# 概述

在我们业务数据量不大的时候，单库单表完全可以支撑现有业务，数据再大一点搞个MySQL主从同步读写分离也能对付。

但随着数据日渐增长，主从同步也扛不住了，就需要对数据库进行分库分表，但分库分表后需要有一个唯一ID来标识一条数据，数据库的自增ID显然不能满足需求；特别一点的如订单、优惠券也都需要有唯一ID做标识。此时一个能够生成全局唯一ID的系统是非常必要的。那么这个全局唯一ID就叫分布式ID。

分布式ID需要满足的条件：

1、全局唯一：必须保证ID是全局性唯一的，基本要求

2、高性能：高可用低延时，ID生成响应要快，否则反倒会成为业务瓶颈

3、高可用：100%的可用性是骗人的，但是也要无限接近于100%的可用性

4、好接入：要秉着拿来即用的设计原则，在系统设计和实现上要尽可能的简单

5、趋势递增：最好趋势递增，这个要求就得看具体业务场景了，一般不严格要求

# 使用场景

数据库主键（唯一，且后面的值比前面的大，如果后面的比前面的小则会造成数据移动，产生读取IO）

业务序列号（如发票号，车票号，订单号等）

# 单机ID

单机数据库生成序列号的方法：

MySQL：**AUTO\_INCREMENT**

SQL Server：IDENTITY/SEQUENCE

Oracle：**SEQUENCE**

PgSQL：SEQUENCE

注：优先选择系统提供的系列号生成方式；在特殊情况下可以使用SQL方式生成序列号。

需求：生成订单序列号，并且订单号的格式如下：

YYYYMMDDNNNNNNN，如20150512000003

使用SQL生成序列号：

DECLARE v\_cnt INT;

DECLARE v\_timestr INT;

DECLARE rowcount BIGINT;

SET v\_timestr=DATE\_FORMAT(NOW(),’%Y%m%d’);

SELECT ROUND(RAND()\*100,0)+1 INTO v\_cnt;

SATRT TRANSACTION;

UPDATE order\_seq SET order\_sn=order\_sn+v\_cnt

WHERE timestr=v\_timestr;

IF ROW\_COUNT()=0 THEN

INSERT INTO order\_seq(timestr,order\_sn) VALUES(v\_timestr,v\_cnt);

END IF;

SELECT CONCAT(v\_timestr,LPAD(order\_sn,7,0)) AS order\_sn

FROM order\_seq WHERE timestr=v\_timestr;

COMMIT;

# 分布式ID

## UUID

**优点：**

1、代码实现简单。

2、本机生成，没有性能问题（本地生成无网络消耗）

3、因为是全球唯一的ID，所以迁移数据容易

4、适合大规模数据。当你把数据分片（例如一组客户数据）存在多个数据库时，使用UUID意味着ID在所有数据分片中都是唯一，而不仅仅是当前那个分片所在数据库。这使得跨数据库移动更为安全。

5、在插入数据之前就可以知道PK，这避免了查询 DB开销，并简化了事务逻辑，比如在使用该键作为其它表外键（FK）时，需要预先获得这个PK。

6、UUID 不会泄露数据信息，因此在URL中暴露会更安全。如果一个用户ID是12345678，很容易猜到还有用户12345677和1234569，这构成了攻击因素。

**缺点：**

1、每次生成的**ID是无序的**，无法保证趋势递增

2、UUID的字符串存储，**查询效率慢**（长度过长16 字节128位，36位长度的字符串，存储以及查询对MySQL的性能消耗较大，MySQL官方明确建议主键要尽量越短越好，作为数据库主键 UUID 的无序性会导致数据位置频繁变动，严重影响性能），随机字符串排序比较麻烦。

