2022/01

函数式编程笔记

- 编程语言是**工具**（简化程序复杂度）模块化问题
    - 软件开发的工具，团队协作软件开发的工具
    - 使用工具到顶了，功能复杂度hold人脑不住了，就要换新工具（人脑几十年的时间不会有什么进化）
    - 应用软件开发：通过历史看未来（从不同角度看历史变迁），结构化编程(模块化) -> 面向对象 -> 函数式编程；可以看到这个路线就是**对“状态”的限制**使用之路
- 函数式编程主要解决的问题
    - 代码复杂度（无代码）
    - 线程管理、迭代（责任渡让）
    - 分布式计算（RPC（Remote Procedure Call）远程过程调用）

- **发明机器**与**人编程**两件事
- **莱布尼兹**（与**牛顿**）曾经有两个梦想：
    - 创建一种“普遍语言”（universal language）使得任何问题都可以用这种语言表述；
        - 对前一个问题的回答就是自弗雷格、罗素开始，经公理集合论运动的最终结果：以一阶谓词逻辑为语言所形式化阐述的集合论，现在已经成为数学的普遍语言，现代逻辑学、特别是将符号逻辑应用于数学领域所产生的数理逻辑，其最重要的目标就是为整个数学提供一个严格精确的语言。这是我们在学习**数理逻辑**时应当把握的方向。
    - 找到一种"判定方法"（decision method）以解决所有可以在“普遍语言”中所表述的问题
        - 第二个问题则是现代哲学和**计算机科学**最关注的问题
        - 1928年提出的一个重要的数学问题：“**判定性问题**”（decision problem），所有计算问题都可以规约到相应的一个判定性问题。“任意正整数是不是素数”这个问题就是可判定的。不可判定问题：“停机问题”、“理发师悖论”-理发师不帮自己理发的人理发，那他该不该帮自己理发？
        - 可计算理论的研究对象有三个： **(1) 判定问题；(2) 可计算性；(3) 计算复杂性（P=？NP）**。
        - 1936年，阿隆佐邱奇**Alonzo Church** 和 阿兰图灵**Alan Turing** 证明了：为“判定性问题”设计一个通用算法这件事是不可能的。
        - 与此同时，对于“可判定”，他们各自详细阐述了两个计算模型：Alonzo Church的**Lambda演算（λ演算）**，Alan Turing的理论机器（之后被称作**图灵机**）
    -

Church vs. Turing

Alonzo Church
Born: 14 June 1903 in Washington, D.C., USA.
Died: 11 Aug 1995 in Hudson, Ohio, USA.
Ph.D.: Princeton University, 1927, USA

Alan Turing
Born: 23 June 1912, Maida Vale, London, UK.
Died: 7 June 1954, Wilmslow, Cheshire, UK.
Ph.D.: Princeton University, 1938, USA.

Alonzo Church, *"An Unsolvable Problem in Elementary Number Theory,"* American J. of Mathematics, vol. 5 (1936), pp. 345-363.

Alonzo Church, *"A Note on the Entscheidungsproblem,"* J. of Symbolic Logic, vol. 1 (1936) pp. 40-41. Correction: *ibid*, pp. 101-102.

Alan Turing, *"On Computable Numbers with an Application to the Entscheidungsproblem,"* Proc. of the London Math. Soc., vol. 42 (1936), pp. 230-267. Correction: vol. 43 (1937), pp. 544-546.

Alan Turing, *"Computability and λ -definability,"* J. Symbolic Logic, vol. 2 (1937), pp. 153-163.

The work of Church and Turing in 1936 was done independently.

4

- 图灵机
  - 一切可计算的问题都能计算，这样的处理器或者编程语言就叫图灵完备的。
  - Brainfuck （是一种极小化的计算机语言（图灵完备）， 它是由Urban Müller在1993年创建。
  - Brainfuck 在线编译 https://copy.sh/brainfuck/

```
++++++++++[>+>+++>+++++++>++++++++++<<<<-]>>>++.>+.+++++++..+++.
<<++.>+++++++++++++++.>.+++.------.--------.<<+.<.
```

字符	含义
>	指针加一
<	指针减一
+	指针指向的字节的值加一
-	指针指向的字节的值减一
.	输出指针指向的单元内容（ASCII码）
,	输入内容到指针指向的单元（ASCII码）
[	如果指针指向的单元值为零，向后跳转到对应的]指令的次一指令处
]	如果指针指向的单元值不为零，向前跳转到对应的[指令的次一指令处

