

# Why Swift? Generics(泛型), Collection(集合类型), POP(协议式编程), Memory Management(内存管理)



星光社的戴铭 (/u/9a4903d7e3d1) +关注

2018.01.24 20:34\* 字数 4496 阅读 171 评论 0 喜欢 14

(/u/9a4903d7e3d1)

## 前言

写这篇文章主要是为了给组内要做的分享准备内容。这段时间几个项目都用到 Swift，在上次 GIAC 大会上就被问到为什么要用 Swift，正好这个主题可以聊聊 Swift 的哪些特性吸引了我。

## 泛型

先来个例子看下泛型是解决什么问题的。

```
let nations = ["中国", "美国", "日本"]
func showNations(arr : [String]) {
    arr.map { str in
        print("\(str)")
    }
}
```

我们先定一个字符串数组，然后把里面的字符串打印出来。这里的 map 写法还可以优化下：

```
arr.map { print("\( $0)") }
```

那么还能做什么优化呢。将 showNations 的入参数组泛型以支持多类型，比如 [int], [double] 等。

```
func showArray<T>(arr: [T]) {
    arr.map { print("\( $0)") }
}
```

可以看出泛型能够很方便的将不同类型数据进行相同操作的逻辑归并在一起。

## 类型约束

先看下我的 HTN 项目里状态机的 Transition 结构体的定义

```
struct HTNTransition<S: Hashable, E: Hashable> {
    let event: E
    let fromState: S
    let toState: S

    init(event: E, fromState: S, toState: S) {
        self.event = event
        self.fromState = fromState
        self.toState = toState
    }
}
```



这里的 `fromState`, `toState` 和 `event` 可以是不同类型的数据, 可以是枚举, 字符串或者整数等, 定义 `S` 和 `E` 两个不同的泛型可以让状态和事件类型不相同, 这样接口会更加的灵活, 更容易适配更多的项目。

大家会注意到 `S` 和 `E` 的冒号后面还有个 `Hashable` 协议, 这就是要求它们符合这个协议的类型约束。使用协议的话可以使得这两个类型更加的规范和易于扩展。

Swift 的基本类型 `String`, `Int`, `Double` 和 `Bool` 等都是遵循 `Hashable` 的, 还有无关联值的枚举也是的。`Hashable` 提供了一个 `hashValue` 方法用在判断遵循协议对象是否相等时用。

`Hashable` 协议同时也是遵守 `Equatable` 协议, 通过实现 `==` 运算符来确定自定义的类或结构是否相同。

## 关联类型

在协议里定义的关联类型也可以用泛型来处理。比如我们先定义一个协议

```
protocol HTNState {
    associatedtype StateType
    func add(_ item: StateType)
}
```

采用非泛型的实现如下:

```
struct states: HTNState {
    typealias StateType = Int
    func add(_ item: Int) {
        //...
    }
}
```

采用泛型遵循协议可以按照下面方式来写:

```
struct states<T>: HTNState {
    func add(_ item: T) {
        //...
    }
}
```

这样关联类型也能够享受泛型的好处了。

## 类型擦除

但是在使用关联类型的时候需要注意当声明一个使用了关联属性的协议作为属性时, 比如下面的代码:

```
class stateDelegate<T> {
    var state: T
    var delegate: HTNState
}
```

先会提示 `no initializers` 的错误, 接着会提示 `error: protocol 'HTNState' can only be used as a generic constraint because it has Self or associated type requirements`。意思是 `HTNState` 协议只能作为泛型约束来用, 因为它里面包含必需的 `self` 或者关联类型。

那么该如何处理呢? 这里需要通过类型擦除来解决, 主要思路就是加个中间层在代码中让这个抽象的类型具体化。实际上在 Swift 的标准库里就有类型擦除很好的运用, 比如 `AnySequence` 的协议。

## Where 语句

函数，扩展和关联类型都可以使用 where 语句。where 语句是对泛型在应用时的一种约束。比如：

```
func stateFilter<FromState:HTNState, ToState:HTNState>(_ from:FromState, _ to:ToState) {
    //...
}
```

这个函数就要求他们的 StateType 具有相同类型。

## 泛型和 Any 类型

这两个类型看起来很相似，但是一定要小心两者的区别。他们区别在于 Any 类型会避开类型的检查，所以尽量少用最好不用。泛型一方面很灵活一方面也很安全，下面举个例子感受下两者的区别：

```
func add<T>(_ input: T) -> T {
    //...
    return input;
}

func anyAdd(_ input: Any) -> Any {
    //...
    return input;
}
```

