

XIRASM 格式教程

这份教程讲如何用 XIRASM 直接生成 PE、ELF、COFF 和 macOS Mach-O 文件。写法从 `format/format.inc` 开始：声明文件类型、节或装载段、入口、导入导出和重定位，文件头、表项、文件偏移、RVA、对齐以及最终回填由 `format.inc` 生成。

如果过去主要在 C、C++、Rust 或其他语言里写内联汇编，文件格式通常由编译器、链接器和运行时处理，不需要自己回答下面这些问题。XIRASM 直接生成输出文件，因此每一项都要明确给出：

- 程序从哪个标签开始执行；
- 哪些字节是代码，哪些字节是数据；
- 哪些范围在运行时可读、可写或可执行；
- 哪些范围只占内存大小，不占初始化文件内容；
- 需要导入或导出哪些外部符号；
- 文件里哪些绝对地址需要让加载器或链接器修正；
- 最终阶段需要回填哪些地址、校验和或表项字段。

整个教程只需要一个导入：

```
import("format/format.inc")
```

`include/format/` 下的文件分三类，日常使用不必全部了解：

- `format.inc` 是常用入口，提供格式配置、节与装载段声明、导入、导出、资源、重定位、符号表和最终生成流程；
- `pe32.inc`、`pe64.inc`、`elf32.inc`、`elf64.inc` 是按位宽拆出的专用入口，维护已有源码或编写专门格式时才直接导入；
- `pe.inc`、`elfexe.inc`、`elfobj.inc` 等文件提供更细的文件头、目录、节表、程序头和动态表构造函数，`format.inc` 无法表达目标布局时才使用。

需要手写 PE/ELF 头部、目录、节表、程序头或动态表时，阅读[高级格式构造指南](#)。

章节

1. [先选输出类型](#)
2. [Windows PE 和 DLL](#)
3. [Linux ELF 可执行文件和共享库](#)
4. [COFF 和 ELF 目标文件](#)

- 5. 通用规则和常见错误
- 6. macOS Mach-O

快速选择

目标	起始函数	主要操作
Windows 可执行文件	<code>format_pe32</code> 或 <code>format_pe64</code>	<code>format_section_begin</code> 、 <code>format_pe_import_section</code> 、 <code>format_pe_resource_section</code> 、 <code>format_pe_reloc_section</code>
Windows DLL	<code>format_pe32</code> 或 <code>format_pe64</code> ，选项含 <code>format_pe_dll</code>	<code>format_pe_export_section</code> ，可选导入、资源 and 重定位
Linux 可执行文件	<code>format_elf32</code> 或 <code>format_elf64</code> ，选项为 <code>format_elf_exec</code>	<code>format_segment_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Linux PIE	<code>format_elf64</code> ，选项为 <code>format_elf_pie</code>	<code>format_segment_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Linux 共享库	<code>format_elf64_so</code>	<code>format_elfso_tables_mut</code> 、 <code>format_segment_begin</code> 、 <code>format_finish</code>
Android 共享库 (AArch64)	<code>format_elf64_so_aarch64</code>	<code>format_elfso_import_slots_mut</code> 、 <code>format_elfso_tables_mut</code> 、 <code>format_segment_begin</code> 、 <code>format_finish</code>
COFF 目标文件	<code>format_coff32</code> 或 <code>format_coff64</code>	<code>format_coff_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>
ELF 目标文件 (x86)	<code>format_elfobj32</code> 或 <code>format_elfobj64</code>	<code>format_elfobj_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>
ELF 目标文件 (AArch64)	<code>format_elfobj64_aarch64</code>	<code>format_elfobj_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>
macOS Mach-O 目标文件	<code>format_macho64_object</code>	<code>format_macho64_target_arm64</code> 或 <code>format_macho64_target_x86_64</code> 、 <code>format_macho64_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>
macOS Mach-O 可执行文件	<code>format_macho64_exe</code>	<code>format_macho64_segment</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>

目标	起始函数	主要操作
macOS Mach-O 动态库	<code>format_macho64_dylib</code>	<code>format_macho64_segment</code> 、 <code>format_macho64_exports_mut</code> 、 <code>format_finish</code>

基本生命周期

所有格式配置都沿同一条主线推进：

```
import("format/format.inc")

// 1. 建立配置：文件角色、子系统、安全选项、ASLR 策略。
let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
    list.of(
        // 2. 声明这个文件有哪些节。
        format_section(".text", format_code | format_readable | format_executable)
    )
)

// 3. 开始输出：头部和节表的空间在这里预留。
format_begin(image)

// 4. 只能写进已经声明过的节。
format_section_begin(image, ".text")
start:
    xor eax, eax
    ret
format_section_end(image, ".text")

// 5. 可执行文件设入口，然后完成全部回填。
format_entry_mut(image, start)
format_finish(image)
```

第 1 章给出了这个骨架的完整版本，以及 ELF 可执行文件对应的写法。

需要交给链接器的元数据（符号表、重定位表、导入导出表）在第 3 步之前挂到配置上，因为头部和动态表要为它们预留空间。判断某个函数属于哪一类，看它是否以 `format_*_tables_mut` 命名；其余生成节内容的函数放在第 3 步之后、第 5 步之前。

使用原则

- 先选输出文件类型，再写节、装载段和表项，不要从 PE/ELF 的原始字段开始推。
- PE、COFF 和 ELF 目标文件用 `format_section(...)` 声明节；ELF 可执行文件和共享库用 `format_segment(...)` 声明装载段。

- 头部、表项、对齐、RVA、文件偏移和最终回填都由 `format.inc` 生成，源文件只声明内容边界和必要元数据。
- 同一个文件不要一边用 `format.inc`，一边手写 PE/ELF 头部或表项；必须手写这些字段时，整份文件都按高级格式构造的方式组织。
- 从最小的 `.text` 模板开始，先让最小模板汇编通过，再逐步加入导入、导出、资源、重定位和 BSS。

1. 先选输出类型

写格式文件，第一步是确定要输出哪一种文件。这一步决定后面所有字段：PE 映像用节表描述内容，ELF 可执行文件用程序头描述内容，目标文件既没有入口地址，也没有装载段。文件类型选错，声明的节或装载段就无法写进输出文件。

`format.inc` 把常见的输出归纳成几组入口函数。这一层是包装层：使用方声明目标文件类型、节或装载段、入口位置，头部、节表或程序头、文件偏移、RVA、对齐以及全部回填字段都由它生成。

统一入口

```
// 声明节或装载段，其余格式结构交给 format.inc。
import("format/format.inc")
```

直接导入 `pe.inc`、`elfexe.inc`、`elfobj.inc` 这几个文件，文件头、目录和表项都要自己填。只有 `format.inc` 表达不了目标布局时，才改用[高级格式构造指南](#)。

配置函数

输出	配置函数	内容描述	选项
PE32 可 执行文 件或 DLL	<code>format_pe32(options, sections)</code>	<code>format_section(...)</code>	必填
PE64 可 执行文 件或 DLL	<code>format_pe64(options, sections)</code>	<code>format_section(...)</code>	必填

输出	配置函数	内容描述	选项
PE64 可 执行文 件或 DLL，机 器由参 数给出	<code>format_pe64_machine(options, sections, machine)</code>	<code>format_section(...)</code>	必填
PE64 AArch64 可执行 文件或 DLL	<code>format_pe64_arm64(options, sections)</code>	<code>format_section(...)</code>	必填
COFF32 目标文 件	<code>format_coff32(sections)</code>	<code>format_section(...)</code>	无
COFF64 目标文 件	<code>format_coff64(sections)</code>	<code>format_section(...)</code>	无
COFF64 目标文 件，机 器由参 数给出	<code>format_coff64_machine(sections, machine)</code>	<code>format_section(...)</code>	无
COFF64 AArch64 目标文 件	<code>format_coff64_arm64(sections)</code>	<code>format_section(...)</code>	无
ELF32 可执行 文件	<code>format_elf32(options, segments)</code>	<code>format_segment(...)</code>	必填，只能是 <code>format_elf_exec</code>
ELF64 可执行 文件	<code>format_elf64(options, segments)</code>	<code>format_segment(...)</code>	必填
ELF64 AArch64 可执行 文件	<code>format_elf64_aarch64(options, segments)</code>	<code>format_segment(...)</code>	必填

输出	配置函数	内容描述	选项
ELF32 目标文件	<code>format_elfobj32(sections)</code>	<code>format_section(...)</code>	无
ELF64 目标文件	<code>format_elfobj64(sections)</code>	<code>format_section(...)</code>	无
ELF64 AArch64 目标文件	<code>format_elfobj64_aarch64(sections)</code>	<code>format_section(...)</code>	无
ELF64 共享库	<code>format_elf64_so(soname, segments)</code>	<code>format_segment(...)</code>	无，库名由 <code>soname</code> 给出
ELF64 AArch64 共享库	<code>format_elf64_so_aarch64(soname, segments)</code>	<code>format_segment(...)</code>	无，库名由 <code>soname</code> 给出

ELF 可执行文件的选项是 `format_elf_exec` 或 `format_elf_pie`，二选一。共享库不接收选项，文件类型由函数本身确定。ELF32 层只支持可执行模式，传入 `format_elf_pie` 会报 `format.inc ELF32 layer currently supports executable mode`。

机器通常是固定的：`format_pe64` 就是 x86-64，`format_coff64` 就是 AMD64。机器名来自配置、需要在运行时选定目标时，用带 `machine` 参数的那两个构造函数，例如 `format_pe64_machine(options, sections, pe_machine_arm64)`。

节描述

PE、COFF 和 ELF 目标文件按节组织内容。一个节用 `format_section(name, attributes)` 声明，两个参数：

参数	写什么
<code>name</code>	节名，例如 <code>".text"</code> 、 <code>".data"</code> 、 <code>".bss"</code> 、 <code>".idata"</code>
<code>attributes</code>	一个用途标志，加上需要的权限标志

用途说明这个节装什么，每个节只能给一个：

用途	装什么
<code>format_code</code>	指令

用途	装什么
<code>format_data</code>	已初始化数据或只读常量
<code>format_uninitialized_data</code>	零初始化内存，也就是 BSS
<code>format_imports</code>	PE 导入表
<code>format_exports</code>	PE 导出表
<code>format_resources</code>	PE 资源
<code>format_fixups</code>	PE 基址重定位表

权限说明这个节在运行时的访问方式，可以叠好几个：

权限	含义
<code>format_readable</code>	运行时可读
<code>format_writeable</code>	运行时可写
<code>format_executable</code>	运行时可执行
<code>format_discardable</code>	加载后可丢弃的元数据

常用组合：

内容	推荐属性
代码	<code>format_code</code> <code>format_readable</code> <code>format_executable</code>
只读数据	<code>format_data</code> <code>format_readable</code>
可写数据	<code>format_data</code> <code>format_readable</code> <code>format_writeable</code>
BSS	<code>format_uninitialized_data</code> <code>format_readable</code> <code>format_writeable</code>
PE 导入表	<code>format_imports</code> <code>format_readable</code> <code>format_writeable</code>
PE 导出表	<code>format_exports</code> <code>format_readable</code>
PE 资源	<code>format_resources</code> <code>format_readable</code>
PE 重定位表	<code>format_fixups</code> <code>format_readable</code> <code>format_discardable</code>

