中级篇：35-50

35. 关闭 HTTP 的响应体

使用 HTTP 标准库发起请求、获取响应时，即使你不从响应中读取任何数据或响应为空，都需要手动关闭响应体。新手很容易忘记手动关闭，或者写在了错误的位置：

```
// 请求失败造成 panic
func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    defer resp.Body.Close()    // resp 可能为 nil，不能读取 Body
    if err != nil {
        fmt.Println(err)
        return
    }

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(string(body))
}

func checkError(err error) {
    if err != nil{
        log.Fatalln(err)
    }
}
```

上边的代码能正确发起请求，但是一旦请求失败，变量 resp 值为 nil，造成 panic：

> panic: runtime error: invalid memory address or nil pointer dereference

应该先检查 HTTP 响应错误为 nil，再调用 resp.Body.Close() 来关闭响应体：

```
// 大多数情况正确的示例
func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    checkError(err)

    defer resp.Body.Close()    // 绝大多数情况下的正确关闭方式
    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(string(body))
}
```

输出：

```
Get https://api.ipify.org?format=...: x509: certificate signed by unknown
authority
```

绝大多数请求失败的情况下，resp 的值为 nil 且 err 为 non-nil。但如果你得到的是重定向错误，那它俩的值都是 non-nil，最后依旧可能发生内存泄露。2 个解决办法：

- 可以直接在处理 HTTP 响应错误的代码块中，直接关闭非 nil 的响应体。
- 手动调用 defer 来关闭响应体：

```
// 正确示例
func main() {
    resp, err := http.Get("http://www.baidu.com")

    // 关闭 resp.Body 的正确姿势
    if resp != nil {
        defer resp.Body.Close()
    }

    checkError(err)
    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(string(body))
}
```

resp.Body.Close() 早先版本的实现是读取响应体的数据之后丢弃，保证了 keep-alive 的 HTTP 连接能重用处理不止一个请求。但 Go 的最新版本将读取并丢弃数据的任务交给了用户，如果你不处理，HTTP 连接可能会直接关闭而非重用，参考在 Go 1.5 版本文档。

如果程序大量重用 HTTP 长连接，你可能要在处理响应的逻辑代码中加入：

```
_, err = io.Copy(ioutil.Discard, resp.Body)     // 手动丢弃读取完毕的数据
```

如果你需要完整读取响应，上边的代码是需要写的。比如在解码 API 的 JSON 响应数据：

```
json.NewDecoder(resp.Body).Decode(&data)
```

36. 关闭 HTTP 连接

一些支持 HTTP1.1 或 HTTP1.0 配置了 connection: keep-alive 选项的服务器会保持一段时间的长连接。但标准库 "net/http" 的连接默认只在服务器主动要求关闭时才断开，所以你的程序可能会消耗完 socket 描述符。解决办法有 2 个，请求结束后：

- 直接设置请求变量的 Close 字段值为 true，每次请求结束后就会主动关闭连接。
- 设置 Header 请求头部选项 Connection: close，然后服务器返回的响应头部也会有这个选项，此时 HTTP 标准库会主动断开连接。

```
// 主动关闭连接
func main() {
    req, err := http.NewRequest("GET", "http://golang.org", nil)
    checkError(err)

    req.Close = true
    //req.Header.Add("Connection", "close")    // 等效的关闭方式

    resp, err := http.DefaultClient.Do(req)
    if resp != nil {
        defer resp.Body.Close()
    }
    checkError(err)

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(string(body))
}
```

你可以创建一个自定义配置的 HTTP transport 客户端，用来取消 HTTP 全局的复用连接：

```
func main() {
    tr := http.Transport{DisableKeepAlives: true}
    client := http.Client{Transport: &tr}

    resp, err := client.Get("https://golang.google.cn/")
    if resp != nil {
        defer resp.Body.Close()
    }
    checkError(err)

    fmt.Println(resp.StatusCode)    // 200

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(len(string(body)))
}
```

