

中文文档合集

XIRASM 中文文档合集

语言指南、可执行文件格式指南与 API 参考

目录

1. XIRASM 语言指南
2. 第 1 章：认识 XIRASM
3. 第 2 章：值与绑定
4. 第 3 章：表达式
5. 第 4 章：流程控制
6. 第 5 章：函数与作用域
7. 第 6 章：集合与文本
8. 第 7 章：词法单元与模式匹配
9. 第 8 章：目标平台、处理器指令与标号
10. 第 9 章：数据与二进制布局
11. 第 10 章：模块与文件
12. 第 11 章：输出区域与虚拟数据
13. 第 12 章：收尾处理
14. 第 13 章：裸二进制文件与自定义二进制格式
15. 第 14 章：可执行文件与目标文件格式
16. 第 15 章：诊断信息与实用约定
17. XIRASM 可执行文件格式指南
18. 第 1 章：认识 XIRASM 可执行文件格式
19. 第 2 章：格式方案与构建流程
20. 第 3 章：节、段、权限与未初始化数据区
21. 第 4 章：地址、符号与重定位
22. 第 5 章：PE32 与 PE64 可执行文件
23. 第 6 章：导入 Windows 接口函数
24. 第 7 章：`DLL` 导出与基址重定位
25. 第 8 章：PE 资源与校验和
26. 第 9 章：COFF32 与 COFF64 目标文件
27. 第 10 章：ELF32 与 ELF64 可执行文件
28. 第 11 章：位置无关可执行文件与动态导入
29. 第 12 章：ELF32 与 ELF64 目标文件
30. 第 13 章：ELF64 共享对象
31. XIRASM 语言 API 参考
32. 第 1 章：源码、绑定与作用域
33. 第 2 章：函数与流程控制
34. 第 3 章：结构体、联合体与复合值
35. 第 4 章：收尾处理语法形式
36. 第 5 章：模块与诊断信息
37. 第 6 章：目标平台、处理器指令与符号
38. 第 7 章：数据写出、预留空间与对齐
39. 第 8 章：主输出区域、输出区与文件游标
40. 第 9 章：读取、写入与最终区域信息
41. 第 10 章：文本、转换与标号名称
42. 第 11 章：字节序列
43. 第 12 章：列表与映射
44. 第 13 章：文件与结构化数据
45. 第 14 章：词法单元与模式匹配

XIRASM 语言指南

XIRASM 是一种汇编器，用来直接编写机器码并构造二进制文件。它的源语言把自然的处理器指令与结构化的编译期语言结合在一起：指令仍然保持汇编代码的可读形式，而值、函数、循环、数据布局 and 输出格式则使用清晰的语言结构来表达。

本指南按照新用户通常的学习顺序介绍 XIRASM。内容从简单的裸二进制文件开始，逐步讲解编译期语言、标号、二进制布局、输出区域、收尾处理以及可执行文件格式。完整的函数签名和速查表由接口参考提供；PE、COFF 和 ELF 的专题教程见可执行文件格式指南。

指南结构

第一部分：语言基础

1. **认识 XIRASM** - 汇编第一个 x86-64 程序。 - 理解处理器指令与编译期代码如何配合。 - 掌握基本源代码规则。
2. **值与绑定** - 声明常量和可变的局部绑定。 - 使用整数、布尔值、字符串和字节。 - 理解作用域和赋值。
3. **表达式** - 使用算术、比较、逻辑、位运算和字段表达式。 - 调用函数并组合编译期计算。
4. **流程控制** - 使用 `if` 选择需要生成的内容。 - 使用 `while` 和 `for` 重复执行操作。 - 根据编译期条件生成数据和指令。
5. **函数与作用域** - 编写不返回值的过程和返回值的函数。 - 使用参数、返回类型、局部绑定和嵌套作用域。
6. **集合与文本** - 创建和处理列表、映射、字符串及字节序列。 - 使用集合描述表格和需要生成的输出内容。
7. **词法单元与模式匹配** - 检查词法单元序列。 - 匹配源代码模式，并据此构建更高层的源代码处理工具。

第二部分：汇编器模型

8. **目标平台、处理器指令与标号** - 选择指令集和指令模式。 - 定义标号，并在指令中使用编译期值。 - 理解地址引用与回填。
9. **数据与二进制布局** - 写出整数、字符串和字节。 - 预留空间并控制对齐方式。 - 定义具有精确二进制布局的结构体和联合体。
10. **模块与文件**
 - 通过包含和导入复用源代码。
 - 从普通文件以及 JSON、TOML 格式中读取源数据。
11. **输出区域与虚拟数据**
 - 区分逻辑地址和文件中的实际位置。
 - 构建输出区域、预留一段空间，并使用虚拟数据作为临时工作区。

- 读取并修改已经生成的字节。

12. 收尾处理

- 等布局稳定后再执行检查和回填。
- 计算校验和与最终字段值。
- 理解哪些操作不得改变最终布局。

第三部分：构建程序

13. 裸二进制文件与自定义格式

- 构建紧凑的裸二进制文件和应用专用文件格式。

14. 可执行文件与目标文件格式

- 理解格式接口负责完成哪些工作。
- 无需手工计算文件头，即可构建一个最小可执行文件。
- 通过可执行文件格式指南继续学习 PE、COFF、ELF、导入、导出、BSS（未初始化数据区）和重定位。

15. 诊断信息与实用约定

- 报告无效输入，并检查布局是否符合预期。
- 组织可以复用的汇编项目。
- 在掌握本指南后继续查阅接口参考和可执行文件格式指南。

第 1 章：认识 XIRASM

一种源语言，两种写法

一个 XIRASM 源文件可以同时包含两种形式的代码：

- **处理器指令**描述所选处理器需要执行的操作。
- **编译期代码**负责计算各种值、生成输出内容，并控制汇编过程。

两者属于同一种源语言。编译期值可以直接用在指令中，布局相关接口可以引用标号，函数也可以同时生成数据和处理器指令。

下面的程序把这两种写法组合在一起：

```
// 选择 64 位 x86 指令编码。
x86.use64();

// 在汇编期间计算常量，随后把它用作指令的立即数。
const answer: u32 = 40 + 2

entry:
    // 把编译期得到的数值写入 eax，然后返回。
    mov eax, answer
    ret
```

可以按照源代码从上到下理解：

- `x86.use64();` 选择 64 位 x86 指令编码。
- `const answer: u32 = 40 + 2` 创建一个编译期常量。
- `entry:` 在当前逻辑地址定义一个标号。
- `mov eax, answer` 是普通的 x86 指令，编译期值 `answer` 会成为它的立即数操作数。
- `ret` 是另一条普通的 x86 指令。

`x86.use64();` 这样的调用使用结构化的编译期语言语法。`mov eax, answer` 这样的处理器指令仍然采用自然的汇编写法，不需要写成函数调用。

汇编第一个程序

把源代码保存为 `hello.xir`，然后运行：

```
xirasm hello.xir -o hello.bin --target x86-64
```

输出文件包含六个字节：

```
b8 2a 00 00 00 c3
```

前五个字节是 `mov eax, 42` 的编码，最后一个字节是 `ret` 的编码。标号本身不会增加任何字节，它只记录一个地址，供其他指令和编译期接口引用。

直接汇编时，XIRASM 默认生成裸二进制文件。这类文件就是源代码产生的原始字节序列，不包含操作系统要求的文件头。后续章节会通过一个简短示例介绍格式接口；完整的 PE、COFF 和 ELF 程序将在可执行文件格式指南中讲解。

编译期代码可以生成输出

编译期代码在 XIRASM 构造输出内容时运行。它不会在最终生成的程序中继续执行。

例如，下面的循环会写出四个字节：

```
// 在汇编期间循环四次，每次写出一个字节。  
const count: u8 = 4  
  
for value in range(0, count) {  
    db(value);  
}
```

生成的字节是：

```
00 01 02 03
```

`range(0, count)` 生成从零开始、到 `count` 之前结束的数值序列。`for` 循环在汇编期间运行，每次执行 `db(value);` 都会写出一个字节。

这一区别非常重要：

任务	写法
手写处理器指令	使用自然的汇编写法
常量和计算结果	使用编译期表达式
重复或按条件生成内容	使用编译期流程控制
二进制数据	调用 <code>db</code> 、 <code>dw</code> 、 <code>dd</code> 和 <code>dq</code> 等数据接口
可复用的生成逻辑	编写编译期函数
PE、COFF 或 ELF 输出	调用对应的格式接口

基本源代码规则

掌握下面这些规则，就足以阅读本指南第一部分中的示例：

- `//` 之后直到行末的内容都是注释。
- 标号以 `:` 结尾。
- 处理器指令行末不写分号。
- 函数调用和接口调用以 `;` 结尾。
- `return` 语句以 `;` 结尾。
- `const` 和 `let` 声明末尾不写分号。
- 代码块使用 `{` 和 `}`。
- 函数调用和复合值可以跨越多行，前提是相应的圆括号或花括号尚未闭合。

例如：

```
// 定义一个在汇编期间调用的数据生成函数。
fn emit_marker(value: u8) {
    // 先写出固定标记，再写出调用者传入的数值。
    db(0x7f, value);
}

const marker: u8 = 3
emit_marker(marker);

start:
    // 在处理器指令中直接使用编译期常量。
    mov eax, marker
    ret
```

函数定义和函数调用属于编译期代码。标号及其后的两条指令属于汇编代码。两种形式共同生成同一个最终输出文件。

一个实用的理解方法

阅读或编写 XIRASM 源代码时，可以先判断每一行的作用：

1. 如果它是处理器指令，就使用自然的汇编写法。
2. 如果它需要在汇编期间进行计算或选择内容，就使用编译期语言。
3. 如果它需要写出字节或预留空间，就使用数据和输出接口。
4. 如果它描述的是一种完整文件格式，就使用对应的格式接口，不要手工计算文件头中的各项数据。

下一章将进一步介绍编译期语言，包括常量、可变绑定、值类型、赋值和作用域。

[返回语言指南](#)

第 2 章：值与绑定

值存在于汇编期间

XIRASM 中的值都是编译期值。它们用于构造输出内容，但绑定本身不会写出字节，也不会预留内存。

```
// 这里只创建编译期值，不会向输出写入任何字节。  
const magic: u16 = 0x5a4d
```

这条声明把值绑定到名称 `magic`，但不会把 `4d 5a` 写入输出内容。要写出这个值，需要把它传给数据接口：

```
// 保存 16 位魔数，再通过数据接口按对应宽度写出。  
const magic: u16 = 0x5a4d  
dw(magic);
```

同样的规则也适用于可变值、字符串、字节序列、集合以及函数返回值。只有当指令、数据调用、布局操作或格式接口使用这些值时，它们才会成为输出内容的一部分。

常量

常量把名称绑定到一个不可重新赋值的值：

```
// 页面大小和文件头大小在整个汇编过程中保持不变。  
const page_size: u64 = 4096  
const header_size = 64  
  
// 布尔常量可直接用于后续的编译期条件。  
const enabled: bool = true
```

常量声明的一般形式是：

```
const name = expression  
const name: type = expression
```

如果 XIRASM 能从初始值判断类型，类型标注可以省略。当位宽或值的类别属于二进制约定的一部分时，应当写出明确类型。

常量声明末尾不写分号。对常量重新赋值会报错：

```
const value = 1
value = 2
```

除非源代码在汇编期间确实需要更新绑定，否则优先使用 `const`。

可变绑定

编译期计算需要保存和更新状态时，使用 `let`：

```
// 逐步增加偏移量，最后写出计算结果。
let offset: u32 = 0
offset = offset + 16
offset = offset + 4

dd(offset);
```

这段代码会写出：

```
14 00 00 00
```

声明和两次赋值都发生在汇编期间，不会在最终程序中创建名为 `offset` 的运行时变量。

可变绑定与赋值的形式是：

```
let name = expression
let name: type = expression
name = expression
```

与声明一样，赋值语句末尾不写分号。作为独立语句使用的函数调用仍然需要分号：

```
// 更新编译期绑定，然后把最终值写成一个字节。
let value = 1
value = value + 1
db(value);
```

类型标注与类型推断

绑定可以明确写出自己的类型：

```
// 明确的类型让字段宽度和文本类别直接体现在源代码中。
const signature: u16 = 0x5a4d
const title: string = "XIRASM"
const marker: bytes = b"OK"
const enabled: bool = true
```

也可以让 XIRASM 根据初始值推断类型：

```
// 这些值的类型都可以由字面量直接确定。
const count = 4
const name = "payload"
const raw = b"DATA"
```

普通源代码中常见的值类型包括：

类型	用途	示例
<code>integer</code>	通用编译期整数	<code>42</code>
<code>u8</code>	8 位无符号值	<code>0xff</code>
<code>u16</code>	16 位无符号值	<code>0x5a4d</code>
<code>u32</code>	32 位无符号值	<code>0x401000</code>
<code>u64</code>	64 位无符号值	<code>0x1400000000</code>
<code>f32</code>	IEEE-754 32 位浮点值	<code>f32(1.5)</code>
<code>f64</code>	IEEE-754 64 位浮点值	<code>1.5</code>
<code>bool</code>	编译期条件	<code>true</code>
<code>string</code>	编译期文本	<code>"kernel"</code>
<code>bytes</code>	明确的字节序列	<code>b"PE"</code>

列表、映射、结构体、联合以及类型值将在后续章节介绍。

指定位宽的整数类型适合表示二进制字段，也适合表达函数接口对参数位宽的要求。如果辅助函数只要求参数是整数，而不要求某个特定编码宽度，可以使用通用的 `integer` 类型。最终写出这个值的接口仍然负责决定输出多少个字节。

带小数部分或指数的十进制字面量属于 `f64`。使用 `f32(value)` 显式转换为有限的 `f32`，使用 `f64(value)` 把 `f32` 扩宽为 `f64`。整数与浮点值之间不会发生隐式转换。

字符串与字节序列

字符串和字节序列属于不同的值类别：

```
// 节名是文本，而文件签名表示精确的两个字节。  
const section_name: string = ".text"  
const signature: bytes = b"MZ"
```

文本名称、路径、生成的处理器指令文本，以及要求文本参数的接口，应使用字符串。需要表示精确的字节序列时，应使用 `bytes`。

部分输出接口同时接受这两种值：

```
// 字符串和显式字节序列会按参数顺序连续写出。  
db("AB", b"CD");
```

这段代码会写出四个字节：

```
41 42 43 44
```

另一些接口会有意限定为其中一种类别。在绑定中保留这一区别，可以让接口约定直接体现在源代码里。

块作用域

绑定属于它声明时所在的作用域。代码块会创建一层嵌套作用域：

```
// 外层常量在内部代码块结束后仍然可见。  
const value = 1  
  
{  
    // 内层绑定只在这个代码块中遮蔽外层同名常量。  
    let value = 2  
    db(value);  
}  
  
// 离开代码块后，再次使用外层的 value。  
db(value);
```

输出内容是：

```
02 01
```

内层 `value` 只在代码块中遮蔽外层常量。代码块结束后，外层 `value` 会重新可见。

函数参数和局部绑定同样属于函数作用域。嵌套代码块可以引入短期使用的名称，而不必改动外层绑定。

当嵌套操作确实有一个自然的局部名称时，名称遮蔽很有用；如果两个含义会让读者难以区分，就应避免使用相同名称。

在 `const` 与 `let` 之间选择

选择能准确表达计算过程、同时限制最严格的绑定形式：

情况	优先选择
固定的选项、大小、名称或计算结果	<code>const</code>
由语言管理的循环计数值	循环绑定
持续更新的偏移、校验和或累加值	<code>let</code>
只在一个小代码块中变化的值	代码块局部的 <code>let</code>
位宽属于文件格式约定的值	明确的类型标注
类型显而易见的临时值	类型推断

格式描述和生成汇编代码中的大多数绑定都应保持为常量。可变状态局限在较小范围、存续时间较短并且只服务于一项计算时，含义最清楚。

下一章将介绍产生这些值的表达式，包括算术、比较、布尔逻辑、位运算、字段访问和函数调用。

[返回语言指南](#)

第 3 章：表达式

表达式产生编译期值

XIRASM 在汇编源代码时计算表达式并得到值。声明、赋值、函数参数、返回语句、条件、断言、指令操作数和数据调用中都可以使用表达式。

```
// 根据表项数量和单项大小，在汇编期间计算整张表的字节数。  
const entry_count = 3  
const bytes_per_entry = 8  
const table_size = entry_count * bytes_per_entry  
  
dd(table_size);
```

这段代码会写出 32 位值 **24**。乘法在汇编期间完成，输出内容只包含计算结果。

算术运算

整数表达式支持以下运算符：

运算符	含义
+	加法
-	减法
*	乘法
/	整数除法
%	取余

```
// 同时演示默认优先级、括号分组、整数除法和取余。  
const total = 2 + 3 * 4  
const grouped = (2 + 3) * 4  
const quotient = 17 / 5  
const remainder = 17 % 5  
  
dd(total, grouped, quotient, remainder);
```

写出的四个值依次是 **14**、**20**、**3** 和 **2**。

加法、减法和乘法都会检查结果。超出受支持整数范围的结果会被拒绝，而不会悄悄改变布局。除数为零的除法和取余同样会报错。

一元 `+` 保持整数值不变。一元 `-` 产生对应的 64 位二进制补码值：

```
// -1 的 64 位二进制补码中，每一位都为 1。  
const all_bits = -1  
dq(all_bits);
```

这段代码会写出八个 `ff` 字节。

位运算与移位

位运算表达式适合处理权限、掩码、指令字段和文件格式标志：

运算符	含义
<code>&</code>	按位与
<code> </code>	按位或
<code>^</code>	按位异或
<code>~</code>	按位取反
<code><<</code>	左移
<code>>></code>	右移

```
// 每一位分别表示一种权限。  
const readable = 1 << 0  
const writeable = 1 << 1  
const executable = 1 << 2  
  
// 组合读取与执行权限，再分别检查两个权限位。  
const permissions = readable | executable  
const can_execute = (permissions & executable) != 0  
const can_write = (permissions & writeable) != 0  
  
assert(can_execute);  
assert(!can_write);  
db(permissions);
```

写出的字节是 `05`。

检查掩码时应使用括号。这样既能直接表明要执行的操作，也能避免依赖位运算符与比较运算符之间的相对优先级。

比较与相等判断

表达式可以比较值：

运算符	含义
<code>==</code>	等于
<code>!=</code>	不等于
<code><</code>	小于
<code><=</code>	小于或等于
<code>></code>	大于
<code>>=</code>	大于或等于

大小比较用于整数：

```
// 载荷必须大于零，并且不能超过允许的最大大小。  
const payload_size = 96  
const maximum_size = 128  
const fits = payload_size > 0 && payload_size <= maximum_size  
  
assert(fits);
```

相等与不等判断也适用于类别相容的非整数值：

```
// 分别核对字符串名称和精确的字节签名。  
const format_name = "raw"  
const signature = b"OK"  
  
assert(format_name == "raw");  
assert(signature == b"OK");
```

不同且不相容的值类别不会被隐式转换，而是会被拒绝。

布尔逻辑与短路求值

布尔表达式使用以下运算符：

运算符	含义
<code>!</code>	逻辑非
<code>&&</code>	逻辑与
<code> </code>	逻辑或

`&&` 和 `||` 会进行短路求值：

- 当 `left` 为假时，`left && right` 不会计算 `right`。
- 当 `left` 为真时，`left || right` 不会计算 `right`。

// 左侧已经为真，因此右侧包含除零的表达式不会被计算。

```
const enabled = true
const safe = enabled || (1 / 0 == 0)
```

```
assert(safe);
```

由于 `enabled` 为真，除法表达式永远不会执行。

当后一个表达式只有在前置条件成立后才有效时，短路求值尤其有用。

在表达式中调用函数

凡是接受相应结果类型的表达式位置，都可以调用有返回值的函数或内置函数：

// 计算名称长度，并把 37 向上调整到 16 的整数倍。

```
const name_length = lengthof("XIRASM")
const aligned_size = ((37 + 15) / 16) * 16
```

```
db(name_length);
dd(aligned_size);
```

函数调用可以嵌套：

// 先去掉两端空格，再把结果转换为大写。

```
const normalized = upper(trim(" kernel "))
assert(normalized == "KERNEL");
```

负责写出内容或执行汇编器操作、但不返回值的函数应作为独立语句调用，不能提供表达式值。第 5 章将详细介绍返回值的函数。

字段访问

带有命名字段的值使用 `.` 选择字段：

```
header.magic
header.entry_offset
```

结构体和联合值将在第 9 章介绍。 `target.bits` 和 `target.isa` 等查询属于目标平台条件，将在流程控制部分介绍；它们不是普通值，不应复制到普通绑定中。

运算符优先级

下表同一行列出的运算符具有相同优先级；越靠上的行结合得越紧密：

优先级	运算符
最高	函数调用、括号分组表达式、字段访问
一元运算	<code>+</code> 、 <code>-</code> 、 <code>~</code> 、 <code>!</code>
乘除与取余	<code>*</code> 、 <code>/</code> 、 <code>%</code>
移位	<code><<</code> 、 <code>>></code>
加减	<code>+</code> 、 <code>-</code>
按位与	<code>&</code>
按位异或	<code>^</code>
按位或	<code> </code>
大小比较	<code><</code> 、 <code><=</code> 、 <code>></code> 、 <code>>=</code>
相等判断	<code>==</code> 、 <code>!=</code>
逻辑与	<code>&&</code>
最低	<code> </code>

处于同一优先级的运算符按从左到右的顺序计算。

XIRASM 严格遵循这张表，它可能与其他语言中的优先级不同。尤其需要注意，移位的优先级高于加法和减法：

```
// 第一项先执行移位；第二项用括号明确要求先做加法。  
const shift_first = 1 + 2 << 3  
const grouped_shift = (1 + 2) << 3  
  
dd(shift_first, grouped_shift);
```

第一个值按 `1 + (2 << 3)` 计算，因此结果是 `17`；第二个值是 `24`。

表达式同时包含移位、算术、掩码或比较时，应当使用括号。括号既能说明二进制布局的计算方式，也能降低以后修改表达式时出错的风险。

表达式错误

如果 XIRASM 无法得到定义明确的编译期值，就会拒绝相应表达式。常见原因包括：

- 使用未定义的名称；
- 操作数属于错误的值类别；
- 除数为零的除法或取余；
- 算术溢出或减法下溢；
- 访问未知字段；
- 使用无效参数调用函数。

这些错误会停止汇编。XIRASM 不会把它们替换为零，也不会忽略它们，因为那样可能悄悄破坏指令操作数或二进制布局。

下一章将在 `if`、`while` 和 `for` 中使用表达式，以便在汇编期间选择和重复执行源代码。

[返回语言指南](#)

第 4 章：流程控制

流程控制在汇编期间运行

XIRASM 的流程控制决定构造输出内容时执行哪些源代码操作。它不会自动生成运行时分支或循环。

```
// 这个编译期选项决定输出调试标记还是发布标记。  
const debug_build = false  
  
if debug_build {  
    db("DEBUG");  
} else {  
    db("RELEASE");  
}
```

只有一个分支会写出字节。这里 `debug_build` 为假，因此输出内容包含 `RELEASE`，生成的文件中不存在运行时条件。

编译期流程控制适合完成以下任务：

- 选择目标平台专用的源代码；
- 重复生成数据或指令；
- 遍历编译期集合；
- 计算表格与偏移；
- 按需加入文件格式记录；
- 验证由源代码控制的配置。

运行时流程控制仍然使用普通处理器指令，例如 x86 的 `jmp`、`call` 和条件分支指令。

if 与 else

`if` 条件必须产生布尔值：

```
// 根据载荷大小选择一个单字节标记。  
const payload_size = 32  
  
if payload_size <= 64 {  
    db(0x01);  
} else {  
    db(0xff);  
}
```

被选中的代码块在自己的作用域中执行。另一个代码块既不会产生输出，也不会执行任何编译期操作。

else 代码块可以省略：

```
// 只有选项启用时才写出标记文本。
const include_marker = true

if include_marker {
    db("MARK");
}
```

当一项计算需要三个或更多分支时，可以使用 **else if**：

```
// 通过链式条件把载荷大小划分为三个区间。
const payload_size = 32

if payload_size < 16 {
    db(1);
} else if payload_size < 64 {
    db(2);
} else {
    db(3);
}
```

这段代码会写出 **02**。

选择处理器指令

if 代码块可以包含自然的处理器指令写法：

```
// 选择 64 位 x86 指令编码。
x86.use64();

// 编译期常量决定采用哪组返回值指令。
const return_zero = true

entry:
    if return_zero {
        // 清零 eax，令例程返回零。
        xor eax, eax
    } else {
        // 只有条件为假时才把返回值设为一。
        mov eax, 1
    }
    ret
```

由于 `return_zero` 为真，最终生成的指令是：

```
// 编译期条件只保留清零返回值的途径。  
xor eax, eax  
// 返回调用者。  
ret
```

未选中的 `mov` 指令不会加入输出内容。这是在编译期选择指令，不是运行时条件分支。

目标平台条件

可以直接在 `if` 条件中查询目标平台：

```
// x86-64 目标使用 64 位指令编码模式。  
if target.isa == .x86_64 {  
    x86.use64();  
}  
  
// 根据当前位宽写出对应的文本标记。  
if target.bits == 64 {  
    db("wide");  
} else {  
    db("narrow");  
}
```

`target.isa` 标识当前选择的指令集类别，`target.bits` 给出当前位宽。`x86.use32()`、`x86.use64()`、`riscv.use32()` 和 `riscv.use64()`；等模式调用会更新后续目标平台条件观察到的位宽。

目标平台查询使用专门的条件语法。应当直接在 `if` 条件中使用它们，不要把 `target.bits` 或 `target.isa` 复制到普通绑定中。

遍历数值范围

迭代次数已知时，可以配合 `range(start, end)` 使用 `for`：

```
// 依次写出从起始值零到结束值四之前的每个索引。  
for index in range(0, 4) {  
    db(index);  
}
```

这段代码会写出：

```
00 01 02 03
```

数值范围包含起始值，但不包含结束值。范围只能向前递增，也可以为空；`range(4, 0)` 这样的递减范围会被拒绝。

循环绑定属于循环体的作用域：

```
// 每次迭代都在索引上加 0x10，再写出编码后的值。
for index in range(0, 3) {
    const encoded = index + 0x10
    db(encoded);
}
```

这段代码会写出 `10 11 12`。每次迭代中的 `index` 和 `encoded` 都只在该次循环的局部作用域内有效。

遍历列表

`for` 循环可以依次访问编译期列表中的值：

```
// 列表按输出顺序保存三个待写出的字节值。
const opcodes: list = list.of(0x90, 0x90, 0xc3)

for opcode in opcodes {
    db(opcode);
}
```

循环按列表顺序访问各个值。这种形式适合处理字节表、导入描述、生成的名称，以及其他已经表示为集合的数据。

映射不能直接迭代。先用 `map.keys(...)` 或 `map.values(...)` 得到列表，再遍历该列表。第 6 章将介绍集合。

while

如果循环是否结束取决于循环中不断更新的值，使用 `while`：

```
// 每次写出当前值，然后推进到下一个值。
let value = 1

while value <= 4 {
    db(value);
    value = value + 1
}
```

这段代码会写出：

```
01 02 03 04
```

每次迭代开始前都会计算条件。如果条件一开始就是假，循环体不会执行。

应当让循环的结束规则紧邻循环并清楚可见。无法终止的编译期循环会使汇编无法完成，因此 XIRASM 会拒绝超过 1,000,000 次迭代的流程控制。

break 与 continue

`break` 会结束最内层的活动 Meta 循环。`continue` 会跳过当前迭代中剩余的语句并开始下一次迭代。二者都可以用于 `for`、`while` 和延迟执行的 `while` 循环体，也可以出现在循环体内嵌套的 `if` 中。

例如：

```
for value in range(0, 8) {
    if value == 6 {
        break;
    }
    if (value & 1) != 0 {
        continue;
    }
    db(value);
}
```

这段代码会写出 `00 02 04`。循环控制语句不会跨越函数调用边界；在循环外使用它们属于错误。

选择流程控制形式

需求	使用方式
在两段源代码中选择一段	<code>if / else</code>
在多段源代码中选择一段	<code>if / else if / else</code>
按指令集或位宽选择源代码	目标平台条件
重复执行已知次数	配合 <code>range</code> 使用 <code>for</code>
访问编译期集合中的值	配合列表使用 <code>for</code>
重复执行，直到可变状态满足条件	<code>while</code>
选择要生成的指令	在 <code>if</code> 中放入处理器指令
提前结束最内层循环	<code>break</code>
跳过当前迭代的剩余部分	<code>continue</code>

如果迭代范围已经确定，应优先使用 `for` 而不是 `while`。这样更容易判断生成的输出内容，也不需要手工更新循环变量。

下一章会把重复使用的计算和输出操作封装成函数，并介绍参数、返回值和局部作用域。

返回语言指南

第 5 章：函数与作用域

函数封装编译期工作

函数可以为需要复用的编译期工作命名。它可以计算一个值、生成输出内容，也可以把多个较底层的操作组合成一个源代码层面的动作。

XIRASM 提供两种函数形式：

- **过程函数**执行操作，但不产生表达式值；
- **返回值函数**计算一个值，并用 `->` 声明结果类型。

两种形式都使用 `fn`、参数和代码块函数体。它们在汇编期间执行，而不是在生成的程序运行时执行。

函数声明只能出现在顶层。声明函数本身不会写出任何字节；只有调用过程函数，或者把计算结果传给输出接口时，才会生成输出内容。

过程函数

过程函数可以把多个生成输出的操作组合在一起：

```
// 把传入字节及其后继值作为一组连续数据写出。  
fn emit_pair(value: u8) {  
    db(value);  
    db(value + 1);  
}  
  
// 两次调用分别生成 02 03 和 08 09。  
emit_pair(2);  
emit_pair(8);
```

