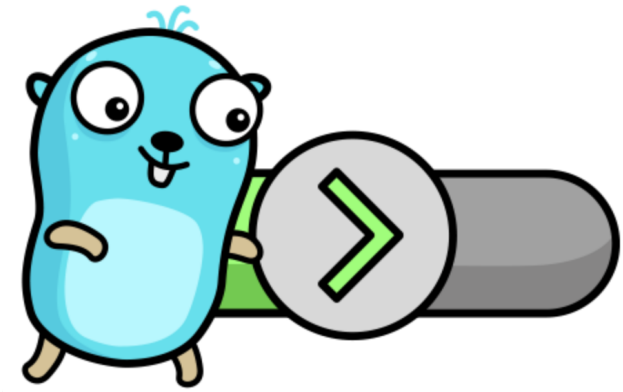
defer: оптимизации к Go 1.14

В этом уроке мы продолжим тему внутреннего устройства и производительности `defer`, а именно посмотрим на "улучшательства", которые внесли в него разработчики Go в релизах **Go 1.13** и **Go 1.14**.

Удостоверимся, что не так страшен ~~чёрт~~ `defer`, как его малюют; и разберёмся, когда всё-таки следует быть начеку и следить за местами использования `defer`.



defer после Go 1.14

Внимательный учащийся заметил, что начальные бенчмарки запускались на устаревшей версии гошки (Go 1.12).

Это было сделано преднамеренно, чтобы показать прогресс в вопросе реализации `defer` в Go.

В Go 1.14 Release Notes можно заметить следующую фразу:

This release improves the performance of most uses of `defer` to incur almost zero overhead compared to calling the deferred function directly. As a result, `defer` can now be used in performance-critical code without overhead concerns.

Расширим набор функций и проверим, так ли это на самом деле (исходник примера):

```
package main

//go:noinline
func withoutDefer() (result int) {
    foo(&result)
    return
}

//go:noinline
func withDefer() (result int) {
    defer foo(&result)
    return
}

//go:noinline
func withDefers() (result int) {
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

//go:noinline
func withNotOpenDefer() (result int) {
    for i := 0; i < 1; i++ {
        defer foo(&result)
    }
    return
}

//go:noinline
func with8Defers() (result int) {
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}
```

```go
//go:noinline
func with9Defers() (result int) {
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

func foo(i *int) {
    *i++
}
```

```
$ go1.14 version
go version go1.14.15 darwin/amd64

$ go1.14 test -benchmem -bench .
Benchmark_withoutDefer-12               959998963        1.12 ns/op
0 B/op          0 allocs/op
Benchmark_withDefer-12                  400825084        3.01 ns/op
0 B/op          0 allocs/op
Benchmark_withDefers-12                 167508066        7.17 ns/op
0 B/op          0 allocs/op
Benchmark_with8Defers-12                56962056         20.4 ns/op
0 B/op          0 allocs/op
Benchmark_withNotOpenDefer-12           21720300         52.6 ns/op
8 B/op          1 allocs/op
Benchmark_with9Defers-12                 5666712         210 ns/op
0 B/op          0 allocs/op
PASS
ok    examples/02-defer-statement/performance-2  8.627s
```

Неплохо! Сравним результаты в разных версиях Go:

```
$ go1.12 test -benchmem -bench . > 1.12.txt
```

```
$ go1.14 test -benchmem -bench . > 1.14.txt

$ benchstat -alpha 1.1 1.12.txt 1.14.txt
name                    old time/op    new time/op    delta
_withoutDefer-12          1.30ns ± 0%    1.12ns ± 0%  -13.85%
_withDefer-12             44.2ns ± 0%     3.0ns ± 0%  -93.19%
_withDefers-12             109ns ± 0%       7ns ± 0%  -93.42%
_with8Defers-12            263ns ± 0%      20ns ± 0%  -92.24%
_withNotOpenDefer-12      53.9ns ± 0%    52.6ns ± 0%   -2.41%
_with9Defers-12            298ns ± 0%     210ns ± 0%  -29.53%

# По аллокациям изменений нет.
```

На основе листинга выше сделаем следующие выводы:

- обработка одного `defer` ускорилась более, чем в 8 раз – теперь он занимает **до 5ns** (`withDefer`);