这两个函数都是可以允许任意类型的 input 参数，不同在于返回的类型在 anyAdd 函数里是可以和入参不一样的，这样就会失控，在后续的操作中容易出错。

## 集合

### 基本概念

先来了解下集合的基本概念，首先集合是泛型的比如：

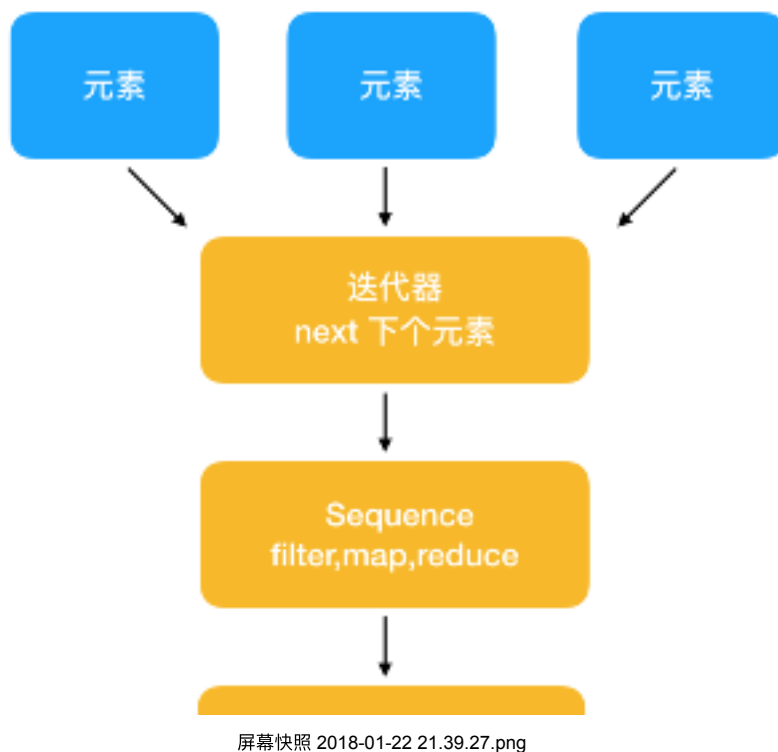
```
let stateArray: Array<String> = ["工作", "吃饭", "玩游戏", "睡觉"]
```

集合它需要先有个遍历的功能，通过 GeneratorType 协议，可以不关注具体元素类型只要不断的用迭代器调 next 就可以得到全部元素。但是使用迭代器没法进行多次的遍历，这时就需要使用 Sequence 来解决这个问题。像集合的 forEach, elementsEqual, contains, minElement, maxElement, map, flatMap, filter, reduce 等功能都是因为有了 Sequence 的多次遍历。

最后 Collection 概念是因为 Sequence 无法确定集合里的位置而在 Sequence 的基础上实现了 Indexable 协议。有了 Collection 就可以确定元素的位置，包括开始位置和结束位置，这样就能够确定哪些元素是已经访问过的，从而避免多次访问同一个元素。还能够通过一个给定的位置直接找到那个位置的元素。

以上描述如下图：





## 迭代器

Swift 里有个简单的 `AnyIterator<Element>` 结构体

```

struct AnyIterator<Element>: IteratorProtocol {
    init(_ body: @escaping () -> Element?)
    //...
}

```

`AnyIterator` 实现了 `IteratorProtocol` 和 `Sequence` 协议。通过下面的例子我们来看看如何使用 `AnyIterator` :

```

class stateItr : IteratorProtocol {
    var num: Int = 1
    func next() -> Int? {
        num += 2
        return num
    }
}

func findNext<I: IteratorProtocol>( elm: I) -> AnyIterator<I.Element> where I.Element == Int {
    {
        var l = elm
        print("\(l.next() ?? 0)")
        return AnyIterator { l.next() }
    }
}

findNext(elm: findNext(elm: findNext(elm: stateItr())))

```

首先是定义个遵循了 `IteratorProtocol` 并实现了 `next` 函数的类。再实现一个 `AnyIterator` 的迭代器方法，这样通过这个方法的调用就可以不断的去找符合的元素了。

