Go语言程序设计

一、概述

- 大道至简的设计哲学
  - 没有继承、构造、析构、虚构、函数重载、默认参数等。
  - 少即是多，只有通过简洁的设计，才能让一个系统保持稳定、安全和持续的进化。
  - Go项目是在Google公司维护超级复杂的几个软件系统遇到的一些问题的反思。
- 为并发而生
  - 语言层次支持并发模型：goroutine

```
go func(){
    ...
}()
```

  - goroutine比线程更轻量，可以轻松跑上万个goroutine
- 支持垃圾回收

  消除了并发编程中的对象生命周期管理的负担
- 非侵入式接口
  - 鸭子类型

    > 如果一个动物像鸭子一样走路，像鸭子一样呱呱叫，那它就是一只鸭子
  - 支持接口查询

```
if v, ok := v.(IFile); ok {
    ...
}
```

- 极度简化但完备的面向对象方法

  废弃了大量OOP特性，只保留组合和嵌入方式
- 标准化的错误处理规范
  - 内置error
  - defer语句编写异常安全代码
- 适合云计算
  - 性能大幅领先python、ruby、php等脚本语言，接近C、C++
  - 腾讯、阿里、京东、360、网易、新浪、金山、豆瓣等都有团队对go做服务端开发进行实践
  - 目前用Go实现比较火的应用：Docker、TiDB
  - 2016再次获得年度编程语言

二、布尔与数字类型

开始

- 命令行运行 go version，如果出错则把如下脚本加入 ~/.profile

```
export GOROOT=/HOME/opt/go
export PATH=$PATH:$GOROOT/bin
```

- 编译 go build，编译速度秒杀C++几条街
- go程序做脚本用：gonow gorun
- IDE: VS Code、LiteIDE、Gogland

基础

- 关键字

  ```
  package import func interface struct map
  switch type case default fallthrough
  if else for range break contiue goto return
  chan select go defer const var
  ```

- 预定义标识符
  - 内建常量:

    ```
    true false iota nil
    ```

  - 内建类型:

    ```
    int int8 int16 int32 int64
    uint uint8 uint16 uint32 uint64 uintptr
    float32 float64 complex128 complex64
    bool byte rune string error
    ```

  - 内建函数:

    ```
    make len cap new delete append copy close
    complex real imag
    panic recover
    ```

- 常量、变量

```
const(
    Cyan = iota
    Magenta
    Yelow
)
```

- 不支持隐式类型转换，不同类型必须显式类型转换

  `type(value)`

- 大数值类型

    - `big.Int`
    - `big.Rat`
- 不支持操作符重载
- ++、--只支持后缀方式
- 除非特殊说明，math包所有函数都用float64

三、字符串

- unicode码点用rune表示（4字节）
- 字符串用双引号“或 反引号` 创建
- []rune(s) 将字符串转换成Unicode码点
- += 拼凑低效，建议用 strings.Join 或 bytes.Buffer
- 字符串保存为utf8，用for...range遍历，非ASCII索引更新的步长将超过1个字节（建议先转[]rune），
- utf8.DecodeRuneInString()获取第一个字符的位置和大小
- strings.Map()可用来替换或去掉字符串中的字符（返回负数则原来字符删除）
- 相关包：fmt、strings、strconv、utf8、unicode、regexp

四、集合类型

- 对于chan、func、map、slice变量，持有的为引用，其他都持有值
- 传递数组按值传递，代价非常大，通常不用数组，用slice
- 创建变量同时获取指针：new(Type)、&Type{}
- 数组创建方式：

```
[len]Type
[len]Type{v1, v2, v3..., vn}
[...]Type{v1, v2, v3..., vn}
```

- s[:cap(s)] :增加切片长度到其容量
- 切片创建方式：

```
make([]Type, len, cap)
make([]Type, len)
[]Type{}
[]Type{v1, v2, v3..., vn}
```

- 使用...加在切片后用于把切片当成多个元素(同不定长参数正好相反)

```
s = append(s, u[2:5]...)
```

- 相关函数：append、copy、len、cap
- 相关包：sort
- map的操作

```
m[k] = v1
delete(m, k)
v := m[k]
v, found := m[k]
len(m)
```

- map比切片的字节索引慢2个数量级（100倍），不过也足够快
- map的创建方式：

```
make(map[KeyType]ValueType, cap)
make(map[KeyType]ValueType)
map[KeyType]ValueType{}
map[KeyType]ValueType{k1: v1, k2: v2..., kn: vn}
```

- struct 可以作为map的key，只要它的成员都支持==和!=运算即可

五、过程式编程

- 类型断言

```
if v, ok := x.(Type); ok {

}
```

- switch

```
switch Suffix(file) {
case ".gz":
    return GzipFileList(file)
case ".tar", "tar.gz", ".tgz":
    return TarFileList(file)
case ".zip":
    return ZipFileList(file)
}
```

