一、从分库分表的一个神坑说起
二、分布式主键要考虑哪些问题?
三、主要的主键生成策略
1、数据库策略
2、应用单独生成
3、第三方服务统一生成
4、与第三方结合的segment策略
四、定制雪花算法
1、如影随形的时钟回拨问题
2、用主键生成策略优化分配工作进程位
3、从序列号字段定制雪花算法的连续性
4、根据雪花算法扩展基因分片法
五、从这几个方面来理解CosID
1、单独搭建测试应用
2、SnowFlake雪花算法
1、基础使用
2、重点机制剖析
3、基于JDBC的工作进程ID分发机制实现分析
3、Segment数据段模式
1、基础使用
2、重点机制剖析
3、基于JDBC的ID分发机制实现分析
六、总结

分库分表与分布式主键生成策略

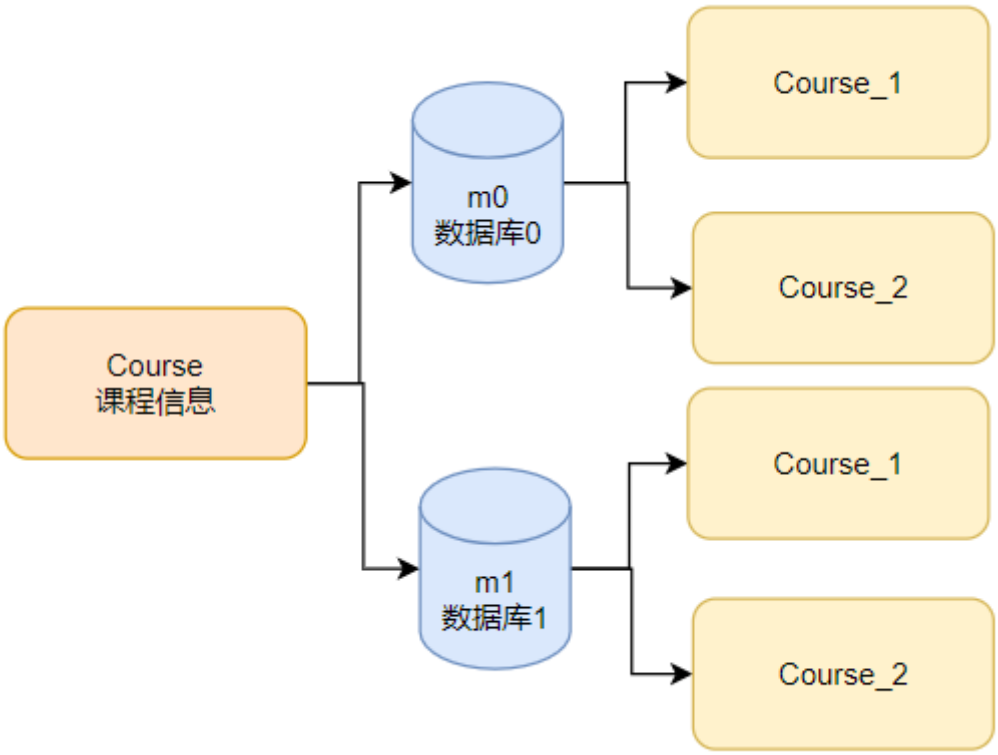
-- 楼兰

主键生成策略，这是一个小问题，但却是一个大世界。很多人开发几十年都没有去想过这个问题，觉得有框架拿来用就是了。但是其实他一点都不简单。正好ShardingSphere5.x版本新集成了一个新的主键生成框架，CosId。这框架功能很强大，性能据说也非常高。只是目前还不太好用，有很多小问题。尤其是与周边生态的版本冲突，简直让人抓狂。官方资料也比较少。因此，如果你想要在大型项目中真正用上CosID，那么，我的建议是，最好先确定自己对于CosID有几把刷子，要不然，就等着被各种莫名其妙的错误折磨得死去活来把。

这里就正好借着梳理CosID的机会，好好把分布式主键生成策略给梳理一下。相信把基础思想梳理清楚了之后，再去了解CosId就比较简单了。

一、从分库分表的一个神坑说起

直接讲问题，或许你会觉得我小题大做。那我们不多啰嗦，从一个分库分表的小实验开始。例如，我想要将一个表的数据分到两个库中的两个表，共四个分片。这应该是分库分表中最为典型的一个场景了。



Course课程信息按照cid字段进行分片，那么分库的算法可以简单设置为按cid奇偶拆分，定制算法m$->{cid%2}就行了。而分表的算法呢？如果也是简单的按照cid奇偶拆分，算法定制为course_$->{cid%2+1}。这个时候，所有的Course课程记录，实际上只能分配到m0.course_1和m2.course_2两个分片表中。这并不是我们期待的结果啊。我们是希望把数据分到四张表里。这时候怎么办？一种很自然的想法是调整分表的算法，让他按照4去轮询，定制分片算法 course_$->{((cid+1)%4).intdiv(2)+1}。 这样简单看起来是没有问题的。如果ID是连续递增的，那么这个算法就可以将数据均匀的分到四个分片中。

```java
public class Test {
    public static void main(String[] args) {
        for (long i = 0; i < 100; i++) {
            long database = i % 2;
            long table = ((i + 1) % 4) / 2 + 1;
            System.out.println("主键："+i+";库分片："+database +":表分片"+table);
        }
    }
}
```

```
Test ×
D:\dev-hook\Java\jdk1.8.0_45\bin\java.exe ...
主键：0;库分片：0:表分片1
主键：1;库分片：1:表分片2
主键：2;库分片：0:表分片2
主键：3;库分片：1:表分片1
主键：4;库分片：0:表分片1
主键：5;库分片：1:表分片2
主键：6;库分片：0:表分片2
```

数据分片很均匀

这时，建议你将这个算法去结合ShardingSphere实际使用一下。下面是是示例配置：

```
# 打印SQL
spring.shardingsphere.props.sql-show = true
spring.main.allow-bean-definition-overriding = true

# ----------------数据源配置
# 指定对应的库
spring.shardingsphere.datasource.names=m0,m1

spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC
spring.shardingsphere.datasource.m0.username=root
spring.shardingsphere.datasource.m0.password=root

spring.shardingsphere.datasource.m1.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.m1.url=jdbc:mysql://localhost:3306/coursedb2?
serverTimezone=UTC
spring.shardingsphere.datasource.m1.username=root
spring.shardingsphere.datasource.m1.password=root
#------------------------分布式序列算法配置
# 雪花算法，生成Long类型主键。
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.type=SNOWFLAKE
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.props.worker.id=1
# 指定分布式主键生成策略
spring.shardingsphere.rules.sharding.tables.course.key-generate-strategy.column=cid
spring.shardingsphere.rules.sharding.tables.course.key-generate-strategy.key-
generator-name=alg_snowflake
```

```
#----------------------配置实际分片节点
spring.shardingsphere.rules.sharding.tables.course.actual-data-nodes=m$->
{0..1}.course_$->{1..2}
#MOD分库策略
spring.shardingsphere.rules.sharding.tables.course.database-
strategy.standard.sharding-column=cid
spring.shardingsphere.rules.sharding.tables.course.database-
strategy.standard.sharding-algorithm-name=course_db_alg

spring.shardingsphere.rules.sharding.sharding-algorithms.course_db_alg.type=MOD
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_db_alg.props.sharding-count=2
#给course表指定分表策略  standard-按单一分片键进行精确或范围分片
spring.shardingsphere.rules.sharding.tables.course.table-strategy.standard.sharding-
column=cid
spring.shardingsphere.rules.sharding.tables.course.table-strategy.standard.sharding-
algorithm-name=course_tbl_alg

# 分表策略-INLINE：按单一分片键分表
spring.shardingsphere.rules.sharding.sharding-algorithms.course_tbl_alg.type=INLINE
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.props.algorithm-expression=course_$->{cid%2+1}
#这种算法如果cid是严格递增的，就可以将数据均匀分到四个片。但是雪花算法并不是严格递增的。
#如果需要做到均匀分片，修改算法同时，还要修改雪花算法。把SNOWFLAKE换成MYSNOWFLAKE
spring.shardingsphere.rules.sharding.sharding-
algorithms.course_tbl_alg.props.algorithm-expression=course_$->
{((cid+1)%4).intdiv(2)+1}
```

然后往course表里连续插入多条消息。

```
    @Test
    public void addcourse() {
        for (int i = 0; i < 10; i++) {
            Course c = new Course();
            //Course表的主键字段cid交由雪花算法生成。
            c.setCname("java");
            c.setUserId(1001L);
            c.setCstatus("1");
            courseMapper.insert(c);
            //insert into course values ....
            System.out.println(c);
        }
    }
```

那么你一定会发现，这十条course信息，很奇怪。库倒是分得挺均匀，但是表却分得很奇怪。就是没有办法插入到四张表的。只能插入到m0.course_1和m2.course_2两张表中。之前试了很多次，问了很多人，也查了很多资料，收效甚微。好像这只是我自己手气不好，自己瞎折腾出来的问题。一直想着是不是算法写错了？或者是运气问题，雪花算法因为某种不可知的神秘因素，不按我的预期生成数据。这应该也是很多人在学习ShardingSphere时经常遇到的问题。

直到后面，随着学习CosID框架的机会，深入梳理了一下分布式主键生成策略，才发现问题就出在雪花算法中。在ShardingSphere中扩展一个自己的雪花算法实现，才最终解决了这个问题。最后这问题细思极恐。雪花算法+取模分片，这应该是很多项目中都通用的一种分片策略。但是在这个场景下，雪花算法埋着一个大坑呢。这个大坑不可能只坑我一个人，应该坑到的是很多人，很多项目。

这到底是怎么回事呢？各位，咱们从头说起。

二、分布式主键要考虑哪些问题?

我们应该如何设计一个分布式主键?

主键是对数据的唯一标识。主键非常重要，尤其当需要用来控制重要数据的生命周期时，主键通常都是标识数据的关键。但是，其实主键并不只是唯一这么简单。

主键除了要标识数据的唯一性之外，其实也是一个挺纠结的东西。在业务层面，我们通常会要求主键与业务不直接相关，这样主键才能够承载更多的，更负载，更频繁变化的业务数据。例如对于订单，要区分订单的唯一性，那么下单时间就是一个天然最好的标识。`这里暂不考虑并发的问题。简单假设，只要时间足够精确，那么下单时间是可以保证唯一性的。`如果用订单的下单时间这样带有明显业务属性的内容来当做主键，那么早期电商业务非常少的时候没有什么问题。但是随着订单业务越来越频繁，为了继续保证区分每一条订单，就会要求对下单时间的区分越来越精确。当电商逐渐演变成现代超大规模，超高并发的场景，以时间作为主键，迟早会无法满足。以其他业务字段来区分，通常也迟早会表现出受业务的制约，影响业务的演变。所以，在设计主键时，最好的方式是使用一个与业务都不相关的字段来作为主键。这样，不管业务如何变化，都可以使用主键来控制数据的生命周期。

但是，另外一个方面，我们通常又会要求主键包含一部分的业务属性，这样可以加速对数据的检索。还是以订单为例，如果我们采用一个与时间完全无关的字段作为主键，当我们需要频繁的统计昨天的订单时，就只能把所有订单都查询出来，然后再按照下单时间字段进行过滤。这样很明显，效率会很低。但是如果我们能够将下单时间作为主键的一部分，例如，以下单时间作为订单的开头部分。那么，我们就可以通过主键前面的下单时间部分，快速检索出一定时间范围内的订单主键，然后再根据主键去获取这一部分订单数据就可以了。这样要查的数据少了，效率自然就能提高了。

所以，对于主键，一方面，要求他与业务不直接相关。这就要求分配主键的服务要足够稳定，足够快速。不能说我辛辛苦苦把业务给弄完了，然后等着分配主键的时候，还要等半天，甚至等不到。`这个要求看似简单，但其实在现在经常讨论的高并发、分布式场景下，一点都不简单。`另一方面，要求他能够包含某一些业务特性。这就要求分配主键的服务能够进行一定程度的扩展。

另外主键也需要考虑安全性，让别人无法通过规律猜出主键来。比如身份证就是一个例子。要是随随便便就能猜到别人的身份证号码，那天下将是一个什么样子?