这里有个对 `where` 语句的运用，`where I.Element == Int`。如果把这句改成 `where I.Element == String` 会出现下面的错误提示



```
Playground execution failed:

error: MyPlayground.playground:18:37: error: cannot invoke 'findNext(elm:)' with an
findNext(elm: findNext(elm: findNext(elm: stateItr())))
                        ^

MyPlayground.playground:11:6: note: candidate requires that the types 'Int' and 'St
func findNext<I: IteratorProtocol>( elm: I) -> AnyIterator<I.Element> where I.Element
      ^
```

编译器会在代码检查阶段通过代码跟踪就发现类型不匹配的安全隐患，这里不得不对 Swift 的设计点个赞

## Sequence

上面的迭代器只会以单次触发的方式反复计算下个元素，但是对于希望能够重新查找或重新生成已生成的元素，这样还需要有个新的迭代器和一个子 Sequence。在 Sequence 协议里可以看到这样的定义：

```
public protocol Sequence {
    //Element 表示序列元素的类型
    associatedtype Element where Self.Element == Self.Iterator.Element
    //迭代接口类型
    associatedtype Iterator : IteratorProtocol
    //子 Sequence 类型
    associatedtype SubSequence
    //返回 Sequence 元素的迭代器
    public func makeIterator() -> Self.Iterator
    //...
}
```

重新查找靠的是这个新的迭代器，而对于切片这样的会重新生成新 Sequence 的操作就需要 SubSequence 进行存储和返回。

## Collection

对 Sequence 进行进一步的完善，最重要的就是使其具有下标索引，使得元素能够通过下标索引方式取到。Collection 是个有限的范围，有开始索引和结束索引，所以 Collection 和 Sequence 的无限范围是不一样的。有了有限的范围 Collection 就可以有 count 属性进行计数了。

除了标准库里的 String，Array，Dictionary 和 Set 外比如 Data 和 IndexSet 也由于遵循了 Collection 协议而获得了标准库了那些集合类型的能力。

## map

在泛型的第一个例子里我们就看到了 map 的使用，我们看看 map 的定义：

```
func map<T>(transform: (Self.Generator.Element) -> T) rethrows -> [T]
```

这里 (Self.Generator.Element) -> T 就是 map 闭包的定义，Self.Generator.Element 就是当前元素的类型。

## flatMap

二维数组经过 flatmap 会降维到一维，还能过滤掉 nil 值。下面看看 Swift 源码 (swift/stdlib/public/core/SequenceAlgorithms.swift.gyb) 中 flatmap 的实现：



```

//=====//
// flatMap()
//=====//

extension Sequence {
    /// Returns an array containing the concatenated results of calling the
    /// given transformation with each element of this sequence.
    ///
    /// Use this method to receive a single-level collection when your
    /// transformation produces a sequence or collection for each element.
    ///
    /// In this example, note the difference in the result of using `map` and
    /// `flatMap` with a transformation that returns an array.
    ///
    ///     let numbers = [1, 2, 3, 4]
    ///
    ///     let mapped = numbers.map { Array(count: $0, repeatedValue: $0) }
    ///     // [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
    ///
    ///     let flatMapped = numbers.flatMap { Array(count: $0, repeatedValue: $0) }
    ///     // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
    ///
    /// In fact, `s.flatMap(transform)` is equivalent to
    /// `Array(s.map(transform).joined())`.
    ///
    /// - Parameter transform: A closure that accepts an element of this
    ///   sequence as its argument and returns a sequence or collection.
    /// - Returns: The resulting flattened array.
    ///
    /// - Complexity: O(*m* + *n*), where *m* is the length of this sequence
    ///   and *n* is the length of the result.
    /// - SeeAlso: `joined()`, `map(_)`
    public func flatMap<SegmentOfResult : Sequence>(
        _ transform: ({GElement}) throws -> SegmentOfResult
    ) rethrows -> [SegmentOfResult.Element] {
        var result: [SegmentOfResult.Element] = []
        for element in self {
            result.append(contentsOf: try transform(element))
        }
        return result
    }
}

extension Sequence {
    /// Returns an array containing the non-`nil` results of calling the given
    /// transformation with each element of this sequence.
    ///
    /// Use this method to receive an array of nonoptional values when your
    /// transformation produces an optional value.
    ///
    /// In this example, note the difference in the result of using `map` and
    /// `flatMap` with a transformation that returns an optional `Int` value.
    ///
    ///     let possibleNumbers = ["1", "2", "three", "///4///", "5"]
    ///
    ///     let mapped: [Int?] = possibleNumbers.map { str in Int(str) }
    ///     // [1, 2, nil, nil, 5]
    ///
    ///     let flatMapped: [Int] = possibleNumbers.flatMap { str in Int(str) }
    ///     // [1, 2, 5]
    ///
    /// - Parameter transform: A closure that accepts an element of this
    ///   sequence as its argument and returns an optional value.
    /// - Returns: An array of the non-`nil` results of calling `transform`
    ///   with each element of the sequence.
    ///
    /// - Complexity: O(*m* + *n*), where *m* is the length of this sequence
    ///   and *n* is the length of the result.
    public func flatMap<ElementOfResult>(
        _ transform: ({GElement}) throws -> ElementOfResult?
    ) rethrows -> [ElementOfResult] {
        var result: [ElementOfResult] = []
        for element in self {
            if let newElement = try transform(element) {
                result.append(newElement)
            }
        }
        return result
    }
}

