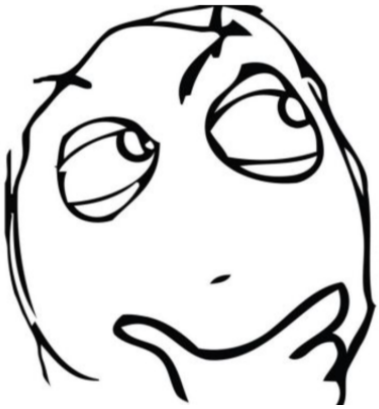
Когда допустимо паниковать?

В этом уроке мы постараемся ответить на этот животрепещущий вопрос.



Don't panic

Как мы помним, одна из заповедей Go гласит – "**Don't panic**". И тут же разработчики Go объясняют её:

See https://golang.org/doc/effective_go.html#errors.

Don't use panic for normal error handling. Use error and multiple return values.

Т.е. в общем случае следует отдать предпочтение возврату ошибки и **не паниковать** в принципе.

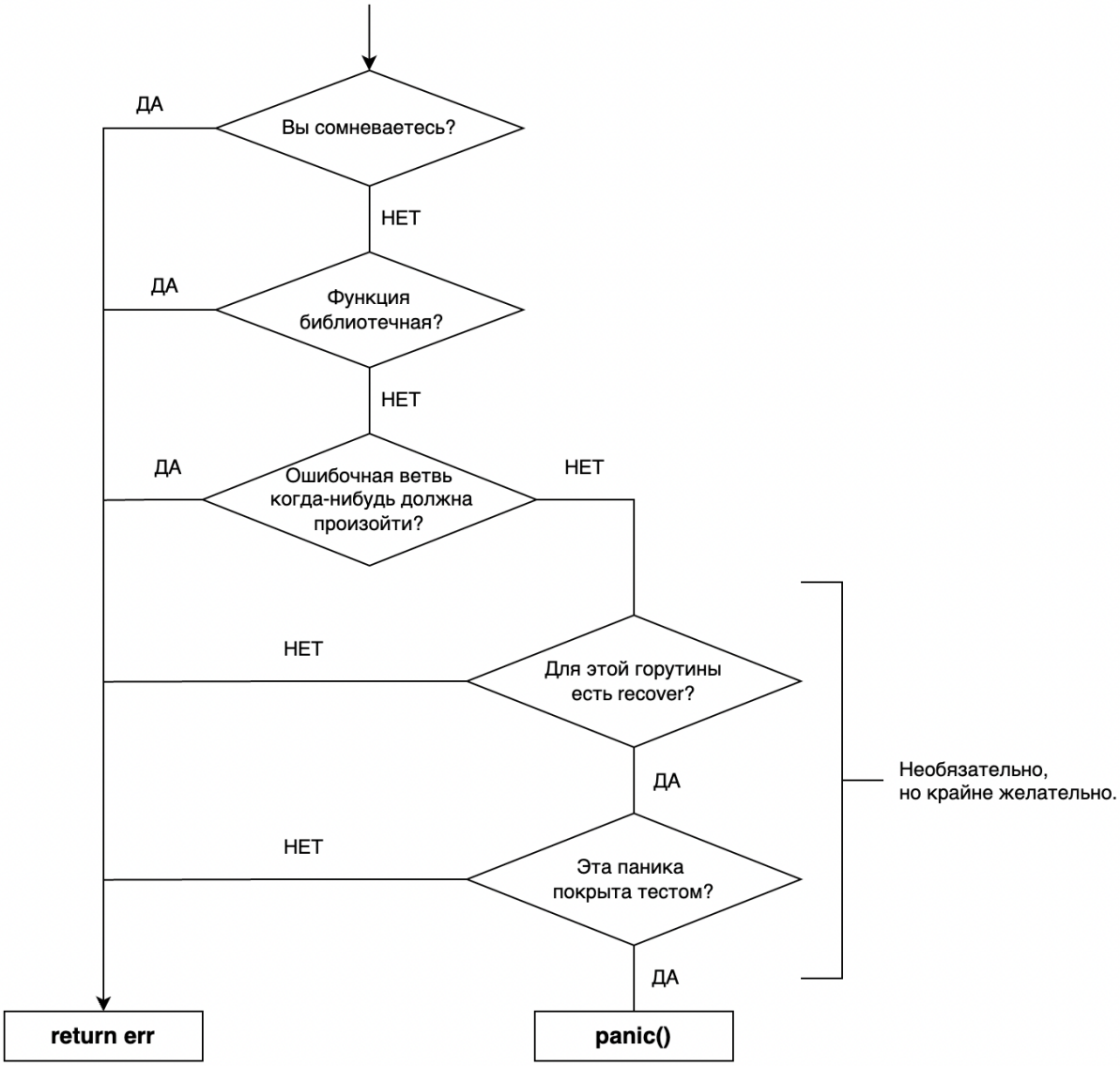
Но при этом сама гошка активно использует панику

```
// https://github.com/golang/go master
$ grep "panic(" -R . | wc -l

    4979
```

