



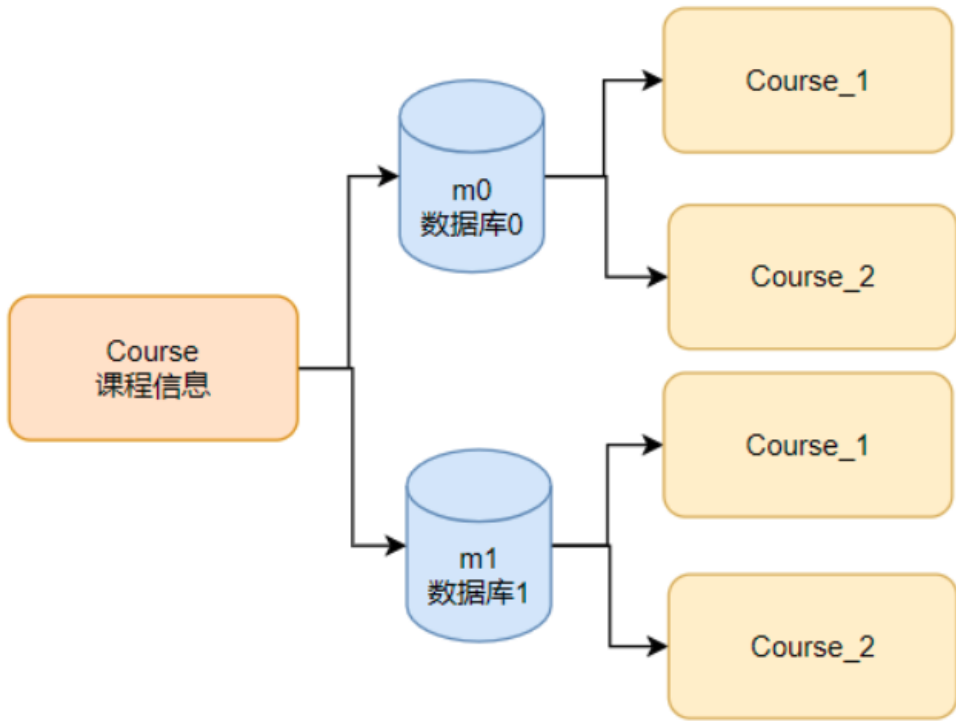
-- 楼兰

前面几个章节，带你体验了非常多ShardingSphere的功能。有没有那么一刻，你会觉得分库分表也没那么麻烦。用好ShardingSphere框架就是了。但分表所带来的问题，其实远比你想象的复杂。这次，我带你一起来看看ShardingSphere5.x版本集成了一个新的主键生成框架CosId，看看分布式主键么一个不起眼的问题，水能够有多深。只有你有足够能力自己去研究分库分表这些问题，你才能真正融会贯通，把ShardingSphere框架真正当成一个具来用，而不是一个呆板的框架。

## 一、从分库分表的一个小坑说起

或许你会觉得我小题大做了。那我们不多啰嗦，从一个分库分表的小实验开始。

现在，我想要将一个course表的数据分到两个库两张表，一共四个分片中。这是一个最典型的分库分表的场景。



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 $m \rightarrow \{cid \% 2\}$ 就行了。而分表的算法呢？如果也是按照cid奇偶拆分，算法定制为 $course\_ \rightarrow \{cid \% 2 + 1\}$ 。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  $c \rightarrow \{((cid + 1) \% 4).intdiv(2) + 1\}$ 。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。

```
6 public class SnowFlakeTest { new *
7
8 public static void main(String[] args) { new *
9     //如果ID是连续的，那么数据能够很平均的分到两个库的两个片里。
10    for (long i = 0; i < 100; i++) {
11        long database = i % 2;
12        long table = ((i + 1) % 4) / 2 + 1;
13        System.out.println("主键: "+i+";库分片: "+database +":表分片"+table);
14    }
```

Run SnowFlakeTest x

/Users/roykingw/tools/jdk/jdk-17.0.8.jdk/Contents/Home/bin/java ...

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  
主键: 7;库分片: 1:表分片1  
主键: 8;库分片: 0:表分片1

数据分布很均匀

算法验证完成，接下来，配置到ShadingSphere中使用一下。下面是示例配置：

```
# 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://192.168.65.212:3306/shardingdb1?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://192.168.65.212:3306/shardingdb2?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.type=COSID_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}
```

应用层，就直接往course表里插入多条消息

```
@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.setName("java");
        c.setUserId(1001L);
        c.setCstatus("1");
        courseMapper.insert(c);
        //insert into course values ....
        System.out.println(c);
    }
}
```

那么你会发现，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\_m2.course\_2两张表中。要怎么解决呢？

解决方案很简单，将表分片算法的type换成COSID\_SNOWFLAKE。

```
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.type=COSID_SNOWFLAKE
```

