通用链接器技术

杨岢瑞

May 30, 2026

链接器位于编译管线的末端，它把多个可重定位目标文件、静态库与系统库组合成最终的可执行文件或共享对象。通用链接器在这一过程中同时支持多种目标文件格式、多种指令集以及多种运行时环境，从而实现了跨语言、跨平台和增量并行等工程价值。本文将沿着「背景—原理—实现—实践—趋势」的路线，系统梳理通用链接器如何把代码碎片拼装成可执行二进制。

1　背景：为什么需要链接器?

在单文件编译时代，编译器把整段源代码翻译成机器码即可直接运行。随着程序规模扩大，开发者把功能拆分到不同模块中，每个模块单独编译成目标文件（.o 或 .obj）。此时，函数调用、全局变量访问都以符号的形式存在于各自目标文件中，链接器必须把这些符号解析并重定位到最终地址，才能生成完整可执行文件。早期 ld 与 link.exe 仅支持单一格式与单一体系结构，面对多语言混合、异构指令集与 WebAssembly 等新兴场景已力不从心，通用链接器应运而生。

2　链接器 101：核心概念与数据结构

可重定位目标文件由多个段（Section）组成，每个段存放代码、数据或元信息。以 ELF 为例，.text 段保存指令，.data 与 .bss 分别存放已初始化与未初始化全局变量，.symtab 则保存符号表。符号表中的每一项记录符号名称、所在段、偏移与绑定属性。重定位表（Relocation Section）描述需要修正的指令或数据地址，例如一条 call 指令的操作数在链接前可能是 0，链接器必须用最终符号地址减去指令地址再加上常数偏移，写回该操作数字段。

符号分为全局、局部与弱三种。全局符号在整个程序可见；局部符号仅在本目标文件可见；弱符号在链接时优先级低于强符号，若同时存在同名强弱符号，链接器选择强符号并忽略弱符号。对齐要求决定段起始地址必须是特定字节数的整数倍，否则处理器取指或访存可能触发异常。地址分配阶段，链接器按脚本或默认规则把段放置到虚拟地址空间，不同段之间可能留下「空洞」，以满足对齐与分页需求。

3　链接过程全流程拆解

符号解析是链接的第一步。C++ 编译器会对函数名进行名称粉碎（Name Mangling），把参数类型编码到符号名中；链接器必须使用相同的 ABI 规则才能找到正确符号。段合并阶段，链接器把所有输入目标文件的同名段拼接成单一输出段，并计算各段在虚拟地址空间的最终位置。重定位阶段，链接器遍历重定位表，根据符号地址与段基址计算修正值，常见类型包括绝对地址重定位、PC 相对重定位以及通过 GOT/PLT 实现的动态重定位。

最终，链接器把合并后的段、程序头、节头与符号表写出，生成 ELF 可执行文件、共享对象或可重定位二进制。整个流程可抽象为：输入目标文件集合 → 构建全局符号表 → 段分配与地址布局 → 重定位计算 → 输出二进制。任何一步出错，都会表现为「undefined reference」或「multiple definition」。

4 通用链接器的核心特性

通用链接器至少具备三层抽象：目标格式抽象、体系结构抽象与后端抽象。目标格式抽象层把 ELF、PE/COFF、Mach-O 与 WASM 对象文件统一映射为内部 Section/Symbol/Relocation 结构；体系结构抽象层提供 x86_64、ARM64、RISC-V 与 WASM32 的指令编码与重定位类型；后端抽象层则允许用户在同一驱动程序中切换 BFD、LLD 或 mold 等不同实现。

并行与增量是现代链接器的工程亮点。符号解析可按目标文件并行扫描；段分配可把不同输出段分配给不同线程；ThinLTO 把中间表示切分为独立单元，仅对跨模块调用图变化的部分重新优化，从而把全量链接时间从分钟级降至秒级。跨语言互操作依赖统一的调用约定与符号可见性规则：Rust 与 C 的 extern C 块、Swift 的 @convention(c) 以及 Go 的 cgo 均在这一层得到支持。

5 典型实现解析

GNU ld 使用 BFD 库把不同格式翻译成统一内部表示，插件机制灵活，但每次符号查找都要经过多次间接调用，速度较慢。LLVM LLD 采用「输入文件即对象」的零拷贝设计，把 ELF、COFF、WASM 解析成轻量 IR，并在多线程中并行处理重定位，链接 Chrome 级别代码库可在 2 秒内完成。mold 用 C++20 编写，核心循环全部基于内存映射文件与 SIMD 指令，单线程链接速度比 LLD 快 5 – 7 倍，内存峰值也更低。wasm-ld 则是 LLD 的特化版本，仅保留 WASM 需要的段类型与重定位，代码量不到 10 k 行，却能生成符合 WebAssembly 规范的模块。

