应用 4：四两拨千斤 —— HyperLogLog

在开始这一节之前，我们先思考一个常见的业务问题：如果你负责开发维护一个大型的网站，有一天老板找产品经理要网站每个网页每天的 UV 数据，然后让你来开发这个统计模块，你会如何实现？

如果统计 PV 那非常好办，给每个网页一个独立的 Redis 计数器就可以了，这个计数器的 key 后缀加上当天的日期。这样来一个请求，incrby 一次，最终就可以统计出所有的 PV 数据。

但是 UV 不一样，它要去重，同一个用户一天之内的多次访问请求只能计数一次。这就要求每一个网页请求都需要带上用户的 ID，无论是登陆用户还是未登陆用户都需要一个唯一 ID 来标识。

你也许已经想到了一个简单的方案，那就是为每一个页面一个独立的 set 集合来存储所有当天访问过此页面的用户 ID。当一个请求过来时，我们使用 sadd 将用户 ID 塞进去就可以了。通过 scard 可以取出这个集合的大小，这个数字就是这个页面的 UV 数据。没错，这是一个非常简单的方案。

但是，如果你的页面访问量非常大，比如一个爆款页面几千万的 UV，你需要一个很大的 set 集合来统计，这就非常浪费空间。如果这样的页面很多，那所需要的存储空间是惊人的。为这样一个去重功能就耗费这样多的存储空间，值得么？其实老板需要的数据又不需要太精确，105w 和 106w 这两个数字对于老板们来说并没有多大区别，So，有没有更好的解决方案呢？

这就是本节要引入的一个解决方案，Redis 提供了 HyperLogLog 数据结构就是用来解决这种统计问题的。HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%，这样的精确度已经可以满足上面的 UV 统计需求了。

HyperLogLog 数据结构是 Redis 的高级数据结构，它非常有用，但是令人感到意外的是，使用过它的人非常少。

使用方法

HyperLogLog 提供了两个指令 pfadd 和 pfcount，根据字面意义很好理解，一个是增加计数，一个是获取计数。pfadd 用法和 set 集合的 sadd 是一样的，来一个用户 ID，就将用户 ID 塞进去就是。pfcount 和 scard 用法是一样的，直接获取计数值。

```
127.0.0.1:6379> pfadd codehole user1
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 1
127.0.0.1:6379> pfadd codehole user2
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 2
127.0.0.1:6379> pfadd codehole user3
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 3
127.0.0.1:6379> pfadd codehole user4
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 4
127.0.0.1:6379> pfadd codehole user5
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 5
127.0.0.1:6379> pfadd codehole user6
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 6
127.0.0.1:6379> pfadd codehole user7 user8 user9 user10
(integer) 1
127.0.0.1:6379> pfcount codehole
(integer) 10
```

简单试了一下，发现还蛮精确的，一个没多也一个没少。接下来我们使用脚本，往里面灌更多的数据，看看它是否还可以继续精确下去，如果不能精确，差距有多大。人生苦短，我用 Python！Python 脚本走起来！😄

```py
# coding: utf-8

import redis

client = redis.StrictRedis()
for i in range(1000):
    client.pfadd("codehole", "user%d" % i)
    total = client.pfcount("codehole")
    if total != i+1:
        print total, i+1
        break
```

当然 Java 也不错，大同小异，下面是 Java 版本：

```java
public class PfTest {
  public static void main(String[] args) {
    Jedis jedis = new Jedis();
    for (int i = 0; i < 1000; i++) {
      jedis.pfadd("codehole", "user" + i);
      long total = jedis.pfcount("codehole");
      if (total != i + 1) {
        System.out.printf("%d %d\n", total, i + 1);
        break;
      }
    }
    jedis.close();
  }
}
```

我们来看下输出：

```
> python pftest.py
99 100
```

当我们加入第 100 个元素时，结果开始出现了不一致。接下来我们将数据增加到 10w 个，看看总量差距有多大。

```
# coding: utf-8

import redis

client = redis.StrictRedis()
for i in range(100000):
    client.pfadd("codehole", "user%d" % i)
print 100000, client.pfcount("codehole")
```