- как следствие кардинально ускорилась обработка трёх и восьми последовательных вызовов `defer` (`withDefers`, `with8Defers`);

- вызов девяти `defer` стал быстрее (`with9Defers`), но все равно он значительно медленнее, чем вызов возьми (`with8Defers`): 210ns vs 20ns 🤔;

- `defer`, который был в цикле (`withNotOpenDefer`), никак несоптимизировался; более того, он подарил нам одну аллокацию на 8 байт;

- вызов тривиальной функции (`withoutDefer`) не ухудшился - это не может не радовать 😆

Исходя из всего этого зададимся вопросом:

Какие же изменения, прокачавшие `defer`, произошли в компиляторе, и что за магическая потеря производительности при переходе от 8 к 9 отложенным вызовам, а также при обработке `defer` в цикле?

Оптимизации defer

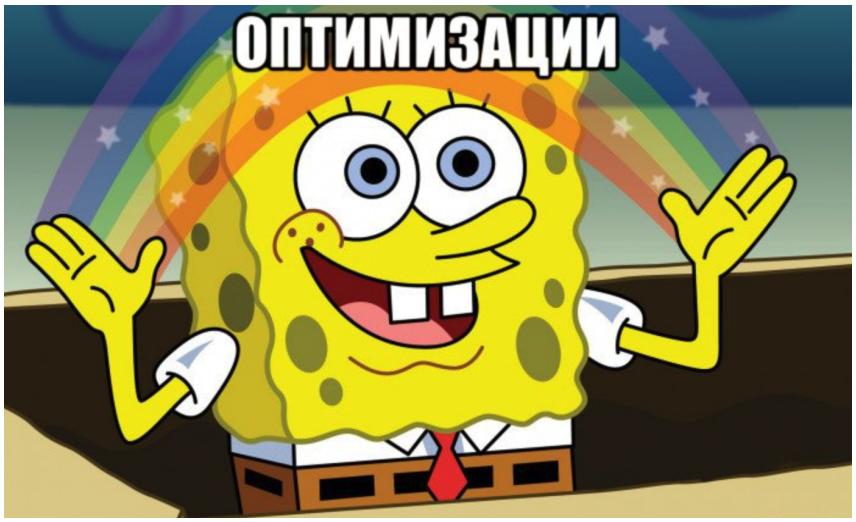
Сообщество Go давно подняло вопрос производительности `defer` (подробнее историю можно проследить в golang/go: Issues: runtime: defer is slow). Появилась практика, когда люди специально отказывались от `defer` ~~в своих супер-пупер высоконагруженных приложениях~~, заменяя его на явные вызовы в местах выхода из функции.

Разработчики языка не остались равнодушными и запилили ряд оптимизаций `defer`, значительно ускорив его (как мы заметили ранее). В списке литературы приведены все основные ссылки, которые позволят вам глубоко понять произошедшие в компиляторе изменения, а мы лишь коснёмся основных принципов.

Глобально можно выделить два основных изменения:

- разделение механизма обработки `defer`'ов и выделение его runtime-структуры на стеке (когда это возможно);
- введение понятия ***open-coded defer*** и встраивание отложенного вызова (когда это возможно).

Коснёмся их чуть подробнее.



Оптимизации defer: выделение на стеке

Предложение выделять `_defer` на стеке возникло очень давно, но реализация подъехала только в Go 1.13. Если для вас неочевидно, почему подобный вариант быстрее, рекомендуем прочитать этот ответ.

Заглянем в asm функции `with9Defers` и увидим, что `runtime.deferproc` сменился на `runtime.deferprocStack`:

```
# with9defers
        TEXT    "".with9Defers(SB), ABIInternal, $736-8

        # ...
        LEAQ    "".foo·f(SB), AX
        MOVQ    AX, ""..autotmp_1+672(SP)
        LEAQ    "".result+744(SP), CX
        MOVQ    CX, ""..autotmp_1+720(SP)
        LEAQ    ""..autotmp_1+648(SP), DX
        MOVQ    DX, (SP)
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc834

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc812
```

```
        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc790

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc768

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc746

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc724

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc702
        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc680

        # ...
        CALL    runtime.deferprocStack(SB)
        TESTL   AX, AX
        JNE     with9Defers_pc658

        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc658:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
```

```
        RET
with9Defers_pc680:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc702:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc724:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc746:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc768:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc790:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc812:
        XCHGL   AX, AX
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP
        RET
with9Defers_pc834:
        XCHGL   AX, AX
```

```
        CALL    runtime.deferreturn(SB)
        MOVQ    728(SP), BP
        ADDQ    $736, SP

        RET
```

На N `defer` мы получили N `runtime.deferprocStack`'ов и (N + 1) `runtime.deferreturn` – в этом плане ничего не поменялось.

