

第39讲 | 谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响？

2018-08-07 杨晓峰

Java核心技术36讲

[进入课程 >](#)



讲述：黄洲君

时长 10:03 大小 4.60M



专栏的绝大部分主题都侧重于 Java 语言和虚拟机，基本都是单机模式下的问题，今天我会补充一个分布式相关的问题。严格来说，分布式并不算是 Java 领域，而是一个单独的大主题，但确实也会在 Java 技术岗位面试中被涉及。在准备面试时，如果有丰富的分布式系统经验当然好；如果没有，你可以选择典型问题和基础技术进行适当准备。关于分布式，我自身的实战经验也非常有限，专栏里就谈谈从理论出发的一些思考。

今天我要问你的问题是，**谈谈常用的分布式 ID 的设计方案？Snowflake 是否受冬令时切换影响？**

典型回答

首先，我们需要明确通常的分布式 ID 定义，基本的要求包括：

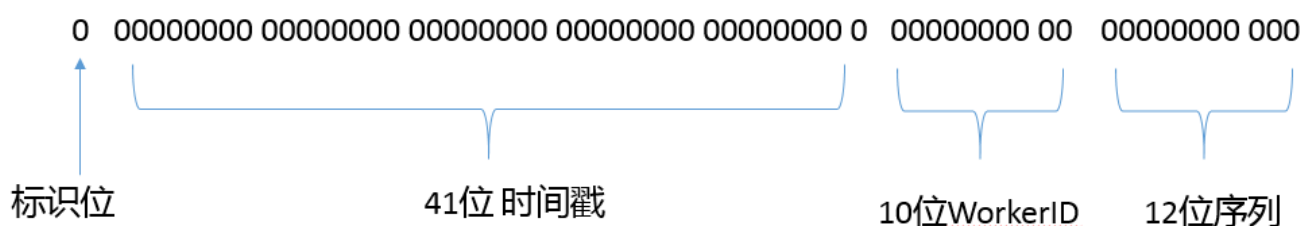
全局唯一，区别于单点系统的唯一，全局是要求分布式系统内唯一。

有序性，通常都需要保证生成的 ID 是有序递增的。例如，在数据库存储等场景中，有序 ID 便于确定数据位置，往往更加高效。

目前业界的方案很多，典型方案包括：

基于数据库自增序列的实现。这种方式优缺点都非常明显，好处是简单易用，但是在扩展性和可靠性等方面存在局限性。

基于 Twitter 早期开源的 [Snowflake](#) 的实现，以及相关改动方案。这是目前应用相对比较广泛的一种方式，其结构定义你可以参考下面的示意图。



整体长度通常是 64 ($1 + 41 + 10 + 12 = 64$) 位，适合使用 Java 语言中的 long 类型来存储。

头部是 1 位的正负标识位。

紧跟着的高位部分包含 41 位时间戳，通常使用 `System.currentTimeMillis()`。

后面是 10 位的 WorkerID，标准定义是 5 位数据中心 + 5 位机器 ID，组成了机器编号，以区分不同的集群节点。

最后的 12 位就是单位毫秒内可生成的序列号数目的理论极限。

Snowflake 的 [官方版本](#) 是基于 Scala 语言，Java 等其他语言的 [参考实现](#) 有很多，是一种非常简单实用的方式，具体位数的定义是可以根据分布式系统的真实场景进行修改的，并不一

定要严格按照示意图中的设计。

Redis、Zookeeper、MongoDB 等中间件，也都有各种唯一 ID 解决方案。其中一些设计也可以算是 Snowflake 方案的变种。例如，MongoDB 的 [ObjectId](#) 提供了一个 12 byte (96 位) 的 ID 定义，其中 32 位用于记录以秒为单位的时间，机器 ID 则为 24 位，16 位用作进程 ID，24 位随机起始的计数序列。

国内的一些大厂开源了其自身的部分分布式 ID 实现，InfoQ 就曾经介绍过微信的 [seqsvr](#)，它采取了相对复杂的两层架构，并根据社交应用的数据特点进行了针对性设计，具体请参考相关[代码实现](#)。另外，[百度](#)、美团等也都有开源或者分享了不同的分布式 ID 实现，都可以进行参考。

关于第二个问题，**Snowflake 是否受冬令时切换影响？**

我认为没有影响，你可以从 Snowflake 的具体算法实现寻找答案。我们知道 Snowflake 算法的 Java 实现，大都是依赖于 `System.currentTimeMillis()`，这个数值代表什么呢？从 Javadoc 可以看出，它是返回当前时间和 1970 年 1 月 1 号 UTC 时间相差的毫秒数，这个数值与夏 / 冬令时并没有关系，所以并不受其影响。

