Dive into testing package

~ Part of Fuzzing Test ~

自己紹介

- 渋谷拓真
  - Twitter/GitHub: @sivchari
  - CyberAgent Inc.
  - CA Go Next Experts

- 過去登壇
  - Go Conference 2021 Autumn
  - Go Conference 2022 Spring
  - Go Conference mini 2022 in Autumn SENDAI

本日のテーマ

- testing packageとは
- go testの実行の仕組み
- Fuzzing testについて
- Fuzzing testの内部実装をみてみる
- まとめ

今回の発表について

- 以前話したDive into testing packageの続編です
- わからなくても大丈夫なように1から全て解説します
- この発表をきくことでtesting packageの読み方、仕組みがわかります

testing packageとは

- Goの標準パッケージの1つ
- Goのテストを書くためのメソッド、構造体が定義されている
- Goのテストはtesting packageと標準コマンドの1つであるgo testを用いてテストを実行することができる
  - testing.T
  - testing.B
  - testing.F
  - ExampleTest

go

```
package main

import “testing”

func TestAdd(t *testing.T) {
  i, j, want := 1, 1, 2
  got := Add(i, j)
  if want != got { t.Error(“oops”) }
}
```

- go testってどう動いてる？
- PrefixがTestかどうかってどこで確認してる？
- なぜTestの引数は必ず*testing.Tである必要があるのか？

そもそもgo testってどう動いてる？

- go testコマンドはsrc/cmd/internalに存在する(e.g. run, fix, vet etc)
- 定義されているコマンドは3つ
  - go test
  - go test flagの説明コマンド
  - Goのtestの書き方の説明(ファイル名、関数名 etc)
- 今回はfunc initで定義されているCmdTestのrunTestをみていく

runTestの概要

- cache resultの確認
- go testと一緒に渡されたflagによる設定を反映する(coverage, fuzzing …)
- fuzzingを実行できるバージョン、OS、Archか確認
- packageをロードして実行するテストが存在するかの確認
- go testを実行するための設定やactionのセットアップ

runTest

```
for _, p := range pkgs {
    buildTest, runTest, printTest, err := builderTest(&b, ctx, pkgOpts, p, allImports[p])
    …
}
root := &work.Action{Mode: “go test”, Func: printExitStatus, Deps, prints}
b.Do(ctx, root)
```

builderTestとAction type

- builderTestは*Action typeのbuilderTest, runTest, printTestを返す
- builderTest, runTest, printTestもそれぞれが参照している状態
  - build <- run <- print etc.. (vet, install, link) 条件により依存関係が変化する
- go install/go getなどもActionとして実行される

今回知っておくこと

- FuncにActionの実行内容がはいる
- DepsはActionを実行するために事前解決が必要な依存関係を表す

builderTest

```
package testing


func builderTest() {
  // Build Package structs describing
  // pmain - pkg.test binary
  // ptest - package + test files
  // pxtest - package of external test files
  pmain, ptest, pxtest, err := load.TestPackagesFor(ctx, pkgOpts, p, cover)
}
```

pmain, ptest, pxtest

- pmain, ptest, pxtestはパッケージ
  - 内部のstructはファイルを生成するために必要な importした内容やファイル名、go:embedのマッピング、ldflagsなどの情報を全て保持している
- pmain: testファイルをbinaryにしたときのentry pointになるmain package
  - _testmain.goというファイル名でTestmainのentry pointとして書き込まれる
- ptest: `package p`テストファイルを追加してコンパイルしたpackage p
- pxtest: `package p_test`テストファイルをコンパイルしたpackage

TestPackagesFor

```
pmain, ptest, pxtest = TestPackagesAndErrors(ctx, opts, p, cover)

// snip

// DepsErrorCheck

if pmain.Error != nil || len(pmain.DepsErrors) > 0 {}

if ptest.Error != nil || len(ptest.DepsErrors) > 0 {}

if pxtest. != nil && (pxtest.Error != nil || len(pmain.DepsErrors) > 0) {}
```

TestPackagesAndErrors

```
// Do initial scan for metadata needed for writing _testmain.go

// Use that metadata to update the list of imports for package main.

// The list of imports is used by recompileForTest and by the loop

// afterward that gathers t.Cover information.

t, err := loadTestFuncs(ptest)
```

loadTestFuncs

```
func loadTestFuncs(ptest *Package) (*testFuncs, error) {
  t := & testFuncs { Package: ptest }
  for _, file := range ptest.TestGoFiles {
      t.load(filepath.Join(ptest.Dir, file), “_test”, &t.ImportTest, &t.NeedTest)
  }
  for _, file := range ptest.XTestGoFiles {
      t.load(filepath.Join(ptest.Dir, file), “_xtest”, &t.ImportTest, &t.NeedTest)
  }
}
```

