iOS (/v2/category/54/ios)

/ 内涵线上内存Zombie工具实践 (/v2/topic/202.rss?uid=472&token=9938048b-492c...

L (/v2/user/likeliang) **likeliang (/v2/user/likeliang)** 18天之前 (/v2/post/208)

作者：互娱研发–内涵研发–李柯良 审稿：刘耀东

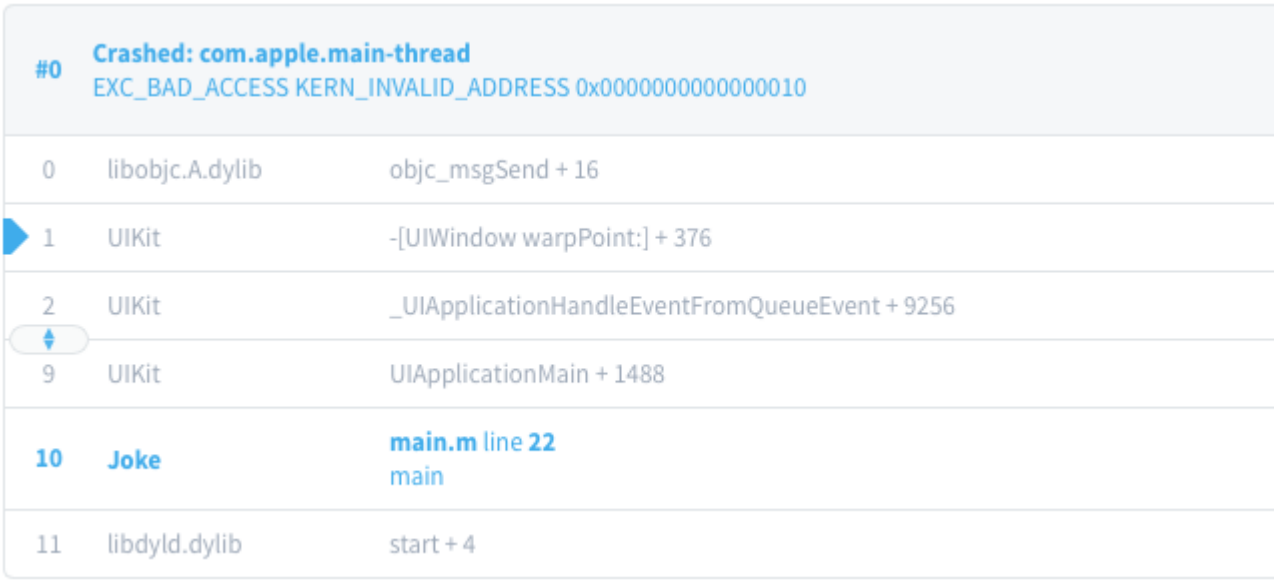
前言

移动软件质量的直接指标是crash率，而多数的crash是因为程序员在编写代码时执行了错误的计算或未能正确的使用系统资源照成。其中野指针异常就属未能正确使用系统资源。

让我们来简述下野指针异常的特点，野指针是指向已经释放对象或访问受限的内存区域的指针，因为OC采用的手动内存管理机制，所以野指针异常一直是crash率最多的问题之一，而Android工程师并未受其困扰（Java内存管理采用GC机制）。野指针异常通常带有随机性，crash堆栈信息缺失，令人无从下手。

野指针的现状

内涵段子iOS的crash排行中，曾Top1和Top3的crash堆栈是指向[UIWindow warpPoint:]的BAD_ACCESS问题，每日约3k次崩溃，崩溃信息如下。



(https://wiki.bytedance.net/download/attachments/118686836/uikit_crash_fabric.png)

从crash机型来看，iOS7/8设备才出现，而且因为系统版本的不同，导致crash堆栈也有差异。而正如之前所说的，从堆栈信息中我们并没有获得更多的有用信息，0x00000010 这个地址是个比较特殊的地址，一般是执行一个被释放的block会崩溃在此位置，不过这里不同系统的野指针地址并不一致，所以并没有帮助确定问题。

在开发环境中，我们能够通过zombies工具帮助定位OC对象的野指针的问题，而线上环境，面对用户的各种使用场景，产生的各种野指针异常我们任然难以定位。那么针对上述野指针问题，我们一方面需要提高测试场景的覆盖，另一方面尝试实现线上zombies工具来帮助定位问题。

dealloc都做了些什么

Zombies工具将已经释放后的对象，转变为僵尸对象，不再被释放，而如果我们想实现线上的zombie工具，则需要了解对象的释放过程并且中断这个过程。

> The Zombies template causes persistent memory growth because it changes your environment so that deallocated objects are never technically deallocated.

