OneV's Den

上善若水，人淡如菊

2016-03-29 • 能工巧匠集

Swift 2 throws 全解析 – 从原理到实践

本文最初于 2015 年 12 月发布在 IBM developerWorks 中国网站发表，其网址是 http://www.ibm.com/developerworks/cn/mobile/mo-cn-swift/index.html。如需转载请保留此行声明。

Swift 2 错误处理简介

throws 关键字和异常处理机制是 Swift 2 中新加入的重要特性。Apple 希望通过在语言层面对异常处理的流程进行规范和统一，来让代码更加安全，同时让开发者可以更加及时可靠地处理这些错误。Swift 2 中所有的同步 Cocoa API 的 `NSError` 都已经被 throw 关键字取代，举个例子，在文件操作中复制文件的 API 在 Swift 1 中使用的是和 Objective-C 类似的 `NSError` 指针方式：

```
func copyItemAtPath(_ srcPath: String, toPath dstPath: String, error: NSErrorPointer)
```

而在 Swift 2 中，变为了 throws：

```
func copyItemAtPath(_ srcPath: String, toPath dstPath: String) throws
```

使用时，Swift 1.x 中我们需要创建并传入 `NSError` 的指针，在方法调用后检查指针的内容，来判断是否成功：

```
let fileManager = NSFileManager.defaultManager()
var error: NSError?
fileManager.copyItemAtPath(srcPath, toPath: dstPath, error: &error)
if error != nil {
    // 发生了错误
} else {
    // 复制成功
}
```

在实践中，因为这个 API 仅会在极其特定的条件下 (比如磁盘空间不足) 会出错，所以开发者为了方便，有时会直接传入 nil 来忽视掉这个错误：

```
let fileManager = NSFileManager.defaultManager()
// 不关心是否发生错误
fileManager.copyItemAtPath(srcPath, toPath: dstPath, error: nil)
```

这种做法无形中降低了应用的可靠性以及从错误中恢复的能力。为了解决这个问题，Swift 2 中在编译器层级就对 throws 进行了限定。被标记为 throws 的 API，我们需要完整的 `try catch` 来捕获可能的异常，否则无法编译通过：

```
let fileManager = NSFileManager.defaultManager()
do {
    try fileManager.copyItemAtPath(srcPath, toPath: dstPath)
} catch let error as NSError {
    // 发生了错误
    print(error.localizedDescription)
}
```

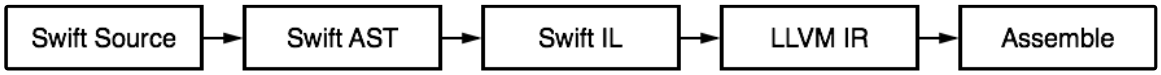
的处理出口，有益于提高应用质量。

throws 技术内幕

throws 关键字究竟做了些什么，我们可以用稍微底层一点的手法来进行一些探索。

Swift 编译器，SIL 及汇编

所有的 Swift 源文件都要经过 Swift 编译器编译后才能执行。Swift 编译过程遵循非常经典的 LLVM 编译架构：编译器前端首先对 Swift 源码进行词法分析和语法分析，生成 Swift 抽象语法树 (AST)，然后从 AST 生成 Swift 中间语言 (Swift Intermediate Language，SIL)，接下来 SIL 被翻译成通用的 LLVM 中间表述 (LLVM Intermediate Representation, LLVM IR)，最后通过编译器后端的优化，得到汇编语言。整个过程可以用下面的框图来表示：



Swift 编译器提供了非常灵活的命令行工具：swiftc，这个命令行工具可以运行在不同模式下，我们通过控制命令行参数能获取到 Swift 源码编译到各个阶段的结果。使用 `swiftc --help` 我们能得知各个模式的使用方法，这篇文章会用到下面几个模式，它们分别将 Swift 源代码编译为 SIL，LLVM IR 和汇编语言。

```
> swiftc --help
...
MODES:
  -emit-sil        Emit canonical SIL file(s)
  -emit-ir         Emit LLVM IR file(s)
  -emit-assembly   Emit assembly file(s) (-S)
 ...
```