**3、存储空间大**

4、ID本身无业务含义，不可读

5、碎片化。由于UUID是随机的，它们没有自然顺序，因此不能用于聚集索引（clustering index）。

**应用场景：**

类似生成token令牌的场景

不适用一些要求有趋势递增的ID场景

## MySQL主键自增

这个方案就是利用了MySQL的主键自增auto\_increment，默认每次ID加1。

**优点：**

数字化，id递增

查询效率高

具有一定的业务可读

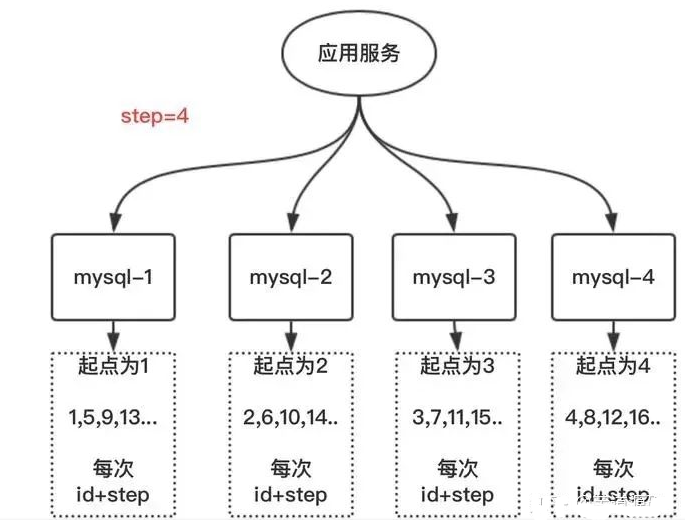
**缺点：**

存在**单点问题**，如果mysql挂了，就没法生成iD了

**数据库压力大，高并发抗不住**

## MySQL多实例主键自增

数据库多主模式（多实例主键自增）：这个方案就是解决mysql的单点问题，在auto\_increment基本上面，设置step步长



每台的初始值分别为1，2，3...N，步长为N（这个案例步长为4）

**优点：**

解决了单点问题

**缺点：**

**一旦把步长定好后，就无法扩容**；而且单个数据库的压力大，数据库自身性能无法满足高并发

**应用场景：**

数据不需要扩容的场景

## 号段模式

号段模式是当下分布式ID生成器的主流实现方式之一，号段模式可以理解为从数据库批量的获取自增ID，每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个ID，具体的业务服务将本号段，生成1~1000的自增ID并加载到内存。表结构如下：

CREATE TABLE id\_generator (

id int(10) NOT NULL,

max\_id bigint(20) NOT NULL COMMENT '当前最大id',

step int(20) NOT NULL COMMENT '号段的布长',

biz\_type int(20) NOT NULL COMMENT '业务类型',

version int(20) NOT NULL COMMENT '版本号',

PRIMARY KEY (`id`)

)

biz\_type ：代表不同业务类型

max\_id ：当前最大的可用id

step ：代表号段的长度

version ：是一个乐观锁，每次都更新version，保证并发时数据的正确性

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | biz\_type | max\_id | step | version |
| 1 | 101 | 1000 | 2000 | 0 |

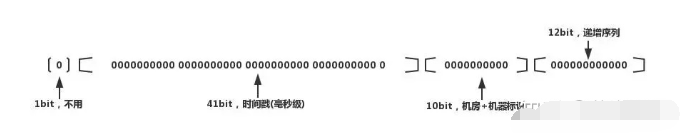
等这批号段ID用完，再次向数据库申请新号段，对max\_id字段做一次update操作，update max\_id= max\_id + step，update成功则说明新号段获取成功，新的号段范围是(max\_id ,max\_id +step]。

update id\_generator set max\_id = #{max\_id+step}, version = version + 1 where version = # {version} and biz\_type = XXX

由于多业务端可能同时操作，所以采用版本号version乐观锁方式更新，这种分布式ID生成方式不强依赖于数据库，不会频繁的访问数据库，对数据库的压力小很多。

## 雪花snowflake算法

雪花算法生成64位的二进制正整数，然后转换成10进制的数。64位二进制数由如下部分组成：



1位标识符：始终是0

41位时间戳：41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截 )得到的值，这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的

10位机器标识码：可以部署在1024个节点，如果机器分机房（IDC）部署，这10位可以由 5位机房ID + 5位机器ID 组成

12位序列：毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号

**优点：**

此方案每秒能够产生409.6万个ID，**性能快**

时间戳在高位，自增序列在低位，整个ID是趋势递增的，按照时间有序递增

灵活度高，可以根据业务需求，调整bit位的划分，满足不同的需求

**缺点：**

**依赖机器的时钟，如果服务器时钟回拨，会导致重复ID生成**

在分布式场景中，服务器时钟回拨会经常遇到，一般存在10ms之间的回拨；有人会说这点10ms，很短可以不考虑吧。但此算法就是建立在毫秒级别的生成方案，一旦回拨，就很有可能存在重复ID。

