SkyPenguinLabs

Samples

Learn How To Utilize Go's STD Lib To Optimize Applications

SPL-PRGC2

SPL-PRGC3 Introduction

> Utilizing Go's Expansive Standard Library to Optimize Applications

Welcome to the third course to be put on the SkyPenguinLabs course pages for free! We hope you are enjoying these little modules so far and definitely hope they are getting you to look at some of our other niche things haha!

Anyway, in today's module, we will be covering the use of Go's extremely expansive standard library to remove the bloat out of standard CLI applications.

You may be asking - *what kind of bloat are we talking about?*

Well, the bloat that comes from libraries of course! So let's not waste anytime and hop straight into this one,

Table of Contents

> Course content outline

Below are the sections listed in this module along with a description explaining what the module is going to be discussing.

Sections start on the next page.

Section 0x00

> Prerequisites

Before we get into this course, it is important that you have a machine setup to run the code if you plan on walking through the module for practicality instead of just reading it for theoretical knowledge.

This course module does not require any extensive background experience in software development, but may require a little bit more experience in software engineering & design to grasp some of the concepts. However, I have done my best to break it down so that even beginners could grasp the content in this module.

As far as environment, this module operates working with Go version 1.24.2 specifically for Linux on an AMD64 target.

The module starts on the next page, best of luck, nerd :D

Section 0x01

> A Hard Introduction To Go's Standard Library

Golang is a language that I have seen many beginners personally undermine. Primarily as their regular goto practice has been to go straight to github for finding a package that somebody has built to already automate a task upon building a project. Taking from other languages, such as Python, where this is an extremely common practice.

While a common practice in some languages, can be a dangerous practice in others. In go, one of the biggest issues with importing module code can be the extra bloat it adds for no reason.

One of the most common examples I like to use is getting the most extensive feature filled logging library for the most simplistic logging operations. While the dead and unused code gets terminated by Go's compiler, what still ends up getting used is the massive amount of space larger more feature filled logging libraries might use due to how much they need to store (*logging configurations, file configurations/write handles, service checking, thread handlers, etc*)

What's fascinating about Go is how it inverts the typical progression you see in other languages. In Python or JavaScript, you often start simple and then hit walls where you need external packages. In Go, you start with a remarkably capable standard library and only reach outside when you have a genuinely specialized need.

Don't get me wrong, libraries can seriously aid us, however, Go is meant for so much more than to be slapped and shoved with a bunch of bloaty library code. So in this section, I want to cover with you some of the most important things about the expansibility of Go's standard library.

The following sections describe multiple things about Go's standard library that may actually captivate your interest.

Golang & Its Primary Use

In order to understand how to use the Go programming language & its standard libraries, we should maybe understand what the language was actually intended for.

Directly based on their wikipedia page - With the rise of multicore processors and networked systems, Go was designed with features like built-in concurrency support (*goroutines and channels*) to handle these environments & to improve programming productivity in large scale enterprise code bases.

This is something MANY people forget when taking on Go. Which is why it disheartens me to know that as a security persons, I watch people all day copy and paste the same methodologies used when writing Python to Go and facepalm when their app crashes because they did not check interface type conversion :)))))))

Rant asides, Google did not invent a programming language in the height of an era with computer networking for no reason. Their docs show it.

Now that we understand that its primary use was to be for heavyweight applications, we can gauge that stuffing it with libraries, *definitely is not right -_-*

Golangs Standard Library

The Go programming language has a HUGE standard library. While there are some really good remote packages that Google has produced to replace portions of the standard library, there is still a really extensive list of libraries.

Its not even just how many names exist in that list, but rather the functionality and flexibility those packages give you to build your own better, if not stronger versions. And oftentimes, like some of the tasks that you will see in this module, you may not even need to use a sliver of a quarter in that library and can most likely deduce and automate some of the functions you are calling yourself.

In this course module, we will be using a multitude of these standard libraries to cover exercises which allow us to learn about how Go's standard library is able to provide a layer of protection against beefy code.

When working with production level applications, such as servers, you may encounter a need to optimize for many different reasons, whether it is the infrastructure you are hosting on being limited, or the amount of resource allowance you have to just burn.

It is not anything new to say that libraries also have their own sets of advantages, which is why some libraries are re-created by organizations like Google and even individuals on GitHub which can end up becoming industry standards due to their enhanced security, performance and more.

However, Go has a massive advantage outside of just its size and its ability to be optimized.

