面试官：会玩牌吧？给我讲讲洗牌算法和它的应用场景吧！

程序员面试 1周前

以下文章来源于程序猿石头 ，作者码农唐磊

程序猿石头
程序猿@阿里云，清华学渣，前大疆后端Leader。用不同的视角分享高质量技术文章，...

有一次参加面试，面试官问我：“会玩牌吧？”

内心：“咋滴，这是要玩德州扑克（或者炸金花），赢了他就能通过面试么？”

结果……

没想到面试官的下一句话：“给我讲讲洗牌算法以及它的应用场景吧！”

哈哈，以上内容纯属虚构

背景

这确实也是一道面试题，我曾经多次面试中都有遇到这个题目或者这个题目的变种。

你不妨花 1 秒，想想？

什么是洗牌算法

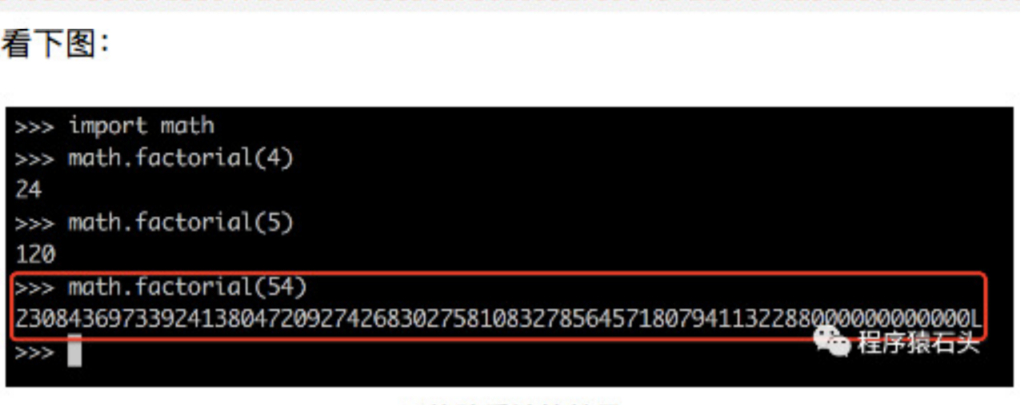
从名字上来看，就是给你一副牌让你洗呗，用怎样的方法才能洗得均匀呢？请大佬表演一下。

不好意思，翻车了

其实洗牌算法就是一种随机算法，你在斗地主的时候，随机把牌的顺序打乱就行。一个足够好的洗牌算法最终结果应该是可以让牌的顺序足够随机。好像有点绕~

这么来说吧，一副牌大家斗地主的话用 54 张（不考虑你们打配配牌的情形哈），那么这 54 张牌的顺序的话，按照排列组合算法，应该是有 `54!` 这么多种，然后你的洗牌算法就是从这 `54!` 种排列中，随机选 1 种。

无聊的石头算了一下，54 的阶乘有多大呢？大概就是这么大一长串数字，`230843697339241380472092744868302758108327856457180794113228800000000000000L`，准确答案看下图：



```
>>> import math
>>> math.factorial(4)
24
>>> math.factorial(5)
120
>>> math.factorial(54)
230843697339241380472092742683027581083278564571807941132288000000000000L
>>> 
```

54的阶乘计算结果

我们还是以 4 张牌作为例子吧。

4 张牌，`JQKA`，所有的排列有 `4!=4*3*2*1=24` 种，分别如下：

```
('J', 'Q', 'K', 'A')
('J', 'Q', 'A', 'K')
('J', 'K', 'Q', 'A')
('J', 'K', 'A', 'Q')
('J', 'A', 'Q', 'K')
('J', 'A', 'K', 'Q')
('Q', 'J', 'K', 'A')
('Q', 'J', 'A', 'K')
('Q', 'K', 'J', 'A')
('Q', 'K', 'A', 'J')
('Q', 'A', 'J', 'K')
('Q', 'A', 'K', 'J')
('K', 'J', 'Q', 'A')
('K', 'J', 'A', 'Q')
('K', 'Q', 'J', 'A')
('K', 'Q', 'A', 'J')
('K', 'A', 'J', 'Q')
('K', 'A', 'Q', 'J')
('A', 'J', 'Q', 'K')
('A', 'J', 'K', 'Q')
('A', 'Q', 'J', 'K')
('A', 'Q', 'K', 'J')
('A', 'K', 'J', 'Q')
('A', 'K', 'Q', 'J')
```

