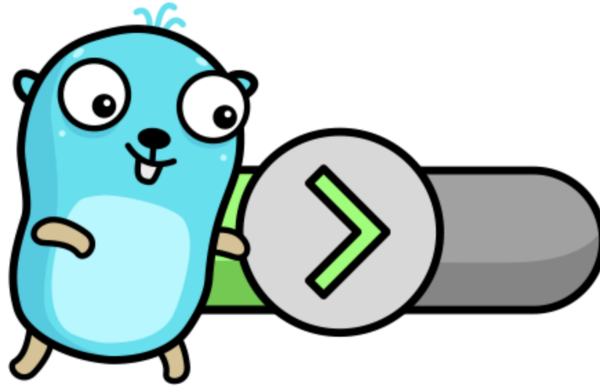
Исключения из правил

В этом уроке мы поговорим о том, какие ошибки обычно не проверяют и в каких случаях валидно использовать возвращаемое значение даже при наличии ошибки.



fmt.Print*

Стандартный пакет fmt содержит ряд функций-братишек сишного семейства printf:

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintf(w io.Writer, format string, a ...interface{}) (n int,
err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)

func Print(a ...interface{}) (n int, err error)
func Printf(format string, a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
```

Как мы видим, все они возвращают `(n int, err error)`, что является на самом деле следствием использования `io.Writer`:

```
package io

type Writer interface {
    Write(p []byte) (n int, err error)
```

```
}
```

При этом `Print* = Fprint* + os.Stdout`:

```
package fmt

// Printf formats according to a format specifier and writes to standard output.
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}

// Fprintf formats according to a format specifier and writes to w.
// It returns the number of bytes written and any write error encountered.
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

Ошибки от `Print*` обычно **игнорируются** и происходит это по двум причинам:

1) Чаще всего функция используется для дебага. В боевом коде её быть не должно, на что по умолчанию ругаются некоторые линтеры:

```
example.go:6:2: use of `fmt.Println` forbidden by pattern `^fmt\.Print(|f|ln)$` (forbidigo)
        fmt.Println("hello")
```

Даже IDE не подсветит нам **unhandled error**:

```go
func main() {
	fmt.Printf( format: "hello")
}
```

2) Как видно из исходников выше, ошибка в `Printf` – это по сути ошибка записи в файловый дескриптор стандартного потока вывода:

```go
package os

var (
	Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
	Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
	Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Сложно представить, что должно случится, чтобы данная операция завершилась с ошибкой.

Стандартный пакет log даже не пробрасывает подобную ошибку наверх:

```go
package log

var std = New(os.Stderr, "", LstdFlags)

func New(out io.Writer, prefix string, flag int) *Logger {
	return &Logger{out: out, prefix: prefix, flag: flag}
}

type Logger struct {
	// ...
}

func (l *Logger) Printf(format string, v ...interface{}) { // <- Тут ошибки уже нет.
	l.Output(2, fmt.Sprintf(format, v...))
}
```

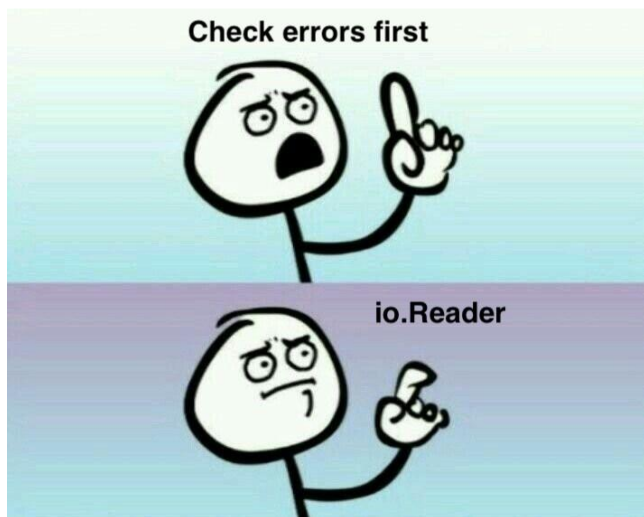
```
func (l *Logger) Output(calldepth int, s string) error { // <- Тут ошибка есть.
    // ...
    _, err := l.out.Write(l.buf)
    return err

