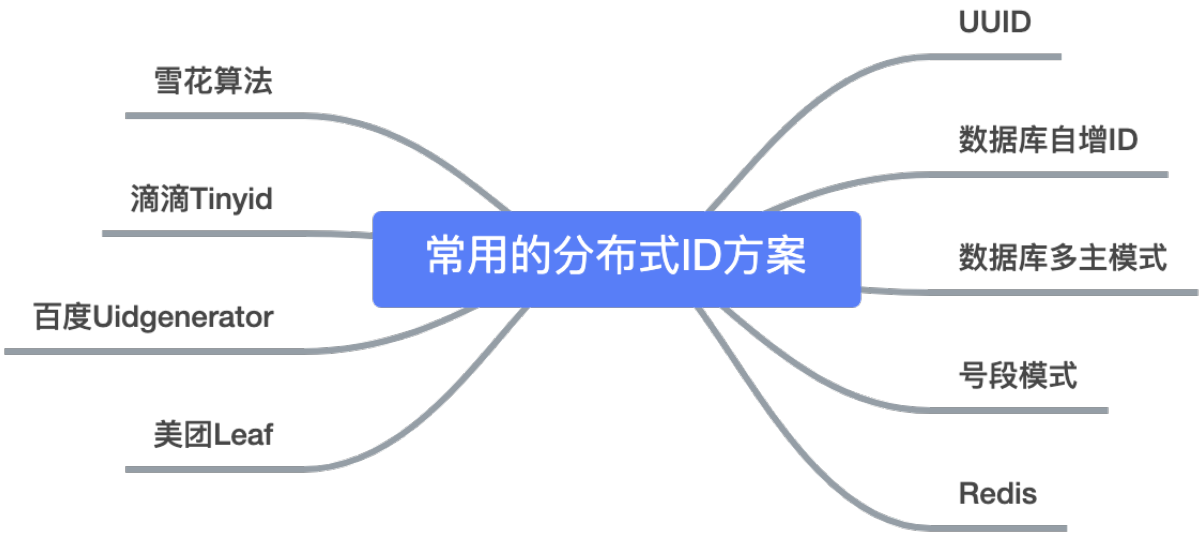
鲁班学院-周瑜

曾参与大型电商平台、互联网金融产品等多家互联网公司的开发，曾就职于大众点评，任项目经理等职位，参与并主导千万级并发电商网站与系统架构搭建

ID是数据的唯一标识，传统的做法是利用UUID和数据库的自增ID，在互联网企业中，大部分公司使用的都是Mysql，并且因为需要事务支持，所以通常会使用Innodb存储引擎，UUID太长以及无序，所以并不适合在Innodb中来作为主键，自增ID比较合适，但是随着公司的业务发展，数据量将越来越大，需要对数据进行分表，而分表后，每个表中的数据都会按自己的节奏进行自增，很有可能出现ID冲突。这时就需要一个单独的机制来负责生成唯一ID，生成出来的ID也可以叫做**分布式ID**，或**全局ID**。下面来分析各个生成分布式ID的机制。



这篇文章并不会分析的特别详细，主要是做一些总结，以后再出一些详细某个方案的文章。

数据库自增ID

第一种方案仍然还是基于数据库的自增ID，需要单独使用一个数据库实例，在这个实例中新建一个单独的表：

表结构如下：

```
CREATE DATABASE `SEQID`;


CREATE TABLE SEQID.SEQUENCE_ID (
    id bigint(20) unsigned NOT NULL auto_increment,
    stub char(10) NOT NULL default '',
    PRIMARY KEY (id),
    UNIQUE KEY stub (stub)
) ENGINE=MyISAM;
```

可以使用下面的语句生成并获取到一个自增ID

```
begin;
replace into SEQUENCE_ID (stub) VALUES ('anyword');
select last_insert_id();
commit;
```

stub字段在这里并没有什么特殊的意义，只是为了方便的去插入数据，只有能插入数据才能产生自增id。而对于插入我们用的是replace，replace会先看是否存在stub指定值一样的数据，如果存在则先delete再insert，如果不存在则直接insert。

这种生成分布式ID的机制，需要一个单独的Mysql实例，虽然可行，但是基于性能与可靠性来考虑的话都不够，**业务系统每次需要一个ID时，都需要请求数据库获取，性能低，并且如果此数据库实例下线了，那么将影响所有的业务系统。**

