分布式唯一 ID 实战

背景

日常开发中，我们需要对系统中的各种数据使用 ID 唯一表示，比如用户 ID 对应且仅对应一个人，商品 ID 对应且仅对应一件商品，订单 ID 对应且仅对应一个订单。我们现实生活中也有各种 ID，比如身份证 ID 对应且仅对应一个人，简单来说，ID 就是数据的唯一标识。

一般情况下，会使用数据库的自增主键作为数据 ID，但是在大数量的情况下，我们往往会引入分布式、分库分表等手段来应对，很明显对数据分库分表后我们依然需要有一个唯一 ID 来标识一条数据或消息，数据库的自增 ID 已经无法满足需求。此时一个能够生成全局唯一 ID 的系统是非常必要的。概括下来，那业务系统对 ID 号的要求有哪些呢？

全局唯一性：不能出现重复的 ID 号，既然是唯一标识，这是最基本的要求。

趋势递增、单调递增：保证下一个 ID 一定大于上一个 ID。

信息安全：如果 ID 是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定 URL 即可；如果是订单号就更危险了，竞对可以直接知道我们一天的单量。所以在一些应用场景下，会需要 ID 无规则、不规则。

同时除了对 ID 号码自身的要求，业务还对 ID 号生成系统的可用性要求极高，想象一下，如果 ID 生成系统不稳定，大量依赖 ID 生成系统，比如订单生成等关键动作都无法执行。所以一个 ID 生成系统还需要做到平均延迟和 TP999 延迟都要尽可能低；可用性 5 个 9；高 QPS。

常见方法介绍

UUID

UUID(Universally Unique Identifier)的标准型式包含 32 个 16 进制数字，以连字号分为五段，形式为 8-4-4-4-12 的 36 个字符，示例：550e8400-e29b-41d4-a716-446655440000，到目前为止业界一共有 5 种方式生成 UUID，详情见 IETF 发布的 UUID 规范 A Universally Unique IDentifier (UUID) URN Namespace。

优点：

性能非常高：本地生成，没有网络消耗。

缺点：

不易于存储：UUID 太长，16 字节 128 位，通常以 36 长度的字符串表示，很多场景不适用。

信息不安全：基于 MAC 地址生成 UUID 的算法可能会造成 MAC 地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。

ID作为主键时在特定的环境会存在一些问题，比如做DB主键的场景下，UUID就非常不适用：

① MySQL官方有明确的建议主键要尽量越短越好[4]，36个字符长度的UUID不符合要求。

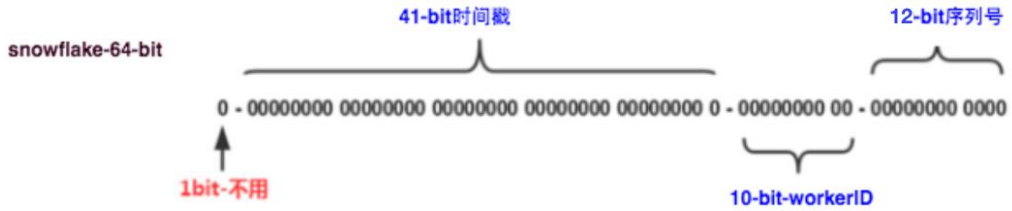
② 对MySQL索引不利：如果作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能。在MySQL InnoDB引擎中使用的是聚集索引，由于多数RDBMS使用B-tree的数据结构来存储索引数据，在主键的选择上面我们应该尽量使用有序的主键保证写入性能。

