OZON tech

Позаботимся о памяти при использовании Value-типов

Алексей Таран

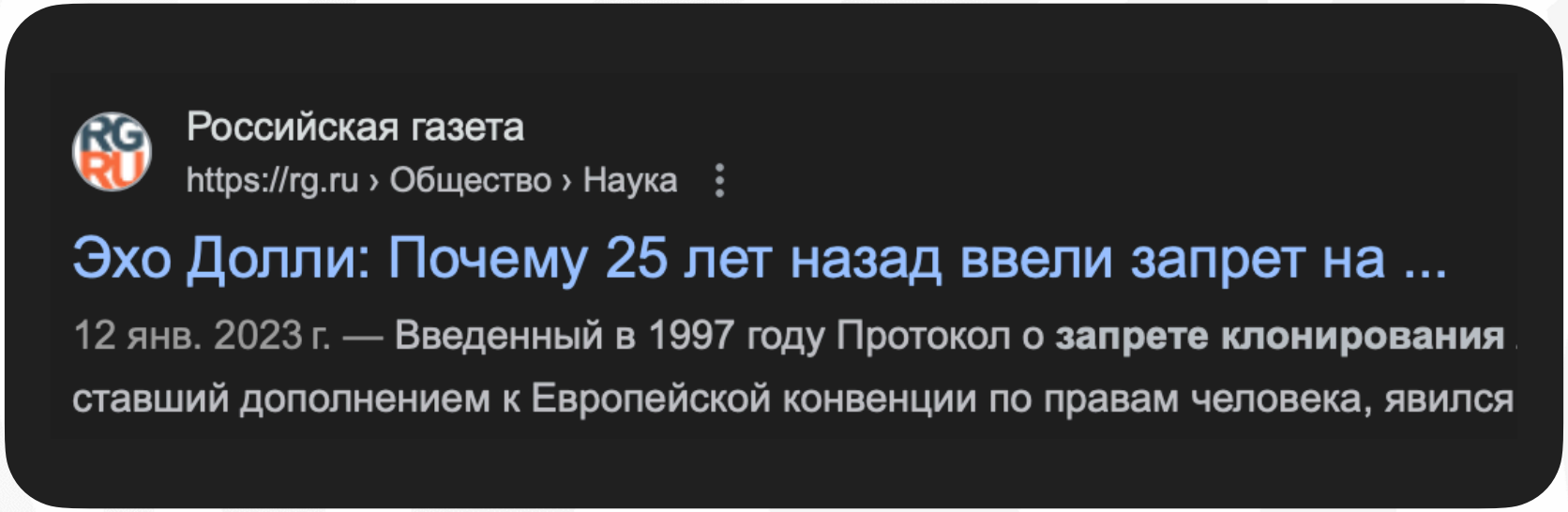
Ozon Seller iOS, руководитель группы

Оптимизации работы с памятью

Хотя бы для Value-типов…

Ты — Value-тип
и логически существуешь
в единственном числе?

Может, запретим тебе клонироваться?



Тем более, нельзя

Ты — Value-тип
и логически существуешь
в единственном числе?

Может, запретим тебе клонироваться?

Ненужные «сору» уберём?

Память сохраним?

А что, если закончить твой lifetime раньше scope?

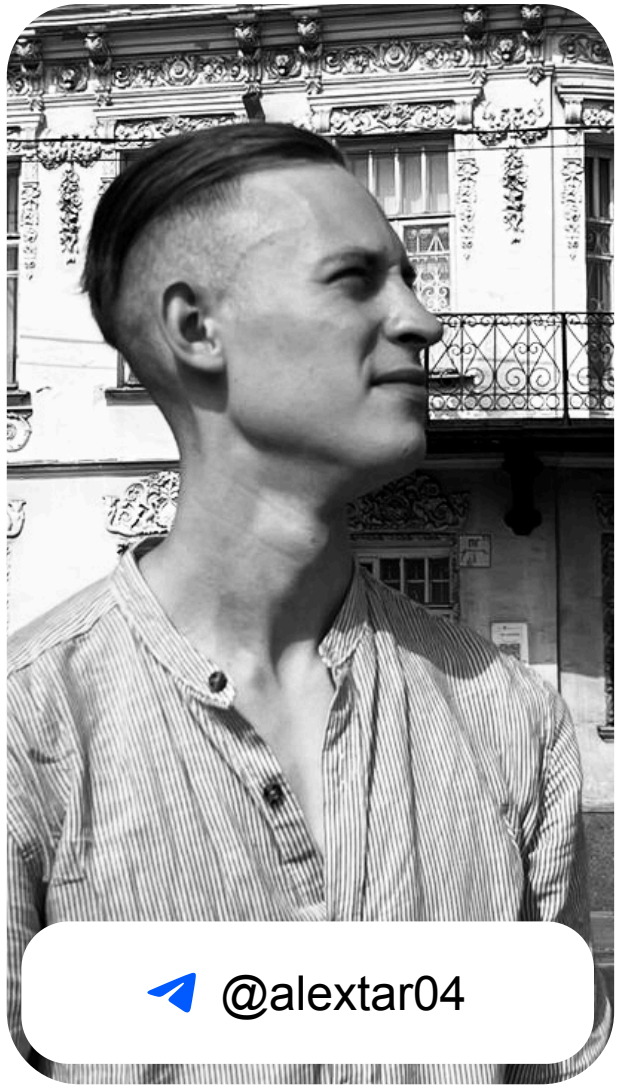
И здесь экономия памяти?

Погрузимся в некопируемые типы!

Хотим экономить память

Конкретно обозначать свои намерения в жизни

*по отношению к инстансу

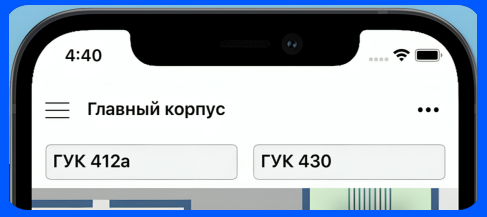


Алексей Таран

Ozon Seller iOS, руководитель группы

FinTech

ВТБ и другие банки,
которых нет в AppStore :(



Indoor-навигация

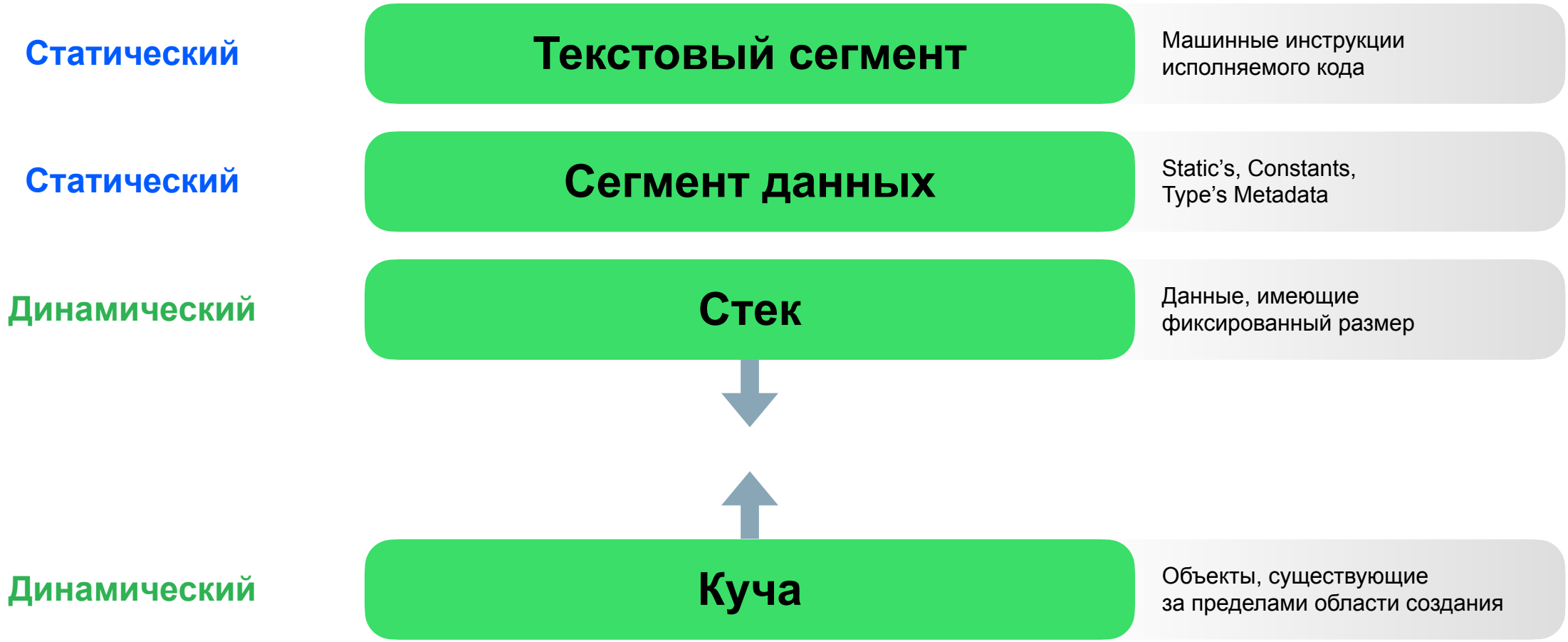
AR Plants Care

OzonSeller App

AGENDA

- Обсудим стоимость «копирования»
- Область применения «некопируемости»
- Идеи, позаимствованные из других технологий
- Эволюция «некопируемых» типов в Swift
- Опыт применения NonCopyable’s в Ozon Seller
- Выводы и потенциал развития

Memory Management в Swift



Стек. Каждый поток имеет собственный стек

Как выделить память под такой инстанс?

```
struct SomeStruct {
  let value: Bool = true
  let value1: Bool = true
}

// 2
print(MemoryLayout<SomeStruct>.size)
```

Стек. Каждый поток имеет собственный стек

```
func someFunc() {
  let value = SomeStruct()
}

// SP - указатель на вершину стека

... <+84>:  bl    0x100ebb074     ; SomeStruct.init()
... <+88>:  strh  w0, [sp, #0x16] ; // Сохранение в SP + 0x16 параметра из w0
... <+92>:  ldrb  w9, [sp, #0x16] ; // Прочитать значение по адресу SP + 0x16 и записать в w9
... <+96>:  ldrb  w8, [sp, #0x17] ; // Прочитать значение по адресу SP + 0x17 и записать в w8

// Всего прочитано из стека: 2 байта
```

Стек. Каждый поток имеет собственный стек

```
func someFunc() {
  let value = SomeStruct()
}

// SP - указатель на вершину стека

... <+84>:  bl    0x100ebb074     ; SomeStruct.init()
... <+88>:  strh  w0, [sp, #0x16] ; // Сохранение в SP + 0x16 параметра из w0
... <+92>:  ldrb  w9, [sp, #0x16] ; // Прочитать значение по адресу SP + 0x16 и записать в w9
... <+96>:  ldrb  w8, [sp, #0x17] ; // Прочитать значение по адресу SP + 0x17 и записать в w8

// Всего прочитано из стека: 2 байта
```

Дёшево!