- Lambda演算（λ）
  - 整个系统是基于表达式的（也叫λ项）：
    - **变量（Variable）**：形式：**x** 变量名可能是一个字符或字符串，它**表示一个参数**（形参）或者**一个值**（实参）
    - **抽象体（Abstraction）**： 形式：**λx.M** 它表示获取一个参数x并返回M的lambda函数，M是一个合法lambda表达式，且符号λ和.表示绑定变量x于该函数抽象的函数体M。简单来说就是**表示一个形参为x的函数M**
    - **应用（Application）** 形式：**M N** 它表示将函数M应用于参数N，其中M、N均为合法lambda表达式。简单来说就是**给函数M输入实参N**
  - Alpha「转换」（α 转换）：一个lambda函数抽象在更名绑定变量前后是等价的。**λx.x ≡ λy.y**
  - Beta「规约」（β-Reduction）：是**把标识符用参数值替换**；执行任何可以由机器来完成的计算。

```
(lambda x . x + 1) 3 == 3 + 1

(lambda y . (lambda x . x + y)) q == lambda x . x + q

(lambda x y. x y) (lambda z . z * z) 3 == (lambda z . z * z) 3 == 3 * 3

//只有在不引起绑定标识符和自由标识符之间的任何冲突的情况下，才可以做Beta规约
lambda z . (lambda x . x + z)
(lambda z . (lambda x . x + z)) (x + 2) == (lambda z . (lambda y . y + z)) (x + 2)
== (lambda y . y + x + 2) 3 == 3 + x + 2
```

  - 数字：**丘奇数**都是带有两个参数的函数
    - 0是“`lambda s z . z`“
    - 1是“`lambda s z . s z` “
    - 2是“`lambda s z . s (s z)`

- **x + y**

  ```
  let add = lambda s z x y . x s (y s z) == lambda x y . (lambda s z . (x s (y s z)))
  ```

- 形如 `if / then / else`语句的表达式

  ```
  let TRUE = lambda x y . x
  let FALSE = lambda x y . y

  let IfThenElse = lambda cond true_expr false_expr . cond true_expr false_expr

  let BoolAnd = lambda x y . x y FALSE
  let BoolOr = lambda x y. x TRUE y
  let BoolNot = lambda x . x FALSE TRUE
  ```

- 图灵机引出了命令式编程
  - FORTRAN（1956 IBM）
  - C（1972 Dennis Ritchie 丹尼斯·里奇）C90 C99
  - C++（1983 贝尔实验室）
  - java （1995 Sun）
  - PHP （1995）
  - C# （2003）
- λ演算引出了函数式编程
  - Lisp（1958） 第二个高级编程语言的，它就是启发自λ演算。
  - Haskell（1985年，Miranda发行，Haskell语言是1990年在编程语言Miranda的基础上标准化的）
  - Python（1991 1994 Python1.0版本发布，这个版本的主要新功能是**lambda, map, filter和reduce**）
  - Scala（2003 更接近Java习惯，集成了面向对象和函数式编程范式）
  - Clojure（2007 Clojure采用Lisp语法）
  - Groovy（2003 动态类型化，闭包，像java风格的脚本语言，接近Ruby或Python）
  - Kotlin（ 2011 Java 虚拟机，作为 Android 官方支持的第二种编程语言）
  - golang（2009 谷歌）
  - F#（2005）

- 函数式编程的语言分类：
  - 只支持函数式范式（如Haskell）
  - 多范式+一流支持（如Scala）
  - 多范式+部分支持（如Javascript、Go）。这类语言中，函数式编程一般通过库来支持，这些库复制前两种语言的标准库中的部分或全部功能。
- 函数式编程的要求：
  - 一等函数/高阶函数
  - 闭包

- **泛型**（go1.18）
- 尾递归优化
- 可变个数参数的函数+可变类型参数
- “柯里化”（Currying）(复用)

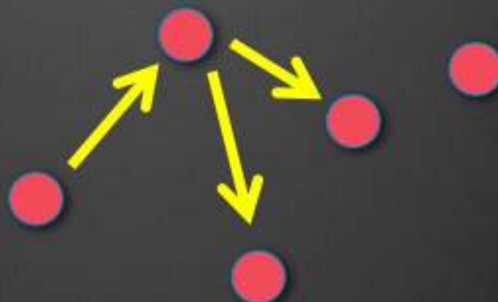
- 范畴论简介
  - Haskell 语言使用了范畴论的概念
  - 范畴论对编程有理论意义
    - 当我们知道有些代数结构具有一些很好的性质，满足某些定律。那如果我们写的程序具有这些代数结构，则这些程序也自然就具有同样的性质，满足同样的定律，可以应用同样的规则。例如**幺半群** $(M, id, \oplus)$满足结合律，即 $a \oplus (b \oplus c) = (a \oplus b) \oplus c$，也就是和结合的次序无关。我们知道列表类型 [a] 是一个幺半群，当我们将列表[a1, a2, a3, a4]内的所有元素相加时，我们可以先计算b = a1 + a2, c = a3 + a4，然后再计算sum = b + c。这样在多核CPU的情况下，我们可以让CPU1计算b = a1 + a2，让CPU2计算c = a3 + a4，再让CPU1计算最后的结果sum = b + c。这样我们就实现了4个元素的并行求和运算，有n个元素的求和运算也可按这种方式并行化。若我们的另一个数据类型也同样具有幺半群的代数结构，则这个数据类型的⊕运算同样具有并行计算的能力。范畴论在这方面则起到了重要的理论指导作用，我们可以将范畴论中得到的范畴的代数性质和定律应用到函数式编程中，通过类型系统以新的方式来构造我们的程序。
    - 柯里化和反柯里化是一对**同构变换**，其变换得到的函数是唯一的。也可以将柯里化和反柯里化看成是普通函数和高阶函数之间的变换，柯里化具有升阶的作用，而反柯里化则具有降阶的作用。这样单参数函数和多参数函数在参数形式上**就没有区别了**，具有统一的形式，从而可以放在同一个列表中，也可以串在一起。
    - 范畴论是从拓扑学这一数学分支发展出来的一门抽象的数学理论，后来成为一个独立的分支，并成为了其他数学分支的基础。再后来，和**类型理论**结合，成为了**编程语言理论的数学基础**，因此我们才会在这里来介绍范畴论。但是，学习范畴论却是不需要学习拓扑学也是可以的，学习编程也是可以不学习范畴论。

