37 | 工具背后的工具：从代码覆盖率到模糊测试

2023-01-03 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

讲述：郑建勋

时长 10:23　大小 9.49M

你好，我是郑建勋。

开始今天的学习之前，我想先问你一个问题，你认为什么样的代码才是高质量的？

代码覆盖率，也就是有效代码的比例为我们提供了一种重要的衡量维度。

代码覆盖率指的是，在测试时被执行的源代码占全部源代码的比例。测试代码覆盖率可以衡量软件的质量，我们甚至还可以用它来识别无效的代码，检验测试用例的有效性。

如果要用单元测试函数测试代码的覆盖率，Go1.2 之后，我们可以用 cover 工具来实现。

cover的基本用法

`go test -cover` 能够用测试函数统计出源代码的覆盖率，我们在项目的 sqldb 库中测试一下代码的覆盖率，输出结果为 83.7%。它反映的是整个 package 中所有代码的覆盖率。


```
» go test -cover
PASS
coverage: 83.7% of statements
ok      github.com/dreamerjackson/crawler/sqldb 0.426s
```

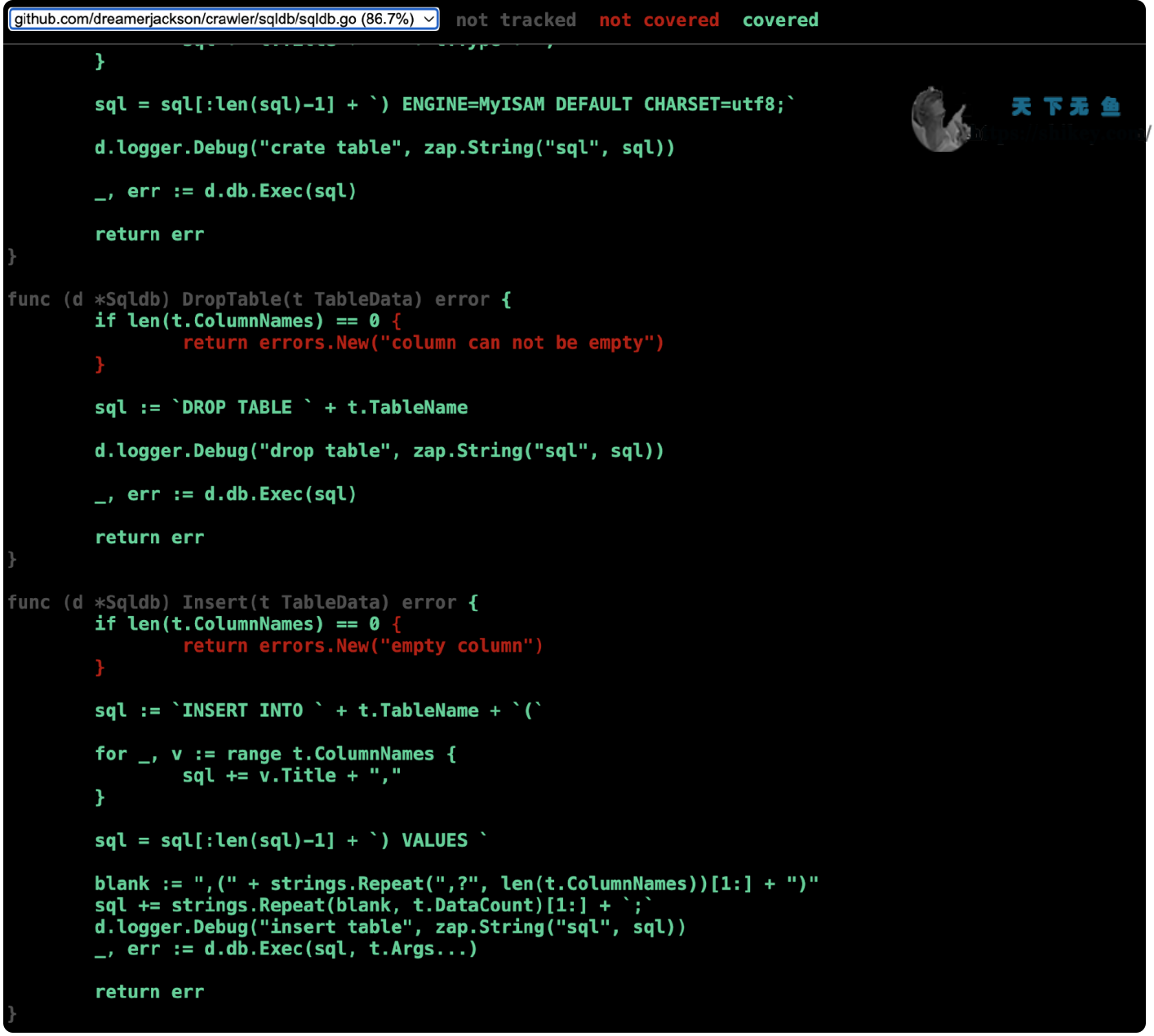
另外，我们还可以收集覆盖率，并进行可视化的展示。具体做法是先将收集到覆盖率的样本文件输出到 coverage.out。

```
go test -coverprofile=coverage.out
```

