通常，开发者都会在扩展名为 .cj 的文本文件中编写仓颉程序，这些程序和文件也被称为源代码和源文件，在程序开发的最后阶段，这些源代码将被编译为特定格式的二进制文件。

在仓颉程序的顶层作用域中，可以定义一系列的变量、函数和自定义类型（如 struct、class、enum 和 interface 等），其中的变量和函数分别被称为**全局变量**和**全局函数**。如果要将仓颉程序编译为可执行文件，需要在顶层作用域中定义一个 main 函数作为**程序入口**，它可以有 Array<String> 类型的参数，也可以没有参数，它的返回值类型可以是整数类型或 Unit 类型。

**注意：**

定义 main 函数时，不需要写 func 修饰符。此外，如果需要获取程序启动时的命令行参数，可以声明和使用 Array<String> 类型参数。

例如在以下程序中，在顶层作用域定义了全局变量 a 和全局函数 b，还有自定义类型 C、D 和 E，以及作为程序入口的 main 函数。

// example.cjlet a = 2023func b() {}struct C {}class D {}enum E { F | G }

main() {

println(a)

}

在非顶层作用域中不能定义上述自定义类型，但可以定义变量和函数，称之为**局部变量**和**局部函数**。特别地，对于定义在自定义类型中的变量和函数，称之为**成员变量**和**成员函数**。

**注意：**

enum 和 interface 中仅支持定义成员函数，不支持定义成员变量。

例如在以下程序中，在顶层作用域定义了全局函数 a 和自定义类型 A，在函数 a 中定义了局部变量 b 和局部函数 c，在自定义类型 A 中定义了成员变量 b 和成员函数 c。

// example.cjfunc a() {

let b = 2023

func c() {

println(b)

}

c()

}

class A {

let b = 2024

public func c() {

println(b)

}

}

main() {

a()

A().c()

}

运行以上程序，将输出：

2023

2024

## 变量

在仓颉编程语言中，一个变量由对应的变量名、数据（值）和若干属性构成，开发者通过变量名访问变量对应的数据，但访问操作需要遵从相关属性的约束（如数据类型、可变性和可见性等）。

变量定义的具体形式为：

修饰符 变量名: 变量类型 = 初始值

其中**修饰符**用于设置变量的各类属性，可以有一个或多个，常用的修饰符包括：

* 可变性修饰符：let 与 var，分别对应不可变和可变属性，可变性决定了变量被初始化后其值还能否改变，仓颉变量也由此分为不可变变量和可变变量两类。
* 可见性修饰符：private 与 public 等，影响全局变量和成员变量的可引用范围，详见后续章节的相关介绍。
* 静态性修饰符：static，影响成员变量的存储和引用方式，详见后续章节的相关介绍。

在定义仓颉变量时，可变性修饰符是必要的，在此基础上，还可以根据需要添加其他修饰符。

* **变量名**应是一个合法的仓颉标识符。
* **变量类型**指定了变量所持有数据的类型。当初始值具有明确类型时，可以省略变量类型标注，此时编译器可以自动推断出变量类型。
* **初始值**是一个仓颉表达式，用于初始化变量，如果标注了变量类型，需要保证初始值类型和变量类型一致。在定义全局变量或静态成员变量时，必须指定初始值。在定义局部变量或实例成员变量时，可以省略初始值，但需要标注变量类型，同时要在此变量被引用前完成初始化，否则编译会报错。

例如，下列程序定义了两个 Int64 类型的不可变变量 a 和可变变量 b，随后修改了变量 b 的值，并调用 println 函数打印 a 与 b 的值。

main() {

let a: Int64 = 20

var b: Int64 = 12

b = 23

println("${a}${b}")

}

编译运行此程序，将输出：

2023

如果尝试修改不可变变量，编译时会报错，例如：

main() {

let pi: Float64 = 3.14159

pi = 2.71828 // Error, cannot assign to immutable value

}

当初始值具有明确类型时，可以省略变量类型标注，例如：

main() {

let a: Int64 = 2023

let b = a

println("a - b = ${a - b}")

}

其中变量 b 的类型可以由其初值 a 的类型自动推断为 Int64，所以此程序也可以被正常编译和运行，将输出：

a - b = 0

在定义局部变量时，可以不进行初始化，但一定要在变量被引用前赋予初值，例如：

main() {

let text: String

text = "仓颉造字"

println(text)

}

编译运行此程序，将输出：

仓颉造字

在定义全局变量和静态成员变量时必须初始化，否则编译会报错，例如：

// example.cjlet global: Int64 // Error, variable in top-level scope must be initialized

// example.cjclass Player {

static let score: Int32 // Error, static variable 'score' needs to be initialized when declaring

}

### [值类型和引用类型变量](https://docs.cangjie-lang.cn/docs/0.53.18/user_manual/source_zh_cn/basic_programming_concepts/program_structure.html" \l "%E5%80%BC%E7%B1%BB%E5%9E%8B%E5%92%8C%E5%BC%95%E7%94%A8%E7%B1%BB%E5%9E%8B%E5%8F%98%E9%87%8F)