- 范畴(category)；随便什么东西，只要能找出它们之间的关系，满足下面条件，就是一个"范畴"
  -

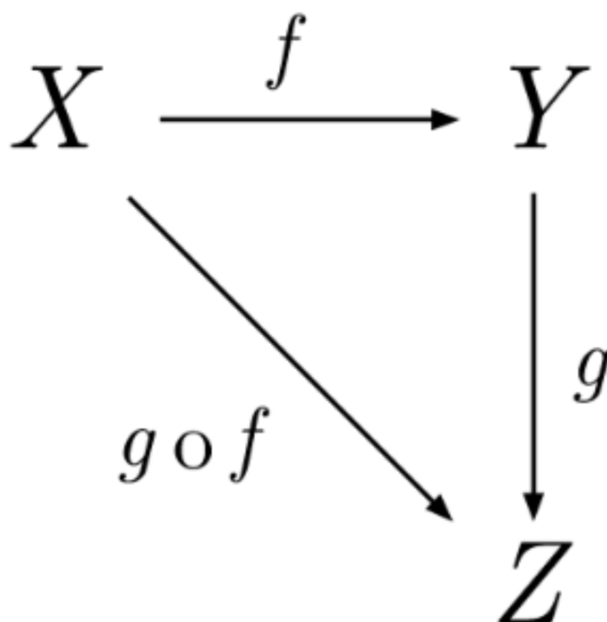
Category

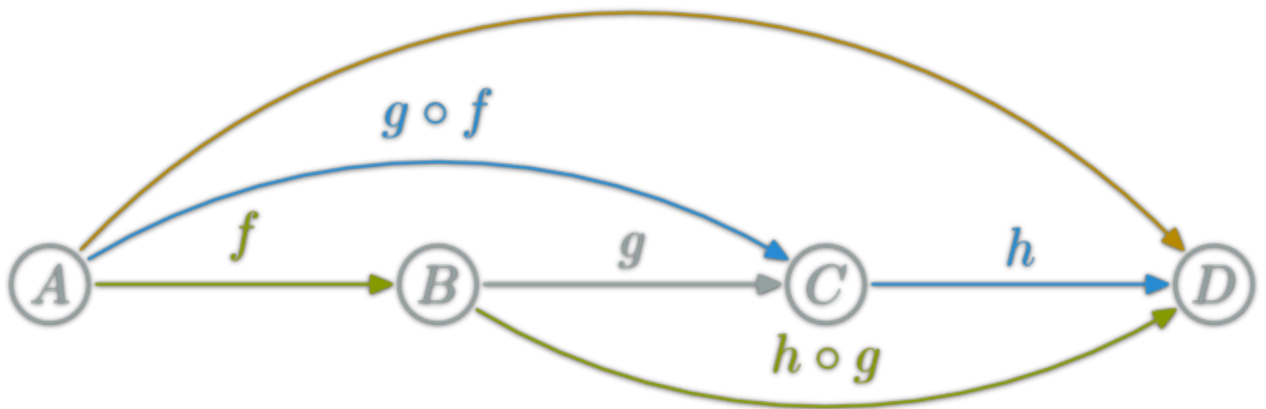
Objects

Arrows or *morphisms*



- 对象/点：object
- 态射/关系/箭头: morphism, f:a→b
- 满足结合律：f:a→b,g:b→c 的组合是 g◦f ； 结合律：(h◦g)◦f = h◦(g◦f)

