浩鲸科技：为什么要用雪花ID替代数据库自增ID?

今天咱们来看一道数据库中比较经典的面试问题：为什么要使用雪花 ID 替代数据库自增 ID？同时这道题也出现在了浩鲸科技的 Java 面试中，下面我们一起来看吧。

浩鲸科技的面试题如下：

其他面试题相对来说比较简单，大部人题目都可以在我的网站上（[www.javacn.site）找到答案，这里就不再赘述，咱们今天只聊“为什么要使用雪花](http://cxyroad.com/”http://www.javacn.site%EF%BC%89%E6%89%BE%E5%88%B0%E7%AD%94%E6%A1%88%EF%BC%8C%E8%BF%99%E9%87%8C%E5%B0%B1%E4%B8%8D%E5%86%8D%E8%B5%98%E8%BF%B0%EF%BC%8C%E5%92%B1%E4%BB%AC%E4%BB%8A%E5%A4%A9%E5%8F%AA%E8%81%8A%E2%80%9C%E4%B8%BA%E4%BB%80%E4%B9%88%E8%A6%81%E4%BD%BF%E7%94%A8%E9%9B%AA%E8%8A%B1”) ID 替代数据库自增 ID？”这个问题。

1.什么是雪花 ID?

雪花 ID（Snowflake ID）是一个用于分布式系统中生成唯一 ID 的算法，由 Twitter 公司提出。它的设计目标是在分布式环境下高效地生成全局唯一的 ID，具有一定的有序性。

雪花 ID 的结构如下所示：

这四部分代表的含义

1. **符号位**：最高位是符号位，始终为 0，1 表示负数，0 表示正数，ID 都是正整数，所以固定为 0。
2. **时间戳部分**：由 41 位组成，精确到毫秒级。可以使用该 41 位表示的

时间戳来表示的时间可以使用 69 年。
3. **节点 ID 部分**：由 10 位组成，用于表示机器节点的唯一标识符。在同一毫秒内，不同的节点生成的 ID 会有所不同。
4. **序列号部分**：由 12 位组成，用于标识同一毫秒内生成的不同 ID 序列。在同一毫秒内，可以生成 4096 个不同的 ID。

2.Java 版雪花算法实现

接下来，我们来实现一个 Java 版的雪花算法：

```
public class SnowflakeIdGenerator {

  // 定义雪花 ID 的各部分位数
  private static final long TIMESTAMP_BITS = 41L;
  private static final long NODE_ID_BITS = 10L;
  private static final long SEQUENCE_BITS = 12L;

  // 定义起始时间戳（可根据实际情况调整）
  private static final long EPOCH = 1609459200000L;

  // 定义最大取值范围
  private static final long MAX_NODE_ID = (1L << NODE_ID_BITS) - 1;
  private static final long MAX_SEQUENCE = (1L << SEQUENCE_BITS) -
1;

  // 定义偏移量
  private static final long TIMESTAMP_SHIFT = NODE_ID_BITS +
SEQUENCE_BITS;
  private static final long NODE_ID_SHIFT = SEQUENCE_BITS;

  private final long nodeId;
  private long lastTimestamp = -1L;
  private long sequence = 0L;

  public SnowflakeIdGenerator(long nodeId) {
    if (nodeId < 0 || nodeId > MAX_NODE_ID) {
      throw new IllegalArgumentException("Invalid node ID");
    }
    this.nodeId = nodeId;
  }

  public synchronized long generateId() {
```

```
    long currentTimestamp = timestamp();
    if (currentTimestamp < lastTimestamp) {
        throw new IllegalStateException("Clock moved backwards");
    }
    if (currentTimestamp == lastTimestamp) {
      sequence = (sequence + 1) & MAX_SEQUENCE;
      if (sequence == 0) {
        currentTimestamp = untilNextMillis(lastTimestamp);
      }
    } else {
      sequence = 0L;
    }
    lastTimestamp = currentTimestamp;
    return ((currentTimestamp - EPOCH) << TIMESTAMP_SHIFT) |
           (nodeId << NODE_ID_SHIFT) |
           sequence;
  }

  private long timestamp() {
    return System.currentTimeMillis();
  }

  private long untilNextMillis(long lastTimestamp) {
    long currentTimestamp = timestamp();
    while (currentTimestamp <= lastTimestamp) {
      currentTimestamp = timestamp();
    }
    return currentTimestamp;
  }
}