节名必须唯一。PE 和 COFF 的节名最长八字节，超了报 `PE/COFF section names must fit in eight bytes`。`format_imports`、`format_exports`、`format_resources`、`format_fixups` 这四种用途，在一个配置里各只能出现一次，重复报 `format special-purpose section is duplicated`。

装载段描述

ELF 可执行文件和 ELF 共享库不按节组织，按装载段组织：程序头描述的就是装载段，加载器据此把文件映射进内存。装载段用 `format_segment(name, attributes)` 声明，与 `format_section(name, attributes)` 对应：

参数	写什么
<code>name</code>	装载段名，例如 <code>".text"</code> 、 <code>".rodata"</code> 、 <code>".data"</code> 、 <code>".bss"</code>
<code>attributes</code>	<code>format_load</code> 加权限标志

`format_load` 是这一层唯一支持的装载段用途。权限还是那三项：`format_readable`、`format_writeable`、`format_executable`。装载段没有可丢弃这个属性，写 `format_discardable` 会报 `unknown format segment attribute`。

内容	推荐属性
代码	<code>format_load</code> <code>format_readable</code> <code>format_executable</code>
只读数据	<code>format_load</code> <code>format_readable</code>
可写数据	<code>format_load</code> <code>format_readable</code> <code>format_writeable</code>
BSS	<code>format_load</code> <code>format_readable</code> <code>format_writeable</code>

装载段名同样必须唯一，重复报 `format segment name is duplicated`。

完整流程

下面是一个最小的 PE64 控制台程序。它声明 `.text` 和 `.bss`，PE 头、节表、入口字段，以及 BSS 的文件大小与内存大小之差，全部由 `format.inc` 生成：

```
import("format/format.inc")

// 四个选项：可执行文件、控制台子系统、开启 NX、关闭 ASLR。
let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
    list.of(
        // 代码节：可读可执行。
        format_section(".text", format_code | format_readable | format_executable),
        // BSS 节：可读可写，只占内存地址。
        format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
    )
)

// 开始输出，头部和节表的空间在这里预留。
format_begin(image)

format_section_begin(image, ".text")
start:
    xor eax, eax
    ret
```

```

format_section_end(image, ".text")

// BSS 只占内存地址，不写文件字节。
format_section_begin(image, ".bss")
scratch:
    rb(64)
format_section_end(image, ".bss")

// 设入口地址，然后回填全部字段。
format_entry_mut(image, start)
format_finish(image)

```

结果是 1024 字节的合法 PE：MZ 头，PE\0\0 签名，机器码 0x8664 (x86-64)，入口 RVA 0x1000，节表里两个节，.bss 的文件大小是 0、内存大小是 64。

顺序分五步，错一步输出就不对：

1. 先用 `let image: map = ...` 建立配置；
2. 再调用 `format_begin(image)` 预留头部和表项；
3. 内容必须写进已经声明过的节；
4. 可执行文件用 `format_entry_mut(image, start)` 设入口；
5. 最后调用 `format_finish(image)` 完成全部回填。

第 3 步到第 5 步之间不能漏掉 `format_section_end`。漏了不报错，汇编仍然成功，但节表那一行定不下来：文件从 1024 字节缩到 515 字节，.text 的文件大小停在 3，也就是它的逻辑长度，没有按 512 字节的文件对齐补齐。这种文件加载器会直接拒绝。

ELF 可执行文件

ELF 可执行文件把 `format_section` 换成 `format_segment`，把 `format_section_begin` / `format_section_end` 换成 `format_segment_begin` / `format_segment_end`，其余顺序一模一样：

```

import("format/format.inc")

// 选项 format_elf_exec 表示固定地址可执行文件；换成 format_elf_pie 就是 PIE。
let image: map = format_elf64(
    format_elf_exec,
    list.of(
        // 装载段：可读可执行，由加载器映射进内存。
        format_segment(".text", format_load | format_readable | format_executable)
    )
)

format_begin(image)

format_segment_begin(image, ".text")
start:
    // 系统调用 60: exit(0)。
    mov eax, 60
    xor edi, edi

```

```
syscall
format_segment_end(image, ".text")

format_entry_mut(image, start)
format_finish(image)
```

结果是 137 字节的 ELF 映像：`\x7fELF` 魔数，64 位小端，类型 `ET_EXEC` (2)，机器 `EM_X86_64` (62)，入口 `0x400080`，一个 `PT_LOAD` 程序头，权限 `R|X`，文件大小 9 字节、内存大小 9 字节，页对齐 `0x1000`。

两种类型不能混用。ELF 映像配 `format_section`，报 `an ELF image plan declares segments, not sections`；反过来，PE 配 `format_segment`，报 `only an ELF image plan declares segments`。目标文件和 ELF 共享库没有普通可执行文件入口，对它们调用 `format_entry_mut`，报 `format plan does not use an executable entry`。

2. Windows PE 和 DLL

Windows 上的可执行程序 and DLL 都用 PE 格式。用 `format.inc` 生成 PE 时，`options` 必须指定三项：文件角色是 exe 还是 DLL、用哪个子系统、ASLR 取哪种策略；`format_pe_nx` 另算一个可选标志。其余头部字段、节表、导入目录、导出目录和基址重定位表，都由 `format.inc` 从这几项推出来。

PE 选项

`format_pe32(options, sections)` 和 `format_pe64(options, sections)` 的 `options` 必须指定一个角色、一个子系统、一个 ASLR 策略，`format_pe_nx` 可选。少给一组就报错：缺角色报 `PE format requires EXE or DLL`，缺子系统报 `PE format requires console or GUI subsystem`，缺 ASLR 报 `PE format requires an ASLR policy`。同一分组给两个值也报错，例如同时给出 `console` 和 `gui`，报 `PE format has multiple subsystems`。

分组	值	含义
角色	<code>format_pe_exe</code>	可执行文件
角色	<code>format_pe_dll</code>	DLL
子系统	<code>format_pe_console</code>	控制台程序
子系统	<code>format_pe_gui</code>	GUI 程序
安全	<code>format_pe_nx</code>	标记为 NX 兼容
ASLR	<code>format_pe_aslr_auto</code>	存在重定位节时才启用 ASLR
ASLR	<code>format_pe_aslr_required</code>	要求启用 ASLR，且必须声明重定位节
ASLR	<code>format_pe_aslr_disabled</code>	不启用 ASLR

`format_pe_nx` 写入 `DllCharacteristics` 的 `NX_COMPAT` 位。ASLR 三档写入同一字段的 `DYNAMIC_BASE` 位：

策略	DllCharacteristics
<code>format_pe_aslr_auto</code> ，未声明重定位节	<code>DllCharacteristics = 0x100</code> ， <code>DYNAMIC_BASE</code> 未置位，重定位目录为 <code>(0, 0)</code>
<code>format_pe_aslr_required</code> ，已声明重定位节	<code>DllCharacteristics = 0x160</code> ， <code>DYNAMIC_BASE</code> 置位，重定位目录指向 <code>.reloc</code>

未声明重定位节却要求 ASLR，报 `PE ASLR required needs a fixups section`。

`format_pe64` 生成 x86-64 映像，`format_pe64_arm64(options, sections)` 生成 AArch64 映像。两种机器共用容器布局、导入导出目录和 `DIR64` 基址重定位，差异只在文件头机器字段与指令编码。

常见 PE 节：

节	推荐属性
<code>".text"</code>	<code>format_code</code> <code>format_readable</code> <code>format_executable</code>
<code>".data"</code>	<code>format_data</code> <code>format_readable</code> <code>format_writeable</code>
<code>".bss"</code>	<code>format_uninitialized_data</code> <code>format_readable</code> <code>format_writeable</code>
<code>".idata"</code>	<code>format_imports</code> <code>format_readable</code> <code>format_writeable</code>
<code>".edata"</code>	<code>format_exports</code> <code>format_readable</code>
<code>".rsrc"</code>	<code>format_resources</code> <code>format_readable</code>
<code>".reloc"</code>	<code>format_fixups</code> <code>format_readable</code> <code>format_discardable</code>

PE 节名不得超过八字节。导入、导出、资源和重定位这四类特殊节在同一配置中各只能出现一次。

最小 PE64 可执行文件

```
import("format/format.inc")

// 可执行文件、控制台子系统，启用 DEP，不启用 ASLR。
let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
    list.of(
        // 代码节：可读可执行。
        format_section(".text", format_code | format_readable | format_executable),
        // BSS：只在内存中占 64 字节，不写入文件字节。
        format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
```

```

    )
format_begin(image)

format_section_begin(image, ".text")
start:
    xor eax, eax
    ret
format_section_end(image, ".text")

format_section_begin(image, ".bss")
    rb(64)
format_section_end(image, ".bss")

// 入口地址不得为零: PE 配置未设置入口时, format_finish 报
// "PE executable or DLL requires an entry address"。
format_entry_mut(image, start)
format_finish(image)

```

`format_begin` 写入并预留 PE 头和节表; `format_finish` 在布局稳定后回填入口、节大小、RVA 和文件偏移。上例的结果是一个 1024 字节的 PE 映像: `Format: COFF-x86-64`、`Subsystem: IMAGE_SUBSYSTEM_WINDOWS_CUI`、`AddressOfEntryPoint: 0x1000`、`SizeOfHeaders: 512`、`SectionAlignment: 4096`、`FileAlignment: 512`。

导入 Windows API

导入表描述加载器在启动时要填的外部函数地址。顺序是: 先声明 `.idata` 节, 再建立导入集合, 最后由 `format_pe_import_section` 生成表内容。不要自行打开 `.idata` 手工填充导入表项——描述符、查找表、地址表、提示名和 DLL 名字字符串之间的 RVA 关系由 `format.inc` 推导。

```

import("format/format.inc")

let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_auto,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        // 导入节必须先声明, 再交给 format_pe_import_section 生成内容。
        format_section(".idata", format_imports | format_readable | format_writeable)
    )
)

// 槽名默认取 API 原名; 导入完成后可在指令中直接引用这些标号。
let imports: map = format_pe_import_new()
format_pe_import_many_mut(
    image,
    imports,
    "KERNEL32.DLL",
    list.of("ExitProcess", "GetCurrentProcessId")
)
// 同一个绑定可以继续加入其他 DLL; pairs 用于给 API 指定本地别名。
format_pe_import_pairs_mut(
    image,
    imports,

```

```
"ADVAPI32.DLL",
list.of("close_registry_key", "RegCloseKey")
)

format_begin(image)

format_section_begin(image, ".text")
start:
    // Windows x64 调用需要 32 字节 shadow space; 此处额外占用 8 字节以保持 16 字节栈对齐。
    sub rsp, 40
    // PE64 通过 RIP 相对的导入槽调用。
    call [rel GetCurrentProcessId]
    xor ecx, ecx
    call [rel ExitProcess]
format_section_end(image, ".text")

// 在此生成 .idata: 不要在该调用之外再次打开同一个节。
format_pe_import_section(image, ".idata", imports)

format_entry_mut(image, start)
format_finish(image)
```

生成的映像为 1536 字节，导入目录里有两张表：KERNEL32.DLL 带 `ExitProcess` 和 `GetCurrentProcessId`，ADVAPI32.DLL 带 `RegCloseKey`。第二张表里没有 `close_registry_key`——`pairs` 的写法是「本地槽名, 真实 API 名」，本地名只用于源码中引用。