Давайте посмотрим на небольшую реализацию runtime.deferprocStack:

```
package runtime

// deferprocStack queues a new deferred function with a defer record
on the stack.
// The defer record must have its siz and fn fields initialized.
// All other fields can contain junk.
// The defer record must be immediately followed in memory by
// the arguments of the defer.
// ...
func deferprocStack(d *_defer) {
    gp := getg()
    if gp.m.curg != gp {
        // go code on the system stack can't defer
        throw("defer on system stack")
    }

    // siz and fn are already set.
    // The other fields are junk on entry to deferprocStack and
    // are initialized here.
    d.started = false
    d.heap = false
    d.openDefer = false
    d.sp = getcallersp()
    d.pc = getcallerpc()
    d.framepc = 0
    d.varp = 0
    // The lines below implement:
    //   d.panic = nil
    //   d.fd = nil
    //   d.link = gp._defer
    //   gp._defer = d
    // ...
```

```
    *(*uintptr)(unsafe.Pointer(&d._panic)) = 0
    *(*uintptr)(unsafe.Pointer(&d.fd)) = 0
    *(*uintptr)(unsafe.Pointer(&d.link)) =
uintptr(unsafe.Pointer(gp._defer))
    *(*uintptr)(unsafe.Pointer(&gp._defer)) =
uintptr(unsafe.Pointer(d))

    return0()
    // No code can go here - the C return register has
    // been set and must not be clobbered.

}
```

Гораздо проще, чем runtime.deferproc! Доинициализировали входящий `d *_defer` и положили в цепочку – никаких `runtime.newdefer`, `runtime.memmove` и пр.

А откуда в `runtime.deferprocStack` попадает `d *_defer` и почему судя по комментарию в функции `// siz and fn are already set.`?

Вернёмся к ассемблеру и заметим, что перед вызовом `runtime.deferprocStack` по определённым офсетам в стеке `SP` сохраняются:

- суммарный размер аргументов и результатов откладываемой функции (в нашем случае – только 8 байт (на 64-разрядной машине) входящего указателя `i`);
- адрес откладываемой функции `foo`;
- адреса её аргументов (в нашем случае `&result`).

```
func with9Defers() (result int) {
    defer foo(&result)
    // ...
    return
}

func foo(i *int) {
    *i++

}
```

```
        MOVL    $8, ""..autotmp_1+648(SP)

        LEAQ    "".foo·f(SB), AX
        MOVQ    AX, ""..autotmp_1+672(SP)

        LEAQ    "".result+744(SP), CX
        MOVQ    CX, ""..autotmp_1+720(SP)

        LEAQ    ""..autotmp_1+648(SP), DX
        MOVQ    DX, (SP)

        CALL    runtime.deferprocStack(SB)
```

Вспомним структуру _defer:

```
// A _defer holds an entry on the list of deferred calls.
// If you add a field here, add code to clear it in freedefer and
deferProcStack.
//
// This struct must match the code in
cmd/compile/internal/gc/reflect.go:deferstruct
// and cmd/compile/internal/gc/ssa.go:(*state).call.
//
// All defers are logically part of the stack, so write barriers to
// initialize them are not required. All defers must be manually
scanned,
// and for heap defers, marked.
type _defer struct {
    siz     int32 // includes both arguments and results
    started bool
    heap    bool
    // openDefer indicates that this _defer is for a frame with
open-coded defers.
    // ...
    openDefer bool
    sp        uintptr  // sp at time of defer
    pc        uintptr  // pc at time of defer
    fn        *funcval // can be nil for open-coded defers
    _panic    *_panic  // panic that is running defer
    link      *_defer
```

```
    fd      unsafe.Pointer // funcdata for the function associated
with the frame
    varp    uintptr        // value of varp for the stack frame
    framepc uintptr

}
```

