

格式指南

XIRASM 可执行文件格式指南

PE、COFF 与 ELF 的常用构建流程

目录

1. XIRASM 可执行文件格式指南
2. 第 1 章：认识 XIRASM 可执行文件格式
3. 第 2 章：格式方案与构建流程
4. 第 3 章：节、段、权限与未初始化数据区
5. 第 4 章：地址、符号与重定位
6. 第 5 章：PE32 与 PE64 可执行文件
7. 第 6 章：导入 Windows 接口函数
8. 第 7 章：`DLL` 导出与基址重定位
9. 第 8 章：PE 资源与校验和
10. 第 9 章：COFF32 与 COFF64 目标文件
11. 第 10 章：ELF32 与 ELF64 可执行文件
12. 第 11 章：位置无关可执行文件与动态导入
13. 第 12 章：ELF32 与 ELF64 目标文件
14. 第 13 章：ELF64 共享对象

XIRASM 可执行文件格式指南

处理器指令序列只是可执行文件的一部分。操作系统加载器和本机链接器还需要按照特定文件格式组织文件头、节或段、访问权限、符号、导入、导出和重定位信息。

XIRASM 提供常用格式接口，让普通汇编源代码可以直接构造这些文件：

```
// 导入面向普通程序的常用格式接口。  
import("format/format.inc");
```

本指南通过完整的 PE、COFF 和 ELF 构建流程介绍这些接口。内容先讲解所有格式共用的模型，再分别说明各个格式系列。更底层、面向特殊格式实现的辅助接口不属于本指南，将在高级格式指南中单独介绍。

如果需要了解值、函数、标号、数据写出、输出区域或收尾处理的一般规则，请先阅读[语言指南](#)。如果希望由 XIRASM 生成可以直接构建的初始项目，请参见第 1 章的[从命令行开始](#)一节。

指南结构

第一部分：格式基础

1. [认识 XIRASM 可执行文件格式](#) - 常用格式接口、第一个可执行文件，以及统一的格式构建流程。
2. [格式方案与构建流程](#) - 声明文件映像、写入命名内容、绑定入口并完成格式。
3. [节、段、权限与未初始化数据区](#) - 运行时映射、文件中有实际字节的数据、预留内存和对齐。
4. [地址、符号与重定位](#) - 虚拟地址、相对虚拟地址、文件偏移、导入、导出和重定位信息。

第二部分：Windows 格式

5. [PE32 与 PE64 可执行文件](#) - 控制台程序、多节布局、数据、未初始化数据区和入口地址。
6. [导入 Windows 接口函数](#) - 导入声明、自动生成的数据表和外部函数调用。
7. [动态链接库导出与基址重定位](#) - 导出函数和数据、重定位声明与地址随机化策略。
8. [PE 资源与校验和](#) - 资源数据、资源文件和可选的映像校验和。
9. [COFF32 与 COFF64 目标文件](#) - 节、公开符号、外部符号、重定位和本机链接。

第三部分：ELF 格式

10. [ELF32 与 ELF64 可执行文件](#)
 - 可加载段、紧凑的文件布局、数据、未初始化数据区和入口地址。
11. [位置无关可执行文件与动态导入](#)
 - 位置无关映像、动态元数据、过程链接表调用和外部函数。

12. ELF32 与 ELF64 目标文件

- 节、符号、`REL` 与 `RELA` 两种重定位记录形式，以及本机链接。

13. ELF64 共享对象

- 导出符号、导入符号、加载方式和其他语言调用者。

本指南只介绍常用格式接口。直接构造格式数据表和使用格式专用的底层辅助接口，属于独立的高级格式指南，以免普通构建流程被实现细节淹没。

第 1 章：认识 XIRASM 可执行文件格式

从处理器指令到可加载文件

自然书写的处理器指令描述处理器需要执行的操作：

```
// 选择 64 位 x86 指令模式，并定义一段返回零的代码。  
x86.use64();  
  
start:  
    xor eax, eax  
    ret
```

直接汇编时，这些指令只会形成连续的字节序列。加载器无法仅凭指令判断它们属于哪一种可执行文件格式，还需要知道：

- 哪些字节是可以执行的代码；
- 哪些内存范围允许读取或写入；
- 程序从哪里开始执行；
- 哪些数据实际占用文件空间，哪些空间只在加载后的内存中存在；
- 哪些符号由其他库提供；
- 文件映像加载到不同地址时，哪些地址需要调整。

标准文件格式通过文件头和数据表记录这些信息。格式接口让源代码描述预期的文件映像，再由 XIRASM 根据最终布局推导常规的表项、位置和计数。

常用格式接口

普通程序从导入一个文件开始：

```
// 导入 PE、COFF 和 ELF 的常用格式构造接口。  
import("format/format.inc");
```

常用格式接口提供以下构造函数：

输出类型	常用构造函数
PE32 与 PE64 映像	<code>format_pe32</code> 、 <code>format_pe64</code>
COFF32 与 COFF64 目标文件	<code>format_coff32</code> 、 <code>format_coff64</code>
ELF32 与 ELF64 可执行文件	<code>format_elf32</code> 、 <code>format_elf64</code>
ELF32 与 ELF64 目标文件	<code>format_elfobj32</code> 、 <code>format_elfobj64</code>
ELF64 共享对象	<code>format_elf64_so</code>

构造函数会创建一份格式方案。格式方案记录所选文件系列、映像选项，以及源代码准备写入的完整节或段列表。

源代码随后遵循统一的构建流程：

```

创建格式方案
开始构造格式
写入每个已经声明的节或段
附加入口、符号、导入、导出或重定位信息
完成格式

```

可执行文件、目标文件和共享库使用的具体声明并不完全相同，但原则一致：源代码提供具有实际含义的信息，格式接口根据这些信息推导数据表的位置和数量。

第一个 PE64 可执行文件

下面的源代码创建一个 64 位 Windows 控制台程序，其中只有一个代码节：

```
// 导入常用格式接口，并选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 声明 PE64 控制台程序及其唯一的可执行代码节。
const image0: 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_begin(image0);

// 在已经声明的 .text 节中写入程序入口代码。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 绑定入口标号，并根据最终布局完成文件映像。
const image: map = format_entry(image0, start)
format_finish(image);
```

汇编时显式选择 x86-64 目标平台：

```
# 生成 PE64 可执行文件，然后运行它。
xirasm hello.asm --target x86-64 -o hello.exe
.\hello.exe
```

生成程序的退出状态为 0。

`xor eax, eax` 和 `ret` 等处理器指令行末不写分号。 `format_begin(image0);` 等编译期函数调用需要分号。

按构建流程阅读源代码

上面的示例可以分成六个阶段：

1. `import("format/format.inc")` 选择常用格式接口；
2. `x86.use64()` 选择文件映像中代码使用的指令模式；

3. `format_pe64` 创建 PE64 格式方案，选择可执行文件、控制台子系统、数据页不可执行策略和自动地址空间布局随机化策略；
4. `format_section` 声明一个命名代码节及其访问权限；
5. `format_section_begin` 与 `format_section_end` 包围真正属于 `.text` 节的指令；
6. `format_entry` 绑定入口标号，`format_finish` 完成并检查文件映像。

格式选择和指令模式选择彼此相关，但职责不同。`x86.use64()` 控制指令编码，`format_pe64(...)` 控制包围这些指令的文件结构。

描述项列表必须出现在 `format_begin` 之前，因为它定义了预期的文件映像。列表长度决定需要多少个节表项，排列顺序决定各节在文件中的顺序，名称则把后续内容块连接到对应的表项。

源代码负责提供什么

常用格式接口会让用户必须作出的决定清楚地保留在源代码中：

源代码提供的内容	示例
文件系列与映像类别	PE64 可执行文件
映像策略	控制台、数据页不可执行、自动地址空间布局随机化
节或段的名称	<code>.text</code>
用途与权限	代码、可读、可执行
指令与数据	<code>xor eax, eax</code> 、 <code>ret</code>
入口、符号、导入、导出与重定位信息	<code>start</code>

这些决定会影响程序含义或加载器行为，因此应当在源代码中明确表达。

格式接口负责推导什么

上面的示例没有手工提供：

- 节的数量；
- 节表项索引；
- `.text` 的文件偏移；
- `.text` 的相对虚拟地址；
- 入口字段的文件偏移；
- 手工计算的文件头大小；
- 原始的对齐填充字节。

格式接口会根据格式方案和已经完成的节布局推导这些值。增加一个描述项时，所需的格式结构会随之变化，用户不必另外修改数量或表项编号。

后续章节会把同一规则应用到自动生成的导入、导出、符号和重定位数据。普通源代码只需指出相关函数、标号、节、段、权限和重定位种类，格式接口负责分配索引、数量、偏移和数据表位置。

可执行文件、目标文件与共享库

本指南介绍三类主要输出：

输出用途	加载者或使用者	常见内容
可执行文件映像	操作系统加载器	入口地址、运行时映射、导入
目标文件	本机链接器	节、符号、外部引用、重定位
共享库	加载器与调用程序	导出、导入、运行时元数据

PE 可执行文件和动态链接库使用节。COFF 目标文件也使用节，但它描述的不是完整运行时映像。ELF 可执行文件和共享对象使用可加载段表示运行时映射，ELF 目标文件则使用节组织链接阶段需要的信息。

格式接口会使用所选文件格式本身的概念：

```
format_section_begin(plan, name)
format_section_end(plan, name)

format_segment_begin(plan, name)
format_segment_end(plan, name)
```

不能因为节和段都可以包含代码和数据，就随意把一种构建流程替换成另一种。

常用接口与高级格式工作

当常用格式接口能够描述要生成的文件时，应使用 `format/format.inc`。它支持普通的多节和多段流程，包括可执行文件、动态链接库、目标文件、共享对象、导入、导出和重定位。

格式专用的包含文件会提供范围更窄或层次更低的辅助接口。它们适合实现特殊布局或扩展常用格式接口，但其中一部分要求调用者自行维护通常由常用接口负责的格式约束。

不要随意混用两个层次。仅仅因为程序需要多个节、导入表或重定位表，并不意味着应该放弃自动计数和自动布局。

独立的格式接口参考会明确区分常用接口与高级接口。本指南只关注完整的用户构建流程。

从命令行开始

命令行工具可以生成已经使用常用格式接口的初始项目。

创建 PE64 项目：

```
# 创建并构建 64 位 Windows 项目，然后运行生成的程序。
xirasm init hello-win --isa x86-64 --os windows --abi msvc
cd hello-win
xirasm build
.\build\app.exe
```

创建 ELF64 项目：

```
# 创建并构建采用 System V 应用二进制接口的 64 位 Linux 项目。
xirasm init hello-linux --isa x86-64 --os linux --abi sysv
cd hello-linux
xirasm build
```

生成的源代码采用本章介绍的同一构建流程：创建格式方案、开始格式、写入命名内容、绑定入口并完成格式。

后续章节

第 2 章将详细介绍格式方案，包括描述项顺序、名称、构建状态、入口绑定和自动生成的内容。第 3 章随后区分节、段、文件中有实际字节的数据、未初始化数据区、权限和对齐。

先记住一条简单规则：

```
普通程序 -> import("format/format.inc")
特殊格式实现 -> 有意识地选择更底层的接口
```

[返回可执行文件格式指南](#)

第 2 章：格式方案与构建流程

格式方案是编译期值

常用格式构造函数会返回一个 `map`：

```
const image0: 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
    )
  )
)
```

这个映射是对要生成文件的编译期描述。它不是已经完成的文件字节映像，创建格式方案也不会写入节的内容。

格式方案会记录对应文件格式所需的信息：

方案类别	核心信息
PE 映像	位数、映像选项、节、入口，以及可选的自动生成数据
COFF 目标文件	位数、节、公开或外部符号，以及重定位
ELF 可执行文件	位数、文件模式、可加载段、入口，以及可选的导入
ELF 目标文件	位数、节、公开或外部符号，以及重定位
ELF 共享对象	共享对象名称（ <code>SONAME</code> ）、可加载段、导出和导入

后续辅助函数可能返回一份加入了更多信息的新方案。每次都用新的绑定保存返回值，可以让整个构建流程清晰可见。

描述项定义预定布局

节和段必须在 `format_begin` 之前声明：

```
const sections: list = list.of(
    format_section(
        ".text",
        format_code | format_readable | format_executable
    ),
    format_section(
        ".rdata",
        format_data | format_readable
    )
)
```

每个描述项包含：

- 供后续构建流程调用使用的名称；
- 一种内容用途；
- 一组访问权限；
- 根据这些值推导出的格式专用属性。

描述项列表本身具有结构含义：

列表属性	作用
长度	决定预定的节表项或段表项数量
顺序	决定表项顺序和运行时映射顺序
名称	将后续内容块连接到对应的描述项
用途	选择代码、数据、未初始化数据区（BSS）、导入、导出或其他行为
权限	选择可读、可写、可执行或可丢弃属性

命名内容应当按照描述项的声明顺序写入。PE 和目标文件中的表会按照描述项顺序排列各个表项；ELF 可加载段在分配逻辑映射时也使用这一顺序。

格式方案在创建时验证

如果一组参数无法描述结构一致的文件格式，构造函数会直接拒绝这份方案。

描述项通常遵循以下规则：

- 节或段列表不能为空；
- 名称不能为空，也不能重复；
- 每个描述项必须恰好具有一种用途；
- 互斥选项不能组合使用；
- PE 中用于导入或导出等特殊用途的节不能重复；

- 段属性必须符合 ELF 常用接口的要求。

PE 和当前的 COFF 常用接口要求节名称能够放入八字节的节名称字段。ELF 目标文件通过 `.shstrtab` 保存名称，因此可以使用更长的名称：

```
.text  
.data  
.rodata.long
```

这些检查会在写出文件头或实际内容之前完成。因此，格式错误的方案会在构造时失败，而不会生成一个只有部分结构有效的文件。

格式构建流程的五阶段

使用常用格式接口的源代码遵循五个阶段。

1. 创建格式方案

选择文件格式、选项和完整的描述项列表：

```
const plan0 = format_*(options, descriptors)
```

目标文件和共享库的构造函数可能使用不同的参数，但它们同样会返回一份格式方案。

2. 开始构造格式

开始构造所选文件格式：

```
format_begin(plan0);
```

`format_begin` 会根据方案类别写出或预留该格式所需的初始结构，例如可执行文件头、程序头表项、目标文件头，以及只能在布局确定后补全的字段。

完整的描述项列表准备好之后、开始写入预定内容之前，只调用一次 `format_begin`。

3. 写入命名内容

依次打开每个已经声明的节或段，写入其中的指令或数据，然后将其关闭：

```
format_section_begin(plan0, ".text");
start:
    xor eax, eax
    ret
format_section_end(plan0, ".text");
```

名称必须已经存在于格式方案中。开始调用会把当前输出位置与对应的描述项表项关联起来；结束调用会关闭实际文件范围，并记录该格式所需的最终逻辑大小和文件中有实际字节的数据大小。

ELF 可执行文件和共享对象的可加载段使用 `format_segment_begin` 与 `format_segment_end`。PE、COFF 和 ELF 目标文件使用节相关调用。

4. 附加最终信息

有些信息只有在相应标号或声明出现之后才能取得。可执行文件的入口是最简单的例子：

```
const image: map = format_entry(image0, start)
```

`format_entry` 会返回一份新方案，而不会修改名为 `image0` 的绑定。下一项构建操作必须接收这个返回值。

附加目标文件符号、重定位、可执行文件导入或共享对象数据表的辅助函数，也遵循同样的值传递方式：

```
const plan1 = attach_one_group(plan0, declarations)
const plan2 = attach_another_group(plan1, more_declarations)
format_finish(plan2)
```

这些辅助函数的准确名称将在对应的格式章节和独立的格式接口参考中介绍。

5. 完成格式

完成所选文件：

```
format_finish(image);
```

不同文件格式在完成阶段承担不同工作：

- PE 映像会确定入口，并补全由方案管理的目录信息；
- ELF 可执行文件会确定入口，并补全可选的动态元数据；
- COFF 和 ELF 目标文件会写出符号表、字符串表和重定位表；
- ELF 共享对象会写出动态元数据、符号表、字符串表、导出和导入信息。

`format_finish` 还会拒绝缺少必要构建信息的方案。PE 映像和 ELF 可执行文件要求入口地址非零；目标文件和共享对象会完成各自的数据表，但不会凭空添加可执行入口。

完整的双节格式方案

下面的可执行文件会先声明两个节，再写入其中任何一个节的内容：

```

// 导入常用格式接口，并选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 提前声明代码节和只读数据节，顺序同时决定节表项顺序。
const sections: list = list.of(
    format_section(
        ".text",
        format_code | format_readable | format_executable
    ),
    format_section(
        ".rdata",
        format_data | format_readable
    )
)

// 使用完整的描述项列表创建并开始 PE64 格式方案。
const image0: map = format_pe64(
    format_pe_exe
        | format_pe_console
        | format_pe_nx
        | format_pe_aslr_auto,
    sections
)
format_begin(image0);

// 写入 .text 节，并记录程序入口标号。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 按声明顺序写入 .rdata 节中的零结尾文本。
format_section_begin(image0, ".rdata");
message:
    db("hello", 0);
format_section_end(image0, ".rdata");

// 保存附加入口后的新方案，再完成整个文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

```

格式方案把 `.text` 确定为第 0 个表项，把 `.rdata` 确定为第 1 个表项。后续开始调用使用这些名称，不需要调用者自行提供表项编号。

增加第三个描述项时，只需增加一个新的命名内容块，不需要手工修改节数量、文件头大小、表项偏移、原始数据指针或相对虚拟地址。

必须保留更新后的格式方案

下面的写法正确：

```
const image: map = format_entry(image0, start)
format_finish(image)
```

下面的写法完成的是旧方案，因此会失败：

```
const image: map = format_entry(image0, start)
format_finish(image0)
```

返回的 `image` 包含入口地址，`image0` 中的入口仍然是原来的零值。

可以使用 `image0`、`image1` 和 `image` 等不同名称，也可以根据附加的数据命名。关键规则是始终把最新的格式方案传递给下一项操作。

构建流程错误会被明确拒绝

常用格式接口不会暗中猜测用户意图，而是直接拒绝无效的构建操作。

错误	结果
描述项列表为空	构造函数拒绝格式方案
描述项名称重复	构造函数拒绝格式方案
一个描述项具有多种用途	创建描述项失败
节或段名称未知	开始调用或相关声明失败
缺少可执行入口	<code>format_finish</code> 失败
构建调用与方案类别不匹配	调用拒绝该方案类别