在 Swift 开源之前，将源码编译到各个阶段是探索 Swift 原理和实现方式的重要方式。即使是在 Swift 开源后的今天，在面对一段代码时，想要知道编译结果和底层的行为，最快的方式还是查看编译后的语句。我们接下来将会分析一段简单的 throw 代码，来看看 Swift 的异常机制到底是如何运作的。

throw，try，catch 深层解析

为了保持问题的简单，我们定义一个最简单的 `ErrorType` 并用一个方法来将其抛出，源代码如下：

```
// throw.swift

enum MyError: ErrorType {
    case SampleError
}

func throwMe(shouldThrow: Bool) throws -> Bool {
    if shouldThrow {
        throw MyError.SampleError
    }
    return true
}
```

使用 swiftc 将其编译为 SIL：

```
swiftc -emit-sil -O -o ./throw.sil ./throw.swift
```

在输出文件中，可以找到 `throwMe` 的对应 Swift 中间语言表述：

```
// throw.throwMe (Swift.Bool) throws -> Swift.Bool
sil hidden @_TF5throw7throwMeFzSbSb :
                                        $@convention(thin) (Bool) -> (Bool, @error ErrorType) {
bb0(%0 : $Bool):
  debug_value %0 : $Bool  // let shouldThrow      // id: %1
  %2 = struct_extract %0 : $Bool, #Bool.value     // user: %3
  cond_br %2, bb1, bb2                            // id: %3

bb1:                                              // Preds: bb0
```

```
bb2:                                              // Preds: bb0
  ...
  return %9 : $Bool                               // id: %10
}
```

`_TF5throw7throwMeFzSbSb` 是 `throwMe` 方法 Mangling 以后的名字。在去掉一些噪音后，我们可以将这个方法的签名等效看做：

```
throwMe(shouldThrow: Bool) -> (Bool, ErrorType)
```

它其实是返回的是一个 `(Bool, ErrorType)` 的多元组。和一般的多元组不同的是，第二个元素 `ErrorType` 被一个 `@error` 修饰了。这个修饰让多元组具有了“排他性”，也就是只要多元组的第一个元素被返回即可：在条件分支 `bb2` (也即没有抛出异常的正常分支) 中，仅只有 Bool 值被返回了。而对于发生错误需要抛出的处理，SIL 层面还并没有具体实现，只是生成了对应的错误枚举对象，然后对其调用了 throw 命令。

这就是说，我们想要探索 throw 的话，还需要更深入一层。用 swiftc 将源代码编译为 LLVM IR：

```
swiftc -emit-ir -O -o ./throw.ir ./throw.swift
```

结果中 `throwMe` 的关键部分为：

```
define hidden i1 @_TF5throw7throwMeFzSbSb(i1,
        %swift.refcounted* nocapture readnone, %swift.error** nocapture) #0 {

}
```

这是我们非常熟悉的形式，参数中的 `swift.error**` 和 Swift 1 以及 Objective-C 中使用 `NSError` 指针来获取和存储错误的做法是一致的。在示例的这种情况下，LLVM 后端针对 swift.error 进行了额外处理，最终得到的汇编码的伪码是这样的 (在未启用 -O 优化的条件下)：

```
int __TF5throw7throwMeFzSbSb(int arg0) {
    rax = arg0;
    var_8 = rdx;
    if ((rax & 0x1) == 0x0) {
            rax = 0x1;
    }
    else {
            rax = swift_allocError(0x1000011c8, __TWPO5throw7MyErrorSs9ErrorTypeS_);
            var_18 = rax;
            swift_willThrow(rax);
            rax = var_8;
            *rax = var_18;
    }
    return rax;
}
```

函数最终的返回是一个 int，它有可能是一个实际的整数值，也有可能是一个指向错误地址的指针。这和 Swift 1 中传入 `NSErrorPointer` 来存储错误指针地址有明显不同：首先直接使用返回值我们就可以判断调用是否出现错误，而不必使用额外的空间进行存储；其次整个过程中没有使用到 `NSError` 或者 Objective-C Runtime 的任何内容，在性能上要优于传统的错误处理方式。

我们在了解了 throw 的底层机理后，对于 `try catch` 代码块的理解自然也就水到渠成了。加入一个 `try catch` 后的 SIL 相关部分是：

```
try_apply %15(%16) : $@convention(thin) (Bool) -> (Bool, @error ErrorType), normal bb1, error bb9 // id: 

bb1(%18 : $Bool):
...
bb9(%80 : $ErrorType):
...
```