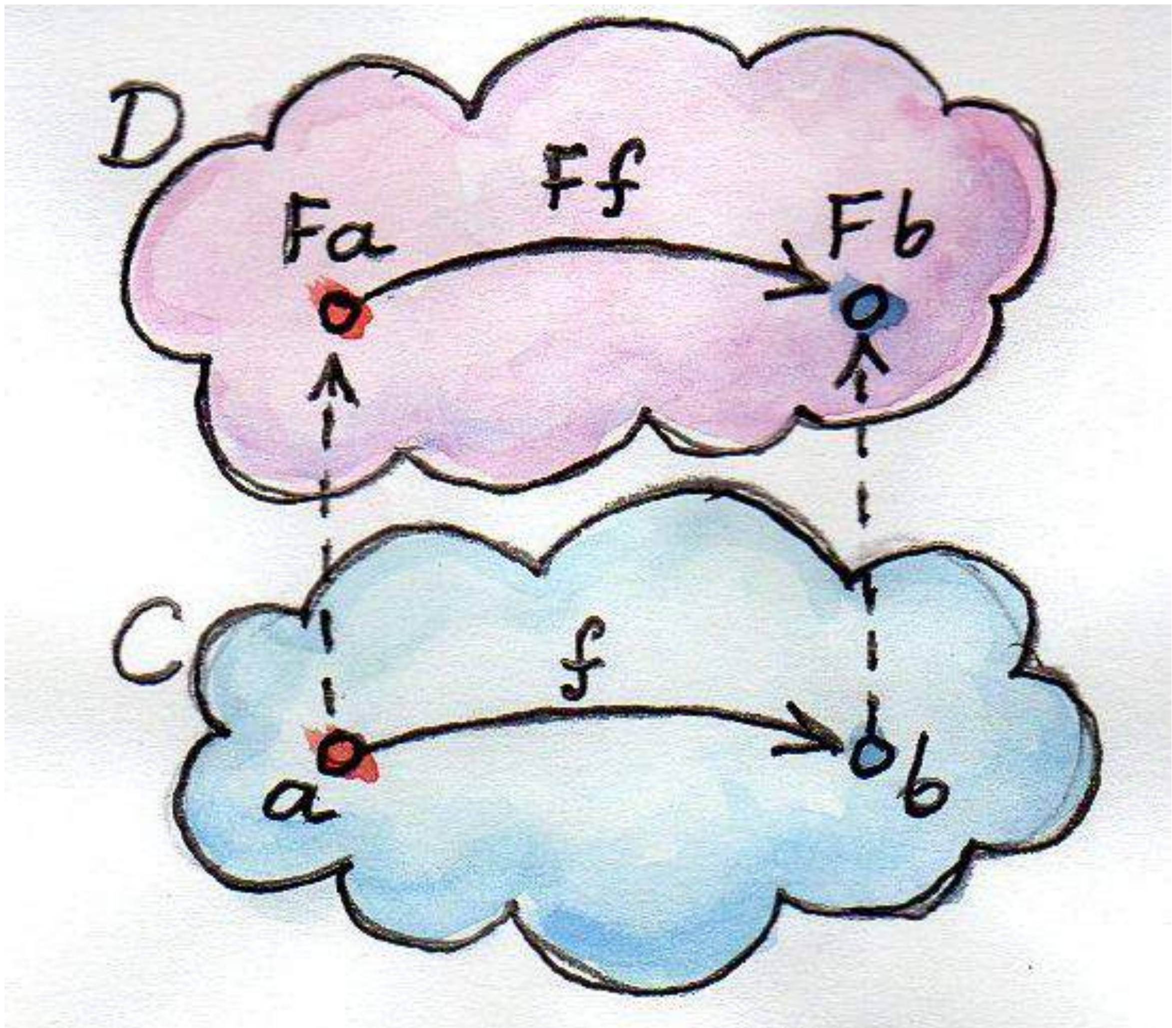




- 存在恒等态射(identity):
  - 对于任何态射f:a→b，有1b∘f=f=f∘1a ；可以简单的认为是；f(x)

- 态射的种类：
  - 单态射：不存在两个a中元素 map 到同一个b中的元素。
  - 满态射：每一个b中的元素，都在a中有至少一个 source mapper。
  - 双态射：即是单态射，又是满态射。
  - 同构：a,b两个对象的元素存在一对一的 map 关系。同构 = 双态射 + 存在逆态射。
  - 自态射：表示一个态射源对象和目标对象是同一个, f:a→a
  - 自同构：如果f既是一种自态射，又是具有同构性。
  - 撤回射：如果存在一个f的右逆，其含义是：g 可以通过f找到 source element。f必定是一个满态射
  - 部分射：如果f的左逆是存在的，也就是说，如果存在 g : b→a, g ∘ f = 1a。 f 是另一个态射g的部分射，其含义是：f 确定了g的同构部分。f必定是一个单态射。
  - 同态：是一个态射，表示一个**数学结构**A到另一个数学结构B的map关系，并且维持了数学结构上的的每一种操作*。 f:A→B，有： f(x∗y)=f(x)∗f(y)
    - 同一种操作在不同的数学结构上定义可以不同。
    - 比如：指数函数是一个同态。e(x+y)=exey→f(x∗y)=f(x)∗f(y)

