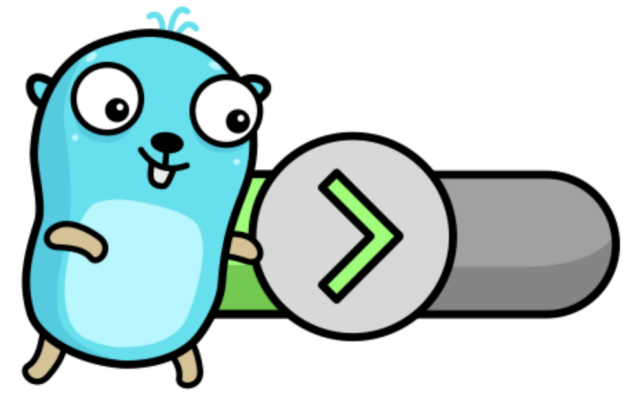
runtime.SetFinalizer

В этом уроке мы немного коснёмся **финализаторов** в Go и функции, которая их назначает.

Эта тема чем-то напоминает `defer`, но это только видимое сходство.



Возвращаясь к истокам

Вспомним одно из решений задачи "**Утечка файловых дескрипторов**", с которой мы столкнулись в самом начале модуля (исходник задачи):

```
func ProcessFiles(paths []string) ([]string, error) {
    results := make([]string, 0, len(paths))

    for _, p := range paths {
        f, err := os.Open(p)
        if err != nil {
            return nil, fmt.Errorf("open file %q: %v", p, err)
        }
        // defer f.Close()

        if strings.HasSuffix(f.Name(), "_skip") {
            continue
        }

        r, err := processFile(f)
        if err != nil {
            return nil, fmt.Errorf("process file %q: %v", p, err)
        }
```

```
        results = append(results, r)
    }

    return results, nil
}
```

В решении выше мы закомментировали отложенное закрытие файла, и тест `TestProcessFiles` начал проходить!

```
$ cd tasks/02-defer-statement/file-descriptors-leak
$ go test -run=TestProcessFiles$ .
ok      tasks/02-defer-statement/file-descriptors-leak      0.927s
```

Чтобы разобраться в этой уличной магии, давайте залезем в `os.Open`, которая через нехитрую цепочку вызовов, приведёт нас к newFile:

```
Open -> OpenFile -> openFileNolog -> newFile
```

```
// src/os/file_unix.go
package os

// newFile is like NewFile, but if called from OpenFile or Pipe
// (as passed in the kind parameter) it tries to add the file to
// the runtime poller.
func newFile(fd uintptr, name string, kind newFileKind) *File {
    fdi := int(fd)
    if fdi < 0 {
        return nil
    }

    f := &File{&file{
        pfd: poll.FD{
            Sysfd:         fdi,
            IsStream:      true,
            ZeroReadIsEOF: true,
        },
        name:        name,
        stdoutOrErr: fdi == 1 || fdi == 2,
```

```
    }}

    // ...

    runtime.SetFinalizer(f.file, (*file).close)  // <- !!!
    return f

}
```

Ага, а вот и зарытая собака! Мы видим, что внутреннему объекту файла (по сути – обертке над файловым дескриптором) назначается **финализатор** в виде его закрытия:

```
runtime.SetFinalizer(f.file, (*file).close)
```

Что же это такое?

runtime.SetFinalizer

Обратимся к документации:

```
package runtime

// SetFinalizer sets the finalizer associated with obj to the
provided
// finalizer function. When the garbage collector finds an
unreachable block
// with an associated finalizer, it clears the association and runs
// finalizer(obj) in a separate goroutine. This makes obj reachable
again,
// but now without an associated finalizer. Assuming that
SetFinalizer
// is not called again, the next time the garbage collector sees
// that obj is unreachable, it will free obj.
// ...
func SetFinalizer(obj interface{}, finalizer interface{})
```

Таким образом **финализатор** – это функция, связанная с объектом, которая вызывается в тот момент, когда сборщик мусора считает объект недостижимым (никто больше не ссылается на наш объект). При этом в следующий раз GC

освободит объект, т.е. финализатор предшествует "реальному" освобождении памяти из-под объекта Go.

`runtime.SetFinalizer` делает ничто иное, как назначает финализатор указанному объекту. В нашем случае объекту `f.file` назначается финализатор в виде метода `(*file).close`, не принадлежащего никакому конкретному объекту, но входящему в множество методов типа `*file`:

```
runtime.SetFinalizer(f.file, (*file).close)

// Когда f.file будет недостижим, вызовется (*file).close(f.file)
// (т.е. f.file будет является ресивером метода close).

// Собственно так выглядит вызов любого метода, если убрать
синтаксический сахар:
// ff := new(file)
// ff.close() -> (*file).close(ff)
```

При этом `close` подчищает за собой финализатор (тыц):

```
func (file *file) close() error {
    if file == nil {
        return syscall.EINVAL
    }
    // ...

    // no need for a finalizer anymore
    runtime.SetFinalizer(file, nil)
    return err

}
```

Играемся с runtime.SetFinalizer

Заведём свой собственный финализатор ~~с блекджеком и распутными дамами~~:

```
func customFileFinalizer(f *os.File) {
    fmt.Printf("%p: I'm closed\n", f)
    f.Close()
```

```
}
```

