让我们来搞崩 Cocoa 吧 （黑暗代码）

2015-11-13 16:16 编辑：suiling 分类：iOS开发 来源：Mike Ash‘s blog 5 3412

NSView selector Cocoa Hash

招聘信息：PHP开发工程师

文本由CocoaChina译者小袋子（博客）翻译

作者：Mike Ash（Blog GitHub）

原文：Friday Q&A 2014-01-10: Let's Break Cocoa

本文最初发布时间为2014年1月10日

Let's Build系列文章是这个博客中我最喜欢的部分。但是，有时候搞崩程序比编写它们更有趣。现在，我将要开发一些好玩且不同寻常的方式去让 Cocoa 崩溃。

带有 NUL 的字符串

NUL（译者：应该为 '\0'） 字符在 ASCII 和 Unicode 中代表 0，是一个不寻常的麻烦鬼。当在 C 字符串中时，它不作为一个字符，而是一个代表字符串结束的标识符。在其他的上下文环境中，它就会跟其他字符一样了。

当你混合 C 字符串和其它上下文环境，就会产生很有趣的结果。例如：`NSString` 对象，使用 NUL 字符毫无问题：

```
1 NSString *s = @"abc\0def";
```

如果我们仔细的话，我们可以使用 lldb 打印它：

热门资讯


这些 iOS 面试基础题目，你都深入了解吗?
点击量 14992


【译】17个提升iOS开发效率的必用工具
点击量 8887


Objective-C 的现代语法和新特性
点击量 8626


iOS开发——UI组件（个人整理）
点击量 7045


关于 @synchronized，这儿比你想知道的还要
点击量 6008


【译】4个你需要知道的Asset Catalog的秘
点击量 5867

出大事了！A商要联合程序员们打淘宝分京东
点击量 4844


来就来全套！敏捷开发知识体系笔记
点击量 4139


XcodeGhost S：变种带来的又一波影响
点击量 4012


源码推荐(11.02)：自动适配所有子view，仿苏
点击量 3839

综合评论

试用了一下，一点效果都没有，5000个uv居然没一个人购买。
tujiao 评论了 因为A商，互联网的寒冬不再冷...

真正的喜欢一件事不容易，尤其是自己的工作，是我等晚辈学习之楷模
勒布朗_杜兰特 评论了 我已经写了48年代码了，我感觉我还能写下去...

干货不错，先mark下
miss703 评论了 iOS 开发之ReactiveCocoa 下的 ...

```
(lldb) p (void)[[NSFileHandle fileHandleWithStandardOutput] writeData: [s dataUs
abcdef
```

然而，展示这个字符串更为典型的方式是，字符串被当做 C 字符串在某个点结束。由于 '\0' 字符意味着 C 字符串的结尾，因此字符串会在转换时缩短：

```
(lldb) po s
abc
(lldb) p (void)NSLog(s)
LetsBreakCocoa[16689:303] abc
```

原始的字符依然包含预计的字符数量：

```
(lldb) p [s length]
(unsigned long long) $1 = 7
```

对这个字符串进行操作会让你真正感到困惑：

```
(lldb) po [s stringByAppendingPathExtension: @"txt"]
abc
```

如果你不知道字符串的中间包含一个 NUL ，这类问题会让你感到这个世界满满的恶意。

一般来说，你不会遇到 NUL 字符，但是它很有可能通过加载外部资源的数据进来。`-initWithData:encoding:` 会很轻易地读入零比特并且在返回的 `NSString` 中产生 NUL 字符。

循环容器

这里有一个数组：

```
NSMutableArray *a = [NSMutableArray array];
```

这里有一个包含其他数组的数组

```
NSMutableArray *a = [NSMutableArray array];
NSMutableArray *b = [NSMutableArray array];
[a addObject: b];
```

目前为止，看起来还不错。现在我们让一个数组包含自身：

```
NSMutableArray *a = [NSMutableArray array];
[a addObject: a];
```

猜猜会打印出什么？

```
NSLog(@"%@", a);
```

以下就是调用堆栈的信息（译者：bt 命令为打印调用堆栈的信息）：

```
(lldb) bt
* thread #1: tid = 0x43eca, 0x00007fff8952815a CoreFoundation`-[NSArray descrip
                frame #0: 0x00007fff8952815a CoreFoundation`-[NSArray descripti
                frame #1: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #2: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #3: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #4: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #5: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #6: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #7: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #8: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #9: 0x00007fff895282da CoreFoundation`-[NSArray descripti
                frame #10: 0x00007fff895282da CoreFoundation`-[NSArray descript
                frame #11: 0x00007fff895282da CoreFoundation`-[NSArray descript
```