(&testFuncs).load

```
case isTest(name, “Fuzz”):
  err := checkTestFunc(n, “F”)
  if err != nil {
    return err
  }
  t.Tests = append(t.FuzzTargets testFunc{ pkg, name, “”, false })
  *doImport, *seen = true, true
```

isTest

```
// name = FuzzXXX, prefix =Fuzz
func isTest(name, prefix string) bool {
  if !strings.HasPrefix(name, prefix) {}
  if len(name) == len(prefix) // FuzzだとOK
  rune, _ := utf8.DecodeRuneInString(name[len(prefix):])
  // FuzzA, Fuzz_はOK, FuzzaだとNG
  return !unicode.IsLower(rune) // Fuzzのあとは大文字化のチェック
}
```

checkTestFunc

```
func checkTestFunc(fn *ast.FuncDecl, arg string) error {
  if !isTestFunc(fn, arg) {
    // snip
  }
  if fn.Type.TypeParams.NumFields() > 0 {
    // snip
  }
}
```

isTestFunc

```
func isTestFunc(fn *ast.FuncDecl, arg string) error {
    // 名前つきimportの場合testingのidentまでチェックできないため*Fや*something.FならOKとする
    if name, ok := ptr.X.(*ast.Ident); ok && name.Name == arg {
        return true
    }
    if sel, ok := ptr.X.(*ast.SelectorExpr); ok && sel.Sel.Name == arg {
        return true
    }
    return false
}
```

formatTestMain

```
func formatTestMain(t *testFuncs) error {
    var buf bytes.Buffer
    if err := testmainTmpl.Execute(&buf, t); err != nil {}
    return buf.Bytes(), nil
}
```

- pmainのGoTestFilesで設定した_testmain.goの内容
- text/templateでここまでに加工した_testmain.goの内容を吐き出す
- pmainのInternal.TestmainGoにも同様の内容がセットされる
- testmainTmpl

Example

```
func TestAdd(t *testing.T) {
  i, j, want := 1, 1, 2
  got := Add(i, j)
  if want != got {}
}
```

- 自分でビルドしてtestmainTmplの内容をプリントするようなGo binaryで結果をみてみる

Example

```
var tests = []testing.InternalTest{
    {"TestAdd", _test.TestAdd},
}
var benchmarks = []testing.InternalBenchmark{}
var fuzzTargets = []testing.InternalFuzzTarget{}
var examples = []testing.InternalExample{}
func init() {
    testdeps.ImportPath = "a"
}
func main() {
    m := testing.MainStart(testdeps.TestDeps{}, tests, benchmarks, fuzzTargets, examples)
    os.Exit(m.Run())
}
```

builderTest

```
if !cfg.BuildN {

  // writeTestmain writes _testmain.go

  if err := os.WriteFile(testDir+”_testmain.go”, *pmain.Internal.TestmainGo, 0666)
err != nil {}

}
```

- BuildNはgo build -n (build結果はprintするが実行しない)
- TestmainGoを_testmain.goに吐き出す

compile and link

b.CompileAction(work.ModeBuild, work.ModeBuild, pmain.Objdir) = testDir

a := b.LinkAction(work.ModeBuild, work.ModeBuild, pmain)

- testDir+_testmain.goの内容をcompile + linkする

(&Builder).Do

- Actionの依存性を解決しながら実行する
- BuilderのActionとSemaphoreがあり依存性が0のActionから実行する
- handleの内部でもhandleが実行終了するたびにpendingを減らし0になれば実行できるActionとみなしAction Queueにputされる
- Actionがrootであれば全て実行したことになる終了する
- Actionはgoroutineで実行されるため毎回順番は変化する

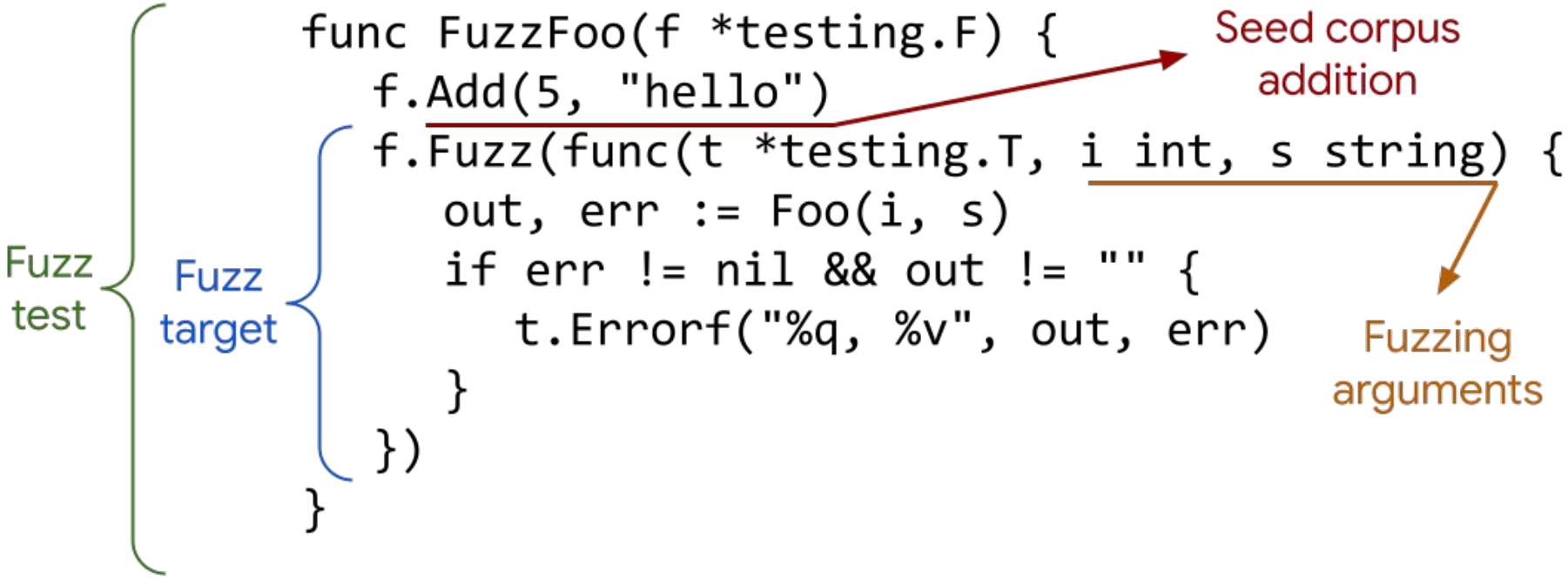
(&Builder).Do


```
{
    "ID": 0,
    "Mode": "go test",
    "Package": "",
    "Deps": [
        1
    ],
    "Priority": 137,
}
```

