iOS 的崩溃分析

一、获取 Crash、dSYM 文件

获取到的 .ips 改后缀为 .crash 即可

- 真机 Crash 文件目录：var/mobile/Library/Logs/CrashReporter

  通过 iTunes 同步后在 macOS 目录：~/Library/Logs/CrashReporter/MobileDevice/

- 在 iOS 设备上直接查看：设置 -> 隐私 -> 分析 -> 分析数据（不同系统版本不一样）
- macOS Archives 目录（.dSYM 和 .app）：~/Library/Developer/Xcode/Archives
- 通过 iTunes Connect ： Manage Your Applications -> View Details -> Crash Reports

  需要用户在设置->隐私里同意共享诊断数据

  使用 bitcode 编译成中间码上传的，本地不会留下 dSYM，而需要从 iTunes Connect > 或者 Xcode 下载 dSYM（编译成机器码才能生成）
- 通过 Xcode：Xcode -> Window -> Devices and Simulators -> 选中设备 -> View Device Logs
- 通过 Xcode 直接查看：Xcode -> Window -> Organizer -> Crashes

  上传到App Store的时候，同时上传dsym文件，那么从Xcode中的 Crash 会自动符号化。
- 通过 iTools -> 工具箱 -> 崩溃日志 -> 在以下路径查看

```
# Mac
~/Library/Logs/CrashReporter/MobileDevice/<DEVICE_NAME>
# Windows
C://Users/<USER_NAME>/AppDataRoamingApple/ComputerLogsCrashReporterMobileDevice/<DEVICE_NAME>/
```

- 第三方 Crash 统计库：KSCrash、plcrashReporter、CrashKit
- 第三方 Crash 统计服务：Crashlytics、Hockeyapp、友盟、Bugly
- 注意：
    - 最好只集成一个 Crash 统计服务，当各家的服务都以保证自己的Crash统计正确完整为目的时，难免出现时序手脚，强行覆盖等等的恶意竞争。
    - 应用层参与收集 Crash日志的服务方越多，越有可能影响iOS系统自带的 Crash Reporter。

二、Crash 符号化（Symbolicating crash logs）

symbols 和 Symbolicate

symbols 就是函数名或变量名。符号化的过程就是把 crash log 中的内存地址转化为相应的函数调用关系。

一般来说，debug 模式构建的 app 会把符号表存储在编译好的 binary 信息中，而 release 模式构建的app会把符号表存储在 dSYM 文件中以节省体积。

系统库符号化文件

每当 Xcode 连接一台从未在当前电脑调试过的 iOS 版本的设备时，都会花一段时间把手机的系统库符号化文件自动导入到 ~/Library/Developer/Xcode/iOS DeviceSupport，这个过程叫 Processing symbol files。每个系统版本的 symbols 文件约占 2GB，所以这个文件夹会占用不少磁盘空间。但是，最好将这些内容备份到外置硬盘，需要符号化的时候再重新拷贝回来，而不是使用清理工具清理掉。因为，系统符号化文件的获取没有那么容易。

系统库符号文件不是通用的，而是对应crash所在设备的系统版本和CPU型号的。获取系统符号化文件的两大方式就是通过真机，或者通过各版本 Xcode 附带，苹果官方没有提供任何下载方式。

通过 Xcode 符号化

需要3个文件，放在同一目录下

- crash报告（.crash文件）
- Debug Symbol 符号文件（.dsym文件）
- 解压 ipa 包后的 .app 文件

操作过程：Xcode –> Devices and Simulators –> 选中设备 –> View Device Logs

然后把 .crash文件 拖到 Device Logs 或者选择下面的import导入.crash文件。这样你就可以看到crash的详细log了。

通过命令行工具 symbolicatecrash 符号化

将 .app、.dSYM、.crash 文件放到同一个目录下。

```
# 找到 symbolicatecrash 工具并拷贝出来
find /Applications/Xcode.app -name symbolicatecrash -type f
# 会返回几个路径，拷贝其中一个
/Applications/Xcode.app/Contents/SharedFrameworks/DVTFoundation.framework/Versions/A/Resources/symbolicatecrash

# 引入环境变量
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer
# 符号解析
./symbolicatecrash appName.crash .dSYM文件路径 > appName.log
./symbolicatecrash appName.crash appName.app > appName.log
# 将符号化的 crash log 保存在 appName.log 中
./symbolicatecrash appName.crash appName.app > appName.log
```

