下载离线版

离线阅读 llama.cpp 学习笔记 的全部内容。

PDF

保留原始排版、代码高亮和流程图，适合在电脑或平板上阅读。

下载 PDF

EPUB

适合在电子书阅读器（Kindle、Apple Books、Calibre 等）上阅读。

下载 EPUB

> 后端 (Backend) 是 GGML 的硬件抽象层，定义了统一的接口将计算图分发到不同硬件设备执行。

为什么需要 backend

不同硬件（CPU、GPU、TPU）有不同的编程模型和内存管理方式。后端抽象使得上层代码（如 Transformer 推理）无需关心硬件细节，同一套计算图可以在不同设备上执行。

核心原理

```
struct ggml_backend {
    struct ggml_backend_i iface;  // 函数指针表（接口）
    void * context;               // 后端私有数据
};

struct ggml_backend_i {
    const char * (*get_name)(ggml_backend_t);
    void (*free)(ggml_backend_t);
    ggml_backend_buffer_t (*alloc_buffer)(ggml_backend_t, size_t);
    bool (*supports_op)(ggml_backend_t, const ggml_tensor *);
    ggml_status (*graph_compute)(ggml_backend_t, ggml_cgraph *);
    // ...
};
```

后端类型：

- **CPU** — 通用后端，SIMD 加速 (AVX2/NEON)
- **CUDA** — NVIDIA GPU
- **Metal** — Apple Silicon GPU
- **Vulkan** — 跨平台 GPU
- **SYCL** — Intel GPU
- **RPC** — 远程执行

在源码中的实现

- `ggml/src/ggml-backend.cpp` — 后端注册与调度
- `ggml/src/ggml-cpu/` — CPU 后端实现
- `ggml/src/ggml-cuda/` — CUDA 后端实现
- `ggml/src/ggml-metal/` — Metal 后端实现
- `ggml/src/ggml-vulkan/` — Vulkan 后端实现

相关概念

- ggml — 后端执行的底层库
- tensor — 后端管理的张量内存
- compute-graph — 后端执行的计算图
- quantization — 各后端的量化 kernel 实现

> 批量解码 (Batch Decode) 是 llama.cpp 的批处理机制，允许在一次 `llama_decode` 调用中同时处理多个 token 或多个序列。

为什么需要 batch-decode

单 token 解码（每次处理 1 个 token）无法充分利用 GPU 的并行计算能力。批量解码将多个 token 或多个序列的推理合并到一次调用中，提高硬件利用率。

核心原理

```
struct llama_batch {
    llama_token * token;      // token IDs 数组
    int32_t     * pos;        // 每个 token 的位置
    int32_t     * n_seq_id;   // 每个 token 的序列数
    llama_seq_id ** seq_id;   // 每个 token 的序列 ID
    int32_t       n_tokens;   // 总 token 数
};
```

使用场景：

- **Prefill** — 将整个 prompt（多个 token）一次处理
- **Parallel Decoding** — 多个请求的 token 合并到一个 batch
- **Speculative Decoding** — 候选 token 批量验证

Prefill vs Decode：

- Prefill: batch 中包含同一序列的多个 token（矩阵 × 矩阵）
- Decode: batch 中包含不同序列的单个 token（矩阵 × 多向量）

在源码中的实现

- `src/llama-batch.cpp` — batch 构建工具

- `src/llama-context.cpp` — `llama_decode()` 执行 batch
- `include/llama.h` — `llama_batch` 结构体定义
- `tools/server/` — server 的并发请求合并到 batch

相关概念

- kv-cache — batch 中每个 token 的 K/V 存入 cache
- backend — 后端负责 batch 的并行执行

> 计算图 (Compute Graph) 是 GGML 中描述张量运算关系的数据结构，以有向无环图 (DAG) 表示，支持前向传播和自动微分。

为什么需要 compute-graph

LLM 的 forward pass 涉及数百个张量运算（矩阵乘法、归一化、激活函数等）。计算图将这些运算组织为 DAG，使得 GGML 可以：

- 自动推导运算顺序
- 优化执行计划（算子融合、内存复用）
- 支持自动微分（反向传播）
- 将运算分发到不同硬件后端

核心原理

```
struct ggml_cgraph {
    int n_nodes;                        // 节点数
    struct ggml_tensor * nodes[GGML_MAX_NODES]; // 运算节点
    int n_leafs;                        // 叶子节点数
    struct ggml_tensor * leafs[GGML_MAX_NODES]; // 输入张量
};
```

工作流程：

1. 构建 — 用户通过 `ggml_mul_mat()`, `ggml_add()` 等创建张量运算
2. 收集 — `ggml_build_forward_expand()` 将目标张量的所有依赖加入图
3. 执行 — `ggml_graph_compute()` 按拓扑序遍历并计算每个节点
4. 微分 — `ggml_build_backward_expand()` 自动构建梯度计算图

在源码中的实现

- `ggml/include/ggml.h` — `ggml_cgraph` 结构和图操作 API
- `ggml/src/ggml.c` — 图构建和执行实现
- `ggml/src/ggml-backend.cpp` — 后端执行计算图
- `src/llama-model.cpp` — 为 Transformer 层构建计算图

相关概念

- ggml — 计算图所在的张量库
- tensor — 计算图中的节点
- backend — 计算图的执行后端

> GGML (Georgi Gerganov Machine Learning) 是 llama.cpp 的底层张量计算库，提供张量定义、计算图构建和自动微分能力。

为什么需要 ggml

llama.cpp 需要一个轻量、高效的张量计算库来执行 LLM 推理。GGML 被设计为无外部依赖的纯 C 实现，这使得 llama.cpp 可以在几乎任何平台上编译运行。

核心原理

GGML 的核心设计包括：

1. 张量抽象 — 统一的 `ggml_tensor` 结构体表示 n 维数组
2. 计算图 — 用 DAG 描述运算关系，支持前向和反向传播
3. 后端抽象 — 通过 `ggml_backend` 接口将计算分发到不同硬件
4. 量化支持 — 内置多种量化数据类型，从 F32 到 Q2_K

关键数据结构：

- `ggml_context` — 内存池，管理所有张量分配
- `ggml_tensor` — 张量，包含维度、步长、数据指针
- `ggml_cgraph` — 计算图，描述张量间的运算

在源码中的实现

- `ggml/include/ggml.h` — 公共 C API
- `ggml/src/ggml.c` — 核心实现（张量操作、计算图）
- `ggml/src/ggml-backend.cpp` — 后端抽象层
- `ggml/src/ggml-cpu/` — CPU 后端
- `ggml/src/ggml-cuda/` — CUDA 后端

- `ggml/src/ggml-metal/` — Metal 后端

相关概念

- tensor — GGML 的张量数据结构
- compute-graph — GGML 的计算图设计
- backend — GGML 的后端抽象接口
- gguf — 基于 GGML 的模型文件格式

> GGUF (GGML Universal File) 是 llama.cpp 使用的统一模型文件格式，将模型权重、词表和元数据打包在单一二进制文件中。

为什么需要 **gguf**

LLM 模型通常包含权重、词表、超参数等多种数据，分散存储不利于分发和使用。GGUF 将所有信息打包到单一文件中，支持 mmap 高效加载，且设计为可扩展格式。

核心原理

GGUF 文件结构：

```
┌──────────────────────┐
│ Header               │  magic (0x46475547) + version + tensor_count + metadata_co
unt
├──────────────────────┤
│ Metadata KV Pairs    │  key-value 存储（架构、超参数、词表等）
├──────────────────────┤
│ Tensor Info Array    │  每个张量的名称、维度、类型、偏移量
├──────────────────────┤
│ Alignment Padding    │  对齐到指定边界
├──────────────────────┤
│ Tensor Data          │  所有张量的二进制数据
└──────────────────────┘
```

关键特性：

- 单一文件 — 权重 + 词表 + 配置一体化
- **mmap** 友好 — 支持内存映射，按需加载权重页
- 量化内置 — 张量可以是任意 GGML 数据类型
- 可扩展 — 通过 metadata KV 支持任意扩展信息

在源码中的实现

- `ggml/src/gguf.cpp` — GGUF 读写实现
- `src/llama-model-loader.cpp` — 使用 GGUF API 加载模型
- `gguf-py/` — Python GGUF 工具库
- `convert_hf_to_gguf.py` — HuggingFace 模型转 GGUF

相关概念

- ggml — GGUF 底层的张量库
- tensor — GGUF 中存储的张量数据
- quantization — GGUF 支持的量化类型

> Importance Matrix (imatrix) 通过校准数据统计各张量的重要性，指导量化时保留关键权重，提升量化模型的质量。

为什么需要 **imatrix**

标准量化对所有权重一视同仁，但模型中不同权重对输出的影响差异很大。imatrix 识别出更"重要"的权重，在量化时给予更高精度，从而在相同压缩率下获得更好的模型质量。

核心原理

1. 收集统计 — 在校准数据上运行推理，记录每个张量的激活值
2. 计算重要性 — 基于激活值的统计量（如方差、范数）评估重要性
3. 指导量化 — 重要的张量使用更高精度的量化类型

```
# 步骤 1: 生成 imatrix
./llama-imatrix -m model-F16.gguf -f calibration.txt -o imatrix.dat

# 步骤 2: 使用 imatrix 量化
./llama-quantize --imatrix imatrix.dat model-F16.gguf model-Q4_K_M.gguf Q4_K_M
```

> 校准数据通常使用 ~100K-1M token 的多样化文本。

在源码中的实现

- `tools/imatrix/` — imatrix 生成工具
- `ggml/src/ggml-quants.c` — 量化时使用 imatrix 数据
- `tools/quantize/` — 量化工具支持 `--imatrix` 参数

相关概念

- quantization — imatrix 服务的量化过程
- gguf — imatrix 数据可以嵌入 GGUF 文件
- ggml — 量化在 GGML 层实现

> KV Cache 缓存 Transformer 推理中已计算的 Key 和 Value 向量，避免重复计算，是自回归生成的核心优化。

为什么需要 **kv-cache**

Transformer 自回归生成时，每生成一个新 token 都需要对之前所有 token 做注意力计算。KV Cache 将已计算的 K/V 向量缓存起来，使得每步只需计算当前 token 的 Q/K/V，然后与缓存的 K/V 做注意力。

核心原理

```
Prefill 阶段：处理整个 prompt
  token[0..N] → 计算 K[0..N], V[0..N] → 存入 cache

Decode 阶段：逐个生成
  token[N+1] → 计算 Q, K, V → K,V 追加到 cache
             → Attention(Q, K_cache, V_cache) → 输出 logits
```

内存布局（每层）：

```
K_cache: [max_positions, kv_hidden_dim]
V_cache: [max_positions, kv_hidden_dim]
```

关键优化：

- **GQA** — Grouped Query Attention 减少 KV head 数
- **Prefix Caching** — 复用已处理的 prompt 前缀
- **Rolling Cache** — 淘汰旧位置以支持无限生成

在源码中的实现

- `src/llama-memory.cpp` — KV Cache 管理逻辑
- `src/llama-kv-cache.cpp` — KV Cache 核心实现
- `src/llama-kv-cells.h` — Cache 单元定义
- `src/llama-context.cpp` — decode 时的 cache 更新

相关概念

- batch-decode — 批量解码利用 KV Cache
- backend — 不同后端的 cache 内存管理
- ggml — cache 使用 GGML 张量存储

> 量化 (Quantization) 将模型权重从高精度浮点数压缩为低比特整数，减少内存占用和计算量。

为什么需要 quantization

LLM 模型通常需要数十 GB 内存（如 LLaMA-70B F16 需要 ~140GB）。量化将权重量化到 4-8 bits，使模型能在消费级硬件上运行，同时保持可接受的精度损失。

核心原理

GGML 使用 **Block Quantization** 方案，以 block（通常 32 个权重）为量化单位：

```
Block Q4_0 (18 bytes / 32 weights = 4.5 bits/weight):
┌──────────┬──────────────────────────────┐
│ d (F16)  │ 16 × uint8                   │
│ 缩放因子  │ 每个 uint8 存 2 个 4-bit 值 │
└──────────┴──────────────────────────────┘

反量化: weight[i] = (quantized[i] - 8) × d
```

量化类型层级：

- 基础量化 — Q4_0, Q4_1, Q5_0, Q5_1, Q8_0
- **K-quant** — Q2_K, Q3_K, Q4_K, Q5_K, Q6_K（混合精度）
- **I-quant** — IQ2, IQ3, IQ4（重要性加权的超低比特）

精度排序：F16 > Q8_0 > Q6_K > Q5_K > Q4_K > Q3_K > Q2_K

在源码中的实现

- `ggml/include/ggml.h` — `ggml_type` 枚举定义
- `ggml/src/ggml-quants.c` — 量化/反量化实现

- `ggml/src/ggml-cpu/ggml-cpu-quants.c` — CPU 量化 kernel
- `tools/quantize/` — 量化命令行工具

相关概念

- ggml — 量化数据类型的定义
- tensor — 量化张量的存储
- imatrix — 提升量化质量的校准技术
- backend — 各后端的量化运算实现

> RoPE (Rotary Position Embedding) 是 llama.cpp 支持的位置编码方案，通过旋转矩阵将位置信息注入 Q 和 K 向量。

为什么需要 **rope**

Transformer 本身没有位置感知能力，需要位置编码来区分不同位置的 token。RoPE 通过在注意力计算中自然地引入相对位置信息，效果优于绝对位置编码。

