https://github.com/smallnest/go_test_workshop

Go Test

从入门到躺平

O'RLY?

鸟窝

第一章 前言

单元测试通常是由软件开发人员编写和运行的自动测试，以确保应用程序的某个部分（称为“单元”）符合其设计并按预期运行。在过程式编程中，一个单元可以是一个完整的模块，但它更常见的是一个单独的函数或过程。在面向对象的编程中，单元通常是整个接口，例如类或单个方法。通过首先为最小的可测试单元编写测试，然后为它们之间的复合行为编写测试，可以为复杂的应用程序建立全面的测试。

在开发过程中，软件开发人员可能会将标准或已知良好的结果编码到测试中，以验证单元的正确性。在测试用例执行期间，框架会记录不符合任何条件的测试，并在摘要中报告这些测试。为此，最常用的方法是测试 - 函数 - 期望值。

好处

单元测试的目标是隔离程序的每个部分，并表明各个部分是正确的。单元测试提供了一个严格的书面契约，你实现的代码逻辑单元必须满足这个契约。因此，它提供了几个好处。

- 早期发现问题： 单元测试在开发周期的早期发现问题。这包括程序员实现中的错误和单元规范的缺陷或缺失部分。编写一组完整的测试的过程迫使作者仔细考虑输入，输出和错误条件，从而更清晰地定义单元的所需行为。在编码开始之前或首次编写代码时查找 Bug 的成本远低于以后检测、识别和更正 Bug 的成本。已发布代码中的错误也可能给软件的最终用户带来代价高昂的问题。如果写得不好，代码可能不可能或难以进行单元测试，因此单元测试可以迫使开发人员以更好的方式构建函数和对象。

- 适合敏捷编程：测试驱动开发（TDD）经常用于极限编程和Scrum，它要求单元测试是在编写代码本身之前创建的。测试通过后，该代码将被视为已完成。如果单元测试失败，则将其视为已更改代码或测试本身中的 Bug。然后，单元测试允许轻松跟踪故障或故障的位置。由于单元测试在将代码交给测试人员或客户端之前会向开发团队发出问题警报，因此在开发过程的早期会发现潜在的问题。

- 适合回归测试: 单元测试允许程序员在以后重构代码或升级系统库，并确保模块仍然正常工作（例如，在回归测试中）。该过程是为所有函数和方法编写测试用例，以便每当更改导致错误时，都可以快速识别它。单元测试检测可能破坏设计合同的更改。

- 确保单元稳定：单元测试可以减少单元本身的不确定性，并且可以用于自下而上的测试样式方法。通过首先测试程序的各个部分，然后测试其各部分的总和，集成测试变得更加容易。

- 单元测试就是"文档"：单元测试提供了一种系统的活文档。希望了解单元提供哪些功能以及如何使用它的开发人员可以查看单元测试，以基本了解单元的接口 （API）。

局限性和缺点

- 测试不会捕获程序中的每个错误，因为它无法覆盖程序中的每个执行路径。
- 单元测试的详细层次结构不等于集成测试。与外围单元的集成应包含在集成测试中，但不应包含在单元测试中。
- 软件测试是一个组合问题。例如，每个布尔决策语句至少需要两个检验：一个结果为“true”，另一个结果为“false”。因此，对于编写的每行代码，程序员通常需要3到5行测试代码。这显然需要时间，收益可能和投入不成正比。
- 有些场景不容易测试。 例如那些非确定性或涉及多个线程的问题。此外，单元测试的代码与它正在测试的代码一样可能存在错误。
- 流程和规范也很重要，规范的制度确保定期审查测试用例失败并立即解决。如果这样的过程没有被实现并根深蒂固地融入到团队的工作流程中，应用程序将与单元测试套件不同步，增加误报并降低测试套件的有效性。

单元测试最初兴起于敏捷社区。1997年，设计模式四巨头之一Erich Gamma(设计模式、Junit、Eclipse、VSCode)和极限编程发明人Kent Beck(极限编程、测试驱动开发、JUnit)共同开发了JUnit，而JUnit框架在此之后又引领了xUnit家族的发展，深刻的影响着单元测试在各种编程语言中的普及。当前，单元测试也成了敏捷开发流行以来的现代软件开发中必不可少的工具之一。同时，越来越多的互联网行业推崇自动化测试的概念，作为自动化测试的重要组成部分，单元测试是一种经济合理的回归测试手段，在当前敏捷开发的迭代中非常流行和需要。

二十年前，单元测试的方法和JUnit引入到国内的时候，大家都是求知若渴，将其奉为圭臬。不幸的是，在当前一个浮躁的追求短期利益的互联网时代，很多公司甚至头部大厂都已经不把单元测试作为提升软件质量的一种手段，甚至几乎不写单元测试了。“能逮着老鼠的就是好猫”成了大家衡量成功的标准，糙快猛的把原型堆出来，原型试错，快速拿奖走人大行其道。万幸的是，在一些公司中，还是有一些工程师，依然坚持着传统的信念，不会把“时间紧需要快速出活”作为理由降低代码质量，还是把单元测试、代码规范等坚持下来。

即使现在有些公司，采用tcpcopy、goreplay复制线上流量对测试环境的系统进行测试，或者像滴滴的写轮眼可以在一定程度上解决手工编写测试的麻烦以及微服务依赖的问题，但是它们更多的是复制线上流量对集成系统所做的正面测试，对于单元测试的全面覆盖、尤其是测试路径的全面性包括负面测试还需要单元测试去实现。当然还有一些通过历史数据进行单元测试的一些方法也是类似。

Go语言不像Java等程序提供JUint、TestNG这样的单元测试框架，而是把单元测试的思想融入到Go编程语言之中，彻彻底底的将单元测试作为代码规范的第一要务，这是很了不起的。就是Go团队本身，在开发Go标准库时，都是有各种单元测试的，如果你想提交对代码逻辑的改动，都需要跑过既有的单元测试，对于新的测试也必须提供足够的单元测试。对于生成对性能提升的代码，还需要有充分的benchmark代码的支撑, 比如下面字节跳动语言团队提供的Go sort算法的提升，提供各种benchmark的性能对比，大部分场景下性能都有提升(delta负值)。

第二章 入门

Go提供了test工具，可以对形式如`func TestXxx(*testing.T)`格式的函数进行自动化单元测试，这样的函数我们称之为单元测试函数。

函数名必须以`Test`开头，并且`Xxx`必须首字母大写，比如`Add`、`Do`等。首字母小写如`add`和`do`不被认为是单元测试函数，`go test`不会自动执行。

一般`Xxx`我们常常使用要单元测试的函数名，或者是要单元测试的结构体(`struct`)名称。但是也不是绝对的，只是大家习惯于这样写。

一般单元测试的名称我们还会命名为`TestXxx_Yyy`等形式，加一些后缀，用来区分一些子测试或者不同场景的测试，这种命名没有强制规定，但G官方标准库的单元测试名称更倾向使用多个单词的大驼峰式命名法，如`TestAbcXyzWithIjk`,而不是以下划线分隔的方式。

所有的单元测试函数都放在以`_test.go`结尾的文件中，此文件放在和要测试的文件相同的package中。

这个单元测试的文件的package名称可以和要测试的package名称相同，也可以以此package加`_test`作为包名。比如要测试`package abc`，则单元测试文件中的包名可以是`package abc_test`，这样的好处是单元测试函数只能访问要测试的公开的函数、结构和变量，更贴近使用这个包的场景。Go标准库中这两种方法都有使用。

比如在包s1下定义了一个`greet`函数(`hello.go`):

```
package s1

func greet(s string) string {
    return "hello " + s
}
```

然后我们可以在相同的文件夹下创建一个`hello_test.go`的文件,然后实现一个单元测试的函数(注意函数名称中的G需要大写，否则不会被认为是单元测试的函数):

```
package s1

import "testing"

func TestGreet(t *testing.T) {
    s := greet("test")

    if s != "hello test" {
        t.Errorf("want 'hello test' but got %s", s)
    }
}
```

在此目录下执行`go test -v .`,就可以看到此package下所有的单元测试就会被执行了。

事实上， 当你执行`go test`执行单元测试时，Go会把单元测试编译成一个test binary,也就是用来单元测试的可执行程序。在第四章我们再详细介绍它。

表格驱动型单元测试

Go官方还推崇一种叫做表格驱动型单元(TableDrivenTests)。

编写好的测试并非易事，但在许多情况下，表格驱动的测试可以覆盖很多领域：每个表条目都是一个完整的测试用例，其中包含输入和预期结果，有时还包含附加信息，例如测试名称，以使测试输出易于阅读。如果您发现自己在编写测试时使用复制和粘贴，请考虑一下重构到表格驱动的测试中或将复制的代码提取到帮助程序函数中是否是更好的选择。

给定一个测试用例表，实际测试只是循环访问所有表条目，并为每个条目执行必要的测试。测试代码编写一次，并在所有表条目上摊销，因此编写具有良好错误消息的仔细测试是有意义的。

表格驱动的测试不是工具，包或其他任何东西，它只是编写更干净测试的一种方式和视角。

表格驱动型单元测试并不是一种强制性方式，不过Go标准库中部分单元测试都是采用这种方式组织的。

cweill/gotests是一个可以为你的函数自动生成表格驱动型的单元测试的工具。事实上常见的Go IDE工具如vs code、Goland、Vim都集成了，通过鼠标右键菜单就可以自动生成表格驱动的单元测试框架。

比如官方标准库中的`fmt`单元测试的例子：

```
var flagtests = []struct {
    in  string
    out string
}{
    {"%a", "[%a]"},
    {"%-a", "[%-a]"},
    {"%+a", "[%+a]"},
    {"%#a", "[%#a]"},
    {"% a", "[% a]"},
    {"%0a", "[%0a]"},
    {"%1.2a", "[%1.2a]"},
    {"%-1.2a", "[%-1.2a]"},
    {"%+1.2a", "[%+1.2a]"},
    {"%-+1.2a", "[%+-1.2a]"},
    {"%-+1.2abc", "[%+-1.2a]bc"},
    {"%-1.2abc", "[%-1.2a]bc"},
}
func TestFlagParser(t *testing.T) {
    var flagprinter flagPrinter
    for _, tt := range flagtests {
        t.Run(tt.in, func(t *testing.T) {
            s := Sprintf(tt.in, &flagprinter)
            if s != tt.out {
                t.Errorf("got %q, want %q", s, tt.out)
            }
        })
    }
}
```

它准备了12个条目，每个条目都有输入和期望值。单元测试执行每一个条目，并和期望值进行比较，不符合的话就是一个错误的测试。

你可以动手写一个例子，比如几个加减乘除的算术函数：

如果你使用IDE,一般常见的IDE都会有自动生成单元测试的能力。选择一个函数或者多个函数生成，并在此基础上修改：

我使用的是vs code，启用了code lens,所以单元测试函数上面有按钮`run test|debug test`，直接点击`run test`就可以执行这个单元测试了。

golden file

Go标准库还使用了另外一种方式进行复杂的测试在一些复杂的测试中，你可以把期望的输出写入到一个测试文件中。当测试的时候的时候，单测的输出和这些文件进行测试。

Go中有一个特殊的文件夹叫**testdata**,你可以把golden file放入到这个文件夹中，这个文件夹不会用来作为编译Go代码。

比如go/gofmt_test.go文件中，读取testdata下所有的input文件和响应的golden文件做比较：

```
// TestRewrite processes testdata/*.input files and compares them to the
// corresponding testdata/*.golden files. The gofmt flags used to process
// a file must be provided via a comment of the form
//
//    //gofmt flags
//
// in the processed file within the first 20 lines, if any.
func TestRewrite(t *testing.T) {
    // determine input files
    match, err := filepath.Glob("testdata/*.input")
    if err != nil {
        t.Fatal(err)
    }

    // add larger examples
    match = append(match, "gofmt.go", "gofmt_test.go")

    for _, in := range match {
        name := filepath.Base(in)
        t.Run(name, func(t *testing.T) {
            out := in // for files where input and output are identical
            if strings.HasSuffix(in, ".input") {
                out = in[:len(in)-len(".input")] + ".golden"
            }
            runTest(t, in, out)
            if in != out && !t.Failed() {
                // Check idempotence.
                runTest(t, out, out)
            }
        })
    }
}
```

