



格式教程

XIRASM 格式教程

用普通格式接口构建 PE、COFF 与 ELF 文件

目录

1. XIRASM 格式教程
2. 选择模板
3. Windows PE 和 DLL
4. Linux ELF 可执行文件和共享库
5. COFF 和 ELF 目标文件
6. 通用规则和常见错误

XIRASM 格式教程

这份教程介绍 XIRASM 的常用格式接口。你只需要先确定要生成哪种文件，再按模板声明节、段、导入、导出和重定位。PE、COFF、ELF 的字段计算由格式封装完成。

普通程序统一从这里开始：

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

`include/format/` 提供三层接口：

- `format.inc` 是普通用户入口，负责格式配置、命名节或段以及常用表。
- `pe32.inc`、`pe64.inc`、`elf32.inc`、`elf64.inc` 是兼容用的位宽包装。
- `pe.inc`、`elfexe.inc`、`elfobj.inc` 等提供直接控制，适合标准模板无法表达的文件。

本教程完整说明 `format.inc`。需要直接控制字段和表项时，再阅读[高级格式构造指南](#)。

章节

1. 选择模板
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

快速选择

输出文件	构造函数	主要接口
Windows 可执行文件	<code>format_pe32</code> 或 <code>format_pe64</code>	<code>format_section_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Windows DLL	<code>format_pe32</code> 或 <code>format_pe64</code> ，选用 <code>format_pe_dll</code>	<code>format_pe_export_pairs_mut</code> 、 <code>format_pe_export_section</code>
Linux 可执行文件	<code>format_elf32</code> 或 <code>format_elf64</code> ，选用 <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> ，选用 <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>
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	<code>format_elfobj32</code> 或 <code>format_elfobj64</code>	<code>format_elfobj_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>

基本流程

所有普通格式都遵循同一顺序：

1. 用构造函数创建 `let` 格式配置。
2. 按需用 `*_mut` 接口添加导入、导出、符号或重定位。
3. 调用 `format_begin`。
4. 在已声明的节或段中写入代码和数据。
5. 可执行文件用 `format_entry_mut` 设置入口。
6. 调用 `format_finish`。

构造函数返回新值，例如 `format_pe64(...)` 和 `format_section(...)`。名称以 `_mut` 结尾的过程会直接修改传入的 `let` 绑定：

```
let image: map = format_pe64(options, sections)
format_entry_mut(image, start)
format_finish(image);
```

不要把 `const`、临时表达式或字段访问传给 `_mut` 参数。

1. 选择模板

先确定输出文件类型，再选择构造函数。普通用户不需要从文件头字段开始设计。

统一入口

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

只有在标准模板无法表达目标文件时，才直接导入 `pe.inc`、`elfexe.inc` 或 `elfobj.inc`。

构造函数

输出文件	构造函数	内容描述
PE32 可执行文件或 DLL	<code>format_pe32(options, sections)</code>	<code>format_section(...)</code>
PE64 可执行文件或 DLL	<code>format_pe64(options, sections)</code>	<code>format_section(...)</code>
COFF32 目标文件	<code>format_coff32(sections)</code>	<code>format_section(...)</code>
COFF64 目标文件	<code>format_coff64(sections)</code>	<code>format_section(...)</code>
ELF32 可执行文件	<code>format_elf32(options, segments)</code>	<code>format_segment(...)</code>
ELF64 可执行文件或 PIE	<code>format_elf64(options, segments)</code>	<code>format_segment(...)</code>
ELF32 目标文件	<code>format_elfobj32(sections)</code>	<code>format_section(...)</code>
ELF64 目标文件	<code>format_elfobj64(sections)</code>	<code>format_section(...)</code>
ELF64 共享库	<code>format_elf64_so(soname, segments)</code>	<code>format_segment(...)</code>

