程序员的 bug 修复宝典

作者：xuexiangjys
原文链接：https://ld246.com/article/1609094004701
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

<h2 id="前言">前言</h2>
<blockquote>
bug, 又名程序缺陷或者程序漏洞, 是每个程序员每天都回避不了的东西。程序员对 bug 的感情谓是五味杂陈：一方面 bug 非常可恶，尤其是一些偶现的 bug，它强大到可以摧毁一个优秀程序员意志；另一方面很多 bug 又是程序员自己亲手写下的，无奈之余只能自嘲一句：不写 bug 我们就要业了！
</blockquote>
作为一名职业程序员，同时也是一名开源创作者, 夸张点说，我解过的 bug 可以绕地球一圈, 每写 bug 解 bug 几乎是我的日常。
但是，作为一个善于思考和总结的技术 up 主，我怎么能止步于每天写 bug 和解 bug 呢？更何，人生在世，总得有点追求。既然我不能够阻止 bug 的产生，那么就让我总结一点 bug 的修复技巧让 bug 消失地更快点吧!
<hr>

<h2 id="1-bug修复的生命周期">1.bug 修复的生命周期</h2>
<blockquote>
中医讲究"望闻问切"，其实修复一个 bug 就像给病人看一场病，本质上是相通的。
</blockquote>
当我们遇到一个 bug(问题)的时候，一般我们需要经历如下 6 个步骤：

1.<code>了解bug</code>。我们首先需要到底出了什么 bug，现象是什么，怎样发生的。
2.<code>复现bug</code>。在了解了 bug 的大致情况之后，我们需要能够找到复现的路径，就为后面 bug 的定位提供可靠的依据。
3.<code>定位bug</code>。当有了稳定的复现途径之后，要做的就是打断点、打日志进行调，来一步一步分析和定位 bug，到底是那块代码导致的错误。
4.<code>确认bug</code>。当我们定位到 bug 出错的地方之后，我们就需要分析这到底是不 bug。如果是 bug，那么这个 bug 出现的根源是什么，到底能不能解决。
5.<code>修复bug</code>。在明确了 bug 的根本原因之后，下面就需要发挥我们的聪明才智修复这个 bug 了。
6.<code>验证bug</code>。并不是每次我们修复完 bug 之后就可以万事大吉了，此时我们还要去重现 bug 以确保 bug 被真正修复。除此之外，有条件的我们还需要去验证相关场景，以保证修该 bug 不会引入其他 bug。

以上可以总结为 12 字方针--"了解、复现、定位、确认、修复、验证"bug。一般在稍微大一点公司，都会有对应的流程对 bug 的修复进行流程控制，最终形成闭环。
可以看到的是，其实修复 bug 只是解决一个 bug 的 6 个步骤中的其中一步。很多刚刚参与工作程序员经常犯的错误就是一遇到 bug，就开始漫无目的地看代码或者是上网各种瞎搜索，又或者各种脑问，最后搞了一圈可能连自己要解决的 bug 到底是什么都不知道，这样解决 bug 的效率可想而知
可能读到这的你此刻非常想问：怎样才可以更快地修复一个 bug 呢？那么下面我就根据上面讲六个步骤来分别讲解一下对应的技巧。
<hr>