再次尝试，course表数据就能均匀的分配到四张表中。

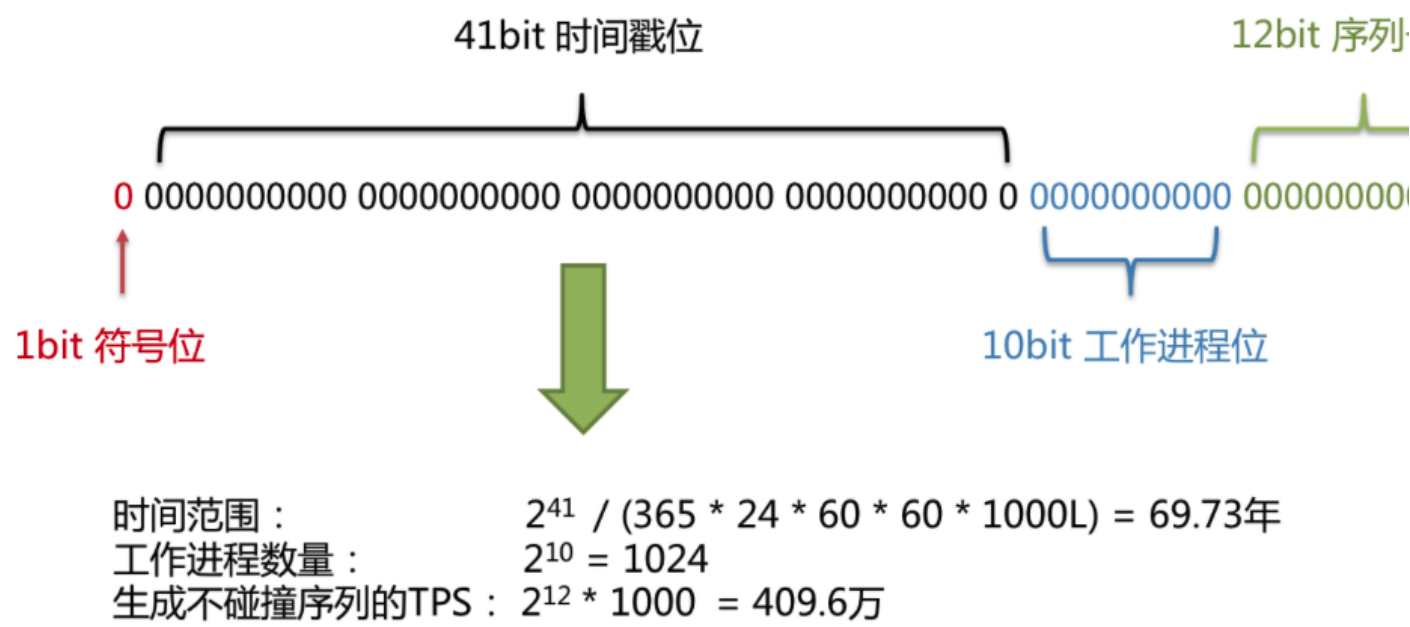
为什么会这样呢？这就需要你能够真正理解在分库分表的场景下，要怎么解决分布式主键生成这么一个看似不起眼的问题。

## 二、雪花算法详细拆解

### 1、什么是雪花算法

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个字节正好就是一个Long类型的变量。即保持足够的区分度，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，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。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，那就会产生相同的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，将工作进程位加一部分成为datacenter和worker两个部分，等等，但是其实万变不离其宗。

### 2、COSID\_SNOWFLAKE如何解决取模分片数据不均匀的问题

回到我们之前说的取模分片数据不均匀的问题。

首先，你要明白一个数学规律。对于任何一个数字，对2取模的结果，实际上就是在取这个数字的二进制表达时的最后一位。对4取模的结果，实际上就是取这个数字的二进制表达时的最后两位。依次类推。所以，回到我们的问题。要让数据均匀分到四个真实片，那么实际上是需要保证生成的一系列雪花算法ID的二进制表达的最后两位是连续递增的。

然后，回到之前的问题。自然就是要比对SNOWFLAKE算法和COSID\_SNOWFLAKE算法他们的最后一个序列位有什么区别。

到现在，你应该能够找到分库分表中配置SNOWFLAKE和COSID\_SNOWFLAKE两种不同算法，分别对应的源码在哪里了。那么我们直接拿来比较。

先来看SNOWFLAKE对应算法实现类是SnowflakeKeyGenerateAlgorithm。他是这样生成雪花算法ID的。

```

@Override
public synchronized Long generateKey() {
    long currentMilliseconds = timeService.getCurrentMillis();
    if (waitTolerateTimeDifferenceIfNeed(currentMilliseconds)) {
        currentMilliseconds = timeService.getCurrentMillis();
    }
    // 时间重复, 序列位就加1。
    if (lastMilliseconds == currentMilliseconds) {
        if (0L == (sequence = (sequence + 1) & SEQUENCE_MASK)) {
            currentMilliseconds = waitUntilNextTime(currentMilliseconds);
        }
    } else {
        //如果时间更新了, 序列位就会重置
        vibrateSequenceOffset();
        sequence = sequenceOffset;
    }
    lastMilliseconds = currentMilliseconds;
    return ((currentMilliseconds - EPOCH) << TIMESTAMP_LEFT_SHIFT_BITS) | (getWorkerId() << WORKER_ID_LEFT_SHIFT_BITS) |
sequence;
}

```

在这种雪花算法下, 只要两次生成ID的时间不同, 那么这个sequence就会在0和1之间震荡。而在我们的项目中, 每生成一次ID后, 还有写入数据库的作, 时间必然是要往后推延的。这样, 对4取模的结果就当然只能有0或1这两个结果。2和3对应的两个分片就分不到了。

