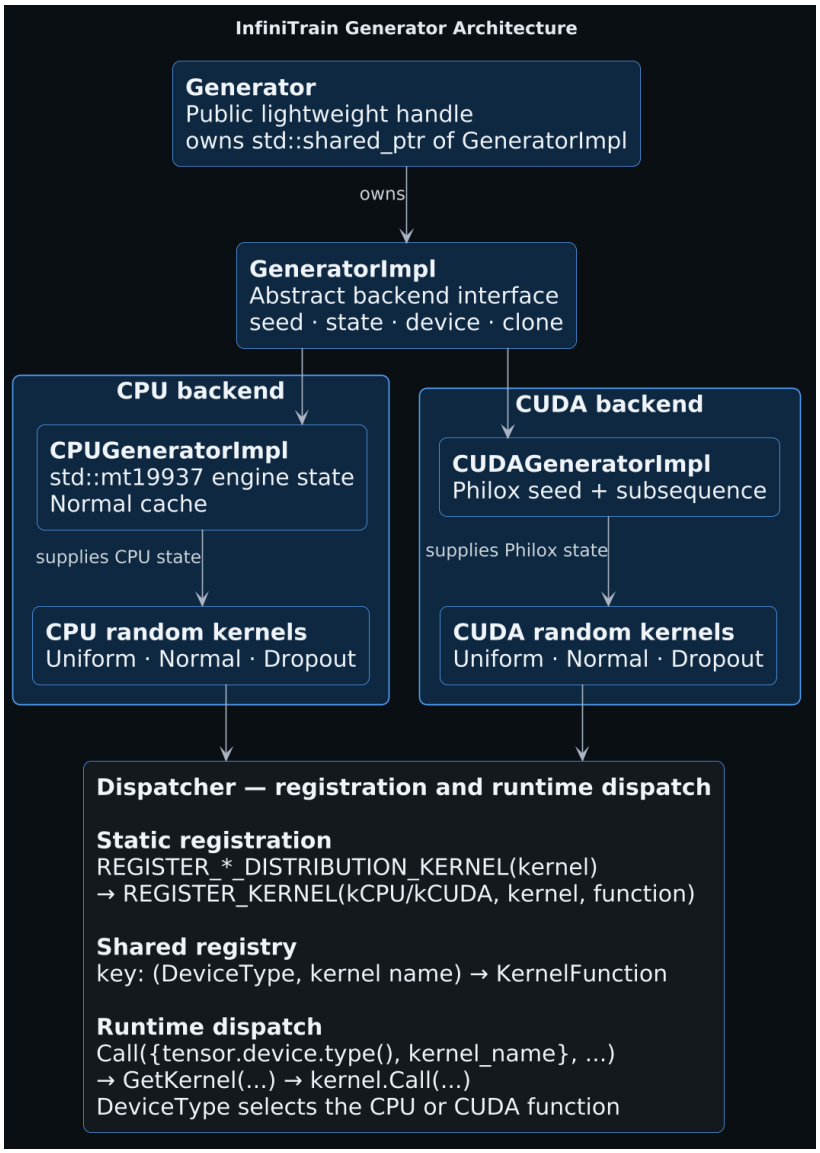
InfiniTrain Generator 抽象实现报告

一、设计与实现

下文按照赛题《【2026 春季人工智能大赛】Generator 抽象选题》的任务拆解，依次说明 Generator 抽象与状态管理、CPU/CUDA 后端实现、默认 Generator 与统一随机种子，以及随机算子的接入方式。每一部分同时说明与 PyTorch 的语义对应和当前实现边界。

下图概览 Generator 句柄如何经由 CPU/CUDA 后端向随机 kernel 提供状态，并由 Dispatcher 按设备完成分发。



1. Generator 抽象与状态管理

1.1 接口

对外暴露的接口集中在 `infini_train/include/generator.h`。底层抽象为 `GeneratorImpl` 基类，其中 seed/state 是纯虚接口，`device()` 和 `clone()` 由基类提供通用实现。