}
```

В отличие от `Print*` с `Fprint*`-функциями ситуация другая, ошибку от них **игнорировать не нужно**:

```
func main() {
    fmt.Fprintf(os.Stdout, format: "hello")
}
    Unhandled error
```

Очевидно, что это связано это с тем, что осуществлять форматированную печать вы можете не только в `os.Stdout`, но и в любой `io.Writer` – файл, буфер, сетевое соединение и т.д. И чаще всего ошибка от подобной операции имеет вес.

io.Reader



Ранее мы рассмотрели идиому "Сначала проверяем ошибку", так вот – перед нами исключение из этого правила:

```
package io
```

```go
// Reader is the interface that wraps the basic Read method.
//
// Read reads up to len(p) bytes into p. It returns the number of
bytes
// read (0 <= n <= len(p)) and any error encountered. Even if Read
// returns n < len(p), it may use all of p as scratch space during
the call.
// If some data is available but not len(p) bytes, Read
conventionally
// returns what is available instead of waiting for more.
//
// When Read encounters an error or end-of-file condition after
// successfully reading n > 0 bytes, it returns the number of
// bytes read. It may return the (non-nil) error from the same call
// or return the error (and n == 0) from a subsequent call.
// An instance of this general case is that a Reader returning
// a non-zero number of bytes at the end of the input stream may
// return either err == EOF or err == nil. The next Read should
// return 0, EOF.
//
// Callers should always process the n > 0 bytes returned before
// considering the error err. Doing so correctly handles I/O errors
// that happen after reading some bytes and also both of the
// allowed EOF behaviors.
//
// Implementations of Read are discouraged from returning a
// zero byte count with a nil error, except when len(p) == 0.
// Callers should treat a return of 0 and nil as indicating that
// nothing happened; in particular it does not indicate EOF.
//
// Implementations must not retain p.
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Пользуясь функцией `Read`, нужно сначала обработать первый возвращаемый аргумент (количество прочитанных байт), а потом уже смотреть на наличие ошибки:

// Callers should always process the n > 0 bytes returned before considering the error err.

// Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviours.

Как видим, это поможет нам обработать как промежуточные ошибки ввода-вывода, так и реализовать корректную вычитку до `io.EOF`.

Метод `Read` принимает слайс байт, а не отдаёт, чтобы переложить ответственность по аллокации буфера с реализации `io.Reader` на его пользователя.

Более того при обратной сигнатуре вида

```
Read() ([]byte, error)
```

возвращаемый буфер всегда будет аллоцироваться новый и всегда будет оказываться на куче, что добавляет работы сборщику мусора и не очень подходит *жадным* читателям.

Подробнее обо всём этом можно почитать [здесь].

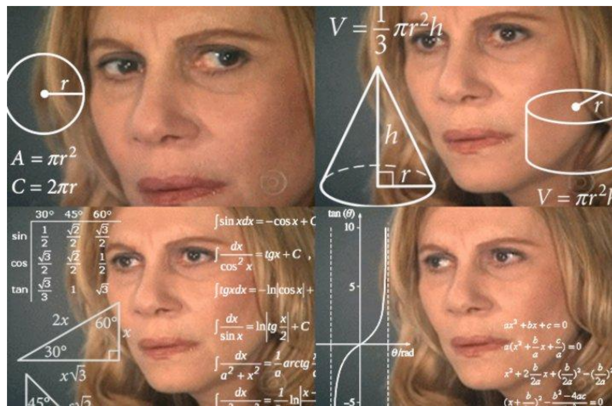
Ради эффективного API разработчикам Go пришлось нарушить базовую идиому языка и позволить тому, кто осуществляет чтение, одновременно пользоваться и возвращаемым значением и ошибкой.

Но пакет `io` – **единственный в языке, позволяющий себе так делать** – мы помним, что в общем случае это недопустимо!

Выводы

Всегда внимательно читаем документацию к стандартной библиотеке Go, особенно к подобным фундаментальным интерфейсам, прежде чем использовать их.

Кроме [io.Reader]() есть ещё [io.Writer]() и множество их производных, в работе которых без пол-литра не разберёшься.



Задача "Read By Chunk"

[Ссылка на заготовку]().