我们可以在runtime的源码中找到dealloc的源码，dealloc在一堆判断之后会最终执行到object_dispose方法，如下

```
id object_dispose(id obj)
{
   if (!obj) return nil;

    objc_destructInstance(obj);

    free(obj);

    return nil;
}
```

而objc_destructInstance主要职责是销毁实例，但不释放内存，代码如下

```
/**
 * Destroys an instance of a class without freeing memory and removes any
 * associated references this instance might have had.
 */

void *objc_destructInstance(id obj)
{
    if (obj) {
        // Read all of the flags at once for performance.

        bool cxx = obj->hasCxxDtor();

        bool assoc = obj->hasAssociatedObjects();
```

从主要执行三步清理步骤 object*cxxDestruct 清除成员变量，*object*remove*assocations清除关联对象，clearDeallocating清除弱引用关联。

Cocoa Zombie工具实现

如果在线下环境，我们知道可以通过配置环境变量NSZombieEnabled = YES，来开启Cocoa的僵尸对象内存调试功能。让我们先来了解下cocoa zombie工具的实现机制。

需要在将NSObject的dealloc替换成如下的实现，伪代码如下

```
Class cls = object_getClass(self);

const char *clsName = class_getName(cls);
const char *zombieClsName = "_NSZombie_" + clsName;
Class zombieCls = objc_lookUpClass(zombieClsName);

if (!zombieCls) {
    Class baseZombieCls = objc_lookUpClass("_NSZombie_");
    zombieCls = objc_duplicateClass(baseZombieCls,
                                    zombieClsName, 0);
}
objc_destructInstance(self);
objc_setClass(self, zombieCls);
```

在向zombie对象发送消息时，处理消息的类实际已经是_NSZombie_，而在向该对象发送消息时，该对象直接进行消息转发，通过打印出原始类名信息，并通过abort()中断掉应用，伪代码如下

```
Class cls = object_getClass(self);
const char *clsName = class_getName(cls);
if (string_has_prefix(clsName, "_NSZombie_") {
    const char *originalClsName = substring_from(clsName, 10);
    const char *selectorName = sel_getName(_cmd);

    Log("*** -[%s %s]: message sent to deallocated instance %p",
        originalClsName, selectorName, self);
    abort();
}
```

zombie工具的实现

在了解dealloc过程之后，我们可以对zombie工作原理进行梳理，对比线下zombie工具实现，我们需要解决的问题如下

性能需要不影响用户使用体验，以帧数和CPU使用率作为评估

内存的合理管理，不能像线下zombie调试工具一样，内存只增不减

在捕获到野指针异常的时候，需要对调用进程保存和上报

下面我们逐一解决上述的问题，实现线上zombie工具

zombie对象

首先，zombie对象的选取，对于多数的系统类，在执行过程中创建和销毁频率很高，并且获取它的信息对我们不大，所以仅选取用户创建的类才生成僵尸对象。

获取所有用户创建的类，方式如下，

```
const char ** nh_lookup_runtime_classes(unsigned int * outCount) {
    const char * bundleImageName = class_getImageName([nh_zombie_refrence_object
    return objc_copyClassNamesForImage(bundleImageName, outCount);
}

static NSArray<NSString *> *nh_zombie_self_build_classes()
{
    static NSArray *_classes = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        unsigned int klassesCount = 0;
        const char **  klasses = nh_lookup_runtime_classes(&klassesCount);
        NSMutableArray *array = [[NSMutableArray alloc] initWithCapacity:klassesC
```

dealloc过程拦截

hook deallocation过程需要在MRC环境中进行，在开启zombie线上检测工具时，我们主要进行以下操作，对需要在释放后成为僵尸对象的class对象进行标记，对原始NSObject的dealloc，retain，release函数IMP进行记录， 对NSObject对象dealloc过程进行hook

在dealloc过程中，对于不需要zombie的class直接调用原始的dealloc implement，对于需要的zombie的class，我们进行以下操作，

调用objc_destructInstance进行内存释放

```
(*g_zombie_destruct)(self);
```

用前缀"NHZombie_"加上原始类名，基于NSObject生成新的class，因为我们要完全接管已经释放对象的内存管理，所以需要顺便hook住关于内存管理的retain, release, autorelease方法，并将释放对象的class设置为zombie class，（系统实现是将class设置为__NSZombie__），代码大致如下

```
    static Class nh_zombie_injection_class(const char * klassName, bool fakeNextRe
        if (!klassName) {
            return nil;
        }

        static const char * prefix = "NHZombie_";
        char *className = (char *) malloc(strlen(prefix)+ strlen(klassName) + 1);

        if (className == NULL) {
            return nil;
        }

        strcpy(className, prefix);
```

