Go泛型设计

×

赵柯

QQ音乐

目录

第一部分

Go泛型发展史

什么是泛型?

1967年，克里斯托弗·斯特雷奇在《Fundamental Concepts in Programming Languages》提出了两个概念：

1. 特设多态（ad-hoc）：

```
void print(string s) {}
void print(int i) {}
```

2. 参数化多态（Parametric）：

```
template <typename T>
T Add(T a, T b)
{
        return a + b;
}
```

3. 子类型多态，行多态……：

Go中的interface

鸭子定律：

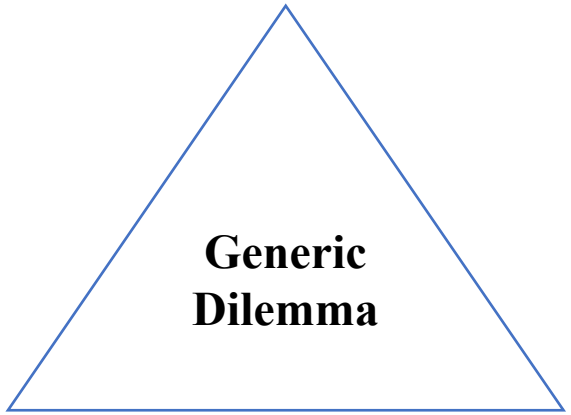
当看到一个动物走起来像鸭子、游起来像鸭子、叫起来也像鸭子，那么这只动物就可以被称为鸭子。

```
type Duck interface {
    walk()
    quack()
}

type unknown struct {}

func (u unknown) walk() {}
func (u unknown) quack() {}
```

泛型困局

编码速度 or 编译速度 or 运行速度？

C++(slow compiler)



C(*slow programmer*)

JAVA(*slow execution times*)

早期提案 – Type Functions

```
type Lesser(t) interface {
        Less(t) bool
}

func Min(a, b t type Lesser(t)) t {
    if a.Less(b) {
        return a
    }
    return b
}
```

1. MyVector(t)(v) 看上去像是两次函数调用
2. type关键字使用混乱
3. 部分场景实现困难：
    1. 支持通用运算符
    2. 支持泛型方法

早期提案 – Generalized Types(广义类型)

```
gen [t] type Lesser(t) interface {
        Less(t) bool
}
gen [t Lesser[T]] func Min(a, b t) t {
    if a.Less(b) {
        return a
    }
    return b
}
```

1. 借鉴了C++的设计，但书写不友好
2. 依然缺乏部分场景解决方案：
    1. 支持通用运算符
    2. 支持泛型方法

早期提案 – Type Parameters

```
type [T] Lesser interface {
        Less(t) bool
}
func [T] Min(a, b T) T {
    if a.Less(b) {
        return a
    }
    return b
}
```

1. 非常接近最终形态
2. 对之前遗留的问题开始寻找解决方案：
    1. 完善了类型推导方案
    2. 完善了类型检查方案
3. 问题：
    1. 语法：类型参数定义在左边，使用在右边
    2. 类型推导虽然提出了方案，但过于复杂不好工程化

其他

1. [] or <> ?

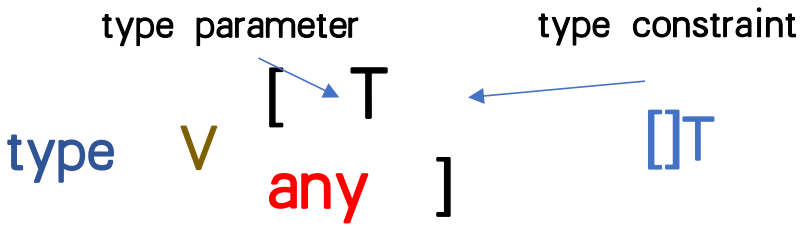
```
a, b = w < x, y > (z) => a = w < x ; b = y > z
```

2. 合约

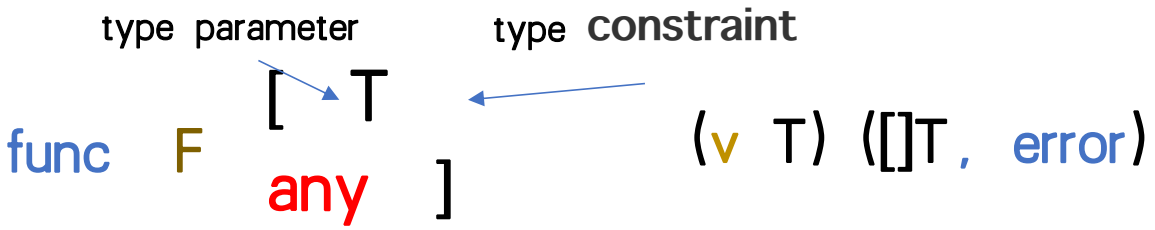
```
contract Comparable(T) {
	T int,int8,int16,float32,float64,string
	// ...
}

func Min(type T Comparable) (a,b T) T {
	if a < b {
		return a
	}
	return b
}
```

