

语言指南

XIRASM 中文语言指南

自然处理器指令与现代编译期语言

目录

1. XIRASM 语言指南
2. 01-introducing-xirasm
3. 02-values-and-bindings
4. 03-expressions
5. 04-control-flow
6. 05-functions-and-scope
7. 第6章：集合和文本
8. 07-tokens-and-pattern-matching
9. 08-targets-isa-text-and-labels
10. 第 9 章：数据与二进制布局
11. 10-modules-and-files
12. 第 11 章：输出区域与虚拟数据
13. 12-finalizers
14. 第 13 章：Flat 二进制与自定义文件格式
15. 第 14 章：可执行文件与目标文件格式
16. 第 15 章：诊断与实用惯例

XIRASM 语言指南

XIRASM 是一种多ISA的汇编器，可直接编写x86/risc-v/spv汇编,并构造二进制文件。

支持裸二进制/自定义二进制格式，支持windows/linux可执行格式。

XIRASM前端语言是类似编译期小语言来代替传统的汇编器那些伪指令,xirasm比宏指令更安全。

支持标准ISA处理器指令与结构化的编译期语言结合在一起：

指令仍然保持汇编代码的可读形式的写法，而值、函数、循环、数据布局和输出格式则使用清晰的高级现代化语言结构来表达,非常Nice.

本指南有点啰嗦,带有部分机翻，但关键点人工审核,不影响阅读和理解。

前端语法部分都是和正常高级语言一样的,但输出/区域/可执行格式需要自己去理解,这些内容在高级语言里是不会面向用户的，

汇编er要理解,这些都是手戳范畴，没有隐藏任何技术细节和黑魔法。

大部分文档是ai编写，噪声过大，人工修复非常耗时，不影响阅读。

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

另外出现一些Meta术语不要慌，meta是我之前汇编器的语言模块名字,XIRASM借用了它然后重构了，

总之就当是XIRASM前端语法就是了。

指南结构

第一部分：语言基础

1. **认识 XIRASM** - 汇编第一个 x86-64 程序。 - 理解处理器指令与编译期代码如何配合。 - 掌握基本源代码规则。
2. **值与绑定** - 声明常量和可变的局部绑定。 - 使用整数、布尔值、字符串和字节。 - 理解作用域和赋值。
3. **表达式** - 使用算术、比较、逻辑、位运算和字段表达式。 - 调用函数并组合编译期计算。
4. **流程控制** - 使用 `if` 选择需要生成的内容。 - 使用 `while` 和 `for` 重复执行操作。 - 根据编译期条件生成数据和指令。
5. **函数与作用域** - 编写不返回值的过程和返回值的函数。 - 使用参数、返回类型、局部绑定和嵌套作用域。

6. **集合与文本** - 创建和处理列表、映射、字符串及字节序列。 - 使用集合描述表格和需要生成的输出内容。
7. **词法单元与模式匹配** - 检查词法单元序列。 - 匹配源代码模式，并据此构建更高层的源代码处理工具。

第二部分：汇编器模型

8. **目标平台、处理器指令与标号** - 选择指令集和指令模式。 - 定义标号，并在指令中使用编译期值。 - 理解地址引用与回填。
9. **数据与二进制布局** - 写出整数、字符串和字节。 - 预留空间并控制对齐方式。 - 定义具有精确二进制布局的结构体和联合体。
10. **模块与文件**
 - 通过包含和导入复用源代码。
 - 从普通文件以及 JSON、TOML 格式中读取源数据。
11. **输出区域与虚拟数据**
 - 区分逻辑地址和文件中的实际位置。
 - 构建输出区域、预留一段空间，并使用虚拟数据作为临时工作区。
 - 读取并修改已经生成的字节。
12. **收尾处理**
 - 等布局稳定后再执行检查和回填。
 - 计算校验和与最终字段值。
 - 理解哪些操作不得改变最终布局。

第三部分：构建程序

13. **裸二进制文件与自定义格式**
 - 构建紧凑的裸二进制文件和应用专用文件格式。
14. **可执行文件与目标文件格式**
 - 理解格式接口负责完成哪些工作。
 - 无需手工计算文件头，即可构建一个最小可执行文件。
 - 通过可执行文件格式指南继续学习 PE、COFF、ELF、导入、导出、BSS（未初始化数据区）和重定位。
15. **诊断信息与实用约定**
 - 报告无效输入，并检查布局是否符合预期。
 - 组织可以复用的汇编项目。
 - 在掌握本指南后继续查阅接口参考和可执行文件格式指南。

同一份源文件，两种代码

XIRASM 源文件可以包含两种形式的代码：

- **机器指令**：就是你熟悉的汇编源码那些指令。
- **编译时代码**：编译期执行，比如计算值，生成数据，控制汇编流程，代码生成，复杂DSL等需求。

两种代码(前端语法和ISA汇编语法)写在同一语言里。

且计算出的值可以直接用在指令中，运行时接口可以引用标签，代码也可以同时产生数据和机器指令(静态数据生成)。

例子：

```
// 开启 64 位 x86 模式。
x86.use64();

// 编译时计算常量，指令里直接用。
const answer: u32 = 40 + 2

entry:
    // 把计算好的数值放进 eax，然后返回。实际上这就是硬编码需要的值而已。
    mov eax, answer
    ret
```

简单说明：

- `x86.use64()`：选择 64 位 x86 指令编码模式，这是后端编码器需要的，否则编码器无法正确编码。
- `const answer: u32 = 40 + 2` 声明编译时常量。
- `entry`：在当前逻辑地址设一个标签。
- `mov eax, answer` 是普通 x86 指令，把 `answer` 移入 `eax`，编译时自动代入常量值。
- `ret` 是普通 x86 指令。

`x86.use64()` 这种调用式语法用于编译时操作，`mov eax, answer` 是自然汇编语法，不用写成函数调用。

运行一个例子

源码存为 `hello.xir`(汇编文件后缀随意)，执行：

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

输出字节：

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

前五个字节是 `mov eax, 42` 的编码，最后一个字节是 `ret`。标签不产生字节，只记录一个地址。

直接输出时，XIRASM 默认生成 flat binary，不带操作系统格式的那些文件头。后面章节会讲怎么通过格式接口生成 PE、COFF、ELF。

编译时代码处理数据

编译时代码在汇编过程中执行，不是在最终程序里运行。

比如循环写四个字节：

```
// 编译时循环四次，每次写一个字节。  
const count: u8 = 4  
  
for value in range(0, count) {  
    db(value);  
}
```

生成的字节：

```
00 01 02 03
```

`range(0, count)` 生成从 0 到 count-1 的序列。`for` 在编译时执行，每次调 `db(value)` 写一个字节。

搞清楚这个区别：

做什么	怎么写
写机器指令	直接写汇编语法
计算和判断	编译时表达式
重复或条件生成	编译时流程控制
输出字节	db、dw、dd、dq等
封装可复用逻辑	函数
生成 PE/COFF/ELF	对应的格式接口

源文件语法注意点

几条简单规则：

- `//` 到行尾是单行注释，也可以写在 ISA 指令后；ISA 字符串操作数中的 `//` 仍是普通文本。
- 标签以 `:` 结尾。
- 函数和接口调用以 `;` 结尾。
- `return` 语句用 `;` 结尾。
- `const` 和 `let` 声明末尾**不加分号**。
- 代码块用 `{ }`。
- 调用和返回值可以跨行，左括号会一直等到对应的右括号。

