

XIRASM 中文语言指南

自然 ISA 文本与现代编译期语言

版本 0.2.16

生成日期 2026-07-19

目录

XIRASM 语言指南	7
指南结构	7
第一部分：语言基础	7
第二部分：汇编器模型	7
第三部分：构建程序	8
第 1 章：认识 XIRASM	9
指令和编译期代码写在一起	9
运行一个例子	9
编译期代码也能写出字节	9
先记住这些语法规则	10
读源码时先分清这几类行	10
第 2 章：值与绑定	12
值只存在于汇编期间	12
常量	12
可变绑定	12
类型注解和类型推断	13
字符串和字节序列	13
块作用域	14
选择 const 还是 let	14
第 3 章：表达式	15
表达式在汇编期间求值	15
算术运算	15
位运算和移位	15
比较和相等判断	16
布尔逻辑和短路求值	16
表达式中的函数调用	17
字段访问	17
运算符优先级	17
表达式错误	18
第 4 章：控制流	19
控制流只影响汇编过程	19
`if` 和 `else`	19
选择要生成的指令	20
目标条件	20
遍历范围	20
遍历列表	21

while	21
break 和 continue	21
选择控制流形式	21
第 5 章：函数与作用域	23
函数复用汇编期间的工作	23
过程函数	23
参数和实参	23
返回值函数	24
返回规则	24
函数局部作用域	25
声明顺序	25
递归和调用深度	26
选哪种函数	26
第 6 章：集合与文本	27
文本、字节、列表和映射	27
字符串保存文本	27
分割和连接文本	27
用字节序列表示二进制值	28
字节序列操作返回新值	28
列表按顺序保存值	28
更新 `let` 绑定的集合	29
列表可以保存任意编译期值	29
映射保存命名值	29
遍历映射内容	30
在函数中组合集合	30
选哪种集合	30
第 7 章：词法单元与模式匹配	32
词法单元保留源码结构	32
匹配词法单元结构	32
字面量模式词法单元	33
捕获种类	33
平衡的词法单元范围	34
最小匹配与回溯	34
空词法单元范围	34
匹配已有词法单元列表	34
尝试不同结构	35
匹配失败与无效模式	35

字符串还是词法单元	36
第 8 章：目标、指令与标号	37
当前目标	37
选择 x86、RISC-V 和 SPIR-V	37
查询目标	38
直接书写指令	38
在指令中使用编译期值	39
静态标号	39
符号引用与地址回填	40
动态指令文本与动态标号	40
读取标号地址	40
实用规则	41
第 9 章：数据与二进制布局	42
输出内容是有序字节序列	42
浮点值	42
字符串与字节序列	43
预留空间	43
填充与对齐	44
声明二进制结构体	44
字段默认值与结构体字面量	45
测量布局	45
在 x86-64 栈上使用结构体布局	46
打包并写出结构体值	46
联合体与当前字段	47
选择布局方式	48
第 10 章：模块与文件	49
导入源模块	49
内联包含源文件	49
源文件路径解析	50
选择 `import` 还是 `include`	50
检查与读取数据文件	50
读取字节范围	51
加载 JSON	51
加载 TOML	52
区分配置与二进制数据	52
实用的模块组织方式	52
第 11 章：输出区域与虚拟数据	54

先分清 RVA 和 FOA	54
`origin` 只改逻辑地址	54
`region.begin` 显式指定 RVA 和 FOA	55
`reserve` 推进逻辑大小, 但尾部可以不进文件	55
`output.section` 裁掉尾部 reserve	56
`output.org` 保留尾部 reserve	56
`region.file_align` 只对齐 raw size	57
虚拟区域是临时输出, 不自动进文件	57
省略虚拟 origin: 从当前位置的逻辑地址开始	58
最终区域信息只能在收尾阶段查询	58
怎么选	58
第 12 章: 收尾处理	60
`defer`: 回填已经存在的字段	60
在 `defer` 中读取和修改最终字节	60
计算校验和	61
查询最终区域信息	61
用函数登记回填逻辑	62
`defer` 的允许范围	62
多个 `defer` 的执行顺序	63
`late_layout`: 封存前新增真实布局	64
`late_layout` 会影响尾部 reserve	64
`late_layout` 的限制	65
选择阶段	65
第三部分: 构建程序	66
第 13 章: Flat Binary 与自定义文件格式	67
原始机器码	67
文件头加 Payload	67
用 `packed struct` 描述记录	67
用 `defer` 回填文件头	68
明确字段要的是 RVA 还是 FOA	68
表示文件间隙和 file-free 尾部	69
`late_layout`: 晚生成但仍参与布局	69
用断言保护自定义格式	70
推荐组织顺序	71
什么时候用格式接口	71
第 14 章: 可执行文件与目标文件格式	72
先声明映像再写内容	72

完整示例：ELF64 程序	72
section 和 segment 怎么选	73
格式族	73
入口点绑定	73
生成的格式内容	74
从 `format.inc` 开始	74
后续阅读	74
第 15 章：诊断与实用惯例	76
阅读诊断信息	76
信息与警告	76
主动报错	76
断言	76
保护目标相关源码	77
用名字替代魔数	78
在名称中写清坐标	78
推导布局事实	78
保持收尾处理聚焦	78
按职责分模块	79
先用 `format.inc`	79
在输出边界检查	79
找到合适的文档	79
最终检查清单	80

XIRASM 语言指南

XIRASM 是一款面向 x86、RISC-V 和 SPIR-V 的汇编器。处理器指令按熟悉的汇编语法直接写；需要在汇编期间计算、选择、循环生成或组织文件格式时，再使用 XIRASM 的编译期语言。

本指南按学习顺序讲：先看一份最小源文件，再学值、表达式、控制流、函数和集合；之后进入标号、数据布局、输出区域与收尾处理；最后说明 flat binary、自定义格式以及 PE/COFF/ELF 文件从哪里开始。查函数签名时看语言 API 参考；照着写 PE、COFF 或 ELF 文件时看格式教程。

指南结构

第一部分：语言基础

- 认识 XIRASM
- 汇编第一个 x86-64 程序。
- 理解指令行与编译期代码如何配合。
- 看懂最小源文件的语法规则。
- 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。
- 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。
- 控制流
- 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。
- 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。
- 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。
- 词法单元与模式匹配
- 检查词法单元序列。
- 匹配类似源码的文本，用来写小型 DSL 或生成辅助工具。

第二部分：汇编器模型

- 目标、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。
- 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。
- 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。
- 输出区域与虚拟数据
- 区分逻辑地址 / RVA、raw 文件偏移 / FOA 和最终进入文件的字节。
- 构建输出区域，区分 raw 文件间隙和只增加逻辑大小的尾部 reserve。
- 使用虚拟输出作为临时组装、测量和转换区域。
- 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

- Flat Binary 与自定义格式
- 构建紧凑的 flat binary 和项目专用文件格式。
- 用标号、区域、`defer` 和 `late_layout` 推导文件头字段。
- 可执行文件与目标文件格式
- 理解 `format.inc` 会替你生成哪些格式结构。
- 使用 `let` 和 `_mut` 函数构建最小可执行文件。
- 通过格式教程继续学习 PE、COFF、ELF、导入、导出、BSS（未初始化数据区）和重定位。
- 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。
- 组织可复用的汇编项目。
- 在掌握本指南后继续查阅接口参考和格式教程。

第 1 章：认识 XIRASM

指令和编译期代码写在一起

XIRASM 源文件里通常有两类内容：

- 处理器指令：x86、RISC-V 或 SPIR-V 指令，按对应汇编语法直接写。
- 编译期代码：汇编时执行，用来计算值、选择分支、循环生成内容，或调用 XIRASM 的输出接口。

两类内容可以直接混在同一个源文件里。编译期常量可以放进指令操作数，标号可以交给布局和格式接口使用，函数也可以写出数据或指令。

例子：

```
// 开启 64 位 x86 模式。
x86.use64();

// 汇编时计算常量，后面的指令直接使用它。
const answer: u32 = 40 + 2

entry:
    // 把计算好的数值放进 eax，然后返回。
    mov eax, answer
    ret
```

简单说明：

- `x86.use64();` 选择 64 位 x86 指令编码模式。
- `const answer: u32 = 40 + 2` 声明编译期常量。
- `entry:` 在当前逻辑地址定义一个标号。
- `mov eax, answer` 是 x86 指令；汇编时会把 `answer` 算成常量值。
- `ret` 也是 x86 指令。

`x86.use64();` 是编译期 API 调用，末尾要写分号。`mov eax, answer` 是处理器指令，按汇编语法直接写，不需要包装成函数调用。

运行一个例子

把源码保存为 `hello.xir`，然后执行：

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

输出字节：

```
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`。标号不产生字节，只记录一个地址，供其他指令和编译期 API 使用。

直接汇编时，XIRASM 默认写出 flat binary，也就是源码实际生成的字节序列；它不带操作系统文件头。需要 PE、COFF 或 ELF 这类标准文件时，后面章节会说明从 `format.inc` 开始，完整模板见格式教程。

编译期代码也能写出字节

编译期代码在汇编过程中执行，不是在最终程序里运行。

比如循环写四个字节：

```
// 编译期循环四次，每次写一个字节。
const count: u8 = 4
```

```
for value in range(0, count) {
    db(value);
}
```

生成的字节：

```
00 01 02 03
```

`range(0, count)` 产生从 0 到 `count-1` 的整数序列。`for` 在汇编期间循环，每次调用 `db(value);` 写出一个字节。

这个区别很关键：

做什么	怎么写
手写处理器指令	直接写汇编指令行
计算和判断	编译期表达式
重复或条件生成	编译期控制流
输出字节	<code>db</code> 、 <code>dw</code> 、 <code>dd</code> 、 <code>dq</code> 等数据调用
复用一段生成逻辑	函数
生成 PE/COFF/ELF	对应的格式接口

先记住这些语法规则

几条简单规则：

- `//` 到行尾是单行注释，也可以写在处理器指令后；指令字符串操作数中的 `//` 仍是普通文本。
- 处理器指令行不以分号结尾。
- 标号以 `:` 结尾。
- 函数和接口调用以 `;` 结尾。
- `return` 语句用 `;` 结尾。
- `const` 和 `let` 声明末尾不加分号。
- 代码块用 `{ }`。
- 函数调用和复合值可以跨行书写，只要括号或花括号最终闭合。

完整例子：

```
// 定义一个汇编时可调用的辅助函数。
fn emit_marker(value: u8) {
    // 写固定标记，后面跟传入的值。
    db(0x7f, value);
}

const marker: u8 = 3
emit_marker(marker);

start:
// 指令里可以直接使用编译期常量；指令行末尾不写分号。
mov eax, marker
ret
```

函数定义和函数调用属于编译期代码；标号和下面两行缩进的内容是 x86 指令。两类内容共同生成同一个最终输出。

读源码时先分清这几类行

读源码时，按这几步判断每行：

- 如果是 x86、RISC-V 或 SPIR-V 指令，就按对应汇编语法直接写。

- 如果要在汇编期间计算、选择或重复生成内容，就用编译期语言。
- 如果要写出字节或预留空间，就使用数据和输出 API。
- 如果要生成完整文件格式，就用 `format.inc`，不要手算文件头和表项。

下一章开始讲编译期语言的基本值、常量、可变绑定和作用域。

[返回语言指南](#)

第 2 章：值与绑定

值只存在于汇编期间

XIRASM 的值都在汇编期间使用。它们可以参与计算、决定要写出什么内容，但“把值绑定到名字”这件事本身不会写字节，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
```

这行只是给值取名为 **magic**，不会把 **4d 5a** 写进输出文件。要写出这个值，需要调用数据写出接口：

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

同样的规则适用于可变绑定、字符串、字节序列、集合和函数返回值。只有指令、数据调用、布局操作或格式接口实际使用这些值时，它们才会影响输出。

常量

常量用 **const** 声明。声明后，这个名字不能再指向别的值：

```
// 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
```

一般形式是：

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

如果 XIRASM 能从初始值推断类型，类型注解可以省略。当宽度或值类型本身就是二进制约定的一部分时，应显式写出类型。

常量声明末尾不加分号。给常量重新赋值会报错：

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

除非这段源码确实需要在汇编期间更新这个名字，否则优先用 **const**。

可变绑定

汇编期间需要逐步更新一个值时，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);
```

输出：

```
14 00 00 00
```

声明和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。

形式是：

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

和常量一样，赋值末尾不加分号。作为语句的调用仍然需要分号：

```
// 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);
```

类型注解和类型推断

绑定可以显式写出类型：

```
// 明确写出类型，让字段宽度和文本用途直接出现在源码里。
const signature: u16 = 0x5a4d
const title: string = "XIRASM"
const marker: bytes = b"OK"
const enabled: bool = true
```

也可以让 XIRASM 从初始值推断类型：

