微信公众号福利：芋艿的后端小屋

- 0. 阅读源码葵花宝典
- 1. RocketMQ / MyCAT / Sharding-JDBC 详细中文注释源码
- 2. 您对于源码的疑问每条留言都将得到认真回复
- 3. 新的源码解析文章实时收到通知，每周六十点更新
- 4. 认真的源码交流微信群

分类

分享

Sharding-JDBC 源码分析 —— 分布式主键

2017-08-12 更新日期:2017-08-05 总阅读量:15次

文章目录

> □□□关注**微信公众号：【芋艿的后端小屋】**有福利：
>
> 1. RocketMQ / MyCAT / Sharding-JDBC **所有**源码分析文章列表
> 2. RocketMQ / MyCAT / Sharding-JDBC **中文注释源码 GitHub 地址**

分享

> 3. 您对于源码的疑问每条留言**都**将得到**认真**回复。**甚至不知道如何读源码也可以请教噢**。
> 4. **新的**源码解析文章**实时**收到通知。**每周更新一篇左右**。
> 5. **认真的**源码交流微信群。

---

1. 概述

本文分享 Sharding-JDBC **分布式主键**实现。

官方文档《分布式主键》对其介绍及使用方式介绍很完整，强烈先阅读。下面先引用下分布式主键的**实现动机**：

> 传统数据库软件开发中，主键自动生成技术是基本需求。而各大数据库对于该需求也提供了相应的支持，比如MySQL的自增键。对于MySQL而言，分库分表之后，不同表生成全局唯一的Id是非常棘手的问题。因为同一个逻辑表内的不同实际表之间的自增键是无法互相感知的，这样会造成重复Id的生成。我们当然可以通过约束表生成键的规则来达到数据的不重复，但是这需要引入额外的运维力量来解决重复性问题，并使框架缺乏扩展性。

> 目前有许多第三方解决方案可以完美解决这个问题，比如UUID等依靠特定算法自生成不重复键，或者通过引入Id生成服务等。 但也正因为这种多样性导致了Sharding-JDBC如果强依赖于任何一种方案就会限制其自身的发展。

> 基于以上的原因，最终采用了以JDBC接口来实现对于生成Id的访问，而将底层具体的Id生成实现分离出来。

> **Sharding-JDBC 正在收集使用公司名单：传送门。**
> **你的登记，会让更多人参与和使用 Sharding-JDBC。传送门**
> **Sharding-JDBC 也会因此，能够覆盖更多的业务场景。传送门**
> **登记吧，骚年！传送门**

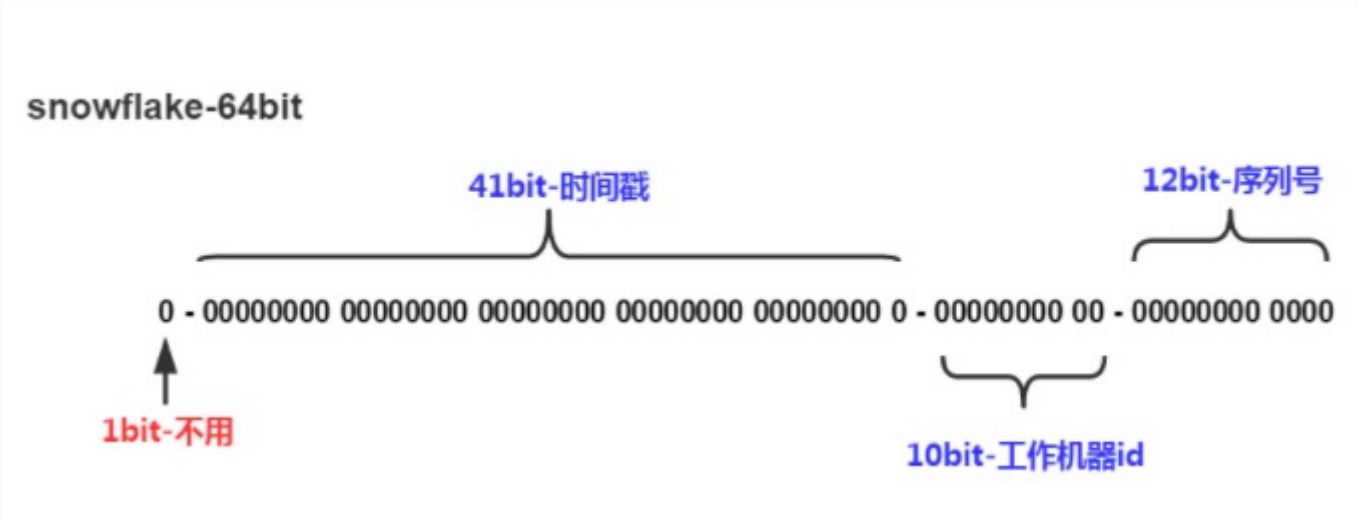
2. KeyGenerator

KeyGenerator，主键生成器接口。实现类通过实现 `#generateKey()` 方法对外提供**生成主键**的功能。

2.1 DefaultKeyGenerator

DefaultKeyGenerator，默认的主键生成器。该生成器采用 Twitter Snowflake 算法实现，生成 **64 Bits** 的 **Long** 型编号。国内另外一款数据库中间件 MyCAT 分布式主键也是基于该算法实现。国内很多大型互联网公司**发号器**服务基于该算法加部分改造实现。所以 DefaultKeyGenerator 必须是**根正苗红**。如果你对**分布式主键**感兴趣，可以看看逗比笔者整理的《谈谈ID》。

