

中文文档合集

XIRASM 中文文档合集

语言指南、格式教程与 API 参考

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. 集合和文本
8. 07-tokens-and-pattern-matching
9. 08-targets-isa-text-and-labels
10. 数据与二进制布局
11. 10-modules-and-files
12. 输出区域与虚拟数据
13. 12-finalizers
14. Flat 二进制与自定义文件格式
15. 可执行文件与目标文件格式
16. 诊断与实用惯例
17. XIRASM 格式教程
18. 选择模板
19. Windows PE 和 DLL
20. Linux ELF 可执行文件和共享库
21. COFF 和 ELF 目标文件
22. 通用规则和常见错误
23. XIRASM 语言 API 参考
24. 源码、绑定与作用域
25. 函数与流程控制
26. 结构体、联合体与复合值
27. 收尾处理语法形式
28. 模块与诊断信息
29. 目标平台、处理器指令与符号
30. 数据写出、预留空间与对齐
31. 主输出区域、输出区与文件游标
32. 读取、写入与最终区域信息
33. 文本、转换与标号名称
34. 字节序列
35. 列表与映射
36. 文件与结构化数据
37. 词法单元与模式匹配

XIRASM 语言指南

XIRASM 是一款面向 x86、RISC-V 和 SPIR-V 的现代汇编器。处理器指令保持熟悉的汇编写法；值、函数、控制流、集合、数据布局 and 输出格式则由结构化的编译期语言表达。

本指南按照实际编程顺序介绍 XIRASM：先学习值、表达式和函数，再进入标号、数据布局、输出区域与收尾处理，最后构建裸二进制文件和标准可执行格式。需要查询精确签名时使用[语言 API 参考](#)；需要构建 PE、COFF 或 ELF 文件时使用[格式教程](#)。

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。
- 通过格式教程继续学习 PE、COFF、ELF、导入、导出、BSS（未初始化数据区）和重定位。

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格式教程。

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	0x140000000
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 `选择` 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
xirasm module.spvasm -o module.spv --target spv
```

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、RISC-V 和 SPIR-V

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
<code>spv.use()</code>	SPIR-V 1.6 模块

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、RISC-V 与 SPIR-V 使用各自的目标配置，不能把 x86 指令模式套用到其他指令集。

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

```
spv.use();

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

命令行可用 `--target spv` 或 `--target spirv`，两者都选择 SPIR-V 1.6。同一个 SPIR-V 输出只能包含同一 section、同一版本的 SPIR-V 指令，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，暂不接受符号 ID。

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。

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

在指令中使用编译期值

编译期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，必须另外满足调用约定的栈对齐与影子空间要求；`sizeof(StackFrame)` 只描述结构体记录本身。

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

打包并写出结构体值

`pack(value)` 会把复合值转换为 `bytes` 值：

```

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

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

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

```

`emit.struct(value)` 会直接完成打包和写出：

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

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

这会写出八个字节，其中自然布局产生的三个填充字节都是零：

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

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

联合体与当前字段

联合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。

文件 API 的边界： 本章的 `fs.*`、`json.*` 和 `toml.*` 用于编译期读取和处理汇编输入数据。运行时文件操作仍由用户自己的 ISA/ABI 代码负责。

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。

本章说明输出区域、逻辑地址和文件位置的关系。这些概念在构建自定义二进制文件或使用格式接口时都很有用。

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，但不会自动汇入最终输出。

`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
region_file_offset(address)	区域在文件中的基偏移
region_file_size(address)	最终物理大小
region_logical_size(address)	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基本用法；完整模板、参数和 API 摘要见《格式教程》。

[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。模板、选项、导入导出、重定位、共享库和目标文件详见《[格式教程](#)》。需要直接控制格式字段和表项时，再阅读《[高级格式构造指南](#)》。

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。
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，随后完成映像中的文件头和表项。
format_entry_mut(image, start)
format_finish(image);

```

在 x86-64 Linux 下，这个程序以状态码 0 退出。

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_mut` 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 [《格式教程》](#)。

入口点绑定

可执行文件在源码定义入口代码后绑定标签：

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

`format_entry_mut` 直接更新传入的 `let` 配置。`format_finish` 随后检查入口信息并完成文件。

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

《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 - 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。

文档分工

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

《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 - 普通格式接口的参数、调用顺序和常见错误

高级格式构造指南负责需要手动控制文件头、表项、区域和重定位的场景。四份文档彼此独立，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
- ☐ 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](#)

XIRASM 格式教程

这份教程介绍 XIRASM 的常用格式接口。你只需要先确定要生成哪种文件，再按模板声明节、段、导入、导出和重定位。PE、COFF、ELF 的字段计算由格式封装完成。

普通程序统一从这里开始：

```
import("format/format.inc");
```

`include/format/` 提供三层接口：

- `format.inc` 是普通用户入口，负责格式配置、命名节或段以及常用表。
- `pe32.inc`、`pe64.inc`、`elf32.inc`、`elf64.inc` 是兼容用的位宽包装。
- `pe.inc`、`elfexe.inc`、`elfobj.inc` 等提供直接控制，适合标准模板无法表达的文件。

本教程完整说明 `format.inc`。需要直接控制字段和表项时，再阅读[高级格式构造指南](#)。

章节

1. 选择模板
2. Windows PE 和 DLL
3. Linux ELF 可执行文件和共享库
4. COFF 和 ELF 目标文件
5. 通用规则和常见错误

快速选择

输出文件	构造函数	主要接口
Windows 可执行文件	<code>format_pe32</code> 或 <code>format_pe64</code>	<code>format_section_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Windows DLL	<code>format_pe32</code> 或 <code>format_pe64</code> ，选用 <code>format_pe_dll</code>	<code>format_pe_export_pairs_mut</code> 、 <code>format_pe_export_section</code>
Linux 可执行文件	<code>format_elf32</code> 或 <code>format_elf64</code> ，选用 <code>format_elf_exec</code>	<code>format_segment_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Linux PIE	<code>format_elf64</code> ，选用 <code>format_elf_pie</code>	<code>format_segment_begin</code> 、 <code>format_entry_mut</code> 、 <code>format_finish</code>
Linux 共享库	<code>format_elf64_so</code>	<code>format_elfso_tables_mut</code> 、 <code>format_segment_begin</code> 、 <code>format_finish</code>
COFF 目标文件	<code>format_coff32</code> 或 <code>format_coff64</code>	<code>format_coff_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>
ELF 目标文件	<code>format_elfobj32</code> 或 <code>format_elfobj64</code>	<code>format_elfobj_tables_mut</code> 、 <code>format_section_begin</code> 、 <code>format_finish</code>

基本流程

所有普通格式都遵循同一顺序：

1. 用构造函数创建 `let` 格式配置。
2. 按需用 `*_mut` 接口添加导入、导出、符号或重定位。
3. 调用 `format_begin`。
4. 在已声明的节或段中写入代码和数据。
5. 可执行文件用 `format_entry_mut` 设置入口。
6. 调用 `format_finish`。

构造函数返回新值，例如 `format_pe64(...)` 和 `format_section(...)`。名称以 `_mut` 结尾的过程会直接修改传入的 `let` 绑定：

```
let image: map = format_pe64(options, sections)
format_entry_mut(image, start)
format_finish(image);
```

不要把 `const`、临时表达式或字段访问传给 `_mut` 参数。

1. 选择模板

先确定输出文件类型，再选择构造函数。普通用户不需要从文件头字段开始设计。

统一入口

```
import("format/format.inc");
```

只有在标准模板无法表达目标文件时，才直接导入 `pe.inc`、`elfexe.inc` 或 `elfobj.inc`。

构造函数

输出文件	构造函数	内容描述
PE32 可执行文件或 DLL	<code>format_pe32(options, sections)</code>	<code>format_section(...)</code>
PE64 可执行文件或 DLL	<code>format_pe64(options, sections)</code>	<code>format_section(...)</code>
COFF32 目标文件	<code>format_coff32(sections)</code>	<code>format_section(...)</code>
COFF64 目标文件	<code>format_coff64(sections)</code>	<code>format_section(...)</code>
ELF32 可执行文件	<code>format_elf32(options, segments)</code>	<code>format_segment(...)</code>
ELF64 可执行文件或 PIE	<code>format_elf64(options, segments)</code>	<code>format_segment(...)</code>
ELF32 目标文件	<code>format_elfobj32(sections)</code>	<code>format_section(...)</code>
ELF64 目标文件	<code>format_elfobj64(sections)</code>	<code>format_section(...)</code>
ELF64 共享库	<code>format_elf64_so(soname, segments)</code>	<code>format_segment(...)</code>

PE、COFF 和 ELF 目标文件使用节。ELF 可执行文件和共享库使用加载段。

节属性

`format_section(name, attributes)` 接收节名和属性。属性必须包含一个用途，可以再组合权限。

用途	内容
<code>format_code</code>	指令
<code>format_data</code>	已初始化数据或只读数据
<code>format_uninitialized_data</code>	只占内存的未初始化空间
<code>format_imports</code>	PE 导入表
<code>format_exports</code>	PE 导出表
<code>format_resources</code>	PE 资源
<code>format_fixups</code>	PE 基址重定位表

权限	含义
<code>format_readable</code>	可读
<code>format_writeable</code>	可写
<code>format_executable</code>	可执行
<code>format_discardable</code>	加载器处理后可以丢弃

常用组合：

内容	属性
代码	<code>format_code format_readable format_executable</code>
只读数据	<code>format_data format_readable</code>
可写数据	<code>format_data format_readable format_writeable</code>
BSS	<code>format_uninitialized_data format_readable format_writeable</code>
PE 导入	<code>format_imports format_readable format_writeable</code>
PE 导出	<code>format_exports format_readable</code>
PE 资源	<code>format_resources format_readable</code>
PE 重定位	<code>format_fixups format_readable format_discardable</code>

段属性

`format_segment(name, attributes)` 用于 ELF 加载段。属性使用 `format_load` 加权限：

内容	属性
代码	<code>format_load \</code> <code>format_readable \</code> <code>format_executable</code>
只读数据	<code>format_load \</code> <code>format_readable</code>
可写数据或 BSS	<code>format_load \</code> <code>format_readable \</code> <code>format_writeable</code>

完整流程

```
import("format/format.inc");

let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_auto,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
    )
)
format_begin(image);

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

format_section_begin(image, ".bss");
    rb(64);
format_section_end(image, ".bss");

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

格式封装会根据命名节推导头部、表项、文件位置、RVA、对齐和 BSS 大小。

2. Windows PE 和 DLL

PE 配置需要指定文件角色、子系统、内存保护策略、地址随机化策略和节列表。

PE 选项

`format_pe32(options, sections)` 和 `format_pe64(options, sections)` 必须各选一个文件角色、子系统和 ASLR 策略。`format_pe_nx` 可以按需加入。

分组	选项	作用
文件角色	<code>format_pe_exe</code>	可执行文件
文件角色	<code>format_pe_dll</code>	DLL
子系统	<code>format_pe_console</code>	控制台程序
子系统	<code>format_pe_gui</code>	图形界面程序
内存保护	<code>format_pe_nx</code>	标记镜像兼容不可执行数据页
ASLR	<code>format_pe_aslr_auto</code>	存在重定位数据时启用 ASLR
ASLR	<code>format_pe_aslr_required</code>	必须提供重定位数据并启用 ASLR
ASLR	<code>format_pe_aslr_disabled</code>	禁用 ASLR

常用节：

节名	属性
<code>".text"</code>	<code>format_code</code> \ <code>format_readable</code> \ <code>format_executable</code>
<code>".data"</code>	<code>format_data</code> \ <code>format_readable</code> \ <code>format_writeable</code>
<code>".bss"</code>	<code>format_uninitialized_data</code> \ <code>format_readable</code> \ <code>format_writeable</code>
<code>".idata"</code>	<code>format_imports</code> \ <code>format_readable</code> \ <code>format_writeable</code>
<code>".edata"</code>	<code>format_exports</code> \ <code>format_readable</code>
<code>".rsrc"</code>	<code>format_resources</code> \ <code>format_readable</code>
<code>".reloc"</code>	<code>format_fixups</code> \ <code>format_readable</code> \ <code>format_discardable</code>

最小 PE64 可执行文件

```
import("format/format.inc");

let image: map = format_pe64(
  format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_auto,
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
  )
)
format_begin(image);

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

format_section_begin(image, ".bss");
  rb(64);
format_section_end(image, ".bss");

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

导入 Windows API

先创建导入集合，再按 DLL 成组添加 API。`format_pe_import_pairs_mut` 的 `pairs` 参数按“本地槽名、导入名”重复排列。

```

import("format/format.inc");

let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_auto,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".idata", format_imports | format_readable | format_writeable)
    )
)

let imports: map = format_pe_import_new()
// 同名批量接口一次加入同一个 DLL 的多个 API。
format_pe_import_many_mut(
    image,
    imports,
    "KERNEL32.DLL",
    list.of("ExitProcess", "GetCurrentProcessId")
)
// 再次调用即可加入另一个 DLL; pairs 还能指定本地槽名。
format_pe_import_pairs_mut(
    image,
    imports,
    "ADVAPI32.DLL",
    list.of("close_registry_key", "RegCloseKey")
)
format_begin(image);

format_section_begin(image, ".text");
start:
    sub rsp, 40
    call [rel GetCurrentProcessId]
    xor ecx, ecx
    call [rel ExitProcess]
format_section_end(image, ".text");

format_pe_import_section(image, ".idata", imports);
format_entry_mut(image, start)
format_finish(image);

```

API	参数	作用
<code>format_pe_import_new()</code>	无	创建空导入集合
<code>format_pe_import_many_mut(plan, imports, dll, names)</code>	配置、导入集合、DLL、名称列表	API 名与本地槽名相同
<code>format_pe_import_pairs_mut(plan, imports, dll, pairs)</code>	配置、导入集合、DLL、槽名/导入名列表	自定义本地槽名
<code>format_pe_import_section(plan, name, imports)</code>	配置、导入节名、导入集合	生成 <code>.idata</code>

每个 DLL 调用一次批量接口，并持续更新同一个 `imports`，最终生成的 `.idata` 就会包含全部 DLL 和 API。PE64 通过 `call [rel slot_name]` 调用导入函数；PE32 使用 `call [slot_name]`。

导出 DLL 函数

`format_pe_export_many_mut` 批量导出同名标签；`format_pe_export_pairs_mut` 的 `pairs` 参数按“目标标签、公开名称”重复排列。