Fuzzing Testについて

- Go1.18から導入された機能
- テスト手法の1つで、想定していない入力を与えることでバグを発見する
- 想定していない値はseedをベースとしたランダムなものである
- 開発者が想定していなかったテストケースや考慮漏れを発見することが目的
- go test -fuzz=FuzzXXXで実行可能(1個しか実行できない)
- fuzztimeを設定しないと無限に実行してしまうため注意が必要
- Go本体ではserealize/deserialize/parse周りで使われている(archive, compress, encoding, image etc…)
- Fuzzing Testが失敗した場合はtestdata/fuzz/FuzzTestNameで吐き出され次の実行はそこから行われる

Fuzzing Testの内部実装をみていく



https://go.dev/security/fuzz/

Fuzzing Testの内部実装をみていく(seed corpus)

```
func (f *F) Add(args …any) {
  var values []any
  for i := range args {
    if t := reflect.TypeOf(args[i]); !supportedTypes[t] { panic(“err”) }
    values = append(values, args[i])
  }
  f.corpus = append(f.corpus, corpusEntry{Values: values, IsSeed: true, Path:
fmt.Sprintf(“seed#%d”, len(f.corpus))})
}
```

Fuzzing Testの内部実装をみていく(seed corpus)

- (&F).Addでcorpusを渡す
- ここで渡されたcorpusはFuzzing Testの後に渡すと使われない
- Fuzzing Testが対応していない型をcorpusとして渡すとpanicする(composite literalが対応していないが、[]byteは対応している)

Fuzzing Testの内部実装をみていく(entry point)

- fuzzingの実装はinternalにあるため直接は使えない
- 自動生成されたコードのなかでtesting.internal.TestDepsがtesting.MainStartに渡されるため隠蔽しながら実装を渡している(そのためtestDepsはinterface/privateになっている)
- InternalFuzzingTestはテストの名前と実際のFuzzingTestのメソッドを持つstruct
- つまりFuzzテストの実行はtesting.MainStartから始まることがわかるためentry pointから見直してみる

Fuzzing Testの内部実装をみていく(entry point)

- MainStartは*testing.M.Run()を呼ぶ
  - 明示的に書かなくても全てのテストは M.Run()から開始する

fuzzingOk := runFuzzing(m.Deps, m.fuzzTargets)

- testDeps interfaceとInternalFuzzTargetsを渡す
- 失敗だった時にfuzzWorker(worker process)ならExitCode(70)
- 違うならExitCode(1)

