美团（Leaf）分布式ID生成器，好用的一批！

[架构师社区](javascript:void(0);) 今天

以下文章来源于程序员内点事 ，作者程序员内点事

**美团（Leaf）**

Leaf是美团推出的一个分布式ID生成服务，名字取自德国哲学家、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”），取个名字都这么有寓意，美团程序员牛掰啊！

Leaf的优势：高可靠、低延迟、全局唯一等特点。

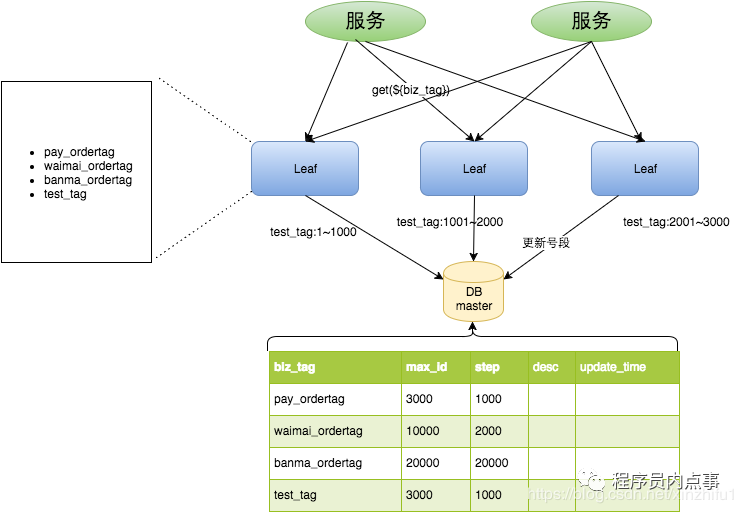
目前主流的分布式ID生成方式，大致都是基于数据库号段模式和雪花算法（snowflake），而美团（Leaf）刚好同时兼具了这两种方式，可以根据不同业务场景灵活切换。

接下来结合实战，详细的介绍一下Leaf的Leaf-segment号段模式和Leaf-snowflake模式

**一、 Leaf-segment号段模式**

Leaf-segment号段模式是对直接用数据库自增ID充当分布式ID的一种优化，减少对数据库的频率操作。相当于从数据库批量的获取自增ID，每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个ID，业务服务将号段在本地生成1~1000的自增ID并加载到内存.。

大致的流程入下图所示：



号段耗尽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。对max\_id字段做一次update操作，update max\_id= max\_id + step，update成功则说明新号段获取成功，新的号段范围是(max\_id ,max\_id +step]。

由于依赖数据库，我们先设计一下表结构：

CREATE TABLE `leaf\_alloc` (  
  `biz\_tag` varchar(128) NOT NULL DEFAULT '' COMMENT '业务key',  
  `max\_id` bigint(20) NOT NULL DEFAULT '1' COMMENT '当前已经分配了的最大id',  
  `step` int(11) NOT NULL COMMENT '初始步长，也是动态调整的最小步长',  
  `description` varchar(256) DEFAULT NULL COMMENT '业务key的描述',  
  `update\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP COMMENT '数据库维护的更新时间',  
  PRIMARY KEY (`biz\_tag`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

预先插入一条测试的业务数据

INSERT INTO `leaf\_alloc` (`biz\_tag`, `max\_id`, `step`, `description`, `update\_time`) VALUES ('leaf-segment-test', '0', '10', '测试', '2020-02-28 10:41:03');

* biz\_tag：针对不同业务需求，用biz\_tag字段来隔离，如果以后需要扩容时，只需对biz\_tag分库分表即可
* max\_id：当前业务号段的最大值，用于计算下一个号段
* step：步长，也就是每次获取ID的数量
* description：对于业务的描述，没啥好说的

将Leaf项目下载到本地：https://github.com/Meituan-Dianping/Leaf

修改一下项目中的leaf.properties文件，添加数据库配置

leaf.name=com.sankuai.leaf.opensource.test  
leaf.segment.enable=true  
leaf.jdbc.url=jdbc:mysql://127.0.0.1:3306/xin-master?useUnicode=true&characterEncoding=utf8  
leaf.jdbc.username=junkang  
leaf.jdbc.password=junkang  
  
leaf.snowflake.enable=false

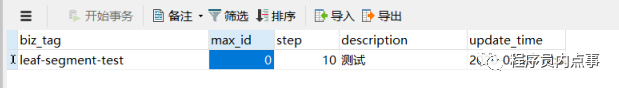
**注意**：leaf.snowflake.enable 与 leaf.segment.enable 是无法同时开启的，否则项目将无法启动。

