隐藏在表面之下：检测和修复常见应用程序性能问题的根本原因的实用技巧

应用程序对业务有直接影响。当关键应用程序性能不佳时，业务会受到影响。

IT 团队在交付应用程序时必须应对的基础架构日益复杂，这也加剧了性能问题的严重程度。包含虚拟化、托管和云环境组合的基础架构。这些应用程序越来越趋向于分散化、多层次，并且包含的各种组件来自越来越多不同的供应商。

当今企业的混合本质使问题更加困难，故障排除人员通常不能即时掌控数据和资源。原来的问题只需要考虑是与网络相关还是与应用程序本身相关，现在还需要考虑 SaaS、云、互联网或其他第三方提供商的环境。

由于这些架构的复杂性，检测和修复应用程序性能问题变得从未如此困难。最终用户事务太慢看起来是由于代码造成的。但是，这种慢通常不是根本原因，而是隐藏在表层之下的底层基础设施问题的症状。

本现场指南检查常见但复杂的应用程序性能问题，这些问题只有从正确的角度查看，并且用可以捕获*所有*数据、而不只是部分数据的工具，才能发现根本原因，本指南以 Jon Hodgson 的实际经验为基础，Jon Hodgson 是 Riverbed APM 主题专家，负责帮助全球数百家组织优化其关键任务应用程序。

第 1 部分*平均值的缺陷*介绍了这样一个概念：性能问题就隐藏在表面之下，但被不充分的监控掩盖了。

第 2 部分*清除“干草堆”*显示了大数据方法如何帮助您从各种干扰项中清除“干草堆”，快速找到其中的“针”。

第 3 部分*关联分析的力量*探讨了一个特别棘手的问题：看起来随机、间歇性的慢速情况从应用程序的一个部分转移到另一个部分。

第 4 部分*性能的三位一体*表明，虽然响应时间和吞吐量经常获得最多关注，但了解负载也是避免误诊和解决许多问题的关键。

第 5 部分*消除泄漏*提供了内存泄漏和类似行为的概述，并介绍了一些常见方法来解决泄漏引发的问题。

第 6 部分*排除类似泄漏的行为*详述了上一节的概念，讨论如何排除其他类型的类似泄漏行为。

隐藏在表面之下，第 1 部分：平均值的缺陷

很多时候，检查 CPU 使用率被认为是揭示应用程序性能问题的一种快速方法。但是，如果您看到百分比过于完美的使用率，例如 25% 或 50%，可能会存在问题。在计算的混乱世界中，没什么程序运行时能自然地*正好*达到 25%。其中一定包含了人为因素，而这是一个危险的信号。

简单的数学计算就能解释原因。假设您的服务器有四个 CPU 核心。整个架构一团乱。一个核心为 100%，另外三个只有很低的负载，0% 或更高一点。在您的监控工具中，您会看到各种示例，但都具有 25% 这样太完美的下限，这意味着值绝不会低于 25%。找出形成使用率峰值的原因，就可能发现导致应用缓慢的原因。

隐藏在表面之下和平均值的缺陷

CPU 使用率是*平均值缺陷*导致的隐藏在眼皮底下的问题的绝佳示例。Sam Savage 教授最先在他的 2000 年 10 月 8 日发表的文章《San Jose Mercury News》中解释了这个概念。这篇文章称：“平均值的缺陷表明：要是计划基于会发生平均状况的假设，那么这些计划通常是错误的。” 图 1 展示了一个幽默的例子，一名统计学家在一条平均只有三英尺深的河里淹死了。

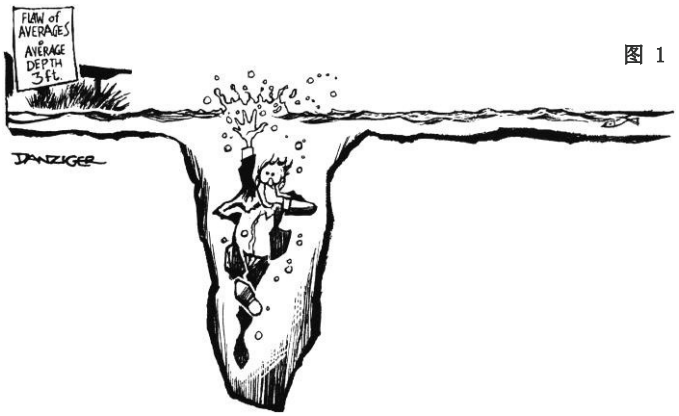


图 1

数据来源：平均值的缺陷：我们低估不确定性面临的风险的原因。作者：Sam L. Savage。插图：Jeff Danziger。

在应用程序性能故障排除中，为什么不应该依赖平均值？ 因为在监控中总是有很大的差异。

事实是，大多数应用程序的绝大多数事务都可能正常。当您把这么多正常事务和糟糕的离群值一起放进统计数据中时，一切仍然会看起来很好 — 离群值被平均值美化隐藏了。

仅仅收集数据抽样的工具会加重平均值缺陷的问题。通过采用大数据方法 — 收集所有数据而不是抽样 — 使应用程序团队能够查看每个用户的每个事务的实际分布，来解决这一问题。

例如，假设您使用 Riverbed® SteelCentral™ AppInternals 测量 CPU 负载，该平台每秒收集一次数据。如图 2 所示，您的 CPU 使用率会间歇性地呈现 100%，而其他时间则呈现较高的使用率。这些峰值指示慢速发生的地方

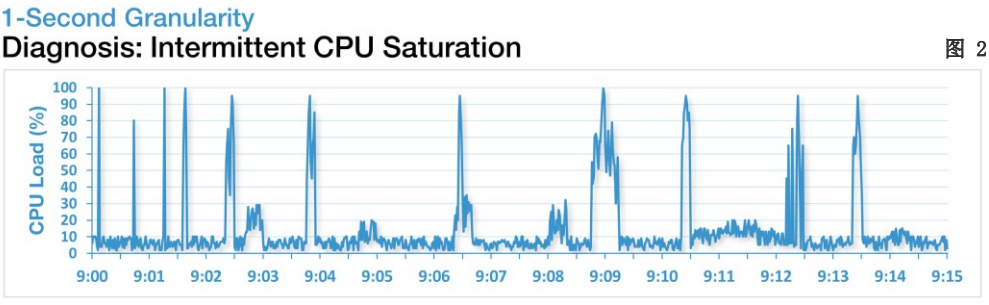


图 2

当您使用只有 15 秒粒度的工具查看完全相同的数据时，显示完全不同，如图 3 所示。这里没有问题，对吧？ 在这两种情况下，问题都存在，但使用 15 秒的粒度时，您意识不到它的存在。数据被平均化了，峰值变得平滑。

