MySQL高级（讲师：启源）

Sakila

课程介绍

第1节 学前必备基础

- MySQL软件下载和安装（建议版本5.7.28）
- 熟悉MySQL工具和基本SQL操作

  Window : MySQL WorkBench, Navicat, SQLyog,HeidiSQL,MySQL Front

  Linux：MySQL WorkBeanch, Navicat

  IOS：Navicat、Sequel Pro

  DDL、DML、DQL、TCL
- 熟悉主键、外键、非空、唯一等约束

  创建主键、外键....
- 熟悉索引、事务概念和基本使用

  概念、创建

第2节 课程主要内容

- MySQL架构原理和存储机制

  MySQL体系结构（内存结构、磁盘结构）、SQL运行机制、存储引擎、Undo/Redo Log等等
- MySQL索引存储机制和工作原理

  索引存储结构、索引查询原理、索引分析和优化、查询优化等
- MySQL事务和锁工作原理

  事务隔离级别、事务并发处理、锁机制和实战等
- MySQL集群架构及相关原理

集群架构设计理念、主从架构、双主架构、分库分表等

- 互联网海量数据处理实战

  ShardingSphere、MyCat中间实战操作，分库分表实战

- MySQL第三方工具实战

  同步工具、运维工具、监控工具等

第3节 MySQL起源和分支

MySQL 是最流行的关系型数据库软件之一，由于其体积小、速度快、开源免费、简单易用、维护成本低等，在集群架构中易于扩展、高可用，因此深受开发者和企业的欢迎。

Rank			DBMS	Database Model	Score		
Apr 2020	Mar 2020	Apr 2019			Apr 2020	Mar 2020	Apr 2019
1.	1.	1.	Oracle	Relational, Multi-model	1345.42	+4.78	+65.48
2.	2.	2.	MySQL	Relational, Multi-model	1268.35	+8.62	+53.21
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1083.43	-14.43	+23.47
4.	4.	4.	PostgreSQL	Relational, Multi-model	509.86	-4.06	+31.14
5.	5.	5.	MongoDB	Document, Multi-model	438.43	+0.82	+36.45
6.	6.	6.	IBM Db2	Relational, Multi-model	165.63	+3.07	-10.42
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	148.91	-0.26	+2.91
8.	8.	↓ 7.	Redis	Key-value, Multi-model	144.81	-2.77	-1.57
9.	↑ 10.	↑ 10.	SQLite	Relational	122.19	+0.24	-2.02
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	121.92	-3.22	-22.73

Oracle和MySQL是世界市场占比最高的两种数据库。

IOE：IBM的服务器，Oracle数据库，EMC存储设备。都是有钱的公司产品采购，例如银行、电信、石油、证券等大企业。

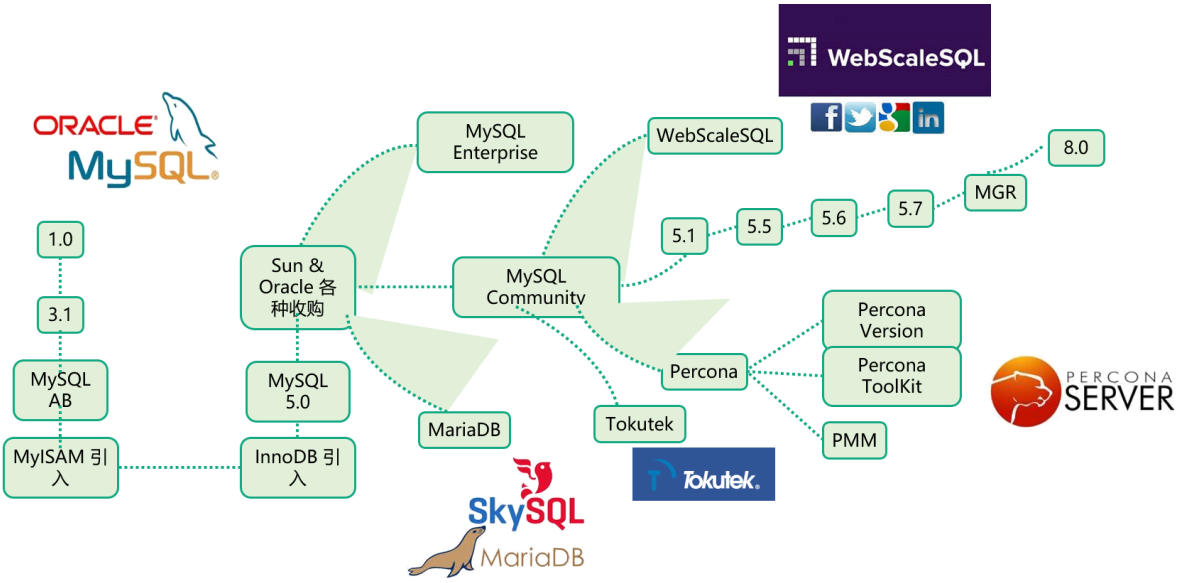
Oracle：垄断，有钱的大企业采用，互联网企业之外使用第一。

MySQL：互联网高速发展，互联网企业使用第一。

MySQL发展历程如下：

时间	事件
1979 年	当时瑞典的 Monty Widenius 在 Tcx DataKonsult 公司工作，他开发了一款名为 Unireg 的工具，它是一个面向报表的存储引擎，利用索引顺序来读取数据，这也是 ISAM 存储引擎算法的前身。
1985 年	Monty 和 David Axmart 等几个小伙子成立了一家公司(MySQL AB 前身)，研发出了 ISAM(Indexed Sequential Access Method)存储引擎工具。
1990 年	客户要求 ISAM 工具能提供 SQL 接口，于是 Monty 找到了 David Hughes(mSQL 的发明人）商讨合作事宜，后来发现 mSQL 的速度也无法满足需求。于是 Monty 决心自己重写一个 SQL 支持，由此着手 MySQL 设计和研发。
1996 年	Monty 与 David Axmart 一起协作，开发出 MySQL 第一个版本 1.0。
1996 年 10 月	MySQL 3.1 发布了，没有 2.x 版本。最开始只提供了 Solaris 下的二进制版本。同年 11 月发布了 Linux 版本。
1999-2000 年	Monty、Allan 和 David 三人在瑞典创建了 MySQL AB 公司，并且与 Sleepycat 合作开发出引入了 BDB 引擎，MySQL 从此开始支持事务处理了。
2000 年	MySQL 公布了自己的源代码，并采用 GPL(GNU General Public License)许可协议正式开源。
2000 年 4 月	MySQL 对旧的存储引擎 ISAM 进行了整理，命名为 MyISAM。
2001 年	Heikki Tuuri 向 MySQL 建议集成他的 InnoDB 存储引擎，这个引擎同样支持事务处理，还支持行级锁。MySQL 与 InnoDB 正式结合版本是 4.0。至此 MySQL 已集成了 MyISAM 和 InnoDB 两种大主力引擎。
2005 年 10 月	MySQL 5.0 版本发布，这是 MySQL 历史上最有里程碑意义的一个版本，在 5.0 版本加入了游标、存储过程和触发器的支持。
2008 年 1 月	MySQL AB 公司被 Sun 公司以 10 亿美金收购，MySQL 数据库进入 Sun 时代。
2009 年 4 月	Oracle 公司以 74 亿美元收购 Sun 公司，自此 MySQL 数据库进入 Oracle 时代，而其第三方的存储引擎 InnoDB 早在 2005 年就被 Oracle 公司收购。
2010 年 4 月	发布了 MySQL 5.5 版本。Oracle 对 MySQL 版本重新进行了划分，分成了社区版和企业版。默认引擎更换为 InnoDB、增加表分区等。
2013 年 2 月	MySQL 5.6 首个正式版 5.6.10 发布。MySQL 5.6 对 InnoDB 引擎进行了改造，提供全文索引能为，使 InnoDB 适合各种应用场景。
2015 年 10 月	MySQL 5.7 首个 GA 正式版 5.7.9 发布。
2016 年 9 月	MySQL 8.0 首个开发版发布，增加了数据字典、账号权限角色表、InnoDB 增强、JSON 增强等等。
2018 年 4 月	MySQL 8.0 首个 GA 正式版 8.0.11 发布

MySQL主流分支如下图所示



MySQL从最初的1.0、3.1到后来的8.0，发生了各种各样的变化。被Oracle收购后，MySQL的版本演化出了多个分支，除了需要付费的MySQL企业版本，还有很多MySQL社区版本。还有一条分支非常流行的开源分支版本叫Percona Server，它是MySQL的技术支持公司Percona推出的，也是在实际工作中经常碰到的。Percona Server在MySQL官方版本的基础上做了一些补丁和优化，同时推出了一些工具。另外一个非常不错的版本叫MariaDB，它是MySQL的公司被Oracle收购后，MySQL的创始人Monty先生，按原来的思路重新写的一套新数据库，同时也把 InnoDB 引擎作为主要存储引擎，也算 MySQL 的分支。

