Anti-pattern in Go

by

Triton Ho 

前言

- 對新手而言：
  - App. Server 有無限 memory

大綱

- Slice 的 make
- Anti pattern: Slice as buffer
- Anti pattern: Map as cache
- Anti pattern: Abuse of ioutil.ReadAll
- Anti pattern: Use channel to pass signal
- Anti pattern: Abuse of go-routine

熱身： Slice 的 make

- https://blog.golang.org/slices-intro
  - To increase the capacity of a slice one must create a new, larger slice and copy the contents of the original slice into it.
- 意思是說，當你在跑 sliceOfX = append(sliceOfX, x)
  而你的 slice 底層 array 已經沒有足夠空間，那麼就會建立 2x size 的新 array ，把本來 array 的 data 抄過去

make 的 capacity

- 有人覺得：
  - append 會引起 data copy ，所以會效能不好
    凡事應該先用 make
- 例子：

```
sliceOfX := make([]X, 0, 1000)
sliceOfX = append(sliceOfX, x)
```

Slice Big-O 解說

- 當你在 append 時，（如果不夠空間）新的 array 是 2x 的，所以
  - 你要 append N 個 item 到空的 Slice 時，最多只會觸發 log N 次 array copy
  - 這最多只會有 2N 個 item copy
  - = O(N) runtime
- 除非你明確知道這 slice 的最大上限，否則別設定 capacity
- Optimization 重點：別去 optimize for linear runtime improvement
  - 除非你在 FAANG
- 常見寫法：
  - sliceOfX := []X{}
  - sliceOfX := make([]X, 0)

Anti pattern: Slice as buffer

- Producer
  - bufferOfX = append(bufferOfX, x)
- Consumer
  - x := bufferOfX[0]
  - bufferOfX = bufferOfX[1:]

問題在？

- bufferOfX = bufferOfX[1:]
  - 這句不會把 array 最前方的 memory 做 release
  - 這單純改動 slice 內對其 array 的 startPos pointer
- 要等到 append 而最後方空間不足，要 copy 到新的 array 時，沒用的 memory 才會 release
- 真正問題是：
  像 []string, []someObj 這類的 slice ，其 item 需要等很長時間才能被 garbage collection

Anti pattern: Map as cache

- 有人會用 map[string]string 用來作為 local-caching
- 問題是：
  - delete(someMap, key) 是單純 mark as deleted
  - 直到整個 map 被 garbage collection ，你的 map 其內的 key-values 都沒法被 garbage collection
- 如果你有一個無限生命期的 map ，然後你不停地 add / delete items ，最終這個 map 會吃光你的 memory

Anti pattern: Abuse of ioutil.ReadAll

- 邪惡例子：從 API 拿回 json data
  - res, err := http.Get("https://abc.com/endpoint1")
    if err != nil {
        log.Fatal(err)
    }
    defer res.Body.Close()

    responseBody, err1 := ioutil.ReadAll(res.Body)
    obj := SomeObject{}
    err2 := json.Unmarshal(responseBody, &obj)
- 以上是先把 responsebody 都放到 local memory ，然後再變回 obj
  - 如果 responsebody 有 100MB 呢？
  - 如果有 100 個 thread 來做相似的事呢？

正解

- obj := SomeObject{}
  decoder := json.NewDecoder(res.Body)
  err := decoder.Decode(&obj)

io.Reader 和 io.Writer

- 在 C 的世界， IO operation 你常常會用上 file pointer
  - 別忘記 stdin, stdout 和 stderr 都是 file pointer
- 在 Go 的世界，你應該善用 io.Reader 和 io.Writer
- res.Body 本身是一個 Reader
  - 意思是：他有 Read(b []byte) (n int, err error)
- json.NewDecoder(r io.Reader)
  - 意思是：他需要一個 data source 來拿 data 作 decoding
- 剛才寫法，不需要把整個 input 都存到 local memory 才能做後續工作，不會引發 OOM 問題
  - 如果一個 data input 是 10MB ，你同在應付 100 個 Request ，那就是 1GB memory

延伸一句：

- 很多 orm，都有 GetAll()，讓你把 resultSet 全先放到一個 slice 內
- 如果 resultSet 很大時，壞主意：
  - resultSet := []ObjX{}
    db.Query(`select……`).GetAll(&objX)
    for _, objX := range resultSet {
        outputToCSV(objX)
    }
- 好主意：
  - fn := func (obj interface{}) {
        objX := obj.(ObjX)
        outputToCSV(objX)
    }
    db.Query(`select……`).Iterate(&ObjX{}, fn)