咳咳咳，有点跑题了。**编号**由四部分组成，从高位到低位（从左到右）分别是：



Bits	名字	说明
1	符号位	等于 0
41	时间戳	从 2016/11/01 零点开始的毫秒数，支持 2 ^41 /365/24/60/60/1000=69.7年
10	工作进程编号	支持 1024 个进程
12	序列号	每毫秒从 0 开始自增，支持 4096 个编号

- 每个工作进程每秒可以产生 4096000 个编号。是不是灰常牛比 💯

```
//
public final class DefaultKeyGenerator implements KeyGenerator {
    /**
     * 时间偏移量，从2016年11月1日零点开始
     */
```

```
    public static final long EPOCH;
    /**
     * 自增量占用比特
     */
    private static final long SEQUENCE_BITS = 12L;
    /**
     * 工作进程ID比特
     */
    private static final long WORKER_ID_BITS = 10L;
    /**
     * 自增量掩码（最大值）
     */
    private static final long SEQUENCE_MASK = (1 << SEQUENCE_BITS) - 1;
    /**
     * 工作进程ID左移比特数（位数）
     */
    private static final long WORKER_ID_LEFT_SHIFT_BITS = SEQUENCE_BITS;
    /**
     * 时间戳左移比特数（位数）
     */
    private static final long TIMESTAMP_LEFT_SHIFT_BITS = WORKER_ID_LEFT_SHIFT_BITS + WORKER_ID_BITS;
    /**
     * 工作进程ID最大值
     */
    private static final long WORKER_ID_MAX_VALUE = 1L << WORKER_ID_BITS;

    @Setter
    private static TimeService timeService = new TimeService();
    /**
```

```
     * 工作进程ID
     */
    private static long workerId;

    static {
        Calendar calendar = Calendar.getInstance();
        calendar.set(2016, Calendar.NOVEMBER, 1);
        calendar.set(Calendar.HOUR_OF_DAY, 0);
        calendar.set(Calendar.MINUTE, 0);
        calendar.set(Calendar.SECOND, 0);
        calendar.set(Calendar.MILLISECOND, 0);
        EPOCH = calendar.getTimeInMillis();
    }
    /**
     * 最后自增量
     */
    private long sequence;
    /**
     * 最后生成编号时间戳，单位：毫秒
     */
    private long lastTime;

    /**
     * 设置工作进程Id.
     *
     * @param workerId 工作进程Id
     */
    public static void setWorkerId(final long workerId) {
        Preconditions.checkArgument(workerId >= 0L && workerId < WORKER_ID_MAX_VALUE);
```

```
        DefaultKeyGenerator.workerId = workerId;
    }

    /**
     * 生成Id.
     *
     * @return 返回@{@link Long}类型的Id
     */
    @Override
    public synchronized Number generateKey() {
        // 保证当前时间大于最后时间。时间回退会导致产生重复id
        long currentMillis = timeService.getCurrentMillis();
        Preconditions.checkState(lastTime <= currentMillis, "Clock is moving backwards, last time is %
        // 获取序列号
        if (lastTime == currentMillis) {
            if (0L == (sequence = ++sequence & SEQUENCE_MASK)) { // 当获得序号超过最大值时，归0，并去获得
                currentMillis = waitUntilNextTime(currentMillis);
            }
        } else {
            sequence = 0;
        }
        // 设置最后时间戳
        lastTime = currentMillis;
        if (log.isDebugEnabled()) {
            log.debug("{}-{}-{}", new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date(last
        }
        // 生成编号
        return ((currentMillis - EPOCH) << TIMESTAMP_LEFT_SHIFT_BITS) | (workerId << WORKER_ID_LEFT_SH
    }
```

```
    /**
     * 不停获得时间，直到大于最后时间
     *
     * @param lastTime 最后时间
     * @return 时间
     */
    private long waitUntilNextTime(final long lastTime) {
        long time = timeService.getCurrentMillis();
        while (time <= lastTime) {
            time = timeService.getCurrentMillis();
        }
        return time;
    }
}
```

- `EPOCH = calendar.getTimeInMillis();` 计算 2016/11/01 零点开始的毫秒数。
- `#generateKey()` 实现逻辑
    1. 校验当前时间**小于等于**最后生成编号时间戳，避免服务器时钟同步，可能产生时间回退，导致产生**重复**编号
    - 获得序列号。当前时间戳可获得自增量到达最大值时，调用 `#waitUntilNextTime()` 获得下一毫秒
    - 设置最后生成编号时间戳，用于校验时间回退情况
    - 位操作生成**编号**