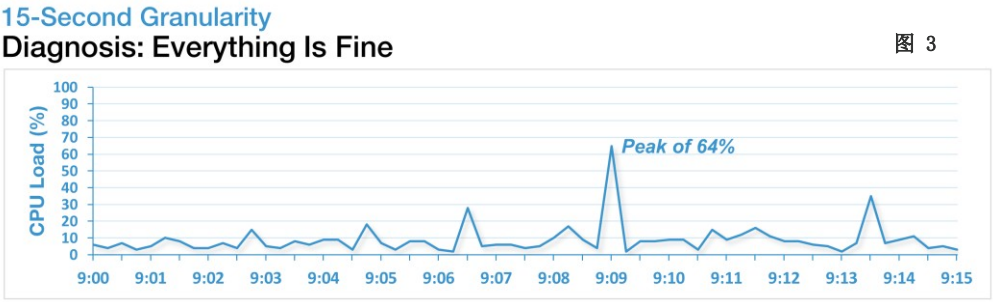


图 3

客户示例：被遗忘的免费软件占用了 10,000 个 CPU

在一个典型的例子中，Jon Hodgson 访问一个客户，并立即发现了一台 16 核机器的 CPU 使用率从未低于 6%。该公司的 IT 团队认为这台机器很好，于是在别处寻找性能问题。

但是，您现在知道，6% 的下限应该是之前讨论的太完美的数字的危险例子。（快速数学计算：整个服务器的 100% 使用率除以 16 个内核，每个内核为 6.25%。）使用 SteelCentral AppInternals，Hodgson 很快发现，一个自 2004 年以来没有更新的免费软件系统管理实用程序占用了 16 核机器中可用的一整个 CPU。

> *“平均值的缺陷会隐藏数据背后的问题，从某个角度来看，这看起来很好，但是如果使用正确的工具查看，就会发现问题。*
>
> – Jon Hodgson
> Riverbed APM 主题专家

问题的复杂之处在于，出现问题的免费软件实用程序是超过 10,000 个公司服务器的默认结构的一部分。免费软件实用程序在这 10,000 个服务器的每一个上面都锁定了一个内核，影响了数千个应用程序和无数项最终用户事务。

没有人知道它，因为它隐藏在表面之下，浪费资源、影响性能。但是，通过使用正确的工具，也就是 SteelCentral AppInternals 查看，客户恢复了相当于 10,000 个 CPU 的处理时间，并且几个难以解释的复杂问题立即消失了，而几乎没花费什么成本和工作量。

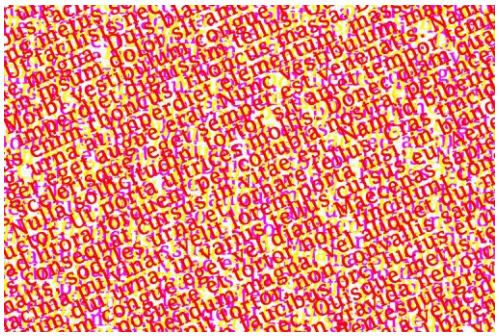
结语：修复问题时揭示其他问题

修复总体性能问题可以通过两种重要方式揭示其他问题。首先，用户通过更频繁或以不同的方式开始使用如今性能更好的服务来改变他们的行为。这给不同的资源带来新的压力。

其次，故障排除人员会注意到之前由于第一个问题的可怕影响而隐藏起来的其他问题区域。为了举例说明，假设有一条装配线，其中 Joe 花费 40 秒来完成任务。当您修正 Joe 的表现时，您突然意识到，流程中的下一个人员 Bob 的表现也不好，只是之前被 Joe 的低效率掩盖了。这通常被称为“向下游移动的问题。”

隐藏在表面之下，第 2 部分：清除“干草堆”

图 4



要在干草堆中找一根针，最简单的方法是清除干草堆。在本示例中，涉及应用程序性能管理 (APM) 时，您需要排除干扰才能找到相关因素。

解码隐藏的消息

为了说明这一点，图 4 展示了一个似乎无法破译的混乱字母组合。但是如果使用正确的工具，本示例中是 3D 眼镜，就会显示模式（图 5）。

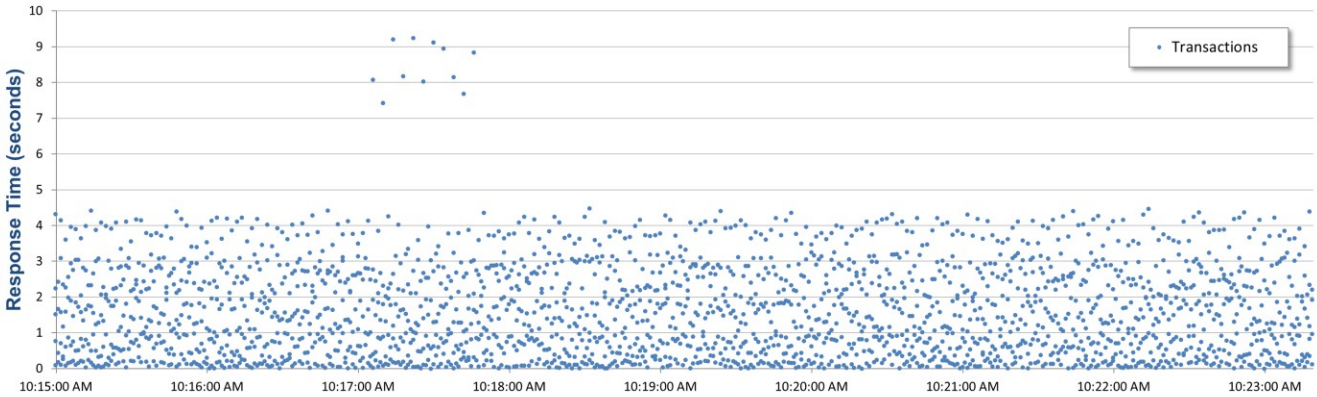
图 5

干草堆：事务干扰

这是一个技术示例。大多数团队从分析最慢的事务开始，然后确定根本原因是一段慢速的代码。图 6 显示了在 8 分钟时间内的 2,000 多个事务。用户投诉不断涌入，因此 APM 团队在上午 10:17 专注处理这些事务，耗时 7 到 9 秒。

