仓颉编程语言 入门教程

作者：仓颉编程语言布道师 刘俊杰

你好，仓颉

```
// hello.cj
main() {
    println("你好，仓颉")
}
```

```
> cjc hello.cj -o hello
> ./hello
你好，仓颉
```

一、基本概念

标识符

普通标识符

cangjie　　cangjie_2024_06

由英文字母开头，后接零至多个英文字母、数字或下划线。

~~&address~~　　cangjie2024

~~_2023~~　　_c

由一至多个下划线开头，后接一个英文字母，最后可接零至多个英文字母、数字或下划线。

_cangjie　　_c919　　_o_o_

原始标识符是在普通标识符或关键字的外面加上一对反引号，主要用于将关键字作为标识符的场景。

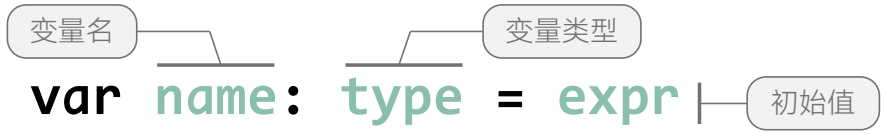
`if`

`while`

`cangjie2024`

变量

变量将一个名字和一个特定类型的值关联起来。



可变变量 `var name: type = expr`

```
var quantum: Int8 = 0
quantum = Random().nextInt8()
```

不可变变量 `let name: type = expr`

```
let result: Int8 = observe()
```

运行时求值

常量 `const name: type = expr`$_{const}$

```
const Planck = 6.626 * 10.0 ** -34
```

编译时求值

当初始值具有明确类型时，可以省略变量类型标注，编译器会自动推断出变量类型。

变量 估算圆周率

```
from std import random.*
from std import math.*

main() {
    const N = 100000
    var n: UInt32 = 0
    let random = Random()
    for (_ in 0..N) {
        let x = random.nextFloat64()
        let y = random.nextFloat64()
        if ((x - 0.5) ** 2 + (y - 0.5) ** 2 < 0.25) {
            n++
        }
    }
    let pi = Float64(n) / Float64(N) * 4.0
    println("π ≈ ${pi}")
    println("deviation: ${abs(Float64.PI - pi)}")
}
```

定义整型常量

定义可变变量，并标注变量类型

定义不可变变量，类型由初值表达式确定

修改可变变量的值

读取变量的值

```
> cjc example.cj -o example
> ./example
π ≈ 3.148600
deviation: 0.007007
```

类型

类型就像一份协议，规定了一块数据的组织结构及相应的解析/操作方式。

X = `01000000010010001111010111000011`

Type	`Float32`	`Int32`
Value	`3.14`	`983644148`
X + X	`6.28` `01110101010000100110111111101000`	`1967288296` `00111011001000010011011111110100`

相同的数据，赋予不同的类型/协议，解析和操作结果并不相同。

如果程序中传递的变量不具有类型信息，就可能导致数据误读/误操作等问题，产生预期之外的运行结果。

```
func f(x) {
    ...
}
```

仓颉编程语言是**静态强类型语言**，具有完备的类型系统，在编译时通过类型检查避免数据误用问题，并提升代码的可维护性。

基础数据类型

整数类型	Int8	Int16	Int32	Int64	UInt8	UInt16	UInt32	UInt64
字面量后缀	i8	i16	i32	i64	u8	u16	u32	u64

整数类型

```
let a: Int64 = 2024
let b = 67u8
```

浮点数类型

浮点数类型	Float16	Float32	Float64
字面量后缀	f16	f32	f64

```
let c: Float64 = 6.21
```

布尔类型

```
let d: Bool = true || false
```

字符类型

```
let e: Rune = '仓'
let f: Rune = '\u{9889}'
```

以 Unicode 值定义字符

字符串类型

```
let g: String = "Cang" + "jie"
let h: String = """
    若到江南赶上春，
        千万和春住。
"""
let i: String = "Cangjie${a}"
```

插值字符串

数组类型

```
let j: Array<Rune> = ['仓', '颉']
let k: VArray<Rune, $2> = ['C', 'J']
```

元组类型

```
let l: (Int64, Float64) = (2024, 6.21)
```

区间类型

```
let m: Range<Int64> = 2019..2024
```

表达式

表达式是可以求值的程序元素，可用于变量赋值、函数传参和返回值等场景。

```
let result = (x ** 2 + y ** 2) ** 0.5

let result = if (x > 2024) { block } else { block }

let result = try { block } catch (e: Exception) { block }

let result = match (color) {
    case Red(value) => block
    case Green(value) => block
    case _ => block
}

let result = data |> fn1 |> fn2 |> fn3

......
```

示例中的 block 表示**代码块**，它代表一个顺序执行流，其中的表达式将按编码顺序依次执行。

$$block := (expr \mid decl_{var})*$$

表达式　　变量声明

在以上求值场景中，**if/try/match** 等表达式的值，等于所执行代码块中最后一个表达式的值。如果代码块是空的，则规定其类型为 **Unit**，**Unit** 类型唯一取值的字面量是 **()**。

if 表达式

```
if (expr_Bool) {
    block
} else if (expr_Bool) {
    block
} else {
    block
}
```

if 分支

*

?

else 分支可以是一个代码块或一个新的 **if** 表达式。

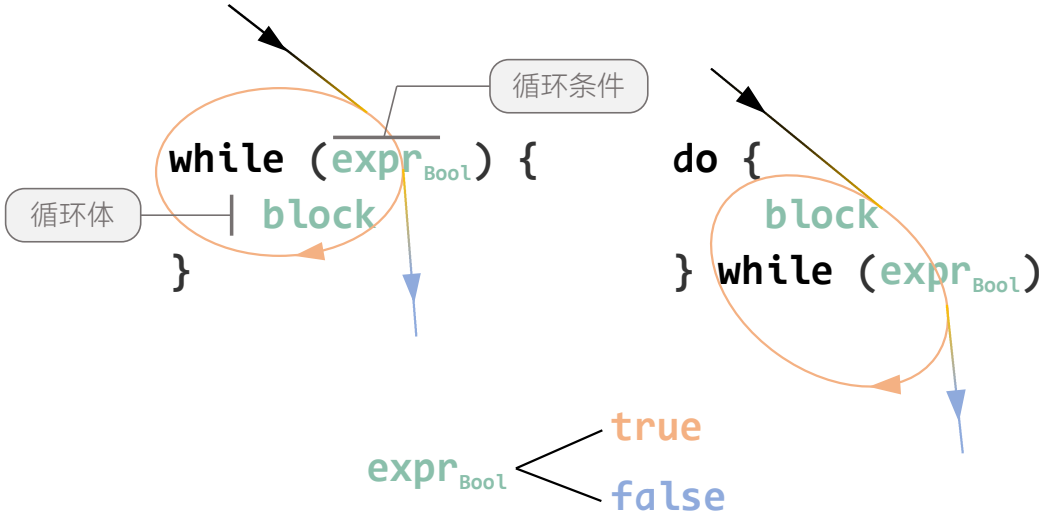
如果 $expr_{Bool}$ 取值为 **true**，将执行 **if** 分支，反之执行 **else** 分支。如果执行了某个分支或没有可选分支，都会跳出 **if** 表达式并执行后续代码。

如果 **if** 表达式具有 **else** 代码块，则 **if** 表达式的值就等于所执行代码块最后一个表达式的值。其他情况的 **if** 表达式类型为 **Unit**。

```
from std import random.*

main() {
    let speed = Random().nextFloat64() * 20.0
    println("${speed} km/s")
    let level = if (speed > 16.7) {
        "第三宇宙速度，鹊桥相会"
    } else if (speed > 11.2) {
        "第二宇宙速度，嫦娥奔月"
    } else if (speed > 7.9) {
        "第一宇宙速度，环游世界"
    } else {
        "脚踏实地，仰望星空"
    }
    println(level)
}
```

```
> cjc example.cj -o example
> ./example
15.004436 km/s
第二宇宙速度，嫦娥奔月
```

while 表达式

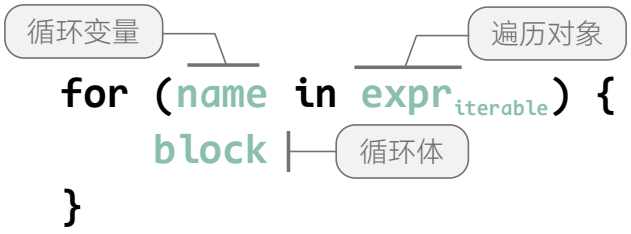


规定 **while** 表达式的类型是 **Unit**。

```
main() {
    var result = 0.0
    var lower = 1.0
    var upper = 2.0

    while (upper - lower > 1.0E-10) {
        result = (lower + upper) / 2.0
        if (result ** 2 - 2.0 > 0.0) {
            upper = result
        } else {
            lower = result
        }
    }
    println("√2 ≈ ${result}")
}
```

```
> cjc example.cj -o example
> ./example
√2 ≈ 1.414214
```

for-in 表达式



遍历对象的类型需要实现迭代器接口 **Iterable<T>**，运行时，将逐次调用迭代器取值并执行循环体，在循环体中可以通过循环变量引用对应值。

规定 **for-in** 表达式的类型是 **Unit**。

```
main() {
    let heaven = ['甲', '乙', '丙', '丁', '戊',
        '己', '庚', '辛', '壬', '癸']
    let earth = ['寅', '卯', '辰', '巳', '午', '未',
        '申', '酉', '戌', '亥', '子', '丑']
    let year = 2024
    // 此年天干序号
    let heavenOfYear = ((year % 10) + 10 - 4) % 10
    // 此年首月天干序号
    var index = (2 * heavenOfYear + 3) % 10 - 1
    println("农历二零二四年各月干支: ")
    for (noumenon in earth) {
        print("${heaven[index]}${noumenon} ")
        index = (index + 1) % 10
    }
}
```