具体查看vibrateSequenceOffset方法。默认情况下, 他会让sequenceOffset分别在0和1之间震荡。

实际上, 如果你看懂了源码。就会发现, 在使用SNOWFLAKE算法时, 如果在props中增加配置一个参数 max-vibration-offset=12。那么这个sequence, 就可以从0递增到10。这样, 也是可以解决之前的数据分配不均匀的问题。也就是

```

spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.type=SNOWFLAKE
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.props.worker-id=1
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.props.max-vibration-offset=12

```

但是这个莫名其妙的配置, 除了源码, 你找不到任何其他资料说明。

然后再来看COSID\_SNOWFLAKE算法生成雪花ID的过程。他的源码在这个地方:

```

//me.ahoo.cosid.snowflake.AbstractSnowflakeId类
@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直接递增。到达maxSequence后再重置。
    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
        | sequence;
}

```

可以看到, 对于sequence序列位。CosID提供的实现就简单粗暴得多。在达到maxSequence最大值之前, sequence都是直接递增的。这样递增的结果花ID的二进制最后几位, 都是严格递增的, 数据自然也就分布均匀了。

这还只是雪花算法中的最后序列位。实际上，在分库分表场景下，雪花算法的问题还不仅仅在于最后的序列位。

下一个问题，就是雪花算法中间的工作进程位。之前分析过，雪花算法的工作进程位是用来区分不同的工作进程的。也就是说，如果我们的这个服务是布式的微服务，那么每一个服务的工作进程位都应该是要不同的。但是，在实际项目中，这个小小的问题其实是很难的。

一方面，绝大部分程序员在用的时候，不会专门为了雪花算法单独设置工作进程位。例如在ShardingSphere中，实际上是可以给SNOWFLAKE主键生worker-id参数来设置进程位的。

```
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.type=SNOWFLAKE
spring.shardingsphere.rules.sharding.key-generators.alg_snowflake.props.worker-id=1
```

但是，这个worker-id参数，就连官方文档中也没有做过单独的说明，基本也就不可能要求所有人在用SNOWFLAKE时都去单独设置了。

另一方面，要给每个进程设置一个不重复的工作进程位，是有点困难的。如果只是简单的一两个服务，那么手动指定一下worker-id参数也就完成了。如果是一个几十个服务的大型微服务系统呢？你如何保证程序员或者运维人员能够保证这几十个服务的worker-id是不重复的？这基本上就是一个不可能任务。

这个问题很隐蔽，之前基本上很少有人想到这个事情。但是，别急，这个小小的COSID框架想到了。接下来，我带你去COSID的源码当中做详细的拆解。

## 三、深入源码全面理解CosID框架

### 1、搭建CosID测试应用

虽然CosID目前已经集成进了ShardingSphere，但是，实际的情况是，ShardingSphere默认只集成了CosID的一部分功能，并没有全部集成进来。这CosID实际上有很多核心功能是需要额外的存储系统的。这必然给ShardingSphere带来更大的复杂性。

这么说有点虚，你可能还难以理解。所以，这次我们单独搭建一个CosID的测试应用，把CosID拆解明白了，你就知道怎么回事了。

搭建步骤，还是以SpringBoot为核心，来引入CosID的支持。

step1、在之前示例项目中增加一个模块CosIDDemo。其中的pom.xml依赖

```
<properties>
    <cosid.version>2.9.1</cosid.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>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

注意这个cosid组件的版本。不同的版本是有一些不同的小坑的。

step2、启动类。没什么特别的

```
@SpringBootApplication
public class DistIDApp {

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

step3、application.properties配置文件

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

有了前面的铺垫，你应该能猜到这个配置是用来生成一个雪花ID的，其中machine就是工作进程位1。

step4、应用中生成主键

在应用中使用CosID就非常简单了。只要从Spring的IOC容器中获取一个IdGeneratorProvider实例，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());
        }
    }
}
```

实际上，ShardingSphere中集成CosId框架的原理也是这样的，通过对IdGeneratorProvider进行封装，从而获取主键。

唯一需要特别注意的是SpringSphere中集成的CosID的版本，没有目前案例新。案例当中默认集成的cosid版本是1.14.1。在这个版本下，有一个小坑，需要在SpringBoot的启动类上增加如下两个注解才能用。--现在最新的2.9.1版本已经不需要了。

```
@EnableConfigurationProperties({MachineProperties.class})
@ComponentScans(value = {@ComponentScan("me.ahoo.cosid")})
```

CosID框架主要集成了三种主键生成模式，1、SnowFlake雪花算法。2、SegmentID号段模式。3、SegmentChainID号段链模式。其中后两种的思路的，都是用的Segment号段模式，只是实现思路不同。主要用来生成严格递增的主键序列。

而这些不同的生成模式，在应用层面，统一由IdGeneratorProvider提供服务。也就是说，应用代码不用做任何调整，只需要调整相关配置，就可以生成型的分布式主键。

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。在之前的演示中，这个machineID就是雪花算法的工作进程位，不过之前配置的manual方式，是手动设置的。那么按照之前的分析，他依然会有与大型项目水土不服的问题。

那么怎么解决呢？当然是自动生成。这其实是不容易的。主键生成框架是要生成一系列唯一的主键。现在为了生成主键，又要先生成一系列不唯一的machineID。这不就成了一个鸡生蛋，蛋生鸡的问题了。怎么办呢？来看看CosID的处理方式。

CosID对于MachineID提供了多种实现形式。关于这些可选方式，具体可以看下他源码当中的这个枚举类型：