PE、COFF 和 ELF 目标文件使用节。ELF 可执行文件和共享库使用加载段。

节属性

`format_section(name, attributes)` 接收节名和属性。属性必须包含一个用途，可以再组合权限。

用途	内容
<code>format_code</code>	指令
<code>format_data</code>	已初始化数据或只读数据
<code>format_uninitialized_data</code>	只占内存的未初始化空间
<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	含义
<code>format_readable</code>	可读
<code>format_writeable</code>	可写
<code>format_executable</code>	可执行
<code>format_discardable</code>	加载器处理后可以丢弃

常用组合：

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

段属性

`format_segment(name, attributes)` 用于 ELF 加载段。属性使用 `format_load` 加权限：

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

完整流程

```
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),
    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");

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

格式封装会根据命名节推导头部、表项、文件位置、RVA、对齐和 BSS 大小。

2. Windows PE 和 DLL

PE 配置需要指定文件角色、子系统、内存保护策略、地址随机化策略和节列表。

PE 选项

`format_pe32(options, sections)` 和 `format_pe64(options, sections)` 必须各选一个文件角色、子系统和 ASLR 策略。`format_pe_nx` 可以按需加入。

分组	选项	作用
文件角色	<code>format_pe_exe</code>	可执行文件
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>	图形界面程序
内存保护	<code>format_pe_nx</code>	标记镜像兼容不可执行数据页
ASLR	<code>format_pe_aslr_auto</code>	存在重定位数据时启用 ASLR
ASLR	<code>format_pe_aslr_required</code>	必须提供重定位数据并启用 ASLR
ASLR	<code>format_pe_aslr_disabled</code>	禁用 ASLR

常用节：

节名	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>

最小 PE64 可执行文件

```
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),
    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");

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

导入 Windows API

先创建导入集合，再按 DLL 成组添加 API。`format_pe_import_pairs_mut` 的 `pairs` 参数按“本地槽名、导入名”重复排列。

```

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),
        format_section(".idata", format_imports | format_readable | format_writeable)
    )
)

let imports: map = format_pe_import_new()
// 同名批量接口一次加入同一个 DLL 的多个 API。
format_pe_import_many_mut(
    image,
    imports,
    "KERNEL32.DLL",
    list.of("ExitProcess", "GetCurrentProcessId")
)
// 再次调用即可加入另一个 DLL; pairs 还能指定本地槽名。
format_pe_import_pairs_mut(
    image,
    imports,
    "ADVAPI32.DLL",
    list.of("close_registry_key", "RegCloseKey")
)
format_begin(image);

format_section_begin(image, ".text");
start:
    sub rsp, 40
    call [rel GetCurrentProcessId]
    xor ecx, ecx
    call [rel ExitProcess]
format_section_end(image, ".text");

format_pe_import_section(image, ".idata", imports);
format_entry_mut(image, start)
format_finish(image);

```

API	参数	作用
<code>format_pe_import_new()</code>	无	创建空导入集合
<code>format_pe_import_many_mut(plan, imports, dll, names)</code>	配置、导入集合、DLL、名称列表	API 名与本地槽名相同
<code>format_pe_import_pairs_mut(plan, imports, dll, pairs)</code>	配置、导入集合、DLL、槽名/导入名列表	自定义本地槽名
<code>format_pe_import_section(plan, name, imports)</code>	配置、导入节名、导入集合	生成 <code>.idata</code>

每个 DLL 调用一次批量接口，并持续更新同一个 `imports`，最终生成的 `.idata` 就会包含全部 DLL 和 API。PE64 通过 `call [rel slot_name]` 调用导入函数；PE32 使用 `call [slot_name]`。

导出 DLL 函数

`format_pe_export_many_mut` 批量导出同名标签；`format_pe_export_pairs_mut` 的 `pairs` 参数按“目标标签、公开名称”重复排列。

```

import("format/format.inc");

let image: map = format_pe64(
  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()
// 两个标签直接使用原名导出。
format_pe_export_many_mut(image, exports, list.of("x_add7", "x_sub3"))
// answer_impl 对外使用 x_answer 这个名称。
format_pe_export_pairs_mut(image, exports, list.of("answer_impl", "x_answer"))
format_begin(image);

format_section_begin(image, ".text");
dll_main:
  // 最小 DLL 入口返回 TRUE。
  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");

format_pe_export_section(image, ".edata", exports, "xirasm_demo.dll");
format_entry_mut(image, dll_main)
format_finish(image);