## Redis生成方案

利用redis的incr原子性操作自增，一般算法为：

年份 + 当天距当年第多少天 + 天数 + 小时 + redis自增

**优点：**

有序递增，可读性强

**缺点：**

**占用带宽，每次要向redis进行请求**

用redis实现需要注意一点，要考虑到redis持久化的问题。redis有两种持久化方式RDB和AOF：

RDB会定时打一个快照进行持久化，假如连续自增但redis没及时持久化，而这会Redis挂掉了，重启Redis后会出现ID重复的情况。

AOF会对每条写命令进行持久化，即使Redis挂掉了也不会出现ID重复的情况，但由于incr命令的特殊性，会导致Redis重启恢复的数据时间过长。

整体测试了这个性能如下：

需求：

1、同时10万个请求获取ID1、并发执行完耗时：9s左右

2、单任务平均耗时：74ms

3、单线程最小耗时：不到1ms

4、单线程最大耗时：4.1s

性能还可以，如果对性能要求不是太高的话，这个方案基本符合老顾的要求。

但不完全符合业务老顾希望id从 1 开始趋势递增。（当然算法可以调整为 就一个 redis自增，不需要什么年份，多少天等）。

# 改造方案

## 改造数据库主键自增

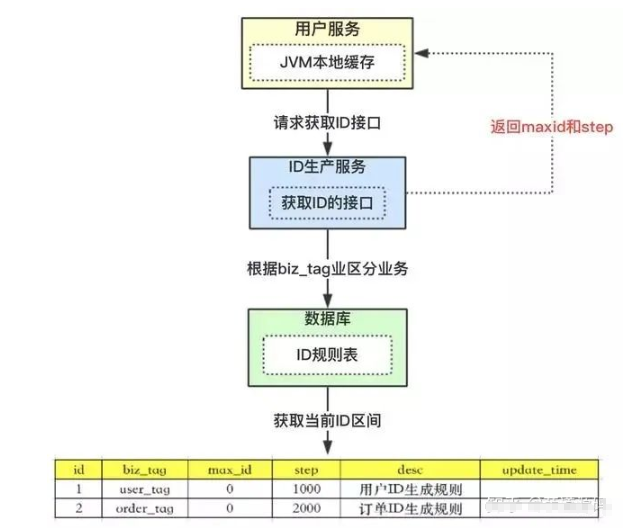
上述利用数据库的自增主键的特性，可以实现分布式ID；这个ID比较简短明了，适合做userId，正好符合如何永不迁移数据和避免热点? 根据服务器指标分配数据量(揭秘篇)文章中的ID的需求。但这个方案有严重的问题：

1、一旦步长定下来，不容易扩容

2、数据库压力大

先看数据库压力大，为什么压力大？是因为我们每次获取ID的时候，都要去数据库请求一次。那我们可以不可以不要每次去取？

思路：我们可以请求数据库得到ID的时候，可设计成获得的ID是一个ID区间段。



**有张ID规则表：**

1、id表示为主键，无业务含义。

2、biz\_tag为了表示业务，因为整体系统中会有很多业务需要生成ID，这样可以共用一张表维护

3、max\_id表示现在整体系统中已经分配的最大ID

4、desc描述

5、update\_time表示每次取的ID时间

**整体流程：**

1、【用户服务】在注册一个用户时，需要一个用户ID；会请求【生成ID服务(是独立的应用)】的接口

2、【生成ID服务】会去查询数据库，找到user\_tag的id，现在的max\_id为0，step=1000

3、【生成ID服务】把max\_id和step返回给【用户服务】；并且把max\_id更新为max\_id = max\_id + step，即更新为1000

4、【用户服务】获得max\_id=0，step=1000；

5、 这个用户服务可以用ID=【max\_id + 1，max\_id+step】区间的ID，即为【1，1000】

6、【用户服务】会把这个区间保存到jvm中

7、【用户服务】需要用到ID的时候，在区间【1，1000】中依次获取id，可采用AtomicLong中的getAndIncrement方法。

8、如果把区间的值用完了，再去请求【生产ID服务】接口，获取到max\_id为1000，即可以用【max\_id + 1，max\_id+step】区间的ID，即为【1001，2000】

这个方案就非常完美的解决了数据库自增的问题，而且可以自行定义max\_id的起点，和step步长，非常方便扩容。

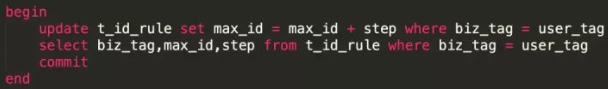
而且也解决了数据库压力的问题，因为在一段区间内，是在jvm内存中获取的，而不需要每次请求数据库。即使数据库宕机了，系统也不受影响，ID还能维持一段时间。