```
// 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"
```

常用的值类型：

类型	用途	示例
integer	通用编译期整数	42
u8	8 位无符号值	0xff
u16	16 位无符号值	0x5a4d
u32	32 位无符号值	0x401000
u64	64 位无符号值	0x140000000
i8	8 位有符号值	-1
i16	16 位有符号值	-200
i32	32 位有符号值	-4096
i64	64 位有符号值	-0x100000000
f32	IEEE-754 32 位浮点数	f32(1.5)
f64	IEEE-754 64 位浮点数	1.5
bool	编译期条件	true
string	编译期文本	"kernel"
bytes	确切的字节序列	b"PE"

列表、映射、结构体、联合体和类型值在后面的章节介绍。

固定位宽整数适合表示二进制字段和函数参数。如果函数只关心“这是整数”，不关心具体宽度，可以使用 **integer**。最终写出几个字节，由调用的输出接口决定。

固定位宽有符号整数使用补码表示。创建绑定或结构字段时会检查取值范围；打包有符号字段时，会按字段宽度写出低 8、16、32 或 64 位。

带小数部分或指数的十进制字面量类型为 **f64**。用 **f32(value)** 显式窄化为 **f32**，用 **f64(value)** 把 **f32** 扩展为 **f64**。整数和浮点值之间不会隐式转换。

字符串和字节序列

字符串和字节序列是不同类型的值：

```
// 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、动态生成的指令文本，以及所有要求文本参数的接口，都用字符串。需要精确字节序列时，用 **bytes**。

有些输出接口两种类型都接受：

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

输出四个字节：

```
41 42 43 44
```

其他接口可能只接受其中一种。提前在绑定里区分清楚，源码会更容易检查。

块作用域

绑定属于它声明时所在的作用域。块会创建一个嵌套作用域：

```
// 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
    // 离开代码块后，再次使用外层的 value。
    db(value);
}

db(value);
```

输出：

```
02 01
```

内部的 **value** 只在块内遮蔽外部的常量。块结束后，外部的 **value** 重新可见。

函数参数和局部绑定也属于函数作用域。嵌套块可以引入临时名字，不影响外部绑定。

当一小段代码确实需要临时名字时，遮蔽可以让代码更短。但如果内外两个名字的含义容易混淆，就不要重复使用同一个名字。

选择 const 还是 let

用最贴近意图的绑定：

情况	优先选
固定选项、大小、名称或计算结果	const
语言自身管理的循环计数器	循环绑定
累计的偏移量、校验和或累加器	let
只在小范围代码块里变化的值	块内 let
宽度属于文件格式约定的值	显式类型注解
类型显而易见的临时值	类型推断

描述格式、常量表和生成规则时，大多数名字都应是 **const**。只有短小、局部、确实需要逐步更新的计算，才使用 **let**。

下一章讲产生这些值的表达式：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

表达式在汇编期间求值

表达式在汇编期间计算。声明、赋值、函数参数、`return` 语句、条件、断言、指令操作数和数据写出调用，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);
```

这会写出 32 位值 `24`。乘法在汇编期间完成，最终输出里只有计算结果。

算术运算

整数表达式支持：

运算符	含义
<code>+</code>	加法
<code>-</code>	减法
<code>*</code>	乘法
<code>/</code>	整数除法
<code>%</code>	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);
```

输出值分别是 `14`、`20`、`3`、`2`。

加、减、乘会检查溢出。结果超出支持的整数范围时报错，不会静默截断。除数为零会报错。

一元 `+` 不改变值。一元 `-` 产生 64 位补码值：

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

输出八个 `ff` 字节。

位运算和移位

位运算常用于权限、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>	<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);

```

输出字节是 **05**。

掩码测试用括号包起来，这样意图更清楚，也不依赖位运算和比较运算的优先级。

比较和相等判断

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>	小于等于
<code>></code>	大于
<code>>=</code>	大于等于

大小比较用于整数：

```

// 数据必须大于零，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);

```

相等和不相等也可以用于兼容的非整数类型：

```

// 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");

```

不相关的值类型不能比较；XIRASM 不会为了比较而隐式转换类型。

布尔逻辑和短路求值

布尔逻辑支持：

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

`&&` 和 `||` 会短路：

- `left && right` 当 `left` 为 `false` 时不计算 `right`。
- `left || right` 当 `left` 为 `true` 时不计算 `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);
```

除法不会被执行，因为 **enabled** 是 **true**。

当前一个条件已经能决定结果时，短路求值可以避免计算后面的表达式。这常用于先检查边界，再读取或计算依赖该边界的值。

表达式中的函数调用

返回值函数和内置函数可以出现在任何能接受其结果类型的位置：

```
// 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);
```

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");
```

只写出内容或执行汇编动作的过程函数，要按语句调用；它们不会产生表达式值。返回值函数在第 5 章介绍。

字段访问

有命名字段的值用 `.` 选择一个字段：

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

结构体和联合体值在第 9 章介绍。**target.bits** 和 **target.isa** 这类目标查询通常直接写在 **if** 条件里，不必先复制到普通绑定中。

运算符优先级

表中同一行的运算符优先级相同。越靠上的行绑定越紧密：

优先级	运算符		
最高	函数调用、括号表达式、字段访问		
一元	+ 、 - 、 ~ 、 !		
乘法	* 、 / 、 %		
移位	<< 、 >>		
加法	+ 、 -		
按位与	&		
按位异或	^		
按位或	\	,	
大小比较	< 、 <= 、 > 、 >=		
相等判断	== 、 !=		
逻辑与	&&		
最低	\	\	,

同优先级从左到右计算。

不要套用其他语言的优先级习惯。XIRASM 中，移位比加减法绑定更紧：

```
// 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);
```

第一个值是 **17**，因为计算方式是 $1 + (2 \ll 3)$ 。第二个值是 **24**。

表达式同时包含移位、算术、掩码或比较时，尽量加括号。括号能说明意图，也能减少后续修改时的误解。

表达式错误

表达式算不出明确的编译期值时，XIRASM 会报错。常见原因：

- 名字未定义
- 操作数的值类型不合适
- 除数为零
- 整数溢出或下溢
- 字段不存在
- 函数参数无效

这些错误会终止汇编。XIRASM 不会用零代替，也不会忽略错误；否则指令操作数或二进制布局可能被悄悄破坏。

下一章讲 **if**、**while** 和 **for** 如何根据表达式选择源码、重复执行源码。

返回语言指南

第 4 章：控制流

控制流只影响汇编过程

XIRASM 的 **if**、**for**、**while** 在汇编期间执行。它们决定哪些源码参与本次输出、哪些源码重复执行；不会自动变成最终程序里的运行时分支或循环。

```
// 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");
}
```

只有选中的分支会写出字节。**debug_build** 为 **false** 时，输出包含 **RELEASE**；生成文件里没有运行时条件判断。

编译期控制流用来：

- 按目标选择源码
- 重复生成数据或指令
- 遍历编译期集合
- 计算表格和偏移量
- 包含可选的文件格式记录
- 验证源码配置

运行时控制流仍然要写处理器指令，比如 x86 的 **jmp**、**call** 和条件分支。

if 和 else

if 条件必须产生布尔值：

```
// 根据数据大小选择一个单字节标记。
const payload_size = 32

if payload_size <= 64 {
    db(0x01);
} else {
    db(0xff);
}
```

选中的块在自己的作用域中执行。另一个块不会写出内容，也不会执行其中的编译期操作。

else 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");
}
```

需要多于两个分支时用 **else if**：

```
// 通过链式条件把数据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);
}
```

输出 02。

选择要生成的指令

if 块里可以直接写处理器指令：

```
// 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
```

因为 `return_zero` 为 true，生成的指令是：

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

没选中的 `mov` 指令不会进入输出。这是汇编期间选择指令，不是运行时条件分支。

目标条件

if 条件里可以直接用目标查询：

```
// 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` 标识选择的 ISA 系列，`target.bits` 报告当前位宽。`x86.use32()`、`x86.use64()`、`riscv.use32()`、`riscv.use64()` 等模式调用会更新后续目标条件看到的位宽。

目标查询专门用于这种条件判断。直接写在 if 条件里即可，不必先复制到另一个绑定。

遍历范围

已知迭代次数时，用 `for` 配合 `range(start, end)`：

```
// 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);
}
```

输出：

```
00 01 02 03
```

起始值包含，结束值不包含。范围必须递增或为空，递减范围如 `range(4, 0)` 会报错。

循环绑定属于循环体：

```
// 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);
}
```

输出 **10 11 12**。**index** 和 **encoded** 都是每次循环的局部变量。

遍历列表

for 循环可以遍历编译期列表的值：

```
// 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);
}
```

按列表顺序遍历。适用于字节表、导入描述、生成的名字等集合数据。

映射不能直接遍历。用 **map.keys(...)** 或 **map.values(...)** 取得列表后再遍历。集合在第 6 章介绍。

while

当循环何时结束取决于循环体里的更新时，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
}
```

输出：

```
01 02 03 04
```

每次迭代前判断条件。初始为 **false** 则不执行循环体。

终止条件应靠近更新逻辑，并且一眼能看明白。编译期循环如果停不下来，汇编就无法完成；XIRASM 最多允许 1,000,000 次迭代，超限会报错。

break 和 continue

break 结束当前最内层的编译期循环。**continue** 跳过本次迭代剩下的语句，直接开始下一次迭代。两者可以用在 **for**、**while** 中，也可以用在收尾处理里的 **while** 中；循环体里嵌套 **if** 时同样可用。

例如：

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

输出 **00 02 04**。循环控制语句不能跨函数调用边界，在循环外使用会报错。

选择控制流形式

需求	使用
两个源码块二选一	if / else

需求	使用
多个源码块选一个	<code>if / else if / else</code>
按指令集或位宽选源码	目标条件
重复固定次数	<code>for + range</code>
遍历编译期集合	<code>for + 列表</code>
重复到条件满足为止	<code>while</code>
选择生成的指令	<code>if</code> 块中直接写指令
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>

迭代次数已知时优先用 `for`，比 `while` 更容易看出输出规律，也不需要手动维护循环变量。

下一章把重复的计算和输出操作打包成函数，带参数、返回值和局部作用域。

返回语言指南

第 5 章：函数与作用域

函数复用汇编期间的工作

函数用来复用汇编期间的逻辑。它可以计算一个值，也可以写出数据、指令或组合多个输出操作。

XIRASM 有两种函数：

- 过程函数执行操作，不返回表达式值。
- 返回值函数计算并返回一个值，用 `->` 声明返回类型。

两种函数都用 `fn`、参数列表和代码块声明。函数在汇编期间执行；声明函数本身不写字节。只有调用过程函数，或把返回值交给输出接口时，输出才会变化。

一个函数最好只做一类事：要么执行输出或布局操作，要么返回一个可用于表达式的值。

过程函数

过程函数适合组合一组输出操作：

```
// 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);
```

输出：

02 03 08 09

没有 `->` 返回类型的函数就是过程函数。过程调用是一条语句，末尾要加分号。

过程函数体可以包含声明、赋值、控制流、数据输出以及其他过程调用。重复的二进制记录、指令序列、表项或格式构建步骤，都适合写成过程函数。

过程函数本身不是值，不能用于初始化绑定：

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

const marker = emit_marker()
```

这会报错，因为 `emit_marker()` 只执行操作，不返回表达式值。

参数和实参

参数写在括号里：

```
// 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);
```

输出四个 `cc` 字节。`index` 控制循环次数，即使循环体里未使用它。

参数类型可以省略，但公开辅助函数最好写清楚：

```
// 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));
```

输出 **2a**。实参值按位置绑定到对应参数。公开的辅助函数建议加类型注解，让调用约定更明确，也能更早发现不适的实参。

实参按位置对应参数。调用时必须为每个参数提供一个实参；没有默认实参机制。同一函数的参数名不能重复。每次调用有独立的参数绑定，互不影响。

返回值函数

函数需要返回表达式值时，加上 **-> 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**，输出：

```
80 00
```

return 末尾加分号。返回表达式必须符合声明的返回类型。返回类型可以是整数、布尔值、字符串、字节序列等：

```
// 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());
```

返回值函数可以放在任何接受其返回类型的位置：声明、实参、条件、函数调用，或更大的表达式里。

返回规则

返回值函数必须执行到 **return**。末尾无返回值会报错：

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

const result = incomplete(4)
```

返回值类型必须匹配：

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

const result = enabled()
```

报错，整数不满足 **bool** 约定。

返回值函数只用于计算，不能写出数据，也不能改变布局：

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

const result = bad_counter()
```

