An Intro to Concurrency Patterns in Go

By **Laura Frank**, blog.codeship.com
February 4th, 2016

UPDATE: You now can read part 2 of this article here.

If you're new to the Go programming language, it's pretty likely that one of the first things you'll hear about is Go's baked-in concurrency support.

If you're curious about the basic concurrency patterns in Go, we'll cover what concurrency is, and then go over a couple examples to help you understand the pattern. We *won't* talk about some of the more in-depth topics around concurrency, like the sync library, using mutexes, etc. Instead, we'll focus on the building blocks to help you get started.

What Is Concurrency?

Concurrency is a design pattern that allows for complex systems with multiple components. It means that processes (in the general sense) can start, run, and complete simultaneously. This doesn't have anything to do with the actual execution of the code, just the design of the system.

For example, at the gym during a training circuit, I might be using a bunch of different things, but I'm only ever using one at a time. But the gym itself is designed to accommodate someone else using the medicine balls when I'm doing inverted crunches. It wouldn't matter to the system if I finished before the other person did, or if maybe I wanted to use the medicine balls first.

It's also important to note that concurrency and parallelism are not the same thing.

While *concurrency* refers to the design of the system, *parallelism* relates to the execution. It's possible to have a concurrent system that runs on a single processor. However, when you can combine a well-designed concurrent system with parallelism, you can

see significant performance gains.

There are entire talks and papers about the differences between parallelism and concurrency, and if you're interested, I recommend checking those out for a deeper understanding.

As a quick takeaway, remember that concurrent systems are designed to deal with many things happening at the same time, and parallelism is about doing lots of things at once.

Concurrency in Go

You can write a very simple concurrent program in Go with just two things:

- goroutines
- channels

Lots of people are drawn to the language specifically for this reason (along with a lot of other great things!). Go makes concurrency really easy to implement without much overhead.

goroutines

Using goroutines allows your program to complete small units of work in a concurrent way.

Very simply put, a goroutine is an independently executing function within your program. They are very similar to a lightweight thread, but goroutines are not threads.

They're much cheaper, using very little memory and having a pretty small initial stack size, plus many goroutines can be multiplexed onto a single thread. In fact, a single-threaded program might have hundreds or thousands of goroutines running at the same time. Because goroutines run in the same address space, their execution must be synchronized.

A goroutine is denoted simply by adding the word 'go' in front of the function call. For example, for some function f:

```
func f(a string, b string) {
    fmt.Printf("%s %s", a, b)
}
```

We can call this function in a new goroutine like this:

```
go f("foo", "bar")
```

If the arguments to function f needed evaluation, they would be evaluated in the currently running goroutine, but the execution of the function happens in the separate routine that we asked for.

Channels

But goroutines alone won't get you very far. We need to have some mechanism that allows the goroutines to communicate with each other, because the access of shared memory must be coordinated.

A channel is a typed communication conduit to send and receive messages, and we can use a channel to let two goroutines communicate with one another.

Make a channel like this: `make (chan int)`

Then, you can send or receive from a channel by using the `<-` operator, where the arrow shows the direction of the message flow.

To send to a channel, say:

```
chan <- 1 // sends 1 to the channel
```

Then to read from that channel, flip it around:

```
v = <- chan // reads from chan, and assigns to v
```

You could opt to use a buffered channel, as well. When creating the channel, just add a second argument of the size of the channel:

```
make (chan int, 16)
```

Communicating between two goroutines using a channel

Let's start out with a really simple function that prints out a message with an index — not using goroutines or channels — and build up from there.

```
func makeMessage(s string) {
    for i := 0; i < 10; i++ {
        fmt.Sprintf("I wanted to tell you '%s' for the %dth
time", s, i)
        time.Sleep(500 * time.Millisecond)
    }
}

func main() {
 makeMessage("Hello world")
}
```

But let's imagine we wanted to print something other than just **"Hello world"**, maybe something that is computed in a different function, running in a separate goroutine.

For this, we can create a channel in the main function. We'll also have to pass that channel to our message function.

```
func makeMessage(s string, channel chan string) {
    for i := 0; i < 5; i++ {
        channel <- fmt.Sprintf("I wanted to say '%s' for the
%dth time", s, i) // write the message to the channel
        time.Sleep(500 * time.Millisecond)// to show that
the main function will wait for the channel to be filled
again
    }
}

func main() {
    channel := make(chan string)// create channel

    go makeMessage("Hello!", channel) // run this in a
separate goroutine

    for i := 0; i < 5; i++ {
        fmt.Printf("Hey there, %s\n", <-channel) // read
from channel and print out message
    }

    fmt.Println("Cool, that's all I wanted to say")
}
```

When **<-** channel (the read from channel) is executed from the main function, it waits until a value is ready. Similarly, when we write to the receiver channel in the **makeMessage** function, we have to wait for the channel to be empty in order to receive the message.

This also requires waiting and synchronization. Both sides of a channel will block, until a sender and receiver are both ready for something to be passed.

An important thing to note is that we're using an unbuffered channel in this example.

This behavior is different for buffered channels.

Using buffered channels

The behavior of inter-goroutine communication changes a little when using buffered channels.

Using unbuffered channels is a great fit for simple workflows, but if you need to process large datasets and bulk operations, it is often better to use a buffered channel. This allows different parts of your application to function independently, regardless of whether a receiver is currently available for the message handoff.

Creating a buffered channel is exactly the same as creating an unbuffered one, except the channel size is also passed to the initialization, as shown in an earlier example.

This simply means that the channel created can store any number of messages up to the maximum size. Using a buffered channel does require more complicated message handling, since your senders and receivers will only block under a specific circumstance.

It also means that graceful shutdowns are more complicated due to the intermittent state of your channel buffers. You should measure your workloads and tune buffer sizes as well as sender and receiver workers accordingly.

A great example of when to use a buffered channel is when taking a stream of data and bulk processing it in some way, such as uploading it to a block storage service.

In this case, the data stream could feed directly into a buffered channel. A bulk handler routine would then read the channel size. Every *N* seconds, or when over *M* messages are in the channel, that routine would read the contents and process them.

When designing such a system, one thing to watch out for is to ensure that the number *M* is appropriately lower than the channel buffer size in such a way that it is unlikely to fill up before being emptied. If not, the sender component may block, potentially keeping web requests open or using significant memory in some other goroutine

within the application.

Resources and Further Practice

If you're more of a visual learner, it might be easier to grasp concurrency by looking at some models of the behavior. Ivan Daniluk wrote an awesome post about visualizing concurrency in Go, and there are some really awesome animations that show how each goroutine executes, as well as the interaction between goroutines via channels.

Concurrency is a powerful pattern in software design. By having a concurrent program that utilizes resources efficiently, it's possible to increase throughput in very significant ways, especially when introducing parallelism (but not always!).

If you want a little more practice with concurrency in Go, there are a couple really great talks that range from introductory to pretty advanced. If you're just starting to wrap your head around concurrency, I would also recommend working through the Golang Tour exercises, as well as the exercises on exercism.io.