为了解决数据库可靠性问题，我们可以使用第二种分布式ID生成方案。

数据库多主模式

如果我们两个数据库组成一个**主从模式**集群，正常情况下可以解决数据库可靠性问题，但是如果主库挂掉后，数据没有及时同步到从库，这个时候会出现ID重复的现象。我们可以使用**双主模式**集群，也就是两个Mysql实例都能单独的生产自增ID，这样能够提高效率，但是如果不经过其他改造的话，这两个Mysql实例很可能会生成同样的ID。需要单独给每个Mysql实例配置不同的起始值和自增步长。

第一台Mysql实例配置：

```
set @@auto_increment_offset = 1;     -- 起始值
set @@auto_increment_increment = 2;  -- 步长
```

第二台Mysql实例配置：

```
set @@auto_increment_offset = 2;      -- 起始值
set @@auto_increment_increment = 2;  -- 步长
```

经过上面的配置后，这两个Mysql实例生成的id序列如下：

mysql1,起始值为1,步长为2,ID生成的序列为：1,3,5,7,9,...

mysql2,起始值为2,步长为2,ID生成的序列为：2,4,6,8,10,...

对于这种生成分布式ID的方案，需要单独新增一个生成分布式ID应用，比如DistributIdService，该应用提供一个接口供业务应用获取ID，业务应用需要一个ID时，通过rpc的方式请求DistributIdService，DistributIdService随机去上面的两个Mysql实例中去获取ID。

实行这种方案后，就算其中某一台Mysql实例下线了，也不会影响DistributIdService，DistributIdService仍然可以利用另外一台Mysql来生成ID。

但是这种方案的扩展性不太好，如果两台Mysql实例不够用，需要新增Mysql实例来提高性能时，这时就会比较麻烦。

现在如果要新增一个实例mysql3，要怎么操作呢？

第一，mysql1、mysql2的步长肯定都要修改为3，而且只能是人工去修改，这是需要时间的。

第二，因为mysql1和mysql2是不停在自增的，对于mysql3的起始值我们可能要定得大一点，以给充分的时间去修改mysql1，mysql2的步长。

第三，在修改步长的时候很可能会出现重复ID，要解决这个问题，可能需要停机才行。

为了解决上面的问题，以及能够进一步提高DistributIdService的性能，如果使用第三种生成分布式ID机制。

号段模式

我们可以使用号段的方式来获取自增ID，号段可以理解成批量获取，比如DistributIdService从数据库获取ID时，如果能批量获取多个ID并缓存在本地的话，那样将大大提供业务应用获取ID的效率。

比如DistributIdService每次从数据库获取ID时，就获取一个号段，比如(1,1000]，这个范围表示了1000个ID，业务应用在请求DistributIdService提供ID时，DistributIdService只需要在本地从1开始自增并返回即可，而不需要每次都请求数据库，一直到本地自增到1000时，也就是当前号段已经被用完时，才去数据库重新获取下一号段。

所以，我们需要对数据库表进行改动，如下：

```
CREATE TABLE id_generator (
  id int(10) NOT NULL,
  current_max_id bigint(20) NOT NULL COMMENT '当前最大id',
  increment_step int(10) NOT NULL COMMENT '号段的长度',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

这个数据库表用来记录自增步长以及当前自增ID的最大值（也就是当前已经被申请的号段的最后一个值），因为自增逻辑被移到DistributIdService中去了，所以数据库不需要这部分逻辑了。

这种方案不再强依赖数据库，就算数据库不可用，那么DistributIdService也能继续支撑一段时间。但是如果DistributIdService重启，会丢失一段ID，导致ID空洞。

为了提高DistributIdService的高可用，需要做一个集群，业务在请求DistributIdService集群获取ID时，会随机的选择某一个DistributIdService节点进行获取，对每一个DistributIdService节点来说，数据库连接的是同一个数据库，那么可能会产生多个DistributIdService节点同时请求数据库获取号段，那么这个时候需要利用乐观锁来进行控制，比如在数据库表中增加一个version字段，在获取号段时使用如下SQL：

