Go的测试覆盖率

2017-05-24 | 技术积累 |

测试覆盖率是一个术语，用于统计通过运行程序包的测试多少代码得到执行。 如果执行测试套件导致80%的语句得到了运行，则测试覆盖率为80%。

计算测试覆盖率的通常方法是埋点二进制可执行文件。 例如，GNU gcov 在二进制文件中设置执行分支断点。 当每个分支执行时，断点被清除，并且分支的目标语句被标记为“被覆盖”。

这种方法是成功和广泛使用的。 Go的早期测试覆盖工具甚至以相同的方式工作。但它有问题。 由于分析二进制文件的执行是很困难的，所以很难实现。 它还需要将执行跟踪绑定回源代码的可靠方法，这也可能是困难的。 那里的问题包括不正确的调试信息和诸如内联功能的问题,使分析变得复杂。 最重要的是，这种方法非常不具有可移植性。 对于每个机器架构需要重新编写，在某种程度上,可能对于每个操作系统都需要重新编写，因为从系统到系统的调试支持差异很大。

Go 1.2 的发布引入了一个 *test coverage* 的新工具， 它采用了一种不寻常的方式来生成覆盖率统计数据，这种方法建立在Godoc的技术的基础上。

1 Go的测试覆盖率

对于Go的新测试覆盖工具，采取了一种避免动态调试的不同方法。 想法很简单：在编译之前重写包的源代码，以埋点，编译和运行修改的源，并转储统计信息。 重写很容易编排，因为 *go的工具链* 控制从源到测试到执行的整个流程。

示例代码如下：

```
1  func Size(a int) string {
2    switch {
3    case a < 0:
4      return "negative"
5    case a == 0:
6      return "zero"
```

```
    case a < 10:
      return "small"
    case a < 100:
      return "big"
    case a < 1000:
      return "huge"
    }
    return "enormous"
}
```

测试代码如下：

```
type Test struct {
    in  int
    out string
}

var tests = []Test{
    {-1, "negative"},
    {5, "small"},
}

func TestSize(t *testing.T) {
    for i, test := range tests {
        size := Size(test.in)
        if size != test.out {
            t.Errorf("#%d: Size(%d)=%s; want %s", i, test.in, size, test.out
        }
    }
}
```

执行代码覆盖率测试如下：

```
cd ../src/cover/size/
go test -cover
cd -
```

```
PASS
coverage: 42.9% of statements
ok      _/home/parallels/program/org/github-pages/source/src/cover/size 0.001s
/home/parallels/program/org/github-pages/source/_posts
```

启用测试覆盖后，/go test/ 运行 *cover* 工具，在编译之前重写源代码。 以下是重写后的 *Size* 函数：

```
func Size(a int) string {
    GoCover.Count[0] = 1
```

```
    switch {
    case a < 0:
        GoCover.Count[2] = 1
        return "negative"
    case a == 0:
        GoCover.Count[3] = 1
        return "zero"
    case a < 10:
        GoCover.Count[4] = 1
        return "small"
    case a < 100:
        GoCover.Count[5] = 1
        return "big"
    case a < 1000:
        GoCover.Count[6] = 1
        return "huge"
    }
    GoCover.Count[1] = 1
    return "enormous"
}
```

上面示例的每个可执行部分用赋值语句进行注解，赋值语句用于在运行时做统计。 计数器与 *cover* 工具生成的第二个只读数据结构记录的语句的原始源位置相关联。 测试运行完成后，收集计数器，通过查看设置的数量的来计算百分比。

虽然分配注解看起来可能很昂贵，但是它被编译为单个“移动”指令。 因此，其运行时开销不大，运行典型（或更实际）测试时只增加约3%开销。 这使得把测试覆盖率作为标准开发流程的一部分是合情合理的。

2 查看结果

上面的例子的测试覆盖率很差。 为了探索具体为什么，需要 *go test* 写一个 *coverage profile* ， 这是一个保存收集的统计信息的文件，以便能详细地研究覆盖的细节。 这很容易做：使用 *-coverprofile* 标志来指定输出的文件：

```
cd ../src/cover/size/
go test -coverprofile=size_coverage.out
```