这段代码会写出：

```
02 03 08 09
```

这个函数没有 `->` 返回类型，因此它是过程函数。过程函数调用属于语句，末尾需要写分号。

过程函数可以包含调用位置能够使用的各种编译期语句，包括声明、赋值、流程控制、数据写出以及其他过程函数调用。因此，过程函数适合生成重复的二进制记录、指令序列、表项和文件格式构造步骤。

过程函数本身不是值，不能用来初始化绑定：

```
fn emit_marker() {
    db(0x90);
}

const marker = emit_marker()
```

这段代码会被拒绝，因为 `emit_marker()` 只执行操作，并不返回值。

形参与实参

形参写在函数声明的圆括号内：

```
// 连续写出 count 个相同字节。
fn emit_run(value: u8, count: u64) {
    // index 负责控制循环次数，循环体只需要使用 value。
    for index in range(0, count) {
        db(value);
    }
}

// 写出四个 cc 字节。
emit_run(0xcc, 4);
```

输出内容是四个 `cc` 字节。虽然循环体没有在其他地方使用 `index` 绑定，但它仍然负责控制循环。

形参类型可以省略：

```
// 形参类型可以省略，调用时传入的值会绑定到对应形参。
fn add(left, right) -> u64 {
    return left + right;
}

// 计算 20 + 22，并把结果 2a 写成一个字节。
db(add(20, 22));
```

这段代码会写出 `2a`。每次调用时传入的实参值决定该次调用的形参值。对于公开使用的辅助函数，类型标注仍然很有用：它既说明函数要求的约定，也能在调用位置附近拒绝不合适的实参。

实参按位置对应形参。调用函数时，必须按照声明顺序为每个形参提供且仅提供一个实参。函数不支持默认实参。

同一个函数声明中的形参名称不能重复。每次调用都会获得各自独立的形参绑定，因此一次调用不会覆盖另一次调用中的值。

返回值函数

如果函数需要产生表达式值，就在声明中添加 `-> type`：

```
// 把 value 向上对齐到 alignment 的整数倍。
fn align_up(value: u64, alignment: u64) -> u64 {
    return ((value + alignment - 1) / alignment) * alignment;
}

// 0x73 按 0x20 对齐后得到 0x80，再以双字节写出。
const header_size = align_up(0x73, 0x20)
dw(header_size);
```

函数返回 `0x80`，因此输出内容是：

```
80 00
```

`return` 语句末尾需要写分号。返回表达式会转换为声明的返回类型。返回类型可以描述整数、布尔值、字符串和字节序列等普通编译期值：

```
// 判断传入值是否等于一个 0x1000 字节的页大小。
fn is_page(value: u64) -> bool {
    return value == 0x1000;
}

// 返回固定的两个签名字节。
fn signature() -> bytes {
    return b"XR";
}

// 先检查计算结果，再写出签名字节。
assert(is_page(0x1000));
db(signature());
```

只要某个位置接受返回结果对应类型的表达式，就可以在那里使用返回值函数，例如声明、实参、条件、另一个函数调用或更大的表达式。

返回规则与副作用

返回值函数被调用后，必须执行到一条 `return` 语句。函数体结束时仍未返回值会被拒绝：

```
fn incomplete(value: u64) -> u64 {
    const doubled = value * 2
}

const result = incomplete(4)
```

返回值必须与声明的类型兼容：

```
fn enabled() -> bool {
    return 1;
}

const result = enabled()
```

这段代码会被拒绝，因为整数结果不符合 `bool` 返回约定。

返回值函数用于执行计算。它可以使用局部绑定、流程控制和其他返回值函数，但不能写出输出内容，也不能产生其他汇编器副作用：

```
fn bad_counter() -> u64 {
    db(1);
    return 1;
}

const result = bad_counter()
```

需要改变输出内容或布局时，应使用过程函数；调用者需要计算结果时，应使用返回值函数。

过程函数不声明返回类型，也不能返回值：

```
fn emit_one() {
    return 1;
}

emit_one();
```

执行到过程函数体末尾时，本次调用会自然结束。

函数局部作用域

形参和函数内部声明的绑定只属于当前这次调用：

```
// 加上固定开销，并确保结果不小于 16。
fn adjusted_size(size: u64) -> u64 {
    const overhead = 4
    let result = size + overhead

    if result < 16 {
        result = 16
    }

    return result;
}

// 两次调用各自使用独立的局部绑定。
db(adjusted_size(3));
db(adjusted_size(20));
```

这段代码会写出 `10 18`。每次调用都会创建新的 `size`、`overhead` 和 `result`，调用结束后便不能再访问这些名称。

函数内部的代码块会创建嵌套作用域，并且可以遮蔽外层名称：

```
// 使用嵌套作用域中的同名常量参与一次局部计算。
fn combine(value: u64) -> u64 {
    let result = value

    {
        // 此处的 value 只在这个代码块内表示常量 5。
        const value = 5
        result = result + value
    }

    return result;
}

// 参数值 3 与局部常量 5 相加，结果为 08。
db(combine(3));
```

这段代码会写出 `08`。在嵌套代码块内，`value` 表示局部常量 `5`；代码块结束后，名为 `value` 的形参会重新可见。可变绑定 `result` 属于函数作用域，因此嵌套代码块可以更新它。

中间计算应优先使用局部名称。这样可以避免临时状态泄漏到源文件的其他部分，也能让每次调用彼此独立。

声明顺序

函数必须在源文件中第一次调用它的语句之前声明：

```
// 先声明函数，后面的源代码才能引用它。
fn add(left: u64, right: u64) -> u64 {
    return left + right;
}

// 调用已经可见的函数，并写出结果。
const answer = add(20, 22)
db(answer);
```

如果把调用移到声明之前，代码会被拒绝，因为此时源文件中还不知道这个函数名称。

函数声明本身必须位于顶层，不能出现在另一个函数、循环、条件代码块或普通作用域代码块内部。应把相关函数相邻放置在顶层，再由后面的源代码调用它们。

第 10 章将说明如何通过包含文件，让另一个源文件能够使用可复用的函数声明。

递归与调用深度

只要计算具有明确的基本情况，返回值函数就可以调用自身：

```
// 递归计算从 value 到 1 的总和。
fn triangular(value: u64) -> u64 {
    // value 为零时停止递归。
    if value == 0 {
        return 0;
    }

    return value + triangular(value - 1);
}

// triangular(4) 计算 4 + 3 + 2 + 1，并写出 0a。
db(triangular(4));
```

这段代码会写出 `0a`，也就是 `4 + 3 + 2 + 1` 的结果。

递归函数仍然在汇编期间执行。递归深度应保持较小，并且终止条件应当清晰。XIRASM 会拒绝深度超过 128 次调用的函数调用链，防止失控的递归耗尽汇编器资源。

如果任务本质上是在一个数值范围或集合中迭代，应使用循环；如果计算本身具有递归结构，并且基本情况容易验证，则应使用递归。

选择函数形式

需求	使用形式
写出字节或指令	过程函数
把多次输出接口调用组合成一个操作	过程函数
为表达式计算一个值	返回值函数
复用不产生副作用的布局或编码计算	返回值函数
让临时名称仅在一次操作内可见	任一种函数形式
对简单的数值范围或集合执行重复操作	通常使用循环
表达天然具有递归结构的计算	递归返回值函数

每个函数都应专注于一项工作。较小的计算函数更容易与其他表达式组合，较小的过程函数则能让输出效果在调用位置清晰可见。

下一章将介绍列表、映射、字符串和字节序列，供需要处理集合和结构化文本的函数使用。

[返回语言指南](#)

第 6 章：集合与文本

集合是编译期值

汇编程序往往不只描述一个个整数。源文件可能需要一组有序字段、一张带名称的属性表、动态生成的符号名称，或者一段必须逐字节保持不变的数据。

XIRASM 为这些用途提供了四种常见值类型：

类型	表示的内容	常见用途
<code>string</code>	源代码层面的文本	名称、路径、诊断信息、解析出的字段
<code>bytes</code>	精确的二进制数据	签名、编码后的整数、二进制记录
<code>list</code>	有序值	表格、重复项、有序描述信息
<code>map</code>	以字符串为键的值	命名选项、记录、查找表

这些类型都属于编译期值。创建字符串、字节序列、列表或映射并不会写出任何内容。只有把值传给输出接口，它的内容才会成为生成文件的一部分。

字符串、字节序列、列表和映射都提供会产生新值的辅助接口。`list.push` 或 `map.set` 之类的操作会返回一个新值，而不是就地修改原值。列表和映射还提供显式的语句接口，用于逐步更新由 `let` 绑定的集合。通过这些接口插入的值仍会被深拷贝，因此不同绑定之间不会产生隐式别名。

用字符串表示源文本

字符串保存汇编过程中使用的文本：

```
// 先清除名称两端的空白，再把 ASCII 字母统一转换为小写。
const raw_name: string = "  Kernel64  "
const name: string = lower(trim(raw_name))

// 检查规范化后的名称及其组成部分。
assert(name == "kernel64");
assert(starts_with(name, "kernel"));
assert(ends_with(name, "64"));
assert(contains(name, "nel"));

// 把最终文本作为字节写入输出内容。
emit.bytes(name);
```

这段代码会写出 `kernel64` 的八个文本字节。

最常用的字符串辅助接口包括：

- `trim(text)` 删除文本两端的空白；
- `lower(text)` 和 `upper(text)` 分别把 ASCII 字母转换为小写和大写；
- `starts_with`、`ends_with` 和 `contains` 检查文本；
- `replace(text, needle, replacement)` 替换匹配的文本；
- `to_string(value)` 生成某个值的文本表示。

字符串的大小写转换面向 ASCII 字符。名称、配置文本、路径、诊断信息和其他供人阅读的值适合使用字符串；如果必须逐字节精确保留二进制数据，则应使用 `bytes`。

拆分与连接文本

`split` 把带分隔符的文本拆成字符串列表，`join` 则把字符串列表连接起来：

```
// 把逗号分隔的节名称拆成列表，再按路径形式重新连接。
const sections: list = split("text,data,bss", ",")
const path: string = join(sections, "/")

// 列表索引从零开始，因此索引 0 和 2 分别对应首尾元素。
assert(len(sections) == 3);
assert(list.get(sections, 0) == "text");
assert(list.get(sections, 2) == "bss");
assert(path == "text/data/bss");

// 写出重新组合后的路径文本。
emit.bytes(path);
```

索引从零开始。`list.get(sections, 0)` 返回第一个元素。

拆分操作适合处理源代码层面的小型格式、类似命令行的文本和动态生成的名称。结构更复杂的外部数据将在第 10 章介绍，其中包括文件、JSON 和 TOML。

用字节序列表示二进制值

`bytes` 值是一段精确的字节序列：

```
// 从十六进制文本构造文件标记，并把版本号编码为两个小端字节。
const magic: bytes = bytes.from_hex("58495200")
const version: bytes = bytes.le(3, 2)
const header: bytes = bytes.concat(magic, version)

// 检查原始标记和版本字段的编码结果。
assert(bytes.eq(magic, bytes.from_hex("58495200")));
assert(bytes.hex(version) == "0300");

// 按连接后的顺序写出完整文件头。
emit.bytes(header);
```

输出内容为：

```
58 49 52 00 03 00
```

`bytes.from_hex` 把十六进制文本转换为二进制数据。`bytes.le(value, width)` 使用指定字节数，按小端顺序编码一个整数。`bytes.concat` 把两段字节序列连接起来。

需要明确比较两段字节序列是否相等时，使用 `bytes.eq`；需要把二进制值显示为小写十六进制文本时，使用 `bytes.hex`。

构造字节序列

字节辅助接口会返回新值，因此可以用清晰的步骤逐步构造二进制记录：

```
// 从 ABC 的字节表示开始，在索引 1 处插入连字符。
const base: bytes = bytes.from_hex("414243")
const marked: bytes = bytes.insert(base, 1, b"-")

// 从索引 2 开始替换一个字节，并追加两个 0xff 字节。
const patched: bytes = bytes.replace(marked, 2, 1, b"Z")
const trailer: bytes = bytes.repeat(2, 0xff)
const result: bytes = bytes.concat(patched, trailer)

// 只写出最终结果，各个中间值仍可继续复用。
emit.bytes(result);
```

这段代码会写出：

```
41 2d 5a 43 ff ff
```

这些操作包括：

- `bytes.new()` 创建空字节序列;
- `bytes.push(value, byte)` 在末尾追加一个字节;
- `bytes.repeat(count, byte)` 创建重复字节;
- `bytes.insert(value, index, addition)` 在从零开始的索引处插入字节;
- `bytes.replace(value, index, count, replacement)` 替换一段字节范围;
- `bytes.concat(left, right)` 连接两段字节序列。

原始 `base` 值仍然是 `41 42 43`。每次操作都会产生一个新值，既可以保留或重复使用，也可以继续传给下一个操作。

列表保留顺序

列表保存一组有序的编译期值：

```
// 先追加一个元素，再替换索引 1 处的值。
const base: list = list.of(1, 2, 3)
const extended: list = list.push(base, 4)
const patched: list = list.set(extended, 1, 0xaa)

// 从索引 1 开始截取两个元素，原列表不会被修改。
const middle: list = list.slice(patched, 1, 2)

assert(list.eq(base, list.of(1, 2, 3)));
assert(list.eq(patched, list.of(1, 0xaa, 3, 4)));
assert(list.eq(middle, list.of(0xaa, 3)));

// 按列表中的固定顺序逐项写出字节。
for value in patched {
    db(value);
}
```

这段代码会写出：

```
01 aa 03 04
```

`list.set` 返回一个替换了指定元素的新列表。`list.slice` 接收起始索引和元素数量。这两个操作都不会修改输入列表。

其他常用列表操作包括：

- `list.new()` 创建空列表;
- `list.get(value, index)` 读取一个元素;
- `list.concat(left, right)` 连接两个列表;
- `list.eq(left, right)` 比较列表内容;

- `len(value)` 返回元素数量。

列表自身包含元素顺序，因此可以自然地配合 `for` 使用。节描述、表格行、字节块以及其他必须按确定顺序处理的数据都适合使用列表。

更新由 `let` 绑定的集合

当一个局部算法需要逐步构造列表或映射时，可以使用可变语句接口：

```
let items: list = list.of(1, 2)
list.push_mut(items, 3);
list.set_mut(items, 0, 4);

let options: map = map.new()
map.set_mut(options, "arch", "x64");
map.set_mut(options, "items", items);
```

第一个参数必须是由 `let` 绑定的直接标识符。`const` 绑定、临时表达式、函数调用结果、字段访问、未定义名称以及类型不匹配的集合都会被拒绝。词法查找选择最近的绑定，因此块内的同名 `let` 会遮蔽外层绑定。普通 lowering 阶段可以更新顶层 `let`；值函数只能更新本次调用内部的局部 `let`。

`list.push_mut` 追加一个深拷贝后的元素；`list.set_mut` 替换已有的从零开始的索引；`map.set_mut` 添加或替换字符串键对应的值，替换已有键时不会改变它的插入位置。这些接口是语句，不会产生表达式结果，也不能在 `defer` 或 `late_layout` 块中使用。

深拷贝仍然是值隔离边界：

```
let items: list = list.of(1)
const snapshot: list = items
list.push_mut(items, 2);

assert(list.eq(snapshot, list.of(1)));
assert(list.eq(items, list.of(1, 2)));
```

列表可以保存更复杂的值

列表元素并不局限于单个整数。列表还可以组织字符串、字节序列、映射或其他编译期值：

```
// 把文本标记和一个双字节小端整数组织成有序字节块。
const chunks: list = list.concat(
  list.of(b"XR"),
  list.of(bytes.le(0x1234, 2))
)

// 列表决定写出顺序，每个元素保留各自精确的字节表示。
for chunk in chunks {
  emit.bytes(chunk);
}
```

这段代码会写出：

```
58 52 34 12
```

当一条记录更适合表示为若干有序片段时，这种写法很有用。列表控制片段顺序，每段字节序列则控制自身精确的二进制表示。

映射保存命名值

映射把字符串键与编译期值关联起来：

```
// 逐步加入命名属性，并用新值替换已有的 arch 属性。
const base: map = map.set(map.new(), "arch", "x64")
const configured: map = map.set(base, "mode", "release")
const changed: map = map.set(configured, "arch", "rv64")

// 映射中的值也可以是列表等更大的编译期值。
const complete: map = map.set(changed, "tags", list.of("asm", "dsl"))

// 检查键是否存在，并读取必需或带默认值的属性。
assert(len(complete) == 3);
assert(map.has(complete, "arch"));
assert(!map.has(complete, "missing"));
assert(map.get(base, "arch") == "x64");
assert(map.get(changed, "arch") == "rv64");
assert(map.get_or(complete, "missing", "default") == "default");
assert(list.eq(map.get(complete, "tags"), list.of("asm", "dsl")));
```

`map.set` 会添加一个键；如果键已经存在，则返回一个替换了该键对应值的新映射。较早的映射值不会改变，因此在 `changed` 保存 `"rv64"` 之后，`base` 中仍然保存着 `"x64"`。

可以使用：

- `map.has(value, key)` 检查键是否存在；

- `map.get(value, key)` 读取必需键对应的值；
- `map.get_or(value, key, fallback)` 读取可选键对应的值，键不存在时返回后备值；
- `map.eq(left, right)` 比较映射内容。

`map.eq` 比较键和值的内容，而不比较插入顺序。

遍历映射内容

映射主要用于按键查找。遍历之前，应先把它的键或值转换成列表：

```
// 用映射保存两个带名称的二进制字段。
const fields: map = map.set(
    map.set(map.new(), "magic", b"XR"),
    "version",
    bytes.le(3, 2)
)

// 键和值分别转换为列表，便于检查数量或逐项处理。
const keys: list = map.keys(fields)
const values: list = map.values(fields)

assert(len(keys) == 2);
assert(len(values) == 2);

// 逐项写出映射中保存的字节序列。
for value in values {
    emit.bytes(value);
}
```

处理名称时使用 `map.keys`，处理已保存的值时使用 `map.values`。如果文件格式规定了明确的处理顺序，应把顺序保存在列表中，而只把映射用于查找。

在函数中组合集合

当函数需要把一份描述转换成二进制数据时，集合尤其有用：

```
// 把列表中的每个数值编码为两个小端字节。
fn encode_u16(values: list) -> bytes {
    let result: bytes = bytes.new()

    // 每次连接都会产生新的字节序列，并更新可变绑定 result。
    for value in values {
        result = bytes.concat(result, bytes.le(value, 2))
    }

    return result;
}

// 调用者决定何时把函数生成的字节序列写入输出内容。
const words: list = list.of(0x1234, 0xabcd)
emit.bytes(encode_u16(words));
```

这段代码会写出：

```
34 12 cd ab
```

函数接收一份有序描述，逐步构造不可变的字节值，最后返回完整的编码结果。调用者负责决定在何时、何处写出该结果。

这种职责划分可以很好地扩展：

- 字符串描述名称和源代码层面的文本；
- 映射描述带名称的属性；
- 列表保留输出顺序；
- 字节序列保存最终的二进制表示；
- 过程函数决定在何处写出这种表示。

选择集合类型

需求	使用方式
供人阅读的源文本	<code>string</code>
精确的二进制表示	<code>bytes</code>
有序值序列	<code>list</code>
按字符串键查找	<code>map</code>
解析带简单分隔符的文本	使用 <code>split</code> 转换为 <code>list</code>
重新构造带分隔符的文本	<code>join</code>
在内存中构造二进制记录	<code>bytes</code> 辅助接口
既保留顺序又支持查找	同时使用 <code>list</code> 和 <code>map</code>

应选择与数据含义一致的类型。不要把字符串当作二进制缓冲区，也不要再在格式本身包含顺序要求时使用映射代替有序结构。

下一章将介绍词法单元和模式匹配，用于处理那些必须按照语言语法理解、而不能只视为普通字符串的源文本。

[返回语言指南](#)

第 7 章：词法单元与模式匹配

词法单元表示源代码结构

字符串辅助接口把文本视为字符序列。对于名称、路径、分隔字段和简单替换，这种方式已经足够。类似源代码的文本往往需要另一种处理模型。

请看下面这段文本：

```
load rax, [rbx+(rcx*4)]
```

其中的逗号、方括号、圆括号、名称、运算符和整数字面量都具有结构意义。如果只按空格拆分，不仅会丢失这些结构，还会对仅有空格差异的等价写法产生不一致的处理结果。

XIRASM 可以把类似源代码的文本转换为词法单元列表：

```
// 把表达式拆成词法单元，并验证空白不会进入结果列表。
const source: string = "count + 0x2a"
const source_tokens: List = tokens.of(source)

assert(len(source_tokens) == 3);
assert(list.get(source_tokens, 0) == "count");
assert(list.get(source_tokens, 1) == "+");
assert(list.get(source_tokens, 2) == "0x2a");

// 按规范形式重新连接词法单元，并把结果写入输出内容。
emit.bytes(tokens.join(source_tokens));
```

这段代码会写出规范化后的文本：

```
count+0x2a
```

`tokens.of` 会丢弃无意义的空白，同时保留词法单元的顺序。`tokens.join` 使用规范间距重新呈现这个序列，并不会逐字节还原原始字符串。

当字符排列本身很重要时，应使用字符串。当名称、标点、运算符、字面量和平衡分组的结构很重要时，应使用词法单元。

匹配词法单元结构

`match.tokens(pattern, input)` 会把类似源代码的输入与词法单元模式进行比较：

```
// 匹配 load 形式，并分别捕获目标名称和完整的源操作数。
const line: string = "load rax, [rbx+(rcx*4)]"
const result: map = match.tokens(
    "load destination:name =, source:tokens",
    line
)

// 完整匹配成功后，再读取命名捕获得到的值。
assert(map.get(result, "ok"));

const captures: map = map.get(result, "captures")
assert(map.get(captures, "destination") == "rax");
assert(tokens.join(map.get(captures, "source")) == "[rbx+(rcx*4)]");
```

结果是一个包含两个字段的映射：

- `"ok"` 是布尔值，表示完整输入是否匹配成功；
- `"captures"` 是一个映射，保存各个命名捕获所提取的值。

只有检查 `"ok"` 后，才能读取捕获结果。

字面量模式词法单元

以 `=` 开头的模式片段会匹配一个完全相同的词法单元：

```
// `=&&` 要求输入中必须出现精确的逻辑与词法单元。
const result: map = match.tokens(
    "left:name =&& right:name",
    "ready && enabled"
)

assert(map.get(result, "ok"));
```

`=&&` 要求输入中出现逻辑与词法单元。同一规则也适用于名称和标点：

<code>=load</code>	精确匹配名称词法单元
<code>=,</code>	精确匹配逗号词法单元
<code>=["</code>	精确匹配左方括号词法单元
<code>===</code>	精确匹配相等运算符词法单元

第一个 `=` 是模式标记。因此，`===` 表示“匹配 `==` 词法单元”，而不是匹配一个由三个字符组成的相等运算符。

模式片段之间使用空白分隔。模式中的空白只用于提高可读性；实际匹配依据的是词法单元，而不是输入中原有的间距。

捕获种类

捕获的形式为 `name:kind`。捕获名称会成为结果中 `"captures"` 映射的键。

种类	匹配内容	捕获的值
<code>token</code>	任意种类的一个词法单元	<code>string</code>
<code>name</code>	一个类似标识符的名称	<code>string</code>
<code>int</code>	一个整数字面量	整数值
<code>quoted</code>	一个带引号的词法单元	去除引号后的 <code>string</code>
<code>tokens</code>	一段保持分组平衡的词法单元	由词法单元字符串组成的 <code>list</code>

同一个模式中的捕获名称必须唯一。

单词法单元捕获可以精确描述小型命令语法：

```
// 从 set 形式中提取目标名称，并把整数字面量转换为整数值。
const assignment: map = match.tokens(
  "=set target:name =, value:int",
  "set count, 0x2a"
)
const assignment_captures: map = map.get(assignment, "captures")

// 保留中间运算符的原始词法单元文本。
const operation: map = match.tokens(
  "left:name operator:token right:name",
  "count + step"
)
const operation_captures: map = map.get(operation, "captures")

// 去除外层引号后捕获消息文本。
const message: map = match.tokens("db text:quoted", "db 'READY'")
const message_captures: map = map.get(message, "captures")

// 验证每种捕获都生成了预期类型和值。
assert(map.get(assignment, "ok"));
assert(map.get(assignment_captures, "target") == "count");
assert(map.get(assignment_captures, "value") == 42);
assert(map.get(operation_captures, "operator") == "+");
assert(map.get(message_captures, "text") == "READY");
```

`int` 捕获会把字面量转换为整数值。`quoted` 捕获会移除外层引号，并解码受支持的转义序列。`token` 捕获只返回词法单元文本，不会要求它属于更具体的类型。

平衡的词法单元范围

`tokens` 捕获可以使用多个词法单元，同时保持圆括号、方括号和花括号的平衡：

```
// 把方括号内的嵌套地址表达式作为一个平衡范围捕获。
const result: map = match.tokens(
  "=load destination:name =, address:tokens",
  "load rax, [rbx+(rcx*4)]"
)
const captures: map = map.get(result, "captures")
const address: list = map.get(captures, "address")

assert(map.get(result, "ok"));
assert(tokens.join(address) == "[rbx+(rcx*4)]");
```

捕获的值是词法单元列表，而不是字符串。继续保留词法单元形式，其他匹配器就能直接检查这段内容，无须再次对文本进行词法分析。

平衡捕获适用于 `()`、`[]` 和 `{}`。`<`、`>` 等比较运算符仍然只是普通的运算符词法单元：

```
// 比较运算符不会被当作分组边界，整个表达式仍可被捕获。
const result: map = match.tokens("expression:tokens", "left < right")
const captures: map = map.get(result, "captures")

assert(map.get(result, "ok"));
assert(tokens.join(map.get(captures, "expression")) == "left<right");
```

最小匹配与回溯

当 `tokens` 捕获后面还有其他模式片段时，它最初会尽可能少地使用词法单元。如果剩余模式无法匹配，捕获范围就会扩展，然后重新尝试匹配：

```
// prefix 会先尝试空范围，再扩展到足以让整数捕获成功的位置。
const result: map = match.tokens(
  "prefix:tokens value:int",
  "name 42"
)
const captures: map = map.get(result, "captures")

assert(map.get(result, "ok"));
assert(tokens.join(map.get(captures, "prefix")) == "name");
assert(map.get(captures, "value") == 42);
```

第一次尝试会给 `prefix` 分配一个空范围，但 `value:int` 无法匹配 `name`。匹配器随后扩展 `prefix`，使其包含 `name`，这样整数捕获就能匹配 `42`。

带类型的捕获遇到错误种类的词法单元时，只是一次普通的匹配失败。一个有效模式不会因此产生汇编错误：

```
// name 不是整数字面量，因此模式有效，但本次输入不匹配。
const result: map = match.tokens("value:int", "name")
assert(!map.get(result, "ok"));
```

这种行为便于依次尝试多个有效模式。

空词法单元范围

`tokens` 捕获可以为空：

```
// call 后没有参数，因此 arguments 捕获得到一个空列表。
const result: map = match.tokens("=call arguments:tokens", "call")
const captures: map = map.get(result, "captures")

assert(map.get(result, "ok"));
assert(len(map.get(captures, "arguments")) == 0);
```

如果词法单元范围必须非空，可以在匹配成功后检查其长度，也可以在模式中加入另一个必需的捕获。

匹配已有词法单元列表

传给 `match.tokens` 的输入也可以是已经生成的词法单元字符串列表：

```
// 先进行一次词法分析，再把同一列表直接交给模式匹配器。
const input: list = tokens.of("load r1, 42")
const result: map = match.tokens(
    "=load destination:name =, value:int",
    input
)
const captures: map = map.get(result, "captures")

assert(map.get(result, "ok"));
assert(map.get(captures, "destination") == "r1");
assert(map.get(captures, "value") == 42);
```

当多个处理步骤需要检查同一段源代码时，可以在辅助接口之间直接传递词法单元列表。只有在确实需要文本值时，才使用 `tokens.join` 转回文本。

尝试不同结构

模式本身不包含“或者”运算符。应在编译期流程控制中依次尝试每一种完整结构：

```
// 先尝试 load 结构；当前输入不匹配时，再尝试 store 结构。
const line: string = "store r1, r2"
const load: map = match.tokens(
    "=load destination:name =, source:name",
    line
)

if map.get(load, "ok") {
    // load 分支只写出目标名称。
    const captures: map = map.get(load, "captures")
    emit.bytes(map.get(captures, "destination"));
} else {
    const store: map = match.tokens(
        "=store destination:name =, source:name",
        line
    )

    // store 分支写出目标名称和源名称。
    assert(map.get(store, "ok"));
    const captures: map = map.get(store, "captures")
    emit.bytes(map.get(captures, "destination"));
    emit.bytes(map.get(captures, "source"));
}
```

这段代码会写出 `r1r2`。每个模式只描述一种有效形式，普通的 `if / else` 逻辑负责判断输入采用了哪一种形式。

对于更大的语法，可以把每一种结构放进一个返回值的小函数中，并让调用位置清楚地显示分派顺序。

匹配失败与无效模式

普通的结构或类型不匹配会返回 `"ok" == false` 的结果。例如：

- 精确字面量不同；
- `name` 捕获收到标点；
- `int` 捕获收到名称；
- `tokens` 捕获中的括号不平衡；
- 模式已经结束，但输入仍有剩余。

格式错误的模式则不同。它会作为汇编错误被拒绝，而不是被报告为匹配失败。未知的捕获种类、空字面量或重复的捕获名称都会造成模式错误：

```
const invalid = match.tokens(  
  "value:name value:int",  
  "left 42"  
)
```

