Capstone 实战项目

> 学完 15 天的概念和练习，是时候把知识串起来动手做了。以下三个项目从易到难，分别检验"能追踪"、"能修改"、"能分析"三个层次的能力。建议按顺序完成。

项目一：端到端请求追踪

难度：中级 耗时：2-3 小时 检验能力：能否在真实运行中将 15 天的知识对应到具体函数和调用栈

目标

用一个 HTTP 请求追踪从 TCP 连接到 SSE 输出的完整生命周期，写出一份调试报告。

步骤

1. 编译 **debug** 版本

```
make clean && make CFLAGS="-O0 -g" ds4-server
```

2. 用 **lldb attach** 并设断点

```
break set -n client_main
break set -n parse_chat_request
break set -n render_chat_prompt_text
break set -n ds4_tokenize_rendered_chat
break set -n ds4_session_sync
break set -n prefill_layer_major_cpu
break set -n layer_forward_raw_swa_one
break set -n sample_full_vocab
break set -n sse_chunk
```

3. 发送请求

```
curl -X POST http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{"model":"deepseek","messages":[{"role":"user","content":"What is Redis?"}],"stream":true}'
```

4. 在每个断点记录
   - 调用栈（`bt`）
   - 关键参数（prompt 文本、token 数量、pos 值、采样结果）
   - 执行时间（用 `finish` + 时间戳估算）

交付物

一份报告，包含：

- 请求的完整调用链（函数名 → 文件:行号 → 关键参数）
- prefill 处理了多少 token，decode 生成了多少 token
- 每个阶段的耗时估算
- 对应到哪一天的学习笔记

项目二：给 ds4-server 添加 /v1/models 端点

难度：中高级 耗时：3-4 小时 检验能力：能否读懂现有代码并做最小修改

背景

很多 OpenAI SDK 兼容客户端（如 Claude Code、Continue）启动时会调用 `GET /v1/models` 获取可用模型列表。ds4-server 目前没有这个端点，返回 404。

目标

在 `ds4_server.c` 中实现 `GET /v1/models`，返回符合 OpenAI API 格式的 JSON 响应。

步骤

1. 研究 **OpenAI API** 格式

```json
{
  "object": "list",
  "data": [
    {
      "id": "deepseek-v4-flash",
      "object": "model",
      "created": 1700000000,
      "owned_by": "deepseek"
    }
  ]
}
```

2. 找到路由逻辑：在 `client_main()` 中定位处理 HTTP 路径的代码
3. 添加路径匹配：在 `/v1/chat/completions` 的判断之前，加一个 `strcmp(path, "/v1/models")` 分支
4. 构造响应：用 `snprintf` 拼接 JSON，通过 `send` 返回
5. 测试

```bash
curl http://localhost:8080/v1/models
# 预期：返回上面的 JSON
```

进阶挑战

- 从 GGUF 元数据中读取真实模型名称，不硬编码
- 支持同时加载多个模型时返回列表

项目三：推理性能 Benchmark 分析

难度：高级 耗时：4-6 小时 检验能力：能否用数据验证学到的理论知识

目标

对 ds4 进行系统性能分析，用数据回答：推理瓶颈到底在哪里？

步骤

1. 基准数据收集

编写脚本，测试以下矩阵：

变量	值
上下文长度	128, 512, 2K, 8K, 32K
温度	0 (argmax), 0.7
后端	Metal vs CPU (`make cpu`)
线程数	1, 2, 4, 8

对每个组合，记录：

- Prefill 时间
- Decode 速度（tokens/s）
- 首 token 延迟（TTFT）

2. 可视化

用 Python + matplotlib 画出：

- 上下文长度 vs prefill 时间（验证 O(n) 假设）
- 上下文长度 vs decode 速度（是否有 KV cache 扫描开销？）
- 线程数 vs decode 速度（验证线性扩展）
- Metal vs CPU 加速比

3. 分析与理论对比

用学到的知识解释数据：

- Day 10 理论说 decode 瓶颈是内存带宽，数据是否支持？
- Day 11 理论说线程池线性加速，实际扩展比是多少？
- Day 8 的 KV Cache 压缩比，在不同上下文长度下对性能的影响？
- Day 15 的 Metal 加速在哪个阶段最有效？

交付物

一份分析报告，包含：

- 原始数据（CSV）
- 4 张图表
- 每个异常数据点的解释（与哪一天学到的理论相关）
- 优化建议（如果让你改 ds4.c，你会从哪里入手提速？）

完成标准

项目	最低标准	理想标准
请求追踪	追踪到 5 个核心函数	追踪到所有 9 个断点 + 耗时分析
/v1/models	硬编码返回正确 JSON	从 GGUF 元数据读取模型名
Benchmark	1 组对比数据 + 1 张图	完整矩阵 + 4 张图 + 分析报告

完成任一项目后，你已从"看懂代码"进阶到"能操作代码"——这是 15 天学习的最终目标。

下载离线版

离线阅读 ds4.c 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

> 合作式中断——Agent 的 Ctrl+C 不再暴力终止推理，而是在安全检查点优雅停止，保留有效的 KV cache 状态。

为什么需要合作式中断

ds4-agent 运行时，用户按 Ctrl+C 可能发生在：

- **Prefill** 中间：正在处理长 prompt 的某个 chunk
- 生成循环中：正在逐 token 采样
- **Compaction** 中间：正在压缩对话上下文

之前的行为：Ctrl+C 直接终止操作，KV cache 可能处于不一致状态——下次推理需要从头 prefill。

合作式中断的核心思想：在安全点检查中断信号，优雅地回退到 **IDLE** 状态。

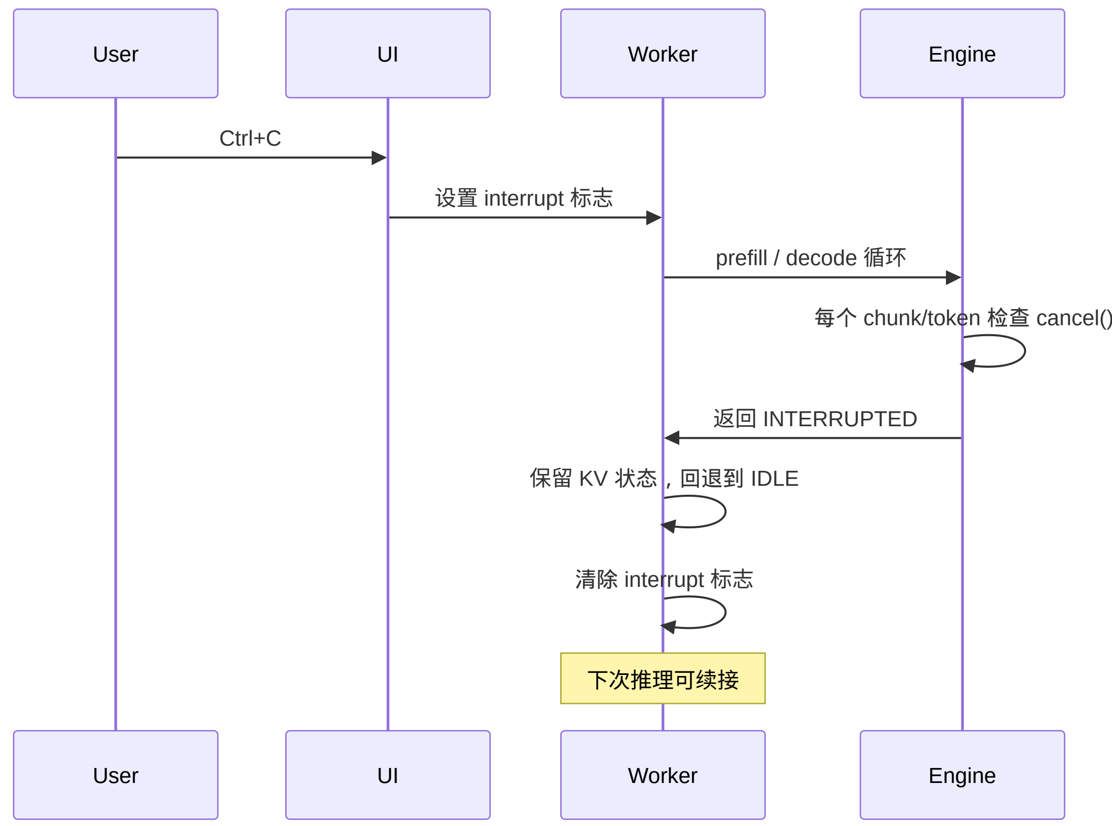
中断回调 API

```
// ds4.h — 中断回调
typedef bool (*ds4_session_cancel_fn)(void *ud);
#define DS4_SESSION_SYNC_INTERRUPTED 2

// 注册回调：推理引擎在每个安全点调用 fn(ud) 检查是否应中断
void ds4_session_set_cancel(ds4_session *s, ds4_session_cancel_fn fn, void *ud);
```

`ds4_session_sync()` 的返回值现在有三种：

- `0` — 正常完成
- `1` — 错误
- `DS4_SESSION_SYNC_INTERRUPTED (2)` — 被中断，但检查点有效

Agent Worker 的中断流



关键步骤：

1. UI 线程设置 `worker_should_interrupt = true`
2. Worker 在 prefill 前 `ds4_session_set_cancel(session, worker_cancel_session_cb, w)`
3. 推理引擎在 chunk/token 边界检查回调
4. 中断后返回 `DS4_SESSION_SYNC_INTERRUPTED` ，`checkpoint_valid = true`
5. Worker 清除中断标志（`worker_clear_interrupt`），防止过期中断阻塞下一轮

各阶段的中断行为

阶段	中断处理	状态保留
Prefill	chunk 边界检查，在最后一个有效前缀处停止	KV cache 到中断点有效
Generation	每个 token 后检查，推送 EOS	完整对话状态保留
Compaction	中断时保留压缩前的对话状态	"Compaction interrupted; keeping the previous conversation state"

Web 工具的中断扩展

合作式中断延伸到 ds4_web 浏览器工具：

```
// ds4_web.h — 可取消的阻塞操作
typedef bool (*ds4_web_cancel_fn)(void *privdata);

// 可中断的睡眠：每 50ms 检查一次取消
web_sleep_ms(web, timeout);

// 所有阻塞操作（CDP WebSocket 连接、页面加载、滚动等）都检查取消
```

之前 `google_search` 或 `visit_page` 可能阻塞 30+ 秒等待 Chrome，现在 Ctrl+C 可以立即中断。

相关概念

- dsml — DSML 解析器支持中断后的状态恢复
- kv-cache — 合作式中断保证 KV cache 在中断后仍然有效
- sse — 服务层的中断传播机制

详见 Part 5 — ds4-agent。

出链

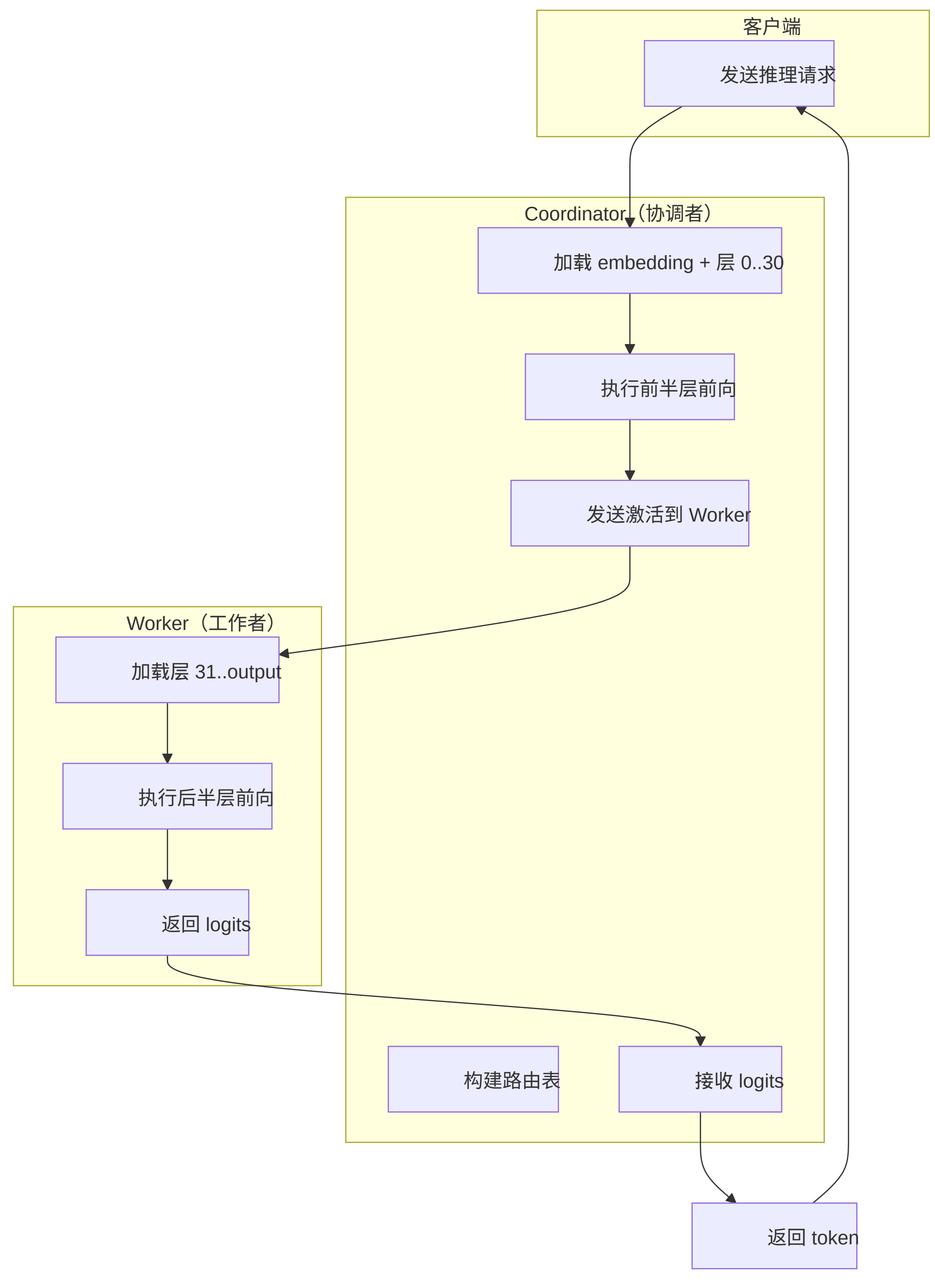
Part 5: 服务层

> 分布式推理——将超大模型（如 DeepSeek V4 PRO，1.6T 参数）的层切片分配到多台机器上协同执行，突破单机内存限制。

为什么需要分布式推理

DeepSeek V4 PRO 的 Q4_K 量化模型需要 ~838GB 存储，远超任何单机内存。即使 IQ2_XXS 版本也需 ~430GB，只有 512GB Mac Studio 才能单机运行。分布式推理将模型按层切分，每台机器只加载一部分，通过高速网络传递中间激活。

Coordinator/Worker 架构



角色	职责	命令
Coordinator	接收请求、执行前半层、转发激活、采样 token	`./ds4 --role coordinator -m pro-q4-layers00-30.gguf`
Worker	注册层范围、执行后半层、返回 logits	`./ds4 --role worker -m pro-q4-layers31-output.gguf`

Worker 启动时发送 HELLO 消息（含 `layer_start`、 `layer_end`、 `model_id`、 `quant_bits`），Coordinator 据此构建路由表。

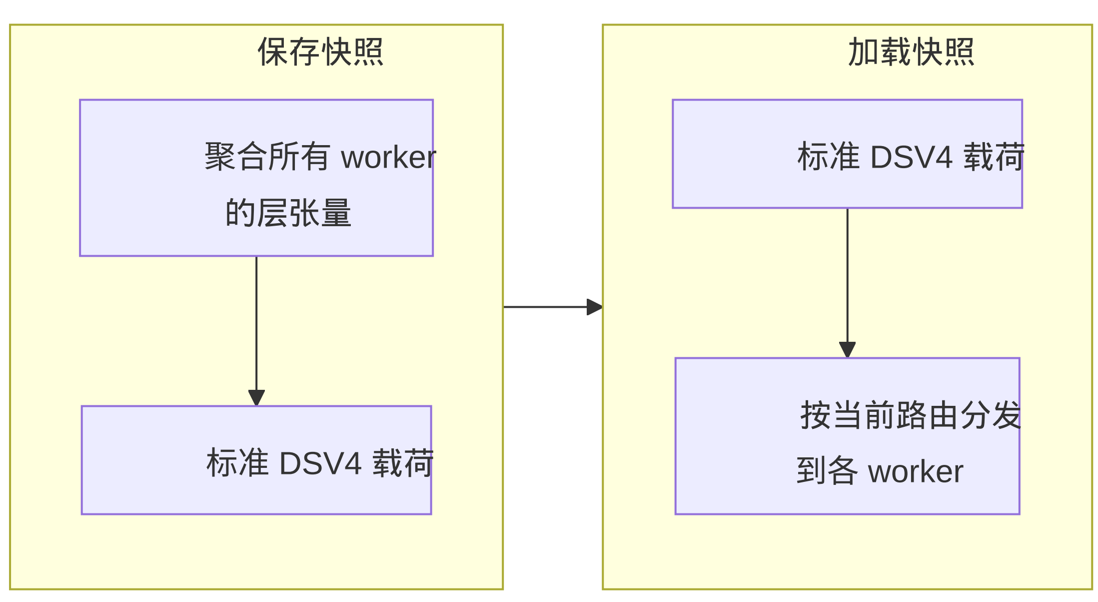
传输协议

自定义 TCP 二进制协议，magic `0x44533444`（"DS4D"）：

```
帧头 → 消息类型（HELLO / WORK / RESULT / SNAPSHOT）→ 载荷
每条消息附带遥测：eval_time、downstream_wait、forward_send、byte_count
```

激活传输支持可配置的 `activation_bits`（默认 32 = float32），低精度传输减少网络带宽消耗。

KV 快照的拓扑无关设计



保存时聚合所有 worker 持有的层张量为统一格式，加载时按当前实际路由分发。这意味着：

- 2-worker 保存的快照可以在 3-worker 配置下加载
- 保存文件格式与单机完全兼容

KV 快照请求 ID 隔离

分布式协议使用递增的 `request_id` 匹配请求-响应。KV 快照操作在独立的数据连接上执行，可能和流水线化的 WORK 请求并发。之前快照复用了 WORK 的 `request_id` 序列，导致响应匹配混乱。

修复：快照使用独立的 `snapshot_request_id` 计数器，从 `UINT64_C(1) << 63`（高半 ID 空间）开始：

```
// ds4_distributed.c
uint64_t snapshot_request_id; // 从高半 ID 空间开始，避免和 WORK 冲突
// 快照操作: d->snapshot_request_id++
// WORK 操作: d->request_id++
```

同时移除了控制 socket 的默认 `SO_RCVTIMEO`——Worker 控制连接在 KV 快照传输期间可能长时间空闲，不应因超时断开。`DS4_DIST_SOCKET_RECV_TIMEOUT_SEC` 环境变量可在需要时显式设置超时。

模型分片加载

分布式引擎使用 model span API 只加载分配的层范围：

```
// 指定需要映射的字节范围（offset/size 对）
ds4_gpu_set_model_map_spans(
    model_map, model_size,
    offsets, sizes, span_count,
    max_tensor_bytes
);
```

Worker 不需要加载 token embedding 和 output head（除非它是路由的最后一跳），大幅减少每台机器的内存占用。

相关概念

- moe — PRO 模型有 384 个路由专家，Q4_K 量化
- kv-cache — 分布式 KV 快照的拓扑无关设计
- quantization — Q4_K 是 PRO 路由专家的量化格式
- mmap — 模型分片通过 mmap + span API 按需加载
- ssd-streaming — 另一种突破单机内存限制的方式（SSD 缓存专家）

详见 Part 6 — 分布式推理。

出链

Part 6: GPU 加速

> DeepSeek Markup Language——DeepSeek 模型的原生工具调用格式，使用类似 XML 的标签结构表示函数调用和参数。

为什么需要 DSML

大语言模型输出的是文本序列。要让模型"调用工具"（如读文件、执行命令），需要一种文本格式来结构化地表达函数名和参数。DeepSeek 使用自有的 DSML 格式，而 OpenAI/Anthropic 使用 JSON 格式。ds4_server 做两种格式的双向转换。

标签结构

```
<｜DSML｜tool_calls>
<invoke name="read_file">
<parameter name="path" string="true">/etc/hosts</parameter>
</invoke>
</｜DSML｜tool_calls>
```

关键元素：

- `<｜DSML｜tool_calls>` — 工具调用块的起始标记
- `<invoke name="...">` — 单个函数调用
- `<parameter name="..." string="true">` — 参数（`string="true"` 表示字符串类型）
- 嵌套参数支持——外层 `<parameter>` 内可包含子参数，解析为 JSON 对象

流式解析状态机

ds4_server 在模型生成过程中实时追踪 DSML 状态：

```
DSML_DECODE_OUTSIDE          — 不在工具调用内
DSML_DECODE_STRUCTURAL       — DSML 标签结构（用温度 0 保证可解析）
DSML_DECODE_STRING_BODY      — 字符串参数值（正常采样温度）
DSML_DECODE_JSON_STRUCTURAL — JSON 参数内的结构部分
DSML_DECODE_JSON_STRING      — JSON 字符串值
```

关键设计：标签结构用温度 0 保证可解析性，参数载荷（文件内容等）用正常温度避免重复。

Agent 工具结果包装

在 ds4-agent 中，工具执行结果现在以 `<tool_result>...</tool_result>` 标签包装后追加到对话历史。`/history` 命令自动剥离这些包装标签显示干净内容。畸形 DSML 调用触发语法提醒（`agent_dsml_syntax_reminder`）。

畸形 DSML 修复

模型输出因 `max_tokens` 截断时，`try_repair_dsml` 尝试修复：

1. 在最后一个 `</thinking>` 后搜索 DSML 标记
2. 统计已打开但未关闭的标签
3. 按逆嵌套顺序追加关闭标签

解析器加固

DSML 流式解析器经过多轮加固，处理模型输出的各种边缘情况：

- 标签校验：`agent_dsml_open_tag_is()` 精确匹配开标签（标签名后必须跟 `>` / 空格 / Tab / 换行），不再使用会误匹配子串的 `strncmp`
- 文本中的 **DSML** 检测：`agent_stream_note_plain_dsml_byte()` 检测正常文本中出现的 DSML 标记，触发 `AGENT_DSML_ERROR`
- 隐式 **Invoke**：当模型省略 `<tool_calls>` 外层包装时，解析器自动合成规范开标签
- 参数关闭标签追踪：`param_close_prefix` 字段追踪部分关闭标签，支持 greedy 采样判断
- 畸形错误报告：终端显示红色 `[invalid tool call: ...]` 错误信息
- **Marker** 检测器：`agent_dsml_marker_detector` 同时检测标准 `｜DSML｜` 和非标准 `DSML｜` 格式

与 **OpenAI Function Calling** 的区别

方面	DSML	OpenAI Function Calling
格式	类 XML 标签	JSON `tool_calls` 数组
流式	天然支持（增量标签）	需要 JSON 拼接
参数类型	`string="true"` 属性	JSON schema 类型
嵌套	支持 `<parameter>` 嵌套	扁平 JSON 对象

相关概念

- kv-cache — DSML 工具记忆的精确回放保护 KV cache 一致性
- sse — 工具调用通过 SSE 流式推送给客户端
- cooperative-interruption — DSML 生成过程中可通过合作式中断优雅停止

> 详见 Part 5 — Tool Calling。

出链

Part 5: 服务层

> SSD 流式加载——让超出 RAM 容量的模型通过"热专家缓存 + 冷专家按需从 SSD 加载"的方式运行，把"能不能跑"从硬截止变成速度的连续谱。

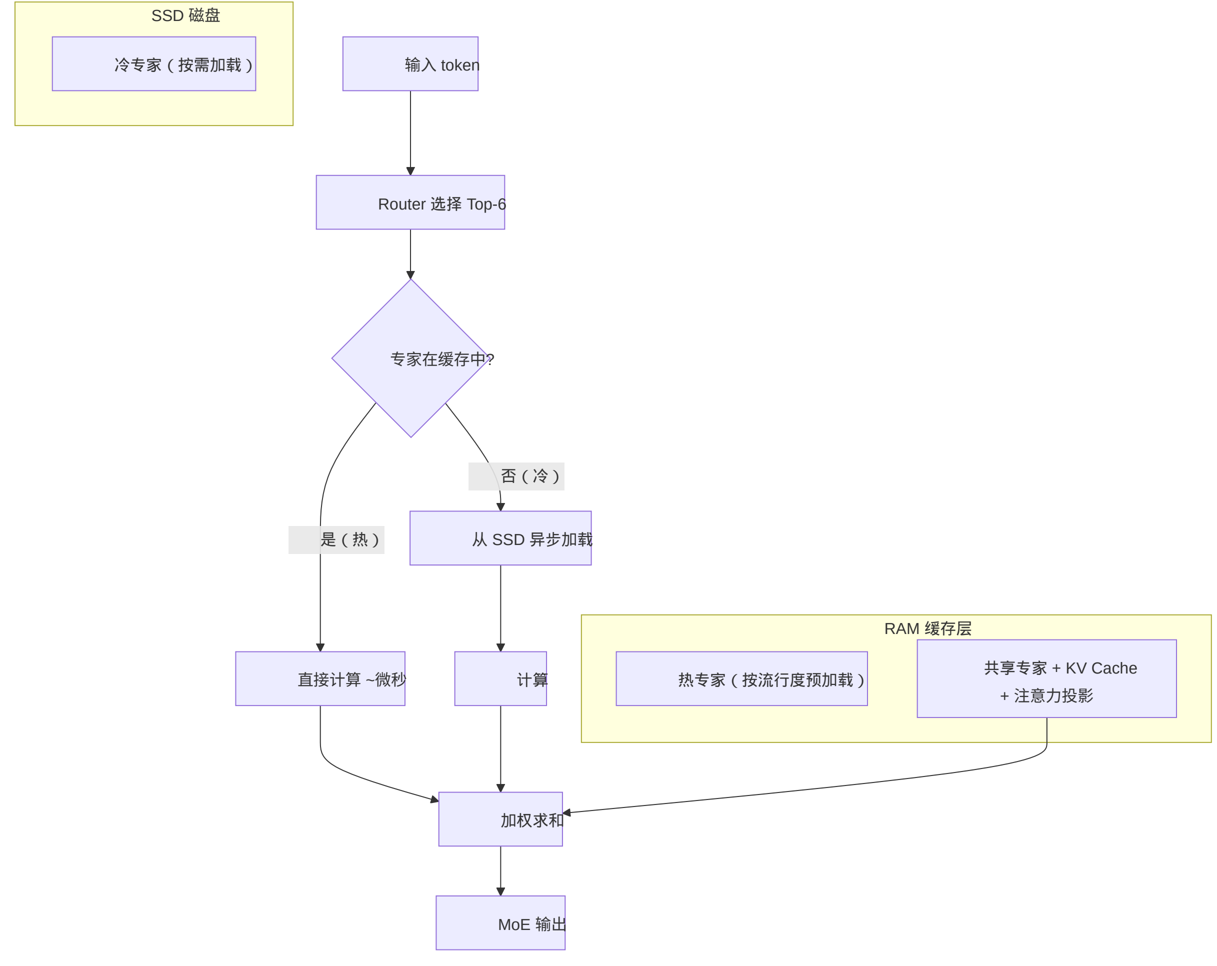
为什么需要 SSD Streaming

DeepSeek V4 Flash 的 IQ2_XXS 模型约 81GB，128GB MacBook 可以完整加载。但 PRO Q4 模型需要 ~838GB，远超单机内存。即使 Flash 模型，如果用户只有 64GB RAM，也无法运行。

SSD Streaming 的核心洞察：MoE 层有 256 个路由专家，但每个 token 只激活 6 个。路由专家占模型大部分空间，但大部分时间都在睡觉。把"热"专家放 RAM，"冷"专家留磁盘按需加载，就能在有限 RAM 下运行超大模型。

```
传统: RAM 必须 ≥ 模型大小 → 硬截止，差 1GB 也不行
SSD:  RAM 越多 → 缓存越多 → 越快，RAM 少 → 缓存少 → 慢但能跑
```

核心原理



仍需 **RAM** 的部分：共享专家、KV cache、embedding、Router、output head——这些每个 token 都要用。只有路由专家的权重可以留在 SSD 上。

缓存计划（**Cache Plan**）

`ds4_ssd_cache_plan` 计算给定 RAM 预算下能缓存多少完整专家：

```c
typedef struct {
    uint64_t model_target_bytes;    // 推荐字节的 80%
    uint64_t cache_bytes;           // 总预算减去非路由部分
    uint64_t effective_cache_bytes; // 实际使用的缓存字节
    uint32_t cache_experts;         // 可缓存的专家数量
} ds4_ssd_cache_plan;

// 自动计算：给定推荐工作集 → 扣除非路由权重 → 算能放几个专家
ds4_ssd_auto_cache_plan(recommended_bytes, non_routed_bytes, per_expert_bytes, max_model_experts, &out);
```

目标：80% 的推荐内存用于缓存，留 20% 给 KV cache 和工作缓冲区。

显式指定的 `--ssd-streaming-cache-experts 32GB` 不是通用字节缓存——引擎把它换算成"当前 GGUF 能放下多少个完整路由专家"（gate + up + down 一起算）。32GB 对 Flash IQ2/Q2 GGUF 约等于 4854 个路由专家。只有自动预算模式才会替你做"扣除非路由权重"的减法；显式设值时，非路由权重、KV cache、graph scratch 和激活所需的内存需要你自己留余量。

缓存上限与 **mlock** 余量

显式请求的 `NGB` 缓存如果太大，会在推理开始前被封顶——否则 macOS 会拒绝 `mlock` 锁定或系统卡死。封顶的判据是"专家缓冲区必须保持可锁定（lockable）"，而不是"尽量满足用户请求"。

当系统处于额外内存压力下、`mlock` 仍然失败时，引擎不会硬塞可换页（pageable）的专家缓存条目，而是：

1. 释放一段已锁定的缓存余量（locked-cache margin）；
2. 用实测可锁定的缓存大小继续运行。

这条策略比"要么全锁、要么报错退出"更实用：内存压力是动态的，宁可缓存小一点也别让系统陷入换页泥潭。环境变量 `DS4_METAL_STREAM_EXPERT_CACHE_*`（及等价的 CUDA/ROCm 路径）可以观测这些决策。

启动热度排名表（Hotlist）

`ds4_streaming_hotlist.inc`（13,334 行）是一个预生成的静态热度排名表，按流行度排列所有 (layer, expert_id) 对。启动时预加载最热门的专家，确保常用专家在第一次交互前已在缓存中。

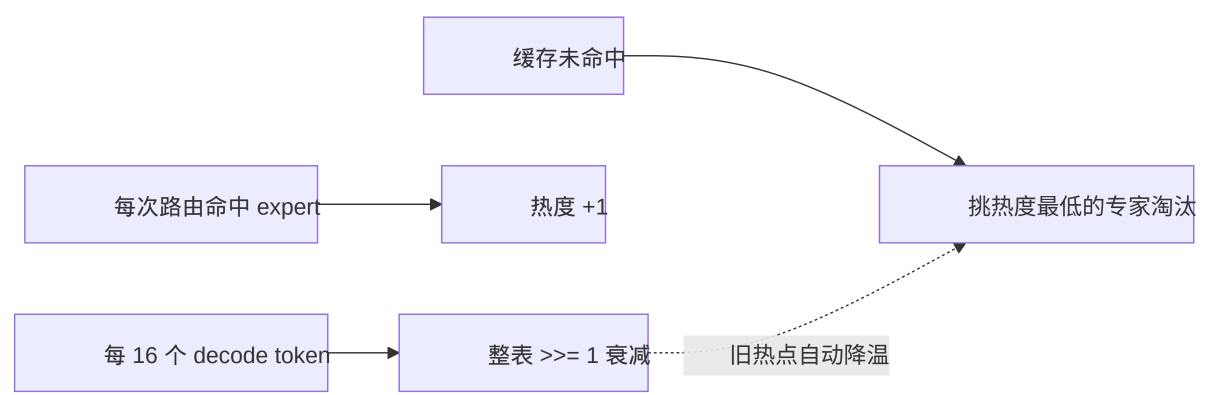
自动预热默认封顶 **4096** 个专家——避免启动时花太久把缓存填满。需要测量更大预热规模时才用 `--ssd-streaming-preload-experts N` 显式指定；正常使用应保留默认预热，不要加 `--ssd-streaming-cold`（冷启动跳过预热，首次交互会很慢）。

运行期路由热度淘汰（Route Hotness）

Hotlist 是启动时的静态预热；真正决定运行期淘汰谁的是一张运行期热度表 `g_stream_expert_cache_route_hotness[layer][expert]`。它取代了纯 LRU：

```
// 每处理 16 个 decode token，整张热度表右移一位（>>= 1）衰减
enum { DS4_METAL_STREAM_EXPERT_HOTNESS_DECAY_TOKENS = 16 };

static void ds4_gpu_stream_expert_cache_decay_route_hotness(void) {
    for (layer ...) for (expert ...)
        g_stream_expert_cache_route_hotness[layer][expert] >>= 1;
}
```

淘汰时优先驱逐热度最低的专家。衰减粒度 **16 token** 是关键参数：比"逐 token 衰减"便宜得多（整张表一次性移位），又比"从不衰减"更适应工作负载变化——一串 token 反复命中某专家会让它迅速变热，而冷却下来的专家会被新热点挤出去。



作用域是**"提示局部"（prompt-local）**：每次新 prompt 开始（`start == 0`）时调用 `ds4_gpu_stream_expert_cache_reset_route_hotness()` 清零热度表——因为新对话的路由分布与上一轮无关。但常驻缓存本身有意跨会话保持热。`ds4_gpu.h` 的注释把这条策略讲得很明白：

```
/* Reset only the prompt-local eviction heuristic.  The resident SSD expert
 * cache itself is intentionally kept warm across sessions. */
void ds4_gpu_stream_expert_cache_reset_route_hotness(void);
```

为什么：热度表只描述"这一轮里谁该被淘汰"，是会过时的启发式，新 prompt 必须重置；而缓存里已加载的专家是真金白银加载过的数据，跨会话复用零成本，没有理由清掉。区分"易变的启发式状态"和"昂贵的物化缓存"是这条设计的核心。

三路 Prefill 调度

短 prompt 在 streaming 下很慢（每层的固定开销主导），因此默认 prefill 调度器根据 prompt 长度走三条路径：

Prompt 规模	策略	原因
极短	逐 token decode 式 prefill	批量加载专家的开销不划算
中等	selected-expert 批量 prefill	只加载被选中的专家，按需
长	全层 layer-major prefill	一次处理多层、多 token，摊薄 SSD 读取

切分点按量化格式调优（`metal_graph_use_streaming_decode_prefill` 的 prefill 截止阈值），避免短 prompt 落到"加载一堆用不上的专家"的慢路径。这也是为什么短 prompt streaming 比 long prefill 慢得多——长 prefill 一层能处理很多 token，缓存命中率高得多。

混合精度路由专家（Mixed-Precision Routed Experts）

当 GGUF 的路由专家跨层不统一量化时（例如全局 IQ2_XXS/Q2_K，但有几层用 `--tensor-type` 提升到 Q4_K，即"per-layer boosted quants"），streaming 路径曾经每个请求都失败：

- batch-selected-addr prefill 用 decode 静态 span 集映射"提升层"（没有 exps 张量）；
- Metal encoder 的 IQ2-only gate 把非 IQ2 层丢给 full-tensor `wrap_model_range` 兜底；
- 结果没有任何已安装视图覆盖该范围（`Metal model range ... is not covered by mapped model views`）。

更隐蔽的是：专家缓存的单尺寸 **slab** 分配器会被"超大专家"毒化——`note_expert_size` 用 last-writer-wins 记录专家尺寸，预算被扭曲，封顶后可能 slab 复用饥饿。

修复分两部分：

1. 检测提升层（`weights_streaming_layer_experts_uniform`：某层的 per-expert 字节数 ≠ 首个路由层推出的 slab 尺寸类），把这些层的 `ffn_{gate,up,down}_exps` 张量纳入 decode-span 构建器（`model_map_span_vec_include_layer_decode`），让现有兜底路径能覆盖它们。
2. 启动时预置 **slab** 尺寸类（`ds4_gpu_set_streaming_expert_cache_expert_bytes`），并让 `note_expert_size` 改为 **freeze+reject** 而非 last-writer-wins——超大尺寸的层确定性绕过缓存、走 mapped-view 路径，不污染 slab 分配器。

启动日志会打印 uniform/boosted 的分布，并在"多数路由层都偏离尺寸类"时告警。均匀模型按构造与改动前逐字节相同（helper 对每层都返回 true，不加额外 span，预置值等于第一次 `note` 会写入的值）。

GPU 后端集成

SSD streaming 现已跨三个 **GPU** 后端实现，统一在 `ds4_gpu.h` 抽象层下：

- **Metal**（`ds4_metal.m`）：最完整的实现，专家缓存管理、异步加载、路由热度淘汰、三路 prefill。
- **CUDA**（`ds4_cuda.cu`）：SSD streaming 完整实现（不再是 stub），含有界缓存分配和进度上报，与 PRO/Flash 路径集成。
- **ROCm**（`ds4_rocm.cu` + `rocm/*.cuh`）：selected-expert 缓存、重叠读取、全层 streaming prefill、常驻缓存查找、ROCm 缓存复用处理。

核心 GPU 抽象层函数（streaming API 已重构为传一张 `ds4_gpu_stream_expert_table` 表，避免长参数列表）：

```
// 重构后：把 model_map/offsets/bytes 打包成一张表，跨后端共用
typedef struct ds4_gpu_stream_expert_table {
    const void *model_map;    uint64_t model_size;
    uint32_t layer;           uint32_t n_total_expert;
    uint64_t gate_offset, up_offset, down_offset;
    uint64_t gate_expert_bytes, down_expert_bytes;
} ds4_gpu_stream_expert_table;

int ds4_gpu_stream_expert_cache_begin_selected_load(
        const ds4_gpu_stream_expert_table *table,
        const int32_t *selected_ids, uint32_t n_selected);  // 异步加载缺失专家
uint32_t ds4_gpu_stream_expert_cache_current_count(void);   // 当前缓存专家数
```

`#ifdef DS4_ROCM_BUILD` 守卫了 ROCm 独有的 layer-major 批量加载钩子（`ds4_gpu_stream_expert_cache_load_layer` 等），非 ROCm 后端提供 no-op stub 保持链接通过。

内存模拟锁（Memory Lock）

`ds4_ssd_memory_lock` 用 `mmap` + `mlock` 模拟已占用内存，用于测试：

```
// --simulate-used-memory 32GB 模拟系统已用 32GB
// 分 256 MiB 块逐块 mlock，避免长时间不可中断
ds4_ssd_memory_lock_acquire(lock);
```

命令行选项

选项	说明
`--ssd-streaming`	启用 SSD streaming 模式
`--ssd-streaming-cache-experts N`	指定缓存专家数量（`32GB` 等字节预算，非专家个数；过大时推理前自动封顶）
`--ssd-streaming-preload-experts N`	显式指定启动预加载专家数（默认自动封顶 4096）
`--ssd-streaming-cold`	冷启动（不预加载 hotlist；仅用于测量最坏情况）
`--simulate-used-memory SIZE`	模拟已占用内存

> 从用户视角看，SSD streaming 是以 **Metal** 为主的容量模式（上游 README 措辞为 "Metal-only capacity mode"）。但 streaming expert cache 的抽象层在 CUDA/ROCm 后端同样已实现（见上文 GPU 后端集成），只是成熟度和文档化程度不同。

相关概念

- moe — MoE 路由专家是 SSD streaming 的缓存对象
- mmap — SSD streaming 使用 mmap 按需加载专家权重
- quantization — 混合精度路由专家（per-layer boosted quants）触发 streaming 的特殊处理路径
- distributed-inference — 另一种突破单机内存限制的方式

详见 Part 6 — GPU 加速。

出链

Part 6: GPU 加速

ds4.c

DeepSeek V4 Flash 推理引擎学习笔记

6 大主题掌握 44K 行零依赖 LLM 推理引擎 —— 从 C 语言基础到 Metal GPU 加速

开始学习 知识地图

纯 C 实现

零外部依赖 —— JSON 解析、HTTP 服务器、BPE 分词器全部手写，从头理解每一行代码

LLM 内核原理

Self-Attention、KV Cache、RoPE 位置编码、MoE 混合专家、SwiGLU 激活 —— 用代码讲清 Transformer

Metal GPU 加速

Objective-C 互操作、Metal Shading Language kernel、Flash Attention，Apple Silicon 原生推理

44K 行代码走读

逐函数注释关键路径，配合 Mermaid 流程图和术语表，让大型 C 项目不再 intimidating

按主题学习

6 大主题覆盖构建加载、分词采样、模型架构、推理流程、服务层、GPU 加速，按需查阅

术语表 & 知识链接

Obsidian 风格双向链接，悬浮预览 + 反向引用，构建个人知识图谱

学习日志

Day 1 — 项目全貌与编译系统

日期：2025-05-09 状态：已完成

今日摘要

- 理解了 Makefile 的完整构建链：`make` → 编译 .o → 链接可执行文件
- 掌握了条件编译：`-DDS4_NO_METAL` 控制平台分支，`__ARM_NEON` 由编译器预定义
- 读了 ds4.c 文件头（行 1-120）：18 个系统头文件、模型架构常量 enum
- 理解了 opaque type 封装：`ds4_engine` / `ds4_session` 前向声明
- 学习了 DeepSeek V4 Flash 架构：43 层、284B 参数、256 专家 MoE（仅激活 6 个）
- 关键洞察：专用引擎 > 通用框架——固定架构换来简化和优化

困惑与突破

- 困惑：为什么 `make` 里 `ds4_metal.o` 用 `-fobjc-arc` 而不是 `-std=c99`？一直以为 C 和 Objective-C 只是语法糖的区别。
- 突破：`.m` 文件是 Objective-C，ARC 是 ObjC 的自动内存管理机制。C 和 ObjC 是两种不同的语言，编译器标志完全不同。ds4.c 用纯 C 是为了让引擎核心可移植，Metal 桥接单独用 ObjC 处理。

Day 2 — 内存管理与 mmap

日期：2025-05-09 状态：已完成

今日摘要

- 掌握了 xmalloc/xcalloc/xrealloc 包装器模式和为什么没有 xfree

- 深入理解 mmap：MAP_SHARED（Metal 零拷贝）vs MAP_PRIVATE（CPU 避内核 bug）
- 理解 xmalloc_zeroed 为什么不用 calloc（Darwin 零页 panic 风险）
- 学习 posix_madvise 预取和 volatile 防优化技巧
- 理解 __thread 线程局部存储和预分配 scratch buffer 模式
- 追踪了完整加载链：open → fstat → mmap → cursor 解析 → weights_bind
- 关键洞察：热路径零分配，所有内存操作集中到初始化阶段
- *上游更新*：`model_open` 新增 `prefetch_cpu` 参数，`ds4_dump_text_tokenization()` 传 false 避免遍历张量数据

困惑与突破

- 困惑：为什么 `malloc + memset` 比 `calloc` 更安全？calloc 不是天然初始化为零吗？
- 突破：calloc 的"零页优化"让所有虚拟页指向同一个物理零页。推理时逐 token 写入 KV cache 会触发大量 page fault。在 macOS 上，大 mmap + 大量 page fault 叠加可能导致内核计数器溢出（kernel panic）。`malloc + memset` 在启动时集中触发所有 page fault，把风险控制在初始化阶段。

Day 3 — GGUF 二进制格式

日期：2025-05-09 状态：已完成

今日摘要

- 学习了 GGUF 二进制文件格式：头部(24B) → 元数据表 → 张量目录 → 张量数据
- 掌握了 cursor 游标模式解析二进制：has/read/skip 操作
- 理解了 struct 对齐和 static_assert 编译时检查技巧
- 学习了量化块格式：Q2_K(84B)、Q4_K(144B)、Q8_K(292B)、IQ2_XXS(66B)
- 理解了 parse_metadata 的懒加载和 parse_tensors 的两步解析
- 学习了 config_validate_model 的"早失败"验证
- 关键洞察：懒加载 + 零拷贝字符串引用，避免不必要的内存分配

困惑与突破

- 困惑：`cursor_string` 返回的 `ds4_str` 没有拷贝数据，那 `mmap` 被释放后不就野指针了？

- 突破：ds4.c 的设计是"加载后永不释放"——模型的生命周期覆盖整个进程。`ds4_model` 持有 mmap 映射直到进程退出，所以 `ds4_str` 指针始终有效。这是推理引擎常见的设计权衡：用内存常驻换取零拷贝。

Day 4 — BPE 分词器

日期：2025-05-09 状态：已完成

今日摘要

- 学习了开放寻址哈希表：线性探测、2 的幂容量、& mask 代替取模
- 掌握了 FNV-1a 哈希函数和动态数组翻倍增长模式
- 深入理解 BPE 算法：字节编码→预分词→合并循环→查表得 token ID
- 理解 GPT-2 字节级编码：不可打印字节映射到 256+ 码点
- 学习了 Chat 模板组装：BOS + system + User + prompt + Assistant + 💭
- 关键洞察：分词器是 LLM 的"眼睛"——文本必须先变成数字，模型才能处理

困惑与突破

- 困惑：BPE 合并顺序为什么是从最低 rank 开始而不是从最高？直觉上应该先合并最频繁的。
- 突破：词表中的 rank 越小表示优先级越高。BPE 每轮扫描所有相邻对，找到 rank 最小（优先级最高）的对先合并。"最低 rank = 最高优先级"——这是训练时按频率排序编码的结果，rank 1 就是训练中出现最频繁的合并。

Day 5 — 采样策略

日期：2025-05-09 状态：已完成

今日摘要

- 学习了数值稳定的 softmax（减最大值防溢出）
- 理解 Temperature/Top-K/Top-P/Min-P 四种采样策略及组合
- 掌握了插入排序维护 top-k 数组（O(n×k) 优于 qsort 的 O(n log n)）

- 学习了 SplitMix64 PRNG 和均匀浮点转换
- 追踪了完整采样链：session_sample → top_p_min_p → argmax/full_vocab
- 关键洞察：采样策略在确定性和多样性之间找平衡，温度是核心控制旋钮

困惑与突破

- 困惑：为什么 softmax 要先减最大值再 exp？直接算 exp 再归一化不行吗？
- 突破：浮点数溢出！比如 exp(1000) 是 infinity，整个计算就废了。减去最大值后，最大值变成 exp(0)=1，其他都是 exp(负数) < 1，不会溢出。这个"数值稳定性"技巧从 Day 5 一路用到 Day 8 的注意力计算。

Day 6 — 公共 API 与权重结构

日期：2025-05-09 状态：已完成

今日摘要

- 深入理解 opaque type 封装：ds4_engine/ds4_session 前向声明
- 掌握了结构体嵌套设计：engine → model + vocab + weights[43 layers]
- 理解 weights_bind 按名称查找张量的绑定过程
- 学习了 Transformer 每层的完整权重组成（注意力/FFN/HC）
- 理解了超连接（Hyper-Connection）替代标准残差连接
- 关键洞察：LoRA 低秩分解把 Query 参数减少 72%
- *上游更新*：ds4.h 新增 session rewrite API（`ds4_session_rewrite_from_common`），用于工具调用后检查点恢复；新增 `ds4_dump_text_tokenization` 独立分词调试
- *上游更新（2025-05-15）*：CUDA 新增 managed KV cache 支持——百万 token 级上下文时，KV cache 张量改用 `cudaMallocManaged`（统一内存按需分页），避免 DGX Spark 统一内存系统中 GPU 显存分配饿死 CPU 侧。普通上下文仍用 `cudaMalloc` 设备分配保持性能。通过 `ds4_gpu_should_use_managed_kv_cache()` 自动判断是否启用。

困惑与突破

- 困惑：LoRA 不是用于微调的吗？为什么 ds4.c 的 Q 投影用了 LoRA 结构？
- 突破：LoRA 的低秩分解不仅用于微调——DeepSeek V4 把它作为模型架构的一部分。Q 投影从 4096 直出 32768 维太重了，拆成 4096→1024→32768 两步，参数减少 72%。这不是"微调补丁"而是"架构瘦身"。

Day 7 — 量化与矩阵运算

日期：2025-05-09 状态：已完成

今日摘要

- 掌握了位操作提取 2-bit/4-bit 值的技巧
- 理解了量化 block 布局：Q2_K(84B/256权)、Q4_K(144B)、IQ2_XXS(66B)
- 学习了非对称量化策略：路由专家 2-bit、其他组件 4-8 bit
- 理解了查表法在 IQ2_XXS 解码中的应用
- 学习了 matvec 矩阵-向量乘法在推理中的核心地位
- 关键洞察：284B 模型压缩到 81GB，在 128GB MacBook 上可运行

困惑与突破

- 困惑：2-bit 量化意味着每个权重只有 4 个可能值（00, 01, 10, 11），这怎么可能不严重损失精度？
- 突破：关键在于"非对称量化"——不是所有权重都用 2-bit。路由专家用 IQ2_XXS（~2 bit）因为每个 token 只激活 6/256 个专家，量化误差被加权平均稀释。但 attention 投影和共享专家用 Q4_K 甚至 Q8_K 保持精度。这是"好钢用在刀刃上"的工程取舍。

Day 8 — 注意力机制

日期：2025-05-09 状态：已完成

今日摘要

- 深入理解 Self-Attention：分数计算 → softmax → 加权求和
- 掌握了 KV Cache 的必要性、滑动窗口和 memmove 实现
- 学习了 Attention Sink：可学习 logit 参与 softmax 但不贡献 value
- 理解了 MLA 压缩：KV cache 大小减少 64 倍
- 掌握了混合注意力：raw (最近128) + compressed (历史) 并行

- 关键洞察：压缩 KV cache 让百万 token 上下文在本地成为可能

困惑与突破

- 困惑：滑动窗口 SWA=128 意味着每个 token 只看最近 128 个 token，那 128 之前的信息不就全丢了？
- 突破：两层机制保住信息：(1) 压缩 KV cache（MLA）把历史 token 压缩后保留，注意力时同时看 raw（最近 128）和 comp（历史压缩行）；(2) 43 层堆叠——每层看 128 个 token 的信息经过残差连接传到下一层，128×43 = 5504 的"感受野"。不是"传话游戏会传错"，而是"每层都在批注原始信息"。

Day 9 — MoE 混合专家

日期：2025-05-09 状态：已完成

今日摘要

- 理解了 MoE 路由：哈希路由（前3层）和 top-k 路由（后续层）
- 掌握了 √(softplus(x)) 路由激活和偏置选择/无偏权重的分离
- 学习了 SwiGLU 激活：silu(gate) × up，以及 2-bit 量化时的 clamp
- 理解了路由专家（2-bit）+ 共享专家（8-bit）的非对称精度设计
- 计算了 MoE 效率：284B 参数中每次只激活约 0.1%
- 关键洞察：MoE 用"宽度换深度"——参数多但计算少

困惑与突破

- 困惑：256 个专家只有 6 个被激活，剩下 250 个不是浪费吗？为什么不直接做 6 个大专家？
- 突破：不同 token 激活不同专家——所有 256 个都会被用到，只是不是同时。43 层 × 6 专家 = 258 次专家调用，通过残差连接逐步积累。这好比一个医院有 256 个专科医生，每个病人只需要看 6 个，但不同病人看不同的 6 个。如果只有 6 个全科医生，每个人都要处理所有问题，反而更差。

Day 10 — RoPE 与推理循环

日期：2025-05-09 状态：已完成

今日摘要

- 理解了 RoPE 旋转位置编码：通过 2D 旋转矩阵编码相对位置
- 学习了 YaRN 扩展：混合外推和内插，支持 16× 上下文扩展
- 掌握了 prefill vs decode：批量处理 vs 逐 token 串行
- 理解了自回归生成的本质：每步输出成为下步输入，无法并行
- 学习了 MTP 推测解码的原理和实现
- 关键洞察：decode 的瓶颈是内存带宽，不是计算——每个 token 要读 81GB 权重

困惑与突破

- 困惑：RoPE 把向量做 2D 旋转编码位置，但为什么旋转后点积只依赖相对距离？这个性质是巧合还是设计？
- 突破：这是 2D 旋转矩阵的数学性质：R(a)^T · R(b) = R(b-a)。两个旋转后的向量做点积，结果只包含角度差（相对位置），不包含绝对角度。RoPE 利用了这个性质，所以它不是"巧合"而是精心选择的编码方式。

Day 11 — 线程池

日期：2025-05-09 状态：已完成

今日摘要

- 学习了 pthread API：mutex、condvar、create/join
- 理解了线程池设计：worker 循环、generation 计数器、行范围分配
- 掌握了 cond_wait 必须在 while 循环中的原因（虚假唤醒）
- 理解了 __thread TLS 防止嵌套并行的机制
- 关键洞察：主线程也参与计算，不浪费一个核

困惑与突破

- 困惑：为什么 `pthread_cond_wait` 必须放在 while 循环里？用 if 不就行了吗——signal 了就说明条件满足了？
- 突破：POSIX 规范明确允许"虚假唤醒"——没有 signal 的情况下 cond_wait 也可能返回。而且多个 worker 被 broadcast 唤醒时，只有一个能拿到任务，其他必须继续等。while 循环保证每次醒来都重新检查条件，不信任任何唤醒。

Day 12 — HTTP 服务器

日期：2025-05-09 状态：已完成

今日摘要

- 学习了 POSIX socket：socket/bind/listen/accept/recv/send
- 理解了每连接一线程模型和单推理 worker 的架构
- 学习了 poll() 处理慢客户端的 EAGAIN 场景
- 掌握了 SSE 流式响应格式（data: + 双换行）
- 关键洞察：推理串行但 HTTP 并发，用队列解耦
- *上游更新（2025-05-15）*：新增 `/v1/responses` 端点（OpenAI Responses API），支持 Codex CLI 等 Agent 工具。Server 日志新增 Responses API 标记和进度日志改进，方便调试多轮工具调用。

困惑与突破

- 困惑：为什么每个连接一个线程而不是用 epoll/kqueue 做事件驱动？C10K 问题怎么办？
- 突破：ds4-server 的瓶颈是推理，不是连接。推理同时只能跑一个请求（GPU 独占），所以并发连接数在正常使用中不会超过几十个。每连接一线程简单可靠，是正确的工程选择。如果未来支持 batching，可能需要更复杂的事件模型。

Day 13 — KV Cache 持久化

日期：2025-05-09 状态：已完成

今日摘要

- 学习了原子写入模式（tmp + rename 防崩溃）
- 理解了 SHA1 缓存 key：基于 token IDs 而非文本
- 掌握了磁盘 KV cache 文件格式（KVC 头 + 文本 + session payload）
- 理解了四种保存触发和前缀匹配查找
- 关键洞察：磁盘 KV cache 让长 prompt 的重复 prefill 从 10 秒降到 1 秒

困惑与突破

- 困惑：SHA1 基于 token IDs 而不是文本做缓存 key——为什么？文本不是更直观吗？
- 突破：同一个文本可能被不同版本的分词器切成不同的 token。比如"你好"可能是一个 token 也可能是两个。用 token IDs 做缓存的 key 保证一致性——同样的 token 序列一定产生同样的 KV cache，不受文本编码变化影响。

Day 14 — API 兼容层

日期：2025-05-09 状态：已完成

今日摘要

- 学习了手写递归下降 JSON 解析器：json_string/number/skip_value
- 理解了 OpenAI vs Anthropic API 的格式差异
- 掌握了 Tool Calling 流程：DSML ↔ OpenAI/Anthropic 双向转换
- 理解了 Thinking 模式的三种级别和流式输出
- 关键洞察：零依赖设计——JSON、HTTP、SHA1 全部手写
- *上游更新*：引入 rax 基数树实现工具调用文本记忆（精确回放模型采样的 DSML）；随机 tool call ID（/dev/urandom）；DSML 参数文本不再转义 `<>` `&`，只转义关闭标签；session rewrite 检查点恢复机制；JSON 解析器新增 JSON_MAX_NESTING 256 嵌套深度限制，防止深层嵌套 JSON 栈溢出 DoS
- *上游更新（2025-05-15）*：Anthropic live tool continuation——工具调用后保留 live KV 状态不变，下一请求通过 `tool_use.id` / `tool_result.tool_use_id` 匹配后只追加后缀 token，跳过完整 prefill。Responses API 同理：`call_id` 绑定 live KV 前沿，避免"工具调用后重建整个上下文"的停顿。核心思想：采样的 **KV** 状态是最高保真度的状态，客户端可见的协议对象只应"选择"这个状态，而不是强迫服务器重建它。详见 `misc/RESPONSE_API.md` 和 `misc/ANTHROPIC_LIVE_CONTINUATION.md`。

困惑与突破

- 困惑：手写 JSON 解析器为什么不用 state machine 而用递归下降？递归不是有栈溢出风险吗？
- 突破：确实有风险——这就是为什么上游加了 `JSON_MAX_NESTING 256` 的深度限制。递归下降的好处是代码清晰、容易跳过不关心的字段（`json_skip_value`）。对于一个只需解析固定几个字段的推理服务器，递归下降的简单性远比完整 JSON 库更实用。

Day 15 — Metal GPU + 总结

日期：2025-05-09 状态：已完成

今日摘要

- 学习了 ObjC 互操作：C 头文件隐藏 ObjC 类型
- 理解了 Metal 计算管线：Device → Pipeline → CommandBuffer → dispatch
- 学习了 Flash Attention 分块计算的思想
- 理解了零拷贝 MTLBuffer：GPU 直接引用 mmap 内存
- 用完整请求生命周期串联了 15 天的知识
- 关键洞察：ds4.c 的哲学——垂直整合、专用化、零拷贝、热路径零分配
- *上游更新*：Metal shader 参数改为 char* 表达只读语义；encoder 可选 buffer 绑定零值占位满足 debug layer 校验
- *上游更新（2025-05-15）*：回退了 Metal Q4 expert tensor model views 的扩展（`DS4_METAL_MODEL_MAX_TENSOR_BYTES` 从 2 GiB 回到 ~672 MiB）。原因：该修复是由 AI 生成的且未经用户确认，上游坚持不接受未经人工验证的修复。

困惑与突破

- 困惑：Metal 可以零拷贝直接引用 mmap 内存，那 81GB 模型岂不是不需要任何显存？
- 突破：Apple Silicon 的"统一内存"架构下确实如此——GPU 和 CPU 共享同一块物理内存。Metal 的 `newBufferWithBytesNoCopy` 只是创建一个指向 mmap 区域的 GPU 缓冲区描述符，不拷贝数据。但 GPU 执行时仍然需要把这些数据读入缓存层次，所以带宽是瓶颈。零拷贝省的是"额外的拷贝"，不是"不读数据"。

上游同步 — 2026-06-16

子模块：`vendor/ds4` 从 `c463029` 前进到 `e34a808`（main 分支，43 文件、~27K 行新增）。本轮主线是 **SSD streaming** 的成熟化 + **ROCm** 后端并入 **main**。已把内容融入各章节：

SSD streaming（融入 Part 6 与 术语表）

- 运行期路由热度淘汰取代纯 LRU：`route_hotness[layer][expert]` 表每 16 个 decode token 整表右移衰减一位，淘汰最低热度者。表是提示局部的（新 prompt 清零），但常驻缓存跨会话保温——区分"易变启发式"与"昂贵物化缓存"。
- 三路 **prefill** 调度：极短走逐 token decode 式、中等走 selected-expert 批量、长 prompt 走全层 layer-major，按量化调阈值。
- 缓存上限与 **mlock** 余量：显式 `NGB` 推理前封顶保持可锁定；`mlock` 失败时释放锁定余量、用实测可锁大小继续，而非硬塞可换页条目。
- 混合精度路由专家（per-layer boosted quants）：检测提升层纳入 decode-span 兜底；slab 尺寸类启动时预置、对超大尺寸 freeze+reject 而非 last-writer-wins，防 slab 毒化。
- streaming API 重构为传一张 `ds4_gpu_stream_expert_table` 表；CUDA/ROCm 不再是 stub，已实现 expert cache。
- 新增 `--ssd-streaming-preload-experts N`（默认预热封顶 4096）。

ROCm / Strix Halo（融入 Part 6 新增 §4c）

- ROCm 后端并入 **main**（此前只在独立 `rocm` 分支）。`make strix-halo`（别名 `make rocm`）用 `hipcc` 编 `ds4_rocm.o` + `rocm/*.cuh`（~18K 行），基于 rocWMMA。
- 难点：补全 rocWMMA 内部头、扩大 GTT aperture（内核参数让 128GB 系统暴露 ~124GB GPU 可见内存）、避免 mixed IQ2/IQ4 GGUF 触发系统 OOM。详见上游 `STRIXHALO.md`。
- 融合算子重构为可选后端钩子：`ds4.c` 不再写死 Metal/CUDA 分支，改为 `ds4_gpu.h` 上的可选钩子，新后端按需实现。

推测解码 / MTP（融入 Part 4）

- 修复 `--mtp-draft > 2` 时验证器调错 `topk_tensor` 参数（单行取 top-N vs 每行取 top-1），导致接受了模型从未生成的 token。bug 只在深度 >2 暴露（深度 2 走专用 argmax 分支）。回归测试 `--mtp-verify-depth` 断言"每个提交 token 是其位置近似 argmax"这条不变量。

Agent / Serving（融入 Part 5）

- 未关闭 `<think>` 内的工具调用恢复：检测到完整 stanza 开头时强制喂 `</think>` 前向恢复，而非重写已采样上下文。

- `edit` 工具锚定 `[upto]` 匹配修复（old tail anchor 在 old head 之后找不到）+ 新增 ds4-agent edit 回归测试纳入 `make test`。
- 终端 raw 模式状态在 quit/help/Ctrl+C 路径上正确恢复。

工程治理（融入 Part 1）

- 新增 `QA_BEFORE_RELEASES.md` 发布前 14 节 checklist（覆盖 Metal/CUDA/ROCm/分布式/SSD streaming/KV cache/agent 等组合）。

困惑与突破

- 困惑：为什么运行期淘汰用"每 16 token 整表右移衰减"而不是逐 token 衰减或 LRU 时间戳？
- 突破：逐 token 衰减每步要扫整张 `[layer][expert]` 表（43×256=11K 项），太贵；纯 LRU 又不适应工作负载切换。整表一次性 `>>= 1` 把衰减摊到 16 token 一个周期，开销恒定且小；新热点会因连续命中迅速盖过衰减变热，旧热点 16 token 后就降温——刚好匹配 decode 的局部性节奏。这是"精度换成本"的经典工程取舍：淘汰启发式不需要逐 token 精确，只需要相对热度排序大致正确。

出链

Part 6: GPU 加速

Part 4: 推理流程

Part 5: 服务层

Part 1: 构建与加载

ds4.c 学习知识地图

项目概览

- **ds4.c**：antirez 编写的 DeepSeek V4 Flash 推理引擎
- **~58,000** 行代码（C + Objective-C + Metal Shading Language + CUDA）
- 零外部依赖：JSON、HTTP、分词器全部手写
- 目标平台：Apple Silicon Mac（Metal）、NVIDIA GPU（CUDA）、AMD GPU（ROCm 分支）
- **GGUF** 量化工具：`gguf-tools/` 提供 imatrix 收集和 HF-to-GGUF 量化（~3000 行）
- 贡献指南：`CONTRIBUTING.md`（142 行）含回归测试流程（`make test`、CUDA regression、质量评分）

知识索引

C 语言知识点

概念	主题	对应文件
Makefile / 编译流程	Part 1	Makefile
malloc/free / xmalloc	Part 1	ds4.c
mmap 零拷贝	Part 1	ds4.c
__thread TLS	Part 1	ds4.c
struct 对齐 / static_assert	Part 1	ds4.c
字节序 / 二进制解析	Part 1	ds4.c
哈希表（开放寻址）	Part 2	ds4.c
字符串操作	Part 2	ds4.c
浮点运算 / softmax	Part 2	ds4.c
union 类型转换	Part 2	ds4.c
opaque type 设计	Part 3	ds4.h
位操作 / 量化	Part 3	ds4.c
多维数组索引	Part 3	ds4.c
指针数组	Part 3	ds4.c
volatile / 信号	Part 4	ds4.c
pthread 线程池	Part 4	ds4.c
mutex / condvar	Part 4	ds4.c
POSIX socket	Part 5	ds4_server.c
poll I/O 多路复用	Part 5	ds4_server.c
文件 I/O / SHA1	Part 5	ds4_server.c
JSON 手写解析	Part 5	ds4_server.c
ObjC 互操作	Part 6	ds4_metal.m

LLM 知识点

概念	主题	对应文件
推理引擎是什么	Part 1	README.md
模型文件格式 (GGUF)	Part 1	ds4.c
BPE 分词	Part 2	ds4.c
采样策略	Part 2	ds4.c
Transformer 权重结构	Part 3	ds4.h
模型量化	Part 3	ds4.c
imatrix 重要性矩阵	—	gguf-tools/
GGUF 量化工具	—	gguf-tools/
Self-Attention	Part 3	ds4.c
KV Cache	Part 3	ds4.c
MLA (压缩注意力)	Part 3	ds4.c
MoE (混合专家)	Part 3	ds4.c
SwiGLU 激活	Part 3	ds4.c
RoPE 位置编码	Part 4	ds4.c
prefill vs decode	Part 4	ds4.c
自回归生成	Part 4	ds4.c
推理并行性	Part 4	ds4.c
SSE 流式响应	Part 5	ds4_server.c
Responses API (/v1/responses)	Part 5	ds4_server.c
KV Cache 持久化	Part 5	ds4.c
OpenAI/Anthropic API	Part 5	ds4_server.c
Tool Calling	Part 5	ds4_server.c
Live Tool Continuation	Part 5	ds4_server.c
KV Cache 命中衰减与驱逐	Part 5	ds4_server.c
KV Cache 用量报告	Part 5	ds4_server.c
ds4-eval 能力评估	Part 5	ds4_eval.c
DeepSeek V4 模型卡	Part 1	MODEL_CARD.md
GPU 加速 / Flash Attention	Part 6	metal/ ds4_cuda.cu

概念	主题	对应文件
CUDA Managed KV Cache	Part 6	ds4_cuda.cu

学习进度

天	日期	状态	对应主题
1	2025-05-09	已完成	Part 1: 构建与加载
2	2025-05-09	已完成	Part 1: 构建与加载
3	2025-05-09	已完成	Part 1: 构建与加载
4	2025-05-09	已完成	Part 2: 分词与采样
5	2025-05-09	已完成	Part 2: 分词与采样
6	2025-05-09	已完成	Part 3: 模型架构
7	2025-05-09	已完成	Part 3: 模型架构
8	2025-05-09	已完成	Part 3: 模型架构
9	2025-05-09	已完成	Part 3: 模型架构
10	2025-05-09	已完成	Part 4: 推理流程
11	2025-05-09	已完成	Part 4: 推理流程
12	2025-05-09	已完成	Part 5: 服务层
13	2025-05-09	已完成	Part 5: 服务层
14	2025-05-09	已完成	Part 5: 服务层
15	2025-05-09	已完成	Part 6: GPU 加速

出链

/topics/01-build-load/concepts\

/topics/02-tokenization-sampling/concepts\

/topics/03-model-architecture/concepts\

/topics/04-generation-pipeline/concepts\

/topics/05-serving/concepts\

/topics/06-gpu/concepts\

/topics/01-build-load/index\

/topics/02-tokenization-sampling/index\

/topics/03-model-architecture/index\

/topics/04-generation-pipeline/index\

/topics/05-serving/index\

/topics/06-gpu/index\

LLM 核心架构概念

> 基于 ds4.c（DeepSeek V4 Flash 推理引擎）代码学习整理。所有概念均可在源码中找到对应实现。

---

1. Embedding

问题：Token ID 是整数（如 token 42），整数之间无法做有意义的数学运算（99 - 42 = 57 没有语义）。需要将离散符号映射到连续向量空间。

方案：Embedding 查表。模型有一个 `(129280, 4096)` 的矩阵，每行对应一个 token 的 4096 维向量。

- 语义相近的 token 在向量空间中距离更近
- 向量运算能表达语义关系：`king - man + woman ≈ queen`
- 纯查表操作，无计算

```
// `embed_token_f16()` (ds4.c 约 2538 行)
const uint16_t *row = base + (uint64_t)token * stride;  // 按 token ID 找行
for (uint64_t i = 0; i < stride; i++)
    out[i] = f16_to_f32(row[i]);  // 16位浮点转32位
```

模型规模与 embedding 维度的典型对应：

模型规模	典型 N_EMBD
小模型 (7B)	4096
中模型 (70B)	8192
大模型 (400B+)	16384

---

2. 注意力机制（Attention）

Q / K / V

借用了数据库检索的概念：

- **Query (Q)**：我在找什么
- **Key (K)**：每个东西的标签
- **Value (V)**：每个东西的内容

注意力分数 = Q 和 K 的点积，然后用 softmax 归一化，最后加权取出 V。

Multi-Query Attention (MQA)

DeepSeek 的配置：64 个 Q 头，1 组 K/V。

```
64 个 Q 头 = 64 种不同的"提问角度"，捕获不同类型的关系
1 组 K/V  = 共享的信息源
```

Q 有 64 个头提供多样性，K/V 只有 1 组节省内存。对效果影响很小，但 kv-cache 内存缩小 64 倍。

```
// `layer_attention_rows_one()` (ds4.c 约 4751 行)（核心注意力计算）
for (uint32_t h = 0; h < DS4_N_HEAD; h++) {          // 遍历64个头
    // Q·K 点积算分数
    score[r] = dot_f32(qh, kv, DS4_N_HEAD_DIM) * kq_scale;
    // 内联 softmax：减最大值 → exp → 求和 → 归一化
    // 加权求和 V
    axpy_f32(oh, kv, weight, DS4_N_HEAD_DIM);
}
```

---

3. RoPE 旋转位置编码

问题：注意力分数只看向量内容，不知道 token 顺序。"我爱猫"和"猫爱我"的 Q/K 值一样。

方案：按位置旋转 Q 和 K（不旋转 V，因为 V 不参与分数计算）。

将向量拆成若干对，每对做二维旋转，不同对用不同旋转频率：

```
高频分量（θ 大）：精确区分相邻位置（"放大镜"）
低频分量（θ 小）：感知远距离关系（"广角镜"）
```

关键性质：点积只依赖相对距离。不管绝对位置在哪，距离相同的 token 对有相同的旋转差。

低频分量天然编码了绝对位置的大致信息（"这个 token 大概在文本前 1/3"），高频分量编码精确的相对距离。

```
// RoPE 核心旋转循环（ds4.c 约 4574 行）
for (uint32_t i = 0; i < n_rot; i += 2) {
    const float c = cosf(theta) * mscale;
    const float s = sin_sign * sinf(theta) * mscale;
    const float x0 = tail[i + 0];
    const float x1 = tail[i + 1];
    tail[i + 0] = x0 * c - x1 * s;    // 二维旋转公式
    tail[i + 1] = x0 * s + x1 * c;
    theta_extrap *= theta_scale;        // θ 递减，产生不同频率
}
```

---

4. MoE 混合专家（Mixture of Experts）

核心思想

把一个大 FFN 拆成 256 个小专家网络，每次只激活 6 个 + 1 个共享专家。

路由过程

```
token 向量 → Router（线性层 + softmax）→ 256 个分数 → 取前 6 名
```

- 不同 token 激活不同专家，所有 256 个专家都会被用到
- 共享专家（所有 token 必经）兜底通用知识
- 每个专家独立计算，输出按权重加权求和

为什么是 **6** 个

实验验证的效率甜点。从 1 到 4 效果提升明显，4 到 6 仍有明显收益，6 之后边际收益迅速下降。再增加专家数量效果提升极小但计算成本线性增加。

"6 个不够"的疑虑

- 每个 token 通常只需少量专家，信息需求集中在特定领域
- 43 层 × 6 个专家 = 258 次专家调用，通过残差连接逐步积累所需知识
- 某层没选到的专家，可能在其他层被选中

```
// MoE Router 打分（ds4.c 约 5023 行）
matvec_any(logits, model, layer->ffn_gate_inp, x);  // x × W_router → 256个分数
for (int i = 0; i < DS4_N_EXPERT; i++)
    probs[i] = sqrtf(softplus_stable(logits[i]));

// Top-K 选择（ds4.c 约 5065 行）
topk_desc(selection, DS4_N_EXPERT, DS4_N_EXPERT_USED, selected);  // 256选6
```

---

5. LoRA 低秩适配

核心思想：不修改原始权重，在旁边加一个低秩的"补丁"。

```
output = X × W + X × A × B
              ↑ 冻结       ↑ 只训练这些
         (4096×4096)  (4096×1024) × (1024×4096)
```

- B 初始化为零，训练从原始模型行为的安全起点逐步调整
- 训练完成后可合并进原始权重（`W_new = W + A×B`），推理无额外开销
- 在 DeepSeek 中，LoRA 不是用于微调，而是模型架构本身的一部分，用于降低注意力投影的参数量

---

6. KV Cache

问题：逐 token 生成时，之前的 K/V 已算过但被丢弃，导致重复计算。

方案：缓存已算过的 K 和 V，每步只算新 token 的。

```
没有缓存：总计 1+2+3+...+n = n(n+1)/2 次 KV 计算
有缓存：   总计 n 次 KV 计算
```

生成 1000 个 token，从约 50 万次降到 1000 次。

MQA（1 组 KV 头）让 KV Cache 内存缩小 64 倍，同样的显存能处理更长的上下文。

```
// KV cache push（ds4.c 约 7273 行）
kv_cache_push_raw(cache, scratch->kv);  // 新 KV 追加到缓存
```

---

7. 滑动窗口注意力（**Sliding Window Attention**）

问题：标准注意力 $O(n^2)$，长文本计算量和内存都爆炸。

方案：每个 token 只看最近 128 个 token。

```
单层感受野：128
43 层层层传递：128 × 43 = 5504
```

远处信息通过残差连接在多层之间逐步传递，类似传话游戏——每个人只能和附近的人说话，但消息经过多次传递能传到很远。

---

8. 残差连接（**Residual Connection**）

核心：`输出 = 输入 + 处理结果`

每一层不是"重写"而是"批注"——在原始输入上追加新信息。

- 原始 embedding 信息经过 43 层后依然存在
- 梯度可以跳过所有层直接回流，解决深层网络的梯度消失问题
- 任何一层"什么都不做"都是安全的（output = input + 0）

```
// 残差保存（ds4.c 约 7242 行）
memcpy(scratch->attn_residual, inp_hc, ...);  // 保存输入用于残差
```

---

9. RMSNorm

问题：层数多了，数值不断放大最终溢出。

方案：每层把数值除以均方根，拉回合理范围。

```
// RMSNorm（ds4.c 约 2563 行）
ss = Σ x[i]²                // 平方和
scale = 1 / sqrt(ss/n + eps)  // 均方根的倒数
out[i] = x[i] * scale * weight[i]  // 缩放 + 可学习权重
```

比 BatchNorm 少了减均值这一步，更简单更快，效果相当。每层用两次：注意力前和 MoE 前。

---

10. SwiGLU 激活函数

MoE 专家内部的激活函数，替代经典的 ReLU。

```
// SwiGLU（ds4.c 约 4876 行）
output = silu(gate) × up    // silu(x) = x / (1 + e^(-x))
```

不像 ReLU 硬性截断，SwiGLU 是软性门控：gate 路决定"保留多少"，up 路提供"要保留的内容"，两者相乘得到过滤结果。

---

11. BPE 分词器（Byte Pair Encoding）

训练时统计相邻 token 对频率，不断合并最高频的对。推理时按学到的规则从长到短匹配切分。

- 高频组合合成一个 token（如"喜欢"），低频组合保持拆分
- 词表 129280 对中英文都有良好覆盖
- 更大的词表 → 中文编码效率高 → 同样的上下文长度装更多内容

---

12. 采样策略

模型输出 129280 个 logits，经过以下步骤选出最终 token：

Temperature

除以 temperature 缩放 logits 差距。低温（0.1）放大差距，输出确定；高温（2.0）缩小差距，输出随机。

Top-P（核采样）

按概率降序排列，累加直到总和达到 p，之后的 token 丢弃。自适应：模型确定时保留少量选项，不确定时保留更多。

Min-P

只保留概率 >= 最高概率 × min_p 的 token。砍掉和最优选差距太大的选项。

Argmax

直接选最高分，等价于 temperature=0，输出完全确定。

```
// 采样 `sample_full_vocab()`（ds4.c 约 14243 行）（轮盘赌采样）
float rnd = sample_rng_f32(rng) * sum;
for (int i = 0; i < DS4_N_VOCAB; i++) {
    rnd -= probs[i];
    if (rnd <= 0.0f) return i;
}
```

---

13. 量化（Quantization）

将高精度浮点数压缩成低精度整数，让大模型能在消费级硬件上运行。

格式	每权重位数	相对 FP16 大小	用途
FP16	16 bit	1.0x	重要权重
Q8_K	8 bit	0.5x	临时计算
Q4_K	4 bit	0.25x	主要权重
Q2_K	2 bit	0.125x	不太重要的权重

混合精度量化：重要权重（Embedding、Q/K/V 投影、Router）用高精度，不常激活的路由专家用极限压缩。

PRO 模型变体

DeepSeek V4 PRO 的参数与 Flash 有显著差异：

参数	Flash	PRO
总参数	284B	1.6T
激活参数	13B	49B
层数	43	相同架构，更大专家
专家数	256	更多路由专家
激活专家/token	6	更多

PRO 模型的 Q4 量化需要 512GB 内存（如 Mac Studio），推理时 KV cache 和 MoE 计算开销都更大。

混合 GGUF 拼接

`gguf-splice` 工具支持字节级替换——将一个 GGUF 文件中的路由专家张量替换为另一个 GGUF 文件中的对应张量，无需反量化。这允许混合不同量化级别或不同训练检查点的专家，创建定制的模型变体。

概念到代码的映射

```
概念                    关键函数                              代码位置
─────────────────────────────────────────────────────────────────────
Embedding 查表       `embed_token_f16()`                  ds4.c 约 2538 行
RoPE 旋转            `rope_tail_ext_inplace()`            ds4.c 约 4548 行
注意力计算           `layer_attention_rows_one()`          ds4.c 约 4751 行
MoE 路由             `layer_router_probs_one()`           ds4.c 约 5023 行
Top-K 选择           `topk_desc()`                        ds4.c 约 5065 行
单层前向            `layer_forward_raw_swa_one()`        ds4.c 约 7212 行
采样                 `sample_full_vocab()`                ds4.c 约 14243 行
```

单层完整流程

```
输入向量（来自上一层 + 残差）
    │
    ↓  RMSNorm（归一化，稳定数值）
    ↓  Q/K/V 投影（带 LoRA 低秩分解）
    ↓  RoPE 旋转（编码位置信息到 Q 和 K）
    ↓  KV Cache（缓存 K/V，避免重复计算）
    ↓  注意力计算（64头 MQA，滑动窗口）
    ↓  残差连接：input + attention_output
    │
    ↓  RMSNorm（再次归一化）
    ↓  MoE（Router 选 6 专家 + 1 共享专家）
    │       └── 每个专家内部：SwiGLU 门控激活
    ↓  残差连接：input + attention + moe
    │
    ↓  传给下一层（共 43 层）
```

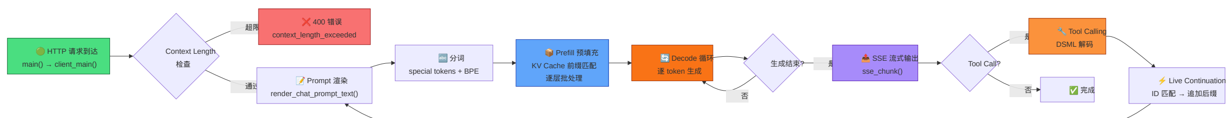
模型参数速查

```
DS4_N_EMBD          = 4096,    // Embedding 维度
DS4_N_VOCAB         = 129280,  // 词表大小
DS4_N_LAYER         = 43,      // Transformer 层数
DS4_N_HEAD          = 64,      // Q 注意力头数
DS4_N_HEAD_KV       = 1,       // K/V 头数（Multi-Query Attention）
DS4_N_HEAD_DIM      = 512,     // 每个头的维度
DS4_N_ROT           = 64,      // RoPE 旋转频率数
DS4_N_LORA_Q        = 1024,    // Query LoRA 秩
DS4_N_LORA_O        = 1024,    // Output LoRA 秩
DS4_N_EXPERT        = 256,     // MoE 专家总数
DS4_N_EXPERT_USED   = 6,       // 每 token 激活专家数
DS4_N_EXPERT_SHARED = 1,       // 共享专家数
DS4_N_FF_EXP        = 2048,    // 专家内部隐藏维度
DS4_N_SWA           = 128,     // 滑动窗口大小
```

端到端推理流程

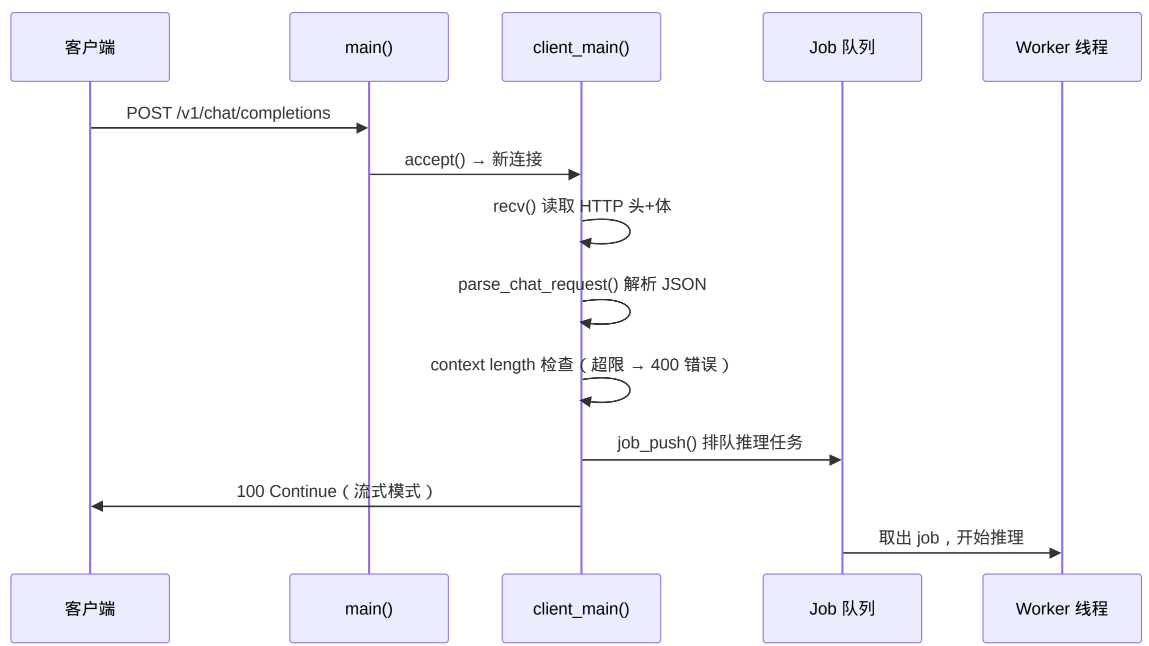
> 从一个 HTTP 请求到达服务器，到 SSE 流式输出最后一个 token，完整追踪一次推理请求的全链路。每个阶段标注对应的 学习笔记 以便深入阅读。

总览



阶段	核心函数	主要文件	Day
HTTP 接收	`main()` → `client_main()`	ds4_server.c	Day 12
Prompt 渲染	`render_chat_prompt_text()`	ds4_server.c	Day 10
分词	`ds4_tokenize_rendered_chat()`	ds4.c	Day 5
Prefill	`ds4_session_sync()` → `prefill_layer_major_cpu()`	ds4.c	Day 9
Decode 循环	`layer_forward_raw_swa_one()` × 43	ds4.c	Day 7 / Day 8
SSE 流式输出	`sse_chunk()`	ds4_server.c	Day 12
Tool Calling	`parse_function_call()`	ds4_server.c	Day 10

阶段一：HTTP 请求接收



入口：`main()`（ds4_server.c 约 8089 行）创建监听 socket，循环 `accept()` 接受连接。每连接启动一个线程运行 `client_main()`（ds4_server.c 约 7678 行）。

请求解析：`parse_chat_request()`（ds4_server.c 约 1984 行）从 JSON body 中提取：

- `messages` 数组 → 对话历史
- `temperature`、 `top_p` 等采样参数
- `stream: true` → 启用 SSE 流式
- `tools` 数组 → Tool Calling 定义

Context Length 检查：`request_exceeds_context()`（ds4_server.c 约 4680 行）在入队前检查 prompt tokens 是否 >= context size。超限时 `http_error_context_length_exceeded()` 返回 400 错误，包含 `n_prompt_tokens` 和 `n_ctx`，让客户端精确了解差距。此检查在入队之前执行，避免浪费 worker 线程时间。错误格式根据 API 类型自动适配（OpenAI 用 `code: "context_length_exceeded"`，Anthropic 用 `type: "error"`）。

```
// `parse_chat_request()` (ds4_server.c 约 1984 行)
static int parse_chat_request(json_t *root, struct ds4_chat_request *req) {
    req->messages = json_object_get(root, "messages");
    req->stream    = json_boolean_value(json_object_get(root, "stream"));
    req->temperature = json_real_value_or(json_object_get(root, "temperature"),
0.0);
    // ...
}
```

解析完成后构造 job 推入队列，Worker 线程（线程池）取出执行。

阶段二：Prompt 渲染与分词

Prompt 渲染

`render_chat_prompt_text()`（ds4_server.c 约 1901 行）将 `messages` 数组拼成模型期望的格式：

```
<｜begin▁of▁sentence｜><｜User｜>你好<｜Assistant｜>
```

