Common Concurrent Programming Mistakes

Go is a language supporting built-in concurrent programming. By using the `go` keyword to create goroutines (light weight threads) and by using (channel-use-cases.html) channels (channel.html) and other concurrency (concurrent-atomic-operation.html) synchronization techniques (concurrent-synchronization-more.html) provided in Go, concurrent programming becomes easy, flexible and enjoyable.

One the other hand, Go doesn't prevent Go programmers from making some concurrent programming mistakes which are caused by either carelessness or lacking of experience. The remaining of the current article will show some common mistakes in Go concurrent programming, to help Go programmers avoid making such mistakes.

No Synchronizations When Synchronizations Are Needed

Code lines might be not executed by their appearance order (memory-model.html).

There are two mistakes in the following program.

- First, the read of `b` in the main goroutine and the write of `b` in the new goroutine might cause data races.
- Second, the condition `b == true` can't ensure that `a != nil` in the main goroutine. Compilers and CPUs may make optimizations by reordering instructions (memory-model.html) in the new goroutine, so the assignment of `b` may happen before the assignment of `a` at run time, which makes that slice `a` is still `nil` when the elements of `a` are modified in the main goroutine.

```
package main

import (
    "time"
    "runtime"
)

func main() {
    var a []int // nil
    var b bool  // false

    // a new goroutine
    go func () {
        a = make([]int, 3)
        b = true // write b
    }()

    for !b { // read b
        time.Sleep(time.Second)
        runtime.Gosched()
    }
    a[0], a[1], a[2] = 0, 1, 2 // might panic
}
```

The above program may run well on one computer, but may panic on another one, or it runs well when it is compiled by one compiler, but panics when another compiler is used.

We should use channels or the synchronization techniques provided in the `sync` standard package to ensure the memory orders. For example,

```
package main

func main() {
    var a []int = nil
    c := make(chan struct{})

    go func () {
        a = make([]int, 3)
        c <- struct{}{}
    }()

    <-c
    // The next line will not panic for sure.
    a[0], a[1], a[2] = 0, 1, 2
}
```

Use `time.Sleep` Calls to Do Synchronizations

Let's view a simple example.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var x = 123

    go func() {
        x = 789 // write x
    }()

    time.Sleep(time.Second)
    fmt.Println(x) // read x
}
```

We expect this program to print `789`. In fact, it really prints `789`, almost always, in running. But is it a program with good synchronization? No! The reason is Go runtime doesn't guarantee the write of `x` happens before the read of `x` for sure. Under certain conditions, such as most CPU resources are consumed by some other computation-intensive programs running on the same OS, the write of `x` might happen after the read of `x`. This is why we should never use `time.Sleep` calls to do synchronizations in formal projects.

Let's view another example.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var n = 123

    c := make(chan int)

    go func() {
        c <- n + 0
    }()

    time.Sleep(time.Second)
    n = 789
    fmt.Println(<-c)
}
```

What do you expect the program will output? `123`, or `789`? In fact, the output is compiler dependent. For the standard Go compiler 1.20, it is very possible the program will output `123`. But in theory, it might also output `789`.

Now, let's change `c <- n + 0` to `c <- n` and run the program again, you will find the output becomes to `789` (for the standard Go compiler 1.20). Again, the output is compiler dependent.

Yes, there are data races in the above program. The expression `n` might be evaluated before, after, or when the assignment statement `n = 789` is processed. The `time.Sleep` call can't guarantee the evaluation of `n` happens before the assignment is processed.

For this specified example, we should store the value to be sent in a temporary value before creating the new goroutine and send the temporary value instead in the new goroutine to remove the data races.

```
...
    tmp := n
    go func() {
        c <- tmp
    }()
...
```

Leave Goroutines Hanging

Hanging goroutines are the goroutines staying in blocking state for ever. There are many reasons leading goroutines into hanging. For example,

- a goroutine tries to receive a value from a channel which no more other goroutines will send values to.
- a goroutine tries to send a value to nil channel or to a channel which no more other goroutines will receive values from.
- a goroutine is dead locked by itself.
- a group of goroutines are dead locked by each other.
- a goroutine is blocked when executing a `select` code block without `default` branch, and all the channel operations following the `case` keywords in the `select` code block keep blocking for ever.

Except sometimes we deliberately let the main goroutine in a program hanging to avoid the program exiting, most other hanging goroutine cases are unexpected. It is hard for Go runtime to judge whether or not a goroutine in blocking state is hanging or stays in blocking state temporarily, so Go runtime will never release the resources consumed by a hanging goroutine.