Вам необходимо реализовать функцию `ReadByChunk`:

```go
import "io"

var ErrInvalidChunkSize error

func ReadByChunk(r io.Reader, chunkSize int) (chunks [][]byte, err error) {
    return nil, nil
}
```

Она вычитывает входящий `io.Reader` по частям размера `chunkSize` и возвращает слайс этих частей.

Например:

```
chunks, err := ReadByChunk(strings.NewReader("Errors are values."),
7)
// err == nil
// chunks == [][]byte{
//    {'E', 'r', 'r', 'o', 'r', 's', ' '}, {'a', 'r', 'e', ' ', 'v',
'a', 'l'}, {'u', 'e', 's', '.'},

// }
```

- При нулевом или отрицательном `chunkSize` мы возвращаем `ErrInvalidChunkSize`;
- При `nil` читателе мы возвращаем `nil` слайс частей, ошибки нет.

Подробности в заготовке задачи и тестах.

io.Closer

```
package io

type Closer interface {
    Close() error

}
```

Наиболее частый метод, который вводит в ступор при вопросе – нужно ли обрабатывать ошибку от него?

При выборе следует ответить на следующие вопросы:

- Важно ли нам знать об этой ошибке?
- Наши действия, если ошибка возникнет и мы узнаем о ней?
- Какие побочные эффекты ошибка несёт за собой?

- Что будет, если мы проигнорируем её?

Бывает и такое, что тип согласно сигнатуре возвращает в методе `Close` ошибку, чтобы реализовывать `io.Closer`, а на самом деле там всегда `return nil` (правда вопрос, должны ли мы полагаться на это знание?).

Разберём несколько наиболее популярных примеров.

(*http.Response).Body.Close

Из [документации](#) к `(http.Response).Body` мы знаем, что тело ответа следует вычитывать и закрывать:

// It is the caller's responsibility to close Body. The default HTTP client's Transport may not reuse

// HTTP/1.x "keep-alive" TCP connections if the Body is not read to completion and closed.

Обычно это делается следующим образом (закроем глаза на использование дефолтного клиента и отсутствие контекста):

```
// ...
    res, err := http.Get("http://www.golang-courses.ru/")
    if err != nil {
        return fmt.Errorf("cannot do GET: %v", err)
    }
    defer res.Body.Close() // Закрыли тело (необязательно через
defer).

    index, err := io.ReadAll(res.Body) // Вычитали тело.
    if err != nil {
        return fmt.Errorf("cannot read body: %v", err)
    }

// ...
```

Возникает вопрос, а должны ли мы обрабатывать ошибку от `Close`? Т.е. иногда можно встретить такой вариант:

```
defer func() {
    if err := res.Body.Close(); err != nil {
        log.Println("cannot close response body: " + err.Error())
    }

}()
```

На самом деле подобное скорее всего избыточно по следующим причинам:

- Мы уже вычитали тело и можем с ним работать. Что нам делать при ошибке: забить на успешный ответ и вернуть ошибку от `Close`? Сомнительно.
- Если посмотреть на несколько реализаций тела ответа, то можно увидеть, что в `Close` они возвращают `nil`.

NoBody:

```
// src/net/http/http.go
package http

// NoBody is an io.ReadCloser with no bytes. Read always returns EOF
// and Close always returns nil.
var NoBody = noBody{}

type noBody struct{}

func (noBody) Read([]byte) (int, error)         { return 0, io.EOF }
func (noBody) Close() error                     { return nil }

func (noBody) WriteTo(io.Writer) (int64, error) { return 0, nil }
```

