# 了解和分析iOS Crash Report

nimomeng


翻译自苹果官方文档：Understanding and Analyzing Application Crash Reports

> nimo: 这篇长达1w多字的文章，大概前后翻译了一个月，“写”了三遍：第一遍是直译，第二遍是把直译改成程序员看着舒服的“行话”，第三遍是把原文里说的过于抽象或者简单的部分加上我的注解（大家看见所有以nimo开头的部分）。 文章发布后我才发现，这并不是针对iOS Crash report唯一的翻译版本。哪篇翻译的更好这个见仁见智，但我希望这篇是翻译的最用心的版本。

当app发生crash时会产生crash report，这对我们定位crash的原因非常有帮助。这篇文

档重点介绍了如何符号化、看懂并解析一篇crash Report。

> nimo: 开篇给出了这个文档的三个阶段，由浅入深为：
>
> 1. 符号化，把不可读的文档转成可读
> 2. 看懂，意思就是知道文档里哪个部分表达的什么
> 3. 解析，意思就是能从文档中定位问题，获取解决问题的有价值的信息。

## 介绍

当app发生crash时，系统会生成crash report并存储在设备上。crash report会描述app在何种情况之下被系统终止运行，一般情况下描述会包括完整的线程调用堆栈，这对app的调试（和问题的定位）是非常有帮助的。所以你应当仔细研读这些crash report，去了解你的app究竟发生的是哪种crash，并尝试修复它们。

`Crash Report`，尤其是堆栈信息，在被**符号化**之前是不可读的。所谓符号化就是把内存地址用可读的函数名和行数来替换。如果你不是从设备直接获取的crash日志，而是通过Xcode的Device Window(即通过视图操作而非手动命令行)，它们会在几秒之后自动被符号化。当然你也可以把 `.crash` 文件加入到Xcode的Device Window并自行将它符号化。

`Low Memory Report` 与其它crash report不同，它没有堆栈信息。当由于低内存而发生crash时，你必须反思你的内存使用模式和你针对低内存警告的应对方法。本文会提供给你几个内存管理的参考实现，供你参考。

## 获取Crash Report和Low Memory Report

如何调试已经部署好的iOS Apps讨论了如何从一个iOS设备直接拿到crash report和low memory report。 App发布指南里的分析Crash Reports讨论了如何查看那些crash report，这些report既包含通过TestFlight下载的测试用户处获得，又包含通过App Store下载的正式用户处获得。

# 符号化一篇Crash report

符号化指的是一种手段，这种手段指的是把堆栈信息（二进制信息）解释成源码里的方法名或者函数名，也就是所谓符号。只有符号化成功后，crash report才能帮助开发者定位问题。

> **注意**：Low Memory Report不需要被符号化（因为没有堆栈信息）。 **注意**：在MacOS平台上产生的crash report在生成的时候一般都会被完全符号化过或者半符号化过。因此本节指的符号化针对的是从iOS、watchOS乃至tvOS中提取出来的crash report。整体处理流程上，macOS的carsh report比较类似。


1
2
3
4
5
6
7
8
9
SWIFT
My Great App.app
My Great App.dsym
My Great App.xcarchive
My Great App
Feb 15, 2016, 10:08 AM
Upload to App Store...
Validate...
Export...
AdHoc/Enterprise Distribution
App Store Distribution
TestFlight Beta Testing
IPA
My Great App.crash
Privacy Diagnostics & Usage
Automatically Send
Don't Send
This Device
All Logs
Process
Type
Date/Time
My Great A...
Crash
2/15/16, 11:19 AM
App Store
1.0 Release
Adventure: -[APATrees count...
2471 devices
Adventure: -[APAMultiplayer...
1129 devices

1. 编译器在把你的源代码转换成机器码的同时，也会生成一份对应的Debug符号表。Debug符号表其实是一个映射表，它把每一个藏在编译好的binary信息中的机器指令映射到生成它们的每一行源代码中。通过build setting里的**Debug Information Format(DEBUG_INFORMATION_FORMAT)**,这些Debug符号表要么被存储在编译好的binary信息中，要么单独存储在Debug Symbol文件中(也就是dSYM文件)：一般来说，debug模式构建的app会把Debug符号表存储在编译好的binary信息中，而release模式构建的app会把debug符号表存储在dSYM文件中以节省体积。 在每一次的编译中，Debug符号表和app的binary信息通过构建时的UUID相互关联。每次构建时都会生成新的唯一的能够标识那次构建的UUID，即便你用同样的源代码，通过同样的编译setting，UUID也不会相同。相应的，dSYM文件也不能用于解析其它（UUID对应的）binary信息，即便构建自于同一个源代码。

> nimo: 意思就是说，同一次构建，app+dSYM+UUID是一套的。如果这几个文件不属于同一次构建，即便是相同的源代码，互相之间在符号化这个事情上也无法互相工作。

2. 当你为了分发app而选择Archive（存档）时，Xcode会把app的二进制信息和.dYSM文件存储在你的home文件夹下的某个地方。你可以在Xcode的Organizer里面通过”Archived”选项找到所有你存档过的app。 更多存档app的细节，请点击官方文档-分发你的App一文。

> 注意：想要解析来自于测试、app review或者客户的crash report，你需要保留分发出去的那些构建过的archive文件。

3. 如果你是通过App Store分发app或者是Test Flight分发的beta版本的app，你将在上传archive到ITC（iTunes Connect）时看见一个“是否将dSYM一起上传”的选项。在上传对话框中，请勾选”在app中包含app符号表”。上传你的dYSM文件对于从TestFlight用户和客户以及愿意分享诊断信息的客户那边接收crash report是很有必要的。更多详情请参考官方文档-分发你的App一文。