例如，下面两个描述项不能同时存在：

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

重复名称会让后续的 `format_section_begin(plan, ".text")` 等调用无法确定对应的描述项，因此格式方案会在开始输出之前被拒绝。

自动生成的内容仍由格式方案管理

并非每一种格式数据表都会由用户作为节或段手工写出。导入、导出、动态元数据、符号表、字符串表和重定位表，都可能根据附加到格式方案的声明自动生成。

这些自动生成的内容仍然遵循同一构建流程：

1. 描述项预留用户可见的结构；
2. 命名内容块确定实际布局信息；
3. 声明辅助函数附加名称、标号和重定位信息；
4. `format_finish` 写出或补全由格式方案管理的元数据。

因此，当常用格式接口已经支持导入或重定位时，不应绕过格式方案，另行手工写入相关数据表字节。手工数据表会绕过负责管理数量、索引和最终位置的格式方案。

格式方案的实用规则

使用常用格式接口编写源代码时，可以按照下面的清单检查：

1. 首先声明完整的节或段列表。
2. 为每个描述项使用唯一且含义明确的名称。
3. 为每个描述项指定恰好一种用途和所需权限。
4. 只调用一次 `format_begin`。
5. 按照声明顺序写入各个描述项。
6. 使用正确类别的构建调用打开和关闭每个命名内容块。
7. 保留入口或数据表辅助函数返回的每一份更新方案。
8. 把最新的格式方案传给 `format_finish`。
9. 让格式接口推导数量、表项、偏移、大小和自动生成的数据表。

第 3 章将说明这些描述项在运行时的含义，包括节、可加载段、权限、文件中有实际字节的数据、未初始化数据区、逻辑大小、实际文件大小和对齐。

第 3 章：节、段、权限与未初始化数据区

可执行文件格式既要描述文件中保存的字节，也要描述加载这些字节后形成的内存映像。节、段、权限和未初始化数据区共同连接这两种视图。

常用格式接口会在内部处理各种文件格式的计算规则。源代码只需声明命名内容及其用途，格式接口会据此推导表项、文件位置、虚拟地址、大小和对齐方式。

本章把只在加载后占用内存、在文件中没有对应初始字节的空间称为未初始化数据区。这类区域通常使用 `.bss` 作为名称。

节和段回答不同的问题

节描述文件中一个有名称的组成部分。**段**描述 ELF 加载器映射到内存中的一段范围。

常用格式接口按以下方式使用它们：

文件格式系列	描述项	主要用途
PE 可执行文件或动态链接库	节	组织文件内容并描述映像映射
COFF 目标文件	节	保存链接器输入、符号和重定位信息
ELF 目标文件	节	保存链接器输入和节元数据
ELF 可执行文件或位置无关可执行文件	段	描述运行时的 <code>LOAD</code> 映射
ELF 共享对象	段	描述运行时的 <code>LOAD</code> 映射

PE 的节同时携带文件布局和内存权限信息。ELF 程序则把运行时内容放入可加载段。ELF 目标文件仍然使用节，因为它们是链接器的输入，而不是完整的运行时映像。

这一区别会影响描述项的构造方式：

```
format_section(name, purpose | permissions)
format_segment(name, format_load | permissions)
```

不要把 `format_data` 等节用途传给 `format_segment`。普通 ELF 程序的段使用 `format_load`，再由权限和实际内容区分代码、只读数据、可写数据与未初始化数据区。

每个节只能有一种用途

一个节描述项必须且只能包含一种用途。常用格式接口定义了以下节用途：

用途	适合保存的内容
<code>format_code</code>	指令和可执行数据
<code>format_data</code>	已初始化数据
<code>format_uninitialized_data</code>	未初始化数据区使用的空间
<code>format_imports</code>	自动生成的 PE 导入数据
<code>format_exports</code>	自动生成的 PE 导出数据
<code>format_resources</code>	PE 资源
<code>format_fixups</code>	PE 基址重定位信息

前三种用途可以用于 PE、COFF 和 ELF 目标文件方案。其余用途描述自动生成的 PE 节，将在 Windows 文件格式相关章节中介绍。

不能在一个节描述项中合并两种用途：

```
format_section(  
    ".bad",  
    format_code | format_data | format_readable  
)
```

格式接口无法从这种描述项推导出一致的节属性，因此会在开始输出前拒绝该格式方案。

权限描述加载后的内存

权限是独立的标志，可以与一种用途组合：

权限	含义
<code>format_readable</code>	映射后的内存范围可以读取
<code>format_writeable</code>	映射后的内存范围可以写入
<code>format_executable</code>	处理器可以执行该内存范围中的指令
<code>format_discardable</code>	PE 可以在使用后丢弃该节

`format_discardable` 只适用于普通 PE 节方案。COFF 目标文件和 ELF 目标文件不接受该标志，ELF 程序段也不接受该标志。

常见的组合如下：

```
format_code | format_readable | format_executable
format_data | format_readable
format_data | format_readable | format_writeable
format_uninitialized_data | format_readable | format_writeable
format_load | format_readable | format_executable
format_load | format_readable | format_writeable
```

汇编器不会阻止一个内存映射同时具有可写和可执行权限，但大多数程序都应把代码与可变数据分开。可以采用以下清晰的默认规则：

- 代码可读、可执行；
- 常量只读；
- 已初始化的可变数据可读、可写；
- 未初始化数据区可读、可写。

这些权限描述的是生成后的文件格式，不会限制汇编器在构造文件时能够读取或写入什么内容。

命名内容必须匹配描述项种类

节使用节的构建流程：

```
format_section_begin(plan, ".text")
...
format_section_end(plan, ".text")
```

段使用段的构建流程：

```
format_segment_begin(plan, ".text")
...
format_segment_end(plan, ".text")
```

名称用于查找已经保存在格式方案中的描述项，并不会声明一个新的节或段。开始调用会确定内容的逻辑起点和实际文件起点，结束调用会关闭这段范围，使格式接口能够确定最终大小。

应使用与格式方案匹配的构建流程：

- PE、COFF 和 ELF 目标文件方案使用节相关调用；
- ELF 可执行文件、位置无关可执行文件和共享对象方案使用段相关调用。

把段方案传给 `format_section_begin`，或者把节方案传给 `format_segment_begin`，都会产生错误，不会自动转换。

文件中的实际字节与仅存在于内存的空间

指令和数据会在文件中写出真实字节。预留空间会增加逻辑大小，但不一定增加最终文件的字节数。

当预留空间位于节或段的末尾时，常用格式接口可以把它表示为仅存在于内存的空间：

```
bss_start:  
    rb(64)
```

这正是未初始化数据区的核心关系：

逻辑大小 > 文件中有实际字节的大小

对于完全由 64 字节预留空间组成的未初始化数据区：

文件中有实际字节的大小 = 0
逻辑大小 = 64

对于一个 ELF 可加载段，如果开头有 4 个已初始化字节，后面有 64 个预留字节：

文件中有实际字节的大小 = 4
逻辑大小 = 68

加载器从文件中取得已初始化的开头部分，并为剩余范围提供已经清零的内存。

预留空间必须留在末尾，才能不占用文件字节。如果在同一输出区域的预留间隙之后继续写出字节，这段间隙就会成为文件布局的一部分。如果一段空间必须完全不出现在文件中，应为它使用单独的未初始化数据区描述项。

底层的逻辑位置 and 实际文件位置模型见[输出区域与虚拟数据](#)。使用常用格式接口的源代码通常不需要直接调用区域相关接口。

节和段中的未初始化数据区

不同文件格式系列使用不同方式表达未初始化数据区。

对于 PE、COFF 和 ELF 目标文件，应声明一个未初始化数据节：

```
format_section(  
    ".bss",  
    format_uninitialized_data  
        | format_readable  
        | format_writeable  
)
```

最终含义取决于具体文件格式：

- PE 记录节的虚拟大小，但不保存该节的原始字节；
- COFF 把该节标记为未初始化数据；
- ELF 目标文件使用一种不在文件中保存数据的节类型。

对于 ELF 可执行文件、位置无关可执行文件和共享对象，应声明一个可写的加载段，并且只在其中预留空间：

```
format_segment(  
    ".bss",  
    format_load | format_readable | format_writeable  
)
```

程序头随后会记录文件大小为零、内存大小不为零。段没有单独的 `format_bss` 用途。

包含四种节角色的 PE64 映像

下面的示例把代码、常量、可变数据和未初始化数据区分别放入不同的节：

```

// 导入常用格式接口, 并选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 按代码、只读数据、可写数据和未初始化数据声明四个节。
const image0: 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(
            ".rdata",
            format_data | format_readable
        ),
        format_section(
            ".data",
            format_data | format_readable | format_writeable
        ),
        format_section(
            ".bss",
            format_uninitialized_data
            | format_readable
            | format_writeable
        )
    )
)
format_begin(image0);

// 在可读、可执行的代码节中写入程序入口。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 只读数据节保存以零结尾的常量字符串。
format_section_begin(image0, ".rdata");
message:
    db("ready", 0);
format_section_end(image0, ".rdata");

// 可写数据节保存具有初始值的计数器。
format_section_begin(image0, ".data");
counter:
    dq(1);
format_section_end(image0, ".data");

```

```
// 未初始化数据节只预留空间，不向文件写入原始字节。
format_section_begin(image0, ".bss");
workspace:
    rb(64);
format_section_end(image0, ".bss");

// 绑定入口标号，并根据最终布局完成 PE64 文件映像。
const image: map = format_entry(image0, start)
format_finish(image);
```

节表会按照声明顺序记录四个表项。 `.text`、`.rdata` 和 `.data` 占用文件空间；`.bss` 的逻辑大小为 64 字节，原始数据大小为零。

PE 的文件对齐与映像对齐是两项不同的规则。文件中有实际字节的节数据使用 PE 文件对齐，节的虚拟地址则按照 PE 节对齐向前排列。常用格式接口会同时应用这两项规则。

在文件数据段之间放置未初始化数据区的 ELF64 映像

下面的示例特意把一个纯未初始化数据段放在最后一个只读段之前：

```

// 导入常用格式接口，并选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 声明代码、可写数据、未初始化数据和只读数据四个加载段。
const image0: 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_segment(
            ".rodata",
            format_load | format_readable
        )
    )
)
format_begin(image0);

// 代码段通过系统调用以状态码零退出。
format_segment_begin(image0, ".text");
start:
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image0, ".text");

// 可写数据段保存一个已经初始化的计数器。
format_segment_begin(image0, ".data");
counter:
    dq(1);
format_segment_end(image0, ".data");

// 该段只预留 128 字节内存，不在文件中写出数据。
format_segment_begin(image0, ".bss");
workspace:
    rb(128);
format_segment_end(image0, ".bss");

// 后续只读段仍会写入文件，但虚拟地址位于未初始化数据区之后。
format_segment_begin(image0, ".rodata");
message:
    db("ready", 0);

```

```
format_segment_end(image0, ".rodata");

// 绑定入口标号，并根据最终布局完成 ELF64 文件映像。
const image: map = format_entry(image0, start)
format_finish(image);
```

未初始化数据段会占用 128 字节内存，但在文件中不占用任何字节。因此，后面的 `.rodata` 段可以从未初始化数据段起点所在的同一文件偏移开始；它的虚拟地址仍然会越过未初始化数据段占用的内存范围。

ELF 加载对齐不要求在文件中留下整页大小的空洞。格式接口会紧凑排列文件偏移，同时选择满足以下关系的虚拟地址：

虚拟地址除以对齐值的余数 = 文件偏移除以对齐值的余数

这样既能满足加载器的对齐要求，也不必在文件中写入无用的整页填充。

对齐由格式接口推导

使用常用格式接口的源代码不应手工对齐节表项、程序头表项、原始数据指针、相对虚拟地址或加载地址。

格式接口会应用对应文件格式的规则：

- PE 分别处理文件对齐和节的虚拟地址对齐；
- COFF 与 ELF 目标文件根据用途和位数选择节对齐；
- ELF 加载段保持所需的页对齐；
- 不占用文件字节的未初始化数据区只增加逻辑内存大小，不会在文件中产生整页空洞；
- 后续文件数据从已经确定的实际文件位置继续写入。

显式的输出区域对齐适合高级的自定义布局。把它混入常用格式方案，可能会重复应用或违背格式接口的布局规则。

无效的属性组合会被尽早拒绝

描述项会拒绝不适用于自身种类的属性：

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

`format_load` 是段的用途，因此上面的节声明无效。

其他限制包括：

- 节描述项拒绝未知标志；
- 段描述项拒绝节用途标志；
- ELF 目标文件的节拒绝 PE 专用的可丢弃属性；
- COFF 目标文件的节只接受代码、数据和未初始化数据三种用途；
- 每个描述项必须且只能包含一种用途；
- 同一格式方案中的名称必须唯一。

这些错误可以防止源代码悄悄请求一种最终文件格式会忽略的属性。

实用布局规则

规划普通内容时，可以采用以下默认规则：

1. 不同的权限或存储角色分别使用独立的描述项。
2. 代码设为可读、可执行，但不可写。
3. 常量设为可读、不可写。
4. 可变数据和未初始化数据区设为可读、可写。
5. PE、COFF 和 ELF 目标文件的未初始化数据区使用 `format_uninitialized_data`。
6. ELF 运行时未初始化数据区使用只预留空间、可写的 `format_load` 段。
7. 不占用文件字节的预留空间必须位于节或段的末尾。
8. 按照格式方案中的顺序写入描述项，并关闭每个已经开始的内容块。
9. 让格式接口推导数量、地址、偏移、大小和对齐。

第 4 章将在这些布局规则的基础上介绍地址、符号、地址回填项和重定位。

[返回可执行文件格式指南](#)

第 4 章：地址、符号与重定位

标号为源代码中的逻辑地址提供稳定名称。可执行文件格式和目标文件格式再把这个地址转换成文件头、符号表、重定位记录、指令字段或加载器所需的坐标。

最重要的原则是保留每种坐标的含义。虚拟地址、相对虚拟地址、节内偏移和文件偏移可以从不同角度描述同一个字节，但它们不能相互替代。

明确所需的地址坐标

常用格式接口会处理以下几种地址形式：

地址坐标	含义
逻辑地址	标号在当前输出区域中的值
虚拟地址	文件映像加载到内存后使用的地址
相对虚拟地址	PE 中相对于映像基址的虚拟地址
节内偏移	相对于目标文件中某个节起点的位置
文件偏移	生成文件中的实际字节位置

例如，PE 入口标号具有逻辑地址，而 PE 文件头保存的是它的相对虚拟地址。目标文件中的符号保存相对于所属节起点的值，节表项则保存该节字节在文件中的偏移。

普通源代码应把标号和节起始标号交给格式接口，不应自行计算这些派生值：

```
format_entry(image0, start)
format_coff_public("caller", ".text", text_start, caller, ...)
format_coff_reloc(".text", text_start, patch_field, "callee", ...)
```

格式接口会根据所选文件格式推导入口的相对虚拟地址、符号值、重定位偏移、表项索引和文件位置。

不要把文件偏移当作指针。文件偏移标识文件中保存的字节，虚拟地址和逻辑地址标识加载后的内存位置。

指令内部引用使用汇编器地址回填项

指令引用源代码中的标号时，通常不需要显式声明格式重定位：

```
start:
    jmp finished
    nop

finished:
    ret
```

汇编器会记录指令中的符号操作数，确定最终的指令布局，再把编码后的位移写入指令字段。这项记录称为**汇编器地址回填项**。

如果目标标号在汇编期间已经确定，而且所选文件格式不需要其他程序再次调整这个字段，地址回填就会在生成文件时完全解决。

只要条件允许，就应直接在指令操作数中使用标号，不要改写为手工计算的位移。

入口地址属于最终文件映像

PE 可执行文件和动态链接库，以及 ELF 可执行文件和可在任意地址加载的位置无关可执行文件，都需要入口地址：

```
const image: map = format_entry(image0, start)
format_finish(image)
```

`format_entry` 把入口标号的逻辑地址保存到一个新的格式方案中。完成格式时，收尾处理会写入相应文件映像要求的地址形式：

- PE 保存相对于映像基址的相对虚拟地址；
- ELF 保存可执行代码的虚拟地址。

必须先定义入口标号，再绑定入口地址，并把更新后的格式方案传给 `format_finish`。

目标文件和 ELF 共享对象不使用这个构建步骤。目标文件通过符号向链接器公布名称；共享对象公布导出符号和动态元数据。把这两类格式方案传给 `format_entry` 会产生错误。

地址回填与重定位是不同操作

“重定位”有时会泛指几种相近的机制。使用时必须区分由谁处理以及解决什么问题：

机制	处理者	用途
汇编器地址回填项	汇编器	解析已知符号，并写入对应的指令字段
目标文件重定位	链接器	在最终链接时解析符号或调整符号位置
PE 基址重定位	加载器	文件映像更换加载基址后，调整其中的绝对地址
动态重定位	动态加载器	在运行时解析导入项或调整可移动数据

同一文件内的分支可能只需要汇编器地址回填项。调用外部目标文件符号时，需要目标文件重定位。可重定位 PE 文件映像中的绝对指针即使指向同一份源代码中定义的标号，也需要 PE 基址重定位。

ELF 动态导入和位置无关数据将在 ELF 相关章节中介绍。本章先建立这些格式共用的地址模型。

稳定的绝对值应在布局完成后写入

指令操作数可以保持符号形式，直到汇编器解析地址回填项。普通整数表达式则不会自动保留这种符号关系。

如果数据中需要保存绝对指针，应先预留字段，再在 `defer` 收尾块中写入最终地址：

```
entry_pointer:
    dq(0)

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

收尾处理会在指令长度、输出区域、标号和整体布局全部稳定后运行，因此写入的是真正的逻辑地址，而不是布局过程中的临时值。

这次回填解决的是字段当前应保存什么值，但它不会自动通知操作系统加载器：当文件映像换到另一个基址时还要修改该字段。PE 文件映像需要通过基址重定位描述第二项操作。

PE 基址重定位标识绝对地址字段

PE 基址重定位描述的是一个字段的位置，该字段保存与首选映像基址相关的绝对虚拟地址。加载器更改文件映像的加载基址时，会把映像基址的变化量加到这个字段中。

常用格式接口把整个过程分成四步：

1. 写出一个绝对地址字段；
2. 把该字段的逻辑地址加入重定位列表；

3. 写出自动生成的重定位节；
4. 回填最终的绝对指针值。

`format_pe_reloc_add` 会根据 PE 格式方案选择重定位宽度：

- PE32 使用 32 位高低位重定位；
- PE64 使用 64 位重定位。

调用者提供绝对地址字段的逻辑地址。格式接口会推导其相对虚拟地址，按页对重定位项分组并排序，然后写出重定位目录。

包含可重定位指针的 PE64 可执行文件

下面的文件映像要求支持地址空间布局随机化，并包含一个绝对指针：

```

// 创建要求支持地址空间布局随机化的 PE64 控制台文件映像。
import("format/format.inc");
x86.use64();

const image0: 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(
            ".reloc",
            format_fixups | format_readable | format_discardable
        )
    )
)
format_begin(image0);