核心原理

RoPE 将 Q 和 K 的相邻维度组成二维子空间，施加旋转：

```
对于位置 pos 和维度对 (2i, 2i+1):
θ_i = pos / 10000^(2i/d)

RoPE(x_{2i}, x_{2i+1}, pos) = [
    x_{2i}   × cos(θ_i) - x_{2i+1} × sin(θ_i),
    x_{2i}   × sin(θ_i) + x_{2i+1} × cos(θ_i)
]
```

> 关键性质：`<RoPE(Q,m), RoPE(K,n)>` 只依赖于 `Q, K` 和相对位置 `m-n`。
>
> 变体：

- **LLaMA RoPE** — 标准实现
- **RoPE Neox** — 调整频率基准 (base)
- **mRoPE** — 多维 RoPE（多模态模型）
- **LongRoPE** — 支持更长上下文的外推

在源码中的实现

- `ggml/src/ggml.c` — `ggml_rope` 操作实现

- `ggml/src/ggml-cpu/` — CPU RoPE kernel
- `ggml/src/ggml-cuda/rope.cu` — CUDA RoPE kernel
- `src/llama-model.cpp` — forward pass 中调用 RoPE

相关概念

- ggml — RoPE 作为 GGML 操作实现
- compute-graph — RoPE 在计算图中的位置
- kv-cache — RoPE 位置与 KV Cache 的关系

> 采样器链 (Sampler Chain) 是 llama.cpp 的可组合采样管道，将多个采样器串联处理 logits 输出最终 token。

为什么需要 **sampler-chain**

LLM 生成的原始 logits 需要经过多步处理才能得到高质量的 token 选择。采样器链允许灵活组合不同的采样策略（温度、top-k、top-p、grammar 等），并支持动态调整。

核心原理

```
// 创建采样器链
struct llama_sampler * chain = llama_sampler_chain_init(params);

// 按顺序添加采样器（顺序影响结果）
llama_sampler_chain_add(chain, llama_sampler_init_temp(0.8));
llama_sampler_chain_add(chain, llama_sampler_init_top_k(40));
llama_sampler_chain_add(chain, llama_sampler_init_top_p(0.95, 1));
llama_sampler_chain_add(chain, llama_sampler_init_dist(seed));

// 采样
llama_token token = llama_sampler_sample(chain, ctx, -1);
```

每个采样器实现统一接口：

- `name()` — 采样器名称
- `apply()` — 处理 logits 或直接采样
- `accept()` — 通知采样器已接受的 token
- `reset()` — 重置状态

常见采样器：Temperature → Top-K → Top-P → Min-P → Typical → Mirostat → Grammar → 重复惩罚

在源码中的实现

- `src/llama-sampler.cpp` — 所有采样器的实现
- `include/llama.h` — 采样器 API 声明
- `common/sampling.cpp` — 高级采样配置封装

相关概念

- ggml — logits 是 GGML 张量
- quantization — 量化影响 logits 精度
- tokenization — 采样的 token 通过词表映射回文本

> 张量 (Tensor) 是 GGML 中的基本数据单元，表示 n 维数组，支持多种数据类型包括浮点和量化格式。

为什么需要 tensor

LLM 推理本质上是大量的矩阵/张量运算。GGML 需要一个统一的张量表示来支持各种运算（矩阵乘法、卷积、归一化等），同时要能表示不同精度的数据以实现量化推理。

核心原理

```
struct ggml_tensor {
    enum ggml_type type;         // 数据类型
    int n_dims;                  // 维度数 (1-4)
    int64_t ne[GGML_MAX_DIMS];   // 每个维度的元素数
    size_t nb[GGML_MAX_DIMS];    // 每个维度的字节步长
    void * data;                 // 数据指针
    struct ggml_tensor * grad;   // 梯度 (用于自动微分)
};
```

维度约定：

- `ne[0]` 是最内层维度（行方向）
- `ne[n_dims-1]` 是最外层维度（batch）
- `nb[0]` = 单个元素的字节数
- `nb[i]` = `nb[i-1] * ne[i-1]`

数据类型：F32、F16、BF16、Q4_0、Q4_1、Q5_0、Q5_1、Q8_0、IQ 系列、K-quants 等。

在源码中的实现

- `ggml/include/ggml.h` — `ggml_tensor` 结构体定义
- `ggml/src/ggml.c` — 张量创建和管理函数

- `ggml/include/ggml-alloc.h` — 张量内存分配

相关概念

- ggml — 张量所在的计算库
- compute-graph — 张量在计算图中的角色
- quantization — 张量的量化数据类型
- backend — 张量在不同设备上的内存管理

> 分词 (Tokenization) 将输入文本拆分为模型能处理的 token ID 序列，是 LLM 推理的第一步。

为什么需要 **tokenization**

模型不能直接处理原始文本，需要将文本转换为固定词表中的 token ID 序列。分词算法的质量直接影响模型的处理效率和能力（如多语言支持、代码处理等）。

核心原理

llama.cpp 支持多种分词算法：

类型	模型	算法
SPM	LLaMA, Mistral	SentencePiece (BPE + byte fallback)
BPE	GPT-2, Qwen	Byte Pair Encoding
WPM	BERT	WordPiece
UGM	T5	Unigram
RWKV	RWKV	Greedy tokenization

分词流程：

1. 文本预处理（Unicode 规范化）
2. 预分词（按规则分割为词）
3. 子词编码（BPE 合并 / SPM 查找）
4. 映射为 token ID
5. 添加特殊 token（BOS 等）

在源码中的实现

- `src/llama-vocab.cpp` — 分词器实现（所有算法）
- `src/unicode.cpp` — Unicode 处理
- `src/unicode-data.cpp` — Unicode 数据表
- `include/llama.h` — `llama_tokenize()` API

相关概念

- gguf — 词表数据存储在 GGUF 文件中
- sampler-chain — 分词是逆过程（token → text = detokenization）

llamacpp

llama.cpp 学习笔记

llama.cpp 学习笔记

开始学习 知识地图

GGML 张量计算库

底层张量运算与计算图，支持自动微分与多种数据类型

多后端硬件加速

CPU (SIMD)、CUDA、Metal、Vulkan 等多种硬件后端统一抽象

GGUF 模型格式

统一的模型权重与元数据容器，支持 mmap 高效加载

丰富量化方案

从 1.5-bit 到 8-bit 的量化支持，平衡精度与性能

可组合采样器链

top-k/top-p/mirostat/grammar 等采样器灵活组合

OpenAI 兼容服务器

llama-server 提供 HTTP API，兼容 OpenAI 接口格式

学习日志

进度总览

模块	状态	笔记
01 项目概览与构建系统	⬜ 未开始	
02 GGML 张量库基础	⬜ 未开始	
03 GGML 后端与硬件抽象	⬜ 未开始	
04 模型加载与 GGUF 格式	⬜ 未开始	
05 分词与词表	⬜ 未开始	
06 Transformer 推理图	⬜ 未开始	
07 KV Cache 与批处理	⬜ 未开始	
08 采样、量化与部署	⬜ 未开始	

2025-05-18

今日摘要

初始化学习笔记站点，开始 llama.cpp 源码学习之旅。

计划

- [ ] 阅读 llama.cpp README 和 CMakeLists.txt
- [ ] 理解项目整体架构
- [ ] 从源码编译 CPU 版本

困惑与突破

（记录学习过程中的困惑点和突破时刻）

笔记

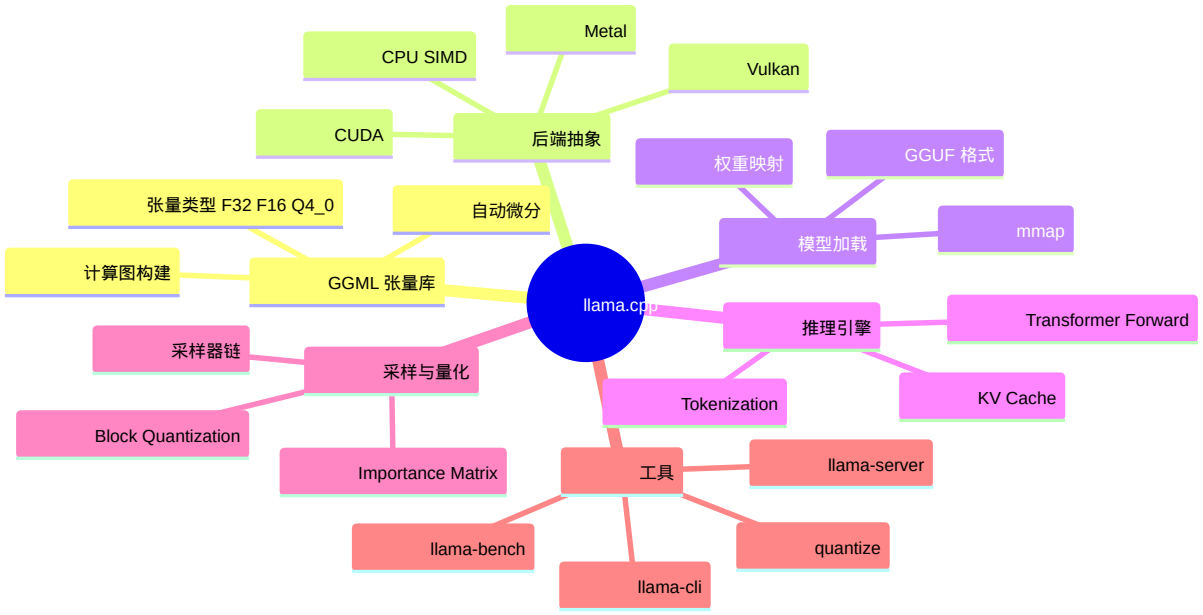
（记录关键发现和心得）

llama.cpp 学习笔记 — 知识地图

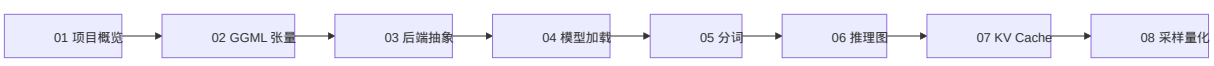
项目概览

llama.cpp 是一个纯 C/C++ 实现的 LLM 推理库，核心特性包括多后端硬件加速、丰富的量化方案和 OpenAI 兼容 API 服务器。

架构全景



学习路径



知识索引

核心架构

概念	主题	术语
张量定义	GGML 张量库	tensor
计算图	GGML 张量库	compute-graph
后端接口	GGML 后端	backend
模型格式	模型加载	gguf

推理核心

概念	主题	术语
位置编码	推理图	rope
分词算法	分词	tokenization
KV 缓存	KV Cache	kv-cache
批量解码	KV Cache	batch-decode

优化与部署

概念	主题	术语
量化方案	采样量化	quantization
采样器链	采样量化	sampler-chain
重要性矩阵	采样量化	imatrix

学习进度

- [ ] 01 项目概览与构建系统
- [ ] 02 GGML 张量库基础
- [ ] 03 GGML 后端与硬件抽象

- [ ] 04 模型加载与 GGUF 格式
- [ ] 05 分词与词表
- [ ] 06 Transformer 推理图
- [ ] 07 KV Cache 与批处理
- [ ] 08 采样、量化与部署

出链

/topics/02-ggml-tensor/index\

/topics/03-ggml-backend/index\

/topics/04-model-loading/index\

/topics/06-inference-graph/index\

/topics/05-tokenization/index\

/topics/07-kv-cache/index\

/topics/08-sampling-quant/index\

项目概览与构建系统 — 代码走读

CMakeLists.txt 入口

顶层 `CMakeLists.txt` 定义了项目的构建结构：

```
# 项目定义
project(llama.cpp C CXX)

# GGML 子项目
add_subdirectory(ggml)

# llama 库
add_library(llama ...)

# 工具目标
add_subdirectory(tools)
```

> 关键构建流程：

1. 先编译 `ggml` 静态库（含选定的后端）
2. 编译 `llama` 库（依赖 ggml）
3. 编译工具和示例

---

include/llama.h — 核心 API

这是整个项目对外的唯一公共头文件，定义了：

核心类型

```
struct llama_vocab;      // 词表（不透明）
struct llama_model;      // 模型（不透明）
struct llama_context;    // 推理上下文（不透明）
struct llama_sampler;    // 采样器（不透明）
typedef int32_t llama_token;  // token ID
typedef int32_t llama_pos;    // 位置编码
```

核心流程

```
// 1. 模型加载
struct llama_model * llama_model_load_from_file(
    const char * path_model, struct llama_model_params params);

// 2. 创建上下文
struct llama_context * llama_init_from_model(
    struct llama_model * model, struct llama_context_params params);

// 3. 分词
int32_t llama_tokenize(
    const struct llama_model * model, const char * text,
    llama_token * tokens, int32_t n_max_tokens, bool add_bos, bool special);

// 4. 推理（batch decode）
int32_t llama_decode(struct llama_context * ctx, struct llama_batch batch);

// 5. 采样
llama_token llama_sampler_sample(struct llama_sampler * smpl,
    struct llama_context * ctx, int32_t idx);

// 6. 清理
void llama_free(struct llama_context * ctx);
void llama_model_free(struct llama_model * model);
```

src/ 目录结构

文件	职责
`llama.cpp`	顶层初始化与注册
`llama-model.cpp`	模型定义与 forward pass
`llama-model-loader.cpp`	GGUF 文件解析与权重加载
`llama-vocab.cpp`	分词器与词表
`llama-sampler.cpp`	可组合采样器链
`llama-context.cpp`	推理上下文管理
`llama-memory.cpp`	KV Cache 管理
`llama-batch.cpp`	批处理编码
`llama-arch.cpp`	模型架构定义
`llama-graph.cpp`	计算图构建
`llama-kv-cache.cpp`	KV Cache 实现