这里还删除了上千个栈帧。描述方法无法处理递归容器，所以它持续尝试去追踪到“树”的结束，并最终发生异常。

我们可以用它跟自身比较对等性：

```
NSLog(@"%d", [a isEqual: a]);
```

这姑且看起来是 YES。让我们创造另一个结构上相同的数组 b 然后用 a 和它比较：

```
NSMutableArray *b = [NSMutableArray array];
[b addObject: b];
NSLog(@"%d", [a isEqual: b]);
```

哎呦：

```
(lldb) bt
* thread #1: tid = 0x4412a, 0x00007fff8946a8d7 CoreFoundation`-[NSArray isEqual'
                frame #0: 0x00007fff8946a8d7 CoreFoundation`-[NSArray isEqualTo.
                frame #1: 0x00007fff8946f6b7 CoreFoundation`-[NSArray isEqual:]
                frame #2: 0x00007fff8946aa07 CoreFoundation`-[NSArray isEqualTo.
                frame #3: 0x00007fff8946f6b7 CoreFoundation`-[NSArray isEqual:]
                frame #4: 0x00007fff8946aa07 CoreFoundation`-[NSArray isEqualTo.
                frame #5: 0x00007fff8946f6b7 CoreFoundation`-[NSArray isEqual:]
                frame #6: 0x00007fff8946aa07 CoreFoundation`-[NSArray isEqualTo.
                frame #7: 0x00007fff8946f6b7 CoreFoundation`-[NSArray isEqual:]
                frame #8: 0x00007fff8946aa07 CoreFoundation`-[NSArray isEqualTo.
                frame #9: 0x00007fff8946f6b7 CoreFoundation`-[NSArray isEqual:]
```

对等性检查同样也不知道如何处理递归容器。

循环视图

你可以用`NSView`实例做同样的实验：

```
NSWindow *win = [self window];
NSView *a = [[NSView alloc] initWithFrame: NSMakeRect(0, 0, 1, 1)];
[a addSubview: a];
[[win contentView] addSubview: a];
```

为了让这个程序崩溃，你只需要尝试去显示视窗。你甚至不需要打印一个描述或者做对等性比较。当试图去显示视窗时，应用就会因尝试追踪底部的视图结构而崩溃。

```
(lldb) bt
* thread #1: tid = 0x458bf, 0x00007fff8c972528 AppKit`NSViewGetVisibleRect + 13
                frame #0: 0x00007fff8c972528 AppKit`NSViewGetVisibleRect + 130
                frame #1: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #2: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #3: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #4: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #5: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #6: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #7: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #8: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #9: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #10: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #11: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #12: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #13: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #14: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #15: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #16: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #17: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #18: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #19: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
                frame #20: 0x00007fff8c9725c6 AppKit`NSViewGetVisibleRect + 288
```

滥用 Hash

让我们创建一个实例一直等于其他类的类 AlwaysEqual，但是 hash 值并不一样：

```
@interface AlwaysEqual : NSObject @end
@implementation AlwaysEqual

- (BOOL)isEqual: (id)object { return YES; }
- (NSUInteger)hash { return random(); }

@end
```

这显然违反了 Cocoa 的要求，当两个对象被认为是相等时，他们的 hash 应该总是返回相等的值。当然，这不是非常

严格的强制要求，所以上述代码依然可以编译和运行。

让我们添加一个实例到 `NSMutableSet` 中：

```
NSMutableSet *set = [NSMutableSet set];
for(;;)
{
    AlwaysEqual *obj = [[AlwaysEqual alloc] init];
    [set addObject: obj];
    NSLog(@"%@", set);
}
```

这产生了一个有趣的日志：

```
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ed70>
)}
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ec40>,
                <AlwaysEqual: 0x61000001ed70>
)}
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ec40>,
                <AlwaysEqual: 0x61000001ed70>
)}
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ec40>,
                <AlwaysEqual: 0x61000001ed70>,
                <AlwaysEqual: 0x61000001f930>
)}
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ec40>,
                <AlwaysEqual: 0x61000001ed70>,
                <AlwaysEqual: 0x61000001f930>
)}
LetsBreakCocoa[17069:303] {(
                <AlwaysEqual: 0x61000001ec40>,
                <AlwaysEqual: 0x61000001ed70>,
                <AlwaysEqual: 0x61000001f930>
)}
```