可以直接使用jdk自带的UUID，原始生成的是带中划线的，如果不需要，可自行去除

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        String rawUUID = UUID.randomUUID().toString();
        System.out.println(rawUUID);
        //去除"-"
        String uuid = rawUUID.replaceAll( regex: "-",  replacement: "");
        System.out.println(uuid);
    }
}
```

雪花算法及其衍生

这种方案大致来说是一种以划分命名空间（UUID也算，由于比较常见，所以单独分析）来生成ID的一种算法，Snowflake是Twitter开源的分布式ID生成算法。Snowflake把64-bit分别划分成多段，分开来标示机器、时间等，比如在snowflake中的64-bit分别表示如下图所示：



第0位： 符号位（标识正负），始终为0，没有用，不用管。

第1~41位：一共41位，用来表示时间戳，单位是毫秒，可以支撑2^41毫秒（约69年）

第42~52位：一共10位，一般来说，前5位表示机房ID，后5位表示机器ID（实际项目中可以根据实际情况调整），这样就可以区分不同集群/机房的节点，这样就可以表示32个IDC，每个IDC下可以有32台机器。

第53~64位：一共12位，用来表示序列号。 序列号为自增值，代表单台机器每毫秒能够产生的最大ID数(2^12 = 4096),也就是说单台机器每毫秒最多可以生成4096个唯一ID。

理论上snowflake方案的QPS约为409.6w/s，这种分配方式可以保证在任何一个IDC的任何一台机器在任意毫秒内生成的ID都是不同的。

有很多基于 Snowflake 算法的开源实现比如美团的 Leaf、百度的 UidGenerator(自 18 年后，UidGenerator 就基本没有再维护了，https://github.com/baidu/uid-generator/blob/master/README.zh_cn.md)，并且这些开源实现对原有的 Snowflake 算法进行了优化。在实际项目中，我们一般也会对 Snowflake 算法进行改造，最常见的就是在算法生成的 ID 中加入业务类型信息。

关于自行实现 Snowflake 算法，可以参考 tulingmall-unqid 下的 com.tuling.tulingmall.service.snowflake 下的代码。

Snowflake 优缺点是：

优点：

毫秒数在高位，自增序列在低位，整个 ID 都是趋势递增的。

不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成 ID 的性能也是非常高的。

可以根据自身业务特性分配 bit 位，非常灵活。

缺点：

强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。

当然，在我们自己的项目如果不想自行实现唯一性 ID，还可以利用外部中间件，比如 Mongdb objectID，它也可以算作是和 snowflake 类似方法，通过“时间+机器码+pid+inc”共 12 个字节，通过 4+3+2+3 的方式最终标识成一个 24 长度的十六进制字符。

其次 Seata 内置了一个分布式 UUID 生成器，用于辅助生成全局事务 ID 和分支事务 ID，我们同样可以拿来使用，完整类名为： io.seata.common.util.IdWorker

数据库生成

MYSQL

以 MySQL 举例，

1.创建一个数据库表。

```
CREATE TABLE `sequence_id` (
    `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
    `stub` char(10) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`),
    UNIQUE KEY `stub` (`stub`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

stub 字段无意义，只是为了占位，便于我们插入或者修改数据。并且，给 stub 字段创建了唯一索引，保证其唯一性。

2.通过 replace into 来插入数据。

```
BEGIN;
```

REPLACE INTO sequence_id (stub) VALUES ('stub');

SELECT LAST_INSERT_ID();

COMMIT;

插入数据这里，我们没有使用 insert into 而是使用 replace into 来插入数据。replace 是 insert 的增强版，replace into 首先尝试插入数据到表中，1. 如果发现表中已经有此行数据（根据主键或者唯一索引判断）则先删除此行数据，然后插入新的数据。 2. 否则，直接插入新数据。

数据库方案的优缺点如下：

优点：

非常简单，利用现有数据库系统的功能实现，成本小，有 DBA 专业维护。ID 号单调自增，存储消耗空间小。

缺点：

支持的并发量不大、存在数据库单点问题（可以使用数据库集群解决，不过增加了复杂度）、ID 没有具体业务含义、安全问题（比如根据订单 ID 的递增规律就能推算出每天的订单量，商业机密啊！ ）、每次获取 ID 都要访问一次数据库（增加了对数据库的压力，获取速度也慢）

对于 MySQL 性能问题，可用如下方案解决：在分布式系统中我们可以多部署几台机器，每台机器设置不同的初始值，且步长和机器数相等。比如有两台机器。设置步长 step 为 2，TicketServer1 的初始值为 1（1，3，5，7，9，11…）、TicketServer2 的初始值为 2（2，4，6，8，10…）。这是 Flickr（雅虎旗下图片分享网站）团队在 2010 年撰文介绍的一种主键生成策略（Ticket Servers: Distributed Unique Primary Keys on the Cheap ）。为了实现上述方案分别设置两台机器对应的参数，TicketServer1 从 1 开始发号，TicketServer2 从 2 开始发号，两台机器每次发号之后都递增 2。

假设我们要部署 N 台机器，步长需设置为 N，每台的初始值依次为 0,1,2…N-1。

这种架构貌似能够满足性能的需求，但有以下几个缺点：

系统水平扩展比较困难，比如定义好了步长和机器台数之后，如果要添加机器该怎么做？假设现在只有一台机器发号是 1,2,3,4,5（步长是 1），这个时候需要扩容机器一台。可以这样做：把第二台机器的初始值设置得比第一台超过很多，比如 140（假设在扩容时间之内第一台不可能发到 140），同时设置步长为 2，那么这台机器下发的号码都是 140 以后的偶数。然后摘掉第一台，把 ID 值保留为奇数，比如 7，然后修改第一台的步长为 2。让它符合我们定义的号段标准，对于这个例子来说就是让第一台以后只能产生奇数。扩容方案看起来复杂吗？貌似还好，现在想象一下如果我们线上有 100 台机器，这个时候要扩容该怎么做？简直是噩梦。所以系统水平扩展方案复杂难以实现。