关键是如何创建这些golden file。你可以手工创建这些golden files， 但是你也可以通过单元测试初始化golden files，或者在某次更新golden files，你可以通过参数传递给单元测试要不要更新：

```
update := flag.Bool("update", false, "update golden files.")
go test -update
```

比如go/gofmt_test.go

```
var update = flag.Bool("update", false, "update .golden files")
func runTest(t *testing.T, in, out string) {
    ......
    expected, err := os.ReadFile(out)
    if err != nil {
        t.Error(err)
        return
    }

    if got := buf.Bytes(); !bytes.Equal(got, expected) {
        if *update {
            if in != out {
                if err := os.WriteFile(out, got, 0666); err != nil {
                    t.Error(err)
                }
                return
            }
            // in == out: don't accidentally destroy input
            t.Errorf("WARNING: -update did not rewrite input file %s", in)
        }

        t.Errorf("(gofmt %s) != %s (see %s.gofmt)\n%s", in, out, in,
            diff.Diff("expected", expected, "got", got))
        if err := os.WriteFile(in+".gofmt", got, 0666); err != nil {
            t.Error(err)
        }
    }
}
```

更友好的测试结果显示

当一个项目下的单元测试非常多的时候，测试结果会刷屏，成功的测试和失败的测试混合在结果中，不容易区分，所以有些开发者提供了定制的工具，对测试结果进行染色，让成功的结果和失败的结果更容易区分出来。如果你以前使用过XUnit风格的测试工具，比如JUint,它会使用红绿颜色做区分。

rakyll是一位知名的Gopher,她提供了一个工具rakyll/gotest,可以彩色化单元测试的结果。

你可以执行`go install github.com/rakyll/gotest@latest`安装这个工具。先前我们进行单元测试的时候敲入`go test ...`,使用这个工具的时候去掉空格就好了`gotest ...`。

第三章 testing.T

现在，你已经基本了解了Go单元测试的函数写法，以及是如何执行的。而且你也知道一个单元测试的函数的函数签名如下所示:

```
func TestXxx(t *testing.T) {
    ...
}
```

注意，这个函数是没有返回类型的，而且必须有且只有一个`testing.T`的参数。当单元测试执行的时候，它会传入`*testing.T`的参数，你可以在函数体中使用这个参数，标记测试是否成功或者失败，输出一些日志，或者执行子测试。

`*testing.T`和用来做benchmark的`*testing.B`以及做模糊测试的`*testing.F`一样，都实现了`testing.TB`接口。这个接口的名字也很有趣，应该先前是`T`、`B`共同的接口，所以叫`TB`,不过后来Go 1.18加入了Fuzzing测试，所以接口名是不是应该是`TBF`了？不过这不重要，一般我们很少直接使用`*testing.TB`。

这一章我们只介绍`*testing.T`,也就顺便介绍了`TB`的方法。`T`、`B`、`F`特有的方法我们遇到的时候专门介绍。

我将`*testing.T`的方法按照功能分成几类进行介绍。

标记测试是否失败 （Fail/FailNow/Failed/Fatal/Fatalf）

- **Fail**: 标记当前测试失败，但是后续的代码还是继续执行，不会中断
- **FailNow**: 标记当前测试失败，并且调用`runtime.Goexit`停止本goroutine的执行。

  因为只是调用的`runtime.Goexit`,而不是`os.Exit`，所以它只是停止了本测试，后续的测试还是会被执行。
- **Fatal**: 先输出一个日志(Log),再标记当前测试失败停止执行。等价于`Log` + `FailNow`。
- **Fatalf**: 等价于`Logf` + `FailNow`。
- **Failed**: 返回当前函数是否已经被标记为failed。

例子：

```
package s2

import (
    "testing"
    "time"
)

func TestFail(t *testing.T) {
    t.Fail()
    t.Log("after Fail")
    t.FailNow()
    t.Log("after FailNow")
}

func TestFatal(t *testing.T) {
    t.Fatal("fataled")
    t.Log("after Fatal")
}

func TestFatalf(t *testing.T) {
    t.Fatalf("there is: %v", "Fatalf")
    t.Log("after Fatalf")
}

func TestFailNowInAnotherGoroutine(t *testing.T) {
    go func() {
        t.FailNow()
        t.Log("after FailNow in another goroutine")
    }()

    time.Sleep(time.Second)
    t.Log("after one second")
}
```

输出日志 (Log/Logf/Error/Errorf)

这几个方法用来输出日志信息。

- **Log**: 类似 `Println`,输出到error log中。对于单元测试，只有测试失败或者使用 `-test.v` 参数时，才会真正输出日志。但是对于benchmark,总是会输出。
- **Logf**: 类似 `Printf`,可以输出格式化的日志。

- **Error**: 等价于 `Log` + `Fail` 。
- **Errorf**: 等价于 `Logf` + `Fail` 。

例子：

```
package s2

import (
    "testing"
    "time"
)

func TestLog(t *testing.T) {
    t.Log("it is a log")
    t.Logf("it is a log at %v", time.Now().Format(time.RFC1123))

    t.Error("it is an error")
    t.Errorf("it is an error at %v", time.Now().Format(time.RFC1123))
}
```

跳过（Skip/SkipNow/Skipped/equivalent）

- **SkipNow**: 跳过本测试，并调用 `runtime.Goexit` 停止执行本测试。
- **Skip**：等价于 `Log` + `SkipNow` 。
- **Skipf**: 等价于 `Logf` + `SkipNow` 。
- **Skipped**: 返回当前测试是否被跳过。

```go
package s2

import (
    "runtime"
    "testing"
)

func TestSkip(t *testing.T) {
    if runtime.GOOS == "darwin" {
        t.Skip("skip MacOS")
    }

    if testing.Short() {
        t.Skip("skip because of short")
    }

    t.Log("there a non-skipped log")
}
```

那么什么时候使用Skip方法呢?

比如你在本机编写单元测试函数,这些函数依赖一些基础的服务，或者只能在Linux环境下运维而你的机器是Mac,这个时候你可能不想执行这些单元测试，而是在依赖可用或者Linux环境下才运行。你可以编写好这些单元测试，指定在某种环境下直接跳过，而不是报错。

还比如有些单元测试需要耗费比较长的时间。当传入`-short`参数时，我们就跳过这些耗时长的测试。

Helper

有时候，我们在编写单元测试代码时，可能会使用一些辅助的函数，这些函数我们称之为 helper 函数。当输出错误信息的时候，会显示辅助函数的信息，这些信息实际是不必要的，所以我们可以在这些函数的开始位置调用`Helper`方法，将此函数标记为辅助函数，这样日志中就不会显示这些辅助函数的信息了。

比如下面这个来自于官方的例子：

```go
package s2

import "testing"

func notHelper(t *testing.T, msg string) {
    t.Error(msg)
}

func helper(t *testing.T, msg string) {
    t.Helper()
    t.Error(msg)
}

func notHelperCallingHelper(t *testing.T, msg string) {
    helper(t, msg)
}

func helperCallingHelper(t *testing.T, msg string) {
    t.Helper()
    helper(t, msg)
}

func TestHelper(t *testing.T) {
    notHelper(t, "0")
    helper(t, "1")
    notHelperCallingHelper(t, "2")
    helperCallingHelper(t, "3")

    fn := func(msg string) {
        t.Helper()
        t.Error(msg)
    }
    fn("4")

    t.Helper()
    t.Error("5")

    t.Run("sub", func(t *testing.T) {
        helper(t, "6")
        notHelperCallingHelper(t, "7")
        t.Helper()
        t.Error("8")
    })
}
```

Parallel

当一个单元测试的函数标记为可以并发执行的时候，它可以并且只能和其它标记为Parallel的函数并发执行。

当使用`-test.count`或`-test.cpu`参数时，同一个单元测试的多个实例之间不会并发执行。

```
package s2

import (
    "net/http"
    "testing"
)

func TestParallel(t *testing.T) {
    var urls = map[string]string{"baidu": "http://baidu.com", "bing":
"http://bing.com", "google": "http://google.com"}
    for k, v := range urls {
        v := v
        ok := t.Run(k, func(t *testing.T) {
            t.Parallel()

            t.Logf("start to get %s", v)
            resp, err := http.Get(v)
            if err != nil {
                t.Fatalf("failed to get %s: %v", v, err)
            }

            resp.Body.Close()
        })

        t.Logf("run: %t", ok)
    }
}
```

Cleanup

有时候我们写单元测试的时候，需要做一些准备的动作，比如建立与数据库的连接，打开一个文件、创建一个临时文件等等，执行案单元测试我们还想执行一些清理的动作，比如关闭与数据库的连接，关闭文件，删除临时文件等等，这个时候我们就可以使用Cleanup方法了：

```go
package s2

import (
    "io/ioutil"
    "os"
    "path/filepath"
    "testing"
)

func TestWrite(t *testing.T) {
    tempDir, err := ioutil.TempDir(".", "temp")
    if err != nil {
        t.Errorf("create tempDir: %v", err)
    }
    t.Cleanup(func() { os.RemoveAll(tempDir) })

    err = ioutil.WriteFile(filepath.Join(tempDir, "test.log"), []byte("hello
test"), 0644)
    if err != nil {
        t.Errorf("want writing success but got %v", err)
    }
}
```

TestMain

Go还提供了TestMain的功能。 TestMain并不是用来测试main函数，而是对同一个package下的所有测试有统一的控制。

比如所有的测试都数据数据库的连接，那么可以在`m.Run`之前把数据库连接准备好，在执行完测试之后把数据库关闭。

你可以看到它和`Cleanup`不太一样，`Cleanup`针对的是一个单元测试函数，而`TestMain`针对的是同一个package所有的测试。

```
import (
    "log"
    "os"
    "testing"
)

func TestMain(m *testing.M) {
    log.Println("do stuff BEFORE the tests!")
    exitVal := m.Run()
    log.Println("do stuff AFTER the tests!")

    os.Exit(exitVal)
}
```

第四章 go test 命令

虽然前面我们已经介绍了使用`go test`命令自动执行单元测试，但是go test命令其实还是比较复杂的，包括很多的参数，让我们一一介绍它们。

你可以通过下面三条命令学习go test相关的命令和知识。

go help test

这条命令主要介绍了go test基础的知识。比如go test命令格式如下:

```
go test [build/test flags] [packages] [build/test flags & test binary flags]
```

可以看到我们可以设置`build/test flags`参数，可以放在包的前面和后面。

既支持test的参数，还支持build参数，比如`-N -l`禁止优化和内联。

可以看到`packages`是复数，也就意味着我们可以测试一组package,传入package列表即可，比如`go test bytes strings`。

执行`go test`会会搜寻指定的package下以`_test.go`结尾的文件，找到其中的单元测试函数并执行，并且会显示单元测试结果的汇总信息:

```
ok   archive/tar   0.011s
FAIL archive/zip   0.022s
ok   compress/gzip 0.033s
...
```

如果有失败的测试，会把错误信息细节显示出来。

Go会把每一个package下以`_test.go`编译成独立的包，然后链接成一个test 二进制文件运行。

在构建test 二进制文件时，会调用`go vet`进行检查，如果发现文件，会直接报告这些问题并且不会运行这些单元测试。go vet值检查子集: `atomic`、`bool`、`buldtags`、`errorsas`、`ifaceassert`、`nilfunc`、`printf`和`stringintconv`。

