

- 第 1 章：认识 XIRASM
 - 指令与编译期代码
 - 汇编第一个程序
 - 编译期代码向输出写入字节
 - 源码规则
- 第 2 章：值与绑定
 - 值只存在于汇编期间
 - 常量
 - 可变绑定
 - 类型注解与类型推断
 - 字符串与字节序列
 - 转义序列
 - 块作用域
- 第 3 章：表达式
 - 表达式在汇编期间求值
 - 算术运算
 - 位运算与移位
 - 比较与相等
 - 布尔逻辑与短路
 - 表达式中的函数调用
 - 字段访问
 - 运算符优先级
 - 表达式错误
- 第 4 章：控制流
 - 控制流在汇编期间执行
 - if 与 else
 - 条件生成的指令
 - 目标条件
 - for 与 range
 - for 与列表
 - while
 - break 与 continue
- 第 5 章：函数与作用域
 - 函数的两种形式
 - 过程
 - 参数与实参
 - let 参数

- 返回值函数
- 返回规则
- 函数局部作用域
- 声明顺序
- 递归与调用深度
- 指令形式的宏
- operand: 捕获的源码
- 在指令操作数里写参数
- 收多个操作数
- 定义位置与执行次数
- AArch64 指令宏库
- 第 6 章: 集合与文本
 - 四种集合类型
 - 字符串
 - 分割与连接
 - 字节序列
 - 拼接字节序列
 - 列表
 - 改写 let 绑定的集合
 - 元素不限于整数
 - 映射
 - 遍历映射
 - 在函数中组合集合
- 第 7 章: 词法单元与模式匹配
 - 词法单元保留源码结构
 - 匹配词法单元结构
 - 字面量模式词法单元
 - 捕获种类
 - 配平的词法单元范围(类似括号平衡)
 - 最小匹配与回溯
 - 空词法单元范围
 - 匹配已有词法单元列表
 - 尝试不同结构
 - 失配与无效模式
- 第 8 章: 目标、指令与标号
 - 目标
 - 模式调用
 - 查询当前目标

- 书写指令
- 在指令中使用编译期值
- 静态标号
- 符号引用与地址回填
- 动态指令文本与动态标号
- 读取标号地址
- 第 9 章：数据与二进制布局
 - 写出数据
 - 浮点值
 - 字符串与字节序列
 - 预留空间
 - 填充与对齐
 - 声明结构体
 - 字段默认值与结构体字面量
 - 测量布局
 - 在 x86-64 栈上使用结构体布局
 - 打包并写出结构体值
 - 联合体与当前字段
 - 何时使用结构体
- 第 10 章：模块与文件
 - 导入源模块
 - 内联包含源文件
 - 源文件路径解析
 - import 与 include
 - 检查与读取数据文件
 - 列出目录
 - 读取字节范围
 - 加载 JSON
 - 加载 TOML
 - 配置数据与二进制数据
 - 模块组织
- 第 11 章：输出区域与虚拟数据
 - RVA 与 FOA
 - 用 print 看中间值
 - origin 只改逻辑地址
 - region.begin 指定 RVA 和 FOA
 - reserve 与尾部预留
 - output.org 保留尾部预留

- output.section 裁掉尾部预留
- region.file_align 只对齐 raw size
- 虚拟区域
- region.place 把虚拟内容落进文件
- load.* / store.* 只在当前活动区域
- 最终区域信息只在收尾阶段查询
- 第 12 章：收尾处理
 - defer 回填已经存在的字段
 - 读取和改写最终字节
 - 计算校验和
 - 查询最终区域信息
 - 用过程登记回填
 - defer 里能写什么
 - 多个 defer 的执行顺序
 - late_layout 封存前新增真实布局
 - late_layout 会影响尾部预留
 - late_layout 里能写什么
 - 三个阶段的分工
 - 第三部分：构建程序
- 第 13 章：Flat Binary 与自定义文件格式
 - 原始机器码
 - 文件头加 Payload
 - 用 packed struct 描述记录
 - 用 defer 回填文件头
 - 明确字段要的是 RVA 还是 FOA
 - 表示文件间隙和 file-free 尾部
 - late_layout：晚生成但仍参与布局
 - 用断言保护自定义格式
 - 推荐组织顺序
 - 什么时候用格式接口
- 第 14 章：可执行文件与目标文件格式
 - 声明映像
 - 完整示例：ELF64 程序
 - section 和 segment 怎么选
 - 格式族
 - 入口点绑定
 - 生成的格式内容
 - 从 format.inc 开始

- 后续阅读
- 第 15 章：诊断与实用惯例
 - 阅读诊断信息
 - 信息与警告
 - 主动报错
 - 断言
 - 保护目标相关源码
 - 用名字替代魔数
 - 在名称中写清坐标
 - 推导布局事实
 - 保持收尾处理聚焦
 - 按用途分模块
 - 从 format.inc 开始
 - 在输出边界检查
 - 找到合适的文档
 - 最终检查清单

第 1 章：认识 XIRASM

指令与编译期代码

XIRASM 源文件混写两类行：**处理器指令**用所选指令集的汇编语法，**编译期代码**在汇编过程中计算值、决定输出内容。

```
x86.use64()                // 编译期调用：选定 64 位 x86 编码

const answer: u32 = 40 + 2  // 汇编时计算

entry:                     // 标号记录当前地址，不写出字节
    mov eax, answer        // 指令：`answer` 成为立即操作数
    ret
```

一行属于哪种形态由这一行本身决定，与文件无关：

这一行要做什么	写法
编码一条处理器指令	汇编指令行
在汇编期间计算一个值	编译期表达式

这一行要做什么	写法
重复或选择要生成的内容	编译期控制流
向输出写入字节	<code>db</code> 、 <code>dw</code> 、 <code>dd</code> 、 <code>dq</code> 等数据调用
描述文件格式结构	<code>format.inc</code> 调用

汇编第一个程序

保存为 `hello.xir` 并运行：

```
xirasm hello.xir
```

命令行只打印一行摘要，字节写入 `hello.bin`：

```
assembled output (6 bytes, 2 instruction fragments, target x86-64)
```

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

前五个字节编码 `mov eax, 42`，最后一个字节编码 `ret`。标号不写出任何字节。

这样汇编得到的输出是 flat binary（平坦二进制）：源码写出的字节本身，不带操作系统文件头。格式接口在第 14 章介绍，PE、COFF 和 ELF 的完整内容见[格式教程](#)。

编译期代码向输出写入字节

编译期代码在 XIRASM 构造输出时运行，不是生成出来的程序里运行的代码。下面的循环每轮写一个字节：

```
const count: u8 = 4

for value in range(0, count) { // 0 <= value < count, 汇编期间展开
    db(value)                  // 每轮一个字节
}
```

源码规则

一行放一条语句，行到行尾就结束。同一行写两个调用是错误，保存为 `bad.xir` 会报：

```
db(1); db(2)
```

```
bad.xir:1:1: error: a line holds one statement: this call is followed by more text,  
so write the rest on its own line (TrailingTextAfterCall)
```

；结束的是一条编译期语句。它后面不能再有别的内容，也不是必须写，本指南一律不写：

```
db(1);    // 可以写  
db(2)     // 同一个语句，没写
```

；不结束指令，所以指令行末尾不能带分号：

```
nop;
```

```
bad.xir:1:1: error: ISA text does not end with a semicolon
```

行注释、标号和代码块：

- `//` 到行尾是注释，可出现在任何语句之后：指令、调用、声明、赋值、标号、代码块声明头、控制语句都一样；引号内的 `//` 仍是普通文本。
- 标号以 `:` 结尾。
- 代码块用 `{` 和 `}`。函数、`if`、`while`、`for` 的声明头在同一行以 `{` 结尾；孤立的 `else` 或不匹配的 `}` 是错误。
- 调用和复合值在括号或花括号未闭合时可以跨行书写。

编译期代码与指令在同一份文件里交错出现：

```
x86.use64()

fn emit_marker(value: u8) {    // 编译期函数：写出两字节标记
    db(0x7f, value)
}

const marker: u8 = 3
emit_marker(marker)           // 调用在汇编期间执行，不在运行时

start:                        // 从这里开始是指令
    mov eax, marker
    ret
```

两类内容写进同一份 8 字节输出。

[返回语言指南](#)

第 2 章：值与绑定

值只存在于汇编期间

XIRASM 的值都是编译期值。绑定只给值起名；只有指令、数据调用、布局操作或格式接口用到它时，它才进入输出。

```
const magic: u16 = 0x5a4d    // 给值起名，此时还没有写出任何字节
dw(magic)                   // 这一行才写出 `4d 5a`
```

常量

const 绑定一个不能重新赋值的名字：

```
const page_size: u64 = 4096    // 宽度是约定的一部分
const header_size = 64        // 由初始值推断
```

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

初始值能决定类型时可以省略类型注解；宽度属于二进制约定时要写出类型。给 `const` 重新赋值是错误：

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

```
bad.xir:2:1: error: the declaration syntax is not valid (InvalidValueDeclaration)
```

可变绑定

`let` 绑定的名字可以在汇编期间更新：

```
let offset: u32 = 0
offset = offset + 16 // 赋值在汇编期间执行
offset = offset + 4

dd(offset)          // 20 = 0x14
```

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

声明和赋值都是编译期工作，输出里没有对应的运行时变量。只在确实要更新这个名字时才用 `let`。

类型注解与类型推断

```
const signature: u16 = 0x5a4d // 显式写出：它是文件格式字段
const title: string = "XIRASM"
const marker: bytes = b"OK"
const enabled: bool = true
```

```
const count = 4           // 推断为 `integer`
const name = "payload"    // 推断为 `string`
const raw = b"DATA"       // 推断为 `bytes`
```

常用的值类型：

类型	用途	示例
<code>integer</code>	通用编译期整数	<code>42</code>
<code>u8</code> 、 <code>u16</code> 、 <code>u32</code> 、 <code>u64</code>	对应宽度的无符号值	<code>0xff</code> 、 <code>0x5a4d</code> 、 <code>0x401000</code> 、 <code>0x140000000</code>
<code>i8</code> 、 <code>i16</code> 、 <code>i32</code> 、 <code>i64</code>	对应宽度的有符号值	<code>-1</code> 、 <code>-200</code> 、 <code>-4096</code> 、 <code>-0x100000000</code>
<code>usize</code>	与宿主地址同宽的无符号值	<code>16</code>
<code>f32</code>	IEEE-754 32 位浮点数	<code>f32(1.5)</code>
<code>f64</code>	IEEE-754 64 位浮点数	<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>	按字面取值的字节序列	<code>b"PE"</code>

列表、映射、结构体和类型值在后面的章节介绍。

固定宽度整数用于锁定二进制字段或函数参数；`integer` 用于只关心"是整数"的场合，写出几个字节由调用的输出接口决定。有符号值使用补码，声明时就会检查取值范围：

```
const offset: i32 = -1
const too_big: i8 = 200      // 200 放不进 i8
```

```
bad.xir:2:1: error: the declaration syntax is not valid (InvalidValueDeclaration)
```

数据调用接收无符号值，所以负数要写成它的补码位型：

```
dd(0xffffffff) // -1 的四个字节
```

有符号绑定不会被自动转换：

```
const offset: i32 = -1  
  
dd(offset)
```

```
bad.xir:3:1: error: lowering failed: InvalidApiInteger
```

带小数部分或指数的十进制字面量类型是 `f64`；`f32(value)` 显式窄化，`f64(value)` 显式扩展。整数和浮点数之间不会隐式转换：

```
emit.f64(1.5) // 带小数的字面量本身就是 f64  
emit.f32(f32(1.5)) // f32(...) 显式窄化
```

```
emit.f64(1) // 1 是整数
```

```
bad.xir:1:1: error: a call argument does not match what the call expects  
(InvalidApiArgument)
```

字符串与字节序列

`string` 是文本，`bytes` 是按字面取值的字节序列：

```
const section_name: string = ".text"  
const signature: bytes = b"MZ"  
  
db(section_name) // 2e 74 65 78 74  
db(signature) // 4d 5a
```

文本名称、路径、生成的指令文本，以及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`。有些数据接口两者都接受：

```
db("AB", b"CD")
```

```
41 42 43 44
```

转义序列

引号字面量使用同一套转义，单引号双引号、带不带 `b` 前缀都一样：

转义	字节
<code>\n</code>	换行， <code>0x0a</code>
<code>\r</code>	回车， <code>0x0d</code>
<code>\t</code>	制表符， <code>0x09</code>
<code>\0</code>	NUL， <code>0x00</code>
<code>\\</code>	一个反斜杠
<code>\"</code> （在 <code>"..."</code> 中）、 <code>\'</code> （在 <code>'...'</code> 中）	开启该字面量的那个引号
<code>\uXXXX</code>	该码位的 UTF-8 字节

```
db("\u0041") // 41: 恰好四位十六进制数字对应一个码位
db("\u00e9") // c3 a9: 它的 UTF-8 字节
db("a"b") // 61 22 62: 把开引号写两遍等于一个引号
db("a\tb") // 61 09 62
db(b"A\tB") // 41 09 42: `b` 前缀解码同一套转义
db('a\tb') // 61 09 62: 单引号也一样
```

