

Training 3 - Concurrency

Calvin Lei-Cramer
May 2025



What are we covering today?

Panic and recover,
goroutines and
concurrency, channels

Panic! Recover.

- **What are panics actually? Are they exceptions?**
- **What triggers panics?**
 - Array/slice index out of bounds
 - Divide by zero
 - Calling `panic()`
 - Nil pointer deference
 - Type assertion (without using 'ok' second return value) failing

```
1 func main() {
2     _ = []int{1, 2, 3}[10]
3
4     div := 0
5     _ = 5 / div
6
7     panic("Something bad happened!")
8
9     var numPtr *int = nil
10    _ = *numPtr
11
12    var something any = struct{ A int }{A: 1000}
13    _ = something.(string)
14 }
```



Panic! Recover.

- **How to catch panics?**
- **Recover() global**
 - Returns value passed to panic() if panicking
 - Otherwise returns nil
- **Named return values, defer, recover()**

```
$ go run .
25 / 2 = 12
43 / 0 = 9223372036854775807
-96 / 0 = -9223372036854775808
$
```

```
1 func SafeDivide(num, div int) (result int) {
2     defer func() {
3         if recover() != nil {
4             // Set to near infinity
5             if num < 0 {
6                 result = math.MinInt
7             } else {
8                 result = math.MaxInt
9             }
10        }()
11    }()
12    result = num / div // may cause panic
13    return
14 }
```

Panic! Recover.

- Pattern to recover normal execution flow like try/catch

Python

```
1 def main():
2     print("Before")
3     try:
4         badFunction()
5     except Exception as e:
6         pass
7     print("After")
```

```
$ go run .
Before
Log panic: something bad happened in badFunc()
After
```

Golang

```
1 func handlePanic(action func()) {
2     defer func() {
3         if val := recover(); val != nil {
4             fmt.Println("Log panic:", val)
5         }
6     }()
7     action()
8 }
9
10 func main() {
11     fmt.Println("Before")
12     handlePanic(badFunc)
13     fmt.Println("After")
14 }
```

Panic! Recover.

- What happens if panic, then panic again in defer function?

```
3 func main() {
4     defer func() {
5         panic("panic #2")
6     }()
7     panic("panic #1")
8 }
```

```
$ go run .
panic: panic #1
    panic: panic #2

goroutine 1 [running]:
main.main.func1()
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x25
panic({0x46f5a0?, 0x499a78?})
    /usr/local/go/src/runtime/panic.go:792 +0x132
main.main()
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x3e
exit status 2
$
```

Panic! Recover.

```
3 func doStuff() {
4     defer func() {
5         if val := recover(); val != nil {
6             fmt.Println("Recovered panic:", val)
7         }
8     }()
9     defer func() {
10        panic("panic #2")
11    }()
12    panic("panic #1")
13 }
14
15 func main() {
16     fmt.Println("Before")
17     doStuff()
18     fmt.Println("After")
19 }
```

- Panic, panic, recover?

```
$ go run .
Before
Recovered panic: panic #2
After
$
```

Panic overrides any current panic value.

Keep panic recover simple.

Golang Idioms

- **Idiomatic Go, mantras, patterns, group think**
- **“Programming in a language vs programming into a language” – Code Complete by Steve McConnell**
- **Error handling → errors are values, don’t code by exception**
- **Defer**
- **Goroutines and concurrency models**

```
3 func fetch(url string) (body string, err error) {  
4     resp, err := http.Get(url)  
5     if err != nil {  
6         return  
7     }  
8     defer resp.Body.Close()  
9     bodyBytes, err := io.ReadAll(resp.Body)  
10    if err != nil {  
11        return  
12    }  
13    body = string(bodyBytes)  
14    return  
15 }  
16  
17 func main() {  
18     resp, err := fetch("https://www.google.com")  
19     if err != nil {  
20         fmt.Println("error fetching URL -", err)  
21         return  
22     }  
23     fmt.Println(resp)  
24 }
```

Goroutines

- **What are goroutines?**
 - Lightweight application-level threads managed by Go runtime
 - Multiple goroutines may run on a single OS-level thread (concurrency) or multiple OS-level threads (parallelism). Go runtime handles this.
 - OS-level threads may be scheduled to run on arbitrary CPU core by OS.
 - All goroutines share the same address space
 - Synchronization → no IPC required. Can use shared memory (mutex, locks, barriers) for synchronization.
- **Spawn goroutine? → go funcCall(args...)**