`go test`可以使用两种模式运行：

- 本地文件夹模式

  在这种模式下， 不指定`packages`参数。它只编译本地文件夹的代码和测试。

  在这种模式下，缓存是被禁止的，所以每次执行总会重新编译。

  最后会显示测试的状态(`ok`或者`FAIL`),包名和耗时。

  比如`go test -v`、`go test -run TestAbc`都是这种模式。

- 包列表模式

  在这种模式下，需要显示指定包名或者包列表。包名的形式有很多种，你可以执行`go help packages`查看包名的格式。

  在这种模式下，`go test`会编译和执行每一个包的代码和测试。

  如果package单测都通过，它仅仅只会输出`ok`信息。如果失败，会输出所有的细节。如果设置了`-bench`或者`-v`,那就对不住了，会显示所有的输出。

  如果任意一个package没有通过单测，最后会输出一个`FAIL`。

在包列表模式下，`go test`会缓存已经成功通过单测的package测试结果，避免不必要的重复测试，它会把缓存的结果显示出来，同时加上`cached`标记。

缓存中匹配的规则是，运行涉及相同的test二进制文件，命令行上的标志完全来自一组受限制的“cachebale”测试标志，比如`-benchtime`、`-cpu`、`-list`、`-parallel`、`-run`、`-short`、`-timeout`、`-failfast`和`-v`。如果运行`go test`时有此集之外的任何测试或非测试标志，则不会缓存结果。若要禁用测试缓存，请使用除cacheable标志以外的任何测试标志或参数。

显式禁用测试缓存的地道方法是使用`-count=1`。

`go clean --testcache`会清除所有cached的单元测试。

`go test` 除了支持build参数(因为它需要把测试代码编译成二进制可执行程序，你可以运行 `go help build` 查看build flag)，还有它自有的一些标志。

```
        -args
             将命令行剩余部分参数传给test二进制。包列表必须在此参数之前。

         -c
             编译test二进制程序为`pkg.test`，只是编译，并不会运行它,后续你可以直接运行这个
程序。`pkg`为包名。
             文件名可以通过`-o`参数指定。

         -exec xprog
             运行二进制程序xprog。

         -i
             安装这个测试所需要的依赖库。已经废弃。

         -json
             转换测试输出为JSON格式，适合工具进行分析。可以运行`go doc test2json`了解细
节。

         -o file
             编译这个test二进制程序为指定的文件,并且会运行此程序。
```

编译的test二进制程序支持控制测试执行的test flag,这些flag在下一节介绍：

go help testflag

执行 `go help testflag` 可以查看 `go test` 支持的test flag。有些是和benchmark相关，有些和fuzz相关，我们会在介绍它们的章节中介绍这些参数，在本节中只介绍和单元测试相关的flag。

```
        -count n
            运行test、benchamrk、fuzz多少次。默认一次。
            如果指定了-cpu，那就是在每个GOMAXPROCS上执行n次。
            Example只会运行一次。使用-fuzz时-count对fuzz test不起作用。

        -cover
            允许代码覆盖率分析。

        -covermode set,count,atomic
            设置代码覆盖率的模式。默认模式是set, 除非-race启用。-race启用下模式为
atomic。
                set: bool: 此语句是否运行?
                count: int: 此语句要执行多少次?
                atomic: int: 类似count, 但是在多线程环境下正确，花费更高、

        -coverpkg pattern1,pattern2,pattern3
            只对模式匹配的包进行代码覆盖率分析。

        -cpu 1,2,4
            为每一个测试指定GOMAXPROCS列表。默认值是当前的GOMAXPROCS。

        -failfast
            第一个测试失败后并不启动新的单元测试。

        -json
            以JSOn格式输出。

        -list regexp
            输出test、benchmark、fuzz test、example列表。

        -parallel n
            允许并发执行设置了t.Parallel的test函数和fuzz targets。

        -run regexp
            只运行匹配的test、example、fuzz test。
        For tests, the regular expression is split by unbracketed
            正则表达式如果有斜杠（/），那么它匹配子测试。比如-run=X/Y 匹配所有X下无子测试
以及X下子测试匹配Y的测试。

        -short
            告诉长时间允许的测试缩短它们的测试时间。

        -shuffle off,on,N
            随机化执行测试和benchmark。默认禁用。

        -timeout d
            如果一个测试运行超过d,就会panic。
            如果d设置为0, timeout参数会被禁用。默认10分钟。

        -v
```

```
        开启详细输出。

    -vet list
        配置go test运行是go vet的检查项。

    -coverprofile cover.out
        将代码覆盖率写入到一个文件中。
```

下面的参数是和profile相关：

```
    -benchmem
        为benchmark输出内存分配统计。

    -blockprofile block.out
        输出goroutine blocking profile到指定的文件。

    -blockprofilerate n
        控制goroutine blocking profile细节，调用runtime.SetBlockProfileRate。

    -cpuprofile cpu.out
        写CPU profile到一个指定的文件。

    -memprofile mem.out
        写allocation profile到一个指定的文件。

    -memprofilerate n
        允许更精准的memory allocation profile,通过设置runtime.MemProfileRate。

    -mutexprofile mutex.out
        写mutex contention profile 到指定的文件。

    -mutexprofilefraction n
        在n个goroutine持有竞争锁的stack trace采样。

    -outputdir directory
        profile输出文件的目录，默认go test运行的目录。

    -trace trace.out
        写execution trace 信息到指定的文件。
```

每一个flag可以加一个`test.`的前缀，比如`-test.v`，这些参数也可以传递给编译的test二进制程序。

比如`go test -v -args -x -v`, 相应的二进制程序`pkg.test -test.v -x -v`。

go help testfunc

在后缀为`_test.go`的程序中，啥样的函数是test、benchmark、fuzz test和example?允许`go help testfunc`可以得到答案。

- **test**: `func TestXxx(t *testing.T) { ... }`
- **benchmark**: `func BenchmarkXxx(b *testing.B) { ... }`
- **fuzz test**: `func FuzzXxx(f *testing.F) { ... }`
- **example**: `func ExamplePrintln() { ... }`

第五章 代码覆盖率

代码覆盖率，也称为测试覆盖率，可衡量自动化测试执行的代码比例。

代码覆盖率工具针对特定的编程语言。 其使用一系列标准衡量覆盖率，包括代码行数、方法或函数、分支和条件。 您可以使用代码覆盖率工具识别代码库尚未被自动化测试覆盖的部分。

监测代码覆盖率指标有助于确保您保持足够的自动化测试水平。 如果代码覆盖率有所下降，则可能表明您没有将自动化测试作为编写新代码的核心要素。

然而，虽然代码覆盖率能够说明测试覆盖了多少代码，但它并不会指示这些测试的有效性或其能否解决所有故障模式。 将代码覆盖率与其他指标相结合，了解自动化测试体系的有效性。

比如针对下面的代码:

```
func Div(a, b int) int {
    return a / b
}
```

单元测试代码如下:

```
func TestDiv(t *testing.T) {
    v := Div(200, 100)
    if v != 2 {
        t.Errorf("expect 2 but got %d", v)
    }
}
```

虽然代码覆盖率是百分之百(`return a/b`被覆盖率)，但是并不是所有的逻辑分支都被覆盖了(Div(200,0)就没有被测试，它会panic)。但是代码覆盖率确实是一个很好的衡量代码质量的一个指标，至少，较少的代码覆盖率吧意味着程序中还很很有的代码并没有被测试到。

举一个非Go语言的例子。SQLite是一个非常流行的嵌入式数据库，它专门有[文章](How SQLite Is Tested)介绍它的测试。它大概有151.3 KSLOC(排除了空格和注释的千行代码)代码，而测试代码是92038.3 KSLOC， 是逻辑代码的608倍。100%的代码覆盖率。

再比如grpc,它的代码覆盖率也是非常的高(如下图，绿色越浓越代码此package或者文件代码覆盖率越高，越红代表覆盖率越低)。

在第四章我们已经介绍了单元测试的几个参数:

- -cover: 开启代码覆盖率分析
- -covermode: 分析代码覆盖率模式。 这个模式的不同的设置，会影响统计的结果。比如set,只是统计代码是否被覆盖。count还是统计代码行覆盖的数量，atomic针对并发测试的统计。
- -coverpkg: 分析指定的package。指定coverpkg好处是它能够跨package统计代码覆盖率, 这篇文章Get Accurate Code Coverage in Go做了详细的测试。最好这个设置成`-coverpkg=./...`而不是`-coverpkg=all`。

生成测试报告

为了更方便查看各单元测试的细节，go提供了工具，可以把代码覆盖率生成网页查看。

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

然后打开cover.html查看各个文件的单侧覆盖情况。如果是covermode=count,还可以显示代码行覆盖的次数的多少。

这种风格很像90年代的网页样式，真的应该好好设计一下网页样式。不过Go已经发布十多年了，居然几乎没有人对此下手，这着实奇怪。

有人也提供了将黑色转换成明亮色的方式:

```
go tool cover -o cover2.html -html=cover.out; sed -i'*.bak'
's/black/whitesmoke/g' cover2.html; rm -fr cover2.html*.bak
```

其中`*.bak`是为了在MacOS也能执行，在Linux环境下`-i'*.bal'`是不需要的。

生成更漂亮的测试报告

matm/gocov-html提供了一种生成junit风格的代码覆盖率的方法。它使用axw/gocov工具生成的测试结果，转换成junit风格的网页。当然gocov也能把go cover生成的测试结果转换成它的格式。

生成treemap图

为了看一个项目的整体的代码状况，我推荐你使用nikolaydubina/go-cover-treemap,它可以生成漂亮的SVG treeamap图，如上面的grpc的代码覆盖率图。

只需要三步就可以生成相应的图。

1. 安装go-cover-treemap工具
2. 使用go test生成代码覆盖率数据
3. 转换数据为svg treemap图

```
$ go install github.com/nikolaydubina/go-cover-treemap@latest
$ go test -coverprofile cover.out ./...
$ go-cover-treemap -coverprofile cover.out > out.svg
```

比如下面这个知名的gohugo项目的代码覆盖率情况：

第六章 第三方工具库

虽然Go testing包中提供了丰富的类型和方法，但是还是有一些非常有用的第三方工具，方便我们编写单元测试代码，这里我给大家介绍几个。

stretchr/testify

第一个不得不说的就是stretchr/testify，它提供了各种各样的断言。

前面我们说过`t.Fail`、`t.Fatal`可以标记单测失败或者停止执行。但是需要预先对期望值和实际的值进行比较，然后在调用`t.Fail`和`t.Fatal`标记失败。

testify可以大大简化相关的判断，如果你使用JUnit相关的工具，应该比较熟悉。

这个库实际有几个子package, 在平常使用中，mock库不太使用，因为我们有更好的mock可以用。suite包有点鸡肋，因为它和官方的概念不相匹配，自定定义出Suite的概念。最常用的是asset包和reuire包。

assert类似Fail,断言失败不会导致此测试停止，后续的逻辑还是会继续执行。

require类似Fatal,一旦测试失败，此测试就会结束。

两个的断言方法都是类似的。

啥事断言呢？看下面的例子：

```
package yours

import (
  "testing"
  "github.com/stretchr/testify/assert"
)

func TestSomething(t *testing.T) {

  // 断言两个值相等，否则就失败,并输出提示文本
  assert.Equal(t, 123, 123, "they should be equal")

  // 断言不相等
  assert.NotEqual(t, 123, 456, "they should not be equal")

  // 断言object是nil值
  assert.Nil(t, object)

  // 断言object非nil
  if assert.NotNil(t, object) {

    // 断言object.Value和某值相等
    assert.Equal(t, "Something", object.Value)

  }
}
```

这里assert也可以换成require.如果换成require,一旦遇到某个测试没通过，那么后续的测试就不再执行了。

什么时候用assert什么时候用require？在处理必要的测试结果用require,比如数据库的连接，如果连接没有建立，那么后续的测试也没有必要测试了，所以此时适合用require。