// 在代码节中写入入口代码，并预留一个保存绝对虚拟地址的字段。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret

entry_pointer:
    dq(0);
format_section_end(image0, ".text");

// 登记绝对地址字段，由格式接口生成 PE 基址重定位节。
const relocs0: list = pe_reloc_new()
const relocs: list = format_pe_reloc_add(
    image0,
    relocs0,
    entry_pointer
)
format_pe_reloc_section(image0, ".reloc", relocs);

// 绑定入口标号并完成文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

defer {
    // 布局稳定后，把入口的最终逻辑地址写入绝对地址字段。
    store.u64(entry_pointer, start);
}

```

指针字段最终保存 `start` 的地址，重定位目录则把 `entry_pointer` 标记为加载器更改映像基址时必须调整的字段。

`format_pe_aslr_required` 明确要求文件映像能够重定位。如果缺少重定位目录，汇编会失败，而不会在实际上无法重定位时仍然声称支持地址空间布局随机化。

目标文件符号描述链接器可见的名称

目标文件还没有最终运行时地址。它通过符号和重定位信息，让链接器随后放置各个节并解析相互引用。

常用目标文件接口使用两类符号：

- **公开符号**由当前目标文件定义；
- **外部符号**由其他目标文件或库提供。

声明公开符号时需要提供：

名称
节名称
节起始标号
符号地址
符号类型

ELF 目标文件还会记录符号大小。格式接口根据下面的关系推导相对于节起点的符号值：

符号值 = 符号地址 - 节起始地址

外部符号没有当前文件中的节或地址，它的名称会成为重定位目标。

目标文件重定位标识待修改字段

一项目标文件重定位会关联四项信息：

包含待修改字段的节
字段在该节中的偏移
目标符号名称
重定位类型

源代码传入节起始标号和待修改字段的标号，格式接口据此推导相对于节起点的重定位偏移：

重定位偏移 = 待修改字段地址 - 节起始地址

符号索引由符号声明列表的内容决定，调用者不需要自行计算。

COFF 的重定位加数保存在待修改字段的字节中。ELF 的某些重定位形式可以另外携带显式加数。目标文件相关章节会分别说明不同字段宽度使用的重定位类型和加数约定。

调用外部函数的 COFF64 目标文件

外部符号不是当前源代码中的标号，因此示例先写入调用位移的占位值，再为该字段声明重定位：

```

// 创建只包含一个代码节的 COFF64 目标文件。
import("format/format.inc");
x86.use64();

const object0: map = format_coff64(
    list.of(
        format_section(
            ".text",
            format_code | format_readable | format_executable
        )
    )
)
format_begin(object0);

// 写入 call 操作码，并为尚未解析的相对位移保留四个字节。
format_section_begin(object0, ".text");
text_start:
caller:
    db(0xe8);
call_displacement:
    dd(0);
    ret
format_section_end(object0, ".text");

// 公布当前目标文件定义的 caller，并声明外部函数 callee。
const symbols: list = list.of(
    format_coff_public(
        "caller",
        ".text",
        text_start,
        caller,
        coff_sym_type_function
    ),
    format_coff_extern(
        "callee",
        coff_sym_type_function
    )
)
// 将调用位移字段连接到 callee 的相对地址重定位。
const relocs: list = list.of(
    format_coff_reloc(
        ".text",
        text_start,
        call_displacement,
        "callee",
        coff_rel_amd64_rel32
    )
)
const object: map = format_coff_tables(object0, symbols, relocs)
format_finish(object);

```

生成的目标文件包含：

- `.text` 节中的一个公开符号 `caller`；
- 一个尚未定义的外部符号 `callee`；
- `call_displacement` 位置的一项相对重定位；
- 一个通过符号名称解析的重定位目标。

符号表和重定位表会在 `format_finish` 阶段写出。表项索引、文件偏移和表项数量均由格式方案推导。

ELF 目标文件使用相同的声明模型

ELF 目标文件也通过相同的源代码关系声明符号和重定位：

```
format_elfobj_public(  
    name,  
    section_name,  
    section_start,  
    address,  
    size,  
    symbol_type  
)  
  
format_elfobj_extern(name, symbol_type)  
  
format_elfobj_reloc(  
    section_name,  
    section_start,  
    patch_address,  
    symbol_name,  
    relocation_type,  
    addend  
)
```

格式接口会创建 ELF 目标文件所需的符号表、字符串表、重定位节和节表项链接。调用者只需提供名称、标号、符号含义、重定位类型和加数。

ELF32 和 ELF64 的具体重定位形式留到 ELF 目标文件章节说明，以便本章始终使用同一套地址坐标模型。

格式接口会检查重定位声明

构建目标文件数据表时，会拒绝以下相互矛盾的声明：

- 公开符号指定的节必须已经声明；
- 用户符号列表中的符号名称不能重复；
- 重定位字段不能位于所属节起始地址之前；

- 格式方案中必须存在用于保存重定位记录的节；
- 每个重定位目标名称都必须对应一个已声明符号。

PE 重定位节还必须至少包含一项重定位，并且对应的节必须以 `format_fixups` 用途声明。

这些检查可以防止无效符号索引或属于其他节的地址坐标进入生成文件。

不要混淆指针值与重定位记录

字段中保存的值和重定位记录解决的是两个不同问题：

保存的指针值 = 字段当前写入的地址
重定位记录 = 以后如何修改该字段的说明

在目标文件中，占位值为零的字段加上一项重定位记录已经足够，因为最终值由链接器填写。完整的 PE 文件映像还必须先在字段中保存当前绝对地址，加载器才能在此基础上加上基址变化量。

同样，如果相对分支的位移已经在汇编期间完全确定，就不需要格式重定位记录。

地址与重定位的实用规则

1. 使用标号表示逻辑地址和指令目标。
2. 让汇编器地址回填项解析指令内部的符号字段。
3. `format_entry` 只用于 PE 或 ELF 可执行文件格式方案。
4. 把绑定入口后的新格式方案传给 `format_finish`。
5. 始终区分文件偏移、虚拟地址和相对虚拟地址。
6. 原始数据字段需要稳定的绝对地址时，在收尾处理中回填。
7. 绝对地址字段需要随 PE 映像基址变化时，为每个字段添加 PE 基址重定位。
8. 通过名称声明目标文件中的公开符号和外部符号。
9. 使用节起始标号和待修改字段标号声明目标文件重定位。
10. 让格式接口分配符号索引、重定位表项和文件位置。

第二部分将把这些基础规则应用到 Windows 文件格式，首先介绍 PE32 和 PE64 可执行文件的构造方法。

[返回可执行文件格式指南](#)

第 5 章：PE32 与 PE64 可执行文件

可移植可执行文件（PE）映像把 Windows 程序划分为多个命名节，并说明每个节如何保存在文件中、又如何映射到内存。常用格式接口根据一份已经声明的格式方案，构造 `DOS` 文件头、PE 标记、文件头、可选文件头、数据目录和节表。

本章构造普通的可执行文件映像。导入、动态链接库导出、资源、校验和以及完整的基址重定位流程将在后续章节介绍。

写入内容前选择位宽

32 位 x86 映像使用 `format_pe32`，64 位 x86 映像使用 `format_pe64`：

```
format_pe32(options, sections)
format_pe64(options, sections)
```

两个构造函数使用相同的节描述项和格式构建流程。位宽会改变机器类型、可选文件头形式、默认映像基址、指针宽度，以及绝对地址字段采用的重定位类型。

`format_begin` 会选择对应的 x86 指令模式。即便如此，仍建议在源代码开头明确调用 `x86.use32()` 或 `x86.use64()`，以便在第一条指令出现前就说明预期的 x86 指令模式。命令行选择的目标平台也应与此一致。

映像	构造函数	命令行目标平台	默认映像基址
32 位 x86	<code>format_pe32</code>	<code>x86</code> 或 <code>x86-32</code>	<code>0x00400000</code>
64 位 x86	<code>format_pe64</code>	<code>x86-64</code>	<code>0x0000000140000000</code>

默认文件对齐为 512 字节，默认内存节对齐为 4096 字节。普通源代码只需声明各节的用途和权限，不应自行计算对齐后的相对虚拟地址或原始数据在文件中的位置。

选择映像角色、子系统和安全策略

PE 构造函数接收一个选项值，这个值由几组彼此独立的选择组合而成：

映像角色	<code>format_pe_exe</code>
子系统	<code>format_pe_console</code> 或 <code>format_pe_gui</code>
内存策略	<code>format_pe_nx</code>
地址随机化策略	<code>format_pe_aslr_auto</code> <code>format_pe_aslr_required</code> <code>format_pe_aslr_disabled</code>

可执行文件方案必须恰好选择一种映像角色、一种子系统和一种地址空间布局随机化策略。未知选项或互相矛盾的选项会在开始生成输出内容前被拒绝。

普通可执行文件通常可以使用以下组合：

```
format_pe_exe
| format_pe_console
| format_pe_nx
| format_pe_aslr_auto
```

`format_pe_nx` 表示该映像支持数据页不可执行策略。每个节是否允许执行，仍由该节自身的权限决定。

子系统描述程序所处的 Windows 运行环境：

- `format_pe_console` 选择控制台子系统；
- `format_pe_gui` 选择图形界面子系统。

子系统选项不会生成运行库、创建窗口或添加导入项，它只记录加载器能够识别的子系统选择。图形界面程序仍需自行提供代码所需要的导入项和启动行为。

一个 PE64 控制台可执行文件

下面的映像把代码、只读数据、可变数据和预留空间分别放入不同的节：

```

// 导入常用格式接口，并明确选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 声明 PE64 控制台可执行文件，以及代码、数据和未初始化数据节。
const image0: 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(
            ".rdata",
            format_data | format_readable
        ),
        format_section(
            ".data",
            format_data | format_readable | format_writeable
        ),
        format_section(
            ".bss",
            format_uninitialized_data
            | format_readable
            | format_writeable
        )
    )
)
format_begin(image0);

// 在可执行代码节中定义返回 0 的程序入口。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 只读数据节保存不会在运行时修改的字符串。
format_section_begin(image0, ".rdata");
message:
    db("XIRASM PE64", 0);
format_section_end(image0, ".rdata");

// 可写数据节保存带有初始值的可变数据。
format_section_begin(image0, ".data");
counter:
    dd(1);
format_section_end(image0, ".data");

```

```
// 未初始化数据节只预留内存，不向文件写入 128 个零字节。
format_section_begin(image0, ".bss");
workspace:
    rb(128);
format_section_end(image0, ".bss");

// 绑定入口标号，并根据最终节布局完成文件映像。
const image: map = format_entry(image0, start)
format_finish(image);
```

使用 x86-64 目标平台汇编：

```
# 生成 64 位 Windows 可执行文件。
xirasm program.asm --target x86-64 -o program.exe
# 运行生成的程序，入口代码会返回 0。
.\program.exe
```

这个最小入口只返回 0。第 6 章会把它替换为对导入的 Windows 接口的显式调用。

按照节的用途理解格式方案

节列表就是文件映像的布局约定：

节	用途	权限	文件内容
<code>.text</code>	代码	读取、执行	指令
<code>.rdata</code>	已初始化数据	读取	常量字节
<code>.data</code>	已初始化数据	读取、写入	可变数据的初始值
<code>.bss</code>	未初始化数据	读取、写入	没有原始文件数据

常用格式接口会按照描述项的原有顺序，为每个描述项生成一个节表项。后续的 `format_section_begin` 和 `format_section_end` 调用会把内容写入对应名称的节。

`.bss` 会让文件映像的逻辑大小增加 128 字节，但不会在文件中增加 128 个零字节。它的节表项记录内存大小，而原始数据大小为零。如果后面还有文件中有实际字节的节，该节可以继续使用同一个文件位置，同时获得一个更靠后的、已经对齐的相对虚拟地址。

`message`、`counter` 和 `workspace` 都是普通的逻辑地址。虽然它们所在的节具有不同的文件位置和内存地址，后续指令、收尾处理、导出声明或重定位声明仍然可以使用这些标号。

定义入口后再进行绑定

创建最终格式方案之前，入口标号必须已经存在：

```
const image: map = format_entry(image0, start)
format_finish(image)
```

`format_entry` 会返回更新后的格式方案。如果把 `image0` 传给 `format_finish`，入口绑定就会被丢弃；由于可执行文件映像必须具有入口点，完成格式时将会失败。

PE 文件头以相对虚拟地址保存入口位置。源代码只需传入逻辑标号，格式接口会根据所选映像基址和最终节布局推导相对虚拟地址。

PE32 使用相同的构建流程

32 位形式只需改变构造函数、x86 指令模式和命令行目标平台，格式方案和节的构建流程保持不变：

```

// 导入常用格式接口，并明确选择 32 位 x86 指令模式。
import("format/format.inc");
x86.use32();

// 声明 PE32 控制台可执行文件，以及代码、数据和未初始化数据节。
const image0: map = format_pe32(
    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(
            ".data",
            format_data | format_readable | format_writeable
        ),
        format_section(
            ".bss",
            format_uninitialized_data
                | format_readable
                | format_writeable
        )
    )
)
format_begin(image0);

// 在代码节中定义返回 0 的 32 位程序入口。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 保存带有初始值的可变数据。
format_section_begin(image0, ".data");
counter:
    dd(1);
format_section_end(image0, ".data");

// 为运行时工作区预留内存，不增加相同大小的文件数据。
format_section_begin(image0, ".bss");
workspace:
    rb(64);
format_section_end(image0, ".bss");

// 绑定入口标号，并完成 32 位文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

```

使用 32 位 x86 目标平台汇编：

```
# 生成 32 位 Windows 可执行文件。
xirasm program32.asm --target x86 -o program32.exe
# 运行生成的程序，入口代码会返回 0。
.\program32.exe
```

PE32 映像使用 32 位可选文件头和 `i386` 机器类型。PE64 映像使用 PE32+ 可选文件头和 `AMD64` 机器类型。

位宽改变地址字段，但不改变格式方案模型

源代码保存或重定位绝对地址时，两个位宽之间的差异最为明显：

项目	PE32	PE64
绝对指针字段	32 位	64 位
常见数据声明	<code>dd(0)</code>	<code>dq(0)</code>
最终回填	<code>store.u32</code>	<code>store.u64</code>
基址重定位类型	<code>HIGHLOW</code>	<code>DIR64</code>
默认映像基址	<code>0x00400000</code>	<code>0x00000000140000000</code>
大地址感知标志	常用格式接口不使用	启用
高熵地址空间布局随机化标志	不适用	可重定位时启用

这些差异不需要两套不同的节管理代码。常用描述项、命名节调用、入口绑定和格式完成操作在两种位宽下完全相同。

不要把 64 位地址写入 PE32 数据字段，也不要为 PE64 指针使用 PE32 重定位类型。第 7 章会介绍完整的可重定位指针和动态链接库构建流程。

理解三种地址空间布局随机化策略

常用格式接口会让文件映像标志与源代码实际提供的重定位数据保持一致。

自动

只有当格式方案包含真正的 `format_fixups` 基址重定位节时，`format_pe_aslr_auto` 才会启用动态基址标志。没有基址重定位节时，文件映像仍然是有效的固定基址可执行文件，而不会错误地声称加载器能够安全地改变它的加载基址。

因此，`auto` 适合作为普通程序的默认选择：

没有基址重定位节	固定基址映像
具有基址重定位节	可重定位映像
带基址重定位节的 PE64	动态基址与高熵虚拟地址

必须启用

如果格式方案中没有 `format_fixups` 节，`format_pe_aslr_required` 会拒绝该方案。当可重定位能力属于程序必须满足的约定时，应使用这个选项。

禁用

`format_pe_aslr_disabled` 不设置动态基址标志。文件映像有意固定在首选基址时，可以使用这个选项。

地址空间布局随机化选项不会自动发现绝对指针。源代码仍须声明每一个需要基址重定位的字段。

文件对齐与内存对齐彼此独立

PE 节使用两套位置坐标：

- 文件中的原始数据按照文件对齐值排列；
- 加载后的节相对虚拟地址按照内存节对齐值排列。

常用格式接口使用 512 字节文件对齐和 4096 字节内存节对齐。因此，小型可执行文件可以在文件中紧凑存放各节的实际数据，同时让加载后的每个节从按内存页对齐的相对虚拟地址开始。

未初始化数据区最能说明为什么必须区分这两套坐标。它可以占用内存，却不占用原始文件字节。格式接口根据最终节信息计算原始数据大小、虚拟大小、原始数据文件位置、相对虚拟地址、`SizeOfHeaders` 和 `SizeOfImage`。

普通源代码不应手工插入文件填充来模仿 PE 的内存节对齐。

PE 可执行文件的常见错误

选项互相矛盾

同时选择 `EXE` 和 `DLL`、同时选择控制台和图形界面子系统，或者选择多种地址空间布局随机化策略，都会使 PE 格式方案失败。

缺少入口或丢弃入口绑定

可执行文件必须绑定入口标号，而且 `format_entry` 返回的更新后方案必须传递给 `format_finish`。

使用未声明的节名称

每个 `format_section_begin` 调用都必须引用原始方案中的一个描述项。任何节都不能打开两次，最终文件映像必须完成所有已经声明的节内容。

权限错误

代码通常需要读取和执行权限，不需要写入权限。可变数据和未初始化数据区通常需要读取和写入权限，不需要执行权限。

没有基址重定位节却声称可以重定位

可重定位能力可选时，使用 `format_pe_aslr_auto`。只有当格式方案包含重定位节，而且所有需要调整的绝对地址字段都具有重定位声明时，才应使用 `format_pe_aslr_required`。

PE 可执行文件的实用规则

1. 普通的 64 位 Windows 程序使用 `format_pe64`。
2. 只有程序必须作为 32 位 x86 代码运行时，才使用 `format_pe32`。
3. 恰好选择一种子系统和一种地址空间布局随机化策略。
4. 普通应用程序启用 `format_pe_nx`。
5. 在 `format_begin` 之前声明完整的节列表。
6. 按照用途和权限分别存放代码、常量、可变数据和未初始化数据区。
7. 使用 `format_entry` 返回的格式方案绑定入口标号。
8. 让格式接口推导节表项、相对虚拟地址、原始数据文件位置和映像大小。
9. 除非文件映像需要更严格的策略，否则使用 `format_pe_aslr_auto`。
10. 通过各自的专用流程添加导入、重定位、资源和校验和，不要手工修改文件头。

第 6 章会在可执行文件方案中加入自动生成的导入表，并真正调用一个 Windows 接口。

第 6 章：导入 Windows 接口函数

PE 可执行文件不会包含它调用的每个函数的实现。导入表记录程序需要使用的动态链接库和函数，Windows 加载器解析这些函数后，会把它们的地址写入文件映像中的导入地址表。

常用格式接口允许源代码按名称声明导入项。它会生成描述项、查找表、提示值与名称记录、动态链接库名称、地址表槽位、结束项，以及 PE 导入目录项。

导入集合是不可变的编译期值

先创建一个空的导入集合：