导入相关函数：

函数	参数	用途
<code>format_pe_import_new()</code>	无	建立空导入集合
<code>format_pe_import_many_mut(plan, imports, dll, names)</code>	配置、集合、DLL 名、函数名列表	按原名批量加入，槽名与函数名相同
<code>format_pe_import_pairs_mut(plan, imports, dll, pairs)</code>	配置、集合、DLL 名、槽名/函数名对	批量加入，并为每个函数指定本地槽名
<code>format_pe_import_section(plan, name, imports)</code>	配置、已声明的导入节名、集合	生成导入节内容并注册导入目录

`pairs` 列表的项数必须为偶数，超过 128 项报 `PE64 import pairs exceed the batch limit`；项数为奇数报 `PE64 import pairs require slot/name entries`。同一个槽名不能指向两个不同的函数，冲突时报 `PE import slot already maps to a different name`。

PE64 通过导入槽调用时写成 `call [rel slot_name]`，PE32 写成 `call [slot_name]`——64 位下导入槽地址是 RIP 相对的，32 位下是绝对地址。

导出 DLL 函数

DLL 导出把内部标签暴露为外部符号。`many` 按标签原名导出，`pairs` 指定公开名称。

```
import("format/format.inc")

let image: map = format_pe64(
  // 角色改为 format_pe_dll, 文件头会置 IMAGE_FILE_DLL 位。
  format_pe_dll | format_pe_console | format_pe_nx | format_pe_aslr_auto,
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".edata", format_exports | format_readable)
  )
)

let exports: list = format_pe_export_new()
// 两个标签按原名导出。
format_pe_export_many_mut(image, exports, list.of("x_add7", "x_sub3"))
// answer_impl 以 x_answer 之名导出。
format_pe_export_pairs_mut(image, exports, list.of("answer_impl", "x_answer"))

format_begin(image)

format_section_begin(image, ".text")
dll_main:
  // DLL 入口返回非零值表示加载成功。
  mov eax, 1
  ret
x_add7:
  lea eax, [ecx + 7]
  ret
x_sub3:
  lea eax, [ecx - 3]
  ret
answer_impl:
  mov eax, 42
  ret
format_section_end(image, ".text")

// 最后一个参数是 DLL 自身的名字，写入导出目录。
format_pe_export_section(image, ".edata", exports, "xirasm_demo.dll")

format_entry_mut(image, dll_main)
format_finish(image)
```

生成的映像为 1536 字节，导出表里有三个导出，序数从 1 开始，名字按字典序排列（`x_add7`、`x_answer`、`x_sub3`）——PE 的名字指针表要求有序，`format.inc` 会自动排序。导出表为空报 `PE export table requires at least one export`，DLL 名字为空报 `PE export DLL name is empty`。`pairs` 列表的项数同样必须为偶数，上限 128。

资源

资源节存放编译好的 `.res` 资源树——它是 `rc.exe` 或 `windres` 的产物，不是手写的文本文件。先声明 `.rsrc` 为 `format_resources`，再把文件路径交给生成函数：

```

import("format/format.inc")

// 带资源节的 PE: .res 是 rc.exe / windres 编译出来的资源树。
let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_auto,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        // .rsrc 必须声明为 format_resources。
        format_section(".rsrc", format_resources | format_readable)
    )
)
format_begin(image)

format_section_begin(image, ".text")
start:
    ret
format_section_end(image, ".text")

// 复制资源树并注册资源目录; 不要在这个调用之外再打开 .rsrc。
format_pe_resource_section(image, ".rsrc", "app.res")
format_entry_mut(image, start)
format_finish(image)

```

生成的映像为 1536 字节：`.rsrc` 节占 364 字节，可选头的资源目录槽（第 3 项数据目录）填成 `RVA = 0x2000`、`size = 364`。

`format_pe_resource_section(plan, name, path)` 按**类型 / 名称 / 语言**三级重排资源树，各级目录、名称字符串和偏移由 `format.inc` 生成，`.res` 里原有的记录顺序不影响结果。`.res` 里没有非空记录时报 `compiled PE resource file contains no non-empty records`。

`path` 的相对路径基准是**读这个文件的代码所在目录**，而读 `.res` 的代码在 `include/format/pe_resource.inc` 里，所以相对路径的基准是 `include/format/`，既不是源文件所在目录，也不是当前工作目录。要找源文件旁边的 `.res`，把路径写成绝对路径，否则会报 `the file this statement reads cannot be opened (FileNotAvailable)`。

基址重定位

基址重定位描述的是「文件内某个槽保存了绝对地址，加载器更换基址时必须修正它」。它与 `rel` 指令引用是两回事：`rel` 引用本身就是相对地址，加载器无需处理；只有写入内存的绝对地址才需要重定位记录。

```

import("format/format.inc")

let image: map = format_pe64(
    // 要求 ASLR，因此必须声明重定位节。
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_required,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".data", format_data | format_readable | format_writeable),
        format_section(".idata", format_imports | format_readable | format_writeable),
    )
)
format_begin(image)

format_section_begin(image, ".text")
start:
    ret
format_section_end(image, ".text")

format_section_begin(image, ".data")
start:
    ret
format_section_end(image, ".data")

format_section_begin(image, ".idata")
start:
    ret
format_section_end(image, ".idata")

format_finish(image)

```

```

        format_section(".reloc", format_fixups | format_readable | format_discardable)
    )
)

let imports: map = format_pe_import_new()
format_pe_import_many_mut(image, imports, "KERNEL32.DLL", list.of("ExitProcess"))

format_begin(image)

format_section_begin(image, ".text")
start:
    // Windows x64 调用前先留出 shadow space。
    sub rsp, 40
    // 从 .data 中取出函数地址再调用。
    mov rax, [rel worker_pointer]
    call rax
    mov ecx, eax
    call [rel ExitProcess]
worker:
    mov eax, 42
    ret
format_section_end(image, ".text")

format_section_begin(image, ".data")
// 该槽保存绝对地址，加载器更换基址时必须修正它。
worker_pointer:
    dq(0)
format_section_end(image, ".data")

format_pe_import_section(image, ".idata", imports)

// 记录的是「槽本身」的地址，不是它指向的 worker。
let relocs: list = pe_reloc_new()
format_pe_reloc_add_mut(image, relocs, worker_pointer)
format_pe_reloc_section(image, ".reloc", relocs)

format_entry_mut(image, start)
format_finish(image)

// 绝对地址必须等全部标签地址稳定后再回填。
defer {
    store.u64(worker_pointer, worker)
}

```

生成的映像为 3072 字节。重定位目录指向 `.reloc`，其中是一个基址重定位块：页 RVA 为 `0x2000`，块大小为 12（8 字节块头，加一条 2 字节记录，再加 2 字节终止项），记录类型为 10，即 `IMAGE_REL_BASED_DIR64`，偏移为 0，合起来指向 `worker_pointer`。`DYNAMIC_BASE` 同时置位，符合 `format_pe_aslr_required` 的要求。

`format_pe_reloc_add_mut` 依据配置中的位宽自动选择 32 位或 64 位重定位类型，不需要手工填写类型码。记录必须按 RVA 升序传给 `format_pe_reloc_section`。PE32 用 `dd(0)` 保存槽、用 `store.u32` 回填，同一个函数会自动选择 32 位类型。

函数	参数	用途
<code>pe_reloc_new()</code>	无	建立空重定位列表

函数	参数	用途
<code>format_pe_reloc_add_mut(plan, relocs, storage)</code>	配置、列表、保存绝对地址的槽地址	追加一条按位宽选择的重定位记录
<code>format_pe_reloc_section(plan, name, relocs)</code>	配置、已声明的重定位节名、已排序列表	生成重定位节并注册重定位目录

观察加载器的修正过程

可执行文件通常看不出重定位有没有生效：加载器按链接时写入的 `ImageBase` 映射，就走不到修正那一步。要观察它，把同样结构写成 DLL，让宿主读回某个绝对指针：

```
import("format/format.inc")

// 把函数地址存进 .data，再由导出函数读回，用来验证加载器是否修正了它。
let image: map = format_pe64(
  format_pe_dll | format_pe_console | format_pe_nx | format_pe_aslr_required,
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".data", format_data | format_readable | format_writeable),
    format_section(".edata", format_exports | format_readable),
    // format_pe_aslr_required 要求必须声明重定位节。
    format_section(".reloc", format_fixups | format_readable | format_discardable)
  )
)

format_begin(image)

format_section_begin(image, ".text")
dll_main:
  // DLL 入口返回非零值表示加载成功。
  mov eax, 1
  ret
worker:
  // 返回固定值，用来证明指针确实指向它。
  mov eax, 42
  ret
read_pointer:
  // 把槽里存着的 worker 地址读回来交给宿主。
  mov rax, [rel worker_pointer]
  ret
format_section_end(image, ".text")

// 槽里写的是 worker 的绝对地址，换基址时必须修正。
format_section_begin(image, ".data")
worker_pointer:
  dq(0)
format_section_end(image, ".data")

let exports: list = format_pe_export_new()
format_pe_export_many_mut(image, exports, list.of("read_pointer"))
format_pe_export_section(image, ".edata", exports, "reloc_probe.dll")

let relocs: list = pe_reloc_new()
format_pe_reloc_add_mut(image, relocs, worker_pointer)
format_pe_reloc_section(image, ".reloc", relocs)
```

```

format_entry_mut(image, dll_main)
format_finish(image)

// 绝对地址要等标签地址稳定后再回填。
defer {
    store.u64(worker_pointer, worker)
}

```

宿主加载它，取回槽里的地址，再调用该地址：

```

#include <windows.h>
#include <stdio.h>

int main(void) {
    HMODULE dll = LoadLibraryA("reloc_probe.dll");
    if (!dll) { printf("load failed: %lu\n", GetLastError()); return 2; }

    /* 取回槽里的地址并调用它：只有加载器改过这个槽，调用才会成功。 */
    int (*read_pointer)(void) = (int (*)(void))GetProcAddress(dll, "read_pointer");
    long long stored = (long long)read_pointer();
    printf("DLL loaded at : %p\n", (void *)dll);
    printf("stored pointer: 0x%11X\n", stored);
    printf("worker() = %d\n", ((int (*)(void))stored)());
    return 0;
}

```

编译时把 DLL 与宿主放在同一目录，运行结果形如：

```

DLL loaded at : 00007FFB4E3E0000
stored pointer: 0x7FFB4E3E1006
worker() = 42

```

加载地址不是链接时写入的 `0x140000000`，说明映像被搬到了别处；而槽里读回的地址落在加载地址范围内，调用它还能返回 42，说明**加载器按 `.reloc` 修正了这个绝对地址**。如果记录写错位置，读回的会是链接时的旧地址，调用它就会崩溃。

校验和

PE 的 `Checksum` 字段覆盖整个文件，因此只能在文件字节定型之后计算：

```

import("format/format.inc")

let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
    list.of(
        format_section(".text", format_code | format_readable | format_executable)
    )
)

```

```

format_begin(image)

format_section_begin(image, ".text")
start:
    xor eax, eax
    ret
format_section_end(image, ".text")

format_entry_mut(image, start)
format_finish(image)

// 校验和依赖最终文件字节，因此放在最后一步。
format_pe_checksum(image)

```

`format_pe_checksum` 从第一个节的开头度量到最后一个节的结尾，因此调用前至少要有有一个节，否则报 `PE checksum requires at least one section`。上例的文件是 1024 字节，`Checksum` 字段为 `0x4A36`。