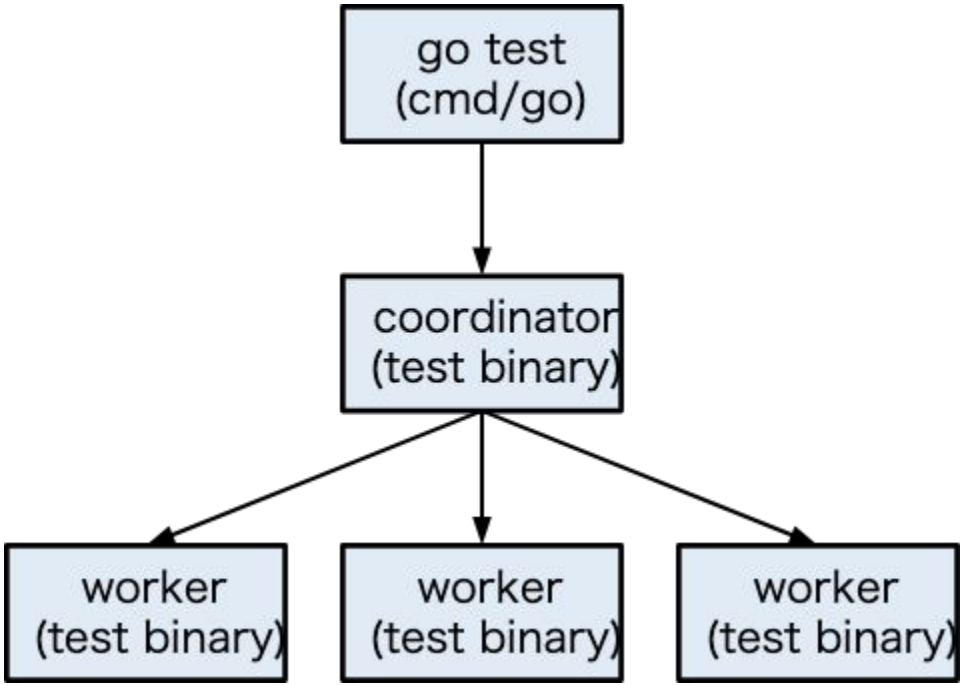
Fuzzing Testの内部実装をみていく(entry point)

- MainStartは*testing.M.Run()を呼ぶ
  - 明示的に書かなくても全てのテストは M.Run()から開始する

fuzzingOk := runFuzzing(m.Deps, m.fuzzTargets)

- testDeps interfaceとInternalFuzzTargetsを渡す
- 失敗だった時にfuzzWorker(worker process)ならExitCode(70)
- 違うならExitCode(1)

Fuzzing Testの内部実装をみていく(entry point)



https://jayconrod.com/posts/123/internals-of-go-s-new-fuzzing-system

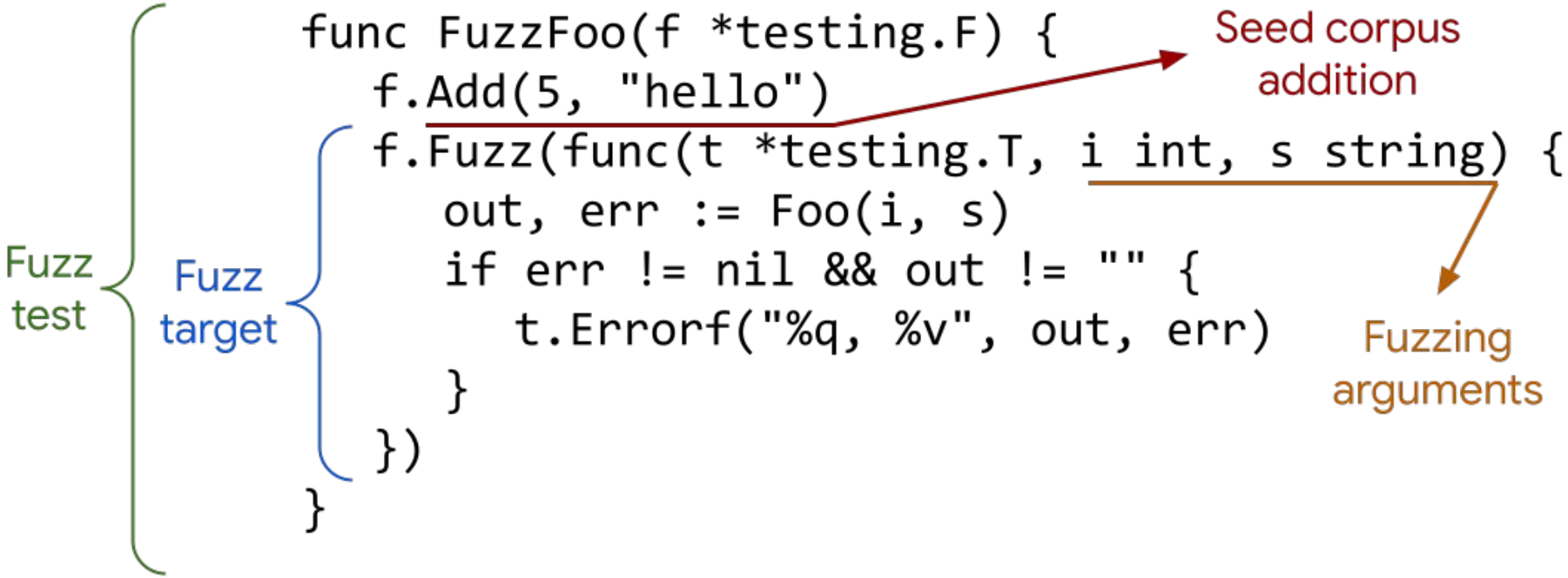
Fuzzing Testの内部実装をみていく(entry point)