А если заменить **«Struct»** на **«Class»**?

Куча. Shared’ся между потоками

- Фрагментация памяти
- Heap Object Header
- Reference Count
- Isa-Pointer

```
class SomeClass {
  let value: Bool = true
  let value1: Bool = true
}

// Размер указателя x64 == 8 байт.
print(MemoryLayout<SomeClass>.size)

...
... <+104>:  bl   0x100e333b0  ; SomeClass.__allocating_init() -> SomeClass
...
```

Куча. Shared’ся между потоками

- Фрагментация памяти
- Heap Object Header
- Reference Count
- Isa-Pointer

```
class SomeClass {
  let value: Bool = true
  let value1: Bool = true
}

// Размер указателя x64 == 8 байт.
print(MemoryLayout<SomeClass>.size)

...
... <+104>:  bl   0x100e333b0  ; SomeClass.__allocating_init() -> SomeClass
...
```

Недёшево!

Так и **где**
хранить уникальные
неклонируемые
объекты?

В стеке? В куче?

Стек

Value — тип

Например, **Struct** или **Enum**

🚩 **Проблемы:**

Копирование из коробки

Ссылочный — тип

Например, **Class** или **Actor**

🚩 **Проблемы:**

Оверхэды использования кучи

Может быть, исправим проблемы Value-типа?

Но как?

Может быть, исправим проблемы Value-типа?

? Запретим объекту копироваться?

Подобьём потенциальные кейсы

01

Клонирование базового ресурса невозможно логически

02

Можно обойтись без копирования

Файлы

Аппаратные периферийные устройства (например, камера)

Одноразовые объекты для взаимодействия различных API

Объекты чтения/записи в хранилища

Отказываемся от копирований

=

Помогаем памяти?

Подсмотрим **идейки**

Такс, такс, такс…

Resource Acquisition Is Initialization (RAII)

Например: **C++**

Region-based memory management

Например: **Cyclone**

Linear Types

Например: **Idris**

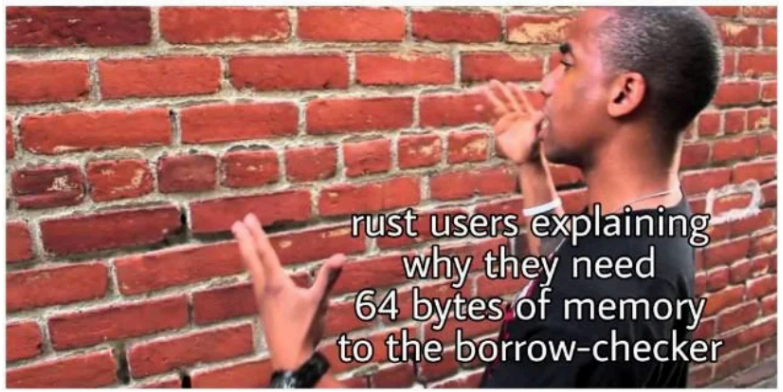
Parrot Concurrency System

Например: **Perl**

Borrow Checker!

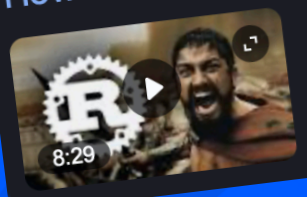
Rust Borrow Checker!

2006



YouTube · Let's Get Rusty
40.8K+ views · 9 months ago

How to fight Rust's borrow checker... and win.



Getting bombarded with compile time errors is all too common when writing Rust. Today we'll learn how to fight **borrow checker** errors and win ...

Reddit · r/rust
90+ comments · 1 year ago

Is the Rust Borrow Checker Really That Challenging?

Взглянем на Rust!

Скомпилируется ли такой код?

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{s1}, world!");
}
```

Операция «Передача права собственности»

Move

Нет :(

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    /// Compile time ошибка.
    /// Право владения на ссылку "s1" перешло к "s2".
    /// Неявная операция "Move".
    println!("{s1}, world!");
}
```

Операция **«Заём»** строки

Borrowing

А так?

```rust
fn main() {
    let s = String::from("hello");
    change(&s);
    println!("{s}");
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Операция **«Заём»** строки

Borrowing

❓ Ещё одна неудача

```rust
fn main() {
    let s = String::from("hello");
    change(&s);
    println!("{s}");
}

fn change(some_string: &String) {
    // Ошибка - модифцировать переданное значение по ссылке - нельзя!
    some_string.push_str(", world");
}
```

Операция «Изменение по ссылке»

Inout

Хоть что-то :)

```rust
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Влияние неявного «*Move*» на конструкции языка

```rust
fn main() {
    let mut v = vec![1, 2, 3];

    for i in v {
        println!("{}", i);
    }

    // Ошибка. For-in забрал "v" через "move"
    println!("{v:?}");
}
```

А где лучше?

```rust
fn main() {
    let mut v = vec![1, 2, 3];

    for i in v {
        println!("{}", i);
    }

    // Ошибка. For-in забрал "v" через "move"
    println!("{v:?}");
}
```

Rust

```swift
func main() {
    let v = [1, 2, 3]

    for i in v {
        print(i)
    }

    // Ок. For-in забрал "v" через неявный borrow
    print(v)
}
```

Swift

Влияние неявного «*Borrow*» на конструкции языка

```rust
fn main() {
    let mut v = vec![1, 2, 3];

    for i in &v {
        println!("{}", i);
    }

    // Ок. For-in забрал "v" через неявный "borrow"
    println!("{v:?}");
}
```

Итого, имеем

Borrow

Гоша занял монетки на пиццу

Итого, имеем

Borrow

Гоша занял монетки на пиццу

Inout

Гоша передал повару деньги.
Повар вернул сдачу

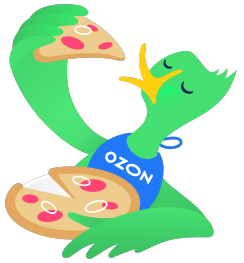
Итого, имеем

Borrow

Гоша занял монетки на пиццу

Inout

Гоша передал повару деньги.
Повар вернул сдачу



Consume

Повар отдал пиццу Гоше
на поглощение

Swift!

Swift 5.9

09/2023

Появление «некопируемых» Struct и Enum

```
struct SomeStruct: ~Copyable {
  private var fd: Int32
}
```

Swift 5.9

2023

Отсутствие возможности соответствия протоколам (кроме Sendable)

- Modern concurrency
- ABI

Swift 5.9

2023

Новые ключевые слова:

- consume
- consuming
- borrowing

Consume == Rust’s Move

```
func someFunc(_: consuming [Int]) {}

let values = [1, 2, 3, 4, 5]
for value in consume values {
  print(values)
}

// Ошибка. 'values' used after consume
someFunc(values)
```

Swift 5.9

2023

У некопируемых типов появился деструктор!

```swift
struct SomeStruct: ~Copyable {
  var field: Int32

  consuming func someFunc() {...}

  deinit {
    print("Deinit")
  }
}

func run() {
  let someStructInstance = SomeStruct(...)
  someStructInstance.someFunc()
  // Вывод "deinit"
  // 'someStructInstance' - недоступен

  let someStructInstance1 = SomeStruct(...)
  // Вывод "deinit"
}
```

Swift 6.0

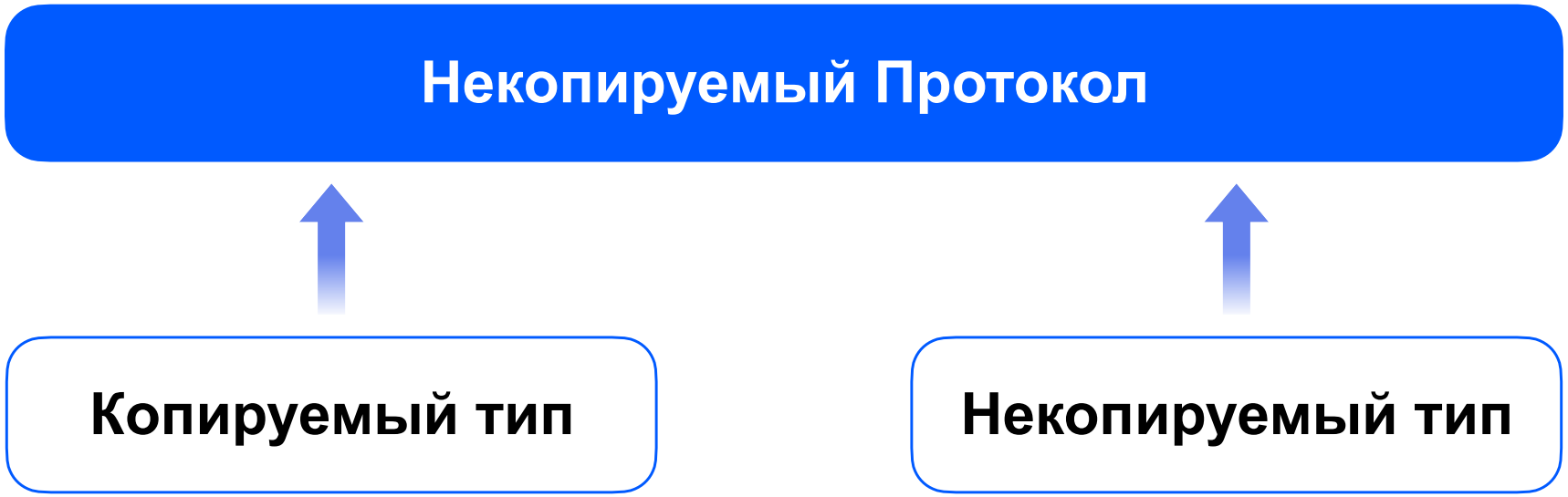
09/2024

Тип: некопируемый протокол

«~Copyable» — возможность, а не ограничение!

```
protocol Resource: ~Copyable {...}

// Ок. Некопируемая структура
struct SomeStruct: Resource, ~Copyable {}

// Ок. Копируемая структура
struct SomeStruct1: Resource {}
```

Что значит **«возможность»**?



Что значит «**возможность**»?