`\uXXXX` 的用途是读回生成的平台文本：Windows API 表用 `"\u0000"` 表示它要的 NUL 字节。除此之外的反斜杠一律原样保留，所以 `\u41` 和 `\ud800` 就是写下的那几个字符：

```
db("\u41") // 位数不够
db("\ud800") // 孤立代理项不是码位
```

要把反斜杠当作数据，就得写成两个：

```
db("a\\tb")    // 61 5c 74 62: 反斜杠是数据
db("a\tb")     // 61 09 62: 反斜杠是转义
```

块作用域

代码块会创建嵌套作用域：

```
const value = 1

{
  let value = 2    // 在块内遮蔽外层的 `value`
  db(value)        // 02
}

db(value)          // 01: 外层绑定重新可见
```

[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 XIRASM 汇编源码时计算出值。声明、赋值、函数参数、`return` 语句、条件、断言、指令操作数和数据调用里都可以写表达式。

```
const entry_count = 3
const bytes_per_entry = 8
const table_size = entry_count * bytes_per_entry

dd(table_size)          // 24 = 0x18
```

算术运算

运算符	含义
+	加法
-	减法
*	乘法
/	整数除法
%	取余

```
const total = 2 + 3 * 4      // 14
const grouped = (2 + 3) * 4  // 20
const quotient = 17 / 5      // 3
const remainder = 17 % 5     // 2

dd(total, grouped, quotient, remainder)
```

加、减、乘都做检查：结果超出整数范围就报错，不会静默改变布局。

```
dq(0xfffffffffffffffffff + 1)
```

```
bad.xir:1:1: error: an expression in this statement is not valid
(InvalidExpression)
```

```
dd(1 / 0)
```

```
bad.xir:1:1: error: the expression divides by zero (DivisionByZero)
```

一元 + 不改变值；一元 - 产生 64 位补码值，负数就是靠它写出去的：

```
const all_bits = -1

dq(all_bits)           // 八个 `ff` 字节
```

位运算与移位

位运算用于描述权限、掩码、指令字段和文件格式标志：

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

```
const readable = 1 << 0
const writeable = 1 << 1
const executable = 1 << 2

const permissions = readable | executable

assert((permissions & executable) != 0)
assert((permissions & writeable) == 0)
db(permissions)           // 1 | 4 = 5
```

移位量达到或超过值的宽度时结果是零，不是错误：

```
dq(1 << 63)           // 00 00 00 00 00 00 00 80
dq(1 << 64)           // 已经没有位可以移进来
```

比较与相等

运算符	含义
==	等于
!=	不等于
<	小于

运算符	含义
<=	小于等于
>	大于
>=	大于等于

大小比较用于整数；相等判断还能用于兼容的非整数类型：

```
const payload_size = 96
const maximum_size = 128

assert(payload_size > 0 && payload_size <= maximum_size)

const format_name = "raw"
const signature = b"OK"

assert(format_name == "raw")
assert(signature == b"OK")
```

不属于同一类的值会被拒绝，而不是被转换：

```
const total = "x" + 1
```

```
bad.xir:1:1: error: an expression in this statement is not valid
(InvalidExpression)
```

布尔逻辑与短路

运算符	含义
!	逻辑非
&&	逻辑与
	逻辑或

&& 和 || 在结果已经确定时立即停止。`left && right` 在 `left` 为假时不计算 `right`；`left || right` 在 `left` 为真时不计算 `right`：

```
const enabled = true
const safe = enabled || (1 / 0 == 0)    // 右边的除法根本不会执行

assert(safe)
```

只有当前面的条件成立、后面的操作数才合法时，短路正是让这种写法成立的原因。

表达式中的函数调用

有返回值的调用可以出现在任何接受其结果类型的位置：

```
const name_length = lengthof("XIRASM")    // 6
const aligned_size = ((37 + 15) / 16) * 16 // 48 = 0x30

db(name_length)
dd(aligned_size)
```

调用可以嵌套，内层先计算完再交给外层：

```
const normalized = upper(trim("  kernel  "))

assert(normalized == "KERNEL")
```

只写出内容或执行汇编动作的过程按语句调用，不产生值。两类函数都在第 5 章介绍。

字段访问

. 选择一个命名字段。`target.bits` 和 `target.isa` 读取当前生效的指令集，第 4 章和第 8 章的条件里会用到；结构体字段在第 9 章介绍。

```
const before = target.bits    // 64: 此刻读到的值

x86.use32()

db(before)                    // 40: 绑定保留了读到的那个值
db(target.bits)               // 20: 再读一次看到的是当前目标
```

运算符优先级

同一行的运算符优先级相同；越靠上的行绑定越紧，同优先级从左到右结合。

优先级	运算符
最高	函数调用、括号表达式、字段访问
一元	<code>+</code> 、 <code>-</code> 、 <code>~</code> 、 <code>!</code>
乘法	<code>*</code> 、 <code>/</code> 、 <code>%</code>
移位	<code><<</code> 、 <code>>></code>
加法	<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>

这不是 C 的那张表。移位比加减法绑定更紧：

```
const shift_first = 1 + 2 << 3      // 1 + (2 << 3) = 17
const grouped_shift = (1 + 2) << 3  // 24 = 0x18

dd(shift_first, grouped_shift)
```

表达式同时混用移位、算术、掩码或比较时，加上括号。括号把计算顺序直接写出来，读者就不必回头查上面这张表。

表达式错误

计算不出明确的编译期值时，XIRASM 停止汇编而不是给出零——一个静默的零会破坏指令操作数或二进制布局。上面已经出现除零、溢出和类型不匹配；名字未定义同样失

败：

```
dd(missing + 1)
```

```
bad.xir:1:1: error: the name is not defined where it is used (UndefinedSymbol)
```

[返回语言指南](#)

第 4 章：控制流

控制流在汇编期间执行

XIRASM 的控制流决定构造输出时要执行哪些源码操作。它不会自动生成运行时的分支或循环。

```
const debug_build = false

if debug_build {
    db("DEBUG")
} else {
    db("RELEASE")
}
```

只有选中的分支会写出字节。运行时分支仍然用普通指令写：x86 的 `jmp`、`call` 和条件跳转。

if 与 else

`if` 的条件必须产生布尔值。选中的代码块在自己的作用域中执行，未选中的代码块什么都不产生：

```
const payload_size = 32

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

```
    db(2)          // 走这个分支
} else {
    db(3)
}
```

`else` 可以省略：

```
const include_marker = true

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

条件生成的指令

`if` 代码块里可以放处理器指令，因此同一份源码能为不同配置生成不同指令：

```
x86.use64()

const return_zero = true

entry:
    if return_zero {
        xor eax, eax
    } else {
        mov eax, 1
    }
    ret
```

没被选中的 `mov` 不会进入输出。这是编译期选指令，不是运行时分支。

目标条件

`target.isa` 是所选指令集族，`target.bits` 是当前位宽。`x86.use32()` 这类模式调用会改变后面的条件看到的值：

```
x86.use32()

if target.isa == .x86_64 {
    db(0x10)          // 指令集族仍是 x86
}
```

```
if target.bits == 32 {  
    db(0x00)          // 但位宽已经变成 32  
}
```

目标查询有专门的条件语法，直接写在 `if` 条件里。

for 与 range

迭代次数已知时用 `range(start, end)`。起点包含在内，终点排除在外：

```
for index in range(0, 4) {  
    db(index)  
}
```

倒序的范围会被拒绝：

```
for index in range(4, 0) {  
    db(index)  
}
```

```
bad.xir:1:1: error: lowering failed: InvalidMetaFor
```

循环绑定属于循环体，而且每一轮都重新绑定一次：

```
for index in range(0, 3) {  
    const encoded = index + 0x10  
    db(encoded)  
}
```

for 与列表

`for` 也按列表顺序遍历编译期列表的值：

```
const opcodes: list = list.of(0x90, 0x90, 0xc3)

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

列表和映射在第 6 章介绍。

while

终止条件依赖循环自己更新的值时用 **while**：

```
let value = 1

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

每轮开始前先判断条件，所以一开始条件就为假的循环体一次也不执行。无法终止的编译期循环会让整个构建停住，因此循环超过 1,000,000 次就直接停止：

```
for index in range(0, 1000001) {
    db(0)
}
```

```
bad.xir:1:1: error: the loop ran past the supported iteration limit
(MetaLoopLimitExceeded)
```

break 与 continue

break 结束最内层循环，**continue** 跳过本轮剩下的部分。两者在 **for**、**while** 和嵌套的 **if** 里都能用：

```
for value in range(0, 8) {
    if value == 6 {
```

```
        break
    }
    if (value & 1) != 0 {
        continue
    }
    db(value)          // 6 以下的偶数
}
```

两者都不会跨过函数调用边界：

```
fn stop() {
    break
}

for index in range(0, 3) {
    stop()
}
```

```
bad.xir:2:5: error: break used outside of a Meta loop
```

[返回语言指南](#)

第 5 章：函数与作用域

函数的两种形式

函数是命名的编译期代码块，供后续调用。它有两种形式：

- **过程**执行动作，不产生值；
- **返回值函数**用 `->` 声明结果类型。

两者都用 `fn`、参数和代码块。函数本身不写出字节，字节在它被调用、或者返回值交给写输出的接口时才产生。

过程

过程没有 `->`，调用它就是一条语句：

```
fn emit_pair(value: u8) {
    db(value)          // 函数体里可以放任何调用处能放的东西
    db(value + 1)
}

emit_pair(2)
emit_pair(8)
```

过程不是值，它的用途是执行生成数据的逻辑。把它赋给绑定会被拒绝：

```
// 过程没有返回值，所以不能写进 const。
fn emit_marker() {
    db(0x90)
}

const marker = emit_marker()
```

```
bad.xir:5:1: error: undefined name in this expression: emit_marker()
```

参数与实参

参数按位置接收实参，没有默认值：

```
fn emit_run(value: u8, count: u64) {
    for index in range(0, count) {
        db(value)      // 每轮一个字节
    }
}

emit_run(0xcc, 4)
```

每次调用有自己的一套参数绑定，互不影响。参数的类型注解可以省略：

```
fn add(left, right) -> u64 {
    return left + right
}

db(add(20, 22))      // 42 = 0x2a
```

建议在函数上方写上注释，说明每个参数是什么：没有类型标注时，隔一段时间就看不懂这段逻辑了。

let 参数

参数默认只读；加 **let** 前缀后传的是调用者的绑定本身，而不是它的值：

```
// plan 带 let: 函数改的是调用者那个 map 本身。
fn set_entry(let plan: map, address: u64) {
    map.set_mut(plan, "entry", address)
}

let image: map = map.new()
set_entry(image, 0x401000)

// 回到调用者这边看，改动确实落在 image 上。
assert(map.get(image, "entry") == 0x401000)
```

实参必须是普通的 **let** 绑定：

```
fn set_entry(let plan: map, address: u64) {
    map.set_mut(plan, "entry", address)
}

// image 是 const, 不能交给 let 参数改写。
const image: map = map.new()
set_entry(image, 0x401000)
```

```
bad.xir:6:1: error: mutable function argument must resolve to a let binding
```

字面量和临时表达式都不行：

```
fn touch(let a: u64) {
    db(0)
}

// 5 是字面量，没有绑定可改。
touch(5)
```

```
bad.xir:5:1: error: mutable function argument must be a direct let binding
```

同一次调用不能把同一个绑定传给两个 **let** 参数：

```
fn touch(let a: u64, let b: u64) {  
    db(0)  
}  
  
// 同一个 shared 传给两个 let 参数，两处会互相覆盖。  
let shared = 1  
touch(shared, shared)
```

```
bad.xir:6:1: error: mutable function arguments cannot alias the same binding
```

let 参数要靠语句调用写回调用者，因此返回值函数不能带 **let** 参数：

```
// 带 -> 的声明不能配 let 参数：写回调用者要靠语句调用，而这里返回的是值。  
fn bump(let value: u64) -> u64 {  
    return value + 1  
}
```

```
bad.xir:1:1: error: the function declaration or call is not valid  
(InvalidMetaFunction)
```

返回值函数

写 **->** 类型后，函数产生一个表达式值：

```
fn align_up(value: u64, alignment: u64) -> u64 {  
    return ((value + alignment - 1) / alignment) * alignment  
}  
  
const header_size = align_up(0x73, 0x20)  
dw(header_size)      // 0x80
```

结果会转换到声明的返回类型，返回类型可以是任何普通的编译期值：

```
// 返回类型可以是 bool, 也可以是 bytes 这类非整数值。
fn is_page(value: u64) -> bool {
    return value == 0x1000
}

fn signature() -> bytes {
    return b"XR"
}

assert(is_page(0x1000))
db(signature())          // 这里写出: 58 52
```

