GeekBrains

Лучшие практики разработки Go-приложений

Продвинутые практики тестирования

На этом уроке

1. Научимся использовать самую популярную библиотеку для тестирования testify
2. Научимся создавать заглушки для компонентов, которые игнорируются в конкретном сценарии
3. Научимся подготавливать общие для разных тестов объекты с помощью фикстур
4. Научимся писать интеграционные тесты
5. Научимся писать код таким образом, чтобы написание тестов занимало минимум времени

Оглавление

Теория урока

Библиотека stretchr/testify

Для более лаконичного и читаемого описания тестовых условий очень часто разработчики используют пакет stretchr/testify. Если вы перешли в Go из другого языка, то наверняка вас смутило отсутствие большого числа вспомогательных функций для тестирования, как то проверка на вхождение элемента в массив, проверка на равенство сложных структур или типов данных, да даже описание ожидаемого результата происходит путем описания условия через if-выражение.

В пакете testify все методы проверки разделены на два вида: assertions и requirements. Собственно, assertions - это проверки заданных условий теста, при которых выполнение теста не прекращается

даже если проверка не была пройдена, requirements - наоборот, то, без чего продолжение выполнения теста не имеет смысла.

После установки в проект "github.com/stretchr/testify/assert" нам становится доступно огромное число вспомогательных функций для тестирования. Выделю те, которые сам часто использую:

```
func Equal(t TestingT, expected, actual interface{}, msgAndArgs ...interface{})
bool
```

Вместо составления if-выражения можем просто проверить, что ожидаемое значение соответствует возвращенному.

```
func NotNil(t TestingT, object interface{}, msgAndArgs ...interface{}) bool
```

Удобно проверять, что возвращенный объект был проинициализирован в функции, например, косвенно указывает на отсутствие ошибки.

```
func NoError(t TestingT, err error, msgAndArgs ...interface{}) bool
```

Позволяет удостовериться, что функция для заданных параметров не вернула ошибки. В отличие от Nil() функции выведет текст ошибки, что удобно при отладке.

```
func True(t TestingT, value bool, msgAndArgs ...interface{}) bool
```

Простая проверка предиката на истинность.

```
func Contains(t TestingT, s, contains interface{}, msgAndArgs ...interface{})
bool
```

Вместо цикла по массиву или ключам хэш-таблицы можно воспользоваться этой функцией для проверки наличия элемента в объекте, по которому можно итерироваться.

```
func JSONEq(t TestingT, expected string, actual string, msgAndArgs
...interface{}) bool
```

Очень удобная функция для проверки двух строк, содержащих JSON-объекты. Поскольку порядок полей в одинаковых JSON может и не совпадать, проверка на равенство подразумевает проверку совпадения всех полей по-отдельности, JSONEq облегчает эту проверку.

Кроме вышеописанных можно использовать более сложные проверки, например, наличия объектов в файловой системе (DirExists, FileExists). Все это позволяет сделать код теста более лаконичным и читаемым, а также не переизобретать test helpers.

Все эти и другие функции доступны в обоих пакетах - require и assert.

Стабы

Есть ситуации, когда тестируемая функция зависит от объектов, которые

- мы не хотим тестировать в данном тестовом сценарии
- муторно инициализировать
- используются данной функцией лишь частично (например, используется лишь один или несколько методов интерфейса, предоставляемого объектом)

В таких случаях мы хотим воспользоваться заглушкой, в которой опишем необходимое для тестового сценария поведение, а сам объект создавать не будем. Стаб - как раз такая заглушка.

Для имитации поведения объекта необходимо реализовать идентичный интерфейс, а это значит, что если функция до этого принимала объект конкретного типа, его следует заменить интерфейсом.

