Close Channels Gracefully in Golang

Several days ago, I wrote an article which explains the channel rules in Golang (golang-channel). That article got many votes on reddit (https://www.reddit.com/r/golang/comments/5k489v/the_full_list_of_channel_rules_in_golang/) and HN (https://news.ycombinator.com/item?id=13252416). But I also collected some criticisms on the the following designs and rules of Go channel:

1. no easy and universal ways to check whether or not a channel is closed without modifying the status of the channel.
2. close a closed channel will panic, so it is dangerous to close a channel if the closers don't know whether or not the channel is closed.
3. send values to a closed channel will panic, so it is dangerous to send values to a channel if the senders don't know whether or not the channel is closed.

The criticisms look reasonable (in fact not). Yes, there is not a built-in function to check whether or not a channel has been closed.

There is indeed a simple method to check whether or not a channel is closed if you can make sure no values were ever sent to the channel (this method will be often used in other examples in this article):

```
package main

import "fmt"

type T int

func IsClosed(ch <-chan T) bool {
    select {
    case <-ch:
        return true
    default:
    }

    return false
}

func main() {
    c := make(chan T)
    fmt.Println(IsClosed(c)) // false
    close(c)
    fmt.Println(IsClosed(c)) // true
}
```

As above mentioned, this is not an universal way to check whether a channel is closed.

However, even if there is a simple `closed(chan T) bool` function to check whether or not a channel has been closed, its usefulness would be very limited, just like the built-in `len` function for checking the current number of values stored in the value buffer of a channel. The reason is the status of the checked channel may have changed just after a call to such functions returns, so that the returned value has already not be able to reflect the latest status of the just checked channel. Although it is okay to stop sending values to a channel `ch` if the call `closed(ch)` returns `true`, it is not safe to close the channel or continue sending values to the channel if the call `closed(ch)` returns `false`.

The Channel Closing Principle

One general principle of using Go channels is ***don't close a channel from the receiver side and don't close a channel if the channel has multiple concurrent senders***. In other words, you should only close a channel in a sender goroutine if the sender is the only sender or the last active sender of the channel, to notify the receiver(s) that there are no more values to read out. Holding this principle ensures sending values to (and closing) closed channels will never happen.

*(Below, we will call the above principle as **channel closing principle**.)*

Solutions Which Close Channels Rudely

If you would close a channel from the receiver side or in one of the multiple senders of the channel for any reason, then you can use the recover mechanizm to prevent the possible panic from crashing your program. Assume the channel element type is `T`:

```
func SafeClose(ch chan T) (justClosed bool) {
    defer func() {
        if recover() != nil {
            justClosed = false
        }
    }()

    // assume ch != nil here.
    close(ch) // panic if ch is closed
    return true // <=> justClosed = true; return
}
```

The function would be almost as performant as the bare `ch <- value` if the channel `ch` is not closed. For the case of the channel is closed, the `SafeClose` function will only be called once in each sender goroutine, so there will be no big overall performance loss for a program.

This solution obviously breaks the **channel closing principle**.

The same idea can be used for sending values to a potential closed channel:

```
func SafeSend(ch chan T, value T) (closed bool) {
    defer func() {
        if recover() != nil {
            // The return result can be altered
            // in a defer function call.
            closed = true
        }
    }()

    ch <- value // panic if ch is closed
    return false // <=> closed = false; return
}
```

Solutions Which Close Channels Politely

Many people prefer using `sync.Once` to close channels:

```
type MyChannel struct {
    C    chan T
    once sync.Once
}

func NewMyChannel() *MyChannel {
    return &MyChannel{C: make(chan T)}
}

func (mc *MyChannel) SafeClose() {
    mc.once.Do(func() {
        close(mc.C)
    })
}
```

Surely, we can also use `sync.Mutex` to avoid closing a channel multiple times:

```
type MyChannel struct {
    C      chan T
    closed bool
    mutex  sync.Mutex
}

func NewMyChannel() *MyChannel {
    return &MyChannel{C: make(chan T)}
}

func (mc *MyChannel) SafeClose() {
    mc.mutex.Lock()
    if !mc.closed {
        close(mc.C)
        mc.closed = true
    }
    mc.mutex.Unlock()
}

func (mc *MyChannel) IsClosed() bool {
    mc.mutex.Lock()
    defer mc.mutex.Unlock()
    return mc.closed
}
```

We should comprehend that the reason of why Go doesn't support built-in `SafeSend` and `SafeClose` functions is that it is not recommended to close a channel from the receiver side or from multiple concurrent senders. Golang even forbids closing receive-only channels.