通过命令行工具 atos 符号化

有多个 .app、.dSYM、.crash 的时候很好用。用于符号化单个地址（可使用脚本批量化）。

每一个可执行程序都有一个build UUID来唯一标识（每次 build 都不同）。Crash日志包含发生crash的这个应用（app）的 build UUID以及crash发生的时候，应用加载的所有库文件的[build UUID]。

```
# 获取 crash 文件的 UUID
grep "appName armv" *crash
# 或者
grep --after-context=2 "Binary Images:" *crash

# 获取 app 的 UUID
```

```
xcrun dwarfdump --uuid appName.app/appName
# 获取 dSYM 的 UUID
xcrun dwarfdump --uuid appName.dSYM

# 对比 app 和 crash 的 UUID 进行匹配

# 用 atos 命令来符号化某个特定模块加载地址（3种方式都可以）
# 0x4000 是模块的加载地址（必须是DWARF文件地址，而不是dSYM地址，dSYM只是一个bundle）
xcrun atos -o appName.app.dSYM/Contents/Resources/DWARF/appName -l 0x4000 -arch armv7
xcrun atos -o appName.app.dSYM/Contents/Resources/DWARF/appName -arch armv7
xcrun atos -o appName.app/appName -arch armv7

# 另外，应用内 获取 UUID 的方法

#import <mach-o/ldsyms.h>
NSString *executableUUID() {
    const uint8_t *command = (const uint8_t *)(&_mh_execute_header + 1);
    for (uint32_t idx = 0; idx < _mh_execute_header.ncmds; ++idx) {
        if (((const struct load_command *)command)->cmd == LC_UUID) {
            command += sizeof(struct load_command);
            return [NSString stringWithFormat:@"%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X",
                    command[0], command[1], command[2], command[3],
                    command[4], command[5],
                    command[6], command[7],
                    command[8], command[9],
                    command[10], command[11], command[12], command[13], command[14], command[15]];
        } else {
            command += ((const struct load_command *)command)->cmdsize;
        }
    }
    return nil;
}

# 通过 iTunes Connect 网站来下载 dSYM 的话，对下载下来的每个 dSYM 文件都执行一次
xcrun dsymutil -symbol-map ~/Library/Developer/Xcode/Archives/[...]/BCSymbolMaps [UUID].dSYM
```

示例：

```
# 有两行未符号化的 crash log
* 3 appName 0x000f462a 0x4000 + 984618
* 4 appName **0x00352aee** 0x4000 + 3468014

# 1. 执行
xcrun atos -o appName.app.dSYM/Contents/Resources/DWARF/appName -l 0x4000 -arch armv7
# 2. 然后输入 0x00352aee
# 3. 符号化结果：
-[UIScrollView(UITouch) touchesEnded:withEvent:] (in appName) (UIScrollView+UITouch.h:26)
```

注意：

- 使用 symbolicatecrash，先拷贝出 symbolicatecrash 文件比较方便。
- 无论是 symbolicatecrash 还是 atos，都只需要 `.crash` 和 `.dSYM` ，或 `.crash` 和 `.app` ，就可以符号化了。
- 系统方法的堆栈符号化需要系统符号化文件，如果本地 macOS 没有该文件，也没有该版本 iOS 设备可拷贝，可通过 Github iOS-System-Symbols(iOS 各版本系统符号库) 下载。

三、Crash 文件结构

1. Process Information（进程信息）

Incident Idnetifier	崩溃报告的唯一标识符，不同的Crash
CrashReporter	Key设备的id（不是uuid）。通常同一个设备上同一版本的app 发生Crash时，该值都是一样的。
Hardware	Model 设备类型
Process	进程名称[进程 id]，进程通常是 app 名字
Path	可执行程序的位置
Identifier	com.companyName.appName
Version	app 版本号

Code Type	CPU 架构
Parent Process	父进程，iOS中App通常都是单进程的，一般父进程都是 launchd

2. Basic Information（基本信息）

Date/Time	Crash发生的时间，可读的字符串
OS	Version 系统版本（build 号）
Report	Version Crash日志的格式，目前基本上都是104，不同的version里面包含的字段可能有不同

3. Exception（异常）

Exception	异常类型
Exception Subtype	异常子类型
Crashed Thread	发生异常的线程号
Exception Information	额外诊断信息

