(не|ну)жная монада Either на практике и в теории

Артём Кобзарь и Дмитрий Махнёв

Disclaimer

Материал для лиц достигших 21 года,
т.к. содержит сцены статической типизации.
Доклад не стремится продать FP
или оскорбить кого-либо.
Все совпадения случайны.
Материал безусловно дискуссионный.
Докладчики не компетентны.

Артём

Дмитрий

Demo

изучаем проект

Обработка исключений

Обработка исключений мечты

1. Отделение исключений от ошибок;
2. Идентичная обработка исключений для синхронного и асинхронного кода;
3. Заглядывать в исключение без блокирования потока исключений;
4. Типизация исключений;
5. Шоб код красивый был!

Аристократичный lookup

```
try {
  doSome();

} lookup (e) {
  logError(e);
}
```

Обработка исключений мечты

1. Отделение исключений от ошибок;
2. Идентичная обработка исключений для синхронного и асинхронного кода;
3. Заглядывать в исключение без блокирования потока исключений;
4. Типизация исключений;
5. Шоб код красивый был!

Рабоче-крестьянский try/catch

```
try {
  doSome();

} catch (e) {
  processError(e);
}
```

HOLY JS/...

2019 Piter

Ruben Bridgewater
Freelance Software Architect

Error handling: doing it right!

youtu.be/bJ3glfA-jqo

Demo

try/catch и типизированные ошибки

Плюсы try/catch

1. Единый стиль обработки синхронного и асинхронного кода.

Проблемы try/catch

1. Громоздкость (вложенность, instanceof, return без await);
2. Проблемы проксирования (съедание ошибок);
3. Не различаем ошибки и исключения;
4. В обработчике нет типа исключения;
5. Не оптимизируется для старых версий V8;
6. Нет описания исключений, которые может выбросить метод (как в java).


```
public void checkAuthorization() throws AuthorizationError {
  // ...
}
```

Что делать?

Demo

callback style для попытки типизации

Плюсы callback style

1. Появились типы исключений;
2. Появились описания исключений, которые может выбросить метод;
3. Раздельная обработка ошибок и исключений;
4. Натягивается единый стиль обработки синхронных и асинхронных исключений.

Проблемы callback style

1. Громоздкость (if != null);
2. callback hell 🤬;
3. Необходимость стыковки с Promise based API;
4. Необходимость в создании обёрток на низком уровне;
5. Нужно иметь договорённости (линтеры) внутри команды.

Что делать?

```go
func main() {
    file, err := os.Open("file.txt")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Print(file)
}
```

Demo

Go Style

Плюсы Go Style

1. Менее громоздко (нет callback);
2. Появились типы исключений;
3. Появились описания исключений, которые может выбросить метод;
4. Раздельная обработка ошибок и исключений;
5. Сводимо к единой обработке синхронных и асинхронных исключений.

Проблемы Go Style

1. Громоздкость (if error !== null, [undefined]);
2. Шумные tuple типы;
3. Необходимость в создании обёрток на низком уровне;
4. Нужно иметь договорённости (линтеры) внутри команды.

Что делать?

Demo

Promise + Ok and Failed (убираем всё в значение)

Плюсы Promise + Ok and Failed

1. Появились типы исключений;
2. Появились описания исключений, которые может выбросить метод;
3. Раздельная обработка ошибок и исключений;
4. Менее громоздко;
5. Сводимо к единой обработке синхронных и асинхронных исключений.

Проблемы Promise + Ok and Failed

1. Громоздкость (if instanceof, new Ok/Failed);
2. Много ручного контроля состояния;
3. Проблемы проксирования (съедание исключений/данных) (ну очень не безопасно!);
4. Необходимость в создании обёрток на низком уровне;
5. Нужно иметь договорённости (линтеры) внутри команды.

Ну очень не безопасно

```
async checkAuthorization(): Promise<Ok<void> | Failed<AuthorizationError>> {
  const user = await this.userService.getUser();
  if (user instanceof Failed) {
    return new Failed(new AuthorizationError("Can't authorize user"));
  }
}
```

Проблемы Promise + Ok and Failed

1. Громоздкость (if instanceof, new Ok/Failed);
2. Много ручного контроля состояния;
3. Проблемы проксирования (съедание исключений/данных) (ну очень не безопасно!);
4. Необходимость в создании обёрток на низком уровне;
5. Нужно иметь договорённости (линтеры) внутри команды.

Секундочку!

```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem opening the file: {:?}", error)
        },
    };
}
```

```haskell
main :: IO ()
main = do
       input <- try $ readFile "hello.txt"
       case input of
         Right file -> print file
         Left e -> error $ "Problem opening the file: " ++ e
```

tver
\o/

Монада — это моноид в категории эндофункторов. Ясно?

youtu.be/IkXg_mjNgG4

prime group
информационные технологии

tver \o/

Обобщение работает!