Swift 6.0

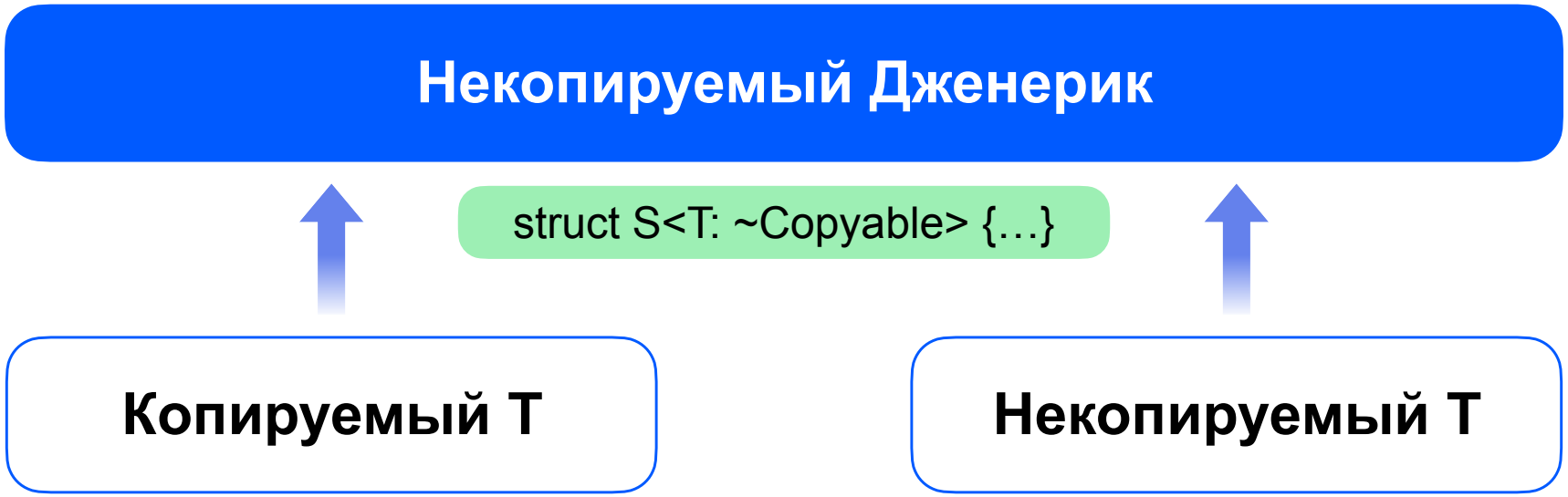
2024

Поддержка дженериков!

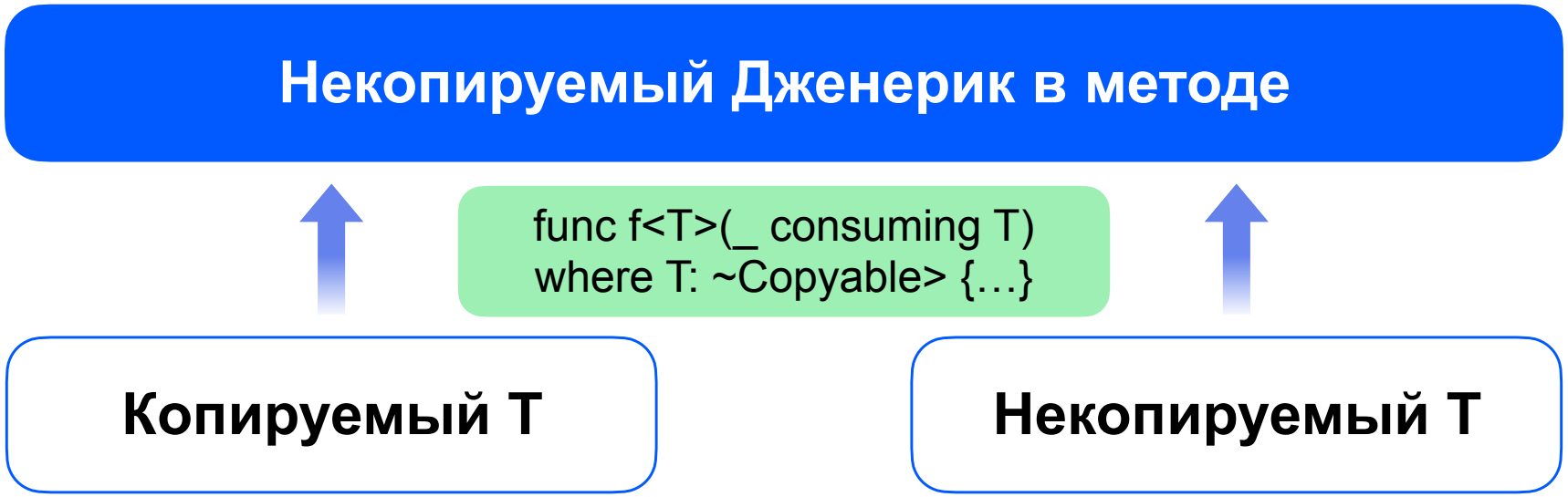
Подавляем «Copyable» на каждом шаге конформанса!

```
protocol Resource: ~Copyable {}

struct SomeResource: Resource, ~Copyable {}
struct SomeStruct<T: Resource & ~Copyable>: ~Copyable {}

struct SomeSimpleStruct{}

struct Client {
    func f<T>(_: consuming T) where T: ~Copyable {}

    func someFunc() {
        let instance = SomeStruct<SomeResource>()
        f(instance)

        let instance1 = SomeSimpleStruct()
        f(instance1)
    }
}
```

Что значит **«возможность»**?



Что значит **«возможность»**?



Основной Proposal

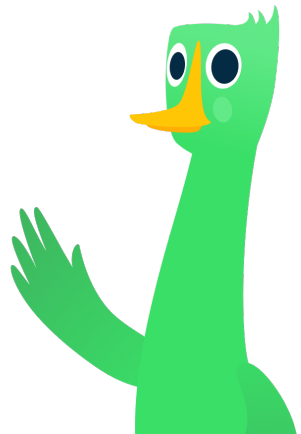
Гоша	Rust	Swift	RU
	Borrow	Borrow	Занять монетки
	Inout	Inout	Передать оплату и получить сдачу
	Move	Consume	Получить пиццу и съесть

Практика!

Классический пайплайн сетевого взаимодействия

01

Формируем запрос



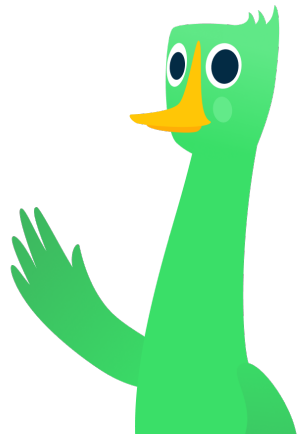
Классический пайплайн сетевого взаимодействия

01

Формируем запрос

02

Совершаем запрос
в сеть



Классический пайплайн сетевого взаимодействия

01

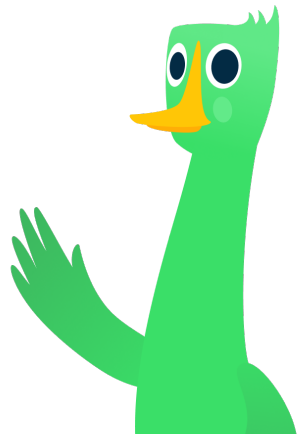
Формируем запрос

02

Совершаем запрос в сеть

03

Обрабатываем полученные данные



Запрос
(Request)

Не хотим, чтобы объект запроса мог быть использован после передачи в сетевой провайдер

```
struct OzonRequest: Encodable {...}
struct OzonResponse: Decodable {...}
struct OzonModel {...}

struct OzonService: IOzonService {
    func fetchData(_ request: OzonRequest) -> OzonResponse {
        return OzonResponse(...)
    }
}

struct Client {
    private let service: IOzonService

    init(...) {...}

    func someFunc() {
        let request = OzonRequest(...)
        let response = service.fetchData(request)
        let mappedResponse = map(response)
    }

    private func map(_ response: OzonResponse) -> [OzonModel] {
        response....map {...}
    }
}
```

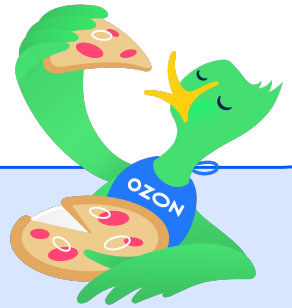
Запрос
(Request)

Добавляем ~Copyable
к структуре запроса
иииии…

```
// "Noncopyable struct 'OzonRequest' cannot conform to 'Encodable'"
struct OzonRequest: Encodable, ~Copyable {
    let payload: String
}
```

Запрос
(Request)

Решение 1:
потребить запрос!




```
func someFunc() {
    let request = OzonRequest(...)
    let response = service.fetchData(consume request)

    // Ошибка: 'request' used after consume
    // print(request.payload)
    let mappedResponse = map(response)
}
```

Запрос
(Request)

Решение 2:
попробовать реализовать некопируемый Wrapper :)

```
struct OzonRequestWrapper<T: Encodable>: ~Copyable {
    private let _value: T

    init(_ value: T) {
        self._value = value
    }

    var wrapped: T {
        consuming get { _value }
    }
}
```

Запрос
(Request)

Решение 2:
и использовать его :)

```
struct OzonService: IOzonService {
    func fetchData(_ request: consuming OzonRequestWrapper<OzonRequest>) -> ... {
        // Доступ к содержимому тела через request.wrapped.someField
        // return ...
    }
}

struct Client {
    private let service: IOzonService

    init(...) {...}

    func someFunc() {
        let request = OzonRequestWrapper(OzonRequest(...))
        let response = service.fetchData(request)

        // 'request' used after consume
        print(request.wrapped)
        ...
    }
  ...
}
```

Ответ
(Response)