Так в чём же дело и всё-таки, когда мы можем паниковать?

Паниковать или не паниковать, вот в чём вопрос

При выборе между паникой или ошибкой можно руководствоваться следующими правилами, которые мы оформили в виде схемы:



Давайте разберём её по шагам.

1) Если вы сомневаетесь, паниковать в своём коде или возвращать ошибку (а вдруг всё-таки упадёт, когда не ждали?!), то возвращайте ошибку. Даже если это породит "лесенку" из её обработки до самого верхнего уровня – это нормально.

---

2) Если вы пишете библиотеку, т.е. модуль, которым будут пользоваться сторонние разработчики, то следует избегать паник. Что достаточно логично и о чём и написано в Effective Go:

```
Real library functions should avoid panic. If the problem can be
masked or worked around,

it's always better to let things continue to run rather than taking
down the whole program.
```

Кому понравится библиотека, которая может невзначай "уронить" работающее приложение (ставьте лайк, если сталкивались с таким)?

---

3) Достижим ли код, в котором вы паникуете? Предполагается, что нет, он никогда не должен выполниться. Это самая частая причина для использования паники – **обнаружение грубых ошибок в реализации задачи, которые можно быстро выявить и исправить.**

Пример из компилятора Go:

```
// src/go/printer/nodes.go

func (p *printer) decl(decl ast.Decl) {
    switch d := decl.(type) {
    case *ast.BadDecl:
        p.print(d.Pos(), "BadDecl")
    case *ast.GenDecl:
        p.genDecl(d)
    case *ast.FuncDecl:
        p.funcDecl(d)
    default:
        // Недостижимый код! Гарантируется, что мы обработали все
возможные ветви.
        panic("unreachable")
    }

}
```

Пример из стороннего приложения:

```
func (h *Handler) Handle(ctx context.Context, r models.Request, w
msg.Writer) error {
    switch h.Type {
    case handlerTypeRequest:
        return h.handleRequest(ctx, r, w)

    case handlerTypeSubscription:
```

```
        return h.handleSubscription(ctx, r, w)

    default:
        // Если мы сюда попали, значит handler имеет неизвестный тип
        // (ни синхронный запрос, ни подписка), и необходимо
исправить это дело.
        panic("invalid handler configuration")
    }
}
```

---

4) Но мы знаем, что согласно Закону Мерфи

```
Anything that can go wrong will go wrong.
```

Поэтому оставляя панику в своём коде, имейте в виду, что когда-нибудь она всё-таки произойдёт. И желательно иметь на этот случай `recover()`.

---

5) Но гарантировать наличие `recover()` сложно, если вы пишите глубоко доменный код, места вызовов которого не так очевидны и необязательно должны находится в одной горутине (как мы помним, `recover()` не спасёт вас от паники в соседней горутинке).

Здесь нам на помощь придут тесты. Протестируйте вашу функцию позитивно, чтобы гарантировать, что она не паникует на всех ожидаемых входных данных.

Тест "В каких случаях паниковать допустимо?"

Выберите все подходящие ответы из списка

- ☐ Паника внутри адаптера доменной модели в доменную модель
- ☐ Паника внутри адаптера входного запроса на транспортном слое
- ☐ Паника при ошибке получения данных в сторадже
- ☐ Паника внутри алгоритма при ситуации, которая может произойти только благодаря ошибке его реализации
- ☐ Паника внутри конструктора типа, предоставляемого вашей библиотекой
- ☐ Паника в любой горутине, отличной от main
- ☐ Паника в недостижимом коде, которая говорит о том, что не все возможные ситуации обработаны, хотя должны
- ☐ Паника при ошибке валидации пользовательского запроса

Задача "Ping"

Задача от одного из наших студентов. Евгений, привет!

