Why Learn Go?

What are Go's Advantages

Ian Harris

<2018-12-14 Fri 05:48>

1 Why Learn Go?

So, why should you learn Go? What's unique and good about the Go language? I'll go over some of the highlights of that. Different people have different opinions, but there are certain features that are pretty uniformly understood as being advantages of the Go language.

1.1 Go's Advantages

So, some of the advantages of Go.

Fast: First, runs fast. Okay, that's always a good thing, and I'll talk a little bit more detail in the slides about why it runs fast, and faster than what, right? Doesn't run fast than everything, but it runs fast than a lot of things, and we'll talk about why.

Garbage Collection: Garbage collection, that's another feature. That in similar languages, meaning languages that run fast like Go does, they don't have garbage collection. So, garbage collections are really useful feature, I'll describe what that is.

Simple Objects: Simpler objects, now this could be an advantage or disadvantage, but the idea is that Go is essentially object oriented although, some might disagree, but it has this concept of objects, and these objects are generally, the object orientation is a little simplified as compared to other languages. So, this is good, it makes it easier to code. You don't have to use these complicated features, of course you never had to use them any way, but it makes it faster, and simpler to use. So, I know somebody might argue that look these features are useful features, I wanted them. Okay. But Go is actually a simpler object-oriented implementation than you would see in other languages like C++.

Built-In Concurrency Features: Another feature of Go is that it has concurrency, efficient concurrency implementation built into language. So, there are a lot of these concurrency primitives that are built into the language, and that are efficiently implemented, and we'll talk about those.

1.2 Why Go Code Runs Fast

So, to start off with, let's go into that code running fast thing and tell you why that is the case. To talk about that, we've got to talk about a little bit about languages in general, differences between languages.

1.2.1 Categories of Languages

So, I've got these three, very broad categories of languages for machines. So, one is machine language, next is Assembly language, next is high-level language, right?

Machine Language: Now, that's a really big category, but let me explain. Machine language to start off with, that is the lowest level language, and it's directly executed on the CPU, on the processor. So, the machine language instructions are very, very straightforward and simple, add, multiply, add might add to register contents, put the result in another register, something like that, very small steps, each machine language instruction. Now, this runs directly on the processor, so there's that.

Assembly Language: There's assembly language and assembly language is basically machine language, almost a one-to-one mapping to machine language. So, in machine language, say you had to add instruction that might be represented as a sequence of zeros and ones, 11110000 that might be an add, Opcode for an add. So, in assembly language, you would use the word ADD. So, it maps one-to-one to the machine language equivalent, but you can read it because it's English.

Now, it's concise English, I would argue it's hard to read, but it's English mnemonic, so a human could read that and could write that if you wanted to, and people do sometimes. If you want something to run really, really fast and really efficiently, then you'll write it directly in assembly language. That's outside of the scope of this class, that's hardcore, I'll call it. That is not something we're going to really cover in this class, but sometimes you do write directly in assembly language. So, there's assembly machine in assembly, and assembly language is basically a one-to-one mapping to machine language not completely, but very close. So, fundamentally they're the same complexity. They're the same language, but assembly language is easier to read.

High-Level Languges: Now, as high-level languages that's everything else, that's a broad category. A high-level language is a language that essentially humans commonly use to program in. They are much easier to use than assembly

language or machine language. They provide you with lots of abstractions that any program would be used to, for instance variables, right? Assembly language and machine language do not have variables, they have memory, and you can put stuff in it, take stuff out, right? There's no idea of a type or anything like that in assembly language or machine language, but high-level languages provide that to you, right? If statements. Now, assembly language, machine language, they have conditional branches and so on, but these are generally much harder to use than your standard if statement that you would see in any normal high-level language or loops or, for loops things like this. You can create these things in assembly and machine language, but they are harder to write than they would be in a high-level language. So, high-level languages are basically everything that most people program in. So, you can imagine these different categories.

The Translation of High-Level Into Machine Language: Now, the language that we're talking about Go. Go is of course a high-level language in this set of three categories, it will be considered high level. Remember that term high-level is subjective, okay? But I'll call it high level. So, all software that I have highlighted in the slide, all software needs to be translated into the machine language of the processor to be executed. So, what that means is, that if you've got a processor or some kind, some i7 or whatever processor you're working with, that processor does not know C or Java or Python or Go or C++ or any of those, right? All it knows is its own machine language, say x86 machine language, if it's an Intel based processor or an AMD or something like that, so it knows that machine language. So, in order for the code to execute on the processor, it has to be first translated into the machine language of the processor. So, even if it's C, Python, Java whatever it is, has to be translated. So, there is this software translation step that has to go on.