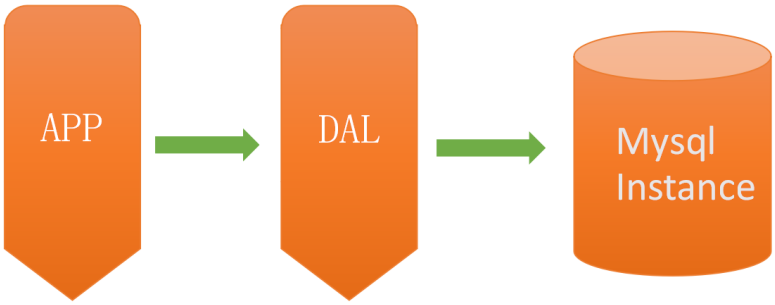
第4节 MySQL应用架构演变

本节主要介绍网站在不同的并发访问量级和数据量级下，MySQL应用架构的演变过程。

用户请求--》 应用层 --》服务层 --》存储层

- 架构V1.0 - 单机单库

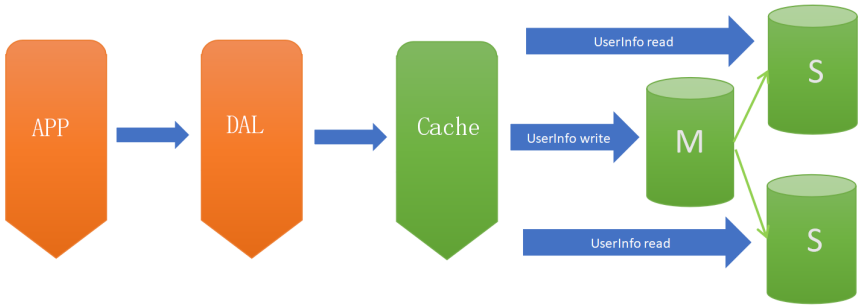
  一个简单的小型网站或者应用背后的架构可以非常简单, 数据存储只需要一个MySQL Instance就能满足数据读取和写入需求（这里忽略掉了数据备份的实例），处于这个的阶段系统，一般会把所有的信息存到一个MySQL Instance里面。



  V1.0 瓶颈

  - 数据量太大，超出一台服务器承受
  - 读写操作量太大，超出一台服务器承受
  - 一台服务器挂了，应用也会挂掉（可用性差）
- 架构V2.0 - 主从架构

V2.0架构主要解决架构V1.0下的高可用和读扩展问题，通过给Instance挂载从库解决读取的压力，主库宕机也可以通过主从切换保障高可用。在MySQL的场景下就是通过主从结构（双主结构也属于特殊的主从），主库抗写压力，通过从库来分担读压力，对于写少读多的应用，V2.0主从架构完全能够胜任。

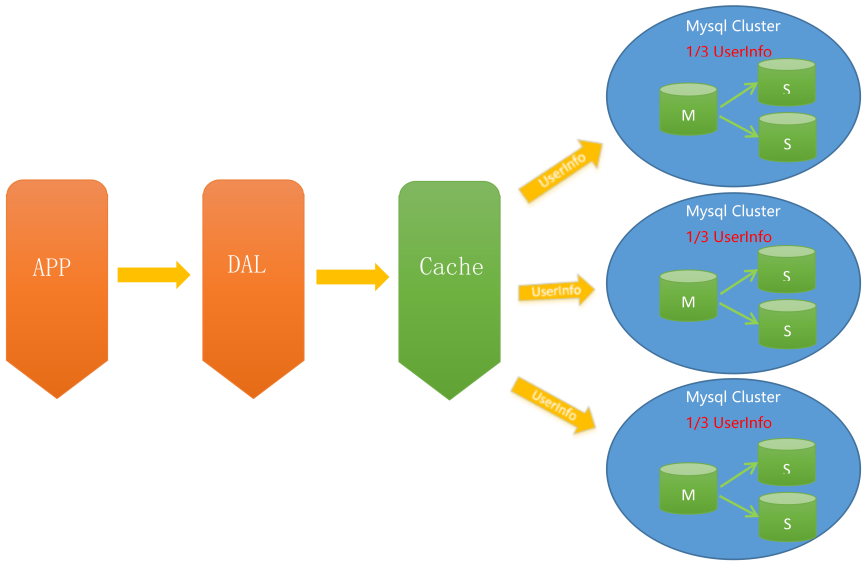


V2.0瓶颈

- 数据量太大，超出一台服务器承受
- 写操作太大，超出一台M服务器承受

- 架构V3.0 - 分库分表

  对于V1.0和V2.0遇到写入瓶颈和存储瓶颈时，可以通过水平拆分来解决，水平拆分和垂直拆分有较大区别，垂直拆分拆完的结果，每一个实例都是拥有全部数据的，而水平拆分之后，任何实例都只有全量的1/n的数据。以下图所示，将Userinfo拆分为3个Sharding，每个Sharding持有总量的1/3数据，3个Sharding数据的总和等于一份完整数据

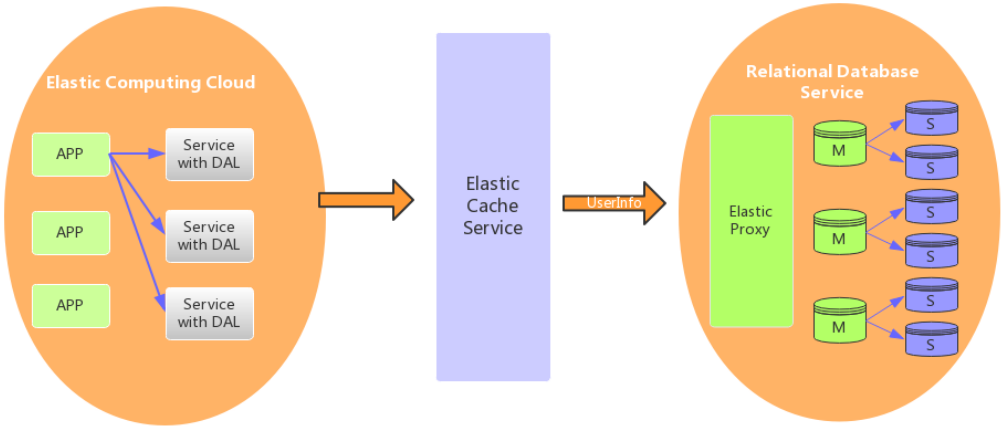


  数据如何路由成为一个关键问题， 一般可以采用范围拆分，List拆分、Hash拆分等。

  如何保持数据的一致性也是个难题。

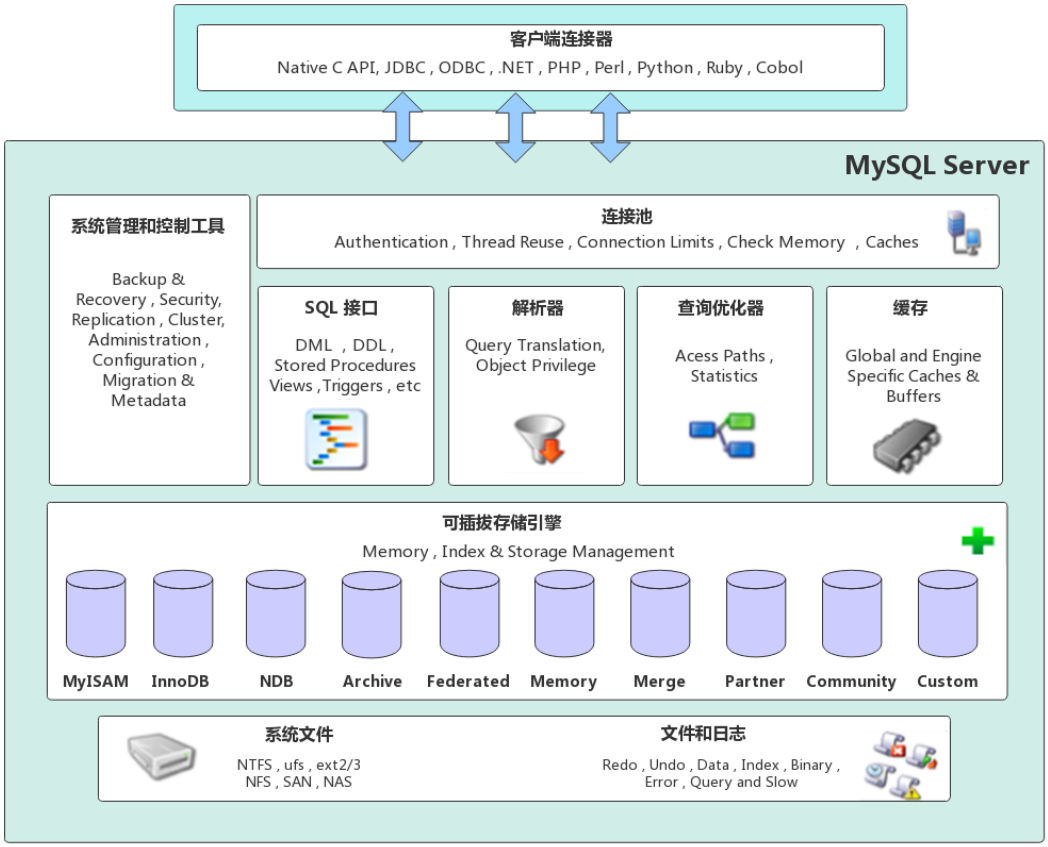
- 架构V4.0 - 云数据库

  云数据库（云计算）现在是各大IT公司内部作为节约成本的一个突破口，对于数据存储的MySQL来说，如何让其成为一个saas（Software as a Service）是关键点。MySQL作为一个saas服务，服务提供商负责解决可配置性，可扩展性，多用户存储结构设计等这些疑难问题。



第一天 MySQL架构原理

第1节 MySQL体系架构



MySQL Server架构自顶向下大致可以分网络连接层、服务层、存储引擎层和系统文件层。

一、网络连接层

- 客户端连接器（Client Connectors）：提供与MySQL服务器建立的支持。目前几乎支持所有主流的服务端编程技术，例如常见的 Java、C、Python、.NET等，它们通过各自API技术与MySQL建立连接。

二、服务层（MySQL Server）

服务层是MySQL Server的核心，主要包含系统管理和控制工具、连接池、SQL接口、解析器、查询优化器和缓存六个部分。

- **连接池（Connection Pool）**：负责存储和管理客户端与数据库的连接，一个线程负责管理一个连接。
- **系统管理和控制工具（Management Services & Utilities）**：例如备份恢复、安全管理、集群管理等
- **SQL接口（SQL Interface）**：用于接受客户端发送的各种SQL命令，并且返回用户需要查询的结果。比如DML、DDL、存储过程、视图、触发器等。
- **解析器（Parser）**：负责将请求的SQL解析生成一个"解析树"。然后根据一些MySQL规则进一步检查解析树是否合法。
- **查询优化器（Optimizer）**：当"解析树"通过解析器语法检查后，将交由优化器将其转化成执行计划，然后与存储引擎交互。

