1、面试题

分库分表之后，id主键如何处理？

2、面试官心里分析

其实这是分库分表之后你必然要面对的一个问题，就是id咋生成？因为要是分成多个表之后，每个表都是从1开始累加，那肯定不对啊，需要一个全局唯一的id来支持。所以这都是你实际生产环境中必须考虑的问题。

3、面试题剖析

（1）数据库自增id

这个就是说你的系统里每次得到一个id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个id。拿到这个id之后再往对应的分库分表里去写入。

这个方案的好处就是方便简单，谁都会用；缺点就是单库生成自增id，要是高并发的话，就会有瓶颈的；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前id最大值，然后自己递增几个id，一次性返回一批id，然后再把当前最大id值修改成递增几个id之后的一个值；但是无论怎么说都是基于单个数据库。

适合的场景：你分库分表就俩原因，要不就是单库并发太高，要不就是单库数据量太大；除非是你并发不高，但是数据量太大导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。

并发很低，几百/s，但是数据量大，几十亿的数据，所以需要靠分库分表来存放海量的数据

（2）uuid

好处就是本地生成，不要基于数据库来了；不好之处就是，uuid太长了，作为主键性能太差了，不适合用于主键。

适合的场景：如果你是要随机生成个什么文件名了，编号之类的，你可以用uuid，但是作为主键是不能用uuid的。

UUID.randomUUID().toString().replace(“-”, “”) -> sfsdf23423rr234sfdaf

（3）获取系统当前时间

这个就是获取当前时间即可，但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。基本就不用考虑了。

适合的场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号，订单编号，时间戳 + 用户id + 业务含义编码

（4）snowflake算法

twitter开源的分布式id生成算法，就是把一个64位的long型的id，1个bit是不用的，用其中的41 bit作为毫秒数，用10 bit作为工作机器id，12 bit作为序列号

1 bit：不用，为啥呢？因为二进制里第一个bit为如果是1，那么都是负数，但是我们生成的id都是正数，所以第一个bit统一都是0

41 bit：表示的是时间戳，单位是毫秒。41 bit可以表示的数字多达2^41 - 1，也就是可以标识2 ^ 41 - 1个毫秒值，换算成年就是表示69年的时间。

10 bit：记录工作机器id，代表的是这个服务最多可以部署在2^10台机器上哪，也就是1024台机器。但是10 bit里5个bit代表机房id，5个bit代表机器id。意思就是最多代表2 ^ 5个机房（32个机房），每个机房里可以代表2 ^ 5个机器（32台机器）。

12 bit：这个是用来记录同一个毫秒内产生的不同id，12 bit可以代表的最大正整数是2 ^ 12 - 1 = 4096，也就是说可以用这个12bit代表的数字来区分同一个毫秒内的4096个不同的id

64位的long型的id，64位的long -> 二进制

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

2018-01-01 10:00:00 -> 做了一些计算，再换算成一个二进制，41bit来放 -> 0001100 10100010 10111110 10001001 01011100 00

机房id，17 -> 换算成一个二进制 -> 10001

机器id，25 -> 换算成一个二进制 -> 11001

snowflake算法服务，会判断一下，当前这个请求是否是，机房17的机器25，在2175/11/7 12:12:14时间点发送过来的第一个请求，如果是第一个请求

假设，在2175/11/7 12:12:14时间里，机房17的机器25，发送了第二条消息，snowflake算法服务，会发现说机房17的机器25，在2175/11/7 12:12:14时间里，在这一毫秒，之前已经生成过一个id了，此时如果你同一个机房，同一个机器，在同一个毫秒内，再次要求生成一个id，此时我只能把加1

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000001

比如我们来观察上面的那个，就是一个典型的二进制的64位的id，换算成10进制就是910499571847892992。

public class IdWorker{

private long workerId;

private long datacenterId;

private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence){

// sanity check for workerId

// 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0

if (workerId > maxWorkerId || workerId < 0) {

throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0",maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0) {

throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0",maxDatacenterId));

}

System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worker id bits %d, sequence bits %d, workerid %d",

timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;

private long datacenterIdBits = 5L;

private long maxWorkerId = -1L ^ (-1L << workerIdBits); // 这个是二进制运算，就是5 bit最多只能有31个数字，也就是说机器id最多只能是32以内

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); // 这个是一个意思，就是5 bit最多只能有31个数字，机房id最多只能是32以内

private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId(){

return workerId;

}

public long getDatacenterId(){

return datacenterId;

}

public long getTimestamp(){

return System.currentTimeMillis();

}

public synchronized long nextId() {

// 这儿就是获取当前时间戳，单位是毫秒

long timestamp = timeGen();

if (timestamp < lastTimestamp) {

System.err.printf("clock is moving backwards. Rejecting requests until %d.", lastTimestamp);

throw new RuntimeException(String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",

lastTimestamp - timestamp));

}

// 0

// 在同一个毫秒内，又发送了一个请求生成一个id，0 -> 1

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask; // 这个意思是说一个毫秒内最多只能有4096个数字，无论你传递多少进来，这个位运算保证始终就是在4096这个范围内，避免你自己传递个sequence超过了4096这个范围

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {

sequence = 0;

}

// 这儿记录一下最近一次生成id的时间戳，单位是毫秒

lastTimestamp = timestamp;

// 这儿就是将时间戳左移，放到41 bit那儿；将机房id左移放到5 bit那儿；将机器id左移放到5 bit那儿；将序号放最后10 bit；最后拼接起来成一个64 bit的二进制数字，转换成10进制就是个long型

return ((timestamp - twepoch) << timestampLeftShift) |

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;

}

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

private long timeGen(){

return System.currentTimeMillis();

}

//---------------测试---------------

public static void main(String[] args) {

IdWorker worker = new IdWorker(1,1,1);

for (int i = 0; i < 30; i++) {

System.out.println(worker.nextId());

}

}

}

怎么说呢，大概这个意思吧，就是说41 bit，就是当前毫秒单位的一个时间戳，就这意思；然后5 bit是你传递进来的一个机房id（但是最大只能是32以内），5 bit是你传递进来的机器id（但是最大只能是32以内），剩下的那个10 bit序列号，就是如果跟你上次生成id的时间还在一个毫秒内，那么会把顺序给你累加，最多在4096个序号以内。

所以你自己利用这个工具类，自己搞一个服务，然后对每个机房的每个机器都初始化这么一个东西，刚开始这个机房的这个机器的序号就是0。然后每次接收到一个请求，说这个机房的这个机器要生成一个id，你就找到对应的Worker，生成。

他这个算法生成的时候，会把当前毫秒放到41 bit中，然后5 bit是机房id，5 bit是机器id，接着就是判断上一次生成id的时间如果跟这次不一样，序号就自动从0开始；要是上次的时间跟现在还是在一个毫秒内，他就把seq累加1，就是自动生成一个毫秒的不同的序号。

这个算法那，可以确保说每个机房每个机器每一毫秒，最多生成4096个不重复的id。

利用这个snowflake算法，你可以开发自己公司的服务，甚至对于机房id和机器id，反正给你预留了5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

这个snowflake算法相对来说还是比较靠谱的，所以你要真是搞分布式id生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。