关键步骤：

1. 插入 BOS token（`<｜begin▁of▁sentence｜>`）
2. 为每条消息包裹角色标记（`<｜User｜>` / `<｜Assistant｜>`）
3. 系统 prompt 插入最前
4. 最后追加 `<｜Assistant｜>` 引导模型开始生成

分词

`ds4_tokenize_rendered_chat()`（ds4.c 约 14716 行）将渲染后的文本切成 token ID 序列：

```
// `ds4_tokenize_rendered_chat()` (ds4.c 约 14716 行)
int ds4_tokenize_rendered_chat(struct ds4_context *ctx,
                                const char *text,
                                int32_t *tokens,
                                int32_t n_max_tokens) {
    // 1. BPE 分词：按学到的合并规则从长到短匹配
    // 2. Special token 优先匹配（<｜User｜> 等是单独 token）
    // 3. 输出 token ID 数组
}
```

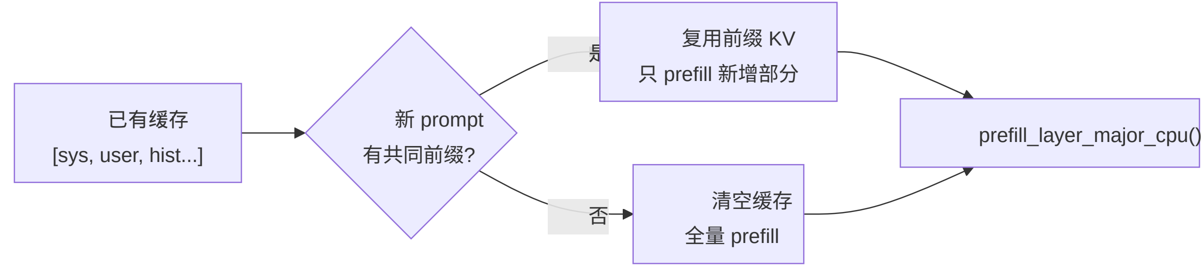
BPE 分词器按词表中 129280 个 token 进行最长匹配。高频组合（如"喜欢"）是单个 token，低频组合被拆成多个。详见 Day 5 — 采样策略 中的分词部分。

阶段三：**Prefill** 预填充

用户发来的是完整对话历史，模型需要一次性"消化"所有 context token。这就是 **Prefill** 阶段。

KV Cache 前缀匹配

`ds4_session_sync()`（ds4.c 约 17201 行）先检查 KV Cache 是否有可复用的前缀：



多轮对话中，前几轮的 KV 已缓存，只需 prefill 用户新输入。这大幅减少重复计算。

逐层批处理

`prefill_layer_major_cpu()`（ds4.c 约 7674 行）以层为主序处理所有 prefill token：

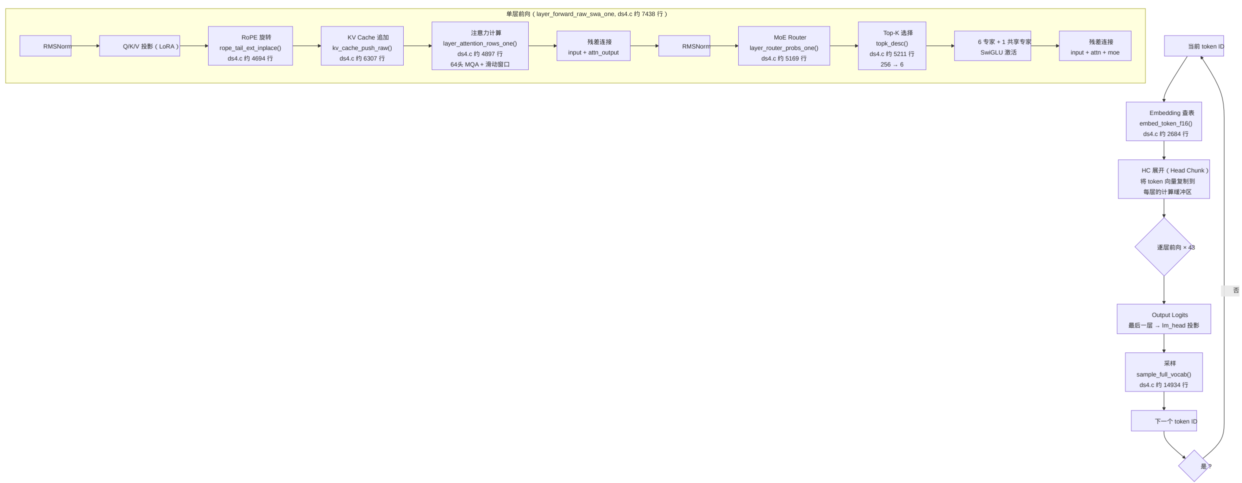
```
token 批次 [t0, t1, ..., tn]
    │
    ├─ Layer 0:  Embedding → RoPE → Attention → MoE → 残差
    ├─ Layer 1:  ...
    ├─ ...
    └─ Layer 42: ...
```

层为主序意味着同一层的权重在 cache 中只加载一次，对所有 token 做完计算再换下一层。这比"token 为主序"（每个 token 穿过所有层）的缓存友好度高得多。

Prefill 阶段结束后，所有 context token 的 K/V 都已写入 KV Cache，后续 decode 只需处理新生成的一个 token。

阶段四：**Decode** 循环

这是推理的核心——逐 token 自回归生成。每一步的完整流程：



4.1 Embedding + HC 展开

```
// `embed_token_f16()` (ds4.c 约 2684 行) — Embedding 查表
const uint16_t *row = base + (uint64_t)token * stride;
for (uint64_t i = 0; i < stride; i++)
    out[i] = f16_to_f32(row[i]);
```

Token ID 通过查表变成 4096 维向量。HC（Head Chunk）将这个向量复制到 43 层各自的计算缓冲区，避免层间数据依赖时的内存冲突。

4.2 单层前向

每层执行两次"归一化 → 计算 → 残差"：

```
# 第一次：注意力
x_norm = RMSNorm(x)
Q, K, V = x_norm @ W_qkv          # LoRA 低秩分解
Q, K    = RoPE(Q, K, position)     # 编码位置
KVCache.append(K, V)               # 缓存
attn    = Attention(Q, KVCache)    # 64 头 MQA, 窗口 128
x       = x + attn                 # 残差

# 第二次：MoE
x_norm = RMSNorm(x)
scores = x_norm @ W_router         # 256 个专家分数
top6   = TopK(scores, 6)           # 选 6 个
expert_out = Σ(top6_i(x_norm)) + shared(x_norm)  # 加权 + 共享
x       = x + expert_out           # 残差
```

43 层层层堆叠，每层在输入上"批注"新信息（残差连接保证原始信息不丢失）。

4.3 采样

最后一层输出经 `lm_head` 投影为 129280 个 logits，然后：

```
// `sample_full_vocab()` (ds4.c 约 14934 行) — 轮盘赌采样
// 1. Temperature 缩放
for (int i = 0; i < n; i++) logits[i] /= temperature;

// 2. Top-P 过滤（`sample_top_p_min_p()` ds4.c 约 15023 行）
//    按概率降序排列，累加到 p 后丢弃剩余

// 3. 采样
float rnd = sample_rng_f32(rng) * sum;
for (int i = 0; i < DS4_N_VOCAB; i++) {
    rnd -= probs[i];
    if (rnd <= 0.0f) return i;
}
```

> 采样策略详见 Day 5。Temperature 越低输出越确定（趋近 argmax），越高越随机。

阶段五：SSE 流式输出

每生成一个 token，`sse_chunk()`（ds4_server.c 约 3119 行）立即推送给客户端：

```
data: {"id":"chatcmpl-xxx","choices":[{"delta":{"content":"你"},"index":0}]}

data: {"id":"chatcmpl-xxx","choices":[{"delta":{"content":"好"},"index":0}]}

data: [DONE]
```

UTF-8 安全处理

中文字符占 3 个 UTF-8 字节，可能跨越多次 `recv`。`sse_chunk()` 内部维护 UTF-8 解码状态：

```
// `sse_chunk()` (ds4_server.c 约 3119 行)
// 如果 token 的 UTF-8 编码不完整（字节被截断）
// 缓存已接收字节，等下一个 token 到来后拼完整再输出
```

> 这保证客户端收到的每个 SSE chunk 都是合法的 UTF-8 字符串，不会出现乱码。
>
> SSE 格式兼容 OpenAI API 兼容，任何支持 OpenAI SDK 的客户端都能直接对接。

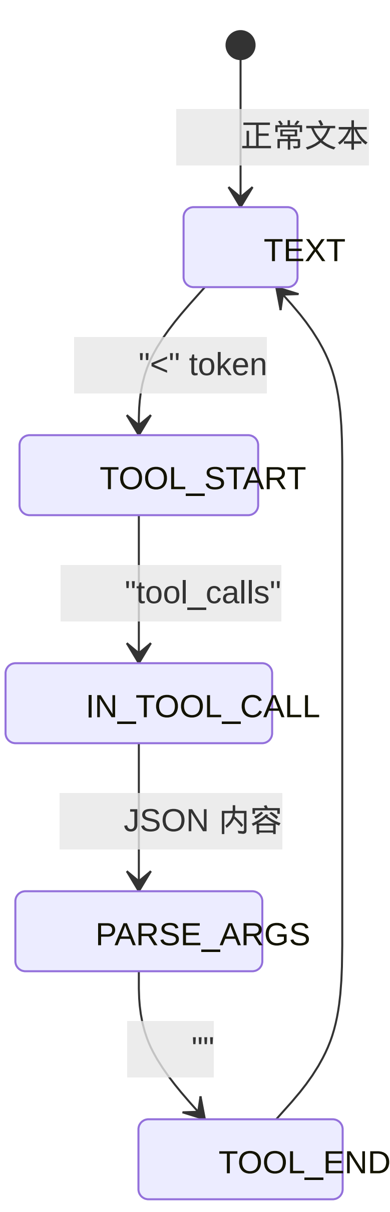
阶段六：**Tool Calling**

当模型决定调用工具时，生成特殊格式的 token 序列：

```
<tool_calls>
{"name": "get_weather", "arguments": {"city": "北京"}}
</tool_calls>
```

DSML 解码状态机

模型输出的 token 序列需要经过 DSML（DeepSeek Markup Language）解码才能还原为结构化数据：



- `parse_function_call()`（ds4_server.c 约 966 行）解析 `<tool_calls>` 中的 JSON

- `parse_tool_calls_value()`（ds4_server.c 约 1014 行）处理参数数组
- `append_dsml_text_escaped()`（ds4_server.c 约 1768 行）处理 DSML 转义

温度覆盖

Tool Calling 模式下，`temperature` 会被覆盖为 0（argmax），确保工具调用的格式稳定不走样。模型需要在精确的结构化输出和自由文本生成之间切换。

阶段七：**Live Tool Continuation**（工具调用快速续接）

> 上游新增（2025-05-15）：工具调用后不再重建整个上下文，而是直接在 live KV 状态上追加后缀。详见 `misc/RESPONSE_API.md` 和 `misc/ANTHROPIC_LIVE_CONTINUATION.md`。

问题

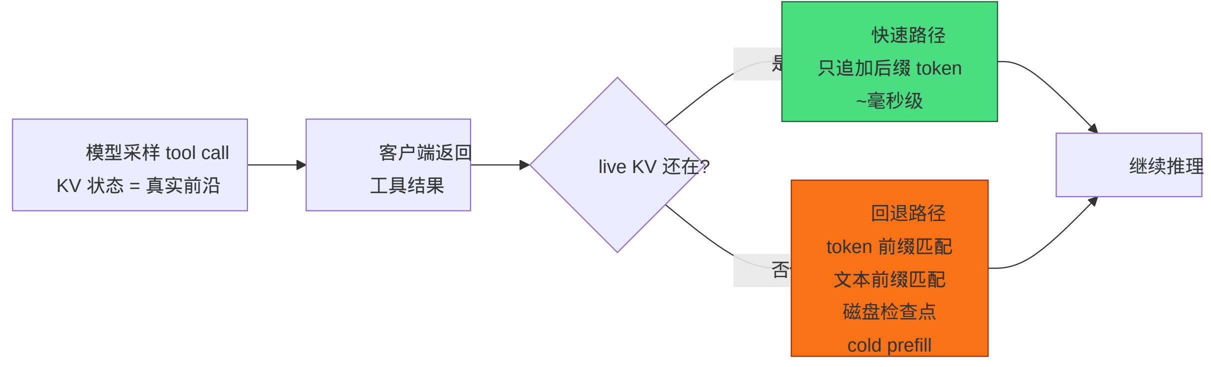
工具调用前的实现：模型采样完工具调用后，服务器尝试"规范化"（canonicalize）live KV 检查点，让它与下一请求的完整 prompt 精确匹配。这导致：

```
tool checkpoint canonicalization needs rebuild ... common=16846 live=16994 canon
ical=16937
tool checkpoint canonicalized ... via=rebuild
```

百万 token 上下文下，这个重建就是一次近乎完整的 prefill，耗时 10 秒以上。

核心思想

采样的 **KV** 状态是最高保真度的状态。 客户端可见的协议对象只应"选择"这个状态，而不是强迫服务器重建它。



两种协议的绑定方式

协议	绑定 ID	端点	用途
OpenAI Responses	`call_id`（function_call_output）	`/v1/responses`	Codex CLI 等 Agent 工具
Anthropic Messages	`tool_use_id`（tool_result）	`/v1/messages`	Claude Code 等

两种协议共享同一套 live KV 续接逻辑：ID 匹配 → 跳过 prefill → 只 tokenize 后缀。

后缀 Token 序列

快速路径只 tokenize 并追加极少 token：

```
<｜end_of_sentence｜><｜User｜><tool_result>...<｜Assistant｜><think-or-/think>
```

避免了整个上下文的重建。

函数索引表

阶段	函数	文件	位置
HTTP 入口	`main()`	ds4_server.c	约 8089 行
连接处理	`client_main()`	ds4_server.c	约 7678 行
请求解析	`parse_chat_request()`	ds4_server.c	约 1984 行
Context 检查	`request_exceeds_context()`	ds4_server.c	约 4680 行
Context 错误	`http_error_context_length_exceeded()`	ds4_server.c	约 4686 行
Prompt 渲染	`render_chat_prompt_text()`	ds4_server.c	约 1901 行
分词	`ds4_tokenize_rendered_chat()`	ds4.c	约 14716 行
Session 同步	`ds4_session_sync()`	ds4.c	约 17201 行
Prefill	`prefill_layer_major_cpu()`	ds4.c	约 7674 行
Embedding	`embed_token_f16()`	ds4.c	约 2684 行
RoPE	`rope_tail_ext_inplace()`	ds4.c	约 4694 行
KV Cache	`kv_cache_push_raw()`	ds4.c	约 6307 行
注意力	`layer_attention_rows_one()`	ds4.c	约 4897 行
MoE Router	`layer_router_probs_one()`	ds4.c	约 5169 行
Top-K	`topk_desc()`	ds4.c	约 5211 行
单层前向	`layer_forward_raw_swa_one()`	ds4.c	约 7438 行
采样	`sample_full_vocab()`	ds4.c	约 14934 行
Top-P/Min-P	`sample_top_p_min_p()`	ds4.c	约 15023 行
Argmax	`sample_argmax()`	ds4.c	约 14874 行
SSE 输出	`sse_chunk()`	ds4_server.c	约 3119 行
Tool 解析	`parse_function_call()`	ds4_server.c	约 966 行
Tool 参数	`parse_tool_calls_value()`	ds4_server.c	约 1014 行
DSML 转义	`append_dsml_text_escaped()`	ds4_server.c	约 1768 行
DSML 词表	`dsml_id` 初始化	ds4.c	约 14621 行

出链

Part 1: 构建与加载

/topics/05-serving/concepts\

/topics/04-generation-pipeline/concepts\

/topics/02-tokenization-sampling/concepts\

/topics/03-model-architecture/concepts\

Part 4: 推理流程

Part 2: 分词与采样

Part 5: 服务层

Part 1: 构建与加载

> 从 Makefile 编译系统到 mmap 零拷贝模型加载，理解项目如何从源码变为可执行文件、81GB 模型如何"不加载"地装进内存。

涵盖内容

章节	核心主题
1. 项目全貌与编译系统	Makefile、条件编译、项目架构、opaque type
2. 内存管理与 mmap	mmap 零拷贝、xmalloc、scratch buffer、page fault
3. GGUF 二进制格式	GGUF 文件结构、cursor 解析、struct 对齐、量化块

核心概念

- gguf — 模型文件格式
- mmap — 零拷贝文件映射
- xmalloc — 失败即退出的内存分配策略
- page-fault — mmap 懒加载的核心机制

前置知识

- C 语言基础（指针、结构体、宏）
- 基本的编译概念（.c → .o → 可执行文件）

学习路径

读完本主题后，你将理解：

1. ds4 项目如何编译、有哪些构建目标
2. 为什么推理引擎在初始化阶段完成所有内存分配
3. 模型文件的二进制结构和解析方式

→ 下一步：Part 2: 分词与采样

Part 1: 构建与加载

> 从 Makefile 到 GGUF 二进制格式，理解项目如何编译、模型如何加载到内存。

---

1. 项目全貌与编译系统

> 看不懂代码就先跑起来。今天从编译系统入手，理解项目怎么从 .c 变成可执行文件，再俯瞰整体架构——后续 14 天的所有代码走读都建立在这个全景之上。

C 知识点

1. Makefile 基础

Makefile 是一个构建脚本，告诉 `make` 工具如何把源文件编译成可执行文件。

核心语法：

```
target: prerequisites
    recipe          # 注意：必须用 Tab 缩进，不能用空格
```

ds4 的 Makefile 展示了几个重要概念：

变量赋值（`?=` 和 `:=`）

```
CC ?= cc                                   # ?= 表示如果未定义才赋值（用户可覆盖）
CFLAGS ?= -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99
UNAME_S := $(shell uname -s)               # := 立即执行 shell 命令
```

- `?=`（条件赋值）：只有变量未定义时才设置，用户可以在命令行覆盖：`make CC=clang`
- `:=`（立即展开）：定义时立即求值，不是使用时才求值
- `=`（延迟展开）：使用时才求值（本项目没用到）

条件编译（平台检测）

```
ifeq ($(UNAME_S),Darwin)
    # macOS：链接 Metal 框架
    METAL_LDLIBS := $(LDLIBS) -framework Foundation -framework Metal
    CORE_OBJS = ds4.o ds4_metal.o
else
    # Linux：禁用 Metal，纯 CPU
    CFLAGS += -DDS4_NO_METAL
    CORE_OBJS = ds4.o
endif
```

伪目标（**.PHONY**）

```
.PHONY: all clean test
```

声明这些目标不对应实际文件，即使存在同名文件也要执行。

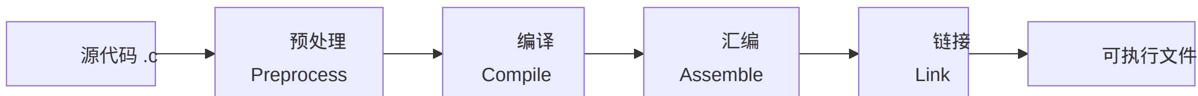
自动变量

```
ds4.o: ds4.c ds4.h ds4_metal.h
    $(CC) $(CFLAGS) -c -o $@ ds4.c
```

- `$@` = 当前目标名（这里是 `ds4.o`）
- `$^` = 所有依赖（这里是 `ds4_cli.o linenoise.o ds4.o ds4_metal.o`）
- `$<` = 第一个依赖

2. 编译流程

从 .c 文件到可执行文件分四步：



在 Makefile 中，这些步骤被拆成两条规则：

编译（**.c → .o**）

```
ds4.o: ds4.c ds4.h ds4_metal.h
    $(CC) $(CFLAGS) -c -o $@ ds4.c
```

`-c` 标志告诉编译器只编译不链接，生成目标文件（.o）。

链接（**.o →** 可执行文件）

```
ds4: ds4_cli.o linenoise.o $(CORE_OBJS)
    $(CC) $(CFLAGS) -o $@ ds4_cli.o linenoise.o $(CORE_OBJS) $(METAL_LDLIBS)
```

把多个 .o 文件和库合并成最终的可执行文件。

编译选项解读：

标志	含义
`-O3`	最高级别优化
`-ffast-math`	放宽浮点精度要求以换取速度（不严格遵守 IEEE 754）
`-mcpu=native`	针对当前 CPU 生成最优指令
`-Wall -Wextra`	开启所有常见警告
`-std=c99`	使用 C99 标准
`-lm`	链接数学库（sin, cos, sqrt 等）
`-pthread`	链接 POSIX 线程库
`-framework Metal`	链接 Apple Metal GPU 框架（仅 macOS）

3. 条件编译（**#ifdef**）

在 C 代码中，用预处理指令控制哪些代码参与编译：

```
#ifndef DS4_NO_METAL
#include "ds4_metal.h"
#endif

#if defined(__ARM_NEON)
#include <arm_neon.h>
#endif
```

- `#ifndef` = "if not defined"：如果没有定义 `DS4_NO_METAL` 宏，就包含 Metal 头文件
- Makefile 中通过 `-DDS4_NO_METAL` 定义这个宏（`-D` = define）
- 这样同一份代码可以在有/无 GPU 的平台上编译

4. 头文件包含约定

```
#include <stdio.h>      // 尖括号：系统头文件，从系统路径搜索
#include "ds4.h"        // 引号：项目头文件，先从当前目录搜索
```

ds4.c 包含了 18 个系统头文件和 2 个项目头文件，覆盖了：

- 文件操作：`<fcntl.h>`, `<unistd.h>`, `<sys/stat.h>`
- 内存映射：`<sys/mman.h>`

- 线程：`<pthread.h>`
- 类型定义：`<stdint.h>`, `<stdbool.h>`, `<stddef.h>`

LLM 知识点

1. 推理引擎是什么

大语言模型（LLM）的训练和推理是两个完全不同的过程：

- 训练：用海量数据更新模型参数，需要巨大的 GPU 集群，花费数周甚至数月
- 推理：用训练好的模型生成文本，单台机器就能运行

  推理引擎就是专门做推理的软件。它的工作是：

1. 从磁盘加载模型文件（包含所有训练好的参数）
2. 接收用户输入（文本），转换成数字序列（token IDs）
3. 用矩阵运算计算"下一个最可能的词"
4. 重复步骤 3，直到生成完整回答

2. ds4.c 的定位

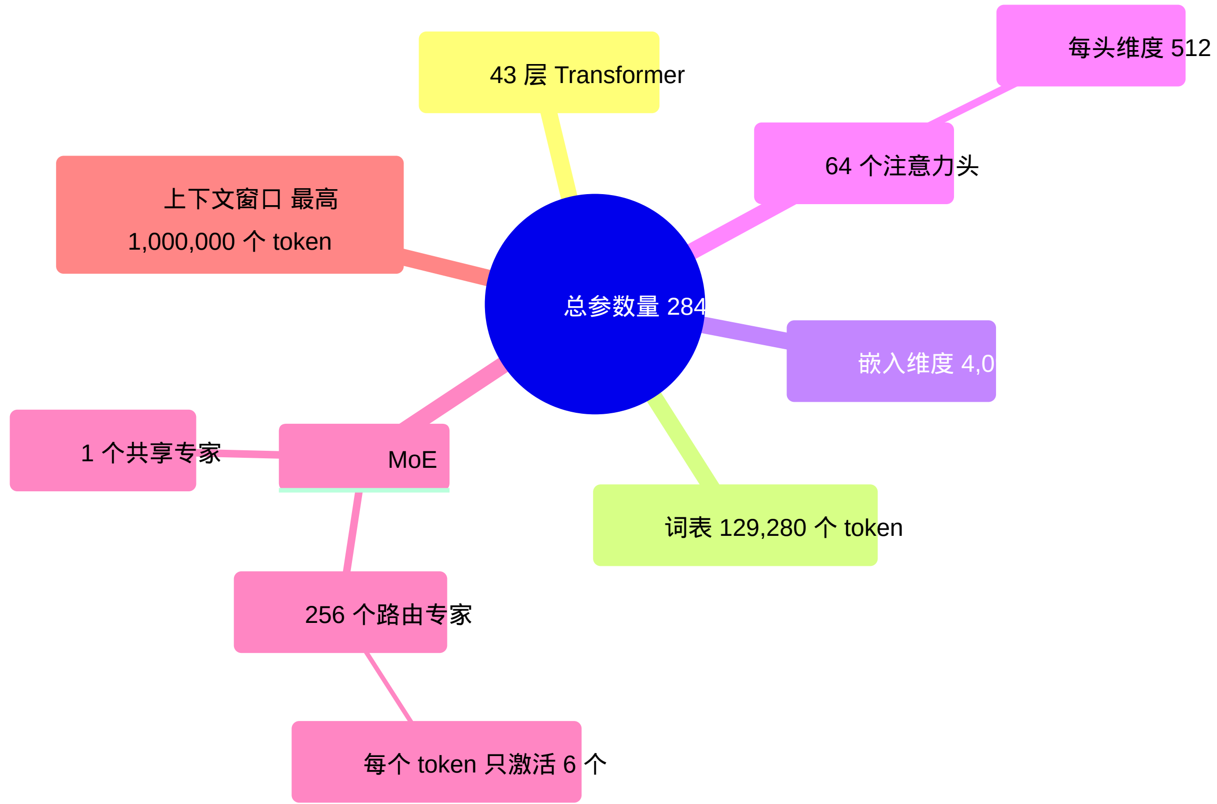
ds4.c 是一个专用推理引擎，只为 DeepSeek V4 模型家族服务（Flash 和 PRO 两个变体）。

  它的设计哲学是"垂直整合"（deliberately vertical）：

- 不依赖外部框架：没有 PyTorch、没有 TensorFlow，JSON 解析、HTTP 服务器、分词器全部手写
- 固定模型架构：所有模型参数（层数、维度、专家数等）都硬编码为常量，不做通用 GGUF 加载器
- **Metal** 优先：专门为 Apple Silicon GPU 优化，CPU 路径仅用于正确性验证
- 模块化扩展：新增 `ds4_agent.c`（自主编码代理）、`ds4_web.c`（浏览器工具）、`ds4_kvstore.c`（模块化 KV 存储）三大模块

3. 模型参数概览

从 ds4.c 的常量定义中，可以看到 DeepSeek V4 Flash 的架构（详细模型信息参见 `MODEL_CARD.md`）：



参数之间的数学关系：

1. 注意力头的维度：64 头 × 512 维/头 = 32,768（多头拼接后的总维度）。嵌入维度 4,096 是输入/输出的隐藏维度，通过投影矩阵连接——QKV 投影将 4,096 映射到 32,768，输出投影再从 32,768 映射回 4,096。
2. 嵌入层参数量：129,280（词表）× 4,096（嵌入维度）≈ 530M 参数。每个 token 对应一个 4,096 维的向量。
3. **MoE** 激活比例：6 / 256 ≈ 2.3%。每次推理只激活 256 个路由专家中的 6 个（加 1 个共享专家 = 7 个），284B 总参数中实际参与计算的只有一小部分。
4. **284B** 总参数的大致构成：嵌入矩阵 ~530M + 43 层 ×（每层注意力参数 + MoE 专家参数）。MoE 专家是参数量的主要来源：256 个专家 × 每个专家的 FFN 权重。
5. 上下文窗口与 **KV** 缓存：1,000,000 tokens × 层数 × 头维度，决定了推理时的显存需求。

关键洞察：虽然模型有 284B 参数，但每次推理只激活约 6/256 的 MoE 专家参数，所以实际计算量远小于同等规模的密集模型。

`MODEL_CARD.md` 包含从 DeepSeek 官方模型卡提取的关键信息，包括 DeepSeek-V4 家族两个模型的对比：

模型	总参数	激活参数	上下文长度
DeepSeek-V4-Flash	284B	13B	1M tokens
DeepSeek-V4-Pro	1.6T	49B	1M tokens

架构上使用压缩稀疏注意力（CSA）+ 重度压缩注意力（HCA）：前两层只用原始滑动窗口（128 token），之后偶数层按 ratio-4 压缩（带 indexer 选择最多 512 行），奇数层按 ratio-128 压缩。这是百万 token 上下文不需要全量 KV cache 的关键。

4. GGUF 格式简介

GGUF（GPT-Generated Unified Format）是 llama.cpp 生态使用的模型文件格式：

- 二进制格式，紧凑高效
- 包含模型参数（张量/tensor）和元数据（模型名称、架构参数等）
- 支持多种量化级别（2-bit、4-bit 等），用更少的位存储参数以节省内存

ds4.c 支持：

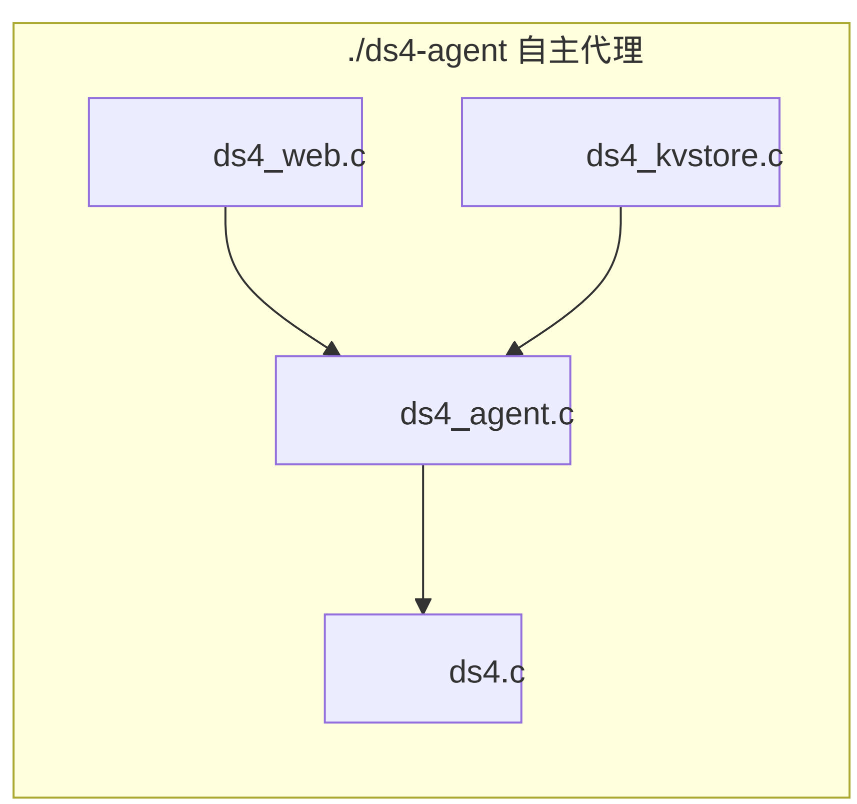
- **Q2（2-bit 量化）**：128GB 内存可运行，模型文件约 81GB
- **Q4（4-bit 量化）**：需要 256GB+ 内存，精度更高

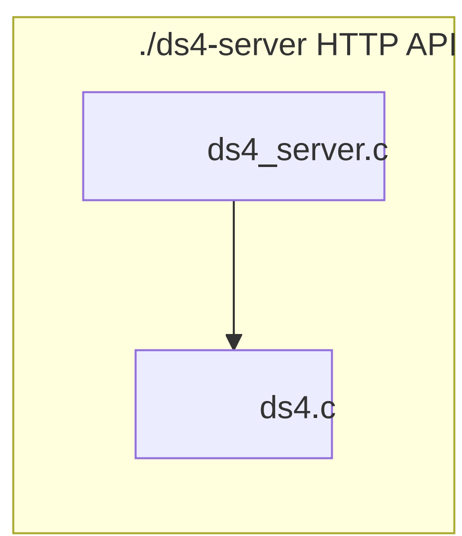
`download_model.sh` 现在支持 `pro` 和 `pro-imatrix` 目标，可下载 DeepSeek V4 PRO（1.6T 总参数）的量化模型。PRO 模型需要 512GB 内存的 Mac Studio。

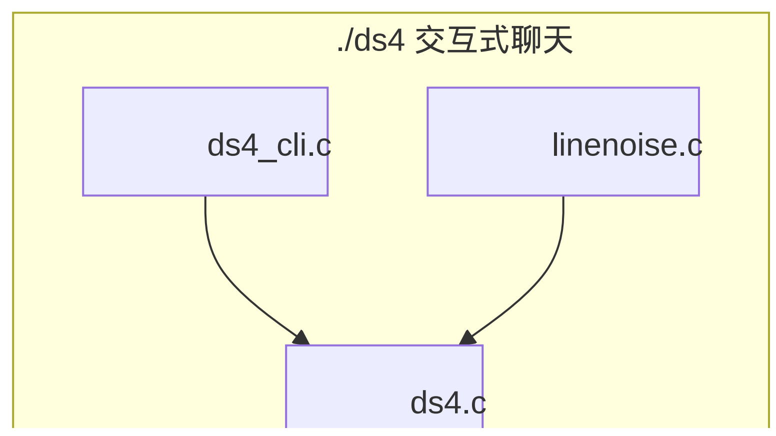
5. 项目的可执行文件

二进制	用途	对应源文件
`./ds4`	交互式命令行聊天	ds4_cli.c + linenoise.c
`./ds4-server`	HTTP API 服务器	ds4_server.c
`./ds4-bench`	吞吐量基准测试	ds4_bench.c
`./ds4-eval`	能力评估（93 道题回归测试）	ds4_eval.c
`./ds4-agent`	自主编码代理（9607 行）	ds4_agent.c + ds4_web.c

`make` 默认编译 `ds4`、`ds4-server`、`ds4-bench`、`ds4-eval`、`ds4-agent`。


./ds4-agent 自主代理
ds4_web.c
ds4_kvstore.c
ds4_agent.c
ds4.c


./ds4-server HTTP API
ds4_server.c
ds4.c


./ds4 交互式聊天
ds4_cli.c
linenoise.c
ds4.c

`./ds4-server` 提供与 OpenAI/Anthropic 兼容的 API，可以被编程工具（如 Claude Code）直接调用。

6. 发布前 **QA** 检查清单（**QA_BEFORE_RELEASES.md**）

`make test` 只覆盖核心回归；发布前还有一份手动 **QA** 清单 `QA_BEFORE_RELEASES.md`（14 节），把每个子系统该验什么列成 checklist：

章节	验证范围
1-2	仓库与构建完整性、核心回归测试
3-4	Metal Flash 路径、Metal PRO 路径
5	SSD Streaming
6-7	CUDA / DGX Spark、ROCm / Strix Halo
8-9	分布式推理、磁盘 KV Cache
10-11	Server APIs、ds4-agent
12-14	下载脚本与模型文件、性能与功耗、发布签字

这份清单的存在本身就说明：推理引擎的"正确"是跨后端、跨模型变体的——同一份代码要在 Metal Flash、CUDA PRO、ROCm Strix Halo、分布式、SSD streaming 这些组合上都跑对，单元测试覆盖不到的组合靠发布前的人工 checklist 兜底。

---

2. 内存管理与 mmap

> 81GB 的模型文件怎么装进内存？答案是——根本不装。今天学习 mmap 零拷贝加载、内存分配的错误处理策略，以及为什么推理引擎必须在初始化阶段完成所有内存分配。

设计决策推导：81GB 模型的内存策略

```
问题 1：81GB 模型文件，直接 fread 读进内存？
  └─ 不行：需要 81GB 连续物理内存，启动慢（磁盘 I/O）
  └─ 方案：mmap — 让操作系统按需加载页面，启动瞬间完成（只建映射）
     首次访问某页时触发 page fault，OS 从磁盘读取该页

问题 2：mmap 的内存可以被 GPU 直接使用吗？
  └─ MAP_SHARED 映射的内存可以被 Metal 包装为"零拷贝 MTLBuffer"
     GPU 直接从文件映射读取权重，不需要额外的显存拷贝
     → 这就是为什么 Makefile 按平台分支编译

问题 3：推理过程中需要各种临时缓冲区（KV cache、注意力分数等），
        能不能在热循环中 malloc/free？
  └─ 绝对不行！malloc 是非确定性操作（可能触发系统调用、锁竞争）
  └─ 方案：预分配 scratch buffer — 启动时一次性分配所有工作内存，
     推理循环中只做指针操作，零 malloc
     → 这就是为什么没有 xfree

问题 4：calloc 天然初始化为零，为什么用 malloc + memset？
  └─ calloc 的"共享零页"优化让所有虚拟页指向同一个物理零页
     推理时逐 token 写入触发大量 page fault
     macOS 上大 mmap + 大量 page fault 可能导致 kernel panic
  └─ 方案：malloc + memset 在启动时集中触发所有 page fault
     把内存初始化的代价集中到可控的启动阶段
```

C 知识点

1. malloc/free 与错误处理

C 语言中动态内存分配的基本 API：

```c
void *malloc(size_t size);      // 分配 size 字节，不初始化
void *calloc(size_t n, size_t size);  // 分配 n×size 字节，初始化为零
void *realloc(void *ptr, size_t size); // 调整已分配内存大小
void free(void *ptr);             // 释放内存
```

> 关键规则：分配可能失败（返回 NULL），必须检查。
>
> ds4.c 用包装函数统一处理错误：

```c
static void *xmalloc(size_t size) {
    void *p = malloc(size);
    if (!p) ds4_die("out of memory");  // 失败就直接退出
    return p;
}
```

这种模式叫 **x-包装器**（x-wrapper）：`x` 前缀表示"失败时退出"。好处：

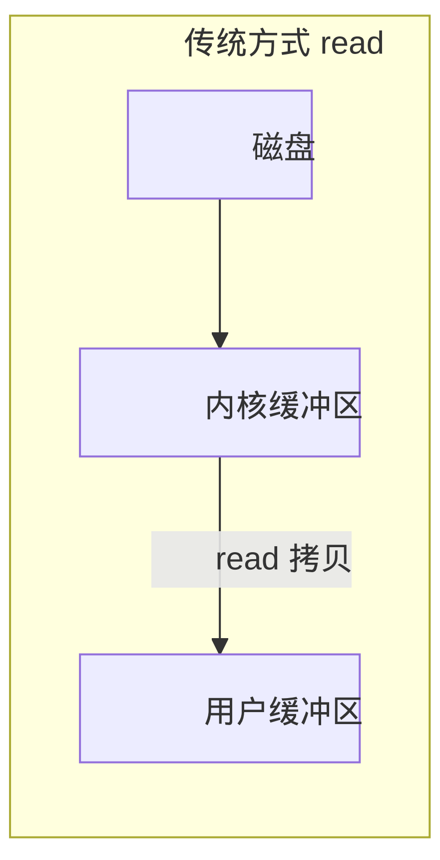
- 每次调用不用重复写 NULL 检查
- 错误处理策略统一
- 代码更简洁

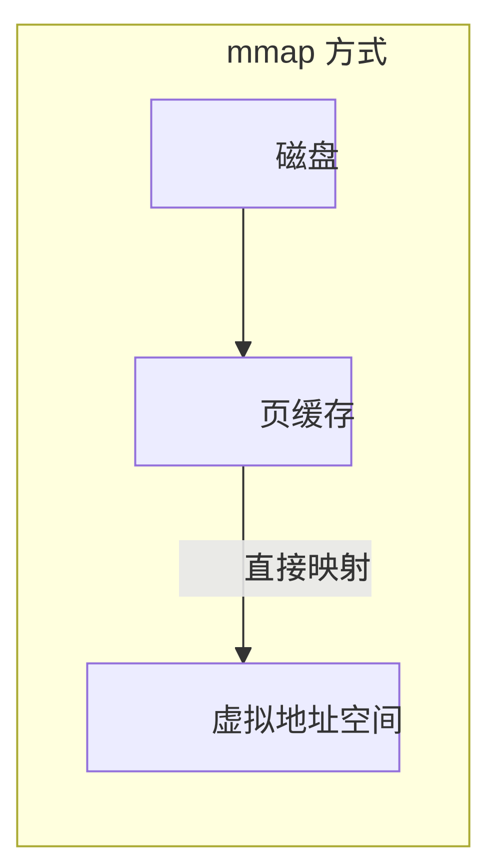
2. mmap — 零拷贝文件映射

`mmap`（memory map）把文件内容直接映射到进程的虚拟地址空间：

```c
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offse
t);
int munmap(void *addr, size_t length);
```

工作原理：





mmap 的优势：

- 零拷贝：没有内核空间到用户空间的数据拷贝
- 懒加载：只有实际访问某页时才从磁盘读取（page fault）
- 统一接口：文件内容当数组用，`m->map[offset]` 直接访问

ds4.c 中的 mmap 调用（`model_open()` 约 1174-1219 行）：

```
static void model_open(ds4_model *m, const char *path, bool metal_mapping) {
    int fd = open(path, O_RDONLY);          // 打开文件
    struct stat st;
    fstat(fd, &st);                          // 获取文件大小

    // Metal 用 MAP_SHARED（Metal 可直接作为 GPU 缓冲区）
    // CPU 用 MAP_PRIVATE（避免 macOS 的 VM 内核 bug）
    const int mmap_flags = metal_mapping ? MAP_SHARED : MAP_PRIVATE;
    void *map = mmap(NULL, st.st_size, PROT_READ, mmap_flags, fd, 0);

    // 之后就可以用 map 指针直接访问文件内容
    m->map = map;
    m->size = st.st_size;
}
```

MAP_SHARED vs MAP_PRIVATE：

`MAP_SHARED`

- 多个进程共享同一映射
- 修改对其他进程可见
- Metal GPU 可以直接引用这块内存（零拷贝 GPU 缓冲区）

`MAP_PRIVATE`

- 写时复制（copy-on-write）
- 不影响原文件
- ds4.c 在 CPU 路径使用它，是为了规避 macOS 内核的 VM 计数 bug

3. xmalloc_zeroed — 为什么不用 calloc？

ds4.c `xmalloc_zeroed()`（约 482-498 行）有一个特殊的函数：

```
static void *xmalloc_zeroed(size_t n, size_t size) {
    const size_t total = n * size;
    void *p = xmalloc(total ? total : 1);
    memset(p, 0, total);  // 手动清零，而不是用 calloc
    return p;
}
```

为什么不用 `calloc`？注释解释了原因：

> 大块未触碰的 calloc 内存可能被虚拟内存系统通过共享零页来管理。CPU 解码的 KV cache 每次增长一个 token，所以用 calloc 会把数千次首次触碰缺页移到生成阶段。在 Darwin 上我们观察到这会导致内核 `cpt_mapcnt_inc` 溢出 panic。

简单说：

- `calloc` 返回的"零"内存可能共享一个物理零页（虚拟内存优化）
- 第一次写入时触发 page fault，需要分配真实物理页
- 对于巨大的 KV cache，大量同时 page fault 可能让 macOS 内核崩溃
- `malloc + memset` 提前触发所有 page fault，把风险控制在初始化阶段

4. posix_madvise — 内存访问建议

```
int posix_madvise(void *addr, size_t len, int advice);
```

ds4.c 用 `POSIX_MADV_WILLNEED` 告诉内核"这块内存马上要用"：

```
posix_madvise(m->map, m->size, POSIX_MADV_WILLNEED);
```

这不是强制操作，只是给内核一个提示，让内核提前把页面从磁盘读入缓存。

5. __thread — 线程局部存储

```
static __thread int g_parallel_depth;  // `g_parallel_depth` 定义处（约 613 行）
```

`__thread` 关键字使每个线程拥有变量的独立副本：

- 线程 A 的 `g_parallel_depth` 和线程 B 的 `g_parallel_depth` 是不同的变量
- 不需要 mutex 保护，因为每个线程只访问自己的副本
- 线程创建时初始化为零

在 ds4.c 中，它跟踪当前线程在 CPU 并行工作池中的递归深度，防止递归爆炸。

6. 预分配缓冲区模式（Scratch Buffer）

ds4.c 不在热循环中分配内存。它在推理开始前一次性分配所有需要的缓冲区：

```c
typedef struct {
    float *plain;      // 50+ 个预分配缓冲区
    float *cur;
    float *next;
    float *q;
    float *qr;
    // ... 每个对应推理中的一个计算步骤
} ds4_cpu_decode_scratch;
```

初始化时一次性分配：

```c
static void cpu_decode_scratch_init(ds4_cpu_decode_scratch *s, uint32_t ctx_siz
e) {
    s->plain = xmalloc(DS4_N_EMBD * sizeof(float));
    s->cur = xmalloc(hc_dim * sizeof(float));
    // ... 50 次 xmalloc ...
}
```

释放时一次性全部释放：

```c
static void cpu_decode_scratch_free(ds4_cpu_decode_scratch *s) {
    free(s->output_norm);
    free(s->output_embd);
    // ... 50 次 free ...
}
```

还有一个分配守卫确保没有意外的热循环分配：

```c
ds4_alloc_guard_begin("CPU token generation");
for (int i = 0; i < n_predict; i++) {
    // 如果这里有 malloc 调用，程序会立即退出并报错
}
ds4_alloc_guard_end();
```

7. 整数溢出保护

```c
static void *xmalloc_zeroed(size_t n, size_t size) {
    if (size != 0 && n > SIZE_MAX / size) ds4_die("allocation size overflow");
    // ...
}
```

当 `n * size` 超过 `SIZE_MAX`（size_t 能表示的最大值）时，乘法会回绕，导致分配过小的缓冲区。这是经典的安全漏洞（CWE-190）。检查 `n > SIZE_MAX / size` 可以防止这种情况。

LLM 知识点

1. 模型文件加载策略

一个 DeepSeek V4 Flash 的 2-bit 量化模型约 81GB。如何高效加载它？

传统方式（不适合）：

- `fread` 把整个文件读入内存 → 81GB 拷贝，启动很慢
- 只需要部分数据时也要全部读取

mmap 方式（ds4.c 的选择）：

- `mmap` 立即返回，没有数据拷贝
- 实际访问某页时才触发 page fault，从磁盘读取
- Metal GPU 可以零拷贝引用映射区域

懒加载的好处：

- 启动速度快：`mmap` 调用几乎是瞬间完成
- 内存按需使用：只有推理用到的权重页面会被加载
- 共享缓存：操作系统页缓存可以被多个进程共享

2. 权重预热（**Weight Warmup**）

可选的预热步骤，在推理前主动触碰所有权重页面：

```
static void model_warm_weights(const ds4_model *m) {
    const uint64_t page = sysconf(_SC_PAGESIZE);
    for (uint64_t off = start; off < end; off += page) {
        checksum += p[off];  // 每页读一个字节，强制 page fault
    }
}
```

为什么要预热？

- 推理时首次访问某权重页面会触发 page fault → 暂停几十毫秒
- MoE 模型中，不同 token 路由到不同专家，page fault 时机不可预测
- 预热把所有 page fault 集中到启动阶段，推理时就不会卡顿

3. KV Cache 的内存布局

KV Cache 存储注意力机制的历史状态，是推理中最大的内存消耗之一。

每个 KV cache 行的大小 = `DS4_N_HEAD_DIM × sizeof(float)` = 512 × 4 = 2048 字节。

DeepSeek V4 Flash 的 KV cache 非常紧凑（这是选择它的重要原因之一），使得百万级上下文在本地成为可能。

4. 零拷贝张量访问

```
static const void *tensor_data(const ds4_model *m, const ds4_tensor *t) {
    return m->map + t->abs_offset;  // 直接返回 mmap 中的偏移
}
```

这行代码是整个内存策略的核心：张量数据永远不离开 mmap 区域。没有拷贝，没有额外的缓冲区。推理引擎的指针直接指向文件映射中的位置。

总结：**ds4.c** 的内存策略

核心理念：在热路径上零分配，把所有内存操作集中到初始化阶段。

---

3. GGUF 二进制格式

> 模型以 GGUF 二进制文件分发。不理解文件格式，就不知道 81GB 里装了什么、怎么找到每一层权重。今天学习二进制解析、struct 对齐和"尽早失败"的验证策略。

C 知识点

1. struct 内存布局与对齐

C 编译器会在结构体成员之间插入"填充字节"（padding），确保每个成员正确对齐：

```c
// 量化块结构体 `block_q2_K`（约 126-151 行）
typedef struct {
    uint8_t  scales[QK_K / 16];   // 16 字节，偏移 0
    uint8_t  qs[QK_K / 4];        // 64 字节，偏移 16
    uint16_t d;                    // 2 字节，偏移 80
    uint16_t dmin;                 // 2 字节，偏移 82
} block_q2_K;                      // 总计 84 字节
```

对齐规则：

- `uint8_t`：1 字节对齐（任意地址）
- `uint16_t`：2 字节对齐（偶数地址）
- `uint32_t`：4 字节对齐（4 的倍数地址）
- `float`：4 字节对齐
- `uint64_t`：8 字节对齐

在 `block_q2_K` 中，字段顺序经过精心安排，没有浪费填充空间。84 字节恰好被 `static_assert` 验证：

```c
DS4_STATIC_ASSERT(ds4_block_q2_k_size, sizeof(block_q2_K) == 84);
```

2. static_assert — 编译时断言

ds4.c 用一个巧妙的技巧实现编译时断言 `DS4_STATIC_ASSERT`（约 153 行）：

```c
#define DS4_STATIC_ASSERT(name, cond) typedef char name[(cond) ? 1 : -1]
```

原理：

- 如果 `cond` 为真，声明 `typedef char name[1]` — 合法
- 如果 `cond` 为假，声明 `typedef char name[-1]` — 非法，编译报错

这是 C99 中的 portable static assert。C11 引入了标准的 `_Static_assert`，但 ds4.c 用 `-std=c99` 编译。

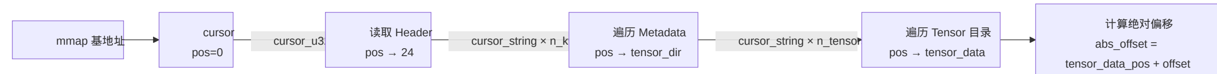
3. 二进制解析 — Cursor 模式

ds4.c 用 cursor（游标）模式解析 GGUF 二进制文件：

```c
typedef struct {
    const uint8_t *base;    // mmap 基地址
    uint64_t size;          // 总大小
    uint64_t pos;           // 当前位置
    char error[256];        // 错误信息
} ds4_cursor;
```

核心操作：

```c
// 检查是否有 n 字节可读
static bool cursor_has(ds4_cursor *c, uint64_t n) {
    if (n > c->size || c->pos > c->size - n) {
        cursor_error(c, "truncated GGUF file");
        return false;
    }
    return true;
}

// 读 n 字节到 dst，推进位置
static bool cursor_read(ds4_cursor *c, void *dst, uint64_t n) {
    if (!cursor_has(c, n)) return false;
    memcpy(dst, c->base + c->pos, (size_t)n);
    c->pos += n;
    return true;
}

// 读一个 u32
static bool cursor_u32(ds4_cursor *c, uint32_t *v) {
    return cursor_read(c, v, sizeof(*v));
}

// 读一个字符串（先读长度，再读内容）
static bool cursor_string(ds4_cursor *c, ds4_str *s) {
    uint64_t len;
    if (!cursor_u64(c, &len)) return false;
    if (!cursor_has(c, len)) return false;
    s->ptr = (const char *)(c->base + c->pos);
    s->len = len;
    c->pos += len;
    return true;
}
```



注意 `cursor_string` 返回的 `ds4_str` 是一个非拥有引用——`s->ptr` 直接指向 mmap 内部，没有拷贝。

4. 字节序（Endianness）

ds4.c 不做字节序转换。所有 `cursor_read` 用 `memcpy` 直接拷贝到原生类型。

```
memcpy(dst, c->base + c->pos, (size_t)n);
```

这意味着 ds4.c 假设运行平台是小端序（little-endian）。GGUF 格式也定义为小端序。Apple Silicon（ARM）和 Intel x86 都是小端序，所以这个假设在目标平台上成立。

5. ds4_str — 轻量字符串视图

```
typedef struct {
    const char *ptr;   // 指向 mmap 中的字符串数据
    uint64_t len;      // 字符串长度
} ds4_str;
```

不同于 C 标准库的以 `\0` 结尾的字符串，`ds4_str` 用长度字段标记边界。好处：

- 不需要拷贝/分配内存
- 可以包含 `\0` 字符
- O(1) 获取长度

LLM 知识点

1. GGUF 文件格式

GGUF 是一种二进制模型文件格式，结构如下：

关键设计：

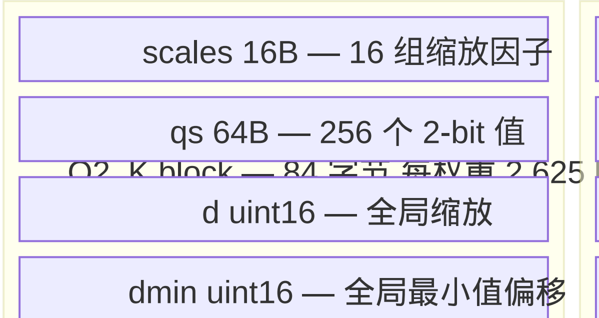
- 头部在前：先读头部，再根据头部信息定位数据
- 元数据与数据分离：元数据（模型名称、参数等）在前面，张量数据在后面
- 相对偏移：张量目录存储相对于 `tensor_data_pos` 的偏移，ds4.c 在解析时转为绝对偏移

2. 量化块格式

模型权重不需要 32 位浮点数精度。量化用更少的位存储权重，节省内存：

格式	每权重位数	块大小	用途
F32	32	1024 字节/256 权重	未量化（参考）
Q8_K	8	292 字节/256 权重	高精度量化
Q4_K	4	144 字节/256 权重	中等量化
Q2_K	~2.6	84 字节/256 权重	激进量化
IQ2_XXS	~2.06	66 字节/256 权重	超激进量化

每个量化块（block）包含 256 个权重值（`QK_K = 256`）：



Q4_K block — 144 字节 每权重 4.5 bit
d uint16 — 全局缩放
dmin uint16 — 全局最小值偏移
scales 12B — 12 组缩放+偏移
qs 128B — 256 个 4-bit 值

```
// Q2_K：每个 block 有 84 字节
typedef struct {
    uint8_t  scales[16];    // 缩放因子：16 组，每组 1 字节
    uint8_t  qs[64];        // 量化权重：256 个值，每个 2 bit（64 字节 = 256×2/8）
    uint16_t d;             // 全局缩放
    uint16_t dmin;          // 全局最小值偏移
} block_q2_K;
```

3. 模型验证 — "早失败"原则

ds4.c 读取模型元数据后，会严格验证每一个参数：

```
config_expect_u32("embedding_length", n_embd, DS4_N_EMBD);  // 必须是 4096
config_expect_u32("expert_count", n_expert, DS4_N_EXPERT);   // 必须是 256
config_expect_u32("attention.head_count", n_head, DS4_N_HEAD); // 必须是 64
// ... 30 多个参数全部检查 ...
```

如果任何一个参数不匹配，程序立即退出。这体现了"早失败"原则：

- 比在推理时产生错误结果好得多
- 比静默使用错误的模型参数好得多
- 错误信息明确指出哪个参数不匹配

4. 张量（Tensor）概念

张量是多维数组的数学术语。在 LLM 中：

维度	名称	例子
0D	标量	学习率
1D	向量	偏置项 bias
2D	矩阵	权重矩阵 W
3D+	张量	MoE 专家权重 [256专家, 2048, 4096]

ds4.c 中的张量结构体支持最多 8 维：

```
typedef struct {
    ds4_str name;
    uint32_t ndim;           // 维度数
    uint64_t dim[DS4_MAX_DIMS]; // 每维大小
    uint32_t type;           // 数据类型（F32/Q2_K/Q4_K 等）
    uint64_t abs_offset;     // 在 mmap 中的绝对偏移
    uint64_t elements;       // 总元素数（dim[0]×dim[1]×...）
    uint64_t bytes;          // 在文件中占用的字节数
} ds4_tensor;
```

反向链接

/glossary/gguf

/glossary/mmap

/glossary/page-fault

/glossary/xmalloc

端到端推理流程

学习日志

Part 3: 模型架构

Part 5: 服务层

Part 1: 练习

---

1. 项目全貌与编译系统

练习 1：阅读 Makefile

题目：打开项目根目录的 `Makefile`，回答以下问题：

1. 默认目标（`make` 不带参数时）会构建哪些可执行文件？
2. `ds4.o` 依赖哪些源文件？如果只修改了 `ds4.h`，`make` 会重新编译 `ds4.o` 吗？
3. 在 Linux 上编译时，`CFLAGS` 会多出什么标志？它的作用是什么？
4. `ds4_metal.o` 的编译命令和 `ds4.o` 有什么不同？为什么？

参考答案

1. `ds4`、`ds4-server`、`ds4-bench`、`ds4-eval`、`ds4-agent`（由 `all:` 目标定义）
2. 依赖 `ds4.c`、`ds4.h`、`ds4_metal.h`。修改 `ds4.h` 会触发重新编译，因为 `ds4.h` 在依赖列表中
3. `-DDS4_NO_METAL`，这会在预处理阶段定义 `DS4_NO_METAL` 宏，使代码中的 `#ifndef DS4_NO_METAL` 条件编译跳过 Metal 相关代码
4. `ds4_metal.o` 用 `$(OBJCFLAGS)`（包含 `-fobjc-arc`）编译，因为 `.m` 文件是 Objective-C，需要 ARC（自动引用计数）；而 `ds4.o` 用 `$(CFLAGS)`（包含 `-std=c99`）编译

练习 2：追踪编译过程

题目：假设你运行 `make clean && make ds4`（在 macOS 上），请按顺序列出会执行的命令。提示：`make` 按依赖顺序构建。

参考答案

```
# 第 1 步：编译 linenoise.o（行编辑库，无特殊依赖）
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o linenoise.o linenoise.c

# 第 2 步：编译 ds4.o（推理引擎核心）
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o ds4.o ds4.c

# 第 3 步：编译 ds4_metal.o（Metal GPU 桥接，Objective-C）
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -fobjc-arc -c -o ds4_metal.o ds4_metal.m

# 第 4 步：编译 ds4_cli.o（CLI 前端）
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o ds4_cli.o ds4_cli.c

# 第 5 步：链接所有 .o 文件为 ds4 可执行文件
cc -O3 -ffast-math ... -o ds4 ds4_cli.o linenoise.o ds4.o ds4_metal.o -lm -pthread -framework Foundation -framework Metal
```

练习 3：理解条件编译

题目：ds4.c 第 39-44 行有如下代码：

```
#ifndef DS4_NO_METAL
#include "ds4_metal.h"
#endif
#if defined(__ARM_NEON)
#include <arm_neon.h>
#endif
```

请解释：

1. 在什么情况下 `ds4_metal.h` 不会被包含？
2. `__ARM_NEON` 是在哪里定义的？为什么要检查它？
3. 如果你在 Intel Mac 上编译，会发生什么？

> 参考答案
>
> 1. 当 Makefile 定义了 `-DDS4_NO_METAL` 时（即 Linux 平台编译时），`DS4_NO_METAL` 被定义，`#ifndef` 条件为假，`ds4_metal.h` 不会被包含
> 2. `__ARM_NEON` 由编译器预定义，当目标架构支持 ARM NEON SIMD 指令集时自动定义。检查它是因为 `<arm_neon.h>` 只在 ARM 处理器上可用，Intel 处理器没有这个头文件
> 3. Intel Mac 上：`__ARM_NEON` 未定义，不包含 `<arm_neon.h>`；但 Intel Mac 仍然是 macOS（Darwin），所以 `DS4_NO_METAL` 未定义，会包含 `ds4_metal.h`（不过 Metal 性能可能不同）

练习 4：理解模型架构常量

题目：根据 ds4.c 第 82-105 行的常量，计算以下数值：

1. 每个 token 的注意力计算需要多少个 query 向量元素？（提示：头数 × 每头维度）
2. MoE 层中，被"跳过"（不激活）的专家占总专家数的比例是多少？
3. `DS4_THINK_MAX_MIN_CONTEXT` 是 393216，这个值大约是多少 K token？

> 参考答案
>
> 1. 64 × 512 = 32,768 个元素（这是 LoRA 压缩前的原始维度）
> 2. (256 - 6) / 256 = 250/256 ≈ 97.7% 的专家不被激活——这就是 MoE 节省计算的核心原因
> 3. 393216 / 1024 = 384K token（约 384K，注释里也写了"384K-token context window"）

练习 5：概念思考题

题目：用自己的话回答：

1. 为什么 ds4.c 不做成一个通用的 GGUF 加载器（像 llama.cpp 那样），而是只支持 DeepSeek V4 Flash？
2. 为什么推理引擎需要两个可执行文件（`ds4` 和 `ds4-server`），而不是一个？
3. opaque type（不透明类型）的设计在什么场景下特别有用？

> 参考答案
>
> 1. 专用化换来简化和优化：固定模型架构意味着不需要运行时动态读取维度、不需要通用张量抽象、可以硬编码所有内存布局，代码更简洁、性能更好。这是"做一件事并做好"的工程选择。
> 2. 不同的使用场景：`ds4` 是给人直接用的交互式工具；`ds4-server` 是给程序调用的网络服务。它们共享推理核心（ds4.o + ds4_metal.o），但前端交互方式完全不同。分开可以独立修改、部署。
> 3. 库/模块的封装：当你的代码被其他模块使用时，opaque type 隐藏内部细节，防止调用者依赖实现细节。这样你可以自由修改内部结构而不破坏兼容性。类似于面向对象语言中的"私有成员"。

今日学习检查清单

- [ ] 能说出 `make` 构建本项目时的完整步骤
- [ ] 理解 `?=`、`:=`、`+=` 在 Makefile 中的区别
- [ ] 能解释 `#ifndef` 条件编译的工作原理
- [ ] 能解释 ds4.c 的文件结构和设计哲学
- [ ] 能列出 DeepSeek V4 Flash 的关键架构参数（层数、维度、专家数等）
- [ ] 能解释 opaque type 的目的和好处
- [ ] 能说出推理引擎在 LLM 系统中的角色

延伸挑战

挑战 1（中级）：自定义编译目标

在 Makefile 中添加一个新目标 `make info`，运行时打印编译器版本、平台和所有 CFLAGS 变量。提示：用 `@echo` 和 `$(CC --version)`。

挑战 2（高级）：阅读 Makefile 以外的构建配置

查看 `ds4_bench.c` 和 `ds4_server.c` 的编译规则，解释为什么 `ds4_server` 不链接 `-framework Metal` 但仍然能使用 GPU。提示：看 `ds4.o` 的依赖。

---

2. 内存管理与 mmap

练习 1：xmalloc 包装器

题目：ds4.c 的 `xmalloc` 定义如下：

```
static void *xmalloc(size_t size) {
    void *p = malloc(size);
    if (!p) ds4_die("out of memory");
    return p;
}
```

而 `xmalloc_zeroed` 用 `xmalloc + memset` 代替 `calloc`。

请回答：

1. 为什么没有 `xfree` 包装器？`free` 不需要检查返回值吗？
2. 如果你在 `xmalloc` 中想添加分配大小限制（比如单次最大 1GB），应该在哪里加？怎么加？
3. `ds4_alloc_guard_check` 是怎么知道当前是否在守卫阶段的？

参考答案

1. `free` 没有返回值（`void`），也不需要检查。`free(NULL)` 在 C 标准中是安全的空操作。所以不需要 `xfree` 包装器。
2. 在 `xmalloc` 中，`malloc` 调用前加检查：

```
if (size > 1024ULL * 1024 * 1024) ds4_die("allocation too large");
```

3. 通过全局变量 `g_alloc_guard_enabled`（行 437）。`ds4_alloc_guard_begin` 设为 `true`，`ds4_alloc_guard_end` 设为 `false`。每次 `xmalloc`/`xcalloc`/`xrealloc` 调用时检查这个标志。

练习 2：mmap 标志选择

题目：ds4.c 行 1197-1199 有如下代码：

```
const int mmap_flags = metal_mapping ? MAP_SHARED : MAP_PRIVATE;
void *map = mmap(NULL, (size_t)st.st_size, PROT_READ, mmap_flags, fd, 0);
```

请回答：

1. Metal 路径用 `MAP_SHARED`，CPU 路径用 `MAP_PRIVATE`。为什么 Metal 需要 `MAP_SHARED`？
2. `PROT_READ` 表示什么？为什么不用 `PROT_WRITE`？
3. `MAP_FAILED` 和 `NULL` 有什么区别？mmap 失败时返回哪个？

参考答案

1. Metal GPU 可以把 `MAP_SHARED` 映射的内存包装为"零拷贝 MTLBuffer"——GPU 直接从文件映射读取，不需要额外拷贝。`MAP_PRIVATE` 创建的是写时复制副本，Metal 无法零拷贝引用。
2. `PROT_READ` 表示只读访问。模型权重文件不需要修改，用只读更安全（防止意外写入）。mmap 用 `PROT_READ` 后，任何写入尝试都会触发段错误（SIGSEGV）。
3. `mmap` 失败返回 `MAP_FAILED`（定义为 `(void *)-1`），不是 `NULL`。这是 mmap 设计上的一个历史"坑"——`NULL` 是合法的返回值（表示系统选择了地址），所以用 `-1` 表示失败。

练习 3：读取 ds4_model 结构体

题目：根据以下 `model_open` 中的代码（简化版），推断 `ds4_model` 各字段是如何被填充的：

```
int fd = open(path, O_RDONLY);
struct stat st;
fstat(fd, &st);

void *map = mmap(NULL, st.st_size, PROT_READ, flags, fd, 0);

ds4_cursor c = cursor_at(m, 0);
cursor_u32(&c, &magic);
cursor_u32(&c, &m->version);
cursor_u64(&c, &m->n_tensors);
cursor_u64(&c, &m->n_kv);
```

请画出从磁盘到结构体字段的数据流。

参考答案

```
磁盘文件 (GGUF)
    │
    ▼ open() + mmap()
    │
m->map ──────────────────────────► 虚拟地址空间基地址
m->fd  ──────────────────────────► 文件描述符
m->size ← st.st_size ◄────────────── fstat() 获取文件大小
    │
    ▼ cursor_at(m, 0) 从偏移 0 开始解析
    │
    ├── 字节 0-3:   cursor_u32 → magic (临时变量，检查后丢弃)
    ├── 字节 4-7:   cursor_u32 → m->version
    ├── 字节 8-15:  cursor_u64 → m->n_tensors
    ├── 字节 16-23: cursor_u64 → m->n_kv
    │
    ▼ 后续解析
    │
    ├── parse_metadata() → m->kv[]
    └── parse_tensors()  → m->tensors[]
```

cursor（游标）是一种常用的二进制解析模式：维护当前位置，每读一个字段自动前进。

练习 4：理解 calloc 零页问题

题目：假设操作系统用"共享零页"优化 calloc：

```
calloc(1000, 4096) = 4MB 内存
  ├── 实际只分配 1 个物理页（全零）
  └── 所有 1000 个虚拟页都指向这个物理页（只读）

当程序首次写入第 500 页时：
  ├── 触发 page fault
  ├── 操作系统分配一个新的物理页
  ├── 复制零页内容
  └── 映射虚拟页 500 到新物理页
```

请解释：为什么这个优化对 ds4.c 的 KV cache 是危险的？

参考答案

KV cache 在推理热循环中逐 token 填充。假设 KV cache 有 10 万个页：

- 用 calloc：10 万个虚拟页都指向共享零页。推理开始后，每处理一个 token，写入新的 KV 行，触发一个 page fault。100 个 token 就可能触发数百个 page fault。
- 用 malloc + memset：初始化时就写入所有页。memset 触发所有 page fault，操作系统为每个虚拟页分配独立的物理页。
- 在 macOS 上，当大量 page fault 同时发生，加上巨大的 mmap 映射，内核的虚拟内存计数器（`cpt_mapcnt_inc`）可能溢出，导致 kernel panic。
- `malloc + memset` 把 page fault 集中在启动时（可控），而不是分散在推理热循环中（不可控）。

练习 5：Scratch Buffer 设计思考

题目：ds4.c 用预分配的 scratch buffer 避免热循环分配。假设你要设计一个类似的结构体来存储矩阵乘法的中间结果，需要以下缓冲区：

- 输入向量 A（1024 个 float）
- 输入向量 B（1024 个 float）
- 输出向量 C（1024 个 float）
- 临时缓冲区 temp（2048 个 float）

请写出：

1. 结构体定义
2. 初始化函数
3. 释放函数

参考答案

```c
typedef struct {
    float *a;
    float *b;
    float *c;
    float *temp;
} matmul_scratch;

static void matmul_scratch_init(matmul_scratch *s) {
    s->a    = xmalloc(1024 * sizeof(float));
    s->b    = xmalloc(1024 * sizeof(float));
    s->c    = xmalloc(1024 * sizeof(float));
    s->temp = xmalloc(2048 * sizeof(float));
}

static void matmul_scratch_free(matmul_scratch *s) {
    if (!s) return;
    free(s->temp);
    free(s->c);
    free(s->b);
    free(s->a);
    memset(s, 0, sizeof(*s));
}
```

要点：

- 用 `xmalloc` 确保分配失败时退出
- 释放后用 `memset` 清零，防止使用已释放的指针
- 释放顺序和分配顺序相反（虽然对 `free` 来说不必须，但这是好习惯）

今日学习检查清单

- [ ] 能解释 xmalloc/xcalloc/xrealloc 的作用和为什么没有 xfree
- [ ] 能说出 mmap 相比 fread 的三个优势
- [ ] 能解释 MAP_SHARED 和 MAP_PRIVATE 的区别以及 ds4.c 的选择原因
- [ ] 理解为什么 ds4.c 用 malloc+memset 代替 calloc

- [ ] 理解 posix_madvise(MADV_WILLNEED) 的作用
- [ ] 能解释 __thread 线程局部存储的工作原理
- [ ] 能描述模型加载的完整调用链（open → mmap → parse → bind）
- [ ] 理解预分配 scratch buffer 模式的好处
- [ ] 能解释 volatile 在 weight warming 中的作用

延伸挑战

挑战 1（中级）：对比 **mmap** 和 **read** 的加载时间

写一个小程序，分别用 `fread` 和 `mmap` 读取一个 1GB 的文件，对比首次访问的时间差异。用 `clock_gettime(CLOCK_MONOTONIC)` 计时。

挑战 2（高级）：分析 **scratch buffer** 的内存布局

阅读 `ds4.c` 中 scratch buffer 的分配代码，画出完整的内存布局图（包含每个子缓冲区的名称、大小和偏移）。计算 43 层总共需要多少 scratch 空间。

---

3. GGUF 二进制格式

练习 1：计算量化块大小

题目：给定以下结构体定义，手动计算 `sizeof` 每个结构体（考虑对齐填充）：

```
typedef struct {
    float   d;          // 4 字节
    int8_t  qs[256];    // 256 字节
    int16_t bsums[16];  // 32 字节
} block_q8_K;
```

> 提示：`float` 是 4 字节对齐，`int16_t` 是 2 字节对齐，`int8_t` 是 1 字节对齐。

参考答案

```
偏移 0:   d (float, 4 字节)
偏移 4:   qs[0..255] (int8_t, 256 字节)
偏移 260: bsums[0..15] (int16_t, 32 字节)
           bsums 需要 2 字节对齐，260 是偶数 ✓
偏移 292: 结束

sizeof(block_q8_K) = 292 字节 ✓（与 static_assert 一致）
```

没有填充！因为字段顺序排列得当。

练习 2：读取 GGUF 头部

题目：假设一个 GGUF 文件的头部 24 字节（十六进制）如下：

```
47 55 47 46  03 00 00 00  F0 01 00 00 00 00 00 00  60 00 00 00 00 00 00 00
```

请解析出 magic、version、n_tensors、n_kv 四个字段（小端序）。

参考答案

```
magic:     47 55 47 46 → 0x46554747 → "GGUF" ✓
version:   03 00 00 00 → 3
n_tensors: F0 01 00 00 00 00 00 00 → 0x01F0 = 496 个张量
n_kv:      60 00 00 00 00 00 00 00 → 0x60 = 96 个元数据 KV 对
```

小端序：低位字节在前。`03 00 00 00` = 0x00000003 = 3。

练习 3：理解 cursor_read

题目：`cursor_read` 函数实现如下：

```c
static bool cursor_read(ds4_cursor *c, void *dst, uint64_t n) {
    if (!cursor_has(c, n)) return false;
    memcpy(dst, c->base + c->pos, (size_t)n);
    c->pos += n;
    return true;
}
```

请回答：

1. 如果 `c->pos = 100, c->size = 200, n = 150`，会发生什么？
2. 为什么用 `memcpy` 而不是直接赋值（如 `*(uint32_t*)dst = ...`）？
3. `cursor_string` 中的字符串存储在哪里？需要 `free` 吗？

参考答案

1. `cursor_has(150)` 检查 `150 > 200 || 100 > 200 - 150`，即 `150 > 200 || 100 > 50`，第二个条件为真，返回 `false`。这是边界检查，防止越界读取。
2. 直接赋值（`*(uint32_t*)ptr = value`）可能在某些架构上触发未对齐访问（alignment fault）。`memcpy` 不要求对齐，总是安全的。编译器通常会把小量 `memcpy` 优化为一条 load 指令。
3. `cursor_string` 返回的字符串直接指向 mmap 内存（`s->ptr = c->base + c->pos`），不需要 `free`。mmap 内存在 `model_close` 时用 `munmap` 整体释放。

练习 4：static_assert 原理

题目：`DS4_STATIC_ASSERT` 定义为：

```c
#define DS4_STATIC_ASSERT(name, cond) typedef char name[(cond) ? 1 : -1]
```

请回答：

1. 为什么 `(cond) ? 1 : -1` 能实现编译时检查？
2. 如果写 `DS4_STATIC_ASSERT(foo, sizeof(int) == 8)` 在 32 位系统上，会发生什么？
3. 为什么不直接用 C11 的 `_Static_assert`？

参考答案

1. C 标准不允许数组大小为负数。当 `cond` 为假时，数组大小为 -1，编译器会报错（如 "size of array 'foo' is negative"）。当 `cond` 为真，数组大小为 1，正常通过。由于 `cond` 必须是编译时常量表达式，这个检查在编译时完成。
2. `sizeof(int)` 在 32 位系统上是 4，`4 == 8` 为假，数组大小为 -1，编译报错。
3. `_Static_assert` 是 C11 引入的，而 ds4.c 用 `-std=c99` 编译。C99 没有标准的 static assert，所以用这个数组技巧。此外 `_Static_assert` 的错误信息更友好（可以指定消息字符串），但数组技巧在所有 C 编译器上都可用。

练习 5：理解模型验证

题目：`config_validate_model` 检查 30 多个参数。假设你有一个 GGUF 文件，它的 `embedding_length` 是 8192 而不是 4096。请描述从文件打开到错误报告的完整过程。

参考答案

```
1. model_open() 打开文件，mmap 成功
2. parse_metadata() 读取所有 KV 对，包括 "deepseek4.embedding_length" = 8192
3. parse_tensors() 读取张量目录
4. ds4_engine_open() 调用 config_validate_model()
5. config_validate_model() 中：
   a. required_u32(m, "deepseek4.embedding_length") → 8192
   b. config_expect_u32("embedding_length", 8192, DS4_N_EMBD=4096)
   c. 8192 != 4096 → 打印：
      "ds4: expected embedding_length=4096 for DeepSeek4 Flash, got 8192"
   d. exit(1) — 程序终止
```

这种"早失败"设计确保不兼容的模型文件不会导致后续推理产生无意义的输出。

今日学习检查清单

- ☐ 能画出 GGUF 文件的字节布局（头部 → 元数据 → 张量目录 → 张量数据）
- ☐ 理解 cursor 模式的三个操作：has / read / skip
- ☐ 能手动计算简单结构体的 sizeof（考虑对齐填充）
- ☐ 能解释 static_assert 的原理
- ☐ 理解为什么 cursor_string 返回的字符串不需要 free
- ☐ 理解小端序的含义以及 ds4.c 为什么不做字节序转换
- ☐ 能解释 parse_metadata 的懒加载策略
- ☐ 能解释 parse_tensors 中为什么要两步解析（先读信息再计算偏移）
- ☐ 能说出 Q2_K 和 Q4_K 量化块的大小和每权重位数
- ☐ 理解"早失败"验证原则的意义

延伸挑战

挑战 1（中级）：写一个最小 **GGUF** 解析器

用 C 写一个独立程序，接受 GGUF 文件路径，打印：magic number、version、张量数量、前 5 条元数据 KV。不使用 ds4.c 的代码，自己实现游标模式。

挑战 2（高级）：量化格式对比

用 `xxd` 或 `hexdump` 打开模型的 GGUF 文件，定位到张量数据区域。对比同一个权重在 Q2_K 和 Q4_K 格式下的二进制布局差异，解释为什么 Q4_K 的精度更高。

Part 1: 代码走读

---

1. 项目全貌与编译系统

追踪 1：从 `make` 到可执行文件的完整路径

依赖关系图

```
make (默认目标: all)
├── ds4                    (CLI 交互式聊天)
│   ├── ds4_cli.o          ← ds4_cli.c + ds4.h + linenoise.h
│   ├── linenoise.o        ← linenoise.c + linenoise.h
│   ├── ds4.o              ← ds4.c + ds4.h + ds4_metal.h
│   └── ds4_metal.o        ← ds4_metal.m + ds4_metal.h + metal/*.metal
│
├── ds4-server             (HTTP API 服务器)
│   ├── ds4_server.o       ← ds4_server.c + ds4.h
│   ├── ds4.o              ← (同上)
│   └── ds4_metal.o        ← (同上)
│
├── ds4-bench              (吞吐量基准测试)
│   ├── ds4_bench.o        ← ds4_bench.c + ds4.h
│   └── ds4.o + ds4_metal.o
│
└── ds4-eval               (能力评估，93 道题回归测试)
    ├── ds4_eval.o         ← ds4_eval.c + ds4.h
    └── ds4.o + ds4_metal.o
```

核心库 `ds4.o + ds4_metal.o` 被所有可执行文件共享。

追踪编译命令

```
# 步骤 1：编译推理引擎核心
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o ds4.o ds4.c

# 步骤 2：编译 Metal GPU 桥接层
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -fobjc-arc -c -o ds4_metal.o ds4_m
etal.m

# 步骤 3：编译 CLI 前端
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o ds4_cli.o ds4_cli.c

# 步骤 4：编译行编辑库
cc -O3 -ffast-math -mcpu=native -Wall -Wextra -std=c99 -c -o linenoise.o linenoi
se.c

# 步骤 5：链接 CLI
cc -O3 -ffast-math ... -o ds4 ds4_cli.o linenoise.o ds4.o ds4_metal.o -lm -pthre
ad -framework Foundation -framework Metal
```

注意第 5 步把所有 .o 文件和库合并成最终可执行文件。

追踪 2：ds4.c 文件头部（行 1-120）

文件头注释（行 1-15）

```
ds4.c:735KB, ~20,000+ 行
```

开头的注释说明了这个文件的设计哲学：

> "This file is deliberately vertical" 这个文件是刻意垂直整合的——它拥有 GGUF 加载、固定张量布局、CPU 参考内核、Metal 图驱动器和分词器连接。

关键词：**mmap based**（基于内存映射加载），张量数据留在内核页缓存中，直到推理需要它。

标准库包含（行 17-35）

ds4.c 的 18 个 `#include` 可以按功能分组：

功能	头文件
文件 I/O	`<fcntl.h>`, `<unistd.h>`, `<sys/stat.h>`, `<sys/file.h>`
内存管理	`<sys/mman.h>`, `<stdlib.h>`
字符串/内存	`<string.h>`, `<ctype.h>`
数学运算	`<math.h>`, `<float.h>`
类型定义	`<stdint.h>`, `<stdbool.h>`, `<stddef.h>`, `<inttypes.h>`
线程	`<pthread.h>`
输入输出	`<stdio.h>`
可变参数	`<stdarg.h>`
时间	`<time.h>`
错误	`<errno.h>`

条件包含（行 **37-44**）

```
#include "ds4.h"                    // 公共 API —— 始终包含

#ifndef DS4_NO_METAL
#include "ds4_metal.h"              // Metal GPU 支持 —— 仅 macOS
#endif

#if defined(__ARM_NEON)
#include <arm_neon.h>               // ARM SIMD 指令 —— 仅 ARM 芯片
#endif
```

这展示了条件编译的两种用法：

- `#ifndef DS4_NO_METAL`：在 Makefile 中通过 `-DDS4_NO_METAL` 禁用 GPU
- `#if defined(__ARM_NEON)`：编译器自动根据 CPU 架构定义的宏

数学常量与模型超参（行 **46-71**）

```
#define DS4_NEG_INF (-1.0e30f)      // 不是真正的 -∞，是足够大的负数
#define DS4_POS_INF ( 1.0e30f)      // 同理，用于 softmax 中的掩码
#define DS4_RMS_EPS  ( 1.0e-6f)     // RMSNorm 的 epsilon，防止除零
```

为什么用 `1.0e30f` 而不是 `INFINITY`？

- `float` 的范围约 ±3.4e38，1e30 足够大

- 避免浮点异常和 NaN 传播
- 在 softmax 等运算中，任何正常值减去 1e30 都会变成 0（exp(-大数) ≈ 0）

DeepSeek V4 Flash 架构常量（行 **82-105**）

```
enum {
    DS4_N_LAYER             = 43,     // Transformer 层数
    DS4_N_EMBD              = 4096,   // 嵌入维度（每个 token 的向量长度）
    DS4_N_VOCAB             = 129280, // 词表大小
    DS4_N_HEAD              = 64,     // 注意力头数
    DS4_N_HEAD_KV           = 1,      // KV 头数（GQA：64:1 分组）
    DS4_N_HEAD_DIM          = 512,    // 每个头的维度
    DS4_N_EXPERT            = 256,    // MoE 专家总数
    DS4_N_EXPERT_USED       = 6,      // 每个 token 激活的专家数
    ...
};
```

为什么用 `enum` 而不是 `#define`？

- 编译器可以进行类型检查
- 调试器中可以看到枚举名（`DS4_N_LAYER` 比 `43` 有意义得多）
- 所有常量在一个 `enum` 块中，便于阅读和维护

追踪 3：ds4.h 公共 API（行 1-161）

设计模式：**opaque type**（不透明类型）

```
typedef struct ds4_engine ds4_engine;      // 前向声明，不暴露内部
typedef struct ds4_session ds4_session;    // 同上
```

这是 C 语言实现封装的经典手法：

- 头文件只声明类型名，不定义结构体字段
- 结构体的完整定义藏在 ds4.c 中
- 外部代码（CLI、server）只能通过指针操作，不能直接访问字段

好处：

1. 内部实现可以随意修改，不影响调用者

2. 强制使用者通过 API 函数操作，不会误用内部状态
3. 编译时间更快（修改 ds4.c 不需要重新编译 ds4_server.c）

API 层次

```
ds4_engine              模型（加载一次，只读）
  ├── open/close        生命周期管理
  ├── generate          简单推理（一次性）
  └── tokenize          分词

ds4_session             会话（每次对话一个，可变）
  ├── create/free       生命周期管理
  ├── sync              同步到指定 prompt 前缀（复用 KV cache）
  ├── argmax/sample     选择下一个 token
  └── eval              执行一步推理

ds4_tokens              动态数组
  └── push/free/copy    管理操作
```

关键概念：`ds4_session_sync()` 是高性能的关键——如果新 prompt 和已有 cache 有共同前缀，只需计算新增部分，不需要从头开始。

追踪 4：项目文件大小对比

```
ds4.c          735 KB   推理引擎核心（最大）
ds4_metal.m    639 KB   Metal GPU 桥接
ds4_server.c   269 KB   HTTP 服务器
linenoise.c     83 KB   行编辑库
ds4_cli.c       48 KB   CLI 前端
ds4_metal.h     31 KB   Metal 头文件
ds4.h            6 KB   公共 API
linenoise.h      5 KB   行编辑头文件
```

ds4.c 占了项目代码的大部分。它是一个典型的"单文件巨石"（monolithic）设计，所有推理逻辑集中在一个文件中。

2. 内存管理与 mmap

追踪 1：模型加载完整调用链

从用户运行 `./ds4 -p "Hello"` 到模型加载完成的调用链：

```
main()                          [ds4_cli.c]
  └── ds4_engine_open()         [ds4.h 约 81 行, ds4.c 约 15716 行]
        ├── xcalloc(1, sizeof(*e))           分配引擎结构体
        ├── model_open(&e->model, path, metal, true)  加载主模型
        │     ├── open(path, O_RDONLY)            打开文件
        │     ├── fstat(fd, &st)                  获取文件大小
        │     ├── mmap(NULL, size, PROT_READ,     映射到内存
        │     │       MAP_SHARED/MAP_PRIVATE, fd, 0)
        │     ├── cursor_u32() × 4                 解析 GGUF 头部
        │     ├── parse_metadata(m, &c)            解析元数据表
        │     │     └── calloc(n_kv, sizeof(kv))     分配 KV 数组
        │     ├── parse_tensors(m, &c)             解析张量目录
        │     │     └── calloc(n_tensors, ...)       分配张量数组
        │     └── model_prefetch_cpu_mapping(m)    若 CPU 后端且 prefetch_cpu=tru
e
        ├── model_warm_weights()                可选预热
        │     └── posix_madvise(WILLNEED)          提示内核预读
        │     └── for each page: p[off]            触发 page fault
        ├── vocab_load()                        加载分词器
        ├── config_validate_model()             验证模型参数
        └── weights_bind()                      绑定权重指针
              └── tensor_data(m, t)                返回 mmap 偏移
```

`model_open` 的 **`prefetch_cpu`** 参数：第四个参数控制是否触发 CPU 映射预取。正常推理引擎传 `true`，但 `ds4_dump_text_tokenization()` 只需要分词器，不需要遍历巨大的张量数据，因此传 `false` 以节省时间。

关键数据流

```
磁盘文件 (81GB GGUF)
     │
     ▼ open() + mmap()
     │
虚拟地址空间 (m->map 指针)
     │
     ├── parse_metadata() → m->kv[]        元数据（模型名称、参数等）
     ├── parse_tensors()  → m->tensors[]   张量目录（名称、偏移、大小）
     │
     ▼ weights_bind()
     │
weights 结构体（指针数组，全部指向 mmap 内部）
     ├── tok_embd      → m->map + offset_0    词嵌入矩阵
     ├── layer[0].wq   → m->map + offset_1    第 0 层 query 权重
     ├── layer[0].wk   → m->map + offset_2    第 0 层 key 权重
     ├── ...
     └── output        → m->map + offset_N    输出投影
```

