XLang-Rust 文档

目录

概述

XLang-Rust 是一个使用 Rust 语言编写的跨平台的实验性动态强类型编程语言，致力于通过精简的语法实现复杂的功能。

编译器使用

XLang-Rust 的编译器通过命令行接口 `xlang-rust`（或其他编译后的可执行文件名）提供服务，支持以下子命令：

- `repl`：启动交互式命令行模式（Read-Eval-Print Loop），类似 `Mathematica`，用于即时执行代码片段。
- `run <input>`：直接运行指定的代码文件。`<input>` 可以是 XLang 源代码文件（如 `.x`）、中间代码文件（`.xir`）或字节码文件（`.xbc`）。
- `compile <input> [-o <output>] [-b|--bytecode]`：将源代码文件（`<input>`）编译为中间代码（`.xir`）或字节码（`.xbc`）。
  - 默认编译为 `.xir` 文件。
  - 使用 `-b` 或 `--bytecode` 选项可直接编译为字节码（`.xbc`）文件。
  - 可通过 `-o <output>` 或 `--output <output>` 选项指定输出文件的路径。如果未指定，输出文件名将基于输入文件名，并使用相应的扩展名（`.xir` 或 `.xbc`）。
- `display-ir <input>`：读取并以可读格式打印指定的中间代码文件（`.xir`）的内容，包括指令、函数入口点和调试信息。
- `translate <input> [-o <output>]`：将指定的中间代码文件（`<input>.xir`）翻译成字节码文件（`.xbc`）。
  - 可通过 `-o <output>` 或 `--output <output>` 选项指定输出字节码文件的路径。如果未指定，输出文件名将基于输入文件名，并使用 `.xbc` 扩展名。
- `lsp [-p <port>]`：启动 Language Server Protocol (LSP) 服务器，用于与集成开发环境（IDE）如 VS Code 进行交互，提供代码补全、诊断等功能。
  - 可通过 `-p <port>` 或 `--port <port>` 选项指定服务器监听的端口号。

表达式/语句

XLang-Rust 的语句结构与 Rust 类似，使用分号 `;` 作为表达式序列的分隔符。当存在多个表达式时，它们将按顺序求值，序列中最后一个表达式的求值结果将作为整个语句的值。

若表达式序列为空（例如 `` 或 `;` 结尾），则其求值结果为 `null`。

```xlang
print(1; 2; 3); // 输出 3，因为 3 是最后一个表达式的值
print(1;) // 输出 null，因为最后一个表达式为空
```

原子表达式

为了简化内部抽象语法树（AST）的构建，XLang-Rust 引入了原子表达式的概念。原子表达式是指被任何类型的括号（`()`，`[]`，`{}`）包围的表达式，或是一个单独的词法单元（token）。原子表达式的求值结果即为其内部表达式的求值结果。

以下均为原子表达式的示例：

```xlang
(1; 2; 3) // 被圆括号包围的表达式序列

```

```xlang
[1, 2, 3] // 被方括号包围的列表字面量（也是原子表达式）

{1, 2, 3} // 被花括号包围的代码块（也是原子表达式）

variable // 单个标识符（变量名）

1 // 单个数字字面量

"xxx" // 单个字符串字面量

... // 其他符合定义的单个词法单元

```

括号/方括号/花括号

括号/方括号

在改变运算优先级方面，XLang-Rust 不区分圆括号 `()` 和方括号 `[]`。两者均可用于控制表达式的求值顺序。它们的区别主要体现在函数调用和索引操作上。

xlang

```xlang
(1 + 2) * 3 // 求值结果为 9
[1 + 2] * 3 // 求值结果为 9
(1 + 2) * [3] // 求值结果为 9
```

当一个非原子表达式后紧跟圆括号时，圆括号内的表达式序列将作为参数传递给前面表达式求值得到的 lambda 对象进行调用。若紧跟方括号，则方括号内的表达式序列将作为索引作用于前面表达式求值得到的元组（或其他可索引）对象。

花括号

花括号 `{}` 用于创建新的作用域（帧作用域）。花括号内的表达式序列将在该新作用域中执行。在此作用域内定义的变量不会影响外部作用域。花括号表达式的求值结果是其内部最后一个表达式的求值结果。

xlang

```xlang
{
    a := 1; // 在新作用域内定义 a
    b := 2; // 在新作用域内定义 b
    a + b   // 此表达式的值将作为花括号表达式的结果
} // 求值结果为 3
```

变量

变量定义/赋值

XLang-Rust 是一种动态强类型语言。变量的类型在运行时确定，但一旦确定，类型约束将被强制执行。变量可以在运行时被重新赋值为相同类型的值。

使用 `:=` 运算符在当前作用域内定义新变量并赋值。使用 `=` 运算符对已存在的变量进行赋值。

XLang-Rust 允许在同一作用域内使用 `:=` 重新定义同名变量（遮蔽）。

xlang

```xlang
a := 1; // 定义变量 a 并初始化为整数 1
```

XLang-Rust 的赋值操作是强类型的。在执行赋值（`=`）时，虚拟机会检查右侧表达式值的类型是否与变量当前持有值的类型兼容。如果不兼容，将引发（`raise`）一个类型错误异常。若此异常未被捕获，程序将终止并报告错误信息。

xlang

```
a := 1;
a = "1"; // 引发类型错误异常，因为字符串 "1" 与整数 1 类型不兼容
a = 2;   // 合法赋值，类型匹配
a = 1.0; // 特殊情况：整数和浮点数之间存在隐式转换规则。
```

默认情况下，XLangVM 将变量名与存储在内存中的对象关联起来。变量定义实际上是在当前作用域（通常是一个哈希映射）中创建了一个指向该对象的引用。虚拟机通过变量名访问该引用。

xlang

```
a := 1; // 定义变量 a，指向一个值为 1 的整数对象
b := a; // 定义变量 b，使其引用 a 所引用的同一个对象
c := a; // 定义变量 c，同样引用 a 所引用的对象
a = 2; // 对 a 进行赋值，实际上是修改了 a 所引用的那个对象的值

assert(b == 2); // b 仍然引用同一个被修改的对象，其值已变为 2
assert(c == 2); // c 也仍然引用同一个被修改的对象，其值已变为 2
```