接着使用 `go tool cover` 可视化分析代码覆盖率信息。

```
go tool cover -html=coverage.out
```

我们还是用 sqldb 库做一下演示，输入上面的命令，浏览器会自动打开并展示下面的信息。其中，绿色的部分就是测试用例走过的部分，而红色的部分是代码没有走过的部分。




```
        }

        sql = sql[:len(sql)-1] + `) ENGINE=MyISAM DEFAULT CHARSET=utf8;`

        d.logger.Debug("crate table", zap.String("sql", sql))

        _, err := d.db.Exec(sql)

        return err
}

func (d *Sqldb) DropTable(t TableData) error {
        if len(t.ColumnNames) == 0 {
                return errors.New("column can not be empty")
        }

        sql := `DROP TABLE ` + t.TableName

        d.logger.Debug("drop table", zap.String("sql", sql))

        _, err := d.db.Exec(sql)

        return err
}

func (d *Sqldb) Insert(t TableData) error {
        if len(t.ColumnNames) == 0 {
                return errors.New("empty column")
        }

        sql := `INSERT INTO ` + t.TableName + `(`

        for _, v := range t.ColumnNames {
                sql += v.Title + ","
        }

        sql = sql[:len(sql)-1] + `) VALUES `

        blank := ",(" + strings.Repeat(",?", len(t.ColumnNames))[1:] + ")"
        sql += strings.Repeat(blank, t.DataCount)[1:] + `;`
        d.logger.Debug("insert table", zap.String("sql", sql))
        _, err := d.db.Exec(sql, t.Args...)

        return err
}
```

根据这些信息，我们可以衡量出当前测试用例和代码的质量。

不过上面这个例子中，我们在测试用例中遗漏掉了 t.ColumnNames 为空时的判断。现在让我们优化一下测试代码，在TestSqldb_InsertTable 测试函数中加入 t.ColumnNames 为空的测试用例。

复制代码

```
{
        name: "no column",
        args: args{TableData{
          TableName:   tableName,
          ColumnNames: nil,
          Args:        []interface{}{"book1", 2},
          DataCount:   1,
        }},
        wantErr: true,
```

```
10      },
```


再次进行覆盖率测试，可以看到，测试代码成功走到了 Insert 函数所有的逻辑分支。

```
func (d *Sqldb) Insert(t TableData) error {
        if len(t.ColumnNames) == 0 {
                return errors.New("empty column")
        }

        sql := `INSERT INTO ` + t.TableName + `(`

        for _, v := range t.ColumnNames {
                sql += v.Title + ","
        }

        sql = sql[:len(sql)-1] + `) VALUES `

        blank := ",(" + strings.Repeat(",?", len(t.ColumnNames))[1:] + ")"
        sql += strings.Repeat(blank, t.DataCount)[1:] + `;`
        d.logger.Debug("insert table", zap.String("sql", sql))
        _, err := d.db.Exec(sql, t.Args...)

        return err
}
```