考点分析

今天的问题不仅源自面试的热门考点，并且也存在着广泛的应用场景，我前面给出的回答只是一个比较精简的典型方案介绍。我建议你针对特定的方案进行深入分析，以保证在面试官可能会深入追问时能有充分准备；如果恰好在现有系统使用分布式 ID，理解其设计细节是很有必要的。

涉及分布式，很多单机模式下的简单问题突然就变得复杂了，这是分布式天然的复杂性，需从不同角度去理解适用场景、架构和细节算法，我会从下面的角度进行适当解读：

我们的业务到底需要什么样的分布式 ID，除了唯一和有序，还有哪些必须要考虑的因素？

在实际场景中，针对典型的方案，有哪些可能的局限性或者问题，可以采取什么办法解决呢？

知识扩展

如果试图深入回答这个问题，首先需要明确业务场景的需求要点，我们到底需要一个什么样的分布式 ID？

除了唯一和有序，考虑到分布式系统的功能需要，通常还会额外希望分布式 ID 保证：

有意义，或者说包含更多信息，例如时间、业务等信息。这一点和有序性要求存在一定关联，如果 ID 中包含时间，本身就能保证一定程度的有序，虽然并不能绝对保证。ID 中包含额外信息，在分布式数据存储等场合中，有助于进一步优化数据访问的效率。

高可用性，这是分布式系统的必然要求。前面谈到的方案中，有的是真正意义上的分布式，有得还是传统主从的思路，这一点没有绝对的对错，取决于我们业务对扩展性、性能等方面的要求。

紧凑性，ID 的大小可能受到实际应用的制约，例如数据库存储往往对长 ID 不友好，太长的 ID 会降低 MySQL 等数据库索引的性能；编程语言在处理时也可能受数据类型长度限制。

在具体的生产环境中，还有可能提出对 QPS 等方面的具体要求，尤其是在国内一线互联网公司的业务规模下，更是需要考虑峰值业务场景的数量级层次需求。

第二，主流方案的优缺点分析。

对于数据库自增方案，除了实现简单，它生成的 ID 还能够保证固定步长的递增，使用很方便。

但是，因为每获取一个 ID 就会触发数据库的写请求，是一个代价高昂的操作，构建高扩展性、高性能解决方案比较复杂，性能上限明显，更不要谈扩容等场景的难度了。与此同时，保证数据库方案的高可用性也存在挑战，数据库可能发生宕机，即使采取主从热备等各种措施，也可能出现 ID 重复等问题。

实际大厂商往往是构建了多层的复合架构，例如美团公开的数据库方案[Leaf-Segment](#)，引入了起到缓存等作用的 Leaf 层，对数据库操作则是通过数据库中间件提供的批量操作，这样既能保证性能、扩展性，也能保证高可用。但是，这种方案对基础架构层面的要求很多，未必适合普通业务规模的需求。

与其相比，Snowflake 方案的好处是算法简单，依赖也非常少，生成的序列可预测，性能也非常好，比如 Twitter 的峰值超过 10 万 /s。

但是，它也存在一定的不足，例如：

时钟偏斜问题（Clock Skew）。我们知道普通的计算机系统时钟并不能保证长久的一致性，可能发生时钟回拨等问题，这就会导致时间戳不准确，进而产生重复 ID。

针对这一点，Twitter 曾经在文档中建议开启 [NTP](#)，毕竟 Snowflake 对时间存在依赖，但是也有人提议关闭 NTP。我个人认为还是应该开启 NTP，只是可以考虑将 stepback 设置为 0，以禁止回调。

从设计和具体编码的角度，还有一个很有效的措施就是缓存历史时间戳，然后在序列生成之前进行检验，如果出现当前时间落后于历史时间的不合理情况，可以采取相应的动作，要么重试、等待时钟重新一致，或者就直接提示服务不可用。

另外，序列号的可预测性是把双刃剑，虽然简化了一些工程问题，但很多业务场景并不适合可预测的 ID。如果你用它作为安全令牌之类，则是非常危险的，很容易被黑客猜测并利用。

ID 设计阶段需要谨慎考虑暴露出的信息。例如，[Erlang 版本](#)的 flake 实现基于 MAC 地址计算 WorkerID，在安全敏感的领域往往是不可以这样使用的。