复制

由于变量存储的是对象的引用，XLang-Rust 提供了 `copy` 和 `deepcopy` 内建函数来创建对象的副本。 `copy` 函数执行浅复制：它创建一个新对象，其内容与原对象相同。如果对象包含对其他对象的引用，则只复制引用本身，而不复制引用的目标对象。 `deepcopy` 函数执行深复制：它递归地复制对象及其包含的所有嵌套对象，确保副本与原始对象完全独立。

xlang

```
a := [1, 2, 3]; // a 引用一个列表对象
b := copy a;   // b 引用 a 列表对象的浅副本

a := {
    'A' : 1,
    'B' : {
        'C' : 2,
        'D' : 3
    }
}; // a 引用一个嵌套字典对象
b := deepcopy a; // b 引用 a 字典对象的深副本
```

万物皆对象

在 XLang-Rust 中，所有的数据，包括基础类型（如整数、浮点数、字符串），都被视为对象（具体实现为 `GCObject`）。因此，变量赋值传递的是对象的引用，而非值本身。这解释了以下行为：

xlang

```
a := 1; // a 引用一个值为 1 的整数对象
b := a; // b 引用与 a 相同的整数对象
a = 2; // 修改 a 引用的对象的值（整数对象可变），或者使 a 引用一个新的值为 2 的对象

assert(b == 2); // b 引用的是同一个被修改的对象
```

注：若需确保获得一个独立的值副本（即使是基础类型），应使用 `copy` 函数。

变量作用域

XLang-Rust 定义了三种主要的作用域类型：函数作用域、帧作用域和边界作用域。

- 函数作用域：在调用 lambda (函数) 时创建，用于存储函数参数和函数内部定义的局部变量。
- 帧作用域：在执行花括号 `{}` 代码块时创建，用于隔离块内定义的变量。
- 边界作用域：通过 `boundary` 关键字创建，功能上类似帧作用域，但具有捕获其内部 `raise` 语句的能力。

与某些语言不同，XLang-Rust 的控制流语句（如 `if`，`while`）本身不创建新的作用域。作用域的创建仅由 lambda 调用、 `{}` 块和 `boundary` 块触发。这是其基于表达式的设计哲学的一部分。

`boundary` 语句创建一个边界作用域。除了提供变量隔离外，它还能捕获在其内部（包括嵌套的作用域和函数调用中）发生的 `raise` 操作。当 `raise value` 执行时，控制流将立即跳转到包含该 `raise` 的最内层 `boundary` 语句的末尾，并且整个 `boundary` 表达式的求值结果为 `value`。XLangVM 内部错误也可能自动触发 `raise`。

xlang

```
A := {       // 创建帧作用域
    B := 2;
    B        // 帧作用域的最后一个表达式
};
assert(A == 2); // A 被赋值为帧作用域的结果 2

A := boundary { // 创建边界作用域
    B := 2;
    raise 1;     // 控制流跳转到 boundary 结束，值为 1
    B            // 这行代码不会执行
};

assert(A == 1); // A 被赋值为 raise 的值 1
```

注意: `boundary` 和 `raise` 并非设计为传统的异常处理机制（尽管可以模拟类似行为）。它们的主要目的是提供一种比 `return` 更强大的非局部控制转移机制，允许从深层嵌套的结构中提前返回值。

xlang

```
A := boundary if true { // boundary 包裹 if 语句
    // ... 一些代码 ...
    { // 嵌套的帧作用域
        // ... 更多代码 ...
        raise 1; // 从嵌套作用域中提前返回到 boundary
        // ... 不会执行的代码 ...
    }
    // ... 不会执行的代码 ...
}
assert(A == 1); // A 的值为 1
```

这种机制同样适用于跨越函数调用的提前返回。

变量类型

XLang-Rust 允许显式构建如下类型：

- `var := 1;` // 整数类型
- `var := 1.0;` // 浮点数类型
- `var := "1";` // 字符串类型
- `var := (1, 2, 3);`, `var := [1, 2, 3];`, `var := {1, 2, 3};` // 元组类型
- `var := key : value;` // 键值对类型
- `var := key ⇒ value;` // 命名参数类型
- `var := (key : value, key2 : value2);` // 复合键值对类型
- `var := tuple → body;` // lambda 函数类型
- `var := tuple → &obj body;` // lambda 函数类型，带捕获变量
- `var := tuple → dyn body;` // lambda 函数类型，动态生成指令集
- `var := tuple → &obj dyn body;` // lambda 函数类型，动态生成指令集，带捕获变量
- `var := int..int;` // 区间类型，表示 [int, int)
- `var := null;` // 空类型，表示无值
- `var := true;`, `var := false;` // 布尔类型，表示真或假
- `var := wrap value;` // 包装类型，表示一个值的包装对象
- `var := $"base64";` // base64 编码的字节数组
- `var := import "path";` // VM 字节码对象，通过 import 语句导入
- `var := load_clambda("path");` // C 库对象，通过 load_clambda 语句导入
- `var := container | lambda;` // 惰性筛选器

可以使用 `typeof value` 来获取变量的类型。

xlang

```
a := 1; // 整数类型
assert(typeof a == "int"); // 类型检查
```

注意: 变量的类型在运行时确定，XLang-Rust 是动态类型语言。

变量类型转换

XLang-Rust 支持隐式和显式的类型转换。隐式转换通常在赋值或运算时自动进行，而显式转换则需要使用内建函数进行。

运算符

XLang-Rust 支持多种运算符，包括算术运算符、比较运算符、逻辑运算符和位运算符。具体可参考示例代码。

内建类型详解

XLang-Rust 提供了多种内建数据类型。所有类型都是对象，变量存储的是对象的引用。（下述所有的 `≠` 运算均为 `==` 的反向操作因此不做介绍）

可以使用 `typeof value` 来获取变量的类型字符串。

整数 (int)

- 描述: 存储 64 位有符号整数 (`i64`)。
- 创建: 通过整数常量创建，例如 `10`, `-5`, `0` 。
- 操作:

- 算术运算: 支持 `+`, `-`, `*`, `%` (模), `**` (幂)。除法 `/` 结果总是 `float` 类型。可与 `int` 或 `float` 运算（后者结果为 `float`）。
- 位运算: 支持 `and` (按位与), `or` (按位或), `xor` (按位异或), `not` (按位非), `<<` (左移), `>>` (右移)。
- 比较运算: 支持 `==`, `<`, `>`。可与 `int` 或 `float` 比较。
- 类型转换: 可转换为 `string`, `float`, `bool` (`0` 为 `false`, 其他为 `true`)。