这个库打磨的真的非常的强大了，在本书中我就不把它的所有方法复制过来了，推荐你看看它的go doc文档,几十个方法提供了丰富的断言。

举个例子，你可以使用`Error`、`NoError`判断error是否为nil。你可以判断两个值是否相等，大于或者小于。你可以判断文件和文件夹是否存在，HTTP的结果的判断。集合是否是增序还是降

序，集合是否包含某值，集合是否为空。判断JSON值是否相等`assert.JSONEq(t, {"hello": "world", "foo": "bar"}, {"foo": "bar", "hello": "world"})`。

google/go-cmp

如果你使用go标准库的方法，判断两个值是否相等并显示值的不同时，如果值的类似是复杂的struct类型，你并不容易把两个值的不同区分出来并显示。google/go-cmp可以提供给你帮助。

你可以使用它打印两个struct的不同。

比如下面的例子：

```
package main

import (
    "fmt"
    "net"
    "time"

    "github.com/google/go-cmp/cmp"
)

func main() {
    // Let got be the hypothetical value obtained from some logic under test
    // and want be the expected golden data.
    got, want := MakeGatewayInfo()

    if diff := cmp.Diff(want, got); diff != "" {
        t.Errorf("MakeGatewayInfo() mismatch (-want +got):\n%s", diff)
    }

}

type (
    Gateway struct {
        SSID      string
        IPAddress net.IP
        NetMask   net.IPMask
        Clients   []Client
    }
    Client struct {
        Hostname  string
        IPAddress net.IP
        LastSeen  time.Time
    }
)

func MakeGatewayInfo() (x, y Gateway) {
    x = Gateway{
        SSID:      "CoffeeShopWiFi",
        IPAddress: net.IPv4(192, 168, 0, 1),
        NetMask:   net.IPv4Mask(255, 255, 0, 0),
        Clients: []Client{{
            Hostname:  "ristretto",
            IPAddress: net.IPv4(192, 168, 0, 116),
        }, {
            Hostname:  "aribica",
            IPAddress: net.IPv4(192, 168, 0, 104),
            LastSeen:  time.Date(2009, time.November, 10, 23, 6, 32, 0,
```

```
time.UTC),
        }, {
            Hostname:  "macchiato",
            IPAddress: net.IPv4(192, 168, 0, 153),
            LastSeen:  time.Date(2009, time.November, 10, 23, 39, 43, 0,
time.UTC),
        }, {
            Hostname:  "espresso",
            IPAddress: net.IPv4(192, 168, 0, 121),
        }, {
            Hostname:  "latte",
            IPAddress: net.IPv4(192, 168, 0, 219),
            LastSeen:  time.Date(2009, time.November, 10, 23, 0, 23, 0,
time.UTC),
        }, {
            Hostname:  "americano",
            IPAddress: net.IPv4(192, 168, 0, 188),
            LastSeen:  time.Date(2009, time.November, 10, 23, 3, 5, 0,
time.UTC),
        }},
    }
    y = Gateway{
        SSID:      "CoffeeShopWiFi",
        IPAddress: net.IPv4(192, 168, 0, 2),
        NetMask:   net.IPv4Mask(255, 255, 0, 0),
        Clients: []Client{{
            Hostname:  "ristretto",
            IPAddress: net.IPv4(192, 168, 0, 116),
        }, {
            Hostname:  "aribica",
            IPAddress: net.IPv4(192, 168, 0, 104),
            LastSeen:  time.Date(2009, time.November, 10, 23, 6, 32, 0,
time.UTC),
        }, {
            Hostname:  "macchiato",
            IPAddress: net.IPv4(192, 168, 0, 153),
            LastSeen:  time.Date(2009, time.November, 10, 23, 39, 43, 0,
time.UTC),
        }, {
            Hostname:  "espresso",
            IPAddress: net.IPv4(192, 168, 0, 121),
        }, {
            Hostname:  "latte",
            IPAddress: net.IPv4(192, 168, 0, 221),
            LastSeen:  time.Date(2009, time.November, 10, 23, 0, 23, 0,
time.UTC),
        }},
    }
```

```
    return x, y
}

var t fakeT

type fakeT struct{}

func (t fakeT) Errorf(format string, args ...interface{}) {
fmt.Printf(format+"\n", args...) }
```

它的输出结果是:

```
MakeGatewayInfo() mismatch (-want +got):
  cmp_test.Gateway{
      SSID:      "CoffeeShopWiFi",
-     IPAddress: s"192.168.0.2",
+     IPAddress: s"192.168.0.1",
      NetMask:   s"ffff0000",
      Clients: []cmp_test.Client{
          ... // 2 identical elements
          {Hostname: "macchiato", IPAddress: s"192.168.0.153", LastSeen:
s"2009-11-10 23:39:43 +0000 UTC"},
          {Hostname: "espresso", IPAddress: s"192.168.0.121"},
          {
              Hostname:  "latte",
-             IPAddress: s"192.168.0.221",
+             IPAddress: s"192.168.0.219",
              LastSeen:  s"2009-11-10 23:00:23 +0000 UTC",
          },
+         {
+             Hostname:  "americano",
+             IPAddress: s"192.168.0.188",
+             LastSeen:  s"2009-11-10 23:03:05 +0000 UTC",
+         },
      },
  }
```

gocovery

smartystreets/goconvey是另外一种单元测试风格。这是一种称之为**行为驱动开发**（Behavior-driven Development）的敏捷开发方式。

BDD的重点是通过与利益相关者的讨论取得对预期的软件行为的清醒认识。它通过用自然语言书写非程序员可读的测试用例扩展了测试驱动开发方法。行为驱动开发人员使用混合了领域中统一的语言的母语语言来描述他们的代码的目的。这让开发者得以把精力集中在代码应该怎么写，而

不是技术细节上，而且也最大程度的减少了将代码编写者的技术语言与商业客户、用户、利益相关者、项目管理者等的领域语言之间来回翻译的代价。

goconvey不仅提供了相应的辅助编写BDD风格的库，还提供了工具，在浏览器中展示测试的结果。

下面是一个使用goconvey编写的单元测试的例子：

```
package s5

import (
    "testing"

    . "github.com/smartystreets/goconvey/convey"
)

func TestGocovery(t *testing.T) {
    Convey("Given a starting integer value", t, func() {
        x := 42

        Convey("When incremented", func() {
            x++

            Convey("The value should be greater by one", func() {
                So(x, ShouldEqual, 43)
            })
            Convey("The value should NOT be what it used to be", func() {
                So(x, ShouldNotEqual, 42)
            })
        })
        Convey("When decremented", func() {
            x--

            Convey("The value should be lesser by one", func() {
                So(x, ShouldEqual, 41)
            })
            Convey("The value should NOT be what it used to be", func() {
                So(x, ShouldNotEqual, 42)
            })
        })
    })
}
```

Convey 在声明一个规范的作用域时使用。每个作用域有一个文本描述信息和一个函数，此函数体内可以调用其它的Convey，Reset或者Should风格的断言。

如你所见Convery可以嵌套。嵌套时最顶层的Convey的函数签名是`Convey(description string, t *testing.T, action func())`，而其它的Convey则不需要传递`t`：`Convey(description string, action func())`或者是`Convey(description string, action func(c C))`。

纵观开源的Go项目和 Go标准库， goconvey这种测试风格使用者还是少数。

第七章 mock使用基础服务的容器以及monkey库

在当今大型的Go应用程序中，整体的架构都会比较复杂，从程序本身来讲，它可能会依赖很多的基础服务，比如数据库存储mysql等、缓存系统redis等，其它的一些基础服务比如Prometheus等，程序的业务逻辑会访问这些基础服务。

同时，微服务架构额流行，也导致一个产品会划分成多个微服务，每个微服务只专注于自己的业务，服务之间通过restful的 API或者RPC框架或者其它方式进行通讯。

这就给测试带来了很大的困难,本来，我们要测试的是很小的一段代码逻辑，比如某个类型的一个方法，但是由于方法体中依赖基础服务或者其它的微服务，这些基础服务和其它的微服务都是线上的服务，没有办法集成测试，即使搭建了线上的环境，也可能因为我们的开发机的环境由于安全等原因，也没有办法直接访问这些服务，导致没有办法执行单元测试。

另外如果我们在开发代码的时候，一开始并没有从易于单元测试的角度去开率设计方法实现，导致很难进行单元测试，比如实现某个方法的时候，把所有的逻辑都写在一个方法实现中:

```
func (s *S) AMethod() {
    读取一些配置

    建立数据库的连接

    访问数据库

    执行一些业务逻辑

    访问第三方的微服务

    访问redis缓存

    执行一些业务逻辑

    调用另一个微服务

    返回
}
```

一些代码规范或者最佳实践会告诉你，每一个函数和方法的实现应该保持尽量的小，它们只执行单一的业务逻辑。面向对象编程五大原则之一就是单一职责原则：一个类或者一个函数应该只负责程序中的单一逻辑部分。比如上面的代码，如果拆分成多个方法，一个方法的返回值可以是另外一个方法的输入参数，这样我们在实现这些方法的单元测试时，就可以mock这些参数，针对这些方法进行测试、

Mock的方式有很多，你可以自己写一些Mock对象，也可以使用第三方的mock库。这里我只举几个mock库例子。

google/mock

google/mock是已个Go testing集合非常好的mock库，star数很高，贡献者也众多，也是值得一试。

你可以通过下面的命令安装mockgen工具，它可以为借口对象生成mock实现。

```
go install github.com/golang/mock/mockgen@latest
```

比如你可以为标准库的`io.Writer,ioReader`生成Mock对象，不不必关心自动生成的Mock对象，而是直接使用它们即可。

```
mockgen -destination io_test.go io Reader,Writer
```

重要的你可以实现`Controler` mock相应的对象:

```
func TestReader(t *testing.T) {
    ctrl := gomock.NewController(t)
    defer ctrl.Finish()

    r := NewMockReader(ctrl)

    r.EXPECT().Read(gomock.All()).Do(func(arg0 []byte) {
        for i := 0; i < 10; i++ {
            arg0[i] = 'a' + byte(i)
        }
    }).Return(10, nil).AnyTimes()

    data := make([]byte, 1024)
    n, err := r.Read(data)
    assert.NoError(t, err)
    assert.Equal(t, 10, n)
    assert.Equal(t, "abcdefghij", string(data[:n]))

}
```

DATA-DOG/go-sqlmock

我们很多的应用都是和·数据库访问相关的，测试和数据库相关函数时就相当的痛苦。有时候我们使用sqlite3这样的嵌入式数据库来作为集成数据库测试，但是我们也可以mock数据库对象。datadog就提供了这样一个非常好用的库：DATA-DOG/go-sqlmock: Sql mock driver for golang to test database interactions。

假定你使用 go-sql-driver/mysql 提供数据库访问的功能，`recordStats` 提供了数据:

```
package main

import (
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

func recordStats(db *sql.DB, userID, productID int64) (err error) {
    tx, err := db.Begin()
    if err != nil {
        return
    }

    defer func() {
        switch err {
        case nil:
            err = tx.Commit()
        default:
            tx.Rollback()
        }
    }()

    if _, err = tx.Exec("UPDATE products SET views = views + 1"); err != nil {
        return
    }
    if _, err = tx.Exec("INSERT INTO product_viewers (user_id, product_id)
VALUES (?, ?)", userID, productID); err != nil {
        return
    }
    return
}

func main() {
    // @NOTE: the real connection is not required for tests
    db, err := sql.Open("mysql", "root@/blog")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    if err = recordStats(db, 1 /*some user id*/, 5 /*some product id*/); err !=
nil {
        panic(err)
    }
}
```