三、主要的主键生成策略

接下来考虑如何生成靠谱的主键呢？常用的策略有很多，大体可以分为几类。

1、数据库策略

在单数据库场景下，主键可以很简单。可以把主键扔给数据库，让他自己生成主键。比如MySQL的自增主键。

优点很明显。应用层使用简单，都不用考虑主键问题了，因此不会有主键稳定性的问题。以现代数据库的设计，自增主键的性能通常也比较高。另外，也不存在并发问题。应用不管部署多少个服务，主键都不会冲突。

但是坏处也同样明显。数据库自增主键不利于扩展。而且按照之前的分析，这类主键的规律太过明显，安全性也不是很高。在内部系统中使用问题不大，但是暴露在互联网环境就非常危险了。另外，在分库分表场景下，依靠数据库自增生成主键也非常不灵活。例如两台数据库服务，虽然可以定制出 让第一台数据库生成奇数序列，第二台数据库生成偶数序列 的方式让主键不冲突，但是由于每个数据库并不知道整个数据库集群的工作情况，所以如果数据库集群要扩缩容，所有的主键就都需要重新调整。

2、应用单独生成

既然数据库不靠谱，那就由应用自己生成。这一类算法有很多，比如UUID、NANOID、SnowFlake雪花算法等。

与数据库自增方案相比，应用自己生成主键的优点就比较明显。简单实用，比如UUID，用JDK自带的工具生成就行，而SNOWFLAKE，按他的规则自行组合就行了。另外主键很容易进行扩展。应用可以根据自己的需求随意组合生成主键。

但是缺点也非常明显。首先，算法不能太复杂。太复杂的算法会消耗应用程序的计算资源和内存空间，在高并发场景下会给应用带来很大的负担。然后，并发问题很难处理。既要考虑单进程下的多线程并发安全问题，又要防止分布式场景下多进程之间的主键冲突问题，对主键生成算法的要求其实是比较高的。所以，这一类算法虽然看起来挺自由，但是可供选择的算法其实并不多。要自己设计一个即高效，又靠谱的出来，那就更难了。

并且，如果与某一些具体的数据库产品结合使用，那么可能还会有一些定制化的需求。比如，如果使用我们最熟悉的MySQL数据库，通常还会要求主键能够趋势递增。因为MySQL的InnoDB引擎底层使用B+树进行数据存储，趋势递增的主键可以最大限度减少B+树的页裂变。所以，像UUID、NANOID这一类无序的字符串型主键，相比就没有SNOWFLAKE雪花算法这类趋势递增的数字型主键性能高。

3、第三方服务统一生成

还一种典型的思路是借助第三方服务来生成主键。 比较典型的工具有Redis，Zookeeper，还有MongoDB。

- Redis

使用incr指令，就可以成成严格递增的数字序列。配合lua脚本，也比较容易防并发。

- Zookeeper

比较原生的方法是使用Zookeeper的序列化节点。Zookeeper在创建序列化节点时，会在节点名称后面增加一个严格递增的数字序列。

另一种方法，在apache提供的Zookeeper客户端Curator中，提供了DistributedAtomicInteger，DistributedAtomicLong等工具，可以用来生成分布式递增的ID。

- MongoDB

比较原生的方法是使用MongoDB的ObjectID。MongoDB中每插入一条记录，就会给这条记录分配一个objectid。

这些方案成本比较低，使用时也比较灵活。应用拿到这些ID后，还是可以自由发挥进行扩展的，因此也都是不错的主键生成工具。

但是他们的缺点也很明显。这些原生的方式大都不是为了分布式主键场景而设计的，所以，如果要保证高效以及稳定，在使用这些工具时，还是需要非常谨慎。

4、与第三方结合的segment策略

segment策略的基本思想就是应用依然从第三方服务中获取ID，但是不是每次获取一个ID，而是每次获取一段ID。然后在本地进行ID分发。等这一段ID分发完了，再去第三方服务中获取一段。

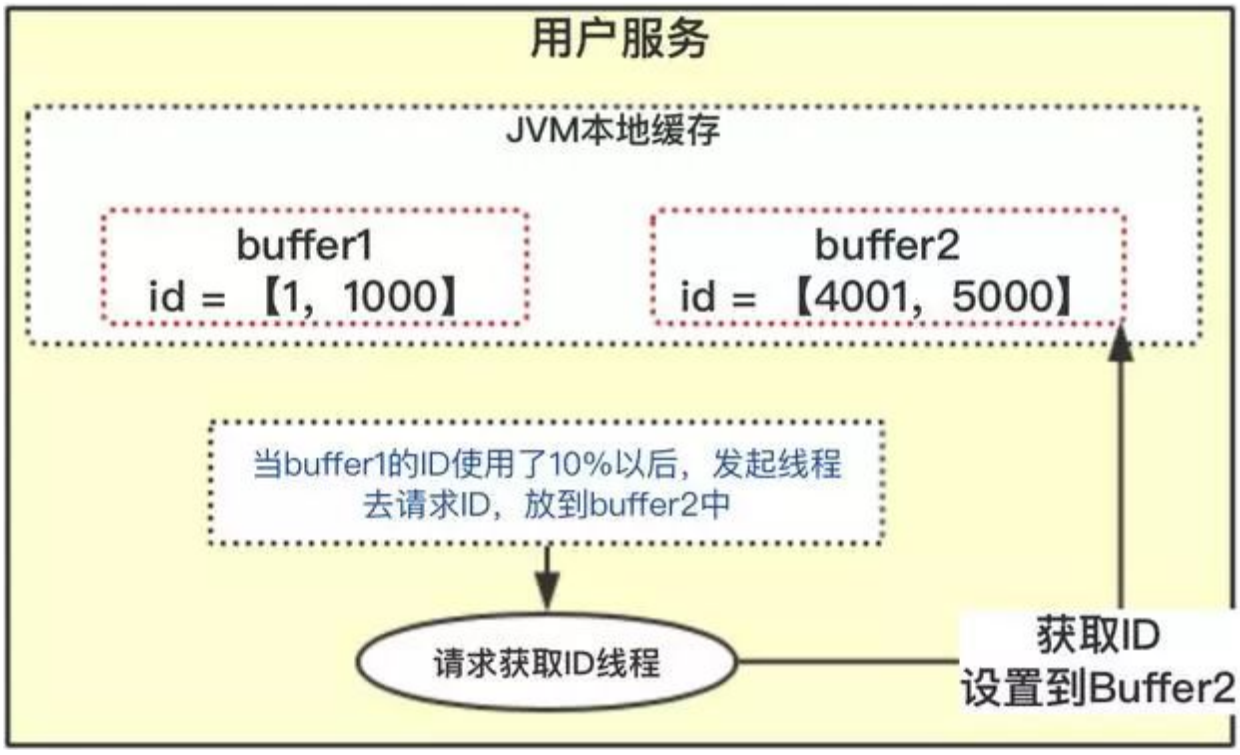
例如，以最常用的数据库为例，我们可以设计一张这样的表：

biz_tag	max_id	step	desc	update_time
user_tag	0	1000	用户ID生成规则	
order_tag	0	2000	订单ID生成规则	

biz_tag只是表示业务，用户服务和订单服务对应的都可能是一大批集群应用。max_id表示现在整个系统中已经分配的最大ID。step表示应用每次过来申请的ID数量。

然后，当第一个订单应用过来申请ID时，就将max_id往前加一个step，变成2000。就表示这2000个ID就分配给这个订单应用了。然后这个订单应用就可以在内存中随意去分配[0,2000)这些ID。而第二个订单应用过来申请ID时，获得的就是[2000,4000)这一批订单应用。这样两个订单应用的ID可以保证不会冲突。

这个策略中有一个最大的问题，就是申请ID是需要消耗网络资源的，在申请资源期间，应用就无法保持高可用了。所以有一种解决方案就是双Buffer写入。



应用既然可以接收一段ID，那就可以再准备一个Buffer，接收另一段ID。当Buffer1的ID使用了10%后，就发起线程去请求ID，放到Buffer2中。等Buffer1中的ID用完了，应用就直接从Buffer2中分配ID。然后等Buffer2用到10%，再同样缓过来。通过双Buffer的交替使用，保证在应用申请ID期间，本地的JVM缓存中一直都是有ID可以分配的。

没错，这就是美团Leaf的完整方案。

他的好处比较明显。ID单调递增，在一定范围内，还可以保持严格递增。通过JVM本地进行号段缓存，性能也很高。

但是这种方案也有几个明显的不足之处。

1、强依赖于DB。其实你可以想象，DB中最为核心的就是max_id和step两个字段而已。这两个字段其实可以往其他存储迁移。想用那个就用哪个不是更方便？这个想法现在不需要自己动手了，CosID已经实现了。数据库、Redis、Zookeeper、MongoDB，想用哪个就用哪个。程序员又找到了一个偷懒的理由。

--2、10%的阈值不太灵活。如果应用中的业务非常频繁，分配ID非常快，10%有可能不够。而如果业务非常慢，10%又有点浪费，因为申请过来的ID，如果应用一停机，就浪费掉了。所以，其实可以添加一个动态控制功能，根据分配ID的频率，灵活调整这个阈值，保持本地缓存内的ID数量基本稳定。并且，这也可以用来定制限流方案。

3、延长本地缓存。不管你用哪种服务来充当号段分配器，还是会有一个问题。如果号段分配器挂了，本地应用就只能通过本地缓存撑一段时间。这时，是不是可以考虑多缓存几个号段，延长一下支撑的时间呢？

CosId也想到了，直接将双Buffer升级成了SegmentChain。用一个链表的方式可以灵活缓存更多的号段。默认保留10个Segment，并且在后面分配ID的过程中，也尽量保证SegmentChain中的Segment个数不少于10个。这不就是为了保证本地缓存能够比较充足吗？

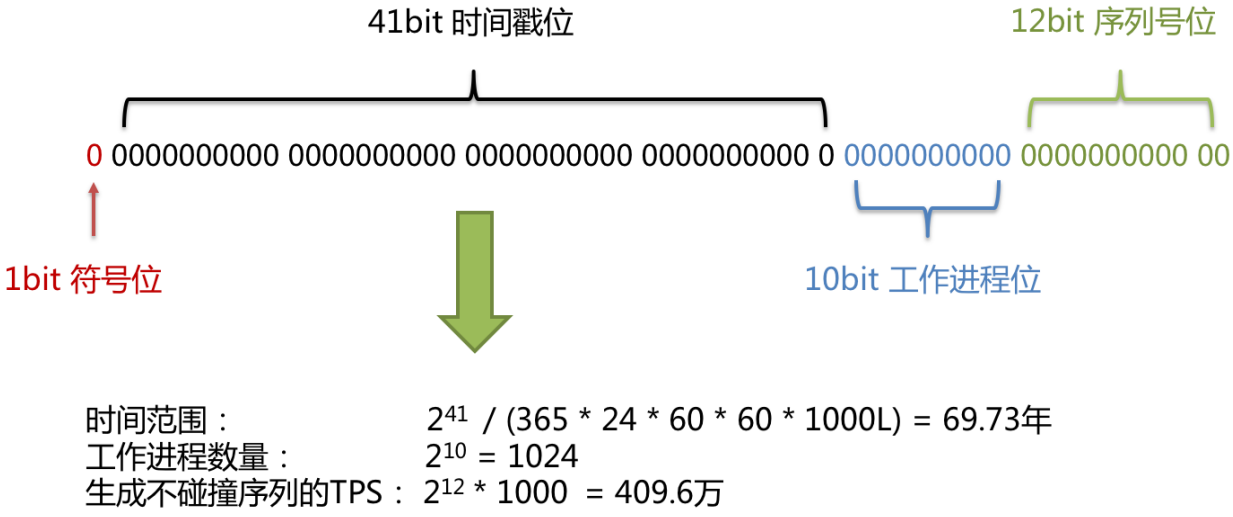
4、ID的安全性其实是不太高的。分配的ID在同一个号段内是连续的，之前分析过，这种规律过于明显的ID其实是不太安全的。在面向互联网使用的时候，还是需要自行进行一些打散的操作。比如下面会提到一种方法，将生成的主键作为雪花算法的工作机器位，再次计算生成主键。

以上这几类可以认为是比较基础的分布式主键生成工具。以这些方案为基础，就诞生了很多其他的玩法。下面分享几种典型的思路。

四、定制雪花算法

雪花算法是twitter公司开源的ID生成算法。他不需要依赖外部组件，算法简单，效率也高。也是实际企业开发过程中，用得最为广泛的一种分布式主键生成策略。

雪花算法的基础思想是采用一个8字节的二进制序列来生成一个主键。为什么用8个字节？因为8字节正好就是一个Long类型的变量。即保持足够的区分度，又能比较自然的与业务结合。



可以看到，SNOWFLAKE其实还是以41个bit的时间戳为主体，放在最高位。接下来10个bit位的工作进程位，是用来标识每一台机器的。但是实现时，是留给应用自行扩展的。后面12个bit的序列号则就是一个自增的序列位。

其核心思想就是将唯一值拼接成一个整体唯一值。首先从整体上来说，时间戳是一个最好的保证趋势递增的数字，所以时间戳自然是主体，放到最高位。但是如果有多个节点同时生成，那么就有可能产生相同的时间戳。怎么办？那就把进程ID给拼接上来。接下来如果在同一个进程中有多个线程同时生成，那么还是会产生相同的ID，怎么办？那就再加上一个严格递增的序列位。这样就整体保证了全局的唯一性。