```

调用代码如下：

```
public class Main {
  public static void main(String[] args) {
    // 创建一个雪花 ID 生成器实例，传入节点 ID
    SnowflakeIdGenerator idGenerator = new SnowflakeIdGenerator(1);
    // 生成 ID
    long id = idGenerator.generateId();
    System.out.println(id);
  }
}
```

```

其中，nodeId 表示当前节点的唯一标识，可以根据实际情况进行设置。generateId 方法用于生成雪花 ID，采用同步方式确保线程安全。具体的生成逻辑遵循雪花 ID 的位运算规则，结合当前时间戳、节点 ID 和序列号生成唯一的 ID。

> 需要注意的是，示例中的时间戳获取方法使用了 System.currentTimeMillis()，根据实际需要可以替换为其他更精确的时间戳获取方式。同时，需要确保节点 ID 的唯一性，避免不同节点生成的 ID 重复。

3.雪花算法问题
---

虽然雪花算法是一种被广泛采用的分布式唯一 ID 生成算法，但它也存在以下几个问题：

1. **时间回拨问题**：雪花算法生成的 ID 依赖于系统的时间戳，要求系统的时钟必须是单调递增的。如果系统的时钟发生回拨，可能导致生成的 ID 重复。时间回拨是指系统的时钟在某个时间点之后突然往回走（人为设置），即出现了时间上的逆流情况。
2. **时钟回拨带来的可用性和性能问题**：由于时间依赖性，当系统时钟发生回拨时，雪花算法需要进行额外的处理，如等待系统时钟追上上一次生成 ID 的时间戳或抛出异常。这种处理会对算法的可用性和性能产生一定影响。
3. **节点 ID 依赖问题**：雪花算法需要为每个节点分配唯一的节点 ID 来保证生成的 ID 的全局唯一性。节点 ID 的分配需要有一定的管理和调度，特别是在动态扩容或缩容时，节点 ID 的管理可能较为复杂。

4.如何解决时间回拨问题?
---

百度 UidGenerator 框架中解决了时间回拨的问题，并且解决方案比较经典，所以咱们这里就来给大家分享一下百度 UidGenerator 是怎么解决时间回拨问题的?

> **UidGenerator 介绍**：UidGenerator 是百度开源的一个分布式唯一 ID 生成器，它是基于 Snowflake 算法的改进版本。与传统的 Snowflake 算法相比，UidGenerator 在高并发场景下具有更好的性能和可用性。它的实现源码在：[github.com/baidu/uid-g...](http://cxyroad.com/
```

”https://github.com/baidu/uid-generator”)

UidGenerator 是这样解决时间回拨问题的：UidGenerator 的每个实例中，都维护一个本地时钟缓存，用于记录当前时间戳。这个本地时钟会定期与系统时钟进行同步，如果检测到系统时钟往前走了（出现了时钟回拨），则将本地时钟调整为系统时钟。

4.为什么要使用雪花 ID 替代数据库自增 ID?

数据库自增 ID 只适用于单机环境，但如果是分布式环境，是将数据库进行分库、分表或数据库分片等操作时，那么数据库自增 ID 就有问题了。

例如，数据库分片之后，会在同一张业务表的分片数据库中产生相同 ID（数据库自增 ID 是由每个数据库单独记录和增加的），这样就会导致，同一个业务表的竟然有相同的 ID，而且相同 ID 背后存储的数据又完全不同，这样业务查询的时候就出问题了。

所以为了解决这个问题，就必须使用分布式中能保证唯一性的雪花 ID 来替代数据库的自增 ID。

5.扩展：使用 UUID 替代雪花 ID 行不行?

如果单从唯一性来考虑的话，那么 UUID 和雪花 ID 的效果是一致的，二者都能保证分布式系统下的数据唯一性，但是即使这样，也**不建议使用 UUID 替代雪花 ID**，因为这样做的问题有以下两个：

1. **可读性问题**：UUID 内容很长，但没有业务含义，就是一堆看不懂的“字母”。
2. **性能问题**：UUID 是字符串类型，而字符串类型在数据库的查询中效率很低。

所以，基于以上两个原因，不建议使用 UUID 来替代雪花 ID。

小结
--

数据库自增 ID 只适用于单机数据库环境，而对于分库、分表、数据分片来说，自增 ID 不具备唯一性，所以要要使用雪花 ID 来替代数据库自增 ID。但雪花算法依然存在一些问题，例如时间回拨问题、节点过度依赖问题等，所以此时，可以使用雪花算法的改进框架，如百度的 UidGenerator 来作为数据库的 ID 生成方案会比较好。

> 本文已收录到我的面试小站 [www.javacn.site](http://cxyroad.com/”https://www.javacn.site”)，其中包含的内容有：Redis、JVM、并发、并发、MySQL、Spring、Spring MVC、Spring Boot、Spring Cloud、MyBatis、设计模式、消息队列等模块。

原文链接: https://juejin.cn/post/7307066138487521289