

# Go, 基于连接与组合的语言

许式伟  
@七牛云存储

# 编程范式

- 过程式
  - C
- 面向对象
  - Java、C#
- 面向消息
  - Erlang
- 函数式
  - Haskell

# Go 语言的编程范式?

范式	Y/N
过程式	Y
面向对象	Y
面向消息	Y (支持但不纯粹)
函数式	Y (支持但不纯粹)

无门派的语言：和C++类似，Go语言是多范式的语言，并没有遵循特定的流派。

# Go 语言的编程范式?

- Go 无门无派
  - Go 支持过程，但只是语言基础特性。
  - Go 支持对象，但将特性最小化，只作为语言基础特性。Go 甚至反对继承，拒绝提供继承语法。
  - Go 支持消息(channel)，但并没有杜绝共享内存，只是将消息作为语言基础特性。
  - Go 支持闭包，但并没有试图成为纯粹的函数式语言，只是将其作为语言基础特性。

# Go 的独门武功

- 一门基于连接与组合的语言
  - 连接：组件之间的耦合方式
    - 非侵入式的 interface
    - 抽象的 `io.Reader`, `io.Writer` 和 `Pipe`
  - 组合：形成复合对象的基础
    - 强大的组合能力
      - 匿名组合
      - 指针组合
      - 接口组合

# 从 Unix 谈起

- Unix 的连接和组合
  - `app1 params1 | app2 params2`
- App 接口
  - 输入: `stdin`, `params`
  - 输出: `stdout`
  - 协议: `text (data stream)`
- Pipeline
  - 将一个 app 的输出(`stdout`) 转为另一个 app 的输入(`stdin`)

# Pipeline 关键点

- 多个 app 是并行执行的
  - 上游每产生一段output，会立即交由下游处理。
- app 间的协议是松散耦合的
  - 上游 app 的 output 是 xml 还是 json，下游 app 需要知晓，但是属于一种松散的耦合关系，并无任何强制的约束。

# Go 对 Unix Pipeline 的仿真

- App
  - `func(in io.Reader, out io.Writer, args []string)`

app1 params1 | app2 params2

`bind(app1, params1)`

`bind(app2, params2)`

`pipe( bind(app1, params1), bind(app2, params2) )`



# Go 对 Unix Pipeline 的仿真

```
func bind(  
    app func(in io.Reader, out io.Writer, args []string),  
    args []string  
) func(in io.Reader, out io.Writer) {  
  
    return func(in io.Reader, out io.Writer) {  
        app(in, out, args)  
    }  
}
```

# Go 对 Unix Pipeline 的仿真

```
func pipe(  
    app1 func(in io.Reader, out io.Writer),  
    app2 func(in io.Reader, out io.Writer)  
) func(in io.Reader, out io.Writer) {  
  
    return func(in io.Reader, out io.Writer) {  
        pr, pw := io.Pipe()  
        defer pw.Close()  
        go func() {  
            defer pr.Close()  
            app2(pr, out)  
        }()  
    }
```

# Go 对 Unix Pipeline 的仿真

```
func pipe(apps ...func(in io.Reader, out io.Writer)) func(in io.Reader, out io.Writer) {  
  
    if len(apps) == 0 { return nil }  
    app := apps[0]  
    for i := 1; i < len(apps); i++ {  
        app1, app2 := app, apps[i]  
        app = func(in io.Reader, out io.Writer) {  
            pr, pw := io.Pipe()  
            defer pw.Close()  
            go func() {  
                defer pr.Close()  
                app2(pr, out)  
            }()  
            app1(in, pw)  
        }  
    }  
    return app  
}
```



难以想象的优雅！

# Pipeline vs. Filter

- Pipeline

```
func tar(io.Reader, out io.Writer, files []string)
func gzip(in io.Reader, out io.Writer)
pipe( bind(tar, files), gzip )(nil, out)
```

- Filter

```
func tar(io.Reader, out io.Writer, files []string)
func gzipf(w io.Writer) io.Writer
tar( nil, gzipf(out), files )
```

# 转换为 Filter

```
func filter(  
    app func(in io.Reader, out io.Writer)  
) func(w io.Writer) io.WriteCloser {  
  
    return func(w io.Writer) io.WriteCloser {  
        pr, pw := io.Pipe()  
        go func() {  
            defer pr.Close()  
            app(pr, w)  
        }()  
        return pw  
    }
```

# 更严格的 Filter 范式

```
func tar(io.Reader, out io.Writer, files []string)
```

```
func gzipf(w io.Writer) io.WriteCloser
```

```
gw := gzipf(out)
```

```
defer gw.Close()
```

```
tar(nil, gw, files)
```

