随机函数

概述

要想使用随机化技巧，前提条件是能够快速生成随机数。本文将介绍生成随机数的常见方法。

随机数与伪随机数

说一个单独的数是「随机数」是无意义的，所以以下我们都默认讨论「随机数列」，即使提到「随机数」，指的也是「随机数列中的一个元素」。

现有的计算机的运算过程都是确定性的，因此，仅凭借算法来生成真正 **不可预测**、**不可重复** 的随机数列是不可能的。

然而在绝大部分情况下，我们都不需要如此强的随机性，而只需要所生成的数列在统计学上具有随机数列的种种特征（比如均匀分布、互相独立等等）。这样的数列即称为 **伪随机数** 序列。

随机数与伪随机数在实际生活和算法中的应用举例：

- 抽样调查时往往只需使用伪随机数。这是因为我们本就只关心统计特征。
- 网络安全中往往要用到（比刚刚提到的伪随机数）更强的随机数。这是因为攻击者可能会利用可预测性做文章。
- OI/ICPC 中用到的随机算法，基本都只需要伪随机数。这是因为，这些算法往往是 通过引入随机数 来把概率引入复杂度分析，从而降低复杂度。这本质上依然只利用了随机数的统计特征。
- 某些随机算法（例如 Moser 算法）用到了随机数的熵相关的性质，因此必须使用真正的随机数。

实现

`rand`

用于生成伪随机数，缺点是比较慢，使用时需要 `#include<cstdlib>`。

调用 `rand()` 函数会返回一个 `[0,RAND_MAX]` 中的随机非负整数，其中 `RAND_MAX` 是标准库中的一个宏，在 Linux 系统下 `RAND_MAX` 等于 $2^{31}-1$。可以用取模来限制所生成的数的大小。

使用 `rand()` 需要一个随机数种子，可以使用 `srand(seed)` 函数来将随机种子更改为 `seed`，当然不初始化也是可以的。

同一程序使用相同的 `seed` 两次运行，在同一机器、同一编译器下，随机出的结果将会是相同的。

有一个选择是使用当前系统时间来作为随机种子：`srand(time(nullptr))`。

> **Warning**
>
> 在 `Windows` 系统下 `rand()` 返回值的取值范围为 $[0, 2^{15})$（即 `RAND_MAX` 等于 $2^{15}-1$），当需要生成的数不小于 2^{15} 时建议使用 `(rand() << 15 | rand())` 来生成更大的随机数。

关于 `rand()` 和 `rand()%n` 的随机性：

- C/C++ 标准并未关于 `rand()` 所生成随机数的任何方面的质量做任何规定。
- GCC 编译器对 `rand()` 所采用的实现方式，保证了分布的均匀性等基本性质，但具有 低位周期长度短 等明显缺陷。（例如在笔者的机器上，`rand()%2` 所生成的序列的周期长约 $2 \cdot 10^6$）
- 即使假设 `rand()` 是均匀随机的，`rand()%n` 也不能保证均匀性，因为 `[0,n)` 中的每个数在 `0%n,1%n,...,RAND_MAX%n` 中的出现次数可能不相同。

预定义随机数生成器

定义了数个特别的流行算法。如没有特别说明，均定义于头文件 `<random>`。

> **Warning**
>
> 预定义随机数生成器仅在于 C++11 标准[2]中开始使用。

mt19937

是一个随机数生成器类，效用同 `rand()`，随机数的范围同 `unsigned int` 类型的取值范围。

其优点是随机数质量高（一个表现为，出现循环的周期更长；其他方面也都至少不逊于 `rand()`），且速度比 `rand()` 快很多。使用时需要 `#include<random>`。

`mt19937` 基于 32 位梅森缠绕器，由松本与西村设计于 1998 年[3]，使用时用其定义一个随机数生成器即可：`std::mt19937 myrand(seed)`，`seed` 可不填，不填 `seed` 则会使用默认随机种子。

`mt19937` 重载了 `operator ()`，需要生成随机数时调用 `myrand()` 即可返回一个随机数。

另一个类似的生成器是 `mt19937_64`，基于 64 位梅森缠绕器，由松本与西村设计于 2000 年，使用方式同 `mt19937`，但随机数范围扩大到了 `unsigned long long` 类型的取值范围。