两个捕获都使用了键 `"value"`，因此这个模式无效。

请记住两者的区别：

- **模式有效，但输入不适用**，表示 `"ok" == false`；
- **模式定义无效**，表示源代码有错误，应当修正模式。

字符串还是词法单元

需求	使用方式
搜索子字符串	<code>contains</code>
检查文本前缀或后缀	<code>starts_with</code> / <code>ends_with</code>
解析使用简单分隔符分开的文本	<code>split</code>
保留名称、标点和分组结构	<code>tokens.of</code>
匹配一种精确的源代码结构	<code>match.tokens</code>
提取带类型的字段	<code>name</code> 、 <code>int</code> 或 <code>quoted</code> 捕获
捕获嵌套表达式或操作数	<code>tokens</code> 捕获
尝试多种命令形式	配合 <code>if</code> / <code>else</code> 使用多个模式

不要仅仅为了汇编普通处理器指令，就先把指令行转换为词法单元。自然的处理器指令本来就是源语言的一部分，可以直接书写。词法单元匹配最适合用于紧凑的用户自定义领域专用语言、生成的源代码片段、命令式记录，以及可复用的编译期辅助函数。

至此，本指南的语言基础部分全部结束。下一章将从目标平台、自然的处理器指令、标号和引用开始介绍汇编器模型。

[返回语言指南](#)

第 8 章：目标平台、处理器指令与标号

编译期语言决定需要生成什么内容；汇编器模型则决定生成的指令和数据放在哪里、使用哪一种指令集进行编码，以及如何把符号引用转换成具体数值。

本章从三个紧密相关的概念开始介绍汇编器模型：

- 当前目标平台选择指令集和对应的模式位数；
- 自然的处理器指令写法会为该目标平台创建指令片段；
- 标号为这些片段提供稳定的符号名称。

手写汇编代码时，这些功能应当保持直接：选择目标平台，正常编写指令和标号即可。只有在编译期生成确实需要动态计算时，才使用动态指令文本和计算得到的标号名称。

当前目标平台

每条指令都按照当前目标平台进行汇编。目标平台包含指令集系列，以及该指令集编码时所需的宽度信息。

命令行负责选择初始目标平台：

```
xirasm program.xir -o program.bin --target x86-64
xirasm program.xir -o program.bin --target x86
xirasm program.xir -o program.bin --target rv64
xirasm program.xir -o program.bin --target rv32
```

如果没有选择其他目标平台，XIRASM 默认使用 64 位 x86。源代码仍然可以明确选择指令模式。这样可以 让源文件自行说明所需模式，因此推荐在示例、可复用的包含文件以及依赖特定宽度的代码中明确设置：

```
// 明确选择后续指令使用的 64 位 x86 编码。
x86.use64();

entry:
    // 生成一个返回零的最小指令序列。
    xor eax, eax
    ret
```

模式调用只影响它之后出现的指令。每条指令都会记住创建它时所使用的目标平台。

选择 x86 和 RISC-V 模式

源代码可以使用下面这些接口选择模式：

接口	当前指令模式
<code>x86.use16()</code>	16 位 x86
<code>x86.use32()</code>	32 位 x86
<code>x86.use64()</code>	64 位 x86
<code>riscv.use32()</code>	32 位 RISC-V
<code>riscv.use64()</code>	64 位 RISC-V

同一个源文件可以切换模式：

```
// 第一条指令使用 16 位 x86 编码。
x86.use16();
mov ax, 1

// 第二条指令使用 32 位 x86 编码。
x86.use32();
mov eax, 2

// 第三条指令使用 64 位 x86 编码。
x86.use64();
mov rax, 3
```

三条指令分别保留自己的 16 位、32 位和 64 位编码环境。切换模式不会改写前面已经创建的指令。

选择编码模式并不会让生成的程序在运行时自动切换处理器模式。启动镜像、内核、固件组件或混合模式程序仍然需要自行完成必要的运行时切换。

RISC-V 的宽度选择同样按照源代码顺序生效：

```
// 选择 XLEN 为 64 位的 RISC-V 模式。
riscv.use64();

// 两条指令都会按照当前的 64 位 RISC-V 目标编码。
addi x1, x0, 1
addi x0, x0, 0
```

不要把 x86 的模式概念套用到 RISC-V 或 SPIR-V。XIRASM 分别保存它们的目标设置，不会因为 x86 需要 `mode_bits`，就把所有指令集都压缩成同一种宽度字段。

查询目标平台

编译期流程控制可以检查当前目标平台：

```
// 选择 x86 后端和 64 位模式。
x86.use64();

// 这个条件在汇编期间判断，不会生成运行时分支。
if target.isa == .x86_64 {
    assert(target.bits == 64);
    mov eax, 1
}
```

可以查询的目标系列值如下：

值	目标系列
<code>.x86_64</code>	使用 16 位、32 位或 64 位模式的 x86
<code>.riscv64</code>	XLEN 为 32 位或 64 位的 RISC-V
<code>.spirv</code>	SPIR-V

这些公开名称标识的是后端系列，并不表示当前模式位数。例如，调用 `x86.use32()` 之后，`target.isa == .x86_64` 仍然成立，而 `target.bits == 32`。

需要判断当前位数时使用 `target.bits`。RISC-V 条件也可以使用 `target.xlen` 判断 XLEN：

```
// 选择 XLEN 为 32 位的 RISC-V 模式。
riscv.use32();

// 同时检查后端系列与 XLEN。
if target.isa == .riscv64 {
    assert(target.xlen == 32);
    addi x1, x0, 1
}
```

目标条件在汇编期间执行，用来选择需要生成的源代码内容，不会写出运行时分支指令。

自然的处理器指令写法

指令使用所选指令集的常规文本语法：

```
// 选择 64 位 x86，然后直接编写普通指令。  
x86.use64();
```

```
entry:  
    mov rax, 1  
    add rax, 2  
    ret
```

一条处理器指令由助记符和后续操作数组成。可以使用空白提高可读性；方括号、圆括号或花括号内部的逗号属于嵌套操作数，不会在错误的位置拆分指令。

处理器指令行末不写分号，结构化的编译期调用则需要分号：

```
// 这两行是编译期接口调用，因此以分号结束。  
x86.use64();  
emit.u8(0x90);  
  
// 这一行是处理器指令，因此不写分号。  
nop
```

这里的 `x86.use64();` 和 `emit.u8(0x90);` 是编译期接口调用，`nop` 是自然书写的处理器指令。

XIRASM 会把指令文本与当前目标平台一起保存，再交给对应的指令集后端编码。标号、布局、输出区域和地址回填项仍然由前端管理。

在指令中使用编译期值

编译期常量可以直接出现在指令操作数中：

```
// 在汇编期间计算立即数。  
x86.use64();  
const initial_value: u32 = 40 + 2  
  
entry:  
    // 编译期常量会直接成为指令操作数。  
    mov eax, initial_value  
    ret
```

不需要为了使用编译期值而把指令改写成函数调用。汇编该指令时，操作数表达式会一并求值。

标号运算也可以保留在自然的处理器指令中：

```
// 把逻辑起始地址设置为 0x1000。
x86.use64();
origin(0x1000);

target:
    // 操作数表示标号地址再加四。
    mov rax, target + 4
    ret
```

这个表达式最终得到 `target` 的地址加四。在指令和标号的布局确定之前，该引用会一直保持符号形式。

能使用清晰的常量和标号时，不要拼接指令字符串。只有当指令助记符、操作数形式或符号名称本身需要计算时，才生成动态指令文本。

静态标号

标号定义由名称和末尾的冒号组成：

```
// 使用静态标号表达普通控制流。
x86.use64();

entry:
    mov eax, 1
    jmp done

done:
    ret
```

定义标号不会写出任何字节，它只是把名称绑定到当前逻辑输出地址。

标号既可以在定义之后引用，也可以在定义之前引用：

```
// finished 在跳转指令之后定义，仍然可以提前引用。
x86.use64();

entry:
    jmp short finished
    mov eax, 1

finished:
    ret
```

不需要额外进行前置声明。这里明确写出的 `short` 属于 x86 指令文本，并且会保留为编码长度约束。

手写控制流、数据名称、入口点以及其他在编写源代码时就已知的名称，都应使用普通静态标号。它最清晰，也能提供最准确的诊断信息。

符号引用与地址回填

带有符号的指令操作数不一定能立即编码。指令大小、目标地址或两者可能取决于尚未完成的布局。

XIRASM 按下面的顺序处理：

1. 源代码定义标号并创建处理器指令片段。
2. 所选后端编码每条指令，并报告仍然缺少具体数值的符号字段。
3. 布局过程为标号和片段分配最终地址。
4. 地址回填过程计算每个符号表达式，并修改已经编码的字段。

因此，向前跳转不需要手工计算地址：

```
// 跳转目标稍后定义，位移由汇编器计算。
x86.use64();

entry:
    jmp target
    nop

target:
    ret
```

源代码通过标号表达意图，具体位移由汇编器负责计算。

只要条件允许，就让指令操作数保留符号表达式。手工相减指令地址或写死位移，会使代码在前面某条指令改变长度后失效。

动态指令文本与动态标号

编译期代码有时需要生成名称，甚至生成整条指令。使用 `isa(text)` 汇编计算得到的指令，使用 `label.define(name)` 定义计算得到的标号：

```
// 根据多个固定部分组成动态标号名称。
const done: string = sym.join("generated_", "done")

// 动态指令和动态标号仍然进入正常的汇编流程。
isa(sym.join("jmp ", done));
label.define(done);
isa("ret");
```

它创建的逻辑结构与下面的静态写法相同：

```
// 这是上一个动态示例所对应的普通静态写法。  
jmp generated_done  
generated_done:  
ret
```

动态生成的指令仍然使用当前目标平台，并参与正常的标号解析和地址回填流程。

`label.define` 接受一个字符串，在当前逻辑地址定义该名称。需要由多个已知部分组成稳定名称时，可以使用 `sym.join`；可复用的生成器每次展开都需要新的内部名称时，可以使用 `sym.unique`。

只有名称确实需要计算时才使用动态标号。不要把 `loop:` 或 `done:` 这样清晰的静态源代码替换成动态字符串。

读取标号地址

`label_addr(label_or_name)` 返回标号对应的逻辑地址。参数既可以是普通标号引用，也可以是包含动态标号名称的字符串：

```
// 定义一个普通标号，并把它的逻辑地址写入输出。  
x86.use64();  
  
entry:  
    ret  
  
dq(label_addr(entry));
```

这个示例先写出一条 `ret` 指令，再写出 `entry` 的逻辑地址。

指令中的符号引用与普通编译期地址查询之间有一个重要区别：

- 处理器指令操作数可以保持符号形式，稍后通过地址回填解析；
- `label_addr(...)` 会在表达式求值时产生一个编译期整数。

如果前面的指令大小、对齐、输出区域或预留空间仍然可能改变最终地址，应当在收尾处理中回填二进制字段，而不是过早固定地址。第 12 章将介绍布局稳定后的查询和 `defer`。

可执行文件和目标文件格式有时需要记录重定位信息，而不是直接写入绝对整数。这种情况下应使用格式接口提供的重定位和导入、导出接口。可执行文件格式指南将详细说明两者的区别。

实用规则

编写普通 XIRASM 源代码时，可以遵循这些默认规则：

- 在源文件开头附近明确选择所需的目标模式；
- 手写指令使用自然的处理器指令写法；
- 处理器指令行末不写分号，编译期调用保留分号；
- 源代码中已经确定的名称使用静态标号；
- 向前引用保持符号形式，让地址回填过程负责计算；
- 只有确实需要计算源代码时才使用 `isa` 和 `label.define`；
- 使用 `target.bits` 判断宽度，使用 `target.isa` 判断后端系列；
- 布局尚未稳定时，将最终二进制地址的回填推迟到收尾处理；
- 如果地址需要由装载程序或链接器调整，应使用格式重定位接口。

下一章将从指令引用转向明确的数据和二进制布局，包括整数写出、字节序列、预留空间、对齐、结构体和联合体。

[返回语言指南](#)

第 9 章：数据与二进制布局

指令只是输出字节的一种来源。文件头、查找表、消息、常量、预留空间和应用程序专用记录，都需要明确的二进制布局。

XIRASM 提供两个互补的层次：

- 数据写出接口负责写出精确的整数、字符串和字节值；
- 二进制复合类型用于描述可复用的结构体和联合体。

对于较短的布局 and 孤立字段，可以直接写出数据。当多个字段共同组成一个具名记录，并且需要把该记录作为整体创建、检查或写出时，应使用复合类型。

输出内容是有序字节序列

数据调用会在当前输出位置追加字节：

```
// 按 1、2、4、8 字节的宽度依次写出整数。  
emit.u8(0x11);  
emit.u16(0x2233);  
emit.u32(0x44556677);  
emit.u64(0x0102030405060708);
```

这些整数接口按小端顺序写出值。生成的字节为：

```
11  
33 22  
77 66 55 44  
08 07 06 05 04 03 02 01
```

每个接口都会检查传入值能否放入指定宽度。如果值超出范围，应改用更宽的接口，而不是让它被静默截断。

对应的简写如下：

简写	宽度	等效用途
<code>db</code>	1 字节	字节值、字符串和 <code>bytes</code> 值
<code>dw</code>	2 字节	小端整数
<code>dd</code>	4 字节	小端整数
<code>dp</code>	6 字节	小端整数
<code>dq</code>	8 字节	小端整数
<code>dt</code>	10 字节	从 <code>u64</code> 零扩展的小端整数
<code>ddq</code>	16 字节	从 <code>u64</code> 零扩展的小端整数
<code>dqq</code>	32 字节	从 <code>u64</code> 零扩展的小端整数
<code>ddqq</code>	64 字节	从 <code>u64</code> 零扩展的小端整数

例如：

```
// 使用简写按固定宽度写出同一组小端整数。
db(0x41);
dw(0x1122);
dd(0x33445566);
dq(0x0102030405060708);
```

这些简写都可以接受多个实参。除 `db` 外，每个实参都必须是整数；`db` 还可以接受字符串和字节序列。

1、2、4、6、8 字节宽度会严格检查范围。当前 Meta 整数为 `u64`，所以 10、16、32、64 字节宽度会进行零扩展。需要精确的宽位模式时应使用 `emit.bytes`。`dt` 只表示 10 字节原始数据，不会引入 `f80` 或 `x87` 浮点类型。

浮点值

浮点写出与整数数据简写保持独立：

```
const scale: f64 = 1.5
const compact: f32 = f32(scale)

emit.f32(compact);
emit.f64(scale * 2.0);
```

`emit.f32` 和 `emit.f64` 分别写出小端 IEEE-754 binary32 和 binary64 编码，实参必须已经具有完全匹配的类型。浮点运算和比较也要求两侧类型一致。字面量、转换和运算结果必须保持有限；溢出、NaN 和

Infinity 会被拒绝，有限下溢与有符号零会被保留。

字符串与字节序列

一次 `db` 调用可以同时组合字节值、字符串和 `bytes` 值：

```
// suffix 保存需要原样接在文本之后的两个字节。  
const suffix: bytes = b"CD"  
  
// 依次写出字节 A、字符串 B、字节序列 CD 和显式终止零。  
db(0x41, "B", suffix, 0);
```

这会写出：

```
41 42 43 44 00
```

字符串和字节序列都会被原样复制。XIRASM 不会自动添加结尾的零字节；如果文件格式、应用程序二进制接口或运行时接口要求终止零，必须显式写出。

`emit.bytes(value)` 用于写出一个字符串或 `bytes` 值：

```
// 从十六进制文本构造精确的四字节签名。  
const signature: bytes = bytes.from_hex("7f454c46")  
  
// 先写出二进制签名，再直接写出四个文本字节。  
emit.bytes(signature);  
emit.bytes("DATA");
```

当一个值表示精确的二进制内容时，应使用 `bytes`。当一个值表示源代码层面的文本，只是恰好需要直接写出时，应使用字符串。

预留空间

`reserve(count)` 会让输出位置前进指定的字节数：

```
// 在两个已初始化字节之间预留两个字节。  
db(0xeb);  
reserve(2);  
db(0xfe);
```

在普通裸二进制文件中，这会写出：

```
eb 00 00 fe
```

预留空间简写会把数量乘以对应的元素宽度：

简写	预留字节数
<code>rb(count)</code>	<code>count</code>
<code>rw(count)</code>	<code>count * 2</code>
<code>rd(count)</code>	<code>count * 4</code>
<code>rp(count)</code>	<code>count * 6</code>
<code>rq(count)</code>	<code>count * 8</code>
<code>rt(count)</code>	<code>count * 10</code>
<code>rdq(count)</code>	<code>count * 16</code>
<code>rqq(count)</code>	<code>count * 32</code>
<code>rdqq(count)</code>	<code>count * 64</code>

```
// 预留两个字节，写出 aa，再预留两个字节并写出 bb。  
rb(2);  
db(0xaa);  
rw(1);  
db(0xbb);
```

这会依次写出两个零字节、`aa`、另外两个零字节和 `bb`。

预留操作表达的是未使用的空间，而不是具有实际意义的已初始化内容。如果后面还有已初始化的输出内容，那么预留范围会在裸二进制文件中成为真实的零填充间隙。如果连续的预留空间位于末尾，则可能只增加逻辑大小，而不占用文件字节。第 11 章会通过输出区域的实际文件游标和潜在文件游标解释这一区别。

填充与对齐

`pad(count, fill)` 会写出指定数量的重复字节：

```
// 在 01 与 02 之间写出三个 aa 填充字节。  
db(1);  
pad(3, 0xaa);  
db(2);
```

结果为：

```
01 aa aa aa 02
```

填充值实参可以省略，默认值为零：

```
// 使用默认填充值写出四个零字节。  
pad(4);
```

`pad_to(position, fill)` 会持续写出填充字节，直到当前输出字节位置达到指定位置：

```
// 当前位置为 2；用 90 填充到位置 6，再写出 33。  
db(0x11, 0x22);  
pad_to(6, 0x90);  
db(0x33);
```

这会写出：

```
11 22 90 90 90 90 33
```

指定位置不能位于当前输出位置之前。

`align(boundary, fill)` 会让输出位置前进到边界的下一个整数倍：

```
// 三个初始字节之后，用 cc 补齐到 8 字节边界。  
db(0x11, 0x22, 0x33);  
align(8, 0xcc);  
db(0x44);
```

这会在 44 之前写出五个 cc 字节。填充值实参默认也是零。

对齐边界必须是非零的二次幂。1、2、4、16 和 4096 等值有效，3 或 24 等值会被拒绝。

应根据意图选择操作：

需求	使用
已知大小的未初始化空间	<code>reserve</code> 或 <code>rb</code> / <code>rw</code> / <code>rd</code> / <code>rp</code> / <code>rq</code> / <code>rt</code> / <code>rdq</code> / <code>rqq</code> / <code>rdqq</code>
精确数量的填充字节	<code>pad</code>
到达已知输出位置	<code>pad_to</code>
到达下一个对齐位置	<code>align</code>

声明二进制结构体

结构体为一组顺序排列的字段指定名称和类型：

```
// 自然布局会按字段类型的对齐要求安排偏移。  
struct NaturalHeader {  
    tag: u8  
    size: u32  
}
```

这是一个自然对齐的结构体。`tag` 从偏移 0 开始。`u32` 字段要求按四字节对齐，因此 `size` 从偏移 4 开始。整个结构体占八个字节：

```
偏移 0: tag  
偏移 1: 填充  
偏移 2: 填充  
偏移 3: 填充  
偏移 4: size  
偏移 5: size  
偏移 6: size  
偏移 7: size
```

自然布局会对齐每个字段，并把最终大小向上取整到结构体的对齐要求。它适合布局应遵循各字段对齐要求的内存记录。

如果需要精确的文件布局，应声明紧凑结构体：

```
// packed 禁止在字段之间和结构体末尾自动插入填充。  
packed struct FileHeader {  
    tag: u8  
    size: u32  
}
```

`FileHeader` 占五个字节。`size` 从偏移 1 开始，紧接在 `tag` 之后，末尾也没有填充。

紧凑布局会移除内部和尾部填充，但类型仍保留其最大字段的对齐属性。紧凑复合类型嵌套在自然布局复合类型中时，外层仍会按照这个对齐属性放置该字段。

文件头、协议记录、指令元数据和其他由外部规范规定的字节布局，通常应使用紧凑布局。原生内存记录通常应使用自然布局。

字段默认值与结构体字面量

结构体的整数字段可以提供编译期默认值：

```

// magic 和 flags 提供默认值, size 由每个实例显式指定。
packed struct Header {
    magic: u16 = 0x5a4d
    flags: u16 = 1
    size: u32
}

// 字面量只覆盖 size, 其余字段沿用声明中的默认值。
const header: Header = Header {
    size: 0x40
}

// 通过普通字段访问读取需要写出的成员。
emit.u16(header.magic);
emit.u32(header.size);

```

这个字面量提供了 `size`，并对 `magic` 和 `flags` 使用默认值。字面量中的字段按名称匹配，而不是按源码中的排列顺序匹配。

每个省略的字段都必须具有默认值。未知字段、重复字段和值类型不正确的字段都会导致源代码错误。

联合体字段不能声明默认值。联合体值必须始终显式选择且只选择一个当前字段。

字段可以使用普通的字段访问语法读取，如示例中的两个写出调用所示。

复合值存在于汇编期间。仅仅声明复合值并不会自动把它写入输出内容。

测量布局

`sizeof(Type)` 返回二进制类型的完整大小：

```

// 自然布局需要为 u32 字段和结构体末尾保留对齐空间。
struct NaturalHeader {
    tag: u8
    size: u32
}

// 紧凑布局保持字段连续排列。
packed struct PackedHeader {
    tag: u8
    size: u32
}

// 同时验证两种布局的总大小和关键字段偏移。
assert(sizeof(NaturalHeader) == 8);
assert(sizeof(PackedHeader) == 5);
assert(offset_of(NaturalHeader, tag) == 0);
assert(offset_of(NaturalHeader, size) == 4);
assert(offset_of(PackedHeader, size) == 1);

```

`offset_of(Type, field)` 返回字段偏移。这两个操作都是编译期表达式，可以用于指令、写出字段、断言和其他布局计算：

```

// 选择 64 位 x86 指令编码。
x86.use64();

// 两个 64 位寄存器槽位组成连续的保存区。
packed struct SaveArea {
    rax: u64
    rcx: u64
}

// 根据结构体大小调整栈指针，避免硬编码保存区字节数。
sub rsp, sizeof(SaveArea)

```

使用符号化的布局计算，可以避免新增字段或更改类型后，源代码其他位置仍残留过期的硬编码大小。

打包并写出结构体值

`pack(value)` 会把复合值转换为 `bytes` 值：

```
// 两个小端 u16 字段共同组成四个连续字节。
packed struct Header {
    magic: u16 = 0x4241
    tail: u16
}

const header: Header = Header {
    tail: 0x4443
}

// 先打包以便比较，再把同一字节序列写入输出。
const encoded: bytes = pack(header)
assert(encoded == b"ABCD");
emit.bytes(encoded);
```

`emit.struct(value)` 会直接完成打包和写出：

```
// 自然布局会在 tag 与 size 之间加入三个零填充字节。
struct NaturalHeader {
    tag: u8 = 0x41
    size: u32 = 0x11223344
}

// 空字面量使用所有字段默认值，并立即写出完整结构体布局。
const header: NaturalHeader = NaturalHeader { }
emit.struct(header);
```

这会写出八个字节，其中自然布局产生的三个填充字节都是零：

```
41 00 00 00 44 33 22 11
```

如果还需要比较、转换、把字节序列保存在集合中，或者把它传给另一个函数，应使用 `pack`。如果只需要立即写出该值，应使用 `emit.struct`。

联合体与当前字段

联合体会让多个不同类型的字段重叠在偏移 0：

```

// Point 的两个坐标以紧凑形式连续存放。
packed struct Point {
    x: u16
    y: u16
}

// raw 与 point 共享同一段四字节存储。
union ValueBits {
    raw: u32
    point: Point
}

// 此字面量选择 point 作为联合体的当前字段。
const coordinates: ValueBits = ValueBits {
    point: Point {
        x: 0x1122,
        y: 0x3344
    }
}

```

联合体的每个字段都从偏移 0 开始。自然布局的联合体以最大字段大小为基础，再按最大的字段对齐要求向上取整。`packed` 联合体则直接使用最大字段的精确大小。

`coordinates` 字面量只选择一个当前字段。联合体字面量既不能省略所有字段，也不能同时初始化多个字段。联合体字段声明不能提供默认值，因为默认值不能表示一次显式的当前字段选择。

联合体还可以嵌套在结构体中：

```

// Point 描述联合体的一种四字节解释方式。
packed struct Point {
    x: u16
    y: u16
}

union ValueBits {
    raw: u32
    point: Point
}

// 紧凑记录把种类字节与联合体内容连续排列。
packed struct Record {
    kind: u8
    value: ValueBits
}

// 此实例选择 raw 作为嵌套联合体的当前字段。
const record: Record = Record {
    kind: 1,
    value: ValueBits {
        raw: 0xaabbccdd
    }
}

// 嵌套字段路径会累计外层和内层字段的偏移。
assert(offset_of(Record, value) == 1);
assert(offset_of(Record, value.point.y) == 3);
emit.struct(record);

```

如示例中的两个断言所示，`offset_of` 支持嵌套字段路径。

第二个结果把 `value` 在 `Record` 中的偏移，与 `y` 在 `Point` 中的偏移相加。

选择布局方式

以下情况适合直接写出整数和字节：

- 布局只有少数字段；
- 字段只写出一次，之后不再需要名称；
- 源代码紧密对应某个外部表格，显式调用比类型声明更清晰。

以下情况适合使用紧凑结构体和联合体：

- 精确的二进制记录会被重复使用；
- 字段名称可以提高可读性；
- 其他计算应由 `sizeof` 和 `offset_of` 驱动；

- 值需要默认值、嵌套、比较或转换为 `bytes`。

以下情况适合使用自然布局结构体：

- 记录表示对齐的内存，而不是序列化后的文件布局；
- 填充是有意保留的，并且应遵循字段对齐要求。

对于由外部规范定义的布局，始终应使用 `sizeof` 和 `offsetof` 断言进行验证。这些断言会把格式假设变成可执行检查，并让后续修改更加稳妥。

下一章将介绍模块与文件：通过 `include` 和 `import` 复用源代码，以及在汇编期间读取外部文本、字节、JSON 和 TOML。

[返回语言指南](#)

第 10 章：模块与文件

汇编项目通常很快就会超出单个源文件的规模。指令辅助函数、二进制记录定义、生成的表格、配置和嵌入数据往往由不同部分负责，也会因为不同原因发生变化。

XIRASM 提供两套彼此独立的文件加载方式：

- `include` 和 `import` 加载 XIRASM 源代码；
- `fs`、`json` 和 `toml` 函数加载供编译期代码使用的数据。

明确区分这两种用途，可以让项目更容易理解。源文件提供声明或输出操作；数据文件产生可以由源代码检查、转换和写出的值。

导入源代码模块

`import(path)` 对同一个源文件最多求值一次：

```
// 同一个模块导入两次，模块内容也只会求值一次。
import("support.inc");
import("support.inc");

emit_word(0x1234);
```

假设 `support.inc` 包含：

```
fn emit_word(value: u64) {
    emit.u16(value);
}
```

两次导入最终指向同一个文件，因此函数只声明一次。示例会写出：

```
34 12
```

定义可复用函数、常量、结构体或其他名称的文件应使用 `import`。即使多个依赖路径都导入同一模块，也不会重复它的声明或输出操作。

`import` 语句必须位于源文件顶层，不能放进局部执行会让模块加载变成条件行为的代码块：

```
if target.bits == 64 {
    import("x64-support.inc");
}
```

需要选择模块时，应在源文件顶层完成；与目标平台相关的行为则放进模块内部。

在当前位置包含源代码

`include(path)` 每次执行到该语句时，都会重新求值指定的源文件：

```
// 同一个包含文件执行两次，因此其中的输出操作也执行两次。  
include("inline.inc");  
include("inline.inc");
```

如果 `inline.inc` 包含：

```
emit.u8(0xaa);
```

输出为：

```
aa aa
```

只有确实需要重复执行时才使用 `include`。常见用途包括：

- 在当前输出位置插入生成的数据；
- 共享一小段写出数据的语句；
- 从计算得到的路径选择源代码片段；
- 多次应用同一个源代码模板。

由于包含操作会重复执行，同一个文件如果重复声明函数、常量或类型，可能产生名称重复错误。这类文件通常属于模块，应改用 `import` 加载。

源代码路径如何解析

相对源代码路径以包含 `include` 或 `import` 语句的文件所在目录为起点进行解析。

假设项目结构如下：

```
project/  
  main.asm  
  tables/  
    records.inc  
  shared/  
    constants.inc
```

`tables/records.inc` 可以这样加载旁边 `shared` 子目录中的文件：

```
import("shared/constants.inc");
```

这个路径相对于 `records.inc`，不一定相对于进程的工作目录，也不一定相对于最初的入口源文件。

如果在当前源文件旁边找不到相对路径指向的文件，XIRASM 会继续检查项目的 `include` 目录，然后检查安装位置的包含目录。这样，项目可以覆盖或补充已经安装的库，而不必在源代码中写入特定机器的绝对路径。

XIRASM 也接受绝对路径，但可移植项目通常应使用相对于源文件的路径，或使用已经配置的包含目录。

源代码加载不能形成循环。如果一个文件正在求值，它就不能直接或间接再次包含或导入自身。

选择 `import` 还是 `include`

应根据执行方式选择，而不是根据文件扩展名选择：

需要完成的工作	使用
只定义一次可复用名称	<code>import</code>
共享编译期函数或类型库	<code>import</code>
在当前输出位置执行源代码片段	<code>include</code>
多次执行同一个源文件	<code>include</code>
避免依赖链重复声明	<code>import</code>

