



科普：为什么 String hashCode 方法选择数字31作为乘子

1. 背景

某天，我在写代码的时候，无意中点开了 String hashCode 方法。然后大致看了一下 hashCode 的实现，发现并不是很复杂。但是我从源码中发现了一个奇怪的数字，也就是本文的主角31。这个数字居然不是用常量声明的，所以没法从字面意思上推断这个数字的用途。后来带着疑问和好奇心，到网上去找资料查询一下。在看完资料后，默默的感叹了一句，原来是这样啊。那么到底是哪样呢？在接下来章节里，请大家带着好奇心和我揭开数字31的用途之谜。

2. 选择数字31的原因

在详细说明 String hashCode 方法选择数字31的作为乘子的原因之前，我们先来看看 String hashCode 方法是怎样实现的，如下：

```
public int hashCode() {  
    int h = hash;  
    if (h == 0 && value.length > 0) {  
        char val[] = value;  
  
        for (int i = 0; i < value.length; i++) {  
            h = 31 * h + val[i];  
        }  
        hash = h;  
    }  
    return h;  
}
```

上面的代码就是 String hashCode 方法的实现，是不是很简单。实际上 hashCode 方法核心的计算逻辑只有三行，也就是代码中的 for 循环。我们可以由上面的 for 循环推导出一个计算公式，hashCode 方法注释中已经给出。如下：

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

这里说明一下，上面的 s 数组即源码中的 val 数组，是 String 内部维护的一个 char 类型数组。这里我来简单推导一下这个公式：

假设 $n=3$

```
i=0 -> h = 31 * 0 + val[0]  
i=1 -> h = 31 * (31 * 0 + val[0]) + val[1]  
i=2 -> h = 31 * (31 * (31 * 0 + val[0]) + val[1]) + val[2]  
      h = 31*31*31*0 + 31*31*val[0] + 31*val[1] + val[2]  
      h = 31^(n-1)*val[0] + 31^(n-2)*val[1] + val[2]
```

上面的公式包括公式的推导并不是本文的重点，大家了解了解即可。接下来来说说本文的重点，即选择31的理由。从网上的资料来看，一般有如下两个原因：

第一，31是一个不大不小的质数，是作为 hashCode 乘子的优选质数之一。另外一些相近的质数，比如37、41、43等等，也都是不错的选择。那么为啥偏偏选中了31呢？请看第二个原因。

第二，31可以被 JVM 优化， $31 * i = (i << 5) - i$ 。

上面两个原因中，第一个需要解释一下，第二个比较简单，就不说了。下面我来解释第一个理由。一般在设计哈希算法时，会选择一个特殊的质数。至于为啥选择质数，我想应该是可以降低哈希算法的冲突率。至于原因，这个就要问数学家了，我几乎可以忽略的数学水平解释不了这个原因。上面说到，31是一个不大不小的质数，是优选乘子。那为啥同是质数的2和101（或者更大的质数）就不是优选乘子呢，分析如下。

这里先分析质数2。首先，假设 $n = 6$ ，然后把质数2和 n 带入上面的计算公式。并仅计算公式中次数最高的那一项，结果是 $2^5 = 32$ ，是不是很小。所以这里可以断定，当字符串长度不是很长时，用质数2做为乘子算出的哈希值，数值不会很大。也就是说，哈希值会分布在一个较小的数值区间内，分布性不佳，最终可能会导致冲突率上升。

上面说了，质数2做为乘子会导致哈希值分布在一个较小区间内，那么如果用一个大质数101会产生什么样的结果呢？根据上面的分析，我想大家应该可以猜出结果了。就是不用再担心哈希值会分布在一个小的区间内了，因为 $101^5 = 10,510,100,501$ 。但是要注意的是，这个计算结果太大了。如果用 int 类型表示哈希值，结果会溢出，最终导致数值信息丢失。尽管数值信息丢失并不一定会导致冲突率上升，但是我们暂且先认为质数101（或者更大的质数）也不是很好的选择。最后，我们再来看看质数31的计算结果： $31^5 = 28629151$ ，结果值相对于 32 和 $10,510,100,501$ 来说。是不是很nice，不大不小。

上面用了比较简陋的数学手段证明了数字31是一个不大不小的质数，是作为 hashCode 乘子的优选质数之一。接下来我会用详细的实验来验证上面的结论，不过在验证前，我们先看看 Stack Overflow 上关于这个问题的讨论，[Why does Java's hashCode\(\) in String use 31 as a multiplier?](#)。其中排名第一的答案引用了《Effective Java》中的一段话，这里也引用一下：

The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: $31 * i == (i \ll 5) - i$. Modern VMs do this sort of optimization automatically.

