bulim / go-for-javascript-developers

A comparison between Go and Javascript

23 commits 1 branch 0 releases 3 contributors

Branch: master New pull request Find file Clone or download

pazams restructure Latest commit ff11ed3 on Nov 14

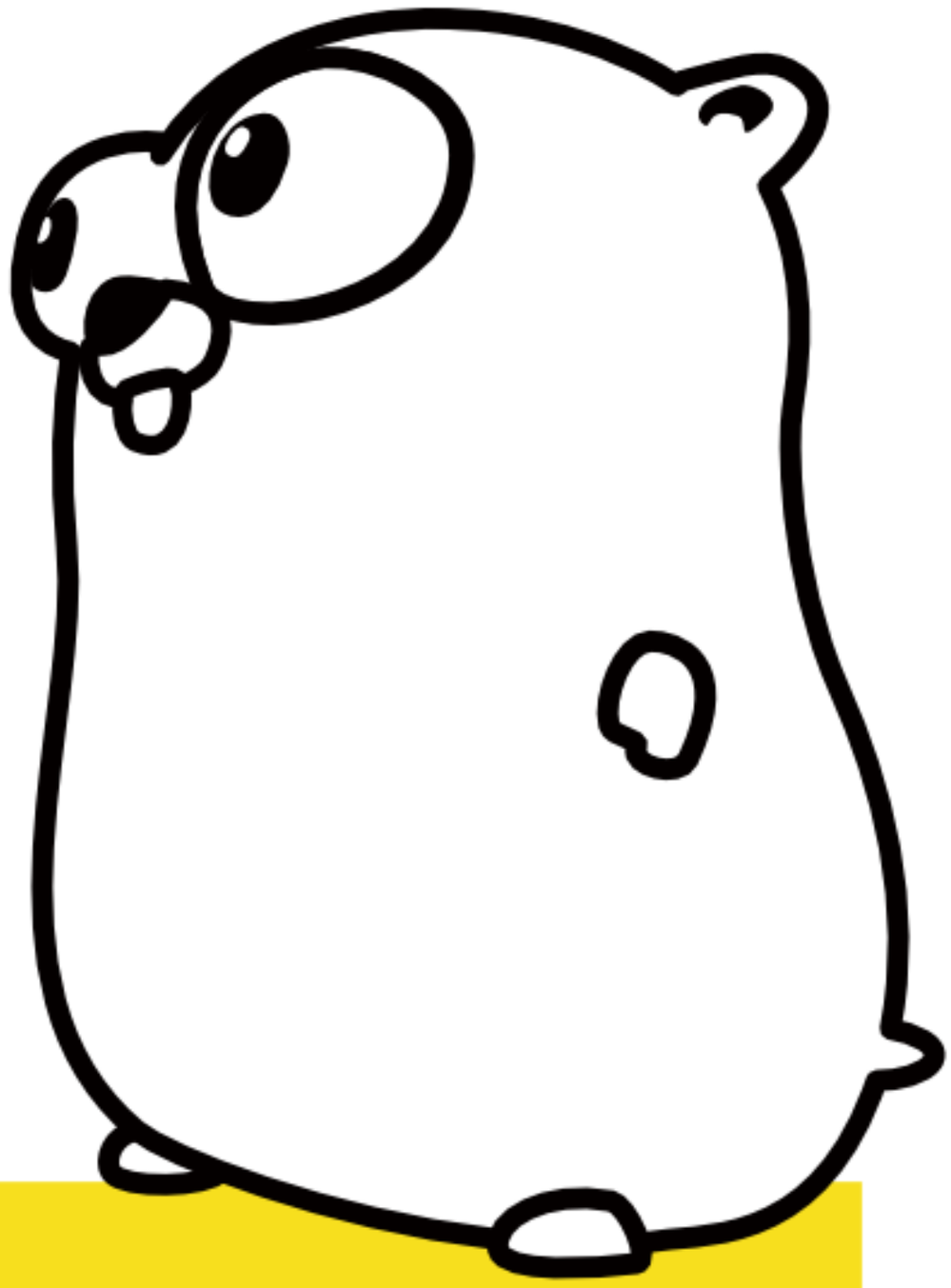
images	add thumb.png	2 months ago
.gitignore	edit .gitignore	3 months ago
README.md	restructure	2 months ago
package.json	edit go foramtting tool	3 months ago