- -test.fuzzworkerフラグでfuzzworkerを起動できる(=GOMAXPROCS)
- このフラグとFuzzing Testのキャッシュを使うフラグは文書化されていない
  - go help testflagで出てこない
- Fuzzing Testが失敗する場合大半のケースにおいてプログラムはクラッシュする(0での割り算)
  - workerとして別プロセスで切り出して coordinatorがそれを記録できる必要がある
- signalが飛ぶまでgoroutineで起動する

Fuzzing Testの内部実装をみていく(fRunner)

- deferでrecoverしている(panic対策)
- 失敗していた場合atomic.Uint32のカウントを増やして他のテストを終了させる(先に他のテストは実行するためFuzzing Testのみ)
- 失敗していない場合signalを飛ばして終了する
- エラーが起きた場合FAILログを流して確実にpanicさせる(doPanic func)
- f.Fuzzのサブテストがある場合は先に全てのsubtestを終わらせる
- 最後にFuzzingTestが実行される

Fuzzing Testの内部実装をみていく



https://go.dev/security/fuzz/

Fuzzing Testの内部実装をみていく(Fuzz)

- Fuzzが2回以上呼ばれていた場合はpanicする
- 関数以外を渡した場合はpanicする
- 第1引数が*testing.Tじゃない場合と引数が*testing.Tのみだとpanicする
- 返り値を渡した場合panicする

Fuzzing Testの内部実装をみていく(Fuzz)

```
if fnType.Kind() != reflect.Func {
	panic("testing: F.Fuzz must receive a function")
}
if fnType.NumIn() < 2 || fnType.In(0) != reflect.TypeOf((*T)(nil)) {
	panic("testing: fuzz target must receive at least two arguments, where the first argument is a *T")
}
if fnType.NumOut() != 0 {
	panic("testing: fuzz target must not return a value")
}
```

Fuzzing Testの内部実装をみていく(Fuzz)

- FuzzWorkerではない(Coordinator/SeedCorpus)の場合はCorpusのチェックを行う
  - 型と値が一致しているか (reflect package)
  - ダメだった場合はFatalで終了する
  - TODOとしてダメだった場合の PosなどをReportするがある
- testdata/fuzzにデータがあるかをチェックするなければnil

Fuzzing Testの内部実装をみていく(mode check)

```
switch f.fuzzContext.mode {
case fuzzCoordinator:
  f.fuzzContext.deps.CoordinateFuzzing(...more args)
case fuzzWorker:
   // runは渡した関数をgoroutineで実行している(go tRunner(t, func(t *T)))
  f.fuzzContext.deps.RunFuzzWorker(func(e corpusEntry) error {run})
}
```

Fuzzing Testの内部実装をみていく(mode check)

```
switch f.fuzzContext.mode {
case fuzzCoordinator:
  f.fuzzContext.deps.CoordinateFuzzing(...more args)
case fuzzWorker:
   // runは渡した関数をgoroutineで実行している(go tRunner(t, func(t *T)))
  f.fuzzContext.deps.RunFuzzWorker(func(e corpusEntry) error {run})
}
```

Fuzzing Testの内部実装をみていく(CoordinateFuzzing)

- Parallel数をGOMAXPROCSなどで決定する
- **決定した数のworkerを作成してgoroutineで実行する**
- 内部は複数のchannelが動いており、signalやcontextでcacnelや終了のsignalを管理している
- それぞれprivate reciver functionのcoordinateを実行する
  - 無限ループになっておりinputを待機する

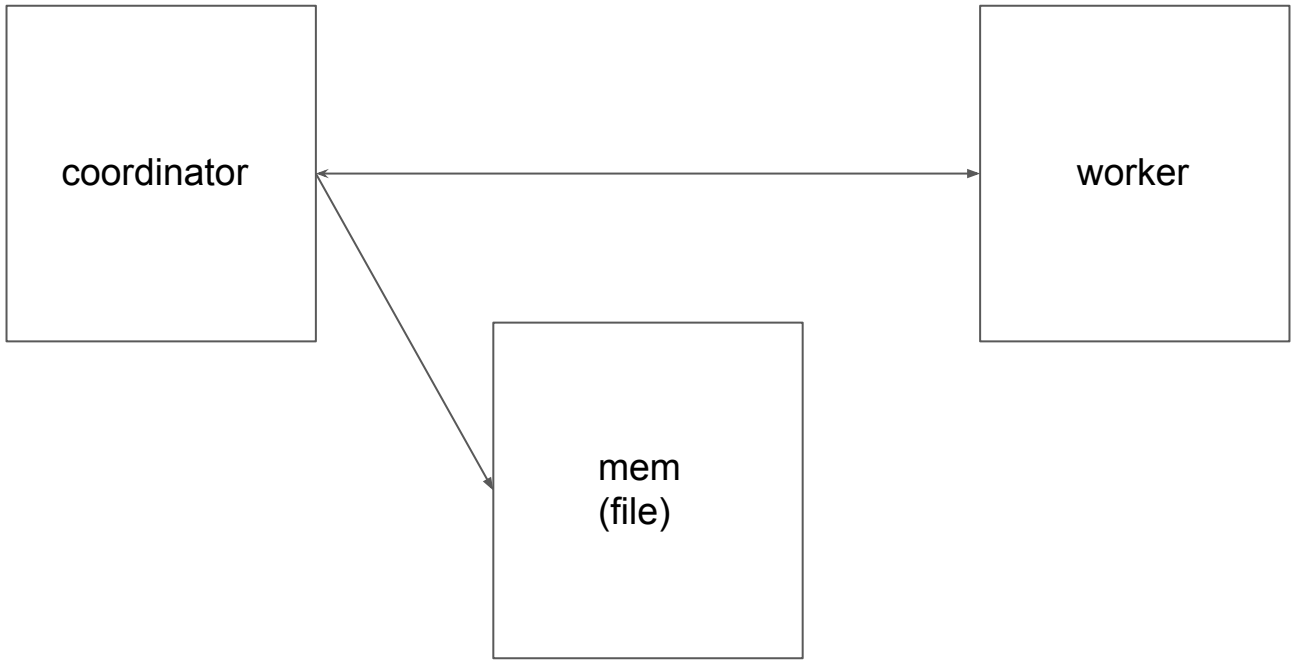
Fuzzing Testの内部実装をみていく(CoordinateFuzzing)