3. Linux ELF 可执行文件和共享库

要在 Linux 上生成可直接运行的文件，或者要被其他程序动态加载的 `.so`，用 ELF 格式。ELF 映像不像 PE 那样按节组织内容，它按**装载段**组织：每一段记录自己在文件里的位置、映射到的虚拟地址、占多少文件字节、占多少内存、运行时是什么权限。加载器只读程序头，按这些记录把文件映射进内存。

节名和装载段名的用处不同：节名给链接器和调试器看，装载段名给加载器看。用 `format.inc` 时声明的是装载段，程序头由它生成，`p_vaddr == p_offset (mod p_align)` 这个加载器依赖的关系也由它维持。

ELF 可执行文件

固定地址可执行文件用 `format_elf_exec`：

```

import("format/format.inc")

// 三段：代码、已初始化数据、零初始化数据。
let image: map = format_elf64(
    format_elf_exec,
    list.of(
        // 代码段可读可执行。
        format_segment(".text", format_load | format_readable | format_executable),
        // 数据段可读可写。
        format_segment(".data", format_load | format_readable | format_writeable),
        // BSS 段可读可写，但只占内存。
        format_segment(".bss", format_load | format_readable | format_writeable)
    )
)
format_begin(image)

format_segment_begin(image, ".text")

```

```

start:
    // 系统调用 60: exit(0)。
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image, ".text")

format_segment_begin(image, ".data")
answer:
    dd(42)
format_segment_end(image, ".data")

format_segment_begin(image, ".bss")
scratch:
    rb(128)
format_segment_end(image, ".bss")

format_entry_mut(image, start)
format_finish(image)

```

这个例子是 253 字节，三个 `PT_LOAD` 程序头：

装载段	文件偏移	虚拟地址	文件大小	内存大小	权限
<code>.text</code>	<code>0xF0</code>	<code>0x4000F0</code>	9	9	<code>R X</code>
<code>.data</code>	<code>0xF9</code>	<code>0x4010F9</code>	4	4	<code>R W</code>
<code>.bss</code>	<code>0xFD</code>	<code>0x4020FD</code>	0	128	<code>R W</code>

`.bss` 那一行把 BSS 的含义说全了：文件里一个字节都不占，内存里保留 128 字节。`format.inc` 把这两个数字分别写进程序头的 `p_filesz` 和 `p_memsz`，加载器据此把这段内存清零。

ELF32 把 `format_elf64` 换成 `format_elf32`，选项仍只能是 `format_elf_exec`：这一层没有 ELF32 PIE 入口。

一个完整的程序

前面的例子只把寄存器清零就退出，不足以说明实际程序如何编写。下面这个程序打印一行文本，并用退出码返回计算结果——退出码是脚本和 CI 最容易核对的一项结果：

```

import("format/format.inc")

// Linux x86-64 实际程序：算出结果，打印一行，用退出码报告状态。
let image: map = format_elf64(
    format_elf_exec,
    list.of(
        // 代码段：可读可执行。
        format_segment(".text", format_load | format_readable | format_executable),
        // 只读数据段：提示文本。
        format_segment(".rodata", format_load | format_readable)
    )
)

```

```

// 从输入值算出结果。
const INPUT: u64 = 35
const DELTA: u64 = 7

format_begin(image)

format_segment_begin(image, ".text")
start:
    // 算入参: lea 在不改动标志位的前提下做加法。
    lea r12, [INPUT + DELTA]

    // write(1, message, length)
    mov eax, 1
    mov edi, 1
    lea rsi, [rel message]
    lea rdx, [rel message_end]
    sub rdx, rsi
    syscall

    // exit(result)
    mov edi, r12d
    mov eax, 60
    syscall
format_segment_end(image, ".text")

format_segment_begin(image, ".rodata")
message:
    db("xirasm: exit status carries the computed result", 10)
// 尾部标号用来算长度, 不用手数字节。
message_end:
format_segment_end(image, ".rodata")

format_entry_mut(image, start)
format_finish(image)

```

这个例子是 271 字节，两个 `PT_LOAD` 段：`.text` 在偏移 `0xB0`，47 字节，权限 `R|X`；`.rodata` 在偏移 `0xDF`，48 字节，权限 `R`——代码段和数据段分开，只读数据不带执行权限。

在 x86-64 Linux 上直接运行，终端打印：

```
xirasm: exit status carries the computed result
```

退出码是 42，也就是 `INPUT + DELTA`。

这段代码里有两条实际写法值得记住：

- **字符串长度用标号算。**`lea rdx, [rel message_end]` 再减去 `rsi`，长度由汇编器算出来；写死字节数，改文本时就得跟着改数字。
- **标签跨段引用要写 `rel`。**`.text` 里的代码引用 `.rodata` 的标号，必须用 RIP 相对形式，这样装载段落在哪个地址都成立。

ELF64 PIE

位置无关可执行文件用 `format_elf_pie`:

```
import("format/format.inc")

let image: map = format_elf64(
    format_elf_pie,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable),
        format_segment(".bss", format_load | format_readable | format_writeable),
        format_segment(".rodata", format_load | format_readable)
    )
)
format_begin(image)

format_segment_begin(image, ".text")
start:
    // 加载器可能把整个映像搬到别处，所以标签一律走 RIP 相对引用。
    lea rbx, [rel scratch]
    lea rsi, [rel message]
    mov dword [rbx], 0x5a
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image, ".text")

format_segment_begin(image, ".bss")
scratch:
    rb(64)
format_segment_end(image, ".bss")

format_segment_begin(image, ".rodata")
message:
    db("XIRASM PIE", 0)
format_segment_end(image, ".rodata")

format_entry_mut(image, start)
format_finish(image)
```

PIE 与固定地址可执行文件只有一个实质区别：基址由加载器决定。因此**所有指向映像内部的引用都必须写成 RIP 相对形式** (`[rel label]`)，这种引用编码的是位移，不管映像被搬到哪里都成立，也不需要动态重定位。反过来，把绝对指针存进 `.data` 就需要一条动态重定位记录，而任意用户指针的动态重定位目前要通过 ELF 直接构造来完成，`format.inc` 这一层不提供。

PIE 不支持可执行文件导入。把导入表挂到 PIE 配置上会报 `ELF executable imports require fixed-address EXEC mode`。

ELF64 可执行文件导入

固定地址可执行文件可以通过 PLT 条目调用外部函数。声明导入之后，`format.inc` 会生成动态段、PLT、GOT 以及配套的重定位：

```
import("format/format.inc")

let image: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable)
    )
)

// 每个库调用一次分组函数，共用一个导入列表。
let imports: list = format_elfexe_import_new()
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("getpid", "getppid"))
// cos 以本地前缀 cos_fn 导入。
format_elfexe_import_pairs_mut(imports, "libm.so.6", list.of("cos_fn", "cos"))
format_elfexe_tables_mut(image, imports)

format_begin(image)

format_segment_begin(image, ".text")
start:
    // 调用生成出来的 PLT 标签。
    call getpid_plt
    call getppid_plt
    xor edi, edi
    mov eax, 60
    syscall
format_segment_end(image, ".text")

format_entry_mut(image, start)
format_finish(image)
```

生成的映像为 960 字节，动态段里有两条 **NEEDED** (**libc.so.6** 和 **libm.so.6**)，动态符号表里有三个未定义函数 **getpid**、**getppid**、**cos**。

声明导入会让文件多出装载段。上例只声明了 **.text** 一个装载段，最终的程序头有五个：

程序头	文件偏移	虚拟地址	文件大小	权限	来源
PT_LOAD	0x0	0x400000	371	R X	声明的 .text
PT_LOAD	0x180	0x401180	64	R X	自动生成，放 PLT
PT_LOAD	0x1C0	0x4021C0	512	R W	自动生成，放 GOT、动态符号表、字符串表、hash、RELA 和动态段
PT_INTERP	0x1F0	0x4021F0	28	R	动态链接器路径
PT_DYNAMIC	0x300	0x402300	192	R W	动态段本身

后两个装载段排在所有声明段之后，各自页对齐。因此文件大小和地址布局不能只按声明的段来算——导入表本身要占一个可写段。动态链接器的路径固定为 **/lib64/ld-linux-x86-64.so.2**，写在 **PT_INTERP** 指向的位置。

可写装载段里的内容按固定顺序排布：GOTPLT、解释器路径、动态符号表、动态字符串表、hash、**RELA.PLT**、动态段。这个顺序不是可以调整的选项，**format.inc** 逐个对齐着写，因此段内偏移是可预测的。

标签的生成规则：**many** 用导入名本身，**getpid** 生成 **getpid_plt** 和 **getpid_gotplt**；**pairs** 的列表按「本地前缀, 真实符号名」成对给出，上例的 **cos_fn** 生成 **cos_fn_plt** 和 **cos_fn_gotplt**。**pairs** 的项数必须是偶数，上限 128。

format_elfexe_tables_mut 必须在 **format_begin** 之前调用：动态段、PLT 和 GOT 的空间要在开始输出时就预留出来。

调用 **libc** 的实际写法是给参数寄存器赋值后 **call** 生成的 PLT 标号。下面这个程序用 **write** 打印一行，再把 **getpid** 的返回值当退出码：

```
import("format/format.inc")

// 带 libc 导入的实际程序：用 write 打印，用 getpid 生成结果。
let image: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable),
        format_segment(".rodata", format_load | format_readable)
    )
)

let imports: list = format_elfexe_import_new()
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("write", "getpid"))
format_elfexe_tables_mut(image, imports)

format_begin(image)

format_segment_begin(image, ".text")
start:
    // write(1, message, length): 经 PLT 调用 libc。
    mov edi, 1
    lea rsi, [rel message]
    lea rdx, [rel message_end]
    sub rdx, rsi
    call write_plt

    // getpid(): 返回当前进程号。
    call getpid_plt

    // 用退出码把这个值带出去，方便脚本核对。
    mov edi, eax
    mov eax, 60
    syscall
format_segment_end(image, ".text")

format_segment_begin(image, ".rodata")
message:
    db("xirasm: write() came from libc through the PLT", 10)
message_end:
format_segment_end(image, ".rodata")
```

```
format_entry_mut(image, start)
format_finish(image)
```

生成的映像为 960 字节。程序在 x86-64 Linux 上打印 `xirasm: write() came from libc through the PLT`，退出码是当前进程号（每次运行不同，属于正常现象）。

`readelf -d` 显示动态段里的 `NEEDED` 一条指向 `libc.so.6`，`PLTRELSZ` 为 48 字节——两个导入函数各占一条 24 字节的 `RELA` 记录，`PLTREL` 为 `RELA`。`ldd` 列出三个依赖：`linux-vdso.so.1`、`libc.so.6`、`/lib64/ld-linux-x86-64.so.2`，最后一项来自程序头中 `PT_INTERP` 指定的解释器，由 `format.inc` 填为 `/lib64/ld-linux-x86-64.so.2`。

两条实际写法：先按 System V AMD64 的寄存器顺序放好参数（`rdi`、`rsi`、`rdx`……）再 `call`，参数个数或顺序写错不会报错，只会得出错误结果；调用返回后先取 `eax` 里的返回值，再据此判断后续动作，上例把 `eax` 搬到 `edi` 就是为了当退出码。