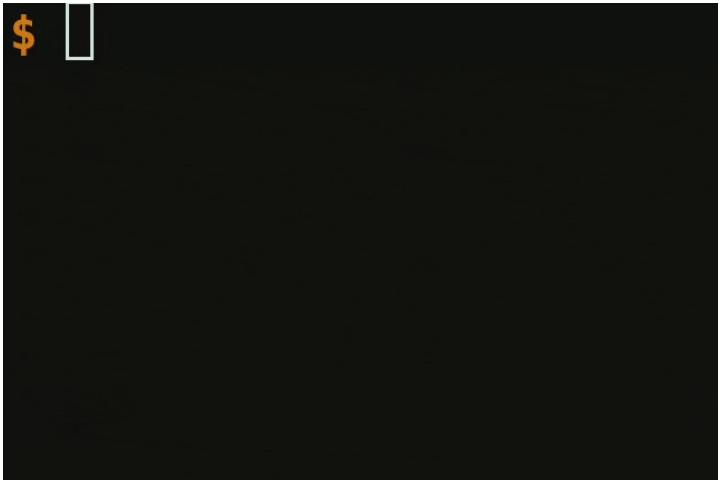
Goroutines Example

```
1 func child() {
2     for {
3         var msSleep int = rand.IntN(2000) + 500
4         time.Sleep(time.Duration(msSleep) * time.Millisecond)
5         fmt.Println("Child: are we there yet?")
6     }
7 }
8
9 func parent() {
10    for {
11        var msSleep int = rand.IntN(3000) + 1000
12        time.Sleep(time.Duration(msSleep) * time.Millisecond)
13        fmt.Println("Parent: 10 more minutes")
14    }
15 }
16
17 func main() {
18     go child()
19     go parent()
20     for { time.Sleep(time.Second) }
21 }
```

```
$ []
```

Goroutines

- Join goroutine? pthread_join?
- Nope! Will need to use other tools
- Sync.Mutex

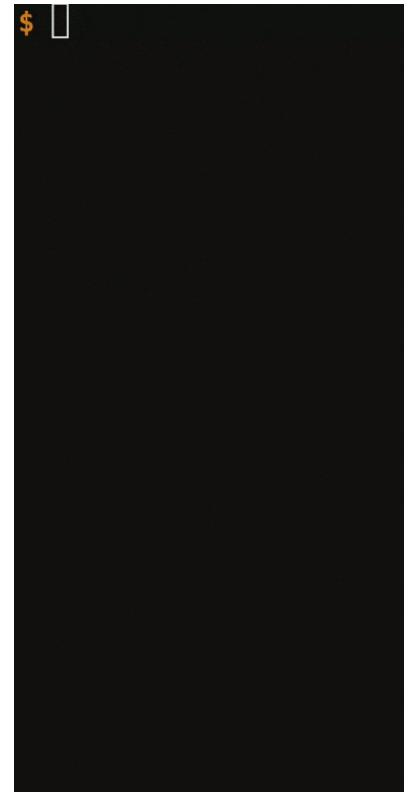


```
1 var childLock sync.Mutex
2
3 func child() {
4     fmt.Println("Getting lunch...")
5     time.Sleep(time.Second)
6     fmt.Println("Getting homework...")
7     time.Sleep(time.Second)
8     fmt.Println("Getting socks...")
9     time.Sleep(time.Second)
10    fmt.Println("Getting jacket...")
11    time.Sleep(time.Second)
12    childLock.Unlock()
13 }
14
15 func main() {
16     childLock.Lock()
17     go child()
18     // Wait for child
19     childLock.Lock()
20     fmt.Println("Walking to school!")
21 }
```

Goroutines

- **Sync.WaitGroup → wait for a group of goroutines**
- `wg.Add(n)` → add n goroutines to wait for (usually 1 or all)
- `wg.Done()` → call by goroutine when done (subtracts 1)
- `wg.Wait()` → block until no goroutines running (wait for 0)

```
1 func main() {
2     var wg sync.WaitGroup
3     const numChildren = 5
4
5     childDoHomework := func(num int) {
6         for {
7             fmt.Printf("Child %d working...\n", num)
8             wait := time.Duration(rand.IntN(2000)+500)
9             time.Sleep(time.Millisecond * wait)
10            if rand.Float64() <= 0.25 {
11                break
12            }
13        }
14        fmt.Printf("Child %d done\n", num)
15        wg.Done()
16    }
17
18    wg.Add(numChildren)
19    for i := range numChildren {
20        go childDoHomework(i)
21    }
22    wg.Wait()
23    fmt.Println("Everyone is done!")
24}
25
```



Goroutines

- Handling synchronization of global data via sync primitives

```
1 func main() {  
2     var total int = 0  
3     var wg sync.WaitGroup  
4     wg.Add(1000000)  
5     for range 1000000 {  
6         go func() {  
7             defer wg.Done()  
8             total += 1  
9         }()  
10    }  
11    wg.Wait()  
12    fmt.Println(total)  
13 }
```

```
$ for i in {1..100}; do go run _; done  
922539  
927714  
929655  
925084  
925332  
923226  
921020  
924350  
919063  
918562  
917953  
911826  
915570  
923962  
915425  
913478
```