## 竞争问题

以上方案中，如果是多个用户服务，同时获取ID，同时去请求【ID服务】，在获取max\_id的时候会存在并发问题。

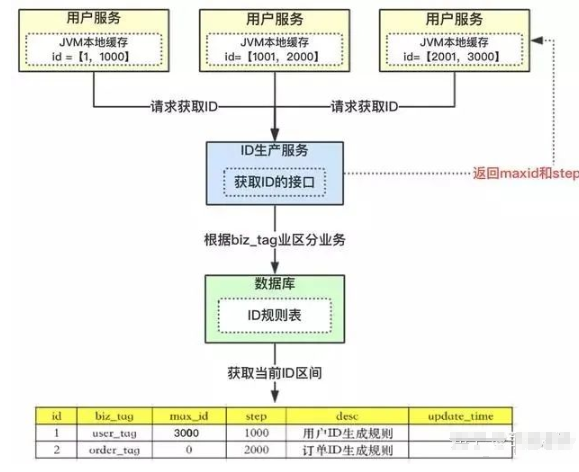
如用户服务A，取到的max\_id=1000 ;用户服务B取到的也是max\_id=1000，那就出现了问题，Id重复了。那怎么解决？

其实方案很多，加分布式锁，保证同一时刻只有一个用户服务获取max\_id。当然也可以用数据库自身的锁去解决。



利用事务方式加行锁，上面的语句，在没有执行完之前，是不允许第二个用户服务请求过来的，第二个请求只能阻塞。

## 突发阻塞问题



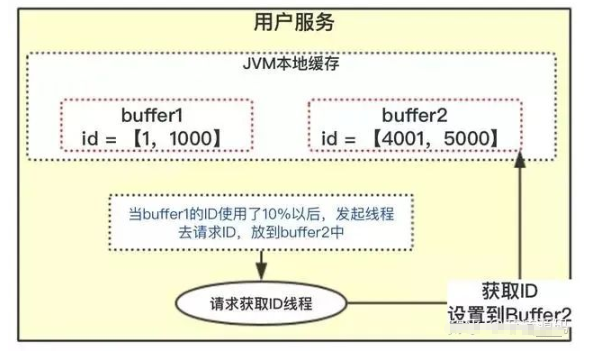
上图中，多个用户服务获取到了各自的ID区间，在高并发场景下，ID用的很快，如果3个用户服务在某一时刻都用完了，同时去请求【ID服务】。因为上面提到的竞争问题，所有只有一个用户服务去操作数据库，其他二个会被阻塞。

有这么巧吗？同时ID用完。我们这里举的是3个用户服务，感觉概率不大；如果是100个用户服务呢？概率是不是一下子大了。

出现的现象就是一会儿突然系统耗时变长，一会儿好了，就是这个原因导致的，怎么去解决？

## 双buffer方案

在一般的系统设计中，双buffer会经常看到，怎么去解决上面的问题也可以采用双buffer方案。



在设计的时候，采用双buffer方案，上图的流程：

1、当前获取ID在buffer1中，每次获取ID在buffer1中获取

2、当buffer1中的Id已经使用到了100，也就是达到区间的10%

3、达到了10%，先判断buffer2中有没有去获取过，如果没有就立即发起请求获取ID线程，此线程把获取到的ID，设置到buffer2中。

4、如果buffer1用完了，会自动切换到buffer2

5、buffer2用到10%了，也会启动线程再次获取，设置到buffer1中

6、依次往返

双buffer的方案，这样就达到了业务场景用的ID，都是在jvm内存中获得的，从此不需要到数据库中获取了。允许数据库宕机时间更长了。

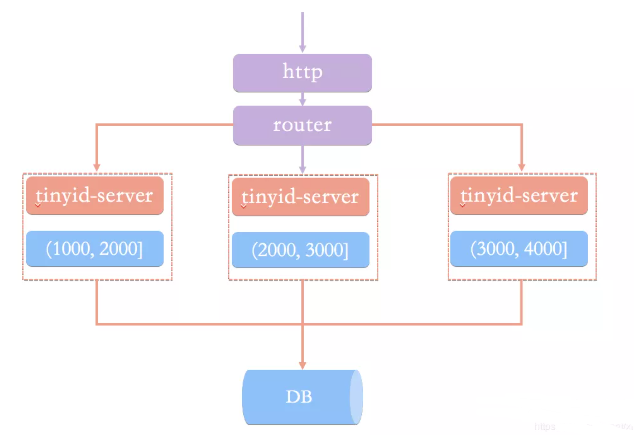
因为会有一个线程，会观察什么时候去自动获取。两个buffer之间自行切换使用。就解决了突发阻塞的问题。

