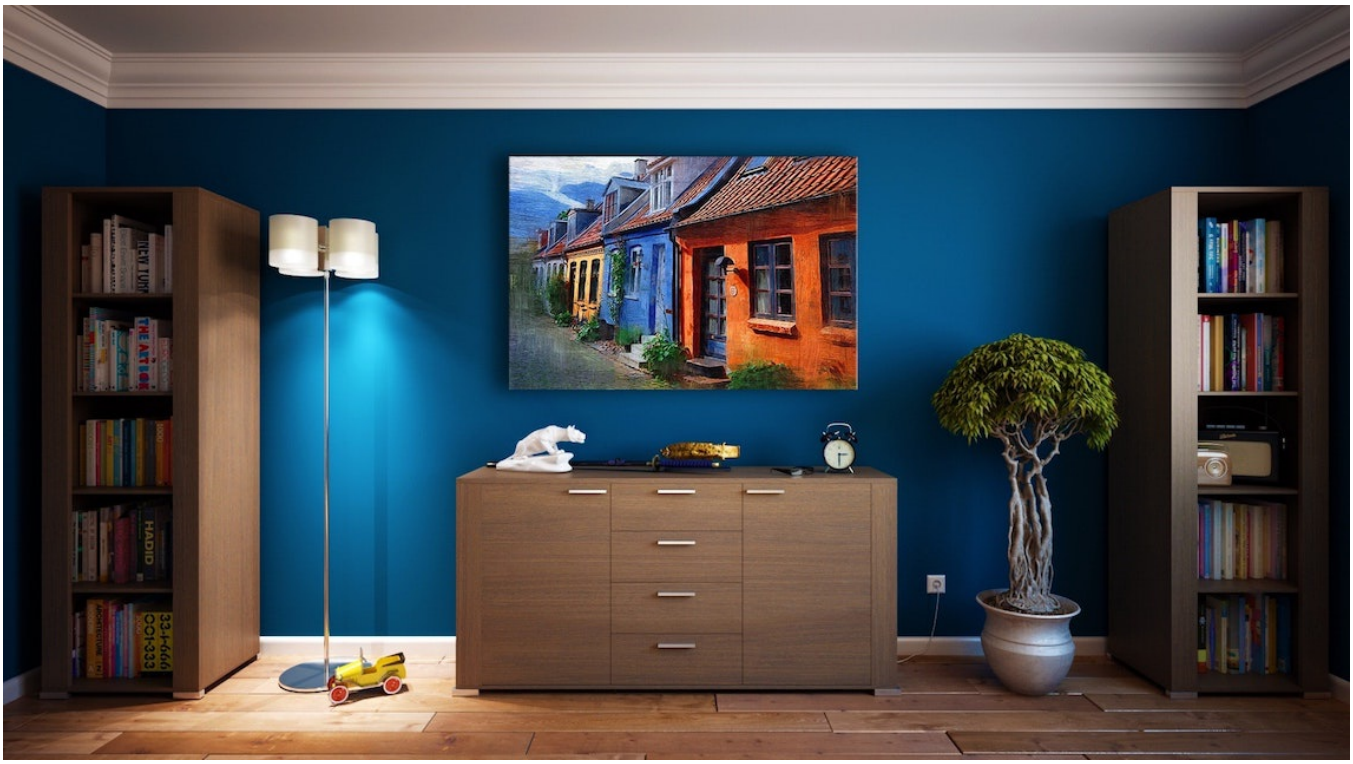
13 | Bug的时间属性：周期特点与非规律性

2018-08-31 胡峰



在上一篇文章中，我说明了“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”。并且我也已讲解了Bug 的**空间维度**特征：程序对运行环境的依赖、反应及应对。