Goroutines

- Handling synchronization of global data via sync primitives

```
1 func main() {
2     var total int = 0
3     var lock sync.Mutex
4     var wg sync.WaitGroup
5     wg.Add(1000000)
6     for range 1000000 {
7         go func() {
8             defer wg.Done()
9             lock.Lock()
10            total += 1
11            lock.Unlock()
12        }()
13    }
14    wg.Wait()
15    fmt.Println(total)
16 }
```

```
$ for i in {1..100}; do go run _; done
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
```

Channels

- **What are channels?**
 - bidirectional communication mechanism between goroutines
- **make(chan TYPE)**
- **Can send/receive things on channels**
 - Send: myChannel <- value
 - Receive: var := <-myChannel
- **Channels include the type of data transferred → chan int, chan MyStruct**

Channels

```
5 func main() {  
6     postOffice := make(chan string)  
7     go penPal(postOffice)  
8     message := "Hi there pen pal!"  
9     fmt.Println("main sending message")  
10    postOffice <- message  
11    reply := <-postOffice  
12    fmt.Println("main received a reply!", reply)  
13 }
```

```
15 func penPal(messageChan chan string) {  
16     message := <-messageChan  
17     fmt.Println("Pen pal received a message!")  
18     newMessage := "Cool message! " + message  
19     messageChan <- newMessage  
20 }
```

Waits for message

```
$ go run _  
main sending message  
Pen pal received a message!  
main received a reply! Cool message! Hi there pen pal!  
$
```

Channels

- What if both try to send messages at same time?

```
1 func main() {
2     postOffice := make(chan string)
$ go run .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:6 +0x70

goroutine 5 [chan send]:
main.penPal(...)
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:10
created by main.main in goroutine 1
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x5f
exit status 2
$
```

Channels

- What if both wait to receive messages at same time?

```
1 func main() {  
$ go run .  
fatal error: all goroutines are asleep - deadlock!  
  
goroutine 1 [chan receive]:  
main.main()  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:8 +0x7b  
  
goroutine 18 [chan receive]:  
main.penPal(0x0?)  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:13 +0x1e  
created by main.main in goroutine 1  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x66  
exit status 2  
$
```

Channels

- **Channels are preferred over synchronization primitives**
- **“Don’t communicate by sharing memory, share memory by communicating”**
- **Synchronization primitives are shared data and sync mechanism**
- **Channels have no shared data, send/receive state with other goroutines**
- **Channels are similar to distributed computing, for example communicating between multiple hosts over HTTP/REST**

Channels

- **Unbuffered vs buffered channels**
- **Unbuffered = channel has no capacity**
 - Think of unbuffered as writer literally handing message to reader
 - Aka **synchronous** communication
 - `make(chan int)`
 - `make(chan int, 0)`
- **Buffered = channel has capacity → `make(chan int, 100)`**
 - Aka **asynchronous** communication

Channels

- When do channels block?

	Read	Write
Unbuffered	Blocks until value available	Blocks until value is read
Buffered	Blocks if channel empty	Blocks if channel is full

Channels

- Buffered channel example

```
$ go run .
.....CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
.....CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
CCCCC.....CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
.....CCCC.....CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
.....CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.
```

```
1 var wg sync.WaitGroup
2
3 func main() {
4     ch := make(chan bool, 100)
5     wg.Add(2)
6     go producer(ch)
7     go consumer(ch)
8     wg.Wait()
9     fmt.Println("\n")
10 }
11
12 func producer(ch chan bool) {
13     defer wg.Done()
14     for range 1000 {
15         ch <- false
16         fmt.Println(".")
17     }
18     ch <- true
19 }
20
21 func consumer(ch chan bool) {
22     defer wg.Done()
23     for {
24         val := <-ch
25         if val == true {
26             break
27         }
28         fmt.Println("C")
29     }
30 }
```



Channels

- Really should treat channels as unidirectional most of the time.
- If need bidirectional then just use two channels. Especially for buffered channels.

```
1 func main() {  
2     fruitSalad := make(chan bool, 10)  
3     pancake := make(chan bool, 10)  
4     wg.Add(2)  
5     go chef1(fruitSalad, pancake)  
6     go chef2(fruitSalad, pancake)  
7     wg.Wait()  
8 }
```

```
1 func chef1(fruitSalad chan bool, pancake chan bool) {  
2     defer wg.Done()  
3     for range 1000 {  
4         <-fruitSalad // chef1 likes fruit salad  
5         pancake <- true // chef1 makes pancaked  
6     }  
7 }
```

```
1 func chef2(fruitSalad chan bool, pancake chan bool) {  
2     defer wg.Done()  
3     for range 1000 {  
4         fruitSalad <- true // chef2 makes fruit salad  
5         <-pancake // chef2 likes pancakes  
6     }  
7 }
```