Например, у нас есть функция по подсчету различных слов в файле со следующей сигнатурой

```
func countWords(f *os.File) map[string]int64
```

Для ее тестирования нам пришлось бы создавать файлы с разным содержанием. Вместо этого мы можем заменить тип аргумента на интерфейс io.Reader, а затем воспользоваться объектом с тем же интерфейсом, но который проинициализировать для нас проще

```
func countWords(rdr io.Reader) map[string]int64
```

```
func TestCountWords(t *testing.T)  {
  content := []byte("alice bob alice")
  buf := bytes.NewBuffer(content)
  assert.Equal(t, map[string]int64{"alice":2, "bob":1}, countWords(buf))
}
```

В качестве более общего примера рассмотрим следующий:

```
type Bar struct {
  // объявление большого числа атрибутов
}

func (b *Bar) Result() int {
  return b.preprocess()
}

func (b *Bar) preprocess() int {
  // сложная логика
```

```
}

func foo(b *Bar) int {
  return b.Result() + 42
}
```

Здесь функция foo зависит от указателя на объект Bar, который инициализировать в тесте выглядит изрядным переусложнением, кроме того тестируемая функция зависит лишь от одного метода, внутренние детали реализации которого хотелось бы оставить за границами теста. Заменим на интерфейс и переопределим поведение метода Result для стаба прямо в тесте:

```
type Bar interface {
  Result() int
}
```

Объявляем интерфейс.

```
type BarImpl struct {
  // объявление большого числа атрибутов
}

func (b *BarImpl) Result() int {
  return b.preprocess()
}

func (b *BarImpl) preprocess() int {
  // сложная логика
}
```

Оставляем как есть реализацию.

```
type BarStub struct {
  result func() int
}

func (bs *BarStub) Result() int {
  return bs.result()
}

func foo(b Bar) int {
  return b.Result() + 42
}
```

Подготавливаем стаб.

Теперь тест можно записать, определив прямо в его теле функцию Result:

```
func TestFoo(t *testing.T)  {
  stub := &BarStub{func() int {
     return 42
  }}
  assert.Equal(t, 84, foo(stub))
}
```

Моки

Предположим, в примере выше наша функция Result, от которой зависит тестируемая foo принимала бы еще какое-то значение. Тогда тестируя функцию foo, мы хотели бы удостовериться, что в Result передается правильное значение. Как бы мы могли это проверить?

Например, выполнив в функцие Result() нашей заглушки проверку на принимаемое значение. Так мы получим мок, фактически, то же самое, что стаб, но с дополнительными проверками на уровне имитирующего объекта.

Для генерации и стабов, и моков часто используют библиотеки, которые при помощи кодогенерации либо рефлексии создают заглушку, удовлетворяющую заданному интерфейсу. Рассмотрим одну из наиболее популярных реализаций такой утилиты - mockery.

Рассмотрим такой калькулятор заказов

```
type DB interface {
  GetUserName(uid uint64) (string, error)
  GetOrderItems(oid uint64) ([]uint64, error)
}

type Calculator struct {
  db DB
}

func (c *Calculator) ProcessOrder(userId, orderId uint64) (string, error) {
  name, err := c.db.GetUserName(userId)
  if err != nil {
     return "", errors.Wrap(err, "get user name from db")
  }

  bought, err := c.db.GetOrderItems(orderId)
  if err != nil {
     return "", errors.Wrap(err, "get order items from db")
  }

  return fmt.Sprintf("user %s spent %d", name, sum(bought)), nil
}
```

Метод ProcessOrder принимает идентификатор пользователя и заказа и пытается вернуть информацию о заказе, которая содержит имя пользователя и сумму потраченных на заказ денег. В процессе выполнения мы делаем два запроса в базу данных. Поскольку мы не хотим задействовать реальную базу данных в тесте, можем сделать для нее заглушку с помощью mockery:

```
$ go get github.com/vektra/mockery/v2/.../  # устанавливаем
$ mockery --name DB  # запускаем генерацию заглушки для интерфейса DB
```