ErrorType 和 NSError

throw 语句的作用对象是一个实现了 `ErrorType` 接口的值，本节将探讨 `ErrorType` 背后的内容，以及 `NSError` 与它的关系。在 Swift 公开的标准库中，`ErrorType` 接口并没有公开的方法：

```
public protocol ErrorType {
}
```

这个接口有一个 extension，但是也没有公开的内容：

```
extension ErrorType {
}
```

我们可以通过使用 LLDB 的类型检索来获取关于这个接口的更多信息。在调试器中运行 `type lookup ErrorType`：

```
(lldb) type lookup ErrorType
protocol ErrorType {
  var _domain: Swift.String { get }
  var _code: Swift.Int { get }
}
extension ErrorType {
  var _domain: Swift.String {
    get {}
  }
}
```

可以看到这个接口实际上需要实现两个属性：domain 描述错误的所属域，code 标记具体的错误号，这和传统的 `NSError` 中定义一个错误所需要的内容是一致的。事实上 `NSError` 在 Swift 2 中也实现了 `ErrorType` 接口，它简单地返回错误的域和错误代码信息，这是 Swift 1 到 2 的错误处理相关 API 转换的兼容性的保证。

虽然 Cocoa/CocoaTouch 框架中的 throw API 抛出的都是 `NSError`，但是应用开发者更为常用的表述错误的类型应该是 enum，这也是 Apple 对于 throw 的推荐用法。对于实现了 `ErrorType` 的 enum 类型，其错误代码将根据 enum 中 case 声明的顺序从 0 开始编号，而错误域的名字就是它的类型全名 (Module 名 + 类型名)：

```
MyError.InvalidUser._code: 0
MyError.InvalidUser._domain: ModuleName.MyError

MyError.InvalidPassword._code: 1
MyError.InvalidPassword._domain: ModuleName.MyError
```

这虽然为按照错误号来处理错误提供了可能性，但是我们在实践中应当尽量依赖 enum case 而非错误号来对错误进行辨别，这可以提高稳定性，同时降低维护的压力。除了 enum 以外，struct 和 class 也是可以实现 `ErrorType` 接口，并作为被 throw 的对象的。在使用非 enum 值来表示错误的时候，我们可能需要显式地指定 `_code` 和 `_domain`，以区分不同的错误。

throws 的一些实践

异步操作中的异常处理

带有 throw 的方法现在只能工作在同步 API 中，这受限于异常抛出方法的基本思想。一个可以抛出的方法实际上做的事情是执行一个闭包，接着选择返回一个值或者是抛出一个异常。直接使用一个 throw 方法，我们无法在返回或抛出之前异步地执行操作并根据操作的结果来决定方法行为。要改变这一点，理论上我们可以通过将闭包的执行和对结果的操作进行分离，来达到“异步抛出”的效果。假设有一个同步方法可以抛出异常：

```
func syncFunc<A, R>(arg: A) throws -> R
```

通过为其添加一次调用，可以将闭包执行部分和结果判断及返回部分分离：

这相当于将原来的方法改写为了：

```
func syncFunc<A, R>(arg: A) -> (Void throws -> R)
```

这样，单次对 `syncFunc` 的调用将返回一个 `Void throws -> R` 类型的方法，这使我们有机会执行代码而不是直接返回或抛出。在执行 `syncFunc` 返回后，我们还需要对其结果用 `try` 来进行判断是否抛出异常。利用这个特点，我们就可以将这个同步的抛出方法改写为异步形式：

```
func asyncFunc<A, R>(arg: A, callback: (Void throws -> R) -> Void) {
        // 处理操作
        let result: () throws -> R = {
                // 根据结果抛出异常或者正常返回
        }
        return callback(result)
}

// 调用
asyncFunc(arg: someArg) { (result) -> Void in
    do {
        let r = try result()
        // 正常返回
    } catch _ {
        // 出现异常
    }
}
```

绕了一大个圈子，我们最后发现这么做本质上其实和简单地使用 `Result<T, E>` 来表示异步方法的结果并没有本质区别，反而增加了代码阅读和理解的难度，也破坏了 Swift 异常机制原本的设计意图，其实并不是可取的选项。除开某些非常特殊的用例外，对于异步 API 现在并不适合使用 throw 来进行错误判断。