注：*-coverprofile* 标志自动设置 *-cover* 来启用覆盖率分析。

测试与以前一样运行，但结果保存在文件中。 要研究它们，需要运行 *test coverage tool* 。 一开始，可以要求 *覆盖率* 按函数分解，虽然在当前情况下没有太多意义，因为只有一个函数：

```
cd ../src/cover/size/
```

```
2  go tool cover -func=size_coverage.out
```

查看的更有趣的方式是获取 *覆盖率信息注释的源代码* 的HTML展示。 该显示由 *-html* 标志调用：

```
1  cd ../src/cover/size/
2  go tool cover -html=size_coverage.out
```

运行此命令时，浏览器将弹出窗口，已覆盖（绿色），未覆盖（红色）和 未埋点（灰色）。下面是一个屏幕截图：

有了这个信息页，问题变得很明显：上面忽略了几个 *case* 的测试！ 可以准确地看出具体是哪一个，这样可以轻松地提高的测试覆盖率。

3 热力图

源代码级方式来测试覆盖率的一大优点在于，可以很容易用不同的方式对代码进行埋点处理。 例如，不仅可以检测是否已执行了一个语句，而且还可以查询执行了多少次。

go test 命令接受 *-covermode* 标志将覆盖模式设置为三种设置之一：

- set: 每个语句是否执行?
- count: 每个语句执行了几次?
- atomic: 类似于 *count*, 但表示的是并行程序中的精确计数

set 是默认设置，上面示例已经看到了。 只有运行并行算法需要精确的计数时，才需要进行 *atomic* 设置。 它使用来自 sync/atomic 包的原子操作，这可能会相当昂贵。 然而，对于大多数情况， *count* 模式工作正常，并且像默认设置模式一样非常快。

下面来试试一个标准包， *fmt* 格式化包语句执行的计数。 进行测试并写出 *coverage profile* ，以便能够很好地进行信息的呈现。

```
go test -covermode=count -coverprofile=../src/cover/count.out fmt
```

这比以前的例子好的测试覆盖率。（覆盖率不受覆盖模式的影响）可以显示函数细节：

```
go tool cover -func=../src/cover/count.out
```

HTML输出产生了巨大的回报：

```
go tool cover -html=../src/cover/count.out
```

pad 函数如下所示：

注意绿色的强度是如何变化。 最明亮的绿色的代表较高的执行数; 较少灰暗的绿色代表较低的执行数。 甚至可以将鼠标悬停在语句上，以便在弹出的 *tool tip* 中提示实际计数。 *test coverage* 产生了关于函数执行的大量信息，在分析中很有用的信息。

4 基础块

你可能已经注意到，上一个示例中/ 有关于闭合大括号中间的行的计数/ 不是你所期望的那样。 这是因为一直以来 *test coverage* 都不是一个不精确的科学。

这里发生的很值得解释。 我们希望覆盖注解由程序中的分支划分，当二进制文件在传统方法中被调用时，它们是分开的。 不过，通过重写源代码很难做到这一点，因为分支没有明确展示在源代码中。

覆盖注解的作用是是埋点，通常由大括号来限定。 一般来说，使之工作正常是非常困难的。 所使用的算法的处理结果是闭合括号看起来像属于它配对的块，而开放大括号看起来像属于块之外。 一个更有趣的结果出现在如下的一个表达式中：

```
f() && g()
```

没有试图单独调用对f和g的调用，无论事实如何，它们总是看起来像是运行相同的次数。

公平来说，即使gcov在这里也有麻烦。 该工具使机制正确，但呈现是基于行的，因此可能会错过一些细微差别。

5 总结

这是关于 *Go 1.2 test coverage* 故事。 具有有趣实现的新工具不仅可以实现测试覆盖率的统计，而且易于解释，甚至可以提取 *profile* 信息。

测试是软件开发和的重要组成部分，/test coverage/ 为测试策略添加一个简单的标准。 走向前， *test* 和 *cover* 。

Last Updated 2017-05-25 Thu 16:59.
Render by hexo-renderer-org with Emacs 25.3.2 (Org mode 8.2.10)

golang # 测试覆盖率 # test

数组，切片：‘append’的机制 在Org-mode中执行Go代码

相关文章

- 在Org-mode中执行Go代码
- Strings, bytes, runes and characters in Go
- go-dlv试用
- Emacs支持gomodifytags
- Go语言不完全工具列表