最终方案 – Type Parameters（2021）

1. 泛型类型定义



2. 泛型函数定义：



1. 使用[]定义，传递类型参数
2. 定义使用都在右边
3. 使用类型集约束参数范围
4. 支持类型推导
5. 不支持泛型方法

第二部分

Go泛型设计实现提案

泛型设计方案

1. 静态方案（C++模版，rust）

   编译期根据模版参数或者类型推导，为所有类型生成函数副本

2. 动态方案（JAVA，Go interface）

   只有一份函数副本，使用类型擦除，在调用时转换为统一类型，记录原始参数信息

   在运行时进行类型转换

模版（stenciling）

```
type Op interface{
        int|float
}

func Add[T Op](m, n T) T {
        return m + n
}

// 生成后 =>
func Add[go.shape.int_0](m, n int) int{}
func Add[go.shape.float_0](m, n float) float{}
```

1. 总体思路，为每种类型生成副本，编译期替换
2. 没有运行时开销
3. 增加了编译成本（额外的时间 + 二进制大小）
4. 部分场景很难在编译期推导类型

字典（dictionaries）

```
type Op interface{
        int|float
}

func Add[T Op](m, n T) T {
        return m + n
}

// 生成后 =>
const dict = map[type] typeInfo{
        int : intInfo{
                newFunc,
                lessFucn,
                // … …
        },
        float : floatInfo
}
func Add(dict[T], m, n T) T{}
```

1. 总体思路：
   a）基于装箱的思路，只生成一份代码
   b）把每个调用类型信息保存在字典中
   c）函数调用时，用AX寄存器(AMD)传递字典信息

2. 编译期开销少，不会增加二进制大小

3. 增加了运行时开销，

4. 增加了编译器工程难度
   a）字典去重
   b）递归
   c）闭包
   d）… …

5. 编译期丢失类型，很多优化无法展开

混合方案（GC Shape Stenciling）

```
type V interface{
        int|float|*int|*float
}
func F[T V](m, n T) {}

// 1. 为常规类型int/float生成模版
func F[go.shape.int_0](m, n int){}
func F[go.shape.float_0](m, n int){}

// 2. 指针类型复用同一份模版
func F[go.shape.*uint8_0](m, n int){}

// 3. 调用时增加字典传递
const dict = map[type] typeInfo{
        int : intInfo{},
        float : floatInfo{}
}
func F[go.shape.int_0](dict[int] ,m, n int){}
```

1. 总体思路:
   a）相同gcshape的类型复用同一份代码
   b）所有指针类型复用*uint8类型
   c）为所有类型生成字典，函数调用时用寄存器传递（编译器负责）
   d）用字典来处理相同gcshape的不同行为

2. 权衡了工程复杂度，编译时间和运行速度

3. 一般情况，运行时开销忽略不计

4. 需要运行时获取字典信息时，有性能损耗

5. 增加了编译成本（额外的时间 + 二进制大小）

Gcshape

判断gcshape是否相同的条件：

- 都是指针，则认定为*uint8
- 基础类型（ underlying type ）相同

```
1. type myInt int , type muInt = int          // 基础类型都是int，复用
2. int32 ,uint32                              // 基础类型不同,不能复用
3. type s1 struct{} , type s2 = struct{}      // 基础类型不同，不能复用
4. type p1 *int , type p2 = *int              // 基础类型都是指针，复用*uint8
```

相同gcshape的不同行为

```
// 类型集
type Print interface {
        String()
}
type myInt int
func (i *myInt) String() { fmt.Println(“im myInt:”, i) }
type myFloat float32
func (i *myFloat) String() {fmt.Println("im myInt:", i)}

// 泛型函数
func String[T Print](n T) { n.String() }
func main() {
    var (
        i myInt = 10
        f myFloat = 10.0
    )
    // 通过指针传参，会复用*uint8
    String(&i)
    String(&f)
}
```

字典

```
// Go 1.20 字典结构
type writerDict struct {

    // 保存类型参数的函数表（类似C++的虚表）
    typeParamMethodExprs []writerMethodExprInfo

    // 子字典。内部调用其他泛型函数/方法时，需要传入当前泛型函数/方法的类型参数
    subdicts []objInfo

    // 类型参数的类型信息
    rtypes []typeInfo

    // 需要用itabs做类型转化。 比如使用reflect（反射）
    itabs []itabInfo
}
```