完整例子：

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

const marker: u8 = 3
emit_marker(marker);

start:
    // 机器指令里直接用编译时定义的常量。但ISA指令后面别带分号；后端不支持这个特性，暂时不支持。
    mov eax, marker
    ret
```

常量和函数是编译时逻辑，`mov`、`ret` 是机器码。写法不同，但写在同一份文件里。

怎么读 XIRASM 源码

读源码时，按这几步判断每行：

1. 如果是机器指令，就是汇编语法。
2. 如果需要编译时计算或生成数据，就是编译时语言。
3. 如果需要写字节或预留空间，就调用数据接口API(类似传统的db,rb,dd,rd等)。
4. 如果需要标准可执行文件格式，就要导入格式接口。

下一章讲编译时语言基础：值、常量绑定和赋值。

[返回语言指南](#)

值只在编译时存在

XIRASM 的值都是编译时值。它们用来构建输出，但绑定本身不会写字节，也不会预留内存。

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

这行把值赋给名字 `magic`，但不会把 `4d 5a` 写到输出里。要输出这个值，需要调数据接口：

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

同样的规则适用于可变值、字符串、字节序列、集合以及函数返回值。只有指令、数据调用、布局操作或格式接口使用了这些值，它们才会成为输出的一部分。

常量

常量把一个名字绑定到一个不可能重新赋值的值：

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

一般形式是：

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

如果 XIRASM 能从初始值推断出类型，类型注解可以省略。当宽度或值类别属于二进制约定的一部分时，应该写类型。

常量声明末尾不加分号。给常量重新赋值会报错：

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

除非源文件在汇编时确实需要更新绑定，否则优先用 `const`。

可变绑定

编译时计算需要状态时用 `let`：

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

输出：

```
14 00 00 00
```

声明和赋值都在汇编时发生，不会在最终程序中创建名为 `offset` 的运行时变量。

形式是：

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

和常量一样，赋值末尾不加分号。作为语句的调用仍然需要分号：

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

类型注解和类型推断

绑定可以显式声明类型：

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

也可以靠推断：

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

常用的值类型：

类型	用途	示例
integer	通用编译时整数	42
u8	8 位无符号值	0xff
u16	16 位无符号值	0x5a4d
u32	32 位无符号值	0x401000
u64	64 位无符号值	0x1400000000
f32	IEEE-754 32 位浮点数	f32(1.5)
f64	IEEE-754 64 位浮点数	1.5
bool	编译时条件	true
string	编译时文本	"kernel"
bytes	确切的字节序列	b"PE"

列表、映射、结构体、联合体和类型值在后面的章节介绍。

指定位宽的整数类型适合表示二进制字段和函数接口。如果一个函数只关心值是整数而不在意具体宽度，可以用 `integer` 类型。最终输出时写多少个字节由调用接口决定。

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

字符串和字节序列

字符串和字节序列是不同类型的值：

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

文本名称、路径、生成的指令文本以及需要文本的接口，用字符串。需要精确字节序列时用 `bytes`。

有些输出接口两种类型都接受：

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

输出四个字节：

```
41 42 43 44
```

其他接口只接受其中一种。在绑定中区分清楚，能让这些约定在源码里更明显。

块作用域

绑定属于它声明时所在的作用域。块会创建一个嵌套作用域：

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

输出：

```
02 01
```

内部的 `value` 只在块内遮盖外部的常量。块结束后，外部的 `value` 重新可见。

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

当嵌套操作确实需要一个自然的局部名字时，遮盖是有用的。但如果两个含义放在源码里很难区分，就不要用相同的名字。

怎么选 const 和 let

选最精确的绑定来表达计算：

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

格式描述和生成汇编中的大多数绑定应该是常量。可变状态在局部、短生命周期、只跟一个计算相关时最清晰。

下一章讲产生这些值的表达式：算术、比较、布尔逻辑、位运算、字段访问和函数调用。

[返回语言指南](#)

表达式产生编译时值

表达式在 XIRASM 汇编时计算值。表达式可以用在声明、赋值、函数参数、return 语句、条件、断言、指令操作数和数据调用中。

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

输出 32 位值 24。乘法在汇编时执行，输出只有计算结果。

算术运算

整数表达式支持：

运算符	含义
+	加法
-	减法
*	乘法
/	整数除法
%	取余

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

输出值分别是 14、20、3、2。

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

一元 `+` 保持值不变。一元 `-` 产生 64 位补码值：

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

输出八个 `ff` 字节。

位运算和移位

位运算用于权限、掩码、指令字段和文件格式标志：

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

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

输出字节是 `05`。

掩码测试用括号包起来，这样意图更清楚，也不依赖位运算和比较运算的优先级。

比较和相等判断

表达式可以比较值：

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

大小比较作用于整数：

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

相等和不相等也可以用于兼容的非整数类型：

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

不相关的值类别之间比较会报错，不会隐式转换。

布尔逻辑和短路求值

布尔表达式：

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

`&&` 和 `||` 会短路：

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

// 左侧已经为真，因此右侧包含除零的表达式不会被计算。

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

```
assert(safe);
```

除法不会被执行，因为 `enabled` 是 `true`。

短路求值在后面的表达式需要前面的条件成立时很有用。

表达式中的函数调用

返回值函数或内置函数可以出现在任何能接受其结果类型的地方：

// 计算名称长度，并把 37 向上调整到 16 的整数倍。

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

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

调用可以嵌套：

// 先去掉两端空格，再把结果转换为大写。

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

只做输出或汇编动作的过程函数以语句方式调用，不产生表达式值。返回值函数在第5章介绍。

字段访问

有命名字段的值用 `target.field` 选择一个字段：

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

结构体和联合体值在第9章介绍。 `target.bits` 和 `target.isa` 这类目标查询属于目标条件，不能复制到绑定中。

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密：

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>

同优先级从左到右计算。

XIRASM 的优先级表可能和其他语言不一样。特别注意移位比加减法绑定更紧密：

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

第一个值是 `17`，因为计算方式是 `1 + (2 << 3)`。第二个值是 `24`。

在混合移位、算术、掩码或比较的表达式中用括号。括号能说明意图，也让后续修改更安全。

表达式错误

表达式无法产生明确的编译时值时，XIRASM 会报错。常见原因：

- 名字未定义
- 操作数值类别不对
- 除数为零
- 整数溢出或下溢
- 字段不存在
- 函数参数无效

这些错误会终止汇编。不会用零代替或忽略，否则可能静默破坏指令操作数或二进制布局。

下一章讲 `if`、`while` 和 `for` 中的表达式如何选择和重复源码。

[返回语言指南](#)

控制流在编译时执行

XIRASM 的控制流是编译期行为，可以执行源码操作。它不会自动生成运行时分支或循环。

```
// 这个编译期选项决定输出调试标记还是发布标记。
const debug_build = false

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

只有一个分支产生字节。 `debug_build` 为 `false` 时，输出包含 `RELEASE`，生成的文件里没有运行时条件判断。

编译时控制流用来：

- 选择目标特定的源码
- 重复生成数据或指令
- 遍历编译时集合
- 计算表格和偏移量
- 包含可选的文件格式记录
- 验证源码配置

运行时控制流还是用普通的 ISA 指令，比如 x86 的 `jmp`、`call` 和条件分支。

if 和 else

`if` 条件必须产生布尔值：

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

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

选中的块在自己的作用域中执行。另一个块不产生输出，也不执行编译时操作。

`else` 块可以省略：

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

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

需要多于两个分支时用 `else if`：

```
// 通过链式条件把数据大小划分为三个区间。
const payload_size = 32

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

输出 `02`。

选择 ISA 指令

`if` 块里可以包含 ISA 指令：

```
// 选择 64 位 x86 指令编码。
x86.use64();

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

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

因为 `return_zero` 为 `true`，生成的指令是：

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

没选中的 `mov` 指令不会进入输出。这是编译时指令选择，不是运行时条件分支。

目标条件

`if` 条件里可以直接用目标查询：

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

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

目标查询用专用的条件语法。直接在 `if` 条件中用，不要复制到普通绑定里。

遍历范围

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

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

输出：

```
00 01 02 03
```

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

循环绑定属于循环体：

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

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

遍历列表

for 循环可以遍历编译时列表的值：

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

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

按列表顺序遍历。适用于字节表、导入描述、生成的名字等集合数据。

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

while

当循环终止依赖循环内更新的值时用 while：

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

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

输出：

```
01 02 03 04
```

每次迭代前判断条件。初始为 false 则不执行循环体。

确保终止条件在循环附近能看出来。编译时循环如果停不下来会阻止汇编完成，所以 XIRASM 限制最多 1,000,000 次迭代，超限报错。

break 和 continue

`break` 结束当前最内层的 Meta 循环。`continue` 跳过本次迭代的剩余部分，开始下一次。两者都能用在 `for`、`while` 以及延迟块的 `while` 中，也包括嵌套的 `if` 语句。