Array<T> 已实现了 Iterable<T> 接口

引用循环变量

```
> cjc example.cj -o example
> ./example
农历二零二四年各月干支:
丙寅 丁卯 戊辰 己巳 庚午 辛未 壬申 癸酉 甲戌 乙亥 丙子 丁丑
```

for-in 表达式

```
var sum = 0
for (i in 1..=99:2) {
    sum += i * i
}
```

遍历对象是 **Range** 表达式。

```
let array = [(1, 2), (3, 4), (5, 6)]
for ((x, y) in array) {
    println("${x}, ${y}")
}
```

如果迭代器取值是元组类型，可以在定义循环变量时进行解构。

```
var number = 2
for (_ in 0..5) {
    number *= number
}
```

如果在循环体中无须引用循环变量，可使用通配符占位。

```
for (i in 0..10 where i % 2 == 1) {
    println(i)
}
```

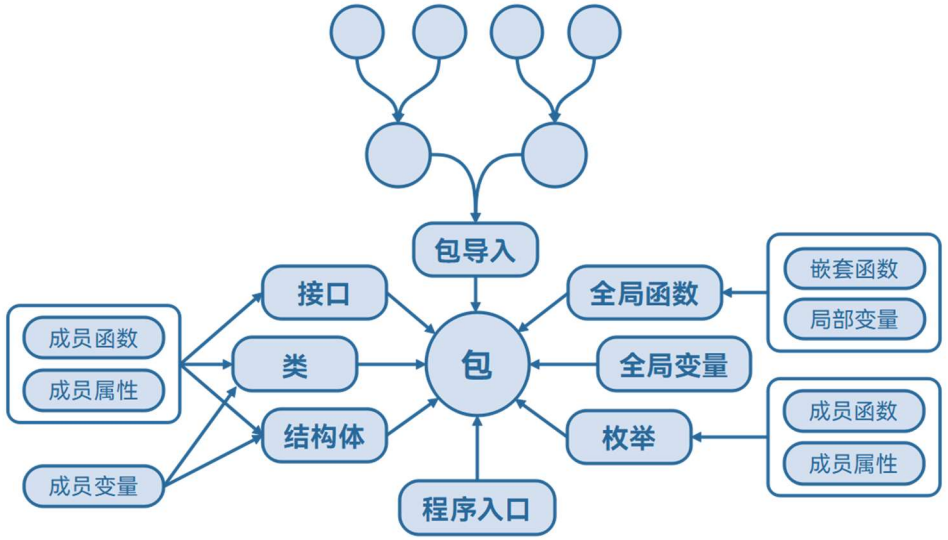
可使用 **where** 引导一个 **Bool** 表达式，取值为 **true** 才会执行循环体。

程序结构

包（package）是仓颉程序的最小编译单元，一个包由一到多个源文件组成，在每个源文件中可以声明当前文件所属包，也可以导入其他包，由此实现程序的高效管理和复用。

在包的顶层作用域中，可以定义一系列的变量、函数和自定义类型（枚举，结构体，类，接口），以及包的声明与导入等，其中的变量和函数被称为**全局变量**和**全局函数**。

在非顶层作用域中可以定义变量和函数，称为**局部变量**和**局部函数**。自定义类型中的局部变量和函数，称为**成员变量**和**成员函数**。



如果要将包编译为可执行文件，需要在顶层作用域中定义一个 **main** 函数作为**程序入口**。

```
main() {
    ...
}
```

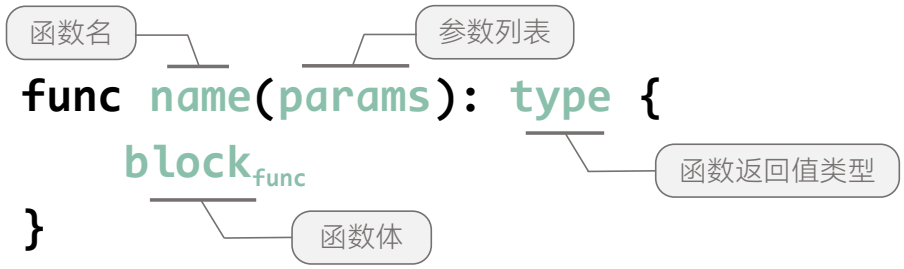
```
main(args: Array<String>) {
    ...
}
```

程序启动参数

返回整型或 Unit 类型值

二、函数

定义函数



$params_{normal}$:= name: type, name: type *

$params_{named}$:= name!: type, name!: type *

$params$:= $params_{normal}$?$params_{named}$?

命名参数只能写在非命名参数之后

可以为命名参数设置默认值 name!: type = $expr_{const}$

$block_{func}$:= (expr | $decl_{var}$ | $decl_{func}$)*

在函数体中还可以定义函数，称为**嵌套函数**。
嵌套函数可以**捕获**其外层作用域中的局部变量，
由此构成**闭包**。

在函数体中**返回值** return expr

函数类型的表达方式

(type, type)* -> type

() -> type

调用函数

实参列表

$name_{func}(args)$

$args := args_{normal}?args_{named}?$

$args_{normal} := expr, expr^*$

$args_{named} := name_{param} : expr, name_{param} : expr^*$

在实参列表中，可以省略有默认值的命名参数，这时对应实参将取其默认值。

函数不仅可以被调用，还可以作为值去使用，如赋值给变量、作为函数的参数和返回值等。

```
> cjc example.cj -o example
> ./example
CDBAFEG
```

```
func void() {}

func node(value: Rune,
    left!: () -> Unit = void, right!: () -> Unit = void) {
    func show() {
        left()
        print(value)
        right()
    }
    return show
}

main() {
    let tree = node('A',
        left: node('B', left: node('C', right: node('D'))),
        right: node('E', left: node('F'), right: node('G')))
    tree()
    return 0
}
```

函数作为参数

嵌套函数

捕获外层局部变量

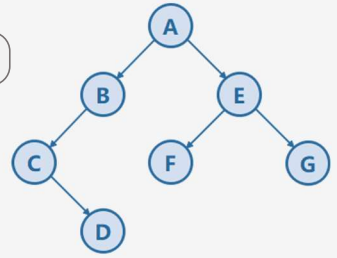
函数作为返回值

函数赋值给变量

调用函数

为命名参数传参

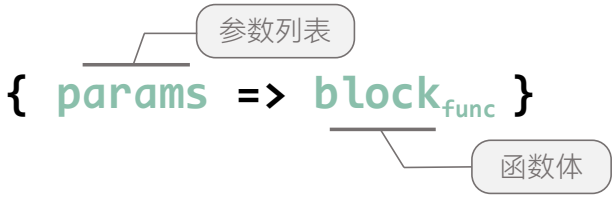
使用参数默认值



这里使用函数实现二叉树，核心是借助**闭包**特性

lambda 表达式

lambda 表达式可以让函数的创建和使用更加灵活， **lambda** 表达式的值就是一个匿名函数。



lambda 表达式中无须标注返回值类型，仓颉编译器会从上下文中自动推导。

```
func iter(n: Int64, x0: Float64, f: (Float64) -> Float64) {
    var x = x0
    for (_ in 0..n) {
        print("${x}, ")
        x = f(x)
    }
    println("${x}")
}

main() {
    iter(5, 0.8, { x: Float64 => 1.0 / (1.0 - x)})
    iter(10, 0.8, { x: Float64 =>
        4.0 * x * (1.0 - x)
    })
}
```

lambda 表达式作为函数参数

```
> cjc example.cj -o example
> ./example
0.800000, 5.000000, -0.250000, 0.800000,
5.000000, -0.250000
0.800000, 0.640000, 0.921600, 0.289014,
0.821939, 0.585421, 0.970813, 0.113339,
0.401974, 0.961563, 0.147837
```