我们还可以根据覆盖率来了解是否真的需要那些未被覆盖的代码。

测试环境下的代码覆盖率

通常情况下，代码覆盖率是 `go test -cover` 运行了 xxx_test.go 文件的测试函数得到的。但是测试用例通常是手动添加的。 如果我们希望能够接入测试环境中的流量来测试代码覆盖率，能不能实现呢？用上一些技巧是完全可以实现的。

假设我们的服务是一个 Web 服务器，我们可以新建一个 main_test.go 文件，在 main_test.go 中执行 main() 函数，这就好像在实际执行程序一样。

复制代码

```
1  func TestSystem(t *testing.T) {
2      handleSignals()
3
4      endRunning = make(chan bool, 1)
5
6      go func() {
7
8        main()
9
```

```
    }()

    <-endRunning
}
```


接着，我们可以在当前目录执行 `go test  -coverprofile=coverage.out`，或者用 `go test -c  -coverprofile=coverage.out` 生成可执行的测试文件。

然后执行该测试函数，当测试结束时终止程序。退出程序后，就会自动生成 coverprofile 文件了。这时我们可以在测试环境调用 HTTP 请求，并最终统计代码覆盖率。

cover 工具的工作原理

那么，cover 工具测试代码覆盖率的原理是什么呢？

实际上，`go test -cover` 会对代码进行打桩，也就是在编译时在逻辑分支的位置加入统计代码。我们以下面的代码为例。

复制代码

```go
package size

func Size(a int) string {
    switch {
    case a < 0:
        return "negative"
    case a == 0:
        return "zero"
    case a < 10:
        return "small"
    case a < 100:
        return "big"
    case a < 1000:
        return "huge"
    }
    return "enormous"
}
```

我们可以通过下面的命令查看打桩后的代码。

复制代码

```
go tool cover -mode=set  -var=size  xxx.go
```

生成的代码如下所示。可以看到，cover 工具自动在逻辑代码分支的位置加入了统计函数，并提前注册了当前位置的信息。当测试程序进入到该逻辑分支后，该位置会被记录。

复制代码

```
//line xxx.go:1
package tt

func Size(a int) string {size.Count[0] = 1;
        switch {
        case a < 0:size.Count[2] = 1;
                return "negative"
        case a == 0:size.Count[3] = 1;
                return "zero"
        case a < 10:size.Count[4] = 1;
                return "small"
        case a < 100:size.Count[5] = 1;
                return "big"
        case a < 1000:size.Count[6] = 1;
                return "huge"
        }
        size.Count[1] = 1;return "enormous"
}

var size = struct {
        Count     [7]uint32
        Pos       [3 * 7]uint32
        NumStmt   [7]uint16
} {
        Pos: [3 * 7]uint32{
                3, 4, 0x90019, // [0]
                16, 16, 0x130002, // [1]
                5, 6, 0x14000d, // [2]
                7, 8, 0x10000e, // [3]
                9, 10, 0x11000e, // [4]
                11, 12, 0xf000f, // [5]
                13, 14, 0x100010, // [6]
        },
        NumStmt: [7]uint16{
                1, // 0
                1, // 1
                1, // 2
                1, // 3
                1, // 4
                1, // 5
                1, // 6
        },
}
```

而要实现能够被可视化的打桩文件会略微复杂一些。该命令会生成一个 _testmain.go 的中间文件，将文件名和花括号等关键信息提前进行注册，并最终生成 coverprofile 协议文件。具体的执行过程可以通过如下命令查看。


```
go test -n -x  -test.coverprofile=coverage.out
```