但是，只要团队修复这些缓慢的事务，就能停止最终用户投诉吗？ 请记住，仅仅因为某些事最慢并不意味着它们就是最影响用户的罪魁祸首。古老的逻辑最能揭示真理：相关并不意味着是因果关系。

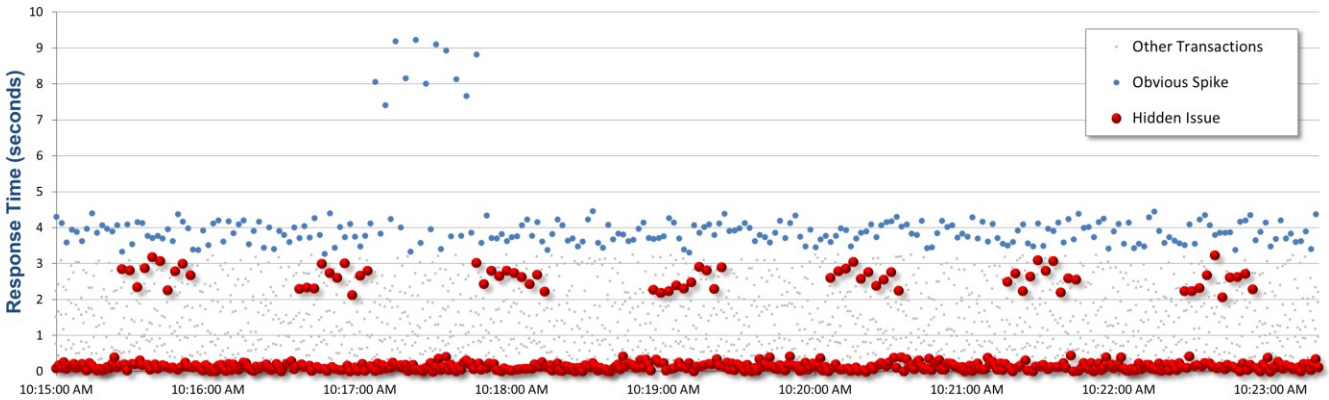
图 6



事实是，图 6 中的数据集是许多事务类型的混合，每种事务类型都有自己典型的和可接受的性能范围。这种混合的行为掩盖了潜伏在表面以下的问题。为了解决问题的根本原因，需要通过清除“干草堆”，深入探寻，找到隐藏在眼皮底下的针头。为此，您需要一种捕获所有事务的大数据方法。

那么，这在实践中如何操作？ 图 7 显示了相同的事务，但现在您可以区分不同的事务类型。蓝色事务通常需要大约四秒钟，但其中有一分钟，有些事务需要大约两倍的时间。这些是 APM 团队关注的事务。

图 7



相比之下，红色事务较快，通常只需要约 250 毫秒或更短。但是每隔一分钟左右，有些事务会花费大约 11 倍的时间。这是一个比 APM 团队最初关注的原始峰值更严重的问题。原因何在？ 三个原因：

1. 这是一个更大的行为变化。
2. 它影响更多的事务。
3. 它长期发生。

在本示例中，问题根源是一个过载的数据库，一个完全不同于初始峰值的问题。修复初始峰值不会解决来自过载数据库的性能问题。但是解决数据库问题会带来最大的性能提升。

要进行交叉分析，您需要所有的数据

要清除“干草堆”，您需要一个工具集，让您可以交叉分析您的数据集，以显示在聚合中不可见的模式。如果您只获取一部分事务，则将只解决一部分问题。因此，至关重要的是，您的 APM 解决方案始终捕获所有事务，而不是依赖于不完全的抽样方法。

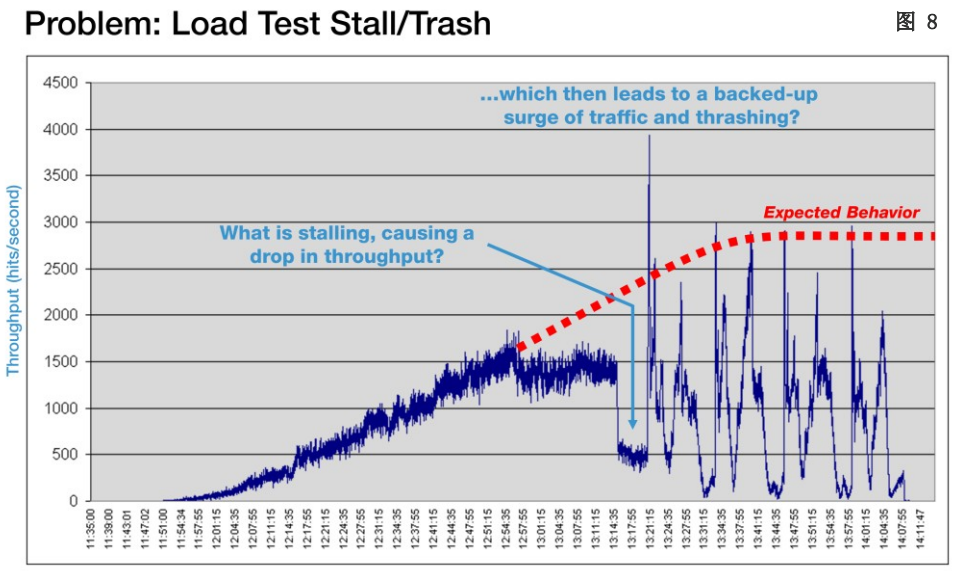
> *“如果您只获取一部分事务，您将只解决一部分问题。”*
>
> – Jon Hodgson
>
> Riverbed APM 主题专家

客户示例：针 — 锯齿模式

让我们用 Jon Hodgson 档案中的一个真实例子来证明这一点。他与一家从事银行、股票交易和投资组合管理的金融机构合作。监管需求要求 APM 团队确保公司的应用程序可以处理流量的激增。过去，事务突发会导致系统锁定，这可能对股市产生级联效应。

