Digo

Distrbuted Golang

About the Team

- Sida Huang, System Architect

sdh21

- Wenqian Yan, Manager

wy2249 Wenqian Yan

- Yufan Chen, Language Guru

xiaomi388

- Hanxiao Lu, Test Designer

lhxxh Hanxiao Lu

About the Team

wy2249 / Digo

Code | Issues 2 | Pull requests | Actions | Projects 4 | Wiki | Security | Insights | Set

main | 7 branches | 0 tags | Go to file | Add file | Code

sdh21 fix typo — af44b1c 6 hours ago — 330 commits

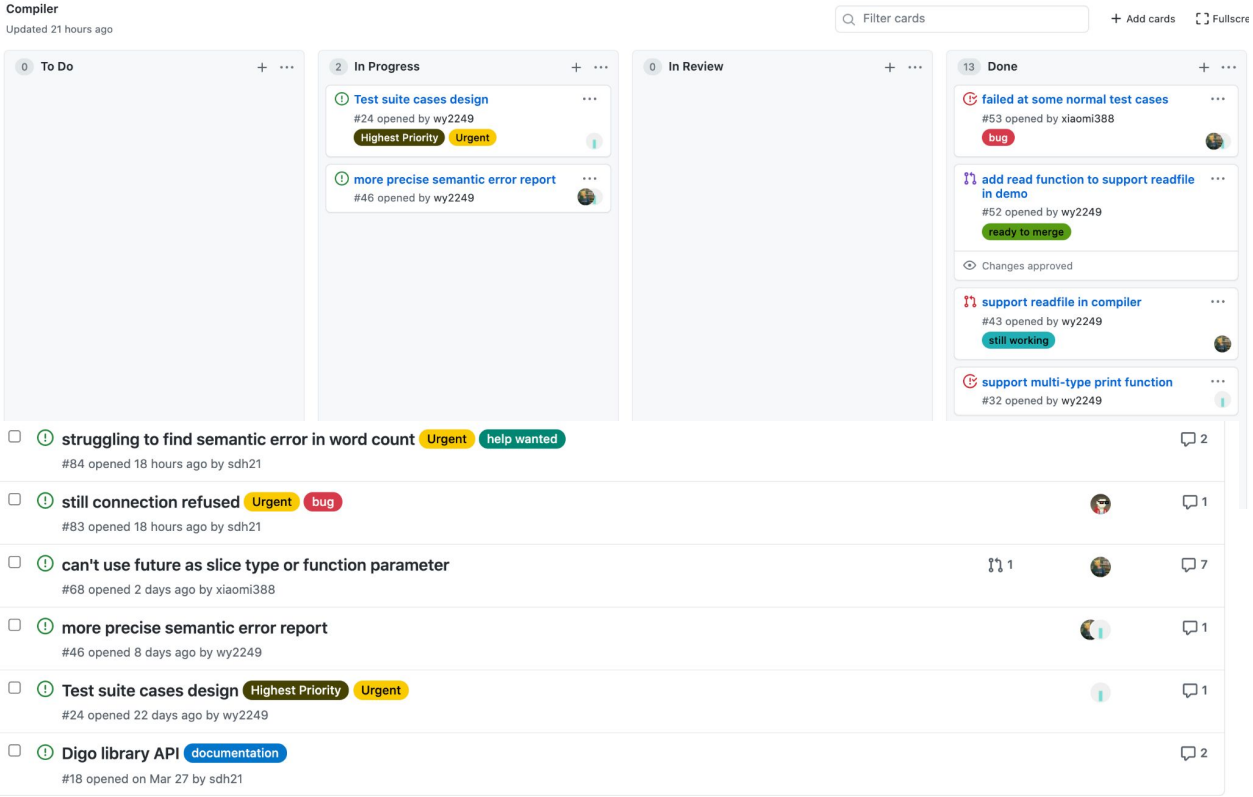
.circleci	Update config.yml	2 days ago
async-remote-lib	fix worker gc leak & add gc leak check in test script	yesterday
demo	fix typo	6 hours ago
digo-compiler	Update parser.mly	9 hours ago
digo-linker	add demo	14 hours ago
digo-test	fix typo	6 hours ago
.gitignore	update tests & fix bugs	3 days ago
.gitmodules	add test framework & finish network::Server::Start	2 months ago
Makefile	fix slice gc bugs	3 days ago
README.md	fixed all warnings in semant	11 hours ago
run-master.sh	add demo	14 hours ago