Solutions Which Close Channels Gracefully

One drawback of the above `SafeSend` function is that its calls can't be used as send operations which follow the `case` keyword in select blocks. The other drawback of the above `SafeSend` and `SafeClose` functions is that many people, including me, would think the above solutions by using `panic` / `recover` and `sync` package are not graceful. Following, some pure-channel solutions without using `panic` / `recover` and `sync` package will be introduced, for all kinds of situations.

(In the following examples, `sync.WaitGroup` is used to make the examples complete. Its usages may be not always essential in real practice.)

1. M receivers, one sender, the sender says "no more sends" by closing the data channel

This is the simplest situation, just let the sender close the data channel when it doesn't want to send more:

```
package main

import (
    "time"
    "math/rand"
    "sync"
    "log"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    log.SetFlags(0)

    // ...
    const MaxRandomNumber = 100000
    const NumReceivers = 100

    wgReceivers := sync.WaitGroup{}
    wgReceivers.Add(NumReceivers)

    // ...
    dataCh := make(chan int, 100)

    // the sender
    go func() {
        for {
            if value := rand.Intn(MaxRandomNumber); value == 0 {
                // The only sender can close the channel safely.
                close(dataCh)
                return
            } else {
                dataCh <- value
            }
        }
    }()

    // receivers
    for i := 0; i < NumReceivers; i++ {
        go func() {
            defer wgReceivers.Done()

            // Receive values until dataCh is closed and
            // the value buffer queue of dataCh is empty.
            for value := range dataCh {
                log.Println(value)
            }
        }()
    }

    wgReceivers.Wait()
}
```

2. One receiver, N senders, the receiver says "please stop sending more" by closing an additional signal channel

This is a situation a little more complicated than the above one. We can't let the receiver close the data channel, for doing this will break the **channel closing principle**. But we can let the receiver close an additional signal channel to notify senders to stop sending values:

```
package main

import (
    "time"
    "math/rand"
    "sync"
    "log"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    log.SetFlags(0)

```

```
    // ...
    const MaxRandomNumber = 100000
    const NumSenders = 1000

    wgReceivers := sync.WaitGroup{}
    wgReceivers.Add(1)

    // ...
    dataCh := make(chan int, 100)
    stopCh := make(chan struct{})
        // stopCh is an additional signal channel.
        // Its sender is the receiver of channel dataCh.
        // Its reveivers are the senders of channel dataCh.

    // senders
    for i := 0; i < NumSenders; i++ {
        go func() {
            for {
                // The first select here is to try to exit the goroutine
                // as early as possible. In fact, it is not essential
                // for this example, so it can be omitted.
                select {
                case <- stopCh:
                    return
                default:
                }

                // Even if stopCh is closed, the first branch in the
                // second select may be still not selected for some
                // loops if the send to dataCh is also unblocked.
                // But this is acceptable, so the first select
                // can be omitted.
                select {
                case <- stopCh:
                    return
                case dataCh <- rand.Intn(MaxRandomNumber):
                }
            }
        }()
    }

    // the receiver
    go func() {
        defer wgReceivers.Done()

        for value := range dataCh {
            if value == MaxRandomNumber-1 {
                // The receiver of the dataCh channel is
                // also the sender of the stopCh cahnnel.
                // It is safe to close the stop channel here.
                close(stopCh)
                return
            }

            log.Println(value)
        }
    }()

    // ...
    wgReceivers.Wait()
}
```

As mentioned in the comments, for the additional signal channel, its sender is the receiver of the data channel. The additional signal channel is closed by its only sender, which holds the **channel closing principle**.

In this example, the channel `dataCh` is never closed. Yes, channels don't have to be closed. A channel will be eventually garbage collected if no goroutines reference it any more, whether it is closed or not.