例如：

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

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

怎么选控制流

需求	用这个
两个源码块二选一	<code>if / else</code>
多个源码块选一个	<code>if / else if / else</code>
按 ISA 或位宽选源码	目标条件
重复固定次数	<code>for + range</code>
遍历编译时集合	<code>for + 列表</code>
重复到条件满足为止	<code>while</code>
选择生成的指令	<code>if</code> 包含 ISA 指令
提前结束最内层循环	<code>break</code>
跳过当前迭代的剩余部分	<code>continue</code>

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

下一章把重复的计算和输出操作打包成函数，带参数、返回值和局部作用域。

[返回语言指南](#)

函数封装编译时工作

函数封装可复用的编译时逻辑。可以计算值、输出数据，或组合多个底层操作。

XIRASM 有两种函数：

- **过程函数**执行操作，不产生表达式值。
- **返回值函数**计算值，用 `->` 声明返回类型。

两种都用 `fn`、参数和块体。函数在汇编时执行。声明函数不产生字节，只有调用过程函数或把计算结果传给输出接口才会产生输出。

注意别把2个作用一起用，比如又要返回值，又要当宏用。

过程函数

过程函数组合输出操作：

```
// 把传入字节及其后继值作为一组连续数据写出。
fn emit_pair(value: u8) {
    db(value);
    db(value + 1);
}

// 两次调用分别生成 02 03 和 08 09。
emit_pair(2);
emit_pair(8);
```

输出：

```
02 03 08 09
```

没有 `->` 返回类型，就是过程函数。调用是语句，末尾加分号。

过程函数体可以包含声明、赋值、流程控制、数据输出、调用其他过程函数。适合封装重复的二进制记录、指令序列、表项和格式构建步骤。

过程函数本身不是值，不能用于初始化绑定：

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

const marker = emit_marker()
```

报错，因为 `emit_marker()` 只执行操作，不返回值。

参数和实参

参数写在括号里：

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

// 写出四个 cc 字节。
emit_run(0xcc, 4);
```

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

参数类型可以省略：

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

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

输出 `2a`。实参值绑定到对应参数。公开的辅助函数建议加类型注解，可明确约定并在调用处拦截不合适的实参。

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

返回值函数

函数需要产生表达式值时加 `-> type` :

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

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

函数返回 `0x80`, 输出:

```
80 00
```

`return` 末尾加分号。表达式转换为声明的返回类型。返回类型可以是整数、布尔值、字符串、字节序列等:

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

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

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

返回值函数可以用在任何能接受其返回类型的位置: 声明、实参、条件、函数调用或更大的表达式。

返回规则和副作用

返回值函数必须执行到 `return`。末尾无返回值会报错:

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

const result = incomplete(4)
```

返回值类型必须匹配：

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

报错，整数不满足 `bool` 约定。

返回值函数用于计算，不能输出数据或产生其他副作用：

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

需要改变输出或布局时用过程函数，需要计算值时用返回值函数。

过程函数不声明返回类型，也不能返回值：

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

过程函数执行到末尾即结束。

函数局部作用域

参数和函数内的绑定只属于当前调用：

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

    if result < 16 {
        result = 16
    }

    return result;
}

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

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

函数内的块创建嵌套作用域，可遮盖外部名字：

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

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

    return result;
}

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

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

中间计算用局部名字，避免临时状态泄漏到外部，多次调用互不干扰。

声明顺序

函数必须在调用前声明：

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

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

把调用移至声明前会报错，此时函数名尚未定义。

函数声明必须是顶层声明。不能将函数声明在另一个函数、循环、条件块或普通块内。相关函数在顶层相邻排列，后续源码调用。

第10章介绍如何通过包含文件将函数声明提供给另一个源文件。

递归和调用深度

返回值函数在有终止条件时可以递归调用自身：

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

    return value + triangular(value - 1);
}

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

输出 `0a`，即 `4 + 3 + 2 + 1` 的结果。

递归在编译时执行。递归深度应控制在一定范围内，终止条件必须明确。XIRASM 拒绝超过128层调用的函数链，防止失控递归。

遍历范围或集合用循环。计算本身具有递归结构且终止条件明确的用递归。

选哪种函数

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

每个函数专注一件事。小计算函数易于组合，小过程函数使输出效果在调用处可见。

下一章介绍列表、映射、字符串和字节序列，供需要处理集合和文本的函数使用。

[返回语言指南](#)

第6章：集合和文本

集合是编译时值

XIRASM 有四种编译时值类型：

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

这些类型不写数据。只有传给输出接口，数据才会写入输出文件。

字符串、字节序列、列表、映射都提供产生新值的 API。`list.push`、`map.set` 等操作返回新值，不修改原值。列表和映射还有可变接口，给 `let` 绑定的集合逐步添加内容。值在传递时复制，不共享内存。

字符串表示源文本

字符串存放编译时用的文本：

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

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

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

输出 `kernel64` 的八个文本字节。

常用字符串 API：

- `trim(text)` 去掉两端空白
- `lower(text)`、`upper(text)` 转 ASCII 字母大小写

- `starts_with`、`ends_with`、`contains` 检测文本
- `replace(text, needle, replacement)` 替换匹配文本
- `to_string(value)` 把值转成文本

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

分割和连接文本

`split` 按分隔符拆成字符串列表，`join` 把列表连成文本：

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

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

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

索引从零开始。用于处理小配置列表、格式选项中的文本、动态生成的名字。

用字节序列表示二进制值

`bytes` 是一段确切的字节序列：

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

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

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

输出 `58 49 52 00 03 00`。

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

构建字节序列

字节操作 API 都返回新值，可以链式调用：

```
// 从 ABC 的字节表示开始，在索引 1 处插入连字符。  
const base: bytes = bytes.from_hex("414243")  
const marked: bytes = bytes.insert(base, 1, b"-")  
// 从索引 2 开始替换一个字节，并追加两个 0xff 字节。  
const patched: bytes = bytes.replace(marked, 2, 1, b"Z")  
const trailer: bytes = bytes.repeat(2, 0xff)  
const result: bytes = bytes.concat(patched, trailer)  
  
// 只写出最终结果，各个中间值仍可继续复用。  
emit.bytes(result);
```

输出 `41 2d 5a 43 ff ff`。

操作包括：

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

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

列表保持顺序

列表存放有序值：

```
// 先追加一个元素，再替换索引 1 处的值。
const base: list = list.of(1, 2, 3)
const extended: list = list.push(base, 4)
const patched: list = list.set(extended, 1, 0xaa)
// 从索引 1 开始截取两个元素，原列表不会被修改。
const middle: list = list.slice(patched, 1, 2)

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

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

输出 `01 aa 03 04`。

`list.set` 返回替换元素后的新列表。`list.slice` 返回从起始索引开始的指定个元素。都不改原列表。

其他：`list.new()`、`list.get()`、`list.concat()`、`list.eq()`、`len()`。

用可变绑定的API版本的集合

局部算法需要逐步构建列表或映射时用可变接口：

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

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

第一个参数必须是 `let` 绑定的标识符。传 `const` 会报错。类型不匹配也会报错。`defer` 块只能更新它内部的局部 `let`。

值复制有明确边界：

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

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

列表可以放各种值

列表元素不限整数，可以放字符串、字节序列、映射等：

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

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

输出 `58 52 34 12`。

映射保存键值对

映射把字符串键关联到值：

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

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

`map.set` 返回新映射。键已存在则替换值。改 `changed` 不影响 `base`。

遍历映射要先把键或值转列表

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

```
// 用映射保存两个带名称的二进制字段。
const fields: map = map.set(
    map.set(map.new(), "magic", b"XR"),
    "version",
    bytes.le(3, 2)
)
// 键和值分别转换为列表，便于检查数量或逐项处理。
const keys: list = map.keys(fields)
const values: list = map.values(fields)
assert(len(keys) == 2);
assert(len(values) == 2);
// 逐项写出映射中保存的字节序列。
for value in values {
    emit.bytes(value);
}
```

