

API 参考

XIRASM 中文 API 参考

语法、内置 API、约束与诊断速查

目录

1. XIRASM 语言 API 参考
2. 源码、绑定与作用域
3. 函数与流程控制
4. 结构体、联合体与复合值
5. 收尾处理语法形式
6. 模块与诊断信息
7. 目标平台、处理器指令与符号
8. 数据写出、预留空间与对齐
9. 主输出区域、输出区与文件游标
10. 读取、写入与最终区域信息
11. 文本、转换与标号名称
12. 字节序列
13. 列表与映射
14. 文件与结构化数据
15. 词法单元与模式匹配

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

只有完整模式和完整输入都被使用后，匹配才算成功。

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

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

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

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

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

示例会写出：

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

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

捕获值

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

`int` 会使用通常的数值前缀，把一个词法单元解析为无符号整数：

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

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

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

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

从最短范围开始匹配与回溯

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

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

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

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

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

普通匹配失败与无效模式

以下情况属于普通匹配失败，返回 `"ok" == false`：

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

以下情况会把调用作为无效表达式拒绝：

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

限制

词法单元匹配使用固定限制，使编译期解析的工作量保持可控：

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

超过限制时，匹配调用会被拒绝，而不是作为普通匹配失败返回。