Ссылка на заготовку.

В проекте используется вызов метода `Ping` legacy-типа, исходный код которого недоступен нам для изменения:

```
if err := db.Ping(ctx); err != nil {
    return fmt.Errorf("ping: %w", err)

}
```

Проблема в том, что оказалось, что если на момент пингования уже существовало соединение, но оно пропало, то вызов паникует.

Вам необходимо реализовать функцию-обёртку `Ping`, которая защитит нас от ~~нерадивых разработчиков~~ паники, и позволит определять, когда та произошла:

```
var ErrConnectionLost = errors.New("connection lost")

type Pinger interface {
    Ping(ctx context.Context) error
}

// Ping возвращает результат вызова p.Ping(ctx),
// или ErrConnectionLost в случае его паникования.

func Ping(ctx context.Context, p Pinger) error
```

Пример использования:

```
if err := Ping(ctx, db); errors.Is(err, ErrConnectionLost) {
    return db.Reconnect()
} else if err != nil {
    // ...

}
```

Внутри паника, но снаружи ошибка

Статья Defer, Panic, and Recover в блоге Golang содержит следующую интересную фразу:

The convention in the Go libraries is that even when a package uses panic internally,

its external API still presents explicit error return values.

Правда нет ссылки на конкретную *конвенцию* (если не считать ею саму статью в блоге), но стандартная библиотека Go действительно удовлетворяет данному правилу.

Давайте посмотрим на примеры превращения паники в ошибку и прокомментируем их.

Внутри паника, но снаружи ошибка: package fmt

Обратим внимание на одну из внутренних структур, реализующих сканирование в пакете `fmt`:

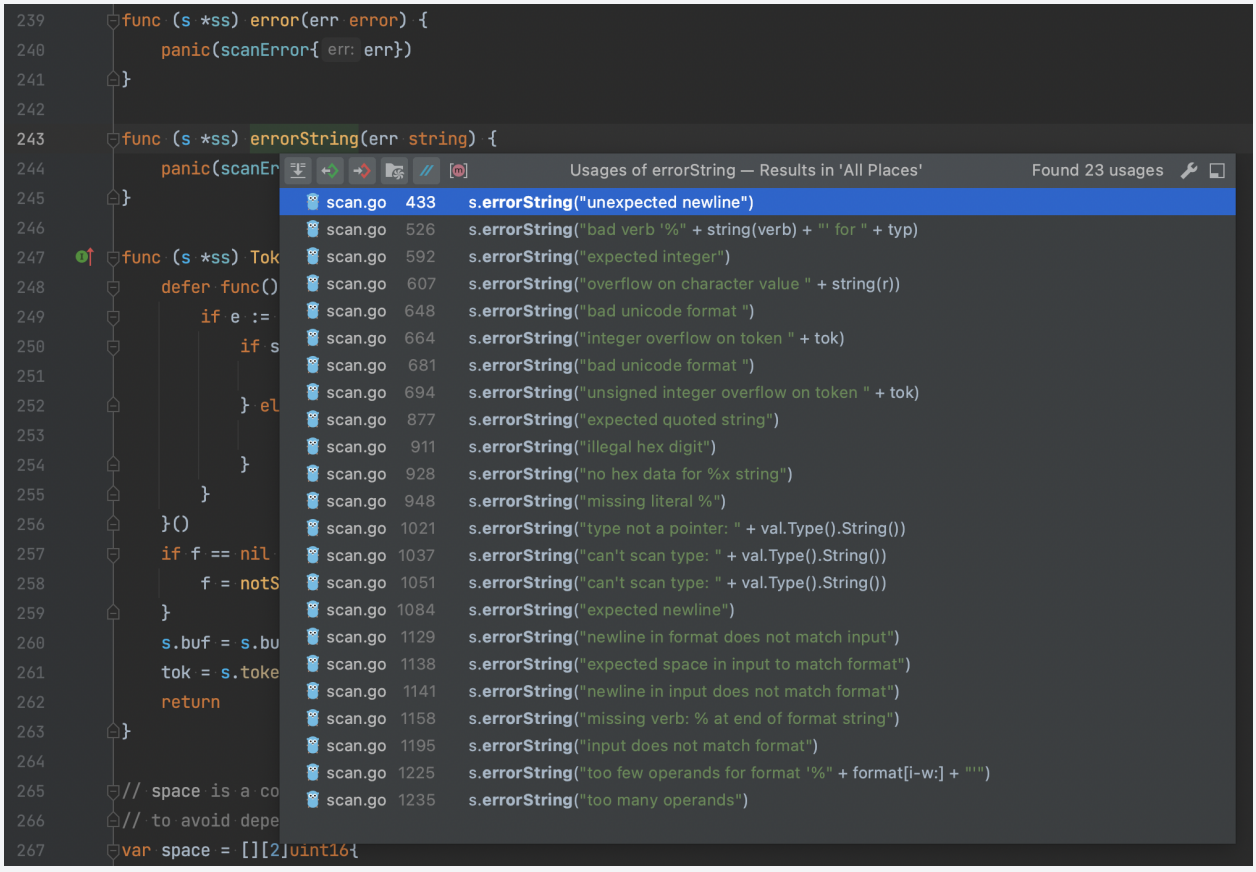
```
// fmt/scan.go
package fmt

import "errors"

// scanError represents an error generated by the scanning software.
// It's used as a unique signature to identify such errors when
recovering.
type scanError struct {
    err error
}
```

```
// ...

// ssave holds the parts of ss that need to be
// saved and restored on recursive scans.
type ssave struct {
    validSave bool // is or was a part of an actual ss.
    nlIsEnd   bool // whether newline terminates scan
    nlIsSpace bool // whether newline counts as white space
    argLimit  int  // max value of ss.count for this arg; argLimit <=
limit
    limit     int  // max value of ss.count.
    maxWid    int  // width of this arg.
}

func (s *ss) Read(buf []byte) (n int, err error) {
    // ...
}

func (s *ss) ReadRune() (r rune, size int, err error) {
    // ...
}

func (s *ss) UnreadRune() error {
    // ...
}

func (s *ss) Width() (wid int, ok bool) {
    // ...
}

func (s *ss) error(err error) {
    panic(scanError{err})
}

func (s *ss) errorString(err string) {
    panic(scanError{errors.New(err)})
}

func (s *ss) Token(skipSpace bool, f func(rune) bool) (tok []byte,
err error) {
    defer func() {
        if e := recover(); e != nil {
            if se, ok := e.(scanError); ok {
                err = se.err
            } else {
```

```
                panic(e)
            }
        }
    }()
    if f == nil {
        f = notSpace
    }
    s.buf = s.buf[:0]
    tok = s.token(skipSpace, f)
    return

}
```