凡是能接受该结果类型的表达式位置，都可以使用返回值调用：声明、实参、条件、另一个调用，或者更大的表达式。

返回规则

返回值函数必须走到 **return**：

```
// 算了一个值却没写 return, 所以函数没有结果可交。
fn incomplete(value: u64) -> u64 {
    const doubled = value * 2
}

const result = incomplete(4)
```

```
bad.xir:1:1: error: function incomplete declares a return value, but its body ended
without a return statement
```

返回的值必须满足声明的类型：

```
// 声明说要 bool, 这里返回的是整数字面量 1。
fn enabled() -> bool {
    return 1
}

const result = enabled()
```

```
bad.xir:2:5: error: function enabled declares a bool return value, but this return
```

```
statement produces a integer
```

返回值函数是计算用的辅助工具：可以用局部绑定、控制流和别的返回值调用，但不能写出输出，也不能改变布局：

```
// 声明了 -> 却写出输出：这条限制把两种形式分开了。
fn bad_counter() -> u64 {
    db(1)
    return 1
}

const result = bad_counter()
```

```
bad.xir:2:5: error: a value-returning function cannot emit instructions or change
layout; declare it without a return type to run it as a procedure
(SideEffectInValueFunction)
```

正是这条限制划分了两种形式：需要改变输出或布局时用过程，调用者需要值时用返回值函数。过程不声明返回类型，也不能返回值：

```
// 过程不声明返回类型，所以它不能 return 一个值。
fn emit_one() {
    return 1
}

emit_one()
```

```
bad.xir:2:5: error: the function declaration or call is not valid
(InvalidMetaFunction)
```

函数局部作用域

参数和函数体内声明的绑定都属于这一次调用(每次都独立)：

```
fn adjusted_size(size: u64) -> u64 {
    const overhead = 4
    let result = size + overhead
```

```
    if result < 16 {
        result = 16
    }

    return result
}

db(adjusted_size(3))    // 16
db(adjusted_size(20))   // 24
```

每次调用都会新建 `size`、`overhead` 和 `result`，它们都不会在这次调用之后留下。实参在参数绑定存在之前、在调用者的作用域里求值，所以 `pair(b, a)` 不会因为某个参数名叫 `a` 而改变第二个实参的读法。

函数体内的代码块会嵌套，可以遮蔽外层名字，而外层作用域仍然可写：

```
fn combine(value: u64) -> u64 {
    let result = value

    {
        const value = 5    // 遮蔽参数
        result = result + value
    }

    return result
}

db(combine(3))            // 3 + 5
```

声明顺序

函数必须声明在第一条调用它的语句之前：

```
// add 在后面才声明，这里先用它。
const answer = add(20, 22)

fn add(left: u64, right: u64) -> u64 {
    return left + right
}

db(answer)
```

```
bad.xir:1:1: error: undefined name in this expression: add(20, 22)
```

声明都在顶层。嵌在函数体里的声明会在该函数体被 lowering 时报错：

```
// 函数里再声明函数：外层被调用时就会报错。
fn outer() {
  fn inner() {
    db(1)
  }
  inner()
}

outer()
```

```
bad.xir:2:5: error: the function declaration or call is not valid
(InvalidMetaFunction)
```

第 10 章会讲包含文件如何把声明提供给另一个源文件。

递归与调用深度

计算有明显基准情况时，返回值函数可以调用自己：

```
fn triangular(value: u64) -> u64 {
  if value == 0 {
    return 0
  }

  return value + triangular(value - 1)
}

db(triangular(4))      // 4 + 3 + 2 + 1
```

递归同样在汇编期间执行，所以必须保持浅。调用链达到 128 层会被拒绝：

```
// 递归到 128 层，超过调用深度上限。
fn down(n: u64) -> u64 {
  if n == 0 {
    return 0
  }
}
```

```
    return down(n - 1) + 1
}

db(down(128))
```

```
bad.xir:5:5: error: an expression in this statement is not valid
(InvalidExpression)
```

指令形式的宏

宏把一行源码起个名字。调用它就照处理器的写法写：行首写宏名，后面跟操作数：

```
macro twice(dst, src) {
    mov dst, src
    mov dst, src
}

x86.use64()
twice eax, ebx           // 这里写出：89 d8 89 d8
```

前端先在行首认出助记符 `twice`，它是宏名，于是把**整行**交给宏体处理，编码器根本看不到这一行。宏体里那两行 `mov dst, src` 才是真正交给编码器的指令。

替换只发生在**指令操作数**位置，而且只看位置、不看内容：

- 替换进去的文本不会被再看一遍，所以它里面即使写着另一个参数名，也不会被当成参数再替换一次；
- 引号只是操作数里的一部分字符。写 `spell "reg"` 时，参数接收的是 `"reg"` 这五个字符（含引号），往里填的永远是整个操作数；
- 行首的助记符和参数替换无关：参数名恰好和某条指令同名，也不会把那一行改掉。

宏和函数在参数上不一样。函数调用时，实参先计算出值再传进去；宏调用时，参数接收的是**调用处写的那段源码文本**：

调用处写的

参数接收的

`twice eax, ebx` 里的 `eax` 文本 `eax`

`twice eax, ebx` 里的 `ebx` 文本 `ebx`

上面这个宏只把文本填进指令，文本本身是什么意思，这个宏并不关心。要计算出一个值，就得让参数把文本当表达式求值——这需要另一种东西来装它，就是 `operand`。

operand：捕获的源码

`operand` 是一个值类型，装的是一段源码文本，外加写下这段文本时能看见的绑定。带上绑定是为了之后再求值：文本里的名字要按调用处的定义去读，不能按宏体里的定义读。

`operand` 有两个取法：

- `operand.text(value)` 取原样的拼写，不求值；
- `operand.eval(value)` 把那段文本当 Meta 表达式求值，得到它计算出的值。

要自己实现一条指令的编码，就用 `operand.eval` 把参数计算出值：

```
macro byte(value) {  
    const n: u64 = operand.eval(value)    // 按调用处的绑定求值  
    emit.u8(n)  
}  
  
const LIMIT: u64 = 7  
byte LIMIT + 1           // 这里写出: 08  
byte (LIMIT + 2)         // 这里写出: 09
```

第一次调用里，`value` 接收的文本是 `LIMIT + 1`，按调用处的 `LIMIT = 7` 求值得到 8。第二次调用写的是 `byte (LIMIT + 2)`，参数文本是 `(LIMIT + 2)`——注意外层那对括号是 Meta 的普通括号，参与运算，所以求值得到 9。写 `byte LIMIT + 2` 不带括号也一样，逗号才是操作数的分界。

`emit.u8(n)` 把一个字节写进当前输出。写输出的接口分两族：

- `db`、`dw`、`dd`、`dq` 把宽度写在助记符上，收整数并按宽度检查范围；其中 `db` 还收字符串和字节序列，`db("AB")` 写出 `41 42`，而 `dw` 及以上只收整数。
- `emit` 族把宽度写成类型：`emit.u8`、`emit.u16`、`emit.u32`、`emit.u64` 只收整数并按宽度检查范围，`emit.u8(0x1234)` 报 `InvalidApiInteger`；`emit.bytes` 收字节序列或字符串；`emit.f32`、`emit.f64` 收浮点数。没有有符号的 `emit.i*`。

宏的参数没有类型注解，也没有默认值。写 `byte(7)` 会报 `macro calls use instruction syntax without parentheses`，因为那样写像函数调用；要带括号就得写成 `byte (7)`，中间留一个空格。末尾的参数写成 `...name`，它把剩下的操作数全部收成一个列表。同名可以同时有一个固定参数个数的版本和一个 `...name` 版本，个数对得上时用固定版本；对不上则报 `MacroArityMismatch`。

AArch64 指令库 (`arm/a64-macros.inc`) 就是这样建起来的：那里的指令能按 AArch64 汇编文本书写，是因为每一条都被写成了宏。要给自己的一套编码规则做成指令的样子，做法相同。

`operand.eval` 求值用的是**捕获那段文本时可见的绑定**，不是宏体里的绑定：

```
macro emit_one(value) {
    const n: u64 = operand.eval(value)
    emit.u8(n)
}

const LIMIT: u64 = 7
emit_one LIMIT + 1      // 这里写出: 08
```

宏体里再声明一个同名 `const` 也改不了它。捕获的文本带着调用处的作用域，宏体的名字不在其中：

```
macro shadowed(value) {
    const LIMIT: u64 = 100           // 对捕获的表达式不可见
    const n: u64 = operand.eval(value)
    emit.u8(n)
}

const LIMIT: u64 = 7
shadowed LIMIT + 1      // 这里写出: 08, 宏体里的 LIMIT 改不了它
```

这条调用同样写出 `08`。同一个道理让宏可以安全地复用调用处的名字：宏体里出现什么名字，都不会改变调用处那段文本的读法。

不要求值、只要原本的拼写时，用 `operand.text`：

```
macro spell(value) {
    emit.bytes(operand.text(value))
}

spell 1 + 2      // 这里写出: 31 20 2b 20 32
```

写出 31 20 2b 20 32，也就是 "1 + 2" 这五个字符。

在指令操作数里写参数

指令操作数位置需要的是文本，所以在那里直接写参数名，由宏把它替换成捕获的文本：

```
macro load(reg, source) {
    mov reg, source
}

x86.use64()
const OFFSET: u64 = 7
load rax, OFFSET          // 编码器收到的是 `mov rax, 7`
```

编码器收到的就是上面注释里那句文本。反过来，`operand.eval` 产出的是值，只能用在 API 实参和 `const` 初始化这类需要值的地方；把它写在操作数位置上，那里已经没有东西可供求值：

```
```asm id=05-bad error=macro operand text operand.eval(7) cannot be encoded
as an operand; bind it to a const first, as in const value: u64 =
operand.eval(text) (UnresolvedFixup) // 操作数位置要的是文本，operand.eval 产
出的是值 —— 位置不对。 macro bad(value) { mov rax, operand.eval(value) }
```

x86.use64() bad 7

```
```text
bad.xir:2:5: error: macro operand text operand.eval(7) cannot be encoded as an
operand; bind it to a const first, as in `const value: u64 = operand.eval(text)`
(UnresolvedFixup)
```

收多个操作数

参数可以只收一个操作数，也可以把末尾的 `...name` 写成变参，把剩余操作数收成一个列表：

```
macro bytes(...values) {
    for item in values {
        emit.u8(operand.eval(item))
    }
}

bytes 1, 2, 3           // 这里写出: 01 02 03
```

`operand` 上还有两个辅助函数:

- `operand.slice(value, start, end)` 取捕获文本的一段字节范围，并检查越界；
- `operand.split(value)` 在顶层逗号处切分，引号和配对的 `()[]{}` 原样保留，所以 `(a, b)` 和 `"a,b"` 都算作一整段。

两者都保留每一段的捕获绑定:

```
macro first_three(arg) {
    const head: operand = operand.slice(arg, 0, 3)
    emit.bytes(operand.text(head))
}

first_three abcdef       // 这里写出: 61 62 63
```

写出 `61 62 63`。宏之间可以互相调用:

```
macro byte(value) {
    const n: u64 = operand.eval(value)
    emit.u8(n)
}

macro twice(dst, src) {
    mov dst, src
    mov dst, src
}

macro bytes(...values) {
    for item in values {
        byte item
    }
}

x86.use64()
twice eax, ebx           // 这里写出: 89 d8 89 d8
bytes 1, 2, 3            // 这里写出: 01 02 03
```

写出 `89 d8 89 d8` 和 `01 02 03`。

绑定到 `operand` 的标识符在转发时带着它原本的捕获，循环和辅助函数里的操作数局部变量同样如此；混合表达式的每个片段各自保留自己的绑定。由 `operand.eval` 调用的函数使用第一个片段的捕获作用域，`here()` 这类位置内建函数使用该片段的捕获位置，每次 `operand.eval` 都重新求值一次。

定义位置与执行次数

宏在顶层定义，并且要先定义后使用，包括经过 `import` 引入的情况。名字区分大小写，可以带点分隔的段。

宏体使用普通的多行代码块语法，可以包含 Meta 变量、控制流、过程、指令和收尾块注册。宏体不返回值；`break` 和 `continue` 不能跳进调用者的循环；已保存的 `defer` 和 `late_layout` 体内不能定义或调用宏；宏也不能绕过返回值函数的副作用限制。

每次真正到达的调用在 lowering 期间只执行一次，布局、松弛、编码和收尾都不会重跑宏体。宏与函数共用 128 层调用深度，宏调用另有每个 lowering 上下文累计 100,000 次的上限；一次调用最多 256 个操作数、64 层操作数分隔符；捕获环境最多链到 128 层，超出报告 `MacroCaptureDepthExceeded`。捕获会复制当时可见的值绑定，把操作数和大量集合长期放在一起有内存代价；不再需要原始语法时，把它求值成普通值存起来。

