Using WASM

How to call WebAssembly from JavaScript in a browser.

You can call a JavaScript function from Go and call a Go function from WebAssembly:

```
package main

// This calls a JS function from Go.
func main() {
    println("adding two numbers:", add(2, 3)) // expecting 5
}

// This function is imported from JavaScript, as it doesn't define a body.
// You should define a function named 'add' in the WebAssembly 'env'
// module from JavaScript.
//
//export add
func add(x, y int) int

// This function is exported to JavaScript, so can be called using
// exports.multiply() in JavaScript.
//
//export multiply
func multiply(x, y int) int {
    return x * y;
}
```

Related JavaScript would look something like this:

```
// Providing the environment object, used in WebAssembly.instantiateStreaming.
// This part goes after "const go = new Go();" declaration.
go.importObject.env = {
    'add': function(x, y) {
        return x + y
    }
    // ... other functions
}

// Calling the multiply function:
console.log('multiplied two numbers:', wasm.exports.multiply(5, 3));
```

You can also simply execute code in `func main()`, like in the standard library implementation of WebAssembly.

Building

If you have `tinygo` installed, it's as simple as providing the correct target:

```
GOOS=js GOARCH=wasm tinygo build -o wasm.wasm ./main.go
```

If you're using the docker image, you need to mount your workspace into the image. Note the `--no-debug` flag, which reduces the size of the final binary by removing debug symbols from the output. Also note that you must change the path to your Wasm file from `/go/src/github.com/myuser/myrepo/wasm-main.go` to whatever the actual path to your file is:

```
docker run -v $GOPATH:/go -e "GOPATH=/go" tinygo/tinygo:0.34.0 GOOS=js GOARCH=wasm tiny
```

Make sure you copy `wasm_exec.js` to your runtime environment:

```
docker run -v $GOPATH:/go -e "GOPATH=/go" tinygo/tinygo:0.34.0 /bin/bash -c "cp /usr/lo
```

More complete examples are provided in the wasm examples.

How it works

Execution of the contents require a few JS helper functions which are called from WebAssembly. We have defined these in `tinygo/targets/wasm_exec.js`. It is based on `$GOROOT/misc/wasm/wasm_exec.js` from the standard library, but is slightly different. Ensure you are using the same version of `wasm_exec.js` as the version of `tinygo` you are using to compile.

The general steps required to run the WebAssembly file in the browser includes loading it into JavaScript with `WebAssembly.instantiateStreaming`, or `WebAssembly.instantiate` in some browsers:

```
const go = new Go(); // Defined in wasm_exec.js
const WASM_URL = 'wasm.wasm';

var wasm;

if ('instantiateStreaming' in WebAssembly) {
    WebAssembly.instantiateStreaming(fetch(WASM_URL), go.importObject).then(function (o
        wasm = obj.instance;
        go.run(wasm);
    })
} else {
    fetch(WASM_URL).then(resp =>
        resp.arrayBuffer()
    ).then(bytes =>
        WebAssembly.instantiate(bytes, go.importObject).then(function (obj) {
            wasm = obj.instance;
            go.run(wasm);
        })
    )
}
```

If you have used explicit exports, you can call them by invoking them under the `wasm.exports` namespace. See the `export` directory in the examples for an example of this.

In addition to the JavaScript, it is important the wasm file is served with the `Content-Type` header set to `application/wasm`. Without it, most browsers won't run it.

```
package main

import (
    "log"
    "net/http"
    "strings"
)

const dir = "./html"

func main() {
    fs := http.FileServer(http.Dir(dir))
    log.Print("Serving " + dir + " on http://localhost:8080")
    http.ListenAndServe(":8080", http.HandlerFunc(func(resp http.ResponseWriter, req *h
        resp.Header().Add("Cache-Control", "no-cache")
        if strings.HasSuffix(req.URL.Path, ".wasm") {
            resp.Header().Set("content-type", "application/wasm")
        }
        fs.ServeHTTP(resp, req)
    }))
}
```

This simple server serves anything inside the `./html` directory on port `8080`, setting any `*.wasm` files `Content-Type` header appropriately.

For development purposes (**only!**), it also sets the `Cache-Control` header so your browser doesn't cache the files. This is useful while developing, to ensure your browser displays the newest wasm when you recompile.

In a production environment you **probably wouldn't** want to set the `Cache-Control` header like this. Caching is generally beneficial for end users.

Further information on the `Cache-Control` header can be found here:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control