总的来说，Twitter Snowflake 算法实现上是相对简单易懂的，较为麻烦的是**怎么解决工作进程编号的分配**？

1. 超过 1024 个怎么办？

2. 怎么保证全局唯一？

第一个问题，将分布式主键生成独立成一个**发号器**服务，提供生成分布式编号的功能。这个不在本文的范围内，有兴趣的同学可以 Google 下。

第二个问题，通过 Zookeeper、Consul、Etcd 等提供分布式配置功能的中间件。当然 Sharding-JDBC 也提供了不依赖这些服务的方式，我们一个一个往下看。

2.2 HostNameKeyGenerator

> 根据**机器名最后的数字编号**获取工作进程编号。
> 如果线上机器命名有统一规范,建议使用此种方式。
> 例如，机器的 HostName 为: `dangdang-db-sharding-dev-01` (公司名-部门名-服务名-环境名-编号)，会截取 HostName 最后的编号 01 作为工作进程编号(workId)。

```
// HostNameKeyGenerator.java
static void initWorkerId() {
    InetAddress address;
    Long workerId;
    try {
        address = InetAddress.getLocalHost();
    } catch (final UnknownHostException e) {
        throw new IllegalStateException("Cannot get LocalHost InetAddress, please check your network!")
    }
    String hostName = address.getHostName();
    try {
        workerId = Long.valueOf(hostName.replace(hostName.replaceAll("\\d+$", ""), ""));
    } catch (final NumberFormatException e) {
```

```
            throw new IllegalArgumentException(String.format("Wrong hostname:%s, hostname must be end with
        }
        DefaultKeyGenerator.setWorkerId(workerId);
    }
```

2.3 IPKeyGenerator

> 根据**机器IP**获取工作进程编号。
> 如果线上机器的IP二进制表示的最后10位不重复,建议使用此种方式。
> 例如，机器的IP为192.168.1.108，二进制表示: `11000000 10101000 00000001 01101100` ，截取最后 10 位 `01 01101100` ，转为十进制 364，设置工作进程编号为 364。

```
// IPKeyGenerator.java
static void initWorkerId() {
    InetAddress address;
    try {
        address = InetAddress.getLocalHost();
    } catch (final UnknownHostException e) {
        throw new IllegalStateException("Cannot get LocalHost InetAddress, please check your network!")
    }
    byte[] ipAddressByteArray = address.getAddress();
    DefaultKeyGenerator.setWorkerId((long) (((ipAddressByteArray[ipAddressByteArray.length - 2] & 0B11)
            + (ipAddressByteArray[ipAddressByteArray.length - 1] & 0xFF)));
}
```

2.4 IPSectionKeyGenerator

来自 **DogFc** 贡献，对 IPKeyGenerator 进行改造。

> 浏览 IPKeyGenerator 工作进程编号生成的规则后，感觉对服务器IP后10位（特别是IPV6）数值比较约束。
> 有以下优化思路：
> 因为工作进程编号最大限制是 2^10，我们生成的工程进程编号只要满足小于 1024 即可。
> 1.针对IPV4:
> ....IP最大 255.255.255.255。而（255+255+255+255) < 1024。
> ....因此采用IP段数值相加即可生成唯一的workerId，不受IP位限制。

2. 针对IPV6:
   ....IP最大 ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
   ....为了保证相加生成出的工程进程编号 < 1024,思路是将每个 Bit 位的后6位相加。这样在一定程度上也可以满足workerId不重复的问题。
   使用这种 IP 生成工作进程编号的方法,必须保证IP段相加不能重复

对于 IPV6 ：2^ 6 = 64。64 * 8 = 512 < 1024。

```
// IPSectionKeyGenerator.java
static void initWorkerId() {
   InetAddress address;
   try {
       address = InetAddress.getLocalHost();
   } catch (final UnknownHostException e) {
```

```
            throw new IllegalStateException("Cannot get LocalHost InetAddress, please check your network!")
        }
        byte[] ipAddressByteArray = address.getAddress();
        long workerId = 0L;
        // IPV4
        if (ipAddressByteArray.length == 4) {
            for (byte byteNum : ipAddressByteArray) {
                workerId += byteNum & 0xFF;
            }
        // IPV6
        } else if (ipAddressByteArray.length == 16) {
            for (byte byteNum : ipAddressByteArray) {
                workerId += byteNum & 0B111111;
            }
        } else {
            throw new IllegalStateException("Bad LocalHost InetAddress, please check your network!");
        }
        DefaultKeyGenerator.setWorkerId(workerId);
    }
```

666. 彩蛋

没有彩蛋。HOHOHO

道友，分享一波朋友圈可好。

感谢你，技术如此只好，还关注我的公众号。

Sharding-JDBC

分享

PREVIOUS:

« Sharding-JDBC 源码分析 —— SQL 执行

NEXT:

» Sharding-JDBC 源码分析 —— SQL 路由改写