```

从代码中可以看出打平的原理是将集合中所有元素都添加到另外一个集合里。在第二个 extension 里通过 if let 语句会挡住那些解包不成功的元素。

## Reduce

Reduce 是编程语言语义学里的归约语义学，也叫累加器。下面同样可以看看 Swift 源码里对其的实现：

```
//=====//
// reduce()
//=====//

extension Sequence {
    /// Returns the result of combining the elements of the sequence using the
    /// given closure.
    ///
    /// Use the `reduce(_:_:)` method to produce a single value from the elements
    /// of an entire sequence. For example, you can use this method on an array
    /// of numbers to find their sum or product.
    ///
    /// The `nextPartialResult` closure is called sequentially with an
    /// accumulating value initialized to `initialResult` and each element of
    /// the sequence. This example shows how to find the sum of an array of
    /// numbers.
    ///
    /// let numbers = [1, 2, 3, 4]
    /// let numberSum = numbers.reduce(0, { x, y in
    ///     x + y
    /// })
    /// // numberSum == 10
    ///
    /// When `numbers.reduce(_:_:)` is called, the following steps occur:
    ///
    /// 1. The `nextPartialResult` closure is called with `initialResult`---`0`
    ///    in this case---and the first element of `numbers`, returning the sum:
    ///    `1`.
    /// 2. The closure is called again repeatedly with the previous call's return
    ///    value and each element of the sequence.
    /// 3. When the sequence is exhausted, the last value returned from the
    ///    closure is returned to the caller.
    ///
    /// If the sequence has no elements, `nextPartialResult` is never executed
    /// and `initialResult` is the result of the call to `reduce(_:_:)`.
    ///
    /// - Parameters:
    ///   - initialResult: The value to use as the initial accumulating value.
    ///     `initialResult` is passed to `nextPartialResult` the first time the
    ///     closure is executed.
    ///   - nextPartialResult: A closure that combines an accumulating value and
    ///     an element of the sequence into a new accumulating value, to be used
    ///     in the next call of the `nextPartialResult` closure or returned to
    ///     the caller.
    /// - Returns: The final accumulated value. If the sequence has no elements,
    ///   the result is `initialResult`.
    public func reduce<Result>(_ initialResult: Result,
                              _ nextPartialResult:
                                (_ partialResult: Result, $GEElement) throws -> Result)
        throws -> Result {
        var accumulator = initialResult
        for element in self {
            accumulator = try nextPartialResult(accumulator, element)
        }
        return accumulator
    }
}
```

可以看到里面会通过 `initialResult` 来记录前面的返回结果和当前元素进行在闭包里的操作。

## Array

看看数组的基本用法



```
//创建数组
var nums = [Int]() //创建空数组
var mArray = nums + [2,3,5] + [5,9]//合并多个有相同类型元素数组的值
var animals: [String] = ["dragon", "cat", "mice", "dog"]

//添加数组
animals.append("bird")
animals += ["ant"]

//获取和改变数组
var firstItem = mArray[0]
animals[0] = "red dragon"
animals[2...4] = ["black dragon", "white dragon"] //使用下标改变多个元素
animals.insert("chinese dragon", at: 0) //在索引值之前添加元素
let mapleSyrup = animals.remove(at: 0) //移除数组中的一个元素
let apples = animals.removeLast() //移除最后一个元素

////数组遍历
for animal in animals {
    print(animal)
}
for (index, animal) in animals.enumerated() {
    print("animal \(String(index + 1)): \(animal)")
}
/*
animal 1: red dragon
animal 2: cat
animal 3: black dragon
animal 4: white dragon
*/
```