```

API	作用
<code>format_pe_export_new()</code>	创建空导出集合
<code>format_pe_export_many_mut(plan, exports, names)</code>	批量导出同名标签
<code>format_pe_export_pairs_mut(plan, exports, pairs)</code>	批量映射目标标签和公开名称
<code>format_pe_export_section(plan, name, exports, dll_name)</code>	生成 <code>.edata</code>

资源

先把 `.rsrc` 声明为 `format_resources`，再把编译好的 `.res` 文件交给封装：

```
format_pe_resource_section(image, ".rsrc", "data/app.res");
```

API	参数	作用
<code>format_pe_resource_section(plan, name, path)</code>	配置、已声明的资源节名、 <code>.res</code> 路径	写入编译后的资源树并登记 PE 资源目录

基址重定位

写入绝对地址和声明基址重定位是两件事。下面是可直接汇编的 PE64 模板：

```

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(".idata", format_imports | format_readable | format_writeable),
        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:
    // 为两次 Windows x64 调用预留 shadow space 并对齐栈。
    sub rsp, 40
    // 指令通过相对地址访问指针槽。
    mov rax, [rel worker_pointer]
    call rax
    // 用 ExitProcess 明确结束示例程序。
    mov ecx, eax
    call [rel ExitProcess]
worker:
    mov eax, 42
    ret
format_section_end(image, ".text");

format_section_begin(image, ".data");
worker_pointer:
    // 文件中保存的是绝对指针，因此这个槽需要基址重定位。
    dq(0);
format_section_end(image, ".data");

format_pe_import_section(image, ".idata", imports);

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);

// 标签地址稳定后再写入最终绝对值。
defer {
    store.u64(worker_pointer, worker);
}

```

`format_pe_reloc_add_mut` 的地址参数是“保存指针的槽”，不是指针指向的目标。它会根据 PE32 或 PE64 选择重定位类型，记录必须按 RVA 升序传给 `format_pe_reloc_section`。`rel` 指令引用本身是相对位移，不需要基址重定位。PE32 对应使用 `dd(0)` 和 `store.u32`。

API	参数	作用
<code>pe_reloc_new()</code>	无	创建空重定位列表
<code>format_pe_reloc_add_mut(plan, relocs, storage)</code>	配置、列表、指针存储地址	添加与 PE 位宽匹配的基址重定位
<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 <code>.reloc</code> 并登记数据目录

校验和

校验和依赖最终文件内容，因此在 `format_finish` 之后调用：

```
format_pe_checksum(image);
```

`format_pe_checksum(plan)` 接收已经完成的 PE 配置，并根据最终文件内容回填校验和字段。

3. Linux ELF 可执行文件和共享库

普通接口用命名加载段描述 ELF 镜像。文件位置、虚拟地址、文件大小、内存大小和权限由格式配置推导。

ELF 可执行文件

固定地址可执行文件使用 `format_elf_exec` :

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

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

format_segment_begin(image, ".text");
start:
    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);
```

ELF32 使用 `format_elf32(format_elf_exec, segments)`。

ELF64 PIE

位置无关可执行文件使用 `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:
    // 加载器改变 PIE 基址后, 这两个标签引用仍然有效。
    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);
```

文件格式是位置无关的, 指令也必须遵守目标 ISA 的位置无关规则。x86-64 访问本镜像内标签时使用 `rel`, 它编码相对位移, 不需要绝对动态重定位。PIE 或共享库中保存的绝对指针需要动态重定位; 普通接口不提供任意用户指针重定位, 这类需求属于直接 ELF 层。普通接口暂不提供 ELF32 PIE。