http2transportResponseBody:

```
// src/net/http/h2_bundle.go
package http
```

```go
// transportResponseBody is the concrete type of
Transport.RoundTrip's
// Response.Body. It is an io.ReadCloser. On Read, it reads from
cs.body.
// On Close it sends RST_STREAM if EOF wasn't already seen.
type http2transportResponseBody struct {
    cs *http2clientStream
}

func (b http2transportResponseBody) Read(p []byte) (n int, err error)
{
    // ...
}

var http2errClosedResponseBody = errors.New("http2: response body
closed")

func (b http2transportResponseBody) Close() error {
    cs := b.cs
    cc := cs.cc

    serverSentStreamEnd := cs.bufPipe.Err() == io.EOF
    unread := cs.bufPipe.Len()

    if unread > 0 || !serverSentStreamEnd {
        cc.mu.Lock()
        cc.wmu.Lock()
        if !serverSentStreamEnd {
            cc.fr.WriteRSTStream(cs.ID, http2ErrCodeCancel)
            cs.didReset = true
        }
        // Return connection-level flow control.
        if unread > 0 {
            cc.inflow.add(int32(unread))
            cc.fr.WriteWindowUpdate(0, uint32(unread))
        }
        cc.bw.Flush()
        cc.wmu.Unlock()
        cc.mu.Unlock()
    }

    cs.bufPipe.BreakWithError(http2errClosedResponseBody)
    cc.forgetStreamID(cs.ID)
    return nil
```

```
}
```

[io.NopCloser]:

```
// src/net/http/client.go
package http

// send issues an HTTP request.
// Caller should close resp.Body when done reading from it.
func send(
    ireq *Request,
    rt RoundTripper,
    deadline time.Time,
) (resp *Response, didTimeout func() bool, err error) {
    // ...
    if resp.Body == nil {
        // The documentation on the Body field says “The http Client
and Transport
        // guarantee that Body is always non-nil, even on responses
without a body
        // or responses with a zero-length body.”

        // ...
        resp.Body = io.NopCloser(strings.NewReader(""))
    }
    // ...

}
```

[streamReader]:

```
// src/net/http/roundtrip_js.go
package http

// streamReader implements an io.ReadCloser wrapper for
ReadableStream.
type streamReader struct {
    pending []byte
    stream  js.Value
    err     error // sticky read error
}
```

```go
func (r *streamReader) Read(p []byte) (n int, err error) {
    if r.err != nil {
        return 0, r.err
    }
    // ...
}

func (r *streamReader) Close() error {
    // This ignores any error returned from cancel method. So far, I
did not encounter any concrete
    // situation where reporting the error is meaningful. Most users
ignore error from resp.Body.Close().
    // If there's a need to report error here, it can be implemented
and tested when that need comes up.
    r.stream.Call("cancel")
    if r.err == nil {
        r.err = errClosed
    }
    return nil

}
```

Обратим внимание на комментарий внутри:

// So far, I did not encounter any concrete situation where reporting the error is meaningful.

// Most users ignore error from resp.Body.Close().

arrayReader:

```go
// arrayReader implements an io.ReadCloser wrapper for ArrayBuffer.
type arrayReader struct {
    arrayPromise js.Value
    pending      []byte
    read         bool
    err          error // sticky read error
}
```

```go
func (r *arrayReader) Read(p []byte) (n int, err error) {
    if r.err != nil {
        return 0, r.err
    }
    // ...
}

func (r *arrayReader) Close() error {
    if r.err == nil {
        r.err = errClosed
    }
    return nil
}
```

Если вы же всё-таки переживаете или сомневаетесь, то одним из вариантов будет возвращать ошибку от `Close` явно (исходник примера):

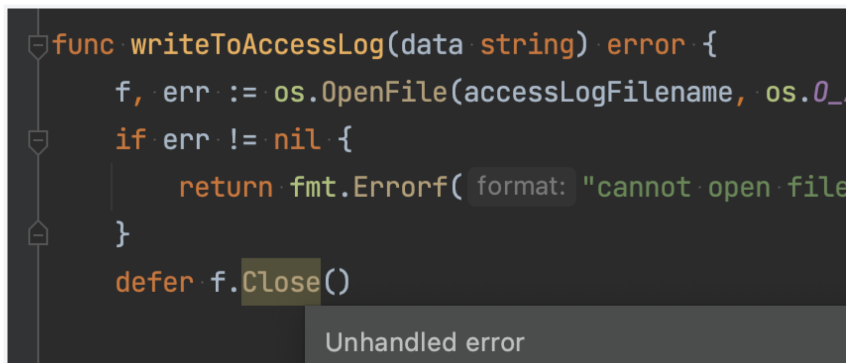
```go
func httpGet(url string) ([]byte, error) {
    res, err := http.Get(url)
    if err != nil {
        return nil, fmt.Errorf("cannot do GET: %v", err)
    }
    defer res.Body.Close() // Обращаем внимание, что этот defer
остаётся!

    body, err := io.ReadAll(res.Body)
    if err != nil {
        return nil, fmt.Errorf("cannot read body: %v", err)
    }

    if err := res.Body.Close(); err != nil {
        return nil, fmt.Errorf("cannot close body: %v", err)
    }
    return body, nil

}
```

Пишите в комментах своё мнение на этот счёт, и как вы делаете в боевом коде.

