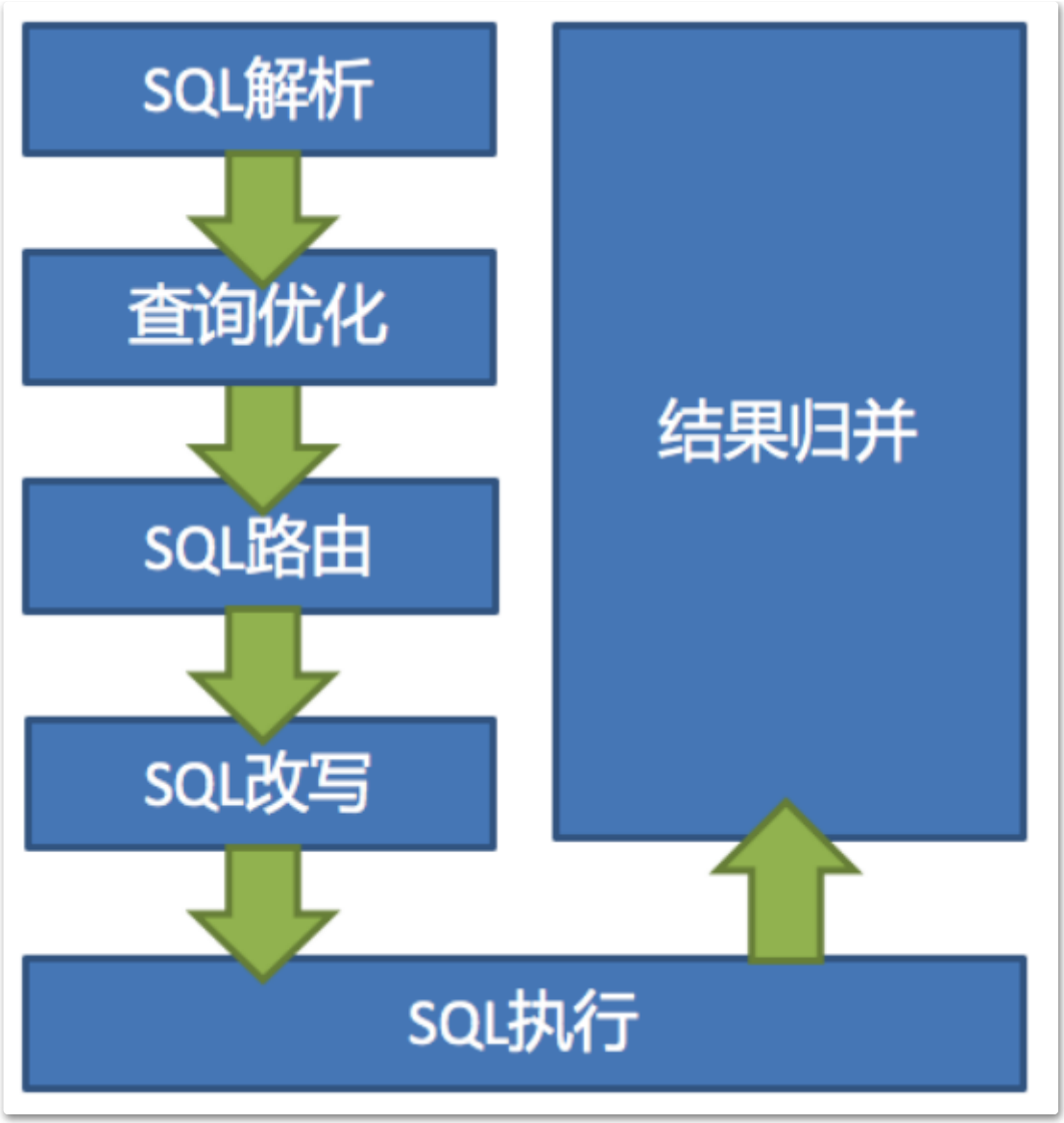
这一部分，我们主要了解ShardingSphere进行分库分表的底层原理，并且深入到源码了解分库分表的实际运行过程。

一方面，我们之前在学习ShardingJDBC时，积累了大量的测试实例，对于学习底层原理是非常好的学习入口。

另一方面，也是为了后面学习ShardingProxy做准备。因为对于ShardingProxy，如果只是学会几个简单的配置和指令，是无法在实际工作中用好的。而ShardingProxy作为一个黑盒产品，要通过ShardingProxy来了解底层原理是比较困难的。