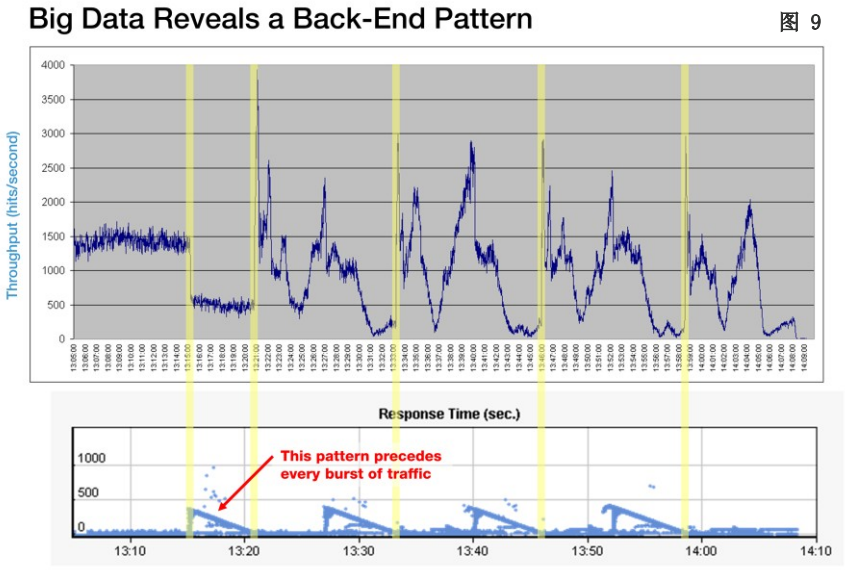
为了了解应用程序在严重压力下会发生什么情况，团队在包括数百台服务器的生产并行平台上将流量提高到了最高交易日峰值负载的 3 倍。在这项测试下，预期吞吐量会成比例地增加，直到当某些资源（如 CPU 或数据库）饱和（见图 8 中的红线）。但是，IT 团队看到了波动行为，这表明与预期完全不同。

所以 IT 团队想了解：是什么导致失速，导致吞吐量下降，然后激增，上下波动？顺便说一下，这是一个死亡螺旋，环境不能恢复，导致情况更糟。



为了找到根本原因，Hodgson 安装了 SteelCentral AppInternals，它能够记录所有事务，即使是每秒数千次点击也不例外。然后，他将来自 Load Generator 的吞吐量图与来自 SteelCentral AppInternals 的响应时间图进行比较，如图 9 所示。注意不同的锯齿模式，并记住这样僵直的模式表明有人为因素。

仔细检查还发现，锯齿模式在每次流量爆发之前发生。有一个下沉，一个激增，一个下沉，一个激增，循环往复。所以关键是要弄清楚是什么导致了锯齿模式。



接下来，Hodgson 查看了一小部分慢速的事务，并注意到在调用名为 *GetQuotes.jws* 的一个远程 Web 服务时出现了许多延迟，这是一个由其他团队管理的股票行情应用程序。

请记住，提出假设时要谨慎。当您修复最慢的几个事务时，您不能假设您也会修复数百万的其他事务。只是因为最慢的五个事务是由于原因 X，并不意味着事务 6 到 10,000 也是因为 X 变慢的。

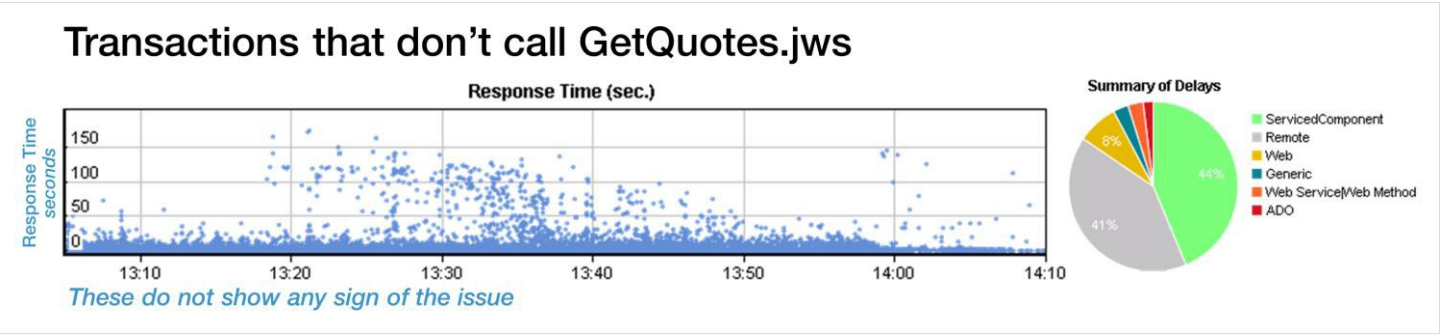
但是，由于单独分析 10,000 个事务是不切实际的，更好的做法是将它们作为聚合集合进行分析。然后，如果整体根本原因事实上是原因 X，您将证明修复它将纠正所有 10,000 个事务。

为了测试这个理论，Hodgson 只关注那些对 GetQuotes.jws Web 服务进行远程调用的事务。锯齿还在，但许多其他事务被过滤掉，进一步确认了最初的假设。

图 10

虽然这些信息本身是令人信服的，但 Hodgson 希望在提出最终会导致大量工程工作、甚至可能导致政治动荡的建议之前做到绝对肯定。所以，为了最终确认，他逆向测试了这个理论 — 只显示不调用 *GetQuotes.jws* 的事务：

图 11



没错！锯齿模式和破坏行为完全消失。这证实了这个理论无可置疑。*GetQuotes.jws*，一个共享的下游 Web 服务，是数百万慢速事务的罪魁祸首。

这个例子清除了“干草堆”来显露所有的“针”，让 APM 团队能够找出所有针具有什么共同点，并确定需要修复的单个组件。然后，团队使用 SteelCentral AppInternals 确定了这个问题，这个问题影响了几十个看起来毫不相关的应用程序中的数百种不同的事务类型，很明显企业需要解决它。

客户自己的工程师花了几个月调查代码和服务器，但他们无法确定问题。但在使用 SteelCentral AppInternals 几个小时后，Riverbed 解决了这个问题，量化了它的影响，并向客户提供了采取行动所需的证据。这只是大数据应用的另一种方式，通过 SteelCentral AppInternals 之类的解决方案实现，可以帮助揭示隐藏在表面下的性能问题。

