业界实现方案抄的思想变为借鉴

1.基于UUID

2.基于DB数据库多种模式(自增主键、segment)

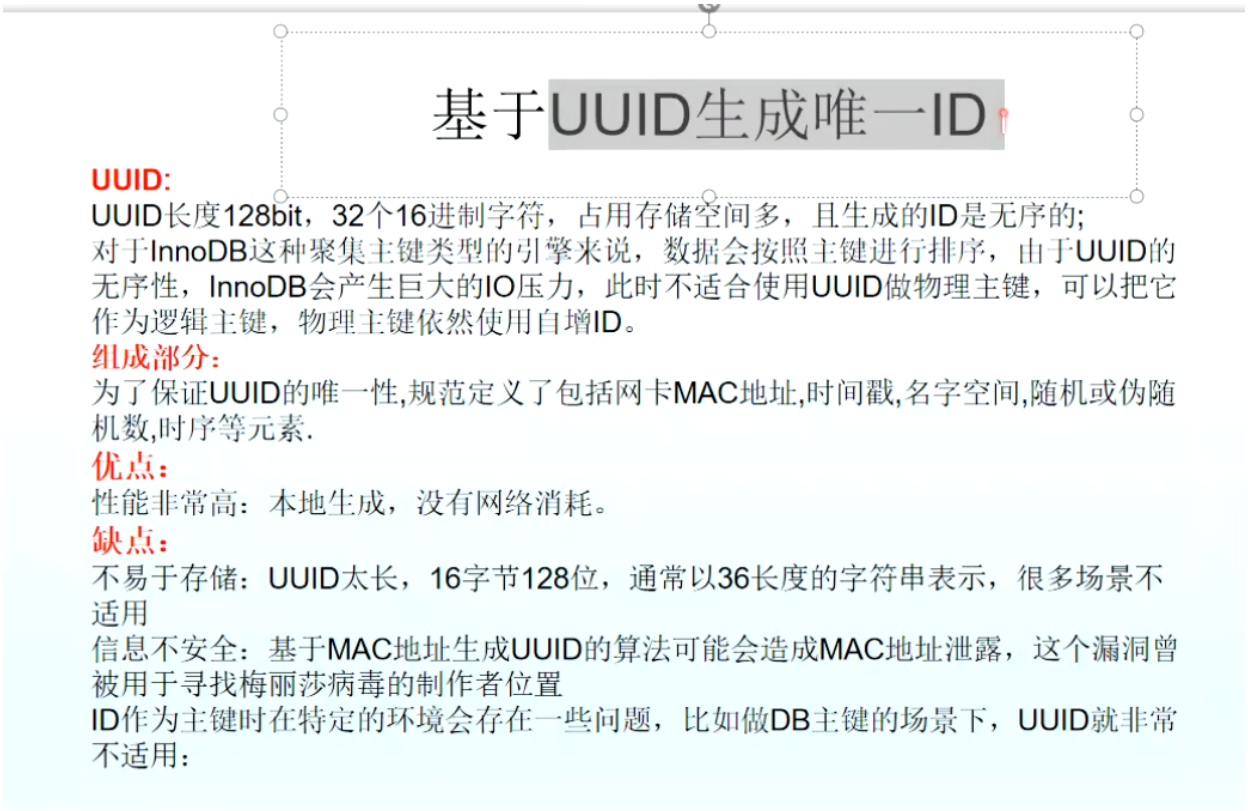
3.基于Redis

4.基于ZK、ETCD

5.基于SnowFlake

6.美团Leaf(DB-Segment、zk+SnowFlake)

7.百度uid-generator()



基于DB的自增主键方案

实现原理:
基于MySQL，最简单的方法是使用auto_increment 来生成全局唯一递增ID，但最致命的问题是在高并发情况下，数据库压力大，DB单点存在宕机风险

优点:
实现简单、基于数据库底层机制

缺点:
高并发情况下，数据库压力大，DB单点存在宕机风险

```
CREATE TABLE `seq_id` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `value` varchar(1) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=821 DEFAULT CHARSET=utf8;
```

基于DB号段实现方案

实现原理:
每次向db申请一个号段，加载到内存中，然后采用自增的方式来生成id，这个号段用完后，再次向db申请一个新的号段，这样对db的压力就减轻了很多，同时内存中直接生成id。向数据库申请新号段，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]。

优点:
利用了缓存，减轻DB压力，性能提升

缺点:
依然存在DB模式下的性能瓶颈，ID最大值的限制

```
CREATE TABLE `id_generator` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `max_id` bigint(20) NOT NULL COMMENT '当前最大id',
  `step` bigint(20) NOT NULL COMMENT '号段的布长',
  `biz_code` varchar(16) NOT NULL COMMENT '业务类型编码',
  PRIMARY KEY (`id`),
  UNIQUE KEY `biz_code` (`biz_code`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

基于DB号段简易版测试用例