文件格式规定了确切顺序的，用列表存。映射只负责查找。

在函数中组合集合

集合转换用函数封装：

```
// 把列表中的每个数值编码为两个小端字节。
fn encode_u16(values: list) -> bytes {
    let result: bytes = bytes.new()
    // 每次连接都会产生新的字节序列，并更新可变绑定 result。
    for value in values {
        result = bytes.concat(result, bytes.le(value, 2))
    }
    return result;
}

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

输出 `34 12 cd ab`。函数逐步构建不可变字节值，调用者控制何时写输出。

职责划分：字符串存名字和文本，映射存命名配置，列表存有序值，字节序列存最终二进制表示，过程函数控制何时写输出。

选哪种集合

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

下一章讲词法元素和模式匹配，用于需要按语法解析的源文本。

[返回语言指南](#)

第 7 章：词法单元与模式匹配

这个用于源码级解析，写DSL等用途，当前只是用于解析场景，比如x86编码形状类。

当前还没实现传统宏语法，当前是现代的api调用形似,写指令文本还是有点麻烦。

后续考虑实现一个传统macro,这样高级用户就能书写正常的isa指令，然后是有匹配模式来匹配各种ISA文本或形状模板直接编码机器码。

词法单元表示源代码结构

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

请看下面这段文本：

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

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

XIRASM 可以把类似源代码的文本转换为词法单元列表：

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

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

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

这段代码会写出规范化后的文本：

```
count+0x2a
```

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

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

匹配词法单元结构

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

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

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

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

结果是一个包含两个字段的映射：

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

只有检查 `"ok"` 后，才能读取捕获结果。

字面量模式词法单元

以 `=` 开头的模式片段会匹配一个完全相同的词法单元：

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

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

`=&&` 要求输入中出现逻辑与词法单元。同一规则也适用于名称和标点：

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

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

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

捕获种类

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

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

同一个模式中的捕获名称必须唯一。

单词法单元捕获可以精确描述小型命令语法：

```

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

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

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

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

```

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

平衡的词法单元范围

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

```

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

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

```

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

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

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

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

最小匹配与回溯

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

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

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

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

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

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

这种行为便于依次尝试多个有效模式。

空词法单元范围

`tokens` 捕获可以为空：

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

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

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

匹配已有词法单元列表

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

```
// 先一次词法分析,再把同一列表直接交给模式匹配器。
const input: list = tokens.of("load r1, 42")
const result: map = match.tokens(
    "=load destination:name =, value:int",
    input
)
const captures: map = map.get(result, "captures")

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

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

尝试不同结构

模式本身不包含“或者”运算符。应在编译期流程控制中依次尝试每完整结构:

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

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

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

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

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

匹配失败与无效模式

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

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

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

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

两个捕获都使用了键 `"value"`，这个模式无效。

请记住两者的区别：

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

字符串还是词法单元

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

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

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

[返回语言指南](#)

第 8 章：目标平台、处理器指令与标号

本章涵盖三件事：选择目标平台和指令集，用自然 ISA 语法写指令，用标号给输出位置命名(所谓的符号逻辑RV地址)。

当前目标平台

每个指令都根据当前目标平台编码。目标平台决定指令集系列，以及编码时用到的目标信息。

命令行选项设置初始目标平台：

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

默认使用 64 位 x86。源文件可以随时切换模式。文件头写明模式能帮助编辑器、语法高亮和特定宽度的代码片段正确配置：

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

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

模式切换只影响之后的指令。每个指令都记住编码时的目标平台。

选择 x86 和 RISC-V 模式

源文件内用这些接口切换模式：

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

同一源文件可以多次切换：

```
// 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
```

每条指令独立记住自己的编码模式。

多模式常见于引导代码、内核、固件。RISC-V 也一样：

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

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

x86 和 RISC-V/SPIR-V 的配置互不影响。

查询目标平台

编译时控制流可以检查当前目标平台：

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

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

可查询的目标系列值：

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

系列标识不反映当前位宽。`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 也可以用 `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
}
```

汇编指令写法

指令用所选指令集的常规文本语法写：

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

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

一行一个指令。空格提高可读性，括号和逗号处理嵌套操作数。

汇编指令末尾不写分号。编译期接口调用需要分号：

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

`x86.use64();` 和 `emit.u8(0x90);` 是编译期调用，`nop` 是原生指令。

XIRASM 把指令文本和当前目标平台一起保存，再交给指令集后端编码。标号、数字、表达式和地址计算在前端处理。

在指令中使用编译期值

编译期变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
```

编译器以符号形式保留表达式，地址确定后再求值。

只在你确实需要动态生成时才拼指令字符串。

静态标号

标号写在名字后加冒号：

```
// 使用静态标号表达普通控制流。
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
```

前向引用需要明确写 `short/near`。x86 编码保留长度约束，由布局器处理。

实际上后端能自动处理，但建议X64还是带near，这样后端速度会快一点。

后端默认不是短跳转优先。

符号引用与地址回填

包含符号的指令宽度不固定。

XIRASM 的处理顺序：

1. 源文件定义标号并生成指令片段。
2. 后端逐条编码指令，符号字段暂时未知。
3. 布局器给标号和片段分配最终地址。
4. 地址确定后计算每个标号表达式，回填已编码字段。

前向引用不需要手动计算地址：

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

entry:
    jmp target
    nop

target:
    ret
```

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

动态指令文本与动态标号

字符串/token拼接。

编译时需要计算名字时使用动态指令和标号：

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

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

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

`isa(text)` 编码指令文本，`label.define(name)` 在当前位置创建标号。命名用 `sym.join`，展开用 `sym.unique`。

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

读取标号地址

`label_addr(label_or_name)` 返回标号的逻辑地址：

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

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

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

实用规则

- 文件头明确选择目标模式。
- 指令用自然 ISA 文本。
- 编译期调用末尾加分号，指令末尾不加。
- 源文件中地址明确时用静态标号。
- 前向引用以符号形式传递地址。
- 只在确实需要时用 `isa` 和 `label.define`。
- 判定位宽用 `target.bits`，判定系列用 `target.isa`。
- 布局未稳定时，推迟绝对地址查询到 `defer`。
- 需要加载重定位地址时用格式重定位接口。

下一章介绍数据声明和二进制布局。

[返回目录](#)

第 9 章：数据与二进制布局

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

XIRASM 提供两层：

- 直接写数据的接口
- 定义结构体/联合体的类型系统

几行数据直接用写接口就够了。常用记录格式定义成结构体。

输出内容是有序字节序列

数据在当前写指针位置追加：

```
// 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
```

值超出宽度报错，不静默截断。换更宽的接口。

简写版本：

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

例如：

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

简写接受多个参数。`db` 还能接字符串和字节序列，其他只接整数。1/2/4/6/8 字节检查范围。10/16/32/64 字节零扩展。要精确位模式用 `emit.bytes`。`dt` 是 10 字节原始数据，不是 f80。

浮点值

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

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

`emit.f32` / `emit.f64` 写 IEEE-754 binary32/binary64。类型必须精确匹配。溢出、NaN、Infinity 直接拒。有限下溢和有符号零保留。

字符串与字节序列

一次 `db` 调用可以同时组合字节值、字符串和 `bytes` 值：

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

输出：

```
41 42 43 44 00
```

字符串和字节序列原样复制。XIRASM 不自动加 NUL。要终止符自己写。

`emit.bytes(value)` 写出字符串或 `bytes` 值：

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

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

预留空间

`reserve(count)` 会让输出位置前进指定的字节数：

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

在普通裸二进制文件中，这会写出：

```
eb 00 00 fe
```

预留简写把数量乘以元素宽度：

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

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

这会依次写出两个零字节、`aa`、另外两个零字节和 `bb`。

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

填充与对齐

`pad(count, fill)` 写 `count` 个重复字节：

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

结果为：