根据需求选择使用场景：

- 若你的程序要向同一服务器发大量请求，使用默认的保持长连接。
- 若你的程序要连接大量的服务器，且每台服务器只请求一两次，那收到请求后直接关闭连接。或增加最大文件打开数 fs.file-max 的值。

37. 将 JSON 中的数字解码为 interface 类型

在 encode/decode JSON 数据时，Go 默认会将数值当做 float64 处理，比如下边的代码会造成 panic：

```
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    if err := json.Unmarshal(data, &result); err != nil {
        log.Fatalln(err)
    }

    fmt.Printf("%T\n", result["status"])    // float64
    var status = result["status"].(int)    // 类型断言错误
    fmt.Println("Status value: ", status)
}
```

> panic: interface conversion: interface {} is float64, not int

如果你尝试 decode 的 JSON 字段是整型，你可以：

- 将 int 值转为 float 统一使用
- 将 decode 后需要的 float 值转为 int 使用

```
// 将 decode 的值转为 int 使用
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    if err := json.Unmarshal(data, &result); err != nil {
        log.Fatalln(err)
    }

    var status = uint64(result["status"].(float64))
    fmt.Println("Status value: ", status)
}
```

- 使用 Decoder 类型来 decode JSON 数据，明确表示字段的值类型

```
// 指定字段类型
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    var decoder = json.NewDecoder(bytes.NewReader(data))
    decoder.UseNumber()

    if err := decoder.Decode(&result); err != nil {
```

```
        log.Fatalln(err)
    }

    var status, _ = result["status"].(json.Number).Int64()
    fmt.Println("Status value: ", status)
}

 // 你可以使用 string 来存储数值数据，在 decode 时再决定按 int 还是 float 使用
 // 将数据转为 decode 为 string
 func main() {
     var data = []byte({"status": 200})
      var result map[string]interface{}
      var decoder = json.NewDecoder(bytes.NewReader(data))
      decoder.UseNumber()
      if err := decoder.Decode(&result); err != nil {
          log.Fatalln(err)
      }
    var status uint64
      err := json.Unmarshal([]byte(result["status"].(json.Number).String()),
&status);
    checkError(err)
       fmt.Println("Status value: ", status)
}
```

- 使用 struct 类型将你需要的数据映射为数值型

```
// struct 中指定字段类型
func main() {
      var data = []byte(`{"status": 200}`)
      var result struct {
          Status uint64 `json:"status"`
      }

      err := json.NewDecoder(bytes.NewReader(data)).Decode(&result)
      checkError(err)
    fmt.Printf("Result: %+v", result)
}
```

- 可以使用 struct 将数值类型映射为 json.RawMessage 原生数据类型 适用于如果 JSON 数据不着急 decode 或 JSON 某个字段的值类型不固定等情况：

```
// 状态名称可能是 int 也可能是 string，指定为 json.RawMessage 类型
func main() {
    records := [][]byte{
        []byte(`{"status":200, "tag":"one"}`),
        []byte(`{"status":"ok", "tag":"two"}`),
    }

    for idx, record := range records {
```

```
        var result struct {
            StatusCode uint64
            StatusName string
            Status     json.RawMessage `json:"status"`
            Tag        string          `json:"tag"`
        }

        err := json.NewDecoder(bytes.NewReader(record)).Decode(&result)
        checkError(err)

        var name string
        err = json.Unmarshal(result.Status, &name)
        if err == nil {
            result.StatusName = name
        }

        var code uint64
        err = json.Unmarshal(result.Status, &code)
        if err == nil {
            result.StatusCode = code
        }

        fmt.Printf("[%v] result => %+v\n", idx, result)
    }
}
```

38. struct、array、slice 和 map 的值比较

可以使用相等运算符 == 来比较结构体变量，前提是两个结构体的成员都是可比较的类型：

```
type data struct {
    num     int
    fp      float32
    complex complex64
    str     string
    char    rune
    yes     bool
    events  <-chan string
    handler interface{}
    ref     *byte
    raw     [10]byte
}

func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2: ", v1 == v2)    // true
}
```