Feb 14, 2021 – Apr 27, 2021

Contributions: Commits

Contributions to main, excluding merge commits and bot accounts

About the Team



About the Team

Commits on Apr 26, 2021

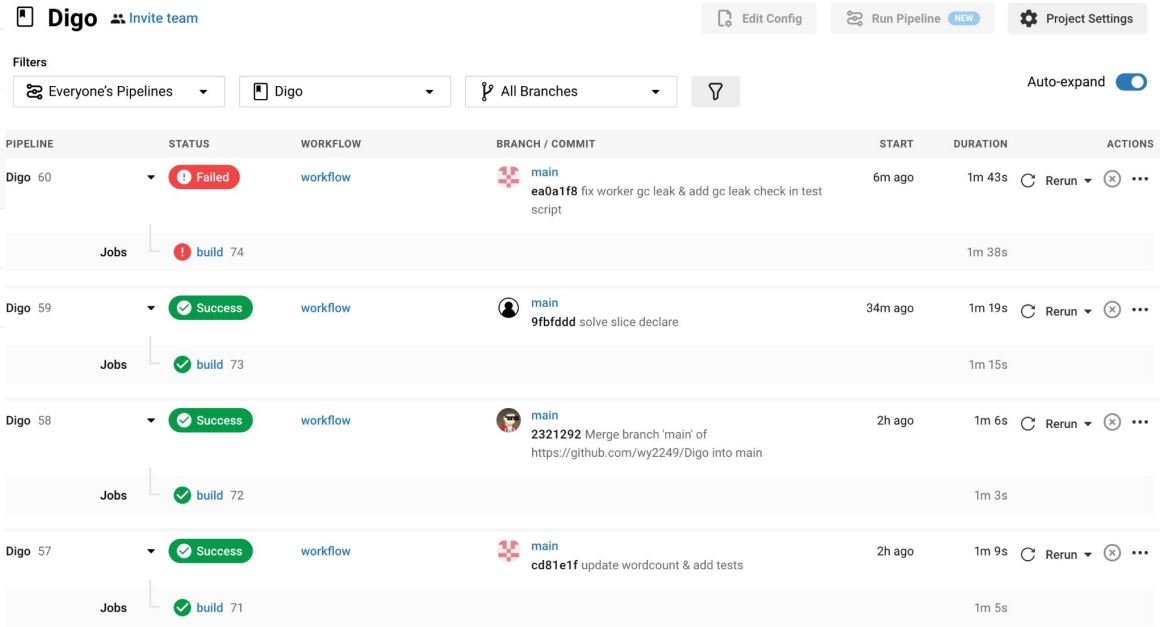
fix worker gc leak & add gc leak check in test script
sdh21 committed 7 minutes ago ✗

solve slice declare
feiyun lu authored and feiyun lu committed 35 minutes ago ✓

Merge branch 'main' of https://github.com/wy2249/Digo into main
xiaomi388 committed 2 hours ago ✓

tweak log at connection fail
xiaomi388 committed 2 hours ago

update wordcount & add tests
sdh21 committed 2 hours ago ✓



Language intro

Digo is an imperative, statically typed language inspired by Golang, but with distributed routines to support master-worker model in distributed system.

network response model? thread management? concurrency?

Not any more. Digo handles them for you!