接下来我再继续分解 Bug 的**时间维度**特征。

Bug 有了时间属性，Bug 的出现就是一个概率性问题了，它体现出如下特征。

周期特点

周期特点，是一定频率出现的 Bug 的特征。

这类 Bug 因为会周期性地复现，相对还是容易捕捉和解决。比较典型的呈现此类特征的 Bug 一般是资源泄漏问题。比如，Java 程序员都不陌生的 `OutOfMemory` 错误，就属于内存泄漏问题，而且一定会周期性地出现。

好多年前，我才刚参加工作不久，就碰到这么一个周期性出现的 Bug。但它的特殊之处在于，出现 Bug 的程序已经稳定运行了十多年了，突然某天开始就崩溃（进程 Crash）了。而程序的原作者，早已不知去向，十多年下来想必也已换了好几代程序员来维护了。

一开始项目组内经验老到的高工认为也许这只是一个意外事件，毕竟这个程序已经稳定运行了十来年了，而且检查了一遍程序编译后的二进制文件，更新时间都还停留在那遥远的十多年前。所

以，我们先把程序重启起来让业务恢复，重启后的程序又恢复了平稳运行，但只是安稳了这么一天，第二天上班没多久，进程又莫名地崩溃了，我们再次重启，但没多久后就又崩溃了。这下没人再怀疑这是意外了，肯定有 Bug。

当时想想能找出一个隐藏了这么多年的 Bug，还挺让人兴奋的，就好像发现了埋藏在地下久远的宝藏。

寻找这个 Bug 的过程有点像《盗墓笔记》中描述的盗墓过程：项目经理（三叔）带着两个高级工程师（小哥和胖子）连续奋战了好几天，而我则是个新手，主要负责“看门”，在他们潜入跟踪分析探索的过程中，我就盯着那个随时有可能崩溃的进程，一崩掉就重启。他们“埋伏”在那里，系统崩溃后抓住现场，定位到对应的源代码处，最后终于找到了原因并顺利修复。

依稀记得，最后定位到的原因与网络连接数有关，也是属于资源泄漏的一种，只是因为过去十来年交易量一直不大且稳定，所以没有显现出来。但在我参加工作那年（2006年），中国股市悄然引来一场有史以来最大的牛市，这个处理银行和证券公司之间资金进出的程序的“工作量”突然出现了爆发性增长，从而引发了该 Bug。

我可以理解上世纪九十年代初那个编写该服务进程的程序员，他可能也难以预料到当初写的用者寥寥的程序，最终在十多年后的一天会服务于成百上千万的用户。

周期性的 Bug，虽然乍一看很难解决的样子，但它总会重复出现，就像可以重新倒带的“案发现场”，找到真凶也就简单了。案例中这个 Bug 隐藏的时间很长，但它所暴露出的周期特点很明显，解决起来也就没那么困难。

其实主要麻烦的是那种这次出现了，但不知道下次会在什么时候出现的 Bug。

非规律性

没有规律性的 Bug，才是让人抓狂的。

曾经我接手过一个系统，是一个典型的生产者、消费者模型系统。系统接过来就发现一个比较明显的性能瓶颈问题，生产者的数据源来自数据库，生产者按规则提取数据，经过系统产生一系列的转换渲染后发送到多个外部系统。这里的瓶颈就在数据库上，生产能力不足，从而导致消费者饥饿。

问题比较明显，我们先优化 SQL，但效果不佳，遂改造设计实现，在数据库和系统之间增加一个内存缓冲区从而缓解了数据库的负载压力。缓冲区的效果，类似大河之上的堤坝，旱时积水，涝时泄洪。引入缓冲区后，生产者的生产能力得到了有效保障，生产能力高效且稳定。

本以为至此解决了该系统的瓶颈问题，但在生产环境运行了一段时间后，系统表现为速度时快时慢，这时真正的 Bug 才显形了。

这个系统有个特点，就是 I/O 密集型。消费者要与多达 30 个外部系统并发通信，所以猜测极有

可能导致系统性能不稳定的 Bug 就在此，于是我把目光锁定在了消费者与外部系统的 I/O 通信上。既然锁定了怀疑区域，接下来就该用证据来证明，并给出合理的解释原因了。一开始假设在某些情况下触碰到了阈值极限，当达到临界点时程序性能则急剧下降，不过这还停留在怀疑假设阶段，接下来必须量化验证这个推测。