> 注意：接收自App Review的crash report是不会被符号化的，及时你再上传你的

> app到ITC时勾选了包含dSYM文件。任何来自于App Review的crash report都需要在Xcode里做符号化。

4. 当你的app 发生crash时，一个没有被符号化的crash report会被创建并存储在设备上。

5. 用户可以通过调试已部署的iOS APP里提到的方法来直接从他们的设备里获得crash report。如果你通过AdHoc或者企业证书分发app，这是你唯一能从用户获取crash report的方法。

6. 从设备上直接获取的crash report是没有被符号化的，你需要通过Xcode来符号化。Xcode会结合dSYM文件和你app的二进制信息把堆栈里的每一个地址对应到源代码中。处理后的结果就是一个符号化过的crash report。

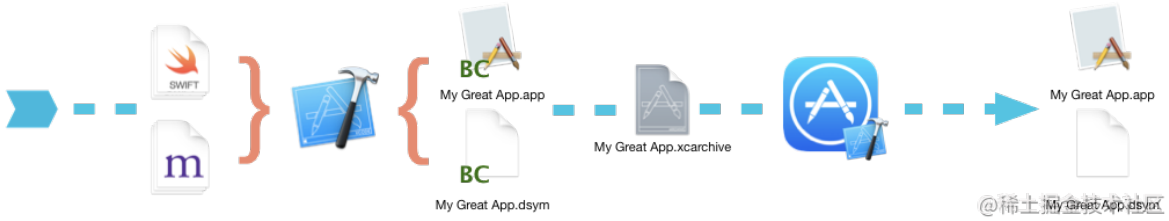
7. 如果用户愿意和Apple共享诊断信息，或者用户通过TestFlight下载了你的beta版本app，那crash report会被上传到App Store。

8. App Store在符号化crash report后会把内部所有的crash reports做汇总并分组，这种聚合（相似crash report）的方法叫做crash聚类。

9. 这些符号化后的crash report可以在你的Xcode的Crash Organizer中进行查看。

## Bitcode

Bitcode（位编码）是一个编译好的项目的中间表现形式。当你在允许bitcode的前提下Archive一个app时，编译器会在二进制中包含bitcode而不是机器码。一旦binary信息被上传到App Store中，bitcode会被再次编译成机器码。也许App Store会在将来二次编译bitcode，例如为提高编译器性能而二次编译等。不过这不重要，因为一切对你来说是透明的，也就不需要你来额外付出什么。




因为你的binary信息的最终编译结果是在App Store上体现的，因此你的Mac将不会包含那些需要对从App Review或者用户的设备那里获取到的Crash report所必须的符号化用的dSYM。

> nimo:这里原文很拗口，大概意思就是需要的东西都在App Store云端，之后的操作会自动进行，见下文。

虽然当你Archive你的app时会创建dSYM文件，但它们只能用在bitcode binary信息中，并不能用于符号化crash report。 App Store允许你从Xcode或者ITC网站中下载这些随着bitcode编译而产生的dSYM文件。 为了解析从App Review或者给你发送crash report的用户的crash report，你必须要下载这些dSYM文件，这样才能符号化crash report。 如果是从crash reporting service那里接收crash report，符号化会自动完成。

**注意**：App Store上编译的binary信息和提交的原始文件的UUID是不同的。

## 从Xcode下载dSYM文件

- 在Archives organizer，选择你之前提交到App Store的Archive文件
- 选择Download dSYM按钮Archive

Xcode会下载dSYM文件并且把他们插入到选择的Archive中。

## 从ITC网站上下载dSYM文件

- 打开App详情页面
- 点击 Activity
- 从所有的构建中，选择一个版本
- 点击 下载dSYM文件的链接

## 把"隐藏的"符号名还原成原始名

当你把一个带有bitcode的app上传到App Store时，你也许在提交对话框中并没有勾选“上传你的app的符号表信息以便从Apple那边接收符号化过的 report”的选项。 当你选择不发送符号表信息给Apple时，Xcode会在你发送app到ITC之前用晦涩难懂的符号例如”*_hidden#109*”等来替换你的app里的dSYM文件。Xcode会创建一个原始符号和”隐藏”符号的对照表，并且将其存储在Archive的app文件中的一个bcsymbolmap文件里。每一个dSYM文件都会有一个对应的bcsymbolmap文件。

在符号化crash report之前，你需要把那些从ITC中下载下来的dSYM文件中的晦涩信息给解析一下。 如果你使用Xcode中的**下载dSYM**按钮，这步解析会自动完成。但是，如果你通过**ITC网站**来下载dSYM的话，你需要打开Terminal并且手动输入下面的命令来做解析（把example的path信息和dSYM信息替换一下）

```
xcrun dsymutil -symbol-map ~/Library/Developer/Xcode/Archives/2017-11-23/MyGreatApp\ 1
```

