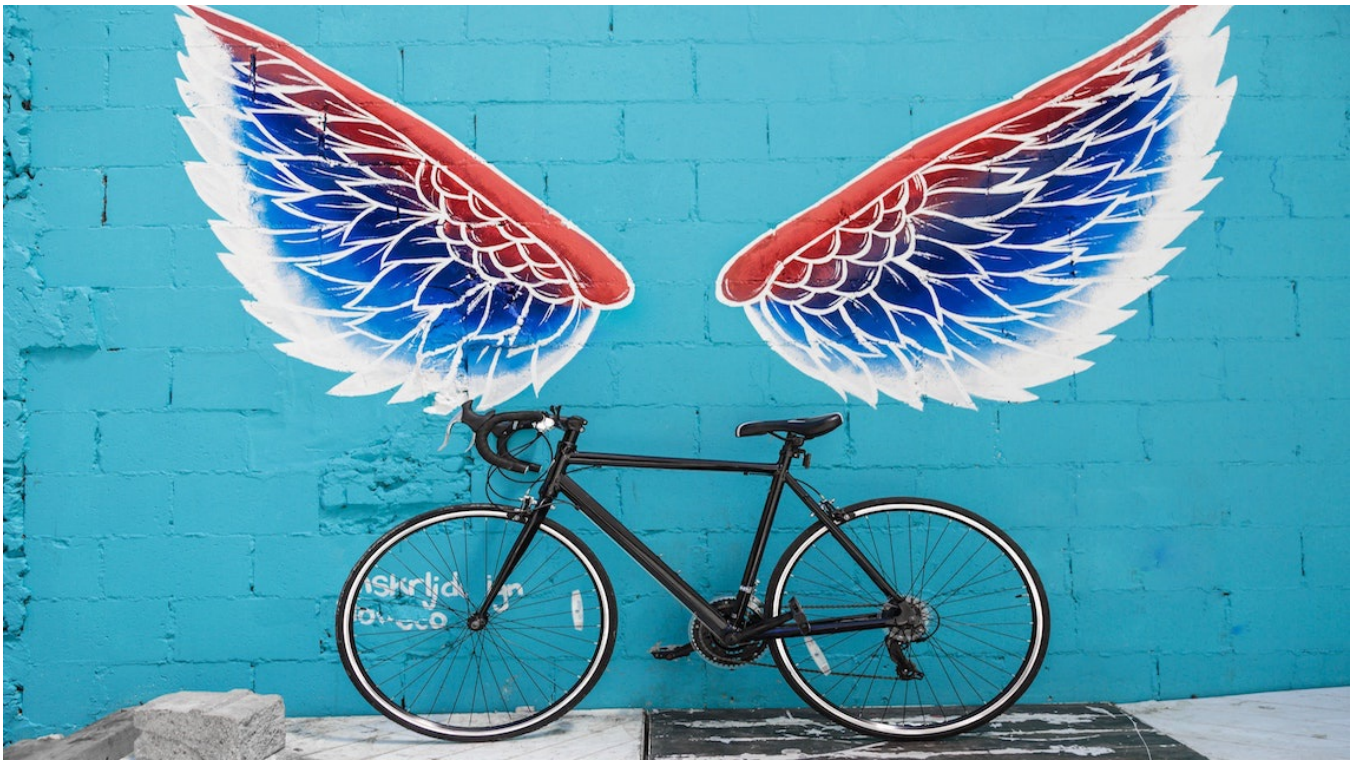
14 | Bug的反复出现：重蹈覆辙与吸取教训

2018-09-03 胡峰



Bug 除了时间和空间两种属性，还有一个特点是和程序员直接相关的。在编程的路上，想必你也曾犯过一些形态各异、但本质重复的错误，导致一些 Bug 总是以不同的形态反复出现。在你捶胸顿足懊恼之时，不妨试着反思一下：为什么你总会写出有 Bug 的程序，而且有些同类型的 Bug 还会反复出现？