从macOS Sierra, iOS 10, watchOS 3, 和 tvOS 10开始，额外诊断信息，包括：

1. 应用的具体信息：在进程被终止前捕捉到的框架错误信息
2. 内核信息：关于代码签名问题的细节
3. Dyld （动态链接库）错误信息：被动态链接器提交的错误信息

```
# 一段因为找不到链接库而导致进程被终止的Crash Report的摘录
Dyld Error Message:

Dyld Message: Library not loaded: @rpath/MyCustomFramework.framework/MyCustomFramework

 Referenced from: /private/var/containers/Bundle/Application/CD9DB546-A449-41A4-A08B-87E57EE11354/TheElements.app/TheElements
 Reason: no suitable image found.

 # 一段因为没能快速加载初始view controller而导致进程被终止的Crash Report的摘录
Application Specific Information:
com.example.apple-samplecode.TheElements failed to scene-create after 19.81s (launch took 0.19s of total time limit 20.00s)
Elapsed total CPU time (seconds): 7.690 (user 7.690, system 0.000), 19% CPU
Elapsed application CPU time (seconds): 0.697, 2% CPU
```

4. Thread Backtrace（线程回溯）

Crash 发生时的线程的调用栈，没有符号化前是内存地址。

5. Thread State（线程状态）

Crash 发生时的寄存器状态。在你读一个 Crash Report 的时候，了解线程状态并非必须，但是如果你想更好地了解crash的细节，这会起一些帮助，这需要一些处理器硬件只是和汇编知识的储备。

LLDB与汇编调试-提高你的调试效率

6. Binary Images（二进制映像）

Crash 发生时 app 可执行文件、加载的所有系统库和第三方库。

```
# app 可执行文件 Elephant
0x104e80000 -        0x107b2bfff +Elephant arm64  <38c058044caa34818a83d88981986fad> /var/containers/Bundle/Application/5694FC83-0

# WCDB 可执行文件。b512f6d343e73a0db1bcb499d2597c8a 是 WCDB 的 UUID
# 符号化时 dsym 的 UUID 需要与之匹配才能符号化
0x10b724000 -        0x10b86ffff  WCDB arm64  <b512f6d343e73a0db1bcb499d2597c8a> /private/var/containers/Bundle/Application/5694FC
```

四、Crash 的类型

4.1 两类主要的 Crash

引发崩溃的代码本质上就两类，一类是 c/c++ 语言层面的错误，比如野指针，除零，内存访问异常等等（相对复杂）

对于前者，无论是 iOS 还是 Android 系统，其底层都是 unix 或者是类 unix 系统,都可以通过信号机制来获取 signal 或者是 sigaction （但是只能捕捉有限的几种类型），设置一个回调函数。

Watchdog 超时、用户强制退出、低内存终止等，系统抛出Unix信号，没有任何的错误堆栈信息

另一类是未捕获异常 Uncaught Exception（相对简单）

iOS 下面最常见的就是 Objective-C 的NSException（@throw 抛出），可以使用NSUncaughtExceptionHandler catch 住防止崩溃。

如数组越界，给对象发送了无法识别的消息（selector方法没有实现，对象调用方法出错）等，系统抛出一个NSException对象，对象中有出错的堆栈，描述了出错的代码位置、类名和方法名。

4.1.1 Bus Error

- Non-existent address（访问不存在的内存地址）
- Unaligned access（访问未对齐的内存地址）
- Paging errors（分页错误）

在检测顺序上，先检测 SIGBUS，再检测 SIGSEGV。

SIGBUS 地址被放到地址总线之后，检测出地址不对齐，发出异常信号，

SIGSEGV 地址已经放到地址总线上后，在后续流程中检测出内存违法访问，发出异常信号。

- SIGBUS （Bus error）访问非法地址

指针所对应的地址是有效地址，但总线不能正常使用该指针，通常是未对齐的数据访问所致。

一些处理器架构上要求对齐访问数据，比如只能从4字节边界上读取一个4字节的数据类型（对于长度4个字节的对象，其存放地址起码要被4整除才可以）。否则向当前进程分发SIGBUS信号。

- SIGSEGV （Segmentation fault、segfault）合法地址的非法访问

在 ARC 后很少遇到，意味着指针所对应的地址是无效地址，没有物理内存对应该地址。

访问不属于本进程的内存地址

往没有写权限的内存地址写数据

访问已被释放的内存

- SEGV（Segmentation Violation）

代表无效内存地址，比如空指针，未初始化指针，栈溢出等。

4.1.2 其他异常类型

- EXC_CRASH(SIGABRT)