ELF64 共享库

共享库没有可执行入口，它对外暴露动态符号，同时也可以自己导入符号。SONAME 作为参数传给构造函数：

```
import("format/format.inc")

let image: map = format_elf64_so(
    "libxirasm_demo.so",
    list.of(
        format_segment(".text", format_load | format_readable | format_executable)
    )
)

let exports: list = format_elfso_export_new()
// 两个函数按标签原名导出，符号大小都是 4 字节。
format_elfso_export_many_mut(exports, list.of("x_add7", "x_sub3"), ".text", 4)
// answer_impl 以 x_answer 之名导出，符号大小 6 字节。
format_elfso_export_pairs_mut(exports, list.of("answer_impl", "x_answer"), ".text", 6)
format_elfso_tables_mut(image, exports, list.new())

format_begin(image)

format_segment_begin(image, ".text")
x_add7:
    lea eax, [rdi + 7]
    ret
x_sub3:
    lea eax, [rdi - 3]
    ret
answer_impl:
    mov eax, 42
    ret
format_segment_end(image, ".text")

format_finish(image)
```

导出声明要给出所在装载段名和符号大小。`.dynsym`、`.dynstr`、`hash`、`.dynamic`、`GOT/PLT` 这些载荷由 `format.inc` 生成。共享库同样不调用 `format_entry_mut`。

共享库带节头表，可执行文件不带。同样是 x86-64 下由 `format.inc` 生成的文件，两者在头部和布局上的差别如下：

项目	ELF 可执行文件	ELF 共享库
文件类型	<code>ET_EXEC</code> 或 PIE 的 <code>ET_DYN</code>	<code>ET_DYN</code>
节头表	没有 (<code>e_shnum = 0</code>)	有，共 7 项：空节、 <code>.text</code> 、 <code>.dynsym</code> 、 <code>.dynstr</code> 、 <code>.hash</code> 、 <code>.dynamic</code> 、 <code>.shstrtab</code>
段基址	<code>0x400000</code> 起	<code>0</code> 起
段对应关系	每个声明的装载段一个 <code>PT_LOAD</code>	声明的段合进 <code>PT_LOAD</code> ，另有 <code>PT_DYNAMIC</code>

共享库的地址从 0 起算，是因为它被加载到哪个地址由加载器决定，节头里的地址都按基址 0 记录。可执行文件没有节头表，`readelf -S` 不会有输出；要用 `readelf -l` 看程序头，或者用 `llvm-readobj --program-headers`。

ELF64 共享库导入

共享库的导入用同一套「按库分组」写法，区别是导入和导出共用一张 `tables_mut`：

```
let imports: list = format_elfso_import_new()
format_elfso_import_many_mut(imports, "libc.so.6", list.of("puts", "getpid"))
format_elfso_import_pairs_mut(imports, "libm.so.6", list.of("cos_fn", "cos"))
format_elfso_tables_mut(image, exports, imports)
```

导入名和导出名不能冲突，生成出来的 `*_plt` 和 `*_gotplt` 标签也必须唯一。映像内部的引用写 `rel`，调用导入函数用生成出来的 PLT 标签。

选择目标机器

`format_elf64` 和 `format_elf64_so` 默认生成 x86-64 映像。AArch64 另有入口，机器名来自配置时可以用带机器参数的通用形式：

入口	产物
<code>format_elf64(options, segments)</code>	ELF64 x86-64 可执行文件或 PIE
<code>format_elf64_aarch64(options, segments)</code>	ELF64 AArch64 可执行文件或 PIE
<code>format_elf64_machine(options, segments, machine)</code>	机器由 <code>elf_machine_x86_64</code> 或 <code>elf_machine_aarch64</code> 指定
<code>format_elf64_so(soname, segments)</code>	ELF64 x86-64 共享库，LOAD 段按 4 KiB 对齐
<code>format_elf64_so_aarch64(soname, segments)</code>	ELF64 AArch64 共享库，LOAD 段按 16 KiB 对齐
<code>format_elf64_so_machine(soname, segments, machine)</code>	机器由参数指定的 ELF64 共享库
<code>format_elfobj64_aarch64(sections)</code>	AArch64 ELF64 目标文件

两种机器有三处差别：

差别	x86-64	AArch64
LOAD 段页对齐	4 KiB，Linux 默认值	16 KiB ，Android 15 及以后设备的要求
导入方式	PLT 条目	GLOB_DAT 槽位，用 <code>format_elfso_import_slots_mut</code> 声明
可执行文件导入	支持	不支持，改由共享库导入

页对齐调大只在装载段必须挪到下一个页边界时才多占文件字节：虚拟地址按整页前进，文件布局仍然紧凑。要换别的页大小，在 `format_begin` 之前给配置设置 `"load_align"`。

每个装载段还会遵守所属节声明的对齐：`format_executable` 段是 16 字节，其余是 8 字节。这个对齐同时作用在文件偏移上，因为加载器要求 `p_vaddr == p_offset (mod p_align)`——地址低位由文件偏移决定，只调地址会破坏这个关系。结果是每个节都满足 `sh_addr % sh_addralign == 0`，做对齐访问的代码依赖的正是这条保证；代价是装载段原本会落在页内进位处时，文件里多几个填充字节。

API 摘要

函数	用途
<code>format_elf32(format_elf_exec, segments)</code>	ELF32 可执行文件
<code>format_elf64(format_elf_exec, segments)</code>	ELF64 x86-64 固定地址可执行文件
<code>format_elf64(format_elf_pie, segments)</code>	ELF64 x86-64 PIE
<code>format_elf64_aarch64(options, segments)</code>	ELF64 AArch64 可执行文件或 PIE
<code>format_elf64_machine(options, segments, machine)</code>	指定机器的 ELF64 可执行文件或 PIE
<code>format_elf64_so(soname, segments)</code>	ELF64 x86-64 共享库
<code>format_elf64_so_aarch64(soname, segments)</code>	ELF64 AArch64 共享库，按 Android 页大小对齐
<code>format_elf64_so_machine(soname, segments, machine)</code>	指定机器的 ELF64 共享库
<code>format_elfobj64_aarch64(sections)</code>	AArch64 ELF64 目标文件
<code>format_elfexe_import_new()</code>	创建 ELF64 可执行文件导入列表
<code>format_elfexe_import_many_mut(imports, library, names)</code>	从同一个库批量加入同名导入
<code>format_elfexe_import_pairs_mut(imports, library, pairs)</code>	加入「本地前缀, 真实符号名」成对导入
<code>format_elfexe_tables_mut(plan, imports)</code>	把可执行文件导入元数据挂到配置上
<code>format_elfso_export_new()</code>	创建共享库导出列表
<code>format_elfso_export_many_mut(exports, names, segment, size)</code>	批量导出同名符号
<code>format_elfso_export_pairs_mut(exports, pairs, segment, size)</code>	批量导出「内部标签, 公开名称」成对符号
<code>format_elfso_import_new()</code>	创建共享库导入列表
<code>format_elfso_import_many_mut(imports, library, names)</code>	从同一个库批量加入共享库导入

函数	用途
<code>format_elfso_import_pairs_mut(imports, library, pairs)</code>	加入共享库「本地前缀, 真实符号名」成对导入
<code>format_elfso_import_slots_mut(imports, library, names)</code>	AArch64 共享库 GLOB_DAT 导入槽位
<code>format_elfso_tables_mut(plan, exports, imports)</code>	把共享库动态元数据挂到配置上

4. COFF 和 ELF 目标文件

发给链接器的文件，和能直接运行的文件，是两回事。目标文件里没有入口地址，也没有装载段：操作系统不会加载它，它只负责说明「有哪些节、哪些符号要让链接器看见、哪些字段要留给链接器回填」，由 `link.exe`、`ld.lld` 之类的工具把多个目标文件拼成一个可执行文件。

用 `format.inc` 生成目标文件时，前面几章的顺序有一处不同：先声明节、写内容，**再**把符号表和重定位表挂到配置上，最后 `format_finish`。符号表和重定位表就是目标文件的全部意义，因此它们不是可选装饰，而是必须提供的内容。

COFF 目标文件模板

COFF 目标文件供 Windows 的 MSVC 风格链接器使用，支持代码、数据、BSS 三类节，节名不能超过八字节。

```
import("format/format.inc")

// 面向 Windows/MSVC 风格链接器的 64 位 COFF 目标文件。
let object: map = format_coff64(
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".data", format_data | format_readable | format_writeable),
    format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
  )
)
format_begin(object)

// .text 里留一个 call 位移占位，链接器负责回填。
format_section_begin(object, ".text")
text_start:
main:
  db(0xe8)
call_disp:
  dd(0)
  xor eax, eax
  ret
format_section_end(object, ".text")
```

```

format_section_begin(object, ".data")
data_start:
answer:
    dd(42)
format_section_end(object, ".data")

format_section_begin(object, ".bss")
bss_start:
scratch:
    rb(64)
format_section_end(object, ".bss")

// 声明公开符号和外部符号：符号值是「地址 - 所在节起始」。
const symbols: list = list.of(
    format_coff_public("main", ".text", text_start, main, coff_sym_type_function),
    format_coff_public("answer", ".data", data_start, answer, coff_sym_type_null),
    format_coff_public("scratch", ".bss", bss_start, scratch, coff_sym_type_null),
    format_coff_extern("puts", coff_sym_type_function)
)

// 重定位指向 call_disp 这 4 字节占位，类型取 32 位相对调用。
const relocs: list = list.of(
    format_coff_reloc(".text", text_start, call_disp, "puts", coff_rel_amd64_rel32)
)
format_coff_tables_mut(object, symbols, relocs)
format_finish(object)

```

生成的映像为 240 字节，符号表里有四个符号：`main` 落在 `.text`、`answer` 落在 `.data`、`scratch` 落在 `.bss`，三个值都是 0（各自节内的偏移），而 `puts` 落在 `IMAGE_SYM_UNDEFINED`，类型是函数，等待链接器解析。

两条参数约定：

- `format_coff_public(name, section_name, section_start, address, sym_type)` 的 `address` 是符号标签，`section_start` 是它所在节的起始标签，写进符号表的值是两者之差；
- `format_coff_reloc(section_name, section_start, address, symbol_name, reloc_type)` 的 `address` 是需要链接器修正的占位，不是目标符号。上例的 `call_disp` 就是 `call rel32` 那条 4 字节位移。

COFF 符号名和重定位引用的符号名都不能超过八字节，超出报 `COFF symbol names must fit in eight bytes` 或 `COFF relocation symbol names must fit in eight bytes`。符号名重复报 `COFF symbol name is duplicated`，重定位引用了没声明的符号报 `COFF relocation target symbol is not declared`。

这八字节限制在构造符号时就检查，因此这一层不会生成 COFF 的字符串表间接名：名字直接写进 8 字节字段，后面用零补齐，符号表末尾的字符串表只留 4 字节头。需要超过八字节的符号名时，走高级格式构造。

常用相对调用重定位：

位宽	重定位类型
32 位	<code>coff_rel_i386_rel32</code>
64 位	<code>coff_rel_amd64_rel32</code>
AArch64	<code>coff_rel_arm64_*</code> 系列