名称	声明与实现	PyTorch 对应语义	用途/与 PyTorch 对齐度
`Generator`（类）	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`at::Generator`，底层为 `c10::GeneratorImpl`	用户侧值类型句柄，浅拷贝共享底层 `GeneratorImpl`；`clone()` 用于深拷贝。句柄与 clone 语义基本对齐，智能指针类型不同。
`Generator::set_current_seed()` `Generator::current_seed()`	`GeneratorImpl` 纯虚接口，`infini_train/include/generator.h` CPU：`infini_train/src/core/runtime/cpu/cpu_generator_impl.cc` CUDA：`infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::Generator::set_current_seed()` `at::Generator::current_seed()`	设置和获取当前种子，核心语义对齐。
`Generator::seed()`	`GeneratorImpl` 纯虚接口，`infini_train/include/generator.h` CPU：`infini_train/src/core/runtime/cpu/cpu_generator_impl.cc` CUDA：`infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::Generator::seed()`	生成一个非确定的新种子，并用它重置 Generator。核心语义对齐。
`Generator::get_state()` `Generator::set_state()`	`GeneratorImpl` 纯虚接口，`infini_train/include/generator.h` CPU：`infini_train/src/core/runtime/cpu/cpu_generator_impl.cc` CUDA：`infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::Generator::get_state()` `at::Generator::set_state()`	保存和恢复随机序列的推进位置。接口语义对齐，state 的二进制布局不同。
`Generator::device()`	`infini_train/include/generator.h` 头文件内联实现	`at::Generator::device()`	查询 Generator 所属设备，语义对齐。
`Generator::clone()`	`infini_train/include/generator.h` 基类 `GeneratorImpl::clone()` 内联实现，后端通过 `clone_impl()` 深拷贝	`at::Generator::clone()`	创建具有独立随机状态的 Generator，语义对齐。
`Generator::mutex()`	`infini_train/include/generator.h` 头文件内联实现	`at::Generator::mutex()`	取得底层 Generator 的互斥锁。两边均公开锁，但 PyTorch 更明确要求调用方为非只读操作持锁。
`infini_train::CreateGenerator()`	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`at::make_generator<Impl>()`，以及后端创建函数	按设备创建 Generator。创建目的对齐，PyTorch 的创建入口按后端拆分。
`infini_train::GetDefaultGenerator()`（见下文）	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`at::globalContext().defaultGenerator(Device)`	获取设备默认 Generator，按设备获取和延迟创建语义对齐。
`infini_train::manual_seed()`（见下文）	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`torch.manual_seed()`	重置所有已启用设备的默认 Generator 种子。影响范围与全局设种子入口对齐。

1.2 设计说明

`Generator` 是上层可持有的轻量句柄，只保存一个 `std::shared_ptr<GeneratorImpl>` 。因此复制 `Generator` 不会复制随机状态，两个句柄会继续消费同一条随机流；调用 `clone()` 才会得到状态独立、初始内容相同的 Generator。这个语义把“共享同一随机流”和“复制出一条新随机流”区分为两个显式操作。

`GeneratorImpl` 负责定义不同设备 Generator 共有的状态接口，CPU 和 CUDA 分别提供派生实现。随机算子只接收 `Generator` ，不直接依赖某个后端的实现类型。后端类和相关头文件放在 `src/core/runtime/` 下，普通调用路径无需接触 `std::mt19937` 、Philox 或 curand 等随机引擎细节。

创建 Generator 时， `CreateGenerator` 根据 `Device` 选择相应后端实现，调用方不需要包含后端 Impl 头文件。**后续接入新的设备类型时，可以增加对应的 `GeneratorImpl` 派生类和创建分支，而无需改变上层随机算子的 Generator 参数形式。**

实现中将 seed 与 state 分开处理。 `set_current_seed()` 用于重新初始化随机流， `get_state()` 与 `set_state()` 用于保存和恢复已经推进到的位置；随机算子每次使用 Generator 后都会推进其状态。state 的具体编码由后端自行维护，因此公共接口只传递状态对象，不依赖某一种随机引擎的内部表示。

互斥锁归属于 `GeneratorImpl` 。Uniform、Normal 和 Dropout 的 kernel 在消费同一 Generator 时持锁，避免并发调用同时推进同一份状态；如果调用方需要将多个 Generator 操作作为一个原子过程执行，则需要通过 `Generator::mutex()` 自行界定加锁范围。

1.3 对齐与差异

以下比较基于 2026-08-01 拉取的 PyTorch main（ `85728e11a5a98c344c4596880855e35046db81b0` ）中的 `ATen/core/Generator.h` 、 `c10/core/GeneratorImpl.h` 及 CPU/CUDA Generator 实现。