## 弱引用的 Swift 数组

Swift 里的数组默认会强引用里面的元素，但是有时候可能希望能够弱引用，那么就可以使用 `NSPointerArray`。它在初始化的时候可以决定是用弱引用方式还是强引用方式。

```
let strongArr = NSPointerArray.strongObjects() // 强引用
let weakArr = NSPointerArray.weakObjects() // Maintains weak references
```

Dictionary 的要想用弱引用可以使用 `NSMapTable`，Set 对应的是 `NSHashTable`。

## Dictionary

看看基本用法：

```
//创建 Dictionary
var strs = [Int: String]()
var colors: [String: String] = ["red": "#e83f45", "yellow": "#ffe651"]
strs[16] = "sixteen"

//updateValue 这个方法会返回更新前的值
if let oldValue = colors.updateValue("#e83f47", forKey: "red") {
    print("The old value for DUB was \(oldValue).")
}

//遍历
for (color, value) in colors {
    print("\(color): \(value)")
}

//map
let newColorValues = colors.map { "hex:\($0.value)" }
print("\(newColorValues)")

//mapValues 返回完整的新 Dictionary
let newColors = colors.mapValues { "hex:\($0)" }
print("\(newColors)")
```

## 协议式编程

Swift 被设计成单继承，如果希望是多继承就需要使用协议。协议还有个比较重要的作用就是通过 `associatedtype` 要求使用者遵守指定的泛型约束。





下面先看看传统编程的开发模式：

```
class Dragon {  
  
}  
  
class BlackDragon: Dragon{  
    func fire() {  
        print("fire!!!")  
    }  
}  
  
class WhiteDragon: Dragon {  
    func fire() {  
        print("fire!!!")  
    }  
}  
  
BlackDragon().fire()  
WhiteDragon().fire()
```

这个例子可以看出 fire() 就是重复代码，那么首先想到的方法就是通过直接在基类里添加这个方法或者通过 extension 来对他们基类进行扩展：

```
extension Dragon {  
    func fire() {  
        print("fire!!!")  
    }  
}
```

这时我们希望加个方法让 Dragon 能够 fly:

```
extension Dragon {  
    func fire() {  
        print("fire!!!")  
    }  
    func fly() {  
        print("fly~~~")  
    }  
}
```

这样 BlackDragon 和 WhiteDragon 就都有这两个能力了，如果我们设计出一个新的龙 YellowDragon 或者更多 Dragon 都没有 fly 的能力，这时该如何。因为没法多继承，那么没法拆成两个基类，这样必然就会出现重复代码。但是有了协议这个问题就好解决了。具体实现如下：

```
protocol DragonFire {}  
protocol DragonFly {}  
  
extension DragonFire {  
    func fire() {  
        print("fire!!!")  
    }  
}  
extension DragonFly {  
    func fly() {  
        print("fly~~~")  
    }  
}  
  
class BlackDragon: DragonFire, DragonFly {}  
class WhiteDragon: DragonFire, DragonFly {}  
class YellowDragon: DragonFire {}  
class PurpleDragon: DragonFire {}  
  
BlackDragon().fire()  
WhiteDragon().fire()  
BlackDragon().fly()  
YellowDragon().fire()
```

可以看到一来没有了重复代码，二来结构也清晰了很多而且更容易扩展，Dragon 的种类和能力的组合也更加方便和清晰。extension 使得协议有了实现默认方法的能力。



关于多继承 Swift 是采用 Trait 的方式，其它语言 C++ 是直接支持多继承的，方式是这个类会持有多个父类的实例。Java 的多继承只继承能做什么，怎么做还是要自己来。和 Trait 类似的解决方案是 Mixin，Ruby 就是用的这种元编程思想。

协议还可以继承，还可以通过 & 来聚合，判断一个类是否遵循了一个协议可以使用 is 关键字。

当然协议还可以作为类型，比如一个数组泛型元素指定为一个协议，那么这个数组里的元素只要遵循这个协议就可以了。

## Swift 内存管理

### 内存分配

#### Heap

在 Heap 上内存分配的时候需要锁定 Heap 上能够容纳存放对象的空闲块，主要是为了线程安全，我们需要对这些进行锁定和同步。

Heap 是完全二叉树，即除最底层节点外都是填满的，最底层节点填充是从左到右。Swift 的 Heap 是通过双向链表实现。由于 Heap 是可以 retain 和 release 所以很容易分配空间就不连续了。采用链表的目的是希望能够将内存块连起来，在 release 时通过调整链表指针来整合空间。

