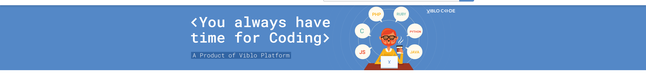

Nguyễn Quốc Tinh @tinhnq | Theo dõi
854 38 21

Đã đăng vào thg 4 10, 2019 4:36 PM · 9 phút đọc

Sự hữu dụng của type alias trong Swift

Một typealias cho phép chúng ta cung cấp một tên mới cho một loại dữ liệu hiện có. Sau khi một typealias được khai báo, nó có thể được sử dụng thay vì loại hiện có trong suốt chương trình. Typealias không tạo ra loại mới. Nó chỉ đơn giản là cung cấp một tên mới cho một loại hiện có. Mục đích chính của typealias là làm cho mã của chúng ta dễ đọc hơn và rõ ràng hơn trong ngữ cảnh cho sự hiểu biết của con người. Chúng ta cùng xem xét sự hữu dụng của nó trong các trường hợp sau đây.

Semantic types

Nhìn chung, việc đưa *ngữ nghĩa* vào trong các kiểu dữ liệu có thể tạo nên một sự khác biệt đáng kể giúp cho code thêm trực quan và làm việc với chúng dễ dàng hơn. Điều này đặc biệt đúng khi *xử lý* các kiểu nguyên thủy (như số hay strings), vì chúng được sử dụng ở mọi nơi. Khi chúng ta thấy một số Int, Double hay String trong một hàm, chúng ta thường phải đưa vào tên của hàm đó và các tham số của nó, để hiểu giá trị đó sẽ được sử dụng để làm gì. Lấy kiểu dữ liệu TimeInterval trong thư viện Foundation làm ví dụ. Bất cứ khi nào chúng ta thấy một giá trị TimeInterval chúng ta biết rằng chúng ta đang làm việc với thời gian (cụ thể là giây), điều đó sẽ đúng nếu một loại số thô thô như Double được sử dụng thay thế. Tuy nhiên, hóa ra TimeInterval thực sự không phải là một kiểu dữ liệu mới và trên thực tế chỉ là một bí danh cho Double:

```
typealias TimeInterval = Double
```

Sử dụng một bí danh như thế có ưu và nhược điểm. Vì TimeInterval thực sự không phải là một kiểu riêng, mà là bí danh - có nghĩa là tất cả các giá trị Double là giá trị TimeInterval đều dữ dụng được cho nhau. Nhược điểm của điều đó là chúng tôi không có được sự an toàn về thời gian biên dịch mà chúng tôi sẽ làm nếu chúng tôi tạo ra một loại hoàn toàn mới thay vào đó, nhưng mặt khác, lợi ích là bất kỳ phương thức nào của Double (bao gồm cả toán tử) cũng có thể được sử dụng trên các giá trị TimeInterval - giảm sự trùng lặp. Tương tự như TimeInterval, chúng ta cũng có thể có những kiểu dữ liệu với những cái tên rất trực quan nhưng giá trị thực sự lại chỉnh là những kiểu dữ liệu sẵn có:

```
typealias Kilograms = Double

struct Package {
    var weight: Kilograms
}
```

Những điều nhỏ nhặt như trên tuy là rất đơn giản nhưng sẽ làm cho code của chúng ta trở nên rõ ràng hơn rất nhiều.

Specializing generics

Typealias cũng cung cấp một cách dễ dàng để chuyên môn hóa, đặc biệt là các loại được sử dụng với cùng loại chung trong toàn bộ cơ sở mã của chúng ta. Giả sử chúng ta tạo một kiểu dữ liệu FileStorage được sử dụng để làm việc với cả hệ thống file local hay remote:

```
class FileStorage<Key: Hashable, Location: FileStorageLocation> {
    ...
}
```

Sử dụng một loại như vậy có thể thực sự tiện lợi, vì nó cho phép chúng ta chứa tất cả các mã cần thiết để xử lý bất kỳ loại hệ thống file nào ở một nơi, trong khi vẫn cho phép chuyên môn hóa tại trang web gọi tới. Ví dụ: NoteSyncContoder có thể sử dụng hai phiên bản của lớp trên - một để theo dõi tất cả các ghi chú được lưu trữ trên thiết bị của người dùng và một để tải tệp lên đám mây, như sau:

```
class NoteSyncController {
    init(localStorage: FileStorage<Note.StorageKey, LocalFileStorageLocation>
         cloudStorage: FileStorage<Note.StorageKey, CloudStorageLocation>) {
        ...
    }
}
```

Tuy nhiên, việc phải nhập các chuyên môn FileStorage dài đó mỗi khi chúng ta sử dụng chúng có thể nhanh chóng trở nên khá tẻ nhạt và khiến code của chúng ta khó đọc hơn - đặc biệt là khi nhiều đối tượng chuyên biệt như vậy được sử dụng ở cùng một nơi, như trên. Đây là một tình huống khác trong đó các typealias có thể trở nên rất hữu ích, vì vậy cơ bản chúng cho phép chúng ta thực hiện chuyên môn đó một lần - và tạo các loại chuyên dụng, nhẹ cho từng trường hợp sử dụng. Trong kịch bản này, chúng ta có thể mở rộng mã hình Lưu ý của mình để chứa hai bí danh loại như vậy, một cho LocalStorage và một cho CloudStorage:

```
extension Note {
    typealias LocalStorage = FileStorage<StorageKey, LocalFileStorageLocation>
    typealias CloudStorage = FileStorage<StorageKey, CloudStorageLocation>
}
```

Với cách đặt như trên, chúng ta đã làm cho việc khởi tạo NoteSyncController trở nên gọn gàng hơn rất nhiều:

```
class NoteSyncController {
    init(localStorage: Note.LocalStorage,
         cloudStorage: Note.CloudStorage) {
        ...
    }
}
```

Thay đổi ở trên cũng che giấu chi tiết thực hiện. Mặc dù các chi tiết đó vẫn có thể truy cập được khi cần, chúng ta không còn phải nhảm lẫn với các kiểu dữ liệu, tham số dài loằng ngoằng nữa, tất cả giờ đã được đặt tên và trông rõ ràng hơn rất nhiều.