```
update id_generator set current_max_id=#{newMaxId}, version=version+1 where version =
#{version}
```

因为newMaxId是DistributIdService中根据oldMaxId+步长算出来的，只要上面的update更新成功了就表示号段获取成功了。

为了提供数据库层的高可用，需要对数据库使用多主模式进行部署，对于每个数据库来说要保证生成的号段不重复，这就需要利用最开始的思路，再在刚刚的数据库表中增加起始值和步长，比如如果现在是两台Mysql，那么

mysql1将生成号段（1,1001]，自增的时候序列为1，3，4，5，7....

mysql1将生成号段（2,1002]，自增的时候序列为2，4，6，8，10...

更详细的可以参考滴滴开源的
TinyId：https://github.com/didi/tinyid/wiki/tinyid%E5%8E%9F%E7%90%86%E4%BB%8B%E7%BB%8D

在TinyId中还增加了一步来提高效率，在上面的实现中，ID自增的逻辑是在DistributIdService中实现的，而实际上可以把自增的逻辑转移到业务应用本地，这样对于业务应用来说只需要获取号段，每次自增时不再需要请求调用DistributIdService了。

雪花算法

上面的三种方法总的来说是基于自增思想的，而接下来就介绍比较著名的雪花算法-snowflake。

我们可以换个角度来对分布式ID进行思考，只要能让负责生成分布式ID的每台机器在每毫秒内生成不一样的ID就行了。

snowflake是twitter开源的分布式ID生成算法，是一种算法，所以它和上面的三种生成分布式ID机制不太一样，它不依赖数据库。

核心思想是：分布式ID固定是一个long型的数字，一个long型占8个字节，也就是64个bit，原始snowflake算法中对于bit的分配如下图：

- 第一个bit位是标识部分，在java中由于long的最高位是符号位，正数是0，负数是1，一般生成的ID为正数，所以固定为0。
- 时间戳部分占41bit，这个是毫秒级的时间，一般实现上不会存储当前的时间戳，而是时间戳的差值（当前时间-固定的开始时间），这样可以使产生的ID从更小值开始；41位的时间戳可以使用69年，(1L << 41) / (1000L * 60 * 60 * 24 * 365) = 69年
- 工作机器id占10bit，这里比较灵活，比如，可以使用前5位作为数据中心机房标识，后5位作为单机房机器标识，可以部署1024个节点。
- 序列号部分占12bit，支持同一毫秒内同一个节点可以生成4096个ID

根据这个算法的逻辑，只需要将这个算法用Java语言实现出来，封装为一个工具方法，那么各个业务应用可以直接使用该工具方法来获取分布式ID，只需保证每个业务应用有自己的工作机器id即可，而不需要单独去搭建一个获取分布式ID的应用。

snowflake算法实现起来并不难，提供一个github上用java实现的：https://github.com/beyondfengyu/SnowFlake

在大厂里，其实并没有直接使用snowflake，而是进行了改造，因为snowflake算法中最难实践的就是工作机器id，原始的snowflake算法需要人工去为每台机器去指定一个机器id，并配置在某个地方从而让snowflake从此处获取机器id。

但是在大厂里，机器是很多的，人力成本太大且容易出错，所以大厂对snowflake进行了改造。

百度（uid-generator）

github地址：uid-generator

uid-generator使用的就是snowflake，只是在生产机器id，也叫做workId时有所不同。

uid-generator中的workId是由uid-generator自动生成的，并且考虑到了应用部署在docker上的情况，在uid-generator中用户可以自己去定义workId的生成策略，默认提供的策略是：应用启动时由数据库分配。说的简单一点就是：应用在启动时会往数据库表(uid-generator需要新增一个WORKER_NODE表)中去插入一条数据，数据插入成功后返回的该数据对应的自增唯一id就是该机器的workId，而数据由host，port组成。

对于uid-generator中的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，同一个应用每重启一次就会消费一个workId。

具体可参考https://github.com/baidu/uid-generator/blob/master/README.zh_cn.md

美团（Leaf）

github地址：Leaf

美团的Leaf也是一个分布式ID生成框架。它非常全面，即支持号段模式，也支持snowflake模式。号段模式这里就不介绍了，和上面的分析类似。

Leaf中的snowflake模式和原始snowflake算法的不同点，也主要在workId的生成，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，在启动时都会都在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。

总结

总得来说，上面两种都是自动生成workId，以让系统更加稳定以及减少人工成功。

Redis

这里额外再介绍一下使用Redis来生成分布式ID，其实和利用Mysql自增ID类似，可以利用Redis中的incr命令来实现原子性的自增与返回，比如：

```
127.0.0.1:6379> set seq_id 1     // 初始化自增ID为1
OK
127.0.0.1:6379> incr seq_id      // 增加1，并返回
(integer) 2
127.0.0.1:6379> incr seq_id      // 增加1，并返回
(integer) 3
```

使用redis的效率是非常高的，但是要考虑持久化的问题。Redis支持RDB和AOF两种持久化的方式。

RDB持久化相当于定时打一个快照进行持久化，如果打完快照后，连续自增了几次，还没来得及做下一次快照持久化，这个时候Redis挂掉了，重启Redis后会出现ID重复。

AOF持久化相当于对每条写命令进行持久化，如果Redis挂掉了，不会出现ID重复的现象，但是会由于incr命令过得，导致重启恢复数据时间过长。