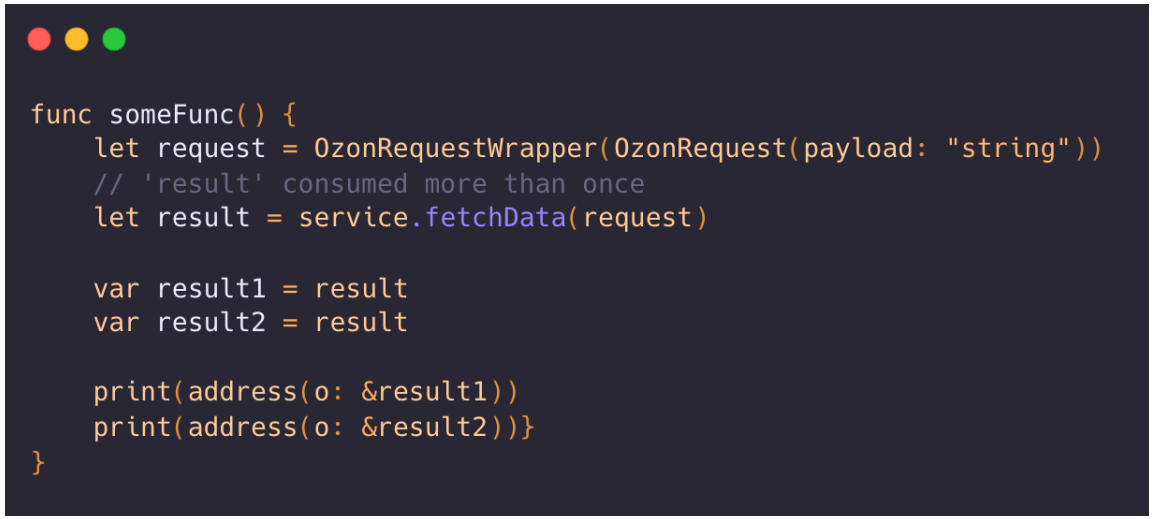
Не хотим, чтобы большой массив данных имел возможность быть скопированным

```
func someFunc() {
    let request = OzonRequestWrapper(OzonRequest(payload: "string"))
    let result = service.fetchData(request)

    var result1 = result
    var result2 = result

    // 6099660432
    print(address(o: &result1))
    // 6099660416
    print(address(o: &result2))}
}
```

```
struct OzonResponseWrapper<T: Decodable>: ~Copyable {
    private let _value: T

    init(_ value: T) {
        self._value = value
    }

    var wrapped: T {
        consuming get { _value }
    }
}
```

Ответ
(Response)

Теперь «result»
не копируется!



```
func someFunc() {
    let request = OzonRequestWrapper(OzonRequest(payload: "string"))
    // 'result' consumed more than once
    let result = service.fetchData(request)

    var result1 = result
    var result2 = result

    print(address(o: &result1))
    print(address(o: &result2))}
}
```

Mapping

Хотим, чтобы после маппинга доступ к данным был логически ограничен

```
struct Client {
    private let service: IOzonService

    init(...) {...}

    func someFunc() {
        let request = OzonRequestWrapper(OzonRequest(...))
        let response = service.fetchData(request)
        let mappedResult = map(response)
    }

    private func map(_ result: consuming OzonResponseWrapper<OzonResponse>) -> ... {
      ...
    }
  ...
}
```

Also

Массив и некопируемые типы

```
// Ошибка. Type "OzonResponseEnum" does not conform to protocol 'Copyable'
struct OzonResponse: ~Copyable {
  let values: [OzonResponseEnum]
}

struct OzonResponseEnum: ~Copyable {
  case one
  case two
  case three
}
```

AssociatedType элемента массива должен быть копируемым!

Мост GCD — ModernConcurrency

Пишем async/ await обёртку для «GCD-like» SDK

```
final class OzonMapsFacade... {
    private init() {}
    static let shared = OzonMapsFacade()

    func geocode(
        query: String
        completion: @escaping (Result<GeocodeResponse, Never>) -> Void
    ) {
        ...
    }
}

struct Client... {
    func geocode(query: String) async -> CLLocationCoordinate2D? {
        await withUnsafeContinuation { value in
            OzonMapsFacade.shared.geocode(query: query) { result in
                switch result {
                case .success(let value):
                    continuation.resume(returning: value.coordinate)
                case .failure:
                    continuation.resume(returning: nil)
                }
            }
        }
    }
}
```

Проблемы

You must call a «resume» method exactly once on every execution path throughout the program!!!

CheckedContinuation совершает проверку в рантайме

UnsafeContinuation не добавляет оверхэд проверки, но небезопасен

А что, если добавить Compile-Time проверку для UnsafeContinuation?

Хотим вызывать .resume(…) — лишь раз!

Не добавим :(

```
struct UnsafeContinuation<T, E> where E : Error
```

continuation копируемый…

И неспроста…

```
struct OzonUnsafeContinuation<T, E> : Sendable, ~Copyable where E: Error {...}

class Client {
    let server = Server()

    func someMethod() async -> ... {
        await withOzonUnsafeContinuation { continuation in
            server.get { value in
                // Noncopyable 'continuation' cannot be consumed when captured by an escaping closure
                continuation.resume(returning: value)
                ...
            }
        }
    }
}
```

Решение проблемы:
ещё одна обёртка:)

```
struct OzonUnsafeContinuation<T>: ~Copyable {
    private let wrapped: UnsafeContinuation<T, Never>

    init(wrapped: UnsafeContinuation<T, Never>) {
        self.wrapped = wrapped
    }

    consuming func resume(returning: T) {
        wrapped.resume(returning: returning)
    }
}
```

Вернёмся к примеру:

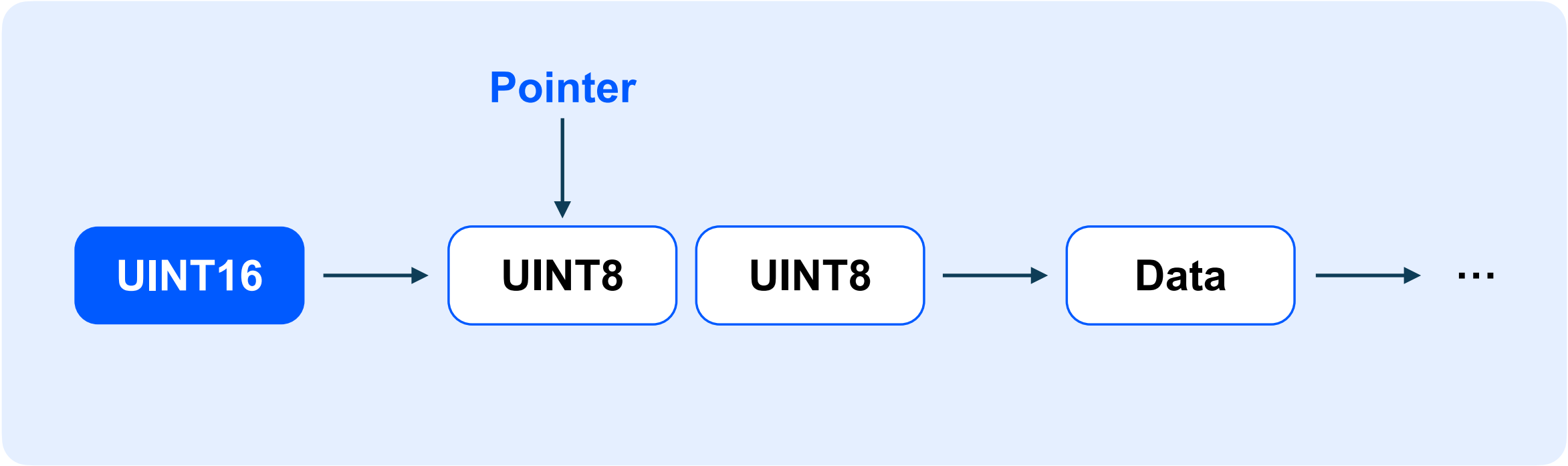
WIN :)

```swift
struct Client {
    func geocode(query: String, language: String) async -> CLLocationCoordinate2D? {
        await withUnsafeContinuation { value in
            OzonMapsFacade.shared.geocode(
                query: query
            ) { result in
                let continuation = OzonUnsafeContinuation(wrapped: value)

                switch result {
                case .success(let response):
                    continuation.resume(returning: response.coordinate)
                    // 'continuation' consumed more than once
                    // continuation.resume(returning: nil)
                case .failure:
                    continuation.resume(returning: nil)
                }
            }
        }
    }
}
```

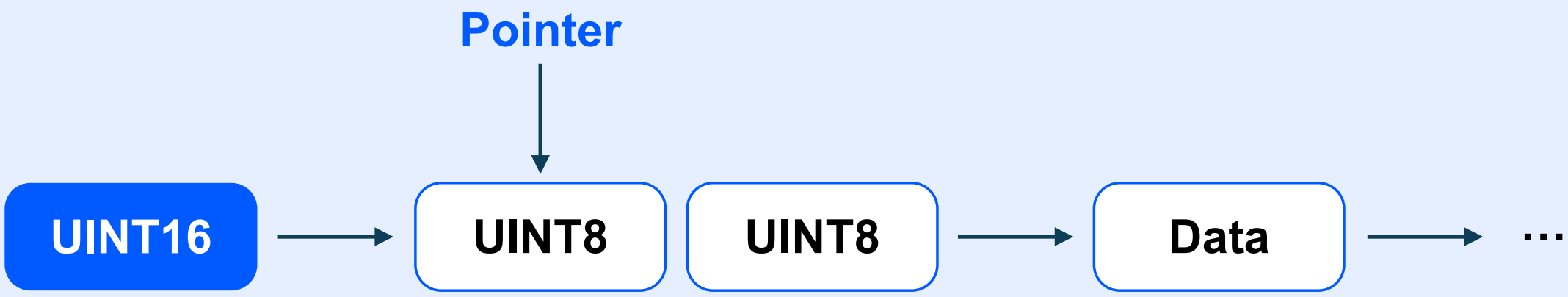
Задача преобразования данных

Data(bytes: **UnsafeRawPointer**, count: **Int**)

Data(bytes: **UnsafeRawPointer**, count: **Int**)



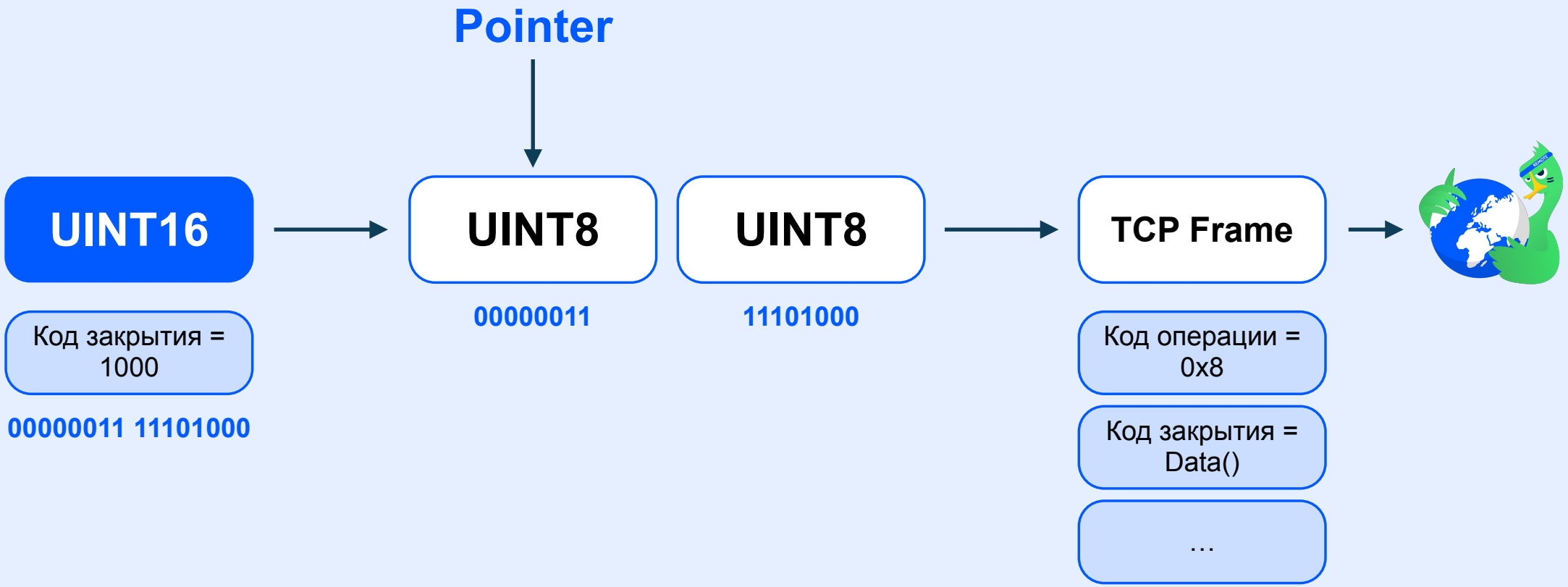
Data(bytes: **UnsafeRawPointer**, count: **Int**)



[Int8](repeating: Int8, count: Int)

Data(bytes: UnsafeRawPointer, count: Int)

Таааак….
И где это применить?