那时的生产环境不太方便直接验证测试，我便在测试环境模拟。用一台主机模拟外部系统，一台主机模拟消费者。模拟主机上的线程池配置等参数完全保持和生产环境一致，以模仿一致的并发数。通过不断改变通信数据包的大小，发现在数据包接近 100k 大小时，两台主机之间直连的千兆网络 I/O 达到满负载。

于是，再回头去观察生产环境的运行状况，当一出现性能突然急剧下降的情况时，立刻分析了生产者的数据来源。其中果然有不少大报文数据，有些甚至高达 200k，至此基本确定了与外部系统的 I/O 通信瓶颈。解决办法是增加了数据压缩功能，以牺牲 CPU 换取 I/O。

增加了压缩功能重新上线后，问题却依然存在，系统性能仍然时不时地急剧降低，而且这个时不时很没有时间规律，但关联上了一个“嫌疑犯”：它的出现和大报文数据有关，这样复现起来就容易多了。I/O 瓶颈的怀疑被证伪后，只好对程序执行路径增加了大量跟踪调试诊断代码，包含了每个步骤的时间度量。

在完整的程序执行路径中，每个步骤的代码块的执行时间独立求和结果仅有几十毫秒，最高也就在一百毫秒左右，但多线程执行该路径的汇总平均时间达到了 4.5 秒，这比我预期值整整高了两个量级。通过这两个时间度量的巨大差异，我意识到线程执行该代码路径的时间其实并不长，但花在等待 CPU 调度的时间似乎很长。

那么是 CPU 达到了瓶颈么？通过观察服务器的 CPU 消耗，平均负载却不高。只好再次分析代码实现机制，终于在数据转换渲染子程序中找到了一段可疑的代码实现。为了验证疑点，再次做了一下实验测试：用 150k 的线上数据报文作为该程序输入，单线程运行了下，发现耗时居然接近 50 毫秒，我意识到这可能是整个代码路径中最耗时的一个代码片段。

由于这个子程序来自上上代程序员的遗留代码，包含一些稀奇古怪且复杂的渲染逻辑判断和业务规则，很久没人动过了。仔细分析了其中实现，基本就是大量的文本匹配和替换，还包含一些加密、Hash 操作，这明显是一个 CPU 密集型的函数啊。那么在多线程环境下，运行这个函数大概平均每个线程需要多少时间呢？

先从理论上来分析下，我们的服务器是 4 核，设置了 64 个线程，那么理想情况下同一时间可以运行 4 个线程，而每个线程执行该函数约为 50 毫秒。这里我们假设 CPU 50 毫秒才进行线程上下文切换，那么这个调度模型就被简化了。第一组 4 个线程会立刻执行，第二组 4 个线程会等待 50 毫秒，第三组会等待 100 毫秒，依此类推，第 16 组线程执行时会等待 750 毫秒。平均下来，每组线程执行前的平均等待时间应该是在 300 到 350 毫秒之间。这只是一个理论值，实际运行测试结果，平均每个线程花费了 2.6 秒左右。

实际值比理论值慢一个量级，这是为什么呢？因为上面理论的调度模型简化了 CPU 的调度机制，在线程执行过程的 50 毫秒中，CPU 将发生非常多次的线程上下文切换。50 毫秒对于 CPU 的时间分片来说，实在是太长了，因为线程上下文的多次切换和 CPU 争夺带来了额外的开销，导致在生产环境上，实际的监测值达到了 4.5 秒，因为整个代码路径中除了这个非常耗时的子程序函数，还有额外的线程同步、通知和 I/O 等操作。