一、内核剖析

ShardingSphere虽然有多个产品，但是他们的数据分片主要流程是完全一致的。



> SQL解析和查询优化都是跟具体的数据库产品有关，在5.x新版本中，被统一进了SQL方言里。

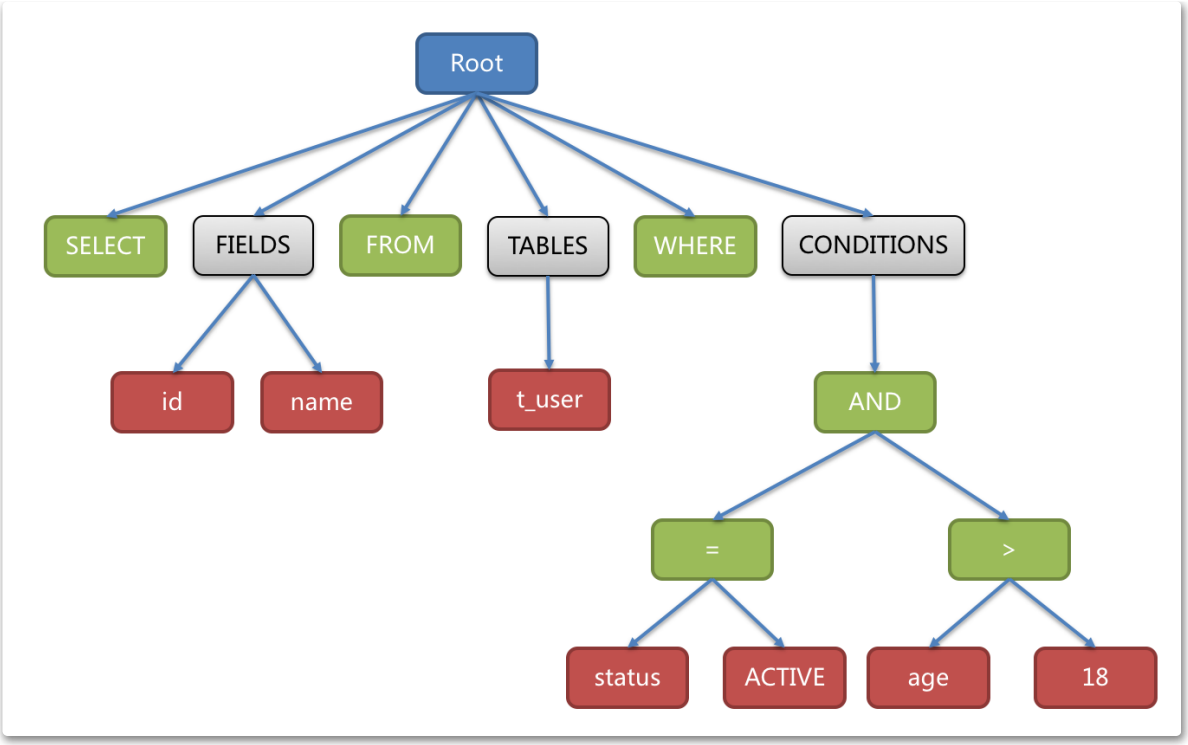
解析引擎

解析过程分为词法解析和语法解析。 词法解析器用于将SQL拆解为不可再分的原子符号，称为Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。 再使用语法解析器将SQL转换为抽象语法树(简称AST，Abstract Syntax Tree)。

例如对下面一条SQL语句：

```
SELECT id, name FROM t_user WHERE status = 'ACTIVE' AND age > 18
```

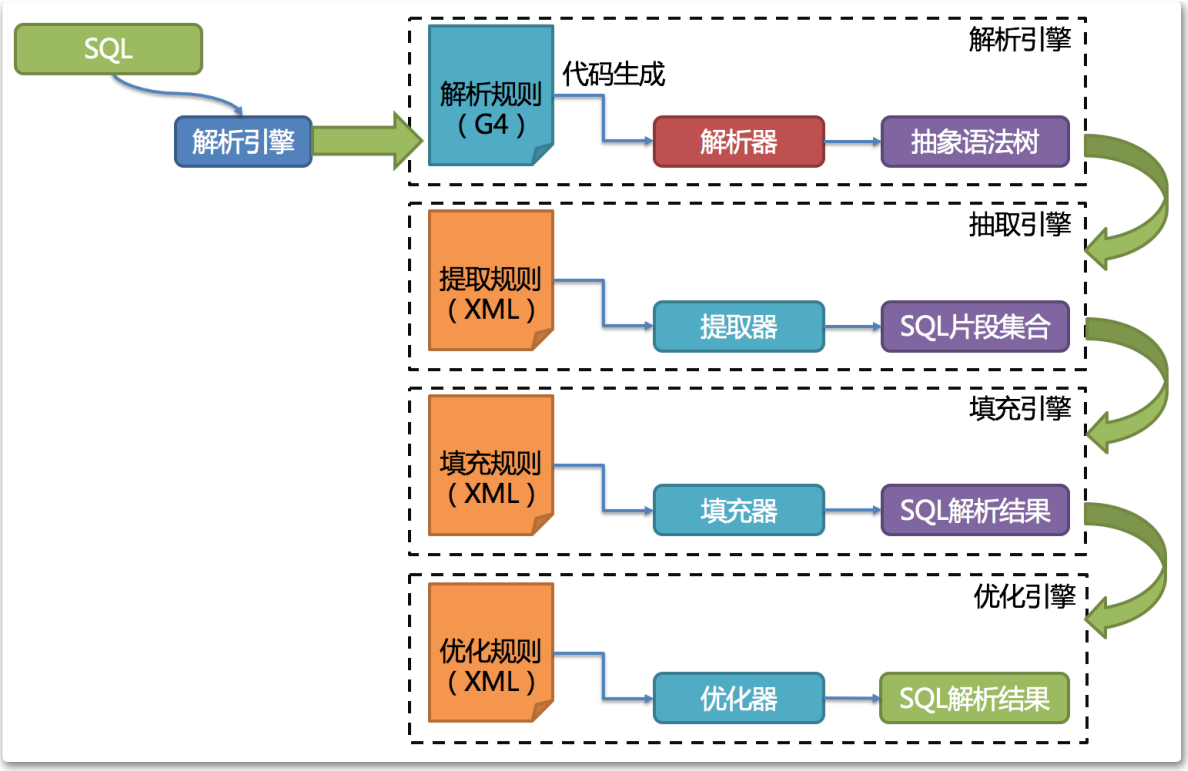
会被解析成下面这样一颗树：



为了便于理解，抽象语法树中的关键字的 Token 用绿色表示，变量的 Token 用红色表示，灰色表示需要
进一步拆分。通过对抽象语法树的遍历，可以标记出所有可能需要改写的位置。
SQL的一次解析过程是不可逆的，所有token按SQL原本的顺序依次进行解析，性能很高。并且在解析过程中，需要考虑各种数据库SQL方言的异同，提供不同的解析模版。