ELF 目标文件模板

ELF 目标文件供 `ld`、`ld.lld` 这类链接器使用，节名不受八字节限制。

```
import("format/format.inc")

// ELF 目标文件的节名可以超过 8 字节。
let object: map = format_elfobj64(
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".bss", format_uninitialized_data | format_readable |
format_writeable),
        format_section(".rodata", format_data | format_readable)
    )
)
format_begin(object)

format_section_begin(object, ".text")
text_start:
_start:
    db(0xe8)
call_disp:
    dd(0)
    xor eax, eax
    ret
format_section_end(object, ".text")

format_section_begin(object, ".bss")
bss_start:
scratch:
    reserve(64)
format_section_end(object, ".bss")

format_section_begin(object, ".rodata")
data_start:
answer:
    dd(42)
format_section_end(object, ".rodata")

// ELF 公开符号除名称、节、节起始、地址外，还要给符号大小和类型。
const symbols: list = list.of(
    format_elfobj_public("_start", ".text", text_start, _start, 8, elfobj_stt_func),
    format_elfobj_public("scratch", ".bss", bss_start, scratch, 64, elfobj_stt_object),
    format_elfobj_public("answer", ".rodata", data_start, answer, 4, elfobj_stt_object),
    format_elfobj_extern("puts", elfobj_stt_func)
)

// RELA 记录带显式 addend: -4 就是 rel32 位移字段自身的宽度。
const relocs: list = list.of(
```

```
format_elfobj_reloc(".text", text_start, call_disp, "puts", elf_r_x86_64_plt32,
0xfffffffffffffffffc)
)
format_elfobj_tables_mut(object, symbols, relocs)
format_finish(object)
```

生成的映像为 992 字节，类型是 `ET_REL`。符号表里有 `_start`（函数，大小 8）、`scratch`（数据，大小 64）、`answer`（数据，大小 4）和未定义的 `puts`；重定位表里是一条 `R_X86_64_PLT32 puts 0xFFFFFFFFFFFFFFFFC`。

ELF64 用的是 RELA 记录，addend 显式写在记录里。上面这条的 addend 是 `-4`，也就是 `0xfffffffffffffffffc`——它就是 `rel32` 位移字段自身的宽度，来自 x86-64 psABI 的约定。这是 ELF 目标文件相对 COFF 最容易出错的一处：COFF 的位移里已经包含了「相对下一条指令」，ELF 则把它交给 addend。

`format_elfobj_public` 比 COFF 多两个参数：符号大小和符号类型（`elfobj_stt_notype`、`elfobj_stt_object`、`elfobj_stt_func`、`elfobj_stt_section`）。符号值同样是「地址 - 所在节起始」。

符号类型和绑定在文件里合成一个字节：类型决定这个符号是函数还是数据，绑定固定为全局（`STB_GLOBAL`）。本地绑定只用在自动生成的节符号上，用户声明的符号没有本地符号这一档。

符号表的排布决定了重定位引用的下标：

1. 下标 0 是保留的空符号；
2. 接着每个节一个节符号，`shndx` 是节号，下标从 1 到节数；
3. 用户声明的符号排在后面，所以第一个用户符号的下标是 `1 + 节数`。

重定位记录里的 `r_info` 高 32 位就是按这个下标写进去的。用 `readelf -s` 核对时，用户符号显示在节符号之后。

链接器接手之后

目标文件的用处是被别的代码链进去，最直接的检验就是真正链接一次。下面导出一个函数给 C 调用：

```
import("format/format.inc")

// 与 C 混编用的 ELF64 目标文件：导出一个函数给 C 调用。
let object: map = format_elfobj64(
  list.of(
    format_section(".text", format_code | format_readable | format_executable)
  )
)
format_begin(object)

format_section_begin(object, ".text")
text_start:
```

```

xirasm_double:
    // long xirasm_double(long v) { return v * 2; }
    lea rax, [rdi + rdi]
    ret
format_section_end(object, ".text")

const symbols: list = list.of(
    format_elfobj_public("xirasm_double", ".text", text_start, xirasm_double, 4,
elfobj_stt_func)
)
format_elfobj_tables_mut(object, symbols, list.new())
format_finish(object)

```

生成的映像为 600 字节。写一个 C 文件声明同一个函数名，两个文件一起交由 **gcc** 链接：

```

#include <stdio.h>

/* xirasm_double 由 XIRASM 生成的 ELF64 目标文件提供。 */
long xirasm_double(long value);

int main(void) {
    /* 调用目标文件里的函数，并核对它返回 42。 */
    long result = xirasm_double(21);
    printf("xirasm_double(21) = %ld\n", result);
    return result == 42 ? 0 : 1;
}

```

```

gcc -o mix main.c mix.o
./mix

```

程序打印 **xirasm_double(21) = 42** 并以 0 退出。能链上就说明符号表里的名字、类型和节号都对，链接器认可这份目标文件。

跨语言调用要按 ABI 对齐：x86-64 Linux 的第一个整型参数放 **rdi**，返回值放 **rax**。上例的 **lea rax, [rdi + rdi]** 就是「把第一个参数乘 2 后返回」。参数类型和寄存器不符，链接一样能过，运行结果却会错——ABI 由调用双方约定，链接器管不了。

readelf -s mix.o 用来核对符号表：**xirasm_double** 的 **Ndx** 指向 **.text**，**Type** 为 **FUNC**，**Size** 为 4。

目标文件还可以先反汇编再链接。**llvm-objdump -d** 按符号表和重定位把机器码还原出来，能同时验证指令字节、符号值和重定位记录三者是否一致——这一步不需要先链成功：

```

llvm-objdump -d mix.o
objdump -dr mix.o

```

objdump -dr 会同时打印反汇编和重定位记录，是核对「占位字节 + 重定位」这一对是否对得上的常用手段。

选择 ELF 机器类型

构造函数	e_machine	说明
<code>format_elfobj32(sections)</code>	<code>EM_386</code>	同时把汇编器切换到 32 位 x86 文本模式
<code>format_elfobj64(sections)</code>	<code>EM_X86_64</code>	同时把汇编器切换到 64 位 x86 文本模式
<code>format_elfobj64_aarch64(sections)</code>	<code>EM_AARCH64</code>	面向 arm64 Linux 与 BSD
<code>format_elfobj64_machine(sections, machine)</code>	由参数指定	接受 <code>elf_machine_x86_64</code> 或 <code>elf_machine_aarch64</code>

AArch64 形式不会切换 x86 文本模式，因此指令字要当数据写：用 `emit.u32` 写，或者走 AArch64 的 DSL 层。重定位类型从 `elf_const.inc` 取：

```
import("format/format.inc")

// AArch64 目标文件：指令字当数据写，每条重定位标明一处占位。
let object: map = format_elfobj64_aarch64(
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".data", format_data | format_readable | format_writeable)
  )
)
format_begin(object)

format_section_begin(object, ".text")
text_start:
  // bl printf: 低 26 位留给链接器填。
  bl_site:
    emit.u32(0x94000000)
  // adrp x0, answer: 页基址部分。
  page_site:
    emit.u32(0x90000000)
  // add x0, x0, :lo12:answer: 低 12 位部分。
  lo12_site:
    emit.u32(0x91000000)
format_section_end(object, ".text")

format_section_begin(object, ".data")
data_start:
  // .xword printf: 8 字节绝对地址，链接器整体回填。
  ptr_slot:
    emit.u64(0)
format_section_end(object, ".data")

const symbols: list = list.of(
  format_elfobj_public("_start", ".text", text_start, bl_site, 16, elfobj_stt_func),
  format_elfobj_public("answer", ".data", data_start, ptr_slot, 8, elfobj_stt_object),
  format_elfobj_extern("printf", elfobj_stt_func)
)

// 每条重定位描述链接器要修正的占位字；AArch64 用 RELA, addend 显式给出。
const relocs: list = list.of(
```

```
format_elfobj_reloc(".text", text_start, bl_site, "printf", elf_r_aarch64_call126, 0),
format_elfobj_reloc(".text", text_start, page_site, "answer",
elf_r_aarch64_adr_prel_pg_hi21, 0),
format_elfobj_reloc(".text", text_start, lo12_site, "answer",
elf_r_aarch64_add_abs_lo12_nc, 0),
format_elfobj_reloc(".data", data_start, ptr_slot, "printf", elf_r_aarch64_abs64, 0)
)
format_elfobj_tables_mut(object, symbols, relocs)
format_finish(object)
```

生成的映像为 1016 字节，`e_machine` 是 `EM_AARCH64`（183），重定位表里有四条记录：偏移 0 的 `R_AARCH64_CALL26`、偏移 4 的 `R_AARCH64_ADR_PREL_PG_HI21`、偏移 8 的 `R_AARCH64_ADD_ABS_LO12_NC`，加上 `.data` 里偏移 0 的 `R_AARCH64_ABS64`。

一条 `adrp` 加一条 `add` 组合出符号地址时，两条指令各要一条重定位：页基址用 `elf_r_aarch64_adr_prel_pg_hi21`，低 12 位用 `elf_r_aarch64_add_abs_lo12_nc`。

`elf_const.inc` 里常用的 AArch64 类型还有：`elf_r_aarch64_abs64`、`elf_r_aarch64_abs32`、`elf_r_aarch64_adr_prel_lo21`、`elf_r_aarch64_ldst{8,16,32,64}_abs_lo12_nc` 系列、`elf_r_aarch64_condbr19`、`elf_r_aarch64_jump26`，以及 GOT 形式的 `elf_r_aarch64_adr_got_page` 和 `elf_r_aarch64_ld64_got_lo12_nc`。

ELF32 用 `format_elfobj32`，常用的 32 位相对调用重定位是 `elf_r_386_pc32`。

Mach-O 目标文件

Mach-O 可重定位目标文件由第 6 章的 `format_macho64_object(target, sections)` 生成，节生命周期与上面两个模板一致。符号表、重定位表和 `LC_SYMTAB` 也在那边讲。

API 摘要

家族	函数	用途
COFF	<code>format_coff32(sections) / format_coff64(sections)</code>	创建 COFF 目标文件配置
COFF	<code>format_coff64_arm64(sections)</code>	创建 AArch64 COFF 目标文件配置
COFF	<code>format_coff_public(name, section_name, section_start, address, sym_type)</code>	声明公开符号
COFF	<code>format_coff_extern(name, sym_type)</code>	声明外部符号

家族	函数	用途
COFF	<code>format_coff_reloc(section_name, section_start, address, symbol_name, reloc_type)</code>	声明需要链接器修正的字段
COFF	<code>format_coff_tables_mut(plan, symbols, relocs)</code>	把符号表和重定位表挂到配置上
ELF	<code>format_elfobj32(sections) / format_elfobj64(sections)</code>	创建 x86 目标文件配置
ELF	<code>format_elfobj64_aarch64(sections)</code>	创建 AArch64 目标文件配置
ELF	<code>format_elfobj64_machine(sections, machine)</code>	以显式机器类型创建目标文件配置
ELF	<code>format_elfobj_public(name, section_name, section_start, address, symbol_size, symbol_type)</code>	声明公开符号
ELF	<code>format_elfobj_extern(name, symbol_type)</code>	声明外部符号
ELF	<code>format_elfobj_reloc(section_name, section_start, address, symbol_name, reloc_type, addend)</code>	声明需要链接器修正的字段
ELF	<code>format_elfobj_tables_mut(plan, symbols, relocs)</code>	把符号表和重定位表挂到配置上

重定位指向的字节必须先写出来。先写占位，再用 `format_*_reloc` 描述这段占位该由链接器怎样修正；反过来，对没有写出的字节声明重定位，链接器拿不到可以覆盖的位置。

5. 通用规则和常见错误

格式文件出错时，绝大多数问题不在 PE、ELF 的字段细节上，而在声明本身：节名写错、一个节给了多种用途、权限给得与运行时需求不符、调用顺序颠倒。头部和表项由 `format.inc` 依据声明生成；声明与意图不一致时，产出的文件在结构上依然合法，只是并非所需。