代码示例

```cpp
#include <ctime>
#include <iostream>
#include <random>

using namespace std;

int main() {
  mt19937 myrand(time(nullptr));
  cout << myrand() << endl;
  return 0;
}
```

`minstd_rand0`

线性同余算法由 Lewis、Goodman 及 Miller 发现于 1969，由 Park 与 Miller 于 1988 采纳为「最小标准」。

计算公式如下，其中 A, C, M 为预定义常数。

$$s_i \equiv s_{i-1} \times A + C \mod M$$

`minstd_rand()` 是较新的「最小标准」，为 Park、Miller 和 Stockmeyer 于 1993 推荐。

对于 `minstd_rand0()`，s 的类型取 32 位无符号整数，A 取 16807，C 取 0，M 取 2147483647。

对于 `minstd_rand()`，s 的类型取 32 位无符号整数，A 取 48271，C 取 0，M 取 2147483647。

`random_shuffle`

用于随机打乱指定序列。使用时需要 `#include<algorithm>`。

使用时传入指定区间的首尾指针或迭代器（左闭右开）即可：`std::random_shuffle(first, last)` 或 `std::random_shuffle(first, last, myrand)`

内部使用的随机数生成器默认为 `rand()`。当然也可以传入自定义的随机数生成器。

关于 `random_shuffle` 的随机性：

- C++ 标准中要求 `random_shuffle` 在所有可能的排列中 **等概率** 随机选取，但 GCC[4]编译器 **并未** 严格执行。
- GCC 中 `random_shuffle` 随机性上的缺陷的原因之一，是因为它使用了 `rand()%n` 这样的写法。如先前所述，这样生成的不是均匀随机的整数。
- 原因之二，是因为 `rand()` 的值域有限。如果所传入的区间长度超过 `RAND_MAX`，将存在某些排列 **不可能** 被产生[1]。

> **Warning**
>
> `random_shuffle` 已于 C++14 标准中被弃用，于 C++17 标准中被移除。

`shuffle`

效用同 `random_shuffle`。使用时需要 `#include<algorithm>`。

区别在于必须使用自定义的随机数生成器：`std::shuffle(first, last, myrand)`。

GCC[4]实现的 `shuffle` 符合 C++ 标准的要求，即在所有可能的排列中等概率随机选取。

下面是用 `rand()` 及 `random_shuffle()` 编写的一个数据生成器。生成数据为 「ZJOI2012」灾难 的随机小数据。

```cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>

int a[100];

int main() {
  srand(time(nullptr));
  int n = rand() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  std::cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    std::random_shuffle(a + 1, a + i);
    int cnt = rand() % i;
    for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
    std::cout << 0 << '\n';
  }
}
```

下面是用 `mt19937` 及 `shuffle()` 编写的同一个数据生成器。

```cpp
#include <algorithm>
#include <ctime>
#include <iostream>
#include <random>

int a[100];

int main() {
  std::mt19937 rng(time(nullptr));
  int n = rng() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  std::cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    std::shuffle(a + 1, a + i, rng);
```

```
    int cnt = rng() % i;
    for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
    std::cout << 0 << '\n';
  }
}
```

下面是随机排列前十个正整数的一个实现。

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  std::random_device rd;
  std::mt19937 g(rd());

  std::shuffle(v.begin(), v.end(), g);

  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, "
"));
  std::cout << "\n";
}
```

非确定随机数的均匀分布整数随机数生成器

> `random_device` 是一个基于硬件的均匀分布随机数生成器，**在熵池耗尽** 前可以高速生成随机数。该类在 C++11 定义，需要 `random` 头文件。由于熵池耗尽后性能急剧下降，所以建议用此方法生成 `mt19937` 等伪随机数的种子，而不是直接生成。

`random_device` 是非确定的均匀随机位生成器，尽管若不支持非确定随机数生成，则允许实现用伪随机数引擎实现。目前笔者尚未接到报告称 NOIP 评测机不支持基于硬件的均匀分布随机数生成。但出于保守考虑，建议使用该算法生成随机数种子。

参考代码如下。

