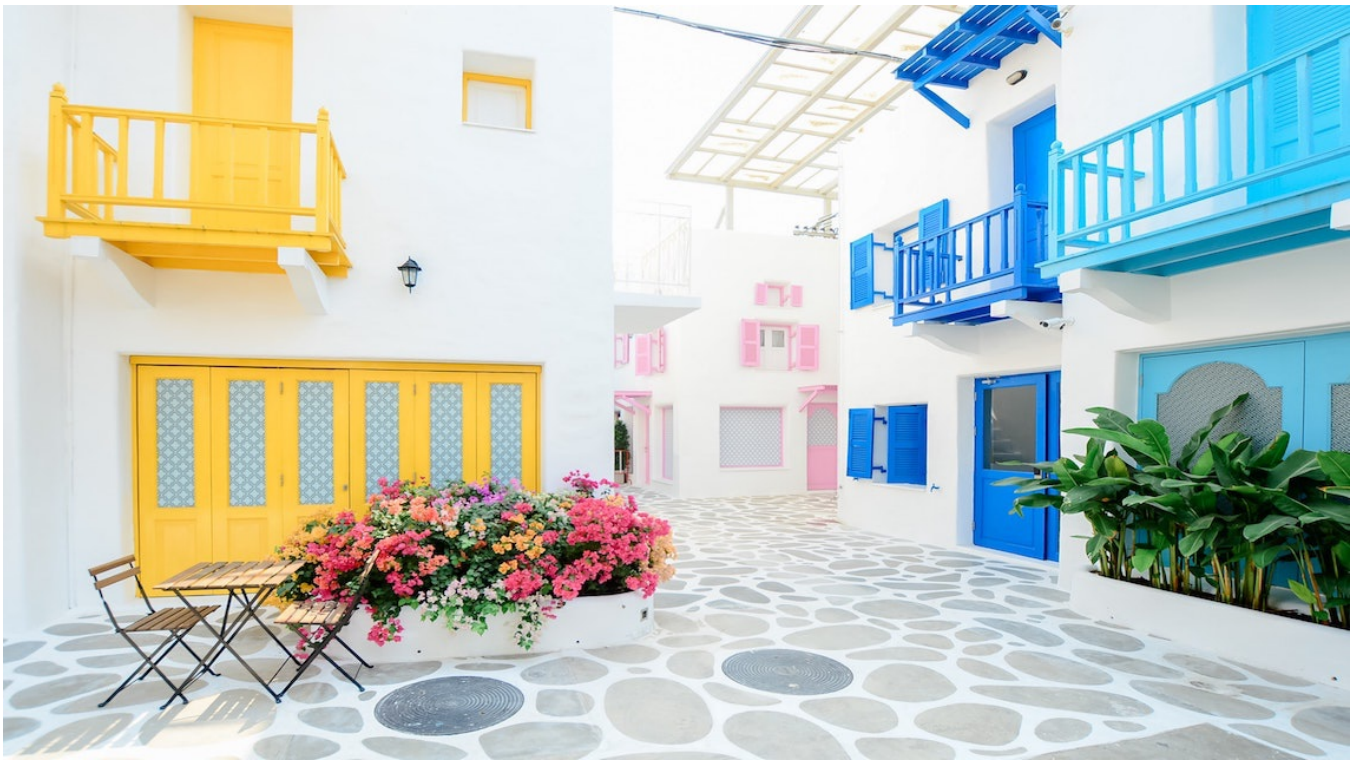
12 | Bug的空间属性：环境依赖与过敏反应

2018-08-29 胡峰



从今天开始，咱们专栏进入“程序之术”中关于写代码的一个你可能非常熟悉，却也常苦恼的小主题：Bug。

写程序的路上，会有一个长期伴随你的“同伴”：Bug，它就像程序里的寄生虫。不过，Bug 最早真的是一只虫子。

1947年，哈佛大学的计算机哈佛二代（Harvard Mark II）突然停止了运行，程序员在电路板编号为 70 的中继器触点旁发现了一只飞蛾。然后把飞蛾贴在了计算机维护日志上，并写下了首个发现 Bug 的实际案例。程序错误从此被称作 Bug。