```
const imports0: map = pe_import_new()
```

每次加入所需函数时，都会返回一个新的映射：

```
const imports1: map = pe_import_use64(
    imports0,
    "KERNEL32.DLL",
    "ExitProcess"
)
```

应根据文件映像的位数选择对应函数：

文件映像	按名称导入	按名称导入并指定本地槽位名称
PE32	<code>pe_import_use32</code>	<code>pe_import_use32_as</code>
PE64	<code>pe_import_use64</code>	<code>pe_import_use64_as</code>

不带 `_as` 的形式会直接使用被导入函数的名称作为本地导入地址表标号。带 `_as` 的形式则把外部函数名称与汇编源代码使用的标号分开：

```
pe_import_use64_as(
    imports,
    "KERNEL32.DLL",
    "ExitProcess",
    "exit_process"
)
```

这段声明记录了三项信息：

动态链接库名称	KERNEL32.DLL
导入的函数	ExitProcess
本地地址表标号	exit_process

该标号表示导入地址表中的一个槽位，而不是函数实现本身。文件映像加载完成后，这个槽位中保存的是已经解析出的函数地址。

在映像方案中声明导入节

格式方案必须包含一个用途为 `format_imports` 的节：

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

加载器需要填写其中的地址表槽位，因此该节必须可写。它不需要执行权限。

使用下面的函数写出完整导入集合：

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

这个操作会自行打开并关闭指定的节。不要再在它外面调用一组单独的 `format_section_begin` 和 `format_section_end`。

调用导入函数的 PE64 可执行文件

下面的可执行文件导入 `ExitProcess`，为其导入地址表槽位指定小写的本地名称，并通过相对于下一条指令寻址的内存操作数调用它：

```

// 导入常用格式接口, 选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 声明 ExitProcess 导入项, 并为地址表槽位指定本地标号。
const imports0: map = pe_import_new()
const imports: map = pe_import_use64_as(
    imports0,
    "KERNEL32.DLL",
    "ExitProcess",
    "exit_process"
)

// 创建包含代码节和可写导入节的 PE64 控制台映像。
const image0: 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
        )
    )
)
format_begin(image0);

// 按 Windows 64 位调用约定传入零, 并通过导入地址表间接调用函数。
format_section_begin(image0, ".text");
start:
    sub rsp, 40
    xor ecx, ecx
    call [rel exit_process]
format_section_end(image0, ".text");

// 根据完整导入集合生成 .idata 节。
format_pe_import_section(image0, ".idata", imports);

// 绑定入口标号并完成文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

```

将它汇编为 64 位 Windows 可执行文件并运行:

生成 PE64 文件，然后运行该程序。

```
xirasm imported-exit.asm --target x86-64 -o imported-exit.exe  
.\imported-exit.exe
```

程序把零传给 `ExitProcess`，并以退出状态 0 结束。

遵循 Windows 64 位调用约定

前四个整数或指针参数依次使用：

参数 0	rcx
参数 1	rdx
参数 2	r8
参数 3	r9

调用者还必须预留 32 字节的影子空间，并在调用边界保持栈地址按 16 字节对齐。上面的示例使用：

```
sub rsp, 40  
xor ecx, ecx  
call [rel exit_process]
```

在通常的函数入口处，40 字节的调整量同时包含 32 字节影子空间和保持对齐所需的额外空间。

`ExitProcess` 不会返回。调用会返回的导入函数时，应在离开当前过程前恢复栈指针：

```
sub rsp, 40  
设置参数寄存器  
call [rel imported_slot]  
add rsp, 40
```

按照 Windows 64 位调用约定，易失寄存器在调用后应视为已经改变。使用非易失寄存器的代码必须保存并恢复它们。

`rel` 操作数很重要。它让指令相对于下一条指令编码导入地址表槽位的位置，而不是把文件映像的绝对基址写入调用指令。

PE32 中的导入调用

PE32 使用 32 位导入地址表槽位，并把这个函数的参数压入栈中：

```

// 导入常用格式接口，选择 32 位 x86 指令模式。
import("format/format.inc");
x86.use32();

// 声明 ExitProcess 导入项，并为地址表槽位指定本地标号。
const imports0: map = pe_import_new()
const imports: map = pe_import_use32_as(
    imports0,
    "KERNEL32.DLL",
    "ExitProcess",
    "exit_process"
)

// 创建关闭地址空间布局随机化的 PE32 控制台映像。
const image0: map = format_pe32(
    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(
            ".idata",
            format_imports | format_readable | format_writeable
        )
    )
)
format_begin(image0);

// 将参数零压栈，并通过导入地址表槽位间接调用函数。
format_section_begin(image0, ".text");
start:
    push 0
    call [exit_process]
format_section_end(image0, ".text");

// 生成导入节，绑定入口并完成文件映像。
format_pe_import_section(image0, ".idata", imports);

const image: map = format_entry(image0, start)
format_finish(image);

```

这个 32 位间接内存操作数包含导入地址表槽位的绝对地址，因此该最小示例选择 `format_pe_aslr_disabled`。第 7 章将说明 PE32 中的绝对地址字段如何参与基址重定位。

不要把 Windows 64 位调用使用的寄存器和影子空间规则套用到 PE32 调用。具体采用哪一种 32 位调用约定、由谁清理参数栈，都由被导入函数的接口约定决定。本例使用 `ExitProcess` 文档规定的栈参数形

式。

加载器会替换导入地址表内容

文件映像加载之前，每个地址表槽位保存的是导入元数据。导入目录告诉加载器该槽位对应哪个动态链接库和函数。

Windows 加载文件映像时会：

- 1. 加载或找到每个按名称指定的动态链接库；
- 2. 解析每个被导入的函数；
- 3. 把解析出的地址写入对应的导入地址表槽位；
- 4. 把控制权交给可执行文件入口。

调用指令从槽位中读取已经解析出的地址：

源代码标号	<code>exit_process</code>
标号位置	<code>.idata</code> 中的一个导入地址表槽位
加载后的槽位值	<code>ExitProcess</code> 的地址
调用目标	加载后的槽位值

源代码不需要计算导入描述项的相对虚拟地址、查找表的相对虚拟地址、地址表的相对虚拟地址，也不需要计算提示值与名称记录的偏移。

添加多个导入项

逐步构造导入集合，并始终保留最近一次返回的映射：

```

const imports0: map = pe_import_new()
const imports1: map = pe_import_use64_as(
    imports0,
    "KERNEL32.DLL",
    "GetStdHandle",
    "get_std_handle"
)
const imports2: map = pe_import_use64_as(
    imports1,
    "KERNEL32.DLL",
    "WriteFile",
    "write_file"
)
const imports: map = pe_import_use64_as(
    imports2,
    "KERNEL32.DLL",
    "ExitProcess",
    "exit_process"
)

```

来自同一个动态链接库的导入项共用一个描述项。来自另一个动态链接库的导入项会使用导入映射中的另一个键，并获得另一个描述项。

以同一个动态链接库、导入函数名称和本地槽位名称重复添加同一项不会改变结果。把同一个槽位名称用于不同的导入项则会被拒绝。

常用格式接口会根据最终导入值生成确定的数据表顺序。调用代码应依赖槽位标号，而不应依赖数据表中的位置。

按名称导入与按序号导入

通常应按名称导入，因为这种写法能让源代码和诊断信息清楚显示所用函数。格式层也提供按映像位数区分的序号导入形式：

```

pe_import_use32_ordinal_as(imports, dll, ordinal, slot)
pe_import_use64_ordinal_as(imports, dll, ordinal, slot)

```

导入序号必须能用 16 位表示。只有当目标动态链接库的二进制接口明确把某个序号规定为稳定约定时，才应按序号导入。调用位置仍然使用本地槽位标号。

不要把导入元数据放进代码节

导入节包含加载器需要修改的元数据，因此应当可写且不可执行。代码节只应保存指令和普通的指令地址回填项，不应手工放置导入描述项或导入表项数组。

清晰的格式方案会把各项职责分开：

<code>.text</code>	读取 + 执行	调用位置
<code>.rdata</code>	读取	接口函数调用使用的常量
<code>.data</code>	读取 + 写入	可变参数和结果
<code>.idata</code>	读取 + 写入	自动生成的导入元数据和地址表槽位

其他数据节并非必需。只声明程序实际使用的节。

常见导入错误

忘记声明导入节

`format_pe_import_section` 要求格式方案中已经声明一个用途为 `format_imports` 的节。

手工打开 `.idata`

常用辅助接口会管理导入节的完整构建流程。手工打开同一个节会造成重复或不匹配的节操作。

丢弃更新后的导入映射

每次调用 `pe_import_use*` 都会返回新的映射。如果把较早的映射传给 `format_pe_import_section`，后续添加的导入项不会写入文件映像。

调用槽位地址而不是槽位内容

导入标号表示一个指针槽位。应使用 `call [rel exit_process]` 或 `call [exit_process]` 这类间接内存调用，不能直接调用槽位本身的地址。

混用位数

PE32 应使用 32 位导入声明，PE64 应使用 64 位导入声明。两者的地址表项宽度和序号标志不同。

忽略应用二进制接口

生成正确的导入表并不会自动安排函数参数或栈状态。每个调用位置都必须遵守被导入函数所在平台的应用二进制接口。

实用导入规则

1. 为整个文件映像创建一份不可变的导入集合。

2. PE32 使用 `pe_import_use32*`，PE64 使用 `pe_import_use64*`。
3. 本地槽位别名最好与源代码的命名风格一致。
4. 声明一个可写、不可执行且用途为 `format_imports` 的节。
5. 由 `format_pe_import_section` 写出并完成整个 `.idata` 节。
6. 使用间接内存操作数，通过生成的导入地址表标号调用函数。
7. 每个调用位置都遵循正确的 Windows 调用约定。
8. 每次声明导入项后，都保留最新返回的导入映射。
9. 依赖标号和名称，不要依赖自动生成的数据表偏移。
10. 明确处理地址空间布局随机化与绝对地址重定位的要求。

第 7 章会把文件映像从可执行程序改为动态链接库，导出可供调用的符号，并添加绝对地址字段在重定位基址时需要使用的基址重定位信息。

[返回可执行文件格式指南](#)

第 7 章：DLL 导出与基址重定位

动态链接库（DLL）是一种 PE 文件映像，由其他进程加载到自己的地址空间中。DLL 可以导出函数、数据，或者同时导出两者。DLL 入口是供加载器调用的初始化代码，与外部调用方按名称查找的导出符号相互独立。

常用格式接口对 DLL 使用与可执行文件相同的 PE32 和 PE64 节构建流程，主要区别如下：

- 选择 `format_pe_dll`，而不是 `format_pe_exe`；
- 提供导出列表和用途为 `format_exports` 的节；
- 为改用其他映像基址后仍须有效的绝对地址字段提供基址重定位；
- 编写适合 DLL 加载过程调用的入口例程。

DLL 入口不是导出

Windows 在加载或卸载模块时可能调用 DLL 入口。入口接收平台规定的参数，并返回布尔结果。下面这个最小入口接受所有通知：

```
dll_entry:
    mov eax, 1
    ret
```

入口标号仍通过 `format_entry` 绑定。导出函数需要单独声明，也可以使用完全不同的标号。

DLL 入口应当只完成少量工作。复杂初始化、调用导入函数、加锁、创建线程以及调用方专用的设置，更适合放在由程序明确调用的导出函数中。

构建导出列表

导出项保存在不可变列表中：

```

const exports0: list = pe_export_new()
const exports1: list = pe_export_use64(
    exports0,
    "answer_value",
    "xir_answer_value"
)
const exports: list = pe_export_use64(
    exports1,
    "answer",
    "xir_answer"
)

```

每项声明连接两部分信息：

目标标号	DLL 内部的地址
导出名称	外部调用方能够查找的名称

PE32 使用 `pe_export_use32`，PE64 使用 `pe_export_use64`。函数导出和数据导出采用相同的声明方式，因为导出表项记录的是相对虚拟地址，而不是源语言中的值类型。

格式接口在创建 PE 名称指针表时会对导出名称排序。源代码可以按照最能表达程序结构的顺序声明导出项，不需要自行遵循表中的名称排序规则。

可重设基址的 PE64 DLL

下面的 DLL 导出名为 `xir_answer` 的函数和名为 `xir_answer_value` 的整数。函数会读取文件映像内部保存的绝对指针，因此该指针需要一项基址重定位：

```

// 导入常用格式接口，并选择 64 位 x86 指令模式。
import("format/format.inc");
x86.use64();

// 将数据标号和函数标号分别登记为外部可见的导出名称。
const exports0: list = pe_export_new()
const exports1: list = pe_export_use64(
    exports0,
    "answer_value",
    "xir_answer_value"
)
const exports: list = pe_export_use64(
    exports1,
    "answer",
    "xir_answer"
)

// 构造允许重设基址的 PE64 DLL，并声明所需的四个节。
const image0: 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_section(
            ".reloc",
            format_fixups | format_readable | format_discardable
        )
    )
)
format_begin(image0);

// DLL 入口只报告成功；导出函数通过指针读取导出的整数。
format_section_begin(image0, ".text");
dll_entry:
    mov eax, 1
    ret

answer:
    mov rax, [rel answer_pointer]

```

```

    mov eax, [rax]
    ret

// 先为绝对指针保留固定宽度的真实文件字段。
answer_pointer:
    dq(0);
format_section_end(image0, ".text");

// 导出的数据保存在可读写节中。
format_section_begin(image0, ".data");
answer_value:
    dd(42);
format_section_end(image0, ".data");

// 根据导出列表生成完整的导出目录。
format_pe_export_section(
    image0,
    ".edata",
    exports,
    "answer.dll"
);

// 登记需要随映像基址变化而调整的绝对指针字段。
const relocs0: list = pe_reloc_new()
const relocs: list = format_pe_reloc_add(
    image0,
    relocs0,
    answer_pointer
)
format_pe_reloc_section(image0, ".reloc", relocs);

// 绑定加载器入口, 并完成 DLL 文件映像。
const image: map = format_entry(image0, dll_entry)
format_finish(image);

// 布局稳定后, 把导出数据在首选映像基址下的绝对地址写入指针字段。
defer {
    store.u64(answer_pointer, answer_value);
}

```

将它汇编为 64 位 **DLL** :

```

# 使用 x86-64 目标平台生成 DLL 文件。
xirasm answer.asm --target x86-64 -o answer.dll

```

生成的 **DLL** 包含:

- 表示 **DLL** 的文件头标志;
- 位于 **dll_entry** 的加载器入口;
- 名为 **xir_answer** 的导出项, 其相对虚拟地址指向 **answer** ;

- 名为 `xir_answer_value` 的导出项，其相对虚拟地址指向 `answer_value`；
- 一项用于 `answer_pointer` 的 `DIR64` 基址重定位；
- 动态基址、高熵虚拟地址和数据页不可执行兼容标志。

指针值与重定位各有职责

示例中的三个操作共同完成绝对指针的构造。

首先，源代码预留一个真实存在的 64 位字段：

```
answer_pointer:  
    dq(0)
```

其次，布局完成后，收尾处理把按照首选映像基址计算的指针值写入该字段：

```
store.u64(answer_pointer, answer_value)
```

最后，重定位列表告诉加载器：映像基址发生变化时，必须调整这个字段：

```
format_pe_reloc_add(image0, relocs0, answer_pointer)
```

字段中存放的是按当前首选基址计算的指针值。`DIR64` 记录则指示加载器把基址差值应用到该字段。写入指针值和声明重定位缺一不可，任何一个操作都不能代替另一个。

函数本身使用相对于 `RIP` 的指令访问 `answer_pointer`，因此该指令不包含绝对映像地址。指针字段中保存的值是绝对地址，所以仍然需要重定位。

DLL 的地址空间布局随机化策略

需要保证 `DLL` 能够改用其他基址时，可以使用 `format_pe_aslr_required`。如果格式方案中没有用途为 `format_fixups` 的节，该策略会拒绝该方案。

`format_pe_aslr_auto` 会在格式方案包含该节时启用动态基址标志。它适合用于同一种源代码结构可能包含、也可能不包含可重定位绝对地址字段的情况。

`format_pe_aslr_disabled` 会保持动态基址标志为关闭状态。只有 `DLL` 被有意固定在首选映像基址时，才应使用该策略。

仅仅存在 `.reloc` 节，并不能让任意绝对地址字段自动变得安全。每一个必须随已加载文件映像移动的字段，都需要单独声明重定位。

PE32 使用 HIGHLOW 重定位

PE32 的构建流程相同，但使用与 32 位宽度对应的值：

事项	PE32 DLL	PE64 DLL
导出声明	<code>pe_export_use32</code>	<code>pe_export_use64</code>
绝对指针字段	<code>dd(0)</code>	<code>dq(0)</code>
最终回填	<code>store.u32</code>	<code>store.u64</code>
生成的基址重定位	HIGHLOW	DIR64
默认映像基址	<code>0x00400000</code>	<code>0x0000000140000000</code>
高熵虚拟地址标志	不适用	为可重定位映像启用

`format_pe_reloc_add` 会根据格式方案种类选择 HIGHLOW 或 DIR64。使用常用格式接口时，源代码只传入字段标号，不需要自行选择重定位类型的数值。

导出函数必须遵循调用方预期的函数调用约定。PE64 导出通常采用 Windows x64 函数调用约定。PE32 导出则必须把所选的 32 位函数调用约定作为外部接口的一部分，并按该约定实现函数。

从其他语言调用 DLL

任何能够加载 Windows DLL 并按名称查找导出项的语言，都可以使用生成的文件映像。下面的 C# 程序调用函数导出并读取数据导出：

```

using System;
using System.Runtime.InteropServices;

internal static class Program
{
    // 使用 Windows 加载器接口载入 DLL，并按名称查找导出地址。
    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
    private static extern IntPtr LoadLibraryW(string path);

    [DllImport("kernel32.dll", CharSet = CharSet.Ansi)]
    private static extern IntPtr GetProcAddress(
        IntPtr module,
        string name
    );

    // 函数声明必须与汇编导出采用相同的函数调用约定。
    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    private delegate int Answer();

    private static int Main()
    {
        // 载入 DLL，失败时返回单独的退出状态。
        IntPtr module = LoadLibraryW("answer.dll");
        if (module == IntPtr.Zero)
            return 1;

        // 分别取得函数导出和数据导出的地址。
        IntPtr functionAddress = GetProcAddress(module, "xir_answer");
        IntPtr dataAddress = GetProcAddress(module, "xir_answer_value");
        if (functionAddress == IntPtr.Zero || dataAddress == IntPtr.Zero)
            return 2;

