The Go Blog

Defer, Panic, and Recover

4 August 2010

Go has the usual mechanisms for control flow: if, for, switch, goto. It also has the go statement to run code in a separate goroutine. Here I'd like to discuss some of the less common ones: defer, panic, and recover.

A **defer statement** pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

For example, let's look at a function that opens two files and copies the contents of one file to the other:

```
func CopyFile(dstName, srcName string) (written int64, err
error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

This works, but there is a bug. If the call to os.Create fails, the function will return without closing the source file. This can be easily remedied by putting a call to src.Close before the second return statement, but if the function were more complex the problem might not be so easily noticed and resolved. By introducing defer statements we can ensure that the files are always closed:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

Defer statements allow us to think about closing each file right after opening it, guaranteeing that, regardless of the number of return statements in the function, the files *will* be closed.

Next article

Go Wins 2010 Bossie Award

Previous article

Share Memory By Communicating

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

The behavior of defer statements is straightforward and predictable. There are three simple rules:

1. *A deferred function's arguments are evaluated when the defer statement is evaluated.*

In this example, the expression "i" is evaluated when the Println call is deferred. The deferred call will print "0" after the function returns.

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}
```

2. *Deferred function calls are executed in Last In First Out order after the surrounding function returns.*

This function prints "3210":

```
func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}
```

3. *Deferred functions may read and assign to the returning function's named return values.*

In this example, a deferred function increments the return value i *after* the surrounding function returns. Thus, this function returns 2:

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
```

This is convenient for modifying the error return value of a function; we will see an example of this shortly.

Panic is a built-in function that stops the ordinary flow of control and begins *panicking*. When the function F calls panic, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller. To the caller, F then behaves like a call to panic. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking panic directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.

Recover is a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions. During normal execution, a call to recover will return nil and have no other effect. If the current goroutine is panicking, a call to recover will capture the value given to panic and resume normal execution.

Here's an example program that demonstrates the mechanics of panic and defer:

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

The function g takes the int i, and panics if i is greater than 3, or else it calls itself with the argument i+1. The function f defers a function that calls recover and prints the recovered value (if it is non-nil). Try to picture what the output of this program might be before reading on.

The program will output:

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

If we remove the deferred function from f the panic is not recovered and reaches the top of the goroutine's call stack, terminating the program. This modified program will output:

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
panic: 4

panic PC=0x2a9cd8
[stack trace omitted]
```

For a real-world example of **panic** and **recover**, see the json package from the Go standard library. It decodes JSON-encoded data with a set of recursive functions. When malformed JSON is encountered, the parser calls panic to unwind the stack to the top-level function call, which recovers from the panic and returns an appropriate error value (see the 'error' and 'unmarshal' methods of the decodeState type in decode.go).

The convention in the Go libraries is that even when a package uses panic internally, its external API still presents explicit error return values.

Other uses of **defer** (beyond the file.Close example given earlier) include releasing a mutex:

```
mu.Lock()
defer mu.Unlock()
```

printing a footer:

```
printHeader()
defer printFooter()
```

and more.

In summary, the defer statement (with or without panic and recover) provides an unusual and powerful mechanism for control flow. It can be used to model a number of features implemented by special-purpose structures in other programming languages. Try it out.

By Andrew Gerrand

Related articles

- HTTP/2 Server Push
- Introducing HTTP Tracing
- Generating code
- Introducing the Go Race Detector
- Go maps in action
- go fmt your code
- Organizing Go code
- Debugging Go programs with the GNU Debugger
- The Go image/draw package
- The Go image package
- The Laws of Reflection
- Error handling and Go
- "First Class Functions in Go"
- Profiling Go Programs
- A GIF decoder: an exercise in Go interfaces
- Introducing Gofix
- Godoc: documenting Go code
- Gobs of data
- C? Go? Cgo!
- JSON and Go
- Go Slices: usage and internals
- Go Concurrency Patterns: Timing out, moving on
- Share Memory By Communicating
- JSON-RPC: a tale of interfaces