我们就可以使用个`go-sqlmock`来进行正面测试和负面测试:

```go
package main

import (
    "fmt"
    "testing"

    "github.com/DATA-DOG/go-sqlmock"
)

// 成功的单测
func TestShouldUpdateStats(t *testing.T) {
    db, mock, err := sqlmock.New()
    if err != nil {
        t.Fatalf("an error '%s' was not expected when opening a stub database
connection", err)
    }
    defer db.Close()

    mock.ExpectBegin()
    mock.ExpectExec("UPDATE products").WillReturnResult(sqlmock.NewResult(1,
1)) // 执行update ...的时候返回值
    mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2,
3).WillReturnResult(sqlmock.NewResult(1, 1)) // 执行insert ...的返回值
    mock.ExpectCommit()

    // now we execute our method
    if err = recordStats(db, 2, 3); err != nil {
        t.Errorf("error was not expected while updating stats: %s", err)
    }

    // we make sure that all expectations were met
    if err := mock.ExpectationsWereMet(); err != nil {
        t.Errorf("there were unfulfilled expectations: %s", err)
    }
}

// 失败的单测
func TestShouldRollbackStatUpdatesOnFailure(t *testing.T) {
    db, mock, err := sqlmock.New()
    if err != nil {
        t.Fatalf("an error '%s' was not expected when opening a stub database
connection", err)
    }
    defer db.Close()

    mock.ExpectBegin()
    mock.ExpectExec("UPDATE products").WillReturnResult(sqlmock.NewResult(1,
```

```
1))
    mock.ExpectExec("INSERT INTO product_viewers").
        WithArgs(2, 3).
        WillReturnError(fmt.Errorf("some error")) // 返回错误
    mock.ExpectRollback()

    // now we execute our method
    if err = recordStats(db, 2, 3); err == nil {
        t.Errorf("was expecting an error, but there was none")
    }

    // we make sure that all expectations were met
    if err := mock.ExpectationsWereMet(); err != nil {
        t.Errorf("there were unfulfilled expectations: %s", err)
    }
}
```

标准库的httptest

Go标准库httptest提供了测试handler和模拟http server的方法。

经常，我们会使用Go的http包写一些web应用程序，我们会实现一些handler,测试这些handler其实很方便,使用httptest.NewRecorder就可以模拟一个http.ResponseWriter:

```go
package httptest

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestHandler(t *testing.T) {
    handler := func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "<html><body>Hello World!</body></html>")
    }

    req := httptest.NewRequest("GET", "http://example.com/foo", nil)
    w := httptest.NewRecorder()
    handler(w, req)

    resp := w.Result()
    body, _ := io.ReadAll(resp.Body)

    fmt.Println(resp.StatusCode)
    fmt.Println(resp.Header.Get("Content-Type"))
    fmt.Println(string(body))

}
```

如果我们想测试client的行为，需要临时启动一个server进行测试，也可以使用httptest启动一个http server,还可以支持http2:

```
package httptest

import (
    "fmt"
    "io"
    "log"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestHTTPServer(t *testing.T) {
    ts := httptest.NewUnstartedServer(http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello, %s", r.Proto)
    }))
    ts.EnableHTTP2 = true
    ts.StartTLS()
    defer ts.Close()

    res, err := ts.Client().Get(ts.URL)
    if err != nil {
        log.Fatal(err)
    }
    greeting, err := io.ReadAll(res.Body)
    res.Body.Close()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s", greeting)

}
```

当然还有一些其它的mock库，如果有需要，你也可以快速评估一下，看看能否应用到你的单元测试中，为你的单元测试提供便利：

- https://github.com/h2non/gock
- GitHub - gavv/httpexpect: End-to-end HTTP and REST API testing for Go.
- GitHub - steinfletcher/apitest: A simple and extensible behavioural testing library for Go. You can use api test to simplify REST API, HTTP handler and e2e tests.
- GitHub - carlmjohnson/be: Generic testing helper for Go
- GitHub - jfilipczyk/gomatch: Library created for testing JSON against patterns.

容器化服务

虽然mock能够解决一部分的单元测试的痛点，但是mock库一般要求mock的对象是接口，你的函数的输入参数类型也得要求是接口类型，这会将大部分的业务函数和方法排除在外。另外写这些mock对象有时候也非常麻烦，没有直接的集成测试方便，所以目前有几个库，提供了集成容器测试的功能，那么依赖的基础服务，比如redis、mysql、kafka等，在测试额时候启动这些容器，测试完毕后再删除这些容器。

使用这些容器的时候，要求你的docker要运行着，因为这些库会和docker服务交互，拉去镜像、启动容器、停止容器和删除容器。

为了减少测试的时间，你最好先把这些镜像手工下载下来。

当然你不使用这些库，你也可以手工启动容器配合测试。使用这些库的好处是自动化，比如自动化在kafka上创建 topic、测试完毕后自动删除容器。

这里我们看三个相关的库。

arikama/go-mysql-test-container

go-mysql-test-container是一个专门用来集成mysql的库。

因为只针对集成mysql,所以它使用起来也非常的简单:

```
package main

import (
    "testing"

    "github.com/arikama/go-mysql-test-container/mysqltestcontainer"
)

func Test(t *testing.T) {
    mySql, _ := mysqltestcontainer.Create("test")
    db := mySql.GetDb()
    err := db.Ping()
    if err != nil {
        log.L.Errorln(err.Error())
    }
}
```

testcontainers/testcontainers-go

事实上上面的go-mysql-test-container底层使用的是testcontainers/testcontainers-go)库，只不过对mysql的集成做了进一步的封装。

`testcontainers-go` 提供了一个通用的集成docker容器的方法。比如集成redis:

```
package container

import (
    "context"
    "testing"

    "github.com/go-redis/redis"
    "github.com/stretchr/testify/assert"
    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

func TestWithRedis(t *testing.T) {
    ctx := context.Background()
    req := testcontainers.ContainerRequest{
        Image:        "redis:latest",
        ExposedPorts: []string{"6379/tcp"},
        WaitingFor:   wait.ForLog("Ready to accept connections"),
    }
    redisC, err := testcontainers.GenericContainer(ctx,
testcontainers.GenericContainerRequest{
        ContainerRequest: req,
        Started:          true,
    })
    if err != nil {
        t.Error(err)
    }
    defer redisC.Terminate(ctx)

    rdb := redis.NewClient(&redis.Options{
        Addr:     "localhost:6379",
        Password: "", // no password set
        DB:       0,  // use default DB
    })

    err = rdb.Set("key", "value", 0).Err()
    assert.NoError(t, err)

    val, err := rdb.Get("key").Result()
    assert.NoError(t, err)
    assert.Equal(t, "value", val)

}
```

又或者集成kafka:

```go
package container

import (
    "context"
    "strings"
    "testing"
    "time"

    "github.com/google/uuid"
    "github.com/segmentio/kafka-go"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
    "github.com/testcontainers/testcontainers-go"
)

func TestWithKafka(t *testing.T) {
    kafkaContainer := testcontainers.NewLocalDockerCompose(
        []string{"testdata/docker-compose.yml"},
        strings.ToLower(uuid.New().String()),
    )
    execError := kafkaContainer.WithCommand([]string{"up", "-d"}).Invoke()
    require.NoError(t, execError.Error)

    // kafka starts ver slow
    time.Sleep(time.Minute)
    defer destroyKafka(kafkaContainer)

    // test write
    w := &kafka.Writer{
        Addr:     kafka.TCP("localhost:9092"),
        Topic:    "test-topic",
        Balancer: &kafka.LeastBytes{},
    }

    err := w.WriteMessages(context.Background(),
        kafka.Message{
            Key:   []byte("Key-A"),
            Value: []byte("Hello World!"),
        },
        kafka.Message{
            Key:   []byte("Key-B"),
            Value: []byte("One!"),
        },
        kafka.Message{
            Key:   []byte("Key-C"),
            Value: []byte("Two!"),
        },
```

```
    )
    assert.NoError(t, err)
    err = w.Close()
    assert.NoError(t, err)

    // test read
    r := kafka.NewReader(kafka.ReaderConfig{
        Brokers:   []string{"localhost:9092"},
        Topic:     "test-topic",
        Partition: 0,
        MinBytes:  10e3, // 10KB
        MaxBytes:  10e6, // 10MB
    })

    m, err := r.ReadMessage(context.Background())
    assert.NoError(t, err)
    assert.NotEmpty(t, m)

}

func destroyKafka(compose *testcontainers.LocalDockerCompose) {
    compose.Down()
    time.Sleep(1 * time.Second)
}
```

其中kafka是通过docker compose启动的,它的配置如下:

```
version: "3"
services:
  zookeeper:
    image: wurstmeister/zookeeper:latest
    container_name: zookeeper
    expose:
      - 2181
  kafka:
    image: wurstmeister/kafka:latest
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "test-topic:1:1"
```

orlangure/gnomock

`testcontainers-go` 使用起来稍显繁琐，gnomock也提供了一个更为简单的办法，并且提供了十几种基础服务的集成，比如redis、kafka、mongodb等。

比如它的kafka集成测试就比较简单了:

```go
package gnomock_test

import (
    "context"
    "os"
    "testing"
    "time"

    "github.com/orlangure/gnomock"
    "github.com/orlangure/gnomock/preset/kafka"
    kafkaclient "github.com/segmentio/kafka-go"
    "github.com/stretchr/testify/require"
)

func TestKafka(t *testing.T) {
    // init some messages
    messages := []kafka.Message{
        {
            Topic: "events",
            Key:   "order",
            Value: "1",
            Time:  time.Now().UnixNano(),
        },
        {
            Topic: "alerts",
            Key:   "CPU",
            Value: "92",
            Time:  time.Now().UnixNano(),
        },
    }

    p := kafka.Preset(
        kafka.WithTopics("topic-1", "topic-2"),
        kafka.WithMessages(messages...),
    )

    // start the kafka container
    container, err := gnomock.Start(
        p,
        gnomock.WithDebugMode(), gnomock.WithLogWriter(os.Stdout),
        gnomock.WithContainerName("kafka"),
        gnomock.WithUseLocalImagesFirst(),
    )
    require.NoError(t, err)
    // stop the kafka container on exit
    defer func() { require.NoError(t, gnomock.Stop(container)) }()
```

```
    ctx, cancel := context.WithTimeout(context.Background(), time.Second*30)
    defer cancel()

    alertsReader := kafkaclient.NewReader(kafkaclient.ReaderConfig{
        Brokers: []string{container.Address(kafka.BrokerPort)},
        Topic:   "alerts",
    })

    m, err := alertsReader.ReadMessage(ctx)
    require.NoError(t, err)
    require.NoError(t, alertsReader.Close())

    require.Equal(t, "CPU", string(m.Key))
    require.Equal(t, "92", string(m.Value))

    eventsReader := kafkaclient.NewReader(kafkaclient.ReaderConfig{
        Brokers: []string{container.Address(kafka.BrokerPort)},
        Topic:   "events",
    })

    m, err = eventsReader.ReadMessage(ctx)
    require.NoError(t, err)
    require.NoError(t, eventsReader.Close())

    require.Equal(t, "order", string(m.Key))
    require.Equal(t, "1", string(m.Value))

    c, err := kafkaclient.Dial("tcp", container.Address(kafka.BrokerPort))
    require.NoError(t, err)

    require.NoError(t, c.DeleteTopics("topic-1", "topic-2"))
    require.Error(t, c.DeleteTopics("unknown-topic"))

    require.NoError(t, c.Close())
}
```

redis的集成也很简单:

```go
package gnomock_test

import (
    "fmt"
    "testing"

    redisclient "github.com/go-redis/redis/v7"
    "github.com/orlangure/gnomock"
    "github.com/orlangure/gnomock/preset/redis"
)

func TestRedis(t *testing.T) {
    vs := make(map[string]interface{})

    vs["a"] = "foo"
    vs["b"] = 42
    vs["c"] = true

    p := redis.Preset(redis.WithValues(vs))
    container, _ := gnomock.Start(p,
        gnomock.WithDebugMode(),
        gnomock.WithUseLocalImagesFirst(),
    )

    defer func() { _ = gnomock.Stop(container) }()

    addr := container.DefaultAddress()
    client := redisclient.NewClient(&redisclient.Options{Addr: addr})

    fmt.Println(client.Get("a").Result())

    var number int

    err := client.Get("b").Scan(&number)
    fmt.Println(number, err)

    var flag bool

    err = client.Get("c").Scan(&flag)
    fmt.Println(flag, err)
}
```

monkey/gomonkey

Mock方式使用起来有很多的限制，容器化的方式又依赖docker环境，测试也很慢，只能集成通用的基础服务，那么有没有一种方法能更好的用来单元测试呢?