一种实用的项目约定是：

- 模块定义可以复用的名称和操作；
- 包含片段完成与当前位置有关的工作；
- 入口源文件选择目标平台并汇编最终输出。

检查和读取数据文件

`fs.exists(path)` 检查是否能够解析指定的数据文件：

```
// 确认文件存在，再把它的全部字节写入输出。  
assert(fs.exists("payload.bin"));  
emit.bytes(fs.read_bytes("payload.bin"));
```

数据路径使用与源代码加载相同的相对路径规则。如果把这段代码移动到模块中，`payload.bin` 就会相对于该模块进行解析。

`fs.read_text(path)` 把整个文件读取为 `string`：

```
// 读取完整文本，检查内容后按原始字节写出。  
const banner: string = fs.read_text("banner.txt");  
assert(contains(banner, "XIRASM"));  
emit.bytes(banner);
```

如果 `banner.txt` 包含：

```
XIRASM
```

XIRASM 会写出文件中的精确字节，不会自动在末尾添加零字节。

`fs.read_bytes(path)` 把整个文件读取为 `bytes`。图像、已经编码的表格、预先生成的记录以及其他二进制载荷通常都适合使用这个函数。

如果文件字节只需要直接写入输出，可以使用 `emit.file(path)`，不必先创建中间绑定。

`emit.file(path, offset, count)` 会写出一个精确范围。两种形式都复用 `fs.read_bytes` 的源文件相对路径解析和边界检查，并且不能在 `late_layout` 或 `defer` 中使用。

读取一段字节

带三个参数的 `fs.read_bytes` 可以读取一个有明确边界的范围：

```
// 从偏移一开始读取两个字节。  
const middle: bytes = fs.read_bytes("payload.bin", 1, 2);  
assert(len(middle) == 2);  
emit.bytes(middle);
```

第二个参数是从零开始计算的文件偏移，第三个参数是需要返回的字节数。

如果 `payload.bin` 包含：

```
10 20 30 40
```

示例会写出：

```
20 30
```

请求的完整范围必须存在。如果读取范围越过文件末尾，XIRASM 会报告错误，而不是悄悄缩短结果。

加载 JSON

`json.file(path)` 一次完成 JSON 文件的读取和解析：

```
// 读取配置，并从映射和列表中取出所需字段。
const config: map = json.file("config.json");
const values: list = map.get(config, "values");

assert(map.get(config, "enabled"));
emit.bytes(map.get(config, "name"));
emit.u8(map.get(config, "bits"));
emit.u8(list.get(values, 0));
emit.u8(list.get(values, 1));
```

输入文件如下：

```
{
  "name": "XR",
  "bits": 64,
  "enabled": true,
  "values": [1, 2]
}
```

输出为：

```
58 52 40 01 02
```

`json.parse(value)` 用来解析已经保存在 `string` 或 `bytes` 值中的 JSON：

```
// 先按文本读取文件，再显式解析其中的 JSON。
const raw: string = fs.read_text("config.json");
const config: map = json.parse(raw);
emit.u8(map.get(config, "bits"));
```

JSON 对象转换成映射，数组转换成列表，字符串和布尔值保持对应的编译期类型，非负整数转换成整数值。JSON 的 `null` 转换成 `void`。

当前编译期数据模型不支持浮点数和负整数，也会拒绝对象中的重复键和语法错误。

加载 TOML

`toml.file(path)` 为 TOML 提供相同的直接处理方式：

```
// 读取 TOML 配置，并从嵌套映射中取得目标宽度。
const config: map = toml.file("project.toml");
const target: map = map.get(config, "target");

emit.bytes(map.get(config, "name"));
emit.u8(map.get(target, "bits"));
```

输入文件如下：

```
name = "XR"

[target]
bits = 64
```

输出为：

```
58 52 40
```

`toml.parse(value)` 用来解析保存在 `string` 或 `bytes` 值中的 TOML：

```
// 先读取原始文本，再显式解析 TOML。
const raw: string = fs.read_text("project.toml");
const config: map = toml.parse(raw);
const target: map = map.get(config, "target");

assert(map.get(target, "bits") == 64);
emit.u8(0x40);
```

TOML 表转换成映射，数组转换成列表，字符串、布尔值和非负整数转换成对应的编译期值。当前数据转换不接受浮点值和时间戳值。

分开管理配置和二进制载荷

结构化配置和原始二进制数据解决的是不同问题。

在这些情况下使用 JSON 或 TOML：

- 字段需要通过名称访问；
- 输入文件需要由用户直接编辑；
- 同一份输入用于生成多个值；
- 需要检查输入并根据条件生成内容。

在这些情况下使用 `fs.read_bytes`：

- 文件内容本身就是所需的二进制表示；
- 必须逐字节保持原样；
- 只需要文件中一个有明确边界的范围；
- 解析不会提供有用的结构。

如果文本本身就是载荷，或希望明确选择后续解析方式，则使用 `fs.read_text`。

实用的模块组织方式

中等规模的项目可以采用下面的结构：

```
project/  
  main.asm  
  include/  
    records.inc  
    encoding.inc  
  data/  
    config.toml  
    payload.bin
```

`main.asm` 导入可复用定义：

```
import("records.inc");  
import("encoding.inc");
```

这些模块可以读取相对于自身的数据，也可以使用 `../data/config.toml` 这样的明确相对路径。

应当让模块行为保持清晰：

- 声明库优先使用 `import`；
- 只有明确需要重复执行时才多次调用 `include`；
- 文件路径相对于负责使用它的源文件；
- 使用 `assert` 和 `fs.exists` 检查必要数据；
- 结构化文件只解析一次，并复用得到的映射或列表；
- 如果只需要大型文件的一部分，就使用有边界的二进制读取。

下一章将介绍输出区域和虚拟数据，说明 XIRASM 如何区分逻辑地址、文件中的实际字节、预留空间和临时布局区域。

[返回语言指南](#)

第 11 章：输出区域与虚拟数据

汇编器经常需要分别回答两个问题：

- 1. 一个值最终位于哪个地址？
- 2. 它的字节会出现在输出文件的哪个位置，或者根本不写入文件？

对于简单的裸二进制文件，这两个位置通常都从零开始，并且同步向前移动。结构更复杂的二进制文件则需要把它们分开处理。例如，一段内容可以装载到较高的地址，却仍然位于文件开头附近；预留空间可以扩大地址范围，而不增加文件字节；临时生成的数据也可以完全不进入最终文件。

XIRASM 使用输出区域描述这些情况。

逻辑位置与文件位置

每个普通输出区域都包含以下信息：

- **起始地址**：作为标号地址的计算基准；
- **文件偏移**：决定该区域的字节放在文件中的什么位置；
- **逻辑大小**：包括预留空间在内的地址范围；
- **文件大小**：实际形成并写入文件的字节数。

这些信息彼此独立。修改起始地址不会自动在文件中插入填充字节，修改文件偏移也不会改变标号的地址。

常用的位置查询如下：

查询	含义
<code>region_base()</code>	当前输出区域的起始地址
<code>here()</code>	当前逻辑地址
<code>file_offset()</code>	当前实际文件位置
<code>file_cursor_real()</code>	当前已经实际写入的文件末端位置
<code>file_cursor_potential()</code>	假设当前所有预留空间都写入文件时的末端位置
<code>tail_reserve_size()</code>	当前区域尾部尚未实际写入文件的预留字节数

这些查询在构造自定义布局或可复用的格式辅助接口时最有用。普通汇编代码通常只需要使用标号和 `here()`。

设置起始地址

`origin(address)` 设置当前输出区域的地址计算基准：

```
// 把当前输出区域的逻辑起点设为 0x4000。
origin(0x4000);

start:
// 在起始地址处写出一个字节。
emit.u8(0xaa);

// 检查区域起点、标号地址、当前位置和文件位置。
assert(region_base() == 0x4000);
assert(label_addr("start") == 0x4000);
assert(here() == 0x4001);
assert(file_offset() == 1);
```

输出文件只包含一个字节：

```
aa
```

起始地址影响的是标号地址，而不是文件中的实际位置。在简单的裸二进制区域中，应当先设置起始地址，再定义标号或写出内容。之后再次调用 `origin`，会更新整个当前区域的地址计算基准，而不是创建一个新的独立区域。

创建明确的输出区域

`region.begin(name, origin, file_offset)` 创建一个普通输出区域，并同时指定逻辑起始地址和文件偏移：

```
// 头部的逻辑地址从 0x1000 开始，文件位置从 0 开始。
region.begin("header", 0x1000, 0);

header:
emit.bytes(b"HD");

// 载荷使用独立的逻辑地址，并从文件偏移 0x10 开始。
region.begin("payload", 0x2000, 0x10);

payload:
emit.bytes(b"DATA");

// 两个标号分别使用各自输出区域的起始地址。
assert(label_addr("header") == 0x1000);
assert(label_addr("payload") == 0x2000);
```

头部占用文件偏移 `0` 和 `1`。载荷从文件偏移 `0x10` 开始。在裸二进制输出中，两者之间没有写入内容的部分会用零填充：

```
48 44 00 00 00 00 00 00 00 00 00 00 00 00 00 00
44 41 54 41
```

创建另一个输出区域时，当前写入位置会切换到新区域。输出区域不存在词法上的嵌套关系，也不需要使⤵用 `end` 语句结束。

区域名称只用于描述源代码中的布局。它本身不会在任何可执行文件或目标文件中创建节记录。此类记录由可执行文件格式指南中介绍的格式接口负责生成。

当逻辑起始地址和文件偏移都已经确定时，可以使用明确的输出区域。需要连续构造多个区域时，`output.section` 和 `output.org` 通常更稳妥，因为它们会根据当前区域推导下一个文件偏移。

实际与潜在文件位置

写出已经初始化的字节时，实际文件位置和潜在文件位置会一起向前移动：

```
// 写出一个实际存在于文件中的字节。
emit.u8(0xaa);

// 实际位置和潜在位置都前进到 1，尾部没有预留空间。
assert(file_cursor_real() == 1);
assert(file_cursor_potential() == 1);
assert(tail_reserve_size() == 0);
```

预留空间会推进潜在文件位置和逻辑地址，但不会立即写入实际文件字节：

```
// 先写出一个字节，再预留三个尚未实际写入的字节。
emit.u8(0xaa);
reserve(3);

// 逻辑位置已经前进四字节，实际文件仍只有一个字节。
assert(here() == 4);
assert(file_cursor_real() == 1);
assert(file_cursor_potential() == 4);
assert(tail_reserve_size() == 3);
```

如果预留空间之后又出现已经初始化的输出内容，这段预留范围就会成为文件中间的间隔：

```
// 后续字节使前面的预留范围成为文件中的实际间隔。
emit.u8(0xaa);
reserve(3);
emit.u8(0xbb);

// 中间间隔已经实际形成，两个文件位置重新一致。
assert(file_cursor_real() == 5);
assert(file_cursor_potential() == 5);
assert(tail_reserve_size() == 0);
```

此时文件内容为：

```
aa 00 00 00 bb
```

这种延后决定是否写入文件的机制，使同一个 `reserve` 调用既能表示普通的零填充间隔，也能表示不占用文件空间的尾部预留空间。

去除尾部预留空间

`output.section(name, origin)` 从当前的**实际文件位置**开始下一个输出区域：

```
// 第一个区域包含一个实际字节和三个尾部预留字节。
emit.u8(0x41);
reserve(3);

// 新区域紧接最后一个实际字节，尾部预留空间不进入文件。
output.section("next", 0x2000);
emit.u8(0x42);
```

尾部预留空间会增加第一个区域的逻辑大小，但下一个区域会紧接在最后一个已经初始化的字节之后。输出结果为：

当一段内存范围需要在程序运行时存在，却不应占用对应文件区域末尾的空间时，这种方式很有用。

`output.section` 接受区域名称和新的逻辑起始地址。它从实际文件位置推导文件偏移，因此只会去除尚未实际写入的尾部预留空间；已经位于文件中间的间隔仍会保留。

保留潜在间隔

`output.org(name, origin)` 从当前的**潜在文件位置**开始下一个输出区域：

```
// 第一个区域的潜在长度为四字节。
emit.u8(0x41);
reserve(3);

// 新区域从潜在位置开始，因此保留前面的三字节间隔。
output.org("next", 0x2000);
emit.u8(0x42);
```

由于预留范围之后出现了已经初始化的输出内容，这段范围会成为文件的一部分：

```
41 00 00 00 42
```

两个接口的区别可以准确概括为：

接口	下一个文件偏移
<code>output.section</code>	使用实际文件位置，去除当前区域尾部未写入的预留空间
<code>output.org</code>	使用潜在文件位置，保留当前区域的预留范围

无论使用哪一个接口，新区域的逻辑起始地址都可以独立于文件位置进行选择。

按文件对齐结束输出区域

`region.file_align(alignment)` 对齐当前区域最终的实际文件大小：

```
// 第一个区域从逻辑地址 0x1000、文件偏移 0 开始。
region.begin("first", 0x1000, 0);

// 三个实际字节之后还有十三个尾部预留字节。
emit.bytes(b"ABC");
reserve(13);

assert(here() == 0x1010);
assert(file_cursor_real() == 3);
assert(file_cursor_potential() == 16);

// 去除尾部预留空间，再把实际文件大小向上对齐到八字节。
region.file_align(8);

// 第二个区域从已经对齐的文件偏移 8 开始。
region.begin("second", 0x2000, 8);
emit.u8(0x5a);
```

对齐时会先去除尾部预留空间，再把三个实际字节占用的文件范围向上取整到八字节边界。第二个区域从文件偏移 8 开始：

```
41 42 43 00 00 00 00 00 5a
```

文件对齐不会推进逻辑大小。在这个示例中，第一个区域描述的逻辑范围仍然是十六字节。

对齐值必须是非零的二次幂。调用 `region.file_align` 后，当前区域不能再写入实际文件内容；继续写出字节之前，必须先创建另一个输出区域。

它与 `align` 的作用不同：

- `align` 推进逻辑位置；当后续输出使对齐空间需要进入文件时，会写出填充字节；
- `region.file_align` 只对最终文件大小向上取整，不改变逻辑地址。

虚拟输出

虚拟区域是在汇编期间使用的独立输出区域。可以检查或转换其中的字节、标号和指令，但它的内容不会自动复制到最终的裸二进制文件。

`virtual.begin(origin)` 使用明确的逻辑地址创建虚拟区域：

```
// 在逻辑地址 0x3000 创建一个临时虚拟区域。
virtual.begin(0x3000);

table:
emit.u32(0x11223344);
// 读取原值、逐位变换后，再写回同一位置。
store.u32(table, load.u32(table) ^ 0x01010101);
const encoded: bytes = load.bytes(table, 4)

virtual.end();

// 只有显式复制出的字节才会进入主输出。
emit.bytes(encoded);
```

临时区域最初包含：

```
44 33 22 11
```

完成变换后，复制到主输出中的字节为：

```
45 32 23 10
```

原来的虚拟区域本身不会增加任何文件字节。

虚拟区域可以包含普通的数据写出、预留空间、对齐、标号和处理器指令。`load.*` 和 `store.*` 可以像操作普通区域一样操作其中生成的字节。每个 `virtual.begin` 都必须与一个 `virtual.end` 配对。

以当前地址作为虚拟区域起点

省略起始地址参数时，虚拟区域会从外围区域的当前逻辑地址开始：

```
// 主输出从逻辑地址 0x4000 开始，并先写出一个字节。
origin(0x4000);
emit.u8(0xaa);

// 省略参数，使虚拟区域沿用当前逻辑地址 0x4001。
virtual.begin();

scratch:
emit.u16(0x1234);
const copied: bytes = load.bytes(scratch, 2)

virtual.end();

// 显式把临时生成的两个字节追加到主输出。
emit.bytes(copied);
```

`scratch` 的逻辑地址是 `0x4001`，但虚拟区域中的字节不会替换外围区域的内容，也不会推进外围区域的位置。执行 `virtual.end` 后，主输出会从暂停的位置继续。最终文件包含：

```
aa 34 12
```

如果临时编码中的标号需要使用与目标位置相同的地址计算基准，并且稍后可能把这些字节复制到该目标位置，这种写法会很方便。

布局稳定后的区域信息

在普通写出过程中，实际文件位置和潜在文件位置反映当前状态。只有布局稳定后，才能取得区域的最终大小。

只能在收尾处理中使用的查询如下：

查询	对包含指定地址的区域返回
<code>region_file_offset(address)</code>	区域的起始文件偏移
<code>region_file_size(address)</code>	最终实际文件大小
<code>region_logical_size(address)</code>	最终逻辑大小

这些查询与写出过程中的当前位置查询相互独立。收尾处理可以使用它们回填文件头，或者检查自定义格式中记录的大小是否与最终布局一致。第 12 章将介绍收尾处理阶段及其限制。

如何选择合适的接口

以下情况适合只设置起始地址：

- 一个连续的裸二进制输出需要使用非零的地址计算基准；
- 文件位置和逻辑位置在其他方面保持同步。

以下情况适合使用 `output.section`：

- 下一个连续区域应当紧接已经实际写入的文件字节；
- 尾部预留空间应当增加内存大小，但不占用文件空间。

以下情况适合使用 `output.org`：

- 下一个连续区域应当从潜在文件位置继续；
- 后续已初始化数据之前的预留空间必须保留在文件中。

以下情况适合使用 `region.begin`：

- 逻辑起始地址和实际文件偏移都已经明确；
- 正在构造自定义二进制布局或较复杂的格式辅助接口。

以下情况适合使用虚拟区域：

- 需要临时汇编、测量、读取或转换数据；
- 临时字节不能自动进入主输出。

使用这些接口时，应当始终明确区分两套位置：

- 标号和 `here()` 描述逻辑地址；
- 实际文件位置只计算已经实际写入的字节；
- 潜在文件位置还包括当前尾部预留空间；
- 区域的最终信息只能在布局稳定后的收尾处理阶段使用。

下一章将介绍收尾处理：如何读取稳定的布局信息、修改已有字节、计算校验和，以及在不改变布局的前提下检查完整输出。

[返回语言指南](#)

第 12 章：收尾处理

二进制文件中的许多字段，在第一次写出时还无法得到最终值，例如：

- 后续数据的大小；
- 另一个输出区域在文件中的位置；
- 一段预留空间最终占用的逻辑大小；
- 根据已编码指令和已完成地址回填计算出的校验和；
- 文件头之后的源代码不断累加得到的数量。

XIRASM 使用明确的收尾阶段处理这些情况，不会为了等待数值趋于稳定而反复执行完整源文件。

可以使用两种不同的工具：

- `late_layout` 在输出布局固定之前执行一次受限操作，允许添加会改变布局的内容；
- `defer` 在最终布局稳定后读取并修改已有字节，但不能再改变布局。

大多数由用户编写的字段回填都应放在 `defer` 中。

为回填预留字段

通常按照下面的顺序编写：

1. 先写出一个宽度固定的占位值；
2. 按正常顺序写出实际数据；
3. 在 `defer` 中把最终值回填到占位位置。

```
// 先用两个字节为数据大小预留位置。  
size_field:  
emit.u16(0);  
  
// 写出实际数据，并用前后标号确定它的长度。  
payload:  
emit.bytes(b"ABC");  
payload_end:  
  
defer {  
    // 布局稳定后，把数据长度回填到预留字段。  
    store.u16(size_field, payload_end - payload);  
}
```

最终输出为：

```
03 00 41 42 43
```

占位字段从一开始就占用了两个字节，因此收尾处理只会修改这两个字节的值，不会插入新字节，也不会移动后面的数据。

顶层收尾块可以写在它所引用的标号之前：

```
defer {  
    // 这些标号可以在收尾块之后定义，最终执行时会得到稳定地址。  
    store.u16(size_field, payload_end - payload);  
}  
  
// 为数据长度预留两个字节。  
size_field:  
emit.u16(0);  
  
// 随后写出实际数据。  
payload:  
emit.bytes(b"ABC");  
payload_end:
```

等到最终输出布局可用时，收尾块引用的标号都会得到解析。

读取最终字节

`load.u8`、`load.u16`、`load.u32` 和 `load.u64` 从已经实际写入输出的字节中读取小端整数。
`load.bytes(address, count)` 返回一段字节序列。

读取接口可以与写入和断言配合使用：

```

// 让本区域的逻辑地址从 0x4000 开始。
origin(0x4000);

// 预留两个 32 位文件头字段。
header:
emit.u32(0);
emit.u32(0);

// 写出正文和一个稍后替换的尾部标记。
body:
emit.bytes(b"ABCD");
tail:
emit.bytes(b"????");
image_end:

defer {
    // 回填正文到输出末尾的长度，以及正文相对区域起点的位置。
    store.u32(header, image_end - body);
    store.u32(header + 4, body - region_base());
    store.bytes(tail, b"OK!!");

    // 读取最终字节，确认回填结果符合预期。
    assert(load.u32(header) == 8);
    assert(load.u32(header + 4) == 8);
    assert(load.bytes(tail, 4) == b"OK!!");
}

```

完成后的字节为：

```
08 00 00 00 08 00 00 00 41 42 43 44 4f 4b 21 21
```

`store.bytes` 可以接收字符串或 `bytes` 值。整数写入接口会拒绝超出目标宽度表示范围的值。

所有读取和写入都会检查范围。收尾块只能访问最终输出文件中确实存在的字节。经过裁剪的预留尾部虽然仍有逻辑地址，但那里没有可供修改的实际字节，因此不能作为回填位置。

计算校验和

收尾块可以声明局部值、修改局部可变绑定，并使用 `while` 循环：

```

// 使用固定逻辑起点，便于按标号读取数据。
origin(0x5000);

// 先为 16 位校验和预留位置。
checksum:
emit.u16(0);

// 写出参与校验和计算的数据。
payload:
emit.bytes(b"ABCD");
payload_end:

defer {
    // 逐字节读取最终数据并累加。
    let cursor = payload
    let sum = 0

    while cursor < payload_end {
        sum = sum + load.u8(cursor)
        cursor = cursor + 1
    }

    // 回填校验和，并立即检查写入结果。
    store.u16(checksum, sum);
    assert(load.u16(checksum) == 266);
}

```

校验和是 266，即 0x010a，因此输出为：

```
0a 01 41 42 43 44
```

收尾块中的循环与普通编译期循环使用相同的迭代次数上限。目前 `defer` 中不能使用 `for`；扫描字节时应使用边界明确的 `while` 循环。

使用稳定的区域信息

第 11 章介绍的最终区域查询可以在 `defer` 中使用：

```
// 创建逻辑起点为 0x5000、文件起点为 0 的输出区域。
region.begin("payload", 0x5000, 0);

// 为区域的文件大小和逻辑大小预留字段。
file_size_field:
emit.u32(0);
logical_size_field:
emit.u32(0);

// 写出一个实际字节，再追加三个只占逻辑空间的预留字节。
body:
emit.u8(0xaa);
reserve(3);

defer {
    // 布局稳定后查询正文所在区域，并回填最终大小。
    store.u32(file_size_field, region_file_size(body));
    store.u32(logical_size_field, region_logical_size(body));

    // 文件大小不包含未实际写出的预留尾部，逻辑大小包含它。
    assert(region_file_offset(body) == 0);
    assert(region_file_size(body) == 9);
    assert(region_logical_size(body) == 12);
}
```

预留尾部计入逻辑大小，但不计入文件大小：

```
09 00 00 00 0c 00 00 00 aa
```

这些接口返回包含指定逻辑地址的输出区域信息：

- `region_file_offset(address)` 返回该区域在文件中的起始位置；
- `region_file_size(address)` 返回该区域最终实际写入文件的大小；
- `region_logical_size(address)` 返回该区域占用的完整地址空间大小。

这些查询必须等布局稳定后才能得到结果，因此在初次写出内容时，不能把它们当作实时位置查询使用。

调用返回值函数

返回值函数适合封装不产生输出、只进行计算的最终处理：

```
// 计算不小于 value 且满足指定对齐要求的数值。
fn align_up(value: u64, alignment: u64) -> u64 {
    return ((value + alignment - 1) / alignment) * alignment;
}

// 为对齐后的数据大小预留字段。
size_field:
emit.u32(0);

payload:
emit.bytes(b"ABC");
payload_end:

defer {
    // 把三字节数据向上对齐到八字节，并回填计算结果。
    store.u32(size_field, align_up(payload_end - payload, 8));
    assert(load.u32(size_field) == 8);
}
```

最终输出为：

```
08 00 00 00 41 42 43
```

这个函数只计算表达式，不会自行写出或修改任何字节。

可复用的回填过程函数

过程函数可以登记一个收尾块，并让该收尾块保存调用参数：

```
// 登记一个稍后写入 16 位整数的收尾块。
fn patch_u16(address: u64, value: u64) {
    defer {
        // address 和 value 保存的是调用过程函数时传入的值。
        store.u16(address, value);
    }
}

// 先写出占位字段，再登记对应的回填操作。
field:
emit.u16(0);

patch_u16(field, 0x1234);
```

过程函数调用发生在正常处理源代码时。传入的参数值会为延后的代码块保存下来，之后产生：

这种写法适合构造小型、参数类型明确的回填辅助函数。过程函数作用域中的参数按值保存；顶层收尾块中的直接表达式则可以一直保留标号等符号关系，直到最终执行时再求值。

不能在 `defer` 内部调用这个过程函数。过程函数必须提前调用，由它登记稍后执行的收尾块。

收尾块中的流程控制

`defer` 代码块可以包含：

- `const` 和 `let` 声明；
- 对局部 `let` 绑定赋值；
- `if` 和 `else` ；
- `while` ；
- `store.u8`、`store.u16`、`store.u32`、`store.u64` 和 `store.bytes` ；
- `assert`、`print`、`warn` 和 `err`。

这些语句中的表达式可以使用普通的纯计算运算符、返回值函数、标号、`load.*` 接口和稳定的区域信息。

每个收尾块都有独立的局部作用域。在一个收尾块中声明的局部名称，不能在另一个收尾块中使用。

下面的操作会改变布局，因此会被拒绝：

```
defer {
    emit.u8(0x22);
}
```

处理器指令、标号、区域切换、对齐、预留空间、嵌套收尾块、函数声明和源文件加载也受到相同限制。

收尾块的执行顺序

延后执行的代码块按照登记顺序运行：

```
// 先写出一个稍后会被连续修改的字节。
emit.u8(0);

defer {
    // 第一个收尾块把初始值改为 1。
    store.u8(0, 1);
}

defer {
    // 第二个收尾块能看到前一次修改，因此最终写入 2。
    store.u8(0, load.u8(0) + 1);
}
```

第二个代码块会读到第一个代码块写入的结果，因此输出为：

02

应当有意识地利用这个顺序。如果两个收尾块修改同一段字节，后登记的代码块会看到先登记代码块的修改结果。

收尾块在以下工作全部完成后运行：

1. 正常处理源代码；
2. 编码处理器指令；
3. 执行后期布局操作；
4. 解析地址回填项；
5. 完成最终布局并生成实际字节；
6. 把地址回填结果写入输出。

因此，收尾块看到的就是即将写入文件的准确字节，包括已编码的指令和已经解析的引用。

必须延后执行的布局操作

有时必须等主源文件登记完所有内容后，才能追加会实际写入文件的字节。这种情况应使用

`late_layout`：

```
// 正常源代码先写出第一个字节。
emit.u8(0x10);

late_layout {
    // 后期布局块按照登记顺序追加实际字节。
    emit.u8(0x20);
}

late_layout {
    emit.u8(0x30);
}
```

后期布局块只执行一次，并按照登记顺序从默认输出区域末尾开始工作。这个示例产生：

```
10 20 30
```

后期布局会在地址回填项解析和最终布局之前完成。它写出的字节会参与最终布局 and 实际输出。

后期布局可以让预留尾部成为实际字节

由于 `late_layout` 仍然可以改变布局，后来追加的已初始化字节会让之前位于末尾的预留空间变成中间空隙：

```
// 写出一个字节，并在逻辑地址中预留三个字节。
emit.u8(0xaa);
reserve(3);

late_layout {
    // 在预留空间之后追加实际字节，使中间空隙进入输出文件。
    emit.u8(0xbb);
}
```

最终输出为：

```
aa 00 00 00 bb
```

只有在预留范围本来就应当成为文件实际内容时，才使用这种行为。如果尾部预留空间不应占用文件字节，应先切换到另一个区域，再登记后续的已初始化输出。

组合后期布局与最终回填

收尾块可以看到后期布局阶段追加的字节：

```
// 为整个区域的最终逻辑大小预留字段。
size_field:
emit.u32(0);

// 正常源代码先写出前两个数据字节。
emit.bytes(b"AB");

late_layout {
    // 后期布局再追加两个会参与最终大小计算的字节。
    emit.bytes(b"CD");
}

defer {
    // 根据稳定后的区域大小回填字段，并检查实际文件大小。
    store.u32(size_field, region_logical_size(size_field));
    assert(region_file_size(size_field) == 8);
}
```

稳定后的区域包含四字节字段和四个数据字节：

```
08 00 00 00 41 42 43 44
```

两种工具的职责应当明确分开：

- `late_layout` 创建必须参与布局的字节；
- `defer` 查看已经形成的布局，并修改现有字段。

后期布局的限制

`late_layout` 有意采用比普通源代码更严格的规则。它允许使用布局和输出接口、诊断接口以及 `if` 选择，可以完成以下操作：

- 写出整数、字节和结构体；
- 预留空间、补齐字节和执行对齐；
- 切换输出区域或对输出区域进行对齐；
- 打开和关闭虚拟区域；
- 修改模块中已有的字节。

它不能声明标号、写出处理器指令、声明局部值、执行循环、定义函数、加载源模块，也不能登记嵌套的后期布局块或收尾块。

后期布局只执行一次。它不是一种隐式的多轮处理机制，也不应用来让不稳定的值反复计算直至趋于一致。

选择正确的阶段

如果字节可以按照正常的源代码顺序写出，就直接使用普通源代码。

在以下情况中使用 `late_layout`：

- 必须等主源文件处理完毕后，再追加实际字节或输出区域；
- 这些字节必须影响最终位置和大小；
- 一次受限的后期布局操作就足以完成工作。

在以下情况中使用 `defer`：

- 宽度固定的占位字段需要最终值；
- 校验和或验证操作需要读取完整的最终字节；
- 需要最终的区域大小或文件位置；
- 必须修改已有字节，同时保持布局不变。

不要使用 `defer` 创建尚不存在的空间。应当在收尾阶段之前写出或预留所需空间，然后只修改这段已经存在的范围。

下一章进入第三部分，介绍裸二进制文件和自定义二进制文件，并把标号、结构体、输出区域、后期布局与收尾处理组合成完整的文件格式。

[返回语言指南](#)

