C++11随机数的正确打开方式

在C++11之前，现有的随机数函数都存在一个问题：在利用循环多次获取随机数时，如果程序运行过快或者使用了多线程等方法，`srand((unsigned)time(null))`这样的设置当前系统时间为种子的方法每次返回的随机数都是一样的。而C++11中提供了真随机数做种子的方法来解决这一问题。

By the way,2019年了，我见过的编译器都不需要特殊指定使用的是C++11的新特征了

random_device

> 标准库提供了一个**非确定性随机数**生成设备.在Linux的实现中,是读取/dev/urandom设备;Windows的实现是用rand_s,使用的是操作系统来生成加密安全的**伪随机数**

注意，urandom实际上也是一种伪随机数，具体见下一节

> random_device提供()操作符,用来返回一个min()到max()之间的一个数字.

注意这里的min()和max()都是只能看不能改的。

random_device一般只用来作为其他伪随机数算法的种子，原因有三：

1. random_device的最大值和最小值不能修改，当然可以通过取模的方式获得想要的范围的数，但是毕竟不太优雅

2. 当熵池用尽后，许多random_device的具体实现的性能会急速下降（原话是："the performance of many implementations of random_device degrades sharply once the entropy pool is exhausted. For practical use random_device is generally only used to seed a PRNG such as mt19937"（来源：https://stackoverflow.com/questions/39288595/why-not-just-use-random-device）。但是这一点也有争议，在linux下random_device的实现其实是`std::fopen("/dev/urandom")`，有人说urandom在熵池耗尽之后输出的随机数是低质量的，但也有人说不是（https://blog.csdn.net/F8qG7f9YD02Pe/article/details/89880266），我没有对它做过多研究，但是下一点是毋庸置疑的

3. 多次调用random_device要花费比其他伪随机数算法更多的时间。在Linux中，正如上文所说，每次调用random_device都需要读urandom这个文件再关闭，而在WIndom中我们需要调用操作系统的API，再销毁实例化对象，这个时间花费显然比设置好种子就能一直产生的其他伪随机数算法要慢得多。

所以，我们一般将random_device只用作种子。

有关于熵池的内容见下一节。划重点：“产生真随机数依赖于熵池中的噪声资源。如果熵池资源耗尽，就需要等到收集足够多的环境噪声时，才能继续产生新的随机数。”

[转载]Linux 内核熵池与 /dev/urandom

原文地址：http://www.codebelief.com/article/2017/10/linux-entropy-pool-and-dev-urandom/

从计算机随机数谈起

我们知道，计算机是一个可预测的系统，因此不可能通过算法来产生真正的随机数。在计算机中，所谓的随机数通常都是伪随机数，就是通过随机算法计算出来的，可以被近似看作随机数的数值。常见的随机数算法有线性同余法（Linear Congruential Generator）、梅森旋转法（Mersenne twister）等，前者是大部分编译器采用的算法，随机性相对差一些；而后者是更为优秀的随机算法，随机性好，被 Python、Ruby 等语言用作默认的随机算法。

但是，随机算法的缺陷也是很明显的。一方面，随机性越好的算法计算复杂度越大；另一方面，即使随机性再好，也无法与真正的随机数相媲美。因此，产生真正的随机数是最理想的方法。

Linux 内核熵池

Linux 内核采用熵来描述数据的随机性。在物理学中，熵（entropy）是一个描述系统混乱程度的物理量，熵越大说明系统越无序、越混乱，不确定性越大。

虽然计算机本身可预测，但计算机的运行环境中充满了各种不可预知的噪声，例如来自设备驱动的噪声、随机的鼠标点击间隔、硬件设备发生中断的时间等等。

Linux 系统维护了一个专门用于收集上述噪声的熵池（entropy pool），这些噪声将被用于产生真正的随机数。

需要注意的是，**产生真随机数依赖于熵池中的噪声资源。如果熵池资源耗尽，就需要等到收集足够多的环境噪声时，才能继续产生新的随机数。**

/dev/urandom

Linux 提供了内核随机数生成器的接口，即字符设备/dev/random，**该字符设备用于生成高质量的随机数，它会确保熵池资源足够时才生成随机数。如上面所说，当熵池为空时，对/dev/random 的读取操作将会阻塞，直到收集足够的噪声为止。**