Channels

- Select control flow
- wait for multiple channels
- Useful for timeout and stopping conditions

```
1 func main() {
2     chNumber := time.Tick(time.Second)
3     chFizz := time.Tick(time.Second * 3)
4     chBuzz := time.Tick(time.Second * 5)
5     chFizzBuzz := time.Tick(time.Second * 3 * 5)
6     chTimeout := time.After(time.Second * 20)
7     num := 1
8
9     for {
10         select {
11             case <-chNumber:
12                 fmt.Println(num)
13                 num += 1
14             case <-chFizz:
15                 fmt.Println("Fizz")
16             case <-chBuzz:
17                 fmt.Println("Buzz")
18             case <-chFizzBuzz:
19                 fmt.Println("Fizz Buzz")
20             case <-chTimeout:
21                 return
22         }
23     }
24 }
```

Channels

- Channel direction “send-only” and “read-only”
- Type of channel changes
- Compile time error if read a send-only channel, and vice-versa

```
1 var wg sync.WaitGroup
2
3 func main() {
4     wg.Add(2)
5     var chNum chan int = make(chan int, 100)
6     go producer(chNum)
7     go consumer(chNum)
8     wg.Wait()
9 }
10
11 func producer(ch chan<- int) {
12     defer wg.Done()
13     for n := range 1000 {
14         ch <- n
15     }
16     close(ch)
17 }
18
19 func consumer(ch <-chan int) {
20     defer wg.Done()
21     for {
22         num, ok := <-ch
23         if !ok {
24             return
25         }
26         fmt.Println(num)
27     }
28 }
```

The diagram consists of two cyan-colored arrows. One arrow originates from the text "Send-only" at the top right and points to the line "ch <- n" in the producer function. The other arrow originates from the text "Read-only" below it and points to the line "num, ok := <-ch" in the consumer function.

Channels

- Multi-reader multi-writer example

```
1 var wg sync.WaitGroup
2
3 func main() {
4     wg.Add(10)
5     ch := make(chan string, 100)
6     go writer(ch, "A")
7     go writer(ch, "B")
8     go writer(ch, "C")
9     go writer(ch, "D")
10    go writer(ch, "E")
11    go reader(ch)
12    go reader(ch)
13    go reader(ch)
14    go reader(ch)
15    go reader(ch)
16    wg.Wait()
17    fmt.Println("\n")
18 }
```

```
1 func writer(ch chan string, letter string) {
2     defer wg.Done()
3     for range 100 {
4         ch <- letter
5     }
6     ch <- "DONE"
7 }
8
9 func reader(ch chan string) {
10    defer wg.Done()
11    for {
12        letter := <-ch
13        if letter == "DONE" {
14            return
15        }
16        fmt.Println(letter)
17    }
18 }
```

```
$ go run .
AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAEBCDDDDDBCAEBBBBBDCA
BBBBBBBCEDCCCCDCCCCCEDBCDCDD
DBCEBBBBBBBBBEBCEBBBBCEDCCCC
CBCEDCCCCCCCCBCDEBCDEBBBBBBBB
BCDEDDBCEDBCEBBDBBBBBBBBBBDCE
BBBBBBCEDBECDCCCCCCCCEDEBCD
EBCDEEEEEEBCDEBCDEBCDEBBB
BBBBBBBDCEBDCBEEEEEEEDCBDE
CCDEBDCEBCDEBCDEBCDEBCBEEEEE
EEBCDEBCDEDDDDDDDDDBCEEEDEEE
EEEEBCDEBCDBBEEDDDDDDDDECED
CEDCCCCDCCCCCCDECDECCCCCCCC
CCCDECDEEEEEECCFFFFDEDEDFFED
DDDDDDDDDDDDDDDDDDDDDDDDDD
```

Channels

- **Closing channels → close(). Closed = no more values sent.**
 - Reading a closed channel → OK while channel has values, after gives zero value
 - Writing a closed channel → panic
 - Closing a closed channel → panic
 - Reader detecting a closed channel → val, ok := <-myChannel
- **Not necessary to close channel, but good way to indicate end-of-transmission EOT**

Extra: Runtime Type Comparison

```
1 func main() {
2     var bird interface{} = ""
3     if bird == "" {
4         fmt.Println("true branch") // taken
5     } else {
6         fmt.Println("false branch")
7     }
8 }
```

```
1 func main() {
2     var bird interface{} = ""
3     switch birdVal := bird.(type) {
4     case string:
5         if birdVal == "" {
6             fmt.Println("true branch") // taken
7         } else {
8             // unexpected value
9         }
10    default:
11        panic("bird really should be a string.")
12    }
13 }
```

- A *runtime* type check is performed first, if types not equal then always false, otherwise do regular comparison

Practice!

- <https://github.com/calvincramer/golang-training-3>



Training 3 - Concurrency

Calvin Lei-Cramer
May 2025



What are we covering today?