需要改变输出或布局时用过程函数；需要计算值时用返回值函数。

过程函数不声明返回类型，也不能返回值：

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

emit_one();
```

过程函数执行到末尾即结束。

函数局部作用域

参数和函数内的绑定只属于当前调用：

```
// 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));
```

输出 **10 18**。每次调用创建新的 **size**、**overhead**、**result**，调用结束后这些名字不再可用。

函数内的块创建嵌套作用域，可遮蔽外部名字：

```
// 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));
```

输出 **08**。嵌套块中 **value** 指向局部常量 **5**。块结束后参数 **value** 重新可见。

中间计算放在局部名字里，临时状态不会泄漏到函数外，多次调用也互不干扰。

声明顺序

函数必须在调用前声明：

```
// 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);
```

把调用移至声明前会报错，此时函数名尚未定义。

函数声明只能出现在顶层，不能写在另一个函数、循环、条件块或代码块内。相关函数可以在顶层相邻排列，供后续源码调用。

第 10 章介绍如何通过包含文件将函数声明提供给另一个源文件。

递归和调用深度

返回值函数在有终止条件时可以递归调用自身：

```
// 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));
```

输出 **0a**，即 **4 + 3 + 2 + 1** 的结果。

递归在汇编期间执行。递归深度应控制在较小范围内，终止条件必须明确。XIRASM 会拒绝超过 128 层调用的函数链，避免失控递归。

遍历范围或集合时用循环。只有计算本身具有递归结构且终止条件明确时，才用递归。

选哪种函数

需求	使用
输出字节或指令	过程函数
组合多个输出调用	过程函数
为表达式计算值	返回值函数
复用纯计算逻辑	返回值函数
临时名字不泄漏	两种均可
遍历范围或集合	通常用循环
递归计算	递归返回值函数

每个函数尽量专注一件事。小的计算函数容易组合；小的过程函数也更容易从调用处看出会写出什么。

下一章介绍列表、映射、字符串和字节序列，供需要处理集合和文本的函数使用。

返回语言指南

第 6 章：集合与文本

文本、字节、列表和映射

XIRASM 常用四种编译期数据类型：

类型	表示什么	常见用途
<code>string</code>	编译期文本	名字、路径、错误信息、文本字段
<code>bytes</code>	确切的二进制数据	签名、魔数、已编码指令字节、打包记录
<code>list</code>	有序的值序列	重复表项、参数列表、配置条目
<code>map</code>	字符串到值的映射	配置选项、记录字段、查找表

这些值本身不会写出数据。只有把它们传给输出接口，数据才会进入输出文件。

`string` 和 `bytes` 的操作通常返回新值；`list` 和 `map` 如果需要逐步构建，放在 `let` 绑定里，然后用 `list.push_mut`、`list.set_mut`、`map.set_mut` 更新。值在传递时会复制，不会在调用者之间共享可变内存。

字符串保存文本

字符串存放编译期使用的文本：

```
// 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);
```

输出 `kernel64` 的八个文本字节。

常用字符串 API：

- `trim(text)` 去掉两端空白
- `lower(text)`、`upper(text)` 转 ASCII 字母大小写
- `starts_with`、`ends_with`、`contains` 检测文本
- `replace(text, needle, replacement)` 替换匹配文本
- `to_string(value)` 把值转成文本

大小写转换只处理 ASCII。名字、路径、错误消息、配置键适合用字符串；二进制数据用 `bytes`。

分割和连接文本

`split` 按分隔符拆成字符串列表，`join` 把列表连成文本：

```
// 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。这种写法适合处理小配置列表、格式选项中的文本，以及动态生成的名字。

用字节序列表示二进制值

bytes 表示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 **58 49 52 00 03 00**。

bytes.from_hex 把十六进制文本转二进制。**bytes.le(value, width)** 用指定字节数按小端序编码整数。**bytes.concat** 拼接字节序列。**bytes.eq** 比较两个序列。**bytes.hex** 把二进制显示为小写十六进制文本。

字节序列操作返回新值

字节序列操作不会修改原值，而是返回一个新值：

```
// 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);
```

输出 **41 2d 5a 43 ff ff**。

操作包括：

- **bytes.new()** 空序列
- **bytes.push(value, byte)** 末尾追加
- **bytes.repeat(count, byte)** 重复字节
- **bytes.insert(value, index, addition)** 在零开始索引处插入
- **bytes.replace(value, index, count, replacement)** 替换范围
- **bytes.concat(left, right)** 拼接

原始 **base** 仍然是 **41 42 43**。每次操作产生新值，旧值不受影响。

列表按顺序保存值

列表存放有序值：

```
// 先追加一个元素，再替换索引 1 处的值。
let items: list = list.of(1, 2, 3)
list.push_mut(items, 4);
list.set_mut(items, 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(items, 1, 2)
assert(list.eq(items, list.of(1, 0xaa, 3, 4)));
```

```
assert(list.eq(middle, list.of(0xaa, 3)));

for value in items { db(value); }
```

输出 **01 aa 03 04**。

`list.push_mut` 和 `list.set_mut` 直接更新 `items` 这个 `let` 绑定。`list.slice` 返回从起始索引开始的指定个元素，不修改原列表。

常用列表接口还有 `list.new()`、`list.get()`、`list.concat()`、`list.eq()` 和 `len()`。

更新 `let` 绑定的集合

一段局部算法需要逐步构建列表或映射时，使用可变语句 API：

```
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);
```

第一个参数必须是直接的 `let` 绑定名。目标不能是 `const`、临时表达式、函数返回值、字段访问、缺失名称，或类型不匹配的集合。查找名称时会使用最近的绑定，因此块内 `let` 会遮蔽外层同名绑定。顶层 `let` 可以在正常汇编流程中更新；返回值函数只能更新本次调用内部的局部 `let`。

`list.push_mut` 追加一个复制后的元素。`list.set_mut` 替换已有的零基索引。`map.set_mut` 插入或替换字符串键对应的值；替换已有键时保留原有插入位置。这些语句不会产生表达式值，也不能出现在 `defer` 或 `late_layout` 块中。

复制边界是明确的：

```
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)));
```

列表可以保存任意编译期值

列表元素不限于整数，也可以是字符串、字节序列、映射等编译期值：

```
// 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))
)

for chunk in chunks { emit.bytes(chunk); }
```

输出 **58 52 34 12**。

映射保存命名值

映射把字符串键关联到值。逐步构建映射时，直接用 `map.set_mut` 更新 `let` 绑定：

```
// 逐步加入命名属性，并用新值替换已有的 arch 属性。
let complete: map = map.new()
map.set_mut(complete, "arch", "x64");
map.set_mut(complete, "mode", "release");
map.set_mut(complete, "arch", "rv64");
// 映射中的值也可以是列表等更大的编译期值。
map.set_mut(complete, "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(complete, "arch") == "rv64");
assert(map.get_or(complete, "missing", "default") == "default");
assert(list.eq(map.get(complete, "tags"), list.of("asm", "dsl")));
```

映射键必须是字符串。添加新键会追加一个条目；替换已有键会更新值，但保留这个键原来的位置。`map.keys` 和 `map.values` 因此会返回互相对应的并行列表。

遍历映射内容

映射用于按键查找。需要遍历时先用 `map.keys` 或 `map.values` 转列表：

```
// 用映射保存两个带名称的二进制字段。
let fields: map = map.new()
map.set_mut(fields, "magic", b"XR");
map.set_mut(fields, "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);
}
```

文件格式规定了确切顺序时，用列表保存顺序。映射主要负责按名称查找。

在函数中组合集合

集合转换可以封装为函数：

```
// 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));
```

输出 `34 12 cd ab`。函数逐步构建字节序列，调用者决定何时把结果写入输出。

职责可以这样划分：字符串保存名字和文本；映射保存命名配置；列表保存有序值；字节序列保存最终二进制表示；过程函数决定什么时候写出。

选哪种集合

需求	使用
面向人的源文本	<code>string</code>
确切的二进制表示	<code>bytes</code>
有序值序列	<code>list</code>
按字符串键查找	<code>map</code>
处理按分隔符拆分的文本	<code>split</code> 转 <code>list</code>
重新组合分隔文本	<code>join</code>
逐字节构建二进制记录	<code>bytes</code> API
既要顺序又要查找	<code>list</code> + <code>map</code>

下一章讲词法单元和模式匹配，用于需要按语法解析的源文本。

[返回语言指南](#)

第 7 章：词法单元与模式匹配

本章讲如何在汇编期间检查一段“像源码一样的文本”。普通字符串 API 适合处理名称、路径和简单替换；如果你关心标点、运算符、字面量、括号分组这些源码结构，就先把文本拆成词法单元，再做模式匹配。

不要为了汇编普通指令而手动分词。x86、RISC-V 和 SPIR-V 指令本来就能直接写在 XIRASM 源码里。词法单元匹配主要用于小型 DSL、生成的源码片段、命令记录和可复用的编译期辅助函数。

词法单元保留源码结构

字符串接口把文本看成字符序列。处理名称、路径、分隔字段和简单替换时，这已经够用。要处理像源码一样的文本时，通常需要保留更细的结构。

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有结构意义。如果只按空格拆分，不仅会丢掉这些结构，还会让只差空格的等价写法得到不同结果。

XIRASM 可以把这类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，同时保留词法单元顺序。`tokens.join` 会用规范间距重新拼回文本，但不会逐字节还原原始字符串。

如果字符排列本身重要，用字符串。如果名称、标点、运算符、字面量和括号分组重要，用词法单元。

匹配词法单元结构

`match.tokens(pattern, input)` 会把输入和词法单元模式比较：

```
// 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 map：

- `"ok"` 是布尔值，表示完整输入是否匹配成功；
- `"captures"` 是一个 map，保存各个命名捕获提取出的值。

先确认 **"ok"** 为 true，再读取捕获结果。

字面量模式词法单元

模式片段以 **=** 开头时，表示精确匹配一个词法单元：

```
// `=&&` 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：

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

第一个 **=** 是模式标记；**===** 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。

模式片段之间用空白分隔。模式里的空白只用于可读性；真正匹配的是词法单元，不是输入原来的空格数量。

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

同一个模式中的捕获名称必须唯一。

单个词法单元捕获适合描述小型命令语法：

```
// 从 set 形式中提取目标名称，并把整数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)]");
```

捕获结果是词法单元列表，不是字符串。继续保留词法单元形式，后续匹配器就能直接检查这段内容，不必再次分词。

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**。

带类型的捕获遇到不合适的词法单元时，只是匹配失败。只要模式本身有效，就不会产生汇编错误：

```
// name 不是整数数字量，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"));
```

这让你可以按顺序尝试多个有效模式。

空词法单元范围

tokens 捕获可以为空：

```
// call 后没有参数，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

模式本身没有“或者”运算符。需要支持多种形式时，用编译期控制流依次尝试：

```
// 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形式，`if/else` 负责判断当前输入属于哪一种。

语法更大时，可以把每种形式放进一个小的返回值函数，让调用处清楚显示尝试顺序。

匹配失败与无效模式

输入结构或捕获类型不匹配时，结果是 `"ok" == false`。比如：

- 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。

模式本身写错时则不同。XIRASM 会把它当成源码错误拒绝，而不是返回“匹配失败”。未知捕获种类、空字面量或重复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"`，这个模式无效。

请记住两者的区别：

- 模式有效，但输入不匹配，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

不要为了汇编普通指令而把指令行转换成词法单元。普通指令直接写；词法单元匹配留给用户自定义 DSL、生成的源码片段、小型命令记录，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。下一章介绍目标选择、指令、标号和地址引用。

[返回语言指南](#)

第 8 章：目标、指令与标号

编译期语言决定生成什么；汇编器模型决定生成的指令和数据放在哪里、由哪个指令集编码，以及符号引用如何变成具体值。

本章先讲三件事：

- 当前目标决定使用哪套指令集和位宽；
- 指令行会按当前目标编码；
- 标号给地址位置取名字，供指令和数据引用。

手写汇编时，直接选择目标、写指令、定义标号。只有确实需要在汇编期间拼出指令或标号名时，才使用 `isa(...)` 和 `label.define(...)`。

当前目标

每条指令都会按当前目标编码。目标包含指令集系列，以及该指令集需要的位宽等信息。

命令行选项设置初始目标：

```
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
xirasm module.spvasm -o module.spv --target spv
```

没有显式选择时，XIRASM 默认使用 64 位 x86。源码仍然可以写明指令模式；示例、可复用 `include` 和依赖特定位宽的代码都建议显式声明：

```
// 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，不会改写前面已经写出的指令。

选择 x86、RISC-V 和 SPIR-V

源文件内用这些接口切换模式：

接口	后续指令按什么模式编码
<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
<code>spv.use()</code>	SPIR-V 1.6 模块

同一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
```