情形：Abnormal Exit 异常退出

未捕获的 Objective-C 异常（NSException），导致系统发送了 Abort 信号退出，导致这类异常崩溃的原因是捕获到 Objective-C/C++ 异常，并且调用了 abort() 函数,会在断言/app内部/操作系统用终止方法抛出。

通常发生在异步执行系统方法的时候，如CoreData/NSUserDefaults等,还有一些其他的系统多线程操作。这并不一定意味着是系统代码存在bug，代码仅仅是成了无效状态,或者异常状态。
通常Foundation库中的容器为了保护状态正常会做一些检测，例如插入nil到数组中等会遇到此类错误。
App Extensions，例如输入法，如果花了太多时间做初始化的话就会以这种异常退出（看门狗机制）。如果扩展程序由于在启动时挂起进而被kill掉，那Report 中的Exception Subtype字段会写 LAUNCH_HANG。因为扩展App没有main函数，所以任何情况下的在static constructors和+load方法里的初始化时间都会体现在你的扩展或者依赖库中。因此你应当尽可能的推迟这些逻辑。

```
example 1: unrecognized selector sent to instance
example 1: attempt to insert nil object from objects
```

对于可能在别处被释放的对象，要自己持有一份（alloc 或 copy）。

- EXC_BREAKPOINT(SIGTRAP)

情形：Trace Trap 追踪捕获

和进程异常退出类似，这种异常是由于在特殊的节点加入debugger调试节点，如果当前没有调试器（debugger）依附，那么则会导致进程被杀掉。可以通过 __builtin_trap() 在代码里手动出发这种异常。

1. 这种 Crash 在 iOS 底层的框架中经常出现，最常见的是GCD。底层库（例如libdispatch）会在遇到fatal错误的时候陷入这个困局。

2. Swift代码会在运行时的时候遇到下述问题时抛出这种异常：

一个non-optional的类型被赋予一个nil值

一个失败的强制转换

遇到这种错误，查下堆栈信息并想清楚是在哪里遇到了未知情况(unexpected condition)。额外信息也可能会在设备的控制台的日志里出现。你应当尽量修改你的代码，去优雅的处理这种运行时错误。例如，处理一个optional的值，通过可选绑定(Optional binding)而不是强制解包来获得其值。

- EXC_BAD_INSTRUCTION(SIGILL)

情形：Illegal Instruction 非法指令

当尝试去执行一个非法或者未定义的指令时会触发该异常。有可能是因为线程在一个配置错误的函数指针的误导下尝试jump到一个无效地址。 在Intel处理器上，ud2操作码会导致一个EXC_BAD_INSTRUCTION异常，但是这个通常用来做调试用途。

在Intel处理器上，Swift会在运行时碰到未知情况时被停止。 详情参考Trace Trap。

- SIGKILL

情形：Killed

进程收到系统指令被干掉。请自行查看Termination Reason（会包含一个命名空间和代码）来定位线程被干掉的原因。

- SIGQUIT

情形：Quit 退出

这个异常是由于其它进程拥有高优先级且可以管理本进程（因此被高优先级进程Kill掉）所导致。SIGQUIT不代表进程发生Crash了，但是它确实反映了某种不合理的行为。

iOS中，如果占用了太长时间，键盘扩展程序会随着宿主app被干掉。因此，这种情况的异常下不太可能会在Crash Report中出现合理可读的异常代码。大概率是因为一些其它代码在启动时占用了太长时间但是在总时间限制前（看门狗的时间限制，见上文中的表格）成功结束了，但是执行逻辑在extension退出的时候被错误的执行了。你应该运行Profile，仔细分析一下extension的各部分消耗时间，把耗时较多的逻辑放到background或者推迟（推迟到extension加载完毕）。

- EXC_ARITHMETIC

除零错误会抛出此类异常

- SIGPIPE 管道破裂

这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。

此外用Socket通信的两个进程，写进程在写Socket的时候，读进程已经终止。

对一个端已经关闭的socket调用两次写入操作，第二次写入将会产生SIGPIPE信号，该信号默认结束进程。

```
// 预防方式，写在 pch 文件
// 仅在 iOS 系统上支持 SO_NOSIGPIPE
#if defined(SO_NOSIGPIPE) && !defined(MSG_NOSIGNAL)
    // We do not want SIGPIPE if writing to socket.
    const int value = 1;
    setsockopt(fd, SOL_SOCKET, SO_NOSIGPIPE, &value, sizeof(int));
#endif
```