Anti pattern: Use channel to pass signal

- Bad code
  - https://play.golang.org/p/sYkDgBHkJXN

```
11 func main() {
12         ticker := time.NewTicker(time.Second)
13         defer ticker.Stop()
14 
15         // capture the Ctrl-C os signal
16         signalChan := make(chan os.Signal, 1)
17         signal.Notify(signalChan, syscall.SIGINT)
18 
19         for {
20                 select {
21                 case <-signalChan:
22                         fmt.Println("Ctrl-C received")
23                         return
24                 case t := <-ticker.C:
25                         fmt.Println("Current time: ", t)
26                         // let's do some task here
27                 }
28         }
29 }
```

Bad Code 解說

- 有一個工作，想每秒都執行一次
  - 所以我用上 time.NewTicker(time.Second)
- 我不想工作跑到一半時被中斷，所以我需要接收 os 的 SIGINT
  讓我收到 signal 後，做完現在的事務，把東西都存起來後才和平結束

badCode 問題

- 以 channel 來傳訊號，每一個 thread 都需要消耗在 channel 內的 item
  - 如果現在有 N 個 thread ，而你不知道現在 N 的數字是多少，怎麼辦？
    （你可以知道開了多少個 goroutine ，但你有可能沒法知道還有多少 goroutine 還未 exit 的）
- 剛才 select 的寫法，如果二個 case 都有滿足時， go 是會 randomly 去跑其中一個
  - 如果你的工作跑得超過 1 秒時（你可以自己加上 time.Sleep 試試看）
    下一次 go 有機會還是會進 case t := <- ticker.C 這一部份
- 有人會說：沒什麼所謂，最大不了是我的工作多跑數次囉 ~
  - 問題 1 ：現在 SIGINT 後，你的程式會 randomly 多執行 0 到無限次，那你的 CICD script 要等待 graceful shutdown 多少秒？
  - 問題 2 ：你的程式現在 in average 多跑一次（ 0.5 + 0.25 + 0.125…… ），那麼你 rolling deployment 所用時間肯定長了的

正解 1

- 如果是以 channel 來接受工作的 goroutine
  單純把 taskCh 關掉就好，讓 goroutine 們自行離開 for-loop
  - goroutine
    ```
    For task := range taskCh {
        // your work here
    }
    ```
  - Main thread
    ```
    close(taskCh)
    ```

正解 2

- 如果你的 goroutine 沒 taskCh 來接受 input
  - 用一個 atomicBool 來 broadcast 就好
    https://play.golang.org/p/MuU-AtCFO7C

```
func main() {
        ticker := time.NewTicker(time.Second)
        defer ticker.Stop()

        // capture the Ctrl-C os signal
        signalChan := make(chan os.Signal, 1)
        signal.Notify(signalChan, syscall.SIGINT)

        // a variable for boardcasting the end-signal to all thread
        var isEnd atomicBool
        isEnd.setFalse()

        go func() {
                // we wait for the Ctrl-C signal here
                <-signalChan
                isEnd.setTrue()
        }()

        for {
                if isEnd.isTrue() {
                        fmt.Println("Ctrl-C received")
                        break
                }

                t := <-ticker.C
                fmt.Println("Current time: ", t)
                // let's do some task here

        }
}
```

Anti pattern: Abuse of go-routine

- 範例： qsort_bad.go

```
1  func qsortBadInternal(input []int, wg *sync.WaitGroup) {
2          // for demo the effect of go scheduler
3          time.Sleep(1 * time.Nanosecond)
4
5          defer wg.Done()
6
7          // sentinal
8          if len(input) <= 1 {
9                  return
10         }
11
12         pivotPos := qsortPartition(input)
13
14         wg.Add(2)
15         go qsortBadInternal(input[:pivotPos], wg)
16         go qsortBadInternal(input[pivotPos+1:], wg)
17 }
```

解說

- 在 quicksort 中，以 pivot 跑完了 partitioning 後把前半的 partition 和後半的 partition 各自以 goroutine 來跑
- 以 sync.WaitGroup 來存起現在還未結束的 goroutine 數量
  - 當 wg 內的數字為 0 ，代表所有 subtask 都做好了，qsort 跑完了

問題

- 你現在 goroutine 的數量跟 len(input) 相關
  - 雖然 goroutine 是 green thread ，不會引起 OS context switching 問題
  - 但是，每一個 goroutine 需要數 KB local memory
- 最大問題點：
  - 現在沒有任何機制，去控制 producer 把工作以 goroutine 型態丟進 local memory 的速度
  - 如果 consumer 速度變慢，這些還沒跑完的 goroutine 會在 local memory 越來越多，直到程式 OOM
    - 特別是你的 consumer 要經過 network 接上第三方時.....