<h2 id="2-解决bug的艺术">2.解决 bug 的艺术</h2>
<blockquote>
在我看来，修复一个 bug 是相对容易的。因为修复一个 bug 的方法可能有很多种，但是如何从本上解决一个 bug，并保证这个 bug 下次不再复现的话，其实是非常难的，这就需要我们学习一下决 bug 的艺术。
</blockquote>
<h3 id="2-1-了解bug">2.1 了解 bug</h3>
<blockquote>
俗话说，知己知彼百战不殆。bug 修复的第一步当然是先了解 bug 了。
</blockquote>
了解 bug 是解决 bug 最重要的一步，它直接决定了后面五步执行的效率和质量。糟糕的错误报和不负责任的问题描述都是埋葬程序员修复 bug 意志的罪魁祸首。
在了解 bug 之前，我们需要收集足够的信息，了解它产生的现象、描述、复现步骤、以及解决的预期是什么等等。那么我们应该怎么做才能更加全面地了解它呢？
下面我给几点建议供大家参考：
<h4 id="2-1-1-观察现象">2.1.1 观察现象</h4>
光凭文字说明是无法准确领悟到 bug 的精髓的。因为每个人思考问题的角度以及文字表达描述是千差万别的。
如果说这个时候能提供一段出错的视频或者问题截图，又或者能够现场演示错误的话，这样观察象，然后再结合问题描述之后，一定能够帮助你快速地了解这个 bug。
<h4 id="2-1-2-询问相关人员">2.1.2 询问相关人员</h4>
这里你询问的可以不仅限于发现 bug 的人（一般是测试人员），当然你首先应当询问的还是这发现 bug 的人。
这里你需要着重询问 bug 报告上你觉得迷惑的点，比如出现 bug 的应用版本、设备型号、bug 现的频率、bug 产生的步骤和恢复的途径、bug 修复的预期效果等。这里你需要进行刨根问底地询，因为可能 bug 发现人觉得一些无关紧要的细节，对你来说却是至关重要的点。
问完发现 bug 的人之后，我们还可以向 bug 对应模块的负责人（测试、开发、产品）询问该模的业务逻辑，说不定能够获取到有价值的信息哦。
<h4 id="2-1-3-提供bug报告模板">2.1.3 提供 bug 报告模板</h4>
一份优秀的 bug 报告模板，可以让程序员直接跳过 bug 修复的前三步，直接进入到 <code>确bug</code> 步骤，从而能够极大地提高 bug 修复的效率。那么一份优秀的 bug 报告模板应当具哪些内容呢？

bug 的标题和问题描述
出现 bug 的应用版本
出现 bug 的设备信息（型号、版本等）
bug 产生相关的视频、截图和错误日志
bug 复现的步骤
bug 出现的必要条件（环境）和恢复途径
bug 修复后的预期效果
bug 对应的模块或者关联 bug

有了以上的内容之后，相信程序员能够很快地了解这个 bug，定位出 bug 产生的原因并予以解。

<h3 id="2-2-复现bug">2.2 复现 bug</h3>
如果你在第一步 <code>了解bug</code> 中获得了良好的 bug 报告的话，则此部分可以很容。你只需要按照 bug 报告中的 bug 复现步骤，按顺序操作即可稳定复现 bug。
当然，很多时候往往并不是一帆风顺的，即使你手握 bug 报告，做了非常详细的调查工作，然而bug 就是无法复现，那么这个时候我们应该怎么做呢？
<h4 id="2-2-1-重新了解bug">2.2.1 重新了解 bug</h4>

此时为了能够复现 bug，你可能需要回到上一步，重现去了解 bug。因为你之前对 bug 的理解能产生偏差，又或者你第一步并没有做好才导致了 bug 未能复现。这个时候带着疑问去重现了解 bu，可能你会发现新的大陆。
<h4 id="2-2-2-将偶现bug转为必现bug">2.2.2 将偶现 bug 转为必现 bug</h4>
偶现 bug 算是 bug 家族中最调皮的一个了。它们以没有规律，难以复现等特性，经常能把以一好好的程序员给逼疯。
但是既然是要复现 bug，那么肯定要找到 bug 稳定复现的路径，这样才能方便下面 bug 的定位这里我推荐大家的一个做法就是想办法把偶现的 bug 转化为必现的 bug。因为即使是偶现的 bug，多也是特定条件下必现的 bug，只不过此时你还没发现这个特定条件而已。
那么怎样才能将偶现 bug 转为必现 bug 呢？这里我简单介绍一下我常用的技巧：

1.<code>对比法</code>：观察并对比复现和不复现的各方面条件，找到那个必现的特定条件
2.<code>注释（删除）代码法</code>：对怀疑的代码进行注释（删除），直到彻底将偶现 bu 转变为必现 bug。

<h3 id="2-3-定位bug">2.3 定位 bug</h3>
<blockquote>
一旦我们找到了 bug 稳定复现的步骤之后，下面的工作就是开始定位 bug 产生的地方了。
</blockquote>
这一步可以说是解决 bug 的关键环节，这一步骤的难易程度一般取决于以下几个因素：