  > select uid,name from user where gender=1;
  >
  > 选取--》投影--》联接 策略
  >
  > 1）select先根据where语句进行选取，并不是查询出全部数据再过滤
  >
  > 2）select查询根据uid和name进行属性投影，并不是取出所有字段
  >
  > 3）将前面选取和投影联接起来最终生成查询结果

- **缓存（Cache&Buffer）**： 缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，权限缓存，引擎缓存等。如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。

三、存储引擎层（Pluggable Storage Engines）

存储引擎负责MySQL中数据的存储与提取，与底层系统文件进行交互。MySQL存储引擎是插件式的，服务器中的查询执行引擎通过接口与存储引擎进行通信，接口屏蔽了不同存储引擎之间的差异 。现在有很多种存储引擎，各有各的特点，最常见的是MyISAM和InnoDB。

四、系统文件层（File System）

该层负责将数据库的数据和日志存储在文件系统之上，并完成与存储引擎的交互，是文件的物理存储层。主要包含日志文件，数据文件，配置文件，pid 文件，socket 文件等。

- 日志文件
  - 错误日志（Error log）

    默认开启，show variables like '%log_error%'
  - 通用查询日志（General query log）

    记录一般查询语句，show variables like '%general%';
  - 二进制日志（binary log）

    记录了对MySQL数据库执行的更改操作，并且记录了语句的发生时间、执行时长；但是它不记录select、show等不修改数据库的SQL。主要用于数据库恢复和主从复制。

    show variables like '%log_bin%'; //是否开启

    show variables like '%binlog%'; //参数查看

    show binary logs;//查看日志文件
  - 慢查询日志（Slow query log）

    记录所有执行时间超时的查询SQL，默认是10秒。

    show variables like '%slow_query%'; //是否开启
    show variables like '%long_query_time%'; //时长

- 配置文件

  用于存放MySQL所有的配置信息文件，比如my.cnf、my.ini等。

- 数据文件
  - db.opt 文件：记录这个库的默认使用的字符集和校验规则。
  - frm 文件：存储与表相关的元数据（meta）信息，包括表结构的定义信息等，每一张表都会有一个frm 文件。
  - MYD 文件：MyISAM 存储引擎专用，存放 MyISAM 表的数据（data)，每一张表都会有一个 .MYD 文件。
  - MYI 文件：MyISAM 存储引擎专用，存放 MyISAM 表的索引相关信息，每一张 MyISAM 表对应一个 .MYI 文件。
  - ibd文件和 IBDATA 文件：存放 InnoDB 的数据文件（包括索引）。InnoDB 存储引擎有两种表空间方式：独享表空间和共享表空间。独享表空间使用 .ibd 文件来存放数据，且每一张 InnoDB 表对应一个 .ibd 文件。共享表空间使用 .ibdata 文件，所有表共同使用一个（或多个，自行配置）.ibdata 文件。
  - ibdata1 文件：系统表空间数据文件，存储表元数据、Undo日志等 。
  - ib_logfile0、ib_logfile1 文件：Redo log 日志文件。
- pid 文件

  pid 文件是 mysqld 应用程序在 Unix/Linux 环境下的一个进程文件，和许多其他 Unix/Linux 服务端程序一样，它存放着自己的进程 id。

- socket 文件

  socket 文件也是在 Unix/Linux 环境下才有的，用户在 Unix/Linux 环境下客户端连接可以不通过 TCP/IP 网络而直接使用 Unix Socket 来连接 MySQL。

第3节 MySQL日志系统原理

3.1 Undo Log

- **3.1.1 Undo Log介绍**

  Undo：意为撤销或取消，以撤销操作为目的，返回指定某个状态的操作。

  Undo Log：数据库事务开始之前，会将要修改的记录存放到 Undo 日志里，当事务回滚时或者数据库崩溃时，可以利用 Undo 日志，撤销未提交事务对数据库产生的影响。

  Undo Log产生和销毁：Undo Log在事务开始前产生；事务在提交时，并不会立刻删除undo log，innodb会将该事务对应的undo log放入到删除列表中，后面会通过后台线程purge thread进行回收处理。Undo Log属于逻辑日志，记录一个变化过程。例如执行一个delete，undolog会记录一个insert；执行一个update，undolog会记录一个相反的update。

  Undo Log存储：undo log采用段的方式管理和记录。在innodb数据文件中包含一种rollback segment回滚段，内部包含1024个undo log segment。可以通过下面一组参数来控制Undo log存储。

  ```
  show variables like '%innodb_undo%';
  ```

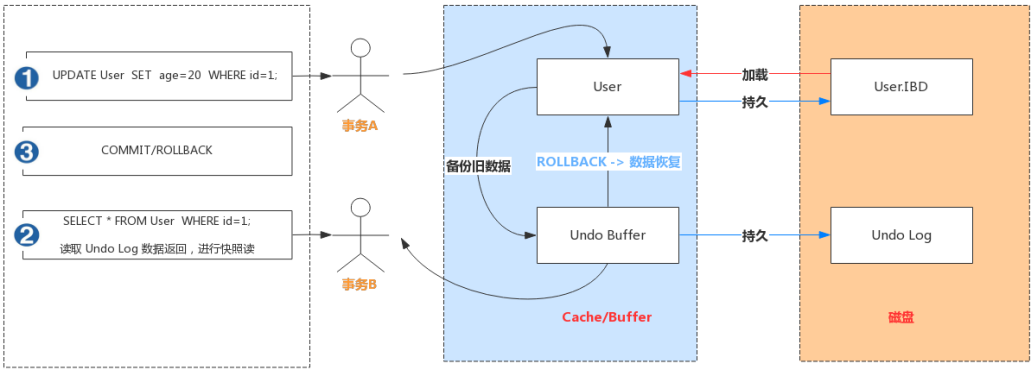
- **3.1.2 Undo Log作用**

- 实现事务的原子性

  Undo Log 是为了实现事务的原子性而出现的产物。事务处理过程中，如果出现了错误或者用户执行了 ROLLBACK 语句，MySQL 可以利用 Undo Log 中的备份将数据恢复到事务开始之前的状态。

- 实现多版本并发控制（MVCC）

Undo Log 在 MySQL InnoDB 存储引擎中用来实现多版本并发控制。事务未提交之前，Undo Log保存了未提交之前的版本数据，Undo Log 中的数据可作为数据旧版本快照供其他并发事务进行快照读。



事务A手动开启事务，执行更新操作，首先会把更新命中的数据备份到 Undo Buffer 中。

事务B手动开启事务，执行查询操作，会读取 Undo 日志数据返回，进行快照读

3.2 Redo Log和Binlog

Redo Log和Binlog是MySQL日志系统中非常重要的两种机制，也有很多相似之处，下面介绍下两者细节和区别。

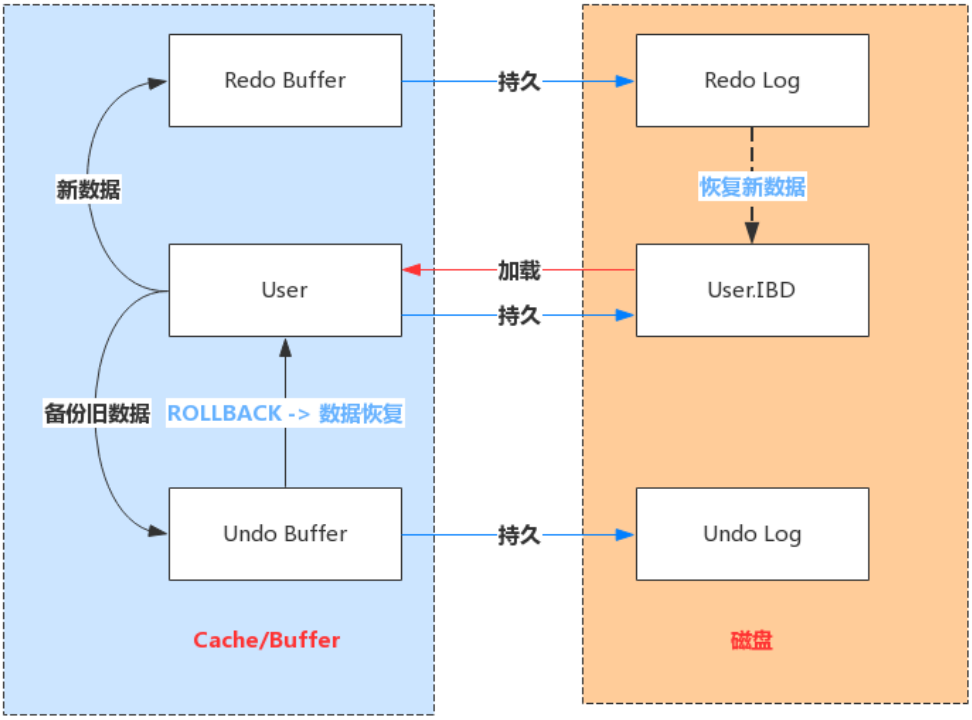
3.2.1 Redo Log日志

- Redo Log介绍

  Redo：顾名思义就是重做。以恢复操作为目的，在数据库发生意外时重现操作。
  Redo Log：指事务中修改的任何数据，将最新的数据备份存储的位置（Redo Log），被称为重做日志。
  Redo Log 的生成和释放：随着事务操作的执行，就会生成Redo Log，在事务提交时会将产生Redo Log写入Log Buffer，并不是随着事务的提交就立刻写入磁盘文件。等事务操作的脏页写入到磁盘之后，Redo Log 的使命也就完成了，Redo Log占用的空间就可以重用（被覆盖写入）。

