# 蓄水池算法改进 - 面向抽奖场景保证等概率性

---

免责声明：禁止任何个人或团体使用本文研究成果用于实施任何违反中华人民共和国法律法规的活动 如有违反，均与本文作者无关

---

在我们通常遇到的抽奖场景，于年会时将所有人的编号都放到箱子里面抽奖，然后每次抽出中奖者 决定奖项。而在这过程中，因为先抽中者已经确定了奖项，然后不能够参与后续的奖项的抽奖；而后 续参与抽奖的人员则其实会以越来越低的概率参与抽奖:

---

例：在上述场景中共有 $n$ 人参与抽取 $m(< n)$ 个奖项，

抽取第一个奖项概率为： $\frac{m}{n}$

那么如果第一个奖项被抽走并 *揭露了*，剩下 $n-1$ 人参与 $m-1$ 个奖项，抽中的概率 为 $\frac{m-1}{n-1}$。 那么 $m < n \Rightarrow -m > -n \Rightarrow mn - m > nm - n \Rightarrow m(n-1) > n(m-1) \Rightarrow \frac{m}{n} > \frac{m-1}{n-1}$ ， 即如果是后续参与抽奖 *并且前面的奖项被拿走了*，后面抽到奖项的概率会更低，**同时也会失去参与部分奖项的机会**。

---

因此，在人数 $n$ 大于奖项数 $m$ 的时候，我们通过以越来越低的概率干涉前面 已经“取得”奖项的结果，来保证先参与抽奖的人中奖的概率随着人数的增多中奖的概率也变低， 最后保证每个人中奖的概率为 $\frac{m}{n}$。但是在实际场景中，$m$ 个奖项可能 不仅相同（如划分了一二三等奖），因此对于蓄水池算法的改进提出了新的要求：

- 将所有的奖项视为各不相同的位置，不论人数多少（当还是要保证有人来参与抽奖 $n > 1$ )所有人占有特定位置的概率相同
- ~~每当新来一人参与抽奖时，如果他没有中奖，可以即场告知未中~~[1]

## 算法描述与等概率性证明

我们分两种情况讨论：

- 一种是当人数不足以覆盖所有的奖项的场景（ $n < m$ ），
- 另外一种是当抽奖人数远大于所有奖项加起来的数目。（ $n > m$ ）。

然后我们再回来看看能不能找到一种很方便的方法桥接两种情况。

同时，我们假设 $m$ 个奖项两两互不相同。

### 抽奖人数不足时（ $n < m$ ）

因为当人数不足时，所有参与者都能抽奖，因此我们要保证每个人获得特定奖项的概率为 $\frac{1}{m}$。 算法描述：

---

记 $\mathrm{Choosen}$ 为容量为 $m$ 的数组， $\mathrm{Choosen[k]}(1 \le k \le m)$ 表示第 k 个奖项的当前占有情况， 初始值为 $\mathrm{None}$ ,

$\mathrm{Players}$ 为参与参与抽奖的人的序列

1. 令 $i := 1$ ，当 $i \le n$ 时，做如下操作：
   - 产生随机数 $r_1(1 \le r_1 \le i)$
   - 如果 $r_1 < i$ ， $\mathrm{Choosen}[i] := \mathrm{Choosen}[r_1]$
   - $\mathrm{Choosen}[r_1] := \mathrm{Players}[i]$
   - $i := i + 1$
2. 当 $i \le m$ 时，做如下操作：

- 产生随机数 $r_2(1 \le r_2 \le i)$
- 如果 $r_2 < i$ :
  - $Choosen[i] := Choosen[r_2]$
  - $Choosen[r_2] := None$
- $i := i + 1$

---

**等概率性证明**

我们先证明，在填入中奖者的第 $k(1 \le k \le m)$ 轮过程中，能够保证对于前 $k$ 个奖项中的每一个奖项，每一位中奖者抽中其中第 $i(1 \le i \le k)$ 个奖项的概率为 $\frac{1}{k}$，证明如下：

我们采用数学归纳法来证明：

1. **奠基**：当 $k = 1$ 时，易知该中奖者一定会抽中第一个奖项，前一个奖项中只有第一个 选项，所以此时每一位中奖者抽中第 $k = 1$ 的概率为 $1 = \frac{1}{1} = \frac{1}{k}$ ；
2. **归纳**:
   - 假设当 $k = j(1 \le j < m)$ 时，每一位抽奖者抽中第 $i(1 \le i \le j)$ 的概率为 $\frac{1}{j}$
   - 当 $k = j + 1$ , 有：
     - 第 $j + 1$ 位抽奖着抽中任意第 $i'(1 \le i' \le j + 1)$ 个奖项的概率为 $\frac{1}{j+1}$ （假设产生的随机数 $r_1$、$r_2$ 足够的均匀）；
     - 对于前 $j$ 位抽奖者，每一位都有 $\frac{1}{j+1}$，的概率将自己的奖项更换位第 $j + 1$ 个奖项；
     - 对于前 $j$ 位抽奖者，每一位依然占有原有第 $i'$ 个奖项的概率为：

