# 唯一序列号

## 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）

## 生成

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：**

代码实现简单。

本机生成，没有性能问题

因为是全球唯一的ID，所以迁移数据容易

**缺点：**

每次生成的ID是无序的，无法保证趋势递增

UUID的字符串存储，查询效率慢

存储空间大

ID本身无业务含义，不可读

**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

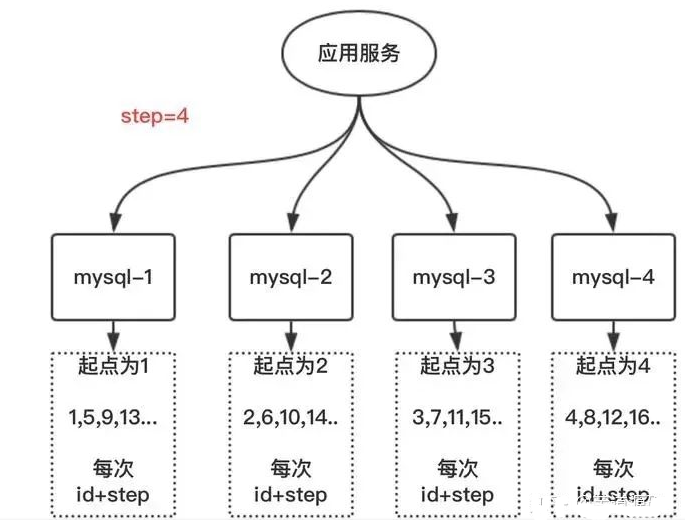
**缺点：**

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

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：**

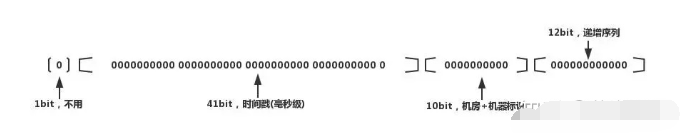
一旦把步长定好后，就无法扩容；而且单个数据库的压力大，数据库自身性能无法满足高并发

**应用场景：**

数据不需要扩容的场景

### 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；小伙伴们就说这点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

整体测试了这个性能如下：

需求：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）。

## 互联网落地方案

### 改造数据库主键自增

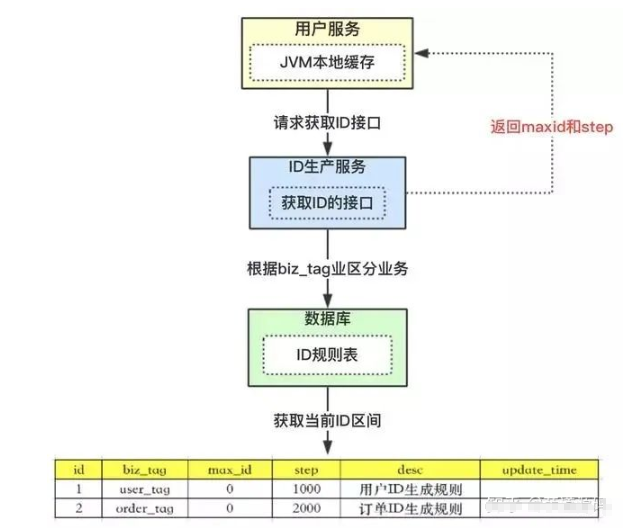
上述利用数据库的自增主键的特性，可以实现分布式ID；这个ID比较简短明了，适合做userId，正好符合如何永不迁移数据和避免热点? 根据服务器指标分配数据量(揭秘篇)文章中的ID的需求。但这个方案有严重的问题：

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。

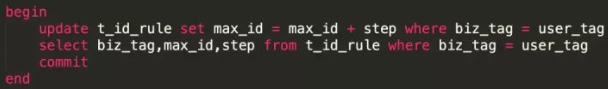
而且也解决了数据库压力的问题，因为在一段区间内，是在jvm内存中获取的，而不需要每次请求数据库。即使数据库宕机了，系统也不受影响，ID还能维持一段时间。

### 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。

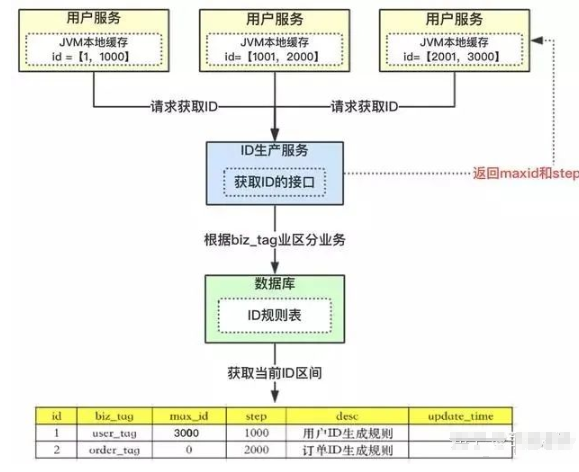
如用户服务A，取到的max\_id=1000 ;用户服务B取到的也是max\_id=1000，那就出现了问题，Id重复了。那怎么解决？

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



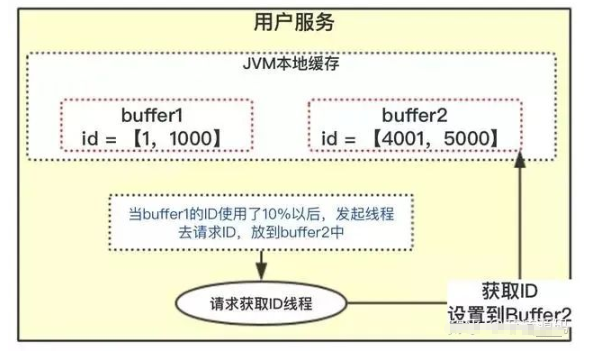
上图中，多个用户服务获取到了各自的ID区间，在高并发场景下，ID用的很快，如果3个用户服务在某一时刻都用完了，同时去请求【ID服务】。因为上面提到的竞争问题，所有只有一个用户服务去操作数据库，其他二个会被阻塞。

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。