        // 把函数地址转换为可调用对象，并从数据地址读取整数。
        Answer answer =
            Marshal.GetDelegateForFunctionPointer<Answer>(functionAddress);
        int value = Marshal.ReadInt32(dataAddress);

        // 同时核对函数返回值和导出数据。
        return answer() == 42 && value == 42 ? 0 : 3;
    }
}

```

外部调用方只需要了解 **DLL** 文件、导出名称和函数调用约定，不依赖 XIRASM 源代码中的标号、节表项位置或重定位表位置。

加载到其他基址

首选地址不可用时，加载器会选择另一个基址，并计算：

基址差值 = 实际加载基址 - 首选基址

对于每一项 `DIR64` 或 `HIGHLOW` 重定位，加载器都会把这个差值加到相应字段中。在上面的示例里，调整后的 `answer_pointer` 仍然指向已经移动的 `answer_value`，因此导出函数仍会返回 42。

这就是重定位目录在程序运行时的含义。只有目录结构并不足够；重定位项必须标识正确的字段，该字段也必须保存按首选基址计算的正确值。

常见 `DLL` 错误

意外导出 `DLL` 入口

加载器入口和公开接口是两套独立接口。只导出确实需要供外部调用方使用的标号。

遗漏导出节

`format_pe_export_section` 要求格式方案中已经声明一个用途为 `format_exports` 的节。

手工打开 `.edata` 或 `.reloc`

常用导出和重定位辅助接口会自行管理这些节的构建流程。不要再在外层添加节开始和结束调用。

忘记回填字段

重定位记录不会初始化字段。每一个保存原始绝对地址的字段，都必须写入布局稳定后的首选基址地址。

忘记声明重定位

在首选基址下正确的指针，改用其他基址后可能失效。所有需要随文件映像移动的绝对地址字段都必须声明重定位。

导出函数与调用约定不兼容

导出表只公开地址，不记录函数调用约定。汇编例程和其他语言中的声明必须在参数、返回结果、栈状态以及需要保持的寄存器方面完全一致。

实用 `DLL` 规则

1. 选择 `format_pe_dll`，并提供只完成少量工作的加载器入口。
2. 将导出函数和导出数据与 `DLL` 入口分开。

3. 使用目标标号和公开名称构建一份不可变的导出列表。
4. 声明一个可读、用途为 `format_exports` 的节。
5. 由 `format_pe_export_section` 生成完整的导出目录。
6. 使用固定宽度的占位字段保存绝对指针。
7. 把每个指针按稳定布局计算出的首选基址地址回填到字段中。
8. 通过 `format_pe_reloc_add` 登记每一个可重定位的绝对地址字段。
9. 写出一个可读、可丢弃且用途为 `format_fixups` 的节。
10. 使用真实的其他语言调用程序，并让 `DLL` 在非首选基址加载，以确认导出和重定位行为。

第 8 章将在最小可执行文件和 `DLL` 构建流程之外，介绍资源数据和可选的 PE 校验和。

[返回可执行文件格式指南](#)

第 8 章：PE 资源与校验和

资源和校验和都是 PE 文件映像中的可选内容，但用途完全不同：

- 资源把结构化的应用程序数据附加到文件映像中；
- PE 校验和记录最终物理文件的校验结果。

这两类内容都不属于代码节。常用格式接口会为资源安排专用节，并且只在文件映像中的字节已经稳定后计算校验和。

资源使用专用节

使用 `format_resources` 用途声明一个可读资源节：

```
format_section(".rsrc", format_resources | format_readable)
```

资源数据通常由操作系统或应用程序代码读取，不需要写入或执行权限。

常用辅助接口负责资源节的完整构建流程：

```
format_pe_resource_section(image, ".rsrc", "app.res")
```

这项调用会完成以下操作：

1. 打开已经声明的资源节；
2. 根据最终 PE 布局推导该节的相对虚拟地址；
3. 解析已经编译的资源记录；
4. 构造资源类型、名称和语言目录树；
5. 写入资源数据项及其实际内容；
6. 关闭资源节；
7. 登记 PE 资源数据目录中的字段。

不要在这项调用外部手工打开 `.rsrc`。普通源代码只需提供节名称和已编译资源文件的路径，不需要提供资源的相对虚拟地址或目录偏移。

使用已编译的资源文件

`format_pe_resource_section` 接受已经编译的 `.res` 文件，不接受 `.rc` 资源脚本，也不会把整个 `.res` 文件当作一段不透明数据直接嵌入。

资源编译器会先把供人编写的资源脚本及其输入文件转换成已编译的资源记录。XIRASM 随后读取这些记录，并重新构造 PE 资源层次结构。

一个已编译资源文件可以包含：

- 使用数字或名称表示的资源类型；
- 使用数字或名称表示的资源标识；
- 多个数字语言标识；
- 共享同一类型或名称的多个资源内容。

格式接口会按确定的顺序排列目录项，保留每份资源内容的准确大小，并应用 PE 资源格式要求的对齐。内容为空的已编译记录不会生成资源叶项。

相对路径遵循其他文件接口使用的源文件相对路径解析规则。因此，源文件与 `app.res` 位于同一目录时，可以直接写：

```
format_pe_resource_section(image, ".rsrc", "app.res")
```

一个文件映像使用的全部资源应合并到同一份已编译资源文件中，再由唯一——一个声明为

`format_resources` 的节读取。

带有资源和校验和的 PE64 可执行文件

下面的示例为一个小型 PE64 可执行文件加入已编译资源文件，并在文件映像完成后计算校验和：

```

// 导入 PE、COFF 和 ELF 的常用格式构造接口。
import("format/format.inc");

// 声明固定基址的 PE64 控制台程序，并为代码和资源分别安排节。
const image0: 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(
            ".rsrc",
            format_resources | format_readable
        )
    )
)

// 开始按照已经声明的节顺序构造文件映像。
format_begin(image0);

// 在代码节中写入返回零的程序入口。
format_section_begin(image0, ".text");
start:
    xor eax, eax
    ret
format_section_end(image0, ".text");

// 解析同目录中的已编译资源文件，并生成完整的资源节。
format_pe_resource_section(
    image0,
    ".rsrc",
    "app.res"
);

// 绑定入口并完成布局，然后对最终物理文件计算 PE 校验和。
const image: map = format_entry(image0, start)
format_finish(image);
format_pe_checksum(image);

```

格式方案确定 `.rsrc` 是第二个节。源代码不需要计算它的节表项索引、原始数据文件位置、相对虚拟地址、目录偏移或最终大小。

校验和调用也不需要接收节列表。它会读取已经声明的 PE 格式方案，并按照声明顺序处理文件头和每个节在文件中的实际字节。

校验和覆盖最终文件

`format_pe_checksum(image)` 会把校验和计算登记为最终操作。它会：

1. 读取 PE 文件头时把校验和字段按零处理；
2. 累加每个已声明节在文件中的实际字节；
3. 归并累计得到的 16 位和；
4. 加上最终物理文件大小；
5. 把结果保存到可选文件头中。

只存在于内存的未初始化数据区不提供任何文件字节。它的逻辑大小仍会影响加载后的内存映像，但不会为校验和增加需要读取的数据。

PE 校验和不是数字签名，也不能证明文件内容未被恶意修改。它只是 PE 格式规定的一个校验字段。签名、信任关系和防篡改能力属于另外的问题。

在校验和之前登记字节回填

各项收尾处理按照登记顺序执行。任何会改变输出字节的收尾处理，都必须在 `format_pe_checksum` 之前登记。

例如，绝对指针的回填应放在校验和之前：

```
format_finish(image)

defer {
    store.u64(pointer_slot, target);
}

format_pe_checksum(image)
```

这样，校验和读取到的就是已经写入的指针值。只读取数据的断言可以登记在校验和之后，因为它不会改变文件内容。

计算校验和后，不要再修改、追加、签名或以其他方式重写文件；如果后续操作确实会改写文件，就必须按照该操作自身的文件处理规则同时更新校验和。

可执行文件和动态链接库都能使用资源

同一个资源辅助接口既适用于 PE32 和 PE64 格式方案，也适用于可执行文件和动态链接库。已编译资源的表示方式不依赖指针位宽。

校验和辅助接口同样不要求用户为不同位宽采用不同流程。PE32 和 PE64 的校验和字段位于可选文件头中相同的相对位置，格式接口会自动使用正确的文件映像布局。

资源与导入、导出和基址重定位彼此独立。较大的文件映像可以同时声明以下自动生成内容所使用的节：

```
list.of(  
    format_section(  
        ".text",  
        format_code | format_readable | format_executable  
    ),  
    format_section(  
        ".idata",  
        format_imports | format_readable | format_writeable  
    ),  
    format_section(  
        ".edata",  
        format_exports | format_readable  
    ),  
    format_section(  
        ".rsrc",  
        format_resources | format_readable  
    ),  
    format_section(  
        ".reloc",  
        format_fixups | format_readable | format_discardable  
    )  
)
```

每个自动生成辅助接口都负责对应节的完整构建流程。这些节完成构建，并登记完所有会改变字节的回填后，再登记校验和计算。

资源和校验和的常见错误

传入 `.rc` 资源脚本

资源辅助接口读取已经编译的 `.res` 记录。应先编译资源脚本，再汇编 PE 文件映像。

把 `.res` 文件当作普通原始数据

辅助接口会解析其中的记录并重新构造 PE 资源树。如果需要的结果只是一段不透明的字节序列，应使用普通的数据写出功能。

声明错误的节用途

`format_pe_resource_section` 要求目标节使用 `format_resources` 用途声明。

手工打开 `.rsrc`

常用资源辅助接口会自行打开资源节、写出内容、完成相关字段并关闭该节。不要在调用外部再添加一组开始和结束操作。

过早计算校验和

应先登记所有会改变字节的 `defer` 块。校验和如果根据占位字节计算，后续收尾处理写入真实值后就会失效。

把校验和当作身份验证

校验和本身无法发现恶意替换。需要确认文件来源和真实性时，应使用合适的签名与验证机制。

资源和校验和的实用规则

1. 把一个文件映像使用的资源脚本编译成一份 `.res` 输入文件。
2. 声明一个可读的 `format_resources` 节。
3. 让 `format_pe_resource_section` 负责 `.rsrc` 的完整构建流程。
4. 传入文件路径、名称和标号，不要传入资源相对虚拟地址或表项索引。
5. 在计算校验和之前完成导入、导出、资源和重定位内容。
6. 在 `format_pe_checksum` 之前登记每一项最终字节回填。
7. 把未初始化数据区视为内存大小，而不是文件字节。
8. 仅在输出流程确实需要时计算校验和。
9. 不要混淆 PE 校验和与数字签名。
10. 后续操作只要改变了校验和覆盖的字节，就应重新计算校验和。

第 9 章将从可以加载的 PE 文件映像转向 COFF 目标文件。在目标文件中，节、符号和重定位描述的是本机链接器需要完成的工作，而不是加载器需要完成的工作。

[返回可执行文件格式指南](#)

第 9 章：COFF32 与 COFF64 目标文件

COFF 目标文件不是操作系统可以直接加载的程序。它由节、符号和交给链接器处理的重定位请求组成。

因此，目标文件的格式构建流程与可执行文件不同：

- 没有 PE 可选文件头；
- 没有子系统或映像基址；
- 没有可执行入口；
- 公开符号表示本目标文件提供的定义；
- 外部符号表示需要由其他目标文件或库提供的定义；
- 重定位告诉链接器，哪些已编码字段依赖这些符号。

常用格式接口会在内部管理节编号、符号索引、重定位表项和各数据表的文件偏移。

创建目标文件方案

32 位 x86 目标文件使用 `format_coff32`，64 位 x86 目标文件使用 `format_coff64`：

```
const object0: 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
    )
  )
)
```

描述项的顺序就是 COFF 节的顺序。格式接口会推导节数量，并为每个描述项写入一个节表项。

目标文件中的节采用与 PE 节相同的常用构建流程：

```
format_begin(object0)
format_section_begin(object0, ".text")
...
format_section_end(object0, ".text")
```

不要调用 `format_entry`。目标文件通过符号公开定义；当多个目标文件被链接成最终映像时，才会选择可执行文件的入口。

公开符号描述本文件提供的定义

`format_coff_public` 声明一个由目标文件内某个节定义的符号：

```
format_coff_public(  
    "main",  
    ".text",  
    text_start,  
    main,  
    coff_sym_type_function  
)
```

各参数依次表示：

1. 链接器可见的符号名称；
2. 声明该符号所属的节名称；
3. 该节起始位置的标号；
4. 符号自身的标号；
5. COFF 符号类型。

格式接口会计算符号所属的节编号，以及符号相对于该节起点的值。源代码不需要传入这两个数值字段。

可调用的代码使用 `coff_sym_type_function`，普通数据使用 `coff_sym_type_null`。

外部符号描述本文件所需的定义

`format_coff_extern` 声明一个必须由其他位置提供的符号：

```
format_coff_extern("helper", coff_sym_type_function)
```

生成的符号没有定义它的节。链接器随后会在参与链接的其他目标文件和库中查找名称匹配的定义。

当前常用 COFF 构建流程使用能够放入八字节 COFF 名称字段的名称。节名称、公开符号名称、外部符号名称和重定位目标名称都应限制在这个范围内。

重定位标记编码中的待修改字段

调用外部函数时，指令中有一个四字节位移字段。链接器知道目标地址后才能确定该字段的最终值，因此需要先写入占位值，并在字段本身的位置定义标号：

```
db(0xe8)
helper_disp:
dd(0)
```

重定位应引用位移字段的标号，而不是调用指令的操作码：

```
format_coff_reloc(
    ".text",
    text_start,
    helper_disp,
    "helper",
    coff_rel_amd64_rel32
)
```

格式接口会推导：

- 根据 `.text` 确定重定位所属的节编号；
- 根据 `helper_disp - text_start` 确定重定位在节内的偏移；
- 通过查找 `helper` 确定符号表索引；
- 在完成格式时确定重定位表的文件位置。

重定位类型必须与处理器架构和已编码字段相匹配。相对调用使用：

目标文件种类	相对调用的重定位
COFF32	<code>coff_rel_i386_rel32</code>
COFF64	<code>coff_rel_amd64_rel32</code>

把符号和重定位附加到方案

符号声明和重定位声明都是普通的编译期列表：

```
const symbols: list = list.of(...)
const relocs: list = list.of(...)
```

使用 `format_coff_tables` 把它们附加到格式方案：

```
const object: map = format_coff_tables(object0, symbols, relocs)
format_finish(object)
```

这个函数会返回一份新的方案。必须保留返回值，并把它传给 `format_finish`。

完成格式时，格式接口会：

1. 按照声明的节对重定位进行分组；
2. 根据符号名称解析每个重定位目标；
3. 写入重定位表项；
4. 按照声明顺序写入符号表；
5. 写入最小的 COFF 字符串表结束项；
6. 回填文件头以及各节表项中的重定位字段。

一个可以参与链接的 COFF64 目标文件

下面的目标文件定义了 `main`、已初始化数据和通常称为 BSS 的未初始化数据区。它通过 64 位 x86 相对重定位调用名为 `helper` 的外部函数：

```

// 导入常用格式接口，并声明代码、已初始化数据和未初始化数据三个节。
import("format/format.inc");

const object0: 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(object0);

// 写入 main，并为链接器稍后处理的 helper 相对调用保留四字节位移字段。
format_section_begin(object0, ".text");
text_start:
main:
    sub rsp, 40
    db(0xe8);
helper_disp:
    dd(0);
    add rsp, 40
    ret
format_section_end(object0, ".text");

// 写入一个具有初始值的公开数据对象。
format_section_begin(object0, ".data");
data_start:
answer:
    dd(42);
format_section_end(object0, ".data");

// 只预留 64 字节逻辑空间，不向目标文件写入对应载荷。
format_section_begin(object0, ".bss");
bss_start:
scratch:
    rb(64);
format_section_end(object0, ".bss");

// 声明本文件提供的符号，以及需要由其他目标文件提供的 helper。
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("helper", coff_sym_type_function)
)

// 把 helper 的相对调用位移交给本机链接器完成重定位。
const relocs: list = list.of(
    format_coff_reloc(
        ".text",
        text_start,
        helper_disp,
        "helper",
        coff_rel_amd64_rel32
    )
)

// 使用包含符号表和重定位表的新方案完成 COFF64 目标文件。
const object: map = format_coff_tables(object0, symbols, relocs)
format_finish(object);

```

栈指针调整会为 64 位 Windows 调用保留影子空间，并使外部调用时的栈满足对齐要求。目标文件本身没有定义 `helper`。

可以使用一份 `C` 源代码提供缺失的定义：

```

// 引用 XIRASM 目标文件公开的 64 字节未初始化数组。
extern unsigned char scratch[64];

// 提供 helper，并确认链接后分配的数组最后一个字节可以写入。
int helper(void) {
    scratch[63] = 7;
    return scratch[63] == 7 ? 0 : 1;
}

```

将 `C` 源代码编译成 COFF64 目标文件，再使用本机链接器把它与 XIRASM 目标文件链接。链接器会解析尚未定义的 `helper` 符号，应用 `REL32` 重定位，并生成最终的可执行文件。

`.bss` 节表项报告的大小为 64，原始数据指针为零。目标文件不会保存 64 字节载荷；链接器会在最终映像中分配这段空间，`C` 函数则确认最后一个字节可以写入。

COFF32 使用相同的方案模型

32 位构建流程需要改变目标文件种类、所选应用二进制接口要求的符号写法，以及重定位类型：

```

// 导入常用格式接口，并创建一个只包含代码节的 COFF32 目标文件。
import("format/format.inc");

const object0: map = format_coff32(
    list.of(
        format_section(
            ".text",
            format_code | format_readable | format_executable
        )
    )
)

format_begin(object0);

// 写入 _main，并为尚未解析的 _helper 相对调用保留位移字段。
format_section_begin(object0, ".text");
text_start:
_main:
    db(0xe8);
helper_disp:
    dd(0);
    ret
format_section_end(object0, ".text");

// 公开 _main，并声明由其他目标文件提供的 _helper。
const symbols: list = list.of(
    format_coff_public(
        "_main",
        ".text",
        text_start,
        _main,
        coff_sym_type_function
    ),
    format_coff_extern("_helper", coff_sym_type_function)
)

// COFF32 相对调用使用 i386 的 REL32 重定位类型。
const relocs: list = list.of(
    format_coff_reloc(
        ".text",
        text_start,
        helper_disp,
        "_helper",
        coff_rel_i386_rel32
    )
)