И будем вешать его на файлы:

```
func ProcessFiles(paths []string) ([]string, error) {
    results := make([]string, 0, len(paths))

    for _, p := range paths {
        f, err := os.Open(p)
        if err != nil {
            return nil, fmt.Errorf("open file %q: %v", p, err)
        }
        runtime.SetFinalizer(f, customFileFinalizer)  // <- !!!
    // ...

}
```

Запустим тесты и удостоверимся, что действительно, в процессе работы программы сборщик мусора добирается до переменной `f` (каждый раз новой), живущей одну итерацию цикла (странно, что компилятор подобное не оптимизирует), и вызывает финализатор, с ней связанный:

```
$ go test -v -run=TestProcessFiles$ .
=== RUN   TestProcessFiles
0x140010fd658: I'm closed
0x140010fd650: I'm closed
...
0x14000010df0: I'm closed
0x14000010de8: I'm closed
0x14000010de0: I'm closed
--- PASS: TestProcessFiles (1.11s)
PASS
ok      tasks/02-defer-statement/file-descriptors-leak       1.246s
```

Если же мы запустим тест `TestProcessFiles_SkipAll`, в котором все открытые файлы пропускаются и чтение из них не идёт, то увидим, что он падает, а финализаторов в выводе не видно:

```
$ go test -v -run=TestProcessFiles_SkipAll  .
=== RUN   TestProcessFiles_SkipAll
```

```
    process_files_test.go:60:
                Error Trace:    process_files_test.go:60
                Error:          Received unexpected error:
                                open
/tmp/TestProcessFiles_3672459350_skip: too many open files
                Test:           TestProcessFiles_SkipAll
--- FAIL: TestProcessFiles_SkipAll (0.14s)
FAIL
FAIL    tasks/02-defer-statement/file-descriptors-leak       0.326s
FAIL
```

Для ответа на вопрос, почему так происходит, снова обратимся к документации:

```
// There is no guarantee that finalizers will run before a program
exits,
// so typically they are useful only for releasing non-memory
resources

// associated with an object during a long-running program.
```

Этот тест гораздо быстрее, чем тот, где мы `100_000` раз читали из файла, поэтому скорее всего сборщик мусора просто не успевает освобождать объекты.

Ну и напоследок, следует быть бдительными с быстродействием финализаторов:

```
// A single goroutine runs all finalizers for a program,
sequentially.
// If a finalizer must run for a long time, it should do so by
starting

// a new goroutine.
```

Предлагаем вам самим добавить в `customFileFinalizer` паузу в **1ms** и посмотреть, что выйдет 🙂

Тест "Запуск финализаторов"

Выберите все подходящие ответы из списка

- ☐ Финализаторы запускаются при выходе из функции подобно функциям, отложенным через defer
- ☐ Все финализаторы запускаются последовательно в одной горутине
- ☐ Финализаторы запускаются в тот момент, когда объект считается недостижимым
- ☐ Все финализаторы запускаются конкуретно в разных горутинах

Тест "Гарантируется ли вызов финализатора при завершении процесса?

Выберите один вариант из списка

- ☐ Да
- ☐ Нет
- ☐ Трудно сказать

Тест "runtime.KeepAlive"

Исходник примера.

Постарайтесь, не запуская кода, выбрать верный вариант того, что окажется на экране после запуска программы:

```
package main

import (
    "fmt"
    "runtime"
    "time"
)
```

```
type Hero struct {
    Name string
}

func NewHero(name string) *Hero {
    h := &Hero{Name: name}
    runtime.SetFinalizer(h, fin)
    return h
}

func fin(h *Hero) {
    fmt.Printf("%s was dead\n", h.Name)
}

func Battle() {
    n := NewHero("Naruto")
    _ = NewHero("Aoi Rokusho")

    go func() {
        for {
            runtime.KeepAlive(n)
        }
    }()
}

func main() {
    Battle()

    for i := 0; i < 3; i++ {
        time.Sleep(time.Second)
        runtime.GC()
    }
}
```

Выберите один вариант из списка

- ☐ Naruto was dead
- ☐ Aoi Rokusho was dead

- [ ] Aoi Rokusho was dead
  Naruto was dead

- [ ] Naruto was dead
  Aoi Rokusho was dead

- [ ] Aoi Rokusho was dead
  Aoi Rokusho was dead
  Aoi Rokusho was dead

- [ ] Вывод недетерминирован

Выводы

За несколько лет разработки на Go авторам курса не приходилось использовать `runtime.SetFinalizer` в боевом коде.

Данное API полезно, например, при взаимодействии с Си-биндингами, когда необходимо вызывать кастомные функции освобождения памяти для специфичных объектов. За пределами подобной сферы вас скорее всего будут бить по рукам, если принесёте на ревью освобождение ресурса с помощью финализатора – уж слишком это тонкий лёд и слишком слабые гарантии даёт `runtime.SetFinalizer`.

Когда-то даже было предложение выпилить эту функцию из стандартной библиотеки, но как сказано выше, в каких-то кейсах она может пригодиться.

Делитесь своими кейсами и пишите, что думаете на этот счёт в комментарии.

В любом случае, знать о таком полезно!