README.md

-- WORK IN PROGRESS --

Maor Zamski

Daniel Singer



Go for Javascript Developers

- Introduction
  - Preface
  - Which language should I use?
  - Semantics
  - Contributions
- Internals
  - (S) Heap/Stack Memory Allocation
  - (S) Garbage Collection
  - (D) Compilation

- Concurrency
- Modules / Packages
  - Spec & Practice
  - Management
- Patterns
- Error Handling
- Keywords & Syntax Comparison
  - (D) `this` keyword
  - (D) `new` keyword
  - (D) bind / method values
  - (S) setTimeout / timer
  - (D) setInterval / ticker
  - (D) String literals
  - (S) Comments
- Variables
  - (D) Values, Pointers, References
  - Types
    - Dynamic VS Static
- Flow control statements
  - (B) Loops and iteration
    - For
    - While
  - (B) If/Else
  - (D) Switch
- Functions
  - (S) first-class functions
  - (D) Multiple returns
  - (S) IIFE
  - (S) Closures
- License

Introduction

Preface

Often, a developer will use more than one programming language at a certain timeframe. Switching back and forth between languages can come with some overhead. These context switches can also result in bugs. For instance, if you switch back and forth between Python and Javascript, there's a likelihood you'll mistake evaluation of an empty array between truthy and falsey. Similarly, if you switch back and forth between Go and Javascript, there's a likelihood you'll mistake `switch` statements default behavior of break/fallthrough. Outlining the differences between languages can help mitigate these potential issues, and make it easier to transition back and forth.

This document compares between two programming languages, Golang (or "Go") and ECMAScript (or "Javascript" / "JS"). The merits of this pairing is the popularity of these languages. That's it. They are not similar, in fact, they are quite different. Javascript is an event driven, dynamically typed and interpreted language, while Go is a statically typed and compiled language.

If you're reading this there's a high chance you already know your Javascript and are just starting with Go. If so, make sure you first complete A tour of go and Effective go.

Which language should I use?

You should always pick the right tool for the right job. Unfortunately, there will never be a simple formula that will instruct you which programming language you should choose to complete a given task.

"Sometimes, science is more art than science, Morty. A lot of people don't get that."

Aside of technical considerations, other considerations, such as community adoption are also important. It was reported that Facebook moved away from the Erlang language because it was hard to find qualified programmers

Having said that, it is worthy to note that Javascript excels in I/O intense applications, and less so in CPU intense applications.

Semantics

Each subchapter/subject is denoted with (D),(S) or (B) to indicate how it compares across both languages with 'mostly **D**ifferent', 'mostly **S**imilar' or 'mostly **B**oth'.

This document uses ECMAScript 2015 (ES6). Also, some subjects will note the run-time environment "NodeJS".

Contributions

This document is a work in progress. Contributions and PRs are most welcomed. If you edit the chapters layout, be sure to rebuild the table of contents by

```
npm install
npm run toc
```

If you edit go code, be sure to format it with (requires mdgofmt-cli)

```
npm run fmt
```

Or just run both commands with

```
npm run build
```

Internals

(S) Heap/Stack Memory Allocation

Concepts such "Heap" and "Stack" are abstracted away in both languages. You do not needed to worry about it. Even though GO has pointers, it uses escape analysis in compile-time to figure out the memory allocation.

(S) Garbage Collection

Garbage collection is implemented in both languages.

(D) Compilation

Go is compiled. Javascript is not, though some Javascript runtimes use JIT compilation. From the developer experience perspective, the biggest effect of compiled languages is compile-time safety. You get compile-time safety with Go, while

in Javascript you can use external code linters to ease the missing of this feature.

Concurrency

The best way to describe concurrency in javascript is with this quote by Felix Geisendörfer:

> Well, in node everything runs in parallel, except your code.

So while your JS runtime may use multiple threads for IO, your own code is getting run just by one. That's just how the *evented* model works. Different JS runtimes offer some options for concurrency or parallelism: NodeJS offers clustering, and Browsers offer web workers.

On the other hand, Go is all about concurrency. It offers Goroutines which enables functions to execute concurrently, and channels to communicate between them. While Go standard library has the "sync" package for synchronization primitives, it encourages more the use of Goroutines and channels, summarized as:

> Do not communicate by sharing memory; instead, share memory by communicating

More on this subject:

- Go Concurrency Patterns
- Advanced Go Concurrency Patterns

Modules / Packages

Spec & Practice

JS

As of es6, the Javascript spec includes a module system, however the external specs of AMD and CommonJS are also popular since the language began to address this issue rather late.

