GOing to WASM?

Building a frontend framework in Go!

18 Oct 2018

Max Gonzih

gonzih.me (//gonzih.me)

What is WASM?

WEBASSEMBLY

What is WASM?

- A standard that defines binary instruction format
- Intended to be run on a stack machine
- Faster to parse and execute than JavaScript
- Can be used outside of the browser

Possibilities

Using WebAssembly on a serverless platform

blog.cloudflare.com/webassembly-on-cloudflare-workers/ (https://blog.cloudflare.com/webassembly-on-cloudflare-workers/)

Cloudflare Workers

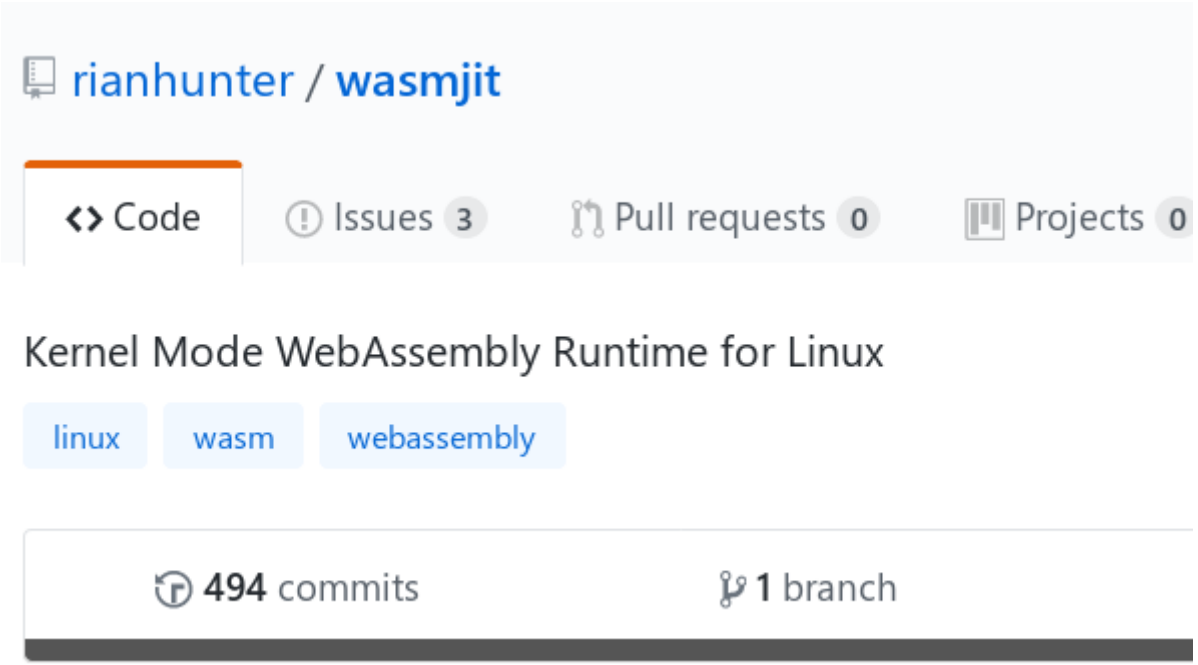
The Network is the Computer™

Build serverless applications on Cloudflare's global cloud network of 152 data centers. Cloudflare Workers provides a lightweight JavaScript execution environment that allows developers to augment existing applications or create entirely new ones without configuring or maintaining infrastructure.