Мы видим хелперы, которые позволяют паниковать из различных мест обработки входного потока символов:

```
func (s *ss) error(err error) {
    panic(scanError{err})                       // Паникуем конкретной приватной ошибкой!
}

func (s *ss) errorString(err string) {
    panic(scanError{errors.New(err)})           // == s.error(errors.New(err))

}
```



А также код в публичном методе получения очередного токена, который ловит "наши" паники и превращает их в возвращаемую ошибку, но при этом не трогает незнакомые паники (чтобы неожиданная исключительная ситуация всё-таки ушла наверх и не была проигнорирована):

```
func (s *ss) Token(skipSpace bool, f func(rune) bool) (tok []byte,
err error) {
    defer func() {
        if e := recover(); e != nil {
            if se, ok := e.(scanError); ok {
                err = se.err
            } else {
                panic(e) // Если что-то незнакомое, то не мешаем
случившейся панике!
            }
        }
    }()
    // ...

}
```

Таким образом, метод `(*fmt.ss).Token` является примером публичного API, внутри себя превращающего панику в возвращаемую ошибку.

Внутри паника, но снаружи ошибка: прочие примеры

encoding/json

Самый популярный пример, который как раз приведён в блоге Go:

```
For a real-world example of panic and recover, see the json package
from the Go standard library.
It encodes an interface with a set of recursive functions. If an
error occurs when traversing the value,
panic is called to unwind the stack to the top-level function call,
which recovers from the panic and returns

an appropriate error value (see the 'error' and 'marshal' methods of
the encodeState type in encode.go).
```

```
// encoding/json/encode.go
package json

// jsonError is an error wrapper type for internal use only.
// Panics with errors are wrapped in jsonError so that the top-level
recover
// can distinguish intentional panics from this package.
type jsonError struct{ error }

func (e *encodeState) marshal(v interface{}, opts encOpts) (err
error) {
    defer func() {
        if r := recover(); r != nil {
            if je, ok := r.(jsonError); ok {
                err = je.error
            } else {
                panic(r)                        // Обращаем внимание на
repanic!
            }
        }
    }()
    e.reflectValue(reflect.ValueOf(v), opts)
    return nil
}
```

```
// error aborts the encoding by panicking with err wrapped in
jsonError.
func (e *encodeState) error(err error) {
    panic(jsonError{err})

}
```