周期 3

周期 ∞，产生伪随机数

应用实例 遍历目录

```cj
from std import fs.*

func forEachFileDo(root: Path, handler: (Path) -> Unit): Unit {
    let current = Directory(root)
    for (file in current.files()) {
        handler(file.path)
    }
    for (directory in current.directories()) {
        forEachFileDo(directory.path, handler)
    }
}

main() {
    forEachFileDo(Path("D:/app/cangjie/tools")) { path: Path =>
        println(path)
    }
}
```

设置遍历对象的处理函数

对当前目录下的每个文件，调用处理函数

递归调用，遍历子目录

打印遍历到的每个文件路径

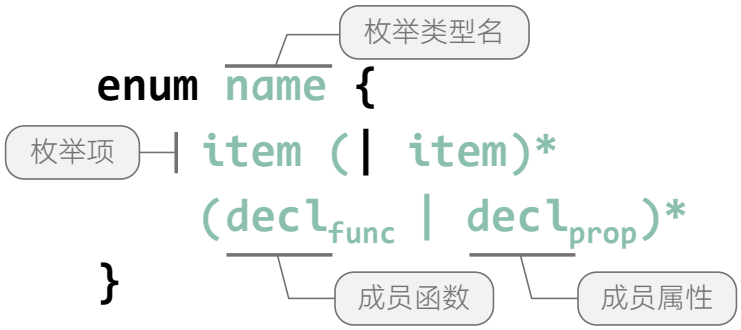
```
> cjc example.cj -o example.exe
> ./example
D:/app/cangjie/tools/bin/cjcov.exe
D:/app/cangjie/tools/bin/cjdb.exe
D:/app/cangjie/tools/bin/cjfmt.exe
D:/app/cangjie/tools/bin/cjlint.exe
D:/app/cangjie/tools/bin/cjpm.exe
...
```

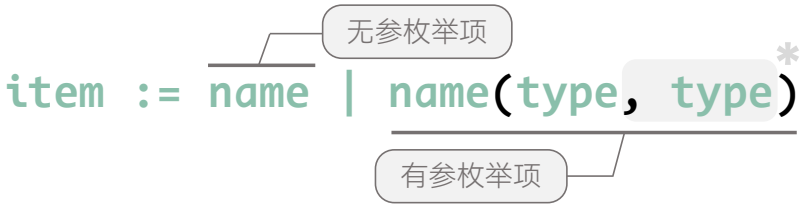
如果需要对不同路径执行相同遍历操作，可以使用**嵌套函数**和**闭包**特性重构左侧函数，这也被称为“函数柯里化”。

```cj
func forEachFileDo(handler: (Path) -> Unit): (Path) -> Unit {
    func processor(root: Path): Unit {
        let current = Directory(root)
        for (file in current.files()) {
            handler(file.path)
        }
        for (directory in current.directories()) {
            processor(directory.path)
        }
    }
    return processor
}

main() {
    let processor = forEachFileDo({ path: Path =>
        println(path)
    })
    processor(Path("D:/app/cangjie/tools"))
    processor(Path("D:/app/cangjie/runtime"))
}
```

三、枚举

定义与实例化





创建枚举实例

$name_{item}$

$name_{item}(args)$

在枚举项名字前也可以添加枚举类型名前缀（由“.”分隔）

成员访问规则

在成员函数和成员属性的声明前可以添加一些**修饰符**

`private` 设置成员仅在枚举类型定义块中可见

`public` 设置成员在枚举定类型定义块内外均可见

`static` 设置成员为**静态成员**，只能通过枚举类型名访问
默认为**实例成员**，只能通过枚举实例访问

在成员函数中都能引用枚举项。在实例成员函数中可以引用其他成员，在静态成员函数中只能引用静态成员。

在实例成员函数中可以使用 **`this`** 变量，它代表当前枚举实例，**`this`** 是不可变变量。

```
> cjc example.cj -o example
> ./example
3421657
12131214121312151213121412131 21
```

```
enum Tree {
    Empty | Leaf(Int64) | Node(Int64, Tree, Tree)

    public func traverse(): Unit {
        match (this) {
            case Empty => ()
            case Leaf(value) => print(value)
            case Node(value, left, right) =>
                left.traverse()
                print(value)
                right.traverse()
        }
    }

    static public func generate(depth: UInt8): Tree {
        if (depth == 1) {
            return Leaf(1)
        }
        return Node(Int64(depth),
            generate(depth - 1), generate(depth - 1))
    }
}

main() {
    let tree = Node(1,
        Node(2, Node(3, Empty, Leaf(4)), Empty),
        Node(5, Leaf(6), Leaf(7)))
    tree.traverse()
    println()
    let fullTree = Tree.generate(5)
    fullTree.traverse()
}
```

使用 **`this`** 变量

实例成员函数

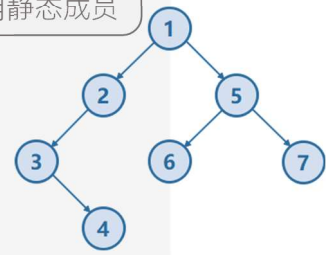
引用枚举项

静态成员函数

引用静态成员

访问实例成员

访问静态成员



match 表达式

```
match(expr) {
    case pattern => block +
}
```

可以用 “|” 连接多个同类型的模式

```
case pattern | pattern *
```

pattern 后还可以用 where 增加约束

```
case pattern where expr_Bool
```

pattern 可以取如下几类模式：

解构枚举项的构造参数

```
枚举模式   case Rot(speed) => rotate(speed)
类型模式   case object: Plane => object.fly()
绑定模式   case other => process(other)
元组模式   case (name, 80) => println(name)
常量模式   case 2024 => println("Cangjie")
通配模式   case _ => default()
```

```
func fib(n: Int64): Int64 {
    match (n) {
        case 0 | 1 => n
        case other where other > 0 =>
            fib(other - 1) + fib(other - 2)
        case _ => 0
    }
}

main() {
    println(fib(-1))
    for (i in 1..=10) {
        print("${fib(i)} ")
    }
}
```

常量模式，匹配 0 或 1

绑定模式，匹配 n > 1

通配模式，匹配 n < 0

```
> cjc example.cj -o example
> ./example
0
1 1 2 3 5 8 13 21 34 55
```

应用实例 表达式计算

```
main() {
    let x = Num(1.2) + Num(3.4) * Num(2.0) - Num(1.0) / Num(2.0)
    println(x.calc())
}
```

```
> cjc example.cj -o example
> ./example
7.500000
```

```
enum Expr {
    Num(Float64) |
    Add(Expr, Expr) | Sub(Expr, Expr) |
    Mul(Expr, Expr) | Div(Expr, Expr)

    public func calc(): Float64 {
        match(this) {
            case Num(number) => number
            case Add(a, b) => a.calc() + b.calc()
            case Sub(a, b) => a.calc() - b.calc()
            case Mul(a, b) => a.calc() * b.calc()
            case Div(a, b) => a.calc() / b.calc()
        }
    }

    public operator func +(that: Expr): Expr {
        return Add(this, that)
    }
    public operator func -(that: Expr): Expr {
        return Sub(this, that)
    }
    public operator func *(that: Expr): Expr {
        return Mul(this, that)
    }
    public operator func /(that: Expr): Expr {
        return Div(this, that)
    }
}
```

用以组织一棵算术运算树

递归计算当前实例对应算术表达式的值

解构出每个算符的操作数

重载加/减/乘/除操作符，简化算术表达式的构造

Option

在部分应用场景中，一个变量无法在整个生命周期内都被赋予有效值，例如存在异常情况或可选的初始化设计等，为了高效且安全地表达这种“或有或无”的值，仓颉语言提供了 **Option** 类型。

```
enum Option<T> {
    None | Some(T)
    public func getOrThrow(): T
    public func getOrThrow(exception: () -> Exception): T
    public func getOrDefault(other: ()->T): T
    public func isNone(): Bool
    public func isSome(): Bool
}
```

表达无值状态

表达有值状态

尝试获取有效值，如果失败就抛出异常

尝试获取有效值，如果失败就执行指定操作

判断当前实例是否持有有效值

创建 Option 实例

```
var result = Some(2024)
var result: Option<Int64> = 2024
var result: ?Int64 = 2024
```

在仓颉语言中，不存在空值或空指针的概念，可能存在无效值的场景只能用 **Option** 去判断处理，避免了空值相关安全问题。

从字符串解析十进制整数

```
func parseInt(text: String): Option<Int64> {
    if (text.isEmpty() || text == "-") {
        return None
    }
    var sign = if (text[0] == 45u8) { 1 } else { 0 }
    var sum = 0
    for (i in sign..text.size) {
        if (text[i] > 57u8 || text[i] < 48u8) {
            return None
        }
        let digit = Int64(text[i] - 48u8)
        sum = 10 * sum + digit
    }
    return if (sign == 1) { -sum } else { sum }
}

main() {
    let number = parseInt("-123456")
    println(number.getOrThrow())
    let result = parseInt("123-456")
    if (result.isNone()) {
        println("parse failed")
    }
}
```