简单翻译一下：

选择数字31是因为它是一个奇质数，如果选择一个偶数会在乘法运算中产生溢出，导致数值信息丢失，因为乘二相当于移位运算。选择质数的优势并不是特别的明显，但这是一个传统。同时，数字31有一个很好的特性，即乘法运算可以被移位和减法运算取代，来获取更好的性能： $31 * i == (i \ll 5) - i$ ，现代的 Java 虚拟机可以自动的完成这个优化。

排名第二的答案设这样说的：

As Goodrich and Tamassia point out, If you take over 50,000 English words (formed as the union of the word lists provided in two variants of Unix), using the constants 31, 33, 37, 39, and 41 will produce less than 7 collisions in each case. Knowing this, it should come as no surprise that many Java implementations choose one of these constants.

这段话也翻译一下：

正如 Goodrich 和 Tamassia 指出的那样，如果你对超过 50,000 个英文单词（由两个不同版本的 Unix 字典合并而成）进行 hash code 运算，并使用常数 31, 33, 37, 39 和 41 作为乘子，每个常数算出的哈希值冲突数都小于7个，所以在上面几个常数中，常数 31 被 Java 实现所选用也就不足为奇了。

上面的两个答案完美的解释了 Java 源码中选用数字 31 的原因。接下来，我将针对第二个答案就行验证，请大家继续往下看。

3. 实验及数据可视化

本节，我将使用不同的数字作为乘子，对超过23万个英文单词进行哈希运算，并计算哈希算法的冲突率。同时，我也将针对不同乘子算出的哈希值分布情况进行可视化处理，让大家可以直观的看到数据分布情况。本次实验所使用的数据是 Unix/Linux 平台中的英文字典文件，文件路径为 `/usr/share/dict/words`。

3.1 哈希值冲突率计算

计算哈希算法冲突率并不难，比如可以一次性将所有单词的 hash code 算出，并放入 Set 中去掉重复值。之后拿单词数减去 set.size() 即可得出冲突数，有了冲突数，冲突率就可以算出来了。当然，如果使用 JDK8 提供的流式计算 API，则可更方便算出，代码片段如下：

```
        hash = multiplier * hash + str.charAt(i);
    }

    return hash;
}

/**
 * 计算 hash code 冲突率，顺便分析一下 hash code 最大值和最小值，并输出
 * @param multiplier
 * @param hashes
 */
public static void calculateConflictRate(Integer multiplier, List<Integer> hashes) {
    Comparator<Integer> cp = (x, y) -> x > y ? 1 : (x < y ? -1 : 0);
    int maxHash = hashes.stream().max(cp).get();
    int minHash = hashes.stream().min(cp).get();
}
```

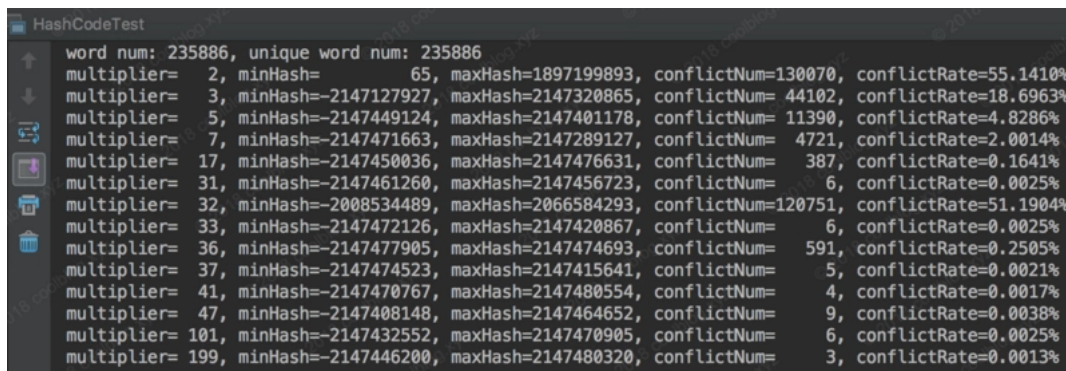
```

int conflictNum = hashes.size() - uniqueHashNum;
double conflictRate = (conflictNum * 1.0) / hashes.size();

System.out.println(String.format("multiplier=%4d, minHash=%11d, maxHash=%10d, conflictNum=%6d, conflictRate=%.4f%%",
    multiplier, minHash, maxHash, conflictNum, conflictRate * 100));
}