// 附加符号和重定位信息，然后完成目标文件。
const object: map = format_coff_tables(object0, symbols, relocs)
format_finish(object);

```

本例中的前导下划线遵循一种常见的 32 位 C 应用二进制接口命名约定。它们是源代码明确声明的符号名称，并不是 COFF 格式接口自动进行的转换。符号名称必须与项目使用的编译器和链接器保持一致。

BSS 必须保持文件中无载荷

未初始化数据描述项会为节设置 COFF 未初始化数据属性。只包含预留操作的节会增加逻辑大小，但不会写入载荷字节。节表项记录这个逻辑大小，而原始数据指针保持为零。

常用格式接口会同时维护两个位置：

- 潜在文件位置保留预留空间的逻辑范围；
- 实际文件位置标记下一个真正写入文件的位置。

重定位表和符号表从实际文件位置开始写入。这样可以保持目标文件紧凑，也能防止元数据被错误地计入 BSS 原始数据。

常见的 COFF 目标文件错误

调用 `format_entry`

COFF 目标文件不包含可执行入口。应当公开一个函数符号，并让最终的链接过程选择入口。

丢弃更新后的方案

`format_coff_tables` 返回的方案拥有符号列表和重定位列表。如果把较早的方案传给 `format_finish`，生成的目标文件就会遗漏这些表。

把重定位放在操作码上

对于相对调用，重定位属于操作码之后的四字节位移字段。

使用了错误架构的重定位

`format_coff32` 应使用 32 位 x86 重定位常量，`format_coff64` 应使用 64 位 x86 重定位常量。

遗漏外部符号

重定位目标必须出现在符号列表中。使用 `format_coff_extern` 声明尚未解析的目标。

重复使用符号名称

常用格式接口会拒绝名称重复的公开符号或外部符号，否则按名称查找重定位目标时会产生歧义。

向 BSS 写入字节

未初始化数据应使用预留操作。具有初始值的字节应放在 `format_data` 节中。

假定调用约定

符号表只描述名称和位置，不描述参数如何传递。汇编代码、编译生成的目标文件和最终的链接命令必须使用一致的应用二进制接口。

COFF 目标文件的实用规则

1. 根据目标链接器接受的输入，选择 `format_coff32` 或 `format_coff64`。
2. 分别声明代码、已初始化数据和 BSS 节。
3. 每个被符号或重定位引用的节都应定义稳定的起始标号。
4. 使用 `format_coff_public` 声明每个需要让链接器看到的定义。
5. 使用 `format_coff_extern` 声明每个尚未解析的外部需求。
6. 把重定位标号放在编码中的待修改字段上，而不是放在周围的指令上。
7. 选择同时匹配处理器架构和字段编码方式的重定位类型。
8. 使用 `format_coff_tables` 附加最终的符号列表和重定位列表。
9. 不要添加可执行入口，并使用更新后的方案完成格式。
10. 使用本机链接器，把目标文件与真正引用它或提供外部定义的其他目标文件链接，以确认生成结果。

第 10 章将重新介绍可加载映像，并说明 ELF32 与 ELF64 可执行文件如何采用紧凑的文件布局，以及如何让各个 `LOAD` 段分别对齐。

第 10 章：ELF32 与 ELF64 可执行文件

ELF（可执行与可链接格式）可执行文件是一种可加载映像，适用于采用 ELF 进程模型的系统。ELF 文件头记录处理器架构和入口地址，程序头则描述加载器需要映射到进程中的文件字节范围与内存范围。

本章构造固定地址可执行文件：

- `format_elf32(format_elf_exec, ...)` 创建 `i386` `ET_EXEC` 映像；
- `format_elf64(format_elf_exec, ...)` 创建 `AMD64` `ET_EXEC` 映像；
- 每个普通描述项生成一个 `PT_LOAD` 程序头；
- `format_entry` 绑定开始执行的地址；
- 格式接口推导程序头的数量、文件偏移、地址、大小、标志和对齐字段。

位置无关映像和动态导入将在第 11 章介绍。

选择 ELF 类别和执行模式

ELF 格式方案接收执行模式和一个有序的段列表：

```
const image0: 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  
    )  
  )  
)
```

32 位 x86 代码使用 `format_elf32`，64 位 x86 代码使用 `format_elf64`。所选格式决定 ELF 类别、机器类型、文件头字段宽度和默认加载基址。

普通固定地址模式的默认值如下：

格式方案	ELF 类别	机器类型	默认映像基址
<code>format_elf32</code>	ELF32	<code>i386</code>	<code>0x08048000</code>
<code>format_elf64</code>	ELF64	<code>AMD64</code>	<code>0x00400000</code>

`format_elf32` 目前只接受 `format_elf_exec`。`format_elf64` 还支持位置无关模式，但该模式会改变加载和动态链接方式，因此放在下一章单独介绍。

ELF 可执行文件使用可加载段

加载器不会按照源代码中的节名称组织可执行文件，而是映射程序头描述的范围。

因此，普通可执行文件格式接口使用段描述项：

```
format_segment(  
    ".text",  
    format_load | format_readable | format_executable  
)
```

每个普通可执行文件段都必须包含 `format_load` 用途标志。权限用于描述加载后的内存映射：

内容	建议权限
指令	可读、可执行
可修改的初始化数据	可读、可写
未初始化数据区（BSS）	可读、可写
常量	可读

描述项的顺序就是程序头的顺序。使用以下接口打开和关闭段内容：

```
format_segment_begin(image0, ".text")  
...  
format_segment_end(image0, ".text")
```

ELF 可执行文件方案不能使用 `format_section_begin`。节是目标文件提供给链接器的组织单位，本章使用的可加载段则是运行时映射单位。

固定地址 ELF64 可执行文件

下面的程序会检查初始化数据，写入未初始化数据区的最后一个字节，读取位于该区域之后的常量，最后通过 Linux x86-64 系统调用接口退出：

```

// 导入常用格式接口。
import("format/format.inc");

// 创建固定地址 ELF64 格式方案，并按顺序声明代码、数据、
// 未初始化数据和只读数据四个可加载段。
const image0: 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_segment(
            ".rodata",
            format_load | format_readable
        )
    )
)

// 开始构造文件映像。
format_begin(image0);

// 写入可执行代码，并通过相对指令指针寻址访问其他可加载段。
format_segment_begin(image0, ".text");
start:
    mov eax, [rel answer]
    cmp eax, 42
    jne failed

    // 验证未初始化数据区的最后一个字节已经映射为可写内存。
    lea rbx, [rel scratch]
    mov byte [rbx + 127], 7
    cmp byte [rbx + 127], 7
    jne failed

    // 验证未初始化数据区之后的只读常量仍可正常访问。
    mov eax, [rel marker]
    cmp eax, 0x11223344
    jne failed

    // 将退出状态设为 0，并调用 Linux x86-64 的 exit 系统调用。
    xor edi, edi
exit:
    mov eax, 60

```

```

    syscall
failed:
    mov edi, 1
    jmp exit
format_segment_end(image0, ".text");

// 写入占用文件空间的可修改初始化数据。
format_segment_begin(image0, ".data");
answer:
    dd(42);
format_segment_end(image0, ".data");

// 只预留运行时内存，不向文件写入初始化字节。
format_segment_begin(image0, ".bss");
scratch:
    rb(128);
format_segment_end(image0, ".bss");

// 写入只读常量。
format_segment_begin(image0, ".rodata");
marker:
    dd(0x11223344);
format_segment_end(image0, ".rodata");

// 绑定入口标号，并使用更新后的格式方案完成文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

```

程序使用相对指令指针的内存引用访问其他可加载段中的标号，也就是 x86-64 的 `RIP` 相对寻址。`jne failed` 等向前条件分支由汇编器解析，不需要手工添加 `near` 限定符。

`exit` 系统调用要求：

- `eax` 保存系统调用号 `60`；
- `edi` 保存进程退出状态。

程序直接使用内核接口，因此不需要运行库，也不需要动态加载所需的元数据。

文件布局与内存布局不同

上面的 ELF64 示例生成四个可加载映射。具体大小取决于指令编码，但各段的文件内容与内存空间始终保持以下关系：

段	文件字节	内存字节	权限
<code>.text</code>	指令	指令	读、执行
<code>.data</code>	初始化数据	初始化数据	读、写
<code>.bss</code>	无	预留空间	读、写
<code>.rodata</code>	常量	常量	读

对于这份源代码，输出文件为 368 字节，可加载段的布局如下：

段	文件偏移	虚拟地址	文件大小	内存大小
<code>.text</code>	<code>0x120</code>	<code>0x400120</code>	<code>0x48</code>	<code>0x48</code>
<code>.data</code>	<code>0x168</code>	<code>0x401168</code>	<code>0x04</code>	<code>0x04</code>
<code>.bss</code>	<code>0x16C</code>	<code>0x40216C</code>	<code>0x00</code>	<code>0x80</code>
<code>.rodata</code>	<code>0x16C</code>	<code>0x40316C</code>	<code>0x04</code>	<code>0x04</code>

未初始化数据区没有文件字节，因此 `.bss` 和后面的只读段可以使用相同的文件偏移，但它们的虚拟地址不同。

每个可加载段都保持页内偏移同余：

```
p_vaddr % p_align == p_offset % p_align
```

格式接口会把逻辑地址推进到合适的内存页，同时让文件内容紧接在前一段实际字节的末尾。它不会在两个可加载段之间写入整页零字节。

因此，每增加一个段不会让文件额外膨胀数千字节，整个文件仍然只有数百字节。

BSS 只占用内存，不包含文件内容

只包含预留空间的可加载段会记录：

```
p_filesz = 0
p_memsz = 预留空间的逻辑大小
```

加载器会为此范围建立可写且初始值为零的内存。示例对 `scratch` 最后一个预留字节的访问，证明整个范围都已经映射并且允许写入。

在 BSS 段中应使用 `rb` 等预留操作。已经初始化的字节应放入文件中有实际字节的数据段。

BSS 后面可以继续放置另一个有文件内容的段。格式接口会让下一个段从当前实际文件末尾继续紧凑排列，同时为它分配后面的虚拟内存页，避免新映射与 BSS 的内存范围重叠。

在入口标号定义后绑定入口

入口标号属于代码段：

```
start:
    ...

const image: map = format_entry(image0, start)
format_finish(image)
```

`format_entry` 会返回更新后的格式方案。调用 `format_finish` 时必须传入这个返回值。

对于 ELF 可执行文件，入口字段保存最终虚拟地址，不是文件偏移，也不是相对于映像基址的数值。

完成 ELF32 或 ELF64 可执行文件之前，格式接口要求入口地址不能为零。

ELF32 使用相同的段模型

ELF32 会改变文件头字段宽度、机器类型、地址宽度以及系统调用所用的应用二进制接口，但格式方案的构建流程保持不变。

源代码必须选择 32 位 x86 指令模式：

```

// 导入常用格式接口，并选择 32 位 x86 指令模式。
import("format/format.inc");

x86.use32();

// 创建固定地址 ELF32 格式方案，段的用途和权限与 ELF64 示例一致。
const image0: map = format_elf32(
    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_segment(
            ".rodata",
            format_load | format_readable
        )
    )
)

// 开始构造文件映像。
format_begin(image0);

// 使用 32 位绝对地址访问数据，并验证各个可加载段。
format_segment_begin(image0, ".text");
start:
    mov eax, [answer]
    cmp eax, 42
    jne failed

    // 验证 BSS 段的最后一个预留字节可以读写。
    mov byte [scratch + 63], 7
    cmp byte [scratch + 63], 7
    jne failed

    // 验证 BSS 之后的只读常量可以访问。
    mov eax, [marker]
    cmp eax, 0x11223344
    jne failed

    // 将退出状态设为 0，并调用 32 位 Linux 的 exit 系统调用。
    xor ebx, ebx
exit:
    mov eax, 1

```

```

    int 0x80
failed:
    mov ebx, 1
    jmp exit
format_segment_end(image0, ".text");

// 写入可修改的初始化数据。
format_segment_begin(image0, ".data");
answer:
    dd(42);
format_segment_end(image0, ".data");

// 为未初始化数据预留 64 字节运行时内存。
format_segment_begin(image0, ".bss");
scratch:
    rb(64);
format_segment_end(image0, ".bss");

// 写入只读常量。
format_segment_begin(image0, ".rodata");
marker:
    dd(0x11223344);
format_segment_end(image0, ".rodata");

// 绑定入口标号, 并完成 ELF32 文件映像。
const image: map = format_entry(image0, start)
format_finish(image);

```

32 位 Linux 退出接口要求 `eax` 保存系统调用号 `1`, `ebx` 保存退出状态。

这个示例生成 269 字节的 ELF32 可执行文件。它的 BSS 可加载段文件大小为零, 内存大小为 64 字节。BSS 后面的只读可加载段继续使用相同的实际文件偏移, 但位于后面的虚拟内存页。

64 位 Linux 系统可能需要启用 `IA32` 执行支持才能运行 32 位可执行文件。这属于操作系统配置问题, 不会改变 ELF 布局。

程序头是运行时约定

本章的固定地址可执行文件不需要节头表。加载器使用以下信息:

- ELF 文件头;
- 入口虚拟地址;
- 有序的程序头表;
- 可加载段的文件偏移、地址、大小、权限和对齐。

`.text` 和 `.bss` 等描述项名称只存在于源代码的格式方案中。源代码通过这些名称打开对应的段, 诊断信息也可以用它们指出错误。这些紧凑映像不会把它们写成运行时名称。

如果链接器需要命名节、符号表和重定位节，应使用 ELF 目标文件。第 12 章将介绍这种构建流程。

固定地址可执行文件不提供动态导入

直接进行系统调用不需要导入符号。调用 `C` 运行库或其他共享库时，则需要更多 ELF 元数据：

- 指定动态加载器的解释器记录；
- 动态字符串表和动态符号表；
- 过程链接表（`PLT`）与全局偏移表（`GOT`）所需的数据；
- 动态重定位；
- 依赖库记录。

不要在本章所示的固定地址可执行文件中调用尚未解析的库符号。第 11 章将介绍 ELF64 动态导入的常用构建流程。

ELF 可执行文件常见错误

省略 `format_elf_exec`

常用构造函数要求显式提供一种执行模式。本章使用 `format_elf_exec`。

使用节而不是可加载段

ELF 可执行文件方案使用 `format_segment_begin` 打开段，使用 `format_segment_end` 关闭段。

省略 `format_load`

每个普通可执行文件段除了权限标志，还必须包含 `format_load` 用途标志。

让数据段具有执行权限

只有可修改的数据和 BSS 需要写权限。常量应保持只读，可写数据也不应具有执行权限。

向 BSS 写入初始化字节

使用 `rb` 或其他预留操作建立 BSS 空间。已经初始化的值应移动到文件中有实际字节的数据段。

手工添加内存页填充

不要在段之间写入整页零字节。格式接口会推导满足页内偏移同余关系的虚拟地址，同时保持实际文件字节紧凑排列。

丢弃更新后的入口格式方案

完成格式时应传入 `format_entry` 返回的值，而不是较早绑定的格式方案。

认为段名称会写入文件

这种紧凑可执行文件由程序头驱动。源代码中的段名称不会形成节头字符串表。

在没有动态元数据时调用库

直接调用内核和动态导入库函数是两种不同的构建流程。如果符号需要由运行时链接器解析，应使用第 11 章介绍的动态导入格式接口。

ELF 可执行文件实用规则

1. 根据指令位数选择 `format_elf32` 或 `format_elf64`。
2. 使用本章的固定地址流程时传入 `format_elf_exec`。
3. 把代码、可修改数据、BSS 和常量分别描述为不同的可加载段。
4. 每个可加载段只设置其内容确实需要的权限。
5. 按照描述项顺序调用 `format_segment_begin` 和 `format_segment_end`。
6. 为 BSS 预留空间，但不要写出初始化字节。
7. 让格式接口推导文件偏移、虚拟地址和页内偏移同余关系。
8. 在可执行代码中定义入口标号。
9. 使用 `format_entry` 返回的更新后格式方案完成文件。
10. 仅在不需要动态运行时依赖时使用直接系统调用。

第 11 章将介绍 ELF64 位置无关可执行文件和动态导入，包括动态加载器、过程链接表、全局偏移表和运行时重定位模型。

第 11 章：位置无关可执行文件与动态导入

ELF64 的常用格式接口还支持两种可执行文件构建流程：

- 使用相对寻址和直接系统调用的位置无关可执行文件；
- 通过自动生成的过程链接表和全局偏移表导入函数的固定地址可执行文件。

这两种流程彼此独立。选择位置无关可执行文件会改变加载器放置映像的方式，但不会自动加入运行时解释器或导入符号。添加动态导入表会生成解释器记录、动态符号数据、过程链接表、全局偏移表和重定位记录，但当前常用导入流程要求使用固定地址的 `EXEC` 模式。

本章分别说明这两种模型，以及它们之间的使用边界。

显式选择 ELF64 位置无关模式

使用下面的调用创建位置无关格式方案：

```
format_elf64(format_elf_pie, segments)
```

生成的 ELF 文件头使用 `ET_DYN`，映像基址为零。各个 `LOAD` 项中的虚拟地址表示映像内部的偏移，不再从第 10 章使用的固定 ELF64 基址推导。

操作系统会选择运行时加载地址。映射可执行文件时，它会把这个地址加到入口值和各个 `LOAD` 虚拟地址上。

常用的位置无关构建流程目前只支持 ELF64。把 `format_elf_pie` 传给 `format_elf32` 会被拒绝。

位置无关性取决于源代码

仅仅使用 `ET_DYN` 文件头，并不能让源代码中的绝对值自动变成位置无关形式。

代码应通过相对于当前指令位置的寻址访问内部标号：

```
lea rsi, [rel message]
mov eax, [rel value]
call helper
```

整个映像移动后，直接相对跳转或相对调用仍然有效，因为指令和目标会移动相同的距离。

绝对数据字段则不同。例如，直接写出 `dq(start)` 会保存 `start` 在链接阶段确定的值。本章的最小位置无关流程不会为这个字段生成动态重定位，因此运行时加载器不知道需要把实际加载地址加到该值上。

如果位置无关可执行文件没有动态重定位，应使用相对代码引用、相对偏移，或者在运行时计算所需地址。

多段 ELF64 位置无关可执行文件

下面的示例包含可执行代码、未初始化数据区和只读数据。代码通过相对于指令指针的寻址访问两个数据段，因此映像加载到任何地址时都能正常工作。

```

// 导入常用格式接口，并创建可在任意地址加载的 ELF64 映像。
import("format/format.inc");

const image0: 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(image0);

// 代码通过相对地址访问内部的预留空间和只读消息。
format_segment_begin(image0, ".text");
start:
    lea rbx, [rel scratch]
    mov dword [rbx], 0x5a
    cmp dword [rbx], 0x5a
    jne failed

    // 直接调用内核，把只读消息写到标准输出。
    mov eax, 1
    mov edi, 1
    lea rsi, [rel message]
    mov edx, message_end - message
    syscall

    xor edi, edi
    jmp finish

failed:
    mov edi, 1

finish:
    // 使用成功或失败状态结束进程。
    mov eax, 60
    syscall
format_segment_end(image0, ".text");