```
//me.ahoo.cosid.spring.boot.starter.machine.MachineProperties
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
}
```

这里可以引入很多的第三方存储来辅助分发machineID。

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>${cosid.version}</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.20</version>
    <!-- 版本冲突 -->
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-autoconfigure</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>${spring.boot.version}</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
</dependency>
```

其中cosid-jdbc是cosid的核心扩展包。而其他相关依赖则是SpringBoot应用操作MySQL数据库所需要的一系列依赖。cosid和mybatis等框架类似，它从Spring容器当中获取DataSource数据源，而不关心如何构建DataSource。

接下来，修改应用的配置文件

```
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://192.168.65.212:3306/test?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
```

接下来，需要创建对应的数据库，并且还需要在数据库中手动创建一张表。建表语句为；

```
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);
```

好了，如果没有依赖版本冲突，那么接下来就可以愉快的跑单元测试案例，获取分布式ID了。

## 2、重点机制剖析

关于Cosid的雪花算法，之前已经做了铺垫。这里重点要了解的是他如何给雪花算法生成中间的那一段MachineID。关于这个问题，另外做一个单元测试了。

```
public class SnowFlakeTest {

    @Resource
    private MachineId machineId;
    @Resource
    private SnowflakeId snowflakeId;

    @Test
    public void snowflakeTest(){
        System.out.println("machineId:"+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。然后这个MachineId实例，会被一个SnowFlakeId实例引用，生成雪花算法。最终，这个SnowFlakeId也会被IdGeneratorProvider引用。

接下来，我们就一起到源码当中逛逛，看下这些具体的实例是如何构建的。

首先，雪花算法的SnowFlakeId示例的注入方式是这样的：

```

// me.ahoo.cosid.spring.boot.starter.snowflake.SnowflakeIdBeanRegistrar
// 注册Bean
public void register() {
    if (customizeSnowflakeIdProperties != null) {
        customizeSnowflakeIdProperties.customize(snowflakeIdProperties);
    }
    SnowflakeIdProperties.ShardIdDefinition shareIdDefinition = snowflakeIdProperties.getShare();
    if (shareIdDefinition.isEnabled()) {
        // 核心构建方法
        registerIdDefinition(IdGeneratorProvider.SHARE, shareIdDefinition);
    }
    snowflakeIdProperties.getProvider().forEach(this::registerIdDefinition);
}

private void registerIdDefinition(String name, SnowflakeIdProperties.IdDefinition idDefinition) {
//创建SnowFlakeId
    SnowflakeId idGenerator = createIdGen(idDefinition, clockBackwardsSynchronizer);
//注册到IdGeneratorProvider中
    registerSnowflakeId(name, idGenerator);
}

private void registerSnowflakeId(String name, SnowflakeId snowflakeId) {
    if (idGeneratorProvider.get(name).isEmpty()) {
        idGeneratorProvider.set(name, snowflakeId);
    }

    String beanName = name + "SnowflakeId";
    applicationContext.getBeanFactory().registerSingleton(beanName, snowflakeId);
}
//构建SnowFlakeId方法
private SnowflakeId createIdGen(SnowflakeIdProperties.IdDefinition idDefinition,
                                ClockBackwardsSynchronizer clockBackwardsSynchronizer) {
    long epoch = getEpoch(idDefinition);
    int machineBit = MoreObjects.firstNonNull(idDefinition.getMachineBit(), machineProperties.getMachineBit());
    String namespace = Namespaces.firstNotBlank(idDefinition.getNamespace(), cosIdProperties.getNamespace());
// 分配machineId
    int machineId = machineIdDistributor.distribute(namespace, machineBit, instanceId,
machineProperties.getSafeGuardDuration()).getMachineId();
    //根据配置创建不同的雪花算法实例
    SnowflakeId snowflakeId;
    if (SnowflakeIdProperties.IdDefinition.TimestampUnit.SECOND.equals(idDefinition.getTimestampUnit())) {
        snowflakeId = new SecondSnowflakeId(epoch, idDefinition.getTimestampBit(), machineBit, idDefinition.getSequenceBit,
machineId, idDefinition.getSequenceResetThreshold());
    } else {
        snowflakeId =
            new MillisecondSnowflakeId(epoch, idDefinition.getTimestampBit(), machineBit, idDefinition.getSequenceBit(),
machineId, idDefinition.getSequenceResetThreshold());
    }
    if (idDefinition.isClockSync()) {
        snowflakeId = new ClockSyncSnowflakeId(snowflakeId, clockBackwardsSynchronizer);
    }
    IdConverterDefinition converterDefinition = idDefinition.getConverter();
    final ZoneId zoneId = ZoneId.of(snowflakeIdProperties.getZoneId());
    return new SnowflakeIdConverterDecorator(snowflakeId, converterDefinition, zoneId, idDefinition.isFriendly()).decorate
}

```

这段方法有个重点需要关注的地方。

1、createIdGen方法构建雪花算法实例时，会根据配置信息选择创建SecondSnowflakeId还是MillisecondSnowflakeId。这两个具体实例的区别是他们前时间的单位不同。一个是获取秒，一个是获取毫秒。

这个区别会涉及到CosId对于雪花算法时钟回拨问题的处理。时钟回拨问题就是雪花算法的第一个部分时间戳可能面临的一种问题。因为计算机中记录会产生波动的。有可能下一刻产生的时间反而比上一刻的时间更早，这就是时钟回拨问题。这种回拨的时钟很显然就有可能造成时钟回拨的问题。

因此雪花算法通常都需要对时钟回拨进行处理。如果在要生成ID时，发现当前时间比上一次生成的时间还早，那就要休眠一段时间，直到时间往后延后重新生成ID。

2、CosID中，实际生成雪花算法的方法在AbstractSnowflakeId中

