# [分布式系统唯一ID生成方案汇总](http://www.cnblogs.com/haoxinyue/p/5208136.html)

系统唯一ID是我们在设计一个系统的时候常常会遇见的问题，也常常为这个问题而纠结。生成ID的方法有很多，适应不同的场景、需求以及性能要求。所以有些比较复杂的系统会有多个ID生成的策略。下面就介绍一些常见的ID生成策略。

**1. 数据库自增长序列或字段**

最常见的方式。利用数据库，全数据库唯一。

优点：

1）简单，代码方便，性能可以接受。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。

2）在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

3）在性能达不到要求的情况下，比较难于扩展。

4）如果遇见多个系统需要合并或者涉及到数据迁移会相当痛苦。

5）分表分库的时候会有麻烦。

优化方案：

1）针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

**2. UUID**

常见的方式。可以利用数据库也可以利用程序生成，一般来说全球唯一。

优点：

1）简单，代码方便。

2）生成ID性能非常好，基本不会有性能问题。

3）全球唯一，在遇见数据迁移，系统数据合并，或者数据库变更等情况下，可以从容应对。

缺点：

1）没有排序，无法保证趋势递增。

2）UUID往往是使用字符串存储，查询的效率比较低。

3）存储空间比较大，如果是海量数据库，就需要考虑存储量的问题。

4）传输数据量大

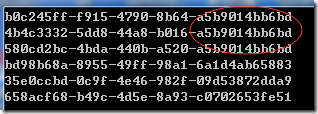
5）不可读。

**3. UUID的变种**

1）为了解决UUID不可读，可以使用UUID to Int64的方法。及

2）为了解决UUID无序的问题，NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

用上面的算法测试一下，得到如下的结果：作为比较，前面3个是使用COMB算法得出的结果，最后12个字符串是时间序（统一毫秒生成的3个UUID），过段时间如果再次生成，则12个字符串会比图示的要大。后面3个是直接生成的GUID。

[](http://images2015.cnblogs.com/blog/15700/201602/15700-20160227213048174-1443183768.png)

**4. Redis生成ID**

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A：1,6,11,16,21

B：2,7,12,17,22

C：3,8,13,18,23

D：4,9,14,19,24

E：5,10,15,20,25

这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以方式单点故障的问题。

另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。

2）需要编码和配置的工作量比较大。

**5. Twitter的snowflake算法**

snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），最后还有一个符号位，永远是0。具体实现的代码可以参看<https://github.com/twitter/snowflake>。

**Snowflake。**  
1 41位的时间序列（精确到毫秒，41位的长度可以使用69年）  
2 10位的机器标识（10位的长度最多支持部署1024个节点）   
3 12位的计数顺序号（12位的计数顺序号支持每个节点每毫秒产生4096个ID序号） 最高位是符号位，始终为0。  
优点：高性能，低延迟；独立的应用；按时间有序。 缺点：需要独立的开发和部署。  
原理如下：

测试代码如下：

snowflake算法可以根据自身项目的需要进行一定的修改。比如估算未来的数据中心个数，每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的bit数。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）ID按照时间在单机上是递增的。

缺点：

1）在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，也许有时候也会出现不是全局递增的情况。

**6. 利用zookeeper生成唯一ID**

zookeeper主要通过其znode数据版本来生成序列号，可以生成32位和64位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用zookeeper来生成唯一ID。主要是由于需要依赖zookeeper，并且是多步调用API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想。

**7. MongoDB的ObjectId**

MongoDB的ObjectId和snowflake算法类似。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。使其在分片环境中要容易生成得多。

其格式如下：



前4 个字节是从标准纪元开始的时间戳，单位为秒。时间戳，与随后的5 个字节组合起来，提供了秒级别的唯一性。由于时间戳在前，这意味着ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率。这4 个字节也隐含了文档创建的时间。绝大多数客户端类库都会公开一个方法从ObjectId 获取这个信息。   
接下来的3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的ObjectId，不产生冲突。   
为了确保在同一台机器上并发的多个进程产生的ObjectId 是唯一的，接下来的两字节来自产生ObjectId 的进程标识符（PID）。   
前9 字节保证了同一秒钟不同机器不同进程产生的ObjectId 是唯一的。后3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有2563（16 777 216）个不同的ObjectId。