宏里的静态标号仍是模块标号。要私有标号，用 `sym.unique` 造名字、用 `label.define` 定义。`isa(text)` 绕过宏查找直接提交原生指令。

AArch64 指令宏库

导入 `arm/a64-macros.inc` 之后，AArch64 指令按汇编文本直接写，由宏库负责编码：

```
// 导入之后，这些助记符就都按宏解析，直接写汇编文本即可。
import("arm/a64-macros.inc")
const OFFSET: u64 = 16
fadd v0.4s, v1.4s, v2.4s           // 浮点 SIMD 加法
movi v3.4s, #255, lsl #8           // 立即数：按 lane 移位铺开
ldr x2, [sp, #OFFSET]              // 基址 + 编译期算出的字节偏移
ld2 {v31.s, v0.s}[3], [sp], x4     // 结构访存：一次取两个 lane，带写回
ldr x5, data                        // 标号作为地址操作数
```

```
data:
emit.u64(0)
```

库由生成器产出：`include/arm/a64/generated/manifest.json` 里登记了 **778 个宏助记符**、**4360 条指令形式**，覆盖浮点与 Advanced SIMD、lane/copy/table、立即数与转换、访存与结构访存、标量整数、分支和系统指令。

分支支持延后解析标签，条件分支写成 `b.eq target` 这种形式；`adrp` 计算目标页与指令所在页之间的差值。系统寄存器接受 `nzcv` 这类固定名称，也接受 `S3_3_C4_C2_0` 这种通用拼写。

加密与扩展指令同样在库内，例如 `aese/aesd` (AES)、`sha256h` (SHA)、`crc32b` (CRC)、`ldadd/cas/swp` (LSE 原子操作)、`sdot/udot` (点积)、`bfdot/bfcvtn` (BF16)、`smmla/ummla` (I8MM)、`paciza` (指针认证)、`bti` (BTI)、`irg/stg` (MTE)。点积的成组 lane 写成 `v2.4b[index]` 或 `v2.2h[index]`，对应直接描述符如 `a64_lane("v2.4b", index)`。

尚不包含 SVE/SME 和 A32/T32。

导入之后，同名助记符在整个 lowering 上下文里都归宏处理，不按原生 target 隔离。混合 ISA 源码可以导入 `arm/a64.inc` 改用直接 API，直接调用接收描述符列表：

```
import("arm/a64.inc")
a64_ldr(list.of(a64_reg("x0"), a64_mem("sp", 16))) // 无论原生 target 是什么，写出的都是 A64 编码
```

立即数表达式沿用 Meta 算术，捕获操作数保留调用者绑定。包装层先检查操作数形状，再求值并检查各指令的寄存器角色、范围、缩放、对齐和写回重叠约束。访存基址允许 X0-X30 或 SP；寄存器偏移的 UXTW/SXTW 使用 W 寄存器，LSL/SXTX 使用 X 寄存器。

literal load 的 `#expression` 表示字节位移；裸标号在布局后解析，捕获地址表达式可写为 `label_addr(data) + ADDEND`。直接调用使用 `a64_rel(displacement)`、`a64_target(name)` 或 `a64_address(address)`。延后目标通过收尾块解析并检查范围，不会重新执行宏；`_word` 函数接收已解析描述符并返回编码。

[返回语言指南](#)

第 6 章：集合与文本

四种集合类型

一个字节不够用时，编译期需要更大的容器。XIRASM 有四种：

类型	表示	典型用途
string	源码级文本	名称、路径、诊断信息、解析出的字段
bytes	精确二进制数据	签名、编码后的整数、二进制记录
list	有序的值序列	表格、重复项、有序描述符
map	以字符串为键的值	命名选项、记录、查找表

四种都是编译期值，可以存在绑定里、当实参传递，也可以直接交给写输出的接口：

```
const name: string = "kernel"
const magic: bytes = b"XR"
const sizes: list = list.of(1, 2, 3)

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

emit.bytes(name)           // 这里写出: 6b 65 72 6e 65 6c
emit.bytes(magic)          // 这里写出: 58 52
emit.u8(len(sizes))        // 这里写出: 03
```

写输出之前，它们不占文件里任何位置。上面四条声明本身不产生字节。

改集合有两种写法。一种是返回值形式，它不动传进来的那个集合，另计算出一个新集合返回：

```
const base: list = list.of(1, 2)
const grown: list = list.push(base, 3)

emit.u8(len(base))          // 这里写出: 02
emit.u8(len(grown))        // 这里写出: 03
```

另一种是语句形式，名字以 `_mut` 结尾，它直接改掉传进来的那个 `let` 绑定。第一个实参必须是 `let` 声明的标识符，`const` 和临时值都会被拒绝（见本章《改写 `let` 绑定的集合》）。

```
let items: list = list.of(1, 2)
list.push_mut(items, 3)

emit.u8(len(items))      // 这里写出: 03
```

把集合交给另一个绑定时会复制一份，两个绑定互不影响：

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

emit.u8(len(snapshot))   // 这里写出: 01
emit.u8(len(items))      // 这里写出: 02
```

字符串

```
const raw_name: string = " Kernel64 "
const name: string = lower(trim(raw_name))

assert(name == "kernel64")
assert(starts_with(name, "kernel"))
assert(ends_with(name, "64"))
assert(contains(name, "nel"))

emit.bytes(name)        // 这里写出: 6b 65 72 6e 65 6c 36 34
```

字符串接口：

调用	作用
<code>trim(text)</code>	去掉两端的空白
<code>lower(text)</code> 、 <code>upper(text)</code>	规范化 ASCII 字母的大小写
<code>starts_with</code> 、 <code>ends_with</code> 、 <code>contains</code>	判断一段文本
<code>replace(text, needle, replacement)</code>	替换匹配到的文本
<code>to_string(value)</code>	把值转成十进制文本： <code>to_string(0x1234)</code> 得到 <code>"4660"</code>

`len(text)`

长度

大小写转换按 ASCII 处理。名称、配置文本、路径和诊断信息用字符串；要求每个字节都原样保留时用 `bytes`。

分割与连接

```
const sections: list = split("text,data,bss", ",")
const path: string = join(sections, "/")

assert(len(sections) == 3)
assert(list.get(sections, 0) == "text")
assert(list.get(sections, 2) == "bss")
assert(path == "text/data/bss")

emit.bytes(path)           // 这里写出: 74 65 78 74 2f 64 61 74 61 2f 62 73 73
```

`split` 按分隔符把文本拆成字符串列表，`join` 把列表连回一段文本。下标从 0 开始。结构化的外部数据——文件、JSON、TOML——在第 10 章。

字节序列

```
const magic: bytes = bytes.from_hex("58495200")
const version: bytes = bytes.le(3, 2)
const header: bytes = bytes.concat(magic, version)

assert(bytes.eq(magic, bytes.from_hex("58495200")))
assert(bytes.hex(version) == "0300")

emit.bytes(header)         // 这里写出: 58 49 52 00 03 00
```

`bytes.from_hex` 读十六进制文本，`bytes.le(value, width)` 按指定字节数把小端整数编码出来，`bytes.concat` 连接两段序列，`bytes.eq` 比较两段序列，`bytes.hex` 渲染成小写十六进制文本。

拼接字节序列

```
const base: bytes = bytes.from_hex("414243")
const marked: bytes = bytes.insert(base, 1, b"-")      // 41 2d 42 43
const patched: bytes = bytes.replace(marked, 2, 1, b"Z")
const trailer: bytes = bytes.repeat(2, 0xff)
const result: bytes = bytes.concat(patched, trailer)

emit.bytes(result)      // 这里写出: 41 2d 5a 43 ff ff
```

调用	作用
<code>bytes.new()</code>	空序列
<code>bytes.push(value, byte)</code>	追加一个字节，返回新序列
<code>bytes.repeat(count, byte)</code>	重复同一个字节
<code>bytes.insert(value, index, addition)</code>	在下标处放进一段字节
<code>bytes.replace(value, index, count, replacement)</code>	换掉一段字节
<code>bytes.concat(left, right)</code>	连接两段序列

走完上面这些步骤，`base` 仍然是 `41 42 43`：每一步都产生一个新值，交给下一步。

列表

```
let items: list = list.of(1, 2, 3)
list.push_mut(items, 4)
list.set_mut(items, 1, 0xaa)
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)      // 这里写出: 01 aa 03 04
}
```

调用	作用
<code>list.new()</code>	空列表
<code>list.of(...)</code>	用给出的值建一个列表

调用	作用
<code>list.get(value, index)</code>	读一个元素
<code>list.slice(value, index, count)</code>	复制一段元素
<code>list.concat(left, right)</code>	连接两个列表
<code>list.eq(left, right)</code>	比较内容
<code>len(value)</code>	元素个数

顺序是列表的一部分，`for` 遍历列表才有意义。节描述、表格行和字节分块都依赖确定的次序。

改写 `let` 绑定的集合

语句形式的接口改写的是一个 `let` 绑定本身，不返回新值：

```
// 三个 _mut 调用改的都是左边的绑定本身。
let items: list = list.of(1, 2)
list.push_mut(items, 3)           // 末尾变 1, 2, 3
list.set_mut(items, 0, 4)         // 下标 0 换成 4 → 4, 2, 3

let options: map = map.new()
map.set_mut(options, "arch", "x64")
map.set_mut(options, "items", items) // 值可以是别的集合
```

第一个实参必须是 `let` 绑定的标识符，`const` 会被拒绝：

```
// items 是 const，没有可改写的绑定。
const items: list = list.of(1)
list.push_mut(items, 2)
```

```
bad.xir:2:1: error: cannot mutate a const collection binding
```

临时值同样会被拒绝：

```
// 前两行没问题，第三行的目标是临时值 list.of(1)。
let items: list = list.of(1)
```

```
list.push_mut(items, 2)
list.push_mut(list.of(1), 3)
```

```
bad.xir:3:1: error: collection mutation target must be a direct let binding
```

查找名字时用最近的那个绑定，所以块里重新 **let** 一个同名变量，会遮住外层的那个；块里改的是块自己的集合，退出块后外层不受影响。顶层的 **let** 在普通 lowering 期间可以改；返回值函数只能改自己的局部绑定。

三种改写各自做的事：

调用	做的事
<code>list.push_mut(value, item)</code>	在末尾追加一个元素
<code>list.set_mut(value, index, v)</code>	换掉已存在下标处的元素
<code>map.set_mut(value, key, v)</code>	写入一个字符串键；键已存在时改值，键的位置不变

这三种都是语句，不返回值，也不能写进表达式。它们还受块的性质限制：

```
// defer 在布局定下来之后运行，那时不能再长出新内容。
let items: list = list.of(1)
defer {
  list.push_mut(items, 2)
}
```

```
error: lowering failed: FinalizerCannotChangeLayout
```

late_layout 里同样不行（报 **InvalidLateLayout**），因为这两个块运行在布局定下来之后，不能再长出内容。

一个绑定被改写，另一个绑定不受影响：

```
// snapshot 拿到的是当时的副本，之后改 items 不影响它。
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)))
```

元素不限于整数

```
const chunks: list = list.concat(
  list.of(b"XR"),
  list.of(bytes.le(0x1234, 2))
)

for chunk in chunks {
  emit.bytes(chunk)      // 先 58 52, 再 34 12
}
```

列表负责顺序，每个元素负责自己的二进制表示。记录的各段宽度不同时，这样最好写。

映射

```
// "arch" 写两次：第二次是改已有键的值，不是新增。
let options: map = map.new()
map.set_mut(options, "arch", "x64")
map.set_mut(options, "mode", "release")
map.set_mut(options, "arch", "rv64")
map.set_mut(options, "tags", list.of("asm", "dsl"))

// 三个不同的键：arch / mode / tags。
assert(len(options) == 3)
assert(map.has(options, "arch"))
assert(!map.has(options, "missing"))
assert(map.get(options, "arch") == "rv64")
// 键不在时用兜底值，而不是报错。
assert(map.get_or(options, "missing", "default") == "default")
assert(list.eq(map.get(options, "tags"), list.of("asm", "dsl")))
```

调用

作用

`map.new()`

空映射

`map.has(value, key)`

键是否存在

调用	作用
<code>map.get(value, key)</code>	读一个必须存在的键
<code>map.get_or(value, key, fallback)</code>	读一个可以不存在的键
<code>map.eq(left, right)</code>	比较内容，不比较插入顺序

上面 "arch" 写了两次，长度仍然是 3：第二次是替换而不是新增，键的位置也不变。

遍历映射

映射是查找结构，遍历要先取出它的键列表或值列表：

```
let fields: map = map.new()
map.set_mut(fields, "magic", b"XR")
map.set_mut(fields, "version", bytes.le(3, 2))

const keys: list = map.keys(fields)
const values: list = map.values(fields)