那么，一个均匀的洗牌算法，就是每次洗牌完后，获得上面每种顺序的概率是相等的，都等于 `1/24`。感觉已经出来了一种算法了，那就是先像前文所述把所有的排列情况都枚举出来，分别标上号 1-24 号，然后从 24 中随机取一个数字（先不考虑如何能做到随机取了，这个话题好像也没那么容易），获取其中这个数字对应的号的排列。

这个算法复杂度是多少？假设为 `N` 张牌的话，应该就是 `1/N!`（注意是阶乘，这里可不是感叹号），显然复杂度太高了。

有没有更好的方法呢？答案当然是肯定的。

经典的洗牌算法

洗牌算法实际上是一个很经典的算法，在经典书籍《算法导论》里面很靠前的部分就有讲解和分析。

我们把这个洗牌过程用更加“程序员”的语言描述一下，就是假设有一个 `n` 个元素的数组 `Array[n]`，通过某种方式，随机产生一个另外一个序列 `Array'[n]` 让数组的每个元素 `Array[i]` 在数组中的每个位置出现的概率都是 `1/n`。

其实方法可以这样，依次从 `Array` 中随机选择 1 个，依次放到 `Array'` 中即可。证明一下：

- `Array[0]`，在新数组的第 0 个位置处的概率为：`1/n`，因为随机选，只能是 `1/n` 的概率能选中；
- `Array[1]`，在新数组的第 1 个位置处的概率为：`1/n`，因为 第一次没选中 `Array[1]` 的概率为 `n-1/n`，再结合第二次（只剩下n-1个了，所以分母为 `n-1`）选中的概率为：`1/n-1`，因此概率为：$\frac{n-1}{n} * \frac{1}{n-1} = \frac{1}{n}$。
- 依此类推，可以证明前述问题。

其实，我们也可以不用另外找个数组来存结果，`Array'` 也可以理解为还是前面的这个 `Array`，只不过里面元素的顺序变化了。

这其实可以理解为一个 “排序”（其实是乱序） 过程，算法如下：

```
void shuffle(List list) {
  int n = list.size();
  for (int i = 0; i < n; i++) {
    int j = random(i, n); // 随机产生 [i, n) 中的一个数，每个概率一致
    // list 中第 i 个元素和 第 j 个元素互换位置
    swap(list[i], list[j]);
  }
}
```

接下来是如何证明呢？不能你说随机就随机吧，你说等概率就等概率吧。下面还是跟着石头哥一起来看看如何证明吧（这也是面试中的考察点）。

我们假设经过排序后，某个元素 `Array[x]` 恰好排在位置 `x` 处的概率为 P_x，则该元素恰好排在第 `x` 处的概率是前 `x-1` 次时都没有被随机到，并且第 `x` 次时，恰好 `random(x, n) = x` 了。

还是在循环中列举几项，更好理解一些（写完，才发现跟前面的解释差不多）：

- `i = 0`，`random(0, n)` 没有返回 x，共 `n` 个数，肯定返回了其他 `n-1` 个中的一个，因此概率为 $\frac{n-1}{n}$
- `i = 1`，`ramdom(1, n)` 没有返回 x，共 `n - 1` 个数，肯定返回了其他 `n-2` 个中的一个，即该为 $\frac{n-2}{n-1}$
- 依此类推……
- `i = x-1`，`random(x-1, n)` 没有返回 x，共 `n - (x-1)` 个数，肯定返回了其他 `n-(x-1)-1` 个中的一个，即为 $\frac{n-(x-1)-1}{n-(x-1)} = \frac{n-x}{n-x+1}$
- `i = x`，`random(x, n)` 恰好返回了 x，共 `n-x` 个数，概率为 $\frac{1}{n-x}$

$$p_x = \frac{n-1}{n} * \frac{n-2}{n-1} * \ldots * \frac{n-x}{n-x+1} * \frac{1}{n-x} = \frac{1}{n}$$

因此，到这算是简单证明了任何元素出现在任何位置的概率是相等的。

注意说明一下，这是理论上的值，概率类的问题在量不大的情况下很有可能有随机性的。就像翻硬币，正反面理论上的值都是一半一半的，但你不能说一定是正反面按照次序轮着来。

看看 JDK 中的实现

我们还是来看看 JDK 中的实现。JDK 中 `Collections` 中有如下的实现方法 `shuffle`：

```
public static void shuffle(List<?> list, Random rnd) {
    int size = list.size();
    // 石头备注：本机特定版本中的常量 SHUFFLE_THRESHOLD=5
    if (size < SHUFFLE_THRESHOLD || list instanceof RandomAccess) {
        for (int i=size; i>1; i--)
            swap(list, i-1, rnd.nextInt(i));
    } else {
        Object arr[] = list.toArray();
        // Shuffle array
        for (int i=size; i>1; i--)
            swap(arr, i-1, rnd.nextInt(i));
        ListIterator it = list.listIterator();
        for (int i=0; i<arr.length; i++) {
            it.next();
            it.set(arr[i]);
        }
    }
}
```