Pointer
UINT16
UINT8
UINT8
TCP Frame
Код закрытия = 1000
00000011 11101000
00000011
11101000
Код операции = 0x8
Код закрытия = Data()
…

Формирование исходных данных для транспортного фрейма

Полезные данные не могут быть скопированы

```
struct PointerBuffer: ~Copyable {
    var pointer = [UInt8](repeating: 0, count: MemoryLayout<UInt16>.size)

    var payload: Data {
        consuming get { Data(bytes: pointer, count: MemoryLayout<UInt16>.size) }
    }
}

func stop(closeCode: UInt16 = CloseCode.normal.rawValue) {
    var pointerBuffer = PointerBuffer()
    writeUint16(&pointerBuffer.pointer, offset: 0, value: closeCode)

    write(data: pointerBuffer.payload, opcode: .connectionClose, completion: {...})
}
```

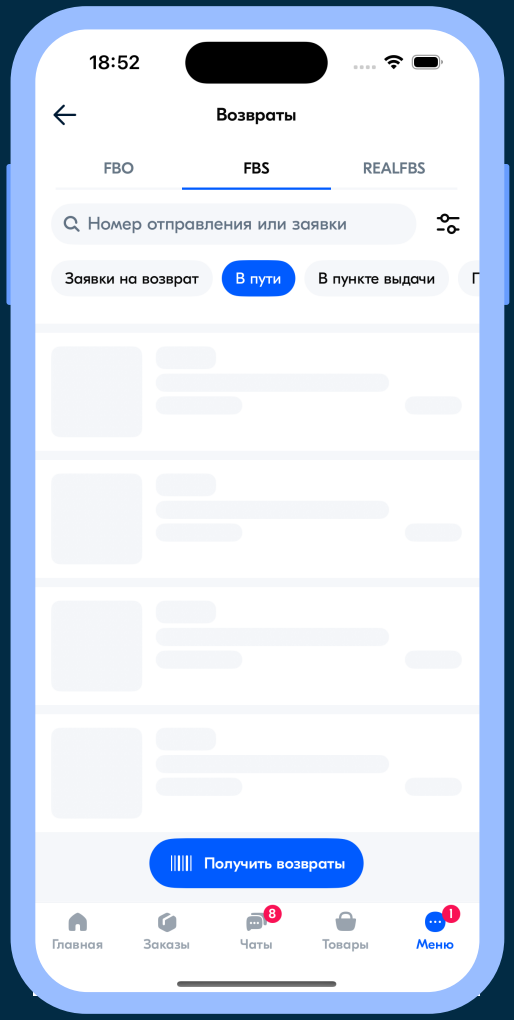
1. Pointer на [Int8]
2. **Convert: Int16 -> [Int8]**

PointerBuffer не жилец после потребления данных из него!

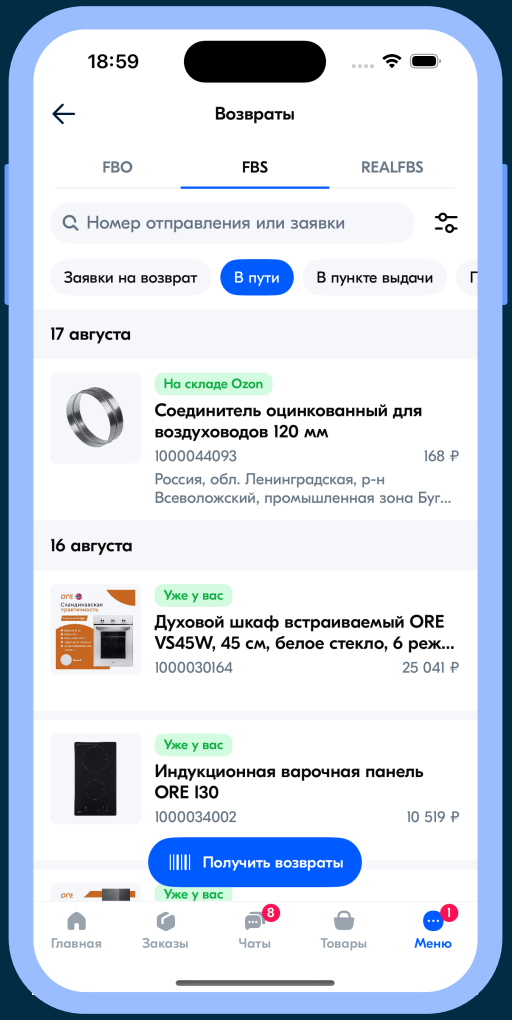
1. Convert: [Int8] -> Data

Последний продуктовый примерчик!