encoding/gob

```
// encoding/gob/error.go
package gob

// A gobError is used to distinguish errors (panics) generated in
this package.
type gobError struct {
    err error
}

// errorf is like error_ but takes Printf-style arguments to
construct an error.
// It always prefixes the message with "gob: ".
func errorf(format string, args ...interface{}) {
    error_(fmt.Errorf("gob: "+format, args...))
}

// error wraps the argument error and uses it as the argument to
panic.
func error_(err error) {                    // Интересный хак по
неймингу!
    panic(gobError{err})
}

// catchError is meant to be used as a deferred function to turn a
panic(gobError) into a
// plain error. It overwrites the error return of the function that
deferred its call.
func catchError(err *error) {
    if e := recover(); e != nil {
        ge, ok := e.(gobError)
        if !ok {
            panic(e)                        // Обращаем внимание на
repanic!
        }
```

```
        *err = ge.err
    }
}
```

Пример вызовов `catchError` и `errorf`:

```
// encoding/gob/decode.go
package gob

// compileDec compiles the decoder engine for a value.
func (dec *Decoder) compileDec(remoteId typeId, ut *userTypeInfo)
(engine *decEngine, err error) {
    defer catchError(&err)
    // ...
}

// decodeInterface decodes an interface value and stores it in value.
func (dec *Decoder) decodeInterface(ityp reflect.Type, state
*decoderState, value reflect.Value) {
    nr := state.decodeUint()
    if nr > 1<<31 {
        errorf("invalid type name length %d", nr)
    }
    // ...

}
```

net/http

```
// net/http/server.go
package http

// ErrAbortHandler is a sentinel panic value to abort a handler.
// While any panic from ServeHTTP aborts the response to the client,
// panicking with ErrAbortHandler also suppresses logging of a stack
// trace to the server's error log.
var ErrAbortHandler = errors.New("net/http: abort Handler")

// Serve a new connection.
func (c *conn) serve(ctx context.Context) {
    c.remoteAddr = c.rwc.RemoteAddr().String()
```

```
    ctx = context.WithValue(ctx, LocalAddrContextKey,
c.rwc.LocalAddr())
    defer func() {
        if err := recover(); err != nil && err != ErrAbortHandler {
            const size = 64 << 10
            buf := make([]byte, size)
            buf = buf[:runtime.Stack(buf, false)]
            c.server.logf("http: panic serving %v: %v\n%s",
c.remoteAddr, err, buf)
        }
        if !c.hijacked() {
            c.close()
            c.setState(c.rwc, StateClosed, runHooks)
        }
    }()
    // ...

}
```

Сервер рвёт соединение при любой панике в вашем обработчике, но при этом позволяет избежать логирования ошибки на стороне сервера, если вы паникуете ошибкой `ErrAbortHandler`:

```
// ...
    if err := recover(); err != nil && err != ErrAbortHandler {
        // ...
        c.server.logf("http: panic serving %v: %v\n%s", c.remoteAddr,
err, buf)
    }
    if !c.hijacked() {
        c.close()
        // ...
    }

// ...
```

Вот такие хаки!

В стандартной библиотеке и компиляторе Go есть ещё ряд примеров "программирования через панику".

Все они имеют схожие черты:

- наличие собственного приватного *ошибочного* типа;
- наличие (в различных местах) паникования значением данного типа (читай – приватной ошибкой);
- наличие обработчика с `recover()`, превращающего приватную ошибку в публичную или продолжающего панику далее.

Какие плюсы и минусы вы видите в данном подходе 🙃?

Тест "Внутри паника, но снаружи ошибка"

Назовите любой пакет стандартной библиотеки кроме рассмотренных, который внутри использует панику в качестве управляющей конструкции (для передачи наверх, в специальный обработчик, какого-либо значения, необязательно ошибки).