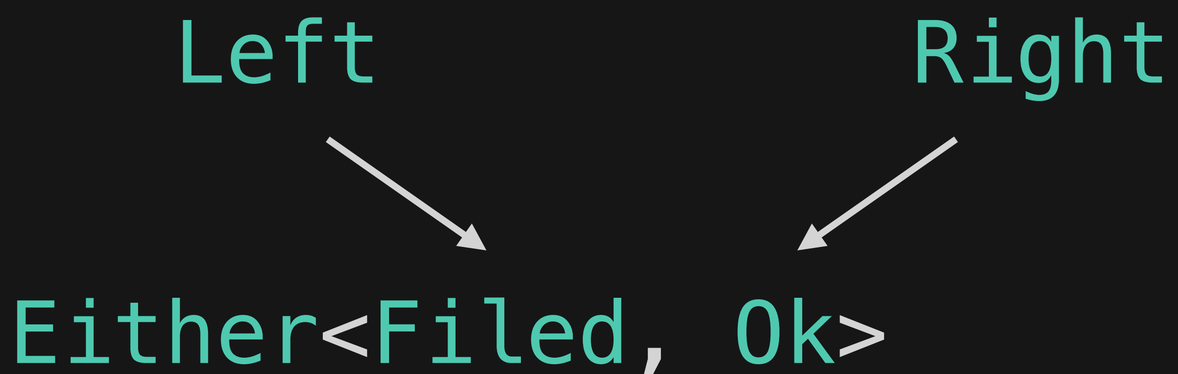
```
p1();
p2();
p3();
```

```
p1 >>= (\ _ -> p2)
   >>= (\_ -> p3)
```

youtu.be/IkXg_mjNgG4

prime group информационные технологии

```
export class Either<Filed, Ok> implements Monad<Ok>
```


Left
Right
Either<Filed, Ok>

Demo

Either monad

Плюсы Either

1. Минималистично (и очень расширяемо);
2. Контроль состояний инкапсулирован;
3. Есть типы исключений;
4. Появились описания исключений, которые может выбросить метод;
5. Раздельная обработка ошибок и исключений;
6. Сводимо к единой обработке синхронных и асинхронных исключений.

Проблемы Either

1. Возможно команде придётся посмотреть этот доклад;
2. Необходимость в создании обёрток на низком уровне;
3. Нужно иметь договорённости (линтеры) внутри команды;

Production ready

Demo

Prod cases

DOTNEXT 2019 Piter

Роман Неволин

Revolut

Почему ваша архитектура функциональная и как с этим жить

youtu.be/9s_4wpzENhg

Минуточку!


```ts
async function getConferenceData(): Promise<Either<Error1 | Error2, [Speaker[], Talk[]]>> {

  const speakers: Either<Error1, Speaker[]> = await getSpeakers();
  const talks: Either<Error2, Talk[]> = await getTalks();

  return Either.mergeInOne(speakers, talks);

}
```


```ts
async function getConferenceData(): Promise<Either<Error1 | Error2, [Speaker[], Talk[]]>> {

  const speakers: Either<Error1, Speaker[]> = await getSpeakers();
  const talks: Either<Error2, Talk[]> = await getTalks();

  return Either.mergeInOne(speakers, talks);

}
```

```
Promise.all([
  getSpeakers(),
  getTalks(),
]);
```

Demo

Either multiple requests

Проблемы Either

1. Возможно команде придётся посмотреть этот доклад;
2. Необходимость в создании обёрток на низком уровне;
3. Нужно иметь договорённости (линтеры) внутри команды;
4. **Много ручной работы при множестве ошибок.**

Demo

Boosted Either multiple requests

Плюсы Boosted Either

1. Минималистично (**и очень расширяемо**);
2. Контроль состояний инкапсулирован;
3. Есть типы исключений;
4. Появились описания исключений, которые может выбросить метод;
5. Раздельная обработка ошибок и исключений;
6. Сводимо к единой обработке синхронных и асинхронных исключений.

Проблемы Boosted Either

1. Возможно команде придётся посмотреть этот доклад;
2. Необходимость в создании обёрток на низком уровне;
3. Нужно иметь договорённости (линтеры) внутри команды;

Выводы

1. Монад не надо боятся;
2. Монаду можно использовать без глубокого знания мат части;
3. Монаду можно использовать.

Спасибо за ваше время

Ссылки

1. Наши примеры
   https://github.com/DmitryMakhnev/HolyJS-2019-Moscow

2. Монадки
   npmjs.com/package/@sweet-monads/either

undefined и @dmitrymakhnev