ТИНЬКОФФ

SwiftMacro + SwiftSyntax

With great power comes great responsibility

Zalutskiy Alexandr

14.12.2023

«Ма́крос - программный алгоритм действий, записанный пользователем. Часто макросы применяют для автоматизации рутинных действий. Также макрос — это символьное имя в шаблонах, заменяемое при обработке препроцессором на последовательность символов, например: фрагмент html-страницы в веб-шаблонах, или одно слово из словаря синонимов в синонимизаторах.»

Wikipedia

Что такое macro с точки зрения Swift?

Кодогенерация

SwiftMacro - инструмент для создания плагинов к компилятору, которые выполняются в процессе этапа компиляции и изменяют каким-либо образом ваш код. Макросы могут **только** генерировать новый код.

Freestanding macros

Freestanding макросы предоставляют нам инструмент, который позволяет нам заменять применение макроса на новые expression, declaration или блоки кода.

```swift
func myFunction() {
    print("Currently running \(#function)")
    #warning("Something's wrong")
}
```

```swift
func myFunction() {
    print("Currently running \("myFunction()")")
}
```

Как и все макросы в Swift, freestanding макросы позволяют генерировать warning и error. Они знают информацию о контексте, в котором применяются и аргументы которые мы передаем в макрос.

DeclarationMacro

```swift
public protocol DeclarationMacro: FreestandingMacro {
  /// Expand a macro described by the given freestanding macro expansion
  /// declaration within the given context to produce a set of declarations.
  static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax]

  /// Whether to copy attributes on the expansion syntax to expanded declarations,
  /// 'true' by default.
  static var propagateFreestandingMacroAttributes: Bool { get }
  /// Whether to copy modifiers on the expansion syntax to expanded declarations,
  /// 'true' by default.
  static var propagateFreestandingMacroModifiers: Bool { get }
}

public extension DeclarationMacro {
  static var propagateFreestandingMacroAttributes: Bool { true }
  static var propagateFreestandingMacroModifiers: Bool { true }
}
```

Attached macros

Attached макросы предоставляют нам инструмент, который позволяет нам создавать новые declaration на основе уже имеющихся.

```
@OptionSet<Int>
struct SundaeToppings {
    private enum Options: Int {
        case nuts
        case lime
        case fudge
    }
}
```

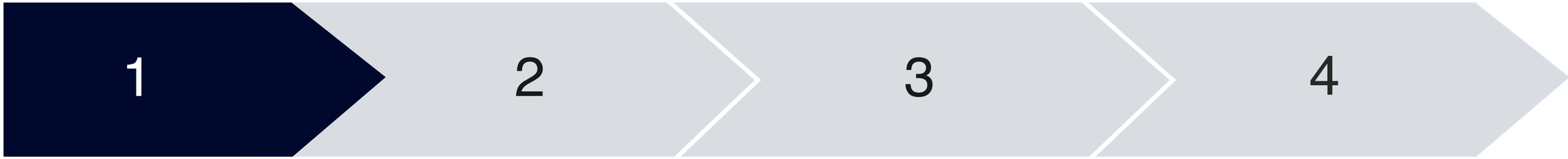
```
typealias RawValue = Int
var rawValue: RawValue
init() { self.rawValue = 0 }
init(rawValue: RawValue) { self.rawValue = rawValue }
static let nuts: Self = Self(rawValue: 1 << Options.nuts.rawValue)
static let lime: Self = Self(rawValue: 1 << Options.lime.rawValue)
static let fudge: Self = Self(rawValue: 1 << Options.fudge.rawValue)
```

Attached макросы не позволяют нам изменять declaration (кроме публичности и атрибутов). Основное применение Attached макросов - это расширение declaration или генерация новых на основе имеющихся.

MemberAttributeMacro

```swift
/// Describes a macro that can add attributes to the members inside the
/// declaration it's attached to.
public protocol MemberAttributeMacro: AttachedMacro {
  /// Expand an attached declaration macro to produce an attribute list for
  /// a given member.
  ///
  /// - Parameters:
  ///   - node: The custom attribute describing the attached macro.
  ///   - declaration: The declaration the macro attribute is attached to.
  ///   - member: The member declaration to attach the resulting attributes to.
  ///   - context: The context in which to perform the macro expansion.
  ///
  /// - Returns: the set of attributes to apply to the given member.
  static func expansion(
    of node: AttributeSyntax,
    attachedTo declaration: some DeclGroupSyntax,
    providingAttributesFor member: some DeclSyntaxProtocol,
    in context: some MacroExpansionContext
  ) throws -> [AttributeSyntax]
}
```

Порядок работы макроса



Парсер

Разбирает текст и возвращает нам AST-дерево

Валидация

Мы проверяем то, а можно ли применить макрос с таким деревом

Генерация

Мы обрабатываем полученное дерево и генерируем новый лес-деревьев

Компиляция

Наш лес подставляется в файл и полученный текст компилируется

```
private func test(int: Int) -> Int
```

```
FunctionDeclSyntax
├─attributes: AttributeListSyntax
├─modifiers: DeclModifierListSyntax
│ └─[0]: DeclModifierSyntax
│   └─name: keyword(SwiftSyntax.Keyword.private)
├─funcKeyword: keyword(SwiftSyntax.Keyword.func)
├─name: identifier("test")
└─signature: FunctionSignatureSyntax
  ├─parameterClause: FunctionParameterClauseSyntax
  │ ├─leftParen: leftParen
  │ ├─parameters: FunctionParameterListSyntax
  │ │ └─[0]: FunctionParameterSyntax
  │ │   ├─attributes: AttributeListSyntax
  │ │   ├─modifiers: DeclModifierListSyntax
  │ │   ├─firstName: identifier("int")
  │ │   ├─colon: colon
  │ │   └─type: IdentifierTypeSyntax
  │ │     └─name: identifier("Int")
  │ └─rightParen: rightParen
  └─returnClause: ReturnClauseSyntax
    ├─arrow: arrow
    └─type: IdentifierTypeSyntax
      └─name: identifier("Int")
```

AST-дерево

AST (абстрактное синтаксическое дерево) предоставляет нам конструкции языка в виде дерева выражений.