在标准的雪花算法基础上，也诞生了很多类似的雪花算法实现。无非就是对这些数据根据业务场景进行重组。比如缩短时间戳位，将工作进程位加长，拆分成为datacenter和worker两个部分，等等，但是其实万变不离其宗。

这里唯一需要注意下的是中间的第二部分，很多人只是简单地把他理解为服务器的ID，其实这是有问题的。同一个服务器上可以跑多个服务应用。雪花算法要求的工作进程是区分这些应用，而不是这些机器的。其实你可以想象，就是在这种情况下，不同进程之间的时间戳相同的概率会更高，工作进程位的作用才更重要。简单理解为机器ID，反而会很容易产生一些误解。

而在具体实现时，雪花算法实际上只是提供了一个思路，并没有提供现成的框架。比如ShardingSphere中的雪花算法就是这样生成的。

```
@Override
public synchronized Long generateKey() {
    long currentMilliseconds = timeService.getCurrentMillis();
    if (waitTolerateTimeDifferenceIfNeed(currentMilliseconds)) {
        currentMilliseconds = timeService.getCurrentMillis();
    }
    if (lastMilliseconds == currentMilliseconds) {
        if (0L == (sequence = (sequence + 1) & SEQUENCE_MASK)) {
            currentMilliseconds = waitUntilNextTime(currentMilliseconds);
        }
    } else {
        vibrateSequenceOffset();
        sequence = sequenceOffset;
    }
    lastMilliseconds = currentMilliseconds;
    return ((currentMilliseconds - EPOCH) << TIMESTAMP_LEFT_SHIFT_BITS) | (getWorkerId() << WORKER_ID_LEFT_SHIFT_BITS) | sequence;
}
```

时间戳位

读取应用配置的worker.id参数

序列号位

这里面其实隐藏三个问题。

1、如影随形的时钟回拨问题

雪花算法强依赖于时钟，而高精度的时钟是很难保持一致的。一方面，在分布式场景下，多个机器之间的时钟很难统一，这个倒是可以依赖于Ntpd这样的服务进行通知。但是在同一个机器上，由于时钟只能依赖内核的电信号维护，而电信号很难保持稳定，这也就造成操作系统上的时钟并不是准确的。在高并发场景下，获得的高精度时间戳，有时候会往前跳，有时候又会往回拨。一旦时钟往回拨，就有可能产生重复的ID，这就是时钟回拨问题。

雪花算法其实并没有提供针对时钟回拨问题的标准解决方案，这其实也造成了一些小分歧。解决时钟回拨问题的基本思路都是应用自己记录上一次生成主键的时间戳，然后拿当前时间和上一次的时间进行比较。如果当前时间小于上一次的时间戳了，那就发生了时钟回拨。但是发现问题后怎么处理呢？ShardingSphere中默认的解决方式是让当前线程休眠一会。例如上图中waitToLerateTimeDifferenceIfNeed方法就是在处理时钟回拨问题。

```java
@SneakyThrows(InterruptedException.class)
private boolean waitTolerateTimeDifferenceIfNeed(final long currentMilliseconds) {
    if (lastMilliseconds <= currentMilliseconds) {
        return false;
    }
    long timeDifferenceMilliseconds = lastMilliseconds - currentMilliseconds;
    Preconditions.checkState( b: timeDifferenceMilliseconds < maxTolerateTimeDifferenceMilliseconds,
            errorMessageTemplate: "Clock is moving backwards, last time is %d milliseconds, current time is %d milliseconds", lastMilliseconds,
    Thread.sleep(timeDifferenceMilliseconds); 休眠到上一次计算的时间
    return true;
}
```

而当前版本ShardingSphere集成了COSID主键生成框架。框架中也包含了雪花算法。他发现时钟回拨，就直接抛出异常了。

```java
@Override
public synchronized long generate() {
    long currentTimestamp = getCurrentTime();
    if (currentTimestamp < lastTimestamp) {  时钟回拨
        throw new ClockBackwardsException(lastTimestamp, currentTimestamp);
    }

    //region Reset sequence based on sequence reset threshold,Optimize the problem of uneven sharding.

    if (currentTimestamp > lastTimestamp
        && sequence >= sequenceResetThreshold) {
        sequence = 0L;
    }

    sequence = (sequence + 1) & maxSequence;

    if (sequence == 0L) {
        currentTimestamp = nextTime();
    }

    //endregion
    lastTimestamp = currentTimestamp;
    long diffTimestamp = (currentTimestamp - epoch);
    if (diffTimestamp > maxTimestamp) {
        throw new TimestampOverflowException(epoch, diffTimestamp, maxTimestamp);
    }
    return diffTimestamp << timestampLeft
        | machineId << machineLeft       拼凑主键
        | sequence;
}
```

这里的重点不是想要讨论哪种处理时钟回拨问题更合理。而是可以看到，只要使用雪花算法，就埋下了时钟回拨这个雷，需要应用自行去处理。这对于一个没有标准实现的工具型算法来说，不得不说是一种遗憾。

另外，这种传统的雪花算法通过本地保存时间戳的方式来判断是不是发生了时钟回拨，也只能保证本地时间戳是递增的。那么在多个服务组成的集群当中，就无法保证时间戳的统一了。虽然可以通过给每个服务配置不同的工作进程位来防止不同服务之间的主键冲突，但是，万一应用没有配呢？至少我就见过大部分的应用不会为了一个小小的雪花算法去单独考虑如何分配工作进程位。然后，当然也可以使用ntpd这样的时间同步服务把多个机器的时间同步一下，但是同样，不会有人为了一个小小的雪花算法就这么去做。

有一种优化方案就是将时间戳也从本地扔到第三方服务上去，比如Zookeeper。这样多个服务就可以根据共同的时间戳往前推进，省却了时间同步的麻烦。美团的Leaf就是这么做的。但是，这样又是会给雪花算法增加强绑定，同时会降低效率。

这似乎又变成了一个需要进行方案取舍、实现优化的头疼环节。就像整个分布式主键生成问题一样。

2、用主键生成策略优化分配工作进程位

雪花算法中的第二个部分，工作进程位是用来区分不同工作进程的。他也是分布式场景下，保证ID不重复的很重要的字段。在雪花算法中，这个工作进程位是交由应用自己指定的。应用可以随意给每个服务分配一个工作进程。 这在小规模集群中是没有问题的。但是，在大规模集群中这就变得有点麻烦了。想想看，你要在一个一百台机器组成的大集群里，给每个机器分配一个不同的ID，这会是什么感觉?

所以，这时有一种思路就是把这个MachineId当做一个短的并发不是很高的分布式主键来处理。用其他分布式主键生成的方式生成工作进程位。没错。 Cosid框架就是这么想的。他对于SnowFlake的改造也就集中在这个工作进程位。关于CosId的具体实现，后面会再做详细分析。这里我们可以先来思考一下这是一个什么样的场景。我们要依赖工作进程位来生成一个分布式唯一主键，然后现在又需要依赖一个分布式唯一主键生成策略来生成一个工作进程位。这个鸡生蛋，蛋生鸡的问题，要如何解套呢?

首先，工作进程位分配并不需要考虑太高的并发。通常工作进程位只需要在一个应用启动的时候进行分配就可以了。在应用运行的过程中，不需要有太多的变化。而且，就算依赖的第三方服务出现问题了，工作进程位分配失败了，在应用启动过程中就抛出错误，等人们把问题处理完了再分配，也没有什么大的问题。所以工作进程位不需要像分布式唯一主键那样，考虑那么多的缓存、高性能等问题，每次单步推进。申请一次，分配一个就行了。

然后，工作进程位有一些天然的带有唯一性的因素。如果只是粗略的把一个工作进程看成是一台机器，也就是每台机器只运行一个服务的简单场景。那么其实读取服务器的IP地址就已经是一个区分不同工作进程的唯一因素了。但是，如果考虑到每台机器运行多个服务，甚至还包含Docker等各种虚拟化技术的场景，IP地址就不够了。但是，此时，如果再加上一个运行端口，就可以区分唯一了。所以，工作进程位的分配，相比于分布式主键生成策略，其实天生就是可以有很多现成的唯一性因素的，不需要像雪花算法那样去设计复杂的结构。

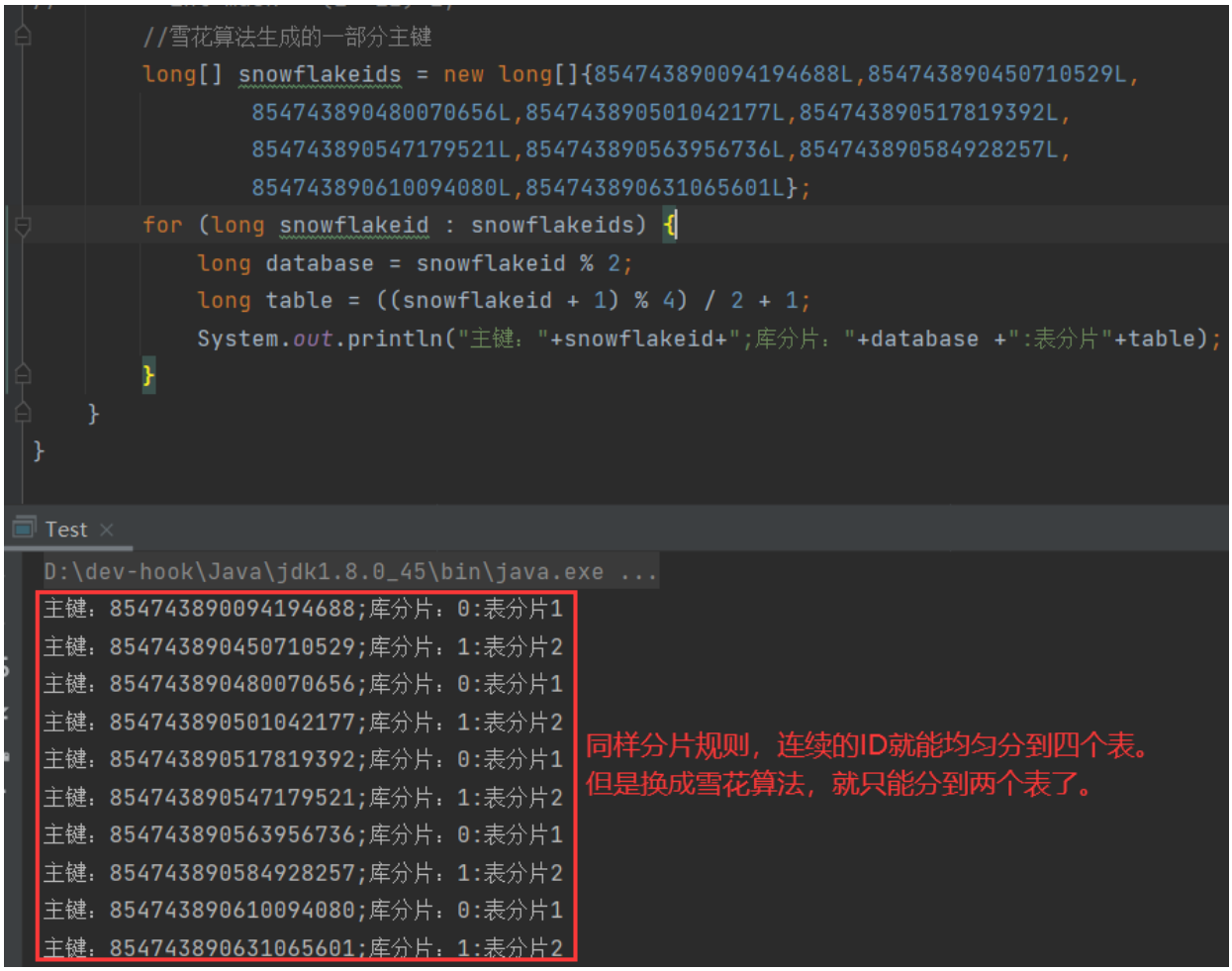
最后，工作进程位的分配需要可以保持稳定。工作进程与分布式主键还有一个区别，就是工作进程毕竟还是要与一个应用服务建立绑定关系的。给一个应用分配了一个工作进程位之后，如果应用崩溃了，为了保证应用重启后，能够保持稳定的工作状态，他后续产生的雪花ID还是能够保持稳定的区分度。所以，在应用当中分配了一个工作进程位之后，就可以用一个本地缓存，把这个结果保存下来。如果应用重启后，还需要继续保持，那么这个缓存还可以持久化到本地文件当中。