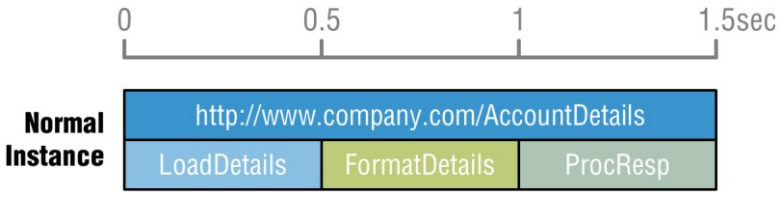
隐藏在表面之下，第 3 部分：关联分析的力量

性能故障排除中的一个常见问题是，看起来随机、间歇性的慢速从应用程序的一个部分转移到另一个部分。没有正确的工具和方法，您可能无法识别表面幻影背后的根本原因。本节重点介绍如何识别一些最常见的（和罕见的）原因，以便您可以从应用程序中一劳永逸地清除这些罪魁祸首。

不要急于下结论

考虑图 12 中所示的假设的关键事务，这个事务通常需要 1.5 秒来执行。

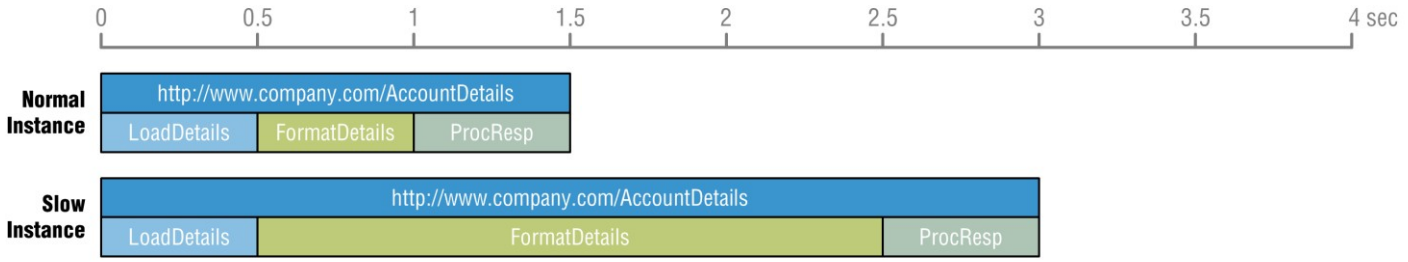
图 12



在实际应用中，这个事务可能包含数千个调用方法，但是为了举例方便，这里简化为三个主要子步骤，每个半秒。如果 APM 团队对此事务的服务级别协议 (SLA) 为 2 秒，则当超过这个值时，团队成员会在其控制面板上收到警报，提示他们深入查看慢速事务，作进一步的调查。

比较慢速实例和典型的快速实例，很明显，"FormatDetails" 代码是变慢的原因：

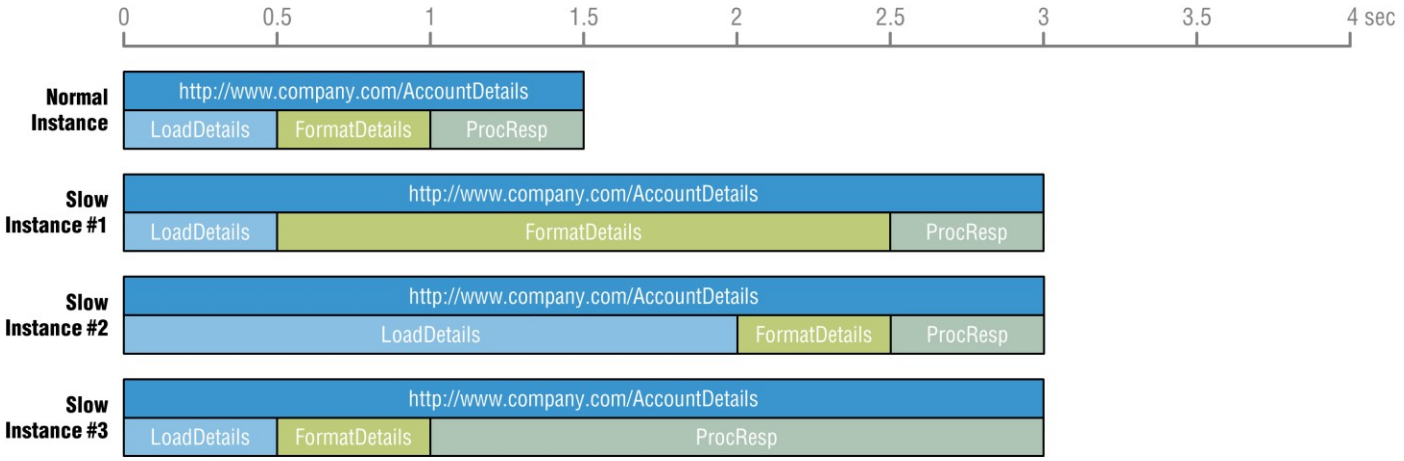
图 13



要假设您现在了解这个问题，很容易。但是要小心，不要过早地下这种结论，以免向开发人员提出不能解决根本问题的修复请求。

为了验证初始理论，让我们通过查看事务的多个实例来作进一步调查：

图 14



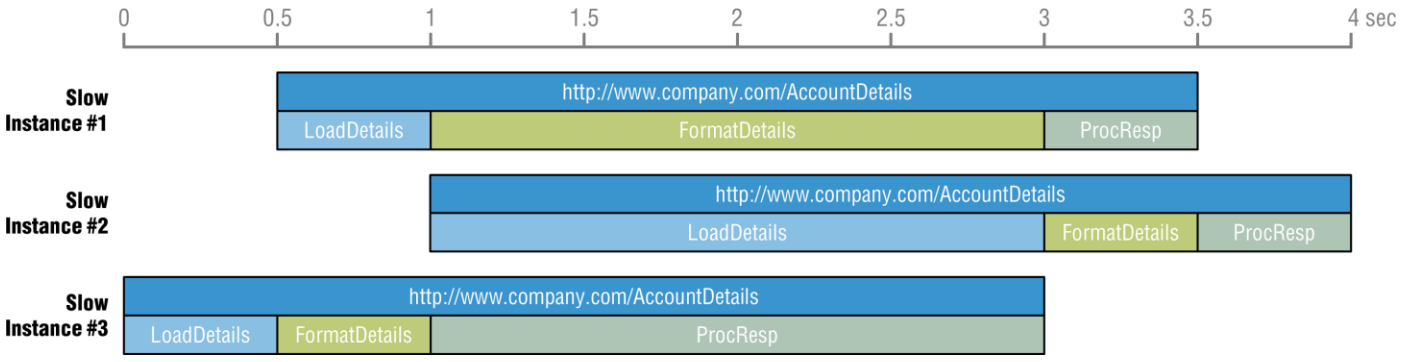
很明显，将问题归因于 "FormatDetails" 方法是错误的。但是，如果尝试分别优化这三个，用最慢的方法是不是就能解决问题？