1.程序员自身的代码量（工作经验）
2.对项目代码（业务）的熟悉程度。
3.分析问题和解决问题的能力

那么我们如何才能更快地定位出 bug 产生的位置呢？下面我提供一些思路供大家参考：

1.<code>断点调试</code>。这是程序员通用，同时也是最有效的定位问题的方式。一个不会点调试的程序员和瞎子没有本质上的区别。
2.<code>日志分析</code>。其实并不是所有 bug 都可以进行断点调试的。比如在一些循环用或者业务较为复杂的场景下，打日志分析定位是较为适合的方式。
3.<code>排除法分析</code>。如果一个 bug 产生的原因可能有多种情况的时候，这个时候取排除法的方式是最优的。你可以把可能导致 bug 产生的代码块都打上日志或者断点，然后重现一下 bug 进行问题的定位。
4.<code>代码回滚分析</code>。如果你这个 bug 在之前的版本是好的，但是在现在版本上出现了，这个时候就可以使用代码回滚大法。把你的代码回滚到你怀疑的版本，运行看 bug 是否消，然后对两个版本之前代码有何区别，最终定位出 bug 产生的位置。这里我们可以使用二分法来提代码的回滚效率。
5.<code>注释（删除）代码法</code>。这个我在上一个步骤中也提到过。对于一些难以理解定位的 bug，我们可以使用这个方法进行尝试。不过这个方法使用起来有一定的风险，因为可能你删的那一串代码虽然能够解决 bug，但是却不是 bug 产生的根源，这个时候你可能会将必现 bug 改成偶现 bug，让问题变得更加复杂。
6.<code>源码分析法</code>。有的时候有些 bug 可能并不是你的代码导致的问题。可能是三方库本身的 bug，又或者是系统本身的 bug，又或者是你误用 api 导致的问题，这个时候就需要拥有源码分析的能力。深入源码中，一层层分析直到最终找到 bug 产生的原因。
7.<code>联想法</code>。通过一些类似的 bug 修改经验从而联想猜测出 bug 产生的位置。个方法对使用者本身有较高的要求。需要使用者对项目代码和业务逻辑非常熟悉，同时对问题分析的力有较高的要求。这就是我们常说的牛人能够一眼就能看出问题，他们常用的就是这种方式。
8.<code>场外支援法</code>。这是实在定位不出 bug 才采取的下下策。因为它并不能提高

定位 bug 的能力，同时请别人帮忙定位 bug，你就需要把你之前所做的工作都要全盘地向他表述一，这样不仅会降低 bug 修复的效率，同时还不一定能保证定位出 bug 产生的位置，它取决于你表述题的能力和帮你的人分析和解决问题的能力。

<h3 id="2-4-确认bug">2.4 确认 bug</h3>
<blockquote>
在我们定位出 bug 产生的位置后，下面的工作就是分析 bug 产生的根源了。
</blockquote>
这一步可以说是 bug 修复 6 个步骤中最为关键的一步。这一步直接决定了这个 bug 能否被彻底解决，同时也是最能体现 bug 修复艺术的步骤。
但是很遗憾的是，这一步往往被很多人给忽视了。我为什么会这样说呢？因为很多时候我们修复ug 的时候，都会受到各方面的限制：

1.<code>自身经验水平的限制</code>：一些初入项目的程序员经常因为修复一个 bug 而导了另一个 bug，又或者只是看到了 bug 的表象却不能感知到 bug 产生更深层次的原因，所以 10 种生 bug 的情况他可能只改了一种，将必现问题改成了偶现问题，让 bug 变得更加复杂。
2.<code>时间限制</code>：这是我们程序员经常碰到的限制。这在互联网企业非常普遍，通解决一个 bug 是有时间限制的，例如：这个项目明天就要上线了，并强制要求你今晚就得想办法解。这个时候你可能就被逼无奈，采取硬编码的方式去临时把这个 bug 给按住。其实这样虽然 bug 是临时解决了，但是却会让这个 bug 变得愈加复杂。很多公司出现的那些祖传代码就是在这种情况下生的，动一下就可能产生无数未知的 bug，令人绝望。
3.<code>业务限制</code>：很多时候导致代码逻辑非常复杂难懂的罪魁祸首就是这种业务限。我们在修复一个 bug 的时候，很多时候就是因为这种业务的限制，导致 bug 的修复一种不能从根上予以解决，只要业务一调整就可能导致这个 bug 反复地出现。