浮点数 (float)

- 描述: 存储 64 位浮点数 (`f64`)。
- 创建: 通过浮点数常量创建，例如 `3.14`, `-0.5` （其实是用了一个 neg 运算）, `1e10`。
- 操作:
  - 算术运算: 支持 `+`, `-`, `*`, `/`, `%`, `**`。可与 `int` 或 `float` 运算。
  - 比较运算: 支持 `==`, `<`, `>`。可与 `int` 或 `float` 比较。
  - 类型转换: 可转换为 `string`, `int` (截断小数部分), `bool` (`0.0` 为 `false`, 其他为 `true`)。

布尔 (bool)

- 描述: 存储逻辑值 `true` 或 `false`。
- 创建: 通过常量 `true` 和 `false` 创建。
- 操作:
  - 逻辑运算: 支持 `and`, `or`, `xor`, `not`。
  - 比较运算: 支持 `==`。
  - 类型转换: 可转换为 `string` (`true` 或 `false`), `int` (`true` 为 `1`, `false` 为 `0`), `float` (`true` 为 `1.0`, `false` 为 `0.0`)。

空值 (null)

- 描述: 表示“无值”或“未定义”的状态。
- 创建: 通过常量 `null` 创建。空表达式序列（如 `( )` 或 `;` 结尾）的求值结果也是 `null`。
- 操作: 仅支持 `==` 比较（`null == null` 为 `true`）。

键值对 (keyval)

- 描述: 存储一个键（key）和一个值（value）的配对。通常作为元组的元素存在。
- 创建: 使用 `key : value` 语法创建。
- 操作:
  - 访问: 可通过 `keyof keyval_obj` 获取键，`valueof keyval_obj` 获取值。
  - 比较: `==` 比较键和值是否都相等。

命名参数 (named)

- 描述: 存储一个键（通常是标识符/字符串）和一个值的配对，专门用于函数参数传递或元组内的命名项。
- 创建: 使用 `key ⇒ value` 语法创建。语法糖 `key?` 等价于 `key ⇒ null`， `key!` 等价于 `key ⇒ key`。
- 操作:
  - 访问: 可通过 `keyof named_obj` 获取键，`valueof named_obj` 获取值。
  - 比较: `==` 比较键和值是否都相等。

区间 (range)

- 描述: 表示一个半开半闭的整数区间 `[start, end)`。
- 创建: 使用 `start_int..end_int` 语法创建。
- 操作:
  - 算术: `range + int`, `range - int` (移动区间两端), `range + range`, `range - range` (对应端点相加减)。
  - 包含判断: `int in range`, `sub_range in range`。
  - 长度: `len(range)` 返回 `end - start`。
  - 迭代: 可迭代区间内的所有整数（从 `start` 到 `end - 1`）。
  - 比较: `==` 比较区间的起始和结束点是否都相等。

字节序列 (bytes)

- 描述: 存储一个 `u8` 字节的有序序列。
- 创建: 使用 `$"base64_encoded_string"` 语法创建。
- 操作:
  - 连接: `bytes1 + bytes2`。
  - 索引访问: `bytes[index]` 返回指定索引处的字节值 (作为 `int`)； `bytes[range]` 返回指定范围的子字节序列 (作为 `bytes`)。
  - 赋值修改: 支持通过索引或范围修改字节内容。
    - `bytes = index : int_0_255`： 修改单个字节。
    - `bytes = index : string` 或 `bytes = index : bytes`： 从指定索引开始，用字符串或字节序列的内容覆盖后续字节。
    - `bytes = range : int_0_255`： 将范围内的所有字节设置为该整数值。
    - `bytes = range : string` 或 `bytes = range : bytes`： 用字符串或字节序列的内容替换范围内的字节（要求长度匹配)。

    注意: 字节序列采用这样的赋值方式是因为 XLangVM 自身不支持索引到字节序列对象的某个字节的地址，因此只能通过键值对的方式来实现对字节序列的赋值。但是不得认定元组对象的赋值方式是这样的。元组对象可以直接通过索引来访问和赋值。
  - 长度: `len(bytes)` 返回字节数。
  - 迭代: 可迭代访问序列中的每个字节 (作为 `int`)。
  - 比较: `==` 比较字节内容是否完全相同。
  - 类型转换: `string(bytes)` 尝试将字节序列按 UTF-8 解码为字符串。

包装器 (wrapper)

- 描述: 包装另一个任意类型的对象。
- 创建: 使用 `wrap value` 语法创建。
- 操作:
  - 解包: `valueof wrapper_obj` 返回内部被包装的对象。
  - 复制/赋值: `copy`, `deepcopy`, `assign` 操作通常会委托给内部对象。

惰性筛选器 (set)

- 描述: 定义一个惰性筛选操作，包含一个源容器（如元组、区间、字节序列等可迭代对象）和一个筛选 Lambda 函数。注意：迭代此对象时，仅迭代源容器，并不会自动应用筛选 **Lambda**。

- 创建: 使用 `container | filter_lambda` 语法创建。
- 操作:
  - 访问: `keyof set_obj` 返回源容器， `valueof set_obj` 返回筛选 Lambda。
  - 包含判断: `value in set_obj` 检查 `value` 是否在 源容器 中（不应用筛选）。
  - 迭代: 迭代 源容器 的元素（不应用筛选）。
  - 比较: `==` 比较源容器和筛选 Lambda 是否都相等。
  - 求值： `collect set_obj` 返回一个新的元组，包含源容器中所有满足筛选 Lambda 条件的元素。

注意: 由于 XLangVM 的一些机制， `in` 关键字只能对惰性筛选器进行使用，对元组等容器对象使用 `in` 关键字会导致虚拟机抛出异常。如果需要判断一个值是否在元组中，可以使用如下方法：

xlang

```xlang
tuple := (1, 2, 3);
set := tuple | (x?) -> true; // 筛选条件永远为真
assert(1 in set); // 这行代码会正常执行
// assert(1 in tuple); // 这行代码会抛出异常
```