- seedデータはshared mem(file)として共有している
  - ここのファイルに使用したコーパスが入る
- peekInputで入力値をみる
  - 初回であればinputC channelにcoordinatorのinput channelを参照させる
  - 直下にあるselect caseでsendし、sendInputメソッドを呼び出す
  - inputCはcoordinatorのinput channelが受け取るため、(&worker).coordinate()内のselectが受け取る

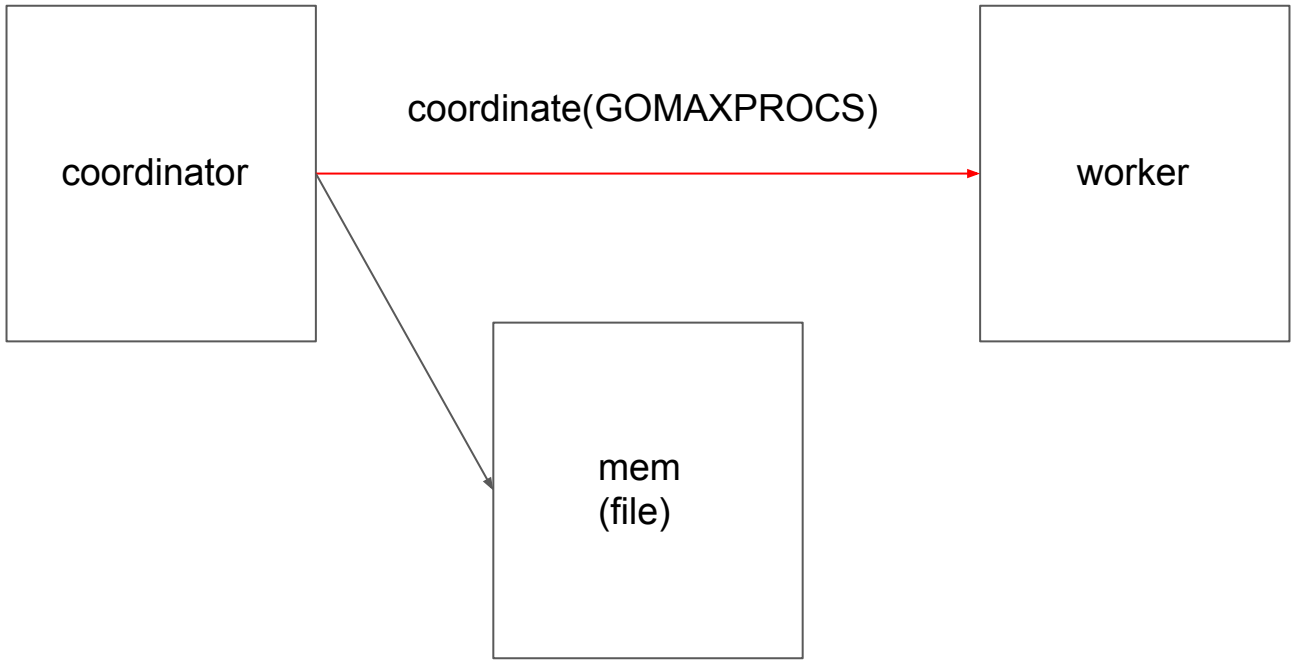
Fuzzing Testの内部実装をみていく(coordinate)

- 受け取ったデータをもとに関数を実行する
- 結果をworkerはresult channelを使いcoordinatorにsendする
- coordinatorは受け取ったデータをベースにstats/coverageなどを計算する
- その後新たなcorpusをファイルに書き込む
- 今のデータを最小化できる場合はminimizationのqueueにputされる
- minimizationのqueueに入った場合はminimization用のchannelが受け取る
- minimizationに入るとcorpusをminimizeする
- minimizeされたresultを再度result channelに流すことで次のfuzzが実行される

