View on GitHub

Sevice Computing

Principle, Technology and Architecture for building effitive, elastic and solid services on cloud

面向对象思考与接口抽象-io库分析

抽象，特别是面向接口的抽象，可以使程序变得更加通用。但通常会牺牲程序的性能，并导致程序设计与开发需要高度的技巧。不合理的抽象会导致软件包内在逻辑变得复杂与难以维护。抽象设计是人们在软件设计过程中积累的，其中最丰富的资产就是 java 类库的设计，几乎所有现代语言程序库设计都会吸收 java 库设计的思想， golang 也不例外。本文通过对 io 库设计的分析，使用一些相关的包，如strings，bytes等，正确处理普通文件、内存、字符、管道等流式应用，以及常用文本和二进制流的读写，理解抽象设计的重要性。最后讲述编写 go 库编程的基本规范。

> 本文要点：
>
> 1. io 库的设计
> 2. 装饰模式（Decorator pattern）与 io 流处理
> 3. 规范的文本流与二进制流的读写
> 4. golang 库编程要点
>
> 前提条件：
>
> 1. 面向对象设计与编程基本概念
> 2. 了解 3-5 种面向对象设计模式

3. 了解 golang 接口值的含义

1、 io.go 源代码与 IO流抽象

阅读 io 库文档 真不如阅读 io.go 源代码，因为源代码是按作者思考逻辑组织的，而参考文档是按固定结构、字典顺序组织的。

1.1 io 包的职责

包设计的基本原则是职责内聚，包设计职责描述是判断设计合理性的重要方面。它通常可以用几句化描述，就 IO 包而言：

- 提供 io流操作原语（I/O primitives） 的基本接口。这些接口为实现 io 流处理的包，如 os 等包在实现流操作时，提供统一的抽象。
- 提供一些实用函数，方便各种IO流的处理
- 除非特别声明，这些原语实现不能假设为线程安全

1.2 IO 操作原语定义与基础接口

在源代码中，对于 IO 流，定义了四个基本操作原语，分别用 Reader，Writer，Closer，Seeker 接口表达二进制流读、写、关闭、寻址操作。 源代码：

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

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

type Closer interface {
        Close() error
}

type Seeker interface {
        Seek(offset int64, whence int) (int64, error)
}
```

然后，定义了一些原语组合，表达一些常用的流文件的处理能力。如只读、只写、可读写流。 源代码：

```
// ReadWriter is the interface that groups the basic Read and Write methods.
type ReadWriter interface {
        Reader
        Writer
}

type ReadCloser interface {...}
type WriteCloser interface {...}
type ReadWriteCloser interface {...}
```

```
type ReadSeeker interface {...}
type WriteSeeker interface {...}
type ReadWriteSeeker interface {...}
```

为了兼容以往流编程习惯，IO 库定义了一些常见的基本操作，例如，读写一个字符，seek 到特定位置读写，字串的读写等，源代码：

```
type ReaderAt interface {
        ReadAt(p []byte, off int64) (n int, err error)
}

type WriterAt interface {...}

type ByteReader interface {
        ReadByte() (byte, error)
}

type ByteWriter interface {...}

type RuneReader interface {
        ReadRune() (r rune, size int, err error)
}

// stringWriter is the interface that wraps the WriteString method.
type stringWriter interface {
        WriteString(s string) (n int, err error)
}
```

这里有一些细节：

- 为社么一些接口命名方法不统一，如：ReadAt?
- 内部接口 `stringWriter` 有什么用?
- 是否需要更多的定义，如 IntReader?

1.3 实用 IO 处理函数

在满足上述原语定义的流，就有许多常见的实用功能。提供这些函数不仅方便用户使用该 IO 库，同时也展现了抽象带来的应用价值。

例如：复制流（文件copy），源代码：

```
func Copy(dst Writer, src Reader) (written int64, err error) {
        return copyBuffer(dst, src, nil)
}

// copyBuffer is the actual implementation of Copy and CopyBuffer.
// if buf is nil, one is allocated.
func copyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error) {
        // If the reader has a WriteTo method, use it to do the copy.
        // Avoids an allocation and a copy.
        if wt, ok := src.(WriterTo); ok {
                return wt.WriteTo(dst)
        }
        // Similarly, if the writer has a ReadFrom method, use it to do the copy.
        if rt, ok := dst.(ReaderFrom); ok {
```

