Error Handling

В этом уроке мы поговорим о болях обработки ошибок и о том, что было предпринято, чтобы попытаться нивелировать их.

Постановка проблемы

Ссылка на примеры.

Чтобы наше небольшое приложение на Go могло без проблем масштабироваться до большой кодовой базы, мы должны писать легкий для понимания, надёжный код; избегать излишнего повторения и не забывать про обработку ошибок.

Давайте представим, что мы живём в мире, где в Go есть исключения и напишем простую программу для копирования файла (несуществующий синтаксис):

```
func CopyFile(src, dst string) throws error {
    r := os.Open(src)
    defer r.Close()

    w := os.Create(dst)
    io.Copy(w, r)
    w.Close()

}
```

Красота, не правда ли? :)

Но на самом деле в коде выше кроется баг – при возникновении ошибки в `io.Copy` или `w.Close` мы должны удалить созданный файл `w`.

Перепишем программу в более привычный нам вид:

```
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return err
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer w.Close()

    if _, err := io.Copy(w, r); err != nil {
        return err
    }

    return w.Sync()

}
```

Вся простота испарилась, и более того новый вариант всё равно ошибочен и имеет следующие недостатки:

- файл `dst` все ещё не удаляется при ошибках внутри функции;
- мы возвращаем "голые" ошибки, пренебрегая врапингом и усложняя отладку этой функции;
- не обрабатываются ошибки отложенных методов `Close`.

Это случилось, потому что обработка ошибок занимает много места и при ревью кода мы неосознанно пропускаем эти блоки, чтобы поскорее вычленить структуру основного кода.

Учтём замечания выше и напишем новую версию программы:

```
func CopyFile(src, dst string) (err error) {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %q to %q: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %q to %q: create dst file: %v", src,
dst, err)
    }
    defer func() {
        if err := w.Close(); err != nil {
            log.Printf("copy %q to %q: cannot close dst file: %v",
src, dst, err)
        }

        if err != nil {
            if err := os.Remove(dst); err != nil {
                log.Printf("copy %q to %q: cannot remove dst file :
%v", src, dst, err)
            }
        }
    }()

    if _, err := io.Copy(w, r); err != nil {
        return fmt.Errorf("copy %q to %q: cannot do io.Copy: %v",
src, dst, err)
    }

    if err := w.Sync(); err != nil {
        return fmt.Errorf("copy %q to %q: cannot sync dst file %v",
src, dst, err)
    }

    return nil

}
```

Программа выросла "не по дням, а по часам" – она стала *более правильной*, но вся лёгкость и элегантность пропала :(

Что хочется получить?

В Go 2 разработчики языка хотят упростить проверку ошибок, уменьшить количества кода, связанного с ней и тем самым сократить объем текста программ на Go в целом.

Хочется сделать обработку ошибок более удобной, чтобы замотивировать программистов как можно меньше игнорировать это дело. При этом и проверка ошибок и их обработка должны оставаться явными, т.е. видимыми в тексте программы.

Конечно, не стоит забывать про обратную совместимость, и новые механизмы обработки ошибок не должны противоречить существующему коду.

if err != nil

try / check

Почему просто не перейти к исключениям?

Go solves the exception problem by not having exceptions. (c) Dave Cheney

Разработчики Go уверены, что наличие в языке конструкций вида `try-catch-finally` приводит к запутанному коду, а также побуждает программистов считать некоторые ошибки исключительными ситуациями, хотя те по своей природе ими не являются (например, ошибка открытия файла).

Кроме того, было замечено, что механизм исключений позволяет располагать обработку ошибки далеко от источника ошибки, что побуждает людей не думать о так называемом **error path**. Когда же нам приходится задумываться об ошибке при каждом вызове, мы в конечном итоге думаем о её пути, что приводит к более качественному коду.

