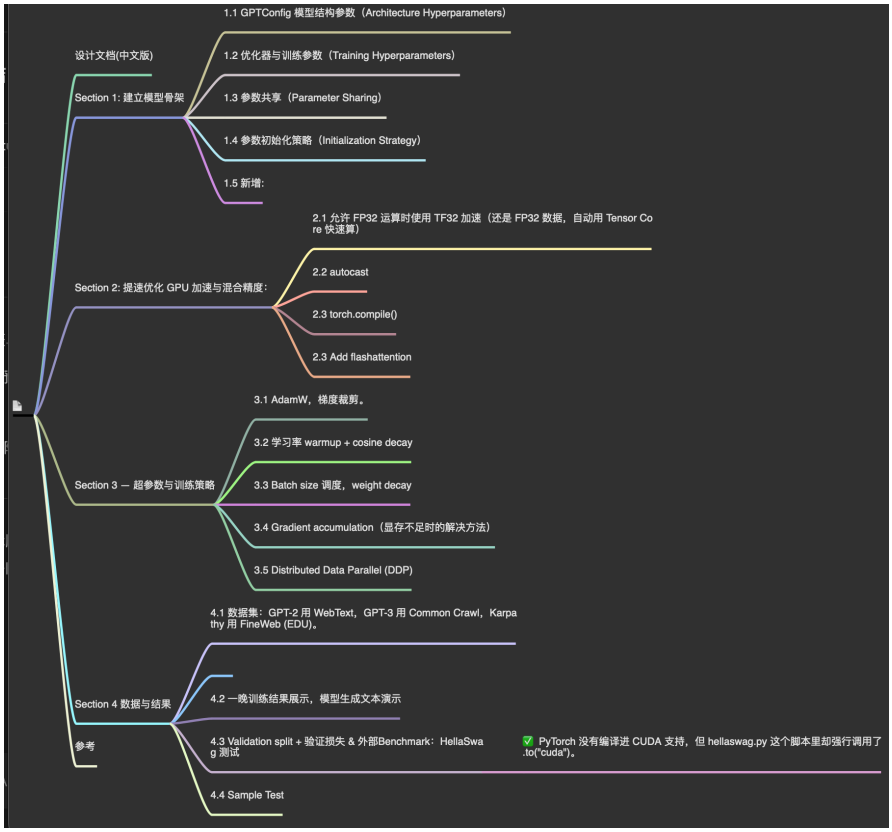
# 设计文档(中文版)




---

# Section 1: 建立模型骨架

- `nn.Module`：从零搭建 GPT-2 基础结构。
- 加载 HuggingFace GPT-2 参数（对比权重）。
- 实现 **forward pass**，输出 logits。
- Tokenizer & Sampling loop。
- Data batches `(B,T) → logits (B,T,C)`，计算 cross entropy loss。
- 写 mini optimization loop（先过拟合一个 batch 验证正确性）。

**重要参数:**

## 1.1 GPTConfig 模型结构参数（Architecture Hyperparameters）

| 参数名 | 含义 | 默认值 | 说明 |
| --- | --- | --- | --- |
| **block_size** | 最大上下文长度（序列长度） | 1024 | 模型一次能看到的 token 数；GPT-2 context window。 |
| **vocab_size** | 词表大小（token 种类数） | 50257 | BPE 编码的词表，OpenAI GPT-2 标准。 |
| **n_layer** | Transformer 层数 | 12 | 堆叠多少个 Block（每个包含 Attention + MLP + 残差）。 |
| **n_head** | Attention 头数 | 12 | 每个注意力层的并行子空间数量。 |
| **n_embd** | Embedding 维度 | 768 | 每个 token 的向量维度（也即隐藏层宽度）。 |

## 1.2 优化器与训练参数（Training Hyperparameters）

| 参数名 | 含义 | 默认值 / 设置 | 大规模分布式设置 | 说明 |
| --- | --- | --- | --- | --- |
| **B** | batch size | 4 | 64 | 微 batch size 64，靠 gradient accumulation 实现更大有效 batch |
| **T** | sequence length | 32 | 1024 | 训练上下文长度恢复 GPT-2 原始 1024 |
| **lr** | learning rate | 3e-4 | 6e-4（cosine schedule） | 与 GPT-2 论文一致（$6\times10^{-4}$） |
| **optimizer** | 优化器 | AdamW | AdamW (fused CUDA 版) | Adam + decoupled weight decay，适合 Transformer。 |
| **seed** | 随机种子 | 1337 | | 控制随机初始化和采样，使实验可复现。PyTorch ≥2.0 新增 fused 参数以加速 |