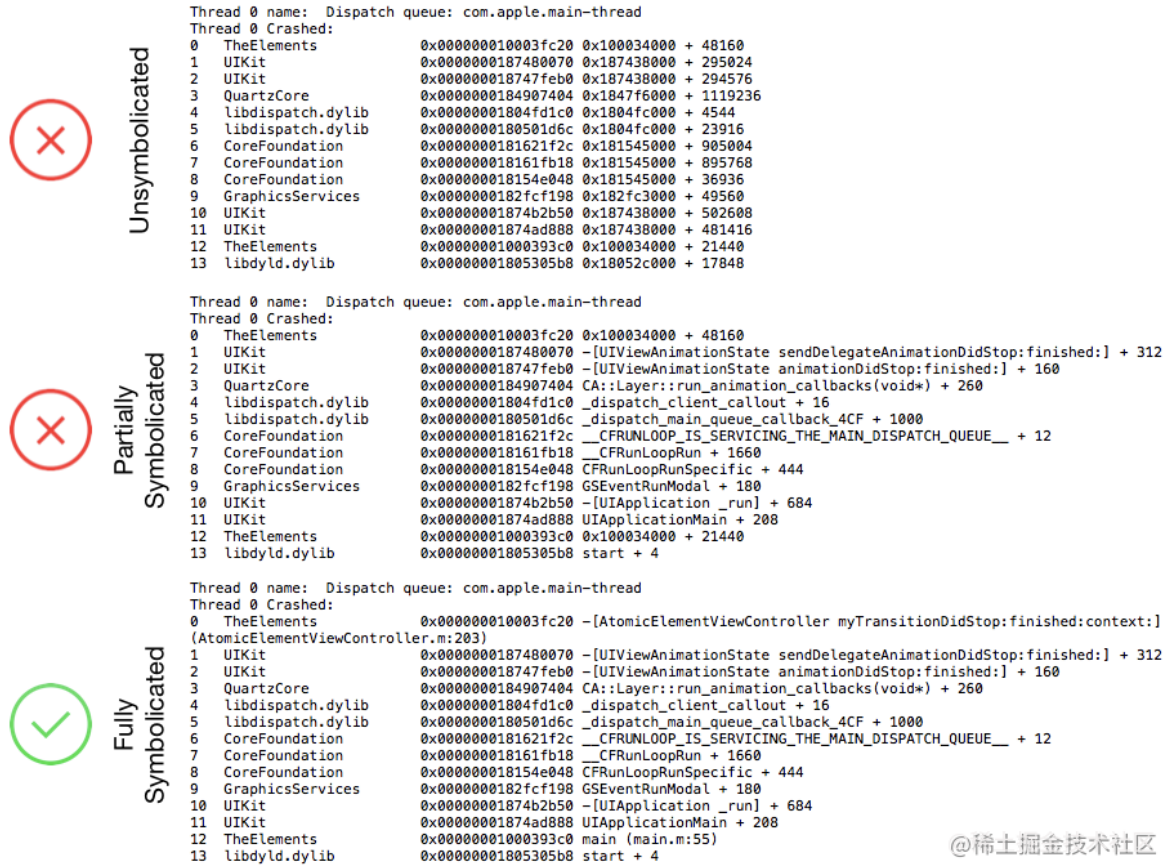


针对每一个dSYMs文件夹下的dSYM文件都运行一次这条命令。

## 如何判断Crash report是否已经符号化

一个crash report有可能未符号化，完全符号化，也有可能部分符号化。未符号化的crash report不会在堆栈信息中包含方法名或者函数名。相反，你会在加载好的binary信息中发现可执行的16进制地址信息。在完全符号化的crash report里，堆栈中的每一行16进制地址信息都会被替换成对应的符号。在部分符号化的crash report中，只有一部分堆栈信息被替换成相应的符号信息。

显然，你应当尽力去完全符号化你的crash report，因为那样你才能够获得crash report里最有价值的信息。一个部分符号化的crash report也许包含了可以理解crash的信息，这取决于crash的类型和哪一部分被成功符号化了。一个未符号化的crash report用处有限。




## 用Xcode符号化iOS的Crash report

一般来说，Xcode会自动尝试符号化它所有的Crash report。所以你只需要把crash report加到Xcode Organizer就可以了。

**Note**：Xcode只认.crash后缀的crash report。如果你收到的crash report没有后缀名或者后缀是txt，在执行下列步骤之前先把它改成.crash。

- 把iOS设备连接到你的Mac
- 从Window菜单栏选择Devices
- 在Devices左侧，选择一个设备
- 点击右边在“Device Information“ 下面的 ”View Device Logs” 按钮
- 把你的Crash report拖拽到左侧panel中
- Xcode会自动符号化Crash report并且显示结果

为了符号化一个Crash report，Xcode需要去定位如下信息：

- 崩溃的app的binary信息以及dSYM文件
- 所有app关联的自定义framework的binary信息以及dSYM文件。如果是从app构建出来的framework，它们的dYSM会随着app的dSYM文件一起拷贝到archive中。如果是第三方的framework，你需要去找作者要dYSM文件。
- 发生crash时app所依赖的OS的符号表信息。这些符号表包含了特定OS版本（例如iOS9.3.3）上的framework所需调试信息。 OS 符号表的架构具有独特性——一个64位的iOS设备不会包含armv7的符号表。Xcode将要自动拷贝你连接到的特定版本的Mac的符号表。

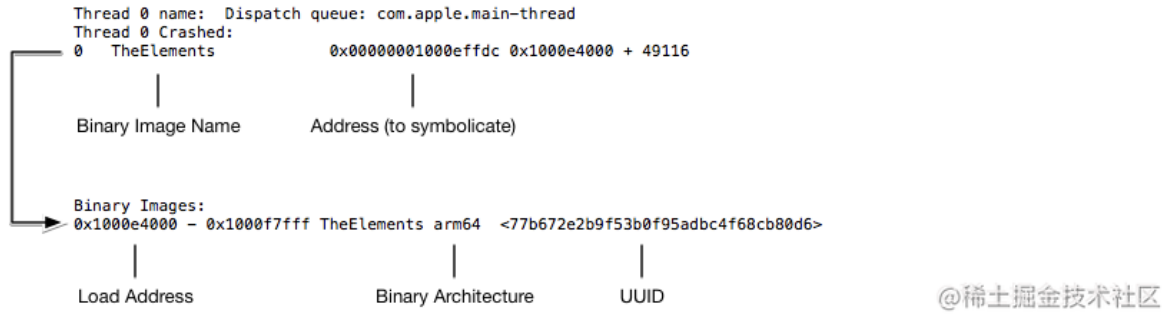
在上述任何一处，如果没有Xcode，你将无法符号化一个crash report，或者只能部分符号化一个crash report。

## 用atos符号化Crash report

`atos` 命令可以把地址里的数字替换成等价的符号。如果调试符号信息是完备的，则atos的输出信息将会包含文件名和对应的资源行数。 `atos` 命令可以被用来单独符号化那些**未符号化**或者**部分符号化**过的crash report（中的堆栈信息里的地址）。 想要使用 `atos` 符号化crash report可以按如下方式操作：