说了这么多限制，那么我们如何才能找到问题的根源呢？

1.<code>多积累经验，提升自身的水平</code>：这是打破自身经验水平的限制。
2.<code>及时处理bug</code>：在收到 bug 报告的第一时间就去处理，尽可能打破时间的限。
3.<code>多熟悉业务</code>：有时间就多去了解和梳理业务，深入研究项目代码。这样我们解决 bug 的时候也不会因为对业务不熟悉而无法分析出 bug 产生的根源。
4.<code>准确定位bug</code>：bug 定位的准确性直接决定了你能否分析出 bug 产生的根。因此你需要仔细分析和定位 bug，使用我上面介绍的 8 种方式去定位 bug。

<h3 id="2-5-修复bug">2.5 修复 bug</h3>
其实前面的四步都是为这一步所做的铺垫。有了前面四步的工作，相信到这儿也是相对微不足道了，剩下的就是如何优美地解决这个 bug 了。
到了这个阶段，bug 通常不需要大的修改来修复，因此这一步往往会非常快，当然也就没有什么的技巧啦。
<h3 id="2-6-验证bug">2.6 验证 bug</h3>
<blockquote>
作为 bug 修复的最后一步，它是确保 bug 被真正修复的最后保障。
</blockquote>
在这里需要我们着重注意以下几点：

1.重复之前复现 bug 的步骤来验证 bug 是否被彻底解决。
2.验证 bug 修复可能改动到的相关模块是否正常，保证 bug 修复不引入新的 bug。

如果上述有任何一点没有达到的话，请返回步骤四和步骤五，重新修复 bug！
<hr>

<h2 id="3-如何提高bug修复的效率">3.如何提高 bug 修复的效率</h2>
<blockquote>
上文我们着重讲解了解决 bug 的艺术，为的是能够更好地解决 bug。但是如何才能保证既有效又快速地修复 bug，提高 bug 修复的效率呢？
</blockquote>
通过上面对于解决 bug 的艺术的讲解，我们可以总结出以下影响 bug 修复效率的几个关键点：

1.bug 信息收集的效率以及有效性。
2.时间限制的压力。
3.人员对项目代码（业务）的熟悉程度。
4.人员自身经验和分析问题的能力。

以上 4 点可以说直接决定了 bug 修复的效率。那么如何才能提高 bug 修复的效率呢？下面我将一给出我的看法。

<h3 id="3-1-建立健全的信息收集机制">3.1 建立健全的信息收集机制</h3>
<blockquote>
bug 信息的收集可以说是修复 bug 过程中最为耗时的环节。提升 bug 信息收集的效率以及有效可以大幅度地提升我们修复 bug 的效率。
</blockquote>
那么我们应该如何建立健全的信息收集机制呢？这里我给出我的几点建议：
<h4 id="3-1-1-提供优秀的bug报告模板">3.1.1 提供优秀的 bug 报告模板</h4>
上文我们在 <code>了解bug</code> 一步中提到过：一份优秀的 bug 报告模板，可以让程序直接跳过 bug 修复的前三步，直接进入到 <code>确认bug</code> 步骤，从而能够极大地提高 bu 修复的效率。
这里我再次重复一步，一份优秀的 bug 报告模板应当具备哪些内容：

bug 的标题和问题描述
出现 bug 的应用版本
出现 bug 的设备信息（型号、版本等）
bug 产生相关的视频、截图和错误日志
bug 复现的步骤
bug 出现的必要条件（环境）和恢复途径
bug 修复后的预期效果
bug 对应的模块或者关联 bug

<h4 id="3-1-2-建立完备的日志体系">3.1.2 建立完备的日志体系</h4>
<blockquote>
一套完备的日志体系，可以让我们更加清晰地知道用户到底做了什么才导致 bug 的出现。
</blockquote>
在项目的初期，我们可能为了赶工期而常常忽视了日志打印的重要性，这很可能就会导致该项目

后的维护将非常得艰难。
那么我们为什么要建立一套完备的日志体系呢？

1.并不是所有的用户（测试）都能够给你描述清楚 bug 产生的信息。
2.即使用户（测试）给你描述了 bug 产生相关的信息，但是他们并不理解你的代码逻辑，他们只根据 bug 出现的现象告诉你问题，可能他们的表述和你的理解不在一个频道。

如果你有这么一套完备的日志体系，那么你就可以在用户（测试）不用开口的情况下，直接 get 用户的操作行为以及对于的代码逻辑。同时，可能一句关键点的报错日志就可以帮你直接定位到 bug 产生的位置，从而直接进入第四步 <code>确认bug</code>。
说了这么多，我们应该如何打印高效的日志呢？