1. 基本一致的行为：两边都采用“用户侧 Generator 句柄 + 设备相关 GeneratorImpl”的多态结构，普通句柄复制共享同一 Impl， `clone()` 才创建独立状态，Impl 的 Copy/Move 均被删除。 `set_current_seed()` 、 `current_seed()` 、 `seed()` 、 `get_state()` 、 `set_state()` 和 `device()` 的核心用途相同；默认 Generator 都按设备维护并延迟创建。公开的 state 载体也都是 CPU 字节 Tensor，而不是裸字节数组。
2. 未对齐的行为：InfiniTrain 用 `std::shared_ptr` ，PyTorch 用 `c10::intrusive_ptr` ，因此没有 `unsafeReleaseGeneratorImpl()` 、 `getIntrusivePtr()` 等所有权和运行时集成接口。两边的 state 二进制布局不同：本实现使用后端 magic、CPU engine 序列化或 CUDA subsequence，PyTorch CPU 使用固定大小 POD，CUDA 使用 seed 与 Philox offset；本实现没有显式 version 字段，公共检查也未验证 state 连续性。线程语义也不完全一致，PyTorch 要求调用方为非只读 Generator 操作持有 mutex，而本实现仅对已接入的 Uniform、Normal、Dropout kernel 自动加锁，直接变更状态或组合多个操作仍由调用方协调。此外，PyTorch 提供

`set_offset()`、`get_offset()`、`philox_state()` 和 CUDA Graph 的 graph-safe state 接口；本实现通过内部 `philox_subsequence()` 满足普通 CUDA kernel 的区间预留需求，这些超出赛题范围的扩展能力未纳入当前实现。

2. CPU 与 CUDA 后端 Generator

2.1 接口

CPU 和 CUDA 后端都继承 `GeneratorImpl`，对外统一实现 seed/state/device/clone 接口。为了服务上层 kernel，两个后端还暴露了一些后端专用方法，这些方法只在 `src/` 内部使用。

名称	声明与实现	PyTorch 对应语义	用途/与 PyTorch 对齐度
`CPUGeneratorImpl`（类）	`infini_train/src/core/runtime/cpu/cpu_generator_impl.h` `infini_train/src/core/runtime/cpu/cpu_generator_impl.cc`	`at::CPUGeneratorImpl`	CPU 后端实现，基于 `std::mt19937`。后端角色与 seed/state 语义对齐，具体 engine 类型不同。
`CUDAGeneratorImpl`（类）	`infini_train/src/core/runtime/cuda/cuda_generator_impl.h` `infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::cuda::CUDAGeneratorImpl`	CUDA 后端实现，基于 Philox。普通 kernel 的随机区间管理语义基本对齐，PyTorch 额外支持 CUDA Graph。
`CPUGeneratorImpl::get_state()` `CPUGeneratorImpl::set_state()`	`infini_train/src/core/runtime/cpu/cpu_generator_impl.h` `infini_train/src/core/runtime/cpu/cpu_generator_impl.cc`	`at::CPUGeneratorImpl::get_state()` `set_state()`	序列化或恢复 engine、seed 与 Normal 缓存。保存与恢复语义对齐，PyTorch 使用固定大小 state，本实现为可变长度序列化。
`CUDAGeneratorImpl::get_state()` `CUDAGeneratorImpl::set_state()`	`infini_train/src/core/runtime/cuda/cuda_generator_impl.h` `infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::cuda::CUDAGeneratorImpl::get_state()` `set_state()`	序列化或恢复 magic、seed 与 Philox subsequence。保存与恢复语义对齐，PyTorch 以 Philox offset 表示推进位置。
`CPUGeneratorImpl::random()` `CPUGeneratorImpl::random64()`	`infini_train/src/core/runtime/cpu/cpu_generator_impl.h` `infini_train/src/core/runtime/cpu/cpu_generator_impl.cc`	`at::CPUGeneratorImpl::random()` `random64()`	返回 32/64 位随机数，供 CPU distribution 使用。接口目的对齐。

`CPUGeneratorImpl::next_float_normal_sample()`
`set_next_float_normal_sample()`
`next_double_normal_sample()`
`set_next_double_normal_sample()`
	`infini_train/src/core/runtime/cpu/cpu_generator_impl.h`
	`infini_train/src/core/runtime/cpu/cpu_generator_impl.cc`
		`at::CPUGeneratorImpl` 的同名 float/double 缓存接口	缓存 Box-Muller 算法的第二个样本。float 与 double 两类缓存语义对齐。