异常情况

异常情况

返回有效值，这里会通过自动类型推导包装为 **Option** 类型

```
> cjc example.cj -o example
> ./example
-123456
parse failed
```

四、结构体

定义与实例化

结构体名

```
struct name {
    constructor* | 构造函数
    (decl_var | decl_func | decl_prop)*
}
```

成员变量　成员函数　成员属性

主构造函数

```
struct Point {
    Point(let x: Float64, let y: Float64) {
        println("Create a point: (${x}, ${y})")
    }
}

main() {
    let p = Point(3.0, 4.0)    // 创建 Point 实例
    println("Visit the point: (${p.x}, ${p.y})")
}
```

访问实例成员

```
> cjc example.cj -o example
> ./example
Create a point: (3.000000, 4.000000)
Visit the point: (3.000000, 4.000000)
```

在结构体中可以定义多个构造函数，它们用于创建结构体实例。

普通构造函数

```
init(params) {
    block_func
}
```

主构造函数

```
name_struct(decl_vars) {
    block_func
}
```

$decl_{vars}$是对成员变量的声明，在此统一了成员变量的定义和初始化，减少冗余编码。

创建结构体实例

```
name_struct(args)
```

成员访问规则

在成员变量、成员函数和成员属性的声明前可以添加一些**修饰符**

- **private** 设置成员仅在结构体内可见
- **public** 设置成员在结构体内外均可见
- **static** 设置成员为**静态成员**，只能通过结构体名访问
  默认为**实例成员**，只能由实例变量访问

在实例成员函数中可以引用其他成员，在静态成员函数中只能引用静态成员。

在实例成员函数中可以使用 **this** 变量，它默认为当前实例的**拷贝**。

如果需要在实例成员函数中修改可变实例成员变量，需要在成员函数前添加 **mut** 修饰符，其中的 **this** 就成为当前实例的**引用**。

```
> cjc example.cj -o example
> ./example
Point
Create a point: (3.000000, 4.000000)
Visit the point: (3.000000, 4.000000)
Visit the point: (1.000000, 2.000000)
```

```
struct Point {
    static let name = "Point"
    public Point(private var x: Float64,
        private var y: Float64) {
        println("Create a point: (${x}, ${y})")
    }

    public func copy() {
        return this
    }

    public mut func set(x: Float64, y: Float64) {
        this.x = x
        this.y = y
    }

    public func show() {
        println("Visit the point: (${x}, ${y})")
    }
}

main() {
    println(Point.name)
    let p1 = Point(3.0, 4.0)
    var p2 = p1.copy()
    p2.set(1.0, 2.0)
    p1.show()
    p2.show()
}
```

this 变量是当前实例的拷贝

mut 成员函数

修改实例成员变量

mut 成员函数中的 this 是当前实例的引用

引用其他成员

访问静态成员

访问实例成员

应用实例 二叉树

定义实例成员变量，存储节点信息

定义实例成员函数，实现中序遍历

静态成员变量

静态构造函数

静态成员函数

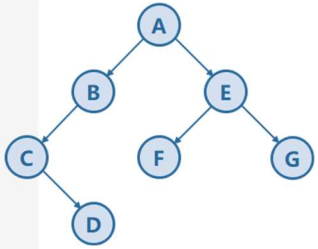
```
struct Node {
    public Node(var value: Rune,
        let left!: ?Node = None,
        let right!: ?Node = None) {}

    public func traverse(): Unit {
        left?.traverse()
        print(value)
        right?.traverse()
    }

    static let name : String
    static init() {
        name = "Binary Tree"
    }

    public static func intro() {
        println(name)
    }
}
```

```
var value: Rune
let left: ?Node
let right: ?Node

public init(value: Rune,
    left!: ?Node = None, right!: ?Node= None) {
    this.value = value
    this.left = left
    this.right = right
}
```

普通构造函数

```
match(right) {
    case Some(value) => value.traverse()
    case None => ()
}
```

```
main() {
    Node.intro()
    var root = Node('A',
        left: Node('B', left: Node('C', right: Node('D'))),
        right: Node('E', left: Node('F'), right: Node('G')))
    root.traverse()
}
```




```
> cjc example.cj -o example
> ./example
Binary Tree
CDBAFEG
```

五、类

定义与实例化

```
class name {
    constructor* | 构造函数
    (decl_var | decl_func | decl_prop)*
}
```

类名

成员变量 成员函数 成员属性

```
class Point {
    Point(let x: Float64, let y: Float64) {
        println("Create a point: (${x}, ${y})")
    }
}

main() {
    let p = Point(3.0, 4.0)
    println("Visit the point: (${p.x}, ${p.y})")
}
```

主构造函数

创建 Point 实例

访问实例成员

```
> cjc example.cj -o example
> ./example
Create a point: (3.000000, 4.000000)
Visit the point: (3.000000, 4.000000)
```

在类中可以定义多个构造函数，它们用于创建类实例（对象）

普通构造函数

```
init(params) {
    block_func
}
```

主构造函数

```
name_class(decl_vars) {
    block_func
}
```

$decl_{vars}$是对成员变量的声明，在此统一了成员变量的定义和初始化，减少冗余编码。

创建类实例

$name_{class}(args)$

成员访问规则

在成员变量、成员函数和成员属性的声明前可以添加一些**修饰符**

private	设置成员仅在类中可见
protected	设置成员在此类及其子类中可见
public	设置成员在类的内外均可见
static	设置成员为**静态成员**，只能通过类名访问 默认为**实例成员**，只能由类实例访问

在实例成员函数中可以引用其他成员，在静态成员函数中只能引用静态成员。

在实例成员函数中可以使用 **this** 变量，它是当前实例的引用，因此可以直接在实例成员函数中修改可变的实例成员变量。

```
> cjc example.cj -o example
> ./example
Point
Create a point: (3.000000, 4.000000)
Visit the point: (1.000000, 2.000000)
Visit the point: (1.000000, 2.000000)
```

```
class Point {
    static let name = "Point"
    public Point(private var x: Float64,
        private var y: Float64) {
        println("Create a point: (${x}, ${y})")
    }

    public func ref() {
        return this
    }

    public func set(x: Float64, y: Float64) {
        this.x = x
        this.y = y
    }

    public func show() {
        println("Visit the point: (${x}, ${y})")
    }
}

main() {
    println(Point.name)
    let p1 = Point(3.0, 4.0)
    var p2 = p1.ref()
    p2.set(1.0, 2.0)
    p1.show()
    p2.show()
}
```

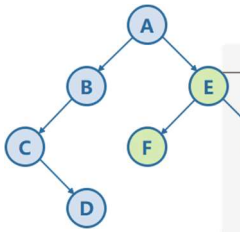
继承

open 修饰的类可以被其他类**继承**，如果类 B 继承了类 A，则类 B 会拥有类 A 的所有成员（但只能访问**非 private** 成员），实现**代码复用**。

```
open class A { ... }
class B <: A { ... }
```

类 B 继承类 A，称 B 为 A 的**子类**，A 为 B 的**父类**

在子类中可以改写继承来的实例成员函数/属性（需要被 **open** 修饰），称为**覆盖**，即便将子类实例转为父类型使用，在调用成员函数时也会优先选择覆盖版本，以此实现一种**多态机制**。




```
main() {
    var root = NodeA('A',
        left: NodeA('B', left: NodeA('C', right: NodeA('D'))),
        right: NodeB('E', left: NodeB('F'), right: NodeB('G')))
    root.traverse()
}
```

基于类的多态机制，这棵子树会按前序遍历

```
> cjc example.cj -o example
> ./example
CDBAEFG
```

在“大同”中实现“小异”，求同存异

```
open class NodeA {
    public NodeA(protected var value: Rune,
        protected let left!: ?NodeA = None,
        protected let right!: ?NodeA = None) {}

    public open func traverse(): Unit {
        left?.traverse()
        print(value)
        right?.traverse()
    }
}

class NodeB <: NodeA {
    public init(value: Rune,
        left!: ?NodeA = None, right!: ?NodeA = None) {
        super(value, left: left, right: right)
    }

    public func traverse(): Unit {
        print(value)
        left?.traverse()
        right?.traverse()
    }
}
```

实现中序遍历

不引入新成员，仅构造父类实例

super 表示父类构造函数

覆盖 NodeA 中的同名函数，改为前序遍历

属性

属性是一种特殊的成员，在使用时类似于成员变量，但它通过 **get** 和 **set** 函数来间接取值和赋值，由此可以实现访问控制、数据监控、跟踪调试、数据绑定等功能。

```
prop name: type {
    get() {
        block_func
    }
    set(name) {
        block_func
    }
}
```

