# 数据与算法的故事: 3

吕正华

2019.10

# 自我介绍

- Pivotal 资深软件工程师，开发 Greenplum 内核
- kainwen@gmail.com
- 个人主页: https://kainwen.com
- 2014 年毕业于电子系智能感知实验室, 工学硕士
- 2012 年第一次担任数据与算法助教

# 今天的话题

# 生日悖论

生日悖论，指如果一个房间里有 23 个或 23 个以上的人，那么至少有两个人的生日相同的概率要大于 50%。这就意味着在一个典型的标准小学班级 (30 人) 中，存在两人生日相同的可能性更高。对于 60 或者更多的人，这种概率要大于 99%。从引起逻辑矛盾的角度来说生日悖论并不是一种悖论，从这个数学事实与一般直觉相抵触的意义上，它才称得上是一个悖论。大多数人会认为，23 人中有 2 人生日相同的概率应该远远小于 50%。计算与此相关的概率被称为生日问题，在这个问题之后的数学理论已被用于设计著名的密码攻击方法：生日攻击。

摘自百度百科

# 生日问题

- 模型：$M$ 个空盒子，有 $N$ 个球随机的放进去，没有任何盒子多于一个球
- 生日模型就是 $M = 365$
- 如果我们知道了上面事件的概率 $Pr_M(N)$，那么平均等候的长度是 $\sum_{N\geq 0}^{\infty} Pr_M\{N\}$
- 这次采用 EGF 的技术解决: $B_M = SEQ_M(E + Z)$
- $B_M(z) = (1+z)^M \implies B_M(N) = N!\binom{M}{N}$
- $Pr_M(N) = \frac{M!}{M^N(M-N)!}$
- 平均等候长度 $= 1 + Q(M) \sim \sqrt{\frac{\pi M}{2}}$
- Thank you, Sir Ramanujan & Sir Laplace

# 拉马努金

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{99^2}\sum_{m=0}^{\infty}(26390m+1103)\frac{(4m)!}{396^{4m}(m!)^4}$$

Figure: 拉马努金 $\pi$ 级数公式

- 电影《知无涯者》

# 选择抽样算法

在上一段中提出的思想立即引出了下列算法:

**算法 S**(选择抽样技术) 从 $N$ 个记录的一个集合中随机地选择 $n$ 个记录,其中 $0 < n \leqslant N$。

**S1.**[初始化] 置 $t \leftarrow 0, m \leftarrow 0$。(在本算法中,$m$ 表示已选得的记录数,而 $t$ 表示我们已经处理过的输入记录的总数。)

**S2.**[生成 $U$] 生成在 0 与 1 之间一致地分布的一个随机数 $U$。

**S3.**[检验] 如果 $(N-t)U \geqslant n-m$,则转到步骤 S5。

**S4.**[选择] 把下一个记录选为样本,$m$ 和 $t$ 加 1。如果 $m < n$,则转到步骤 S2;否则取样完成,算法终止。

**S5.**[跳] 跳过下一个记录(不把它选为样本),$t$ 加 1,并转到步骤 S2。▌

乍一看,这个算法可能显得不可靠而且简直是不正确的;但经过细心的分析(见下面的习题),证明它是完全信赖的。不难验证:

a)至多输入 $N$ 个记录(在选择 $n$ 个项目之前,我们绝不会跑出文件的末尾)。

b)这个抽样是完全无偏倚的;特别是,任何给定的元素(例如文件的最后元素)被选择的概率是 $n/N$。

虽然有这样的事实:即我们不以概率 $n/N$,而是以等式(1)中的概率选择第 $t+1$ 项,但命题 b)是真的! 这在发表的著作中引起了某些混乱。读者能否说明这种表面上的矛盾?

Figure: 选择抽样算法摘自 TAOCP

**Q:** 怎么证明算法是正确的?

# 伪代码

```
/*
 * SelectSample : List -> int -> int -> List
 * - Data: the dataset to sample from, index start from 1
 * - N   : the number of elements of data in Data
 * - n   : we sample exact n elements from Data
 * - 0 < n <= N
 */
List select_sample(List Data, int N, int n)
{
    Assert(0 < n && n <= N);
    List result = new List;
    int i = 1;
    int already_scanned = 0;
    int already_picked = 0;
    while(true) {
        int remain_to_pick = n - already_picked;
        int remain_to_scan = N - already_scanned;
        if (remain_to_pick = 0)
            return result;
        double u = random_uniform(0, 1);
        if (remain_to_scan * u < remain_to_pick) {
            //pick the element Data[i]
            result.append(Data[i]);
            already_picked++;
            already_scanned++;
        } else {
            already_scanned++;
        }
    }
}
```

# 理解算法和证明算法的关键: 循环不变的模式

- loop invariant: 算法导论中证明算法的技术, Induction
- The more things change, the more they stay the same
- $S = select\_sample(Data, N, n) \implies length(S) = n$
- $\forall 1 \leq i \leq N, Pr(Data[i] \in select\_sample(Data, N, n)) = \frac{n}{N}$
- 我们集中精力证明第二条，第一条留给大家当作业