```
//me.ahoo.cosid.snowflake.AbstractSnowflakeId
@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 //注入机器位
        | sequence;
}
```

从这里可以看到，这个machine就是作为雪花算法的工作进程位使用的。

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。

```
//me.ahoo.cosid.spring.boot.starter.machine.CosIdMachineAutoConfiguration
@Bean
@ConditionalOnMissingBean({MachineId.class})
public MachineId machineId(MachineIdDistributor machineIdDistributor, InstanceId instanceId) {
    int machineId = machineIdDistributor.distribute(this.cosIdProperties.getNamespace(), this.machineProperties.getMachine
instanceId, this.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。

```
@AutoConfiguration
@ConditionalOnCosIdEnabled
@ConditionalOnCosIdMachineEnabled
@ConditionalOnClass({JdbcMachineIdDistributor.class})
@ConditionalOnProperty(
    value = {"cosid.machine.distributor.type"},
    havingValue = "jdbc"
)
public class CosIdJdbcMachineIdDistributorAutoConfiguration {
    public 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 = {"cosid.machine.distributor.type"},
    matchIfMissing = true,
    havingValue = "manual"
)
public ManualMachineIdDistributor machineIdDistributor(MachineStateStorage localMachineState, ClockBackwardsSynchronizer clockBackwardsSynchronizer) {
    MachineProperties.Manual manual = this.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，其实也可以参照这种方式，往里面注入一个MachineIdDistributor的实现类即可。

当然，说起来简单，但是，具体实现时还是会有一些小问题的。如果你真有这样的想法，我非常鼓励你自己动手试试。相信我。这种成熟的开源框架上的任何项目都更有锻炼价值。

### 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cosic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。他并不需要什么实际的意义。

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机制往下看把。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。

```
//me.ahoo.cosid.spring.boot.starter.machine.CosIdMachineAutoConfiguration
@Bean
@ConditionalOnMissingBean
public InstanceId instanceId(HostAddressSupplier hostAddressSupplier) {
    boolean stable = Boolean.TRUE.equals(this.machineProperties.getStable());
    if (!Strings.isNullOrEmpty(this.machineProperties.getInstanceId())) {
        return InstanceId.of(this.machineProperties.getInstanceId(), stable);
    } else {
        int port = ProcessId.CURRENT.getProcessId();
        if (Objects.nonNull(this.machineProperties.getPort()) && this.machineProperties.getPort() > 0) {
            port = this.machineProperties.getPort();
        }

        return InstanceId.of(hostAddressSupplier.getHostAddress(), port, stable);
    }
}
```

这个InstanceId是用来区分不同的服务实例的。那怎么区分呢？

首先读取machineProperties的instanceId。这个是由应用自己配的。如果应用有这个功夫单独配置instanceId，那就不用自动生成machineID了，1个instanceId，大概率是不会配的。

然后接下来就是从IP+port的方式进行区分。

接下来这个stable参数，实际上用来保持IP稳定的。因为cosId考虑到machineid还是可以回收利用的，比如machineid为1的进程，如果下线了，而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。但是如果1是稳定的，后面的进程就不能再用1这个进程号了。

**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
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!");
    MachineState localState = this.machineStateStorage.get(namespace, instanceId);
    if (!MachineState.NOT_FOUND.equals(localState)) {
        this.clockBackwardsSynchronizer.syncUninterruptibly(localState.getLastTimeStamp());
        return localState;
    } else {
        localState = this.distributeRemote(namespace, machineBit, instanceId, safeGuardDuration);
        if (ClockBackwardsSynchronizer.getBackwardsTimeStamp(localState.getLastTimeStamp()) > 0L) {
            this.clockBackwardsSynchronizer.syncUninterruptibly(localState.getLastTimeStamp());
            localState = MachineState.of(localState.getMachineId(), System.currentTimeMillis());
        }

        this.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.machir 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。这里可以证明，stable稳定的服务，就会占用稳定的machir算应用停了，文件里还记着呢。

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：

```
//me.ahoo.cosid.jdbc.JdbcMachineIdDistributor
@Override
protected MachineState distributeRemote(String namespace, int machineBit, InstanceId instanceId, Duration safeGuardDuratio
    if (log.isInfoEnabled()) {
        log.info("Distribute Remote instanceId:[{}] - machineBit:[{}] @ namespace:[{}].", instanceId, machineBit, namespac
    }
    try (Connection connection = dataSource.getConnection()) {
//本地发
        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。

## 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.guard 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。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。

# 3、Segment数据段模式

## 1、Segment模式基础使用

雪花算法生成的主键ID属于趋势递增，但并不连续。Segment号段模式主要是用来生成一系列连续增长的分布式主键ID。先来看看CosID怎么生成连续后再来分析里面的门道。