- **函子(Functor)**
  - 函子是范畴之间的关系。可以**理解为范畴之间的态射**。
  - 图中，函数 `f` 完成object的转换（`a` 到 `b`），将它传入函子，就可以实现范畴的转换（`Fa` 到 `Fb`）。

- 

- 范畴之间的态射，得到函子，由函子之间的态射，得到自然变换
- 函子 F 将简单类型a 构造为复杂类型 F a，将简单类型的函数 f 变换为复杂类型的函数 F f；**类型构造子只有同时变换类型和函数，且满足函子定律时，才是一个函子。函子的用处**是将简单类型的函数通过类型构造子提升为复杂类型的函数，且不需要写额外的代码，使得一些基本函数可以应用到多种复杂数据类型的成员中，提高了这些基本函数的**重用性**。

- Maybe 函子
  - Either 函子(运算`if...else`)
  - ap函子，函子里面包含的值，完全可能是函数。我们可以想象这样一种情况，一个函子的值是数值，另一个函子的值是函数。
  - 协变函子 F:C→D
  - 对于每一个 C 中的对象x，对应一个 D 中的F(x)。
    - 对于每一个C中的态射f:x→y，对应D的态射F(f):F(x)→F(y)。
    - 逆变函子
  - 和协变函子类似，只不过态射的方向是从D到C。
    - 自然转换(Natural transformation)
      - 自然转换是两个函子之间的关系。函子描述“自然构造(natural constructions)”，自然转换描述两个这样构造的"自然同态(natural homomorphisms)"。
- 函数式编程实践
  - Map-Reduce-Filter 模式

```
function foldr(f,z,list){
  //为了简洁起见，把类型判断省略了
  // Object.prototype,toString.call(list) === '[object Array]'
  if(list === null || list.length == 0){
    return z;
  }
  return f(list[0],foldr(f,z,list.slice(1)));
}

function add(a,b){
  return a+b;
}

function sum(list){
  return foldr(add,0,list);
}

let a = [1,2,3];
sum(a); // 6

function muti(a,b){
  return a*b;
}

function product(list){
  return foldr(muti,1,list);
}
```

```
function fandcons(f, el, list){
  return [f(el)].concat(list);
}
//需要柯里化
var fandcons_ = _.curry(fandcons);

function map(f, list){
  return foldr(fandcons_(f),[],list);
}
//调用
console.log(map(function(x){return 2*x},[1,2,3,4]));
// 输出[ 2, 4, 6, 8 ]
```

-
- Continuation 延续

```
System.out.println("Please enter your name: ");
System.in.readLine();
```

```
System.out.println("Please enter your name: ", System.in.readLine);
```

-
- https://github.com/thoas/go-funk
  - funk.Contains 包含

```
// slice of string
funk.Contains([]string{"foo", "bar"}, "bar") // true

// string
funk.Contains("florent", "rent") // true
funk.Contains("florent", "foo") // false

// even map
funk.Contains(map[int]string{1: "Florent"}, 1) // true
funk.Contains(map[int]string{1: "Florent"}, func(key int, name string)
bool {
    return key == 1 // or `name == "Florent"` for the value type
}) // true
```

  - funk.Intersect 交集

```
funk.Intersect([]int{1, 2, 3, 4}, []int{2, 4, 6})  // []int{2, 4}
funk.Intersect([]string{"foo", "bar", "hello", "bar"}, []string{"foo",
"bar"})  // []string{"foo", "bar"}
```

  - funk.Difference 集合差

```
funk.Difference([]int{1, 2, 3, 4}, []int{2, 4, 6})  // []int{1, 3},
[]int{6}
funk.Difference([]string{"foo", "bar", "hello", "bar"}, []string{"foo",
"bar"})  // []string{"hello"}, []string{}
```

  - funk.IndexOf 首次出现值的索引，如果找不到值，则返回 -1

```
// slice of string
funk.IndexOf([]string{"foo", "bar"}, "bar") // 1
funk.IndexOf([]string{"foo", "bar"}, func(value string) bool {
    return value == "bar"
}) // 1
funk.IndexOf([]string{"foo", "bar"}, "gilles") // -1
```

  - funk.LastIndexOf 找到值的最后一次出现的索引，如果找不到该值，则返回 -1

```go
// slice of string
funk.LastIndexOf([]string{"foo", "bar", "bar"}, "bar") // 2
funk.LastIndexOf([]string{"foo", "bar"}, func(value string) bool {
    return value == "bar"
}) // 2
funk.LastIndexOf([]string{"foo", "bar"}, "gilles") // -1
```