(*os.File).Close

Посмотрим на небольшой пример, подобный код которого можно встретить в реальных проектах (исходник примера):

```
func writeToAccessLog(data string) error {
    f, err := os.OpenFile(accessLogFilename,
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return fmt.Errorf("cannot open file: %v", err)
    }
    defer f.Close() // Закрываем файл.

    if _, err := f.WriteString(data + "\n"); err != nil {
        return fmt.Errorf("cannot write data: %v", err)
    }

    return nil
}
```



С ошибкой от закрытия файла не всё так просто, при её игнорировании мы можем пропустить ошибку записи данных или в редких случаях утечку файлового дескриптора. Подробности ниже.

Давайте покурим мануал Linux и узнаем, как он предлагает нам вести себя с ошибками от системного вызова `close` (к которому сводится гошный вызов `(*os.File).Close`):

A careful programmer will check the return value of ***close****(), since it is quite possible that errors on a previous write(2) operation are reported only on the final* ***close****() that releases*

the open file description. Failing to check the return value when closing a file may lead to silent loss of data. This can especially be observed with NFS and with disk quota.

Т.е. одна из ошибок от `close` говорит нам, что предыдущая операция записи была неуспешна (при этом при самом `Write` мы ошибки не получим). Это ошибка `EIO`:

```
$ man close
...

NAME
     close -- delete a descriptor

...

DESCRIPTION
     The close() call deletes a descriptor from the per-process
object reference table.
     If this is the last reference to the underlying object, the
object will be deactivated.

...

ERRORS
     The close() system call will fail if:

     [EBADF]            fildes is not a valid, active file
descriptor.

     [EINTR]            Its execution was interrupted by a signal.


     [EIO]              A previously-uncommitted write(2) encountered
an input/output error.
```

Перепишем пример выше так, чтобы явно проверять ошибку от `Close` (похожий код мы писали в предыдущем шаге, когда говорили о `resp.Body.Close`):

```go
func writeToAccessLog(data string) error {
    f, err := os.OpenFile(accessLogFilename,
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return fmt.Errorf("cannot open file: %v", err)
```

```
    }
    defer f.Close() // Обращаем внимание, что defer остаётся!

    if _, err := f.WriteString(data + "\n"); err != nil {
        return fmt.Errorf("cannot write data: %v", err)
    }

    return f.Close()
}
```

С помощью `defer` аккуратно покрываем преждевременный выход из функции (прочие ошибки имеют приоритет, но не допускаем утечки файлового дескриптора), а в конце возвращаем явно ошибку от `Close` (теперь она имеет приоритет).

На самом деле в общем случае не рекомендуется вызывать системный вызов `close` дважды подряд, так как вы можете закрыть дескриптор, который уже прибрал себе другой поток (после того как дескриптор был освобождён первым вызовом `close`):

Retrying the close() after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to) be closed. This can occur because the Linux kernel always releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Но в Go это операция безопасная, в позитивном сценарии [мы получим]() в `defer` ошибку "**file already closed**", на которую и так не обращаем внимания.

Более того из цитаты выше мы можем сделать вывод, что независимо от результата `close` – файловый дескриптор скорее всего будет закрыт:

This can occur because the Linux kernel always releases the file descriptor early in the close operation, freeing it for reuse.

*Many other implementations similarly always close the file descriptor (except in the case of **EBADF**, meaning that the file descriptor was invalid) even if they subsequently report an error on return from **close**(). POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.*

Поэтому больше важно само наличие вызова `close`, чем проверка ошибки от него (которую мы в принципе можем не проверять, если работаем с файлом на чтение, а не на запись).

И, наконец, на самом деле `close` не гарантирует то, что данные сбросятся на диск, освободив кэши ОС:

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use [fsync(2)](). (It will depend on the disk hardware at this point.)