Panic and recover,
goroutines and
concurrency, channels

Panic! Recover.

- What are panics actually? Are they exceptions?

- What triggers panics?

- Array/slice index out of bounds
- Divide by zero
- Calling `panic()`
- Nil pointer deference
- Type assertion (without using 'ok' second return value) failing

```
1 func main() {  
2     _ = []int{1, 2, 3}[10]  
3  
4     div := 0  
5     _ = 5 / div  
6  
7     panic("Something bad happened!")  
8  
9     var numPtr *int = nil  
10    _ = *numPtr  
11  
12    var something any = struct{ A int }{A: 1000}  
13    _ = something.(string)  
14 }
```

3

On a panic, all DEFERRED functions in current function executed. Very similar to exception. Then the caller of the current function has its DEFERRED functions called.

If panic is not handled in deferred functions then it will cause program to terminate. Same as exceptions.

Panics and exceptions are SIMILAR in how they are INVOKED, but Golang and other language are different in how you are allowed to HANDLE exceptions, and where normal execution can RECOVER from.

Anonymous struct

Panic! Recover.

- How to catch panics?
- Recover() global
 - Returns value passed to panic() if panicking
 - Otherwise returns nil
- Named return values, defer, recover()

```
$ go run _  
25 / 2 = 12  
43 / 0 = 9223372036854775807  
-96 / 0 = -9223372036854775808  
$
```

```
1 func SafeDivide(num, div int) (result int) {  
2     defer func() {  
3         if recover() != nil {  
4             // Set to near infinity  
5             if num < 0 {  
6                 result = math.MinInt  
7             } else {  
8                 result = math.MaxInt  
9             }  
10        }  
11    }()  
12    result = num / div // may cause panic  
13    return  
14 }
```

4

Deferred function may change named return values in order to return an appropriate value that indicates an error, or any adjustment that's necessary because of the panic.

Keep in mind, any deferred function call may be ran in the context of a panic happening. SHARED USE OF NORMAL EXECUTION AND PANIC EXECUTION.

Panic! Recover.

- Pattern to recover normal execution flow like try/catch

Python

```
1 def main():
2     print("Before")
3     try:
4         badFunction()
5     except Exception as e:
6         pass
7     print("After")
```

```
$ go run _
Before
Log panic: something bad happened in badFunc()
After
```

Golang

```
1 func handlePanic(action func()) {
2     defer func() {
3         if val := recover(); val != nil {
4             fmt.Println("Log panic:", val)
5         }
6     }()
7     action()
8 }
9
10 func main() {
11     fmt.Println("Before")
12     handlePanic(badFunc)
13     fmt.Println("After")
14 }
```

In Golang the try/catch forces the function that recovers from the panic to return.

Panic! Recover.

- What happens if panic, then panic again in defer function?

```
3 func main() {  
4     defer func() {  
5         panic("panic #2")  
6     }()  
7     panic("panic #1")  
8 }
```

```
$ go run _  
panic: panic #1  
        panic: panic #2  
  
goroutine 1 [running]:  
main.main.func1()  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x25  
panic({0x46f5a0?, 0x499a78?})  
    /usr/local/go/src/runtime/panic.go:792 +0x132  
main.main()  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x3e  
exit status 2  
$
```

6

Both panics printed

Execution trace shows line numbers of both panics

Can panic while panicking.

Panic! Recover.

```
3 func doStuff() {
4     defer func() {
5         if val := recover(); val != nil {
6             fmt.Println("Recovered panic:", val)
7         }
8     }()
9     defer func() {
10         panic("panic #2")
11     }()
12     panic("panic #1")
13 }
14
15 func main() {
16     fmt.Println("Before")
17     doStuff()
18     fmt.Println("After")
19 }
```

- **Panic, panic, recover?**

```
$ go run .
Before
Recovered panic: panic #2
After
$
```

Panic overrides any current panic value.

Keep panic recover simple.

Golang Idioms

- **Idiomatic Go, mantras, patterns, group think**
- **“Programming in a language vs programming into a language” – Code Complete by Steve McConnell**
- **Error handling → errors are values, don’t code by exception**
- **Defer**
- **Goroutines and concurrency models**

```
3 func fetch(url string) (body string, err error) {
4     resp, err := http.Get(url)
5     if err != nil {
6         return
7     }
8     defer resp.Body.Close()
9     bodyBytes, err := io.ReadAll(resp.Body)
10    if err != nil {
11        return
12    }
13    body = string(bodyBytes)
14    return
15 }
16
17 func main() {
18     resp, err := fetch("https://www.google.com")
19     if err != nil {
20         fmt.Println("error fetching URL -", err)
21         return
22     }
23     fmt.Println(resp)
24 }
```

8

Pythonic python

Programming IN a language means thinking in terms of what the language offers. This may limit how you think and solve problems, for example working in C.