可能不是 — 问题根源可能不是代码本身，或者代码的慢速可能是其他因素的副作用。您必须找出影响所有三个组件性能的常见根本原因。

时间就是一切

在这种情况下，您不能独立地分析每个有问题的事务。您需要将每个事务放到其他事务的背景环境中进行考虑。背景环境的一个关键作用是理解每个事务相对于其他事务的运行情况。

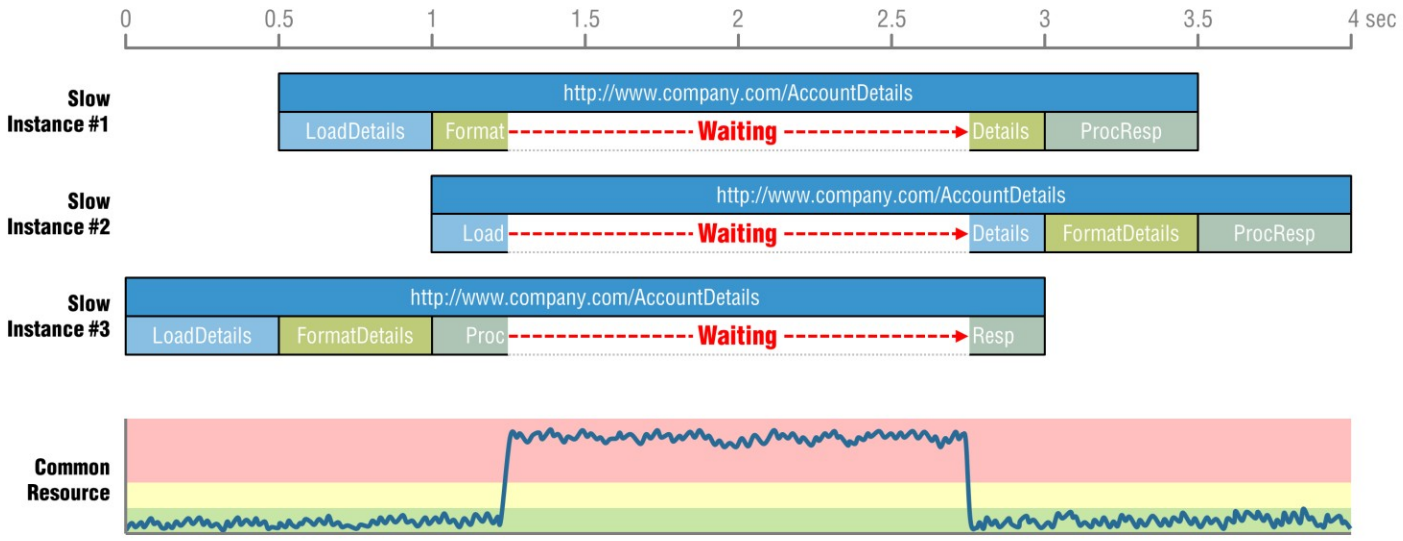
图 15



这里，各项延迟彼此对齐，表明在这个时间期间可能发生一些常见的根本原因。某些共享资源（如 CPU）可能饱和，或磁盘可能正在排队，或者垃圾回收可能已暂停 JVM/CLR。

我们假设一下可能发生的事情：

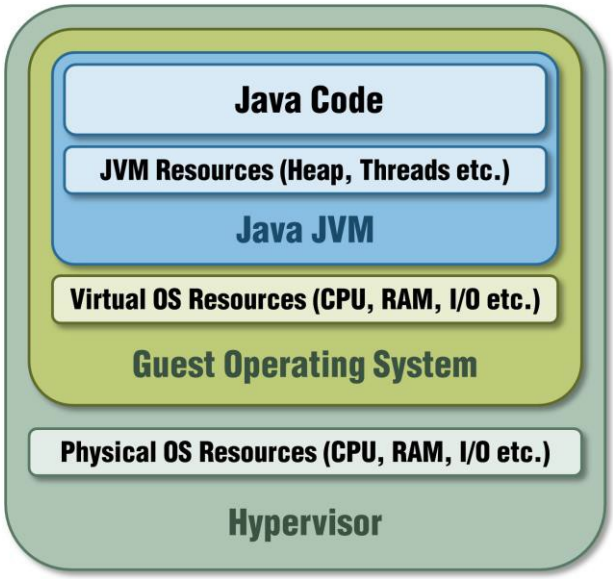
图 16



无论具体原因是什么，任何在那时运行的应用程序代码都会停止或减慢，直到可以访问完成执行所需的资源。

请记住，代码在 JVM/CLR 中运行，而 JVM/CLR 又在操作系统上运行，操作系统可能是在虚拟机管理程序上运行的 VM，因此代码依赖于堆栈中较低的所有资源（图 17）。