其中，SQL解析是整个分库分表产品的核心，其性能和兼容性是最重要的衡量指标。ShardingSphere在1.4.x之前采用的是性能较快的Druid作为SQL解析器。1.5.x版本后，采用自研的SQL解析器，针对分库分表场景，采取对SQL半理解的方式，提高SQL解析的性能和兼容性。然后从3.0.x版本后，开始使用ANLTR作为SQL解析引擎。这是个开源的SQL解析引擎，ShardingSphere在使用ANLTR时，还增加了一些AST的缓存功能。针对ANLTR4的特性，官网建议尽量采用PreparedStatement的预编译方式来提高SQL执行的性能。

sql解析整体结构：



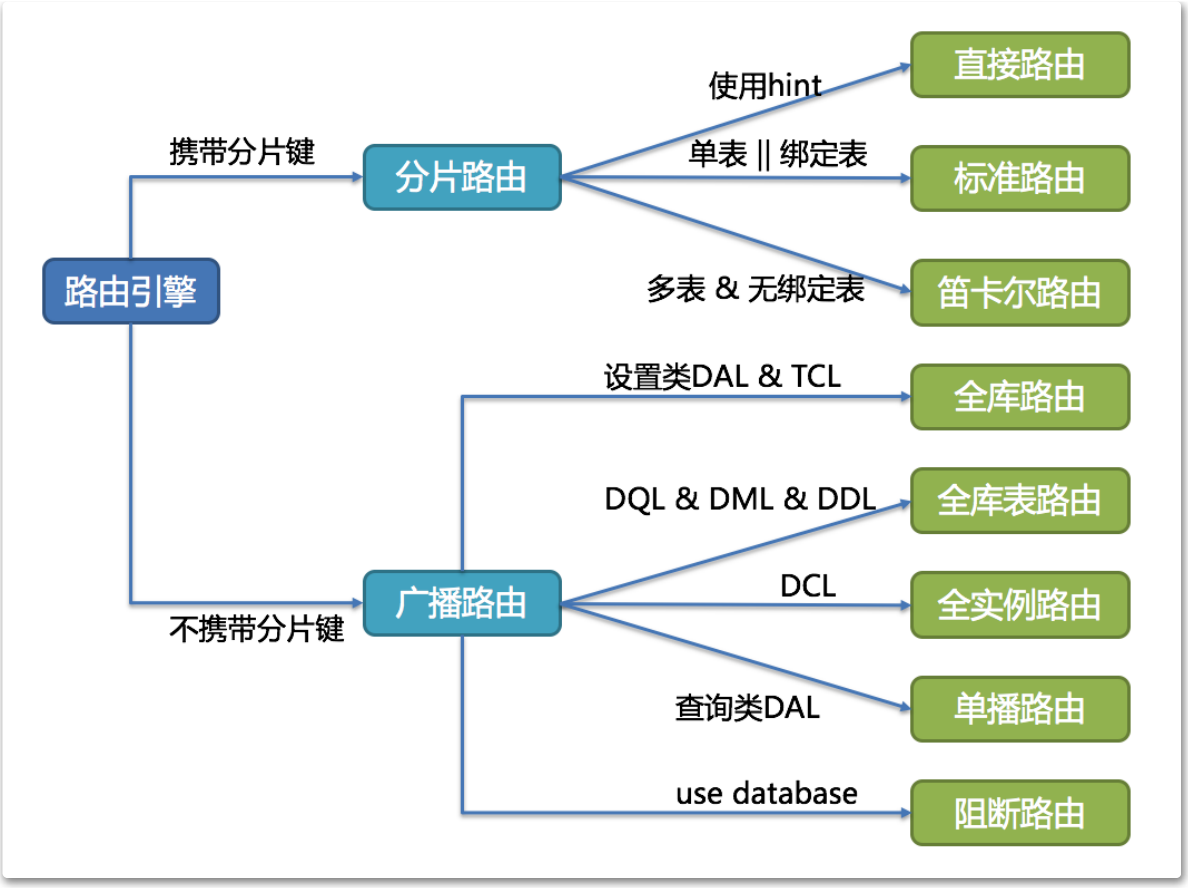
路由引擎

根据解析上下文匹配数据库和表的分片策略，生成路由路径。

ShardingSphere的分片策略主要分为单片路由(分片键的操作符是等号)、多片路由(分片键的操作符是IN)和范围路由(分片键的操作符是Between)。不携带分片键的SQL则是广播路由。

分片策略通常可以由数据库内置也可以由用户方配置。内置的分片策略大致可分为尾数取模、哈希、范围、标签、时间等。 由用户方配置的分片策略则更加灵活，可以根据使用方需求定制复合分片策略。