属性名　属性类型

get　属性被读取/求值时将调用的函数，此函数需要返回 type 类型的值。

set　属性被赋值时将调用的函数，它的唯一参数就是被赋的值（类型为 type）。只有 **mut** 修饰的属性才能定义 **set** 函数。

```
class Node {
    private var value: Int64 = 0
    public Node(private var name: Rune,
        private let left!: ?Node = None,
        private let right!: ?Node = None) {}

    public mut prop param: Int64 {
        set(number) {
            value = number
            update()
            left?.param = number / 2
            right?.param = number / 2
        }
        get() { value }
    }

    private func update() {
        println("${name} has been updated to ${value}")
    }
}

main() {
    var root = Node('A',
        left: Node('B', left: Node('C', right: Node('D'))),
        right: Node('E', left: Node('F'), right: Node('G')))
    println(root.param)
    root.param = 128
}
```

与属性关联的成员变量

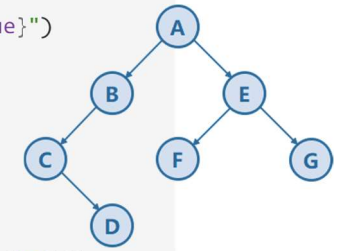
mut 属性中才能定义 set

set 函数也支持递归调用，这里按前序遍历同步更新子树各节点

读取属性

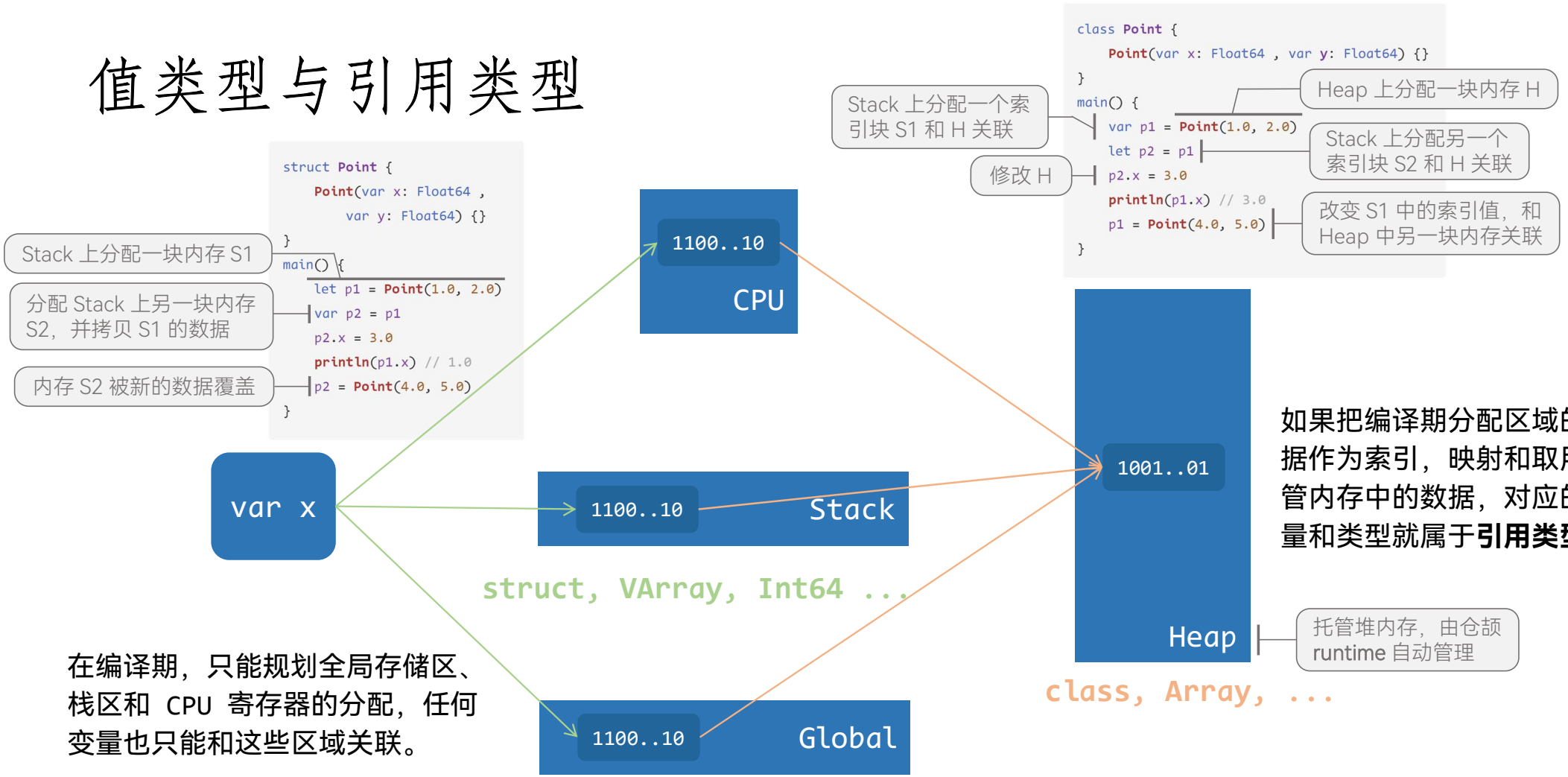
修改属性

```
> cjc example.cj -o example
> ./example
0
A has been updated to 128
B has been updated to 64
C has been updated to 32
D has been updated to 16
E has been updated to 64
F has been updated to 32
G has been updated to 32
```



本例可拓展应用于控制树更新场景，例如 UI 组件树的刷新等。

值类型与引用类型



如果把编译期分配区域的数据作为索引，映射和取用托管内存中的数据，对应的变量和类型就属于**引用类型**。

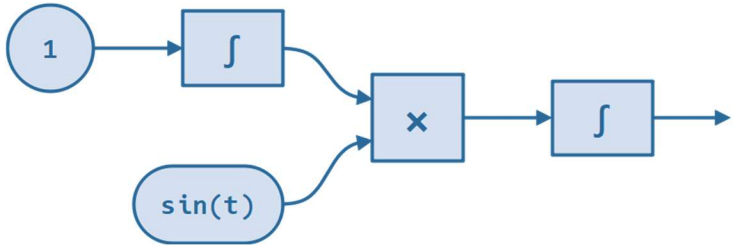
在编译期，只能规划全局存储区、栈区和 CPU 寄存器的分配，任何变量也只能和这些区域关联。

如果对一个变量的访问只涉及编译期分配区域的数据读/写，对应的变量和类型就属于**值类型**。可变/不可变性也是针对这部分数据而言的。

六、接口与扩展

接口

接口用来定义一个抽象类型，它不具有成员变量，仅约定一组功能对应的成员函数或属性原型。其他类型可以实现接口中的成员函数或属性，并成为该接口的子类型。



定义接口

```
interface name {
    (decl_func | decl_prop)*
}
```

成员属性

成员函数

实现接口

```
enum name <: interface { ... }
struct name <: interface { ... }
class name <: interface { ... }
extend type <: interface { ... }
```

表示接口名

实现接口中声明的成员

定义类型时实现接口

为已定义的类型扩展和实现接口

接口名

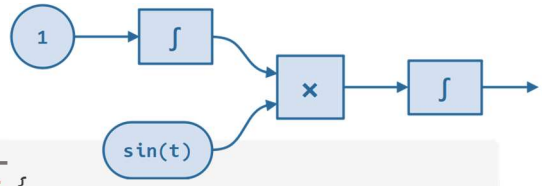
```
interface Slot {
    func compute(t: Float64): Float64

    operator func <<(that: Slot): Slot {
        return this
    }

    operator func >>(that: Slot): Slot {
        that << this
        return that
    }
}
```

声明成员函数原型

表示实现此接口的类型实例

成员函数可以有默认实现

实现接口 信号系统仿真



为内建类型实现接口，定义常值信号源

```
extend Float64 <: Slot {
    public func compute(t: Float64): Float64 {
        return this
    }
}
```

为 class 实现接口，定义正弦波信号源

```
class Wave <: Slot {
    public Wave(let freq: Float64, let phi: Float64) {}
    public func compute(t: Float64): Float64 {
        return sin(2.0 * Float64.PI * freq * t + phi)
    }
}
```

定义乘法器

```
class Mul <: Slot {
    public Mul(let a: Slot, let b: Slot) {}
    public func compute(t: Float64): Float64 {
        a.compute(t) * b.compute(t)
    }
}
```

定义积分器

```
class Integrator <: Slot {
    var input: ?Slot = None
    var sum = 0.0
    public Integrator(let dt: Float64) {}
    public func compute(t: Float64): Float64 {
        sum += dt * input.getOrThrow().compute(t)
        return sum
    }
    public operator func <<(that: Slot): Slot {
        input = Some(that)
        this
    }
}
```

覆盖接口中有默认实现的成员函数

```
main() {
    const DT = 0.001
    let left = 1.0 >> Integrator(DT)
    let right = Wave(0.5 / Float64.PI, 0.0)
    let flow = Mul(left, right) >> Integrator(DT)
    for (t in 0..1000) {
        println(flow.compute(Float64(t) * DT))
    }
}
```