有!

这里就不得不提到bouk/monkey，这个库可以说是开创了一个运行时修改函数的门派，好几个库都是基于他的思想去实现的，尽管后来的库有各种方法和提供遍历的手段，但是核心思想还是受到`boulk/monkey`的影响。

此库的作者专门写了一篇文章，介绍这个库的实现方法：Monkey Patching in Go (bou.ke)。

这个库的实现的功能太过“邪恶”，除了测试作者不建议你把它应用到线上产品。，所以它的版权也很特殊，这个库也被被归档了：

> Copyright Bouke van der Bijl
>
> I do not give anyone permissions to use this tool for any purpose. Don't use it.
>
> I’m not interested in changing this license. Please don’t ask.

这个库最大的本事就是可以在运行时修改方法，比如下面的程序就修改了`fmt.Println`方法：

```
package main

import (
    "fmt"
    "os"
    "strings"

    "bou.ke/monkey"
)

func main() {
    monkey.Patch(fmt.Println, func(a ...interface{}) (n int, err error) {
        s := make([]interface{}, len(a))
        for i, v := range a {
            s[i] = strings.Replace(fmt.Sprint(v), "hell", "*bleep*", -1)
        }
        return fmt.Fprintln(os.Stdout, s...)
    })
    fmt.Println("what the hell?") // what the *bleep*?
}
```

但是这个库如果用来辅助单元测试，就太方便了。因为我们可以在单元测试的时候，修改相关依赖的函数和方法，不实际调用这些依赖的函数和方法，而是使用一个期望的返回值做替换(当然也可以修改输入参数)，可以完全不依赖基础服务和第三方服务，单元测试就不会再有什么阻碍了。但是这个库的版权限制了我们的使用。

中兴通讯资深架构师张晓龙基于monkey思想，实现了一个适合用来单元测试的monkey patching库：agiledragon/gomonkey。

gomonkey不是一个monkey的简单克隆，它专注于单元测试，所以提供了很多的便利的方法：

- 可以patch 函数
- 可以patch public 方法
- 可以patch private 方法
- 可以patch 接口
- 可以patch函数变量
- 可以patch全局变量
- 可以为函数、方法、接口和函数变量提供指定顺序的返回值，适合它们多次调用的场景

这个库提供很大量的单元测试，可以用来参考和学习。这里我举一个简单的例子：

```
func TestPrintlnByGomonkey(t *testing.T) {
    patches := gomonkey.ApplyFunc(fmt.Println, func(a ...any) (n int, err
error) {

        return fmt.Fprintln(os.Stdout, "I have changed the arguments")
    })
    defer patches.Reset()

    fmt.Println("hello world")
}
```

不过monkey/gomonkey在MacOS上使用有问题，会报错或者不起作用。下面是在MacPro M1上的一种配置。

1. 首先在MacOS安装的Go架构版本是Amd64, 而不是arm64。安装 Amd64de1Go版本可以正常运行。
2. 使用eisenxp/macos-golink-wrapper解决mprotect权限问题

注意运行gomonkey的时候，你需要加上`-gcflags=all=-l`避免函数内联。

第八章 web 程序单元测试实战

前面我们学习了很多的知识，也看到了一些例子，大家也对Go Test相关的之后有了一个全面的了解。这一章我通过一个常见的web程序示例，演示如何组织和应用单元测试。

首先这是一个普通的多层的web应用程序，按照分层的架构程序也分成了几个package:

- model: 数据模型， 定义了Book类型和Author，只是类型的定义，并没有业务逻辑，不需要单元测试
- dao: 提供数据库的访问。作为例子这里提供了一个`InsertBook`的实现，往数据库中插入一条书籍的信息
- service：提供CreateBook方法，实际可以检查书籍是否存在，作者是否存在等逻辑判断，如果满足则调用dao的函数插入一条书籍信息
- api：也叫controller,提供CreateBook handler,检查request的参数，如果满足则调用service层创建书籍

DAO层测试

dao层依赖数据库，根据先前的介绍，有多重方法可以提供测试。

这一次我想通过使用sqlmock的方式提供测试，不依赖任何数据库或者容器。

```
func TestBookStore_InsertBook(t *testing.T) {
    db, mock, err := sqlmock.New()
    assert.NoError(t, err, "an error '%s' was not expected when opening a stub database connection", err)
    defer db.Close()

    store := &bookStore{
        db: db,
    }

    book := &model.Book{
        Title:    "The Go Programming Language",
        AuthorID: 1,
        ISBN:     "978-0134190440",
        Subject:  "computers",
    }

    mock.ExpectExec("INSERT INTO books").WillReturnResult(sqlmock.NewResult(1, 1))

    err = store.InsertBook(context.TODO(), book)

    require.NoError(t, err)
    assert.NotZero(t, book.ID)
}
```

这里使用sqlmock mock Exec方法，假定它执行成功，返回lastID和影响行数。执行这个单元测试可以看到测试成功。

Service层的测试

service依赖dao层，依然可以使用sqlmock模拟测试，但是这一次我想换一种测试方法。使用嵌入式数据sqlite3进行真正的数据库访问。

```
func TestBookService_CreateBook(t *testing.T) {
    db := util.CreateTestDB(t)
    defer db.Close()

    bookService := NewBookService(db)

    book := &model.Book{
        Title:    "The Go Programming Language",
        AuthorID: 1,
        ISBN:     "978-0134190440",
        Subject:  "computers",
    }
    err := bookService.CreateBook(context.TODO(), book)

    require.NoError(t, err)
    assert.NotZero(t, book.ID)
}
```

```
func CreateTestDB(t *testing.T) *sql.DB {
    db, err := sql.Open("sqlite3", "file:../testdata/test.db?cache=shared")
    assert.NoError(t, err)

    db.Exec(`
    CREATE TABLE IF NOT EXISTS books (
        Id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT,
        isbn TEXT,
        subject TEXT,
        author_id INTEGER
     );
     DELETE FROM books;
    `)

    return db
}
```

一般我们会把测试用的材料放在testdata文件夹下。testdata文件夹是一个特殊的文件夹。go编译器不会把它作为一个子package进行编译。

API 层的测试

同样， API层也有多重测试方法。首先我们看使用嵌入式数据库的方法，它把service层、dao层串联起来，其实是一个集成测试:

```go
package api

import (
    "encoding/json"
    "io"
    "net/http"
    "net/http/httptest"
    "net/url"
    "strings"
    "testing"

    "github.com/smallnest/go_test_workshop/s7/book/model"
    "github.com/smallnest/go_test_workshop/s7/book/service"
    "github.com/smallnest/go_test_workshop/s7/book/util"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestBookController_GetBook(t *testing.T) {
    db := util.CreateTestDB(t)
    defer db.Close()

    bookService := service.NewBookService(db)
    bc := &BookController{bookService: bookService}

    data := url.Values{}
    data.Set("title", "The Go Programming Language")
    data.Set("autherID", "1")
    data.Set("isbn", "978-0134190440")
    data.Set("subject", "computers")

    r := httptest.NewRequest("POST", "http://example.com/foo",
strings.NewReader(data.Encode()))
    r.Header.Add("Content-Type", "application/x-www-form-urlencoded")

    w := httptest.NewRecorder()
    bc.CreateBook(w, r)

    resp := w.Result()
    require.Equal(t, http.StatusOK, resp.StatusCode)
    body, _ := io.ReadAll(resp.Body)

    var book model.Book
    err := json.Unmarshal(body, &book)
    require.NoError(t, err)
    assert.NotZero(t, book.ID)
}
```

我们也可以使用gomonkey只针对api这一层进行测试。哪一种测试方法好呢？并无定式，哪一种写起来更方便，能够覆盖你的测试场景就用哪种。你最熟练使用哪一种就用哪一种。

```go
package api

import (
    "context"
    "encoding/json"
    "io"
    "net/http"
    "net/http/httptest"
    "net/url"
    "strings"
    "testing"

    "github.com/agiledragon/gomonkey/v2"
    "github.com/smallnest/go_test_workshop/s7/book/model"
    "github.com/smallnest/go_test_workshop/s7/book/service"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestBookController_GetBook_By_Monkey(t *testing.T) {
    bookService := service.NewBookService(nil)
    bc := &BookController{bookService: bookService}

    data := url.Values{}
    data.Set("title", "The Go Programming Language")
    data.Set("autherID", "1")
    data.Set("isbn", "978-0134190440")
    data.Set("subject", "computers")

    r := httptest.NewRequest("POST", "http://example.com/foo",
strings.NewReader(data.Encode()))
    r.Header.Add("Content-Type", "application/x-www-form-urlencoded")

    book := &model.Book{
        ID:       6,
        Title:    "The Go Programming Language",
        AuthorID: 1,
        ISBN:     "978-0134190440",
        Subject:  "computers",
    }
    patches := gomonkey.ApplyMethodFunc(bookService, "CreateBook", func(ctx
context.Context, b *model.Book) error {
        b.ID = 6

        return nil
    })
    defer patches.Reset()
```

```
    w := httptest.NewRecorder()
    bc.CreateBook(w, r)

    resp := w.Result()
    require.Equal(t, http.StatusOK, resp.StatusCode)
    body, _ := io.ReadAll(resp.Body)

    err := json.Unmarshal(body, &book)
    require.NoError(t, err)
    assert.NotZero(t, book.ID)
}
```

第九章 Benchmark

`Test`是Go Test概念中的一种测试，我们把它称之为单元测试或者叫做测试。Go还提供了其它三种类型：

- benchmark：性能测试
- example: 示例
- fuzz: 模糊测试

这一章，我们重点介绍benchmark测试。benchmark测试主要用来测试函数和方法的性能。Go标准库中有很多的benchmark的代码，以便验证Go运行时和标准库的性能。如果你给Go标准库提供一个性能优化的代码，你一定要运行相关的benchmark或者补充一系列场景下的benchmark。比如字节跳动语言团队在Go 1.19提交了一个Sort方法的性能优化，就提供了大量的排序场景，以便验证新的算法比先前的算法确实有优化：