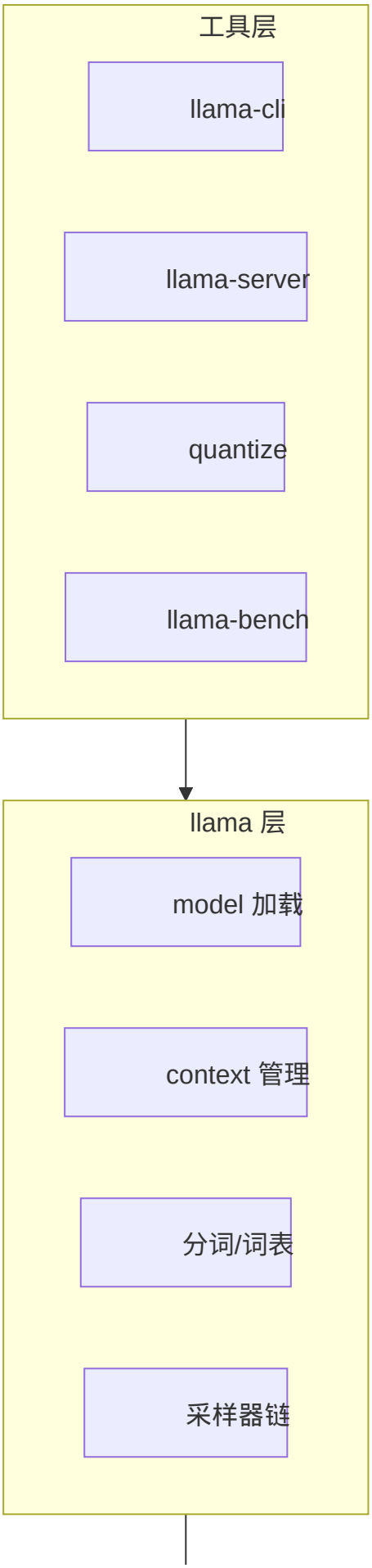
关键函数索引

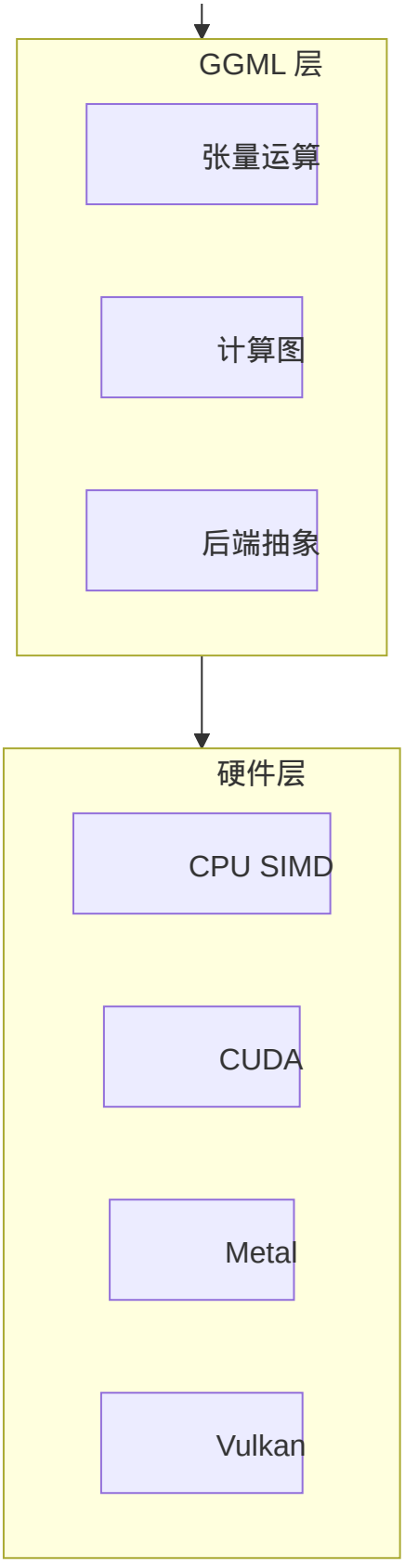
函数	文件	说明
`llama_model_load_from_file`	`llama-model-loader.cpp`	从 GGUF 加载模型
`llama_init_from_model`	`llama-context.cpp`	从模型创建推理上下文
`llama_decode`	`llama-context.cpp`	执行 batch decode
`llama_tokenize`	`llama-vocab.cpp`	文本转 token
`llama_sampler_sample`	`llama-sampler.cpp`	从 logits 采样 token

项目概览与构建系统 — 概念

项目架构

llama.cpp 采用分层架构设计：


工具层
llama-cli
llama-server
quantize
llama-bench
llama 层
model 加载
context 管理
分词/词表
采样器链


GGML 层
张量运算
计算图
后端抽象
硬件层
CPU SIMD
CUDA
Metal
Vulkan

模块划分

目录	职责
`ggml/`	GGML 张量库：张量定义、计算图、后端抽象
`include/llama.h`	对外 C API 接口
`src/`	llama 核心实现：模型加载、推理、采样、KV Cache
`common/`	通用工具：参数解析、采样、日志
`examples/`	示例程序
`tools/`	完整工具：cli、server、quantize、imatrix
`gguf-py/`	Python GGUF 读写工具

构建系统

llama.cpp 使用 CMake 构建，关键选项：

```
# 基础编译
cmake -B build
cmake --build build --config Release

# 启用 CUDA 后端
cmake -B build -DGGML_CUDA=ON

# 启用 Metal 后端 (macOS)
cmake -B build -DGGML_METAL=ON

# 启用 Vulkan 后端
cmake -B build -DGGML_VULKAN=ON
```

核心 CMake 变量：

- `GGML_CUDA` / `GGML_METAL` / `GGML_VULKAN` — 后端开关
- `GGML_BLAS` — BLAS 加速
- `LLAMA_CURL` — 启用 HTTP 下载支持
- `CMAKE_BUILD_TYPE` — Release / Debug

C API 设计哲学

`include/llama.h` 遵循 C 风格 OOP：

- 不透明指针（`llama_model *`, `llama_context *`）
- 创建/销毁配对（`llama_model_load` / `llama_model_free`）
- 枚举驱动的配置（`llama_context_param`）

相关概念

- ggml — 底层张量库
- gguf — 模型文件格式
- backend — 硬件后端抽象
- compute-graph — 计算图概念

项目概览与构建系统 — 练习

练习 1：从源码编译 llama.cpp

将 llama.cpp 克隆到本地，使用 CMake 编译一个仅 CPU 后端的版本：

```
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release -j$(nproc)
```

验证编译产物：确认 `llama-cli` 和 `llama-server` 已生成。

<details>
<summary>参考答案</summary>

编译完成后，在 `build/bin/` 目录下应能找到：

- `llama-cli` — 命令行推理工具
- `llama-server` — OpenAI 兼容 API 服务器

可以运行 `./build/bin/llama-cli --help` 验证是否正常工作。

</details>

练习 2：梳理模块依赖关系

阅读 `CMakeLists.txt`，画出 `ggml` → `llama` → `tools` 的依赖链。思考：

- `llama` 库链接了哪些 `ggml` 的子库？
- `llama-server` 相比 `llama-cli` 多依赖了什么？

参考答案

依赖链：

- `ggml`（核心张量）+ `ggml-base` + `ggml-cpu`（+ 可选 `ggml-cuda` / `ggml-metal`）
- `llama` 链接上述 ggml 子库
- `llama-cli` 链接 `llama` + `common`
- `llama-server` 额外链接 HTTP 库（如 libcurl）、JSON 解析等

在 `tools/server/CMakeLists.txt` 中可以看到 server 的额外依赖。

练习 3：探索 C API 的生命周期

阅读 `include/llama.h`，列出一次完整推理调用的 API 调用顺序（从模型加载到 token 生成）。

参考答案

完整生命周期：

1. `llama_model_default_params()` — 获取默认模型参数
2. `llama_model_load_from_file()` — 加载 GGUF 模型
3. `llama_context_default_params()` — 获取默认上下文参数
4. `llama_init_from_model()` — 创建推理上下文
5. `llama_tokenize()` — 文本转 token
6. `llama_batch_get_one()` — 创建 batch
7. `llama_decode()` — 执行前向传播
8. `llama_sampler_chain_init()` — 创建采样器链
9. `llama_sampler_chain_add()` — 添加采样器
10. `llama_sampler_sample()` — 采样下一个 token
11. `llama_token_to_piece()` — token 转文本
12. 重复 6-11 直到生成结束
13. `llama_sampler_free()` / `llama_free()` / `llama_model_free()` — 清理

拓展挑战

- [ ] 编译 CUDA 后端并在 GPU 上运行推理
- [ ] 使用 `llama-bench` 对比不同量化级别的性能
- [ ] 阅读 `examples/simple/simple.cpp` 理解最小推理示例

项目概览与构建系统

> llama.cpp 是一个纯 C/C++ 实现的 LLM 推理库，以最小依赖和高性能著称，支持多种硬件后端与量化方案。

涵盖内容

章节	核心主题
概念	项目架构、模块划分、构建流程
代码走读	CMakeLists.txt、入口文件、核心头文件
练习	编译配置、模块关系梳理

核心概念

- 纯 **C/C++** 实现 — 无外部依赖，单一代码库包含推理、分词、采样全流程
- 分层架构 — GGML 张量层 → 后端抽象层 → llama 推理层 → 工具层
- 多后端支持 — CPU (SIMD)、CUDA、Metal、Vulkan、SYCL 等
- **CMake** 构建 — 通过选项开关控制后端编译

前置知识

- C/C++ 编译基础（gcc/clang、make）
- CMake 基础语法
- 大语言模型推理的基本概念

学习路径

读完本主题后，你将理解：

- llama.cpp 的整体架构与模块划分
- 如何从源码编译并启用不同后端
- 核心头文件 `llama.h` 提供的 C API 设计
- 从 `ggml` 到 `llama` 的分层关系

→ 下一步：GGML 张量库基础

GGML 张量库基础 — 代码走读

---

ggml/include/ggml.h — 公共 API

这是 GGML 的主要头文件，导出了所有张量操作。

张量创建

```
// 创建上下文（内存池）
struct ggml_context * ggml_init(struct ggml_init_params params);

// 创建张量
struct ggml_tensor * ggml_new_tensor_1d(struct ggml_context * ctx,
    enum ggml_type type, int64_t ne0);
struct ggml_tensor * ggml_new_tensor_2d(struct ggml_context * ctx,
    enum ggml_type type, int64_t ne0, int64_t ne1);
struct ggml_tensor * ggml_new_tensor_4d(struct ggml_context * ctx,
    enum ggml_type type, int64_t ne0, int64_t ne1, int64_t ne2, int64_t ne3);
```

算术运算

```
// 逐元素运算
struct ggml_tensor * ggml_add(ctx, a, b);
struct ggml_tensor * ggml_mul(ctx, a, b);
struct ggml_tensor * ggml_div(ctx, a, b);

// 矩阵乘法
struct ggml_tensor * ggml_mul_mat(ctx, a, b);  // b @ a^T

// 激活函数
struct ggml_tensor * ggml_gelu(ctx, a);
struct ggml_tensor * ggml_silu(ctx, a);
struct ggml_tensor * ggml_relu(ctx, a);
```

计算图执行

```
// 构建计算图
struct ggml_cgraph * ggml_new_graph(struct ggml_context * ctx);
void ggml_build_forward_expand(struct ggml_cgraph * graph, struct ggml_tensor *
tensor);

// 计算
void ggml_graph_compute(struct ggml_cgraph * graph);
```

---

ggml/src/ggml.c — 核心实现

关键实现细节：

张量操作注册

每个张量操作定义为一个 `ggml_op`，包含：

- 前向函数指针
- 反向函数指针（用于自动微分）
- 参数映射（用于不同后端的分发）

内存布局

GGML 使用行主序 (row-major) 存储，`nb[0]` 是最小步长（单个元素的字节数）：

```
对于一个 shape [ne0, ne1] 的张量：
nb[0] = sizeof(element)
nb[1] = nb[0] * ne0  (一行)
```

关键函数索引

函数	说明
`ggml_init`	创建上下文内存池
`ggml_new_tensor_*d`	创建指定维度张量
`ggml_mul_mat`	矩阵乘法（核心运算）
`ggml_new_graph`	创建计算图
`ggml_graph_compute`	执行计算图
`ggml_set_param`	标记可训练参数
`ggml_build_backward_expand`	构建反向传播图

GGML 张量库基础 — 概念

张量 (Tensor)

GGML 中的张量用 `ggml_tensor` 结构体表示：

```
struct ggml_tensor {
    enum ggml_type type;    // 数据类型 (F32, F16, Q4_0, ...)
    int n_dims;             // 维度数
    int64_t ne[GGML_MAX_DIMS]; // 每个维度的大小
    size_t nb[GGML_MAX_DIMS];  // 每个维度的步长 (bytes)
    void * data;            // 数据指针
    struct ggml_tensor * grad; // 梯度张量
    // ... 操作与图信息
};
```