CosID使用segment号段非常方便。应用中依然只要从Spring容器里获取IdGeneratorProvider实例，然后通过这个实例获取ID即可。唯一需要修改的信息。

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，这里直接修改配置，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://192.168.65.212:3306/test?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root

cosid.namespace=cosid-example
cosid.enabled=true
#关闭雪花算法功能
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=segment
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2
#cosid.segment.chain.safe-distance=10
#步数，每个segment里的ID数量，默认10
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());
        }
    }
}
```




这次就会拿到从1到100的ID。

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。

对象

 cosid@test (192.168.65....)



name	last_max_id	last_fetch_time
 varchar(100)	 bigint	 bigint
cosid-example.__share__	100	1720675879

从这个数据就能看到，cosid表中的name字段就是表示一个命名空间。当应用来申请ID时，cosid框架会把对应命名空间的一段ID一起分配给这个应用 last\_max\_id就是记录上一次分配后的最大ID。应用拿到这一批ID后，就可以自由分配。在全部使用完之前，不需要再次向cosid框架申请新的ID段，为了与主键生成服务的交互频率。这种模式就是典型的segment模式。

## 2、Segment模式的优化方案

segment模式其实并不复杂，但是要用好却并不容易。

segment模式的基本思想就是很多应用从一个统一的第三方服务中获取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。段ID分发完了，再去第三方服务中获取下一段ID。

以最常用的数据库为例，一个典型的segment服务，可以用这样一张表来设计。

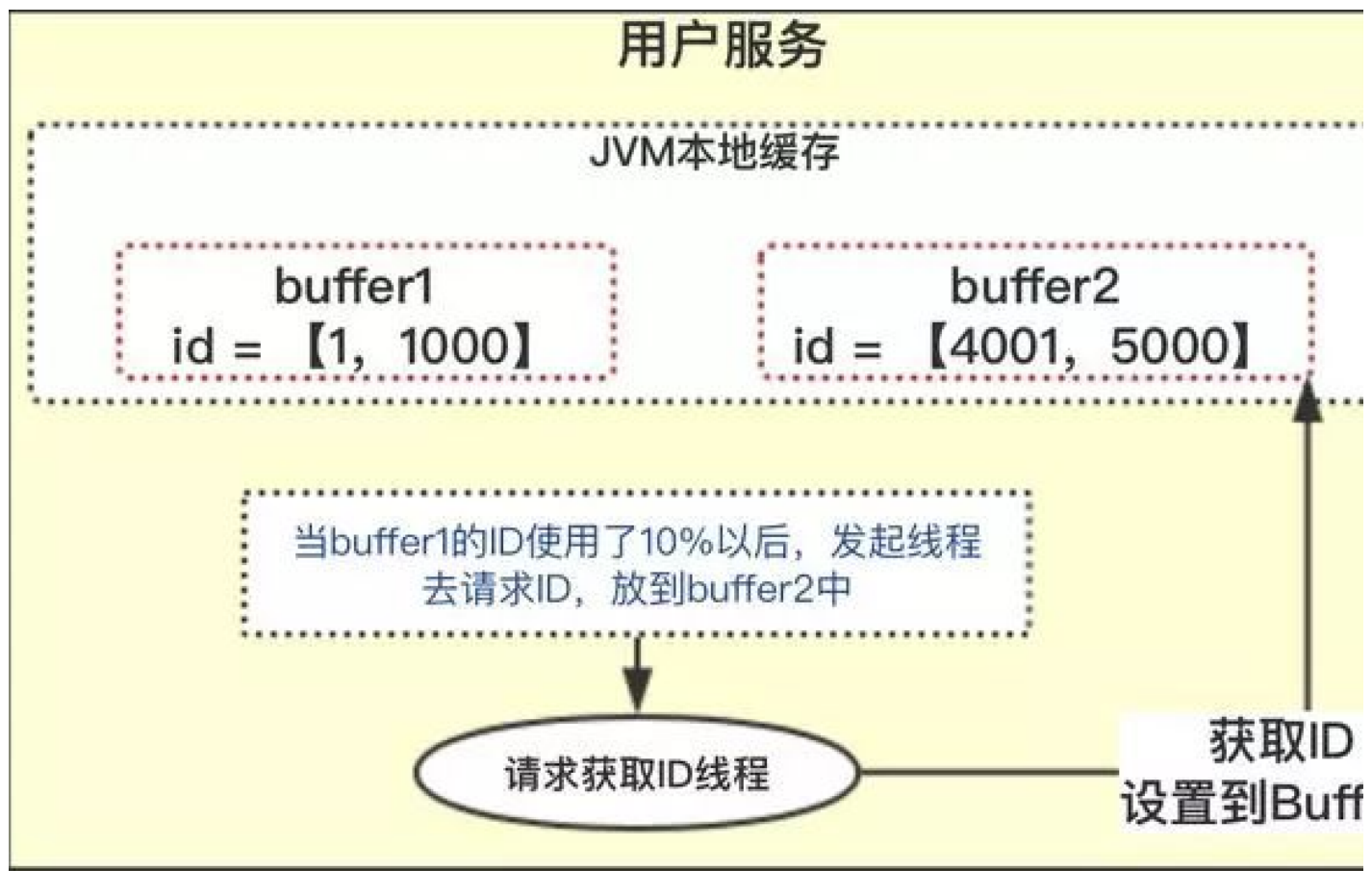
biz_tag	max_id	step	desc	update_time
user_tag	0	1000	用户ID生成规则	
order_tag	0	2000	订单ID生成规则	

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表示每个segment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

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





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用完了，应用就直接从Buffer2中分配ID。然后等Buffer2用到10%，再同样反过来，申请一段新的ID放到Buffer1里。通过双Buffer的交替使用，保证在请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。而其实很多互联网主流的分布式主键生成框架也都是用的这样一个思路，比如百度的Uid。

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。

但是这种方案面向复杂业务时，其实也是有很多不足的。CosID框架就主要针对下面两个问题，对这种方案进行了改进。

1、强依赖于DB。虽然DB是几乎所有项目的标配。但是，DB是用来存储重要的业务数据的，将主键生成这样一个边缘服务强行依赖于DB是否合理呢？

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、延长本地缓存。不管你用哪种中间件来充当号段分配器，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？

### 3、理解SegmentChain模式

之前已经演示了基础的Segment模式的使用案例。CosID的Segment模式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。之前在申请新号段的过程中，服务是短暂不可用的。



为什么这个last\_max\_id更新成了300。就是因为这次申请了两个Segment段，每个segment段的ID长度是100。这两个参数也可以通过配置文件进行