使用/dev/random 来生成随机数，很可能导致应用被阻塞。幸运的是，Linux 中还有另一个随机数生成器/dev/urandom，**该字符设备是/dev/random 的非阻塞版本，准确说它是一个伪随机数生成器，它的随机数种子来自于熵池，不过即使熵池为空，/dev/urandom 仍然能产生随机数。**

Linux 的 man(4)手册中这么写道：

/dev/random 是一个遗留下来的接口，在所有使用场景中，

/dev/random 接口遗留下来的原因主要是早期/dev/urandom 所采用的密码算法未被大家信任，但现在，/dev/urandom 已经被广泛的采用了。

解决由/dev/random 引起的阻塞

目前，仍有一些应用使用/dev/random 来生成随机数，这些应用在运行时，可能由于熵池耗尽而阻塞。例如 tomcat7.0 以上的版本，依赖于该生成器来生成随机数，可能启动时便没有反应，实际上是等待熵池收集噪声。此外，strongSwan 生成 CA 时使用的 `ipsec pki` 命令也可能因此而阻塞。

解决该问题可以通过安装 havged 程序来解决。haveged 是一个简单易用的不可预测随机数生成器，基于 HAVEGE 算法。haveged 可以解决在某些情况下，系统熵过低的问题。

参考 man 手册

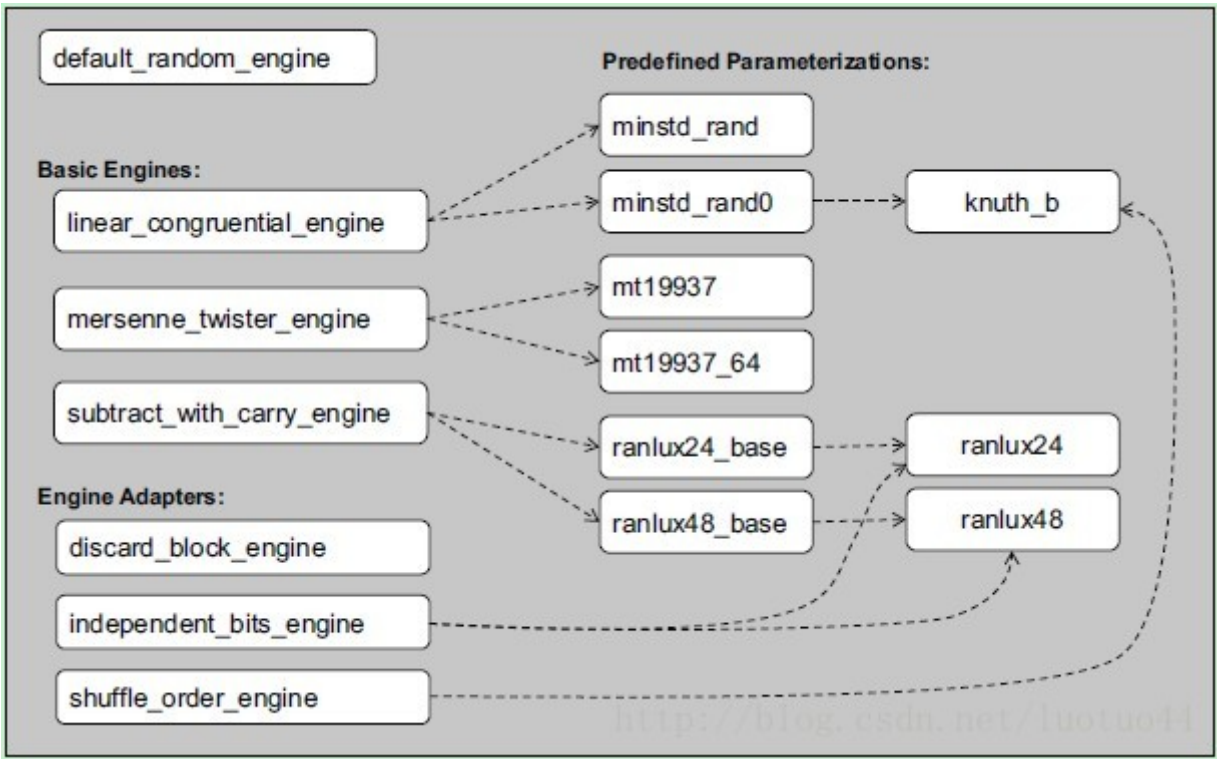
random(4) - Linux manual page

常用的随机数算法

上面我们说了，random_device一般只用来做种子，C++11中常用来做为伪随机数算法的有以下几种：

- linear_congruential_engine线性同余法,这种速度最快、最常用
- mersenne_twister_engine梅森旋转法，这种生成的随机数质量比较高
- substract_with_carry_engine滞后Fibonacci

但是这些类可以直接使用吗？这是不行的，这三个都是模板类，只是定义了接口，需要我们自己将它实例化。但是显然每个人实例化一次是不现实的，所以C++11中预先实现了一些给我们，我们只需要直接创建这些类的对象就好了。



> 还有一个default_random_engine的类。它是一个实例化的类。之所以不归入那三种算法，是因为它的实现是由编译器厂家决定的，有的可能用linear_congruential_engine实现，有的可能用mersenne_twister_engine实现。这种现象在C/C++中见多了。不过，对于其他的类，C++11是有明确规定用哪种算法和参数实现的。
>
> 对于default_random_engine来说，其产生的随机数范围是在[min()， max()]之间，其中min()和max()为它的两个成员函数，是闭区间。
>
> 来源：https://blog.csdn.net/luotuo44/article/details/33690179

在C++11里面，把这些随机数生成器叫做引擎(engines)

注意，random_device也是一种随机数引擎

虽然和每次都直接使用random_device相比，使用这些算法大大减少了多次生成随机数时的平均时间和空间花费，但是这些算法也存在问题，就是他们产生的范围还是太大了，如果我们需要特定范围下的随机数，依然需要取模。为了解决这一问题，我们要引出随机分布模板类。