1. 重蹈覆辙

重蹈覆辙的错误，老实说曾经我经历过不止一次。

也许每次具体的形态可能有些差异，但仔细究其本质却是类似的。想要写出没有 Bug 的程序是不可能的，因为所有的程序员都受到自身能力水平的局限。而我所经历的重蹈覆辙型错误，总结下来大概都可以归为以下三类原因。

1.1 粗心大意

人人都会犯粗心大意的错误，因为这就是“人”这个系统的普遍固有缺陷（Bug）之一。所以，作为人的程序员一定会犯一些非常低级的、因为粗心大意而导致的 Bug。

这就好比写文章、写书都会有错别字，即使经历过三审三校后正式出版的书籍，都无法完全避免错别字的存在。

而程序中也有这类“错别字”类型的低级错误，比如：条件`if`后面没有大括号导致的语义变化，==、= 和 === 的数量差别，++ 或-- 的位置，甚至 ; 的有无在某些编程语言中带来的语义

差别。即使通过反复检查也可能有遗漏，而自己检查自己的代码会更难发现这些缺陷，这和自己不容易发现自己的错别字是一个道理。

心理学家汤姆·斯塔福德（Tom Stafford）曾在英国谢菲尔德大学研究拼写错误，他说：“当你在书写的时候，你试图传达想法，这是非常高级的任务。而在做高级任务时，大脑将简单、零碎的部分（拼词和造句）概化，这样就可以更专注于更复杂的任务，比如将句子变成复杂的观点。”

而在阅读时，他解释说：“我们不会抓住每个细节，相反，我们吸收感官信息，将感觉和期望融合，并且从中提炼意思。”这样，如果我们读的是他人的作品，就能帮助我们用更少的脑力更快地理解含义。

但当我们验证自己的文章时，我们知道想表达的东西是什么。因为我们预期这些含义都存在，所以很容易忽略掉某些感官（视觉）表达上的缺失。我们眼睛看到的，在与我们脑子里的印象交战。这，便是我们对自己的错误视而不见的原因。

写程序时，我们是在进行一项高级的复杂任务：将复杂的需求或产品逻辑翻译为程序逻辑，并且还要补充上程序固有的非业务类控制逻辑。因而，一旦我们完成了程序，再来复审写好的代码，这时我们预期的逻辑含义都预先存在于脑中，同样也就容易忽略掉某些视觉感官表达上的问题。

从进化角度看，粗心写错别字，还看不出来，不是因为我们太笨，而恰恰还是进化上的权衡优化选择。

1.2 认知偏差

认知偏差，是重蹈覆辙类错误的最大来源。

曾经，我就对 Java 类库中的线程 API 产生过认知偏差，导致反复出现问题。Java 自带线程池有三个重要参数：核心线程数（core）、最大线程数（max）和队列长度（queues）。我曾想当然地以为当核心线程数（core）不够了，就会继续创建线程达到最大线程数（max），此时如果还有任务需要处理但已经没有线程了就会放进队列等待。

但实际却不是这样工作的，类库的实现是核心线程（core）满了就会进队列（queues）等待，直到队列也满了再创建新线程直至达到最大线程数（max）的限制。这类认知偏差曾带来线上系统的偶然性异常故障，然后还怎么都找不到原因。因为这进入了我的认知盲区，我以为的和真正的现象之间的差异一度让我困惑不解。

还有一个来自生活中的小例子，虽然不是关于程序的，但本质是一个性质。

有时互联网上，朋友圈中小道消息满天飞，与此类现象有关的一个成语叫“空穴来风”，现在很多媒体文章有好多是像下面这样用这个成语的：

> 他俩要离婚了？看来空穴来风，事出有因啊！
> 物价上涨的传闻恐怕不是空穴来风。

第一句是用的成语原意：指有根据、有来由，“空”发三声读 kǒng，意同“孔”。第二句是表达：没有根据和由来，“空”发一声读kōng。第二种的新意很多名作者和普通大众沿用已久，约定俗成，所以又有辞书与时俱进增加了这个新的义项，允许这两种完全相反的解释并存，自然发展，这在语义学史上也不多见。