Например, так бы выглядело использование `CopyFile`, если бы в Go были исключения:

```
try {
    CopyFile("src.txt", "dst.txt")
} catch (
    FileOpenErr,
    CreateFileErr,
    IOCopyErr,
    FileCloseErr,
) as err {
  log.Println(err)

}
```

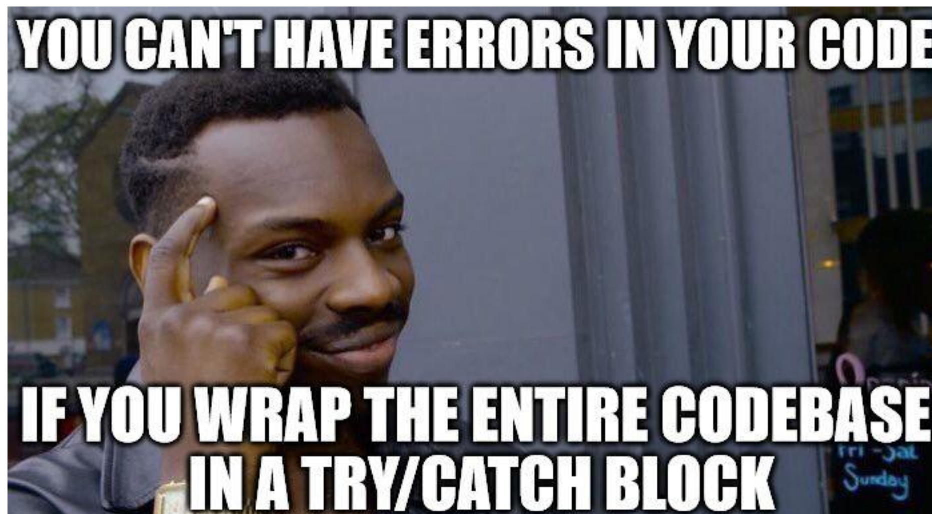
Или даже так:

```
try {
    CopyFile("src.txt", "dst.txt")
} catch GeneralException as err {
    log.Println(err)

}
```

А в худшем случае вообще так (единственный `try-catch` на самом верхнем уровне приложения):

```
try {
    main() // CopyFile где-то глубоко внутри.
} catch GeneralException as err {
    log.Println(err)

}
```

В большой системе может быть сложно отследить, какие функции порождают исключения и какие именно. При этом, если вы решите более аккуратно и выборочно работать с исключениями, то скорее всего придёте к коду, внешне похожему на работу с ошибками в Go.



Тема отсутствия исключений в языке по-настоящему больная и может сравниться разве что с дженериками. Мы не будем подробно разбирать доводы за и против (обращаем ваше внимание на литературу данного модуля), а примем как факт: обработка ошибок в том виде, что есть, может не нравится разработчикам, привыкшим к другим языкам (C++, Java, Python и т.д.), но то, что многие считают большим недостатком Go, на самом деле является его достоинством: **ошибки – это значения**, а работа с ними проста, удобна и унифицирована от проекта к проекту.

Команда Golang не ставит перед собой цели завезти исключения, но они пытаются сделать механизм обработки ошибок ещё более удобным и всё-таки не занимающим столько программного места (читай – добавить синтаксического сахарку).

Какие есть идеи на этот счёт? Узнаем дальше.

[ABANDONED] check & handle API

Со временем появилось много предложений по улучшению обработки ошибок в Go.

В их числе:

- golang.org/issue/21161: упрощение обработки ошибки с помощью суффикса `|| err`:

```
func Chdir(dir string) error {
    syscall.Chdir(dir) || &PathError{"chdir", dir, err}
    return nil

}
```

- golang.org/issue/18721: добавление оператора "must" в виде `#`:

```
result := #foo(1,2,3)

// Эквивалент для:

result, err := foo(1,2,3)
if err != nil {
    return 0, "", person{}, err // Если функция, внутри которой мы
находимся, возвращает ошибку.
}

result, err := foo(1,2,3)
if err != nil {
    panic(err) // Если функция, внутри которой мы находимся, не
возвращает ошибку.

}
```

- golang.org/issue/21182: убрать мусор из `return`, если большая его часть – это **zero values**:

```
if err != nil {
    return ..., err // Многоточие - валидный синтаксис.

}
```

- golang.org/issue/19991: добавление встроенного типа `result` (как в Rust или OCaml):

```
func f1(arg int) result<int, error> {
    return result.Ok(value)
    // ...
    return result.Err(error)

}
```

и многие-многие другие.