# 结论

- 在 Go 中实施 Pipeline 非常容易。
- 在 Go 中让任务并行化非常容易。

# 面向对象经典故事：Shape

- Java 版本

```
interface Shape {  
    double area();  
}
```

名不符实的 Shape 定义



# 面向对象经典故事：Shape

- Circle

```
class Circle implement Shape {  
    private double x, y, r;  
    public double area() {  
        return math.Pi/2 * this.r * this.r;  
    }  
}
```

- Rect

```
class Rect implement Shape {  
    private double x, y, w, h;  
    public double area() {
```

# 面向对象经典故事：Shape

```
class Algorithm {  
    static public double area(Shape... shapes) {  
        double result = 0;  
        for (Shape shape: shapes) {  
            result += shape.Area();  
        }  
        return result;  
    }  
}
```

# 面向对象经典故事：Shape

- Go 版本?

# 面向对象经典故事：Shape

- Circle

```
type Circle struct {  
    x, y, r float64  
}  
func (this *Circle) Area() float64 {  
    return math.Pi/2 * this.r * this.r  
}
```

- Rect

```
type Rect struct {  
    x, y, w, h float64
```

# 面向对象经典故事：Shape

- Shape

```
type Shape interface {  
    Area() float64  
}
```

名不符实的 Shape 定义

```
func Area(shapes ...Shape) float64 {  
    var result float64  
    for _, shape := range shapes {  
        result += shape.Area()  
    }  
    return result  
}
```

# 面向对象经典故事：Shape

- Shape

```
type areaGetter interface {  
    Area() float64  
}
```

更贴切的含义

```
func Area(shapes ...areaGetter) float64 {  
    var result float64  
    for _, shape := range shapes {  
        result += shape.Area()  
    }  
    return result  
}
```

# 面向对象经典故事：Shape

- 或者：

```
func Area(shapes ...interface{ Area() float64 }) float64
{
    var result float64
    for _, shape := range shapes {
        result += shape.Area()
    }
    return result
}
```

# 面向对象经典故事：Shape

- 如何连接这些组件？

```
area := Area( &Rect{w: 20, h: 30}, &Circle{r: 20} )  
fmt.Println("area:", area)
```



# 结论

- Go 组件的连接是松散耦合的。彼此之间有最自然的独立性。
- Go 组件间的协议由 `interface` 描述，并在编译期进行 `check`。

# 强大的组合能力

- 简单组合

```
type Foo struct {  
    x, y float64  
    bar Bar  
}
```

# 强大的组合能力

- 匿名组合

```
type Foo struct {  
    Bar  
}
```

```
type Foo struct {  
    *Bar  
}
```

```
type Foo struct {  
    io.Reader
```

# 强大的组合能力

- eg. archive/zip.ReadCloser

```
type ReadCloser struct {  
    Reader  
    io.Closer  
}
```

```
func OpenReader(fname string) (*ReadCloser, error)  
{
```

```
    f, err := os.Open(fname)  
    if err != nil { return nil, err }
```

# 联想： Windows COM 编程思想

- QueryInterface
- 聚合
- 盲目聚合

# 结论

- 不支持继承，却胜过继承
- 不是 COM，但更胜 COM

# Go 不支持的编程范式

- 虚函数重载
  - 典型案例：MFC

```
class MyDialog : public CDialog {  
public:  
    void OnInitDialog() { ... }  
    void OnDestroy() { ... }  
};
```

# why not 虚函数重载?

- 违背了连接与组合的编程范式
  - 虚函数重载将算法与环境抽象的代码揉在一起，加深了系统的耦合。



# 推荐做法

- 如果环境是稳定的：

```
type DialogEvent interface {  
    OnInitDialog()  
    OnDestroy()  
}  
  
type Dialog struct {  
    ...  
    DialogEvent  
}  
  
func NewDialog(event DialogEvent) *Dialog  
{  
    return &Dialog{..., event}  
}
```

```
type MyDlgEvent struct {  
    ...  
}  
  
func (this *MyDlgEvent) OnInitDialog() { ...  
}  
func (this *MyDlgEvent) OnDestroy() { ... }  
  
func NewMyDialog() *Dialog {  
    event := &MyDlgEvent{...}  
    return NewDialog(event)  
}
```

# 推荐做法

- 如果环境是不稳定的：

```
type Dialog struct { ... }
```

```
func (this *Dialog) Bind(eventName string, eventAction func()) { ... }
```

```
type MyDialog struct {
```

```
    Dialog
```

```
}
```

```
func NewMyDialog() *MyDialog {
```

```
    dlg = new(MyDialog)
```

```
    dlg.Bind("OnInitDialog", func() { ... })
```

```
    dlg.Bind("OnDestroy", func() { ... })
```

```
    ...
```

```
    return dlg
```

```
}
```

# Q & A

@许式伟

@七牛云存储