在 retain 时不可避免需要遍历 Heap，找到合适大小的内存块，能优化的也只是记录以前遍历的情况减少一些遍历。但是 Heap 是很大的，这样每次遍历还是很耗时，而且 release 为了能够整合空间还需要判断当前内存块的前一块和后面那块是否为空闲等，如果空闲还需要遍历链表查询，所以最终的解决方式是双向链表。只把空闲内存块用指针连起来形成链表，这样 retain 时可以减少遍历，效率理论上可以提高一倍，在 release 时将多余空间插入到 Heap 开始的位置和先前移到前面的空间进行整合。

即使效率高了但是还是比不过 Stack，所以苹果也将以前 OC 里的一些放在 Heap 里的类型改造成了值类型。

#### Stack

Stack 的结构很简单，push 和 pop 就完事了，内存上只需要维护 Stack 末端的指针即可。由于它的简单所以处理一些时效性不高，临时的事情是非常合适的，所以可以把 Stack 看成是一个交换临时数据的内存区域。在多线程上，由于 Stack 是线程独有的，所以也不需要考虑线程安全相关问题。

### 内存对齐

Swift 也有内存对齐的概念

```
struct DragonFirePosition {
    var x:Int64 //8 Bytes
    var y:Int32 //4 Bytes
    //8 + 4
}
struct DragonHomePosition {
    var y:Int32 //4 Bytes + 对齐内存(4 Bytes)
    var x:Int64 //8 Bytes
    //4 + 4 + 8
}
let firePositionSize = MemoryLayout<DragonFirePosition>.size //12
let homePositionSize = MemoryLayout<DragonHomePosition>.size //16
```

## Swift 派发机制



派发目的是让 CPU 知道被调用的函数在哪里。Swift 语言是支持编译型语言的直接派发，函数表派发和消息机制派发三种派发方式的，下面分别对这三种派发方式说明下。

## 直接派发

C++ 默认使用的是直接派发，加上 virtual 修饰符可以改成函数表派发。直接派发是最快的，原因是调用指令会少，还可以通过编译器进行比如内联等方式的优化。缺点是由于缺少动态性而不支持继承。

```
struct DragonFirePosition {
    var x:Int64
    var y:Int32
    func land() {}
}

func DragonWillFire(_ position:DragonFirePosition) {
    position.land()
}

let position = DragonFirePosition(x: 342, y: 213)
DragonWillFire(position)
```

编译 inline 后 DragonWillFire(DragonFirePosition(x: 342, y: 213)) 会直接跳到方法实现的地方，结果就变成 position.land()。

## 函数表派发

Java 默认就是使用的函数表派发，通过 final 修饰符改成直接派发。函数表派发是有动态性的，在 Swift 里函数表叫 witness table，大部分语言叫 virtual table。一个类里会用数组来存储里面的函数指针，override 父类的函数会替代以前的函数，子类添加的函数会被加到这个数组里。举个例子：

```
class Fish {
    func swim() {}
    func eat() {
        //normal eat
    }
}

class FlyingFish: Fish {
    override func eat() {
        //flying fish eat
    }
    func fly() {}
}
```

编译器会给 Fish 类和 FlyingFish 类分别创建 witness table。在 Fish 的函数表里有 swim 和 eat 函数，在 FlyingFish 函数表里有父类 Fish 的 swim，覆盖了父类的 eat 和新增加的函数 fly。

一个函数被调用时会先去读取对象的函数表，再根据类的地址加上该的函数的偏移量得到函数地址，然后跳到那个地址上去。从编译后的字节码这方面来看就是两次读取一次跳转，比直接派发还是慢了些。

## 消息机制派发

这种机制是在运行时可以改变函数的行为，KVO 和 CoreData 都是这种机制的运用。OC 默认就是使用的消息机制派发，使用 C 来直接派发获取高性能。Swift 可以通过 dynamic 修饰来支持消息机制派发。

当一个消息被派发，运行时就会按照继承关系向上查找被调用的函数。但是这样效率不高，所以需要通过缓存来提高效率，这样查找性能就能和函数派发差不多了。

## 具体派发

### 声明



值类型都会采用直接派发。无论是 class 还是协议 的 extension 也都是直接派发。class 和协议是函数表派发。

### 指定派发方式

- final: 让类里的函数使用直接派发，这样该函数将会没有动态性，运行时也没法取到这个函数。
- dynamic: 可以让类里的函数使用消息机制派发，可以让 extension 里的函数被 override。