只有生成了 coverprofile 协议文件才能够被 cover 工具完成可视化。

coverprofile 协议文件遵循的格式是，第一行为"mode: foo", 其中 foo 是 set 、count 或 atomic 等类型 。协议文件中的其余格式形如: encoding/base64/base64.go:34.44,37.40 3 1，包含了文件名，行号、列号、统计个数等信息。

coverprofile 协议文件需要的信息可以通过 go test -cover 自动生成的代码获得。这里，我们可以参考开源库goc的实现逻辑，goc 库本身是对 go test -cover 命令的封装。借助 goc build --debug 得到的在临时目录生成的文件，我们可以查看到 goc 是如何生成 coverprofile 协议文件的，核心函数位于 loadFileCover 中。

```
func loadFileCover(coverCounters map[string][]uint32, coverBlocks map[string][]testing.CoverBlock, fileName string, counter []uint32, pos
 []uint32, numStmts []uint16) {
        if 3*len(counter) != len(pos) || len(counter) != len(numStmts) {
                panic("coverage: mismatched sizes")
        }
        if coverCounters[fileName] != nil {
                // Already registered.
                return
        }
        coverCounters[fileName] = counter
        block := make([]testing.CoverBlock, len(counter))
        for i := range counter {
                block[i] = testing.CoverBlock{
                        Line0: pos[3*i+0],
                        Col0:  uint16(pos[3*i+2]),
                        Line1: pos[3*i+1],
                        Col1:  uint16(pos[3*i+2] >> 16),
                        Stmts: numStmts[i],
                }
        }
        coverBlocks[fileName] = block
```

我们自己也可以和 goc 一样，通过生成个性化的 coverprofile 协议文件。这样就可以借助 go tool cover 的功能，将该协议转换为 HTML 文件，直观地展示出来。当然，在一些非常高级的场景我们才会考虑生成这样的工具，一般直接用 Cover 工具提供的能力就已经足够了。

最后我们再来看看 cover 工具是如何实现代码打桩的。打桩的原理是借助 AST 语法树对源代码进行扫描。遍历整个文件，在识别到花括号、swith、select 的地方插入一行进行记录，因为这些位置就是程序的逻辑出现分叉的位置。核心函数位于源码 go1.16.14/src/cmd/cover/cover.go 文件的 visit 函数中。

复制代码

```
// go1.16.14/src/cmd/cover/cover.go:202
func (f *File) Visit(node ast.Node) ast.Visitor {
  switch n := node.(type) {
  case *ast.BlockStmt:
    // If it's a switch or select, the body is a list of case clauses; don't ta
    if len(n.List) > 0 {
      switch n.List[0].(type) {
      case *ast.CaseClause: // switch
        for _, n := range n.List {
          clause := n.(*ast.CaseClause)
          f.addCounters(clause.Colon+1, clause.Colon+1, clause.End(), clause.Bo
        }
        return f
      case *ast.CommClause: // select
        for _, n := range n.List {
          clause := n.(*ast.CommClause)
          f.addCounters(clause.Colon+1, clause.Colon+1, clause.End(), clause.Bo
        }
        return f
      }
    }
    f.addCounters(n.Lbrace, n.Lbrace+1, n.Rbrace+1, n.List, true) // +1 to step
  ...
```

模糊测试

介绍完代码覆盖率之后，我们来看一看在代码覆盖率基础上实现的测试技术：模糊测试。

在上一节课程中，我们看到的单元测试都是我们提前放入的测试用例，这些用例依赖于一些提前规划好的功能和后续发现的异常。但是这样有时仍然难以确保充分的测试，因为代码中可能会有一些逻辑分支我们的测试用例没有覆盖到，也有一些极端的场景没有覆盖到。