В результате будет сгенерирована заглушка в файле mocks/DB.go. Рассмотрим, как ее можно использовать для тестирования метода ProcessOrder:

```
func TestCalculator_ProcessOrder(t *testing.T) {
  dbMock := &mocks.DB{}
  calc := &Calculator{db: dbMock}

  // Проверяем  аргументы и заодно явно специфицируем возвращаемые значения
  dbMock.On("GetUserName", uint64(42)).Return("Bob", nil)
  dbMock.On("GetOrderItems", uint64(100500)).Return([]uint64{100, 200, 250},
nil)

  res, err := calc.ProcessOrder(42, 100500)
  require.NoError(t, err)
  assert.Equal(t, "user Bob spent $550", res)
}
```

Если в логике теста не предполагается проверка одного или нескольких аргументов вызываемого метода, такие аргументы можно заменить на mock.Anything.

```
func TestCalculator_ProcessOrder(t *testing.T) {
  dbMock := &mocks.DB{}
  calc := &Calculator{db: dbMock}

  // Проверяем  аргументы и заодно явно специфицируем возвращаемые значения
  dbMock.On("GetUserName", mock.Anything).Return("Bob", nil)
  dbMock.On("GetOrderItems", mock.Anything).Return([]uint64{100, 200, 250}, nil)

  res, err := calc.ProcessOrder(42, 100500)
  require.NoError(t, err)
  assert.Equal(t, "user Bob spent $550", res)
}
```

Старайтесь использовать только в том случае, когда действительно не знаете, что будет передано в качестве аргумента - не избегайте дополнительных проверок.

Какие недостатки у использования моков на основе рефлексии? При изменении сигнатуры метода тест сломается после запуска. Если есть возможность не допустить ошибки еще на этапе компиляции,

почему бы этим не воспользоваться? Так gojuno/minimock заранее генерирует тип мока со всеми нужными методами, а также вспомогательными методами специально для тестирования.

```
go install github.com/gojuno/minimock/v3/cmd/minimock  # устанаваливаем

minimock -i DB  # запускаем генерацию заглушки для интерфейса DB
```

В результате будет сгенерирована заглушка в файле db_mock_test.go. Перепишем тест для использования minimock:

```
func TestCalculator_ProcessOrder(t *testing.T)  {
  dbMock := &DBMock{}
  calc := &Calculator{db: dbMock}

  dbMock.GetUserNameMock.Expect(42).Return("Bob", nil)
  dbMock.GetOrderItemsMock.Expect(100500).Return([]uint64{100, 200, 250}, nil)

  res, err := calc.ProcessOrder(42, 100500)
  require.NoError(t, err)
  assert.Equal(t, "user Bob spent $550", res)
}
```

Если сигнатуры методов, которые мы замокали изменятся, наш тест не сможет скомпилироваться, мы заранее узнаем, где нам следует его поправить.

Какой недостаток у такого подхода? Необходимость добавить шаг генерации мока перед непосредственным запуском теста.

Чаще всего в проектах я вижу использование mockery/testify.mock, но если вы начинаете проект с нуля, я бы предложил использовать minimock,

Фикстуры

Часто при написании тестов встречается необходимость выносить общую логику по установке начального состояния системы. Для удобства придумали fixtures. В библиотеке testify есть специальный пакет suite, который, во-первых, позволяет задать команды по подготовке теста в

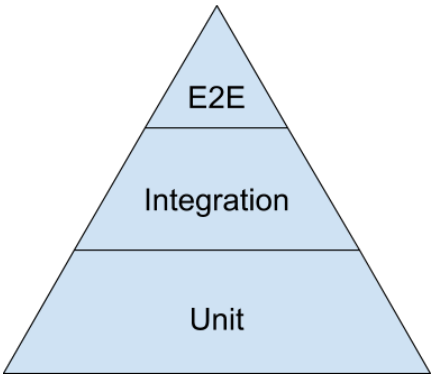
функции с префиксом Setup, а во-вторых, объединяет набор тестов с общей логикой подготовки в один так называемый test suite.