That being said, we need to know when and where to do this, as sometimes, in critical scenarios, going all self hosted beast mode is not really the way to do it, as this can introduce more flaws in your codebase if you do not know what you are doing.

The section on the next page hops into Section 0x02, which defines where we need to apply this, and some theoretical scenarios where it is absolutely necessary.

Section 0x02

> Theoretical Scenarios of Optimization Points

In order to start going code monkey style on Go, understanding where to apply the proper technique is important. To me, I like to think of libraries as either PoC or production. They are either one or the other.

Some people use them as half, which means you are using a library that should only be for concepts (*meaning you had to hack your way around it to get the PoC to work)* in production. And I have seen this a lot with the rise of vibe-coding. And while libraries are not always 'PoC' only, sometimes their functionality only allows them to be, and this is the case with a lot of useless libraries that add extra bloat such as color libraries that have 40 modules to handle ANSI escape sequences.

That being said, finding the middle ground can be hard in some cases, but this scenario aims to give you a good foundation for at least learning where to start making the optimizations.

Command Line Application Frontend

One of the most notorious types of applications for this mistake to be made in is command line applications where frontend is often used to display information in a structured, organizational manner.

Lets say, you are trying to highlight text in the shell, bold it, or you have a logging library with error messages that needs 'ERROR' to be bold for some reason, what is your first instinct? Is it to use a color package to highlight the color text?

While this is not inherently wrong, installing a package and adding an extra layer of dependency, if the references are repeated and are not handled properly, will hurt the applications performance.

Note: *I wanted to make a subtle note about this but could not find much of a way to get it across better than saying it. Sometimes (ALL THE TIMES), blindly installing a library without doing an internal check yourself behind the scenes* *and genuine research for performance is one of the worst practices I think I can see developers do. While library versions get checksum, and yes this prevents collisions, it does not prevent stupid stuffy code from entering your code base. So, if you see a library that is small enough to audit yourself, for a task as much as say graphic rendering, make sure there is no absolutely obvious code that ruins performance. Additionally, do not ever hesitate to do performance checking with go-perf. Additionally, some libraries hosted on GitHub publicly might not actually be intended to be installed, and if it's a really really non-talked about package, probably should not even be installed without being checked and double checked again.*

In command line apps, this can also include table libraries, or libraries that convert regular native data to CSV or JSON. Libraries for functionalities as such yes are nice, but sometimes doing it yourself saves you the time, and additionally allows you to build a sort of framework for building projects (*I talk about this in a paid course: Frameworking projects with Golang*).

Frontend is really easy to do yourself, and is just not talked about nor explained enough in the computer science community until you touch specific areas like graphics engineering, or graphic design, and etc.

Command Line App Initialization

Specific components apart from command line applications such as the flag systems can be replaced with Go's standard library or can be replaced with your own module if you wish to. Using libraries for flags is not much of a problem and is unlikely to take chunks out of your code's performance, but its always worth it to make sure the library is never over-reaching.

Its command line arguments, how far does it need to push you know? Some of them have way too many syntactic features trying to handle every single method of passing arguments via command line switches in the book.

Other initialization components include configuration systems, and service loaders which libraries tend to beef up. Configuration libraries can be pretty beefy because people love adding extra parsing and conversion options, you know loading via XML, and JSON and CSV. Of course you do not have to use all of this, but if you can pack lighter, its always nicer right?

Task Heavy Critical Components

Spanning away from code size, focusing more on practicality, scalability, and flexibility, one of the biggest things with developing task heavy critical components is finding a means of performance and practicality.

For example, when working with a custom network protocol, built on the transport layer, we would not use package gorilla/mux to implement the protocol. Instead, we would probably want to build up the layers from scratch since in such a scenario, we need as much control over the program code as possible and need to have as much trust as possible.

Other libraries in scenarios as critical as this, due to version changes may introduce the possibility of vulnerabilities. However, you could always locally install the library and keep using that version. Until a vulnerability comes out for the feature you are using within that library haha.

Go is also really good with handling a large amount of concurrency and if you can learn how to leverage such a library, it outbeats a lot of what other people try to recreate by building on top of Go's threading system. Asynchronization is not the first thing on every dev's mind and don't expect it to be.

That being said, we can now go into the use of the library instead of chit chatting away at it. Because there is only one way to really knock it in the goal…that's by showing it.

The next page starts a section which teaches executing on one of the optimization points mentioned above.