Golang channels

December 6, 2013

Golang has built-in instruments for writing concurrent programs. Placing a go statement before a function call starts the execution of that function as an independent concurrent thread in the same address space as the calling code. Such thread is called `goroutine` in Golang. Here I should mention that concurrently doesn't always mean in parallel. Goroutines are means of creating concurrent architecture of a program which could possibly execute in parallel in case the hardware allows it. There is a great talk on that topic Concurrency is not parallelism.

Let's start with an example of a goroutine:

```go
func main() {
     // Start a goroutine and execute println concurrently
     go println("goroutine message")
     println("main function message")
}
```

This program will print `main function message` and **possibly** `goroutine message`. I say **possibly** because spawning a goroutine has some peculiarities. When you start a goroutine the calling code (in our case it is the `main` function) doesn't wait for a goroutine to finish, but continues running further. After calling a `println` the main function ends its execution and in Golang it means stopping of execution of the whole program with all spawned goroutines. But before it happens our goroutine could possibly finish executing its code and print the `goroutine message` string.

As you understand there must be some way to avoid such situations. And for that there are **channels** in Golang.

Channels basics

Channels serve to synchronize execution of concurrently running functions and to provide a mechanism for their communication by passing a value of a specified type. Channels have several characteristics: the type of element you can send through a channel, capacity (or buffer size) and direction of communication specified by a `<-` operator. You can allocate a channel using the built-in function make:

```go
i := make(chan int)       // by default the capacity is 0
s := make(chan string, 3) // non-zero capacity

r := make(<-chan bool)          // can only read from
w := make(chan<- []os.FileInfo) // can only write to
```

Channels are first-class values and can be used anywhere like other values: as struct elements, function arguments, function returning values and even like a type for another channel:

```
// a channel which:
//  - you can only write to
//  - holds another channel as its value
c := make(chan<- chan bool)

// function accepts a channel as a parameter
func readFromChannel(input <-chan string) {}

// function returns a channel
func getChannel() chan bool {
     b := make(chan bool)
     return b
}
```

For writing and reading operations on channel there is a <- operator. Its position relatively to the channel variable determines whether it will be a read or a write operation. The following example demonstrates its usage, but I have to warn you that this code **does not work** for some reasons described later:

```
func main() {
     c := make(chan int)
     c <- 42    // write to a channel
     val := <-c // read from a channel
     println(val)
}
```

Now, as we know what channels are, how to create them and perform basic operations on them, let's return to our very first example and see how channels can help us.

```
func main() {
     // Create a channel to synchronize goroutines
     done := make(chan bool)

     // Execute println in goroutine
     go func() {
          println("goroutine message")

          // Tell the main function everything is done.
          // This channel is visible inside this goroutine because
          // it is executed in the same address space.
          done <- true
     }()

     println("main function message")
     <-done // Wait for the goroutine to finish
}
```

This program will print both messages without any possibilities. Why? `done` channel has no buffer (as we did not specify its capacity). All operations on unbuffered channels block the execution until both sender and receiver are ready to communicate. That's why unbuffered channels are also called synchronous. In our case the reading operation `<-done` in the main function will block its execution until the goroutine will write data to the channel. Thus the program ends only after the reading operation succeeds.

In case a channel has a buffer all read operations succeed without blocking if the buffer is not empty, and write operations - if the buffer is not full. These channels are called asynchronous. Here is an example to demonstrate the difference between them:

```go
func main() {
	message := make(chan string) // no buffer
	count := 3

	go func() {
		for i := 1; i <= count; i++ {
			fmt.Println("send message")
			message <- fmt.Sprintf("message %d", i)
		}
	}()

	time.Sleep(time.Second * 3)

	for i := 1; i <= count; i++ {
		fmt.Println(<-message)
	}
}
```

In this example `message` is a synchronous channel and the output of the program is:

```
send message
// wait for 3 seconds
message 1
send message
send message
message 2
message 3
```

As you see after the first write to the channel in the goroutine all other writing operations on that channel are blocked until the first read operation is performed (about 3 seconds later).

Now let’s provide a buffer to out `message` channel, i.e. the creation line will look as `message := make(chan string, 2)`. This time the output will be the following:

```
send message
send message
send message
// wait for 3 seconds
message 1
message 2
message 3
```

Here we see that all writing operations are performed without waiting for the first read for the buffer of the channel allows to store all three messages. By changing channels capacity we can control the amount of information being processed thus limiting throughput of a system.