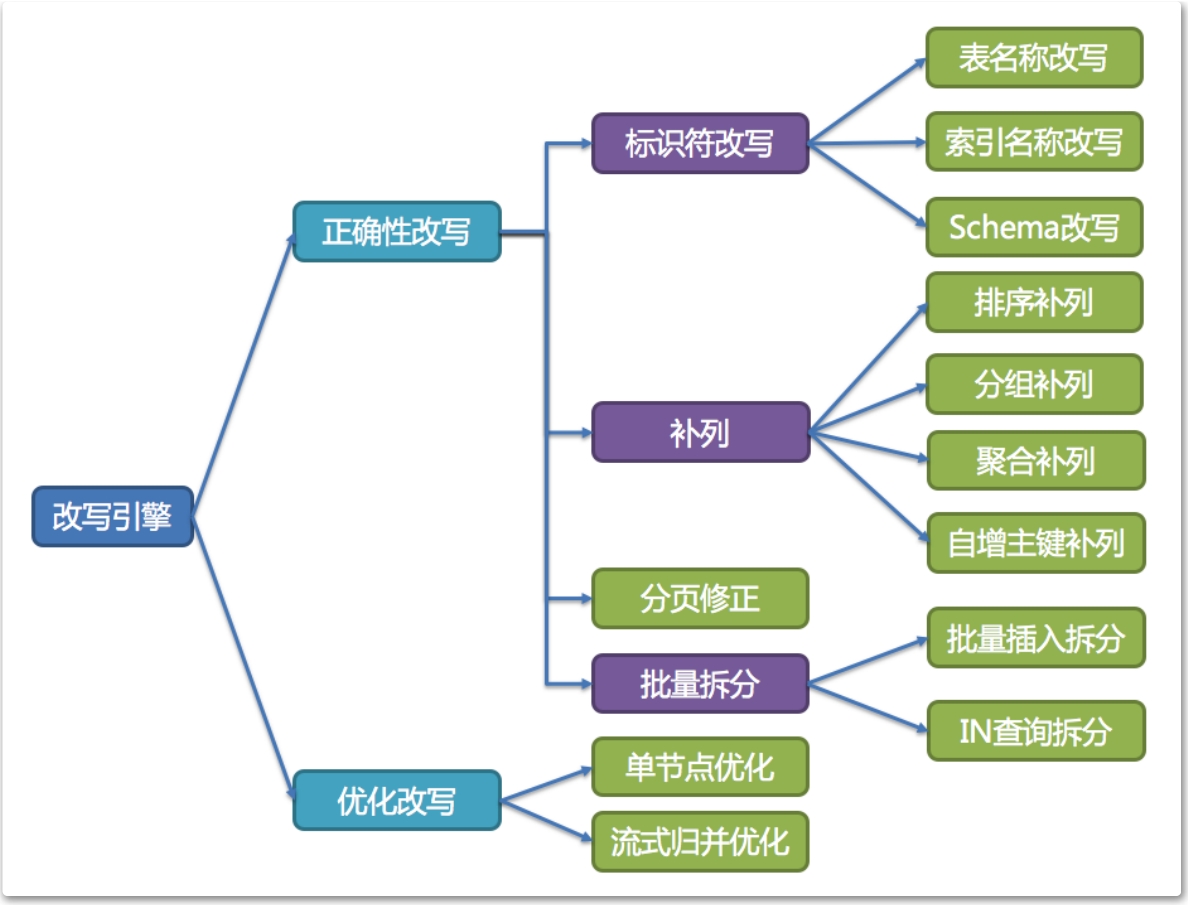
实际使用时，应尽量使用分片路由，明确路由策略。因为广播路由影响过大，不利于集群管理及扩展。



> 全库表路由：对于不带分片键的DQL、DML以及DDL语句，会遍历所有的库表，逐一执行。例如 select * from course 或者 select * from course where ustatus='1'(不带分片键)
>
> 全库路由：对数据库的操作都会遍历所有真实库。 例如 set autocommit=0
>
> 全实例路由：对于DCL语句，每个数据库实例只执行一次，例如 CREATE USER customer@127.0.0.1 identified BY '123';
>
> 单播路由：仅需要从任意库中获取数据即可。 例如 DESCRIBE course
>
> 阻断路由：屏蔽SQL对数据库的操作。例如 USE coursedb。就不会在真实库中执行，因为针对虚拟表操作，不需要切换数据库。