第 13 章：裸二进制文件与自定义二进制格式

XIRASM 默认写出裸二进制文件。输出文件只包含指令、数据声明、填充和输出区域实际产生的字节；除非源代码明确构造相应内容，否则不会自动加入操作系统所需的文件头。

这种直接构造字节的方式适合：

- 引导扇区和固件映像；
- 只读存储器数据表和嵌入式资源；
- 协议消息和测试向量；
- 紧凑的资源容器；
- 项目专用的二进制格式；
- 由其他程序装入的小型机器码片段。

裸二进制文件仍然可以具有丰富的内部结构。标号描述逻辑位置，结构体描述记录，输出区域区分逻辑地址与文件位置，收尾处理则可以计算依赖完整文件布局的字段。

写出原始机器码

一个裸二进制文件可以完全由自然书写的处理器指令组成：

```
// 选择 64 位 x86 指令编码，输出文件只包含下面两条指令的机器码。
x86.use64();

entry:
    // 把数值 42 写入 eax，然后返回给调用者。
    mov eax, 42
    ret
```

输出内容为：

```
b8 2a 00 00 00 c3
```

其中没有可执行文件头、入口点字段或加载元数据。`entry` 只是汇编器中的标号，并不是操作系统识别的程序入口点。装入这些字节的程序必须事先知道目标指令集、指令模式、装入地址和调用约定。

组合文件头与载荷

数据接口可以直接构造一个小型文件：

```
// 先写出四字节文件标识，再写出两个小端顺序的 16 位头字段。  
emit.bytes(b"RAW1");  
emit.u16(1);  
emit.u16(0);
```

payload:

```
// 标号记录载荷的逻辑起点，随后写出三个载荷字节。  
emit.bytes(b"ABC");
```

输出内容为：

```
52 41 57 31 01 00 00 00 41 42 43
```

开头四个字节是文件标识，第一个 **u16** 字段保存版本号，第二个 **u16** 字段暂时保留，其余字节构成载荷。

这里不需要专用的文件格式声明。源代码中的写出顺序就是文件中的字节顺序。

使用结构体描述记录

如果二进制记录具有具名字段，并且字段之间不能出现隐式对齐空隙，可以使用紧凑结构体：

```
// 紧凑结构体按声明顺序连续存放两个 16 位字段。  
packed struct ChunkHeader {  
    kind: u16  
    size: u16  
}  
  
// 写出记录头，随后紧接着写出三个字节的记录内容。  
emit.struct(ChunkHeader {  
    kind: 1,  
    size: 3,  
});  
emit.bytes(b"ABC");
```

输出内容为：

```
01 00 03 00 41 42 43
```

结构体让字段名称和位宽清楚可见，**emit.struct** 则写出它的紧凑字节表示。只有当文件格式本身要求相同的自然对齐和末尾填充时，才应使用普通结构体。

如果需要重复写出多条记录，可以把写出逻辑放入过程函数：

```
// 每次调用都按“一个字节的标签、四个字节的数值”写出一条记录。  
fn emit_record(tag: u8, value: u32) {  
    emit.u8(tag);  
    emit.u32(value);  
}
```

```
// 两条记录按照调用顺序连续写入文件。  
emit_record(1, 0x11223344);  
emit_record(2, 0xaabbccdd);
```

两条记录会连续写出：

```
01 44 33 22 11 02 dd cc bb aa
```

过程函数有助于让自定义文件中的同类记录保持一致，但最终的文件顺序仍然由普通的源代码顺序决定。

计算文件头字段

不要手工计算偏移、大小和校验和。可以先写出固定宽度的占位字段，用标号标记相关数据，再在 `defer` 收尾块中计算最终值：

```

// 让本区域的逻辑地址从零开始，便于用标号差表示文件内距离。
origin(0);

magic:
emit.bytes(b"XIF1");

// 先为总大小、载荷偏移和校验和各写出一个四字节占位字段。
size_field:
emit.u32(0);
payload_offset_field:
emit.u32(0);
checksum_field:
emit.u32(0);

payload:
emit.bytes(b"OK!!");
payload_end:

defer {
    // 布局稳定后，根据标号回填大小、偏移和前两个载荷字节的校验和。
    store.u32(size_field, payload_end - magic);
    store.u32(payload_offset_field, payload - magic);
    store.u32(checksum_field, load.u8(payload) + load.u8(payload + 1));

    // 检查最终文件中的标识和三个回填字段。
    assert(load.bytes(magic, 4) == b"XIF1");
    assert(load.u32(size_field) == 20);
    assert(load.u32(payload_offset_field) == 16);
    assert(load.u32(checksum_field) == 0x9a);
}

```

完整文件为：

```
58 49 46 31 14 00 00 00 10 00 00 00 9a 00 00 00 4f 4b 21 21
```

源代码没有把文件总大小或载荷偏移写成固定数值。如果文件头或载荷发生变化，收尾处理会把新值回填到已有字段中。这个示例只有一个从逻辑地址零开始的区域，区域内的文件字节连续排列，因此从 `magic` 计算的逻辑距离也正好等于载荷在文件中的相对偏移。

写入字段时，应明确它使用哪一种位置关系：

- 如果格式需要逻辑距离，应计算两个标号之差；
- 如果写出数据时需要当前文件位置，应使用 `file_offset()`；
- 如果 `defer` 中需要某个区域最终确定的文件起始位置，应使用 `region_file_offset(address)`；
- 如果需要稳定后的最终大小，应使用 `region_file_size(address)` 和 `region_logical_size(address)`。

区分逻辑地址与文件偏移

文件中的字节可以连续存放，同时拥有互不连续的逻辑地址：

```
// 文件头位于文件偏移 0，但它的逻辑地址从 0x1000 开始。
region.begin("header", 0x1000, 0);
emit.bytes(b"HDR0");

// 载荷紧接文件头写入文件，但逻辑地址改从 0x2000 开始。
output.section("payload", 0x2000);
payload:
emit.bytes(b"DATA");

defer {
    // payload 正好位于区域起点，因此可用它查询区域的起始文件偏移。
    assert(payload == 0x2000);
    assert(region_file_offset(payload) == 4);
}
```

文件中的字节仍然紧密排列：

```
48 44 52 30 44 41 54 41
```

载荷从文件偏移 `4` 开始，但它的逻辑地址是 `0x2000`。固件、只读存储器映像、内存映像以及描述数据装入位置的格式经常需要区分这两个位置。

除非格式明确规定了逻辑地址与文件偏移之间的换算关系，否则不要用逻辑地址推导文件偏移。应分别查询或记录这两种位置。

表示已初始化的空隙与不占文件空间的尾部

预留空间后面是否还有已初始化数据，会影响它是否占用文件字节。内部空隙会变成实际的零字节，而连续的预留尾部可以不写入文件：

```
// 建立逻辑地址从 0x5000 开始、文件偏移从零开始的映像区域。
region.begin("image", 0x5000, 0);

emit.bytes(b"HDR0");
file_size_field:
emit.u32(0);
logical_size_field:
emit.u32(0);

// 中间三字节空隙会因后续的 0xee 而写入文件，末尾八字节只增加逻辑大小。
emit.u8(0xaa);
reserve(3);
emit.u8(0xee);
reserve(8);

defer {
    // 布局稳定后，分别回填文件大小与逻辑大小。
    store.u32(file_size_field, region_file_size(file_size_field));
    store.u32(logical_size_field, region_logical_size(logical_size_field));

    assert(load.u32(file_size_field) == 17);
    assert(load.u32(logical_size_field) == 25);
}
```

文件实际只有 17 个字节：

```
48 44 52 30 11 00 00 00 19 00 00 00 aa 00 00 00 ee
```

三字节的内部预留空间后面还有已初始化数据，因此会实际写成零字节。末尾连续预留的八个字节把逻辑大小增加到 25，却不会增加文件字节。

这种布局适合表示需要初始化为零的存储空间。文件只保存已经初始化的前缀，其余逻辑空间由加载程序或使用该文件的程序提供。

在后期布局中追加尾部

如果必须在普通源代码处理完成后追加真实的文件尾部或数据表，可以使用 `late_layout`：

```
// 文件开头先为最终大小写出一个四字节占位字段。
total_size:
emit.u32(0);
emit.bytes(b"DATA");

late_layout {
    // 普通源代码处理完成后，把真实尾部加入最终布局。
    emit.bytes(b"END!");
}

defer {
    // 收尾处理能够看到包含尾部在内的最终文件大小。
    store.u32(total_size, region_file_size(total_size));
    assert(load.u32(total_size) == 12);
}
```

输出内容为：

```
0c 00 00 00 44 41 54 41 45 4e 44 21
```

追加的尾部会参与最终布局。收尾处理读取完整映像的大小，并回填已经分配好的文件头字段。

只有在字节确实必须等到主要源代码处理完毕后才能创建时，才使用后期布局。如果普通的源代码顺序能够表达相同布局，直接按顺序写出通常更清楚。

验证完整文件

构造自定义格式时，应检查保证输出文件可被正确读取的条件：

- 文件标识和版本字段具有预期值；
- 偏移指向目标区域内部；
- 记录数量与实际写出的记录数一致；
- 已保存的大小与区域的最终大小一致；
- 校验和覆盖预期的字节范围；
- 逻辑大小和文件大小符合格式规则。

`defer` 中的断言会检查最终真正写入文件的字节，其中包括已经编码的指令和完成解析的地址回填项。如果条件不成立，汇编会立即停止，而不会生成一个表面正常、实际无效的文件。

检查应尽量靠近它所保护的字段。某个大小字段的回填操作及其对应断言通常应放在同一个收尾块中。

组织自定义格式的构造代码

对于小型格式，清晰的源代码顺序通常是：

1. 声明记录类型和常量；
2. 写出固定文件头和占位字段；
3. 写出普通载荷区域；
4. 追加确实需要延后生成的数据表或尾部；
5. 回填稳定字段并检查各项条件。

可复用的包含文件可以封装记录声明和写出过程函数。格式状态应通过参数和值明确传递，不要在源代码各处散布彼此无关的固定偏移。

随着自定义格式变得复杂，仍应保持同样的职责划分：

- 源代码和过程函数决定存在哪些记录；
- 标号和输出区域描述布局；
- 后期布局创建必须参与最终布局的字节；
- 收尾处理计算并检查稳定后的字段。

何时使用格式接口

不要仅仅因为裸二进制文件也能表示相同字节，就手工构造标准化的可执行文件或目标文件格式。这些格式还要求文件头、数据表、访问权限、导入、导出、重定位和加载规则彼此协调。

XIRASM 为这些任务提供面向用户的格式接口。语言指南只介绍这些接口的用途；独立的[可执行文件格式指南](#)会说明常用格式接口和更底层的高级辅助接口。

下一章将从源语言的角度简要介绍如何选择和使用格式接口，不会重复完整的可执行文件格式指南。

[返回语言指南](#)

第 14 章：可执行文件与目标文件格式

标准的可执行文件和目标文件不只是指令与数据的集合。它们还必须按照链接器和操作系统装载程序所要求的规则，组织文件头、节表或段表、访问权限、入口地址、导入项、导出项、符号和重定位信息。

XIRASM 为这些文件提供统一的格式接口：

```
// 导入用于构造标准文件格式的统一接口。  
import("format/format.inc");
```

通过这套接口，源代码只需描述期望生成的映像，不必手工填写表项数量、表项位置、文件偏移、虚拟地址和文件头字段。

本章介绍各类格式共用的语言写法。完整的格式选项，以及导入、导出、重定位、共享库、目标文件和高级辅助接口，请参阅[可执行文件格式指南](#)。

先声明映像，再写入内容

使用格式接口时，程序首先声明映像包含哪些节或段。每个描述项都要给出名称、用途和权限。

描述项列表会直接决定生成结果：

- 列表长度决定表项数量；
- 列表顺序决定各项在格式表中的顺序；
- 名称用于标识稍后写入的内容块；
- 用途决定所选格式采用的处理方式；
- 权限会转换为节或段的属性。

源代码不需要另行传入表项数量或行号。增加或删除描述项后，生成的格式结构会自动随之更新。

最小的 ELF64 可执行文件

下面的程序生成一个 x86-64 ELF 可执行文件，其中只有一个可装载、可读取、可执行的段：

```

// 导入格式接口，并选择 64 位 x86 指令编码。
import("format/format.inc");
x86.use64();

// 声明 ELF64 可执行映像及其唯一的可装载代码段。
const image0: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(
            ".text",
            format_load | format_readable | format_executable
        )
    )
)
format_begin(image0);

// 在声明的 .text 段中写入程序入口代码。
format_segment_begin(image0, ".text");
start:
    // 调用 Linux 退出系统调用，并把退出状态设为零。
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image0, ".text");

// 绑定入口标号，随后完成映像中的文件头和表项。
const image: map = format_entry(image0, start)
format_finish(image);

```

在 x86-64 Linux 上，生成的程序会以状态码零退出。

可以把这段源代码分成五步理解：

1. `format_elf64` 创建一份声明式 ELF64 映像方案。
2. `format_segment` 声明一个有名称和权限的段。
3. `format_begin` 根据映像方案写出所需结构。
4. `format_segment_begin` 和 `format_segment_end` 圈定这个段中实际写入的指令与数据。
5. `format_entry` 绑定入口标号，`format_finish` 完成整个映像。

源代码没有提供程序头数量、表项行号、文件偏移、装载地址或入口字段的位置。格式接口会根据映像方案和最终确定的段布局推导这些信息。

节与段

常规使用流程会采用所选文件格式自身的概念：

- PE 映像和目标文件格式主要使用有名称的节；

- ELF 可执行文件和共享对象使用可装载的段；
- ELF 目标文件使用节，而不是运行时装载映射。

相应的内容写入接口也使用这些名称：

```
format_section_begin(image, name)
format_section_end(image, name)

format_segment_begin(image, name)
format_segment_end(image, name)
```

只能打开映像方案中已经声明的描述项，并按照声明的用途使用每个名称。这样，格式接口才能把写出的字节、逻辑大小、权限和自动生成的表项关联到正确的格式条目。

格式系列

常用格式接口为各类受支持映像提供以下构造函数系列：

映像类别	常用构造函数系列
PE32 和 PE64 映像	<code>format_pe32</code> 、 <code>format_pe64</code>
COFF32 和 COFF64 目标文件	<code>format_coff32</code> 、 <code>format_coff64</code>
ELF32 和 ELF64 可执行文件	<code>format_elf32</code> 、 <code>format_elf64</code>
ELF32 和 ELF64 目标文件	<code>format_elfobj32</code> 、 <code>format_elfobj64</code>
ELF64 共享对象	<code>format_elf64_so</code>

构造选项用于区分可执行文件或动态链接库模式、控制台或图形界面子系统、位置无关执行、数据页不可执行策略和地址随机化等映像属性。节和段的描述项则记录内容用途，以及读取、写入、执行或可丢弃等权限。

完整的选项和描述项列表属于参考资料，本语言指南不再重复列出。

绑定入口地址

可执行文件应在源代码定义起始代码之后，把对应标号绑定为入口地址：

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

`format_entry` 会返回更新后的映像方案。显式保留这个返回值，可以清楚表示映像状态已经改变，也便于 `format_finish` 检查可执行文件是否具备必要的入口信息。

目标文件和部分库格式不使用可执行入口。应遵循所选格式自身的构造流程，不要为它们添加没有实际意义的入口标号。

自动生成的格式内容

有些格式内容由较高层的声明生成，而不是作为普通内容字节直接写入。例如：

- 导入表和导出表；
- 符号表和字符串表；
- 重定位记录；
- 基址重定位节；
- 动态链接元数据；
- 资源目录。

相关格式接口接收名称、标号、权限和重定位种类，并在内部推导节编号、符号索引、表偏移、表项位置和表项数量。

不要用手工计算的表项位置替代这些声明。格式接口的作用，就是让格式元数据始终跟随实际布局。

常用格式接口与高级格式接口

普通程序应使用 `format/format.inc`。这是面向用户的稳定接口，负责协调格式属性、描述项、有名称的内容块、入口信息和自动生成的表。

各格式专用的包含文件提供更底层的构造部件。有些提供特定位数的便捷封装，有些则直接提供文件头、节、段或表的辅助接口。只有在实现新的统一格式接口，或者构造常用接口无法表达的特殊布局时，才需要使用这些接口。

仅仅为了生成多节可执行文件、动态链接库、目标文件、共享对象、导入表、导出表或重定位表，并不需要使用底层接口。只要常用格式接口能够表达目标映像，就应优先使用它。

不要随意混用不同层级的接口。底层辅助接口可能要求调用者自行维护表项数量、行号、偏移，或者保证某些格式规则始终成立；这些工作通常由常用格式接口负责。

后续阅读

[可执行文件格式指南](#)提供以下内容的完整可运行示例：

- PE32 和 PE64 可执行文件与动态链接库；

- COFF32 和 COFF64 目标文件;
- ELF32 和 ELF64 可执行文件与位置无关可执行文件;
- ELF32 和 ELF64 目标文件;
- ELF64 共享对象;
- 多个节和段;
- 已初始化数据和未初始化数据;
- 导入、导出、符号和重定位;
- 常用格式接口与高级底层辅助接口之间的使用边界。

当格式系列和构造流程已经确定，只需查询准确的函数签名、标志或常量时，请查阅接口参考手册。

最后一章将介绍诊断信息和实用的源代码约定，帮助编写便于维护的 XIRASM 程序。

[返回语言指南](#)

第 15 章：诊断信息与实用约定

如果源代码明确写出自身依赖的条件，汇编器报告的问题通常更容易定位和修正。XIRASM 可以输出普通提示、不会中止汇编的警告、主动触发的错误，以及可直接执行的约束检查。

源代码的组织方式也应遵循同一原则。清晰的名称、明确的地址或文件位置、小而专一的辅助函数，以及合适的抽象层，都能让二进制布局更容易审查，也更便于安全修改。

阅读诊断信息

诊断信息会指出源代码位置、严重程度和具体消息：

```
source.asm:12:5: error: header size must be four
```

这里的位置指向报告问题或触发问题的语句。语法分析、编译期求值、指令编码、布局、地址回填项解析和收尾处理产生的错误，都采用这种以源代码位置为中心的报告方式。

应当先处理第一条错误。后续错误可能只是前面某个无效声明、缺失标号或被拒绝指令造成的连带结果。

提示与警告

`print` 输出普通提示，`warn` 输出不会中止汇编的警告：

```
// 输出当前位置，帮助用户了解本次汇编采用的映像起点。
print("image origin", here());

// 输出仍然有效，但使用默认对齐值时给出明确提醒。
warn("using default alignment", 16);

// 确认继续汇编所需的配置条件成立。
assert(true, "configuration must be valid");

// 诊断信息不会改变输出内容，这里实际写出一个字节。
emit.u8(0x7d);
```

这段源代码可以成功汇编，并写出一个字节：

```
7d
```

诊断信息还会显示消息之后传入的值：

```
note: image origin 0
warning: using default alignment 16
```

`print` 适合输出用户确实需要知道的汇编期信息，不应当用来长期保留大量跟踪消息。当输出仍然有效，但所选配置值得用户注意时，应使用 `warn`。

警告不会改变输出内容，也不会导致汇编失败。如果某个条件会生成无效文件或不受支持的程序，应当报告错误，而不是只发出警告。

主动报告错误

`err` 会输出致命诊断信息并停止汇编：

```
if target.bits != 64 {
    err("this source requires a 64-bit target", target.bits);
}
```

消息之后的其他参数会依次显示出来。如果某个值导致配置无效，并且显示它有助于用户修正源代码，就应当把这个值一并传给 `err`。

错误消息应说明具体违反了什么要求：

```
err("section alignment must be a nonzero power of two", alignment);
```

如果能够明确指出预期条件，就不要只写 `invalid value` 这类含义模糊的消息。

断言

`assert` 是表达约束条件最简洁的方式：

```
// 定义由两个 16 位字段组成的紧凑文件头布局。
packed struct Header {
    kind: u16
    size: u16
}

// 文件头大小发生变化时立即停止汇编。
assert(sizeof(Header) == 4, "Header must remain four bytes");

// 写出文件头及其后紧接的三个字节载荷。
emit.struct(Header {
    kind: 1,
    size: 3,
});
emit.bytes(b"ABC");
```

条件成立时，断言不会写出任何内容。最终生成的字节为：

```
01 00 03 00 41 42 43
```

如果条件为假，汇编会停止，传入的消息会成为错误文本。

普通断言适合检查在处理源代码时已经确定的事实：

- 类型大小和字段偏移；
- 选项组合；
- 列表和映射中的内容；
- 目标平台要求；
- 常量范围；
- 已声明值之间的关系。

如果某项事实只有在最终映像稳定后才能确定，应在 `defer` 收尾块中使用断言，例如：

- 最终文件大小和逻辑大小；
- 已解析的偏移和地址；
- 已回填的文件头字段；
- 校验和；
- 生成的表内容；
- 指令地址回填产生的字节。

让断言在其输入已经稳定的阶段执行，可以避免意外依赖尚未确定的临时布局值。

保护依赖特定目标平台的源代码

应在依赖目标平台的代码附近明确写出要求：

```
// 明确选择 64 位 x86 指令编码。
x86.use64();

// 不满足位数要求时，在汇编期间直接拒绝该目标平台。
if target.bits != 64 {
    err("this routine requires x86-64");
}

entry:
    // 例程只为 x86-64 生成清零和返回指令。
    xor eax, eax
    ret
```

生成的字节为：

```
31 c0 c3
```

目标平台条件在汇编期间执行。它们负责选择或拒绝源代码，不会在生成的程序中加入运行时检查。

明确选择指令模式，也能让可复用源代码更容易审查。读者不应当依靠猜测来判断一条指令是为 x86-16、x86-32、x86-64、32 位 RISC-V 还是 64 位 RISC-V 编写的。

用名称代替含义不明的数字

如果一个值表示文件格式、协议或布局含义，应当为它命名：

```
const header_size: u32 = 16
const page_alignment: u64 = 0x1000
const record_kind_code: u16 = 3
```

算法内部使用的数字指令操作数可以保留在局部位置。偏移、标志、结构大小、格式值和对齐规则通常都值得使用清晰的名称。

如果一个值表示有限选项中的某一种，应使用具名常量。多个字段共同构成一条记录时，应使用结构体。源代码需要动态构造一份计划时，可以使用映射或列表。

在名称中明确坐标含义

二进制文件生成逻辑经常同时处理多种坐标。名称应当直接表明数值属于哪一种坐标：

```
header_foa
text_rva
entry_address
payload_file_size
payload_logical_size
```

其中，名称中的 `foa` 表示文件偏移，`rva` 表示相对虚拟地址。完整单词形式同样适合不常使用这些缩写的项目。

当 `offset` 可能表示文件偏移、区域内相对偏移、虚拟地址或结构体字段偏移时，不要使用这样过于笼统的名称。

辅助函数的参数也应遵循同一规则。接受 `logical_address` 和 `file_offset` 的函数，比只接受 `start` 和 `position` 的函数更不容易被错误调用。

从布局中推导事实

应优先使用标号和布局查询，而不是手工维护常量：

- 用标号相减得到逻辑距离；
- 用 `sizeof` 和 `offset_of` 查询复合值的布局；
- 用 `file_offset()` 查询当前实际文件位置；
- 在 `defer` 中查询稳定后的输出区域信息；
- 让格式描述列表决定表项数量和顺序；
- 让符号与重定位声明决定生成的索引。

只有外部格式规范明确要求某个固定值时，才适合硬编码偏移。即使如此，也应给这个常量起一个能够说明用途的名称，并尽可能用断言检查它周围的布局。

让收尾处理保持专一

在可以正常按顺序写出字节时，普通源代码就应当采用正常顺序。

只有确实需要在主体源代码之后追加的字节或区域，才应放入 `late_layout`。只有检查和回填稳定后的映像时，才应使用 `defer`。

不要只是为了回避源代码组织问题，就把普通的数据写出操作移到更晚的阶段。清楚分开的文件头、载荷、尾部和收尾处理，比一组相互牵连的延后副作用更容易理解。

每个收尾块应集中处理彼此相关的字段。一个只回填某个文件头并检查其数值的代码块，比一个在整个文件中修改多个无关区域的代码块更容易审查。

按职责组织模块

一种实用的项目组织方式会分别存放：

- 共享常量和数据类型；
- 可复用的数据写出过程；
- 依赖特定目标平台的指令例程；
- 文件格式或容器的构造逻辑；
- 用于生成输出的数据输入；
- 负责选择选项并汇编最终映像的顶层源文件。

需要只初始化一次的模块应使用 `import`。只有确实需要重复执行文本内容时，才应使用 `include`。

对外提供的辅助函数应保持简短，并明确列出所需输入。如果可以直接传入计划、列表、映射、标号或选项值，就不要依赖隐藏的可变状态。

选择正确的格式层

生成标准可执行文件或目标文件时，应从下面的格式接口开始：

```
// 导入用于构造标准文件格式的常规接口。  
import("format/format.inc");
```

常规格式接口负责描述项数量、表项顺序、生成的索引、偏移以及常见格式约束。

只有在实现新的格式接口，或者所需布局无法由常规格式接口表达时，才应使用更底层的格式文件。不要仅仅因为底层接口公开了更多数值，就改用更低的层次。这些数值通常属于内部记录信息，应继续由常规格式接口负责管理。

项目自有文件应采用第 13 章介绍的裸二进制文件和自定义二进制文件方法，不要强行套用操作系统使用的标准文件格式。

在交付边界验证结果

当 XIRASM 即将把输出内容交给其他系统时，应验证相关事实：

- 写出自定义文件前，检查最终结构；
- 明确声明可执行权限和重定位规则；
- 写出导入的数据前，先验证其内容；
- 尽早拒绝不受支持的目标平台组合；
- 为入口点和导出符号保留清晰名称；
- 为每个可复用的公开包含文件保留一个可运行的小型示例。

验证内容应说明双方约定，而不是重复内部实现细节。检查表项数量是否与声明列表一致很有价值；逐项重复每一步内部算术过程，则更难长期维护。

查找合适的文档

语言指南用于理解：

- 编译期源代码如何与处理器指令配合；
- 值、流程控制、函数、集合和词法单元如何工作；
- 标号、数据、输出区域、虚拟输出和收尾处理如何共同塑造文件；
- 如何在裸二进制输出和格式接口之间作出选择。

接口参考手册用于查询：

- 函数的准确签名；
- 接受的值类型；
- 常量和选项标志；
- 简明的行为说明和错误条件。

可执行文件格式指南用于学习：

- 完整的 PE、COFF 和 ELF 程序；
- 可执行文件、动态链接库、目标文件、位置无关可执行文件和共享库的构造方法；
- 导入、导出、符号、资源和重定位；
- 多节和多段布局；
- 常规格式接口与底层格式辅助接口之间的边界。

这几份文档分别承担概念讲解、接口查询和特定格式操作指南的职责，使每一份文档都能保持清晰易读。

完成前检查

在认为一个源文件已经完成之前，应确认：

- 已明确指定目标平台和指令模式；
- 常量和坐标使用了能够说明含义的名称；
- 标号代替了手工计算的地址；
- 外部使用者依赖结构体布局时，已经用断言检查该布局；
- `import` 和 `include` 采用了预期的执行方式；
- 预留尾部和文件中的实际空隙都是有意安排的；
- `late_layout` 只生成确实需要延后写出的字节；
- 收尾块只回填已有空间，并检查稳定后的事实；
- 标准文件格式默认使用常规格式接口，只有确实存在更底层需求时才例外；
- 致命条件会产生能够指导用户修正问题的消息；

- 为最终输出保留了一项可重复执行的小型汇编或运行检查。

这些做法可以让 XIRASM 源代码始终贴近它的实际目的：编写易读的汇编代码，明确表达编译期逻辑，并构造可以验证的二进制布局。

[返回语言指南](#)