性能基准显示：在同一台 32 核工作站上，链接 1.2 GB 目标文件时，GNU ld 需要 48 秒，LLD 需要 3.2 秒，mold 仅需 0.9 秒；内存峰值分别为 27 GB、9 GB 与 6 GB。输出体积方面，启用 ICF（Identical Code Folding）后，三者均可减少约 15 % 的 .text 段大小。

6 工程实践与性能调优

链接脚本（Linker Script）是控制最终内存布局的唯一手段。一个最小嵌入式脚本示例：

```
ENTRY(_start)
SECTIONS {
  . = 0x08000000;
  .text : { *(.text .text.*) }
  .rodata : { *(.rodata .rodata.*) }
  .data : { *(.data .data.*) }
  .bss : { *(.bss .bss.*) }
}
```

脚本首先把入口符号设为 _start，把代码段放在 Flash 起始地址 0x08000000，随后依次放置只读数据、已

初始化数据与未初始化数据。编译器标志 -ffunction-sections 与 -fdata-sections 把每个函数与全局变量单独放入独立段，配合 –gc-sections 可把未使用代码从最终镜像中删除。-flto 让链接器看到整个程序的中间表示，从而执行跨模块内联与常量折叠；ThinLTO 与 Propeller 在此基础上进一步把优化与代码布局解耦，实现分布式增量编译。–icf=safe 只合并完全相同的常量与函数，避免误伤多态调用。

Debug 符号剥离可通过 objcopy -S 或 -strip-all 完成；Split DWARF 把调试信息写入独立 .dwo 文件，链接时仅保留符号索引，大幅降低内存占用。可重现构建要求链接器把时间戳、构建路径等非确定信息清零，mold 与 LLD 均提供 –build-id=sha1 与 –no-insert-timestamp 选项。

7 安全与可靠性

现代链接器直接参与安全机制实现。RELRO 把 GOT 设为只读，防止运行时重写；PIE 生成位置无关可执行文件，配合 ASLR 增加攻击难度；CET 与 PAC 在 x86_64 与 ARM64 上分别提供影子栈与指针认证，链接器需要在 .note.gnu.property 或 PT_GNU_PROPERTY 段中写入对应属性。控制流完整性（CFI）需要链接器收集所有有效调用目标并生成跳转表，运行时检查跳转地址是否在白名单内。可重现构建与包格式签名结合，可在供应链层面检测二进制是否被篡改。

8 新兴领域与未来趋势

WebAssembly 把链接器从「操作系统工具」变成「语言运行时组件」。wasm-ld 不仅解析 .o 格式，还要在链接期生成「接口适配器」，把 Component Model 中的 Interface Types 翻译成 WASM 指令序列。异构计算场景下，GPU 端代码以 ELF 或自定义格式存在，链接器需同时处理主机与设备端符号，并在最终可执行文件中嵌入设备镜像。云原生 FaaS 平台要求按需链接：当函数首次调用时，链接器在毫秒级完成符号解析与重定位，生成临时共享对象并立即执行。下一代二进制格式如 eBPF ELF、WASM-GC 与 CHERI 能力安全指针，都在重新定义链接器需要解析的重定位类型与安全属性。

9 动手实验

以最小 C 程序为例：

```
int main() { return 42; }
```

使用 gcc -c 生成 main.o 后，执行 readelf -s main.o 可看到符号表中存在 main 与 _start 等符号；readelf -r main.o 则列出对 __libc_start_main 的重定位条目。把系统 ld 替换为 mold，只需把 /usr/bin/ld 软链接到 mold，或在 CMake 中设置 -DCMAKE_LINKER=mold。Rust 项目启用 LTO 与 mold 的命令为：

```
RUSTFLAGS="-C link-arg=-fuse-ld=mold -C target-cpu=native" cargo build --release
```

交叉编译到 aarch64-unknown-linux-musl 时，结合 cargo-zigbuild 可在不安装 aarch64 工具链的情况下生成静态二进制。

10 常见问题与 FAQ

「undefined reference」通常意味着符号在任何输入目标文件或库中都未定义，或库的链接顺序晚于引用它的目标文件。静态库与共享库的链接顺序遵循「最早解析」原则：链接器从左到右扫描输入文件，一旦符号被解析，后续库中的同名符号不再生效。段合并导致的地址漂移可通过 readelf -l 查看程序头，若某段的虚拟地址与预期不符，需检查链接脚本中的对齐指令。WASM 链接器与原生链接器的最大差异在于地址空间：WASM 使用 32 位或 64 位线性内存，而非虚拟地址空间，因此重定位计算仅涉及内存偏移，不涉及页表与特权级。

链接器早已从幕后工具演变为影响语言生态、部署效率与安全策略的关键基础设施。理解其内部结构与可插拔设计，能帮助工程师在性能、尺寸与安全之间做出最优权衡。延伸阅读推荐《Linkers and Loaders》、LLVM LLD 源码、mold 仓库以及 WebAssembly Component Model 规范。