改写引擎

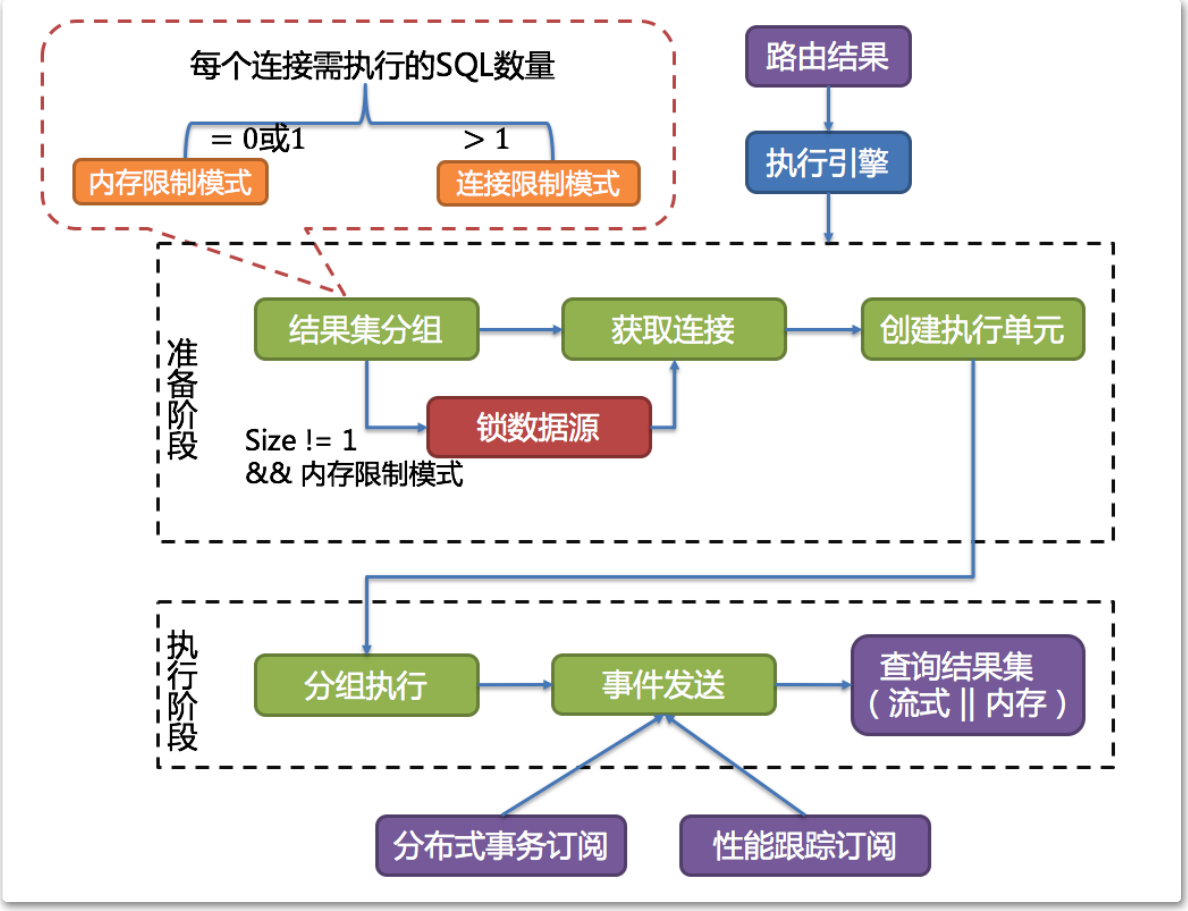
用户只需要面向逻辑库和逻辑表来写SQL，最终由ShardigSphere的改写引擎将SQL改写为在真实数据库中可以正确执行的语句。SQL改写分为正确性改写和优化改写。



执行引擎

ShardingSphere并不是简单的将改写完的SQL提交到数据库执行。执行引擎的目标是自动化的平衡资源控制和执行效率。

例如他的连接模式分为内存限制模式(MEMORY_STRICTLY)和连接限制模式(CONNECTION_STRICTLY)。内存限制模式只关注一个数据库连接的处理数量，通常一张真实表一个数据库连接。而连接限制模式则只关注数据库连接的数量，较大的查询会进行串行操作。



ShardingSphere引入了连接模式的概念，分为内存限制模式(MEMORY_STRICTLY)和连接限制模式(CONNECTION_STRICTLY)。

这两个模式的区分涉及到一个参数spring.shardingsphere.props.**max.connections.size.per.query**=50(默认值1，配置参见源码中ConfigurationPropertyKey类)。ShardingSphere会根据 路由到某一个数据源的路由结果 计算出 所有需在数据库上执行的SQL数量，用这个数量除以 用户的配置项，得到每个数据库连接需执行的SQL数量。数量>1就会选择连接限制模式，数量<=1就会选择内存限制模式。

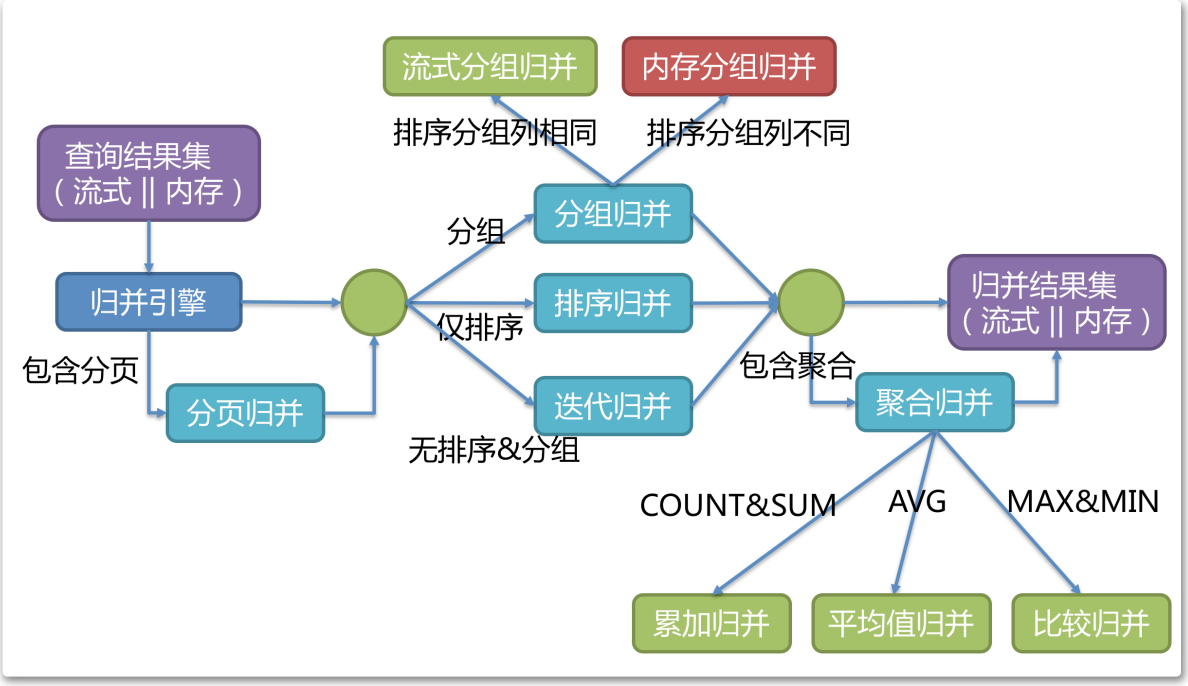
内存限制模式不限制连接数，也就是说会建立多个数据连接，然后并发控制每个连接只去读取一个数据分片的数据。这样可以最快速度的把所有需要的数据读出来。并且在后面的归并阶段，会选择以每一条数据为单位进行归并，就是后面提到的流式归并。这种归并方式归并完一批数据后，可以释放内存了，可以很好的提高数据归并的效率，并且防止出现内存溢出或垃圾回收频繁的情况。他的吞吐量比较大，比较适合OLAP场景。