- 类型开关

```
switch x.(type){
case bool:
    fmt.Printf("bool\n")
case float64:
    fmt.Printf("float64\n")
}
```

- for

```
//1
for{
    ...
}
//2
for boolexp{
    ...
}
//3
for pre; boolexp; postexp {
    ...
}
//4
for index, char := range s{
    ...
}
//5
for index := range s{
    ...
}
//6
for key, value := range a_map{
    ...
}
//7
for key := range a_map{
    ...
}
//8
for item := range a_chan{
    ...
}
```

- 通信和并发
  - goroutine创建方式：
    - `go function(arg)`
    - `go func(param){...}(arg)`
  - 通道创建方式：
    - `make(chan Type)`
    - `make(chan Type, cap)`
  - 发送

    `channel <- value`
  - 接收

```
<- channel //接收并丢弃
value := <- channel //接收并保存
value, ok := <- channel //接收并保存，同时检查通道是否关闭或者是否为空
```

- select语句

```
select {
case send_or_recv: block1
...
case send_or_recvN: blockn
default: block_d
}
```

  - 没带default语句的是阻塞的
  - 带default语句的是非阻塞的

- defer语句

```
if file, err := os.Open(filename); err != nil {
    log.Println("file open err", err)
    return
}
defer file.Close()
```

  - 多defer语句时按LIFO（后进先出）方式执行

- panic、recover函数
  - error指可能出错的东西
  - panic指不可能发生的事情
  - 绝大多数情况，go语言标准库使用error而非异常
  - 如果使用panic()，需要避免panic跨越package的边界，可以使用recover()来捕捉异常并且返回一个error
  - panic to error

```
func IntFromInt64(x int64) (i int, err error){
    defer func(){
        if e := recover(); e != nil{
            err = fmt.Errorf("%v", e)
        }
    }()
    i = ConvertInt64ToInt(x)
    return i, nil
}
```

  - log panic wrapper

```
func logPanic(function func(http.ResponseWriter, *http.Request))
    func(http.ResponseWriter, *http.Request){
    return func(write http.ResponseWriter, request *http.Request){
        defer func(){
            if (x := recover(); x != nil){
                log.Printf("[%v] caugth panic: %v", request.RemoteAddr,
x)
            }
        }()
        function(writer, request)
    }
}
```

- 可变参数函数

```
func MinimumInt(first int, rest...int){
    for _, x := range rest{
        if x < first{
            first = x
        }
    }
    return first
}
```

  - `arg... Type` 多参数变切片
  - `slice...` 切片变多个参数

- 可选参数

```
type Options struct{
    First int
    Last  int
    Audit bool
    ErrorHandle func(item Item)
}
//default arg
ProcessItems(items, Options{})
//assign some arg
errorHandle := func(item Item){ log.Println("Invalid:", item)}
ProcessItems(items, Options{Audit: true, ErrorHandle: errorHandle})
```

- init,main函数
  - 包被引入多次，init函数也只执行一次
- 闭包函数
  - 捕获了和它同一作用域的其他常量和变量，只要闭包还在使用，变量还会存在
  - 所有匿名函数都是闭包

- 递归函数
  - 一个跳出条件
  - 一个递归体
- 使用map代替if、switch分支

```
var FunctionForSuffix = map[string] func(string)([]string, error){
    ".gz": GzipFileList,
    ".tar": TarFileList,
    ".tar.gz": TarFileList,
    ".tgz": TarFileList,
    ".zip": ZipFileList
}
func ArchiveFileListMap(file string)([]string, error){
    if function, ok := FunctionForSuffix[Suffix(file)]; ok {
        return function(file)
    }
    return nil, errors.New("unrecognized archive")
}
```

  - 50个以上分支，使用map速度会超过switch
- 纯记忆函数
  - 纯函数：对同一组输入总是产生同样的输出，不存在副作用
  - 记忆技术：保存当前计算结果，下次直接获取

```
type memFunction func(int, ...int) interace{}
var Fibonacci memFunction
func init(){
    Fibonacci = Memoize(func(x int, xs...int) interface{}{
        if x < 2{
            return x
        }
        return Fibonacci(x -1).(int) + Fibonacci(x-2).(int)
    }
}

func Memoize(function memFunction) memFunction {
    cache := make(map[string]interface{})
    return func(x int, xs...int) interface{}{
        key:= fmt.Sprint(x)
        for _, i := range(xs){
            key += fmt.Sprintf(",%d", i)
        }

        if value, ok := cache[key]; ok{
            return value
        }
        value :=function(x, xs...)
        cache[key] = value
        return value
    }
}
```