```
01 aa aa aa 02
```

填充值实参可以省略，默认值为零：

```
// 使用默认填充值写出四个零字节。  
pad(4);
```

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

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

输出：

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

指定位置不能位于当前输出位置之前。

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

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

这会在 `44` 之前写出五个 `cc` 字节。填充值实参默认也是零。

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

应根据意图选择操作：

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

声明二进制结构体

结构体为一组顺序排列的字段指定名称和类型：

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

自然对齐。tag 从偏移 0 开始。u32 要 4 字节对齐，所以 size 从偏移 4 开始。总共 8 字节：

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

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

如果需要精确的文件布局，应声明紧凑结构体：

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

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

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

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

字段默认值与结构体字面量

结构体的整数字段可以提供编译期默认值：

```

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

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

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

```

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

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

联合体字段不能声明默认值。联合体值必须始终显式选择且只选择一个当前字段。

字段可以使用普通的字段访问语法读取，如示例中的两个写出调用所示。

复合值存在于汇编期间。仅仅声明复合值并不会自动把它写入输出内容。

测量布局

`sizeof(Type)` 返回二进制类型的完整大小：

```

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

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

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

```

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

```

// 选择 64 位 x86 指令编码。
x86.use64();

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

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

```

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

在 x86-64 栈上使用结构体布局

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

```

x86.use64();

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

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

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

```

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

完整的编码测试与可运行 PE64 测试位于 `tests/format/pe64_stack_struct_member_access/`。

打包并写出结构体值

`pack(value)` 会把复合值转换为 `bytes` 值：

```

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

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

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

```

`emit.struct(value)` 会直接完成打包和写出：

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

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

这会写出八个字节，其中自然布局产生的三个填充字节都是零：

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

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

联合体与当前字段

联合体会让多个不同类型的字段重叠在偏移 0：

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

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

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

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

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

联合体还可以嵌套在结构体中：

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

union ValueBits {
    raw: u32
    point: Point
}

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

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

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

如示例中的两个断言所示，`offset_of` 支持嵌套字段路径。

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

选择布局方式

以下情况适合直接写出整数和字节：

- 布局只有少数字段；
- 字段只写出一次，之后不再需要名称；
- 源代码紧密绑在外部格式上，显式调用比类型声明更清晰。

以下情况适合使用紧凑结构体和联合体：

- 精确的二进制记录会被重复使用；
- 字段名称可以提高可读性；
- 其他计算应由 `sizeof` 和 `offset_of` 驱动；
- 值需要默认值、嵌套、比较或转换为 `bytes`。

以下情况适合使用自然布局结构体：

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

外部规范定义的布局用 `sizeof` 和 `offset_of` 验证。断言把格式假设变成可执行检查，改起来也稳。

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

[返回语言指南](#)

第 10 章：模块与文件

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

XIRASM 提供两种独立的文件加载模型：

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

区分这两种角色可使项目的结构更清晰。源文件产生声明或输出操作，数据文件产生供源码检查、变换和写出的值。

当前 I/O 状态： `include/io/` 仍处于开发阶段，目前不是受支持的标准库接口，用户程序不能导入或使用其中的模块。本章介绍的编译期 `fs.*` API 仍可用于读取汇编输入数据。在 `include/io/` 正式声明稳定之前，运行时 I/O 需要由用户自己的 ISA/ABI 例程实现。

导入源模块

`import(path)` 对给定的源文件至多求值一次：

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

emit_word(0x1234);
```

假设 `support.inc` 包含：

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

两次导入指向同一个已解析文件，因此函数只声明一次。示例输出：

```
34 12
```

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

`import` 语句必须位于源文件顶层，不可放置于可能按条件执行的块中：

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

应在顶层选择模块，与目标平台相关的行为应置于模块内部。

内联包含源文件

`include(path)` 每执行到一次就对目标源文件重新求值：

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

如果 `inline.inc` 包含：

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

输出为：

```
aa aa
```

仅在确实需要重复执行时使用 `include`。典型用途包括：

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

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

源文件路径解析

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

对于以下项目：

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

`tables/records.inc` 可以加载相邻文件：

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

该路径相对于 `records.inc`，而非进程工作目录或入口源文件。

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

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

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

选择 `import` 还是 `include`

应根据执行语义选择，而非文件扩展名：

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

项目约定的常见做法：

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

检查与读取数据文件

`fs.exists(path)` 检查数据文件是否可解析：

```
// 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解析模型。当这段代码移入模块时，`payload.bin` 相对于该模块解析。

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

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

如果 `banner.txt` 包含：

```
XIRASM
```

按原样写出文件中的字节，不会自动追加终止零字节。

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

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

读取字节范围

三参数形式的 `fs.read_bytes` 读取带边界的范围：

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

第二个参数是基于零的文件偏移，第三个参数是返回的字节数。

如果 `payload.bin` 包含：

```
10 20 30 40
```

示例输出:

```
20 30
```

所请求的范围必须完整存在。XIRASM 会报告越界读取错误, 而非静默截断。

加载 JSON

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

```
const config: map = json.file("config.json")
const values: list = map.get(config, "values")

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

对于以下输入:

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

输出为:

```
58 52 40 01 02
```

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

```
const raw: string = fs.read_text("config.json")
const config: map = json.parse(raw)
emit.u8(map.get(config, "bits"));
```

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

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

加载 TOML

`toml.file(path)` 为 TOML 提供相同的直接工作流程：

```
const config: map = toml.file("project.toml")
const target: map = map.get(config, "target")

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

对于以下输入：

```
name = "XR"

[target]
bits = 64
```

输出为：

```
58 52 40
```

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

```
const raw: string = fs.read_text("project.toml")
const config: map = toml.parse(raw)
const target: map = map.get(config, "target")

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

TOML 表转换为 map，数组转换为 list，字符串、布尔值和非负整数转换为相应的 Meta 值。浮点数和时间戳值目前不被 Meta 数据转换接受。

区分配置与二进制数据

结构化配置和原始二进制数据解决不同的问题。

使用 JSON 或 TOML 的场景：

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

使用 `fs.read_bytes` 的场景：

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

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

实用的模块组织方式

中等规模的项目可采用以下结构：

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

`main.asm` 导入可复用的定义：

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

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

应保持模块行为的可读性：

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

下一章介绍输出区域与虚拟数据：XIRASM 如何分离逻辑地址、物理文件字节、预留空间和临时布局区域。

[返回目录](#)

第 11 章：输出区域与虚拟数据

汇编里两个基本问题：值在什么地址？字节写在文件什么位置？

flat binary 里两者同步，从零一起涨。结构化二进制里两者分开：代码段加载到高地址，文件偏移可能就在文件头后面；reserve 占地址空间但不写文件；临时拼的数据不进入最终文件。

这章节涉及手戳可执行格式有关,也和链接器基础有关，没兴趣的可不用看。

XIRASM 用输出区域分开这两套值。

地址和文件位置

每个普通区域有四条属性：

- **origin**：标签地址的基准
- **file offset**：字节在文件里的位置
- **logical size**：含预留的逻辑地址范围
- **file size**：实际落盘的字节数

四项互相独立。改 origin 不插入填充，改 file offset 不改标签值。

常用查询：

查询	含义
<code>region_base()</code>	当前区域的 origin
<code>here()</code>	当前逻辑地址
<code>file_offset()</code>	当前文件位置
<code>file_cursor_real()</code>	已写入的文件末端
<code>file_cursor_potential()</code>	假设所有预留均落盘时的文件末端
<code>tail_reserve_size()</code>	当前区域尾部未落盘的预留字节数

光标查询用于构造自定义布局或格式辅助函数。普通代码用标签和 `here()` 就够。

origin 设地址基准

`origin(address)` 设定当前区域的地址起点：

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

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

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

输出一字节：

```
aa
```

origin 改标签地址，不改文件位置。flat 区域里先设 origin 再写出。再次调 `origin` 更新当前区域基准，不创建新区域。

region.begin 指定两个值

`region.begin(name, origin, file_offset)` 同时设逻辑地址和文件偏移：

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

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

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

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

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

header 占文件偏移 0 和 1，payload 从 0x10 开始。中间未写的部分在 flat 输出里补零：

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

开新区域即切换当前输出区。区域之间没有嵌套关系，不需要 end 语句。

区域名描述源码的布局意图，不自动生成目标文件格式的 section 记录——后者由格式库管。

两个值都已知时用 `region.begin`。顺序构造场景用 `output.section` 和 `output.org` 更稳妥，它们自动推导后续文件偏移。