ELF64 可执行文件导入

当前普通接口为固定地址 ELF64 可执行文件生成 PLT/GOT 动态导入。PIE 不使用这条导入流程。

```

import("format/format.inc");

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

let imports: list = format_elfexe_import_new()
// 同一个库一次加入多个 API。
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("getpid", "getppid"))
// 再加入另一个库，并把 cos 的本地前缀改成 cos_fn。
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:
    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);

```

每个库调用一次批量接口，并持续更新同一个 `imports`。`format_elfexe_import_many_mut` 会为同名导入生成 `<name>_gotplt` 和 `<name>_plt`；别名导入 `cos_fn` 对应 `cos_fn_gotplt` 和 `cos_fn_plt`。

ELF64 共享库

共享库没有普通可执行入口。它通过动态符号表导出符号，也可以声明导入。

```

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()
// 两个四字节函数直接使用标签名导出。
format_elfso_export_many_mut(exports, list.of("x_add7", "x_sub3"), ".text", 4)
// answer_impl 对外使用 x_answer。
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);

```

ELF64 共享库导入

共享库也可以把多个库的导入收集到同一个列表：

```

import("format/format.inc");

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

let exports: list = format_elfso_export_new()
format_elfso_export_pairs_mut(exports, list.of("report_impl", "x_report"), ".text", 18)

let imports: list = format_elfso_import_new()
// libc 中的两个 API 使用同名本地标签。
format_elfso_import_many_mut(imports, "libc.so.6", list.of("puts", "getpid"))
// 第二个库把 cos 映射到本地前缀 cos_fn。
format_elfso_import_pairs_mut(imports, "libm.so.6", list.of("cos_fn", "cos"))
format_elfso_tables_mut(image, exports, imports)

format_begin(image);

format_segment_begin(image, ".text");
report_impl:
    lea rdi, [rel message]
    call puts_plt
    call getpid_plt
    ret
format_segment_end(image, ".text");

format_segment_begin(image, ".data");
message:
    db("XIRASM shared object", 0);
format_segment_end(image, ".data");

format_finish(image);

```

即使示例没有调用 `cos_fn_plt`，该标签和 `cos_fn_gotplt` 仍会生成。本镜像内的数据使用 `rel` 引用，外部函数则通过生成的 PLT 标签调用。

API 摘要

API	作用
<code>format_elf32(format_elf_exec, segments)</code>	ELF32 可执行文件
<code>format_elf64(format_elf_exec, segments)</code>	ELF64 固定地址可执行文件
<code>format_elf64(format_elf_pie, segments)</code>	ELF64 PIE
<code>format_elf64_so(soname, segments)</code>	ELF64 共享库
<code>format_elfexe_import_new()</code>	创建可执行文件导入集合
<code>format_elfexe_import_many_mut(imports, library, names)</code>	批量添加同名 PLT/GOT 导入
<code>format_elfexe_import_pairs_mut(imports, library, pairs)</code>	批量映射本地名称和导入名
<code>format_elfexe_tables_mut(plan, imports)</code>	把导入集合附加到可执行文件配置
<code>format_elfso_export_new()</code>	创建共享库导出集合
<code>format_elfso_export_many_mut(exports, names, segment, size)</code>	批量导出同名标签
<code>format_elfso_export_pairs_mut(exports, pairs, segment, size)</code>	批量映射目标标签和公开名称
<code>format_elfso_import_new()</code>	创建共享库导入集合
<code>format_elfso_import_many_mut(imports, library, names)</code>	批量添加共享库导入
<code>format_elfso_import_pairs_mut(imports, library, pairs)</code>	批量映射本地名称和导入名
<code>format_elfso_tables_mut(plan, exports, imports)</code>	附加共享库动态符号信息

4. COFF 和 ELF 目标文件

目标文件交给链接器继续处理。普通接口需要三类信息：节、链接器可见符号，以及需要链接器修补的位置。

COFF 目标文件

```

import("format/format.inc");

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);

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");

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)
)

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);

