Golang: Features and details

Go is an open-source programming language that makes it easy to build simple, reliable, and efficient software.

Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection.

Go's overarching goal is simplicity.

Memory Management

garbage collection offer benefits like:
- increased security
- better portability across operating systems
- less code to write
- runtime verification of code
- bounds checking of arrays

Go's garbage collector is designed to prioritize latency and avoid stop-the-world pauses, which is particularly important in servers. This may come with a higher CPU cost, but in a horizontally scalable architecture, this is easily solved by adding more machines.

Collector Behavior

When a collection starts, the collector runs through three phases of work. Two of these phases create Stop The World (STW) latencies and the other phase creates latencies that slow down the throughput of the application. The three phases are:

Mark Setup - STW

When a collection starts, the first activity that must be performed is turning on the Write Barrier. The purpose of the Write Barrier is to allow the collector to maintain data integrity on the heap during a collection since both the collector and application goroutines will be running concurrently.

Marking - Concurrent

Once the Write Barrier is turned on, the collector commences with the Marking phase. The first thing the collector does is take 25% of the available CPU capacity for itself. The collector uses Goroutines to do the collection work and needs the same P's and M's the application Goroutines use. This means for our 4 threaded Go program, one entire P will be dedicated to collection work.

Mark Termination - STW

Once the Marking work is done, the next phase is Mark Termination. This is when the Write Barrier is turned off, various clean up tasks are performed, and the next collection goal is calculated. Goroutines that find themselves in a tight loop during the Marking phase can also cause Mark Termination STW latencies to be extended.

Goroutines:

One of the main reasons that the Go Language has gained incredible popularity in the past few years is the simplicity it deals with concurrency with its lightweight goroutines and channels.

Go routines rely upon threads to run.

What are Threads?

A thread is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process — that is, a single process may contain multiple threads.
A good example is a web server.
A webserver normally is designed to handle multiple requests at once. And these requests normally are independent from each other.
So a thread can be created, or taken from a thread pool, and requests can be delegated, to achieve concurrency. But remember from the famous Rob Pike talk, "Concurrency is not Parallelism".

Thread limitations

- Threads have a large stack size therefore consume a lot of memory. So creating a large number of threads leads to huge memory requirements.
- Threads need to restore a lot of registers, some of which include AVX(Advanced vector extension), SSE (Streaming SIMD Ext.), Floating Point registers, Program Counter (PC), Stack Pointer (SP) which hurts the application performance.

- Threads setup and teardown requires call to OS for resources (such as memory) which is slow.

Advantages of Goroutines over threads

- Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to the needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- The Goroutines are multiplexed to a fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread. All these are taken care of by the runtime and we as programmers are abstracted from these intricate details and are given a clean API to work with concurrency.
- Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate. We will discuss channels in detail in the next tutorial.

Code example

```
package main

import (
    "fmt"
    "time"
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    time.Sleep(1 * time.Second)
    fmt.Println("main function")
}
```

Channels

In the above code, the way of using sleep in the main Goroutine to wait for other Goroutines to finish their execution is a hack we are using to understand how Goroutines work. Channels can be used to block the main Goroutine until all other Goroutines finish their execution.

Channels can be thought of as pipes using which Goroutines communicate. Similar to how water flows from one end to another in a pipe, data can be sent from one end and received from the other end using channels.

Each channel has a type associated with it. This type is the type of data that the channel is allowed to transport. No other type is allowed to be transported using the channel.

Sends and receives to a channel are blocking by default. What does this mean? When data is sent to a channel, the control is blocked in the send statement until some other Goroutine reads from that channel. Similarly, when data is read from a channel, the read is blocked until some Goroutine writes data to that channel.

This property of channels is what helps Goroutines communicate effectively without the use of explicit locks or conditional variables that are quite common in other programming languages.

Go Runtime Scheduler

The Go Runtime Scheduler keeps track of each goroutine, and will schedule them to run in turn on a pool of threads belonging to a process.
The Go Runtime Scheduler does cooperative scheduling, which means another goroutine will only be scheduled if the current one is blocking or done, and that is easily done via code. Here are some examples:
Blocking syscalls like file and network operations.
After being stopped for garbage collection cycle.
This is better than pre-emptive scheduling which uses timely system interrupts (e.g. every 10 ms) to block and schedule a new thread which may lead a task to take longer than needed to finish when number of threads increases or when a higher priority tasks need to be scheduled while a lower priority task is running.

Resources:

- https://bluxte.net/musings/2018/04/10/go-good-bad-ugly/
- https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8
- https://medium.com/the-polyglot-programmer/what-are-goroutines-and-how-do-they-actually-work-f2a734f6f991
- https://golangbot.com/goroutines/

- https://golang.org/doc/