// 未初始化数据区只预留运行时内存，不在文件中写出初始字节。
format_segment_begin(image0, ".bss");
scratch:
    rb(64);
format_segment_end(image0, ".bss");

// 只读段保存程序需要输出的消息。
format_segment_begin(image0, ".rodata");
message:
    db("XIRASM PIE", 10);
message_end:
format_segment_end(image0, ".rodata");

```

```
// 绑定入口标号，并完成文件映像。  
const image: map = format_entry(image0, start)  
format_finish(image);
```

生成的文件大小为 316 字节，包含 3 个 **LOAD** 项：

- 代码使用可读、可执行的 **RX** 映射；
- 未初始化数据区使用可读、可写的 **RW** 映射，文件大小为零，内存大小为 64；
- 消息使用只读的 **R** 映射。

未初始化数据区不占用文件字节，因此它和只读 **LOAD** 在紧凑文件布局中使用同一个后续物理位置，但分别占用不同的虚拟页。每个 **LOAD** 仍满足 ELF 的要求：虚拟地址和文件偏移对页对齐值取模后结果相同。

程序先向未初始化数据区写入数值，再通过相对地址读取消息并输出。作为 **ET_DYN** 可执行文件加载后，它会正常结束。

位置无关可执行文件不一定需要动态链接器

上一个示例直接进行 Linux 系统调用，没有导入任何库符号，因此不需要：

- **PT_INTERP** ；
- **PT_DYNAMIC** ；
- 动态符号表或动态字符串表；
- 过程链接表（**PLT**）或全局偏移表（**GOT**）；
- 动态重定位表。

它是位置无关可执行文件，但不是动态链接的可执行文件。

必须区分这两个概念：位置无关模式描述映像可以放在什么地址，动态链接描述如何在运行时解析符号。一个文件可能只需要其中一种能力，也可能两种都需要，或者两种都不需要。当前常用格式接口分别实现本章介绍的两种组合，而不是把它们当作同一个选项。

声明过程链接表导入项

常用动态导入流程从固定地址的 ELF64 可执行文件方案开始：

```
format_elf64(format_elf_exec, segments)
```

为每个导入过程创建一项声明：

```
format_elfexe_import_plt(library, name, slot_label, plt_label)
```

各个参数分别表示：

- 由 `DT_NEEDED` 记录的共享库名称；
- 运行时链接器需要查找的外部符号名称；
- 本地全局偏移表槽位及对应过程链接表入口的标号；
- 调用指令使用的本地过程链接表入口标号。

把这些声明收集到列表中，再附加到格式方案：

```
const image1: map = format_elfexe_tables(image0, imports)
```

后续的 `format_begin`、段构建调用、`format_entry` 和 `format_finish` 都应使用 `image1`。导入声明属于这个更新后的格式方案。

调用 `libc` 的固定 ELF64 可执行文件

下面的可执行文件从 `libc.so.6` 导入 `getpid`。这个函数没有参数，因此示例可以集中展示 ELF 导入模型。

```

// 导入常用格式接口，并创建固定地址的 ELF64 可执行文件。
import("format/format.inc");

const image0: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable)
    )
)
// 声明共享库、外部函数以及本地 GOT/PLT 槽位和调用入口。
const imports: list = list.of(
    format_elfexe_import_plt(
        "libc.so.6",
        "getpid",
        "getpid_gotplt",
        "getpid_plt"
    )
)
// 导入表属于返回的新格式方案，后续构建流程都使用该方案。
const image1: map = format_elfexe_tables(image0, imports)
format_begin(image1);

format_segment_begin(image1, ".text");
start:
    // 通过本地过程链接表入口调用运行时解析的 getpid。
    call getpid_plt
    test eax, eax
    jle failed

    xor edi, edi
    jmp finish

failed:
    mov edi, 1

finish:
    // 根据 getpid 的返回值，以成功或失败状态结束进程。
    mov eax, 60
    syscall
format_segment_end(image1, ".text");

// 绑定入口并完成包含动态导入元数据的映像。
const image: map = format_entry(image1, start)
format_finish(image);

```

这个示例会生成 768 字节的 ELF64 可执行文件。程序启动时，动态链接器会解析 `getpid`，更新过程链接表使用的全局偏移表槽位，再通过 `getpid_plt` 转移控制。程序把大于零的进程标识符视为成功，并以状态 0 退出。

导入格式接口会生成什么

只要提供导入声明，常用格式接口就会生成：

- 指向 ELF64 运行时解释器的 `PT_INTERP` 项；
- `PT_DYNAMIC` 项；
- 动态符号表和动态字符串表；
- 每个不同共享库对应的一项依赖记录；
- 每个导入过程对应的过程链接表入口；
- 运行时解析器使用的全局偏移表和过程链接表状态；
- `R_X86_64_JUMP_SLOT` 重定位记录；
- 连接这些表的动态标签。

用户不需要计算符号索引、表地址、重定位表项或程序头位置。

生成的 `LOAD` 权限会把可执行内容与可写状态分开：

- 用户代码使用 `RX` 权限；
- 自动生成的过程链接表使用 `RX` 权限；
- 全局偏移表和动态元数据使用 `RW` 权限；
- 不会生成同时可写和可执行的 `LOAD`。

过程链接表与动态元数据在紧凑文件中物理相邻，但它们的虚拟地址会放在分别满足页内偏移关系的 `LOAD` 映射中。

调用过程链接表标号

调用 `format_elfexe_import_plt` 提供的本地过程链接表标号：

```
call getpid_plt
```

不要直接调用未定义的外部名称。XIRASM 需要这个本地过程链接表标号，才能让常用格式接口把调用指令连接到自动生成的运行时解析入口。

槽位标号表示过程链接表使用的全局偏移表项。当代码需要取得已经解析的函数指针时，可以使用它；普通的过程调用通常使用过程链接表标号。

导入过程仍需遵守平台应用二进制接口

格式接口负责构造 ELF 元数据，不会改变被导入函数的调用约定。

调用库过程之前，应当：

- 按照平台应用二进制接口的要求，把参数放入指定寄存器或栈位置；
- 保留该接口要求调用方保留的寄存器；
- 维持要求的栈对齐；
- 按照函数约定解释返回值。

即使加载器正确解析了符号，调用方使用错误的应用二进制接口仍会导致调用失败。

解释器路径属于文件内容

当前 ELF64 可执行文件导入流程会记录：

```
/lib64/ld-linux-x86-64.so.2
```

目标系统必须在这个路径提供兼容的运行时解释器。运行时目录布局不同的系统可能会在程序入口执行之前拒绝该文件。

解释器路径与 `DT_NEEDED` 中的共享库名称不是一回事。解释器负责读取可执行文件的动态元数据，再查找 `libc.so.6` 等依赖库。

当前常用接口的边界

当前常用格式接口支持：

- 不生成动态导入的 ELF64 位置无关可执行文件；
- 通过过程链接表进行动态导入的固定地址 ELF64 `EXEC` 文件。

它目前不允许把 `format_elfexe_tables` 附加到：

- ELF64 位置无关格式方案；
- ELF32 可执行文件；
- ELF32 位置无关格式方案。

这些组合会被拒绝，避免生成不完整的动态元数据。

不要把常用格式方案与按位宽划分或按表项逐行操作的辅助接口混合起来，以此绕过上述边界。需要直接控制格式细节时，应使用单独的高级格式指南所介绍的流程。

位置无关模式与动态导入的常见错误

在位置无关数据中保存绝对内部地址

在最小位置无关流程中，`dq(start)` 这样的原始指针不会自动根据加载地址调整。应使用相对访问，或者通过合适的高级流程生成所需的动态重定位。

在位置无关代码中使用绝对寻址

代码需要访问同一映像中的其他内容时，应使用 `[rel label]` 等相对寻址形式。

认为位置无关模式会添加库导入

`format_elf_pie` 只选择位置无关的映像放置方式，不会添加解释器、导入符号或过程链接表。

把导入表附加到位置无关方案

当前常用的可执行文件导入辅助接口要求使用 `format_elf_exec`。

继续使用原始格式方案

调用 `format_elfexe_tables` 后，后续构建流程必须使用它返回的新格式方案。

直接调用外部名称

应调用导入声明中指定的本地过程链接表标号。

为自动生成的元数据设置可写且可执行权限

常用格式接口会把可执行的过程链接表字节与可写的全局偏移表及动态状态分开。不要把它们合并到手工创建的可写且可执行段中。

忽略导入函数的应用二进制接口

ELF 符号解析不会检查参数、寄存器保留规则或栈对齐。

认为每个 Linux 系统都使用同一解释器

应确认目标系统提供生成文件中记录的解释器路径。

位置无关模式与动态导入实用规则

1. 使用 `format_elf64(format_elf_pie, segments)` 构造直接进行系统调用的位置无关可执行文件。
2. 位置无关代码及其内部数据引用应使用相对于当前指令位置的形式。
3. 除非运行时重定位会修正绝对标号值，否则不要把它写入数据字段。
4. 当前动态导入流程应使用 `format_elf64(format_elf_exec, segments)`。

5. 使用 `format_elfexe_import_plt` 声明导入过程。
6. 使用 `format_elfexe_tables` 把导入列表附加到格式方案。
7. 后续构建流程应继续使用更新后的格式方案。
8. 调用自动生成的本地过程链接表标号。
9. 遵守被导入函数的平台应用二进制接口。
10. 让格式接口生成并分隔过程链接表、全局偏移表、解释器记录和动态数据。
11. 现阶段应把位置无关模式和动态导入视为两种独立的受支持流程。

第 12 章将从运行时加载的映像转向 ELF32 和 ELF64 目标文件。链接器会使用这些目标文件中的命名节、符号，以及 `REL` 或 `RELA` 重定位记录。

第 12 章：ELF32 与 ELF64 目标文件

ELF 目标文件是交给链接器处理的可重定位输入文件，不能直接作为进程映像加载。

目标文件记录以下信息：

- 包含代码、初始化数据或预留空间的命名节；
- 由当前目标文件定义的公开符号；
- 需要由其他目标文件或库提供的外部符号；
- 最终值尚未确定的编码字段所需的重定位请求。

链接器会把这些信息组合成可执行文件或共享对象。常用格式接口会自行维护节索引、符号索引、表偏移和节头数量，使用者不需要手工计算这些字段。

创建 ELF 目标文件方案

使用 `format_elfobj32` 创建 `i386` 目标文件，使用 `format_elfobj64` 创建 `x86-64` 目标文件：

```
const object0: 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(  
      ".data",  
      format_data | format_readable | format_writeable  
    )  
  )  
)
```

描述项的顺序就是目标文件中各个用户节的顺序。常用格式接口还会生成普通构建流程所需的重定位节、符号表、字符串表、节名称表和栈权限说明节。

开始构造目标文件后，按照声明顺序写入各个节：

```
format_begin(object0)  
format_section_begin(object0, ".text")  
...  
format_section_end(object0, ".text")
```

ELF 目标文件没有映像基址、程序头、子系统或可执行文件入口字段。 `_start` 之类的符号在目标文件中只是一个公开定义，只有链接器把它选为最终可执行文件的入口时，它才会成为运行起点。

节描述链接器的输入内容

常用目标文件格式接口接受以下节用途：

- `format_code` 表示可执行代码；
- `format_data` 表示已经初始化的字节；
- `format_uninitialized_data` 表示未初始化数据区形式的预留空间。

未初始化数据节会生成 `SHT_NOBITS` 节。它的节头记录逻辑大小，但目标文件不会保存这些预留字节。

因此，未初始化数据节和后面一个在文件中有实际字节的节可以具有相同的文件偏移。两者仍然是不同的节：前者占用内存空间，后者占用文件字节。

应为每个节保留起始标号。公开符号和重定位会利用该标号计算相对于节起点的值：

```
format_section_begin(object0, ".bss")
bss_start:
scratch:
    reserve(64)
format_section_end(object0, ".bss")
```

公开符号描述当前文件中的定义

使用 `format_elfobj_public` 声明一个定义：

```
format_elfobj_public(
    "scratch",
    ".bss",
    bss_start,
    scratch,
    64,
    elfobj_stt_object
)
```

各个参数依次表示：

1. 链接器可见的符号名称；
2. 定义所在的节；
3. 该节起点处的标号；
4. 符号自身的标号；

5. 符号大小;
6. ELF 符号类型。

格式接口会计算节索引和符号相对于节起点的值。

符号类型可以选择:

- 可调用代码使用 `elfobj_stt_func` ;
- 数据或预留空间使用 `elfobj_stt_object` ;
- 没有更具体类型时使用 `elfobj_stt_notype` 。

外部符号描述尚未满足的需求

使用 `format_elfobj_extern` 声明一个未定义符号:

```
format_elfobj_extern("helper", elfobj_stt_func)
```

生成的符号使用 `SHN_UNDEF` 作为节索引。链接器必须从其他输入文件或库中找到名称匹配的定义。

符号名称按照字符串精确匹配。名称必须采用目标平台应用二进制接口以及参与链接的其他目标文件所要求的拼写。

重定位描述需要修补的编码字段

对外部符号的相对调用包含一个四字节位移, 而它的最终值取决于链接后目标符号的地址。

先写出操作码, 为位移写入占位值, 并在位移字段本身定义标号:

```
db(0xe8)
helper_disp:
dd(0)
```

然后声明重定位:

```
format_elfobj_reloc(
    ".text",
    text_start,
    helper_disp,
    "helper",
    elf_r_x86_64_plt32,
    0xfffffffffffffffffc
)
```

各个参数依次表示：

- 1. 编码字段所在的节；
- 2. 该节的起点；
- 3. 编码字段的地址；
- 4. 目标符号名称；
- 5. 与处理器架构对应的重定位类型；
- 6. 重定位加数。

格式接口会计算重定位字段相对于节起点的偏移，并按名称解析目标符号的索引。

加数 `-4` 用于计入四字节位移字段本身：

目标文件类别	相对调用重定位类型	加数的编码值
ELF32	<code>elf_r_386_pc32</code>	<code>0xffffffffc</code>
ELF64	<code>elf_r_x86_64_plt32</code>	<code>0xffffffffffffffffc</code>

重定位类型必须按照目标处理器架构的应用二进制接口选择。即使两个字段宽度相同，它们的重定位类型也不一定可以互换。

REL 与 RELA 使用不同方式保存加数

常用 ELF32 构建流程会生成 `SHT_REL` 重定位节。`REL` 把加数保存在编码字段中，因此格式接口会在收尾处理阶段把声明的加数写入四字节占位字段。

常用 ELF64 构建流程会生成 `SHT_RELA` 重定位节。`RELA` 把加数保存在重定位表项中，编码字段则继续保留为供链接器处理的占位值。

这种差异由生成的表在内部处理。两种构建流程都使用相同形式的 `format_elfobj_reloc` 声明。

关联符号与重定位

把各项声明收集到列表中：

```
const symbols: list = list.of(...)
const relocs: list = list.of(...)
```

再把列表关联到格式方案：

```
const object: map = format_elfobj_tables(object0, symbols, relocs)
format_finish(object)
```

`format_elfobj_tables` 会返回更新后的格式方案。调用 `format_finish` 时必须传入这个返回值。

完成目标文件时，格式接口会：

1. 按照被重定位的节对重定位项分组；
2. 按照符号名称解析重定位目标；
3. 生成 `REL` 或 `RELA` 节；
4. 生成各个节对应的局部符号；
5. 生成已经声明的公开符号和外部符号；
6. 生成 `.symtab`、`.strtab` 和 `.shstrtab`；
7. 生成大小为零且不要求可执行栈的说明节；
8. 写入最终节头表以及 ELF 文件头中的数量字段。

可由本机链接器处理的 ELF64 目标文件

下面的目标文件定义 `_start`、初始化数据和 64 字节未初始化数据空间，并调用名为 `helper` 的外部函数：

```

// 导入常用格式接口。
import("format/format.inc");

// 创建 ELF64 目标文件方案，并声明代码、未初始化数据和初始化数据节。
const object0: 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(
            ".data",
            format_data | format_readable | format_writeable
        )
    )
)

// 开始构造目标文件。
format_begin(object0);

// 写出入口代码，并为外部函数调用保留四字节相对位移。
format_section_begin(object0, ".text");
text_start:
_start:
    db(0xe8);
helper_disp:
    dd(0);
    mov edi, eax
    mov eax, 60
    syscall
_start_end:
format_section_end(object0, ".text");

// 预留 64 字节未初始化数据，不向目标文件写入对应载荷。
format_section_begin(object0, ".bss");
bss_start:
scratch:
    reserve(64);
format_section_end(object0, ".bss");

// 写出一个八字节初始化数据对象。
format_section_begin(object0, ".data");
data_start:
marker:
    dq(0x1122334455667788);
format_section_end(object0, ".data");

// 声明当前目标文件提供的三个公开符号和一个外部函数符号。

```

```

const symbols: list = list.of(
  format_elfobj_public(
    "_start",
    ".text",
    text_start,
    _start,
    _start_end - _start,
    elfobj_stt_func
  ),
  format_elfobj_public(
    "marker",
    ".data",
    data_start,
    marker,
    8,
    elfobj_stt_object
  ),
  format_elfobj_public(
    "scratch",
    ".bss",
    bss_start,
    scratch,
    64,
    elfobj_stt_object
  ),
  format_elfobj_extern("helper", elfobj_stt_func)
)

// 让链接器在 .text 节中修补 helper 调用的相对位移字段。
const relocs: list = list.of(
  format_elfobj_reloc(
    ".text",
    text_start,
    helper_disp,
    "helper",
    elf_r_x86_64_plt32,
    0xfffffffffffffffffc
  )
)

// 关联符号与重定位表，并使用更新后的方案完成目标文件。
const object: map = format_elfobj_tables(object0, symbols, relocs)
format_finish(object);

```

另一个目标文件可以提供 `helper`，并使用当前文件公开的未初始化数据符号：

```
// 引用另一个目标文件中定义的 64 字节未初始化数据对象。
extern volatile unsigned char scratch[64];

// 写入并读回最后一个字节，确认链接器为完整对象分配了空间。
int helper(void) {
    scratch[63] = 0x5a;
    return scratch[63] == 0x5a ? 0 : 1;
}
```

完成本机链接后，`_start` 会调用 `helper`，并使用它的返回值作为退出状态。提供方会写入并读回 `scratch` 的最后一个字节，从而证明链接器为这个定义分配了完整的 64 字节未初始化数据空间。

目标文件本身不保存 64 字节的未初始化数据载荷。它的 `.bss` 节类型为 `SHT_NOBITS`，大小为 64，并带有可写和运行时分配标志。

ELF32 使用相同的格式方案模型

32 位源代码需要更换格式方案构造函数、指令序列和重定位类型：

```

// 导入常用格式接口。
import("format/format.inc");