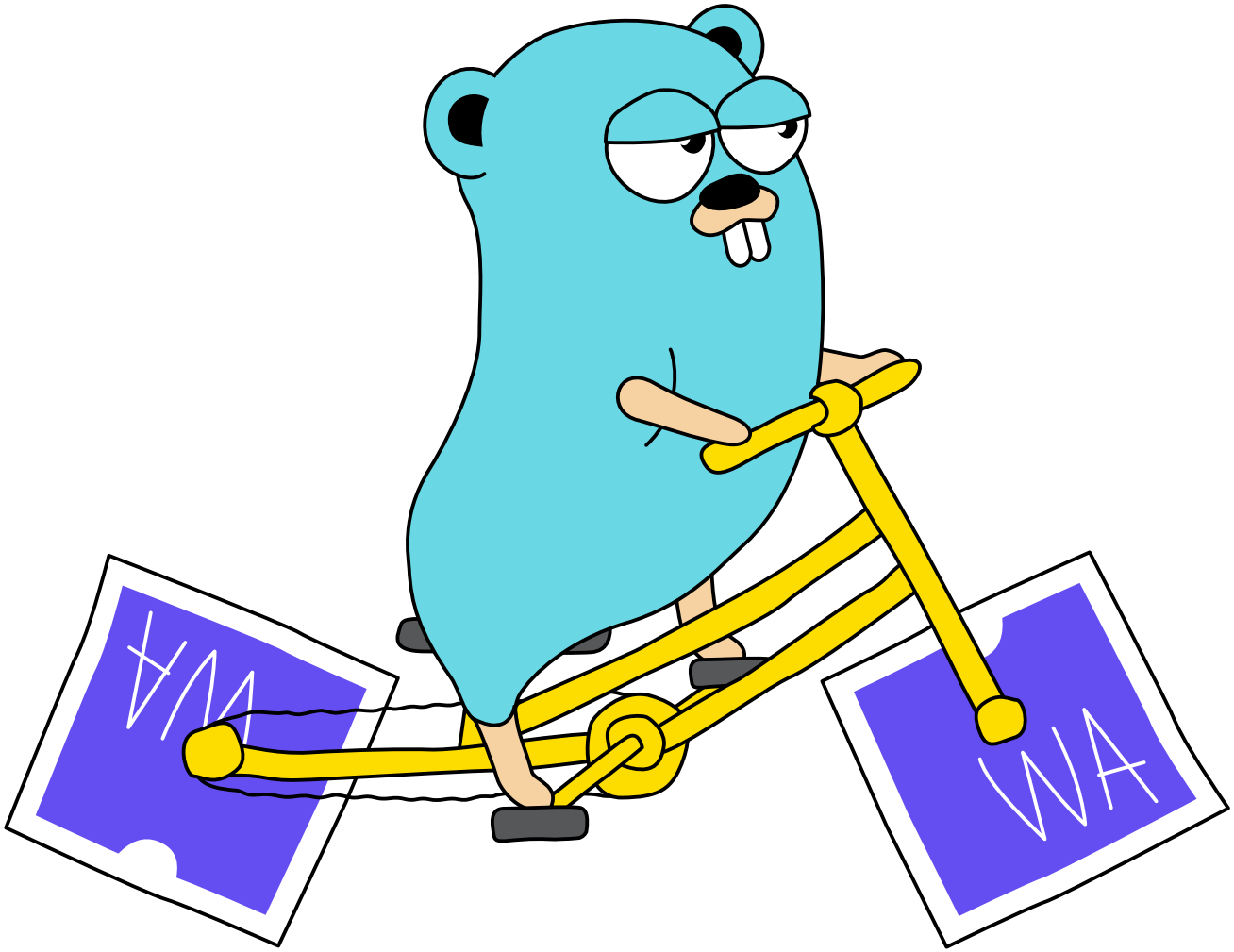
Possibilities

Kernel-space WebAssembly Runtime for Linux

github.com/rianhunter/wasmjit (https://github.com/rianhunter/wasmjit)



Go on WASM



Why is this exciting?

- Go can be used to build web apps using existing ecosystem and tooling
- Backend language on a frontend
- Sharing logic between backend and frontend
- Can target not just web browser

Running Go on WASM

- Available since 1.11
- Adds `GOOS=js` and `GOARCH=wasm` variables to the compiler

```
GOARCH=wasm GOOS=js go build -o test.wasm
```

- Interop with JavaScript is implemented using `syscall/js` package

GoDoc Home About Search

Go: syscall/js Index | Examples | Files

package js

```
import "syscall/js"
```

Package js gives access to the WebAssembly host environment when using the js/wasm architecture. Its API is based on JavaScript semantics.

This package is EXPERIMENTAL. Its current scope is only to allow tests to run, but not yet to provide a comprehensive API for users. It is exempt from the Go compatibility promise.

Hello World time

```go
package main

import "syscall/js"

func main() {
  js.Global().Call("alert", "Hello Wasm!")
}
```

Getting JS and HTML helper files

Can be found in go source tree under `misc/wasm` directory

```
cp $GOSRC/misc/wasm/wasm_exec.html wasm_exec.html
cp $GOSRC/misc/wasm/wasm_exec.js wasm_exec.js
```

JS helper

`wasm_exec.js`

- Unifies Node.js and browser WASM APIs into one JS object called Go
- Sets up JS -> Go interop

```
const setInt64 = (addr, v) => {
  mem().setUint32(addr + 0, v, true);
  mem().setUint32(addr + 4, Math.floor(v / 4294967296), true);
}
```