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楚可见。

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，不应对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
```

加载器会为这个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>0xfffffffffffffffffc</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。

用户只需提供名称、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。

XIRASM 语言 API 参考

本参考手册集中说明 XIRASM 的语法形式和内置 API，适合在已经了解基本概念后快速查找准确规则。需要按学习顺序理解语言特性时，请先阅读[语言指南](#)。

可执行文件与目标文件格式使用单独的文档：

- [可执行文件格式指南](#)介绍常用的 PE、COFF 和 ELF 构建流程；
- 直接控制文件头、数据表、输出区域和重定位的底层接口属于高级格式指南。

本参考手册不收录 `format_*` 过程和格式系列专用辅助接口。

阅读约定

语法形式使用以下记号：

- `name` 表示由程序提供的标识符；
- `expression` 表示该位置允许使用的任意表达式；
- `type` 表示可选的显式类型；
- 语法形式中方括号内的内容可以省略。

除非另有说明，示例使用 x86-64 处理器指令。编译期语言规则不依赖所选指令集。

内容结构

第一部分：核心语言

1. [源码、绑定与作用域](#) - 标号、处理器指令、常量、变量、赋值、代码块和跨行参数。
2. [函数与流程控制](#)
3. [结构体、联合体与复合值](#)
4. [收尾处理语法形式](#)

第二部分：汇编器操作

5. [模块与诊断信息](#)
6. [目标平台、处理器指令与符号](#)
7. [数据写出、预留空间与对齐](#)
8. [输出区域、输出区与游标](#)
9. [读取、写入与最终区域信息](#)

第三部分：编译期辅助库

- 10. 文本、转换与标号名称
- 11. 字节序列
- 12. 列表与映射
- 13. 文件与结构化数据
- 14. 词法单元与模式匹配

每章保持紧凑的速查结构：先列出语法或函数形式，再说明行为、限制、错误条件和最小示例。

第 1 章：源码、绑定与作用域

语法摘要

形式	语法	用途
标号	<code>name:</code>	在当前逻辑地址定义源码标号。
处理器指令	<code>mnemonic operands</code>	写出当前目标平台的普通指令。
常量	<code>const name [: type] = expression</code>	声明不可变的编译期值。
变量	<code>let name [: type] = expression</code>	声明可变的编译期值。
赋值	<code>name = expression</code>	更新已经存在的可变绑定。
代码块	<code>{ statements }</code>	建立嵌套的词法作用域。
跨行参数	<code>callee(...)</code>	让一个表达式跨越多行继续书写。

标号

```
// 在当前位置定义 start 标号，并写出一条设置 rax 的指令。  
start:  
    mov rax, 1
```

标号会把名称绑定到它在源码位置对应的逻辑地址。后续指令、表达式、地址回填项和符号 API 都可以引用标号。只要所选指令编码或输出操作能够由汇编流程解析，就允许向前引用尚未定义的标号。

标号不是编译期的 `const` 或 `let` 绑定。它表示汇编器符号，最终数值取决于布局结果。

处理器指令

```
// 使用普通指令语法写出三条 x86-64 指令。  
mov rax, 1  
add rax, 2  
ret
```

处理器指令使用正常的文本语法，不写成编译期函数调用。当前目标平台决定可以使用哪些助记符、寄存器、操作数和指令编码。

编译期语句和处理器指令可以出现在同一个源文件中。编译期表达式决定写出哪些内容，处理器指令行描述要写出的指令。

常量

```
// 声明一个由类型推断的常量和一个显式指定类型的常量。  
const page_size = 4096  
const marker: u8 = 0x90
```

`const` 会计算初始化表达式，并在当前作用域中建立不可变绑定。当值的类型可以推断时，可以省略显式类型。

不能向常量赋值：

```
const value = 1  
value = 2
```

配置值、计算得到的地址、描述项数值以及其他不应在声明后改变的信息，适合使用 `const`。

变量

```
// 建立两个可变绑定，再分别更新它们。  
let count = 1  
let flags: u32 = 0  
  
count = count + 1  
flags = flags | 0x20
```

`let` 会在当前作用域中建立可变绑定。赋值会更新具有指定名称且距离当前位置最近的可见可变绑定。

赋值不会声明新名称。目标绑定必须已经存在，而且必须可变。

代码块与名称遮蔽

```
// 内层代码块使用同名变量，但不会改变外层常量。  
const value = 1  
  
{  
  let value = 2  
  value = value + 1  
  assert(value == 3)  
}  
  
assert(value == 1)
```

花括号会建立词法作用域。代码块中的声明可以遮蔽外层作用域中的同名绑定，而不会改变外层绑定。

代码块结束后，其中声明的绑定不再存在：

```
{  
  let hidden = 1  
}  
  
emit.u8(hidden)
```

代码块适合限制临时值的可见范围、隔离辅助计算，并明确表示名称遮蔽。

跨行表达式参数

只要调用表达式的圆括号尚未闭合，就可以跨越多行继续书写：

```
// 在多行中书写同一个 assert 调用。  
const value = 1  
  
assert(  
  value == 1  
)
```

缩进可以提高可读性，但不能代替右侧结束符。嵌套调用也可以使用相同形式。只有全部结束符闭合后，完整表达式才会求值。

完整示例

下面的源代码组合了本章介绍的七种形式：

```
// 外层 value 是不可变绑定。  
const value = 1  
  
// 内层 value 是只在代码块中存在的可变绑定。  
{  
    let value = 2  
    value = value + 1  
    assert(value == 3)  
}  
  
// 跨行检查外层绑定仍然保持原值。  
assert(  
    value == 1  
)  
  
// 定义标号并写出指令，最后写出外层 value 的一个字节。  
start:  
    mov rax, 1  
  
emit.u8(value)
```

源代码会写出：

```
b8 01 00 00 00 01
```

内层 `value` 可以修改，而且只在代码块中可见。外层 `value` 仍然是不可变值 `1`，并在指令之后写入输出内容。

选用指南

需求	使用形式
绑定由布局产生的地址	标号
描述目标平台指令	处理器指令行
命名不可变的编译期信息	<code>const</code>
保存可变的编译期状态	<code>let</code>
更新可变状态	赋值
限制名称的可见范围或遮蔽名称	代码块
整理较长的调用或嵌套表达式	跨行参数

第 2 章：函数与流程控制

函数和流程控制语句在汇编过程中执行。它们用于计算值、选择源码中的操作和重复执行这些操作。除非函数体或语句块写出相应的处理器指令，否则不会生成运行时函数调用、分支或循环。

语法摘要

形式	语法	用途
过程函数	<code>fn name(parameters) { statements }</code>	封装编译期操作。
过程调用	<code>name(arguments)</code>	执行过程函数。
返回值函数	<code>fn name(parameters) -> type { statements }</code>	计算表达式值。
返回	<code>return expression</code>	结束返回值函数并给出结果。
返回值调用	<code>name(arguments)</code>	在表达式中使用返回值函数。
条件分支	<code>if condition { statements }</code>	布尔条件为真时执行代码块。
备选分支	<code>} else { statements }</code>	执行备选代码块。
条件循环	<code>while condition { statements }</code>	布尔条件保持为真时重复执行。
数值范围循环	<code>for name in range(start, end) { statements }</code>	迭代左闭右开的数值范围。
列表循环	<code>for name in list_value { statements }</code>	按顺序迭代列表中的值。

过程函数

```
// 定义一个过程函数，连续写出输入值及其后继值。
fn emit_pair(value: u8) {
    emit.u8(value)
    emit.u8(value + 1)
}

// 分别调用两次，写出两组相邻字节。
emit_pair(2)
emit_pair(8)
```

没有 `-> type` 的函数是过程函数。过程函数执行操作，但不产生表达式值。函数体可以写出数据或指令、改变布局、声明局部值、使用流程控制，以及调用其他过程函数。

参数按位置对应。类型标注可以省略；如果提供了类型标注，调用时会用它检查对应实参。每次调用都必须为每个参数提供且只提供一个实参。

过程调用是一条语句。末尾可以写分号，但不要求写。

返回值函数

```
// 将 value 向上对齐到 alignment 的整数倍。
fn align_up(value: u64, alignment: u64) -> u64 {
    return ((value + alignment - 1) / alignment) * alignment
}

// 计算对齐后的值，并将 16 位结果写入输出内容。
emit.u16(align_up(0x73, 0x20))
```

带有 `-> type` 的函数是返回值函数。它的调用属于表达式，可以出现在声明、条件、实参、`return` 或更大的表达式中。

`return` 会计算表达式，并把结果转换为声明的返回类型。返回值函数中每条实际执行的路径都必须到达 `return`。

返回值函数用于辅助计算。它可以使用局部绑定、流程控制和其他返回值函数，但不得写出输出内容，也不得执行其他会改变布局的操作。

函数声明与调用规则

- 函数必须在顶层声明。
- 函数声明必须出现在首次调用之前。
- 同一声明中的参数名称不得重复。
- 每次调用都有各自的参数和局部绑定。
- 过程调用不能用作表达式值。
- `return` 只能出现在返回值函数中。
- 函数调用链最多允许 128 层。

只要递归返回值函数能在达到调用深度上限前结束，就允许使用递归。如果操作本身适合迭代，应使用循环。

`if` 与 `else`

```
// enabled 为 false, 因此只执行备选分支。
const enabled = false

if enabled {
    emit.u8(0x11)
} else {
    emit.u8(0x22)
}
```

条件必须计算为 `bool`。只有选中的分支会执行。每个分支都有自己的词法作用域。

备选分支应写成规范的 `} else if condition {` 和 `} else {` 形式。`else if` 和最后的 `else` 都可以省略。

`if` 语句在汇编期间选择源码操作。要实现运行时条件行为，仍需写出处理器分支指令。

while

```
// 每轮写出当前值，再更新控制循环的可变绑定。
let value = 1

while value <= 3 {
    emit.u8(value)
    value = value + 1
}
```

`while` 会在每次迭代前计算布尔条件。如果条件一开始就是假，循环体不会执行。

`break` 会结束最内层的活动 Meta 循环。`continue` 会跳过循环体中剩余的语句并开始下一次迭代。它们也可以用于 `for` 循环和延迟执行的 `while` 代码块。编译期循环最多执行 1,000,000 次。在循环外使用循环控制语句，或试图让它跨越函数调用边界，均属于错误。

数值范围迭代

```
// 迭代左闭右开的范围 [0, 4)，依次写出四个字节。
for index in range(0, 4) {
    emit.u8(index)
}
```

`range(start, end)` 包含 `start`，不包含 `end`。此示例写出 `00 01 02 03`。

数值范围只能向前移动，也可以为空。降序范围无效：

```
for value in range(2, 0) {  
    emit.u8(value)  
}
```

循环绑定是只读的局部值。每次迭代都会获得新的绑定和新的循环体作用域。

列表迭代

```
// 按列表顺序写出三个预先选定的操作码字节。  
const opcodes: list = list.of(0x90, 0x90, 0xc3)  
  
for opcode in opcodes {  
    emit.u8(opcode)  
}
```

列表循环按列表顺序访问各个值。循环绑定只在本次迭代中有效，不能对它赋值。

这种形式只直接接受列表值。要迭代映射内容，应先用 `map.keys(...)` 或 `map.values(...)` 等映射辅助函数取得列表。

无效的函数形式

过程函数不能返回值：

```
fn invalid() {  
    return 1  
}  
  
invalid()
```

返回值函数不能在未返回结果的情况下执行到函数末尾：

```
fn invalid() -> u64 {  
    const value = 1  
}  
  
const result = invalid()
```

返回值函数不能写出输出内容：

```
fn invalid() -> u64 {  
    emit.u8(1)  
    return 1  
}  
  
const result = invalid()
```

操作会改变输出内容或布局时，应使用过程函数。调用方需要计算结果时，应使用返回值函数。

完整示例

```

// 过程函数负责写出一对相邻字节。
fn emit_pair(value: u8) {
    emit.u8(value)
    emit.u8(value + 1)
}

// 返回 value 与 limit 中较小的值。
fn choose(value: u64, limit: u64) -> u64 {
    if value > limit {
        return limit
    } else {
        return value
    }
}

// 写出过程函数结果和返回值函数结果。
emit_pair(1)
emit.u8(choose(9, 5))

// 编译期条件选择第一个源码分支。
const enabled = true

if enabled {
    emit.u8(0xaa)
} else {
    emit.u8(0xff)
}

// 条件循环写出两个连续值。
let counter = 0

while counter < 2 {
    emit.u8(0x10 + counter)
    counter = counter + 1
}

// 数值范围循环写出 0x20 和 0x21。
for index in range(0, 2) {
    emit.u8(0x20 + index)
}

// 列表循环按顺序写出列表中的两个值。
const values: list = list.of(0x30, 0x31)

for value in values {
    emit.u8(value)
}

```

源代码会写出：

选用指南

需求	使用形式
封装会改变输出内容或布局的操作	过程函数
计算可复用的表达式值	返回值函数
选择一个代码块	<code>if</code>
在两个代码块中选择一个	<code>if / else</code>
在多个代码块中选择一个	<code>if / else if / else</code>
重复执行，直到可变状态满足条件	<code>while</code>
迭代固定的数值范围	搭配 <code>range</code> 的 <code>for</code>
迭代列表中的已知值	搭配列表的 <code>for</code>
提前结束最内层循环	<code>break</code>
跳过当前迭代的剩余部分	<code>continue</code>

第 3 章：结构体、联合体与复合值

结构体和联合体用于描述二进制布局。它们的值会在汇编期间存在，直到被打包成字节或写入输出内容。

语法摘要

形式	用途
<code>struct Name { fields }</code>	声明自然布局结构体。
<code>packed struct Name { fields }</code>	声明不含填充的结构体。
<code>union Name { fields }</code>	声明自然布局联合体。
<code>packed union Name { fields }</code>	声明不含尾部填充的联合体。
<code>Name { field: value }</code>	构造复合值。
<code>value.field</code>	读取复合值的字段。
<code>emit.struct(value)</code>	打包复合值并立即写出。
<code>sizeof(Type)</code>	返回类型的二进制大小。
<code>offset_of(Type, field_path)</code>	返回字段偏移。
<code>pack(value)</code>	将复合值转换为 <code>bytes</code> 。

自然布局结构体与紧凑结构体

```

// 自然布局会按字段的对齐要求插入填充。
struct NaturalHeader {
    tag: u8
    size: u32
}

// 紧凑布局让字段连续排列，不插入填充。
packed struct PackedHeader {
    tag: u8
    size: u32
}

// 对比两种布局的总大小和 size 字段偏移。
assert(sizeof(NaturalHeader) == 8)
assert(offset_of(NaturalHeader, size) == 4)
assert(sizeof(PackedHeader) == 5)
assert(offset_of(PackedHeader, size) == 1)

```

自然布局结构体会按照每个字段的类型对齐，并将最终大小向上取整到结构体的对齐要求。字段之间和结构体末尾都可能出现填充。

紧凑结构体会把每个字段紧接在前一个字段之后，不添加尾部填充。需要精确描述文件记录和协议记录时使用紧凑布局；记录中的字段需要对齐时使用自然布局。

紧凑布局只移除填充，不会清除复合类型自身的对齐属性。紧凑复合类型仍保留最大字段对齐，因此自然布局的外层复合类型会按照该边界放置嵌套的紧凑字段。

同一个声明中的字段名称必须唯一。

字段默认值与结构体字面量

```

// magic 使用声明中的字段默认值，tail 由结构体字面量指定。
packed struct Header {
    magic: u16 = 0x4241
    tail: u16
}

const header: Header = Header {
    tail: 0x4443
}

// 读取默认字段，并写出完整的紧凑结构体。
assert(header.magic == 0x4241)
emit.struct(header)

```

结构体字面量按名称为字段赋值。字面量中的书写顺序不必与声明顺序一致。

省略的结构体整数字段会使用声明的字段默认值。省略任何其他字段都会报错。字面量中出现未知字段或重复字段也会报错。联合体字段不能声明默认值。

复合值字面量可以直接传给内置表达式：

```
// 构造 Pair 后立即打包，并把所得两个字节写入输出内容。
packed struct Pair {
    low: u8
    high: u8
}

emit.bytes(pack(Pair {
    low: 3,
    high: 4
})))
```

自然布局联合体与紧凑联合体

```
// ThreeBytes 的紧凑布局大小为三个字节。
packed struct ThreeBytes {
    tag: u8
    value: u16
}

// 自然布局联合会按最大字段的对齐要求取整最终大小。
union NaturalValue {
    bytes: ThreeBytes
    word: u16
}

// 紧凑联合体只保留最大字段的精确大小。
packed union PackedValue {
    bytes: ThreeBytes
    word: u16
}

assert(sizeof(NaturalValue) == 4)
assert(sizeof(PackedValue) == 3)
```

联合体的每个字段都从偏移零开始。自然布局联合体采用最大字段的大小，并把最终大小向上取整到最大字段的对齐要求。紧凑联合体则采用最大字段的精确大小。

联合体字面量必须恰好选择一个当前字段：

```
// word 是这个联合体值唯一的当前字段。  
packed union Value {  
    byte: u8  
    word: u16  
}  
  
const value: Value = Value {  
    word: 0x1234  
}  
  
// 按 word 字段的声明宽度打包并写出联合体。  
emit.struct(value)
```

联合体初始化时不指定字段，或指定多个字段，都是无效写法。联合体字段声明不能提供默认值；字面量必须始终显式选择当前字段。

嵌套复合值字面量

```

// Point 将两个坐标连续存放。
packed struct Point {
    x: u16
    y: u16
}

// ValueBits 的当前字段可以是原始整数，也可以是 Point。
union ValueBits {
    raw: u32
    point: Point
}

packed struct Record {
    kind: u8
    value: ValueBits
}

// 在 Record 中逐层构造联合体和结构体字段。
const record: Record = Record {
    kind: 1,
    value: ValueBits {
        point: Point {
            x: 0x1122,
            y: 0x3344
        }
    }
}

// 点分字段路径会累加每一层的字段偏移。
assert(offset_of(Record, value.point.y) == 3)
emit.struct(record)

```

嵌套的复合值字段可以接受其声明类型对应的嵌套字面量。`offset_of` 接受点分字段路径，并累加各层的字段偏移。

字段访问

// 构造头部后，通过字段访问分别读取两个值。

```
packed struct Header {  
    magic: u16  
    size: u32  
}  
  
const header: Header = Header {  
    magic: 0x5a4d,  
    size: 0x40  
}
```

// 按字段各自的声明宽度写出读取结果。

```
emit.u16(header.magic)  
emit.u32(header.size)
```

字段访问会从已经保存的复合值中读取一个值。字段名称必须存在于该值的声明类型中。

sizeof 与 offset_of

```
sizeof(Type)  
offset_of(Type, field_path)
```

sizeof 返回完整的二进制大小，其中包括自然布局产生的填充。

offset_of 返回直接字段或嵌套字段的字节偏移。它不需要复合值；这两个操作查询的都是声明类型的布局。

pack

```
pack(value) -> bytes
```

pack 会创建包含复合值二进制表示的字节序列。整数字段会按照其声明宽度编码。自然布局中的填充字节为零。

需要检查、比较、保存这些字节，或把它们传给另一个函数时，使用 **pack**。

emit.struct

```
emit.struct(value)
```

`emit.struct` 会打包复合值，并把所得字节写入当前输出区域。它等价于写出 `pack` 的结果，但不会公开中间的 `bytes` 值。

无效的复合值字面量

以下形式都会被拒绝：

```
const unknown: Pair = Pair {  
    missing: 1  
}
```

```
const duplicate: Pair = Pair {  
    low: 1,  
    low: 2  
}
```

```
const incomplete: Pair = Pair {  
    low: 1  
}
```

```
const invalid_union: Value = Value {  
    byte: 1,  
    word: 2  
}
```

这些错误依次表示：未知字段、重复字段、缺少没有默认值的字段，以及联合体存在多个当前字段。

完整示例

```

// 自然布局头部包含字段对齐和尾部填充。
struct NaturalHeader {
    tag: u8 = 0x41
    size: u32 = 0x11223344
}

// 紧凑头部连续存放 tag 和 tail。
packed struct PackedHeader {
    tag: u8 = 0x42
    tail: u16
}

packed struct Point {
    x: u16
    y: u16
}

// ValueBits 让原始整数和坐标值共享存储空间。
union ValueBits {
    raw: u32
    point: Point
}

packed struct Record {
    kind: u8
    value: ValueBits
}

// 三字节字段用于演示联合体最终大小的布局差异。
packed struct ThreeBytes {
    tag: u8
    value: u16
}

union NaturalOdd {
    bytes: ThreeBytes
    word: u16
}

packed union PackedOdd {
    bytes: ThreeBytes
    word: u16
}

// 使用全部字段默认值，并写出自然布局结构体及其布局信息。
const natural: NaturalHeader = NaturalHeader { }
emit.struct(natural)
emit.u8(sizeof(NaturalHeader))
emit.u8(offset_of(NaturalHeader, size))
emit.u32(natural.size)

// 为 tail 指定值，tag 继续使用字段默认值。

```

```

emit.bytes(pack(PackedHeader { tail: 0x4443 })))

// 嵌套构造 Record, 并选择 point 作为联合体的当前字段。
const record: Record = Record {
    kind: 1,
    value: ValueBits {
        point: Point {
            x: 0x1122,
            y: 0x3344
        }
    }
}

emit.bytes(pack(record))

// 选择 ThreeBytes 作为紧凑联合体的当前字段。
const odd: PackedOdd = PackedOdd {
    bytes: ThreeBytes {
        tag: 0x55,
        value: 0x7766
    }
}

// 写出联合体内容、两种联合体大小以及 Record 的 kind 字段。
emit.bytes(pack(odd))
emit.u8(sizeof(NaturalOdd))
emit.u8(sizeof(PackedOdd))
emit.u8(record.kind)

```

源代码会写出：

```

41 00 00 00 44 33 22 11 08 04 44 33 22 11
42 43 44 01 22 11 44 33 55 66 77 04 03 01

```

选用指南

需求	使用形式
按字段自然对齐，并对最终大小取整	<code>struct</code>
精确的连续字节布局	<code>packed struct</code>
让字段重叠，并按自然对齐要求确定最终大小	<code>union</code>
让字段重叠，并采用最大字段的精确大小	<code>packed union</code>
查询声明的布局	<code>sizeof</code> 或 <code>offsetof</code>
以值的形式取得复合值字节	<code>pack</code>
立即写出复合值字节	<code>emit.struct</code>

第 4 章：收尾处理语法形式

XIRASM 提供两个有意分开的后期处理阶段：

- `late_layout` 在最终输出内容确定之前执行受限的布局变更；
- `defer` 在布局确定后读取并回填最终字节映像，但不会改变布局。

不要把这两种形式互换使用。

阶段顺序

阶段	允许承担的职责
普通源码	声明标号、写出内容、预留空间并登记后期代码块。
指令编码	编码普通处理器指令片段。
<code>late_layout</code>	一次性追加或重新组织受限的布局内容。
地址回填与布局	解析引用，并计算最终的逻辑位置和物理位置。
生成最终字节映像	创建最终字节映像，并应用已经解析的地址回填项。
<code>defer</code>	读取、验证和回填已经生成的现有字节。

每一种代码块都按照登记顺序执行。

`late_layout`

```
// 普通源码先写出第一个字节。
emit.u8(0x10)

late_layout {
    // 第一个后期布局块追加第二个字节。
    emit.u8(0x20)
}

late_layout {
    // 第二个后期布局块继续按登记顺序追加字节。
    emit.u8(0x30)
}
```

这个示例会写出 `10 20 30`。

后期布局块在普通源码写出和指令编码结束后执行一次，但早于地址回填项解析和最终布局。它写出的数据会参与最终偏移和大小的计算。

`late_layout` 接受：

- 输出与布局 API 调用；
- 整数、字节和复合值写出；
- 预留、补齐和对齐调用；
- 输出区域与虚拟区域调用；
- 向模块中已有内容执行写入；
- 诊断与断言；
- `if` 和 `else`。

它不接受局部声明、赋值、循环、标号、处理器指令、函数声明、源码加载、`defer` 或另一个 `late_layout`。

后期布局是一次性阶段，不是隐式的多轮处理机制。

预留尾部与后期布局

在同一个区域的预留尾部之后追加已初始化数据，会使预留空隙成为文件中的实际字节：

```
// 先写出一个字节，再预留两个只占逻辑空间的字节。
emit.u8(0xaa)
reserve(2)

late_layout {
    // 在预留尾部之后追加数据，使中间空隙进入实际输出。
    emit.u8(0xbb)
}
```

输出为 `aa 00 00 bb`。如果预留尾部必须只保留逻辑空间，请改用另一个输出区域。

`defer`

```

// 让当前输出区域从逻辑地址 0 开始。
origin(0)

// 为最终文件大小预留一个 16 位字段。
size_field:
emit.u16(0)

// 写出两字节正文。
payload:
emit.bytes(b"AB")

defer {
    // 布局稳定后回填区域文件大小，并读取结果进行验证。
    store.u16(size_field, region_file_size(size_field))
    assert(load.u16(size_field) == 4)
}

```

输出为 `04 00 41 42`。

收尾块在最终布局、最终字节映像生成和地址回填项写入完成后执行。它看到的正是即将写入文件的准确字节映像。

收尾块可以包含：

- `const` 和 `let` 声明；
- 对局部 `let` 值赋值；
- `if`、`else` 和 `while`；
- `store.u8`、`store.u16`、`store.u32`、`store.u64` 和 `store.bytes`；
- `assert`、`print`、`warn` 和 `err`。

表达式可以使用纯计算运算符、返回值函数、标号、`load.*` 和稳定的区域信息。

每个收尾块都有自己的局部作用域。循环最多执行 1,000,000 次。

收尾块内的局部计算

```

// 使用固定逻辑起点，并为校验和预留两个字节。
origin(0)

checksum_field:
emit.u16(0)

// 写出需要参与校验和计算的数据。
payload:
emit.bytes(b"ABCD")

defer {
    // 使用局部可变值逐字节扫描最终映像。
    let cursor = payload
    let checksum = 0
    const end = region_base() + region_file_size(payload)

    while cursor < end {
        checksum = checksum + load.u8(cursor)
        cursor = cursor + 1
    }

    // 把累加结果回填到已有字段。
    store.u16(checksum_field, checksum)
}

```

`let` 会建立可变的收尾块局部状态。赋值用于更新这些状态，`while` 则可以对已经完成的映像执行有界扫描和汇总计算。

从过程函数登记收尾块

```

// 定义一个登记 16 位回填操作的过程函数。
fn patch_u16(address: u64, value: u64) {
    defer {
        // 收尾块保存调用时传入的地址和值。
        store.u16(address, value)
    }
}

// 先写出占位字段，再登记稍后执行的回填。
field:
emit.u16(0)

patch_u16(field, 0x1234)

```

输出为 `34 12`。

过程函数调用发生在普通源码处理期间。从过程函数作用域捕获的值会为收尾块固定下来。过程函数不会从 `defer` 内部调用。

收尾块执行顺序

```
// 先写出一个会被两个收尾块连续修改的字节。
origin(0)
emit.u8(0)

defer {
    // 第一个收尾块把初始值改为 1。
    store.u8(0, 1)
}

defer {
    // 第二个收尾块读取前一次修改，再把数值增加 1。
    store.u8(0, load.u8(0) + 1)
}
```

输出为 `02`。后登记的收尾块可以观察到先登记代码块完成的回填。

收尾处理限制

`defer` 不能创建字节、标号、区域、对齐或预留空间：

```
defer {
    emit.u8(0x22)
}
```

在一个后期处理阶段中再嵌套另一个后期处理阶段同样无效：

```
defer {
    late_layout {
        emit.u8(0x22)
    }
}
```

每个回填目标都必须在收尾阶段之前写出或预留。写入操作只能访问已经生成的实际字节。经过裁剪的预留尾部虽然具有逻辑范围，但没有可以回填的实际字节：

```
origin(0)
emit.u8(0x11)
reserve(1)

defer {
    store.u8(1, 0x22)
}
```

这个写入操作会被拒绝，不会越过实际生成的字节范围写入数据。

完整示例

```

// 为大小和校验和分别预留两个 16 位字段。
origin(0)

size_field:
emit.u16(0)

checksum_field:
emit.u16(0)

// 普通源码先写出正文的前两个字节。
payload:
emit.bytes(b"AB")

late_layout {
    // 后期布局追加会参与最终大小和校验和计算的字节。
    emit.bytes(b"CD")
}

defer {
    // 扫描完整正文并计算校验和。
    let cursor = payload
    let checksum = 0
    const end = region_base() + region_file_size(payload)

    while cursor < end {
        checksum = checksum + load.u8(cursor)
        cursor = cursor + 1
    }

    // 回填两个文件头字段，并验证最终字节映像。
    store.u16(size_field, region_file_size(size_field))
    store.u16(checksum_field, checksum)

    assert(load.u16(size_field) == 8)
    assert(load.u16(checksum_field) == 0x010a)
    assert(load.bytes(payload, 4) == b"ABCD")
}

```

源代码会写出：

```
08 00 0a 01 41 42 43 44
```

`late_layout` 会添加正文最后两个字节。随后，`defer` 看到完整的八字节映像，计算校验和，并回填两个已经存在的文件头字段。

选用指南

需求	使用形式
写出普通源码内容	普通源码
在偏移和大小确定之前追加实际字节	<code>late_layout</code>
在布局完成后回填宽度固定的占位字段	<code>defer</code>
根据最终字节计算校验和	使用 <code>load.*</code> 和局部状态的 <code>defer</code>
验证最终偏移、大小或内容	使用 <code>assert</code> 的 <code>defer</code>

第 5 章：模块与诊断信息

语法摘要

API	语法	结果
包含源码	<code>include(path)</code>	每次调用都执行解析后的源码。
导入源码	<code>import(path)</code>	解析后的源码只执行一次。
提示	<code>print(value, ...)</code>	记录不会终止汇编的提示信息。
警告	<code>warn(value, ...)</code>	记录不会终止汇编的警告信息。
错误	<code>err(value, ...)</code>	记录错误并停止汇编。
断言	<code>assert(condition[, message])</code>	条件为假时停止汇编。

源码路径

传给 `include` 和 `import` 的 `path` 参数必须求值得到文本。相对路径从包含该调用的源文件所在位置开始解析，因此一个被加载的文件可以继续按自己的目录加载其他文件。

解析后的源码路径同时也是模块标识。两种不同写法只要解析到同一个源文件，就表示同一次导入。

重复包含源码

每次调用 `include(path)` 都会执行被加载的源码：

```
include("row.inc")
include("row.inc")
```

如果 `row.inc` 写出一个字节，这个字节就会写出两次。重复包含也会重复执行声明，因此多次包含声明了同一个函数或类型的文件，可能产生重复声明错误。

需要有意重复展开源码时，或者源码片段中的声明能够安全地在每个调用位置执行时，可以使用 `include`。

模块只导入一次

`import(path)` 只在第一次导入时执行被加载的源码：