Посмотрев на всё это дело и почитав отзывы (к слову, вы тоже при желании можете набросить там своих мнений), разработчики Go предложили ввести два новых ключевых слова – `check` и `handle`:

- `check` позволяет проверять вызов функции на ошибку, если ошибка случается, то она сама пробрасывается наверх, проходя через заданный явно `handle` или существующий по умолчанию обработчик;

- `handle` позволяет определить блок, являющийся обработчиком для ошибок, обнаруженных `check`'ами. Обработчиков может быть несколько, тогда они будут составлять **handler chain** и вызываться в порядке, обратном определению, пока очередной из обработчиков не будет содержать `return`.

Посмотрим на примере!

```
// Было.

func printSum(a, b string) error {
    x, err := strconv.Atoi(a)
    if err != nil {
        return fmt.Errorf("printSum(%q + %q): %v", a, b, err)
```

```
    }

    y, err := strconv.Atoi(b)
    if err != nil {
        return fmt.Errorf("printSum(%q + %q): %v", a, b, err)
    }

    fmt.Println("result:", x+y)
    return nil
}

// Стало.

func printSum(a, b string) error {
    handle err {
        return fmt.Errorf("printSum(%q + %q): %v", a, b, err)
    }
    x := check strconv.Atoi(a)
    y := check strconv.Atoi(b)
    fmt.Println("result:", x + y)
    return nil

}
```

Выглядит неплохо, а попробуем на нашей монструозной [CopyFile из начала урока]:

```
func CopyFile(src, dst string) (err error) {
    handle err {
        return fmt.Errorf("copy %q to %q: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    handle err {
        return fmt.Errorf("copy %q to %q: create dst file: %v", src,
dst, err)
    }
    w := check os.Create(dst)
    defer func() {
        if err := w.Close(); err != nil {
            log.Printf("copy %q to %q: cannot close dst file: %v",
src, dst, err)
        }
```

```
        if err != nil {
            if err := os.Remove(dst); err != nil {
                log.Printf("copy %q to %q: cannot remove dst file :
%v", src, dst, err)
            }
        }
    }()

    handle err {
        return fmt.Errorf("copy %q to %q: cannot do io.Copy: %v",
src, dst, err)
    }
    check io.Copy(w, r)

    handle err {
        return fmt.Errorf("copy %q to %q: cannot sync dst file %v",
src, dst, err)
    }
    check w.Sync()

    return nil

}
```

Лучше точно не стало. Более того, мы теперь как магистр Йода программируем задом наперёд. Всё дело в том, что мы каждую операцию врапили в уникальный текст, что породило большое количество хендлеров. Давайте попробуем закрыть это глаза и оставить только префикс `"copy %q to %q"`:

```
func CopyFile(src, dst string) (err error) {
    handle err {
        return fmt.Errorf("copy %q to %q: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    defer func() {
        if err := w.Close(); err != nil {
            log.Printf("copy %q to %q: cannot close dst file: %v",
src, dst, err)
        }

        if err != nil {
```

```
            if err := os.Remove(dst); err != nil {
                log.Printf("copy %q to %q: cannot remove dst file :
%v", src, dst, err)
            }
        }
    }()

    check io.Copy(w, r)
    check w.Sync()
    return nil

}
```

Кажется, стало чуть лучше, но `defer`'ы портят нам жизнь. При этом записать вот так нельзя, потому что это будет логически неверно:

```
w := check os.Create(dst)
defer func() {
    hande err {
        log.Printf("copy %q to %q: %v", src, dst, err)
    }

    check w.Close()
    check os.Remove(dst)

}()
```

Как вариант, можно попробовать отказаться и от логирования:

```
func CopyFile(src, dst string) (err error) {
    handle err {
        return fmt.Errorf("copy %q to %q: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    defer func() {
        _ = w.Close()

        if err != nil {
            _ = os.Remove(dst)
        }
    }()
```

```
    check io.Copy(w, r)
    check w.Sync()
    return nil

}
```

Стало ли лучше относительно самого первого варианта? Решать вам!

[ABANDONED] Встроенная функция try

В результате обсуждения предыдущего решения было установлено, что ключевое слово `handle` слишком неочевидно и запутанно перекрывалось с `defer`.