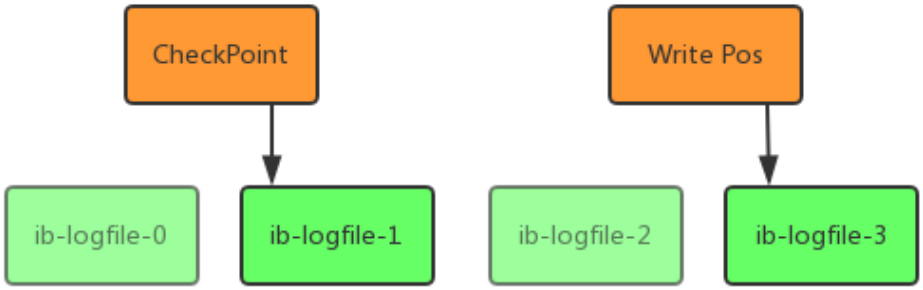
- Redo Log工作原理

  Redo Log 是为了实现事务的持久性而出现的产物。防止在发生故障的时间点，尚有脏页未写入表的 IBD 文件中，在重启 MySQL 服务的时候，根据 Redo Log 进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。



- Redo Log写入机制

  Redo Log 文件内容是以顺序循环的方式写入文件，写满时则回溯到第一个文件，进行覆盖写。



  如图所示：

  - write pos 是当前记录的位置，一边写一边后移，写到最后一个文件末尾后就回到 0 号文件开头；
  - checkpoint 是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件；

  write pos 和 checkpoint 之间还空着的部分，可以用来记录新的操作。如果 write pos 追上 checkpoint，表示写满，这时候不能再执行新的更新，得停下来先擦掉一些记录，把 checkpoint 推进一下。

- Redo Log相关配置参数

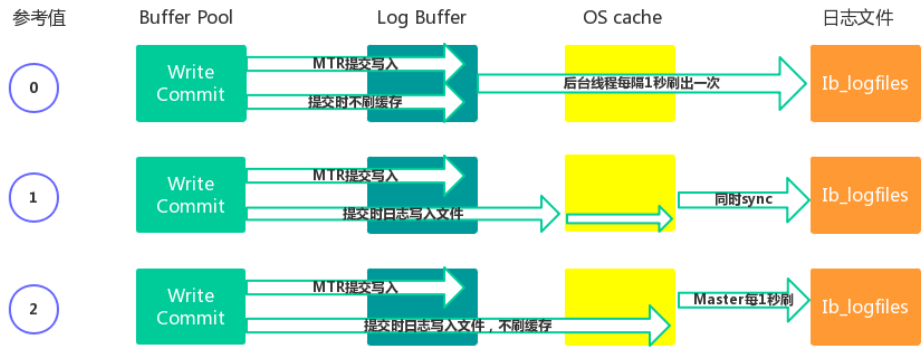
  每个InnoDB存储引擎至少有1个重做日志文件组（group），每个文件组至少有2个重做日志文件，默认为ib_logfile0和ib_logfile1。可以通过下面一组参数控制Redo Log存储：

```
show variables like '%innodb_log%';
```

Redo Buffer 持久化到 Redo Log 的策略，可通过 Innodb_flush_log_at_trx_commit 设置：

- 0：每秒提交 Redo buffer ->OS cache -> flush cache to disk，可能丢失一秒内的事务数据。由后台Master线程每隔 1秒执行一次操作。
- 1（默认值）：每次事务提交执行 Redo Buffer -> OS cache -> flush cache to disk，最安全，性能最差的方式。
- 2：每次事务提交执行 Redo Buffer -> OS cache，然后由后台Master线程再每隔1秒执行OS cache -> flush cache to disk 的操作。

一般建议选择取值2，因为 MySQL 挂了数据没有损失，整个服务器挂了才会损失1秒的事务提交数据。



3.2.2 Binlog日志

- Binlog记录模式

  Redo Log 是属于InnoDB引擎所特有的日志，而MySQL Server也有自己的日志，即 Binary log（二进制日志），简称Binlog。Binlog是记录所有数据库表结构变更以及表数据修改的二进制日志，不会记录SELECT和SHOW这类操作。Binlog日志是以事件形式记录，还包含语句所执行的消耗时间。开启Binlog日志有以下两个最重要的使用场景。

  - 主从复制：在主库中开启Binlog功能，这样主库就可以把Binlog传递给从库，从库拿到Binlog后实现数据恢复达到主从数据一致性。
  - 数据恢复：通过mysqlbinlog工具来恢复数据。

  Binlog文件名默认为“主机名_binlog-序列号”格式，例如oak_binlog-000001，也可以在配置文件中指定名称。文件记录模式有STATEMENT、ROW和MIXED三种，具体含义如下。

  - ROW（row-based replication, RBR）：日志中会记录每一行数据被修改的情况，然后在slave端对相同的数据进行修改。

    优点：能清楚记录每一个行数据的修改细节，能完全实现主从数据同步和数据的恢复。

    缺点：批量操作，会产生大量的日志，尤其是alter table会让日志暴涨。

  - STATMENT（statement-based replication, SBR）：每一条被修改数据的SQL都会记录到master的Binlog中，slave在复制的时候SQL进程会解析成和原来master端执行过的相同的SQL再次执行。简称SQL语句复制。

    优点：日志量小，减少磁盘IO，提升存储和恢复速度

    缺点：在某些情况下会导致主从数据不一致，比如last_insert_id()、now()等函数。

  - MIXED（mixed-based replication, MBR）：以上两种模式的混合使用，一般会使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择写入模式。

- Binlog文件结构

MySQL的binlog文件中记录的是对数据库的各种修改操作，用来表示修改操作的数据结构是Log event。不同的修改操作对应的不同的log event。比较常用的log event有：Query event、Row event、Xid event等。binlog文件的内容就是各种Log event的集合。

Binlog文件中Log event结构如下图所示：

- Binlog写入机制
  - 根据记录模式和操作触发event事件生成log event（事件触发执行机制）
  - 将事务执行过程中产生log event写入缓冲区，每个事务线程都有一个缓冲区

    Log Event保存在一个binlog_cache_mngr数据结构中，在该结构中有两个缓冲区，一个是stmt_cache，用于存放不支持事务的信息；另一个是trx_cache，用于存放支持事务的信息。
  - 事务在提交阶段会将产生的log event写入到外部binlog文件中。

    不同事务以串行方式将log event写入binlog文件中，所以一个事务包含的log event信息在binlog文件中是连续的，中间不会插入其他事务的log event。
- Binlog文件操作
  - Binlog状态查看

    ```
    show variables like 'log_bin';
    ```
  - 开启Binlog功能

    ```
    mysql> set global log_bin=mysqllogbin;
    ERROR 1238 (HY000): Variable 'log_bin' is a read only variable
    ```

    需要修改my.cnf或my.ini配置文件，在[mysqld]下面增加log_bin=mysql_bin_log，重启MySQL服务。

```
#log-bin=ON
#log-bin-basename=mysqlbinlog
binlog-format=ROW
log-bin=mysqlbinlog
```

- 使用show binlog events命令

```
show binary logs; //等价于show master logs;
show master status;
show binlog events;
show binlog events in 'mysqlbinlog.000001';
```

- 使用mysqlbinlog 命令

```
mysqlbinlog "文件名"
mysqlbinlog "文件名" > "test.sql"
```

- 使用 binlog 恢复数据

```
//按指定时间恢复
mysqlbinlog --start-datetime="2020-04-25 18:00:00" --stop-
datetime="2020-04-26 00:00:00" mysqlbinlog.000002 | mysql -uroot -p1234
//按事件位置号恢复
mysqlbinlog --start-position=154 --stop-position=957 mysqlbinlog.000002
| mysql -uroot -p1234
```

mysqldump：定期全部备份数据库数据。mysqlbinlog可以做增量备份和恢复操作。

- 删除Binlog文件

```
purge binary logs to 'mysqlbinlog.000001'; //删除指定文件
purge binary logs before '2020-04-28 00:00:00'; //删除指定时间之前的文件
reset master; //清除所有文件
```

可以通过设置expire_logs_days参数来启动自动清理功能。默认值为0表示没启用。设置为1表示超出1天binlog文件会自动删除掉。

- Redo Log和Binlog区别
  - Redo Log是属于InnoDB引擎功能，Binlog是属于MySQL Server自带功能，并且是以二进制文件记录。
  - Redo Log属于物理日志，记录该数据页更新状态内容，Binlog是逻辑日志，记录更新过程。
  - Redo Log日志是循环写，日志空间大小是固定，Binlog是追加写入，写完一个写下一个，不会覆盖使用。
  - Redo Log作为服务器异常宕机后事务数据自动恢复使用，Binlog可以作为主从复制和数据恢复使用。Binlog没有自动crash-safe能力。

第二天 MySQL分库分表实战方案

第1节 背景描述

- 刚开始我们的系统只用了**单机数据库**
- 随着用户的不断增多，考虑到系统的高可用和越来越多的用户请求，我们开始使用数据库**主从架构**
- 当用户量级和业务进一步提升后，写请求越来越多，这时我们开始使用了**分库分表**