```
                return rt.ReadFrom(src)
        }
        if buf == nil {
                size := 32 * 1024
                if l, ok := src.(*LimitedReader); ok && int64(size) > l.N {
                        if l.N < 1 {
                                size = 1
                        } else {
                                size = int(l.N)
                        }
                }
                buf = make([]byte, size)
        }
        for {
                nr, er := src.Read(buf)
                if nr > 0 {
                        nw, ew := dst.Write(buf[0:nr])
                        if nw > 0 {
                                written += int64(nw)
                        }
                        if ew != nil {
                                err = ew
                                break
                        }
                        if nr != nw {
                                err = ErrShortWrite
                                break
                        }
                }
                if er != nil {
                        if er != EOF {
                                err = er
                        }
                        break
                }
        }
        return written, err
}
```

这段代码表明，默认的 Reader 输出到 Writer 采用 32K 缓冲读写。如果有些设备不支持 Buffer 如 Console Input，如果 scr 实现了 WriteTo，就直接写入 dst。

1.4 典型的应用结构

自己阅读 SectionReader， 了解 golang 对象的编程习惯。

- NewXXXX(...)
- 定义 XXX 结构
- 定义 XXX 的方法

2、IO 抽象与流处理应用

io 包是基础抽象库，它的目标是支持各种类型的流应用。因此，研究提供不同应用场景流服务的方法远比 分析 `io/ioutil` 子包有价值。本节参照 java io 库的思路，首先介绍装饰模式（Decorator），也称 Wrapper 在 IO 库设计中的应用，然后探讨 golang 文字流和二进制流处理的相关库。

2.1 装饰模式

装饰模式（Decorator pattern）是 IO 流处理中最典型。例如：

```
type rot13Reader struct {
        r io.Reader
}

func (rr rot13Reader) Read(p []byte) (n int, err error) {...}

func main() {
        sr := strings.NewReader("Lbh penpxrq gur pbqr!")
        rr := rot13Reader{sr}
        io.Copy(os.Stdout, &rr)
}
```

原来是这个熟悉的程序！正如官网所述

> 有种常见的模式是一个 io.Reader 包装另一个 io.Reader，然后通过某种方式修改其数据流。
>
> 例如，gzip.NewReader 函数接受一个 io.Reader（已压缩的数据流）并返回一个同样实现了 io.Reader 的 *gzip.Reader（解压后的数据流）。

类似的，对 Writer 也可以做包装。

2.2 字符流

strings 包

strings 包提供了 Reader，Builder 类型帮助我们用流的方式高效地读、构建文字流。

例如：将一个字符文件读入内存最方便的方法是 `io.Copy(sb, fr)`。

```
package main

import (
    "fmt"
    "strings"
    "io"
)

func main() {
    sr := strings.NewReader("Hello")
    sb := strings.Builder{}
    io.Copy(&sb,sr)
    io.WriteString(&sb,", 世界")
```

```
        fmt.Println(sr, &sb)
}
```

注意

- strings.Builder 需要 **golang 1.10 版本**以上，你可能需要升级 go 语言
- String 值是不可变的（**Immutable**）。即不能修改部分内容，如 `s[i] = '2'` 会引发 panic
- String 值是 utf-8 格式存储， len 返回是 bytes。 如：`s:="Go编程"; fmt.Println(len(s))` 输出为 8；
- 如果要编辑或处理非 ascii（127内）应将字串转为数组，如 `[]byte(s)` 或 `[]rune(s)`
    - 千万别以为切片指向原字串，而是指向处理原字串新申请的数组
    - 如果字串很长，转换代价是非常之大，请使用 Reader 是最有效的选择！
- io.WriteString 执行的代码是 `&sb.WriteString` 吗？

验证:

- go语言字符串拼接性能分析；原文
- 上述方法，没有使用流实现字串拼接，请测试之。注意：字串长度**可能**影响流的效率
    - builder.go 源代码似乎很关注效率

升级 go

```
$ sudo yum erase golang
$ wget https://dl.google.com/go/go1.11.linux-amd64.tar.gz
$ sudo tar zxvf go1.11.linux-amd64.tar.gz -C /usr/local
$ vi ~/.profile
export GOPATH=$HOME/gowork
export GOROOT=/usr/local/go
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
$ source ~/.profile
```

fmt 包

有了 Reader 和 Writer 抽象，我们就可以使用 FScan?(...) Fprint?(...) 格式化读写文件、内存块、字符串、网络文件等。例如：