Java 版：

```java
public class JedisTest {
  public static void main(String[] args) {
    Jedis jedis = new Jedis();
    for (int i = 0; i < 100000; i++) {
      jedis.pfadd("codehole", "user" + i);
    }
    long total = jedis.pfcount("codehole");
    System.out.printf("%d %d\n", 100000, total);
    jedis.close();
  }
}
```

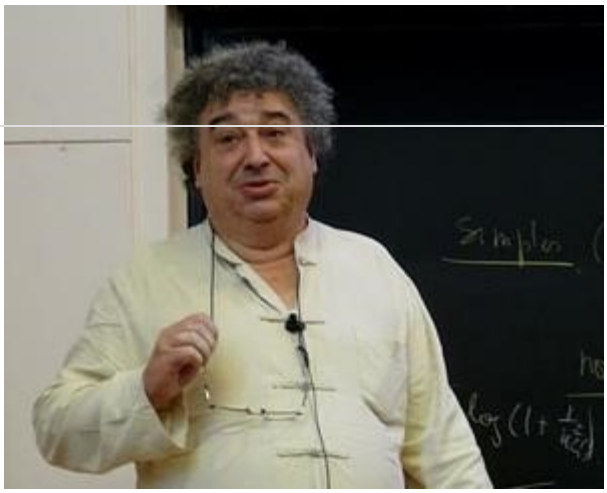
跑了约半分钟，我们看输出：

```
> python pftest.py
100000 99723
```

差了 277 个，按百分比是 0.277%，对于上面的 UV 统计需求来说，误差率也不算高。然后我们把上面的脚本再跑一边，也就相当于将数据重复加入一边，查看输出，可以发现，pfcount 的结果没有任何改变，还是 99723，说明它确实具备去重功能。

pfadd 这个 pf 是什么意思?

它是 HyperLogLog 这个数据结构的发明人 Philippe Flajolet 的首字母缩写，老师觉得他发型很酷，看起来是个佛系教授。



pfmerge 适合什么场合用?

HyperLogLog 除了上面的 pfadd 和 pfcount 之外，还提供了第三个指令 pfmerge，用于将多个 pf 计数值累加在一起形成一个新的 pf 值。

比如在网站中我们有两个内容差不多的页面，运营说需要这两个页面的数据进行合并。其中页面的 UV 访问量也需要合并，那这个时候 pfmerge 就可以派上用场了。

注意事项

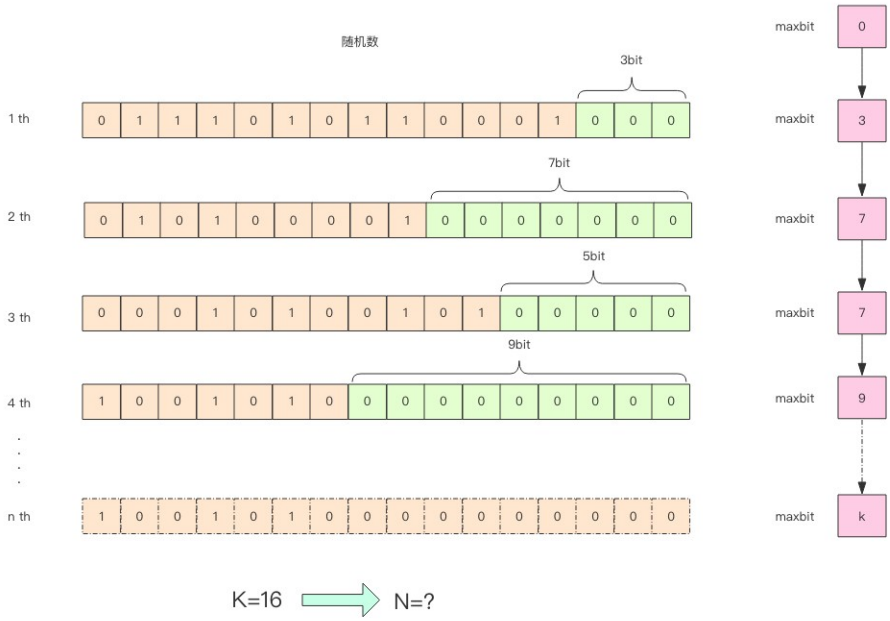
HyperLogLog 这个数据结构不是免费的，不是说使用这个数据结构要花钱，它需要占据一定 12k 的存储空间，所以它不适合统计单个用户相关的数据。如果你的用户上亿，可以算算，这个空间成本是非常惊人的。但是相比 set 存储方案，HyperLogLog 所使用的空间那真是可以使用千斤对比四两来形容了。