字典

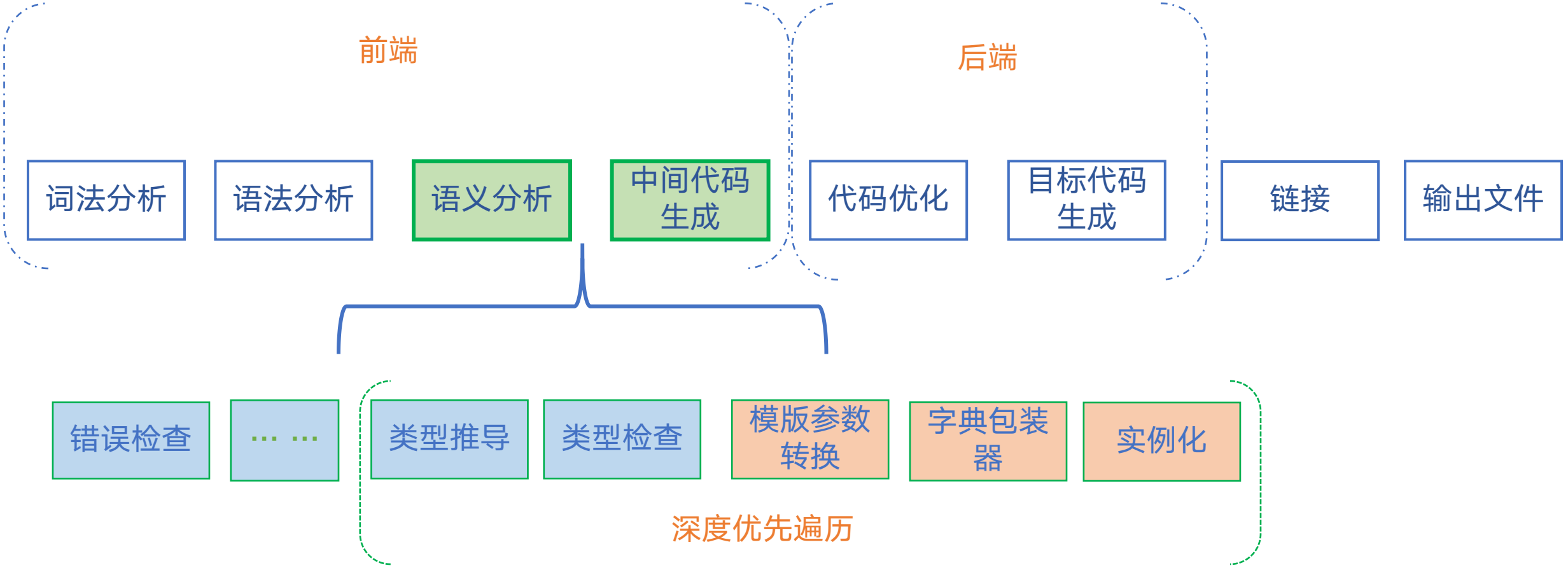
```
// 泛型函数
func Compare[T Comparable](n
T) {
    n.Less()
    n.Greater()
}

func main() {
    var i Foo
    Compare(&i)
}
```

```
main..dict.Compare[*main.Foo] SRODATA dupok size=24
    rel 0+0 t=23 type:*main.Foo+0
    // 函数表
    rel 0+8 t=1 main.(*Foo).Less+0
    rel 8+8 t=1 main.(*Foo).Greater+0

    // 类型信息
    rel 16+8 t=1 type:*main.Foo+0

    // 子字典和itabs为空
```

第三部分

Go泛型底层实现原理

编译过程


前端
后端
词法分析
语法分析
语义分析
中间代码生成
代码优化
目标代码生成
链接
输出文件
错误检查
… …
类型推导
类型检查
模版参数转换
字典包装器
实例化
深度优先遍历

单态化（Monomorphization）

模版参数替换：

```
func F[T any,U any](a T,b U) {
        return
}

// 单态化

F[int,string](1,"1")        => F[go.shape.int_0, go.shape.string_0]

// 使用编号区分类型
F[int,int](1,1)             => F[go.shape.int_0, go.shape.int_1]

// 指针复用
F[*int,*string](nil,nil)    => F[go.shape.*uint8_0, go.shape.*uint8_1]
```

单态化调用

```go
package main

func F[T any, U any](a T, b U) {
        return
}

func main() {
    F[int, string](1, "hello")
    F[int, int](1, 2)
    F[*int, *string](nil, nil)
}
```

```
LEAQ main..dict.F[int,string](SB), AX
CALL main.F[go.shape.int,go.shape.string](SB)
```

```
LEAQ main..dict.F[int,int](SB), AX
CALL main.F[go.shape.int,go.shape.int](SB)
```

```
LEAQ main..dict.F[*int,*string](SB), AX
CALL main.F[go.shape.*uint8,go.shape.*uint8](SB)
```