如果两个结构体中有任意成员是不可比较的，将会造成编译错误。注意数组成员只有在数组元素可比较时候才可比较。

```
type data struct {
    num    int
    checks [10]func() bool        // 无法比较
    doIt   func() bool          // 无法比较
    m      map[string]string     // 无法比较
    bytes  []byte              // 无法比较
}

func main() {
    v1 := data{}
    v2 := data{}

    fmt.Println("v1 == v2: ", v1 == v2)
}
```

> invalid operation: v1 == v2 (struct containing [10]func() bool cannot be compared) Go 提供了一些库函数来比较那些无法使用 == 比较的变量，比如使用 "reflect" 包的 DeepEqual()：

```
// 比较相等运算符无法比较的元素
func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))    // true

    m1 := map[string]string{"one": "a", "two": "b"}
    m2 := map[string]string{"two": "b", "one": "a"}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(m1, m2))    // true

    s1 := []int{1, 2, 3}
    s2 := []int{1, 2, 3}
       // 注意两个 slice 相等，值和顺序必须一致
    fmt.Println("v1 == v2: ", reflect.DeepEqual(s1, s2))    // true
}
```

这种比较方式可能比较慢，根据你的程序需求来使用。DeepEqual() 还有其他用法：

```
func main() {
    var b1 []byte = nil
    b2 := []byte{}
    fmt.Println("b1 == b2: ", reflect.DeepEqual(b1, b2))    // false
}
```

注意：

DeepEqual() 并不总适合于比较 slice

```
func main() {
    var str = "one"
    var in interface{} = "one"
    fmt.Println("str == in: ", reflect.DeepEqual(str, in))    // true

    v1 := []string{"one", "two"}
    v2 := []string{"two", "one"}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))    // false

    data := map[string]interface{}{
        "code":  200,
        "value": []string{"one", "two"},
    }
    encoded, _ := json.Marshal(data)
    var decoded map[string]interface{}
    json.Unmarshal(encoded, &decoded)
    fmt.Println("data == decoded: ", reflect.DeepEqual(data, decoded))    // false
}
```

如果要大小写不敏感来比较 byte 或 string 中的英文文本，可以使用 "bytes" 或 "strings" 包的 ToUpper() 和 ToLower() 函数。比较其他语言的 byte 或 string，应使用 bytes.EqualFold() 和 strings.EqualFold()

如果 byte slice 中含有验证用户身份的数据（密文哈希、token 等），不应再使用 reflect.DeepEqual()、bytes.Equal()、 bytes.Compare()。这三个函数容易对程序造成 timing attacks，此时应使用 "crypto/subtle" 包中的 subtle.ConstantTimeCompare() 等函数

reflect.DeepEqual() 认为空 slice 与 nil slice 并不相等，但注意 byte.Equal() 会认为二者相等：

```
func main() {
    var b1 []byte = nil
    b2 := []byte{}

    // b1 与 b2 长度相等、有相同的字节序
    // nil 与 slice 在字节上是相同的
    fmt.Println("b1 == b2: ", bytes.Equal(b1, b2))    // true
}
```

39. 从 panic 中恢复

在一个 defer 延迟执行的函数中调用 recover()，它便能捕捉 / 中断 panic

```
// 错误的 recover 调用示例
func main() {
    recover()    // 什么都不会捕捉
    panic("not good")    // 发生 panic，主程序退出
    recover()    // 不会被执行
```

```
    println("ok")
}

// 正确的 recover 调用示例
func main() {
    defer func() {
        fmt.Println("recovered: ", recover())
    }()
    panic("not good")
}
```

从上边可以看出，recover() 仅在 defer 执行的函数中调用才会生效。

```
// 错误的调用示例
func main() {
    defer func() {
        doRecover()
    }()
    panic("not good")
}

func doRecover() {
    fmt.Println("recobered: ", recover())
}
```