# 互联网落地方案

## 滴滴出品（TinyID）

Tinyid由滴滴开发，Github地址：https://github.com/didi/tinyid。

Tinyid是基于号段模式原理实现的与Leaf如出一辙，每个服务获取一个号段（1000,2000]、（2000,3000]、（3000,4000]



## 百度（Uidgenerator）

uid-generator是由百度技术部开发，项目GitHub地址 https://github.com/baidu/uid-generator

uid-generator是基于Snowflake算法实现的，与原始的snowflake算法不同在于，uid-generator支持自定义时间戳、工作机器ID和 序列号 等各部分的位数，而且uid-generator中采用用户自定义workId的生成策略。

uid-generator需要与数据库配合使用，需要新增一个WORKER\_NODE表。当应用启动时会向数据库表中去插入一条数据，插入成功后返回的自增ID就是该机器的workId数据由host，port组成。

对于uid-generator ID组成结构：

workId，占用了22个bit位，时间占用了28个bit位，序列化占用了13个bit位，需要注意的是，和原始的snowflake不太一样，时间的单位是秒，而不是毫秒，workId也不一样，而且同一应用每次重启就会消费一个workId。

## 美团Leaf方案

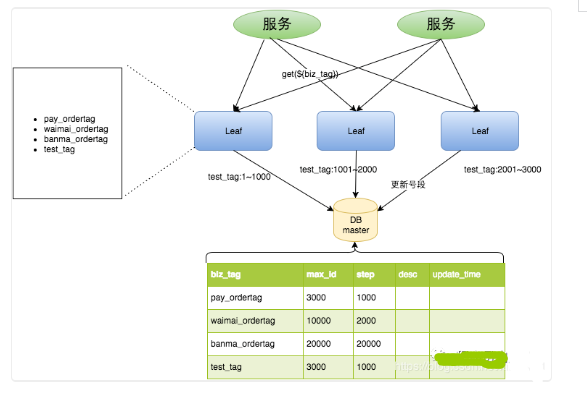
Leaf是美团推出的一个分布式ID生成服务，Leaf的优势：高可靠、低延迟、全局唯一等特点。

目前主流的分布式ID生成方式，大致都是基于数据库号段模式和雪花算法（snowflake），而美团（Leaf）刚好同时兼具了这两种方式，可以根据不同业务场景灵活切换。

### Leaf-segment号段模式

Leaf-segment号段模式是对直接用数据库自增ID充当分布式ID的一种优化，减少对数据库的频率操作。相当于从数据库批量的获取自增ID，每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个ID，业务服务将号段在本地生成1~1000的自增ID并加载到内存。

大致的流程入下图所示：



号段耗尽之后再去数据库获取新的号段，可以大大的减轻数据库的压力。对max\_id字段做一次update操作，update max\_id= max\_id + step，update成功则说明新号段获取成功，新的号段范围是(max\_id ,max\_id +step]。

通常在用号段模式的时候，取号段的时机是在前一个号段消耗完的时候进行的，可Leaf方案中才取了一个ID，数据库中却已经更新了max\_id，也就是说leaf已经多获取了一个号段，这是什么鬼操作？

**Leaf为啥要这么设计呢？**

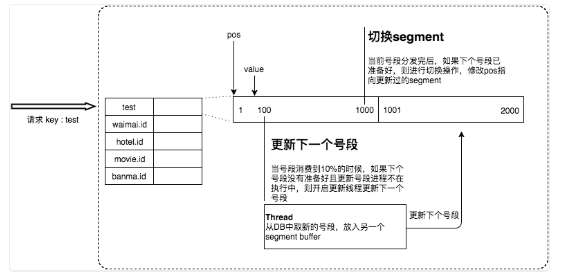
Leaf 希望能在DB中取号段的过程中做到无阻塞！

当号段耗尽时再去DB中取下一个号段，如果此时网络发生抖动，或者DB发生慢查询，业务系统拿不到号段，就会导致整个系统的响应时间变慢，对流量巨大的业务，这是不可容忍的。