И посчитаем офсеты её полей, особенно нас интересуют `siz` и `fn`:

```
type _defer struct {        // offset   size
    siz       int32         // +0 (!)   4 bytes
    started   bool          // +4       1 byte
    heap      bool          // +5       1 byte
    openDefer bool          // +6       1 byte
                            // padding  1 byte
    sp      uintptr         // +8       8 bytes (на 64-разрядной
машине)
    pc      uintptr         // +16      8 bytes
    fn      *funcval        // +24 (!)  8 bytes
    _panic *_panic          // +32      8 bytes
    link   *_defer          // +40      8 bytes

    fd      unsafe.Pointer // +48      8 bytes
    varp    uintptr        // +56      8 bytes
    framepc uintptr        // +64      8 bytes
}

                            // +72 (!)  По этому смещению лежит
область памяти за _defer.
```

Соотнесём это с ассемблером и всё становится предельно ясно:

```
       MOVL    $8, ""..autotmp_1+648(SP)      # d.size = 8    (offset =
0)

        LEAQ    "".foo·f(SB), AX
        MOVQ    AX, ""..autotmp_1+672(SP)      # d.fn = &foo   (offset
= 672-648 = 24)

        LEAQ    "".result+744(SP), CX
        MOVQ    CX, ""..autotmp_1+720(SP)      # The defer record must
be immediately
                                               # followed in memory by
the arguments of the defer.
```

```
                                                # &result       (offset
= 720-648 = 72)

        LEAQ    ""..autotmp_1+648(SP), DX
        MOVQ    DX, (SP)                        # 0(SP) = &d (выставили
первый аргумент функции ниже)

        CALL    runtime.deferprocStack(SB)
```

Оптимизации defer: встраивание (inlining)

Подробно данная оптимизация описана в предложении Proposal: Low-cost defers through inline code, and extra funcdata to manage the panic case, но основная его суть такова:

Зачем нам страдать организацией defer-цепочек, если можно просто встраивать код откладываемой функции в конец текущей?

Нам понадобится [битовая маска] (чтобы помечать достигнутые `defer`'ы) и временные переменные, куда можно сохранять отложенные функции и их аргументы.

Например,

```
func foo() {
    defer f1(a)

    if cond {
        defer f2(b)
    }

}
```

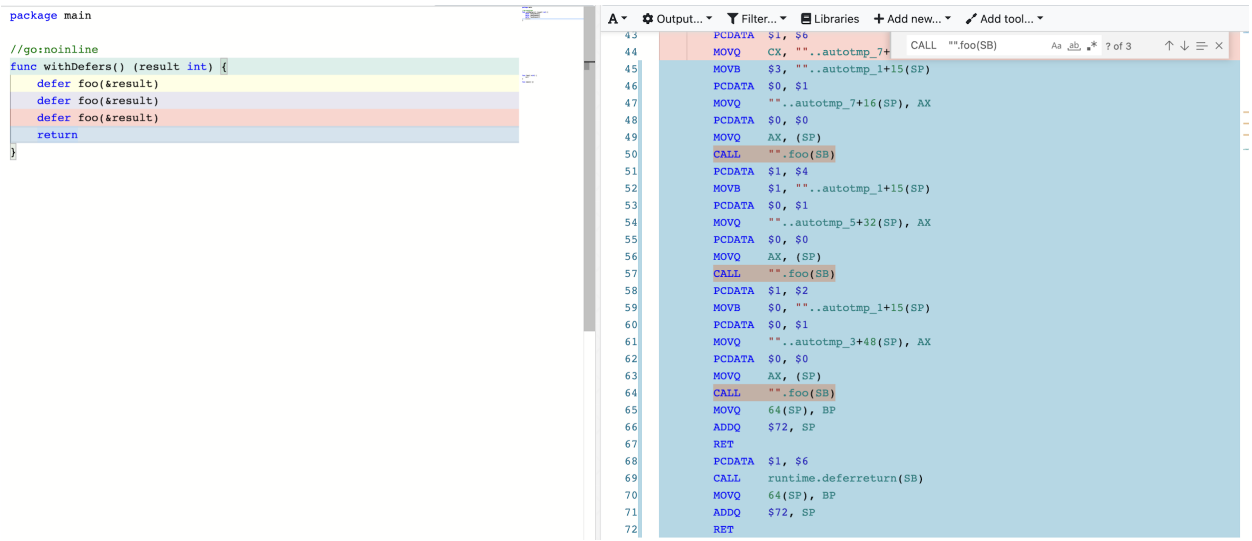
будет компилироваться в псевдокод вида

```
func foo() {
    deferBits |= 1<<0
    tmpF1 = f1
    tmpA = a

    if cond {
        deferBits |= 1<<1
        tmpF2 = f2
        tmpB = b
    }

    // Return.
```

```
    if deferBits & 1<<1 != 0 {
        deferBits &^= 1<<1
        tmpF2(tmpB)
    }

    if deferBits & 1<<0 != 0 {
        deferBits &^= 1<<0
        tmpF1(tmpA)
    }
}

// &^ - bit clear (AND NOT)

// https://golang.org/ref/spec#Arithmetic_operators
```

