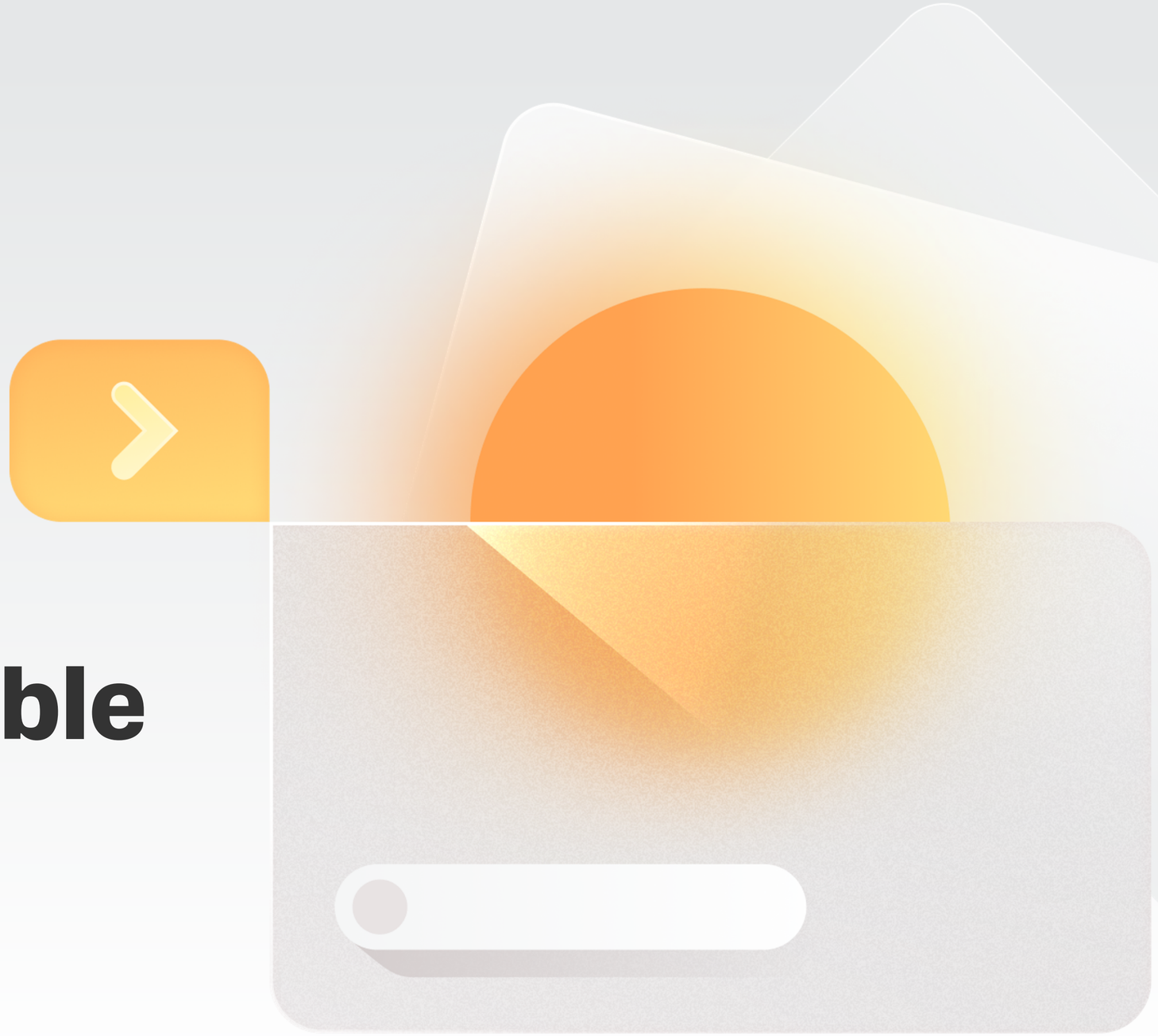


Ускоряем Codable

или как мой код попал в Apple

Обо мне

Опыт:

2.5 года в команде производительности в Т-Банке

Что люблю:

Копаться в кишках Swift-а

Улучшать производительность кода других команд

Чем горжусь:

Своей командой

Коммитами в Swift, Lottie, Tuist

План

01 JSONDecoder/Encoder медленный?
- Почему?

02 Оптимизации
- Найдем проблемные места
- Придумаем решение
- Обсудим результаты

03 Что не так с бенчмарком Apple?

04 Наш контрибьют в Swift Foundation

05 Фишки для успешного внедрения оптимизаций в больших проектах

Спойлер!

Покажу, как ускорить **`JSONDecoder`**
в 6-10 раз

Как родилась идея?

2024 Q4

Осознание

Появляется понимание масштаба проблемы

2025 Q1

Deep Dive

Исследуем работу `JSONDecoder/Encoder`, пробуем первые оптимизации

2025 Q2

Начало миграции

Первая оптимизация готова, готовим компонент для подмены реализаций

2025 Q3-4

Пик миграции

Вторая оптимизация, проверки, гайды, EoL, дашборды

Как родилась идея?

2024 Q4

Осознание

Появляется понимание масштаба проблемы

2025 Q1

Deep Dive

Исследуем работу `JSONDecoder/Encoder`, пробуем первые оптимизации

2025 Q2

Начало миграции

Первая оптимизация готова, готовим компонент для подмены реализаций

2025 Q3-4

Пик миграции

Вторая оптимизация, проверки, гайды, EoL, дашборды

Как родилась идея?

2024 Q4

Осознание

Появляется понимание масштаба проблемы

2025 Q1

Deep Dive

Исследуем работу `JSONDecoder/Encoder`, пробуем первые оптимизации

2025 Q2

Начало миграции

Первая оптимизация готова, готовим компонент для подмены реализаций

2025 Q3-4

Пик миграции

Вторая оптимизация, проверки, гайды, EoL, дашборды

Как родилась идея?

2024 Q4

Осознание

Появляется понимание масштаба проблемы

2025 Q1

Deep Dive

Исследуем работу `JSONDecoder/Encoder`, пробуем первые оптимизации

2025 Q2

Начало миграции

Первая оптимизация готова, готовим компонент для подмены реализаций

2025 Q3-4

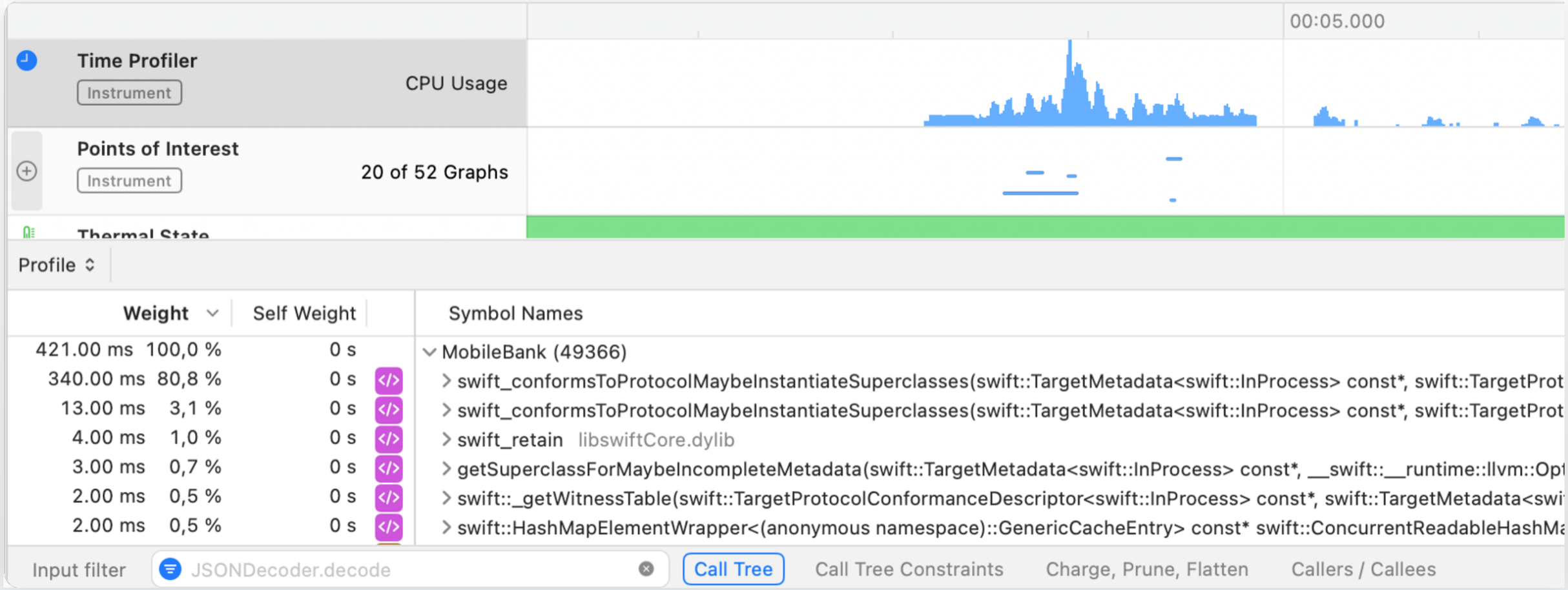
Пик миграции

Вторая оптимизация, проверки, гайды, EoL, дашборды

JSONDecoder/Encoder

Не такой быстрый!