Actor Model

- 這不是 Go 發明的，而是傳統 Concurrency 手法
- Actor 就是從 inbox 拿出「工作」來做，完成後把結果丟到 outbox
- 理想上，每一件「工作」都是獨立的
  - 不需要跟別的工作 share memory ，所以不會發生 race condition
  - 在 multi-thread system ，你可以替同一種類的工作建立多個 actor（就跟銀行有多個服務台一樣）
  - 沒工作時， actor 單純是 blocking 狀態，不佔 CPU

Actor Model 好處

- 在 actor 之間流動的只有「工作」本身
- 你可以自己決定系統中有多少 thread 在跑
  - 別忘記一堆 language 不是用 green thread 的
- Scalibility 遠遠比較好
  - 在 multi-machine 環境，你只需要把 go channel 改成像 MPI 之類的 library 就行，你大部份程式碼都不用改
- 很多時，你能迴避使用 mutex
  - locking 的 bug 很臭，也很難用 testcase 抓出來

常見寫法

- for task := range taskCh {
  // your own logic
  }
  waitGroup.Done()
- 重點是： taskCh 是預先建好的 buffer channel
  只會佔用固定 memory
- 如果 consumer 跑慢了， buffer channel 滿掉後
  producor 會被卡住
  - 比起你系統 OOM 崩潰，你的系統跑慢一點不是壞事

正解

- 範例： qsort_good.go

```
1  // WARNING: this qsortGood is for demo only, not for production usage.
2  // The actual performance of qsortGood is MUCH worse than the standard library
3  func qsortGood(input []int) {
4          wg := sync.WaitGroup{}
5          remainingTaskNum := sync.WaitGroup{}
6
7          threadNum := runtime.NumCPU() * 2
8          inputCh := make(chan []int, len(input)/2+1)
9          wg.Add(threadNum)
10         for i := 0; i < threadNum; i++ {
11                 go qsortGoodWorker(inputCh, &wg, &remainingTaskNum)
12         }
13
14         // add the input to channel, and wait for all subtask completed
15         remainingTaskNum.Add(1)
16         inputCh <- input
17         remainingTaskNum.Wait()
18
19         // let worker thread die peacefully, we SHOULD NOT leave the worker thread behind
20         close(inputCh)
21         wg.Wait()
22 }
```

Main thread 解說

- Main thread 自己不做工作的，他只是管理者
- Main thread 做的事
  - 建立 taskCh ，然後把 taskCh 作為N個 worker thread 的 inbox
  - 把第一份工作丟到 channel 內，然後等待 remainingTask 歸 0
  - 關掉 taskCh ，讓 workerThread 自然死亡
  - 等待所有 workerthread 死亡後，才返回結果

Worker Thread

```
4  func qsortGoodWorker(inputCh chan []int, wg *sync.WaitGroup, remainingTaskNum *sync.WaitGroup) {
5          defer wg.Done()
6
7          for input := range inputCh {
8                  // for demo the effect of go scheduler
9                  time.Sleep(1 * time.Nanosecond)
10
11                 // end condition of recursion
12                 if len(input) <= 1 {
13                         remainingTaskNum.Done()
14                         continue
15                 }
16
17                 pivotPos := qsortPartition(input)
18
19                 // add the sub-tasks to the queue
20                 remainingTaskNum.Add(2)
21                 inputCh <- input[:pivotPos]
22                 inputCh <- input[pivotPos+1:]
23
24                 // mark the current task is done
25                 remainingTaskNum.Done()
26         }
27 }
```

Worker thread 解說

- 單純以 for-loop 來拿取工作，直到 taskCh 被關掉為止
- 把 qsort partitioning 後的 2 個 subtask 丟回 taskCh
  - 警告：這是因為簡潔範例而做的示範，現實絕對別這麼做

效能

- qsortBad 的效能比較差
  - 主因是 goroutine 會用上比較多 memory
- Open Question ：
  - 如果把 2 個範例中的 time.Sleep 拿掉， qsortBad 的效能遠比 qsortGood 好， why???
  - 怎改寫 qsortGood ？
  - 真正 production grade 的 qsort 應該怎寫？

結語

- 大部份情況，程式內的 thread 數量應該是固定的，不應該跟 input 相關
- Channel 是用來 thread 之間傳送工作
- 善用 channel ，你可以減少在 multithread 下自己使用 locking
- 請善用 sync.waitGroup 來等待 worker

最後一句：

祝願台灣石虎

能見證人類離開太陽系那一天

完