标点符

全局唯一ID生成方案

2018年8月1日 · 1 min read

在实现大型分布式程序时，通常会有全局唯一ID生成的需求，用来对每一个对象标识一个代号。另外，业务层对于全局唯一ID生成也有要求：

- 全局唯一性：不能出现重复的ID号。
- 趋势递增：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。
- 单调递增：保证下一个ID一定大于上一个ID，例如事务版本号、IM增量消息、排序等特殊需求。
- 信息安全：如果ID是连续的，恶意用户的抓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，竞争对手可以直接知道一天的单量。所以在一些应用场景下，会需要ID无规则、不规则。

先前的文章中介绍介绍了SnowFlake及SnowFlake变种。这篇文章再做一些补充。

常见的全局唯一ID生成方案

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

UUID变种

UUID变种比较流行的是基于MySQL UUID的变种：timestamp + machine number + random，具体介绍见：GUID/UUID Performance Breakthrough

优点：

- 开发成本较低

缺点：

- 基于MySQL的存储过程，性能较差

另外，随着UUID_TO_BIN(str, swap_flag)方法的出现，以上实现方式已不太适用。

Snowflake或其变种

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，这种方案把64-bit分别划分成多段：timestamp + work number + seq number，分开来标示机器、时间等。详细内容可以查看先前的文章。

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。
- 需要引入zookeeper 和独立的snowflake专用服务器

MongoDB官方文档 ObjectID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。相比snowflake长度及可读性要差一些。

Flickr的数据库自增

Flickr的数据库自增方式在先前的文章中也介绍过，flickr是用的一个叫做ticketserver的玩意，使用纯mysql来实现的。

```
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
```

先插入一条记录，然后再用replace去获取这个id

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

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，可以实现一些对ID有特殊要求的业务。

缺点：

- 强依赖DB，当DB异常时整个系统不可用，属于致命问题。配置主从复制可以尽可能的增加可用性，但是数据一致性在特殊情况下难以保证。主从切换时的不一致可能会导致重复发号。
- ID发号性能瓶颈限制在单台MySQL的读写性能。

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...）

Instagram的存储过程

同样，Instagram的ID生成方式在前面的文章中也介绍过，简单的描述为：41b ts + 13b shard id + 10b increment seq，具体实现方式如下：

创建存储过程：

```
CREATE OR REPLACE FUNCTION insta5.next_id(OUT result bigint) AS $$
DECLARE
    our_epoch bigint := 1314220021721;
    seq_id bigint;
    now_millis bigint;
    shard_id int := 5;
BEGIN
    SELECT nextval('insta5.table_id_seq') %% 1024 INTO seq_id;
    SELECT FLOOR(EXTRACT(EPOCH FROM clock_timestamp()) * 1000) INTO now_m
    result := (now_millis - our_epoch) << 23;
    result := result | (shard_id <<10);
    result := result | (seq_id);
END;
$$ LANGUAGE PLPGSQL;
```

创建表：

```
CREATE TABLE insta5.our_table (
    "id" bigint NOT NULL DEFAULT insta5.next_id(),
    ...rest of table schema...
  )
```

详细介绍见：Sharding & IDs at Instagram

优点：

- 开发成本低

缺点：

- 基于postgreSQL的存储过程，通用性差

美团点评分布式ID生成系统

Leaf-segment数据库方案

在使用数据库的方案上，做了如下改变：

- 原方案每次获取ID都得读写一次数据库，造成数据库压力大。改为利用proxy server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。