从理论上来说，类似 Snowflake 的方案由于时间数据位数的限制，存在与[2038 年问题](#)相似的理论极限。虽然目前的系统设计考虑数十年后的问题还太早，但是理解这些可能的极限是有必要的，也许会成为面试的过程中的考察点。

如果更加深入到时钟和分布式系统时序的问题，还有与分布式 ID 相关但又有所区别的问题，比如在分布式系统中，不同机器的时间很可能是不一致的，如何保证事件的有序性？Lamport 在 1978 年的论文（[Time, Clocks, and the Ordering of Events in a Distributed System](#)）中就有很深入的阐述，有兴趣的同学可以去查找相应的翻译和解读。

最后，我再补充一些当前分布式领域的面试热点，例如：

分布式事务，包括其产生原因、业务背景、主流的解决方案等。

理解[CAP](#)、[BASE](#)等理论，懂得从最终一致性等角度来思考问题，理解[Paxos](#)、[Raft](#)等一致性算法。

理解典型的分布式锁实现，例如最常见的[Redis 分布式锁](#)。

负载均衡等分布式领域的典型算法，至少要了解主要方案的原理。

这些方面目前都已经有了相对比较深入的分析，尤其是来自于一线大厂的实践经验。另外，在[左耳听风专栏的“程序员练级攻略”](#)里，提供了非常全面的分布式学习资料，感兴趣的同学可以参考。

今天我简要梳理了当前典型的分布式 ID 生成方案，并探讨了 ID 设计的一些考量，尤其是应用相对广泛的 Snowflake 的不足之处，希望对你有所帮助。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，从理论上来看，Snowflake 这种基于时间的算法，从形式上天然地限制了 ID 的并发生成数量，如果在极端情况下，短时间需要更多 ID，有什么办法解决呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Java 核心技术 36 讲

—— 前 Oracle 首席工程师
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

上一篇 第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？

下一篇 结束语 | 技术没有终点

精选留言 (13)

写留言



董朱明

2018-08-10

23

69年的极限问题不难解决，timestamp减个常量就可以了，对于已生成的历史id，可以导表刷id，当然，这里涉及到个数据库设计原则，系统之间传递数据不应使用物理主键，这样刷id 就容易了

作者回复: 高手



卡特

2018-09-12

5

因为snowflake的可预测性，可以提前生成好放到队列里，获取的时候直接获取。相当于做了一层缓存；
理论上可以解决短时间大量获取id的需求；



黄琨

2018-08-07

5

缩减workID长度，增加序列号长度



影子

2019-01-24

2

生成32位的自增长Id(int)老师有什么思路嘛



RoverYe

2018-09-15

2

我们这边利用zk的唯一id特性



袁伟

2018-08-08

👍 2

一直有个疑问就是Snowflake 文中说的极限问题，目前确定它只能用69年，大家都用数据库的数字类型来存储，那么到了69年之后，后来人怎么处理，也许那个时候有更大数字来表示。但我还是想不出更合理的方式，也许我想多了，这个问题交给69年后的人来考虑，但我也想知道老师您是如何思考这个问题的

展开 ▾



XiaoYeGe

2018-11-30

👍 1

前后历时半年多 终于看完了, 下面就是再回头巩固一遍!



安小依

2018-08-08

👍 1

老师自己有没有计划，针对分布式单独出一个专栏，一直以来自己都想研究分布式，但是很多问题依旧搞不懂: zookeeper 选举过程、hdfs 存储出现故障namenode是怎么处理、MapReduce 作业调度问题需要做哪些权衡，不同异常下应该怎么解决，是忽略错误，还是直接退出...各个方案背后是什么样的利弊在协调着这些...

展开 ▾

作者回复: 术业有专攻，有特定专家出专栏



吴科

2018-12-29

👍

我们通常用UUID或者数据自增主键，这样的方式效率都不高。snowflake分布式全局生成的ID，效率高，设计优雅。



airong

2018-12-28

👍

请问大牛10集群编号怎么获取的啊



Yang.

2018-09-06

👍

使用System.currentTimeMillis的话不是存在时钟回拨问题么，能不能从网络获取时间，去生成这个🐼🐼



涛哥

2018-08-07



能讲下时钟偏斜和时钟回拨吗，不是理解



有铭

2018-08-07



为啥最后一段是12的长度而不是别的数

作者回复: 我提到了，各部分不是固定的，看业务需求，例如，集群小，位数可以设计短点儿，seq就可以更多位，时间也未必非要41位