从编译器实现层面看，任何变量总会关联一个值（一般是通过内存地址/寄存器关联），只是在使用时，对有些变量，将直接取用这个值本身，这被称为**值类型变量**，而对另一些变量，将这个值作为索引、取用这个索引指示的数据，这被称为**引用类型变量**。值类型变量通常在线程栈上分配，每个变量都有自己的数据副本；引用类型变量通常在进程堆中分配，多个变量可引用同一数据对象，对一个变量执行的操作可能会影响其他变量。

从语言层面看，值类型变量对它所绑定的数据/存储空间是独占的，而引用类型变量所绑定的数据/存储空间可以和其他引用类型变量共享。

基于上述原理，在使用值类型变量和引用类型变量时，会存在一些行为差异，以下几点值得注意：

1. 在给值类型变量赋值时，一般会产生拷贝操作，且原来绑定的数据/存储空间被覆写。在给引用类型变量赋值时，只是改变了引用关系，原来绑定的数据/存储空间不会被覆写。
2. 用 let 定义的变量，要求变量被初始化后都不能再赋值。对于引用类型，这只是限定了引用关系不可改变，但是所引用的数据是可以被修改的。

在仓颉编程语言中，class 和 Array 等类型属于引用类型，其他基础数据类型和 struct 等类型属于值类型。

例如，以下程序演示了 struct 和 class 类型变量的行为差异：

struct Copy {

var data = 2012

}

class Share {

var data = 2012

}

main() {

let c1 = Copy()

var c2 = c1

c2.data = 2023

println("${c1.data}, ${c2.data}")

let s1 = Share()

let s2 = s1

s2.data = 2023

println("${s1.data}, ${s2.data}")

}

运行以上程序，将输出：

2012, 2023

2023, 2023

由此可以看出，对于值类型的 Copy 类型变量，在赋值时总是获取 Copy 实例的拷贝，如 c2 = c1，随后对 c2 成员的修改并不影响 c1。对于引用类型的 Share 类型变量，在赋值时将建立变量和实例之间的引用关系，如 s2 = s1，随后对 s2 成员的修改会影响 s1。

如果将以上程序中的 var c2 = c1 改成 let c2 = c1，则编译会报错，例如：

struct Copy {

var data = 2012

}

main() {

let c1 = Copy()

let c2 = c1

c2.data = 2023 // Error, cannot assign to immutable value

}

## [作用域](https://docs.cangjie-lang.cn/docs/0.53.18/user_manual/source_zh_cn/basic_programming_concepts/program_structure.html" \l "%E4%BD%9C%E7%94%A8%E5%9F%9F)

在前文中，初步介绍了如何给仓颉程序元素命名，实际上，除了变量，还可以给函数和自定义类型等命名，在程序中将使用这些名字访问对应的程序元素。

但在实际应用中，需要考虑一些特殊情况：

* 当程序规模较大时，那些简短的名字很容易重复，即产生命名冲突。
* 结合运行时考虑，在有些代码片段中，另一些程序元素是无效的，对它们的引用会导致运行时错误。
* 在某些逻辑构造中，为了表达元素之间的包含关系，不应通过名字直接访问子元素，而是要通过其父元素名间接访问。

为了应对这些问题，现代编程语言引入了“作用域”的概念及设计，将名字和程序元素的绑定关系限制在一定范围里。不同作用域之间可以是并列或无关的，也可以是嵌套或包含关系。一个作用域将明确开发者能用哪些名字访问哪些程序元素，具体规则是：

1. 当前作用域中定义的程序元素与名字的绑定关系，在当前作用域和其内层作用域中是有效的，可以通过此名字直接访问对应的程序元素。
2. 内层作用域中定义的程序元素与名字的绑定关系，在外层作用域中无效。
3. 内层作用域可以使用外层作用域中的名字重新定义绑定关系，根据规则 1，此时内层作用域中的命名相当于遮盖了外层作用域中的同名定义，对此可以称内层作用域的级别比外层作用域的级别高。

在仓颉编程语言中，用一对大括号“{}”包围一段仓颉代码，即构造了一个新的作用域，其中可以继续使用大括号“{}”包围仓颉代码，由此产生了嵌套作用域，这些作用域均服从上述规则。特别的，在一个仓颉源文件中，不被任何大括号“{}”包围的代码，它们所属的作用域被称为“顶层作用域”，即当前文件中“最外层”的作用域，按上述规则，其作用域级别最低。

**注意：**

仓颉不允许使用单独的大括号“{}”，大括号必须依赖 if、match、函数体、类体、结构体等其他语法结构存在。

例如在以下名为 test.cj 的仓颉源文件里，在顶层作用域中定义了名字 element，它和字符串“仓颉”绑定，而 main 和 if 引导的代码块中也定义了名字 element，分别对应整数 9 和整数 2023。由上述作用域规则，在第 4 行，element 的值为“仓颉”，在第 8 行，element 的值为 2023，在第 10 行，element 的值为 9。

// test.cjlet element = "仓颉"main() {

println(element)

let element = 9

if (element > 0) {

let element = 2023

println(element)

}

println(element)

}

运行以上程序，将输出：

仓颉

2023

9