指令集 (instructions)

- 描述: 存储由 XLang-Rust 编译器生成的虚拟机指令包。
- 创建: 通常由 `import "path/to/compiled.xbc"` 语句创建并赋值给变量。
- 操作: 主要供虚拟机内部使用，作为 `lambda` 函数的执行体。用户通常不直接操作其内容。

下面是一个示例：

xlang

```xlang
module_instructions := import "path/to/compiled.xbc"; // 导入编译后的指令集
module_lambda := () -> dyn module_instructions; // 定义一个 lambda 函数，使用导入的指令集作为函数体
result := module_lambda(); // 调用 lambda 函数，执行导入的指令集
print(result); // 输出结果
```

C Lambda 指令 (clambda)

- 描述: 表示一个已加载的外部 C 动态链接库（DLL/SO）的接口。
- 创建: 由 `load_clambda("path/to/library")` 语句创建并赋值给变量。
- 操作: 主要供虚拟机内部使用，允许 `lambda` 函数调用 C 库中定义的函数。用户通常不直接操作其内容。

使用 C Lambda 的方式和使用普通的指令集类似。唯一区别在于其调用时会将 Lambda 的第一个别名当作函数签名并调用 C 库中的 `clambda_{signature}` 函数。

下面是一个示例：

xlang

```xlang
module_clambda := load_clambda("path/to/library"); // 加载 C 动态链接库
module_lambda := myfunction::() -> dyn module_clambda; // 定义一个 lambda 函数，使用加载的 C 库作为函数体，函数签名是 `myfunction`
result := module_lambda(); // 调用 lambda 函数，执行加载的 C 库
print(result); // 输出结果
```

元组

XLang-Rust 的元组是一个有序的元素集合，可以包含不同类型的元素。元组是一串使用逗号 `,` 分隔的表达式。元组的值和长度是可变的。

尽管元组的定义不要求显式添加括号（因为括号只是用来更改运算顺序，元组构建仅依赖于逗号分隔），但为了避免歧义，建议在定义元组时使用括号。

元组的构建会跳过空表达式，因此 `1, 2, 3` 和 `1, , 2, 3` 都是合法的元组定义并且等价。

元组和其他语言的列表类似，但是考虑到构建元组的语法，称其为元组更为合适。

行为

元素访问

可以使用 `tuple[indx]` 来访问元组的元素。索引从 0 开始。

xlang

```xlang
tuple := (1, 2, 3); // 定义一个元组
print(tuple[0]); // 输出 1
print(tuple[1]); // 输出 2
print(tuple[2]); // 输出 3
tuple[0] = 4; // 修改元组的第一个元素
print(tuple[0]); // 输出 4
```

可以使用 `tuple.key` 来访问元组的命名参数以及键值对。具体行为是：VM 会在元组中查找键为 `key` 的命名参数或者键值对，如果找到，则返回对应的 `value`。如果没有找到，则 `raise` 一个异常。

默认情况下，AST 会视 `tuple.key` 中的 `key` 为一个字符串字面量（`"key"`），但如果 `key` 是一个变量，如果要显式动态生成键，可以使用帧作用域包裹表达式来避免 AST 的错误解析。

xlang

```xlang
tuple := {
    'A0' ⇒ 1,
    'B0' : 2,
    'C0' : 3
};
assert (tuple.A0 == 1); // 访问命名参数 A0
assert (tuple.(A0) == 1); // 访问命名参数 A0，括号仅仅是为了改变运算顺序
assert (tuple.{'B' + '0'} == 2); // 访问键值对 B0
tuple.A0 = 4; // 修改命名参数 A0 的值
assert (tuple.A0 == 4); // 验证修改成功
```

可以使用内建的 `len(tuple)` 函数来获取元组的长度。

Lambda 参数赋值

当调用一个 Lambda 函数时，传递给它的参数（构成一个调用参数元组）会按照特定规则赋值给 Lambda 定义时声明的参数（构成 Lambda 的参数元组）。赋值过程如下：

1. 参数分类：首先，将调用时提供的参数分为两类：
    1. 命名参数：形如 `key ⇒ value`
    2. 位置参数：其他没有显式指定键的参数值。
2. 处理命名参数：

1. 系统会遍历调用时提供的所有命名参数。
2. 对于每一个命名参数，会在 Lambda 定义的参数元组中查找具有相同键的参数。
3. 如果找到了匹配的键，则将调用时命名参数的值赋给 Lambda 定义中对应键的参数。
4. 如果在 Lambda 定义的参数元组中没有找到匹配的键，则这个来自调用的命名参数（键和值）会被添加到当前 Lambda 调用的参数元组中。

3. 处理位置参数：
   1. 在处理完所有命名参数后，系统会按顺序处理调用时提供的 位置参数。
   2. 对于每一个位置参数，系统会在 Lambda 定义的参数元组中查找下一个 尚未被赋值 的参数槽位。
   3. 如果找到了这样的槽位，则将该位置参数的值赋给这个槽位。
   4. 如果 Lambda 定义的参数元组中所有槽位都已被赋值（无论是通过匹配命名参数还是先前的位置参数），那么多余的位置参数会被 追加 到当前 Lambda 调用的参数元组末尾。

这个赋值过程允许灵活地混合使用命名参数和位置参数，优先通过键匹配，然后按顺序填充剩余的位置，并能动态扩展参数列表以容纳额外的参数。

Lambda 函数

XLang-Rust 的设计思想是完全抛弃传统的固定分配函数名称的方式，转而完全使用 lambda 函数（匿名函数）来实现函数的定义和调用。这意味着函数的定义和调用都是动态的。

定义

XLang-Rust 的函数定义使用 `→` 符号。函数可以接受参数，并返回一个值。函数体可以是一个表达式。

具体来说，XLang-Rust 的函数定义语法如下：

```
atomic_expression → <&atomic_expression> <dyn> expression
```

其中 `atomic_expression` 是一个原子表达式，表示函数的参数列表；`<&atomic_expression>` 表示函数体中可以捕获的变量；`<dyn>` 表示动态生成指令集的标志；`expression` 是函数体的表达式。尖括号内的内容是可选的。