Before es6 modules, the spec only supported the *script* mode, of which every file shares the same top-level global scope. This means that there was no official "file scope" for scripts. In practice, file-module scope was common since it was either introduced by code (`window.moduleA = …`), an external tool (requireJS), or by a runtime that baked-in a module system (NodeJS).

Therefore, it is safe to say that Javascript programs are commonly structured with a 1-to-1 relationship between files and modules with local scope.

Go

Go's import statement and package support were part of the spec from the beginning. In Go there is no file scope, only package scope. As of Go 1.6 (or 1.5 + flag), there's better support for encapsulating dependent packages inside a project with the vendor folder. However, it doesn't attempt so solve everything:

> … this does not attempt to solve the problem of vendoring resulting in multiple copies of a package being linked into a single binary. Sometimes having multiple copies of a library is not a problem; sometimes it is. At least for now, it doesn’t seem that the go command should be in charge of policing or solving that problem.

The differences

A Javascript module can be any valid Javascript type. By exporting an object, it can *package* multiple functionalities. By exporting a function it can surface a single functionality. On the other hand, a Go package, is as its a name- just a package. So while a Javascript module can be directly invoked if it is a function type, this is not a possibility with a Go package.

Another difference is the consumption of other internal components within your project. In Javascript, since each file is (usually) a module, then each of the files that were decoupled from the current file must be imported. On the other hand, in Go, all files within the same package can have access to each other since there is no file scope.

Management

For Javascript development, NPM is the de-facto package manager for NodeJS, and may also be used for client side projects. Bower is also a popular for client side projects.

The `go get` tool will only get you as far as getting a dependency latest master code. This will not suffice if you need accurate dependency management with pinned versions. The Go community came up with several package managers, here's a partial list:

- https://github.com/kovetskiy/manul
- https://github.com/tools/godep
- https://github.com/kardianos/govendor
- https://github.com/FiloSottile/gvt
- https://github.com/Masterminds/glide
- https://github.com/mattn/gom

Patterns

TBD

Error Handling

TBD

Keywords & Syntax Comparison

(D) `this` keyword

JS

Inside an object method, `this` refers to the object (with some exceptions).

Go

In Go, the closest analogy would be receivers inside method functions. You *may* use `this` as a receiver:

```
type Boo struct {
    foo string
}

func (this *Boo) Foo() string {
    return this.foo
}
```

It is more idiomatic to use short variables as receivers. In the example above `b` would have been a better fit over `this`.

(D) `new` keyword

JS

`new Foo()` instantiates an object from `Foo`, a constructor function.

Go

`new(T)` allocates zeroed storage for a new item of type `T` and returns a pointer, `*T`. This is different than Javascript and most other languages where `new` will **initialize** the object, while in Golang it only **zeros** it.

It is worthy to mention that it is idiomatic to name methods with a "New" prefix to denote it returns a pointer to the type

following in the method name. e.g:

```
timer := time.NewTimer(d) // timer is a *time.Timer
```

(D) bind / method values

JS

```
var f = boo.foo.bind(boo2); // when calling f(), "this" will refer to boo2
```

Go

```
f := boo.foo // f(), is same as boo.foo()
```

(S) setTimeout / timer

JS

```
setTimeout(3*1000, somefunction)
```

Go

```
time.AfterFunc(3*time.Second, somefunction)
```

(D) setInterval / ticker

JS

```
setInterval(3*1000, somefunction)
```

Go

```
ticker := time.NewTicker(3 * time.Second)
go func() {
    for t := range ticker.C {
        somefunction()
    }
}()
```

(D) String literals

JS

Strings are initialized with single quotes (`'hello'`) or double quotes (`"hello"`). Most coding styles prefer the single quotes variation.

Go

Strings are initialized with double quotes (`"hello"`) or raw string literals with backticks (`` `hello` ``)

(S) Comments

Both languages use the same `/* block comments */` and `// line comments` .

Variables

(D) Values, Pointers, References

In Javascript there are value types and reference types. Primitives such as `string` and `number` are value types. Objects, including arrays and functions, are reference types.