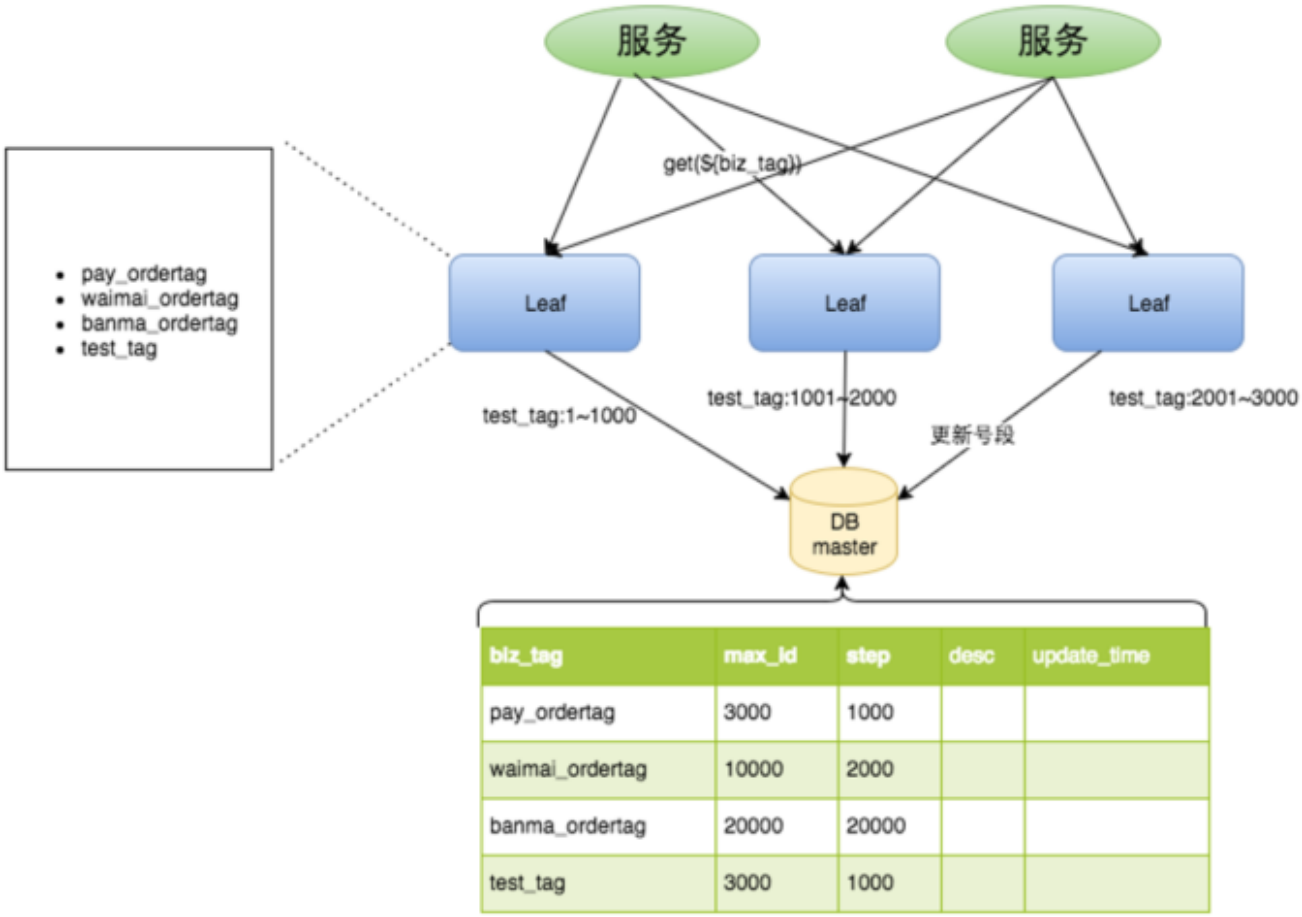
数据库表设计如下：

```
+-------------+--------------+------+-----+-------------------+-----------
| Field       | Type         | Null | Key | Default           | Extra
+-------------+--------------+------+-----+-------------------+-----------
| biz_tag     | varchar(128) | NO   | PRI |                   |
| max_id      | bigint(20)   | NO   |     | 1                 |
| step        | int(11)      | NO   |     | NULL              |
| desc        | varchar(256) | YES  |     | NULL              |
| update_time | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update (
+-------------+--------------+------+-----+-------------------+-----------
```