另外一些异常类型

为了防止一个应用占用过多的系统资源，设计了 watchdog 的机制，watchdog 会监测应用的性能。如果超出了该场景所规定的运行时间，watchdog 强制终结这个应用的进程。

Exception Code	说明
0xbaaaaaad	并非一个真正的Crash，由用户同时按Home键和音量键触发。

0xbad22222	当VoIP程序在后台太过频繁的激活时，系统可能会终止此类程序。
0x8badf00d（badfood）	launch/resume/suspend/quit/background 响应超过规定时间会被 Watchdog 终止（详见下表）， 并产生一个崩溃日志。在连接Xcode调试时为了便于调试，系统会暂时禁用掉Watchdog，所以此类问题的发现需要使用正常的启动模式。
0xc00010ff	程序执行大量耗费CPU和GPU的运算，导致设备过热，触发系统过热保护被系统终止。这个也许是和发生crash的特定设备有关，或者是和它所在的环境有关。
0xdead10cc（deadlock）	程序退到后台时还占用系统资源（如通讯录）被系统终止。或者程序挂起时拿到了文件锁或者sqlite数据库所长期不释放直到被冻结。如果你的app在挂起时拿到了文件锁或者sqlite数据库锁，它必须请求额外的后台执行时间(request additional background execution time)并在被挂起前完成解锁操作。
0xdeadfa11（deadfall）	程序无响应用户强制退出。当用户长按电源键，直到屏幕出现关机确认画面后再长按Home键，将强制退出应用。（不是双击 home 的强退）Exception Note 会有 SIMULATED 字段。
0x2bad45ec	app因为安全违规操作被iOS系统终止。终止描述会写：“进程被查到在安全模式进行非安全操作”，暗示app尝试在禁止屏幕绘制的时候绘制屏幕，例如当屏幕锁定时。用户可能会忽略这种异常，尤其当屏幕是关闭的或者当这种终止发生时正好锁屏。

说明：通过App Switcher(就是双击home键出现的那个界面)并不会生成Crash Report。一旦app进入挂起状态，被iOS在任何时间终止掉都是合理的，因此这时候不会生成Crash Report。

以下异常代码只针对 watchOS

Exception Code	说明
0xc51bad01	在后台任务占用了过多的cpu时间而导致watch app被干掉。想要解决这个问题，优化后台任务，提高CPU执行效率，或者减少后台的任务运行数量。
0xc51bad02	在后台的规定时间内没有完成指定的后台任务而导致watch app被干掉。想要解决这个问题，需要当app在后台运行时减少app的处理任务。
0xc51bad03	没有在规定时间内完成后台任务，且系统一直非常忙以至于app无法获取足够的CPU时间来完成后台任务。虽然一个app可以通过减少自身在后台的运行任务来避免这个问题，但是0xc51bad03这个错误把矛头指向了过高的系统负载，而非app本身有什么问题。

Watchdog 超时时间

场景	超时时间
launch（启动）	20s
resume（恢复）	10s
suspend（挂起）	10s
quit（退出）	6s
background（后台）	10min

简单说，就是以下代理必须在规定时间内执行完毕，让程序响应起来。

```
- (void)applicationDidFinishLaunching:(UIApplication *)application;
- (void)applicationDidBecomeActive:(UIApplication *)application;
- (void)applicationWillResignActive:(UIApplication *)application;
- (void)applicationDidEnterBackground:(UIApplication *)application;
- (void)applicationWillEnterForeground:(UIApplication *)application;
- (void)applicationWillTerminate:(UIApplication *)application;
```

崩溃（准确的说是程序异常终止）是程序接收到未处理信号的结果。

未处理信号有三个来源：内核、其他进程和应用本身。导致崩溃最常见的两个信号如下：

- EXC_BAD_ACCESS 是一种由内核发出的Mach异常，通常是因为应用试图访问不存在的内存空间导致的。如果未能在Mach内核级别进行处理，它将被转化为SIGBUS或者SIGSEGV BSD信号。
- SIGABRT 是当产生未捕获的NSException或者obj_exception_throw时，应用发给自身的BSD信号。

在 Objective-C 异常中，导致异常抛出最常见的原因是应用向对象发送了未实现的方法选择器（比如拼写错误，对象混淆或者向已经释放的对象发送消息）。

4.2 Low Memory Report 低内存报告

Low Memory Termination

跟一般的Crash结构不太一样，通常有Free pages，Wired Pages，Purgeable pages，largest process 组成，同时会列出当前时刻系统运行所有进程的信息。