不过你也不必过于当心，因为 Redis 对 HyperLogLog 的存储进行了优化，在计数比较小时，它的存储空间采用稀疏矩阵存储，空间占用很小，仅仅在计数慢慢变大，稀疏矩阵占用空间渐渐超过了阈值时才会一次性转变成稠密矩阵，才会占用 12k 的空间。

HyperLogLog 实现原理

HyperLogLog 的使用非常简单，但是实现原理比较复杂，如果读者没有特别的兴趣，下面的内容暂时可以跳过不看。

为了方便理解 HyperLogLog 的内部实现原理，我画了下面这张图



这张图的意思是，给定一系列的随机整数，我们记录下低位连续零位的最大长度 k，通过这个 k 值可以估算出随机数的数量。 首先不问为什么，我们编写代码做一个实验，观察一下随机整数的数量和 k 值的关系。

```
import math
import random

# 算低位零的个数
def low_zeros(value):
    for i in xrange(1, 32):
        if value >> i << i != value:
            break
    return i - 1


# 通过随机数记录最大的低位零的个数
class BitKeeper(object):

    def __init__(self):
        self.maxbits = 0
```

```
        def random(self):
            value = random.randint(0, 2**32-1)
            bits = low_zeros(value)
            if bits > self.maxbits:
                self.maxbits = bits


    class Experiment(object):

        def __init__(self, n):
            self.n = n
            self.keeper = BitKeeper()

        def do(self):
            for i in range(self.n):
                self.keeper.random()

        def debug(self):
            print self.n, '%.2f' % math.log(self.n, 2), self.keeper.maxbits


    for i in range(1000, 100000, 100):
        exp = Experiment(i)
        exp.do()
        exp.debug()
```

Java 版：

```
    public class PfTest {

      static class BitKeeper {
        private int maxbits;

        public void random() {
          long value = ThreadLocalRandom.current().nextLong(2L << 32);
          int bits = lowZeros(value);
          if (bits > this.maxbits) {
            this.maxbits = bits;
          }
        }

        private int lowZeros(long value) {
          int i = 1;
          for (; i < 32; i++) {
            if (value >> i << i != value) {
              break;
            }
```

```
    }
    return i - 1;
  }
}

static class Experiment {
  private int n;
  private BitKeeper keeper;

  public Experiment(int n) {
    this.n = n;
    this.keeper = new BitKeeper();
  }

  public void work() {
    for (int i = 0; i < n; i++) {
      this.keeper.random();
    }
  }

  public void debug() {
    System.out.printf("%d %.2f %d\n", this.n, Math.log(this.n) / Math.log(2), this
  }
}

public static void main(String[] args) {
  for (int i = 1000; i < 100000; i += 100) {
    Experiment exp = new Experiment(i);
    exp.work();
    exp.debug();
  }
}

}
```

运行观察输出：

```
36400 15.15 13
36500 15.16 16
36600 15.16 13
36700 15.16 14
36800 15.17 15
36900 15.17 18
37000 15.18 16
37100 15.18 15
37200 15.18 13
37300 15.19 14
37400 15.19 16
```

```
37500 15.19 14
37600 15.20 15
```