下面按「先看错误是什么样，再讲如何避免」的顺序排列。每条规则都附一条真实诊断或可核对的结果，可直接用于对照源文件。

不要混用两套写法

正常程序只导入一个文件：

```
import("format/format.inc")
```

不要在同一文件里既用 `format.inc` 的生命周期函数，又手写 PE/ELF 头部或表项。两套写法同时负责同一张表时，最终文件里会出现两个都要填同一个字段的值，先写入的被覆盖，而无法判断哪一个生效。需要完全控制字段时，整份文件都改用[高级格式构造指南](#)。

节和装载段的混用也会被抓出来，因为两种计划的键不同：ELF 映像配 `format_section` 报 `an ELF image plan declares segments, not sections`；PE 配 `format_segment` 报 `only an ELF image plan declares segments`。

可变配置必须是 `let`

`_mut` 结尾的函数要改写传入的绑定，所以该实参必须是一个可以改写的 `let`：

```
import("format/format.inc")

// 配置要被 _mut 函数改写，所以用 let 而不是 const。
let image: map = format_pe64(
  format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
  list.of(
    format_section(".text", format_code | format_readable | format_executable)
  )
)
format_begin(image)
format_section_begin(image, ".text")
start:
  ret
format_section_end(image, ".text")
format_entry_mut(image, start)
format_finish(image)
```

下面三种写法都不合法，报错各不相同：

写法	诊断
<code>const image: map = format_pe64(...)</code> 之后调 <code>_mut</code>	<code>mutable function argument must resolve to a let binding</code>
直接把构造函数的结果传给 <code>_mut</code>	<code>mutable function argument must be a direct let binding</code>
传 <code>map.get(holder, "image")</code> 这类字段取值	同上，解析不到绑定

区别在于：只读的描述值可以放进 `const`，例如 `format_section(...)`、`format_segment(...)`、`format_coff_public(...)` 的返回值；会被 `_mut` 函数改写的配置或集合必须用 `let`。

先声明，再写内容

传给 `format_section_begin`、`format_section_end`、`format_segment_begin`、`format_segment_end` 的名字，必须已经出现在配置的节或装载段列表里：

```
// 配置里声明了哪些，就只能写哪些。
format_section(".text", format_code | format_readable | format_executable)
```

写一个没有声明过的节，报 `format section is not declared`。该诊断指向 `format_section_begin` 那一行，但需要修改的是配置里遗漏的声明。

每个描述只选一个用途

一个节只能是代码、数据、BSS、导入、导出、资源、基址重定位中的一种。同时给两种用途，报 `format section has multiple purposes`：

```
// 不合格：一个节同时声称是代码和数据。
format_section(".mixed", format_code | format_data | format_readable)
```

用途之外的部分是权限，可以叠加；用途本身不能叠加。ELF 装载段目前只有 `format_load` 一种用途，没有第二个可选项。

权限按运行时需求给

内容	常见权限
指令	可读、可执行
常量和字符串	可读
可修改数据	可读、可写
BSS	可读、可写
PE 导入表	可读、可写
PE 基址重定位表	可读、可丢弃

除了自修改代码或特殊加载场景，不要把代码节标成可写，也不要将数据节标成可执行。多给的权限不会报错，但会让加载器放开本来该拦住的写入或执行。

BSS 是内存大小，不是文件内容

`format_uninitialized_data` 声明的是「运行时需要一段零初始化内存」。这一节里用 `rb(...)`、`reserve(...)` 这类预留操作登记大小：

```
import("format/format.inc")

// BSS 的正确写法：只用 rb() 预留大小，不写初始化字节。
let image: map = format_pe64(
  format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_disabled,
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
  )
)
format_begin(image)

format_section_begin(image, ".text")
start:
  ret
format_section_end(image, ".text")

// rb(8) 只登记内存大小，文件里不占字节。
format_section_begin(image, ".bss")
scratch:
  rb(8)
format_section_end(image, ".bss")

format_entry_mut(image, start)
format_finish(image)
```

这样写出来的 `.bss` 文件大小是 0、内存大小是 8。

反过来，在 `.bss` 里写真实字节**不会报错**，但结果自相矛盾：节的属性仍是 `IMAGE_SCN_CNT_UNINITIALIZED_DATA`（声明没有初始化数据），文件大小却不再是 0。把上面那行替换为 `dd(0x11223344)` 与 `dd(0x55667788)`，`.bss` 的文件大小变为 512——按文件对齐补齐后多占一个完整块，而加载器按属性仍然认为这段不需要从文件读取。文件能够装配出来，含义却是错的，因此这类字节应写在 `format_data` 的节里。

重定位不是指针

写指针值和声明重定位是两件事，两件都要做：

```
import("format/format.inc")

let image: map = format_pe64(
  format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_required,
  list.of(
```

```

        format_section(".text", format_code | format_readable | format_executable),
        format_section(".data", format_data | format_readable | format_writeable),
        format_section(".reloc", format_fixups | format_readable | format_discardable)
    )
)
format_begin(image)

format_section_begin(image, ".text")
start:
    ret
worker:
    mov eax, 42
    ret
format_section_end(image, ".text")

// 槽先按最终宽度写出来，defer 里只改值。
format_section_begin(image, ".data")
worker_pointer:
    dq(0)
format_section_end(image, ".data")

let relocs: list = pe_reloc_new()
// 参数是保存地址的槽，不是被指向的 worker。
format_pe_reloc_add_mut(image, relocs, worker_pointer)
format_pe_reloc_section(image, ".reloc", relocs)

format_entry_mut(image, start)
format_finish(image)

// 地址稳定后再回填数值。
defer {
    store.u64(worker_pointer, worker)
}

```

`format_pe_reloc_add_mut` 接收的是**槽的地址**。传入目标标签 `worker` 不会报错，但生成的记录指向 `.text` 中某条指令的内部，加载器更换基址时会去修改一个不属于任何槽的位置；上例若传入 `worker`，记录落在 `0x1001`，正是 `.text` 内部。核对方法是将记录中的 RVA 与节表对照，它必须落在 `.data` 的槽上。

`defer` 中回填数值是允许的，因为数值不改变文件大小；上例的 `store.u64` 就放在 `defer` 里。

入口只属于可执行输出

PE 可执行文件、PE DLL、ELF 可执行文件要设入口：

```
format_entry_mut(image, start)
```

COFF 目标文件、ELF 目标文件和 ELF 共享库都不使用入口这条路。给它们设置入口，报 `format plan does not use an executable entry`；反过来，PE 或 ELF 可执行文件缺少入口时，`format_finish` 报 `PE executable or DLL requires an entry address` 或 `ELF executable requires an entry address`。

表项挂载的时机

只有一类函数必须在 `format_begin` 之前调用：把符号表、重定位表、导入导出元数据挂到配置上的 `format_*_tables_mut`，例如 `format_coff_tables_mut`、`format_elfobj_tables_mut`、`format_elfexe_tables_mut`、`format_elfso_tables_mut`。它们决定头部和动态表需要预留多少空间，因此要在开始输出之前交出数据。

PE 的几个节生成函数不受该限制。`format_pe_import_section`、`format_pe_export_section`、`format_pe_resource_section`、`format_pe_reloc_section` 都是写出节内容，放在 `format_begin` 之后、`format_finish` 之前调用；在 `format_begin` 之后调用时，导入目录同样被正确注册（`RVA = 0x2040`、`size = 40`）。它们唯一的要求是不要自行再套一层 `format_section_begin` 去打开同一个节。

defer 只做最终回填

`defer` 适合回填地址、写指针槽、计算校验和这类不改变布局的操作。往其中写入内容会报错：

```
// 不合格：在 defer 里写节内容。
defer {
    format_section_begin(image, ".text")
    dd(0x11223344)
    format_section_end(image, ".text")
}
```

诊断是 `FinalizerCannotChangeLayout`。参与布局的字节必须在 `format_finish` 之前写完，`defer` 只在最终字节确定之后修改数值。

有些错误由用法决定

上面每条错误都有诊断或可核对的结果，但有几类错误 `format.inc` 不会拦，只能靠写法本身避免：

情况	后果
<code>.bss</code> 里写真实字节	装配成功，节属性与文件内容矛盾，且多占一个文件块
重定位记录指向目标标签	装配成功，记录指向无关位置
漏写 <code>format_section_end</code>	装配成功，节表行没有定型，文件被截短
跨语言调用时不按 ABI 放置参数寄存器	装配和链接都成功，运行结果错误

这四类问题都不会有任何提示，因此修改这几处之后必须**核对产物**：用 `llvm-readobj` 读节表和目录，或在真机运行一遍查看退出码。第 3 章和第 4 章给出了具体的核对命令。

6. macOS Mach-O

macOS 上的可执行程序、共享库和供链接器使用的目标文件都用 Mach-O 格式。它和 PE、ELF 的组织方式都不同：内容按**段（segment）分组，每个段下面再放节（section）**，段和节的名字各占 16 字节，而且名字有固定含义——`__TEXT` 放代码，`__DATA` 放可写数据，加载器和系统工具按这些名字定位内容。

`format.inc` 这一层负责**结构**：头部、段命令、节表、符号表、重定位表，以及它们之间的偏移关系。它不参与指令编码，也不解析 ARM 汇编文本——写进节里的字节由源文件给出。

先选产物

Mach-O 的三类产物用三个不同的构造函数建立，选择依据是「谁来收尾」：

```
import("format/format.inc")

// macOS 14 作为最低版本和 SDK 版本。
const version: u64 = macho_exe_macos_version(14, 0, 0)

// 可执行文件：首段必须是 __TEXT，且必须含一个代码节。
let image: map = format_macho64_exe(
    format_macho64_target_x86_64(version, version),
    list.of(format_macho64_segment(
        "__TEXT",
        format_load | format_readable | format_executable,
        list.of(format_section("__text", format_code | format_readable | format_executable))
    ))
)

format_begin(image)
format_section_begin(image, "__text")
entry:
    // 机器码由你给出：xor eax, eax; ret 的 x86-64 编码。
    db(0x31, 0xc0, 0xc3)
format_section_end(image, "__text")
format_entry_mut(image, entry)
format_finish(image)
```

生成的映像为 4096 字节，头字段为 `Magic64 (0xFEEDFACF)`、`CpuType X86-64`、`FileType Executable`，共 6 条 load command。页大小按 CPU 决定：x86-64 取 4 KiB，arm64 取 16 KiB，和 ELF 那章 AArch64 的取值一致。

公共入口

只导入 `format/format.inc` 就能拿到上面这些构造函数。需要直接写段命令、节表、符号表时，才改用更底层的那几个文件：

产物	底层文件	收尾工具
可重定位目标文件	<code>macho_obj.inc</code>	<code>ld64</code> 或 <code>ld64.lld</code>
可执行文件	<code>macho_exe.inc</code>	macOS 加载器；签名由外部工具完成
共享库	<code>macho_dylib.inc</code>	dyld；导入与签名由外部工具完成
dyld 函数导入	<code>macho_import.inc</code>	用于直接生成的可执行文件

三种产物

`format_macho64_object`、`format_macho64_exe`、`format_macho64_dylib` 共用同一套节生命周期，差别只在头部和段命令。

段和节的名字都是 ≤ 16 字节，而且**段名必须是 `__TEXT`、`__DATA` 这类系统认识的名字**（两个下划线开头）。`__PAGEZERO` 和 `__LINKEDIT` 由 `format.inc` 自己生成，不要声明。