配置相当的简单，直接启动LeafServerApplication后就OK了，接下来测试一下，leaf是基于Http请求的发号服务， LeafController 中只有两个方法，一个号段接口，一个snowflake接口，key就是数据库中预先插入的业务biz\_tag。

@RestController  
public class LeafController {  
    private Logger logger = LoggerFactory.getLogger(LeafController.class);  
  
    @Autowired  
    private SegmentService segmentService;  
    @Autowired  
    private SnowflakeService snowflakeService;  
  
    /\*\*  
     \* 号段模式  
     \* @param key  
     \* @return  
     \*/  
    @RequestMapping(value = "/api/segment/get/{key}")  
    public String getSegmentId(@PathVariable("key") String key) {  
        return get(key, segmentService.getId(key));  
    }  
  
    /\*\*  
     \* 雪花算法模式  
     \* @param key  
     \* @return  
     \*/  
    @RequestMapping(value = "/api/snowflake/get/{key}")  
    public String getSnowflakeId(@PathVariable("key") String key) {  
        return get(key, snowflakeService.getId(key));  
    }  
  
    private String get(@PathVariable("key") String key, Result id) {  
        Result result;  
        if (key == null || key.isEmpty()) {  
            throw new NoKeyException();  
        }  
        result = id;  
        if (result.getStatus().equals(Status.EXCEPTION)) {  
            throw new LeafServerException(result.toString());  
        }  
        return String.valueOf(result.getId());  
    }  
}

访问：http://127.0.0.1:8080/api/segment/get/leaf-segment-test，结果正常返回，感觉没毛病，但当查了一下数据库表中数据时发现了一个问题。





通常在用号段模式的时候，取号段的时机是在前一个号段消耗完的时候进行的，可刚刚才取了一个ID，数据库中却已经更新了max\_id，也就是说leaf已经多获取了一个号段，这是什么鬼操作？

**Leaf为啥要这么设计呢？**

Leaf 希望能在DB中取号段的过程中做到无阻塞！

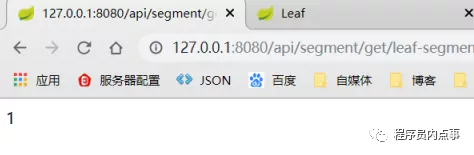
当号段耗尽时再去DB中取下一个号段，如果此时网络发生抖动，或者DB发生慢查询，业务系统拿不到号段，就会导致整个系统的响应时间变慢，对流量巨大的业务，这是不可容忍的。

所以Leaf在当前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。这样做很大程度上的降低了系统的风险。

**那么某个点到底是什么时候呢？**

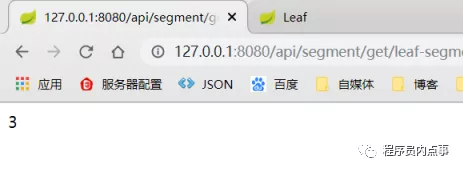
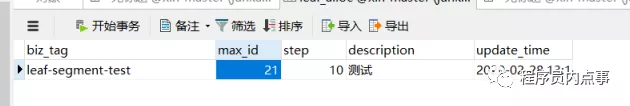
这里做了一个实验，号段设置长度为step=10，max\_id=1