ID 没有了单调递增的特性，只能趋势递增，这个缺点对于一般业务需求不是很重要，可以容忍。

数据库压力还是很大，每次获取 ID 都得读写一次数据库，只能靠堆机器来提高性能。

Redis

通过 Redis 的 incr 命令即可实现对 id 原子顺序递增，例如：

127.0.0.1:6379> incr sequence_id_biz_type

(integer) 2

为了提高可用性和并发，我们可以使用 Redis Cluster。

除了高可用和并发之外，我们知道 Redis 基于内存，我们需要持久化数据，避免重启机器或者机器故障后数据丢失。很明显，Redis 方案性能很好并且生成的 ID 是有序递增的。

不过，我们也知道，即使 Redis 开启了持久化，不管是快照（snapshotting，RDB）、只追加文件（append-only file, AOF）还是 RDB 和 AOF 的混合持久化依然存在着丢失数据的可能，那就意味着产生的 ID 存在着重复的概率。

分布式 ID 微服务

从上面的分析可以看出，每种方案都各有优劣，在我们的商城系统中则基于美团的 Leaf 实现了自己的分布式 ID 微服务。我们先来看看美团 Leaf 方案。

美团 Leaf 方案实现

Leaf 这个名字是来自德国哲学家、数学家莱布尼茨的一句话： There are no two identical leaves in the world（“世界上没有两片相同的树叶”）

Leaf 分别在 MySQL 和雪花上做了相应的优化，实现了 Leaf-segment 和 Leaf-snowflake 方案。

Leaf-segment 数据库方案

Leaf-segment 方案，在使用数据库的方案上，做了如下改变：

原 MySQL 方案每次获取 ID 都得读写一次数据库，造成数据库压力大。改为批量获取，每次获取一个 segment(step 决定大小)号段的值。用完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。

各个业务不同的发号需求用 biz_tag 字段来区分，每个 biz-tag 的 ID 获取相互隔离，互不影响。如果以后有性能需求需要对数据库扩容，不需要上述描述的复杂的扩容操作，只需要对 biz_tag 分库分表就行。

数据库表设计如下：

名	类型	长度	小
biz_tag	varchar	128	0
max_id	bigint	0	0
step	int	0	0
description	varchar	256	0
update_time	timestamp	0	0

重要字段说明：biz_tag 用来区分业务，max_id 表示该 biz_tag 目前所被分配的 ID 号段的最大值，step 表示每次分配的号段长度。原来获取 ID 每次都需要写

数据库，现在只需要把 step 设置得足够大，比如 1000。那么只有当 1000 个号被消耗完了之后才会去重新读写一次数据库。读写数据库的频率从 1 减小到了 1/step。