其实把这几个方面想明白了，Cosid当中的工作进程位分配机制也就大致成型了。到底是不是这样呢？别急，后面马上就轮到Cosid出场了。

3、从序列号字段定制雪花算法的连续性

雪花算法生成的ID是不连续的，这很容易理解。但是很多时候，在进行分库分表时，我们还是希望雪花算法生成的ID能够保持某一种规律，这样在定制分库分表算法时，才可以比较好的定制数据分片算法。但是，很可惜，雪花算法给你埋着坑呢。

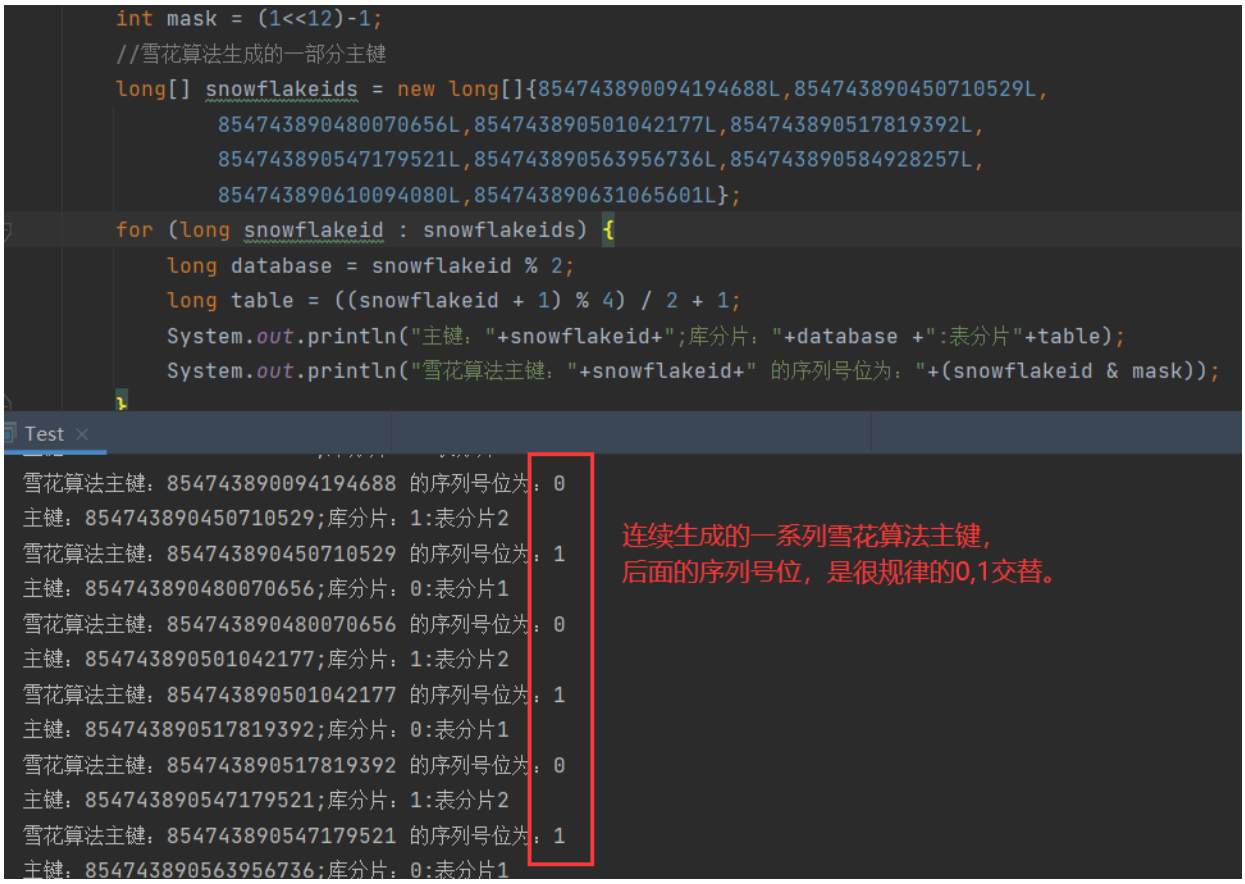
回到我们开头的小问题。我用ShardingSphere原生的雪花算法生成了一批主键，然后尝试按照之前配置的规则进行拆分。会发现很奇怪的现象，库对2取模，很均匀的分配到了两个片。但是表是对4取模的，也均匀的分到了两个片。用这样的算法去实际分库分表，当然只能分到两个片。



这是为什么呢？简单理解，就是雪花算法生成的结果不连续呗。但是为什么会体现出奇偶分配很均匀的情况呢？这就要深入到雪花算法的实现里了。

因为雪花算法虽然最后规定了一个序列位，只有在lastMilliseconds == currentMilliseconds时才往上加1，否则就重置为-1。简单理解，就是只有在时间戳相同的时候，序列号才往上加1.如果时间戳不同，序列号就会从0开始往上叠加。再加上只有在lastMilliseconds == currentMilliseconds时，才会将currentMilliseconds推进到下一个时间点。这时，在单线程情况下，雪花算法生成的一系列ID的序列位就是有规律的0,1,0,1。

是不是这样呢？验证一下就知道了。

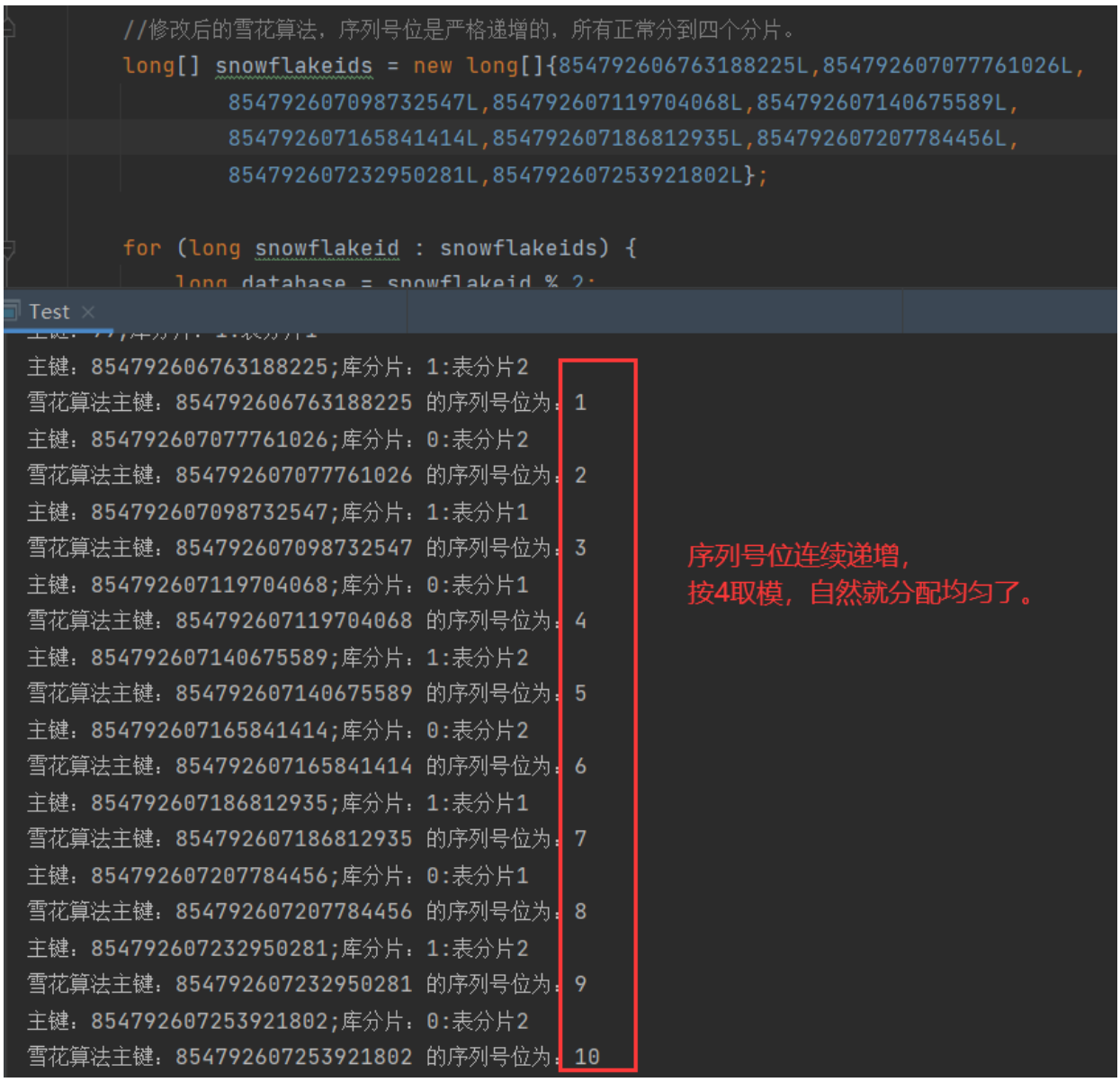


搞清楚问题后，解决的方案比较简单了。在这个场景下，最简单的一种修改方式，就是让sequence每次都加1呗。不管时间戳相不相同，都加1。然后只要不超过12bit的范围，这样对2取模，对4取模这些不就都能均匀分布了吗？

例如简单的将雪花算法的生成逻辑做一下调整。

```
@Override
    public synchronized Long generateKey() {
        long currentMilliseconds = timeService.getCurrentMillis();
        if (waitTolerateTimeDifferenceIfNeed(currentMilliseconds)) {
            currentMilliseconds = timeService.getCurrentMillis();
        }
        if (lastMilliseconds == currentMilliseconds) {
//            if (0L == (sequence = (sequence + 1) & SEQUENCE_MASK)) {
                currentMilliseconds = waitUntilNextTime(currentMilliseconds);
//            }
        } else {
            vibrateSequenceOffset();
//            sequence = sequenceOffset;
            //让sequence单调递增
            sequence = sequence >= SEQUENCE_MASK ? 0:sequence+1;
            //sequence = SEQUENCE_MASK==sequence&SEQUENCE_MASK ? 0 : sequence +1;
        }
        lastMilliseconds = currentMilliseconds;
        return ((currentMilliseconds - EPOCH) << TIMESTAMP_LEFT_SHIFT_BITS) |
(getWorkerId() << WORKER_ID_LEFT_SHIFT_BITS) | sequence;
    }
```

这样生成出来的雪花算法，就能保证序列号位基本上是连续的。数据也能正常分到四个分片了。



可以看到，这个序列号位连续递增了之后，数据分片的结果也均匀了。

由此可见，雪花算法虽然在保证全局唯一的场景下，是几乎无可挑剔的，但是结合具体的业务场景，他却并不是金科玉律。需要结合业务场景进行灵活定制。

当然，这个分库分表的场景，改改雪花算法，你可能觉得是我小题大做，那么下面这个业务场景，你一定要仔细看看。

> 雪花算法一直是分布式ID的标准。CosId算法也自行实现了一套SnowFlake的实现，并且这套实现已经集成到了ShardingSphere的正式版本当中。那么他对这个雪花算法到底动了些什么手脚呢?