关键设计：

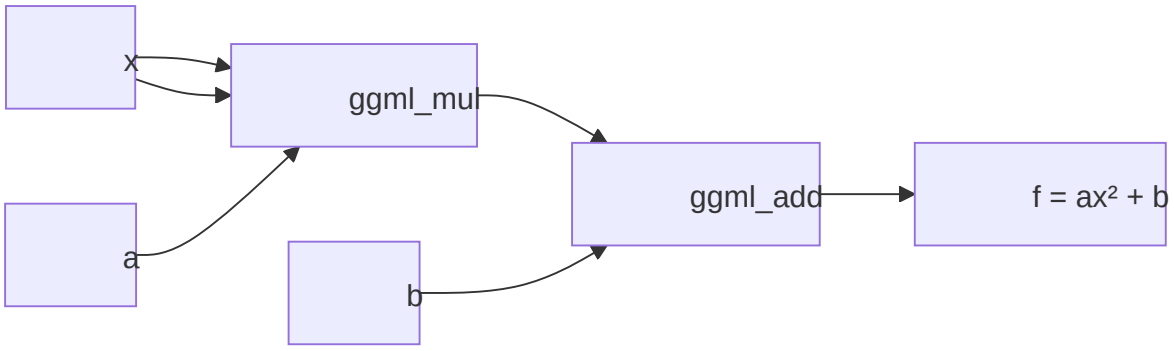
- `ne[]` (number of elements) — 每维元素数，如 `[rows, cols]`
- `nb[]` (number of bytes) — 每维字节步长，支持非连续内存
- `type` — 支持 F32、F16、以及多种量化类型 (Q4_0, Q5_1, Q8_0 等)

数据类型

类型	比特数	说明
`GGML_TYPE_F32`	32	标准 float
`GGML_TYPE_F16`	16	半精度
`GGML_TYPE_BF16`	16	BF16
`GGML_TYPE_Q4_0`	4.5	4-bit 量化 (block)
`GGML_TYPE_Q5_1`	5.5	5-bit 量化
`GGML_TYPE_Q8_0`	8.5	8-bit 量化
`GGML_TYPE_IQ4_XS`	4.25	超低比特量化

计算图

GGML 使用计算图描述运算：



构建方式：

```
struct ggml_context * ctx = ggml_init(params);
struct ggml_tensor * x = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1);
ggml_set_param(ctx, x);
struct ggml_tensor * a = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1);
struct ggml_tensor * b = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1);
struct ggml_tensor * x2 = ggml_mul(ctx, x, x);
struct ggml_tensor * f = ggml_add(ctx, ggml_mul(ctx, a, x2), b);
```

自动微分

- 标记为 `param` 的张量会自动计算梯度
- `ggml_set_param(ctx, tensor)` 将张量标记为可训练参数
- 反向传播通过 `ggml_build_backward_expand()` 自动构建梯度图

内存管理

- `ggml_context` 是一个内存池，所有张量从中分配
- `ggml_init()` 创建上下文，需指定内存大小
- `ggml_free()` 一次性释放所有张量

相关概念

- compute-graph — 计算图的详细设计
- quantization — 张量量化原理
- backend — 计算图在后端上的执行

GGML 张量库基础 — 练习

练习 1：构建简单计算图

用 GGML API 构建函数 `f(x) = 3x² + 2x + 1`，并计算 `x = 2.0` 时的值。

> 提示：使用 `ggml_mul`、`ggml_add` 等操作。

参考答案

```
struct ggml_init_params params = {
    .mem_size = 16 * 1024 * 1024,
    .mem_buffer = NULL,
};
struct ggml_context * ctx = ggml_init(params);

struct ggml_tensor * x = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1);
ggml_set_name(x, "x");

// 常量
float val_3 = 3.0f, val_2 = 2.0f, val_1 = 1.0f;
struct ggml_tensor * c3 = ggml_new_f32(ctx, val_3);
struct ggml_tensor * c2 = ggml_new_f32(ctx, val_2);
struct ggml_tensor * c1 = ggml_new_f32(ctx, val_1);

// 3x² + 2x + 1
struct ggml_tensor * x2 = ggml_mul(ctx, x, x);
struct ggml_tensor * term1 = ggml_mul(ctx, c3, x2);
struct ggml_tensor * term2 = ggml_mul(ctx, c2, x);
struct ggml_tensor * f = ggml_add(ctx, ggml_add(ctx, term1, term2), c1);

// 设置 x = 2.0 并计算
((float *)x->data)[0] = 2.0f;
struct ggml_cgraph * graph = ggml_new_graph(ctx);
ggml_build_forward_expand(graph, f);
ggml_graph_compute(graph);
// 结果: ((float *)f->data)[0] == 17.0
```

练习 2：理解张量内存布局

创建一个 shape 为 `[4, 3]` 的 F32 张量，验证 `nb[0]` 和 `nb[1]` 的值。

参考答案

```
struct ggml_tensor * t = ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 3);
// t->ne[0] = 4, t->ne[1] = 3
// t->nb[0] = sizeof(float) = 4 bytes
// t->nb[1] = 4 * 4 = 16 bytes (一行的大小)
```

练习 3：矩阵乘法维度分析

给定 `A: [m, k]` 和 `B: [k, n]`，使用 `ggml_mul_mat(ctx, A, B)` 后结果的 shape 是什么？

参考答案

`ggml_mul_mat(ctx, A, B)` 计算的是 `B @ A^T`，即 `[k, n] @ [k, m]^T = [k, n] @ [m, k]`。

实际上 GGML 的 `ggml_mul_mat` 结果 shape 是 `[m, n]`：

- A: `ne[0]=k, ne[1]=m` (在 GGML 中是列主序视角)
- B: `ne[0]=n, ne[1]=k`
- 结果: `ne[0]=n, ne[1]=m`

这是因为 GGML 中 `ggml_mul_mat(A, B)` 等价于 `B^T @ A` 的转置形式。具体行为需看 GGML 的定义：`ggml_mul_mat(A, B)` = `sum(A[i][k] * B[j][k])`，结果为 `[ne0_A, ne0_B]` = `[m, n]`。

拓展挑战

- [ ] 使用自动微分计算 `f(x) = x²` 在 `x = 3.0` 处的梯度
- [ ] 阅读量化类型 `Q4_0` 的 block 结构，理解 4.5 bits/weight 的计算方式
- [ ] 对比 `ggml_mul_mat` 在不同后端上的实现差异

GGML 张量库基础

> GGML 是 llama.cpp 的底层张量计算库，提供张量定义、计算图构建和自动微分。

涵盖内容

章节	核心主题
概念	张量、计算图、自动微分
代码走读	ggml.h API、ggml.c 实现
练习	构建计算图、理解前向/反向传播

核心概念

- tensor — n 维数组，GGML 的基本数据单元
- compute-graph — 有向无环图（DAG），描述张量间的运算关系
- 自动微分 — 前向传播后自动计算梯度
- 内存池 — 通过 `ggml_context` 管理张量的内存分配

前置知识

- 线性代数基础（矩阵运算）
- C 语言结构体与指针
- 了解神经网络的 forward/backward pass

学习路径

读完本主题后，你将理解：

- GGML 张量的内存布局与类型系统
- 如何用 C API 构建计算图
- 前向传播与梯度计算的执行流程
- GGML 的内存管理策略

→ 下一步：GGML 后端与硬件抽象

GGML 后端与硬件抽象 — 代码走读

---

ggml/src/ggml-backend.cpp — 后端注册表

后端通过注册表模式管理：

```
// 注册后端
void ggml_backend_register(ggml_backend_t backend);

// 按类型查找后端
ggml_backend_t ggml_backend_get_by_type(enum ggml_backend_type type);
```

核心结构：

```
struct ggml_backend {
    struct ggml_backend_i iface;  // 函数指针表
    void * context;               // 后端私有数据
};
```

`ggml_backend_i` 定义了所有接口方法：

```
struct ggml_backend_i {
    const char * (*get_name)(ggml_backend_t backend);
    void (*free)(ggml_backend_t backend);
    ggml_backend_buffer_t (*alloc_buffer)(ggml_backend_t backend, size_t size);
    size_t (*get_buffer_alignment)(ggml_backend_t backend);
    bool (*supports_op)(ggml_backend_t backend, const struct ggml_tensor * op);
    ggml_status (*graph_compute)(ggml_backend_t backend, struct ggml_cgraph * gr
aph);
    // ...
};
```

---

ggml/src/ggml-cpu/ — CPU 后端

关键文件：

- `ggml-cpu.c` — CPU 后端主实现
- `ggml-cpu.cpp` — C++ 封装
- `ggml-cpu-aarch64.cpp` — ARM NEON 优化
- `ggml-cpu-quants.c` — 量化操作实现

SIMD Kernel 选择

运行时检测 CPU 特性并选择最优 kernel：

```
    #if defined(__AVX512F__)
        // AVX-512 实现 (512-bit SIMD)
    #elif defined(__AVX2__)
        // AVX2 实现 (256-bit SIMD)
    #elif defined(__ARM_NEON)
        // NEON 实现 (128-bit SIMD)
    #else
        // 标量回退
    #endif
```

---

ggml/src/ggml-cuda/ — CUDA 后端

关键文件：

- `ggml-cuda.cpp` — CUDA 后端入口
- `mmq.cuh` — 量化矩阵乘法 kernel
- `fattn.cuh` — Flash Attention kernel
- `dequantize.cuh` — 反量化 kernel

---

ggml/src/ggml-metal/ — Metal 后端

关键文件：

- `ggml-metal.m` — Objective-C 后端实现
- `ggml-metal.metal` — Metal Compute Shaders

  Metal 后端利用 Apple Silicon 的统一内存：

```
    // Metal buffer 直接映射到 CPU 地址空间
    id<MTLBuffer> metal_buffer = [device newBufferWithBytesNoCopy:ptr
        length:size options:0 deallocator:nil];
```

关键函数索引

函数	文件	说明
`ggml_backend_cpu_init`	`ggml-cpu.cpp`	初始化 CPU 后端
`ggml_backend_cuda_init`	`ggml-cuda.cpp`	初始化 CUDA 后端
`ggml_backend_metal_init`	`ggml-metal.m`	初始化 Metal 后端
`ggml_backend_alloc_ctx_tensors`	`ggml-backend.cpp`	为张量分配设备内存
`ggml_backend_graph_compute`	各后端	在设备上执行计算图

GGML 后端与硬件抽象 — 概念

后端接口设计

GGML 定义了统一的后端接口 `ggml_backend`：

每个后端实现以下核心能力：

- **Buffer** 分配 — 在设备上分配/释放内存
- **Tensor** 操作 — 实现支持的操作集合
- 数据传输 — Host ↔ Device 数据拷贝
- 计算执行 — 执行计算图的节点

CPU 后端

CPU 后端是最基础的后端，支持所有 GGML 操作：

- **SIMD** 加速 — 利用 AVX2/AVX512 (x86) 或 NEON (ARM) 指令集
- 线程池 — `ggml-threading.cpp` 管理 OpenMP 或自定义线程池
- 量化运算 — 优化的量化 kernel（Q4_0 矩阵乘法等）

CUDA 后端

CUDA 后端利用 NVIDIA GPU 加速：

- 自定义 CUDA kernel 实现核心操作
- 支持 tensor parallel 多 GPU 推理
- 异步执行与流 (stream) 管理
- 支持 Flash Attention 等 GPU 优化

Metal 后端

Metal 后端针对 Apple Silicon 优化：

- 使用 Metal Compute Shaders
- 统一内存架构（CPU/GPU 共享内存）
- 通过 `ggml-metal.metal` 实现 shader
- Metal Performance Shaders (MPS) 加速

后端选择策略

Scheduler 自动将计算图节点分配到最优后端：

1. 检查每个操作的支持情况
2. 优先使用 GPU 后端
3. 不支持的操作 fallback 到 CPU
4. 跨后端操作自动插入数据传输

相关概念

- backend — 后端抽象详解
- tensor — 张量数据在设备间的传输
- quantization — 各后端的量化 kernel 实现

GGML 后端与硬件抽象 — 练习

练习 1：查看编译时可用的后端

阅读 CMakeLists.txt 中的后端选项，列出所有 `GGML_*` 开头的选项及其含义。

> 参考答案
>
> 主要后端选项：
>
> - `GGML_CUDA=ON` — NVIDIA GPU (CUDA)
> - `GGML_METAL=ON` — Apple GPU (Metal)
> - `GGML_VULKAN=ON` — 跨平台 GPU (Vulkan)
> - `GGML_OPENCL=ON` — OpenCL GPU
> - `GGML_BLAS=ON` — BLAS 加速 (OpenBLAS, MKL)
> - `GGML_HIP=ON` — AMD GPU (ROCm/HIP)
> - `GGML_SYCL=ON` — Intel GPU (SYCL)
> - `GGML_RPC=ON` — 远程过程调用后端

练习 2：追踪 buffer 分配流程

从 `llama_init_from_model` 开始，追踪 tensor 是如何分配到设备内存的。提示：关注 `ggml_backend_alloc_ctx_tensors` 的调用路径。