这只飞蛾也就成了人类历史上的第一个程序 Bug。

回想下，在编程路上你遇到得最多的 Bug 是哪类？我的个人感受是，经常被测试或产品经理要求修改和返工的 Bug。这类 Bug 都来自于对需求理解的误差，其实属于沟通理解问题，我并不将其归类为真正的技术性 Bug。

技术性 Bug 可以从很多维度分类，而我则习惯于从 Bug 出现的“时空”特征角度来分类。可划为如下两类：

- 空间：环境过敏
- 时间：周期规律

我们就先看看 Bug 的**空间维度**特征。

环境过敏

环境，即程序运行时的空间与依赖。

程序运行的依赖环境是很复杂的，而且一般没那么可靠，总是可能出现这样或那样的问题。曾经我经历过一次因为运行环境导致的故障案例：一开始系统异常表现出来的现象是，有个功能出现时不时的不可用；不久之后，系统开始报警，不停地接到系统的报警短信。

这是一个大规模部署的线上分布式系统，从一开始能感知到的个别系统功能异常到逐渐演变成大面积的报警和业务异常，这让我们陷入了一个困境：到底异常根源在哪里？为了迅速恢复系统功能的可用性，我们先把线上流量切到备用集群后，开始紧急地动员全体团队成员各自排查其负责的子系统和服务，终于找到了原因。

只是因为有个别服务器容器的磁盘故障，导致写日志阻塞，进程挂起，然后引发调用链路处理上的连锁雪崩效应，其影响效果就是整个链路上的系统都在报警。

互联网企业多采用普通的 PC Server 作为服务器，而这类服务器的可靠性大约在 99.9%，换言之就是出故障的概率是千分之一。而实际在服务器上，出问题概率最高的可能就是其机械硬盘。

Backblaze 2014 年发布的硬盘统计报告指出，根据对其数据中心 38000 块硬盘（共存储 100PB 数据）的统计，消费级硬盘头三年出故障的几率是 15%。而在一个足够大规模的分布式集群部署上，比如 Google 这种百万级服务器规模的部署级别上，几乎每时每刻都有硬盘故障发生。

我们的部署规模自是没有 Google 那么大，但也不算小了，运气不好，正好赶上我们的系统碰上磁盘故障，而程序的编写又并未考虑硬盘 I/O 阻塞导致的挂起异常问题，引发了连锁效应。

这就是当时程序编写缺乏对环境问题的考虑，引发了故障。人有时换了环境，会产生一些从生理到心理的过敏反应，程序亦然。运行环境发生变化，程序就出现异常的现象，我称其为“程序过敏反应”。

以前看过一部美剧《豪斯医生》，有一集是这样的：一个手上出现红色疱疹的病人来到豪斯医生的医院，豪斯医生根据病症现象初步诊断为对某种肥皂产生了过敏，然后开了片抗过敏药，吃过后疱疹症状就减轻了。但一会儿后，病人开始出现呼吸困难兼并发哮喘，豪斯医生立刻给病人注射了 1cc 肾上腺素，之后病人呼吸开始变得平稳。但不久后病人又出现心动过速，而且很快心跳便停止了，经过一番抢救后，最终又回到原点，病人手上的红色疱疹开始在全身出现。

这个剧情中表现了在治疗病人时发生的身体过敏反应，然后引发了连锁效应的问题，这和我之前描述的例子有相通之处：都是局部的小问题，引发程序过敏反应，再到连锁效应。

过敏在医学上的解释是：“有机体将正常无害的物质误认为是有害的东西。”而我对“程序过敏反

应”的定义是：“程序将存在问题的环境当作正常处理，从而产生的异常。”而潜在的环境问题通常就成了程序的“过敏原”。

该如何应对这样的环境过敏引发的 Bug 呢？

应对之道

应对环境过敏，自然要先从**了解环境**开始。

不同的程序部署和运行的环境千差万别，有的受控，有的不受控。比如，服务端运行的环境，一般都在数据中心（IDC）机房内网中，相对受控；而客户端运行的环境是在用户的设备上，存在不同的品牌、不同的操作系统、不同的浏览器等等，多种多样，不可控。