4、根据雪花算法扩展基因分片法

业务场景：用户表，是应用当中最为常见的一张表。如果要对用户表进行分库分表。那么很简单的就会按照userId进行取模分片，这时自然后续的查询也需要按照userId来查，才能比较高效的定位到某一个数据分片。否则的话，就要走全路由。如果分片数量比较多，这样全路由查询的性能消耗，几乎是不可接受的。

现在问题来了。对于用户登录的场景，应该要怎么支持呢？在用户登录场景，只会传过来用户名，你甚至都不知道这个用户名是否存在。这样自然就无法定位在哪个分片了。这怎么查呢？难道就只能全路由查？去几十上百个分片表里都查一次？想想就觉得恐怖。

这时可以用一种业界称为基因法的分片算法来解决这个问题。他的基础思想有点类似于雪花算法的序列部分。基础思想是在给用户分配userId时，就将用户名当中的某种序列信息插入到userId当中。从而保证userId和用户名可以按照某一种对应的规则分到同一个分片上。这样，在用户登录时，就可以根据用户名确定对应的用户信息只有可能分布在某一个数据分片当中。这样就只要去对应的分片上进行一次查询，就能查询到用户对应的信息。

具体在实现时，可以参照雪花算法的实现。

例如，在用户注册时，用户输入了一个username。然后，从主键分发器获得了一个原始的userid。

```
        //原始预备生成的用户名
        String username = "testroy";
        System.out.println("原始预备插入的用户名："+username);
        //原始预备生成的唯一ID
        long originId = 12394846L;
        System.out.println("原始预备插入的用户ID："+originId);
```

先确定一个MASK的长度，也就是工作序列号位用二进制表示的长度。例如确定为3。然后从用户名当中抽取出一个用三个bit表示的分片关键字，称为分片基因。如果用户名是一个数字，那么就直接按照雪花算法的处理方式就行了。如果是个字符串，那么就进行一次hash运算，转成一个数字。也就是创建一个用2进制表示全为1的3个bit组成的数组，然后与目标列进行相位与操作，这样就能够拿到对应数字的二进制表达的后三位。这个就称为分片基因。

```
        public static final int datasize = 3; //二进制基因片段的长度
        int mask = (int)(Math.pow(2,datasize) -1);//掩码，二进制表述为全部是1.  111
        long userGene = username.hashCode() & mask;
        //根据用户名查询时，获取到的分片结果：
        System.out.println("根据用户名获取到的分片基因："+userGene);
```

然后将这个基因片段与原始ID，按照二进制的方式组合在一起，生成一个包含了username的基因片段的新的userid。这个新的userId按照算法的话，一定是会与username保持一样的。

```
        //给ID添加用户名的基因片段后的新ID
        long newId = (originId<<datasize)|userGene;
        System.out.println("添加分片基因后的用户ID2：  "+newId);
        long newIdGene = newId & mask;
        System.out.println("根据用户ID2获取到的分片结果：  "+newIdGene);
```

未来进行取模分片时，只要是按照2,4,8这样的数字取模，那么这个userGene和newIdGene的分片结果一定是一样的。

```
        //按照新的用户ID对8取模进行数据分片
        long actualNode = newId % 8;
        System.out.println("用户信息实际保存的分片：  "+actualNode);
        long userNode = (username.hashCode() & mask) % 8;
        System.out.println("根据用户名判断，用户信息可能的分片：  "+userNode);
```

整个示例整个到一起是这样的

```
public class GeneDemo2 {
    //二进制基因片段的长度
    public static final int datasize = 3;

    public static void main(String[] args) {
        //原始预备生成的用户名
        String username = "testroy";
        System.out.println("原始预备插入的用户名：  "+username);
        //原始预备生成的唯一ID
        long originId = 12394846L;
        System.out.println("原始预备插入的用户ID：  "+originId);

        int mask = (int)(Math.pow(2,datasize) -1);//掩码，二进制表述为全部是1.  111
        long userGene = username.hashCode() & mask;
        //根据用户名查询时，获取到的分片结果：
        System.out.println("根据用户名获取到的分片基因：  "+userGene);

        //给ID添加用户名的基因片段后的新ID  -- 将username.hashCode二进制左移三位，再添加用户名的分片结果。 这样保持了原始ID的唯一性。
        long newId = (originId<<datasize)|userGene;
        System.out.println("添加分片基因后的用户ID：  "+newId);
        long newIdGene = newId & mask;
        System.out.println("新用户ID的分片基因"+newIdGene);

        //按照新的用户ID对8取模进行数据分片
        long actualNode = newId % 8;
        System.out.println("用户信息实际保存的分片：  "+actualNode);
        long userNode = (username.hashCode() & mask) % 8;
        System.out.println("根据用户名判断，用户信息可能的分片：  "+userNode);
    }
}
```

执行结果是这样的：

```
原始预备插入的用户名：testroy
原始预备插入的用户ID：12394846
根据用户名获取到的分片基因：2
添加分片基因后的用户ID：99158770
新用户ID的分片基因2
用户信息实际保存的分片：2
根据用户名判断，用户信息可能的分片：2
```

最后，整个基因法分片的方案就是这样设计：

1、用户注册时，先从主键生成器获取一个唯一的用户ID。这是原始ID。

2、按照上面示例的方式，从用户注册时输入的用户名中抽取分片基因。并将分片基因插入到原始ID中，生成一个新的用户ID。

3、根据新的用户ID，将用户数据按照 2或4或8 的数量进行分片存储，保存到数据库中。这样保证了根据用户名和新的用户ID都是可以拿到相同的分片结果的，也就是数据实际存储的分片。

4、用户登录时，根据用户输入的用户名，获取分片基因，并对数据分片数取模，这样就能获得这个用户名可能存在的用户分片。如果用户信息存在，就只可能保存在这一个分片里，不可能在其他分片。这样就只要到这一个用户分片上进行查询，就能获得用户的信息了。如果查不到，那就证明用户输入的用户名不存在，也不用去其他分片上确认了。

基因分片法总结：

从实现中也能看到，基因分片法还是有很多限制的

1、分片数量和基因位数强绑定。 比如基因片段长度如果设置为3，那么数据的分片数就只能是2或4或8。当然，这其实也有一个好处，那就是如果数据集群从4扩展到8，那么用户数据的迁移量是最少的。只要迁移一半数据。如果基因片段设置更长，也意味着更大的扩展空间。

2、基因分片法只能在主键中插入一个基因片段。如果还想要按照其他字段查询，就无法做到了。比如在登录场景，也有可能要根据用户的手机号码查询，这时，基因法就没有用了。这时，通常的解决方案就只有在插入用户信息的时候，单独维护一个从手机号码到所在分片的倒排索引。这个思想比较简单，但是具体实现时的麻烦会比较多。

> 基因分片法，其实一直以来会觉得有点过于偏理论，并没有一个框架来帮你实现一个现成的算法，所以很多人对他其实不太感冒。但是，结合雪花算法来看，是不是就是对后面的工作序列位做做手脚？雪花算法都弄好了，源码都有现成的可以抄了，基因分片法你就不想自己实现一下？

五、从这几个方面来理解CosID

CosId的官方资料太简略了，几乎等于没有。而且上手时也非常容易出错，更别说理解了。但是有了前面过节章节对于主键生成策略的各种铺垫，再来理解CosID就会容易很多。

1、单独搭建测试应用

虽然CosID目前已经集成进了ShardingSphere，但是在你真正想要跟ShardingSphere一起用之前，最好先搭建一个单独的Maven应用，确定你对CosID有足够了解了，再开始往ShardingSphere中去集成。

搭建步骤，非常简单，就按照maven项目三板斧就可以了。

Step1、pom.xml依赖

```xml
<properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <spring.boot.version>2.6.14</spring.boot.version>
        <cosid.version>1.18.6</cosid.version>
        <curator.version>5.1.0</curator.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>me.ahoo.cosid</groupId>
            <artifactId>cosid-spring-boot-starter</artifactId>
            <version>${cosid.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <version>${spring.boot.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <version>${spring.boot.version}</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Step2、启动类

```java
@SpringBootApplication
//这两个注解应该放到框架里面去声明的。
@EnableConfigurationProperties({MachineProperties.class})
@ComponentScans(value = {@ComponentScan("me.ahoo.cosid")})
public class DistIDApp {

    public static void main(String[] args) {
        SpringApplication.run(DistIDApp.class,args);
    }
}
```

> 这里是CosID目前版本的第一个小坑。启动类上那两个注解，明显应该要集成到starter内部的，但是现在却需要应用手动配置。

Step3、application.properties配置

```
cosid.namespace=cosid-example
cosid.enabled=true
cosid.machine.enabled=true
cosid.machine.distributor.manual.machine-id=1
cosid.snowflake.enabled=true
cosid.snowflake.provider.test.friendly=true
```

这个配置是一个最简单可以跑起来的配置。有了前面的铺垫，应该就能猜到这个配置是用来生成一个雪花算法的主键的，其中工作进程位配置为1。

> 为什么要叫machine呢？之前分析过，雪花算法的第二部分是要区分工作进程，而不是机器。这里简单称为machine其实反而更容易让人产生误解。shardingsphere中原本的worker.id这个参数名我都觉得比machine更好。

Step4、应用中生成主键

应用当中使用CosID就非常简单了。只要从Spring容器当中获取IdGeneratorProvider示例，就可以获取ID了。

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class DistIDTest {
    @Resource
    private IdGeneratorProvider provider;
    @Test
    public void getId(){
        for (int i = 0; i < 100; i++) {
            System.out.println(provider.getShare().generate());
        }
    }
}
```

对CosID的调整，也就集中在对配置文件的修改。单元测试的代码基本不需要动。这样用起来还真是挺方便的。

CosID虽然集成了很多种实现机制，但是主键生成模式也就三种。1、SnowFlake雪花算法。2、SegmentID号段模式。3、SegmentChainID号段链模式。其中后两种应该算是同一类模式，都是用segment模式，依靠第三方服务生成分布式主键。只不过SegmentID号段模式属于单Segment，而SegmentChainID号段链模式是扩展成为多Segment。

接下来逐一了解这几种模式。

2、SnowFlake雪花算法

1、基础使用

之前搭建的简单示例，就是一个使用雪花算法的示例。从这个示例就能看到，CosID对雪花算法的扩展主要是对其中的工作进程位进行扩展，后面的序列号位还是传统的雪花算法的实现。工作进程位在CosID中被称为MachineID，机器ID。而接下来，CosID对于MachineID提供了多种实现形式。之前演示了manual，表示手动扩展模式。

可以通过参数指定具体的实现方式。可选的方式有很多种，具体可以看他这个枚举类型：

```
public enum Type {
    MANUAL, //手动分配
    STATEFUL_SET, //与K8s结合的状态机机制
    JDBC,
    MONGO,
    REDIS,
    ZOOKEEPER,
    PROXY //类似ShardingProxy，搭建一个第三方CosID服务分配
}
```

如果你想要使用最为常见的JDBC的方式，那么只要指定配置即可。

```
cosid.machine.distributor.type=jdbc
```

如果你要使用jdbc模式，那么还需要添加cosid-jdbc的扩展依赖包，并且自行引入jdbc相关的依赖。

```
<dependency>
    <groupId>me.ahoo.cosid</groupId>
    <artifactId>cosid-jdbc</artifactId>
    <version>${cosid.version}</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.20</version>
    <!-- 版本冲突太恶心了 -->
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-autoconfigure</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>${spring.boot.version}</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

其中cosid-jdbc是cosid的核心扩展包。而其他相关依赖则是要往Spring容器当中注入一个DataSource数据源。对于DataSource数据源，cosid是像mybatis一样，直接从Spring容器当中引用，而不管如何注册。

然后修改配置

```
cosid.machine.distributor.type=jdbc
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/coursedb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
```

配置好数据源后，还需要初始化表结构。初始化表结构的语句在源码当中有提供。 当前版本只能自己手动去数据库里建表。

```
create table if not exists cosid_machine
(
    name            varchar(100)     not null comment '{namespace}.{machine_id}',
    namespace       varchar(100)     not null,
    machine_id      integer unsigned not null default 0,
    last_timestamp  bigint unsigned  not null default 0,
    instance_id     varchar(100)     not null default '',
    distribute_time bigint unsigned  not null default 0,
    revert_time     bigint unsigned  not null default 0,
    constraint cosid_machine_pk
        primary key (name)
) engine = InnoDB;

create index idx_namespace on cosid_machine (namespace);
create index idx_instance_id on cosid_machine (instance_id);
```

> 这里的实现很迷。对于jdbc模块，源码当中已经构建了一个JdbcMachineIdInitializer对象用于自动建表。但是却并没有与Spring-starter模块集成，也就是说只能手动建表。而对于segment模式，则在集成jdbc时，实现了一个参数可以自动建表。姑且认为未来会修复把。

好的，如果没有版本冲突，那么接下来就可以愉快的跑示例，获取分布式ID了。

2、重点机制剖析

关于雪花算法的实现机制，做一个单元测试就清楚了。

```
    @Resource
    private MachineId machineId;

    @Resource
    private SnowflakeId snowflakeId;
    @Test
    public void snowflakeId(){
        System.out.println("机器ID: "+machineId.getMachineId());
        for (int i = 0; i < 100; i++) {
            System.out.println("SnowflakeId:"+snowflakeId.generate());
        }
    }
```

可以看到，其实CosId就是通过注入一个MachineId实例，提供机器位。然后通过注入SnowflakeId实例，生成雪花算法。

首先，雪花算法的实现其实之前已经看过。SnowFlakeId实例的注入方式是这样的：