实际位置 vs 预留位置

写已初始化字节时，两个光标同步推进：

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

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

`reserve` 只推潜在光标和逻辑地址，不落盘：

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

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

预留后面出现初始化数据时，预留范围变成文件中间的间隙：

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

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

文件内容：

```
aa 00 00 00 bb
```

`reserve` 延迟落盘的特性让它同时表达两种形态：文件中间的零填充间隙，和不占文件空间的尾部预留。

output.section 裁掉尾部

`output.section(name, origin)` 从当前**实际**光标处开新区域。

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

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

尾部预留增大第一区域的逻辑大小，但新区域从最后一个已写入字节之后开始。输出：

```
41 42
```

适用于运行时需要地址范围但文件末尾不占空间的场景。

`output.section` 只丢弃未落盘的尾部预留，文件中间的间隙不受影响。

output.org 保留间隙

`output.org(name, origin)` 从当前**潜在**光标处开新区域。

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

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

预留之后写入数据，预留范围保留在文件中：

```
41 00 00 00 42
```

区别：

操作	下一个文件偏移起点
<code>output.section</code>	实际光标，裁尾部预留
<code>output.org</code>	潜在光标，保留预留范围

两个操作各自独立选择新的逻辑 origin，与文件位置无关。

region.file_align 对齐文件大小

`region.file_align(alignment)` 对齐当前区域最终的物理文件大小。

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

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

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

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

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

裁掉尾部预留后，三个物理字节按 8 字节边界对齐。第二区域起始文件偏移 8：

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

文件对齐不改逻辑大小——第一区域逻辑大小仍为 16。

对齐值必须是 2 的正整数次幂。调 `region.file_align` 后当前区域的物理输出已关闭，写字节前要先开新区域。

与 `align` 的区别：`align` 推进逻辑位置并在间隙落盘时填充字节；`region.file_align` 只确定文件大小的对齐边界，不改逻辑地址。

虚拟区域：临时工作区

虚拟区域是汇编期间的独立工作区。字节、标签和指令可检查可修改，但不自动汇入最终输出。

`virtual.begin(origin)` 在指定逻辑地址创建虚拟区域：

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

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

virtual.end();

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

临时区域初始内容：

```
44 33 22 11
```

变换后复制到主输出：

```
45 32 23 10
```

虚拟区域本身不贡献文件字节。

虚拟区域里可以正常写出、预留、对齐、定义标签、写 ISA 指令。 `load.*` 和 `store.*` 对其字节同样生效。每个 `virtual.begin` 对应一个 `virtual.end`。

省略 origin：用当前地址

省略 origin 参数时，虚拟区域从外围区域的当前逻辑地址开始：

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

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

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

virtual.end();

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

`scratch` 逻辑地址 0x4001，但其虚拟字节不替换也不推进外围区域。`virtual.end` 后主输出从暂停处继续。最终文件：

```
aa 34 12
```

适用于编码需临时组装、且标签地址需与目标位置共用同一基准的场景。

区域最终数据 defer 才可查

写出阶段只有实时光标可用。区域最终的文件偏移和大小等布局稳定后才确定。以下查询只在收尾处理阶段可用：

查询	对包含指定地址的区域返回
<code>region_file_offset(address)</code>	区域在文件中的基偏移
<code>region_file_size(address)</code>	最终物理大小
<code>region_logical_size(address)</code>	最终逻辑大小

收尾处理中可用这些值填充文件头，或校验自定义格式的大小与最终布局是否吻合。下一章详述收尾处理。

怎么选

只用 `origin`：

- 单个 flat 流需非零地址基准

- 文件和逻辑位置同步推进

`output.section` :

- 下一段从已写字节后开始
- 尾部预留扩内存但不占文件

`output.org` :

- 下一段从潜在位置继续
- 预留范围须保留在文件中

`region.begin` :

- 逻辑 `origin` 和文件偏移都明确
- 自定义二进制布局或格式辅助函数

虚拟区域:

- 临时组装、测量、读取或变换数据
- 临时字节不得自动进入主输出

总结:

- 标签和 `here()` 表逻辑地址
- 实际光标只反映已落盘的字节
- 潜在光标包含当前尾部预留
- 区域最终数据只在收尾处理阶段可读

下一章介绍收尾处理: 读取已稳定的布局信息、修补字节、计算校验和、验证映像, 不改变布局。

[返回目录](#)

第 12 章：收尾处理

二进制文件中的许多字段在首次写入时无法获得最终值。例如：

- 稍后才声明的数据区大小；
- 另一区域的文件偏移；
- 预留区域的最终逻辑大小；
- 对编码指令和已修补修正值的校验和；
- 由头文件后方的源码累加得到的计数。

XIRASM 通过显式的收尾处理阶段处理这些情况，而无需反复执行整个源文件。

该阶段提供两种工具：

- `late_layout` 在映像封存前执行受限的布局变更操作；
- `defer` 在不改变布局的前提下读取并修补已稳定的最终映像。

用户编写的回填操作应位于 `defer` 中。

为回填预留字段

标准流程如下：

1. 写入固定宽度的占位值；
2. 正常写入数据；
3. 在 `defer` 中修补占位字段。

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

最终输出：

```
03 00 41 42 43
```

占位字段已占据两字节，收尾处理仅修改其值，不插入字节，也不移动数据区。

顶层的收尾处理块可在其引用的标签之前声明：

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

当稳定映像可用时，标签已解析完成。

读取最终映像

`load.u8`、`load.u16`、`load.u32` 和 `load.u64` 从已落盘的输出字节中读取小端整数。
`load.bytes(address, count)` 返回字节范围。

这些读取操作可与存储和断言组合使用：

```

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

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

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

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

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

```

完成后的字节：

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

`store.bytes` 接受字符串或 `bytes` 值。整数存储会拒绝超出目标宽度的值。

所有加载和存储均进行范围检查。收尾处理只能访问物理映像中实际存在的字节。位于已裁掉的尾部预留中的逻辑地址不可写，将被拒绝。

计算校验和

收尾处理可声明局部值、更新可变值并使用 `while`：

```

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

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

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

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

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

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

```

校验和为 266（即 0x010a），输出：

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

收尾处理中的循环使用与普通 Meta 循环相同的编译期迭代限制。`for` 目前不允许在收尾处理中使用；字节扫描应使用有界的 `while` 循环。

使用稳定的区域信息

第 11 章介绍的最终区域查询可在 `defer` 中使用：

```

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

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

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

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

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

```

预留尾部计入逻辑大小，但不计入文件大小：

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

这些查询返回包含指定逻辑地址的区域信息：

- `region_file_offset(address)` 返回区域的物理基偏移；
- `region_file_size(address)` 返回最终的实体化大小；
- `region_logical_size(address)` 返回完整的地址空间大小。

它们需要稳定的布局，因此在初始写出阶段不充当实时的游标查询。

调用值函数

返回值的 Meta 函数可用于纯粹的最终计算：

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

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

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

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

最终输出：

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

该函数仅执行表达式运算，不自行写出或修补字节。

可复用的回填过程

过程可以注册收尾处理块并捕获其参数：

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

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

patch_u16(field, 0x1234);
```

调用在普通源码处理过程中执行。其参数值被冻结在收尾处理块中，后者随后写出：

此模式适用于小型的、带类型的回填辅助函数。作用域内的过程参数按值冻结。顶层的收尾处理表达式可保持符号状态直至最终求值。

过程并不是在 `defer` 内部调用；过程调用在更早的阶段注册了收尾处理块。

收尾处理的控制流

`defer` 体内可包含：

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

上述语句中的表达式可使用普通的纯运算符和值函数、标签、`load.*` 以及稳定的区域信息。

每个收尾处理块拥有独立的局部作用域。在一个块中声明的局部变量在另一个块中不可见。

以下操作因会改变布局而被拒绝：

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

相同的限制适用于 ISA 指令、标签、区域切换、对齐、预留、嵌套的收尾处理块、函数声明和源文件加载。

收尾处理的执行顺序

收尾处理块按注册顺序执行：