- funk.ToMap 将切片或结构体数组转换为基于透视字段的Map

```go
f := &Foo{
    ID:         1,
    FirstName: "Gilles",
    LastName:  "Fabio",
    Age:       70,
}

b := &Foo{
    ID:         2,
    FirstName: "Florent",
    LastName:  "Messa",
    Age:       80,
}

results := []*Foo{f, b}

mapping := funk.ToMap(results, "ID") // map[int]*Foo{1: f, 2: b}
```

- funk.ToSet 将数组或切片转换为Set（零值的Map）

```go
f := Foo{
    ID:         1,
    FirstName: "Gilles",
    LastName:  "Fabio",
    Age:       70,
}

b := Foo{
    ID:         2,
    FirstName: "Florent",
    LastName:  "Messa",
    Age:       80,
}

mapping := funk.ToSet([]Foo{f, b}) // map[Foo]struct{}{f: struct{}{}, b: struct{}{}}

mapping := funk.ToSet([4]int{1, 1, 2, 2}) // map[int]struct{}{1: struct{}{}, 2: struct{}{}}
```

- funk.Filter 根据谓词筛选切片

```
r := funk.Filter([]int{1, 2, 3, 4}, func(x int) bool {
    return x%2 == 0
}) // []int{2, 4}
```

- funk.Reduce 根据累积函数或数字操作符迭代。

```
// Using operation runes. '+' and '*' only supported.
r := funk.Reduce([]int{1, 2, 3, 4}, '+', float64(0)) // 10
r := funk.Reduce([]int{1, 2, 3, 4}, '*', 1) // 24

// Using accumulator function
r := funk.Reduce([]int{1, 2, 3, 4}, func(acc float64, num int) float64 {
    return acc + float64(num)
}, float64(0)) // 10

r := funk.Reduce([]int{1, 2, 3, 4}, func(acc string, num int) string {
    return acc + fmt.Sprint(num)
}, "") // "1234"
```

- funk.Find 根据谓词在切片中查找元素

```
r := funk.Find([]int{1, 2, 3, 4}, func(x int) bool {
    return x%2 == 0
}) // 2
```

- funk.Map 操作（映射、切片）将其转换为另一种Set（映射、切片）

```
r := funk.Map([]int{1, 2, 3, 4}, func(x int) int {
    return x * 2
}) // []int{2, 4, 6, 8}

r := funk.Map([]int{1, 2, 3, 4}, func(x int) string {
    return "Hello"
}) // []string{"Hello", "Hello", "Hello", "Hello"}

r = funk.Map([]int{1, 2, 3, 4}, func(x int) (int, int) {
    return x, x
}) // map[int]int{1: 1, 2: 2, 3: 3, 4: 4}
```

- funk.FlatMap 操作（映射、切片）并将其转换为另一种类型的平展Set

```go
r := funk.FlatMap([][]int{{1, 2}, {3, 4}}, func(x []int) []int {
    return append(x, 0)
}) // []int{1, 2, 0, 3, 4, 0}

mapping := map[string][]int{
    "Florent": {1, 2},
    "Gilles": {3, 4},
}

r = funk.FlatMap(mapping, func(k string, v []int) []int {
    return v
}) // []int{1, 2, 3, 4}
```

- funk.Get 检索Struct或Map路径的值

```go
var bar *Bar = &Bar{
    Name: "Test",
    Bars: []*Bar{
        &Bar{
            Name: "Level1-1",
            Bar: &Bar{
                Name: "Level2-1",
            },
        },
        &Bar{
            Name: "Level1-2",
            Bar: &Bar{
                Name: "Level2-2",
            },
        },
    },
}

var foo *Foo = &Foo{
    ID:        1,
    FirstName: "Dark",
    LastName:  "Vador",
    Age:       30,
    Bar:       bar,
    Bars: []*Bar{
        bar,
        bar,
    },
}

funk.Get([]*Foo{foo}, "Bar.Bars.Bar.Name") // []string{"Level2-1",
"Level2-2"}
funk.Get(foo, "Bar.Bars.Bar.Name") // []string{"Level2-1", "Level2-2"}
```

```
funk.Get(foo, "Bar.Name") // Test
```

- funk.GetOrElse 检索指针点 Else 默认的值

```
str := "hello world"
GetOrElse(&str, "foobar")   // string{"hello world"}
GetOrElse(str, "foobar")    // string{"hello world"}
GetOrElse(nil, "foobar")    // string{"foobar"}
```