Type-driven logic

Để làm việc với các indexes một cách thống nhất trong toàn bộ code base của chúng ta, chúng ta có thể tạo một loại Index chung - sau đó có thể được chuyên môn hóa thông qua giao thức Indexed. Vì Index chỉ có thể được sử dụng với các loại conform với Indexed, chúng ta có thể sử dụng loại RawIndex của nó để xác định giá trị cơ bản mà index của chúng ta được tạo thành:

```
protocol Indexed {
    associatedtype RawIndex
    var index: Index<Self> { get }
}

struct Index<Object: Indexed> {
    typealias RawValue = Object.RawIndex
    let rawValue: RawValue
}
```

Với thiết lập ở trên, bây giờ chúng ta có thể sử dụng các typealias để khai báo chúng ta muốn mỗi loại được indexed như thế nào. Ví dụ: Người dùng có thể được indexed dựa trên số identifier, trong khi Album (nếu chúng ta xây dựng ứng dụng âm nhạc) có thể indexed theo thể loại âm nhạc:

```
extension User: Indexed {
    typealias RawIndex = Identifier<User>
}

extension Album: Indexed {
    typealias RawIndex = Genre
}
```

Điều thú vị ở trên là giờ đây chúng ta có thể tạo các indexes hoàn toàn an toàn, sử dụng thông tin được cung cấp bởi các bí danh loại trên của chúng ta để đảm bảo rằng giá trị thô chính xác được sử dụng:

```
let albumIndex = Index<Album>(rawValue: .rock)
```

Generic closure aliases

Cuối cùng, chúng ta hãy xem cách sử dụng typealias để tạo ra các generic shorthands cho closures. Ví dụ: nếu code base của chúng ta sử dụng nhiều loại Result để mô hình hóa các kết quả khác nhau của các hoạt động không đồng bộ - chúng ta có thể muốn define một shorthand cho closure có kết quả như vậy, đó là cách chúng ta sẽ sử dụng nhiều nhất cho nhiều completion handlers của chúng ta:

```
typealias Handler<T> = (Result<T>) -> Void
```

Bây giờ, bất cứ khi nào một trong các chức năng của chúng ta chấp nhận completion handler, chúng ta chỉ cần sử dụng Handler type ở trên và chuyển môn hóa với bất kỳ kết quả mà chúng ta sẽ chuyển vào handler đó:

```
func searchForNotes(matching query: String,
                    then handler: @escaping Handler<[Note]>) {
    ...
}
```

Một lần nữa, sự thay đổi ở trên có vẻ như hoàn toàn là một mỹ phẩm, nhưng nó có thể ảnh hưởng lớn đến việc mã của chúng tôi có thể đọc và viết như thế nào - đặc biệt là khi cơ sở mã của chúng tôi phát triển, và chúng tôi kết thúc với một số lượng lớn các khai báo hàm mà sử dụng xử lý hoàn thành.

Tóm lại

Typealias là một trong những tính năng Swift ban đầu có vẻ rất đơn giản, nhưng một khi chúng ta đi sâu vào và xem xét kỹ hơn, hóa ra chúng có khả năng trong nhiều tình huống. Mặc dù việc sử dụng chúng quá mức có thể khiến code base của chúng ta khó đọc/bảo hơn (trong trường hợp chúng ta liên tục cần tìm loại cụ thể nào đằng sau mỗi bí danh), tuy nhiên nếu sử dụng chúng hợp lý sẽ giúp cho code đơn giản và thanh lịch hơn.

Nguồn: https://www.swiftbysundell.com/posts/the-power-of-type-aliases-in-swift

iOS Swift

MỤC LỤC

CÁC TỔ CHỨC ĐƯỢC ĐỀ XUẤT

Sun* Cyber Security Team
176 10 270

Sun* Blockchain Team
163 8 238

Avengers Group
55 35 120

Bài viết liên quan

3 lỗi nghiêm trọng của lập trình viên iOS
Phương VNC
13 phút đọc
1070 5 6

Logic Controllers trong Swift
Nguyễn Thanh Tùng
13 phút đọc
457 2 0 0

Static Factory Methods In Swift
Nguyễn Thanh Tùng
11 phút đọc
215 2 0 3

Ultimate Guide to JSON Parsing With Swift 4
Nguyen Cong Anh
3 phút đọc
1109 2 0 1

Bài viết khác từ Nguyễn Quốc Tinh

(SwiftUI) GridStack - layout lưới trong vài dòng code
Nguyễn Quốc Tinh
4 phút đọc
253 0 0 2

Empty Strings in Swift
Nguyễn Quốc Tinh
2 phút đọc
260 1 0 1

Mở rộng ứng dụng iOS bằng module
Nguyễn Quốc Tinh
13 phút đọc
502 6 0 3

Làm sao để tạo CocoaPods library?
Nguyễn Quốc Tinh
9 phút đọc
599 1 0 3

Bình luận