```

import("format/format.inc");

let image: map = format_pe64(
  format_pe_dll | format_pe_console | format_pe_nx | format_pe_aslr_auto,
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".edata", format_exports | format_readable)
  )
)

let exports: list = format_pe_export_new()
// 两个标签直接使用原名导出。
format_pe_export_many_mut(image, exports, list.of("x_add7", "x_sub3"))
// answer_impl 对外使用 x_answer 这个名称。
format_pe_export_pairs_mut(image, exports, list.of("answer_impl", "x_answer"))
format_begin(image);

format_section_begin(image, ".text");
dll_main:
  // 最小 DLL 入口返回 TRUE。
  mov eax, 1
  ret
x_add7:
  lea eax, [ecx + 7]
  ret
x_sub3:
  lea eax, [ecx - 3]
  ret
answer_impl:
  mov eax, 42
  ret
format_section_end(image, ".text");

format_pe_export_section(image, ".edata", exports, "xirasm_demo.dll");
format_entry_mut(image, dll_main)
format_finish(image);

```

API	作用
<code>format_pe_export_new()</code>	创建空导出集合
<code>format_pe_export_many_mut(plan, exports, names)</code>	批量导出同名标签
<code>format_pe_export_pairs_mut(plan, exports, pairs)</code>	批量映射目标标签和公开名称
<code>format_pe_export_section(plan, name, exports, dll_name)</code>	生成 <code>.edata</code>

资源

先把 `.rsrc` 声明为 `format_resources`，再把编译好的 `.res` 文件交给封装：

```
format_pe_resource_section(image, ".rsrc", "data/app.res");
```

API	参数	作用
<code>format_pe_resource_section(plan, name, path)</code>	配置、已声明的资源节名、 <code>.res</code> 路径	写入编译后的资源树并登记 PE 资源目录

基址重定位

写入绝对地址和声明基址重定位是两件事。下面是可直接汇编的 PE64 模板：

```

import("format/format.inc");

let image: map = format_pe64(
    format_pe_exe | format_pe_console | format_pe_nx | format_pe_aslr_required,
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".data", format_data | format_readable | format_writeable),
        format_section(".idata", format_imports | format_readable | format_writeable),
        format_section(".reloc", format_fixups | format_readable | format_discardable)
    )
)

let imports: map = format_pe_import_new()
format_pe_import_many_mut(image, imports, "KERNEL32.DLL", list.of("ExitProcess"))
format_begin(image);

format_section_begin(image, ".text");
start:
    // 为两次 Windows x64 调用预留 shadow space 并对齐栈。
    sub rsp, 40
    // 指令通过相对地址访问指针槽。
    mov rax, [rel worker_pointer]
    call rax
    // 用 ExitProcess 明确结束示例程序。
    mov ecx, eax
    call [rel ExitProcess]
worker:
    mov eax, 42
    ret
format_section_end(image, ".text");

format_section_begin(image, ".data");
worker_pointer:
    // 文件中保存的是绝对指针，因此这个槽需要基址重定位。
    dq(0);
format_section_end(image, ".data");

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

let relocs: list = pe_reloc_new()
format_pe_reloc_add_mut(image, relocs, worker_pointer)
format_pe_reloc_section(image, ".reloc", relocs);

format_entry_mut(image, start)
format_finish(image);

// 标签地址稳定后再写入最终绝对值。
defer {
    store.u64(worker_pointer, worker);
}

```

`format_pe_reloc_add_mut` 的地址参数是“保存指针的槽”，不是指针指向的目标。它会根据 PE32 或 PE64 选择重定位类型，记录必须按 RVA 升序传给 `format_pe_reloc_section`。`rel` 指令引用本身是相对位移，不需要基址重定位。PE32 对应使用 `dd(0)` 和 `store.u32`。

API	参数	作用
<code>pe_reloc_new()</code>	无	创建空重定位列表
<code>format_pe_reloc_add_mut(plan, relocs, storage)</code>	配置、列表、指针存储地址	添加与 PE 位宽匹配的基址重定位
<code>format_pe_reloc_section(plan, name, relocs)</code>	配置、已声明的重定位节名、已排序列表	生成 <code>.reloc</code> 并登记数据目录

校验和

校验和依赖最终文件内容，因此在 `format_finish` 之后调用：

```
format_pe_checksum(image);
```

`format_pe_checksum(plan)` 接收已经完成的 PE 配置，并根据最终文件内容回填校验和字段。

3. Linux ELF 可执行文件和共享库

普通接口用命名加载段描述 ELF 镜像。文件位置、虚拟地址、文件大小、内存大小和权限由格式配置推导。

ELF 可执行文件

固定地址可执行文件使用 `format_elf_exec` :

```
import("format/format.inc");

let image: map = format_elf64(
    format_elf_exec,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable),
        format_segment(".data", format_load | format_readable | format_writeable),
        format_segment(".bss", format_load | format_readable | format_writeable)
    )
)
format_begin(image);

format_segment_begin(image, ".text");
start:
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image, ".text");

format_segment_begin(image, ".data");
answer:
    dd(42);
format_segment_end(image, ".data");

format_segment_begin(image, ".bss");
scratch:
    rb(128);
format_segment_end(image, ".bss");

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

ELF32 使用 `format_elf32(format_elf_exec, segments)`。

ELF64 PIE

位置无关可执行文件使用 `format_elf_pie`：

```
import("format/format.inc");

let image: map = format_elf64(
    format_elf_pie,
    list.of(
        format_segment(".text", format_load | format_readable | format_executable),
        format_segment(".bss", format_load | format_readable | format_writeable),
        format_segment(".rodata", format_load | format_readable)
    )
)
format_begin(image);

format_segment_begin(image, ".text");
start:
    // 加载器改变 PIE 基址后，这两个标签引用仍然有效。
    lea rbx, [rel scratch]
    lea rsi, [rel message]
    mov dword [rbx], 0x5a
    mov eax, 60
    xor edi, edi
    syscall
format_segment_end(image, ".text");

format_segment_begin(image, ".bss");
scratch:
    rb(64);
format_segment_end(image, ".bss");

format_segment_begin(image, ".rodata");
message:
    db("XIRASM PIE", 0);
format_segment_end(image, ".rodata");

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

文件格式是位置无关的，指令也必须遵守目标 ISA 的位置无关规则。x86-64 访问本镜像内标签时使用 `rel`，它编码相对位移，不需要绝对动态重定位。PIE 或共享库中保存的绝对指针需要动态重定位；普通接口不提供任意用户指针重定位，这类需求属于直接 ELF 层。普通接口暂不提供 ELF32 PIE。

ELF64 可执行文件导入

当前普通接口为固定地址 ELF64 可执行文件生成 PLT/GOT 动态导入。PIE 不使用这条导入流程。

```

import("format/format.inc");

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

let imports: list = format_elfexe_import_new()
// 同一个库一次加入多个 API。
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("getpid", "getppid"))
// 再加入另一个库，并把 cos 的本地前缀改成 cos_fn。
format_elfexe_import_pairs_mut(imports, "libm.so.6", list.of("cos_fn", "cos"))
format_elfexe_tables_mut(image, imports)
format_begin(image);

format_segment_begin(image, ".text");
start:
    call getpid_plt
    call getppid_plt
    xor edi, edi
    mov eax, 60
    syscall
format_segment_end(image, ".text");

format_entry_mut(image, start)
format_finish(image);

```

每个库调用一次批量接口，并持续更新同一个 `imports`。`format_elfexe_import_many_mut` 会为同名导入生成 `<name>_gotplt` 和 `<name>_plt`；别名导入 `cos_fn` 对应 `cos_fn_gotplt` 和 `cos_fn_plt`。

ELF64 共享库

共享库没有普通可执行入口。它通过动态符号表导出符号，也可以声明导入。

```

import("format/format.inc");

let image: map = format_elf64_so(
    "libxirasm_demo.so",
    list.of(
        format_segment(".text", format_load | format_readable | format_executable)
    )
)

let exports: list = format_elfso_export_new()
// 两个四字节函数直接使用标签名导出。
format_elfso_export_many_mut(exports, list.of("x_add7", "x_sub3"), ".text", 4)
// answer_impl 对外使用 x_answer。
format_elfso_export_pairs_mut(exports, list.of("answer_impl", "x_answer"), ".text", 6)
format_elfso_tables_mut(image, exports, list.new())
format_begin(image);

format_segment_begin(image, ".text");
x_add7:
    lea eax, [rdi + 7]
    ret
x_sub3:
    lea eax, [rdi - 3]
    ret
answer_impl:
    mov eax, 42
    ret
format_segment_end(image, ".text");

format_finish(image);

```

ELF64 共享库导入

共享库也可以把多个库的导入收集到同一个列表：

```

import("format/format.inc");

let image: map = format_elf64_so(
    "libxirasm_report.so",
    list.of(
        format_segment(".text", format_load | format_readable | format_executable),
        format_segment(".data", format_load | format_readable | format_writeable)
    )
)

let exports: list = format_elfso_export_new()
format_elfso_export_pairs_mut(exports, list.of("report_impl", "x_report"), ".text", 18)

let imports: list = format_elfso_import_new()
// libc 中的两个 API 使用同名本地标签。
format_elfso_import_many_mut(imports, "libc.so.6", list.of("puts", "getpid"))
// 第二个库把 cos 映射到本地前缀 cos_fn。
format_elfso_import_pairs_mut(imports, "libm.so.6", list.of("cos_fn", "cos"))
format_elfso_tables_mut(image, exports, imports)

format_begin(image);

format_segment_begin(image, ".text");
report_impl:
    lea rdi, [rel message]
    call puts_plt
    call getpid_plt
    ret
format_segment_end(image, ".text");

format_segment_begin(image, ".data");
message:
    db("XIRASM shared object", 0);
format_segment_end(image, ".data");

format_finish(image);

```

即使示例没有调用 `cos_fn_plt`，该标签和 `cos_fn_gotplt` 仍会生成。本镜像内的数据使用 `rel` 引用，外部函数则通过生成的 PLT 标签调用。

API 摘要

API	作用
<code>format_elf32(format_elf_exec, segments)</code>	ELF32 可执行文件
<code>format_elf64(format_elf_exec, segments)</code>	ELF64 固定地址可执行文件
<code>format_elf64(format_elf_pie, segments)</code>	ELF64 PIE
<code>format_elf64_so(soname, segments)</code>	ELF64 共享库
<code>format_elfexe_import_new()</code>	创建可执行文件导入集合
<code>format_elfexe_import_many_mut(imports, library, names)</code>	批量添加同名 PLT/GOT 导入
<code>format_elfexe_import_pairs_mut(imports, library, pairs)</code>	批量映射本地名称和导入名
<code>format_elfexe_tables_mut(plan, imports)</code>	把导入集合附加到可执行文件配置
<code>format_elfso_export_new()</code>	创建共享库导出集合
<code>format_elfso_export_many_mut(exports, names, segment, size)</code>	批量导出同名标签
<code>format_elfso_export_pairs_mut(exports, pairs, segment, size)</code>	批量映射目标标签和公开名称
<code>format_elfso_import_new()</code>	创建共享库导入集合
<code>format_elfso_import_many_mut(imports, library, names)</code>	批量添加共享库导入
<code>format_elfso_import_pairs_mut(imports, library, pairs)</code>	批量映射本地名称和导入名
<code>format_elfso_tables_mut(plan, exports, imports)</code>	附加共享库动态符号信息

4. COFF 和 ELF 目标文件

目标文件交给链接器继续处理。普通接口需要三类信息：节、链接器可见符号，以及需要链接器修补的位置。

COFF 目标文件

```

import("format/format.inc");

let object: map = format_coff64(
    list.of(
        format_section(".text", format_code | format_readable | format_executable),
        format_section(".data", format_data | format_readable | format_writeable),
        format_section(".bss", format_uninitialized_data | format_readable |
format_writeable)
    )
)
format_begin(object);

format_section_begin(object, ".text");
text_start:
main:
    db(0xe8);
call_disp:
    dd(0);
    xor eax, eax
    ret
format_section_end(object, ".text");

format_section_begin(object, ".data");
data_start:
answer:
    dd(42);
format_section_end(object, ".data");

format_section_begin(object, ".bss");
bss_start:
scratch:
    rb(64);
format_section_end(object, ".bss");

const symbols: list = list.of(
    format_coff_public("main", ".text", text_start, main, coff_sym_type_function),
    format_coff_public("answer", ".data", data_start, answer, coff_sym_type_null),
    format_coff_public("scratch", ".bss", bss_start, scratch, coff_sym_type_null),
    format_coff_extern("puts", coff_sym_type_function)
)

const relocs: list = list.of(
    format_coff_reloc(".text", text_start, call_disp, "puts", coff_rel_amd64_rel32)
)
format_coff_tables_mut(object, symbols, relocs)
format_finish(object);

```

32 位相对调用使用 `coff_rel_i386_rel32`，64 位使用 `coff_rel_amd64_rel32`。

ELF 目标文件

```
import("format/format.inc");

let object: map = format_elfobj64(
  list.of(
    format_section(".text", format_code | format_readable | format_executable),
    format_section(".bss", format_uninitialized_data | format_readable |
format_writeable),
    format_section(".rodata", format_data | format_readable)
  )
)
format_begin(object);

format_section_begin(object, ".text");
text_start:
_start:
  db(0xe8);
call_disp:
  dd(0);
  xor eax, eax
  ret
format_section_end(object, ".text");

format_section_begin(object, ".bss");
bss_start:
scratch:
  reserve(64);
format_section_end(object, ".bss");

format_section_begin(object, ".rodata");
data_start:
answer:
  dd(42);
format_section_end(object, ".rodata");

const symbols: list = list.of(
  format_elfobj_public("_start", ".text", text_start, _start, 8, elfobj_stt_func),
  format_elfobj_public("scratch", ".bss", bss_start, scratch, 64, elfobj_stt_object),
  format_elfobj_public("answer", ".rodata", data_start, answer, 4,
elfobj_stt_object),
  format_elfobj_extern("puts", elfobj_stt_func)
)

const relocs: list = list.of(
  format_elfobj_reloc(".text", text_start, call_disp, "puts", elf_r_x86_64_plt32,
0xfffffffffffffffffc)
)
format_elfobj_tables_mut(object, symbols, relocs)
format_finish(object);
```

ELF32 使用 `format_elfobj32`。32 位相对调用常用 `elf_r_386_pc32`。

API 摘要

格式	API	作用
COFF	<code>format_coff32(sections)</code> / <code>format_coff64(sections)</code>	创建目标文件配置
COFF	<code>format_coff_public(...)</code>	定义公开符号
COFF	<code>format_coff_extern(...)</code>	声明外部符号
COFF	<code>format_coff_reloc(...)</code>	描述重定位字段
COFF	<code>format_coff_tables_mut(plan, symbols, relocs)</code>	附加符号和重定位集合
ELF	<code>format_elfobj32(sections)</code> / <code>format_elfobj64(sections)</code>	创建目标文件配置
ELF	<code>format_elfobj_public(...)</code>	定义公开符号及大小
ELF	<code>format_elfobj_extern(...)</code>	声明外部符号
ELF	<code>format_elfobj_reloc(...)</code>	描述带 addend 的重定位字段
ELF	<code>format_elfobj_tables_mut(plan, symbols, relocs)</code>	附加符号和重定位集合

先在代码中写入占位字段，再用 `format*_reloc` 描述它。链接器会根据重定位记录修补这些字节。

5. 通用规则和常见错误

只使用一层接口

普通程序统一导入：