- funk.Set 在Struct的路径节点赋值
- funk.MustSet 如果Struct不包含接口字段类型，则丢弃错误

```
var bar Bar = Bar{
    Name: "level-0",
    Bar: &Bar{
        Name: "level-1",
        Bars: []*Bar{
            {Name: "level2-1"},
            {Name: "level2-2"},
        },
    },
}

_ = Set(&bar, "level-0-new", "Name")
fmt.Println(bar.Name) // "level-0-new"

MustSet(&bar, "level-1-new", "Bar.Name")
fmt.Println(bar.Bar.Name) // "level-1-new"

Set(&bar, "level-2-new", "Bar.Bars.Name")
fmt.Println(bar.Bar.Bars[0].Name) // "level-2-new"
fmt.Println(bar.Bar.Bars[1].Name) // "level-2-new"
```

- funk.Prune 复制仅包含选定的结构体字段的结构。切片是通过修剪所有元素来处理的。
- funk.PruneByTag 与 funk 相同的功能。但使用tag而不是结构体字段名称

```
bar := &Bar{
    Name: "Test",
}

foo1 := &Foo{
    ID:        1,
    FirstName: "Dark",
    LastName:  "Vador",
    Bar:       bar,
}

pruned, _ := Prune(foo1, []string{"FirstName", "Bar.Name"})
```

```
// *Foo{
//    ID:         0,
//    FirstName: "Dark",
//    LastName:  "",
//    Bar:       &Bar{Name: "Test},
// }
```

- funk.Keys 创建一个数组来列举Map的Key或者结构体字段名称

```
funk.Keys(map[string]int{"one": 1, "two": 2}) // []string{"one", "two"}
(iteration order is not guaranteed)

foo := &Foo{
    ID:        1,
    FirstName: "Dark",
    LastName:  "Vador",
    Age:       30,
}

funk.Keys(foo) // []string{"ID", "FirstName", "LastName", "Age"}
(iteration order is not guaranteed)
```

- funk.Values 创建一个数组来列举Map的值或者结构体字段的值

```
funk.Values(map[string]int{"one": 1, "two": 2}) // []int{1, 2} (iteration
order is not guaranteed)

foo := &Foo{
    ID:        1,
    FirstName: "Dark",
    LastName:  "Vador",
    Age:       30,
}

funk.Values(foo) // []interface{}{1, "Dark", "Vador", 30} (iteration order
is not guaranteed)
```

- funk.ForEach 遍历一个Map或者切片

```
funk.ForEach([]int{1, 2, 3, 4}, func(x int) {
    fmt.Println(x)
})
```

- funk.ForEachRight 从右侧遍历一个Map或者切片

```go
results := []int{}

funk.ForEachRight([]int{1, 2, 3, 4}, func(x int) {
    results = append(results, x)
})

fmt.Println(results) // []int{4, 3, 2, 1}
```

- funk.Chunk 创建一个数组，这些元素按大小的长度拆分为组。如果数组不能均匀拆分，则最后一个块将是剩余的元素。

```go
funk.Chunk([]int{1, 2, 3, 4, 5}, 2) // [][]int{[]int{1, 2}, []int{3, 4},
[]int{5}}
```

- funk.FlattenDeep 递归平展数组

```go
funk.FlattenDeep([][]int{[]int{1, 2}, []int{3, 4}}) // []int{1, 2, 3, 4}
```

- funk.Uniq 创建具有唯一值的数组

```go
funk.Uniq([]int{0, 1, 1, 2, 3, 0, 0, 12}) // []int{0, 1, 2, 3, 12}
```

- funk.Drop 创建一个数组/切片，其中删除了 n 个元素

```go
funk.Drop([]int{0, 0, 0, 0}, 3) // []int{0}
```

- funk.Initial 获取数组中除最后一个元素之外的所有元素

```go
funk.Initial([]int{0, 1, 2, 3, 4}) // []int{0, 1, 2, 3}
```

- funk.Tail 获取数组中除第一个元素之外的所有元素

```go
funk.Tail([]int{0, 1, 2, 3, 4}) // []int{1, 2, 3, 4}
```

- funk.Shuffle 创建随机值的数组

```go
funk.Shuffle([]int{0, 1, 2, 3, 4}) // []int{2, 1, 3, 4, 0}
```

- funk.Subtract 返回两个集合之间的减法。它保持顺序

```go
funk.Subtract([]int{0, 1, 2, 3, 4}, []int{0, 4}) // []int{1, 2, 3}
funk.Subtract([]int{0, 3, 2, 3, 4}, []int{0, 4}) // []int{3, 2, 3}
```

- funk.Sum 累加

```go
funk.Sum([]int{0, 1, 2, 3, 4}) // 10.0
funk.Sum([]interface{}{0.5, 1, 2, 3, 4}) // 10.5
```

- funk.Reverse 把一个数组反转顺序

  ```
  funk.Reverse([]int{0, 1, 2, 3, 4}) // []int{4, 3, 2, 1, 0}
  ```

- funk.SliceOf 返回基于元素的切片

  ```
  funk.SliceOf(f) // will return a []*Foo{f}
  ```

