

Overview

自 WWDC 2015 推出和开源 Swift 2.0 后，大家对 Swift 的热情又一次高涨起来，在羡慕创业公司的朋友们大谈 Swift 新特性的同时，也有很多像我一样工作上依然需要坚守着 Objective-C 语言的开发者们。今年的 WWDC 中介绍了几个 Objective-C 语言的新特性，还是在“与 Swift 协同工作”这种 Topic 里讲的，越发凸显这门语言的边缘化了，不过有新特性还是极好的，接下来，本文将介绍下面三个主要的新特性：

- Nullability
- Lightweight Generics *
- __kindof

Nullability

然而 Nullability 并不算新特性了，从上一个版本的 llvm 6.1 (Xcode 6.3) 就已经支持。这个简版的 Optional，没有 Swift 中 ? 和 ! 语法糖的支持，在 Objective-C 中就显得非常啰嗦了：

```
1 | @property (nonatomic, strong, nonnull) Sark *sark;
2 | @property (nonatomic, copy, readonly, nullable) NSArray *friends;
3 | + (nullable NSString *)friendWithName:(nonnull NSString *)name;
```

假如用来修饰一个变量，前面还要加双下划线，放到 block 里面就更加诡异，比如一个 Request 的 start 方法可以写成：

```
1 | - (void)startWithCompletionBlock:(nullable void (^)(NSError * __nullable error))block;
```

除了这俩外，还有个 null_resettable 来表示 setter nullable，但是 getter nonnull，绕死了，最直观例子就是 UIViewController 中的 view 属性：

```
1 | @property (null_resettable, nonatomic, strong) UIView *view;
```

它可以被设成 nil，但是调用 getter 时会触发 -loadView 从而创建并返回一个非 nil 的 view。

从 iOS9 SDK 中可以发现，头文件中所有 API 都已经增加了 Nullability 相关修饰符，想了解这个特性的用法，翻几个系统头文件就差不多了。接口中 nullable 的是少数，所以为了防止写一大堆 nonnull，Foundation 还提供了一对儿宏，包在里面的对象默认加 nonnull 修饰符，只需要把 nullable 的指出来就行，黑话叫 Audited Regions：

```
1 | NS_ASSUME_NONNULL_BEGIN
2 | @interface Sark : NSObject
3 | @property (nonatomic, copy, nullable) NSString *workingCompany;
4 | @property (nonatomic, copy) NSArray *friends;
5 | - (nullable NSString *)gayFriend;
6 | @end
7 | NS_ASSUME_NONNULL_END
```

Nullability 在编译器层面提供了空值的类型检查，在类型不符时给出 warning，方便开发者第一时间发现潜在问题。不过我想更大的意义在于能够更加清楚的描述接口，是主调者和被调者间的一个协议，比多少句文档描述都来得清晰，打个比方：

```
1 | + (nullableinstancetype)URLWithString:(NSString *)URLString;
```

NSURL 的这个 API 前面加了 nullable 后，更加显式的指出了这个接口可能因为 URLString 的格式错误而创建失败，使用时自然而然的就考虑到了判空处理。

不仅是属性和方法中的对象，对于局部的对象、甚至 c 指针都可以用带双下划线的修饰符，可以理解成能用 const 关键字的地方都能用 Nullability。

所以 Nullability 总的来说就是，写着丑B，用着舒服 --

Lightweight Generics

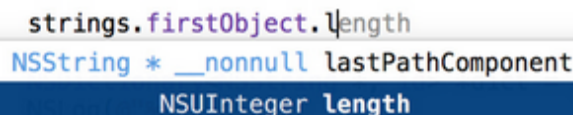
Lightweight Generics 轻量级泛型，轻量是因为这是个纯编译器的语法支持（llvm 7.0），和 Nullability 一样，没有借助任何 objc runtime 的升级，也就是说，这个新语法在 Xcode 7 上可以使用且完全向下兼容（更低的 iOS 版本）

带泛型的容器

这无疑是本次最重大的改进，有了泛型后终于可以指定容器类中对象的类型了：

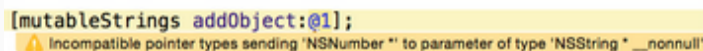
```
1 | NSArray *strings = @[@"sun", @"yuan"];
2 | NSDictionary *mapping = @{@"a": @1, @"b": @2};
```

返回值的 id 被替换成具体的类型后，令人感动的代码提示也出来了：



The image shows a code completion menu for the expression `strings.firstObject`. The menu lists three options: `length`, `NSString * __nonnull lastPathComponent`, and `NSUInteger length`. The `length` option is currently selected, and the `NSString * __nonnull lastPathComponent` option is highlighted in blue.

假如向泛型容器中加入错误的对象，编译器会不开心的：



The image shows a compiler error message in Xcode. The error message is: `Incompatible pointer types sending 'NSNumber *' to parameter of type 'NSString * __nonnull'`. The error is triggered by the code `[mutableStrings addObject:@1];`.