> 连接限制模式会对连接数进行限制，也即是说至少有一个数据库连接会要去读取多个数据分片的数据。这样他会对这个数据库连接采用串行的方式依次读取多个数据分片的数据。而这种方式下，会将数据全部读入到内存，进行统一的数据归并，也就是后面提到的内存归并。这种方式归并效率会比较高，例如一个MAX归并，直接就能拿到最大值，而流式归并就需要一条条的比较。比较适合OLTP场景。

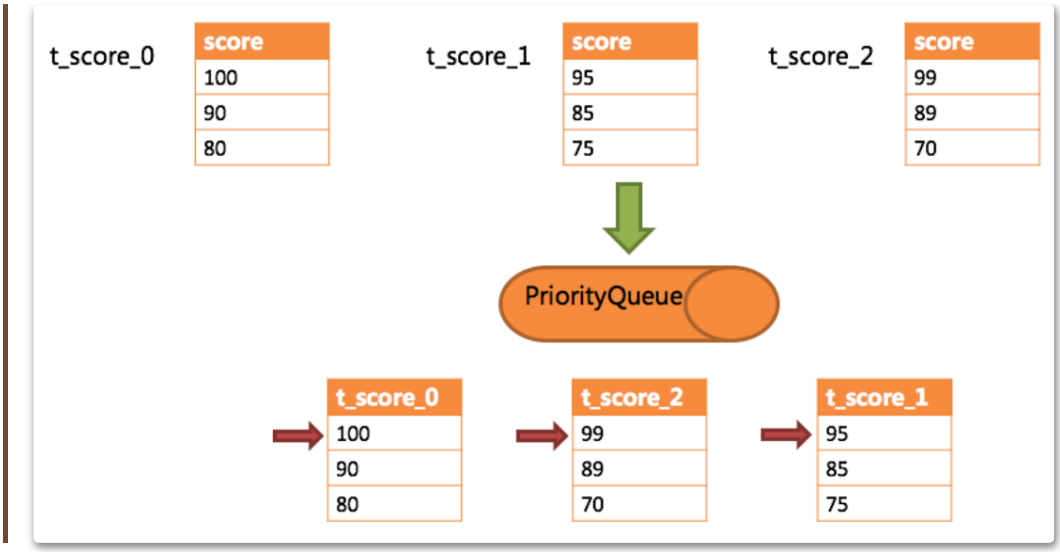
归并引擎

将从各个数据节点获取的多数据结果集，组合成为一个结果集并正确的返回至请求客户端，称为结果归并。

其中，流式归并是指以一条一条数据的方式进行归并，而内存归并是将所有结果集都查询到内存中，进行统一归并。



> 例如： AVG归并就无法直接进行分片归并，需要转化成COUNT&SUM的累加归并，然后再计算平均值。
>
> 排序归并的流程如下图：



分布式主键

内置生成器支持：UUID、SNOWFLAKE，并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。

UUID

采用UUID.randomUUID()的方式产生唯一且不重复的分布式主键。最终生成一个字符串类型的主键。缺点是生成的主键无序。

SNOWFLAKE

雪花算法,能够保证不同进程主键的不重复性，相同进程主键的有序性。二进制形式包含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)