```
package main

import "fmt"
import "strings"
import "io"

func main() {
        sr := strings.NewReader("Hello, 世界")
        sb := &strings.Builder{}
        io.Copy(sb,sr)
        fmt.Fprintln(sb)
        fmt.Fprintf(sb,"value is %v!\v", 42)
        fmt.Println(sb)
}
```

感觉是否很强大！ 其实 java 1.5 就这样玩了哦。

bytes 包

bytes 包和 strings 包有些类似，有许多函数帮助你处理文字。 如果把 []byte 看作内存块，该包提供的 Reader 和 Buffer。

- bytes.Reader 实现了 io.Reader, io.ReaderAt, io.WriterTo, io.Seeker, io.ByteScanner, and io.RuneScanner 接口
- bytes.Buffer 提供了 io.ReadWriter 等接口。 配合切片指针，用户可以方便读写内存块任意位置。

官方给的案例：

```
var b bytes.Buffer // A Buffer needs no initialization.
b.Write([]byte("Hello "))
fmt.Fprintf(&b, "world!")
b.WriteTo(os.Stdout)
```

os.File与bufio包

os.File 实现了文件的读写操作，支持几乎所有 io 原语。 除了用 os.OpenFile 打开文件后，一定要使用 defer 函数 close 文件外，其他 io 操作都和前面类似。

bufio包使用 Wrapper 包装原始的 Reader 和 Writer， 提供 buffered 服务、文字处理、字符流扫描服务：

- bufio.Reader，提供了原始文件 Reader 不能提供的文字处理的功能， 如： ReadLine，ReadRune，WriteTo 等操作
- bufio.Writer, 提供了原始文件 Writer 不能提供的文字处理的功能， 如： ReadFrom，WriteString，WriteRune 等操作
- bufio.Scanner，将字符流转为 Token 流
    - 请阅读官方案例 Example (Words) 和 Example (Custom)
    - 请阅读源代码 scan.go
    - 因为解析 XML 等文档都可以用类似的思想处理

你可能观察到 bufio.Reader 、bytes.Reader 与 strings.Reader 很多功能相同，如 ReadRune。 这时需要用 bufio.Reader 包装 strings.Reader 吗?

io，io/ioutil 包其他功能

- 必须关注的功能，io.Pipe() 提供了 **线程安全** 的管道服务
- 可以使用的功能，ioutil.TempFile 和 ioutil.TempDir， 毕竟你不打算在这些小功能上怼 google 工程师
- ioutil 其他功能，太鸡肋了。建议你不要使用 io/ioutil 子包，几乎没有存在的价值

2.3 二进制流

如何将 `[]int64{1,2,3,4,5}` 或 `User{“WuKong”, 3800, true}` 以二进制方式写入文件或读出?

嗯啊吖，难道就仅支持原始的 `[]byte` 读写吗？

encoding/binary 包

直接上代码：

```
package main

import "fmt"
import "bytes"
import "encoding/binary"

func main() {
        rw := &bytes.Buffer{}
        data := []int64{49,50,51,52}
        binary.Write(rw,binary.LittleEndian,data)
    fmt.Println(rw, rw.Len(), data)
    fmt.Printf("% x\n", rw.Bytes())
        size := binary.Size(data[0])
        fmt.Println(size)
        orgin := make([]int64,rw.Len()/size)
        binary.Read(rw,binary.LittleEndian,orgin)
        fmt.Println(orgin)
}
```

上述程序按官方说明 `Numbers are translated by reading and writing fixed-size values. A fixed-size value is either a fixed-size arithmetic type (bool, int8, uint8, int16, float32, complex64, ...) or an array or struct containing only fixed-size values.`

这意味结构中包含 string，[]byte 就搞不定了。 是的！ 正确的处理代码如下：