Low Memory Report 与其它 Crash Report 不同，它没有堆栈信息，所以不需要符号化。一个低内存 Report的Header会和 Crash Report 的header有些类似。紧接着Header的时各个字段的系统级别的内存统计信息。记录下页大小（Page Size）字段。每一个进程的内存占用大小是根据内存的页的数量来Report的。一个低内存 Report最重要的部分是进程表格。这个表格列出了所有的运行进程，包括系统在生成低内存 Report时的守护进程。如果一个进程被”遗弃”了，会在[原因]一列附上具体的原因。一个进程可能被遗弃的原因有：

- [per-process-limit]

进程占用超过了它的最大内存值。每一个进程在常驻内存上的限制是早已经由系统为每个应用分配好了的。超过这个限制会导致进程被系统干掉。

注意：扩展程序(nimo: Extension app, 例如输入法等)的最大内存值更少。一些技术，例如地图视图和SpriteKit，占用非常多的基础内存，因此不适合用在扩展程序里。

- [vm-pageshortage]/[vm-thrashing]/[vm]

由于系统内存压力被干掉。

- [vnode-limit]

打开太多文件了。

注意：系统会尽量避免在vnodes已经枯竭的时候干掉高频app。因此你的应用如果在后台，即便并没有占用什么vnode，而有可能被杀掉。

- [highwater]

一个系统守护进程超过过了它的内存占用高水位（就是已经很危险了）。

- [jettisoned]

进程因为其它不可描述的原因被杀掉。

当你发现一个低内存crash，与其去担心哪一部分的代码出现问题，还不如仔细审视一下自己的内存使用习惯和针对低内存告警（low-memory warning）的处理措施。Locating Memory Issues in Your App 列出了如何使用Leaks Instrument工具来检查内存泄漏，和如何使用Allocations Instrument的Mark Heap 功能来避免内存浪费。 Memory Usage Performance Guidelines 讨论了如何处理接受到低内存告警的问题，以及如何高效使用内存。当然，也推荐你去看下2010年的WWDC中的 Advanced Memory Analysis with Instruments 那一章节。

重要：Leaks和Allocation工具不能检测所有的内存使用情况。你需要和VM Tracker工具一起运行(包含在Allocation工具里)来查看你的内存运行。默认VM Tracker是不可用的。如果想通过VM Tracker来profile你的应用，点击instrument工具，选中”Automatic Snapshotting”标签或者手动点击”Snapshot Now”按钮。

五、Crash 的捕获

5.0 Last Exception Backtrace

若程序因 NSException 而 Crash，系统日志中的 Last Exception Backtrace 信息是完整准确的，不会受应用层的 Crash 统计服务影响，可作为排查问题的参考线索。如果 Last Exception Backtrace，只包含16进制信息的日志，必须进行符号化来获取有价值的堆栈信息

```
# 未符号化的异常堆栈
Last Exception Backtrace:

(0x18eee41c0 0x18d91c55c 0x18eee3e88 0x18f8ea1a0 0x195013fe4 0x1951acf20 0x18ee03dc4 0x1951ab8f4 0x195458128 0x19545fa20 0x19545fc7c 0x19545ff70 0x194de4594 0x194e94e8c 0x194f47d8c 0x194f39b40 0x194ca92ac 0x18ee917dc 0x18ee8f40c 0x18ee8f89c 0x18edbe048 0x19083f198 0x194d21bd0 0x194d1c908 0x1000ad45c 0x18dda05b8)
```

5.1 处理未捕获异常（uncaught exceptions）

有两种方式可以捕获那些会导致崩溃的未捕获状态。

- 使用 NSUncaughtExceptionHandler 函数来安装未捕获 Objective-C 异常的处理器。
- 使用 signal 函数来安装 BSD 信号处理器。

注意：signal 要在没有附加 debugger 的环境下获取，否则会被 debugger 优先拦截。UncaughtExceptionHandler可以在调试状态下捕获