将zombie对象放入内存管理类进行管理

```
    nh_add_zombie_unretained(self);
```

同时也可以保留特殊的Retain/Release入口，通过_builtinreturn_address()来判断代码执行的入口

```
id nh_zombie_retain(id self, SEL _cmd)
{
    if (__builtin_return_address(1) == nh_zombie_retain_return_address) {
        return g_zombie_original_retain_imp(self, _cmd);
    }

    return self;
}
```

手动管理zombie对象

因为需要手动管理zombie对象，但是在dealloc过程中得到的zombie对象是一个retaincount为0的对象，我们如果需要将对象放入集合中进行管理，则有两种方式，一方式是不使用OC的集合类型进行存储，另一种方式是通一个中间的对象存储和释放zombie对象，我们操作集合对中间对象进行管理，这里选择了第二种方式

```
void nh_add_zombie_unretained(id zombie)
{
    nh_zombie_refrence_object *obj = [[nh_zombie_refrence_object alloc] init];
    obj.zombie = zombie;

    if (zombie) {
        [NHZombieManager addZombie:obj];
    }
    [obj release];

}
```

在nh*zombie*refrence*object对象释放时需要，释放它所持有的zombie对象，**这里有个小坑，按照我们runtime源码看到的因为之前调用了objcdestructInstance，所以这时只用执行free()就可以**，但是实际调试中发直接掉用free(_zombie)，内存并没有被系统回收，猜测是在新版的iOS系统里dealloc实现发生了变化，所以我们在需要释放掉zombie对象时，再次调用一遍原dealloc方法即可

```
-(void)dealloc
{
    g_zombie_original_dealloc_imp(_zombie, _cmd);

    _zombie = nil;

    [super dealloc];
}
```

而nh*zombie*refrence*object对象我们通过HZombieManager进行管理，管理策略是每当zombies内存超过阈值3M时，清理出1M空间。而为了避免线程问题，对于nh*zombie*refrence*object的管理都是在独立的com.bytdance.zombies.io线程进行。

上报BAD_ACCESS信息

最后到了收集zombie对象的调用时候的BAD*ACCESS信息，收集BAD*ACCESS的方式是实现消息转发方法，在发生消息转发时候，进行处理，记录函数的调用栈信息。

```
BOOL nh_zombie_resolve_instance_method(id self, SEL _cmd, SEL selector)
{
    if (selector == NSSelectorFromString(@"_dynamicContextEvaluation:patternStrin
    {
        return NO;
    }
    else {
        class_addMethod([self class], selector, (IMP)nh_zombie_dynamic_safe_metho
        return YES;
    }
}
```

nh*zombie*dynamic*safe*method方法里面主要是对BAD_ACCESS进行保护，同时记录堆栈进行，如果不需要保护，这里直接。

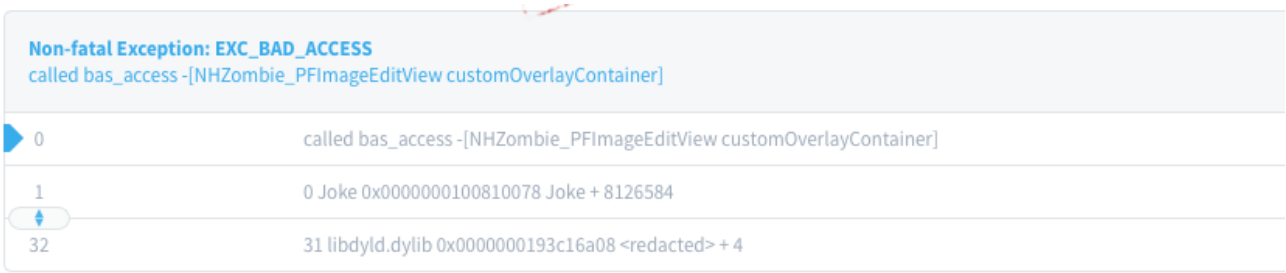
为了得到更有用的堆栈信息， 我们将BAD_ACCESS的捕获时间滞后，对于NSLog的打印和NSObject基类方法调用时候不进行记录， 所以zombie class创建时用NSObject为模版创建，并且在消息转发时忽略NSLog方法dynamicContextEvaluation:patternString的调用，如果希望尽早的暴露问题，创建rootclass并且不进行消息转发即可。