JSONDecoder медленный?

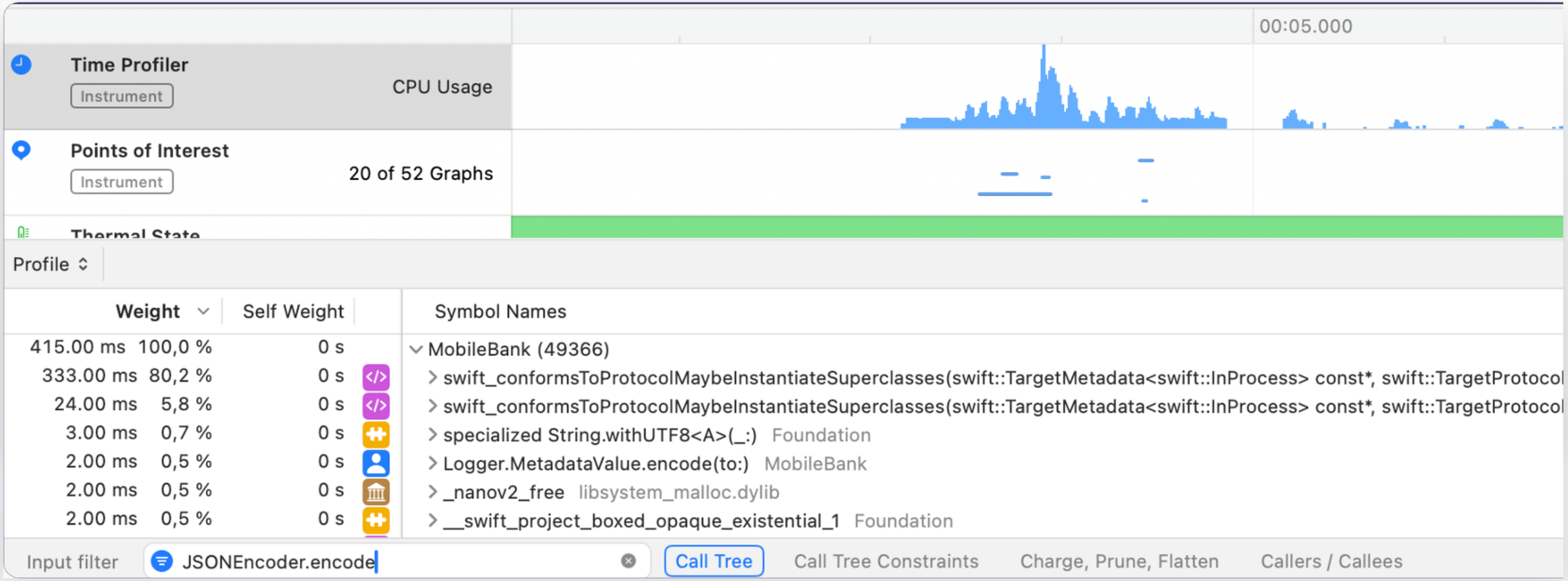


`JSONDecoder.decode` на всех потоках занимает целых 420 мс!
Хотя мы парсим не такие большие объемы данных на старте.
Почти все время уходит на `swift_conformsToProtocol`. Что это и почему?

А на проде? 90-ый перцентиль

Суммарно более 1.4 с (неполное покрытие)

JSONEncoder медленный?



`JSONEncoder.encode` на всех потоках занимает целых более 400 мс!
Как и в `JSONDecoder`-е все время уходит на `swift_conformsToProtocol`.
Что это и почему?

А на проде? 90-ый перцентиль

Суммарно почти 600 мс! (неполное покрытие)

swift_conformsToProtocol

01

Что делает?

Ищет нужный protocol conformance descriptor. Protocol conformance descriptor описывает, как тип реализует конкретный протокол.

02

Почему медленно?

В худшем случае, будет линейный поиск по всем существующим protocol conformance descriptor-ам. Первый вызов – худший случай

03

Сколько их?

В нашем приложении protocol conformance descriptor-ов уже более 200 тысяч

swift_conformsToProtocol

01

Что делает?

Ищет нужный protocol conformance descriptor. Protocol conformance descriptor описывает, как тип реализует конкретный протокол.

02

Почему медленно?

В худшем случае, будет линейный поиск по всем существующим protocol conformance descriptor-ам. Первый вызов – худший случай

03

Сколько их?

В нашем приложении protocol conformance descriptor-ов уже более 200 тысяч

swift_conformsToProtocol

01

Что делает?

Ищет нужный protocol conformance descriptor. Protocol conformance descriptor описывает, как тип реализует конкретный протокол.

02

Почему медленно?

В худшем случае, будет линейный поиск по всем существующим protocol conformance descriptor-ам. Первый вызов – худший случай

03

Сколько их?

В нашем приложении protocol conformance descriptor-ов уже более 200 тысяч

Если только первый вызов долгий, зачем его оптимизировать?

Почему нужна оптимизация?

Старт страдает сильнее

Именно на старте парсим сетевые ответы в первый раз. Там может происходить первый долгий вызов этого метода

01

Первое открытие экрана

Когда пользователь заходит на экран – случаются первые вызовы `swift_conformsToProtocol` для уникальных типов этого экрана

02

Старт в фоне

На запуск в фоне выделяется малое количество ресурсов. Если не успеть, watchdog убьет приложение, это замедлит следующий запуск

03

Шум в метриках

Сократим разницу в измерениях между первыми и последующими открытиями экранов

04

Почему нужна оптимизация?

Старт страдает сильнее

Именно на старте парсим сетевые ответы в первый раз. Там может происходить первый долгий вызов этого метода

01

Первое открытие экрана

Когда пользователь заходит на экран – случаются первые вызовы `swift_conformsToProtocol` для уникальных типов этого экрана

02

Старт в фоне

На запуск в фоне выделяется малое количество ресурсов. Если не успеть, watchdog убьет приложение, это замедлит следующий запуск

03

Шум в метриках

Сократим разницу в измерениях между первыми и последующими открытиями экранов

04

Почему нужна оптимизация?

Старт страдает сильнее

Именно на старте парсим сетевые ответы в первый раз. Там может происходить первый долгий вызов этого метода

01

Первое открытие экрана

Когда пользователь заходит на экран – случаются первые вызовы `swift_conformsToProtocol` для уникальных типов этого экрана

02

Старт в фоне

На запуск в фоне выделяется малое количество ресурсов. Если не успеть, watchdog убьет приложение, это замедлит следующий запуск

03

Шум в метриках

Сократим разницу в измерениях между первыми и последующими открытиями экранов

04

Почему нужна оптимизация?

Старт страдает сильнее

Именно на старте парсим сетевые ответы в первый раз. Там может происходить первый долгий вызов этого метода

01

Первое открытие экрана

Когда пользователь заходит на экран – случаются первые вызовы `swift_conformsToProtocol` для уникальных типов этого экрана

02

Старт в фоне

На запуск в фоне выделяется малое количество ресурсов. Если не успеть, watchdog убьет приложение, это замедлит следующий запуск

03

Шум в метриках

Сократим разницу в измерениях между первыми и последующими открытиями экранов

04

План

01 JSONDecoder/Encoder медленный?
Почему?

02 Оптимизации
Найдем проблемные места
Придумаем решение
Обсудим результаты

03 Что не так с бенчмарком Apple?