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。

大致架构如下图所示：



biz_tag	max_id	step	desc	update_time
pay_ordertag	3000	1000		
waimai_ordertag	10000	2000		
banma_ordertag	20000	20000		
test_tag	3000	1000		

test_tag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
```

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。

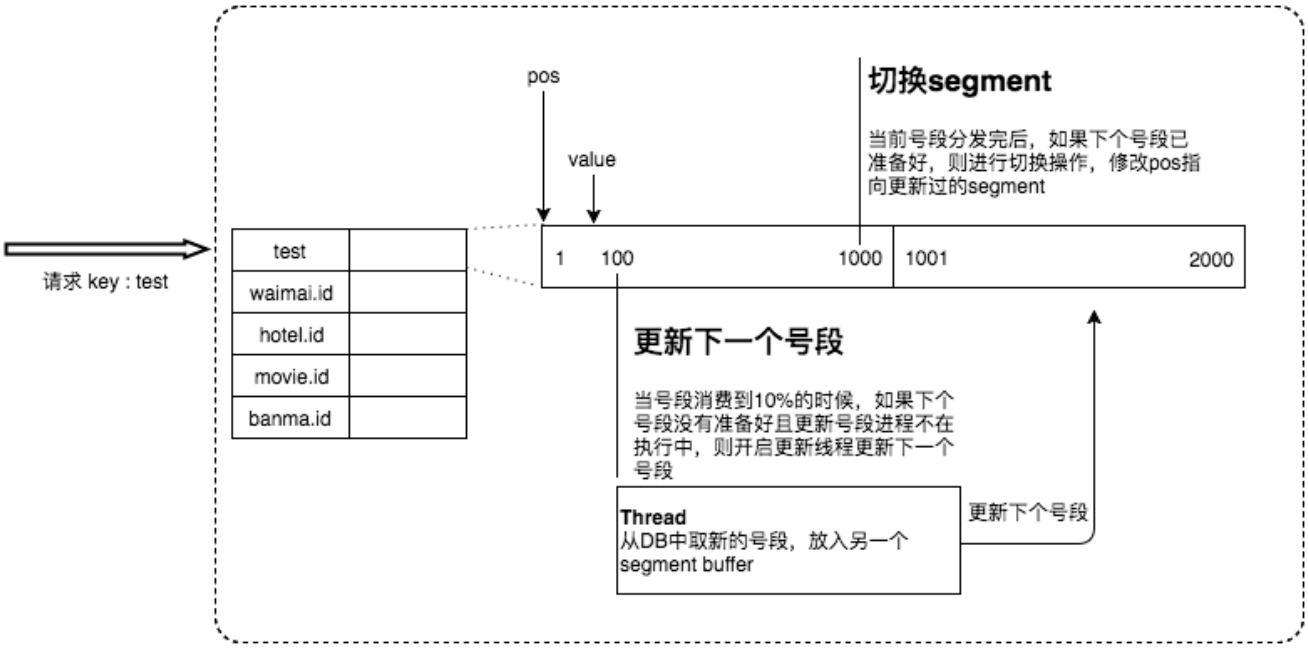
- TP999数据波动大，当号段使用完之后还是会hang在更新数据库的I/O上，tg999数据会出现偶尔的尖刺。
- DB宕机会造成整个系统不可用。

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，我们希望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。详细实现如下图所示：



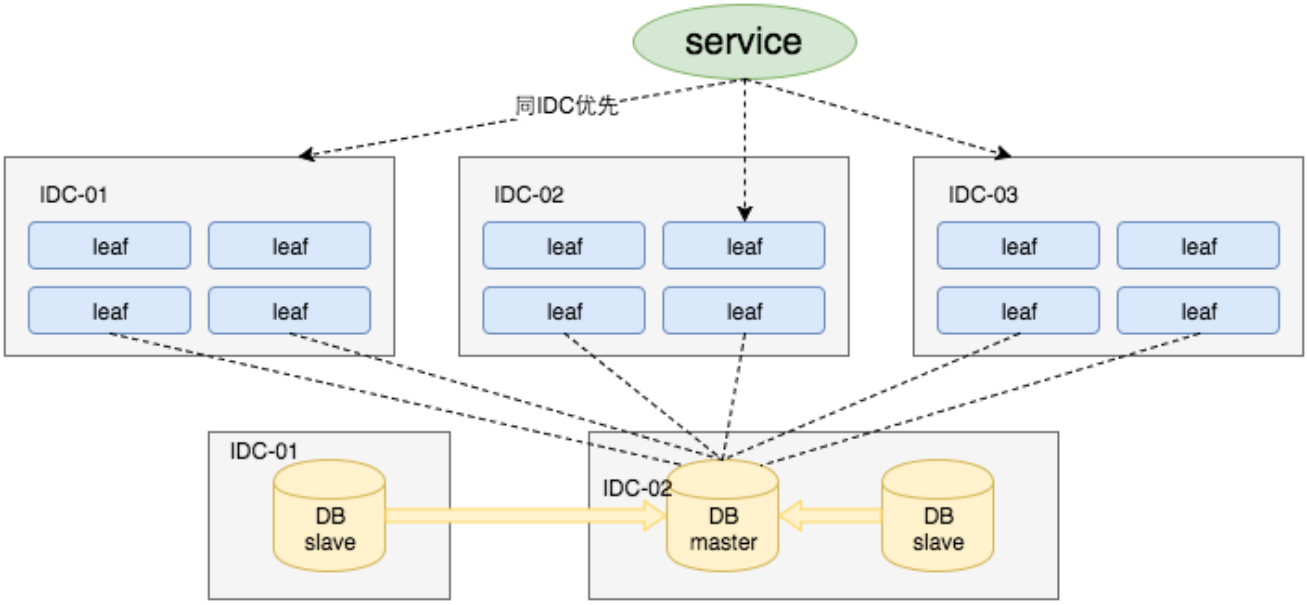
采用双buffer的方式，Leaf服务内部有两个号段缓存区segment。当前号段已下发10%时，如果下一个号段未更新，则另启一个更新线程去更新下一个号段。当前号段全部下发完后，如果下个号段准备好了则切换到下个号段为当前segment接着下发，循环往复。