`CUDAGeneratorImpl::philox_subsequence()`	`infini_train/src/core/runtime/cuda/cuda_generator_impl.h` `infini_train/src/core/runtime/cuda/cuda_generator_impl.cc`	`at::cuda::CUDAGeneratorImpl::philox_cuda_state(increment)`	为一次 kernel 调用预留 Philox subsequence，并推进内部计数器。用途相近，但 PyTorch 使用 offset 并兼顾 CUDA Graph 捕获。

上层 kernel 不直接引用 `CPUGeneratorImpl` 或 `CUDAGeneratorImpl`，而是通过 `check_generator<T>()` 和 `get_generator_or_default<T>()` 在内部转换。

2.2 设计说明

CPU 后端使用 `std::mt19937`。构造时将 64 位 seed 的高、低 32 位同时交给 `std::seed_seq` 初始化 engine，避免直接调用 `std::mt19937(seed)` 时高 32 位被截断。CPU 的 Uniform 和 Normal 通过 `distributions_helper.h` 消费 engine；Normal 采用 Box-Muller，并把尚未消费的第二个样本缓存到 Generator 中。因此，CPU state 除 engine 以外，还需要保存 seed 和 float/double 两种缓存。

CUDA 后端不保存每个元素的 `curandState`，而是保存 seed 和下一段可分配的 Philox subsequence。一次随机 kernel 启动前，在锁保护下调用 `philox_subsequence(n)` 预留区间；kernel 随后以 `curand_init(seed, subsequence + index, 0, &state)` 为各元素构造独立状态。这样后续调用从新的 subsequence 开始，不会与已经启动的 kernel 重用随机区间。

两个后端都将 state 作为 CPU `UINT8` Tensor 返回。`set_state()` 先检查该公共载体，再由后端检查长度和 magic，因此 CPU/CUDA state 会相互拒绝，长度、dtype、设备或 magic 不符合要求的 state 也会失败。CPU 还会检查 engine 序列化内容和缓存标志；CUDA 的固定字段格式不对 seed 与 subsequence 的数值额外作完整性校验。当前 state 使用 backend magic 区分来源，但没有显式 version 字段；CUDA state 也不编码 device index，因此同一 CUDA backend 的 state 可以恢复到另一张 CUDA 卡。state 格式属于后端私有实现，不保证跨版本兼容。

2.3 对齐与差异

1. 基本一致的行为：CPU 使用 Mersenne Twister、CUDA 使用 Philox；CUDA 默认 Generator 按设备维度独立维护。两边都不要求 CPU 与 CUDA 在相同 seed 下生成逐元素一致的结果。

2. 未对齐的行为：PyTorch CPU state 是固定大小的 POD 结构，本实现将 engine 序列化为字符串，因此大小可变。PyTorch CUDA 的常规 state 使用 seed 与 Philox offset，本实现使用带 magic 的 seed 与 subsequence。PyTorch 还提供公开的 offset 控制和 CUDA Graph 相关接口；本实现通过内部 `philox_subsequence()` 满足普通 CUDA kernel 的区间预留需求，未覆盖这些超出赛题范围的扩展能力。

3. 默认 Generator 与全局随机种子

3.1 接口

名称	声明与实现	PyTorch 对应语义	用途/与 PyTorch 对齐度
`infini_train::GetDefaultGenerator(Device)`	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`at::globalContext().defaultGenerator(Device)`	返回指定设备的默认 Generator。按设备返回默认状态来源的语义对齐。
`infini_train::manual_seed()`	`infini_train/include/generator.h` `infini_train/src/generator.cc`	`torch.manual_seed()`	重置 CPU 默认 Generator 和所有 CUDA 设备的默认 Generator。全局默认状态的设种子语义对齐。

3.2 设计说明

默认 Generator 是框架持有的随机状态来源。CPU 维护一个进程级单例；CUDA 按设备索引维护默认 Generator，首次请求该设备时才创建。`GetDefaultGenerator(Device)` 根据设备类型与 CUDA 设备索引分派并返回对应句柄，因此重复获取同一设备会共享同一份随机状态，不同 CUDA 设备的默认状态彼此独立。

随机算子的 Generator 参数为可选项。调用方传入已定义的 `Generator` 时，算子直接使用该句柄，不会消耗默认 Generator；未传入参数或传入未定义的 `Generator{}` 时，算子根据目标 Tensor 所在设备取得默认 Generator。显式 Generator 仍会校验后端类型，CPU Generator 不能用于 CUDA kernel，反之亦然。

