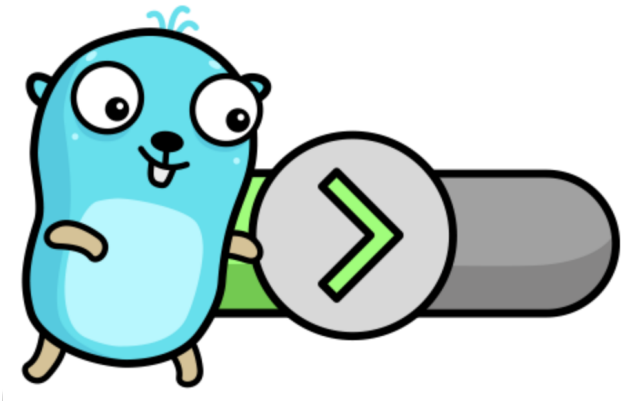
Практика по боевому применению defer

В этом уроке мы предлагаем вам решить ряд задач, посвящённых наиболее популярным техниками и примерам работы с `defer`.

Поэтому, решая очередную задачу, постарайтесь найти место для `defer` или, наоборот, задумайтесь, а нужен ли он в данном конкретном случае?

Также обращаем ваше внимание на тесты к задачам – нередко из них можно почерпнуть опыта не меньше, чем из самих задач, в том числе и по использованию `defer`.



Задача "Timekeeper"

Ссылка на заготовку.

Предлагаем вам решить задачу, которую часто можно встретить на собеседованиях.

Но у нас она представлена в более production форме.

Перед вами тип `Timekeeper`, позволяющий записывать время операции в хранилище метрик:

```
type MetricsStorage interface {
    Record(operation string, d time.Duration)
}

type Timekeeper struct {
    storage MetricsStorage

}
```

Это происходит в методе `(*TimeKeeper).MeasureExecutionTime`, который необходимо реализовать и использование которого выглядит следующим образом:

```
func ExampleTimekeeper() {
    s := newMetricsStorage()
    t := NewTimekeeper(s)

    defer t.MeasureExecutionTime("ExampleTimekeeper")()
    time.Sleep(time.Second)
}

// При выходе из функции ExampleTimekeeper в методе
MeasureExecutionTime
// происходит вызов метода Record хранилища метрик и в базе
появляется запись о том,

// что операция "ExampleTimekeeper" заняла 1s (+ некоторая
погрешность).
```

В данной задаче вам необходимо самим догадаться о **сигнатуре метода**:

```
defer t.MeasureExecutionTime("ExampleTimekeeper")() // Обратите внимание на двойной вызов ;)
```

Больше подробностей см. в заготовке задачи и тестах.

Удачи!

Задача "SMS repository"

[Ссылка на заготовку](#).

Вам необходимо реализовать хранилище статусов SMS-сообщений:

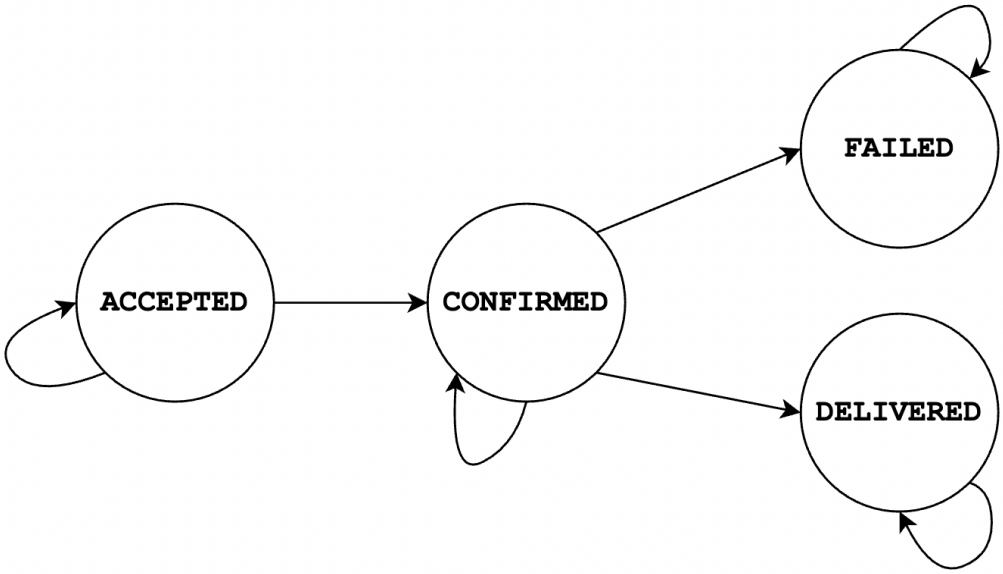
```
type (
    MessageID     string
    MessageStatus string
)

const (
    // MessageStatusAccepted - сообщение принято внешней системой.
    MessageStatusAccepted = MessageStatus("ACCEPTED")

    // MessageStatusConfirmed - сообщение подтверждено к отправке внешней системой.
```

```
    MessageStatusConfirmed = MessageStatus("CONFIRMED")

    // MessageStatusFailed - сообщение не доставлено по какой-то
причине.
    MessageStatusFailed = MessageStatus("FAILED")

    // MessageStatusDelivered - сообщение доставлено.
    MessageStatusDelivered = MessageStatus("DELIVERED")
)

// Repo представляет собой конкурентнобезопасное
// in-memory хранилище статусов SMS-сообщений.
type Repo struct {
    db   map[MessageID]MessageStatus
    lock sync.RWMutex

}
```

Мы можем:

- Сохранять сообщение в хранилище с помощью метода `(*Repo).Save`;
- Получать сообщение из хранилища с помощью метода `(*Repo).Get`;
- И обновлять статус сообщения с помощью метода `(*Repo).Update`.

При этом граф переходов состояний для `MessageStatus` выглядит следующим образом:



Больше подробностей см. в заготовке задачи и тестах.

Не забудьте прогнать своё решение через race detector:

```
$ cd tasks/02-defer-statement/sms-repository
$ go test -race .
```

```
ok      tasks/02-defer-statement/sms-repository      1.294s
```

Задача "ZIP archive"

Ссылка на заготовку.

Вам необходимо реализовать функцию `Archive`:

```
var ErrNothingToArchive = errors.New("nothing to archive")

// Archive объединяет файлы по путям inPaths в один ZIP-архив по пути
outPath.
// Если список входящих путей пуст, то возвращает ошибку
ErrNothingToArchive.

func Archive(fSys FS, outPath string, inPaths ...string) error
```

Пример работы функции представлен в `TestArchive_SmokeWithRealFS`:

```
func TestArchive_SmokeWithRealFS(t *testing.T) {
    // ...
    err := Archive(fSys, "proverbs.zip",
        "testdata/1.txt",
        "testdata/2.txt",
        "testdata/3.txt",
    )
    require.NoError(t, err)
    // ...

}
```

```
$ cd tasks/02-defer-statement/zip-archive
```

```
$ go test -run=TestArchive_SmokeWithRealFS .
ok      tasks/02-defer-statement/zip-archive 0.178s

$ ls -lah testdata
total 24
drwxr-xr-x  5 anthony  staff   160B Jan 30 15:38 .
drwxr-xr-x  6 anthony  staff   192B Jan 30 20:38 ..
-rw-r--r--  1 anthony  staff   249B Jan 30 15:36 1.txt
-rw-r--r--  1 anthony  staff   279B Jan 30 15:37 2.txt
-rw-r--r--  1 anthony  staff   233B Jan 30 15:38 3.txt

$ unzip -l proverbs.zip
Archive:  proverbs.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
      249  01-30-2022 15:36   testdata/1.txt
      279  01-30-2022 15:37   testdata/2.txt
      233  01-30-2022 15:38   testdata/3.txt
---------                     -------
      761                     3 files
```

Обращаем внимание, что файлы в архиве имеют то же время модификации, что и оригиналы.