3. M receivers, N senders, random one of them says "let's end the game" by notifying a moderator to close an additional signal channel

This is a the most complicated situation. We can't let any of the receivers and the senders close the data channel. And we can't let any of the receivers

close an additional signal channel to notify all senders and receivers to exit the game. Doing either will break the **channel closing principle**. However, we can introduce a moderator role to close the additional signal channel. One trick in this example is how to notify the moderator to close the additional signal channel:

```
package main

import (
    "time"
    "math/rand"
    "sync"
    "log"
    "strconv"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    log.SetFlags(0)

    // ...
    const MaxRandomNumber = 100000
    const NumReceivers = 10
    const NumSenders = 1000

    wgReceivers := sync.WaitGroup{}
    wgReceivers.Add(NumReceivers)

    // ...
    dataCh := make(chan int, 100)
    stopCh := make(chan struct{})
        // stopCh is an additional signal channel.
        // Its sender is the moderator goroutine shown below.
        // Its reveivers are all senders and receivers of dataCh.
    toStop := make(chan string, 1)
        // The channel toStop is used to notify the moderator
        // to close the additional signal channel (stopCh).
        // Its senders are any senders and receivers of dataCh.
        // Its reveiver is the moderator goroutine shown below.

    var stoppedBy string

    // moderator
    go func() {
        stoppedBy = <- toStop
        close(stopCh)
    }()

    // senders
    for i := 0; i < NumSenders; i++ {
        go func(id string) {
            for {
                value := rand.Intn(MaxRandomNumber)
                if value == 0 {
                    // Here, a trick is used to notify the moderator
                    // to close the additional signal channel.
                    select {
                    case toStop <- "sender#" + id:
                    default:
                    }
                    return
                }

                // The first select here is to try to exit the goroutine
                // as early as possible. This select blocks with one
                // receive operation case and one default branches will
                // be optimized as a try-receive operation by the
                // official Go compiler.
                select {
                case <- stopCh:
                    return
                default:
```

```
                }

                // Even if stopCh is closed, the first branch in the
                // second select may be still not selected for some
                // loops (and for ever in theory) if the send to
                // dataCh is also unblocked.
                // This is why the first select block is needed.
                select {
                case <- stopCh:
                    return
                case dataCh <- value:
                }
            }
        }(strconv.Itoa(i))
    }

    // receivers
    for i := 0; i < NumReceivers; i++ {
        go func(id string) {
            defer wgReceivers.Done()

            for {
                // Same as the sender goroutine, the first select here
                // is to try to exit the goroutine as early as possible.
                select {
                case <- stopCh:
                    return
                default:
                }

                // Even if stopCh is closed, the first branch in the
                // second select may be still not selected for some
                // loops (and for ever in theory) if the receive from
                // dataCh is also unblocked.
                // This is why the first select block is needed.
                select {
                case <- stopCh:
                    return
                case value := <-dataCh:
                    if value == MaxRandomNumber-1 {
                        // The same trick is used to notify
                        // the moderator to close the
                        // additional signal channel.
                        select {
                        case toStop <- "receiver#" + id:
                        default:
                        }
                        return
                    }

                    log.Println(value)
                }
            }
        }(strconv.Itoa(i))
    }

    // ...
    wgReceivers.Wait()
    log.Println("stopped by", stoppedBy)
}
```

In this example, the **channel closing principle** is still held.

Please note that the buffer size of channel `toStop` is one. This is to avoid the first notification is missed when it is sent before the moderator goroutine gets ready to receive nitification from `toStop` .

4. more situations?

There are more situation variants based on the above three ones. For example, one variant based on the most complicated one may require the receivers read all the remaining values out of the buffered data channel. This would be easy to handle and this article will not cover it.

Although the above three situations can't cover all the Go channel using situations, they are the basic ones. Most situations in practice can be classified into the three ones.

Conclusion

There is no situations which will force you to break the **channel closing principle**. If you encounter such a situation, please rethink your design and rewrite you code.

Programming with Go channels likes making art.

This article is written by @TapirLiu (https://twitter.com/tapirliu). who made some mobile (Android and iPhone/iPad) and web games (http://www.tapirgames.com), He also invented a touch programming language, IfLoop (http://www.tapirgames.com/App/Ifloop).