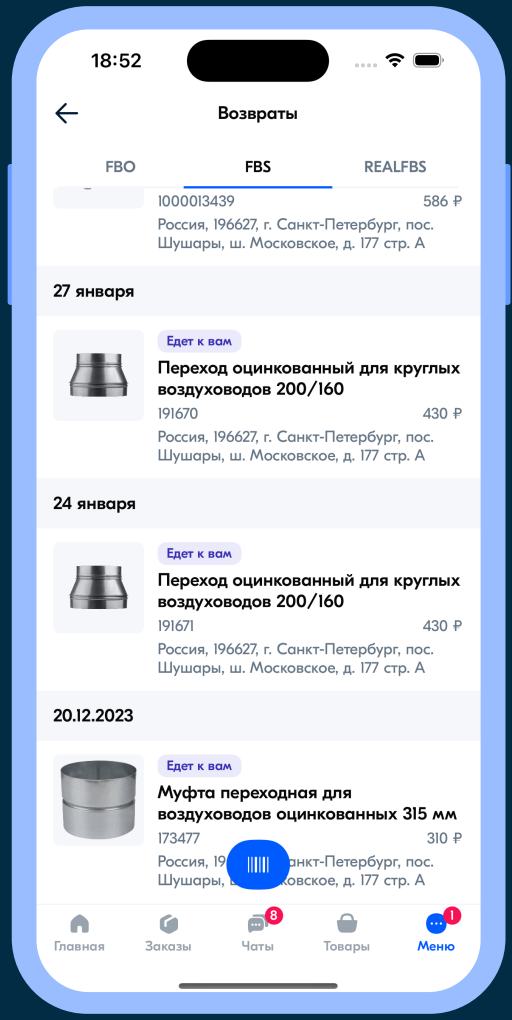
Пагинатор



Загрузка 1-ой страницы

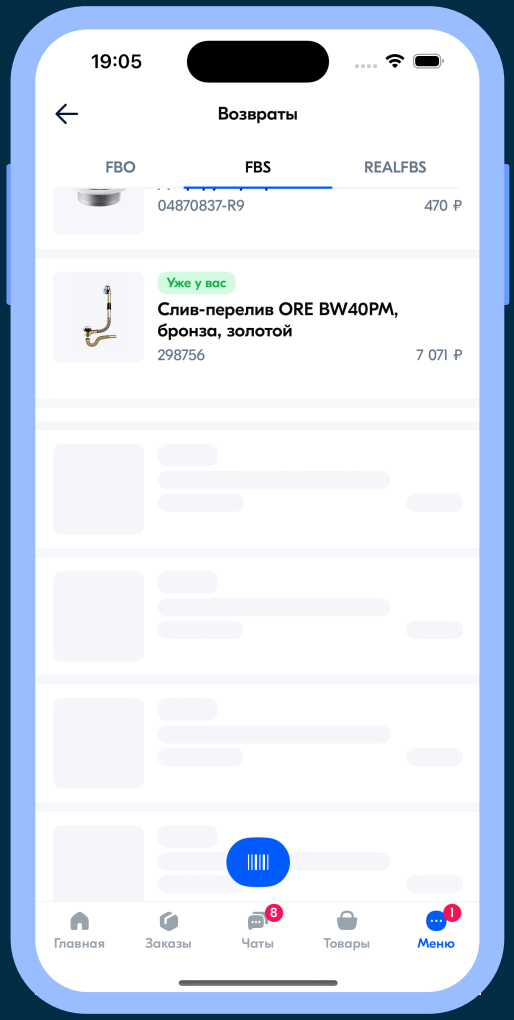


Загрузка завершена

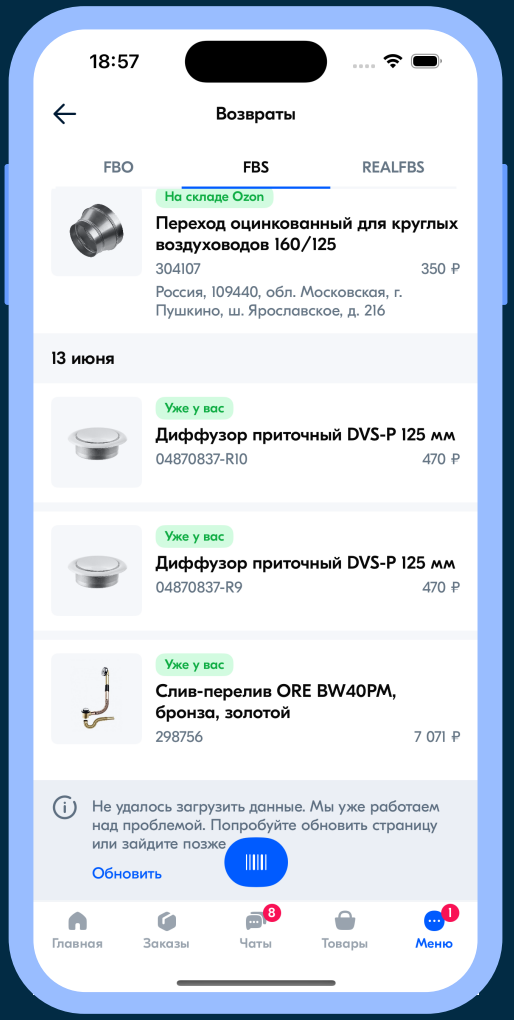


Конец контента и конец страницы

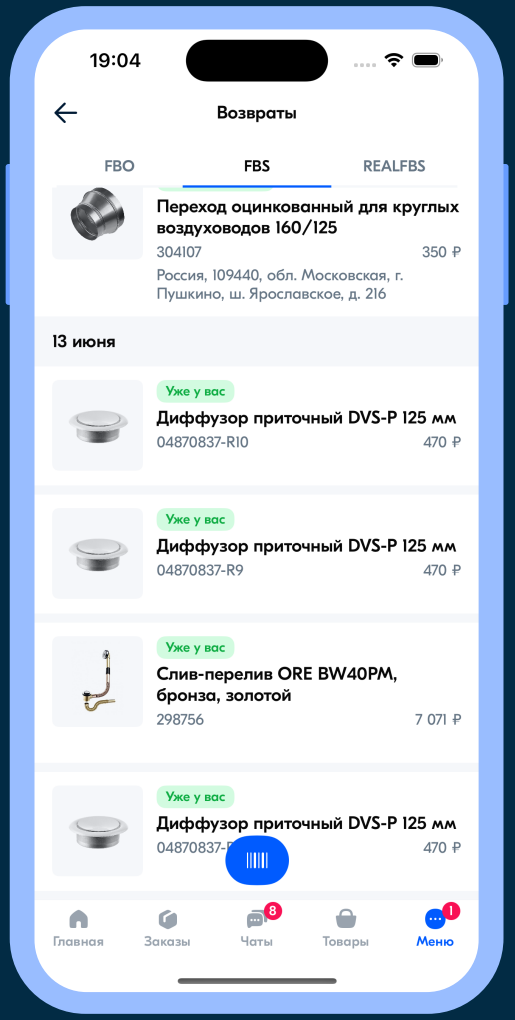
Пагинатор



Загрузка n-ой страницы

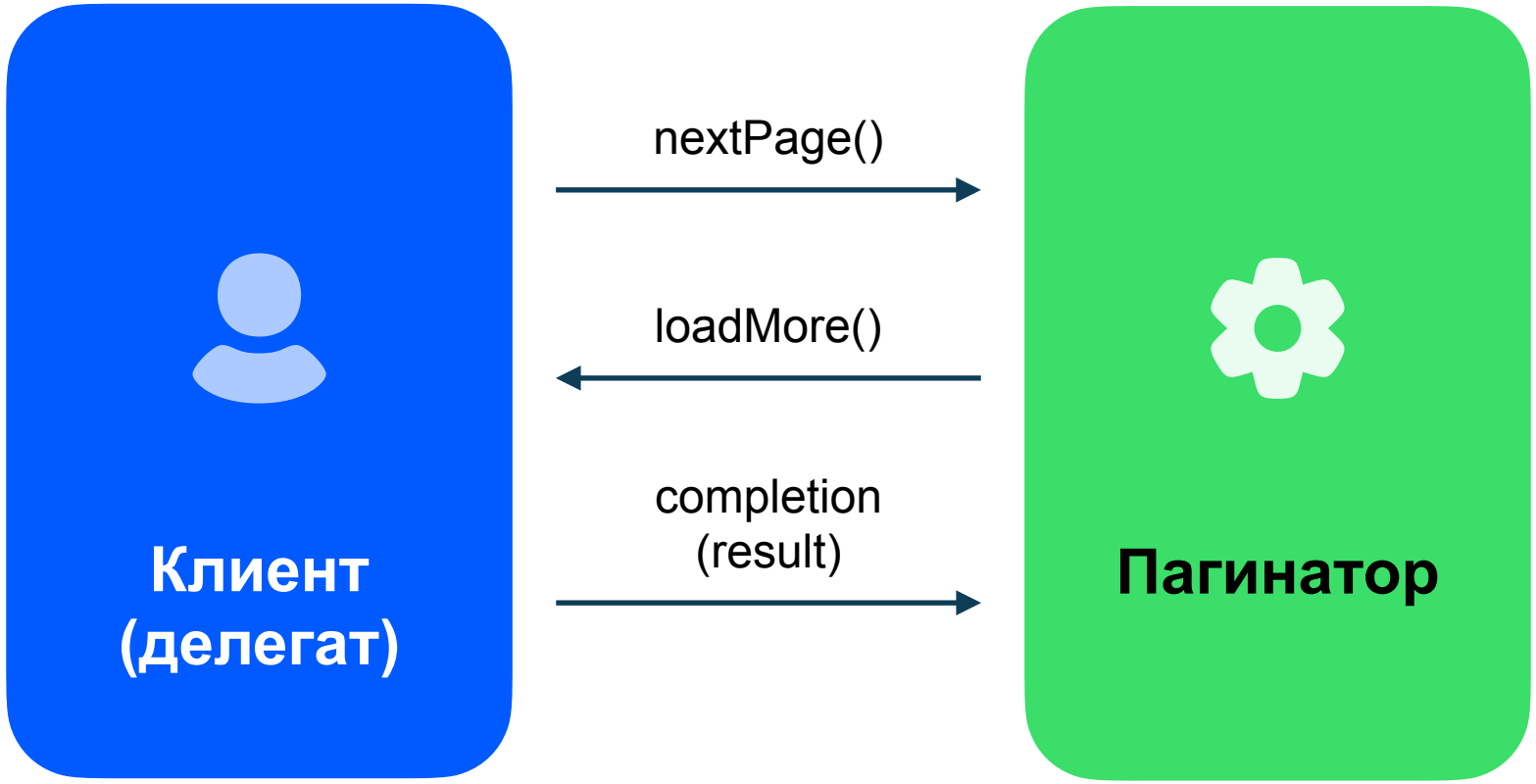


Ошибка загрузки



Успешная загрузка n-ой страницы

Как должно работать



Как настроить

```
final class ReturnsListViewModel: IReturnsListViewModel {
    let paginator: IPaginator

    init(paginator: IPaginator) {
        self.paginator = paginator
        self.paginator.delegate = self
    }
    // ...
}
```

1. Стать делегатом пагинатора

Как настроить

```
final class Paginator: IPaginator {
    private var _page = 0
    private var _lastId = ""
    private var _state: PaginatorState = .ready
    ...
    weak var delegate: IPaginablePresenter?
    ...
    func nextPage() {
        ...
        switch _state {
            case .ready:
                delegate?.loadMore(...) { ...
                    ...
                    switch result {
                    case let .success(newValue):
                        _page += 1
                        _lastId = newValue.lastId
                        _state = newValue.canLoadMore ? .ready : .end
                    case .failure:
                        return
                    }
                }
            case .end:
                return
            }
        ...
    }
    ...
}
```

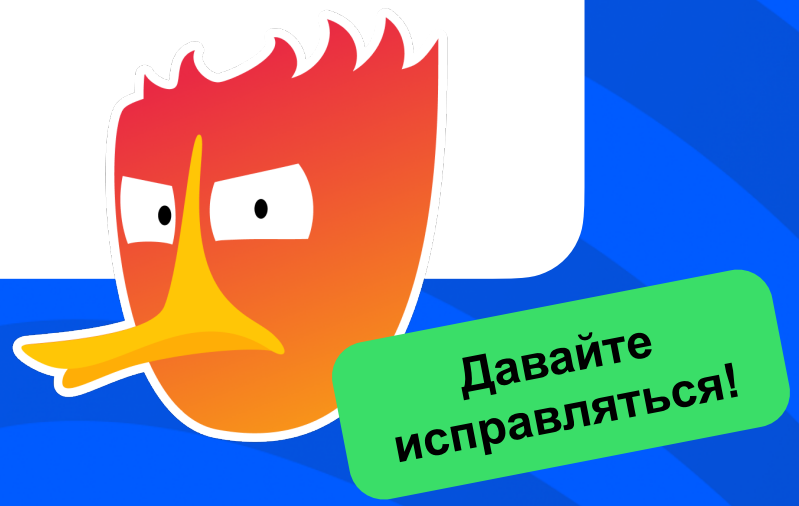
2. Пагинатор совершает вызов метода делегата

```
extension ReturnsListViewModel: IPaginablePresenter {
    func loadMore(
      page: Int, lastId: String, _ completion: consuming PaginationHelperCompletionHolder
    ) {
        do {
            let result = try service.getList()
            completion.receive(.success(.init(...)))
        } catch {
            completion.receive(.failure(error))
        }
    }
}
```

3. Делегат идёт в сеть и вызывает completion по завершении работы

Клиент не должен вызывать такой «completion» несколько раз!

Риск иметь неверное число загруженных страниц!



Как исправить

```
struct PaginationHelperCompletionHolder: ~Copyable {
    private var completion: ResultCallback<PaginatorModel>

    init(completion: @escaping ResultCallback<PaginatorModel>) {
        self.completion = completion
    }

    consuming func receive(_ result: Result<PaginatorModel, Error>) {
        switch result {
        case let .success(value):
            completion(.success(value))
        case .failure(let failure):
            completion(.failure(failure))
        }
    }
}
```

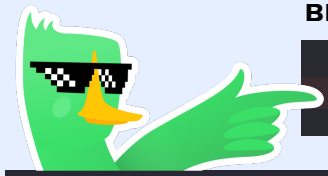
1. Добавить некопируемую обёртку к completion’у

Как исправить

```
final class Paginator: IPaginator {
  ...
  delegate?.loadMore(
    ...
    .init { ...
        ...
        switch result {
        case let .success(newValue):
            // Обновить _page, _lastId, _state
            ...
        case .failure:
            return
        }
    }
  )
  ...
}
```

2. Инициализируем «completion» внутри пагинатора

Попытка многократного вызова completion ==



```
'completion' consumed more than once
```

```
extension ReturnsListViewModel: IPaginablePresenter {
    func loadMore(
      ...,
      _ completion: consuming PaginationHelperCompletionHolder
    ) {
        do {
            let result = try service.getList(...)
            ...
            completion.receive(.success(.init(...)))
        } catch {
            completion.receive(.failure(error))
        }
    }
}
```

3. Вызовем «receive» через обёртку, полученную из «completion»

Бонус!

Некопируемые типы +
ModernConcurrency!