XLang-Rust 要求静态定义的函数的所有参数都必须是命名参数（即 `key ⇒ value` 形式）。

为了使得代码更加简洁，XLang-Rust 提供了两个语法糖 `x?` 和 `x!`，分别等价于 `x ⇒ null` 和 `x ⇒ x`。

xlang

```
// 定义一个简单的函数，接受一个参数并返回其平方
square := (x ⇒ 0) → x * x; // 定义一个函数 square，接受一个参数 x，返回 x 的平方

// 然后调用这个函数
result := square(5); // 调用 square 函数，传入参数 5，结果为 25
```

XLang-Rust 的 Lambda 函数有以下特性：

- Lambda 的参数绝对是一个元组（tuple）。可以使用 `keyof lambda` 来获取参数元组（一般是完全由命名参数组成）。
- 在静态分析模式下，Lambda 的参数必须是命名参数（即 `key ⇒ value` 形式）。除非在定义时使用 `@dynamic` 注解绕过静态分析。

- 除非 Lambda 被 GC 回收，否则其参数和返回值会持续存在于内存中，不随着函数调用的结束而消失。
- Lambda 的参数和返回值可以是任意类型，这保证了函数的灵活性和可扩展性。

也就是说下面的代码是合法的：

xlang

```
// 定义一个函数，接受一个参数并返回其平方
square := (x ⇒ 0) → x * x; // 定义一个函数 square，接受一个参数 x，返回 x 的平方
square(5);

print(valueof square); // 打印函数 square 缓存的值，输出 `5`
```

这也意味着 XLang-Rust 其实将 Lambda 视为一种可以动态计算的键值对，其值只在被调用时才会被计算并在此之后长期缓存。

Lambda 函数的调用

XLang-Rust 的函数调用使用 `()` 符号。函数调用的参数可以是任意类型的表达式，包括其他函数的返回值。

一般情况下，AST 会将调用里的非元组表达式转换为元组表达式以适配单参数函数调用。但是这种情况导致了一个问题： 如果函数的参数是单个的元组，那么 AST 会认为这个元组是实际上的传参，而不是函数的参数。如下代码所示：

xlang

```
lambda((1, 2, 3));// 它实际上是传递了三个参数，而不是一个元组参数！最外一次括号仅仅是为了改变运算顺序

// 如果想要传递一个元组参数，可以使用以下方式：
lambda((1, 2, 3),); // 传递一个元组参数（使用逗号告诉AST这是一个单参数的调用）
```

如果想要将一个元组变量的值当作参数传递给函数，可以使用 `...` 符号来解包元组（仅限单参数）

xlang

```
lambda(...(1, 2, 3)); // 将元组 (1, 2, 3) 解包为三个参数传递给函数
```

同理 `...` 也可以用在 Lambda 的定义上，表示这个 Lambda 的参数定义是由一个元组的值提供的

xlang

```
params := (A?, B?); // 定义一个元组参数
foo := (...params) → @dynamic { // 定义一个函数，参数是一个元组
    print(A); // 打印 A 的值
    print(B); // 打印 B 的值
};
foo(1, 2); // 调用函数，传递参数 1 和 2
```

一旦 Lambda 被调用，VM 先会创建一个函数作用域，然后尝试解包参数并尝试将所有的命名参数挂载到当前的作用域上（成功与否在于键是否是一个字符串），之后开始执行函数体。

异步任务与并发

异步任务的定义

XLangVM 实现了一种单线程的协作式并发模型。任何使用 VM 指令集的 Lambda 函数都可以通过 `async` 关键字启动为一个独立的异步任务。

异步任务一旦启动，就会在 VM 的调度下运行，直到它执行完成或遇到 `await` 语句而暂停。这种模型允许在等待操作（如等待其他任务完成）时切换执行其他任务，从而实现并发，但并非真正的并行执行。其行为模式更接近于管理在单个线程上交错执行的异步操作。

异步任务的启动

使用 `async lambda()` 语法来启动一个新的异步任务。此操作会立即将任务提交给 VM 调度器执行，并返回 `lambda` 对象自身。

xlang

```
my_async_task := () → {
    // 这里是异步任务的代码
    return 42; // 任务完成时返回一个值
};

async my_async_task(); // 启动异步任务
```

任务最终的返回值（通过 `return` 或最后一个表达式的值）会被缓存到其对应的 Lambda 对象中，直到被 GC 回收。

VM 会持续运行，直到所有活动的异步任务都执行完毕。

emit 语句

使用 `emit expr` 语句可以设置当前正在执行的异步任务的当前返回值（可以通过 `valueof` 在任务完成前读取），但这并不会暂停或停止任务的执行。任务会继续执行后续代码。

xlang

```
my_task := () → {
    emit 42; // 设置当前返回值为 42，但任务继续执行
    // ... 其他代码 ...
    return 100; // 最终返回值为 100
};
```

其他任务可以随时使用 `valueof task_lambda` 来获取目标任务当前缓存的返回值（可能是 `emit` 设置的，也可能是最终 `return` 的）。

await 语句

使用 `await task_lambda` 语句会暂停当前异步任务的执行，并将控制权交还给 VM 调度器，直到被 `await` 的 `task_lambda` 任务执行完成。`await` 语句本身的求值结果是已完成任务的最终返回值。 `await` 是实现协作式调度的关键，它允许任务在等待其他任务时主动让出执行权。

行为特性

- 立即执行: 异步任务一旦通过 `async` 创建，就会被调度并尽快开始执行。
- 独立作用域: 启动的异步任务不会继承调用者的作用域。其初始作用域仅包含传递给它的参数和内建函数。这有助于减少任务间的副作用。

