iOS 开发中，如何合理地制造“BUG”并且查找 BUG

可能在平时的编程实践中，往往简单的把 BUG 与 Crash 基本等价了。而且我们很多精力也都放在解决 Crash 的 Bug 上面。而对于没有 Crash 掉的 BUG，似乎没有过多的关注。但是，实际情况上那些让人痛彻心扉的“天坑”往往是那些没有 Crash 掉的 BUG 造成的，比如前一段时间 OpenSSL 心脏大出血。为什么这么说呢？且听我慢慢道来。

什么是 BUG，简单点说就是，程序没有按照我们预想的方式运行。我比较喜欢把 BUG 分成两类：

1. Crash 掉的
2. 没有 Crash 掉的

可能在平时的编程实践中，往往简单的把 BUG 与 Crash 基本等价了。而且我们很多精力也都放在解决 Crash 的 Bug 上面。而对于没有 Crash 掉的 BUG，似乎没有过多的关注。但是，实际情况上那些让人痛彻心扉的“天坑”往往是那些没有 Crash 掉的 BUG 造成的，比如前一段时间 OpenSSL 心脏大出血。为什么这么说呢？且听我慢慢道来。

如何合理的制造 BUG

Crash 掉的 BUG，用程序的死证明了你的程序存在问题，你必须抓紧时间来解决程序的问题了。而没有 Crash 掉的 Bug，像是一个善于撒谎的 人，伪装成可以正常运转的样子，让整个程序运行在一个不稳定的状态下。虽然外表看起来好好地（没有 crash），但是里子早就烂透了，一旦报露出问题往往 是致命的，比如 OpenSSL 的心脏大出血。这就是前人总结的“死程序不说谎”。

Crash 不可怕，可怕的是程序没有 Crash 而是运行在一个不稳定的状态下，如果程序还操作了数据，那带来的危害将是灾难性的。

所以放心的让程序 Crash 掉吧，因为当他 Crash 掉的时候，你还有机会去修正自己的错误。如果没有 Crash，那就有可能要给整个程序和产品收 尸了。因此合理制造“BUG”的原则之一，也是最大的原则就是：尽量制造 Crash 的 BUG，减少没有 Crash 的 BUG，如果有可能将没有 Crash 掉 的 Bug 转换成 Crash 的 BUG 以方便查找。

NSAssert

这个应该都比较熟悉，他的名字叫做“断言”。断言（assertion）是指在开发期间使用的、让程序在运行时进行自检的代码（通常是一个子程序或 宏）。断言为真，则表明程

序运行正常，而断言为假，则意味着它已经在代码中发现了意料之外的错误。断言对于大型的复杂程序或可靠性要求极高的程序来说尤其 有用。而当断言为假的时候，几乎所有的系统的处理策略都是，让程序死掉，即 Crash 掉。方便你知道，程序出现了问题。

断言其实是“防御式编程”的常用的手段。防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据。这种思 想是将可能出现的错误造成的影响控制在有限的范围内。断言能够有效的保证数据的正确性，防止因为脏数据让整个程序运行在不稳定的状态下面。

关于如何使用断言，还是参考《代码大全 2》中“防御式编程”一章。这里简单的做了一点摘录，概括其大意：

1. 用错误处理代码来处理预期会发生的状况，用断言来处理绝不应该发生的状况。

2. 避免把需要执行的代码放到断言中

3. 用断言来注解并验证前条件和后条件

4. 对于高健壮性的代码，应该先使用断言再处理错误

5. 对来源于内部系统的可靠的数据使用断言，而不要对外部不可靠的数据使用断言，对于外部不可靠数据，应该使用错误处理代码。

而在 iOS 编程中，我们可以使用 NSAssert 来处理断言。比如：

```
1. - (void)printMyName:(NSString *)myName
2. {
3.     NSAssert(myName == nil, @"名字不能为空！");
4.     NSLog(@"My name is %@.",myName);
5. }
```

我们验证 myName 的安全性，需要保证其不能为空。NSAssert 会检查其内部的表达式的值，如果为假则继续执行程序，如果不为假让程序 Crash 掉。

每一个线程都有它自己的断言捕获器（一个 NSAssertionHanlder 的实例），当断言发生时，捕获器会打印断言信息和当前的类名、方法名等信息。然后抛出一个 NSInternalInconsistencyException 异常让整个程序 Crash 掉。并且在当前线程的断言捕获器中执行 handleFailureInMethod:object:file:lineNumber:description:以上述信息为输出。