1. 找到你想要符号化的那一行，记下第二列的binary信息名，以及第三列的地址。
2. 从crash report底部的binary信息名列表中找到那个名字，记下来架构名和加载的地址。

> nimo: 例如在下图里，我们想符号化的部分就是 `0x00000001000effdc` ,binary信息名是 `The Elements` ， 底部能找到对应的名字的架构名称是 `arm64` ， 加载地址是 `0x1000e4000` 。




1. 定位二进制对应的dSYM文件。你可以用Splotlight，结合UUID，来寻找匹配的dSYM文件。（请查看相关章节。）dSYM是一个bundle，包含通过编译器在build时编译出来的DWARF调试信息（nimo: DWARF的可能的解释是，Debugging With Attributed Record Formats，是一种调试文件结构标准，结构相当的复杂）。你在使用 `atos` 时必须提供这个文件的路径，而不是dSYM的bundle路径。
2. 有了上述信息之后，你就可以把堆栈里的地址通过 `atos` 命令来符号化了。你可以符号化多条地址，通过空格来进行区分。

```xml
atos -arch <Binary Architecture> -o <Path to dSYM file>/Contents/Resources/DWARF/<bina
```

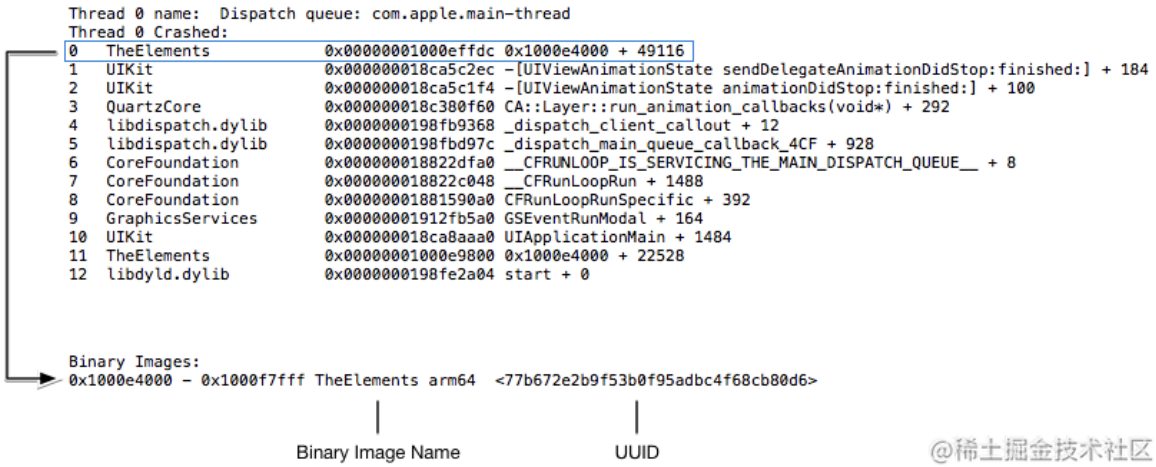
清单1 使用atos命令的样例，以及结果输出

```shell
$ atos -arch arm64 -o TheElements.app.dSYM/Contents/Resources/DWARF/TheElements -l 0x1
```

```css
-[AtomicElementViewController myTransitionDidStop:finished:context:]
```

## 利用符号化排查问题

如果Xcode没有完全符号化一个crash report，很可能是你的Mac丢失了app binary信息对应的dSYM文件，或者是丢了一个或多个app关联的framework的dSYM文件，也有可能在发生Crash时OS层面的app的设备符号表丢失了。下列步骤显示了如何使用Spotlight来判断那些可以符号化对应堆栈地址信息的dSYM文件是否在你的Mac上。




1. 在Xcode无法符号化的堆栈里找一行，注意第二列的binary信息的名字。
2. 在crash report的底部中的二进制信息列表里找到那个名字。这个列表包含了每一个crash事故现场存在于进程里的二进制信息的UUID。

> nimo:本例中需要关注的binary信息的名字是 `The Element` ， 在底部列表中对应的二进制信息的UUID是 `77b672e2b9f53b0f95adbc4f68cb80d6`

列表2 你可以用grep命令来快速找到二进制信息的列表信息

```ini
$ grep --after-context=1000 "Binary Images:" <Path to Crash Report> | grep <Binary Nam
```

3. 把二进制信息的UUID按照 8-4-4-4-12格式(XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX)转换成32个字符组成的字符串。注意所有字母必须大写。
4. 用mdfind命令，结合”com_apple_xcode_dsym_uuids == ”（包含引号）来查找UUID信息。

列表3 使用mdfind命令来通过给定UUID查找dSYM文件。

```ruby
$ mdfind "com_apple_xcode_dsym_uuids == <UUID>"
```

5. 如果spotlight找到了UUID对应的dSYM文件，mdfind会把dSYM文件和可能包含的归档文件的路径打印出来。如果一个UUID对应的dSYM文件没有找到，mdfind会直接

退出。

如果spotlight找到了二进制对应的dSYM文件，但是Xcode没有能结合二进制信息成功把地址符号化，那你应该上报一枚bug并且把crash report和对应的dSYM文件一起附到bug report中。作为权宜之策，你可以手动用atos来对地址进行符号化。

如果spotlight没有找到二进制信息对应的dSYM文件，确保你还有app发生crash的那个版本的Xcode归档文件，并且这个文件存在于spotlight可以找到的某个地方。如果你的app是支持bitcode方式构建的，确保你已经从App Store下载了最终编译版本的dSYM文件。

如果你觉得你已经有了二进制信息对应的正确的dSYM文件，那你可以用dwarfdump命令来打印对应的匹配UUID。你也可以用用dwarfdump命令来打印二进制的UUID。

```
xcrun dwarfdump --uuid <Path to dSYM file>
```