```
// 先写出一个稍后会被连续修改的字节。
emit.u8(0);

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

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

第二个块可观察到第一个块的修补结果，输出：

```
02
```

应审慎利用此顺序。当两个收尾处理块修改相同的字节时，后执行的块可看到前一个块的结果。

收尾处理块在以下步骤之后执行：

1. 普通源码处理；
2. 指令编码；
3. 后期布局工作；
4. 修正值解析；
5. 最终布局和字节落盘；
6. 修正值修补。

因此，它们看到的是将被实际写入的精确映像，包括已编码的指令和已解析的引用。

必须延迟执行的布局工作

有时源文件必须在主源码完成内容注册后才追加实际字节。这是 `late_layout` 的职责：

```
// 正常源代码先写出第一个字节。
emit.u8(0x10);

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

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

后期布局块按注册顺序执行一次，从默认输出区域的末尾开始。示例输出：

```
10 20 30
```

后期布局在修正值解析和最终布局之前完成。其写出的字节参与最终布局和落盘。

后期布局可将尾部预留实体化

由于 `late_layout` 仍可改变布局，追加的已初始化字节可将先前的尾部预留转化为中间间隙：

```
// 写出一个字节，并在逻辑地址中预留三个字节。  
emit.u8(0xaa);  
reserve(3);  
  
late_layout {  
    // 在预留空间之后追加实际字节，使中间空隙进入输出文件。  
    emit.u8(0xbb);  
}
```

最终输出：

```
aa 00 00 00 bb
```

仅在预留范围确实需要变为物理字节时使用此行为。如果尾部应保持无文件状态，应在注册后续已初始化输出之前切换到另一区域。

组合后期布局与最终回填

收尾处理块可看到后期布局期间追加的字节：

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

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

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

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

稳定后的区域包含四字节字段加四字节数据：

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

这是预期的职责划分：

- `late_layout` 创建须参与布局的字节；
- `defer` 观察最终的布局并修补现有字段。

后期布局的限制

`late_layout` 比普通源码的范围窄。它接受布局和输出 API 调用、诊断和 `if` 选择。它可以：

- 写出整数、字节和结构体；
- 预留、填充和对齐；
- 切换或对齐输出区域；
- 打开和关闭虚拟区域；
- 存储到现有的模块字节中。

它不能声明标签、写出 ISA 指令文本、声明局部值、循环、定义函数、加载源模块，或注册嵌套的后期/收尾处理块。

后期布局仅执行一次。它不是隐式的多遍机制，不应被用于反复收敛不稳定的值。

选择正确的pass阶段

字节可在正常源码顺序中写出时，使用普通源码。

使用 `late_layout` 的场景：

- 必须在主源码完成后追加实际字节或区域；
- 这些字节必须影响最终的偏移和大小；
- 受限的一次性后期布局步骤已足够。

使用 `defer` 的场景：

- 固定宽度的占位字段需要最终值；
- 校验和或验证需要完整的字节映像；
- 需要最终的区域大小或偏移；
- 必须修补现有字节且不改布局。

切勿使用 `defer` 创建缺失的空间。在收尾处理之前预留或写出所需的存储，然后仅修补该现有范围。

下一章进入第三部分，介绍 flat 和自定义二进制文件：将标签、结构体、区域、后期布局和收尾处理组合为完整的文件格式。

第三部分：构建程序

[返回目录](#)

第 13 章：Flat 二进制与自定义文件格式

XIRASM 默认输出 flat binary：文件只有指令和数据落盘后的字节，没有操作系统文件头。

flat binary 的用途：

- 引导扇区、固件映像
- ROM 表、嵌入式资源
- 协议消息、测试数据
- 紧凑的资源容器
- 私有二进制格式
- 被其他程序加载的小段可执行代码

flat binary 内部可以有结构。标签标位置，结构体描述记录，区域分开逻辑地址和物理位置，收尾处理根据完整映像推导字段值。

原始机器码

ISA 指令就是 flat 文件的全部内容。

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

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

输出：

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

没有文件头、没有入口点字段、没有加载器元数据。`entry` 只是汇编标签。加载这些字节的程序必须知道目标 ISA、指令模式、加载地址和调用约定。

用 API 搭文件头

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

payload:

```
// 标号记录数据的逻辑起点，随后写出三个数据字节。  
emit.bytes(b"ABC");
```

输出：

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

前四字节是文件签名，接着两个 `u16` 分别是版本号和保留字段，之后是数据。源文件顺序就是输出顺序，不需要声明格式。

packed struct 对应记录

二进制记录有命名字段、字段间无对齐间隙时用 `packed struct`。

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

输出：

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

字段类型和宽度在源码里一目了然。`emit.struct` 按 packed 布局写。文件格式本身要求相同对齐和尾部填充时才用普通 struct。

重复的写出逻辑封装成函数：

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

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

输出：

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

函数保证写出格式一致，输出顺序仍由源文件顺序决定。

文件头字段 defer 回填

文件头有些字段（大小、偏移、校验和）要等写完才知道。先占位，defer 里回填。

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

magic:
emit.bytes(b"XIF1");
// 先为总大小、数据偏移和校验和各写出一个四字节占位字段。
size_field:
emit.u32(0);
payload_offset_field:
emit.u32(0);
checksum_field:
emit.u32(0);

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

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

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

输出：

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

回填的值从哪来：

- 两端标签的距离 → 标签相减（如 `payload_end - magic`）
- 当前写出位置 → 写出时 `file_offset()`
- 某段内容在文件中的最终位置 → `defer` 里 `region_file_offset(label)`
- 某段内容的最终大小 → `region_file_size(label)`（文件大小）、`region_logical_size(label)`（逻辑大小）

文件偏移 ≠ 逻辑地址

文件连续存字节，但可以给它们不同的逻辑地址。

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

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

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

输出还是 8 字节：

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

数据区在文件偏移 4，逻辑地址却是 0x2000。这在固件、ROM、内存映像和各种需要描述加载位置的格式里很常见。

不要从逻辑地址推文件偏移，除非格式明确规定了关系。两个值各自独立查询。

reserve：中间补零，尾部不占空间

`reserve` 在不同位置行为不同：

- 中间间隙（前后都有数据）→ 实化为零字节
- 区域尾部 → 只增逻辑大小，不写文件

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

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

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

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

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

文件只写 17 字节：

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

中间的 3 字节预留因为后续有数据被实化。尾部 8 字节预留只增逻辑大小（25），不占文件空间。

适合 BSS 段等场景：文件只记录已初始化的前缀，剩余空间由加载器提供。

late_layout 追加尾部

主源码处理完后需要追加字节时用 `late_layout`。

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

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

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

输出：

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

尾部参与最终布局。`defer` 看到的是完整映像，往已分配的文件头字段写值。

仅在尾部确实需要在主源码之后创建时才用 `late_layout`。普通源文件顺序能表达同样布局时更清晰。

断言确认文件正确

自定义写入器应该用断言保证输出可读：

- 签名和版本字段值符合预期
- 偏移指向区域内部
- 计数与写出的记录数一致
- 存的大小与区域最终信息匹配
- 校验和覆盖了正确的字节范围
- 逻辑大小和物理大小符合格式规则

`defer` 里的断言验证实际要写出的字节，包括编码后的指令和已解析的重定位。断言失败停止汇编，不产生文件。

断言紧贴它保护的字段。大小回填和对应的断言通常在同一个 `defer` 块里。

自定义格式的组织顺序

小格式的源码组织顺序：

1. 声明记录类型和常量
2. 写出固定头字段（含占位符）
3. 写出数据区
4. 追加真正需要晚出的表或尾部
5. 回填稳定字段并做断言

可重用的 `include` 封装记录声明和写出函数。格式状态通过参数传递，不在源码里散落硬编码的偏移。

格式变复杂后保持同样分离：

- 源码和函数决定记录内容
- 标签和区域描述布局
- `late_layout` 追加参与最终布局的字节
- `defer` 推导和验证稳定字段

标准格式用库

有 flat 输出能力，不等于应该手拼那些标准可执行或目标文件格式。PE、COFF、ELF 还需要文件头、表、权限、导入导出、重定位和加载规则。

XIRASM 提供了 `format/format.inc` 库处理这些。下一章从语言层面介绍用法。完整说明见《可执行文件格式指南》。

[返回目录](#)

第 14 章：可执行文件与目标文件格式

PE、COFF、ELF 不止是数据加指令。它们还包含文件头、节区段/段表、权限、入口点、导入导出表、符号表、重定位表，由链接器和加载器按规则解析。

XIRASM 用格式库处理这些：

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

格式库让源码描述目标映像，不需要手写表的计数、行位置、文件偏移、虚拟地址和文件头字段。

本章只介绍基本用法。完整的选项、导入导出、重定位、共享库、目标文件和底层辅助函数详见《可执行文件格式指南》。

声明段和权限

格式库程序的第一步：声明映像包含哪些 segment 或 section。每个描述符给出名称、用途和权限。

描述符列表是结构性的： - 长度决定所需表的数量 - 顺序决定格式表的顺序 - 名称标识后续写入的内容块 - 用途选择格式特定的行为 - 权限成为 section/segment 属性

源文件不需要表的计数或行号。增删描述符，格式结构自动更新。

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。
const image0: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(
            ".text",
            format_load | format_readable | format_executable
        )
    )
)
format_begin(image0);

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

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