这三条指令分别按 16 位、32 位和 64 位 x86 编码。模式调用不会回头修改前面的指令。

选择编码模式不等于让生成的程序在运行时切换处理器模式。引导映像、内核、固件组件或混合模式程序仍然需要自己安排运行时模式转换。RISC-V 宽度选择也遵循同样的源码顺序规则：

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

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

不要把 x86 的模式概念套到 RISC-V 或 SPIR-V 上。XIRASM 会分别保存各 ISA 的目标设置，而不是把所有 ISA 都压成一个通用的 `mode_bits` 值。

SPIR-V 不是逐条独立编码的机器指令流，而是一个完整的逻辑模块。用 `spv.use()` 选择 SPIR-V 1.6，然后直接书写标准 `Op*` 指令和数字结果 ID：

```
spv.use();

OpCapability Shader
OpMemoryModel Logical GLSL450
%1 = OpTypeVoid
```

命令行可用 `--target spv` 或 `--target spirv`，两者都选择 SPIR-V 1.6。同一个 SPIR-V 输出只能包含同一个 section、同一个模块版本的 SPIR-V 指令行，不能混入 x86/RISC-V 指令，也不能混入数据写出、预留或对齐片段。结果 ID 目前必须写成 `%1` 这类数字形式，不接受符号 ID。

查询目标

编译期控制流可以检查当前目标：

```
// 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
}
```

可查询的目标系列值：

值	目标系列
<code>.x86_64</code>	x86 (use16/use32/use64)
<code>.riscv64</code>	RISC-V (xlen 32/64)
<code>.spirv</code>	SPIR-V

这些公开的目标系列名称标识后端系列，不报告当前指令宽度。例如 `x86.use32()` 之后，`target.isa == .x86_64` 仍然成立，但 `target.bits == 32`。

需要判定位宽时用 `target.bits`；RISC-V 条件也可以使用 `target.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
}
```

直接书写指令

指令使用所选指令集的常规汇编语法：

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

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

指令行由助记符和操作数组成。空格用于提高可读性；方括号、圆括号或花括号中的逗号属于嵌套操作数的一部分，不会把指令错误拆开。

指令行末尾不写分号。结构化的编译期调用需要分号：

```
// 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
```

`x86.use64();` 和 `emit.u8(0x90);` 是编译期 API 调用，`nop` 是处理器指令。

XIRASM 会把指令内容和当前目标一起交给对应后端编码。标号、布局、输出区域和地址回填仍由前端负责。

SPIR-V 是例外：汇编器会按源码顺序收集整个模块的指令，再一次性交给后端，使模块头、ID 上界、类型上下文和扩展指令集保持一致。

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
```

标号也可以在指令中参与运算：

```
// 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
```

汇编器会以符号形式保留表达式，等地址确定后再求值。

只有确实需要动态生成指令时，才拼接指令字符串。

静态标号

标号写在名字后加冒号：

```
// 使用普通标号表达控制流。
x86.use64();

entry:
    mov eax, 1
    jmp done

done:
    ret
```

标号不产生任何字节，只把名字绑定到当前位置。

标号可以先使用后定义：

```
// 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
```

前向引用的长度选择属于指令编码约束。源码可以显式写出 **short** 或 **near**，也可以让布局和后端在约束允许时处理符号位移；不要在源码中手算跳转偏移。

符号引用与地址回填

引用标号的指令，有时要等布局确定后才能知道最终位移。

XIRASM 的处理顺序：

- 源文件定义标号并生成指令片段。
- 后端先编码指令，把依赖标号的字段暂时留下。
- 布局器给标号和片段分配最终地址。
- 地址确定后计算标号表达式，并把结果写回指令字段。

前向引用不需要手动计算地址：

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

entry:
  jmp target
  nop

target:
  ret
```

只要源文件描述指令及其标号表达式，而非手动填偏移，前面的指令改变长度也不会失效。

动态指令文本与动态标号

编译期需要计算名字时使用动态指令和标号：

```
// 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");

// 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
```

isa(text) 把指令文本交给当前目标编码，**label.define(name)** 在当前位置创建标号。命名用 **sym.join**，需要唯一名称时用 **sym.unique**。

只在确实需要时用动态标号。不要替代 **loop:**、**done:** 等静态写法。

读取标号地址

label_addr(label_or_name) 返回标号的逻辑地址：

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

指令内的标号引用通过地址计算完成，不需要手动查询。**label_addr()** 用于数据段写入标号地址这类求值场景。

布局未稳定时，在 **defer**（第 12 章）里延迟查询。需要重定位时用格式接口，不要手动填偏移。

实用规则

- 源文件开头明确选择目标模式。
- 指令按对应 ISA 的汇编语法直接写。
- 编译期调用末尾加分号，指令末尾不加。
- 源文件中地址明确时用静态标号。
- 前向引用以符号形式传递地址。
- 只在确实需要时用 **isa** 和 **label.define**。
- 判定位宽用 **target.bits**，判定系列用 **target.isa**。
- 布局未稳定时，推迟绝对地址查询到 **defer**。
- 需要可加载文件中的重定位时，使用格式接口提供的重定位声明。

下一章介绍数据声明和二进制布局。

[返回目录](#)

第 9 章：数据与二进制布局

指令不是输出字节的唯一来源。文件头、查找表、消息字面量、预留空间，都需要明确的二进制布局。

XIRASM 提供两层：

- 直接写数据的接口
- 定义结构体/联合体的类型系统

几行数据直接用写接口就够了。常用记录格式定义成结构体。

输出内容是有序字节序列

数据从当前位置开始追加：

```
// 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);
```

按小端序写：

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

值超出目标宽度会报错，不会静默截断；需要更大的字段时，应选择更宽的写出接口。

简写版本：

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

简写接受多个参数。**db** 还能接字符串和字节序列，其他简写只接整数。1、2、4、6、8 字节写出会检查范围；10、16、32、64 字节写出会从 **u64** 零扩展。需要精确位模式时用 **emit.bytes**。**dt** 只是 10 字节整数数据写出形式，不表示 x87 f80 浮点格式。

浮点值

```
const scale: f64 = 1.5
const compact: f32 = f32(scale)
emit.f32(compact);
```

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

`emit.f32/emit.f64` 写出 IEEE-754 binary32/binary64。实参类型必须精确匹配：`emit.f32` 接受 `f32`，`emit.f64` 接受 `f64`。溢出、NaN 和 Infinity 都会报错；有限下溢和有符号零会按 IEEE-754 位模式保留。

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);
```

输出：

```
41 42 43 44 00
```

字符串和字节序列原样复制。XIRASM 不自动加 NUL。要终止符自己写。

`emit.bytes(value)` 写出字符串或 `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 flat binary 输出中，这会写出：

```
eb 00 00 fe
```

预留简写把数量乘以元素宽度：

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，预留范围会在 flat binary 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 raw 文件字节。第 11 章会用 FOA 和尾部 reserve 解释这一区别。

填充与对齐

pad(count, fill) 写 count 个重复字节：

```
// 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);
```

输出：

```
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	reserve 或 rb/rw/rd/rp/rq/rt/rdq/rqq/rdqq
精确数量的填充字节	pad
到达已知输出位置	pad_to
到达下一个对齐位置	align

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
}
```

自然对齐。**tag** 从偏移 0 开始。**u32** 要 4 字节对齐，所以 **size** 从偏移 4 开始。总共 8 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。运行时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，可以用在指令、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。

在 x86-64 栈上使用结构体布局

复合类型是汇编期的布局描述。x86 没有可以把整个结构体值一次性压栈的指令。正确做法是先在栈上分配记录空间，再把 **offset_of** 直接用于普通内存操作数：

```
x86.use64();

struct StackFrame {
    tag: u8,
    value: u32,
    tail: u16,
}

assert(sizeof(StackFrame) == 12);
assert(offset_of(StackFrame, value) == 4);
assert(offset_of(StackFrame, tail) == 8);

sub rsp, sizeof(StackFrame)
mov dword [rsp + offset_of(StackFrame, value)], 0x44332211
mov eax, [rsp + offset_of(StackFrame, value)]
mov word [rsp + offset_of(StackFrame, tail)], 0x6655
add rsp, sizeof(StackFrame)
```

紧凑布局与嵌套布局使用相同写法。嵌套字段可以直接写成 **[rsp + offset_of(NestedFrame, point.y)]**。如果这段代码还要调用 Windows x64 函数，必须另外满足调用约定的栈对齐与影子空间要求；**sizeof(StackFrame)** 只描述结构体记录本身。

在 PE64 程序中使用结构体栈帧时，同样要同时满足 Windows x64 调用约定；结构体大小 只负责描述局部记录的布局，不会自动补齐调用边界需要的栈空间。

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

联合会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；
- 源代码与外部格式强绑定，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。

外部规范定义的布局用 `sizeof` 和 `offset_of` 验证。断言把格式假设变成可执行检查，后续修改也更安全。

下一章介绍模块和文件：include/import 组织源代码，编译期读取 JSON、TOML 等外部数据。

返回语言指南

第 10 章：模块与文件

汇编项目很快就会超过单个源文件的规模。指令辅助函数、二进制记录定义、生成的表格、配置数据和嵌入数据，通常由不同的人维护，变更原因也各异。

XIRASM 提供两种独立的文件加载模型：

- `include` 和 `import` 加载 XIRASM 源文件；
- `fs`、`json` 和 `toml` 函数加载 Meta 代码使用的数据。

区分这两种角色可以让项目结构更清晰：源文件提供声明或执行输出操作；数据文件被读取后变成源码可以检查、转换和写出的值。

本章只讲汇编期间读取文件，也就是 `fs.*`、`json.*` 和 `toml.*` 这些编译期接口。最终程序运行时怎样读写文件，属于操作系统 ABI、系统调用或运行时库的问题，不在本章范围内。

导入源模块

`import(path)` 对给定的源文件至多求值一次：

```
// 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);
}
```

两次导入指向同一个已解析文件，因此函数只声明一次。示例输出：

```
34 12
```

定义可复用函数、常量、结构体或其他名字的文件应使用 `import`。即使通过多条依赖路径引入同一模块，其声明和输出也不会重复。

`import` 语句必须位于源文件顶层，不可放置于可能按条件执行的块中：

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

应在顶层选择要导入的模块；如果行为取决于目标，就把目标判断写在模块内部。

内联包含源文件

`include(path)` 每执行到一次就对目标源文件重新求值：

```
// 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
```

仅在确实需要重复执行时使用 `include`。典型用途包括：

- 在当前输出位置插入生成的数据；
- 共享一段短的写出语句序列；
- 从计算得出的路径选取源文件片段；
- 多次应用同一个源文件模板。

由于包含操作重复执行，多次包含一个声明了相同函数、常量或类型的文件可能导致重名错误。这类文件通常是模块，应通过 **import** 加载。

源文件路径解析

相对路径以包含 **include** 或 **import** 语句的文件所在目录为基准进行解析。

对于以下项目：

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

tables/records.inc 可以加载相邻文件：

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

该路径相对于 **records.inc**，而非进程工作目录或入口源文件。

如果在当前源文件旁未找到相对路径对应的文件，XIRASM 会依次检查项目的 **include** 目录及其安装目录下的 **include** 目录。这使项目模块能够覆盖或补充已安装的库，而无需在源码中写入本机绝对路径。

绝对路径也可使用，但可移植的项目应优先使用相对于源文件的路径或已配置的 **include** 根目录。

源文件加载不允许形成循环。一个文件正在求值期间，不得直接或间接地再次包含或导入自身。

选择 **import** 还是 **include**

应根据执行语义选择，而非文件扩展名：

需求	使用
仅声明一次可复用名字	import
共享 Meta 函数或类型库	import
在当前输出位置执行源文件片段	include
重复执行同一源文件	include
避免依赖链中的重复声明	import

项目约定的常见做法：

- 模块定义可复用的名字和类型；
- 包含的片段执行与输出位置相关的工作；
- 入口源文件选择目标并组装最终输出。

检查与读取数据文件

fs.exists(path) 检查数据文件是否能按当前路径规则找到：

```
// 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"));
```

数据路径使用与源文件加载相同的相对路径解析模型。当这段代码移入模块时，`payload.bin` 会相对于该模块所在文件解析。

`fs.read_text(path)` 将整个文件读为 `string`：

```
const banner: string = fs.read_text("banner.txt")
assert(contains(banner, "XIRASM"));
emit.bytes(banner);
```

如果 `banner.txt` 包含：

```
XIRASM
```

按原样写出文件中的字节，不会自动追加终止零字节。

`fs.read_bytes(path)` 将整个文件读为 `bytes`。图片、编码后的表格、预构建的记录及其他二进制数据均适用。

当字节仅用于输出时，`emit.file(path)` 可避免创建中间绑定。`emit.file(path, offset, count)` 写出精确的范围。这两种形式均使用与 `fs.read_bytes` 相同的相对路径解析器和边界检查，但不能在 `late_layout` 和 `defer` 中使用。