04 Наш контрибьют в Swift Foundation

05 Фишки для успешного внедрения оптимизаций в больших проектах

Где тормозит? JSONDecoder

Чтобы проверить, соответствует ли тип T протоколу – нужно вызвать `swift_conformsToProtocol` – это и есть узкое место в `JSONDecoder`-е

Document1

```
func unwrap<T: Decodable>(...) throws -> T {
  if type == Date.self {
    return try self.unwrapDate(...) as! T
  }
  ...
  if T.self is _JSONStringDictionaryDecodableMarker.Type {
    return try self.unwrapDictionary(...)
  }
  ...
}
```

Где тормозит? JSONDecoder

Чтобы проверить, соответствует ли тип T протоколу – нужно вызвать `swift_conformsToProtocol` – это и есть узкое место в `JSONDecoder`-е

Document1

```
func unwrap<T: Decodable>(...) throws -> T {
  if type == Date.self {
    return try self.unwrapDate(...) as! T
  }
  ...
  if T.self is _JSONStringDictionaryDecodableMarker.Type {
    return try self.unwrapDictionary(...)
  }
  ...
}
```

Где тормозит? JSONEncoder

Чтобы проверить, соответствует ли тип encodable протоколу `JSONStringDictionaryEncodableMarker` и `JSONDirectArrayEncodable` – нужно вызвать `swift_conformsToProtocol` – это и есть узкое место в `JSONEncoder`-е

Document1

```
func wrapGeneric<
  T: Encodable
>(...) throws -> JSONEncoderValue? {
  if let date = value as? Date {
    return try self.wrap(date, for: additionalKey)
  }
  ...
  else if let encodable = value as?
_JSONStringDictionaryEncodableMarker {
    return try self.wrap(
      encodable as! [String:Encodable],
      for: additionalKey)
  } else if let array = value as?
_JSONDirectArrayEncodable {
     ...
  }
  ...
}
```

Где тормозит? JSONEncoder

Чтобы проверить, соответствует ли тип encodable протоколу `JSONStringDictionaryEncodableMarker` и `JSONDirectArrayEncodable` – нужно вызвать `swift_conformsToProtocol` – это и есть узкое место в `JSONEncoder`-е

Document1

```
func wrapGeneric<
  T: Encodable
>(...) throws -> JSONEncoderValue? {
  if let date = value as? Date {
    return try self.wrap(date, for: additionalKey)
  }
  ...
  else if let encodable = value as?
_JSONStringDictionaryEncodableMarker {
    return try self.wrap(
      encodable as! [String:Encodable],
      for: additionalKey)
  } else if let array = value as?
_JSONDirectArrayEncodable {
     ...
  }
  ...
}
```

А еще где?

KeyedDecodingContainer

A concrete container that provides a view into a decoder’s storage, making the encoded properties of a decodable type accessible by keys.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
struct KeyedDecodingContainer<K> where K : CodingKey
```

Структура с type generic constraint-ом: при каждом упоминании типа попадаем на вызов `swift_conformsToProtocol`.

А еще где?

KeyedEncodingContainer

A concrete container that provides a view into an encoder’s storage, making the encoded properties of an encodable type accessible by keys.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
struct KeyedEncodingContainer<K> where K : CodingKey
```

Структура с type generic constraint-ом: при каждом упоминании типа попадаем на вызов `swift_conformsToProtocol`.

Как будем исправлять?

01 Перепишем `JSONDecoder/Encoder` с нуля

02 Удалим проблемный код и будем надеяться, что ничего не сломается

03 Разберемся, зачем нужны эти касты, придумаем, как и в каких случаях можем от них избавиться

Как будем исправлять?

01 Перепишем `JSONDecoder/Encoder` с нуля

02 Удалим проблемный код и будем надеяться, что ничего не сломается

03 Разберемся, зачем нужны эти касты, придумаем, как и в каких случаях можем от них избавиться

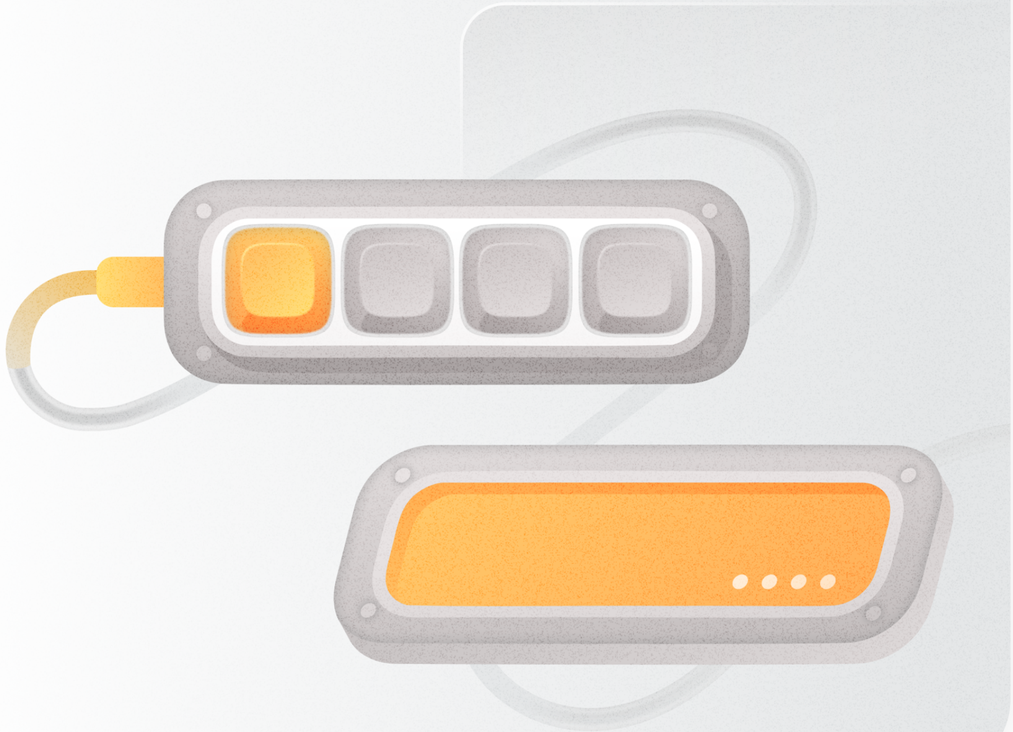
Как будем исправлять?

01 Перепишем `JSONDecoder/Encoder` с нуля

02 Удалим проблемный код и будем надеяться, что ничего не сломается

03 Разберемся, зачем нужны эти касты, придумаем, как и в каких случаях можем от них избавиться

Для чего нужны касты?



`_JSONDirectArrayEncodable`

Используется для оптимизации сериализации массива из примитивов (int/string/float/double)

`_JSONStringDictionary*Marker`

Для словарей с типом ключа String не применяет `keyedEncodingStrategy` и `keyedDecodingStrategy`

`_JSONDirectArrayEncodable`

01

Просто удалить этот код. Если у вас в приложении мало таких сценариев – ок. Не универсальное решение

02

Флаг в `JSONEncoder` - использовать эту оптимизацию или нет: сложнее пользоваться Уже более гибкое решение

03

Принципиально другой подход

Третий путь

JSONEncoder.swift

```
/// A protocol used to determine whether a value is an
`Array` containing values that allow/// us to bypass
UnkeyedEncodingContainer overhead by directly encoding the
contents as/// strings as passing that down to the
JSONWriter.
fileprivate protocol _JSONDirectArrayEncodable {
  @inline(__always)
  func nonPrettyJSONRepresentation(…) throws -> [UInt8]
  @inline(__always)
  func individualElementRepresentation(…) throws ->
  ([UInt8], lengths: [Int])
}

extension Array: _JSONDirectArrayEncodable where Element:
_JSONSimpleValueArrayElement {...}
```

`_JSONDirectArrayEncodable`

Протокол приватный

А, значит, ему могут конформить только типы из Foundation.

Третий путь

JSONEncoder.swift

```
fileprivate protocol _JSONSimpleValueArrayElement {
  @inline(__always)
  func serializeJsonRepresentation(…) throws -> Int
}
```

_JSONDirectArrayEncodable

Протокол приватный

А, значит, ему могут конформить только типы из Foundation.

*SimpleValueArrayElement

Тоже приватный!

Третий путь

Какие типы реализуют протокол _JSONSimpleValueArrayElement?

JSONEncoder.swift

```
extension _JSONSimpleValueArrayElement where Self:
FixedWidthInteger & CustomStringConvertible { }
...
extension Int : _JSONSimpleValueArrayElement { }
extension Int8 : _JSONSimpleValueArrayElement { }
...
extension UInt : _JSONSimpleValueArrayElement { }
extension UInt8 : _JSONSimpleValueArrayElement { }
...
extension String: _JSONSimpleValueArrayElement {...}
extension Float: _JSONSimpleValueArrayElement {...}
extension Double: _JSONSimpleValueArrayElement {...}
```

_JSONDirectArrayEncodable

Протокол приватный

А, значит, ему могут конформить только типы из Foundation.

*SimpleValueArrayElement

Тоже приватный!

Вывод

Только фиксированный набор типов реализуют этот протокол!

«Протокол реализуют всего 15 типов, значит, можно проверить, что текущий тип является одним из этих 15 БЕЗ каста к протоколу»

200 000 → 15

Итого: вместо тяжелого каста на 200к проверок получили всего 15 проверок

Третий путь: как перепишем код?

Было

```
// Было
if let array = value as?
_JSONDirectArrayEncodable {
  ...
}
```

Стало

```
// Стало
func _asDirectArrayEncodable<
  T: Encodable
>(_ value: T) -> _JSONDirectArrayEncodable? {
  return if let array = _specializingCast(value,
to: [Int].self) {
  array
}
...else if let array = _specializingCast(value,
to: [UInt].self) {
  array
} else if let array = _specializingCast(value,
to: [String].self) {
  array
} ...
```

Противоречим SOLID?