```

结果如下：



从上图可以看出，使用较小的质数做为乘子时，冲突率会很高。尤其是质数2，冲突率达到了 55.14%。同时我们注意观察质数2作为乘子时，哈希值的分布情况。可以看得出来，哈希值分布并不是很广，仅仅分布在了整个哈希空间的正半轴部分，即 $0 \sim 2^{31}-1$ 。而负半轴 $-2^{31} \sim -1$ ，则无分布。这也证明了我们上面断言，即质数2作为乘子时，对于短字符串，生成的哈希值分布性不佳。然后再来看看我们之前所说的 31、37、41 这三个不大不小的质数，表现都不错，冲突数都低于7个。而质数 101 和 199 表现的也很不错，冲突率很低，这也说明哈希值溢出并不一定会导致冲突率上升。但是这两个家伙一言不合就溢出，我们认为他们不是哈希算法的优选乘子。最后我们再来看看 32 和 36 这两个偶数的表现，结果并不好，尤其是 32，冲突率超过了50%。尽管 36 表现的要好一点，不过和 31、37相比，冲突率还是比较高的。当然并非所有的偶数作为乘子时，冲突率都会比较高，大家有兴趣可以自己验证。

3.2 哈希值分布可视化

上一节分析了不同数字作为乘子时的冲突率情况，这一节来分析一下不同数字作为乘子时，哈希值的分布情况。在详细分析之前，我先说说哈希值可视化的过程。我原本是打算将所有的哈希值用一维散点图进行可视化，但是后来找了一圈，也没找到合适的画图工具。加之后来想了想，一维散点图可能不合适做哈希值可视化，因为这里有超过23万个哈希值。也就意味着会在图上显示超过23万个散点，如果不出意外的话，这23万个散点会聚集的很密，有可能会变成一个大黑块，就失去了可视化的意义了。所以这里选择了另一种可视化效果更好的图表，也就是 excel 中的平滑曲线的二维散点图（下面简称散点曲线图）。当然这里同样没有把23万散点都显示在图表上，太多了。所以在实际绘图过程中，我将哈希空间等分成了64个子区间，并统计每个区间内的哈希值数量。最后将分区编号做为X轴，哈希值数量为Y轴，就绘制出了我想要的二维散点曲线图了。这里举个例子说明一下吧，以第0分区为例。第0分区数值区间是[-2147483648, -2080374784)，我们统计落在该数值区间内哈希值的数量，得到 `<分区编号, 哈希值数量>` 数值对，这样就可以绘图了。分区代码如下：

```

* @param hashes
*/
public static Map<Integer, Integer> partition(List<Integer> hashes) {
    // step = 2^32 / 64 = 2^26
    final int step = 67108864;
    List<Integer> nums = new ArrayList<>();
    Map<Integer, Integer> statistics = new LinkedHashMap<>();
    int start = 0;
    for (long i = Integer.MIN_VALUE; i <= Integer.MAX_VALUE; i += step) {
        final long min = i;
        final long max = min + step;
        int num = (int) hashes.parallelStream()
            .filter(x -> x >= min && x < max).count();

        statistics.put(start++, num);
        nums.add(num);
    }

    // 为了防止计算出错，这里验证一下
    int hashNum = nums.stream().reduce((x, y) -> x + y).get();
    assert hashNum == hashes.size();

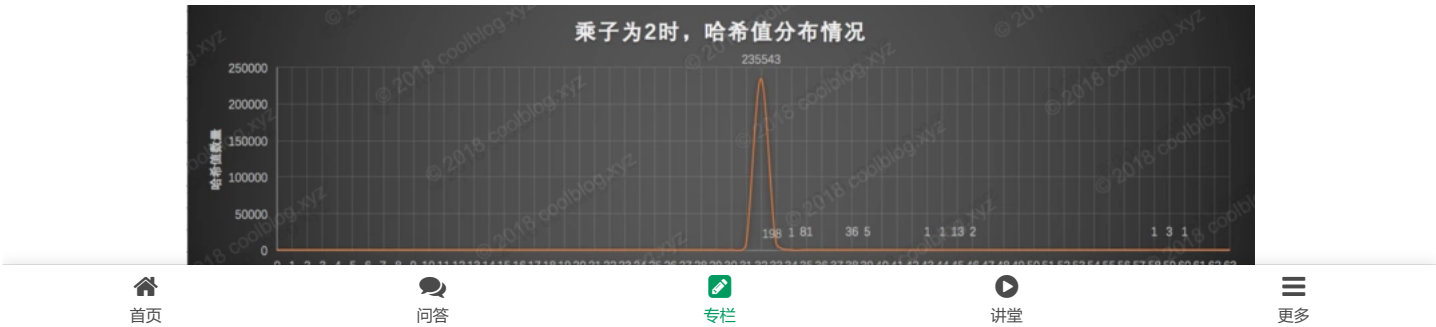
    return statistics;
}