Async/Await сетевой клиент

```
extension ReturnsListViewModel: IPaginablePresenter {
    func loadMore(
      ...,
      _ completion: consuming PaginationHelperCompletionHolder
    ) async {
        do {
            let result = await try service.getList(...)
            ...
            completion.receive(.success(.init(...)))
        } catch {
            completion.receive(.failure(error))
        }
    }
}
```

1. Добавить «async» для метода делегата «loadMore»

```
func class AsyncPaginator: IPaginator {
  ...
  await delegate?.loadMore(
    page: _page,
    lastId: _lastId,
    .init { [weak self] result in
      ...
    }
  )
  ...
}
```

2. Вызов «async» метода

Async/Await сетевой клиент

```
extension ReturnsListViewModel: IPaginablePresenter {
    func loadMore(
      ...,
      _ completion: consuming PaginationHelperCompletionHolder
    ) async {
        do {
            let result = await try service.getList(...)
            ...
            completion.receive(.success(.init(...)))
        } catch {
            completion.receive(.failure(error))
        }
    }
}
```

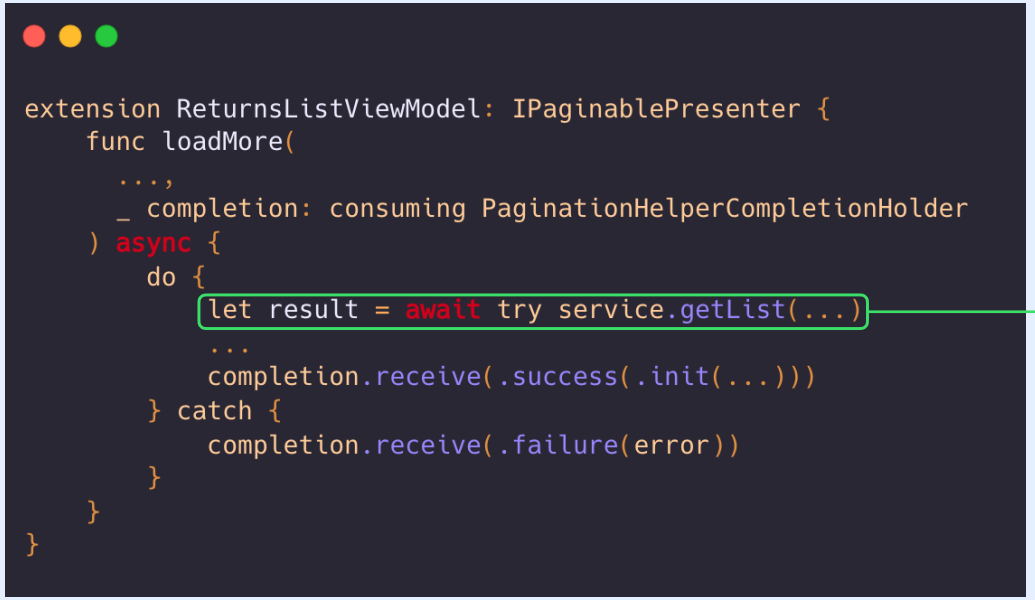
1. Добавить «async» для метода делегата «loadMore»

```
func class AsyncPaginator: IPaginator {
  ...
  await delegate?.loadMore(
    page: _page,
    lastId: _lastId,
    .init { [weak self] result in
      ...
    }
  )
  ...
}
```

2. Вызов «async» метода

Closure-based ModernConcurrency…
Не пишите так:)

Async/Await сетевой клиент



```
extension ReturnsListViewModel: IPaginablePresenter {
    func loadMore(
      ...,
      _ completion: consuming PaginationHelperCompletionHolder
    ) async {
        do {
            let result = await try service.getList(...)
            ...
            completion.receive(.success(.init(...)))
        } catch {
            completion.receive(.failure(error))
        }
    }
}
```

Неявное сохранение «completion» в куче перед передачей управления другому актору

1. Добавить «async» для метода делегата «loadMore»

Also: «Compile-time» защита от гонок в конкурентной среде

```
public struct PaginationHelperCompletionHolder: ~Copyable, Sendable {
    private let completion: @Sendable (Result<PaginatorModel, Error>) -> Void
    ...
}
```

Метрики!

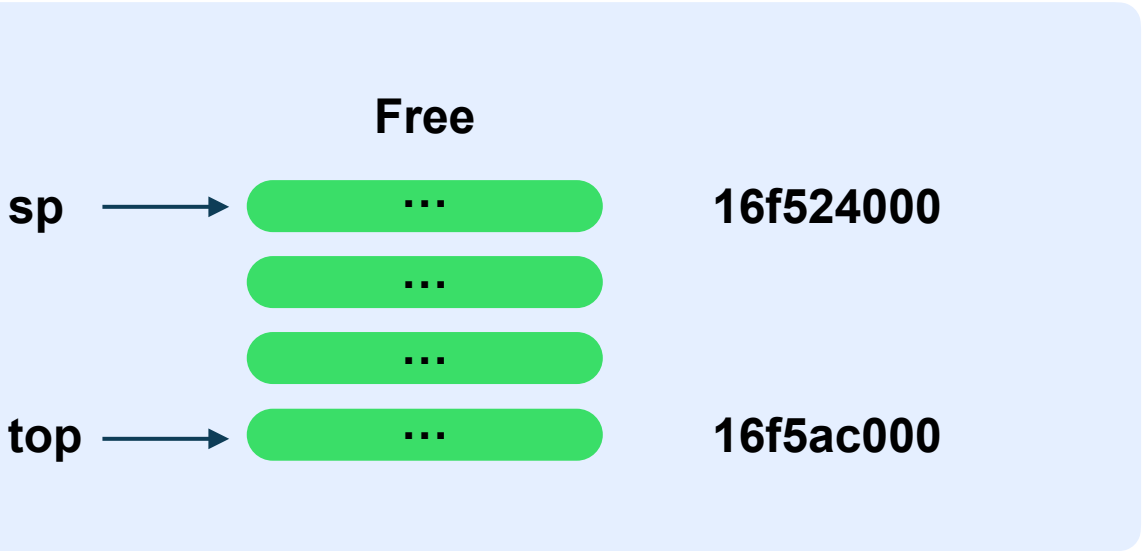
Пик вершины стека от досрочного вызова deinit

Простая некопируемая структура

```
struct NonCopyableStruct: ~Copyable {
    var payload: Int8

    consuming func consumingFunc() { }

    deinit { print("Deinited \(payload)") }
}
```

Метод измерения используемой памяти

```
func getStackUsage(sp: UnsafeMutableRawPointer) {
    let top = pthread_get_stackaddr_np(pthread_self())
    let size = pthread_get_stacksize_np(pthread_self())

    print("Thread stack usage: \(top - sp) bytes [\(100 * (top - sp)/size)%]"
}
```



Пик вершины стека от досрочного вызова deinit

```
func simulate() {
    var helperApproximateStartSP: Int8 = .zero
    getStackUsage(sp: &helperApproximateStartSP)

    let instance1 = NonCopyableStruct(payload: 1)
    instance1.consumingFunc()

    let instance2 = NonCopyableStruct(payload: 2)
    instance2.consumingFunc()

    let instance3 = NonCopyableStruct(payload: 3)
    instance3.consumingFunc()

    var helperApproximateEndSP: Int8 = .zero
    getStackUsage(sp: &helperApproximateEndSP)
}
```

```
Thread stack usage: 1025 bytes [0%]
Deinited 1
Deinited 2
Deinited 3
Thread stack usage: 1029 bytes [0%]
```

Нет выигрыша :(

А вот и причина…

```
struct CopyableStruct {
    var payload: Int8
}

func someFunc() {
  let value = CopyableStruct(payload: 1)
}
```

VS

```
struct NonCopyableStruct: ~Copyable {
  var payload: Int8
  deinit {...}
}

func someFunc() {
  let value = NonCopyableStruct(payload: 1)
}
```

```
0x100ae2e1c <+88>:  bl     0x100ae1c54  ; CopyableStruct.init(...) -> ...
0x100ae2e20 <+92>:  str    x0, [sp, #0x10]
0x100ae2e24 <+96>:  ldp    x29, x30, [sp, #0x30]
0x100ae2e28 <+100>: add    sp, sp, #0x40
0x100ae2e2c <+104>: ret
```

```
0x105606e14 <+88>:  bl     0x105606e54  ; NonCopyableStruct.init(...) -> ...
0x105606e18 <+92>:  strb   w0, [sp, #0x17]
0x105606e1c <+96>:  ldrb   w0, [sp, #0x17]
0x105606e20 <+100>: bl     0x1056078a8  ; NonCopyableStruct.deinit
0x105606e24 <+104>: ldp    x29, x30, [sp, #0x30]
0x105606e28 <+108>: add    sp, sp, #0x40
0x105606e2c <+112>: ret
```

Пик использования кучи при досрочном вызове deinit

```
struct NoCopyableStruct<T>: ~Copyable {
    var array: [T]

    init(value: T) {
        self.array = [T].init(
          repeating: value, count: 100_000_000
        )
    }

    consuming func consumingFunc() {}
}
```

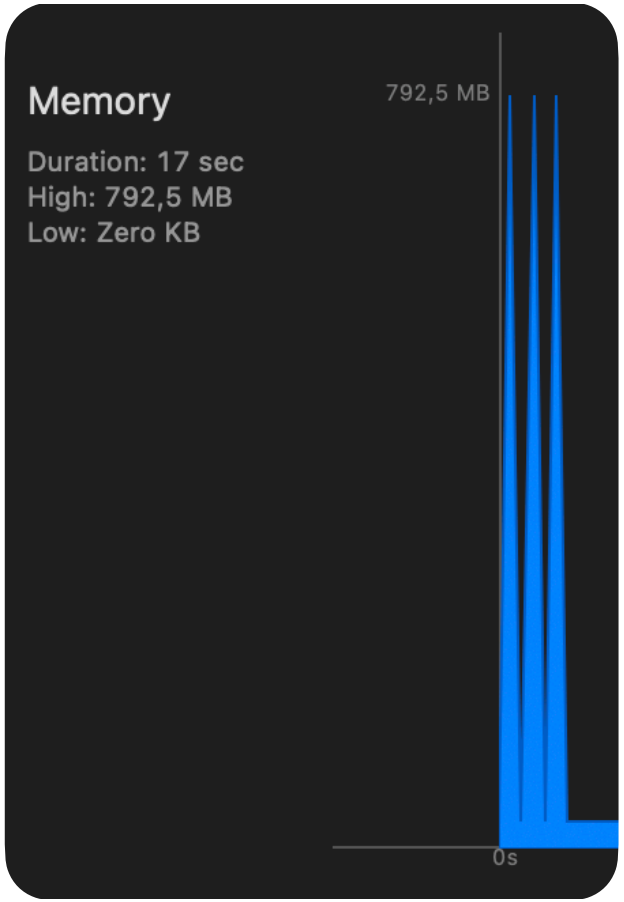
Обновлённая некопируемая структура

```
func simulate() {
    let instance1 = NoCopyableStruct(value: 1)
    instance1.consumingFunc()

    let instance2 = NoCopyableStruct(value: 1)
    instance2.consumingFunc()

    let instance3 = NoCopyableStruct(value: 1)
    instance3.consumingFunc()
}
```