```
#安全距离，segment缓存数量 默认2
cosid.segment.chain.safe-distance=10
#步数，每个segment里的ID数量。默认10
cosid.segment.share.step=100
```

其中这个安全距离就可以简单理解为SegmentChain中Segment的个数。CosID会尽量保证SegmentChain能够保持这个安全距离。

## 4、Segment机制源码解析

CosID框架到底是怎么实现Segment模式的呢？同样可以从一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());
    }
}
```

也就是说，CosID实现Segment模式的核心，就是通过往Spring的IOC容器当中注入的这个SegmentID实例。

那么接下来就来看看这个实例是怎么创建的。

```
//me.ahoo.cosid.spring.boot.starter.segment.SegmentIdBeanRegistrar
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());
    //构建SegmentID实例。
    SegmentId segmentId;
    if (SegmentIdProperties.Mode.SEGMENT.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();
    return new SegmentIdConverterDecorator(segmentId, converterDefinition).decorate();
}
```

可以看到。在创建SegmentID实例时，会根据配置信息选择创建DefaultSegmentId还是SegmentChainId。其中DefaultSegmentId就是单Segment模式，而SegmentChainId自然就是SegmentChain模式的分发器。

接下来，将这个SegmentID实例注入到Spring的IOC容器当中，同时保存到idGeneratorProvider中。

```
//me.ahoo.cosid.spring.boot.starter.segment.SegmentIdBeanRegistrar
private void registerSegmentId(String name, SegmentId segmentId) {
    if (!idGeneratorProvider.get(name).isPresent()) {
        idGeneratorProvider.set(name, segmentId);
    }

    String beanName = name + "SegmentId";
    applicationContext.getBeanFactory().registerSingleton(beanName, segmentId);
}
```

了解了这个工作机制后，再来看看ID是如何分发的。首先来看单Segment模式的实现方式。这个实现比较简单，就是获取号段之后本地分配，本地分配去重新申请。

```

//me.ahoo.cosid.segment.DefaultSegmentId
public long generate() {
    if (this.maxIdDistributor.getStep() == 1L) {
        GroupedAccessor.setIfNotNever(this.maxIdDistributor.group());
        return this.maxIdDistributor.nextMaxId();
    } else {
        long nextSeq;
        if (this.segment.isAvailable()) {
            nextSeq = this.segment.incrementAndGet();
            if (!this.segment.isOverflow(nextSeq)) {
                return nextSeq;
            }
        }

        synchronized(this) {
            while(true) {
                if (this.segment.isAvailable()) {
                    nextSeq = this.segment.incrementAndGet();
                    if (!this.segment.isOverflow(nextSeq)) {
                        return nextSeq;
                    }
                }

                IdSegment nextIdSegment = this.maxIdDistributor.nextIdSegment(this.idSegmentTtl);
                if (!this.maxIdDistributor.allowReset()) {
                    this.segment.ensureNextIdSegment(nextIdSegment);
                }

                this.segment = nextIdSegment;
            }
        }
    }
}

```

接下来看看SegmentChain模式分发ID的实现方式：

```

//me.ahoo.cosid.segment.SegmentChainId
public long generate() {
    while(true) {
        //找到一个可用的segment，分发ID。
        for(IdSegmentChain currentChain = this.headChain; currentChain != null; currentChain = currentChain.getNext()) {
            if (currentChain.isAvailable()) {
                long nextSeq = currentChain.incrementAndGet();
                if (!currentChain.isOverflow(nextSeq)) {
                    this.forward(currentChain);
                    return nextSeq;
                }
            }
        }
        //找不到，链表空了就添加一个
        try {
            IdSegmentChain preIdSegmentChain = this.headChain;
            if (preIdSegmentChain.trySetNext((preChain) -> {
                return this.generateNext(preChain, this.safeDistance);
            })) {
                IdSegmentChain nextChain = preIdSegmentChain.getNext();
                this.forward(nextChain);
                if (log.isDebugEnabled()) {
                    log.debug("Generate [{}] - headChain.version:[{}->{}].", new Object[]
                    {this.maxIdDistributor.getNamespacedName(), preIdSegmentChain.getVersion(), nextChain.getVersion()});
                }
            }
        } catch (NextIdSegmentExpiredException var4) {
            NextIdSegmentExpiredException nextIdSegmentExpiredException = var4;
            if (log.isWarnEnabled()) {
                log.warn("Generate [{}] - gave up this next IdSegmentChain.", this.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();
    }
}