$$
\begin{aligned}
P\{\text{前j位抽奖者j+1轮中仍然持有}i'\} &= P\{\text{前j位抽奖者j轮已经持有}i'\} \cdot P\{\text{第j+1位抽奖者没有抽中}i'\} \\
&= P\{\text{前j位抽奖者j轮已经持有}i'\} \cdot (1 - P\{\text{第j+1位抽奖者抽中}i'\}) \\
&= \frac{1}{j} \cdot (1 - \frac{1}{j+1}) \\
&= \frac{1}{j} \cdot \frac{j}{j+1} \qquad (1.1) \\
&= \frac{1}{j+1} \\
&= \frac{1}{k}
\end{aligned}
$$

由上，可知每一轮迭代之后，前 $k$ 个奖项对于已经参与的 $k$中奖者来说抽中的概率均等，为 $\frac{1}{k}$， 故到了第 $n$ 轮操作后，我们可以通过不断填充 $None$ 值来稀释概率，最后达到 $\frac{1}{m}$ 的等概率性。

特殊地，当 $n == m$ 时，每个抽奖者抽到特定奖项的概率也为 $\frac{1}{n}$。

## 抽奖人数足够多时（ $n > m$ ）

类似地，当 $n > m$ 时，对于每一个抽奖序号 $k > m$ 的抽奖者，我们生成随机数 $r_3(1 \le r_3 \le n)$ ，并且在 $r_3 \le m$ 的时候，替换对应原本占有奖项的抽奖者；可以证明在这种情况下，能保证每个人抽到特定奖项的概率为 $\frac{1}{n}$[2]。

## 整合后的算法

---

记 $Choosen$ 为容量为 $m$ 的数组， $Choosen[k](1 \le k \le m)$ 表示第 $k$ 个奖项的当前占有情况， 初始值为 $None$ ,

$replaced$ 为原本已经中奖，但是被人替换的抽奖者

$Players$ 为参与参与抽奖的人的序列，每次只能获取一个 player

记 $n := 0$ 为当前参与抽奖的人数

1. 在抽奖结束前，每次遇到一个新的 player 执行以下操作：
    - $replaced := None$
    - $n := n + 1$
    - 产生随机数 $r(1 \le r \le n)$
    - 如果 $r \le m$ ：
        - $replaced := Choosen[r]$
        - $Choosen[r] := player$
    - 如果 $r < n$ 并且 $n \le m$ ：
        - $Choosen[n] := replaced$
2. 在抽奖结束时，如果 $n < m$ , 执行以下操作：
    - $i := n$
    - 当 $i < m$ 时，重复执行以下操作：
        - $i := i + 1$
        - 产生随机数 $r_2(1 \le r_2 \le i)$
        - 如果 $r_2 < i$ :
            - $Choosen[i] := Choosen[r_2]$
            - $Choosen[r_2] := None$

---

# 程序实现

## Rust

作者偏好 Rust 编程语言，故使用 Rust 实现。

### 特质（trait）

Rust 中的特质（trait） 是其用于复用行为抽象的特性，尽管比起 Java 或 C# 的接口 （Interface）更加强大，但在此文中，熟悉 Java/C# 的读者把特质视作接口就可以了。