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

for value in values {
    emit.bytes(value)    // 先 58 52, 再 03 00
}
```

查名字用 `map.keys`，查存下来的值用 `map.values`。当处理顺序是文件格式的一部分时，把顺序放进列表，映射只负责查找。

在函数中组合集合

```
fn encode_u16(values: list) -> bytes {
    let result: bytes = bytes.new()

    for value in values {
        result = bytes.concat(result, bytes.le(value, 2))
    }

    return result
}
```

```
const words: list = list.of(0x1234, 0xabcd)
emit.bytes(encode_u16(words))    // 这里写出: 34 12 cd ab
```

函数接收一份有序描述，计算出编码结果返回；结果放进输出的哪一段由调用者决定。四种类型在这里各司其职：字符串和映射负责描述，列表决定次序，字节序列承载表示，过程决定它落在哪里。

[返回语言指南](#)

第 7 章：词法单元与模式匹配

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

字符串辅助函数把文本当成一串字符处理，这对名称、路径、分隔字段和简单替换够用。类源码的文本带的结构比这更多：

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

这里的逗号、方括号、圆括号、名字、运算符和字面量各自都有含义。按空格切分就会丢掉这层含义：`[rbx+(rcx*4)]` 被切成三块，中间的运算关系看不出来了。反过来，两个只差空格、意思完全一样的写法，切出来的结果又不一样。`tokens.of` 把这类文本转成词法单元列表：

```
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")
```

```
emit.bytes(tokens.join(source_tokens)) // 规范化空格，不是原样输出
```

`tokens.of` 只保留词法单元本身，空白一律丢掉。`tokens.join` 再用统一的空格把它们拼回一段文本。所以 `tokens.join(tokens.of(x))` 和 `x` 不一定逐字节相同：上例里 `"count + 0x2a"` 拼回来是 `count+0x2a`。

匹配词法单元结构

```
// 拿这一行源码当输入：要求它以 load 开头，逗号后跟一段配平的词法单元。
const line: string = "load rax, [rbx+(rcx*4)]"
const result: map = match.tokens(
  "=load destination:name =, source:tokens",
  line
)

// 整条输入都匹配上了，才有捕获可读。
assert(map.get(result, "ok"))

const captures: map = map.get(result, "captures")
assert(map.get(captures, "destination") == "rax")
// source 是 tokens 捕获，拿到的是词法单元列表，拼回来仍是原文那一段。
assert(tokens.join(map.get(captures, "source")) == "[rbx+(rcx*4)]")
```

结果是带两个字段的映射：`"ok"` 是布尔值，表示整条输入是否匹配；`"captures"` 是各命名捕获取出的值。只有在 `"ok"` 为真之后才去读捕获。

字面量模式词法单元

以 `=` 开头的模式片段匹配一个确切的词法单元：

```
// `=&&` 要求这里出现的就是逻辑与这个词法单元本身。
const result: map = match.tokens(
  "left:name =&& right:name",
  "ready && enabled"
)

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

片段	匹配
<code>=load</code>	名字词法单元 <code>load</code>
<code>=,</code>	逗号词法单元
<code>=[</code>	左方括号词法单元
<code>===</code>	<code>==</code> 词法单元

第一个 `=` 是模式标记，所以 `===` 的意思是"匹配 `==` 这个词法单元"，不是三字符的相等运算符。模式片段之间用空白分隔。匹配比较的是词法单元，输入里用多少空格都不影响结果。

捕获种类

捕获写成 `name:kind`，名字会成为 `"captures"` 里的键；同一个模式里捕获名必须唯一。

种类	匹配	捕获到的值
<code>token</code>	任意一个词法单元	<code>string</code>
<code>name</code>	一个类标识符的名字	<code>string</code>
<code>int</code>	一个整数字面量	整数值
<code>quoted</code>	一个引号词法单元	去掉引号的 <code>string</code>
<code>tokens</code>	一段配平的词法单元范围	词法单元字符串的 <code>list</code>

```

const assignment: map = match.tokens("=set target:name =, value:int", "set count, 0x2a")
const operation: map = match.tokens("left:name operator:token right:name", "count + step")
const message: map = match.tokens("db text:quoted", "db 'READY'")

assert(map.get(assignment, "ok"))

const assignment_captures: map = map.get(assignment, "captures")
const operation_captures: map = map.get(operation, "captures")
const message_captures: map = map.get(message, "captures")

assert(map.get(assignment_captures, "target") == "count")
assert(map.get(assignment_captures, "value") == 42)           // `int` 变成了数
assert(map.get(operation_captures, "operator") == "+")       // `token` 是文本
assert(map.get(message_captures, "text") == "READY")         // `quoted` 去掉了引号

```

配平的词法单元范围(类似括号平衡)

一个 **tokens** 捕获可以连续吃下多个词法单元，条件是里面的圆括号、方括号和花括号必须成对：

```

// address 要吃掉一个整体：从 [ 开始，到配平的 ] 结束。
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)]")

```

它捕到的是词法单元列表，不是字符串。所以下一层匹配器可以直接拿它继续匹配，不用再分一次词。

配平只认 **()**、**[]** 和 **{}**。**<** 和 **>** 是普通的运算符词法单元，不参与配平：

```

// < 不是配平字符，所以整条输入都能被 expression 吃掉。
const result: map = match.tokens("expression:tokens", "left < right")
const captures: map = map.get(result, "captures")

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

```

```
// 拼回来时空格按规范重建, left<right 之间不再有空格。
assert(tokens.join(map.get(captures, "expression")) == "left<right")
```

最小匹配与回溯

tokens 捕获后面还跟着模式片段时, 它先尽量少吃, 只有剩下的模式匹配不上才扩张:

```
// prefix 先试空; 空的时候 value:int 匹配不上 name, 于是 prefix 扩张到 name 再试。
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)
```

带类型的捕获遇到不对的词法单元只是一次普通失配, 不是模式本身出错:

```
// int 捕获碰上名字, 只是一次没匹配上: ok 为假, 不报错。
const result: map = match.tokens("value:int", "name")

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

正因如此, 依次尝试多个合法模式是可行的。

空词法单元范围

tokens 捕获可以是空的:

```
// arguments 一个词法单元都没吃到, 也是合法的捕获。
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)
```

要求范围非空时, 在匹配成功后检查它的长度, 或者在模式里再放一个必需捕获。

匹配已有词法单元列表

`match.tokens` 的输入也可以已经是词法单元列表：

```
// 先自己分好词，再把列表交给 match.tokens，省掉重复分词。
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)
```

`match.tokens` 也可以直接匹配一段已经分好词的输入。多步处理同一段源码时，把词法单元列表传给下一个辅助函数，比每次都重新分词省事；只有确实需要文本时才用 `tokens.join` 转回去。

尝试不同结构

模式里没有"或"运算符，所以每种完整结构都放在编译期控制流里依次尝试：

```
// 两种结构各试一次：load 先试，不中再试 store。
const line: string = "store r1, r2"
const load: map = match.tokens(
  "=load destination:name =, source:name",
  line
)

if map.get(load, "ok") {
  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
  )

  assert(map.get(store, "ok"))

  const captures: map = map.get(store, "captures")
  emit.bytes(map.get(captures, "destination"))
}
```

```
emit.bytes(map.get(captures, "source"))
}
```

结构再多就不好这样堆 `if` 了。每种结构写成一个返回 `map` 的函数，然后按顺序试，成功就停：

```
// 每种结构一个函数，函数名就是它的模式。
fn try_load(line: string) -> map {
    return match.tokens("=load destination:name =, source:name", line)
}

fn try_store(line: string) -> map {
    return match.tokens("=store destination:name =, source:name", line)
}

const line: string = "store r1, r2"

// 从上往下试：先命中的那个返回搜索结果，后面的不再试。
if map.get(try_load(line), "ok") {
    emit.bytes("load")
} else if map.get(try_store(line), "ok") {
    const captures: map = map.get(try_store(line), "captures")
    emit.bytes(map.get(captures, "destination")) // 这里写出: 72 31 72 32
    emit.bytes(map.get(captures, "source"))
} else {
    err("unknown instruction shape", line)
}
```

写成函数有两个好处：模式本身集中在函数体里，调用处一眼就能看出先试哪种、后试哪种；要加新结构，就再加一个函数、在链尾加一个分支。

失配与无效模式

普通的形状或类型不符返回 `"ok" == false`：字面量对不上、`name` 捕获碰到标点、`int` 捕获碰到名字、`tokens` 捕获里括号不配平、模式匹配完了输入还有剩余，都是这一类。下面的例子里右方括号没配平，`tokens` 捕获吃不下，于是 `"ok"` 为假：

```
// 右方括号没配平，tokens 捕获吃不下 → ok 为假。
const result: map = match.tokens("=load dest:name =, rest:tokens", "load rax,
[r bx")

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

模式写错了是另一回事：它作为源码错误被拒绝，而不是报成失配。未知的捕获种类、空字面量、重复的捕获名都会让汇编停下来：

```
// 两个捕获都叫 value: 模式本身写错了，不是输入不匹配。
const invalid: map = match.tokens(
  "value:name value:int",
  "left 42"
)
```

```
bad.xir:1:1: error: an expression in this statement is not valid
(InvalidExpression)
```

上面的例子属于无效模式：两个捕获都叫 **"value"**。

捕获名必须符合标识符写法（字母或下划线开头，后面是字母、数字或下划线）。**v**：这种空种类、**=** 这种空字面量、未知的种类名，都会按无效模式拒绝。

匹配也有上限：输入最多 256 个词法单元，模式最多 64 个片段，回溯尝试最多 4096 次，这三项超出都报 **InvalidArgument**。

[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。

目标

每条指令都按当前目标编码。目标包含指令集系列，以及该指令集需要的位宽。

命令行选项给出起始目标：

```
xirasm program.xir --isa x86-64
xirasm program.xir --isa x86
xirasm program.xir --isa rv64
xirasm program.xir --isa rv32
xirasm module.spvasm --isa spv
```

汇编单个文件不需要选项：`xirasm program.xir` 在源文件旁输出纯二进制 `program.bin`，`-o` 只是覆盖这个路径。`--isa`（旧拼写 `--target`）是默认值，只作用于自己没有选择指令集的源码；源码里写了 `x86.use32()`，或 `import` 了 A64 宏库，结果以源码为准。

没有显式选择时，XIRASM 默认使用 64 位 x86。示例、可复用 `include` 和依赖特定位宽的代码都建议在源文件里写明：

```
// 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的指令，不回改已经写出的指令。

模式调用

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

同一个源文件里可以切换：

```
// 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。

上表只列原生目标。AArch64 不在其中：x86、RISC-V、SPIR-V 由后端编码器直接支持，而 AArch64 指令是在宏库 `arm/a64-macros.inc` 里用宏实现的，属于可执行格式扩展那一层，不是原生 API。要写 A64 指令，导入该库之后按 AArch64 汇编文本直接写即可（见第 5 章《AArch64 指令宏库》）。

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、固件组件和混合模式程序仍要自己安排运行时模式转换。RISC-V 的宽度选择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
```

x86 的模式概念不适用于 RISC-V 或 SPIR-V。XIRASM 分别保存各 ISA 的目标设置，不会把所有 ISA 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：

```
// SPIR-V 按模块写：这里是模块头部和类型声明，不是逐条机器指令。
spv.use()

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

命令行可用 `--isa spv` 或 `--isa 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。

查询当前目标

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

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`。

判定位宽用 `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
}
```

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
```

一行由助记符和操作数组成，操作数之间用空格分隔。方括号、圆括号或花括号里的逗号属于嵌套操作数的一部分，不会把指令错误地拆开。

编译期调用按编译期语法写，处理器指令按汇编语法写。指令行尾不写分号——那是错误：

```
// 这两行是编译期接口调用。  
x86.use64()  
emit.u8(0x90)  
  
// 这一行是处理器指令，末尾同样不写分号。  
nop
```

XIRASM 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
```

距离还没确定的跳转按 **near**（近跳）形式编码，所以即使两条指令相邻，`jmp target` 也是五字节。距离已知时写明要哪种形式：

```
x86.use64()

loop:
    nop
    jmp short loop    // short 选两字节形式，所以是 eb fd
```

short 选两字节形式，**near** 选宽形式，与普通 x86 汇编一致。

汇编器以符号形式保留整个表达式，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

标号写在名字后面加冒号：

```
// 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；不要在源码中手动计算跳转偏移。

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 的处理顺序是：

1. 源文件定义标号并生成指令片段。
2. 后端先编码指令，把依赖标号的字段暂时留下。
3. 布局器给标号和片段分配最终地址。
4. 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
```

只要源文件描述的是指令及其标号表达式，而不是手动填写的偏移，前面的指令改变长度就不会让这里失效。

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")
```

上面那段等价于直接写出来：

```
// 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`。

动态标号只在确实需要时使用，不要用它替代 `loop:`、`done:` 这样的静态写法。

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))
```