注意：你必须保存你最开始上传到App Store的发生crash的app的归档文件。dSYM文件和app二进制文件是一一对应，且每次构建都不相同。即便通过相同的源码和配置，再执行一次构建，生成的dSYM文件也无法和之前的crash report做符号化匹配。 如果你不在存有这个归档文件，你应该重新提交一次有归档的新版本，以确保再发生crash的时候你可以符号化crash report。

### 分析Crash report

这段将会讨论一篇标准crash report的各章节的含义。

## Header

每一篇crash report都有一个header。

列表4 一篇crash report的header部分

```yaml
Incident Identifier: B6FD1E8E-B39F-430B-ADDE-FC3A45ED368C
CrashReporter Key: f04e68ec62d3c66057628c9ba9839e30d55937dc
Hardware Model: iPad6,8
```

```
Process: TheElements [303]
Path: /private/var/containers/Bundle/Application/888C1FA2-3666-4AE2-9E8E-62E2F787DEC1/
Identifier: com.example.apple-samplecode.TheElements
Version: 1.12
Code Type: ARM-64 (Native)
Role: Foreground
Parent Process: launchd [1]
Coalition: com.example.apple-samplecode.TheElements [402]

Date/Time: 2016-08-22 10:43:07.5806 -0700
Launch Time: 2016-08-22 10:43:01.0293 -0700
OS Version: iPhone OS 10.0 (14A5345a)
Report Version: 104
```

大部分字段的含义是不言自明的，但是有一些值得特别指出：

- Incident Identifier: 一个crash report的唯一ID。两个 report不会使用同一个Incident Identifier。
- CrashReporter Key: 一个匿名的设备相关ID。同一个设备的两篇crash report会有相同的CrashReporter Key。
- Beta Identifier：一个整合了发生crash app的设备和供应商信息的ID。来自同一个供应商和设备的两篇report会包含相同的ID值。这个字段只有当app通过TestFlight分发的时候出现，并且出现在应该出现Crash Reporter Key Field的地方。
- Process：发生Crash时的进程名。这个和app信息属性列表里的CFBundleExecutable Key中的值可以匹配上。
- Version：发生crash的版本号。这个值可以关联到发生 crash的app的CFBundleVersion 和 CFBundleVersionString上。
- Code Type：发生crash的上下文所在架构环境。有ARM-64,ARM,X86-64和X86.
- Role：在发生crash时进程的的task_role。

> nimo：task_role的定义如下：

ini 复制代码

```
enum task_role {
        TASK_RENICED = -1,
        TASK_UNSPECIFIED = 0,
        TASK_FOREGROUND_APPLICATION,
```

```
        TASK_BACKGROUND_APPLICATION,
        TASK_CONTROL_APPLICATION,
        TASK_GRAPHICS_SERVER,
        TASK_THROTTLE_APPLICATION,
        TASK_NONUI_APPLICATION,
        TASK_DEFAULT_APPLICATION
};
```

- OS Version: OS version，包含发生crash时的所属app的编译码。

## 异常信息

遇到Objective-C/C++时不要懵（即便有些会导致Crash）。这章列出了Mach异常类型和相应的能提供crash的蛛丝马迹的一些字段信息。当然，不是所有字段都会出现在每一篇crash report里。

列表5 由于uncaught Objective-C exception而导致的进程被停止的crash report的摘录

```yaml
Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Exception Note: EXC_CORPSE_NOTIFY
Triggered by Thread: 0
```

列表6 由于反向引用了一个NULL指针而造成进程被终止的crash report的摘录

```yaml
Exception Type: EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x0000000000000000
Termination Signal: Segmentation fault: 11
Termination Reason: Namespace SIGNAL, Code 0xb
Terminating Process: exc handler [0]
Triggered by Thread: 0
```

可能出现在这一章节的某些字段解读如下。

- Exception Codes： 和异常是有关的处理器指定信息，这些信息会被编码成一个或者

多个64位二进制数字。一般来说，这个字段不应该存在，因为crash report生成时会把exception code转化成可读的信息并在其它字段进行体现。

- Exception Subtype：可读的exception code的名称。
- Exception Message：从exception code中解析出来的附加的可读信息。
- Exception Note：不特指某一种异常的额外信息。如果这个字段包含”SIMULATED”(不是Crash)，则进程并没有发生crash，而是在系统层面被kill掉了，比如看门狗机制。

> nimo: 为了防止一个应用占用过多的系统资源，苹果工程师门设计了一个“看门狗”的机制。“看门狗”会监测应用的性能。如果超出了该场景所规定的运行时间，“看门狗”就会强制终结这个应用的进程。 开发者们在crashlog里面，会看到诸如0x8badf00d这样的错误代码(看起来很像bad food，看门狗吃到了坏的食物，不嗨森)。 看门狗触发条件如下：
>
> | 触发时机 | 看门狗出动的时间 |
> | --- | --- |
> | 启动 | 20秒 |
> | 恢复运行 | 10秒 |
> | 悬挂进程 | 10秒 |
> | 退出应用 | 6秒 |
> | 后台运行 | 10分钟 |

- Termination Reason：当进程被终止时的原因及信息。关键的信息模块，不论是进程内还是进程外，当遇到一个致命错误（fatal error，例如bad code signature，缺失依赖库，不恰当的访问私有敏感信息等）。MacOS Sierra,iOS 10, watch OS3和tvOS 10已经采用新的架构去记录这些错误信息，所以这些系统之下的crash report会在Termination Reason这个字段里描述error message信息。
- Triggered by Thread：指出异常是在哪个线程发生的

接下来的章节会解释常见的异常类型：

## Bad Memory Access [EXC_BAD_ACCESS // SIGSEGV // SIGBUS]

进程试图去访问无效的内存空间，或者尝试访问的方法是不被允许的（例如给只读的内存空间做写操作）。在Exception Subtype字段中如果出现kern_return_t的话，说明内存地址空间被不正确的访问了。 这里有几个调试bad memory crash的小贴士：