- With Digo, users can focus on only writing necessary functions and Digo can hide everything else.
- Users only need to give function annotations (async/ async remote) to run the function asynchronously either locally or on workers

continued overview

- Go-like syntax to accelerate learning curve for golangers (short declare, multi values, slice, etc.)
- Hides socket, threading, and network packet serialization/deserialization
- Master distributes work across slave nodes or another local thread (distributed routine)
- Automatic garbage collection (reference count analysis)

Language Features

Basic Syntax

The basic syntax of Digo follows Golang syntax.

For basic data types, Digo supports int, float, bool, string, slice. The semantics of them are totally same with those in Golang.

For basic control flows, Digo supports if statement and for loop statement.

```
func foo(s []string) []string {
    s2 := append(s, "haha")
    return s2
}

func digo_main() void {
  a := []string {"hello", "word"}
  b := foo(a)
  println("%d",len(a))
  println("%d",len(b))
  println("%s", a[1])
  println("%s", b[2])
}
```

```
func gcd(a int, b int) int {
  for (;a != b;) {
    if (a > b) {
        a = a - b
    } else {
        b = b - a
    }
  }
  return a
}

func digo_main() void {
  println("%d", gcd(2,14))
  println("%d", gcd(3,15))
  println("%d", gcd(99,121))
}
```

Async/Await (Distributed Routines)

An async function defines a job to be performed in a local worker or a remote worker.

Async function returns a future object denoting the state of a job.

Await a future object retrieves the return result from the worker.

```
func digo_main() void {
    a := 10
    b := 1.0
    c := try_return_one(a)
    e := try_return_two(a,b)
    f, g := await e

    println("%d", f)
    println("%f", g)
}

async func try_return_one(a int) int {
    return a+10
}

async func try_return_two(a int,b float) (int,float) {

    return a+10,b+20.0
}
```

```
func digo_main() void {
    a := 10
    b := 1.0
    c := try_return_one(a)
    e := try_return_two(a,b)
    f, g := await e

    println("%d", f)
    println("%f", g)
}

async remote func try_return_one(a int) int {
    return a+10
}

async remote func try_return_two(a int,b float) (int,float) {
    return a+10,b+20.0
}
```

Garbage Collection

Values of type slice, string and future are allocated on the heap, therefore Digo will do garbage collection towards them by reference counting.

```
func digo_main() void {
    a := []string{"s1", "s2", "s3"}

    f, g := f1(a, a)
}

func f1(a []string, b []string) ([]string, []string) {
    return b, a
}
```

```
GC Debug: 0x1c5d3f0 is created
GC Debug: 0x1c5d360 is created
GC Debug: ref cnt of String Object, 0x1c5d360 is incremented to 2
GC Debug: 0x1c5d5b0 is created
GC Debug: 0x1c5d7a0 is created
GC Debug: ref cnt of String Object, 0x1c5d7a0 is incremented to 2
GC Debug: 0x1c5d860 is created
GC Debug: 0x1c5da10 is created
GC Debug: ref cnt of String Object, 0x1c5da10 is incremented to 2
GC Debug: 0x1c5d6d0 is created
GC Debug: 0x1c5dd40 is created
GC Debug: ref cnt of Slice Object, 0x1c5dd40 is incremented to 2
GC Debug: ref cnt of Slice Object, 0x1c5dd40 is incremented to 3
GC Debug: ref cnt of Slice Object, 0x1c5dd40 is decremented to 2
GC Debug: ref cnt of Slice Object, 0x1c5dd40 is decremented to 1
GC Debug: ref cnt of Slice Object, 0x1c5dd40 is decremented to 0
GC Debug: ref cnt of Slice Object, 0x1c5d6d0 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5da10 is decremented to 1
GC Debug: ref cnt of Slice Object, 0x1c5d5b0 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5d7a0 is decremented to 1
GC Debug: ref cnt of Slice Object, 0x1c5d860 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5d360 is decremented to 1
GC Debug: ref cnt of Slice Object, 0x1c5d3f0 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5d360 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5d7a0 is decremented to 0
GC Debug: ref cnt of String Object, 0x1c5da10 is decremented to 0
```

Short Declaration (Type Inference)

The := short assignment statement can be used in place of a var declaration with implicit type.

