

Protocol Extension

ishkawa



Yosuke Ishikawa

ishkawa

Developer Program Member

LINE

Tokyo, Japan

y@ishkawa.org

<http://ishkawa.org>

Joined on Aug 9, 2011

241
Followers

299
Starred

44
Following

Organizations



Contributions

Repositories

Public activity

Edit profile

Popular repositories



APIKit

A networking library for building type safe web...

270 ★



ISRefreshControl

iOS4-compatible UIRefreshControl

237 ★



ISColumnsController

paginated container view controller.

121 ★



ISDiskCache

LRU disk cache for iOS.

69 ★



UINavigationController-Transit...

extension for custom transition by blocks.

53 ★

Repositories contributed to



antityypical/Result

396 ★

Swift type modelling the success/failure of arbit...



AliSoftware/OHHTTPStubs

1,538 ★

Stub your network requests easily! Test your a...



CocoaPods/Specs

2,390 ★

The CocoaPods Master Repo



Alamofire/Alamofire

9,152 ★

Elegant HTTP Networking in Swift

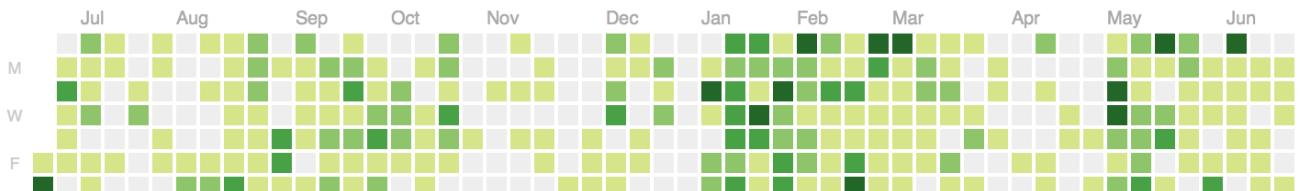


matteocrippa/awesome-swift

3,684 ★

A collaborative list of awesome swift resource...

Contributions



Less More

Contributions in the last year

1,815 total

Jun 27, 2014 – Jun 27, 2015

Longest streak

64 days

January 9 – March 13

Current streak

0 days

Last contributed 12 hours ago

Protocol Extension

```
protocol MyProtocol {  
    var name: String { get }  
    func doSomething()  
}
```

```
extension MyProtocol {  
    var name: String {  
        return "ishkawa"  
    }  
  
    func doSomething() {  
    }  
}
```

Swift 1 の protocol

Swift 1 の protocol

- インターフェースを定義

```
protocol DataSource {  
    func numberOfSections() -> Int  
    func numberOfItemsInSection(section: Int) -> Int  
}
```

Swift 1 の protocol

- 適合する型で実装

```
class ViewController: UIViewController, DataSource {  
    func numberOfSections() -> Int {  
        return 1  
    }  
  
    func numberOfItemsInSection(section: Int) -> Int {  
        return 1  
    }  
}
```

Swift 1 の protocol

- @objcなら optional を定義できる

```
@objc protocol DataSource {  
    optional func numberOfSections() -> Int  
    func numberOfItemsInSection(section: Int) -> Int  
}
```

Swift 1 の protocol

- optional は実装しなくても良い

```
class ViewController: UIViewController, DataSource {  
    func numberOfItemsInSection(section: Int) -> Int {  
        return 1  
    }  
}
```

Swift 1 の protocol

- `numberOfSections()` の実装の有無は呼び出し側でハンドル
- デフォルトの動作は利用側が用意する

```
let dataSource: DataSource  
let numberOfRowsInSection = dataSource.numberOfSections() ?? 1
```

Swift 2 の protocol

Swift 2 の protocol

- インターフェースを定義

```
protocol DataSource {  
    func numberOfSections() -> Int  
    func numberOfItemsInSection(section: Int) -> Int  
}
```

- extension でデフォルトの実装を定義

```
protocol DataSource {  
    func numberOfSections() -> Int {  
        return 1  
    }  
}
```

Swift 2 の protocol

- protocol extension で実装されているものは実装しなくても良い

```
class ViewController: UIViewController, DataSource {  
    func numberOfItemsInSection(section: Int) -> Int {  
        return 1  
    }  
}
```

Swift 2 の protocol

- デフォルトの動作は protocol extension が用意している
- 利用側は適合している型が実装しているか気にする必要はない

```
let dataSource: DataSource  
let numberOfSections = dataSource.numberOfSections()
```

Swift 2 の protocol

- protocolにデフォルト実装を定義できるようになった

デフォルト実装による設計の変化

- @objcにしなくてもoptionalのようなものを提供できる
- デフォルトの動作の定義を利用側からprotocol側に移せる

```
// optionalの場合(Swift 1, 2)
let numberOfRowsInSection = dataSource.numberOfSections?() ?? 1

// protocol extensionの場合(Swift 2)
let numberOfRowsInSection = dataSource.numberOfSections
```

Swift標準ライブラリの変化

Swift 1 の map

Swift 1 の map

- 定義

```
func map<C: CollectionType, T>
(source: C, transform: (C.Generator.Element) -> T) -> [T]
```

- 利用例

```
let names = ["foo", "bar", "baz"]
let counts = map(names) { name in
    count(name)
}
```

- 汎用的だがグローバル関数はコード補完から探すのが面倒

Swift 1 の map

- 型ごとにメソッドとしても定義されている

```
extension Array {  
    func map<U>(transform: (T) -> U) -> [U]  
}
```

- 利用例

```
let names = ["foo", "bar", "baz"]  
let counts = names.map { name in  
    count(name)  
}
```

- 型ごとに定義...