`manual_seed(uint64_t)` 是统一的显式设种子入口。它先重置 CPU 默认 Generator，再初始化并逐个重置所有可见 CUDA 设备的默认 Generator。重置过程在各自的 mutex 保护下进行。该函数只影响默认 Generator，不会改变调用方通过 `CreateGenerator()` 创建的独立 Generator；在相同种子、设备与随机算子调用顺序下，默认路径可稳定复现。

3.3 对齐与差异

1. 基本一致的行为：PyTorch 通过 `at::globalContext().defaultGenerator(Device)` 获取设备默认 Generator，CPU 与 CUDA 后端再分别维护对应实例。本实现的 `GetDefaultGenerator(Device)` 采

用相同的按设备获取语义。两边的 `manual_seed` 都会重置 CPU 默认 Generator 与当前可见 CUDA 设备的默认 Generator，显式传入的 Generator 也都会优先于默认路径使用。

2. 未对齐的行为：PyTorch 的 Context 可以通过 accelerator hooks 路由到更多设备后端，本实现当前只覆盖 CPU 与 CUDA。PyTorch 的默认 Generator 获取主要位于 Context 与后端内部接口中，本实现将 `GetDefaultGenerator(Device)` 作为直接的公共入口。全局设种子只覆盖框架默认状态，不会替显式创建的 Generator 改种子，这一边界与 PyTorch 一致；若后续接入新的设备后端，需要补充对应的默认 Generator 管理与 `manual_seed` 路径。

4. 随机算子接入

4.1 接口

名称	声明与实现	PyTorch 对应语义	用途/与 PyTorch 对齐度
`infini_train::nn::init::Uniform()`	`infini_train/include/nn/init.h` `infini_train/src/nn/init.cc`	`torch.nn.init.uniform_()`	对已有 Tensor 做均匀分布初始化。初始化语义对齐，均支持显式 Generator。
`infini_train::nn::init::Normal()`	`infini_train/include/nn/init.h` `infini_train/src/nn/init.cc`	`torch.nn.init.normal_()`	对已有 Tensor 做正态分布初始化。初始化语义对齐，均支持显式 Generator。
`infini_train::nn::init::KaimingUniform()`	`infini_train/include/nn/init.h` `infini_train/src/nn/init.cc`	`torch.nn.init.kaiming_uniform_()`	计算 Kaiming uniform 边界后调用 `Tensor::Uniform()`。计算 fan、gain 和边界的目标语义对齐。
`Tensor::Uniform()`	`infini_train/include/tensor.h` `infini_train/src/tensor.cc`	`Tensor.uniform_()`	Tensor 成员方法，转发到 `nn::init::Uniform()`。原地均匀填充语义对齐。
`infini_train::nn::function::Rand()`	`infini_train/include/nn/functional.h` `infini_train/src/nn/functional.cc`	`torch.rand()`	创建并返回均匀分布的新 tensor。创建语义和可选 Generator 参数对齐。
`infini_train::nn::function::Randn()`	`infini_train/include/nn/functional.h` `infini_train/src/nn/functional.cc`	`torch.randn()`	创建并返回正态分布的新 tensor。创建语义和可选 Generator 参数对齐。
`infini_train::nn::function::Dropout()`	`infini_train/include/nn/functional.h` `infini_train/src/nn/functional.cc`	`torch.nn.functional.dropout()`	Dropout 前向，training 时按概率丢弃元素。前向随机语义对齐，但 InfiniTrain 额外暴露可选 Generator 参数，PyTorch 公共接口使用默认 Generator。

上述接口均以 `std::optional<Generator>` 接收可选 Generator。未传入参数或传入未定义的 `Generator{}` 时，后端回退到目标 Tensor 设备的默认 Generator。

4.2 设计说明

初始化与训练期随机算子共用同一条 Generator 接入路径。`Rand` 与 `Randn` 创建目标 Tensor 后，分别转发到 `Uniform` 与 `Normal`；`KaimingUniform` 先根据 fan 和 gain 计算边界，再调用

`Tensor::Uniform` 。这些上层接口只传递可选 Generator，并通过 `Dispatcher::Instance().Call` 进入按设备注册的 CPU 或 CUDA kernel，不直接依赖 `std::mt19937` 、curand 或 Philox 的具体类型。