指令内部的标号引用通过地址计算完成，不需要手动查询。`label_addr()` 用于在数据里写入标号地址这类场景。

布局未稳定时，在 `defer` 里延迟查询（见第 12 章）。需要重定位时用格式接口提供的重定位声明，不要手动填偏移。

[返回目录](#)

第 9 章：数据与二进制布局

汇编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。

为什么要自己布局，而不是交给外部链接器去拼二进制？

因为 XIRASM 一开始就被设计成既是编译器、又是解释器，靠这个能力它还能当链接器——一个工具做所有事。安卓 APK 不必依赖外部工具就能打包、任意可执行格式都能生成，原因之一就在这里：所有能力都源自对二进制的控制能力，也就是布局。

布局本来是高级静态编译器和链接器干的事。传统的 NASM 那类汇编器只能生成中间目标文件，没有成熟的可执行格式层，还得依赖 GNU 工具链；XIRASM 不需要这些。写 Windows、Linux、macOS 都行，x86、RISC-V、AArch64 也都行。

XIRASM 在这一层提供两种写法：直接写数据的调用，以及定义结构体与联合体的类型系统。几行数据直接用写接口就够了；会被反复使用的记录格式，定义成结构体。

写出数据

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；需要更宽的字段时，换一个更宽的写出接口。

简写形式把宽度写在助记符上：

简写	宽度	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 字节写出从 u64 零扩展。需要精确位模式时用 emit.bytes。dt 只是 10 字节整数写出形式，不表示 x87 f80 浮点格式。

浮点值

```
// 带小数点的字面量是 f64；窄化成 f32 要显式写 f32(...)。
const scale: f64 = 1.5
const compact: f32 = f32(scale)

emit.f32(compact)           // 这里写出: 00 00 c0 3f
emit.f64(scale * 2.0)       // 这里写出: 00 00 00 00 00 00 08 40
```

`emit.f32` 和 `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——要终止符就自己写。

`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")
```

要写出的是一段精确的二进制内容，用 `bytes`；要写出的是一段文本，用字符串。

预留空间

`reserve(count)` 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`。

预留表达的是未使用的空间，不是已初始化内容。后面还有已初始化的输出时，预留范围会在 flat binary 中成为真实的零填充间隙；连续的预留空间位于末尾时，可能只增加逻辑大小而不占用 raw 文件字节。第 11 章用 FOA 和尾部 reserve 解释这一区别。

填充与对齐

`pad(count, fill)` 写出 `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，默认是零：

```
// 使用默认填充值写出四个零字节。  
pad(4)
```

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

```
// 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)` 让输出位置前进到该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、4096 有效，3 或 24 会被拒绝。

四种调用各自的用法：

需求	使用
已知大小的未初始化空间	<code>reserve</code> 或 <code>rb/rw/rd/rp/rq/rt/rdq/rqq/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
}
```

这是自然对齐：tag 从偏移 0 开始，u32 要求 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，适合布局应遵循各字段对齐要求的内存记录。

需要精确的文件布局时，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。

紧凑布局移除内部和尾部填充，但类型本身的对齐要求不变：紧凑结构体仍然要对齐到它最大字段的宽度。

这在外层是自然布局时看得出来：

```
packed struct Inner {
    a: u8
    b: u64
}

struct Outer {
    tag: u8
    inner: Inner
    tail: u8
}

assert(sizeof(Inner) == 9)           // 紧凑：中间和末尾都没有填充
assert(sizeof(Outer) == 24)
assert(offset_of(Outer, inner) == 8) // Inner 要求 8 字节对齐，所以跳过 7 字节
assert(offset_of(Outer, tail) == 17)
```

Inner 内部没有填充，但它含有 **u64**，所以它自己要求 8 字节对齐。**Outer** 是自然布局，于是把 **inner** 放到偏移 8，末尾再补到 24。

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，使用紧凑布局；运行时内存记录使用自然布局。

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`，`magic` 和 `flags` 使用默认值。字面量中的字段按名称匹配，不按源码中的排列顺序匹配。

每个省略的字段都必须有默认值。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。

字段用普通的字段访问语法读取，如上面两个写出调用。字段访问是从值上读字段，所以先把字面量写进绑定，再从绑定读字段：

```
struct Pair {
    left: u32
    right: u32
}

// 字段要从值上读，所以先绑到名字上，再取字段。
const pair: Pair = Pair { left: 1, right: 2 }

emit.u8(pair.left)      // 这里写出: 01
```

`Pair { left: 1, right: 2 }.left` 这种直接写在字面量上的字段访问不被接受。

复合值存在于汇编期间，仅仅声明它并不会自动写入输出内容。

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。两者都是编译期表达式，可以用在指令、写出字段、断言和其他布局计算里：

```
// 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` 用于普通内存操作数：

```
x86.use64()

// tag 占 1 字节，value 要对齐到 4，所以中间补 3 字节，尾部再补 2 字节。
```

```

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

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

// 偏移由 offset_of 算，不要在源码里写死数字。
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，不会补齐调用边界需要的栈空间。在 PE64 程序里使用结构体栈帧时同样如此。

打包并写出结构体值

`pack(value)` 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)` 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
```

还需要比较、转换、把字节序列存进集合，或者传给另一个函数时用 `pack`；只需要立即写出该值时用 `emit.struct`。

联合体与当前字段

联合体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，因为默认值无法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)
```

`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。

何时使用结构体

情况	使用
只有少数字段	直接写出整数和字节
字段只写一次，之后不再需要名称	直接写出整数和字节

情况	使用
与外部格式强绑定，显式调用比类型声明更清晰	直接写出整数和字节
精确的二进制记录会被重复使用	紧凑结构体和联合体
字段名称能提高可读性	紧凑结构体和联合体
其他计算应由 <code>sizeof</code> 、 <code>offsetof</code> 驱动	紧凑结构体和联合体
值需要默认值、嵌套、比较或转成 <code>bytes</code>	紧凑结构体和联合体
记录表示对齐的内存而非序列化布局	自然布局结构体
填充是有意保留的，且应遵循字段对齐要求	自然布局结构体

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

[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，常常由不同的人维护，变更原因也各异。

XIRASM 分两条路加载外部文件：

- `include` 和 `import` 加载 XIRASM 源文件；
- `fs`、`json` 和 `toml` 加载 Meta 代码要用的数据。

分开之后项目结构更清楚：源文件提供声明，或执行输出操作；数据文件被读进来，变成源码可以检查、转换和写出的值。

`fs.*`、`json.*` 和 `toml.*` 读的都是汇编期间的数据。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，不在语言指南范围内。

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，所以函数只声明一次。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用 `import`。即使通过多条依赖路径引入同一个模块，声明和输出也不会重复。

`import` 语句必须写在源文件顶层，不能放进可能按条件执行的块里：

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

要在顶层选择导入哪些模块；如果行为取决于目标，就把目标判断写进模块内部。

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
```

include 只用在确实需要重复执行的地方：

- 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；
- 多次套用同一个源文件模板。

因为包含操作会重复执行，多次包含一个声明了相同函数、常量或类型的文件会报重名错误。这类文件属于模块，用 **import** 加载。

源文件路径解析

相对路径以包含 **include** 或 **import** 语句的那个文件所在目录为基准解析。

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")
```

这个路径相对于 **records.inc**，而不是进程工作目录或入口源文件。

当前源文件旁找不到相对路径对应的文件时，XIRASM 依次检查项目的 **include** 目录及其安装目录下的 include 目录。项目模块因此可以覆盖或补充已安装的库，不必在源码

里写本机绝对路径。

绝对路径也能用，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。

import 与 include

按实际执行方式自行选择，不看文件扩展名(都一样)：

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

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。这段代码移入模块后，`payload.bin` 相对于该模块所在文件解析。

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

```
const banner: string = fs.read_text("banner.txt")  
  
// 读进来的是字符串，可以直接用字符串接口检查。  
assert(contains(banner, "XIRASM"))
```

```
// 再交给写输出接口，原样进文件。  
emit.bytes(banner)
```

如果 `banner.txt` 包含：

```
XIRASM
```

写出的是文件里的原始字节，**不追加终止零字节，也不做换行转换**：文件最后一行的换行符照样写出去。

`fs.read_bytes(path)` 把整个文件读成 `bytes`，适用于图片、编码后的表格、预构建的记录及其他二进制数据。

字节只用于输出时，`emit.file(path)` 省掉中间绑定；`emit.file(path, offset, count)` 写出精确的范围。两者使用与 `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` 里使用。

列出目录

`fs.list_dir(path)` 返回目录中的条目名，`fs.is_dir(path)` 检查某个路径是否为目录：

```
// 把 assets 下的文件依次写进输出，子目录跳过。  
for entry in fs.list_dir("assets") {  
    if fs.is_dir(sym.join("assets/", entry)) {  
        continue  
    }  
    emit.file(sym.join("assets/", entry))  
}
```

列表里是条目名而不是路径，目录和文件一起出现；列表按字节升序排列，所以生成的映像不依赖宿主文件系统返回条目的顺序。这两个函数与上面的读取函数使用完全相同的路径解析规则。`fs.is_dir` 在路径无法定位时返回 `false`，所以适合作为“目录可有可无”时的判断依据。

读取字节范围

三参数形式的 `fs.read_bytes` 读取带边界的范围。第二个参数是基于零的文件偏移，第三个参数是要读的字节数：

```
// 跳过第 0 个字节，从第 1 个字节起读 2 个字节。
const middle: bytes = fs.read_bytes("payload.bin", 1, 2)

assert(len(middle) == 2)
emit.bytes(middle)
```

`payload.bin` 里是 10 20 30 40，所以写出的是：

```
20 30
```

所请求的范围必须完整存在。越界读取会报错，不会静默截断。

加载 JSON

`json.file(path)` 一次完成 JSON 文件的读取和解析：

```
// 一次读入并解析：得到的是一个 map。
const config: map = json.file("config.json")
const values: list = map.get(config, "values")

// 布尔值可以直接当条件用。
assert(map.get(config, "enabled"))

emit.bytes(map.get(config, "name"))    // 字符串
emit.u8(map.get(config, "bits"))      // 整数
emit.u8(list.get(values, 0))          // 数组变成 list，按下标取
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)` 解析已经装在 `string` 或 `bytes` 值里的 JSON：

```
// 自己读进来再解析：文件里装的必须是合法的 JSON。
const raw: string = fs.read_text("config.json")
const config: map = json.parse(raw)

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

JSON 对象转成 `map`，数组转成 `list`，字符串和布尔值保持相应的 Meta 类型，非负整数转成整数值，`null` 转成 `void`。

浮点数和负整数目前不在 Meta 数据模型内。重复的键和格式错误的 JSON 会被拒绝。

加载 TOML

`toml.file(path)` 是 TOML 的便捷入口：

```
// TOML 的表（[target]）会变成嵌套的 map。
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：

`toml.parse(value)` 从 `string` 或 `bytes` 值中解析 TOML:

```
// 同样可以先读成文本，再交给 toml.parse。
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 表转成 map，数组转成 list，字符串、布尔值和非负整数转成相应的 Meta 值。Meta 数据转换目前不接受浮点数和时间戳。

配置数据与二进制数据

结构化配置和原始二进制数据解决的是不同问题。

情况	使用
字段要按名称访问	JSON 或 TOML
输入预期由人工编辑	JSON 或 TOML
同一份输入要驱动多个生成值	JSON 或 TOML
需要校验和条件生成	JSON 或 TOML
文件内容已经是所需的二进制表示	<code>fs.read_bytes</code>
需要字节级精确	<code>fs.read_bytes</code>
只需要有限的字节范围	<code>fs.read_bytes</code>
解析不会带来有用的结构	<code>fs.read_bytes</code>
文本本身就是数据	<code>fs.read_text</code>
要显式应用某个解析器	<code>fs.read_text</code>

模块组织

中等规模的项目可以这样组织：

```
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，也可以通过显式路径（如 `../data/config.toml`）读取。

模块行为保持可读的做法：

- 声明库用 `import`；
- `include` 只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；
- 结构化文件只解析一次，复用得到的 map 或 list；
- 大型文件只需要部分内容时，用有界的二进制读取。

[返回目录](#)

第 11 章：输出区域与虚拟数据

写最简单的 flat binary 时，“地址”和“文件偏移”是一样的（标号值 / 偏移值）：第一个字节的地址是 0，文件偏移也是 0，往后一起长，这在汇编器里叫地址跟进，反正中文就这么理解就对了。

如果你不用自己构造可执行格式，比如自定义格式，或不想专研 OS 那些可执行格式，那么可以跳过这部分，只了解 `origin`/`here` 这两个 API 就行。