```java
@Test
public void testSegment() throws InterruptedException {
    ConcurrentHashMap<String,String> idMap = new ConcurrentHashMap<>();
    CountDownLatch countDownLatch = new CountDownLatch(THREADS);
    MysqlNumberSegmentUIDGenerator numberSegmentUIDGenerator =
            new MysqlNumberSegmentUIDGenerator(dataSource,bizCode);
    List<String> list = new ArrayList<>();
    for(int i=0;i<THREADS;i++){
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                countDownLatch.countDown();
                String uuid = numberSegmentUIDGenerator.generateUId();
                System.out.println(Thread.currentThread().getName() + ":" + uuid);
                idMap.put(uuid,"1");
            }
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    executorService.awaitTermination( timeout: 60, TimeUnit.SECONDS);
    System.out.println(idMap.size());
}
```

```
pool-1-thread-35:90
pool-1-thread-39:91
pool-1-thread-43:92
pool-1-thread-42:93
pool-1-thread-16:94
pool-1-thread-44:95
pool-1-thread-45:96
pool-1-thread-48:97
pool-1-thread-46:98
pool-1-thread-47:99
100
```

基于Redis实现分布式ID

因为Redis是单线程的，所以天然没有资源争用问题，可以采用 incr 指令，实现ID的原子性自增。
但是因为Redis的数据备份-RDB，会存在漏掉数据的可能，所以理论上存在已使用的ID再次被使用，所以备份方式可以加上AOF方式，这样的话性能会有所损耗。

Master集群+slave　异步复制

INCR key

将 key 中储存的数字值增一。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

这是一个针对字符串的操作，因为 Redis 没有专用的整数类型，所以 key 内储存的字符串被解释为十进制 64 位有符号整数来执行 INCR 操作。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

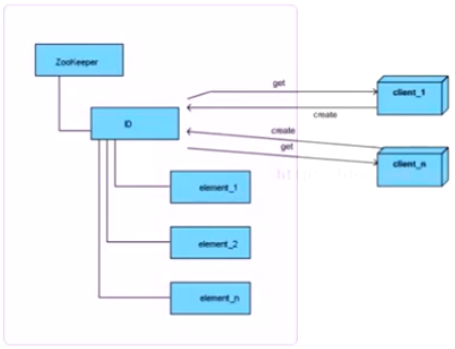
执行 INCR 命令之后 key 的值。

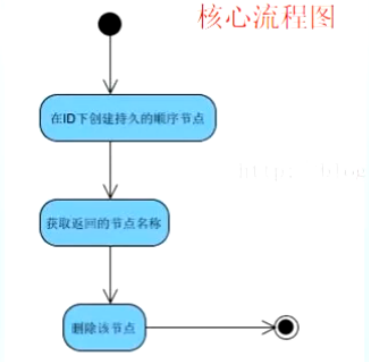
```
redis> SET page_view 20
OK

redis> INCR page_view
(integer) 21

redis> GET page_view    # 数字值在 Redis 中以字符串的形式保存
"21"
```

基于Zookeeper实现分布式ID

原理:

利用zookeeper中的顺序节点的特性,制作分布式的序列号生成器(ID生成器)





http://www.mashibing.com

测试案例:

```
private static RetryPolicy retryPolicy;
private static Executor executor;
private static String ip =  ip:port                ";
private static String ROOT = "/root";
private static String NODE_NAME = "idGenerator";
public ZookeeperGenerater(String bizCode) { super(bizCode); }
static {
    retryPolicy = new ExponentialBackoffRetry( baseSleepTimeMs: 1000,  maxRetries: 3);
    curatorFrameworkClient = CuratorFrameworkFactory
            .builder()
            .connectString(ip)
            .sessionTimeoutMs(5000)
            .connectionTimeoutMs(5000)
            .retryPolicy(retryPolicy)
            .build();
    curatorFrameworkClient.start();
    try {
        Stat stat = curatorFrameworkClient.checkExists().forPath(ROOT);
        if (stat == null) {
            curatorFrameworkClient.create().withMode(CreateMode.PERSISTENT).forPath(ROOT,  bytes: null);
```

基于ETCD实现分布式ID

原理:
每个tx事务有唯一事务ID，在etcd中叫做main ID，全局递增不重复。
一个tx可以包含多个修改操作（put和delete），每一个操作叫做一个revision（修订），共享同一个main ID。
一个tx内连续的多个修改操作会被从0递增编号，这个编号叫做sub ID。
每个revision由（main ID，sub ID）唯一标识。

Revision的定义:

```
// A revision indicates modification of the key-value space.
// The set of changes that share same main revision changes the key-
value space atomically.
type revision struct
{
// main is the main revision of a set of changes that happen atomically. main int64

// sub is the the sub revision of a change in a set of changes that happen
// atomically. Each change has different increasing sub revision in that // set. sub int64 }
```

测试案例

```
public String generateUId() {
    KV kvClient = client.getKVClient();
    String id = null;
    try {
        PutResponse putResponse = kvClient.put(ByteSequence.from("key".getBytes()), ByteSequence.from("value".getBytes())).get();
        Response.Header header = putResponse.getHeader();
        long revision = header.getRevision();
        id = revision+"";
    }catch (Exception e){
        e.printStackTrace();
    }
    return id;
```

测试案例