当时，当程序发布的时候，不能把断言带入安装包，你不想让程序在用户机器上 Crash 掉吧。打开和关闭断言可以在项目设置中设置：

在 release 版本中设置了 NS_BLOCK_ASSERTIONS 之后断言失效。

尽可能不要用 Try-Catch

并不是说 Try-Catch 这样的异常处理机制不好。而是，很多人在编程中，错误了使用了 Try-Catch，把异常处理机制用在了核心逻辑中。把其当成了一个变种的 GOTO 使用。把大量的逻辑写在了 Catch 中。弱弱的说一句，这种情况干嘛不用 ifelse 呢。

而实际情况是，异常处理只是用户处理软件中出现异常的情况。常用的情况是子程序抛出错误，让上层调用者知道，子程序发生了错误，并让调用者使用合适的策略来处理异常。一般情况下，对于异常的处理策略就是 Crash，让程序死掉，并且打印出堆栈信息。

而在 iOS 编程中，抛出错误的方式，往往采用更直接的方式。如果上层需要知道错误信息，一半会传入一个 NSError 的指针的指针：

```
1. - (void) doSomething:(NSError* __autoreleasing*)error
2. {
3.     ...
4.     if(error != NULL)
5.     {
6.         *error = [NSError new];
7.     }
8.     ....
9. }
```

而能够留给异常处理的场景就极少了，所以在 iOS 编程中尽量不要使用 Try-Catch。

（PS：见到过使用 Try-Catch 来防止程序 Crash 的设计，如果不是迫不得已，尽量不要使用这种策略）

尽量将没有 Crash 掉的 BUG，让它 Crash 掉

上面主要讲的是怎么知道 Crash 的“BUG”。对于合理的制造“BUG”还有一条就是尽量把没有 Crash 掉的“BUG”，让他 Crash 掉。这 个没有比较靠谱的方法，靠暴力吧。比如写一些数组越界在里面之类的。比如那些难调的多线程 BUG，想办法让他 Crash 掉吧，crash 掉查找起来就比较 方便了。

总之，就是抱着让程序“死掉”的心态去编程，向死而生。

如何查找 BUG

其实查找 BUG 这个说法，有点不太靠谱。因为 BUG 从来都不需要你去找，他就在那里，只增不减。都是 BUG 来找你，你很少主动去找 BUG。程序死 了，然后我们就得加班加点。其实我们找的是发生 BUG 的原因。找到引发 BUG 的罪魁祸首。说的比较理论化一点就是：在一堆可能的原因中，找到那些与 BUG 有因果性的原因（注意，是因果性，不是相关性）。

于是解决 BUG 一般可以分两步进行：

1. 合理性假设，找到可能性最高的一系列原因。
2. 对上面找到的原因与 BUG 之间的因果性进行分析。必须确定，这个 BUG 是由某个原因引起的，而且只由改原因引起。即确定特定原因是 BUG 的充分必要条件。

找到原因之后，剩下的事情就比较简单了，改代码解决掉。

合理性假设

其实，BUG 发生的原因可以分成两类：

1. 我们自己程序的问题。
2. 系统环境，包括 OS、库、框架等的问题。

前者找到了，我们可以改。后者就比较无能为力了，要么发发牢骚，要么 email 开发商，最后能不能被改掉就不得而知了。比如 iOS 制作 framework 的时候，category 会报方法无法找的异常，到现在都没有解决掉。

当然，一般情况下导致程序出问题的原因的 99.999999%都是我们自己造成的。所以合理性假设第一条：

```
首先怀疑自己和自己的程序，其次怀疑一切。
```

而程序的问题，其实就是开发者自己的问题。毕竟 BUG 是程序员的亲子亲孙，我们一手创造了 BUG。而之所以能够创造 BUG，开发者的原因大致有三：

1. 知识储备不足，比如 iOS 常见的空指针问题，发现很多时候就是因为对于 iOS 的内存管理模型不熟悉导致。
2. 错心大意，比较典型的就是数组越界错误。还有在类型转化的时候没注意。比如下面这个程序：

```
1. //array.count = 9
2. for (int i = 100; array.count - (unsigned int)i &gt; 10 ; )
3. {
4.     i++
5.     .....
6. }
```