Цена SOLID-а очень высока

Да, код выглядит не канонично, зато работает в разы быстрее. Я был готов заплатить такую цену за скорость. Разработчики Swift Foundation со мной согласились

JSONStringDictionaryEn/DecodableMarker

Используется, чтобы не менять ключи в словарях по логике из `keyEncodingStrategy` и `keyDecodingStrategy`

Первый путь

Просто удалить этот код. Подходит, если у вас в приложении не используется `keyEncodingStrategy/keyDecodingStrategy`. Не универсальное решение

Второй путь

Не делать проверку, если установлена стандартная `keyEncodingStrategy/keyDecodingStrategy`

Второй путь: как перепишем код?

Document1

```
// Было

if let encodable = value as? _JSONStringDictionaryEncodableMarker {
  return try self.wrap(encodable as! [String:Encodable], for: additionalKey)
}

// Стало

if !options.keyEncodingStrategy.isDefault,
  let encodable = value as? _JSONStringDictionaryEncodableMarker {
  return try self.wrap(encodable as! [String:Encodable], for: additionalKey)
}
```

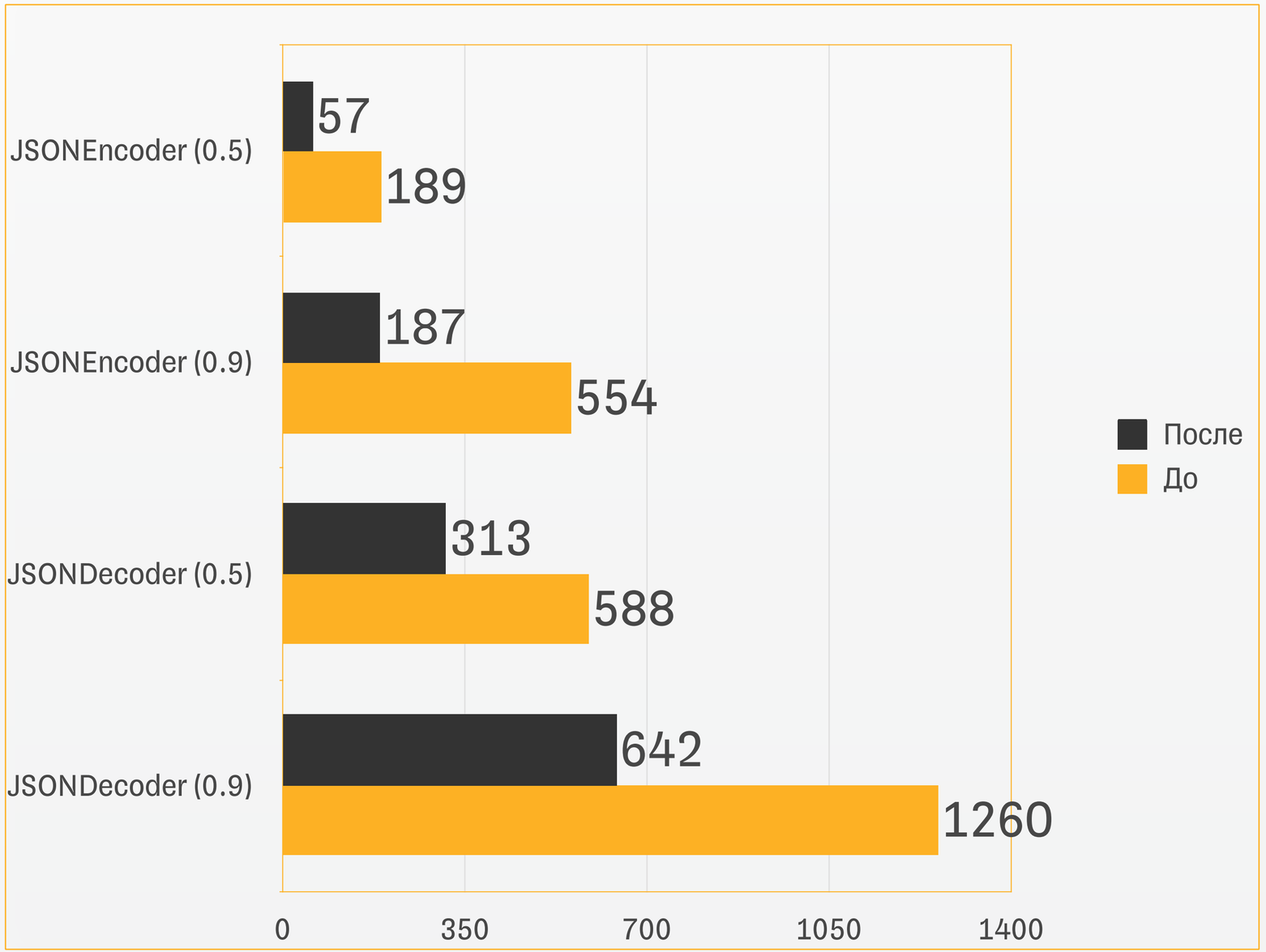
Второй путь: как перепишем код?

Document1

```
// Было JSONDecoder

if T.self is _JSONStringDictionaryDecodableMarker.Type {
  return try self.unwrapDictionary(...)
}

// Стало JSONDecoder

if !options.keyDecodingStrategy.isDefault,
  T.self is _JSONStringDictionaryDecodableMarker.Type {
  return try self.unwrapDictionary(...)
}
```

Результаты

- `JSONDecoder` стал быстрее в 2 раза
- `JSONEncoder` – в 3 раза



Итоги

Касты – не приговор

Выяснили, что если каст уже есть в коде - можно придумать, как от него избавиться

Каст не нужен, если

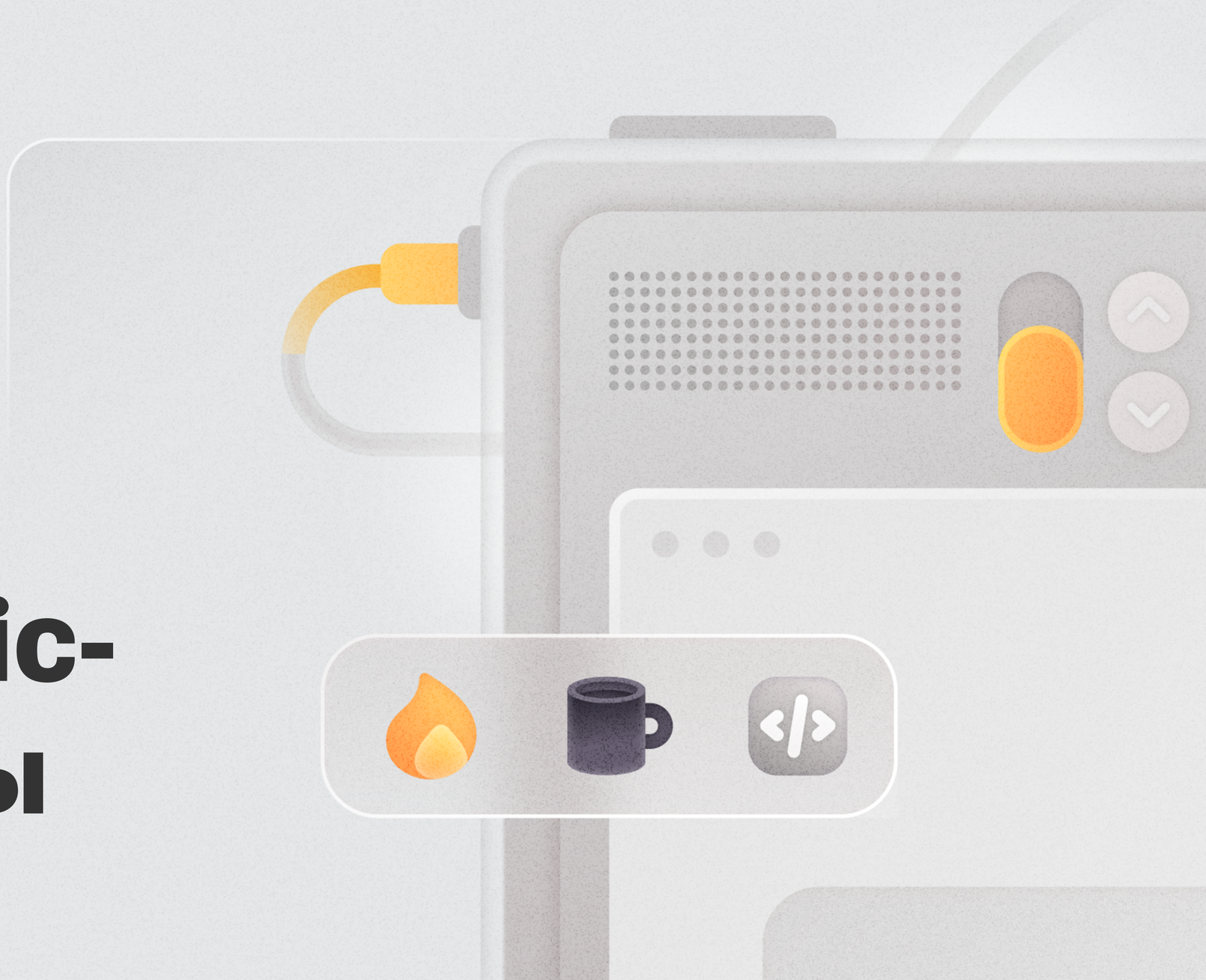
Протокол реализует фиксированное количество типов. Замените каст на сравнение типов.

