24 MAY 2012

ID即标示符，在某个搜索域内能唯一标示其中某个对象。在关系型数据库中每个表都需要定义一个主键来唯一标示一条记录。为了方便一般都会使用一个auto*increment属性的整形数做为ID。因为数据库本身能保证这个数是在这个表范围内一直累加的，所以任何两条记录不会有相同的ID值，包括已经删除的记录。可是一旦表大到一定程度，要跨机器分表的时候，那么就不能再依靠这个auto*increment字段唯一表示一条记录了。因为此时的搜索域已经扩大到多个机器，而每台机器的auto_increment都是独立增长的。本文总结了几种在分布式环境下可用的ID生成方式。主要内容来自Instagram的一篇Blog。

UUID(universally unique identifier)

正如其名字一样，UUID就是为了要在分布式环境中产生唯一标示符而发布的一个标准。标准中规定UUID长度为16Bytes（128Bits），一般将其表示为550e8400-e29b-41d4-a716-446655440000这种16进制格式，同时将其分为5部分，每部分用-分割，各部分长度分别为8,4,4,12。现在使用的UUID算法有5个版本，分别使用5种不同的算法计算产生。

- UUID1: 依据当前计算机的MAC地址和时钟来生成uuid。
- UUID2: 和版本1类似，不过使用域标示符和本地UID代替了版本1中的时钟信息。
- UUID3: 根据url，域标示符等标示符做MD5 Hash产生的。
- UUID4: 根据产生的随机数来生成。
- UUID5: 和版本3类似，只不过替换成了SHA-1算法。

C++中可以使用Boost.Uuid库来生成UUID。Python中同样有UUID模块，可以生成上述5个版本的UUID。

MongoDB ObjectID

MongoDB中每一条记录都有一个’id’字段用来唯一标示本记录。如果用户插入数据时没有显示提供’id’字段，那么系统会自动生成一个。ObjectID一共12Bytes，设计的时候充分考虑了分布式环境下使用的情况，所以能保证在一个分布式MongoDB集群中唯一。ObjectID格式如下：

```
0         4       7    9       12
+--------+------+----+------+
|time    |pc    |pid |inc   |
+--------+------+----+------+
```

前四个字节是Unix Timestamp。 接着三个字节是当前机器“hostname/mac地址/虚拟编号”其中之一的MD5结果的前3个字节。 接着两个字节是当前进程的PID。 最后三个字节是累加计数器或是一个随机数（只有当不支持累加计数器时才用随机数）。 最后生成的仍然是一个用16进制表示的串，如47cc67093475061e3d95369d。这里MongoDB的ObjectID相对UUID有个很大的优点就是ObjectID是时间上有序的。另外还有ObjectID本身也包含了很多其它有用的信息，通过直接解码ObjectID即可直接获得这些信息。

Snowflake

Snowflake是twitter开源的一款独立的适用于分布式环境的ID生成服务器。生成的ID是64Bits，同时满足高性能（>10K ids/s），低延迟（<2ms）和高可用。与MongoDB ObjectID类似这里生成的ID也是时间上有序的。编码方式也和ObjectID类

似，如下：

```
0            41      51      64
+-----------+------+------+
|time        |pc     |inc    |
+-----------+------+------+
```

前41bits是以微秒为单位的timestamp。 接着10bits是事先配置好的机器ID。 最后12bits是累加计数器。 Ticket Server

这个是Flickr在遇到生成全局ID问题时采用的办法。利用了数据库中auto_increment的特性和MySQL特有的REPLACE INFO命令，专门一个数据库实例用来产生ID。大致的过程是这样的：

首先建立一个表，比如用来产生64bitsID的，叫做’Ticket64’

```
CREATE TABLE `Tickets64` (
  `id` bigint(20) unsigned NOT NULL auto_increment,
  `stub` char(1) NOT NULL default '',
  PRIMARY KEY  (`id`),
  UNIQUE KEY `stub` (`stub`)
) ENGINE=MyISAM
```

向里边插入一条记录后大致是这样：

```
+-------------------+------+
| id                | stub |
+-------------------+------+
| 72157623227190423 |    a |
+-------------------+------+
```

当需要一个64Bits ID的时候，执行如下SQL 语句：

```
REPLACE INTO Tickets64 (stub) VALUES ('a');
SELECT LAST_INSERT_ID();
```

另外为了防止这个Ticket Server单点故障，可以设置两个Ticket Server实例。其中一个产生奇数ID，另一个产生偶数ID。

```
TicketServer1:
auto-increment-increment = 2
auto-increment-offset = 1

TicketServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

应用交替请求两个Server，这样不仅压力减小一半，故障风险也降低一半。不过这里也有个问题，就是当一台机器故障时，另一台正常机器产生的ID将会领先故障机器一截，可能会造成不再是时间上有序的ID。按照Flickr的说法，这并不影响他们的应用。

Instagram采用的方式

Instagram要将其中存储的图片分片到多个PostgreSQL中，其中生成ID的方案和MongoDB ObjectID类似。整个ID的长度为64Bits，设定为这个长度是为了优化在redis中的存储。ID的编码格式如下：

41bits以微秒为单位的timestamp，时间起点从2011-01-01开始。

13bits表示进行逻辑分片的Shard ID。

10bits表示一个累加计数器。

ID的生成逻辑用PL/PGSQL语言写到PostgreSQL数据库中，当每次插入数据时由数据库自动计算生成。