实现的源码可以到MongoDB官方网站下载。

# 数据库分库分表(sharding)系列(二) 全局主键生成策略

本文将主要介绍一些常见的全局主键生成策略，然后重点介绍flickr使用的一种非常优秀的全局主键生成方案。关于分库分表(sharding)的拆分策略和实施细则，

**第一部分：一些常见的主键生成策略**

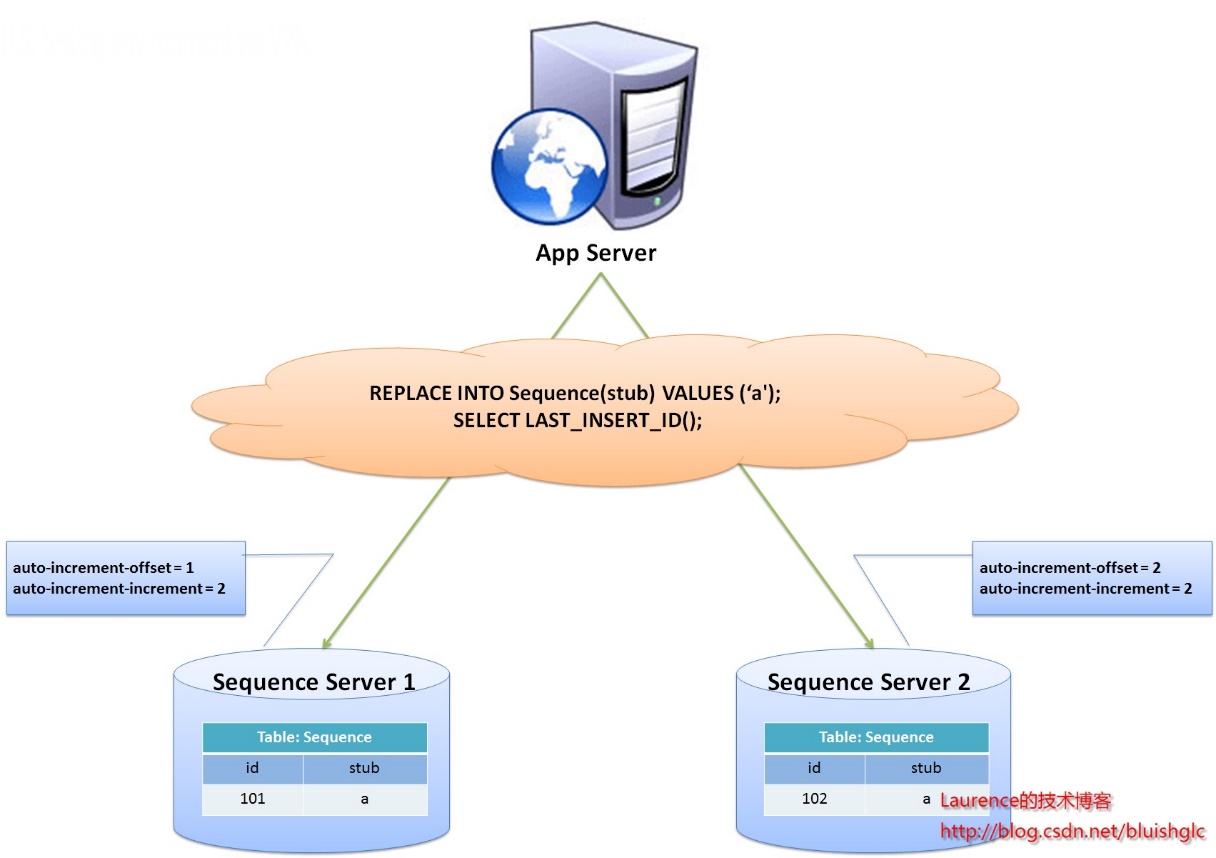
一旦数据库被切分到多个物理结点上，我们将不能再依赖数据库自身的主键生成机制。一方面，某个分区数据库自生成的ID无法保证在全局上是唯一的；另一方面，应用程序在插入数据之前需要先获得ID,以便进行SQL路由。目前几种可行的主键生成策略有：  
1. UUID：使用UUID作主键是最简单的方案，但是缺点也是非常明显的。由于UUID非常的长，除占用大量存储空间外，最主要的问题是在索引上，在建立索引和基于索引进行查询时都存在性能问题。  
2. 结合数据库维护一个Sequence表：此方案的思路也很简单，在数据库中建立一个Sequence表，表的结构类似于：

1. **CREATE** **TABLE** `**SEQUENCE**` (
2. `tablename` **varchar**(30) NOT NULL,
3. `nextid` **bigint**(20) NOT NULL,
4. **PRIMARY** **KEY** (`tablename`)
5. ) ENGINE=InnoDB