```

32 位相对调用使用 `coff_rel_i386_rel32`，64 位使用 `coff_rel_amd64_rel32`。

ELF 目标文件

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

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");

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)
)

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);
```

ELF32 使用 `format_elfobj32`。32 位相对调用常用 `elf_r_386_pc32`。

API 摘要

格式	API	作用
COFF	<code>format_coff32(sections)</code> / <code>format_coff64(sections)</code>	创建目标文件配置
COFF	<code>format_coff_public(...)</code>	定义公开符号
COFF	<code>format_coff_extern(...)</code>	声明外部符号
COFF	<code>format_coff_reloc(...)</code>	描述重定位字段
COFF	<code>format_coff_tables_mut(plan, symbols, relocs)</code>	附加符号和重定位集合
ELF	<code>format_elfobj32(sections)</code> / <code>format_elfobj64(sections)</code>	创建目标文件配置
ELF	<code>format_elfobj_public(...)</code>	定义公开符号及大小
ELF	<code>format_elfobj_extern(...)</code>	声明外部符号
ELF	<code>format_elfobj_reloc(...)</code>	描述带 addend 的重定位字段
ELF	<code>format_elfobj_tables_mut(plan, symbols, relocs)</code>	附加符号和重定位集合

先在代码中写入占位字段，再用 `format*_reloc` 描述它。链接器会根据重定位记录修补这些字节。

5. 通用规则和常见错误

只使用一层接口

普通程序统一导入：

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

不要同时使用普通格式配置和手写底层表项。需要完全控制格式时，改用高级格式构造指南中的 direct 层。

配置必须使用 `let`

构造函数返回配置； `_mut` 过程直接修改配置或集合：

```
let image: map = format_elf64(format_elf_exec, segments)
let imports: list = format_elfexe_import_new()
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("getpid"))
format_elfexe_tables_mut(image, imports)
format_entry_mut(image, start)
```

传给 `_mut` 的目标必须是直接 `let` 绑定，不能是 `const` 或临时表达式。

先声明，再写内容

传给 `format_section_begin`、`format_section_end`、`format_segment_begin` 和 `format_segment_end` 的名字必须已经在配置中声明。节名和段名不能重复。

每个描述只选一个用途

```
format_section(".text", format_code | format_readable | format_executable)
```

`format_code` 和 `format_data` 不能同时用于同一个节。用途只选一个，权限可以组合。

权限按运行需求设置

内容	常用权限
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 重定位表	可读、可丢弃

除非确实需要自修改代码，否则不要让代码可写；数据也不应具有执行权限。

BSS 不写入初始化文件内容

在 `format_uninitialized_data` 中使用 `rb(...)` 或 `reserve(...)`。它们增加内存大小，但不会把同样数量的零字节写进文件。

重定位和地址值是两件事

```
absolute_slot:
    dq(0);

let relocs: list = pe_reloc_new()
format_pe_reloc_add_mut(image, relocs, absolute_slot)
format_pe_reloc_section(image, ".reloc", relocs);

defer {
    store.u64(absolute_slot, start);
}
```

`store.u64` 写入地址值；重定位记录告诉加载器这个位置需要随镜像基址调整。

入口只属于可执行文件

PE 和 ELF 可执行文件使用 `format_entry_mut(plan, label)`。COFF、ELF 目标文件和 ELF 共享库不使用这条入口流程。

defer 只做最终回填

`defer` 可以检查最终字节、回填地址和写校验和，但不能生成会改变布局的新内容。需要参与布局的字节必须在 `format_finish` 前写入正常节或段。

从最小模板逐步增加功能

1. 先完成只有 `.text` 的文件。
2. 再加入 `.data` 或 `.bss`。
3. 需要系统函数时加入导入。
4. 构建 DLL 或共享库时加入导出。
5. 文件保存绝对地址时加入基址重定位。
6. 需要链接器继续处理时选择 COFF 或 ELF 目标文件模板。