六、面向对象编程

- 只支持组合和嵌入
- 使用类型和值，自定义的值可以包含方法
- 接口、值、方法都相互保持独立
- 鸭子类型

```
type Writer interface {
    Write(p []byte)(n int, err error)
}

type ZipWriter struct{
    ...
}

func (writer * ZipWrite) Write(p []byte)(n int, err error){
    ...
}

Writer w = ZipWriter{}
```

- 自定义类型
    - 基于内置类型的自定义类型与内置类型的转换在编译时完成

```
type Count int
func (c *Count)Increment(){ *count ++}
```

    - 方法表达式

```
type Part struct{
    Id int
    Name string
}
func (part Part) String() string{
    return fmt.Printf("<<%d %q>>", part.Id, part.Name)
}
...
stringFunc := (* Part).String
sv := stringFunc(&part)
```

- 接口习惯以er结尾
- 接口的嵌入

```
type LowerCaser interface{
    LowerCase()
}
type UpperCaser interface{
    UpperCase()
}
type LowerUpperCaser interface{
    LowerCaser
    UpperCaser
}
```

- 为接口添加方法，建议创建一个新的接口，嵌套老接口在里面，同时包含新老方法（升级）

七、并发编程

- sync / atomic 原子操作
- sync.Once 执行一次性调用
- sync.WaitGroup 上层同步机制，Add、Done、Wait：等待Done调用次数和Add一致
- 保证通道里传递指针或引用类型的安全性的3种方法
    - 使用互斥
    - 设定规则，发送后不再访问，由接受者释放
    - 所有外部方法都不修改值，内部实现goroutine修改值，修改函数不导出
- 只有在后面需要检查通道是否关闭（用到`for...rang, select, <-`）才需要显式关闭通道
- 发送端关闭 chan
- 并发常见的3种模式：管道、独立并发任务、相互依赖的并发任务
- `chan <- Type` 只允许发送的通道
- `<- chan Type` 只允许接受的通道

八、文件处理

- 处理Json文件
    - 使用encoding/json包
    - 需要定义转换结构体，用于处理包含类似time.Time类型的结构体
    - 写json文件

```
type JSONMarshaler struct{}
func (JSONMarshaler) MarshalInvoices(writer io.Writer, invoices
[]*Invoice) error {
    encoder := json.NewEncoder(writer)
    //json.Encode()会检查值是否支持json.MarshalJSON接口，如果支持，会自动调用该
值的MarshalJSON方法而非内置的编码代码
    if error := encoder.Encode(fileType); error != nil{
        return err
    }
    ...
}
```

    - 读json文件

```
func (JSONMarshaler) UnMarshalInvoices(reader io.Reader)([]*Invoie,
error){
    decoder := json.NewDecoder(reader)
    var kind string
    //json.Decode()会检查值是否支持json.UnmarshalJSON接口，如果支持，会自动调用
该值的UnmarshalJSON方法
    if err := decoder.Decode(&king); err != nil{
        return nil, err
    }
    ...
}
```

    - json.Encoder.Encode函数与json.Decoder.Decode函数不是完美可逆
- 处理XML文件
    - 适用encoding/xml包
    - 要求结构体字段包含格式合理的标签（encoding / json不需要）
        - 结构体的标签本质没有任何语义，它们只是可以通过反射接口获得的字符串
    - 一般也需要定义转换结构体（带标签）
    - 写xml文件

```
type XMLInvoies struct{
    //
    XMLName xml.Name `xml:"INVOICES"`
    Version int `xml:"version,attr"`
    Invoice []*XMLInvoice `xml:"INVOICE"`
}
type XMLInvoice struct {
    XMLName xml.name `xml:"INVOICE"`
    Id int `xml:",atrr"`
    CustomerId int `xml:",attr"`
    ...
}
```

- 处理文本文件
  - 使用fmt包
  - fmt.Fprintf
  - fmt.Sscanf
- 处理gob二进制文件
  - gob格式是一个自描述的二进制序列
  - 使用encoding/gob包
- 处理自定义二进制文件
  - 使用encoding / binary包
  - 读写二进制数据时其字节序必须一致
  - 随机访问必须使用os.OpenFile函数打开文件（而非os.Open）
- 处理zip归档文件
  - 使用archive/zip包
  - 创建zip归档文件

```
file := os.Create(zipfilename)
zipper := zip.NewWriter(file)

//loop process input file...
inputfile := os.Open(inputfilename)
info := inputfile.Stat()
header := zip.FileInfoHeader(info)
writer := zipper.CreateHeader(header)
io.Copy(writer, inputfile)
```