```
    //me.ahoo.cosid.spring.boot.starter.snowflake.CosIdSnowflakeAutoConfiguration
    @Bean
    @ConditionalOnMissingBean
    public SnowflakeId shareSnowflakeId(final MachineId machineId,
IdGeneratorProvider idGeneratorProvider,
                                        ClockBackwardsSynchronizer
clockBackwardsSynchronizer) {
        SnowflakeIdProperties.IdDefinition shareIdDefinition =
snowflakeIdProperties.getShare();
        //根据配置构建不同的雪花算法实例。
        SnowflakeId shareIdGen = createIdGen(machineId, shareIdDefinition,
clockBackwardsSynchronizer);
        idGeneratorProvider.setShare(shareIdGen);
        if (snowflakeIdProperties.getProvider().isEmpty()) {
            return shareIdGen;
        }
        snowflakeIdProperties.getProvider().forEach((name, idDefinition) -> {
            IdGenerator idGenerator = createIdGen(machineId, idDefinition,
clockBackwardsSynchronizer);
            //添加到IdGenerator中，这样就可以通过IdGenerator统一生成ID了。
            idGeneratorProvider.set(name, idGenerator);
        });

        return shareIdGen;
    }
```

这里可以看到两点：

1、createIdGen方法构建雪花算法实例。

核心就在这里：

```
public abstract class AbstractSnowflakeId implements SnowflakeId {
    //.....
    @Override
    public synchronized long generate() {
        long currentTimestamp = getCurrentTime();
        if (currentTimestamp < lastTimestamp) {
            throw new ClockBackwardsException(lastTimestamp, currentTimestamp);
        }
        //region Reset sequence based on sequence reset threshold,Optimize the
problem of uneven sharding.
        if (currentTimestamp > lastTimestamp
            && sequence >= sequenceResetThreshold) {
            sequence = 0L;
        }
        sequence = (sequence + 1) & maxSequence;
```

```
        if (sequence == 0L) {
            currentTimestamp = nextTime();
        }
        //endregion
        lastTimestamp = currentTimestamp;
        long diffTimestamp = (currentTimestamp - epoch);
        if (diffTimestamp > maxTimestamp) {
            throw new TimestampOverflowException(epoch, diffTimestamp,
maxTimestamp);
        }
        return diffTimestamp << timestampLeft
            | machineId << machineLeft  //注入的机器位
            | sequence;
    }
}
```

可以看到，就只是将工作序列位改成了引入的machineid机器位。

然后，其中的机器位machineId，就是通过注入到Spring容器当中的MachineID对象获取的。注入MachineID的方式如下。

```
//  me.ahoo.cosid.spring.boot.starter.machine.CosIdMachineAutoConfiguration
    @Bean
    @ConditionalOnMissingBean(value = MachineId.class)
    public MachineId machineId(MachineIdDistributor machineIdDistributor, InstanceId
instanceId) {
        int machineId =
machineIdDistributor.distribute(cosIdProperties.getNamespace(),
machineProperties.getMachineBit(), instanceId,
machineProperties.getSafeGuardDuration()).getMachineId();
        return new MachineId(machineId);
    }
```

所以，对于MachineId分配这个功能，在CosId框架当中，都是通过MachineIdDistributor接口的distribute方法扩展出来的。

而使用JDBC方式，具体的MachineIdDistributor对象实例，是这样注入的。

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnCosIdEnabled
@ConditionalOnCosIdMachineEnabled
@ConditionalOnClass(JdbcMachineIdDistributor.class)
@ConditionalOnProperty(value = MachineProperties.Distributor.TYPE, havingValue =
"jdbc")
public class CosIdJdbcMachineIdDistributorAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public JdbcMachineIdDistributor jdbcMachineIdDistributor(DataSource dataSource,
MachineStateStorage localMachineState, ClockBackwardsSynchronizer
clockBackwardsSynchronizer) {
```

```
        return new JdbcMachineIdDistributor(dataSource, localMachineState,
clockBackwardsSynchronizer);
    }

}
```

CosId就是通过配置类上一通眼花缭乱的@Conditional注解，注入不同的MachineIdDistributor实例，从而实现MachineId生成。其他类型的机器生成器也都是类似的。例如，手动指定机器ID时，他注入的MachineIdDistributor实例是这样的：

```
// me.ahoo.cosid.spring.boot.starter.machine.CosIdMachineAutoConfiguration
    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnProperty(value = MachineProperties.Distributor.TYPE,
matchIfMissing = true, havingValue = "manual")
    public ManualMachineIdDistributor machineIdDistributor(MachineStateStorage
localMachineState, ClockBackwardsSynchronizer clockBackwardsSynchronizer) {
        MachineProperties.Manual manual =
machineProperties.getDistributor().getManual();
        Preconditions.checkNotNull(manual, "cosid.machine.distributor.manual can not
be null.");
        Integer machineId = manual.getMachineId();
        Preconditions.checkNotNull(machineId,
"cosid.machine.distributor.manual.machineId can not be null.");
        Preconditions.checkArgument(machineId >= 0,
"cosid.machine.distributor.manual.machineId can not be less than 0.");
        return new ManualMachineIdDistributor(machineId, localMachineState,
clockBackwardsSynchronizer);
    }
```

未来如果你想要自己实现一个MachineId分配机制，也只需要参照这种方式，往里面注册一个MachineIdDistributor的实现类即可。不过，如果你真有这样的想法，那么我的建议是，慎重行事。因为目前版本cosid的扩展机制有点迷。

3、基于JDBC的工作进程ID分发机制实现分析

上层的这些接口其实还只是与Spring框架集成的一层入口。那么从MachineIdDistributor接口往下的具体实现，才算是进入了Cosid的核心。那么cosid是怎么实现机器位分配的呢？这就开始进入了真正让人迷糊的阶段了。

其实工作进程ID原本认为是一个比较简单的东西，只要在不同进程之间进行区分就行了。他并不需要有什么实际的意义。但是之前分析过，我觉得要注意下不要把这个工作进程ID和机器ID给搞混了。把这两个概念混到一起，反而更容易让人产生误解。

cosid定制了一套基础的机器位分发的流程，与每种第三方服务结合时，都是按这一套相同的流程工作。这个流程是什么样呢？那就从最熟悉的JDBC的实现机制往下看看把。其实这个问题，可以分两步来看。

首先：如何区分不同的工作进程？

cosid中区分不同的工作进程主要是依靠两个数据，cosid的命名空间 + 应用的IP和端口？？

其中命名空间可以在配置文件中通过cosid.namespace参数指定。这属于cosid自己的定义，没什么解释。

然后应用的IP可以直接通过应用读取。但是端口还是需要通过参数配置。

```
    @Bean
    @ConditionalOnMissingBean
    public InstanceId instanceId(InetUtils inetUtils) {
        //实例是否稳定
        boolean stable = Boolean.TRUE.equals(machineProperties.getStable());
        if (!Strings.isNullOrEmpty(machineProperties.getInstanceId())) {
            return InstanceId.of(machineProperties.getInstanceId(), stable);
        }
        InetUtils.HostInfo hostInfo = inetUtils.findFirstNonLoopbackHostInfo();
        int port = ProcessId.CURRENT.getProcessId();
        if (Objects.nonNull(machineProperties.getPort()) &&
machineProperties.getPort() > 0) {
            port = machineProperties.getPort();
        }
        return InstanceId.of(hostInfo.getIpAddress(), port, stable);
    }
```

> 这里又是一个很迷的地方。这个端口要自己配置是怎么回事？还要配置一下stable是否稳定？应用要是有那功夫单独为了这个主键分配功能配置上不同的端口，那为什么不直接配置一个机器ID算了？所以用到cosid的时候，大概率是不会配的。
>
> 如果不配，那么就只有IP作为区分维度了。也就是说一个机器上，如果跑多个服务实例，那么这些服务会被分配相同的machineid，那这样生成的雪花算法不就更容易重复了吗？
>
> 其实这个工作进程号，就是作为不同服务进程的一个区分标志，不需要跟实际的服务器有什么关系。既然要区分工作进程，为什么不是统一检查namespace，然后以namespace为维度，然后依托第三方服务，维护一个单调递增的短小精悍的分布式序列，自动分配不同的工作进程号？当然，可能cosid考虑的是machineid还要可以回收利用。比如machineid为1的进程，如果下线了，而这个1号进程又不是一个稳定的服务，那么后面的进程还可以重新分到1这个进程号。是不是这样呢？接着往下看吧。

然后：如何给不同的工作进程分发不同的MachineId?

分发MachineId时，首先有一层统一的入口逻辑，维护一个本地缓存。

```
 // me.ahoo.cosid.machine.AbstractMachineIdDistributor
    @Nonnull
    @Override
    public MachineState distribute(String namespace, int machineBit, InstanceId
instanceId, Duration safeGuardDuration) throws MachineIdOverflowException {

        Preconditions.checkArgument(!Strings.isNullOrEmpty(namespace), "namespace
can not be empty!");
        Preconditions.checkArgument(machineBit > 0, "machineBit:[%s] must be greater
than 0!", machineBit);
        Preconditions.checkNotNull(instanceId, "instanceId can not be null!");
        //维护本地缓存
        MachineState localState = machineStateStorage.get(namespace, instanceId);
        if (!MachineState.NOT_FOUND.equals(localState)) {
            //处理时钟回拨
```

```
 clockBackwardsSynchronizer.syncUninterruptibly(localState.getLastTimeStamp());
            return localState;
        }

        localState = distributeRemote(namespace, machineBit, instanceId,
safeGuardDuration);
        if
(ClockBackwardsSynchronizer.getBackwardsTimeStamp(localState.getLastTimeStamp()) >
0) {

 clockBackwardsSynchronizer.syncUninterruptibly(localState.getLastTimeStamp());
            localState = MachineState.of(localState.getMachineId(),
System.currentTimeMillis());
        }

        machineStateStorage.set(namespace, localState.getMachineId(), instanceId);
        return localState;
    }
```

这个本地缓存就跟之前看不懂的stable是否稳定扯上关系了。如果stable是true，那就基于本地文件进行持久化保存。 文件地址通过参数cosid.machine.state-storage.local.state-location指定。否则，就基于本地内存维护缓存，应用停止就消失了。这里似乎可以证明之前的猜想了。stable稳定的服务，就会占用稳定的machineid，就算应用停了，文件里还记着呢。

> 等等，文件？这个缓存是在应用端，文件有什么用？为Proxy服务用的？
>
> 另外，我也不是很理解，在工作进程分配这个问题上，处理时钟回拨有意义吗？分配工作进程的并发量不太可能达到分布式主键那么高吧。

后面的distributeRemote方法就是交由各种具体实现类去扩展实现的抽象方法了。例如JDBC的分发方式是这样的：

```
 @Override
    protected MachineState distributeRemote(String namespace, int machineBit,
InstanceId instanceId, Duration safeGuardDuration) {
        if (log.isInfoEnabled()) {
            log.info("Distribute Remote instanceId:[{}] - machineBit:[{}] @
namespace:[{}].", instanceId, machineBit, namespace);
        }
        try (Connection connection = dataSource.getConnection()) {
            //自己发
            MachineState machineState = distributeBySelf(namespace, instanceId,
connection, safeGuardDuration);
            if (machineState != null) {
                return machineState;
            }
            //回滚发?
            machineState = distributeByRevert(namespace, instanceId, connection,
safeGuardDuration);
            if (machineState != null) {
                return machineState;
```

```
            }
            //远程发
            return distributeMachine(namespace, machineBit, instanceId, connection);
        } catch (SQLException sqlException) {
            if (log.isErrorEnabled()) {
                log.error(sqlException.getMessage(), sqlException);
            }
            throw new CosIdException(sqlException.getMessage(), sqlException);
        }
    }
```

虽然各种服务的具体实现各不相同，但是基本的分发逻辑都是这三个步骤。 先自己发布，然后再回滚发布，然后再远程发布。

> 我也不知道这是什么意思。姑且就按字面意义这么叫把。

1、自己发布

执行的SQL语句是

```
select machine_id, last_timestamp from cosid_machine where namespace=? and
instance_id=? and last_timestamp>?
```

意思就是获取当前实例获取过的machine_id。不过在分配时，会根据last_timestamp进行安全监测。简单来说，就是只获取在安全时间内分配过的ID。安全时间外的不算。这个安全时间，如果对于stable稳定的机器，那么安全时间就是从0开始。不稳定的机器，安全时间可以通过参数cosid.machine.guarder.safe-guard-duration指定。默认5分钟。然后源码中甚至还定了一个永久的安全时间。Duration FOREVER_SAFE_GUARD_DURATION = Duration.ofMillis(Long.MAX_VALUE); 这样也会忽略安全时间的查询条件。

如果查到了历史记录，那么就更新last_timestamp，然后返回历史的machine_id。如果没查到，就进行回滚发布。

2、回滚发布

执行的SQL语句是

```
select machine_id, last_timestamp from cosid_machine where namespace=? and
(instance_id='' or last_timestamp<=?)
```

这个意思应该是获取别的进程不用了的MachineId。可能是无人认领的，也可能是超过了安全时间的。查到了就更新instance_id，last_timestamp和distribute_time，然后返回历史的machine_id。如果没查到，就进行远程发布。

> 是不是表示认领不包含具体实例的公共machine_id？但是我把源码看到最后也没看到instance_id=''的数据是怎么插入进去的。

3、远程发布

远程发布时，获取机器ID的SQL是

```
select max(machine_id)+1 as next_machine_id from cosid_machine where namespace=?
```

从MySQL中重新分配一个新的machine_id。获取到的next_machine_id就是分配的机器ID。如果没有记录，就返回1。获取完机器ID后，就会往cosid_machine里插入一条记录，把这个分配的机器ID记录下来。