- Defines funcs from `syscall/js` package on JS side

```
// func valueGet(v ref, p string) ref
"syscall/js.valueGet": (sp) => {
  storeValue(sp + 32, Reflect.get(loadValue(sp + 8), loadString(sp + 16)));
},
```

Running it

test.wasm needs to be served with Content-Type header set to **application/wasm**

```
func wasmHandler(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/wasm")
  http.ServeFile(w, r, "test.wasm")
}

func main() {
  mux := http.NewServeMux()
  mux.Handle("/", http.FileServer(http.Dir(".")))
  mux.HandleFunc("/test.wasm", wasmHandler)
  log.Fatal(http.ListenAndServe(":3000", mux))
}
```

The Future is here

It works!

Yes, but... file size 😕

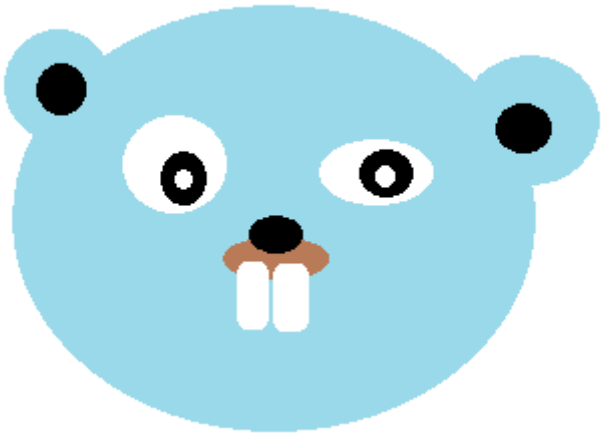
```
$ l -lha test.wasm
-rwxr-xr-x 1 gnzh gnzh 2.4M Sep 15 09:13 test.wasm*
```

Why?

- WASM is a stack machine. Go has to maintain its own stack (GC)
- No GOTO instruction, so Go has to generate huge switch statements to resume goroutines
- Binary size was not a priority for this release

More Information: WebAssembly architecture for GO (https://docs.google.com/document/d/131vjr4DH6JFnb-blm_uRdaC0_Nv3OUwjEY5qVCxCup4/edit)

Building a frontend framework



Reactive framework

- Markup format/DSL
- State management
- Instantiation of a component
- Rendering loop

Markup format

```
<template id="rootTemplate">
    <div>
        <span :class="LabelClass">Clicked me {{ Count }}</span>
        <input type="button" value="Click Me" @click="ClickHandler"></input>
    </div>
</template>
```

- computed properties

```
:class="LabelClass" and {{ Count }}
```

- event handlers

```
@click="ClickHandler"
```

State managment

```go
type Store struct {
  store map[string]interface{}
}

func (s *Store) Set(k string, v interface{}) {
  // ...
}

func (s *Store) Get(k string) interface{} {
  // ...
}

func (s *Store) Subscribe(ch chan bool) {
  // ...
}
```

Instantiation of a component

- Configure computed properties
- Configure event handlers
- Subscribe component to updates from Store
- Generate tree of elements

Instantiation of a component

```
store := NewStore()

computed := map[string]func() string{
  "Counter": func() string {
    // ...
  },
}

handlers := map[string]func(js.Value){
  "ClickHandler": func(event js.Value) {
    // ...
  },
}

cmp := NewComponent("rootTemplate", handlers, computed)
store.Subscribe(cmp.notificationChan)
cmp.MountTo("root")
```

Configure computed properties

```go
computed := map[string]func() string{
  "Count": func() string {
    return fmt.Sprintf("%d times", store.Get("counter"))
  },
  "LabelClass": func() string {
    c := store.Get("counter").(int)
    if c%2 == 0 {
      return "even"
    }
    return "odd"
  },
}
```

Configure event handlers

```
handlers := map[string]func(js.Value){
  "ClickHandler": func(event js.Value) {
    c := store.Get("counter").(int)
    store.Set("counter", c+1)
  },
}
```

Subscribe component to updates from Store

```go
func (s *Store) Set(k string, v interface{}) {
  s.store[k] = v
  for _, sub := range s.subs {
    select {
    case sub <- true:
    default:
    }
  }
}

func (s *Store) Subscribe(ch chan bool) {
  s.subs = append(s.subs, ch)
}
```

Generate tree of elements

- Get HTML markup from <template>
- Parse HTML into tree of elements
- Create dynamic attributes
- Create event handlers

Get HTML markup from <template>

```
// Go
js.Global().Get("document").Call("getElementById", templateID).Get("innerHTML").String()

// JavaScript
document.getElementById(templateID).innerHTML
```