```
#include <iostream>
#include <map>
#include <random>
#include <string>

int main() {
  std::random_device rd;
  std::map<int, int> hist;
  std::uniform_int_distribution<int> dist(0, 9);
  for (int n = 0; n < 20000; ++n) {
    ++hist[dist(rd)];  // 注意：仅用于演示：一旦熵池耗尽，
                       // 许多 random_device 实现的性能就急剧下滑
                       // 对于实践使用， random_device 通常仅用于
                       // 播种类似 mt19937 的伪随机数生成器
```

```
    }
    for (auto p : hist) {
      std::cout << p.first << " : " << std::string(p.second / 100, '*') <<
'\n';
    }
}
```

可能的输出如下。

```
0 : ********************
1 : ******************
2 : ********************
3 : ********************
4 : ********************
5 : ******************
6 : ********************
7 : ********************
8 : ******************
9 : ********************
```

随机数分布

这里介绍的是要求生成的随机数按照一定的概率出现，如等概率，伯努利分布，二项分布，几何分布，标准正态（高斯）分布。

具体类名请参见 伪随机数生成——随机数分布 的列表。

实现

下面的程序模拟了一个六面体骰子。

```
#include <iostream>
#include <random>

int main() {
  std::random_device rd;   // 将用于为随机数引擎获得种子
  std::mt19937 gen(rd());  // 以播种标准 mersenne_twister_engine
  std::uniform_int_distribution<> dis(1, 6);

  for (int n = 0; n < 10; ++n)
    // 用 dis 变换 gen 所生成的随机 unsigned int 到 [1, 6] 中的 int
    std::cout << dis(gen) << ' ';
  std::cout << '\n';
}
```

其他实现方法

有的时候我们需要实现自己的随机数生成器。下面是一些常用的随机数生成方法。

线性同余随机数生成器

利用下式来生成随机数序列 $\{R_i\}$：

$$R_{i+1} = (A \times R_i + B) \bmod P$$

其中 A, B, P 均为常数。

该方法实现难度低，但生成的随机序列周期长度较短（周期最大为 P，但大多数情况下都会比短）。

参考实现

```cpp
#include <iostream>
using namespace std;

struct myrand {
  int A, B, P, x;

  myrand(int A, int B, int P) {
    this->A = A;
    this->B = B;
    this->P = P;
  }

  // 生成随机序列的下一个随机数
  int next() { return x = (A * x + B) % P; }
};

myrand rnd(3, 5, 97);  // 初始化一个随机数生成器

int main() {
  int x = rnd.next();
  cout << x << endl;
  return 0;
}
```

时滞斐波那契随机数生成器

利用下式来生成随机数序列 $\{R_i\}$（其中 $0 < j < k$）：

$$R_i \equiv R_{i-j} \star R_{i-k} \bmod P$$

这里的 P 通常取 2 的幂（常用 2^{32} 或 2^{64}），$\star$ 表示二元运算符，可以使用加法，减法，乘法，异或。

该方法较传统的线性同余随机数生成器而言，拥有更长的周期，但随机性受初始条件影响较大。

参考实现

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct myrand {
  vector<unsigned> vec;
  int l, j, k, cur;

  myrand(int l, int j, int k) {
    this->l = l;
    this->j = j;
    this->k = k;
    cur = 0;
    for (int i = 0; i < l; i++) {
      vec.push_back(rand());  // 先用其他方法生成随机序列中的前几个元素
    }
  }

  unsigned next() {
    vec[cur] = vec[(cur - j + l) % l] * vec[(cur - k + l) % l];
    // 这里用 unsigned 类型是为了实现自动对 2^32 取模
    return vec[cur++];
  }
};

myrand rnd(11, 4, 7);

int main() {
  unsigned x = rnd.next();
  cout << x << endl;
  return 0;
}
```

参考资料与注释

1. Don't use rand(): a guide to random number generators in C++ ↩
2. 伪随机数生成 - cppreference.com ↩
3. Mersenne Twister algorithm ↩
4. 版本号为 GCC 9.2.0 ↩↩