追踪 2：ds4_model 结构体（行 895-908）

```c
typedef struct {
    int fd;                     // 文件描述符（mmap 需要）
    const uint8_t *map;         // mmap 返回的基地址
    uint64_t size;              // 文件/映射总大小

    uint32_t version;           // GGUF 版本号（必须是 3）
    uint64_t n_kv;              // 元数据 KV 对数量
    uint64_t n_tensors;         // 张量数量
    uint64_t alignment;         // 数据对齐要求
    uint64_t tensor_data_pos;   // 张量数据在文件中的起始位置

    ds4_kv *kv;                 // 元数据数组（calloc 分配）
    ds4_tensor *tensors;        // 张量目录数组（calloc 分配）
} ds4_model;
```

内存所有权：

- `map`：mmap 拥有，`munmap` 释放
- `kv`：`calloc` 分配，`free` 释放
- `tensors`：`calloc` 分配，`free` 释放
- `fd`：`open` 返回，`close` 关闭

model_close（行 1057-1065）— 释放顺序

```
static void model_close(ds4_model *m) {
    if (!m) return;
    free(m->kv);                                          // 先释放小对象
    free(m->tensors);
    if (m->map) munmap((void *)m->map, (size_t)m->size); // 再释放大映射
    if (m->fd >= 0) close(m->fd);                        // 最后关文件
    memset(m, 0, sizeof(*m));                             // 清零结构体
    m->fd = -1;                                          // 重置为无效值
}
```

注意 `memset(m, 0, sizeof(*m))` 和 `m->fd = -1`：确保关闭后结构体处于安全的"空"状态。

追踪 3：错误处理层次

ds4.c 有三个层次的错误处理：

致命错误（不可恢复）

```
// 行 400-403
static void ds4_die(const char *msg) {
    fprintf(stderr, "ds4: %s\n", msg);
    exit(1);
}

// 行 413-416
static void ds4_die_errno(const char *what, const char *path) {
    fprintf(stderr, "ds4: %s '%s': %s\n", what, path, strerror(errno));
    exit(1);
}
```

用于：内存不足、文件不存在、格式错误等。这类错误发生后程序无法继续运行。

警告（可恢复）

```
// posix_madvise 失败时只打印警告
if (rc != 0) {
    ds4_log(stderr, DS4_LOG_WARNING,
            "ds4: warning: POSIX_MADV_WILLNEED failed: %s\n", strerror(rc));
}
```

`posix_madvise` 只是建议，失败了不影响正确性，只是性能可能受影响。

分配守卫（调试用）

```
// 行 437-459: 分配守卫基础设施
static bool g_alloc_guard_enabled;
static const char *g_alloc_guard_phase;

static void ds4_alloc_guard_check(const char *op, size_t size) {
    if (!g_alloc_guard_enabled) return;    // 守卫未启用，跳过
    fprintf(stderr, "ds4: internal allocation during %s: %s(%zu)\n",
            g_alloc_guard_phase, op, size);
    exit(1);                               // 热循环中有分配就报错
}

// 行 14470: 在生成循环中启用守卫
ds4_alloc_guard_begin("CPU token generation");
for (int i = 0; i < n_predict && pos < ctx_size; i++) {
    // 这里任何 malloc/calloc/realloc 都会触发 ds4_die
}
ds4_alloc_guard_end();
```

追踪 4：xmalloc_zeroed vs calloc（行 482-498）

```
static void *xmalloc_zeroed(size_t n, size_t size) {
    // 溢出检查
    if (size != 0 && n > SIZE_MAX / size) ds4_die("allocation size overflow");

    const size_t total = n * size;
    void *p = xmalloc(total ? total : 1);

    /* 注释解释了为什么不用 calloc：
     * 大块未触碰的 calloc 内存可能通过共享零页管理。
     * CPU 解码 KV cache 每次增长一个 token，用 calloc 会把
     * 首次缺页移到生成阶段。Darwin 上观察到内核 panic。 */
    memset(p, 0, total);

    return p;
}
```

调用链：

```
kv_cache_init()                         [行 6027]
  └── for each layer:
        └── xmalloc_zeroed(raw_cap * DS4_N_HEAD_DIM, sizeof(float))   [行 482]
              ├── 溢出检查
              ├── xmalloc(total)      malloc + NULL 检查
              └── memset(p, 0, total) 手动清零
```

追踪 5：线程局部存储（行 605-614）

```
// 行 605-610: 线程池结构体
typedef struct {
    // ... pthread 同步原语 ...
} ds4_thread_pool;

// 行 612-614: 全局变量
static ds4_thread_pool g_pool;              // 全局共享的线程池
static __thread int g_parallel_depth;       // 每线程独立的递归深度
static uint32_t g_requested_threads;        // 请求的线程数
```

`__thread` 的内存模型：

```
线程 1 (主线程)              线程 2 (工作线程)
┌─────────────────┐          ┌─────────────────┐
│ g_parallel_depth │          │ g_parallel_depth │
│     = 0          │          │     = 1          │
└─────────────────┘          └─────────────────┘
      独立副本                     独立副本

g_pool ←── 两个线程共享同一个线程池 ──→ g_pool
```

追踪 6：模型预热的页面扫描（行 1328-1353）

```c
static void model_warm_weights(const ds4_model *m) {
    const uint64_t start = m->tensor_data_pos;   // 张量数据起始位置
    const uint64_t end = m->size;                 // 文件末尾
    const uint64_t page = sysconf(_SC_PAGESIZE);  // 系统页大小（通常 16KB）

    volatile uint64_t checksum = 0;               // volatile 防止优化器删除

    // 先用 posix_madvise 提示内核
    posix_madvise(p + start, end - start, POSIX_MADV_WILLNEED);

    // 每页读一个字节，强制 page fault
    for (uint64_t off = start; off < end; off += page) {
        checksum += p[off];
    }
    checksum += p[end - 1];                       // 最后一个字节
}
```

为什么用 `volatile`？编译器看到 `checksum` 计算后没有被使用，可能会优化掉整个循环。`volatile` 告诉编译器"这个变量可能被外部修改，不要优化掉对它的读写"。

实际上这里的目的不是真的要 checksum，而是强制触发每一页的 page fault。

---

3. GGUF 二进制格式

追踪 1：GGUF 文件头部解析

调用链

```
model_open()                              [行 1174]
  ├── open(path, O_RDONLY)                   打开文件
  ├── fstat(fd, &st)                         获取大小
  ├── mmap(NULL, size, PROT_READ, ...)       内存映射
  │
  ├── cursor_at(m, 0)                        创建游标，从偏移 0 开始
  │     base = m->map
  │     size = m->size
  │     pos = 0
  │
  ├── cursor_u32(&c, &magic)                 读 4 字节 → 0x46554747 ("GGUF")
  │     └── cursor_read(&c, &magic, 4)
  │           └── memcpy(&magic, base+0, 4)  → pos=4
  │
  ├── cursor_u32(&c, &version)               读 4 字节 → 3
  │     └── memcpy(&version, base+4, 4)    → pos=8
  │
  ├── cursor_u64(&c, &n_tensors)             读 8 字节 → 张量数量
  │     └── memcpy(&n_tensors, base+8, 8)  → pos=16
  │
  └── cursor_u64(&c, &n_kv)                 读 8 字节 → 元数据 KV 数量
        └── memcpy(&n_kv, base+16, 8)      → pos=24
```

GGUF 头部字节布局

```
偏移  大小  字段          值
0     4     magic         "GGUF" = 0x47 0x55 0x47 0x46
4     4     version       3
8     8     n_tensors     ~500+（取决于模型）
16    8     n_kv          ~100+（模型元数据对数）
24    ...   metadata[0]   开始
```

追踪 2：parse_metadata（行 1093-1119）

解析流程

```
parse_metadata(m, &c)                    [行 1093]
  │ pos = 24（头部之后）
  │
  ├── calloc(n_kv, sizeof(ds4_kv))       分配 KV 数组
  │
  └── for i = 0..n_kv-1:                 遍历每个 KV 对
        │
        ├── cursor_string(&c, &kv->key)      读 key
        │     ├── cursor_u64(&c, &len)         读字符串长度
        │     └── s->ptr = base + pos; pos += len  指向 mmap 内部
        │
        ├── cursor_u32(&c, &kv->type)        读类型（0=bool, 1=u32, ...）
        │
        ├── kv->value_pos = c->pos           记录 value 起始位置（不解析！）
        │
        └── skip_value(&c, kv->type, 0)      跳过 value 内容
```

懒加载策略

`ds4_kv` 只存储 `value_pos`（value 在文件中的偏移），不立即解析：

```c
typedef struct {
    ds4_str key;          // key 字符串（指向 mmap）
    uint32_t type;        // value 类型
    uint64_t value_pos;   // value 在 mmap 中的偏移
} ds4_kv;
```

只有在需要时（如 `config_validate_model`），才用 `cursor_at(m, kv->value_pos)` 读取具体值。这避免了为每个元数据值分配内存。

skip_value — 递归跳过（行 932-970）

```c
static bool skip_value(ds4_cursor *c, uint32_t type, int depth) {
    uint64_t scalar = scalar_value_size(type);
    if (scalar != 0) return cursor_skip(c, scalar);     // 标量类型：直接跳过

    if (type == GGUF_VALUE_STRING) {                    // 字符串：读长度+跳过
        ds4_str ignored;
        return cursor_string(c, &ignored);
    }

    if (type == GGUF_VALUE_ARRAY) {                     // 数组：递归跳过每个元素
        uint32_t item_type;
        uint64_t len;
        cursor_u32(c, &item_type);
        cursor_u64(c, &len);
        for (uint64_t i = 0; i < len; i++)
            skip_value(c, item_type, depth + 1);
        return true;
    }
}
```

追踪 3：parse_tensors（行 1123-1170）

解析流程

```
parse_tensors(m, &c)                      [行 1123]
  │
  ├── calloc(n_tensors, sizeof(ds4_tensor))   分配张量数组
  │
  └── for i = 0..n_tensors-1:                 遍历每个张量
        │
        ├── cursor_string(&c, &t->name)        名称
        ├── cursor_u32(&c, &t->ndim)           维度数
        ├── for d = 0..ndim-1:
        │     └── cursor_u64(&c, &t->dim[d])   每维大小
        │     └── t->elements *= t->dim[d]     计算总元素数
        ├── cursor_u32(&c, &t->type)           数据类型
        └── cursor_u64(&c, &t->rel_offset)     相对偏移

  │
  ├── m->tensor_data_pos = align_up(c->pos, m->alignment)
  │     计算张量数据的起始位置（对齐到 alignment 字节边界）
  │
  └── for i = 0..n_tensors-1:                   转换为绝对偏移
        t->abs_offset = tensor_data_pos + t->rel_offset
        验证: abs_offset + bytes <= m->size    不超出文件范围
```

为什么分两步解析？

1. 先读元信息：名称、维度、类型、相对偏移
2. 再计算绝对偏移：需要知道 `tensor_data_pos`，它是在所有张量元信息读完之后才确定的

追踪 4：量化块结构体大小验证

```c
// 行 153
#define DS4_STATIC_ASSERT(name, cond) typedef char name[(cond) ? 1 : -1]

// 行 154-157
DS4_STATIC_ASSERT(ds4_block_q2_k_size,    sizeof(block_q2_K)    == 84);   // ✓ 16+64+2+2 = 84
DS4_STATIC_ASSERT(ds4_block_q4_k_size,    sizeof(block_q4_K)    == 144);  // ✓ 2+2+12+128 = 144
DS4_STATIC_ASSERT(ds4_block_q8_k_size,    sizeof(block_q8_K)    == 292);  // ✓ 4+256+32 = 292
DS4_STATIC_ASSERT(ds4_block_iq2_xxs_size, sizeof(block_iq2_xxs) == 66);   // ✓ 2+64 = 66
```

block_q4_K 内存布局验证

```
typedef struct {
    uint16_t d;             // 2 字节，偏移 0
    uint16_t dmin;          // 2 字节，偏移 2
    uint8_t  scales[12];    // 12 字节，偏移 4
    uint8_t  qs[QK_K/2];    // 128 字节，偏移 16 (QK_K=256, 256/2=128)
} block_q4_K;               // 总计 144 字节 ✓
```

128 个 `uint8_t` 存储 256 个 4-bit 权重：每个字节存 2 个权重（高 4 bit + 低 4 bit）。

追踪 5：config_validate_model（行 2343-2425）

```
config_validate_model(&e->model)         [行 2343]
  │
  ├── 从元数据读取模型参数
  │     required_u32(m, "deepseek4.block_count")          → 43
  │     required_u32(m, "deepseek4.embedding_length")     → 4096
  │     required_u32(m, "deepseek4.vocab_size")           → 129280
  │     required_u32(m, "deepseek4.attention.head_count") → 64
  │     ...
  │     （30 多个参数）
  │
  ├── 逐个与硬编码常量比较
  │     config_expect_u32("block_count", 43, DS4_N_LAYER)       ✓
  │     config_expect_u32("vocab_size", 129280, DS4_N_VOCAB)   ✓
  │     ...
  │
  └── 任何一个不匹配 → fprintf(stderr, ...) + exit(1)
```

这个函数确保"文件里的模型"和"代码期望的模型"完全一致。是"专用引擎"设计的关键体现。

Part 2: 分词与采样

> 文本如何变成数字？模型输出如何变回文本？本主题覆盖 BPE 分词器和采样策略——LLM 的输入和输出边界。

涵盖内容

章节	核心主题
1. bpe 分词器	哈希表、BPE 合并规则、GPT-2 字节编码、Chat 模板
2. 采样策略	softmax、temperature、top-k、top-p、xorshift64 PRNG

核心概念

- bpe — Byte Pair Encoding 分词
- softmax — logits → 概率分布
- prng — 伪随机数生成（推理中的采样随机性）

前置知识

- Part 1: 构建与加载（项目结构、内存管理）
- 哈希表基础
- 浮点数运算

学习路径

读完本主题后，你将理解：

1. 模型如何将文本拆分为 token 序列

2. 如何从 129280 个 logits 中采样下一个 token

3. temperature、top-k、top-p 等采样参数的实际效果

→ 下一步：Part 3: 模型架构

Part 2: 分词与采样

> BPE tokenizer、采样策略、softmax

---

1. BPE 分词器

> 模型只能处理数字。分词器是文本和数字之间的桥梁——"你好"变成 [3472, 19238] 才能进入模型。今天学习哈希表、BPE 合并算法，以及 Chat 模板如何组装最终输入。

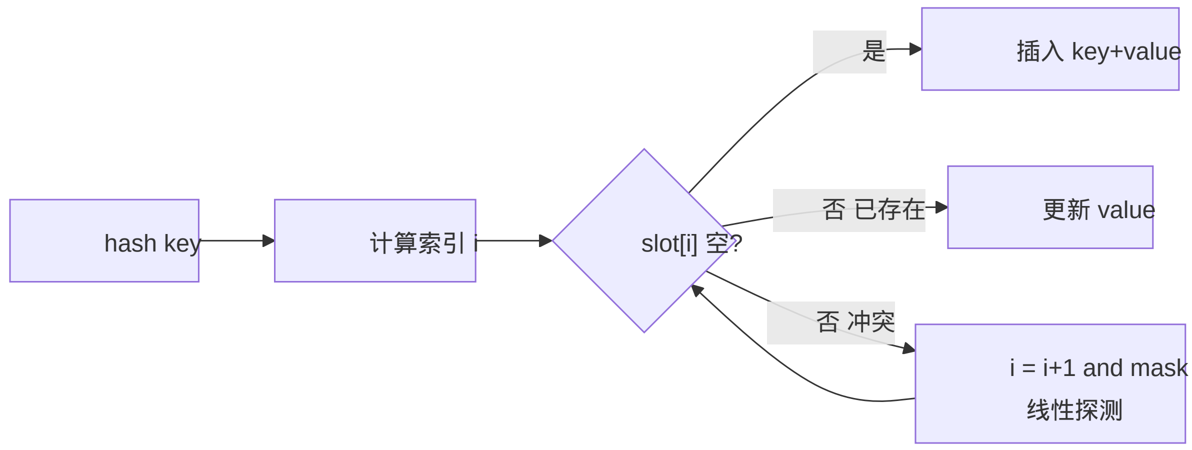
C 知识点

1. 开放寻址哈希表（**Open Addressing Hash Table**）

ds4.c 实现了一个简单的开放寻址哈希表，用于 token 字符串到 ID 的映射：

```
typedef struct {
    ds4_str key;        // 键（字符串视图，指向 mmap）
    int value;          // 值（token ID）
    bool used;          // 该槽位是否被占用
} str_i32_entry;

typedef struct {
    str_i32_entry *entry;  // 槽位数组
    uint64_t cap;          // 容量（2 的幂次）
    uint64_t used;         // 已使用数
} str_i32_table;
```

插入操作（`table_put`）：




```c
static void table_put(str_i32_table *t, ds4_str key, int value) {
    uint64_t mask = t->cap - 1;                        // 容量必须是 2 的幂
    uint64_t i = hash_bytes(key.ptr, key.len) & mask;  // 计算起始位置

    while (t->entry[i].used) {                         // 线性探测
        if (ds4_str_eq(t->entry[i].key, key)) {        // 已存在→更新
            t->entry[i].value = value;
            return;
        }
        i = (i + 1) & mask;                            // 环形前进
    }

    t->entry[i].used = true;                           // 找到空位→插入
    t->entry[i].key = key;
    t->entry[i].value = value;
    t->used++;
}
```

关键点：

- 容量必须是 **2** 的幂：这样 `mask = cap - 1` 可以用位与 `& mask` 代替取模 `% cap`，更快
- 线性探测：冲突时检查下一个位置 `i = (i+1) & mask`
- 环形：到达末尾后回到开头
- 不扩容：初始化时预分配 `2 * expected + 16` 的空间，保证负载因子 < 50%

2. FNV-1a 哈希函数

```c
static uint64_t hash_bytes(const void *ptr, uint64_t len) {
    const uint8_t *p = ptr;
    uint64_t h = 14695981039346656037ull;   // FNV offset basis
    for (uint64_t i = 0; i < len; i++) {
        h ^= p[i];                            // XOR 当前字节
        h *= 1099511628211ull;                // 乘以 FNV prime
    }
    return h;
}
```

FNV-1a 简单、快速、分布均匀。对于 token 字符串查找足够好。

3. 动态数组（Dynamic Array）

```c
typedef struct {
    int *v;
    int len;
    int cap;
} ds4_tokens;

static void token_vec_push(token_vec *tv, int token) {
    if (tv->len == tv->cap) {                // 容量不够
        tv->cap = tv->cap ? tv->cap * 2 : 64; // 翻倍（首次用 64）
        tv->v = xrealloc(tv->v, (size_t)tv->cap * sizeof(tv->v[0]));
    }
    tv->v[tv->len++] = token;               // 追加元素
}
```

这是 C 语言中最常见的"可增长数组"模式：

- `len`：当前元素数
- `cap`：分配的容量
- 增长策略：翻倍（amortized O(1) 插入）

4. UTF-8 字符串操作

ds4.c 包含 UTF-8 辅助函数：

```c
// 从首字节判断 UTF-8 字符长度
static int utf8_len_from_first_byte(uint8_t c) {
    if (c < 0x80) return 1;         // ASCII: 1 字节
    if ((c & 0xE0) == 0xC0) return 2; // 110xxxxx: 2 字节
    if ((c & 0xF0) == 0xE0) return 3; // 1110xxxx: 3 字节
    if ((c & 0xF8) == 0xF0) return 4; // 11110xxx: 4 字节
    return 1;                          // 无效字节，当作 1 字节
}
```

UTF-8 编码中首字节的前导 1 的数量指示了该字符的字节数。这个技巧在处理多语言文本时很常用。

LLM 知识点

1. 什么是 Token

LLM 不直接处理字符，而是处理 **token**——文本的子词单元：

```
"Hello world" → [15496, 995]        (2 个 token)
"Hello 世界"  → [15496, 995, 105396] (3 个 token)
```

一个 token 可以是：

- 一个完整单词（常见英文词）
- 一个子词（"un"+"believable"）
- 一个字符（罕见词被拆成单字符）
- 一个特殊标记（`<eos>`, `<bos>` 等）

DeepSeek V4 Flash 有 129,280 个 token。

2. BPE（Byte Pair Encoding）算法

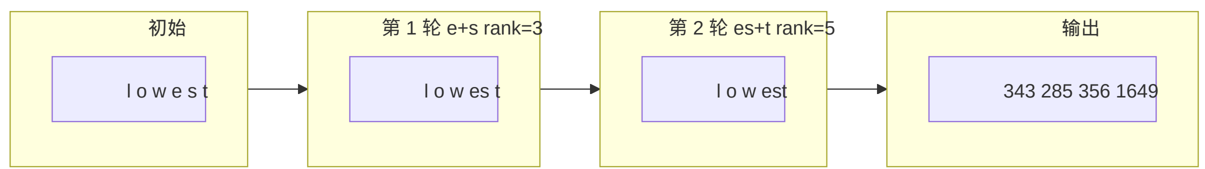
BPE 是最常用的分词算法，工作原理：

训练阶段（已完成，数据存在 GGUF 中）：

1. 从字符级开始，每个字节是一个 token
2. 统计相邻 token 对的出现频率
3. 把最高频的 token 对合并为新 token

4. 重复直到达到目标词表大小
5. 记录合并顺序（rank，越小越优先）

编码阶段（ds4.c 中的 `bpe_emit_piece`）：



核心代码（行 13632-13698）：

```
for (;;) {
    int best_i = -1;
    int best_rank = INT32_MAX;

    // 找到 rank 最小的相邻对
    for (int i = 0; i + 1 < n_sym; i++) {
        int rank = bpe_rank(vocab, &sym[i], &sym[i + 1]);
        if (rank >= 0 && rank < best_rank) {
            best_rank = rank;
            best_i = i;
        }
    }

    if (best_i < 0) break;   // 没有可合并的对

    // 合并 best_i 和 best_i+1
    // ... 拼接字符串，移除 best_i+1 ...

    n_sym--;
}
```

3. GPT-2 字节级 BPE

标准 BPE 需要处理一个问题：输入文本包含所有可能的字节值（0-255），但其中有些不是合法的 Unicode 字符。

GPT-2 的解决方案：字节到 **Unicode** 码点的映射

```c
static uint32_t gpt2_byte_to_codepoint(uint8_t b) {
    // 可打印 ASCII 和部分 Latin-1 字符直接使用
    if ((b >= 33 && b <= 126) || (b >= 161 && b <= 172) || (b >= 174)) {
        return b;
    }
    // 不可打印字节映射到 256 开始的码点
    // ...
}
```

这样每个字节都能表示为一个合法的 Unicode 字符，BPE 可以在"字符级"上操作。

4. 预分词（Pre-tokenization）

在 BPE 合并之前，文本先按规则拆分成片段（行 13808-13876）：

```
"I love C99! 你好"
→ ["I", " love", " C", "99", "!", " 你", "好"]
    ↑       ↑       ↑    ↑    ↑
    字母    空格+字母  数字  标点  CJK 字符
```

预分词规则（JoyAI regex）：

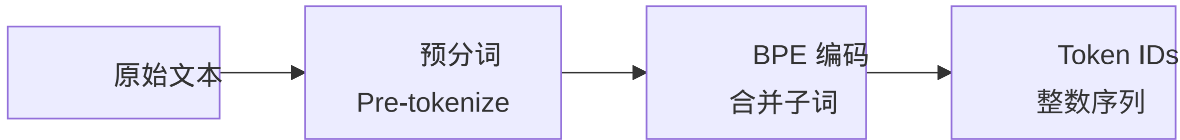
- 数字最多 3 位一组
- CJK 字符单独处理
- 标点符号后跟字母
- 空格与后续字符组合

5. Chat 模板

LLM 不是直接处理原始文本，而是用特殊 token 标记对话结构：

```
<｜begin_of_sentence｜>          ← BOS（句首）
System message here
<｜User｜>                       ← 用户消息开始
What is Redis?
<｜Assistant｜>💭                ← 助手回复开始 + 思考模式
Let me think about this...
📤                               ← 思考结束
Redis is...
<｜end_of_sentence｜>            ← EOS（句尾）
```

ds4.c 的 `encode_chat_prompt` 函数（行 13940-13961）负责组装这些 token。



---

2. 采样策略

> 模型输出 129280 个原始分数（logits），但用户只需要一个词。采样策略决定怎么从 12 万个候选项中挑出一个——太确定则输出呆板，太随机则输出胡言。今天学习 softmax、temperature、top-k/p 在确定性和多样性之间找平衡。

设计决策推导：从 **logits** 到 **token**

```
问题 1：模型输出 129280 个 logits（原始分数），怎么变成概率？
  └─ 方案：softmax — exp(x) / Σexp(x)，把任意实数映射到 [0,1] 且总和为 1
  └─ 副作用：exp(大数) 会溢出 → 减去最大值后再 exp（数值稳定技巧）

问题 2：直接按概率随机采样（轮盘赌），输出太随机了怎么办？
  └─ 方案 A：Temperature — logits / T 后再 softmax
     T→0: 趋近 argmax（完全确定），T→∞: 趋近均匀分布（完全随机）

问题 3：即使调了温度，低概率的 token 仍可能被选中，输出不稳定？
  └─ 方案 B：Top-K — 只保留概率最高的 K 个 token，其余归零
  └─ 问题：K 是固定的，模型确定时 K 个太多，不确定时 K 个太少

问题 4：能不能让截断阈值自适应？
  └─ 方案 C：Top-P（核采样）— 按概率降序排列，累加到 P 后截断
     模型确定时：前几个 token 就凑够 P → 保留少 → 更确定
     模型不确定时：需要很多 token 才凑够 P → 保留多 → 更多样

问题 5：Top-P 可能保留一些"和最高概率差距太大"的 token？
  └─ 方案 D：Min-P — 只保留概率 >= 最高概率 × min_p 的 token
     例：最高 0.5, min_p=0.1 → 保留所有 >= 0.05 的 → 砍掉尾部噪音

最终组合：Temperature 缩放 → Top-P + Min-P 双重过滤 → 轮盘赌采样
```

C 知识点

1. 浮点运算技巧

数值稳定的 softmax：

```
// 先找最大值
float max_logit = DS4_NEG_INF;
for (uint32_t i = 0; i < n_vocab; i++)
    if (logits[i] > max_logit) max_logit = logits[i];

// 减去最大值再取指数（防止溢出）
const float p = expf((v - max_logit) / temperature);
```

为什么减去最大值？`exp(大数)` 会溢出变成 `+inf`，`exp(大数 - 最大数)` 的最大值是 `exp(0) = 1`，安全。

float 精度注意事项：

- ds4.c 用 `double` 累加（`double ss = 0.0`），最后转回 `float`，减少累加误差
- `isfinite(v)` 检查排除 NaN 和无穷大
- `DS4_NEG_INF = -1e30f`（不是 `-INFINITY`），避免 NaN 传播

2. 插入排序 vs qsort

ds4.c 在 top-k 选择中用了两种排序：

插入排序（维护 **top-k** 数组）：

```
// 边扫描边维护已排序的 top-k，最多 1024 个元素
int j = n < top_k ? n++ : n - 1;
while (j > 0 && vals[j - 1] < v) {
    vals[j] = vals[j - 1];    // 后移
    ids[j] = ids[j - 1];
    j--;
}
vals[j] = v;
ids[j] = (int)i;
```

为什么用插入排序？

- 每次 only 移动最多 k 个元素
- 对于 k=1024，比 qsort 全排序快得多
- 时间复杂度：O(n × k)，当 k << n 时比 O(n log n) 好

qsort（全词表排序）：

```
// 当 top_k <= 0 时，对整个词表排序
qsort(cand, n, sizeof(cand[0]), sample_candidate_cmp_desc);
```

标准库的快速排序，O(n log n)。

3. PRNG — xorshift64

```
static uint64_t sample_rng_next(uint64_t *state) {
    uint64_t x = *state;
    if (x == 0) x = 0x9e3779b97f4a7c15ULL;   // 黄金比例常数
    x ^= x >> 12;
    x ^= x << 25;
    x ^= x >> 27;
    *state = x;
    return x * 0x2545f4914f6cdd1dULL;
}

static float sample_rng_f32(uint64_t *state) {
    const uint64_t x = sample_rng_next(state);
    return (float)((x >> 40) & 0xffffffu) / 16777216.0f;  // [0, 1)
}
```

- xorshift64：3 次移位异或 + 一次乘法，非常快
- 零状态保护：如果 state 为 0，替换为黄金比例常数
- 浮点转换：取高 24 位，除以 2^24，得到 [0, 1) 均匀分布

4. union 类型转换（比较函数）

```
static int sample_candidate_cmp_desc(const void *a, const void *b) {
    const sample_candidate *ca = a;
    const sample_candidate *cb = b;
    return (cb->logit > ca->logit) - (cb->logit < ca->logit);
}
```

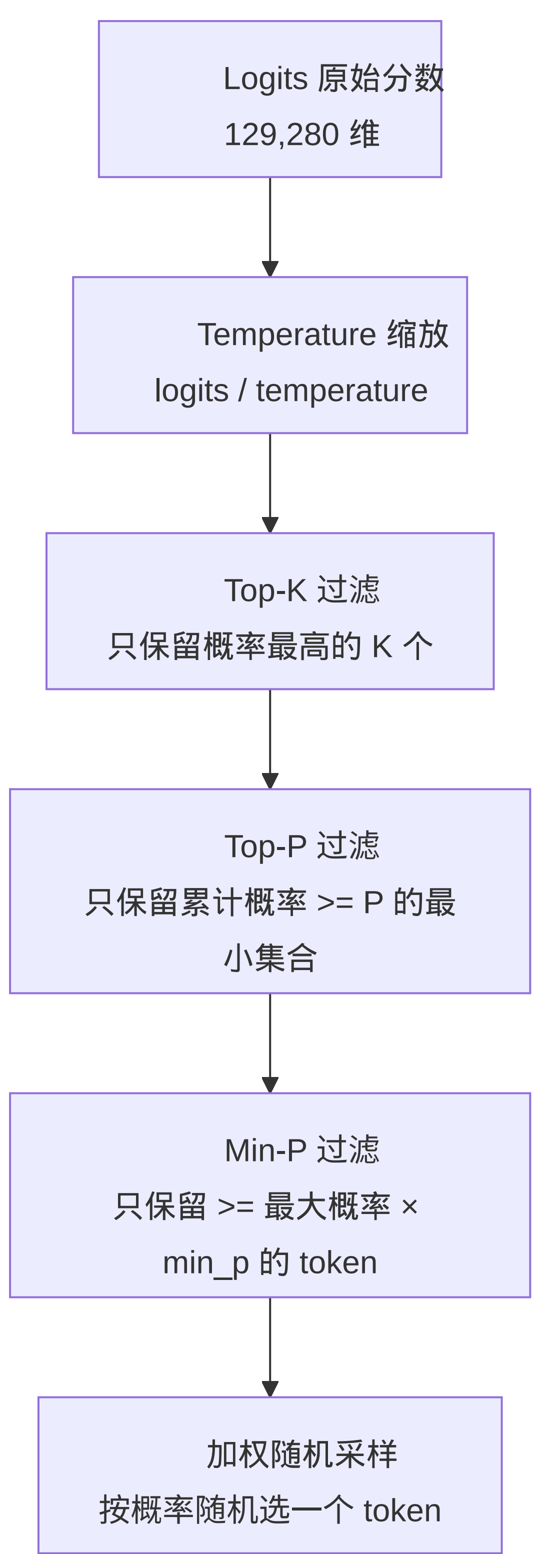
这是无分支的三路比较：

- `a > b` 返回 -1
- `a == b` 返回 0
- `a < b` 返回 1 比 `if/else` 更简洁，编译器可以生成无分支代码。

LLM 知识点

1. 采样算法概览

推理的最后一步：从 129,280 个 token 的概率分布中选择下一个 token。



2. Temperature

```
const float p = expf((v - max_logit) / temperature);
```

- **temperature = 0**：贪心解码（argmax），总是选概率最高的 token。输出确定、无聊
- **temperature < 1**：增强高概率 token 的优势，输出更集中
- **temperature = 1**：原始分布，不做缩放
- **temperature > 1**：拉平分布，输出更多样、更随机

DSML 结构性温度覆盖

生成工具调用时，服务器对 DSML 结构性 token 强制 temperature=0，只对参数载荷使用正常采样温度：

```
// 在 decode 循环中，每个 token 步骤检查 DSML 解码状态
if (dsml_decode_state_is_tool(state) &&
    !dsml_decode_state_uses_payload_sampling(state)) {
    // 标签、属性名、JSON 标点 → 强制 temperature=0
    temperature = 0;
}
// string_body 和 json_string 状态 → 使用请求的正常温度
```

这确保了工具调用语法始终可解析（确定性），同时文件内容、编辑文本等长载荷保持多样性（避免重复）。

贪心解码用于回归测试

temperature=0 的确定性使得它可以用于回归测试。ds4 的长上下文回归测试（`test_long_story_fact_recall`）用贪心解码验证模型的长程记忆能力：

- 用 Python 脚本生成虚构故事，在 190 个场景中嵌入 16 个人名-数字对应关系
- 加入干扰数字（年龄、价格等）测试模型的分辨能力
- 数字用英文拼写（"thirty-four"），模型需转换为数字（"34"）
- 采样参数：`temperature=0.0, top_k=0, top_p=1.0, frequency_penalty=0.0`
- 输出格式固定为 `Name=number`，解析验证每个事实

相比之前的采样文本子串匹配（因随机性不稳定），贪心解码让测试完全确定性，只需 350 token 即可验证 16 个事实。

3. Top-K 采样

```c
if (top_k > 0) {
    // 维护概率最高的 top_k 个 token（插入排序）
    for (uint32_t i = 0; i < n_vocab; i++) {
        if (n == top_k && v <= vals[n-1]) continue;  // 太低，跳过
        // 插入到正确位置
    }
}
```

- `top_k = 50`：只从概率最高的 50 个 token 中选
- 防止选到极低概率的 token（"幻觉"）
- ds4.c 限制 top_k 最大 1024

4. Top-P（Nucleus）采样

```c
float filtered_sum = 0.0f;
for (int i = 0; i < n; i++) {
    filtered_sum += probs[i];
    filtered++;
    if (filtered_sum / sum >= top_p) break;  // 累计概率够了
}
```

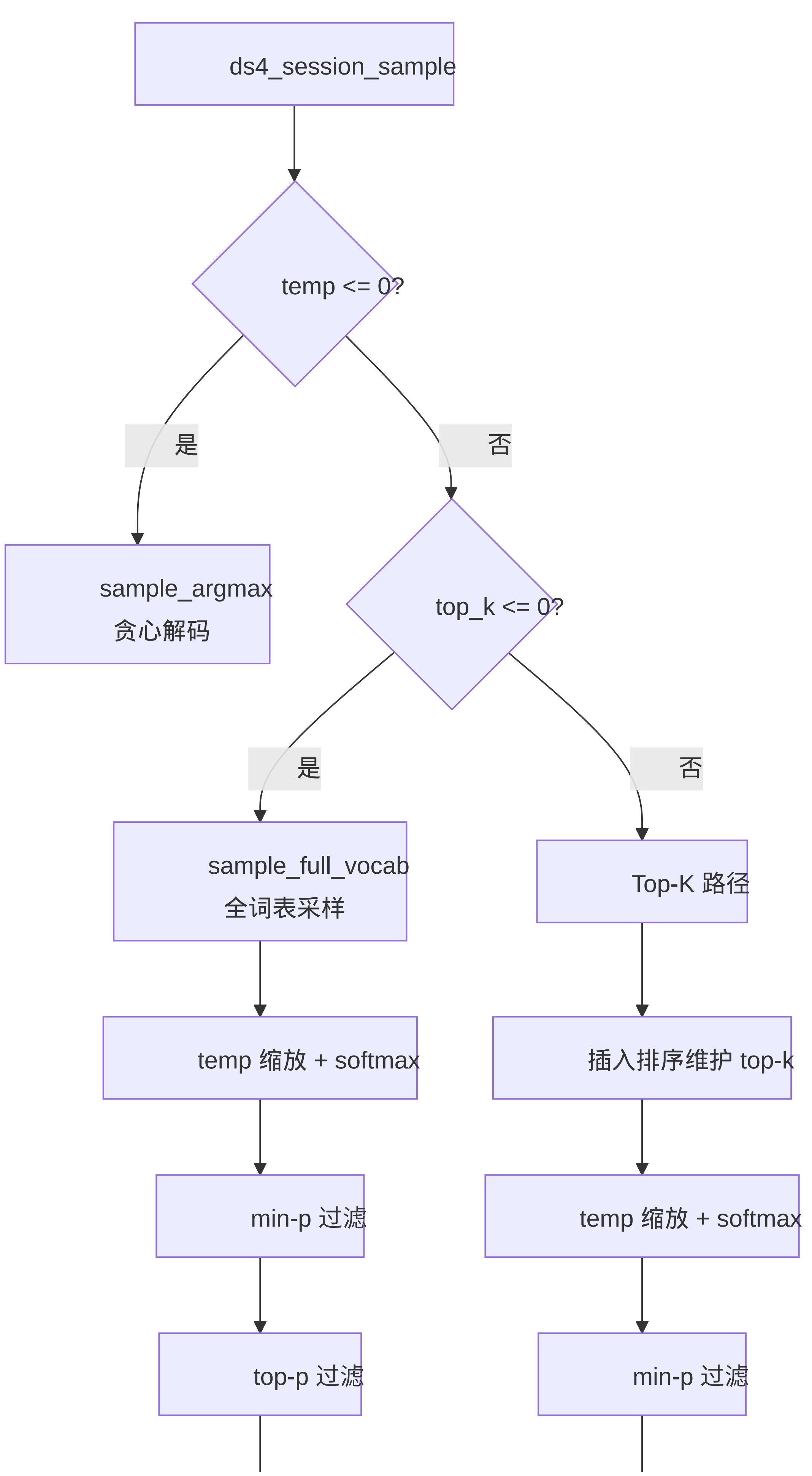
- `top_p = 0.9`：选最少的高概率 token，使累计概率 >= 90%
- 比 top_k 更灵活：高确定性时选少量 token，不确定时选更多
- 两者可以组合：先 top-k，再 top-p

5. Min-P 采样

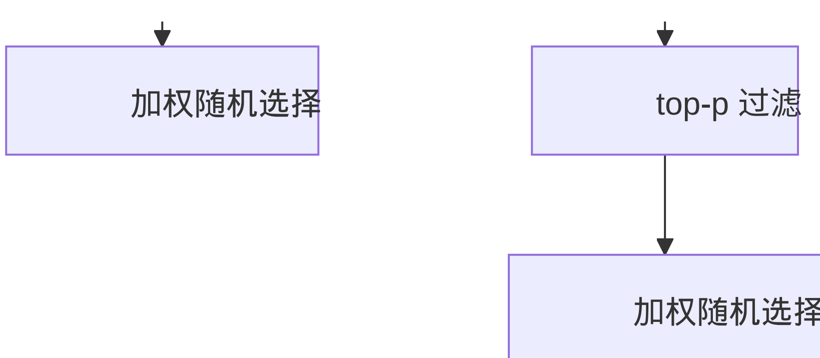
```c
const float min_prob = (probs[0] / sum) * min_p;
for (int i = 0; i < n; i++) {
    float p = probs[i] / sum;
    if (i > 0 && p < min_prob) break;  // 概率太低，排除
}
```

- `min_p = 0.05`：只保留概率 >= 最大概率 × 5% 的 token（ds4 现在默认启用，`DS4_DEFAULT_MIN_P=0.05`，同时 `top_p` 保持 1.0）
- 相对阈值：不是固定值，而是相对于最高概率
- 适合高确定性场景（最高概率 99% 时只保留 > 4.95% 的）
- CLI、server、ds4-eval 均使用 `DS4_DEFAULT_MIN_P` 作为默认值，thinking 模式的固定采样器也用 `min_p=0.05`

6. 采样流程图



ds4_session_sample
temp <= 0?
是
否
sample_argmax
贪心解码
top_k <= 0?
是
否
sample_full_vocab
全词表采样
Top-K 路径
temp 缩放 + softmax
插入排序维护 top-k
min-p 过滤
temp 缩放 + softmax
top-p 过滤
min-p 过滤



反向链接

/glossary/bpe

/glossary/hash-table

/glossary/prng

/glossary/softmax

端到端推理流程

Part 3: 模型架构

Part 2: 练习

---

1. BPE 分词器

练习 1：哈希表容量计算

题目：`table_init` 用 `next_pow2(expected * 2 + 16)` 计算容量。DeepSeek V4 Flash 有 129,280 个 token。

1. `token_to_id` 表的容量是多少？
2. 负载因子（used/cap）是多少？
3. 为什么负载因子要保持在 50% 以下？

参考答案

1. `next_pow2(129280 * 2 + 16)` = `next_pow2(258576)` = 262144 = 2^18
2. 129280 / 262144 ≈ 49.3%
3. 开放寻址哈希表的性能随负载因子增加而恶化。超过 50% 后，冲突增多，探测链变长，查找从 O(1) 退化到接近 O(n)。50% 以下能保证大部分查找只需 1-2 次探测。

练习 2：线性探测追踪

题目：假设哈希表 `cap=8`（mask=7），当前状态如下：

```
slot:  0    1    2    3    4    5    6    7
used:  F    T    T    F    T    F    F    T
key:   -   "a"  "b"   -   "c"   -    -   "d"
```

现在要插入 key="e"，hash("e") & 7 = 1。请追踪 `table_put` 的执行过程。

参考答案

```
i = 1
slot[1].used = true, key = "a" ≠ "e" → i = (1+1) & 7 = 2
slot[2].used = true, key = "b" ≠ "e" → i = (2+1) & 7 = 3
slot[3].used = false → 插入这里!
slot[3] = {key="e", value=..., used=true}
```

线性探测从 hash 位置开始，逐个检查，直到找到空位或匹配的 key。

练习 3：BPE 合并模拟

题目：假设词表中有以下 token 和 merge rank：

```
Token IDs: 'l'=0, 'o'=1, 'w'=2, 'e'=3, 's'=4, 't'=5
Merge ranks: "e s"=1, "es t"=2, "lo"=3, "low"=5, "lo w"=4
```

请模拟 `bpe_emit_piece` 处理输入 "lowest" 的过程。

参考答案

```
初始: ['l', 'o', 'w', 'e', 's', 't']

第 1 轮: 扫描所有相邻对
  'l'+'o': "l o" → 无 rank
  'o'+'w': "o w" → 无 rank
  'w'+'e': "w e" → 无 rank
  'e'+'s': "e s" → rank=1 ★ 最小
  's'+'t': "s t" → 无 rank
  合并 'e'+'s' → 'es'
  → ['l', 'o', 'w', 'es', 't']

第 2 轮:
  'l'+'o': 无 rank
  'o'+'w': 无 rank
  'w'+'es': 无 rank
  'es'+'t': "es t" → rank=2 ★
  合并 'es'+'t' → 'est'
  → ['l', 'o', 'w', 'est']

第 3 轮:
  'l'+'o': "lo" → rank=3 ★
  'o'+'w': 无 rank
  'w'+'est': 无 rank
  合并 'l'+'o' → 'lo'
  → ['lo', 'w', 'est']

第 4 轮:
  'lo'+'w': "lo w" → rank=4 ★
  'w'+'est': 无 rank
  合并 'lo'+'w' → 'low'
  → ['low', 'est']

第 5 轮:
  'low'+'est': 无 rank
  无可合并对 → 停止

最终: ['low', 'est'] → 查表得到 token IDs
```

练习 4：理解 GPT-2 字节编码

题目：`gpt2_byte_to_codepoint` 把字节 0x20（空格）映射到码点 286（Ġ）。请解释为什么需要这个映射，以及解码时如何还原。

> 参考答案
>
> BPE 的 token 字符串存储在 GGUF 中作为 UTF-8 文本。但原始字节 0x00-0x1F 是控制字符（换行、制表等），不是"可见"的 Unicode 字符。如果 token 字符串直接包含这些字节，会破坏字符串处理。
>
> GPT-2 的映射方案：
>
> - 可打印 ASCII（33-126）和部分 Latin-1（161-172, 174-255）直接使用原码点
> - 不可打印的字节映射到 256 开始的码点区域
>
> 空格（0x20=32）不在可打印范围内（33-126），所以被映射到 256 + N。在 0-255 中，跳过可打印字符后，0x20 是第 30 个不可打印字节，所以映射到 256 + 30 = 286 = 'Ġ'（带点 G）。
>
> 解码时 `gpt2_codepoint_to_byte` 做逆映射：286 → 0x20 → ' '（空格）。

练习 5：动态数组增长

题目：`token_vec_push` 的增长策略是翻倍。如果依次 push 1000 个 token：

1. `cap` 会经历哪些值？
2. 总共会发生多少次 realloc？
3. 翻倍策略的均摊时间复杂度是多少？

参考答案

1. cap 变化：0 → 64 → 128 → 256 → 512 → 1024 → 2048
2. 6 次 realloc（64, 128, 256, 512, 1024, 2048）
3. 均摊 O(1)。证明：N 次 push 的总拷贝量 = 64 + 128 + 256 + ... < 2N。平均每次 push 拷贝 < 2 个元素，是常数。

今日学习检查清单

- [ ] 能解释开放寻址哈希表的工作原理（线性探测）
- [ ] 理解为什么哈希表容量要是 2 的幂（用 & mask 代替 % cap）
- [ ] 能手动模拟 BPE 合并过程
- [ ] 理解 GPT-2 字节编码的目的（为什么空格变成 Ġ）
- [ ] 能描述预分词的作用
- [ ] 理解 Chat 模板中特殊 token 的作用
- [ ] 能追踪从文本到 token ID 的完整调用链
- [ ] 理解动态数组翻倍增长的均摊复杂度

延伸挑战

挑战 1（中级）：追踪真实文本的分词过程

运行 `./ds4` 并输入一段包含中英文混合的文本。在 ds4.c 的 `bpe_emit_piece` 函数中加一个 `fprintf(stderr, ...)` 打印每次合并的 token 对，观察 BPE 如何处理中文和数字。

挑战 2（高级）：实现前缀匹配分词

分词器需要支持 special token 的最长匹配。写一个简化版：给定一个有序的 token 列表和输入字符串，用二分查找实现最长前缀匹配，时间复杂度 O(log n)。

2. 采样策略

练习 1：数值稳定的 softmax

题目：假设 logits = [3.0, 1.0, -1.0, 5.0]，temperature = 1.0。请手动计算：

1. 减去最大值后的值
2. 每个值的 exp
3. 概率分布

参考答案

```
原始:     [3.0,  1.0,  -1.0,  5.0]
最大值:  5.0
减最大值: [-2.0, -4.0, -6.0,  0.0]
exp:      [0.135, 0.018, 0.0025, 1.0]
总和:     1.1555
概率:     [0.117, 0.016, 0.002, 0.865]
```

token 3（logit=5.0）有 86.5% 的概率被选中。

练习 2：Temperature 效果

题目：同样的 logits = [3.0, 1.0, -1.0, 5.0]，分别计算 temperature = 0.5 和 temperature = 2.0 时的概率分布。

参考答案

T=0.5（更集中）：

```
减最大值/T: [-4.0, -8.0, -12.0, 0.0]
exp:        [0.018, 0.000335, 0.0000061, 1.0]
概率:       [0.018, 0.0003, 0.000006, 0.982]
```

最高概率的 token 从 86.5% 提升到 98.2%。

T=2.0（更均匀）：

```
减最大值/T: [-1.0, -2.0, -3.0, 0.0]
exp:        [0.368, 0.135, 0.0498, 1.0]
概率:       [0.231, 0.085, 0.031, 0.628]
```

分布变平，其他 token 的概率增加了。

练习 3：Top-P 模拟

题目：假设排序后的概率分布为：

```
Token   概率      累计
D      0.50     0.50
A      0.25     0.75
B      0.15     0.90
C      0.08     0.98
E      0.02     1.00
```

`top_p = 0.9` 时，哪些 token 会被保留？

参考答案

```
D: 0.50 → 累计 0.50 < 0.90 ✓ 保留
A: 0.25 → 累计 0.75 < 0.90 ✓ 保留
B: 0.15 → 累计 0.90 >= 0.90 ✓ 保留（包含达到阈值的那一个）
C: 0.08 → 累计 0.98 > 0.90 ✗ 不保留
E: 0.02 → 不保留
```

保留 D, A, B（累计 90%）。这三个 token 的概率会被重新归一化后用于随机采样。

练习 4：理解 xorshift64 PRNG

题目：`sample_rng_next` 的初始状态为 `state = 1`。请手动执行一次：

```c
uint64_t x = 1;
x ^= x >> 12;
x ^= x << 25;
x ^= x >> 27;
return x * 0x2545f4914f6cdd1dULL;
```

只需要描述每一步的变换类型，不需要精确计算 64 位结果。

参考答案

```
初始:   x = 1 (0x0000000000000001)
步骤1:  x ^= x >> 12     — 右移 12 位后异或，高位不变，低位被高位影响
步骤2:  x ^= x << 25     — 左移 25 位后异或，低位不变，高位被低位影响
步骤3:  x ^= x >> 27     — 右移 27 位后异或
步骤4:  x * 0x2545...    — 乘法混合，增加随机性
```

三次移位异或确保所有 bit 都被充分混合。这个设计的关键是三个移位量（12, 25, 27）经过数学验证，能产生良好的统计特性（完整周期、均匀分布）。

练习 5：采样策略选择

题目：在以下场景中，你会选择什么采样参数？

1. 代码生成（需要精确、确定）
2. 创意写作（需要多样性）
3. 事实问答（需要平衡准确和自然）
4. 批量测试（需要完全可复现）

> 参考答案
>
> 1. `temperature=0`（贪心），或 `temperature=0.2, top_p=0.9`：代码需要精确，低温度确保选最可能的 token
> 2. `temperature=0.8-1.0, top_p=0.95, min_p=0.05`：保持多样性，但避免极端低概率 token
> 3. `temperature=0.5-0.7, top_p=0.9`：偏向正确答案，但保留一定变化
> 4. `temperature=0, seed=固定值`：完全确定性，每次运行结果完全相同

今日学习检查清单

- [ ] 能解释 softmax 为什么需要减去最大值
- [ ] 理解 temperature 对概率分布的影响
- [ ] 能手动模拟 top-k 和 top-p 过滤
- [ ] 理解 min-p 的相对阈值设计
- [ ] 能描述完整的采样调用链
- [ ] 理解插入排序在 top-k 中的应用
- [ ] 能解释 xorshift64 PRNG 的工作原理
- [ ] 理解无分支三路比较的技巧

延伸挑战

挑战 1（中级）：实现 **Min-P** 采样

Top-P 按累积概率截断，Min-P 按相对阈值截断。在 `sample_top_p_min_p` 的基础上，用你自己的话解释 `min_p` 参数如何与 `top_p` 协同工作，然后构造一个例子：top_p 保留的 token 比预期多，min_p 如何进一步裁剪。

挑战 2（高级）：对比采样策略的输出分布

写一个脚本，用 ds4-server 的 API 分别以 temperature=0.1、0.7、1.5 生成 20 次相同的 prompt。统计每个温度下输出长度的方差和词汇多样性（unique token / total tokens），验证"温度越高输出越多样"的直觉。

Part 2: 代码走读

---

1. BPE 分词器

追踪 **1**：分词完整调用链

从用户输入到 token ID 序列：

```
用户输入: "Hello world"
    │
    ▼
ds4_tokenize_text(e, "Hello world", &tokens)    [行 13963]
    │
    ▼
bpe_tokenize_text(&vocab, "Hello world", &out)  [行 13808]
    │
    │  预分词：按 JoyAI regex 拆分
    ├── "Hello" → bpe_emit_piece(...)           [行 13632]
    └── " world" → bpe_emit_piece(...)
          │
          ▼ 每个 piece
    bpe_emit_piece(vocab, {" world", 6}, out)    [行 13632]
          │
          ├── byte_encode(" world") → 编码为 GPT-2 字节字符
          │     每个字节 → Unicode 码点
          │     ' '→286, 'w'→119, 'o'→111, 'r'→114, 'l'→108, 'd'→100
          │     → UTF-8 字符串: "Ġworld"（Ġ 是编码后的空格）
          │
          ├── 拆分为初始符号（每个 UTF-8 字符一个符号）
          │     ['Ġ', 'w', 'o', 'r', 'l', 'd']
          │
          ├── BPE 合并循环：
          │     for(;;) {
          │       找 rank 最小的相邻对
          │       找不到 → break
          │       合并该对
          │     }
          │     例如: 'w'+'o'→rank=321 → 合并 → ['Ġ', 'wo', 'r', 'l', 'd']
          │     继续: 'wo'+'r'→rank=... → ...
          │     最终: ['Ġworld']  （或多个子词）
          │
          └── 查表获取 token ID
                table_get(token_to_id, "Ġworld") → 995
                token_vec_push(out, 995)
```

追踪 2：哈希表操作

初始化

```
vocab_load()                              [行 13888]
  │
  ├── model_get_array(model, "tokenizer.ggml.tokens", &tokens)
  │     从 GGUF 元数据获取 token 字符串数组的位置
  │
  ├── vocab->token = xcalloc(n_vocab, sizeof(ds4_str))
  │     分配 token 字符串数组
  │
  ├── table_init(&vocab->token_to_id, tokens.len)   [行 13438]
  │     cap = next_pow2(129280 * 2 + 16) = 262144
  │     entry = xcalloc(262144, sizeof(str_i32_entry))
  │
  └── for i = 0..n_vocab-1:
        cursor_string(&c, &vocab->token[i])   读 token 字符串（指向 mmap）
        table_put(&token_to_id, token[i], i)   插入哈希表
```

查找过程

```
table_get(&token_to_id, "Ġworld", 6, &id)   [行 13467]
  │
  ├── hash_bytes("Ġworld", 6) → 某个 64-bit 哈希值
  ├── i = hash & (cap - 1)     → 起始槽位
  │
  └── 线性探测：
        slot[i].used == true
        slot[i].key.len == 6 && memcmp(key.ptr, "Ġworld", 6) == 0
        → 找到！返回 id = 995
```

追踪 3：BPE 合并的内存管理

`bpe_emit_piece` 中的内存操作：

```c
// 初始符号数组（动态增长）
int cap_sym = 32;
owned_str *sym = xcalloc(cap_sym, sizeof(sym[0]));

// 合并时：
// 1. 分配新字符串
merged.ptr = xmalloc(sym[i].len + sym[i+1].len);
memcpy(...);  // 拼接

// 2. 释放旧的两个字符串
free(sym[i].ptr);
free(sym[i+1].ptr);

// 3. 替换
sym[i] = merged;

// 4. 前移后续元素
for (int j = i+1; j+1 < n_sym; j++)
    sym[j] = sym[j+1];
n_sym--;

// 最终：释放所有符号和编码字符串
for (int i = 0; i < n_sym; i++) free(sym[i].ptr);
free(sym);
free(encoded);
```

注意：`bpe_emit_piece` 是分词阶段调用的，不在推理热循环中，所以这里的 `xmalloc` / `free` 不会触发分配守卫。

追踪 4：Token 解码（ID → 文本）

```
ds4_token_text(e, 995, &len)              [行 14138]
  │
  ├── vocab->token[995] = {"Ġworld", 6}   从 token 数组获取字符串
  │
  ├── vocab_token_is_literal_special()?    检查是否是特殊 token
  │     （包含 U+FF5C 全角竖线的是特殊 token）
  │     → 否，执行 GPT-2 字节解码
  │
  └── 逐 UTF-8 字符解码：
        cp = utf8_decode_one(s.ptr, ...)   读一个 Unicode 码点
        b = gpt2_codepoint_to_byte(cp)     码点 → 原始字节
        out[n++] = (char)b

        'Ġ' → 码点 286 → 字节 0x20 → ' '  （空格的 GPT-2 编码）
        'w' → 码点 119 → 字节 0x77 → 'w'
        ...
        → " world"
```

追踪 5：Chat 模板组装

```
ds4_encode_chat_prompt(e, "You are helpful.", "Hello", DS4_THINK_HIGH, &out)
    │                                                            [行 13940]
    │
    ├── token_vec_push(out, bos_id)          <｜begin▁of▁sentence｜>
    │
    ├── bpe_tokenize_text(vocab,             "You are helpful."
    │     "You are helpful.", out)           → [token_ids...]
    │
    ├── token_vec_push(out, user_id)         <｜User｜>
    │
    ├── bpe_tokenize_text(vocab,             "Hello"
    │     "Hello", out)                      → [token_ids...]
    │
    ├── token_vec_push(out, assistant_id)    <｜Assistant｜>
    │
    └── token_vec_push(out, think_start_id)  💭（思考模式开启）

最终 token 序列:
[BOS] You are helpful. <｜User｜> Hello <｜Assistant｜> 💭
```

2. 采样策略

追踪 1：随机数生成器（xorshift64）

PRNG 状态与演进

```
// 行 14907: SplitMix64 状态转移 + 乘法输出混合
static uint64_t sample_rng_next(uint64_t *state) {
    uint64_t x = *state;
    if (x == 0) x = 0x9e3779b97f4a7c15ULL;  // 黄金比例常数，防止全零种子
    x ^= x >> 12;     // 右移异或
    x ^= x << 25;     // 左移异或
    x ^= x >> 27;     // 再右移异或
    *state = x;
    return x * 0x2545f4914f6cdd1dULL;       // SplitMix64 输出混合
}
```

三步异或移位（xorshift）的几何意义：

```
初始:  1011001110101011...
>>12:  0000000000001011001110101011
XOR:   1011001110100000...          ← 高 12 位不变，低位被打乱
<<25:  ...被打乱的低位左移25位填充高位
>>27:  再次从高位取回低位
```

浮点均匀分布

```
// 行 14917: 将 64-bit 随机数映射到 [0, 1) 区间
static float sample_rng_f32(uint64_t *state) {
    const uint64_t x = sample_rng_next(state);
    return (float)((x >> 40) & 0xffffffu) / 16777216.0f;
    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^
    //      取高 24 位 (0 ~ 2^24-1)       除以 2^24 = 16777216
}
```

为什么取高 24 位？`float` 的尾数只有 23 位 + 1 隐含位 = 24 位精度。取更多位无法提高分辨率。

追踪 2：Argmax 贪心采样

```
// 行 14874: 线性扫描找最大 logit 对应的 token ID
static int sample_argmax(const float *logits, uint32_t n_vocab) {
    int best = 0;
    float best_v = DS4_NEG_INF;      // -1e30f 作为初始哨兵
    for (uint32_t i = 0; i < n_vocab; i++) {
        const float v = logits[i];
        if (v > best_v) {
            best_v = v;
            best = (int)i;
        }
    }
    return best;
}
```

```
logits[0] = -2.3   best_v=-1e30  → 更新: best=0, best_v=-2.3
logits[1] = 5.1    best_v=-2.3   → 更新: best=1, best_v=5.1
logits[2] = 3.7    best_v=5.1    → 不更新
...
logits[N]= 8.2     best_v=5.1    → 更新: best=N, best_v=8.2

返回 best = N（概率最大的 token）
```

另有通用版本 `argmax_f32`（行 8019），返回 `uint64_t` 索引，用于非采样场景。

追踪 3：降序排序比较器

```
// 行 14922-14926: 采样候选项结构
typedef struct {
    int id;         // token ID
    float logit;    // 原始 logit 值
    float prob;     // softmax 后的概率
} sample_candidate;

// 行 14928: qsort 降序比较器
static int sample_candidate_cmp_desc(const void *a, const void *b) {
    const sample_candidate *ca = a;
    const sample_candidate *cb = b;
    return (cb->logit > ca->logit) - (cb->logit < ca->logit);
    //     ^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^
    //     b > a 时为 1               b < a 时为 1
    //     组合结果: b>a 返回 1（b 排前面 → 降序）
}
```

无分支三路比较技巧：`(b > a) - (b < a)` 等价于：

```
b > a: 返回  1  (b 应排在 a 前面 → 降序)
b < a: 返回 -1  (a 应排在 b 前面)
b = a: 返回  0
```

避免了 `if-else` 分支，对现代 CPU 流水线友好。

追踪 4：全词表温度采样

完整流水线

```
// 行 14934: 完整采样流水线
static int sample_full_vocab(
        const float *logits,    // 模型输出的 129280 个 logit
        uint32_t     n_vocab,   // 129280
        float        temperature,
        float        top_p,
        float        min_p,
        uint64_t    *rng) {     // 随机数状态
```

```
logits[129280]
    │
    ▼ 第一步: 找最大值 + 统计有限值个数
    │
    max_logit = max(logits[])
    finite = count(isfinite(logits[i]))
    │
    ▼ 第二步: Softmax + 温度缩放
    │
    for each i:
        if !isfinite(v): skip
        p = expf((v - max_logit) / temperature)
        //                      ^^^^^^^^^  温度 T > 1 使分布更平滑
        //                                  T < 1 使分布更尖锐
    │
    ▼ 第三步: 快速路径 vs 排序路径
    │
    ┌─ top_p >= 1.0? ──→ 快速路径: 无排序，线性扫描采样
    │                      无堆分配，适合大部分场景
    │
    └─ top_p < 1.0  ──→ 排序路径: qsort 降序
                         累积概率直到 ≥ top_p
                         截断并重归一化
```

快速路径详解（top_p >= 1.0 时）

```
// 行 14955: 不排序，直接在线性扫描中采样
double sum = 0.0;
for (uint32_t i = 0; i < n_vocab; i++) {
    const float v = logits[i];
    if (!isfinite(v)) continue;
    const float p = expf((v - max_logit) / temperature);
    candidates[i].prob = p;    // 直接存在大数组中
    sum += p;
}
const float r = sample_rng_f32(rng) * sum;  // 随机阈值
float acc = 0.0f;
for (uint32_t i = 0; i < n_vocab; i++) {
    acc += candidates[i].prob;
    if (acc > r) return (int)i;  // 命中！
}
```

追踪 5：Top-P / Min-P 过滤采样

```c
// 行 15023: 带拓扑筛选的采样
static int sample_top_p_min_p(
        const float *logits,
        uint32_t     n_vocab,
        float        temperature,
        int          top_k,      // 只保留前 k 个候选
        float        top_p,      // 累积概率阈值
        float        min_p,      // 相对最小概率阈值
        uint64_t    *rng)
```

栈上插入排序（零堆分配）

```c
// 行 15041: 固定大小栈数组，避免 malloc
int ids[1024];        // 最多 1024 个候选
float vals[1024];     // 对应的 logit 值
int nk = 0;

for (uint32_t i = 0; i < n_vocab; i++) {
    // 降序插入排序，只保留 top_k
    for (int j = 0; j < nk; j++) {
        if (logits[i] > vals[j]) {
            // 后移腾出位置
            for (int m = MIN(nk, top_k - 1); m > j; m--) {
                vals[m] = vals[m - 1];
                ids[m]  = ids[m - 1];
            }
            vals[j] = logits[i];
            ids[j]  = (int)i;
            if (nk < top_k) nk++;
            goto next_logit;
        }
    }
    // 比所有已有候选都小，但数组未满
    if (nk < top_k) {
        vals[nk] = logits[i];
        ids[nk]  = (int)i;
        nk++;
    }
    next_logit:;
}
```

过滤与采样

```
top-k 候选 (k=50)
    │ softmax → 概率分布 probs[50]
    │
    ▼ Min-P 过滤
    │
    min_prob = (probs[0] / sum) * min_p
    //           ^^^^^^^^       ^^^^^
    //           最高概率        相对阈值
    // 任何 probs[i] < min_prob 的候选被丢弃
    │
    ▼ Top-P 累积截断
    │
    从最高概率开始累加: probs[0] + probs[1] + ...
    当累积概率 >= top_p 时停止
    │
    ▼ 重归一化 + 加权随机采样
    │
    r = rng_f32() * filtered_sum
    线性扫描找到累积值 > r 的 token
    → 返回该 token ID
```

为什么用插入排序而非 **qsort**？

- `top_k` 最多 1024，远小于 129280
- 插入排序对"维护一个小的有序子集"是 O(n×k)，当 k << n 时比 O(n log n) 的 qsort 更快
- 栈上固定数组避免了 `malloc`/`free`，满足热循环零分配约束

Part 3: 模型架构

> Transformer 的核心组件：权重结构、量化、Self-Attention、MoE 混合专家。这是理解"模型怎么算"的关键主题。

涵盖内容

章节	核心主题
1. 公共 API 与权重结构	opaque type、weights_bind、Transformer 权重、LoRA
2. 量化与矩阵运算	位操作、block 布局、非对称量化、matvec
3. 注意力机制	Self-Attention、kv-cache、滑动窗口、MLA、Attention Sink
4. moe 混合专家	路由（hash/top-k）、swiglu、专家前向传播

核心概念

- kv-cache — 注意力历史状态缓存
- moe — 混合专家路由
- quantization — 模型权重压缩
- rmsnorm — 归一化层
- swiglu — MoE 专家的激活函数
- attention-sink — 注意力锚点机制
- mla — 多头潜在注意力（压缩 KV）

前置知识

- Part 1（GGUF 格式、mmap）
- Part 2（softmax）
- 矩阵乘法基础

学习路径

读完本主题后，你将理解：

1. Transformer 权重在内存中的组织方式
2. 量化如何将 32-bit 浮点数压缩到 2-bit
3. Self-Attention 的计算流程和 KV Cache 的作用
4. MoE 如何用"宽度换深度"实现 284B 参数但低计算量

→ 下一步：Part 4: 推理流程

Part 3: 模型架构

> 权重结构、量化、Attention、MoE

---

1. 公共 API 与权重结构

> 284B 参数是什么、存在哪里、怎么找到它们？今天通过 ds4.h 公共 API 和 weights_bind 绑定过程，理解 Transformer 每层的完整权重组成——后续所有计算都是对这些权重的矩阵运算。

C 知识点

1. opaque type（不透明类型）

ds4.h 只暴露类型名，不暴露内部定义：

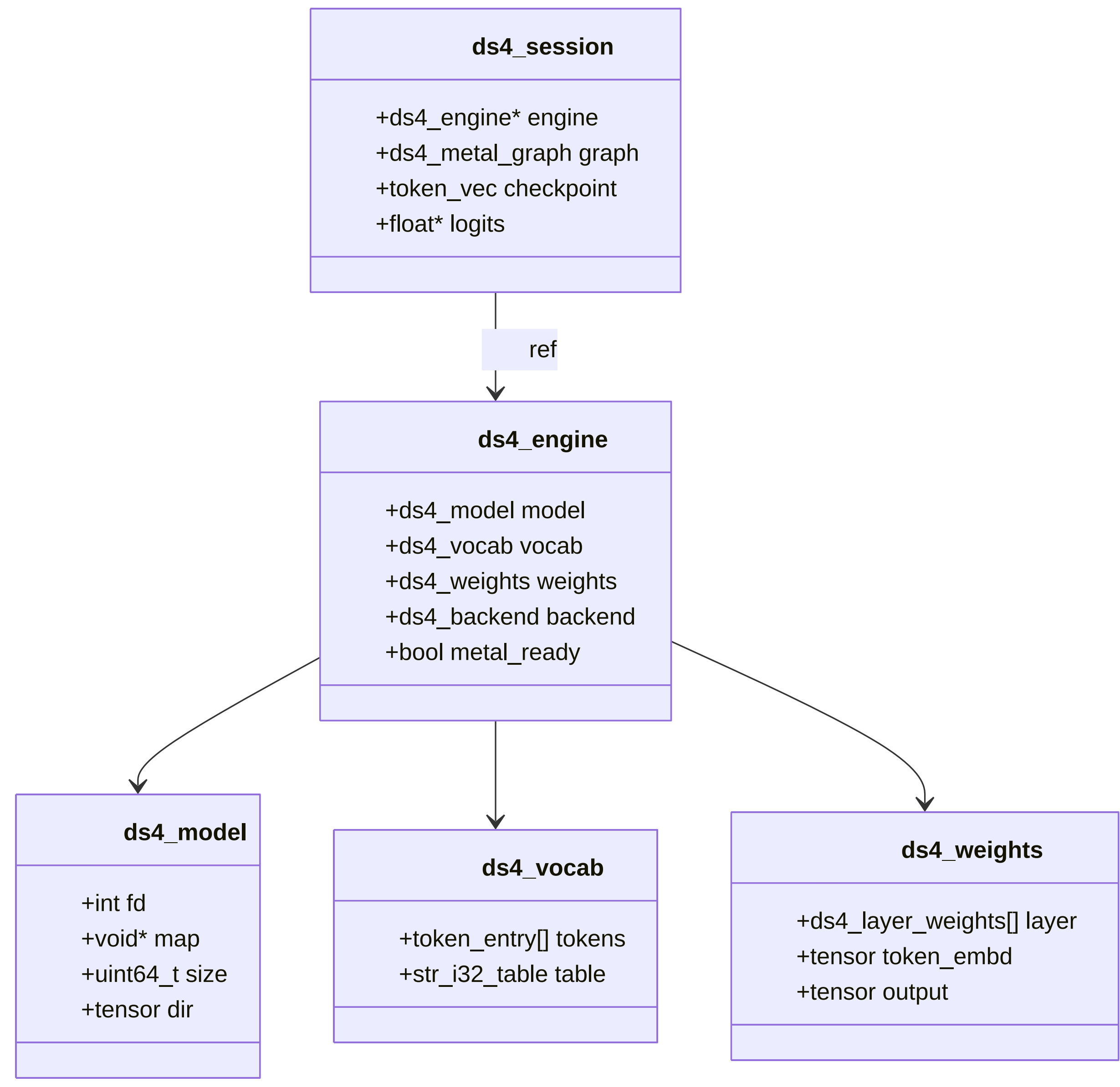
```
// ds4.h（公共头文件）
typedef struct ds4_engine ds4_engine;   // 前向声明
typedef struct ds4_session ds4_session;
```

```
// ds4.c（内部定义）
struct ds4_engine {
    ds4_model model;
    ds4_model mtp_model;
    ds4_vocab vocab;
    ds4_weights weights;
    ds4_mtp_weights mtp_weights;
    ds4_backend backend;        // DS4_BACKEND_METAL / CUDA / CPU
    int mtp_draft_tokens;
    float mtp_margin;
    bool quality;
    bool metal_ready;           // GPU 后端已初始化
    bool mtp_ready;
};
```

调用者只能通过指针操作，不能访问字段。修改 `ds4_engine` 内部结构不需要重新编译 `ds4_server.c`。

2. 结构体嵌套设计

ds4 的类型层次：



3. Session Rewrite API — 检查点恢复

工具调用后，模型生成的 DSML 文本可能与"规范"提示不完全一致（格式等价但字节不同）。此时 session 的检查点需要"重写"来匹配下一轮的规范提示。

```c
// 重写结果枚举
typedef enum {
    DS4_SESSION_REWRITE_ERROR = -1,         // 错误
    DS4_SESSION_REWRITE_OK = 0,             // 成功
    DS4_SESSION_REWRITE_REBUILD_NEEDED = 1, // 需要从头重建
} ds4_session_rewrite_result;

// 判断是否需要完整重建（Metal 图中有不可就地替换的 KV 行）
bool ds4_session_rewrite_requires_rebuild(int live_len, int canonical_len, int c
ommon);

// 从公共前缀开始重写 session
ds4_session_rewrite_result ds4_session_rewrite_from_common(
        ds4_session *s, const ds4_tokens *prompt, int common,
        char *err, size_t errlen);
```

关键洞察：session 检查点不仅是 token 向量，Metal 图中还有 raw SWA 行、压缩 KV 行、indexer 行等。替换 live tail 后面的内容需要恢复整个图前沿，因此不能简单就地修改。

渲染文本前缀匹配

KV cache 查找从基于 token ID 的 SHA1 改为渲染文本字节的 **SHA1**：

- 模型生成一个 token 的解码文本，客户端可能在下一轮把它作为两个 prompt token 发回（BPE 边界差异）
- 渲染字节前缀匹配可以处理这种情况：命中后用 checkpoint 的精确 token 作为前缀，只对新增后缀重新分词
- cache 文件名 `<sha1>.kv` 直接是渲染文本的 SHA1

对齐前沿检查点

持续保存使用绝对对齐位置（如每 ~10k token），不相对于上一次保存点。这防止早期 cold checkpoint 偏移整个保存计划，确保长生成过程中有均匀分布的重启点。

Cold checkpoint 的裁剪锚定在 chat task boundary（最后一个 `<User>` token 之后、第一个 `<Assistant>` token 之前），保留稳定的用户角色脚手架，最大化跨会话复用。无 chat anchor 时回退到 trim + 对齐启发式。

4. Session 快照 API

增量基准测试（`ds4-bench`）需要在不同 context frontier 处保存和恢复 session 状态：

```c
// 进度回调
typedef void (*ds4_session_progress_fn)(void *ud, int progress);
void ds4_session_set_progress(ds4_session *s, ds4_session_progress_fn fn, void *
ud);

// 快照保存/加载/释放
void *ds4_session_save_snapshot(ds4_session *s);
int ds4_session_load_snapshot(ds4_session *s, void *snapshot);
void ds4_session_snapshot_free(void *snapshot);
```

快照保存 session 在某个 token 位置的完整状态（KV cache 行、logits 等），恢复后可以继续推理。`ds4-bench` 利用这个 API 在每个 frontier 处保存快照、测 decode 速度、再恢复快照继续 prefill，无需从头重新加载。

5. 独立分词调试

```c
// 不加载推理引擎，只打开模型文件的分词器部分
int ds4_dump_text_tokenization(const char *model_path, const char *text, FILE *f
p);
```

这个函数调用 `model_open(..., false, false)` —— `prefetch_cpu=false` 意味着不预取 CPU 映射，因为只需要分词器数据，不需要遍历 81GB 的张量数据。

6. 函数指针回调

```c
typedef void (*ds4_token_emit_fn)(void *ud, int token);
typedef void (*ds4_generation_done_fn)(void *ud);
```

生成循环中，每产生一个 token 就调用 `emit`，结束时调用 `done`。`void *ud` 是用户数据指针，用于在回调中传递上下文。这是 C 语言实现"观察者模式"的常见方式。

7. required_tensor — 字符串匹配绑定

```c
static void weights_bind(ds4_weights *w, const ds4_model *m) {
    w->token_embd = required_tensor(m, "token_embd.weight");
    w->output     = required_tensor(m, "output.weight");

    for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
        ds4_layer_weights *l = &w->layer[il];
        l->attn_norm = required_tensorf(m, "blk.%u.attn_norm.weight", il);
        l->attn_q_a  = required_tensorf(m, "blk.%u.attn_q_a.weight", il);
        // ... 30+ 个权重张量
    }
}
```

`required_tensor` 在张量目录中按名称查找。如果找不到就报错退出——再次体现"早失败"。`required_tensorf` 是格式化版本，用 `sprintf` 构造名称如 `"blk.0.attn_norm.weight"`。

sscanf %n 锚点防止部分匹配

`deepseek4-quantize.c` 中的 `parse_expert_tensor` 用 `sscanf` 匹配 MoE 专家张量名，但旧写法会误匹配：

```c
// 旧：会匹配 blk.3.ffn_gate_exps.weight_bias 等不相关张量
sscanf(name, "blk.%d.ffn_%15[^_]_exps.weight", &layer, type)

// 新：用 %n 捕获解析位置，验证消耗了整个字符串
int rest = 0;
sscanf(name, "blk.%d.ffn_%15[^_]_exps.weight%n", &layer, type, &rest);
if (rest != strlen(name)) return false;  // 不是精确匹配
```

`%n` 是 `sscanf` 的特殊格式符，写入已消耗的字符数而不消耗输入。`rest == strlen(name)` 保证没有多余后缀。

LLM 知识点

1. Transformer 权重组成

一个 Transformer 层（`ds4_layer_weights`）的权重可以分组：

注意力子系统：

权重	形状	作用
`attn_norm`	[4096]	RMSNorm 缩放
`attn_q_a`	[4096, 1024]	Query LoRA 压缩
`attn_q_a_norm`	[1024]	压缩后归一化
`attn_q_b`	[1024, 64×512]	Query LoRA 解压
`attn_kv`	[4096, 512]	Key-Value 投影
`attn_sinks`	[64]	注意力汇聚点
`attn_output_a/b`	[64×512, 1024], [1024, 4096]	输出投影（LoRA）

MoE FFN 子系统：

权重	形状	作用
`ffn_norm`	[4096]	FFN RMSNorm
`ffn_gate_inp`	[4096, 256]	专家路由器
`ffn_gate/up/down_exps`	[256, 2048, 4096]	256 个路由专家
`ffn_gate/up/down_shexp`	[2048, 4096]	1 个共享专家

2. 超连接（**Hyper-Connection**）

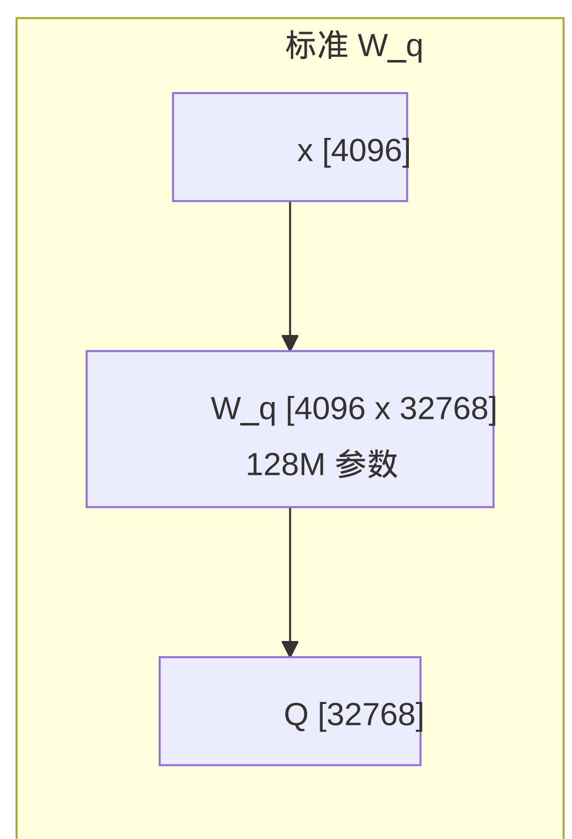
ds4.c 的一个独特设计——用可学习的门控代替标准残差连接：

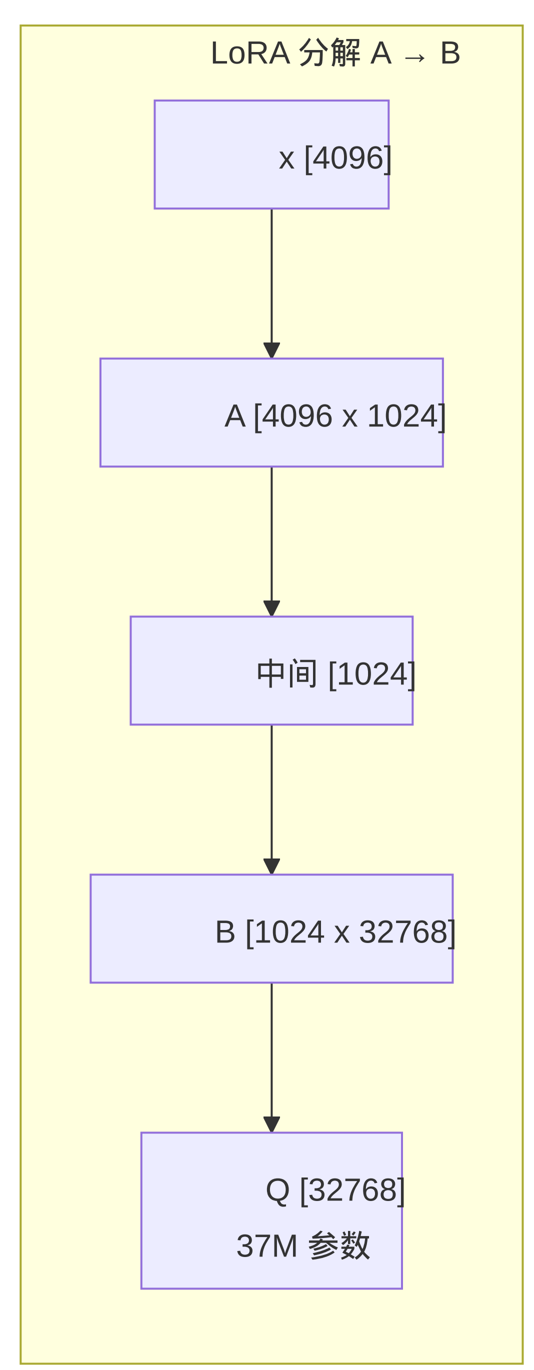
```
// 标准残差：out = x + sublayer(x)
// 超连接：out = HC_post(x, sublayer(HC_pre(x)))

// HC_pre: 计算 HC 维度 = DS4_N_EMBD × DS4_N_HC (4096 × 4 = 16384)
hc_from_plain_embedding(cur, plain, DS4_N_EMBD, DS4_N_HC);

// HC_post: 学习每个流的权重
// fn × scale × sigmoid(x) + base → 门控权重
```

DS4_N_HC = 4 意味着每个位置有 4 个并行流（streams），通过可学习门控混合。

3. LoRA 低秩分解

Query 和 Output 投影使用 LoRA（Low-Rank Adaptation）分解：





秩 r = 1024（`DS4_N_LORA_Q`），大幅减少参数量。虽然 LoRA 通常用于微调，DeepSeek V4 把它作为架构的一部分来压缩 KV cache。

4. 重要性矩阵（imatrix）校准

量化时不同权重对模型输出影响不同。imatrix（importance matrix）通过在校准数据上运行推理，记录每个权重的贡献度，指导量化器为重要权重保留更高精度：

```
校准数据（prompts）→ 前向推理 → 记录每层激活 → 统计重要性 → 量化时参考
```

ds4 的校准数据集位于 `gguf-tools/imatrix/dataset/`，已扩展覆盖多语言编程、工具调用、算法回忆、Bash 脚本、多语言翻译、语言/散文任务、代码合成、agent transcript 回放、trace 诊断、长上下文检索与比较等场景，并包含 ds4-eval 评测题（不含答案键）作为校准输入，确保量化后的模型在各种任务上保持质量。

---

2. 量化与矩阵运算

> 284B 参数的 FP16 模型需要 568GB 内存，消费级硬件根本装不下。量化把每个参数从 16 bit 压缩到 2-4 bit，让 81GB 的模型在 128GB MacBook 上可运行。今天学习位操作提取压缩值、block 布局和矩阵-向量乘法。

C 知识点

1. 位操作

量化核心是把 32 位浮点数压缩为 2-8 位整数。这需要大量位操作：

提取 2-bit 值：

```c
// 从 1 字节中提取 4 个 2-bit 值
uint8_t byte = qs[i];              // 8 位 = 4 个 2-bit 值
uint8_t v0 = byte & 0x03;          // 最低 2 位
uint8_t v1 = (byte >> 2) & 0x03;  // 次低 2 位
uint8_t v2 = (byte >> 4) & 0x03;  // 次高 2 位
uint8_t v3 = (byte >> 6) & 0x03;  // 最高 2 位
```

Q8_K 量化（8-bit）：

```c
// 把 float 数组量化为 int8
static void ds4_quantize_row_q8_K(const float *x, block_q8_K *y, int64_t n) {
    for (int j = 0; j < n; j += QK_K) {
        float amax = 0.0f;
        for (int i = 0; i < QK_K; i++) {
            float ax = fabsf(x[j + i]);
            if (ax > amax) amax = ax;
        }
        float scale = amax / 127.0f;
        y->d = scale;
        for (int i = 0; i < QK_K; i++)
            y->qs[i] = (int8_t)roundf(x[j + i] / scale);
    }
}
```

2. 查表法（**Lookup Table**）

IQ2_XXS 量化使用预计算表来解码 2-bit 编码：

```c
// 预计算的 IQ2_XXS 解码表（来自 llama.cpp/GGML）
static const uint64_t iq2xxs_grid[256] = { ... };

// 从 2-bit 索引查表得到 8 个 int8 值
const uint64_t grid = iq2xxs_grid[qs_byte];
```

查表法是经典的性能优化：用内存（表）换取计算（乘法/移位），特别适合量化解码这种简单但高频的操作。

3. Block 布局

量化数据以 block 为单位存储，每个 block 包含 256 个权重值：

```
Q2_K block (84 字节):
┌──────────────────────────────────────
──────┐
│ scales[16]    │ 16 组缩放因子 (1
B)  │  16 字节
│ qs[64]        │ 256 个 2-bit 值
│  64 字节
│ d (uint16)    │ 全局缩放
│   2 字节
│ dmin (uint16) │ 全局最小值偏移
│   2 字节
└──────────────────────────────────────
──────┘
  每权重: 84 × 8 / 256 = 2.625 bit
```

```
Q4_K block (144 字节):
┌──────────────────────────────────────
──────┐
│ d (uint16)    │ 全局缩放
│   2 字节
│ dmin (uint16) │ 全局最小值偏移
│   2 字节
│ scales[12]    │ 12 组缩放+偏移
│  12 字节
│ qs[128]       │ 256 个 4-bit 值
│ 128 字节
└──────────────────────────────────────
──────┘
  每权重: 144 × 8 / 256 = 4.5 bit
```

4. 定点点积

量化推理的核心操作是"量化向量的点积"：

```
// Q8_K × Q2_K 点积（简化版）
float dot_q2_k_q8_k(const block_q2_K *a, const block_q8_K *b) {
    float sum = 0.0f;
    // 解量化 a 的值: value = d * (scale_i * qs_value + dmin * offset_i)
    // 乘以 b 的值: value_b = b->d * b->qs[i]
    // 累加
    return sum;
}
```