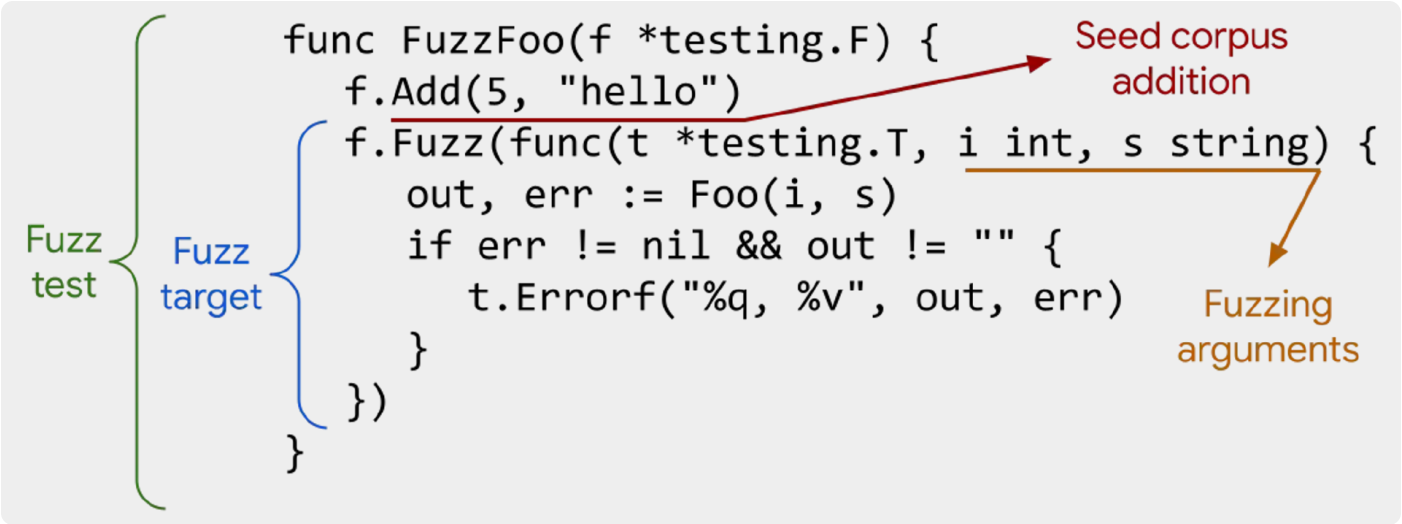
那么有没有一种方式有足够启发性地产生足够多的用例，覆盖更多的逻辑分支，测试各种特别的边界条件呢？这就不得不提到 Go 1.18 之后测试包中支持的模糊测试了。

模糊测试（Fuzzing）是一种可以识别代码中的错误和安全漏洞的优秀测试方法，对于查找整数溢出、输入被截断、无效的编码长度和格式等极端 case 都有极大的帮助。

模糊测试的重点不是取代传统的测试，而是通过随机生成输入值来实现更充分的测试，它经常可以测试到开发者会忽视的极端场景。

和单元测试类似，模糊测试以单词 Fuzz 开头，因此必须以 `f *testing.F` 为参数。而且和单元测试被称为 Unit Test 类似，模糊测试也被称为 Fuzz Test。

模糊测试包含了两个部分，第一个部分是添加初始的测试输入，这些测试输入可以被看做是种子数据或样本数据，也被称为 Seed Corpus。种子数据会在模糊测试中调用 f.Add 添加， 同时也会添加当前目录下 testdata/fuzz/{FuzzTestName} 文件中的数据。



模糊测试将根据样本数据生成新的输入，所以样本数据应该尽可能地反映函数的真实输入情况，以便获得最佳结果。添加样本数据由 f.Add() 完成，它接受以下数据类型：

- string, []byte, rune
- int, int8, int16, int32, int64
- uint, uint8, unit16, uint32, uint64
- float32, float64
- bool

模糊测试的第二个部分是执行目标测试函数（Fuzz Target），它是每个样本数据和模糊测试生成的随机数据需要执行的函数。`Fuzz Target` 函数的第一个参数必须为 testing.T，接着是

输入数据的类型，类型的顺序需要与添加到样本数据中的顺序相同。

下面让我们在项目中实践一下模糊测试，我们以 proxy 包中的 GetProxy 函数为例。

复制代码

```
func (r *roundRobinSwitcher) GetProxy(pr *http.Request) (*url.URL, error) {
  index := atomic.AddUint32(&r.index, 1) - 1
  u := r.proxyURLs[index%uint32(len(r.proxyURLs))]

  return u, nil
}
```