> 虽然这样每获取一次MachineId就会要往MySQL里插入一条数据，但是已插入的旧数据还可以被后面的进程重复利用，所以使用的效率还是挺高的。并且这个流程很容易移植到其他服务中。例如MongoDB。cosid的其他几种服务实现也都按照这样一个统一的流程。
>
> 不过我还是觉得，这样的历史数据还是有可能太多把。而一旦数据多了，对性能的影响就会越来越大。

3、Segment数据段模式

1、基础使用

cosid使用segment号段也非常方便。应用中也是只要从Spring容器里获取IdGeneratorProvider实例，然后通过这个实例获取分布式ID就行。唯一需要修改的就是配置信息。

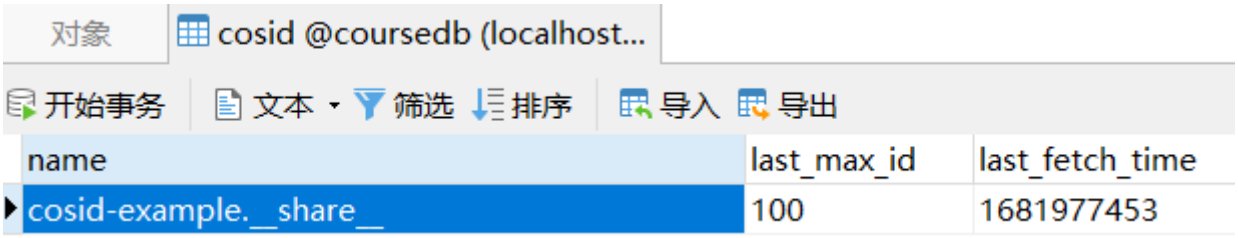
以最常用的JDBC为例，pom依赖已经在上一个章节当中添加完了，这里直接改配置就可以换成segment的实现。

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/coursedb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root

cosid.namespace=cosid-example
cosid.enabled=true

cosid.snowflake.enabled=false
#machineid还是要注入
cosid.machine.enabled=true
cosid.machine.distributor.type=jdbc
#使用segment模式
cosid.segment.enabled=true
#单segment模式，chain:segmentchain模式
cosid.segment.mode=default
cosid.segment.distributor.type=jdbc
#初始化建表
cosid.segment.distributor.jdbc.enable-auto-init-cosid-table=true
#安全距离，segment缓存数量 默认10
#cosid.segment.chain.safe-distance=10
#步数，每个segment里的ID段
cosid.segment.share.step=100
```

改完配置之后，就可以运行之前的单元测试案例获取分布式ID了。

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class DistIDTest {
    @Resource
    private IdGeneratorProvider provider;

    @Test
    public void getId(){
        for (int i = 0; i < 100; i++) {
            System.out.println(provider.getShare().generate());
        }
    }
}
```

执行完成后，会在MySQL中自动创建一张cosid表。里面记录了主键的segment信息。这次不用手动建表了

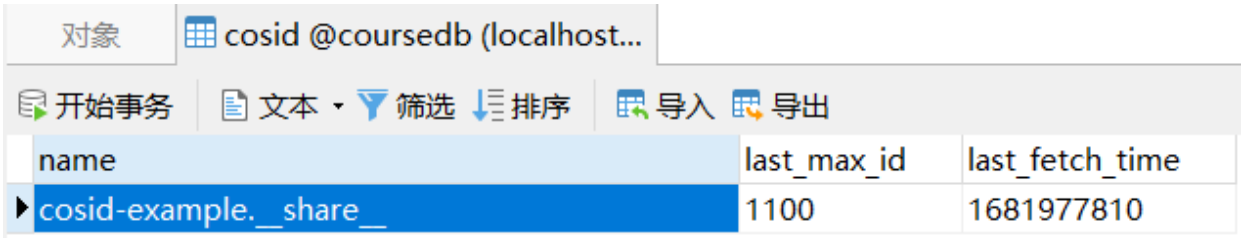


对照之前对segment模式的理解，这个数据是很容易看懂的。cosid表中记录了当前命名空间的last_max_id，应用就可以自由分配0~100的ID。但是之前分析过，这种单segment模式，在这一批号段用完之后，就需要重新向数据库申请。申请的过程中，分布式ID服务是短暂不可用的。这时，可以升级到segmentChain模式。使用也非常简单，只要修改一个配置，将cosid.segment.mode改为chain。

```
cosid.segment.mode=chain
```

name当中的__share__是IdGeneratorProvider当中的一个容器名字，用来保存这些IdGenerator的。这个是他的默认名字。这也是可以通过参数配置的，不记得哪个参数了。不过除了强迫症，应该是没什么人愿意特意去修改这个名字的。

然后再次执行之前的单元测试案例，获取100个ID，这时再来观察cosid表，last_max_id更新成了1100。

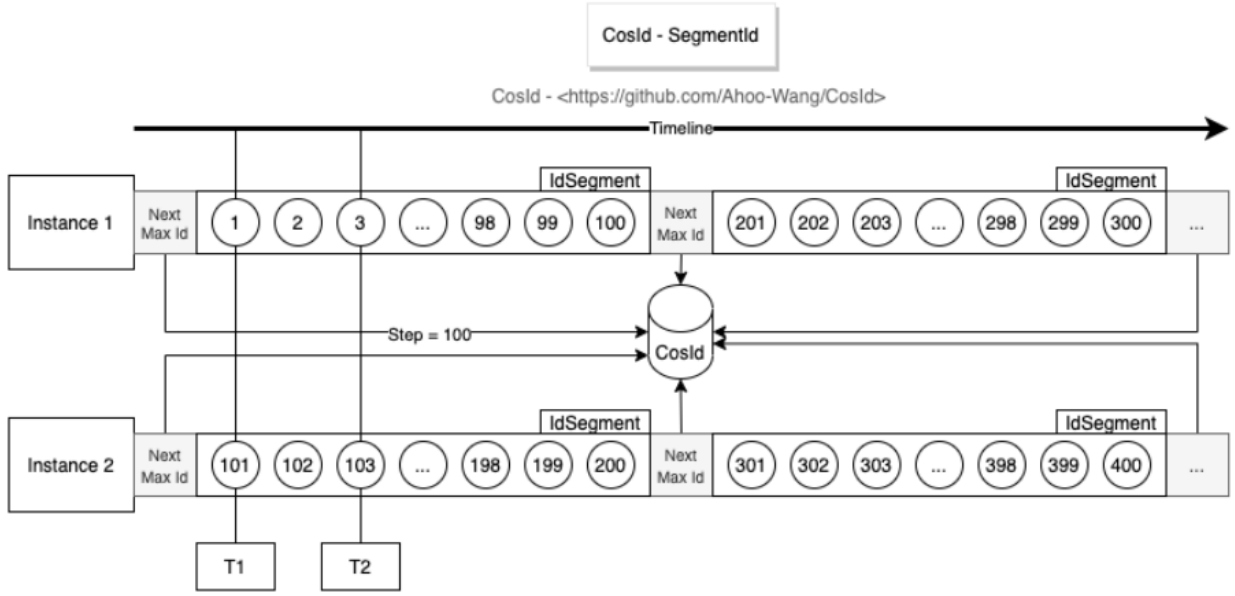


为什么这次直接增大了1000呢？这就是因为改成chain模式后，cosid会去构建一个有10个(cosid.segment.chain.safe-distanced的默认值)segment的链表。

2、重点机制剖析

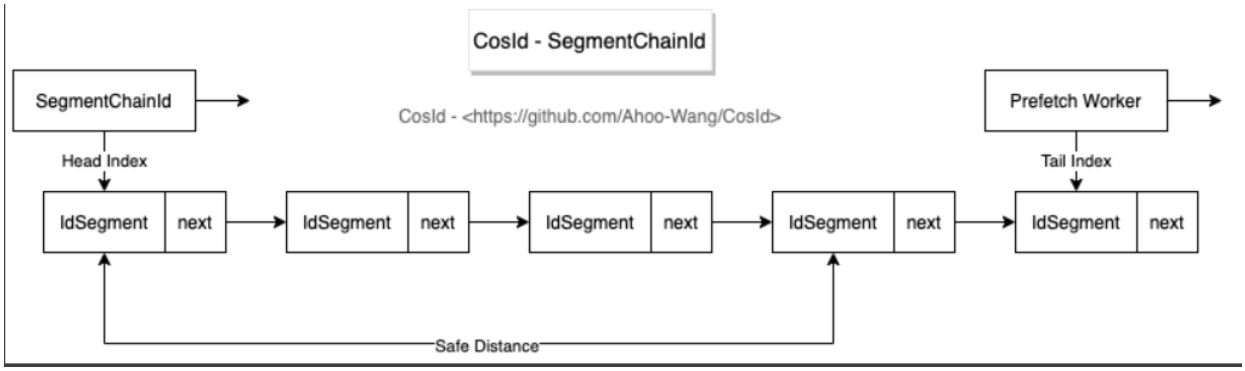
这两种模式的基础设计官网已经有明确介绍

跟之前介绍的Segment策略是一样的，也是缓存一个单独的号段。一个号段用完了，就再去申请下一个号段。之前分析过，在申请新号段的过程中，服务是短暂不可用的。



对于Segment号段，需要维护两个核心参数，NextMaxId和Step。一个表示已经分配的最大ID，一个表示每次分配号段的步长。

SegmentChainID号段链模式的基础思路之前也介绍过。用一个链表结构把多个segment串起来。整个设计图是这样的：



怎么理解他这种设计方式呢？结合之前的示例，就很容易理解了。在cosid中，对每个segment，同样是定义了maxid和step。其中step默认定义为100。然后，他定义了一个Safe Distance，默认值是10。这样，在启动时，Cosid就会构建一个有10个segment组成的segmentChain。每次分配ID时，也都是从头部的segment开始分配。如果第一个segment分配完了，就从第二个segment继续分配。但与此同时，检查segmentChain中的segment个数，如果少于SafeDistance，那么就开始尝试构建新的segment，并往链表当中添加。

在实现时，这两种方式其实也抽象出了相同的接口。我们同样可以来个简单的单元测试案例入手。

```
    @Resource
    private SegmentId segmentId;

    @Test
    public void getId(){
        for (int i = 0; i < 100; i++) {
            System.out.println(segmentId.generate());
        }
    }
```

而这个SegmentId接口下，就有两个具体的实现类，分别实现单Segment模式和SegmentChain模式。注入方式如下：

```
    //me.ahoo.cosid.spring.boot.starter.segment.CosIdSegmentAutoConfiguration
    private static SegmentId createSegment(SegmentIdProperties segmentIdProperties,
SegmentIdProperties.IdDefinition idDefinition, IdSegmentDistributor
idSegmentDistributor,
                                           PrefetchWorkerExecutorService
prefetchWorkerExecutorService) {
        long ttl = MoreObjects.firstNonNull(idDefinition.getTtl(),
segmentIdProperties.getTtl());
        SegmentIdProperties.Mode mode =
MoreObjects.firstNonNull(idDefinition.getMode(), segmentIdProperties.getMode());
        SegmentId segmentId;
        //构建segmentId实例
        if (SegmentIdProperties.Mode.DEFAULT.equals(mode)) {
            segmentId = new DefaultSegmentId(ttl, idSegmentDistributor);
        } else {
            SegmentIdProperties.Chain chain =
MoreObjects.firstNonNull(idDefinition.getChain(), segmentIdProperties.getChain());
            segmentId = new SegmentChainId(ttl, chain.getSafeDistance(),
idSegmentDistributor, prefetchWorkerExecutorService);
        }

        IdConverterDefinition converterDefinition = idDefinition.getConverter();
        //结果转换器
        IdConverter idConverter = ToStringIdConverter.INSTANCE;
        switch (converterDefinition.getType()) {
            case TO_STRING: {
                IdConverterDefinition.ToString toString =
converterDefinition.getToString();
                if (toString != null) {
                    idConverter = new ToStringIdConverter(toString.isPadStart(),
toString.getCharSize());
                }
                break;
            }
            case RADIX: {
                IdConverterDefinition.Radix radix = converterDefinition.getRadix();
                idConverter = Radix62IdConverter.of(radix.isPadStart(),
radix.getCharSize());
```

```
                break;
            }
            default:
                throw new IllegalStateException("Unexpected value: " +
converterDefinition.getType());
        }

        if (!Strings.isNullOrEmpty(converterDefinition.getPrefix())) {
            idConverter = new PrefixIdConverter(converterDefinition.getPrefix(),
idConverter);
        }
        if (!Strings.isNullOrEmpty(converterDefinition.getSuffix())) {
            idConverter = new SuffixIdConverter(converterDefinition.getSuffix(),
idConverter);
        }
        return new StringSegmentId(segmentId, idConverter);

    }
```