Fuzzing Testの内部実装をみていく(全体構成図)



Fuzzing Testの内部実装をみていく(全体構成図)



Fuzzing Testの内部実装をみていく(coordinate)

```
for {
    if !w.isRunning() {
        if err := w.startAndPing(ctx); err != nil {
            return err
        }
}
```

Fuzzing Testの内部実装をみていく(startAndPing)

```
if err := w.start(); err != nil {
    return err
}
```

Fuzzing Testの内部実装をみていく(start)

```
fuzzInR, fuzzInW, err := os.Pipe()

if err != nil {}

defer fuzzInR.Close()

fuzzOutR, fuzzOutW, err := os.Pipe()

defer fuzzOutW.Close()


cmd.Start()
```

Fuzzing Testの内部実装をみていく(start)

cmd.Start()

/var/folders/rg/jsdry3fd7w13xyf7s88jb3b11y75q3/T/go-build1568676446/b001/a.test -test.fuzzworker -test.paniconexit0 -test.fuzzcachedir=/Users/s15301/Library/Caches/go-build/fuzz/a -test.timeout=10m0s -test.fuzz=Fuzz

新しいプロセスをfuzzworkerモードで立ち上げている

Fuzzing Testの内部実装をみていく(mode check)

```
switch f.fuzzContext.mode {
case fuzzCoordinator:
  f.fuzzContext.deps.CoordinateFuzzing(...more args)
case fuzzWorker:
   // runは渡した関数をgoroutineで実行している(go tRunner(t, func(t *T)))
  f.fuzzContext.deps.RunFuzzWorker(func(e corpusEntry) error {run})
}
```

Fuzzing Testの内部実装をみていく(RunFuzzWorker)

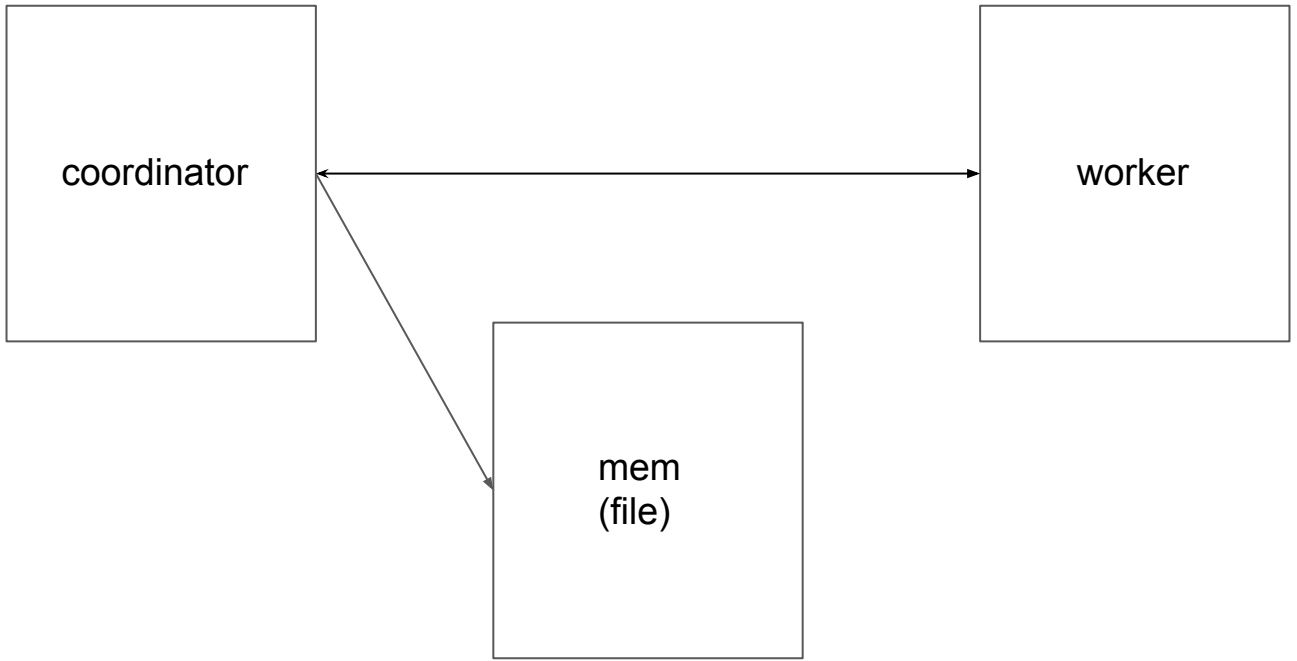
https://github.com/golang/go/blob/40c7be9b0f92d88b90a5aa35838d786579e4fa1d/src/internal/fuzz/worker.go#L349

```
// coordinatorのfuzzInR, fuzzOutWを継承する

comm, err := getWorkerComm()

// snip

return srv.serve(ctx)
```