```

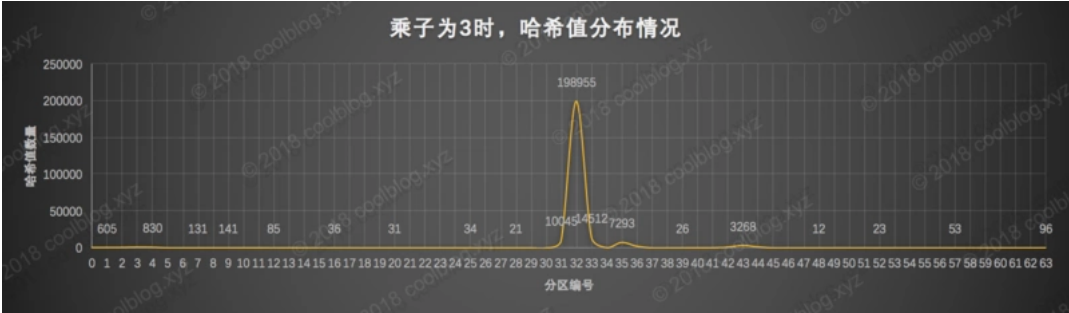
本文中的哈希值是用整形表示的，整形的数值区间是 `[-2147483648, 2147483647]`，区间大小为 `2^32`。所以这里可以将区间等分成64个子区间，每个子区间大小为 `2^26`。详细的分区对照表如下：

分区编号	分区下限	分区上限	分区编号	分区下限	分区上限
0	-2147483648	-2080374784	32	0	67108864
1	-2080374784	-2013265920	33	67108864	134217728
2	-2013265920	-1946157056	34	134217728	201326592
3	-1946157056	-1879048192	35	201326592	268435456
4	-1879048192	-1811939328	36	268435456	335544320
5	-1811939328	-1744830464	37	335544320	402653184
6	-1744830464	-1677721600	38	402653184	469762048
7	-1677721600	-1610612736	39	469762048	536870912
8	-1610612736	-1543503872	40	536870912	603979776
9	-1543503872	-1476395008	41	603979776	671088640
10	-1476395008	-1409286144	42	671088640	738197504
11	-1409286144	-1342177280	43	738197504	805306368
12	-1342177280	-1275068416	44	805306368	872415232
13	-1275068416	-1207959552	45	872415232	939524096
14	-1207959552	-1140850688	46	939524096	1006632960
15	-1140850688	-1073741824	47	1006632960	1073741824
16	-1073741824	-1006632960	48	1073741824	1140850688
17	-1006632960	-939524096	49	1140850688	1207959552
18	-939524096	-872415232	50	1207959552	1275068416
19	-872415232	-805306368	51	1275068416	1342177280
20	-805306368	-738197504	52	1342177280	1409286144
21	-738197504	-671088640	53	1409286144	1476395008
22	-671088640	-603979776	54	1476395008	1543503872
23	-603979776	-536870912	55	1543503872	1610612736
24	-536870912	-469762048	56	1610612736	1677721600
25	-469762048	-402653184	57	1677721600	1744830464
26	-402653184	-335544320	58	1744830464	1811939328
27	-335544320	-268435456	59	1811939328	1879048192
28	-268435456	-201326592	60	1879048192	1946157056
29	-201326592	-134217728	61	1946157056	2013265920
30	-134217728	-67108864	62	2013265920	2080374784
31	-67108864	0	63	2080374784	2147483648