Short declaration also supports multiple values declaration

```
func digo_main() void {
    a, b, c := 3, 4, 5
    println("%d", a)
    println("%d", b)
    println("%d", c)
}
```

Multiple Return Values

Functions can return multiple values

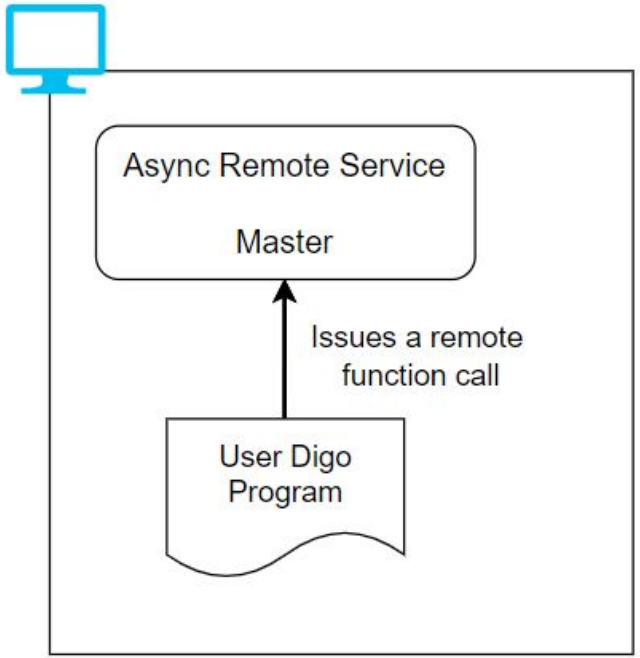
```
func foo() (int, int) {
    return 3, 4
}

func digo_main() void {
    a, b := foo()
    println("%d %d", a, b)
}
```