> 实际实现考虑了分组缩放、偏移、以及 SIMD 优化（ARM NEON）。

LLM 知识点

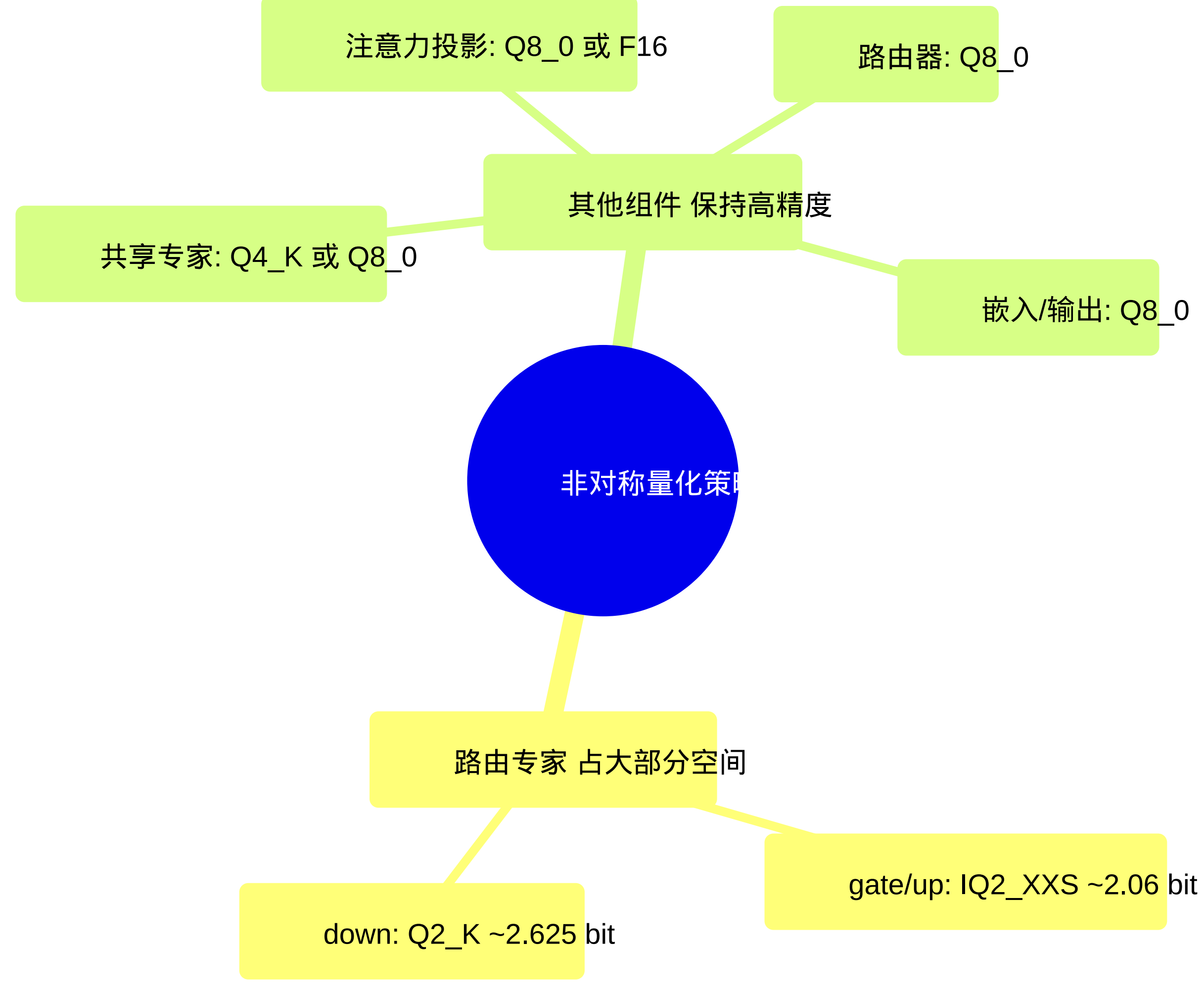
1. 模型量化原理

训练时使用 FP32（32-bit 浮点），推理时可以用更少的位数：

```
FP32 → 每个权重 32 bit → 284B 模型 ≈ 1.1 TB
Q8   → 每个权重  8 bit → 284B 模型 ≈ 284 GB
Q4   → 每个权重  4 bit → 284B 模型 ≈ 142 GB
Q2   → 每个权重 ~2 bit → 284B 模型 ≈  81 GB
```

2. 非对称量化

DeepSeek V4 Flash 使用"非对称量化"——不同组件用不同精度：



为什么可以这样？路由专家占模型空间的绝大部分，但每个 token 只使用 6/256。即使单个专家的精度较低，6 个专家的混合效果仍然很好。

3. 量化对推理的影响

量化的好处：

- 内存减少：81GB（Q2）vs 284GB（FP32）
- 带宽减少：读写更快，推理速度提升
- 可运行：128GB MacBook 可以运行 Q2 模型

量化的代价：

- 精度损失：模型输出质量略有下降
- 计算开销：需要量化解码（查表、缩放）
- 但实际影响很小：2-bit 量化的 DeepSeek V4 Flash 仍能通过编码代理测试

4. 矩阵-向量乘法（matvec）

推理中最频繁的操作：一个向量乘以一个权重矩阵。

```
y = x @ W

输入向量 x: [4096]      （1D）
权重矩阵 W: [4096, 2048] （2D，可能被量化）
输出向量 y: [2048]      （1D）
```

在 ds4.c 中有多个变体：

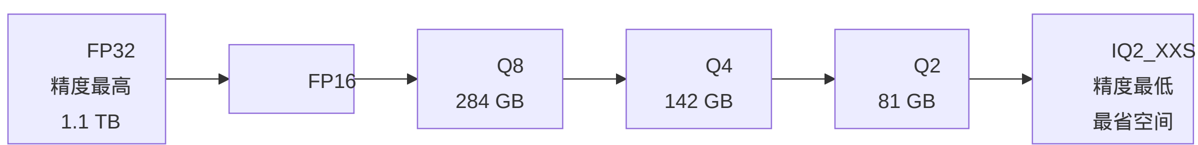
- `matvec_f16`: FP16 权重
- `matvec_q8_0`: Q8_0 权重
- `matvec_q2_k`: Q2_K 权重
- `matvec_iq2_xxs`: IQ2_XXS 权重（用于路由专家）
- `matvec_q8_0_pair`: 同时计算 gate 和 up（共享输入）

5. GGUF 量化工具链与 imatrix

`gguf-tools/` 目录提供独立的 HF-to-GGUF 量化器 `deepseek4-quantize.c`（1885 行），不依赖 GGML/llama.cpp：

- 支持 imatrix 感知的路由专家量化，内置 Q8_0/Q4_K/Q2_K/IQ2_XXS 量化器
- **imatrix**（重要性矩阵）：通过校准数据集收集各张量的 token 频率，量化时对高频张量分配更高精度
- 新增 `quality-testing/` 质量测试框架，含评分脚本（`score_official`）和校准 prompt 语料库
- **q2-imatrix** 版本：用 imatrix 调优的 q2 权重，logits 误差接近 q4——以极低 bit 宽获得接近高精度量化的质量

总结：量化与精度的权衡



ds4.c 的选择: 路由专家用 IQ2_XXS/Q2_K，其他用 Q4/Q8 → 在 128GB 内存上运行 284B 参数模型。

---

3. 注意力机制

> 注意力是 Transformer 的核心——它让模型决定"看哪里"。今天理解 Self-Attention 的分数计算、KV Cache 避免重复计算、MLA 压缩让百万 token 上下文成为可能。这是整个学习中最关键的一天。

设计决策推导：从问题到方案

今天的每个知识点都不是孤立的——它们是一条解决问题的链条：

```
问题 1：如何让模型"关注"不同位置的信息？
  └─ 方案：Self-Attention（Q·K^T 点积算分数，softmax 加权）

问题 2：逐 token 生成时，之前的 K/V 已经算过了，为什么要重复算？
  └─ 方案：KV Cache（缓存历史的 K/V，每步只算新 token 的）

问题 3：KV Cache 随序列线性增长，长上下文时内存爆了怎么办？
  └─ 方案 1：滑动窗口（只保留最近 128 个 token 的精确 KV）
  └─ 方案 2：MLA 压缩（历史 token 用低维压缩表示，64 倍压缩）

问题 4：滑动窗口丢弃了旧 token，模型怎么"记得"很久以前的内容？
  └─ 方案：混合注意力（raw 最近 128 + comp 压缩历史，统一 softmax）

问题 5：标准注意力是 O(n²)，压缩后行数仍然很多怎么办？
  └─ 方案：Indexer（模型学习哪些压缩行值得看，跳过无关行）

问题 6：64 个注意力头的 KV Cache 是 1 个头的 64 倍，太大了？
  └─ 方案：MQA（64 个 Q 头共享 1 组 K/V，KV 体积减少 64 倍）
```

每个方案解决了一个具体问题，同时又引入了新问题，驱动下一个设计决策。理解这条链条比记住每个概念的定义更重要。

C 知识点

1. 多维数组索引

注意力涉及多维数据：`[heads, seq_len, head_dim]`。在 C 中用一维数组 + 手动索引：

```c
// 第 h 个注意力头，第 r 个 KV 行，第 d 个维度
float val = kv_rows[(uint64_t)r * DS4_N_HEAD_DIM + d];

// 第 h 个头的 query 向量
const float *qh = q + (uint64_t)h * DS4_N_HEAD_DIM;

// 第 h 个头的输出
float *oh = out_heads + (uint64_t)h * DS4_N_HEAD_DIM;
```

公式：`array[i][j][k]` = `base[i * J * K + j * K + k]`

2. 指针算术

```c
const float *kv = raw_kv + (uint64_t)r * DS4_N_HEAD_DIM;  // 指向第 r 行
float *head_out = out + (uint64_t)h * DS4_N_HEAD_DIM;       // 指向第 h 个头的输出
```

`raw_kv` 是 `float *`，`+ n` 实际偏移 `n * sizeof(float)` 字节。这是 C 指针算术的核心。

3. 向量操作（BLAS 风格）

ds4.c 实现了基本的 BLAS-1 操作：

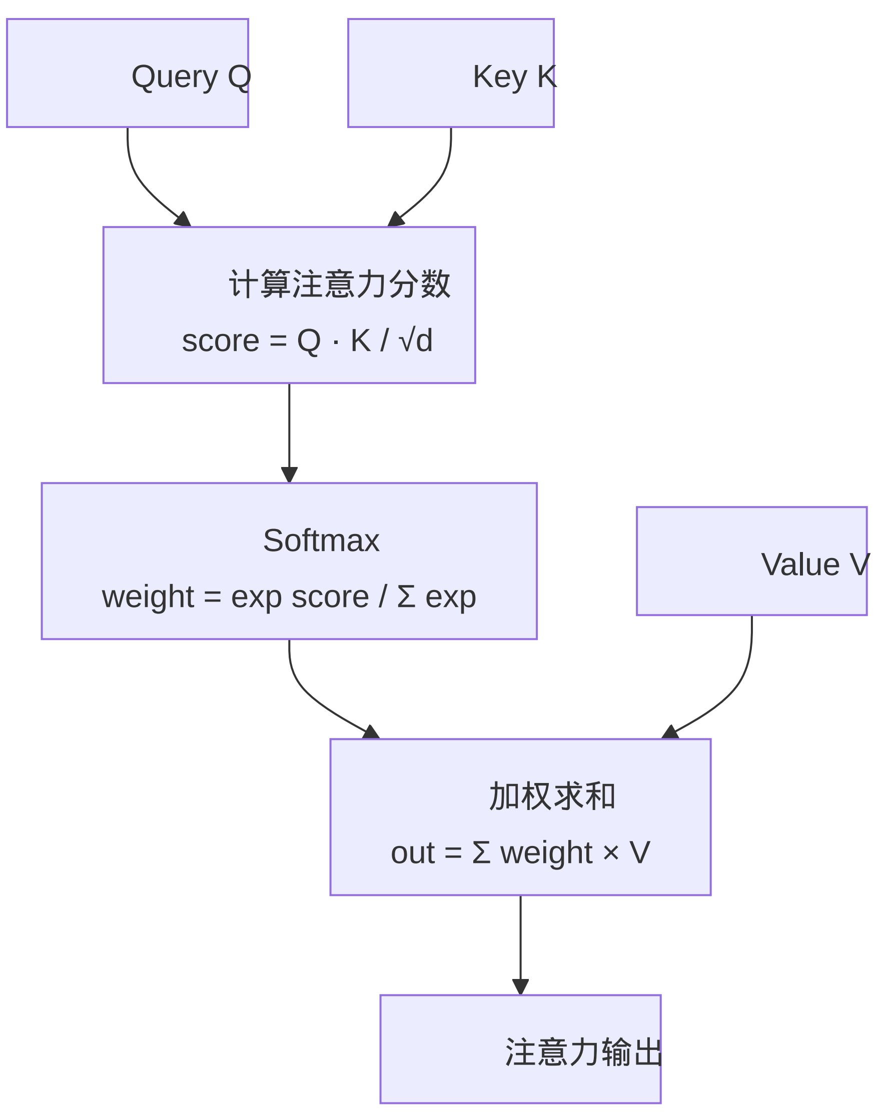
```c
// 点积: sum(a[i] * b[i])
static float dot_f32(const float *a, const float *b, uint64_t n);

// 缩放: a[i] *= s
static void scale_f32(float *a, float s, uint64_t n);

// AXPY: y[i] += alpha * x[i]
static void axpy_f32(float *y, const float *x, float alpha, uint64_t n);
```

这些是线性代数的基本操作（DOT, SCAL, AXPY 来自 BLAS 标准）。

LLM 知识点

1. Self-Attention 机制

注意力是 Transformer 的核心：让每个位置"关注"所有相关位置。采样阶段使用 softmax（参见 采样 softmax），KV Cache 通过 mmap & KV Cache 的内存映射加载。



ds4.c 实现 `layer_attention_rows_one()`（约 4746-4786 行）：

```
for (uint32_t h = 0; h < DS4_N_HEAD; h++) {         // 64 个头
    const float *qh = q + h * DS4_N_HEAD_DIM;         // 当前头的 query

    float max_score = sinks[h];                        // sink 作为初始最大值
    for (uint32_t r = 0; r < n_kv; r++) {              // 遍历所有 KV 行
        score[r] = dot_f32(qh, kv + r * HEAD_DIM, HEAD_DIM) * kq_scale;
        if (score[r] > max_score) max_score = score[r];
    }

    float denom = expf(sinks[h] - max_score);          // softmax 分母
    for (uint32_t r = 0; r < n_kv; r++) {
        float weight = expf(score[r] - max_score);     // softmax 权重
        axpy_f32(oh, kv + r * HEAD_DIM, weight, HEAD_DIM); // 加权累加
        denom += weight;
    }
    scale_f32(oh, 1.0f / denom, HEAD_DIM);             // 归一化
}
```

2. KV Cache

KV Cache 存储历史 token 的 Key 和 Value，避免重复计算。

为什么需要 **KV Cache**？

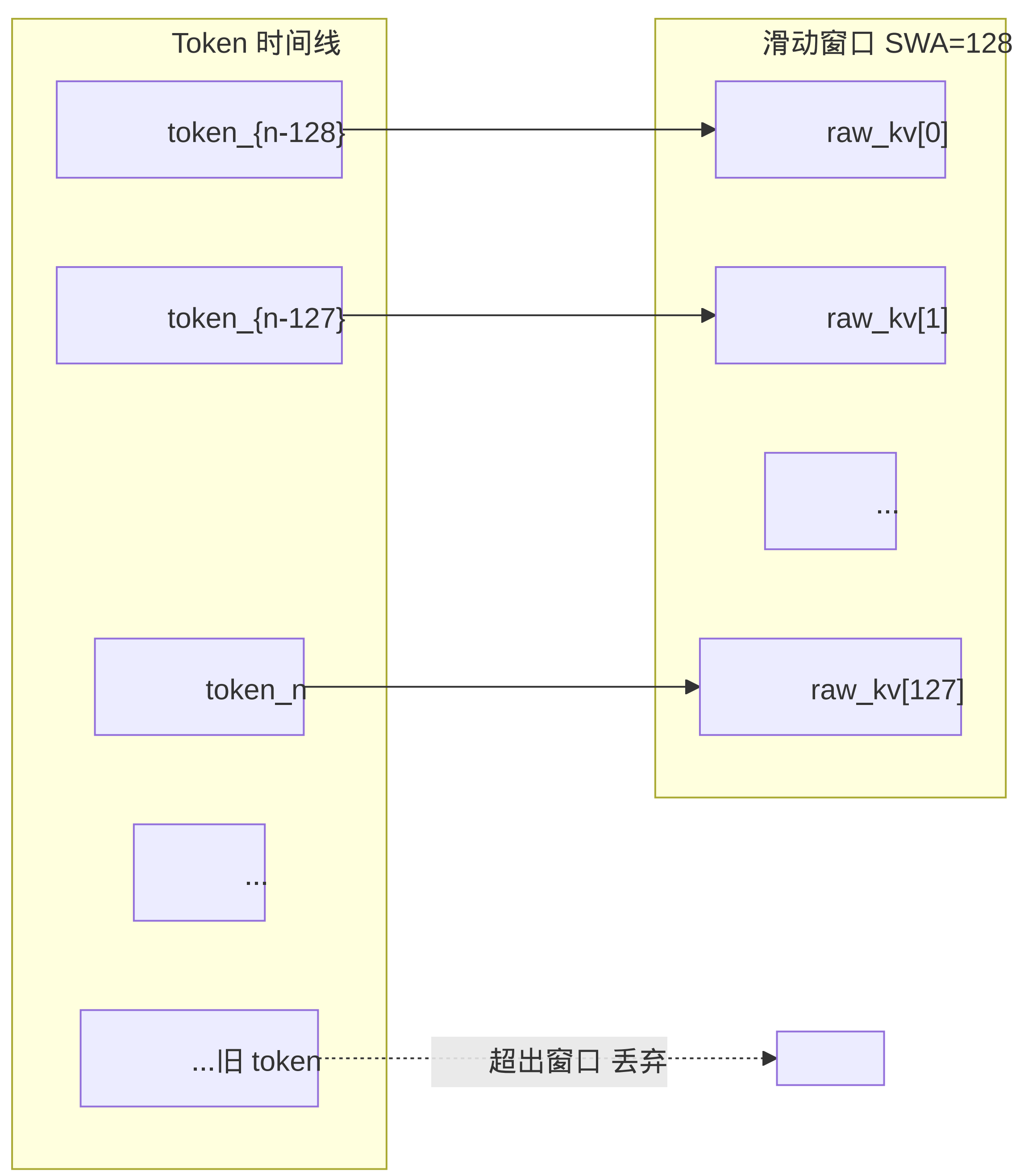
```
没有 KV Cache:
  生成 token 100 时，重新计算 token 0-99 的 K 和 V
  → O(n²) 计算
```

```
有 KV Cache:
  生成 token 100 时，只计算 token 100 的 K 和 V
  → O(n) 计算
  用缓存的 K[0-99] 和 V[0-99] 做注意力
```

ds4.c 的 **KV Cache** 结构：

```c
typedef struct {
    float *raw_kv;          // 原始滑动窗口 KV 行
    uint32_t n_raw;         // 当前行数
    uint32_t cap_raw;       // 容量（滑动窗口大小）

    float *attn_comp_kv;  // 压缩后的 KV 行
    uint32_t n_comp;        // 压缩行数

    float *attn_state_kv;   // 压缩器状态
    float *attn_state_score; // 压缩器注意力分数
} ds4_layer_cache;
```

滑动窗口（**Sliding Window Attention**）：



KV Cache 不保留所有历史，而是只保留最近 N 个 token（SWA = 128）：

```c
static void kv_cache_push_raw(ds4_layer_cache *cache, const float *kv) {
    if (cache->n_raw < cache->cap_raw) {
        // 还有空间，直接追加
        memcpy(dst, kv, ...);
        cache->n_raw++;
    } else {
        // 满了，滑动窗口：移除最旧的，添加最新的
        memmove(raw_kv, raw_kv + HEAD_DIM, (cap-1) * HEAD_DIM);
        memcpy(raw_kv + (cap-1) * HEAD_DIM, kv, ...);
    }
}
```

3. 注意力汇聚（**Attention Sinks**）

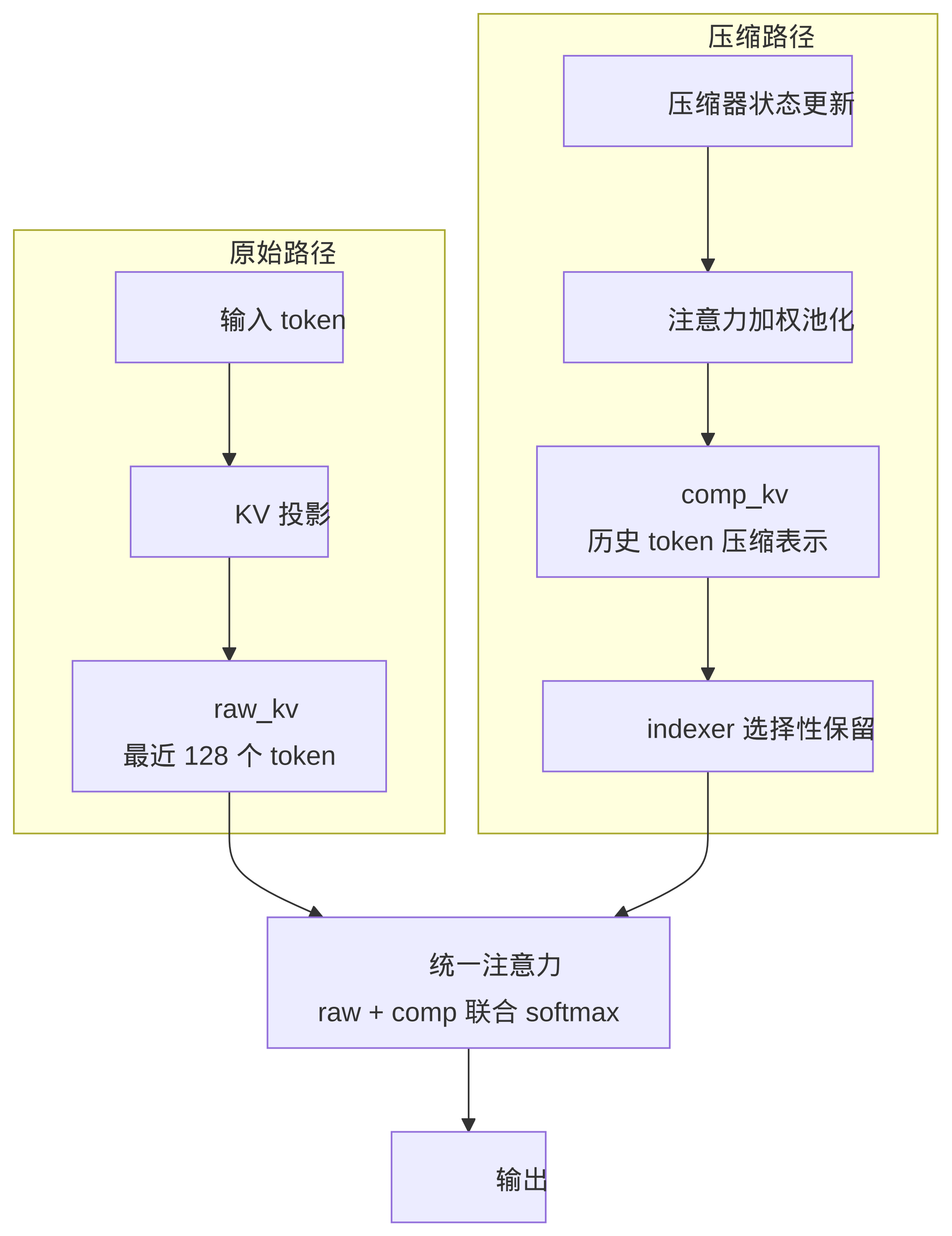
ds4.c 的独特设计：每个头有一个可学习的"sink"值：

```
const float *sinks = tensor_data(model, layer->attn_sinks);
float max_score = sinks[h];  // sink 作为 softmax 的基准分数
float denom = expf(sinks[h] - max_score);  // sink 参与 softmax 分母
```

Sink 不贡献 value 向量，但参与 softmax 归一化。它允许模型"丢弃"不重要的位置，而不是被迫分配注意力。

4. MLA（**Multi-head Latent Attention**）

DeepSeek V4 Flash 使用 MLA 压缩 KV cache：




```
标准注意力:  KV cache 大小 = n_layer × seq_len × n_head × head_dim
            = 43 × seq_len × 64 × 512 = 1,413,376 × seq_len

MLA 压缩后:  KV cache 大小 = n_layer × seq_len × 1 × head_dim
            = 43 × seq_len × 1 × 512 = 22,016 × seq_len
```

GQA 头分组比：64×（64 个 Q 头共享 1 个 KV 头）。 MLA 实际内存压缩比：128×（原始 KV 总维度 65536 ÷ 潜在表示 512）。详见 mla。

压缩器工作流程：

```
每 ratio 个 token:
1. 输入 token → KV 投影 → 压缩器状态更新
2. 压缩器状态 → 注意力加权池化 → 压缩 KV 行
3. 压缩 KV 行存入 attn_comp_kv

注意力时:
→ 同时关注 raw_kv（最近 128 个）和 attn_comp_kv（历史压缩行）
→ 由 indexer 选择哪些压缩行可以关注
```

5. 混合注意力（Mixed Attention）

ds4.c 的 `layer_attention_mixed_one` 同时使用原始和压缩 KV：

```
// 原始行：最近 128 个 token
for (r = 0; r < n_raw; r++)
    score[idx] = dot(q, raw_kv[r]);

// 压缩行：历史 token 的压缩表示
for (r = 0; r < n_comp; r++) {
    if (!comp_allowed[r]) continue;   // indexer 决定是否允许
    score[idx] = dot(q, comp_kv[r]);
}

// 统一 softmax + 加权求和
```

这就是为什么 DeepSeek V4 Flash 可以支持百万 token 上下文：大部分历史被压缩，但关键信息通过 indexer 选择性地保留。

5b. 逐层压缩缓存容量

Metal GPU graph 使用 `comp_cap` 控制每层压缩 KV 行的最大数量。之前的实现用全局 `comp_cap`（基于最小压缩比计算），所有层共用同一个悲观容量。对于 ratio-128 的层（128 个 token 才产生一行压缩行），实际只需要 ratio-4 层 1/32 的容量：

```
// 旧：全局容量，所有层用 ratio-4 的悲观值
g->comp_cap = ctx_size / min_ratio + 2;  // min_ratio 通常为 4

// 新：逐层容量，按每层自己的压缩比计算
for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
    const uint32_t ratio = ds4_layer_compress_ratio(il);
    g->layer_comp_cap[il] = ctx_size / ratio + 2;
}
```

效果：ratio-128 层的持久缓存（`layer_attn_comp_cache`、`layer_index_comp_cache`）分配量大幅减少，节省 GPU 显存。工作缓冲区仍使用全局 `comp_cap`，因为 ratio-4 的 indexer 层可能达到该上限。

---

4. MoE 混合专家

> 284B 参数全部计算太慢了。MoE 的思路是"参数可以多，但每次只用一小部分"——256 个专家里只激活 6 个，实际计算量不到总参数的 0.1%。今天学习路由策略、SwiGLU 激活和非对称精度设计。

MoE 在注意力机制（同一主题前一章节）的基础上，用稀疏路由替换全连接 FFN。

设计决策推导：为什么用 **MoE**？

```
问题 1：284B 参数的模型，每次推理要读取所有权重——decode 瓶颈是内存带宽
  └ 方案：把大 FFN 拆成 256 个小专家，每次只用 6 个
      → 读取量从 256× 降到 6×，但参数总量不变

问题 2：如何决定每个 token 用哪些专家？
  └ 方案 A（前 3 层）：Hash 路由——确定性分配，不同 token 分散到不同专家
  └ 方案 B（后续层）：Top-K 路由——学习到的 Router 打分，选分数最高的 6 个

问题 3：为什么是 6 个而不是 3 个或 12 个？
  └ 实验验证的甜点：1→4 效果提升明显，4→6 仍有收益，6 之后边际递减
      → 更多专家 = 更多内存读取 = 更慢，收益不足以抵消成本

问题 4：6 个专家够吗？知识覆盖会不会不够？
  └ 方案：加 1 个共享专家（所有 token 必经）兜底通用知识
      → 43 层 × 6 专家 = 258 次选择，通过残差连接逐步积累

问题 5：256 个路由专家的参数量太大（~6.4B），怎么压缩模型文件？
  └ 方案：非对称量化——路由专家用 2-bit（不常被激活，误差被稀释），
     共享专家和注意力投影用 4-8 bit（每次都用，需要精度）
      → 284B 参数压缩到 81GB，在 128GB MacBook 上可运行
```

C 知识点

1. 指针数组

MoE 中每个专家有自己的权重矩阵，通过指针数组索引：

```
// 256 个专家，每个有独立的 gate/up/down 权重
ds4_tensor *ffn_gate_exps;   // 指向 256 个专家的 gate 权重
ds4_tensor *ffn_up_exps;     // 指向 256 个专家的 up 权重
ds4_tensor *ffn_down_exps;   // 指向 256 个专家的 down 权重
```

`selected[6]` 数组存储被选中的 6 个专家索引，计算时只加载这 6 个专家的权重。

2. 稀疏索引

top-k 选择只关心前 6 个最大的值，不关心其余 250 个：

```
static void topk_desc(const float *score, int n, int k, int *idx) {
    for (int i = 0; i < k; i++) idx[i] = -1;        // 初始化为无效
    for (int i = 0; i < n; i++) {                     // 扫描所有 n=256 个
        for (int j = 0; j < k; j++) {                 // 与当前 top-k 比较
            if (idx[j] < 0 || score[i] > score[idx[j]]) {
                for (int m = k - 1; m > j; m--)         // 后移
                    idx[m] = idx[m - 1];
                idx[j] = i;                             // 插入
                break;
            }
        }
    }
}
```

这是"部分排序"——只需要前 k 个，不需要完全排序所有 256 个。

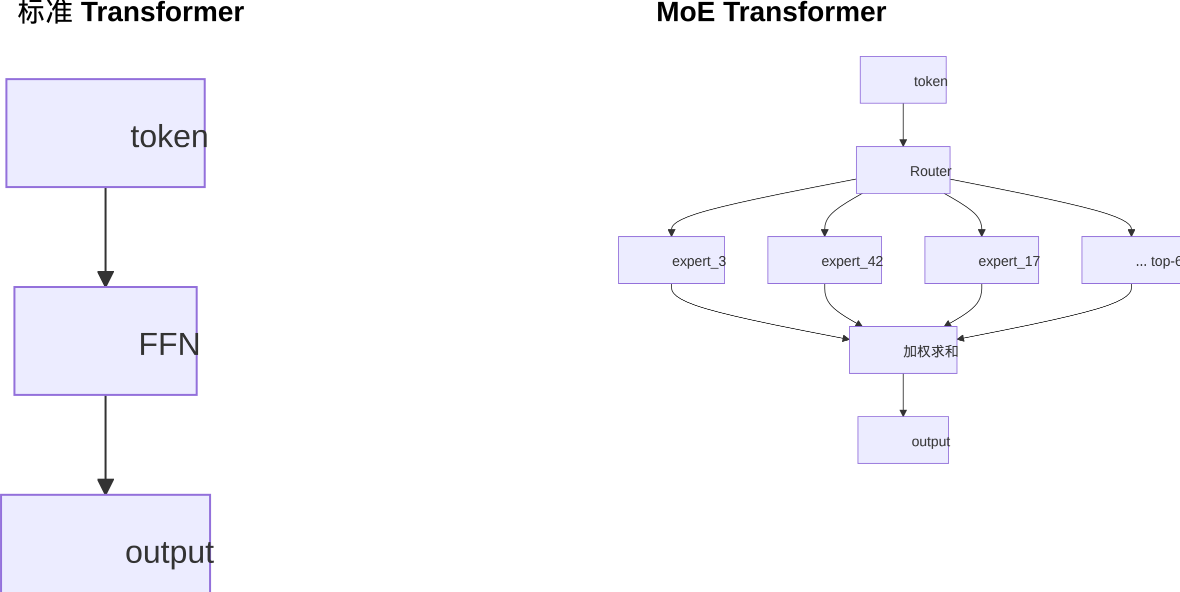
3. 数学辅助函数

```c
// SiLU 激活: x × sigmoid(x)
static float silu(float x) {
    return x * sigmoid_stable(x);
}

// 数值稳定的 softplus: log(1 + exp(x))
static float softplus_stable(float x) {
    if (x > 20.0f) return x;            // 避免 exp 溢出
    if (x < -20.0f) return expf(x);     // log(1+小量) ≈ 小量
    return log1pf(expf(x));             // log1p 比 log(1+x) 更精确
}
```

LLM 知识点

1. MoE（Mixture of Experts）

传统 Transformer 每个 token 经过同一个 FFN。MoE 把 FFN 替换为多个"专家"：



DeepSeek V4 Flash 的 MoE 配置：

- 256 个路由专家（`DS4_N_EXPERT = 256`）
- 每个 token 只激活 6 个（`DS4_N_EXPERT_USED = 6`）
- 1 个共享专家（`DS4_N_EXPERT_SHARED = 1`）

DeepSeek V4 **PRO** 是更大的模型变体（`DS4_VARIANT_PRO = 1`），MoE 规模远大于 Flash：

- **384** 个路由专家（`DS4_SHAPE_PRO`），每个 token 激活 6 个
- 61 层（Flash 为 43 层），7168 嵌入维度
- 路由专家全部使用 **Q4_K** 量化（gate、up、down 张量均为 Q4_K）
- 总参数 ~1.6T，模型文件 ~430GB（IQ2_XXS）或 ~838GB（Q4 分片）
- 单机推理需要 512GB 内存的 Mac Studio，或通过分布式推理跨两台机器运行

2. 路由（Gating）

ds4.c 有两种路由策略：

哈希路由（前 **3** 层，**`DS4_N_HASH_LAYER = 3`**）：

```
// 直接通过 token ID 查表得到专家
const int32_t *row = table + (uint64_t)token * DS4_N_EXPERT_USED;
for (int i = 0; i < DS4_N_EXPERT_USED; i++)
    selected[i] = row[i];
```

固定映射：同一个 token ID 永远路由到同一组专家。简单、无计算开销。

Top-K 路由（后续层）：

```
// 1. 计算路由分数
float logits[256];
matvec_any(logits, model, layer->ffn_gate_inp, x);  // 输入 → 256 维

// 2. 特殊激活: √(softplus(logit))
for (int i = 0; i < 256; i++)
    probs[i] = sqrtf(softplus_stable(logits[i]));

// 3. 选择 top-6
topk_desc(probs, 256, 6, selected);

// 4. 归一化权重（加偏置选，用无偏倚的权重）
// 选择时可以加 bias（ffn_exp_probs_b）
// 但权重从无偏倚的 probs 计算
float sum = Σ probs[selected[i]];
for (int i = 0; i < 6; i++)
    expert_weight[i] = probs[selected[i]] / sum * DS4_EXPERT_WEIGHT_SCALE;
```

为什么用 `√(softplus(x))` 而不是标准 softmax？

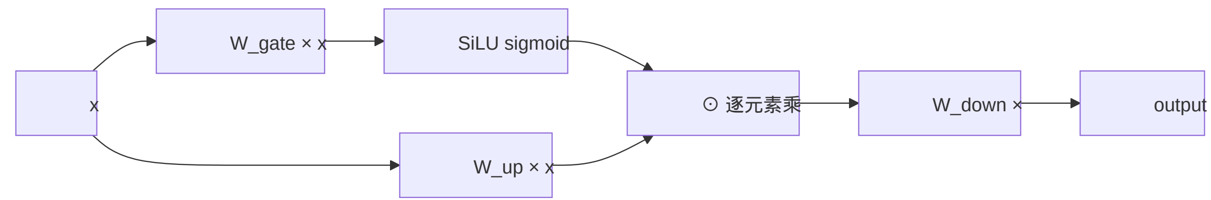
- softplus 保证非负
- 平方根压缩分布，避免单一专家占主导

- 实验验证这个变体效果更好

3. SwiGLU 激活

每个专家的 FFN 使用 SwiGLU 激活函数：

```
标准 FFN:  output = W_down × ReLU(W_gate × x)
SwiGLU:    output = W_down × (SiLU(W_gate × x) ⊙ (W_up × x))
```




```
static void swiglu(float *out, const float *gate, const float *up, uint64_t n) {
    for (uint64_t i = 0; i < n; i++) {
        out[i] = silu(gate[i]) * up[i];    // ⊙ 是逐元素乘法
    }
}
```

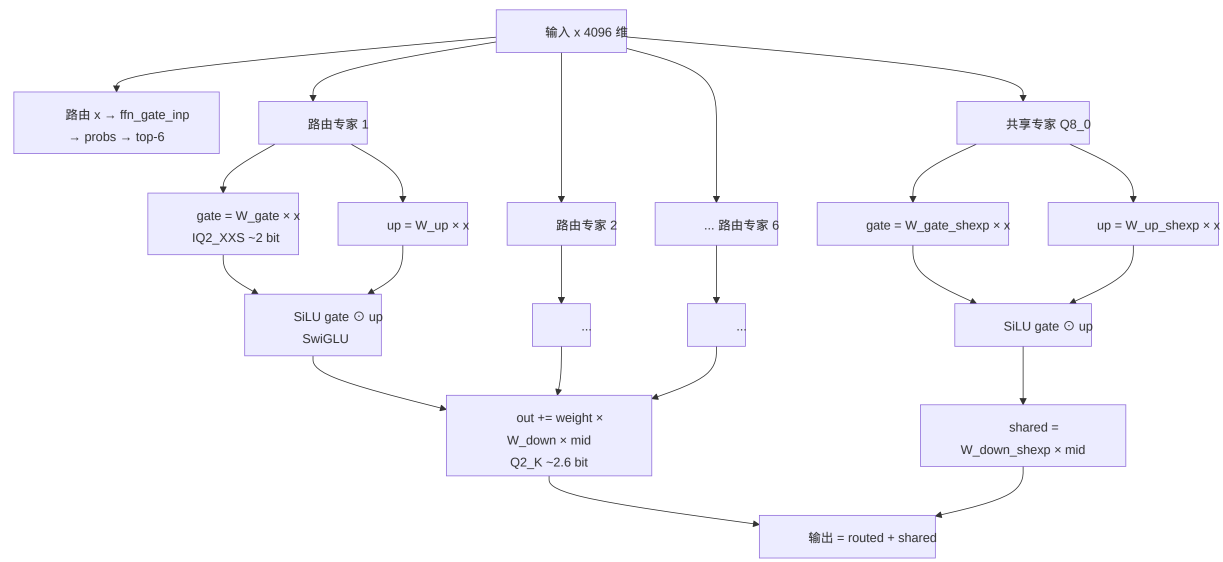
SwiGLU 比 ReLU 效果更好：

- 门控机制（gate）让网络学会"什么时候激活"
- SiLU 的平滑性避免了 ReLU 的"死神经元"问题

在 2-bit 量化中还有额外的 clamp：

```
const float limit = DS4_SWIGLU_CLAMP_EXP;   // 10.0f
if (gate[j] > limit) gate[j] = limit;
if (up[j] > limit) up[j] = limit;
if (up[j] < -limit) up[j] = -limit;
```

4. 完整的 MoE 前向传播



关键洞察：路由专家用 2-bit 量化（省空间），共享专家用 8-bit（保质量）。路由专家的权重占模型大部分空间，所以压缩它们收益最大。

PRO 模型的路由专家使用 Q4_K 量化（~4.5 bit），比 Flash 的 IQ2_XXS 精度更高但体积也更大。ds4 支持三种后端的 PRO Q4_K 推理路径：

- **CPU**：`matvec_q4_k_experts_mid_prequant()` + `ds4_vec_dot_q4_K_q8_K()`，ARM NEON dot-product 加速
- **Metal**：`metal_graph_decode_pro_q4_expert_table_expected()` 优化的专家查找表
- **CUDA**：专用 Q4_K rowspan kernel（`moe_gate_up_mid_q4K_expert_tile8_rowspan_kernel` 等）

5. MoE 的效率优势

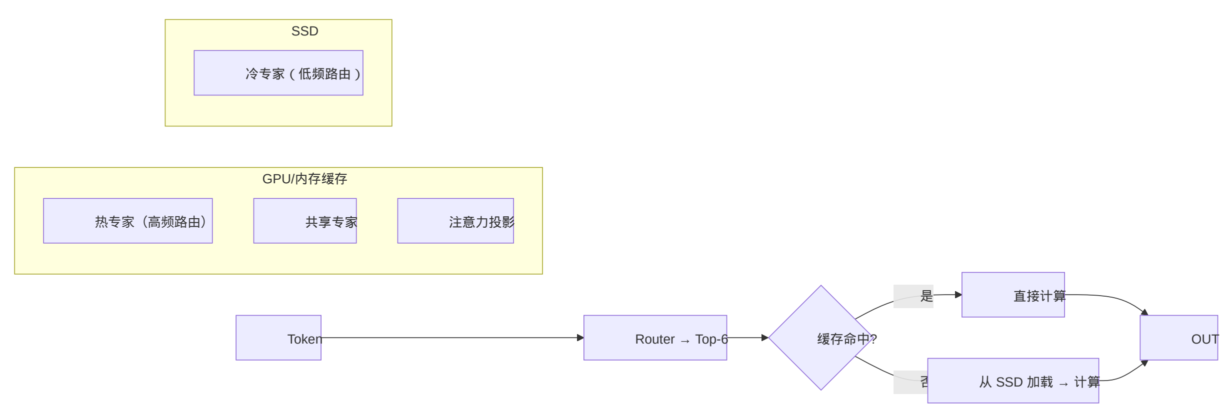
```
284B 参数模型（Flash），每个 token 只激活:
- 6/256 路由专家 × 3 矩阵 × 2048×4096 = 151M 参数
- 1 共享专家 × 3 矩阵 × 2048×4096 = 25M 参数
- 注意力参数: ~100M
- 总计激活: ~276M 参数 (不到总参数的 0.1%)
```

这就是为什么 DeepSeek V4 Flash 虽然有 284B 参数，但推理速度接近 27B 的密集模型。

PRO 模型更大但激活参数仅 49B（总参数 1.6T），同样得益于 MoE 的稀疏激活。详见 分布式推理。

6. MoE 专家缓存与 SSD Streaming

MoE 的稀疏激活（每 token 只用 6/256 专家）天然适合缓存策略——只把"热"专家放 RAM，"冷"专家留磁盘按需加载：



这就是 SSD Streaming 的基础——把"模型能否运行"从"RAM 必须 ≥ 模型大小"的硬截止，变成"RAM 越多越快"的连续速度谱。启动时根据 RAM 预算计算 `ds4_ssd_cache_plan`，按流行度排名（`ds4_streaming_hotlist.inc`）预加载热门专家。Metal 后端（`ds4_metal.m`）实现了完整的异步加载和 LRU 淘汰。

详见 SSD Streaming 术语表和 Part 6 — GPU 加速。

反向链接

/glossary/attention-sink

/glossary/kv-cache

/glossary/lora

/glossary/mla

/glossary/moe

/glossary/quantization

/glossary/rmsnorm

/glossary/softmax

/glossary/swiglu

Part 4: 推理流程

Part 5: 服务层

Part 6: GPU 加速

出链

Part 2: 分词与采样

Part 1: 构建与加载

Part 6: GPU 加速

Part 3: 练习

---

1. 公共 API 与权重结构

练习 1：opaque type 设计

题目：假设你要设计一个 "文件读取器" API，要求：

- 用户可以打开文件、读取一行、关闭
- 不暴露内部缓冲区细节

请写出头文件声明。

参考答案

```c
// filereader.h
typedef struct filereader filereader;

filereader *filereader_open(const char *path);
const char *filereader_readline(filereader *fr);  // 返回 NULL 表示 EOF
void filereader_close(filereader *fr);
```

内部实现可以自由使用任何缓冲策略（mmap、stdio、自定义缓冲），调用者完全不关心。

练习 2：权重绑定追踪

题目：`weights_bind` 中，`blk.3.attn_q_a.weight` 这个张量名如何被找到？追踪查找过程。

参考答案

```
1. required_tensorf(m, "blk.%u.attn_q_a.weight", 3)
   → 格式化为 "blk.3.attn_q_a.weight"

2. required_tensor(m, "blk.3.attn_q_a.weight")
   → 在 m->tensors[0..n_tensors-1] 中线性搜索
   → 比较 ds4_str_eq(tensor.name, "blk.3.attn_q_a.weight")

3. 找到后返回 ds4_tensor 指针
   → 该指针的 abs_offset 指向 mmap 内的数据位置

4. 使用时: tensor_data(model, tensor) → m->map + tensor->abs_offset
```

线性搜索意味着 O(n) 查找，但只在加载时执行一次，不影响推理性能。

练习 3：理解结构体大小

题目：估算 `ds4_engine` 结构体占多少内存（不算指向的数据）。

参考答案

```
ds4_model:  fd(4) + padding(4) + map(8) + size(8) + version(4) + ...
            ≈ ~100 字节
ds4_vocab:  token 指针(8) + n_vocab(4) + 7 个 int(28) + 2 个 hash table(48)
            ≈ ~100 字节
ds4_weights: token_embd 指针(8) + 5 个全局指针(40) + 43 层 × 37 指针(12716)
             ≈ ~12,800 字节

ds4_engine 总计约 ~13 KB（不含指向的实际数据）
```

实际数据（模型权重 81GB）全在 mmap 中，结构体本身很小。

练习 4：LoRA 参数计算

题目：标准 Q 投影需要 `4096 × (64 × 512) = 4096 × 32768` 个参数。LoRA 分解为 `A: [4096, 1024]` 和 `B: [1024, 32768]`。计算参数量节省了多少？

参考答案

```
标准：4096 × 32768 = 134,217,728 (128M) 参数
LoRA：4096 × 1024 + 1024 × 32768 = 4,194,304 + 33,554,432 = 37,748,736 (36M) 参数
节省: (128M - 36M) / 128M = 71.9%
```

LoRA 把参数量减少了约 72%。代价是多一次矩阵乘法，但对于带宽受限的推理（模型大、内存慢），减少内存访问通常更划算。

今日学习检查清单

- [ ] 能解释 opaque type 的目的和实现方式
- [ ] 能画出 ds4_engine / ds4_session 的类型层次
- [ ] 理解 weights_bind 的工作原理（按名称查找张量）
- [ ] 能列出 Transformer 层的主要权重及其作用
- [ ] 理解超连接（Hyper-Connection）与标准残差的区别
- [ ] 能计算 LoRA 分解的参数量节省
- [ ] 理解函数指针回调模式（emit/done）

延伸挑战

挑战 **1**（中级）：画出完整的权重绑定图

`weights_bind` 按名称查找张量并绑定到对应字段。列出 DeepSeek V4 Flash 单层所有需要绑定的张量名称（如 `blk.0.attn_q.weight`），画出名称 → 字段的完整映射表。

挑战 **2**（高级）：估算模型参数量

根据 ds4.h 中的架构常量和本主题学到的权重结构，手动计算 DeepSeek V4 Flash 的总参数量。与官方公布的 284B 对比，解释差异来源。提示：注意 LoRA 的降维和 MoE 的稀疏结构。

---

2. 量化与矩阵运算

练习 **1**：手动量化

题目：把以下浮点数量化为 8-bit 整数（对称量化）：

```
值: [-0.5, 0.3, -1.0, 0.8]
```

步骤：找最大绝对值 → 计算缩放因子 → 量化为 int8

参考答案

```
最大绝对值: 1.0
缩放因子: 1.0 / 127 = 0.00787
量化:
  -0.5 / 0.00787 = -63.5 → -64
   0.3 / 0.00787 =  38.1 →  38
  -1.0 / 0.00787 = -127.0 → -127
   0.8 / 0.00787 = 101.7 → 102

反量化验证:
  -64 × 0.00787 = -0.504
   38 × 0.00787 =  0.299
 -127 × 0.00787 = -0.999
  102 × 0.00787 =  0.803

最大误差: |0.8 - 0.803| = 0.003 (0.375%)
```

练习 2：计算模型大小

题目：DeepSeek V4 Flash 有 284B 参数。计算：

1. FP32 需要多少 GB？
2. Q4（平均 4.5 bit/weight）需要多少 GB？
3. ds4.c 的 Q2 量化（81GB 文件），平均每权重多少 bit？

参考答案

```
1. FP32: 284B × 4 bytes = 1,136 GB ≈ 1.1 TB
2. Q4:   284B × 4.5/8 bytes = 159.75 GB ≈ 160 GB
3. Q2:   81 GB / 284B = 0.285 bytes = 2.28 bit/weight
```

2.28 bit/weight 是"平均"值——实际上非对称量化使得路由专家约 2 bit，其他组件更高。

练习 3：Block 布局分析

题目：`block_q4_K` 的 `qs[128]` 数组存储 256 个 4-bit 值。请解释如何从一个字节中提取两个 4-bit 值。

参考答案

```
uint8_t byte = qs[i];
uint8_t low4  = byte & 0x0F;       // 低 4 位: 值 0-15
uint8_t high4 = (byte >> 4) & 0x0F; // 高 4 位: 值 0-15

// 反量化:
// value = d * (scale * qs_value - dmin * offset)
```

128 字节 × 8 bit = 1024 bit，256 个值 × 4 bit = 1024 bit ✓

练习 4：理解非对称量化策略

题目：ds4.c 对路由专家用 IQ2_XXS（~2 bit），对共享专家用 Q4_K（~4.5 bit）。为什么不统一用同一精度？

参考答案

1. 路由专家占空间最大：256 个专家 × (gate + up + down) 是模型参数的主体。降低它们的精度可以大幅减少总大小
2. 每个 **token** 只用 **6/256** 专家：即使单个专家有量化误差，6 个专家加权平均后误差被稀释
3. 共享专家每次都激活：它对每个 token 都有贡献，量化误差会累积，所以需要更高精度
4. 注意力投影决定模型质量：query/key/value 的精度直接影响注意力计算，不能太低
5. 实际测试验证：这种非对称方案在编码代理测试中表现良好

今日学习检查清单

- [ ] 能手动量化/反量化一组浮点数
- [ ] 理解 Q2_K 和 Q4_K block 的字节布局
- [ ] 能从字节中提取 2-bit 和 4-bit 值
- [ ] 理解查表法在量化解码中的作用
- [ ] 能解释非对称量化策略（为什么不同组件用不同精度）
- [ ] 理解 matvec 操作在推理中的核心地位

延伸挑战

挑战 1（中级）：量化误差实验

随机生成 1000 个 [-1, 1] 的 float 值。分别用 Q4_K 和 Q8_K 格式量化再反量化，计算 MAE（平均绝对误差）和最大误差。对比两种格式的精度差异。

挑战 2（高级）：优化 matvec 的内存访问

阅读 `matvec_f16` 的实现，分析它的缓存访问模式。提出一个优化方案：如何利用分块（tiling）提高缓存命中率？不需要实现，画出分块后的访问模式图即可。

---

3. 注意力机制

练习 1：手动计算注意力

题目：简化模型：2 个头，2 个 KV 行，head_dim = 2。

```
Q[0] = [1.0, 0.0]    Q[1] = [0.0, 1.0]
KV[0] = [1.0, 0.0]   KV[1] = [0.0, 1.0]
```

计算 kq_scale = 1/√2 ≈ 0.707。手动计算头 0 的注意力输出。

参考答案

```
头 0: qh = [1.0, 0.0]

scores:
  score[0] = dot([1,0], [1,0]) × 0.707 = 1.0 × 0.707 = 0.707
  score[1] = dot([1,0], [0,1]) × 0.707 = 0.0 × 0.707 = 0.0

max_score = 0.707
exp(0.707 - 0.707) = 1.0
exp(0.0 - 0.707) = 0.493
denom = 1.0 + 0.493 = 1.493

weights: [1.0/1.493, 0.493/1.493] = [0.670, 0.330]

out[0] = 0.670 × [1,0] + 0.330 × [0,1] = [0.670, 0.330]
```

头 0 主要关注 KV[0]（相似度最高的行）。

练习 2：KV Cache 大小计算

题目：DeepSeek V4 Flash 的配置：43 层，head_dim = 512，SWA = 128，压缩比 ratio=2 的层有 30 个，ratio=4 的层有 10 个，其余无压缩。

计算在 32K 上下文下：

1. 原始 KV cache 的内存（所有层）
2. 压缩 KV 的内存（ratio=2 和 ratio=4）
3. 总 KV cache 大小

参考答案

```
1. 原始 KV（每层最多 128 行）:
   每层: 128 × 512 × 4 bytes = 256 KB
   43 层: 43 × 256 KB = 11 MB

2. 压缩 KV:
   ratio=2 层: 32000/2 + 2 = 16002 行 × 512 × 4 = 32 MB/层
               30 层 × 32 MB = 960 MB

   ratio=4 层: 32000/4 + 2 = 8002 行 × 512 × 4 = 16 MB/层
               10 层 × 16 MB = 160 MB

3. 总计: 11 MB + 960 MB + 160 MB ≈ 1.1 GB

对比: 如果不压缩 (43 × 32K × 512 × 4 = 2.7 GB per head × 64 heads...)
     → 压缩后的 KV cache 极其紧凑
```

练习 3：滑动窗口模拟

题目：`kv_cache_push_raw` 在窗口满时用 `memmove` 滑动。假设 cap_raw=3, head_dim=2：

```
初始: [A, B, C]  n_raw=3 (满)
push D: memmove → [B, C, ?]  然后 → [B, C, D]
push E: memmove → [C, D, ?]  然后 → [C, D, E]
```

为什么用 `memmove` 而不是 `memcpy`？

参考答案

`memmove` 保证源和目标区域重叠时也能正确工作。这里 `raw_kv` 既是源也是目标（向前移动一行），内存区域重叠。`memcpy` 在重叠情况下是未定义行为——可能正确，也可能产生错误结果。

练习 4：Sink 的作用

题目：`attn_sinks` 是一个可学习的 logit 值，参与 softmax 但不贡献 value。请解释：如果没有 sink，当所有 score 都很低时会发生什么？

参考答案

没有 sink 时:

```
所有 score 都很小（比如 -10, -10, -10）
exp(-10) ≈ 0.0000454
denom = 3 × 0.0000454 = 0.000136
weight[0] = 0.0000454 / 0.000136 ≈ 0.333
→ 均匀分布，模型被迫平均关注所有位置
```

有 sink 时:

```
sink = 5.0 (可学习，较高)
scores = [-10, -10, -10]
max = 5.0
exp(5.0-5.0) = 1.0 (sink)
exp(-10-5.0) = 3.06e-7 (其他位置)
denom ≈ 1.0 + 3 × 3.06e-7 ≈ 1.0
→ sink 占了几乎 100% 的权重，其他位置被忽略
→ out ≈ 0（sink 不贡献 value）
```

Sink 让模型学会"不关注任何位置"的能力。

今日学习检查清单

- ☐ 能手动计算简单的注意力分数和输出
- ☐ 理解 softmax 数值稳定性的实现
- ☐ 能解释 KV Cache 的必要性和工作方式
- ☐ 理解滑动窗口注意力的滑动机制
- ☐ 能解释 Attention Sink 的作用
- ☐ 理解 MLA 压缩如何减少 KV cache 大小

- ☐ 能描述混合注意力（raw + compressed）的工作方式
- ☐ 理解多维数组在一维内存中的索引方式

延伸挑战

挑战 **1**（中级）：计算不同上下文长度的 **KV Cache** 开销

在 32K、128K、512K、1M 四种上下文长度下，分别计算原始 KV + 压缩 KV 的总内存（参考练习 2 的方法）。画出"上下文长度 vs 内存"的曲线，找出消费级硬件（128GB 统一内存）能支持的最大上下文长度。

挑战 **2**（高级）：对比 **MQA** 和 **MHA** 的 **tradeoff**

DeepSeek 用 64:1 的 Multi-Query Attention（64 个 Q 头，1 组 KV）。如果改为 Grouped-Query Attention（64 个 Q 头，8 组 KV），KV Cache 增加多少？注意力质量理论上如何变化？在 ds4.c 中找到需要修改的常量。

---

4. MoE 混合专家

练习 1：Top-K 选择模拟

题目：8 个专家的分数：`[0.3, 0.8, 0.1, 0.5, 0.9, 0.2, 0.7, 0.4]`。手动模拟 `topk_desc` 选择 top-3。

参考答案

```
扫描 0.3: idx = [0, -1, -1]
扫描 0.8: idx = [1, 0, -1]  (0.8 > 0.3)
扫描 0.1: idx = [1, 0, -1]  (0.1 < 0.3，跳过)
扫描 0.5: idx = [1, 0, 3]   (0.5 > 0.3)
扫描 0.9: idx = [4, 1, 0]   (0.9 > 0.8 > 0.5 > 0.3，把 0.3 挤出去)
扫描 0.2: idx = [4, 1, 0]   (0.2 < 0.3，跳过)
扫描 0.7: idx = [4, 1, 6]   (0.7 > 0.5 > 0.3)
扫描 0.4: idx = [4, 1, 6]   (0.4 < 0.5，跳过)

结果: selected = [4, 1, 6]（分数 0.9, 0.8, 0.7）
```

练习 2：专家权重计算

题目：基于练习 1 的结果，计算专家权重：

- probs = [0.3, 0.8, 0.1, 0.5, 0.9, 0.2, 0.7, 0.4]
- selected = [4, 1, 6]
- DS4_EXPERT_WEIGHT_SCALE = 1.5

参考答案

```
未归一化权重:
  w[0] = probs[4] = 0.9
  w[1] = probs[1] = 0.8
  w[2] = probs[6] = 0.7

sum = 0.9 + 0.8 + 0.7 = 2.4

归一化 × scale:
  w[0] = (0.9 / 2.4) × 1.5 = 0.5625
  w[1] = (0.8 / 2.4) × 1.5 = 0.5000
  w[2] = (0.7 / 2.4) × 1.5 = 0.4375

总和 = 1.5（等于 DS4_EXPERT_WEIGHT_SCALE）
```

专家 4 获得最高权重（0.5625），因为它的路由分数最高。

练习 3：SwiGLU 计算

题目：假设 gate = [2.0, -1.0, 0.5], up = [1.0, 1.0, 1.0]。计算 SwiGLU 输出。

提示：sigmoid(x) = 1/(1+exp(-x)), silu(x) = x × sigmoid(x)

参考答案

```
gate[0] = 2.0: silu(2.0) = 2.0 × sigmoid(2.0) = 2.0 × 0.881 = 1.762
              out[0] = 1.762 × 1.0 = 1.762

gate[1] = -1.0: silu(-1.0) = -1.0 × sigmoid(-1.0) = -1.0 × 0.269 = -0.269
               out[1] = -0.269 × 1.0 = -0.269

gate[2] = 0.5: silu(0.5) = 0.5 × sigmoid(0.5) = 0.5 × 0.622 = 0.311
              out[2] = 0.311 × 1.0 = 0.311

输出: [1.762, -0.269, 0.311]
```

SiLU 对负值产生接近零的输出（自门控），对正值接近线性。

练习 4：MoE 效率计算

题目：284B 参数模型，每个 token 激活 6/256 路由专家 + 1 共享专家。计算：

1. 总共有多少路由专家参数？（假设每个专家 3 个 2048×4096 矩阵）
2. 每个 token 激活多少路由专家参数？
3. 激活比例是多少？

参考答案

```
1. 每个路由专家参数: 3 × 2048 × 4096 = 25,165,824 (25M)
   256 个路由专家: 256 × 25M = 6.4B

2. 每次激活: 6 × 25M = 150M + 1 × 25M(共享) = 175M

3. 路由专家激活比: 150M / 6.4B = 2.34%

如果模型剩余 ~277.6B 参数是注意力/嵌入/其他，
总激活: ~175M + ~277.6B = ~277.8B
实际节省不多（因为非专家参数每次都激活）

但关键区别：6.4B 路由专家权重是推理的带宽瓶颈
→ 只需从内存读取 150M 而非 6.4B → 大幅减少内存带宽
```

今日学习检查清单

- [ ] 能手动模拟 top-k 选择过程
- [ ] 理解哈希路由和 top-k 路由的区别
- [ ] 能计算专家权重（归一化 × scale）
- [ ] 能手动计算 SwiGLU 输出
- [ ] 理解为什么用 √(softplus(x)) 而不是 softmax 做路由
- [ ] 能计算 MoE 的激活参数比例

- [ ] 理解共享专家和路由专家的分工

延伸挑战

挑战 1（中级）：**追踪 token** 的专家选择

在 ds4.c 的 MoE 路由代码中加日志，记录一个真实 token 在 43 层中分别选择了哪些专家。观察：是否有某些专家被频繁选择？前 3 层（Hash 路由）和后续层（Top-K 路由）的专家分布有何不同？

挑战 2（高级）：**MoE vs Dense** 的计算量对比

假设把 256 个 MoE 专家合并成一个等价的 Dense FFN（隐藏维度 = 256 × 2048），计算单层前向的计算量（FLOPs）对比。为什么 MoE 更适合推理而非训练？

Part 3: 代码走读

---

1. 公共 **API** 与权重结构

追踪 **1**：权重结构层次

三级结构

```
ds4_weights                           全局模型权重
  ├── token_embd                      词嵌入矩阵 [DS4_N_EMBD, DS4_N_VOCAB]
  ├── output_hc_base/fn/scale         超连接输出头（3 个张量）
  ├── output_norm                     输出层 RMSNorm 权重
  ├── output                          最终输出投影
  └── layer[43]                       43 层，每层一个 ds4_layer_weights

ds4_layer_weights                     单层权重（行 1987-2023）
  ├── HC 注意力控制
  │     ├── hc_attn_fn/scale/base     超连接参数（3 个张量）
  ├── 注意力机制
  │     ├── attn_norm                 注意力 RMSNorm
  │     ├── attn_q_a / attn_q_a_norm / attn_q_b   两级低秩 Q 投影
  │     ├── attn_kv / attn_kv_a_norm KV 投影 + Norm
  │     ├── attn_sinks                Sink 注意力参数
  │     ├── attn_output_a / attn_output_b          两级低秩输出投影
  │     ├── attn_compressor_*         KV 压缩器（条件绑定）
  │     └── indexer_*                 索引器（仅 ratio==4 层）
  ├── HC FFN 控制
  │     └── hc_ffn_fn/scale/base      FFN 超连接参数
  └── FFN / MoE
        ├── ffn_norm                  FFN RMSNorm
        ├── ffn_gate_tid2eid          Hash 路由表（仅 Hash 层）
        ├── ffn_gate_inp              路由器权重
        ├── ffn_exp_probs_b           专家概率偏置（可选）
        ├── ffn_gate/up/down_exps   256 个专家权重
        └── ffn_gate/up/down_shexp  共享专家权重

ds4_mtp_weights                       多 token 预测头（行 2035-2045）
  ├── e_proj / h_proj               嵌入/隐藏投影
  ├── enorm / hnorm / norm          归一化层
  ├── hc_head_base/fn/scale         超连接头
  └── block                         一整套 ds4_layer_weights
```

追踪 2：权重绑定过程

调用链

```
ds4_engine_open()                        [行 16950]
  └── weights_bind(&e->weights, &e->model)  [行 2578]
        │
        ├── memset(w, 0, sizeof(*w))           清零结构体
        │
        ├── 绑定顶层张量（按 GGUF 名称查找）
        │     required_tensor(m, "token_embd.weight")     → w->token_embd
        │     required_tensor(m, "output_hc_base.weight") → w->output_hc_base
        │     required_tensor(m, "output_norm.weight")    → w->output_norm
        │     required_tensor(m, "output.weight")         → w->output
        │
        ├── for il = 0..42:                     逐层绑定
        │     compress_ratio = ds4_layer_compress_ratio(il)
        │
        │     绑定通用注意力张量:
        │     required_tensorf("blk.%d.attn_norm.weight", il)
        │     required_tensorf("blk.%d.attn_q_a.weight", il)
        │     ...
        │
        │     if (compress_ratio != 0):         压缩层额外绑定
        │       required_tensorf("blk.%d.attn_compressor_kv.weight", il)
        │
        │     if (compress_ratio == 4):         索引器层额外绑定
        │       required_tensorf("blk.%d.indexer_proj.weight", il)
        │
        │     ffn_exp_probs_b = tensor_by_namef(...)  ← 可选！
        │
        │     if (il < DS4_N_HASH_LAYER):       Hash 路由层绑定
        │       required_tensorf("blk.%d.ffn_gate_tid2eid.weight", il)
        │
        └── weights_validate_layout(w)          [行 2285] 验证所有张量维度
```

条件绑定的内存布局

```
                    Layer 0-28        Layer 29-36       Layer 37-42
compress_ratio:         0                  4                  8
                     ┌──────────┐     ┌──────────┐     ┌──────────┐
attn_norm           │    ✓     │     │    ✓     │     │    ✓     │
attn_q_a/b          │    ✓     │     │    ✓     │     │    ✓     │
compressor_kv       │          │     │    ✓     │     │    ✓     │
indexer_proj        │          │     │    ✓     │     │          │
ffn_gate_tid2eid    │    ✓     │     │          │     │          │
                     └──────────┘     └──────────┘     └──────────┘
```

追踪 3：Engine / Session 生命周期

Engine 初始化

```c
// 行 16950: 引擎打开
int ds4_engine_open(ds4_engine **out, const ds4_engine_options *opt) {
    ds4_engine *e = xcalloc(1, sizeof(*e));
    e->fd = -1;                              // 文件描述哨兵
    e->backend = opt->backend;               // CPU / Metal / CUDA
    e->mtp_draft_tokens = CLAMP(opt->mtp_draft_tokens, 1, 16);
    e->mtp_margin = opt->mtp_margin < 0 ? 3.0f : opt->mtp_margin;

    // 核心加载流程
    model_open(&e->model, opt->model_path, ...);  // mmap GGUF
    vocab_load(&e->vocab, &e->model);              // 分词器
    config_validate_model(&e->model);              // 参数校验
    weights_bind(&e->weights, &e->model);          // 指针绑定

    // GPU 后端初始化
    if (backend == DS4_BACKEND_METAL) {
        ds4_gpu_init();
        ds4_gpu_set_model_map(e->model.map, e->model.size);
    }
}
```

Session 创建与释放

```
ds4_session_create(&sess, engine, ctx_size)   [行 17094]
  │
  ├── CPU 路径:
  │     kv_cache_init(&s->cache, ctx_size)      分配 KV 缓存
  │     cpu_decode_scratch_init(&s->scratch)    分配解码暂存区
  │     s->logits = xmalloc(DS4_N_VOCAB * sizeof(float))  129280 × 4 字节
  │
  └── GPU 路径:
        metal_graph_alloc_raw_cap(...)          分配 Metal 计算图
        s->logits = ds4_gpu_tensor_alloc(...)

ds4_session_free(s)                             [行 17142]
  ├── kv_cache_free / scratch_free  (CPU)
  ├── ds4_gpu_graph_free            (GPU)
  ├── free(s->checkpoint.data)
  ├── free/ds4_gpu_tensor_free(s->logits)
  └── free(s)
```

追踪 4：Session 同步与前缀复用

同步策略

```
// 行 17201: 将 session 状态推进到指定 prompt
int ds4_session_sync(ds4_session *s, const ds4_tokens *prompt, ...) {
```

```
场景 1: 完全匹配（checkpoint 是 prompt 的前缀）

checkpoint: [A, B, C]              已处理 3 个 token
prompt:     [A, B, C, D, E, F]     需要 6 个 token

  → 增量计算 D, E, F
  → 如果 suffix 长度 <= 8: 逐 token decode
  → 如果 suffix 长度 > 8:  批量 prefill

场景 2: 无匹配

checkpoint: [A, B, C]
prompt:     [X, Y, Z]

  → 丢弃 checkpoint
  → 从头 prefill [X, Y, Z]
```

公共前缀查找

```
// 行 17397: O(n) 线性比较
int ds4_session_common_prefix(ds4_session *s, const ds4_tokens *prompt) {
    if (!s->checkpoint.data) return 0;          // 无缓存，直接返回 0
    int n = MIN(s->checkpoint.len, prompt->len);
    for (int i = 0; i < n; i++) {
        if (s->checkpoint.data[i] != prompt->data[i]) return i;  // 不匹配
    }
    return n;                                   // 全部匹配
}
```

改写验证

```c
// 行 17360: 从公共前缀处改写
ds4_session_rewrite_result ds4_session_rewrite_from_common(
        ds4_session *s, const ds4_tokens *prompt, int common, ...) {
    // 验证前 common 个 token 确实一致
    for (int i = 0; i < common; i++) { ... }
    // 如果 common == checkpoint.len → 可以直接 ds4_session_sync
    // 否则检查是否需要重建 KV cache
}
```

追踪 5：公共采样 API

一行代理模式

```c
// 行 17405: 贪心采样
int ds4_session_argmax(ds4_session *s) {
    return sample_argmax(s->logits, DS4_N_VOCAB);
}

// 行 17424: 带参采样
int ds4_session_sample(ds4_session *s,
        float temperature, int top_k, float top_p, float min_p, uint64_t *rng) {
    return sample_top_p_min_p(s->logits, DS4_N_VOCAB,
                              temperature, top_k, top_p, min_p, rng);
}
```

API 调用层次

```
用户代码
  │
  ├── ds4_session_argmax(s)
  │     └── sample_argmax(s->logits, 129280)        [行 14874]
  │           └── 线性扫描最大值
  │
  └── ds4_session_sample(s, temp, k, p, min_p, rng)
        └── sample_top_p_min_p(...)                   [行 15023]
              ├── temperature <= 0 → sample_argmax    退化为贪心
              ├── top_k <= 0 → sample_full_vocab      全词表采样
              │                     ├── 快速路径 (top_p >= 1.0): 无排序
              │                     └── 排序路径: qsort + 截断
              └── top_k > 0 → 栈上插入排序 + min-p + top-p 过滤
```

ds4.h 的设计模式：对外只暴露 `ds4_session_argmax` 和 `ds4_session_sample` 两个简单接口，内部自动路由到最优实现。

---

2. 量化与矩阵运算

追踪 1：量化块结构体

编译期大小验证

```
// 行 157: 利用数组大小为负的 typedef 触发编译错误
#define DS4_STATIC_ASSERT(name, cond) typedef char name[(cond) ? 1 : -1]
//                                                      ^^^^   ^^^^
//                                                      条件真   条件假 → 负大小 → 编
译失败
```

四种量化块

```c
// 行 132: Q2_K — 2-bit 量化，84 字节/块（256 个权重）
typedef struct {
    uint8_t  scales[QK_K / 16];   // 16 个缩放因子    = 16 字节
    uint8_t  qs[QK_K / 4];        // 256×2bit/8=64 字节 = 64 字节
    uint16_t d;                    // 块缩放            = 2 字节
    uint16_t dmin;                 // 最小值偏移        = 2 字节
} block_q2_K;                     //                      = 84 字节 ✓

// 行 139: Q4_K — 4-bit 量化，144 字节/块
typedef struct {
    uint16_t d;                    // = 2 字节
    uint16_t dmin;                 // = 2 字节
    uint8_t  scales[12];           // = 12 字节
    uint8_t  qs[QK_K / 2];        // 256×4bit/8=128 字节
} block_q4_K;                     //   = 144 字节 ✓

// 行 146: Q8_K — 8-bit 量化，292 字节/块
typedef struct {
    float    d;                    // = 4 字节
    int8_t  qs[QK_K];             // 256 字节
    int16_t bsums[QK_K / 16];     // 32 字节（用于矩阵乘分块累加）
} block_q8_K;                     //   = 292 字节 ✓

// 行 152: IQ2_XXS — 超低比特，66 字节/块
typedef struct {
    uint16_t d;                    // = 2 字节
    uint16_t qs[QK_K / 8];        // 256/8=32 个索引 × 2 字节 = 64 字节
} block_iq2_xxs;                  //   = 66 字节 ✓
```

每个 Q2_K 字节的权重存储

```
qs[0] = 0b11_10_01_00    ← 4 个 2-bit 权重打包在 1 字节中
        ^^ ^^ ^^ ^^
        v3 v2 v1 v0       值 0-3，需要 scales 和 d/dmin 解码为实际浮点值
```

追踪 2：F16 转换与 FP8 量化

IEEE 754 半精度转单精度

```
// 行 1514: f16 → f32 纯软件实现（ARM NEON 有硬件加速）
static inline float f16_to_f32(uint16_t h) {
    // ARM 路径: vcvt_f32_f16 硬件指令，1 周期
    // 标量路径: 手动位操作
    const uint32_t sign = (h >> 15) & 1;      // 最高位
    const uint32_t exp  = (h >> 10) & 0x1f;   // 5 位指数
    const uint32_t mant = h & 0x3ff;          // 10 位尾数
    // 非正规数处理: exp==0 && mant!=0 → while 循环正规化
}
```

```
F16 (16 bit):               F32 (32 bit):
S EEEEE MMMMMMMMMM          S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
1  5bit   10bit             1   8bit          23bit

指数偏移: F16=15, F32=127
尾数位差: 23 - 10 = 13（需要左移 13 位补齐）
```

FP8 KV 缓存原地量化

```
// 行 1635: E4M3FN 动态量化（FP8 格式）
static void dsv4_fp8_kv_quantize_row_inplace_cpu(
        float *x, uint32_t head_dim, uint32_t n_rot) {
    const uint32_t n_nope = head_dim - n_rot;  // 非 RoPE 部分
    // 每 64 个元素一块
    for (uint32_t off = 0; off < n_nope; off += 64) {
        float amax = 找到块内最大绝对值;
        amax = fmaxf(amax, 1.0e-4f);           // 防止除零

        // 动态缩放: ldexp 形式的 E4M3FN
        float scale = ldexpf(1.0f, ceilf(log2f(amax / 448.0f)));
        //                                      ^^^^
        //                                   E4M3FN 最大可表示值 = 448

        for (int i = 0; i < block_size; i++) {
            float v = x[off + i] / scale;
            v = clamp(v, -448.0f, 448.0f);
            x[off + i] = dsv4_e4m3fn_dequant_cpu(v) * scale;  // 量化后立即反量化
        }
    }
}
```

```
原始 float (32bit) → 量化 FP8 (8bit) → 反量化 float (32bit)
    3.14159              0b_0_1000_011              3.125
                         ↑    ↑    ↑
                         S   E4   M3

精度损失不可避免，但节省 4× KV 缓存内存
```

追踪 3：IQ2_XXS 查找表点积

预计算查找表

```
// 行 307: 初始化 256×128×8 的三维查找表
static void iq2xxs_signed_grid_init(void) {
    // 第一步: 构建 128 种符号模式
    for (int i = 0; i < 128; i++) {
        // ksigns_iq2xs[i] → 8 个 ±1 符号
        // kmask_iq2xs[i]  → 8 位掩码
    }
    // 第二步: 组合 256 个网格点 × 128 种符号
    for (int grid_idx = 0; grid_idx < 256; grid_idx++) {
        for (int sign_idx = 0; sign_idx < 128; sign_idx++) {
            // iq2xxs_signed_grid[grid_idx][sign_idx][8]
            // = iq2xxs_grid[grid_idx] × 符号模式[sign_idx]
        }
    }
}
```

点积计算（3 路硬件加速）

```
// 行 327: 16 元素对点积
static inline int32_t dot_iq2_pair_16(
        const int8_t *grid0,      // 第一个网格查表结果（8 个值）
        const int8_t *grid1,      // 第二个网格查表结果（8 个值）
        const int8_t *q8) {       // Q8 激活值（16 个 int8）
    // ARM NEON + dotprod:
    int32x4_t s0 = vdotq_s32(vdupq_n_s32(0), v0, q8_0);  // 硬件点积
    int32x4_t s1 = vdotq_s32(vdupq_n_s32(0), v1, q8_1);

    // ARM NEON 无 dotprod:
    int16x8_t p0 = vmull_s8(v0, q8_0);   // 8 位乘法
    acc = vpaddlq_s16(p0);                // 成对加法

    // 标量回退:
    int32_t sum = 0;
    for (int i = 0; i < 8; i++)
        sum += grid0[i] * q8[i] + grid1[i] * q8[8 + i];
}
```

```
IQ2_XXS 反量化流水线:

索引 qs[i] (8-bit) → 查表 iq2xxs_grid[qs[i]] → 8 个 2-bit 值
                                   ↓
               × 符号模式 → 8 个 int8 值
                                   ↓
               · Q8 激活 → 点积累加
```

追踪 4：Q8_0 激活量化与 matvec

Q8_0 量化（块大小 32）

```
// 行 3123: 激活值 float → int8 量化
static void quantize_q8_0_activation(
        const float *x, int8_t *xq, float *scale, uint64_t n) {
    const uint64_t blocks = (n + 31) / 32;    // 块大小 32（不是 256）

    for (uint64_t b = 0; b < blocks; b++) {
        // 找块内最大绝对值
        float amax = 0.0f;
        for (int i = 0; i < 32; i++)
            amax = fmaxf(amax, fabsf(x[b * 32 + i]));

        float d = amax / 127.0f;              // 缩放因子
        float id = (d != 0.0f) ? 1.0f / d : 0.0f;

        scale[b] = d;                         // 存储缩放因子
        for (int i = 0; i < 32; i++) {
            xq[b * 32 + i] = (int8_t)clamp(lrintf(x[b*32+i] * id), -128, 127);
        }
    }
}
```

Decode 热路径 matvec

```c
// 行 3494: 单 token Q8_0 矩阵向量乘
static void matvec_q8_0(float *out, const ds4_model *m,
                         const ds4_tensor *w, const float *x) {
    matvec_q8_0_rows(out, m, w, x, 0, w->dim[1]);  // 委托给行范围版本
}

// 行 3510: Decode 热路径（零分配）
static void matvec_q8_0_decode_scratch(
        float *out, const ds4_model *m, const ds4_tensor *w,
        const float *x, ds4_cpu_decode_scratch *scratch) {
    cpu_decode_quantize_q8_0(scratch, x, in_dim);  // 量化到预分配暂存区
    matvec_q8_0_prequant(out, m, w, scratch);       // 用预量化结果做 matvec
}
```

```
Decode 每层的 matvec 调用:

float x[4096]  →  quantize_q8_0  →  int8 xq[4096] + float scale[128]
                                          |
                                          ▼ matvec_q8_0_prequant
                                          |
float out[dim]  ←  Q8_0 权重矩阵 × Q8_0 激活  ←  权重已在模型文件中预量化
```

追踪 5：Q2_K / IQ2_XXS 专家 matvec

IQ2_XXS 专家对投影

```c
// 行 3775: 同时计算一个专家的 gate 和 up 投影
static void matvec_iq2_xxs_expert_pair_prequant(
        float *out0, float *out1,             // gate 输出和 up 输出
        const ds4_model *m,
        const ds4_tensor *w0, *w1,            // gate 和 up 权重
        const block_q8_K *xq,                 // 共享的 Q8_K 激活
        uint32_t expert) {                    // 专家编号
    // 验证两个张量同为 IQ2_XXS 类型
    // 计算专家偏移: base + expert * expert_bytes
    // 填充 pair_ctx，调用并行内核
}
```

Q2_K 专家 Down 投影

```c
// 行 3909: 单专家 Q2_K 下投影（诊断/追踪用）
static void matvec_q2_k_expert(
        float *out, const ds4_model *m,
        const ds4_tensor *w, const float *x,
        uint32_t expert) {
    // 注: 这个版本内部做 Q8_K 量化（xmalloc），不在热路径
    block_q8_K *xq = xmalloc(...);              // 临时分配
    ds4_quantize_row_q8_K(x, xq, in_dim);       // 量化激活
    ds4_parallel_for(out_dim, matvec_q2_k_worker, &ctx);
    free(xq);                                    // 释放临时缓冲
}
```

MoE matvec 数据流

```
输入激活 float x[in_dim]
    │
    ▼ quantize Q8_K（一次）
    │
    xq[QK_K 块...]  ← 所有 6 个专家共享
    │
    ├── 专家 0: matvec_iq2_xxs_expert_pair(w_gate[0], w_up[0], xq)
    │              → gate[0][mid_dim], up[0][mid_dim]
    │
    ├── 专家 1: matvec_iq2_xxs_expert_pair(w_gate[1], w_up[1], xq)
    │              → gate[1][mid_dim], up[1][mid_dim]
    │
    ├── ... (共 6 个专家)
    │
    ▼ SwiGLU: mid[i] = silu(gate[i]) * up[i]
    │
    ▼ quantize Q8_K（per-expert）
    │
    ├── 专家 0: matvec_q2_k(w_down[0], midq[0]) → out += weight[0]
    ├── 专家 1: matvec_q2_k(w_down[1], midq[1]) → out += weight[1]
    └── ...
    │
    ▼ 加权累加
    │
float out[out_dim] = Σ weight[i] * expert_down[i]
```

3. 注意力机制

追踪 1：RMS 归一化

三种 RMSNorm 变体

```
// `rms_norm_no_weight()` 中（约 2700 行）: 无权重（用于 HC 控制向量）
static void rms_norm_no_weight(float *out, const float *x,
                                uint64_t n, float eps) {
    double ss = 0.0;                          // double 精度累加
    for (uint64_t i = 0; i < n; i++) ss += (double)x[i] * x[i];
    float scale = 1.0f / sqrtf((float)(ss / n) + eps);
    for (uint64_t i = 0; i < n; i++) out[i] = x[i] * scale;
}

// `rms_norm_weight()` 中（约 2709 行）: 带可学习权重（用于层归一化）
static void rms_norm_weight(float *out, const float *x,
                             const float *weight, uint64_t n, float eps) {
    // ... 同上计算 scale ...
    for (uint64_t i = 0; i < n; i++)
        out[i] = x[i] * scale * weight[i];  // 逐通道乘以学习到的 γ
}

// `head_rms_norm_inplace()` 中（约 2718 行）: 单头原地归一化（用于 Q/KV 投影后）
static void head_rms_norm_inplace(float *x, uint32_t n_head,
                                   uint32_t head_dim, float eps) {
    for (uint32_t h = 0; h < n_head; h++) {
        float *xh = x + (uint64_t)h * head_dim;
        rms_norm_no_weight(xh, xh, head_dim, eps);  // 每头独立归一化
    }
}
```

```
RMSNorm 公式:  out[i] = x[i] / sqrt(mean(x²) + ε) × weight[i]

LayerNorm vs RMSNorm:
  LayerNorm: (x - μ) / sqrt(σ² + ε) × γ + β      ← 减均值、有偏移
  RMSNorm:    x / sqrt(mean(x²) + ε) × γ           ← 不减均值、无偏移

RMSNorm 省略了均值计算和偏移参数，计算更简单，推理更快
```

追踪 2：HC 头压缩（MLA）

Sinkhorn 分解

```
// `hc_split_sinkhorn_one()` 中（约 4186 行）：将混合向量分解为 pre/post/comb 三个输
出
static void hc_split_sinkhorn_one(
        float *out,           // 输出: [2*n_hc + n_hc*n_hc] 元素
        const float *mix,     // 输入: 混合系数
        const float *scale,   // [pre_scale, post_scale, comb_scale]
        const float *base,    // 基础偏置
        int n_hc, int iters, float eps) {
    // Sigmoid 生成 pre 权重: 1/(1 + exp(-z))，z = mix[i]*pre_scale + base[i]
    // 双重归一化的 comb 矩阵（Sinkhorn 迭代）
}
```

HC Pre / Post 操作

```c
// `hc_pre_from_state_one()` 中（约 4317 行）: 从隐藏状态计算 HC 控制向量
static void hc_pre_from_state_one(
        const ds4_model *model,
        const ds4_tensor *fn,            // 投影权重
        const ds4_tensor *scale_tensor,  // 缩放参数
        const ds4_tensor *base_tensor,   // 基础偏置
        const float *residual_hc,        // HC 残差状态
        float *out, float *post, float *comb) {
    // 分配暂存区，调用 hc_pre_from_state_one_scratch
    // 输出: pre 权重、post 门控、comb 组合矩阵
}

// `hc_post_one()` 中（约 4366 行）: 将子层输出与 HC 残差合并
static void hc_post_one(
        float *out_hc, const float *block_out,
        const float *residual_hc, const float *post,
        const float *comb, uint32_t n_embd, uint32_t n_hc) {
    for (uint32_t dst = 0; dst < n_hc; dst++) {
        for (uint32_t d = 0; d < n_embd; d++) {
            out_hc[dst * n_embd + d] = block_out[d] * post[dst];
            for (uint32_t src = 0; src < n_hc; src++) {
                out_hc[dst*n_embd + d] += comb[dst + src*n_hc]
                                        * residual_hc[src*n_embd + d];
            }
        }
    }
}
```

```
HC（Hyper-Connection）数据流:

residual_hc[n_hc × n_embd]
    │
    ▼ hc_pre_from_state_one
    │
    ├── pre[n_hc]       ← 每个子层输入的门控
    ├── post[n_hc]      ← 每个子层输出的门控
    └── comb[n_hc×n_hc] ← 子层间的路由矩阵
    │
    ▼ 子层计算（注意力或 FFN）
    │
    block_out[n_embd]
    │
    ▼ hc_post_one
    │
    out_hc[dst] = block_out × post[dst]
                + Σ comb[dst,src] × residual_hc[src]
```

追踪 3：Q / KV 投影

两级低秩 Q 投影

```
// `layer_q_projection_normed_one()` 中（约 4596 行）: Q = norm(proj_b(norm(proj_
a(x))))
static void layer_q_projection_normed_one(
        const ds4_model *model, const ds4_layer_weights *layer,
        const float *norm, float *q) {
    // 第一级: 4096 → 1024（低秩压缩）
    float qr[1024];  // 中间维度
    matvec_q8_0(qr, model, layer->attn_q_a, norm);

    // 中间归一化
    rms_norm_weight(qr, qr, layer->attn_q_a_norm, 1024, DS4_RMS_EPS);

    // 第二级: 1024 → 64×512 = 32768（展开到所有头）
    matvec_q8_0(q, model, layer->attn_q_b, qr);

    // 头级归一化
    head_rms_norm_inplace(q, DS4_N_HEAD, DS4_N_HEAD_DIM, DS4_RMS_EPS);
}
```