Не кастите впустую

Если каст нужен только в определенной ветке исполнения, выполняйте каст только в этой ветке!

01

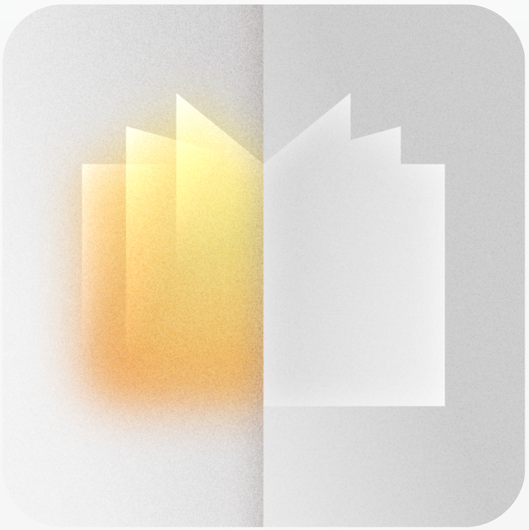
Type-generic-constraint-ы



Вводные

KeyedDecoding Container

Создается почти в каждом `init(from:)`. Аналогично `KeyedEncodingContainer`



У всех свои `CodingKeys`

При создании `KeyedEn/DecodingContainer` мы обречены на вызов `swift_conformsToProtocol`

Первый путь

VS

Второй путь

Переписываем

Переписать `KeyedEncodingContainer` и `KeyedDecodingContainer` без type-generic-constraint-ов

Подстраиваемся

Универсальный `CodingKey`!

Первый путь: идея

01 Попробуем стандартный трюк: убираем generic-constraint-ы в `extension`

Второй путь: как перепишем код?

Document1

```
// Было
struct KeyedEncodingContainer<K: CodingKey> :
  KeyedEncodingContainerProtocol{...}

// Стало
struct KeyedEncodingContainer<K> {...}

extension KeyedEncodingContainer:
  KeyedEncodingContainerProtocol
  where K: CodingKey {...}
```

Первый путь: идея

01 Попробуем стандартный трюк: убираем generic-constraint-ы в `extension`

02 PWT инжектится в compile-time, а не в Runtime при создании мета-данных типа

03 Придется править в Swift Foundation. И ломать публичное ABI. Сломать ABI в Swift Foundation не разрешат

Второй путь: идея

01

Метод `swift_conformsToProtocol` работает долго только первый раз для пары (тип, протокол)

02

Если использовать общий `CodingKey`, то оверхед будет только при первом создании `KeyedDe/EncodingContainer`

03

Можно сделать отдельную структуру, а можно использовать `String`

Второй путь: как перепишем код?

Document1

```
// Было
struct A: Decodable {
  let a: Int

  init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.a = try container.decode(Int.self, forKey: .a)
  }
}

// Стало
struct A: Decodable {
  let a: Int

  init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: AnyCodingKey.self)
    self.a = try container.decode(Int.self, forKey: AnyCodingKey("a"))
  }
}
```

Второй путь: неочевидные плюсы

Экономим размер

Каждый `CodingKey` занимает от 1 до 2 КБ размера приложения

Сокращаем descriptor-ы

Каждый `CodingKey` генерирует 5 protocol-conformance-descriptor-ов

`swift_conformsToProtocol`

Ускоряем этот метод за счет уменьшения protocol-conformance-descriptor-ов!

Результаты оптимизации (малая выборка)

Метрика	Было	Стало
`InsuranceModels.ActiveResponse`	10-18 мс	0-0.5 мс
`Loyalty.LoyaltyClientOffer`	12-17 мс	4-6 мс
`MobileBankIO.Account`	20-24 мс	8-9 мс
`TBundleIO.FullBundle`	30-40 мс	4-8 мс
`InvestmentGrowthShared.InvestmentPortfolio`	5-9 мс	0-0.4 мс
`InvestSavingAccountInterfaces.InvestSavingsAccount`	4-10 мс	1-3.2 мс

Результаты

01 A/B-тест провести сложно и трудозатратно. Ограничились локальными замерами

02 Совместив обе оптимизации мы получаем ускорение в 6-10 раз, в зависимости от особенностей типа

03 -6 мб размера: Уже избавились от 3000 `CodingKey`-ев!

План

01 JSONDecoder/Encoder медленный?
- Почему?

02 Оптимизации
- Найдем проблемные места
- Придумаем решение
- Обсудим результаты

03 Что не так с бенчмарком Apple?

04 Наш контрибьют в Swift Foundation

05 Фишки для успешного внедрения оптимизаций в больших проектах

Бенчмарки

Бенчмарк Codable в Swift Foundation

Что проверяют?

Прогоняют декодинг/енкодинг простых моделек миллиард раз

Их бенчмарк слеп

Из-за повторений без перезапуска, бенчмарк игнорирует оверхед от кастов

Маленький бинарь

protocol-conformance-descriptor-ов там мало, тоже искажает замеры

Выводы

01 Успех в бенчмарке не гарантирует успех в проде

02 Чтобы грамотно написать бенчмарк, нужно знать как работает и твой код, и runtime твоего языка

03 Измеряйте реальный сценарий, иначе рискуете получить ложные выводы

План

01 JSONDecoder/Encoder медленный?
- Почему?

02 Оптимизации
- Найдем проблемные места
- Придумаем решение
- Обсудим результаты

03 Что не так с бенчмарком Apple?

04 Наш контрибьют в Swift Foundation

05 Фишки для успешного внедрения оптимизаций в больших проектах

Контрибьют в Swift Foundation

Подготовка

Пруфы

Готовим бенчмарк, демонстрирующий улучшения. Выкладываем на github

01

Тред на forums.swift.org

Готовим лонг-рид для обсуждения. На площадке можно получить широкий охват: я получил около 1.5к просмотров

02

Issue на github

Без него не получится завести MP

03

Готовим PR

Создаем первую версию PR-а. Так разработчики стандартной библиотеки смогут сразу оценить идею

04

Как готовил бенчмарк?

01 Сделал бенчмарк, которые ровно 1 раз декодит/енкодит каждую из 10 тысяч моделей

02 Сравнил производительность `JSONDe/Encoder`: стандартный, с оптимизациями, с оптимизациями и с общим `CodingKey`

03 Симуляция большого приложения: 70к protocol conformance descriptor-ов

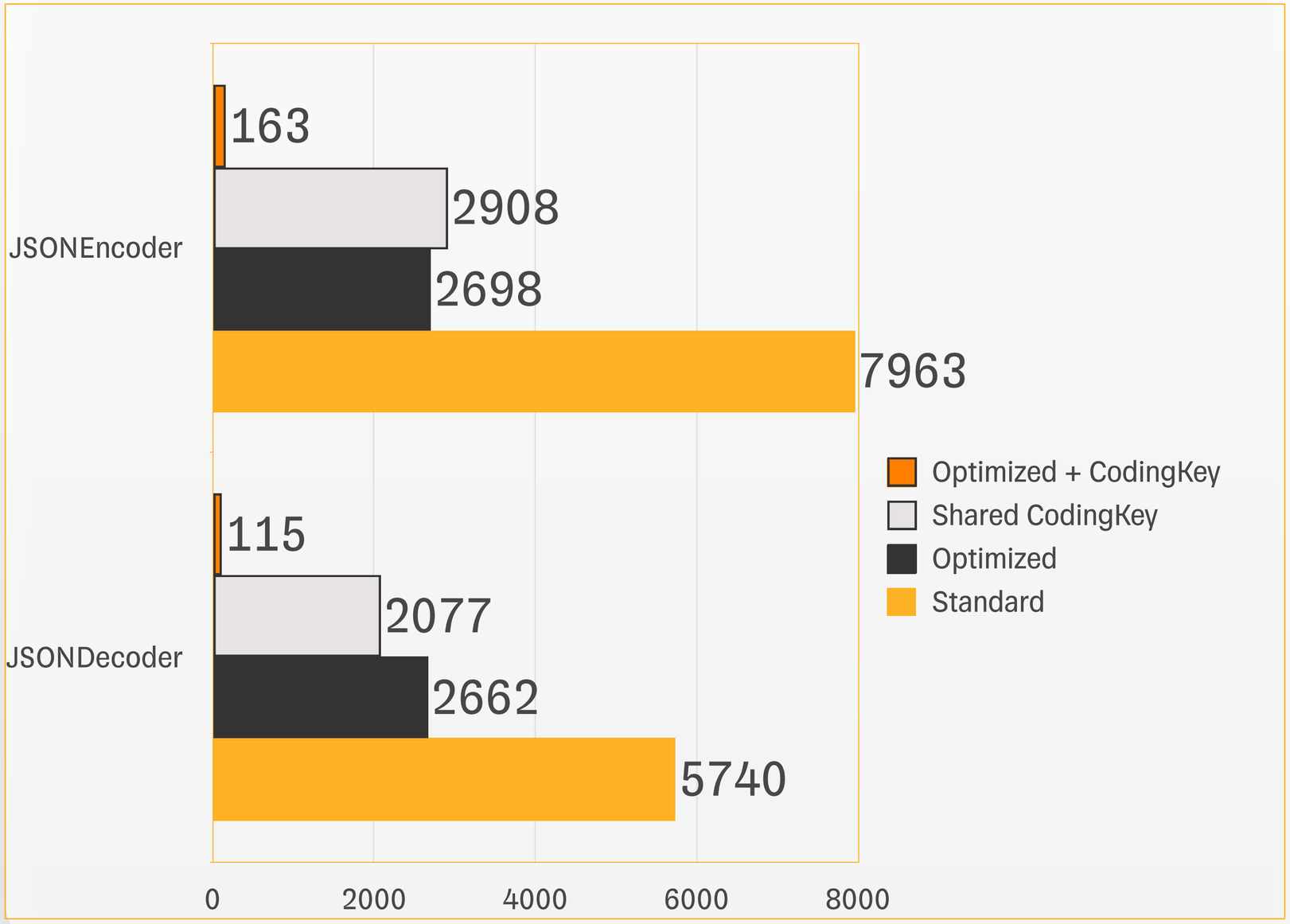
04 Подготовил скрипты для билда и для запуска