赋值

对 Lambda 的赋值仅仅只会将键（参数），值（返回值）和上下文（捕获的变量）赋值到指定的对象上。

对象绑定

XLang-Rust 提供关键字 `bind obj` 来将一个元组里的所有 Lambda 以及所有命名参数的值（如果是 Lambda）的 `self` 引用绑定为该元组自身。因此可以使用 `bind` 关键字来模拟类的行为。

xlang

```
obj := bind {
    'v' : 'Hello World',
    say ⇒ () → {
        print(self.v); // 访问绑定的对象的属性
    }
};
obj.say(); // 调用绑定的对象的"方法"
```

self 变量

`self` 变量是一个特殊的变量，用于指代当前 Lambda 所绑定的对象。它可以在 Lambda 内部被访问和使用。 `self` 变量的值是由 `bind` 关键字绑定的对象。

this 变量

`this` 变量是一个特殊的变量，用于指代当前 Lambda 自身。它可以在 Lambda 内部被访问和使用。

xlang

```
fib := (n ⇒ 0) → {
    if n < 2 {
        return n; // 返回 n
    } else {
        return this(n - 1) + this(n - 2); // 递归调用自身
    }
};
print(fib(10)); // 输出斐波那契数列的第 10 项
```

return 语句

`return` 语句用于从当前函数或协程中返回一个值。它会立即终止函数的执行，并将指定的值作为结果缓存到函数对象中然后返回。 `return` 语句可以在函数的任何位置使用。

dyn 标记

使用 `dyn` 标记可以将一个函数标记为动态生成的函数。动态生成的函数会在运行时根据传入的指令集进行 Lambda 的构建。

模块

每一个 XLang-Rust 的源文件都是一个“模块”，其被编译成 VM 字节码文件。当 VM 加载一个模块时，它会执行如下操作：

1. 载入模块的字节码文件，并将其装载进 VM
2. 隐式构建一个 Lambda 对象，指令集为该模块的字节码
3. 执行这个隐式构建的 Lambda 对象

上述过程等价于下面的代码：

```xlang
__new_module := import "path/to/module.xbc"; // 载入模块的字节码文件
__entry := () → dyn __new_module; // 隐式构建一个Lambda对象，指令集为该模块的字节码
__entry(); // 执行这个隐式构建的Lambda对象
```

同时上述代码也是 XLang-Rust 实现模块化的基础。

变量捕获

XLang-Rust 支持在 Lambda 函数中捕获外部变量。使用 `&` 符号来捕获一个原子表达式求值的结果。

可以使用 `captureof lambda` 或者 `$lambda` 来获取 Lambda 函数捕获的变量。

```xlang
a := 1; // 定义一个变量 a
foo := () → &a {
    print($this); // 打印本函数捕获的变量 a 的值
};
foo(); // 调用函数 foo，输出 1
print(captureof foo); // 打印捕获的变量 a 的值，输出 1。也可以写为 `print($foo)`。
```

当然，我们也可以暴力使用参数来捕获变量：

```xlang
a := 1; // 定义一个变量 a
foo := (a!) → {
    print(a); // 打印参数 a 的值
};
foo(); // 调用函数 foo，输出 1
```

区别在于：

- 使用 `&` 捕获的变量是一个纯引用，一旦构建不可能对其类型进行修改。
- 使用参数捕获的变量和一般的变量一样，可以使用 `:=` 来遮蔽掉原有的变量。
- `copy` 和 `deepcopy` 不能保证复制的参数的引用指向原值，但其一定不会影响捕获的变量并且保证 Lambda 复制后的捕获变量是一致的

一半来说，XLang 会尝试自动将跨函数调用的变量捕获为 `var!` 形式的参数传递给 Lambda 函数。这样可以避免在函数调用时出现作用域问题。但也可以使用 `@dynamic` 和 `@required` 来阻止自动捕获。

内置函数

XLang-Rust 提供了一些内置函数来操作和处理数据。以下是一些常用的内置函数：

- `print(value)`：打印值到标准输出。
- `len(value)`：返回值的长度或大小。现在可以使用 `lengthof value` 来获取长度。但为了兼容性，`len` 仍然是一个内置函数。
- `int(value)`：将值转换为整数。
- `float(value)`：将值转换为浮点数。
- `string(value)`：将值转换为字符串。
- `bool(value)`：将值转换为布尔值。
- `bytes(value)`：将值转换为字节序列。

- `input(value)`：从标准输入读取值。先输出提示信息，然后等待用户输入。输入的值会被转换为字符串。
- `load_clambda(path)`：加载一个 C 动态链接库（DLL/SO），并返回一个 clambda 对象。
- `json_encode(value)`：将值编码为 JSON 字符串。
- `json_decode(value)`：将 JSON 字符串解码为值。

注意：

1. 所有内置函数都是 Lambda 对象，其只在主协程被创建时绑定在初始的作用域上。因此可以被遮蔽。
2. XLang-Rust 只保证由编译器启动的代码存在上述内置函数的绑定。嵌入到 Rust 内部执行的代码并不保证存在上述内置函数的绑定，内置函数可以是任意的，只要在 VM 启动的主线程里绑定就可以被使用（异步任务除外，为了保证纯度，异步任务只接受参数绑定和捕获变量）

考虑到 Lambda 实现的特殊性（参数缓存），建议用以下代码包装一个自动擦除参数的 Lambda 对象防止参数出现类型错误

xlang

```
builtins := bind {
    'builtin_print' : print,
    'builtin_int' : int,
    'builtin_float' : float,
    'builtin_string' : string,
    'builtin_bool' : bool,
    'builtin_bytes' : bytes,
    'builtin_input' : input,
    'builtin_len' : len,
    'builtin_load_clambda' : load_clambda,
    'builtin_json_decode' : json_decode,
    'builtin_json_encode' : json_encode,
    print ⇒ () → {
        result := self.builtin_print(...keyof this);
        keyof this = (); // 清空参数
        keyof self.builtin_print = (); // 清空参数
        return result;
    },
    int ⇒ () → {
        result := self.builtin_int(...keyof this);
        keyof this = ();
        keyof self.builtin_int = ();
        return result;
    },
    float ⇒ () → {
        result := self.builtin_float(...keyof this);
        keyof this = ();
        keyof self.builtin_float = ();
        return result;
    },
    string ⇒ () → {
        result := self.builtin_string(...keyof this);
        keyof this = ();
        keyof self.builtin_string = ();
        return result;
    },
```

```
    bool ⇒ () → {
        result := self.builtin_bool(...keyof this);
        keyof this = ();
        keyof self.builtin_bool = ();
        return result;
    },
    bytes ⇒ () → {
        result := self.builtin_bytes(...keyof this);
        keyof this = ();
        keyof self.builtin_bytes = ();
        return result;
    },
    len ⇒ () → {
        result := self.builtin_len(...keyof this);
        keyof this = ();
        keyof self.builtin_len = ();
        return result;
    },
    input ⇒ () → {
        result := self.builtin_input(...keyof this);
        keyof this = ();
        keyof self.builtin_input = ();
        return result;
    },
    load_clambda ⇒ () → {
        result := self.builtin_load_clambda(...keyof this);
        keyof this = ();
        keyof self.builtin_load_clambda = ();
        return result;
    },
    json_decode ⇒ () → {
        result := self.builtin_json_decode(...keyof this);
        keyof this = ();
        keyof self.builtin_json_decode = ();
        return result;
    },
    json_encode ⇒ () → {
        result := self.builtin_json_encode(...keyof this);
        keyof this = ();
        keyof self.builtin_json_encode = ();
        return result;
    }
};

builtins.print("Hello World!"); // 打印 "Hello World!"
```