### 派发优化

Swift 会在这上面做优化，比如一个函数没有 override，Swift 就可能会使用直接派发的方式，所以如果属性绑定了 KVO 它的 getter 和 setter 方法可能会被优化成直接派发而导致 KVO 的失效，所以记得加上 dynamic 的修饰来保证有效。后面 Swift 应该会在这个优化上去做更多的处理。

## 基本数据类型内存管理

通过 MemoryLayout 来看看基本数据类型的内存是占用多大

```
MemoryLayout<Int>.size      //8
MemoryLayout<Int16>.size   //2
MemoryLayout<Bool>.size    //1
MemoryLayout<Float>.size   //4
MemoryLayout<Double>.size  //8
```

## Struct 内存管理

对于 Struct 在编译中就能够确定空间，也就不需要额外的空间给运行时用，运行过程调用时就是直接传地址。

下面我们再看看 Struct 的 MemoryLayout

```
struct DragonFirePosition {
    var x:Int64 //8 Bytes
    var y:Int32 //4 Bytes
    //8 + 4
    func land() {}
}

MemoryLayout<DragonFirePosition>.size      //12
MemoryLayout<DragonFirePosition>.alignment //8
MemoryLayout<DragonFirePosition>.stride    //16
```

alignment 可以看出是按照 8 Bytes 来对齐的，所以这里 struct 是用到了字节对齐，实际占用大小通过 stride 可以看出就是 8 \* 2 为16。

如果把 var x:Int64 改成可选类型会增加 4个 Bytes，不过就这个 case 来说实际大小还是 16，这个也是因为内存对齐的原因。

## Class 内存管理

Class 本身是在 Stack 上分配的，在 Heap 上还需要保存 Class 的 Type 信息，这个 Type 信息里有函数表，在函数派发时就可以按照这个函数表进行派发了。继承的话子类只要在自己的 Type 信息里记录自己的信息即可。

## 协议类型内存管理

协议类型的内存模型是 Existential Container。先看下面例子



```
protocol DragonFire {}
extension DragonFire {
    func fire() {
        print("fire!!!")
    }
}

struct YellowDragon: DragonFire {
    let eyes = "blue"
    let teeth = 48
}

let classSize = MemoryLayout<YellowDragon>.size //32
let protocolSize = MemoryLayout<DragonFire>.size //40
```

这个例子里看这个结构体即遵循了这个协议而且内容还比这个协议多，而协议的 size 还要大，这个是为啥呢？这个是由于 Existential Container 的前三个 word 是叫 Value buffer 用来存储 inline 的值。第四个 word 是 Value Witness Table，存储值的各种操作比如 allocate, copy, destruct 和 deallocate 等。第五个 word 是 Protocol Witness Table 是存储协议的函数。

泛型的内存管理

泛型采用的和 Existential Container 原理类似。Value Witness Table 和 Protocol Witness Table 会作为隐形的参数传递到泛型方法里。不过经过编译器的层层 inline 优化，最终类型都会被推导出来也就不再需要 Existential Container 这一套了。

小礼物走一走，来简书关注我

赞赏支持

开发 (nb/159322) 举报文章 © 著作权归作者所有



星光社的戴铭 (/u/9a4903d7e3d1) 

写了 109729 字，被 2629 人关注，获得了 1408 个喜欢 (/u/9a4903d7e3d1)

+ 关注


微博：@戴铭，github帐号ming1016，喜欢画画,instagram帐号ming1016，qq:36270359

喜欢 (/sign\_in?utm\_source=desktop&utm\_medium=not-signed-in-like-button) | 14




更多分享

(http://cwb.assets.jianshu.io/notes/images/2311664




下载简书 App ▶


随时随地发现和创作内容



(/apps/download?utm\_source=nbc)

被以下专题收入，发现更多相似内容

- 

首页投稿 (/c/bDHhpK?utm\_source=desktop&utm\_medium=notes-included-collection)
- 

程序员 (/c/NEt52a?utm\_source=desktop&utm\_medium=notes-included-collection)

↑

🔗

## Java面试宝典Beta5.0 (/p/fb7d48083e5e?utm\_campaign=maleskine&utm...

pdf下载地址: Java面试宝典 第一章内容介绍 20 第二章JavaSE基础 21 一、Java面向对象 21 1. 面向对象都有哪些特性以及你对这些特性的理解 21 2. 访问权限修饰符public、private、protected, 以及不写 (默认) ...