获取到调用栈信息后，我们需要消费这个信息，所以需要将捕获到的野指针信息进行上报，调研后选用fabric的non-fatal crash进行上报，代码如下，

```
#if NH_ZOMBIE
    if ([SSCommonLogic getZombieProtectType] > 0) {
        [NHZombiesProtect enableZombiesProtect:(int)[SSCommonLogic getZombieProte
        [NHZombiesProtect setExceptionHandler:^(NHZombieBacktrace *backtrace) {
            NSString *exceptionReason = [NSString stringWithFormat:@"called bas_a
            if (backtrace.responderChain && ![backtrace.responderChain isEqualToS
                exceptionReason = [exceptionReason stringByAppendingFormat:@", an
            }
            NSMutableArray *stackFrameArray = [NSMutableArray arrayWithArray:[CLS
            [stackFrameArray insertObject:[CLSStackFrame stackFrameWithReason:exc
            [[Crashlytics sharedInstance] recordCustomExceptionName:@"EXC_BAD_ACC

#if TEST_MODE
```

因为时间太久了，symbol文件丢失，文章开头提到的野指针错误大致得到的堆栈如下，



(https://wiki.bytedance.net/download/attachments/118686836/pfimahe_crash.png?version=1&modificationDate=1504519270000&api=v2)

经过排查原因是iOS9以前，UIGestureRecognizers的target是unsafe*unretained类型，而之前代码逻辑是将PFImageEditView的一个子View加入tap手势，同时加入到KeyWindow上，而在ViewController dismiss的时候，子view并没有被移除，所以在响应手势的时候，则会触发野指针异常。(PS: iOS8及之前版本，UIKit的控件属性是unsafe*unretained类型，需要注意)

线上的优化性能

因为工具是在线上运行，所以性能是关注的重点，而对于dealloc这样超高调用频率的场景里面，更是需要注意的重点，下面列举了一些优化的点

使用NSMutableSet而非NSMutableArray，因为hash存储结构，所以在存储、查询等操作上有更高的效率，而项目已开始确实是用NSMutableArray实现LRU的cache模式，但是后来发现效率比Set差太多，所以后来就改用set进行存储了。

只对非系统类的对象使用zombie工具管理，可以减少CPU资源消耗。

dealloc中objc_destructInstance执行在原线程，但是保存zombie对象的操作放在的独立线程进行，保证了主线程释放的对象较多时候，不会卡住主线程。

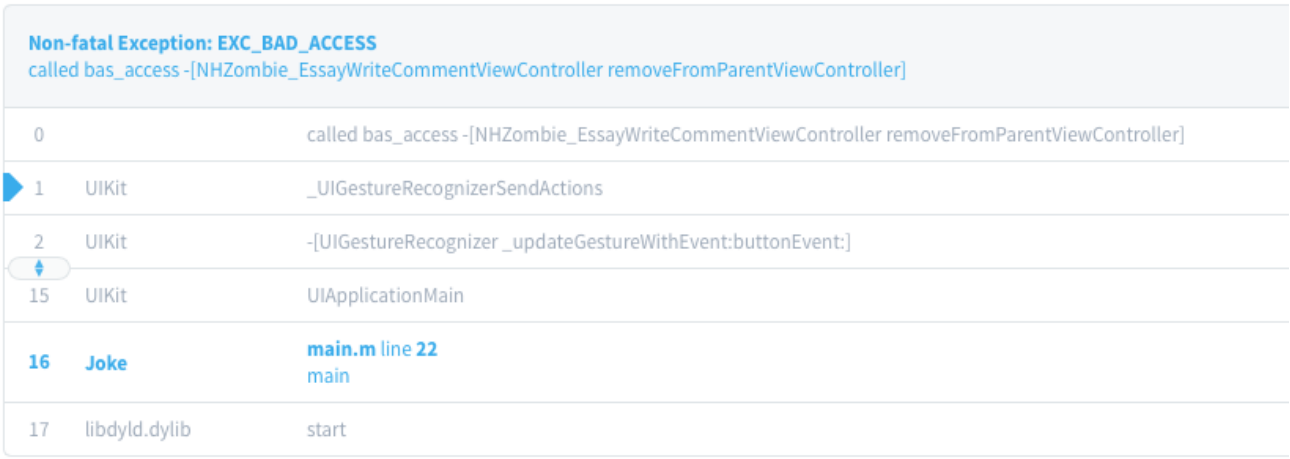
减少dealloc中使用到的临时变量，每一个临时对象生成和销毁，会将原本的一个dealloc的效率降低不止1倍。例如，NSStringFromClass会使生成对象进入autoreleasepool造成额外的开销。

避免NSStringFromClass，NSClassFromString的操作，这里和第三点原因有部分相似，直接通过object*getClassName*，*objc*getClass等操作读取class信息会更快，并且能避免临时变量的生成。