Programming INTO a language means thinking what you want to accomplish first, then accomplishing that with the tools in the language.

Goroutines

- **What are goroutines?**

- Lightweight application-level threads managed by Go runtime
- Multiple goroutines may run on a single OS-level thread (concurrency) or multiple OS-level threads (parallelism). Go runtime handles this.
- OS-level threads may be scheduled to run on arbitrary CPU core by OS.
- All goroutines share the same address space
- Synchronization → no IPC required. Can use shared memory (mutex, locks, barriers) for synchronization.

- **Spawn goroutine? → go funcCall(args...)**

9

Goroutines not guaranteed to run in parallel on multiple CPUs.

Goroutines Example

```
1 func child() {
2     for {
3         var msSleep int = rand.Intn(2000) + 500
4         time.Sleep(time.Duration(msSleep) * time.Millisecond)
5         fmt.Println("Child: are we there yet?")
6     }
7 }
8
9 func parent() {
10    for {
11        var msSleep int = rand.Intn(3000) + 1000
12        time.Sleep(time.Duration(msSleep) * time.Millisecond)
13        fmt.Println("Parent: 10 more minutes")
14    }
15 }
16
17 func main() {
18     go child()
19     go parent()
20     for { time.Sleep(time.Second) }
21 }
```

```
$ []
```

10

The main() function is running as the “MAIN” goroutine.

Once the main thread exits then the application terminates. That is why we need to wait forever in the main function to allow the other goroutines to run. The program is terminated with a CTRL+C

Goroutines

- Join goroutine? `pthread_join?`
- Nope! Will need to use other tools
- `Sync.Mutex`



```
1 var childLock sync.Mutex
2
3 func child() {
4     fmt.Println("Getting lunch...")
5     time.Sleep(time.Second)
6     fmt.Println("Getting homework...")
7     time.Sleep(time.Second)
8     fmt.Println("Getting socks...")
9     time.Sleep(time.Second)
10    fmt.Println("Getting jacket...")
11    time.Sleep(time.Second)
12    childLock.Unlock()
13 }
14
15 func main() {
16     childLock.Lock()
17     go child()
18     // Wait for child
19     childLock.Lock()
20     fmt.Println("Walking to school!")
21 }
```

11

Spawning a goroutine does not give an ID of the new goroutine. This is intentional design.

Goroutines

- **Sync.WaitGroup → wait for a group of goroutines**
- `wg.Add(n)` → add n goroutines to wait for (usually 1 or all)
- `wg.Done()` → call by goroutine when done (subtracts 1)
- `wg.Wait()` → block until no goroutines running (wait for 0)

```
1 func main() {  
2     var wg sync.WaitGroup  
3     const numChildren = 5  
4  
5     childDoHomework := func(num int) {  
6         for {  
7             fmt.Printf("Child %d working...\n", num)  
8             wait := time.Duration(rand.Intn(2000)+500)  
9             time.Sleep(time.Millisecond * wait)  
10            if rand.Float64() <= 0.25 {  
11                break  
12            }  
13        }  
14        fmt.Printf("Child %d done\n", num)  
15    }  
16    wg.Done()  
17  
18    wg.Add(numChildren)  
19    for i := range numChildren {  
20        go childDoHomework(i)  
21    }  
22    wg.Wait()  
23    fmt.Println("Everyone is done!")  
24 }  
25
```

12

Goroutines

- Handling synchronization of global data via sync primitives

```
1 func main() {  
2     var total int = 0  
3     var wg sync.WaitGroup  
4     wg.Add(1000000)  
5     for range 1000000 {  
6         go func() {  
7             defer wg.Done()  
8             total += 1  
9         }()  
10    }  
11    wg.Wait()  
12    fmt.Println(total)  
13 }
```

```
$ for i in {1..100}; do go run _; done  
922539  
927714  
929655  
925084  
925332  
923226  
921020  
924350  
919063  
918562  
917953  
911826  
915570  
923962  
915425  
913478
```

13

Golang will NOT protect you from data races.

Golang will NOT warn you about possible data races.

Channels

- **What are channels?**
 - bidirectional communication mechanism between goroutines
- **make(chan TYPE)**
- **Can send/receive things on channels**
 - Send: myChannel <- value
 - Receive: var := <-myChannel
- **Channels include the type of data transferred → chan int, chan MyStruct**

Channels

```
5 func main() {  
6     postOffice := make(chan string)  
7     go penPal(postOffice)  
8     message := "Hi there pen pal!"  
9     fmt.Println("main sending message")  
10    postOffice <- message  
11    reply := <-postOffice  
12    fmt.Println("main received a reply!", reply)  
13 }
```

```
15 func penPal(messageChan chan string) {  
16     message := <-messageChan  
17     fmt.Println("Pen pal received a message!")  
18     newMessage := "Cool message! " + message  
19     messageChan <- newMessage  
20 }
```

```
$ go run _  
main sending message  
Pen pal received a message!  
main received a reply! Cool message! Hi there pen pal!  
$
```