- funk.RandomInt 基于最小值和最大值生成随机整数

  ```
  funk.RandomInt(0, 100) // will be between 0 and 100
  ```

- funk.RandomString 生成具有固定长度的随机字符串

  ```
  funk.RandomString(4) // will be a string of 4 random characters
  ```

- funk.Shard 生成具有固定长度和深度的分片字符串

  ```
  funk.Shard("e89d66bdfdd4dd26b682cc77e23a86eb", 1, 2, false) //
  []string{"e", "8", "e89d66bdfdd4dd26b682cc77e23a86eb"}

  funk.Shard("e89d66bdfdd4dd26b682cc77e23a86eb", 2, 2, false) //
  []string{"e8", "9d", "e89d66bdfdd4dd26b682cc77e23a86eb"}

  funk.Shard("e89d66bdfdd4dd26b682cc77e23a86eb", 2, 3, true) //
  []string{"e8", "9d", "66", "bdfdd4dd26b682cc77e23a86eb"}
  ```

- funk.Subset 如果一个集合是另一个集合的子集，则返回 true

  ```
  funk.Subset([]int{1, 2, 4}, []int{1, 2, 3, 4, 5}) // true
  funk.Subset([]string{"foo", "bar"},[]string{"foo", "bar", "hello", "bar",
  "hi"}) //true
  ```

- 例子

```
abstract class MessageHandler {
    void handleMessage(Message msg) {
        // ...
        msg.setClientCode(getClientCode());
        // ...

        sendMessage(msg);
    }

    abstract String getClientCode();
    // ...
```

```
}

class MessageHandlerOne extends MessageHandler {
    String getClientCode() {
        return "ABCD_123";
    }
}

class MessageHandlerTwo extends MessageHandler {
    String getClientCode() {
        return "123_ABCD";
    }
}
```

```
class MessageHandler {
    void handleMessage(Message msg, Function getClientCode) {
        // ...
        Message msg1 = msg.setClientCode(getClientCode());
        // ...
        sendMessage(msg1);
    }
    // ...
}

String getClientCodeOne() {
    return "ABCD_123";
}

String getClientCodeTwo() {
    return "123_ABCD";
}

MessageHandler handler = new MessageHandler();
handler.handleMessage(someMsg, getClientCodeOne);
```

-
- 函数式选项

```
type User struct {
    ID     string
    Name   string
    Age    int
    Email  string
    Phone  string
    Gender string
}
```

```go
type Option func(*User)

func WithAge(age int) Option {
    return func(u *User) {
        u.Age = age
    }
}

func WithEmail(email string) Option {
    return func(u *User) {
        u.Email = email
    }
}

func WithPhone(phone string) Option {
    return func(u *User) {
        u.Phone = phone
    }
}

func WithGender(gender string) Option {
    return func(u *User) {
        u.Gender = gender
    }
}

func NewUser(id string, name string, options ...func(*User)) (*User, error) {
    user := User{
        ID:     id,
        Name:   name,
        Age:    0,
        Email:  "",
        Phone:  "",
        Gender: "female",
    }
    for _, option := range options {
        option(&user)
    }
    //...
    return &user, nil
}

func main() {
    user, err := NewUser("1", "Ada", WithAge(18), WithPhone("123456"))
    if err != nil {
        fmt.Printf("NewUser: err:%v", err)
    }
    fmt.Printf("NewUser Success")
```

```
}
```

- 工厂方法

```go
// 工厂方法模式
type Person03 struct {
  name string
  age  int
}

func NewPersonFactory(age int) func(name string) Person03 {
  return func(name string) Person03 {
    return Person03{
      name: name,
      age:  age,
    }
  }
}

func use() {
  newBaby := NewPersonFactory(1)
  baby := newBaby("john")
  fmt.Print(baby)

  newTeenager := NewPersonFactory(16)
  teen := newTeenager("jill")
  fmt.Print(teen)
}
```

- 装饰器模式

```go
type Object func(int) int

func LogDecorate(fn Object) Object {
  return func(n int) int {
    log.Println("Starting the execution with the integer", n)

    result := fn(n)

    log.Println("Execution is completed with the result", result)

    return result
  }
}
```

- Spring webflux 函数式编程web框架

```java
@Component
public class HelloWorldHandler {

    public Mono<ServerResponse> helloWorld(ServerRequest request){
        return ServerResponse.ok()
                .contentType(MediaType.TEXT_PLAIN)
                .body(BodyInserters.fromObject("hello flux"));
    }
}
```

```java
@Configuration
public class RouterConfig {

    @Autowired
    private HelloWorldHandler helloWorldHandler;

    @Bean
    public RouterFunction<?> helloRouter() {
        return RouterFunctions.route(RequestPredicates.GET("/hello"),
helloWorldHandler::helloWorld);
    }
}
```

-