- 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。你应当用Zombie instrument方式来运行profile，来更好地了解发生crash的原因。
- 如果gpus_ReturnNotPermittedKillClient在近crash的线程附近，则进程有可能是尝试在后台通过OpenGL ES或者Metal来做渲染。可以参见 QA1766: How to fix OpenGL ES application crashes when moving to the background
- 通过在运行你的app时勾选Address Sanitizer。address sanitizer会在编译期间在内存访问时添加额外的操作，当你的app运行，Xcode会在内存可能发生crash的时候给出提示信息。

## Abnormal Exit [EXC_CRASH // SIGABRT]

进程异常退出。这种异常最常见的原因在于 `uncaught Objective-C/C++ exception` 并且调用了 `abort()` 。 扩展App(nimo：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。

## Trace Trap [EXC_BREAKPOINT // SIGTRAP]

和 `Abnormal Exit` 类似，这种异常是由于在特殊的节点加入debugger调试节点的原因。你可以在你自己的代码里通过使用 `__builtin_trap()` 函数来触发这个异常。如果没有debugger存在，则线程会被终止并生成一个crash report。 底层库（例如libdispatch）

会在遇到fatal错误的时候陷入这个困局。关于错误的相关信息会在crash report的章节或者是设备的的打印信息里找到。 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。

> nimo: 可选绑定，就是类似如下语句的使用

```scss 复制代码
if let actualValue = maybeHasValue(){
        foo(actualValue)
}
```

## Illegal Instruction [EXC_BAD_INSTRUCTION // SIGILL]

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_INSTRUCTIONY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。

## Quit [SIGQUIT]

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）。

## Killed[SIGKILL]

进程收到系统指令被干掉。请自行查看Termination Reason来定位线程被干掉的原因。Termination Reason字段会包含一个命名空间和代码。以下代码只针对watchOS：

- 代码 `0xc51bad01` 表示watchOS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。
- 代码 `0xc51bad02` 表示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。
- 代码 `0xc51bad03` 表示watch app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。

## Guarded Resource Violation [EXC_GUARD]

进程违规访问了一个被保护的资源。系统库会把特定的文件描述符标记为被 `被保护` ，因此任何对这些文件描述符的常规操作都会抛出 `EXC_GUARD` 异常（nimo: 当系统想操作这些文件描述符时，它们会用特殊的被授权过的私有API）。所以遇到诸如私自关闭掉系统打开的文件描述符之类的操作时您可以快速察觉。例如，如果一个app关闭掉了曾经支持Core Data 存储的SQLite文件的文件描述符，你会发现Core Data过一会儿神秘crash。guard exception会在不久之后注意到并且让他们更容易被debug。 更新版本的iOS crash report会在Exception Subtype和Exception Message字段里包含关于 `EXC_GUARD` 异常的可读详细信息。在macOS或者是更老版本的iOS的crash report中，这条信息会被加密成第一个Exception Code并以位信息进行呈现，它可以被这么解读：

- [63:61] - Guard Type:被保护的资源的类型。0x2值表示资源是一个文件描述符。
- [60:32] - Flavor:在何种情况之下出现的问题。 如果第一个(1<<0)bit被设值，则进程尝试在一个被保护的文件描述符上调用close()

如果第二个(1<<1)bit被设值，则进程尝试在被保护的文件描述符上用 `F_DUPFD 或 F_DUPFD_CLOEXEC` 调用 `dup(), dup2(), 或 fcntl()` 命令。

如果第三个(1<<2)bit被设值，则进程尝试通过socket发送一个被保护的文件描述符。

如果第五个(1<<4)bit被设值，则进程尝试写一个被保护的文件描述符。

- [31:0] - File Descriptor：进程尝试修改被保护的文件描述符。

## Resource Limit [EXC_RESOURCE]

进程的资源超过限定阈值。这条推送是OS发出的，表示进程占有了太多的资源。准确的资源列在了 `Exception Subtype` 的字段里。如果 `Exception Note` 字段包含了 `NON-FATAL CONDITION` （非严重错误），则即便是生成了crash report，进程也不会被kill掉。

- 如果 `EXCEPTION SUBTYPE` 里出现 `MEMORY` 则暗示了进程占用已经超过系统限制。如果之后出现由于系统占用过多进程被Kill，可能和这有关。
- 如果 `EXCEPTION SUBTYPE` 里出现 `WAKEUP` 则暗示线程每秒被进程唤醒太多次了，进而导致CPU被频繁唤醒并且造成电量损耗。 通常，这种事发生在进程间通信（通过 `peformSelector:onThread:` 或者 `dispatch_async` ），而且会远比预想的发生的更频繁。因为发生这种异常的通信被触发的如此频繁，所以很多后台进程会出现彼此高度雷同的堆栈信息——恰恰暗示了它们是从哪儿来的。

## Other Exception Types

有些crash report可能会出现无名的Exception Type，这时候在这个字段上会出现纯16进制值（例如00000020）。如果你收到这样的crash report，直接去Exception Code查看更多信息。

- 如果Exception Code是 `0xbaaaaaad` 则说明此条logs是系统堆栈快照，并非crash report。可以通过同时按（手机）侧边按钮和音量键来记录堆栈快照。通常情况下，这些logs是用户无意中生成的，并非表示错误。
- 如果Exception Code是 `0xbad22222` 表示一个VoIP应用因为频繁暂停被iOS系统终止掉。
- 如果Exception Code是 `0x8badf00d` （读起来像badfood）则说明一个应用因为触发了看门狗机制被iOS系统终止掉，有可能是应用花了太长时间启动，终止，或者是响应系统事件。一种常见原因是在主线程上做网络同步逻辑。不论Thread0上（也就是主线程）想做什么（重要的事），都应该转移到后台线程，或者换一种方式触发，这样它才不会阻塞主线程。
- 如果Exception Code是 `0xc00010ff` 则说明app因为环境过热（的事件）被iOS系统干掉了。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。如果想知道更多高效运行app的tips，请参考WWDC的文章: iOS Performance and Power Optimization with Instruments。
- 如果Exception Code是 `0xdead10cc` (读起来像deadlock)则说明一个应用被系统终止掉，原因是在应用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。
- 如果Exception Code是 `0x2bad45ec` 则说明app因为违规操作（安全违规）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。