当我拿第一个ID时，看到号段增加了，1/10



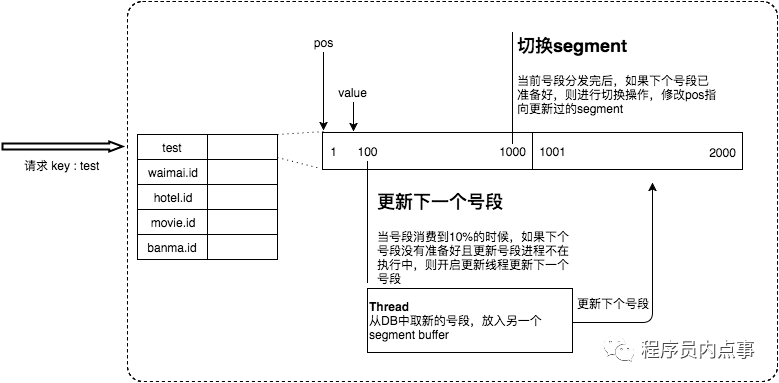


当我拿第三个Id时，看到号段又增加了，3/10



Leaf采用双buffer的方式，它的服务内部有两个号段缓存区segment。当前号段已消耗10%时，还没能拿到下一个号段，则会另启一个更新线程去更新下一个号段。

简而言之就是Leaf保证了总是会多缓存两个号段，即便哪一时刻数据库挂了，也会保证发号服务可以正常工作一段时间。



通常推荐号段（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。

**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。
* 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。

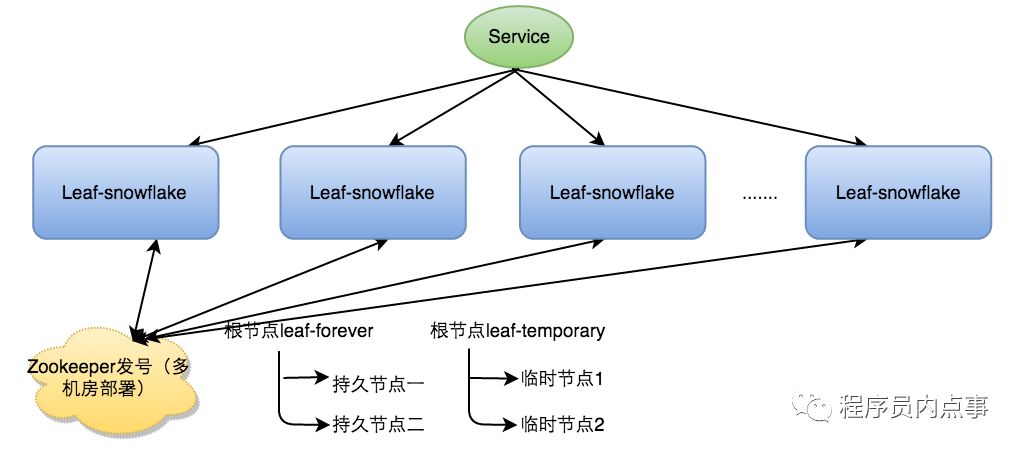
**缺点：**

* ID号码不够随机，能够泄露发号数量的信息，不太安全。
* DB宕机会造成整个系统不可用（用到数据库的都有可能）。

**二、Leaf-snowflake**

Leaf-snowflake基本上就是沿用了snowflake的设计，ID组成结构：正数位（占1比特）+ 时间戳（占41比特）+ 机器ID（占5比特）+ 机房ID（占5比特）+ 自增值（占12比特），总共64比特组成的一个Long类型。

Leaf-snowflake不同于原始snowflake算法地方，主要是在workId的生成上，Leaf-snowflake依靠Zookeeper生成workId，也就是上边的机器ID（占5比特）+ 机房ID（占5比特）。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，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。



Leaf-snowflake启动服务的过程大致如下：

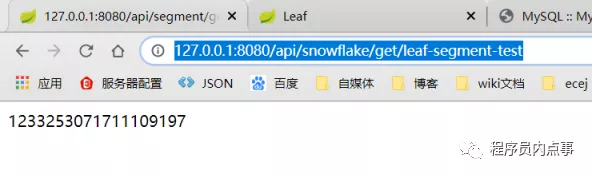
* 启动Leaf-snowflake服务，连接Zookeeper，在leaf\_forever父节点下检查自己是否已经注册过（是否有该顺序子节点）。
* 如果有注册过直接取回自己的workerID（zk顺序节点生成的int类型ID号），启动服务。
* 如果没有注册过，就在该父节点下面创建一个持久顺序节点，创建成功后取回顺序号当做自己的workerID号，启动服务。

但Leaf-snowflake对Zookeeper是一种弱依赖关系，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。一旦ZooKeeper出现问题，恰好机器出现故障需重启时，依然能够保证服务正常启动。