Parse HTML into tree of elements

Using `net/html` package

```
r := strings.NewReader(templateData)
z := html.NewTokenizer(r)

tt := z.Next()
switch {
case tt == html.StartTagToken:
  el := &El{}
  el.Type = token.Data
  for _, attr := range token.Attr {
    // ...
  }
  el.Children = parseChildren(z)
case tt == html.EndTagToken:
  return el
}
```

Create dynamic attributes

```
for _, attr := range token.Attr {
  if strings.HasPrefix(attr.Key, ":") {
    k := strings.Replace(attr.Key, ":", "", 1)
    f, ok := component.Computed[k]
    if ok {
      el.Attr = append(el.Attr, &DynamicAttribute{K:  k, Fn: f})
    }
  }
}
```

Create event handlers

```
for _, attr := range token.Attr {
  if strings.HasPrefix(attr.Key, "@") {
    k := strings.Replace(attr.Key, "@", "", 1),
    method, ok := component.Handlers[k]
    if ok {
      callback := js.NewEventCallback(js.PreventDefault, method)
      el.Handlers[k] = callback
    }
  }
}
```

Handle text nodes

```
case tt == html.TextToken:
  t := z.Token()
  el := &El{Type: TEXT_TYPE, NodeValue: t.Data}
  // ...

// <span>This is an example template for {{ Name }}</span>
for k, fn := range component.Computed {
  re := regexp.MustCompile(fmt.Sprintf(`\{\{\s*%s\s*\}\}`, k))
  if re.MatchString(el.NodeValue) {
    at := &DynamicAttribute{
      K:  k,
      Fn: fn,
    }
    el.Attr = append(el.Attr, at)
  }
}
```

Rendering loop

- Start rendering loop
- Generate Virtual DOM (VDOM)
- Diff VDOM to generate set of changes
- Apply changes to the DOM tree

Start rendering loop

```
func (cmp *Component) MountTo(rootID string) {
  cmp.RenderTo(rootID)

  for range cmp.notificationChan {
    var cb js.Callback

    callback := js.NewCallback(func(_ []js.Value) {
      cmp.RenderTo(rootID)
      cb.Release()
    })

    js.Global().Get("window").Call("requestAnimationFrame", callback)
  }
}
```

window.requestAnimationFrame()

Tells the browser that we wish to perform an animation and requests that the browser call a specified function before the next repaint.

Diff VDOM against previous version

```go
func (cmp *Component) RenderTo(rootID string) {
  changes := make([]Change, 0)
  vdom := component.RenderToVDom()
  vdom.Diff(component.OldVDom, &changes, rootID)
  component.OldVDom = vdom
  // ...
```

Diff logic
- **CREATE** change: if no old version of VDOM is present
- **UPDATE** change: otherwise diff properties

Diffing properties
- All properties that are not present in new VDOM version need to be **deleted**
- All properties value of which has changed need to be **updated**

Applying changes

```go
for _, ch := range changes {
  switch ch.Type {
  case "CREATE":
    ch.parentNode.Call("appendChild", ch.domNode)
  case "UPDATE":
    for _, attrName := range ch.attributesToDelete {
      ch.domNode.Call("removeAttribute", attrName)
    }

    for _, attr := range ch.attributesToUpdate {
      ch.domNode.Call("setAttribute", attr.Key(), attr.Val())
    }

    //...
  }
}
```

UPDATE change for text nodes

```
if ch.NewNode.Tag == TEXT_TYPE {
  content := ch.NewNode.Data
  for _, attr := range ch.NewNode.Attr {
    // regexp for Label will be `{{ Label }}`
    re := regexp.MustCompile(fmt.Sprintf(`\{\{\s*%s\s*\}\}`, attr.Key()))
    content = re.ReplaceAllString(content, attr.Val())
  }

  ch.domNode.Set("textContent", content)
}
```

DEMO

What is not implemented in this demo

- Nested components
- No branching in markup (if-else)
- No loops in markup
- Not handling removed nodes or change of order
- A lot of edge cases

Conclusion

Go on WASM

- It works
- It's not perfect
- Please give it a try!

github.com/Gonzih/talks (https://github.com/Gonzih/talks/tree/master/wasm-go-toronto)

Thank you

Building a frontend framework in Go!

18 Oct 2018

Max Gonzih

gonzih.me (//gonzih.me)
gonzih@gmail.com (mailto:gonzih@gmail.com)
@Gonzih (http://twitter.com/Gonzih)