benchmark的函数依然放在以`_test.go`为后缀的文件中，你可以和包含单元测试函数的文件分开，也可以和包含单元测试的函数写在一起。

benchmark函数的签名如下所示：

```
func BenchmarkXxx(b *testing.B) { ... }
```

和单元测试的函数签名类似，只不过`Test`换成了`Benchmark`,函数参数变成了`b *testing.B`。

使用`b.N`，你可以根据benchmark自动调整的运行次数退出测试。

下面是Benchmark的一个例子。

```go
package s8

import (
    "crypto/md5"
    "crypto/rand"
    "crypto/sha1"
    "crypto/sha256"
    "crypto/sha512"
    "encoding/hex"
    "fmt"
    "hash/crc32"
    "hash/fnv"
    _ "runtime"
    "testing"
    "unsafe"

    xxhashasm "github.com/cespare/xxhash"
    "github.com/creachadair/cityhash"
    afarmhash "github.com/dgryski/go-farm"
    farmhash "github.com/leemcloughlin/gofarmhash"
    "github.com/minio/highwayhash"
    "github.com/pierrec/xxHash/xxHash64"
    "github.com/spaolacci/murmur3"
)

var n int64
var testBytes []byte

func BenchmarkHash(b *testing.B) {
    sizes := []int64{32, 64, 128, 256, 512, 1024}
    for _, n = range sizes {
        testBytes = make([]byte, n)
        readN, err := rand.Read(testBytes)
        if readN != int(n) {
            panic(fmt.Sprintf("expect %d but got %d", n, readN))
        }
        if err != nil {
            panic(err)
        }

        b.Run(fmt.Sprintf("Sha1-%d", n), benchmarkSha1)
        b.Run(fmt.Sprintf("Sha256-%d", n), benchmarkSha256)
        b.Run(fmt.Sprintf("Sha512-%d", n), benchmarkSha512)
        b.Run(fmt.Sprintf("MD5-%d", n), benchmarkMD5)
        b.Run(fmt.Sprintf("Fnv-%d", n), benchmarkFnv)
        b.Run(fmt.Sprintf("Crc32-%d", n), benchmarkCrc32)
        b.Run(fmt.Sprintf("CityHash-%d", n), benchmarkCityhash)
```

```
        b.Run(fmt.Sprintf("FarmHash-%d", n), benchmarkFarmhash)
        b.Run(fmt.Sprintf("Farmhash_dgryski-%d", n), benchmarkFarmhash_dgryski)
        b.Run(fmt.Sprintf("Murmur3-%d", n), benchmarkMurmur3)
        b.Run(fmt.Sprintf("Highwayhash-%d", n), benchmarkHighwayhash)
        b.Run(fmt.Sprintf("XXHash64-%d", n), benchmarkXXHash64)
        b.Run(fmt.Sprintf("XXHash64_ASM-%d", n), benchmarkXXHash64_ASM)
        b.Run(fmt.Sprintf("MapHash64-%d", n), benchmarkMapHash64)
        fmt.Println()
    }

}

func benchmarkSha1(b *testing.B) {
    x := sha1.New()

    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}
func benchmarkSha256(b *testing.B) {
    x := sha256.New()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}

func benchmarkSha512(b *testing.B) {
    x := sha512.New()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}
```

```
func benchmarkMD5(b *testing.B) {
    x := md5.New()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}

func benchmarkCrc32(b *testing.B) {
    x := crc32.NewIEEE()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum32()
    }
}

func benchmarkFnv(b *testing.B) {
    x := fnv.New64()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum64()
    }
}

func benchmarkCityhash(b *testing.B) {
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        _ = cityhash.Hash64WithSeed(testBytes, 0xCAFE)
    }
}

func benchmarkFarmhash(b *testing.B) {
    b.SetBytes(n)
    b.ResetTimer()
```

```
    for i := 0; i < b.N; i++ {
        _ = farmhash.Hash64WithSeed(testBytes, 0xCAFE)
    }
}

func benchmarkFarmhash_dgryski(b *testing.B) {
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        _ = afarmhash.Hash64WithSeed(testBytes, 0xCAFE)
    }
}

func benchmarkMurmur3(b *testing.B) {
    x := murmur3.New64()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum64()
    }
}
func benchmarkHighwayhash(b *testing.B) {
    key, _ :=
hex.DecodeString("000102030405060708090A0B0C0D0E0FF0E0D0C0B0A090807060504030201
000") // use your own key here
    x, _ := highwayhash.New64(key)
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum64()
    }
}

func benchmarkXXHash64(b *testing.B) {
    x := xxHash64.New(0xCAFE)
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
```

```
        x.Write(testBytes)
        _ = x.Sum64()
    }
}

func benchmarkXXHash64_ASM(b *testing.B) {
    x := xxhashasm.New()
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum64()
    }
}

//go:noescape
//go:linkname memhash runtime.memhash
func memhash(p unsafe.Pointer, h, s uintptr) uintptr

type stringStruct struct {
    str unsafe.Pointer
    len int
}

func MemHash(data []byte) uint64 {
    ss := (*stringStruct)(unsafe.Pointer(&data))
    return uint64(memhash(ss.str, 0, uintptr(ss.len)))
}
func benchmarkMapHash64(b *testing.B) {
    b.SetBytes(n)
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        _ = MemHash(testBytes)
    }
}
```

这段代码摘自我的一个github 项目: smallnest/hash-bench,用来比较各种Hash算法的性能。

为了测试哈希不同的数据大小的各种算法的性能，我实现了子测试。针对几种数据大小，分别执行子测试:

```
64
var testBytes []byte

func BenchmarkHash(b *testing.B) {
    sizes := []int64{32, 64, 128, 256, 512, 1024}
    for _, n = range sizes {
        testBytes = make([]byte, n)
        readN, err := rand.Read(testBytes)
        if readN != int(n) {
            panic(fmt.Sprintf("expect %d but got %d", n, readN))
        }
        if err != nil {
            panic(err)
        }

        b.Run(fmt.Sprintf("Sha1-%d", n), benchmarkSha1)
        b.Run(fmt.Sprintf("Sha256-%d", n), benchmarkSha256)
        b.Run(fmt.Sprintf("Sha512-%d", n), benchmarkSha512)
        b.Run(fmt.Sprintf("MD5-%d", n), benchmarkMD5)
        b.Run(fmt.Sprintf("Fnv-%d", n), benchmarkFnv)
        b.Run(fmt.Sprintf("Crc32-%d", n), benchmarkCrc32)
        b.Run(fmt.Sprintf("CityHash-%d", n), benchmarkCityhash)
        b.Run(fmt.Sprintf("FarmHash-%d", n), benchmarkFarmhash)
        b.Run(fmt.Sprintf("Farmhash_dgryski-%d", n), benchmarkFarmhash_dgryski)
        b.Run(fmt.Sprintf("Murmur3-%d", n), benchmarkMurmur3)
        b.Run(fmt.Sprintf("Highwayhash-%d", n), benchmarkHighwayhash)
        b.Run(fmt.Sprintf("XXHash64-%d", n), benchmarkXXHash64)
        b.Run(fmt.Sprintf("XXHash64_ASM-%d", n), benchmarkXXHash64_ASM)
        b.Run(fmt.Sprintf("MapHash
```

运行`go test -benchmem -bench . -v`:

它会把每一个函数按照耗时罗列出来。这里我们使用了`-benchmem`参数，所以还会把每个op的分配的次数和字节数显示出来。 一般来说内存分配次数越少，分配的字节数越少性能会越高，所以它给你指明了一个优化的方向。当然你在测试代码中调用`b.ReportAllocs()`也会报告内存分配信息，即使你没有设置`-benchmem`参数。

在代码中我们还加入了`b.SetBytes(n)`,所以结果中还报告了每秒处理的字节数，当然在本测试中处理的越多越好。

`-benchtime t`可以设置充足的测试次数，直到指定的测试时间。如果不是时间，而是倍数的话，指定的是测试次数，如`-benchtime 100x`。

并发benchmark

有时候我们想测试一下并发情况下的性能，比如对一个kv数据结构，我们想了解它在并发读写情况下的性能。这个时候就要用到并发benchmark。

比如下面这段代码有两个benchmark函数，分别测试并发情况下的性能:

```
package s8_test

import (
    "crypto/sha256"
    "fmt"
    "runtime"
    "testing"
)

func init() {
    fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(-1))
}

func BenchmarkParallelHash(b *testing.B) {
    testBytes := make([]byte, 1024)

    b.ResetTimer()
    b.SetParallelism(20)
    b.RunParallel(func(pb *testing.PB) {
        x := sha256.New()
        for pb.Next() {
            x.Reset()
            x.Write(testBytes)
            _ = x.Sum(nil)
        }
    })
}

func BenchmarkNonParallelHash(b *testing.B) {
    testBytes := make([]byte, 1024)

    b.ResetTimer()
    x := sha256.New()
    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}
```

注意这里`xxx ns/op`可能会导致误导。这里的时间是墙上的时钟流逝的时间，并不是一个op所需的时间，所以你可以看到并发情况下因为有多个goroutine(此图中是6个)并发执行，所以总体的时间加快了，接近串行执行的六分之一。

你可以调用`b.SetParallelism(p)`调大并发度为`p*GOMAXPROCS`

同一函数优化后的性能比较

除了同时比较多个不同函数的性能，比如序列化反序列化，defer函数的影响，不同http touter的性能，我们还可以比较同一个函数不同优化后的性能，比如前面提到的Sort函数的优化。、

假设我们有一个Hash函数，先前是通过sha256实现的:

```
func BenchmarkExample(b *testing.B) {
    x := sha256.New()
    testBytes := make([]byte, 1024)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}
```

我们运行benchmark，测试5次， 将结果写入到`old.txt`:

```
go test -bench BenchmarkExample . -v -count=5 > old.txt
```

我们换一种实现，换成Go 1.19中的maphash实现:

```
func BenchmarkExample(b *testing.B) {
    x := maphash.Hash{}
    testBytes := make([]byte, 1024)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        x.Reset()
        x.Write(testBytes)
        _ = x.Sum(nil)
    }
}
```

也执行benchmark5次，写入到`new.txt`中:

```
go test -bench BenchmarkExample . -v -count=5 > new.txt
```

接下来就可以使用benchstat进行比较了。你如果没有安装benchstat命令，那么你可以执行`go install golang.org/x/perf/cmd/benchstat@latest`安装先。

我们看第一行就好了。第二行和第三行是对分配内存和分配次数的比较。

可以看到先前的Hash实现美哦个op需要3.60us左右，而新的hash算法值需要0.17us,减少了95.29%,说明新的Hash优势巨大。

`p=0.008`是可信度，越小可信度越高。统计学中通常把`p=0.05`作为临界值，超过此值说明结果不可信。

后面的`n=5+5`说明采用了老的测试的5个样本，新的测试的5个样本。如果某次测试的数据偏差太大，那么会被踢出，样本数相应的减少一个。这里我们都执行的5次测试都可信。执行次数多会减少测试的偏差，所以一般最好测试20次以上。

第十章 Example

当你浏览Go标准库的文档的时候，你经常会看到一些类型或者函数、方法的使用的例子，比如包io下有这么多的例子:

点击一个例子还可以查看例子的代码，甚至可以运行它:

这其实是Go语言的Example,Example甚至可以当Test运行，比如

不过Example最重要的功能是给开发者提供示例。Example也是放在以`_test.go`后缀的文件中，为了区分，你可以放在`example_xxx_test.go`中，也可以和单元测试放在一起。