然后，这个生成的SegmentId实例构建出来后，会添加到统一的idGeneratorProvider中。

```
    //me.ahoo.cosid.spring.boot.starter.segment.CosIdSegmentAutoConfiguration
    @Bean
    @ConditionalOnMissingBean
    public SegmentId shareSegmentId(IdSegmentDistributorFactory distributorFactory,
IdGeneratorProvider idGeneratorProvider, PrefetchWorkerExecutorService
prefetchWorkerExecutorService) {
        SegmentIdProperties.IdDefinition shareIdDefinition =
segmentIdProperties.getShare();
        IdSegmentDistributorDefinition shareDistributorDefinition =
asDistributorDefinition(IdGeneratorProvider.SHARE, shareIdDefinition);
        IdSegmentDistributor shareIdSegmentDistributor =
distributorFactory.create(shareDistributorDefinition);
        //构建实例
        SegmentId shareIdGen = createSegment(segmentIdProperties, shareIdDefinition,
shareIdSegmentDistributor, prefetchWorkerExecutorService);
        //添加到idGeneratorProvider中
        if (Objects.isNull(idGeneratorProvider.getShare())) {
            idGeneratorProvider.setShare(shareIdGen);
        }

        if (segmentIdProperties.getProvider().isEmpty()) {
            return shareIdGen;
        }
        //根据定义生成实例。应该没人会想要用这种方式把。
        segmentIdProperties.getProvider().forEach((name, idDefinition) -> {
            IdSegmentDistributorDefinition distributorDefinition =
asDistributorDefinition(name, idDefinition);
            IdSegmentDistributor idSegmentDistributor =
distributorFactory.create(distributorDefinition);
```

```
            SegmentId idGenerator = createSegment(segmentIdProperties, idDefinition,
idSegmentDistributor, prefetchWorkerExecutorService);
            idGeneratorProvider.set(name, idGenerator);
        });
        return shareIdGen;
    }
```

通过这个注入的SegmentId对象，就可以生成分布式ID了。单Segment模式的实现比较简单，就是获取到号段之后分配呗。分配完了就去重新申请。

```
    //me.ahoo.cosid.segment.DefaultSegmentId
    @Override
    public long generate() {
        if (maxIdDistributor.getStep() == ONE_STEP) {
            return maxIdDistributor.nextMaxId();
        }
        long nextSeq;
        if (segment.isAvailable()) {
            nextSeq = segment.incrementAndGet();
            if (!segment.isOverflow(nextSeq)) {
                return nextSeq;
            }
        }

        synchronized (this) {
            while (true) {
                if (segment.isAvailable()) {
                    nextSeq = segment.incrementAndGet();
                    if (!segment.isOverflow(nextSeq)) {
                        return nextSeq;
                    }
                }
                IdSegment nextIdSegment =
maxIdDistributor.nextIdSegment(idSegmentTtl);
                segment.ensureNextIdSegment(nextIdSegment);
                segment = nextIdSegment;
            }
        }
    }
```

cosid真正核心的就是他的segmentchain的模式。他的generate方法如下：

```
    //me.ahoo.cosid.segment.SegmentChainId
    @Override
    public long generate() {
        //找到一个可用的segment，分发ID
        while (true) {
            IdSegmentChain currentChain = headChain;
            while (currentChain != null) {
                if (currentChain.isAvailable()) {
                    long nextSeq = currentChain.incrementAndGet();
```

```
                    if (!currentChain.isOverflow(nextSeq)) {
                        forward(currentChain);
                        return nextSeq;
                    }
                }
                currentChain = currentChain.getNext();
            }
            //找不到，链表空了就添加一个。
            try {
                final IdSegmentChain preIdSegmentChain = headChain;

                if (preIdSegmentChain.trySetNext((preChain) ->
generateNext(preChain, safeDistance))) {
                    IdSegmentChain nextChain = preIdSegmentChain.getNext();
                    forward(nextChain);
                    if (log.isDebugEnabled()) {
                        log.debug("Generate [{}] - headChain.version:[{}->{}].",
maxIdDistributor.getNamespacedName(), preIdSegmentChain.getVersion(),
nextChain.getVersion());
                    }
                }
            } catch (NextIdSegmentExpiredException nextIdSegmentExpiredException) {
                if (log.isWarnEnabled()) {
                    log.warn("Generate [{}] - gave up this next IdSegmentChain.",
maxIdDistributor.getNamespacedName(), nextIdSegmentExpiredException);
                }
            }
            //通过hungry模式激发prefetchService去检查链表上的segment是否充足
            this.prefetchJob.hungry();
        }
    }
```

然后官网提到过，SegmentChain模式是通过一个PrefetchWorker，异步进行链表扩充。PrefetchWorker实际是一个线程池，通过不断的执行prefetchJob任务，来尽量保证SegmentChain的长度是足够的。

线程调度的逻辑这里就不多做梳理了，挺多挺杂的。核心扩充Segment的逻辑就在这个prefetchJob当中。

```
//me.ahoo.cosid.segment.SegmentChainId
    public class PrefetchJob implements AffinityJob {
        public void prefetch() {
            long wakeupTimeGap = Clock.CACHE.secondTime() - lastHungerTime;
            final boolean hunger = wakeupTimeGap < hungerThreshold;
            //计算安全距离。也就是链表当中的Segment个数是否满足。
            final int prePrefetchDistance = this.prefetchDistance;
            if (hunger) {
                this.prefetchDistance =
Math.min(Math.multiplyExact(this.prefetchDistance, 2), MAX_PREFETCH_DISTANCE);
                if (log.isInfoEnabled()) {
                    log.info("Prefetch [{}] - Hunger, Safety distance expansion.[{}-
>{}]", maxIdDistributor.getNamespacedName(), prePrefetchDistance,
this.prefetchDistance);
```

```
                }
            } else {
                this.prefetchDistance =
Math.max(Math.floorDiv(this.prefetchDistance, 2), safeDistance);
                if (prePrefetchDistance > this.prefetchDistance) {
                    if (log.isInfoEnabled()) {
                        log.info("Prefetch [{}] - Full, Safety distance shrinks.[{}-
>{}]", maxIdDistributor.getNamespacedName(), prePrefetchDistance,
this.prefetchDistance);
                    }
                }
            }
            IdSegmentChain availableHeadChain = SegmentChainId.this.headChain;
            while (!availableHeadChain.getIdSegment().isAvailable()) {
                availableHeadChain = availableHeadChain.getNext();
                if (availableHeadChain == null) {
                    availableHeadChain = tailChain;
                    break;
                }
            }
            forward(availableHeadChain);
            //计算链表当中的segment数量
            final int headToTailGap = availableHeadChain.gap(tailChain,
maxIdDistributor.getStep());
            //计算与链表安全个数之间的差距。
            final int safeGap = safeDistance - headToTailGap;
            //链表中segment个数已经不够了。但是不急着要。
            if (safeGap <= 0 && !hunger) {
                if (log.isTraceEnabled()) {
                    log.trace("Prefetch [{}] - safeGap is less than or equal to 0,
and is not hungry - headChain.version:[{}] - tailChain.version:[{}].",
maxIdDistributor.getNamespacedName(),
                        availableHeadChain.getVersion(), tailChain.getVersion());
                }
                return;
            }
            //需要添加几个segment
            final int prefetchSegments = hunger ? this.prefetchDistance : safeGap;
            //申请并添加segment到链表当中
            appendChain(availableHeadChain, prefetchSegments);
        }
    }
```

这里核心的hunger模式，其实就是用来保证就算数据库不可用了， 也还是用自己的segmentChain先撑着。只要数据库可用，马上开始扩充Segment。

3、基于JDBC的ID分发机制实现分析

基于JDBC的ID分发器首先需要扩展一个IdSegmentDistributorFactory工厂类，用来构建IdSegmentDistributor。这个IdSegmentDistributor就是用来申请NextMaxId的。JDBC的扩展中就实现了一个工厂类。

```
public class JdbcIdSegmentDistributorFactory implements IdSegmentDistributorFactory
{
    private final DataSource dataSource;
    private final boolean enableAutoInitIdSegment;
    private final JdbcIdSegmentInitializer jdbcIdSegmentInitializer;
    private final String incrementMaxIdSql;
    private final String fetchMaxIdSql;

    public JdbcIdSegmentDistributorFactory(DataSource dataSource, boolean
enableAutoInitIdSegment, JdbcIdSegmentInitializer jdbcIdSegmentInitializer, String
incrementMaxIdSql, String fetchMaxIdSql) {
        this.dataSource = dataSource;
        this.enableAutoInitIdSegment = enableAutoInitIdSegment;
        this.jdbcIdSegmentInitializer = jdbcIdSegmentInitializer;
        this.incrementMaxIdSql = incrementMaxIdSql;
        this.fetchMaxIdSql = fetchMaxIdSql;
    }

    @Nonnull
    @Override
    public IdSegmentDistributor create(IdSegmentDistributorDefinition definition) {
        if (enableAutoInitIdSegment) {

 jdbcIdSegmentInitializer.tryInitIdSegment(definition.getNamespacedName(),
definition.getOffset());
        }
        return new JdbcIdSegmentDistributor(
            definition.getNamespace(), definition.getName(), definition.getStep(),
            incrementMaxIdSql, fetchMaxIdSql, dataSource
        );
    }
}
```

另外，当链表当中需要扩充Segment时，就通过扩展JdbcIdSegmentDistributor的nextMaxId方法实现。

```
@Override
    public long nextMaxId(long step) {
        IdSegmentDistributor.ensureStep(step);
        try (Connection connection = dataSource.getConnection()) {
            connection.setAutoCommit(false);
            try (PreparedStatement accStatement =
connection.prepareStatement(incrementMaxIdSql)) {
                accStatement.setLong(1, step);
                accStatement.setString(2, getNamespacedName());
                int affected = accStatement.executeUpdate();
```

```
                if (affected == 0) {
                    throw new SegmentNameMissingException(getNamespacedName());
                }
            }

            long nextMaxId;
            try (PreparedStatement fetchStatement =
connection.prepareStatement(fetchMaxIdSql)) {
                fetchStatement.setString(1, getNamespacedName());
                try (ResultSet resultSet = fetchStatement.executeQuery()) {
                    if (!resultSet.next()) {
                        throw new NotFoundMaxIdException(getNamespacedName());
                    }
                    nextMaxId = resultSet.getLong(1);
                }
            }
            connection.commit();
            return nextMaxId;
        } catch (SQLException sqlException) {
            if (log.isErrorEnabled()) {
                log.error(sqlException.getMessage(), sqlException);
            }
            throw new CosIdException(sqlException.getMessage(), sqlException);
        }
    }
```

其实，看懂这两个SQL语句，大致就明白什么情况了。

```
    public static final String INCREMENT_MAX_ID_SQL
        = "update cosid set last_max_id=(last_max_id +
?),last_fetch_time=unix_timestamp() where name = ?;";
    public static final String FETCH_MAX_ID_SQL
        = "select last_max_id from cosid where name = ?;";
```

六、总结

在这个不短的过程当中，我们从一个简单的分库分表常见问题入手，又借着了解ShardingSphere新接入的Cosid框架的机会，不太全面也不太深入的把分布式ID这样一个不太起眼的问题详细总结了一下。分布式主键生成策略，这或许是一个不起眼的技术路线，但是当他与具体业务结合时，却也是一个很重要的技术。

一方面，其实我也不明白，雪花算法导致取模分片数据不均匀的问题，其实应该是很常见，很容易发现的。我也不知道为什么好像之前从来没看到有人特意分析过这样的问题。或许是因为在实际分库分表的业务场景，直接用雪花算法的很少把。但是，其实从这一个小点可以看出，分库分表无小事啊。虽然我对分库分表已经算是很熟悉了， 但是对于分库分表，给大家的建议还是，能不分就不分。虽然都是分布式的问题，但是分库分表不像微服务，微服务只要用得差不多，总能看到一些好处。但是分库分表，如果没有一个深入挖掘问题的心，那么就轻易不要动了吧。

另一方面，我希望这也给了大家一个学习新框架的思路把。其实后面在了解CosID的时候，我经历的过程比这篇文章中的过程曲折得多，经历了各种莫名其妙的版本冲突、配置错误、使用报错。可是网上能给的帮助却非常优先。最后还是结合官网简短的介绍，再一点点慢慢抠源码，才算把这思路大致给整理清楚了。而等思路整理清楚了之后，再来看CosID框架，虽然还是有点小瑕疵，但是，他的很多设计处理思路，确实是分布式场景下非常有用的。这应该就是程序员成长的经验，也希望这能够成为你以后的经验。

> 有道云笔记链接：https://note.youdao.com/s/Z6ISUBc3