> **Note**：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。

### 额外的诊断信息

本章节包含终止相关的额外诊断信息，包括：

- 应用的具体信息：在进程被终止前捕捉到的框架错误信息

- 内核信息：关于代码签名问题的细节
- Dyld （动态链接库）错误信息：被动态链接器提交的错误信息

从macOS Sierra, iOS 10, watchOS 3, 和 tvOS 10开始，大部分这种信息都在 `Exception Information` 的 `Termination Reason` 字段下了。 你应当阅读本章节来更好的明白当进程被终止的时候发生了什么。

表7：一段因为找不到链接库而导致进程被终止的crash report的摘录

```typescript
Dyld Error Message:
Dyld Message: Library not loaded: @rpath/MyCustomFramework.framework/MyCustomFramework
  Referenced from: /private/var/containers/Bundle/Application/CD9DB546-A449-41A4-A08B-
  Reason: no suitable image found.
```

表8：一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

```sql
Application Specific Information:
com.example.apple-samplecode.TheElements failed to scene-create after 19.81s (launch t
Elapsed total CPU time (seconds): 7.690 (user 7.690, system 0.000), 19% CPU
Elapsed application CPU time (seconds): 0.697, 2% CPU
```

## 堆栈信息

一个crash report最有意思的部分一定是每个线程在被终止时的堆栈信息。这些信息和你在debug时看到的很类似。

列表9：一个完全符号化的crash report的堆栈部分摘录

```objectivec
Thread 0 name: Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   TheElements                     0x000000010006bc20 -[AtomicElementViewControll
1   UIKit                           0x0000000194cef0f0 -[UIViewAnimationState send
2   UIKit                           0x0000000194ceef30 -[UIViewAnimationState anim
3   QuartzCore                      0x0000000192178404 CA::Layer::run_animation_ca
4   libdispatch.dylib               0x000000018dd6d1c0 _dispatch_client_callout +
```

```
5   libdispatch.dylib                0x000000018dd71d6c _dispatch_main_queue_callba
6   CoreFoundation                   0x000000018ee91f2c __CFRUNLOOP_IS_SERVICING_TH
7   CoreFoundation                   0x000000018ee8fb18 __CFRunLoopRun + 1660
8   CoreFoundation                   0x000000018edbe048 CFRunLoopRunSpecific + 444
9   GraphicsServices                 0x000000019083f198 GSEventRunModal + 180
10  UIKit                            0x0000000194d21bd0 -[UIApplication _run] + 684
11  UIKit                            0x0000000194d1c908 UIApplicationMain + 208
12  TheElements                      0x0000000100653c0 main (main.m:55)
13  libdyld.dylib                    0x000000018dda05b8 start + 4

Thread 1:
0   libsystem_kernel.dylib           0x000000018deb2a88 __workq_kernreturn + 8
1   libsystem_pthread.dylib          0x000000018df75188 _pthread_wqthread + 968
2   libsystem_pthread.dylib          0x000000018df74db4 start_wqthread + 4

...
```

第一行列出了当前的线程号，以及当前的执行队列的id。其余各行列出来每一个堆栈中堆栈片段信息，从左到右分别是：

- 堆栈片段号。堆栈的展示顺序会和调用顺序一致，片段0是在程序被终止时执行的函数。片段1是调用片段0的函数，以此类推。
- 在堆栈片段中驻留的执行函数的名称
- 片段0代表机器指令在被终止的生活所在的地址。其它片段表示如果片段0执行完成之后下一个执行的片段地址
- 在一个符号化的crash report中，代表在堆栈片段中的函数名称

## 异常

Objective-C中的异常通常用来表明在运行时发生的代码错误，例如越界访问数组，或者更改immutable的对象，没有实现protocol中必须实现的方法，或者给接收者无法识别的对象发送信息。

> **Note**：给之前已经释放的对象发送消息会引发 `NSInvalidArgumentException` 异常进而crash，而非内存访问违规。这会在新的变量正好占据了之前释放变量所在内存时。如果你的app因为 `NSInvalidArgumentException` 发生crash（在堆栈信

> 息中查看 `[NSObject(NSObject) doesNotRecognizeSelector:]` ），考虑通过 Zombies instrument 来profiling你的应用，来排除刚才提到的内存管理问题。

如果异常没有被捕捉到，他会被一个叫 `uncaught exception` 方法所拦截。默认的 `uncaught exception` 的日志会显示到设备的控制台，之后会终止进程。异常堆栈信息会在生成的crash report的 `上一个异常堆栈（Last Exception Backtrace）` 下，就像列表10所写。异常消息会被crash report忽略。如果你收到了一个带有上一个异常堆栈（Last Exception Backtrace）的crash report，你应当去获取原始设备并获取其控制台日志信息，来更好的了解发生crash的原因。

List10：发生了上一个异常堆栈（Last Exception Backtrace）的未符号化crash report摘录

```php
Last Exception Backtrace:

(0x18eee41c0 0x18d91c55c 0x18eee3e88 0x18f8ea1a0 0x195013fe4 0x1951acf20 0x18ee03dc4 0
```