# 从小的具体例子做起

- $i = 1$ 的时候，第一个元素被即将扫描到，此时还剩 $N$ 个需要扫描，还剩 $n$ 个需要挑选出来
- 根据算法，它被选择出来的概率是 $\frac{n}{N}$，满足我们需要证明的
- $i = 2$ 的时候，第二个元素即将被扫描出来，此时还剩 $N - 1$ 个需要扫描，至于还剩多少要挑出来，依赖于历史情况
  - 第一个元素被选中且第二个被选中：$\frac{n}{N} * \frac{n-1}{N-1}$
  - 第一个元素没选中且第二个被选中：$\frac{N-n}{N} * \frac{n}{N-1}$
  - 还是满足需要证明的论断
- 更一般的情况呢？需要考虑所有历史的可能情况，用 $m$ 表示已经挑出的数目，$k$ 表示我们即将考察第 $k$ 个元素
- $0 \leq m < n$(不然算法就终止了)
- $P(data[k]\ be\ picked) = \sum_{m=0}^{n-1} P(already\ pick\ exact\ m) \frac{n-m}{N-k+1}$

# 一般情况的推理

- 某个元素选中造成的影响：还剩扫描的数目减一，还剩需要选出的数目也减一
- 某个元素没中选造成的影响：还剩扫描的数目减一，还剩需要选出的数目不变
- $P(already\ pick\ exact\ m) = \binom{k-1}{m} \frac{n^{\underline{m}}(N-n)^{\underline{k-1-m}}}{N^{\underline{k-1}}}$
- $P_k = \frac{n}{N} \frac{\sum_{m=0}^{n-1} \binom{k-1}{m} (n-1)^{\underline{m}} (N-n)^{\underline{k-1-m}}}{N^{\underline{k-1}}}$
- 代入下降幂的二项式公式上面的公式就得到了证明
- Hint1: $n^{\underline{n}} = n!, n^{\underline{n+1}} = 0$(参考具体数学第二章)
- Hint2: $\binom{3}{5} = 0, \binom{n}{m} = 0 (m > n)$(参考具体数学第五章)

# 什么是一致性哈希?

- 分布式系统数据被分片到不同的机器上，需要有分片的策略，很常用的是根据数据的哈希值，计算它所属的分片
- 分布式系统需要能快速扩容和缩容 (云计算环境下尤其如此)
- 扩容和缩容不希望数据库在老机器之间搬移，期待只能从老机器迁移到新机器 (单调性)
- Q: 还有哪个重要的性质是我们对一致性哈希算法想提出的要求？
- 常见的一致性哈希算法的技巧是用环

# Jump Consistent Hashing

- Google 的一个 paper 里提到的算法:
  A Fast, Minimal Memory, Consistent Hash Algorithm
- 精妙的思路:
  - 用数据本身作为伪随机数生成器的种子
  - 这样我们算法本身是确定的 (每次都保证数据被分片到同一个机器)
  - 同时我们又可以用概率的技术去分析算法
- 我们需要分析什么呢?
  - 单调性: $jump\_hash(data, N)$ 关于 $N$ 单调 (留给大家思考)
  - 均匀性: $jump\_hash(data, N)$ 等概率的落到任何一个机器 (数据分布均匀的前提)

# 线性扫描程序设计

```
int ch(int key, int N)
{
    random.seed(key);
    int b = 0; int i;
    for (i = 1; i < N; i++)
        if (random.next() < (1.0 / (i+1)))
            b = i;
    return b;
}
```

- 观察什么是循环不变 (思考每次循环的变换，抽象出不变的地方，领悟" 模式")
- 伪随机被初始化后，其生成的序列就确定了 $R_0, R_1, R_2, \ldots,$ 是周期的，只不过周期特别特别长
- 上述算法的本质：扫描前 $N$ 个伪随机数，对每个位置判断一个随机事件，如果事件发生了，记录下它的 index，返回最大的 index
- $p(ch(key, N) = i) = \ p(R_i < \frac{1}{i+1} \wedge R_j > \frac{1}{j+1} \forall i < j < N) = \frac{1}{i+1} \prod_{i+1}^{N-1} \frac{j}{j+1} = \frac{1}{N}$
- Q: 上述算法的时间复杂度是多少？

# 改善性能

- 线性复杂度太高了，我们期待对数复杂度
- 观察事实：新加机器后，某个老机器上的每个数据以一定的概率跳转 (这个概率是啥？)
- 事实的启发：每个机器上，只有少数数据，会发送上述算法的“事件”
- 进一步的思路：这些发生事件的 index，可以建模成一个随机过程么？马氏链？
- $p(i) = \sum_{j=0}^{i-1} p(j)p(j\_jump\_to\_i|i), p(i) = \frac{1}{i+1}$
- $p(j \rightarrow i) = \frac{j+1}{i(i+1)} \quad = p(\frac{j+1}{i+1} < r \leq \frac{j+1}{i}) = p(i = \lfloor \frac{j+1}{r} \rfloor)$
- Q: 时间复杂度怎么分析？Hint: 结合扫描的算法分析

# PHash VS Maglev

- Pivotal's NO.1 Engineer Heikki 觉得 Jump Consistent Hash 会引入额外的 CPU 开销，毕竟复杂度是 $log(n)$
- 他搜到了一个基于查找表的论文，Google 的磁悬浮列车一致性哈希算法，
  Maglev: A Fast and Reliable Software Network Load Balancer
- 我认为 Maglev 并不是优秀的一致性哈希算法
  - 首先不是单调的，且老机器之间的数据迁移的分析也没有量化
  - 它也不是最佳均匀的
  - 初始化过程过于复杂，且浪费空间
- Pivotal Hash: 闪现出一个《具体数学》的公式
- $n = \lceil \frac{n}{m} \rceil + \lceil \frac{n-1}{m} \rceil + \cdots + \lceil \frac{n-m+1}{m} \rceil = \sum_{k=0}^{k=m-1} \lceil \frac{n-k}{m} \rceil$

# 参考资料

- 个人主页: Thoughts on Computing
- 具体数学
- 算法分析导论

# 结束

谢谢！

Q&A