KV 投影（单头）

```
// `layer_kv_projection_normed_one()` 中（约 4633 行）: KV = norm(proj(x))，输出维
度 = DS4_N_HEAD_DIM = 512
static void layer_kv_projection_normed_one(
        const ds4_model *model, const ds4_layer_weights *layer,
        const float *normed, float *kv) {
    float raw[DS4_N_HEAD_DIM];  // 512
    matvec_q8_0(raw, model, layer->attn_kv, normed);
    rms_norm_weight(kv, raw, layer->attn_kv_a_norm,
                    DS4_N_HEAD_DIM, DS4_RMS_EPS);
}
```

```
DeepSeek V4 的 GQA (Grouped Query Attention):

      Q: 64 头 × 512 维 = 32768 维
     KV:  1 头 × 512 维 =   512 维

  Q 投影: 4096 → 1024 → 32768 (两级低秩)
  KV 投影: 4096 → 512 (单级)

  所有 64 个 Q 头共享 1 组 KV → GQA 64:1
  KV 体积减小 64×，是 MLA 的核心优化
```

追踪 **4**：单 **token** 注意力计算

Sink-Aware 注意力

```c
// `layer_attention_rows_one()` 中（约 4897 行）：带 Sink 的缩放点积注意力
static void layer_attention_rows_one(
        float *out_heads, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *q, const float *kv_rows, uint32_t n_kv) {

    for (uint32_t h = 0; h < DS4_N_HEAD; h++) {
        const float *qh = q + (uint64_t)h * DS4_N_HEAD_DIM;  // 当前头的 Q
        float *oh = out_heads + (uint64_t)h * DS4_N_HEAD_DIM; // 当前头输出

        // 加载 Sink（可学习的注意力汇聚点）
        const float sink_logit = tensor_data(layer->attn_sinks, ...)[h];

        float scores[512];  // 栈缓冲（n_kv <= 512 时）
        // 或堆分配（n_kv > 512 时）

        float max_score = sink_logit;   // Sink 参与最大值计算

        // 计算注意力分数
        for (uint32_t r = 0; r < n_kv; r++) {
            scores[r] = dot_f32(qh, kv_rows + r * DS4_N_HEAD_DIM,
                                DS4_N_HEAD_DIM) / sqrtf(DS4_N_HEAD_DIM);
            if (scores[r] > max_score) max_score = scores[r];
        }

        // Softmax
        float sum = expf(sink_logit - max_score);  // Sink 贡献分母但不贡献分子
        for (uint32_t r = 0; r < n_kv; r++) {
            scores[r] = expf(scores[r] - max_score);
            sum += scores[r];
        }
        float inv_sum = 1.0f / sum;

        // 加权求值（Sink 不贡献值向量）
        memset(oh, 0, DS4_N_HEAD_DIM * sizeof(float));
        for (uint32_t r = 0; r < n_kv; r++) {
            axpy_f32(scores[r] * inv_sum,
                     kv_rows + r * DS4_N_HEAD_DIM, oh, DS4_N_HEAD_DIM);
        }
    }
}
```

```
Sink-Aware Softmax:

标准:  softmax(scores) = exp(score) / Σ exp(score)
Sink:  分母 = exp(sink) + Σ exp(score)
       分子 = 仅 Σ score[r] × value[r]
       （sink 不对应任何 value 向量）

效果: sink 作为"虚拟 token"吸收多余的注意力权重，
      防止首个真实 token 被过度关注（StreamingLLM 的核心思想）
```

追踪 5：KV 缓存管理与混合注意力

KV 缓存写入

```c
// `kv_cache_push_raw()` 中（约 6307 行）: 原始 SWA（滑动窗口注意力）缓存
static void kv_cache_push_raw(ds4_layer_cache *cache, const float *kv) {
    if (cache->n_raw < cache->raw_cap) {
        // 未满: 直接追加，F16 精度舍入
        float *dst = cache->raw + (uint64_t)cache->n_raw * DS4_N_HEAD_DIM;
        for (int i = 0; i < DS4_N_HEAD_DIM; i++)
            dst[i] = f16_to_f32(f32_to_f16(kv[i]));  // float → f16 → float
    } else {
        // 已满: 滑动窗口，最老的行被丢弃
        memmove(cache->raw, cache->raw + DS4_N_HEAD_DIM,
                (cache->n_raw - 1) * DS4_N_HEAD_DIM * sizeof(float));
        // 新行写到末尾
    }
    cache->n_raw++;
}

// `kv_cache_push_comp()` 中（约 6322 行）: 压缩缓存（追加式，无滑动窗口）
static void kv_cache_push_comp(float *rows, uint32_t *n_rows,
                                uint32_t cap_rows, uint32_t row_dim,
                                const float *kv) {
    // 超过容量直接报错（压缩缓存不支持丢弃）
    rows[(*n_rows)++] = kv[...];  // 同样 F16 舍入
}
```

混合注意力

```c
// `layer_attention_mixed_one()` 中（约 6613 行）：同时在 raw 和 comp 缓存上做注意力
static void layer_attention_mixed_one(
        float *out_heads, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *q,
        const float *raw_kv, uint32_t n_raw,
        const float *comp_kv, uint32_t n_comp,
        const bool *comp_allowed) {       // 哪些 comp 行可参与
    const uint32_t n_total = n_raw + n_comp;

    // 先算 raw 行的分数
    for (uint32_t r = 0; r < n_raw; r++) { ... }

    // 再算 comp 行的分数（跳过 comp_allowed[r]==false 的行）
    for (uint32_t r = 0; r < n_comp; r++) {
        if (!comp_allowed[r]) continue;   // ratio-4 层的间歇性压缩
        ...
    }
}
```

```
混合注意力内存布局:

┌──────────────────────── Raw SWA 缓存 ────────────────────────┐
│ [tok_0] [tok_1] ... [tok_W-1]  ← 滑动窗口，最多 W 行      │
└──────────────────────────────────────────────────────────────┘
                          +
┌──────────────────────── Compressed 缓存 ─────────────────────┐
│ [comp_0] [comp_1] ... [comp_N]  ← 累积压缩，最多 N 行      │
└──────────────────────────────────────────────────────────────┘

总注意力行数 = n_raw + n_comp
Raw 保留最近 W 个 token 的精确 KV
Comp 存储更早的 token 的压缩表示
两者在同一个 softmax 中统一计算
```

---

4. MoE 混合专家

追踪 1：SiLU / Softplus / SwiGLU 激活

三种激活函数

```c
// 行 5012: SiLU (Sigmoid Linear Unit) = x × σ(x)
static float silu(float x) {
    return x * sigmoid_stable(x);
}

// 行 5016: 数值稳定的 Softplus = log(1 + exp(x))
static float softplus_stable(float x) {
    if (x > 20.0f)  return x;           // 大正数: log(1+exp(x)) ≈ x
    if (x < -20.0f) return expf(x);     // 大负数: log(1+exp(x)) ≈ exp(x)
    return log1pf(expf(x));             // 中间区间: 精确计算
}

// 行 5022: SwiGLU = SiLU(gate) × up
static void swiglu(float *out, const float *gate,
                   const float *up, uint64_t n) {
    for (uint64_t i = 0; i < n; i++) {
        out[i] = silu(gate[i]) * up[i];
    }
}
```

```
SiLU 图形:
          |        /
    3     |      /
    2     |    /
    1     |  /
    0 ----+--------
   -1     |  \      ← x < 0 时有轻微负输出（"自门控"特性）
          |
          +---------------→ x

SwiGLU 数据流:
gate[i] → SiLU → × up[i] → out[i]
                 ↑
         门控信号    信息流
```

追踪 2：共享 FFN（密集前馈）

单 token 共享专家

```c
// 行 5029: 共享专家的完整前馈
static void layer_shared_ffn_one(
        float *out, const ds4_model *model,
        const ds4_layer_weights *layer, const float *x) {

    // gate 和 up 投影同时计算
    float *gate = xmalloc(DS4_N_FF_EXP * sizeof(float));
    float *up   = xmalloc(DS4_N_FF_EXP * sizeof(float));
    float *mid  = xmalloc(DS4_N_FF_EXP * sizeof(float));

    // 激活量化 + pair matvec（gate 和 up 共享量化激活）
    quantize_q8_0_activation(x, xq, scale, in_dim);
    matvec_q8_0_pair_prequant(gate, up, model,
                              layer->ffn_gate_shexp,
                              layer->ffn_up_shexp, xq, scale);

    // SwiGLU 激活
    swiglu(mid, gate, up, DS4_N_FF_EXP);

    // Down 投影
    matvec_q8_0(out, model, layer->ffn_down_shexp, mid);

    free(gate); free(up); free(mid);
}
```

Decode 热路径（零分配）

```c
// 行 5063: 使用预分配暂存区的版本
static void layer_shared_ffn_one_decode_scratch(
        float *out, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *x, ds4_cpu_decode_scratch *scratch) {
    // 所有中间缓冲区从 scratch 中分配，无 malloc
}
```

批量并行 SwiGLU

```c
// 行 5094: 批量 worker
static void swiglu_batch_worker(void *vctx, uint64_t t0, uint64_t t1) {
    for (uint64_t t = t0; t < t1; t++)
        swiglu(mid + t * DS4_N_FF_EXP,
               gate + t * DS4_N_FF_EXP,
               up + t * DS4_N_FF_EXP, DS4_N_FF_EXP);
}
```

追踪 3：Hash 路由与 Top-K 选择

Hash 查表路由

```
// 行 5148: 直接查表获取被选专家
static void layer_hash_selected_experts(
        int selected[DS4_N_EXPERT_USED],   // 输出: 6 个专家 ID
        const ds4_model *model,
        const ds4_layer_weights *layer,
        int token) {                        // 当前 token ID
    // ffn_gate_tid2eid 是预计算的 [DS4_N_EXPERT_USED, n_vocab] 查找表
    // selected[i] = table[i * n_vocab + token]
    // 无需实时计算路由，O(1) 查表
}
```

路由概率计算

```
// 行 5169: 基于路由器的概率
static void layer_router_probs_one(
        float probs[DS4_N_EXPERT],          // 256 个专家的概率
        const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *x) {
    float logits[DS4_N_EXPERT];
    matvec_q8_0(logits, model, layer->ffn_gate_inp, x);
    for (int i = 0; i < DS4_N_EXPERT; i++)
        probs[i] = sqrtf(softplus_stable(logits[i]));
    //               ^^^^^^^             ^^^^^^^^^^^^
    //               开方使分布更尖锐        softplus 保证非负
}
```

插入排序 Top-K

```c
// 行 5211: O(n×k) 的插入排序 top-k
static void topk_desc(const float *score, int n, int k, int *idx) {
    for (int i = 0; i < k; i++) idx[i] = -1;         // 初始化
    for (int i = 0; i < n; i++) {                     // 扫描 n=256 个
        for (int j = 0; j < k; j++) {                 // 与当前 top-k 比较
            if (idx[j] < 0 || score[i] > score[idx[j]]) {
                for (int m = k - 1; m > j; m--)          // 后移
                    idx[m] = idx[m - 1];
                idx[j] = i;                              // 插入
                break;
            }
        }
    }
}
```

```
Top-K 选择过程 (n=256, k=6):

score[0] = 0.3  → idx = [0, -1, -1, -1, -1, -1]
score[1] = 0.8  → idx = [1, 0, -1, -1, -1, -1]
score[2] = 0.1  → idx 不变（太小）
score[3] = 0.5  → idx = [1, 3, 0, -1, -1, -1]
...
score[255] = 0.9 → idx = [255, 1, 3, 7, 42, 0]

最终选中 6 个专家: [255, 1, 3, 7, 42, 0]
```

追踪 4：路由 MoE 完整流水线

单 token MoE

```c
// 行 5278: 路由 MoE 的核心
static void layer_routed_moe_one(
        float *out, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *x, uint32_t il, int token,
        float clamp, bool trace) {

    // 1. 路由选择
    if (layer->ffn_gate_tid2eid) {
        layer_hash_selected_experts(selected, model, layer, token);  // Hash 查表
        layer_hash_router_weights_one(weight, model, layer, x);      // 计算权重
    } else {
        layer_topk_selected_experts(selected, weight, model, layer, x);  // Top-K 选择
    }

    // 2. 快速路径: 所有专家的 gate+up 一起计算
    matvec_iq2_xxs_experts_mid_prequant(
        gate_all, up_all, model, layer,
        selected, weight, xq, clamp);

    // 3. SwiGLU 激活（逐专家）
    for (int e = 0; e < DS4_N_EXPERT_USED; e++)
        swiglu(mid + e * DS4_N_FF_EXP,
               gate + e * DS4_N_FF_EXP,
               up + e * DS4_N_FF_EXP, DS4_N_FF_EXP);

    // 4. 量化 mid 并做 down 投影
    ds4_quantize_row_q8_K(mid, midq, ...);
    matvec_q2_k_experts_accum_prequant(
        out, model, layer, selected, weight, midq);
}
```

MoE 数据流

```
输入 x[in_dim]
    │
    ├─→ 路由选择 → selected[6], weight[6]
    │
    ├─→ Q8_K 量化 x → xq
    │
    ├── 专家 0: gate[0],up[0] = IQ2_XXS_matvec(w_gate[0],w_up[0],xq)
    ├── 专家 1: gate[1],up[1] = IQ2_XXS_matvec(w_gate[1],w_up[1],xq)
    ├── ...
    ├── 专家 5: gate[5],up[5] = IQ2_XXS_matvec(w_gate[5],w_up[5],xq)
    │
    ├── mid[0] = SwiGLU(gate[0], up[0])
    ├── mid[1] = SwiGLU(gate[1], up[1])
    ├── ...
    │
    ├── Q8_K 量化 mid → midq
    │
    ├── out += weight[0] × Q2_K_matvec(w_down[0], midq[0])
    ├── out += weight[1] × Q2_K_matvec(w_down[1], midq[1])
    ├── ...
    └── out += weight[5] × Q2_K_matvec(w_down[5], midq[5])
```

追踪 5：FFN 层完整前向

单 token FFN（shared + routed）

```c
// 行 5576: 完整的 FFN 层
static void layer_ffn_one(
        float *out_hc, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *inp_hc, uint32_t il, int token,
        const float *steering_dirs, float steering_scale,
        bool trace) {

    // 1. HC 前置（提取 pre/post/comb 控制向量）
    hc_pre_from_state_one(model, fn, scale, base, inp_hc,
                          pre, post, comb);

    // 2. 注意力层 RMSNorm 后的状态
    rms_norm_weight(norm, inp_hc, layer->ffn_norm, DS4_N_EMBD, DS4_RMS_EPS);

    // 3. 路由 MoE
    layer_routed_moe_one(moe, model, layer, norm, il, token, clamp, trace);

    // 4. 共享专家
    layer_shared_ffn_one(shared, model, layer, norm);

    // 5. 合并
    for (int i = 0; i < DS4_N_EMBD; i++)
        ffn_out[i] = moe[i] + shared[i];

    // 6. 可选 steering 投影
    if (steering_dirs) { ... }

    // 7. HC 后置（将输出写回 HC 状态）
    hc_post_one(out_hc, ffn_out, inp_hc, post, comb, n_embd, n_hc);
}
```

FFN 层全貌

```
inp_hc[n_hc × n_embd]            ← HC 残差状态
    │
    ├─→ HC pre → pre, post, comb  ← 控制向量
    │
    ├─→ RMSNorm → norm            ← 归一化
    │
    ├──→ 路由 MoE(norm) → moe     ← 6/256 个专家
    │
    ├──→ 共享 FFN(norm) → shared  ← 1 个共享专家
    │
    ├─→ moe + shared → ffn_out    ← 合并
    │
    ├─→ [steering projection]     ← 可选方向控制
    │
    └─→ HC post → out_hc          ← 写回 HC 状态

DeepSeek V4 FFN 设计: 每层 256 个路由专家 + 1 个共享专家
                       每个 token 只激活 6 个路由专家
                       稀疏度 = 1 - 6/256 ≈ 97.7%
```

Part 4: 推理流程

> 从 token 输入到 token 输出：RoPE 位置编码、prefill/decode 两阶段、线程池并行。本主题追踪一次完整的推理循环。

涵盖内容

章节	核心主题
1. rope 与推理循环	旋转位置编码、yarn 扩展、prefill vs decode、自回归生成
2. 线程池	pthread、mutex/condvar、TLS 防嵌套并行

核心概念

- rope — 旋转位置编码
- yarn — YaRN 上下文扩展
- kv-cache — prefill 填充、decode 追加
- rmsnorm — 每层前向中的归一化

前置知识

- Part 3: 模型架构（Attention、MoE、量化）

学习路径

读完本主题后，你将理解：

1. RoPE 如何在 Q/K 向量中编码位置信息
2. prefill（批量消化上下文）和 decode（逐 token 生成）的区别

3. 线程池如何并行化 43 层的计算

→ 下一步：Part 5: 服务层

Part 4: 推理流程

> RoPE、prefill/decode、线程池

---

1. RoPE 与推理循环

> "我爱猫"和"猫爱我"的注意力分数一样——模型分不清词序。RoPE 通过旋转向量编码位置信息。今天还学习 prefill vs decode 两个推理阶段，以及自回归生成的本质：每步输出成为下步输入，无法并行。

RoPE 在 Self-Attention 的 Q/K 投影中注入位置信息，使模型感知 token 顺序。

设计决策推导：位置编码与推理阶段

```
问题 1：注意力分数只看向量内容，不知道 token 顺序
     → "我爱猫"和"猫爱我"的 Q/K 值完全相同，模型分不清
  └─ 方案：RoPE — 按 token 位置旋转 Q 和 K
     2D 旋转矩阵有性质 R(a)·R(b) = R(b-a)
     → Q·K 点积只依赖相对距离，不依赖绝对位置

问题 2：训练时只见过 64K 长度，推理时要处理 1M token？
  └─ 方案：YaRN — 混合外推和内插
     高频分量（精确区分相邻位置）→ 内插（缩小频率）
     低频分量（感知远距离关系）→ 外推（直接延伸）
     → 支持 16× 上下文扩展（64K → 1M）

问题 3：用户发来 5000 token 的 prompt，怎么处理？
  └─ Prefill 阶段 — 所有 prompt token 批量通过 43 层
     → 可以并行（矩阵乘法天然并行）
     → GPU 利用率高，速度 250+ t/s

问题 4：生成回复时，每步只产生 1 个 token，GPU 利用率极低？
  └─ Decode 阶段 — 逐 token 串行生成（自回归）
     每步要读取 81GB 权重但只算 1 个 token → 带宽瓶颈
     → 速度只有 5-10 t/s
     → 这就是 decode 比 prefill 慢 25-50 倍的根本原因

问题 5：decode 能不能加速？
  └─ 方案：MTP 推测解码 — 用小模型猜 N 个 token，
     大模型一次验证，猜对就"免费"获得 N 个 token
     → 接受率 70% 时理论加速 2×+
```

C 知识点

1. 三角函数

RoPE 用 cos/sin 实现旋转变换：

```
const float c = cosf(theta) * mscale;
const float s = sin_sign * sinf(theta) * mscale;
const float x0 = tail[i + 0];
const float x1 = tail[i + 1];

tail[i + 0] = x0 * c - x1 * s;    // 旋转后的第一个分量
tail[i + 1] = x0 * s + x1 * c;    // 旋转后的第二个分量
```

这是 2D 旋转矩阵的数值实现：

```
[x0']   [cos θ  -sin θ] [x0]
[x1'] = [sin θ   cos θ] [x1]
```

2. 幂运算与频率

```
const float theta_scale = powf(freq_base, -2.0f / (float)n_rot);
// ...
theta_extrap *= theta_scale;   // 每对维度乘以一个衰减因子
```

RoPE 的频率按指数衰减：第 i 对维度使用 `freq_base^(-2i/n_rot)` 作为频率。低维度变化快（高频），高维度变化慢（低频）。

3. 推理循环的状态管理

生成循环维护一个位置计数器 `pos`：

```
for (int i = 0; i < n_predict && pos < ctx_size; i++) {
    int token = sample_argmax(logits, DS4_N_VOCAB);   // 选 token
    if (token == vocab->eos_id) break;                 // 遇到 EOS 停止

    if (emit) emit(emit_ud, token);                    // 回调输出
    n_generated++;

    forward_token(logits, model, weights, &cache,      // 前向传播一个 token
                  token, (uint32_t)pos, &scratch);
    pos++;                                               // 位置 +1
}
```

LLM 知识点

1. RoPE（Rotary Position Embedding）

RoPE 通过旋转向量来编码位置信息：

```
位置 0: 向量旋转 0°
位置 1: 向量旋转 θ°
位置 2: 向量旋转 2θ°
位置 n: 向量旋转 nθ°
```

关键性质：两个位置的内积只依赖于它们的相对距离：

```
Q_pos_m · K_pos_n = f(m - n)
```

这使得注意力自然地编码相对位置——不需要额外的位置参数。

2. YaRN 扩展

RoPE 在超出训练长度后效果下降。YaRN（Yet another RoPE extensioN）通过插值解决：

```
// 在训练长度内：正常 RoPE
// 超出训练长度：混合外推和内插
const float ramp_mix = rope_yarn_ramp(corr_dims[0], corr_dims[1], i) * ext_facto
r;
theta = theta_interp * (1.0f - ramp_mix) + theta_extrap * ramp_mix;
```

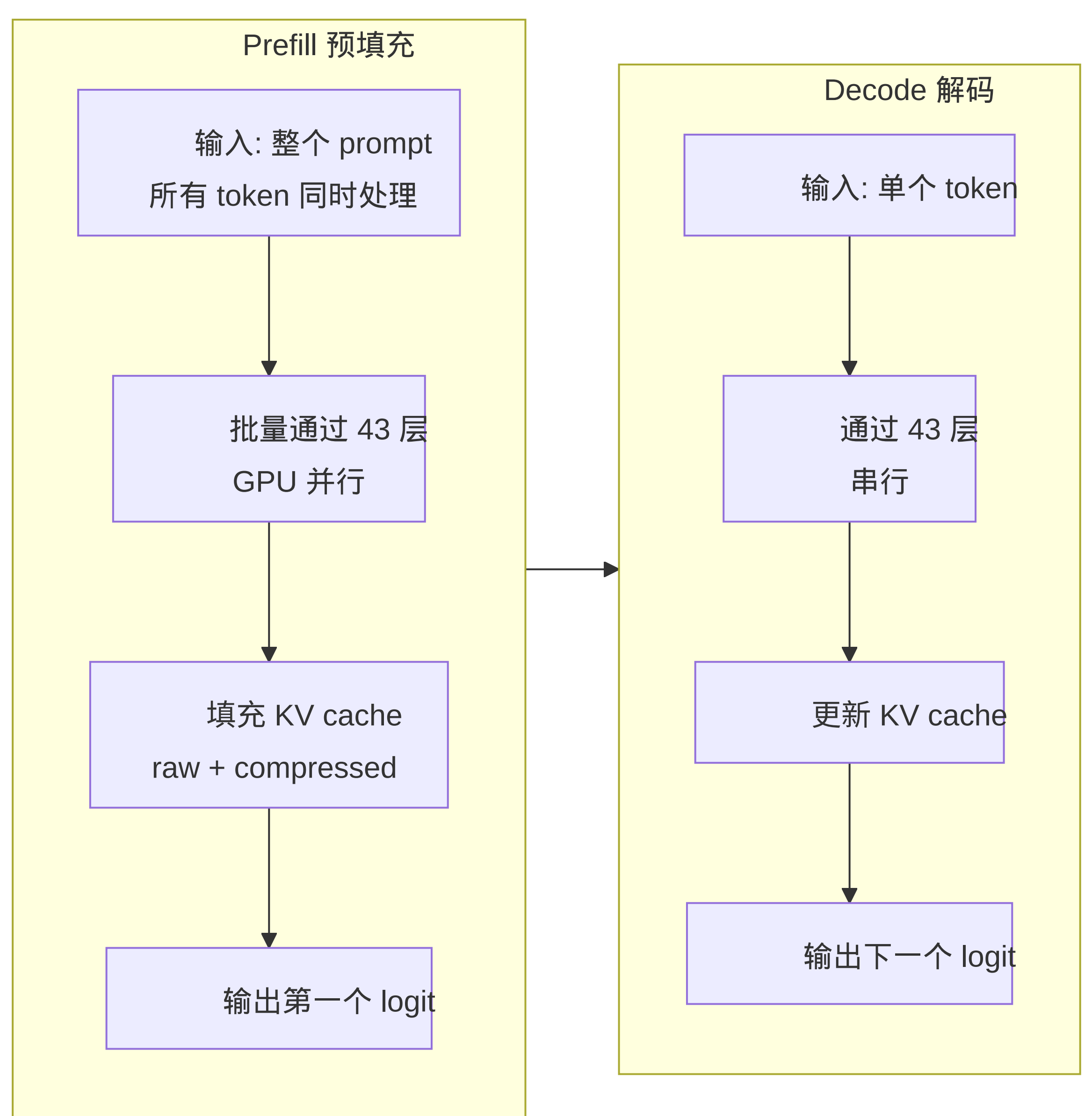
低维度（高频）：保持外推（因为它们已经学到了长距离关系） 高维度（低频）：改为内插（缩放到更小的角度）

ds4.c 的配置：

- `DS4_ROPE_FREQ_BASE = 10000.0`（标准 base）
- `DS4_ROPE_SCALE_FACTOR = 16.0`（扩展 16 倍）
- `DS4_ROPE_ORIG_CTX = 65536`（训练时上下文长度）
- 扩展后支持 1M token 上下文

3. Prefill vs Decode

推理分两个阶段：



Prefill（预填充）： 处理整个 prompt

```
输入: [token_0, token_1, ..., token_n]  (整个 prompt)
处理: 所有 token 同时通过所有层
输出: 最后一个 token 的 logits
复杂度: O(n × L × d²)  (n = prompt 长度)
```

ds4.c 用层主序（layer-major）处理：

```
// 先把所有 token 通过第 0 层，再通过第 1 层...
for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
    // 批量注意力：所有 token 互相关注
    layer_attention_raw_swa_batch(attn, ..., n_tok, il, 0);
    // 批量 FFN：可以分组处理
    layer_ffn_batch(next, ..., n_tok, il);
}
```

Decode（解码）： 逐个生成 token

```
输入: token_t  (一个新 token)
处理: 通过所有层，用 KV cache 做注意力
输出: 下一个 token 的 logits
复杂度: O(L × d²) (常量，与历史长度无关... 不完全对，因为有压缩 KV)
```

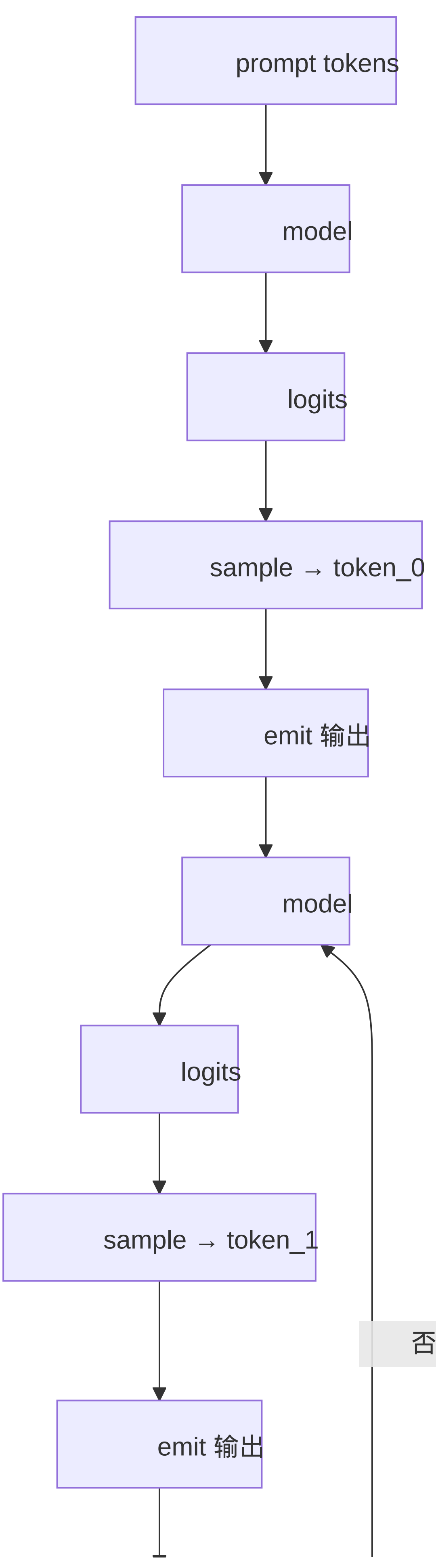
```
// 每个 token 独立通过所有层
for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
    layer_forward_raw_swa_one(next, model, &weights->layer[il],
                              &cache->layer[il], cur, il, pos, token, scratch);
    swap(cur, next);
}
```

性能对比：

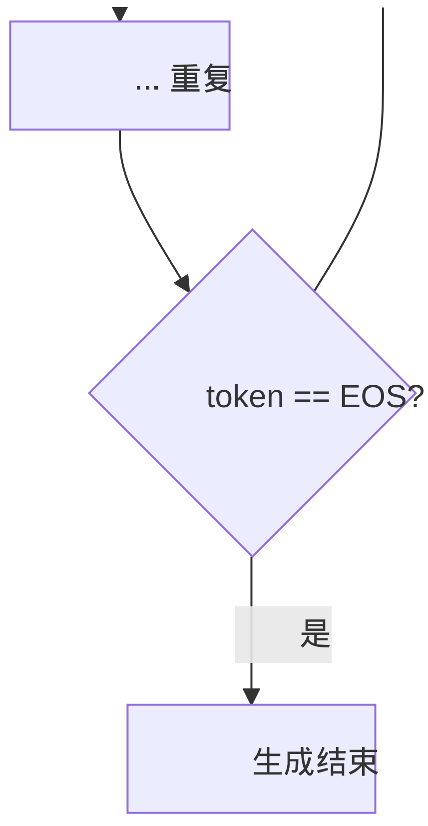
- Prefill: 快速，因为可以批量处理（GPU 并行）
- Decode: 较慢，因为每次只处理一个 token（串行）

4. 自回归生成

LLM 的生成是自回归的——每一步的输出成为下一步的输入：



prompt tokens
model
logits
sample → token_0
emit 输出
model
logits
sample → token_1
emit 输出
否



这个过程本质上是串行的——无法并行生成多个 token（因为每一步依赖上一步的结果）。这就是为什么 decode 速度（t/s）是 LLM 推理的关键指标。

5. 推测解码（Speculative Decoding / MTP）

ds4.c 支持 MTP（Multi-Token Prediction）推测解码来加速：

```
1. 主模型生成 token A
2. MTP 小模型快速推测: A → B → C → ...（多步草稿）
3. 主模型验证: 确认哪些草稿 token 正确
4. 接受正确的前缀，丢弃错误的
5. 重复

理想情况: 一次验证接受多个 token → 加速
实际情况: 接受率取决于 MTP 模型质量
```

ds4.c 的实现（行 16071-16320）：

- 草稿深度由 `--mtp-draft N` 控制；深度 2 时走专用的 argmax 快速路径
- 有置信度门槛（`mtp_margin`）：低置信度时跳过 MTP
- Metal 路径上用 `decode2_exact` 等内核一次验证多个位置

推测验证的"逐行 argmax"不变量

推测解码的验证步必须保证一个不变量：每个被提交（**commit**）的草稿 **token**，必须是其位置上主模型 **logits** 的（近似）**argmax**——否则就把模型从没生成过的 token 当成它生成的，污染上下文。

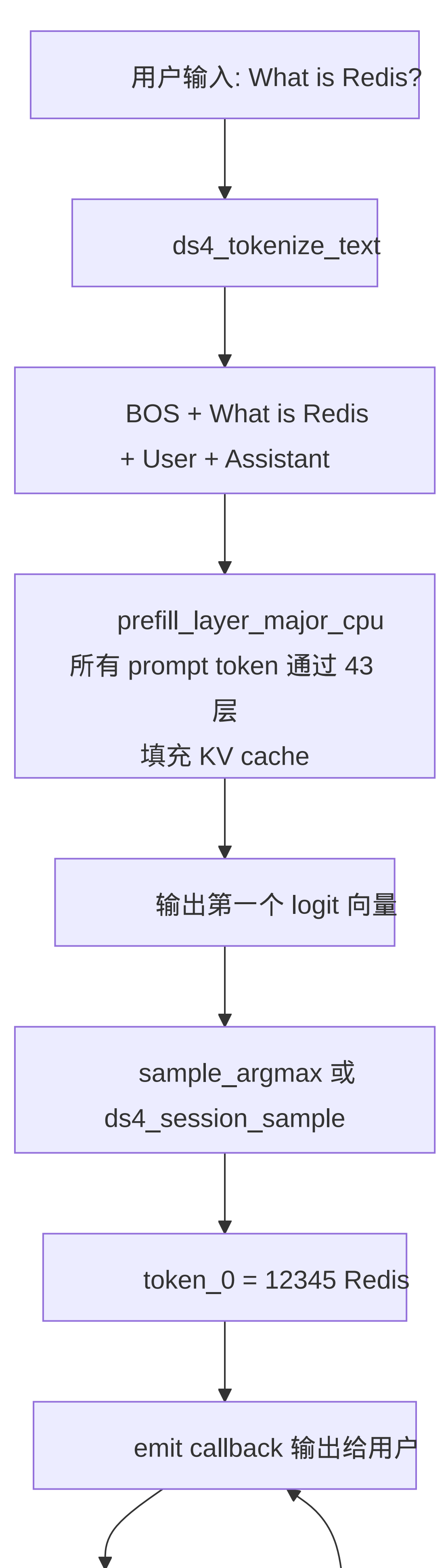
这条不变量一度在 `--mtp-draft > 2` 时被破坏：`metal_graph_verify_suffix_tops()` 本该对 `top_rows` 个 logits 行各取 top-1，却误调成了"对单行取 top-`top_rows`"（`n_tokens=1, top_k=top_rows` 而非 `n_tokens=top_rows, top_k=1`）。于是 `row_tops[i>0]` 拿到的是第 0 行的亚军 token，验证器拿错误的行去匹配草稿，接受了模型根本没生成的 token。

为什么深度 ≤2 看不到这个 bug？因为 `top_rows == 1` 时代码走的是上方的专用 argmax 分支，根本到不了那段错误调用。所以这是一个"只在开深推测时才暴露"的隐藏缺陷：

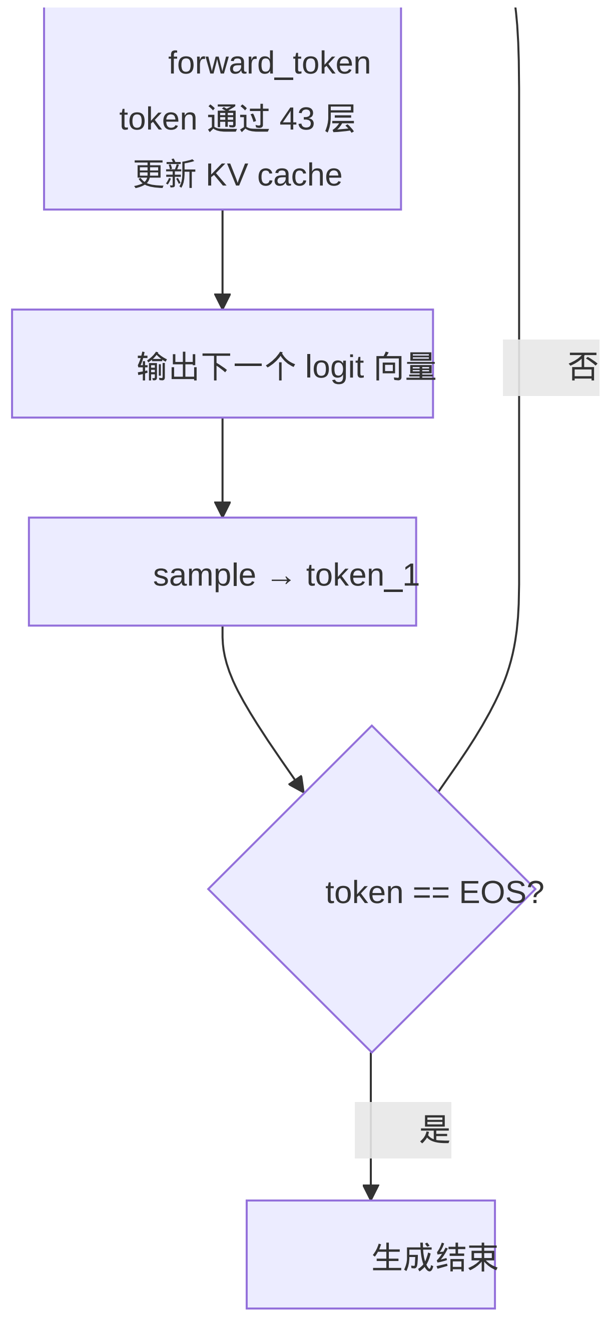
```
// ds4_gpu_indexer_topk_tensor() 的签名是 (selected, scores, n_comp, n_tokens, to
p_k)
//  错误: (..., n_tokens=1,         top_k=top_rows)  // 单行的前 top_rows 名
//  正确: (..., n_tokens=top_rows, top_k=1)         // 每行的 top-1
```

修复后每个被提交 token 的 argmax 间距（worst gap）从 ~20.96 logits 降到 0.00。该调用点在后端共享的 `ds4.c` 里，所以 CUDA 同样受影响、同样修复。回归测试 `tests/ds4_test.c --mtp-verify-depth`：在深度 4 跑贪心推测解码一个逐字复制任务，再把提交的 token 教师强制喂回普通解码，断言每个都是其位置的近似 argmax——这恰是推测验证必须保持的不变量。

6. 完整推理流程图



用户输入: What is Redis?
ds4_tokenize_text
BOS + What is Redis
+ User + Assistant
prefill_layer_major_cpu
所有 prompt token 通过 43
层
填充 KV cache
输出第一个 logit 向量
sample_argmax 或
ds4_session_sample
token_0 = 12345 Redis
emit callback 输出给用户



---

2. 线程池

> 43 层的矩阵运算、MoE 的 256 个专家——单线程太慢。线程池把行范围分配给多个 CPU 核心并行计算，推理速度随核心数线性增长。今天学习 pthread API、worker 主循环和防嵌套并行的 TLS 技巧。

线程池为 HTTP 服务器 提供并发请求处理能力。

C 知识点

1. pthread 基础

POSIX 线程（pthread）是 C 语言的标准线程 API：

```
pthread_t thread;
pthread_create(&thread, NULL, worker_fn, arg);   // 创建线程
pthread_join(thread, NULL);                       // 等待结束
pthread_mutex_t mu = PTHREAD_MUTEX_INITIALIZER;  // 互斥锁
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;    // 条件变量
```

2. Mutex（互斥锁）

保护共享数据的访问：

```
pthread_mutex_lock(&g_pool.mutex);     // 加锁
// ... 访问共享数据 ...
pthread_mutex_unlock(&g_pool.mutex);   // 解锁
```

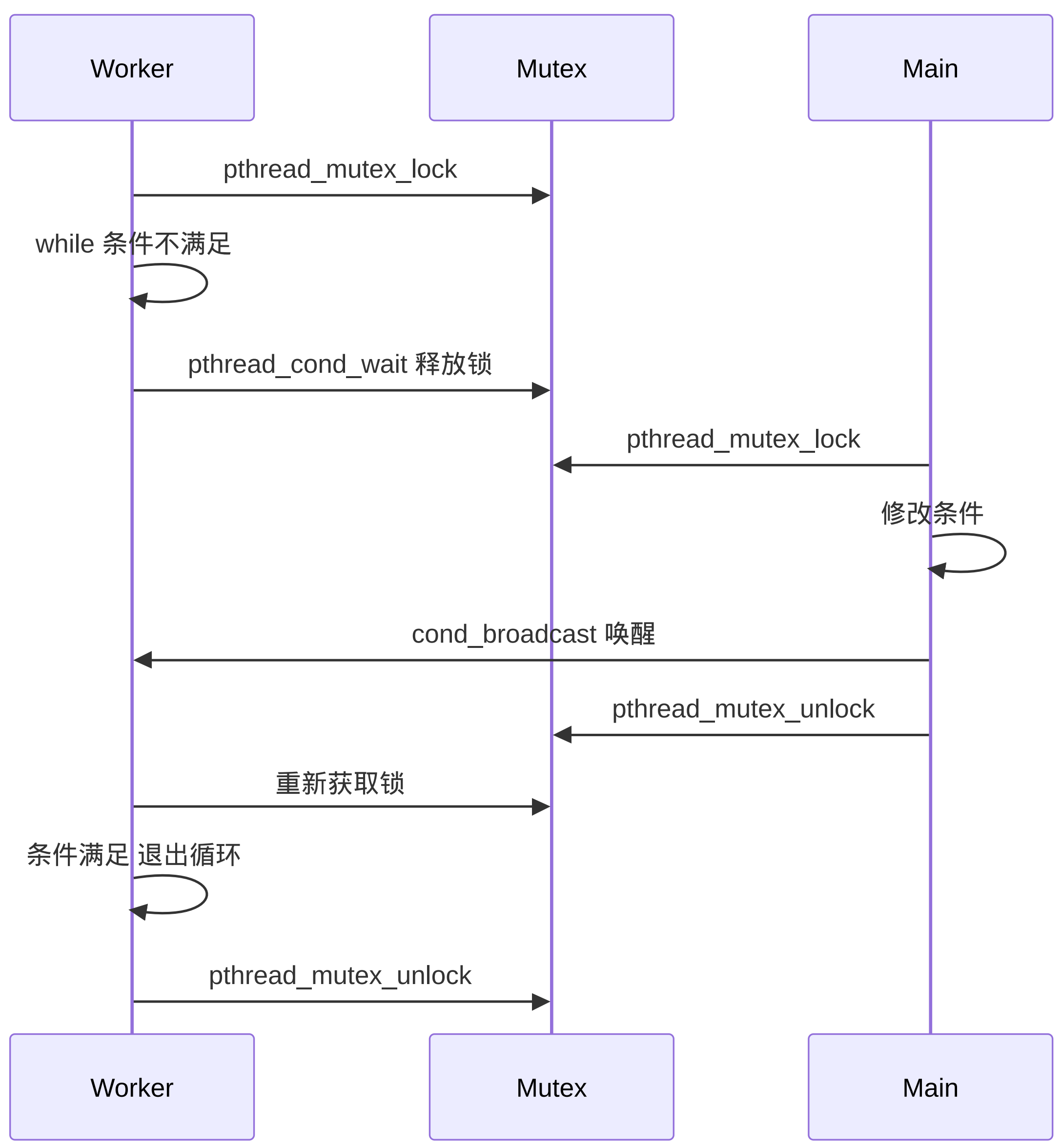
> 同一时间只有一个线程可以持有锁。其他线程在 `lock` 处阻塞等待。

3. Condition Variable（条件变量）

用于线程间通知：

```
// 等待方（worker）：
pthread_mutex_lock(&mu);
while (条件不满足)
    pthread_cond_wait(&cv, &mu);    // 释放锁 + 等待信号 + 重新获取锁
pthread_mutex_unlock(&mu);

// 通知方（主线程）：
pthread_mutex_lock(&mu);
修改条件;
pthread_cond_broadcast(&cv);          // 唤醒所有等待者
pthread_mutex_unlock(&mu);
```

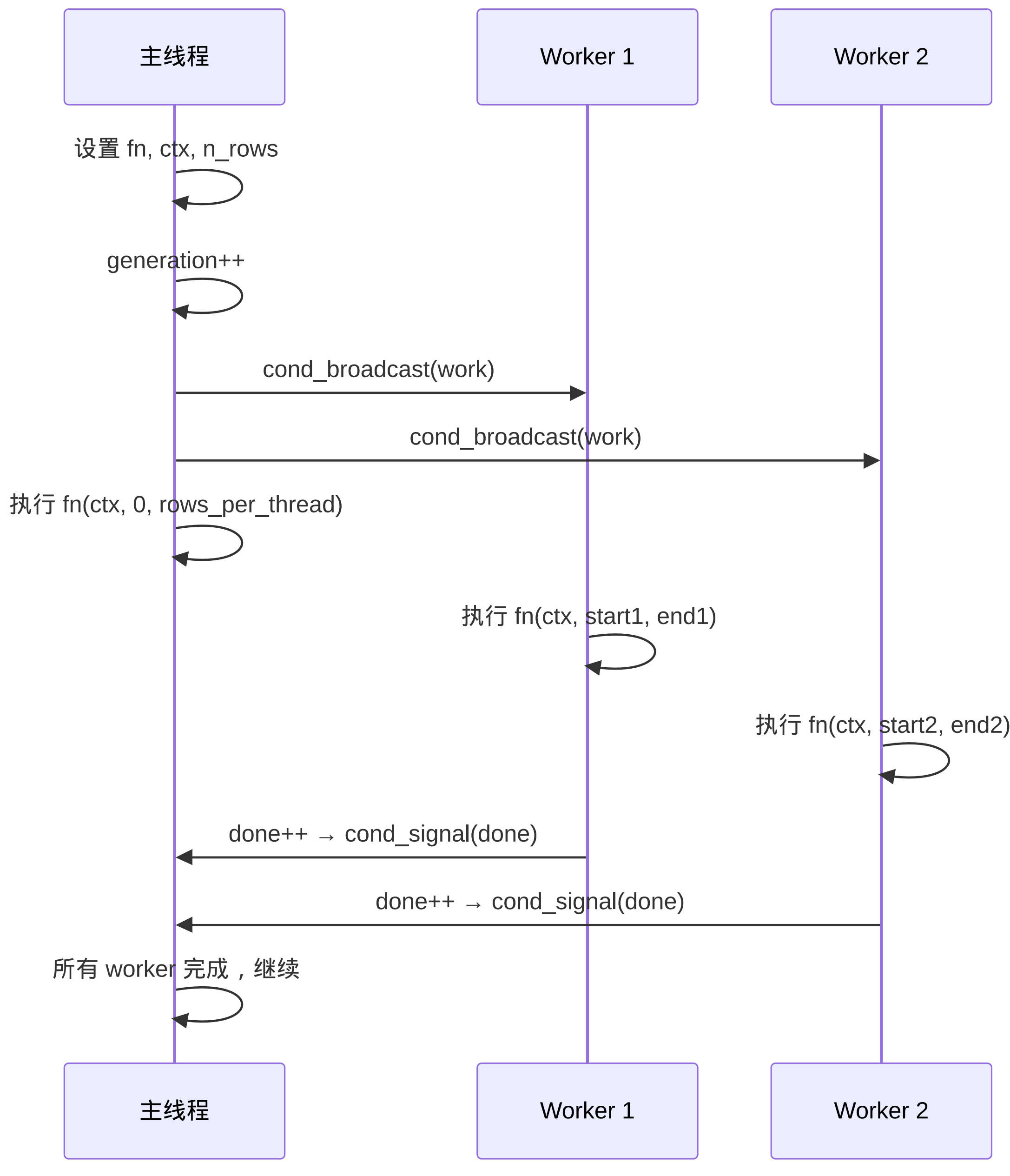
> `wait` 必须在 `while` 循环中（不是 `if`），因为可能被虚假唤醒。



4. 线程池设计

ds4.c 的线程池（行 594-752）：

```
主线程                              Worker 线程 1..N-1
   │                                    │
   ├─ 设置 fn, ctx, n_rows              │ (阻塞在 cond_wait)
   ├─ generation++                      │
   ├─ cond_broadcast(work) ─────────────┼→ 被唤醒
   ├─ 自己也执行一部分工作              ├─ 检查 generation
   │  fn(ctx, 0, rows_per_thread)       ├─ 执行 fn(ctx, my_start, my_end)
   │                                    │
   ├─ cond_wait(done) ←─────────────────┼─ done++ → cond_signal(done)
   │                                    │
   └─ 所有 worker 完成，继续            └─ 回到 cond_wait
```



关键设计：

- 主线程也工作：不浪费一个核
- **generation** 计数器：防止旧的 broadcast 唤醒新的任务
- 行范围分配：`rows_per_thread = (n_rows + n_threads - 1) / n_threads`
- **`__thread g_parallel_depth`**：防止嵌套并行（递归时退化为串行）

5. TLS（Thread-Local Storage）

```
static __thread int g_parallel_depth;
```

每个线程有自己的副本。在 `ds4_parallel_for` 中：

```
g_parallel_depth++;      // 标记"当前线程已在并行执行"
fn(ctx, 0, main_row1);  // 如果 fn 内部又调用 parallel_for
g_parallel_depth--;      // → g_parallel_depth > 0 → 退化为串行
```

LLM 知识点

推理并行性

ds4.c 在 prefill 阶段利用多线程加速：

```
// 并行处理多个 token 的 SwiGLU
ds4_parallel_for(n_tok, swiglu_batch_worker, &ctx);

// 并行处理多个 token 的 FFN
ds4_parallel_for(n_tok, ffn_worker, &ctx);
```

但 decode 阶段无法并行（每次只处理 1 个 token）。多线程只在 prefill 的批量操作中有效。线程池的最小行数阈值（512）确保小任务不会因线程同步开销而变慢。

困惑度评分（**Perplexity Scoring**）

`--perplexity-file` 模式让 ds4 评估一段文本与模型预期的吻合程度：

1. 将文件内容分词后送入模型
2. 跳过前 32 个 token 作为"种子"（避免 BOS 附近的异常值）
3. 对每个后续 token 累加负对数似然（NLL）
4. 最终报告平均 NLL 和 perplexity（`exp(mean_nll)`）

困惑度越低，模型对这段文本越"不意外"——用于评估量化质量、比较模型变体、或验证模型行为是否正确。CLI 通过 `ds4_engine_options.perplexity_file` 字段传入文件路径。

GPU 功耗节流（**Power Throttling**）

推理引擎支持 `power_percent`（1-100）参数控制 GPU 使用率。低于 100 时，引擎在每个 prefill 层和 decode token 后注入 sleep：

```
实际 sleep 微秒 = (100 - power_percent) / power_percent × step_time
```

`ds4_engine_set_power()` API 允许运行时动态调整（agent 模式下根据用户交互状态自动调节）。设为 100 时不插入任何 sleep，满速运行。

反向链接

/glossary/rmsnorm

/glossary/rope

/glossary/yarn

端到端推理流程

学习日志

Part 5: 服务层

出链

Part 3: 模型架构

Part 5: 服务层

Part 4: 练习

---

1. RoPE 与推理循环

练习 1：RoPE 旋转计算

题目：位置 pos=2, head_dim=4, n_rot=2, freq_base=10000。计算第 0 对维度的旋转角度。

提示：theta = pos × freq_base^(-2×0/n_rot) = pos × freq_base^0 = pos

参考答案

```
第 0 对维度（i=0）:
  theta = 2 × 10000^(−0/2) = 2 × 1 = 2

  cos(2) ≈ -0.416
  sin(2) ≈  0.909

  如果 tail[0] = 1.0, tail[1] = 0.0:
    tail[0]' = 1.0 × (-0.416) - 0.0 × 0.909 = -0.416
    tail[1]' = 1.0 × 0.909 + 0.0 × (-0.416) = 0.909

  向量从 (1, 0) 旋转到 (-0.416, 0.909) ≈ 旋转了 2 弧度 (114.6°)
```

练习 2：Prefill vs Decode 性能

题目：MacBook Pro M3 Max 上，ds4 的 prefill 速度是 250 t/s，decode 速度是 27 t/s。为什么 decode 慢这么多？

参考答案

1. **Prefill** 可以批量：所有 prompt token 同时通过每一层，GPU 可以并行计算多个 token 的矩阵乘法
2. **Decode** 是串行的：每次只处理一个 token，GPU 利用率低
3. 内存带宽瓶颈：decode 时每步需要读取大量权重（81GB 模型），但只计算一个 token。计算/带宽比极低
4. **KV cache** 访问：decode 需要读取所有 KV cache 行做注意力，随着生成进行，KV cache 越来越大

简单说：prefill 是"计算密集"（GPU 擅长），decode 是"带宽密集"（GPU 的瓶颈）。

练习 3：自回归生成模拟

题目：假设 logits 经过 softmax 后的概率分布为：

```
Token  概率
"cat"  0.5
"dog"  0.3
"the"  0.15
"is"   0.05
```

用贪心解码（argmax），模拟生成 "the cat is" 的过程（假设前两步已经确定了 "the cat"）。

参考答案

```
Step 0: 输入 "the"
  logits → softmax → argmax → "cat" (0.5)
  输出: "cat"

Step 1: 输入 "cat"（KV cache 已有 "the"）
  logits → softmax → argmax → "is" (0.5)
  输出: "is"

完整生成: "the" → "cat" → "is"
```

关键点：每一步的输入是上一步的输出，这就是"自回归"的含义。如果某一步的概率分布不同，后续所有步骤都会改变。

练习 4：推测解码效率

题目：假设 MTP 模型的草稿接受率为 70%。每次 MTP 推测 2 个 token。

1. 平均每次验证接受多少 token？
2. 相比不用推测解码，加速多少？（假设验证一次的成本 = 正常 decode 一步）

参考答案

```
每次推测 2 个 token 的验证:

情况1: 前 2 个都正确 (70% × 70% = 49%)
  → 接受 2 个 token + 验证产生 1 个额外 token = 3 个 token
  → 成本: 1 次验证

情况2: 第 1 个正确，第 2 个错误 (70% × 30% = 21%)
  → 接受 1 个 token + 验证产生 1 个额外 token = 2 个 token
  → 成本: 1 次验证

情况3: 第 1 个就错误 (30%)
  → 接受 0 个 token + 验证产生 1 个 token = 1 个 token
  → 成本: 1 次验证

平均每次验证产出:
  0.49 × 3 + 0.21 × 2 + 0.30 × 1 = 1.47 + 0.42 + 0.30 = 2.19 token

不用推测: 每步 1 token
加速比: 2.19 / 1 = 2.19× (理论值)

实际加速更低: MTP 推测本身也有成本，且需要额外的 MTP 模型权重。
```

今日学习检查清单

- ☐ 能解释 RoPE 通过旋转编码位置的原理
- ☐ 理解 YaRN 扩展如何支持超长上下文
- ☐ 能区分 prefill 和 decode 阶段
- ☐ 理解自回归生成的串行本质
- ☐ 能描述完整的推理流程（从输入到输出）
- ☐ 理解推测解码（MTP）的加速原理
- ☐ 理解 decode 为什么比 prefill 慢

延伸挑战

挑战 **1**（中级）：**YaRN** 扩展的可视化

画一张图：x 轴是维度对编号（0 到 64），y 轴是位置（0 到 1M）。用不同颜色标注"外推区域"和"内插区域"。标注训练长度 65536 和 scale factor 16 的关系。

挑战 **2**（高级）：**MTP** 接受率分析

阅读 ds4.c 中 MTP 推测解码的实现（行 16071-16320）。解释 `mtp_margin` 阈值的作用：在什么条件下 MTP 草稿会被丢弃？如果 MTP 草稿的准确率只有 50%，MTP 是加速还是减速？用数学推导。

---

2. 线程池

练习 1：线程池行分配

题目：8 个 token，3 个线程。计算每个线程的行范围。

参考答案

```
rows_per_thread = (8 + 3 - 1) / 3 = 10 / 3 = 3

Thread 0 (主线程): [0, 3)   → token 0, 1, 2
Thread 1:          [3, 6)   → token 3, 4, 5
Thread 2:          [6, 8)   → token 6, 7  (只有 2 个)
```

练习 2：为什么 wait 用 while 不用 if

题目：解释为什么 `pthread_cond_wait` 必须在 `while` 循环中。

参考答案

1. 虚假唤醒：POSIX 允许 `cond_wait` 在没有 `signal/broadcast` 的情况下返回
2. 竞态条件：多个 worker 被唤醒但只有一个能获取任务
3. **generation** 检查：旧的 broadcast 可能在新任务设置前唤醒 worker

```
错误:
if (done < n_workers) cond_wait(...)   // 可能虚假唤醒后继续

正确:
while (done < n_workers) cond_wait(...)  // 每次醒来都重新检查
```

练习 3：并行退化的场景

题目：`g_parallel_depth` 防止嵌套并行。举一个具体场景说明为什么需要这个保护。

参考答案

```
1. 主线程调用 ds4_parallel_for(1000, layer_ffn_batch, ...)
2. 所有线程开始执行 layer_ffn_batch
3. layer_ffn_batch 内部调用 ds4_parallel_for(256, swiglu_worker, ...)
4. 如果不阻止:
   - 每个线程又创建 N 个任务 → N² 个任务
   - 但只有 N 个线程 → 大量等待和同步开销
5. g_parallel_depth > 0 时退化为串行:
   - swiglu_worker 在当前线程内顺序执行
   - 避免嵌套线程同步的开销
```

练习 4：线程交错执行追踪

题目：2 个 worker 线程 + 主线程，generation 从 10 开始。主线程调用 `ds4_parallel_for(7, fn, ctx)` 执行 n_rows=7 的任务。

要求：

1. 计算每个线程的行范围
2. 给定 Worker 0 先完成、Worker 1 后完成的顺序，追踪 `done` 值和 `done_cond` signal 的时机
3. 画出完整的时序图

参考答案

```
rows_per_thread = (7 + 3 - 1) / 3 = 3
Thread 0 (主):  [0, 3)
Thread 1:       [3, 6)
Thread 2:       [6, 7)

完成顺序追踪:
1. Worker 0 完成 → __sync_fetch_and_add(&done, 1) → done=1, 1≠2, 不 signal
2. Worker 1 完成 → __sync_fetch_and_add(&done, 1) → done=2, 2==n_workers, s
ignal done_cond
3. 主线程在 cond_wait 中被唤醒 → 检查 done >= n_workers → 条件满足 → 继续

时序图:
主线程                Worker 0           Worker 1
  │                     │                  │
  ├─ generation=11 ─────┤                  │
  ├─ broadcast ────────→│ ← cond_wait      │ ← cond_wait
  │                     │                  │
  ├─ fn(ctx,0,3)       ├─ fn(ctx,3,6)     ├─ fn(ctx,6,7)
  │                     │                  │
  │                     ├─ done=1          │
  │                     │   (1≠2, 不signal) │
  │                     │                  ├─ done=2
  │ ←──────────────────── signal done_cond ─┤
  │                     │                  │
  ▼                     ▼                  ▼
```

练习 5：generation 竞态分析

题目：假设去掉 generation 机制，改为只用一个 `bool has_work` 标志。给出一个具体的交错序列，展示在没有 generation 的情况下 worker 会如何"偷懒"（拿到旧任务或丢失新任务）。

参考答案

```
场景: 两次快速连续的 parallel_for 调用

步骤:
1. 主线程完成第一次 parallel_for，done=2
2. Worker 0 还没回到 cond_wait (还在循环顶部)
3. 主线程发起第二次 parallel_for:
   - has_work = true
   - done = 0
   - broadcast
4. Worker 0 恰好执行到检查 has_work → true
5. Worker 0 拿到第二次的任务参数
6. Worker 1 从 cond_wait 醒来, 但 has_work 仍为 true
7. Worker 1 也拿到第二次的任务 → 重复执行!

对比: 有 generation 时:
- Worker 0 的 seen_generation = 10 (第一次)
- 第二次 generation 变为 11
- Worker 0 醒来发现 seen_generation ≠ generation → 正确拿新任务
- Worker 1 醒来同样检查 generation → 正确

关键: generation 让每个 worker 能区分"我已经处理过的任务"和"新任务"
```

今日学习检查清单

- ☐ 能解释 mutex 和 condvar 的作用
- ☐ 理解线程池的 worker 主循环
- ☐ 能计算行范围分配
- ☐ 理解 generation 计数器的作用
- ☐ 能解释为什么 cond_wait 必须在 while 中
- ☐ 理解 TLS 防止嵌套并行的机制

延伸挑战

挑战 1（中级）：线程扩展性实验

修改 ds4.c 的线程数（通过环境变量或修改 `n_workers`），分别用 1、2、4、8 线程运行推理。记录 decode 速度（t/s），画出"线程数 vs 速度"曲线。是否线性增长？瓶颈在哪里？

挑战 2（高级）：对比线程池和 **work-stealing**

ds4.c 的线程池用固定行范围分配（`rows_per_thread = ceil(n / n_threads)`）。这可能导致负载不均——如果某些行的计算量更大（如 MoE 专家命中不均）。设计一个简单的 work-stealing 方案：worker 完成自己的任务后如何"偷"其他 worker 的任务？需要哪些同步原语？

Part 4: 代码走读

---

1. RoPE 与推理循环

追踪 1：YaRN 频率缩放

频率修正斜坡函数

```
// 行 4675: 线性衰减斜坡，用于平滑过渡维度区间
static float rope_yarn_ramp(float low, float high, int i0) {
    const float y = ((float)(i0 / 2) - low) / fmaxf(0.001f, high - low);
    return 1.0f - fminf(1.0f, fmaxf(0.0f, y));
    //     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //     当 i0 < 2*low: 返回 1.0（完全应用原始频率）
    //     当 i0 > 2*high: 返回 0.0（完全应用缩放频率）
    //     中间: 线性过渡
}
```

修正维度范围计算

```
// 行 4680: 计算需要频率修正的维度边界
static float rope_yarn_corr_dim(int n_dims, uint64_t n_ctx_orig,
                                 float n_rot, float base) {
    return (float)n_dims * logf((float)n_ctx_orig / (n_rot * 2.0f * (float)M_P
I))
           / (2.0f * logf(base));
}

// 行 4684: 计算 [低频修正起点, 低频修正终点]
static void rope_yarn_corr_dims(int n_dims, uint64_t n_ctx_orig,
                                 float freq_base, float beta_fast,
                                 float beta_slow, float dims[2]) {
    // dims[0] = rope_yarn_corr_dim(..., beta_fast)   ← 快速衰减开始
    // dims[1] = rope_yarn_corr_dim(..., beta_slow)   ← 慢速衰减结束
}
```

```
维度索引 vs 频率修正:

维度:  0   2   4   6   ...  dims[0]  ...  dims[1]  ...  n_rot
频率:  高 ←──────────────────────────────────────→ 低
修正:  无修正          ramp: 1→0              全修正
                          ↑过渡区↑
```

追踪 2：RoPE 尾部旋转应用

逐层频率配置

```
// 行 4745: 压缩层使用不同的 RoPE 频率基数
static float layer_rope_freq_base(uint32_t il) {
    return ds4_layer_compress_ratio(il) != 0 && DS4_COMPRESS_ROPE_FREQ_BASE > 0.
0f
        ? DS4_COMPRESS_ROPE_FREQ_BASE    // 压缩层: 独立频率
        : DS4_ROPE_FREQ_BASE;            // 普通层: 全局频率
}

// 行 4751: 压缩层的频率缩放
static float layer_rope_freq_scale(uint32_t il) {
    if (ds4_layer_compress_ratio(il) == 0 || DS4_ROPE_SCALE_FACTOR <= 0.0f)
        return 1.0f;
    return 1.0f / DS4_ROPE_SCALE_FACTOR;
}
```

旋转应用（带 YaRN 扩展）

```c
// 行 4758: 对 Q/KV 的尾部维度应用 RoPE
static void rope_tail_layer_inplace(
        float *x, uint32_t n_head, uint32_t head_dim,
        uint32_t n_rot, uint32_t pos, uint32_t il, bool inverse) {

    const bool compressed = ds4_layer_compress_ratio(il) != 0;
    const float freq_base = layer_rope_freq_base(il);
    const float freq_scale = layer_rope_freq_scale(il);
    const float ext_factor = compressed && DS4_ROPE_SCALE_FACTOR > 1.0f
                             ? 1.0f : 0.0f;

    // 对每个头的尾部长度应用旋转
    for (uint32_t h = 0; h < n_head; h++) {
        float *tail = x + (uint64_t)h * head_dim;  // 尾部维度
        float theta_scale = powf(freq_base, -2.0f / (float)n_rot);

        for (uint32_t i = 0; i < n_rot; i += 2) {
            float theta = pos * powf(theta_scale, i / 2);  // θ = pos × base^(-2
i/d)
            // YaRN 修正: 根据 ext_factor 和 ramp 混合原始/缩放频率
            const float c = cosf(theta) * mscale;
            const float s = sin_sign * sinf(theta) * mscale;

            // 2D 旋转
            float x0 = tail[i + 0], x1 = tail[i + 1];
            tail[i + 0] = x0 * c - x1 * s;
            tail[i + 1] = x0 * s + x1 * c;
        }
    }
}
```

```
RoPE 旋转矩阵:
[x0']   [cos θ  -sin θ] [x0]
[x1'] = [sin θ   cos θ] [x1]

θ = pos × base^(-2i/d)

pos=0:  θ=0,  旋转 0°  → 不旋转（位置 0 无位置信息）
pos=1:  θ=θ₁, 旋转 θ₁ → 低维度旋转角度大（高频）
pos=N:  θ=θₙ, 旋转 θₙ → 高维度旋转角度小（低频）

每个 token 的位置信息编码在旋转角度中，
不同维度的旋转频率不同 → 自然而然地捕获相对位置关系
```

追踪 3：Prefill 层主序前向

批量层主序

```
// 行 7674: 所有 prompt token 依次通过每一层
static void prefill_layer_major_cpu(
        float *logits, const ds4_model *model,
        const ds4_weights *weights,
        ds4_kv_cache *cache, const token_vec *prompt, ...) {

    const uint32_t n_tok = prompt->len;
    // 分配双缓冲
    float *cur  = xmalloc(n_tok * hc_dim * sizeof(float));
    float *next = xmalloc(n_tok * hc_dim * sizeof(float));
    float *attn = xmalloc(n_tok * hc_dim * sizeof(float));

    // Token 嵌入
    for (uint32_t t = 0; t < n_tok; t++)
        embed_token_f16(cur + t * hc_dim, model, weights, prompt->data[t]);

    // 层主序: 所有 token 通过层 0，再通过层 1，...
    for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
        // 注意力子层（token 间并行）
        for (uint32_t t = 0; t < n_tok; t++) {
            layer_attention_prefill_one(attn + t*hc_dim, model,
                                        layer, cache, cur, il, t, ...);
        }
        // FFN 子层（批量 128 token 一组）
        const uint32_t ffn_batch = 128;
        for (uint32_t b = 0; b < n_tok; b += ffn_batch) {
            uint32_t end = MIN(b + ffn_batch, n_tok);
            layer_ffn_batch(next + b*hc_dim, model, layer,
                            attn + b*hc_dim, prompt->data + b, end - b, ...);
        }
        // 交换缓冲区
        float *tmp = cur; cur = next; next = tmp;
    }
    // 输出 logits
    for (uint32_t t = 0; t < n_tok; t++)
        output_logits_one(logits, model, weights, cur + t * hc_dim);
}
```

```
Prefill 层主序 vs Token 主序:

层主序（ds4 使用）:
  Layer 0: [tok0, tok1, ..., tokN] → 全部处理
  Layer 1: [tok0, tok1, ..., tokN] → 全部处理
  ...
  Layer 42: [tok0, tok1, ..., tokN]

Token 主序:
  tok0: [Layer 0, Layer 1, ..., Layer 42] → 一个 token 走完全程
  tok1: [Layer 0, Layer 1, ..., Layer 42]
  ...

层主序优势:
  - 每层只需加载一次权重（所有 token 共享）
  - FFN 可以批量处理（MoE 按专家分组）
  - 内存访问模式更友好（权重驻留在缓存中）
```

追踪 4：Decode 单 token 生成

完整层前向

```c
// 行 7438: 单 token 的完整 Transformer 层
static void layer_forward_raw_swa_one(
        float *out_hc, const ds4_model *model,
        const ds4_layer_weights *layer,
        ds4_layer_cache *cache,
        const float *inp_hc, uint32_t il,
        uint32_t pos, int token,
        const float *steering_dirs,
        float steering_attn_scale, float steering_ffn_scale,
        ds4_cpu_decode_scratch *scratch) {

    // 注意力子层:
    //   HC pre → RMSNorm → Q/KV 投影 → RoPE → KV 缓存写入
    //   → 注意力计算 → 输出投影 → HC post

    // FFN 子层:
    //   HC pre → RMSNorm → routed MoE + shared FFN → HC post
    //   → 可选 steering
}
```

Decode 循环

```
// 行 7649: 单 token decode 入口
static void forward_token_raw_swa_cpu(
        float *logits, const ds4_model *model,
        const ds4_weights *weights,
        ds4_kv_cache *cache, int token, uint32_t pos) {
    // 初始化暂存区（根据上下文长度估算大小）
    // 逐层调用 layer_forward_raw_swa_one_decode_scratch
    // 最终输出 logits
}
```

```
Decode 循环:

for each token to generate:
    │
    ▼ forward_token_raw_swa_cpu(logits, model, cache, token, pos)
    │
    ├── Layer 0:  attention(1 token, n_kv rows) + FFN(1 token)
    ├── Layer 1:  attention(1 token, n_kv rows) + FFN(1 token)
    ├── ...
    └── Layer 42: attention(1 token, n_kv rows) + FFN(1 token)
    │
    ▼ output_logits_one → logits[129280]
    │
    ▼ sample_top_p_min_p → token_id
    │
    pos++
    继续...
```

追踪 5：输出 Logits 与首 token

首 token 特殊处理

```c
// 行 7836: 首个 token 的层前向（无 KV 缓存历史）
static void layer_forward_self_one(
        float *out_hc, const ds4_model *model,
        const ds4_layer_weights *layer,
        const float *inp_hc, uint32_t il,
        uint32_t pos, int token) {
    // Pipeline: HC pre → RMSNorm → Q/KV → RoPE → FP8 量化
    //   → F16 舍入 → 自注意力（只有自身）→ 逆 RoPE → 输出投影
    //   → HC post → FFN
    //
    // 特殊: 无 KV 缓存写入，无压缩，无 steering
    // 这是诊断模式，用于理解单 token 的行为
}

// 行 7891: 从 BOS token 开始的首 token 生成
static void forward_first_token_cpu(
        float *out_hc, const ds4_model *model,
        const ds4_weights *weights, int token) {

    // 嵌入 token
    float *cur = xmalloc(hc_dim * sizeof(float));
    float *next = xmalloc(hc_dim * sizeof(float));
    embed_token_f16(cur, model, weights, token);
    hc_from_plain_embedding(cur, ...);  // 转换为 HC 状态

    // 逐层前向
    for (uint32_t il = 0; il < DS4_N_LAYER; il++) {
        layer_forward_self_one(next, model, layer, cur, il, 0, token);
        float *tmp = cur; cur = next; next = tmp;  // 双缓冲交换
    }

    memcpy(out_hc, cur, hc_dim * sizeof(float));
    free(cur); free(next);
}
```

```
完整推理流程:

用户输入: "Hello world"
    |
    ▼ 分词 → [BOS, Hello, _world]
    |
    ▼ Prefill (层主序)
    |   所有 3 个 token 依次通过 43 层
    |   每层: 注意力 + FFN + KV 缓存写入
    |
    ▼ 输出 logits → 采样 → token_4 = "!"
    |
    ▼ Decode 循环
    |   token_4 → 43 层 → logits → 采样 → token_5 = " How"
    |   token_5 → 43 层 → logits → 采样 → token_6 = " are"
    |   ...
    |   token_N → EOS → 停止
    |
    ▼ 解码 → "Hello world! How are you?"
```

---

2. 线程池

追踪 1：线程池结构体设计

完整字段解析

```c
// 行 612: 工作函数签名
typedef void (*ds4_parallel_fn)(void *ctx, uint64_t row0, uint64_t row1);

// 行 614: 最大线程数
#define DS4_MAX_THREADS 32

// 行 616-630: 线程池结构体
typedef struct {
    pthread_t threads[DS4_MAX_THREADS];  // 线程句柄数组
    pthread_mutex_t mutex;               // 保护以下共享字段的锁
    pthread_cond_t  work_cond;           // 工作线程等待条件
    pthread_cond_t  done_cond;           // 主线程等待条件

    uint32_t n_threads;                  // 总线程数（含主线程）
    uint32_t n_workers;                  // 工作线程数（= n_threads - 1）
    uint32_t generation;                 // 任务代号（避免虚假唤醒）
    uint32_t done;                       // 已完成的工作线程数

    bool initialized;                    // 是否已初始化
    bool shutdown;                       // 是否正在关闭

    ds4_parallel_fn fn;                  // 当前工作函数
    void *ctx;                           // 工作函数上下文
    uint64_t n_rows;                     // 总行数
} ds4_thread_pool;

// 行 632-634: 全局变量
static ds4_thread_pool g_pool;           // 单例线程池
static __thread int g_parallel_depth;    // 线程局部递归深度
static uint32_t g_requested_threads;     // 用户请求的线程数
```

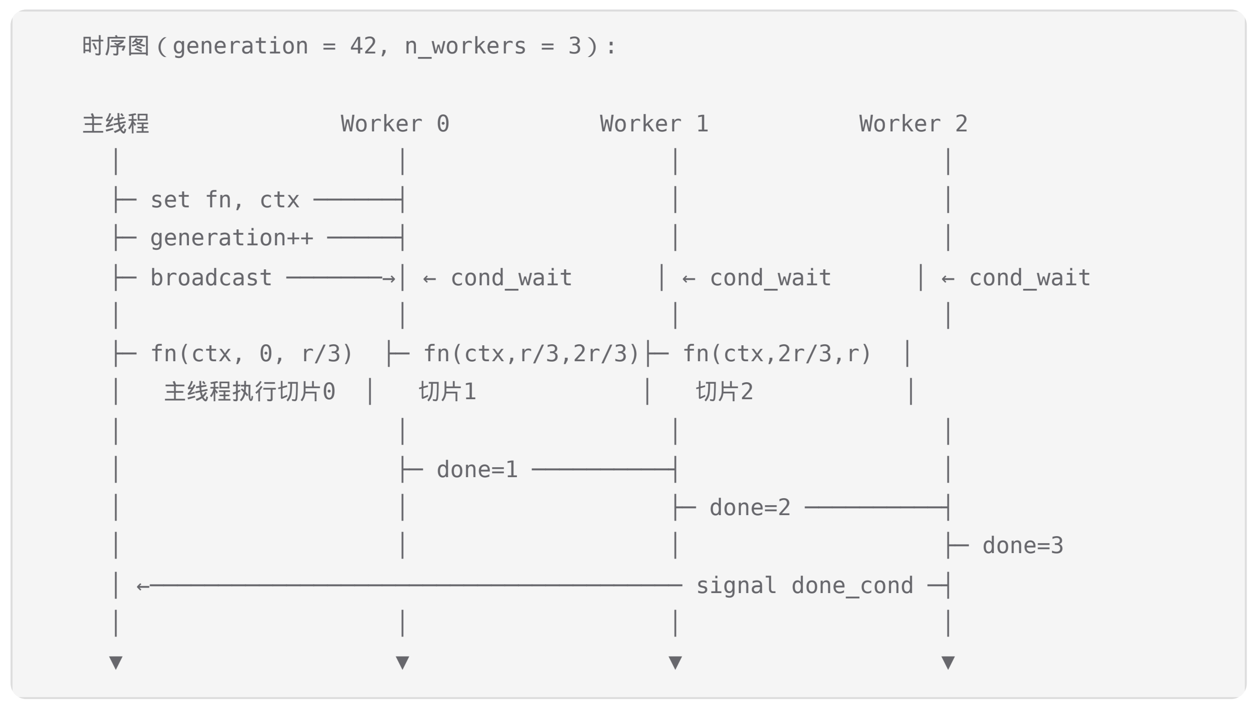
内存布局

```
g_pool (全局单例)

┌──────────────────────────────────────┐
│ threads[0..31]  (pthread_t × 32)    │  ← 线程句柄
│ mutex            (pthread_mutex_t)   │  ← 互斥锁
│ work_cond        (pthread_cond_t)    │  ← 唤醒工人
│ done_cond        (pthread_cond_t)    │  ← 唤醒主线程
│ n_threads = 9                        │  ← 例如 8 核 + 主线程
│ n_workers = 8                        │
│ generation = 42                      │  ← 当前任务编号
│ done = 0                             │  ← 已完成计数
│ fn = matvec_q8_0_rows_worker         │  ← 当前任务函数
│ ctx = &matvec_ctx                    │  ← 任务上下文指针
│ n_rows = 32768                       │  ← 总行数
└──────────────────────────────────────┘


线程 0 (主线程)     线程 1..8 (工作线程)
  g_parallel_depth=0    g_parallel_depth=1
  (独立副本)            (独立副本)
```

追踪 2：Worker 主循环与 barrier 同步

代号机制

```c
// 行 636: 工作线程入口
static void *ds4_worker_main(void *arg) {
    const int tid = (int)(uintptr_t)arg;        // 线程编号
    uint32_t seen_generation = 0;               // 已处理的代号

    for (;;) {
        // 等待新任务
        pthread_mutex_lock(&g_pool.mutex);
        while (seen_generation == g_pool.generation && !g_pool.shutdown) {
            pthread_cond_wait(&g_pool.work_cond, &g_pool.mutex);
            //                    ^^^^^^^^^^^^      ^^^^^^^^^^^^
            //                    等待被唤醒          释放锁并等待
        }
        if (g_pool.shutdown) {
            pthread_mutex_unlock(&g_pool.mutex);
            return NULL;                        // 关闭退出
        }
        // 取任务参数（在锁内，安全）
        const ds4_parallel_fn fn = g_pool.fn;
        void *ctx = g_pool.ctx;
        const uint64_t n_rows = g_pool.n_rows;
        seen_generation = g_pool.generation;
        pthread_mutex_unlock(&g_pool.mutex);

        // 计算行范围
        const uint64_t rows_per_thread = (n_rows + g_pool.n_threads - 1)
                                         / g_pool.n_threads;
        const uint64_t row0 = tid * rows_per_thread;
        const uint64_t row1 = MIN(row0 + rows_per_thread, n_rows);

        // 执行工作（标记递归深度）
        g_parallel_depth++;
        fn(ctx, row0, row1);                    // ← 实际工作
        g_parallel_depth--;

        // 完成通知
        __sync_fetch_and_add(&g_pool.done, 1);   // 原子递增
        if (g_pool.done == g_pool.n_workers) {
            pthread_mutex_lock(&g_pool.mutex);
            pthread_cond_signal(&g_pool.done_cond);  // 唤醒主线程
            pthread_mutex_unlock(&g_pool.mutex);
        }
    }
}
```



追踪 3：线程池初始化与关闭

初始化

```c
// 行 678: 初始化线程池
static void ds4_threads_init(void) {
    if (g_pool.initialized) return;

    // 默认: min(CPU 核心数, 12)
    long nprocs = sysconf(_SC_NPROCESSORS_ONLN);
    uint32_t n = (uint32_t)nprocs;
    if (n > 12) n = 12;                        // 上限 12 线程

    // 环境变量覆盖
    const char *env = getenv("DS4_THREADS");
    if (env) n = atoi(env);
    if (n < 1) n = 1;
    if (n > DS4_MAX_THREADS) n = DS4_MAX_THREADS;

    g_pool.n_threads = n;
    g_pool.n_workers = n - 1;                  // 主线程也算一个

    // 创建 n-1 个工作线程（主线程是 worker 0）
    for (uint32_t i = 1; i < n; i++) {
        pthread_create(&g_pool.threads[i], NULL,
                       ds4_worker_main, (void *)(uintptr_t)i);
    }
    g_pool.initialized = true;
}
```

关闭

```c
// 行 715: 优雅关闭
static void ds4_threads_shutdown(void) {
    if (!g_pool.initialized) return;

    pthread_mutex_lock(&g_pool.mutex);
    g_pool.shutdown = true;                    // 设置关闭标志
    g_pool.generation++;                       // 代号递增触发唤醒
    pthread_cond_broadcast(&g_pool.work_cond); // 广播唤醒所有工人
    pthread_mutex_unlock(&g_pool.mutex);

    // 等待所有工作线程退出
    for (uint32_t i = 1; i < g_pool.n_threads; i++) {
        pthread_join(g_pool.threads[i], NULL);
    }
    g_pool.initialized = false;
}
```

追踪 4：parallel_for 分发机制

行范围分配策略

```c
// 行 736: 带最小行数阈值的并行分发
static void ds4_parallel_for_min_rows(
        uint64_t n_rows, ds4_parallel_fn fn, void *ctx,
        uint64_t min_parallel_rows) {

    // 退化为串行的情况:
    if (g_parallel_depth > 0)       goto serial;  // 嵌套并行 → 串行
    if (g_pool.n_threads <= 1)      goto serial;  // 单线程
    if (n_rows < min_parallel_rows) goto serial;  // 行数太少

    // 并行分发
    ds4_threads_init();                          // 懒初始化
    pthread_mutex_lock(&g_pool.mutex);
    g_pool.fn = fn;
    g_pool.ctx = ctx;
    g_pool.n_rows = n_rows;
    g_pool.done = 0;
    g_pool.generation++;                         // 新代号
    pthread_cond_broadcast(&g_pool.work_cond);   // 唤醒工人
    pthread_mutex_unlock(&g_pool.mutex);

    // 主线程执行切片 0
    {
        const uint64_t rpt = (n_rows + g_pool.n_threads - 1) / g_pool.n_threads;
        g_parallel_depth++;
        fn(ctx, 0, MIN(rpt, n_rows));            // 主线程的行范围
        g_parallel_depth--;
    }

    // 等待所有工人完成
    pthread_mutex_lock(&g_pool.mutex);
    while (g_pool.done < g_pool.n_workers) {
        pthread_cond_wait(&g_pool.done_cond, &g_pool.mutex);
    }
    pthread_mutex_unlock(&g_pool.mutex);
    return;

serial:
    fn(ctx, 0, n_rows);                          // 直接串行执行
}

// 行 770: 默认最小行数 512 的便捷封装
static void ds4_parallel_for(uint64_t n_rows,
                             ds4_parallel_fn fn, void *ctx) {
    ds4_parallel_for_min_rows(n_rows, fn, ctx, 512);
}
```

```
行分配示例 (n_rows=32768, n_threads=9):

rows_per_thread = ceil(32768 / 9) = 3641

Thread 0 (主):  [    0, 3641)
Thread 1:       [ 3641, 7282)
Thread 2:       [ 7282, 10923)
...
Thread 8:       [29128, 32768)

每个线程处理约 3641 行，负载基本均衡
最后一线程可能略少（32768 - 29128 = 3640）
```

追踪 5：TLS 与并发安全

递归并行防护

```
// 行 633: 线程局部存储
static __thread int g_parallel_depth;
```

```
为什么需要防止递归并行？

场景: matvec_q8_0_rows 被 parallel_for 调用
       内部某个函数又调用 parallel_for

ds4_parallel_for → fn(ctx, 0, rows)
  └── some_matvec_worker(...)
        └── ds4_parallel_for(n, sub_fn, sub_ctx)
              └── 如果不加防护 → 死锁！
                  所有工作线程都在执行第一个任务
                  没有线程空闲来执行第二个任务

防护机制:
  g_parallel_depth++ → 值变为 1
  ds4_parallel_for_min_rows 检测到 depth > 0
  → 退化为串行执行 fn(ctx, 0, n_rows)
  g_parallel_depth-- → 恢复为 0
```

原子操作

```c
// Worker 完成通知使用原子操作（无锁）
__sync_fetch_and_add(&g_pool.done, 1);   // 原子递增
if (g_pool.done == g_pool.n_workers) {   // 最后一个完成
    pthread_cond_signal(&g_pool.done_cond);
}
```

`done` 的递增不需要 mutex 保护，因为 `__sync_fetch_and_add` 是编译器内置的原子操作。只有当最后一个 worker 完成时才需要锁来 signal 条件变量。

Part 5: 服务层

> 从单用户 CLI 到多用户 HTTP 服务：POSIX socket、poll I/O 复用、SSE 流式、KV 持久化、OpenAI/Anthropic API 兼容。让推理引擎变成可编程的服务。

涵盖内容

章节	核心主题
1. HTTP 服务器	POSIX socket、poll()、sse 流式、Worker 架构
2. KV Cache 持久化	文件 I/O、SHA1、序列化、原子写入、前缀匹配
3. API 兼容层	JSON 手写解析、OpenAI/Anthropic API、Tool Calling、DSML

核心概念

- sse — Server-Sent Events 流式响应
- kv-cache — 持久化到磁盘的 KV 缓存
- hash-table — DSML replay map 中的 radix tree

前置知识

- Part 4: 推理流程（线程池、prefill/decode）
- 基本的网络编程概念（socket、HTTP）

学习路径

读完本主题后，你将理解：

1. ds4-server 如何用单线程事件循环 + Worker 线程池处理并发请求
2. KV Cache 如何序列化到磁盘并按前缀匹配复用
3. 如何兼容 OpenAI 和 Anthropic 两种 API 协议

→ 下一步：Part 6: GPU 加速

Part 5: 服务层

> HTTP 服务器、KV 持久化、API 兼容

---

1. HTTP 服务器

> 推理引擎做好了，但用户怎么用它？需要一个 HTTP 服务器接收请求、排队推理、流式返回结果。今天学习从 POSIX socket 到 SSE 流式输出的完整网络栈——全部手写，零依赖。

HTTP 服务器使用 线程池 处理并发连接。

设计决策推导：推理服务器的并发模型

```
问题 1：推理引擎是 C 代码，怎么让 Python/JS 客户端调用？
  └─ 方案：HTTP 服务器 — 任何语言都有 HTTP 客户端库

问题 2：推理很慢（decode 约 5-10 t/s），用户要等几十秒才看到输出？
  └─ 方案：SSE（Server-Sent Events）— 每生成一个 token 立即推送
     用户看到"逐字输出"的效果，体感延迟大幅降低

问题 3：推理是串行的（GPU 独占），但 HTTP 连接是并发的？
  └─ 方案：生产者-消费者模式 — client 线程接收请求并入队，
     worker 线程按 FIFO 顺序处理推理任务
     → HTTP 并发，推理串行，用队列解耦

问题 4：TCP 发送缓冲区满时，send() 返回 EAGAIN，丢数据怎么办？
  └─ 方案：poll() I/O 多路复用 — 等到 fd 可写再重新发送
     → 不阻塞其他线程，不丢数据

问题 5：每个连接一个线程，线程数不就爆炸了吗？
  └─ 实际不会：推理是瓶颈，同时只有 1 个活跃推理。
     并发连接数通常 < 10，线程开销可忽略。
     如果要做 C10K，需要改为事件驱动 + 协程，但对推理服务器不需要。
```