### 1.3 参数共享（Parameter Sharing）

| 共享部分 | 说明 |
| --- | --- |
| `self.transformer.wte.weight = self.lm_head.weight` | GPT-2 采用 **权重共享机制**，即 token embedding 与输出层权重相同。<br>这样能让输入与输出空间对齐，减少参数量，提高泛化能力。 |

### 1.4 参数初始化策略（Initialization Strategy）

| 部分 | 方法 | 细节 |
| --- | --- | --- |
| **Linear层 / Embedding层** | `torch.nn.init.normal_` | 均值 = 0.0，标准差 = 0.02（和 OpenAI GPT-2 一致） |
| **Residual scaling** | 特殊情况 → 乘以 `2 × (n_layers)⁻¹/²` => 改写为标志变量 `NANOGPT_SCALE_INIT` | 这是 **NANOGPT_SCALE_INIT** 对残差路径进行缩放，防止梯度爆炸。 |
| **Bias项** | 初始化为 0 | 确保线性层初始输出平稳。 |

### 1.5 新增:

| 新功能 | 说明 |
| --- | --- |
| `from_pretrained()` | 新增了直接加载 HuggingFace GPT-2 权重的类方法 |
| `configure_optimizers()` | 抽象出优化器配置逻辑（分 decay / no_decay 参数组） |
| DDP 初始化与 `master_process` 标志 | 支持多 GPU 并行训练 |
| FlashAttention | 调用 `F.scaled_dot_product_attention(..., is_causal=True)` 代替原始 mask |

---

# Section 2: 提速优化 GPU 加速与混合精度：

## 2.1 允许 FP32 运算时使用 TF32 加速（还是 FP32 数据，自动用 Tensor Core 快速算）

```
torch.set_float32_matmul_precision("high")
```

TF32 的 **执行机制**：

```
FP32 matrix（输入）
  ↓
Format to TF32（转换）
  ↓
Multiply（TF32×TF32）
  ↓
Accumulate（FP32 累加）
  ↓
FP32 matrix（输出）
```

## 2.2 autocast

**直接把部分 FP32 运算降成 BF16 精度**，进一步加速和省显存。

既能启用 TF32（优化 FP32 路径），又能自动混合 BF16（进一步节省显存、加速）。

```
with torch.autocast(device_type=device, dtype=torch.bfloat16):
    logits, loss = model(x, y)
```

> 在这一个 `with` 代码块中，PyTorch 自动把能安全用低精度算的操作（例如 matmul、conv、attention）
> 换成 **bfloat16 (BF16)** 精度来执行，
> 同时保留关键操作（如 loss 计算、归一化）用高精度 FP32，避免数值爆炸或溢出。

## 2.3 torch.compile()

> `torch.compile()` 是 PyTorch 2.0 引入的动态图编译器接口。
> 它会把你原本一行行执行的 Python 模型代码，
> **转换成底层优化后的、高性能的 GPU 执行图**，

```
model = torch.compile(model)
```

compile的额外问题:

1. **生成（inference）冲突**
   编译后的模型被"冻结"成一个图，输入 shape 必须一致。
   但推理阶段会循环追加 token（序列长度不断变化），这会让编译缓存失效或直接报：

   ```
   RuntimeError: Input shape changed after graph compilation
   ```

   所以推理部分不能用 compile。

2. **混合精度（autocast）冲突**
   `torch.compile` 与 `torch.autocast(device_type=device, dtype=torch.bfloat16)` 在某些 GPU 驱动下会有精度不一致问题。
   Karpathy 在 fine-tuning 时发现 loss 不稳定，因此选择关闭。
3. **HellaSwag 评估部分不兼容**
   评估阶段会反复进入/退出 no_grad + autocast 环境，也会导致 Dynamo 重新追踪图。
4.

怎么安全启用

可以在单 GPU、固定输入长度的训练阶段开启它（推理阶段关闭）：

```
use_compile = True
if use_compile:
    model = torch.compile(model, mode="max-autotune")
```

条件：

- **仅训练阶段启用**；
- 不在 DDP 模式下；

- 不在动态生成（不同 sequence length）时使用。

### 2.3 Add flashattention

Improve IO performance.