  - 解开zip归档文件

```
reader := zip.OpenReader(filename)
defer reader.Close()
for _, zipFile := range reader.Reader.File{
    name := sanitizedName(zipFile.Name)
    mode := zipFile.Mode()
    if mode.IsDir(){
        os.MkDirAll(name, 0755)
    }else{
        unpackZippedFile(name, zipFile)
    }
}
func unpackZippedFile(filename string, zipFile *zipFile) error{
    writer := os.Create(filename)
    reader := zipFile.Open()
    io.Copy(writer, reader)
}
```

- 处理压缩的tar包
  - 使用archive / tar、archive / gzip
  - 创建压缩tar包

```
file := os.Create(filename)
fileWriter := gzip.NewWriter(file)
writer := tar.NewWriter(fileWriter)

//loop process input file
inputfile := os.Open(inputfilename)
stat := file.Stat()
header := &tar.Header{
    //净化文件名
    Name: sanitizedName(filename),
    MOde: int64(stat.Mode()),
    Uid: os.Getuid(),
    Gid: os.Getuid(),
    Size: stat.Size(),
    ModTime: stat.ModTime(),
}
writer.WriteHeader(header)
io.Copy(writer, file)
```

  - 解开压缩tar包

```
file := os.Open(filename)
fileReader := gzip.NewReader(file)
reader := tar.NewReader(fileReader)
unpackTarFiles(reader)

func unpackTarFiles(reader *tar.Reader) error{
    for {
        header, err := reader.Next()
        if err != nil {
            if err == io.EOF {
                return nil
            }
            return err
        }
        filename := sanitizedName(header.Name)
        switch header.Typeflag{
        case tar.TypeDir:
            os.MkdirAll(filename, 0755)
        case tar.TypeReq:
            unpackTarFile(file, header.Name, reader)
        }
    }
}

func unpackTarFile(filename, tarFilename string, reader *tar.Reader)
error{
    writer := os.Create(filename)
    io.Copy(writer, reader)
}
```

九、包

- 自定义包
  - 包可以分隔成多个文件保存，只需将这些文件存放在同一个目录
  - 如果希望包能够被其他应用程序共享，那就应该放在GOPATH / src目录下
  - 平台特定代码
    - 判断runtime.GOOS：windows、linux、darwin、freebsd
    - 文件名后缀：例入util_windows.go util_linux.go util_darwin.go util_freebsd.go
  - 文档化相关包
    - 默认只有可导出类型、类、常量、变量才会在godoc出现，因此全部这些内容都应该添加合适的注释
    - 可导出类型的注释必须紧接在类型声明之前，而且必须总是描述该类型的0值是否有效
    - 注释通常以被注释内容的名字开头，这是一个惯例
  - 包的单元测试和基准测试
    - 单元测试文件名格式为 包名_test.go

        - 黑盒测试：单独的测试包，导入被测试包
        - 白盒测试：测试文件放包文件目录下，不需要倒入被测试包，可以为方便测试而增加新的方法
    - 单元测试文件没有main函数，取而代之是一写以Test开头的函数，带一个常数，没有返回值

```
func TestStringKeyOMapInsertion(t *testing.T){
    ...
}
```

    - 基准测试以Benchmark开头
        - 默认不会执行基准测试
        - 需要指定 -test.bench=.* (正则)
        - -test.benchtime选项可以指定基准测试的执行时间

- 导入包
    - 别名用于切换版本：

```
import bio "bio_v1"
import bio "bio_v2"
```

    - 避免未使用的警告

```
import _ "image/gif"
import _ "image/jpeg"
import _ "image/png"
```

- go命令行弓弩
    - 帮助：`go help`
    - 升级包：`go fix`
    - 工具包：`go tool`
    - 格式化代码：`gofmt`
- 标准库
    - 实验性质包
        - exp：除非参与标准库的开发，否则不要使用
    - 归档和压缩
        - archive/tar
        - archive/zip
        - compress/gzip
        - compress / bzip2
        - compress/lzw
    - 字节流和字符串
        - bytes、strings

        - strconv
        - fmt
        - unicode、unicode / utf8、unicode / utf16
        - text/template、html/template
    - 容器包
        - container/heap
        - container/list
        - container/ring
        - database/sql
    - 文件和操作系统相关
        - os
        - bufio
        - io
        - ioutil
        - path、path/filepath
        - runtime
    - 文件格式
        - gob
        - csv
        - encoding、encoding/json、encoding/xml、encoding/binary、encoding/base64
    - 图像处理
        - image、image/png、image / jpeg、image/draw
        - freetype、freetype / raster
    - 数学处理
        - math
        - math/big
        - math/cmplx
        - math / rand
    - 网络
        - net
        - net/http、net / url、net/rpc、net/smtp
    - 反射包
        - reflect
    - 其他
        - cypto、cypto/sha512、cypto/aes、cypto/des
        - exec
        - flag
        - log
        - regexp
        - sort
        - time