图 17

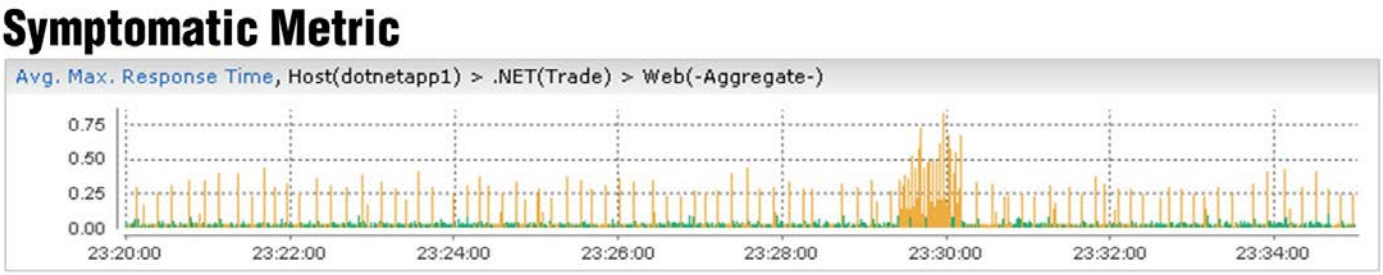


相关性，我的侦探能手

要确定代码为什么变慢，还必须监视整个堆栈的行为和资源使用率。AppInternals 通常每秒为应用程序中的每个服务器监视 5,000 个指标。您需要一种方法来确定哪些指标与您在应用程序中看到的响应时间症状有关。幸运的是，AppInternals 提供了一个简单的方法来实现这一点：关联分析。

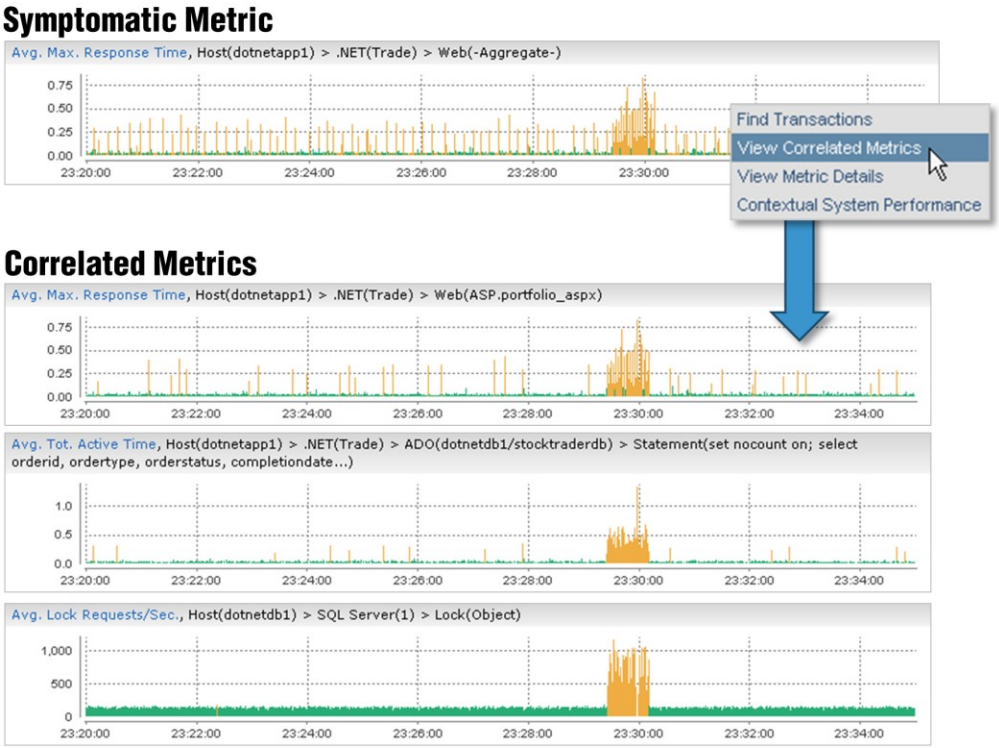
首先挑选症状典型的指标。查看所有事务的总体响应时间通常是一个好方法，因为它考虑很多不同类型的事务，这些事务可能都受到共同的根本原因的影响。

图 18



SteelCentral AppInternals 动态分析所有指标，以确定其典型的行为范围，如上所示的绿色范例。该范围之外的橙色范例被视为“偏差”，并基于严重程度进行评分。关联分析使用此信息扫描数以万计的指标，以找到与症状最相似的几个指标。如果查看响应时间的峰值，并找出哪些指标相关，就可以弄清楚最终的根本原因是什么。一些相关指标可能是症状，有些可能是副作用，而其余的将是您寻找的根本原因。

图 19



搜索将产生您不曾想到或已知的查找结果。您不再受自己的专业知识限制。数据库管理员可以在 Java 代码中发现问题，开发人员可以在操作系统中发现问题。

换句话说：将从应用程序收集的 150,000 个指标视为来自 3,000 个不同谜题的一组拼图。关联分析在功能上是一块磁铁，它能进入问题堆里，然后吸出 50 个相关的问题。它能告诉您：“这是 50 个与您相关的问题，其余是干扰因素。”

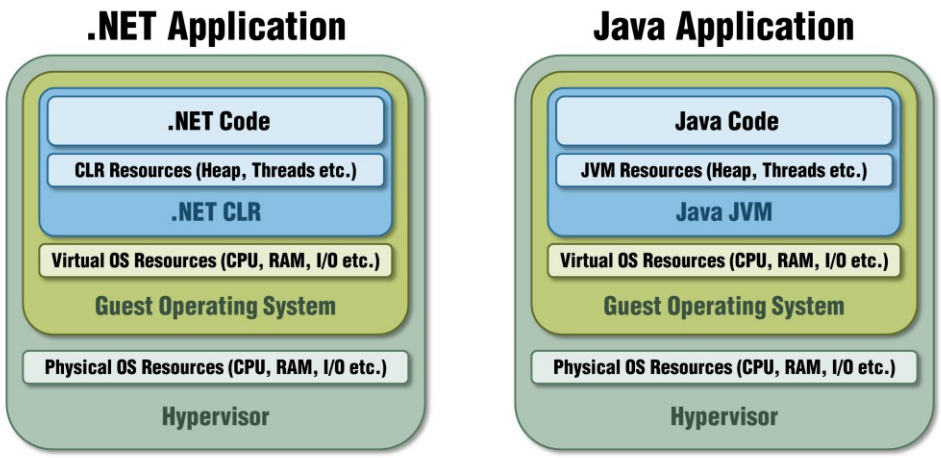
现在您需要做的是组装更简单的拼图。即使这些因素不属于您的专业领域，您也会知道需要向哪些主题专家寻求帮助，并为他们提供快速解决问题所需的可操作信息。