(пример, чтобы поиграться с маской).

В итоге вместо `runtime.deferproc` / `runtime.deferprocStack` мы получим явный вызов отложенной функции (что очевидно в разы быстрее):



При этом, как мы видим, "запасной" `runtime.deferreturn` никуда не делся:

```
        RET
          PCDATA  $1, $6
          CALL    runtime.deferreturn(SB)
          MOVQ    64(SP), BP
          ADDQ    $72, SP
```

```
        RET
```

Он нужен для корректной обработки **паники**, которая гораздо сложнее, чем в схеме без встраивания. На первый взгляд этот код недостижим (судя по предшествующему `RET`), но на самом деле компилятор знает, в какой момент и как сюда прыгать.

С **паникой** мы познакомимся позже, а более подробно об этом механизме в контексте встроенных `defer` читайте в proposal.

Внимательный читатель заметил, что в предыдущем шаге для рассмотрения `runtime.deferprocStack` мы использовали функцию `with9defers` и никого инлайнинга там не наблюдалось. Почему так? – сейчас узнаем 🙂

Open-coded defer

Хаки, о которых мы говорили ранее, работают только с **open-coded** `defer`'ами, т.е. такими, чьё количество вызовов можно строго определить на этапе компиляции:

```
func withDefer() { // Соптимизируется - defer ясен, как день.
    defer foo()
}

func withNotOpenDefer() { // Не соптимизируется - defer неочевиден.
    for i := 0; i < 1; i++ {
        defer foo()
    }

}
```

Таким образом, "открытый/ясный" `defer`:

- предсказуем по факту своего вызова (и их количеству);
- как следствие, не должен находится в блоке `for`;
- может находится в блоке `if` (как мы видели на предыдущем шаге, пометить его не составляет труда).

Такой `defer` мы сможем встроить или саллоцировать на стеке, в обратном случае мы снова приходим к медленной схеме `runtime.deferproc` + `runtime.deferreturn`.

Вспомним вопрос, почему же мы получили просадку производительности от `with8Defers` к `with9Defers`?

Дело в том, что переменная `deferBits`, с помощью которой мы помечаем `defer` для встраивания имеет размер **1 байт** (8 бит). Поэтому максимум мы можем встроить не более 8 `defer`!

Это было сделано специально, так как данная оптимизация рассчитана на небольшие функции с ограниченным количеством явных (не в цикле) отложенных вызовов, коими являются большинство функций в коде на Go.

Если вы выходите за этот предел, то **оптимизация встраивания** пропадает и остаётся лишь **оптимизация выделения на стеке** (при условии, что ваши `defer` не находятся в цикле).

Встраивание `defer` – сложная оптимизация, которая потребовала изменений как в рантайме, так и в самом компиляторе. Более того, за несколько лет это место сильно изменилось.