## 建模与实现

本文使用面向对象（Object-Oriented）编程范式[3]来进行抽象，如下所示：

```
use rand::random;

use std::fmt::Debug;

trait ReservoirSampler {
    // 每种抽样器只会在一种总体中抽样，而总体中所有个体都属于相同类型
    type Item;

    // 流式采样器无法知道总体数据有多少个样本，因此只逐个处理，并返回是否将样本纳入
    // 样本池的结果，以及可能被替换出来的样本
    fn sample(&mut self, it: Self::Item) -> (bool, Option<Self::Item>);

    // 任意时候应当知道当前蓄水池的状态
    fn samples(&self) -> &[Option<Self::Item>];
}

struct Lottery<P> {
    // 记录当前参与的总人数
    total: usize,

    // 奖品的名称与人数
```

```
        prices: Vec<Price>,

        // 当前的幸运儿
        lucky: Vec<Option<P>>,
    }

    #[derive(Clone, Debug)]
    struct Price {
        name: String,
        cap: usize,
    }

    impl<P> ReservoirSampler for Lottery<P> {
        type Item = P;

        fn sample(&mut self, it: Self::Item) -> (bool, Option<Self::Item>) {
            let lucky_cap = self.lucky.capacity();

            self.total += 1;

            // 概率渐小的随机替换
            let r = random::<usize>() % self.total + 1;
            let mut replaced = None;
            if r <= lucky_cap {
                replaced = self.lucky[r - 1].take();
                self.lucky[r - 1] = Some(it);
            }

            if self.total <= lucky_cap && r < self.total {
                self.lucky[self.total - 1] = replaced.take();
            }

            (r <= lucky_cap, replaced)
        }

        fn samples(&self) -> &[Option<Self::Item>] {
            &self.lucky[..]
        }
    }

    impl<P: Debug> Lottery<P> {
        fn release(self) -> Result<Vec<(String, Vec<P>)>, &'static str> {
            let lucky_cap = self.lucky.capacity();

            if self.lucky.len() == 0 {
                return Err("No one attended to the lottery!");
            }

            let mut final_lucky = self.lucky.into_iter().collect::<Vec<Option<P>>>();
            let mut i = self.total;
            while i < lucky_cap {
                i += 1;

                // 概率渐小的随机替换
                let r = random::<usize>() % i + 1;
                if r <= lucky_cap {
                    final_lucky[i - 1] = final_lucky[r - 1].take();
                }
            }
            println!("{:?}", final_lucky);

            let mut result = Vec::with_capacity(self.prices.len());
            let mut counted = 0;
            for p in self.prices {
                let mut luck = Vec::with_capacity(p.cap);

                for i in 0 .. p.cap {
                    if let Some(it) = final_lucky[counted + i].take() {
                        luck.push(it);
                    }
                }

                result.push((p.name, luck));
                counted += p.cap;
            }
```

```
            Ok(result)
        }
    }

    // 构建者模式（Builder Pattern），将所有可能的初始化行为提取到单独的构建者结构中，以保证初始化
    // 后的对象(Target)的数据可靠性。此处用以保证所有奖品都确定后才能开始抽奖
    struct LotteryBuilder {
        prices: Vec<Price>,
    }

    impl LotteryBuilder {
        fn new() -> Self {
            LotteryBuilder {
                prices: Vec::new(),
            }
        }

        fn add_price(&mut self, name: &str, cap: usize) -> &mut Self {
            self.prices.push(Price { name: name.into(), cap });
            self
        }

        fn build<P: Clone>(&self) -> Lottery<P> {
            let lucky_cap = self.prices.iter()
                .map(|p| p.cap)
                .sum::<usize>();

            Lottery {
                total: 0,
                prices: self.prices.clone(),
                lucky: std::vec::from_elem(Option::<P>::None, lucky_cap),
            }
        }
    }

    fn main() {
        let v = vec![8, 1, 1, 9, 2];
        let mut lottery = LotteryBuilder::new()
            .add_price("一等奖", 1)
            .add_price("二等奖", 1)
            .add_price("三等奖", 5)
            .build::<usize>();


        for it in v {
            lottery.sample(it);
            println!("{:?}", lottery.samples());
        }

        println!("{:?}", lottery.release().unwrap());
    }
```

## 优点

- 流式处理，可以适应任意规模的参与人群
- 在保证每一位抽奖者都有相同的概率获得特定奖项的同时，还能保证每一个抽奖者的获得的奖项均不相同

## 缺点

- 所有参与抽奖的人都必须**依次**经过服务器处理，因为需要获知准确的总人数来保证等概率性。 一个改进的方法是，在人数足够多的时候，将总人数用总人数的特定数量级替代（给后续参加者的 一点点小福利——但是因为总人数足够多，所以总体中奖概率还是很低），在客户端完成中奖的选定
- **等概率性完全依赖随机数 `r` 生成**。 因为奖品初始化时不需要考虑打乱顺序，因此如果在 随机这一步被技术破解，使得抽奖者可以选择自己能获取的奖项，则会破坏公平性。改进方案是， 在 `release` 的时候再一次对奖品顺序进行随机的

打乱。
- 这种抽奖方式还限定了每人只能抽取一次奖品，否则会出现一个人占有多个奖项的情况。

## 下一步可能展开的工作

目前所有抽奖者都按照相等的概率抽奖，而在一些场景下可能按照一些规则给与某些抽奖者优惠 （例如绩效越高的员工中奖概率越大），因此下一步可能考虑如何按照权重赋予每位抽奖者各自的 中奖概率。

## 致谢

感谢茶壶君（@ksqsf）一语惊醒梦中人，清楚明确地表达了需求； 感谢张汉东老师 (@ZhangHanDong)老师提点了之后可以开展研究的方向； 感谢在这次讨论中提供意见的其他 Rust 社区的朋友，谢谢你们!

---

1. 该条件为用以减轻开奖时发通知的压力，并非核心需求，因为对参与抽奖的玩家负责的原因，我们还是需要储存每个玩家每次的抽奖情况信息 ↩
2. 可以参考博主以前的博客 ↩
3. 作者理解的面向对象 = 对象是交互的最基本单元 + 对象通过相互发送消息进行交互。而特质/接口以及对象其他公开的方法定义了对象可以向外发送/从外接收的消息。 ↩