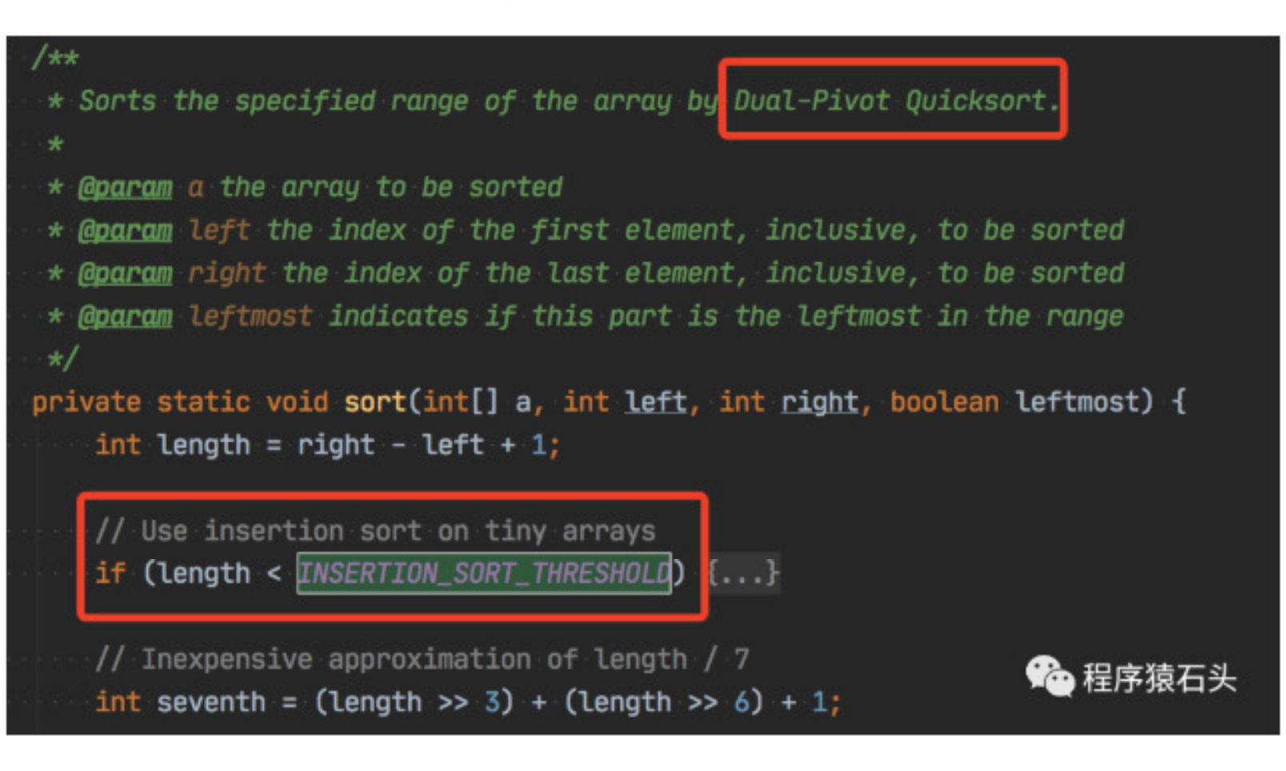
基本上能看懂大概，不过是不是看看源码还是能获得新技能的。

上面条件分支大概分两类：

- 如果是数组类型，就是可以 `O(1)` 随机访问的List；或者传入的 list 小于 `SHUFFLE_THRESHOLD`。
- 否则的话不能随机访问的链表类型，则花 `O(n)` 转成数组，再 shuffle，最后又回滚回链表。转成数组的目的很简单，可以快速定位某个下标的元素。

第一步的这个 `SHUFFLE_THRESHOLD` 其实就是一个经验调优值，即便假设不能通过快速下标定位某个元素（即需要遍历的方式定位），当输入的 size 比较小的时候，和先花 `O(n)` 转成数组最后又转回成链表 相比，也能有更快的速度。