Поэтому более грамотным вариантом для гарантированной записи будет использование `fsync` + `close`:

*A careful programmer who wants to know about I/O errors may precede **close**() with a call to [fsync(2)]().*

```
func writeToAccessLog(data string) error {
    f, err := os.OpenFile(accessLogFilename,
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return fmt.Errorf("cannot open file: %v", err)
    }
    defer f.Close() // Обращаем внимание, что defer остаётся!

    if _, err := f.WriteString(data + "\n"); err != nil {
        return fmt.Errorf("cannot write data: %v", err)
    }
```

```
    return f.Sync() // <- !!!
}
```

Тест "Выберите наиболее грамотный порядок операций"

При записи в файл, когда вы хотите гарантировать, что запись на диск прошла и прошла успешно.

Выберите один вариант из списка

- ☐ f.Open() + defer f.Close() + f.Write() + return f.Sync()
- ☐ f.Open() + defer f.Close() + f.Write() + return f.Close()
- ☐ f.Open() + f.Write() + defer f.Close() + return f.Sync()
- ☐ f.Open() + f.Write() + несколько вызовов f.Close() по местам + return f.Sync()
- ☐ f.Open() + f.Write() + defer f.Close() + return f.Close()
- ☐ f.Open() + f.Write() + return f.Sync()
- ☐ f.Open() + return f.Write()
- ☐ f.Open() + f.Close() + return f.Write()
- ☐ f.Open() + f.Write() + return f.Close()

http.ResponseWriter

При реализации HTTP сервера [стандартными средствами Go]() часто возникает вопрос, нужно ли обрабатывать ошибки от записи в `http.ResponseWriter`?

```
package http

// A ResponseWriter interface is used by an HTTP handler to
// construct an HTTP response.
type ResponseWriter interface {
    // Header returns the header map that will be sent by
WriteHeader.
    Header() Header

    // Write writes the data to the connection as part of an HTTP
reply.
    Write([]byte) (int, error)
```

```
    // WriteHeader sends an HTTP response header with the provided status code.
    WriteHeader(statusCode int)
}
```

```
import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("content-type", "application/json")
        w.Write([]byte(`{"msg": "OK"}`))
    })

    if err := http.ListenAndServe(":8080", http.DefaultServeMux); err != nil {
        log.Fatal(err)
    }
}
```

```
func main() {
    http.HandleFunc( pattern: "/", func(w http.ResponseWriter, req *http.Request) {
        w.Header().Set( key: "content-type",  value: "application/json")
        w.Write([]byte(`{"msg": "OK"}`))
    })
```

Unhandled error

На самом деле – **решать вам.**

Обычно данную ошибку опускают, но давайте посмотрим, при каких обстоятельствах она может произойти.

Для начала определим, какие ошибки (кроме непосредственно ошибки записи в соединение) в целом может вернуть `(http.ResponseWriter).Write`:

```
package http
```

```go
// Errors used by the HTTP server.
var (
    // ErrBodyNotAllowed is returned by ResponseWriter.Write calls
    // when the HTTP method or response code does not permit a body.
    ErrBodyNotAllowed = errors.New("http: request method or response
status code does not allow body")

    // ErrHijacked is returned by ResponseWriter.Write calls when
    // the underlying connection has been hijacked using the
    // Hijacker interface. A zero-byte write on a hijacked
    // connection will return ErrHijacked without any other side
effects.
    ErrHijacked = errors.New("http: connection has been hijacked")

    // ErrContentLength is returned by ResponseWriter.Write calls
    // when a Handler set a Content-Length response header with a
    // declared size and then attempted to write more bytes than
declared.
    ErrContentLength = errors.New("http: wrote more than the declared
Content-Length")

)
```

Чтобы понять, когда они происходят, обратимся к [приватной реализации](#) интерфейса `http.ResponseWriter`, используемой гошным HTTP сервером:

```go
func (w *response) write(lenData int, dataB []byte, dataS string) (n
int, err error) {
    if w.conn.hijacked() {
        // ...
        return 0, ErrHijacked
    }

    // ...
    if !w.bodyAllowed() {
        return 0, ErrBodyNotAllowed
    }

    // ...
    w.written += int64(lenData)
    if w.contentLength != -1 && w.written > w.contentLength {
        return 0, ErrContentLength
    }
```

```go
    // ...
    if dataB != nil {
        return w.w.Write(dataB)
    } else {
        return w.w.WriteString(dataS)
    }

}
```