На момент **Go 1.17** мы имеем следующий код, который включает или выключает оптимизацию для функции (тыц и тыц):

```
// cmd/compile/internal/walk/walk.go
package walk

// The max number of defers in a function using open-coded defers. We
enforce this
// limit because the deferBits bitmask is currently a single byte (to
minimize code size)

const maxOpenDefers = 8

// cmd/compile/internal/walk/stmt.go
package walk
```

```
// ...
    case ir.ODEFER:
        n := n.(*ir.GoDeferStmt)
        ir.CurFunc.SetHasDefer(true)
        ir.CurFunc.NumDefers++
        if ir.CurFunc.NumDefers > maxOpenDefers {
            // Don't allow open-coded defers if there are more than
            // 8 defers in the function, since we use a single
            // byte to record active defers.
            ir.CurFunc.SetOpenCodedDeferDisallowed(true)
        }
        if n.Esc() != ir.EscNever {
            // If n.Esc is not EscNever, then this defer occurs in a
loop,
            // so open-coded defers cannot be used in this function.
            ir.CurFunc.SetOpenCodedDeferDisallowed(true)
        }
        fallthrough

// ...
```

А также [код](), который работает с битовой маской `deferBits`:

```
// cmd/compile/internal/ssagen/ssa.go
// ...

func (s *state) openDeferRecord(n *ir.CallExpr) {
    //...

    // Update deferBits only after evaluation and storage to stack of
    // args/receiver/interface is successful.
    bitvalue := s.constInt8(types.Types[types.TUINT8],
1<<uint(index))
    newDeferBits := s.newValue2(
        ssa.OpOr8,
        types.Types[types.TUINT8],
        s.variable(deferBitsVar, types.Types[types.TUINT8]),
        bitvalue)
    s.vars[deferBitsVar] = newDeferBits
    s.store(types.Types[types.TUINT8], s.deferBitsAddr, newDeferBits)

}
```

и т.д.

Если вам интересно проследить историю изменений с самого начала, то вам сюда.

Тест "Какие сниппеты содержат только open defer'ы?"

1)

```
package httptest

func (s *Server) goServe() {
    s.wg.Add(1)
    go func() {
        defer s.wg.Done()
        s.Config.Serve(s.Listener)
    }()

}
```

2)

```
package os_test

import (
    "fmt"
    . "os"
    "testing"
)

func TestExpandEnvShellSpecialVar(t *testing.T) {
    for _, tt := range shellSpecialVarTests {
        Setenv(tt.k, tt.v)
        defer Unsetenv(tt.k)

        argRaw := fmt.Sprintf("$%s", tt.k)
        argWithBrace := fmt.Sprintf("${%s}", tt.k)
        if gotRaw, gotBrace := ExpandEnv(argRaw),
ExpandEnv(argWithBrace); gotRaw != gotBrace {
            t.Errorf(
                "ExpandEnv(%q) = %q, ExpandEnv(%q) = %q; expect them
to be equal",
                argRaw, gotRaw, argWithBrace, gotBrace)
        }
```

```
    }

}
```

3)

```
package httputil

func (m *maxLatencyWriter) stop() {
    m.mu.Lock()
    defer m.mu.Unlock()

    m.flushPending = false
    if m.t != nil {
        m.t.Stop()
    }

}
```

4)

```
package os_test

func TestVariousDeadlines4Proc(t *testing.T) {
    // Cannot use t.Parallel - modifies global GOMAXPROCS.
    if testing.Short() {
        t.Skip("skipping in short mode")
    }
    defer runtime.GOMAXPROCS(runtime.GOMAXPROCS(4))
    testVariousDeadlines(t)

}
```

5)

```
package httputil

func (sc *ServerConn) Read() (*http.Request, error) {
    var req *http.Request
    var err error

    // ...

    sc.mu.Lock()
```

```
    if sc.we != nil { // no point receiving if write-side broken or
closed
        defer sc.mu.Unlock()
        return nil, sc.we
    }

    // ...

}
```