参考答案

流程：

1. `llama_init_from_model()` 创建 context
2. 内部调用 `llama_context::init()`
3. 使用 `ggml_backend_alloc_ctx_tensors(backend, ctx)` 将模型权重张量分配到后端 buffer
4. 后端根据自己的内存管理策略分配设备内存
5. CPU 后端使用 `malloc` / `mmap`
6. CUDA 后端使用 `cudaMalloc`
7. Metal 后端使用 `[MTLDevice newBufferWithBytes:]` 或映射共享内存

练习 3：对比 CPU 和 GPU kernel

在 `ggml-cpu/` 和 `ggml-cuda/` 或 `ggml-metal/` 中找到矩阵乘法的实现，对比它们的实现策略。

参考答案

CPU 矩阵乘法：

- 按行分块，利用 SIMD 指令并行计算
- 使用 cache-friendly 的分块策略
- 量化矩阵乘法使用查表 + 累加

CUDA 矩阵乘法 (mmq.cuh)：

- 每个 thread block 处理输出矩阵的一个 tile
- 利用 shared memory 缓存输入 tile
- warp-level 矩阵乘法指令
- 针对不同量化类型有专门 kernel

Metal 矩阵乘法 (ggml-metal.metal)：

- threadgroup 共享内存缓存
- SIMD group 矩阵操作
- 利用 Apple GPU 的统一内存特性减少拷贝

拓展挑战

- [ ] 阅读一个量化矩阵乘法 kernel（如 Q4_0），理解 block-level 反量化过程
- [ ] 对比同一模型在 CPU-only 和 GPU 后端上的推理速度
- [ ] 理解 Scheduler 如何处理跨后端的 tensor 依赖

GGML 后端与硬件抽象

> GGML 通过后端抽象层将计算图分发到不同硬件，实现 CPU、CUDA、Metal、Vulkan 等多种加速。

涵盖内容

章节	核心主题
概念	后端接口、缓冲区管理、任务调度
代码走读	ggml-backend.cpp、ggml-cpu/、ggml-metal/
练习	后端注册、缓冲区分配流程

核心概念

- backend — 硬件后端的统一抽象接口
- 缓冲区 **(Buffer)** — 后端管理的设备内存
- 任务调度 — 将计算图节点分配到对应后端执行

前置知识

- GGML 张量库基础
- 了解 CPU SIMD、GPU 编程基础概念
- 了解设备内存管理

学习路径

读完本主题后，你将理解：

- GGML 后端接口的设计模式
- CPU 后端如何利用 SIMD 指令加速
- CUDA/Metal 后端的 kernel 调度方式
- 多后端协同工作的机制

→ 下一步：模型加载与 GGUF 格式

模型加载与 GGUF 格式 — 代码走读

---

src/llama-model-loader.cpp — 模型加载器

这是模型加载的核心文件。

加载流程

```
llama_model_load_from_file()
  └── llama_model::load()
      ├── 1. 打开 GGUF 文件
      ├── 2. 解析 header & metadata
      ├── 3. 读取超参数 (hparams)
      ├── 4. 构建词表 (vocab)
      ├── 5. 分配 GGML context
      ├── 6. mmap 权重数据
      └── 7. 映射张量到模型结构
```

GGUF 解析

```
// 读取文件头
uint32_t magic = gguf_get_magic(ctx);
uint32_t version = gguf_get_version(ctx);
uint64_t n_tensors = gguf_get_n_tensors(ctx);
uint64_t n_kv = gguf_get_n_kv(ctx);

// 读取元数据
std::string arch = gguf_get_val_str(ctx, "general.architecture");
uint32_t n_layers = gguf_get_val_u32(ctx, "llama.block_count");
```

权重映射

模型加载器将 GGUF 张量名映射到内部结构：

```
// 遍历所有张量
for (int i = 0; i < n_tensors; i++) {
    const char * name = gguf_get_tensor_name(ctx, i);
    struct ggml_tensor * tensor = ggml_get_tensor(model_ctx, name);
    // 通过名称模式匹配分配到对应层
}
```

---

src/llama-arch.h — 架构定义

每种架构定义了其层结构和张量映射：

```
struct llm_layer {
    // Attention
    struct ggml_tensor * wq;
    struct ggml_tensor * wk;
    struct ggml_tensor * wv;
    struct ggml_tensor * wo;
    // FFN
    struct ggml_tensor * ffn_gate;
    struct ggml_tensor * ffn_up;
    struct ggml_tensor * ffn_down;
    // Norm
    struct ggml_tensor * attn_norm;
    struct ggml_tensor * ffn_norm;
};
```

---

gguf-py/ — Python GGUF 工具

提供 Python 接口来读写 GGUF 文件：

```
from gguf import GGUFReader

reader = GGUFReader("model.gguf")
for tensor in reader.tensors:
    print(f"{tensor.name}: {tensor.tensor_type}, shape={tensor.shape}")
```

关键函数索引

函数	文件	说明
`llama_model_load_from_file`	`llama-model.cpp`	API 入口
`llm_load_tensors`	`llama-model-loader.cpp`	加载所有张量
`llm_load_hparams`	`llama-model-loader.cpp`	解析超参数
`llm_load_vocab`	`llama-model-loader.cpp`	加载词表
`gguf_get_*`	`ggml/src/gguf.cpp`	GGUF 元数据读取

模型加载与 GGUF 格式 — 概念

GGUF 文件格式

GGUF (GGML Universal File) 是二进制格式，结构如下：

```
┌──────────────────────┐
│ Header               │  magic + version + tensor_count + metadata_count
├──────────────────────┤
│ Metadata KV Pairs    │  key-value 元数据（架构、超参数、词表等）
├──────────────────────┤
│ Tensor Info Array    │  每个张量的 name、dims、type、offset
├──────────────────────┤
│ Alignment Padding    │  对齐填充
├──────────────────────┤
│ Tensor Data          │  所有张量的实际数据
└──────────────────────┘
```

元数据 (Metadata)

GGUF 存储丰富的模型信息：

- `general.architecture` — 模型架构名（如 "llama", "gpt2"）
- `llama.context_length` — 最大上下文长度
- `llama.embedding_length` — embedding 维度
- `llama.block_count` — Transformer 层数
- `llama.attention.head_count` — 注意力头数
- `tokenizer.ggml.tokens` — 词表
- `tokenizer.ggml.scores` — token 分数

张量存储

每个张量记录：

- 名称（如 `blk.0.attn_q.weight`）
- 维度（n_dims）
- 数据类型（F16, Q4_0 等）

- 在文件中的偏移量

模型架构

`llama-arch.h` 定义了支持的模型架构枚举：

```
enum llm_arch {
    LLM_ARCH_LLAMA,
    LLM_ARCH_GPT2,
    LLM_ARCH_FALCON,
    LLM_ARCH_BAICHUAN,
    LLM_ARCH_STARCODER,
    LLM_ARCH_QWEN2,
    // ... 50+ 架构
};
```

每种架构定义了：

- 层结构（attention, FFN 的组成）
- 张量命名规则
- 特殊操作（如 RoPE 变体）

权重映射

模型加载时，将 GGUF 中的张量名映射到模型结构：

```
GGUF tensor name                    → 模型位置
blk.0.attn_q.weight                 → layers[0].attention.wq
blk.0.attn_k.weight                 → layers[0].attention.wk
blk.0.attn_v.weight                 → layers[0].attention.wv
blk.0.attn_output.weight            → layers[0].attention.wo
blk.0.ffn_gate.weight               → layers[0].ffn.w1
blk.0.ffn_up.weight                 → layers[0].ffn.w3
blk.0.ffn_down.weight               → layers[0].ffn.w2
```

内存映射 **(mmap)**

llama.cpp 使用 mmap 加载大模型：

- 不将整个文件读入内存
- 按需映射权重页到地址空间
- 操作系统自动管理物理内存
- 允许加载超过物理内存的模型

相关概念

- gguf — GGUF 格式详解
- tensor — 张量存储与类型
- quantization — 权重量化

模型加载与 GGUF 格式 — 练习

练习 1：用 Python 查看 GGUF 元数据

使用 `gguf-py` 库读取一个 GGUF 模型文件，列出所有元数据和张量名称。

参考答案

```
pip install gguf
python -c "
from gguf import GGUFReader
reader = GGUFReader('model.gguf')
# 打印元数据
for key, val in reader.fields.items():
    print(f'{key}: {val}')
# 打印张量
for tensor in reader.tensors:
    print(f'{tensor.name}: type={tensor.tensor_type}, shape={tensor.shap
e}')
"
```

练习 2：追踪模型加载流程

在 `llama-model-loader.cpp` 中，从 `llama_model_load_from_file` 开始，梳理完整的加载调用链。

参考答案

调用链：

1. `llama_model_load_from_file(path, params)`
2. → `llama_model::load(model_loader, params)`
3. → `llm_load_hparams(loader)` — 解析架构和超参数
4. → `llm_load_vocab(loader)` — 加载词表
5. → `llm_load_tensors(loader, progress_cb)` — 加载权重
6. → 对每个张量：检查类型、创建 ggml_tensor、设置 mmap 映射
7. → 根据架构创建 `llm_layer` 结构并关联张量

练习 3：添加新架构支持

阅读 `docs/development/HOWTO-add-model.md`，理解添加新模型架构需要的步骤。

参考答案

添加新架构的步骤（摘自 HOWTO）：

1. 在 `llama-arch.h` 的 `llm_arch` 枚举中添加新架构
2. 在 `llama-arch.cpp` 中注册架构的层定义和张量映射
3. 在 `llm_load_hparams` 中添加超参数解析
4. 在 `llama-model.cpp` 中实现 `llm_build_*` 函数（forward pass）
5. 在 `llama-model.cpp` 的 switch 中添加架构分发
6. 编写转换脚本将 HuggingFace 权重转为 GGUF
7. 添加测试用例

拓展挑战

- [ ] 使用 `gguf-py` 手动构建一个最小的 GGUF 文件
- [ ] 对比同一模型在 F16 和 Q4_0 下的 GGUF 文件大小
- [ ] 阅读 `convert_hf_to_gguf.py` 理解权重转换过程

模型加载与 GGUF 格式

> GGUF 是 llama.cpp 使用的统一模型格式，将权重、词表和元数据打包在单一文件中。

涵盖内容

章节	核心主题
概念	GGUF 格式、模型架构、权重映射
代码走读	llama-model-loader.cpp、llama-arch.h
练习	解析 GGUF 文件、理解模型架构注册

核心概念

- gguf — GGML Universal File，统一的模型容器格式
- 模型架构 — 不同 LLM 架构（LLaMA, GPT-NeoX, Falcon 等）的统一加载
- 权重映射 — 将 GGUF 中的张量名映射到模型计算图的对应位置

前置知识

- GGML 张量库基础
- GGML 后端与硬件抽象
- 了解 Transformer 模型的基本结构

学习路径

读完本主题后，你将理解：

- GGUF 文件格式的二进制布局
- 模型加载的完整流程（文件读取 → 张量分配 → 权重映射）
- 如何添加对新模型架构的支持
- 量化权重在加载时的处理方式

→ 下一步：分词与词表

分词与词表 — 代码走读

---

src/llama-vocab.cpp — 分词器实现

核心类结构

`llama_vocab` 类管理整个词表和分词逻辑：

```
struct llama_vocab {
    // 词表数据
    std::vector<llama_vocab_token> tokens;
    std::unordered_map<std::string, llama_token> token_to_id;

    // 分词类型
    enum llama_vocab_type type;

    // 特殊 token ID
    llama_token bos_token_id;
    llama_token eos_token_id;
    llama_token pad_token_id;

    // BPE 合并规则
    std::vector<std::pair<std::string, std::string>> merges;
};
```

分词入口

```
int32_t llama_tokenize(
    const struct llama_model * model,
    const char * text,
    llama_token * tokens,
    int32_t n_max_tokens,
    bool add_bos,
    bool special);
```

BPE 编码流程

```
// 1. 预处理：Unicode 规范化
text = unicode_normalize(text);

// 2. 预分词：按正则分割为词
auto words = pre_tokenize(text);

// 3. 对每个词执行 BPE
for (auto & word : words) {
    auto tokens = bpe_encode(word);
    result.insert(result.end(), tokens.begin(), tokens.end());
}

// 4. 添加特殊 token
if (add_bos) result.insert(result.begin(), vocab.bos_token_id);
```

BPE 合并

```
// BPE 核心：重复合并最高优先级的 token 对
while (true) {
    auto best = find_best_merge(tokens, merges);
    if (best.score == -INF) break;
    tokens = merge_pair(tokens, best.pair);
}
```

Detokenization

```
// 将 token ID 转回文本
int32_t llama_token_to_piece(
    const struct llama_model * model,
    llama_token token,
    char * buf,
    int32_t length,
    int32_t lstrip,
    bool special);
```

关键函数索引

函数	说明
`llama_tokenize`	文本 → token IDs
`llama_token_to_piece`	token ID → 文本片段
`llama_vocab_get_text`	获取 token 的文本表示
`llama_vocab_get_type`	获取分词器类型
`llm_load_vocab`	从 GGUF 加载词表

分词与词表 — 概念

分词算法

llama.cpp 支持以下分词算法：

BPE (Byte Pair Encoding)

GPT 系列模型使用的算法：

1. 从字符级开始
2. 反复合并最高频的 token 对
3. 编码时使用最长匹配

```
    "hello world" → ["he", "llo", " world"]
```

SPM (SentencePiece)

LLaMA 系列使用，基于 byte-level BPE + byte fallback：

- 预留 256 个 byte token 作为 fallback
- 空格替换为 `▁` (U+2581)
- 支持添加 BOS token

WPM (WordPiece)