// 创建 ELF32 目标文件方案，并声明代码、初始化数据和未初始化数据节。
const object0: map = format_elfobj32(
    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
        )
    )
)

// 开始构造目标文件。
format_begin(object0);

// 写出 32 位入口代码，并为外部函数调用保留相对位移。
format_section_begin(object0, ".text");
text_start:
_start:
    db(0xe8);
helper_disp:
    dd(0);
    mov ebx, eax
    mov eax, 1
    db(0xcd, 0x80);
_start_end:
format_section_end(object0, ".text");

// 写出一个四字节初始化数据对象。
format_section_begin(object0, ".data");
data_start:
marker:
    dd(0x11223344);
format_section_end(object0, ".data");

// 预留 64 字节未初始化数据空间。
format_section_begin(object0, ".bss");
bss_start:
scratch:
    reserve(64);
format_section_end(object0, ".bss");

// 声明公开符号和需要在链接时解析的外部函数。

```

```

const symbols: list = list.of(
    format_elfobj_public(
        "_start",
        ".text",
        text_start,
        _start,
        _start_end - _start,
        elfobj_stt_func
    ),
    format_elfobj_public(
        "marker",
        ".data",
        data_start,
        marker,
        4,
        elfobj_stt_object
    ),
    format_elfobj_public(
        "scratch",
        ".bss",
        bss_start,
        scratch,
        64,
        elfobj_stt_object
    ),
    format_elfobj_extern("helper", elfobj_stt_func)
)

// 让链接器使用 32 位相对调用重定位修补位移字段。
const relocs: list = list.of(
    format_elfobj_reloc(
        ".text",
        text_start,
        helper_disp,
        "helper",
        elf_r_386_pc32,
        0xffffffffc
    )
)

// 关联表并完成 ELF32 目标文件。
const object: map = format_elfobj_tables(object0, symbols, relocs)
format_finish(object);

```

把同一份 C 定义编译为匹配的 32 位目标平台后，也可以由它提供 `helper`。

ELF32 的重定位节是 `.rel.text`，ELF64 的重定位节是 `.rela.text`。符号名称以及符号与节之间的关系保持不变。

自动生成的栈权限

常用格式接口会添加一个大小为零、没有可执行标志的 `.note.GNU-stack` 节。

它会告诉兼容的链接器，当前目标文件不要求进程栈具有执行权限。这是自动生成的元数据，不是用户声明的节描述项，也不会增加载荷字节。

需要可执行栈的代码不属于常用格式接口的安全模型，应改用高级目标文件构建流程。

ELF 目标文件常见错误

调用 `format_entry`

目标文件通过公开符号提供定义，不包含可执行文件入口字段。

传入节索引

应传入已经声明的节名称及其起始标号，由格式接口计算数值形式的节索引。

传入符号索引

应传入目标符号名称，由格式接口解析生成后的符号表索引。

重定位操作码而不是编码字段

相对调用应在操作码之后的四字节位移字段上定义标号。

使用错误的重定位类型

重定位类型与处理器架构和字段用途有关，必须与已经编码的指令以及目标平台应用二进制接口相匹配。

省略相对程序计数器加数

本章的普通相对调用示例使用 `-4`，因为位移是从四字节字段的末尾开始计算的。

把未初始化数据区当作文件数据

应在 `format_uninitialized_data` 节中预留空间，不要向其中写出初始化字节。

继续使用原始格式方案

调用 `format_finish` 时，应使用 `format_elfobj_tables` 返回的格式方案。

期待汇编器选择运行时入口

最终可执行文件的入口符号由链接器或链接驱动程序选择。

混用 ELF32 与 ELF64 输入文件

同一次链接中的所有目标文件必须使用彼此兼容的机器类型、文件类别、应用二进制接口和函数调用约定。

ELF 目标文件实用规则

1. 根据目标文件类别选择 `format_elfobj32` 或 `format_elfobj64`。
2. 在调用 `format_begin` 前声明所有用户节。
3. 为符号或重定位会引用的每个节保留起始标号。
4. 使用 `format_uninitialized_data` 和 `reserve` 建立未初始化数据区。
5. 使用 `format_elfobj_public` 描述当前文件提供的定义。
6. 使用 `format_elfobj_extern` 描述尚未解析的外部需求。
7. 在链接器需要修补的准确编码字段上定义标号。
8. 按照目标平台的应用二进制接口选择重定位类型。
9. 对本章所示的相对调用传入用补码表示的 `-4` 加数。
10. 使用 `format_elfobj_tables` 关联符号与重定位。
11. 使用返回的格式方案完成目标文件。
12. 让格式接口生成节索引、符号索引、各类表和不可执行栈说明节。
13. 只链接使用兼容处理器架构和应用二进制接口的目标文件。

第 13 章将构建 ELF64 共享对象。共享对象中的动态导出符号和导入符号由运行时加载器解析，不只依赖静态链接。

第 13 章：ELF64 共享对象

ELF 共享对象是一种 `ET_DYN` 映像，用于加载到另一个进程中。它通常没有自己的进程入口，而是发布动态符号，供程序或其他共享对象在运行时解析。

常用的 ELF64 共享对象格式接口可以：

- 通过动态符号表发布函数或数据；
- 通过自动生成的过程链接表和全局偏移表状态，从指定共享库导入过程；
- 描述多个权限彼此独立的用户可加载段；
- 表示只预留空间的 BSS，而不在文件中保存填零内容；
- 生成运行时加载器需要的库标识名、动态数据表、散列表、重定位、节头和程序头。

本章将构造一个共享对象：它导入 `puts`，导出一个可调用函数，使用 BSS 保存状态，并且可以通过 `dlopen` 加载。

创建共享对象格式方案

使用下面的接口创建常用的 ELF64 共享对象格式方案：

```
format_elf64_so(soname, segments)
```

库标识名会记录在动态数据表中，用来标识这个共享库：

```
// 导入常用格式接口，使本示例可以单独汇编。
import("format/format.inc");

// 创建 ELF64 共享对象，并按代码、BSS、只读数据和可写数据的顺序声明段。
const image0: map = format_elf64_so(
    "libxirasm_ch13.so",
    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_segment(".data", format_load | format_readable | format_writeable)
    )
)
```

描述项的顺序决定用户 `LOAD` 项和用户节头的顺序。权限模型与可执行文件相同：

- 代码可读、可执行；
- BSS 和已经初始化的可修改数据可读、可写；
- 常量只需可读；

- 用户段不应同时具有写入和执行权限。

当前常用共享对象构建流程只支持 ELF64。

声明导出符号

使用下面的接口声明导出符号：

```
format_elfso_export(target_label, export_name, segment_name, symbol_size)
```

四个参数依次表示：

1. 源代码定义的目标标号；
2. 动态符号表中发布的名称；
3. 包含该符号的已声明用户段；
4. 记录给运行时工具和使用方的符号大小。

例如：

```
// 导入常用格式接口，使本示例可以单独汇编。
import("format/format.inc");

// 发布代码段中的函数，并记录它在动态符号表中的名称和大小。
const exports: list = list.of(
    format_elfso_export(
        "xirasm_call_puts",
        "xirasm_call_puts",
        ".text",
        46
    )
)
```

目标标号和发布名称可以不同。段名称必须与格式方案中的描述项一致，符号大小必须大于零。

当前常用格式接口要求至少声明一个导出项。它会在收尾处理阶段解析目标标号和自动生成的节索引，用户不需要计算动态符号表项或节索引。

声明过程链接表导入项

使用下面的接口声明导入过程：

```
format_elfso_import_plt(library, name, slot_label, plt_label)
```

例如：

```
// 导入常用格式接口，使本示例可以单独汇编。
import("format/format.inc");

// 从 libc.so.6 导入 puts，并为全局偏移表槽位和过程链接表入口指定本地标号。
const imports: list = list.of(
    format_elfso_import_plt(
        "libc.so.6",
        "puts",
        "puts_gotplt",
        "puts_plt"
    )
)
```

共享库名称会形成一项 `DT_NEEDED` 记录。外部符号名称会形成一个未定义的动态符号。两个本地标号分别表示自动生成的全局偏移表和过程链接表槽位，以及过程链接表入口。

普通调用应使用过程链接表标号：

```
call puts_plt
```

运行时加载器会解析 `puts`，并把函数地址写入自动生成的全局偏移表和过程链接表槽位。

开始映像前附加动态数据表

使用下面的接口附加导出列表和导入列表：

```
format_elfso_tables(plan, exports, imports)
```

后续完整构建流程都应使用返回的新格式方案：

```
// 导入常用格式接口，并补齐本示例依赖的格式方案与声明。
import("format/format.inc");

const image0: map = format_elf64_so(
    "libxirasm_ch13.so",
    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_segment(".data", format_load | format_readable | format_writeable)
    )
)
const exports: list = list.of(
    format_elfso_export(
        "xirasm_call_puts",
        "xirasm_call_puts",
        ".text",
        46
    )
)
const imports: list = list.of(
    format_elfso_import_plt(
        "libc.so.6",
        "puts",
        "puts_gotplt",
        "puts_plt"
    )
)

// 把导出和导入声明附加到格式方案，再开始构造共享对象映像。
const image: map = format_elfso_tables(image0, exports, imports)
format_begin(image);
```

必须在 `format_begin` 之前附加这些列表。导入项会影响自动生成的程序头数量，还需要分别建立可执行的过程链接表映射和可写的元数据映射。

只导出、不导入的共享对象应传入空导入列表：

```
format_elfso_tables(image0, exports, list.new())
```

构造多段共享对象

下面的源代码从 `libc.so.6` 导入 `puts`，导出 `xirasm_call_puts`，在 BSS 中保存调用次数，并把代码、常量、BSS、初始化数据、自动生成的过程链接表和动态元数据放入权限合适的映射中：

```

// 导入常用格式接口。
import("format/format.inc");

// 创建共享对象格式方案，并为不同用途的内容声明独立段。
const image0: map = format_elf64_so(
    "libxirasm_ch13.so",
    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_segment(".data", format_load | format_readable | format_writeable)
    )
)

// 发布可供加载方解析和调用的函数。
const exports: list = list.of(
    format_elfso_export(
        "xirasm_call_puts",
        "xirasm_call_puts",
        ".text",
        46
    )
)

// 声明外部过程，并指定本地全局偏移表槽位和过程链接表入口标号。
const imports: list = list.of(
    format_elfso_import_plt("libc.so.6", "puts", "puts_gotplt", "puts_plt")
)

// 导入和导出会改变自动生成的运行时元数据，因此先附加数据表再开始映像。
const image: map = format_elfso_tables(image0, exports, imports)
format_begin(image);

// 导出函数更新 BSS 中的调用次数，通过过程链接表调用 puts，并返回最新计数。
format_segment_begin(image, ".text");
xirasm_call_puts:
    mov rax, [rel call_count]
    add rax, 1
    mov [rel call_count], rax
    lea rdi, [rel message_text]
    sub rsp, 8
    call puts_plt
    add rsp, 8
    mov rax, [rel call_count]
    ret
format_segment_end(image, ".text");

// 只预留运行时内存，用于保存跨调用持续存在的计数。
format_segment_begin(image, ".bss");
call_count:
    reserve(64);
format_segment_end(image, ".bss");

// 保存传给 puts 的零结尾只读文本。
format_segment_begin(image, ".rodata");

```

```
message_text:
    db("XIRASM shared object", 0);
format_segment_end(image, ".rodata");

// 写入共享对象自带的可修改初始化数据。
format_segment_begin(image, ".data");
library_state:
    dq(0x1122334455667788);
format_segment_end(image, ".data");

// 生成动态数据表、程序头、节头及其最终地址和大小。
format_finish(image);
```

这个函数遵守 x86-64 的 **System V** 调用约定。它在调用 **puts** 前保持所需的栈对齐，通过相对于指令指针的寻址访问内部数据，并在 **rax** 中返回更新后的调用次数。

共享对象没有可执行文件入口。**format_finish** 会在用户段之后写出运行时元数据，并确定所有自动生成内容的地址、大小、索引和数量。

加载并调用导出函数

使用 **C** 编写的加载程序可以打开文件并解析导出的函数：

```

#include <dlfcn.h>

// 导出函数没有参数，并以 long 返回当前调用次数。
typedef long (*entry_fn)(void);

int main(void) {
    // 立即解析共享对象需要的动态符号。
    void *handle = dlopen("./libxirasm_ch13.so", RTLD_NOW);
    if (handle == 0) {
        return 1;
    }

    // 按发布名称查找导出函数，而不是使用源代码标号或符号索引。
    entry_fn entry = (entry_fn)dlsym(handle, "xirasm_call_puts");
    if (entry == 0) {
        dlclose(handle);
        return 2;
    }

    // 连续调用两次，验证 BSS 状态在已加载映像中持续存在。
    const long first = entry();
    const long second = entry();
    dlclose(handle);
    return first == 1 && second == 2 ? 0 : 3;
}

```

第一次调用会输出消息并返回 1。第二次调用会返回 2，证明 BSS 状态在已加载映像中持续存在。

`dlsym` 使用发布的导出名称查找函数，不使用源代码标号、节名称或数字符号索引。

BSS 只占用内存，不包含文件内容

`.bss` 段只包含：

```

// 为调用计数预留 64 字节运行时空间，不在文件中写入初始字节。
reserve(64);

```

它的程序头中文件大小为零，内存大小为 64。对应节头使用 `SHT_NOBITS`，大小同样为 64。后面的 `.rodata` 段可以从相同的紧凑文件偏移开始，因为 BSS 占用的是虚拟内存，而不是文件字节。

加载器会把 BSS 映射到独立的可写虚拟内存页，并把这段内存初始化为零。

常用 BSS 段应只包含预留空间。如果初始化字节和预留空间必须共用一个逻辑区域，应使用不同的常用段，或者使用专门的高级布局。

自动生成的运行时元数据

对于同时包含导入和导出的构建流程，格式接口会生成：

- 保存导入和导出名称的 `.dynsym` 与 `.dynstr`；
- `System V` ELF 散列表；
- 用于导入过程的 `.plt` 节及全局偏移表和过程链接表状态；
- `R_X86_64_JUMP_SLOT` 重定位记录；
- 包含 `DT_SONAME`、`DT_NEEDED` 和各数据表连接信息的 `.dynamic` 节；
- 节名称字符串表和最终节头表；
- `LOAD` 与 `DYNAMIC` 程序头。

用户只须提供名称、权限、标号和符号大小。格式接口会推导出程序头表项、节索引、符号索引、字符串偏移、重定位表项、文件偏移、虚拟地址和数据表大小。

可执行内容与可写状态保持分离

存在导入项时，自动生成的内容会放入另外两个 `LOAD` 项：

- 过程链接表可读、可执行；
- 全局偏移表、符号、字符串、重定位、动态数据和节元数据可读、可写。

任何 `LOAD` 都不会同时具有写入和执行权限。

实际文件仍然保持紧凑。不同虚拟内存页可以分别实施权限，而不必在文件中填充整页零字节；每个 `LOAD` 的文件偏移和虚拟地址都保持 ELF 要求的页内偏移同余关系。

共享对象必须遵守使用方的应用二进制接口

动态符号解析不会定义函数的调用约定。每个导出过程和导入过程都必须遵守调用方或被调用方要求的应用二进制接口。

对于 x86-64 的 `System V` 函数：

- 从规定的寄存器接收参数；
- 保留应用二进制接口要求被调用方保留的寄存器；
- 在调用其他函数前保持要求的栈对齐；
- 在规定的寄存器中返回结果；
- 使用兼容的数据大小和结构布局。

导入函数也必须遵守同一规则。即使过程链接表重定位完全正确，也无法修复错误的调用指令序列。

当前常用接口边界

当前常用共享对象格式接口提供：

- ELF64 共享对象；
- 至少一个导出的动态符号；
- 可选的过程链接表函数导入；
- 多个用户 `LOAD` 段；
- 只包含预留空间的 BSS；
- 自动生成的库标识名、散列表、符号、字符串、重定位、过程链接表、全局偏移表、动态数据和节头状态。

它目前不提供常用的 ELF32 共享对象构造函数，也不提供符号版本表、线程局部存储、构造函数数组、`GNU` 散列表或任意自定义动态标签。

不要把按位宽划分的兼容接口或按表项逐行操作的辅助接口混入常用格式方案，以此绕过这些边界。特殊布局应使用单独的高级格式指南。

共享对象常见错误

调用 `format_entry`

共享对象发布动态符号。常用构建流程不使用可执行文件入口。

没有声明任何导出项

当前常用格式接口要求至少声明一个导出项。

在 `format_begin` 之后附加数据表

导入项会改变自动生成的程序头布局。开始映像之前必须附加导出列表和导入列表。

直接调用外部名称

应调用 `format_elfso_import_plt` 提供的本地过程链接表标号。

为导出项声明错误的段

导出项中声明的段必须是实际包含目标标号的用户段。

记录错误的符号大小

导出大小应与实际写出的函数或对象保持一致。

向 BSS 写入字节

只包含预留空间的 BSS 段会使用 `SHT_NOBITS`。初始化字节应放入文件中有实际字节的数据段。

让一个段同时具有写入和执行权限

代码与可修改状态应保持分离。常用格式接口已经把自动生成的过程链接表字节和可写元数据分开。

忽略平台应用二进制接口

动态调用的两端必须对寄存器用法、栈对齐、数据布局和返回值保持一致。

认为库标识名可以找到文件

库标识名只在文件已经找到后标识这个共享库。搜索路径、加载器配置和调用程序仍然决定如何定位文件。

ELF64 共享对象实用规则

1. 使用 `format_elf64_so` 创建格式方案。
2. 为文件提供非空的库标识名。
3. 使用满足实际需求的最小权限声明用户段。
4. 把只包含预留空间的 BSS 放入独立的可写段。
5. 使用 `format_elfso_export` 声明每个发布符号。
6. 使用 `format_elfso_import_plt` 声明导入过程。
7. 在 `format_begin` 之前使用 `format_elfso_tables` 附加导出列表和导入列表。
8. 后续每个构建流程调用都使用返回的新格式方案。
9. 通过自动生成的本地过程链接表标号调用导入过程。
10. 每个导入过程和导出过程都应遵守平台应用二进制接口。
11. 让格式接口推导动态数据表、索引、偏移和程序头。
12. 让可执行映射与可写映射保持分离。
13. 通过目标平台的动态加载接口加载共享对象并解析导出名称。

至此，常用可执行文件格式指南已经介绍完毕：用户可以通过 `format.inc` 格式接口构造 PE 可执行文件和动态链接库、COFF 目标文件、ELF 可执行文件和位置无关可执行文件、ELF 目标文件，以及 ELF64 共享对象。

直接导入格式专用包含文件、使用兼容接口、显式构造数据表项和生成特殊元数据，都属于高级接口。这些内容会在单独的高级格式指南中介绍，不与常用构建流程混在一起。