例如现在有 3 台机器，每台机器各取 1000 个，很明显在第一台 Leaf 机器上是 1~1000 的号段，当这个号段用完时，会去加载另一个长度为 step=1000 的号段，假设另外两台号段都没有更新，这个时候第一台机器新加载的号段就应该是 3001~4000。同时数据库对应的 biz_tag 这条数据的 max_id 会从 3000 被更新成 4000，更新号段的 SQL 语句如下：

```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

这种模式有以下优缺点：

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。

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。

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。

可以自定义 max_id 的大小，非常方便业务从原有的 ID 方式上迁移过来。

缺点：

ID 号码不够随机，能够泄露发号数量的信息，不太安全。

TP999 数据波动大，当号段使用完之后还是会在获取新号段时在更新数据库的 I/O 依然会存在着等待，tg999 数据会出现偶尔的尖刺。

DB 宕机会造成整个系统不可用。

双 buffer 优化

对于第二个缺点，Leaf-segment 做了一些优化，简单的说就是：

Leaf 取号段的时机是在号段消耗完的时候进行的，也就意味着号段临界点的 ID 下发时间取决于下一次从 DB 取回号段的时间，并且在这期间进来的请求也会因为 DB 号段没有取回来，导致线程阻塞。如果请求 DB 的网络和 DB 的性能稳定，这种情况对系统的影响是不大的，但是假如取 DB 的时候网络发生抖动，或者 DB 发生慢查询就会导致整个系统的响应时间变慢。

为此，希望 DB 取号段的过程能够做到无阻塞，不需要在 DB 取号段的时候阻塞请求线程，即当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。这样做就可以很大程度上的降低系统的 TP999 指标。

采用双 buffer 的方式，Leaf 服务内部有两个号段缓存区 segment。当前号段已下发 10%时，如果下一个号段未更新，则另启一个更新线程去更新下一个号段。

当前号段全部下发完后，如果下个号段准备好了则切换到下个号段为当前 segment 接着下发，循环往复。

通常推荐 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。

每次请求来临时都会判断下个号段的状态，从而更新此号段，所以偶尔的网络抖动不会影响下个号段的更新。

Leaf 高可用容灾

对于第三点“DB 可用性”问题，可以采用一主两从的方式，同时分机房部署，Master 和 Slave 之间采用半同步方式同步数据。美团内部使用了奇虎 360 的 Atlas 数据库中间件（已开源，改名为 DBProxy）做主从切换。当然这种方案在一些情况会退化成异步模式，甚至在非常极端情况下仍然会造成数据不一致的情况，但是出现的概率非常小。如果要保证 100%的数据强一致，可以选择使用“类 Paxos 算法”实现的强一致 MySQL 方案，如 MySQL 5.7 中的 MySQL Group Replication。但是运维成本和精力都会相应的增加，根据实际情况选型即可。

Leaf-snowflake 方案

Leaf-segment 方案可以生成趋势递增的 ID，同时 ID 号是可计算的，不适用于订单 ID 生成场景，比如竞对在两天中午 12 点分别下单，通过订单 id 号相减就能大致计算出公司一天的订单量，这个是不能忍受的。面对这一问题，美团提供了 Leaf-snowflake 方案。

Leaf-snowflake 方案完全沿用 snowflake 方案的 bit 位设计，即是“1+41+10+12”的方式组装 ID 号。对于 workerID 的分配，当服务集群数量较小的情况下，完全可以手动配置。Leaf 服务规模较大，动手配置成本太高。所以使用 Zookeeper 持久顺序节点的特性自动对 snowflake 节点配置 wokerID。Leaf-snowflake 是按照下面几个步骤启动的：

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）。

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。

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。

弱依赖 ZooKeeper

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。当 ZooKeeper 出现问题，恰好机器出现问题需要重启时，能保证服务能够正常启动。这样做到了对三方组件的弱依赖。

解决时钟问题

因为这种方案依赖时间，如果机器的时钟发生了回拨，那么就会有可能生成重复的 ID 号，需要解决时钟回退的问题。

首先在启动时，服务会进行检查：

1、新节点通过检查综合对比其余 Leaf 节点的系统时间来判断自身系统时间是否准确，具体做法是取所有运行中的 Leaf-snowflake 节点的服务 IP：Port，然后通过 RPC 请求得到所有节点的系统时间，计算 sum(time)/nodeSize，然后看本机时间与这个平均值是否在阈值之内来确定当前系统时间是否准确，准确正常启动服务，不准确认为本机系统时间发生大步长偏移，启动失败并报警。

2、在 ZooKeeper 中登记过的老节点，同样会比较自身系统时间和 ZooKeeper 上本节点曾经的记录时间以及所有运行中的 Leaf-snowflake 节点的时间，不准确同样启动失败并报警。

另外，在运行过程中，每隔一段时间节点都会上报自身系统时间写入 ZooKeeper 。

在服务运行过程中，机器的 NTP 同步也会造成秒级别的回退，由于强依赖时钟，对时间的要求比较敏感，美团建议有三种解决方案，一是可以直接关闭 NTP 同步；二是在时钟回拨的时候直接不提供服务直接返回 ERROR_CODE，等时钟追上即可，三是做一层重试，然后上报报警系统，更或者是发现有时钟回拨之后自动摘除本身节点并报警，代码如下：