> recobered: panic: not good

40. 在 range 迭代 slice、array、map 时通过更新引用来更新元素

在 range 迭代中，得到的值其实是元素的一份值拷贝，更新拷贝并不会更改原来的元素，即是拷贝的地址并不是原有元素的地址：

```
func main() {
    data := []int{1, 2, 3}
    for _, v := range data {
        v *= 10         // data 中原有元素是不会被修改的
    }
    fmt.Println("data: ", data)    // data:  [1 2 3]
}
```

如果要修改原有元素的值，应该使用索引直接访问：

```
func main() {
    data := []int{1, 2, 3}
    for i, v := range data {
        data[i] = v * 10
    }
```

```
    fmt.Println("data: ", data)    // data:  [10 20 30]
}
```

如果你的集合保存的是指向值的指针，需稍作修改。依旧需要使用索引访问元素，不过可以使用 range 出来的元素直接更新原有值：

```
func main() {
    data := []*struct{ num int }{{1}, {2}, {3},}
    for _, v := range data {
        v.num *= 10    // 直接使用指针更新
    }
    fmt.Println(data[0], data[1], data[2])    // &{10} &{20} &{30}
}
```

41. slice 中隐藏的数据

从 slice 中重新切出新 slice 时，新 slice 会引用原 slice 的底层数组。如果跳了这个坑，程序可能会分配大量的临时 slice 来指向原底层数组的部分数据，将导致难以预料的内存使用。

```
func get() []byte {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    return raw[:3]    // 重新分配容量为 10000 的 slice
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 10000 0xc420080000
}
```

可以通过拷贝临时 slice 的数据，而不是重新切片来解决：

```
func get() (res []byte) {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    res = make([]byte, 3)
    copy(res, raw[:3])
    return
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 3 0xc4200160b8
}
```

42. Slice 中数据的误用

举个简单例子，重写文件路径（存储在 slice 中）

分割路径来指向每个不同级的目录，修改第一个目录名再重组子目录名，创建新路径：

```
// 错误使用 slice 的拼接示例
func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    println(sepIndex)

    dir1 := path[:sepIndex]
    dir2 := path[sepIndex+1:]
    println("dir1: ", string(dir1))        // AAAA
    println("dir2: ", string(dir2))        // BBBBBBBBB

    dir1 = append(dir1, "suffix"...)
       println("current path: ", string(path))    // AAAAsuffixBBBB

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // uffixBBBB

    println("new path: ", string(path))    // AAAAsuffix/uffixBBBB    // 错误结果
}
```

拼接的结果不是正确的 AAAAsuffix/BBBBBBBBB，因为 dir1、 dir2 两个 slice 引用的数据都是 path 的底层数组，第 13 行修改 dir1 同时也修改了 path，也导致了 dir2 的修改

解决方法：

- 重新分配新的 slice 并拷贝你需要的数据
- 使用完整的 slice 表达式：input[low:high:max]，容量便调整为 max - low

```
// 使用 full slice expression
func main() {

    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    dir1 := path[:sepIndex:sepIndex]        // 此时 cap(dir1) 指定为4， 而不是先前的
16
    dir2 := path[sepIndex+1:]
    dir1 = append(dir1, "suffix"...)

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // BBBBBBBBB
    println("new path: ", string(path))    // AAAAsuffix/BBBBBBBBB
}
```

第 6 行中第三个参数是用来控制 dir1 的新容量，再往 dir1 中 append 超额元素时，将分配新的 buffer 来保存。而不是覆盖原来的 path 底层数组

43. 旧 slice

当你从一个已存在的 slice 创建新 slice 时，二者的数据指向相同的底层数组。如果你的程序使用这个特性，那需要注意 "旧"（stale） slice 问题。