```
import("format/format.inc");
```

不要同时使用普通格式配置和手写底层表项。需要完全控制格式时，改用高级格式构造指南中的 `direct` 层。

配置必须使用 `let`

构造函数返回配置； `_mut` 过程直接修改配置或集合：

```
let image: map = format_elf64(format_elf_exec, segments)
let imports: list = format_elfexe_import_new()
format_elfexe_import_many_mut(imports, "libc.so.6", list.of("getpid"))
format_elfexe_tables_mut(image, imports)
format_entry_mut(image, start)
```

传给 `_mut` 的目标必须是直接 `let` 绑定，不能是 `const` 或临时表达式。

先声明，再写内容

传给 `format_section_begin`、`format_section_end`、`format_segment_begin` 和 `format_segment_end` 的名字必须已经在配置中声明。节名和段名不能重复。

每个描述只选一个用途

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

`format_code` 和 `format_data` 不能同时用于同一个节。用途只选一个，权限可以组合。

权限按运行需求设置

内容	常用权限
指令	可读、可执行
常量和字符串	可读
可修改数据	可读、可写
BSS	可读、可写
PE 导入表	可读、可写
PE 重定位表	可读、可丢弃

除非确实需要自修改代码，否则不要让代码可写；数据也不应具有执行权限。

BSS 不写入初始化文件内容

在 `format_uninitialized_data` 中使用 `rb(...)` 或 `reserve(...)`。它们增加内存大小，但不会把同样数量的零字节写进文件。

重定位和地址值是两件事

```
absolute_slot:
    dq(0);

let relocs: list = pe_reloc_new()
format_pe_reloc_add_mut(image, relocs, absolute_slot)
format_pe_reloc_section(image, ".reloc", relocs);

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

`store.u64` 写入地址值；重定位记录告诉加载器这个位置需要随镜像基址调整。

入口只属于可执行文件

PE 和 ELF 可执行文件使用 `format_entry_mut(plan, label)`。COFF、ELF 目标文件和 ELF 共享库不使用这条入口流程。

defer 只做最终回填

`defer` 可以检查最终字节、回填地址和写校验和，但不能生成会改变布局的新内容。需要参与布局的字节必须在 `format_finish` 前写入正常节或段。

从最小模板逐步增加功能

1. 先完成只有 `.text` 的文件。
2. 再加入 `.data` 或 `.bss`。
3. 需要系统函数时加入导入。
4. 构建 DLL 或共享库时加入导出。
5. 文件保存绝对地址时加入基址重定位。
6. 需要链接器继续处理时选择 COFF 或 ELF 目标文件模板。

XIRASM 语言 API 参考

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

可执行文件与目标文件格式使用单独的文档：

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

本参考手册不收录 `format_*` 过程和格式系列专用辅助接口。

阅读约定

语法形式使用以下记号：

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

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

内容结构

第一部分：核心语言

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

第二部分：汇编器操作

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

第三部分：编译期辅助库

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

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

第 1 章：源码、绑定与作用域

语法摘要

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

标号

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

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

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

处理器指令

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

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

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

常量

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

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

不能向常量赋值：

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

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

变量

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

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

赋值不会声明新名称。目标绑定必须已经存在，而且必须可变。

代码块与名称遮蔽

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

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

代码块结束后，其中声明的绑定不再存在：

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

代码块适合限制临时值的可见范围、隔离辅助计算，并明确表示名称遮蔽。

跨行表达式参数

只要调用表达式的圆括号尚未闭合，就可以跨越多行继续书写：

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

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

完整示例

下面的源代码组合了本章介绍的七种形式：

```

// 外层 value 是不可变绑定。
const value = 1

// 内层 value 是只在代码块中存在的可变绑定。
{
    let value = 2
    value = value + 1
    assert(value == 3)
}

// 跨行检查外层绑定仍然保持原值。
assert(
    value == 1
)

// 定义标号并写出指令，最后写出外层 value 的一个字节。
start:
    mov rax, 1

emit.u8(value)

```

源代码会写出：

```

b8 01 00 00 00 01

```

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

选用指南

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

第 2 章：函数与流程控制

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

语法摘要

形式	语法	用途
过程函数	<code>fn name(parameters) { statements }</code>	封装编译期操作。
过程调用	<code>name(arguments)</code>	执行过程函数。
返回值函数	<code>fn name(parameters) -> type { statements }</code>	计算表达式值。
返回	<code>return expression</code>	结束返回值函数并给出结果。
返回值调用	<code>name(arguments)</code>	在表达式中使用返回值函数。
条件分支	<code>if condition { statements }</code>	布尔条件为真时执行代码块。
备选分支	<code>} else { statements }</code>	执行备选代码块。
条件循环	<code>while condition { statements }</code>	布尔条件保持为真时重复执行。
数值范围循环	<code>for name in range(start, end) { statements }</code>	迭代左闭右开的数值范围。
列表循环	<code>for name in list_value { statements }</code>	按顺序迭代列表中的值。

过程函数

```
// 定义一个过程函数，连续写出输入值及其后继值。
fn emit_pair(value: u8) {
    emit.u8(value)
    emit.u8(value + 1)
}

// 分别调用两次，写出两组相邻字节。
emit_pair(2)
emit_pair(8)
```

没有 `-> type` 的函数是过程函数。过程函数执行操作，但不产生表达式值。函数体可以写出数据或指令、改变布局、声明局部值、使用流程控制，以及调用其他过程函数。

参数按位置对应。类型标注可以省略；如果提供了类型标注，调用时会用它检查对应实参。每次调用都必须为每个参数提供且只提供一个实参。

过程调用是一条语句。末尾可以写分号，但不要求写。

返回值函数

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

// 计算对齐后的值，并将 16 位结果写入输出内容。
emit.u16(align_up(0x73, 0x20))
```

带有 `-> type` 的函数是返回值函数。它的调用属于表达式，可以出现在声明、条件、实参、`return` 或更大的表达式中。

`return` 会计算表达式，并把结果转换为声明的返回类型。返回值函数中每条实际执行的路径都必须到达 `return`。

返回值函数用于辅助计算。它可以使用局部绑定、流程控制和其他返回值函数，但不得写出输出内容，也不得执行其他会改变布局的操作。

函数声明与调用规则

- 函数必须在顶层声明。
- 函数声明必须出现在首次调用之前。
- 同一声明中的参数名称不得重复。
- 每次调用都有各自的参数和局部绑定。
- 过程调用不能用作表达式值。
- `return` 只能出现在返回值函数中。
- 函数调用链最多允许 128 层。

只要递归返回值函数能在达到调用深度上限前结束，就允许使用递归。如果操作本身适合迭代，应使用循环。

`if` 与 `else`

```
// enabled 为 false, 因此只执行备选分支。
const enabled = false

if enabled {
    emit.u8(0x11)
} else {
    emit.u8(0x22)
}
```

条件必须计算为 `bool`。只有选中的分支会执行。每个分支都有自己的词法作用域。

备选分支应写成规范的 `} else if condition {` 和 `} else {` 形式。`else if` 和最后的 `else` 都可以省略。

`if` 语句在汇编期间选择源码操作。要实现运行时条件行为，仍需写出处理器分支指令。

while

```
// 每轮写出当前值，再更新控制循环的可变绑定。
let value = 1

while value <= 3 {
    emit.u8(value)
    value = value + 1
}
```

`while` 会在每次迭代前计算布尔条件。如果条件一开始就是假，循环体不会执行。

`break` 会结束最内层的活动 Meta 循环。`continue` 会跳过循环体中剩余的语句并开始下一次迭代。它们也可以用于 `for` 循环和延迟执行的 `while` 代码块。编译期循环最多执行 1,000,000 次。在循环外使用循环控制语句，或试图让它跨越函数调用边界，均属于错误。

数值范围迭代

```
// 迭代左闭右开的范围 [0, 4)，依次写出四个字节。
for index in range(0, 4) {
    emit.u8(index)
}
```

`range(start, end)` 包含 `start`，不包含 `end`。此示例写出 `00 01 02 03`。

数值范围只能向前移动，也可以为空。降序范围无效：

```
for value in range(2, 0) {
    emit.u8(value)
}
```

循环绑定是只读的局部值。每次迭代都会获得新的绑定和新的循环体作用域。

列表迭代

```
// 按列表顺序写出三个预先选定的操作码字节。
const opcodes: list = list.of(0x90, 0x90, 0xc3)

for opcode in opcodes {
    emit.u8(opcode)
}
```

列表循环按列表顺序访问各个值。循环绑定只在本次迭代中有效，不能对它赋值。

这种形式只直接接受列表值。要迭代映射内容，应先用 `map.keys(...)` 或 `map.values(...)` 等映射辅助函数取得列表。

无效的函数形式

过程函数不能返回值：

```
fn invalid() {
    return 1
}

invalid()
```

返回值函数不能在未返回结果的情况下执行到函数末尾：

```
fn invalid() -> u64 {
    const value = 1
}

const result = invalid()
```

返回值函数不能写出输出内容：

```
fn invalid() -> u64 {  
    emit.u8(1)  
    return 1  
}  
  
const result = invalid()
```

操作会改变输出内容或布局时，应使用过程函数。调用方需要计算结果时，应使用返回值函数。

完整示例

```

// 过程函数负责写出一对相邻字节。
fn emit_pair(value: u8) {
    emit.u8(value)
    emit.u8(value + 1)
}

// 返回 value 与 limit 中较小的值。
fn choose(value: u64, limit: u64) -> u64 {
    if value > limit {
        return limit
    } else {
        return value
    }
}

// 写出过程函数结果和返回值函数结果。
emit_pair(1)
emit.u8(choose(9, 5))

// 编译期条件选择第一个源码分支。
const enabled = true

if enabled {
    emit.u8(0xaa)
} else {
    emit.u8(0xff)
}

// 条件循环写出两个连续值。
let counter = 0

while counter < 2 {
    emit.u8(0x10 + counter)
    counter = counter + 1
}

// 数值范围循环写出 0x20 和 0x21。
for index in range(0, 2) {
    emit.u8(0x20 + index)
}

// 列表循环按顺序写出列表中的两个值。
const values: list = list.of(0x30, 0x31)

for value in values {
    emit.u8(value)
}

```

源代码会写出：

选用指南

需求	使用形式
封装会改变输出内容或布局的操作	过程函数
计算可复用的表达式值	返回值函数
选择一个代码块	<code>if</code>
在两个代码块中选择一个	<code>if / else</code>
在多个代码块中选择一个	<code>if / else if / else</code>
重复执行，直到可变状态满足条件	<code>while</code>
迭代固定的数值范围	搭配 <code>range</code> 的 <code>for</code>
迭代列表中的已知值	搭配列表的 <code>for</code>
提前结束最内层循环	<code>break</code>
跳过当前迭代的剩余部分	<code>continue</code>

第 3 章：结构体、联合体与复合值

结构体和联合体用于描述二进制布局。它们的值会在汇编期间存在，直到被打包成字节或写入输出内容。

语法摘要

形式	用途
<code>struct Name { fields }</code>	声明自然布局结构体。
<code>packed struct Name { fields }</code>	声明不含填充的结构体。
<code>union Name { fields }</code>	声明自然布局联合体。
<code>packed union Name { fields }</code>	声明不含尾部填充的联合体。
<code>Name { field: value }</code>	构造复合值。
<code>value.field</code>	读取复合值的字段。
<code>emit.struct(value)</code>	打包复合值并立即写出。
<code>sizeof(Type)</code>	返回类型的二进制大小。
<code>offset_of(Type, field_path)</code>	返回字段偏移。
<code>pack(value)</code>	将复合值转换为 <code>bytes</code> 。

自然布局结构体与紧凑结构体

```

// 自然布局会按字段的对齐要求插入填充。
struct NaturalHeader {
    tag: u8
    size: u32
}

// 紧凑布局让字段连续排列，不插入填充。
packed struct PackedHeader {
    tag: u8
    size: u32
}

// 对比两种布局的总大小和 size 字段偏移。
assert(sizeof(NaturalHeader) == 8)
assert(offset_of(NaturalHeader, size) == 4)
assert(sizeof(PackedHeader) == 5)
assert(offset_of(PackedHeader, size) == 1)

```

自然布局结构体会按照每个字段的类型对齐，并将最终大小向上取整到结构体的对齐要求。字段之间和结构体末尾都可能出现填充。

紧凑结构体会把每个字段紧接在前一个字段之后，不添加尾部填充。需要精确描述文件记录和协议记录时使用紧凑布局；记录中的字段需要对齐时使用自然布局。

紧凑布局只移除填充，不会清除复合类型自身的对齐属性。紧凑复合类型仍保留最大字段对齐，因此自然布局的外层复合类型会按照该边界放置嵌套的紧凑字段。

同一个声明中的字段名称必须唯一。

字段默认值与结构体字面量

```

// magic 使用声明中的字段默认值，tail 由结构体字面量指定。
packed struct Header {
    magic: u16 = 0x4241
    tail: u16
}

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

// 读取默认字段，并写出完整的紧凑结构体。
assert(header.magic == 0x4241)
emit.struct(header)

```

结构体字面量按名称为字段赋值。字面量中的书写顺序不必与声明顺序一致。

省略的结构体整数字段会使用声明的字段默认值。省略任何其他字段都会报错。字面量中出现未知字段或重复字段也会报错。联合体字段不能声明默认值。

复合值字面量可以直接传给内置表达式：

```
// 构造 Pair 后立即打包，并把所得两个字节写入输出内容。
packed struct Pair {
    low: u8
    high: u8
}

emit.bytes(pack(Pair {
    low: 3,
    high: 4
})))
```

自然布局联合体与紧凑联合体

```
// ThreeBytes 的紧凑布局大小为三个字节。
packed struct ThreeBytes {
    tag: u8
    value: u16
}

// 自然布局联合会按最大字段的对齐要求取整最终大小。
union NaturalValue {
    bytes: ThreeBytes
    word: u16
}

// 紧凑联合体只保留最大字段的精确大小。
packed union PackedValue {
    bytes: ThreeBytes
    word: u16
}

assert(sizeof(NaturalValue) == 4)
assert(sizeof(PackedValue) == 3)
```

联合体的每个字段都从偏移零开始。自然布局联合体采用最大字段的大小，并把最终大小向上取整到最大字段的对齐要求。紧凑联合体则采用最大字段的精确大小。

联合体字面量必须恰好选择一个当前字段：