王震阳 (/u/773a782d9d83?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## 【译】Swift 泛型宣言 (/p/81bcc2d409f5?utm\_campaign=maleskine&utm...

原文: Generics Manifesto -- Douglas Gregor 译者注 在我慢慢地深入使用 Swift 之后, 碰壁了很多次, 很大一部分都是因为 Swift 的泛型系统导致的, 很多抽象都没办法很好地表达出来, 所以就翻译了这篇文章来...



kemchenj (/u/293e6ce1e789?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## swift3.0 基础知识点 (/p/ffece78e61c0?utm\_campaign=maleskine&utm...

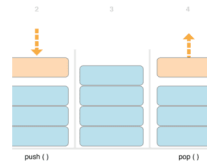
```
importUIKit classViewController:UITabBarController{ enumDayssofaWeek {//星期 caseSunday caseMonday caseTUESDAY caseWEDNESDAY caseThursday c...
```



明哥\_Young (/u/231825b77808?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/2f52b1c6778b?)



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## Swift 4.0 编程语言 (八) (/p/2f52b1c6778b?utm\_campaign=maleskine...

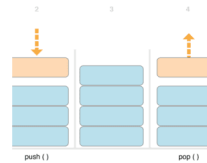
136. 泛型 泛型代码让你可以写出灵活,可重用的函数和类型,它们可以使用任何类型,受你定义的需求的约束。你可以写出代码,避免重复而且可以用一个清晰抽象的方式来表达它的意图。泛型是Swift中最有力的特征...



无泮 (/u/6f8025ebfc51?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/f22b4c379cfc?)



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## swift泛型讲解 (/p/f22b4c379cfc?utm\_campaign=maleskine&utm\_conte...

泛型代码可以确保你写出灵活的, 可重用的函数和定义出任何你所确定好的需求的类型。你的可以写出避免重复的代码, 并且用一种清晰的, 抽象的方式表达出来。泛型是Swift需要强大特征中的其中一个, 许多S...



iOS\_Developer (/u/51a912d7cb40?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## 千万不要为不值得的人付出情绪, 因为一开始你就输了 (/p/d53803c1f7ed?...

深圳入秋了, 早上起来感到一丝凉意, 此时是北京时间7点, 我穿着一条裤衩, 光着膀子坐在客厅的沙发上, 喷嚏连连, 问我为什么起这么早, 肯定是玩王者荣耀啊。我早上有一个习惯, 不想起床, 这应该不是我一...



菜鸟0神 (/u/5ca294e90955?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## 这有什么用 (/p/536491799bc8?utm\_campaign=maleskine&utm\_content...

这有什么用? 这有什么用? 记得小时候, 每次我一要干嘛, 父母就会问, 这有什么用。想买什么东西, 他们会说, 这有什么用。想学什么, 他们会说, 这有什么用。想干什么, 他们会说, 这有什么用。天啊, 我怎...




舟一 (/u/9f03f34da160?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)




如果我可以生活在我的世界以外的世界 (/p/5f598d904c72?utm\_campaign=...

她死了，是替我死的，可..... ——题记 我是一个儿皇帝。在很小的时候，我就“享受”到任何孩子都无法比拟的所谓的幸福。十岁那年，我又被带到那个很漂亮的大房子，又坐在那个极不舒服的椅子上，母后不只一...

 乌小四 (/u/7832c55c7d05?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

《钢铁是怎样炼成的》 (/p/3b70af5bbdbc?utm\_campaign=maleskine&ut...

怀着无比激动的心情，我拜读了《钢铁是怎样炼成的》一书。保尔·柯察金小的时候生活在社会最低层，饱受折磨和侮辱。后来在别人的影响下，他逐步走上革命道路。其后经历了一系列的人生挑战，但无论是战...

 外媒聂宁峰 (/u/8212b905f92a?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)


(/p/46d819541f71?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

要么折腾，要么平凡 (/p/46d819541f71?utm\_campaign=maleskine&utm...

——妈妈说理财·出品 -1- 一：Maggie自豪的拿起两粒骰子，在手心摇晃着，期待着可以摇出一个大数，跑起来更快。骰子落下。不负期待，俩粒儿点数加起来，非常高，一个4，一个5，加起来9步。Maggie兴奋的...

 ——妈妈说理财 (/u/687fedfad021?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)