随机分布模板类

常见的随机分布模板类

均匀分布：

uniform_int_distribution 整数均匀分布

uniform_real_distribution 浮点数均匀分布

注意，uniform_int_distribution的随机数的范围不是半开范围[ )，而是[ ]，对于uniform_real_distribution却是半开范围[ )。

伯努利类型分布：（仅有yes/no两种结果，概率一个p，一个1-p）

bernoulli_distribution 伯努利分布

binomial_distribution 二项分布

geometry_distribution 几何分布

negative_biomial_distribution 负二项分布

Rate-based distributions:

poisson_distribution 泊松分布

exponential_distribution指数分布

gamma_distribution 伽马分布

weibull_distribution 威布尔分布

extreme_value_distribution 极值分布

正态分布相关：

normal_distribution 正态分布

chi_squared_distribution卡方分布

cauchy_distribution 柯西分布

fisher_f_distribution 费歇尔F分布

student_t_distribution t分布

分段分布相关：

discrete_distribution离散分布

piecewise_constant_distribution分段常数分布

piecewise_linear_distribution分段线性分布

这些模板类都是定义好了的、可以直接使用的。

这些概率分布函数都是有参数的，在类的构造函数中把参数传进去即可。我们最常用的还是均匀分布，这里以 uniform_int_distribution为例介绍以下如何使用这些算法：

```
#include <random>
#include<iostream>
using namespace std;//要是不使用std名字空间，下面的就都需要加std::
void formData(){
        random_device sd;//生成random_device对象sd做种子
    minstd_rand linearRan(sd());//使用种子初始化linear_congruential
```

```
    uniform_int_distribution<int>dis1(0,1);//生成01序列
    for(int i=0;i<100;i++){
        cout<<dis1(linearRan)<<endl;//使用linear engine做种子，注意
    }
}
int main(){
    formData();
}
```

总结

还是有点绕的。总之，要得到不止一个一个我们最常需要的、符合一定分布规律的且随机质量较高的随机数，我们要做的是：

1. 定义random_device对象
2. 选择随机引擎（默认、线性、梅森、斐波那契）的实现类，将random_device的随机结果传入作为种子
3. 选择要的分布，创建分布对象，将引擎传入作为种子，让分布对象输出随机数。

分类: C++learning