后端 kernel 通过 `get_generator_or_default()` 解析可选参数。CPU kernel 取得 Generator 后持锁，逐样本消费 `std::mt19937` engine；CUDA kernel 在持锁范围内读取 seed 并按 Tensor 元素数预留 Philox subsequence，随后在 kernel 中以 `curand_init(seed, subsequence + index, 0, &state)` 初始化每个元素的随机状态。一次实际抽样会推进所选 Generator 的状态，因而相同 seed、设备、Tensor 形状和调用顺序能够复现同一后端上的结果。并发调用虽受 mutex 保护，但锁竞争顺序本身不构成可复现的调用顺序。

`Dropout` 将可选 Generator 保存在 `autograd::Dropout` 中，并交给 `DropoutForward` kernel 生成输出与 `UINT8` mask。mask 被标记为不可微并保存到上下文，反向阶段由 `DropoutBackward` 依照同一 mask 缩放梯度，不再消耗随机状态。`training=false` 、 `p=0` 、 `p=1` 或空 Tensor 不进行实际抽样。CPU/CUDA 的 distribution 和 Dropout kernel 均覆盖 `FLOAT16` 、 `BFLOAT16` 、 `FLOAT32` 、 `FLOAT64` 四种浮点 dtype。

4.3 对齐与差异

1. 基本一致的行为： `Uniform` 、 `Normal` 、 `KaimingUniform` 、 `Rand` 与 `Randn` 都可选择显式 Generator 或设备默认 Generator，语义对应 PyTorch 初始化函数及 `torch.rand` 、 `torch.randn` 的 generator 参数。两边的 Dropout 都在前向阶段生成并保存 mask，反向阶段复用该 mask；CPU 与 CUDA 的随机算法不同，因此只承诺各后端在相同调用条件下可复现，不承诺跨设备逐元素一致。
2. 未对齐的行为：本实现将 `std::optional<Generator>` 直接暴露给 `Dropout` ，方便训练代码显式控制随机流；PyTorch 的 `torch.nn.functional.dropout` 公共接口没有 generator 参数，通常使用当前设备默认 Generator。本实现的 Dropout 没有 inplace 选项，随机初始化覆盖范围也只包含本题接入的 Uniform、Normal 和 KaimingUniform，未扩展到 PyTorch 的全部随机算子与初始化策略。后续新增随机算子时，可沿用可选 Generator 参数、后端校验和 kernel 内状态预留这条接入路径。

二、其他改动与后续工作

DDP 构造阶段参考 PyTorch 的初始化同步语义，由 rank 0 将模块参数和 buffer 原地广播到其余 rank，使各卡在训练开始前持有一致的模型状态。目前的实现不试图改变单进程多线程训练时共享 CPU 默认 Generator 的语义。多个 rank 对默认随机流的消费顺序受线程调度影响，因此即使重复相同操作，也不保证跨次运行得到完全相同的随机序列。后续可考虑两条路线：采用类似 `torchrun` 的一 GPU 一进程启动方式，使各 rank 拥有独立进程内 RNG；或参考 Megatron 的思路，为不同线程提供隔离的局部随机流。后者会扩展全局默认 Generator 的设计边界，因此未纳入本题实现范围。

三、验证与复现

3.1 验证环境

本节正式结果来自租用的 7 卡 CUDA 服务器，运行日志见后文图片。配置如下。

项目	配置
GPU	NVIDIA RTX 4090 D，7 张
CUDA Toolkit	12.8
CMake	3.31.4
NCCL	2.25.1
C++ 编译器	`/usr/bin/g++-13`
CUDA 架构	`CMAKE_CUDA_ARCHITECTURES=89`
构建选项	`USE_CUDA=ON` ， `USE_NCCL=ON` ， `BUILD_TEST=ON`

3.2 测试内容与结果

Generator 测试位于 `tests/generator/` 。 `test_generator_cpu` 与 `test_generator_cuda` 由同一套参数化 gtest 源码生成；本次分别通过 `--gtest_filter='CPU/*'` 和 `--gtest_filter='CUDA/*'` 选择对应参数实例。因此两行覆盖的用例互不重复，可以合并阅读。