```

在 x86-64 Linux 下，这个程序以状态码 0 退出。

五步走：1. `format_elf64` 创建 ELF64 计划。2. `format_segment` 声明一个 segment 及其权限。3. `format_begin` 写出计划决定的头部结构。4. `format_segment_begin` 和 `format_segment_end` 包裹 segment 的实际指令和数据。5. `format_entry` 绑定入口标签，`format_finish` 完成映像。

源文件不提供程序头计数、表行、文件偏移、加载地址或入口点字段偏移。格式库从计划表和已完成布局推导这些值。

Section vs Segment

不同格式用不同的术语： - 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)
```

只打开计划中声明过的描述符，每个名称的用途与声明时一致。格式库由此把已写出的字节、逻辑大小、权限和生成的表关联到正确的格式条目。

构造函数系列

格式库支持以下映像类：

映像类	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、NX 策略、地址随机化等。Section/segment 描述符携带用途和读写执行权限。

完整的选项和描述符列表见《可执行文件格式指南》。

入口点绑定

可执行文件在源码定义入口代码后绑定标签：

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

`format_entry` 返回更新后的计划。显式保持返回值让状态变化可见，也让 `format_finish` 能验证入口信息是否完整。

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。

表自动生成

部分格式内容由高级声明生成，不作为普通数据字节写入： - 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

这些功能的格式库 API 接受名称、标签、权限和重定位类型。节区段编号、符号索引、表偏移、行号和计数由库内部推导。

不要用硬编码的表位置替代这些声明。格式库的作用就是让格式的簿记与实际布局保持一致。

普通库就够了

`format/format.inc` 是稳定的用户层，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 文件提供更低层次的构建块。某些暴露特定宽度的便捷封装，某些暴露直接的文件头、section、segment 或表辅助函数。这些适用于实现新的格式库布局，或普通库无法表达的专用布局。

仅仅为了生成常见的多 section 可执行文件、DLL、目标文件或共享库，不需要用底层 include。普通库支持目标映像时就优先用普通库。

不要随意混用抽象层次。底层辅助函数可能要求调用者管理计数、行号、偏移或格式不变式——这些在普通层由库维护。

后续阅读

《可执行文件格式指南》涵盖完整示例： - 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 - 普通库与底层辅助函数的界限

已知所需格式系列时直接查 API Reference。

下一章讲诊断和源文件组织惯例。

[返回目录](#)

第 15 章：诊断与实用惯例

诊断分四级：`print`（信息）、`warn`（警告）、`err`（终止）、`assert`（不变式检查）。源码组织也一样：命名清晰、函数小巧、层次明确。

诊断格式

错误消息带源文件位置、严重级别和说明：

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

所有阶段共用同一错误报告模型：解析、Meta 求值、指令编码、布局、重定位、收尾处理。

先看第一个错。后面的可能是连带出来的。

print 和 warn

`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
```

有意义的汇编时信息用 `print`，不是调试印迹。配置值得注意但输出有效时用 `warn`。条件会导致非法文件或不支持的程序时用 `err`。

err 终止汇编

err 打出致命错误并终止：

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

参数跟在消息后面。把导致失败的值传进去，方便使用者纠正。

消息应说明违反的要求：

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

不要只说 **invalid value**，要说哪个属性不合法。

assert 编码不变式

assert 检查源码中应始终成立的条件：

```
// 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");
```

通过时不产生输出。输出：

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

条件不成立时停止汇编，消息就是错误文本。

源码处理阶段已知的事实直接用 `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

需等映像稳定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 - 指令重定位产生的字节

断言和输入应在同一阶段，免依赖临时布局值。

目标需求就近检查

目标检查放在依赖它的代码旁边：

```
// 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，不生成运行时检查。

显式指定模式后，读者一眼就知道跑在什么 ISA 上，不需要猜是 x86-16、x86-32、x86-64、RISC-V 32 还是 RISC-V 64。

常量代替魔数

格式、协议、布局相关的数值取名字：

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

算法内嵌的数值操作数可以保持局部。偏移、标志、结构大小、格式值、对齐策略这些应该命名。

封闭集合中的选择用命名常量。多个字段组成一个记录用 struct。动态计划用 map 或 list。

命名体现值类型

二进制写入器同时涉及文件偏移、虚拟地址、逻辑距离等多种值。命名体现含义：

```
header_foa
text_rva
entry_address
payload_file_size
payload_logical_size
```

`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，不要用这种泛称。

函数参数也一样。接受 `logical_address` 和 `file_offset` 的函数，比接受 `start` 和 `position` 的容易用对。

用标签查询替代硬编码

- 逻辑距离 → 标签相减
- 结构体布局 → `sizeof`、`offset_of`
- 当前物理位置 → `file_offset()`
- 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

外部格式规范要求精确值时才硬编码。即使硬编码，也要命名并断言周围布局。

defer 和 late_layout 只在必要时用

普通源文件按普通顺序写出。

只有主源码后必须追加的字节才用 `late_layout`。只有检查和回填稳定映像才用 `defer`。

不要把普通写出挪到后期阶段，仅仅因为不想组织源码。一个明确的文件头 + 数据区 + 尾部 + 收尾处理序列，比一张延迟副作用网络好理解。

每个 `defer` 聚焦相关字段。回填文件头并验证的块，比到处改不相关区域的块容易审查。

按职责分模块

项目布局按功能切分： - 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`。

公共函数参数要显式。能传 `plan`、`list`、`map`、`label`、`option` 过去的地方，不要依赖隐藏状态。

标准格式用库

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");
```

格式库管描述符计数、表顺序、索引、偏移和格式不变式。

仅当需要新格式库、或普通库不支持某种布局时，才用底层 `include`。不要因为底层辅助函数暴露了更多数字就换层——那些数字通常是格式库该管的。

项目专用格式用第 13 章的 `flat` 自定义路由，不需要塞进操作系统格式。

在输出边界检查

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

验证描述约定，不重复实现。断言表的计数和声明一致是有用的。断言每一步内部算术则难维护。

三份文档各管各的

本文档（语言指南）管： - 编译时源文件和 ISA 文本如何协作 - 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

《可执行文件格式指南》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` - 普通库与底层辅助函数的界限

三份文档互相独立，各保持可读。

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
- ☐ 预留的尾部和物理间隙是故意的
- ☐ 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
- ☐ 标准格式用格式库，除非有高级需求
- ☐ 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](#)