快速调用

XLang-Rust 提供了一个快速调用的语法糖 `#`。

定义为 `#atomic_expression expression`，其中 `atomic_expression` 是一个值为 Lambda 的原子表达式，`expression` 是一个表达式，表示传入的参数。

`#` 语法糖会将 `expression` 作为参数传递给 `atomic_expression`，并返回 `atomic_expression` 的返回值。示例：

```xlang
```

```
foo := (x ⇒ 0) → x * x; // 定义一个函数 foo，接受一个参数 x，返回 x 的平方
A := #foo 5; // 调用 foo 函数，传入参数 5，结果为 25
print(A); // 输出 25
```

C Lambda

XLang-Rust 支持调用 C 语言编写的动态链接库（DLL/SO）。可以使用 `load_clambda` 函数加载一个 C 动态链接库，并返回一个 clambda 对象。

下面是一个示例：

xlang

```
mathlib := {
    clambda := @dynamic load_clambda("../../modules/clambda_math_lib/clambda_math.so"); // 加载 C 动态链接库
    {
        // 封装，由于 C 库一般不接受命名参数，所以这里包装一层，直接使用位置参数
        sin ⇒ (x?) → &clambda (sin::() → dyn $this)(x),
        cos ⇒ (x?) → &clambda (cos::() → dyn $this)(x),
        tan ⇒ (x?) → &clambda (tan::() → dyn $this)(x),
        pow ⇒ (x?, y?) → &clambda (pow::() → dyn $this)(x, y),
        sqrt ⇒ (x?) → &clambda (sqrt::() → dyn $this)(x),
        round ⇒ (x?) → &clambda (round::() → dyn $this)(x),
        floor ⇒ (x?) → &clambda (floor::() → dyn $this)(x),
        ceil ⇒ (x?) → &clambda (ceil::() → dyn $this)(x),
        log ⇒ (x?) → &clambda (log::() → dyn $this)(x),
        log10 ⇒ (x?) → &clambda (log10::() → dyn $this)(x),
        exp ⇒ (x?) → &clambda (exp::() → dyn $this)(x),
        max ⇒ (x?) → &clambda (max::() → dyn $this)(x),
        min ⇒ (x?) → &clambda (min::() → dyn $this)(x),
        abs ⇒ (x?) → &clambda (abs::() → dyn $this)(x),
        pi ⇒ (pi::() → dyn clambda)(), // 直接调用 C 库中的 pi 函数
        e ⇒ (e::() → dyn clambda)(),
    }
};
print(mathlib.sin(1));
print(mathlib.cos(1));
print(mathlib.tan(1));
print(mathlib.pow(2, 3));
print(mathlib.sqrt(4));
print(mathlib.round(1.5));
print(mathlib.floor(1.5));
print(mathlib.ceil(1.5));
print(mathlib.log(2));
print(mathlib.log10(100));
print(mathlib.exp(1));
print(mathlib.max(1, 2));
print(mathlib.min(1, 2));
print(mathlib.abs(-1));
print(mathlib.pi);
print(mathlib.e);
```

别名系统

由于 XLang-Rust 采用的是结构化类型系统，因此在 XLang-Rust 中，变量的类型是由其值决定的，而不是由其声明时的类型决定的。这使得 XLang-Rust 的数据结构更加灵活和动态。但也

导致了一个问题：如何在显著指定一个对象到底是什么。为了解决这个问题，XLang-Rust 引入了别名系统。

别名系统允许用户为一个对象创建一个别名，这个别名可以在后续的代码中使用。

使用 `alias_name::value` 语法来为对象附加一个别名。这个别名是静态的，不能被修改。

```xlang
A := MyType::(1, 2, 3); // 创建一个对象 A，别名为 MyType
```

可以使用 `aliasof value` 来获取对象的别名元组，返回一个元组，包含对象的所有别名的字符串。

当然别名可以嵌套：

```xlang
A := A::B::(1, 2, 3); // 创建一个对象 A，别名为 MyType
assert(alias of A == ('B', 'A')); // 获取对象 A 的别名
```

使用 `wipe value` 可以浅拷贝一个不带别名的对象

注意: 定义别名的时候会对原对象进行浅拷贝，因此会导致 Lambda 对象丢失 `self` 引用。不建议对带 `self` 引用的 Lambda 对象使用别名。

注解

XLang-Rust 支持使用注解来标记函数或变量的特定属性。注解以 `@` 符号开头，后面跟随注解名称。不支持参数。

下面是所有的注解列表：

- `@dynamic`：标记一个表达式为动态的，允许表达式产生明显的副作用（比如跨函数作用域的变量使用）。AST 在识别到后会认为该表达式是一个动态表达式，并不会进行静态分析。
- `@static`：标记一个表达式为静态的，不允许表达式产生明显的副作用（比如跨函数作用域的变量使用）。AST 在识别到后会认为该表达式是一个静态表达式，并会进行强制静态分析。
- `@compile`：标记一个字符串为要被编译的模块路径。AST 在识别到后会将该字符串编译为一个模块，模块路径相对于当前文件。例如 `@compile "path/to/module.x"`。编译失败会产生警告。
- `@required`：标记一个变量在动态作用域中是必需的。静态分析在识别到后会认为该变量是在运行时一定存在并添加一个占位符号阻止静态分析提示变量不存在。

所有注解都不影响表达式的值，只是影响 AST 的静态分析。

下面是一个示例：

```xlang
x := 1; // 定义一个变量 x
foo := () → {
    @dynamic x = 2; // x 不在当前作用域中，虽然静态分析会自动添加 `x!` 参数，但是也可以直接使用 @dynamic 绕过静态分析并阻止自动添加参数
};
foo2 := () → {
    @required x; // 同样可以使用 @required 来标记某个变量在动态作用域中一定存在
    x = 2;
}
foo(); // 调用函数 foo
print(x); // 输出 2
```