Формат ответа – import path пакета без кавычек:

`net/http`

Напишите текст

_____

Задача "htmlparser.Parse"

Ссылка на заготовку.

Перед вами функция `Parse`, задача которой уметь парсить простой HTML вида

```
<h1>Go Proverbs</h1><p>Don't panic!</p>
```

```
// Parse разбирает HTML из идущих подряд заголовков первого уровня и/или параграфов.
func Parse(html []byte) (tags []Tag, err error) {
    defer catchErr(&tags, &err)
```

```
    // Во время работы функции могут произойти вызовы throwErr.
    // ...
}

type parseError struct {
    error
}

func catchErr(tags *[]Tag, err *error) {
    // ...
}

func throwErr(err error) {
    panic(parseError{error: err})

}
```

Вам необходимо:

- реализовать пару приватных функций, используемых в `Parse`, чтобы она заработала и начала удовлетворять тестам;
- реализовать `catchErr`.

Больше подробностей см. в заготовке задачи и тестах.

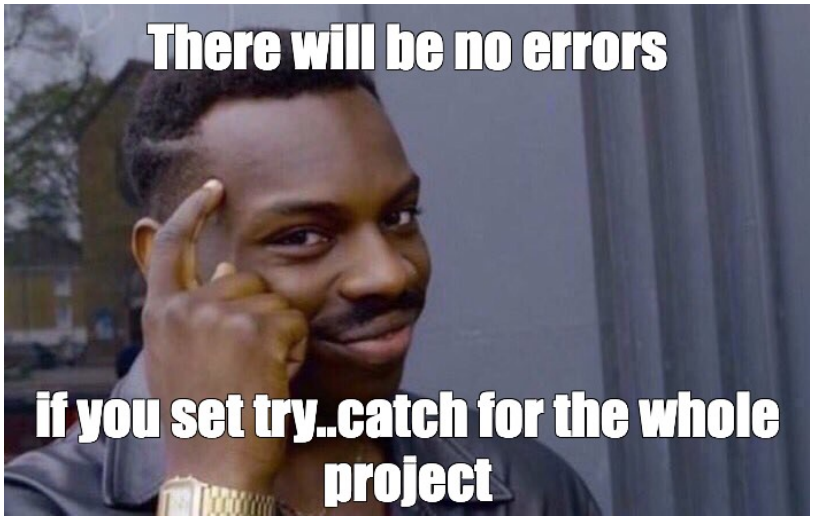
*** Задание со звёздочкой**

Удалите весь исходный код `Parse` и попробуйте реализовать её самостоятельно.

**** Задание двумя звёздочками**

Подумайте, что и как нужно изменить, чтобы `Parse` поддерживала вложенные друг в друга теги.

panic-defer-recover – это аналог throw-try-catch?



В каком-то роде да, но на самом деле абсолютно нет 🙂.

Создатели Golang хотели строго разделить

- *нормальные ошибки* **aka errors** (ошибки валидации данных, открытия файла, закрытия соединения и т.д. – то, что можно и нужно обработать и двигаться дальше);

- *по-настоящему исключительные ситуации* **aka exceptions** (ошибка программирования алгоритма; невалидная конфигурация приложения, с которой оно не должно запуститься и т.д. – то, что при желании можно обработать, но скорее всего завершение процесса неизбежно).

Поэтому работа с паникой не была реализована через управляющие конструкции на уровне синтаксиса языка.

Приведём пару важных цитат на этот счёт:

This proposal differs from the usual model of exceptions as a control structure, but that is a deliberate decision. ***We don't want to encourage the conflation of errors and exceptions*** *that occur in languages such as Java.*

*Instead, this proposal uses ... a couple of runtime functions to provide **a clean mechanism for handling truly exceptional conditions**.*

(c) [Proposal for an exception-like mechanism, Rob Pike]

*We believe that coupling exceptions to a control structure, as in the `try-catch-finally` idiom, **results in convoluted code**. It also tends to encourage programmers to label too many **ordinary errors**, such as failing to open a file, as exceptional.*

*Go takes **a different approach**. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. A canonical error type, coupled with Go's other features, **makes error handling pleasant but quite different from that in other languages**.*

*Go also has a couple of built-in functions to signal and recover **from truly exceptional conditions**. The recovery mechanism is executed only as part of a function's state being torn down after an error, which is sufficient to handle catastrophe but requires no extra control structures and, **when used well, can result in clean error-handling code**.*