抓取 NSException

```
// 安装 Objective-C 异常处理器和信号处理的代码如下:
```

```
void InstallUncaughtExceptionHandler() {
    NSSetUncaughtExceptionHandler(&MyUncaughtExceptionHandler);
    signal(SIGABRT, SignalHandler);
    signal(SIGILL, SignalHandler);
    signal(SIGSEGV, SignalHandler);
    signal(SIGFPE, SignalHandler);
    signal(SIGBUS, SignalHandler);
    signal(SIGPIPE, SignalHandler);
}
// 对于异常和信号的响应会在 MyUncaughtExceptionHandler 和 SignalHandler 中实现。在样例程序中，以上二者的处理方式相同。

void MyUncaughtExceptionHandler(NSException *exception) {
    NSString *ret = [NSString stringWithFormat:@"异常名称:\n%@\n\n异常原因:\n%@\n\n出错堆栈内容:\n%@\n",exception.name, exception.rea
son, exception.callStackSymbols];
    // 将捕获到的 exception 细节上传到后台
}
```

抓取 Signal

signal信号是Unix系统中的,是一种异步通知机制.信号传递给进程后,在没有处理函数的情况下,程序可以指定三种行为:

1. 忽略该信号,但是对于信号 `SIGKILL` 和 `SIGSTOP` 不可忽略
2. 使用默认的处理函数 `SIG_DFL (即 signal(sig, SIG_DFL);)` ，大多数信号的默认动作是终止进程
3. 捕获信号,执行用户定义的函数

有两个特殊的常量:

- `SIG_IGN` ，向内核表示忽略此信号.对于不能忽略的两个信号 `SIGKILL` 和 `SIGSTOP` ，调用时会报错
- `SIG_DFL` ，执行该信号的系统默认动作

还有两个常用的函数

- int kill(pid_t pid, int signo);,发送信号到指定的进程
- int raise(int signo);,发送信号给自己.

```
// UNIX系统中常用的信号有以下几种:
SIGABRT--程序中止命令中止信号
SIGBUS--程序内存字节未对齐中止信号
SIGFPE--程序浮点异常信号
SIGILL--程序非法指令信号
SIGSEGV--程序无效内存中止信号
SIGTERM--程序kill中止信号
SIGKILL--程序结束接收中止信号

SIGALRM--程序超时信号
SIGHUP--程序终端中止信号
SIGINT--程序键盘中断信号
SIGSTOP--程序键盘中止信号
SIGPIPE--程序Socket发送失败中止信号

// 抓取的是以下几种
static int Beacon_errorSignals[] = {
    SIGABRT,
    SIGBUS,
    SIGFPE,
    SIGILL,
    SIGSEGV,
    SIGTRAP,
    SIGTERM,
    SIGKILL,
};
for (int i = 0; i < Beacon_errorSignalsNum; i++) {
    signal(Beacon_errorSignals[i], &mysighandler);
}
// 抓取信号的处理函数
void mysighandler(int sig) {
    void* callstack[128];
    NSString* name ;
    int i, frames = backtrace(callstack, 128);
    for (i = 0; i < Beacon_errorSignalsNum; i++) {
        if (Beacon_errorSignals[i] == sig ) {
            name = [Beacon_errorSignalNames[i] copy];
```

```
            break;
        }
    }
    char** strs = backtrace_symbols(callstack, frames);
    NSMutableString* exceptionStr = [[NSMutableString alloc]initWithFormat:@"异常名称:\n%@\n\n出错堆栈内容:\n",name];
    for (i =0; i <frames; i++) {
        [exceptionStr appendFormat:@"%s\n",strs[i]];
    }
    free(strs);
}

// 在应用崩溃后，保持运行状态而不退出，让响应更加友好
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFArrayRef allModes = CFRunLoopCopyAllModes(runLoop);

while (!dismissed) {
    for (NSString *mode in (__bridge NSArray *)allModes) {
        CFRunLoopRunInMode((__bridge CFStringRef)mode, 0.001, false);
    }
}

CFRelease(allModes);
```

这里只处理最常见的信号，但是，你可以为自己的程序添加所需的所有异常信号。

注意，有两种异常是不能捕获的：SIGKILL和SIGSTOP。它们会终止或者暂停应用。（SIGKILL是命令行函数kill -9发出的，SIGSTOP是键入Control-Z发出的）。

如果你发现本应该被捕捉的异常并没有被捕捉到，请确定您没有在building应用或者library时添加了-no_compact_unwind标签。

64位 iOS 用了zero-cost的异常实现机制。在zero-cost系统里，每一个函数都有一个额外的数据，它会描述如果一个异常在跨函数范围内实现，该如何展开相应的堆栈信息。如果一个异常发生在多个堆栈但是没有可展开的数据，那么异常处理函数自然无法跟踪并记录。也许在堆栈很上层的地方有异常处理函数，但是如果那里没有一个片段的可展开信息，没办法从发生异常的地方到那里。指定了-no_compact_unwind标签表明你那些代码没有可展开信息，所以你不能跨越函数抛出异常（也就是说无法通过别的函数捕捉当前函数的异常）。