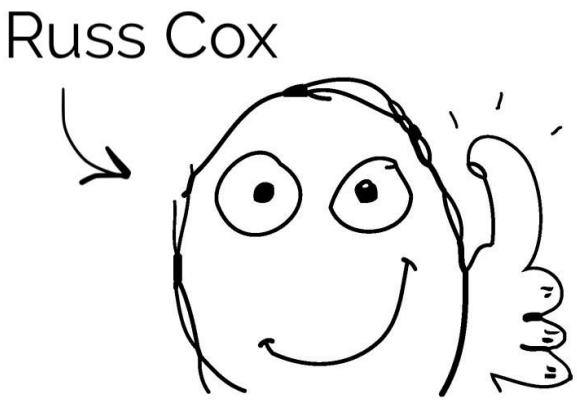
Вам пригодятся пакеты **archive/zip**, **io** и **fmt**.

Мы не стали тестировать все возможные ветви внутри функции, оставив реализацию на вашу совесть.

Главное – это получить удовольствие от процесса работы 🙂

Задача "errd.Wrap"

Ссылка на заготовку.



Вам необходимо реализовать функцию `Wrap`.

Чтобы понять, что она делает, взглянем на пример использования функции:

```
func ExampleWrap() {
    err := Copy("unknown.txt", "known.txt")
    fmt.Println(err)

    // Output:
    // copy "unknown.txt" to "known.txt": open src file: open
unknown.txt: no such file or directory
}

func Copy(src, dst string) error {
    return copyFile(src, dst)
}

func copyFile(src, dst string) (err error) {
    defer Wrap(&err, "copy %q to %q", src, dst) // Добавляем контекст
возвращаемой ошибке.

    // ...

}
```

В данной задаче вам необходимо догадаться о **сигнатуре функции** и требованиях к её реализации по тестам.

Удачи!

P.S. Пишите в комментариях свои варианты документации для `Wrap`, у нас не хватило фантазии 🙂

Задача "Port Scanner"

Ссылка на заготовку.

Вам необходимо реализовать функцию `Scan`:

```
const (
    maxPortNumber = 65535
    workers       = 10
    connTimeout   = 10 * time.Millisecond
)

type Dialer interface {
    DialContext(ctx context.Context, network, address string) (Conn, error)
}

// Scan сканирует hostname на предмет открытых TCP-портов.

func Scan(d Dialer, hostname string) []int
```

Функция конкуретно с помощью диалера `d` пытается создать TCP-соединение по адресам

```
hostname:1
...
hostname:65535
```

и возвращает список портов, для которых это удаётся.

Требования:

- Одновременно может работать не более `workers` горутин (как следствие, не может быть установлено более, чем `workers` соединений).

- Возвращаемый список портов должен быть отсортирован по возрастанию.

- Если `d.DialContext` возвращает ошибку (соединение установить не удаётся), то она игнорируется. При желании неожиданные ошибки можно логировать. Ожидаемыми ошибками являются:
  - `syscall.ECONNREFUSED`;
  - ошибки таймаута, т.е. имеющие метод `Timeout() bool`, возвращающий `true`.
- Если соединение открыть удаётся, то необходимо добавить порт в результирующий список, а само соединение не забыть закрыть. Ошибку от закрытия допустимо игнорировать.

- Предполагается, что если соединение устанавливается дольше, чем `connTimeout` времени, то значит перед нами honeypot – данный порт

следует отбросить, а соединение прервать. В этом вам поможет пакет context.

Больше подробностей см. в заготовке задачи и тестах.

При желании сначала можно реализовать простую синхронную версию функции и проверить её с помощью `TestScan_Example`:

```
$ cd tasks/02-defer-statement/port-scanner

$ go test -run=TestScan_Example -v .
=== RUN   TestScan_Example
    scan_test.go:28: [631 3306 5000 6942 7000 33060 55412 63342]
--- PASS: TestScan_Example (2.27s)
PASS

ok      tasks/02-defer-statement/port-scanner        2.366s
```

Не забудьте прогнать своё решение через race detector:

```
$ cd tasks/02-defer-statement/port-scanner
$ go test -race .

ok      tasks/02-defer-statement/port-scanner        10.007s
```

Тест "defer для внешнего вызова"

Исходник примера.