```
// word 是这个联合体值唯一的当前字段。  
packed union Value {  
    byte: u8  
    word: u16  
}  
  
const value: Value = Value {  
    word: 0x1234  
}  
  
// 按 word 字段的声明宽度打包并写出联合体。  
emit.struct(value)
```

联合体初始化时不指定字段，或指定多个字段，都是无效写法。联合体字段声明不能提供默认值；字面量必须始终显式选择当前字段。

嵌套复合值字面量

```

// Point 将两个坐标连续存放。
packed struct Point {
    x: u16
    y: u16
}

// ValueBits 的当前字段可以是原始整数，也可以是 Point。
union ValueBits {
    raw: u32
    point: Point
}

packed struct Record {
    kind: u8
    value: ValueBits
}

// 在 Record 中逐层构造联合体和结构体字段。
const record: Record = Record {
    kind: 1,
    value: ValueBits {
        point: Point {
            x: 0x1122,
            y: 0x3344
        }
    }
}

// 点分字段路径会累加每一层的字段偏移。
assert(offset_of(Record, value.point.y) == 3)
emit.struct(record)

```

嵌套的复合值字段可以接受其声明类型对应的嵌套字面量。`offset_of` 接受点分字段路径，并累加各层的字段偏移。

字段访问

// 构造头部后，通过字段访问分别读取两个值。

```
packed struct Header {  
    magic: u16  
    size: u32  
}  
  
const header: Header = Header {  
    magic: 0x5a4d,  
    size: 0x40  
}
```

// 按字段各自的声明宽度写出读取结果。

```
emit.u16(header.magic)  
emit.u32(header.size)
```

字段访问会从已经保存的复合值中读取一个值。字段名称必须存在于该值的声明类型中。

sizeof 与 offset_of

```
sizeof(Type)  
offset_of(Type, field_path)
```

sizeof 返回完整的二进制大小，其中包括自然布局产生的填充。

offset_of 返回直接字段或嵌套字段的字节偏移。它不需要复合值；这两个操作查询的都是声明类型的布局。

pack

```
pack(value) -> bytes
```

pack 会创建包含复合值二进制表示的字节序列。整数字段会按照其声明宽度编码。自然布局中的填充字节为零。

需要检查、比较、保存这些字节，或把它们传给另一个函数时，使用 **pack**。

emit.struct

```
emit.struct(value)
```

`emit.struct` 会打包复合值，并把所得字节写入当前输出区域。它等价于写出 `pack` 的结果，但不会公开中间的 `bytes` 值。

无效的复合值字面量

以下形式都会被拒绝：

```
const unknown: Pair = Pair {  
    missing: 1  
}
```

```
const duplicate: Pair = Pair {  
    low: 1,  
    low: 2  
}
```

```
const incomplete: Pair = Pair {  
    low: 1  
}
```

```
const invalid_union: Value = Value {  
    byte: 1,  
    word: 2  
}
```

这些错误依次表示：未知字段、重复字段、缺少没有默认值的字段，以及联合体存在多个当前字段。

完整示例

```

// 自然布局头部包含字段对齐和尾部填充。
struct NaturalHeader {
    tag: u8 = 0x41
    size: u32 = 0x11223344
}

// 紧凑头部连续存放 tag 和 tail。
packed struct PackedHeader {
    tag: u8 = 0x42
    tail: u16
}

packed struct Point {
    x: u16
    y: u16
}

// ValueBits 让原始整数和坐标值共享存储空间。
union ValueBits {
    raw: u32
    point: Point
}

packed struct Record {
    kind: u8
    value: ValueBits
}

// 三字节字段用于演示联合体最终大小的布局差异。
packed struct ThreeBytes {
    tag: u8
    value: u16
}

union NaturalOdd {
    bytes: ThreeBytes
    word: u16
}

packed union PackedOdd {
    bytes: ThreeBytes
    word: u16
}

// 使用全部字段默认值，并写出自然布局结构体及其布局信息。
const natural: NaturalHeader = NaturalHeader { }
emit.struct(natural)
emit.u8(sizeof(NaturalHeader))
emit.u8(offset_of(NaturalHeader, size))
emit.u32(natural.size)

// 为 tail 指定值，tag 继续使用字段默认值。

```

```

emit.bytes(pack(PackedHeader { tail: 0x4443 })))

// 嵌套构造 Record, 并选择 point 作为联合体的当前字段。
const record: Record = Record {
    kind: 1,
    value: ValueBits {
        point: Point {
            x: 0x1122,
            y: 0x3344
        }
    }
}

emit.bytes(pack(record))

// 选择 ThreeBytes 作为紧凑联合体的当前字段。
const odd: PackedOdd = PackedOdd {
    bytes: ThreeBytes {
        tag: 0x55,
        value: 0x7766
    }
}

// 写出联合体内容、两种联合体大小以及 Record 的 kind 字段。
emit.bytes(pack(odd))
emit.u8(sizeof(NaturalOdd))
emit.u8(sizeof(PackedOdd))
emit.u8(record.kind)

```

源代码会写出：

```

41 00 00 00 44 33 22 11 08 04 44 33 22 11
42 43 44 01 22 11 44 33 55 66 77 04 03 01

```

选用指南

需求	使用形式
按字段自然对齐，并对最终大小取整	<code>struct</code>
精确的连续字节布局	<code>packed struct</code>
让字段重叠，并按自然对齐要求确定最终大小	<code>union</code>
让字段重叠，并采用最大字段的精确大小	<code>packed union</code>
查询声明的布局	<code>sizeof</code> 或 <code>offsetof</code>
以值的形式取得复合值字节	<code>pack</code>
立即写出复合值字节	<code>emit.struct</code>

第 4 章：收尾处理语法形式

XIRASM 提供两个有意分开的后期处理阶段：

- `late_layout` 在最终输出内容确定之前执行受限的布局变更；
- `defer` 在布局确定后读取并回填最终字节映像，但不会改变布局。

不要把这两种形式互换使用。

阶段顺序

阶段	允许承担的职责
普通源码	声明标号、写出内容、预留空间并登记后期代码块。
指令编码	编码普通处理器指令片段。
<code>late_layout</code>	一次性追加或重新组织受限的布局内容。
地址回填与布局	解析引用，并计算最终的逻辑位置和物理位置。
生成最终字节映像	创建最终字节映像，并应用已经解析的地址回填项。
<code>defer</code>	读取、验证和回填已经生成的现有字节。

每一种代码块都按照登记顺序执行。

`late_layout`

```
// 普通源码先写出第一个字节。
emit.u8(0x10)

late_layout {
    // 第一个后期布局块追加第二个字节。
    emit.u8(0x20)
}

late_layout {
    // 第二个后期布局块继续按登记顺序追加字节。
    emit.u8(0x30)
}
```

这个示例会写出 `10 20 30`。

后期布局块在普通源码写出和指令编码结束后执行一次，但早于地址回填项解析和最终布局。它写出的数据会参与最终偏移和大小的计算。

`late_layout` 接受：

- 输出与布局 API 调用；
- 整数、字节和复合值写出；
- 预留、补齐和对齐调用；
- 输出区域与虚拟区域调用；
- 向模块中已有内容执行写入；
- 诊断与断言；
- `if` 和 `else`。

它不接受局部声明、赋值、循环、标号、处理器指令、函数声明、源码加载、`defer` 或另一个 `late_layout`。

后期布局是一次性阶段，不是隐式的多轮处理机制。

预留尾部与后期布局

在同一个区域的预留尾部之后追加已初始化数据，会使预留空隙成为文件中的实际字节：

```
// 先写出一个字节，再预留两个只占逻辑空间的字节。
emit.u8(0xaa)
reserve(2)

late_layout {
    // 在预留尾部之后追加数据，使中间空隙进入实际输出。
    emit.u8(0xbb)
}
```

输出为 `aa 00 00 bb`。如果预留尾部必须只保留逻辑空间，请改用另一个输出区域。

`defer`

```

// 让当前输出区域从逻辑地址 0 开始。
origin(0)

// 为最终文件大小预留一个 16 位字段。
size_field:
emit.u16(0)

// 写出两字节正文。
payload:
emit.bytes(b"AB")

defer {
    // 布局稳定后回填区域文件大小，并读取结果进行验证。
    store.u16(size_field, region_file_size(size_field))
    assert(load.u16(size_field) == 4)
}

```

输出为 `04 00 41 42`。

收尾块在最终布局、最终字节映像生成和地址回填项写入完成后执行。它看到的正是即将写入文件的准确字节映像。

收尾块可以包含：

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

表达式可以使用纯计算运算符、返回值函数、标号、`load.*` 和稳定的区域信息。

每个收尾块都有自己的局部作用域。循环最多执行 1,000,000 次。

收尾块内的局部计算

```

// 使用固定逻辑起点，并为校验和预留两个字节。
origin(0)

checksum_field:
emit.u16(0)

// 写出需要参与校验和计算的数据。
payload:
emit.bytes(b"ABCD")

defer {
    // 使用局部可变值逐字节扫描最终映像。
    let cursor = payload
    let checksum = 0
    const end = region_base() + region_file_size(payload)

    while cursor < end {
        checksum = checksum + load.u8(cursor)
        cursor = cursor + 1
    }

    // 把累加结果回填到已有字段。
    store.u16(checksum_field, checksum)
}

```

`let` 会建立可变的收尾块局部状态。赋值用于更新这些状态，`while` 则可以对已经完成的映像执行有界扫描和汇总计算。

从过程函数登记收尾块

```

// 定义一个登记 16 位回填操作的过程函数。
fn patch_u16(address: u64, value: u64) {
    defer {
        // 收尾块保存调用时传入的地址和值。
        store.u16(address, value)
    }
}

// 先写出占位字段，再登记稍后执行的回填。
field:
emit.u16(0)

patch_u16(field, 0x1234)

```

输出为 `34 12`。

过程函数调用发生在普通源码处理期间。从过程函数作用域捕获的值会为收尾块固定下来。过程函数不会从 `defer` 内部调用。

收尾块执行顺序

```
// 先写出一个会被两个收尾块连续修改的字节。
origin(0)
emit.u8(0)

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

defer {
    // 第二个收尾块读取前一次修改，再把数值增加 1。
    store.u8(0, load.u8(0) + 1)
}
```

输出为 `02`。后登记的收尾块可以观察到先登记代码块完成的回填。

收尾处理限制

`defer` 不能创建字节、标号、区域、对齐或预留空间：

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

在一个后期处理阶段中再嵌套另一个后期处理阶段同样无效：

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

每个回填目标都必须在收尾阶段之前写出或预留。写入操作只能访问已经生成的实际字节。经过裁剪的预留尾部虽然具有逻辑范围，但没有可以回填的实际字节：

```
origin(0)
emit.u8(0x11)
reserve(1)

defer {
    store.u8(1, 0x22)
}
```

这个写入操作会被拒绝，不会越过实际生成的字节范围写入数据。

完整示例

```

// 为大小和校验和分别预留两个 16 位字段。
origin(0)

size_field:
emit.u16(0)

checksum_field:
emit.u16(0)

// 普通源码先写出正文的前两个字节。
payload:
emit.bytes(b"AB")

late_layout {
    // 后期布局追加会参与最终大小和校验和计算的字节。
    emit.bytes(b"CD")
}

defer {
    // 扫描完整正文并计算校验和。
    let cursor = payload
    let checksum = 0
    const end = region_base() + region_file_size(payload)

    while cursor < end {
        checksum = checksum + load.u8(cursor)
        cursor = cursor + 1
    }

    // 回填两个文件头字段，并验证最终字节映像。
    store.u16(size_field, region_file_size(size_field))
    store.u16(checksum_field, checksum)

    assert(load.u16(size_field) == 8)
    assert(load.u16(checksum_field) == 0x010a)
    assert(load.bytes(payload, 4) == b"ABCD")
}

```

源代码会写出：

```
08 00 0a 01 41 42 43 44
```

`late_layout` 会添加正文最后两个字节。随后，`defer` 看到完整的八字节映像，计算校验和，并回填两个已经存在的文件头字段。

选用指南

需求	使用形式
写出普通源码内容	普通源码
在偏移和大小确定之前追加实际字节	<code>late_layout</code>
在布局完成后回填宽度固定的占位字段	<code>defer</code>
根据最终字节计算校验和	使用 <code>load.*</code> 和局部状态的 <code>defer</code>
验证最终偏移、大小或内容	使用 <code>assert</code> 的 <code>defer</code>

第 5 章：模块与诊断信息

语法摘要

API	语法	结果
包含源码	<code>include(path)</code>	每次调用都执行解析后的源码。
导入源码	<code>import(path)</code>	解析后的源码只执行一次。
提示	<code>print(value, ...)</code>	记录不会终止汇编的提示信息。
警告	<code>warn(value, ...)</code>	记录不会终止汇编的警告信息。
错误	<code>err(value, ...)</code>	记录错误并停止汇编。
断言	<code>assert(condition[, message])</code>	条件为假时停止汇编。

源码路径

传给 `include` 和 `import` 的 `path` 参数必须求值得到文本。相对路径从包含该调用的源文件所在位置开始解析，因此一个被加载的文件可以继续按自己的目录加载其他文件。

解析后的源码路径同时也是模块标识。两种不同写法只要解析到同一个源文件，就表示同一次导入。

重复包含源码

每次调用 `include(path)` 都会执行被加载的源码：

```
include("row.inc")
include("row.inc")
```

如果 `row.inc` 写出一个字节，这个字节就会写出两次。重复包含也会重复执行声明，因此多次包含声明了同一个函数或类型的文件，可能产生重复声明错误。

需要有意重复展开源码时，或者源码片段中的声明能够安全地在每个调用位置执行时，可以使用 `include`。

模块只导入一次

`import(path)` 只在第一次导入时执行被加载的源码：

```
import("library.inc")
import("library.inc")
```

第二次调用不会产生任何效果。因此，定义函数、类型、常量或可复用数据的文件通常应使用 `import`。

导入是模块顶层操作。在词法作用域的代码块或函数内部调用 `import` 是无效的。遇到递归的包含或导入链时，汇编器会拒绝整个循环，而不会只执行其中一部分。

诊断信息

`print`、`warn` 和 `err` 可以接收一个或多个值：

```
// 设置逻辑原点，再记录当前位置、警告和断言信息。
origin(0)

print("offset", here())
warn("diagnostic example", true)
assert(here() == 0, "unexpected origin")

// 写出一个字节，证明提示和警告不会停止汇编。
emit.u8(0x5a)
```

各个值会使用常规文本形式格式化，并以一个空格连接。上面的示例会记录：

```
note: offset 0
warning: diagnostic example true
```

`print` 和 `warn` 不会停止汇编。`err` 会在调用位置记录错误并停止汇编：

```
err("unsupported width", 24)
```

诊断 API 也可以在 `defer` 收尾块中使用。模块加载 API 不能在收尾处理中调用。

断言

`assert` 接收一个布尔条件和一条可选消息：

