# 写给筑梦的 `Milk` 指南

# 写在前面 {#chapter-1}

**谨献给筑梦，以及像她一样帮助过我的人们。**

> 本篇手册受筑梦师V2.0委托创作，以教学解释为目的创作。如有错误之处和不妥的表达，那么还请大家多多包涵～
>
> 正文开始前，请允许我首先向筑梦师V2.0(以下简称 筑梦 )表示感谢。没有你，`Milk` 语言无法诞生。 同时也要感谢雨岚之忆(以下简称 雨岚 )和简律纯，你们在语言诞生之时帮助我敲定了不少语言范式以及用法，陪我彻夜维修 `Milk` 的部分致命性错误。 最后，感谢广大群友对于 `Milk` 的语法细节的意见，没有你们， `Milk` 自然不会如此人性化。

# 我们要做什么？我们能做什么？{#chapter-2}

> 关于 `Milk` 的定位，我重新思考了一下，大概是这么想的：速度高于 python 的 1/10 甚至 1/100 即可，因为我们根本不可能用纯 `Milk` 去搞大量的数据处理，算法，要是要搞我就推荐用 golang 去写得了，安装包也没必要和 lua 一样那么精简，因为我们也不做嵌入式，所以大可不必那么担心性能。

在我们一开始实现一门语言的时候，一定需要确定语言本身的设计模式与应用场景，这将直接影响语言的设计结构。故本章主要介绍语言的定位细节，以及为什么要写这样一门 `Milk` 语言。

# 我们要做什么？{#chapter-2.1}

我开始写这门语言的时候，雨岚和我打了一通通宵电话，确定了一些范式和应用场景上的问题，这也直接促成了我心中的一个想法：我要写一门函数意义上**静态类型**，**弱类型**，**基于golang**，**依附于lua**，**面向对象**的**解释型语言**。

### 静态类型 or 动态类型 {#chapter-2.1.1}

**静态类型语言**在编译时就会进行类型检查。这意味着变量的类型在编写代码时就必须明确声明，并且在编译阶段就会进行类型验证。例如以下 java 语言的代码在编译时会报错，证明 java 是一门静态类型语言。

```
public class Example {
    public static void main(String[] args) {
        int x = 5;
        String s = "Hello";
        x = s; // 编译错误
        System.out.println(x + ", " + s);
    }
}
```

**动态类型语言**在运行时进行类型检查。变量的类型是在运行过程中根据赋值来确定的，而不需要事先声明。例如以下 lua 语言的代码在运行时不会报错，证明 lua 是一门动态类型语言。

```
local x = 5
local s = "Hello"
x = s -- 运行时不报错
print(x, s) --> Hello Hello
```

但是我们这里的静态类型语言和动态类型语言仅指函数意义上的 静态 和 动态 。 也就是说，静态类型语言的函数定义时参数的类型就会被确定下来，传参时会做严格的类型检查。而动态类型语言则不会。

```
func add(a int, b int) int {
    return a + b
}
add(1,2) //此时会检查`1`和`2`的类型是否是int
```

```
```

```
def add(a,b):
    return a+b
add("1",2) # 运行时不报错，不会检查参数类型
```

但是对于赋值等语句，函数意义上的 静态类型语言 则没有强行规定是否要进行类型检查。

```
-- example.milk
function add(a:Number, b:Number):Number
    return a+b;
end
local a = 1;
local b = 2;
add(a,b);
b = "2"; -- 可以进行赋值操作
add(a,b); -- 报错，b不为"Number"
```

```
-- example.milk.lua
function add(a, b)
    assert(type(a) == "number" and type(b) == "number", "参数类型错误") -- 编译器自动生成assert语句
    assert(type(a + b) == "number", "参数类型错误") -- 编译器自动生成assert语句
    return a + b
end

local a = 1
local b = 2
add(a, b)
b = "2" -- 可以进行赋值操作
add(a, b) -- 报错，b不为"number"
```

这点类似于 TypeScript 的混合类型系统，结合了静态和动态类型的优点。使其引入了可选的静态类型检查，同时保留了一部分动态特性。

#### 强类型 or 弱类型 {#chapter-2.1.2}

**强类型语言**严格限制不同类型之间的运算操作，任何不符合类型规则的运算操作都会导致错误。例如以下 golang 代码会在编译时抛出类型错误，证明 golang 是一门强类型语言。

```
package main

import "fmt"
```

```go
func main() {
    var x int = 5
    var s string = "Hello"
    x = x + s // 编译错误
    fmt.Println(x, s)
}
```