C 知识点

1. POSIX Socket

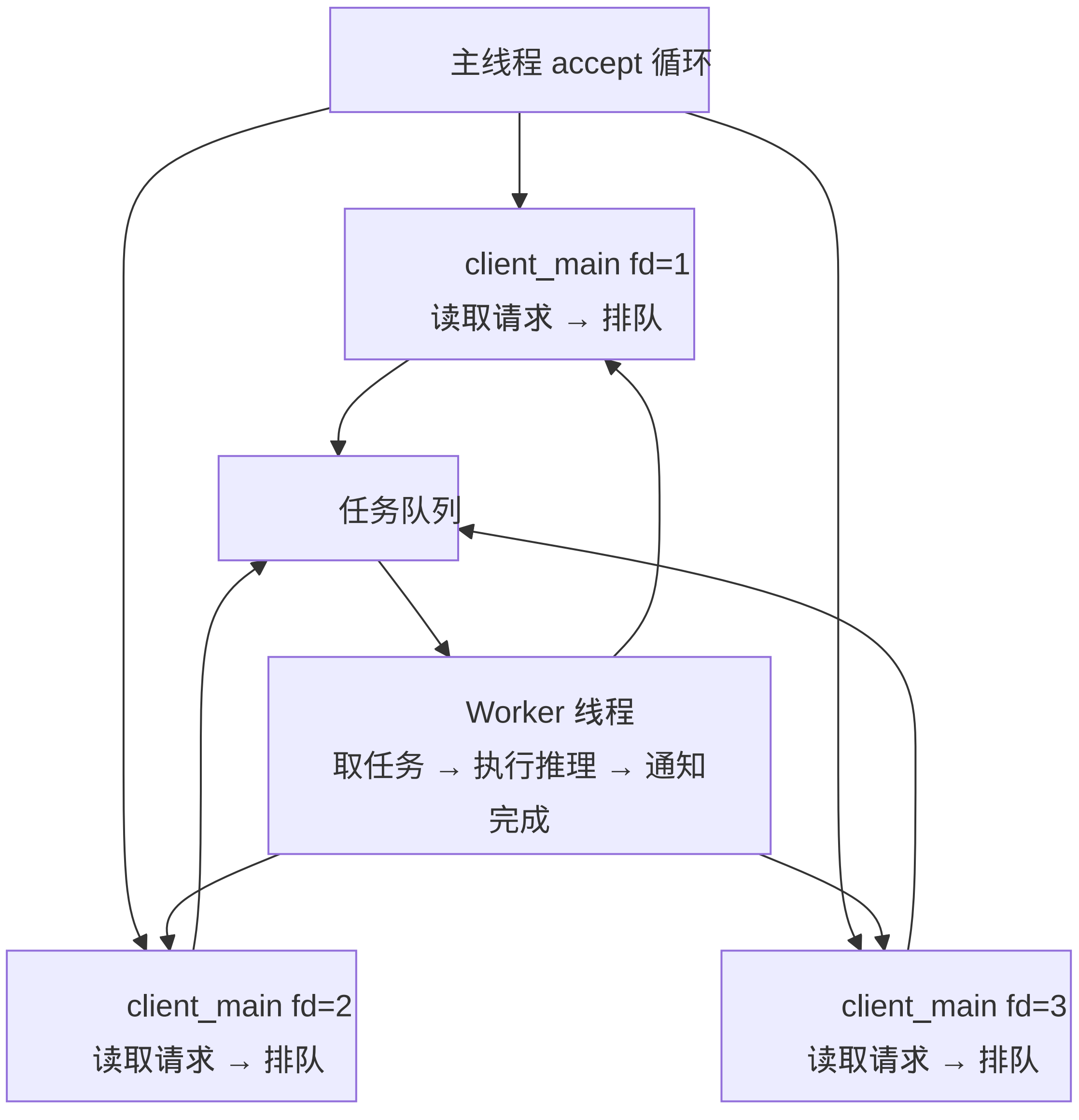
网络通信的基础 API：

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);      // 创建 TCP socket
bind(fd, (struct sockaddr *)&addr, sizeof(addr)); // 绑定地址
listen(fd, 16);                                  // 开始监听
int client = accept(fd, NULL, NULL);             // 接受连接
recv(client, buf, sizeof(buf), 0);               // 读取数据
send(client, buf, len, 0);                       // 发送数据
close(client);                                   // 关闭连接
```

> ds4_server.c 的 `listen_on`（行 6020-6045）创建监听 socket。

2. 每连接一线程模型

ds4_server 的架构：



推理是串行的（一个 GPU），但 HTTP 处理是并发的（多个线程）。

3. poll() I/O 多路复用

ds4_server 用 `poll` 处理慢客户端：

```
ssize_t w = send(fd, s, n, 0);
if (w < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
    struct pollfd pfd = {.fd = fd, .events = POLLOUT};
    poll(&pfd, 1, timeout);    // 等待 socket 可写
    continue;
}
```

当发送缓冲区满时，`send` 返回 `EAGAIN`。`poll` 等待直到缓冲区有空间，避免忙等。

3a. 服务器启动选项

`ds4-server` 支持以下关键启动选项：

选项	说明
`--host`	绑定地址（默认 `127.0.0.1`，LAN 需显式 `0.0.0.0`）
`--port`	监听端口
`--threads`	工作线程数
`--cors`	启用 CORS，为浏览器 JavaScript 客户端添加 `Access-Control-Allow-*` 响应头，处理 OPTIONS 预检请求。不改变绑定地址
`--chdir DIR`	启动时切换工作目录，使相对路径（模型、trace、KV cache 目录、`metal/*.metal`）相对于指定目录解析。适用于 service manager 从任意目录启动 ds4-server 的场景
`--gpu-stats`	报告 GPU 统计信息

LLM 知识点

1. SSE 流式响应

SSE（Server-Sent Events）让服务器可以逐步发送数据：

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
Cache-Control: no-cache

data: {"choices":[{"delta":{"content":"Hello"}}]}

data: {"choices":[{"delta":{"content":" world"}}]}

data: [DONE]
```

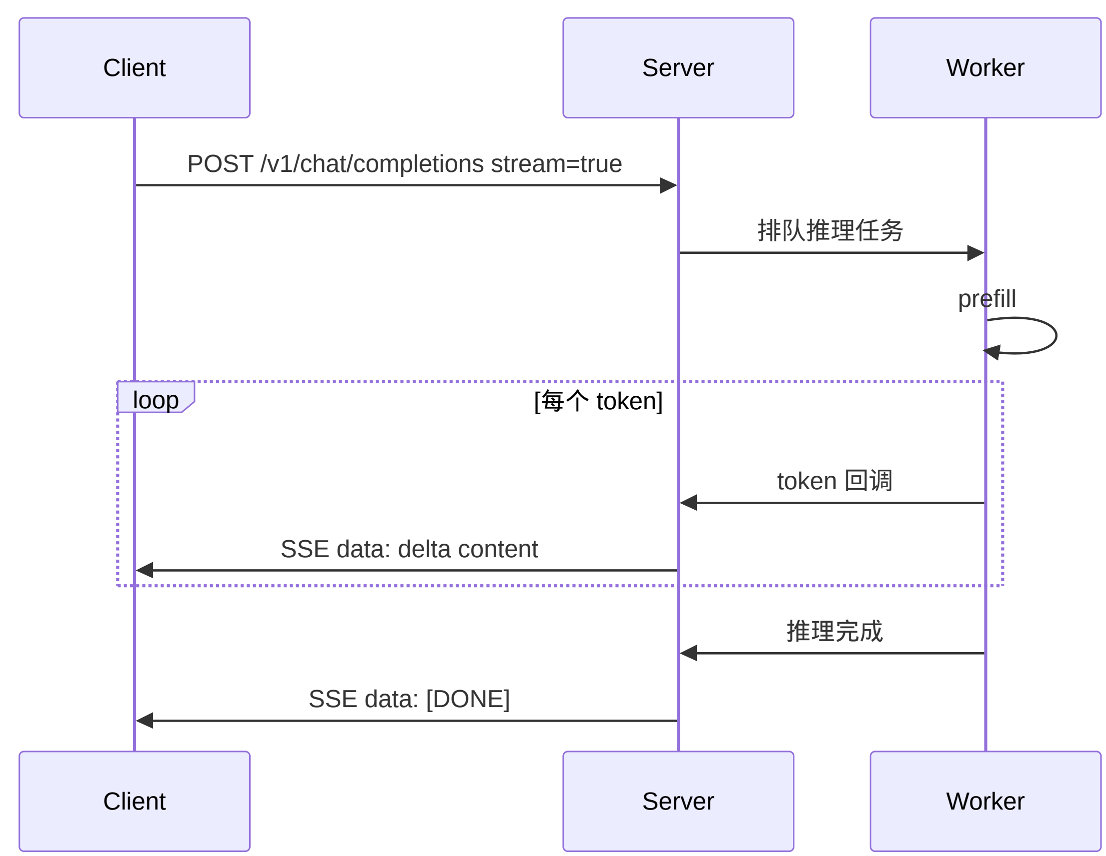
每条消息以 `data: ` 开头，以两个换行结束。客户端收到一条就显示一条，实现"打字机"效果。

SSE Keepalive（防 **TCP** 超时）

长 prompt 的 prefill 可能持续数十秒，期间 SSE 流没有新数据。部分代理/负载均衡器会在 5-10 秒无数据后关闭连接。ds4_server 在 prefill 期间每 5 秒发送 SSE 注释行作为心跳：

```
: prefill\n\n
```

SSE 规范中 `:` 开头的行是注释，客户端会忽略，但足以保持 TCP 连接活跃。`server_prefill_progress` 结构体跟踪上次心跳时间，在 prefill 进度回调中检查是否需要发送。



2. OpenAI vs Anthropic API

ds4_server 同时兼容两种 API：

OpenAI 格式：

```
POST /v1/chat/completions
{"messages":[{"role":"user","content":"Hello"}],"stream":true}
```

Anthropic 格式：

```
POST /v1/messages
{"messages":[{"role":"user","content":"Hello"}],"stream":true}
```

差异：字段名不同、thinking 的表示方式不同、tool calling 格式不同。ds4_server 在内部统一处理，只是输入/输出的适配层不同。

所有三种 API 格式现在都在 `usage` 字段中报告 KV cache 命中详情：

```
// OpenAI: prompt_tokens_details.cached_tokens + cache_write_tokens
{"usage":{"prompt_tokens":25000,"prompt_tokens_details":{"cached_tokens":2400
0,"cache_write_tokens":600}}}

// Anthropic: cache_read_input_tokens + cache_creation_input_tokens
{"usage":{"input_tokens":400,"output_tokens":150,"cache_read_input_tokens":2400
0,"cache_creation_input_tokens":600}}

// Responses: input_tokens_details.cached_tokens + cache_write_tokens
{"usage":{"input_tokens":25000,"input_tokens_details":{"cached_tokens":24000,"ca
che_write_tokens":600},"output_tokens":150}}
```

`cached_tokens` 是从磁盘 KV cache 直接命中的 token 数，`cache_write_tokens` 是本次新增并写入 cache 的 token 数。OpenAI 的 `cached_tokens` 保持只读（不计入 write），以兼容标准客户端。

3. Responses API (`/v1/responses`)

OpenAI Responses API 端点，支持 Codex CLI 等 Agent 工具的多轮工具调用。与 `/v1/chat/completions` 的核心区别是请求-响应模型——每个响应对象包含完整的工具调用结果，客户端通过 `call_id` 绑定后续请求与 live KV 状态，实现毫秒级续接。Server 日志新增 Responses API 标记和进度日志改进，方便调试多轮工具调用。

---

2. KV Cache 持久化

> 长 prompt 的 prefill 需要数秒。多轮对话中如果每轮都从头 prefill，用户体验极差。将 KV Cache 序列化到磁盘，下次对话复用前缀——重复 prefill 从 10 秒降到 1 秒。

将 KV Cache 结构 序列化到磁盘，利用 mmap 快速重新加载。

C 知识点

1. 文件 I/O

KV 持久化已从 `ds4_server.c` 提取为独立模块 `ds4_kvstore.c/h`，实现格式和策略与服务器解耦。

ds4_kvstore 用标准文件 I/O 保存 KV cache（不用 mmap，避免增加 VM 映射）：

```c
FILE *fp = fopen(path, "wb");              // 打开文件写
fwrite(header, 1, 48, fp);                 // 写头部
fwrite(text, 1, text_len, fp);             // 写文本
ds4_session_save_payload(session, fp, ...); // 写 session 数据
fclose(fp);
rename(tmp_path, final_path);              // 原子重命名
```

原子写入模式：先写 `.tmp.pid` 临时文件，完成后 `rename` 为最终文件名。如果中途崩溃，临时文件不完整但不会损坏已有缓存。

2. SHA1 哈希

手写的 SHA-1 实现（行 4095-4221）用于生成缓存文件名：

```c
// 缓存 key = SHA1(token_ids 的字节)
static void sha1_tokens_hex(const ds4_tokens *tokens, int n, char out[41]) {
    sha1_ctx c;
    sha1_init(&c);
    for (int i = 0; i < n; i++) {
        uint8_t b[4];
        le_put32(b, (uint32_t)tokens->v[i]);   // token ID → 4 字节小端
        sha1_update(&c, b, sizeof(b));
    }
    sha1_final(&c, digest);
    hex20(digest, out);                         // 20 字节 → 40 字符 hex
}
```

为什么用 token IDs 而不是文本？

- 同一段文本可能被不同分词器产生不同的 token 序列
- token IDs 是精确的，避免文本编码差异导致的缓存 miss

3. 序列化格式

磁盘上的 KV cache 文件格式（由 `ds4_kvstore` 模块管理）：

KVC 固定头部 48 字节
magic KVC
version
model_id
save_reason
cached_token_co
hit_count
creation_time
last_used_time
rendered_text
text_len u32
rendered_text UTF-8 人类可读
DS4 session payload
magic DSV4
token IDs array
logits float32
compressed KV rows
compressor states
indexer states
协议特定尾部（可选）
KTM 工具记忆映射
responses_visible 标记

头部新增 `model_id` 字段区分不同模型变体的缓存。尾部（trailer）挂载协议特定数据——工具记忆映射、Responses API 可见性标记等——由调用方通过钩子注入，`ds4_kvstore` 本身不解析尾部内容。

LLM 知识点

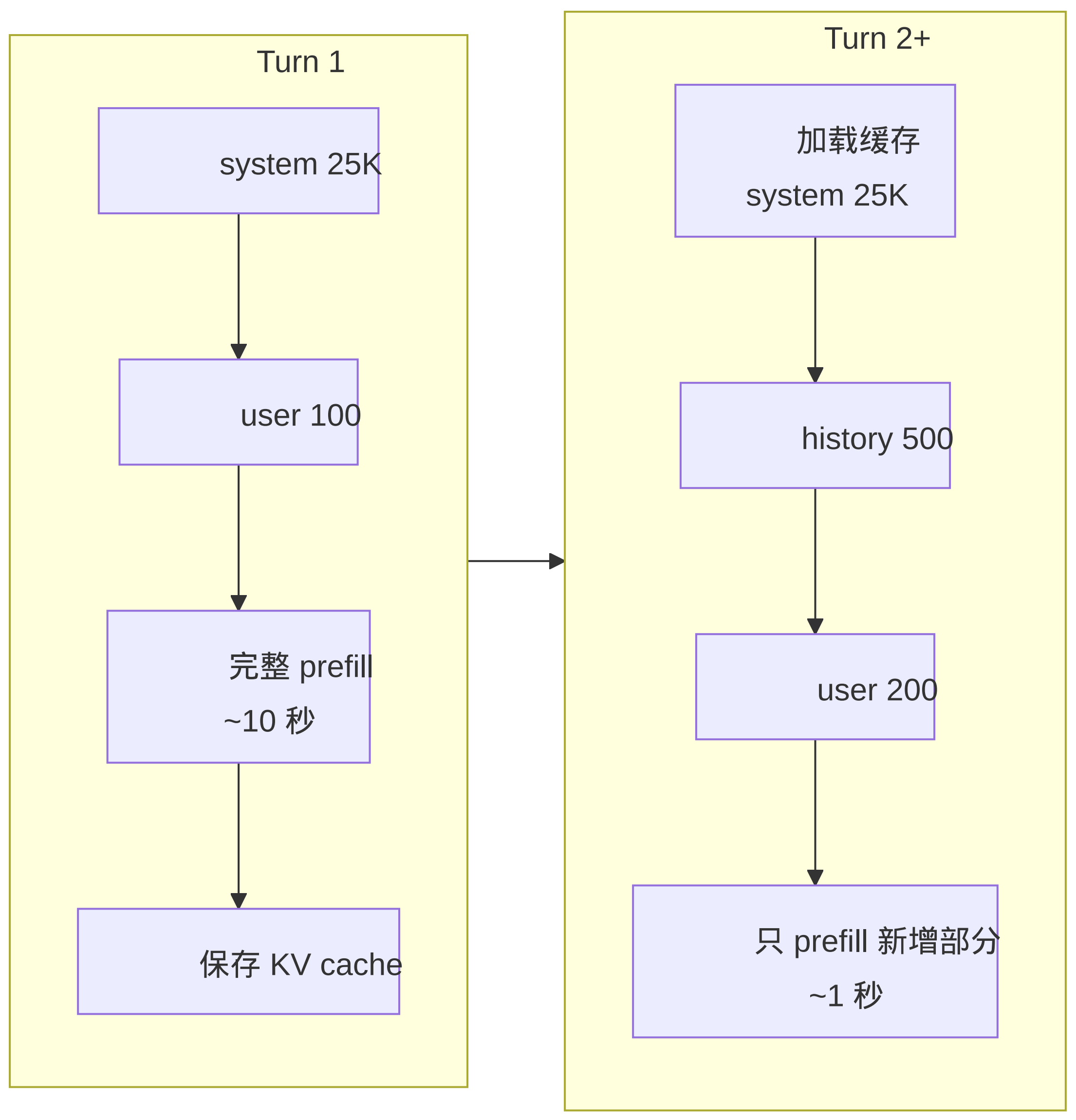
1. 为什么需要 KV Cache 持久化？

编码代理（如 Claude Code）的工作模式：

1. 发送很长的系统提示（~25K token）+ 用户消息
2. 模型回复
3. 下一轮发送更长的提示（系统 + 历史 + 新消息）

```
第 1 轮: [system(25K)] [user(100)] → model reply
第 2 轮: [system(25K)] [history(500)] [user(200)] → model reply
                 ↑ 这 25K token 每次都要重新处理！
```

没有磁盘缓存：每轮都要 prefill 25K+ token（~10 秒） 有磁盘缓存：第一轮 prefill 后保存，后续轮次直接加载（~1 秒）



2. 缓存触发时机

```
cold:            首次 prefill 后，长 prompt 达到稳定前缀
continued:       推理过程中按间隔保存（绝对对齐前沿，约每 10K token）
evict:           新请求替换当前 session 前，保存旧的
shutdown:        服务器退出时保存
agent_system:    ds4-agent 系统提示稳定后保存
agent_session:   ds4-agent 会话状态保存
```

`ds4_kvstore_chat_anchor_pos()` 在冷保存时定位 chat task boundary（最后一个 `<User>` special token 之后），最大化跨独立 agent 会话的复用率。

驱逐策略：命中率衰减 **+** 当前存储保护

磁盘 KV cache 有容量限制（`--kv-cache-budget-mb`，默认 4096 MiB），满了需要驱逐旧条目。驱逐逻辑已提取到 `ds4_kvstore` 模块，评分公式：

```
score = (effective_hits + 1) × tokens / file_size
```

命中率衰减：`effective_hits` 不是原始 `hits` 计数，而是按时间衰减：

```
// 半衰期 6 小时：不活跃的旧 checkpoint 的命中奖励逐渐归零
effective_hits = hits × 2^(-elapsed_seconds / 21600)
// 低于 0.01 时直接归零，避免微小的浮点奖励干扰排序
```

为什么需要衰减？工作负载变化（prompt 或 tool schema 改变）会让曾经热门的 checkpoint 无法匹配。不衰减的话，一个 24 小时前的 100-hit 条目会永远排在新条目前面。

当前存储保护：`evict` 时传入刚写入文件的 SHA1，驱逐循环给这个文件最高评分（`DBL_MAX`），防止满缓存把刚保存的文件立即删掉——那等于白写了。如果文件本身超出预算，保护自动取消。

预算预检：保存前检查文件是否能放进预算（1% 余量），放不进的直接跳过，不触发无意义的驱逐循环。

分数相同时，按 `last_used` 时间排序——旧的先删。

预存储驱逐与兼容性加固

预存储驱逐（pre-store eviction）：驱逐在写入新条目之前执行，传入待写入条目的上下文（`model_id`、`quant_bits`、`ctx_size`、`text`）。这样避免了两个问题：

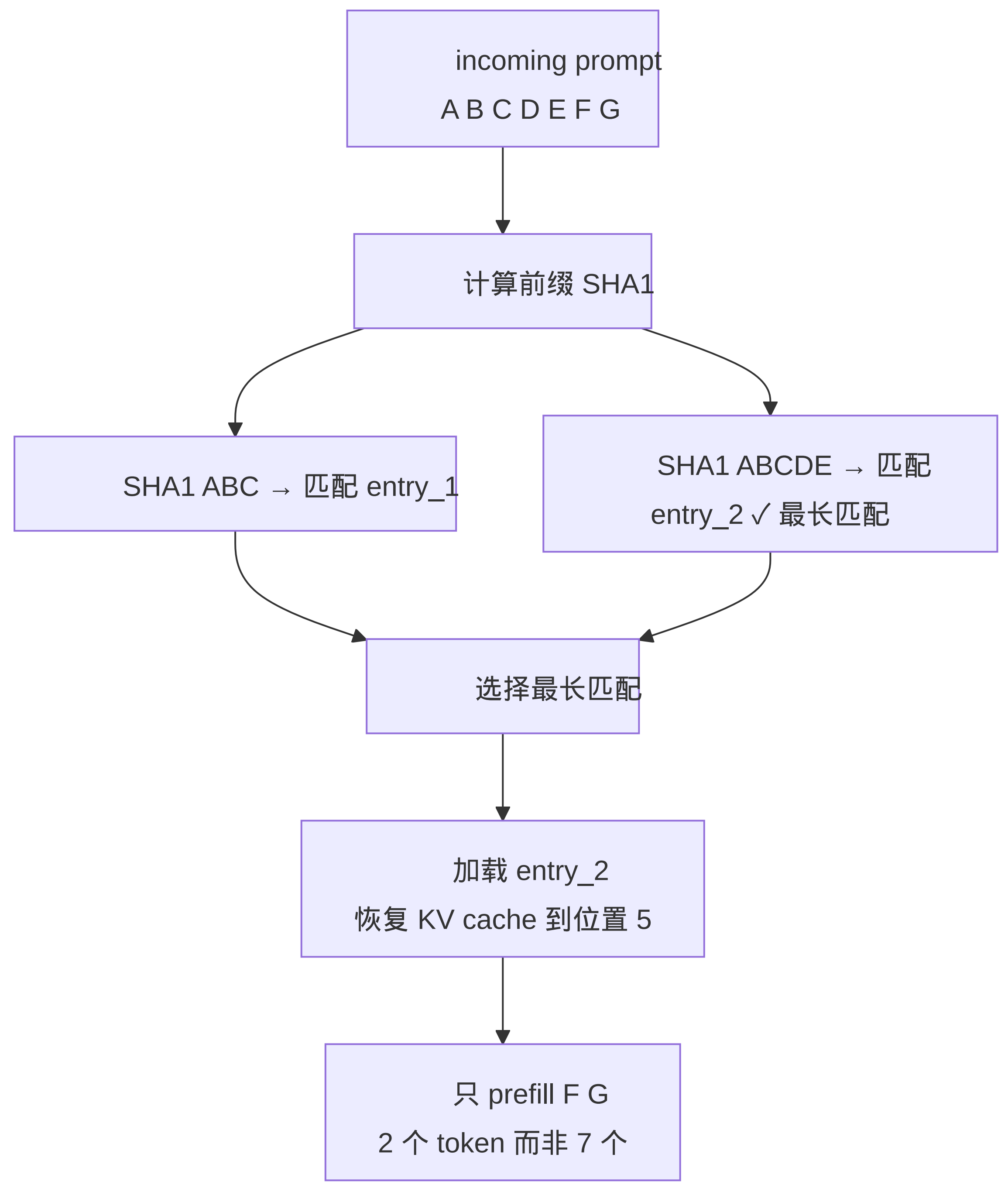
1. 自驱逐：新条目刚写入就被自己的驱逐循环选中删除（"进来就被选为自己的受害者"）
2. 前缀替代：如果新条目是旧条目的严格前缀扩展（同一 `model_id` + `quant_bits`、更长或相等的 `ctx_size`），旧条目自动获得评分惩罚，自然被新条目替代

锚点保护：`cold`、`evict`、`shutdown` 类型的检查点是"锚点"（高价值的刻意保存），评分乘以 `KV_CACHE_ANCHOR_REASON_SCORE_FACTOR`，更难被驱逐。

兼容性加固（`kv_cache_existing_compatible`）：加载磁盘 KV 缓存时，严格校验文件头与当前运行时的兼容性——`model_id`、`quant_bits`、`ctx_size`、以及文件内容的 SHA1 完整性。不兼容的文件自动删除并重建。这修复了不干净关机后磁盘缓存"看似正常但 prefill 总是失败"的无限循环问题。

3. 前缀匹配

缓存查找过程：



4. 冷启动优化

`cold` 保存时会修剪尾部 token：

```
原始 prompt: 10000 个 token
trim 32 个尾部 token → 保存前 9968 个
对齐到 2048 边界 → 保存前 8192 个
```

为什么？BPE 分词器在追加文本时可能改变前面几个 token 的编码。修剪 + 对齐减少这种"边界 retokenization miss"。

Cold Checkpoint 的 Chat Task Boundary 锚定

冷 KV 检查点的裁剪位置现在锚定在 **chat task boundary**——最后一个 `<User>` special token 之后、第一个 `<Assistant>` special token 之前。这样最大化跨独立 agent 会话的复用率：保留稳定的用户角色脚手架（如 Codex 的 `environment_context`），而不保留属于对话历史的后续多轮 user 标记。

实现使用精确的 chat special token ID 在 token 层面定位锚点。如果 prompt 中没有有效的 chat anchor（非聊天场景或过短的 prompt），则回退到原有的 trim + 对齐启发式。

---

3. API 兼容层

> 服务器能跑了，但客户端期望 OpenAI 格式的请求和响应。今天学习手写递归下降 JSON 解析器、OpenAI/Anthropic API 双向兼容，以及 Tool Calling 的 DSML 解码——全部手写，没有用任何外部库。

API 兼容层为同一主题的 HTTP 服务器章节提供 OpenAI 兼容接口。

C 知识点

1. 手写 JSON 解析器

ds4_server 不依赖任何 JSON 库，自己实现了一个递归下降解析器（行 141-437）：

```
// 跳过空白
static const char *json_ws(const char *p);

// 解析字符串（处理转义）
static const char *json_string(const char *p, char **out);

// 解析数字
static const char *json_number(const char *p, double *out);

// 解析整数
static const char *json_int(const char *p, int64_t *out);

// 跳过任意 JSON 值
static const char *json_skip_value(const char *p);
```

为什么不 cJSON / jansson？

- 零依赖：整个项目不需要安装任何第三方库
- 精确控制：只需要解析请求中的特定字段
- 安全性：避免第三方库的未知漏洞

嵌套深度限制（防止栈溢出 **DoS**）

`json_skip_value` 内部使用带深度跟踪的版本递归解析：

```
#define JSON_MAX_NESTING 256

// 内部版本：跟踪递归深度
static const char *json_skip_value_depth(const char *p, int depth);
static const char *json_skip_array_depth(const char *p, int depth);
static const char *json_skip_object_depth(const char *p, int depth);

// 公开 API：入口包装
const char *json_skip_value(const char *p) {
    return json_skip_value_depth(p, 0);
}
```

为什么需要深度限制？忽略的请求字段可能包含 `{"x":[[[...]]]]}` 这样的深层嵌套结构。如果不加限制，递归调用会逐层消耗 C 调用栈，直到栈溢出才被操作系统拒绝。`JSON_MAX_NESTING 256` 在合理范围内截断，返回 `false` 拒绝请求。

2. JSON 解析模式

```c
// 查找 JSON 对象中的字段
const char *p = body;
if (!json_expect(p, '{')) return error;
while (*p != '}') {
    char *key;
    p = json_string(p, &key);
    p = json_ws(p);
    if (!strcmp(key, "messages")) {
        p = parse_messages(p, &r->messages);
    } else if (!strcmp(key, "temperature")) {
        p = json_number(p, &r->temperature);
    } else {
        p = json_skip_value(p);    // 跳过不认识的字段
    }
    free(key);
    p = json_ws(p);
    if (*p == ',') p++;
}
```

> 关键：`json_skip_value` 让解析器可以跳过不需要的字段，只解析关心的字段。

3. UTF-8 处理

JSON 字符串中的 `\uXXXX` 转义需要转换为 UTF-8：

```c
// 代理对处理：😀 → 😀 (U+1F600)
if (cp >= 0xD800 && cp <= 0xDBFF) {
    // 高代理 → 等待低代理
    uint32_t hi = cp;
    uint32_t lo;
    // \uYYYY
    cp = 0x10000 + ((hi - 0xD800) << 10) + (lo - 0xDC00);
}
// UTF-8 编码
utf8_put(&out, cp);
```

LLM 知识点

1. OpenAI Chat Completions API

```
POST /v1/chat/completions
{
  "model": "deepseek-v4-flash",
  "messages": [
    {"role": "system", "content":
"You are helpful."},
    {"role": "user", "content":
"Hello"}
  ],
  "temperature": 0.7,
  "max_tokens": 1024,
  "stream": true,
  "tools": [{"type": "function",
"function": {...}}]
}
```

```
POST /v1/messages
{
  "model": "deepseek-v4-flash",
  "system": "You are helpful.",
  "messages": [
    {"role": "user", "content":
"Hello"}
  ],
  "max_tokens": 1024,
  "stream": true,
  "thinking": {"type": "enabled"}
}
```

ds4_server 解析流程：

1. 解析 messages 数组 → 渲染为 DeepSeek 聊天模板
2. 解析 tools → 渲染为 DSML 工具格式
3. 分词 → 推理 → 流式输出

差异：

- system 是顶层字段（不在 messages 里）
- thinking 控制不同的值
- tool calling 格式不同

2b. Context Length 错误（协议标准化）

当 prompt tokens 数量达到或超过 context size 时，`client_main()` 在请求进入 job 队列之前就拦截并返回 400 错误。这比让推理引擎在后端失败更友好——客户端可以精确知道差多少 token。

OpenAI / Responses 格式：

Anthropic 格式：

```
HTTP/1.1 400 Bad Request
{
  "error": {
    "message": "Prompt has 16 tokens, but the configured context size is 16 tokens",
    "type": "invalid_request_error",
    "param": "messages",
    "code": "context_length_exceeded",
    "n_prompt_tokens": 16,
    "n_ctx": 16
  }
}
```

```
HTTP/1.1 400 Bad Request
{"type":"error","error":{
  "type":"invalid_request_error",
  "message": "Prompt has 20 tokens, but the configured context size is 20 tokens",
  "n_prompt_tokens": 20,
  "n_ctx": 20
}}
```

关键设计点：

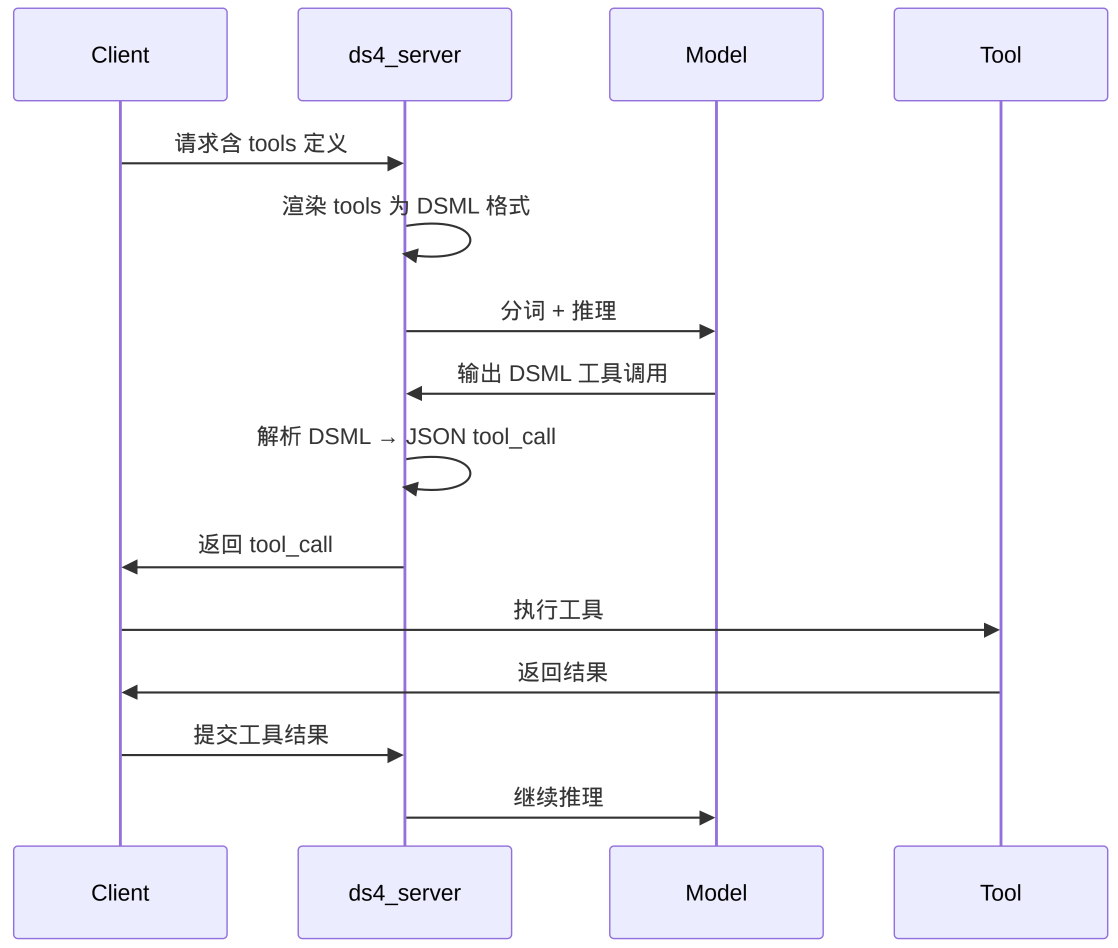
- **`param` 字段自适应**：`context_length_error_param()` 根据请求类型返回不同参数名——`"messages"` 用于 chat、`"prompt"` 用于 completions、`"input"` 用于 Responses API
- 边界精确匹配：`request_exceeds_context()` 检查 `prompt.len >= ctx_size`（而非 `>`），因为 `ds4_session_sync()` 需要至少一个空闲 **slot** 才能生成第一个 token
- 提前拦截：在 `client_main()` 中、入队之前执行检查，避免浪费 worker 线程时间

3. Tool Calling

模型生成工具调用的流程：

```
1. 请求中包含 tools 定义
2. ds4_server 把工具描述渲染为 DSML 格式（现在放在客户端 system prompt **之前**，以保护 KV cache 边界裁剪时优先切掉客户端内容而非工具 schema）：
   <tool_calls>
   <invoke name="get_weather">
   <parameter name="city" string="true">Tokyo</parameter>
   </invoke>
   </tool_calls>

3. 模型输出 DSML 格式的工具调用
4. ds4_server 解析 DSML → 转换为 OpenAI/Anthropic 的 tool_call 格式
5. 客户端执行工具，返回结果
6. 继续对话
```

DSML 是 DeepSeek 的工具调用格式。ds4_server 做 DSML ↔ OpenAI/Anthropic 格式的双向转换。



畸形工具调用恢复

模型有时生成无法解析的 DSML（格式错误、嵌套异常等）。之前的做法是终止整个助手轮次，返回 `finish_reason="error"`。现在改为优雅降级：

```
// parse_generated_message_for_response 包装核心解析器
bool parsed_ok = parse_generated_message(text, &content, &reasoning, &calls);
if (!parsed_ok) {
    // 解析失败 → 将原始文本作为普通助手消息返回
    content = xstrdup(text);
    finish = tool_parse_failure_recovery_finish(finish);
    // finish_reason="length" 保持不变，其他变为 "stop"
}
```

DSML 修复算法（`try_repair_dsml`）

当模型因 `max_tokens` 截断而输出不完整的 DSML 时，`try_repair_dsml` 尝试修复：

1. 从最后一个 `</thinking>` 之后搜索 DSML 起始标记
2. 检测 DSML 风格（标准 `<｜DSML｜tool_calls>` 格式）
3. 统计已打开但未关闭的标签，计算嵌套深度

4. 按嵌套的逆序追加缺失的关闭标签（先关 `<parameter>` 再关 `<invoke>`）

修复后的 DSML 重新送入解析器。如果修复失败，仍然回退到原始文本作为普通内容返回。

嵌套 DSML 参数解析

模型有时生成松散的嵌套结构——外层 `<parameter>` 省略 `string` 属性，内部包含子 `<parameter>`：

```
<parameter name="changes">
  <parameter name="file" string="true">main.py</parameter>
  <parameter name="content" string="true">...</parameter>
</parameter>
```

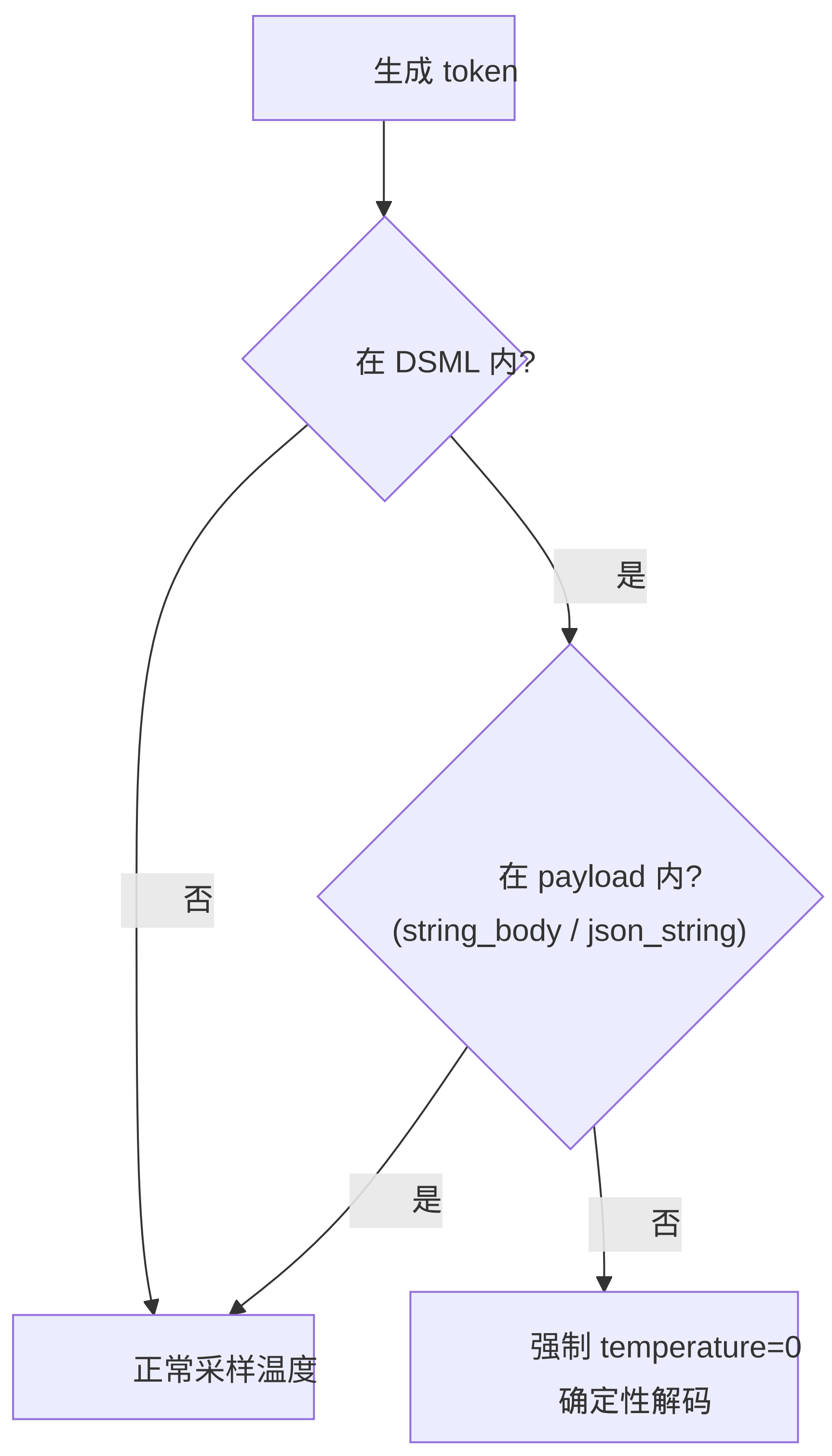
`dsml_parse_nested_params_object` 递归解析这些嵌套参数，将它们合并为 JSON 对象 `{file: "main.py", content: "..."}`，作为外层参数的值。

DSML 解码状态机与采样温度控制

生成过程中，服务器跟踪模型在 DSML 中的位置，对不同部分应用不同温度：

```
typedef enum {
    DSML_DECODE_OUTSIDE,         // 不在工具调用内
    DSML_DECODE_STRUCTURAL,      // DSML 标签、JSON 标点
    DSML_DECODE_STRING_BODY,     // string=true 参数值
    DSML_DECODE_JSON_STRUCTURAL, // JSON 参数内的非字符串部分
    DSML_DECODE_JSON_STRING,     // JSON 字符串值
} dsml_decode_state;
```

中出现重复文本的问题。



关键设计：DSML 标签结构（`<invoke>`、`<parameter>`、JSON 花括号等）用温度 0 保证可解析性；参数载荷（文件内容、编辑文本等）使用正常采样温度避免重复。这解决了之前工具调用参数中出现重复文本的问题。

3b. 工具调用文本记忆（**Tool Replay Memory**）

问题：模型说的是 DSML，客户端来回传递的是 JSON。重新渲染 JSON 不一定产生相同字节——客户端可能重新排序对象键、改变空白等。如果下一轮渲染的 DSML 与模型之前看到的不一致，KV cache 就失效了。

解决方案：rax 基数树（radix tree）精确回放模型采样的原始 DSML。

```
rax.c / rax.h / rax_malloc.h  (~3,091 行)
├── rax:          基数树（压缩前缀树），O(k) 查找，k = key 长度
├── raxNew()      创建树
├── raxInsert()  插入 key-value
├── raxFind()    查找（返回 raxNotFound 表示不存在）
├── raxRemove()  删除
└── 来自 Redis 的成熟数据结构，经过大规模生产验证
```

ds4_server 的 tool_memory 架构：

tool_memory

+rax by_id 按 ID 索引
+rax by_block 按 DSML 内容索引
+entry head LRU 最新
+entry tail LRU 最旧
+int max_entries
+int max_bytes

LRU 链表

tool_memory_entry

+string id 随机 tool call ID
+block block 指向 DSML 共享块
+int source RAM 或 DISK
+entry prev LRU 链表
+entry next LRU 链表

ref

tool_memory_block

+string dsml 原始 DSML 文本
+int refs 引用计数
+entry[] entries

工作流程：

1. 模型生成 **tool call** → `parse_generated_message` 捕获 `raw_dsml` 原始文本
2. 分配随机 **ID** → `random_tool_id()` 从 `/dev/urandom` 读取 16 字节，生成如 `call_a1b2c3...` 或 `toolu_d4e5f6...`
3. 存入 **tool_memory** → 以 ID 为 key、raw_dsml 为 value 存入 rax 树
4. 客户端回传历史 → 携带相同 ID 的 tool_call 出现在 messages 中

5. `tool_memory_attach_to_messages` → 按 ID 查找，将保存的 raw_dsml 附加到 message 的 `calls.raw_dsml`
6. 渲染时直接回放 → `append_dsml_tool_calls_text` 发现 `raw_dsml` 非空就直接使用，不再从 JSON 重新渲染

这样确保了 KV cache 的 token 序列与模型之前看到的完全一致。

工具记忆的磁盘持久化（**KTM** 格式）

工具记忆不只在内存中——它还能保存到 KV cache 文件中，在服务器重启后恢复：

```
KTM (Tool-id Map) 磁盘格式:
0   u8[3]  magic = "KTM"
3   u8     version = 1
4   u32    entry count

For each entry:
0   u32    tool id byte length
4   u32    sampled DSML byte length
8   bytes  tool id
... bytes  exact sampled DSML block
```

恢复流程：服务器启动后，在渲染请求前扫描 cache 文件，查找客户端历史中出现的 tool ID，提前加载对应的 DSML 映射。即使匹配的 KV 快照不是最终使用的那个，精确回放也能在重启后生效。

容量限制：默认最多 100,000 个 ID（`--tool-memory-max-ids`），最大 512 MiB 总字节。`--disable-exact-dsml-tool-replay` 可禁用精确回放，回退到规范 JSON-to-DSML 渲染。

3c. DSML 参数文本与工具结果文本规则

参数文本（`<parameter>` 标签内）：不再转义普通字符，只转义恰好匹配 DSML 关闭标签的文本：

```
// append_dsml_parameter_text
// 只有当文本中恰好出现 "</｜DSML｜parameter>" 时才转义开头 <
// 其他 < > & 字符原样保留

static void append_dsml_parameter_text(buf *b, const char *s) {
    const char *end = "</｜DSML｜parameter>";
    for (s = s ? s : ""; *s;) {
        if (!strncmp(s, end, strlen(end))) {
            buf_puts(b, "&lt;");   // 只转义关闭标签中的 <
            s++;
        } else {
            buf_putc(b, *s++);     // 其他字符原样输出
        }
    }
}
```

工具结果文本（`<tool_result>` 标签内）：保留原始文本不做转义，只保护 `</tool_result>` 关闭标签不被提前截断：

```
// append_tool_result_text
// 工具输出是数据——shell 输出、文件内容等包含大量 < > & 字符
// DeepSeek 渲染器保留原始文本，所以必须原样传入
static void append_tool_result_text(buf *b, const char *s) {
    const char *end = "</tool_result>";
    for (s = s ? s : ""; *s;) {
        if (!strncmp(s, end, strlen(end))) {
            buf_puts(b, "&lt;");   // 防止关闭标签被误匹配
            s++;
        } else {
            buf_putc(b, *s++);     // console.log('<<< < > >>>') 等原样保留
        }
    }
}
```

之前的 `append_dsml_text_escaped` 对所有 `< > &` 做 XML 转义，会导致 `read-file` 工具返回的代码被破坏（如 `console.log('a < b')` 变成 `console.log('a < b')`）。现在参数和结果各有独立的转义函数，只保护各自的关闭标签。

3d. 检查点恢复与 **Session Rewrite**

当 tool replay 匹配失败时（如 ID 不存在于记忆中），ds4_server 使用 session rewrite API 尝试恢复：

```
// 统计工具回放匹配情况
typedef struct {
    int mem;          // 从 RAM 记忆中匹配
    int disk;         // 从磁盘 KV cache 中匹配
    int canonical;    // 需要规范重渲染
    int missing_ids;  // 找不到对应 ID
} tool_replay_stats;
```

匹配失败时，调用 `ds4_session_rewrite_from_common()`：

- 如果 session 检查点与新提示有公共前缀且只需扩展 → 正常 sync
- 如果需要修改已采样的 token → 返回 `DS4_SESSION_REWRITE_REBUILD_NEEDED`
- 服务器回退到磁盘 KV cache 检查点，或最终完整重新 prefill

渲染文本前缀匹配（替代 **token ID** 匹配）

KV cache 的查找键从 token ID 的 SHA1 改为渲染文本字节的 **SHA1**。文件名就是 `<sha1>.kv`，其中 SHA1 是 tokenizer 解码后的文本前缀。

为什么？模型可能生成一个 token，其解码文本在下一轮被客户端拆分为两个规范 prompt token（BPE 边界差异）。渲染字节前缀匹配可以处理这种情况：

```
// byte_prefix_match: 比较渲染文本是否匹配 cache 条目的前缀
// 命中后，使用 checkpoint 中存储的精确 token 作为前缀
// 只对新增的文本后缀重新分词
build_prompt_from_exact_prefix_and_text_suffix(...)
```

对齐前沿（**Aligned Frontiers**）

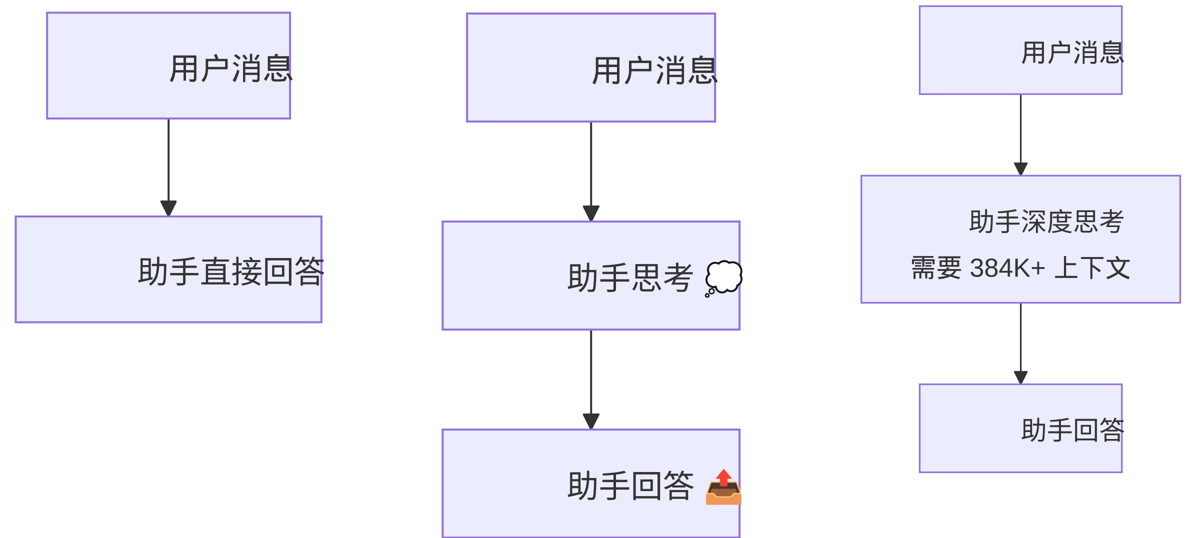
持续保存检查点不再按相对间隔，而是按绝对对齐前沿：

```
// kv_cache_continued_store_target
// 只在绝对对齐位置（如 10k, 20k, 30k... token）保存
// 不相对于上次 cold/evict 保存点
// 防止早期 cold checkpoint 偏移整个保存计划
```

默认约每 10,000 token 保存一次（对齐到 2048 token 块）。这样长生成过程会留下均匀分布的重启点，而不依赖于第一个 cold 检查点落在哪里。

4. Thinking 模式

非 Thinking	Thinking	Think Max



ds4_server 在流式输出中分别处理 thinking 内容和正文内容：

- OpenAI: `reasoning_content` delta vs `content` delta
- Anthropic: `thinking` block vs `text` block

Thinking Checkpoint 重建优化

当 thinking 模式下请求因 `length`（达到 max_tokens）停止时，服务器之前会同步重建 thinking checkpoint，导致客户端感知延迟大幅增加。

修复：提取 `should_canonicalize_thinking_checkpoint()` 函数，在特定条件下跳过 canonicalization：

```
// 跳过 canonicalization 的条件:
// - 非 chat 请求或有 tools 的请求
// - prompt 本身保留了 reasoning 内容
// - 非 thinking 模式
// - finish reason 是 "error" 或 "length" ← 关键修复
// - 仍在 thinking 块内部
```

效果：4k context 的总时间从 30.0s 降到 17.1s，generation t/s 从 7.4 提升到 31.0。

Prefill 进度报告 — Suffix 模式

之前服务器报告 prefill 进度时使用绝对 token 数，但当有缓存命中时，已缓存的 token 不需要重新 prefill，导致进度显示不准确。

修复：改为报告 **suffix** 进度——只计算 `cached_tokens` 之后的 token 区间作为显示范围和百分比基数，更准确地反映实际需要处理的工作量。

无工具 **Thinking** 模式的 **Cache** 修复

当 thinking 模式的请求不使用工具时，渲染逻辑需要正确判断是否保留 reasoning 标记。如果判断错误，会导致字节前缀不匹配，造成 KV cache miss：

```
// chat_history_uses_tool_context: 判断会话是否涉及工具
// 如果请求有 tools 定义，或历史中有 tool/function 角色 → 工具上下文
// 工具上下文影响是否在渲染中包含 think 标记

// prompt_preserves_reasoning: 记录渲染后是否保留了 reasoning 标记
// 用于 KV cache 复用时的前缀匹配判断
```

Thinking 内的 **Tool Call** 隔离

DSML 工具调用只在 thinking 关闭后才能执行。实时解码器在 thinking 开启期间跳过 tool-marker 检测，最终解析器只在最后一个 `</thinking>` 之后搜索可执行 DSML。这防止 thinking 内容中的工具讨论被错误地当作结构化 tool call 处理——同一内容不会被同时作为 thinking 文本和 tool call 重复输出。

未关闭 `<think>` 内的工具调用恢复

上面的隔离规则有个边界情况：模型有时没关闭 **thinking** 就开了一个 **DSML stanza**。由于解码器刻意忽略 `<think>` 内的 tool marker，这种 turn 会一直生成到 token 上限，然后解析阶段把那次调用丢弃，agent 会话卡死（#318）。

修复策略是前向恢复而非重写已采样上下文：当未关闭的 think 块内出现一个完整的 stanza 开头时，强制喂入 `</think>` 加一个空行，让模型接着生成。实测在该位置模型足够强地预测一个全新的 stanza 开头，调用在可执行侧干净重启；那个悬空的起始标签无害地留在推理文本里。曾经试过"由服务器重新发出 stanza 开头"——但重复的开头会被模型读作"已经发起过调用"从而直接结束 turn，所以被否决。

关键细节：检测基于累积文本而非 marker token（故 marker 的分词方式无关），且只有完整的 **stanza** 开头才触发——引号片段或孤立的 `<` 不影响解码。

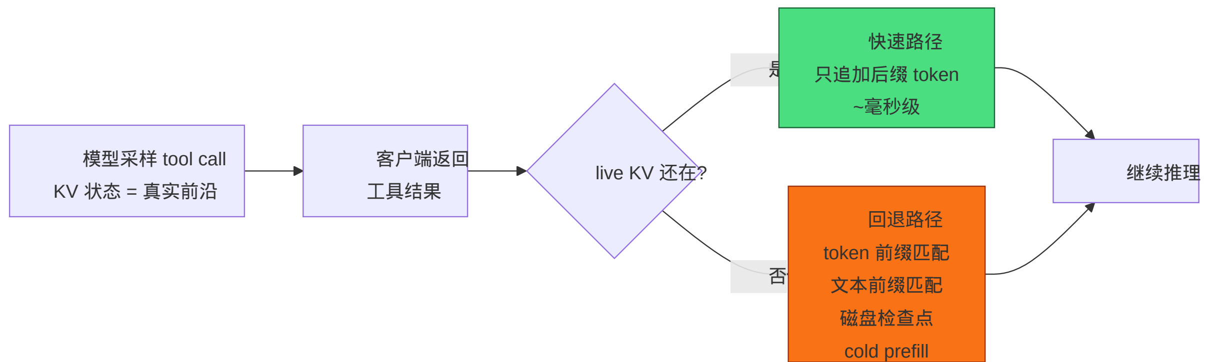
`DS4_SERVER_DISABLE_THINK_TOOL_RECOVERY` 可恢复旧行为。

3e. Live Tool Continuation（工具调用快速续接）

工具调用后保留 live KV 状态，不再重建整个上下文。

问题：之前模型采样完工具调用后，服务器尝试 canonicalize live KV 检查点。百万 token 上下文下，这个重建就是一次近乎完整的 prefill，耗时 10 秒以上。

核心思想：采样的 KV 状态是最高保真度的状态。客户端可见的协议对象只应"选择"这个状态，而不是强迫服务器重建它。



两种协议的绑定方式：OpenAI Responses 通过 `call_id`（`function_call_output`）绑定 `/v1/responses` 端点；Anthropic Messages 通过 `tool_use_id`（`tool_result`）绑定 `/v1/messages` 端点。两种协议共享同一套 live KV 续接逻辑：ID 匹配 → 跳过 prefill → 只 tokenize 后缀。详见 `misc/RESPONSE_API.md` 和 `misc/ANTHROPIC_LIVE_CONTINUATION.md`。

5. ds4-eval 能力评估工具

`ds4-eval` 是内置的回归测试基准（~3340 行，独立可执行文件），不是排行榜跑分工具。它加载真实 GGUF 模型，渲染 DS4 聊天提示，逐 token 采样，然后在终端 TUI 中显示结果并自动评分。

```
./ds4-eval -m ds4flash.gguf --trace /tmp/ds4-eval.txt
```

93 道题的组成（交替排列，由易到难）：

来源	数量	特点
GPQA Diamond	25	研究生级科学，多选题
SuperGPQA	25	广域专业知识，跨域迁移（经审计清理）
AIME 2025	25	竞赛数学，精确答案，零容错
COMPSEC	18	计算机安全与本地化，源自公开 CVE

设计意图：不是追求 93/93 满分，而是回答一个工程问题——在修改 kernel、量化、prompt 渲染、KV cache 或 tool streaming 后，模型是否仍然能解出一个有代表性的难题混合？

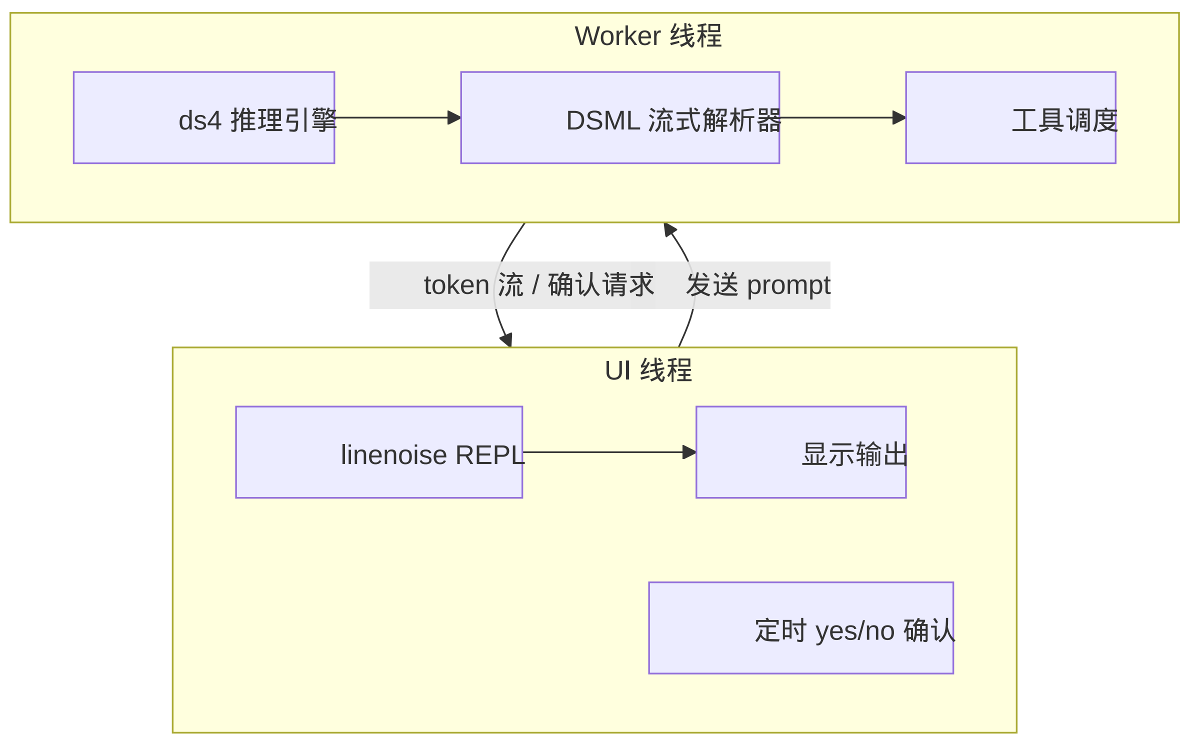
TUI 交互：分屏布局，ANSI 颜色，`p` 暂停、`q` 退出并打印报告、Up/Down 选择题目、Enter 运行选中的题目。`--plain` 禁用 TUI 用于自动化。上下文大小默认自动适配（`--ctx` 可选覆盖），`--tokens 16000`、thinking 模式开启。每个 case 的生成长度会被钳制到剩余上下文预算内，避免 KV 容量溢出导致整个评测中止。

Makefile 中 `make` 默认会编译 `ds4-eval`（和 `ds4`、`ds4-server`、`ds4-bench` 一起）。

4. ds4-agent 自主编码代理

> ds4-agent 是 ds4 项目新增的核心组件——一个完全在本地运行的 AI 编码代理，通过 ds4 推理引擎的 DSML 工具调用协议与文件系统、终端和浏览器交互。

架构



双线程架构：UI 线程处理用户输入和输出显示；Worker 线程独占推理引擎。两者通过共享缓冲区和条件变量通信。

10 个内置工具

工具	功能	需确认
`read`	读取文件	否
`more`	追加读取（分页）	否
`write`	写入文件	是
`list`	列出目录	否
`edit`	搜索替换编辑	是
`search`	grep 搜索	否
`google_search`	Google 搜索	是
`visit_page`	访问网页	是
`bash`	执行 shell 命令	是
`bash_status`	检查后台任务状态	否

写文件、执行命令、访问网页等危险操作需要用户通过定时 yes/no 提示确认（默认 10 秒超时自动拒绝）。

`edit` 工具的锚定匹配与回归测试

`edit`（搜索替换）的核心是旧文本匹配——在文件里定位"要被替换的那段"。锚定语法 `[upto ...]` 让旧文本可以锚定到某个标记为止，处理"同一段文本在文件里多次出现"的歧义。这块匹配逻辑曾有两个边界缺陷：

- **old tail anchor not found after old head**：当旧文本同时包含 head 和 tail 锚点时，head 匹配成功后 tail 锚点找不到，导致整次 edit 失败。
- 修复后，`ds4-agent` 新增了锚定 **`[upto]`** 编辑的单元测试 **harness**，并纳入 `make test` 默认套件——以后旧文本匹配的回归会被默认测试立刻捕获。

这是"编辑工具的健壮性 = agent 可用性"的直接体现：agent 大量依赖 edit 做精确改动，匹配失败会逼它退化到整文件重写。

上下文压缩（**Context Compaction**）

当 token 使用量达到上下文的 85% 时触发软压缩：

```
1. 保留系统提示 + 工具定义（不变）
2. 将对话历史摘要为一段简短文本
3. 保留最近的 N 个 token 逐字（verbatim tail）
4. 重建上下文：系统 + 摘要 + verbatim tail + 最新消息
```

如果软压缩后仍然溢出，触发硬压缩——更激进地截断历史。压缩协议详见 `misc/COMPACT.md`。

会话持久化

agent 会话状态（对话历史、工作目录、压缩标记等）可以保存到 JSON 文件，支持中断后恢复。`--session` 指定会话文件路径。

非交互模式（`--prompt`）直接执行单个 prompt 后退出，适合脚本化使用。`--cwd` 设置工作目录。

ds4_web 浏览器工具

`ds4_web.c` 提供 `google_search` 和 `visit_page` 两个工具，通过 CDP（Chrome DevTools Protocol）WebSocket 连接到本地 Chrome 浏览器（端口 9333）：

```
ds4_agent → tool call: google_search("ds4 inference engine")
          → ds4_web: CDP WebSocket → chrome://devtools
          → 返回搜索结果文本
```

浏览器工具集成 agent 的定时 yes-no 确认机制——每次搜索或页面访问前都需要用户批准。Chrome 需以 `--remote-debugging-port=9333` 启动。

Agent 工具结果 **DSML** 包装

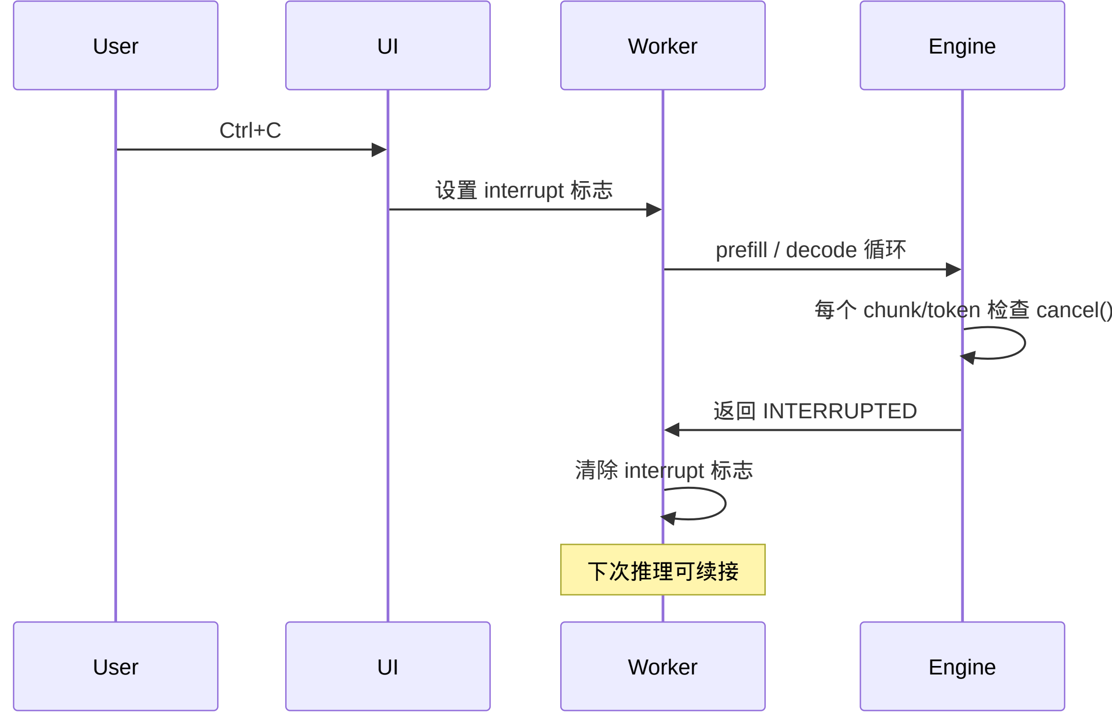
工具执行结果现在以 `<tool_result>...</tool_result>` XML 标签包装，通过 `ds4_chat_append_message()` 以 `"tool"` 角色追加到对话历史。这样工具输出在 DSML 上下文中有明确的边界标记，避免与模型生成的文本混淆：

```
// 工具结果包装
ds4_chat_append_message(w->engine, &w->transcript, "tool", tool_result);
// tool_result 内容: "<tool_result>...</tool_result>"
```

`/history` 命令的显示逻辑会自动检测并剥离这些包装标签（`agent_history_tool_result_payload()`），用户看到的是干净的工具输出。畸形 DSML 调用会触发 `agent_dsml_syntax_reminder` 语法提醒。

合作式中断（**Cooperative Interruption**）

Agent 的 Ctrl+C 从暴力终止改为合作式中断——在安全检查点优雅停止，保留有效的 KV cache 状态：



中断回调 API：

```c
// ds4.h — 中断回调
typedef bool (*ds4_session_cancel_fn)(void *ud);
#define DS4_SESSION_SYNC_INTERRUPTED 2
void ds4_session_set_cancel(ds4_session *s, ds4_session_cancel_fn fn, void *ud);
```

各阶段的中断行为：

- **Prefill**：chunk 边界检查，保留到最后一个有效前缀的 KV 状态
- **Generation**：token 边界检查，推送 EOS 后优雅退出
- **Compaction**：中断时保留压缩前的对话状态

Web 工具同样支持中断——`google_search` 和 `visit_page` 的阻塞操作（CDP WebSocket 连接、页面加载）都可通过 Ctrl+C 立即中断，不再需要等待 30+ 秒超时。

终端 **raw** 模式状态恢复

Agent 的交互式 REPL 基于 linenoise，运行在终端 **raw** 模式（关闭行缓冲、逐字节读取，这样才能拦截 Ctrl+C、做行编辑）。raw 模式是进程级终端状态——退出时若不恢复，shell 提示符会残留 raw 模式（回车不换行、输入不可见）。本轮同步修复了 raw 模式在多个退出路径上的泄漏：

- **quit / help**：在这些非错误退出路径上也要正确恢复终端状态（之前只在正常退出路径恢复，quit/help 漏了）。
- 中断恢复：`fix(agent): restore raw mode` 确保 Ctrl+C 中断后 raw 模式状态一致，下一轮输入仍可正常编辑。

这是合作式中断的"终端侧"配套——engine 侧保留了 KV cache，REPL 侧也得把终端交还给用户一个干净的状态。

DSML 解析增强

流式 DSML 解析器经过多轮加固，处理模型输出的各种边缘情况：

标签校验：`agent_dsml_open_tag_is()` 精确验证 DSML 开标签（标签名后必须跟 `>`、空格、Tab 或换行），替换了之前会匹配子串的 `strncmp`。

文本中的 DSML 检测：`agent_stream_note_plain_dsml_byte()` 检测正常文本中出现的 DSML 标记（模型在思考块或普通文本中误输出 `<｜DSML｜`），触发 `AGENT_DSML_ERROR`。

隐式 Invoke 检测：当模型省略 `<tool_calls>` 外层包装，直接输出 `<｜DSML｜invoke` 时，解析器自动合成规范的 `<tool_calls>` 开标签，捕获模型意图。

参数关闭标签前缀追踪：`param_close_prefix` 字段追踪 `</｜DSML｜parameter...｜>` 关闭标签的部分匹配。这让终端可视化器可以在关闭标签完成前就隐藏部分标签，同时支持 greedy 采样判断。

畸形 DSML 错误报告：`agent_stream_malformed_dsml()` 在终端显示红色 `[invalid tool call: ...]` 错误信息，同时保留 debug trace。

Marker 检测器泛化：`agent_dsml_marker_detector` 结构体泛化了"尾部缓冲"模式，同时检测标准 `｜DSML｜` 和非标准 `DSML｜`（缺少全角竖线）两种格式。

Agent 状态增强

Agent 状态栏新增三项实时信息，让用户了解推理引擎的内部状态：

Greedy 采样指示器 ❄️：当 DSML 解析器检测到模型正在生成结构标签（`<invoke>`、`<parameter>` 名等）时，采样被强制切换为 temperature=0 的确定性模式。状态栏显示 `❄` 标记：

```
ctx 12K/128K | generation 42 tokens ❄ 15.3 t/s 🔋
```

完整格式为 `ctx <已用>/<总量> | generation <N> tokens [❄] <速度> t/s [🔋]`，其中 `❄` 标记 greedy 采样，`🔋` 标记使用电池供电（影响功率策略）。上例为简化展示。

`agent_stream_wants_greedy_sampling()` 根据解析器状态判断当前 token 是否应使用 greedy 采样——标签结构用 greedy 保证可解析性，参数内容用正常温度避免重复。

Prefill 速度显示：进度条内嵌实时 tokens/second 速率：

```
[████████░░░░] 342t/s
```

`prefill_tps` 字段记录 `done / elapsed` 的比值，渲染在进度条的未填充区域。

Web 工具状态消息 ✦：`google_search` 和 `visit_page` 执行时显示粉红色的系统状态消息：

```
✦ Searching Google for "ds4 inference engine"
✦ Opening page https://...
```

`agent_publishf_system_status()` 是可复用的状态发布函数，所有工具都可以使用。

反向链接

/glossary/cooperative-interruption

/glossary/dsml

/glossary/kv-cache

/glossary/sse

端到端推理流程

学习日志

Part 4: 推理流程

出链

Part 4: 推理流程

Part 3: 模型架构

Part 1: 构建与加载

Part 5: 练习

---

1. HTTP 服务器

练习 1：HTTP 请求格式

题目：一个 OpenAI chat 请求的 HTTP 报文长什么样？写出完整的请求。

参考答案

```
POST /v1/chat/completions HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Content-Length: 95

{"model":"deepseek-v4-flash","messages":[{"role":"user","content":"Hell
o"}],"stream":true}
```

ds4_server 用 `sscanf` 解析第一行，用 `Content-Length` 确定请求体长度。

练习 2：SSE 格式

题目：写出两个 SSE 事件，分别发送 `{"text":"Hi"}` 和结束标记。

参考答案

```
data: {"text":"Hi"}

data: [DONE]
```

注意每行 `data:` 前缀，每条消息后有两个换行（一个在内容后，一个空行分隔）。

练习 3：为什么推理只用一个 worker 线程

题目：ds4_server 只有一个推理 worker。为什么不并行处理多个请求？

参考答案

1. **GPU** 独占：Metal GPU 一次只能执行一个计算图，多个请求无法同时使用 GPU
2. **KV cache** 独占：内存中只有一个 live session，多请求会互相覆盖
3. 内存限制：模型本身占 81GB，一个 KV cache 占数 GB，128GB 机器没有空间给多个 session
4. 串行化保证：按先来先服务的顺序处理，避免复杂的并发控制

如果未来支持 batching（多请求合并），可以同时处理多个短请求。

练习 4：poll() 事件追踪

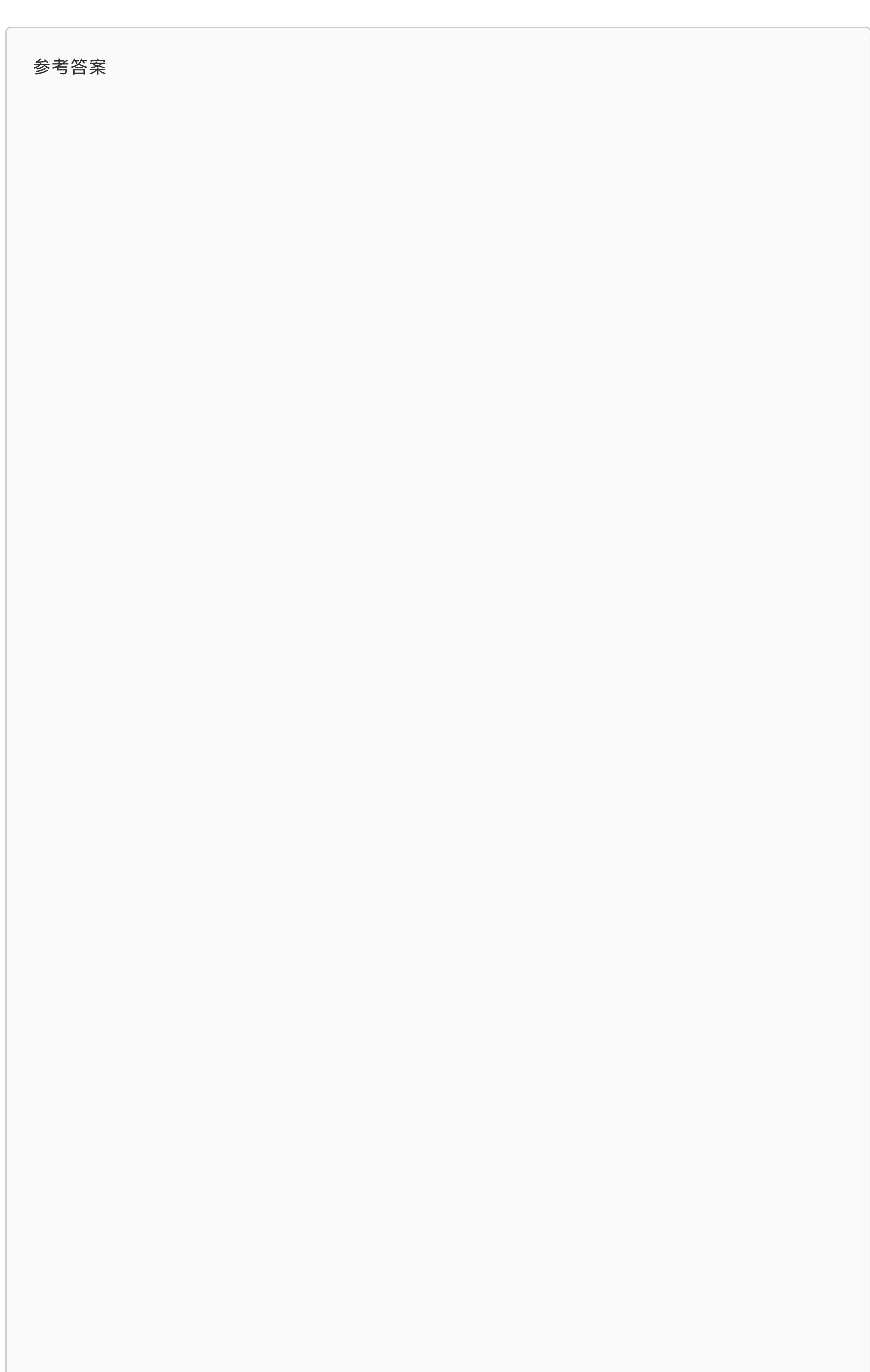
题目：SSE 推理产生 3 个 token "Hi"、"!"、"EOS"。`write_all` 在发送第 2 个 SSE chunk 时，`send()` 返回 -1 且 errno=EAGAIN（TCP 发送缓冲区满）。追踪 `poll()` 调用的事件序列，标出每次 poll 的 `events` 和 `revents` 值。

参考答案

```
初始: TCP 发送缓冲区 = 65536 字节

步骤 1: sse_headers(fd)
  → send() 成功, 写入 HTTP 响应头 ~120 字节
  → 缓冲区剩余 ~65416

步骤 2: sse_chunk(fd, "Hi")
  → send() 成功, 写入 "data: {\"text\":\"Hi\"}\n\n" ~25 字节
  → 缓冲区剩余 ~65391

步骤 3: sse_chunk(fd, "!")
  → send() 尝试写入 ~25 字节
  → 缓冲区满(假设之前的响应数据还没被客户端读走)
  → send() 返回 -1, errno = EAGAIN (would block)

步骤 4: poll() 调用
  struct pollfd pfd = {fd, POLLOUT, 0};
  poll(&pfd, 1, -1);           // 无限等待, 直到 fd 可写

  → 此时:
    pfd.events  = POLLOUT (= 0x0004)  // 关注可写事件
    pfd.revents = 0                    // 还没有事件

步骤 5: 客户端读走数据, TCP 缓冲区释放
  → poll() 返回
    pfd.revents = POLLOUT (= 0x0004)  // fd 现在可写了

步骤 6: 重新 send()
  → send() 成功, 写入剩余的 SSE chunk 数据

步骤 7: sse_done(fd)
  → send() 成功, 写入 "data: [DONE]\n\n"
  → 流结束

时序:
send(headers)  → OK
send(chunk_1)  → OK
send(chunk_2)  → EAGAIN ← 缓冲区满
poll(POLLOUT)  → 等待...
                ← 客户端读走数据
poll 返回       → POLLOUT
send(chunk_2)  → OK (重试成功)
send([DONE])   → OK
```

练习 5：job 队列交错追踪

题目：3 个客户端几乎同时发送请求（fd=5, fd=6, fd=7），1 个 worker 线程。追踪 enqueue/dequeue 的执行顺序，画出队列状态变化。


参考答案
参考答案

```
初始状态: head=NULL, tail=NULL

步骤 1: client_main(fd=5) 调用 enqueue(job_5)
  lock(mu)
  head = job_5, tail = job_5
  signal(cv)      // 唤醒 worker
  unlock(mu)

步骤 2: client_main(fd=6) 调用 enqueue(job_6)
  lock(mu)
  tail->next = job_6  // job_5 → job_6
  tail = job_6
  signal(cv)
  unlock(mu)

步骤 3: client_main(fd=7) 调用 enqueue(job_7)
  lock(mu)
  tail->next = job_7  // job_5 → job_6 → job_7
  tail = job_7
  signal(cv)
  unlock(mu)

步骤 4: worker_main 调用 dequeue()
  lock(mu)
  head = job_5 → 取出 job_5
  head = job_6
  unlock(mu)
  → generate_job(job_5)    // 处理 fd=5 的推理

步骤 5: worker 完成 job_5，再次 dequeue()
  lock(mu)
  head = job_6 → 取出 job_6
  head = job_7
  unlock(mu)
  → generate_job(job_6)

步骤 6: worker 完成 job_6，再次 dequeue()
  取出 job_7
  head = NULL, tail = NULL
  → generate_job(job_7)

队列状态变化:
enqueue(job_5): [job_5]
enqueue(job_6): [job_5, job_6]
enqueue(job_7): [job_5, job_6, job_7]
dequeue:        [job_6, job_7]        → 处理 job_5
dequeue:        [job_7]               → 处理 job_6
dequeue:        []                    → 处理 job_7

关键观察:
```

```
  - enqueue 的顺序决定了处理顺序（FIFO）
  - signal(cv) 可能合并（多个 signal 只唤醒一次）
  - 但 while(!head) 确保不会丢失任务
```

今日学习检查清单

- [ ] 理解 socket/bind/listen/accept/recv/send 的流程
- [ ] 能描述 ds4_server 的线程架构
- [ ] 理解 poll() 处理慢客户端的原理
- [ ] 能手写 SSE 事件格式
- [ ] 理解 OpenAI 和 Anthropic API 的区别

延伸挑战

挑战 1（中级）：用 **curl** 测试 **ds4-server**

启动 `./ds4-server`，用 `curl` 发送一个非流式请求和一个流式请求（`"stream": true`）。对比两次响应的格式差异。如何用 `jq` 解析非流式响应中的 assistant 回复？

挑战 2（高级）：实现 **/v1/models** 端点

ds4_server.c 目前没有实现 `/v1/models` 端点（很多客户端会调用它）。阅读 `client_main` 的路由逻辑，添加一个处理函数返回 `{"data": [{"id": "deepseek-v4-flash", "object": "model"}]}`。提示：找到处理 `POST /v1/chat/completions` 的地方，在前面加一个路径匹配。

---

2. KV Cache 持久化

练习 1：SHA1 缓存 key

题目：token IDs = [100, 200, 300]。计算 SHA1 的输入字节（不需要算完整 SHA1）。

参考答案

```
token 100: le_put32 → [0x64, 0x00, 0x00, 0x00]  (100 = 0x64)
token 200: le_put32 → [0xC8, 0x00, 0x00, 0x00]  (200 = 0xC8)
token 300: le_put32 → [0x2C, 0x01, 0x00, 0x00]  (300 = 0x012C)

SHA1 输入: 64 00 00 00 C8 00 00 00 2C 01 00 00 (12 字节)
SHA1 输出: 20 字节 → 40 字符 hex 文件名 + ".kv"
```

练习 2：原子写入

题目：为什么用 `rename(tmp, final)` 而不是直接 `fwrite` 到最终路径？

参考答案

1. 崩溃安全：如果写一半进程崩溃，临时文件不完整，最终文件完好
2. 原子性：`rename` 在 POSIX 上是原子的——要么成功（新文件替换旧文件），要么失败（旧文件不变）
3. 无半写状态：读取方要么看到完整的旧文件，要么看到完整的新文件，不会看到写了一半的数据

练习 3：前缀匹配效率

题目：缓存目录有 100 个文件，incoming prompt 有 5000 个 token。最坏情况下需要计算多少次 SHA1？

参考答案

```
最坏情况：每个缓存文件的 token 数都 <= 5000

对于每个缓存文件:
  计算该文件 token 数量的 prompt 前缀的 SHA1 → 1 次 SHA1

最多 100 次 SHA1 计算

优化: 先按 token 数量排序，从最长开始匹配
→ 找到第一个匹配就停止
```

练习 4：二进制头部布局计算

题目：给定以下参数，手动构造 KV cache 文件的二进制头部：

- quant_bits = 4, reason = 2 (continued), ext_flags = 0
- tokens = 5000, hits = 3, ctx_size = 8192
- created_at = 1700000000, last_used = 1700000300
- 文本 "Hello, world!" 长度 13 字节

要求：

1. 根据 code-walkthrough 中的布局表，写出头部各字段的十六进制字节值
2. 计算文本段的起始偏移（固定头部 = 48 字节）
3. 写出 text_len 前缀的字节值（4 字节 LE）
4. 计算整个文件的前两部分（header + text）共多少字节

参考答案

```
1. 头部字节值 (48 字节):

偏移   字节值                     字段
0-2   44 53 34                  magic "DS4"
3     01                        version = 1
4     04                        quant_bits = 4
5     02                        reason = 2 (continued)
6     00                        ext_flags = 0
7-10  88 13 00 00               tokens = 5000 (0x1388) LE
11-14 03 00 00 00               hits = 3 LE
15-18 00 20 00 00               ctx_size = 8192 (0x2000) LE
19-26 00 F1 53 65 00 00 00 00 created_at = 1700000000 LE
27-34 2C F2 53 65 00 00 00 00 last_used = 1700000300 LE
35-42 00 00 00 00 00 00 00 00 payload_bytes = 0 (先算文件大小再填)
43-47 00 00 00 00 00            预留/对齐填充到 48 字节

2. 文本段起始偏移: 48

3. text_len 前缀: 0D 00 00 00 (13 = 0x0D, 4 字节 LE)

4. 文件前两部分:
   固定头部: 48 字节
   text_len 前缀: 4 字节
   文本内容: 13 字节
   合计: 48 + 4 + 13 = 65 字节

注: payload_bytes = 文件总大小 - 48 (header) - 4 (text_len) - 13 (text)
   = session 序列化数据的大小
```

练习 5：前缀匹配追踪

题目：磁盘缓存中有 4 个条目：

文件	text	tokens	quant_bits
a1...	"You are a helpful"	4	4
b2...	"You are a helpful assistant"	5	4
c3...	"Write a poem"	3	4
d4...	"You are"	2	4

新请求: prompt_text = "You are a helpful assistant. Tell me about AI"，quant_bits=4, ctx_size=8192, min_tokens=2

追踪 `kv_cache_find_text_prefix` 的执行过程，记录每个条目的匹配结果。

参考答案

```
扫描每个条目:

条目 a1 "You are a helpful" (4 tokens, quant=4):
  - tokens >= min_tokens? 4 >= 2 ✓
  - quant_bits match? 4 == 4 ✓
  - strncmp("You are a helpful assistant. Tell me about AI",
            "You are a helpful", 18) == 0 ✓
  - 匹配长度 = 18
  - best_len = 18, best_path = "a1..."

条目 b2 "You are a helpful assistant" (5 tokens, quant=4):
  - tokens >= min_tokens? 5 >= 2 ✓
  - quant_bits match? 4 == 4 ✓
  - strncmp("You are a helpful assistant. Tell me about AI",
            "You are a helpful assistant", 27) == 0 ✓
  - 匹配长度 = 27 > best_len=18 → 更新
  - best_len = 27, best_path = "b2..."

条目 c3 "Write a poem" (3 tokens, quant=4):
  - tokens >= min_tokens? 3 >= 2 ✓
  - quant_bits match? 4 == 4 ✓
  - strncmp("You are a helpful assistant. Tell me about AI",
            "Write a poem", 12) ≠ 0 → 不匹配

条目 d4 "You are" (2 tokens, quant=4):
  - tokens >= min_tokens? 2 >= 2 ✓
  - quant_bits match? 4 == 4 ✓
  - strncmp("You are a helpful assistant. Tell me about AI",
            "You are", 7) == 0 ✓
  - 匹配长度 = 7 < best_len=27 → 不更新

最终结果:
  best_len = 27
  匹配缓存: b2... ("You are a helpful assistant")
  effective_prompt = ". Tell me about AI"

加载效果:
  - 跳过 5 个 token 的 prefill（直接从磁盘恢复 KV cache）
  - 只需要增量推理 ". Tell me about AI" 部分
```

今日学习检查清单

- [ ] 理解原子写入模式（tmp + rename）
- [ ] 能解释为什么用 token IDs 而不是文本做缓存 key
- [ ] 能画出 KV cache 文件的格式布局
- [ ] 理解四种保存触发时机（cold/continued/evict/shutdown）
- [ ] 理解前缀匹配的工作方式
- [ ] 理解冷启动时修剪尾部 token 的原因

延伸挑战

挑战 **1**（中级）：**KV Cache** 文件逆向分析

运行 `./ds4-server` 进行多轮对话，然后在 `.cache/ds4/` 目录找到生成的 KVC 文件。用 `xxd` 查看头部，对照代码中的 `KVC_MAGIC` 和文件格式，解析出：token 数量、前缀长度、session payload 大小。

挑战 **2**（高级）：前缀匹配的最长公共前缀算法

KV Cache 查找需要计算新旧 token 序列的最长公共前缀。ds4.c 用简单的逐 token 比对。分析其时间复杂度，并提出优化方案：如果缓存了几百个 session，如何快速找到 LCP 最长的那个？提示：考虑 Trie 或哈希。

---

3. API 兼容层

练习 **1**：**JSON** 解析追踪

题目：追踪 `json_skip_value` 对以下 JSON 的解析过程：

```
{"name":"test","value":42}
```

从 `{` 开始，描述如何跳过整个对象。

参考答案

```
'{': 进入对象模式
  "name": 字符串 → json_string → 跳过
  ':': 跳过
  "test": 字符串 → json_string → 跳过
  ',': 跳过
  "value": 字符串 → json_string → 跳过
  ':': 跳过
  42: 数字 → json_number → 跳过
'}': 对象结束

指针停在 '}' 之后
```

追问：如果 JSON 是 `[[[[...256 层...]]]]` 会怎样?