某些情况下，向一个 slice 中追加元素而它指向的底层数组容量不足时，将会重新分配一个新数组来存储数据。而其他 slice 还指向原来的旧底层数组。

```
// 超过容量将重新分配数组来拷贝值、重新存储
func main() {
    s1 := []int{1, 2, 3}
    fmt.Println(len(s1), cap(s1), s1)    // 3 3 [1 2 3 ]

    s2 := s1[1:]
    fmt.Println(len(s2), cap(s2), s2)    // 2 2 [2 3]

    for i := range s2 {
        s2[i] += 20
    }
    // 此时的 s1 与 s2 是指向同一个底层数组的
    fmt.Println(s1)        // [1 22 23]
    fmt.Println(s2)        // [22 23]

    s2 = append(s2, 4)    // 向容量为 2 的 s2 中再追加元素，此时将分配新数组来存

    for i := range s2 {
        s2[i] += 10
    }
    fmt.Println(s1)        // [1 22 23]    // 此时的 s1 不再更新，为旧数据
    fmt.Println(s2)        // [32 33 14]
}
```

44. 类型声明与方法

从一个现有的非 interface 类型创建新类型时，并不会继承原有的方法：

```
// 定义 Mutex 的自定义类型
type myMutex sync.Mutex

func main() {
    var mtx myMutex
    mtx.Lock()
    mtx.UnLock()
}
```

> mtx.Lock undefined (type myMutex has no field or method Lock)... 如果你需要使用原类型的方法，可将原类型以匿名字段的形式嵌到你定义的新 struct 中：

```
// 类型以字段形式直接嵌入
type myLocker struct {
    sync.Mutex
}

func main() {
    var locker myLocker
    locker.Lock()
    locker.Unlock()
}
```

interface 类型声明也保留它的方法集：

```
type myLocker sync.Locker

func main() {
    var locker myLocker
    locker.Lock()
    locker.Unlock()
}
```

45. 跳出 for-switch 和 for-select 代码块

没有指定标签的 break 只会跳出 switch/select 语句，若不能使用 return 语句跳出的话，可为 break 跳出标签指定的代码块：

```
// break 配合 label 跳出指定代码块
func main() {
loop:
    for {
        switch {
        case true:
            fmt.Println("breaking out...")
            //break    // 死循环，一直打印 breaking out...
            break loop
        }
    }
    fmt.Println("out...")
}
```

goto 虽然也能跳转到指定位置，但依旧会再次进入 for-switch，死循环。

46. for 语句中的迭代变量与闭包函数

for 语句中的迭代变量在每次迭代中都会重用，即 for 中创建的闭包函数接收到的参数始终是同一个变量，在 goroutine 开始执行时都会得到同一个迭代值：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        go func() {
            fmt.Println(v)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 three three three
}
```

最简单的解决方法：无需修改 goroutine 函数，在 for 内部使用局部变量保存迭代值，再传参：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        vCopy := v
        go func() {
            fmt.Println(vCopy)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

另一个解决方法：直接将当前的迭代值以参数形式传递给匿名函数：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        go func(in string) {
            fmt.Println(in)
        }(v)
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

注意下边这个稍复杂的 3 个示例区别：

```
type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

// 错误示例
func main() {
    data := []field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 three three three
}


// 正确示例
func main() {
    data := []field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {
        v := v
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 one two three
}

// 正确示例
func main() {
    data := []*field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {    // 此时迭代值 v 是三个元素值的地址，每次 v 指向的值不
同
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

47. defer 函数的参数值

对 defer 延迟执行的函数，它的参数会在声明时候就会求出具体值，而不是在执行时才求值：

```
// 在 defer 函数中参数会提前求值
func main() {
    var i = 1
```

```
    defer fmt.Println("result: ", func() int { return i * 2 }())
    i++
}
```

```
result: 2
```