```

CosID在后台会启动一个线程池PrefetchWorker，异步进行链表扩中。而具体进行链表扩充的方法，就是这个prefetchJob任务。

线程调度的逻辑这里就不多做梳理了，挺多挺复杂的。最终核心的扩充Segment的逻辑是这样的。

```
//me.ahoo.cosid.segment.SegmentChainId#PrefetchJob
public class PrefetchJob implements AffinityJob {

    public void prefetch() {
        long wakeupTimeGap = Clock.SYSTEM.secondTime() - this.lastHungerTime;
        boolean hunger = wakeupTimeGap < 5L;

        //安全距离
        int prePrefetchDistance = this.prefetchDistance;
        if (hunger) {
            this.prefetchDistance = Math.min(Math.multiplyExact(this.prefetchDistance, 2), 100000000);
            if (SegmentChainId.log.isInfoEnabled()) {
                SegmentChainId.log.info("Prefetch [{}] - Hunger, Safety distance expansion.[{}->{}]", new Object[]
{SegmentChainId.this.maxIdDistributor.getNamespacedName(), prePrefetchDistance, this.prefetchDistance});
            }
        } else {
            this.prefetchDistance = Math.max(Math.floorDiv(this.prefetchDistance, 2), SegmentChainId.this.safeDistance);
            if (prePrefetchDistance > this.prefetchDistance && SegmentChainId.log.isInfoEnabled()) {
                SegmentChainId.log.info("Prefetch [{}] - Full, Safety distance shrinks.[{}->{}]", new Object[]
{SegmentChainId.this.maxIdDistributor.getNamespacedName(), prePrefetchDistance, this.prefetchDistance});
            }
        }

        IdSegmentChain availableHeadChain = SegmentChainId.this.headChain;

        while(!availableHeadChain.getIdSegment().isAvailable()) {
            availableHeadChain = availableHeadChain.getNext();
            if (availableHeadChain == null) {
                availableHeadChain = this.tailChain;
                break;
            }
        }

        SegmentChainId.this.forward(availableHeadChain);

        //计算链表当中的Segment数量。
        int headToTailGap = availableHeadChain.gap(this.tailChain, SegmentChainId.this.maxIdDistributor.getStep());
        //计算链表数量与安全距离之间的差距
        int safeGap = SegmentChainId.this.safeDistance - headToTailGap;
        //链表中的segment个数已经不够了，但是不急着想。
        if (safeGap <= 0 && !hunger) {
            if (SegmentChainId.log.isTraceEnabled()) {
                SegmentChainId.log.trace("Prefetch [{}] - safeGap is less than or equal to 0, and is not hungry -
headChain.version:[{}] - tailChain.version:[{}].", new Object[] {SegmentChainId.this.maxIdDistributor.getNamespacedName(),
availableHeadChain.getVersion(), this.tailChain.getVersion()});
            }
        } else {
            //需要添加几个Segment
            int prefetchSegments = hunger ? this.prefetchDistance : safeGap;
            //申请并添加Segment到SegmentChain链表当中。
            this.appendChain(availableHeadChain, prefetchSegments);
        }
    }
}
```

这里核心的hungry模式，其实就是用来保证数据库不可用时，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。

## 5、基于JDBC的ID分发机制实现分析

接下来在实际构建新的segment时，就需要注册一个IdSegmentDistributor接口，来计算新Segment的maxId。这个接口的具体实现，就会交由与各种务集成的扩展组件去完成。例如基于JDBC的ID分发器提供的实现类是JdbcIdSegmentDistributor。他的具体实现是这样的：

```
//me.ahoo.cosid.jdbc.JdbcIdSegmentDistributor
@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.isDebugEnabled()) {
            log.error(sqlException.getMessage(), sqlException);
        }
        throw new CosIdException(sqlException.getMessage(), sqlException);
    }
}
```

这段逻辑，如果你觉得挺麻烦，那么，其实只要看懂下面这两个SQL语句，就知道怎么回事了。

```
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 = ?";
```

## 四、章节总结

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，把分布式ID这样一个不问题详细梳理了一下。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。

其实在分库分表这个小领域，早就有了美团Leaf，百度Uid等等很多成熟的方案在前了。但是依然冒出了CosID这样一个后起之秀。可见深挖需求，融合才是技术的发展之道。而且CosID框架现在也在不断发展。现在也在不断的和更多其他业务场景融合，不断折腾出更大的水花。所以，今天这个章节，家一个学习新框架的思路，同时也是个大家一个新的发展方向。分布式主键这只是一个不起眼的小领域，尚且有这么大的发展空间。那么其他方向呢？

【有道云笔记】五、融会贯通：[详细分析ShardingSphere新接入的CosID主键生成框架.md](https://note.youdao.com/s/caEAnN4P)  
<https://note.youdao.com/s/caEAnN4P>