05 Подготовил маленький `JSON` (320 кб) для декодинга, небольшую модель для енкодинга

06 В отдельном файле описал свое железо и мои результаты

Результаты Бенчмарк

- `JSONDecoder` – в 50 раз быстрее с двумя оптимизациями
- `JSONEncoder` – аналогично



Подготовка

Пруфы

Готовим бенчмарк, демонстрирующий улучшения. Выкладываем на github

01

Тред на forums.swift.org

Готовим лонг-рид для обсуждения. На площадке можно получить широкий охват: я получил около 1.5к просмотров

02

Issue на github

Без него не получится завести MP

03

Готовим PR

Создаем первую версию PR-а. Так разработчики стандартной библиотеки смогут сразу оценить идею

04

Структура поста на форуме

01 **Введение**: почему каст к протоколам долгий, когда вызывается

02 **Проблемные места**: описал их, объяснил, какую проблему этот код решает

03 **Пруфы 1**: скриншоты из профайлера, доказывающие наличие проблем

04 **Предполагаемые оптимизации**: сначала неинвазивные, потом инвазивные

05 **Пруфы 2 и результаты**: привел замеры внутреннего AB-теста, замеры на собственном бенчмарке

06 **Прожарка бенчмарка Apple**: Бенчмарк замеряет сценарий, не учитывающий особенности языка

Подготовка

Пруфы

Готовим бенчмарк, демонстрирующий улучшения. Выкладываем на github

01

Тред на forums.swift.org

Готовим лонг-рид для обсуждения. На площадке можно получить широкий охват: я получил около 1.5к просмотров

02

Issue на github

Без него не получится завести MP

03

Готовим PR

Создаем первую версию PR-а. Так разработчики стандартной библиотеки смогут сразу оценить идею

04

Подготовка

Пруфы

Готовим бенчмарк, демонстрирующий улучшения. Выкладываем на github

01

Тред на forums.swift.org

Готовим лонг-рид для обсуждения. На площадке можно получить широкий охват: я получил около 1.5к просмотров

02

Issue на github

Без него не получится завести MP

03

Готовим PR

Создаем первую версию PR-а. Так разработчики стандартной библиотеки смогут сразу оценить идею

04

Неожиданности

01

В первый же день мне ответил разработчик. Обсудили неинвазивную оптимизацию.

02

Заодно исправил проблему в ZippyJSON и ReerJSON

03

В диалоге придумали оптимизацию про замену каста на сравнение типов

Неожиданности

01

В первый же день мне ответил разработчик. Обсудили неинвазивную оптимизацию.

02

Заодно исправил проблему в ZippyJSON и ReerJSON

03

В диалоге придумали оптимизацию про замену каста на сравнение типов

Неожиданности

01

В первый же день мне ответил разработчик. Обсудили неинвазивную оптимизацию.

02

Заодно исправил проблему в ZippyJSON и ReerJSON

03

В диалоге придумали оптимизацию про замену каста на сравнение типов

Итоги

01 Улучшили производительность всех приложений!

02 Улучшили другие реализации JSON-кодеров

03 От поста на forums.swift.org до влития PR прошло чуть более 30 дней

04 Разработчики стандартной библиотеки теперь знают о всех ошибках и, надеюсь, не повторят их

05 Узнал, что разработке новый подход к работе с сериализацией в Swift

06 Если бы не решился поправить код в Swift, то так бы и не додумался до одной из оптимизаций

План

01 JSONDecoder/Encoder медленный?
- Почему?

02 Оптимизации
- Найдем проблемные места
- Придумаем решение
- Обсудим результаты

03 Что не так с бенчмарком Apple?

04 Наш контрибьют в Swift Foundation

05 Фишки для успешного внедрения оптимизаций в больших проектах

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

Плохой подход

01

Использовать везде новую реализацию

Делаем в модуле X реализацию со всеми оптимизациями. Используем везде модуль X

02

Любое мажорное изменение - боль

Множество модулей будет зависеть от модуля X, сломать публичное API/ABI – обречь себя на длительную миграцию

03

Отсутствует гибкость

Захочешь добавить логгирование – делай изменения в модуле X. Никак не провести АБ-тест

Плохой подход

01

Использовать везде новую реализацию

Делаем в модуле X реализацию со всеми оптимизациями. Используем везде модуль X

02

Любое мажорное изменение - боль

Множество модулей будет зависеть от модуля X, сломать публичное API/ABI – обречь себя на длительную миграцию

03

Отсутствует гибкость

Захочешь добавить логгирование – делай изменения в модуле X. Никак не провести АБ-тест

Плохой подход

01

Использовать везде новую реализацию

Делаем в модуле X реализацию со всеми оптимизациями. Используем везде модуль X

02

Любое мажорное изменение - боль

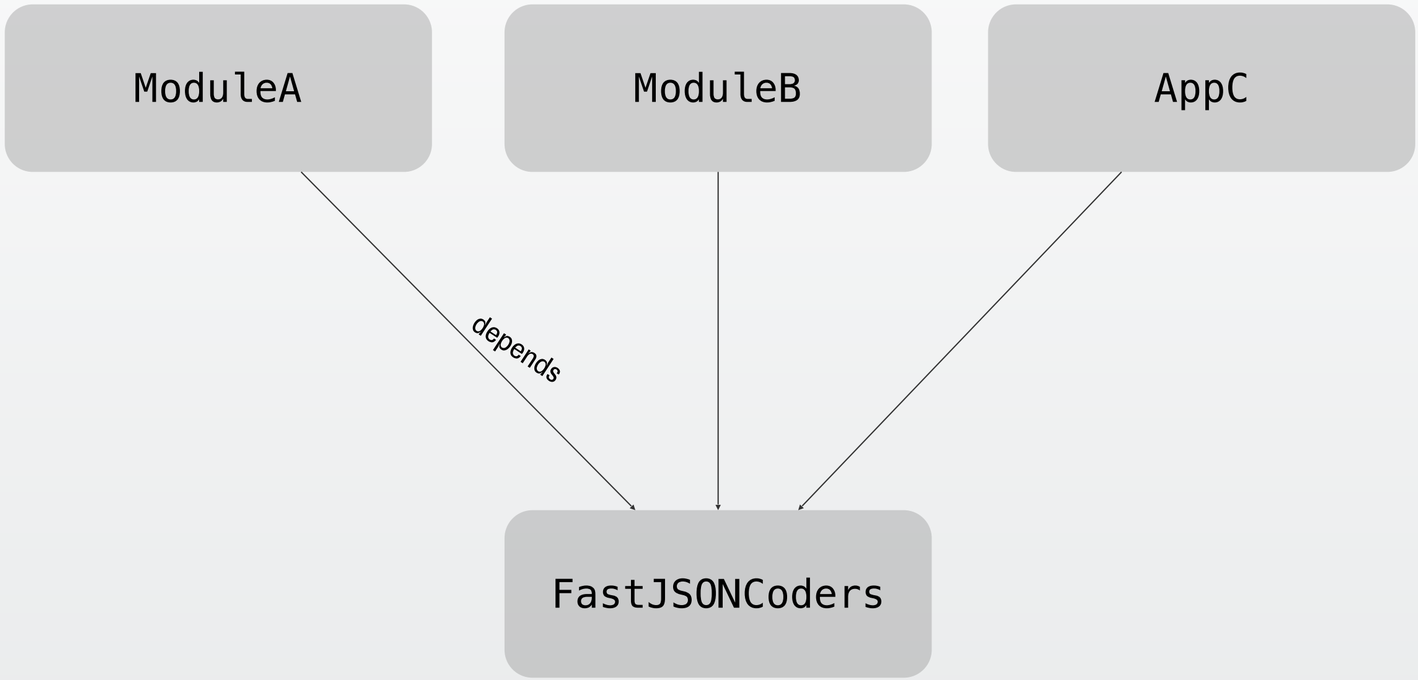
Множество модулей будет зависеть от модуля X, сломать публичное API/ABI – обречь себя на длительную миграцию