```
package main

import (
        "fmt"
        "bytes"
        "encoding/binary"
)

type User struct {
        name [20]byte
        age int16
        sex bool
}

func main() {
        man := User{age:21, sex: true}
        name := man.name[:]
        copy(name, "Jack")
        fmt.Println(man, binary.Size(man))

        rw := &bytes.Buffer{}
        binary.Write(rw,binary.LittleEndian,man)
        fmt.Printf("% x\n", rw.Bytes())

        orgin := User{}
```

```
        binary.Read(rw, binary.LittleEndian, &orgin.name)
        binary.Read(rw, binary.LittleEndian, &orgin.age)
        binary.Read(rw, binary.LittleEndian, &orgin.sex)
        fmt.Println(orgin)

        fmt.Println("Hello, ", string(name))
}
```

你可能有疑问，为什么不使用 `binary.Read(rw, binary.LittleEndian, &orgin)`？出错了， bug!

为社么没改正？因为高手都用 unsafe!

unsafe 包

如果你不考虑 intel CPU 和 一些 RISC CPU 字节序（Endian）不同，二进制文件读写就是 byte数组 与 其他类型 向 c 语言那样转换的问题。golang 提供了 `unsafe` 包，让你比较方便处理指针。

```
package main

import (
        "fmt"
        "bytes"
        "unsafe"
)

const RECORD_LEN = 23

type User struct {
        name [20]byte
        age int16
        sex bool
}

func main() {
        man := User{age:21, sex: true}
        name := man.name[:]
        copy(name, "Jack")
        fmt.Println(man)

        rw := &bytes.Buffer{}
        pBytes := (*[RECORD_LEN]byte)(unsafe.Pointer(&man))
        rw.Write(pBytes[:])
        fmt.Printf("% x\n", rw.Bytes())

        pUser := (*User)(unsafe.Pointer(&rw.Bytes()[0]))
        orgin := *pUser //copy!
        fmt.Println(orgin)

        fmt.Println("Hello, ", string(name))
}
```

这是处理二进制记录目前最直接的方法，缺点是要事前知道CPU兼容和记录长度。计算这个长度也不容易，你的知道结构的存储方式，即 packed or not！最后注意，理解垃圾回收机制工作模式，控制不安全指针的作用域。

encoding/gob 包

参考官网案例 Basic

在 encoding 子包中，人们提出基于各种格式的交换数据的方法。如果你只打算 golang 进程之间交换数据， gob 就是较好的选择， 否则就的使用 json， xml等编码和解码方法。

3、包的组织结构

在编程学习中，“依葫芦画瓢”大概是最有效的方法了。阅读完 io 包，你大概知道了写一个包的套路，让我们总结一下：

3.1 版本版权申明与包描述

- 写在与包名相同的 go 文件中。 例如 io 包 io.go
- 写在独立的 doc 文件中。例如 fmt 包 doc.go

3.2 包中的 go 文件

- go 文件一般成对出现，例如：`io.go` 与 `io_test.go` 以方便包实现自动化测试
- 每个 go 文件一定是功能内聚的，如流原语，就围绕流设计 api，而不宜设计文件、管道等
- example_test.go 文件， 文件中有 `func Example...` 函数，用以帮助生成 api 文件
- doc.go 复杂 api 的说明文件

3.3 常量与变量

go 程序的开始，一定是包用到的常量与变量，复杂的包可以用 `vars.go` 申明。 定义 errors 是包设计的重要内容之一。 例如 io.go：

```
// Seek whence values.
const (
        SeekStart   = 0 // seek relative to the origin of the file
        SeekCurrent = 1 // seek relative to the current offset
        SeekEnd     = 2 // seek relative to the end
)

// ErrShortWrite means that a write accepted fewer bytes than requested
// but failed to return an explicit error.
var ErrShortWrite = errors.New("short write")

// ErrShortBuffer means that a read required a longer buffer than was provided.
var ErrShortBuffer = errors.New("short buffer")
```

3.4 api 设计顺序

- 抽象事物操作原语接口
- 具体事物结构与方法实现
- 实用辅助函数

编程不是字典顺序的，要以程序猿逻辑组织代码，别于大家阅读！

4、小结

从具体到抽象，在从抽象到具体应用，是人类认识世界的基本方法，程序设计也是如此！如果你理解了 io 的 Reader 和 Writer，文字、内存块、文件等的处理就会得心应手，也不需要记忆那么多烦人的 API 了。例如，遇到复杂的 split ， 你就想到 Scanner 的设计， “依葫芦画瓢” 做一个绝不会比一般人编写的 api 差的。

api 原代码阅读的重要性不用多说，这是 go 进阶的唯一方法。