环境那么复杂，你需要了解到何种程度呢？我觉得你至少必须关心与程序运行直接相关联的那一层环境。怎么理解呢？以后端 Java 程序的运行为例，Java 是运行在 JVM 中，那么 JVM 提供的运行时配置和特性就是你必须要关心的一层环境了。而 JVM 可能是运行在 Linux 操作系统或者是像 Docker 这样的虚拟化容器中，那么 Linux 或 Docker 这一层，理论上你的关心程度就没太多要求，当然，学有余力去了解到这一层次，自是更好的。

那么前文案例中的磁盘故障，已经到了硬件的层面，这个环境层次比操作系统还更低一层，这也属于我们该关心的？虽说故障的根源是磁盘故障，但直接连接程序运行的那一层，其实是日志库依赖的 I/O 特性，这才是我们团队应该关心、但实际却被忽略掉的部分。

同理，现今从互联网到移动互联网时代，几乎所有的程序系统都和网络有关，所以网络环境也必须是你关心的。但网络本身也有很多层次，而对于在网络上面开发应用程序的你我来说，可以把网络模糊抽象为一个层次，只用关心网络距离延时，以及应用程序依赖的具体平台相关网络库的 I/O 特性。

当然，如果能对网络的具体层次有更深刻的理解，自然也是更好的。事实上，如果你和一个对网络具体层次缺乏理解的人调试两端的网络程序，碰到问题时，经常会发现沟通不在一个层面上，产生理解困难。（这里推荐下隔壁的“趣谈网络协议”专栏）

了解了环境，也难免不出 Bug。因为我们对环境的理解是渐进式的，不可能一下子就完整掌握，全方位，无死角。当出现了因为环境产生的过敏反应时，收集足够多相关的信息才能帮助快速定位和解决问题，这就是前面《代码与分类》文章中“运维”类代码需要提供的服务。

收集信息，不仅仅局限于相关直接依赖环境的配置和参数，也包括用户输入的一些数据。真实场景确实大量存在这样一种情况：同样的环境只针对个别用户发生异常过敏反应。

有一种药叫抗过敏药，那么也可以有一种代码叫“抗过敏代码”。在收集了足够的信息后，你才能编写这样的代码，因为现实中，程序最终会运行在一些一开始你可能没考虑到的环境中。收集到了这样的环境信息，你才能写出针对这种环境的“抗过敏代码”。

这样的场景针对客户端编程特别常见，比如客户端针对运行环境进行的自检测和自适应代码。检测和适应范围包括：CPU、网络、存储、屏幕、操作系统、权限、安全等各方面，这些都属于环境抗过敏类代码。

而服务端相对环境一致性更好，可控，但面临的环境复杂性更多体现在“三高”要求，即：高可用、高性能、高扩展。针对“三高”的要求，服务端程序生产运行环境的可靠性并不如你想象的高，虽然平时的开发、调试中你可能很难遇到这些环境故障，但大规模的分布式程序系统，面向失败设计和编码（Design For Failure）则是服务端的“抗过敏代码”了。

整体简单总结一下就是：空间即环境，包括了程序的运行和依赖环境；环境是多维度、多层次的，你对环境的理解越全面、越深入，那么出现空间类 Bug 的几率也就越低；对环境的掌控有广度和深度两个方向，更有效的方法是先广度全面了解，再同步与程序直接相连的一层去深度理解，最后逐层深入，“各个击破”。

文章开头的第一只飞蛾 Bug，按我的分类就应该属于空间类 Bug 了，空间类 Bug 感觉麻烦，但若单独出现时，相对有形（异常现场容易捕捉）；如果加上时间的属性，就变得微妙多了。

下一篇，我将继续为你分解 Bug 的时间维度特征。