而关于程序上有些 API 的定义和实现也犯过“空穴来风”的问题，一个 API 可以表达两种完全相反的含义和行为。不过这样的 API 就很容易引发认知偏差导致的 Bug，所以在设计和实现 API 时我们就要避免这种情况的出现，而是要提供单一原子化的设计。

1.3 熵增问题

熵增，是借用了物理热力学的比喻，表达更复杂混乱的现象；程序规模变大，复杂度变高之后，再去修改程序或添加功能就更容易引发未知的 Bug。

腾讯曾经分享过 QQ 的架构演进变化，到了 3.5 版本 QQ 的用户在线规模进入亿时代，此时在原有架构下去新增一些功能，比如：

> “昵称”长度增加一半，需要两个月；
>
> 增加“故乡”字段，需要两个月；
>
> 最大好友数从 500 变成 1000，需要三个月。

后端系统的高度复杂性和耦合作用导致即使增加一些小功能特性，也可能带来巨大的牵连影响，所以一个小改动才需要数月时间。

我们不断进行架构升级的本质，就在于随着业务和场景功能的增加，去控制住程序系统整体“熵”的增加。而复杂且耦合度高（熵很高）的系统，正是容易滋生 Bug 的温床。

2. 吸取教训

为了避免重蹈覆辙，我们有什么办法来吸取曾经犯错的教训么？

2.1 优化方法

粗心大意，可以通过开发规范、代码风格、流程约束，代码评审和工具检查等工程手段来加以避免。甚至相对写错别字，代码更进一步，通过补充单元测试在运行时做一个正确性后验，反过来去发现这类我们视而不见的低级错误。

认知偏差，一般没什么太好的自我发现机制，但可以依赖团队和技术手段来纠偏。每次掉坑里爬出来后的经验教训总结和团队内部分享，另外就是像一些静态代码扫描工具也提供了内置的优化实践，通过它们的提示来发现与你的认知产生碰撞纠偏。

熵增问题，业界不断迭代更新流行的架构模式就是在解决这个问题。比如，微服务架构相对曾经的单体应用架构模式，就是通过增加开发协作，部署测试和运维上的复杂度来换取系统开发的敏

捷性。在协作方式、部署运维等方面付出的代价都可以通过提升自动化水平来降低成本，但只有编程活动是没法自动化的，依赖程序员来完成，而每个程序员对复杂度的驾驭能力是有不同上限的。

所以，微服务本质上就是将一个大系统的熵增问题，局部化在一个又一个的小服务中。而每个微服务都有一个熵增的极限值，而这个极限值一般是要低于该服务负责人的驾驭能力上限的。对于一个熵增接近极限附近的微服务，服务负责人就需要及时重构优化，降低熵的水平。而高水平和低水平程序员负责的服务本质差别在于熵的大小。

而熵增问题若不及时重构优化，最后可能会付出巨大的代价。

丰田曾陷入的“刹车门”事件，就是因为其汽车动力控制系统软件存在缺陷。而为追查其原因，在十八个月中，有 12 位嵌入式系统专家受原告诉讼团所托，被关在马里兰州一间高度保安的房间内对丰田动力控制系统软件（主要是 2005 年的凯美瑞）源代码进行深度审查。最后得到的结论把丰田的软件缺陷分为三类：

- 非常业余的结构设计
- 不符合软件开发规范
- 对关键变量缺乏保护

第一类属于熵增问题，导致系统规模不断变大、变复杂，结果驾驭不了而失控；第二类属于开发过程的认知与管理问题；第三类才是程序员实现上的水平与粗心大意问题。

2.2 塑造环境

为了修正真正的错误，而不是头痛医头、脚痛医脚，我们需要更深刻地认识问题的本质，再来开出“处方单”。

在亚马逊（Amazon），严重的故障需要写一个 COE（Correction of Errors）的文档，这是一种帮助去总结经验教训，加深印象避免再犯的形式。其目的也是为了帮助认识问题的本质，修正真正的错误。