每个biz-tag都有消费速度监控，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，我们目前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。同时使用公司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。如果你的系统要保证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前段时间刚刚GA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。

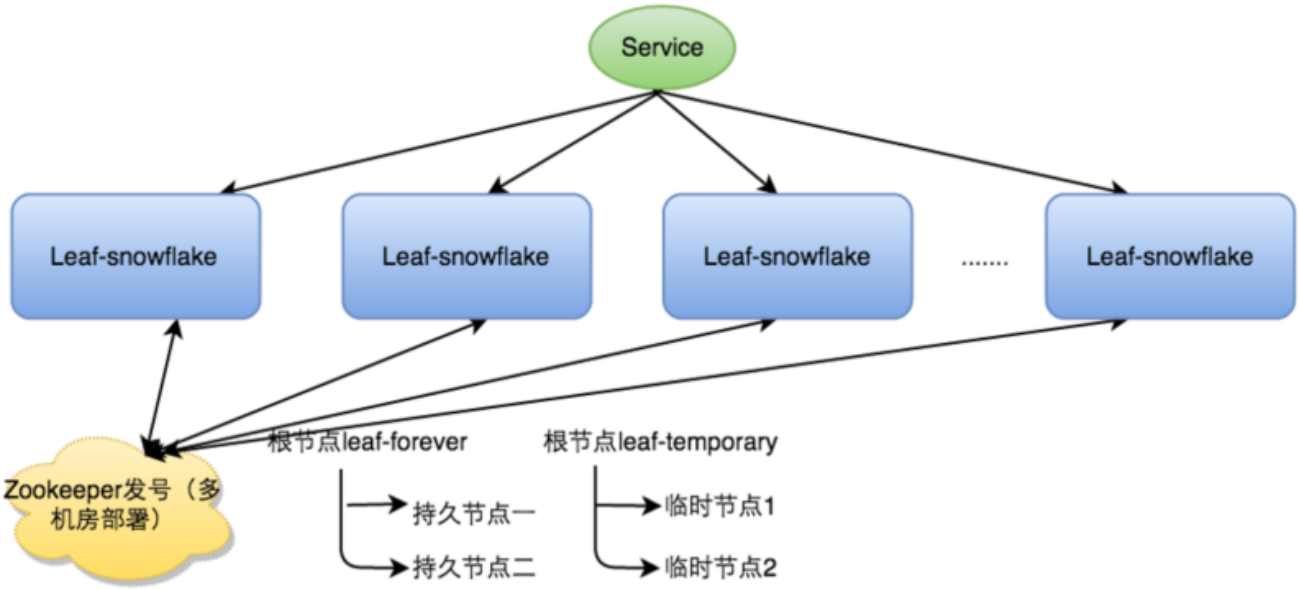


同时Leaf服务分IDC部署，内部的服务化框架是“MTthrift RPC”。服务调用的时候，根据负载均衡算法会优先调用同机房的Leaf服务。在该IDC内Leaf服务不可用的时候才会选择其他机房的Leaf服务。同时服务治理平台OCTO还提供了针对服务的过载保护、一键截流、动态流量分配等对服务的保护措施。

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。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。

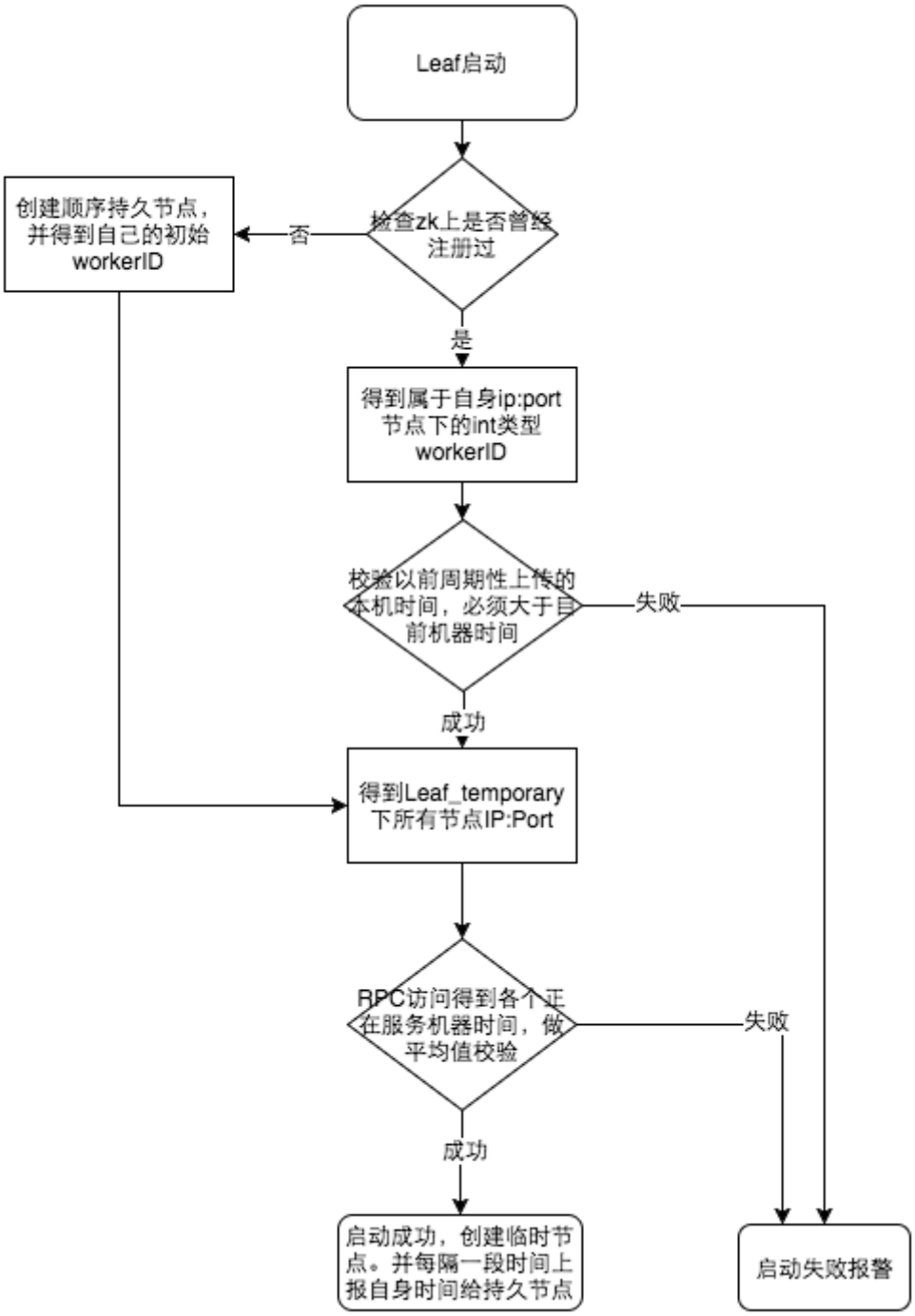


弱依赖ZooKeeper

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。一定程度上提高了SLA

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。



参见上图整个启动流程图，服务启动时首先检查自己是否写过ZooKeeper leaf_forever节点：

- 若写过，则用自身系统时间与leaf_forever/${self}节点记录时间做比较，若小于leaf_forever/${self}时间则认为机器时间发生了大步长回拨，服务启动失败并报警。
- 若未写过，证明是新服务节点，直接创建持久节点leaf_forever/${self}并写入自身系统时间，接下来综合对比其余Leaf节点的系统时间来判断自身系统时间是否准确，具体做法是取leaf_temporary下的所有临时节点(所有运行中的Leaf-snowflake节点)的服务IP：Port，然后通过RPC请求得到所有节点的系统时间，计算sum(time)/nodeSize。
- 若abs(系统时间-sum(time)/nodeSize) < 阈值，认为当前系统时间准确，正常启动服务，同时写临时节点leaf_temporary/${self} 维持租约。

- 否则认为本机系统时间发生大步长偏移，启动失败并报警。
- 每隔一段时间(3s)上报自身系统时间写入leaf_forever/${self}。

由于强依赖时钟，对时间的要求比较敏感，在机器工作时NTP同步也会造成秒级别的回退，建议可以直接关闭NTP同步。要么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。或者做一层重试，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，如下：