每次运行都不能保证一样，但是综合看起来就是这样。`addObject:`通常先添加一个新对象，然后在更多的对象添加进来的时候很少成功，最后顶部只有三个对象。现在这个集合包含三个看起来是独一无二的对象，而且看起来应该不会包含更多的对象了。所以，在重写 `isEqual:` 时总是应该重写 `hash`方法。

滥用 **Selector**

Selector 是一个特殊的数据类型，在运行期用于表示方法名。在我们习惯中，它们必须是独一无二的字符串，尽管它们并不是严格地要求是字符串。在现在的 Objective-C 运期间，它们是字符串，并且我们都知道利用 Selector 去搞崩程序是很好玩儿的事。

马上行动，下面就是一个例子：

```
SEL sel = (SEL)"";
[NSObject performSelector: sel];
```

编译和运行后，在运行期产生了很令人费解的错误：

```
LetsBreakCocoa[17192:303] *** NSForwarding: warning: selector (0x100001f86) for
LetsBreakCocoa[17192:303] +[NSObject ]: unrecognized selector sent to class 0x7f
```

通过创建奇怪的 selector，会产生真正奇怪的错误：

```
SEL sel = (SEL)"]: unrecognized selector sent to class 0x7fff75570810";
[NSObject performSelector: sel];
LetsBreakCocoa[17262:303] +[NSObject ]: unrecognized selector sent to class 0x7f
```

你甚至让错误看起来像是停止响应完整信息的 NSObject：

```
SEL sel = (SEL)"alloc";
[NSObject performSelector: sel];
LetsBreakCocoa[46958:303] *** NSForwarding: warning: selector (0x100001f77) for
LetsBreakCocoa[46958:303] +[NSObject alloc]: unrecognized selector sent to class
```

显然，这不是真正的 alloc selector，它是一个碰巧指向一个包含 "alloc" 字符串的伪装 selector。但是，runtime 依然把它打印为 alloc 。

伪造对象

虽然现在越来越复杂，但是 Objective-C 对象依然是分配给所有对象类的大内存中的一小块内存。在这样的思维下，我们就可以创造一个伪造对象：

```
id obj = (__bridge id)(void *)&(Class){ [NSObject class] };
```

这些伪造对象也完全能工作：

```
NSMutableArray *array = [NSMutableArray array];
for(int i = 0; i < 10; i++)
{
    id obj = (__bridge id)(void *)&(Class){ [NSObject class] };
    [array addObject: obj];
}
NSLog(@"%@", array);
```

上述代码不仅可以运行，并且打印日志如下：

```
LetsBreakCocoa[17543:303] (
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>",
    "<NSObject: 0x7fff5fbfe760>"
)
```

可惜的是，看起来所有伪造对象都是以同样的地址结束的。但是还是可以继续工作。好了，当你退出方法并且 autorelease pool 试图去清理时：

```
(lldb) bt
* thread #1: tid = 0x46790, 0x00007fff8b3d55c9 libobjc.A.dylib`realizeClass(obj
    frame #0: 0x00007fff8b3d55c9 libobjc.A.dylib`realizeClass(objc_class*) + 15
    frame #1: 0x00007fff8b3d820c libobjc.A.dylib`lookUpImpOrForward + 98
    frame #2: 0x00007fff8b3cb169 libobjc.A.dylib`objc_msgSend + 233
    frame #3: 0x00007fff8940186f CoreFoundation`CFRelease + 591
    frame #4: 0x00007fff89414ad9 CoreFoundation`-[__NSArrayM dealloc] + 185
    frame #5: 0x00007fff8b3cd65a libobjc.A.dylib`(anonymous namespace)::Autorel
    frame #6: 0x00007fff89420d72 CoreFoundation`_CFAutoreleasePoolPop + 50
    frame #7: 0x00007fff8551ada7 Foundation`-[NSAutoreleasePool drain] + 147
```

因为这些伪造对象没有合适分配内存，所以一旦autorelease pool 试图在方法返回时去操作它们，就会出现严重的错误，并且内存会被重写。

KVC

下面是一个类数组：

```
NSArray *classes = @[
    [NSObject class],
    [NSString class],
    [NSView class]
];
NSLog(@"%@", classes);
LetsBreakCocoa[17726:303] (
    NSObject,
```

```
    NSString,
    NSView
)
```

下面一个这些类实例的数组：

```
NSArray *instances = [classes valueForKeyPath: @"alloc.init.autorelease"];
NSLog(@"%@", instances);
LetsBreakCocoa[17726:303] (
    "<NSObject: 0x61000000a600>",
    "",
    "<NSView: 0x610000136bc0>"
)
```