计算过程也体现了基于接口的多态机制

```
> cjc example.cj -o example
> ./example
0.000000
......
0.297017
0.297852
0.298689
0.299527
0.300366
0.301207
```

注： t*sin(t) 的原函数为 -t*cos(t) + sin(t) + C，可以借此验证以上数值计算的准确性。

扩展

除了接口扩展，仓颉还支持为一个类型直接扩展成员函数或属性，并且不引入新的子类型关系。当我们不想破坏原有类型的封装性，但需要添加额外的功能时，就可以使用这种扩展能力。

默认情况下，扩展仅在它被定义的包中有效，如果需要导入/导出扩展，相关规则请参看仓颉语言文档。

```
extend String {
    operator func >>(n: Int64): String {
        if (n <= 0) {
            return this.clone()
        }
        let size = this.size
        let offset = size - n % size
        this[offset..size] + this[0..offset]
    }
}

main() {
    let text = "Cangjie2024"
    println(text >> 2 >> 2)
}
```

为字符串扩展实现“循环右移”操作

```
> cjc example.cj -o example
> ./example
2024Cangjie
```

泛型

泛型即参数化类型，通过给自定义类型增加类型参数，可定义类型构造器。

在使用处通过给定不同的类型实参，即可构造出各种具体类型。

仓颉中可以泛型化的类型有**函数，结构体，类，枚举，接口**。

```
class List<T> where T <: P { ... }
```

类型参数：T

对一系列类型均适用的代码，实现代码复用

泛型约束：T <: P

```
var data: List<Int64>
```

给定类型实参，构造出具体的 class 类型

这一约束保证 T 类型实例可打印

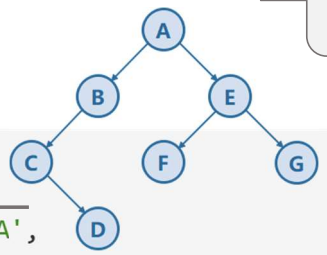
使用类型参数

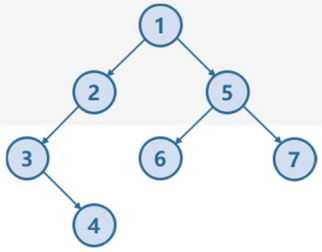
```
struct Node<T> where T <: ToString {
    public Node(var value: T,
        let left!: ?Node<T> = None,
        let right!: ?Node<T> = None) {}

    public func traverse(): Unit {
        left?.traverse()
        print(value)
        right?.traverse()
    }
}
```

Node<Rune> 类型

Node<Int64> 类型

```
main() {
    var tree1 = Node('A',
        left: Node('B', left: Node('C', right: Node('D'))),
        right: Node('E', left: Node('F'), right: Node('G')))
    tree1.traverse()
    println()

    var tree2 = Node(1,
        left: Node(2, left: Node(3, right: Node(4))),
        right: Node(5, left: Node(6), right: Node(7)))
    tree2.traverse()
}
```






```
> cjc example.cj -o example
> ./example
CDBAFEG
3421657
```

七、异常处理

异常类型

仓颉提供了 **Exception** 和 **Error** 两个类型，用于描述程序运行时的异常行为。

Exception

描述业务逻辑问题或 I/O 问题等导致的异常，例如协议解析失败或试图打开不存在的文件等，这类异常可以由开发者构造、抛出和处理。

```
public open class Exception <: ToString {
    public init()
    public init(message: String)
    public open prop message: String
    public open func toString(): String
    public func printStackTrace(): Unit
    public func getStackTrace(): Array<StackTraceElement>
}
```

构造异常实例并设置异常描述信息

获取异常描述信息

异常类型名 + 异常描述

打印异常堆栈

获取异常堆栈

Error

描述仓颉 **runtime** 内部故障或资源耗尽导致的异常，只能由 **runtime** 抛出，通常无法恢复，程序中捕获后应尽量安全地终止程序。

```
sealed class Error <: ToString {
    public open prop message: String
    public open func toString(): String
    public open func printStackTrace(): Unit
    public func getStackTrace(): Array<StackTraceElement>
}
```

开发者可以继承 **Exception** 或其子类来自定义异常类，但不能继承 **Error** 或其子类。

构造和抛出异常

构造异常即是构造异常类实例，在 **throw** 关键字后接一个异常类实例，即可抛出此异常。

定义异常

```
class ParseException <: Exception {
    public init() {
        super("Parse Failed")
    }
}
```

抛出异常

构造异常

抛出异常

构造异常

```
func parseInt(text: String): Int64 {
    if (text.isEmpty() || text == "-") {
        throw ParseException()
    }
    var sign = if (text[0] == 45u8) { 1 } else { 0 }
    var sum = 0
    for (i in sign..text.size) {
        if (text[i] > 57u8 || text[i] < 48u8) {
            throw ParseException()
        }
        let digit = Int64(text[i] - 48u8)
        sum = 10 * sum + digit
    }
    if (sign == 1) { -sum } else { sum }
}
```

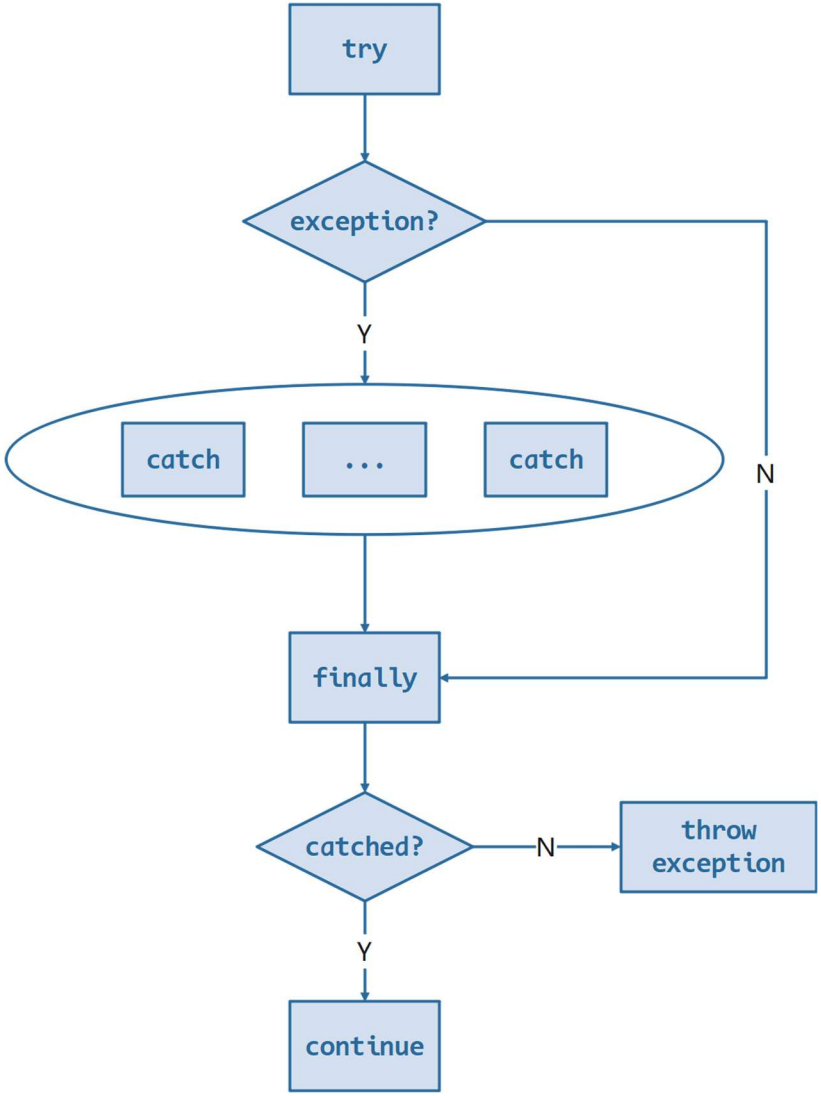
异常处理

```
try {
    block
} catch (pattern) {
    block
} finally {
    block
}
```

- `block`：可能抛出异常的业务代码
- `*`：异常处理代码，异常类型匹配后执行
- `pattern`：异常类型模式
- `?`：收尾代码，无论是否产生异常、异常是否被匹配处理，都会执行，一般用于资源回收

在 try 表达式中，至少要有一个 catch 分支或一个 finally 分支。

如果产生异常且被捕获处理，try 表达式的值由所执行的 catch 代码块决定，反之由 try 代码块决定。



异常处理流程

异常处理

```
main() {
    println(parseInt("-123456"))
    let number = try {
        parseInt("123-456")
    } catch (e: ParseException) {
        println("not an integer")
        0
    }
    println(number)

    try {
        parseInt("123x456")
        println(parseInt("-123456"))
    } catch (e: ParseException) {
        println(e.message)
    } finally {
        println("clean up")
    }
    parseInt("x123456")
    println("end")
}
```