第2节 遇到的问题

- 用户请求量太大

  单服务器TPS、内存、IO都是有上限的，需要将请求打散分布到多个服务器

- 单库数据量太大

  单个数据库处理能力有限；单库所在服务器的磁盘空间有限；单库上的操作IO有瓶颈

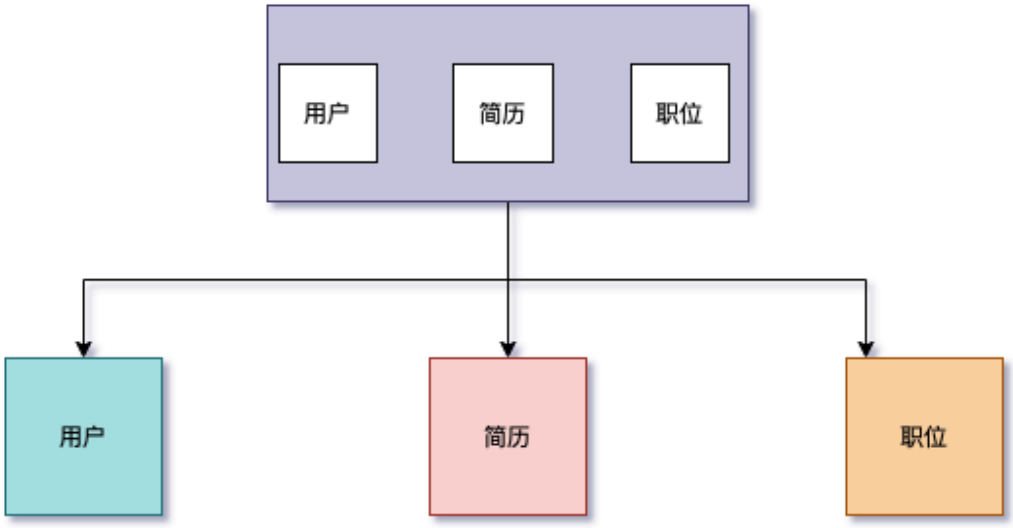
- 单表数据量太大

  查询、插入、更新操作都会变慢，在加字段、加索引、机器迁移都会产生高负载，影响服务

第3节 如何解决

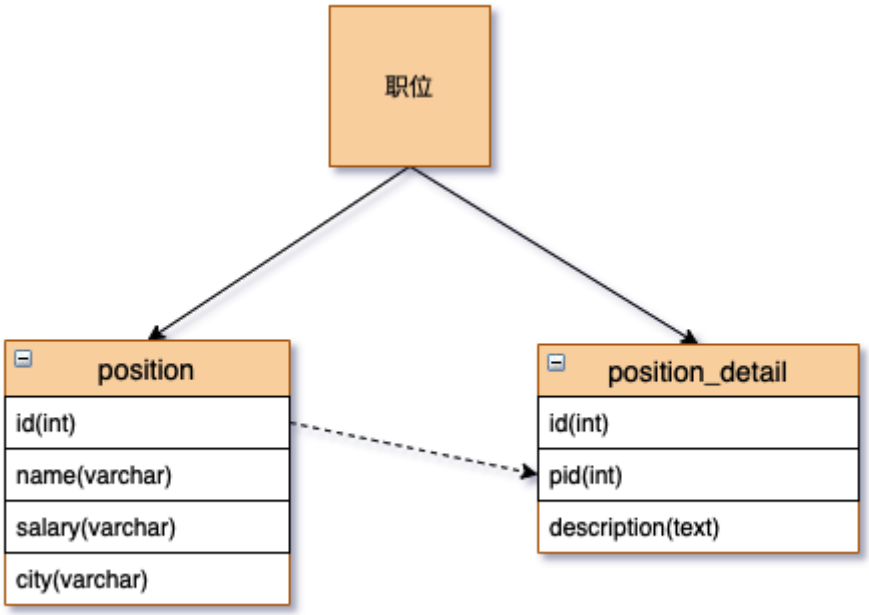
- 垂直拆分
  - 垂直分库

    微服务架构时，业务切割得足够独立，数据也会按照业务切分，保证业务数据隔离，大大提升了数据库的吞吐能力



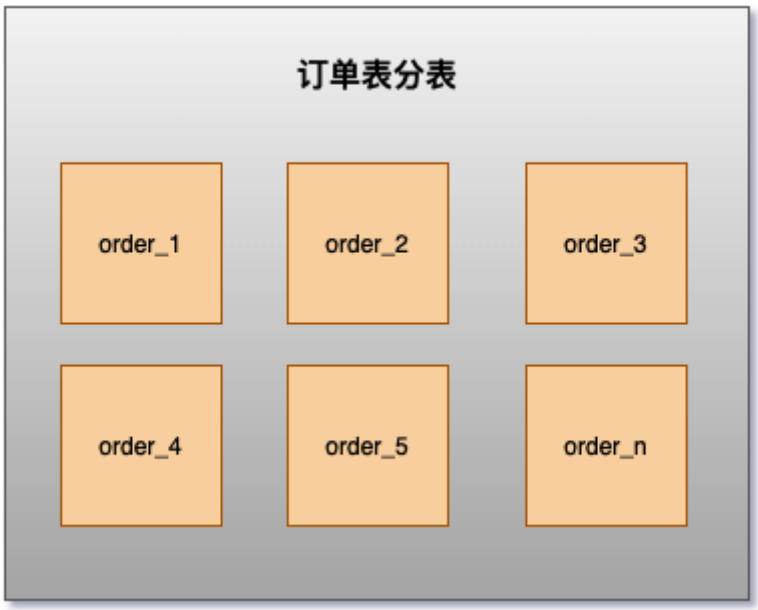
  - 垂直分表

    表中字段太多且包含大字段的时候，在查询时对数据库的IO、内存会受到影响，同时更新数据时，产生的binlog文件会很大，MySQL在主从同步时也会有延迟的风险



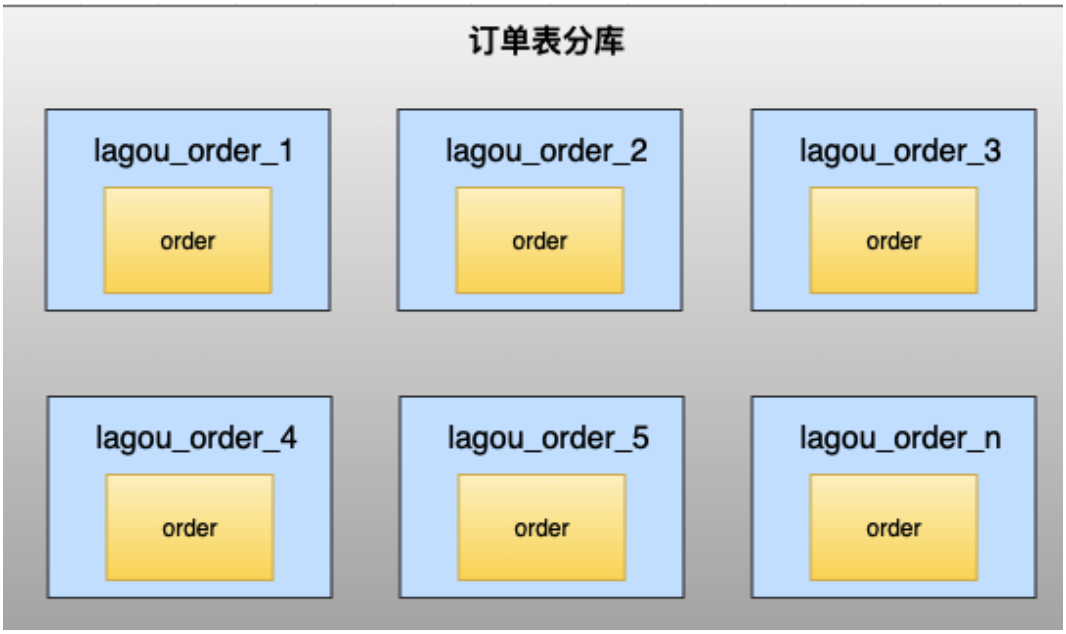
- 水平拆分
  - 水平分表

    针对数据量巨大的单张表（比如订单表），按照规则把一张表的数据切分到多张表里面去。但是这些表还是在同一个库中，所以库级别的数据库操作还是有IO瓶颈。



  - 水平分库

    将单张表的数据切分到多个服务器上去，每个服务器具有相应的库与表，只是表中数据集合不同。 水平分库分表能够有效的缓解单机和单库的性能瓶颈和压力，突破IO、连接数、硬件资源等的瓶颈



- 水平分库规则

  不跨库、不跨表，保证同一类的数据都在同一个服务器上面。

  数据在切分之前，需要考虑如何高效的进行数据获取，如果每次查询都要跨越多个节点，就需要谨慎使用。

- 水平分表规则
  - RANGE
    - 时间：按照年、月、日去切分。例如order_2020、order_202005、order_20200501
    - 地域：按照省或市去切分。例如order_beijing、order_shanghai、order_chengdu
    - 大小：从0到1000000一个表。例如1000001-2000000放一个表，每100万放一个表
  - HASH
    - 用户ID取模

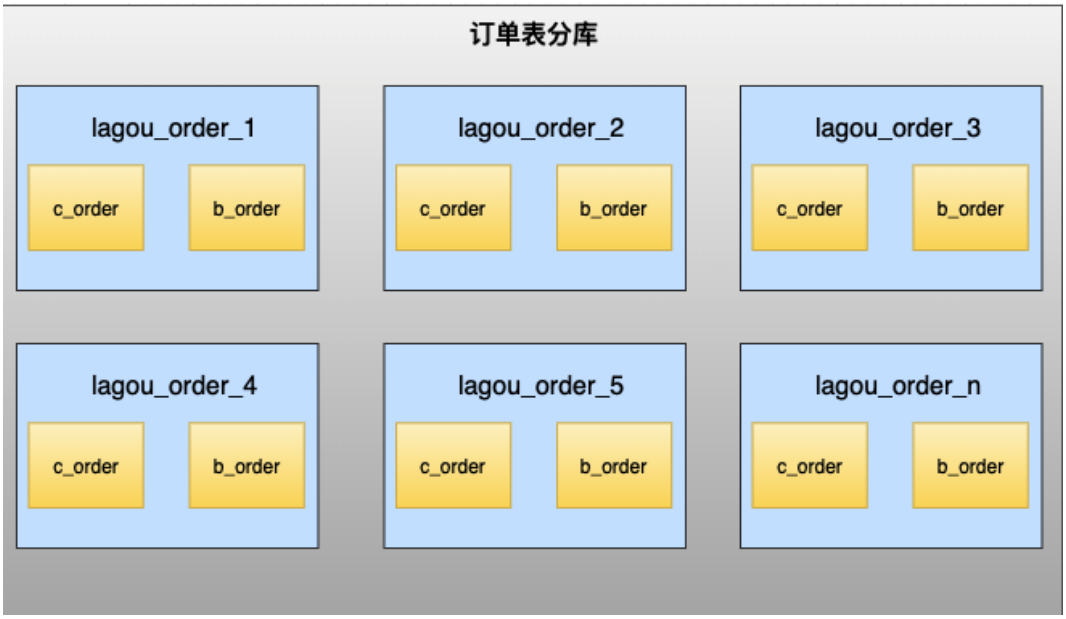
      不同的业务使用的切分规则是不一样，就上面提到的切分规则，举例如下：
  - 站内信
    - 用户维度：用户只能看到发送给自己的消息，其他用户是不可见的，这种情况下是按照用户ID hash分库，在用户查看历史记录翻页查询时，所有的查询请求都在同一个库内
  - 用户表
    - 范围法：以用户ID为划分依据，将数据水平切分到两个数据库实例，如：1到1000W在一张表，1000W到2000W在一张表，这种情况会出现单表的负载较高
    - 按照用户ID HASH尽量保证用户数据均衡分到数据库中