Для работы с AST используется пакет swift-syntax который является обязательным при написании макросов.

Каждая внутренняя вершина этого дерева - синтаксическая конструкция языка. Каждый лист дерева - идентификатор.

With great power comes great responsibility

Сложности с swift-syntax

Система типов

Работа с системой типов в SwiftSyntax немного отличается от общепринятых практик в iOS разработки и некоторые операции будут выглядеть непривычными на первый взгляд

Value types

Собственные методы для приведения типов и других операций

With-pattern

Паттерн для изменения данных используемый только в swift-syntax

Типы в SwiftSyntax

SwiftSyntax оперирует синтаксическими конструкциям (такими как FuncDeclSyntax, TypeSyntax, DeferStmtSyntax, DeclReferenceExprSyntax), структура которых может быть достаточно непредсказуемой.

Вложенность конструкций может стремиться в бесконечность.

```
declaration → import-declaration
declaration → constant-declaration
declaration → variable-declaration
declaration → typealias-declaration
declaration → function-declaration
declaration → enum-declaration
declaration → struct-declaration
declaration → class-declaration
declaration → actor-declaration
declaration → protocol-declaration
declaration → initializer-declaration
declaration → deinitializer-declaration
declaration → extension-declaration
declaration → subscript-declaration
declaration → macro-declaration
declaration → operator-declaration
declaration → precedence-group-declaration \
```

Type Casting

Предположим нам пришел DeclSyntax в наш макрос. Как проверить что он нужного нам типа и привести его к этому типу.

```swift
if let declSyntax = declSyntax as? ProtocolDeclSyntax {

}
```

```swift
if let declSyntax = declSyntax.as(ProtocolDeclSyntax.self) {

}
```

Так как и DeclSyntax, и ProtocolDeclSyntax являются структурами в swift-syntax мне не можем использовать стандартный type casting.

With pattern

Предположим у нас есть FuncDeclSyntax и мы хотим взять его как есть и изменить какое-либо из полей чтобы получить новый FuncDeclSyntax

```
funcDecl.modifiers = DeclModifierListSyntax {
    DeclModifierSyntax(name: TokenSyntax.keyword(.public))
}
return funcDecl
```

```
return funcDecl
    .with(\.modifiers, DeclModifierListSyntax {
        DeclModifierSyntax(name: TokenSyntax.keyword(.public))
    })
```

Все свойства у структур в SwiftSyntax неизменяемые, поэтому нам необходимо использовать with метод с KeyPath.

Сравнение токенов

В SwiftSyntax все наше выравние кода сохраняется, поэтому мы не можем сравнить токены напрямую

```
token == SyntaxToken.keyword(.public)
```

```
token.trimmed.text == SyntaxToken.keyword(.public).text
```

Внутреннее представление токене полученного при парсинге может отличаться от того, что мы ожидаем. Токены надо сравнивать текстом.

Trivia

В SwiftSyntax все наше выравние кода сохраняется, поэтому мы не можем просто переиспользовать данные которые нам пришли с парсера

```
return funcDecl
    .with(\.modifiers, DeclModifierListSyntax {
        DeclModifierSyntax(name: TokenSyntax.keyword(.public))
    })
```

```
return funcDecl
    .with(\.modifiers, DeclModifierListSyntax {
        DeclModifierSyntax(name: TokenSyntax.keyword(.public))
    })
    .with(\.leadingTrivia, Trivia(pieces: []))
```

Очищайте trivia у тех данных, которые собираетесь переиспользовать в сгенерированном коде

Варианты методов

```swift
// By effect specifier
public protocol ExampleProtocol1 {
    func simple(int: Int) -> Int
    func asyncSimple(int: Int) async -> Int
    func throwsSimple(int: Int) throws -> Int
    func asyncThrowsSimple(int: Int) async throws -> Int
    func rethrowsSimple(f: (Int) throws -> Int) rethrows -> Int
    func asyncRethrowsSimple(f: (Int) throws -> Int) async rethrows -> Int
}

// By argument types
public protocol ExampleProtocol2 {
    func simpleGeneric<T>(argument: T)
    func simpleVardic(argument: Int...)
    func simpleSome(argument: some Decodable)
    func simpleAny(argument: any Decodable)
    func simpleEscaping(argument: @escaping (Int) -> Int)
    func simpleAutoClosure(argument: @autoclosure (Int) -> Int)
}
```

Проблема

Обработать все варианты невозможно

- Совет 1
  Старайтесь проверять что конструкция поддерживается
- Совет 2
  Старайтесь модифицировать конструкции
- Совет 3
  Поддерживаемые конструкции должны быть покрыты тестами

ТИНЬКОФФ

Zalutskiy Alexandr I Ведущий разработчик

metalhead.sanya@gmail.com

@metalheadsanya

+7 (707) 508-60-98

ТИНЬКОФФ