In Go, there are value types, reference types, and pointers. References types are slices, maps, and channels. All the rest are value types, but have the ability "*to be referenced*" with pointers. The most practical difference to remember between references and pointers, is that while you can use both to mutate the underlaying value (when it is mutable), with pointers you can also reassign it.

JS

```
var a = {
    message: 'hello'
}

var b = a;

// mutate
b.message = 'goodbye';
console.log(a.message === b.message); // prints 'true'

// reassign
b = {
    message: 'galaxy'
}
console.log(a.message === b.message); // prints 'false'
```

Go

```
a := struct {
    message string
}{"hello"}

b := &a

// mutate
// note b.message is short for (*b).message
b.message = "goodbye"
fmt.Println(a.message == b.message) // prints "true"

// reassign
*b = struct {
    message string
}{"galaxy"}
fmt.Println(a.message == b.message) // prints "true"
```

Types

Dynamic VS Static

TBD

Flow control statements

(B) Loops and iteration

For

JS

```
for(let i=0;i<10;i++){
    console.log(i);
}
```

Go

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

While

In Go, the `for`'s init and post statement are optional, effectively making it also a "while" statement:

JS

```
var i=0;
while(i<10){
    console.log(i);
    i++;
}
```

Go

```
i := 0
for i < 10 {
    fmt.Println(i)
    i++
}
```

Iterating over an Array/Slice

JS

```
['Rick','Morty','Beth','Summer','Jerry'].forEach(function(value,index){
    console.log(value + ' at index ' + index);
});
```

Go

```
for i, v := range []string{"Rick", "Morty", "Beth", "Summer", "Jerry"} {
    fmt.Printf("%v at index %d", v, i)
}
```

(B) If/Else

Go's `if` can contain an init statement, with variables declared scoped only to the `if` and `else` blocks.

Go

```
if value := getSomeValue(); value < limit {
    return value
} else {
    return value / 2
}
```

(D) Switch

The switch statement was one of the motivation for writing this document.

Go defaults to break, and `fallthrough` needed for otherwise.

Javascript defaults to fallthrough, and `break` needed for otherwise.

JS

```
switch (favorite) {
    case "yellow":
        console.log("yellow");
        break;
    case "red":
        console.log("red");
    case "pruple":
        console.log("(and) purple");
    default:
        console.log("white");
}
```

Go

```
switch favorite {
case "yellow":
    fmt.Println("yellow")
case "red":
    fmt.Println("red")
    fallthrough
case "pruple":
    fmt.Println("(and) purple")
default:
    fmt.Println("white")
}
```

Functions

(S) first-class functions

Both langauges treat functions as first-class citizens. Both allow functions to be passed as arguments, to be a returned value, to be nested, and have closures.

Function nesting in Javascript can be done both with named and anonymous functions, while in Go this can only be done with anonymous functions.

(D) Multiple returns

Go functions can return multiple values

Go

```
func hello() (string, string) {
    return "hello", "world"
}

func main() {
    a, b := hello()
    fmt.Println(a, b)
}
```

Javascript cannot, however by using destructuring assignment syntax, we can get a similar behavior

JS

```
function hello() {
  return ["hello", "world"];
}

var [a, b] = hello();
console.log(a,b);
```

(S) IIFE

JS

```
(function () {
  console.log('hello');
}());
```

Go

```
func main() {
    func() {
        fmt.Println("Hello")
    }()
}
```

(S) Closures

Both languages have closures. Both require caution when creating closures inside loops. Here are examples in both languages that demonstrate a similar technique to bypass the closure/loop trap:

JS (with bug)

```
var i = 0;
for (;  i < 10 ; i++) {
  setTimeout((function() {console.log(i);}),0);
}
```

JS (solved) (note that using `for(let i=0; …` instead of `var` is a more practical solution)

```
var i = 0;
for (;  i < 10 ; i++) {
  (function (i) {
    setTimeout((function() {console.log(i);}),0);
  }(i));
}
```

Go (with bug)

```
var wg sync.WaitGroup
wg.Add(10)
for i := 0; i < 10; i++ {
    go func() {
        defer wg.Done()
        fmt.Println(i)
    }()
}
wg.Wait()
```

Go (solved)

```
var wg sync.WaitGroup
wg.Add(10)
for i := 0; i < 10; i++ {
    go func(i int) {
        defer wg.Done()
        fmt.Println(i)
    }(i)
}
wg.Wait()
```

License