03

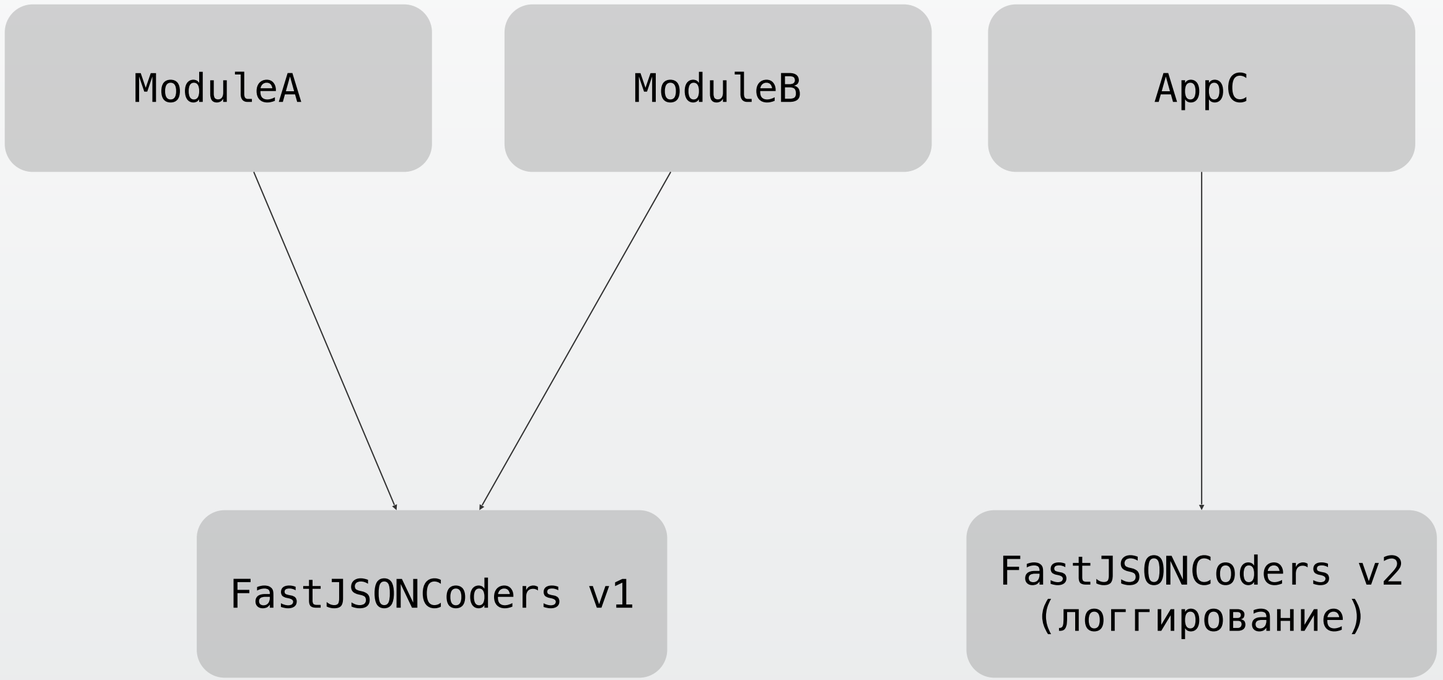
Отсутствует гибкость

Захочешь добавить логгирование – делай изменения в модуле X. Никак не провести АБ-тест

Плохой подход



Плохой подход



Правильный подход

01

Прокси-объект

Делаем в модуле X прокси-объект, в который можно подложить нужную реализацию. Используем везде модуль X

02

Минимум мажорных изменений

Так как в модуле X есть только прокси-объект и протоколы

03

Максимальная гибкость

Захочешь добавить логгирование – реализуй в хосте нужную логику, подложи в прокси-объект.

Прокси-объект

Document1

```
public final class TJSONDecoder:
 IJSONDecoder, @unchecked Sendable {
    private static var factory: () -> IJSONDecoder = { JSONDecoder() }
    private let wrapee: IJSONDecoder

    public init() {
        self.wrapee = Self.factory()
    }

    public static func bootstrap(_ factory: @escaping () -> IJSONDecoder) {
        self.factory = factory
    }
...
}
```

Правильный подход

01

Прокси-объект

Делаем в модуле X прокси-объект, в который можно подложить нужную реализацию. Используем везде модуль X

02

Минимум мажорных изменений

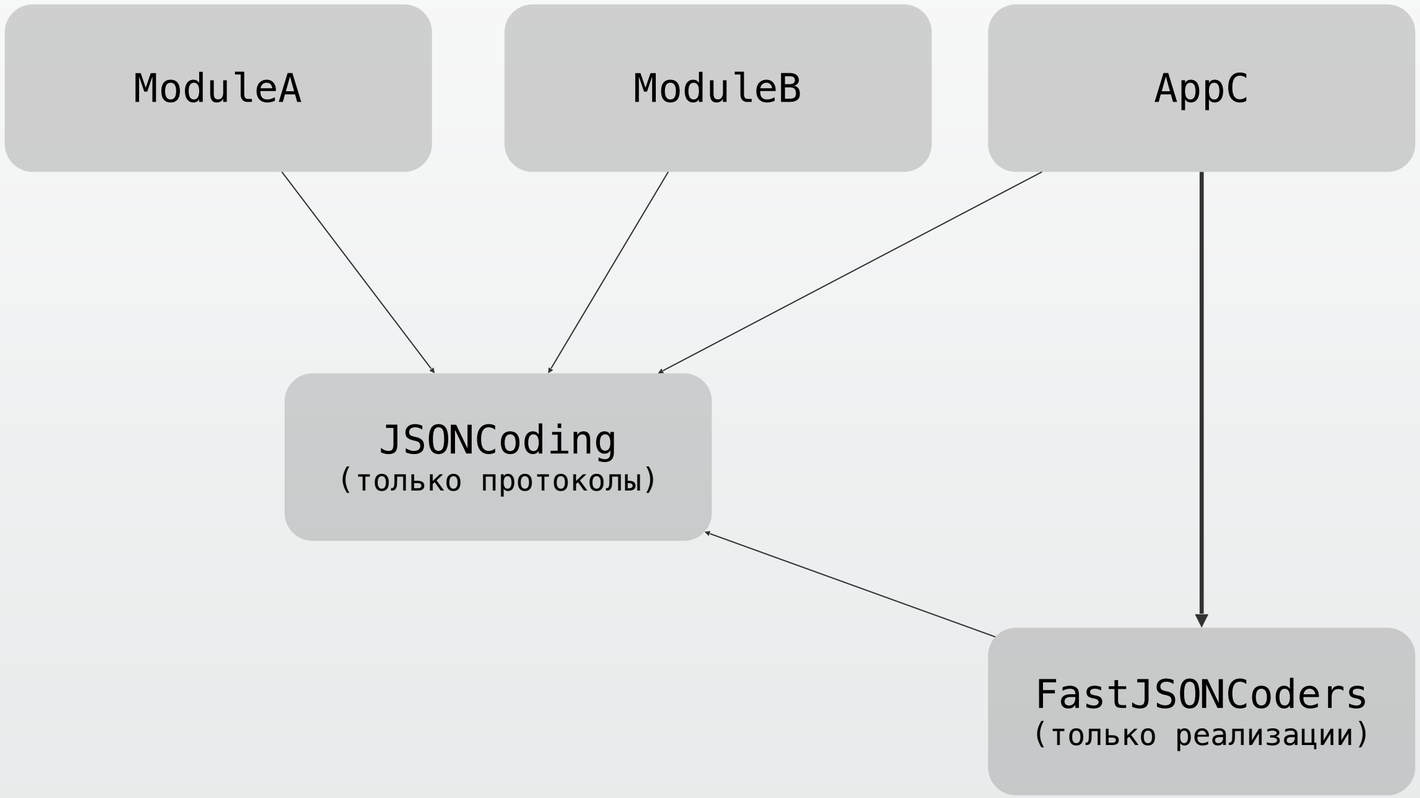
Так как в модуле X есть только прокси-объект и протоколы

03

Максимальная гибкость

Захочешь добавить логгирование – реализуй в хосте нужную логику, подложи в прокси-объект.

Правильный подход



Правильный подход

01

Прокси-объект

Делаем в модуле X прокси-объект, в который можно подложить нужную реализацию. Используем везде модуль X

02

Минимум мажорных изменений

Так как в модуле X есть только прокси-объект и протоколы

03

Максимальная гибкость

Захочешь добавить логгирование – реализуй в хосте нужную логику, подложи в прокси-объект.