```
// 检查位宽是否有效，再把对应的字节数写入输出内容。  
const width = 64  
assert(width == 64)  
assert(width % 8 == 0, "width must use whole bytes")  
  
emit.u8(width / 8)
```

条件为真时，`assert` 不会写出内容，也不会记录诊断信息。条件为假时，汇编会停止并显示提供的消息。省略消息时，诊断文本为 `assertion failed`。

完整模块示例

下面四个文件共同演示重复包含、只导入一次、嵌套相对路径和诊断信息。

`main.asm` :

```
origin(0)  
  
include("repeat.inc")  
include("repeat.inc")  
  
import("module/once.inc")  
import("module/once.inc")  
  
print("module bytes", here())  
warn("diagnostic example", true)  
assert(here() == 4, "unexpected module output")  
  
emit.u8(0x44)
```

`repeat.inc` :

```
emit.u8(0x11)
```

`module/once.inc` :

```
include("nested.inc")  
emit.u8(0x22)
```

`module/nested.inc` :

```
emit.u8(0x33)
```

最终输出为：

```
11 11 33 22 44
```

`repeat.inc` 会执行两次。`module/once.inc` 只执行一次，其中嵌套的 `include` 会相对于 `module` 目录解析路径。

第 6 章：目标平台、处理器指令与符号

语法摘要

形式	语法	结果
目标平台宽度	<code>target.bits</code>	返回当前 x86 指令模式或 RISC-V XLEN。
目标平台系列条件	<code>target.isa == .name</code>	在 <code>if</code> 中检查当前指令集系列。
x86 模式	<code>x86.use16()</code> 、 <code>x86.use32()</code> 、 <code>x86.use64()</code>	选择 x86 及其指令模式。
RISC-V 模式	<code>riscv.use32()</code> 、 <code>riscv.use64()</code>	选择 RISC-V 及其 XLEN。
SPIR-V 模块	<code>spv.use()</code>	选择 SPIR-V 1.6 完整模块输出。
动态处理器指令	<code>isa(text)</code>	根据文本值追加一条指令。
逻辑原点	<code>origin(address)</code>	设置当前输出区域的基地址。
当前位置	<code>here()</code>	返回当前逻辑地址。
动态标号	<code>label.define(name)</code>	根据文本值定义标号。
标号地址	<code>label_addr(name)</code>	返回标号或绝对符号的地址。

目标平台查询

`target.bits` 是整数表达式。对于 x86，它表示当前指令模式；对于 RISC-V，它表示当前 XLEN：

```
// 仅在当前目标平台系列为 x86 时检查并写出模式位数。
if target.isa == .x86_64 {
    assert(target.bits == 64)
    emit.u8(target.bits)
}
```

SPIR-V 没有适用于整个目标平台的整数位宽，因此该目标平台不能使用 `target.bits`。

`target.isa` 是目标平台条件形式，而不是通用值表达式。请在源码 `if` 中把它放在 `==` 或 `!=` 左侧：

```

if target.isa == .x86_64 { ... }
if target.isa != .riscv64 { ... }
if target.isa == .spirv { ... }

```

可用的系列字面量是 `.x86_64`、`.riscv64` 和 `.spirv`。系列名称不会随宽度变化：x86 的 16 位、32 位和 64 位模式都使用 `.x86_64`，RISC-V 的 XLEN 32 和 XLEN 64 都使用 `.riscv64`。

选择指令模式

模式过程会选择后续处理器指令片段保存的目标平台。已经记录的指令仍然保留先前的目标平台。

调用	后续处理器指令
<code>x86.use16()</code>	x86 16 位模式
<code>x86.use32()</code>	x86 32 位模式
<code>x86.use64()</code>	x86-64 模式
<code>riscv.use32()</code>	XLEN 为 32 的 RISC-V
<code>riscv.use64()</code>	XLEN 为 64 的 RISC-V
<code>spv.use()</code>	SPIR-V 1.6 模块文本

// 依次记录使用 16 位、32 位和 64 位模式的 x86 指令。

```

x86.use16()
isa("mov ax, 0x1234")

```

```

x86.use32()
isa("mov eax, 0x12345678")

```

```

x86.use64()
isa("mov rax, 0x0102030405060708")

```

// 随后切换到两种 RISC-V XLEN，并分别记录一条指令。

```

riscv.use32()
isa("addi x1, x0, 1")

```

```

riscv.use64()
isa("addi x2, x0, 2")

```

这些过程是普通源码操作。它们不会改变早先指令的含义，也不是收尾处理操作。

`spv.use()` 开始一个完整的 SPIR-V 模块。SPIR-V 指令使用标准 `Op*` 拼写和数字结果 ID：

```
spv.use()

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

整个输出只能包含同一 section、同一版本的 SPIR-V 指令片段。混入其他 ISA、数据写出、预留或对齐操作都会被拒绝。SPIR-V 结果 ID 目前必须写成 `%1`、`%2` 等数字形式，暂不接受符号 ID。命令行的 `--target spv` 和 `--target spirv` 都选择 SPIR-V 1.6。

动态处理器指令

`isa(text)` 只接受一个非空的单行文本值：

```
// 编译期绑定提供需要追加的单行处理器指令。
const instruction = "nop"
isa(instruction)
```

指令已经固定写在源码中时，应使用普通处理器指令行；编译期逻辑生成指令文本时，才使用 `isa`。汇编器会按照调用位置当前生效的目标平台解析并编码这条指令。

普通处理器指令不使用编译期语言的语句结束符。不要在普通指令行末添加 `;`，也不要把它放入传给 `isa` 的字符串：

```
nop;           // 无效的处理器指令文本。
isa("nop;")    // 无效的处理器指令文本。
```

空字符串、其他类型的值、包含换行符的文本或以分号结尾的文本都无效。

逻辑原点与当前位置

`origin(address)` 会改变当前输出区域的逻辑基地址。它不会写出字节、插入填充，也不会改变文件中的物理位置。

`here()` 返回：

```
active region origin + current logical offset
```

```
// 设置逻辑原点，再验证写出两个字节后的当前位置和标号地址。
origin(0x4000)

header:
emit.u16(0x1234)

assert(here() == 0x4002)
assert(label_addr(header) == 0x4000)
```

请在定义依赖新基地址的地址之前调用 `origin`。普通源码和后期布局都可以调用它，但 `defer` 中不能调用。

静态标号与动态标号

源码标号定义固定名称：

```
entry:
```

`label.define(name)` 根据文本值定义同一种锚定标号：

```
// 在新的逻辑原点定义动态标号，并验证它锚定在写出字节之前。
origin(0x5000)

const generated_name = "generated_entry"
label.define(generated_name)
emit.u8(0xaa)

assert(label_addr(generated_name) == 0x5000)
```

生成的名称必须是有效的标号标识符。 `label.define` 是普通源码操作，并且要求已经打开输出区域。

`label_addr` 既可以接受直接书写的标号名称，也可以接受文本表达式：

```
label_addr(entry)
label_addr("entry")
label_addr(generated_name)
```

未知符号会被拒绝。值绑定不是标号，不能通过 `label_addr` 获得地址。

处理器指令编码期间的地址稳定性

数据写出、预留和对齐操作会立即更新普通源码游标。处理器指令则先被记录，稍后再编码。处理器指令之后的标号会锚定到对应源码位置，并在指令编码和分支长度调整完成后获得最终偏移。

如果标号依赖前面的处理器指令长度，不要把普通 `label_addr` 的结果直接写死到数据中。应先写出固定宽度的占位值，再在 `defer` 中回填：

```
// 先为最终地址预留固定宽度字段。
origin(0x4000)

target_field:
emit.u64(0)

    jmp done
done:
    ret

// 指令长度稳定后，把 done 的最终逻辑地址写回占位字段。
defer {
    store.u64(target_field, label_addr(done))
}
```

执行 `defer` 时，布局和指令长度都已经稳定，因此 `label_addr(done)` 是最终逻辑地址。这个规则也适用于位于处理器指令之后的动态标号。

第 7 章：数据写出、预留空间与对齐

语法摘要

形式	语法	结果
固定宽度整数	<code>emit.u8(value)</code> 至 <code>emit.u64(value)</code>	按小端顺序写出一个整数。
固定宽度浮点	<code>emit.f32(value)</code> 、 <code>emit.f64(value)</code>	按小端顺序写出一个 IEEE-754 值。
数据序列	<code>db / dw / dd / dp / dq / dt / ddq / dq / d</code> <code>dqq</code>	写出 1 至 64 字节宽度的整数元素。
字节序列	<code>emit.bytes(value)</code>	原样写出字符串或 <code>bytes</code> 值。
文件字节	<code>emit.file(path, offset?, count?)</code>	写出完整的源文件相对文件或精确范围。
按字节预留	<code>reserve(count)</code>	预留 <code>count</code> 个逻辑字节。
按元素预留	<code>rb / rw / rd / rp / rq / rt / rdq / rqq / r</code> <code>dqq</code>	预留 1 至 64 字节宽度的元素。
固定填充	<code>pad(count, fill?)</code>	恰好写出 <code>count</code> 个填充字节。
绝对位置填充	<code>pad_to(position, fill?)</code>	持续写出字节，直到活动输出区域到达 <code>position</code> 。
对齐	<code>align(boundary, fill?)</code>	前进到下一个对齐位置。

固定宽度整数写出

`emit.u*` 过程接受一个无符号整数，并按小端顺序写出：

调用	接受的数值范围	写出字节数
<code>emit.u8(value)</code>	0 至 0xff	1
<code>emit.u16(value)</code>	0 至 0xffff	2
<code>emit.u32(value)</code>	0 至 0xffffffff	4
<code>emit.u64(value)</code>	0 至 0xffffffffffffffff	8

```
// 依次写出 1、2、4、8 字节宽度的整数，以展示各宽度的小端顺序。  
emit.u8(0x11)  
emit.u16(0x2233)  
emit.u32(0x44556677)  
emit.u64(0x8899aabbccddeeff)
```

输出内容的开头为：

```
11 33 22 77 66 55 44 ff ee dd cc bb aa 99 88
```

超出所选宽度数值范围的值会被拒绝，而不会被截断。

数据序列

数据写出简写接受一个或多个实参：

调用	元素宽度	接受的实参
<code>db(values...)</code>	1 字节	整数、字符串和 <code>bytes</code> 值
<code>dw(values...)</code>	2 字节	整数
<code>dd(values...)</code>	4 字节	整数
<code>dp(values...)</code>	6 字节	整数
<code>dq(values...)</code>	8 字节	整数
<code>dt(values...)</code>	10 字节	从 <code>u64</code> 零扩展的整数
<code>ddq(values...)</code>	16 字节	从 <code>u64</code> 零扩展的整数
<code>dqq(values...)</code>	32 字节	从 <code>u64</code> 零扩展的整数
<code>ddqq(values...)</code>	64 字节	从 <code>u64</code> 零扩展的整数

```
// db 原样组合整数字节、字符串和字节序列；其余调用按固定宽度写出整数。  
db(0x10, "AZ", b"BC")  
dw(0x1234, 0xabcd)  
dd(0x10203040)  
dq(0x0102030405060708)
```

传给 `db` 的字符串和字节序列会逐字节复制。宽度更大的简写只接受整数，并按小端顺序编码每个元素。调用数据写出简写时不提供实参是无效的。

小于八字节的宽度会检查范围且绝不截断。 `dt` 是 10 字节原始数据，不是 `f80` 或 x87 值。精确的宽位模式应使用 `emit.bytes`。

浮点写出

带小数部分或指数的十进制字面量属于 `f64`。 `f32(value)` 显式窄化有限的 `f64`； `f64(value)` 扩宽 `f32` 或保留 `f64`。

```
const compact: f32 = f32(1.5)
emit.f32(compact)
emit.f64(-0.0)
```

`emit.f32` 和 `emit.f64` 要求实参类型完全匹配，并以小端顺序写出 IEEE-754 位模式。语言不提供隐式整数转换或 `f80` 表面。NaN、Infinity 和溢出会被拒绝；有限下溢与有符号零会被保留。

字节序列

`emit.bytes(value)` 接受一个字符串或 `bytes` 值：

```
// 保存三个原始字节，再与字符串一起原样写入输出内容。
const signature = b"XIR"
emit.bytes(signature)
emit.bytes("ASM")
```

该调用不会添加终止符、长度字段，也不会执行编码转换。如果一次调用需要组合文本、字节序列和整数形式的字节值，请使用 `db`。

预留空间

`reserve(count)` 会让逻辑位置前进 `count` 个字节。以下简写使用固定宽度元素表示数量：

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

```
// 在两个已初始化字节之间预留三个逻辑字节。
db(0xaa)
reserve(3)
db(0xbb)

// 使用两个 8 字节元素定义尾部预留空间，并验证其逻辑大小。
tail_start:
rq(2)
tail_end:

assert(tail_end - tail_start == 16)
```

预留空间后继续写出已初始化内容时，中间的间隙会以零填充。位于输出区域末尾的预留空间可以只保留为逻辑空间，而不向文件添加字节。第 8 章会介绍决定这一区别的实际文件游标和潜在文件游标。

预留空间的算术运算会经过检查。如果按元素宽度换算后的字节数无法表示，该数量会被拒绝。

填充与对齐

填充始终写出已初始化字节。可选的填充值默认为零，并且必须能放入一个字节。

```
// 依次按固定数量、绝对位置和边界执行填充与对齐，再写出末尾数据。  
db(0x11, 0x22)  
pad(2, 0xaa)  
pad_to(8, 0xbb)  
align(16, 0xcc)  
db(0x33)
```

`pad(count, fill)` 恰好写出 `count` 个字节。

`pad_to(position, fill)` 把 `position` 视为相对于活动输出区域的文件位置。计算填充量时，从下一个需要实际写出的输出位置开始，因此前面的预留空间只会计算一次。目标位置不能位于该起点之前。

`align(boundary, fill)` 会让逻辑位置和文件位置前进到 `boundary` 的下一个整数倍。边界必须是非零的二次幂。如果当前位置已经对齐，则不会写出任何字节。

需要逻辑上的未初始化空间时，应使用预留空间；当填充字节属于文件格式本身时，应使用填充或对齐。

第 8 章：主输出区域、输出区与文件游标

语法摘要

形式	语法	结果
明确主输出区域	<code>region.begin(name, origin, file_offset)</code>	在明确的逻辑地址和文件位置开始一个主输出区域。
文件大小对齐	<code>region.file_align(boundary)</code>	对齐主输出区域的实际文件大小，并结束该区域。
潜在位置续接	<code>output.org(name, origin)</code>	从前一个潜在文件游标开始新的主输出区。
实际位置续接	<code>output.section(name, origin)</code>	从前一个实际文件游标开始新的主输出区。
虚拟输出区	<code>virtual.begin([origin])</code>	开始一个具有逻辑地址但不产生文件字节的嵌套输出区。
结束虚拟输出区	<code>virtual.end()</code>	返回 <code>virtual.begin</code> 之前处于活动状态的输出区。
区域逻辑原点	<code>region_base()</code>	返回当前主输出区域的逻辑基地址。
当前文件位置	<code>file_offset()</code>	返回当前主输出区域的绝对实际文件游标。
实际文件游标	<code>file_cursor_real()</code>	返回当前实际文件字节之后的下一个绝对位置。
潜在文件游标	<code>file_cursor_potential()</code>	返回由逻辑游标推导出的绝对文件位置。
尾部预留空间	<code>tail_reserve_size()</code>	返回当前主输出区域尾部尚未写入文件的逻辑字节数。