`json_skip_value_depth` 在第 256 层递归时检测到 `depth >= JSON_MAX_NESTING` ，返回 false，请求被拒绝。这防止了恶意深层嵌套 JSON 耗尽 C 调用栈的 DoS 攻击。

练习 2：API 格式对比

题目：同一个请求，写出 OpenAI 和 Anthropic 两种格式的 JSON body。

请求：系统提示 "Be brief"，用户消息 "What is 2+2?"，temperature=0，流式。

参考答案

OpenAI:

```
{
  "model": "deepseek-v4-flash",
  "messages": [
    {"role": "system", "content": "Be brief"},
    {"role": "user", "content": "What is 2+2?"}
  ],
  "temperature": 0,
  "stream": true
}
```

Anthropic:

```
{
  "model": "deepseek-v4-flash",
  "system": "Be brief",
  "messages": [
    {"role": "user", "content": "What is 2+2?"}
  ],
  "temperature": 0,
  "stream": true
}
```

主要区别：system 在 OpenAI 中是 messages 的一项，在 Anthropic 中是顶层字段。

练习 3：为什么手写 JSON 解析器

题目：列出 ds4_server 手写 JSON 解析器而非使用第三方库的三个理由。

参考答案

1. 零依赖：整个项目（ds4 + ds4-server）不需要安装任何库，`make` 就能编译
2. 精确控制：只需要解析请求中的特定字段，`json_skip_value` 可以高效跳过不关心的字段
3. 二进制大小：手写解析器编译后很小，不会引入整个 JSON 库的代码
4. 学习价值：展示了递归下降解析器的设计模式
5. 安全性：自己控制的代码比第三方库的攻击面小

今日学习检查清单

- [ ] 能解释递归下降 JSON 解析器的工作方式
- [ ] 理解 json_skip_value 的作用
- [ ] 能对比 OpenAI 和 Anthropic API 的格式差异
- [ ] 理解 Tool Calling 的完整流程（DSML ↔ OpenAI/Anthropic）
- [ ] 理解工具调用文本记忆（rax 基数树精确回放）
- [ ] 理解 DSML 参数文本和工具结果文本的转义规则（各自只保护关闭标签）
- [ ] 理解 KV cache 命中率衰减驱逐策略（6 小时半衰期）
- [ ] 理解 API 响应中 KV cache 用量报告（cached_tokens / cache_read_input_tokens）
- [ ] 理解工具 schema 前置到 system prompt 的原因（KV cache 边界裁剪保护）
- [ ] 理解检查点恢复机制（session rewrite API）
- [ ] 理解 Thinking 模式的三种级别
- [ ] 理解 SSE 流式输出的格式
- [ ] 理解 JSON 解析的嵌套深度限制（防止栈溢出 DoS）

延伸挑战

挑战 1（中级）：用 Claude Code 调用 ds4-server

配置 Claude Code 使用本地 ds4-server 作为后端（设置 `OPENAI_API_BASE`）。尝试一个多轮对话和一次 tool calling。观察 ds4-server 的日志输出，追踪请求格式是否与 API 兼容层学到的 JSON 解析路径一致。

挑战 **2**（高级）：扩展 **JSON** 解析器

ds4.c 的 JSON 解析器只支持 ds4-server 需要的字段。如果需要支持 `logprobs: true` 参数（返回每个 token 的 top log probabilities），需要修改哪些解析函数？画出从 HTTP 请求到 logprobs 输出的数据流图。

Part 5: 代码走读

---

1. HTTP 服务器

追踪 **1**：动态缓冲区与 **HTTP** 响应

buf 家族（动态字符串缓冲区）

```c
// `buf` 家族（约 109-155 行）: 自增长堆缓冲区

// 扩容（2 倍几何增长）
static void buf_reserve(buf *b, size_t add) {
    while (b->len + add + 1 > b->cap) {
        b->cap = b->cap ? b->cap * 2 : 256;   // 初始 256，之后翻倍
        b->data = realloc(b->data, b->cap);
    }
}

// 追加数据
static void buf_append(buf *b, const void *p, size_t n) {
    buf_reserve(b, n);
    memcpy(b->data + b->len, p, n);
    b->len += n;
    b->data[b->len] = '\0';                   // 始终以 null 结尾
}

// 格式化追加（两次 va_copy: 先测量长度，再写入）
static void buf_printf(buf *b, const char *fmt, ...) {
    va_list ap, ap2;
    va_start(ap, fmt);
    va_copy(ap2, ap);
    int n = vsnprintf(NULL, 0, fmt, ap);       // 测量长度
    buf_reserve(b, n);
    vsnprintf(b->data + b->len, n + 1, fmt, ap2);  // 实际写入
    b->len += n;
    va_end(ap); va_end(ap2);
}

// 所有权转移（调用者负责 free）
static char *buf_take(buf *b) {
    char *data = b->data;
    memset(b, 0, sizeof(*b));                  // 重置为空状态
    return data;                               // 调用者 free()
}
```

HTTP 响应

```c
// `http_response()` 中（约 3080 行）: 发送完整 HTTP/1.1 响应
static bool http_response(int fd, int code, const char *type,
                          const char *body) {
    const char *status = (code == 200) ? "OK"
                       : (code == 400) ? "Bad Request"
                       : (code == 404) ? "Not Found"
                       : (code == 429) ? "Too Many Requests"
                       : "Internal Server Error";
    buf_printf(&b, "HTTP/1.1 %d %s\r\n", code, status);
    buf_printf(&b, "Content-Type: %s\r\n", type);
    buf_printf(&b, "Content-Length: %zu\r\n", strlen(body));
    buf_puts(&b, "Connection: close\r\n\r\n");
    buf_puts(&b, body);
    write_all(fd, b.data, b.len);
}

// `http_error()` 中（约 3097 行）: JSON 错误响应
static bool http_error(int fd, int code, const char *msg) {
    // {"error":{"message":"...","type":"invalid_request_error"}}
    buf_printf(&b, "{\"error\":{\"message\":");
    json_encode_string(&b, msg);
    buf_puts(&b, ",\"type\":\"invalid_request_error\"}}");
    http_response(fd, code, "application/json", b.data);
}
```

追踪 1b：Context Length 错误处理（协议标准化）

参数名自适应

```c
// 根据请求类型返回不同的错误参数名
static const char *context_length_error_param(const request *r) {
    if (!r) return "prompt";
    if (r->api == API_RESPONSES) return "input";
    return r->kind == REQ_COMPLETION ? "prompt" : "messages";
}
```

上下文越界检测

```c
// 检查 prompt tokens 是否 >= context size
// ds4_session_sync() 拒绝 prompt->len >= ctx_size，因为生成至少需要一个空闲 slot
static bool request_exceeds_context(const request *r, int ctx_size) {
    return r && r->prompt.len >= ctx_size;
}
```

协议感知错误响应

```c
// 根据请求的 API 类型（OpenAI vs Anthropic）发送不同格式的错误
static bool http_error_context_length_exceeded(int fd, const request *r,
                                               int n_prompt_tokens,
                                               int ctx_size) {
    char msg[160];
    snprintf(msg, sizeof(msg),
             "Prompt has %d tokens, but the configured context size is %d token
s",
             n_prompt_tokens, ctx_size);

    if (r && r->api == API_ANTHROPIC) {
        // Anthropic 格式: {"type":"error","error":{...}}
        buf_puts(&b, "{\"type\":\"error\",\"error\":{\"type\":\"invalid_request_
error\",...");
    } else {
        // OpenAI 格式: {"error":{"message":...,"code":"context_length_exceede
d"}}
        buf_puts(&b, "{\"error\":{\"message\":...,\"code\":\"context_length_exce
eded\",...");
        // 包含 "param" 字段（messages/input/prompt）
        json_escape(&b, context_length_error_param(r));
    }
    // 两种格式都包含 n_prompt_tokens 和 n_ctx
    http_response(fd, 400, "application/json", b.ptr);
}
```

client_main 中的集成（提前拦截）

```c
// 在 client_main() 中，请求解析后、入队前执行检查
static void *client_main(void *arg) {
    // ... 读取 HTTP 请求 ...
    // ... parse_chat_request() / parse_anthropic_request() ...

    // 提前拦截 context length 超限
    if (request_exceeds_context(&req, ctx_size)) {
        http_error_context_length_exceeded(fd, &req, req.prompt.len, ctx_size);
        request_free(&req);
        goto done;       // 直接关闭连接，不浪费 worker 时间
    }

    // ... enqueue(job) → worker 处理推理 ...
}
```

```
请求处理管线（新增早期检查）：

HTTP 请求到达
    │
    ▼ 读取请求
    │
    ▼ 解析 JSON（parse_chat_request / parse_anthropic_request）
    │
    ▼ ★ context length 检查（新增）
    │   └── 超限 → 400 错误，连接关闭
    │
    ▼ enqueue(job)
    │
    ▼ worker 取出 → generate_job → 推理 + SSE
```

追踪 2：SSE 流式推送

Server-Sent Events 协议

```c
// `sse_headers()` 中（约 3110 行）: 建立 SSE 连接
static bool sse_headers(int fd) {
    const char *hdr =
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/event-stream\r\n"
        "Cache-Control: no-cache\r\n"
        "Connection: close\r\n\r\n";
    return write_all(fd, hdr, strlen(hdr));
}

// `sse_chunk()` 中（约 3119 行）: 发送一个 SSE 数据块
static bool sse_chunk(int fd, const request *r,
                       const char *id, const char *text,
                       const char *finish) {
    buf_puts(&b, "data: ");                    // SSE 格式前缀
    // 根据 API 类型格式化
    if (r->kind == REQ_CHAT) {
        // OpenAI chat completion chunk:
        // {"id":"...","object":"chat.completion.chunk",
        //  "choices":[{"delta":{"content":"text"}}]}
        buf_printf(&b, "{\"id\":\"%s\",\"object\":\"chat.completion.chunk
\",...", id);
    } else {
        // Text completion chunk 格式
    }
    buf_puts(&b, "\n\n");                      // SSE 消息结束标记
    write_all(fd, b.data, b.len);
}

// `sse_done()` 中（约 3174 行）: 流结束
static bool sse_done(int fd, const request *r,
                      const char *id, int prompt_tokens,
                      int completion_tokens) {
    if (r->stream_include_usage)
        sse_usage_chunk(fd, r, id, ...);       // 发送 token 使用统计
    write_all(fd, "data: [DONE]\n\n", 14);     // 终止标记
}
```

```
SSE 流示例:

→ sse_headers(fd)
← HTTP/1.1 200 OK
← Content-Type: text/event-stream
←
← data: {"id":"chatcmpl-123","choices":[{"delta":{"content":"Hello"}}]}
←
← data: {"id":"chatcmpl-123","choices":[{"delta":{"content":" world"}}]}
←
← data: {"id":"chatcmpl-123","choices":[{"delta":{},"finish_reason":"stop"}]}
←
← data: [DONE]
←
```

追踪 **3**：**JSON** 解析器（手写递归下降）

核心解析函数

```c
// `json_ws()` 中（约 160 行）：跳过空白
static void json_ws(const char **p) {
    while (**p == ' ' || **p == '\t' || **p == '\n' || **p == '\r')
        (*p)++;
}

// `json_string()` 中（约 209 行）：解析 JSON 字符串（处理所有转义序列）
static bool json_string(const char **p, char **out) {
    if (**p != '"') return false;
    (*p)++;                                   // 跳过开头引号
    buf b = {0};
    while (**p != '"') {
        if (**p == '\\') {                    // 转义序列
            (*p)++;
            switch (**p) {
                case '"':  buf_putc(&b, '"');  break;
                case '\\': buf_putc(&b, '\\'); break;
                case '/':  buf_putc(&b, '/');  break;
                case 'n':  buf_putc(&b, '\n'); break;
                case 't':  buf_putc(&b, '\t'); break;
                case 'u':  // \uXXXX Unicode 转义
                    uint32_t cp = parse_hex4(*p);
                    // 处理 UTF-16 代理对
                    if (cp >= 0xd800 && cp <= 0xdbff) {
                        uint32_t cp2 = parse_hex4(*p + 2);  // \uXXXX
                        cp = 0x10000 + ((cp - 0xd800) << 10) + (cp2 - 0xdc00);
                    }
                    utf8_put(&b, cp);
                    break;
            }
        } else {
            buf_putc(&b, **p);                // 普通字符
        }
        (*p)++;
    }
    (*p)++;                                   // 跳过结尾引号
    *out = buf_take(&b);                      // 返回堆分配字符串
    return true;
}

// `json_number()` 中（约 253 行）：解析 JSON 数字
static bool json_number(const char **p, double *out) {
    char *end;
    *out = strtod(*p, &end);                  // 标准库转换
    if (end == *p) return false;
    *p = end;
    return true;
}
```

JSON 解析模式

```
递归下降解析器结构:

json_object:
  '{' json_ws ( json_string ':' json_value (',' json_string ':' json_value)* )?
'}'

json_array:
  '[' json_ws ( json_value (',' json_value)* )? ']'

json_value:
  json_string | json_number | json_object | json_array |
  "true" | "false" | "null"

ds4_server 只实现了项目实际用到的子集（字符串、数字、跳过值），
不追求完整的 JSON 规范兼容
```

追踪 4：服务器架构 — Job 队列

server 结构体

```
// `server` 结构体定义（约 4717 行）: 服务器状态
struct server {
    ds4_engine  *engine;            // 推理引擎（只读共享）
    ds4_session *session;           // 推理会话（互斥使用）
    int default_tokens;             // 默认生成 token 数
    kv_disk_cache kv;               // KV 磁盘缓存
    tool_memory tool_mem;           // 工具调用内存
    bool disable_exact_dsml_tool_replay;

    pthread_mutex_t tool_mu;        // 工具调用锁
    pthread_mutex_t mu;             // job 队列锁
    pthread_cond_t  cv;             // job 队列条件变量
    pthread_cond_t  clients_cv;     // 客户端等待条件
    job *head;                      // 队列头
    job *tail;                      // 队列尾
    bool stopping;                  // 停止标志
};
```

生产者-消费者模式

```c
// `enqueue()` 中（约 7482 行）：入队（client_main 调用）
static bool enqueue(server *s, job *j) {
    pthread_mutex_lock(&s->mu);
    if (s->stopping) { pthread_mutex_unlock(&s->mu); return false; }
    j->next = NULL;
    if (s->tail) s->tail->next = j;    // 尾部追加
    else         s->head = j;           // 队列为空
    s->tail = j;
    pthread_cond_signal(&s->cv);        // 通知 worker
    pthread_mutex_unlock(&s->mu);
    return true;
}

// `dequeue()` 中（约 7495 行）：出队（worker_main 调用）
static job *dequeue(server *s) {
    pthread_mutex_lock(&s->mu);
    while (!s->head && !s->stopping)
        pthread_cond_wait(&s->cv, &s->mu);  // 等待新 job
    if (s->stopping && !s->head) {
        pthread_mutex_unlock(&s->mu);
        return NULL;                         // 服务器关闭
    }
    job *j = s->head;
    s->head = j->next;
    if (!s->head) s->tail = NULL;            // 队列变空
    pthread_mutex_unlock(&s->mu);
    return j;
}

// `worker_main()` 中（约 7510 行）：Worker 主循环
static void *worker_main(void *arg) {
    server *s = arg;
    for (;;) {
        job *j = dequeue(s);
        if (!j) return NULL;                 // 关闭信号
        generate_job(s, j);                  // 执行推理
        j->done = true;
        pthread_cond_signal(&j->cv);         // 通知 client_main
    }
}
```

```
架构图:

client_main(fd=5)  client_main(fd=6)  client_main(fd=7)
    │ enqueue           │ enqueue           │ enqueue
    ▼                   ▼                   ▼
  ┌───────────────────────────────────────────────┐
  │          Job Queue (FIFO 链表)                 │
  │   head → [job_5] → [job_6] → [job_7] ← tail │
  └───────────────────────────────────────────────┘
                      │ dequeue
                      ▼
                 worker_main
                      │
                 generate_job(s, job)
                      │
                 ds4_session_sync + sample loop
                      │
                 SSE 流式推送 → fd → 网络 → 用户
```

追踪 5：连接处理与 accept 循环

client_main

```c
// `client_main()` 中（约 7678 行）: 处理单个客户端连接
static void *client_main(void *arg) {
    client_arg ca = *(client_arg *)arg;
    free(arg);
    server *s = ca.s;
    int fd = ca.fd;

    // 读取 HTTP 请求
    char *request = read_http_request(fd);

    // 路由
    if (GET "/v1/models")
        send_models(fd, s->engine);
    else if (GET "/v1/models/deepseek-v4-flash")
        send_model(fd, s->engine);
    else if (POST "/v1/chat/completions" || POST "/v1/completions")
        handle_chat(fd, s, request);
    else if (POST "/v1/messages")
        handle_anthropic(fd, s, request);
    else
        http_error(fd, 404, "Not found");

    close(fd);
    return NULL;
}
```

main 函数

```c
// `main()` 中（约 8089 行）：服务器入口
int main(int argc, char **argv) {
    signal(SIGPIPE, SIG_IGN);                   // 忽略 SIGPIPE（断开的 write）
    // 安装优雅退出信号处理
    struct sigaction sa = { .sa_handler = stop_signal_handler };
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);

    parse_options(argc, argv, &opts);
    ds4_engine_open(&engine, &opts);             // 加载模型
    server s = { .engine = engine, ... };
    ds4_threads_init();                          // 启动线程池

    // 创建 worker 线程（推理线程）
    for (int i = 0; i < n_workers; i++)
        pthread_create(&workers[i], NULL, worker_main, &s);

    // accept 循环
    int listen_fd = listen_on(port);
    while (!s.stopping) {
        int fd = accept(listen_fd, NULL, NULL);
        if (fd < 0) continue;
        // 为每个连接创建独立线程
        client_arg *ca = malloc(sizeof(*ca));
        ca->s = &s; ca->fd = fd;
        pthread_t t;
        pthread_create(&t, NULL, client_main, ca);
        pthread_detach(t);                      // 自动回收
    }
}
```

```
完整服务器生命周期:

1. main() 启动
2. 加载模型 (81GB mmap, ~10s)
3. 创建 worker 线程 (等待 job)
4. listen on port 8080
5. accept 循环:
   ├─ 连接到来 → pthread_create(client_main)
   ├─ client_main 读取请求 → enqueue(job)
   ├─ worker dequeue → generate_job
   ├─ generate_job: sync → sample loop → SSE push
   └─ 完成后 close(fd)
6. SIGINT → stop_signal_handler → stopping=true
7. worker 线程退出 → join → 清理
```

2. KV Cache 持久化

追踪 1：SHA1 哈希实现

SHA-1 压缩函数

```
// 行 5192: 标准 SHA-1 压缩（处理一个 64 字节块）
static void sha1_transform(sha1_ctx *c, const uint8_t block[64]) {
    uint32_t w[80];
    // 前 16 个字直接从输入读取
    for (int i = 0; i < 16; i++)
        w[i] = (block[i*4]<<24) | (block[i*4+1]<<16)
             | (block[i*4+2]<<8) | block[i*4+3];
    // 扩展到 80 个字
    for (int i = 16; i < 80; i++)
        w[i] = rol32(w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16], 1);
    //               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  ^^^
    //               四个字的异或                           左旋 1 位

    uint32_t a = c->h[0], b = c->h[1], c_ = c->h[2];
    uint32_t d = c->h[3], e = c->h[4];
    // 注: 局部变量用 c_ 避免与参数 c 冲突

    // 80 轮运算，分 4 段
    for (int i = 0; i < 80; i++) {
        uint32_t f, k;
        if (i < 20)       { f = (b & c_) | (~b & d);  k = 0x5a827999; }
        else if (i < 40)  { f = b ^ c_ ^ d;            k = 0x6ed9eba1; }
        else if (i < 60)  { f = (b & c_) | (b & d) | (c_ & d); k = 0x8f1bbcdc; }
        else               { f = b ^ c_ ^ d;            k = 0xca62c1d6; }

        uint32_t temp = rol32(a, 5) + f + e + k + w[i];
        e = d; d = c_; c_ = rol32(b, 30); b = a; a = temp;
    }
    c->h[0] += a; c->h[1] += b; c->h[2] += c_;
    c->h[3] += d; c->h[4] += e;
}
```

```
SHA-1 处理流程:

输入消息 (任意长度)
    |
    ▼ 填充: 追加 0x80 + 零字节 + 64 位大端长度
    |
    ▼ 分割为 64 字节块
    |
    ├── sha1_transform(block_0) → 更新 h[0..4]
    ├── sha1_transform(block_1) → 更新 h[0..4]
    └── ...
    |
    ▼ 输出: h[0]||h[1]||h[2]||h[3]||h[4] = 20 字节哈希值
```

初始化与最终化

```
// 行 5233: 标准初始值
static void sha1_init(sha1_ctx *c) {
    c->h[0] = 0x67452301;
    c->h[1] = 0xefcdab89;
    c->h[2] = 0x98badcfe;
    c->h[3] = 0x10325476;
    c->h[4] = 0xc3d2e1f0;
    c->len = 0;
}

// 行 5260: 填充 + 最终化
static void sha1_final(sha1_ctx *c, uint8_t out[20]) {
    // 追加 0x80 填充
    // 对齐到 56 mod 64
    // 追加 64 位大端位长度
    // 输出 20 字节哈希值
}
```

追踪 2：缓存条目序列化格式

二进制头部布局

读取头部

```
// 行 5539: 填充固定头部
static void kv_fill_header(uint8_t h[KV_CACHE_FIXED_HEADER],
                            uint8_t quant_bits, uint8_t reason,
                            uint8_t ext_flags,
                            uint32_t tokens, uint32_t hits,
                            uint32_t ctx_size,
                            uint64_t created_at, uint64_t last_used,
                            uint64_t payload_bytes) {
    // 字节 0-2: magic ('D','S','4')
    // 字节 3:   version
    // 字节 4:   quant_bits (2 或 4)
    // 字节 5:   reason (store 触发原因)
    // 字节 6:   ext_flags
    // 字节 7-10:  tokens (LE uint32)
    // 字节 11-14: hits (LE uint32)
    // 字节 15-18: ctx_size (LE uint32)
    // 字节 19-26: created_at (LE uint64)
    // 字节 27-34: last_used (LE uint64)
    // 字节 35-42: payload_bytes (LE uint64)
}
```

头部二进制布局

```
偏移  大小  字段            说明
0     3     magic           "DS4" 魔数
3     1     version         版本号
4     1     quant_bits      量化位数 (2 或 4)
5     1     reason          存储触发原因
6     1     ext_flags       扩展标志
7     4     tokens          token 数量 (LE)
11    4     hits            缓存命中次数 (LE)
15    4     ctx_size        上下文长度 (LE)
19    8     created_at      创建时间戳 (LE)
27    8     last_used       最后使用时间 (LE)
35    8     payload_bytes   负载大小 (LE)
---固定头部结束---

43    var   text_bytes      原始文本
?     var   session_data    序列化的 session 状态
```

读取头部

```c
// 行 5560: 从文件读取并验证头部
static bool kv_read_header(FILE *fp, kv_entry *e, uint32_t *text_bytes) {
    uint8_t h[KV_CACHE_FIXED_HEADER];
    if (fread(h, 1, KV_CACHE_FIXED_HEADER, fp) != KV_CACHE_FIXED_HEADER)
        return false;
    if (h[0] != 'D' || h[1] != 'S' || h[2] != '4') return false;
    // 使用 le_get32 / le_get64 读取小端字段
    e->quant_bits = h[4];
    e->tokens     = le_get32(h + 7);
    e->hits       = le_get32(h + 11);
    // ...
}
```

追踪 3：磁盘缓存打开 / 关闭 / 刷新

缓存初始化

```c
// 行 5723: 打开/创建磁盘缓存
static bool kv_cache_open(kv_disk_cache *kc, const char *dir,
                           uint64_t budget_mb,
                           bool reject_different_quant,
                           kv_cache_options opt) {
    mkdir(dir, 0755);                          // 创建缓存目录
    kc->dir = strdup(dir);
    kc->budget = budget_mb * 1024 * 1024;      // MB → 字节
    kc->enabled = true;
    kc->reject_different_quant = reject_different_quant;
    kv_cache_evict(kc, NULL, NULL);             // 先清理超预算条目（无保护 SHA）
}
```

缓存刷新（扫描磁盘目录）

```c
// 行 5603: 从磁盘重建内存索引
static void kv_cache_refresh(kv_disk_cache *kc) {
    // 清空内存中的条目列表
    for (int i = 0; i < kc->n_entries; i++)
        free(kc->entries[i].text);
    kc->n_entries = 0;

    // 扫描目录
    DIR *d = opendir(kc->dir);
    struct dirent *de;
    while ((de = readdir(d))) {
        // 只处理 40 字符的 SHA-1 hex 文件名
        if (strlen(de->d_name) != 40) continue;
        // 读取文件头部
        kv_entry e;
        kv_read_entry_file(kc, de->d_name, &e);
        // 推入内存数组
        kc->entries[kc->n_entries++] = e;
    }
    closedir(d);
}
```

```
磁盘缓存结构:

kv_cache/
  ├── 2ef7bde608ce5404e97d5f042f95f89f1c232871  ← SHA-1(token_ids) 作为文件名
  │    └── [DS4 头部] + [文本] + [session 序列化数据]
  ├── a3f2c8d...
  └── ...

内存索引:
  entries[n_entries]
    ├── {sha1="2ef7b...", text="Hello world", tokens=42, hits=5, ...}
    ├── {sha1="a3f2c...", text="What is AI",  tokens=38, hits=2, ...}
    └── ...

查找时: SHA-1(新 prompt) 匹配磁盘文件名 → 命中
```

追踪 4：原子写入与存储

原子写入模式

```c
// 行 5921: 将当前 session 的前缀缓存到磁盘
static bool kv_cache_store_live_prefix(server *s,
                                        const ds4_tokens *tokens,
                                        int store_len,
                                        const char *reason) {
    // 前置检查
    if (!kc->enabled) return false;
    if (store_len < min_tokens) return false;
    if (quant_bits != 2 && quant_bits != 4) return false;

    // 关键验证: session checkpoint 必须精确匹配要存储的前缀
    // 这确保不会为未推理的内容创建缓存
    if (s->session->checkpoint.len != store_len) return false;
    for (int i = 0; i < store_len; i++)
        if (s->session->checkpoint.data[i] != tokens->data[i]) return false;

    // 原子写入: 先写临时文件，再 rename
    char tmp[512];
    snprintf(tmp, sizeof(tmp), "%s/%s.tmp.%d", dir, sha1_hex, getpid());
    FILE *fp = fopen(tmp, "wb");

    // 写头部
    kv_fill_header(header, quant_bits, reason, ext_flags, ...);
    fwrite(header, 1, KV_CACHE_FIXED_HEADER, fp);

    // 写文本
    fwrite(text, 1, text_len, fp);

    // 写 session 序列化数据
    ds4_session_save_payload(s->session, fp, ...);

    fclose(fp);

    // 原子重命名（同一文件系统的 rename 是原子的）
    rename(tmp, final_path);
}
```

```
原子写入保证:

1. 创建 /kv_cache/abc123.tmp.12345    ← 临时文件（带 PID）
2. 写入全部数据到临时文件
3. fclose()                             ← 确保数据落盘
4. rename(tmp, abc123)                  ← 原子替换

如果步骤 1-3 中间崩溃:
  → 临时文件不完整，但 abc123 不存在或仍是旧版本
  → 不会损坏已有缓存

如果步骤 4 后崩溃:
  → abc123 已是完整的新缓存，没问题
```

追踪 **5**：前缀匹配与缓存加载

文本前缀匹配

```c
// 行 6068: 查找最长文本前缀匹配
static int kv_cache_find_text_prefix(kv_disk_cache *kc,
                                     const char *prompt_text,
                                     int quant_bits, int ctx_size) {
    kv_cache_refresh(kc);                          // 刷新磁盘索引

    int best_len = 0;
    const char *best_path = NULL;

    for (int i = 0; i < kc->n_entries; i++) {
        kv_entry *e = &kc->entries[i];
        // 过滤条件
        if (e->tokens < kc->min_tokens) continue;
        if (e->ctx_size != ctx_size) continue;
        if (kc->reject_different_quant && e->quant_bits != quant_bits) continue;

        // 检查 prompt_text 是否以 e->text 为前缀
        int tlen = strlen(e->text);
        if (tlen > strlen(prompt_text)) continue;
        if (strncmp(prompt_text, e->text, tlen) == 0) {
            if (tlen > best_len) {
                best_len = tlen;                   // 更长匹配
                best_path = e->path;
            }
        }
    }
    return best_len;                               // 返回匹配长度（0 = 未命中）
}
```

缓存加载入口

```c
// 行 6092: 尝试从磁盘缓存加载
static int kv_cache_try_load_text(server *s, const char *prompt_text,
                                   ds4_tokens *effective_prompt,
                                   char **loaded_path_out) {
    // 1. 查找最长前缀匹配
    int match_len = kv_cache_find_text_prefix(&s->kv, prompt_text, ...);
    if (match_len == 0) return 0;              // 未命中

    // 2. 从磁盘文件恢复 session
    FILE *fp = fopen(matched_path, "rb");
    kv_read_header(fp, &entry, &text_bytes);
    ds4_session_load_payload(s->session, fp, ...);
    fclose(fp);

    // 3. 计算 effective_prompt = 匹配前缀之后的剩余部分
    // 如果 prompt_text = "Hello world, how are you?"
    // 匹配的缓存文本 = "Hello world" (11 字符)
    // 则 effective_prompt = ", how are you?"

    return match_len;
}
```

```
缓存查找流程:

新请求: "Hello world, how are you?"
    │
    ▼ kv_cache_find_text_prefix
    │
    ├── 扫描所有缓存条目
    ├── "Hello" → 匹配 5 字符
    ├── "Hello world" → 匹配 11 字符 ← 最长匹配
    └── "Goodbye" → 不匹配
    │
    ▼ 加载 "Hello world" 对应的 session
    │
    ▼ effective_prompt = ", how are you?"
    │
    ▼ ds4_session_sync(effective_prompt)
      只需增量推理 ", how are you?" 部分
      跳过了 "Hello world" 的完整 prefill

节省: 11 个 token 的 prefill → 直接恢复 KV cache
```

3. API 兼容层

追踪 1：OpenAI 请求解析

Chat Completion 请求

```
// 行 1984: 解析 OpenAI /v1/chat/completions 请求
static bool parse_chat_request(ds4_engine *e, server *s,
                                const char *body, int def_tokens,
                                int ctx_size, request *r,
                                char *err, size_t errlen) {
    r->kind = REQ_CHAT;                        // 标记请求类型
    r->api = API_OPENAI;

    // 解析 JSON body 中的字段
    // "model": "deepseek-v4-flash"
    // "messages": [{role, content}, ...]
    // "temperature": 0.7
    // "top_p": 0.9
    // "max_tokens": 1024
    // "stream": true
    // "tools": [...]
    // "thinking": {type: "enabled", budget_tokens: ...}

    if (parse_messages(&p, &r->msgs) == false)  return false;
    if (parse_tools_value(&p, &r->tools_raw, &r->tool_orders) == false)
        return false;

    r->reasoning_effort = DS4_THINK_HIGH;      // 默认高推理
    r->n_predict = def_tokens;
}
```

Messages 解析

```c
// 行 1244: 解析消息数组
static bool parse_messages(const char **p, chat_msgs *msgs) {
    // 期望格式: [{"role":"user","content":"Hello"}, ...]
    json_ws(p); expect('[', p);
    while (**p != ']') {
        json_ws(p); expect('{', p);
        while (**p != '}') {
            char *key;
            json_string(p, &key);
            if (strcmp(key, "role") == 0) {
                json_string(p, &role);
            } else if (strcmp(key, "content") == 0) {
                json_string(p, &content);
            }
            free(key);
            if (**p == ',') (*p)++;
        }
        expect('}', p);
        // 追加到 msgs
        msgs->role[msgs->n] = role;
        msgs->content[msgs->n] = content;
        msgs->n++;
        if (**p == ',') (*p)++;
    }
    expect(']', p);
}
```

Tools 解析

```c
// 行 1211: 解析 OpenAI tools 数组
static bool parse_tools_value(const char **p, char **out,
                              tool_schema_orders *orders) {
    // 每个工具提取 function schema:
    // {"type":"function","function":{"name":"get_weather",
    //  "parameters":{"type":"object",...}}}
    // → 序列化为 JSON 行格式，便于后续 DSML 解析
}
```

追踪 2：Anthropic 请求解析

Messages API

```c
// 行 2154: 解析 Anthropic /v1/messages 请求
static bool parse_anthropic_request(ds4_engine *e, server *s,
                                     const char *body, int def_tokens,
                                     int ctx_size, request *r,
                                     char *err, size_t errlen) {
    r->kind = REQ_CHAT;
    r->api = API_ANTHROPIC;                    // 标记 Anthropic API
    // Anthropic 特有字段:
    // "system": "You are helpful"       ← 系统提示在顶层，不在 messages 中
    // "max_tokens": 1024               ← 必填字段
    // "thinking": {type: "enabled", budget_tokens: ...}

    // 解析 messages（Anthropic 格式）
    parse_anthropic_messages(&p, &r->msgs);
}
```

Anthropic Messages 解析

```c
// 行 1494: 解析 Anthropic 消息格式
static bool parse_anthropic_messages(const char **p, chat_msgs *msgs) {
    // Anthropic 消息格式与 OpenAI 不同:
    // content 可以是字符串或数组:
    //   "content": "Hello"                                    ← 简单字符串
    //   "content": [{"type":"text","text":"Hello"},{"type":"image"...}]  ← 内容
块数组

    for each message object:
        parse "role" and "content"
        if content is array:
            for each block:
                if type == "text": extract text
                if type == "tool_use": extract tool info
                if type == "tool_result": extract result
}
```

```
OpenAI vs Anthropic 格式对比:

OpenAI:
  {"messages":[{"role":"user","content":"Hello"}],
   "tools":[{"type":"function","function":{...}}]}

Anthropic:
  {"messages":[{"role":"user","content":"Hello"}],
   "system":"You are helpful",
   "tools":[{"name":"get_weather","input_schema":{...}}]}

关键差异:
  - system 提示: OpenAI 在 messages 中，Anthropic 在顶层
  - tools 格式: OpenAI 嵌套 function，Anthropic 扁平 name + input_schema
  - content 块: Anthropic 支持 image、tool_use、tool_result 类型
```

追踪 3：DSML 工具调用解析状态机

参考实现（测试用）

```
// 行 3556: 完整 DSML 解析（慢速参考版本）
static dsml_decode_state dsml_decode_state_for_text(
        const char *raw, size_t raw_len) {
    // 搜索 DSML 工具起始标记
    const char *start = dsml_find_tool_start(raw, raw_len);
    if (!start) return DSML_DECODE_TEXT;

    // 遍历 invoke_start / invoke_end 标记
    // 判断当前文本是在 DSML 结构中还是载荷内容中
    // 返回: DSML_DECODE_TEXT / DSML_DECODE_STRUCTURAL / DSML_DECODE_PAYLOAD
}
```

流式状态机

```c
// 行 3646: 增量流式 DSML 追踪器
static void dsml_decode_tracker_update(
        dsml_decode_tracker *dt,
        const char *raw, size_t raw_len) {

    // 三种状态:
    // DSML_TRACK_SEARCH:       扫描工具调用起始标记
    // DSML_TRACK_STRUCTURAL:   在标记边界中
    // DSML_TRACK_DONE:         工具调用块已完成

    // 关键: 保留 hold-back 缓冲区
    // 防止工具标记被 chunk 边界截断
    size_t holdback = dsml_max_tool_start_len();
    // 如果最后 holdback 字节可能是不完整标记，暂不发送
}
```

```
流式 DSML 追踪时序:

Chunk 1: "The answer is <dsml:invoke"     ← 可能是不完整标记
  → hold back "<dsml:invoke"              ← 不发送，等下一 chunk

Chunk 2: '_start tool="weather">'          ← 与 holdback 拼接
  → 完整标记: "<dsml:invoke_start tool="weather">"
  → 状态切换到 STRUCTURAL
  → 发送已确认的文本

Chunk 3: '{"city":"SF"}<dsml:invoke_end>'
  → 检测到 invoke_end
  → 状态切换到 DONE
  → 完整的工具调用记录
```

追踪 4：SSE 流式响应格式化

OpenAI 流式格式

```c
// 行 4042: OpenAI 格式 SSE 更新
static bool openai_sse_stream_update(
        int fd, server *s, const request *r,
        const char *id, openai_stream *st,
        const char *raw, size_t raw_len, bool final) {

    // 思考模式检测
    if (st->state == OPENAI_STREAM_THINKING) {
        // 等待思考前缀确认后才发送
        // 前缀字符是特殊的 Unicode 码点
        if (raw[0] == think_marker) {
            st->state = OPENAI_STREAM_THINKING_CONFIRMED;
        } else {
            st->state = OPENAI_STREAM_CONTENT;
        }
    }

    // 格式化 SSE 数据行
    buf_printf(&b, "data: {\"id\":\"%s\",", id);
    buf_printf(&b, "\"object\":\"chat.completion.chunk\",");
    buf_printf(&b, "\"choices\":[{\"index\":0,");
    buf_printf(&b, "\"delta\":{\"content\":");
    json_encode_string(&b, text);
    buf_printf(&b, "},\"finish_reason\":%s}]}\n\n",
           final ? "\"stop\"" : "null");
    write_all(fd, b.data, b.len);
}
```

Anthropic 流式格式

```
Anthropic SSE 事件序列:

event: message_start
data: {"type":"message_start","message":{"id":"msg_123",...}}

event: content_block_start
data: {"type":"content_block_start","index":0,"content_block":{"type":"thinkin
g"}}

event: content_block_delta
data: {"type":"content_block_delta","delta":{"type":"thinking_delta","thinkin
g":"Let me..."}}

event: content_block_stop
data: {"type":"content_block_stop","index":0}

event: content_block_start
data: {"type":"content_block_start","index":1,"content_block":{"type":"text"}}

event: content_block_delta
data: {"type":"content_block_delta","delta":{"type":"text_delta","text":"Hell
o"}}

event: content_block_stop
data: {"type":"content_block_stop","index":1}

event: message_delta
data: {"type":"message_delta","delta":{"stop_reason":"end_turn"}}

event: message_stop
data: {"type":"message_stop"}
```

追踪 5：生成主循环

generate_job 完整流程

```c
// 行 6937: 请求处理主函数
static void generate_job(server *s, job *j) {
    const request *r = &j->r;

    // 1. 计算公共前缀（cache reuse）
    int common = ds4_session_common_prefix(s->session, &prompt);

    // 2. 多级缓存查找
    //    Level 1: 内存 token 精确匹配
    //    Level 2: 内存文本前缀匹配
    //    Level 3: 磁盘文本前缀匹配

    if (common > 0 && common == s->session->checkpoint.len) {
        // 前缀完全匹配，只需增量推理
        ds4_session_sync(s->session, &effective_prompt, ...);
    } else {
        // 需要完整 prefill
        ds4_session_sync(s->session, &prompt, ...);
    }

    // 3. 首次推理获取 logits
    ds4_session_eval(s->session, ...);

    // 4. 采样循环
    for (int i = 0; i < n_predict; i++) {
        // 采样下一个 token
        int token;
        if (r->temperature <= 0)
            token = ds4_session_argmax(s->session);
        else
            token = ds4_session_sample(s->session,
                        r->temperature, r->top_k, r->top_p, r->min_p, &rng);

        // EOS 检查
        if (token == ds4_token_eos(s->engine)) break;

        // 解码 token 文本
        const char *text = ds4_token_text(s->engine, token, &len);

        // DSML 工具调用追踪
        dsml_decode_tracker_update(&dt, text, len);
        // 如果检测到完整工具调用 → 执行并注入结果

        // 思考检查点
        // 如果模型在思考模式，记录 thinking token 计数

        // SSE 流式推送
        if (r->stream) {
            if (r->api == API_OPENAI)
                openai_sse_stream_update(fd, s, r, id, &st, text, len, false);
```

```
            else
                anthropic_sse_stream_update(fd, s, r, id, &st, text, len, fals
e);
        }

        // 推理下一个 token
        ds4_session_eval(s->session, token);
    }

    // 5. 流结束
    if (r->stream)
        sse_done(fd, r, id, prompt_tokens, completion_tokens);

    // 6. 可选: 存储到 KV 缓存
    kv_cache_store_current(s, "generate");
}
```

```
generate_job 完整时序:

客户端 POST /v1/chat/completions
    │
    ▼ 解析请求 → request 结构体
    │
    ▼ 查找缓存
    │   ├── 内存 token 精确匹配 → 跳过 prefill
    │   ├── 内存文本前缀匹配 → 部分跳过
    │   ├── 磁盘文本前缀匹配 → 从文件恢复
    │   └── 未命中 → 完整 prefill
    │
    ▼ 推理循环:
    │   for each token:
    │     ├── ds4_session_eval → logits
    │     ├── ds4_session_sample → token_id
    │     ├── ds4_token_text → "Hello"
    │     ├── dsml_decode_tracker_update → 工具调用检测
    │     ├── openai_sse_stream_update → SSE 推送
    │     └── ds4_session_eval(token_id) → 下一步
    │
    ▼ sse_done → data: [DONE]
    │
    ▼ kv_cache_store_current → 异步写入磁盘
    │
    ▼ close(fd)
```

4. ds4_kvstore 模块化 KV 存储

头部/负载格式

KV 持久化逻辑从 `ds4_server.c` 提取为独立模块 `ds4_kvstore.c/h`，实现与服务器逻辑解耦：

```
// ds4_kvstore.h — 公共接口
typedef struct ds4_kvstore ds4_kvstore;

// 创建/销毁
ds4_kvstore *ds4_kvstore_open(const char *dir, uint64_t budget_mb, ...);
void ds4_kvstore_close(ds4_kvstore *kvs);

// 存储操作
bool ds4_kvstore_save(ds4_kvstore *kvs, const char *sha1_hex,
                       const uint8_t *header, size_t header_len,
                       const char *text, size_t text_len,
                       ds4_session *session, ...);

// 查找与加载
int ds4_kvstore_find_prefix(ds4_kvstore *kvs, const char *text, ...);
bool ds4_kvstore_load(ds4_kvstore *kvs, const char *sha1_hex,
                       ds4_session *session, ...);

// 驱逐
void ds4_kvstore_evict(ds4_kvstore *kvs, const char *protect_sha1);

// Chat anchor 定位
int ds4_kvstore_chat_anchor_pos(const ds4_tokens *tokens, int len);
```

头部格式（48 字节固定）：magic `KVC` + version + model_id + reason + token_count + hits + creation_time + last_used。尾部钩子允许调用方注入协议特定数据（KTM 工具记忆、responses_visible 标记等）。

---

5. ds4_agent 代理架构

双线程设计

```c
// ds4_agent.c — 简化结构
typedef struct {
    pthread_mutex_t mu;
    pthread_cond_t  worker_cv;
    pthread_cond_t  ui_cv;

    // 共享状态
    buf user_input;          // UI → Worker: 用户输入
    buf worker_output;       // Worker → UI: 模型输出
    buf tool_confirmation;   // Worker → UI: 确认请求

    bool worker_busy;
    bool confirmation_result;
    bool stopping;
} agent_shared;

// Worker 线程主循环
static void *agent_worker(void *arg) {
    for (;;) {
        // 等待用户输入
        pthread_mutex_lock(&shared->mu);
        while (!has_input && !shared->stopping)
            pthread_cond_wait(&shared->worker_cv, &shared->mu);

        // 推理 + DSML 流式解析
        // 检测到工具调用 → 请求确认 → 执行工具 → 继续推理
    }
}

// UI 线程主循环
static void agent_ui_loop(agent_shared *shared) {
    for (;;) {
        char *line = linenoise("> ");   // 读取用户输入
        // 传递给 worker
        // 同时监听 worker 输出和确认请求
    }
}
```

`ds4_agent.c` 约 9607 行，是项目中仅次于 `ds4.c` 和 `ds4_metal.m` 的第三大源文件。

Part 6: GPU 加速

> CPU 是正确性的参照，GPU 是性能的主力。本主题覆盖 Metal（macOS）和 CUDA（NVIDIA）两个 GPU 后端，以及 Flash Attention 等关键优化。

涵盖内容

章节	核心主题
1. GPU 后端 + 总结	ObjC 互操作、Metal kernel、CUDA 后端、Flash Attention、总结回顾

核心概念

- kv-cache — GPU 端的 KV 缓存管理（CUDA Managed Memory）
- softmax — Flash Attention 中的 online softmax
- quantization — GPU 反量化 kernel
- rope — GPU 端的 RoPE 计算

前置知识

- Part 3（Attention、量化）
- Part 4（推理循环、RoPE）
- Part 5（推理服务全貌）

学习路径

读完本主题后，你将理解：

1. Metal/CUDA 如何将 CPU 参考实现映射为 GPU kernel

2. Flash Attention 的 online softmax 和分块计算策略
3. ds4.c 整体架构的总结和回顾

→ 完成！回到 <u>知识地图</u> 查看全局视图。

Part 6: GPU 加速

> Metal、CUDA、Flash Attention

---

1. GPU 后端 + 总结

> CPU decode 速度约 5-10 t/s，对话体验卡顿。GPU 后端将矩阵运算卸载到 Metal/CUDA/ROCm，decode 速度提升 5-10 倍。今天学习 ObjC 互操作、Metal 计算管线和 Flash Attention，最后用一个完整的请求生命周期串联 15 天的知识。

GPU 后端将 CPU 注意力 卸载到 Metal/CUDA/ROCm，大幅提升推理速度。

C 知识点

1. Objective-C 互操作

ds4_metal.m 用 Objective-C 调用 Metal API，ds4_cuda.cu 用 CUDA C++ 调用 NVIDIA API。ds4.c 是纯 C，通过 `ds4_gpu.h` 统一抽象层与 GPU 后端交互：

```c
// ds4_gpu.h — GPU 后端无关的抽象接口
// 所有函数签名使用 ds4_gpu_tensor * 不透明指针
// 不暴露 Metal (MTLBuffer) 或 CUDA (CUdeviceptr) 类型

typedef struct ds4_gpu_tensor ds4_gpu_tensor;

// 张量生命周期
ds4_gpu_tensor *ds4_gpu_tensor_alloc(ds4_gpu_graph *g, const char *name, uint64_
t nbytes);
void ds4_gpu_tensor_free(ds4_gpu_tensor *t);
void ds4_gpu_tensor_read(ds4_gpu_tensor *t, void *dst, uint64_t offset, uint64_t
len);
void ds4_gpu_tensor_write(ds4_gpu_tensor *t, const void *src, uint64_t offset, u
int64_t len);

// 命令队列
void ds4_gpu_begin(ds4_gpu_graph *g);
void ds4_gpu_flush(ds4_gpu_graph *g);
void ds4_gpu_end(ds4_gpu_graph *g);
void ds4_gpu_synchronize(ds4_gpu_graph *g);

// ~60 个推理内核: embedding, RoPE, attention, MoE, steering 等
void ds4_gpu_embedding(ds4_gpu_graph *g, ...);
void ds4_gpu_flash_attn(ds4_gpu_graph *g, ...);
// ...
```

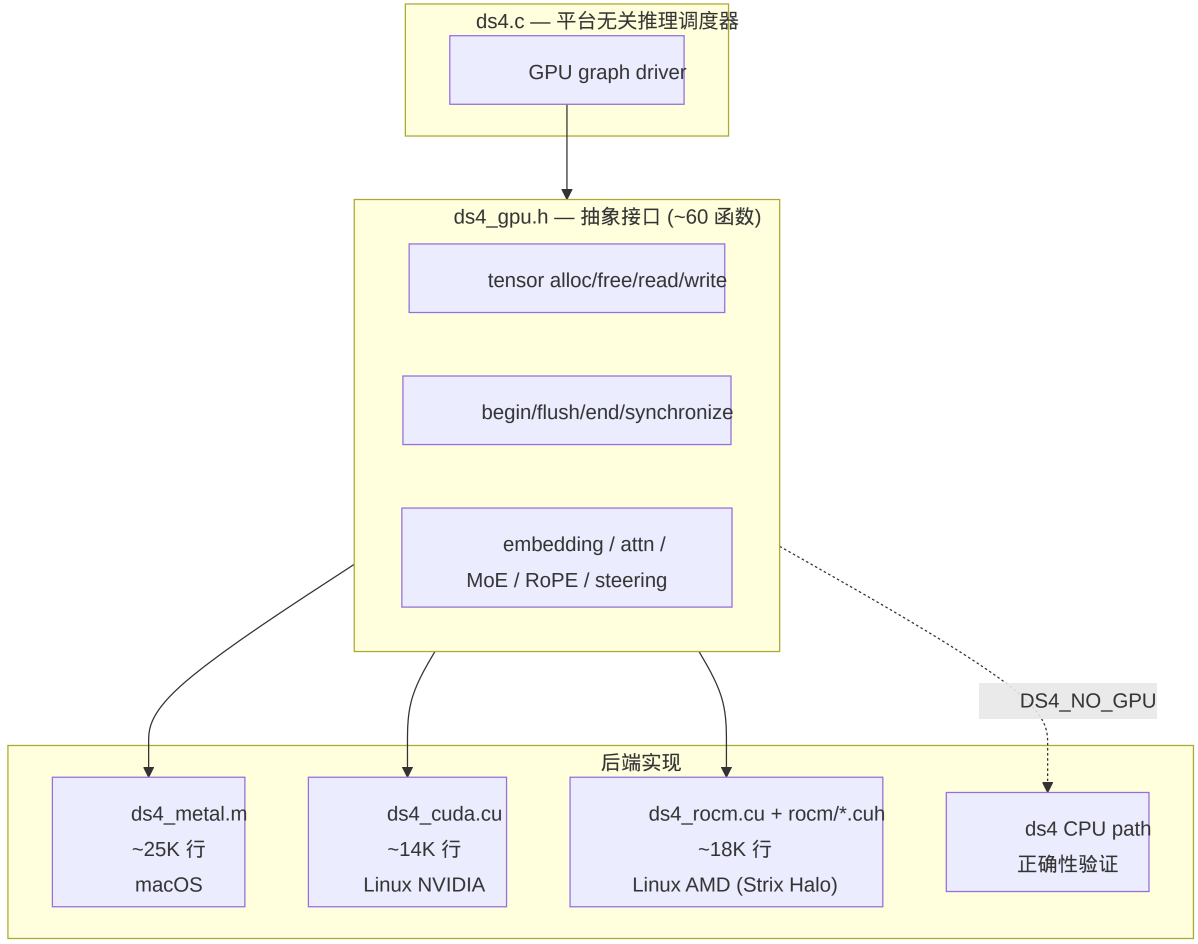
后端枚举 (`ds4.h`):

```c
typedef enum {
    DS4_BACKEND_METAL,
    DS4_BACKEND_CUDA,
    DS4_BACKEND_CPU,
} ds4_backend;
```

编译选择:

- macOS: 默认 Metal 后端 (`CORE_OBJS = ds4.o ds4_metal.o`)
- Linux: 默认 CUDA 后端 (`CORE_OBJS = ds4.o ds4_cuda.o` ，用 `nvcc` 链接)
- `make cpu` : 两平台通用， `-DDS4_NO_GPU` 编译纯 CPU 版本
- CUDA 构建目标:
  - `make cuda-spark` : DGX Spark / GB10 专用（无需指定 arch）
  - `make cuda-generic` : 通用 CUDA GPU
  - `make cuda CUDA_ARCH=sm_N` : 显式指定架构（如 `sm_80` ）
- ROCm 构建目标（AMD Strix Halo / `gfx1151` ）:

  - `make strix-halo`（别名 `make rocm`）：用 `hipcc` 链接 `ds4_rocm.o` + `rocm/*.cuh`，定义 `-DDS4_ROCM_BUILD`，默认 `ROCM_ARCH=gfx1151`
  - ROCm 后端现已并入 **main** 分支（此前长期只在独立的 `rocm` 分支，由社区因 antirez 无对应硬件而维护）
- 质量评分工具 `gguf-tools quality-score` 也支持跨平台：macOS 用 `$(CC)` 链接 Metal，Linux 用 `$(NVCC)` 链接 `-lcudart -lcublas`，通过 `#ifdef __APPLE__` 编译时选择后端

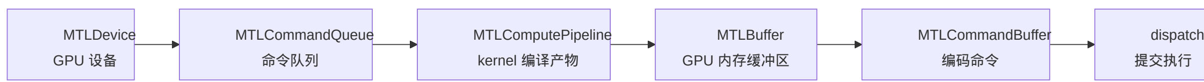


关键设计：

- `ds4_gpu.h` 对 C 完全隐藏具体 GPU API 类型（Metal/CUDA/HIP）
- 新增 GPU 后端只需实现 `ds4_gpu.h` 中的 ~60 个函数，不改动 `ds4.c`
- CPU 路径通过 `DS4_NO_GPU` 宏完全绕过 GPU 代码
- ROCm 后端靠编译期 `-DDS4_ROCM_BUILD` 选择（而非运行期 enum 值）：它在 `ds4_gpu.h` 里用 `#ifdef DS4_ROCM_BUILD` 暴露若干独有钩子（layer-major 批量加载、`q8_f16` 缓存释放等），其它后端提供 no-op stub 保持链接。`ds4_backend` 枚举仍是 `METAL/CUDA/CPU`——ROCm 复用了非 Apple（Linux）的代码路径，只是把实现文件从 `ds4_cuda.cu` 换成 `ds4_rocm.cu`。

2. Metal 计算管线

Metal GPU 执行模型推理的核心流程：



ds4_metal.m 中的流程：

```
id<MTLDevice> device = MTLCreateSystemDefaultDevice();
// 启动时打印设备名称和内存
// ds4: Metal device Apple M2 Ultra, 192.00 GiB RAM
id<MTLCommandQueue> queue = [device newCommandQueue];

// 编译 .metal shader
id<MTLLibrary> lib = [device newLibraryWithSource:shader_source ...];
id<MTLFunction> fn = [lib newFunctionWithName:@"kernel_name"];
id<MTLComputePipelineState> pipeline = [device newComputePipelineStateWithFuncti
on:fn ...];

// 执行
id<MTLCommandBuffer> cmd = [queue commandBuffer];
id<MTLComputeCommandEncoder> enc = [cmd computeCommandEncoder];
[enc setComputePipelineState:pipeline];
[enc setBuffer:input offset:0 atIndex:0];
[enc setBuffer:output offset:0 atIndex:1];
[enc dispatchThreadgroups:... threadsPerThreadgroup:...];
[enc endEncoding];
[cmd commit];
[cmd waitUntilCompleted];
```

可选 **buffer** 绑定占位

encoder 绑定 buffer 时，可选参数即使不使用也必须绑定占位值，否则 Metal debug layer 会报校验错误：

```
// ds4_metal.m — router finalize
float zero_f32 = 0;
if (has_bias)
    [enc setBuffer:bias_buf offset:0 atIndex:3];
else
    [enc setBytes:&zero_f32 length:4 atIndex:3];  // 零值占位

if (hash_mode)
    [enc setBuffer:hash_buf offset:0 atIndex:5];
else
    [enc setBytes:&zero_f32 length:4 atIndex:5];  // 零值占位
```

Metal debug layer 要求 shader 声明的每个 `[[buffer(N)]]` 都有对应绑定，不用的可选参数用 `setBytes:&zero` 填充即可。

3. Metal Shading Language

`.metal` 文件用 MSL 编写 GPU kernel：

```
// metal/softmax.metal (简化)
kernel void softmax(
    device float* input  [[buffer(0)]],
    device float* output [[buffer(1)]],
    uint gid [[thread_position_in_grid]])
{
    // 每个 GPU 线程处理一个元素
    output[gid] = exp(input[gid]) / sum;
}
```

ds4 项目有 19 个 `.metal` 文件，覆盖：

- `dense`: 矩阵乘法
- `flash_attn`: Flash Attention
- `moe`: MoE 专家计算
- `norm`: RMSNorm
- `softmax`: Softmax
- `dsv4_kv/dsv4_rope`: KV cache 和 RoPE
- `argsort`: 排序（用于 top-k）
- 等等

Metal Neural Acceleration (NAX)

M5 芯片引入了 MPP/Neural Accelerator 硬件（NAX），ds4 在检测到 M5 时自动利用两个 NAX kernel：

1. **Indexer Scores NAX** — 压缩 KV 注意力的索引评分，将评分计算卸载到 Neural Accelerator
2. **Tensor Matmul NAX** — MoE 专家的 Q8_0 矩阵乘法，支持 32/64/128 tile sizes

检测方式：`g_metal4_m5_neural_accelerators_hint`（从系统属性读取的 M5 标志）。当 `n_tokens >= 16` 时启用 NAX 路径，低于此阈值时退回标准 GPU 计算。

Compressed KV in F16

Metal 后端新增 `comp_kv_f16` 模式——将压缩 KV cache 以半精度（float16）而非 float32 存储。每行 KV 内存减半（从 `DS4_N_HEAD_DIM × 4B` 降为 `DS4_N_HEAD_DIM × 2B`），在大上下文场景下节省大量内存。对应的 Metal kernel 位于 `metal/dsv4_kv.metal`，在读入时将 F16 解压缩为 F32 用于注意力计算。

GPU 功耗节流

GPU 后端支持 `power_percent`（1-100）参数控制推理速度。实现方式是在每个 prefill 层和 decode token 后调用 `graph_power_sleep()`：

```
sleep_us = (100 - power) / power × elapsed_us
```

agent 模式下通过 `ds4_engine_set_power()` 动态调整——等待用户确认时降低功耗，活跃推理时恢复满速。

Typed Pointer vs void Pointer

Metal shader 参数应使用 `device const char *` 而非 `device const void *`，让 Metal 反映正确的只读访问语义：

```
// 改前：void * — 不表达访问语义，debug layer 可能报校验错误
kernel void get_rows(device const void* x [[buffer(0)]], ...)

// 改后：char * — 明确只读，满足 Metal debug layer 校验
kernel void get_rows(device const char* x [[buffer(0)]], ...)
```

`get_rows.metal` 和 `set_rows.metal` 等文件中的参数已从 `void *` 改为 `char *`。`void *` 在 Metal 中不传达内存访问意图，`char *` 配合 `const` 明确表达只读语义。

LLM 知识点

1. GPU 加速原理

CPU vs GPU 的根本区别：

```
CPU: 4-12 个强核心，每个核心处理复杂任务
GPU: 数千个弱核心，每个核心做简单计算但并行执行

矩阵乘法 A[n,m] × B[m,k]:
CPU: 逐元素计算，串行（或少量并行）
GPU: 每个元素一个线程，数千元素同时计算
```

为什么 LLM 推理需要 GPU？

- 矩阵乘法是高度并行的（每个输出元素独立）

- GPU 的内存带宽远高于 CPU
- Metal GPU 可以直接引用 mmap 内存（零拷贝）

2. Flash Attention

标准注意力：先算完整的 Q·K^T 矩阵，再 softmax。内存 $O(n^2)$。

Flash Attention：分块计算，每次只加载一小块 Q 和 K 到高速缓存，在片上完成 softmax。内存 O(n)。

ds4 的 `metal/flash_attn.metal` 实现了这个优化。

3. 零拷贝 MTLBuffer

```
// Metal 可以直接引用 mmap 的内存，不需要拷贝到 GPU 专用内存
id<MTLBuffer> buf = [device newBufferWithBytesNoCopy:
    (void*)(mmap_base + tensor_offset)
    length:tensor_bytes
    options:MTLResourceStorageModeShared
    deallocator:nil];
```

这就是为什么模型加载用 `MAP_SHARED`——Metal GPU 直接从文件映射读取权重，不需要额外的 GPU 内存拷贝。81GB 的模型只需要 81GB 的磁盘空间和 mmap 映射，不需要额外的 GPU 显存。

模型视图与张量覆盖不变量

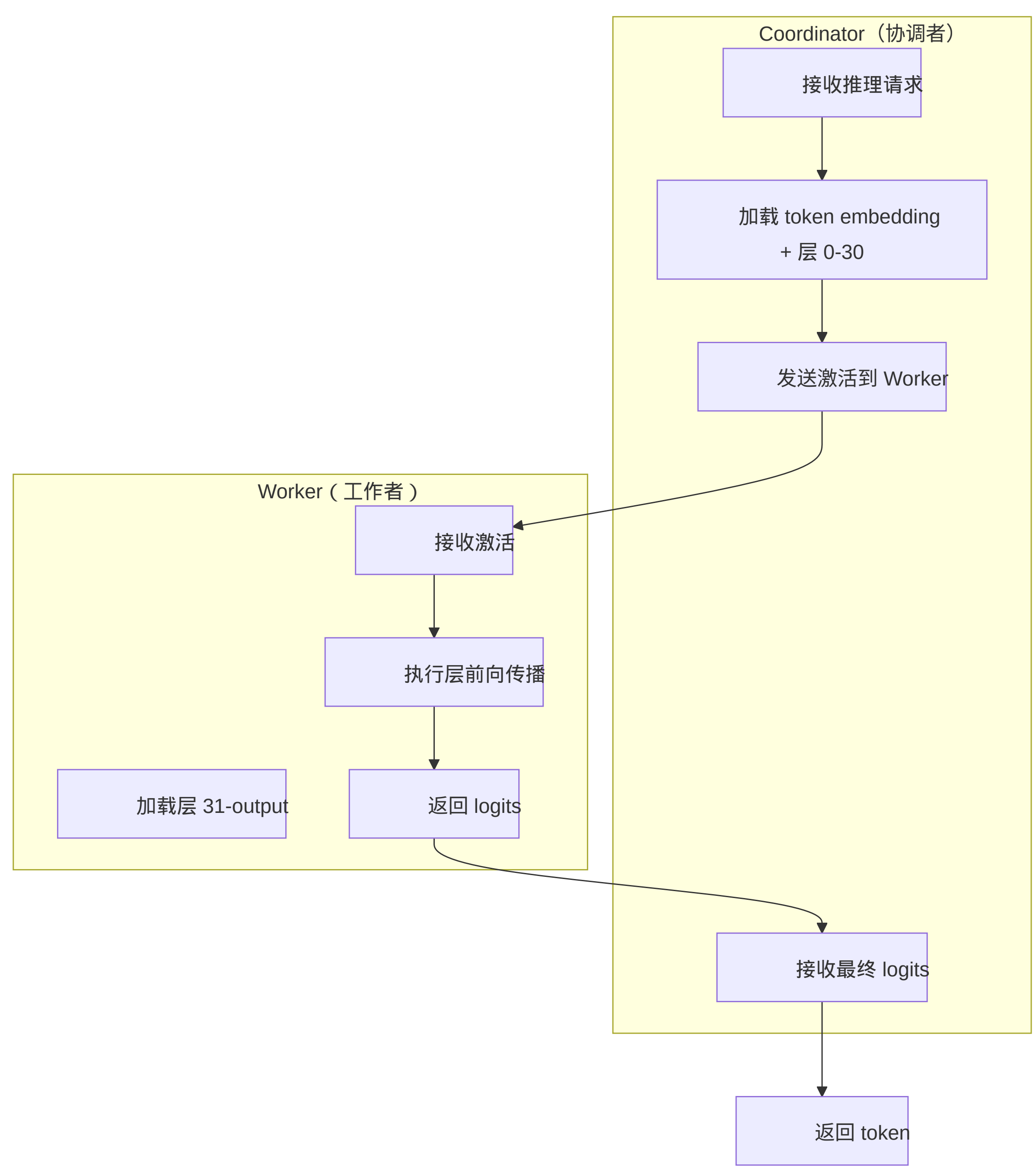
当模型文件超过 Metal 设备的 `maxBufferLength` 时，引擎创建多个重叠的 `MTLBuffer` 视图来覆盖整个文件。相邻视图必须有足够大的重叠区域，确保每个张量至少完整包含在一个视图中：

```
// 最大单张量大小限制，决定视图重叠区域
#define DS4_METAL_MODEL_MAX_TENSOR_BYTES (2ull * 1024ull * 1024ull * 1024ull) //
2 GiB

// 重叠 = round_up(MAX_TENSOR_BYTES, page) + page
// 视图步进 = max_buffer - overlap
```

Q4_K 量化模型的 MoE 专家张量约 1.125 GiB，之前的 672 MiB 限制不足以覆盖。2 GiB 阈值确保所有量化格式（包括 Q4 专家张量）都能完整映射到至少一个视图中。

4. 分布式推理（Distributed Inference）

PRO 模型（~430GB IQ2_XXS 或 ~838GB Q4 分片）太大，单机无法运行。ds4 新增 **coordinator/worker** 分布式推理架构，将模型层切片分配到多台机器上执行：



核心设计：

组件	职责
Coordinator	接收请求、执行前半层、转发激活、接收 logits
Worker	注册层范围、执行后半层、返回结果
传输协议	自定义 TCP 二进制协议（magic `0x44533444` = "DS4D"）
KV 快照	拓扑无关：保存时聚合所有 worker 的层张量，加载时按当前路由分发

Worker 通过 HELLO 消息注册（包含 `layer_start`、`layer_end`、`model_id`、`quant_bits`），Coordinator 据此构建路由表。激活传输支持可配置的 `activation_bits`（默认 32-bit float，可压缩为更低精度减少网络开销）。

```
# Coordinator（加载前半层）
./ds4 --role coordinator -m pro-q4-layers00-30.gguf -p "Hello"

# Worker（加载后半层）
./ds4 --role worker -m pro-q4-layers31-output.gguf
```

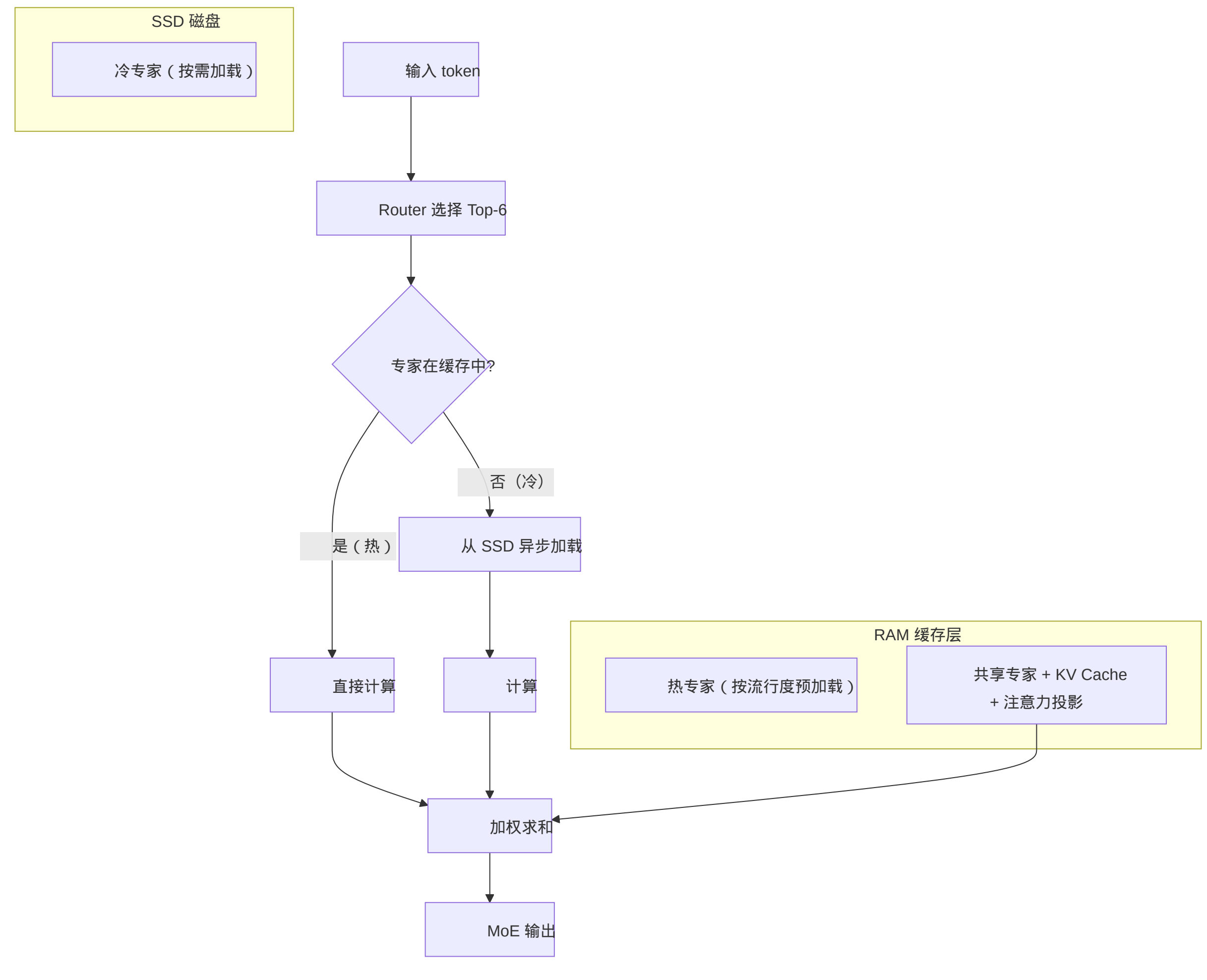
模型分片加载（Model Span Loading）：分布式引擎使用 `ds4_gpu_set_model_map_spans()` API，只加载分配给本机的层范围，而非整个模型。这通过 offset/size 对描述需要映射的字节范围，配合 chunked preloading 异步拷贝到设备内存。

```
// 只加载本 worker 负责的层切片
ds4_gpu_set_model_map_spans(
    model_map, model_size,
    offsets, sizes, span_count,
    max_tensor_bytes
);
```

详见 分布式推理。

4a. SSD Streaming（SSD 流式推理）

除了分布式推理，ds4 还提供另一种突破内存限制的方式：SSD Streaming。核心洞察是 MoE 层有 256 个路由专家，但每个 token 只激活 6 个——路由专家占模型大部分空间，但大部分时间都在"睡觉"。



缓存计划：`ds4_ssd_cache_plan` 根据可用 RAM 预算自动计算能缓存多少专家。自动预算取推荐工作集的 80%、扣除非路由权重，剩下的用于路由专家；显式 `--ssd-streaming-cache-experts 32GB` 则直接给字节预算（换算成完整专家数），过大时推理前自动封顶以保持可锁定（lockable）。详见 SSD Streaming 术语表。

热度排名（启动）：`ds4_streaming_hotlist.inc`（13,334 行）按流行度预排列所有 (layer, expert_id) 对。启动时预加载最热门的专家，自动预热默认封顶 4096 个。

运行期淘汰（路由热度）：决定"淘汰谁"的是一张运行期热度表 `route_hotness[layer][expert]`，每处理 16 个 decode token 整表右移衰减一位（`>>= 1`），淘汰时驱逐热度最低者。这张表是提示局部的——新 prompt 开始时清零（`ds4_gpu_stream_expert_cache_reset_route_hotness()`），但常驻缓存本身跨会话保持热：易变的启发式状态要重置，昂贵的物化缓存要复用。

三路 **Prefill**：默认 prefill 调度器按 prompt 长度分流——极短走逐 token decode 式、中等走 selected-expert 批量、长 prompt 走全层 layer-major。切分点按量化格式调优，避免短 prompt 落到"加载一堆用不上的专家"的慢路径。

混合精度路由专家：当 GGUF 跨层不统一量化（per-layer boosted quants，例如全局 IQ2_XXS 但有几层升到 Q4_K）时，提升层的 `ffn_exps` 张量被纳入 decode-span 构建器走 mapped-view 兜底，且 slab 分配器在启动时预置尺寸类、对超大尺寸 freeze+reject 而非 last-writer-wins，避免 slab 被毒化。

多后端实现：SSD streaming 不再是 Metal 独享——CUDA（`ds4_cuda.cu`，有界缓存 + 进度上报）和 ROCm（`ds4_rocm.cu`，selected-expert 缓存 + 重叠读取 + 全层 streaming prefill）都已实现。streaming API 已重构为传一张 `ds4_gpu_stream_expert_table` 表：

```
typedef struct ds4_gpu_stream_expert_table {
    const void *model_map;   uint64_t model_size;
    uint32_t layer, n_total_expert;
    uint64_t gate_offset, up_offset, down_offset;
    uint64_t gate_expert_bytes, down_expert_bytes;
} ds4_gpu_stream_expert_table;

// 异步加载缺失专家（重构后参数更简洁）
int ds4_gpu_stream_expert_cache_begin_selected_load(
        const ds4_gpu_stream_expert_table *table,
        const int32_t *selected_ids, uint32_t n_selected);
uint32_t ds4_gpu_stream_expert_cache_current_count(void);
```

命令行选项：

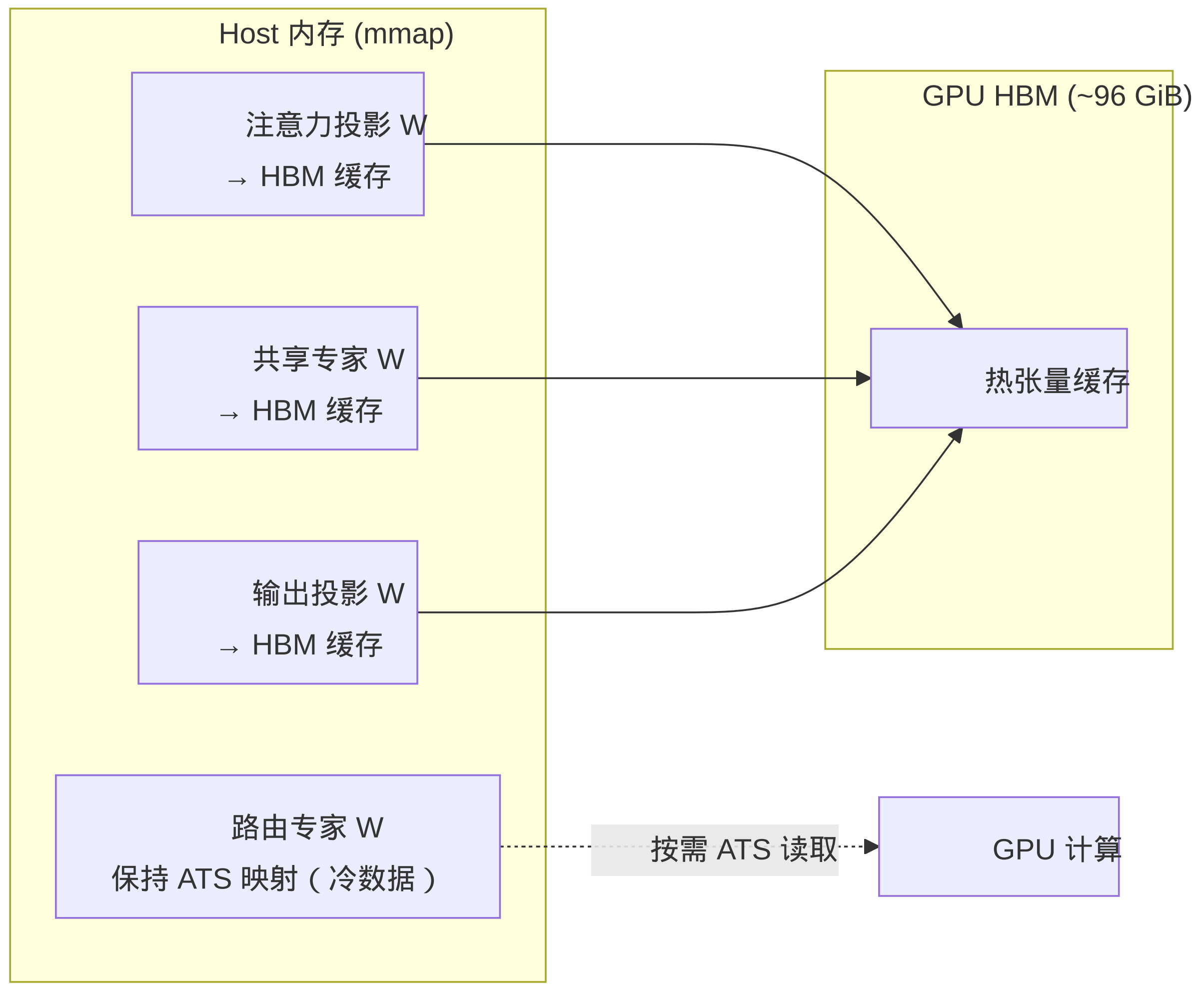
选项	说明
`--ssd-streaming`	启用 SSD streaming 模式
`--ssd-streaming-cache-experts N`	缓存字节预算（过大时推理前封顶）
`--ssd-streaming-preload-experts N`	显式指定启动预加载专家数（默认自动封顶 4096）
`--ssd-streaming-cold`	冷启动（不预加载 hotlist；仅用于测量）

与分布式推理的区别：SSD streaming 是单机方案，适合内存不足但有大容量 SSD 的场景；分布式推理是多机方案，适合 PRO 模型等超出单机存储极限的情况。两者可以结合使用。

4b. DGX Spark / GB10 后端（HBM 缓存）

NVIDIA DGX Spark（GB10 芯片，121 GiB UMA）有一个特殊的 CUDA 推理挑战：ATS（Address Translation Service）允许 GPU 直接消费 host mmap 权重，但有效带宽远低于 HBM 驻留数据。

ds4 的解决方案：启动时 **HBM 缓存**——将"热"张量（注意力投影、MoE 共享专家、输出投影）拷贝到 GPU HBM，冷 MoE 路由专家保持 ATS 映射：



默认缓存预算 96 GiB（`cuda_model_cache_limit_bytes()`），通过 `DS4_CUDA_WEIGHT_CACHE_LIMIT_GB` 环境变量可覆盖。IQ2 模型（~81GB）和混合 Q2/Q4 模型（~91GB）可完整缓存；完整 Q4 模型（~153GB）超出预算，需使用分布式层加载。

HBM 缓存查找优先于 UVA 映射指针——直接 HBM 读取比通过 host 页表映射快约 10%。`cuda_model_range_ptr()` 使用 hash-keyed 查找实现单次读取命中。

效果（DGX Spark / GB10，ds4flash，n=256）：

- 优化前：~13.9 t/s（纯 ATS 映射）
- 优化后：~16.13 t/s（HBM 缓存热张量 + 小 batch kernel 融合）

编译目标：`make cuda-spark`（自动检测 GB10 架构，无需指定 `CUDA_ARCH`）。

Metal View Cap 修复

分布式推理的 span 映射使用 `ds4_gpu_add_model_view_range()` 创建 Metal 视图。之前的实现给所有大映射都应用 128 GiB 视图上限，包括普通的单次完整模型映射，导致 Metal VM 验证变慢。

修复：新增 `use_default_view_cap` 参数——普通完整模型映射（`ds4_gpu_map_model_views()`）不设上限，分布式 span 映射（`ds4_gpu_set_model_map_spans()`）保留 128 GiB 分割。这样普通映射保持单次映射的高效性，只有需要切分的分布式场景才做视图分割。

模型缓存 **FD** 绑定

CUDA 后端使用直接文件 I/O（非 mmap）加载路由专家权重，需要正确的文件描述符和 mmap 基地址来计算文件偏移。当 MTP 模型在主模型 fd 绑定和缓存准备之间加载时，fd 可能被错误覆盖。

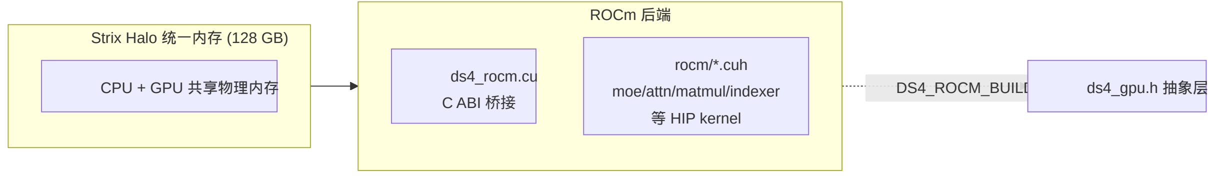
修复：新增 `ds4_gpu_set_model_fd_for_map(int fd, const void *model_map)` 接受显式的 fd 和 model_map 指针，替换隐式使用 `g_model_host_base` 的旧 `ds4_gpu_set_model_fd()`。MTP 模型设置完成后，重新绑定主模型的 fd：

```
// 缓存前绑定正确的 fd
ds4_gpu_set_model_fd_for_map(e->model.fd, e->model.map);
accelerator_cache_model_tensors(...);
// MTP 设置后重新绑定主模型 fd
ds4_gpu_set_model_fd_for_map(e->model.fd, e->model.map);
```

4c. Strix Halo / ROCm 后端

AMD Strix Halo（如 Framework Desktop，Radeon 8060S，`gfx1151`，128 GB 统一内存）是 ds4 的第三个 GPU 后端。ROCm 后端现已并入 main 分支（此前长期只在独立的 `rocm` 分支，由社区维护，因为 antirez 没有对应硬件）。

构建：`make strix-halo`（别名 `make rocm`），用 `hipcc` 编译 `ds4_rocm.o` + `rocm/*.cuh`（约 18K 行新代码），定义 `-DDS4_ROCM_BUILD`。后端基于 rocWMMA（AMD 的矩阵乘加速库），需要 `libhipblas` / `libhipblaslt` / `librocblas` / `librocwmma` 等。



关键工程难点（详见上游 `STRIXHALO.md`）：

1. **ROCm** 安装补全：Ubuntu 26.04 的 `librocwmma-dev` 缺 `rocwmma/internal/` 头文件，需手动 clone rocWMMA 7.1.0 补齐；若 ROCm 装在 `/usr` 而工具链期望 `/opt/rocm`，要建符号链接。
2. 显存扩展（**GT TTM aperture**）：128 GB Strix Halo 系统默认只暴露约 62 GB GPU 可见内存，不够 80.76 GiB 模型 + 运行时缓冲。需设内核参数扩大 GTT：

```
amd_iommu=off amdgpu.gttsize=126976 ttm.pages_limit=32505856 ttm.page_pool_si
ze=32505856
```

重启后 `rocminfo` 应报 `gfx1151 pool: 130023424 KB`（约 124 GB）。

3. **GGUF** 选择：在 Strix Halo 上用标准 IQ2XXS/Q2K/Q8 imatrix GGUF；避免 mixed IQ2/IQ4 或 IQ2/Q4 GGUF——它们给 ROCm 路径更大的内存压力，可能触发系统 OOM 而非干净的 ds4 失败。

ROCm 特定修复（并入本次同步）：

- `Fix ROCm distributed slice mapping` — 分布式层切片映射
- `Fix ROCm MTP model residency` — MTP 模型驻留
- `Fix ROCm mixed streaming expert fallback` — 混合精度 streaming 专家兜底

可选后端钩子（**Fused GPU Ops as Backend Hooks**）：融合算子（如 fused RMSNorm/RoPE、fused MoE）从 `ds4.c` 里写死的 Metal/CUDA 分支重构为 `ds4_gpu.h` 上的可选后端钩子。每个后端按需实现（Metal 131 行、CUDA 94 行新增），`ds4.c` 侧减为统一的钩子分发（-40 行）。这让 ROCm 这类新后端能选择性提供融合内核而不必复制所有分支逻辑。

4d. 帮助系统（Help System）

新增结构化帮助模块 `ds4_help.c/h`，替代各工具中散乱的 usage 打印。API：

```
// ds4_help.h
typedef enum {
    DS4_HELP_DS4,       // ds4 CLI
    DS4_HELP_SERVER,    // ds4-server
    DS4_HELP_AGENT,     // ds4-agent
    DS4_HELP_BENCH,     // ds4-bench
    DS4_HELP_EVAL,      // ds4-eval
} ds4_help_tool;

void ds4_help_print(FILE *fp, ds4_help_tool tool, const char *topic);
```

关键设计：

- 主题过滤：`topic` 参数可查询特定子主题（如 `--help distributed`）
- **TTY** 感知：`isatty()` 自动检测，终端中显示 ANSI 256-color 彩色输出，非终端输出纯文本
- 格式化助手：`opt()` 打印彩色选项行、`para()` 打印黄色段落、`title()` 打印章节标题
- 分布式模式指导：帮助文本中包含 distributed mode 的用法说明和示例

4e. 模型下载脚本更新

`download_model.sh` 新增 PRO 模型下载目标：

目标	说明	大小
`pro-q2-imatrix`	PRO IQ2_XXS 单文件	~430 GB
`pro-q4-layers00-30`	PRO Q4 前半层（coordinator）	~426 GB
`pro-q4-layers31-output`	PRO Q4 后半层（worker）	~412 GB
`pro-q4-split`	下载两个 Q4 分片	~838 GB

PRO 文件过大，脚本自动检测并使用 **Hugging Face CLI**（`hf` from `huggingface_hub`）下载，支持断点续传。未安装时提示 `python3 -m pip install -U huggingface_hub hf_xet`。

非 PRO 目标下载后自动符号链接到 `./ds4flash.gguf`，PRO 目标打印分布式用法指引。

4f. 方向性激活引导（**Directional Steering**）

↗

基于论文 Refusal in Language Models Is Mediated by a Single Direction 的思想，在推理时对每层激活做低秩编辑：

```
y = y - scale * direction[layer] * dot(direction[layer], y)
```

- 引导文件：43 × 4096 的 f32 矩阵（43 层，每层一个 4096 维归一化方向向量）
- 可在 attention 输出后（`--dir-steering-attn`）和/或 FFN 输出后（`--dir-steering-ffn`）施加
- 正 scale 移除该方向，负 scale 放大该方向
- 提取工具：`dir-steering/tools/build_direction.py`（用好/坏两组 prompt 取激活差异，归一化后生成 `.f32` 文件）
- 示例：`dir-steering/` 包含预构建的 verbosity 方向向量，负 FFN scale 使回答更简洁

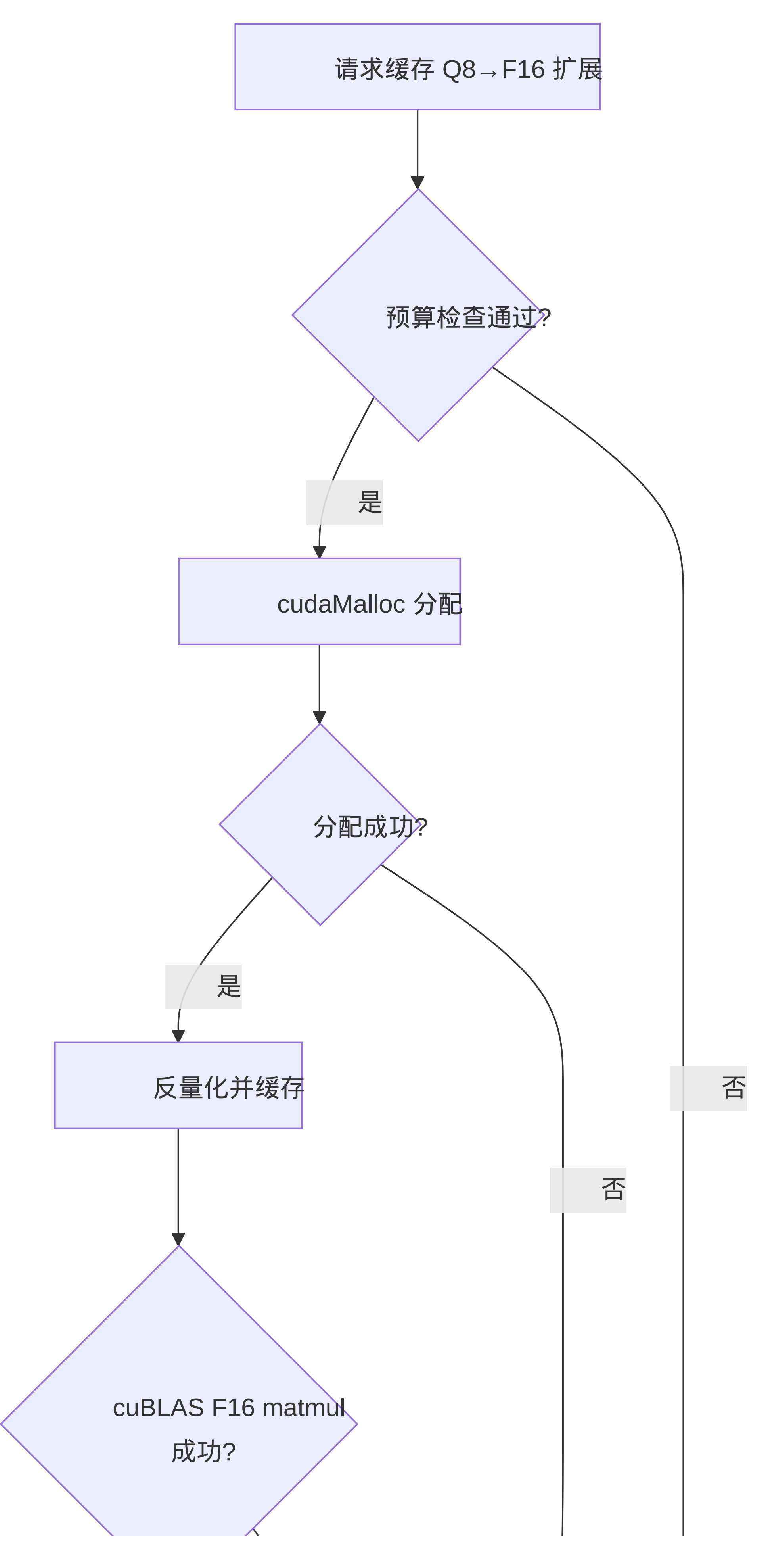
4g. CUDA Q8→F16 缓存显存预算守卫

PRO 模型 GPU 注意：DeepSeek V4 PRO（1.6T 总参数，49B 激活参数）的专家张量更大（更多专家 × 更高维度），Q4 量化模型需要 512GB 内存的 Mac Studio。GPU 后端在加载 PRO 模型时会自动调整 KV cache 容量和 buffer 分配。

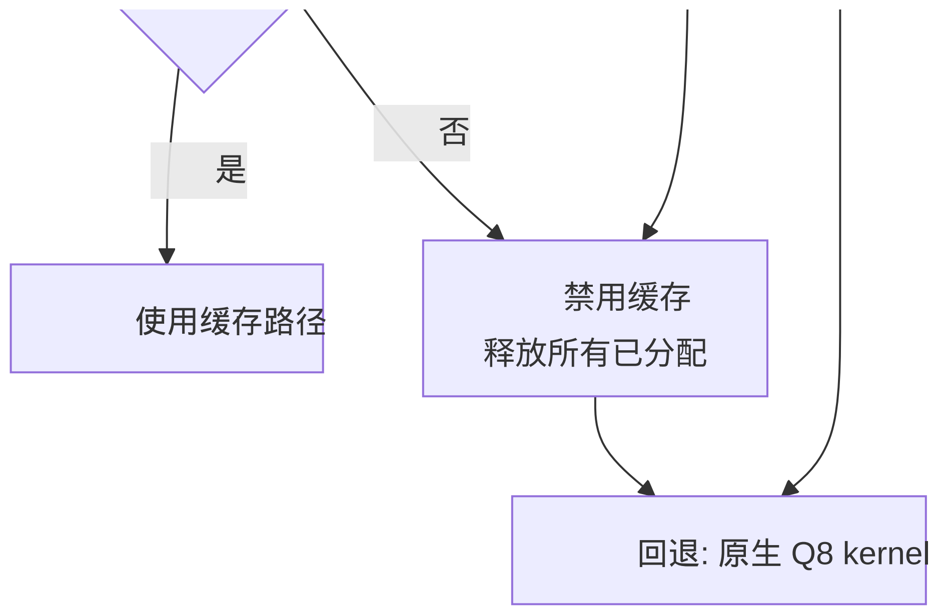
Q8 权重在推理时被扩展为 F16 存入 GPU 显存，加速后续 cuBLAS 矩阵乘法。但显存有限的 GPU 上，缓存可能占满 VRAM 导致后续分配失败：

```
// 显存预算检查流程
static int cuda_q8_f16_cache_has_budget(uint64_t request_bytes, const char *labe
l) {
    // 1. 检查用户设定的硬上限
    const uint64_t limit = cuda_parse_mib_env("DS4_CUDA_Q8_F16_CACHE_MB");
    // 2. 查询当前显存空闲量
    cudaMemGetInfo(&free_bytes, &total_bytes);
    // 3. 计算保留量：max(4 GiB, 5% VRAM)
    reserve = max(4ull * 1024 * 1024 * 1024, total_bytes / 20);
    // 4. 分配后剩余空间必须 >= reserve
    return (free_bytes - request_bytes >= reserve);
}
```

三层回退机制：



请求缓存 Q8→F16 扩展
预算检查通过?
是
cudaMalloc 分配
分配成功?
是
反量化并缓存
否
否
cuBLAS F16 matmul
成功?