类型为 Int64

不会被执行

总会被执行

异常未被处理，程序终止

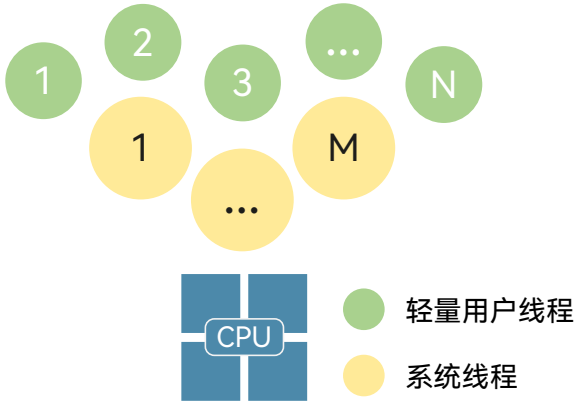
```
> cjc example.cj -o example
> ./example
-123456
not an integer
0
Parse Failed
clean up
An exception has occurred:
Exception: Parse Failed
         at default.ParseException::init()(test.cj:20)
         at default.String::toInteger()(test.cj:27)
         at default.main()(test.cj:62)
```

注：此程序引用了“构造和抛出异常”小节的示例代码

八、并发编程

线程模型

仓颉语言实现了 M:N 轻量线程模型，支持在少量系统线程之上创建海量用户线程，在实现层面用户线程对应协程，仓颉 **runtime** 会自动管理和调度这些协程。



仓颉 M:N 轻量线程模型

当用户线程 **t** 做 I/O 等资源访问操作时，若资源尚未就绪，线程 **t** 就会被 **runtime** 挂起等待、并调入其他线程运行，当资源就绪后又会适时恢复 **t** 的执行，高效利用 CPU 资源，实现高并发能力。

创建线程

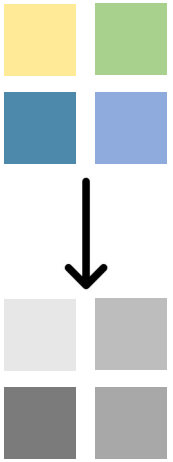
用 **spawn** 关键字修饰一个无参 **lambda**，就可以创建一个线程，并在线程中执行此函数。

```
spawn {
    block_func
}
```

spawn 表达式的类型为 **Future<T>**，**T** 是线程函数的返回值类型。

```
public class Future<T> {
    public func get(): T                          // 等待线程执行结束，获取线程返回值
    public func get(ns: Int64): Option<T>         // 等待线程 ns 纳秒，如果超时返回 None，反之得到线程返回值
    public func tryGet(): Option<T>               // 相当于get(0)
    public func cancel(): Unit                    // 向线程发送终止信号
    public prop thread: Thread                    // 获取线程对应的 Thread 类实例
}
```

```
let blocks = image.split(N)                       // 图像数据分成 N 块
let futures = ArrayList<Future<Unit>>()
for (block in blocks) {
    let future = spawn {                          // 创建线程，各块图像并行处理
        for (i in 0..block.size) {
            let (r, g, b) = block[i]              // 图像灰度化
            let gray = UInt8(0.299 * Float64(r) +
                0.587 * Float64(g) +
                0.114 * Float64(b))
            block[i] = (gray, gray, gray)
        }
    }
    futures.append(future)
}
for (future in futures) {
    future.get()                                  // 各线程在此与主线程同步
}
blocks.toImage().save("gray.png")                 // 将处理后的数据重组为图片并保存
```



应用实例 估算圆周率

```
from std import collection.*
from std import random.*
from std import math.*

const M = 200000
const N = 16
func task(): Int64 {
    var n: Int64 = 0
    let random = Random()
    for (_ in 0..M) {
        let x = random.nextFloat64()
        let y = random.nextFloat64()
        if ((x - 0.5) ** 2 + (y - 0.5) ** 2 < 0.25) {
            n++
        }
    }
    return n
}
```

向正方形内随机投点 M 次，统计落入内接圆中的次数

```
main() {
    let futures = ArrayList<Future<Int64>>()
    for (_ in 0..N) {
        let future = spawn { task() }
        futures.append(future)
    }
    var n = 0
    for (future in futures) {
        n += future.get()
    }
    let pi = Float64(n) / Float64(M * N) * 4.0
    println("π ≈ ${pi}")
    println("deviation: ${abs(Float64.PI - pi)}")
}
```

在多个线程中做投点实验

等待各线程计算结束，并获取计算结果

综合各线程统计数据，估算圆周率

```
> cjc example.cj -o example
> ./example
π ≈ 3.141509
deviation: 0.000084
```

九、跨语言互操作

跨语言互操作

按照使用方式或编程范式的差异，跨语言互操作可以分为两类：

接口式

```
auto runtime = XXLangEngine()
runtime.push(3.14)
runtime.call("process")
auto result = runtime.get(0).asBool()
```

宿主语言提供一个接口库，开发者调用其中的接口与目标语言进行交互。

声明式

```
extern {
    bool process(float x)
}
...
auto result = process(3.14)
```

用宿主语言的语法描述目标语言中的元素，在调用目标程序时，就像在调用宿主程序一样便捷。

仓颉语言支持和 C、ArkTS 等语言之间的声明式互操作。

仓颉 C 互操作基本步骤

```
// test.c
double process(float x, float y) { ... }
```

```
> clang test.c -shared -fPIC -o libtest.so
```

1、用仓颉函数相关语法和特定修饰符声明 C 函数原型

```
foreign func process(x: Float32, y: Float32): Float64
```

2、在互操作场景中，像调用普通仓颉函数一样调用已声明的 C 函数

```
let result = unsafe { process(3.14, 2.71) }
```

需要在 unsafe 块中调用 C 函数

3、在编译时指定依赖的 C 库

```
> cjc example.cj -L. -ltest -o example
```

类型映射 基础类型

在声明 C 函数时，核心在于仓颉如何描述 C 数据类型，因此我们需要知道 C 与仓颉的类型映射关系。

C	仓颉	Size (byte)
`void`	`Unit`	0
`char`	`UInt8`	1
`int8_t`	`Int8`	1
`uint8_t`	`UInt8`	1
`int16_t`	`Int16`	2
`uint16_t`	`UInt16`	2
`int32_t`	`Int32`	4
`uint32_t`	`UInt32`	4
`int64_t`	`Int64`	8
`uint64_t`	`UInt64`	8
`ssize_t`	`IntNative`	platform dependent
`size_t`	`UIntNative`	platform dependent
`float`	`Float32`	4
`double`	`Float64`	8

类型映射 其他类型

C	仓颉
struct	@C struct
char[]	CString
type*	CPointer<type>

结构体成员由对应的类型映射关系进行声明

```
extend CPointer<T> {
    public func isNull(): Bool
    public func isNotNull(): Bool
    public func toUIntNative(): UIntNative
    public unsafe func read(): T
    public unsafe func read(idx: Int64): T
    public unsafe func write(value: T): Unit
    public unsafe func write(idx: Int64, value: T): Unit
    public unsafe operator func +(offset: Int64): CPointer<T>
    public unsafe operator func -(offset: Int64): CPointer<T>
    public func asResource(): CPointerResource<T>
}
```

```
extend CString <: ToString {
    public func getChars(): CPointer<UInt8>
    public func isNull(): Bool
    public func size(): Int64
    public func isEmpty(): Bool
    public func isNotEmpty(): Bool
    public func startsWith(prefix: CString): Bool
    public func endsWith(suffix: CString): Bool
    public func equals(rhs: CString): Bool
    public func equalsLower(rhs: CString): Bool
    public func subCString(beginIndex: UIntNative): CString
    public func subCString(beginIndex: UIntNative,
        subLen: UIntNative): CString
    public func compare(str: CString): Int32
    public func toString(): String
    public func asResource(): CStringResource
}
```

在标准库中为 **CString** 和 **CPointer** 扩展了一些成员函数，便于操作 C 字符串和指针。

应用实例

```c
// ffi.c
#include <math.h>

typedef struct {
    double x;
    double y;
} Point;

Point rotate(Point input, double a) {
    Point output;
    output.x = input.x * cos(a) - input.y * sin(a);
    output.y = input.x * sin(a) + input.y * cos(a);
    return output;
};
```

```
> clang ffi.c -shared -fPIC -o libffi.so
> ls
ffi.c libffi.so ...
```

```
// example.cj
@C
struct Point {
    Point(var x: Float64, var y: Float64) {}
}

foreign func rotate(point: Point, alpha: Float64): Point

unsafe main() {
    let input = Point(1.2, 3.4)
    let output = rotate(input, 1.4)
    println("${output.x}, ${output.y}")
}
```

