TinyGo

Introduction

TinyGo is an alternative compiler for Go source code. It can generate `%.wasm` files instead of architecture-specific binaries through two targets:

- `wasm` : for browser (JavaScript) use.
- `wasi` : for use outside the browser.

This document is maintained by wazero, which is a WebAssembly runtime that embeds in Go applications. Hence, all notes below will be about TinyGo's `wasi` target.

Overview

When TinyGo compiles a `%.go` file with its `wasi` target, the output `%.wasm` depends on a subset of features in the [WebAssembly 2.0 Core specification] (https://wazero.io/specs/#core) and WASI host functions.

Unlike some compilers, TinyGo also supports importing custom host functions and exporting functions back to the host.

Here's a basic example of source in TinyGo:

```
package main

//export add
func add(x, y uint32) uint32 {
	return x + y
}

// main is required for the `wasi` target, even if it isn't used.
func main() {}
```

The following is the minimal command to build a `%.wasm` binary.

```
tinygo build -o main.wasm -target=wasi main.go
```

The resulting wasm exports the `add` function so that the embedding host can call it, regardless of if the host is written in Go or not.

Disclaimer

This document includes notes contributed by the wazero community. While wazero includes TinyGo examples, and maintainers often contribute to TinyGo, this isn't a TinyGo official document. For more help, consider the TinyGo Using WebAssembly Guide or joining the #TinyGo channel on the Gophers Slack.

Meanwhile, please help us maintain this document and star our GitHub repository, if it is helpful. Together, we can make WebAssembly easier on the next person.

Constraints

Please read our overview of WebAssembly and constraints. In short, expect limitations in both language features and library choices when developing your software.

Unsupported standard libraries

TinyGo does not completely implement the Go standard library when targeting `wasi` . What is missing is documented here.

The first constraint people notice is that `encoding/json` usage compiles, but panics at runtime.

```
package main

import "encoding/json"

type response struct {
	Ok bool `json:"ok"`
}

func main() {
	var res response
	if err := json.Unmarshal([]byte(`{"ok": true}`), &res); err != nil {
		println(err)
	}
}
```

This is due to limited support for reflection, and effects other serialization tools also. See Frequently Asked Questions for some workarounds.

Unsupported System Calls

You may also notice some other features not yet work. For example, the below will compile, but print "readdir unimplemented: errno 54" at runtime.

```
package main

import "os"

func main() {
	if _, err := os.ReadDir("."); err != nil {
		println(err)
	}
}
```

The underlying error is often, but not always `syscall.ENOSYS` which is the standard way to stub a syscall until it is implemented. If you are interested in more, see System Calls.

Memory

When TinyGo compiles go into wasm, it configures the WebAssembly linear memory to an initial size of 2 pages (128KB), and marks a position in that memory as the heap base. All memory beyond that is used for the Go heap.

Allocations within Go (compiled to `%.wasm`) are managed as one would expect. The allocator can grow until `memory.grow` on the host returns -1.

Host Allocations

Sometimes a host function needs to allocate memory directly. For example, to write JSON of a given length before invoking an exported function to parse it.

The below snippet is a realistic example of a function exported to the host, who needs to allocate memory first.

```
//export configure
func configure(ptr uintptr, size uint32) {
  json := ptrToString(ptr, size)
}
```

Note: WebAssembly uses 32-bit memory addressing, so a `uintptr` is 32-bits.

The general flow is that the host allocates memory by calling an allocation function with the size needed. Then, it writes data, in this case JSON, to the memory offset (`ptr`). At that point, it can call a host function, ex `configure` , passing the `ptr` and `size` allocated. The guest wasm (compiled from Go) will be able to read the data. To ensure no memory leaks, the host calls a free function, with the same `ptr` , afterwards and unconditionally.

Note: wazero includes an example project that shows this.

The general call patterns are the following. Host is the process embedding the WebAssembly runtime, such as wazero. Guest is the TinyGo source compiled to target wasi.

- Host allocates a string to call an exported Guest function
  - Host calls the built-in export `malloc` to get the memory offset to write the string, which is passed as a parameter to the exported Guest function. The host owns that allocation, so must call the built-in export `free` when done. The Guest uses `ptrToString` to retrieve the string from the Wasm parameters.
- Guest passes a string to an imported Host function
  - Guest uses `stringToPtr` to get the memory offset needed by the Host function. The host reads that string directly from Wasm memory. The original string is subject to garbage collection on the Guest, so the Host shouldn't call the built-in export `free` on it.
- Guest returns a string from an exported function
  - Guest uses `ptrToLeakedString` to get the memory offset needed by the Host, and returns it and the length. This is a transfer of ownership, so the string won't be garbage collected on the Guest. The host reads that string directly from Wasm memory and must call the built-in export `free` when complete.

The built-in `malloc` and `free` functions the Host calls like this in the WebAssembly text format.

```
(func (export "malloc") (param $size i32) (result (;$ptr;) i32))
(func (export "free") (param $ptr i32))
```

The other Guest function, such as `ptrToString` are too much code to inline into this document, If you need these, you can copy them from the example project or add a dependency on tinymem.