坐标模型

主输出区域分别维护逻辑坐标和文件坐标：

坐标	含义
逻辑地址	标号、 <code>here()</code> 、指令和地址回填项使用的地址。
实际文件游标	当前输出内容中已经存在的字节之后的绝对文件位置。
潜在文件游标	全部逻辑输出内容所推导出的绝对文件位置，其中包括尾部预留空间。

对于当前主输出区域：

```
here()                = region_base() + logical offset
file_cursor_real()    = region file offset + real relative cursor
file_cursor_potential() = region file offset + logical offset
tail_reserve_size()   = logical bytes beyond the real relative cursor
```

`file_offset()` 是查询当前实际文件游标的常用名称，返回值与 `file_cursor_real()` 相同。

实际写出的字节会同时推进两个文件游标。尾部预留空间只推进逻辑游标和潜在文件游标：

```
// 在逻辑地址 0x4000、文件偏移 0x20 处开始载荷区域。
region.begin("payload", 0x4000, 0x20)

// 写出一个实际文件字节，再增加三个尾部预留字节。
emit.u8(0x11)
reserve(3)

// 逻辑位置包含预留空间，实际文件游标只越过已写入的字节。
assert(region_base() == 0x4000)
assert(here() == 0x4004)
assert(file_offset() == 0x21)
assert(file_cursor_real() == 0x21)
assert(file_cursor_potential() == 0x24)
assert(tail_reserve_size() == 3)
```

如果同一主输出区域在预留空间之后继续写出已经初始化的内容，中间的间隔就会成为输出文件的一部分。实际文件游标会先追上潜在位置，再越过新写出的字节。

开始明确的主输出区域

`region.begin(name, origin, file_offset)` 开始一个新的主输出区域：

- `name` 用于在诊断信息和布局信息中标识该区域；
- `origin` 是该区域的逻辑基地址；
- `file_offset` 是该区域在输出文件中的绝对起始位置。

新区域的相对逻辑游标和相对文件游标都从零开始。各参数彼此独立，因此一种格式可以把紧凑的文件范围映射到不同的逻辑地址。

`region.begin` 不会根据前一个区域推断连续位置。如果新区域必须续接已有布局，应当在调用之前查询相应的文件游标。

实际位置与潜在位置续接

`output.section(name, origin)` 和 `output.org(name, origin)` 都会开始新的主输出区，并为它设置新的逻辑原点。两者的区别在于从前一个输出区选择哪个文件位置：

过程	新文件位置	对尾部预留空间的影响
<code>output.section</code>	前一个实际文件游标	不把尚未写入文件的尾部预留空间计入后续文件布局。
<code>output.org</code>	前一个潜在文件游标	后续写出字节时，将尾部预留空间保留为文件间隔。

```
// 头部写出一个字节，并留下三个尾部预留字节。
region.begin("header", 0x4000, 0x20)
emit.u8(0x11)
reserve(3)

// 从实际文件游标续接，因此去除前面的尾部预留空间。
output.section("trimmed", 0x5000)
assert(file_offset() == 0x21)
emit.u8(0x22)
reserve(2)

// 从潜在文件游标续接，因此保留两个预留字节形成文件间隔。
output.org("preserved", 0x6000)
assert(file_offset() == 0x24)
emit.u8(0x33)
```

第一次切换会把 `trimmed` 放在字节 `0x11` 之后，三个尾部预留字节不会写入文件。第二次切换从潜在位置开始，因此 `0x22` 与 `0x33` 之间的两个预留字节会成为文件中间的间隔。

这两个过程都要求当前存在尚未结束的主输出区域。在虚拟输出区内调用它们属于无效操作。

结束并对齐文件区域

`region.file_align(boundary)` 会把当前主输出区域的实际文件大小向上取整到非零的 2 的幂边界，并结束该区域，禁止继续写出内容：

```
// 写出一个字节后，把主输出区域的实际文件大小对齐到八字节。
region.begin("record", 0x8000, 0)
emit.u8(0x7f)
region.file_align(8)

// 文件对齐只扩展实际文件范围，不会推进潜在文件游标。
assert(file_cursor_real() == 8)
assert(file_cursor_potential() == 1)
```

对齐产生的文件尾部属于输出文件，并使用零填充。因此，实际文件游标可以越过潜在文件游标。调用成功后，不能在该区域中继续写出、预留或对齐数据；必须开始另一个主输出区域或输出区才能继续。

虚拟输出区

`virtual.begin()` 会在当前逻辑地址开始一个嵌套的虚拟输出区。`virtual.begin(origin)` 则使用明确的逻辑原点。虚拟输出区可以定义标号、预留空间并保存临时字节，但不会向最终文件贡献任何字节。

```
// 主输出区域先写出一个字节，虚拟输出区从下一逻辑地址开始。
region.begin("main", 0x7000, 0)
emit.u8(0xaa)

// 虚拟输出区可以推进自身逻辑位置，但不会推进主输出区域。
virtual.begin()
assert(region_base() == 0x7001)
emit.u16(0x1234)
reserve(2)
assert(here() == 0x7005)
virtual.end()

// 返回后恢复主输出区域的逻辑原点和当前位置。
assert(region_base() == 0x7000)
assert(here() == 0x7001)
emit.u8(0xbb)
```

最终文件只包含 `aa bb`。`virtual.end()` 会恢复先前的输出区及其游标。虚拟输出区可以嵌套，但每个 `virtual.begin` 都必须有一个对应的 `virtual.end`。

文件游标查询描述的是主输出文件，不能用来推断虚拟输出区的存储范围。虚拟输出区处于活动状态时，应当使用逻辑地址和标号。

错误条件

区域 API 会拒绝以下情况：

- `region.file_align` 的边界为零或不是 2 的幂；
- `region.file_align` 已经结束当前主输出区域后，仍尝试继续写出内容；
- 虚拟输出区处于活动状态时调用 `output.section` 或 `output.org`；
- 在没有对应开始操作时调用 `virtual.end`；
- 源文件结束时仍有尚未关闭的虚拟输出区。

第 9 章：读取、写入与最终区域信息

语法摘要

形式	语法	结果
读取整数	<code>load.u8(address)</code>	读取一个字节。
读取整数	<code>load.u16(address)</code>	读取一个小端顺序的 16 位整数。
读取整数	<code>load.u32(address)</code>	读取一个小端顺序的 32 位整数。
读取整数	<code>load.u64(address)</code>	读取一个小端顺序的 64 位整数。
读取字节	<code>load.bytes(address, count)</code>	返回 <code>count</code> 个字节。
写入整数	<code>store.u8(address, value)</code>	写入一个字节。
写入整数	<code>store.u16(address, value)</code>	写入一个小端顺序的 16 位整数。
写入整数	<code>store.u32(address, value)</code>	写入一个小端顺序的 32 位整数。
写入整数	<code>store.u64(address, value)</code>	写入一个小端顺序的 64 位整数。
写入字节	<code>store.bytes(address, value)</code>	写入字符串或字节序列。
区域文件偏移	<code>region_file_offset(address)</code>	返回区域的最终文件偏移。
区域文件大小	<code>region_file_size(address)</code>	返回区域最终存在于文件中的字节数。
区域逻辑大小	<code>region_logical_size(address)</code>	返回区域的最终逻辑大小。

常规访问与收尾阶段访问

读取和写入操作在两个阶段访问不同的数据状态：

阶段	访问的数据
常规求值阶段	当前模块布局中已经存在的字节。
<code>defer</code> 收尾阶段	稳定的最终输出内容中的字节。

常规写入可以在目标字节已经生成后初始化或替换这些字节。收尾阶段写入适合处理依赖最终布局的字段，例如大小、偏移、校验和以及目录项。

```

// 创建逻辑地址从 0x3000、文件偏移从 0 开始的数据区域。
region.begin("data", 0x3000, 0)

// 先写出不同宽度整数和字节序列所需的存储位置。
byte_slot:
emit.u8(0)

word_slot:
emit.u16(0)

dword_slot:
emit.u32(0)

qword_slot:
emit.u64(0)

bytes_slot:
db(0, 0, 0)

// 常规求值阶段可以修改已经写出的字节。
store.u8(byte_slot, 0xaa)
store.u16(word_slot, 0x2233)

// 立即读取刚才写入的值，确认小端整数访问正确。
assert(load.u8(byte_slot) == 0xaa)
assert(load.u16(word_slot) == 0x2233)

defer {
    // 布局稳定后，回填其余整数和三字节标记。
    store.u32(dword_slot, 0x44556677)
    store.u64(qword_slot, 0x0102030405060708)
    store.bytes(bytes_slot, b"XYZ")

    // 收尾块能同时看到常规阶段和本收尾块完成的写入。
    assert(load.u8(byte_slot) == 0xaa)
    assert(load.u16(word_slot) == 0x2233)
    assert(load.u32(dword_slot) == 0x44556677)
    assert(load.u64(qword_slot) == 0x0102030405060708)
    assert(load.bytes(bytes_slot, 3) == b"XYZ")
}

```

整数形式只接受其指定宽度能够表示的值。读取和写入必须完全落在最终文件中实际存在的字节范围内。这些操作不会分配存储空间、扩大区域或改变布局。

标号与输出区身份

仅有逻辑地址不一定足以确定文件中的字节。两个输出区可以使用相互重叠的逻辑地址范围，同时占用不同的文件范围。

当读取、写入或最终区域信息表达式直接包含标号时，XIRASM 会保留该标号的输出区身份，并把它与逻辑地址结合起来。即使另一个输出区使用相同地址，也能据此选择正确的文件范围。

```
// 外层区域从逻辑地址 0x1000 和文件偏移 0 开始。
region.begin("outer", 0x1000, 0)

outer:
emit.u32(0x44332211)
// 尾部预留空间扩大逻辑大小，但不会立即形成文件字节。
reserve(4)

// 新输出区从实际文件游标继续，因此紧接 outer 已写出的四个字节。
output.section("inner", 0x1002)

inner:
emit.u16(0x6655)

defer {
    // 两个标号分别查询各自输出区的最终文件偏移和大小。
    assert(region_file_offset(outer) == 0)
    assert(region_file_size(outer) == 4)
    assert(region_logical_size(outer) == 8)

    assert(region_file_offset(inner) == 4)
    assert(region_file_size(inner) == 2)
    assert(region_logical_size(inner) == 2)

    // inner 与 outer + 2 地址相同，但输出区身份让它们访问不同字节。
    assert(load.u16(inner) == 0x6655)
    store.u16(inner, 0x8877)
    assert(load.u16(inner) == 0x8877)
    assert(load.u16(outer + 2) == 0x4433)
}
```

这里的 `outer + 2` 与 `inner` 具有相同逻辑地址。两个标号保留了不同的输出区身份，因此引用的是不同字节。

普通整数不带输出区身份。如果逻辑地址范围可能重叠，应当在访问表达式中保留标号，不要先把它转换成数值变量或函数参数。

最终区域信息

只有布局和最终输出都稳定后，才能使用以下三种最终区域信息；通常应当在 `defer` 中查询：

表达式	含义
<code>region_file_offset(label)</code>	区域在文件中的绝对起始位置，也就是最终文件偏移。
<code>region_file_size(label)</code>	区域最终存在于文件中的字节数，包括文件对齐产生的字节。
<code>region_logical_size(label)</code>	区域占用的逻辑地址范围，包括尾部预留空间。

未初始化的预留空间使这一区别十分重要。在前一个示例中，`outer` 的逻辑大小为八字节，文件大小只有四字节。`output.section` 从实际文件游标开始 `inner`，因此四字节尾部预留空间不会写入文件。

这些预留地址仍属于逻辑范围，但不是可以读取或写入的最终输出字节。收尾处理不能通过读取或写入，把已经裁剪的尾部预留空间变成实际输出。

地址与范围错误

以下情况会被这些 API 拒绝：

- 整数写入值超出所选宽度能够表示的范围；
- 读取或写入超出实际写入的文件字节范围；
- 访问已经从文件布局中裁剪的尾部预留空间；
- 在最终输出稳定之前查询最终区域信息；
- 最终区域信息查询的地址不属于任何最终逻辑区域；
- 一个标号表达式组合了来自不同输出区的标号。

使用数据写出、预留和区域操作定义布局。读取、写入和最终区域信息只应用于检查或回填布局已经建立的存储空间。

第 10 章：文本、转换与标号名称

语法摘要

函数	结果	说明
<code>lengthof(value)</code>	<code>u64</code>	返回字节长度，或整数的十进制位数。
<code>len(value)</code>	<code>u64</code>	返回字节数、列表项数或映射条目数。
<code>to_string(value)</code>	<code>string</code>	把整数、布尔值、字符串或字节序列转换为文本。
<code>trim(text)</code>	<code>string</code>	删除文本两端的 ASCII 空格、制表符、回车符和换行符。
<code>lower(text)</code>	<code>string</code>	把 ASCII 字母转换为小写。
<code>upper(text)</code>	<code>string</code>	把 ASCII 字母转换为大写。
<code>starts_with(text, prefix)</code>	<code>bool</code>	检查字符串是否以指定前缀开头。
<code>ends_with(text, suffix)</code>	<code>bool</code>	检查字符串是否以指定后缀结尾。
<code>contains(value, needle)</code>	<code>bool</code>	在字符串或字节序列中查找内容。
<code>replace(text, needle, replacement)</code>	<code>string</code>	替换字符串中所有互不重叠的匹配项。
<code>split(text, separator)</code>	<code>list</code>	按分隔符把字符串拆分为字符串列表。
<code>join(parts, separator)</code>	<code>string</code>	使用分隔符连接字符串列表。
<code>sym.join(parts...)</code>	<code>string</code>	转换并直接拼接多个值，不插入分隔符。
<code>sym.unique(prefix)</code>	<code>string</code>	返回本次汇编中不会重复的生成名称。

本章函数都是普通的编译期表达式。它们会创建新值，不会修改传入的值。

长度查询

`lengthof` 和 `len` 回答的问题不同：

输入	<code>lengthof</code>	<code>len</code>
<code>string</code>	字节长度	字节长度
<code>bytes</code>	字节长度	字节长度
无符号整数	十进制位数	不接受
<code>list</code>	不接受	列表项数
<code>map</code>	不接受	映射条目数