16

Point out:

- making channel
- reading value from channel
- reading value from channel blocks until message is available
- writing to channel

Channels

- What if both try to send messages at same time?

```
1 func main() {
2     postOffice := make(chan string)
$ go run _
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:6 +0x70

goroutine 5 [chan send]:
main.penPal(...)
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:10
created by main.main in goroutine 1
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x5f
exit status 2
$
```

Channels

- What if both wait to receive messages at same time?

```
1 func main() {  
$ go run _  
fatal error: all goroutines are asleep - deadlock!  
  
goroutine 1 [chan receive]:  
main.main()  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:8 +0x7b  
  
goroutine 18 [chan receive]:  
main.penPal(0x0?)  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:13 +0x1e  
created by main.main in goroutine 1  
    /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x66  
exit status 2  
$
```

Channels

- Channels are preferred over synchronization primitives
- “Don’t communicate by sharing memory, share memory by communicating”
- Synchronization primitives are shared data and sync mechanism
- Channels have no shared data, send/receive state with other goroutines
- Channels are similar to distributed computing, for example communicating between multiple hosts over HTTP/REST

19

Distributed computed can get ugly really fast.

Consider higher level abstractions than channels if building complex systems.

Channels

- **Unbuffered vs buffered channels**
- **Unbuffered = channel has no capacity**
 - Think of unbuffered as writer literally handing message to reader
 - Aka **synchronous** communication
 - `make(chan int)`
 - `make(chan int, 0)`
- **Buffered = channel has capacity → `make(chan int, 100)`**
 - Aka **asynchronous** communication

Channels

- When do channels block?

	Read	Write
Unbuffered	Blocks until value available	Blocks until value is read
Buffered	Blocks if channel empty	Blocks if channel is full

Channels

- Buffered channel example

```
$ go run _  
.....CCCCCCCCCCCCCCCCCCCCCCCCCCCC.....  
$
```

```
1 var wg sync.WaitGroup  
2  
3 func main() {  
4     ch := make(chan bool, 100)  
5     wg.Add(2)  
6     go producer(ch)  
7     go consumer(ch)  
8     wg.Wait()  
9     fmt.Println("\n")  
10 }  
11  
12 func producer(ch chan bool) {  
13     defer wg.Done()  
14     for range 1000 {  
15         ch <- false  
16         fmt.Print(".")  
17     }  
18     ch <- true  
19 }  
20  
21 func consumer(ch chan bool) {  
22     defer wg.Done()  
23     for {  
24         val := <-ch  
25         if val == true {  
26             break  
27         }  
28         fmt.Print("C")  
29     }  
30 }
```

Channels

- Same example but UNBUFFERED

```
$ go run .
1 var wg sync.WaitGroup
2
3 func main() {
4     ch := make(chan bool, 0)
5     wg.Add(2)
6     go producer(ch)
7     go consumer(ch)
8     wg.Wait()
9     fmt.Println("\n")
10 }
11
12 func producer(ch chan bool) {
13     defer wg.Done()
14     for range 1000 {
15         ch <- false
16         fmt.Println(".")
17     }
18     ch <- true
19 }
20
21 func consumer(ch chan bool) {
22     defer wg.Done()
23     for {
24         val := <-ch
25         if val == true {
26             break
27         }
28         fmt.Println("C")
29     }
30 }
```

23

Channels

- Really should treat channels as unidirectional most of the time.
- If need bidirectional then just use two channels. Especially for buffered channels.

```
1 func main() {  
2     fruitSalad := make(chan bool, 10)  
3     pancake := make(chan bool, 10)  
4     wg.Add(2)  
5     go chef1(fruitSalad, pancake)  
6     go chef2(fruitSalad, pancake)  
7     wg.Wait()  
8 }
```

```
1 func chef1(fruitSalad chan bool, pancake chan bool) {  
2     defer wg.Done()  
3     for range 1000 {  
4         <-fruitSalad // chef1 likes fruit salad  
5         pancake <- true // chef1 makes pancake  
6     }  
7 }  
  
1 func chef2(fruitSalad chan bool, pancake chan bool) {  
2     defer wg.Done()  
3     for range 1000 {  
4         fruitSalad <- true // chef2 makes fruit salad  
5         <-pancake // chef2 likes pancakes  
6     }  
7 }
```

Channels

- Select control flow
- wait for multiple channels
- Useful for timeout and stopping conditions

```
1 func main() {
2     chNumber := time.Tick(time.Second)
3     chFizz := time.Tick(time.Second * 3)
4     chBuzz := time.Tick(time.Second * 5)
5     chFizzBuzz := time.Tick(time.Second * 3 * 5)
6     chTimeout := time.After(time.Second * 20)
7     num := 1
8
9     for {
10         select {
11             case <-chNumber:
12                 fmt.Println(num)
13                 num += 1
14             case <-chFizz:
15                 fmt.Println("Fizz")
16             case <-chBuzz:
17                 fmt.Println("Buzz")
18             case <-chFizzBuzz:
19                 fmt.Println("Fizz Buzz")
20             case <-chTimeout:
21                 return
22         }
23     }
24 }
```

25

time.Tick() returns a channel which periodically sends a message of the current time

time.After() returns a channel that sends a single message after a specified duration.