可执行文件	本次结果	覆盖内容
`test_generator_cpu`	17 passed，2 skipped，2454 ms	CPU 参数实例。两条仅适用于 CUDA 的用例按条件跳过。
`test_generator_cuda`	17 passed，2 skipped，5719 ms	CUDA 参数实例。两条仅适用于 CPU 的用例按条件跳过；7 卡默认 Generator 独立性与 CUDA Generator 跨卡使用用例实际通过。
`test_generator_ddp`	1 passed，1603 ms	DDP 构造后将 rank 0 参数广播到其余 GPU 的测试。7 卡环境实际执行并通过。

两组 Generator 测试共 34 条通过、4 条按后端条件跳过，没有失败用例。CPU 筛选跳过 CUDA 专用项，CUDA 筛选跳过 CPU 专用项；由于服务器有 7 张 GPU，所有需要至少 2 张 GPU 的用例均实际运

行。

核心测试覆盖了本题的四类功能。`test_generator_core.cc` 验证 Generator 的句柄和 state 语义、默认 Generator 与显式 Generator 的选择、CPU/CUDA 后端不匹配时的拒绝，以及初始化入口的可复现性；`test_generator_random_ops.cc` 验证 `Rand`、`Randn`、`Dropout` 的固定随机脚本重放、默认状态不受显式 Generator 消耗、四种浮点 dtype、非连续 view 的写入范围、Dropout mask 与反向计算，以及非法参数的失败路径。DDP 用例是附带验证，关注多 GPU 环境中模块参数的初始同步，不改变 Generator 的公共语义。

```
[----------] Global test environment tear-down
[==========] 19 tests from 2 test suites ran. (2454 ms total)
[  PASSED  ] 17 tests.
[  SKIPPED ] 2 tests, listed below:
[  SKIPPED ] CPU/GeneratorCoreTest.CUDAGeneratorCanDriveAnotherCUDADevice/0
[  SKIPPED ] CPU/GeneratorCoreTest.DefaultCUDAGeneratorsAreIndependentAcrossAllDevices/0
```

```
[----------] Global test environment tear-down
[==========] 19 tests from 2 test suites ran. (5719 ms total)
[  PASSED  ] 17 tests.
[  SKIPPED ] 2 tests, listed below:
[  SKIPPED ] CUDA/GeneratorCoreTest.HighBitsOfCPUSeedAffectTheSequence/0
[  SKIPPED ] CUDA/GeneratorCoreTest.CrossBackendGeneratorsAreRejected/0
```

```
root@f03cddbcc3f7:/data/chaos/InfiniTrain-todo-generator-current/build# ./tests/generator/test_generator_ddp
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from DistributedDataParallelTest
[ RUN      ] DistributedDataParallelTest.SynchronizesParametersFromRankZeroAcrossAvailableGPUs
[       OK ] DistributedDataParallelTest.SynchronizesParametersFromRankZeroAcrossAvailableGPUs (1603 ms)
[----------] 1 test from DistributedDataParallelTest (1603 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1603 ms total)
[  PASSED  ] 1 test.
```

3.3 复现步骤

在仓库根目录配置并构建测试：

```
cmake -S . -B build \
  -DCMAKE_CXX_COMPILER=/usr/bin/g++-13 \
  -DCMAKE_CUDA_HOST_COMPILER=/usr/bin/g++-13 \
  -DUSE_CUDA=ON \
  -DUSE_NCCL=ON \
  -DBUILD_TEST=ON \
  -DCMAKE_CUDA_ARCHITECTURES=89 \
  -DCMAKE_POLICY_VERSION_MINIMUM=3.5
cmake --build build -j$(nproc)
```

进入构建目录后运行 Generator 测试。若运行环境未为 glog 或 CUDA 配置动态库搜索路径，可先设置 `LD_LIBRARY_PATH` 。

```
cd build
export LD_LIBRARY_PATH="$PWD/third_party/glog:/usr/local/cuda/lib64\
${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}"

./tests/generator/test_generator_cpu --gtest_filter='CPU/*'
./tests/generator/test_generator_cuda --gtest_filter='CUDA/*'
./tests/generator/test_generator_ddp
```

CPU 筛选会跳过 CUDA 专用用例，CUDA 筛选会跳过 CPU 专用用例。若机器少于两张 GPU，CUDA 的跨卡 Generator、各设备默认 Generator 独立性和 DDP 测试还会按条件跳过。本报告使用的 RTX 4090 D 架构为 89；在其他 GPU 上复现时，应按实际架构修改 `CMAKE_CUDA_ARCHITECTURES` 。