**弱类型语言**允许不同类型之间的操作，通常会进行隐式类型转换。例如以下 JavaScript 代码在运行时不会报错，证明 JavaScript 是一门弱类型语言。

```javascript
var x = 5;
var s = "Hello";
x = x + s; // 运行时不报错，x变为字符串"5Hello"
```

在 Milk 语言中，我们采用了**弱类型**的设计，这意味着在大多数情况下，类型转换是自动进行的，不需要显式声明。例如：

```lua
-- example.milk
local x = 5;
local s = "Hello";
x = x + s; -- 不报错，x变为字符串"5Hello"
```

但是在函数定义中，我们仍然保留了类型检查的机制，以确保函数调用的安全性和可靠性。这种设计使得 Milk 语言在保持灵活性的同时，也能在关键部分提供必要的类型安全检查。

### 依附于lua? {#chapter-2.1.3}

有关于Milk和lua的关系，我们可以将其与TypeScript和JavaScript做类比。Milk 基于 gopherlua 项目进行了魔改，使用了与 lua51 相同的虚拟机。这意味着 Milk 代码可以与 lua 代码无缝互操作，并且可以利用 lua 生态系统中的大量库和工具。

> gopherlua 是一个用 Go 语言编写的 Lua 5.1 虚拟机实现。它允许开发者在 Go 项目中嵌入 Lua 解释器，从而利用 Lua 语言的灵活性和简洁性来编写脚本。gopherlua 提供了与原生 Lua 解释器几乎相同的功能，同时还可以与 Go 代码进行无缝交互，使得在 Go 项目中使用 Lua 变得非常方便。

通过这种方式，我们既能享受到 lua 语言的灵活性和高效性，又能在 Milk 中引入更多现代编程语言的特性，如静态类型检查和面向对象编程。这使得 Milk 既适合快速脚本编写，又能用于更复杂的应用开发。

例如，以下是一个简单的 Milk 代码示例，它调用了一个 lua 函数：

```
-- example.milk
function greet(name:String):String
    return "Hello, " .. name
end

local name = "Milk"
print(greet(name)) -- 输出: Hello, Milk
```

在这个示例中，我们定义了一个 Milk 函数 greet，并在 Milk 代码中调用它。由于 Milk 和 lua 共享同一套虚拟机，这种互操作是无缝的。

这种设计使得 Milk 语言既能保持 lua 的简洁和高效，又能引入更多现代编程语言的特性，满足不同开发者的需求。

### 面向对象? {#chapter-2.1.4}

在 Milk 语言中，我们引入了更为完善的面向对象编程范式，以弥补 lua 在这方面的不足。虽然 lua 可以通过元表和函数实现面向对象编程，但其语法和机制相对复杂，不够直观。而 Milk 则提供了更为简洁和直接的面向对象支持，使得开发者可以更方便地定义和使用类与对象。

以下是一个 Milk 代码示例，展示了如何定义和使用类与对象：

```
-- example.milk
tbl = {}
private_tbl = {
    privateData = 10,
    privateFunction= function()
        print("这是一个私有函数");
        print("私有数据:", private_tbl.privateData);
    end
}
tbl.publicMethod = function() -- 公共接口
    print("这是一个公共方法");
    getprivate(tbl):privateFunction(); -- 面向对象式调用
end
setprivate(tbl, private_tbl);
tbl.publicMethod();
-- tbl这个表中根本没有private_tbl这个元素，因为二者在luavm虚拟机中毫无关系，是仅仅通过
golang中的LTable结构体完成的绑定
```

在 Milk 中，我们可以通过 setprivate 和 getprivate 函数来实现私有属性和方法的封装。这种设计使得 Milk 的面向对象编程更加直观和易于使用。

而在 lua 中，实现相同功能的代码如下：

```lua
-- example.lua
function createObject()
  -- 私有变量
  local privateData = 10

  -- 私有函数
  local function privateFunction()
    print("这是一个私有函数")
    print("私有数据:", privateData)
  end

  -- 公共接口
  return {
    publicMethod = function()
      print("这是一个公共方法")
      privateFunction()
    end
  }
end

local obj = createObject()
obj.publicMethod()
-- obj.privateFunction() -- 这会导致错误,因为privateFunction是私有的
```

在这个 lua 示例中，我们使用闭包函数来实现私有属性和方法的封装。通过 createTable 函数，我们创建了一个包含私有属性和方法的表，并返回了访问这些私有属性和方法的接口函数。这种设计使得 lua 也能实现类似 Milk 的私有属性和方法封装。