字符串长度按字节计算，而不是按书写字符的数量计算。对于整数，`lengthof(0)` 的结果是 `1`，因为它的十进制文本是 `"0"`。

```
// 查询整数与字符串的长度，并把 4096 的十进制位数写成文本。
assert(lengthof(4096) == 4)
assert(lengthof("XIRASM") == 6)
assert(len(split("code::data::", "::")) == 3)

db(to_string(lengthof(4096)))
```

输出是 ASCII 字节 `0x34`，表示文本 `"4"`。

转换为文本

`to_string(value)` 使用以下转换规则：

输入	文本形式
整数	无符号十进制
布尔值	<code>"true"</code> 或 <code>"false"</code>
字符串	内容不变的副本
字节序列	小写十六进制，每个字节使用两位数字

```
// 检查整数、布尔值、字符串和字节序列的文本转换结果。
assert(to_string(42) == "42")
assert(to_string(true) == "true")
assert(to_string("ready") == "ready")
assert(to_string(b"AZ") == "415a")
```

结构体、列表和映射不会自动转换。需要文本时，应分别转换其中的字段或元素。

文本变换与查找

`trim`、`lower` 和 `upper` 处理 ASCII 文本。`trim` 只把空格、水平制表符、回车符和换行符视为两端空白。大小写转换不会改变非 ASCII 字节。

`starts_with` 和 `ends_with` 接受字符串。`contains` 可以接受两个字符串，也可以接受两个字节序列；两个参数必须是同一种值。

```
// 清理并统一库名称的大小写，再检查其中的各个文本片段。
const name: string = lower(trim(" Kernel32.DLL "))

assert(name == "kernel32.dll")
assert(starts_with(name, "kernel"))
assert(ends_with(name, ".dll"))
assert(contains(name, "32"))

db(name)
```

替换、拆分与连接

`replace` 会替换 `needle` 的每个互不重叠的匹配项。`needle` 不能为空。

`split` 会在非空分隔符的每个匹配位置拆分字符串。空字段会被保留，包括相邻分隔符或末尾分隔符产生的字段。因此，空输入字符串会得到只包含一个空字符串的列表。

`join` 要求列表中的每一项都是字符串。空列表会得到空字符串。

```
// 拆分包含末尾分隔符的文本，再连接字段并执行全部匹配项替换。
const fields: list = split("red::green::", "::")

assert(len(fields) == 3)
assert(join(fields, "|") == "red|green|")
assert(replace("one fish, one fish", "one", "two") == "two fish, two fish")

db(join(fields, "|"))
```

构造标号名称

`sym.join(parts...)` 会按照与 `to_string` 相同的规则转换整数、布尔值、字符串和字节序列参数，然后直接拼接结果，不自动插入分隔符：

```
// 拼接固定文本、整数和布尔值，得到可直接定义的标号名称。
const slot: string = sym.join("_slot_", 12, "_", true)

assert(slot == "_slot_12_true")
label.define(slot)
emit.u8(0xaa)
```

需要分隔符时，应把所需标点明确写在参数中。

`sym.unique(prefix)` 每调用一次，都会返回本次汇编中不同的生成字符串。生成顺序是确定的，但后缀格式不是源码可以依赖的约定。应保存返回的字符串并始终使用该值，不要自行重新构造名称。

```
// 使用相同前缀生成两个不同名称，并分别定义标号和写出数据。
const first: string = sym.unique("_temporary")
const second: string = sym.unique("_temporary")

assert(first != second)

label.define(first)
emit.u8(0x11)

label.define(second)
emit.u8(0x22)
```

`sym.unique` 只保证名称不重复，不保证名称符合标号语法。把结果传给 `label.define` 时，应选择能使完整结果成为有效标号名称的前缀。

错误条件

以下输入会被拒绝：

- 向 `lengthof` 传入字符串、字节序列和无符号整数以外的值；
- 向 `len` 传入字符串、字节序列、列表和映射以外的值；
- 向 `to_string` 或 `sym.join` 传入聚合值；
- 向 `contains` 同时传入字符串和字节序列；
- 向 `replace` 传入空的 `needle`，或向 `split` 传入空分隔符；
- 向 `join` 传入包含非字符串项的列表；
- 把生成名称用于 `label.define` 时，该名称不符合标号语法。

第 11 章：字节序列

语法摘要

函数	结果	说明
<code>bytes.new()</code>	<code>bytes</code>	创建空字节序列。
<code>bytes.push(value, byte)</code>	<code>bytes</code>	在末尾追加一个字节。
<code>bytes.concat(left, right)</code>	<code>bytes</code>	连接两段字节序列。
<code>bytes.repeat(count, byte)</code>	<code>bytes</code>	创建由同一字节重复 <code>count</code> 次组成的序列。
<code>bytes.le(value, width)</code>	<code>bytes</code>	取整数的低 <code>width</code> 个字节，并按小端顺序编码。
<code>bytes.insert(value, index, addition)</code>	<code>bytes</code>	在从零开始的字节索引处插入字节。
<code>bytes.replace(value, index, count, replacement)</code>	<code>bytes</code>	替换一段字节范围。
<code>bytes.eq(left, right)</code>	<code>bool</code>	检查两段字节序列是否完全相同。
<code>bytes.hex(value)</code>	<code>string</code>	把字节序列转换为小写十六进制文本。
<code>bytes.from_hex(text)</code>	<code>bytes</code>	解析大写或小写的十六进制文本。

每项操作都会返回一个新值，不会修改作为输入的字节序列。因此，同一个中间值可以安全地用于多个后续构造。

创建并组合字节序列

`bytes.new()` 返回空字节序列。 `bytes.push` 在末尾追加一个取值范围为 0 到 255 的整数。
`bytes.concat` 连接两段字节序列， `bytes.repeat` 则创建由同一个字节填充的序列。

```
// 从空字节序列开始，追加操作不会改变 empty 本身。
const empty: bytes = bytes.new()
const tag: bytes = bytes.push(empty, 0x7f)
const padding: bytes = bytes.repeat(3, 0xaa)
const result: bytes = bytes.concat(tag, padding)

// 检查原值仍为空，并确认连接后的精确字节内容。
assert(len(empty) == 0)
assert(bytes.eq(result, bytes.from_hex("7faaaaaa")))

// 只把最终构造完成的字节序列写入输出内容。
emit.bytes(result)
```

调用 `bytes.push` 后，原来的 `empty` 值仍然是空字节序列。

小端整数编码

`bytes.le(value, width)` 会生成零到八个字节。第一个字节保存整数的最低八位。当 `width` 小于八时，更高位会被丢弃。

```
// 文件标记按原有字节顺序保留，版本号编码为两个小端字节。
const magic: bytes = bytes.from_hex("58495200")
const version: bytes = bytes.le(3, 2)
const header: bytes = bytes.concat(magic, version)

// 检查完整文件头、截断高位以及零字节宽度的结果。
assert(bytes.hex(header) == "584952000300")
assert(bytes.eq(bytes.le(0x1234, 1), bytes.from_hex("34")))
assert(bytes.eq(bytes.le(0x1234, 0), bytes.new()))

// 写出由文件标记和版本字段组成的文件头。
emit.bytes(header)
```

输入整数是无符号编译期值。`bytes.le` 不会进行有符号扩展，也不会检查被丢弃的高位是否为零。

插入与替换字节范围

字节索引从零开始。`bytes.insert(value, index, addition)` 接受从零到 `len(value)` 的任意索引，其中 `len(value)` 表示紧接最后一个字节之后的位置。

`bytes.replace(value, index, count, replacement)` 从字节索引 `index` 开始移除 `count` 个字节，再在同一位置插入 `replacement`：

- `count` 为零时，只插入新字节，不移除原有字节；

- `replacement` 为空时，删除选中的字节范围；
- 同时指定移除数量和非空替换内容时，执行普通替换。

```
// 在 ABC 的字节索引 1 处插入连字符，再替换索引 2 处的一个字节。
const base: bytes = b"ABC"
const marked: bytes = bytes.insert(base, 1, b"-")
const patched: bytes = bytes.replace(marked, 2, 1, b"Z")
const trailer: bytes = bytes.repeat(2, 0xff)
const result: bytes = bytes.concat(
    bytes.push(bytes.new(), 0x7f),
    bytes.concat(patched, trailer)
)

// 原始字节序列保持不变；零长度替换用于插入，空替换内容用于删除。
assert(bytes.eq(base, b"ABC"))
assert(bytes.eq(bytes.replace(b"AB", 1, 0, b"-"), b"A-B"))
assert(bytes.eq(bytes.replace(b"ABCD", 1, 2, bytes.new()), b"AD"))

// 写出前导字节、修改后的正文和两个尾部字节。
emit.bytes(result)
```

这个示例写出 `7f 41 2d 5a 43 ff ff`。

十六进制转换与相等比较

`bytes.from_hex` 接受字符数为偶数的十六进制字符串。字母形式的十六进制数字可以使用大写或小写，每两个字符生成一个字节。空字符串会生成空字节序列。

`bytes.hex` 执行反向转换，并且始终使用小写数字。需要明确比较两段字节序列的精确内容时，使用 `bytes.eq`。

```
// 把大小写混合的十六进制字符串解析为字节序列，再转换为规范的小写文本。
const value: bytes = bytes.from_hex("DEadBEEF")
const text: string = bytes.hex(value)

// 检查文本形式、往返转换和空字符串的解析结果。
assert(text == "deadbeef")
assert(bytes.eq(bytes.from_hex(text), value))
assert(bytes.eq(bytes.from_hex(""), bytes.new()))

// 写出解析得到的四个字节，而不是十六进制字符串本身。
emit.bytes(value)
```

错误条件

字节辅助接口会拒绝以下输入：

- 字节参数不在 0 到 255 的范围内；
- 需要字节序列的位置传入了非 `bytes` 值；
- `bytes.le` 的宽度大于八；
- 插入索引大于当前字节数量；
- 替换范围超过当前字节数量；
- 十六进制字符串包含奇数个字符；
- 十六进制字符串包含非十六进制字符；
- 函数参数数量不正确。

第 12 章：列表与映射

列表和映射都是编译期值集合。表达式接口在添加、替换或组合值时返回新集合，所有输入集合保持不变。当算法需要逐步构造集合时，也可以使用仅作用于直接 `let` 绑定的显式可变语句。

列表函数

函数	结果	说明
<code>list.new()</code>	<code>list</code>	创建空列表。
<code>list.of(values...)</code>	<code>list</code>	使用零个或多个值创建列表。
<code>list.push(value, item)</code>	<code>list</code>	在列表末尾追加一个元素。
<code>list.concat(left, right)</code>	<code>list</code>	连接两个列表。
<code>list.get(value, index)</code>	值	返回从零开始的索引所指向的元素。
<code>list.set(value, index, item)</code>	<code>list</code>	返回替换了一个元素的新列表。
<code>list.slice(value, start, count)</code>	<code>list</code>	返回一段连续范围组成的新列表。
<code>list.eq(left, right)</code>	<code>bool</code>	按顺序递归比较两个列表是否相等。

以下语句直接更新 `let` 绑定，不返回值：

语句	说明
<code>list.push_mut(target, item);</code>	把深拷贝后的元素追加到 <code>target</code> 。
<code>list.set_mut(target, index, item);</code>	使用深拷贝后的值替换已有元素。

`list.get` 和 `list.set` 要求索引小于列表长度。`list.slice` 的起始索引可以从零取到列表长度，但所选范围不能超出列表。允许在列表末尾取得长度为零的切片。

```
// 每次列表操作都返回新列表, base 和 extended 不会被后续操作修改。
const base: list = list.of(1, 2, 3)
const extended: list = list.push(base, 4)
const patched: list = list.set(extended, 1, 0xaa)
const middle: list = list.slice(patched, 1, 2)
const combined: list = list.concat(list.of(0x10, 0x11), middle)

// 索引从零开始; 列表相等比较同时检查元素顺序和内容。
assert(list.get(base, 1) == 2)
assert(list.eq(base, list.of(1, 2, 3)))
assert(list.eq(patched, list.of(1, 0xaa, 3, 4)))
assert(list.eq(middle, list.of(0xaa, 3)))

// 按 combined 中的顺序逐项写出字节。
for value in combined {
    emit.u8(value)
}
```

这段代码会写出 `10 11 aa 03`。列表相等比较与元素顺序有关，并会递归比较嵌套的列表、映射、结构体、字符串和字节序列。

映射函数

函数	结果	说明
<code>map.new()</code>	<code>map</code>	创建空映射。
<code>map.set(value, key, item)</code>	<code>map</code>	添加字符串键，或替换已有键对应的值，并返回新映射。
<code>map.has(value, key)</code>	<code>bool</code>	检查指定字符串键是否存在。
<code>map.get(value, key)</code>	值	返回必需键对应的值。
<code>map.get_or(value, key, fallback)</code>	值	键存在时返回对应值，否则返回给定的后备值。
<code>map.keys(value)</code>	<code>list</code>	按插入顺序返回键列表。
<code>map.values(value)</code>	<code>list</code>	按与键相同的插入顺序返回值列表。
<code>map.eq(left, right)</code>	<code>bool</code>	递归比较两个映射是否相等，不考虑键的顺序。

`map.set_mut(target, key, item)`；会在由直接 `let` 绑定的映射中添加或替换深拷贝后的值。键必须是字符串；替换已有键时，该键的插入位置保持不变。

映射的键必须是字符串。添加新键时，该项会追加到插入顺序末尾；替换已有键时，该键原有的位置保持不变。因此，`map.keys` 和 `map.values` 返回的两个列表——对应，相同索引表示同一个映射项。

```

// map.set 返回新映射; empty、first 和 configured 都保持原值。
const empty: map = map.new()
const first: map = map.set(empty, "arch", "x64")
const configured: map = map.set(first, "mode", 64)
const updated: map = map.set(configured, "arch", "rv64")
const complete: map = map.set(updated, "tags", list.of("asm", "dsl"))

// 检查键、后备值和插入顺序, 并确认替换不会改变旧映射。
assert(len(empty) == 0)
assert(map.has(complete, "arch"))
assert(!map.has(complete, "missing"))
assert(map.get(first, "arch") == "x64")
assert(map.get(updated, "arch") == "rv64")
assert(map.get_or(complete, "missing", "default") == "default")
assert(list.eq(map.keys(complete), list.of("arch", "mode", "tags")))
assert(list.eq(map.get(complete, "tags"), list.of("asm", "dsl")))

// 以不同顺序插入相同的键和值。
const reordered: map = map.set(
  map.set(
    map.set(map.new(), "tags", list.of("asm", "dsl")),
    "mode",
    64
  ),
  "arch",
  "rv64"
)

// map.eq 不要求插入顺序相同, 最后写出 mode 对应的字节。
assert(map.eq(complete, reordered))
emit.u8(map.get(complete, "mode"))