System Calls

Please read our overview of WebAssembly and System Calls. In short, WebAssembly is a stack-based virtual machine specification, so operates at a lower level than an operating system.

For functionality the operating system would otherwise provide, TinyGo imports host functions defined in WASI.

For example, `tinygo build -o main.wasm -target=wasi main.go` compiles the below `main` function into a WASI function exported as `_start` .

When the WebAssembly runtime calls `_start` , you'll see the effective `GOARCH=wasm` and `GOOS=linux` .

```
package main

import (
	"fmt"
	"runtime"
)

func main() {
	fmt.Println(runtime.GOARCH, runtime.GOOS)
}
```

Note: wazero includes an example WASI project including source code that implements `cat` without any WebAssembly-specific code.

WASI Internals

While developing WASI in TinyGo is outside the scope of this document, the below pointers will help you understand the underlying architecture of the `wasi` target. Ideally, these notes can help you frame support or feature requests with the TinyGo team.

A close look at the wasi target reveals how things work. Underneath, TinyGo leverages the `wasm32-unknown-wasi` LLVM target for the system call layer (libc), which is eventually implemented by the `wasi-libc` library.

Similar to normal code, TinyGo decides which abstraction to use with GOOS and GOARCH specific suffixes and build flags.

For example, `os_Args` is implemented directly using WebAssembly host functions in runtime_wasm_wasi.go. `syscall.Chdir` is implemented with the same syscall_libc.go used for other architectures, while `syscall.ReadDirent` is stubbed (returns `syscall.ENOSYS`), in syscall_libc_wasi.go.

Concurrency

Please read our overview of WebAssembly and concurrency. In short, the current WebAssembly specification does not support parallel processing.

Tinygo uses only one core/thread regardless of target. This happens to be a good match for Wasm's current lack of support for (multiple) threads. Tinygo's goroutine scheduler on Wasm currently uses Binaryen's Asyncify, a Wasm postprocessor also used by other languages targeting Wasm to provide similar concurrency.

In summary, TinyGo supports goroutines by default and acts like `GOMAXPROCS=1` . Since goroutines are not threads, the following code will run with the expected output, despite goroutines defined in opposite dependency order.

```
package main

import "fmt"

func main() {
	msg := make(chan int)
	finished := make(chan int)
	go func() {
		<-msg
		fmt.Println("consumer")
		finished <- 1
	}()
	go func() {
		fmt.Println("producer")
		msg <- 1
	}()
	<-finished
}
```

There are some glitches to this. For example, if that same function was exported (`//export notMain`), and called while main wasn't running, the line that creates a goroutine currently panics at runtime.

Given problems like this, some choose a compile-time failure instead, via `-scheduler=none` . Since code often needs to be custom in order to work with wasm anyway, there may be limited impact to removing goroutine support.

Optimizations

Below are some commonly used configurations that allow optimizing for size or performance vs defaults. Note that sometimes one sacrifices the other.

Binary size

Those with `%.wasm` binary size constraints can set `tinygo` flags to reduce it. For example, a simple `cat` program can reduce from default of 240KB to 60KB using both flags below.

- `-scheduler=none` : Reduces size, but fails at compile time on goroutines.
- `--no-debug` : Strips DWARF, but retains the WebAssembly name section.

Performance

Those with runtime performance constraints can set `tinygo` flags to improve it.

- `-gc=leaking` : Avoids GC which improves performance for short-lived programs.
- `-opt=2` : Enable additional optimizations, frequently at the expense of binary size.

Frequently Asked Questions

Why do I have to define main?

If you are using TinyGo's `wasi` target, you should define at least a no-op `func main() {}` in your source.

If you don't, instantiation of the WebAssembly will fail unless you've exported the following from the host:

```
(func (import "env" "main.main") (param i32) (result i32))
```

How do I use json?

TinyGo doesn't yet implement reflection APIs needed by `encoding/json` . Meanwhile, most users resort to non-reflective parsers, such as gjson.

Why does my wasm import WASI functions even when I don't use it?

TinyGo has a `wasm` target (for browsers) and a `wasi` target for runtimes that support WASI. This document is written only about the `wasi` target.

Some users are surprised to see imports from WASI (`wasi_snapshot_preview1`), when their nether has a main function nor uses memory. At least implementing `panic` requires writing to the console, and `fd_write` is used for this.

A bare or standalone WebAssembly target doesn't yet exist, but if interested, you can follow this issue.

Why is my `%.wasm` binary so big?

TinyGo defaults can be overridden for those who can sacrifice features or performance for a smaller binary. After that, tuning your source code may reduce binary size further.

TinyGo minimally needs to implement garbage collection and `panic` , and the wasm to implement that is often not considered big (~4KB). What's often surprising to users are APIs that seem simple, but require a lot of supporting functions, such as `fmt.Println` , which can require 100KB of wasm.