通过这实验可以发现 K 和 N 的对数之间存在显著的线性相关性：

```
N=2^K  # 约等于
```

如果 N 介于 2^K 和 2^(K+1) 之间，用这种方式估计的值都等于 2^K，这明显是不合理的。这里可以采用多个 BitKeeper，然后进行加权估计，就可以得到一个比较准确的值。

```
import math
import random

def low_zeros(value):
    for i in xrange(1, 32):
        if value >> i << i != value:
            break
    return i - 1


class BitKeeper(object):

    def __init__(self):
        self.maxbits = 0

    def random(self, m):
        bits = low_zeros(m)
        if bits > self.maxbits:
            self.maxbits = bits


class Experiment(object):

    def __init__(self, n, k=1024):
        self.n = n
        self.k = k
        self.keepers = [BitKeeper() for i in range(k)]

    def do(self):
        for i in range(self.n):
            m = random.randint(0, 1<<32-1)
            # 确保同一个整数被分配到同一个桶里面，摘取高位后取模
            keeper = self.keepers[((m & 0xfff0000) >> 16) % len(self.keepers)]
            keeper.random(m)
```

```
    def estimate(self):
        sumbits_inverse = 0  # 零位数倒数
        for keeper in self.keepers:
            sumbits_inverse += 1.0/float(keeper.maxbits)
        avgbits = float(self.k)/sumbits_inverse  # 平均零位数
        return 2**avgbits * self.k  # 根据桶的数量对估计值进行放大


for i in range(100000, 1000000, 100000):
    exp = Experiment(i)
    exp.do()
    est = exp.estimate()
    print i, '%.2f' % est, '%.2f' % (abs(est-i) / i)
```

下面是 Java 版：

```
public class PfTest {

  static class BitKeeper {
    private int maxbits;

    public void random(long value) {
      int bits = lowZeros(value);
      if (bits > this.maxbits) {
        this.maxbits = bits;
      }
    }

    private int lowZeros(long value) {
      int i = 1;
      for (; i < 32; i++) {
        if (value >> i << i != value) {
          break;
        }
      }
      return i - 1;
    }
  }

  static class Experiment {
    private int n;
    private int k;
    private BitKeeper[] keepers;

    public Experiment(int n) {
      this(n, 1024);
    }
```

```
    public Experiment(int n, int k) {
      this.n = n;
      this.k = k;
      this.keepers = new BitKeeper[k];
      for (int i = 0; i < k; i++) {
        this.keepers[i] = new BitKeeper();
      }
    }

    public void work() {
      for (int i = 0; i < this.n; i++) {
        long m = ThreadLocalRandom.current().nextLong(1L << 32);
        BitKeeper keeper = keepers[(int) (((m & 0xfff0000) >> 16) % keepers.length)];
        keeper.random(m);
      }
    }

    public double estimate() {
      double sumbitsInverse = 0.0;
      for (BitKeeper keeper : keepers) {
        sumbitsInverse += 1.0 / (float) keeper.maxbits;
      }
      double avgBits = (float) keepers.length / sumbitsInverse;
      return Math.pow(2, avgBits) * this.k;
    }
  }

  public static void main(String[] args) {
    for (int i = 100000; i < 1000000; i += 100000) {
      Experiment exp = new Experiment(i);
      exp.work();
      double est = exp.estimate();
      System.out.printf("%d %.2f %.2f\n", i, est, Math.abs(est - i) / i);
    }
  }

}
```

代码中分了 1024 个桶，计算平均数使用了调和平均 (倒数的平均)。普通的平均法可能因为个别离群值对平均结果产生较大的影响，调和平均可以有效平滑离群值的影响。

avg=(3+4+5+104)/4=29

avg=4/(1/3+1/4+1/5+1/101)=5.044

观察脚本的输出，误差率控制在百分比个位数：

```
100000 97287.38 0.03
200000 189369.02 0.05
300000 287770.04 0.04
400000 401233.52 0.00
500000 491704.97 0.02
600000 604233.92 0.01
700000 721127.67 0.03
800000 832308.12 0.04
900000 870954.86 0.03
1000000 1075497.64 0.08
```

真实的 HyperLogLog 要比上面的示例代码更加复杂一些，也更加精确一些。上面的这个算法在随机次数很少的情况下会出现除零错误，因为 maxbits=0 是不可以求倒数的。

pf 的内存占用为什么是 12k?

py

我们在上面的算法中使用了 1024 个桶进行独立计数，不过在 Redis 的 HyperLogLog 实现中用到的是 16384 个桶，也就是 2^14，每个桶的 maxbits 需要 6 个 bits 来存储，最大可以表示 maxbits=63，于是总共占用内存就是 `2^14 * 6 / 8 = 12k` 字节。

思考 & 作业

尝试将一堆数据进行分组，分别进行计数，再使用 pfmerge 合并到一起，观察 pfcount 计数值，与不分组的情况下的统计结果进行比较，观察有没有差异。

扩展阅读

- HyperLogLog 复杂的公式推导请阅读 Count-Distinct Problem，如果你的概率论基础不好，那就建议不要看了（另，这个 PPT 需要翻墙观看）。