In the first-response-wins (channel-use-cases.html#first-response-wins) channel use case, if the capacity of the channel which is used a future is not large enough, some slower response goroutines will hang when trying to send a result to the future channel. For example, if the following function is called, there will be 4 goroutines stay in blocking state for ever.

```
1  func request() int {
2      c := make(chan int)
3      for i := 0; i < 5; i++ {
4          i := i
5          go func() {
6              c <- i // 4 goroutines will hang here.
7          }()
8      }
9      return <-c
10 }
```

To avoid the four goroutines hanging, the capacity of channel `c` must be at least `4`.

In the second way to implement the first-response-wins (channel-use-cases.html#first-response-wins-2) channel use case, if the channel which is used as a future/promise is an unbuffered channel, like the following code shows, it is possible that the channel receiver will miss all responses and hang.

```
func request() int {
    c := make(chan int)
    for i := 0; i < 5; i++ {
        i := i
        go func() {
            select {
            case c <- i:
            default:
            }
        }()
    }
    return <-c // might hang here
}
```

The reason why the receiver goroutine might hang is that if the five try-send operations all happen before the receive operation `<-c` is ready, then all the five try-send operations will fail to send values so that the caller goroutine will never receive a value.

Changing the channel `c` as a buffered channel will guarantee at least one of the five try-send operations succeed so that the caller goroutine will never hang in the above function.

Copy Values of the Types in the `sync` Standard Package

In practice, values of the types (except the `Locker` interface values) in the `sync` standard package should never be copied. We should only copy pointers of such values.

The following is bad concurrent programming example. In this example, when the `Counter.Value` method is called, a `Counter` receiver value will be copied. As a field of the receiver value, the respective `Mutex` field of the `Counter` receiver value will also be copied.

The copy is not synchronized, so the copied `Mutex` value might be corrupted. Even if it is not corrupted, what it protects is the use of the copied field `n`, which is meaningless generally.

```
import "sync"

type Counter struct {
    sync.Mutex
    n int64
}

// This method is okay.
func (c *Counter) Increase(d int64) (r int64) {
    c.Lock()
    c.n += d
    r = c.n
    c.Unlock()
    return
}

// The method is bad. When it is called,
// the Counter receiver value will be copied.
func (c Counter) Value() (r int64) {
    c.Lock()
    r = c.n
    c.Unlock()
    return
}
```

We should change the receiver type of the `Value` method to the pointer type `*Counter` to avoid copying `sync.Mutex` values.

The `go vet` command provided in Go Toolchain will report potential bad value copies.

Call the `sync.WaitGroup.Add` Method at Wrong Places

Each `sync.WaitGroup` value maintains a counter internally, The initial value of the counter is zero. If the counter of a `WaitGroup` value is zero, a call to the `Wait` method of the `WaitGroup` value will not block, otherwise, the call blocks until the counter value becomes zero.

To make the uses of `WaitGroup` value meaningful, when the counter of a `WaitGroup` value is zero, the next call to the `Add` method of the `WaitGroup` value must happen before the next call to the `Wait` method of the `WaitGroup` value.

For example, in the following program, the `Add` method is called at an improper place, which makes that the final printed number is not always `100`. In fact, the final printed number of the program may be an arbitrary number in the range `[0, 100)`. The reason is none of the `Add` method calls are guaranteed to happen before the `Wait` method call, which causes none of the `Done` method calls are guaranteed to happen before the `Wait` method call returns.

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {
    var wg sync.WaitGroup
    var x int32 = 0
    for i := 0; i < 100; i++ {
        go func() {
            wg.Add(1)
            atomic.AddInt32(&x, 1)
            wg.Done()
        }()
    }

    fmt.Println("Wait ...")
    wg.Wait()
    fmt.Println(atomic.LoadInt32(&x))
}
```

To make the program behave as expected, we should move the `Add` method calls out of the new goroutines created in the `for` loop, as the following code shown.

```
...
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            atomic.AddInt32(&x, 1)
            wg.Done()
        }()
    }
...
```

Use Channels as Futures/Promises Improperly

From the article channel use cases (channel-use-cases.html), we know that some functions will return channels as futures (channel-use-cases.html#future-promise). Assume `fa` and `fb` are two such functions, then the following call uses future arguments improperly.

```
doSomethingWithFutureArguments(<-fa(), <-fb())
```

In the above code line, the generations of the two arguments are processed sequentially, instead of concurrently.

We should modify it as the following to process them concurrently.

