, 11);
```

其中`t_order`在FROM的最左侧，ShardingSphere将会以它作为整个绑定表的主表。 所有路由计算将会只使用主表的策略，那么`t_order_item`表的分片计算将会使用`t_order`的条件。故绑定表之间的分区键要完全相同。

绑定表配置示例

```
shardingRule:
  tables:
    t_order:
      actualDataNodes: ds${0..1}.t_order${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order${order_id % 2}
      keyGenerator:
        column: order_id
    t_order_item:
      actualDataNodes: ds${0..1}.t_order_item${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order_item${order_id % 2}
  bindingTables:
    - t_order,t_order_item
  broadcastTables:
    - t_config
```

绑定表配置说明

```
  bindingTables: # 绑定表,也就是实际上哪些配置的sharidng表规则需要实际生效的列表,配置为yaml列表,并且允许单个条目中以逗号切割,所配置表必须已经配置为逻辑表
    - sharding_t1
    - sharding_t2,sharding_t3
```

广播表

指所有的分片数据源中都存在的表，表结构和表中的数据在每个数据库中均完全一致。适用于数据量不大且需要与海量数据的表进行关联查询的场景，例如：字典表。

配置示例

```
shardingRule:

  bindingTables:
    - t_order,t_order_item
  broadcastTables:
    - t_config
```

配置说明

```
  broadcastTables: # 广播表 这里配置的表列表,对于发生的所有数据变更,都会不经sharidng处理,而是直接发送到所有数据节点,注意此处为列表,每个项目为一个表名称
    - broad_1
    - broad_2
```

逻辑索引

某些数据库（如：PostgreSQL）不允许同一个库存在名称相同索引，某些数据库（如：MySQL）则允许只要同一个表中不存在名称相同的索引即可。 逻辑索引用于同一个库不允许出现相同索引名称的分表场景，需要将同库不同表的索引名称改写为`索引名 + 表名`，改写之前的索引名称成为逻辑索引。

4.2 数据分片+读写分离

```
dataSources:
  ds0: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ds0
    username: root
    password:
```

```
  ds0_slave0: !!org.apache.commons.dbcp.BasicDataSource
      driverClassName: com.mysql.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ds0_slave0
      username: root
      password:
  ds0_slave1: !!org.apache.commons.dbcp.BasicDataSource
      driverClassName: com.mysql.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ds0_slave1
      username: root
      password:
  ds1: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ds1
    username: root
    password:
  ds1_slave0: !!org.apache.commons.dbcp.BasicDataSource
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/ds1_slave0
        username: root
        password:
  ds1_slave1: !!org.apache.commons.dbcp.BasicDataSource
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/ds1_slave1
        username: root
        password:

shardingRule:
  tables:
    t_order:
      actualDataNodes: ms_ds${0..1}.t_order${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order${order_id % 2}
      keyGenerator:
        Column: order_id
    t_order_item:
      actualDataNodes: ms_ds${0..1}.t_order_item${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order_item${order_id % 2}
```

```
  bindingTables:
    - t_order,t_order_item
  broadcastTables:
    - t_config

  defaultDataSourceName: ds_0
  defaultDatabaseStrategy:
    inline:
      shardingColumn: user_id
      algorithmExpression: ms_ds${user_id % 2}
  defaultTableStrategy:
    none:
  defaultKeyGenerator:
    type: SNOWFLAKE

  masterSlaveRules:
      ms_ds0:
        masterDataSourceName: ds0
        slaveDataSourceNames:
          - ds0_slave0
          - ds0_slave1
        loadBalanceAlgorithmType: ROUND_ROBIN
        configMap:
          master-slave-key0: master-slave-value0
      ms_ds1:
        masterDataSourceName: ds1
        slaveDataSourceNames:
          - ds1_slave0
          - ds1_slave1
        loadBalanceAlgorithmType: ROUND_ROBIN
        configMap:
          master-slave-key1: master-slave-value1

props:
  sql.show: true
```

分片时的数据源是读写分离中定义的集群名。