```
//发生了回拨，此刻时间小于上次发号时间
 if (timestamp < lastTimestamp) {

            long offset = lastTimestamp - timestamp;
            if (offset <= 5) {
                try {
                    //时间偏差大小小于5ms，则等待两倍时间
                    wait(offset << 1);//wait
                    timestamp = timeGen();
                    if (timestamp < lastTimestamp) {
                       //还是小于，抛异常并上报
                        throwClockBackwardsEx(timestamp);
                      }
                } catch (InterruptedException e) {
                   throw  e;
                }
            } else {
                //throw
                throwClockBackwardsEx(timestamp);
            }
        }
 //分配ID
```

参考链接：https://tech.meituan.com/MT_Leaf.html

打赏作者

微信支付 

支付宝

程序开发

#分布式

« Paul Graham 的创业建议

用户模型之三层身份模型 »

0

Windows安装Docker记录

Docker是一种容器技术，可以将应用和环境等进行打包，形成一个独立的、类似于iOS的APP形式的应用。这个应用可以直接分发到任意一个支持Docker的环境中，通过简单的命令即可启动运行。Docker是一种最流行的容器化实现方案，和虚拟化技术类似，它极大的方便 ...

Aug 14, 2018 · 22 sec read

Windows下安装Tesserocr

在爬虫过程中，经常会遇到各种验证码，大多数验证码是图形验证码，先前的文章中有介绍到破解图形验证码的原理。最简单的破击验证码的方式是使用OCR。

Aug 13, 2018 · 1 min read

Selenium在Windows 上的安装

Selenium是一个用于Web应用程序自动化测试工具。Selenium测试直接运行在浏览器中，就像真正的用户在操作一样。Selenium是一款使用Apache License 2.0协议发布的开源框架。

Aug 13, 2018 · 1 min read

Leave a Reply

Write a response...

Name

E-mail address

Website Link

Post Comment