Deadlock

Now let's get back to our not working example with read/write operations.

```go
func main() {
	c := make(chan int)
	c <- 42    // write to a channel
	val := <-c // read from a channel
	println(val)
}
```

On running you'll get this error (details will differ):

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
	/fullpathtofile/channelsio.go:5 +0x54
exit status 2
```

The error you got is called a **deadlock**. This is a situation when two goroutines wait for each other and non of them can proceed its execution. Golang can detect deadlocks in runtime that's why we can see this error. This error occurs because of the blocking nature of communication operations.

The code here runs within a single thread, line by line, successively. The operation of writing to the channel (`c <- 42`) blocks the execution of the whole program because, as we remember, writing operations on a synchronous channel can only succeed in case there is a receiver ready to get this data. And we create the receiver only in the next line.

To make this code work we should had written something like:

```go
func main() {
	c := make(chan int)

	// Make the writing operation be performed in
	// another goroutine.
	go func() {
	   c <- 42
	}()
	val := <-c
	println(val)
}
```

Range channels and closing

In one of the previous examples we sent several messages to a channel and then read them. The receiving part of code was:

```go
for i := 1; i <= count; i++ {
	fmt.Println(<-message)

}
```

In order to perform reading operations without getting a deadlock we have to know the exact number of sent messages (`count`, to be exact), because we cannot read more then we sent. But it’s not quite convenient. It would be nice to be able to write more general code.

In Golang there is a so called **range expression** which allows to iterate through arrays, strings, slices, maps and channels. For channels, the iteration proceeds until the channel is closed. Consider the following example (does not work for now):

```go
func main() {
	message := make(chan string)
	count := 3

	go func() {
		for i := 1; i <= count; i++ {
			message <- fmt.Sprintf("message %d", i)
		}
	}()

	for msg := range message {
		fmt.Println(msg)
	}
}
```

Unfortunately this code does not work now. As was mentioned above the `range` will work until the channel is closed explicitly. All we have to do is to close the channel with a close function. The goroutine will look like:

```go
go func() {
	for i := 1; i <= count; i++ {
		message <- fmt.Sprintf("message %d", i)
	}
	close(message)
}()
```

Closing a channel has one more useful feature - reading operations on closed channels do not block and always return default value for a channel type:

```go
done := make(chan bool)
close(done)

// Will not block and will print false twice
// because it’s the default value for bool type
println(<-done)
println(<-done)
```

This feature may be used for goroutines synchronization. Let’s recall one of our examples with synchronization (the one with `done` channel):

```go
func main() {

	done := make(chan bool)
```

```

    go func() {
        println("goroutine message")

        // We are only interested in the fact of sending itself,
        // but not in data being sent.
        done <- true
    }()

    println("main function message")
    <-done
}
```

Here the `done` channel is only used to synchronize the execution but not for sending data. There is a kind of pattern for such cases:

```
func main() {
    // Data is irrelevant
    done := make(chan struct{})

    go func() {
        println("goroutine message")

        // Just send a signal "I'm done"
        close(done)
    }()

    println("main function message")
    <-done
}
```

As we close the channel in the goroutine the reading operation does not block and the main function continues to run.

Multiple channels and select

In real programs you'll probably need more than one goroutine and one channel. The more independent parts are - the more need for effective synchronization. Let's look at more complex example:

```
func getMessagesChannel(msg string, delay time.Duration) <-chan string {
    c := make(chan string)
    go func() {
        for i := 1; i <= 3; i++ {
            c <- fmt.Sprintf("%s %d", msg, i)
            // Wait before sending next message
            time.Sleep(time.Millisecond * delay)
        }
    }()
    return c
}

```

```go
func main() {
	c1 := getMessagesChannel("first", 300)
	c2 := getMessagesChannel("second", 150)
	c3 := getMessagesChannel("third", 10)

	for i := 1; i <= 3; i++ {
		println(<-c1)
		println(<-c2)
		println(<-c3)
	}
}
```

Here we have a function that creates a channel and spawns a goroutine which will populate the channel with three messages in a specified interval. As we see the third channel `c3` has the least interval, thus we except its messages to appear prior to others. But the output will be the following:

```
first 1
second 1
third 1
first 2
second 2
third 2
first 3
second 3
third 3
```

Obviously we got a successive output. That is because the reading operation on the first channel blocks for `300` milliseconds for each loop iteration and other operations must wait. What we actually want is to read messages from all channels as soon as they are any.

For communication operations on multiple channels there is a select statement in Golang. It's much like the usual `switch` but all cases here are communication operations (both reads and writes). If the operation in `case` can be performed than the corresponding block of code executes. So, to accomplish what we want, we have to write:

```go
for i := 1; i <= 9; i++ {
	select {
	case msg := <-c1:
		println(msg)
	case msg := <-c2:
		println(msg)
	case msg := <-c3:
		println(msg)
	}
}
```

Pay attention to the number `9`: for each of the channels there were 3 writing operations, that's why I have to perform 9 loops of the select statement. In a program which is meant to run as a daemon there is a common practice to run `select` in an infinite loop, but here I'll get a deadlock if I'll run one.

Now we get the expected output, and non of reading operations block others. The output is:

```
first 1
second 1
third 1 // this channel does not wait for others
third 2
third 3
second 2
first 2
second 3
first 3
```

Conclusion

Channels is a very powerful and interesting mechanism in Golang. But in order to use them effectively you have to understand how they work. In this article I tried to explain the very necessary basics. For further learning I recommend you look at the following:

- Concurrency is not parallelism - early mentioned talk from Rob Pike
- Go Concurrency Patterns
- Advanced Go Concurrency Patterns

Contact me via email