      > 如果在登录场景下，用户输入手机号和验证码进行登录，这种情况下，登录时是不是需要扫描所有分库的信息？
      >
      > 最终方案：用户信息采用ID做切分处理，同时存储用户ID和手机号的映射的关系表（新增一个关系表），关系表采用手机号进行切分。可以通过关系表根据手机号查询到对应的ID，再定位用户信息。
  - 流水表
    - 时间维度：可以根据每天新增的流水来判断，选择按照年份分库，还是按照月份分库，甚至也可以按照日期分库
  - 订单表

在拉勾网，求职者（下面统称C端用户）投递企业（下面统称B端用户）的职位产生的记录称之为订单表。在线上的业务场景中，C端用户看自己的投递记录，每次的投递到了哪个状态，B端用户查看自己收到的简历，对于合适的简历会进行下一步沟通，同一个公司内的员工可以协作处理简历。

如何能同时满足C端和B端对数据查询，不进行跨库处理？

最终方案：为了同时满足两端用户的业务场景，采用空间换时间，将一次的投递记录存为两份，C端的投递记录以用户ID为分片键，B端收到的简历按照公司ID为分片键

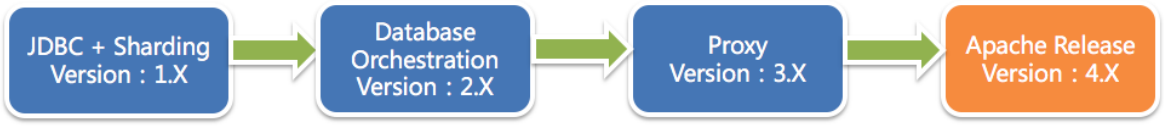


第三天 MySQL分库分表编程实战

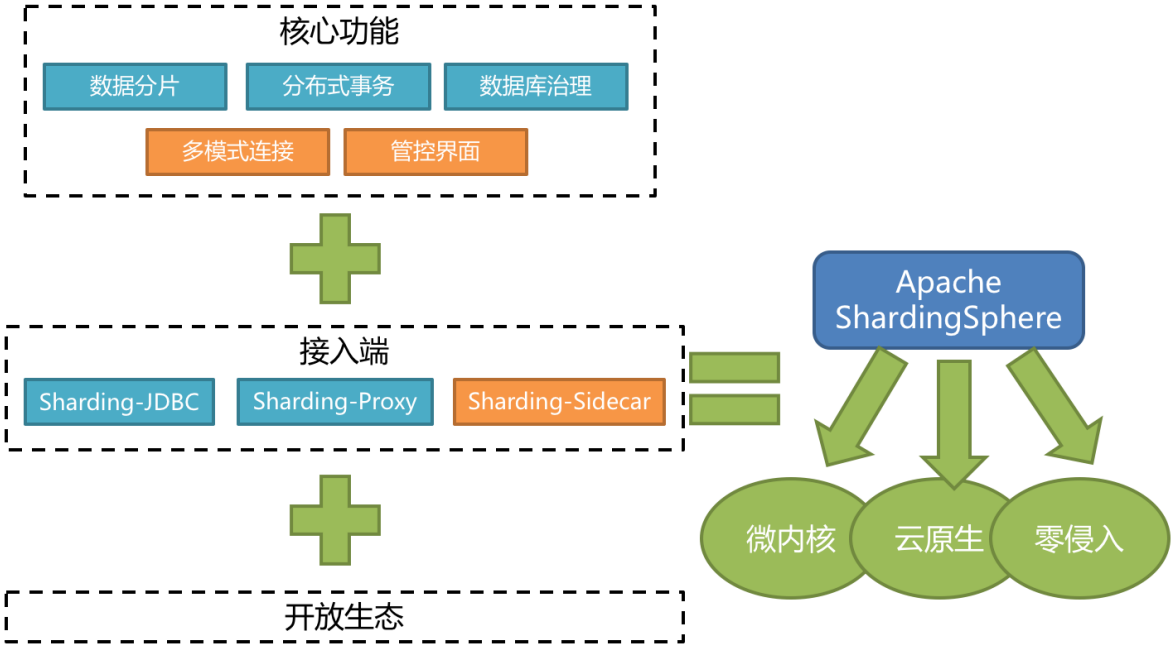
第1节 ShardingSphere介绍

Apache ShardingSphere是一款开源的分布式数据库中间件组成的生态圈。它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（规划中）这3款相互独立的产品组成。 他们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、容器、云原生等各种多样化的应用场景。

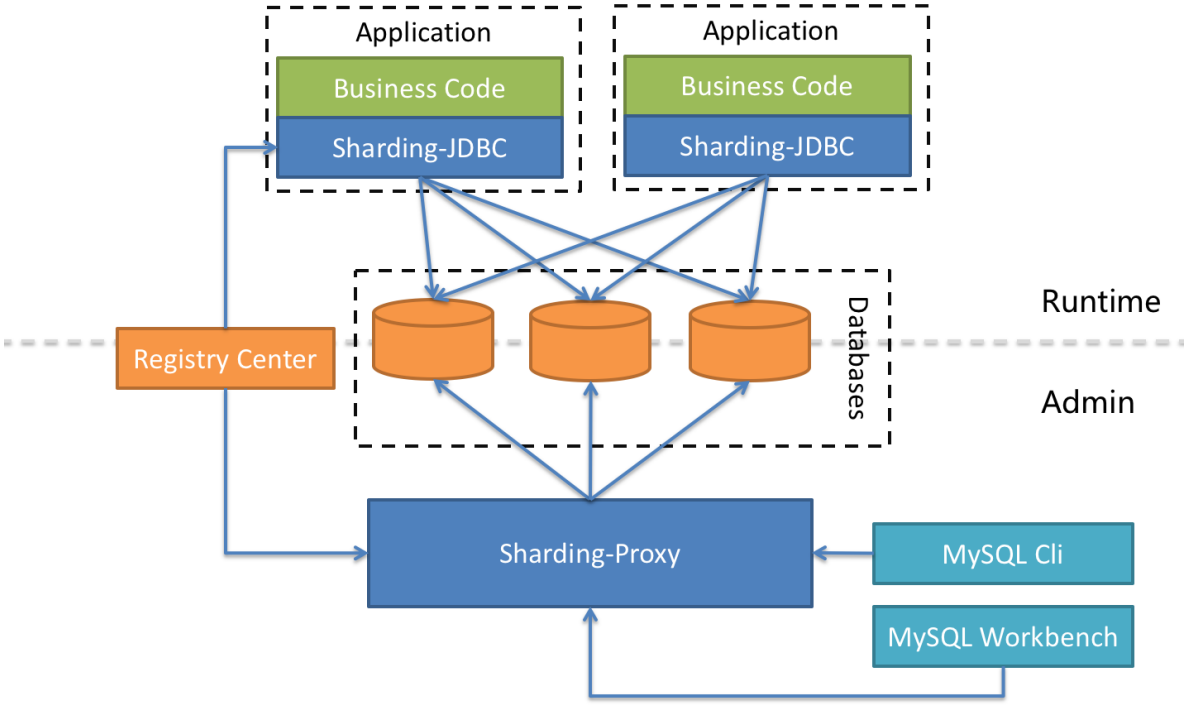
ShardingSphere项目状态如下：



ShardingSphere定位为关系型数据库中间件，旨在充分合理地在分布式的场景下利用关系型数据库的计算和存储能力，而并非实现一个全新的关系型数据库。



- Sharding-JDBC：被定位为轻量级Java框架，在Java的JDBC层提供的额外服务，以jar包形式使用。
- Sharding-Proxy：被定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。
- Sharding-Sidecar：被定位为Kubernetes或Mesos的云原生数据库代理，以DaemonSet的形式代理所有对数据库的访问。



Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar三者区别如下：

	Sharding-JDBC	Sharding-Proxy	Sharding-Sidecar
数据库	任意	MySQL	MySQL
连接消耗数	高	低	高
异构语言	仅Java	任意	任意
性能	损耗低	损耗略高	损耗低
无中心化	是	否	是
静态入口	无	有	无

ShardingSphere安装包下载：https://shardingsphere.apache.org/document/current/cn/downloads/

