The Pragmatic Programmers

Effective Go Recipes

Fast Solutions to Common Tasks

Miki Tebeka

edited by Margaret Eldridge

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Recipe 44

Handling Panics in Goroutines

Task

You are the owner of an internal library that executes jobs from a channel in goroutines. When you arrive in the morning, you see the following issue assigned to you from the operations team lead, Mia:

> Last night the system crashed several times. We triaged the bug to a new handler that panicked. We disabled the handler, and now the system runs. Please catch panics in the handlers. I don't like the sound the pager makes at 3 a.m.

Solution

You look at the code, and the main loop seems simple:

errors/drain/drain.go

```
type Message struct {
    Time time.Time
    Type string
    Data []byte
}

func drain(ch <-chan Message, handler func(Message)) {
    for msg := range ch {
        msg.Time = time.Now()
        go handler(msg)
    }
}
```

You create a test handler that will panic:

errors/drain/drain.go

```
func testHandler(msg Message) {
    ts := msg.Time.Format("15:04:03")
    log.Printf("%s: %s: %x...\n", ts, msg.Type, msg.Data[:20])
}
```

Then you run the code:

errors/drain/drain.go

```
func main() {
    ch := make(chan Message)

    // Populate some data
    go func() {
        defer close(ch)
        for i := 0; i < 5; i++ {
            msg := Message{
                Type: "test",
                Data: []byte(fmt.Sprintf("payload %d", i)),
            }
            ch <- msg
        }
    }()

    drain(ch, testHandler)
    time.Sleep(time.Second) // let goroutines run
    fmt.Println("DONE")
}
```

And you see the problem:

```
panic: runtime error: slice bounds out of range [:20] with capacity 16

goroutine 19 [running]:
main.testHandler(0xbfd5745aa34b8eee, 0x12c99, 0x569420, 0x4cb528, \
                0x4, 0xc0000b8040, 0x9, 0x10)
        ./code/errors/drain.go:30 +0x17e
created by main.drain
        ./code/errors/drain.go:36 +0x117
```

Solution

You write safelyGo, which will launch a goroutine that wraps the handler in defer + recover:

errors/drain/fix/drain.go

```
// safelyGo will run fn in a goroutine, and guard it from panics
func safelyGo(fn func()) {
    go func() {
        defer func() {
            if err := recover(); err != nil {
                log.Printf("error: %s", err)
            }
        }()
        fn()
    }()
}
```

And then you use it in the drain function:

errors/drain/fix/drain.go

```
func drain(ch <-chan Message, handler func(Message)) {
    for msg := range ch {
        msg.Time = time.Now()
        safelyGo(func() {
            handler(msg)
        })
    }
}
```

When you run the code now, you see error messages, but it runs to completion:

```
2020/10/01 07:36:06 error: runtime error: slice bounds out of range [:20] \
    with capacity 16
2020/10/01 07:36:06 error: runtime error: slice bounds out of range [:20] \
    with capacity 16
2020/10/01 07:36:06 error: runtime error: slice bounds out of range [:20] \
    with capacity 16
2020/10/01 07:36:06 error: runtime error: slice bounds out of range [:20] \
    with capacity 16
2020/10/01 07:36:06 error: runtime error: slice bounds out of range [:20] \
    with capacity 16
DONE
```

Discussion

Crashing the whole program when a goroutine panics might seems like an odd design choice, but a good reason is behind it.

When you run a thread in another programming language and it crashes, the program will continue to run. However, the program runs in a bad state, and the main program won't know that the thread is no longer running. Eventually, the program will crash, and it'll be much harder to figure the cause of the issue. See the previous Discussion, on page ?, for more information.

The Go developers believe (as do I) that it's better to crash than continue running in a bad state. It's much harder to fix bugs in the latter approach.

In some cases, you'll want to guard from panics in goroutines. A method like safelyGo can solve most of the problem. However, if the handler code is started in a different goroutine, the program will still crash. There's no bulletproof way to guard against panics.

Here's an example with the standard library HTTP server:

errors/drain/httpd/crasher.go

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func crashHandler(w http.ResponseWriter, r *http.Request) {
    go func() {
        panic("down we go!")
    }()
    fmt.Fprintln(w, "OK")
}

func main() {
    http.HandleFunc("/carsh", crashHandler)

    addr := ":8080"
    log.Printf("server starting on %s", addr)
    if err := http.ListenAndServe(addr, nil); err != nil {
        log.Fatalf("error: %s", err)
    }
}
```

If you run this code and then hit http://localhost:8080/carsh, you'll see "OK". When you try again, you'll see that the server is no longer running.

What you need is another layer that will restart crashing services. Systems such as Docker, Kubernetes, and others know how to do that. Don't forget to monitor such crashes and alter on them. See more on this at *Shipping Your Code*.