异常处理的测试

在 XCTest 中暂时还没有直接对 Swift 2 异常处理进行测试的方法，如果想要测试某个调用应当/不应当抛出某个异常的话，我们可以对 XCTest 框架的方法进行一些额外但很简单包装，传入 block 并运行，然后在 try 块或是 catch 块内进行 XCTAssert 的断言检测。在 Apple 开发者论坛有关于这个问题的更详细的讨论，完整的示例代码和使用例子可以在这里找到。

类型安全的异常抛出

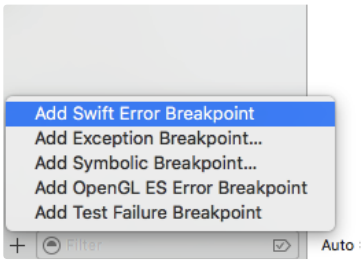
Swift 2 中异常另一个严重的不足是类型不安全。throw 语句可以作用于任意满足 `ErrorType` 的类型，你可以 throw 任意域的错误。而在 catch 块中我们也同样可以匹配任意的错误类型，这一切都没有编译器保证。由于这个原因，现在的异常处理机制并不好用，需要处理异常的开发者往往需要通读文档才能知道可能会有哪些异常，而文档的维护又是额外的工作。缺少强制机制来保证异常抛出和捕获的类型的正确性，这为程序中 bug 的出现埋下了隐患。

事实上从我们之前对 throw 底层实现的分析来看，在语言层面上实现只抛出某一特定类型的错误并不是很困难的事情。但是考虑到与 `NSError` 和传统错误处理 API 兼容问题，Swift 2 中并没有这样实现，也许我们在之后的 Swift 版本中能看到限定类型的异常机制。

异常的调试和断点

Swift 的异常抛出并不是传统意义的 exception，在调试时抛出异常并不会触发 Exception 断点。另外，throw 本身是语言的关键字，而不是一个 symbol，它也不能触发 Symbolic 类型的断点。如果我们希望在所有 throw 语句执行的时候让程序停住的话，需要一些额外的技巧。在之前 throw 的汇编实现中，可以看到所有 throw 语句在返回前都会进行一次 `swift_willThrow` 的调用，这就是一个有效的 Symbolic 语句，我们设置一个 `swift_willThrow` 的 Symbolic 断点，就可以让程序在 throw 的时候停住，并使用调用栈信息来获知程序在哪里抛出了异常。

> 补充，在最新版本的 Xcode 中，Apple 直接为我们在断点类型中加上了 “Swift Error Breakpoint” 的选项，它背后做的就是在 `swift_willThrow` 上添加一个断点。不过因为有了更直接的方法，我们现在不再需要手动去添加



参考资料

- MikeAsh Friday Q&A，Swift 中 Name Mangling 的定义和使用：Friday Q&A: Swift Name Mangling
- Apple 开发者论坛，关于 Swift 中 throw 的测试方法：How to write a unit test which passes if a function throws?

最近的文章
ObjC 中国的工作回顾和之后的计划
小时候因为成绩还算凑合，所以经常会被任命做个班干部什么的。其实这并不是一份很有意思的工作，除了上课要被老师重点“关照”点名起来回答问题以外，最烦人的事情就是开学要写工作计划，期末要写工作总结了。耗时耗力不说，写出来的东西也并不会有什么人看。所以我大抵对写计划和写总结这样的事情是抵触的。顺便还希望这篇总结加计划的东西能有人有兴趣看。时隔十几二十年后，再提笔 (其实是拿键盘) 开始写一份工作回顾和计划的时候，我却是怀着满心欢喜的。从 2014 年 3 月第一个 commit 开始，ObjC 中.....
2016-04-07 • 南箕北斗集 继续阅读

更早的文章
Swift 性能探索和优化分析
本文首发在 CSDN《程序员》杂志，订阅地址 http://dingyue.programmer.com.cn/。Apple 在推出 Swift 时就将其冠以先进，安全和高效的新一代编程语言之名。前两点在 Swift 的语法和语言特性中已经表现得淋漓尽致：像是尾随闭包，枚举关联值，可选值和强制的类型安全等都是 Swift 显而易见的优点。但是对于高效一点，就没有那么明显了。在 2014 年 WWDC 大会上 Apple 宣称 Swift 具有超越 Objective-C 的性能，甚至某些情.....
2016-02-25 • 能工巧匠集 继续阅读