```
import("library.inc")
import("library.inc")
```

第二次调用不会产生任何效果。因此，定义函数、类型、常量或可复用数据的文件通常应使用 `import`。

导入是模块顶层操作。在词法作用域的代码块或函数内部调用 `import` 是无效的。遇到递归的包含或导入链时，汇编器会拒绝整个循环，而不会只执行其中一部分。

诊断信息

`print`、`warn` 和 `err` 可以接收一个或多个值：

```
// 设置逻辑原点，再记录当前位置、警告和断言信息。
origin(0)

print("offset", here())
warn("diagnostic example", true)
assert(here() == 0, "unexpected origin")

// 写出一个字节，证明提示和警告不会停止汇编。
emit.u8(0x5a)
```

各个值会使用常规文本形式格式化，并以一个空格连接。上面的示例会记录：

```
note: offset 0
warning: diagnostic example true
```

`print` 和 `warn` 不会停止汇编。`err` 会在调用位置记录错误并停止汇编：

```
err("unsupported width", 24)
```

诊断 API 也可以在 `defer` 收尾块中使用。模块加载 API 不能在收尾处理中调用。

断言

`assert` 接收一个布尔条件和一条可选消息：

```
// 检查位宽是否有效，再把对应的字节数写入输出内容。
const width = 64
assert(width == 64)
assert(width % 8 == 0, "width must use whole bytes")

emit.u8(width / 8)
```

条件为真时，`assert` 不会写出内容，也不会记录诊断信息。条件为假时，汇编会停止并显示提供的消息。省略消息时，诊断文本为 `assertion failed`。

完整模块示例

下面四个文件共同演示重复包含、只导入一次、嵌套相对路径和诊断信息。

`main.asm` :

```
origin(0)

include("repeat.inc")
include("repeat.inc")

import("module/once.inc")
import("module/once.inc")

print("module bytes", here())
warn("diagnostic example", true)
assert(here() == 4, "unexpected module output")

emit.u8(0x44)
```

`repeat.inc` :

```
emit.u8(0x11)
```

`module/once.inc` :

```
include("nested.inc")
emit.u8(0x22)
```

`module/nested.inc` :

```
emit.u8(0x33)
```

最终输出为：

```
11 11 33 22 44
```

`repeat.inc` 会执行两次。`module/once.inc` 只执行一次，其中嵌套的 `include` 会相对于 `module` 目录解析路径。

第 6 章：目标平台、处理器指令与符号

语法摘要

形式	语法	结果
目标平台宽度	<code>target.bits</code>	返回当前 x86 指令模式或 RISC-V XLEN。
目标平台系列条件	<code>target.isa == .name</code>	在 <code>if</code> 中检查当前指令集系列。
x86 模式	<code>x86.use16()</code> 、 <code>x86.use32()</code> 、 <code>x86.use64()</code>	选择 x86 及其指令模式。
RISC-V 模式	<code>riscv.use32()</code> 、 <code>riscv.use64()</code>	选择 RISC-V 及其 XLEN。
动态处理器指令	<code>isa(text)</code>	根据文本值追加一条指令。
逻辑原点	<code>origin(address)</code>	设置当前输出区域的基地址。
当前位置	<code>here()</code>	返回当前逻辑地址。
动态标号	<code>label.define(name)</code>	根据文本值定义标号。
标号地址	<code>label_addr(name)</code>	返回标号或绝对符号的地址。

目标平台查询

`target.bits` 是整数表达式。对于 x86，它表示当前指令模式；对于 RISC-V，它表示当前 XLEN：

```
// 仅在当前目标平台系列为 x86 时检查并写出模式位数。
if target.isa == .x86_64 {
    assert(target.bits == 64)
    emit.u8(target.bits)
}
```

SPIR-V 没有适用于整个目标平台的整数位宽，因此该目标平台不能使用 `target.bits`。

`target.isa` 是目标平台条件形式，而不是通用值表达式。请在源码 `if` 中把它放在 `==` 或 `!=` 左侧：

```

if target.isa == .x86_64 { ... }
if target.isa != .riscv64 { ... }
if target.isa == .spirv { ... }

```

可用的系列字面量是 `.x86_64`、`.riscv64` 和 `.spirv`。系列名称不会随宽度变化：x86 的 16 位、32 位和 64 位模式都使用 `.x86_64`，RISC-V 的 XLEN 32 和 XLEN 64 都使用 `.riscv64`。

选择指令模式

模式过程会选择后续处理器指令片段保存的目标平台。已经记录的指令仍然保留先前的目标平台。

调用	后续处理器指令
<code>x86.use16()</code>	x86 16 位模式
<code>x86.use32()</code>	x86 32 位模式
<code>x86.use64()</code>	x86-64 模式
<code>riscv.use32()</code>	XLEN 为 32 的 RISC-V
<code>riscv.use64()</code>	XLEN 为 64 的 RISC-V

```

// 依次记录使用 16 位、32 位和 64 位模式的 x86 指令。
x86.use16()
isa("mov ax, 0x1234")

x86.use32()
isa("mov eax, 0x12345678")

x86.use64()
isa("mov rax, 0x0102030405060708")

// 随后切换到两种 RISC-V XLEN，并分别记录一条指令。
riscv.use32()
isa("addi x1, x0, 1")

riscv.use64()
isa("addi x2, x0, 2")

```

这些过程是普通源码操作。它们不会改变早先指令的含义，也不是收尾处理操作。

动态处理器指令

`isa(text)` 只接受一个非空的单行文本值：

```
// 编译期绑定提供需要追加的单行处理器指令。  
const instruction = "nop"  
isa(instruction)
```

指令已经固定写在源码中时，应使用普通处理器指令行；编译期逻辑生成指令文本时，才使用 `isa`。汇编器会按照调用位置当前生效的目标平台解析并编码这条指令。

普通处理器指令不使用编译期语言的语句结束符。不要在普通指令行末添加 `;`，也不要把它放入传给 `isa` 的字符串：

```
nop;           // 无效的处理器指令文本。  
isa("nop;")    // 无效的处理器指令文本。
```

空字符串、其他类型的值、包含换行符的文本或以分号结尾的文本都无效。

逻辑原点与当前位置

`origin(address)` 会改变当前输出区域的逻辑基地址。它不会写出字节、插入填充，也不会改变文件中的物理位置。

`here()` 返回：

```
active region origin + current logical offset
```

```
// 设置逻辑原点，再验证写出两个字节后的当前位置和标号地址。  
origin(0x4000)  
  
header:  
emit.u16(0x1234)  
  
assert(here() == 0x4002)  
assert(label_addr(header) == 0x4000)
```

请在定义依赖新基地址的地址之前调用 `origin`。普通源码和后期布局都可以调用它，但 `defer` 中不能调用。

静态标号与动态标号

源码标号定义固定名称：

```
entry:
```

`label.define(name)` 根据文本值定义同一种锚定标号：

```
// 在新的逻辑原点定义动态标号，并验证它锚定在写出字节之前。
origin(0x5000)

const generated_name = "generated_entry"
label.define(generated_name)
emit.u8(0xaa)

assert(label_addr(generated_name) == 0x5000)
```

生成的名称必须是有效的标号标识符。`label.define` 是普通源码操作，并且要求已经打开输出区域。

`label_addr` 既可以接受直接书写的标号名称，也可以接受文本表达式：

```
label_addr(entry)
label_addr("entry")
label_addr(generated_name)
```

未知符号会被拒绝。值绑定不是标号，不能通过 `label_addr` 获得地址。

处理器指令编码期间的地址稳定性

数据写出、预留和对齐操作会立即更新普通源码游标。处理器指令则先被记录，稍后再编码。处理器指令之后的标号会锚定到对应源码位置，并在指令编码和分支长度调整完成后获得最终偏移。

如果标号依赖前面的处理器指令长度，不要把普通 `label_addr` 的结果直接写死到数据中。应先写出固定宽度的占位值，再在 `defer` 中回填：

```
// 先为最终地址预留固定宽度字段。  
origin(0x4000)  
  
target_field:  
emit.u64(0)  
  
    jmp done  
done:  
    ret  
  
// 指令长度稳定后, 把 done 的最终逻辑地址写回占位字段。  
defer {  
    store.u64(target_field, label_addr(done))  
}
```

执行 `defer` 时, 布局和指令长度都已经稳定, 因此 `label_addr(done)` 是最终逻辑地址。这个规则也适用于位于处理器指令之后的动态标号。

第 7 章：数据写出、预留空间与对齐

语法摘要

形式	语法	结果
固定宽度整数	<code>emit.u8(value)</code> 至 <code>emit.u64(value)</code>	按小端顺序写出一个整数。
固定宽度浮点	<code>emit.f32(value)</code> 、 <code>emit.f64(value)</code>	按小端顺序写出一个 IEEE-754 值。
数据序列	<code>db / dw / dd / dp / dq / dt / ddq / dq / d</code> <code>dqq</code>	写出 1 至 64 字节宽度的整数元素。
字节序列	<code>emit.bytes(value)</code>	原样写出字符串或 <code>bytes</code> 值。
文件字节	<code>emit.file(path, offset?, count?)</code>	写出完整的源文件相对文件或精确范围。
按字节预留	<code>reserve(count)</code>	预留 <code>count</code> 个逻辑字节。
按元素预留	<code>rb / rw / rd / rp / rq / rt / rdq / rqq / r</code> <code>dqq</code>	预留 1 至 64 字节宽度的元素。
固定填充	<code>pad(count, fill?)</code>	恰好写出 <code>count</code> 个填充字节。
绝对位置填充	<code>pad_to(position, fill?)</code>	持续写出字节，直到活动输出区域到达 <code>position</code> 。
对齐	<code>align(boundary, fill?)</code>	前进到下一个对齐位置。

固定宽度整数写出

`emit.u*` 过程接受一个无符号整数，并按小端顺序写出：

调用	接受的数值范围	写出字节数
<code>emit.u8(value)</code>	0 至 0xff	1
<code>emit.u16(value)</code>	0 至 0xffff	2
<code>emit.u32(value)</code>	0 至 0xffffffff	4
<code>emit.u64(value)</code>	0 至 0xffffffffffffffff	8

```
// 依次写出 1、2、4、8 字节宽度的整数，以展示各宽度的小端顺序。
emit.u8(0x11)
emit.u16(0x2233)
emit.u32(0x44556677)
emit.u64(0x8899aabbccddeeff)
```

输出内容的开头为：

```
11 33 22 77 66 55 44 ff ee dd cc bb aa 99 88
```

超出所选宽度数值范围的值会被拒绝，而不会被截断。

数据序列

数据写出简写接受一个或多个实参：

调用	元素宽度	接受的实参
<code>db(values...)</code>	1 字节	整数、字符串和 <code>bytes</code> 值
<code>dw(values...)</code>	2 字节	整数
<code>dd(values...)</code>	4 字节	整数
<code>dp(values...)</code>	6 字节	整数
<code>dq(values...)</code>	8 字节	整数
<code>dt(values...)</code>	10 字节	从 <code>u64</code> 零扩展的整数
<code>ddq(values...)</code>	16 字节	从 <code>u64</code> 零扩展的整数
<code>dqq(values...)</code>	32 字节	从 <code>u64</code> 零扩展的整数
<code>ddqq(values...)</code>	64 字节	从 <code>u64</code> 零扩展的整数

```
// db 原样组合整数字节、字符串和字节序列；其余调用按固定宽度写出整数。
db(0x10, "AZ", b"BC")
dw(0x1234, 0xabcd)
dd(0x10203040)
dq(0x0102030405060708)
```

传给 `db` 的字符串和字节序列会逐字节复制。宽度更大的简写只接受整数，并按小端顺序编码每个元素。调用数据写出简写时不提供实参是无效的。

小于八字节的宽度会检查范围且绝不截断。 `dt` 是 10 字节原始数据，不是 `f80` 或 x87 值。精确的宽位模式应使用 `emit.bytes`。

浮点写出

带小数部分或指数的十进制字面量属于 `f64`。 `f32(value)` 显式窄化有限的 `f64`； `f64(value)` 扩宽 `f32` 或保留 `f64`。

```
const compact: f32 = f32(1.5)
emit.f32(compact)
emit.f64(-0.0)
```

`emit.f32` 和 `emit.f64` 要求实参类型完全匹配，并以小端顺序写出 IEEE-754 位模式。语言不提供隐式整数转换或 `f80` 表面。NaN、Infinity 和溢出会被拒绝；有限下溢与有符号零会被保留。

字节序列

`emit.bytes(value)` 接受一个字符串或 `bytes` 值：

```
// 保存三个原始字节，再与字符串一起原样写入输出内容。
const signature = b"XIR"
emit.bytes(signature)
emit.bytes("ASM")
```

该调用不会添加终止符、长度字段，也不会执行编码转换。如果一次调用需要组合文本、字节序列和整数形式的字节值，请使用 `db`。

预留空间

`reserve(count)` 会让逻辑位置前进 `count` 个字节。以下简写使用固定宽度元素表示数量：

调用	预留的逻辑字节数
<code>rb(count)</code>	<code>count</code>
<code>rw(count)</code>	<code>count * 2</code>
<code>rd(count)</code>	<code>count * 4</code>
<code>rp(count)</code>	<code>count * 6</code>
<code>rq(count)</code>	<code>count * 8</code>
<code>rt(count)</code>	<code>count * 10</code>
<code>rdq(count)</code>	<code>count * 16</code>
<code>rqq(count)</code>	<code>count * 32</code>
<code>rdqq(count)</code>	<code>count * 64</code>

```
// 在两个已初始化字节之间预留三个逻辑字节。
db(0xaa)
reserve(3)
db(0xbb)

// 使用两个 8 字节元素定义尾部预留空间，并验证其逻辑大小。
tail_start:
rq(2)
tail_end:

assert(tail_end - tail_start == 16)
```

预留空间后继续写出已初始化内容时，中间的间隙会以零填充。位于输出区域末尾的预留空间可以只保留为逻辑空间，而不向文件添加字节。第 8 章会介绍决定这一区别的实际文件游标和潜在文件游标。

预留空间的算术运算会经过检查。如果按元素宽度换算后的字节数无法表示，该数量会被拒绝。

填充与对齐

填充始终写出已初始化字节。可选的填充值默认为零，并且必须能放入一个字节。

```
// 依次按固定数量、绝对位置和边界执行填充与对齐，再写出末尾数据。  
db(0x11, 0x22)  
pad(2, 0xaa)  
pad_to(8, 0xbb)  
align(16, 0xcc)  
db(0x33)
```

`pad(count, fill)` 恰好写出 `count` 个字节。

`pad_to(position, fill)` 把 `position` 视为相对于活动输出区域的文件位置。计算填充量时，从下一个需要实际写出的输出位置开始，因此前面的预留空间只会计算一次。目标位置不能位于该起点之前。

`align(boundary, fill)` 会让逻辑位置和文件位置前进到 `boundary` 的下一个整数倍。边界必须是非零的二次幂。如果当前位置已经对齐，则不会写出任何字节。

需要逻辑上的未初始化空间时，应使用预留空间；当填充字节属于文件格式本身时，应使用填充或对齐。

第 8 章：主输出区域、输出区与文件游标

语法摘要

形式	语法	结果
明确主输出区域	<code>region.begin(name, origin, file_offset)</code>	在明确的逻辑地址和文件位置开始一个主输出区域。
文件大小对齐	<code>region.file_align(boundary)</code>	对齐主输出区域的实际文件大小，并结束该区域。
潜在位置续接	<code>output.org(name, origin)</code>	从前一个潜在文件游标开始新的主输出区。
实际位置续接	<code>output.section(name, origin)</code>	从前一个实际文件游标开始新的主输出区。
虚拟输出区	<code>virtual.begin([origin])</code>	开始一个具有逻辑地址但不产生文件字节的嵌套输出区。
结束虚拟输出区	<code>virtual.end()</code>	返回 <code>virtual.begin</code> 之前处于活动状态的输出区。
区域逻辑原点	<code>region_base()</code>	返回当前主输出区域的逻辑基地址。
当前文件位置	<code>file_offset()</code>	返回当前主输出区域的绝对实际文件游标。
实际文件游标	<code>file_cursor_real()</code>	返回当前实际文件字节之后的下一个绝对位置。
潜在文件游标	<code>file_cursor_potential()</code>	返回由逻辑游标推导出的绝对文件位置。
尾部预留空间	<code>tail_reserve_size()</code>	返回当前主输出区域尾部尚未写入文件的逻辑字节数。

坐标模型

主输出区域分别维护逻辑坐标和文件坐标：

坐标	含义
逻辑地址	标号、 <code>here()</code> 、指令和地址回填项使用的地址。
实际文件游标	当前输出内容中已经存在的字节之后的绝对文件位置。
潜在文件游标	全部逻辑输出内容所推导出的绝对文件位置，其中包括尾部预留空间。

对于当前主输出区域：

```
here()                = region_base() + logical offset
file_cursor_real()    = region file offset + real relative cursor
file_cursor_potential() = region file offset + logical offset
tail_reserve_size()   = logical bytes beyond the real relative cursor
```

`file_offset()` 是查询当前实际文件游标的常用名称，返回值与 `file_cursor_real()` 相同。

实际写出的字节会同时推进两个文件游标。尾部预留空间只推进逻辑游标和潜在文件游标：

```
// 在逻辑地址 0x4000、文件偏移 0x20 处开始载荷区域。
region.begin("payload", 0x4000, 0x20)

// 写出一个实际文件字节，再增加三个尾部预留字节。
emit.u8(0x11)
reserve(3)

// 逻辑位置包含预留空间，实际文件游标只越过已写入的字节。
assert(region_base() == 0x4000)
assert(here() == 0x4004)
assert(file_offset() == 0x21)
assert(file_cursor_real() == 0x21)
assert(file_cursor_potential() == 0x24)
assert(tail_reserve_size() == 3)
```

如果同一主输出区域在预留空间之后继续写出已经初始化的内容，中间的间隔就会成为输出文件的一部分。实际文件游标会先追上潜在位置，再越过新写出的字节。

开始明确的主输出区域

`region.begin(name, origin, file_offset)` 开始一个新的主输出区域：

- `name` 用于在诊断信息和布局信息中标识该区域；
- `origin` 是该区域的逻辑基地址；
- `file_offset` 是该区域在输出文件中的绝对起始位置。

新区域的相对逻辑游标和相对文件游标都从零开始。各参数彼此独立，因此一种格式可以把紧凑的文件范围映射到不同的逻辑地址。

`region.begin` 不会根据前一个区域推断连续位置。如果新区域必须续接已有布局，应当在调用之前查询相应的文件游标。

实际位置与潜在位置续接

`output.section(name, origin)` 和 `output.org(name, origin)` 都会开始新的主输出区，并为它设置新的逻辑原点。两者的区别在于从前一个输出区选择哪个文件位置：

过程	新文件位置	对尾部预留空间的影响
<code>output.section</code>	前一个实际文件游标	不把尚未写入文件的尾部预留空间计入后续文件布局。
<code>output.org</code>	前一个潜在文件游标	后续写出字节时，将尾部预留空间保留为文件间隔。

```
// 头部写出一个字节，并留下三个尾部预留字节。
region.begin("header", 0x4000, 0x20)
emit.u8(0x11)
reserve(3)

// 从实际文件游标续接，因此去除前面的尾部预留空间。
output.section("trimmed", 0x5000)
assert(file_offset() == 0x21)
emit.u8(0x22)
reserve(2)

// 从潜在文件游标续接，因此保留两个预留字节形成文件间隔。
output.org("preserved", 0x6000)
assert(file_offset() == 0x24)
emit.u8(0x33)
```

第一次切换会把 `trimmed` 放在字节 `0x11` 之后，三个尾部预留字节不会写入文件。第二次切换从潜在位置开始，因此 `0x22` 与 `0x33` 之间的两个预留字节会成为文件中间的间隔。

这两个过程都要求当前存在尚未结束的主输出区域。在虚拟输出区内调用它们属于无效操作。

结束并对齐文件区域

`region.file_align(boundary)` 会把当前主输出区域的实际文件大小向上取整到非零的 2 的幂边界，并结束该区域，禁止继续写出内容：

```
// 写出一个字节后，把主输出区域的实际文件大小对齐到八字节。
region.begin("record", 0x8000, 0)
emit.u8(0x7f)
region.file_align(8)

// 文件对齐只扩展实际文件范围，不会推进潜在文件游标。
assert(file_cursor_real() == 8)
assert(file_cursor_potential() == 1)
```

对齐产生的文件尾部属于输出文件，并使用零填充。因此，实际文件游标可以越过潜在文件游标。调用成功后，不能在该区域中继续写出、预留或对齐数据；必须开始另一个主输出区域或输出区才能继续。

虚拟输出区

`virtual.begin()` 会在当前逻辑地址开始一个嵌套的虚拟输出区。`virtual.begin(origin)` 则使用明确的逻辑原点。虚拟输出区可以定义标号、预留空间并保存临时字节，但不会向最终文件贡献任何字节。

```
// 主输出区域先写出一个字节，虚拟输出区从下一逻辑地址开始。
region.begin("main", 0x7000, 0)
emit.u8(0xaa)

// 虚拟输出区可以推进自身逻辑位置，但不会推进主输出区域。
virtual.begin()
assert(region_base() == 0x7001)
emit.u16(0x1234)
reserve(2)
assert(here() == 0x7005)
virtual.end()

// 返回后恢复主输出区域的逻辑原点和当前位置。
assert(region_base() == 0x7000)
assert(here() == 0x7001)
emit.u8(0xbb)
```

最终文件只包含 `aa bb`。`virtual.end()` 会恢复先前的输出区及其游标。虚拟输出区可以嵌套，但每个 `virtual.begin` 都必须有一个对应的 `virtual.end`。

文件游标查询描述的是主输出文件，不能用来推断虚拟输出区的存储范围。虚拟输出区处于活动状态时，应当使用逻辑地址和标号。

错误条件

区域 API 会拒绝以下情况：

- `region.file_align` 的边界为零或不是 2 的幂；
- `region.file_align` 已经结束当前主输出区域后，仍尝试继续写出内容；
- 虚拟输出区处于活动状态时调用 `output.section` 或 `output.org`；
- 在没有对应开始操作时调用 `virtual.end`；
- 源文件结束时仍有尚未关闭的虚拟输出区。

第 9 章：读取、写入与最终区域信息

语法摘要

形式	语法	结果
读取整数	<code>load.u8(address)</code>	读取一个字节。
读取整数	<code>load.u16(address)</code>	读取一个小端顺序的 16 位整数。
读取整数	<code>load.u32(address)</code>	读取一个小端顺序的 32 位整数。
读取整数	<code>load.u64(address)</code>	读取一个小端顺序的 64 位整数。
读取字节	<code>load.bytes(address, count)</code>	返回 <code>count</code> 个字节。
写入整数	<code>store.u8(address, value)</code>	写入一个字节。
写入整数	<code>store.u16(address, value)</code>	写入一个小端顺序的 16 位整数。
写入整数	<code>store.u32(address, value)</code>	写入一个小端顺序的 32 位整数。
写入整数	<code>store.u64(address, value)</code>	写入一个小端顺序的 64 位整数。
写入字节	<code>store.bytes(address, value)</code>	写入字符串或字节序列。
区域文件偏移	<code>region_file_offset(address)</code>	返回区域的最终文件偏移。
区域文件大小	<code>region_file_size(address)</code>	返回区域最终存在于文件中的字节数。
区域逻辑大小	<code>region_logical_size(address)</code>	返回区域的最终逻辑大小。

常规访问与收尾阶段访问

读取和写入操作在两个阶段访问不同的数据状态：

阶段	访问的数据
常规求值阶段	当前模块布局中已经存在的字节。
<code>defer</code> 收尾阶段	稳定的最终输出内容中的字节。

常规写入可以在目标字节已经生成后初始化或替换这些字节。收尾阶段写入适合处理依赖最终布局的字段，例如大小、偏移、校验和以及目录项。

```

// 创建逻辑地址从 0x3000、文件偏移从 0 开始的数据区域。
region.begin("data", 0x3000, 0)

// 先写出不同宽度整数和字节序列所需的存储位置。
byte_slot:
emit.u8(0)

word_slot:
emit.u16(0)

dword_slot:
emit.u32(0)

qword_slot:
emit.u64(0)

bytes_slot:
db(0, 0, 0)

// 常规求值阶段可以修改已经写出的字节。
store.u8(byte_slot, 0xaa)
store.u16(word_slot, 0x2233)

// 立即读取刚才写入的值，确认小端整数访问正确。
assert(load.u8(byte_slot) == 0xaa)
assert(load.u16(word_slot) == 0x2233)

defer {
    // 布局稳定后，回填其余整数和三字节标记。
    store.u32(dword_slot, 0x44556677)
    store.u64(qword_slot, 0x0102030405060708)
    store.bytes(bytes_slot, b"XYZ")

    // 收尾块能同时看到常规阶段和本收尾块完成的写入。
    assert(load.u8(byte_slot) == 0xaa)
    assert(load.u16(word_slot) == 0x2233)
    assert(load.u32(dword_slot) == 0x44556677)
    assert(load.u64(qword_slot) == 0x0102030405060708)
    assert(load.bytes(bytes_slot, 3) == b"XYZ")
}

```

整数形式只接受其指定宽度能够表示的值。读取和写入必须完全落在最终文件中实际存在的字节范围内。这些操作不会分配存储空间、扩大区域或改变布局。

标号与输出区身份

仅有逻辑地址不一定足以确定文件中的字节。两个输出区可以使用相互重叠的逻辑地址范围，同时占用不同的文件范围。

当读取、写入或最终区域信息表达式直接包含标号时，XIRASM 会保留该标号的输出区身份，并把它与逻辑地址结合起来。即使另一个输出区使用相同地址，也能据此选择正确的文件范围。

```
// 外层区域从逻辑地址 0x1000 和文件偏移 0 开始。
region.begin("outer", 0x1000, 0)

outer:
emit.u32(0x44332211)
// 尾部预留空间扩大逻辑大小，但不会立即形成文件字节。
reserve(4)

// 新输出区从实际文件游标继续，因此紧接 outer 已写出的四个字节。
output.section("inner", 0x1002)

inner:
emit.u16(0x6655)

defer {
    // 两个标号分别查询各自输出区的最终文件偏移和大小。
    assert(region_file_offset(outer) == 0)
    assert(region_file_size(outer) == 4)
    assert(region_logical_size(outer) == 8)

    assert(region_file_offset(inner) == 4)
    assert(region_file_size(inner) == 2)
    assert(region_logical_size(inner) == 2)

    // inner 与 outer + 2 地址相同，但输出区身份让它们访问不同字节。
    assert(load.u16(inner) == 0x6655)
    store.u16(inner, 0x8877)
    assert(load.u16(inner) == 0x8877)
    assert(load.u16(outer + 2) == 0x4433)
}
```

这里的 `outer + 2` 与 `inner` 具有相同逻辑地址。两个标号保留了不同的输出区身份，因此引用的是不同字节。

普通整数不带输出区身份。如果逻辑地址范围可能重叠，应当在访问表达式中保留标号，不要先把它转换成数值变量或函数参数。

最终区域信息

只有布局和最终输出都稳定后，才能使用以下三种最终区域信息；通常应当在 `defer` 中查询：

表达式	含义
<code>region_file_offset(label)</code>	区域在文件中的绝对起始位置，也就是最终文件偏移。
<code>region_file_size(label)</code>	区域最终存在于文件中的字节数，包括文件对齐产生的字节。
<code>region_logical_size(label)</code>	区域占用的逻辑地址范围，包括尾部预留空间。

未初始化的预留空间使这一区别十分重要。在前一个示例中，`outer` 的逻辑大小为八字节，文件大小只有四字节。`output.section` 从实际文件游标开始 `inner`，因此四字节尾部预留空间不会写入文件。

这些预留地址仍属于逻辑范围，但不是可以读取或写入的最终输出字节。收尾处理不能通过读取或写入，把已经裁剪的尾部预留空间变成实际输出。

地址与范围错误

以下情况会被这些 API 拒绝：

- 整数写入值超出所选宽度能够表示的范围；
- 读取或写入超出实际写入的文件字节范围；
- 访问已经从文件布局中裁剪的尾部预留空间；
- 在最终输出稳定之前查询最终区域信息；
- 最终区域信息查询的地址不属于任何最终逻辑区域；
- 一个标号表达式组合了来自不同输出区的标号。

使用数据写出、预留和区域操作定义布局。读取、写入和最终区域信息只应用于检查或回填布局已经建立的存储空间。

第 10 章：文本、转换与标号名称

语法摘要

函数	结果	说明
<code>lengthof(value)</code>	<code>u64</code>	返回字节长度，或整数的十进制位数。
<code>len(value)</code>	<code>u64</code>	返回字节数、列表项数或映射条目数。
<code>to_string(value)</code>	<code>string</code>	把整数、布尔值、字符串或字节序列转换为文本。
<code>trim(text)</code>	<code>string</code>	删除文本两端的 ASCII 空格、制表符、回车符和换行符。
<code>lower(text)</code>	<code>string</code>	把 ASCII 字母转换为小写。
<code>upper(text)</code>	<code>string</code>	把 ASCII 字母转换为大写。
<code>starts_with(text, prefix)</code>	<code>bool</code>	检查字符串是否以指定前缀开头。
<code>ends_with(text, suffix)</code>	<code>bool</code>	检查字符串是否以指定后缀结尾。
<code>contains(value, needle)</code>	<code>bool</code>	在字符串或字节序列中查找内容。
<code>replace(text, needle, replacement)</code>	<code>string</code>	替换字符串中所有互不重叠的匹配项。
<code>split(text, separator)</code>	<code>list</code>	按分隔符把字符串拆分为字符串列表。
<code>join(parts, separator)</code>	<code>string</code>	使用分隔符连接字符串列表。
<code>sym.join(parts...)</code>	<code>string</code>	转换并直接拼接多个值，不插入分隔符。
<code>sym.unique(prefix)</code>	<code>string</code>	返回本次汇编中不会重复的生成名称。

本章函数都是普通的编译期表达式。它们会创建新值，不会修改传入的值。

长度查询

`lengthof` 和 `len` 回答的问题不同：

输入	<code>lengthof</code>	<code>len</code>
<code>string</code>	字节长度	字节长度
<code>bytes</code>	字节长度	字节长度
无符号整数	十进制位数	不接受
<code>list</code>	不接受	列表项数
<code>map</code>	不接受	映射条目数

字符串长度按字节计算，而不是按书写字符的数量计算。对于整数，`lengthof(0)` 的结果是 `1`，因为它的十进制文本是 `"0"`。