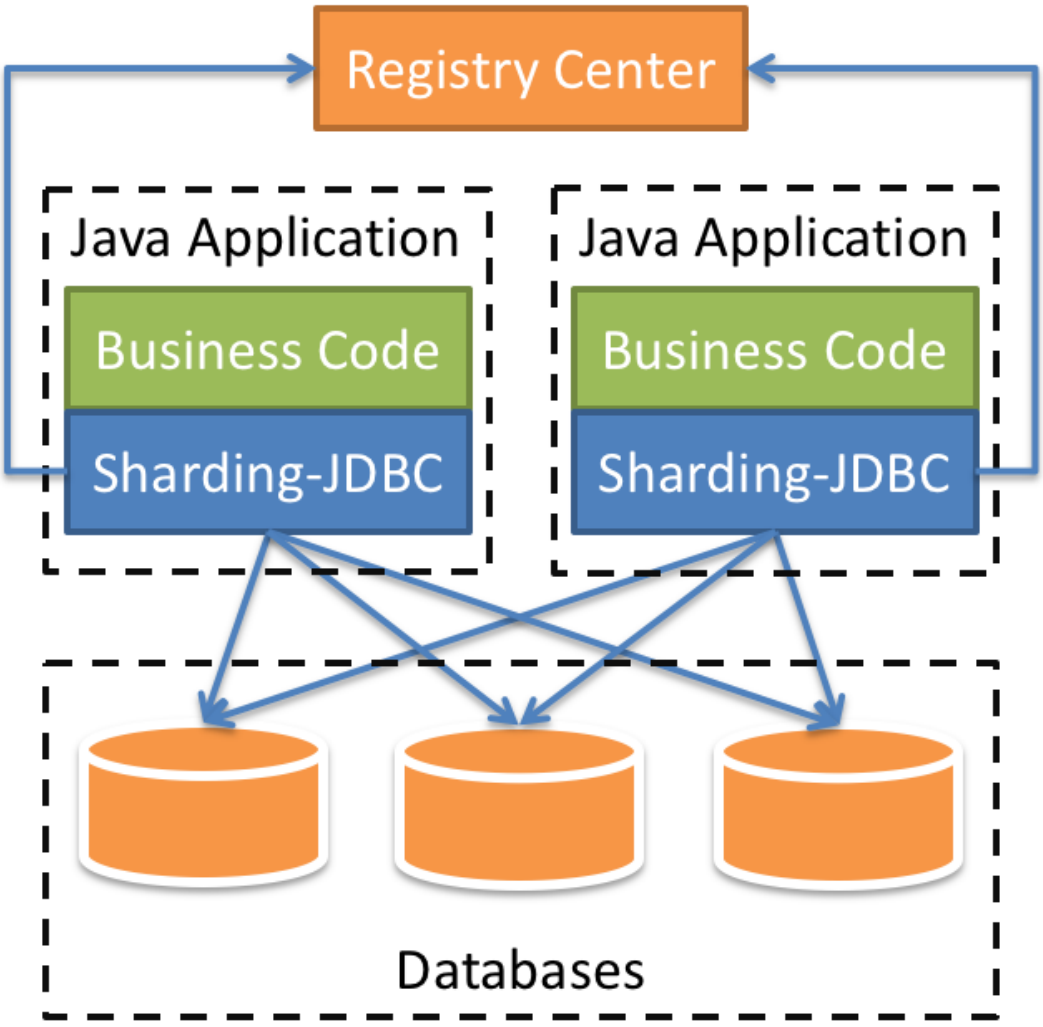
版本	发布日期	源码下载	Sharding-JDBC下载	Sharding-Proxy下载	Sharding-UI 下载	Sharding-Scaling 下载
4.0.1	2020 Mar 9	source (asc sha512)	binary (asc sha512)	binary (asc sha512)	binary (asc sha512)	
4.1.0	2020 Apr 30	source (asc sha512)	binary (asc sha512)	binary (asc sha512)		binary (asc sha512)

使用Git下载工程：git clone https://github.com/apache/incubator-shardingsphere.git

第2节 Sharding-JDBC

Sharding-JDBC定位为轻量级Java框架，在Java的JDBC层提供的额外服务。 它使用客户端直连数据库，以jar包形式提供服务，无需额外部署和依赖，可理解为增强版的JDBC驱动，完全兼容JDBC和各种ORM框架的使用。

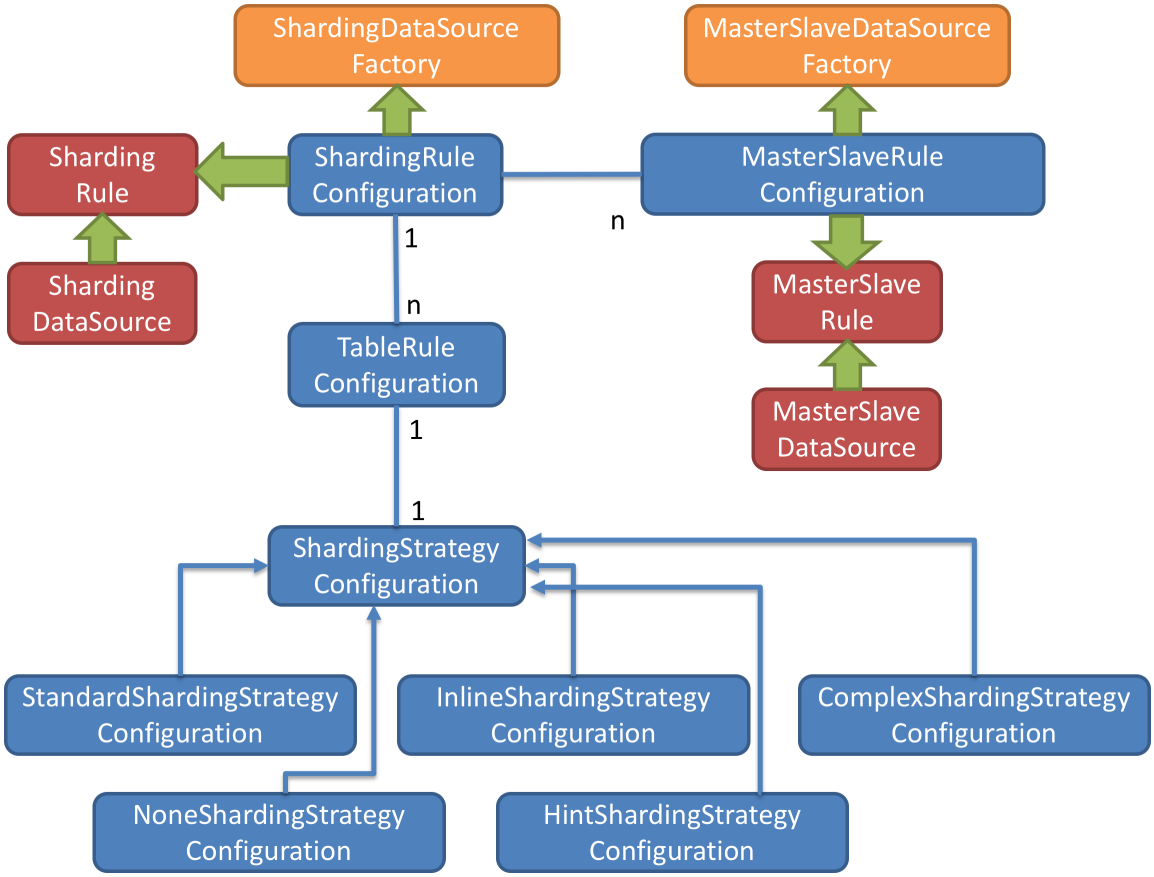
- 适用于任何基于Java的ORM框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template或直接使用JDBC。
- 基于任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, Druid, HikariCP等。
- 支持任意实现JDBC规范的数据库。目前支持MySQL，Oracle，SQLServer和PostgreSQL。



Sharding-JDBC主要功能：

- 数据分片
  - 分库、分表
  - 读写分离
  - 分片策略
  - 分布式主键
- 分布式事务
  - 标准化的事务接口
  - XA强一致性事务
  - 柔性事务
- 数据库治理
  - 配置动态化
  - 编排和治理
  - 数据脱敏
  - 可视化链路追踪

Sharding-JDBC 内部结构：



- 图中黄色部分表示的是Sharding-JDBC的入口API，采用工厂方法的形式提供。 目前有ShardingDataSourceFactory和MasterSlaveDataSourceFactory两个工厂类。
- ShardingDataSourceFactory支持分库分表、读写分离操作
- MasterSlaveDataSourceFactory支持读写分离操作
- 图中蓝色部分表示的是Sharding-JDBC的配置对象，提供灵活多变的配置方式。ShardingRuleConfiguration是分库分表配置的核心和入口，它可以包含多个TableRuleConfiguration和MasterSlaveRuleConfiguration。
- TableRuleConfiguration封装的是表的分片配置信息，有5种配置形式对应不同的Configuration类型。
- MasterSlaveRuleConfiguration封装的是读写分离配置信息。
- 图中红色部分表示的是内部对象，由Sharding-JDBC内部使用，应用开发者无需关注。Sharding-JDBC通过ShardingRuleConfiguration和MasterSlaveRuleConfiguration生成真正供ShardingDataSource和MasterSlaveDataSource使用的规则对象。ShardingDataSource和MasterSlaveDataSource实现了DataSource接口，是JDBC的完整实现方案。

Sharding-JDBC初始化流程：

- 根据配置的信息生成Configuration对象
- 通过Factory会将Configuration对象转化为Rule对象
- 通过Factory会将Rule对象与DataSource对象封装
- Sharding-JDBC使用DataSource进行分库分表和读写分离操作

Sharding-JDBC 使用过程：

- 引入maven依赖

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

注意: 请将${latest.release.version}更改为实际的版本号。

- 规则配置

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

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

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

第3节 数据分片剖析实战

3.1 核心概念

- 表概念
  - 真实表

    数据库中真实存在的物理表。例如b_order0、b_order1
  - 逻辑表

    在分片之后，同一类表结构的名称（总成）。例如b_order。
  - 数据节点

    在分片之后，由数据源和数据表组成。例如ds0.b_order1
  - 绑定表

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

    ```
    b_order：b_order0、b_order1
    b_order_item：b_order_item0、b_order_item1

    select * from b_order o join b_order_item i on(o.order_id=i.order_id)
    where o.order_id in (10,11);
    ```

    如果不配置绑定表关系，采用笛卡尔积关联，会生成4个SQL。

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

select * from b_order0 o join b_order_item1 i on(o.order_id=i.order_id)
where o.order_id in (10,11);

select * from b_order1 o join b_order_item0 i on(o.order_id=i.order_id)
where o.order_id in (10,11);