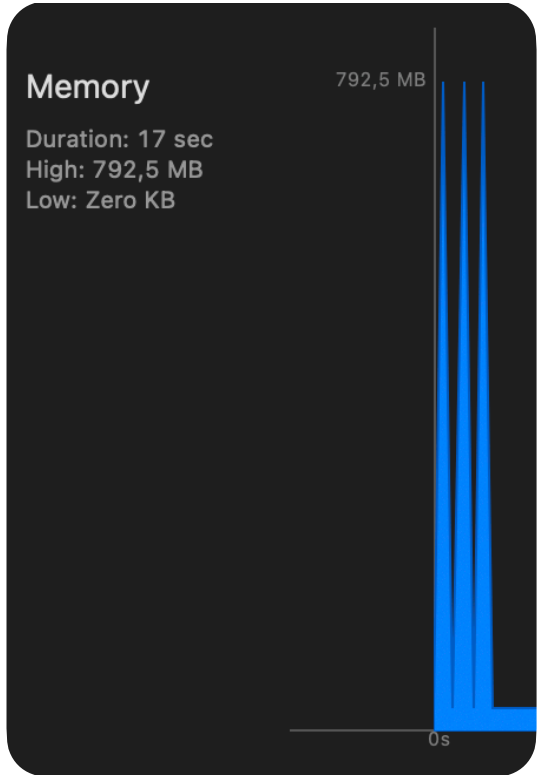
Тестируем

Пик использования кучи при досрочном вызове deinit

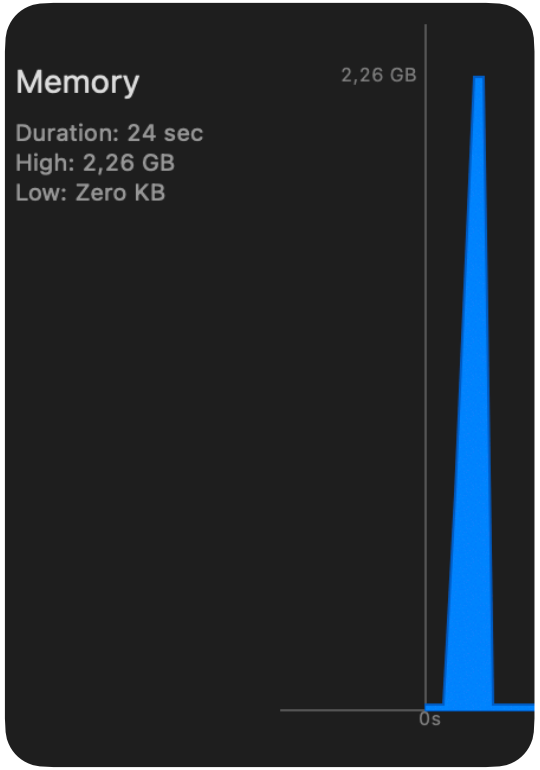
```swift
func simulate() {
    let instance1 = NoCopyableStruct(value: 1)
    instance1.consumingFunc()

    let instance2 = NoCopyableStruct(value: 1)
    instance2.consumingFunc()

    let instance3 = NoCopyableStruct(value: 1)
    instance3.consumingFunc()
}
```



Пик использования кучи при досрочном вызове deinit

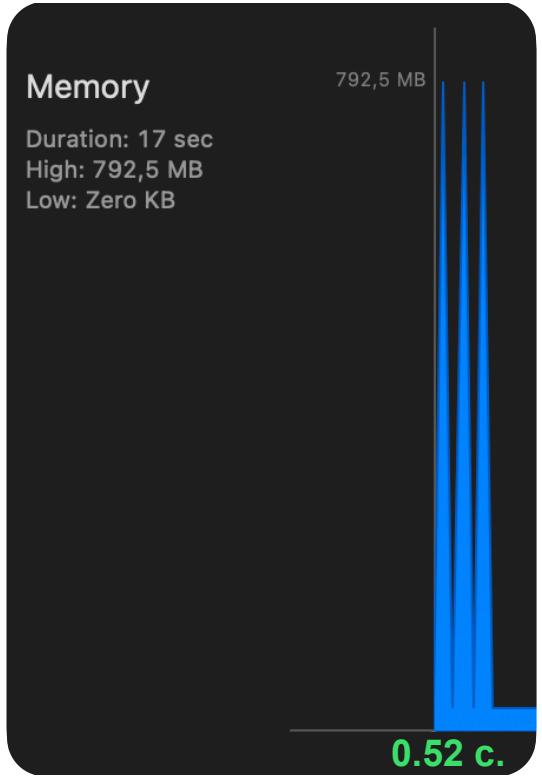


~Copyable + consuming



Copyable

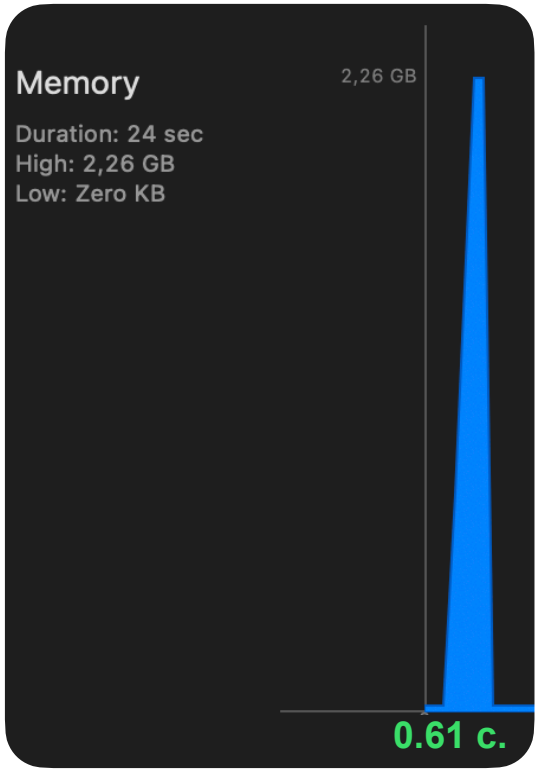
Пик использования кучи при досрочном вызове deinit



~Copyable + consuming

2 доп. обращения к памяти

Стоимость 1 «consume» ≈ 15 мс.



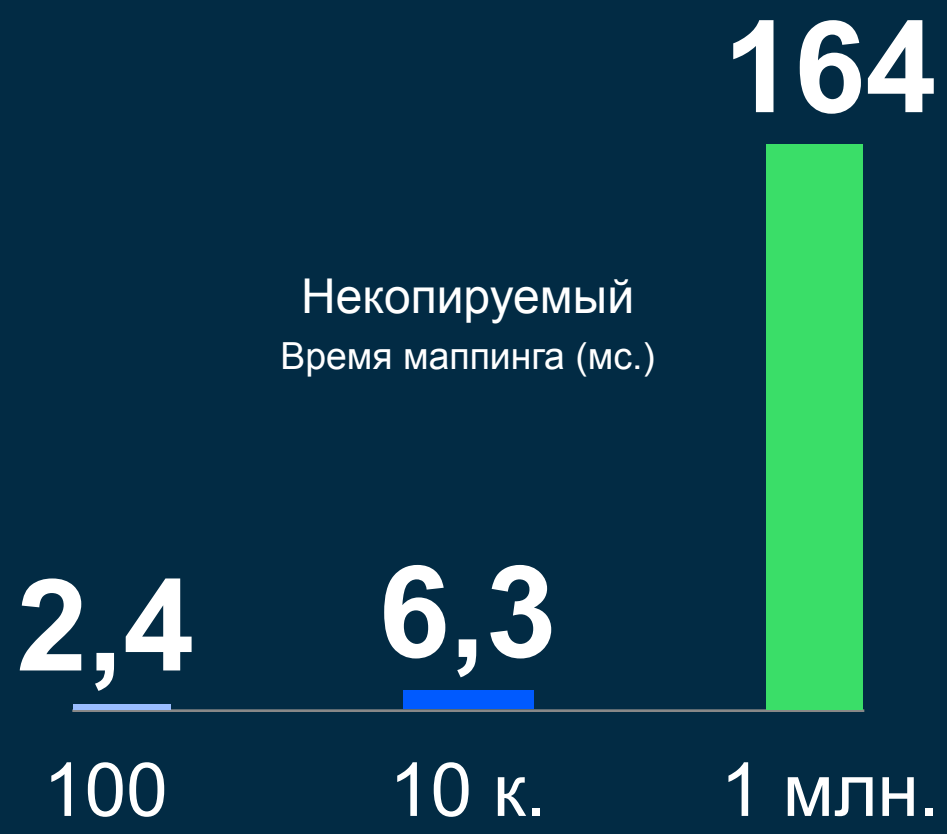
Copyable

Жирный deinit в конце

Стоимость маппинга

Размер server модели: **10 полей**

Размер mapped модели: **10 полей**



Стоимость маппинга

Размер server модели: **10 полей**
Размер mapped модели: **10 полей**

Дельта пиков памяти

Выводы/ рекомендации

Итоги

- Некопируемые типы — это повышение прозрачности логики кода
- Compile-Time защита клиента от необдуманных шагов
- С выходом Swift 6 стало более применимо
- Может помочь уменьшить пики памяти на Heap
- Не поможет уменьшению пика вершины Stack
- ~Copyable-wrapper может дать оверхэд времени и памяти
- Пока достаточно Runtime и Compile-ошибок

Struct<T>: ~Copyable» + consuming get
Escaping/non-escaping closures + «consume»

ПОКА БОЛИТ :(

Куда двигаемся

Решение: Coroutine's ***_read/_modify***!

```
var descriptor: SomeNonCopyable {
    return _descriptor // ERROR: attempt to copy `_descriptor`
}
```

- Associated Types в Generic-протоколах
- Поддержка Standard library (коллекции)
- Noncopyable + Sequence
- «Discard self» для ~Copyable типов с «Non-trivially-destroyed» полями
- Повышение прозрачности «underhood»
  Например, «borrowing switch»
- ~Copyable Tuples

OZON tech

Спасибо!

Алексей Таран

Ozon Seller iOS, руководитель группы

@alextar04

@alextar04