BERT 系列使用：

- 类似 BPE 但使用最长优先匹配
- 未知词用 `##` 前缀标记子词

Unigram

T5 系列使用：

- 从大词表中逐步删减
- 编码时使用概率最大的路径

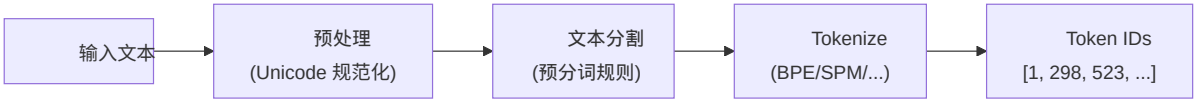
词表结构

```
enum llama_vocab_type {
    LLAMA_VOCAB_TYPE_NONE = 0,
    LLAMA_VOCAB_TYPE_SPM  = 1,  // SentencePiece
    LLAMA_VOCAB_TYPE_BPE  = 2,  // Byte Pair Encoding
    LLAMA_VOCAB_TYPE_WPM  = 3,  // WordPiece
    LLAMA_VOCAB_TYPE_UGM  = 4,  // Unigram
    LLAMA_VOCAB_TYPE_RWKV = 5,  // RWKV greedy
};
```

每个 token 包含：

- 文本表示（text）
- 分数（score）— 用于 BPE 合并优先级
- 类型（normal, control, unknown, byte 等）
- 特殊标记（BOS, EOS, PAD, EOT 等）

分词流程



相关概念

- tokenization — 分词算法详解
- gguf — 词表在 GGUF 中的存储

分词与词表 — 练习

练习 1：使用 llama-cli 测试分词

使用 `llama-cli --verbose-prompt` 查看一段中文文本的分词结果，分析哪些 token 是完整的汉字，哪些被拆分。

参考答案

运行命令：

```
./llama-cli -m model.gguf -p "你好世界" --verbose-prompt
```

输出会显示 token 序列和对应的文本。对于 LLaMA 模型，中文通常被编码为多个 byte-fallback token，因为 SPM 词表中中文字符覆盖有限。

练习 2：追踪 BPE 编码过程

在 `llama-vocab.cpp` 中找到 BPE 编码的实现，画出从输入 `"hello"` 到 token IDs 的完整流程。

参考答案

流程：

1. `llama_tokenize()` 调用内部 `llama_vocab::tokenize()`
2. 文本 `"hello"` 经过预分词（GPT-2 风格正则分割）
3. 对 `"hello"` 执行 BPE 编码：
   - 初始: `[h, e, l, l, o]`
   - 合并 `ll` → `["h", "e", "ll", "o"]`
   - 查找词表得到对应的 token IDs
4. 返回 `[token_id_h, token_id_e, token_id_ll, token_id_o]`

练习 3：比较不同模型的词表

下载两个不同架构的 GGUF 模型，用 `gguf-py` 分别查看它们的词表大小和类型。

参考答案

```
from gguf import GGUFReader

for model_path in ["llama.gguf", "gpt2.gguf"]:
    reader = GGUFReader(model_path)
    vocab_type = reader.fields.get("tokenizer.ggml.model")
    n_tokens = reader.fields.get("tokenizer.ggml.n_tokens")
    print(f"{model_path}: type={vocab_type}, n_tokens={n_tokens}")
```

典型结果：

- LLaMA 2: SPM, 32000 tokens
- GPT-2: BPE, 50257 tokens
- Qwen2: BPE, 151936 tokens

拓展挑战

- [ ] 阅读 Unicode 处理代码（`unicode.cpp`），理解 byte-fallback 机制
- [ ] 实现一个简单的 BPE 分词器（Python），与 llama.cpp 的结果对比
- [ ] 分析 chat template 如何影响分词结果

分词与词表

> llama.cpp 支持多种分词算法（BPE、SPM、WPM、Unigram），通过 `llama-vocab.cpp` 统一实现。

涵盖内容

章节	核心主题
概念	分词算法、词表结构、特殊 token
代码走读	llama-vocab.cpp 实现
练习	分词流程追踪、词表分析

核心概念

- tokenization — 将文本拆分为 token 序列
- 词表 — 模型支持的 token 集合及对应的嵌入向量
- **BPE / SPM / WPM** — 不同的子词分词算法

前置知识

- 模型加载与 GGUF 格式
- Unicode 与 UTF-8 编码基础
- 了解 BPE 分词原理

学习路径

读完本主题后，你将理解：

- llama.cpp 如何在 C++ 中实现多种分词算法
- 词表在 GGUF 中的存储与加载方式
- 特殊 token（BOS、EOS、PAD）的处理
- 分词与反分词（detokenization）的流程

→ 下一步：Transformer 推理图

Transformer 推理图 — 代码走读

---

src/llama-model.cpp — Forward Pass

`llama-model.cpp` 是 llama.cpp 最大的文件，包含所有模型架构的 forward pass 实现。

计算图构建入口

```
// 每次推理调用
struct ggml_cgraph * llm_build_graph(llama_context & lctx, const llama_batch & b
atch) {
    // 创建新计算图
    struct ggml_cgraph * graph = ggml_new_graph(ctx);

    // 根据架构分发
    switch (model.arch) {
        case LLM_ARCH_LLAMA:
            result = llm_build_llama(lctx, batch);
            break;
        case LLM_ARCH_GPT2:
            result = llm_build_gpt2(lctx, batch);
            break;
        // ... 其他架构
    }

    return graph;
}
```

LLaMA Forward Pass

```
struct ggml_tensor * llm_build_llama(llama_context & lctx, const llama_batch & b
atch) {
    // 1. Token embedding
    struct ggml_tensor * cur = ggml_get_rows(ctx, model.tok_embd, tokens);

    // 2. 逐层处理
    for (int il = 0; il < n_layer; il++) {
        // 2a. Attention
        cur = llm_build_norm(ctx, cur, layers[il].attn_norm);
        struct ggml_tensor * Q = ggml_mul_mat(ctx, layers[il].wq, cur);
        struct ggml_tensor * K = ggml_mul_mat(ctx, layers[il].wk, cur);
        struct ggml_tensor * V = ggml_mul_mat(ctx, layers[il].wv, cur);

        // 2b. RoPE
        Q = ggml_rope(ctx, Q, positions, n_rot, rope_type);
        K = ggml_rope(ctx, K, positions, n_rot, rope_type);

        // 2c. Attention + KV Cache
        // (将 K, V 存入 cache)
        cur = llm_build_attn(ctx, Q, K_cache, V_cache, mask);

        // 2d. Output projection + 残差
        cur = ggml_mul_mat(ctx, layers[il].wo, cur);
        cur = ggml_add(ctx, cur, ffn_inp);

        // 2e. FFN (SwiGLU)
        ffn_inp = cur;
        cur = llm_build_norm(ctx, cur, layers[il].ffn_norm);
        cur = llm_build_ffn(ctx, cur, layers[il]);

        // 2f. 残差
        cur = ggml_add(ctx, cur, ffn_inp);
    }

    // 3. Final norm
    cur = llm_build_norm(ctx, cur, model.output_norm);

    // 4. Output projection
    cur = ggml_mul_mat(ctx, model.output, cur);

    return cur;
}
```

src/llama-graph.cpp — 图构建辅助

提供计算图构建的辅助函数：

```
// 构建归一化层 (RMSNorm)
struct ggml_tensor * llm_build_norm(
    struct ggml_context * ctx,
    struct ggml_tensor * cur,
    struct ggml_tensor * weight) {
    cur = ggml_rms_norm(ctx, cur, epsilon);
    cur = ggml_mul(ctx, cur, weight);
    return cur;
}
```

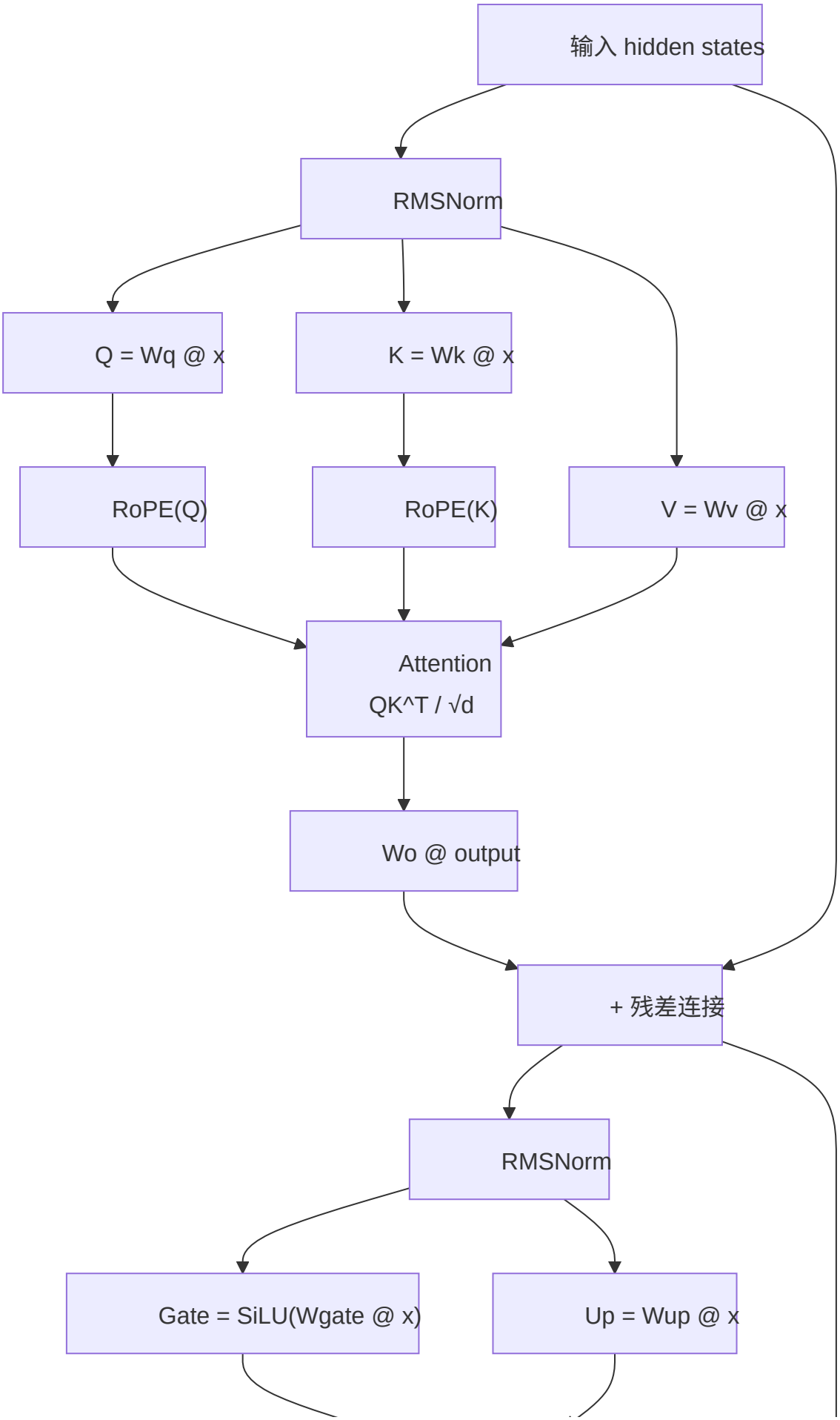
关键函数索引

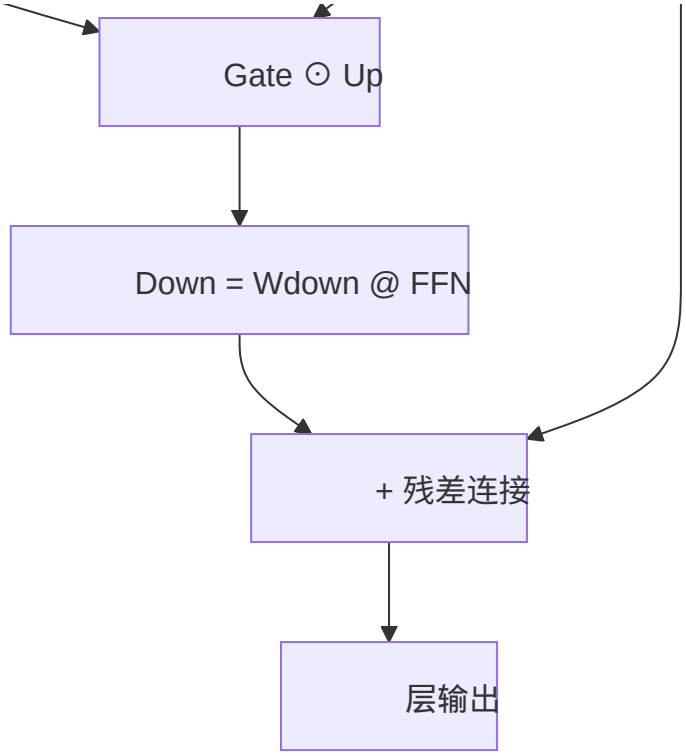
函数	文件	说明
`llm_build_graph`	`llama-model.cpp`	构建完整推理图
`llm_build_llama`	`llama-model.cpp`	LLaMA 架构 forward pass
`llm_build_attn`	`llama-graph.cpp`	注意力计算
`llm_build_ffn`	`llama-graph.cpp`	FFN (SwiGLU)
`llm_build_norm`	`llama-graph.cpp`	RMSNorm
`ggml_rope`	`ggml.c`	RoPE 位置编码