```
if (timestamp < lastTimestamp) {
    long offset = lastTimestamp - timestamp;
    if (offset <= 5) {
        try {
            wait( timeout: offset << 1);
            timestamp = timeGen();
            if (timestamp < lastTimestamp) {
                return new Result( id: -1, Status.EXCEPTION);
            }
        } catch (InterruptedException e) {
            LOGGER.error("wait interrupted");
            return new Result( id: -2, Status.EXCEPTION);
        }
    } else {
        return new Result( id: -3, Status.EXCEPTION);
    }
}
```

从美团的实际运行情况来看，在 2017 年闰秒出现那一次出现过部分机器回拨，由于 Leaf-snowflake 的策略保证，成功避免了对业务造成的影响。

美团 Leaf 现状

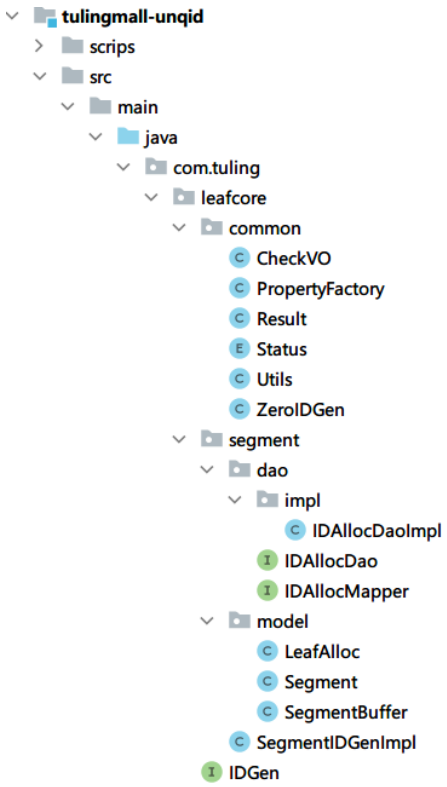
Leaf 在美团点评公司内部服务包含金融、支付交易、餐饮、外卖、酒店旅游、猫眼电影等众多业务线。目前 Leaf 的性能在 4C8G 的机器上 QPS 能压测到近 5

万/s，TP999 1ms，已经能够满足大部分的业务的需求。每天提供亿数量级的调用量。

tulingmall-unqid 实现

从上面的分析可以看到，生成全局唯一 ID 的系统对于我们的项目来说是必须的，从诸多因素考虑，我们选用了美团的 Leaf 并根据项目的实际情况做了裁剪和改造。

首先，在我们的整个的商品系统中并没有安装 Zookeeper 而且也不考虑竞对，所以在 tulingmall-unqid 中完全去除了有关 Leaf-snowflake 的部分，从美团 Leaf 和我们的代码比较即可看出：



其次，在美团 Leaf 的实现中，可以看到对外提供 ID 的方法

```
@RequestMapping(value = "/api/segment/get/{key}")
public String getSegmentId(@PathVariable("key") String key) {
    return get(key, segmentService.getId(key));
}

@RequestMapping(value = "/api/snowflake/get/{key}")
public String getSnowflakeId(@PathVariable("key") String key) {
    return get(key, snowflakeService.getId(key));
}
```

很明显，一次只能提供一个 ID，但是仔细考察商城系统的业务需求，比如订单，我们知道一个订单往往分为两个部分，订单的基本信息和订单详情，订单详情往往包含该订单的产品列表，在保存时我们往往也会用两张表来保存，一是订单表，二是订单详情表。订单表的 ID 很好说，每次从唯一 ID 服务取一个 ID

即可，但是订单详情表呢？我们会一次性插入一条订单记录和多条订单详情记录，如果对于订单详情记录的 ID 每次都从唯一 ID 服务取，这个无疑会对性能有影响，解决办法有两个：

1、订单详情记录的 ID 不保证全局唯一，依然使用数据库的自增主键；

2、订单详情记录的 ID 需要全局唯一，但并不每次从唯一 ID 服务，而是在生成订单时，一次性从唯一 ID 服务获得。

在我们的商城系统中，我们选择了第二种方式，很自然就需要对原来的美团 Leaf 进行改造：

```
@RequestMapping(value = "/api/segment/get/{key}")
public String getSegmentId(@PathVariable("key") String key) {
    return get(key, segmentService.getId(key));
}

Mark
@RequestMapping(value = "/api/segment/getlist/{key}")
public List<String> getSegmentIdList(@PathVariable("key") String key,@RequestParam int keyNumber) {
    if (keyNumber == 0 || keyNumber > 5000) keyNumber = 5000;
    return getList(key, segmentService.getIds(key,keyNumber));
}
```

可以看到，我们新增了一个批量获得唯一 ID 的方法，并限定每次可以获得 ID 的最大数量为 5000 个。

我们的 tulingmall-unqid 本身是个无状态的服务，可以很方便的进行服务集群，以高伸缩性来应对服务的高可用、性能上的需求。

本文档分享地址：

http://note.youdao.com/noteshare?id=13a93cf835f8c9a4c37a437d358e8f8a&sub=CA5C157A2F3841A7A0E8DBE72BC2BA2D