所以Leaf在当前号段消费到某个点时，就异步的把下一个号段加载到内存中。而不需要等到号段用尽的时候才去更新号段。这样做很大程度上的降低了系统的风险。

**Leaf采用双buffer的方式**，它的服务内部有两个号段缓存区segment。当前号段已消耗10%时，还没能拿到下一个号段，则会另启一个更新线程去更新下一个号段。

简而言之就是Leaf保证了总是会多缓存两个号段，即便哪一时刻数据库挂了，也会保证发号服务可以正常工作一段时间。



通常推荐号段（segment）长度设置为服务高峰期发号QPS的600倍（10分钟），这样即使DB宕机，Leaf仍能持续发号10-20分钟不受影响。

**优点：**

Leaf服务可以很方便的线性扩展，性能完全能够支撑大多数业务场景。

容灾性高：Leaf服务内部有号段缓存，即使DB宕机，短时间内Leaf仍能正常对外提供服务。

**缺点：**

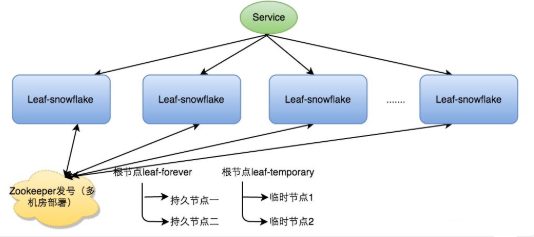
ID号码不够随机，能够泄露发号数量的信息，不太安全。

DB宕机会造成整个系统不可用（用到数据库的都有可能）。

### Leaf-snowflake

Leaf-snowflake基本上就是沿用了snowflake的设计，ID组成结构：正数位（占1比特）+ 时间戳（占41比特）+ 机器ID（占5比特）+ 机房ID（占5比特）+ 自增值（占12比特），总共64比特组成的一个Long类型。

Leaf-snowflake不同于原始snowflake算法地方，主要是在workId的生成上，Leaf-snowflake依靠Zookeeper生成workId，也就是上边的机器ID（占5比特）+ 机房ID（占5比特）。Leaf中workId是基于ZooKeeper的顺序Id来生成的，每个应用在使用Leaf-snowflake时，启动时都会都在Zookeeper中生成一个顺序Id，相当于一台机器对应一个顺序节点，也就是一个workId。



Leaf-snowflake启动服务的过程大致如下：

1、启动Leaf-snowflake服务，连接Zookeeper，在leaf\_forever父节点下检查自己是否已经注册过（是否有该顺序子节点）。

2、如果有注册过直接取回自己的workerID（zk顺序节点生成的int类型ID号），启动服务。

3、如果没有注册过，就在该父节点下面创建一个持久顺序节点，创建成功后取回顺序号当做自己的workerID号，启动服务。

但Leaf-snowflake对Zookeeper是一种弱依赖关系，除了每次会去ZK拿数据以外，也会在本机文件系统上缓存一个workerID文件。一旦ZooKeeper出现问题，恰好机器出现故障需重启时，依然能够保证服务正常启动。

启动Leaf-snowflake模式也比较简单，起动本地ZooKeeper，修改一下项目中的leaf.properties文件，关闭leaf.segment模式，启用leaf.snowflake模式即可。

leaf.segment.enable=false

#leaf.jdbc.url=jdbc:mysql://127.0.0.1:3306/xin-master?useUnicode=true&characterEncoding=utf8

#leaf.jdbc.username=junkang

#leaf.jdbc.password=junkang

leaf.snowflake.enable=true

leaf.snowflake.zk.address=127.0.0.1

leaf.snowflake.port=2181

/\*\*

\* 雪花算法模式

\* @param key

\* @return

\*/

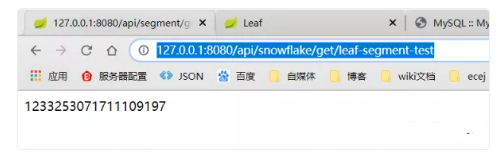
@RequestMapping(value = "/api/snowflake/get/{key}")

public String getSnowflakeId(@PathVariable("key") String key) {

return get(key, snowflakeService.getId(key));

}

测试一下，访问：http://127.0.0.1:8080/api/snowflake/get/leaf-segment-test



**优点：**

ID号码是趋势递增的8byte的64位数字，满足上述数据库存储的主键要求。

**缺点：**

依赖ZooKeeper，存在服务不可用风险