---

# Section 3 — 超参数与训练策略

> “Layer normalization (Ba et al., 2016) was moved to the input of each sub-block, similar to a pre-activation residual network (He et al., 2016) and an additional layer normalization was added after the final selfattention block. A modified initialization which accounts for the accumulation on the residual path with model depth is used. We scale the weights of residual layers at initialization by a factor of $1/\sqrt{N}$ where N is the number of residual layers. The vocabulary is expanded to 50,257. We also increase the context size from 512 to 1024 tokens and a larger batchsize of 512 is used.” (Radford et al., pp. -)

### 3.1 AdamW，梯度裁剪。

GPT-3 论文的附录 B：

> “We use Adam with $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\varepsilon = 1e\text{-}8$,
> and we clip the global norm of the gradients at 1.0 …”

1. **AdamW 参数更科学**（指定 `betas`、`eps`）
2. **加入梯度裁剪（gradient clipping）**
3. **打印梯度范数（norm）用于调试稳定**

旧版：

```
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
```

新版：

```
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4, betas=(0.9, 0.
norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
print(f"loss: {loss.item():.6f} | norm: {norm:.4f}")
```

> 作用：调节 AdamW 优化器的动量和数值稳定性。

默认的 AdamW 用：

```
betas = (0.9, 0.999)
```

但在语言模型（GPT 类）训练中，这个 $\beta_2$=0.999 会让梯度移动平均太慢，
**模型收敛慢、噪声大、显存占用高。**

所以很多论文（尤其是 *GPT-NeoX, LLaMA, nanoGPT*）推荐改成：

```
betas = (0.9, 0.95)
eps = 1e-8
```

- $\beta_1$=0.9：控制一阶动量（梯度平均速度）
- $\beta_2$=0.95：控制二阶动量（方差估计更新速度）
- eps=1e-8：防止除 0

