**Why Go, motivation and philosophy for the golang language**

**Go is a compiled, statically typed language with a C-like syntax and garbage collection.**

Below is the typical hello world program for the go language

Once we have a development environment setup we can compile and build this application with the go environment tool “go”

**package main**

**import (**

**"fmt"**

**)**

**func main() {**

**fmt.Println("Go reports: %+v\n ", “Hello from Go toolchain”)**

**}**

In the next module “Setting up a Go development environment” we carefully step thru everything that's involved for setting up the development environment but for now let's quickly build package and deploy this application using the Golang toolchain

**go install github.com/nsavageJVM/dev1**

**Introducing the Go Toolchain**

For system programming we use the term “**Toolchain”** to define the set of software engineering tool we employ to compile build package and deploy applications from one operating system to another. In the domain of embedded programming when we use the term “cross compile” this refers to building on a host system then deploying to a different system.

|  |  |  |  |
| --- | --- | --- | --- |
| Host System | Target System | Language | Toolchain |
| Unix on x86 | Unix on ARM | C | [gnu](https://en.wikipedia.org/wiki/GNU_toolchain) |
| Unix on x86 | Android | Java | [Gradle](http://gradle.org/) |
| Unix on x86 | Android | Go | [Android NDK](https://developer.android.com/tools/sdk/ndk/index.html) |

**Common Go toolchain commands**

1. compile and run from source **go run <file-name>.go**

2. explicitly compile and package with dependencies **go build <file-name>.go**

3. compile packages and dependencies and produce an executable binary **go install**

Go’s compiler is so fast that the command “go build” compiles packages with dependencies giving the user the impression of running an interpreted language. This fact is a key Motivation behind the design of Go language

**Other Common Go toolchain commands**

4. download packages (source code) **go get <package-name>**

5. remove object files clean  **go clean**

6, test packages **go test**

see this link for a definitive list

<https://golang.org/cmd/go/>

The Go toolchain is a direct descendant of the Plan 9 toolchain. The toolchain consists of a Go compiler, assembler and a linker. The latest version of Go(1.5) has a refactored and streamlined toolchain where the main components are now completely independent of C.

The source code can be compiled to special object files <>.a, <>.so. These are libraries, \*.a is a static library and \*.so is a dynamic library. Object files can be needed for the compile link phase (static) or required at runtime (dynamic).

You can think of the linker as the glue that will bind the parts of your program together. Generally you only work directly with a dynamic library when interfacing with other languages then you will build with the **buildmode** flag. As long as the source being compiled is native go, the go compiler will statically link the executable.

**go build -buildmode**

To delve further into the tool chain internals if you build with the -x flag will put all the internal commands of the toolchain

**go build -x**

the assembler converts the Go source into executable machine code <https://golang.org/doc/asm>

The linker creates a statically linked ELF file which includes the Go runtime, your program, and dependencies. It arranges for it's own data structures to be initialized, then calls the initializers of your program, and all dependencies. The runtime has platform and and OS-specific artifacts, which are compiled in according the the target platform.

<https://golang.org/doc/go1.5#link>

The Go Package runtime contains operations that interact with Go's runtime system and is not to be confused with the Go's runtime system.

<https://golang.org/pkg/runtime/>