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

如果配置绑定表关系，生成2个SQL

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

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

- 广播表

  在使用中，有些表没必要做分片，例如字典表、省份信息等，因为他们数据量不大，而且这种表可能需要与海量数据的表进行关联查询。广播表会在不同的数据节点上进行存储，存储的表结构和数据完全相同。

- 分片算法（ShardingAlgorithm）

  由于分片算法和业务实现紧密相关，因此并未提供内置分片算法，而是通过分片策略将各种场景提炼出来，提供更高层级的抽象，并提供接口让应用开发者自行实现分片算法。目前提供4种分片算法。

  - 精确分片算法PreciseShardingAlgorithm

    用于处理使用单一键作为分片键的=与IN进行分片的场景。

  - 范围分片算法RangeShardingAlgorithm

    用于处理使用单一键作为分片键的BETWEEN AND、>、<、>=、<=进行分片的场景。

  - 复合分片算法ComplexKeysShardingAlgorithm

    用于处理使用多键作为分片键进行分片的场景，多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。

  - Hint分片算法HintShardingAlgorithm

    用于处理使用Hint行分片的场景。对于分片字段非SQL决定，而由其他外置条件决定的场景，可使用SQL Hint灵活的注入分片字段。例：内部系统，按照员工登录主键分库，而数据库中并无此字段。SQL Hint支持通过Java API和SQL注释两种方式使用。

- 分片策略（ShardingStrategy）

  分片策略包含分片键和分片算法，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StandardShardingStrategy

    只支持单分片键，提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持。提供PreciseShardingAlgorithm和RangeShardingAlgorithm两个分片算法。

    PreciseShardingAlgorithm是必选的，RangeShardingAlgorithm是可选的。但是SQL中使用了范围操作，如果不配置RangeShardingAlgorithm会采用全库路由扫描，效率低。

- 复合分片策略ComplexShardingStrategy

  支持多分片键。提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持。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InlineShardingStrategy

  只支持单分片键。使用Groovy的表达式，提供对SQL语句中的=和IN的分片操作支持，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。

- Hint分片策略HintShardingStrategy

  通过Hint指定分片值而非从SQL中提取分片值的方式进行分片的策略。

- 不分片策略NoneShardingStrategy

  不分片的策略。

- 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，两种策略的API完全相同。

  - 数据源分片策略
    用于配置数据被分配的目标数据源。

  - 表分片策略

    用于配置数据被分配的目标表，由于表存在与数据源内，所以表分片策略是依赖数据源分片策略结果的。

3.2 流程剖析

ShardingSphere 3个产品的数据分片功能主要流程是完全一致的，如下图所示。

```
![](images/mysql-5-1.png)
```

- SQL解析

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

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

  - MySQL解析器
  - Oracle解析器
  - SQLServer解析器
  - PostgreSQL解析器
  - 默认SQL解析器

- 查询优化
  负责合并和优化分片条件，如OR等。

- SQL路由

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

- SQL改写

  将SQL改写为在真实数据库中可以正确执行的语句。SQL改写分为正确性改写和优化改写。

- SQL执行

  通过多线程执行器异步执行SQL。

- 结果归并

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

3.3 SQL使用规范

- SQL使用规范
  - 支持项
    - 路由至单数据节点时，目前MySQL数据库100%全兼容，其他数据库完善中。
    - 路由至多数据节点时，全面支持DQL、DML、DDL、DCL、TCL。支持分页、去重、排序、分组、聚合、关联查询（不支持跨库关联）。以下用最为复杂的查询为例：

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

  - 不支持项（路由至多数据节点）
    - 不支持CASE WHEN、HAVING、UNION (ALL)
    - 支持分页子查询，但其他子查询有限支持，无论嵌套多少层，只能解析至第一个包含数据表的子查询，一旦在下层嵌套中再次找到包含数据表的子查询将直接抛出解析异常。

    例如，以下子查询可以支持：

    ```
    SELECT COUNT(*) FROM (SELECT * FROM b_order o)
    ```

    以下子查询不支持：

    ```
    SELECT COUNT(*) FROM (SELECT * FROM b_order o WHERE o.id IN (SELECT id
    FROM b_order WHERE status = ?))
    ```

    简单来说，通过子查询进行非功能需求，在大部分情况下是可以支持的。比如分页、统计总数等；而通过子查询实现业务查询当前并不能支持。
  - 由于归并的限制，子查询中包含聚合函数目前无法支持。
  - 不支持包含schema的SQL。因为ShardingSphere的理念是像使用一个数据源一样使用多数据源，因此对SQL的访问都是在同一个逻辑schema之上。
  - 当分片键处于运算表达式或函数中的SQL时，将采用全路由的形式获取结果。

    例如下面SQL，create_time为分片键：

    ```
    SELECT * FROM b_order WHERE to_date(create_time, 'yyyy-mm-dd') = '2020-
    05-05';
    ```

    由于ShardingSphere只能通过SQL字面提取用于分片的值，因此当分片键处于运算表达式或函数中时，ShardingSphere无法提前获取分片键位于数据库中的值，从而无法计算出真正的分片值。

不支持的SQL示例：

```
  INSERT INTO tbl_name (col1, col2, …) VALUES(1+2, ?, …)    //VALUES语句不支持运算表达式
  INSERT INTO tbl_name (col1, col2, …) SELECT col1, col2, … FROM tbl_name WHERE col3 = ?    //INSERT .. SELECT
  SELECT COUNT(col1) as count_alias FROM tbl_name GROUP BY col1 HAVING count_alias > ?  //HAVING
  SELECT * FROM tbl_name1 UNION SELECT * FROM tbl_name2 //UNION
  SELECT * FROM tbl_name1 UNION ALL SELECT * FROM tbl_name2 //UNION ALL
  SELECT * FROM ds.tbl_name1    //包含schema
  SELECT SUM(DISTINCT col1), SUM(col1) FROM tbl_name    //同时使用普通聚合函数和DISTINCT
  SELECT * FROM tbl_name WHERE to_date(create_time, 'yyyy-mm-dd') = ?   //会导致全路由
```

- 分页查询

  完全支持MySQL和Oracle的分页查询，SQLServer由于分页查询较为复杂，仅部分支持.

  - **性能瓶颈：**

    查询偏移量过大的分页会导致数据库获取数据性能低下，以MySQL为例：

    ```
    SELECT * FROM b_order ORDER BY id LIMIT 1000000, 10
    ```

    这句SQL会使得MySQL在无法利用索引的情况下跳过1000000条记录后，再获取10条记录，其性能可想而知。 而在分库分表的情况下（假设分为2个库），为了保证数据的正确性，SQL会改写为：

    ```
    SELECT * FROM b_order ORDER BY id LIMIT 0, 1000010
    ```

    即将偏移量前的记录全部取出，并仅获取排序后的最后10条记录。这会在数据库本身就执行很慢的情况下，进一步加剧性能瓶颈。 因为原SQL仅需要传输10条记录至客户端，而改写之后的SQL则会传输1,000,010 * 2的记录至客户端。

  - **ShardingSphere的优化：**

    ShardingSphere进行了以下2个方面的优化。

    - 首先，采用流式处理 + 归并排序的方式来避免内存的过量占用。
    - 其次，ShardingSphere对仅落至单节点的查询进行进一步优化。

  - **分页方案优化：**

    由于LIMIT并不能通过索引查询数据，因此如果可以保证ID的连续性，通过ID进行分页是比较好的解决方案：

    ```
    SELECT * FROM b_order WHERE id > 1000000 AND id <= 1000010 ORDER BY id
    ```

    或通过记录上次查询结果的最后一条记录的ID进行下一页的查询：

    ```
    SELECT * FROM b_order WHERE id > 1000000 LIMIT 10
    ```

3.4 其他功能

- Inline行表达式

  InlineShardingStrategy：采用Inline行表达式进行分片的配置。

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

语法格式：

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。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多个${}或$->{}表达式，整个表达式结果会将每个子表达式结果进行笛卡尔(积)组合。例如，以下行表达式：

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

最终会解析为：

```
online_table1, online_table2, online_table3,
offline_table1, offline_table2, offline_table3
```

数据节点配置：

对于均匀分布的数据节点，如果数据结构如下：

```
 db0
  ├── b_order2
    └── b_order1
  db1
    ├── b_order2
    └── b_order1
```

用行表达式可以简化为：

```
  db${0..1}.b_order${1..2}
  或者
  db$->{0..1}.b_order$->{1..2}
```

对于自定义的数据节点，如果数据结构如下：

```
  db0
    ├── b_order0
    └── b_order1
  db1
    ├── b_order2
    ├── b_order3
    └── b_order4
```

用行表达式可以简化为：

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

分片算法配置：

行表达式内部的表达式本质上是一段Groovy代码，可以根据分片键进行计算的方式，返回相应的真实数据源或真实表名称。

```
ds${id % 10}
或者
ds$->{id % 10}
```

结果为：ds0、ds1、ds2... ds9

- 分布式主键

  ShardingSphere不仅提供了内置的分布式主键生成器，例如UUID、SNOWFLAKE，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。

  内置主键生成器：

  - UUID

    采用UUID.randomUUID()的方式产生分布式主键。

  - SNOWFLAKE

    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，生成64bit的长整型数据。

    自定义主键生成器：

    - 自定义主键类，实现ShardingKeyGenerator接口
    - 按SPI规范配置自定义主键类
      在Apache ShardingSphere中，很多功能实现类的加载方式是通过SPI注入的方式完成的。 注意：在resources目录下新建META-INF文件夹，再新建services文件夹，然后新建文件的名字为org.apache.shardingsphere.spi.keygen.ShardingKeyGenerator，打开文件，复制自定义主键类全路径到文件中保存。
    - 自定义主键类应用配置

      ```
      #对应主键字段名
      spring.shardingsphere.sharding.tables.t_book.key-generator.column=id
      #对应主键类getType返回内容
      spring.shardingsphere.sharding.tables.t_book.key-generator.type=LAGOUKEY
      ```