分析清楚后，通过简单优化该子程序的渲染算法，从近 50 毫秒降低到 3、4 毫秒后，整个代码路径的线程平均执行时间下降到 100 毫秒左右。收益是明显的，该子程序函数性能得到了 10 倍的提高，而整体执行时间从 4.5 秒降低为 100 毫秒，性能提高了 45 倍。

至此，这个非规律性的 Bug 得到了解决。

虽然案例中最终解决了 Bug，但用的方法却非正道，更多依靠的是一些经验性的怀疑与猜测，再去反过来求证。这样的方法局限性非常明显，完全依赖程序员的经验，然后就是运气了。如今再来反思，一方面由于是刚接手的项目，所以我对整体代码库掌握还不够熟悉；另一方面也说明当时对程序性能的分析工具了解有限。

而更好的办法就应该是采用工具，直接引入代码 Profiler 等性能剖析工具，就可以准确地找到有性能问题的代码段，从而避免了看似有理却无效的猜测。

面对非规律性的 Bug，最困难的是不知道它的出现时机，但一旦找到它重现的条件，解决起来也没那么困难了。

神出鬼没

能称得上神出鬼没的 Bug 只有一种：**海森堡 Bug（Heisenbug）**。

这个 Bug 的名字来自量子物理学的“海森堡不确定性原理”，其认为观测者观测粒子的行为会最终影响观测结果。所以，我们借用这个效应来指代那些无法进行观测的 Bug，也就是在生产环境下不经意出现，费尽心力却无法重现的 Bug。

海森堡 Bug 的出现场景通常都是和分布式的并发编程有关。我曾经在写一个网络服务端程序时就碰到过一次海森堡 Bug。这个程序在稳定性负载测试时，连续跑了十多个小时才出现了一次异常，然后在之后的数天内就再也不出现了。

第一次出现时捕捉到的现场信息太少，然后增加了更多诊断日志后，怎么测都不出现了。最后是怎么定位到的？还好那个程序的代码量不大，就天天反复盯着那些代码，好几天过去还真就灵光一现发现了一个逻辑漏洞，而且从逻辑推导，这个漏洞如果出现的话，其场景和当时测试发现的情况是吻合的。

究其根源，该 Bug 复现的场景与网络协议包的线程执行时序有关。所以，一方面比较难复现，另一方面通过常用的调试和诊断手段，诸如插入日志语句或是挂接调试器，往往会修改程序代

码，或是更改变量的内存地址，或是改变其执行时序。这都影响了程序的行为，如果正好影响到了 Bug，就可能诞生了一个海森堡 Bug。

关于海森堡 Bug，一方面很少有机会碰到，另一方面随着你编程经验的增加，掌握了很多编码的优化实践方法，也会大大降低撞上海森堡 Bug 的几率。

综上所述，每一个 Bug 都是具体的，每一个具体的 Bug 都有具体的解法。但所有 Bug 的解决之道只有两类：事后和事前。

事后，就是指 Bug 出现后容易捕捉现场并定位解决的，比如第一类周期特点的 Bug。但对于没有明显重现规律，甚至神出鬼没的海森堡 Bug，靠抓现场重现的事后方法就比较困难了。针对这类 Bug，更通用和有效的方法就是在事前预防与埋伏。

之前在讲编程时说过一类代码：运维代码，它们提供的一种能力就像人体血液中的白细胞，可以帮助发现、诊断、甚至抵御 Bug 的“入侵”。

而为了得到一个更健康、更健壮的程序，运维类代码需要写到何种程度，这又是编程的“智慧”领域了，充满了权衡选择。

程序员不断地和 Bug 对抗，正如医生不断和病菌对抗。不过Bug 的存在意味着这是一段活着的、有价值的代码，而死掉的代码也就无所谓 Bug 了。

在你的程序员职业生涯中，有碰到过哪些有意思的 Bug呢？欢迎你给我留言分享讨论。