读取字节范围

三参数形式的 `fs.read_bytes` 读取带边界的范围：

```
const middle: bytes = fs.read_bytes("payload.bin", 1, 2)
assert(len(middle) == 2);
emit.bytes(middle);
```

第二个参数是基于零的文件偏移，第三个参数是返回的字节数。

如果 `payload.bin` 包含：

```
10 20 30 40
```

示例输出：

```
20 30
```

所请求的范围必须完整存在。XIRASM 会报告越界读取错误，而非静默截断。

加载 JSON

`json.file(path)` 一次性完成 JSON 文件的读取和解析：

```
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));
```

对于以下输入：

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

输出为：

```
58 52 40 01 02
```

`json.parse(value)` 解析已包含在 `string` 或 `bytes` 值中的 JSON：

```
const raw: string = fs.read_text("config.json")
const config: map = json.parse(raw)
```

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

JSON 对象转换为 map，数组转换为 list，字符串和布尔值保持相应的 Meta 类型，非负整数转换为整数值。JSON 的 **null** 转换为 **void**。

浮点数和负整数不在当前 Meta 数据模型的范围。重复的键和格式错误的 JSON 将被拒绝。

加载 TOML

toml.file(path) 为 TOML 提供同样的便捷入口：

```
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"));
```

对于以下输入：

```
name = "XR"

[target]
bits = 64
```

输出为：

```
58 52 40
```

toml.parse(value) 从 **string** 或 **bytes** 值中解析 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 表转换为 map，数组转换为 list，字符串、布尔值和非负整数转换为相应的 Meta 值。Meta 数据转换目前不接受浮点数和时间戳值。

区分配置与二进制数据

结构化配置和原始二进制数据解决不同的问题。

使用 JSON 或 TOML 的场景：

- 字段需通过名称访问；
- 输入预期由人工编辑；
- 同一输入驱动多个生成值；
- 需进行校验和条件生成。

使用 **fs.read_bytes** 的场景：

- 文件内容已是所需的二进制表示；
- 需保持字节级的精确性；
- 仅需有限的字节范围；
- 解析不会带来有用的结构。

当文本本身就是数据，或需显式应用某个解析器时，使用 **fs.read_text**。

实用的模块组织方式

中等规模的项目可采用以下结构：

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

main.asm 导入可复用的定义：

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

这些模块可以相对于自身读取数据，或通过显式路径（如 `../data/config.toml`）读取。

应保持模块行为的可读性：

- 声明库优先使用 **import**;
- **include** 仅用于确实需要重复执行的工作;
- 文件路径相对于所属源文件书写;
- 用 **assert** 和 **fs.exists** 验证必要数据;
- 结构化文件仅解析一次，复用得到的 map 或 list;
- 大型文件仅需部分内容时，使用有界的二进制读取。

下一章介绍输出区域与虚拟数据：XIRASM 如何分离逻辑地址 / RVA、raw 文件偏移 / FOA、尾部 reserve 和临时虚拟输出。

[返回目录](#)

第 11 章：输出区域与虚拟数据

写最简单的 flat binary 时，可以把“地址”和“文件偏移”当成同一件事：第一个字节地址是 0，FOA 也是 0，后面一起增长。

但只要开始写 PE、ELF、COFF，或者手写自己的二进制格式，这个假设马上就会出错：

- `.text` 运行时可能映射到 `0x401000`，但它在文件里通常从较小的 raw 偏移开始；
- BSS 需要占一段运行时地址，却不应该把整段零都写进文件；
- 文件头里的 RVA、raw pointer、raw size、virtual size 往往要等各段都写完后才能回填；
- 有些表要先在临时空间里生成、测量、改字节，再复制到真正的输出里。

所以本章只讲一件事：XIRASM 怎样把 逻辑地址 / RVA、raw 文件偏移 / FOA、最终进入文件的字节 和 临时虚拟输出 分开管理。

先分清 RVA 和 FOA

一个真实输出区域至少有四个量：

名称	说明
<code>origin</code>	区域的逻辑地址基准。标号地址从这里开始算；写 PE 时通常对应 RVA 或 image base + RVA，写 ELF 时通常对应虚拟地址。
<code>file_offset</code> / FOA	区域在 raw 文件中的起始偏移。
<code>logical size</code>	区域占用的逻辑地址范围。 <code>reserve</code> 会计入这个大小。
<code>file size</code>	区域最终实际写进 raw 文件的字节数。尾部 <code>reserve</code> 可以不计入这个大小。

这四个量不能互相代替。改 `origin` 不会自动插入文件填充；改 FOA 也不会改变标号地址。PE 里的 `VirtualAddress`、`PointerToRawData`、`VirtualSize`、`SizeOfRawData` 之所以容易写错，就是因为它们分别来自这几类信息。

写布局时常用这些查询：

查询	返回什么
<code>region_base()</code>	当前区域的 <code>origin</code> ，也就是标号地址的基准。
<code>here()</code>	当前逻辑地址。
<code>file_offset()</code>	当前已经确定的 raw 文件偏移；尾部只有 <code>reserve</code> 时，它仍停在真实文件尾部。
<code>file_cursor_real()</code>	下一个已经确定会写进 raw 文件的位置。可以把它当成“当前真实 FOA”。
<code>file_cursor_potential()</code>	如果把当前尾部 <code>reserve</code> 也保留成文件里的零填充，下一个 FOA 会是多少。
<code>tail_reserve_size()</code>	当前区域末尾还有多少 <code>reserve</code> 尚未进入 raw 文件。

注意：`file_cursor_potential()` 只是 API 名。实际判断很简单：它等于“真实 FOA + 尾部还没落地的 `reserve`”。

这些 API 主要给自定义格式和格式库内部使用。普通 PE/COFF/ELF 用法优先看 `format.inc` 包装层；它会替你维护大部分 RVA/FOA 关系。

`origin` 只改逻辑地址

`origin(address)` 修改当前区域的逻辑地址基准，不修改文件偏移。

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

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

```
assert(region_base() == 0x4000);
assert(label_addr("start") == 0x4000);
assert(here() == 0x4001);
assert(file_offset() == 1);
```

输出只有一个字节：

```
aa
```

这里 **start** 的地址是 **0x4000**，但文件里仍然只写了一个字节。**origin** 适合 flat 输出设置装载地址，或者在某个区域内调整标号地址基准。它不会创建新区，也不会把文件位置跳到 **0x4000**。

region.begin 显式指定 RVA 和 FOA

region.begin(name, origin, file_offset) 开始一个新的真实输出区域。它同时指定：

- **origin**：这个区域的逻辑地址基准；
- **file_offset**：这个区域在 raw 文件中的起始 FOA。

```
// header 的逻辑地址从 0x1000 开始，raw 文件偏移从 0 开始。
region.begin("header", 0x1000, 0);
```

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

```
// payload 有自己的逻辑地址，并从 FOA 0x10 开始。
region.begin("payload", 0x2000, 0x10);
```

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

```
assert(label_addr("header") == 0x1000);
assert(label_addr("payload") == 0x2000);
```

header 占 FOA 0 和 1，**payload** 从 FOA 0x10 开始。中间没有真实区域写入的 raw 范围，最终 flat 文件会补零：

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

region.begin 不是“插入字节”，也不是 PE/ELF/COFF 的 section 声明。它只告诉 XIRASM：接下来的输出属于一个新区域，并且这个区域的逻辑地址和 raw 文件偏移是多少。

区域之间是否按 FOA 递增、是否重叠、是否留洞，调用者自己负责。写标准格式时不要直接散用 **region.begin** 拼 PE/ELF/COFF 头；优先使用 **format.inc**，只在手写自定义格式或实现格式辅助函数时直接管理这些区域。

reserve 推进逻辑大小，但尾部可以不进文件

写真实字节时，逻辑地址和 raw 文件尾部都会前进：

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

```
assert(file_cursor_real() == 1);
assert(file_cursor_potential() == 1);
assert(tail_reserve_size() == 0);
```

reserve(n) 不一样。它推进逻辑地址，但如果它还在区域尾部，XIRASM 不会立刻把它写成文件里的零：

```
emit.u8(0xaa);
reserve(3);
```

```
assert(here() == 4);
assert(file_cursor_real() == 1);
```

```
assert(file_cursor_potential() == 4);
assert(tail_reserve_size() == 3);
```

这时区域逻辑上已经占 4 字节，但 raw 文件只有 **aa** 一个字节。尾部 3 字节可以被裁掉，也可以在后续选择中变成文件里的零填充。

如果 **reserve** 后面又写真实字节，它就不再是“尾部预留”，而是文件中间的间隙，必须进入 raw 文件：

```
emit.u8(0xaa);
reserve(3);
emit.u8(0xbb);

assert(file_cursor_real() == 5);
assert(file_cursor_potential() == 5);
assert(tail_reserve_size() == 0);
```

文件内容：

```
aa 00 00 00 bb
```

这条规则是写 BSS、section 尾部 padding、raw size / virtual size 的基础：尾部 reserve 增加逻辑大小；只有它被后续真实字节夹在中间，或你主动选择保留它时，才会变成 raw 文件里的零。

output.section 裁掉尾部 reserve

output.section(name, origin) 用“当前真实 FOA”开始下一个区域。也就是：前一个区域末尾还没有进入文件的 **reserve** 会被裁掉。

```
emit.u8(0x41);
reserve(3);

// next 从 FOA 1 开始，前面的尾部 reserve 不写入文件。
output.section("next", 0x2000);
emit.u8(0x42);
```

第一个区域的逻辑大小仍然是 4，但 raw 文件大小只有 1。新区域接在 FOA 1，所以输出是：

```
41 42
```

这就是 BSS 一类区域常要的行为：内存里要有地址范围，文件里不要写一大段零。

output.section 只裁掉“还在区域尾部”的 reserve。已经位于文件中间的间隙不会被删除。

output.org 保留尾部 reserve

output.org(name, origin) 用“把尾部 reserve 也算进去后的 FOA”开始下一个区域。也就是：前一个区域末尾的 **reserve** 会变成 raw 文件里的零填充。

```
emit.u8(0x41);
reserve(3);

// next 从 FOA 4 开始，前面的三个 reserve 字节保留为文件间隙。
output.org("next", 0x2000);
emit.u8(0x42);
```

输出是：

```
41 00 00 00 42
```

所以二者的选择可以直接按 FOA 说：

操作	新区域从哪里开始
output.section	从真实 raw 文件尾部开始，尾部 reserve 不进文件。
output.org	从 reserve 之后开始，尾部 reserve 作为零填充保留在文件里。

二者都会给新区域设置新的 **origin**。**origin** 只影响标号地址，不影响上面这个 FOA 选择。

region.file_align 只对齐 raw size

`region.file_align(alignment)` 对齐的是当前区域最终写入 raw 文件的大小。它不推进逻辑地址，也不会把尾部 reserve 变成逻辑内容。

```
region.begin("first", 0x1000, 0);

emit.bytes(b"ABC");
reserve(13);

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

// 裁掉尾部 reserve 后，把 raw size 对齐到 8。
region.file_align(8);

region.begin("second", 0x2000, 8);
emit.u8(0x5a);
```

ABC 三个真实字节参与 raw size 对齐，XIRASM 补 5 个零，把第一区域的 raw size 补到 8。第二区域从 FOA 8 开始：

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

第一区域的逻辑大小仍然是 16，因为 `reserve(13)` 已经推进了逻辑地址。`region.file_align` 改的是 raw 文件大小，不是 RVA 范围。

对齐值必须是非零的 2 的幂。调用 `region.file_align` 后，当前区域的文件输出已经结束；继续写真实字节前，应开始另一个区域。

它和 `align` 的区别很重要：

- `align` 是普通输出操作，会推进逻辑地址；如果形成文件间隙，就会写填充字节。
- `region.file_align` 是区域收尾操作，只对齐该区域的 raw size。

需要“RVA 也往前走”时用 `align`；只需要“raw size 对齐到 FileAlignment”时用 `region.file_align`。

虚拟区域是临时输出，不自动进文件

虚拟区域用于临时组装、测量、读取或改写字节。它有自己的逻辑地址和字节内容，但不会自动写进最终文件。

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

table:
emit.u32(0x11223344);
store.u32(table, load.u32(table) ^ 0x01010101);
const encoded: bytes = load.bytes(table, 4)

virtual.end();

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

虚拟区域里最初是：

```
44 33 22 11
```

变换后复制到主输出的是：

```
45 32 23 10
```

虚拟区域适合做资源表、导出表、字符串池、校验数据等临时构造。里面可以写数据、`reserve`、`align`、定义标号、写 ISA 指令，也可以用 `load.*` 和 `store.*` 读写这些临时字节。