每当需要为某个表的新纪录生成ID时就从Sequence表中取出对应表的nextid,并将nextid的值加1后更新到数据库中以备下次使用。此方案也较简单，但缺点同样明显：由于所有插入任何都需要访问该表，该表很容易成为系统性能瓶颈，同时它也存在单点问题，一旦该表数据库失效，整个应用程序将无法工作。有人提出使用Master-Slave进行主从同步，但这也只能解决单点问题，并不能解决读写比为1:1的访问压力问题。  
  
除此之外，还有一些方案，像对每个数据库结点分区段划分ID,以及网上的一些ID生成算法，因为缺少可操作性和实践检验，本文并不推荐。实际上，接下来，我们要介绍的是Fickr使用的一种主键生成方案，这个方案是目前我所知道的最优秀的一个方案，并且经受了实践的检验，可以为大多数应用系统所借鉴。

**第二部分：一种极为优秀的主键生成策略**

flickr开发团队在2010年撰文介绍了flickr使用的一种主键生成测策略，同时表示该方案在flickr上的实际运行效果也非常令人满意，原文连接：[Ticket Servers: Distributed Unique Primary Keys on the Cheap](http://code.flickr.com/blog/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/)这个方案是我目前知道的最好的方案，它与一般Sequence表方案有些类似，但却很好地解决了性能瓶颈和单点问题，是一种非常可靠而高效的全局主键生成方案。



**图1. flickr采用的sharding主键生成方案示意图**([点击查看大图](http://ww1.sinaimg.cn/large/67a6a651gw1dujgqcx9ncj.jpg))

flickr这一方案的整体思想是：建立两台以上的数据库ID生成服务器，每个服务器都有一张记录各表当前ID的Sequence表，但是Sequence中ID增长的步长是服务器的数量，起始值依次错开，这样相当于把ID的生成散列到了每个服务器节点上。例如：如果我们设置两台数据库ID生成服务器，那么就让一台的Sequence表的ID起始值为1,每次增长步长为2,另一台的Sequence表的ID起始值为2,每次增长步长也为2，那么结果就是奇数的ID都将从第一台服务器上生成，偶数的ID都从第二台服务器上生成，这样就将生成ID的压力均匀分散到两台服务器上，同时配合应用程序的控制，当一个服务器失效后，系统能自动切换到另一个服务器上获取ID，从而保证了系统的容错。  
  
关于这个方案，有几点细节这里再说明一下：

1. flickr的数据库ID生成服务器是**专用服务器，服务器上只有一个数据库，数据库中表都是用于生成Sequence的，这也是因为auto-increment-offset和auto-increment-increment这两个数据库变量是数据库实例级别的变量。**  
2. flickr的方案中表格中的stub字段只是一个char(1) NOT NULL存根字段，并非表名，因此，一般来说，**一个Sequence表只有一条纪录，可以同时为多张表生成ID，如果需要表的ID是有连续的，需要为该表单独建立Sequence表**。

3. 方案使用了mysql的LAST\_INSERT\_ID()函数，这也决定了Sequence表只能有一条记录。  
4. 使用REPLACE INTO插入数据，这是很讨巧的作法，主要是希望利用mysql自身的机制生成ID,不仅是因为这样简单，更是因为我们需要ID按照我们设定的方式(初值和步长)来生成。

5. SELECT LAST\_INSERT\_ID()必须要于REPLACE INTO语句在同一个数据库连接下才能得到刚刚插入的新ID，否则返回的值总是0  
6. 该方案中Sequence表使用的是**MyISAM**引擎，以获取更高的性能，注意：MyISAM引擎使用的是表级别的锁，**MyISAM对表的读写是串行的**，因此不必担心在并发时两次读取会得到同一个ID(另外，应该程序也不需要同步，每个请求的线程都会得到一个新的connection,不存在需要同步的共享资源)。经过实际对比测试，使用一样的Sequence表进行ID生成，MyISAM引擎要比InnoDB表现高出很多！

7. 可使用纯JDBC实现对Sequence表的操作，以便获得更高的效率，实验表明，即使只使用Spring JDBC性能也不及纯JDBC来得快！

实现该方案，应用程序同样需要做一些处理，主要是两方面的工作：

1. 自动均衡数据库ID生成服务器的访问  
2. 确保在某个数据库ID生成服务器失效的情况下，能将请求转发到其他服务器上执行。