但一旦这个东西和 KPI 之类的挂上钩，引起的负面作用是 COE 的数量会变少，但真正的问题并没有减少，只是被隐藏了。而其正面的效应像总结经验、吸取教训、找出真正问题等，就会被大大削弱。

关于如何构造一个鼓励修正错误的环境，我们可以看看来自《异类》一书讲述的大韩航空的例子，大韩航空曾一度困扰于它的飞机损失率：

> 美国联合航空 1988 年到 1998 年的飞机损失率为百万分之 0.27，也就是说联合航空每飞行 400 万次，会在一次事故中损失一架飞机；而大韩航空同期的飞机损失率为百万分之 4.79，是前者的 17 倍之多。

事实上大韩航空的飞机也是买自美国，和联合航空并无多大差别。它的飞行员们的飞行时长，经验和训练水平从统计数据看也差别不大，那为什么飞机损失率会如此地高于其他航空公司的平均水平呢？在《异类》这本书中，作者以此为案例做了详细分析，我这里直接引用结论。

> 现代商业客机，就目前发展水平而言，跟家用烤面包机一样可靠。空难很多时候是一系列人为的小失误、机械的小故障累加的结果，一个典型空难通常包括 7 个人为的错误。

一个飞机上有正副两个机长，副机长的作用是帮助发现、提醒和纠正机长在飞行过程中可能发生的一些人为小错误。大韩航空的问题正在于副机长是否敢于以及如何提醒纠正机长的错误。其背后的理论依据源自荷兰心理学家吉尔特·霍夫斯泰德（Geert Hofstede）对不同族裔之间文化差异的研究，就是今天被社会广泛接受的跨文化心理学经典理论框架：霍夫斯泰德文化纬度（Hofstede's Dimensions）。

> 在霍夫斯泰德的几个文化维度中，最引人注目的大概就是“权力距离指数（Power Distance Index）”。权力距离是指人们对待比自己更高等级阶层的态度，特别是指对权威的重视和尊重程度。

> 而霍夫斯泰德的研究也提出了一个航空界专家从未想到过的问题：让副机长在机长面前维护自己的意见，必须帮助他们克服所处文化的权力距离。

想想我们看过的韩国电影或电视剧中，职场上后辈对前辈、下级对上级的态度，就能感知到韩国文化相比美国所崇尚的自由精神所表现出来的权力距离是特别远的。因而造成了大韩航空未被纠正的人为小错误比例更高，最终的影响是空难率也更高，而空难就是航空界的终极系统故障，而且结果不可挽回。

吸取大韩航空的教训应用到软件系统开发和维护上，就是：需要**建立和维护有利于程序员及时暴露并修正错误，挑战权威和主动改善系统的低权力距离文化氛围，这其实就是推崇扁平化管理和“工程师文化”的关键所在**。

一旦系统出了故障非技术背景的管理者通常喜欢用流程、制度甚至价值观来应对问题，而技术背景的管理者则喜欢从技术本身的角度去解决当下的问题。我觉着两者需要结合，站在更高的维度去考虑问题：**规则、流程或评价体系的制定所造成的文化氛围，对于错误是否以及何时被暴露，如何被修正有着决定性的影响**。

我们常与错误相伴，查理·芒格说：

> 世界上不存在不犯错误的学习或行事方式，只是我们可以通过学习，比其他人少犯一些错误，也能够在犯了错误之后，更快地纠正错误。但既要过上富足的生活又不犯很多错误是不可能的。实际上，生活之所以如此，是为了让你们能够处理错误。

人固有缺陷，程序固有 Bug；吸取教训避免重蹈覆辙，除了不断提升方法，也要创造环境。你觉

得呢？欢迎你留言和我分享。