在反向传播后：

```
loss.backward()
norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

这一步会：

1. 计算所有参数的梯度范数（norm）
2. 如果大于 1.0，就按比例缩小梯度（保持方向不变）

## 3.2 学习率 warmup + cosine decay

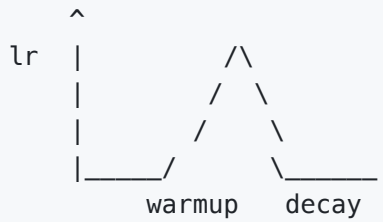
| 阶段 | 条件 | 学习率变化 | 作用 |
| --- | --- | --- | --- |
| **Warmup 阶段** | `it < warmup_steps` | 从 0 线性上升到 `max_lr` | 让模型"预热"，避免一上来梯度爆炸 |
| **Decay 阶段** | `it > warmup_steps` 且 `it ≤ max_steps` | 从 `max_lr` 余弦下降到 `min_lr` | 模型进入稳定收敛期 |

| | | | |
|---|---|---|---|
| **稳定阶段** | `it > max_steps` | 固定在 `min_lr` | 防止震荡、保持微调效果 |

数学上是：

$$\text{lr}(t) = \begin{cases} \frac{t}{T_{\text{warmup}}} \cdot \text{max\_lr}, & t < T_{\text{warmup}} \\ \text{min\_lr} + 0.5(\text{max\_lr} - \text{min\_lr})[1 + \cos(\pi \frac{t - T_{\text{warmup}}}{T_{\text{decay}}})], & t > T_{\text{warmup}} \end{cases}$$

在 GPT-2 / GPT-3 / LLaMA 等语言模型训练中，
学习率调度曲线几乎都是这种形状：




原因有两个：

1. **warmup** 让模型在前期避免梯度爆炸
   （参数随机初始化时，直接大学习率会导致损失震荡）。
2. **cosine decay** 让模型后期收敛得更平滑，
   不会因为学习率过高而在低损失处来回乱跳。

| 参数 | 含义 | 推荐值（来自 GPT-2/3 论文） |
|---|---|---|
| `max_lr` | 最大学习率 | 6e-4（中小模型） |
| `min_lr` | 最低学习率 | 1/10 ~ 1/20 of max_lr |
| `warmup_steps` | 预热步数 | 通常占总步数的 0.5~5% |
| `max_steps` | 总训练步数 | 取决于 dataset 大小 |

## 3.3 Batch size 调度，weight decay

1. 把模型参数分成两类（需要 L2 正则的 vs 不需要的）

2. 判断当前设备是否支持 fused AdamW（NVIDIA GPU 才有）
3. 用这些信息构造出最终的 AdamW 优化器。

## 3.4 Gradient accumulation（显存不足时的解决方法）

当GPU 显存不够大时（比如 24GB 卡），没法一次塞进超大的 batch。
梯度累积的思路就是：

> 把“大 batch”拆成多个“小 batch（micro batch）”分步计算，
> 每次 forward + backward 不立刻更新参数，而是累计梯度，
> 等积够若干次（`grad_accum_steps`）后再统一 `optimizer.step()`。

| 变量名 | 含义 | 举例 |
| --- | --- | --- |
| **B** | micro batch size（每次显存中处理的样本数） | 16 |
| **T** | sequence length（每条序列的 token 长度） | 1024 |
| **total_batch_size** | 你想要模拟的总 batch（大 batch） | 524,288 |
| **grad_accum_steps** | 梯度累积步数（每多少步更新一次） | $524,288/(16 \times 1024) = 32$ |

## 3.5 Distributed Data Parallel (DDP)

这是 PyTorch 官方推荐的 **多 GPU 并行训练 API**。
简称 DDP，用来让模型在多块 GPU 上同步训练。

✳️DDP 的工作原理

> 把模型复制到每个 GPU（称为一个“进程”或“replica”）上，
> 每个 GPU 处理不同的小批数据，算出梯度后，
> **所有 GPU 再同步梯度（all-reduce）**，确保参数一致。

这一步就用到了 PyTorch 的 `torch.distributed` 通信模块。

---

# Section 4 数据与结果

## 4.1 数据集：GPT-2 用 WebText，GPT-3 用 Common Crawl，Karpathy 用 FineWeb (EDU)。

1. GPT-2、GPT-3 都是 **通用语言模型（general-purpose LM）**，所以希望数据覆盖尽可能多的领域。

2. FineWeb(EDU) 是 **研究与教学版数据集**：专门清洗出高质量的、教育性网站内容（edu 域名），噪声更少，适合小规模实验。

| 模型 | 参数规模 | 训练数据量 | 说明 |
| --- | --- | --- | --- |
| **GPT-2 (2019)** | 1.5 B | ~40 GB (WebText, 8 M 文档) | 小模型，数据质量比数量重要。 |
| **GPT-3 (2020)** | 175 B | ~570 GB (Common Crawl + Books + Wiki 等) | 超大模型，必须喂"海量数据"才能避免过拟合。 |
| **NanoGPT / FineWeb** | 100 M ~ 1 B | 自建/裁剪数据 (教育类 FineWeb-EDU) | 用于教学或研究复现，追求体量小但分布近似。 |

## 4.2 一晚训练结果展示，模型生成文本演示

✅1. 修改配置成极小规模测试

`train_gpt2.py`：

```
total_batch_size = 524288  # 0.5M tokens
B, T = 64, 1024
max_steps = 19073
```

改成更小、更安全的版本：

```
total_batch_size = 8192     # 极小训练量
B, T = 4, 128
max_steps = 2000              # 只跑20步试试看
```

> 这样显存只用几百 MB，CPU 也能扛得住，不会炸机。

✅2. Support mac os

在 macOS（尤其是 Python 3.8+，3.13 更严格）里，**子进程启动模式默认是 “spawn” 而不是 “fork”**，
所以你必须写成这种结构