但它有两个边界：

- 每个 `virtual.begin` 必须有对应的 `virtual.end`。
- 虚拟区域里不能启动主输出区域；`output.section` 和 `output.org` 只能在回到真实输出后调用。

虚拟区域里的地址不是最终文件位置。要让虚拟内容进入文件，必须像上面的例子一样显式 `emit.bytes(...)` 或用格式库提供的复制流程。

省略虚拟 origin：从当前位置的逻辑地址开始

`virtual.begin()` 可以不传参数。省略时，虚拟区域的逻辑地址从外围区域当前的 `here()` 开始。

```
origin(0x4000);
emit.u8(0xaa);

// 当前 here() 是 0x4001，虚拟区域也从这里开始算地址。
virtual.begin();

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

virtual.end();

emit.bytes(copied);
```

`scratch` 的逻辑地址是 `0x4001`，但虚拟输出不会替换主输出，也不会推进主输出位置。最终文件是：

```
aa 34 12
```

这种写法适合“按当前地址假装组装一段内容，然后只取它的字节结果”的场景。

最终区域信息只能在收尾阶段查询

写出阶段能知道“当前正在写到哪里”，但不能查询所有区域最终的 raw size / logical size。原因很简单：后面的区域、尾部 reserve、文件对齐、late layout 和最终收尾都可能影响最终布局事实。

区域最终事实只能在收尾阶段查询：

查询	返回什么
<code>region_file_offset(address)</code>	包含该地址的区域最终从哪个 FOA 开始。
<code>region_file_size(address)</code>	该区域最终写进 raw 文件的字节数。
<code>region_logical_size(address)</code>	该区域最终占用的逻辑地址大小，包含尾部 reserve。

这三个查询通常用于回填文件头。例如 PE section header 里的 `PointerToRawData`、`SizeOfRawData`、`VirtualSize`，或者 ELF program header 里的 `p_offset`、`p_filesz`、`p_memsz`，都应该等布局稳定后再写。

如果在普通写出阶段调用这些查询，会因为没有最终 output image 而报错。下一章介绍收尾阶段：它可以读取最终布局、回填已经存在的字节、做断言和校验，但不能再改变布局。

怎么选

只设置 `origin`：

- 单个 flat 输出需要非零装载地址；
- 文件偏移和逻辑位置仍然同步推进。

用 `region.begin`：

- 逻辑地址和 FOA 都已经明确；
- 你正在手写自定义格式，或者实现 `format.inc` 这类格式接口。

用 `output.section`:

- 新区域应该紧接真实 raw 文件尾部;
- 前一区域尾部 reserve 只表示逻辑大小, 不应该占文件空间。

用 `output.org`:

- 新区域应该从 reserve 之后开始;
- 前一区域尾部 reserve 必须变成文件里的零填充。

用 `region.file_align`:

- 区域 raw size 需要按文件格式对齐;
- 不希望改变逻辑地址范围。

用虚拟区域:

- 需要临时生成、测量、读取或改写一段字节;
- 这些临时字节只有显式复制后才进入最终文件。

一句话总结:

- 标号、`here()`、`origin` 讲的是逻辑地址 / RVA;
- `file_offset()`、`file_cursor_real()` 讲的是已经确定的 FOA;
- `file_cursor_potential()` 只是在问 “如果尾部 reserve 也进文件, FOA 会到哪里” ;
- `region_file_*` 和 `region_logical_size` 只能在最终布局稳定后用于回填和检查。

下一章介绍收尾处理: 读取稳定布局、回填字节、计算校验和、验证映像, 但不再改变布局。

[返回目录](#)

第 12 章：收尾处理

很多二进制字段第一次写出来时还不知道最终值：

- 文件头里的大小字段要等 payload 写完才知道；
- PE/ELF/COFF 的 raw pointer、raw size、virtual size 要等区域布局稳定后才能写；
- 校验和要等指令编码、fixup 修补、后期布局都完成后才能算；
- 符号表、字符串表、重定位表可能要等主源码登记完后再生成。

XIRASM 不会反复执行整份源码去“碰运气收敛”。它把后期工作分成两个阶段：

阶段	能做什么	不能做什么
late_layout	在最终映像封存前，追加或创建仍会参与布局的真实输出。	不能当普通源码块用；不能声明局部变量、标号、函数、循环或写 ISA 指令。
defer	在布局稳定后，读取最终字节、回填已有字段、断言和报错。	不能改变布局；不能 emit、reserve、align、切换区域或创建新字节。

一句话：需要新增字节或新增区域，用 late_layout；只需要回填已有字节或检查最终结果，用 defer。

defer：回填已经存在的字段

最常见流程是：

- 先写固定宽度的占位字段；
- 正常写 payload；
- 在 defer 里把最终值写回占位字段。

```
size_field:
emit.u16(0);

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

defer {
    store.u16(size_field, payload_end - payload);
}
```

最终输出：

```
03 00 41 42 43
```

size_field 的两个字节在普通写出阶段已经存在。defer 只是把这两个字节从 00 00 改成 03 00；它没有插入新字节，也没有移动后面的 payload。

defer 可以写在它引用的标号之前：

```
defer {
    store.u16(size_field, payload_end - payload);
}

size_field:
emit.u16(0);

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

等 defer 执行时，标号已经解析，布局也已经稳定。

在 defer 中读取和修改最终字节

`load.u8`、`load.u16`、`load.u32`、`load.u64` 读取最终输出中的小端整数。`load.bytes(address, count)` 读取一段字节。

```
origin(0x4000);

header:
emit.u32(0);
emit.u32(0);

body:
emit.bytes(b"ABCD");
tail:
emit.bytes(b"????");
image_end:

defer {
    store.u32(header, image_end - body);
    store.u32(header + 4, body - region_base());
    store.bytes(tail, b"OK!!");

    assert(load.u32(header) == 8);
    assert(load.u32(header + 4) == 8);
    assert(load.bytes(tail, 4) == b"OK!!");
}
```

最终输出：

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

所有 `load.*` 和 `store.*` 都会检查范围。`defer` 只能访问最终文件里真实存在的字节。第 11 章说过，尾部 `reserve` 可能只增加逻辑大小，不进入 raw 文件；这种被裁掉的尾部预留不能当成可写占位字段。

`store.bytes` 接受字符串或 `bytes` 值。整数写入会检查宽度，值放不进目标宽度时会报错。

计算校验和

`defer` 内可以声明 `const`、`let`，可以给局部 `let` 赋值，也可以用有界 `while`。

```
origin(0x5000);

checksum:
emit.u16(0);

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

defer {
    let cursor = payload
    let sum = 0

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

    store.u16(checksum, sum);
    assert(load.u16(checksum) == 266);
}
```

266 是 0x010a，最终输出：

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

`while` 使用普通 Meta 循环的编译期迭代限制。`for` 目前不能放进 `defer`；需要扫描字节时，用边界清楚的 `while`。

查询最终区域信息

第 11 章的区域最终信息只能在 `defer` 中查询，因为它们依赖最终布局：

```
region.begin("payload", 0x5000, 0);
file_size_field:
```

```

emit.u32(0);
logical_size_field:
emit.u32(0);

body:
emit.u8(0xaa);
reserve(3);

defer {
    store.u32(file_size_field, region_file_size(body));
    store.u32(logical_size_field, region_logical_size(body));

    assert(region_file_offset(body) == 0);
    assert(region_file_size(body) == 9);
    assert(region_logical_size(body) == 12);
}

```

输出：

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

这里文件大小是 9：两个 **u32** 字段占 8 字节，**body** 占 1 字节。尾部 **reserve(3)** 不写入 raw 文件，所以不计入 **region_file_size**。逻辑大小是 12，因为尾部 reserve 仍然占逻辑地址范围。

三个查询分别是：

- **region_file_offset(address)**：包含该地址的区域最终从哪个 FOA 开始；
- **region_file_size(address)**：该区域最终写入 raw 文件的字节数；
- **region_logical_size(address)**：该区域最终占用的逻辑地址大小，包含尾部 reserve。

它们不是写出阶段的“当前 FOA”查询。需要当前 FOA 时用 **file_offset()** / **file_cursor_real()**；需要最终区域大小时，在 **defer** 里用 **region_file_*** / **region_logical_size**。

用函数登记回填逻辑

过程函数可以登记 **defer** 块，并捕获调用时传入的参数：

```

fn patch_u16(address: u64, value: u64) {
    defer {
        store.u16(address, value);
    }
}

field:
emit.u16(0);

patch_u16(field, 0x1234);

```

最终输出：

```
34 12
```

注意执行时机：**patch_u16(...)** 在普通源码阶段调用；调用时的 **address** 和 **value** 被保存到登记的 **defer** 块里。真正写字节发生在最终布局稳定之后。

这种写法适合做类型明确的小型回填工具，比如 **patch_u32**、**patch_size_field**、**patch_checksum_field**。

defer 的允许范围

defer 适合最终检查和回填。它可以使用：

- **const**、**let** 和局部赋值；
- **if** / **else**；
- **while**、**break**、**continue**；

- `load.*`、`store.u8/u16/u32/u64`、`store.bytes`;
- `assert`、`print`、`warn`、`err`;
- 纯表达式和值函数;
- 标签、最终区域信息、已稳定的输出字节。

它不能做任何会改变布局的事。例如：

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

会被拒绝。下列操作也不允许放在 `defer` 中：

- 写 ISA 指令;
- 定义标号;
- `emit.*`、`db/dw/dd/...`;
- `reserve`、`align`、`pad`、`pad_to`;
- `origin`、`region.begin`、`output.section`、`output.org`、`virtual.begin`;
- 嵌套 `defer` 或 `late_layout`;
- 声明函数、结构体或加载源文件;
- 读外部文件。

需要空间就提前写占位或在 `late_layout` 里创建；`defer` 只能修改已经存在的字节。

多个 `defer` 的执行顺序

`defer` 按登记顺序执行：

```
emit.u8(0);  
  
defer {  
    store.u8(0, 1);  
}  
  
defer {  
    store.u8(0, load.u8(0) + 1);  
}
```

第二个块能看到第一个块的修改，所以最终输出是：

02

如果多个 `defer` 修改同一地址，后执行的块会看到前面的结果。不要把同一个字段分散到多个互相依赖的 `defer`，除非顺序就是你想要的格式规则。

`defer` 运行在这些步骤之后：

- 普通源码处理;
- `late_layout`;
- 指令编码和布局松弛;
- fixup 解析;
- 最终文件字节生成;
- fixup 修补。

因此 **defer** 看到的是即将写出的最终映像：指令已经编码，引用已经解析，后期布局创建的字节也已经进入最终布局。

late_layout: 封存前新增真实布局

late_layout 用于“主源码已经登记完，但最终布局还没封存”时创建真实输出。

最简单的形式是追加字节：

```
emit.u8(0x10);

late_layout {
    emit.u8(0x20);
}

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

late_layout 块按登记顺序执行一次，默认从默认输出区域的尾部继续。输出是：

```
10 20 30
```

这说明默认行为是“接在当前默认输出尾部”。但 **late_layout** 不是只能追加到整个文件末尾。它允许调用输出区域 API，所以你可以在块中显式打开一个真实区域，把晚生成的数据放到你指定的 FOA。

例如，先在虚拟区域里生成表，再在后期布局阶段把表放到指定 raw 文件偏移：

```
table_foa_field:
emit.u32(0);
emit.bytes(b"HDR");

const table_origin: u64 = 0x8000
const table_foa: u64 = 0x10

virtual.begin(0);
table_tmp:
emit.bytes(b"TAB");
table_tmp_end:
virtual.end();

late_layout {
    region.begin("late-table", table_origin, table_foa);
    emit.bytes(load.bytes(table_tmp, table_tmp_end - table_tmp));
}

defer {
    store.u32(table_foa_field, table_foa);
}
```

这里 **late_layout** 没有“往最终文件里插入字节”。它是在最终映像生成前创建了一个真实区域：逻辑地址从 **0x8000** 开始，raw 文件偏移从 **0x10** 开始。最终文件怎么补洞、是否重叠、头字段是否一致，都由调用者的区域布局负责。

如果你已经用 **region.begin** / **output.section** / **output.org** 构造了类似 PE 的多段布局，那么 **late_layout** 中的晚生成表也可以放进某个明确的自定义区域；前提是你显式切到那个区域或给出正确 FOA。只写 **emit.*** 时才是沿默认输出尾部继续。