Всё достаточно очевидно – `w.Write` вернёт ошибку, когда ([исходник примера](#)):

1) Мы до этого `hijack`-нули наше соединение (после этого мы можем работать с ним только в "сыром" виде)

```go
http.HandleFunc("/hijack-err", func(w http.ResponseWriter, r
*http.Request) {
    hj, ok := w.(http.Hijacker)
    if !ok {
        http.Error(w, "no hijacking support",
http.StatusInternalServerError)
        return
    }

    conn, _, err := hj.Hijack()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    defer conn.Close()

    w.Header().Set("content-type", "application/json")
    w.Write([]byte(`{"msg": "OK"}`)) // http: connection has been
hijacked

})
```

2) Мы пытаемся записать тело ответа, хотя для нашего кода ответа (или метода запроса) оно не разрешено:

```go
http.HandleFunc("/body-not-allowed-err", func(w http.ResponseWriter,
r *http.Request) {
    w.WriteHeader(http.StatusNoContent)
    w.Header().Set("content-type", "application/json")
```

```
    w.Write([]byte(`{"msg": "OK"}`)) // http: request method or
response status code does not allow body

})

package http

// bodyAllowedForStatus reports whether a given response status code
// permits a body. See RFC 7230, section 3.3.
func bodyAllowedForStatus(status int) bool {
    switch {
    case status >= 100 && status <= 199:
        return false
    case status == 204:
        return false
    case status == 304:
        return false
    }
    return true

}
```

3) Мы записали байтиков больше, чем сами выставили до этого в заголовке "**Content-Length**":

```
http.HandleFunc("/content-length-err", func(w http.ResponseWriter, r
*http.Request) {
    w.Header().Set("content-type", "application/json")
    w.Header().Set("content-length", "1")
    w.Write([]byte(`{"msg": "OK"}`)) // wrote more than the declared
Content-Length

})
```

4) Мы не смогли записать непосредственно в соединение, например, оно было закрыто ([исходник примера]()):

```
go func() {
    client := &http.Client{Timeout: time.Second}
    resp, err := client.Get("http://localhost:8080")
    if err != nil {
```

```
        // context deadline exceeded (Client.Timeout exceeded while
awaiting headers)
        log.Println("cannot do GET: " + err.Error())
    }
    _ = resp.Body.Close()
}()

http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
    time.Sleep(3 * time.Second) // Обрабатываем запрос три секунды, а
у клиента таймаут в одну.

    data := make([]byte, 1<<20) // 1Mb
    w.Write(data) // write tcp [::1]:8080->[::1]:60756: write: broken
pipe

})
```

LogWriter

Таким образом, мы поняли, что в основном `(http.ResponseWriter).Write` возвращает ошибки, связанные с ошибками разработчика. Их логирование поможет отдебажить ошибку и быстро исправить её.

Но актуально ли это для небольших проектов с примитивной логикой HTTP-обработчиков? – вопрос открытый.

Если же вы всё-таки решили, что *лучше перебздеть, чем недобздеть*, то есть следующие варианты.

Используем функцию-хелпер для логирования ошибки [(исходник примера)]

```
func logWriteErr(_ int, err error) {
    if err != nil {
        log.Println("cannot write response: " + err.Error())
    }
}

// ...
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
        w.Header().Set("content-type", "application/json")
```

```
        logWriteErr(w.Write([]byte(`{"msg": "OK"}`)))
    })

// ...
```

(понятно, что в боевом коде в хелпер скорее всего добавится контекст, ваш продвинутый логер и т.д.)