不完全单态化

```go
package main

// 类型集
type Comparable interface {
    Less()
    Greater()
}
type Foo struct{}

func (i Foo) Less() {}
func (i Foo) Greater() {}

// 泛型函数
func Compare[T Comparable](n T) {
    n.Less()
    n.Greater()
}
func main() {
    var i Foo
    Compare(&i)
}
```

main.Compare[go.shape.*uint8]

```
// 1.20之前会有两次
MOVQ (AX), CX
MOVQ AX, DX
MOVQ BX, AX
PCDATA $1, $1
CALL CX
```

```
MOVQ main..dict+32(SP), DX
LEAQ 8(DX), CX
MOVQ 8(DX), BX
MOVQ main..autotmp_2+8(SP),
AX
MOVQ CX, DX
PCDATA $1, $2
NOP
CALL BX
```

interface实现

```go
package main
type Comparable interface {
    Less()
    Greater()
}
type Foo struct{}

func (i Foo) Less() {}
func (i Foo) Greater() {}

func Compare(n Comparable) {
    n.Less()
    n.Greater()
}

func main() {
    var i Foo
    Compare(&i)
}
```

main.Compare

```
    MOVQ AX, main.n+24(SP)
    MOVQ BX, main.n+32(SP)
    PCDATA $3, $-1
    MOVQ 32(AX), CX
    MOVQ BX, AX
    PCDATA $1, $0
    CALL CX
```

```
    MOVQ main.n+24(SP), CX
    MOVQ 24(CX), CX
    MOVQ main.n+32(SP), AX
    PCDATA $1, $1
    CALL CX
```

性能对比 — 模版

```go
// 普通函数
func Add(a, b int) int {
	return a + b
}

// 泛型函数
type IntegerType interface {
	int | uint
}

func AddGeneric[T IntegerType](a, b T) T {
	return a + b
}
```

测试结果：没有性能损耗

```
goos: darwin
goarch: arm64

BenchmarkAdd-10             0.3225 ns/op
BenchmarkAddGeneric-10      0.3201 ns/op
```

性能对比 — 字典（查找函数表）

```go
// 接口
type Foo interface {
	Print()
}
// 泛型类型A
type A[T any] struct { n T}
func (a A[T]) Print() {}
```

```go
// 1. 直接调用
var a A[int]
a.Print()
// 2. 泛型函数调用
func Print[T Foo](a T) {
	a.Print()
}
Print(a)
// 3.接口调用
var f Foo = a
f.Print()
```

测试结果： 查找函数表 有一定性能损耗

```
goos: darwin
goarch: arm64

BenchmarkNormal-10          0.3768 ns/op

BenchmarkGeneric-10         1.132 ns/op

BenchmarkPrintIF-10         1.125 ns/op
```

性能对比 — 字典（查找类型）

```go
// 接口
type Foo interface {
	Print()
}
// 泛型类型A
type A[T any] struct { n T}
func (a *A[T]) Print() {}
```

```go
// 1. 直接调用
var a A[int]
a.Print()
// 2. 泛型函数调用
func Print[T Foo](a T) {
	a.Print()
}
Print(&a)
// 3.接口调用
var f Foo = a
f.Print()
```

测试结果： 查找类型信息，性能损耗更高

```
goos: darwin
goarch: arm64

BenchmarkNormal-10          0.3752 ns/op

BenchmarkGenericPtr-10      2.402 ns/op

BenchmarkPrintIF-10         1.124 ns/op
```

编译速度优化

unified IR（统一IR）：

1. 添加泛型支持后，编译器冗余逻辑太多，拖慢了编译速度
2. 编译器对泛型的处理逻辑独立，导致BUG不断，往往解决完一个问题，又引入新的问题
3. unified IR重构了IR部分逻辑，把重复遍历AST的步骤融合，加快了编译速度。
4. 内联器可以处理更多的场景，如：for/switch, 以及PGO
5. 优化字典结构，减少运行时开销
6. … …

第四部分

总结

总结:

1. Go当前泛型方案对基本数据结构支持较好，无运行时开销。
2. 尽量不要把接口类型传递给泛型函数
3. Go团队对目前的泛型方案一直在优化
4. 不支持泛型方法，不支持匿名结构体/匿名函数，不支持类型断言

思考

1. 考虑支持泛型方法
2. 能否完全单态化？ 彻底消除运行时开销

end

附1: iface

```
type iface struct {
    tab *itab               //offset 0
    data unsafe.Pointer
}


type itab struct {
    inter *interfacetype    // offset 0
    _type *_type            // offset 8
    hash uint32             // offset 16
    _ [4]byte               // offset 20
    fun [1]uintptr          // offset 24
    ...
}
```