系统中常用的一系列容器类型都增加了泛型支持，甚至连 NSEnumerator 都支持了，这是非常 Nice 的改进。和 Nullability 一样，我认为最大的意义还是丰富了接口描述信息，对比下面两种写法：

```
1 | @property (readonly) NSArray *imageURLs;
2 | @property (readonly) NSArray *imageURLs;
```

不用多想就清楚下面的数组中存的是什么，避免了 NSString 和 NSURL 的混乱。

自定义泛型类

比起使用系统的泛型容器，更好玩的是自定义一个泛型类，目前这里还没什么文档，但拦不住我们写测试代码，假设我们要自定义一个 Stack 容器类：

```

1 | @interface Stack : NSObject
2 | - (void)pushObject:(ObjectType)object;
3 | - (ObjectType)popObject;
4 | @property (nonatomic, readonly) NSArray *allObjects;
5 | @end

```

这个 `ObjectType` 是传入类型的 placeholder，它只能在 `@interface` 上定义（类声明、类扩展、Category），如果你喜欢用 `T` 表示也 ok，这个类型在 `@interface` 和 `@end` 区间的作用域有效，可以把它作为入参、出参、甚至内部 `NSArray` 属性的泛型类型，应该说一切都是符合预期的。我们还可以给 `ObjectType` 增加类型限制，比如：

```

1 | // 只接受 NSNumber * 的泛型
2 | @interface Stack : NSObject
3 | // 只接受满足 NSCopying 协议的泛型
4 | @interface Stack<objectType: id> : NSObject</objecttype: id>

```

若什么都不加，表示接受任意类型 (`id`)；当类型不满足时编译器将产生 `error`。

实例化一个 `Stack`，一切工作正常：

```

Stack<NSString *> *stringStack = [Stack new];
[stringStack pushObject:(NSString *)];
void pushObject:(NSString *)

```

对于多参数的泛型，用逗号隔开，其他都一样，可以参考 `NSDictionary` 的头文件。

协变性和逆变性

当类支持泛型后，它们的 `Type` 发生了变化，比如下面三个对象看上去都是 `Stack`，但实际上属于三个 `Type`：

```

1 | Stack *stack; // Stack *
2 | Stack *stringStack; // StackStack *mutableStringStack; // Stack

```

当其中两种类型做类型转化时，编译器需要知道哪些转化是允许的，哪些是禁止的，比如，默认情况下：

```

Stack *stack;
Stack<NSString *> *stringStack;
Stack<NSMutableString *> *mutableStringStack;

stack = stringStack;
stack = mutableStringStack;
stringStack = stack;
stringStack = mutableStringStack;
mutableStringStack = stack;
mutableStringStack = stringStack;

```

我们可以看到，不指定泛型类型的 `Stack` 可以和任意泛型类型转化，但指定了泛型类型后，两个不同类型间是不可以强转的，假如你希望主动控制转化关系，就需要使用泛型的协变性和逆变性修饰符了：

`__covariant` - 协变性，子类型可以强转到父类型（里氏替换原则）

`__contravariant` - 逆变性，父类型可以强转到子类型（WTF?）

协变：

```
1 | @interface Stack : NSObject
```

效果：

```
Stack<NSString*> *stringStack;
Stack<NSMutableString*> *mutableStringStack;

stringStack = mutableStringStack;
mutableStringStack = stringStack;
// incompatible pointer types assigning to 'Stack<NSMutableString*>' from 'Stack<NSString*>'
```

逆变：

```
1 | @interface Stack : NSObject
```

效果：

```
Stack<NSString*> *stringStack;
Stack<NSMutableString*> *mutableStringStack;

stringStack = mutableStringStack;
mutableStringStack = stringStack;
// incompatible pointer types assigning to 'Stack<NSString*>' from 'Stack<NSMutableString*>'
```

协变是非常好理解的，像 NSArray 的泛型就用了协变的修饰符，而逆变我还没有想到有什么实际的使用场景。

__kindof

__kindof 这修饰符还是很实用的，解决了一个长期以来的小痛点，拿原来的 UITableView 的这个方法来说：

```
1 | - (id)dequeueReusableCellWithIdentifier:(NSString *)identifier;
```

使用时前面基本会使用 UITableViewCell 子类型的指针来接收返回值，所以这个 API 为了让开发者不必每次都蛋疼的写显式强转，把返回值定义成了 id 类型，而这个 API 实际上的意思是返回一个 UITableViewCell 或 UITableViewCell 子类的实例，于是新的 __kindof 关键字解决了这个问题：

```
1 | - (__kindof UITableViewCell *)dequeueReusableCellWithIdentifier:(NSString *)identifier;
```

既明确表明了返回值，又让使用者不必写强转。再举个带泛型的例子，UIView 的 subviews 属性被修改成了：

```
1 | @property (nonatomic, readonly, copy) NSArray *subviews;
```

这样，写下面的代码时就没有任何警告了：

```
1 | UIButton *button = view.subviews.lastObject;
```

Where to go

有了上面介绍的这些新特性以及如instancetype这样的历史更新，Objective-C 这门古老语言的类型检测和类型推断终于有所长进，现在不论是接口还是代码中的 id 类型都越来越少，更多潜在的类型错误可以被编译器的静态检查发现。

同时，个人感觉新版的 Xcode 对继承链构造器的检测也加强了，**NS_DESIGNATED_INITIALIZER** 这个宏并不是新面孔，可以使用它标志出像 Swift 一样的指定构造器和便捷构造器。

最后，附上一段用上了所有新特性的代码，Swift 是发展趋势，如果你暂时依然要写 Objective-C 代码，把所有新特性都用上，或许能让你到新语言的迁移更无痛一点。

```
NS_ASSUME_NONNULL_BEGIN

@interface IMDashboardModel : NSObject

- (instancetype)initWithTeam:(IMTeam *)team NS_DESIGNATED_INITIALIZER;

@property (nonatomic, readonly, strong) IMTeam *team;
@property (nonatomic, readonly, copy) NSArray<__kindof IMMember *> *members;

- (void)updateAllDashboardDataThen:(nullable void (^)(NSError * __nullable error))then;

@end

NS_ASSUME_NONNULL_END
```