关键设计：缓存是纯加速路径，失败不影响正确性。环境变量控制：

- `DS4_CUDA_Q8_F16_CACHE_MB=0`：完全禁用
- `DS4_CUDA_Q8_F16_CACHE_MB=8192`：限制缓存 8 GiB
- `DS4_CUDA_Q8_F16_CACHE_RESERVE_MB=8192`：覆盖默认保留量

4h. CUDA 长上下文修复

长上下文（>100K token）暴露了多个 CUDA kernel 的固定缓冲区假设：

Shared Memory 分数溢出 → 在线注意力 kernel 回退：

```
// 编译时常量限制了 shared memory 中的分数缓冲区大小
enum { DS4_CUDA_ATTENTION_SCORE_CAP = 8192u };
// 当压缩行数超过缓冲区容量时，切换到在线注意力 kernel
// 在线 kernel 不需要预分配所有分数，逐块处理
```

多级 **Tree-Merge TopK** → 替代单层 chunk+merge：

```
// 旧：单层 chunk(2048) → merge，无法处理 >4096 压缩行
// 新：chunk(4096) → 多级 tree-merge，每 8 组合并一次
// n_sets: n_chunks → ceil(n/8) → ceil(n/64) → ... → ≤8 → 最终合并
```

回归测试 `tests/cuda_long_context_smoke.c`：模拟长上下文场景（多个 prefill frontier + decode），验证 CUDA 路径在边界条件下的正确性。

MoE Down 路径清理

移除了 block16 MoE down 诊断路径（-172 行），因存在 top-logit 不稳定性和贪心首 token 行为差异。默认保留更快的 tile16/row2048 路径，block16 改为 `DS4_CUDA_MOE_DOWN_BLOCK16` 环境变量 opt-in。

CUDA 内存优化

三项改进减少不必要的 GPU 开销：compressor state 清零改用 `cudaMemsetAsync`（替代 fill kernel 调用）；graph/session tensor 改用 `cudaMalloc` 设备内存替代 managed memory（避免多余的页面迁移）；新增 backend `fill_f32` hook 让 CUDA 设备端直接初始化张量。

CUDA Managed KV Cache

百万 token 级上下文时，KV cache 张量改用 `cudaMallocManaged`（统一内存按需分页），避免 DGX Spark 统一内存系统中 GPU 显存分配饿死 CPU 侧。普通上下文仍用 `cudaMalloc` 设备分配保持性能。通过 `ds4_gpu_should_use_managed_kv_cache()` 自动判断是否启用。

Metal Q4 Model View 回退

Metal Q4 expert tensor model views 的扩展（`DS4_METAL_MODEL_MAX_TENSOR_BYTES` 从 2 GiB 回到 ~672 MiB）已被回退——该修复由 AI 生成且未经用户确认，上游坚持不接受未经人工验证的修复。

4i. CUDA Prefill 性能优化

大幅优化长上下文 prefill 速度（32K prefill 255→346 tok/s，全曲线均值 316→370 tok/s），核心改动：

更宽的 WMMA 评分 **kernel**： 原有 `indexer_scores_wmma_kernel` 每 tile 处理 16 个压缩 token。新增 wmma32/64/128 变体（默认 128 宽，256 线程），将压缩索引预加载到 shared memory 后迭代所有注意力头，减少 kernel 启动开销并提高内存吞吐。

CUB BlockRadixSort 替代手写 **bitonic sort**： 新 `indexer_topk_8192_cub_kernel` 用 NVIDIA CUB 库的 `BlockRadixSort`（512 线程 × 16 items = 8192/block），将 float key 打包为 `uint64_t` 实现降序排序。通过 `cudaFuncSetAttribute` 请求额外 shared memory。

uint16_t 索引的 **bitonic sort**： 对 ≤65536 压缩 token 的场景，用 `uint16_t` 代替 `uint32_t` 索引，减半 shared memory 占用。

Top-K 预排序提升注意力访存： 新 `indexed_topk_sort_512_asc_kernel` 将 top-k 选出的 512 个索引按升序排列后再送入注意力 kernel，把对压缩 KV cache 的随机访问转变为顺序访问，显著改善内存合并。

Templatized 注意力 kernel： `attention_indexed_mixed_heads8_online_kernel` 重构为模板，参数化 `ROWS_PER_STAGE` 和 `HEADS_PER_GROUP` ，启动配置从 `<8, 8>` 改为 `<8, 16>` （16 head/group, 512 线程），配合动态 shared memory。

Shared memory 复用： 原有 wmma kernel 交换加载顺序，压缩索引 tile 只加载一次（在头循环之前），避免每头重复读取全局内存。

```
// ds4.h — 引导选项
typedef struct {
    // ...
    const char *directional_steering_file;
    float directional_steering_attn;
    float directional_steering_ffn;
} ds4_engine_options;
```

好/坏 prompt 对 → build_direction.py → 激活差异 good - bad → 归一化方向向量 43 × 4096 f32 → 推理时施加 y -= scale · d · (d·y)

5. 增量吞吐量基准测试（**ds4-bench**）

传统基准测试报告全流程平均速度，ds4-bench 在不同 context frontier 处测量瞬时吞吐量：

- 每个 frontier（如 2048, 4096, 6144...）只计算新增 token 区间的 prefill 速度
- prefill 后保存内存快照，做固定 128 token 贪心解码测 generation 速度
- 快照保存/恢复时间不计入测量窗口
- 输出 CSV：每个 frontier 的 prefill t/s、generation t/s、kvcache_bytes

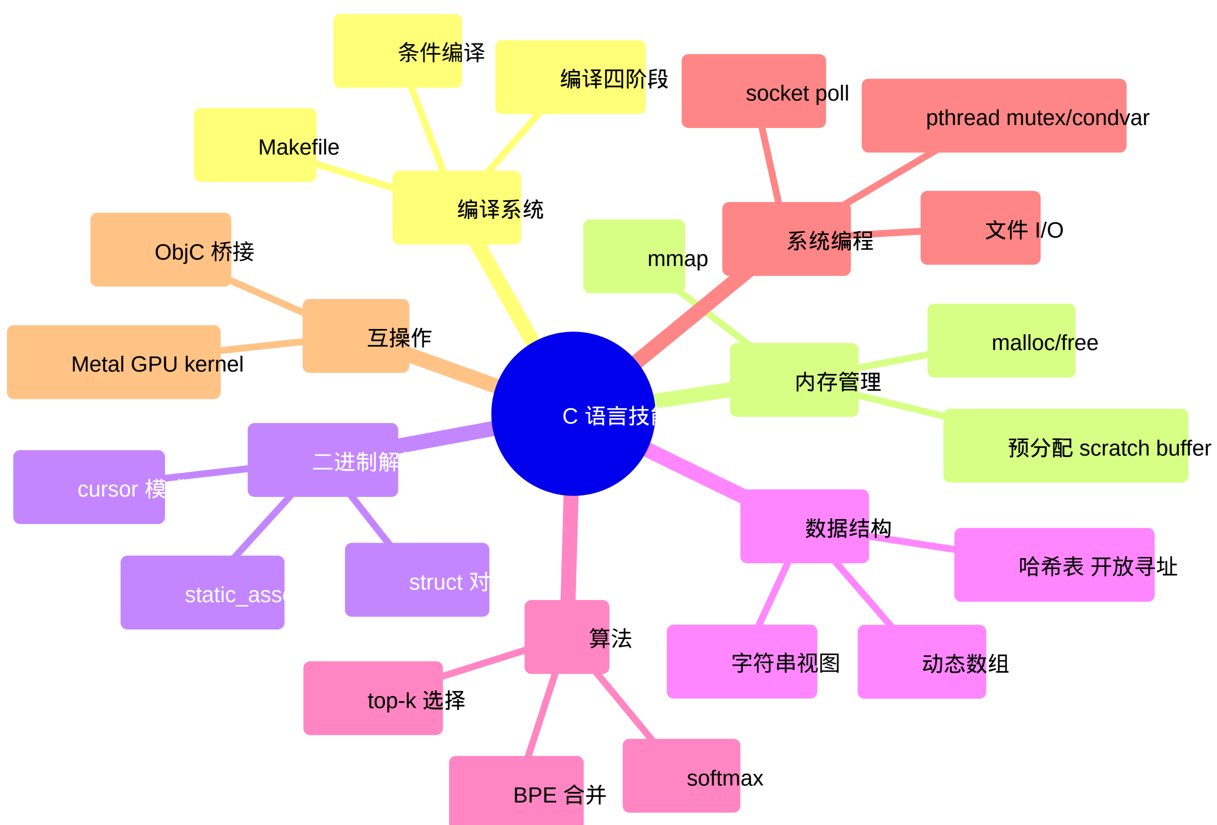
```
// ds4.h — 快照 API
typedef void (*ds4_session_progress_fn)(void *ud, int progress);
void ds4_session_set_progress(ds4_session *s, ds4_session_progress_fn fn, void *
ud);

void *ds4_session_save_snapshot(ds4_session *s);
int ds4_session_load_snapshot(ds4_session *s, void *snapshot);
void ds4_session_snapshot_free(void *snapshot);
```

测试数据使用 `speed-bench/promessi_sposi.txt` （23329 行意大利语公共领域文本）作为固定输入序列。 `bench/` 目录已重命名为 `speed-bench/` ，新增 M2 Ultra (192 GB) 和 M4 Max 基准数据，以及 `plot_speed.py` CSV→SVG 可视化脚本。

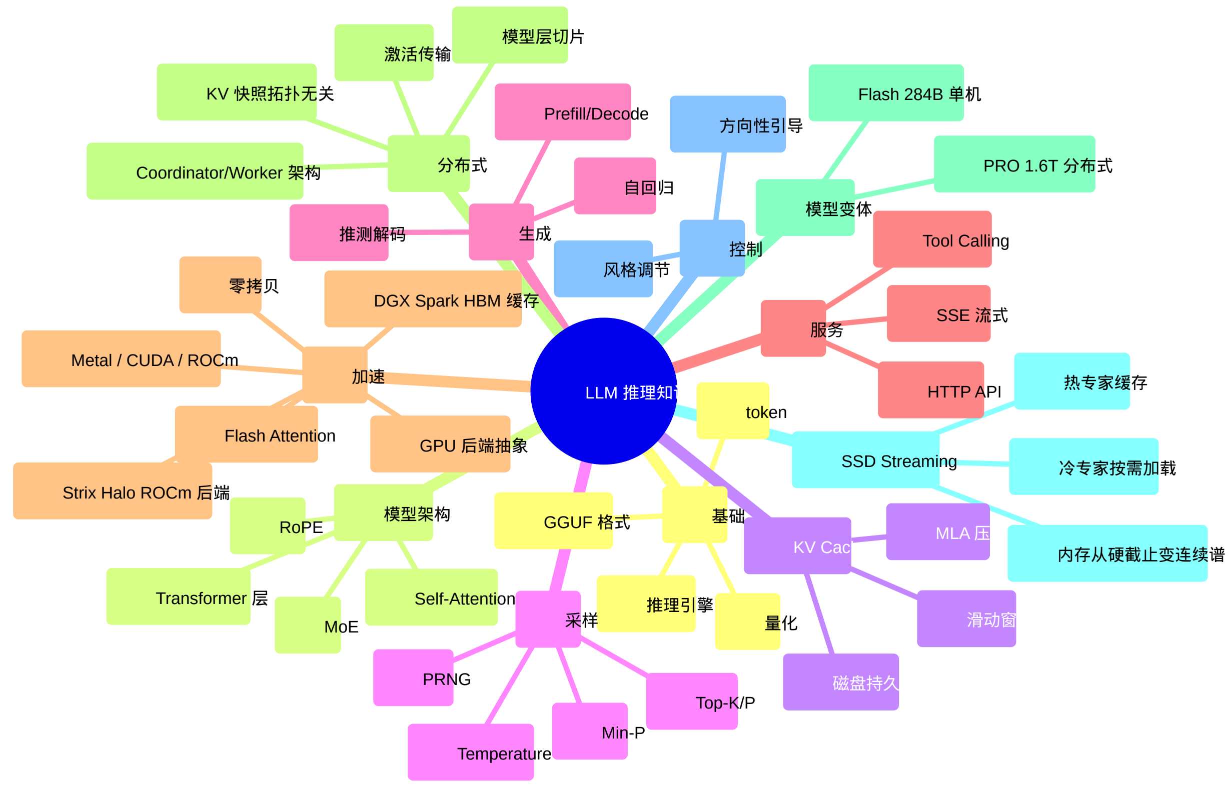
---

15 天学习总结

C 语言技能树



LLM 推理知识树



ds4.c 的设计哲学

1. 垂直整合：一个文件做所有事，零外部依赖
2. 专用化：只服务一个模型，硬编码所有参数
3. 零拷贝：mmap + Metal/CUDA no-copy buffer，数据永远不离开映射区
4. 热路径零分配：预分配 scratch buffer，生成循环中不 malloc
5. 早失败：加载时严格验证，不匹配就退出
6. 可扩展后端：通过 `ds4_gpu.h` 抽象接口，Metal/CUDA/ROCm/CPU 四后端可互换；融合算子作为可选后端钩子，新后端按需提供
7. **KV cache** 是一等公民：渲染文本前缀匹配 + 对齐前沿 + KTM 工具记忆，让长上下文推理跨重启存活
8. 方向性引导：推理时低秩编辑激活空间，无需重训练即可控制输出风格
9. 分布式推理：coordinator/worker 模型切片执行，PRO 模型可跨多台机器运行
10. **PRO** 模型支持：384 专家 Q4_K 量化路径，CPU/Metal/CUDA 三后端完整覆盖
11. **SSD Streaming**：路由专家缓存 + SSD 按需加载 + 路由热度淘汰（每 16 token 衰减、提示局部重置但缓存跨会话保温）+ 三路 prefill，让模型在有限 RAM 下也能运行
12. 合作式中断：Ctrl+C 在安全检查点优雅停止，KV cache 状态不丢失

反向链接

/glossary/distributed-inference

/glossary/kv-cache

/glossary/ssd-streaming

学习日志

Part 3: 模型架构

出链

Part 3: 模型架构

Part 6: 练习

---

1. GPU 后端 + 总结

练习 1：Metal 管线追踪

题目：描述一个 Metal kernel 从源码到执行的完整路径。

参考答案

```
1. .metal 源码 → 编译为 MTLLibrary（GPU 字节码）
   id<MTLLibrary> lib = [device newLibraryWithSource:...];

2. 从 library 提取 kernel 函数
   id<MTLFunction> fn = [lib newFunctionWithName:@"kernel_name"];

3. 创建计算管线（编译为 GPU 可执行的微码）
   id<MTLComputePipelineState> pipe = [device newComputePipelineStateWithF
unction:fn ...];

4. 创建命令缓冲区
   id<MTLCommandBuffer> cmd = [queue commandBuffer];

5. 编码计算命令（设置 buffer、分发线程组）
   [encoder setComputePipelineState:pipe];
   [encoder setBuffer:buf offset:0 atIndex:0];
   [encoder dispatchThreadgroups:grid threadsPerThreadgroup:group];

6. 提交执行
   [cmd commit];
   [cmd waitUntilCompleted];
```

练习 2：CPU vs GPU 对比

题目：对于矩阵乘法 [4096, 2048] × [2048, 4096]（约 160 亿 FLOP），比较 CPU 和 GPU 的处理方式。

参考答案

```
输出矩阵大小: 4096 × 4096 = 16,777,216 个元素
每个元素: 2048 次乘加 = 4096 FLOP
总 FLOP: 16.8M × 4096 ≈ 68.7B

CPU (8 核):
  每个 core 处理 ~2.1M 元素
  串行计算每个元素
  时间: ~68.7B FLOP / (8 × ~100 GFLOP/s) ≈ 0.09s

GPU (M3 Max, ~4000 核):
  每个 core 处理 ~4,200 元素
  所有 core 并行执行
  时间: ~68.7B FLOP / (~13 TFLOP/s) ≈ 0.005s

加速比: ~18× (理论值，实际受内存带宽限制)
```

练习 3：知识串联

题目：用一个请求的完整生命周期串联所有主题学到的知识。

> "用户发送 `POST /v1/chat/completions` 请求，得到流式回复。"

参考答案

```
HTTP:    服务器接收请求（socket → recv → 解析 HTTP）
HTTP:    JSON 解析请求体（手写递归下降解析器）
HTTP:    解析 messages、tools、temperature 等字段
分词:    Chat 模板渲染（特殊 token 组装）
分词:    BPE 分词（text → token IDs）
服务:    检查磁盘 KV cache（SHA1 匹配前缀）
加载:    模型权重通过 mmap 加载（零拷贝）
加载:    Metal GPU 框架初始化
推理:    Prefill 阶段（所有 prompt token 通过 43 层）
架构:    每层访问 weights_bind 绑定的权重指针
加载:    权重从 GGUF 张量数据中读取
架构:    量化权重解码（IQ2_XXS/Q2_K → float）
架构:    Self-Attention 计算（Q·K^T → softmax → V 加权）
架构:    KV Cache 写入（raw sliding window + MLA 压缩）
架构:    MoE 前向传播（路由 → 6 专家 → SwiGLU）
推理:    RoPE 旋转位置编码
推理:    Decode 循环（自回归，逐 token 生成）
采样:    采样（temperature + top-p → 选择 token）
分词:    Token 解码（ID → 文本）
HTTP:    SSE 流式发送给客户端
服务:    KV cache 保存到磁盘
```

今日学习检查清单

- ☐ 能描述 Metal kernel 从源码到执行的路径
- ☐ 理解 GPU 并行计算的优势
- ☐ 理解零拷贝 MTLBuffer 的工作原理
- ☐ 能解释 Flash Attention 的核心思想
- ☐ 能用完整请求生命周期串联 15 天的知识
- ☐ 能总结 ds4.c 的 6 大设计哲学

延伸挑战

挑战 1（中级）：**Metal vs CPU** 性能对比

分别用 `./ds4`（Metal 加速）和 `make cpu && ./ds4`（纯 CPU）运行相同的 prompt。对比 prefill 速度和 decode 速度。GPU 加速比是多少？在哪些阶段加速最明显？

挑战 2（高级）：端到端请求追踪（**capstone** 预习）

用 `lldb` attach 到 ds4-server，在 `client_main`、`ds4_tokenize_rendered_chat`、`prefill_layer_major_cpu`、`sample_full_vocab`、`sse_chunk` 五个函数设断点。发送一个 HTTP 请求，记录每个断点的调用栈和参数。写一份完整的"一个 HTTP 请求的一生"调试报告。

Part 6: 代码走读

---

1. GPU 后端 + 总结

追踪 1：Metal 初始化与管线编译

GPU 初始化

```
// 行 2661: 单例初始化 Metal 设备
int ds4_gpu_init(void) {
    g_device = MTLCreateSystemDefaultDevice();     // 获取 GPU 设备
    g_queue  = [g_device newCommandQueue];          // 命令队列

    // 加载 Metal Shading Library（编译好的 .metallib）
    // 从程序嵌入的 Metal 源码编译
    g_library = [g_device newLibraryWithSource:metal_source
                                       options:nil
                                         error:&err];

    // 初始化缓存字典
    g_model_buffer_cache  = [NSMutableDictionary new];  // 模型缓冲区
    g_pipeline_cache      = [NSMutableDictionary new];  // 管线状态
    g_transient_buffers   = [NSMutableArray new];        // 临时缓冲区
}
```

管线状态缓存

```
// 行 624: 按名称获取或创建计算管线
static id<MTLComputePipelineState> ds4_gpu_get_pipeline(
        const char *function_name) {
    NSString *key = @(function_name);

    // 查缓存
    id<MTLComputePipelineState> pipe = g_pipeline_cache[key];
    if (pipe) return pipe;                      // 命中

    // 缓存未命中: 从库中查找函数并编译管线
    id<MTLFunction> fn = [g_library newFunctionWithName:key];
    pipe = [g_device newComputePipelineStateWithFunction:fn
                                                   error:&err];
    g_pipeline_cache[key] = pipe;               // 存入缓存
    return pipe;
}
```

```
Metal 初始化层次:

MTLDevice (GPU 硬件抽象)
  └── MTLCommandQueue (命令提交队列)
        └── MTLCommandBuffer (一次提交的命令批次)
              └── MTLComputeCommandEncoder (计算命令编码器)
                    └── MTLComputePipelineState (着色器管线)
                          └── MTLFunction (从 .metal 源码编译)

管线缓存:
  "matmul_q8_0"        → pipeline_state_0
  "flash_attn_prefill" → pipeline_state_1
  "routed_moe"         → pipeline_state_2
  ...
  (~60 个推理内核)
```

追踪 2：零拷贝模型映射与张量管理

模型映射

```
// 行 4399: 将 mmap 的模型文件映射到 Metal 缓冲区
int ds4_gpu_set_model_map(const void *model_map, uint64_t model_size) {
    // 委托给 ds4_gpu_set_model_map_range
    // 在 macOS 上使用 MTLResourceStorageModeShared
    // GPU 和 CPU 共享同一块物理内存
    // → 零拷贝: 权重数据不需要从 CPU 复制到 GPU
}
```

张量分配

```
// 行 3767: 分配 GPU 张量
ds4_gpu_tensor *ds4_gpu_tensor_alloc(uint64_t bytes) {
    // 创建 Metal Buffer（共享内存模式）
    id<MTLBuffer> buffer = [g_device newBufferWithLength:bytes
                                                 options:MTLResourceStorageModeS
hared];
    // 包装为 DS4MetalTensor 对象
    DS4MetalTensor *obj = [[DS4MetalTensor alloc] init];
    obj.buffer = buffer;
    obj.offset = 0;
    obj.bytes  = bytes;

    // 追踪分配统计
    g_live_bytes  += bytes;
    g_peak_bytes   = MAX(g_peak_bytes, g_live_bytes);

    // __bridge_retained: Objective-C → C 所有权转移
    return (__bridge_retained ds4_gpu_tensor *)obj;
}
```

张量写入

```
// 行 3859: CPU → GPU 数据传输
int ds4_gpu_tensor_write(ds4_gpu_tensor *tensor, uint64_t offset,
                          const void *data, uint64_t bytes) {
    DS4MetalTensor *obj = (__bridge DS4MetalTensor *)tensor;
    // 验证边界
    if (offset + bytes > obj.bytes) return -1;
    // 直接 memcpy（共享内存模式，CPU/GPU 看到同一块内存）
    memcpy((uint8_t *)[obj.buffer contents] + obj.offset + offset,
           data, bytes);
    return 0;
}
```

```
零拷贝内存架构:

CPU 虚拟地址空间                    GPU
┌─────────────────┐
│ mmap 区域        │ ← 81GB GGUF 文件
│ (权重数据)       │
│                  │    MTLResourceStorageModeShared
│ ─ ─ ─ ─ ─ ─ ─ ─ ─│ ← 同一块物理内存
│                  │
│ Metal Buffer     │ ← ds4_gpu_tensor_alloc 分配
│ (激活值/输出)    │
└─────────────────┘

CPU 通过指针直接读写 → GPU 通过同一物理地址访问
无需 cudaMemcpy / [buffer didModifyRange]
```

追踪 3：GPU matmul 与 matvec

Q8_0 量化矩阵乘

```
// 行 4919: Q8_0 权重 × Q8_0 激活的矩阵乘
int ds4_gpu_matmul_q8_0_tensor(
        ds4_gpu_tensor *out,            // 输出张量
        const void *model_map,          // 模型 mmap 基地址
        uint64_t model_size,
        uint64_t weight_offset,         // 权重在 mmap 中的偏移
        uint64_t in_dim, uint64_t out_dim,
        const ds4_gpu_tensor *x,        // 输入激活张量
        uint64_t n_tok) {              // token 数量

    // 要求 in_dim % 32 == 0（Q8_0 块大小对齐）
    // 获取 Metal 缓冲区
    id<MTLBuffer> out_buf  = ...;
    id<MTLBuffer> x_buf    = ...;
    id<MTLBuffer> w_buf    = model_map + weight_offset;  // 零拷贝权重

    // 设置计算管线和参数
    id<MTLComputePipelineState> pipe = ds4_gpu_get_pipeline("matmul_q8_0");
    id<MTLComputeCommandEncoder> enc = [cb computeCommandEncoder];
    [enc setComputePipelineState:pipe];
    [enc setBuffer:out_buf offset:0 atIndex:0];
    [enc setBuffer:w_buf  offset:0 atIndex:1];
    [enc setBuffer:x_buf  offset:0 atIndex:2];
    // ... 设置其他参数 ...

    // 分发计算线程
    [enc dispatchThreadgroups:threadgroups
           threadsPerThreadgroup:threads];
    [enc endEncoding];
}
```

F16 矩阵乘

```
// 行 5141: FP16 权重矩阵乘（用于 embedding 等未量化的权重）
int ds4_gpu_matmul_f16_tensor(
        ds4_gpu_tensor *out,
        const void *model_map, uint64_t model_size,
        uint64_t weight_offset,
        uint64_t in_dim, uint64_t out_dim,
        const ds4_gpu_tensor *x,
        uint64_t n_tok) {
    // 与 Q8_0 版本结构相同
    // 但使用 "matmul_f16" 着色器管线
    // 无 in_dim 对齐要求
}
```

```
GPU matvec 线程分发:

权重矩阵 [out_dim × in_dim]
    │
    ▼ 分割为 threadgroups
    │
    每个 threadgroup:
      处理一个输出行的子范围
      加载权重块到共享内存
      与输入向量做分块点积
      累加结果

threadgroup 大小: 例如 [64, 1, 1]
总 threadgroups: [ceil(out_dim/64), n_tok, 1]
```

追踪 4：GPU Flash Attention

Prefill 注意力

```
// 行 10569: Prefill 阶段的 Flash Attention
int ds4_gpu_attention_prefill_raw_heads_tensor(
        ds4_gpu_tensor *heads,          // 输出: [n_head × head_dim]
        const void *model_map,
        uint64_t model_size,
        uint64_t sinks_offset,          // Sink 参数偏移
        const ds4_gpu_tensor *q,        // Query [n_tok × n_head × head_dim]
        const ds4_gpu_tensor *raw_kv,   // KV 缓存 [n_tokens × head_dim]
        uint32_t n_tokens,              // KV 缓存行数
        uint32_t window,                // SWA 窗口大小
        uint32_t n_head, uint32_t head_dim) {

    // 包装 Sink 参数（零拷贝）
    ds4_gpu_wrap_model_range(model_map, sinks_offset, ...);

    // 获取专用 Flash Attention 管线
    ds4_gpu_encode_flash_attention_prefill_raw_heads(
        cb, pipe, heads, q, raw_kv,
        n_tokens, window, n_head, head_dim);
}
```

Decode 注意力

```
// 行 10619: Decode 阶段的 Flash Attention
int ds4_gpu_attention_decode_raw_batch_heads_tensor(
        ds4_gpu_tensor *heads,
        const void *model_map, uint64_t model_size,
        uint64_t sinks_offset,
        const ds4_gpu_tensor *q,
        const ds4_gpu_tensor *raw_kv,
        uint32_t n_tokens,
        uint32_t pos0,                  // 序列起始位置
        uint32_t n_raw,                 // raw 缓存行数
        uint32_t raw_cap,               // raw 缓存容量
        uint32_t raw_start,             // raw 缓存起始索引
        uint32_t window,
        uint32_t n_head, uint32_t head_dim) {

    // 额外参数管理 KV 缓存窗口:
    // pos0:       当前 token 在完整序列中的位置
    // n_raw:      实际有效的 raw KV 行数
    // raw_cap:    raw 缓存的总容量
    // raw_start:  循环缓冲区的起始索引

    ds4_gpu_encode_flash_attention_decode_raw_batch_heads(
        cb, pipe, heads, q, raw_kv,
        n_tokens, pos0, n_raw, raw_cap, raw_start, ...);
}
```

```
Flash Attention GPU 实现:

标准注意力:                Flash Attention:
  1. Q×K^T → scores          1. 分块加载 Q, K, V 到共享内存
  2. softmax(scores)         2. 在线 softmax（分块计算最大值和指数和）
  3. × V → output            3. 分块累加输出
  内存: O(n²)               4. 不显式存储完整的 attention 矩阵
                             内存: O(n)

Prefill: 所有 prompt token 的 Q 对所有 KV 做注意力
Decode:  1 个新 token 的 Q 对所有 KV 做注意力
```

追踪 5：GPU MoE 与 HC 操作

路由 MoE

```c
// 行 12736: GPU 上的路由 MoE
int ds4_gpu_routed_moe_one_tensor(
        ds4_gpu_tensor *out,
        ds4_gpu_tensor *gate, ds4_gpu_tensor *up,
        ds4_gpu_tensor *mid,  ds4_gpu_tensor *experts,
        const void *model_map, uint64_t model_size,
        uint64_t gate_offset, uint64_t up_offset,
        uint64_t down_offset,
        uint32_t gate_type, uint32_t down_type,
        uint64_t gate_expert_bytes, uint64_t gate_row_bytes,
        uint64_t down_expert_bytes, uint64_t down_row_bytes,
        uint32_t expert_in_dim, uint32_t expert_mid_dim,
        uint32_t out_dim,
        const ds4_gpu_tensor *selected,    // 6 个专家索引
        const ds4_gpu_tensor *weights,     // 6 个专家权重
        uint32_t n_expert,                 // 6
        float clamp,
        const ds4_gpu_tensor *x) {         // 输入

    // 对齐要求:
    // expert_in_dim  % 256 == 0
    // expert_mid_dim % 256 == 0

    // 验证 8 个 Metal 缓冲区大小
    // gate, up, mid, out, experts, selected, weights, x

    // GPU 内核执行:
    // 1. 6 个专家的 gate+up 投影（并行）
    // 2. SwiGLU 激活
    // 3. 6 个专家的 down 投影（并行）
    // 4. 加权累加
}
```

HC Split Sinkhorn

```
// 行 13503: GPU 上的 HC 分解
int ds4_gpu_hc_split_sinkhorn_tensor(
        ds4_gpu_tensor *out,
        const ds4_gpu_tensor *mix,         // 混合系数
        const void *model_map,
        uint64_t model_size,
        uint64_t scale_offset,             // 缩放参数偏移
        uint64_t base_offset,              // 基础偏置偏移
        uint32_t n_hc,                     // HC 维度
        uint32_t sinkhorn_iters,           // Sinkhorn 迭代次数
        float eps) {

    // 计算输出大小
    uint32_t mix_hc = 2 * n_hc + n_hc * n_hc;  // pre + post + comb
    // out 大小 = mix_hc × sizeof(float)

    // 验证 n_hc ∈ [1, 16]

    // 包装模型范围（零拷贝）
    ds4_gpu_wrap_model_range(model_map, scale_offset, ...);
    ds4_gpu_wrap_model_range(model_map, base_offset, ...);

    // 获取管线并分发
    id<MTLComputePipelineState> pipe =
        ds4_gpu_get_pipeline("hc_split_sinkhorn");
}
```

追踪 6：GPU 命令提交与同步

命令缓冲区管理

```
// 行 3903: 开始新的命令批次
int ds4_gpu_begin_commands(void) {
    g_batch_cb = [g_queue commandBuffer];       // 从队列获取命令缓冲区
    return 0;
}

// 行 3910: 刷新（提交当前批次，立即开始新批次）
int ds4_gpu_flush_commands(void) {
    [g_batch_cb commit];                        // 提交到 GPU
    [g_pending_cbs addObject:g_batch_cb];       // 追踪待完成命令
    g_batch_cb = [g_queue commandBuffer];        // 立即创建新批次
    return 0;
}

// 行 3929: 结束命令批次
int ds4_gpu_end_commands(void) {
    [g_batch_cb commit];
    [g_pending_cbs addObject:g_batch_cb];
    g_batch_cb = nil;
    return 0;
}

// 行 3937: 等待所有 GPU 操作完成
int ds4_gpu_synchronize(void) {
    if (g_batch_cb) ds4_gpu_end_commands();     // 先结束当前批次
    for (id<MTLCommandBuffer> cb in g_pending_cbs)
        [cb waitUntilCompleted];                // 等待每个命令完成
    [g_pending_cbs removeAllObjects];
    return 0;
}
```

```
GPU 命令流水线:

CPU 时间线:
  begin → 编码命令 → flush → 编码更多 → flush → end → synchronize
          |              |                |          |        |
          ▼              ▼                ▼          ▼        ▼
GPU 时间线:
       [执行批次1]    [执行批次2]     [执行批次3]  [等待完成]

flush 的作用:
  - 提交已编码的命令，让 GPU 立即开始执行
  - 同时创建新的命令缓冲区继续编码
  - CPU/GPU 重叠工作，隐藏延迟

synchronize:
  - 等待所有批次完成
  - 通常在需要读取 GPU 输出时调用
  - 例如: 采样前需要 logits 已经写好
```

追踪 7：CUDA Prefill 性能优化

宽 WMMA 评分 Kernel

CUDA 后端使用 NVIDIA WMMA（Warp Matrix Multiply-Accumulate）张量核心进行压缩索引评分：

```
// 原有: 16 压缩 token/tile (16x16 WMMA)
indexer_scores_wmma_kernel

// 新增变体（默认使用 128 宽）:
indexer_scores_wmma32_kernel   // 32/tile, 64 threads
indexer_scores_wmma64_kernel   // 64/tile, 128 threads
indexer_scores_wmma128_kernel  // 128/tile, 256 threads

// 优化: 压缩索引预加载到 shared memory，只加载一次
// 然后迭代所有注意力头，避免每头重复读全局内存
```

CUB BlockRadixSort Top-K

```
// 替代手写 bitonic sort，处理 n_comp <= 8192
// 512 threads × 16 items = 8192 items per block
indexer_topk_8192_cub_kernel

// Float → uint64_t key packing 实现降序排序:
// key = (topk_float_ordered_key(v) << 32) | (0xffffffff - idx)
// 整数降序 = float 降序

// 动态 shared memory:
cudaDevAttrMaxSharedMemoryPerBlockOptin  // 查询设备上限
cudaFuncSetAttribute(..., shared_size)   // 请求额外 shared memory
```

Top-K 预排序优化访存

```
// 将 512 个 top-k 索引按升序排列后再送入注意力 kernel
indexed_topk_sort_512_asc_kernel
// 随机访问 → 顺序访问，改善压缩 KV cache 的内存合并
```

实测：32K prefill 255→346 tok/s（+36%），全曲线均值 316→370 tok/s（+17%）。

追踪 8：Metal NAX 加速

Neural Accelerator 检测

```
// ds4_metal.m — M5 NAX 检测
// 启动时检查系统属性中的 M5 标志
g_metal4_m5_neural_accelerators_hint = ...;  // 读取设备属性

// NAX kernel 选择逻辑：
if (g_metal4_m5_neural_accelerators_hint && n_tokens >= 16) {
    // 使用 NAX 加速路径
    // indexer_scores: 卸载到 Neural Accelerator
    // MoE expert matmul: Q8_0 → NAX tensor matmul
} else {
    // 标准金属计算路径
}
```

NAX kernel 自动在 M5 设备上启用，无需用户配置。在非 M5 设备上编译安全（条件分支指向标准路径）。

总结：ds4 代码架构全景

```
ds4.c (18189 行) — 推理引擎核心
├── 量化内核 (158-3990)        量化与矩阵运算
│   ├── block_q2_K/q4_K/q8_K/iq2_xxs
│   ├── matvec_q8_0 / matvec_q2_k / matvec_iq2_xxs
│   └── f16_to_f32 / fp8 量化
├── 线程池 (612-770)           推理流程
├── 注意力机制 (2700-7092)     模型架构
│   ├── RMS Norm / RoPE        推理流程
│   ├── Q/KV 投影 / Flash Attention
│   ├── KV 缓存管理
│   └── 混合注意力 (raw + comp)
├── MoE / FFN (5012-6031)      模型架构
│   ├── SwiGLU 激活
│   ├── Hash/Top-K 路由
│   └── 共享专家 + 路由专家
├── 推理循环 (6055-7919)       推理流程
│   ├── Prefill (层主序)
│   └── Decode (逐 token)
├── 分词器 (13438-14165)       分词与采样
├── 采样 (14874-15083)         分词与采样
└── 公共 API (15716-17440)     模型架构

ds4_server.c (10349 行) — HTTP 服务
├── JSON 解析器 (141-437)      服务层
├── HTTP/SSE (3080-3180)       服务层
├── API 兼容层 (735-4259)      服务层
├── KV 持久化 (5138-6195)      服务层
├── SHA-1 (5192-5270)          服务层
└── 服务器架构 (4717-8100)     服务层

ds4_metal.m (14647 行) — GPU 后端
├── 初始化与管线 (624-2661)    GPU 加速
├── 张量管理 (3767-3937)       GPU 加速
├── 矩阵运算 (4919-5160)       GPU 加速
├── Flash Attention (10569+)   GPU 加速
├── MoE / HC (12736+)          GPU 加速
└── Metal Shader (.metal)       编译到 .metallib
```

> 注意力锚点——一种让模型"学会不关注任何位置"的机制，防止注意力在无关上下文上分散。

为什么需要 Attention Sink

在长上下文中，模型需要对当前位置之前的所有 token 计算注意力权重。但有些位置的信息并不相关，强制分配注意力会降低效果。Attention Sink 给模型一个"退出"选项：通过一个可学习的 sink logit，让模型在 softmax 时可以"不关注"任何真实 token。

核心原理

```
标准注意力: softmax(Q·K^T) → 每个位置获得非零权重
Attention Sink: 额外添加一个 sink logit 到 softmax 分母
    → softmax 可以把大量概率分配给 sink（相当于"不关注"）
    → 真实位置的注意力权重被稀释，减少噪声
```

在 ds4.c 中的实现

在注意力计算中，每个头有一个可学习的 sink logit，参与 softmax 分母计算但不贡献 value 向量。这让模型学会了"注意力分流"的能力——在信息不相关的位置主动降低权重。

相关概念

- softmax — sink logit 参与 softmax 归一化
- kv-cache — sink 的 KV 存储位置
- moe — 注意力和 MoE 在同一层前向中串联

详见 Part 3 — 注意力机制。

出链

Part 3: 模型架构

> Byte Pair Encoding，字节对编码 —— LLM 最常用的分词算法，将文本拆分为模型词表中的 token。

为什么需要分词

模型只能处理数字。文本必须先变成 token ID 序列，才能做 Embedding 查表和后续计算。分词器决定"怎么切"直接影响模型能力：

- 切得太细（按字符）→ 序列太长，注意力开销大
- 切得太粗（按整词）→ 词表巨大，出现新词就废了
- BPE 在两者之间自动平衡

核心原理

```
训练阶段: 从字符级开始，迭代合并最高频的相邻 token 对
  "h e l l o" → 合并 "ll" → "h e ll o" → 合并 "hel" → "hel lo"

推理阶段: 按学到的合并规则，从长到短匹配切分
  "hello world" → [hello, _world] → [token_id_42, token_id_1089]
```

ds4.c 实现了 **GPT-2** 字节级 **BPE**：所有 256 个字节值通过 `gpt2_byte_to_codepoint` 映射为可见 Unicode 码点（不可打印字节如 0x00 映射到 256+ 码点），这样任何二进制输入都能分词。

预分词规则：

- 数字按三位分组（`12345` → `123,45`，从左往右贪婪匹配）
- CJK 字符单独切分
- 空格与后续字符合并（`_hello`）

在 **ds4.c** 中的实现

```
// ds4.c — 分词主函数
ds4_tokenize_rendered_chat()    // 入口：渲染后的文本 → token ID 数组

// 关键步骤
bpe_emit_piece()                // 迭代查找最小 rank 的相邻对并合并
gpt2_byte_to_codepoint()        // 字节 → Unicode 码点映射
```

词表大小 129,280 个 token，对中英文都有良好覆盖。更大的词表意味着中文编码效率高，同样的上下文窗口能装更多内容。

相关概念

- gguf — 词表存储在 GGUF 文件的元数据中
- softmax — 分词后 token ID 经 Embedding 查表，最终通过 softmax 采样选出下一个 token

详见 Part 2 — BPE 分词器。

出链

Part 2: 分词与采样

> GPT-Generated Unified Format —— llama.cpp 生态的二进制模型文件格式，一个文件包含完整的模型参数和元数据。

为什么需要专用文件格式

训练好的模型有几十 GB 的参数（张量）和大量配置信息（层数、维度、词表等）。如果用 JSON 存元数据 + 单独文件存权重，加载慢且容易版本不一致。GGUF 把所有东西打包成一个紧凑的二进制文件，支持零拷贝 mmap 加载。

文件结构

```
┌──────────────────────────────┐
│ Header (24 bytes)            │  magic "GGUF" + version + 张量数 + KV 数
├──────────────────────────────┤
│ 元数据 KV 表                 │  模型名、架构参数（层数/维度/词表...）
│  string/int/float/array      │  零拷贝：ds4_str 直接指向 mmap 区域
├──────────────────────────────┤
│ 张量目录                     │  每个张量的名称、维度、类型、偏移量
├──────────────────────────────┤
│ 对齐填充                     │
├──────────────────────────────┤
│ 张量数据                     │  实际权重，支持多种量化格式
│  Q2_K / Q4_K / Q8_K / ...    │ 通过 mmap 按需加载，不全部读入内存
└──────────────────────────────┘
```

在 ds4.c 中的实现

```
// ds4.c — 游标模式（cursor）解析 GGUF
ds4_cursor cur = {data, end};         // 初始化游标，指向 mmap 内存
uint32_t magic = cursor_u32(&cur);    // 读 4 字节，游标自动前进
ds4_str name  = cursor_string(&cur); // 零拷贝：返回 {ptr, len}，不复制字符串
```

关键设计：

- 游标模式：`cursor_u32` / `cursor_string` / `cursor_skip` 等函数顺序读取，比 fseek + fread 简洁得多
- 零拷贝字符串：`ds4_str` 是 `{ptr, len}` 结构体，直接指向 mmap 区域
- 尽早失败：`config_validate_model` 执行 30+ 参数校验，加载阶段就发现不兼容的模型
- 按需加载：张量数据留在内核页缓存中，推理需要时才触发 page fault

相关概念

- mmap — GGUF 通过 mmap 加载，张量数据零拷贝访问
- page-fault — 大模型不会全部读入内存，靠 page fault 按需加载
- xmalloc — 游标和元数据结构使用 xmalloc 分配

详见 Part 1 — GGUF 二进制格式。

出链

Part 1: 构建与加载

> 哈希表——O(1) 平均查找的数据结构。ds4.c 使用开放寻址哈希表实现 BPE 分词查找和 DSML replay map。

为什么需要哈希表

BPE 分词时需要在 129280 个 token 的词表中快速查找子串。线性搜索太慢（O(n)），二分搜索需要排序（O(log n)）。哈希表提供 O(1) 平均查找。

核心原理

```
开放寻址法（Open Addressing）：
  hash(key) → slot
  如果 slot 被占用且 key 不同 → 线性探测下一个 slot
  直到找到空 slot（插入）或匹配 key（查找）

vs. 链式哈希（Chaining）：
  每个 slot 是一个链表，冲突时追加到链表
  开放寻址缓存更友好（连续内存），适合 C 实现
```

在 **ds4.c** 中的实现

BPE 分词器中的哈希表（ds4.c）用于 token 词表查找：

- 开放寻址 + 线性探测
- 固定大小（词表大小 × 负载因子）
- 初始化时插入所有 token，之后只做查找

DSML replay map 使用 `rax.c`（Redis 的 radix tree 实现），不是哈希表，但用途类似——快速查找 token 序列到函数调用的映射。

相关概念

- bpe — BPE 分词依赖哈希表在词表中快速查找合并规则
- moe — 前 3 层的 hash 路由使用简单的 ID 查找表

详见 Part 2 — BPE 分词器。

出链

Part 2: 分词与采样

> 缓存注意力机制的历史 Key/Value，避免每步重复计算，是自回归推理的核心优化。

为什么需要 **KV Cache**

自回归生成每步产生一个 token，下一步需要用所有已生成的 **token**（包括历史）做注意力计算。如果不缓存，第 N 步要重新计算 N 组 K 和 V，总计 1+2+3+…+N = N(N+1)/2 次 KV 计算。生成 1000 个 token 就是约 50 万次重复计算。

缓存后每步只算 1 组新 KV，总计 N 次。

核心原理

```
步骤 1: 输入 [A]        → 算 K1V1, 缓存 → 输出 token B
步骤 2: 输入 [B]        → 算 K2V2, 追加  → 用 K1V1K2V2 做注意力 → 输出 token C
步骤 3: 输入 [C]        → 算 K3V3, 追加  → 用 K1V1K2V2K3V3 做注意力 → 输出 token D
```

ds4.c 的 KV Cache 由两部分组成：

类型	作用	大小
原始 KV（raw）	最近 128 个 token 的完整 KV	每层 128 × 512 × 4B = 256KB
压缩 KV（compressed）	128 之前的历史，经 MLA 压缩	按 ratio 缩减，可低至原始的 1/64

窗口满时通过 `memmove` 滑动移除最旧行，保证固定内存占用。

在 **ds4.c** 中的实现

```
// KV cache push（ds4.c 约 7273 行）
kv_cache_push_raw(cache, scratch->kv);
```

关键函数：

- `kv_cache_push_raw()` — 追加新 KV 行，窗口满时滑动
- `kv_cache_push_comp()` — 追加压缩 KV 行
- `layer_attention_rows_one()` — 在 raw + compressed 上混合做注意力

DeepSeek 使用 Multi-head Latent Attention (MLA)，KV 通过低秩投影压缩存储，且所有 Q 头共享 1 组压缩 KV，KV Cache 内存比标准 Multi-Head 大幅缩小。

相关概念

- softmax — 注意力分数的归一化
- moe — MoE 层和注意力层各自使用 KV Cache
- rope — K 在缓存前需先经 RoPE 旋转编码位置
- mmap — KV Cache 可持久化到磁盘，复用历史前缀

磁盘持久化

KV Cache 可以序列化到磁盘文件（`<sha1>.kv`），下次请求时用文本前缀匹配复用历史计算。持久化逻辑已从 `ds4_server.c` 提取为独立模块 `ds4_kvstore.c/h`。

磁盘缓存有容量预算（`--kv-cache-budget-mb`），满了通过评分公式驱逐：

```
score = (effective_hits + 1) × tokens / file_size
```

`effective_hits` 按 6 小时半衰期衰减，防止过时的热门条目挤占新条目。驱逐时保护刚写入的文件不被立即删除。

预存储驱逐

驱逐在写入新条目之前执行，并传入待写入条目的上下文（`model_id`、`quant_bits`、`ctx_size`、`text`），避免新条目成为自己的受害者。当新条目是旧条目的严格前缀扩展时，旧条目自动获得评分惩罚。

兼容性加固

加载磁盘缓存时严格校验 `model_id`、`quant_bits`、`ctx_size` 以及文件内容的 SHA1 完整性。不兼容的文件自动删除重建，避免不干净关机后缓存"看似正常但 prefill 总失败"的循环。

分布式 **KV** 快照

分布式推理模式下，KV 快照是拓扑无关的：保存时聚合所有 worker 的层张量为标准 DSV4 载荷，加载时按当前路由分发。保存的文件在任何 worker 配置下都可复用。

Metal 后端支持 `comp_kv_f16` 模式——压缩 KV cache 以 float16 存储，每行内存减半，详见 Part 6 — GPU 加速。

详见 Part 5 — KV Cache 持久化。

详见 Part 3 — 注意力机制 和 端到端推理流程。

出链

Part 6: GPU 加速

Part 5: 服务层

Part 3: 模型架构

> Low-Rank Adaptation——低秩自适应，用小矩阵分解替代大权重矩阵，实现参数高效微调或低存储推理。

为什么需要 LoRA

全量微调需要更新模型所有参数，284B 参数的模型微调成本极高。LoRA 的思路：冻结原始权重，只训练两个小矩阵的乘积来近似权重更新。

在 ds4.c 中，LoRA 不仅用于微调，还被用于 QKV 投影——将高维投影分解为低秩形式，减少推理时的矩阵运算量。

核心原理

```
标准线性层:  y = x × W         W 是 [4096 × 32768] = 134M 参数
LoRA 分解:    y = x × (A × B)  A 是 [4096 × r], B 是 [r × 32768]
                               r << min(4096, 32768) 时参数量大幅减少

例: r = 512 → 4096×512 + 512×32768 = 2M + 16.7M = 18.7M (vs 134M)
```

在 ds4.c 中的实现

QKV 投影使用 LoRA 分解：`W_qkv ≈ A_qkv × B_qkv`，将 4096→32768 的直接投影分解为两步小矩阵乘法。权重以低秩形式存储在 GGUF 文件中，推理时逐步还原。

相关概念

- moe — MoE 专家也使用低秩结构优化
- quantization — LoRA 矩阵和量化结合进一步减少存储
- gguf — LoRA 权重作为独立张量存储在 GGUF 中

详见 Part 3 — 公共 API 与权重结构。

出链

Part 3: 模型架构

> Multi-head Latent Attention——DeepSeek 的压缩注意力机制，将 KV Cache 压缩到低维潜在空间，大幅减少缓存内存。

为什么需要 MLA

标准 Multi-Head Attention 中，每个 token 需要缓存所有头的 K 和 V 向量。DeepSeek V4 有 64 个 Q 头，head_dim = 512。MLA 的思路：不缓存完整的 **K/V**，而是缓存一个低维"潜在表示"，需要时再投影回高维。

核心原理

DeepSeek V4 的 KV Cache 节省来自两个机制叠加：

1. **GQA（Grouped-Query Attention）**：`n_head_kv = 1`，64 个 Q 头共享 1 组 KV，将标准 KV 从 `64×512×2 = 65536` 压缩到 `1×512×2 = 1024` floats
2. **MLA** 低秩压缩：KV 投影到 512 维潜在空间，每个位置存 512 floats（而非 1024）
3. 序列压缩：部分层按 ratio-4 或 ratio-128 压缩序列长度（多个 token 共享一个压缩位置）

```
标准 Multi-Head KV Cache:  64 × 512 × 2 = 65536 floats/token/layer
仅 GQA:                     1 × 512 × 2  = 1024 floats/token/layer
MLA (GQA + 低秩压缩):      1 × 512      = 512 floats/token/layer
序列压缩后 (ratio=128):    512 / 128     ≈ 4 floats/token/layer
```

DeepSeek V4 Flash 使用 512 维的潜在表示，配合序列压缩，使得百万级上下文在本地成为可能。

在 ds4.c 中的实现

KV Cache 存储压缩后的潜在向量（`attn_comp_kv`），以及压缩器的状态（`attn_state_kv`）。推理时通过投影矩阵将潜在表示解压为完整的 K/V 向量。

相关概念

- kv-cache — MLA 是 KV Cache 的压缩变体
- softmax — 解压后的 KV 用于注意力计算
- moe — 注意力层和 MoE 层在每层前向中串联

详见 Part 3 — 注意力机制。

出链

Part 3: 模型架构

> Mixture of Experts，混合专家 —— 把一个大 FFN 拆成 256 个小网络，每个 token 只激活 6 个，用"宽度换深度"实现超大参数量但低计算量。

为什么需要 **MoE**

增大模型参数能提升能力，但每次推理都算遍所有参数的话，计算量随参数线性增长。284B 参数的模型在消费级硬件上无法实时推理。

MoE 的思路：参数可以多，但每次只用一小部分。就像一个公司有 256 个专家，每次只派 6 个相关领域的专家处理任务。

核心原理

```
输入 token 向量 x
    │
    ↓  Router: x × W_router → 256 个分数 → √softplus → 选 Top-6
    │
    ├→ 专家 17 (权重 0.3): SwiGLU(x) → out₁₇
    ├→ 专家 42 (权重 0.25): SwiGLU(x) → out₄₂
    ├→ 专家 89 (权重 0.2): SwiGLU(x) → out₈₉
    ├→ 专家 134 (权重 0.1): SwiGLU(x) → out₁₃₄
    ├→ 专家 201 (权重 0.08): SwiGLU(x) → out₂₀₁
    ├→ 专家 248 (权重 0.07): SwiGLU(x) → out₂₄₈
    └→ 共享专家: SwiGLU(x) → out_shared  (所有 token 必经)
    │
    ↓  output = Σ(权重ᵢ × outᵢ) + out_shared
```

关键数字：

- 256 个路由专家 + 1 个共享专家
- 每次激活 6/256 ≈ 2.3% 的路由专家
- 43 层 × 6 = 258 次专家调用，通过残差连接逐步积累知识

在 ds4.c 中的实现

```
// Router 打分（ds4.c 约 5023 行）
matvec_any(logits, model, layer->ffn_gate_inp, x);  // x × W → 256 分数

// 路由激活（ds4.c 约 5035 行）
for (int i = 0; i < DS4_N_EXPERT; i++)
    probs[i] = sqrtf(softplus_stable(logits[i]));

// Top-K 选择（ds4.c 约 5065 行）
topk_desc(selection, DS4_N_EXPERT, DS4_N_EXPERT_USED, selected);  // 256 选 6
```

两种路由策略：

层	策略	原理
前 3 层	Hash 路由	token ID → 查找表 → 固定专家，零计算开销
第 4-42 层	Top-K 路由	线性层算分数 → 选最高的 6 个

非对称量化：路由专家用 IQ2_XXS/Q2_K（~2 bit），共享专家用 Q8_0（8 bit），因为共享专家每个 token 都用，需要更高精度。

PRO 模型的 MoE

DeepSeek V4 PRO 是更大的模型变体（`DS4_VARIANT_PRO`），MoE 规模远大于 Flash：

- **384** 个路由专家（vs Flash 的 256），每个 token 仍激活 6 个
- 路由专家全部使用 **Q4_K** 量化（~4.5 bit），比 Flash 的 IQ2_XXS 精度更高
- 61 层（vs Flash 的 43 层），FFN 中间维度 3072
- 需要通过 分布式推理 跨多台机器运行

相关概念

- softmax — Router 用 sqrt(softplus(x)) 归一化分数（非标准 softmax）
- kv-cache — MoE 层和注意力层在单层前向中串联执行

- gguf — 不同专家的权重使用不同量化级别存储在 GGUF 中
- ssd-streaming — 利用 MoE 稀疏激活，只缓存热专家到 RAM，冷专家从 SSD 按需加载

详见 Part 3 — MoE 混合专家 和 LLM 核心架构概念 的 MoE 章节。

出链

Part 3: 模型架构

> Pseudo-Random Number Generator——伪随机数生成器，推理采样的随机性来源。ds4.c 使用 SplitMix64 算法。

为什么需要 PRNG

LLM 采样需要随机性：给定概率分布 `[0.1, 0.3, 0.05, 0.55]`，需要按概率随机选择一个 token。PRNG 生成均匀随机数，然后通过"轮盘赌"算法映射到概率分布。

核心原理

```
SplitMix64 状态转移:
  state ^= state >> 12
  state ^= state << 25
  state ^= state >> 27
  output = state * 0x2545f4914f6cdd1d  // 乘法混合输出

特性:
  - 64 bit 状态，周期 2^64
  - 3 次位操作 + 1 次乘法，极快
  - 通过所有标准统计测试
  - 不可用于密码学（可预测）
  - 全零种子时用黄金比例常数 0x9e3779b97f4a7c15 替代
```

轮盘赌采样：

```
1. 生成 [0, 1) 均匀随机数 r
2. r *= sum(probs)     // 缩放到概率总和
3. 从第一个 token 开始累减: r -= probs[i]
4. 当 r ≤ 0 时返回当前 token i
```

在 **ds4.c** 中的实现

`sample_rng_next()` 函数封装 SplitMix64 状态。`sample_rng_f32()` 生成 [0,1) 浮点数用于采样。`seed` 参数控制可重现性——相同 seed 产生相同输出。

相关概念

- softmax — PRNG 用于从 softmax 输出的概率分布中采样
- bpe — 分词是确定性的，不涉及 PRNG

详见 Part 2 — 采样策略。

出链

Part 2: 分词与采样

> 程序访问的虚拟地址尚未映射到物理内存时 CPU 触发的中断，内核借此按需加载页面，是虚拟内存的核心机制。

为什么需要了解 **Page Fault**

ds4.c 用 mmap 加载 81GB 模型文件，但不会一次性读入全部数据。程序访问某个地址时，如果对应的页面还在磁盘上，CPU 触发 page fault，内核把这一页从磁盘读入物理内存，然后恢复程序执行。

这意味着 page fault 的时机直接影响性能：

- 推理中触发 → 延迟抖动（毫秒级停顿）
- 预热阶段触发 → 集中承受延迟，推理时无卡顿

核心原理

```
程序访问 ptr[0]
    │
    ↓  CPU 查页表: 这一页在物理内存中吗？
    │
    ├─ 是 → 直接访问（纳秒级）
    │
    └─ 否 → 触发 Page Fault
              │
              ↓  内核从磁盘读入该页到物理内存
              ↓  更新页表映射
              ↓  恢复程序执行（重新访问 ptr[0]，这次命中）
              │
              ── 耗时约 1-10ms（取决于磁盘速度）
```

在 **ds4.c** 中的两种策略

1. 预热（主动触发 **page fault**）

```c
// model_warm_weights(): 逐页强制触发 page fault
for (size_t i = 0; i < size; i += 4096) {
    volatile char c = ((char *)map)[i];  // 读每页首字节
}
```

启动时承受所有延迟，推理时不再卡顿。

2. 避免 **calloc**（延迟 **page fault** 的陷阱）

```c
// xmalloc_zeroed(): 用 malloc + memset，不用 calloc
void *p = xmalloc(size);
memset(p, 0, size);  // 立即触发所有 page fault
```

为什么不用 `calloc`？Darwin 内核对大块 `calloc` 使用共享零页优化——首次写操作才触发 page fault。如果延迟到推理阶段，密集的 page fault 可能在 macOS 上触发内核 panic。

相关概念

- mmap — mmap 的按需加载依赖 page fault 机制
- xmalloc — xmalloc_zeroed 避免 calloc 的延迟 page fault 问题
- gguf — GGUF 张量数据通过 mmap 按需加载

详见 Part 1 — 内存管理与 mmap。

出链

Part 1: 构建与加载

> 模型量化——将 32-bit 浮点权重压缩到 2~8 bit，用精度换内存和速度，是本地推理的核心技术。

为什么需要量化

DeepSeek V4 Flash 有 284B 参数。如果用 F32 存储：

```
284B × 4 bytes = 1.136 TB — 消费级硬件完全无法容纳
```

量化将每个权重从 32 bit 压缩到 2~8 bit：

```
Q2_K:  284B × ~0.33 bytes ≈ 94GB → 实际文件约 81GB（含元数据）
Q4_K:  284B × ~0.56 bytes ≈ 159GB
Q8_0:  284B × 1 byte ≈ 284GB
```

量化是本地推理的前提——没有量化，消费级硬件无法运行这个模型。

核心原理

```
块量化（Block Quantization）: 每 256 个权重为一组

Q2_K 块（84 字节 / 256 权重 = 2.625 bit/权重）:
  scales[16]  — 16 组缩放因子（每组 16 个权重共享）
  qs[64]      — 256 个 2-bit 值（64 × 4 = 256 bits）
  d           — 全局缩放（float16）
  dmin        — 全局最小值偏移（float16）

反量化: weight_float = (qs_value - dmin) × scales[group] × d
```

非对称量化：缩放因子和偏移允许权重分布不对称（不像对称量化强制零中心），对 2-bit 这种极端量化特别重要。

在 ds4.c 中的实现

量化类型	用途	每权重 bit
Q2_K / IQ2_XXS	MoE 路由专家	~2-2.6
Q4_K	注意力投影、MoE 门控	~4.5
Q8_0 / Q8_K	共享专家、高精度路径	8
F16	嵌入层、输出层	16

> `dequantize_row_*()` 函数族将量化块还原为 float32 进行计算。GPU kernel 内联反量化。

Q4_K 路由专家（PRO 模型）

DeepSeek V4 PRO 模型的路由专家全部使用 Q4_K 量化。ds4 新增了 CPU 端的 Q4_K 路由专家推理支持：

- `block_q4_K`：144 字节/block，256 元素/block，含 super-block scales 和 min values
- `ds4_vec_dot_q4_K_q8_K()`：Q4_K × Q8_K 点积，ARM NEON dot-product 指令加速
- `matvec_q4_k_experts_mid_prequant()`：使用预量化 Q8_K 激活执行 gate+up 矩阵-向量乘
- `matvec_q4_k_experts_accum_prequant()`：down 投影累积

相关概念

- gguf — 量化权重以块为单位存储在 GGUF 文件中
- moe — 不同专家使用不同量化级别（共享专家用高精度，路由专家用低精度）
- mmap — 量化后的模型文件通过 mmap 按需加载

> 详见 Part 3 — 量化与矩阵运算。

出链

Part 3: 模型架构

> Root Mean Square Normalization——用均方根归一化替代 LayerNorm，去掉均值计算，更快更简单。

为什么需要 RMSNorm

Transformer 中每层前向需要两次归一化（Attention 前、MoE/FFN 前）。LayerNorm 计算均值和方差，两轮遍历；RMSNorm 只计算均方根，一轮遍历，且不需要均值偏移。

核心原理

```
LayerNorm: y = (x - mean) / sqrt(var + ε) × γ + β
           需要 mean 和 var 两个统计量，有可学习偏移 β

RMSNorm:   y = x / rms(x) × γ
           rms(x) = sqrt(mean(x²) + ε)
           只有可学习缩放 γ，无偏移项
```

RMSNorm 去掉了均值减法和偏移项，实验表明对 Transformer 效果几乎无影响。现代 LLM（LLaMA、DeepSeek、Mistral）都使用 RMSNorm。

在 ds4.c 中的实现

```
// RMSNorm（ds4.c 约 4577 行）
float rms = 0.0f;
for (int i = 0; i < n; i++) rms += x[i] * x[i];
rms = sqrtf(rms / n + DS4_RMS_EPS);  // ε 默认 1e-6，从 GGUF 元数据加载
for (int i = 0; i < n; i++) out[i] = x[i] / rms * weight[i];
```

每层前向调用两次 RMSNorm：一次在 Attention 前，一次在 MoE 前。

相关概念

- softmax — Attention 中 softmax 前的归一化（RMSNorm 是 QKV 投影前的归一化）
- moe — MoE Router 前也使用 RMSNorm
- swiglu — 专家激活前的归一化

详见 Part 3 和 Part 4。

出链

Part 3: 模型架构

Part 4: 推理流程

> Rotary Position Embedding，旋转位置编码 —— 通过旋转向量编码位置信息，使注意力自然感知 token 的相对距离。

为什么需要位置编码

注意力分数只看向量内容，不知道 token 顺序。"我爱猫"和"猫爱我"的 Q/K 值一样，但含义完全不同。如果不注入位置信息，模型无法区分词序。

核心原理

RoPE 将 Q/K 向量按维度拆成若干对，每对做二维旋转：

```
位置 0: 向量旋转 0°
位置 1: 向量旋转 θ°
位置 2: 向量旋转 2θ°
位置 n: 向量旋转 nθ°
```

```
高频分量（前面维度对，θ 大）: 精确区分相邻位置 —— "放大镜"
低频分量（后面维度对，θ 小）: 感知远距离关系 —— "广角镜"
```

关键性质：点积只依赖相对距离。不管绝对位置在哪，距离相同的 token 对有相同的旋转差：

```
Q_m · K_n = f(m - n)
```

这意味着注意力天然编码相对位置，不需要额外的位置参数。

在 ds4.c 中的实现

```
// RoPE 核心旋转（ds4.c 约 4574 行）
for (uint32_t i = 0; i < n_rot; i += 2) {
    const float c = cosf(theta) * mscale;
    const float s = sin_sign * sinf(theta) * mscale;
    const float x0 = tail[i + 0];
    const float x1 = tail[i + 1];
    tail[i + 0] = x0 * c - x1 * s;    // 二维旋转公式
    tail[i + 1] = x0 * s + x1 * c;
    theta_extrap *= theta_scale;        // θ 递减，产生不同频率
}
```

YaRN 长上下文扩展：当位置超过训练长度（65536）时，低频分量切换到内插（缩放角度），高频分量保持外推。`DS4_ROPE_SCALE_FACTOR = 16.0` 将上下文从 64K 扩展到 100 万 token。

相关概念

- kv-cache — K 在进入缓存前需先经 RoPE 旋转
- softmax — 旋转后的 Q·K 点积经 softmax 归一化为注意力权重
- moe — 注意力（含 RoPE）和 MoE 在每层中串联

详见 Part 4 — RoPE 与推理循环 和 LLM 核心架构概念 的 RoPE 章节。

出链

Part 4: 推理流程

> Server-Sent Events——服务器向客户端单向推送文本事件的 HTTP 流式协议，LLM 逐 token 输出的标准传输方式。

为什么需要 SSE

LLM 生成文本是逐 token 的，如果等全部生成完毕再返回，用户需要等待几十秒。SSE 让服务器每生成一个 token 就立即推送给客户端，实现"打字机"效果。

核心原理

```
HTTP 响应头:
  Content-Type: text/event-stream
  Cache-Control: no-cache

SSE 数据格式（每个事件以双换行分隔）:
  data: {"id":"chatcmpl-xxx","choices":[{"delta":{"content":"你"}}]}\n\n
  data: {"id":"chatcmpl-xxx","choices":[{"delta":{"content":"好"}}]}\n\n
  data: [DONE]\n\n
```

SSE vs WebSocket：

- SSE 是单向的（服务器 → 客户端），协议简单
- WebSocket 是双向的，LLM 场景不需要双向通信
- SSE 基于 HTTP，任何 HTTP 客户端都支持

在 ds4.c 中的实现

`sse_chunk()`（ds4_server.c 约 3119 行）每生成一个 token 立即推送 SSE 事件。内部维护 UTF-8 解码状态，处理中文字符跨越多次 `recv` 的情况，保证每个 chunk 都是合法 UTF-8。

SSE Keepalive

长 prompt 的 prefill 可能持续数十秒，期间 SSE 流没有新数据。ds4_server 在 prefill 期间每 5 秒发送 SSE 注释行 `: prefill\n\n` 作为心跳。SSE 规范中 `:` 开头的行是注释，客户端会忽略，但足以保持 TCP 连接活跃，防止代理/负载均衡器超时断开。

相关概念

- kv-cache — SSE 流式输出和 KV Cache 生成同步
- moe — MoE 路由每 token 执行一次

详见 Part 5 — HTTP 服务器。

出链

Part 5: 服务层

> 将原始分数（logits）转换为概率分布，是注意力和采样的核心归一化操作。

为什么需要 Softmax

注意力分数是 Q 和 K 的点积，值域没有约束（可以是任意实数）。我们需要把这些分数变成"权重"——非负且总和为 1，才能对 V 做加权平均。同理，模型输出 129280 个 logits，也需要归一化为概率才能采样。

核心原理

```
输入: 任意实数 [x₁, x₂, ..., xₙ]
输出: 概率分布 [p₁, p₂, ..., pₙ]，每个 pᵢ ∈ (0,1)，Σpᵢ = 1

公式: pᵢ = exp(xᵢ - max) / Σexp(xⱼ - max)
                    ↑ 减最大值是数值稳定技巧，防止 exp 溢出
```

不减最大值的问题：如果最大值是 1000，`exp(1000)` = ∞（float 溢出），整个计算产生 NaN。减去最大值后最大的 exp 项 = exp(0) = 1，其余都 < 1，安全。

在 **ds4.c** 中的实现

```
// 注意力中的内联 softmax（ds4.c 约 4751 行）
// 1. 找最大值
for (uint32_t r = 0; r < n_rows; r++)
    if (score[r] > max) max = score[r];
// 2. 减最大值 → exp → 累加
for (uint32_t r = 0; r < n_rows; r++) {
    score[r] = expf(score[r] - max);
    sum += score[r];
}
// 3. 归一化
for (uint32_t r = 0; r < n_rows; r++)
    score[r] /= sum;
```

采样阶段使用 `double` 精度累加，减少 129280 个浮点数的累加误差。

ds4.c 还引入了 **Attention Sink** 机制：每个注意力头有一个可学习的 sink logit，参与 softmax 分母但不贡献 value，让模型学会"不关注任何位置"的能力。详见 Part 3 — 注意力机制。

相关概念

- kv-cache — softmax 在缓存的 KV 行上计算注意力权重
- rope — RoPE 旋转后的 Q·K 点积作为 softmax 输入
- moe — Router 使用 sqrt(softplus(x)) 而非标准 softmax 归一化专家分数

详见 Part 2 — 采样策略。

出链

Part 3: 模型架构

Part 2: 分词与采样

> Swish-Gated Linear Unit——MoE 专家和 FFN 使用的激活函数，用门控机制混合两个线性变换。

为什么需要 SwiGLU

传统 FFN 使用 ReLU 激活：`output = ReLU(x × W1) × W2`。ReLU 的问题是：所有负值直接归零，信息完全丢失。SwiGLU 用门控机制替代 ReLU，允许负值区域也有梯度流动。

核心原理

```
ReLU:     output = max(0, x × W_gate) × W_up
SwiGLU:   output = (SiLU(x × W_gate) ⊙ (x × W_up)) × W_down
                         ↑ 门控信号         ↑ 信息流          ↑ 投影回原维度

SiLU(x) = x × σ(x)  (Sigmoid Linear Unit, 也叫 Swish)
⊙ = element-wise 乘法（门控）
```

SwiGLU 需要三个权重矩阵（gate、up、down），比 ReLU-FFN 的两个多一个，但效果更好。现代 LLM（LLaMA、DeepSeek、Mistral）几乎全部使用 SwiGLU。

在 ds4.c 中的实现

每个 MoE 专家有三个权重矩阵（`ffn_gate`、`ffn_up`、`ffn_down`），共享专家同理：

```
// 专家前向（伪代码）
gate = matvec(x, ffn_gate);   // 门控投影
up   = matvec(x, ffn_up);     // 上投影
hidden = silu(gate) * up;     // 门控混合
output = matvec(hidden, ffn_down);  // 下投影回原维度
```

相关概念

- moe — 每个 MoE 专家内部使用 SwiGLU 激活
- quantization — 专家的 SwiGLU 权重使用激进量化（~2 bit）

> 详见 Part 3 — MoE 混合专家。

出链

Part 3: 模型架构

> Yet another RoPE extensioN——通过调整 RoPE 的频率缩放，让训练时短上下文的模型能够外推到更长的上下文。

为什么需要 YaRN

RoPE 位置编码在训练时有固定的上下文长度（如 128K）。推理时如果输入超过训练长度，模型对超出部分的位置编码没有见过，性能骤降。

YaRN 不需要重新训练，通过数学技巧调整 RoPE 的频率，让模型"认为"远处的 token 仍在熟悉的位置范围内。

核心原理

```
标准 RoPE:    θ_i = 1 / (10000^(2i/d))
YaRN 缩放:   θ_i = 1 / (10000^(2i/d)) × scale_factor

scale_factor = 目标长度 / 原始长度
例如: 训练 128K → 推理 1M，scale_factor = 1M/128K = 8

低频分量（i 大）: 直接缩放频率
高频分量（i 小）: 保持原始频率（高频信息更重要）
中频分量:        线性插值
```

三段式策略：低频缩放、中频插值、高频保持，在长度外推和信息保留之间取得平衡。

在 ds4.c 中的实现

`rope_tail_ext_inplace()`（ds4.c 约 4694 行）接受 `rope_freq_base` 和 `rope_ext_factor` 参数，YaRN 通过调整这两个参数实现上下文扩展。

相关概念

- rope — YaRN 是 RoPE 的扩展方法
- kv-cache — 更长的上下文意味着更大的 KV Cache

> 详见 Part 4 — RoPE 与推理循环。

出链

Part 4: 推理流程

> 将文件内容直接映射到进程的虚拟地址空间，像操作内存数组一样访问文件，无需 read/write 系统调用。

为什么用 **mmap** 加载模型

DeepSeek V4 Flash 的模型文件约 81GB。如果用 `fread` 加载，需要：

1. 分配 81GB 用户空间内存
2. 从内核空间拷贝到用户空间（81GB 的内存拷贝）
3. 等待全部拷贝完成才能开始推理

mmap 避免了这一切：文件直接映射到虚拟地址，内核按需加载页面，无需拷贝。Metal GPU 还能直接引用 mmap 内存，实现真正的零拷贝。

核心原理

```
传统 read:                    mmap:
磁盘 → 内核缓冲区              磁盘 ←→ 物理内存
      → 用户空间缓冲区                  ↕ (页表映射)
      → 使用                           用户空间指针直接访问
```

mmap 后，访问 `ptr[0]` 就像访问数组。如果对应页面还没加载到物理内存，CPU 触发 page fault，内核自动从磁盘读入，对程序透明。

在 **ds4.c** 中的实现

```c
// ds4.c — model_open 中的 mmap 调用
void *map = mmap(NULL, st.st_size, PROT_READ, flags, fd, 0);
```

两种 flags 策略：

平台	flags	原因
Metal（macOS）	`MAP_SHARED`	GPU 可直接零拷贝引用同一块物理内存
CPU only	`MAP_PRIVATE`	避免 macOS VM 内核 bug（写时复制）

张量数据通过偏移访问，始终在 mmap 区域内：

```
void *tensor_data(model, tensor) = model->map + tensor->abs_offset;
```

预热优化：`model_warm_weights()` 逐页（每 4096 字节）读取触发 page fault，将延迟集中在启动阶段，避免推理时的延迟抖动。

相关概念

- page-fault — mmap 的按需加载依赖 page fault 机制
- gguf — GGUF 文件通过 mmap 加载，游标直接在映射区解析
- xmalloc — 预热后的数据结构用 xmalloc 分配，与 mmap 区域配合

详见 Part 1 — 内存管理与 mmap。

出链

Part 1: 构建与加载

> 带 `x` 前缀的内存分配包装器 —— 分配失败时直接退出程序，省去每个调用点重复的 NULL 检查。

为什么需要 xmalloc

C 语言的 `malloc` 失败时返回 NULL，理论上每次调用都要检查：

```
// 没有 xwrapper 时：
void *p = malloc(size);
if (p == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(1);
}
// ... 每次调用都要重复这三行
```

ds4.c 有几百处内存分配，重复的 NULL 检查让代码膨胀且容易遗漏。xmalloc 把错误处理集中到一处，调用点只写 `p = xmalloc(size)`。

核心原理

```
xmalloc(size)  → malloc(size)  + NULL → fprintf + exit(1)
xcalloc(n, s)  → calloc(n, s) + NULL → fprintf + exit(1)
xrealloc(p, s) → realloc(p,s) + NULL → fprintf + exit(1)
```

为什么没有 `xfree`？释放 NULL 是安全的（`free(NULL)` 是 no-op），不需要包装。

在 ds4.c 中的实现

```c
// ds4.c — xwrapper 家族
void *xmalloc(size_t size) {
    void *p = malloc(size);
    if (!p) { fprintf(stderr, "xmalloc: out of memory\n"); exit(1); }
    return p;
}
```

特别注意 **`xmalloc_zeroed`**：

```c
void *xmalloc_zeroed(size_t size) {
    void *p = xmalloc(size);
    memset(p, 0, size);   // 手动清零
    return p;
}
```

为什么不用 `xcalloc`？Darwin 内核对大块 `calloc` 使用共享零页（CoW zero page）——物理内存只在首次写入时分配。这把大量 page fault 延迟到推理阶段，在 macOS 上可能导致内核 panic。`malloc + memset` 立即触发所有 page fault，在启动阶段集中处理。

相关概念

- page-fault — xmalloc_zeroed 避免 calloc 的延迟 page fault 陷阱
- mmap — 模型数据通过 mmap 零拷贝加载，不经过 xmalloc

详见 Part 1 — 内存管理与 mmap。

出链

Part 1: 构建与加载