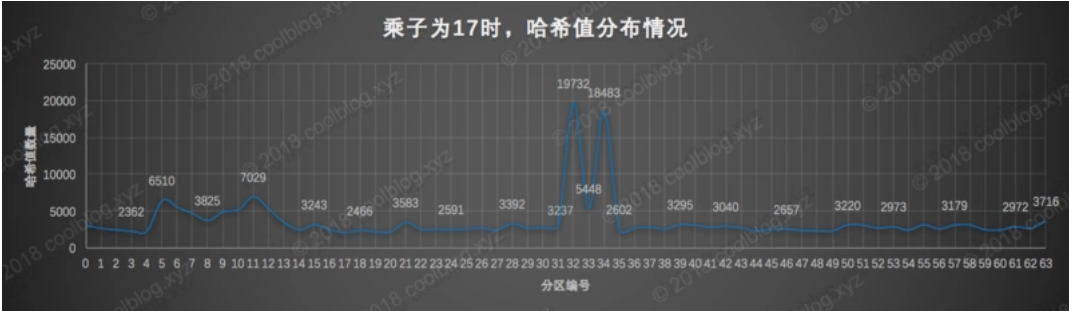
接下来，让我们对照上面的分区表，对数字2、3、17、31、101的散点曲线图进行简单的分析。先从数字2开始，数字2对于的散点曲线图如下：



上面的图还是很一幕了然的，乘子2算出的哈希值几乎全部落在第32分区，也就是 $[0, 67108864)$ 数值区间内，落在其他区间内的哈希值数量几乎可以忽略不计。这也就不难解释为什么数字2作为乘子时，算出哈希值的冲突率如此之高的原因了。所以这样的哈希算法要它有何用啊，拖出去斩了吧。接下来看看数字3作为乘子时的表现：



3作为乘子时，算出的哈希值分布情况和2很像，只不过稍微好了那么一点点。从图中可以看出绝大部分的哈希值最终都落在了第32分区里，哈希值的分布性很差。这个也没啥用，拖出去枪毙5分钟吧。在看看数字17的情况怎么样：



数字17作为乘子时的表现，明显比上面两个数字好点了。虽然哈希值在第32分区和第34分区有一定的聚集，但是相比较上面2和3，情况明显好好了很多。除此之外，17作为乘子算出的哈希值在其他区也均有分布，且较为均匀，还算是一个不错的乘子吧。



接下来看看我们本文的主角31了，31作为乘子算出的哈希值在第33分区有一定的小聚集。不过相比于数字17，主角31的表现又好了一些。首先是哈希值的聚集程度没有17那么严重，其次哈希值在其他区分布的情况也要好于17。总之，选31，准没错啊。



最后再来看看大质数101的表现，不难看出，质数101作为乘子时，算出的哈希值分布情况要好于主角31，有点喧宾夺主的意思。不过不可否认的是，质数101的作为乘子时，哈希值的分布性确实更加均匀。所以如果不在意质数101容易导致数据信息丢失问题，或许其是一个更好的选择。

4.写在最后

经过上面的分析与实践，我想大家应该明白了 String hashCode 方法中选择使用数字 31 作为乘子的原因了。本文本质是一篇简单的科普文而已，并没有银弹 ☺️。如果大家读完后觉得又涨知识了，那这篇文章的目的就达到了。最后，本篇文章的配图画的还是很辛苦的，所以如果大家觉得文章不错，不妨就给个赞吧，就当是对我的鼓励了。另外，如果文章中有不妥或者错误的地方，也欢迎指出来。如果能不吝赐教，那就更好了。最后祝大家生活愉快，再见。

本文在知识共享许可协议 4.0 下发布，转载请注明出处
作者：coolblog
为了获得更好的分类阅读体验，
请移步至本人的个人博客：<http://www.coolblog.xyz>