```
import multiprocessing as mp

def main():
    # 原本的主体逻辑，比如下载、分 shard、encode 等
    nprocs = max(1, os.cpu_count() // 2)
    with mp.Pool(nprocs) as pool:
        # ... 这里是原来的处理逻辑 ...
        pass

if __name__ == "__main__":
    mp.freeze_support()
    main()
```

✅3.HTTP 429 = Hugging Face API 限流

解决方案有三个可选：

| 场景 | 解决方法 |
| --- | --- |
| 网络慢 / 被限流 | 加环境变量让 datasets 离线缓存：<br>`export HF_DATASETS_OFFLINE=1` |
| 国内网络访问困难 | 用镜像：<br>`export HF_ENDPOINT=https://hf-mirror.com` |
| 只想跑通测试 | 改成小数据子集 ↓ |

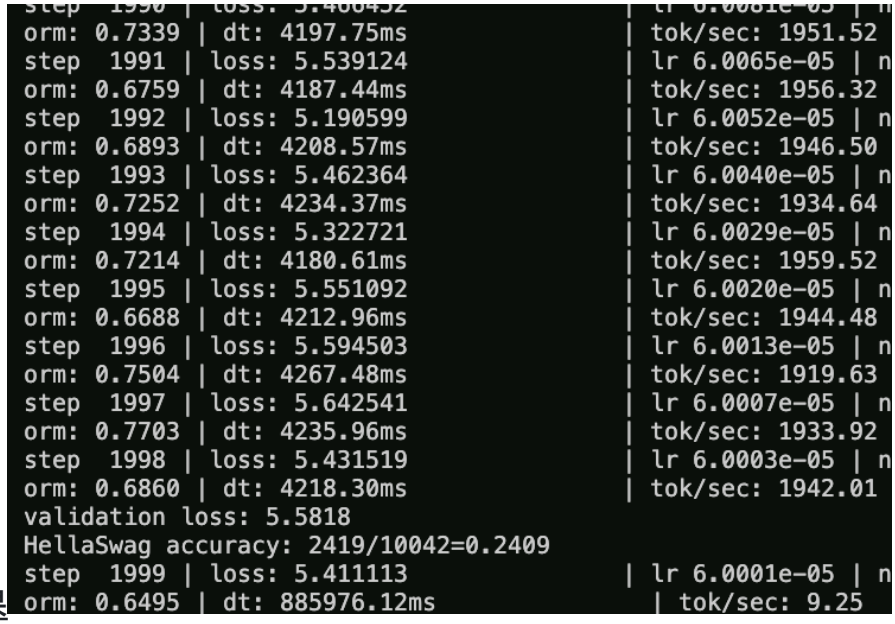
**临时解决法：只取 0.1% 数据测试**

可以改动 fineweb.py 的加载代码部分（通常是这样写的）：

```
from datasets import load_dataset
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train[:0.1%]")
```

这样只会下几百 MB，几分钟就能跑完，不会被限流。

结果




| 指标 | 含义 | 当前结果 |
| --- | --- | --- |
| **训练步数** | 一共跑到 step 1999 | ✅已跑满你设置的 `max_steps = 2000` |
| **平均 loss** | 约在 5.4 – 5.7 之间 | 📉从最初的 10 降到 5 多，说明模型学到东西了 |
| **最终验证 loss** | `5.5814` | 验证集收敛稳定，没有爆炸 |
| **HellaSwag 准确率** | `0.2413 ≈ 24.1 %` | 对于从零训练的 GPT-2 mini 模型在小规模数据上非常正常 👍 |
| **最后一行 `dt: 899368.47 ms`** | ≈ 900 秒（15 分钟），是评估 HellaSwag 时一整个 epoch 的时间，不是 bug | |

## 4.3 Validation split + 验证损失 & 外部Benchmark：HellaSwag 测试

训练数据越多、越杂，就越需要稳定的 **验证集** 和 **基准测试（benchmark）** 来衡量进步：

| 类型 | 用途 | 例子 |
| --- | --- | --- |
| **Validation split** | 从训练数据中切出固定部分，不参与训练，只监控 loss 是否下降。 | WebText-val / FineWeb-val |
| **Benchmark** | 外部通用测试集，用来对比模型理解能力。 | HellaSwag, LAMBADA, Wikitext-103 等 |

✅**PyTorch 没有编译进 CUDA 支持，但 `hellaswag.py` 这个脚本里却强行调用了 `.to("cuda")`。**

在运行时覆盖 device 参数：