键值编码并不意味着要这样使用，但是看起来也可以正常运行。

调用者检查

编译器的 `builtin __builtin_return_address` 方法可以返回调用你的代码的地址：

```
void *addr = __builtin_return_address(0);
```

因此，我们可以获取调用者的信息，包括它的名字：

```
Dl_info info;
dladdr(addr, &info);
NSString *callerName = [NSString stringWithUTF8String: info.dli_sname];
```

通过这个，我们可以做一些穷凶极恶的事（译者：并不认为是穷凶极恶的事，反而可作为调用动态方法的一种可选方法，虽然并不可靠），比如说完全可以根据不同的调用者调用合适的方法：

```
@interface CallerInspection : NSObject @end
@implementation CallerInspection

- (void)method
{
    void *addr = __builtin_return_address(0);
    Dl_info info;
    dladdr(addr, &info);
    NSString *callerName = [NSString stringWithUTF8String: info.dli_sname];
    if([callerName isEqualToString: @"__CFNOTIFICATIONCENTER_IS_CALLING_OUT_TO_
        NSLog(@"Do some notification stuff");
    else
        NSLog(@"Do some regular stuff");
}

@end
```

这里是一些测试的代码：

```
id obj = [[CallerInspection alloc] init];
[[NSNotificationCenter defaultCenter] addObserver: obj selector: @selector(metho
[[NSNotificationCenter defaultCenter] postNotificationName: @"notification" obje
[obj method];

LetsBreakCocoa[47427:303] Do some notification stuff
LetsBreakCocoa[47427:303] Do some regular stuff
```

当然，这种方式不是很可靠，因为 `__CFNOTIFICATIONCENTER_IS_CALLING_OUT_TO_AN_OBSERVER__`是 Apple 的内部符号，并且很有可能在未来修改。

Dealloc Swizzle

让我们使用 swizzle （方法调配技术）去调配`-[NSObject dealloc]`到一个不做任何事情的方法。在 ARC 下获得 @selector(dealloc) 有点棘手，因为我们不能直接读取它：

```
Method m = class_getInstanceMethod([NSObject class], sel_getUid("dealloc"));
method_setImplementation(m, imp_implementationWithBlock(^{}));
```

现在我们来欣赏这个例子所产生的混乱（简直就是代码界的黑暗料理）：

```
for(;;)
    @autoreleasepool {
        [[NSObject alloc] init];
    }
```

调配 dealloc 方法导致这个代码完美且合理地疯狂泄露，因为对象不能被摧毁。

总结

用全新和有趣的方法搞崩 Cocoa 能够提供无尽的娱乐性。这也在真实的代码里体现出来了。想起我第一次遇到字符串中嵌入了 NUL ，那是充满痛苦的调试经历。其他只是为了好玩和适当的教学目的。

就是这些了！如果你有任何想要讨论的问题，可以给我发送邮件（mike@mikeash.com）。

（译者注：作者此前已经将网站上Friday Q&A系列文章整理成了一本书，开发者可在iBooks和Kindle上查看，另外还有PDF和ePub格式供下载。点击此处查看详细信息。）

上一篇：源码推荐(11.13)：AFN封装实时更新网络状态，快速集成展示新特性页面
下一篇：iOS 开发之 ReactiveCocoa 下的 MVVM（干货分享）

相关资讯

iOS 开发之 ReactiveCocoa 下的 MVVM（干货分享）
利用Cocoa Layout Instrument检视自动布局
细说ReactiveCocoa的冷信号与热信号（二）：为什么要
细说ReactiveCocoa的冷信号与热信号（一）
MVVM without ReactiveCocoa
【投稿】Cocoapods 应用第二部分-私有库相关
【投稿】CocoaPods的一些略为高级一丁点的使用
[iOS翻译] Cocoa编码规范
Cocoapods 应用第一部分 - Xcode 创建 .framework 相关
关于 Cocoa Auto Layout，你需要知道10件事

我来说两句

您还没有登录！请 登录 或 注册

所有评论（5）

Jacin 2015-11-14 02:34:55

好会玩儿？竟然看到这么灵活的运用！

0 0 回复

zhangyuchang1 2015-11-14 02:16:11

obj－c要被玩坏啦

0 0 回复

wzh945290270 2015-11-13 13:27:17

感觉挺不错的

0 0 回复

自来也大人 2015-11-13 11:19:52

mark，了解语法，才能做得更好，运用的更棒

1 2 回复

superlee 2015-11-13 08:33:49

蛇精病

2 0 回复