按道理讲，这应该是个可以正常执行的程序，但是你运行的话是个死循环。可能死循环的问题，你改了很多天也没解决。直到同事和你说 array.count 返回的是 NSUInterge，当与无符号整形相间的时候，如果出现负值是回越界的啊。你才恍然大悟：靠，类型的问题。

3. 逻辑错误

这个就是思维方式的问题，但是也是问题最严重的。一旦发生，很难查找。人总是最难怀疑自己的思维方式。比如死循环的问题，最严重的是函数间的循环引用，还有多线程的问题。

但是庆幸的是绝大多数的 BUG 都是由于知识储备不足和粗心大意造成的。所以合理性假设的第二条：

首先怀疑基础性的原因，比如自己知识储备和粗心大意等人为因素，通过这些原因查找具体的问题。之后再去怀疑难处理的逻辑错误。

有了上面的合理性怀疑的一些基本策略，也不能缺少一些基本的素材啊。就是常见的 Crash 原因，最后我们还是得落地到这些具体的原因或者代码上，却找与 BUG 的因果性联系。

访问了一个已经被释放的对象，比如

```
1. NSObject * aObj = [[NSObject alloc] init];
2. [aObj release];
3. NSLog(@"%@", aObj);
```

2. 访问数组类对象越界或插入了空对象
3. 访问了不存在的方法
4. 字节对齐，(类型转换错误)

5. 堆栈溢出
6. 多线程并发操作
7. Repeating NSTimer

合理性假设第三条：尽可能的查找就有可能性的具体原因。

因果性分析

首先必须先说明的是，我们要找的是“因果性”而不是“相关性“。这是两个极度被混淆的概念。而且，很多时候我们错误的把相关性当成了因果性。比如， 在解决一个多线程问题的时候，发现了一个数据混乱的问题，但是百思不得其解。终于，有一天你意外的给某个对象加了个锁，数据就正常了。然后你就说这个问题 是这个对象没有枷锁导致的。

但是，根据上述你的分析，只能够得出该对象枷锁与否与数据异常有关系，而不能得出就是数据异常的原因。因为你没能证明对象加锁是数据异常的充分必要 条件，而只是使用了一个单因变量实验，变量是枷锁状态，取值 x=[0，1],x 为整形。然后实验结果是枷锁与否与数据异常呈现正相关性。

1. 相关性：在概率论和统计学中，相关（Correlation，或称相关系数或关联系数），显示两个随机变量之间线性关系的强度和方向。在统计学中，相关的意义是用来衡量两个变量相对于其相互独立的距离。在这个广义的定义下，有许多根据数据特点而定义的用来衡量数据相关的系数。
2. 因果性：因果是一个事件（即“因”）和第二个事件（即“果”）之间的关系，其中后一事件被认为是前一事件的结果。

错误的把相关性等价于因果性。不止是程序员，几乎所有人常见的逻辑错误。为了加深认识，可以看一下这篇小科普：相关性 ≠ 因果性。

因果性分析的首要问题就是，别被自己的逻辑错误欺骗，正确的分辨出相关性和因果性之间的区别。不要把相关性等价于因果性。

之后便是因果性分析的内容了，之前一直反复说，因果性分析的目的就是确定特定原因是 BUG 发生的充分必要条件。那么确定这个事情，就需要两步：

1. 充分性证明
2. 必要性证明

关于充分性证明，这个基本上就是正常的逻辑推理。基本思路就是，能够还原出 BUG 出现的路径，从原因到 BUG 发生处的代码，走了怎样的函数调用和控制逻辑。确定了这个基本上就能够证明充分性。一般情况下根据 Crash 的堆栈信息能够，非常直接的证明充分性。

关于必要性证明，这个就比较困难了。充分性和必要性的定义如下：当命题“若 A 则 B”为真时，A 称为 B 的充分条件，B 称为 A 的必要条件。那么必要性就 是，BUG 能够作为导致 BUG 的原因的原因。这个说法比较拗口。换种说法，就是你得确认这个 BUG 能够解释原因，这个 BUG 就是而且只是这个原因造成的。

只有证明了充分必要性，才能算是真正找到了 BUG 的原因。