Fuzzing Testの内部実装をみていく(srv.serve)

```
func (ws *workerServer) serve(ctx context.Context) error {
	// snip
	for {
		var resp any
		switch {
		case c.Fuzz != nil:
			resp = ws.fuzz(ctx, *c.Fuzz)
		case c.Minimize != nil:
			resp = ws.minimize(ctx, *c.Minimize)
		case c.Ping != nil:
			resp = ws.ping(ctx, *c.Ping)
		default:
			return errors.New("no arguments provided for any call")
		}
		if err := enc.Encode(resp); err != nil {
			return err
		}
	}
}
```

Fuzzing Testの内部実装をみていく(全体構成図)



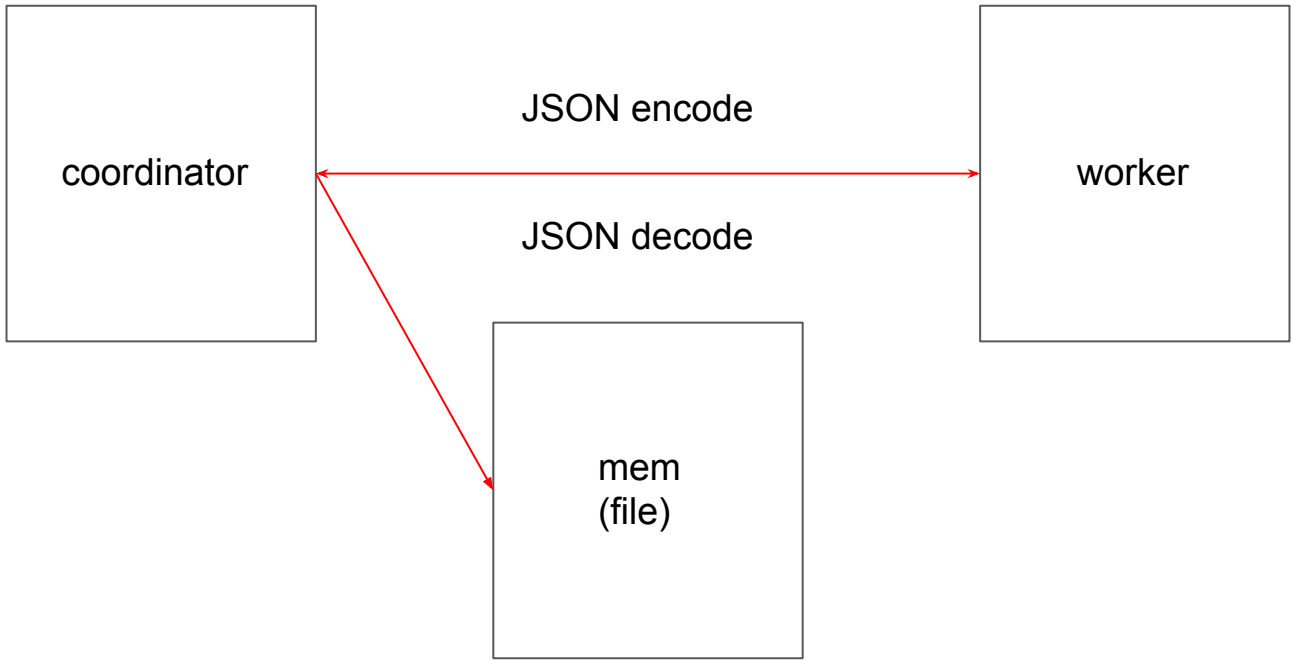
Fuzzing Testの内部実装をみていく(ws.fuzz)

```
func (ws *workerServer) serve(ctx context.Context) error {
    // テストを実行したり、coverageをとったりする
  for {
        select {
        case <-ctx.Done():
            return resp
        default:
            // snip
            ws.m.mutate(vals, cap(mem.valueRef()))
        }
}
```

Fuzzing Testの内部実装をみていく(ws.mutate)

- randで乱数で取得したcorpusの要素の型を確認する
- 型に合わせてintとかなら擬似的な数字stringならmutator.goにあるbyteSliceMutatorsのどれか1つを実行する
  - XOR/Swap/Shuffle/Overwrite/Remove/Insert etc…

Fuzzing Testの内部実装をみていく(全体構成図)



Fuzzing Testの内部実装をみていく(callLocked)

```
enc := json.NewEncoder(wc.fuzzIn)
dec := json.NewDecoder(&contextReader{ctx: ctx, r: wc.fuzzOut})
if err := enc.Encode(c); err != nil {}
return dec.Decode(resp)
```

- fuzzIn/fuzzOutが共有しているos.Pipe
- ここで受け取ったresponseをcoordinatorにchannelで渡す
- その後shared memに書き込む
- 次のworker requestを作成してRPC通信をする

まとめ

- go testはtext/templateを使った自動生成されたプログラムである
- go testは明示的でないフラグが複数ある
- fuzzing testのcorpusはsharedにあるファイルを読み込んでいる
- fuzzingの結果はcoordinatorとworkerがos.Pipe/shared memを使って同期することでランダムな値を導き出している
- fuzzing周りのchannelの使い方を追うのは難しい

おわり