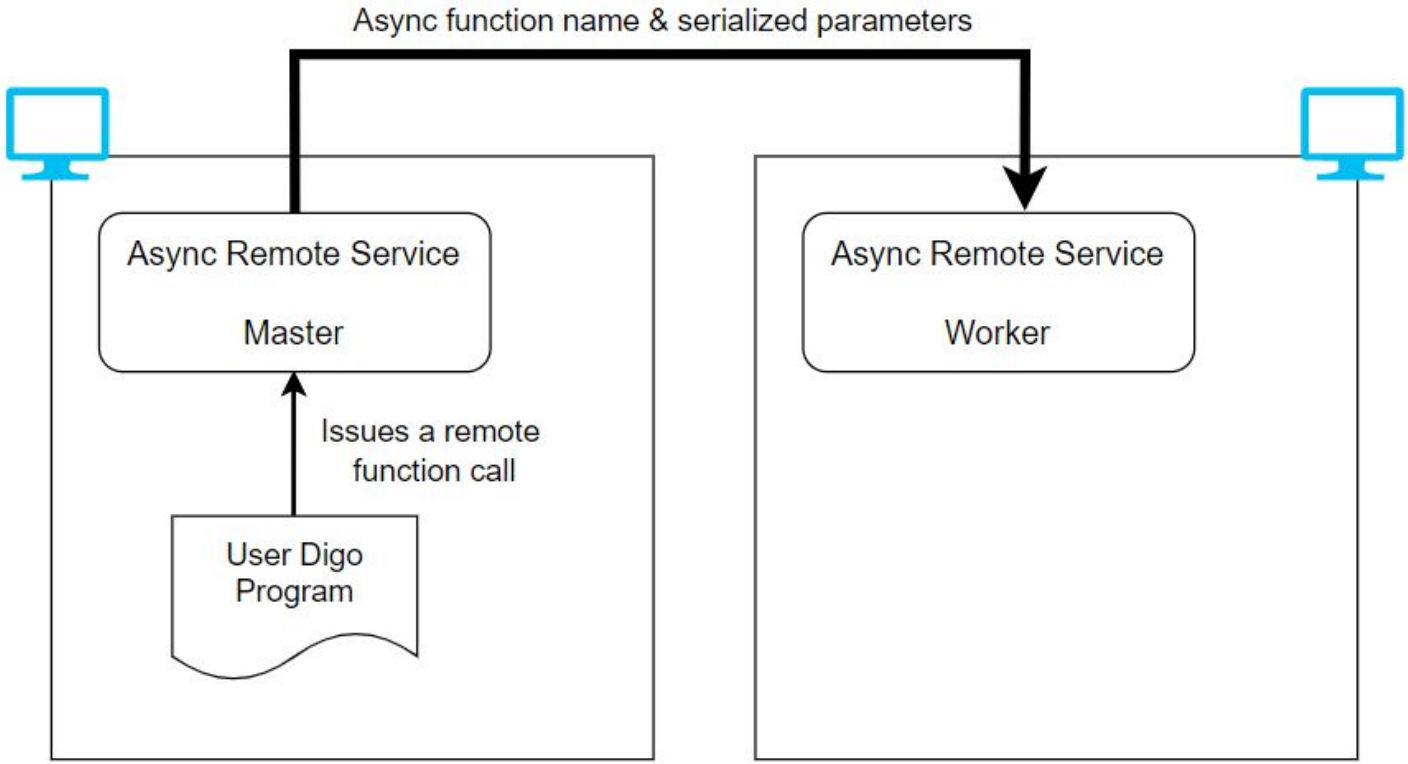
Implementation details

Remote Digo function call

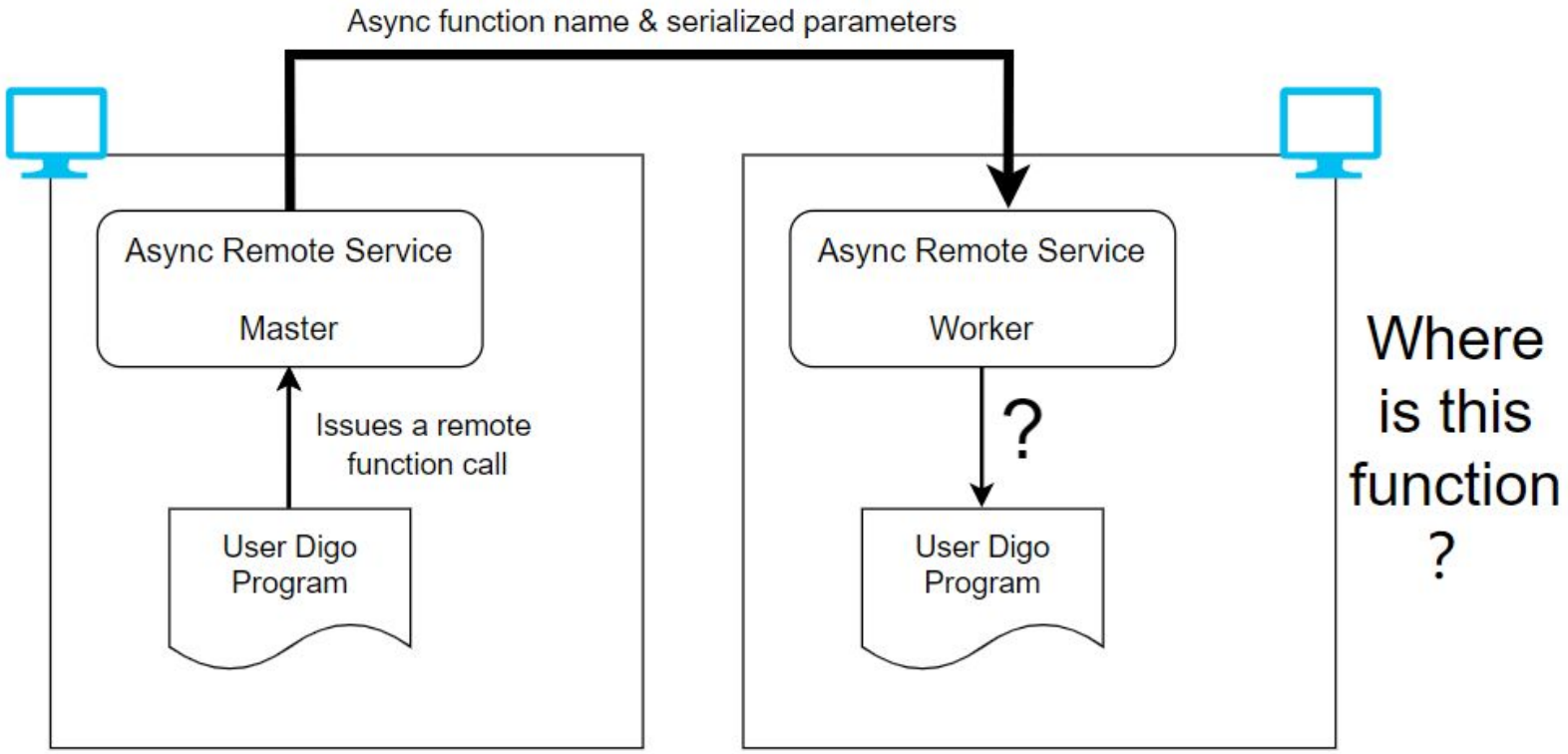
Implementation details



Implementation details



Implementation details



Implementation details

If we want this to be done at compile time……

The Digo program needs to provide an interface, with which the Worker can call a digo function by its name (or by function id).

We do not want to make the code generator too complex, so we have an abstraction layer called Digo Linker to provide this interface and hide all the complexity.

Implementation details

The Digo Linker will also hide the Serialization and Deserialization,

An example of LLVM IR generated by Digo Linker:

Implementation details

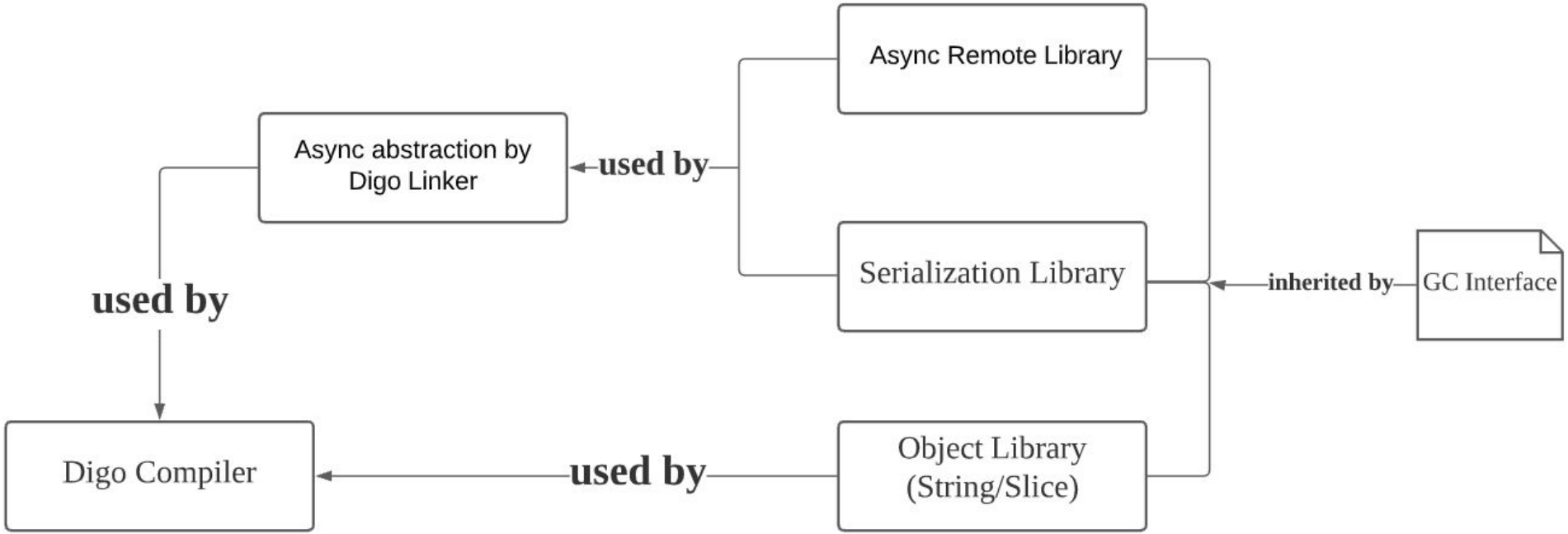
```
192 define i32 @linker_call_function(i32 %func_id, i8* %arg, i32 %arg_len, i8** %result, i32* %result_len) {
193   call void @Debug_Real_LinkerCallFunction(i32 %func_id, i32 %arg_len)
194   %wrapper = call i8* @SW_CreateWrapper()
195   %extractor = call i8* @SW_CreateExtractor(i8* %arg, i32 %arg_len)
196   switch i32 %func_id, label %if.nomatch [
197     i32 0, label %if.func0
198   ]
199
200 if.func0:                                         ; preds = %0
201   %arg0_0 = call i8* @SW_ExtractSlice(i8* %extractor)
202   %arg0_1 = call i8* @SW_ExtractSlice(i8* %extractor)
203   %arg0_2 = call i8* @SW_ExtractSlice(i8* %extractor)
204   %aggResult0 = call { i8*, i8*, i8* } @f1(i8* %arg0_0, i8* %arg0_1, i8* %arg0_2)
205   %aggResult0_tmp_0 = extractvalue { i8*, i8*, i8* } %aggResult0, 0
206   call void @SW_AddSlice(i8* %wrapper, i8* %aggResult0_tmp_0)
207   %aggResult0_tmp_1 = extractvalue { i8*, i8*, i8* } %aggResult0, 1
208   call void @SW_AddSlice(i8* %wrapper, i8* %aggResult0_tmp_1)
209   %aggResult0_tmp_2 = extractvalue { i8*, i8*, i8* } %aggResult0, 2
210   call void @SW_AddSlice(i8* %wrapper, i8* %aggResult0_tmp_2)
211   call void @SW_GetAndDestroy(i8* %wrapper, i8** %result, i32* %result_len)
212   call void @__GC_DecRef(i8* %aggResult0_tmp_0)
213   call void @__GC_DecRef(i8* %aggResult0_tmp_1)
214   call void @__GC_DecRef(i8* %aggResult0_tmp_2)
215   call void @__GC_DecRef(i8* %arg0_0)
216   call void @__GC_DecRef(i8* %arg0_1)
217   call void @__GC_DecRef(i8* %arg0_2)
218   br label %if.end
```