标准 PE/COFF/ELF 优先用第 14 章的 **format.inc** 接口。直接写 **late_layout + region.begin** 更适合自定义格式，或者实现格式库内部的符号表、字符串表、重定位表等晚生成内容。

late_layout 会影响尾部 reserve

因为 **late_layout** 发生在最终布局之前，它新增的真实字节会参与 raw 文件布局。追加在尾部 reserve 后面时，前面的 reserve 会变成文件中间的零填充：

```
emit.u8(0xaa);
reserve(3);
```

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

最终输出：

```
aa 00 00 00 bb
```

如果这不是你想要的行为，不要在同一个区域尾部 `reserve` 后直接追加真实字节。先用 `output.section` 裁掉尾部 `reserve`，或用 `region.begin` 切到明确的目标区域。

late_layout 的限制

`late_layout` 比普通源码窄得多。它只接受 API 调用和 `if` 分支；没有普通局部作用域。

允许的方向包括：

- `emit.*`、`emit.bytes`、`emit.struct`、`db/dw/dd/...`;
- `reserve`、`pad`、`pad_to`、`align`;
- `origin`、`region.begin`、`region.file_align`;
- `output.section`、`output.org`;
- `virtual.begin`、`virtual.end`;
- `store.u8/u16/u32/u64`、`store.bytes`;
- `assert`、`print`、`warn`、`err`;
- `if` / `else`。

不允许：

- `let` / `const` 声明;
- 赋值;
- `while` / `for`;
- 标号;
- ISA 指令文本;
- 函数、结构体、嵌套 `late_layout` 或 `defer`;
- 读外部文件或加载源模块。

需要计算的值、字节数组、表大小、目标 FOA，应在普通源码阶段先算好，再作为 API 参数用于 `late_layout`。需要读文件时，也应在普通阶段读取成 `bytes`，不要在 `late_layout` 或 `defer` 里读。

`late_layout` 只执行一次。它不是依赖反复执行源码来收敛不稳定值的多遍模型。

选择阶段

用普通源码：

- 字节可以按正常顺序写出;
- 标号、局部变量、函数、循环、文件读取都需要正常语言能力。

用 `late_layout`：

- 必须等主源码登记完后才知道要写哪些真实字节;

- 晚生成表、字符串池、重定位记录需要进入最终文件；
- 需要显式放到某个 FOA 或沿默认输出尾部继续；
- 这些字节必须影响最终 raw size、logical size、偏移和回填。

用 **defer**:

- 固定宽度占位字段需要最终值；
- 校验和需要完整最终字节；
- 需要最终 **region_file_offset** / **region_file_size** / **region_logical_size**；
- 只修补已有字节，不创建空间。

常见安全规则：

- 不要用 **defer** 创建缺失空间；
- 不要在 **late_layout** 里写需要普通局部变量和循环的逻辑；
- 不要把尾部 reserve 是否进入文件交给猜测，明确选择 **output.section** 或 **output.org**；
- 构造标准 PE/COFF/ELF 时优先使用 **format.inc**，不要手写重复的布局关系。

下一章进入第三部分，介绍 flat 和自定义二进制文件：如何把标签、结构体、区域、后期布局和收尾回填组合成完整文件格式。

第三部分：构建程序

返回目录

第 13 章：Flat Binary 与自定义文件格式

XIRASM 默认生成 flat binary：输出文件只包含源码明确写出的指令、数据、填充、区域字节和后期布局字节。它不会自动添加 PE、ELF、COFF 头，也不会自动生成入口点、section 表、segment 表或重定位表。

这种模式适合：

- 引导扇区、固件映像、ROM 表；
- 协议消息、测试数据、资源容器；
- 私有二进制格式；
- 由其他程序加载的一小段机器码；
- 格式库内部用于生成头、表、字符串池的底层构造。

flat binary 不等于“没有结构”。你仍然可以用标号表示位置，用 **packed struct** 描述记录，用区域分开 RVA 和 FOA，用 **late_layout** 创建晚生成表，用 **defer** 回填最终字段。

原始机器码

最简单的 flat 文件就是指令字节本身：

```
x86.use64();

entry:
    mov eax, 42
    ret
```

输出：

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

这里没有操作系统文件头，也没有“入口点字段”。**entry** 只是 XIRASM 标号。加载这些字节的程序必须自己知道 ISA、模式、装载地址和调用约定。

文件头加 Payload

自定义格式通常先写固定头，再写 payload：

```
emit.bytes(b"RAW1");
emit.u16(1);
emit.u16(0);

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

输出：

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

前四字节是签名，第一个 **u16** 是版本号，第二个 **u16** 保留，后面是 payload。没有额外格式声明；源码写出顺序就是文件顺序。

用 **packed struct** 描述记录

二进制记录需要固定字段和固定宽度时，用 **packed struct** 把格式写清楚：

```
packed struct ChunkHeader {
    kind: u16
    size: u16
}

emit.struct(ChunkHeader {
    kind: 1,
```

```
    size: 3,
  });
  emit.bytes(b"ABC");
```

输出：

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

packed struct 不插入隐式对齐间隙，适合描述磁盘格式、网络格式和紧凑表项。只有文件格式本身要求普通结构体的对齐和尾部填充时，才用普通 **struct**。

重复记录可以封装成函数：

```
fn emit_record(tag: u8, value: u32) {
  emit.u8(tag);
  emit.u32(value);
}

emit_record(1, 0x11223344);
emit_record(2, 0xaabbccdd);
```

输出：

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

函数负责保持记录写法一致；最终文件仍然按调用顺序生成。

用 defer 回填文件头

不要手工维护大小、偏移和校验和。先写固定宽度占位字段，再用标号和最终字节回填：

```
origin(0);

magic:
emit.bytes(b"XIF1");
size_field:
emit.u32(0);
payload_foa_field:
emit.u32(0);
checksum_field:
emit.u32(0);

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

defer {
  store.u32(size_field, payload_end - magic);
  store.u32(payload_foa_field, payload - region_base());
  store.u32(checksum_field, load.u8(payload) + load.u8(payload + 1));

  assert(load.bytes(magic, 4) == b"XIF1");
  assert(load.u32(size_field) == 20);
  assert(load.u32(payload_foa_field) == payload - magic);
  assert(load.u32(checksum_field) == 0x9a);
}
```

输出：

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

这个例子没有硬编码总大小和 payload 偏移。头或 payload 改了，**defer** 会按最终布局写回新值。

本例从 **origin(0)** 开始，逻辑地址差值正好等于文件内相对偏移。只要你开始使用多个区域，这个关系就不一定成立。

明确字段要的是 RVA 还是 FOA

自定义格式里最容易出错的是把逻辑地址和 raw 文件偏移混用。写字段前先问清楚它要哪一个：

- 需要逻辑地址 / RVA：用标号值，或用标号相减得到逻辑距离；

- 需要当前 raw 文件偏移 / FOA: 写出阶段用 `file_offset()` 或 `file_cursor_real()`;
- 需要某个区域最终 FOA: 在 `defer` 里用 `region_file_offset(address)`;
- 需要 raw size: 在 `defer` 里用 `region_file_size(address)`;
- 需要 virtual size / logical size: 在 `defer` 里用 `region_logical_size(address)`。

例子:

```
region.begin("header", 0x1000, 0);
emit.bytes(b"HDR0");

output.section("payload", 0x2000);
payload:
emit.bytes(b"DATA");

defer {
    assert(payload == 0x2000);
    assert(region_file_offset(payload) == 4);
}
```

raw 文件仍然紧凑:

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

`payload` 的逻辑地址是 `0x2000`, 但它在文件里从 FOA 4 开始。除非格式明确规定二者相等, 否则不要从逻辑地址推导 FOA。

表示文件间隙和 file-free 尾部

`reserve` 是否进入 raw 文件, 取决于它是不是仍然处在区域尾部:

- 中间间隙: 后面还有真实字节, `reserve` 会写成文件里的零;
- 尾部预留: 只增加逻辑大小, 可以不占 raw 文件空间。

```
region.begin("image", 0x5000, 0);

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

emit.u8(0xaa);
reserve(3);
emit.u8(0xee);
reserve(8);

defer {
    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);
}
```

raw 文件只有 17 字节:

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

中间 3 字节 `reserve` 因为后面有 `0xee`, 所以进入 raw 文件。最后 8 字节 `reserve` 仍在区域尾部, 只把 logical size 增加到 25, 不增加 raw size。

这正是 BSS、未初始化尾部、节尾虚拟空间这类格式字段的基础: 文件记录已初始化前缀, 剩余地址范围由加载器或消费程序提供。

late_layout: 晚生成但仍参与布局

只有在真实字节必须等主源码登记完之后才能创建时，才用 `late_layout`。最简单的例子是追加尾部：

```
total_size:
emit.u32(0);
emit.bytes(b"DATA");

late_layout {
    emit.bytes(b"END!");
}

defer {
    store.u32(total_size, region_file_size(total_size));
    assert(load.u32(total_size) == 12);
}
```

输出：

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

这里 `late_layout` 写出的 `END!` 会参与最终 raw size。 `defer` 能看到它，并把总大小回填到开头。

如果 `late_layout` 里只写 `emit.*`，它就是从默认输出区域的尾部继续。若晚生成内容应该落到某个自定义表区、数据区或指定 FOA，就必须在块里显式切区域：

```
table_foa_field:
emit.u32(0);
emit.bytes(b"HDR");

const table_origin: u64 = 0x8000
const table_foa: u64 = 0x10

virtual.begin(0);
table_tmp:
emit.bytes(b"TAB");
table_tmp_end:
virtual.end();

late_layout {
    region.begin("late-table", table_origin, table_foa);
    emit.bytes(load.bytes(table_tmp, table_tmp_end - table_tmp));
}

defer {
    store.u32(table_foa_field, table_foa);
}
```

这不是在最终文件里“插入”字节，而是在最终映像封存前创建一个真实区域。它可以放到文件尾，也可以放到你指定的 FOA；关键是你必须自己保证区域不重叠、头字段一致、raw size/logical size 符合格式规则。

如果只是回填头部已有字段，用 `defer`。如果缺的是变长表、字符串池、重定位记录等真实字节，就要在普通源码或 `late_layout` 里创建它们。

用断言保护自定义格式

自定义格式应该把关键不变量写成断言：

- 签名、版本和标志字段正确；
- 偏移字段指向预期区域；
- 计数等于实际写出的记录数；
- raw size 和 logical size 符合格式规则；
- 校验和覆盖正确字节范围；
- 尾部 reserve 是否进入文件符合预期。

`defer` 中的断言检查最终输出字节，包括已经编码的指令、已解析的 fixup、`late_layout` 生成的字节和所有回填结果。断言失败时汇编停止，避免产出表面上有文件头、实际上字段已经错位的文件。

大小回填和对应断言通常放在同一个 `defer` 块里，这样字段来源和验证条件靠在一起。

推荐组织顺序

小型自定义格式可以按这个顺序写：

- 声明常量、记录类型和辅助函数；
- 写固定文件头，占位字段先写 0；
- 写 payload 和普通表；
- 用 `late_layout` 创建确实需要晚出的真实字节；
- 用 `defer` 回填大小、偏移、校验和，并断言最终结果。

格式变复杂后仍然保持同样分工：

- 源码和函数决定写哪些记录；
- 标号和区域描述逻辑地址、FOA 和大小；
- `virtual.begin` 用于临时生成和测量；
- `late_layout` 创建必须参与最终布局的晚生成字节；
- `defer` 只回填和验证稳定后的字段。

不要把一堆硬编码偏移散落在源码里。把固定格式常量放在一起；把记录写出封装成函数；把最终值统一由标号、区域查询和断言推导出来。

什么时候用格式接口

flat 输出能表达任意字节，但不代表应该手写标准可执行文件或目标文件格式。PE、COFF、ELF 还要维护文件头、section/segment 表、权限、导入、导出、重定位、BSS、对齐和加载器规则。

标准格式优先用 `format.inc` 包装层。语言指南只解释底层机制：RVA/FOA、区域、虚拟输出、`late_layout` 和 `defer`。完整普通用法见《格式教程》。只有在实现新的格式接口或手写私有格式时，才需要直接使用本章这些底层能力。

[返回目录](#)

第 14 章：可执行文件与目标文件格式

PE、COFF、ELF 不是“把指令和数据拼在一起”就够了。它们还要有文件头、section 或 segment 表、权限、入口点、导入导出表、符号表和重定位表；链接器和加载器会按这些结构解释文件。

XIRASM 用格式库生成这些结构：