48. defer 函数的执行时机

对 defer 延迟执行的函数，会在调用它的函数结束时执行，而不是在调用它的语句块结束时执行，注意区分开。

比如在一个长时间执行的函数里，内部 for 循环中使用 defer 来清理每次迭代产生的资源调用，就会出现问题：

```
// 命令行参数指定目录名
// 遍历读取目录下的文件
func main() {

    if len(os.Args) != 2 {
        os.Exit(1)
    }

    dir := os.Args[1]
    start, err := os.Stat(dir)
    if err != nil || !start.IsDir() {
        os.Exit(2)
    }

    var targets []string
    filepath.Walk(dir, func(fPath string, fInfo os.FileInfo, err error) error {
        if err != nil {
            return err
        }

        if !fInfo.Mode().IsRegular() {
            return nil
        }

        targets = append(targets, fPath)
        return nil
    })

    for _, target := range targets {
        f, err := os.Open(target)
        if err != nil {
            fmt.Println("bad target:", target, "error:", err)    //error:too many
open files
            break
        }
        defer f.Close()    // 在每次 for 语句块结束时，不会关闭文件资源
```

```
        // 使用 f 资源
    }
}
```

先创建 10000 个文件：

```
#!/bin/bash
for n in {1..10000}; do
    echo content > "file${n}.txt"
done
```

运行效果：

```

```

解决办法：defer 延迟执行的函数写入匿名函数中：

```
// 目录遍历正常
func main() {
    // ...

    for _, target := range targets {
        func() {
            f, err := os.Open(target)
            if err != nil {
                fmt.Println("bad target:", target, "error:", err)
                return    // 在匿名函数内使用 return 代替 break 即可
            }
            defer f.Close()    // 匿名函数执行结束，调用关闭文件资源

            // 使用 f 资源
        }()
    }
}
```

当然你也可以去掉 defer，在文件资源使用完毕后，直接调用 f.Close() 来关闭。

49. 失败的类型断言

在类型断言语句中，断言失败则会返回目标类型的“零值”，断言变量与原来变量混用可能出现异常情况：

```
// 错误示例
func main() {
    var data interface{} = "great"

    // data 混用
```

```
    if data, ok := data.(int); ok {
        fmt.Println("[is an int], data: ", data)
    } else {
        fmt.Println("[not an int], data: ", data)    // [isn't a int], data:  0
    }
}


// 正确示例
func main() {
    var data interface{} = "great"

    if res, ok := data.(int); ok {
        fmt.Println("[is an int], data: ", res)
    } else {
        fmt.Println("[not an int], data: ", data)    // [not an int], data:  great
    }
}
```

50. 阻塞的 gorutinue 与资源泄露

在 2012 年 Google I/O 大会上，Rob Pike 的 Go Concurrency Patterns 演讲讨论 Go 的几种基本并发模式，如完整代码 中从数据集中获取第一条数据的函数：

```
func First(query string, replicas []Search) Result {
    c := make(chan Result)
    replicaSearch := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go replicaSearch(i)
    }
    return <-c
}
```

在搜索重复时依旧每次都起一个 goroutine 去处理，每个 goroutine 都把它的搜索结果发送到结果 channel 中，channel 中收到的第一条数据会直接返回。

返回完第一条数据后，其他 goroutine 的搜索结果怎么处理？他们自己的协程如何处理？

在 First() 中的结果 channel 是无缓冲的，这意味着只有第一个 goroutine 能返回，由于没有 receiver，其他的 goroutine 会在发送上一直阻塞。如果你大量调用，则可能造成资源泄露。

为避免泄露，你应该确保所有的 goroutine 都能正确退出，有 2 个解决方法：

- 使用带缓冲的 channel，确保能接收全部 goroutine 的返回结果：

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,len(replicas))
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
```

```
        go searchReplica(i)
    }
    return <-c
}
```

- 使用 select 语句，配合能保存一个缓冲值的 channel default 语句： default 的缓冲 channel 保证了即使结果 channel 收不到数据，也不会阻塞 goroutine

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,1)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        default:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

- 使用特殊的废弃（cancellation） channel 来中断剩余 goroutine 的执行：

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    done := make(chan struct{})
    defer close(done)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        case <- done:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }

    return <-c
}
```

Rob Pike 为了简化演示，没有提及演讲代码中存在的这些问题。不过对于新手来说，可能会不加思考直接使用。