1.在异常分支返回前打印日志。
2.在复杂业务流程的关键点打印日志。
3.在对外交互或者模块交互点打印日志。
4.在用户交互或者生命周期的关键点打印日志。
5.对重要的信息点打印日志，记录用户画像。
6.按重要性分等级打印日志。
7.禁止在循环中打印日志，禁止打印无效的日志。
8.禁止打印用户隐私相关的信息。

<h3 id="3-2-建立自动化测试机制">3.2 建立自动化测试机制</h3>
<blockquote>
建立自动化测试机制,可以让突破 <code>时间限制</code> 成为可能。
</blockquote>
<h4 id="3-2-1-更早地发现bug">3.2.1 更早地发现 bug</h4>
很多时候来自时间限制的压力，往往是测试不充分导致的。很多 bug 直到产品临近上线或者交的时候才被发现，这个时候唯一解决问题的方式就是在时间上做出限制，无情压迫我们这些活在公司力最底层的程序员们。
如果这个时候能有一套自动化测试机制，每天下班后都进行自动测试的话，那样很多 bug 就能提前发现，从而为我们修复 bug 预留了不少宝贵的时间。
<h4 id="3-2-2-节约bug验证的时间">3.2.2 节约 bug 验证的时间</h4>
对于一些复杂难解、偶现的 bug，我们往往会在 <code>确认bug</code> 到 <code>验证bu</code> 之间花费大量的时间。这个时候如果有一套自动化测试机制或者工具帮助我们验证 bug 的，就可以极大地缩减我们修复 bug 的时间。

<h3 id="3-3-提高项目代码-业务-的熟悉程度">3.3 提高项目代码（业务）的熟悉程度</h3>
<blockquote>
提高人员对项目代码（业务）的熟悉程度，这样就可以极大地提高人员 <code>定位bug</code> 的效率。
</blockquote>
<h4 id="3-3-1-建立丰富的知识库体系">3.3.1 建立丰富的知识库体系</h4>
<blockquote>
建立一套丰富的知识库体系，可以帮助我们加深对自己责任内项目代码（业务）的理解，同时还帮助我们快速了解我们所不了解的业务模块。
</blockquote>
那么如何才能建立起丰富的知识库体系呢？下面我给出我的几点建议：

1.对知识进行分类。
2.定期添加和更新知识库的内容。
3.提高知识库的检索效率。
4.定期组织知识的分享。

5.激励贡献知识库的人员。

<h4 id="3-3-2-建立责任田划分机制">3.3.2 建立责任田划分机制</h4>
<blockquote>
划分责任田的目的就是：让专业的人做专业的事情。
</blockquote>
划分责任田的好处：

1.专业的人做专业的事情。划分完责任田后，田主需要对自己负责的模块负责，这就必然要求其模块内的代码（业务）更加熟悉。
2.责任到人利于追责和 bug 跟踪。

当然责任田也不是想象中的那么完美，它也存在一定的缺陷：

1.职责明确之后可能导致缺少全局视野。一些复杂的 bug 可能是几个模块共同作用下才产生的，于这类 bug 的定位势必会大大增加难度。
2.划分责任田之后，可能导致踢皮球的情况。

责任田划分机制，它是一把双刃剑。所以是否需要建立责任田划分机制，还是需要结合企业自身情况而定的。
<h3 id="3-4-提高人员的综合素质">3.4 提高人员的综合素质</h3>
<blockquote>
提高人员的综合素质，可以帮助我们提高 <code>定位bug</code>、<code>确认bug</code> 以及 <code>修复bug</code> 三个步骤下的效率。
</blockquote>

1.<code>建立公平的员工晋升制度</code>。这样可以充分调动人员的主动性的积极性，提升员的个人素质和业务能力。
2.<code>建立岗位轮换制度</code>。让人员定期负责不同的模块，可以极大地提升人员的综素质和全局视野。
3.<code>定期组织培训学习</code>。

<hr>
<h2 id="4-最后">4.最后</h2>
以上内容，是我参与工作五年，开源六年以来所总结下来的全部经验，喜欢的可以点击收藏或者连支持一下！最后，还是祝福大家从此代码无 bug，哈哈哈！！！
<h2 id="5-微信公众号">5.微信公众号</h2>
<blockquote>
更多资讯内容，欢迎搜索我的微信公众号:【我的 Android 开源之旅】
</blockquote>