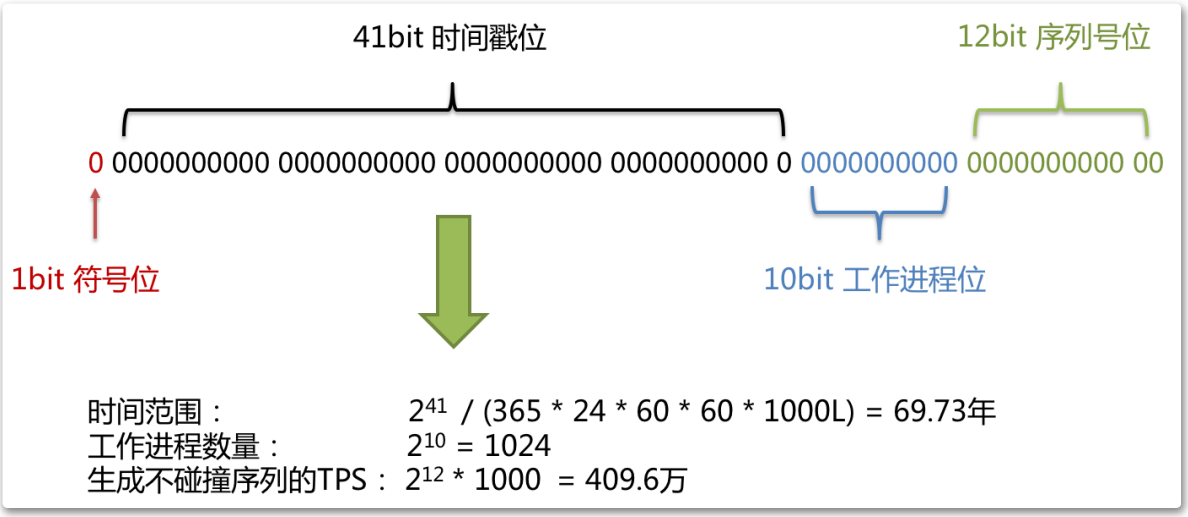
41位的时间戳可以容纳的毫秒数是2的41次幂，一年所使用的毫秒数是：365 * 24 * 60 * 60 * 1000 Math.pow(2, 41) / (365 * 24 * 60 * 60 * 1000L) = 69.73年不重复;

- 工作进程位(10bit)

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

- 序列号位(12bit)

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



优点：

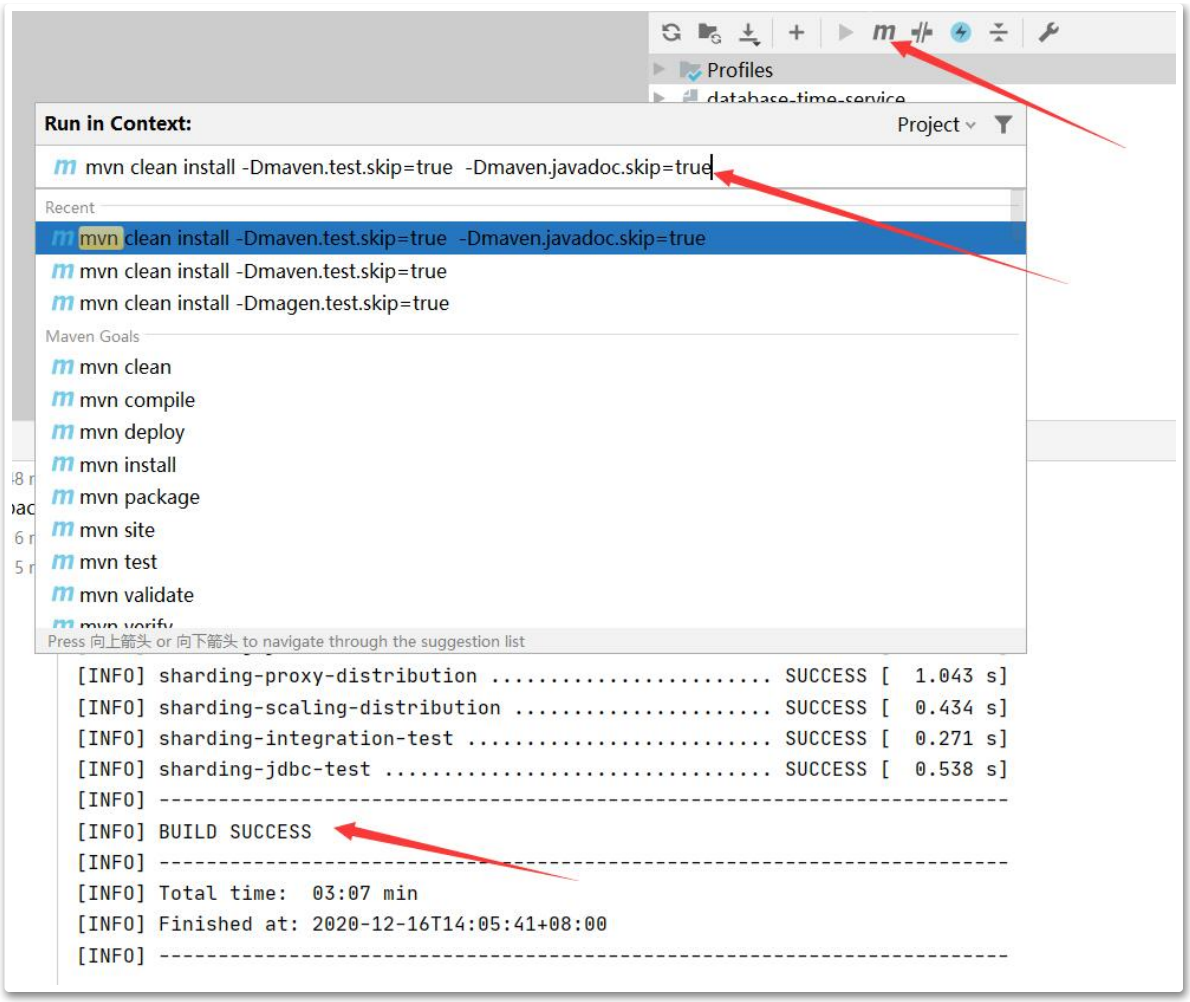
- 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。
- 不依赖第三方组件，稳定性高，生成ID的性能也非常高。
- 可以根据自身业务特性分配bit位，非常灵活