Разработчик неверно понял принцип работы `defer` и реализовал хелпер `RunInTransaction`:

```
type DB struct{}

// RunInTransaction создаёт транзакцию и коммитит её,
```

```
// когда завершается функция, в которой произошёл вызов
RunInTransaction().
func (d DB) RunInTransaction() {
    d.Begin()
    defer d.Commit()
}

func (d DB) Begin() {
    fmt.Println("BEGIN TRANSACTION")
}

func (d DB) Commit() {
    fmt.Println("COMMIT TRANSACTION")

}
```

Который заиспользовал следующим образом:

```
func Operation(d DB) {
    d.RunInTransaction()
    execQuery(d)
}

func execQuery(_ DB) {
    fmt.Println("EXEC QUERY")
}
```

Но вместо `BEGIN - EXEC - COMMIT`, при запуске он получил:

```
BEGIN TRANSACTION
COMMIT TRANSACTION

EXEC QUERY
```

Пришёл тимлид и исправил метод `RunInTransaction`:

```
func (d DB) RunInTransaction() func() {
    d.Begin()
    return func() { // Эквивалентно `return d.Commit`.
        d.Commit()
    }
```

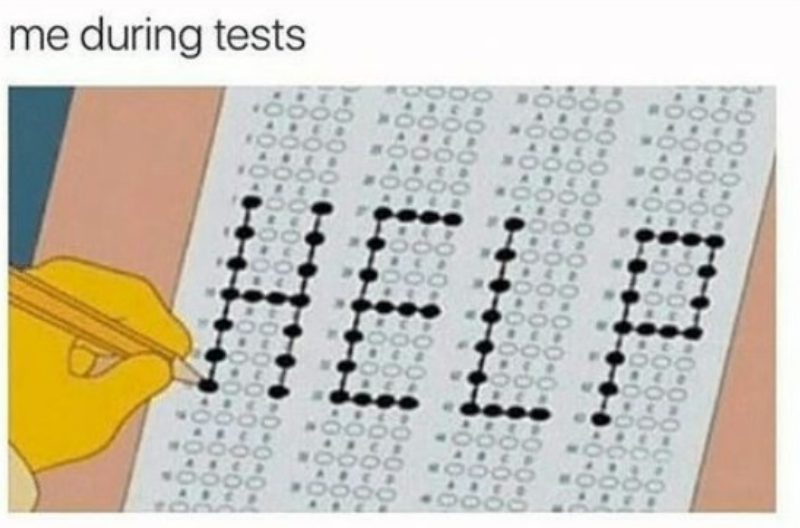
```
}
```

Вам необходимо исправить вызов метода в функции `Operation`.

Заполните пропуски

```
func Operation(d DB) {

    __________

   execQuery(d)

}
```

Тест "defer"



Выберите все подходящие ответы из списка

- ☐ аргументы отложенной через defer функции вычисляются сразу же, а не во время её вызова
- ☐ defer в цикле – это стандартная общепринятая практика
- ☐ аргументы отложенной через defer функции вычисляются в момент её вызова; defer – это синтаксический сахар над замыканием

- [ ] можно откладывать анонимные и именованные функции, а также методы
- [ ] нельзя откладывать анонимные функции
- [ ] стоит задумываться, допустимо ли игнорировать ошибку от отложенной операции
- [ ] defer отрабатывает при выходе из текущей функции
- [ ] defer может влиять на неименованные возвращаемые результаты
- [ ] отказываться от defer из-за его производительности – не самая лучшая идея
- [ ] defer отрабатывает при выходе из текущей области видимости ({}, if, for, func и пр.)
- [ ] любые ошибки отложенных функций следует игнорировать
- [ ] defer отрабатывает до выставление результатов работы функции
- [ ] чтобы defer смог изменить возвращаемое значение, оно должно быть именованным
- [ ] если видишь defer в цикле, то следует насторожиться
- [ ] defer отрабатывает после выставления результатов работы функции
- [ ] в общем случае defer несёт за собой большие накладные расходы, вследствие чего следует избегать его
- [ ] в актуальных версиях языка defer практически бесплатный

Выводы

Мы исколесили тему `defer` вдоль и поперёк. Сложно здесь что-то добавить, кажется, вы уже и так всё знаете 🙂

Так что давайте выдохнем и двинемся дальше.

Так держать!