Рассмотрим пример тестирования LRU кэша. Если мы хотим, чтобы все тесты гонялись с одинаковыми исходными данными, мы можем подготовить их с помощью Setup функции

```
type CacheLRUTestSuite struct {
  suite.Suite
  cache CacheLRU
}

func (s *CacheLRUTestSuite) SetupTest() {
  s.cache = NewCacheLRU(5)
  s.cache.Set("foo", 5)
  s.cache.Set("bar", 10)
}

func (s *CacheLRUTestSuite) TestSet() {
  s.T().Run("new key, enough space in cache", func(t *testing.T) {
     s.Require().Nil(s.cache.Set("zoo", 100))
     s.Assert().Equal("zoo", s.cache.MostRecent())
  })

  s.T().Run("new key, not enough space in cache", func(t *testing.T) {
     s.Require().Nil(s.cache.Set("buz", 7))
     s.Assert().Equal("buz", s.cache.MostRecent())
     _, err := s.cache.Get("foo")
     s.Assert().Error(err)
  })
}

func (s *CacheLRUTestSuite) TestGet() {
  s.Require().Equal("foo", s.cache.LeastRecent())
  item, err := s.cache.Get("foo")
  s.Require().NoError(err)
  s.Assert().Equal(5, item)
  s.Assert().Equal("bar", s.cache.LeastRecent())
}

func TestCacheLRUTestSuite(t *testing.T) {
  suite.Run(t, new(CacheLRUTestSuite))
}
```

Интеграционные тесты

Вспомним пирамиду тестирования



Мы уже умеем писать юнит-тесты, теперь необходимо вспомнить, что тестирование проекта не будет полным, если мы тестируем наш код в вакууме, не задействуя внешние модули. Допустим, наш проект взаимодействует с базой данных или с внешней программой. Когда мы писали юнит-тесты, мы старались мокать базу данных и внешние процессы, позволяя таким образом не запускать на каждый тест бд и не запускать внешний процесс. Это позволяло тестам прогоняться за короткое время, а нам - не задумываться над подготовкой внешних зависимостей (например, мы не заботились о накатывании миграций на базу или корректном завершении внешнего процесса). Тем не менее для того, чтобы убедиться, что наш модуль корректно взаимодействует с внешними, необходимо написать интеграционные тесты.

Интеграционные тесты не должны тестировать внутреннюю реализацию нашего модуля - ее мы покрыли юнит-тестами - они должны тестировать саму интеграцию.

Предположим, наш модуль представляет из себя сервис, который хранит информацию о пользователях в базе данных. Всю логику работы сервиса мы покрыли юнит-тестами путем выноса интерфейса базы данных в отдельный тип *DB* и заменив обращения к базе данных вызовами методов мока/стаба *DBMock*. Теперь нам необходимо убедиться, что сама интеграция с базой корректно реализована, а значит необходимо протестировать сами методы имплементации интерфейса *DBImpl.*

```
// +build integration

package main

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

func TestGetUsers(t *testing.T)  {
  dsn := os.Getenv("DB_TEST_DSN")
  require.NotEmpty(t, dsn)

  db := &DBImpl{}
  insertUser(t, 42, "Bob")
```

```
  assert.Equal(t, []int64{42}, db.GetUsers())
}

func insertUser(t *testing.T, id int64, name string) {
  t.Helper()
  // sql insert query
}
```

В рамках этого примера обратим внимание на следующие особенности:

1. Комментарий на первой строке файла указывает, что файл будет скомпилирован только при указании аргумента -integration компилятору. Это позволяет игнорировать интеграционные тесты в случае, если их прогон занимает продолжительное время или если мы запускаем их в окружении, где отсутствуют необходимые компоненты (база данных)
2. Вызов t.Helper() позволяет проигнорировать печать номера строки в этой функции.
3. С помощью переменной окружения можно подставить базу данных, на которой мы хотим проводить тестирование.