一个只包含16进制信息的有Last Exception Backtrace信息的crash日志必须被符号化，以获取有价值的堆栈信息，就像列表11所写。

列表11：一个包含Last Exception Backtrace信息的符号化的crash report。这个异常出现在加载app的storyboard时，需要响应的IBOutlet的对应元素丢失了。

```objectivec
Last Exception Backtrace:

0   CoreFoundation                  0x18eee41c0 __exceptionPreprocess + 124
1   libobjc.A.dylib                 0x18d91c55c objc_exception_throw + 56
2   CoreFoundation                  0x18eee3e88 -[NSException raise] + 12
3   Foundation                      0x18f8ea1a0 -[NSObject(NSKeyValueCoding) setVa
4   UIKit                           0x195013fe4 -[UIViewController setValue:forKey
5   UIKit                           0x1951acf20 -[UIRuntimeOutletConnection connec
6   CoreFoundation                  0x18ee03dc4 -[NSArray makeObjectsPerformSelect
7   UIKit                           0x1951ab8f4 -[UINib instantiateWithOwner:optio
8   UIKit                           0x195458128 -[UIStoryboard instantiateViewCont
9   UIKit                           0x19545fa20 -[UIStoryboardSegueTemplate instan
10  UIKit                           0x19545fc7c -[UIStoryboardSegueTemplate _perfo
```

```
11  UIKit                       0x19545ff70 -[UIStoryboardSegueTemplate perfor
12  UIKit                       0x194de4594 -[UITableView _selectRowAtIndexPat
13  UIKit                       0x194e94e8c -[UITableView _userSelectRowAtPend
14  UIKit                       0x194f47d8c _runAfterCACommitDeferredBlocks +
15  UIKit                       0x194f39b40 _cleanUpAfterCAFlushAndRunDeferred
16  UIKit                       0x194ca92ac _afterCACommitHandler + 168
17  CoreFoundation              0x18ee917dc __CFRUNLOOP_IS_CALLING_OUT_TO_AN_O
18  CoreFoundation              0x18ee8f40c __CFRunLoopDoObservers + 372
19  CoreFoundation              0x18ee8f89c __CFRunLoopRun + 1024
20  CoreFoundation              0x18edbe048 CFRunLoopRunSpecific + 444
21  GraphicsServices            0x19083f198 GSEventRunModal + 180
22  UIKit                       0x194d21bd0 -[UIApplication _run] + 684
23  UIKit                       0x194d1c908 UIApplicationMain + 208
24  TheElements                 0x1000ad45c main (main.m:55)
```



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）。

## Thread State(线程状态)

这章列出了crash线程的线程状态。这里列出了注册过的值。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，这也许会起一些帮助。

列表12：ARM64设备的crash report的一段Thread State摘录

```yaml
Thread 0 crashed with ARM Thread State (64-bit):
```

```
 x0: 0x0000000000000000   x1: 0x000000019ff776c8   x2: 0x0000000000000000   x3: 0x0
 x4: 0x0000000000000000   x5: 0x0000000000000001   x6: 0x0000000000000000   x7: 0x0
 x8: 0x0000000100023920   x9: 0x0000000000000000  x10: 0x000000019ff7dff0  x11: 0x0
x12: 0x000000013e63b4d0  x13: 0x000001a19ff75009  x14: 0x0000000000000000  x15: 0x0
x16: 0x0000000187b3f1b9  x17: 0x0000000181ed488c  x18: 0x0000000000000000  x19: 0x0
x20: 0x000000013fa49560  x21: 0x0000000000000001  x22: 0x000000013fc05f90  x23: 0x0
x24: 0x0000000000000000  x25: 0x000000019ff776c8  x26: 0xee009ec07c8c24c7  x27: 0x0
x28: 0x0000000000000000  fp: 0x000000016fdf29e0   lr: 0x0000000100017cf8
 sp: 0x000000016fdf2980   pc: 0x0000000100017d14 cpsr: 0x60000000
```

## Binary Images

这一章列出了在进程被终止时加载在进程中的二进制文件（binary images）。

列表13：一段crash report的完整二进制文件摘录

```
Binary Images:

0x100060000 - 0x100073fff TheElements arm64 <2defdbea0c873a52afa458cf14cd169e> /var/co

...
```

每一行都包含了一个二进制文件的以下细节信息：

- 在进程内的二进制文件的地址空间
- 一段二进制的名称或者bundle id（仅针对macOS）。一个MacOS的crash report，如果二进制时OS的一部分，会在前面加上 `a` 。
- （仅针对macOS）二进制的短版本(short version)和bundle版本，通过破折号来分割。
- （仅针对iOS）二进制文件的架构名。一个二进制可能包含多个分片，每一个架构它都支持。其中只有一个可以被加载到进程中。
- 一个可以唯一标示二进制文件的id，即UUID。这个值会随每一次构建而发生变化，并且它会用来定位需要符号化时的dSYM文件。
- 磁盘上二进制文件的path。

## 读懂低内存 report(Low Memory Reports)

当系统检测到内存不足时，iOS系统里的虚拟内存系统会协同各应用来做内存释放。每个运行着的应用都会接收到系统发来低内存推送（Low-memory notification)，要求释放内存空间，从而达到减少整体内存消耗的目的。如果内存压力依然存在，系统可能会终止后台进程以减轻内存压力。如果（整体）内存释放够了，你的应用将可以继续运行；不然，你的应用会被iOS终止，因为可供你的应用运行的内存不够，这时候会生成一个低内存 report（Low-Memory Report）并存储在你的设备中。 低内存 report的格式和其它crash report略有不同，它没有应用的堆栈信息。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。

如果你没有在你的应用或者扩展程序里看到原因，那crash的原因就不是低内存压力。仔细查看一下.crash文件（在之前章节里有写）。

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。

## 相关文档

如果想查看如何使用Zombies模板工具来修复内存释放的crash，可以查看Eradicating Zombies with the Zombies Trace Template 。 如果想查看应用归档的信息，请参考 App Distribution Guide 。 如果想了解关于crash logs的解读，请参考Understanding Crash Reports on iPhone OS WWDC 2010 Session 。

分类： iOS 标签： iOS