Example函数签名如`func ExampleXxx()`,以Example开头，没有参数。

一个Example如下:

```go
package io_test

import (
    "bytes"
    "fmt"
    "io"
    "log"
    "os"
    "strings"
)

func ExampleCopy() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }

    // Output:
    // some io.Reader stream to be read
}

func ExampleCopyBuffer() {
    r1 := strings.NewReader("first reader\n")
    r2 := strings.NewReader("second reader\n")
    buf := make([]byte, 8)

    // buf is used here...
    if _, err := io.CopyBuffer(os.Stdout, r1, buf); err != nil {
        log.Fatal(err)
    }

    // ... reused here also. No need to allocate an extra buffer.
    if _, err := io.CopyBuffer(os.Stdout, r2, buf); err != nil {
        log.Fatal(err)
    }

    // Output:
    // first reader
    // second reader
}

func ExampleCopyN() {
    r := strings.NewReader("some io.Reader stream to be read")

    if _, err := io.CopyN(os.Stdout, r, 4); err != nil {
        log.Fatal(err)
```

```
    }

    // Output:
    // some
}

func ExampleReadAtLeast() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    buf := make([]byte, 14)
    if _, err := io.ReadAtLeast(r, buf, 4); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s\n", buf)

    // buffer smaller than minimal read size.
    shortBuf := make([]byte, 3)
    if _, err := io.ReadAtLeast(r, shortBuf, 4); err != nil {
        fmt.Println("error:", err)
    }

    // minimal read size bigger than io.Reader stream
    longBuf := make([]byte, 64)
    if _, err := io.ReadAtLeast(r, longBuf, 64); err != nil {
        fmt.Println("error:", err)
    }

    // Output:
    // some io.Reader
    // error: short buffer
    // error: unexpected EOF
}

func ExampleReadFull() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    buf := make([]byte, 4)
    if _, err := io.ReadFull(r, buf); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s\n", buf)

    // minimal read size bigger than io.Reader stream
    longBuf := make([]byte, 64)
    if _, err := io.ReadFull(r, longBuf); err != nil {
        fmt.Println("error:", err)
    }
```

```
    // Output:
    // some
    // error: unexpected EOF
}

func ExampleWriteString() {
    if _, err := io.WriteString(os.Stdout, "Hello World"); err != nil {
        log.Fatal(err)
    }

    // Output: Hello World
}

func ExampleLimitReader() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    lr := io.LimitReader(r, 4)

    if _, err := io.Copy(os.Stdout, lr); err != nil {
        log.Fatal(err)
    }

    // Output:
    // some
}

func ExampleMultiReader() {
    r1 := strings.NewReader("first reader ")
    r2 := strings.NewReader("second reader ")
    r3 := strings.NewReader("third reader\n")
    r := io.MultiReader(r1, r2, r3)

    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }

    // Output:
    // first reader second reader third reader
}

func ExampleTeeReader() {
    var r io.Reader = strings.NewReader("some io.Reader stream to be read\n")

    r = io.TeeReader(r, os.Stdout)

    // Everything read from r will be copied to stdout.
    if _, err := io.ReadAll(r); err != nil {
        log.Fatal(err)
    }
```

```
    // Output:
    // some io.Reader stream to be read
}

func ExampleSectionReader() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    if _, err := io.Copy(os.Stdout, s); err != nil {
        log.Fatal(err)
    }

    // Output:
    // io.Reader stream
}

func ExampleSectionReader_Read() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    buf := make([]byte, 9)
    if _, err := s.Read(buf); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s\n", buf)

    // Output:
    // io.Reader
}

func ExampleSectionReader_ReadAt() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    buf := make([]byte, 6)
    if _, err := s.ReadAt(buf, 10); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s\n", buf)

    // Output:
    // stream
}

func ExampleSectionReader_Seek() {
```

```
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    if _, err := s.Seek(10, io.SeekStart); err != nil {
        log.Fatal(err)
    }

    if _, err := io.Copy(os.Stdout, s); err != nil {
        log.Fatal(err)
    }

    // Output:
    // stream
}

func ExampleSectionReader_Size() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    fmt.Println(s.Size())

    // Output:
    // 17
}

func ExampleSeeker_Seek() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    r.Seek(5, io.SeekStart) // move to the 5th char from the start
    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }

    r.Seek(-5, io.SeekEnd)
    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }

    // Output:
    // io.Reader stream to be read
    // read
}

func ExampleMultiWriter() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    var buf1, buf2 bytes.Buffer
    w := io.MultiWriter(&buf1, &buf2)
```

```
    if _, err := io.Copy(w, r); err != nil {
        log.Fatal(err)
    }

    fmt.Print(buf1.String())
    fmt.Print(buf2.String())

    // Output:
    // some io.Reader stream to be read
    // some io.Reader stream to be read
}

func ExamplePipe() {
    r, w := io.Pipe()

    go func() {
        fmt.Fprint(w, "some io.Reader stream to be read\n")
        w.Close()
    }()

    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }

    // Output:
    // some io.Reader stream to be read
}

func ExampleReadAll() {
    r := strings.NewReader("Go is a general-purpose language designed with
systems programming in mind.")

    b, err := io.ReadAll(r)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s", b)

    // Output:
    // Go is a general-purpose language designed with systems programming in
mind.
}
```

你可以看到一些Example中包含一样`// Output:`，后面跟着一些文本。当Example当Test运行时，它的输出会和Output下面的行数做比较，如果不等测试就失败。

还可以匹配未排序的行:

```
func ExamplePerm() {
    for _, value := range Perm(5) {
        fmt.Println(value)
    }
    // Unordered output: 4
    // 2
    // 1
    // 3
    // 0
}
```

一般来说Example命令如下:

```
func Example() { ... } //package例子
func ExampleF() { ... }  // 函数
func ExampleT() { ... } // 类型
func ExampleT_M() { ... } // 类型的方法
```

如果例子比较多，还可以加后缀:

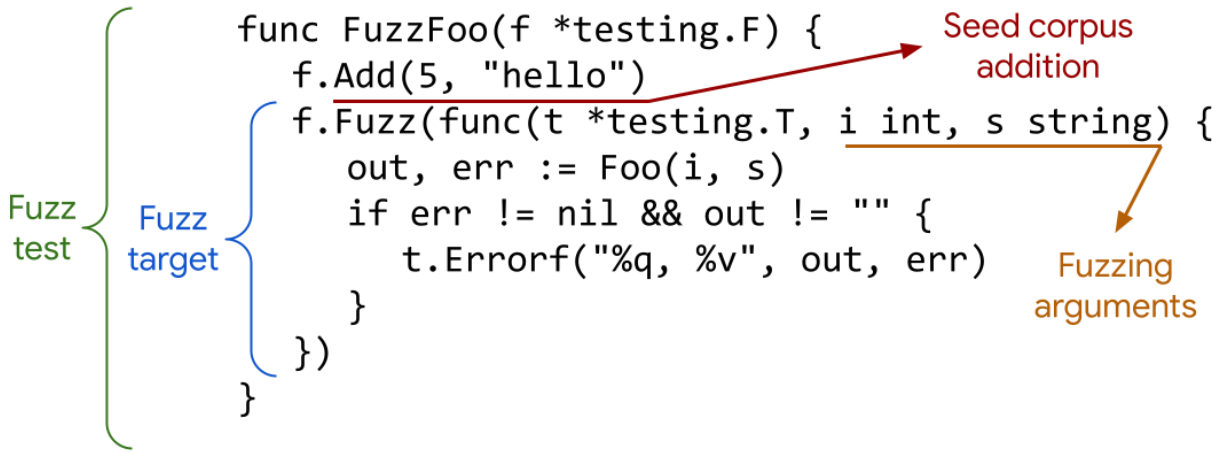
```
func Example_suffix() { ... }
func ExampleF_suffix() { ... }
func ExampleT_suffix() { ... }
func ExampleT_M_suffix() { ... }
```

比如代码库包s9中，有一个Add的示例，运行godoc(go install golang.org/x/tools/cmd/godoc@latest, godoc -http :8080)可以查看go doc的显示:

第十一章 模糊测试

模糊测试是Go 1.18新加的一个测试类型。实际上自Go 1.5, Go代码内部就开始使用dvyukov/go-fuzz发现了标准库和运行时的200多个bug，dvyukov/go-fuzz，数量还是相当惊人的，go-fuzz工具虽好,但是毕竟还是属于第三方的工具，不能很好的和Go语言自身结合，所以在Go 1.18中Fuzz作为一个语言特性得到了支持。

想学习Go Fuzz技术，我建议你先读一读Go Fuzzing 文档。



你应该知道，即使代码覆盖率是100%,也不能说我们的代码都得到了测试，因为各种分支、边界以及它们的组合我们很多情况下未能全面覆盖。一是有时候我们想不到所有的场景，而是即使想到所有的场景，场景太多我们也不会为这些场景写单元测试，这会导致有些场景未被真正测试，留有风险。

模糊测试专门处理这样的问题，它会根据种子语料库以及代码覆盖率和测试的情况，智能探索测试的场景，以期尽量覆盖所有的场景。

本文就不分析它是如何智能的准备输入，覆盖全面的场景的，而是给大家介绍如何编写模糊测试。

模糊测试必须遵循下面的规则：

- 模糊测试函数名必须是`FuzzXxx`,参数是`*testing.F`，没返回值,如`func FuzzXxx(f *testing.F)`
- 模糊测试必须放在以`*_test.go`后缀的文件中
- fuzz target 必须是调用`[(*testing.F).Fuzz](https://pkg.go.dev/testing#F.Fuzz)`,第一个参数`*testing.T`,后面是模糊测试的参数
- 一个模糊测试只能有一个fuzz target
- 种子语料(seed corpus)的数据类型必须和 fuzzing arguments一样，顺序一致。包括调用`[(*testing.F).Add](https://pkg.go.dev/testing#F.Add)`增加种子语料，以及写在testdata/fuzz文件夹下语料
- fuzzing arguments 只能是下面的类型。如果你要测试复杂的struct,需要使用这些参数组装你的struct,然后进一步测试:
  - `string`, `[]byte`
  - `int`, `int8`, `int16`, `int32` / `rune`, `int64`
  - `uint`, `uint8` / `byte`, `uint16`, `uint32`, `uint64`
  - `float32`, `float64`
  - `bool`

fuzz target是并发执行的，所以不要依赖全局性的变量等。fuzz target应该简单且快、准确，方便fuzz引擎更高效。

有两种情况可以运行模糊测试

- 当做普通的单元测试: 运行 `go test`，使用每一条种子语料进行测试，报告结果
- 当做模糊测试: 运行 `go test -fuzz=FuzzTestName`,此包下的单元测试和benchmark会先执行，全部通过才会进行模糊测试，执行一个模糊测试

默认情况下，模糊测试会一直运行，直到遇到错误，它会把错误的语料写入到 `testdata/fuzz/Xxx` 下，以便你分析和将来使用它回归测试。如果你想设置模糊测试的时间，你可以使用 `-fuzztime t`， `t` 可以是时间，比如 `1h30m`,也可以是次数，比如 `1000x` 。

`-fuzzminimizetime t` 设置最少的模糊测试时间。

比如下面这个例子，我们实现了一个普通的单元测试，一个模糊测试， 测试 `Reverse` 函数，这个函数用来翻转字符串:

```go
package s10

import (
    "testing"
    "unicode/utf8"
)

func TestReverse(t *testing.T) {
    testcases := []struct {
        in, want string
    }{
        {"Hello, world", "dlrow ,olleH"},
        {" ", " "},
        {"!12345", "54321!"},
    }
    for _, tc := range testcases {
        rev := Reverse(tc.in)
        if rev != tc.want {
            t.Errorf("Reverse: %q, want %q", rev, tc.want)
        }
    }
}

func FuzzReverse(f *testing.F) {
    testcases := []string{"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc) // Use f.Add to provide a seed corpus
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev := Reverse(orig)
        doubleRev := Reverse(rev)
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
}
```

`Reverse`函数的实现如下,通过翻转byte slice实现(当然我们知道这个实现是有问题的，因为字符串的单位是rune,而不是byte)：

```
func Reverse(s string) string {
    b := []byte(s)
    for i, j := 0, len(b)-1; i < len(b)/2; i, j = i+1, j-1 {
        b[i], b[j] = b[j], b[i]
    }
    return string(b)
}
```

运行模糊测试，很容易就发现了问题：

而且我们可以看到造成失败的语料已经被写入了语料文件中：

这个模糊测试在收集基线数据的时候就发现错误了，还没有施展拳脚呢。正常情况输出应该如下所示：

- elapsed: 自测试开始已过的时间
- exec: 已测试的输入的次数
- interesting: 推算出来的新加入的语料数

啥时候一个模糊测试算失败呢?

- panic发生
- fuzz target调用了`t.Fail`, 或者 `t.Error` 和 `t.Fatal`.
- 不可恢复的错误发生，比如 `os.Exit` 或者栈溢出
- fuzz target花费了太长的时间去完成， 当前一次fuzz target最大设置为1秒。如果因为死循环等不期望的行为超过了1秒，也会失败

你可以通过代码增加语料，比如`f.Add([]byte("hello\xbd\xb2=\xbc ⌘"), int64(572293))`，它的语料有两个参数，第一个参数类型是`[]byte`,第二个是`int64`,所以fuzz target是`f.Fuzz(func(*testing.T, []byte, int64) {})`。你也可以在语料文件中增加测试，格式和参数保持一致即可，比如:

```
go test fuzz v1
[]byte("hello\\xbd\\xb2=\\xbc ⌘")
int64(572293)
```