创建 @C struct 实例，它可以传递给 C 程序

调用 C 函数，并获取返回的结构体实例

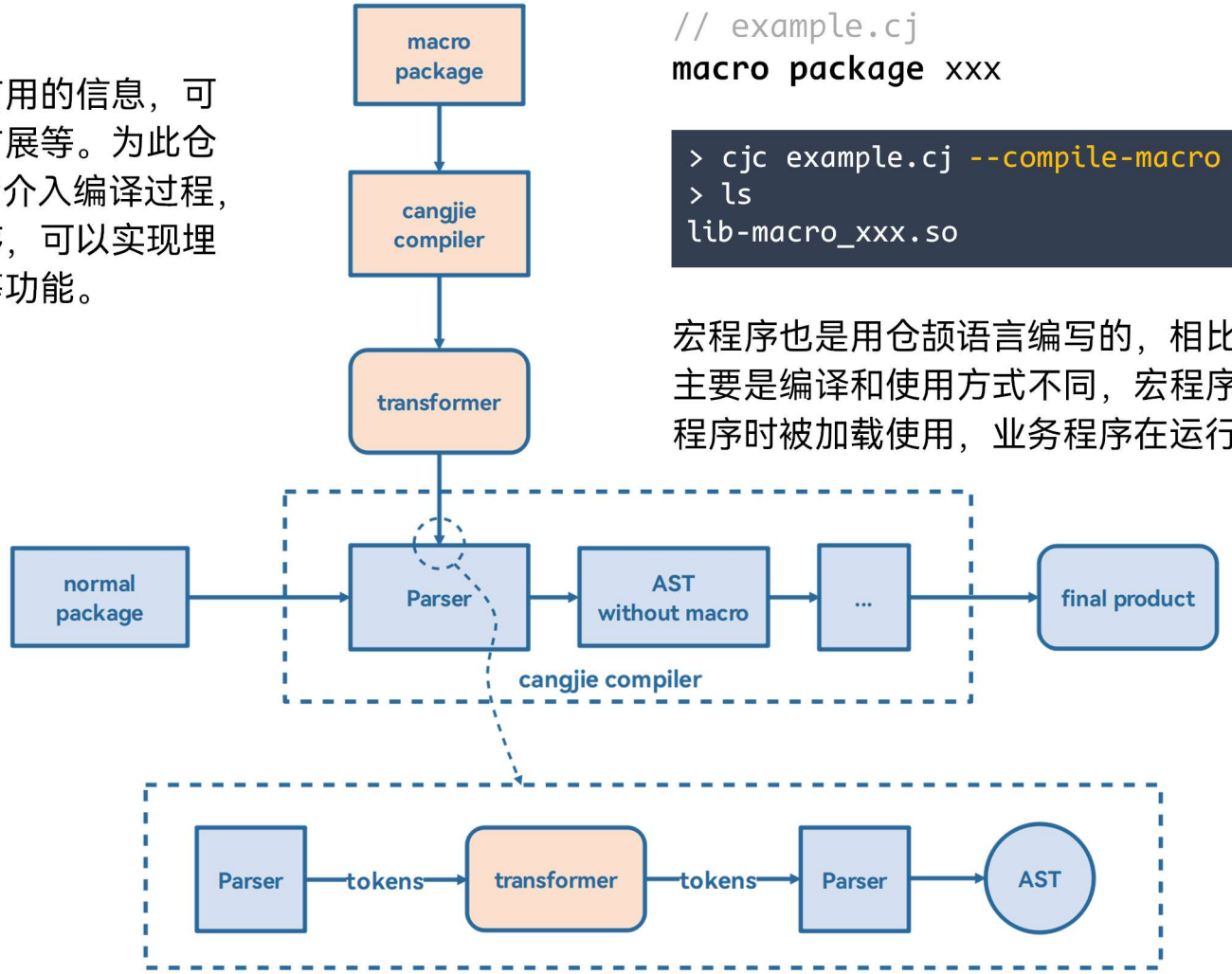
访问 C 结构体成员

```
> cjc example.cj -L. -lffi -o example
> ./example
-3.146569, 1.760428
```

十、宏

概述

在程序编译阶段，会产生很多有用的信息，可用于程序的分析、优化和功能扩展等。为此仓颉提供了“宏”特性，允许开发者介入编译过程，获取部分编译期信息并修改程序，可以实现埋点插桩、静态反射和语法扩展等功能。



宏的作用机制

```
// example.cj
macro package xxx
```

```
> cjc example.cj --compile-macro
> ls
lib-macro_xxx.so
```

宏程序也是用仓颉语言编写的，相比普通仓颉程序，主要是编译和使用方式不同，宏程序只在编译其他程序时被加载使用，业务程序在运行时不会涉及宏。

定义与调用

宏名

调用处代码对应的单词集合，由编译器解析传入

```
macro name(name: Tokens): Tokens {
    (expr | decl_var)*
}
```

宏返回的单词集合，将替换调用处的代码

```
// macro.cj
macro package meta
from std import ast.*

public macro transform(tokens: Tokens): Tokens {
    for (token in tokens) {
        // 也可以调用 token.dump() 查看更多信息
        println("${token.value}\t\t${token.kind}")
    }
    println("--------------------")
    return tokens
}
```

宏需要定义在宏包内

可以获取各单词的词法信息

本例没有对输入代码作修改，原样返回

```
// example.cj
import meta.*

@transform
func add(x: Int64, y: Int64) {
    return x + y
}

main() {
    @transform(add(1, 2))
}
```

导入宏包

在函数定义处调用宏

在表达式上调用宏

```
> cjc macro.cj --compile-macro
> cjc example.cj -o example
func                FUNC
add                 IDENTIFIER
(                   LPAREN
x                   IDENTIFIER
:                   COLON
Int64               INT64
,                   COMMA
y                   IDENTIFIER
:                   COLON
Int64               INT64
)                   RPAREN
{                   LCURL

                    NL
return              RETURN
x                   IDENTIFIER
+                   ADD
y                   IDENTIFIER

                    NL
}                   RCURL
--------------------
add                 IDENTIFIER
(                   LPAREN
1                   INTEGER_LITERAL
,                   COMMA
2                   INTEGER_LITERAL
)                   RPAREN
--------------------
```

注意这些信息是在编译时输出的

Tokens 及 **Token** 类型的详细用法，请参看仓颉标准库文档。此外，本节只介绍了**非属性宏**，仓颉还支持**属性宏**，宏的详细内容请参看仓颉语言文档。

在编译时修改程序

```cj
// macro.cj
macro package meta
from std import ast.*

public macro transform(tokens: Tokens): Tokens {
    let output = Tokens()
    for (token in tokens) {
        if (token.kind == TokenKind.ADD) {
            output.append(Token(TokenKind.SUB))
        } else {
            output.append(token)
        }
    }
    return output
}
```

将原程序中的加号变成减号

其他代码保持不变

```cj
// example.cj
import meta.*

@transform
func f(x: Int64, y: Int64) {
    return x + y
}

main() {
    println(f(1, 2))
}
```

编译时调用宏，x + y 将被替换为 x - y

```
> cjc macro.cj --compile-macro
> cjc example.cj -o example
> ./example
-1
```

可以使用如下命令输出被宏修改后的源代码：

```
> cjc --debug-macro example.cj
```

这会生成名为 example.cj.macrocall 的文件，其中的内容是：

```cj
import meta.*

/* ===== Emitted by MacroCall @transform in example.cj:3:1 ===== */
/* 3.1 */func f(x: Int64, y: Int64) {
/* 3.2 */    return x - y
/* 3.3 */}
/* ===== End of the Emit ===== */

main() {
    println(f(1, 2))
}
```

应用实例 语言扩展

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

C# 的语言集成查询（LINQ）特性

example.cj

```
import linq.*
from std import collection.*

main() {
    let scores = [97, 92, 81, 60]
    @LINQ(from score in scores where score > 80 select score)
}
```

基于宏为仓颉实现简易 LINQ

```
> cjc macro.cj --compile-macro
> cjc example.cj -o example
> ./example
97
92
81
```

```
for (score in scores) {
    if (score > 80) {
        println(score)
    }
}
```

macro.cj

```
macro package linq
from std import ast.*

public macro LINQ(tokens: Tokens): Tokens {
    let attribute = tokens[1]
    let table = tokens[3]

    var index = 5
    let condition = Tokens()
    for (i in index..tokens.size) {
        if (tokens[i].value == "select") {
            index = i + 1
            break
        }
        condition.append(tokens[i])
    }
    let select = Tokens()
    for (i in index..tokens.size) {
        select.append(tokens[i])
    }

    return quote(
        for ($attribute in $table) {
            if ($condition) {
                println($select)
            }
        }
    )
}
```

在实际应用中，应该对扩展语法做严格解析和异常处理，
这里限于篇幅和演示目的，只做了简单解析和处理。


仓颉
Cangjie