Выберите все подходящие ответы из списка

- [ ] 1
- [ ] 2
- [ ] 3
- [ ] 4
- [ ] 5

Оптимизации defer: возвращаясь к примеру

Применим полученные знания и прокомментируем пример, для которого мы запускали бенчмарки:

```
package main

//go:noinline
func withoutDefer() (result int) {
    foo(&result)                                // Прямой (direct,
open-coding) вызов.
    return
}

//go:noinline
func withDefer() (result int) {
    defer foo(&result)                          // Open defer - встроится.
    return
}

//go:noinline
func withDefers() (result int) {                // Open defer'ы - все
встроятся.
    defer foo(&result)
    defer foo(&result)
```

```
    defer foo(&result)
    return
}

//go:noinline
func withNotOpenDefer() (result int) {
    for i := 0; i < 1; i++ {
        defer foo(&result)                  // Не open defer (несмотря
на 1 итерацию) - не встроится.
    }                                       // Будет выделен на куче
через runtime.deferproc.
    return
}

//go:noinline
func with8Defers() (result int) {           // Open defer'ы - все
встроятся.
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

//go:noinline
func with9Defers() (result int) {          // Open defer'ы, но их
больше 8 - не встроятся.
    defer foo(&result)                     // Все выделятся на стеке
через runtime.deferprocStack.
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

func foo(i *int) {
```

```
    *i++

}
```

Тест "Великолепная девятка"

Исходник примера.

Перед вами две функции

```
func with9Defers() (result int) {
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    defer foo(&result)
    return
}

func with9DefersHack() (result int) {
    func() {
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
        defer foo(&result)
    }()
    defer foo(&result)
    return

}
```

Какая из них отрабатывает быстрее на версиях Go моложе 1.14?

Пишите в комментарии, почему так 😳

Выберите один вариант из списка

- [ ] with9Defers
- [ ] with9DefersHack
- [ ] Это невозможно определить
- [ ] Функции в целом отрабатывают за одинаковое время

Тест "Среднее время выполнения defer после Go 1.14"

```
func main() {
    defer someFunc()

}
```

Выберите один вариант из списка

- [ ] 100-500 нс
- [ ] 50-100 нс
- [ ] 10-50 нс
- [ ] 2-5 нс

Тест "defer и собеседование"

Кандидат реализовал на собеседовании некоторую задачу и предоставил следующий код:

```
func DoSomeWork() (Result, error) {
    // ...

    var wg sync.WaitGroup
    wg.Add(n)
    for i := 0; i < n; i++ {
        go func() {
            // ...
            if err := someOp1(); err != nil {
                wg.Done()
                return
            }
```

```
            // ...
            if err := someOp2(); err != nil {
                wg.Done()
                return
            }

            // ...
            wg.Done()
        }()
    }

    // ...

}
```

На последовавший вопрос, почему бы не сделать

```
    // ...
    for i := 0; i < n; i++ {
        go func() {
            defer wg.Done
            // ...
        }()
    }

    // ...
```

Кандидат ответил, что всегда избегает `defer` во избежание потери производительности программы, так как когда-то где-то читал, что использовать `defer` – это плохо и дорого, но сам глубоко в вопросе не разбирался.

Выберите верные утверждения исходя из материала последних двух уроков, а также вашего личного опыта.

Выберите все подходящие ответы из списка

- ☐ В общем случае экспертиза кандидата ставится под сомнение и требует дополнительных проверок.
- ☐ В целом допустимо, если кандидат приведёт минусы и плюсы такого подхода и пообещает больше так не делать.
- ☐ Огонь, мы вообще избегаем defer в нашем коде. Берём товарища.

- [ ] Сомнительный вариант решения для собеседования, так как позволяет собеседующим накидать вам неудобных вопросов.
- [ ] Premature optimization is the root of all evil.
- [ ] Возможно, человек сидит на проектах с древней гошкой и делает это по привычке. В любом случае – это маркер для собеседующего.

Задача "Какое максимальное количество функций можно отложить через defer?"

Постарайтесь применить полученные в предыдущих уроках знания и ответить на этот непростой вопрос.

Данная задача имеет свободную форму ответа – любой ответ будет считаться верным. Она нужна больше для самопроверки, попробуйте представить ход своих мыслей, если бы, например, вам задали такой вопрос на собеседовании.

Промежуточные выводы

Мы узнали, что ужасно дорогой `defer` – это давно уже миф.

Если вы не используете `defer` в цикле или в вашей функции до 8 `defer` (что соответствует 99% программ на Go), то можно не переживать, т.к. накладные расходы на него будут минимальны.

В следующем модуле (после того, как познакомимся с **паникой**) мы узнаем, по каким ещё причинам следует склоняться в пользу использования `defer` вместо явных вызовов по местам выхода из функции.

На этой прекрасной ноте, держа в голове, что **преждевременные оптимизации – это зло** (только если вы не разработчик C++), мы двигаемся дальше 🙂