控制流

XLang-Rust 支持常见的控制流语句，包括条件语句、循环语句。

条件语句

XLang-Rust 支持 `if` 语句。

定义为 `if atomic_expression_1 atomic_expression_2 <else expression>`，其中 `atomic_expression_1` 是一个布尔表达式，`atomic_expression_2` 是一个表达式，`<else expression>` 是可选的 else 分支，else 后可携带 AST 能自动匹配最长长度的表达式。`if` 语句的返回值是被执行的分支的值。如果没有分支被执行，则返回 `null`。

```xlang
A := if true 1 else 2; // A 的值为 1
B := if false 1 else 2; // B 的值为 2
C := if false 1; // C 的值为 null

// 当然也可以组合出更复杂的条件语句
/*
if (condition_1) {
    // 执行代码块 1
} else if (condition_2) {
    // 执行代码块 2
} else {
    // 执行代码块 3
}
*/
```

循环语句

XLang-Rust 支持 `while` 循环语句。

定义为 `while atomic_expression_1 expression`，其中 `atomic_expression_1` 是一个布尔表达式，`expression` 是一个 AST 能自动匹配最长长度的表达式。`while` 正常结束循环的返回值为 `null`。

```xlang
i := 0; // 定义一个变量 i
while (i < 10) {
    print(i); // 打印 i 的值
    i := i + 1; // 将 i 加 1
};
// 输出 0 到 9
```

break 语句

`break expression` 语句用于跳出当前循环。它会立即终止循环的执行，并返回携带的值。

continue 语句

`continue expression` 语句用于跳过当前循环的剩余部分，并继续下一次循环。其携带的值不会影响 `while` 循环的返回值。

边界作用域

XLang-Rust 的边界作用域是一个特殊的作用域，用于为 `raise` 语句提供一个跳转点，之前已做介绍，现在用边界作用域来实现一个其他语言常见的 `try...catch` 机制。

xlang

```xlang
Err := (v?) -> bind Err::{
    'result' : v,
    value => () -> self.result,
};
Ok := (v?) -> bind Ok::{
    'result' : v,
    value => () -> self.result,
};
is_err := (v?) -> "Err" in {aliasof v | () -> true};

try := (f?, is_err!) -> bind {
    'result': wrap null,
    value => () -> return valueof self.result,
    catch => (err_handler?, f!, is_err!) -> {
        result := boundary f(...(keyof f));
        if (is_err(result)) {
            err_handler(result);
        } else {
            self.result = result;
        };
        return self;
    },
    finally => (finally_handler?) -> {
        result := boundary finally_handler(...(keyof finally_handler));
        return self;
    },
};

try_catch := (pair?, Ok!) -> {
    return (valueof pair)(keyof pair, boundary {
        return Ok((keyof pair)());
    });
};
```

映射运算

XLang-Rust 支持 `▷` 运算符来进行映射运算。

定义为 `value ▷ lambda`，其中 `value` 是一个可迭代的值（字符串，字节序列，范围，元组），`lambda` 是一个 Lambda 函数。迭代传参和正常的函数调用一样（意味着迭代命名参数的操作是危险的，可能会导致意想不到的结果）。

`▷` 运算符会将 `value` 的每个元素传递给 `lambda` 函数，并返回一个映射后的元组。

xlang

```xlang
// 定义一个函数，接受一个参数并返回其平方
square := (x => 0) -> x * x; // 定义一个函数 square，接受一个参数 x，返回 x 的平方
squared := (1, 2, 3) ▷ square; // 调用 square 函数，传入参数 1，2，3，结果为 (1, 4, 9)
print(squared); // 输出 (1, 4, 9)
```

对于字符串和字节序列，`▷` 运算符会将每个字符或字节传递给 `lambda` 函数。

对于范围，`▷` 运算符会将范围内的每个值传递给 `lambda` 函数。

运算优先级

优先级	运算符/构造	结合性	说明
1 (最低)	`;`	从左到右	表达式分隔符
2	`@`	N/A	注解
3	`return`, `emit`, `raise`	N/A	流程控制
4	`,`	从左到右	元组构造
5	`:=`	从右到左	变量定义
6	`=`	从右到左	赋值
7	`▷`	从左到右	映射运算
8	`|`	从左到右	惰性筛选器定义
9	`#expr`	N/A	快速调用 (语法糖)
10	`→`	从右到左	Lambda 定义
11	`⇒`	从左到右	命名参数
12	`:`	从左到右	键值对
13	`while`	N/A	控制流/块
14	`if`	N/A	控制流/块
15	`break`, `continue`	N/A	控制转移
16	`or`	从左到右	逻辑或/按位或
17	`and`	从左到右	逻辑与/按位与
18	`xor`	从左到右	逻辑异或/按位异或
19	`not`	从右到左	逻辑非/按位非
20	`>`, `<`, `⩾`, `⩽`, `⩵`, `≠`	从左到右	比较运算
21	`in`	从左到右	成员检查
22	二元 `+`, `-`	从左到右	加减

23	`*` , `/` , `%`	从左到右	乘除模
24	`<<` , `>>`	从左到右	位移
25	一元 `+` , `-`	从右到左	一元加减
26	`**`	从右到左	幂运算
27	`..`	从左到右	区间构造
28	`deepcopy` , `copy` , `ref` , `deref` , `keyof` , `valueof` , `selfof` , `assert` , `import` , `wrap` , `typeof` , `wipe` , `aliasof` , `collect` , `captureof` , `bind` , `boundary` , `await` , `lengthof`	N/A	一元修饰符/操作
29	`key?` , `key!`	N/A	快速命名参数 (语法糖)
30	`...`	N/A	展开运算 (元组暗示)
31	`::`	从右到左	别名定义
32	`$expr`	N/A	捕获访问 (语法糖)
33	`.` (属性访问), `[index]` (索引), `(args)` (调用), `async expr (args)` (协程调用)	从左到右	成员访问/调用
34 (最高)	字面量, 标识符, `this` , `self` , `(...)` , `[...]` , `{...}`	N/A	原子值/分组/作用域