5.2 Xcode 提供的调试工具

都在 Edit Scheme -> Diagnostics（诊断） 依次可以找到

Runtime Sanitization

- Address Sanitizer（地址消毒剂）

AddressSanitizer的原理是当程序创建变量分配一段内存时，将此内存后面的一段内存也冻结住，标识为中毒内存。当程序访问到中毒内存时（越界访问），就会抛出异常，并打印出相应log信息。调试者可以根据中断位置和的log信息，识别bug。如果变量释放了，变量所占的内存也会标识为中毒内存，这时候访问这段内存同样会抛出异常（访问已经释放的对象）。

- Thread Sanitizer

用于解决多线程问题：如何用Xcode8解决多线程问题

Use of uninitialized mutexes（使用未初始化的互斥器）
Thread leaks (missing pthread_join) 线程泄漏（缺少pthread_join）
Unsafe calls in signal handlers (ex:malloc) 信号处理程序中的不安全调用（例如：malloc）
Unlock from wrong thread 从错误的线程解锁
Data races 数据竞争（只要涉及到多线程编程，遇到的概率非常之高，写多线程代码时最容易遇到的问题，一旦踩坑，现象往往是偶现的，难以调试）
大致原理是记录每个线程访问变量的信息来做分析，值得一提的是，现阶段的Thread Sanitizer最多只同时记录4个线程的访问信息，在复杂的场景下，可能出现偶尔检测不出data race的场景，所以需要长时间经常性的运行来尽可能多的发现data race，这也是为什么苹果建议默认开启Thread Sanitizer，而且 Thread Sanitizer 造成的额外性能损耗非常之小。

Thread Sanitizer 现阶段只能在模拟器环境下执行，真机还不支持，有同学测试发现，只支持64位系统，也就是说iPhone 5及其更早的模拟器也不支持，iPhone 5s 之后才是64位系统。

Memory Management

- Malloc Scribble

申请内存后在申请的内存上填 0xAA，内存释放后在释放的内存上填 0x55；再就是说如果内存未被初始化就被访问，或者释放后被访问，就会引发异常，这样就可以使问题尽快暴漏出来。

Scribble 其实是 malloc 库 libsystem_malloc.dylib 自身提供的调试方案

- Malloc Guard Edges

申请大片内存的时候在前后page上加保护，详见保护模式。

- Guard Malloc

使用 libgmalloc 捕获常见的内存问题，比如越界、释放之后继续使用。

由于 libgmalloc 在真机上不存在，因此这个功能只能在模拟器上使用.

- Zombie Objects（僵尸对象）

Instrument 也有一个 Zombie 工具，使用起来差不多。

Zombie 的原理是用生成僵尸对象来替换 dealloc 的实现，当对象引用计数为 0 的时候，将需要 dealloc 的对象转化为僵尸对象。如果之后再给这个僵尸对象发消息，则抛出异常，并打印出相应的信息，调试者可以很轻松的找到异常发生位置。

如果 objc_msgSend 或者 objc_release出现在crash的线程的附近，则进程有可能尝试去给一个被释放的对象发送消息，那么可使用 Zombie 调试

```
# 控制台会多一些调试信息
message sent to deallocated instance 0x60800000c380
```

Analyze（静态代码分析）

不是那么准确，但是会发现一些问题

可以发现编译中的 warning，内存泄漏隐患，甚至还可以检查出逻辑上的问题；所以在自测阶段一定要解决Analyze发现的问题，可以避免出现严重的bug。

主要分析以下四种问题：

- 逻辑错误：访问空指针或未初始化的变量等；
- 内存管理错误：如内存泄漏等；
- 声明错误：从未使用过的变量；
- 调用错误：未包含使用的库和框架。

```
# 内存泄漏隐患
Potential(潜在) Leak of an object allocated on line ……
# 数据赋值隐患
The left operand of …… is a garbage value;
# 对象引用隐患
Reference-Counted object is used after it is released;
```

Profile（就是运行 Instrument）

真正运行程序，对程序进行内存分析（查看内存分配情况、内存泄露）

优点：分析非常准确，如果发现有提示内存泄露，基本可以断定代码问题

缺点：分析效率低（真正运行了一段代码，才能对该代码进行内存分析）