Transformer 推理图 — 概念

Transformer 层结构

标准 LLaMA 层的计算图：


输入 hidden states
RMSNorm
Q = Wq @ x
K = Wk @ x
RoPE(Q)
RoPE(K)
V = Wv @ x
Attention
QK^T / √d
Wo @ output
+ 残差连接
RMSNorm
Gate = SiLU(Wgate @ x)
Up = Wup @ x



RoPE (Rotary Position Embedding)

rope 将位置信息编码到 Q 和 K 中：

```
对于位置 pos 和维度 d：
cos(θ) = cos(pos / 10000^(2d/dim))
sin(θ) = sin(pos / 10000^(2d/dim))

RoPE(x, pos) = [x_even * cos(θ) - x_odd * sin(θ),
                x_even * sin(θ) + x_odd * cos(θ)]
```

变体：

- **LLaMA RoPE** — 标准 RoPE
- **RoPE Neox** — 调整频率基准
- **mRoPE** — 多维 RoPE（用于多模态）
- **LongRoPE** — 支持更长上下文

注意力计算

```
// Scaled Dot-Product Attention
QK = ggml_mul_mat(ctx, K, Q)           // Q @ K^T
QK = ggml_scale(ctx, QK, 1/sqrt(d_k))  // 缩放
QK = ggml_add(ctx, QK, mask)            // 因果 mask
S  = ggml_soft_max(ctx, QK)             // softmax
O  = ggml_mul_mat(ctx, V, S)            // S @ V
```

SwiGLU FFN

LLaMA 系列使用 SwiGLU 激活：

```
// SwiGLU(x) = (SiLU(x @ W_gate) ⊙ (x @ W_up)) @ W_down
gate = ggml_mul_mat(ctx, w_gate, x);
up   = ggml_mul_mat(ctx, w_up, x);
gate = ggml_silu(ctx, gate);      // SiLU = x * sigmoid(x)
ffn  = ggml_mul(ctx, gate, up);
out  = ggml_mul_mat(ctx, w_down, ffn);
```

相关概念

- rope — 旋转位置编码详解
- compute-graph — 计算图的构建与执行
- kv-cache — KV Cache 如何加速注意力计算

Transformer 推理图 — 练习

练习 1：绘制单层计算图

阅读 `llm_build_llama` 函数，为单层 Transformer 绘制完整的计算图（从输入到输出），标注每个 GGML 操作。

参考答案

单层计算图（简化）：

```
input → rms_norm → mul_mat(Wq) → rope → ─┐
          ↓                              │
        mul_mat(Wk) → rope → store_K ──────┤
          ↓                              │
        mul_mat(Wv) → store_V ─────────────┤
                                         │
        attention(Q, K_cache, V_cache) ←─────┘
          ↓
        mul_mat(Wo) → add(residual) ─┐
                                     │
        rms_norm → mul_mat(Wgate) → silu → mul → mul_mat(Wdown) → add(residu
al)
                   mul_mat(Wup)  ──┘
```

练习 2：理解 RoPE 的维度变换

RoPE 作用于 Q 和 K 的 head 维度上。对于一个 hidden_dim=4096, n_heads=32 的模型，计算每个 head 的维度和 RoPE 作用的部分。

参考答案

- head_dim = hidden_dim / n_heads = 4096 / 32 = 128
- RoPE 作用于 head_dim 的一半：n_rot = 128 / 2 = 64 对旋转
- 对于 GQA 模型（如 n_kv_heads < n_heads），K/V 的 head_dim 相同但 head 数不同
- RoPE 只影响 Q 和 K，不影响 V

练习 3：对比不同架构的 FFN

在 `llama-model.cpp` 中找到至少两种不同的 FFN 实现（如 LLaMA 的 SwiGLU 和 GPT-2 的 GeLU），对比差异。

参考答案

LLaMA (SwiGLU):

```
gate = ggml_mul_mat(ctx, layers[il].ffn_gate, cur);
up   = ggml_mul_mat(ctx, layers[il].ffn_up, cur);
gate = ggml_silu(ctx, gate);
cur  = ggml_mul(ctx, gate, up);
cur  = ggml_mul_mat(ctx, layers[il].ffn_down, cur);
```

GPT-2 (GeLU):

```
cur = ggml_mul_mat(ctx, layers[il].ffn_up, cur);
cur = ggml_gelu(ctx, cur);
cur = ggml_mul_mat(ctx, layers[il].ffn_down, cur);
```

差异：SwiGLU 使用两个投影（gate + up）做门控，GeLU 使用单个投影 + 激活。

拓展挑战

- [ ] 阅读 `ggml_rope` 的实现，理解旋转矩阵的构造方式
- [ ] 对比 LLaMA 和 Mistral 架构在代码中的差异
- [ ] 理解 GQA (Grouped Query Attention) 如何减少 KV Cache 大小

Transformer 推理图

> llama.cpp 通过 GGML 计算图构建 Transformer 的 forward pass，支持 50+ 种模型架构。

涵盖内容

章节	核心主题
概念	Transformer 层结构、RoPE、注意力机制
代码走读	llama-model.cpp 的 forward pass
练习	计算图构建、RoPE 位置编码

核心概念

- 计算图构建 — 每次推理动态构建 GGML 计算图
- rope — Rotary Position Embedding，旋转位置编码
- 注意力 — Scaled Dot-Product Attention + KV Cache
- **FFN** — 前馈网络（SwiGLU / GeLU）

前置知识

- GGML 张量库基础
- 模型加载与 GGUF 格式
- Transformer 架构基础

学习路径

读完本主题后，你将理解：

- llama.cpp 如何用 GGML 操作构建完整的 Transformer forward pass
- RoPE 位置编码的计算方式
- 注意力计算中的优化技巧
- 不同模型架构在代码中的差异

→ 下一步：KV Cache 与批处理

KV Cache 与批处理 — 代码走读

---

src/llama-memory.cpp — KV Cache 管理

核心数据结构

```
struct llama_kv_cache {
    // Cache 存储
    struct ggml_tensor * k_l;  // [n_kv_max, n_embd_k_gqa, n_layer]
    struct ggml_tensor * v_l;  // [n_kv_max, n_embd_v_gqa, n_layer]

    // 单元管理
    std::vector<struct llama_kv_cell> cells;
    uint32_t head;    // 最旧未使用位置
    uint32_t size;    // 已使用位置数
    uint32_t used;    // 活跃位置数

    // 序列追踪
    // 每个 cell 记录属于哪个序列、哪个位置
};
```

Cache 写入

```
// 在 decode 过程中，将新的 K/V 存入 cache
void llama_kv_cache_update(llama_context & lctx) {
    auto & kv = lctx.kv_self;
    for (int il = 0; il < n_layer; il++) {
        // 将新计算的 K 复制到 cache 对应位置
        ggml_backend_tensor_set(kv.k_l[il], k_data,
            offset, k_size);
        // 将新计算的 V 复制到 cache 对应位置
        ggml_backend_tensor_set(kv.v_l[il], v_data,
            offset, v_size);
    }
}
```

Cache 查找与复用

```
// 查找可以复用的 cache 前缀
// 例如 prompt "Hello world" 的 cache 可以在 "Hello world!" 中复用
int32_t llama_kv_cache_find_prefix(
    const llama_kv_cache & kv,
    const llama_pos * pos,
    int32_t n_tokens);
```

---

src/llama-batch.cpp — 批处理编码

```
struct llama_batch llama_batch_get_one(llama_token * tokens, int32_t n_tokens) {
    struct llama_batch batch = {
        .token    = tokens,
        .pos      = positions,   // 连续位置
        .n_seq_id = seq_count,   // 每个token的序列数
        .seq_id   = seq_ids,     // 序列ID
        .n_tokens = n_tokens,
    };
    return batch;
}

// 批量构建：同时处理多个序列
struct llama_batch llama_batch_get_token(llama_token token) {
    // 单 token batch，用于 decode
}
```

---

src/llama-context.cpp — 推理上下文

```
// 核心推理函数
int32_t llama_decode(struct llama_context * ctx, struct llama_batch batch) {
    // 1. 构建计算图
    auto * graph = llm_build_graph(*ctx, batch);

    // 2. 分配后端资源
    ggml_backend_alloc_graph(backend, graph);

    // 3. 执行计算
    ggml_backend_graph_compute(backend, graph);

    // 4. 获取输出 logits
    // logits 现在可用于采样
}
```

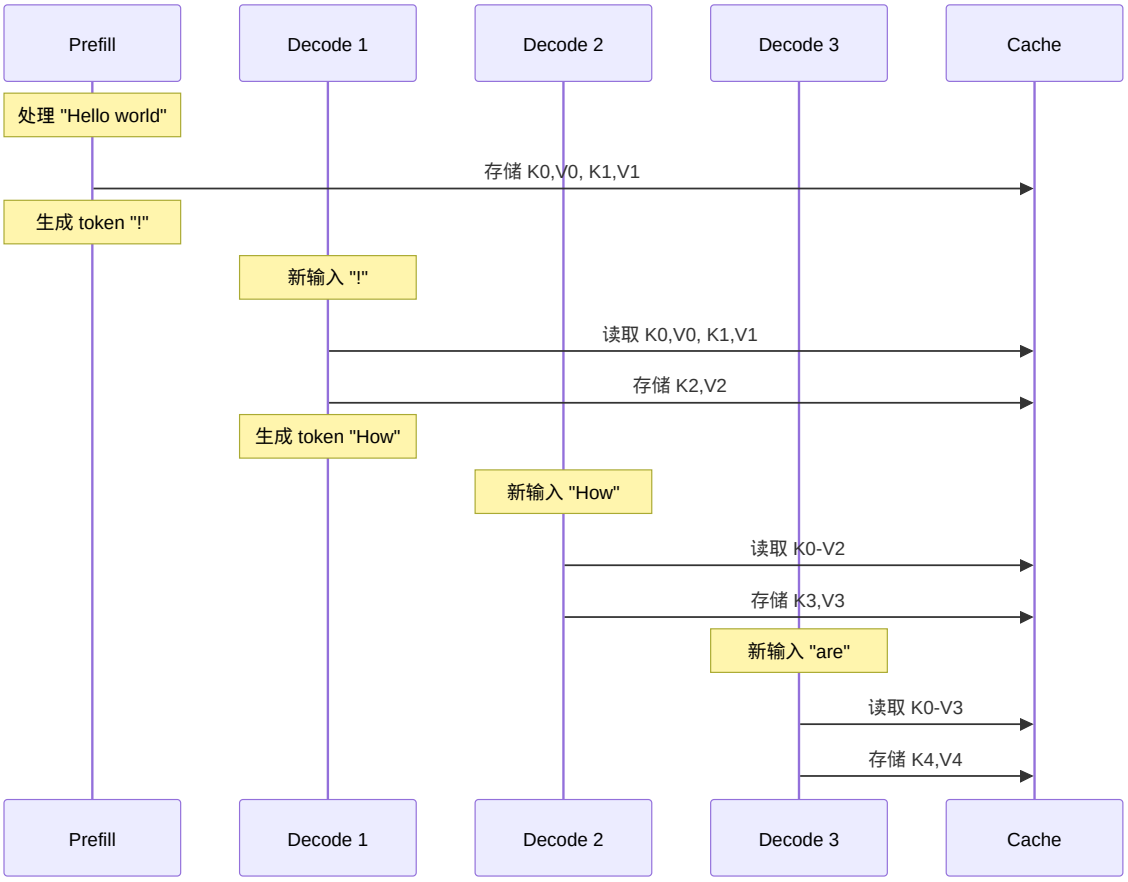
关键函数索引

函数	文件	说明
`llama_decode`	`llama-context.cpp`	执行一次 batch decode
`llama_kv_cache_update`	`llama-memory.cpp`	更新 KV Cache
`llama_kv_cache_clear`	`llama-memory.cpp`	清空 cache
`llama_batch_get_one`	`llama-batch.cpp`	构建单序列 batch
`llama_kv_cache_seq_rm`	`llama-memory.cpp`	删除指定序列的 cache

KV Cache 与批处理 — 概念

KV Cache 原理

Transformer 自回归生成时，每个新 token 只依赖于之前所有 token 的 K/V：



关键优势：每个 decode step 只需计算 1 个 token 的 Q/K/V，之前的 K/V 从 cache 读取。

内存布局

KV Cache 在内存中按层组织：

```
Layer 0: [K_cache(n_kv_max, n_embd), V_cache(n_kv_max, n_embd)]
Layer 1: [K_cache(n_kv_max, n_embd), V_cache(n_kv_max, n_embd)]
...
Layer N: [K_cache(n_kv_max, n_embd), V_cache(n_kv_max, n_embd)]
```

- `n_kv_max` = 最大 cache 容量（通常等于上下文长度）
- `n_embd` = KV 的 head 维度 × KV head 数

Batch 处理

llama.cpp 的 batch 允许同时处理多个 token（来自同一或不同序列）：

```
struct llama_batch {
    llama_token * token;      // token IDs
    int32_t     * pos;        // 每个token的位置
    int32_t     * n_seq_id;   // 每个token所属序列数
    llama_seq_id ** seq_id;   // 每个token的序列ID列表
    int32_t       n_tokens;   // 总token数
};
```