Поэтому задачи `handle` решили переложить на `defer`, а ключевое слово `check` преобразовать во встроенную функцию `try`:

```
func try(expr) (T1, T2, … Tn)
```

- если случается ошибка, то `try` назначает её возвращаемому аргументу с типом `error` и выходит из функции;
- иначе `try` осуществляет обычное присваивание результатов соответствующим переменным.

```
// Было.

func foo() error {
    a, b, err := bar()
    if err != nil {
        return err
    }
    // ...
}

// Стало.

func foo() error {
    a, b = try(bar())
    // ...
}
```

Автором данного предложения является "мастодонт", которого мы упоминали в самом начале курса – **Роберт Гризмер**.

Перепишем знакомую нам `printSum`:

```
func printSum(a, b string) error {
    fmt.Println("result:", try(strconv.Atoi(a)) +
try(strconv.Atoi(b)))
    return nil

}
```

Выглядит совсем неплохо. А что насчёт злосчастной `CopyFile`?

```
func CopyFile(src, dst string) (err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("copy %q to %q: %v", src, dst, err)
        }
    }()

    r := try(os.Open(src))
    defer r.Close()

    w := try(os.Create(dst))
    defer func() {
        _ = w.Close()

        if err != nil {
            _ = os.Remove(dst)
        }
    }()

    try(io.Copy(w, r))
    try(w.Close())
    return nil

}
```

Выглядит тоже неплохо, только мы как и в случае `handle-check` отказались от логирования ошибок в defer и специфичного текста при врапинге каждой ошибки.

В результате длительного обсуждения (~800 комментариев, Карл! Попробуйте как-нибудь почитать перед сном) proposal был **отклонён**.

Подробное заключение можно почитать в сообщении от самого Роберта Гризмера, но Расс Кокс пишет:

We spent most of June discussing this proposal publicly on GitHub.

The fundamental idea of check or try was to shorten the amount of syntax repeated at each error check, and in particular to remove the return statement from view, keeping the error check explicit and better highlighting interesting variations. One interesting point raised during the public feedback discussion, however, was that without an explicit if statement and return, there's nowhere to put a debugging print, there's nowhere to put a breakpoint, and there's no code to show as unexecuted in code coverage results. The benefits we were after came at the cost of making these situations more complex. On balance, from this as well as other considerations, it was not at all clear that the overall result would be simpler Go development, so we abandoned this experiment.

Т.е. в погоне за синтаксическим сахаром мы получили следующие проблемы. Без явного `if-return`:

- нам некуда добавить дебажных принтов;
- нам некуда поставить точку останова;
- и нет кода, который можно пометить как непокрытый (выделено красным) в отчётах о покрытии (**coverage reports**):

```go
func (idx Index) LoadPost(title string) (*PostSpec, string, error) {
	postFolder := strings.Replace(strings.ToLower(title), " ", "_", -1)
	spec, err := idx.getSpec(postFolder)
	if err != nil {
		return nil, "", err
	}
	f, err := os.Open(path.Join(idx.blogdir, postFolder, "content.html"))
	if err != nil {
		return nil, "", err
	}
	defer f.Close()

	b, err := ioutil.ReadAll(f)
	if err != nil {
		return nil, "", err
	}
	return spec, string(b), nil
}
```

Этих доводов стало достаточно, чтобы решения выше также не вышли в свет.

Тест "Какие фичи появились в языке, начиная с Go 1.14?"

Выберите все подходящие ответы из списка

- ☐ Built-in тип `result`
- ☐ Ключевые слова `handle` и `check`
- ☐ Built-in функция `try`
- ☐ Ключевые слова `try` и `catch`
- ☐ Обработка ошибок с помощью `|| err`

- [ ] Оператор `noerror` (аналог `noexcept` из C++)

Что было дальше?

Мы имеем на руках уйму информации о том, что разработчики хотели получить в Go 2, что в итоге получилось, а что нет.

Но за последние пару лет (момент окончания курса выпал на конец 2021), к сожалению, мы не наблюдаем новостей по поводу новых инструментов работы с ошибками в Go.

Пишите в комментариях, когда мимо вас проскочит что-то интересненькое на данную тему.

А в целом нам остаётся только ждать :(