Channels

- Channel direction “send-only” and “read-only”
- Type of channel changes
- Compile time error if read a send-only channel, and vice-versa

```
1 var wg sync.WaitGroup
2
3 func main() {
4     wg.Add(2)
5     var chNum chan int = make(chan int, 100)
6     go producer(chNum)
7     go consumer(chNum)
8     wg.Wait()
9 }
10
11 func producer(ch chan<- int) {
12     defer wg.Done()
13     for n := range 1000 {
14         ch <- n
15     }
16     close(ch)
17 }
18
19 func consumer(ch <-chan int) {
20     defer wg.Done()
21     for {
22         num, ok := <-ch
23         if !ok {
24             return
25         }
26         fmt.Println(num)
27     }
28 }
```

Send-only

Read-only

Channels

- Multi-reader multi-writer example

```
1 var wg sync.WaitGroup
2
3 func main() {
4     wg.Add(10)
5     ch := make(chan string, 100)
6     go writer(ch, "A")
7     go writer(ch, "B")
8     go writer(ch, "C")
9     go writer(ch, "D")
10    go writer(ch, "E")
11    go reader(ch)
12    go reader(ch)
13    go reader(ch)
14    go reader(ch)
15    go reader(ch)
16    wg.Wait()
17    fmt.Println("\n")
18 }
```

```
1 func writer(ch chan string, letter string) {
2     defer wg.Done()
3     for range 1..100 {
4         ch <- letter
5     }
6     ch <- "DONE"
7 }
8
9 func reader(ch chan string) {
10    defer wg.Done()
11    for {
12        letter := <-ch
13        if letter == "DONE" {
14            return
15        }
16        fmt.Println(letter)
17    }
18 }
```

```
$ go run _
AAAAAAA.....AAAAAAA.....AAAAAAA.....AAAAAAA
AAAAAAA.....AAAAAAA.....AAAAAAA.....AAAAAAA
AAAAAAA.....AAAAAAA.....AAAAAAA.....AAAAAAA
AAAAAAAEBCD.....DBCAEBABBBBDC
BBBBBBBCEDCCCCD.....CCCCCEDBCD
DBCEBBBBBBB.....BCEBBBBBCEDCCC
CBCEDCCCCCCC.....BCDEBCDEBBBBBBB
BCDEBCEDBC.....EBBBBBDBBBBBDDCE
BBBBBCEDBECD.....BCDCCCCCCCCDEBCD
EBCDEEE.....BCEDEBCDEBCDEBBB
BBBBBBBDCEBDC.....EEDCBDE
CCDEBCEDBC.....DEBCDEBCBEEEEEEE
EBCDEBCDEDDDDDDDD.....DBCEEEDEEE
EEEBCDEBCDBBEEDDDDDDDDECED
CEDCCCCD.....CCCCCDEDECCCCCCCC
CCCDECDEEEEEE.....CEEEDEDEDEEE
DDDDDDDDDDDDDDDDDDDDDD
```

Channels

- **Closing channels → close(). Closed = no more values sent.**
 - Reading a closed channel → OK while channel has values, after gives zero value
 - Writing a closed channel → panic
 - Closing a closed channel → panic
 - Reader detecting a closed channel → val, `ok := <-myChannel`
- **Not necessary to close channel, but good way to indicate end-of-transmission EOT**

28

Golang will garbage collect non-closed channels

ONLY THE SENDER SHOULD CLOSE CHANNEL

Since closing a closed channel gives panics, be careful with multiple writers.

Extra: Runtime Type Comparison

```
1 func main() {
2     var bird interface{} = ""
3     if bird == "" {
4         fmt.Println("true branch") // taken
5     } else {
6         fmt.Println("false branch")
7     }
8 }
```

```
1 func main() {
2     var bird interface{} = ""
3     switch birdVal := bird.(type) {
4     case string:
5         if birdVal == "" {
6             fmt.Println("true branch") // taken
7         } else {
8             // unexpected value
9         }
10    default:
11        panic("bird really should be a string.")
12    }
13 }
```

- A ***runtime*** type check is performed first, if types not equal then always false, otherwise do regular comparison

29

Very interesting to see this behavior of Golang,
since it checks types at compile time mostly.

Practice!

- <https://github.com/calvincramer/golang-training-3>



30