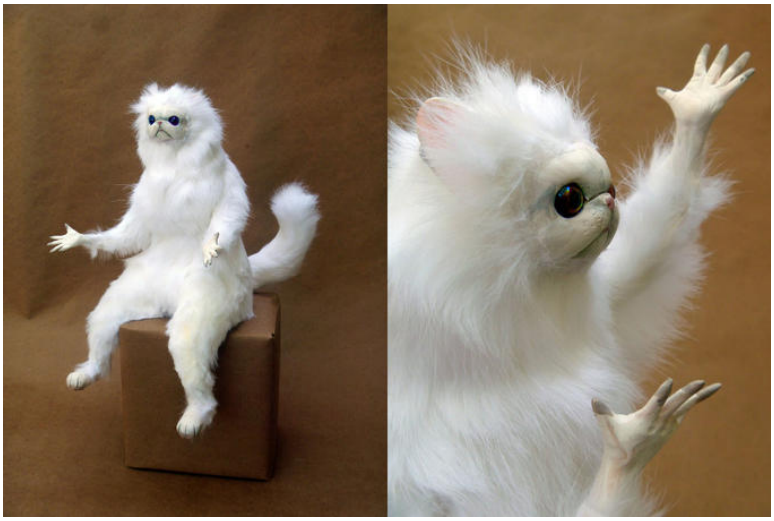
(c) [Why does Go not have exceptions?, FAQ]

Исходя из этого, становится очевидно, что не стоит пренебрегать ошибками в пользу паник и использовать `panic-defer-recover` в качестве `throw-try-catch`, даже если вы только пришли из другого языка и вас тошнит от повсеместного `err != nil`.

В следующем шаге мы обсудим данный вопрос чуть подробнее.

Программирование через панику – это ОК?

Сначала мы говорим о **"Don't panic"**, о том, что в общем случае паниковать – это плохо, а затем приводим примеры из стандартной библиотеки, где сами разработчики Go реализуют свой фунционал через паники. Так что же делать, кому верить?



Во-первых, ещё раз отметим, что чаще всего внутренние паники разработчики Go превращают во внешнюю ошибку.

Во-вторых, несмотря на примеры из стандартной библиотеки, мы всё-таки не рекомендуем так делать. Банально не следует использовать инструменты не по их прямому назначению: это может привести к сильносвязному коду, а в худшем случае – к багам.

Сведём в табличку известные нам плюсы и минусы программирования через ошибки или через паники:

	`return err`	`panic(newLocalError("..."))`
Плюсы	• Простой в использовании механизм (на первый взгляд). • Просто объяснить.	• В общем случае ускоряет программу. • "Уплотняет" код, увеличивает его читаемость.
Минусы	• Шаблонный, повторяющийся код. • Много "шума" при чтении и сопровождении кода. • Пункты выше иногда приводят к ошибкам написания `err == nil` вместо `err != nil`, `return nil` вместо `return err` и т.д. • В общем случае замедляет программу.	• Сложнее объяснить. • Сложнее аргументировать причины использования. • Сбивает с толку новичков. • Увеличивает связность кода (нельзя переиспользовать паникующую функцию as is, с ней придётся тащить обработчик с `recover()`). • Требует концентрации и правильного использования `defer`. • Требует эмпирического анализа, на каком уровне уже пора отказываться от паники и превращать её в возвращаемую ошибку?
Чем нивелируются минусы?	• Тестами. • Линтерами. • Плагинами для IDE, скрывающими блоки `err != nil`. • В 99% случаев вряд ли обработка ошибок будет bottleneck'ом приложения.	• Тестами. • Написанными единожды и переиспользуемыми хелперами. • Красноречием.

Пишите в комментарии, что думаете на этот счёт и каким практикам следуете вы в своих проектах.

Выводы

Мы узнали, когда (несмотря на все запреты) всё-таки полезно паниковать. Более того, посмотрели на примеры использования паники в качестве фундамента для реализации алгоритма работы функции.

Но всё же:

- Не стоит делать панику частью своего публичного API.
- Не стоит паниковать в экспортируемых вашей библиотекой функциях.
- Не стоит паниковать, если сердце подсказывает, что тут лучше подойдёт возврат ошибки.

Когда и как применять те или иные практики (пусть даже самые экзотические), как всегда – решать вам!