```
// 查询整数与字符串的长度，并把 4096 的十进制位数写成文本。
assert(lengthof(4096) == 4)
assert(lengthof("XIRASM") == 6)
assert(len(split("code::data::", "::")) == 3)

db(to_string(lengthof(4096)))
```

输出是 ASCII 字节 `0x34`，表示文本 `"4"`。

转换为文本

`to_string(value)` 使用以下转换规则：

输入	文本形式
整数	无符号十进制
布尔值	<code>"true"</code> 或 <code>"false"</code>
字符串	内容不变的副本
字节序列	小写十六进制，每个字节使用两位数字

```
// 检查整数、布尔值、字符串和字节序列的文本转换结果。
assert(to_string(42) == "42")
assert(to_string(true) == "true")
assert(to_string("ready") == "ready")
assert(to_string(b"AZ") == "415a")
```

结构体、列表和映射不会自动转换。需要文本时，应分别转换其中的字段或元素。

文本变换与查找

`trim`、`lower` 和 `upper` 处理 ASCII 文本。`trim` 只把空格、水平制表符、回车符和换行符视为两端空白。大小写转换不会改变非 ASCII 字节。

`starts_with` 和 `ends_with` 接受字符串。`contains` 可以接受两个字符串，也可以接受两个字节序列；两个参数必须是同一种值。

```
// 清理并统一库名称的大小写，再检查其中的各个文本片段。
const name: string = lower(trim(" Kernel32.DLL "))

assert(name == "kernel32.dll")
assert(starts_with(name, "kernel"))
assert(ends_with(name, ".dll"))
assert(contains(name, "32"))

db(name)
```

替换、拆分与连接

`replace` 会替换 `needle` 的每个互不重叠的匹配项。`needle` 不能为空。

`split` 会在非空分隔符的每个匹配位置拆分字符串。空字段会被保留，包括相邻分隔符或末尾分隔符产生的字段。因此，空输入字符串会得到只包含一个空字符串的列表。

`join` 要求列表中的每一项都是字符串。空列表会得到空字符串。

```
// 拆分包含末尾分隔符的文本，再连接字段并执行全部匹配项替换。
const fields: list = split("red::green::", "::")

assert(len(fields) == 3)
assert(join(fields, "|") == "red|green|")
assert(replace("one fish, one fish", "one", "two") == "two fish, two fish")

db(join(fields, "|"))
```

构造标号名称

`sym.join(parts...)` 会按照与 `to_string` 相同的规则转换整数、布尔值、字符串和字节序列参数，然后直接拼接结果，不自动插入分隔符：

```
// 拼接固定文本、整数和布尔值，得到可直接定义的标号名称。
const slot: string = sym.join("_slot_", 12, "_", true)

assert(slot == "_slot_12_true")
label.define(slot)
emit.u8(0xaa)
```

需要分隔符时，应把所需标点明确写在参数中。

`sym.unique(prefix)` 每调用一次，都会返回本次汇编中不同的生成字符串。生成顺序是确定的，但后缀格式不是源码可以依赖的约定。应保存返回的字符串并始终使用该值，不要自行重新构造名称。

```
// 使用相同前缀生成两个不同名称，并分别定义标号和写出数据。
const first: string = sym.unique("_temporary")
const second: string = sym.unique("_temporary")

assert(first != second)

label.define(first)
emit.u8(0x11)

label.define(second)
emit.u8(0x22)
```

`sym.unique` 只保证名称不重复，不保证名称符合标号语法。把结果传给 `label.define` 时，应选择能使完整结果成为有效标号名称的前缀。

错误条件

以下输入会被拒绝：

- 向 `lengthof` 传入字符串、字节序列和无符号整数以外的值；
- 向 `len` 传入字符串、字节序列、列表和映射以外的值；
- 向 `to_string` 或 `sym.join` 传入聚合值；
- 向 `contains` 同时传入字符串和字节序列；
- 向 `replace` 传入空的 `needle`，或向 `split` 传入空分隔符；
- 向 `join` 传入包含非字符串项的列表；
- 把生成名称用于 `label.define` 时，该名称不符合标号语法。

第 11 章：字节序列

语法摘要

函数	结果	说明
<code>bytes.new()</code>	<code>bytes</code>	创建空字节序列。
<code>bytes.push(value, byte)</code>	<code>bytes</code>	在末尾追加一个字节。
<code>bytes.concat(left, right)</code>	<code>bytes</code>	连接两段字节序列。
<code>bytes.repeat(count, byte)</code>	<code>bytes</code>	创建由同一字节重复 <code>count</code> 次组成的序列。
<code>bytes.le(value, width)</code>	<code>bytes</code>	取整数的低 <code>width</code> 个字节，并按小端顺序编码。
<code>bytes.insert(value, index, addition)</code>	<code>bytes</code>	在从零开始的字节索引处插入字节。
<code>bytes.replace(value, index, count, replacement)</code>	<code>bytes</code>	替换一段字节范围。
<code>bytes.eq(left, right)</code>	<code>bool</code>	检查两段字节序列是否完全相同。
<code>bytes.hex(value)</code>	<code>string</code>	把字节序列转换为小写十六进制文本。
<code>bytes.from_hex(text)</code>	<code>bytes</code>	解析大写或小写的十六进制文本。

每项操作都会返回一个新值，不会修改作为输入的字节序列。因此，同一个中间值可以安全地用于多个后续构造。

创建并组合字节序列

`bytes.new()` 返回空字节序列。`bytes.push` 在末尾追加一个取值范围为 0 到 255 的整数。`bytes.concat` 连接两段字节序列，`bytes.repeat` 则创建由同一个字节填充的序列。

```
// 从空字节序列开始，追加操作不会改变 empty 本身。
const empty: bytes = bytes.new()
const tag: bytes = bytes.push(empty, 0x7f)
const padding: bytes = bytes.repeat(3, 0xaa)
const result: bytes = bytes.concat(tag, padding)

// 检查原值仍为空，并确认连接后的精确字节内容。
assert(len(empty) == 0)
assert(bytes.eq(result, bytes.from_hex("7faaaaaa")))

// 只把最终构造完成的字节序列写入输出内容。
emit.bytes(result)
```

调用 `bytes.push` 后，原来的 `empty` 值仍然是空字节序列。

小端整数编码

`bytes.le(value, width)` 会生成零到八个字节。第一个字节保存整数的最低八位。当 `width` 小于八时，更高位会被丢弃。

```
// 文件标记按原有字节顺序保留，版本号编码为两个小端字节。
const magic: bytes = bytes.from_hex("58495200")
const version: bytes = bytes.le(3, 2)
const header: bytes = bytes.concat(magic, version)

// 检查完整文件头、截断高位以及零字节宽度的结果。
assert(bytes.hex(header) == "584952000300")
assert(bytes.eq(bytes.le(0x1234, 1), bytes.from_hex("34")))
assert(bytes.eq(bytes.le(0x1234, 0), bytes.new()))

// 写出由文件标记和版本字段组成的文件头。
emit.bytes(header)
```

输入整数是无符号编译期值。`bytes.le` 不会进行有符号扩展，也不会检查被丢弃的高位是否为零。

插入与替换字节范围

字节索引从零开始。`bytes.insert(value, index, addition)` 接受从零到 `len(value)` 的任意索引，其中 `len(value)` 表示紧接最后一个字节之后的位置。

`bytes.replace(value, index, count, replacement)` 从字节索引 `index` 开始移除 `count` 个字节，再在同一位置插入 `replacement`：

- `count` 为零时，只插入新字节，不移除原有字节；

- `replacement` 为空时，删除选中的字节范围；
- 同时指定移除数量和非空替换内容时，执行普通替换。

```
// 在 ABC 的字节索引 1 处插入连字符，再替换索引 2 处的一个字节。
const base: bytes = b"ABC"
const marked: bytes = bytes.insert(base, 1, b"-")
const patched: bytes = bytes.replace(marked, 2, 1, b"Z")
const trailer: bytes = bytes.repeat(2, 0xff)
const result: bytes = bytes.concat(
    bytes.push(bytes.new(), 0x7f),
    bytes.concat(patched, trailer)
)

// 原始字节序列保持不变；零长度替换用于插入，空替换内容用于删除。
assert(bytes.eq(base, b"ABC"))
assert(bytes.eq(bytes.replace(b"AB", 1, 0, b"-"), b"A-B"))
assert(bytes.eq(bytes.replace(b"ABCD", 1, 2, bytes.new()), b"AD"))

// 写出前导字节、修改后的正文和两个尾部字节。
emit.bytes(result)
```

这个示例写出 `7f 41 2d 5a 43 ff ff`。

十六进制转换与相等比较

`bytes.from_hex` 接受字符数为偶数的十六进制字符串。字母形式的十六进制数字可以使用大写或小写，每两个字符生成一个字节。空字符串会生成空字节序列。

`bytes.hex` 执行反向转换，并且始终使用小写数字。需要明确比较两段字节序列的精确内容时，使用 `bytes.eq`。

```
// 把大小写混合的十六进制字符串解析为字节序列，再转换为规范的小写文本。
const value: bytes = bytes.from_hex("DEadBEEF")
const text: string = bytes.hex(value)

// 检查文本形式、往返转换和空字符串的解析结果。
assert(text == "deadbeef")
assert(bytes.eq(bytes.from_hex(text), value))
assert(bytes.eq(bytes.from_hex(""), bytes.new()))

// 写出解析得到的四个字节，而不是十六进制字符串本身。
emit.bytes(value)
```

错误条件

字节辅助接口会拒绝以下输入：

- 字节参数不在 0 到 255 的范围内；
- 需要字节序列的位置传入了非 `bytes` 值；
- `bytes.le` 的宽度大于八；
- 插入索引大于当前字节数量；
- 替换范围超过当前字节数量；
- 十六进制字符串包含奇数个字符；
- 十六进制字符串包含非十六进制字符；
- 函数参数数量不正确。

第 12 章：列表与映射

列表和映射都是编译期值集合。表达式接口在添加、替换或组合值时返回新集合，所有输入集合保持不变。当算法需要逐步构造集合时，也可以使用仅作用于直接 `let` 绑定的显式可变语句。

列表函数

函数	结果	说明
<code>list.new()</code>	<code>list</code>	创建空列表。
<code>list.of(values...)</code>	<code>list</code>	使用零个或多个值创建列表。
<code>list.push(value, item)</code>	<code>list</code>	在列表末尾追加一个元素。
<code>list.concat(left, right)</code>	<code>list</code>	连接两个列表。
<code>list.get(value, index)</code>	值	返回从零开始的索引所指向的元素。
<code>list.set(value, index, item)</code>	<code>list</code>	返回替换了一个元素的新列表。
<code>list.slice(value, start, count)</code>	<code>list</code>	返回一段连续范围组成的新列表。
<code>list.eq(left, right)</code>	<code>bool</code>	按顺序递归比较两个列表是否相等。

以下语句直接更新 `let` 绑定，不返回值：

语句	说明
<code>list.push_mut(target, item);</code>	把深拷贝后的元素追加到 <code>target</code> 。
<code>list.set_mut(target, index, item);</code>	使用深拷贝后的值替换已有元素。

`list.get` 和 `list.set` 要求索引小于列表长度。`list.slice` 的起始索引可以从零取到列表长度，但所选范围不能超出列表。允许在列表末尾取得长度为零的切片。

```
// 每次列表操作都返回新列表, base 和 extended 不会被后续操作修改。
const base: list = list.of(1, 2, 3)
const extended: list = list.push(base, 4)
const patched: list = list.set(extended, 1, 0xaa)
const middle: list = list.slice(patched, 1, 2)
const combined: list = list.concat(list.of(0x10, 0x11), middle)

// 索引从零开始; 列表相等比较同时检查元素顺序和内容。
assert(list.get(base, 1) == 2)
assert(list.eq(base, list.of(1, 2, 3)))
assert(list.eq(patched, list.of(1, 0xaa, 3, 4)))
assert(list.eq(middle, list.of(0xaa, 3)))

// 按 combined 中的顺序逐项写出字节。
for value in combined {
    emit.u8(value)
}
```

这段代码会写出 `10 11 aa 03`。列表相等比较与元素顺序有关，并会递归比较嵌套的列表、映射、结构体、字符串和字节序列。

映射函数

函数	结果	说明
<code>map.new()</code>	<code>map</code>	创建空映射。
<code>map.set(value, key, item)</code>	<code>map</code>	添加字符串键，或替换已有键对应的值，并返回新映射。
<code>map.has(value, key)</code>	<code>bool</code>	检查指定字符串键是否存在。
<code>map.get(value, key)</code>	值	返回必需键对应的值。
<code>map.get_or(value, key, fallback)</code>	值	键存在时返回对应值，否则返回给定的后备值。
<code>map.keys(value)</code>	<code>list</code>	按插入顺序返回键列表。
<code>map.values(value)</code>	<code>list</code>	按与键相同的插入顺序返回值列表。
<code>map.eq(left, right)</code>	<code>bool</code>	递归比较两个映射是否相等，不考虑键的顺序。

`map.set_mut(target, key, item)`；会在由直接 `let` 绑定的映射中添加或替换深拷贝后的值。键必须是字符串；替换已有键时，该键的插入位置保持不变。

映射的键必须是字符串。添加新键时，该项会追加到插入顺序末尾；替换已有键时，该键原有的位置保持不变。因此，`map.keys` 和 `map.values` 返回的两个列表——对应，相同索引表示同一个映射项。

```

// map.set 返回新映射; empty、first 和 configured 都保持原值。
const empty: map = map.new()
const first: map = map.set(empty, "arch", "x64")
const configured: map = map.set(first, "mode", 64)
const updated: map = map.set(configured, "arch", "rv64")
const complete: map = map.set(updated, "tags", list.of("asm", "dsl"))

// 检查键、后备值和插入顺序, 并确认替换不会改变旧映射。
assert(len(empty) == 0)
assert(map.has(complete, "arch"))
assert(!map.has(complete, "missing"))
assert(map.get(first, "arch") == "x64")
assert(map.get(updated, "arch") == "rv64")
assert(map.get_or(complete, "missing", "default") == "default")
assert(list.eq(map.keys(complete), list.of("arch", "mode", "tags")))
assert(list.eq(map.get(complete, "tags"), list.of("asm", "dsl")))

// 以不同顺序插入相同的键和值。
const reordered: map = map.set(
  map.set(
    map.set(map.new(), "tags", list.of("asm", "dsl")),
    "mode",
    64
  ),
  "arch",
  "rv64"
)

// map.eq 不要求插入顺序相同, 最后写出 mode 对应的字节。
assert(map.eq(complete, reordered))
emit.u8(map.get(complete, "mode"))

```

这段代码会写出 40。map.eq 会比较键和嵌套值, 但不要求两个映射具有相同的插入顺序。

可变集合绑定规则

list.push_mut、list.set_mut 和 map.set_mut 的目标必须是直接标识符, 并解析到词法作用域中最近的 let 绑定。目标不能是 const、临时表达式、调用结果、字段访问或类型不匹配的值。普通 lowering 阶段允许更新顶层 let; 值函数中只能更新本次调用内部的局部绑定。这些接口是语句, 不能作为表达式使用, 也不能出现在 defer 或 late_layout 块中。

插入的值会在提交修改之前完成深拷贝, 因此目标之前的副本以及插入到其他集合中的值都保持独立。分配失败时, 目标集合保持原样。

错误条件

集合辅助接口会拒绝以下情况：

- 向 `list.*` 操作传入非列表参数；
- 向 `map.*` 操作传入非映射参数；
- 函数参数数量错误；
- 列表索引超出可用元素范围；
- 列表切片超出列表范围；
- 映射键不是字符串；
- 向 `map.get` 传入不存在的键。

如果键不存在属于预期情况，应使用 `map.has` 或 `map.get_or`。

第 13 章：文件与结构化数据

语法摘要

函数	结果	说明
<code>fs.exists(path)</code>	<code>bool</code>	检查指定路径能否定位到文件。
<code>fs.read_text(path)</code>	<code>string</code>	把整个文件读取为字符串。
<code>fs.read_bytes(path)</code>	<code>bytes</code>	把整个文件读取为字节序列。
<code>fs.read_bytes(path, offset, count)</code>	<code>bytes</code>	读取一段长度明确的字节范围。
<code>emit.file(path)</code>	语句	写出完整的源文件相对文件。
<code>emit.file(path, offset, count)</code>	语句	写出精确的文件字节范围。
<code>json.parse(value)</code>	值	解析字符串或字节序列中的 JSON。
<code>json.file(path)</code>	值	读取并解析 JSON 文件。
<code>toml.parse(value)</code>	<code>map</code>	解析字符串或字节序列中的 TOML 文档。
<code>toml.file(path)</code>	<code>map</code>	读取并解析 TOML 文件。

文件路径遵循与 `include` 和 `import` 相同的受控路径规则。相对路径以包含该调用的源文件为基准。因此，把调用移动到嵌套模块中时，它所使用的相对数据路径基准也会随之改变。

各阶段的可用性

操作	常规求值阶段	<code>late_layout</code>	<code>defer</code>
<code>fs.exists</code> 、 <code>fs.read_text</code> 、 <code>fs.read_bytes</code> 、 <code>emit.file</code>	可用	不可用	不可用
<code>json.file</code> 、 <code>toml.file</code>	可用	不可用	不可用
<code>json.parse</code> 、 <code>toml.parse</code>	可用	可在值表达式中使用	可在值表达式中使用

文件操作需要当前阶段能够按照源文件位置解析路径。`late_layout` 和 `defer` 阶段不能重新访问源文件。需要在后期使用结构化数据时，应当在常规求值阶段完成文件读取与解析，并保留得到的值。

`parse` 函数不访问文件系统，只处理传入的字符串或字节序列。

检查与读取文件

路径无法定位到文件时，`fs.exists` 返回 `false`。读取函数遇到缺失文件时则会报告错误。

`fs.read_text` 保留文件的完整内容，不会附加结尾零字节。`fs.read_bytes` 返回内容相同的字节序列。

```
// 先检查必需文件与可选文件是否存在。
assert(fs.exists("payload.bin"))
assert(fs.exists("banner.txt"))
assert(!fs.exists("optional.bin"))

// 从二进制文件读取四字节文件头，并把文本文件读取为字符串。
const header: bytes = fs.read_bytes("payload.bin", 0, 4)
const banner: string = fs.read_text("banner.txt")

// 按读取顺序写出二进制文件头和文本内容。
emit.bytes(header)
emit.bytes(banner)
```

如果 `payload.bin` 包含 `XIR!`，而 `banner.txt` 包含 `ready` 和紧随其后的换行符，示例会写出：

```
58 49 52 21 72 65 61 64 79 0a
```

范围读取形式使用从零开始的偏移和字节数量。整个范围必须位于文件以内。在文件末尾读取长度为零的范围是有效操作。

`emit.file` 使用与 `fs.read_bytes` 完全相同的解析器和范围规则，但会直接写入输出，而不是返回 `bytes` 值。

JSON 值

JSON 值按下表转换为编译期值：

JSON 值	编译期值
<code>null</code>	<code>void</code>
布尔值	<code>bool</code>
字符串	<code>string</code>
非负整数	整数
数组	<code>list</code>
对象	<code>map</code>

`json.file` 在一次调用中完成读取和解析。`json.parse` 接受已经包含 JSON 内容的字符串或字节序列。

```
// 分别直接读取配置文件，以及读取字节后再次解析同一份配置。
const config: map = json.file("config.json")
const parsed_again: map = json.parse(fs.read_bytes("config.json"))
const values: list = map.get(config, "values")

// 两种读取方式应得到相同映射，值为 null 的键也仍然存在。
assert(map.eq(config, parsed_again))
assert(map.get(config, "enabled"))
assert(map.has(config, "nothing"))

// 从映射中读取文本、整数和列表，并按配置内容写出字节。
emit.bytes(map.get(config, "name"))
emit.u8(map.get(config, "bits"))

for value in values {
    emit.u8(value)
}
```

对应输入为：

```
{
  "name": "XR",
  "bits": 64,
  "enabled": true,
  "values": [1, 2],
  "nothing": null
}
```

示例会写出 `58 52 40 01 02`。

JSON 浮点数、负整数、重复的对象键、格式错误的输入，以及超出支持范围的整数都会被拒绝。

TOML 值

TOML 文档转换为映射。嵌套表转换为嵌套映射，数组转换为列表，字符串、布尔值和非负整数保持对应的编译期值类型。

```
// 分别直接读取配置文件，以及读取文本后再次解析同一份配置。
const config: map = toml.file("config.toml")
const parsed_again: map = toml.parse(fs.read_text("config.toml"))
const target: map = map.get(config, "target")
const values: list = map.get(parsed_again, "values")

// 两种读取方式应得到相同映射，并保留布尔配置。
assert(map.eq(config, parsed_again))
assert(map.get(config, "enabled"))

// 从顶层映射和嵌套映射中读取输出字段。
emit.bytes(map.get(config, "name"))
emit.u8(map.get(target, "bits"))

for value in values {
    emit.u8(value)
}
```

对应输入为：

```
# 顶层配置值。
name = "XR"
enabled = true
values = [3, 4]

# 目标平台配置表。
[target]
bits = 32
```

示例会写出 58 52 20 03 04。

浮点数、时间戳、负整数、格式错误的文档和重复键都会在转换过程中被拒绝。

错误条件

以下情况会被这些辅助接口拒绝：

- 文件路径不是字符串；
- 传给读取函数或 `*.file` 函数的文件不存在；
- 字节范围超出文件边界；

- 传给 `json.parse` 或 `toml.parse` 的值既不是字符串，也不是字节序列；
- JSON 或 TOML 格式错误；
- 对象或表中存在重复键；
- 结构化数据包含无法表示为编译期值的内容；
- 函数实参数量不正确。

只有文件缺失属于预期情况时，才需要在读取前调用 `fs.exists`。

第 14 章：词法单元与模式匹配

语法摘要

功能	调用形式	结果
对源代码形式的文本进行词法分析	<code>tokens.of(source)</code>	由词法单元字符串组成的 <code>list</code>
呈现词法单元字符串	<code>tokens.join(tokens)</code>	规范形式的 <code>string</code>
匹配词法单元结构	<code>match.tokens(pattern, input)</code>	结果 <code>map</code>

`tokens.of` 接受字符串或已有的字符串列表。传入字符串时会进行词法分析；传入列表时会验证每一项并复制该列表。

`tokens.join` 接受所有元素都是字符串的列表。

`match.tokens` 的模式和输入都可以是字符串或字符串列表。模式为列表时，每一项必须是一个完整的模式片段。输入为列表时，其中的词法单元会直接参与匹配，不会再次进行词法分析。

词法分析

`tokens.of` 用于处理具有源代码结构的文本。它会分离名称、字面量、标点、括号和运算符，并丢弃不影响结构的空白。

```
// 把一条具有地址表达式的文本拆成词法单元列表。
const input: list = tokens.of("load rax, [rbx + 4]")

// 空白不会成为词法单元，标点和括号会被单独保留。
assert(len(input) == 8);
assert(list.get(input, 0) == "load");
assert(list.get(input, 2) == ",");
assert(list.get(input, 3) == "[");
assert(tokens.join(input) == "load rax, [rbx+4]");

// 写出词法单元数量和规范形式文本。
emit.u8(len(input));
emit.bytes(tokens.join(input));
```

示例会写出：

```
08 6c 6f 61 64 20 72 61 78 2c 20 5b 72 62 78 2b 34 5d
```

带引号的文本会保留为一个词法单元，其中包括引号字符。没有结束引号的词法单元无效。

以下双字符运算符会分别形成一个词法单元：

```
== != <= >= && || << >> -> => ::
```

以下单字符也会分别形成一个词法单元：

```
, ( ) [ ] { } < > : = + - * / % & | ^ ~ . ! ? ;
```

其他相邻的非空白字符会留在同一个词法单元中，直到遇到引号或能够识别的运算符。

规范形式呈现

`tokens.join` 会生成规范形式的源代码文本。它会在普通相邻词法单元之间插入空格，并删除括号、标点和紧密运算符周围不需要的空格。

规范形式呈现不会逐字节还原原始字符串：

```
// 原始文本中的多余空格不会保留在规范形式结果中。
const input: list = tokens.of("left  && middle ||  right")
assert(tokens.join(input) == "left&&middle||right");
```

如果空白的精确形式本身有意义，应保留原始字符串。

模式片段

词法单元模式由若干使用空白分隔的片段组成。每个片段必须是精确字面量或命名捕获。

片段	含义
<code>=token</code>	精确匹配一个词法单元
<code>name:token</code>	捕获任意一个词法单元，并返回字符串
<code>name:name</code>	捕获一个名称形式的词法单元，并返回字符串
<code>name:int</code>	捕获一个整数词法单元，并返回整数
<code>name:quoted</code>	捕获一个带引号的词法单元，并返回去除引号后的字符串
<code>name:tokens</code>	捕获一段保持括号平衡的词法单元，并返回列表

开头的 `=` 是字面量标记。例如：

```
=load    匹配 load 词法单元
=,       匹配逗号词法单元
===      匹配 == 词法单元
```

捕获名称使用标识符语法，并且在同一个模式中不能重复。

匹配结果

`match.tokens` 返回包含两个条目的映射：

键	值
<code>"ok"</code>	表示整个输入是否匹配的布尔值
<code>"captures"</code>	从捕获名称到捕获值的映射

只有完整模式和完整输入都被使用后，匹配才算成功。

```
// 匹配 load 形式，并分别捕获目标名称和完整地址表达式。
const result: map = match.tokens(
    "=load destination:name =, address:tokens",
    "load rax, [rbx+(rcx*4)]"
)

assert(map.get(result, "ok"));

// 匹配成功后，从 captures 映射中读取命名捕获。
const captures: map = map.get(result, "captures")
const destination: string = map.get(captures, "destination")
const address: list = map.get(captures, "address")

assert(destination == "rax");
assert(tokens.join(address) == "[rbx+(rcx*4)]");

// 写出捕获到的目标名称和规范形式地址文本。
emit.bytes(destination);
emit.bytes(tokens.join(address));
```

示例会写出：

```
72 61 78 5b 72 62 78 2b 28 72 63 78 2a 34 29 5d
```

读取命名捕获之前必须先检查 `"ok"`。匹配失败时，结果中的 `"ok"` 为 `false`，捕获映射为空。

捕获值

`token` 和 `name` 捕获都返回字符串。`token` 接受任意一个词法单元；`name` 要求标识符以 ASCII 字母或下划线开头，后续字符只能是 ASCII 字母、数字或下划线。

`int` 会使用通常的数值前缀，把一个词法单元解析为无符号整数：

```
// 捕获目标名称和带十六进制前缀的整数。
const result: map = match.tokens(
  "set target:name =, value:int",
  "set count, 0x2a"
)
const captures: map = map.get(result, "captures")

// 整数捕获会返回数值，而不是原始词法单元字符串。
assert(map.get(result, "ok"));
assert(map.get(captures, "target") == "count");
assert(map.get(captures, "value") == 42);
```

`quoted` 接受单引号或双引号，去除两端引号，并处理 `\n`、`\r`、`\t`、转义引号和转义反斜杠。其他转义形式会保留反斜杠后的字符。

`tokens` 返回由词法单元字符串组成的列表，可以捕获零个或多个词法单元。捕获范围内的圆括号、方括号和花括号必须保持平衡。

从最短范围开始匹配与回溯

`tokens` 捕获最初只使用最短的平衡范围。如果后续模式片段无法匹配，它会扩大捕获范围并重新尝试：

```
// prefix 先尝试空范围；整数捕获失败后，它会扩大到包含 name。
const result: map = match.tokens(
  "prefix:tokens value:int",
  "name 42"
)
const captures: map = map.get(result, "captures")

// 回溯后，prefix 捕获名称词法单元，value 捕获整数 42。
assert(map.get(result, "ok"));
assert(tokens.join(map.get(captures, "prefix")) == "name");
assert(map.get(captures, "value") == 42);
```

比较运算符 `<` 和 `>` 只是普通词法单元，不是分组边界。只有 `()`、`[]` 和 `{}` 参与平衡检查。

模式不提供表示多个选择的运算符。如果输入可能具有多种有效形式，应在普通 `if / else` 流程控制中依次尝试完整模式。

普通匹配失败与无效模式

以下情况属于普通匹配失败，返回 `"ok" == false`：

- 精确字面量不同；
- 带类型的捕获遇到不符合要求的词法单元；
- `tokens` 捕获无法形成平衡范围；
- 模式先于输入结束；
- 输入先于模式结束。

以下情况会把调用作为无效表达式拒绝：

- 模式片段既不是字面量，也不是捕获；
- 字面量标记后没有词法单元；
- 捕获名称无效或重复；
- 捕获种类未知；
- 输入中的带引号词法单元没有结束引号；
- 参数不是字符串或字符串列表；
- `tokens.join` 收到包含非字符串值的列表；
- 实参数量不正确。

限制

词法单元匹配使用固定限制，使编译期解析的工作量保持可控：

限制项	最大值
模式片段	64
输入词法单元	256
匹配尝试次数	4096
嵌套括号深度	64

超过限制时，匹配调用会被拒绝，而不是作为普通匹配失败返回。