Swift 2 の map

グローバル関数 → protocol extension

Swift 2 の map

- 定義

```
extension CollectionType {
    func map<T>
        (@noescape transform: (Self.Generator.Element) -> T) -> [T]
}
```

- 利用例

```
let counts: [Int] = names.map { name in
    name.characters.count
}
```

- 型ごとに定義しなくてもメソッドとして利用できる

mapはどう変わったか

- 達成したい条件
 - CollectionTypeに対して適用できるようにしたい
 - メソッドとして提供したい
- Swift 1
 - グローバル関数でCollectionTypeへの適用を提供
 - Arrayなど型ごとにメソッドを提供
- Swift 2
 - CollectionTypeのprotocol extensionでメソッドを提供

型制約つき protocol extension

型制約つきグローバル関数

- 特定の条件を満たす型だけに実行できる関数もある

```
// Swift 1
func sorted<C: SequenceType where C.Generator.Element: Comparable:
(source: C) -> [C.Generator.Element]
```

SequenceTypeに適合している型Cのうち、要素となる
C.Generator.ElementがComparableに適合していればsorted()を
実行できる。

型制約つき protocol extension

- 条件を満たす型のみprotocol extensionを定義することもできる

```
extension SequenceType where Self.Generator.Element : Comparable  
    func sort() -> [Self.Generator.Element]  
}
```

Protocol Oriented Programming

WWDC 2015の"Protocol Oriented Programming in Swift"
というトークから理解したことをまとめます。

3行で

- classにはいくつか問題がある
- protocol + structで解決できる
- なるべく protocol + structを使おう

抽象クラスの継承の問題

- 何をoverrideすればいいのかわからない
- ドキュメント以外から知ることは難しい

```
// 抽象クラス
class Ordered {
    func precedes(other: Ordered) -> Bool {
        fatalError("implement me!")
    }
}
```

overrideが必要なものの抽象クラス側での実装を、`fatalError`にしておけばサブクラスでの実装が必要であることを知らせることはできる。

- エラーはコンパイル時に知りたいが`fatalError()`は実行時

抽象クラスの継承の問題

- protocolでは実装すべきものが明らか

```
// 抽象クラスをprotocolに置き換え
protocol Ordered {
    func precedes(other: Ordered) -> Bool
}
```

- 実装漏れはコンパイル時に知ることができる
- デフォルト実装はprotocol extensionで定義できる

抽象化による型の損失

```
class Ordered {
    func precedes(other: Ordered) -> Bool {
        fatalError()
    }
}

class Number: Ordered {
    override func precedes(other: Ordered) -> Bool {
        let otherNumber = other as! Number
        ...
    }
}
```

- overrideしたメソッドの型は変更できない
- Numberのprecedes()に他のサブクラスが入る可能性もある
- つまり、抽象的なコードを型安全に書けない

抽象化による型の損失

```
protocol Ordered {  
    func precedes(other: Self) -> Bool  
}  
  
struct Number: Ordered {  
    func precedes(other: Number) -> Bool {  
        ...  
    }  
}
```

- protocolではSelfが利用できる
- precedes()の引数にはNumber以外の型は入らない

抽象的かつ型安全なコードの他の例

```
public protocol Request {
    typealias Response
    func execute(Response -> Void)
}

struct GetRepositoryRequest: Request {
    typealias Response = Repository
}

struct GetUserRequest: Request {
    typealias Response = User
}

let repositoryRequest = GetRepositoryRequest()
repositoryRequest.execute { response in
    // responseはRepository
}

let userRequest = GetUserRequest()
userRequest.execute { response in
    // responseはUser
}
```

どちらを採用するべきか

どちらもインターフェースと実装の両方を提供し、継承が可能という点では似ているが、抽象的なコードにはプロトコルの方が向いている。

- 抽象クラスは実装の要求が不明確
- 抽象クラスでは抽象的なコードを型安全に書けない

とはいっても、プロトコルに弱点がないわけでもない

- デフォルト実装(クラスのsuper相当)への参照ができない
- stored propertyを持つことができない

まとめ

- プロトコルにデフォルト実装が定義できるようになった
- 標準ライブラリもそれを活かしたつくりに変わってきた
- 抽象クラスをプロトコルで置き換えるという流れがある