本作品采用[知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议](#)进行许可。



赞 | 68

收藏 | 60

- 1 视频通话SDK
声网Agora.io，API接口，4行代码接入。每月1万分钟免费。 www.agora.io
- 2 亿速云高防服务器送防御增强防CC
亿速云高防服务器，20+行业领袖强烈推荐 免备案服务器低延时CN2高速带宽即买即用

你可能感兴趣的

- 从字节码角度看String、StringBuffer、StringBuilder的不同 拿客_三产 string stringbuilder stringbuffer java
- Java源码中有意思的代码片段 Lazy 源码学习 源码分析 java
- Java思想之容器implements与接口interface 夏日小草 java
- Equinox MANIFEST.MF 中文处理 Bug khotyn equinox eclipse java
- Java 8之stream介绍和使用 Half java
- 树状数组理解 diert java 数据结构
- 代理技术简介 TheHope java 代理模式
- 为mysql数据库建立索引 js好难 java

22 条评论

默认排序 时间排序

 孤独的探索号 · 2017年08月24日

之前我看到源码也困惑了一阵，只是觉得应该和 2^5 (0-31) 有关系？

👍 赞 回复


我第一次看的时候，也比较困惑，源码注释中也没有说使用31的原因

— coolblog 作者 · 2017年08月24日

@code4fun 是啊，不过你找到了原因？

— 孤独的探索号 · 2017年08月24日

添加回复

 过好今天 · 2018年02月26日

这是一篇质量非常高的好文章，我之前看string源代码也是非常好奇，为什么不选择其他的数字作为乘法因子，偏偏选择31这个数字，虽然我也明白hashCode散列算法是为了分布的广，避免出现很大的hash冲突，但是我没有楼主这么认真去探索，想办法去求取hash冲突率。你的这种精神值得学习。

👍 赞 回复

过奖了过奖了。网上相关的文章都只有理论，缺少实验验证。所以我特地验证一下，用于支撑理论，这样子感觉文章会更充实一些。

— coolblog 作者 · 2018年02月27日

添加回复

 誓不为妃0 · 2018年03月19日

👍 赞 回复

谢谢阅读，过奖了

— [coolblog](#) 作者 · 2018年03月20日

[添加回复](#)



[交不起学费](#) · 2018年06月19日

写的很清晰, 最后有个疑问.

所以如果不在意质数101容易导致数据信息丢失问题, 或许其是一个更好的选择。

为什么101会导致数据丢失呢..大佬可以解释一下吗

👍 赞 回复

使用 101 计算 hash code 容易导致整型溢出, 导致计算精度丢失

— [coolblog](#) 作者 · 2018年06月19日

[@coolblog](#) 原来如此. 3q

— [交不起学费](#) · 2018年06月20日

[添加回复](#)



[leopold](#) · 2018年06月29日

非常高质量的一篇文章, 这个原理终于搞明白了, 赞楼主。

👍 赞 回复

能帮到你, 我很开心

— [coolblog](#) 作者 · 2018年07月18日

[添加回复](#)



[whataKitty_57c81c1df3e34](#) · 2018年07月17日

赞一个, 很少看到分析这么深的文章, jdk里面有很多这种数值, 希望楼主有空可以多分析分析~

👍 赞 回复

以后用空看看, 现在在分析其他框架的源码

— [coolblog](#) 作者 · 2018年07月18日

[添加回复](#)



[chigend](#) · 1月25日

赞一个

👍 赞 回复



[滕林](#) · 2月15日

为了给你点赞,我注册了一个账号

👍 赞 回复

感谢支持

— [coolblog](#) 作者 · 2月15日

[添加回复](#)



[greenHard](#) · 3月8日

写的太棒了。赞!

👍 赞 回复



[greenHard](#) · 3月8日

是否可以转载。

👍 赞 回复

可以, 在显著位置标明文章来源和原文链接。谢谢配合



首页



问答



专栏



讲堂



更多

好的。谢谢！

— greenHard · 3月11日

[添加回复](#)



骆剑 · 4月6日

为什么我做的实验，32作为乘子效果和31差不多？

[赞](#) [回复](#)

文明社会，理性评论

发布评论



Copyright © 2011-2019 SegmentFault. 当前呈现版本 19.02.27

浙ICP备 15005796号-2 浙公网安备 33010602002000号 杭州堆栈科技有限公司版权所有

CDN 存储服务由 又拍云 赞助提供

[移动版](#) [桌面版](#)