Тестируемый код

Написание тестов - это всегда дополнительная работа, которая занимает значительный процент времени разработки. Чтобы сократить время на написание тестов можно писать меньше тестов (нельзя) или писать изначально код таким образом, чтобы на написание тестов уходило меньше времени. Если вспомнить, на что именно уходит больше всего времени при написании теста, можно выделить два пункта:

- рефакторинг кода из-за того, что код в текущем виде не может быть эффективно протестирован
- прединициализация зависимостей тестируемого кода.

Некоторые принципы тестируемого кода мы уже выявили, когда говорили о написании моков, о другие поговорим впервые.

Используем интерфейсы

При создании заглушек было очевидно, что гораздо проще их составление или генерация в коде, где зависимости уже представлены в виде интерфейсов. Если еще на этапе написания кода представлять, как мы его будем тестировать, то скорее всего мы будем использовать меньше конкретных типов и больше интерфейсов. Так при определении атрибута, от которого зависит наш код, стоит думать не о типе объекта, а о действиях (методах), которые будут над ним производиться.

Избавляемся от глобального состояния

У использования глобальных переменных много недостатков, и затруднение тестирования - одно из них.

Кроме того, следует избегать конфигурирования системы через константы. Если константы задают параметры системы в продакшене, и наш код их задействует, то мы не сможем подставить параметры окружения, в котором хотим проводить тестирование. Это относится к сетевым настройкам веб-серверов, к путям до файлов и директорий, названиям и параметрам установки соединения с базой данных. Если есть необходимость задать параметры по-умолчанию прямо в коде, то следует предусмотреть отдельную структуру, которую мы будем инициализировать константами, однако иметь возможность их переопределить.

SRP позволяет проще писать тесты

Single Responsibility Principle - один из столпов набора принципов SOLID. Им не стоит злоупотреблять, но и полное игнорирование приводит к сложностям как в разработке системы, так и ее тестировании. Принцип утверждает, что каждый объект должен отвечать за одну конкретную функциональность. На практике несоблюдение этого принципа приводит к невозможности изменения части функциональности без изменения всех ее зависимостей. Если при тестировании одной функции вам необходимо проинициализировать всю систему целиком, скорее всего код нуждается в разделении на компоненты, каждый из которых можно протестировать гораздо меньшими усилиями.

Чистые функции

Чистота функции характеризуется

- Невозможностью функции возвращать различные результаты при подаче одинаковых входных значений
- Отсутствием побочных эффектов (модификация глобального состояния, модификация аргументов переданных в функцию, операции ввода-вывода и др.)

Домашнее задание

Доработать тесты в программе по поиску дубликатов файлов

1. Рефакторим код в соответствие с рассмотренными принципами (SRP, чистые функции, интерфейсы, убираем глобальный стэйт)
2. Пробуем использовать testify

3. Делаем стаб/мок (например, для файловой системы) и тестируем свой код без обращений к внешним компонентам (файловой системе)
4. Делаем отдельно 1-2 интеграционных теста, запускаемых с флагом -integration

Глоссарий

Стаб - заглушка объекта, имитирующая методы реального объекта, необходима для передачи нужных значений в функцию.

Мок - как стаб, но проверяет входные параметры при вызове своих методов.

Фикстура - обвязка над набором тестов, служащая для установки начального состояния системы перед прогоном самого теста.

SRP - Single Responsibility Principle, один из принципов SOLID, означает необходимость каждому объекту иметь не более чем одну ответственность.

Чистая функция - детерминированная функция без побочных эффектов.

Дополнительные материалы

1. GopherCon 2017: Mitchell Hashimoto - Advanced Testing with Go
2. Введение в тестирование в Go от Алексея Махова