Используем тип-обёртку для логирования ошибки (исходник примера)

```
type LogWriter struct {
    http.ResponseWriter
}

func (w LogWriter) Write(p []byte) (n int, err error) {
    n, err = w.ResponseWriter.Write(p)
    if err != nil {
        log.Println("cannot write response: " + err.Error())
    }
    return
}

// ...
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
        w.Header().Set("content-type", "application/json")
        LogWriter{w}.Write([]byte(`{"msg": "OK"}`))
    })

// ...
```

Но здесь есть следующие проблемы:

1) Линтеры и IDE всё равно будут ругаться, что мы не обработали ошибку от

```
LogWriter{w}.Write([]byte(`{"msg": "OK"}`))
```

и придётся подтюнивать соответствующие конфигурации.

2) На самом деле настоящая реализация HTTP-ответа (в исходники которой мы заглядывали ранее) реализует большее количество интерфейсов, а не только лишь `http.ResponseWriter`:

```
// A response represents the server side of an HTTP response.
type response struct {
    conn          *conn
```

Type response Implements 10 Interfaces

```
CloseNotifier in net/http/server.go
Flusher in net/http/server.go
Hijacker in net/http/server.go
ReaderFrom in io/io.go
ResponseWriter in net/http/server.go
StringWriter in io/io.go
Writer in io/io.go
http2stringWriter in net/http/h2_bundle.go
requestTooLarger in net/http/request.go
writeFlusher in net/http/httputil/reverseproxy.go
```

```
onse
TP exits
has been (logically) written
response was written
Connection "keep-alive"
has Connection "close"
an atomic int32
an be written
    // writeContinueMu must be held while writing the header.
    // These two fields together synchronize the body reader
    // (the expectContinueReader, which wants to write 100 Continue)
    // against the main writer.
    canWriteContinue atomicBool
    writeContinueMu  sync.Mutex
```

И нашей обёрткой мы ломаем поддержку этих интерфейсов (http.Hijacker, http.Flusher и др.) – подробнее об этом можно почитать здесь – и как следствие её нельзя прокидывать дальше в цепочку функций, принимающих `http.ResponseWriter` (если таковая имеется).

Один из вариантов решения обеих этих проблем – убрать возврат ошибки из `(LogWriter).Write`:

```
func (w LogWriter) Write(p []byte) {
    _, err := w.ResponseWriter.Write(p)
    if err != nil {
        log.Println("cannot write response: " + err.Error())
    }
    return

}
```

так её не нужно будет проверять на уровне выше (а зачем, мы ведь уже это сделали) и обёртка перестанет реализовывать `io.Writer` – как следствие её уже нельзя будет передать в функцию, принимающую `http.ResponseWriter`:

```
func writeOK(w http.ResponseWriter) {
    w.Write([]byte(`{"msg": "OK"}`))
}

// ...
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
        w.Header().Set("content-type", "application/json")
        writeOK(LogWriter{w}) // LogWriter does not implement
http.ResponseWriter (wrong type for Write method)
    })

// ...
```

Задача "JSONWriter"

Ссылка на заготовку.

Вам необходимо реализовать вспомогательный тип `jsonWriter`

```
type jsonWriter struct {
    // ...
}

func (jw jsonWriter) Write(v any) {
    // ...
}
```

который:

- инициализируется логером и `http.ResponseWriter`;
- маршалит входное значение `v` в JSON и записывает его как тело HTTP-ответа, выставляя соответствующий заголовок "**content-type**";
- при ошибке маршалинга или записи логирует ошибку, и отдаёт "**500 Internal Server Error**" с текстом ошибки в теле ответа.

Пример использования типа:

```
type Server struct {
    log      ILogger
    provider IDataProvider
}

func New(l ILogger, d IDataProvider) *Server {
    return &Server{log: l, provider: d}
}

func (s *Server) HandleIndex(w http.ResponseWriter, _ *http.Request)
{
    s.newJSONWriter(w).Write(s.provider.Data())
}

func (s *Server) newJSONWriter(w http.ResponseWriter) jsonWriter {
    return jsonWriter{w: w, log: s.log}
}
```

Подробности в заготовке задачи и тестах.

Тест "Какие ошибки лучше никогда не проверять?"

Выберите все подходящие ответы из списка

- ☐ Ошибку от вызова метода Close реализации интерфейса io.Closer
- ☐ Ошибки от вызовов группы функций fmt.Fprint*
- ☐ Ошибки от вызовов методов io.Reader, io.Writer и подобных
- ☐ Ошибки от вызовов методов и функций пакета net/http