另外多说一句，其实这种参数化调优方式在各种语言实现的时候很常见的，比如你去看排序算法的实现中，比如 Java 中 `Arrays.sort` 就是用的 `DualPivotQuicksort`（源码在 `java.util.DualPivotQuicksort` 中），里面实现逻辑中，当数组大小较小时也是用的其他如 $O(n^2)$ 的插入排序，如下图所示。



```
/**
 * Sorts the specified range of the array by Dual-Pivot Quicksort.
 *
 * @param a the array to be sorted
 * @param left the index of the first element, inclusive, to be sorted
 * @param right the index of the last element, inclusive, to be sorted
 * @param leftmost indicates if this part is the leftmost in the range
 */
private static void sort(int[] a, int left, int right, boolean leftmost) {
    int length = right - left + 1;

    // Use insertion sort on tiny arrays
    if (length < INSERTION_SORT_THRESHOLD) {...}

    // Inexpensive approximation of length / 7
    int seventh = (length >> 3) + (length >> 6) + 1;
```

洗牌算法的应用

~~肝到 凌晨 2 点了，明天继续写吧....~~

第二天继续肝

回到本篇标题说的应用场景上来，比如开篇提到的 Eureka 注册中心的 Client 就是通过把 server 的 IPList 打乱顺序，然后挨个取来实现理论上的均匀的负载均衡。

代码（在 github: Netflix/eureka 中，公众号就不单独贴出来了）在这里 `com.netflix.discovery.shared.resolver.ResolverUtils`。看代码如下，是不是跟前文的算法差不多？（具体写法不一样而已）

```
public static <T extends EurekaEndpoint> List<T> randomize(List<T> list) {
    List<T> randomList = new ArrayList<>(list);
    if (randomList.size() < 2) {
        return randomList;
    }
    Random random = new Random(LOCAL_IPV4_ADDRESS.hashCode());
    int last = randomList.size() - 1;
    for (int i = 0; i < last; i++) {
        int pos = random.nextInt(randomList.size() - i);
        if (pos != i) {
            Collections.swap(randomList, i, pos);
        }
    }
    return randomList;
}
```

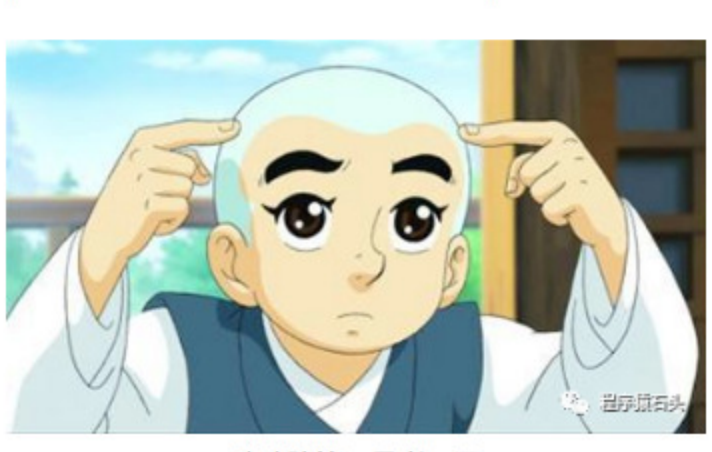
其实，在任何需要打乱顺序的场景里面都可以用这个算法，举个例子，播放器一般都有随机播放的功能，比如你自己有个歌单 list，但想随机播放里面的歌曲，就也可以用这个方法来实现。

还有，就比如名字中的“洗牌”，那些棋牌类的游戏，当然会用到名副其实的“洗牌”算法了。其实在各种游戏的随机场景中应该都可以用这个算法的。

扩展一下，留道作业题

跟这个问题类似的，还有一些常见的面试题，本人之前印象中也被问到过（石头特地去翻了翻当年校招等找工作的时候收集和积累的面试题集）。

以下题目来源于网络，因为是当初准备面试时候收集的，具体来源不详了。



动动脑筋，思考一下

题目 1

给你一个文本文件，设计一个算法随机从文本文件中抽取一行，要保证每行被抽取到的概率一样。

最简单的思路其实就是：先把文件每一行读取出来，假设有 `n` 行，这个时候随机从 `1-n` 生成一个数，读取对应的行即可。

这种方法当然可以解决，咱们加深一下难度，假设文件很大很大很大呢，或者直接要求只能遍历该文件内容一遍，怎么做到呢？

题目 2

其实题目 1 还可以扩展一下，不是选择 1 行了，是选择 `k` 行，又应该怎么做呢？

好，文章结束了。本人才疏学浅，如果有不对的地方，还望大家指出。