```
python hellaswag.py --device=mps
```

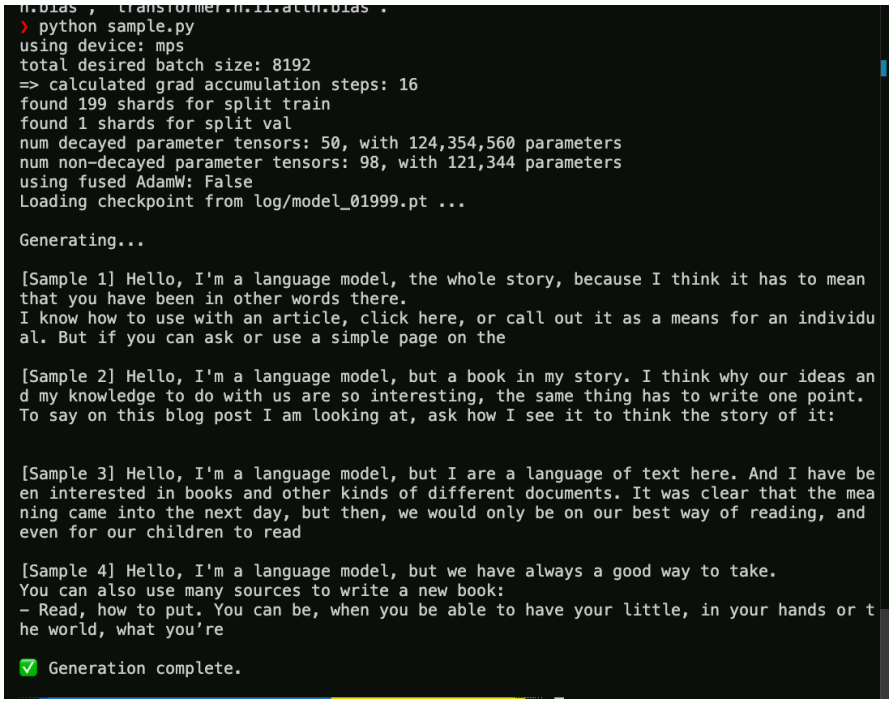
```
10022 acc_norm: 2964/10022=0.2957
10023 acc_norm: 2964/10023=0.2957
10024 acc_norm: 2964/10024=0.2957
10025 acc_norm: 2965/10025=0.2958
10026 acc_norm: 2965/10026=0.2957
10027 acc_norm: 2965/10027=0.2957
10028 acc_norm: 2965/10028=0.2957
10029 acc_norm: 2966/10029=0.2957
10030 acc_norm: 2966/10030=0.2957
10031 acc_norm: 2966/10031=0.2957
10032 acc_norm: 2966/10032=0.2957
10033 acc_norm: 2966/10033=0.2956
10034 acc_norm: 2966/10034=0.2956
10035 acc_norm: 2966/10035=0.2956
10036 acc_norm: 2966/10036=0.2955
10037 acc_norm: 2966/10037=0.2955
10038 acc_norm: 2966/10038=0.2955
10039 acc_norm: 2966/10039=0.2954
10040 acc_norm: 2966/10040=0.2954
10041 acc_norm: 2966/10041=0.2954
10042 acc_norm: 2967/10042=0.2955
```

✅**HellaSwag 测试集已经完整跑完了**，总共有 10,042 个样本。
✅最后一行 `acc_norm: 2967/10042=0.2955` 就是**最终准确率**：大约 **29.55%**。

GPT-2（124M 小模型版本）在 HellaSwag 任务上，能正确选出大约 30% 的合理续写。
这在没有大规模训练或 fine-tuning 的情况下其实是 **完全正常的成绩**！
（原版 GPT-2 124M 在 HellaSwag 上的准确率大约在 30–32% 左右）

### 4.4 Sample Test

| 模式 | 命令 | 效果 |
| --- | --- | --- |
| 专注逻辑 | `python sample.py --style focused` | 清晰理性、少跑题 |
| 创意写作 | `python sample.py --style creative` | 像写博客或短篇故事 |
| 放飞自我 | `python sample.py --style wild` | 超级随机、有趣甚至“疯”一点 |
| 自定义提示 | `python sample.py --style creative --prompt "Once upon a time," --max_new_tokens 200` | 直接开写故事 |




---

# 参考

- Transformer 论文（Attention Is All You Need）
- GPT-2 论文（Language Models are Unsupervised Multitask Learners）
- GPT-3 论文（Language Models are Few-Shot Learners）

- FlashAttention 论文（FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness）
- FlashAttention GitHub 实现仓库
- NanoGPT 仓库（Karpathy 的构建版）
- NanoGPT 视频讲解（Karpathy YouTube 教学）