```
ca, cb := fa(), fb()
doSomethingWithFutureArguments(<-ca, <-cb)
```

Close Channels Not From the Last Active Sender Goroutine

A common mistake made by Go programmers is closing a channel when there are still some other goroutines will potentially send values to the channel later. When such a potential send (to the closed channel) really happens, a panic might occur.

This mistake was ever made in some famous Go projects, such as this bug and this bug in the kubernetes project.

Please read this article (channel-closing.html) for explanations on how to safely and gracefully close channels.

Do 64-bit Atomic Operations on Values Which Are Not Guaranteed to Be 8-byte Aligned

The address of the value involved in a non-method 64-bit atomic operation is required to be 8-byte aligned. Failure to do so may make the current goroutine panic. For the standard Go compiler, such failure can only happen on 32-bit architectures. Since Go 1.19, we can use 64-bit method atomic operations to avoid the drawback. Please read memory layouts (memory-layout.html) to get how to guarantee the addresses of 64-bit word 8-byte aligned on 32-bit OSes.

Not Pay Attention to Too Many Resources Are Consumed by Calls to the `time.After` Function

The `After` function in the `time` standard package returns a channel for delay notification (channel-use-cases.html#timer). The function is convenient, however each of its calls will create a new value of the `time.Timer` type. The new created `Timer` value will keep alive in the

duration specified by the passed argument to the `After` function. If the function is called many times in a certain period, there will be many alive `Timer` values accumulated so that much memory and computation is consumed.

For example, if the following `longRunning` function is called and there are millions of messages coming in one minute, then there will be millions of `Timer` values alive in a certain small period (several seconds), even if most of these `Timer` values have already become useless.

```
import (
    "fmt"
    "time"
)

// The function will return if a message
// arrival interval is larger than one minute.
func longRunning(messages <-chan string) {
    for {
        select {
        case <-time.After(time.Minute):
            return
        case msg := <-messages:
            fmt.Println(msg)
        }
    }
}
```

To avoid too many `Timer` values being created in the above code, we should use (and reuse) a single `Timer` value to do the same job.

```
func longRunning(messages <-chan string) {
    timer := time.NewTimer(time.Minute)
    defer timer.Stop()

    for {
        select {
        case <-timer.C: // expires (timeout)
            return
        case msg := <-messages:
            fmt.Println(msg)

            // This "if" block is important.
            if !timer.Stop() {
                <-timer.C
            }
        }

        // Reset to reuse.
        timer.Reset(time.Minute)
    }
}
```

Note, the `if` code block is used to discard/drain a possible timer notification which is sent in the small period when executing the second branch code block.

Use `time.Timer` Values Incorrectly

An idiomatic use example of `time.Timer` values has been shown in the last section. Some explanations:

- the `Stop` method of a `*Timer` value returns `false` if the corresponding `Timer` value has already expired or been stopped. If the `Stop` method returns `false`, and we know the `Timer` value has not been stopped yet, then the `Timer` value must have already expired.
- after a `Timer` value is stopped, its `C` channel field can only contain most one timeout notification.
- we should take out the timeout notification, if it hasn't been taken out, from a timeout `Timer` value after the `Timer` value is stopped and before resetting and reusing the `Timer` value. This is the meaningfulness of the `if` code block in the example in the last section.

The `Reset` method of a `*Timer` value must be called when the corresponding `Timer` value has already expired or been stopped, otherwise, a data race may occur between the `Reset` call and a possible notification send to the `C` channel field of the `Timer` value.

If the first `case` branch of the `select` block is selected, it means the `Timer` value has already expired, so we don't need to stop it, for the sent notification has already been taken out. However, we must stop the timer in the second branch to check whether or not a timeout notification exists. If it does exist, we should drain it before reusing the timer, otherwise, the notification will be fired immediately in the next loop step.

For example, the following program is very possible to exit in about one second, instead of ten seconds. And more importantly, the program is not data race free.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    timer := time.NewTimer(time.Second/2)
    select {
    case <-timer.C:
    default:
        // Most likely go here.
        time.Sleep(time.Second)
    }
    // Potential data race in the next line.
    timer.Reset(time.Second * 10)
    <-timer.C
    fmt.Println(time.Since(start)) // about 1s
}
```

A `time.Timer` value can be leaved in non-stopping status when it is not used any more, but it is recommended to stop it in the end.

It is bug prone and not recommended to use a `time.Timer` value concurrently among multiple goroutines.

We should not rely on the return value of a `Reset` method call. The return result of the `Reset` method exists just for compatibility purpose.

Index↡

The ***Go 101*** project is hosted on Github . Welcome to improve ***Go 101*** articles by submitting corrections for all kinds of mistakes, such as typos, grammar errors, wording inaccuracies, description flaws, code bugs and broken links.

If you would like to learn some Go details and facts every serveral days, please follow Go 101's official Twitter account @go100and1 .