```

这段代码会写出 40。map.eq 会比较键和嵌套值, 但不要求两个映射具有相同的插入顺序。

可变集合绑定规则

list.push_mut、list.set_mut 和 map.set_mut 的目标必须是直接标识符, 并解析到词法作用域中最近的 let 绑定。目标不能是 const、临时表达式、调用结果、字段访问或类型不匹配的值。普通 lowering 阶段允许更新顶层 let; 值函数中只能更新本次调用内部的局部绑定。这些接口是语句, 不能作为表达式使用, 也不能出现在 defer 或 late_layout 块中。

插入的值会在提交修改之前完成深拷贝, 因此目标之前的副本以及插入到其他集合中的值都保持独立。分配失败时, 目标集合保持原样。

错误条件

集合辅助接口会拒绝以下情况：

- 向 `list.*` 操作传入非列表参数；
- 向 `map.*` 操作传入非映射参数；
- 函数参数数量错误；
- 列表索引超出可用元素范围；
- 列表切片超出列表范围；
- 映射键不是字符串；
- 向 `map.get` 传入不存在的键。

如果键不存在属于预期情况，应使用 `map.has` 或 `map.get_or`。

第 13 章：文件与结构化数据

语法摘要

函数	结果	说明
<code>fs.exists(path)</code>	<code>bool</code>	检查指定路径能否定位到文件。
<code>fs.read_text(path)</code>	<code>string</code>	把整个文件读取为字符串。
<code>fs.read_bytes(path)</code>	<code>bytes</code>	把整个文件读取为字节序列。
<code>fs.read_bytes(path, offset, count)</code>	<code>bytes</code>	读取一段长度明确的字节范围。
<code>emit.file(path)</code>	语句	写出完整的源文件相对文件。
<code>emit.file(path, offset, count)</code>	语句	写出精确的文件字节范围。
<code>json.parse(value)</code>	值	解析字符串或字节序列中的 JSON。
<code>json.file(path)</code>	值	读取并解析 JSON 文件。
<code>toml.parse(value)</code>	<code>map</code>	解析字符串或字节序列中的 TOML 文档。
<code>toml.file(path)</code>	<code>map</code>	读取并解析 TOML 文件。

文件路径遵循与 `include` 和 `import` 相同的受控路径规则。相对路径以包含该调用的源文件为基准。因此，把调用移动到嵌套模块中时，它所使用的相对数据路径基准也会随之改变。

各阶段的可用性

操作	常规求值阶段	<code>late_layout</code>	<code>defer</code>
<code>fs.exists</code> 、 <code>fs.read_text</code> 、 <code>fs.read_bytes</code> 、 <code>emit.file</code>	可用	不可用	不可用
<code>json.file</code> 、 <code>toml.file</code>	可用	不可用	不可用
<code>json.parse</code> 、 <code>toml.parse</code>	可用	可在值表达式中使用	可在值表达式中使用

文件操作需要当前阶段能够按照源文件位置解析路径。`late_layout` 和 `defer` 阶段不能重新访问源文件。需要在后期使用结构化数据时，应当在常规求值阶段完成文件读取与解析，并保留得到的值。

`parse` 函数不访问文件系统，只处理传入的字符串或字节序列。

检查与读取文件

路径无法定位到文件时，`fs.exists` 返回 `false`。读取函数遇到缺失文件时则会报告错误。

`fs.read_text` 保留文件的完整内容，不会附加结尾零字节。`fs.read_bytes` 返回内容相同的字节序列。

```
// 先检查必需文件与可选文件是否存在。
assert(fs.exists("payload.bin"))
assert(fs.exists("banner.txt"))
assert(!fs.exists("optional.bin"))

// 从二进制文件读取四字节文件头，并把文本文件读取为字符串。
const header: bytes = fs.read_bytes("payload.bin", 0, 4)
const banner: string = fs.read_text("banner.txt")

// 按读取顺序写出二进制文件头和文本内容。
emit.bytes(header)
emit.bytes(banner)
```

如果 `payload.bin` 包含 `XIR!`，而 `banner.txt` 包含 `ready` 和紧随其后的换行符，示例会写出：

```
58 49 52 21 72 65 61 64 79 0a
```

范围读取形式使用从零开始的偏移和字节数量。整个范围必须位于文件以内。在文件末尾读取长度为零的范围是有效操作。

`emit.file` 使用与 `fs.read_bytes` 完全相同的解析器和范围规则，但会直接写入输出，而不是返回 `bytes` 值。

JSON 值

JSON 值按下表转换为编译期值：

JSON 值	编译期值
<code>null</code>	<code>void</code>
布尔值	<code>bool</code>
字符串	<code>string</code>
非负整数	整数
数组	<code>list</code>
对象	<code>map</code>

`json.file` 在一次调用中完成读取和解析。`json.parse` 接受已经包含 JSON 内容的字符串或字节序列。

```
// 分别直接读取配置文件，以及读取字节后再次解析同一份配置。
const config: map = json.file("config.json")
const parsed_again: map = json.parse(fs.read_bytes("config.json"))
const values: list = map.get(config, "values")

// 两种读取方式应得到相同映射，值为 null 的键也仍然存在。
assert(map.eq(config, parsed_again))
assert(map.get(config, "enabled"))
assert(map.has(config, "nothing"))

// 从映射中读取文本、整数和列表，并按配置内容写出字节。
emit.bytes(map.get(config, "name"))
emit.u8(map.get(config, "bits"))

for value in values {
    emit.u8(value)
}
```

对应输入为：

```
{
  "name": "XR",
  "bits": 64,
  "enabled": true,
  "values": [1, 2],
  "nothing": null
}
```

示例会写出 `58 52 40 01 02`。

JSON 浮点数、负整数、重复的对象键、格式错误的输入，以及超出支持范围的整数都会被拒绝。

TOML 值

TOML 文档转换为映射。嵌套表转换为嵌套映射，数组转换为列表，字符串、布尔值和非负整数保持对应的编译期值类型。

```
// 分别直接读取配置文件，以及读取文本后再次解析同一份配置。
const config: map = toml.file("config.toml")
const parsed_again: map = toml.parse(fs.read_text("config.toml"))
const target: map = map.get(config, "target")
const values: list = map.get(parsed_again, "values")

// 两种读取方式应得到相同映射，并保留布尔配置。
assert(map.eq(config, parsed_again))
assert(map.get(config, "enabled"))

// 从顶层映射和嵌套映射中读取输出字段。
emit.bytes(map.get(config, "name"))
emit.u8(map.get(target, "bits"))

for value in values {
    emit.u8(value)
}
```

对应输入为：

```
# 顶层配置值。
name = "XR"
enabled = true
values = [3, 4]

# 目标平台配置表。
[target]
bits = 32
```

示例会写出 58 52 20 03 04。

浮点数、时间戳、负整数、格式错误的文档和重复键都会在转换过程中被拒绝。

错误条件

以下情况会被这些辅助接口拒绝：

- 文件路径不是字符串；
- 传给读取函数或 `*.file` 函数的文件不存在；
- 字节范围超出文件边界；

- 传给 `json.parse` 或 `toml.parse` 的值既不是字符串，也不是字节序列；
- JSON 或 TOML 格式错误；
- 对象或表中存在重复键；
- 结构化数据包含无法表示为编译期值的内容；
- 函数实参数量不正确。

只有文件缺失属于预期情况时，才需要在读取前调用 `fs.exists`。

第 14 章：词法单元与模式匹配

语法摘要

功能	调用形式	结果
对源代码形式的文本进行词法分析	<code>tokens.of(source)</code>	由词法单元字符串组成的 <code>list</code>
呈现词法单元字符串	<code>tokens.join(tokens)</code>	规范形式的 <code>string</code>
匹配词法单元结构	<code>match.tokens(pattern, input)</code>	结果 <code>map</code>

`tokens.of` 接受字符串或已有的字符串列表。传入字符串时会进行词法分析；传入列表时会验证每一项并复制该列表。

`tokens.join` 接受所有元素都是字符串的列表。

`match.tokens` 的模式和输入都可以是字符串或字符串列表。模式为列表时，每一项必须是一个完整的模式片段。输入为列表时，其中的词法单元会直接参与匹配，不会再次进行词法分析。

词法分析

`tokens.of` 用于处理具有源代码结构的文本。它会分离名称、字面量、标点、括号和运算符，并丢弃不影响结构的空白。

```
// 把一条具有地址表达式的文本拆成词法单元列表。
const input: list = tokens.of("load rax, [rbx + 4]")

// 空白不会成为词法单元，标点和括号会被单独保留。
assert(len(input) == 8);
assert(list.get(input, 0) == "load");
assert(list.get(input, 2) == ",");
assert(list.get(input, 3) == "[");
assert(tokens.join(input) == "load rax, [rbx+4]");

// 写出词法单元数量和规范形式文本。
emit.u8(len(input));
emit.bytes(tokens.join(input));
```

示例会写出：

```
08 6c 6f 61 64 20 72 61 78 2c 20 5b 72 62 78 2b 34 5d
```

带引号的文本会保留为一个词法单元，其中包括引号字符。没有结束引号的词法单元无效。

以下双字符运算符会分别形成一个词法单元：

```
== != <= >= && || << >> -> => ::
```

以下单字符也会分别形成一个词法单元：

```
, ( ) [ ] { } < > : = + - * / % & | ^ ~ . ! ? ;
```

其他相邻的非空白字符会留在同一个词法单元中，直到遇到引号或能够识别的运算符。

规范形式呈现

`tokens.join` 会生成规范形式的源代码文本。它会在普通相邻词法单元之间插入空格，并删除括号、标点和紧密运算符周围不需要的空格。

规范形式呈现不会逐字节还原原始字符串：

```
// 原始文本中的多余空格不会保留在规范形式结果中。
const input: list = tokens.of("left  && middle ||  right")
assert(tokens.join(input) == "left&&middle||right");
```

如果空白的精确形式本身有意义，应保留原始字符串。

模式片段

词法单元模式由若干使用空白分隔的片段组成。每个片段必须是精确字面量或命名捕获。

片段	含义
<code>=token</code>	精确匹配一个词法单元
<code>name:token</code>	捕获任意一个词法单元，并返回字符串
<code>name:name</code>	捕获一个名称形式的词法单元，并返回字符串
<code>name:int</code>	捕获一个整数词法单元，并返回整数
<code>name:quoted</code>	捕获一个带引号的词法单元，并返回去除引号后的字符串
<code>name:tokens</code>	捕获一段保持括号平衡的词法单元，并返回列表

开头的 `=` 是字面量标记。例如：

```
=load    匹配 load 词法单元
=,       匹配逗号词法单元
===      匹配 == 词法单元
```

捕获名称使用标识符语法，并且在同一个模式中不能重复。

匹配结果

`match.tokens` 返回包含两个条目的映射：

键	值
"ok"	表示整个输入是否匹配的布尔值
"captures"	从捕获名称到捕获值的映射

只有完整模式和完整输入都被使用后，匹配才算成功。

```
// 匹配 load 形式，并分别捕获目标名称和完整地址表达式。
const result: map = match.tokens(
    "=load destination:name =, address:tokens",
    "load rax, [rbx+(rcx*4)]"
)

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

// 匹配成功后，从 captures 映射中读取命名捕获。
const captures: map = map.get(result, "captures")
const destination: string = map.get(captures, "destination")
const address: list = map.get(captures, "address")

assert(destination == "rax");
assert(tokens.join(address) == "[rbx+(rcx*4)]");

// 写出捕获到的目标名称和规范形式地址文本。
emit.bytes(destination);
emit.bytes(tokens.join(address));
```

示例会写出：

```
72 61 78 5b 72 62 78 2b 28 72 63 78 2a 34 29 5d
```

读取命名捕获之前必须先检查 `"ok"`。匹配失败时，结果中的 `"ok"` 为 `false`，捕获映射为空。

捕获值

`token` 和 `name` 捕获都返回字符串。`token` 接受任意一个词法单元；`name` 要求标识符以 ASCII 字母或下划线开头，后续字符只能是 ASCII 字母、数字或下划线。

`int` 会使用通常的数值前缀，把一个词法单元解析为无符号整数：

```
// 捕获目标名称和带十六进制前缀的整数。
const result: map = match.tokens(
  "set target:name =, value:int",
  "set count, 0x2a"
)
const captures: map = map.get(result, "captures")

// 整数捕获会返回数值，而不是原始词法单元字符串。
assert(map.get(result, "ok"));
assert(map.get(captures, "target") == "count");
assert(map.get(captures, "value") == 42);
```

`quoted` 接受单引号或双引号，去除两端引号，并处理 `\n`、`\r`、`\t`、转义引号和转义反斜杠。其他转义形式会保留反斜杠后的字符。

`tokens` 返回由词法单元字符串组成的列表，可以捕获零个或多个词法单元。捕获范围内的圆括号、方括号和花括号必须保持平衡。

从最短范围开始匹配与回溯

`tokens` 捕获最初只使用最短的平衡范围。如果后续模式片段无法匹配，它会扩大捕获范围并重新尝试：

```
// prefix 先尝试空范围；整数捕获失败后，它会扩大到包含 name。
const result: map = match.tokens(
  "prefix:tokens value:int",
  "name 42"
)
const captures: map = map.get(result, "captures")

// 回溯后，prefix 捕获名称词法单元，value 捕获整数 42。
assert(map.get(result, "ok"));
assert(tokens.join(map.get(captures, "prefix")) == "name");
assert(map.get(captures, "value") == 42);
```

比较运算符 `<` 和 `>` 只是普通词法单元，不是分组边界。只有 `()`、`[]` 和 `{}` 参与平衡检查。

模式不提供表示多个选择的运算符。如果输入可能具有多种有效形式，应在普通 `if / else` 流程控制中依次尝试完整模式。

普通匹配失败与无效模式

以下情况属于普通匹配失败，返回 `"ok" == false`：

- 精确字面量不同；
- 带类型的捕获遇到不符合要求的词法单元；
- `tokens` 捕获无法形成平衡范围；
- 模式先于输入结束；
- 输入先于模式结束。

以下情况会把调用作为无效表达式拒绝：

- 模式片段既不是字面量，也不是捕获；
- 字面量标记后没有词法单元；
- 捕获名称无效或重复；
- 捕获种类未知；
- 输入中的带引号词法单元没有结束引号；
- 参数不是字符串或字符串列表；
- `tokens.join` 收到包含非字符串值的列表；
- 实参数量不正确。

限制

词法单元匹配使用固定限制，使编译期解析的工作量保持可控：

限制项	最大值
模式片段	64
输入词法单元	256
匹配尝试次数	4096
嵌套括号深度	64

超过限制时，匹配调用会被拒绝，而不是作为普通匹配失败返回。