该函数帮助我们实现了 roundRobin 算法，让我们可以均衡地选择代理服务器地址。你能看出这个函数的问题吗？让我们用模糊测试来诊断一下。新建 proxy_test.go 文件，书写模糊测试函数。

复制代码

```
func FuzzGetProxy(f *testing.F) {
  f.Add(uint32(1), uint32(10))
  f.Fuzz(func(t *testing.T, index uint32, urlCounts uint32) {

    r := roundRobinSwitcher{}
    r.index = index
    r.proxyURLs = make([]*url.URL, urlCounts)

    for i := 0; i < int(urlCounts); i++ {
      r.proxyURLs[i] = &url.URL{}
      r.proxyURLs[i].Host = strconv.Itoa(i)
    }

    p, err := r.GetProxy(nil)
    assert.Nil(t, err)

    e := r.proxyURLs[index%urlCounts]

    if !reflect.DeepEqual(p, e) {
      t.Fail()
    }
  })
}
```

在 FuzzGetProxy 中我们定义了两个参数，一个是当前的 Index，另一个参数是 URLs 的数量。f.Add(uint32(1), uint32(10)) 为我们添加了一个样本数据。

这里变量 p 是 GetProxy 函数返回的代理地址，而变量 e 是我们预期应该生成的代理地址。当返回结果与预期结果完全相同时，GetProxy 函数就是正确的。执行模糊测试的命令为 go test -fuzz={FuzzTestName}，和单元测试一样，-fuzz 后跟要测试函数的名字，可以使用正则表达式。

执行模糊测试的结果如下。

```
» go test --fuzz=Fuzz
fuzz: elapsed: 0s, gathering baseline coverage: 0/1 completed
fuzz: elapsed: 0s, gathering baseline coverage: 1/1 completed, now fuzzing with
fuzz: elapsed: 0s, execs: 3 (41/sec), new interesting: 0 (total: 1)
--- FAIL: FuzzGetProxy (0.08s)
    --- FAIL: FuzzGetProxy (0.00s)
        testing.go:1349: panic: runtime error: integer divide by zero
            goroutine 43 [running]:
            runtime/debug.Stack()
                /usr/local/opt/go/libexec/src/runtime/debug/stack.go:24 +0x90
            testing.tRunner.func1()
                /usr/local/opt/go/libexec/src/testing/testing.go:1349 +0x1f2
            panic({0x128f4c0, 0x1468b50})
                /usr/local/opt/go/libexec/src/runtime/panic.go:838 +0x207
            github.com/dreamerjackson/crawler/proxy.(*roundRobinSwitcher).GetPr
                /Users/jackson/career/crawler/proxy/proxy.go:19
            github.com/dreamerjackson/crawler/proxy.FuzzGetProxy.func1(0xc00018
                /Users/jackson/career/crawler/proxy/proxy_test.go:24 +0x265
            reflect.Value.call({0x12891e0?, 0x12deaf8?, 0x13?}, {0x12c7be6, 0x4
                /usr/local/opt/go/libexec/src/reflect/value.go:556 +0x845
            reflect.Value.Call({0x12891e0?, 0x12deaf8?, 0x514?}, {0xc0001903c0,
                /usr/local/opt/go/libexec/src/reflect/value.go:339 +0xbf
            testing.(*F).Fuzz.func1.1(0x0?)
                /usr/local/opt/go/libexec/src/testing/fuzz.go:337 +0x231
            testing.tRunner(0xc0001809c0, 0xc0001a8900)
                /usr/local/opt/go/libexec/src/testing/testing.go:1439 +0x102
            created by testing.(*F).Fuzz.func1
                /usr/local/opt/go/libexec/src/testing/fuzz.go:324 +0x5b8


    Failing input written to testdata/fuzz/FuzzGetProxy/70e685ddfc993d5486f7eb6
    To re-run:
    go test -run=FuzzGetProxy/70e685ddfc993d5486f7eb6700d3c1bda950e873fd05c02c7
FAIL
exit status 1
FAIL    github.com/dreamerjackson/crawler/proxy 0.118s
```