Jump Table

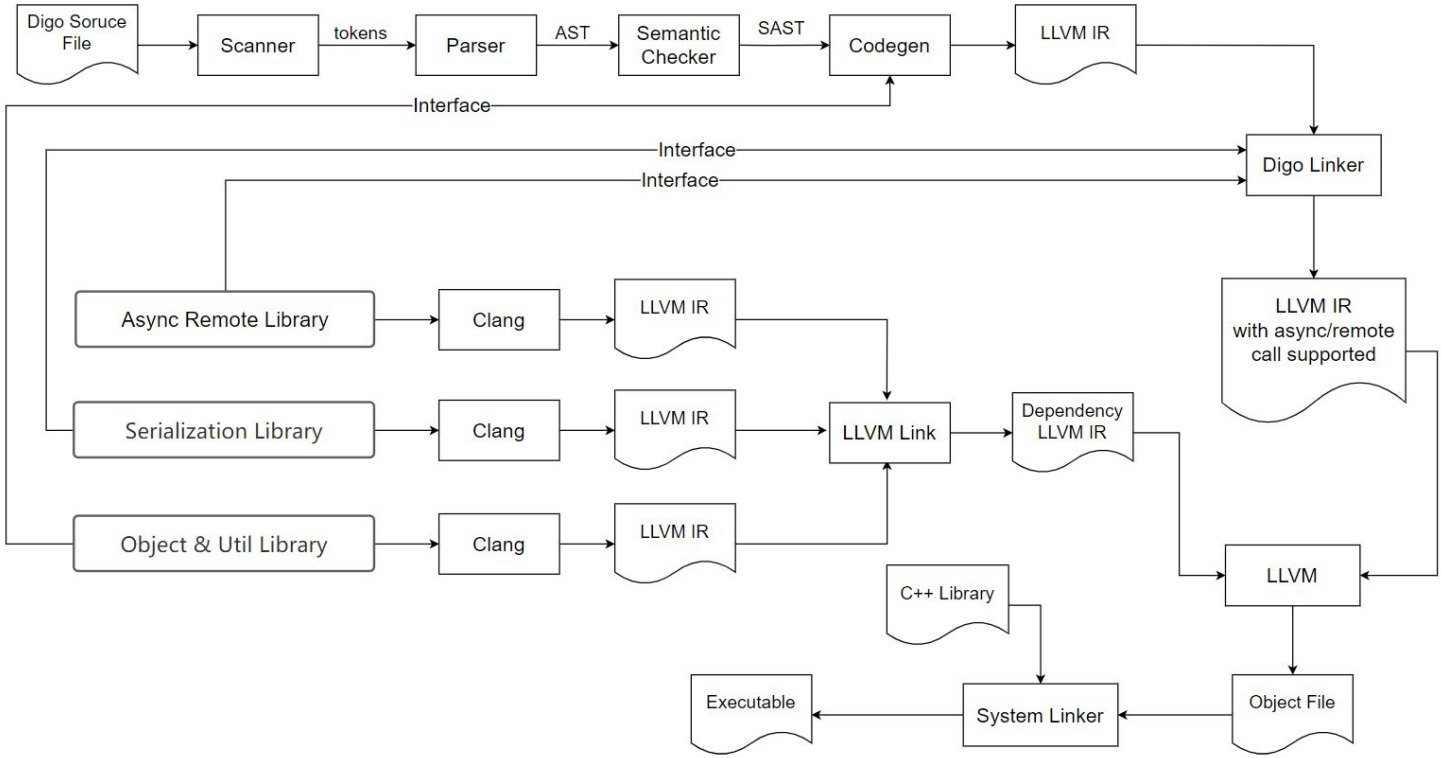
Argument de-serialization

Return Value serialization

Implementation details



Implementation details



Test suite

- Divide test cases into 8 categories and each one is responsible for a particular feature of digo

  (Async, Basic, ControlFlow, GC, Remote, Semantic, Syntax, Utils)

- An automated test script that compares results with expected value.

Future work

- More flexibility: support future object being passed as function parameter.
- Digo Objects in different language: current in C++, maybe in digo lang
- More complicated GC: Current GC is simple...
- More control flow: break and continue
- More built-in functions: Gather: await all future objects in a future object list.
- Optimizer: Delete unnecessary llvm command produced by compiler and merge multiple commands into one

Demo

“Hello world” in distributed system: **word count!**