只要开始写 PE、ELF、COFF，或者自己定一种二进制格式，这个假设就不成立了：

- `.text` 运行时可能映射到 `0x401000`，但在文件里从一个很小的偏移开始；
- BSS 要占住一段运行时地址，却不能把那一大段零都写进文件；
- 文件头里的 RVA、raw pointer、raw size、virtual size，要等各段都写完了才能回填；
- 有些表要先在临时空间里生成、测量、改字节，最后才复制到真正的输出里。

XIRASM 用四个互相独立的量来描述这些差别：**逻辑地址（RVA）、raw 文件偏移（FOA）、最终写进文件的字节，以及临时虚拟输出。**

RVA 与 FOA

一个真实输出区域有四个量：

名称	含义
<code>origin</code>	这个区域的逻辑地址基准。标号地址从这里开始计算。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)	这个区域在 raw 文件里的起始偏移。
<code>logical size</code>	这个区域占掉的逻辑地址范围。 <code>reserve</code> 计入这个大小。
<code>file size</code>	这个区域最后真正写进 raw 文件的字节数。区域尾部的 <code>reserve</code> 可以不计入。

这四个量互相代替不了。改 `origin` 不会顺手补上文件里的填充；改 FOA 也不会挪动标号地址。PE 头里 `VirtualAddress`、`PointerToRawData`、`VirtualSize`、`SizeOfRawData` 容易写错，就是因为它们分别来自这四个量。

写布局时常用这几个查询。注意它们回答的是**两个不同的坐标系**：逻辑地址那一套（和标号同一个坐标系），和文件偏移那一套：

查询	它回答的问题
<code>region_base()</code>	这个区域的逻辑地址基准是多少（标号地址从这个基准开始计算）。
<code>here()</code>	当前逻辑地址是多少。作用相当于传统汇编器里的 <code>\$</code> 。

查询	它回答的问题
<code>file_offset()</code>	已经写进 raw 文件的字节数是多少。它就是下一个真字节要落的文件位置。
<code>file_cursor_real()</code>	同上，和 <code>file_offset()</code> 取值完全一样。
<code>file_cursor_potential()</code>	如果把尾部那段还没落地的 <code>reserve</code> 也计入，下一个文件位置会是多少。
<code>tail_reserve_size()</code>	当前区域末尾还有多少个保留字节没进 raw 文件。

`file_offset()` 和 `file_cursor_real()` 是一回事，两个名字都在用。真正有区别的是后两个：`file_cursor_potential()` 把待落的预留计入，`file_offset()` 不计入。

用 `print` 看中间值

上面这几个查询在不同阶段的取值不一样，光看文档容易记混。`print` 能把值直接打出来，是排查布局问题时最顺手的工具：

- 它不写输出文件，只往诊断里加一条 `note`；
- 值按内容打印：整数给十进制数值（不是字节），字符串给文本，`bytes` 给 `b"..."`，`list` 和 `map` 带长度展开。

`print` 可以接多个实参，它们按顺序拼在一条 `note` 里。`origin(0x4000)`、写一个字节、再 `reserve(3)` 之后：

```
origin(0x4000)

emit.u8(0xaa)
reserve(3)           // 逻辑上占 3 字节，文件里还没有它们

print("here", here())
print("file_offset", file_offset())
print("cursor_real", file_cursor_real())
print("cursor_potential", file_cursor_potential())
print("tail_reserve", tail_reserve_size())
```

装配时打出来的是：

```
note: here 16388
note: file_offset 1
note: cursor_real 1
```

```
note: cursor_potential 4
note: tail_reserve 3
```

读法：**here** 是逻辑地址 **0x4004**，**file_offset** 是文件偏移 **1**——两个数在各自的坐标系里都表示"当前位置"，所以本来就该不一样，不能拿数值直接比。

file_cursor_potential() 的 **4** 意思是"如果那 3 个预留字节也写进文件，下一个文件位置会是 4"。

warn 报 warning，**err** 报 error 并让汇编失败；**print** 只是 note，不影响结果。

这些接口主要给自定义格式和格式库内部用。写普通 PE/COFF/ELF 时先看 **format.inc** 的包装层，大部分 RVA/FOA 关系它已经替你维护好了。

origin 只改逻辑地址

origin(address) 改当前区域的逻辑地址基准，不动文件偏移：

```
// 区域逻辑起点设为 0x4000；文件里仍然从 0 开始。
origin(0x4000)

start:
emit.u8(0xaa)

assert(region_base() == 0x4000)
assert(label_addr("start") == 0x4000)
assert(here() == 0x4001)      // 写出一个字节后逻辑地址前进一格
assert(file_offset() == 1)
```

输出只有一个字节：

```
aa
```

start 的地址是 **0x4000**，文件里却只写了一个字节。**origin** 用来给 flat 输出设装载地址，或者在某个区域里挪动标号地址基准。它不会创建新区，也不会把文件位置跳到 **0x4000**。

region.begin 指定 RVA 和 FOA

`region.begin(name, origin, file_offset)` 开始一个新的真实输出区域，同时给出这个区域的逻辑地址基准和起始 FOA。三个参数都要写：参数个数不对会报 `InvalidApiArity`。

```
// 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，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。中间那一段没有区域写入，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。它只说明：接下来的输出属于一个新区域，这个区域的逻辑地址和 raw 文件偏移分别是多少。

区域之间是否按 FOA 递增、是否重叠、是否留洞，由调用者自己负责。写标准格式时不要拿 `region.begin` 去拼 PE/ELF/COFF 的文件头；优先用 `format.inc`，只在手写自定义格式或实现格式辅助函数时才直接管理区域。

reserve 与尾部预留

写真实字节时，逻辑地址和 raw 文件尾部一起前进：

```
emit.u8(0xaa)

assert(file_cursor_real() == 1)           // 一个字节真的进了文件
assert(file_cursor_potential() == 1)
assert(tail_reserve_size() == 0)         // 没有待落的预留
```

`reserve(n)` 不一样：它推进逻辑地址，但只要这段预留还在区域末尾，XIRASM 就不会立刻把它写成文件里的零：

```
emit.u8(0xaa)      // 文件里 1 个字节
reserve(3)         // 逻辑地址前进 3 格，文件没动

assert(here() == 4) // 逻辑上已经占 4 字节
assert(file_cursor_real() == 1) // 文件里还是 1 个字节
assert(file_cursor_potential() == 4) // 这 3 字节也可以选择保留
assert(tail_reserve_size() == 3)
```

这时区域逻辑上占 4 字节，raw 文件里只有 `aa`。尾部那 3 字节可以被裁掉，也可以在后面选择保留成文件里的零填充。

如果 `reserve` 之后又写了真实字节，它就不再是尾部预留，而是文件中间的间隙，必须进 raw 文件：

```
emit.u8(0xaa)
reserve(3)
emit.u8(0xbb) // 夹在两个真实字节中间，3 个零必须写出来

assert(file_cursor_real() == 5)
assert(file_cursor_potential() == 5)
assert(tail_reserve_size() == 0) // 已经不是尾部预留了
```

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：尾部预留增加逻辑大小；只有它被后面的真实字节夹住，或者你主动选择保留，才会变成 raw 文件里的零。

output.org 保留尾部预留

`output.org(name, origin)` 从逻辑偏移开始下一个区域。它把前面区域末尾那段还没落地的 `reserve` 一起计入，于是预留变成 raw 文件里真实的零填充：

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

```
// 逻辑偏移是 4，所以 next 从 FOA 4 开始，前面三个预留字节被写出来。
output.org("next", 0x2000)
emit.u8(0x42)
```

输出是：

```
41 00 00 00 42
```

output.section 裁掉尾部预留

`output.section(name, origin)` 从文件偏移开始下一个区域。它不看逻辑偏移，所以前一个区域末尾那段还没进文件的 `reserve` 直接被裁掉：

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

```
// 文件偏移是 1，所以 next 从 FOA 1 开始，尾部预留不写进文件。
output.section("next", 0x2000)
emit.u8(0x42)
```

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

```
41 42
```

BSS 一类区域要的正是这个：内存里要有地址范围，文件里不要写一大段零。

注意裁掉的是**区域尾部**那段预留。如果 `reserve` 后面还跟着真实字节，它早就是文件中间的间隙了，这段不会被删。

两者的区别在于新区域从哪里接：

操作	接在哪个位置	尾部预留的去向
<code>output.org</code>	当前 逻辑偏移 (<code>here()</code> 那个位置)	变成文件里的零填充

操作	接在哪个位置	尾部预留的去向
<code>output.section</code>	当前 文件偏移 （真实写到的位置）	被裁掉，不进文件

两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，也不会把尾部预留变成逻辑内容：

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

emit.bytes(b"ABC")      // 3 个真实字节
reserve(13)             // 推到逻辑地址 16，但文件里还是 3 字节

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

// 裁掉尾部预留后，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，把第一个区域的 raw size 补到 8。第二个区域从 FOA 8 开始：

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

第一个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` 之后，当前区域的文件输出就结束了；要接着写真实字节，得开始另一个区域。

它和 `align` 的区别：

- `align` 是普通输出操作，会推进逻辑地址；如果形成文件间隙，就要写出填充字节。

- `region.file_align` 是区域收尾操作，只对齐这个区域的 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

虚拟区域用来临时组装、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)           // 区域里先有 44 33 22 11
store.u32(table, load.u32(table) ^ 0x01010101) // 原地异或成 45 32 23 10
const encoded: bytes = load.bytes(table, 4)     // 取出四字节快照

virtual.end()

// 只有显式复制出来的字节才进主输出。
emit.bytes(encoded)
```

变换后复制到主输出的是：

```
45 32 23 10
```

对照一下：那个临时区域刚创建时里面是 `44 33 22 11`，异或之后才变成上面这四个字节；主输出里只有这最后的四字节，`44 33 22 11` 从没进过文件。

虚拟区域适合做资源表、导出表、字符串池、校验数据这类临时构造。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。

`virtual.begin()` 也可以不带参数，这时逻辑地址取**当前地址**：

```
emit.bytes(b"AB")

// 不给 origin: 临时区域从当前地址接着算。
virtual.begin()
emit.bytes(b"CT")
virtual.end()
```

```
emit.bytes(b"D")           // 这里写出: 41 42 44
```

虚拟区域里的 **CT** 没有进文件，主输出只有 **AB** 和 **D**。参数只能给 0 个或 1 个，给两个会报 **InvalidApiArity**。

它有三条边界：

- 每个 **virtual.begin** 都要有对应的 **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之后调用。
- 需要回填的指令在虚拟区域里保持“未回填”：**emit.bytes** 拷贝的是虚拟字节的**快照**，而快照发生在回填之前，所以里面还是编码时的占位值。scratch 里放的是带引用的代码时，要用 **region.place** 把整个区域落位（见下）。

虚拟区域里的地址不是最终文件位置。要让虚拟内容进文件，可以用 **emit.bytes(...)** 或格式库的复制流程拷贝字节，也可以用 **region.place** 把整个区域落位。

虚拟区域和普通区域一样，字节只存在于逻辑地址上。写到尾部预留那一段会失败：

```
emit.u8(0xaa)
reserve(3)           // 尾部预留：逻辑上占 3 字节，文件里只有 1 字节

defer {
    store.u8(2, 0x55) // 地址 2 落在那段预留里，写不进去
}
```

诊断会把地址和文件长度都说清楚：

```
error: this store writes 1 byte at 0x2, but the finished output image holds 1 byte;
a reserved tail is not in the file (InvalidApiArgument)
```

region.place 把虚拟内容落进文件

region.place(label, origin, file_offset) 把 **label** 所在的虚拟区域变成真实输出区域，坐标由调用者给出。和 **emit.bytes** 不同，它保留这个区域里的指令、标号和引用，所以 scratch 里写的跳转和调用会由正常的 fixup 流程按**落位后的地址**解析：

```

x86.use64()

main_target:
emit.u8(0x90)

virtual.begin(0x9000)

gen_start:
call main_target      // 目标在虚拟区域外面，落位时才解析
jmp gen_start         // 自跳，同样是落位后解析

virtual.end()

late_layout {
    region.place("gen_start", 0x8000, 0x10)
}

```

虚拟区域从逻辑地址 **0x8000**、FOA **0x10** 落位。输出的头 16 个字节是主输出里那个 **0x90** 加上补齐，接着是落位过来的 **call** 和 **jmp**：

```

90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e8 fb 7f ff ff
e9 f6 ff ff ff

```

call 的位移 **0x7ffb** 指向落位后的 **main_target** (**0x0000**)，**jmp** 的位移 **0xfffffffff6** 指回落位后的 **gen_start** (**0x8010**)。两者都由落位后的地址计算得到，源文件里没有手写偏移。

load.* / store.* 只在当前活动区域

load.* 和 **store.*** 读写的是当前活动区域里已经生成的字节：

```

emit.u8(0x5a)

// 先读出原值，再原地改掉：first 留下的是改写前的 0x5a。
const first: u64 = load.u8(0)
store.u8(0, 0x42)

assert(load.u8(0) == 0x42)
assert(first == 0x5a)