```
private static String server = "http:/  ip:port  ";
@Test
public void test() throws InterruptedException {
    Client client = Client.builder().endpoints(server).build();
    for (int i = 0; i < THREADS; i++) {
        executorService.execute(()->{
            EtcdGenerater etcd = new EtcdGenerater(client);
            String s = etcd.generateUId();
            System.out.println(s);
        });
    }
    executorService.shutdown();
    executorService.awaitTermination( timeout: 60, TimeUnit.SECONDS);
```

美团Leaf-基于DB的Segment模式

实现原理:

- 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
- 内存实现基于双Buffer实现性能提升

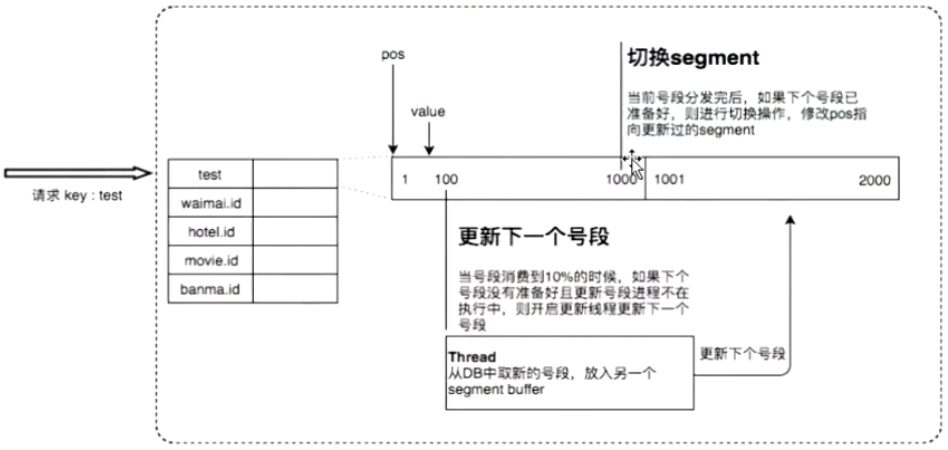
优点:

- 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。
- DB宕机会造成整个系统不可用

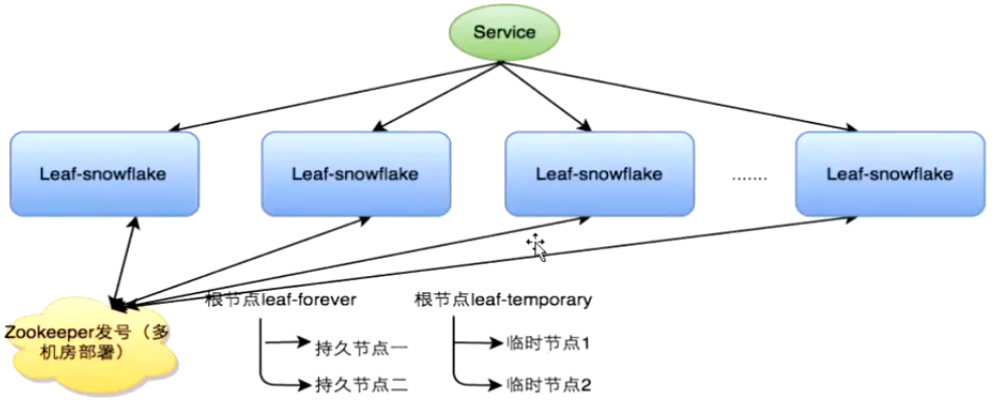
美团Leaf-双Buffer优化



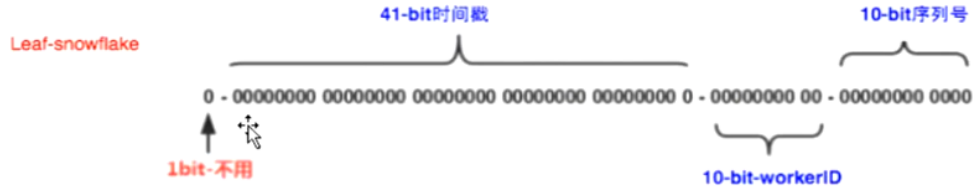
image

美团Leaf-基于ZK的SnowFlake算法

官方设计文档地址:https://tech.meituan.com/2017/04/21/mt-leaf.html



美团Leaf-SnowFlake的 ID格式



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

百度uid-generator分布式ID生成器

UidGenerator是Java实现的, 基于Snowflake算法的唯一ID生成器。
UidGenerator以组件形式工作在应用项目中, 支持自定义workerId位数和初始化策略
从而适用于docker等虚拟化环境下实例自动重启、漂移等场景。
在实现上, UidGenerator通过借用未来时间来解决sequence天然存在的并发限制;
采用RingBuffer来缓存已生成的UID, 并行化UID的生产和消费, 同时对CacheLine补
避免了由RingBuffer带来的硬件级「伪共享」问题. 最终单机QPS可达600万。

其实现原理和雪花算法并无二致，自定义号段，并且采用RingBuffer作为缓冲
从而提升性能。详见官网地址：
https://github.com/baidu/uidgenerator/blob/master/README.zh_cn.md