缺点：

强依赖机器时钟，如果机器上时钟回拨，会导致发号重复。

二、源码环境安装

将配套资料中的源码包导入到IDEA后，就可以执行指令mvn clean install -Dmaven.test.skip=true -Dmaven.javadoc.skip=true -Dcheckstyle.skip=true -Drat.numUnapprovedLicenses=100来完成编译。



然后我们的源码调试从ShardingJDBCDemo.java这个测试类开始。这个示例是重现我们之前示例application02.properties中配置的分库分表规则。

> ShardingSphere的分库分表功能，不管是JDBC还是Proxy，最终都是会转化成Java API的配置方式。具体参见官网的配置说明 https://shardingsphere.apache.org/document/legacy/4.x/document/cn/manual/sharding-jdbc/configuration/config-java/

三、ShardingSphere的SPI扩展点

ShardingSphere为了兼容更多的应用场景，在源码中保留了大量的SPI扩展点。所以在看源码之前，需要对JAVA的SPI机制有足够的了解。

1、SPI机制

SPI的全名为：Service Provider Interface。在java.util.ServiceLoader的文档里有比较详细的介绍。

简单的总结下 Java SPI 机制的思想。我们系统里抽象的各个模块，往往有很多不同的实现方案，比如日志模块的方案，xml解析模块、jdbc模块的方案等。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。

一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。

Java SPI 就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要

Java SPI 的具体约定为:当服务的提供者，提供了服务接口的一种实现之后，在jar包的META-INF/services/目录里同时创建一个以服务接口命名的文件。该文件里就是实现该服务接口的具体实现类。

而当外部程序装配这个模块的时候，就能通过该jar包META-INF/services/里的配置文件找到具体的实现类名，并装载实例化，完成模块的注入。

基于这样一个约定就能很好的找到服务接口的实现类，而不需要再代码里制定。jdk提供服务实现查找的一个工具类：java.util.ServiceLoader。

2、ShardingSphere中的SPI扩展点

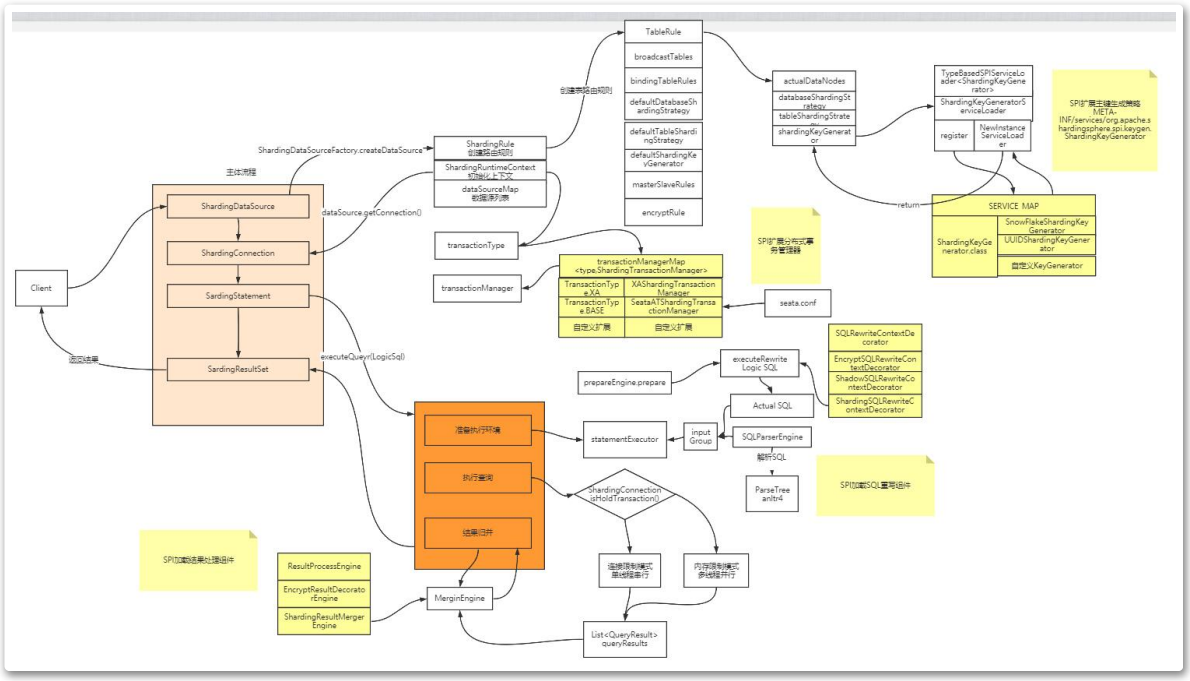
ShardingSphere的开发思想是对源码中主体流程封闭，而对SPI开放。在配套的官方文档《shardingsphere_docs_cn.pdf》的开发者手册部分详细列出了ShardingSphere的所有SPI扩展点。

3、实现自定义主键生成策略

使用ShardingSphere提供的SPI扩展点，实现自定义分布式主键生成策略。参见示例代码。

四、源码大图

配合视频及源码来理解。注意其中的SPI扩展点。



> 有道云笔记链接：
>
> 文档：VIP03-ShardingSphere内核原理以及核心...
> 链接：http://note.youdao.com/noteshare?id=397e945fe2139ac7021c76fa555fcf8f&sub=0B0A51A91DE24681A21956CE38DA162F