```

`load.u8(0)` 读出的是改写前的值 `0x5a`，`store.u8(0, 0x42)` 把第 0 个字节改掉。最终写出的只有 `42`。

地址是相对于当前活动区域起点的偏移。虚拟区域的字节要靠 `load.*` 读出来，或者用 `region.place` 落位。

最终区域信息只在收尾阶段查询

这几个查询只有在收尾阶段才有确定的答案：

查询	返回什么
<code>region_file_offset(address)</code>	<code>address</code> 所在区域最终的起始 FOA。
<code>region_file_size(address)</code>	这个区域最终写进 raw 文件的字节数。
<code>region_logical_size(address)</code>	这个区域最终的逻辑大小，含尾部预留。

用它们回填文件头。例如 PE 的 section 头需要 `PointerToRawData`、`SizeOfRawData`、`VirtualSize`，ELF 的程序头需要 `p_offset`、`p_filesz`、`p_memsz` 这类字段，这些都要等布局定下来之后再写。

在普通输出阶段调用这几个查询会失败，因为最终映像那时还不存在。要等 `defer`（第 12 章）里再查。

[返回语言目录](#)

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计算；
- 符号表、字符串表、重定位表可能要等主源码登记完再生成。

同第十一章一样，不想了解可执行格式可不用看，因为如果你用了可执行格式，那么这些乱用会让二进制布局混乱。也是普通用户不是必须学习的章节。

XIRASM 不会反复执行整份源码去"碰运气收敛"。后期工作分两个阶段：

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。

要新增字节或新增区域，用 late_layout；只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

最常见的流程是：先写固定宽度的占位字段，再正常写 payload，最后在 defer 里把最终值写回占位字段。

```
size_field:
emit.u16(0)                // 占位两字节，值还不知道

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

defer {
    // payload 的真实长度在这里才算得出来。
    store.u16(size_field, payload_end - payload)
}
```

最终输出：

```
03 00 41 42 43
```

size_field 那两个字节在普通写出阶段就已经存在。defer 只是把它们从 00 00 改成 03 00；它没有插入新字节，也没有挪动后面的 payload。

`defer` 可以写在它引用的标号之前：

```
defer {
    // defer 写在标号之前也可以：执行时这些标号已经解析完了。
    store.u16(size_field, payload_end - payload)
}

size_field:
emit.u16(0)

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

等 `defer` 执行时，这些标号已经解析完，布局也已经稳定。

读取和改写最终字节

`load.u8`、`load.u16`、`load.u32`、`load.u64` 读最终输出里的小端整数，`load.bytes(address, count)` 读一段字节。

`load.*` 和 `store.*` 只能碰最终映像里真实存在的字节。尾部 `reserve` 不在文件里，终结器读写那一段会失败，诊断会写明地址、宽度和文件长度。

```
origin(0x4000)

header:
emit.u32(0)                // 后面回填 body 的长度
emit.u32(0)                // 后面回填 body 的逻辑地址

body:
emit.bytes(b"ABCD")
tail:
emit.bytes(b"????")       // 占位，稍后改写
image_end:

defer {
    store.u32(header, image_end - body)    // body 到映像末尾的长度
    store.u32(header + 4, body - region_base()) // body 的逻辑地址
    store.bytes(tail, b"OK!!")            // 覆盖占位

    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 raw 文件。这种被裁掉的尾部预留不能当成可写的占位字段。

`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报错。

计算校验和

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

```
origin(0x5000)

checksum:
emit.u16(0)                // 校验和占位

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

defer {
    let cursor = payload
    let sum = 0

    // 逐字节累加 payload。
    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` 受普通 Meta 循环的编译期迭代上限约束。`for` 目前不能放进 `defer`；需要扫字节时用边界清楚的 `while`。

查询最终区域信息

第 11 章那三个区域查询依赖最终布局，所以只能在 `defer` 里用：

```
region.begin("payload", 0x5000, 0)

file_size_field:
emit.u32(0)                // 回填区域的文件大小
logical_size_field:
emit.u32(0)                // 回填区域的逻辑大小

body:
emit.u8(0xaa)
reserve(3)                 // 尾部预留，不进文件

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
```

文件大小是 9：两个 `u32` 字段占 8 字节，`body` 占 1 字节。尾部 `reserve(3)` 不写进 raw 文件，所以不计入 `region_file_size`。逻辑大小是 12，因为尾部预留仍然占着逻辑地址范围。

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)`：该区域最终写进 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，含尾部预留。

它们不是写出阶段的"当前 FOA"查询。要当前 FOA 就用 `file_offset()` 或 `file_cursor_real()`；要最终区域大小就在 `defer` 里用 `region_file_*` 和 `region_logical_size`。

用过程登记回填

过程可以登记 `defer` 块，并把调用时传入的参数捕获进去：

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

field:
emit.u16(0)

patch_u16(field, 0x1234)    // 登记回填，此刻还不写字节
```

最终输出：

```
34 12
```

执行时机要注意：`patch_u16(...)` 在普通源码阶段就被调用了，调用时的 `address` 和 `value` 被保存进登记的 `defer` 块；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 里能写什么

`defer` 用于最终检查和回填。它可以写：

- `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；
- 标号、最终区域信息、已经稳定的输出字节。

它不能做任何改变布局的事。例如：

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

会被拒绝。下面这些同样不允许放进 `defer`：

- 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，或者在 `late_layout` 里创建；`defer` 只能改已经存在的字节。

多个 `defer` 的执行顺序

`defer` 按登记顺序执行：

```
emit.u8(0)  
  
defer {  
    store.u8(0, 1)           // 先把第 0 字节改成 1  
}  
  
defer {  
    store.u8(0, load.u8(0) + 1)  // 读到的是上一个 defer 写下的 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` 改同一个地址，后执行的块会看到前面的结果。不要把同一个字段拆到多个互相依赖的 `defer` 里，除非这个顺序就是你想要的格式规则。

`defer` 运行在这些步骤之后：

1. 普通源码处理；
2. `late_layout`；
3. 指令编码和布局松弛；
4. `fixup` 解析；
5. 最终文件字节生成；
6. `fixup` 修补。

所以 `defer` 看到的的就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` 用在"主源码已经登记完、但最终布局还没封存"的时候创建真实输出。

最简单的形式是追加字节：

```
emit.u8(0x10)

// 两个块按登记顺序各执行一次，接在默认输出尾部。
late_layout {
    emit.u8(0x20)
}

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

`late_layout` 块按登记顺序各执行一次，默认从默认输出区域的尾部接着写。输出是：

```
10 20 30
```

也就是说，默认行为是"接在当前默认输出尾部"。但 `late_layout` 不是只能往整个文件末尾追加：它允许调用输出区域 API，所以可以在块里显式打开一个真实区域，把晚生成的数据放到指定的 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的 raw 文件偏移：

```
table_foa_field:
emit.u32(0)                // 回填表所在的 FOA
emit.bytes(b"HDR")

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

virtual.begin(0)
table_tmp:
emit.bytes(b"TAB")        // 先在临时区域里把表拼出来
table_tmp_end:
virtual.end()

late_layout {
    // 建一个真实区域，把临时表拷到指定坐标。
    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` 并没有“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` 搭了类似 PE 的多段布局，那么 `late_layout` 里生成的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的 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，或者实现格式库里那些晚生成的符号表、字符串表、重定位表。

late_layout 会影响尾部预留

`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。追加在尾部预留后面时，前面那段预留就变成了文件中间的零填充：

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

```
late_layout {  
    emit.u8(0xbb)           // 尾部预留被夹在中间，必须写出来  
}
```

最终输出：

```
aa 00 00 00 bb
```

不想要这个结果，就不要在同一个区域的尾部预留后面直接追加真实字节。先用 `output.section` 裁掉尾部预留，或者用 `region.begin` 切到明确的目标区域。

late_layout 里能写什么

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

允许写：

- `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`。

不允许写：

- `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，都应在普通源码阶段先计算好，再作为参数交给 `late_layout`。需要读文件时也在普通阶段读成 `bytes`，不要在 `late_layout`

或 `defer` 里读。

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

三个阶段的分工

情况	用什么
字节能按正常顺序写出来	普通源码
需要标号、局部变量、函数、循环、读文件这些正常语言能力	普通源码
必须等主源码登记完才知道要写哪些真实字节	<code>late_layout</code>
晚生成的表、字符串池、重定位记录需要进入最终文件	<code>late_layout</code>
需要显式放到某个 FOA，或沿默认输出尾部继续	<code>late_layout</code>
这些字节必须影响最终 raw size、logical size、偏移和回填	<code>late_layout</code>
固定宽度占位字段需要最终值	<code>defer</code>
校验和需要完整的最终字节	<code>defer</code>
需要最终的 <code>region_file_offset / region_file_size / region_logical_size</code>	<code>defer</code>
只修补已有字节，不创建空间	<code>defer</code>

几条防错规则：

- 不要用 `defer` 创建缺失的空间；
- 不要在 `late_layout` 里写需要局部变量和循环的逻辑；
- 尾部预留是否进文件要明确选择 `output.section` 或 `output.org`，不要靠猜；
- 构造标准 PE/COFF/ELF 时优先用 `format.inc`，不要手写重复的布局关系。

[返回目录](#)

第三部分：构建程序

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。

flat binary 是 XIRASM 最基础的一层，也是它和传统汇编器拉开距离的地方。NASM 那类汇编器只能产出中间目标文件，可执行格式层要另外交给链接器和 GNU 工具链；XIRASM 把打包能力做在自己内部，所以从一段裸机器码到完整的可执行文件，都可以由同一份源码写出来。这一章讲的就是这层基础：字节怎么排、区域怎么分、晚生成的表和最终回填怎么落进文件。

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()

// 两个字节：mov eax, 42 和 ret。
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

自定义格式先写固定头，再写 payload：

```
// 固定头：4 字节签名 + 版本号 + 保留字段。
emit.bytes(b"RAW1")
emit.u16(1)
emit.u16(0)

// 头后面紧跟 payload。
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: 字段紧挨着排，不插对齐间隙。
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：

```
// 一条记录 = 1 字节标签 + 4 字节值。
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)

// 固定头：签名 + 三个占位字段，值都要等后面才知道。
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)           // 整个文件的大小
    store.u32(payload_foa_field, payload - region_base()) // payload 的文件偏移
    store.u32(checksum_field, load.u8(payload) + load.u8(payload + 1)) // '0' +
    'k'

    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：

```
// header 占文件前 4 字节, payload 的逻辑地址是另一套。
region.begin("header", 0x1000, 0)
emit.bytes(b"HDR0")

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

defer {
    assert(payload == 0x2000)           // 逻辑地址
    assert(region_file_offset(payload) == 4) // 文件偏移
}
```

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)           // 后面还有真实字节 → 中间间隙，进文件
emit.u8(0xee)
reserve(8)           // 区域尾部 → 只加逻辑大小

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)           // 占位，最后回填
emit.bytes(b"DATA")

late_layout {
    emit.bytes(b"END!") // 封存前追加，仍然参与布局
}

defer {
    store.u32(total_size, region_file_size(total_size))
    assert(load.u32(total_size) == 12) // 4 + 4 + 4
}
```

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)           // 回填表所在的 FOA
emit.bytes(b"HDR")

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

// 先在临时区域里把表拼出来。
virtual.begin(0)
table_tmp:
emit.bytes(b"TAB")
table_tmp_end:
virtual.end()

late_layout {
    // 建一个真实区域，把临时表拷到指定坐标。
    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

自定义格式应该把必须成立的规则写成断言：

- 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：

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

格式变复杂之后，读写的顺序不变：

- 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。语言指南只解释语言这一层能直接用的东西：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。

完整的格式选项、导入、导出、重定位、共享库和目标文件示例见 [《格式教程》](#)；自己安排文件头和表项时见 [《高级格式构造指南》](#)。

声明映像

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：

1. `format_elf64` 创建 ELF64 文件配置。
2. `format_segment` 声明一个 segment 及其权限。
3. `format_begin` 写出这份配置需要的格式结构。
4. `format_segment_begin` 和 `format_segment_end` 包住这个 segment 的实际指令和数据。
5. `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。更细的构造函数要求调用者自己管理计数、行号、偏移和格式规则；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。

[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。

解析不到的名字会写进消息里，因为位置本身说不清是哪个名字出的问题：

```
app.asm:7:1: error: undefined name in this expression: missing_helper(app)
app.asm:9:1: error: unknown call: no_such_function
```

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)
```

消息要写清违反了哪一条要求，并带上出问题的值。`invalid value` 这种空泛消息说明不了问题出在哪里。

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、格式值和对齐策略都应该命名。

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。需要在编译器收集一组声明、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`、`offsetof`
- 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。

按用途分模块

实用的项目布局按用途拆分：

- 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。

从 `format.inc` 开始

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；那些数字仍然应该交给 `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

断言应该检查简单的事实，不要重新计算内部实现。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](#)