Prefill vs Decode

特性	Prefill	Decode
每次 token 数	N（整个 prompt）	1
计算类型	矩阵 × 矩阵	矩阵 × 向量
并行度	高	低
用途	处理输入 prompt	逐个生成 token

Cache 淘汰

当 cache 满时，策略包括：

- **Rolling** — 保留最近的 token，淘汰最旧的
- **Session** — 保存/恢复 cache 状态
- **Swa** — Sliding Window Attention，只缓存窗口内的 token

相关概念

- kv-cache — KV Cache 实现细节
- batch-decode — 批量解码机制
- backend — 不同后端的 cache 实现

KV Cache 与批处理 — 练习

练习 1：计算 KV Cache 的内存占用

对于一个 LLaMA-7B 模型（32 层, 32 heads, 4096 hidden, F16）和上下文长度 4096，计算 KV Cache 需要多少内存。

<details>
<summary>参考答案</summary>

每层 KV Cache 大小：

- K: 4096 positions × (4096 hidden / 32 heads × 32 heads) × 2 bytes (F16) = 4096 × 4096 × 2 = 33.5 MB
- V: 同样 33.5 MB
- 每层合计: 67 MB
- 32 层合计: 32 × 67 = 2,144 MB ≈ 2.1 GB

使用 GQA (如 8 KV heads)：33.5 × 2 × (8/32) × 32 = 536 MB

</details>

练习 2：理解 Batch 的序列管理

阅读 `llama-batch.cpp`，理解如何构建一个包含多个序列的 batch（parallel decoding 场景）。

参考答案

多序列 batch 的构建：

```
// 序列 0: 生成 token A
// 序列 1: 生成 token B
struct llama_batch batch = llama_batch_init(2);
batch.token[0] = token_A;
batch.pos[0] = seq0_pos;
batch.seq_id[0] = {0};  // 属于序列 0
batch.token[1] = token_B;
batch.pos[1] = seq1_pos;
batch.seq_id[1] = {1};  // 属于序列 1
batch.n_tokens = 2;
```

两个 token 会在一次 `llama_decode()` 中并行计算，各自更新对应序列的 KV Cache。

练习 3：Cache 前缀复用

分析 llama.cpp 如何在连续对话中复用 KV Cache。当一个已有 100 个 token cache 的对话继续生成时，cache 如何更新？

参考答案

连续对话的 cache 复用：

1. 第一轮对话后，KV Cache 包含 100 个位置的数据
2. 第二轮输入新的 prompt + 生成的回复（假设 50 个 token）
3. 首先检查前缀匹配：前 100 个 token 可能已经 cache
4. 新 token 从 position 100 开始追加
5. 只需要对新 token 执行 decode，已有的 K/V 从 cache 读取
6. 这就是 "prefix caching" 优化

相关 API:

- `llama_kv_cache_seq_keep()` — 保留指定序列的 cache
- `llama_kv_cache_seq_shift()` — 移动 cache 位置

拓展挑战

- [ ] 实现一个简单的多轮对话程序，验证 KV Cache 复用
- [ ] 对比不同上下文长度下的内存占用
- [ ] 阅读 Flash Attention 在 CUDA 后端中的实现

KV Cache 与批处理

> KV Cache 是 Transformer 推理的核心优化，避免重复计算已处理 token 的 K/V 向量。

涵盖内容

章节	核心主题
概念	KV Cache 原理、批处理、多序列管理
代码走读	llama-memory.cpp, llama-batch.cpp
练习	Cache 管理策略、批处理构建

核心概念

- kv-cache — 缓存已计算的 Key/Value 向量
- batch-decode — 批量并行解码多个 token
- 序列管理 — 多对话并发时的 cache 分配与复用

前置知识

- Transformer 推理图
- 了解自回归生成的原理
- 内存管理基础

学习路径

读完本主题后，你将理解：

- KV Cache 的内存布局与数据结构
- Prompt processing (prefill) vs. token generation (decode) 的区别
- 批处理如何实现多序列并行推理
- Cache 淘汰策略（如 rolling cache）

→ 下一步：采样、量化与部署

采样、量化与部署 — 代码走读

---

src/llama-sampler.cpp — 采样器实现

采样器链架构

```
// 创建采样器链
struct llama_sampler * chain = llama_sampler_chain_init(
    llama_sampler_chain_default_params());

// 添加采样器（顺序很重要！）
llama_sampler_chain_add(chain, llama_sampler_init_temp(0.8f));
llama_sampler_chain_add(chain, llama_sampler_init_top_k(40));
llama_sampler_chain_add(chain, llama_sampler_init_top_p(0.95f, 1));
llama_sampler_chain_add(chain, llama_sampler_init_dist(LLAMA_DEFAULT_SEED));

// 采样
llama_token token = llama_sampler_sample(chain, ctx, -1);
```

采样器接口

每个采样器实现统一接口：

```
struct llama_sampler_i {
    const char *       (*name)  (const llama_sampler * smpl);
    void               (*accept)(llama_sampler * smpl, llama_token token);
    llama_token        (*apply) (llama_sampler * smpl, llama_context * ctx, int3
2_t idx);
    void               (*reset) (llama_sampler * smpl);
    void               (*free)  (llama_sampler * smpl);
    struct llama_sampler * (*clone)(const llama_sampler * smpl);
};
```

Top-P 实现

```
// Top-P: 保留累积概率 <= p 的 token
static llama_token llama_sampler_top_p_apply(
    llama_sampler * smpl, llama_context * ctx, int32_t idx) {
    auto * cur_p = llama_sampler_get_candidates(ctx, idx);
    // 按 logit 排序
    // 计算累积 softmax 概率
    // 截断到 top-p
    // 从截断后的候选中采样
}
```

---

tools/quantize/ — 量化工具

量化入口：

```
// quantize 工具核心
int main(int argc, char ** argv) {
    // 1. 加载源模型
    // 2. 按类型量化每个张量
    // 3. 写入新的 GGUF 文件
}
```

量化过程：

```
// 对每个张量执行量化
for (auto & tensor : model.tensors) {
    switch (quant_type) {
        case GGML_TYPE_Q4_0:
            ggml_quantize_q4_0(src_data, dst_data, n_elements);
            break;
        case GGML_TYPE_Q5_K:
            ggml_quantize_q5_K(src_data, dst_data, n_elements);
            break;
        // ...
    }
}
```

tools/server/ — HTTP 服务器

```
// server 核心循环
int main(int argc, char ** argv) {
    // 1. 解析参数、加载模型
    // 2. 创建 HTTP 服务器 (httplib)
    // 3. 注册 API 路由

    svr.Post("/v1/chat/completions", handle_chat_completion);
    svr.Post("/v1/completions", handle_completion);
    svr.Post("/v1/embeddings", handle_embeddings);

    // 4. 启动事件循环
    svr.listen("0.0.0.0", port);
}
```

任务队列

```
// server 使用任务队列处理并发请求
struct server_task {
    int id;
    llama_token prompt;
    json params;  // temperature, top_p, etc.
};

// 多个请求可以合并到一个 batch 中并行处理
```

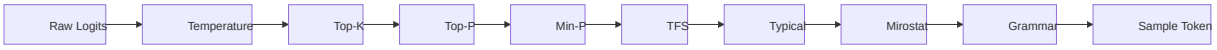
关键函数索引

函数	文件	说明
`llama_sampler_chain_init`	`llama-sampler.cpp`	创建采样器链
`llama_sampler_chain_add`	`llama-sampler.cpp`	添加采样器
`llama_sampler_sample`	`llama-sampler.cpp`	从 logits 采样 token
`llama_sampler_init_top_k`	`llama-sampler.cpp`	创建 Top-K 采样器
`llama_sampler_init_top_p`	`llama-sampler.cpp`	创建 Top-P 采样器
`llama_sampler_init_temp`	`llama-sampler.cpp`	创建温度采样器

采样、量化与部署 — 概念

采样器链 (Sampler Chain)

sampler-chain 采样器链是一个管道，logits 依次经过多个采样器处理：



常见采样器

采样器	作用
Temperature	缩放 logits 温度
Top-K	只保留概率最高的 K 个
Top-P	只保留累积概率前 P 的 token
Min-P	过滤概率低于最大概率 × min_p 的 token
Typical	基于信息熵的采样
Mirostat	自适应温度控制
Grammar	基于语法约束输出格式
Penalties	重复惩罚、频率惩罚
Logit Bias	手动调整特定 token 的概率

量化 (Quantization)

quantization 将模型权重从 F16 压缩为低比特整数：

量化类型对比

类型	比特/权重	模型大小 (7B)	质量损失
F16	16	13.5 GB	基准
Q8_0	8.5	7.2 GB	极小
Q5_1	5.5	4.7 GB	小
Q5_0	5.0	4.3 GB	小
Q4_1	4.5	3.9 GB	中等
Q4_0	4.5	3.8 GB	中等
IQ4_XS	4.25	3.6 GB	中等
Q3_K_S	3.5	3.0 GB	较大
Q2_K	2.75	2.4 GB	大

Block Quantization

量化以 block 为单位（通常 32 个权重）：

```
Block Q4_0 (18 bytes for 32 weights):
┌─────────┬──────────────────────────┐
│ d (F16) │ 16 × uint8 (4-bit × 32)  │
│ scale   │ quantized values         │
└─────────┴──────────────────────────┘

dequantize: weight = (quantized - 8) × d
```

Importance Matrix (imatrix)

imatrix 通过在校准数据上统计各张量的重要性，指导量化时保留关键权重：

```
# 生成 imatrix
./llama-imatrix -m model.gguf -f calibration.txt -o imatrix.dat

# 使用 imatrix 量化
./llama-quantize --imatrix imatrix.dat model.gguf model-Q4_K_M.gguf Q4_K_M
```

llama-server

`llama-server` 提供 OpenAI 兼容 API：

```
# 启动服务器
./llama-server -m model.gguf --port 8080

# 调用 chat completion
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{"model":"model","messages":[{"role":"user","content":"Hello"}]}'
```

支持的端点：

- `/v1/chat/completions` — Chat Completion
- `/v1/completions` — Text Completion
- `/v1/embeddings` — Embeddings
- `/v1/models` — 模型列表
- `/health` — 健康检查

相关概念

- sampler-chain — 采样器链设计
- quantization — 量化算法
- imatrix — 重要性矩阵

采样、量化与部署 — 练习

练习 **1**：量化模型对比

将同一个模型分别量化为 Q4_0、Q5_1、Q8_0，对比文件大小和推理速度。

参考答案

```
# 量化
./llama-quantize model-F16.gguf model-Q4_0.gguf Q4_0
./llama-quantize model-F16.gguf model-Q5_1.gguf Q5_1
./llama-quantize model-F16.gguf model-Q8_0.gguf Q8_0

# 性能测试
./llama-bench -m model-Q4_0.gguf
./llama-bench -m model-Q5_1.gguf
./llama-bench -m model-Q8_0.gguf
```

预期观察：

- Q4_0 最快、最小，但质量损失最大
- Q8_0 接近 F16 质量，但体积约为 F16 的一半
- 推理速度主要受内存带宽限制，更小的量化 → 更快

练习 **2**：自定义采样器配置

使用 llama-cli 的参数配置不同的采样策略，观察生成结果的差异。

参考答案

```
# 贪心解码（确定性）
./llama-cli -m model.gguf --temp 0 -p "The meaning of life is"

# 高温度（更随机）
./llama-cli -m model.gguf --temp 1.5 -p "The meaning of life is"

# 保守采样
./llama-cli -m model.gguf --temp 0.3 --top-k 10 --top-p 0.9 -p "The meanin
g of life is"

# Mirostat
./llama-cli -m model.gguf --mirostat 2 --mirostat-lr 5 -p "The meaning of
life is"
```

练习 3：启动 llama-server 并调用 API

启动 llama-server，使用 curl 调用 chat completion API。

参考答案

```
# 启动服务器（后台）
./llama-server -m model.gguf --port 8080 &

# 调用 API
curl -s http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "What is llama.cpp?"}
    ],
    "temperature": 0.7,
    "max_tokens": 200
  }' | jq .

# 流式响应
curl -s http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{"messages":[{"role":"user","content":"Hello"}],"stream":true}'
```

拓展挑战

- [ ] 使用 imatrix 优化量化质量，对比有无 imatrix 的 perplexity 差异
- [ ] 配置 Grammar 约束输出为 JSON 格式
- [ ] 使用 llama-server 的多 slot 功能实现并发推理
- [ ] 对比不同量化级别在特定任务（如代码生成）上的表现差异

采样、量化与部署

> llama.cpp 提供可组合的采样器链和多种量化方案，以及 OpenAI 兼容的 HTTP 服务器用于生产部署。

涵盖内容

章节	核心主题
概念	采样器链、量化类型、llama-server
代码走读	llama-sampler.cpp、quantize、server
练习	量化对比、API 调用、性能调优

核心概念

- sampler-chain — 可组合的采样器管道
- quantization — 将浮点权重压缩为低比特整数
- imatrix — Importance Matrix，提升量化质量
- **llama-server** — OpenAI 兼容的 HTTP API 服务器

前置知识

- Transformer 推理图
- KV Cache 与批处理
- 了解 LLM 推理的 sampling 过程

学习路径

读完本主题后，你将理解：

- 采样器链的设计模式与各采样器的作用
- 不同量化级别的精度/速度权衡
- 如何使用 llama-server 部署 API 服务
- 量化的数学原理与实践技巧

→ 恭喜完成全部学习模块！