启动Leaf-snowflake模式也比较简单，起动本地ZooKeeper，修改一下项目中的leaf.properties文件，关闭leaf.segment模式，启用leaf.snowflake模式即可。

leaf.segment.enable=false  
#leaf.jdbc.url=jdbc:mysql://127.0.0.1:3306/xin-master?useUnicode=true&characterEncoding=utf8  
#leaf.jdbc.username=junkang  
#leaf.jdbc.password=junkang  
  
leaf.snowflake.enable=true  
leaf.snowflake.zk.address=127.0.0.1  
leaf.snowflake.port=2181

    /\*\*  
     \* 雪花算法模式  
     \* @param key  
     \* @return  
     \*/  
    @RequestMapping(value = "/api/snowflake/get/{key}")  
    public String getSnowflakeId(@PathVariable("key") String key) {  
        return get(key, snowflakeService.getId(key));  
    }

测试一下，访问：http://127.0.0.1:8080/api/snowflake/get/leaf-segment-test

**优点：**

* 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。

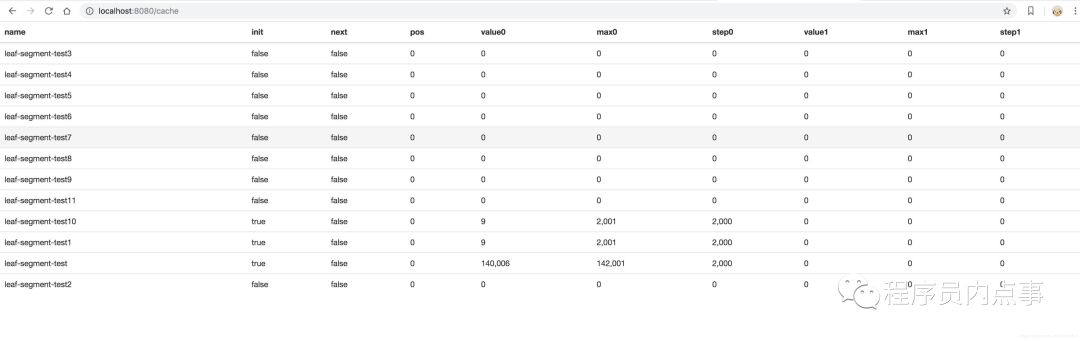
**缺点：**

* 依赖ZooKeeper，存在服务不可用风险（实在不知道有啥缺点了）

**三、Leaf监控**

请求地址：http://127.0.0.1:8080/cache

针对服务自身的监控，Leaf提供了Web层的内存数据映射界面，可以实时看到所有号段的下发状态。比如每个号段双buffer的使用情况，当前ID下发到了哪个位置等信息都可以在Web界面上查看。



**总结**

对于Leaf具体使用哪种模式，还是根据具体的业务场景使用，本文并没有对Leaf源码做过多的分析，因为Leaf 代码量简洁很好阅读。后续还会把其他几种分布式ID生成器，依次结合实战介绍给大家，欢迎大家关注。

<https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247490422&idx=2&sn=f9ba426d79c355350bcc0aa275de62f8&chksm=fbb29288ccc51b9e38b6d72493de78ea2b018b8f78c50a9e7ba72f463fbfa5d72c0dd6736679&mpshare=1&scene=1&srcid=0809irqMOrEDVT4U9uUExRhc&sharer_sharetime=1596950104597&sharer_shareid=6409037e4f38f0936cfb68aeee2b74f0&key=82003248020682fadf3a06f57fd16b42e40b5c7873522ac0eb5594f00fd7ee1b1799d129a9be3ba5dfd03c3dd0fe8f86ad62323a97652c0131b7138864d8d161b88f169a0f027426fa2ec83fc1f7e9ca&ascene=1&uin=NDM2MDg2NzU1&devicetype=Windows+10+x64&version=62090529&lang=zh_CN&exportkey=AZO5ZHEmnyl13G5D8uo1kAQ%3D&pass_ticket=sjiKuh5Fgot5S95NPHxTTNxBKTX6ks%2FhqHm%2Fb2XQcAMae1fZflBdguxf%2BCnlP8sr>