Итоги

Прокси-объект

Реализация через прокси-объект позволяет гибко настраивать реализации на уровне хоста.

Минимум миграций

2nd-party модули зависят от модуля с абстракциями, а не от конкретных реализаций

АБ-тесты

За счет гибкой настройки на уровне хоста можно проводить АБ-тесты, добавлять логгирование и не только

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

«Манглированное имя – уникальное имя, которое генерирует компилятор для вашего класса, структуры, функции»

Зачем?

Чтобы обрабатывать перегрузки, вложенность типов, модули и многое другое

Модульность

Например, в модуле A и в модуле B может быть функция foo. У них будут разные манглированные имена

Перегрузки

На манглированное имя влияет сигнатура функции. За счет этого компилятор может выбрать нужную функцию

Вложенность

Тип A.B и тип B должны быть разными. Поэтому манглированные имена учитывают вложенность

Generics

Ограничения на generic-параметры, условия conditional-conformance – все попадает в манглированное имя

Примеры

Document1

```
swift demangle s10Foundation11JSONDecoderCACycfC
# $s10Foundation11JSONDecoderCACycfC ---> Foundation.JSONDecoder.__allocating_init() -> Foundation.JSONDecoder

swift demangle s10Foundation11JSONEncoderCACycfc
# $s10Foundation11JSONEncoderCACycfc ---> Foundation.JSONEncoder.init() -> Foundation.JSONEncoder
```

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

Поиск по исходникам

VS

Медленно, идеальная локализация проблемы

Нужно скачать исходники и искать в них строку через регулярки

Поиск по символам

Быстро, нет локализации

Не нужно качать исходники, достаточно вызвать nm

Наш контроль миграции на TJSONDecoder

Document1

```
nm path/to/your/binary \|
grep -E 's10Foundation11JSON(Decoder|
Encoder)CACycf[Cc]'  \|
wc -l
# 0 - все ок, >= 1 - модулю нужна миграция
```

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

Количество CodingKey-ев

Нам было необходимо оценить текущее количество `CodingKey`-ев в бинаре для оценки уменьшения размера приложения и количества protocol-conformance-descriptor-ов

Document1

```
nm path/to/your/binary | grep 'CodingKeys.*OMf' |
wc -l# Количество CodingKey в бинаре

# OMf — символ с метаданными типа
swift demangle
s10AcademyKit22RadioElementItemEntityV10CodingKeys
OMf
#$s10AcademyKit22RadioElementItemEntityV10CodingKe
ysOMf ---> full type metadata for
AcademyKit.RadioElementItemEntity.CodingKeys
```

Конкретный CodingKey

Для варнингов в MPe

Document1

```
nm path/to/your/binary | \
# ищем CodingKey
grep -E 'CodingKeys.*OMf' | \
# убираем префикс _$
awk '{gsub(/^_$/, "", $3); print $3}' | \
swift demangle | \
# получаем mangledName -> full type metadata for
ModuleName.ClassName.CodingKeys
awk -F'full type metadata for ' '{
  if ($2) {
    print $2
  }
}'
```

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

Статистика по модулям

Чтобы построить дашборд и выявить модули с самым высоким количеством `CodingKey`-ев. Упрощенный вариант

Document1

```
nm path/to/your/binary | \
# ищем CodingKey
grep -E 'CodingKeys.*OMf' | \
# убираем префикс _$
awk '{gsub(/^_$/, "", $3); print $3}' | \
# получаем mangledName -> full type metadata for
ModuleName.ClassName.CodingKeys
swift demangle | \
awk -F'full type metadata for ' '{
  if ($2) {
    # сплитим ModuleName.ClassName.CodingKeys
    n = split($2, parts, "\.")
    if (n > 1) {
      # Выводим модуль
      print parts[1]
}}}'
```

Статистика по протоколам

Сколько сущностей реализуют конкретный протокол

Document1

```
nm path/to/your/binary | \
# ищем protocol conformance descriptor-ы
grep -E 'MC$|Mc$' | \
awk '{print $3}' | \
swift demangle | \
grep "protocol conformance descriptor for"
# достаем текст вида Swift.Decodable in MyModule
awk -F': ' '{print $2}' | \
# достаем текст вида Swift.Decodable
awk -F'{print $1}' | \
# Считаем статистику
sort | uniq -c | sort -rn
```

Статистика по протоколам

Сколько сущностей реализуют конкретный протокол

Document1

```
30328 Swift.Equatable
21014 Swift.Hashable
10031 Swift.RawRepresentable
6715 Swift.Decodable
4843 Swift.Encodable
3633 Swift.CustomStringConvertible
3384 Swift.CustomDebugStringConvertible
3034 Swift.CodingKey
1896 MobileBankIO.JSONParsable
1751 Swift.CaseIterable
1743 ScreenVisitorInterfaces.ITrackableScreen
1407 Swift.Error
1109 ScreenVisitorInterfaces.IHasScreenName
1106 TinkoffDesignKit.TCSConfigurableView
```

Итоги

Mangling

Узнали, что такое манглированные имена и зачем они нужны

Исходники не нужны

Научились искать код в бинаре, без анализа исходного кода

Инструмент

Получили ценный инструмент для анализа конкретных модулей и всего приложения

О чем поговорим

01 Как обезопасить себя от еще одной подобной миграции?

02 Что такое манглированное имя?

03 Контроль миграции с одного компонента на другой на примере `JSONDecoder` → `TJSONDecoder`

04 Как посчитали количество `CodingKey`-ев? А по модулям? Какому классу принадлежит `CodingKey`?

05 Какую еще интересную статистику можно собрать с помощью манглированных имен?

06 Как мы внедряли универсальный `CodingKey`

Идеи

01

Минимум бойлерплейта

Точно не хотим писать руками реализацию `Codable`. Либо кодген, либо макрос

02

Как поддерживать кодген?

Код меняется, добавляются новые поля, сгенерированный код нужно обновлять. А если забыть – будет баг. Поэтому выбираем макрос

03

А что с миграцией

Нужно минимизировать ручной труд и количество ошибок

Идеи

01

Минимум бойлерплейта

Точно не хотим писать руками реализацию `Codable`. Либо кодген, либо макрос

02

Как поддерживать кодген?

Код меняется, добавляются новые поля, сгенерированный код нужно обновлять. А если забыть – будет баг. Поэтому выбираем макрос

03

А что с миграцией

Нужно минимизировать ручной труд и количество ошибок

Макрос

VS

Удобно, увеличивает время билда

Не нужно поддерживать актуальность реализации `Codable`

Кодген

Менее удобно, не влияет на время билда

Необходимо поддерживать актуальность реализации `Codable`

Макрос

Удобно, увеличивает время билда

Не нужно поддерживать актуальность реализации `Codable`

VS

Кодген

Менее удобно, не влияет на время билда

Необходимо поддерживать актуальность реализации `Codable`

Идеи

01

Минимум бойлерплейта

Точно не хотим писать руками реализацию `Codable`. Либо кодген, либо макрос

02

Как поддерживать кодген?

Код меняется, добавляются новые поля, сгенерированный код нужно обновлять. А если забыть – будет баг.

03

А что с миграцией

Нужно минимизировать ручной труд и количество ошибок

Автоматика по миграции. Скрипт

01

Скрипт по замене Codable → макрос

Осуществлять переезд ручным трудом – долго, грустно, и чревато ошибками

02

Требования к скрипту

Автоматизация 90% кейсов, учитывает внутреннюю специфику проекта.

03

Реализация на Swift Syntax

Намного легче поддерживать и развивать, чем самопальные скрипты

Автоматика по миграции. Тесты

01

Минимум багов из-за миграции

Поведение макроса и стандартного кодгена Apple отличается в редких случаях. Поэтому нужны тесты

02

Какие нужны тесты?

Необходимо протестировать сериализацию и десериализацию всех `Codable`-структур

03

Генерация через ИИ-агентов

Разработали двухагентный промпт, с помощью которого генерируем тесты

Итоги

Миграция без боли

Выбрали макрос, так как легче поддерживать, нежели кодген

Миграция без боли

За счет скрипта по автоматизировали переезд в 90% случаев

Обложились тестами

С помощью ИИ-агентов генерируем тесты до миграции, после миграции проверяем, что тесты зеленые

Финальные выводы

01 Погрузились в кишки `Codable`, нашли проблемные места

02 Рассмотрели разные подходы к оптимизации кастов к протоколам

03 Обсудили, что не так с JSON-бенчмарком Apple для `Codable`

04 Поняли, что контрибьютить в Swift не страшно. Главное – качественно осветить проблему

05 Улучшили производительность большинства iOS-приложений

06 Научились анализировать код по манглированным именам: используем для статистики и контроля

Полезные материалы

Статья про ускорение `Codable`

Статья про Swift Runtime

Тред на forums.swift.org