Compilation: Okay, now, this translation step, it can go on in one of roughly two ways. It can be compiled, it can be a compilation or interpretation, okay? Now, a compiled language is a language where the translation from high-level language to machine code happens one time before you execute the code, before you deploy the code, happens one time, okay? So, like in C, C++, Go, Java partially, there's a compiler, and you compile the code. So, somebody writes the source code, they compile it, and then they execute it, and they execute the compiled executable, right? The compiled executable is basically machine language code plus other stuff, but it's basically machine language code. So, the idea behind a compiled languages, the key thing we want to bring out anyway is the fact that this translation occurs once, it doesn't occur while you're running the code, right? It happens before you run the code, and then when you run the code, you are just running the machine language instructions directly because they're already compiled into machine language by the compiler.

Interpretation: So, the other way to do this is interpreted, interpretation. In interpreted language what happens is, instructions are translated while the

code is executed. So, what happens is, it happens it adds time to the execution because every time you see an instruction, and say Python, that code, that instruction has to be translated into machine code on a fly, and that takes a certain amount of time just to do that translation. So, in addition to actually executing the instruction, you've got to do this translation from the instruction into the equivalent machine code, so that slows you down. So, the translation occurs every time you run the Python code, say or Java, the Java byte code. I put Java as partially in both categories because Java it's compiled, but it generates what's called byte code, not actual machine code and then the byte code has to be interpreted at runtime. So, there's an Interpreter also in the Java virtual machine, but these interpretive languages require an interpreter to be executing while you're running your code because it has to be doing this translation as you execute the code, and so that slows you down.

Trade-offs Between Compilation and Interpretation: Now, this is a trade off between compiled code and interpreted code.

Compiled Languages Run Faster. The first level is trade off. One big difference you can see is, its compiled code is generally faster to execute, that's because you don't have to do the translation every time you run the code, so it is going to be faster. Now, there are people who would argue the opposite, but generally compiled code is a lot faster. Now, on the other hand though, interpreters make coding easier. So, the thing about interpreters is that, the interpreter itself, that program that is doing the translation of your code, it can help you, it can handle things that you, as a programmer, don't want to handle. For instance, in Python, I don't have to declare my variable types. I can just start using a variable and the interpreter will say, "It looks like he's using it as an integer, make it an integer." So, that's something that the programmer doesn't have to think about.

Interpreters Have Memory Management. Another thing that interpreters commonly often have is memory management. In fact, almost always they have this. They can manage their memory, and by that I mean, getting rid of variables and other pieces of data when you're not using them. So, when you use a variable, that variable has to go into memory somewhere, and that memory, if you keep making variables and using a memory space, you will eventually run out of memory, things will slow down and you'll run ot of memory. So, you have to manage your memory, that is when you're done using an object, you want to get rid of that, de-allocate it from memory, and that happens automatically in an interpreted language, so the interpreter handles that. So, that's a good thing about interpreters.

Advantages of Go So, Go is a good compromise between this compiled and interpreted type of language. It's a compiled language, but it has some of the good features of interpreted language, specifically it has garbage collection. So,

garbage collection is the automatic memory management that I'm talking about. So, this memory management, where should memory be allocated, so I'd say, "I need a variable x. Where should I put it in memory? What type of memory I should put it in?" We'll, talk about that a little bit later, but also when am I done with that memory, right? Because when you're done with the memory, you can get rid of a memory. You don't have to use it anymore, you can use it for something else. So, that's what I talk about memory management, that's what I mean and this happens automatically. The garbage collector can figure that out, it says, "Oh, it looks like this program is done with variable x, I will free that memory now," and that happens automatically. Manual memory management is hard. This is, if you've ever done C or something like this, you know this, deallocate memory too early, if you stop using it too early, then you will have false memory accesses, you still need it, so you'll use the memory that's already deallocated and errors crop up because of that. Also if you deallocate it too late, then you're wasting memory, you can have what's called a memory leak, where you have more and more memory that's not actually being used, but it's being blocked up because your machine thinks it's being used, right? So, at memory management manually is very difficult, and there are lots of errors, security errors too. So, Go has a garbage collection code included, so when it compiles your code, it also compiles garbage collection into your code automatically, and this is typically only done by interpreter. So, this is a compiled language that actually has garbage collection, which is a really good feature. Now, downside is that it slows down execution a bit, but it's an efficient garbage collector, so it doesn't slow down much and you get a lot of advantage of having this automatic garbage collection.