可以看到，当前的程序直接 panic 了，而且打印出了堆栈信息。

这里第一行 gathering baseline coverage 表示，模糊测试以实现更高的代码覆盖率为目标来生成目标数据。panic 输出的结果为：runtime error: integer divide by zero，表明我们遇到了除 0 错误。

在输出的下方我们还可以看到，测试失败后，失败的用例会输出到文件 testdata/fuzz/FuzzGetProxy/70e685xxx 中，查看这个文件，可以看到失败用例的参数为 1，0。

复制代码

```
» cat testdata/fuzz/FuzzGetProxy/70e685ddfc993d5486f7eb6700d3c1bda950e873fd05c0
go test fuzz v1
uint32(1)
uint32(0)
```

我们继续查看堆栈信息和失败用例，会发现在 GetProxy 中，当 `len(r.proxyURLs)` 为 0 时，取余操作会触发除零错误，这样我们就可以有方向地优化我们的代码了。

复制代码

```
func (r *roundRobinSwitcher) GetProxy(pr *http.Request) (*url.URL, error) {

  if len(r.proxyURLs) == 0 {
    return nil, errors.New("empty proxy urls")
  }

  index := atomic.AddUint32(&r.index, 1) - 1
  u := r.proxyURLs[index%uint32(len(r.proxyURLs))]

  return u, nil
}
```

不过很快我们又遇到了新的问题。对于 GetProxy 返回报错的用例，如果我们的预期是“应该报错”，那么就是符合预期的，这些用例本不应该让整个测试失败。要实现这一点，我们可以使用 t.Skip 函数跳过符合预期的特殊用例。

```go
func FuzzGetProxy(f *testing.F) {
  f.Add(uint32(1), uint32(10))
  f.Fuzz(func(t *testing.T, index uint32, urlCounts uint32) {

    r := roundRobinSwitcher{}
    r.index = index
    r.proxyURLs = make([]*url.URL, urlCounts)

    for i := 0; i < int(urlCounts); i++ {
      r.proxyURLs[i] = &url.URL{}
      r.proxyURLs[i].Host = strconv.Itoa(i)
    }

    p, err := r.GetProxy(nil)
    if err != nil && strings.Contains(err.Error(), "empty proxy urls") {
      t.Skip()
    }

    assert.Nil(t, err)

    e := r.proxyURLs[index%urlCounts]

    if !reflect.DeepEqual(p, e) {
      t.Fail()
    }
  })
}
```


模糊测试默认会一直执行。如果我们不希望始终执行模糊测试，也可以设置 fuzztime 参数指定运行模糊测试的时间。

```go
go test --fuzz=Fuzz -fuzztime=10s
```


学完上面这些内容，你应该可以搞定绝大多数的模糊测试场景了。另外，如果你想进一步了解模糊测试的内部机制，可以查看这篇文章。

总结

这节课，我们重点介绍了代码覆盖率和它的实现原理。

代码覆盖率是非常有用的指标，它可以帮助我们诊断代码和测试用例的质量，查找无用代码等。

此外，我们还看到了基于代码覆盖率而实现的模糊测试。模糊测试是 Go1.18 引入的工具，它本身就为 Go 标准库发现了二百多个代码 Bug。模糊测试通过随机生成一些测试用例，可以为我们覆盖更多的逻辑分支，及时发现程序的 Bug，保证程序的安全性。

课后题

学完这节课，给你留一道思考题。

模糊测试找到了 Go 标准库上百个代码 Bug，这些代码都是非常优秀的工程师书写的。这给了你什么启发？请你通过模糊测试的原理，谈一谈你的看法。我们下节课见！

上一篇 36丨测试的艺术：依赖注入、表格测试与压力测试

下一篇 38丨高级调试：怎样利用Delve调试复杂的程序问题？

精选留言 (1)

徐曙辉

2023-01-11 来自湖南

启发：不测没bug，测测有bug，少测少bug，多测多bug