```
// 导入常用的 PE/COFF/ELF 格式接口。
import("format/format.inc");
```

使用 **format.inc** 时，源码描述“文件里有哪些 section 或 segment、入口在哪里、需要哪些表”。表计数、表项顺序、文件偏移、虚拟地址和文件头字段由格式库推导。

本章只讲通用工作顺序。完整的格式选项、导入、导出、重定位、共享库和目标文件示例，请看《格式教程》。如果你确实要自己安排文件头和表项，再看《高级格式构造指南》。

先声明映像再写内容

使用格式库的程序，第一步是声明输出文件包含哪些 section 或 segment。每个描述符给出名称、用途和权限。

描述符列表直接决定后面的格式结构：

- 列表长度决定表项数量；
- 列表顺序决定表项顺序；
- 名称用于后续打开对应内容块；
- 用途决定这块内容是代码、数据、BSS、导入表、导出表还是重定位表；
- 权限会写入 section 或 segment 属性。

源文件不用再手填表计数或行号。增删描述符时，生成的格式结构会跟着变化。

完整示例：ELF64 程序

以下示例创建 x86-64 ELF 可执行文件，含一个可加载、可读、可执行的 segment：

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

// 在声明的 .text 段中写入程序入口代码。
format_segment_begin(image, ".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(image, ".text");

// 绑定入口标号，随后完成文件头和表项。
format_entry_mut(image, start)
format_finish(image);
```

在 x86-64 Linux 下，这个程序以状态码 0 退出。

基本顺序是：

- `format_elf64` 创建 ELF64 文件配置。
- `format_segment` 声明一个 segment 及其权限。
- `format_begin` 写出这份配置需要的格式结构。
- `format_segment_begin` 和 `format_segment_end` 包住这个 segment 的实际指令和数据。
- `format_entry_mut` 绑定入口标号，`format_finish` 完成文件。

源文件不需要提供程序头计数、表行号、文件偏移、加载地址或入口点字段偏移。格式库会根据配置和已经完成的 segment 布局推导这些值。

section 和 segment 怎么选

调用名称沿用文件格式自己的术语：

- PE 映像和目标文件用 section；
- ELF 可执行文件和共享库用 segment；
- ELF 目标文件用 section，不涉及运行时装载。

对应的生命周期调用：

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

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

只能打开配置里声明过的名称，并按声明时的用途使用它。这样格式库才能把已写出的字节、逻辑大小、权限和生成的表关联到正确的格式条目。

格式族

格式库支持以下映像类：

映像类	常规构造函数
PE32 / PE64 映像	<code>format_pe32</code> 、 <code>format_pe64</code>
COFF32 / COFF64 目标文件	<code>format_coff32</code> 、 <code>format_coff64</code>
ELF32 / ELF64 可执行文件	<code>format_elf32</code> 、 <code>format_elf64</code>
ELF32 / ELF64 目标文件	<code>format_elfobj32</code> 、 <code>format_elfobj64</code>
ELF64 共享库	<code>format_elf64_so</code>

构造函数选项区分可执行文件与 DLL、控制台/GUI 子系统、位置无关执行、NX 策略、地址随机化等文件属性。Section/segment 描述符携带内容用途，以及读、写、执行或可丢弃等权限。

完整的选项和描述符列表属于参考材料，本语言指南不重复列出。

入口点绑定

可执行文件应在源码定义完入口代码后绑定入口标号：

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

`format_entry_mut` 会直接更新第一个参数传入的 `let` 绑定；这里不需要维护多个不可变中间副本。`format_finish` 随后验证可执行文件具备所需入口信息。

目标文件和部分库形式不需要入口点。按所选格式的生命周期来，不要加无意义的入口标签。

生成的格式内容

有些格式内容不应该当成普通字节手写。它们来自专门的声明：

- 导入导出表
- 符号表和字符串表
- 重定位记录
- 基址重定位段
- 动态元数据
- 资源目录

这些 API 接受名称、标号、权限和重定位类型。section 编号、符号索引、表偏移、行号和计数由格式库推导。

不要用硬编码的表位置替代这些声明。格式库的价值正在于让表项和实际布局保持一致。

从 `format.inc` 开始

写常见 PE、COFF、ELF 文件时，先从 `format/format.inc` 开始。它负责协调格式属性、描述符、命名内容块、入口和生成表。

各格式特有的 include 文件会暴露更细的构造函数，例如文件头、section、segment、目录项或动态表。只有在 `format.inc` 无法表达目标布局时，才直接使用这些函数。

只是生成常见的多 section 可执行文件、DLL、目标文件、共享库、导入表、导出表或重定位表时，不需要绕过 `format.inc`。

不要混用两套写法。更细的构造函数通常要求调用者自己管理计数、行号、偏移和格式不变式；这些正是 `format.inc` 会替你维护的内容。

第 11 到 13 章的 `region.begin`、虚拟区域、`late_layout` 和 `defer` 只负责布局：字节放在文件哪里、逻辑地址是多少、最终字段何时回填。它们不会自动生成 PE/COFF/ELF 的 section、segment、符号表或重定位表记录。需要这些格式记录时，用 `format_section_begin`、`format_segment_begin` 和对应的格式 API。只有在实现新格式接口，或 `format.inc` 表达不了的专用布局时，才直接组合区域和收尾阶段。

后续阅读

《格式教程》提供可直接使用的模板、参数说明和 API 摘要：

- PE32/PE64 可执行文件和 DLL
- COFF32/COFF64 目标文件
- ELF32/ELF64 可执行文件和位置无关可执行文件
- ELF32/ELF64 目标文件
- ELF64 共享库
- 多 section 和多 segment
- 已初始化和未初始化数据
- 导入、导出、符号、重定位
- 什么时候用 `format.inc`，什么时候直接构造格式字段

已知所需格式系列时直接查 API Reference。

下一章讲诊断和源文件组织惯例。

[返回目录](#)

第 15 章：诊断与实用惯例

当源码把自己的假设说清楚时，汇编错误最容易修。XIRASM 提供了用于信息输出、非致命警告、主动失败和不变式检查的诊断工具。

源码组织也遵循同一个原则：命名清晰、坐标明确、辅助函数小而直接，并在该用 `format.inc` 的地方使用格式库。这样二进制布局才容易审查，也更容易修改。

阅读诊断信息

诊断会指出源文件位置、严重级别和说明：

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

该位置指向报告或触发问题的语句。解析、Meta 求值、指令编码、布局、fixup 解析和收尾处理都使用同一套面向源码的位置报告模型。

先看第一个错误。后面的失败可能只是早先的无效声明、缺失标号或被拒绝指令连带出来的结果。

信息与警告

`print` 输出信息，`warn` 输出非致命警告：

```
// 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);
```

汇编照常成功，输出：

```
7d
```

消息后的参数值也会打出：

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

有意保留的汇编时信息才用 `print`，不要把永久调试噪声留在源码里。输出仍然有效、但所选配置值得注意时用 `warn`。

警告不会改变输出，也不会让汇编失败。如果某个条件会产生非法文件或不受支持的程序，应改用错误。

主动报错

`err` 打出致命错误并终止：

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

额外参数会格式化在消息之后。传入导致配置无效的值，可以帮助使用者修正源码。

消息应说明违反的要求：

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

当源码能够说出预期属性时，不要只写 `invalid value` 这种空泛消息。

断言

assert 是编码不变式最直接的方式：

```
// 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数据。
emit.struct(Header {
    kind: 1,
    size: 3,
});
emit.bytes(b"ABC");
```

断言通过时不产生输出。完成后的字节是：

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

条件为假时，汇编停止，提供的消息会成为错误文本。

普通源码处理阶段已经知道的事实，直接用普通 **assert**：

- 类型大小和字段偏移
- 选项组合
- 列表和 map 内容
- 目标需求
- 常量范围
- 声明值之间的关系

需要稳定最终映像才能确认的事实，放在 **defer** 的 **assert** 里：

- 最终文件和逻辑大小
- 已解析的偏移和地址
- 回填的文件头字段
- 校验和
- 生成表的内容
- 指令 fixup 产生的字节

把断言放在输入事实变稳定的同一阶段，可以避免无意依赖临时布局值。

保护目标相关源码

把目标需求写在依赖它的代码附近：

```
// 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
```

输出：

```
31 c0 c3
```

目标条件在汇编时求值；它们选择或拒绝源码，不会生成运行时检查。

显式选择模式也让可复用源码更容易审查。读者不应靠猜来判断某段指令是写给 x86-16、x86-32、x86-64、RISC-V 32 还是 RISC-V 64。

用名字替代魔数

带有格式、协议或布局含义的数值应取名字：

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

算法内部的数值操作数可以保持局部。偏移、标志、结构大小、格式值和对齐策略通常应该命名。

当数值代表固定选项时，用命名常量。多个字段组成一条记录时，用 `struct`。需要在编译期收集一组声明、表项或配置时，用 `map` 或 `list`。

在名称中写清坐标

二进制写入器经常同时处理多套坐标。名称应体现坐标含义：

```
header_foa
text_rva
entry_address
payload_file_size
payload_logical_size
```

当 `offset` 可能指文件偏移、区域相对偏移、虚拟地址或结构字段偏移时，不要使用这种泛称。

辅助函数参数也一样。接受 `logical_address` 和 `file_offset` 的函数，比接受 `start` 和 `position` 的函数更不容易被误用。

推导布局事实

- 逻辑距离 → 标签相减
- 结构体布局 → `sizeof`、`offset_of`
- 当前 FOA → `file_offset()` / `file_cursor_real()`
- 稳定区域事实 → `defer` 里查
- 表计数和顺序 → 格式描述符列表决定
- 生成索引 → 符号和重定位声明决定

只有外部格式规范要求精确常量时，才硬编码偏移。即使硬编码，也要给常量取描述性名字，并尽可能断言周围布局。

保持收尾处理聚焦

普通源文件按普通顺序写出。

只有真实字节或区域必须在主源码之后追加时，才用 `late_layout`。只有需要检查或修补稳定映像时，才用 `defer`。

不要只是为了逃避源码组织，就把普通写出挪到后期阶段。一个明确的“文件头、payload、尾部、收尾处理”序列，比一张延迟副作用网络更容易理解。

让每个 `defer` 聚焦一组相关字段。一个回填并验证文件头的块，比一个到处修改无关区域的块更容易审查。

按职责分模块

实用的项目布局通常按职责拆分：

- 共享常量和数据类型
- 可重用的写出过程
- 目标相关指令例程
- 格式或容器构造
- 数据输入
- 顶层源文件（选配置、拼最终映像）

只需初始化一次的模块用 `import`。确实需要重复文本求值时才用 `include`。

公共辅助函数应明确说明输入。可以直接传配置、list、map、label 或 option 时，不要依赖隐藏的可变状态。

先用 `format.inc`

标准可执行文件和目标文件从 `format/format.inc` 开始：

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

`format.inc` 负责描述符计数、表顺序、生成索引、偏移和常见格式不变式。

只有在实现新的格式接口，或 `format.inc` 无法表达所需布局时，才直接导入更细的格式 `include`。不要仅仅因为某个函数暴露了更多数字字段就换过去；那些数字通常应该继续由 `format.inc` 维护。

项目专用文件应使用第 13 章的 flat 与自定义二进制技术，不要强行塞进操作系统格式。

在输出边界检查

在 XIRASM 把输出交给外部系统的边界处验证：

- 写自定义文件前断言最终结构
- 显式声明可执行文件的权限和重定位策略
- 写出前验证导入数据
- 尽早拒绝不支持的目标组合
- 入口点和导出符号保持命名
- 每个公共 `include` 保留一个可运行的小例子

验证应描述约定，而不是重复实现细节。断言表计数与声明列表一致很有用；断言每一步内部算术则更难维护。

找到合适的文档

本文档（语言指南）管：

- 编译期代码和处理器指令如何协作
- 值、控制流、函数、集合、token 的行为
- 标签、数据、区域、虚拟输出、收尾处理如何组织文件
- flat 输出和格式库的选择

API Reference 管：

- 精确的函数签名
- 接受的值类型
- 常量和选项标志
- 行为和错误约束

《格式教程》管：

- 完整的 PE、COFF、ELF 程序
- 可执行文件、DLL、目标文件、PIE、共享库
- 导入、导出、符号、资源、重定位
- 多 section 和多 segment
- `format.inc` 的参数、调用顺序和常见错误

这些文档把概念解释、查询材料和格式专用流程分开，各自保持可读。

最终检查清单

源文件完成前确认：

- ☐ 目标和指令模式显式指定
- ☐ 常量和值类型有描述性名字
- ☐ 标签替代硬编码地址
- ☐ 外部依赖的结构体大小已断言
- ☐ `import` 和 `include` 用对了模型
- ☐ 尾部 `reserve` 和 `raw` 文件间隙是故意的
- ☐ `late_layout` 只创建真正晚出的字节
- ☐ 收尾处理修改已有存储并验证稳定事实
- ☐ 标准 PE/COFF/ELF 先用 `format.inc`，除非确实要手写格式字段
- ☐ 致命条件给出可操作的消息
- ☐ 最终输出有可复现的汇编或执行验证

[返回目录](#)