节的身份是「段名 + 节名」，因为不同段可以放同名节：

```
// 两个段共用一个节名时，两个名字都要给出。
format_macho64_section_begin(image, "__DATA", "__data")

// 关闭时同样给出两个名字。
format_macho64_section_end(image, "__DATA", "__data")
```

只有一个段时，`format_section_begin` 和 `format_section_end` 只收节名就够了，和 PE、ELF 的用法一致。

可执行文件和共享库的约束比目标文件多，违反时报错：

约束	诊断
首段必须是 <code>__TEXT</code>	<code>Mach-O first segment must be __TEXT</code>
至少有一个代码节	<code>Mach-O executable or dylib requires a code section</code>
代码节必须落在可执行段里	<code>Mach-O code section requires an executable segment</code>
节的权限不能超过所在段	<code>Mach-O section permissions exceed its segment permissions</code>
零填充节必须排在段内最后	<code>Mach-O zerofill sections must be last in their segment</code>

约束	诊断
节名、段名不能重复	Mach-O section name is duplicated / Mach-O segment name is duplicated
可执行文件的入口必须落在代码节内	Mach-O executable entry is outside its code sections
共享库至少要有有一个导出	Mach-O dylib facade requires at least one export
最后一个节必须已经关闭	Mach-O final section is still open

可执行文件的入口落在数据节里会被拒绝，因此 `format_entry_mut` 要传代码节内的标签。共享库的导出不是可选项——`format_macho64_dylib` 要求至少声明一个导出，否则 `format_finish` 直接报错。

节名可以用到 16 字节，比 PE 的 8 字节宽；但节的用途只支持三种：`format_code`、`format_data`、`format_uninitialized_data`。Mach-O 没有「可丢弃」这个概念，`format_discardable` 会报 `Mach-O sections do not support the discardable attribute`。

目标文件符号与重定位

目标文件里供链接器使用的两项内容是符号表和重定位表。`format_macho64_object` 在写头部时预留 `LC_SYMTAB` 命令，`format_finish` 回填其中的偏移，因此不用自己计算符号下标或字符串表偏移：

```
import("format/format.inc")

const version: u64 = macho_exe_macos_version(14, 0, 0)
let object: map = format_macho64_object(
    format_macho64_target_arm64(version, version),
    list.of(
        format_section("__text", format_code | format_readable | format_executable),
        format_section("__data", format_data | format_readable | format_writeable)
    )
)

format_begin(object)

// 三处占位指令字，立即数字段留给链接器。
format_section_begin(object, "__text")
text_start:
call_site:
    emit.u32(0x94000000)
page_site:
    emit.u32(0x90000000)
lo12_site:
    emit.u32(0x91000000)
format_section_end(object, "__text")

format_section_begin(object, "__data")
```

```
data_start:
    answer:
        emit.u64(42)
pointer_slot:
    emit.u64(0)
format_section_end(object, "__data")

const symbols: list = list.of(
    format_macho64_public("__entry", "__text", text_start, call_site, macho_n_sect |
macho_n_ext),
    format_macho64_public("_answer", "__data", data_start, answer, macho_n_sect |
macho_n_ext),
    format_macho64_extern("_printf")
)
const relocs: list = list.of(
    format_macho64_arm64_reloc("__text", text_start, call_site, "_printf",
macho_arm64_reloc_branch26),
    format_macho64_arm64_reloc("__text", text_start, page_site, "_answer",
macho_arm64_reloc_page21),
    format_macho64_arm64_reloc("__text", text_start, lo12_site, "_answer",
macho_arm64_reloc_pageoff12),
    format_macho64_arm64_reloc("__data", data_start, pointer_slot, "_printf",
macho_arm64_reloc_unsigned)
)
format_macho64_tables_mut(object, symbols, relocs)
format_finish(object)
```

生成的映像为 448 字节，FileType 为 Relocatable，带 MH_SUBSECTIONS_VIA_SYMBOLS 标志。符号表里有三个符号：_entry 落在 __text、_answer 落在 __data，两者类型都是 Section；_printf 类型是 Undef，落在段 0，等待链接器解析。

符号和重定位的对应关系：

项目	取值方式
符号值	地址减去所在节的起始地址；节号是 1 起算的 n_sect
已定义的全局符号	类型写成 macho_n_sect macho_n_ext
未定义的外部符号	由 format_macho64_extern 生成，类型是 macho_n_undef macho_n_ext，值为 0
符号名	写进字符串表，n_strx 由 format.inc 计算
重定位地址	同样是节内偏移，对应 Mach-O 的 r_address
重定位目标	按名字在挂载的符号表里查，r_extern 恒为 1
重定位的 pcrel 与长度	format_macho64_arm64_reloc 从 AArch64 类型推出；用 format_macho64_reloc 时要自己给出

format_macho64_arm64_reloc 推出的两个位域，按 AArch64 重定位类型固定取值：

重定位类型	pcrel	length
macho_arm64_reloc_branch26	1	2 (4 字节)
macho_arm64_reloc_page21	1	2 (4 字节)
macho_arm64_reloc_pageoff12	0	2 (4 字节)
macho_arm64_reloc_unsigned	0	3 (8 字节)

pcrel 为 1 表示这个字段是相对寻址，加载器不需要修正；为 0 表示字段里是绝对地址，必须修正。长度是幂次：2 表示 4 字节，3 表示 8 字节。

format_macho64_tables_mut 会检查两张表：符号名必须唯一，重定位引用的节和符号必须已经声明。不挂表的目标文件也合法，只是符号表为空。

能力矩阵

能力	arm64	x86_64	所属层
Mach-O 64 头部 / 段 / 节	支持	支持	format.inc
目标文件的符号、重定位、LC_SYMTAB	支持	支持	format.inc
直接生成 MH_EXECUTE	支持	支持	format.inc
LC_LOAD_DYLINKER	支持	支持	可执行文件的底层构造函数
MH_DYLIB install name	支持	支持	format.inc
普通函数 export trie	支持	支持	macho_export.inc
目标文件重定位	BRANCH26、PAGE21、PAGEOFF12	BRANCH、UNSIGNED	ISA 专用包装函数
直接可执行文件的函数导入	支持	支持	macho_import.inc

在直接可执行文件中导入函数

直接生成的可执行文件要让 dyld 在加载时解析外部函数，就得自己写绑定信息。macho_import.inc 提供这条路径，生成的是传统的 non-lazy 绑定，不是 chained fixups：

```
import("format/macho_import.inc")

let imports: list = macho_import_new()
imports = macho_import_use64(imports, "@executable_path/libmath.dylib", "_add7")
```

```
// 各段的地址和文件偏移由外部布局给出，这里只按顺序写出各段信息。
macho_import_emit_stubs_arm64(imports, stubs_vaddr, slots_vaddr)
macho_import_emit_slots64(imports)
macho_import_emit_bind64(imports, data_segment_index)
macho_import_emit_symbols64(imports)
macho_import_emit_indirect64(imports)
macho_import_emit_strings64(imports)
```

导入槽的默认标签是 `<symbol>_stub` 和 `<symbol>_got`，需要别的名字时用

`macho_import_use64_as`。一个导入列表可以放多个依赖库，库序号按首次出现的顺序分配（1 到 15），同时写进绑定流和未定义符号描述。每个依赖库的 `LC_LOAD_DYLIB` 只用 `macho_import_emit_load_dylibs64` 写一次。

这一层不实现 lazy binding、stub helper、chained fixups，也不是通用链接器，够小型命令行工具和 FFI 调用方使用。导入型可执行文件含有未定义符号，因此不能设置 `MH_NOUNDEFS`。完整布局见 `tests/format/macho64_arm64_exe_import.asm` 和 `tests/format/macho64_x86_64_exe_import.asm`；共享库自己导入符号仍要走链接器。

导出 FFI 函数

导出和 PE 一样分两步：先收集，目标标签完成布局后再生成 export trie。

```
import("format/macho_dylib.inc")

let exports: list = macho_export_new()
exports = macho_export_use64(exports, "add7", "_add7")

// 这里写出 Mach-O 头部、__TEXT 段和 load command，并定义 add7。
add7:
    emit.u32(0xd28000e0)
    emit.u32(0xd65f03c0)

let export_size: u64 = macho_export_trie_size64(exports, 0)
exports_data:
    macho_export_trie_emit64(exports, 0)

defer {
    // 把 export_size 写进已经写出的
    // LC_DYLD_EXPORTS_TRIE.datasize 和 __LINKEDIT.filesize 字段。
}
```

这段代码是布局示意：命令偏移由源文件负责，`defer` 之前必须先把固定宽度字段写出来。当前只支持普通的已定义导出，re-export、weak 定义、stub/resolver、chained fixups 和代码签名不在此列。

坐标与延迟字段

自己排布 Mach-O 时要分清两套坐标：

- 标签和 `here()` 是**逻辑地址**，用于 `vmaddr` 和符号值；
- `segment_64`、`section_64` 以及 `dyld` 数据命令里的文件位置是**文件偏移（FOA）**。

最终的文件偏移来自显式布局计算，或者由 `region_file_offset(label)` 给出。`store.u32`、`store.u64` 的目标是逻辑地址，不是原始文件偏移。某个头部字段依赖后面的区域时，先把该字段按最终宽度写出来，在普通源码或 `late_layout` 中生成该区域，最后在 `defer` 里回填数值。`defer` 不能创建字节、标签、区域或对齐。

与链接器的边界

由多个目标文件组成的程序用目标文件形式，交由 `ld64` 或 `ld64.lld` 链接。直接生成的可执行文件和共享库适合小型镜像，但不替代 Apple 的链接器和代码签名。

这一层不提供 universal binary、arm64e、chained fixups、lazy binding、每张表超过 15 个导入库、UUID 和代码签名，也不保证在非 macOS 主机上运行。`dyld` 在加载时解析声明的依赖和符号；Apple Silicon 的可执行文件通常还需要外部的 ad-hoc 或正式签名。

验证方法

在没有 macOS 主机的环境里，用 LLVM 工具核对结构：

```
llvm-readobj --file-headers --sections --symbols --relocs file.o
llvm-objdump -d --macho file.o
llvm-objdump --macho --private-headers --exports-trie file.dylib
llvm-objdump --macho --private-headers --bind --indirect-symbols file
ld64.lld -arch arm64 -platform_version macos 14.0 14.0 ...
radare2 -q -n -a arm -b 64 -c "pd 4 @ <text-address>" file
```

`llvm-objdump -d` 最有价值：它按重定位和符号表把指令反汇编出来，是唯一能同时验证机器码、符号值和重定位记录三者对得上的一步。上面那份目标文件反汇编后得到：

```
_entry:
    0:  bl      _answer
    4:  ret
_answer:
    8:  mov     w0, #0x2a
    c:  ret
```

`b1 _answer` 说明相对分支的重定位被正确解析，`mov w0, #0x2a` 说明写进节里的指令字是有效的 A64 编码。如果重定位位域或符号下标写错，这一步会显示错误的符号名或直接拒绝反汇编。

`llvm-readobj --relocs` 用来核对重定位记录本身，会列出每条记录所在节、偏移、`pcrel`、`length`、`extern` 和类型。

这些命令能核对 Mach-O 结构、重定位记录、导出名称和链接器互操作性。真正运行 arm64 镜像、测试 `dlopen` 或 `dlsym` 仍然需要 Apple Silicon 的 macOS 主机或对应的 CI runner。