其他方面还可以尝试尽量使用内联force inline减少函数调用的参数压栈开销，避免所有的OC消息发送，直接找到对应的C的操作会更高效。

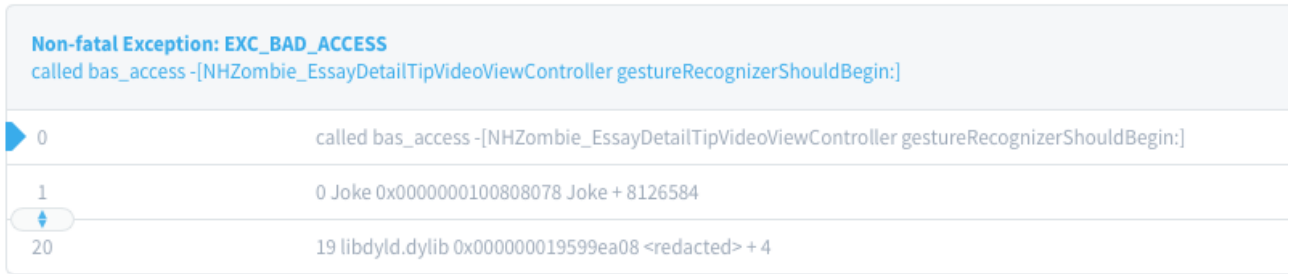
帮助解决了其他一些疑难的野指针问题

评论输入ViewController野指针问题



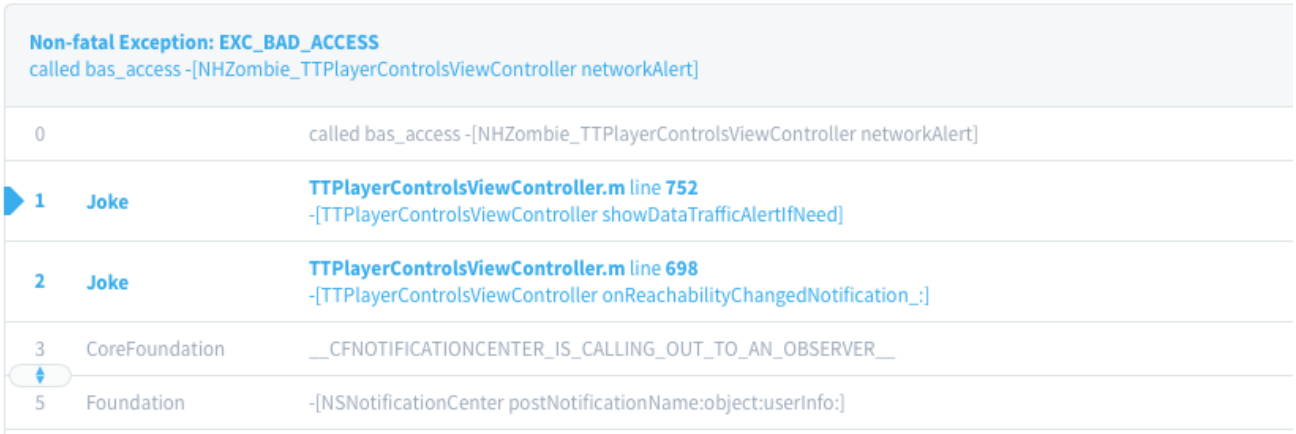
(https://wiki.bytedance.net/download/attachments/118686836/viewcontroller_crash.png?version=2&modificationDate=1504589608000&api=v2)

视频广告野指针问题



(https://wiki.bytedance.net/download/attachments/118686836/tip_crash.png?version=2&modificationDate=1504589601000&api=v2)

alertview的crash



(https://wiki.bytedance.net/download/attachments/118686836/alertview_crash.png?version=2&modificationDate=1504589582000&api=v2)

总结

线上Zombie工具并不是银弹，它能够帮助我们提供更多的信息进行排查，但是真正需要排查和解决问题还需要花费精力去排查和定位。

Zombie工具虽然在性能方面做了优化，实现了不影响用户使用体验，但是因为额外的性能消耗，使用对策略需要进行考虑，目前段子是在发现疑难野指针问题时，对部分用户开启的策略，同时在测试包开启zombie，帮助提高非必现野指针问题的出现概率。

野指针检测和修复是事后的补救方案，之后也将尝试使用infer等静态检测工具提高对代码质量优化，从源头上减少各类carsh的发生。

回复 引用 0

IES半年刊 48 (/v2/tags/ies半年刊)

回复 (/v2/compose?tid=202&title=内涵线上内存Zombie工具实践)