图 20

客户示例：将应用程序放在背景环境中分析

很久以前，Jon Hodgson 在客户的网站遇到了类似的问题。客户使用另一个 APM 工具监控两个不同服务器上的两个关键应用程序。客户在 .NET 应用程序上遇到间歇性慢速问题，但无法通过使用现有工具确定根本原因。

图 21

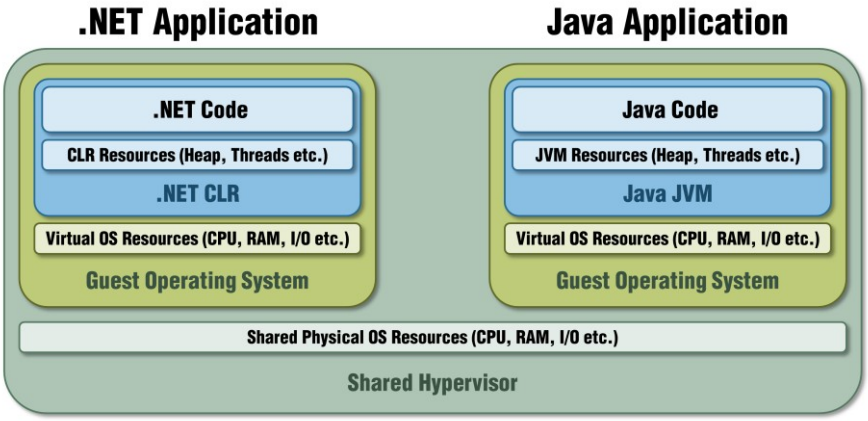


继续讨论示例之前的一个重要提示：在故障排除时请注意不要限制自己。人类有偏见，并经常做出关于根本原因的假设。您可能对环境和应用程序基础架构的知识有限，因此不要太过武断地确定您认为可能相关的内容。正如在其他事务的背景中分析事务非常重要，您需要在应用程序运行的环境这一更大背景中分析应用程序。

按照这一原理，Hodgson 安装了 SteelCentral AppInternals 来监控两个应用程序。他决定提取 .NET 应用程序中的响应时间峰值，并将其与两个应用程序的所有指标进行相关分析。这个措施发现了意想不到的事情：.NET 应用程序和看似不相关的 Java 应用程序之间存在明显的关系。每当 .NET 应用程序中出现响应时间峰值时，都与 Java 应用程序中的 CPU 峰值和垃圾回收相关。

不适用

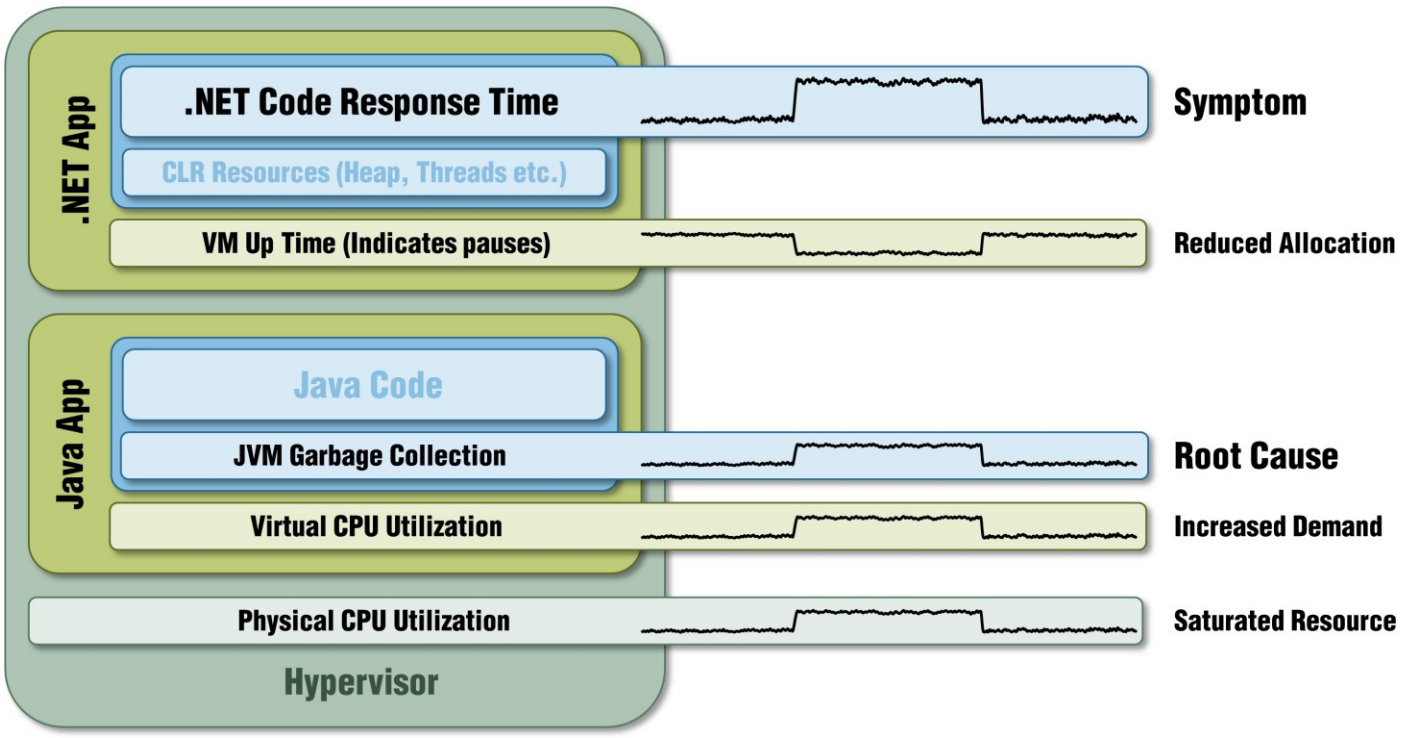
图 22



AppInternals 揭示了与其相关的 VMware 指标的其他信息。与客户最初的想法相反，两个应用程序服务器都是在同一虚拟机管理程序上运行的 VM（图 22）。

不是将服务器当作并行运行的独立对等体，而是将它们视为同一堆栈的一部分，因为它们共享公共资源。Java 应用程序正在进行垃圾回收，消耗 CPU 并需要更多的物理资源。共享虚拟机管理程序过度使用，所以它必须拆东墙补西墙 — 从 .NET 应用程序的 VM 窃取 CPU 周期，从而暂停 .NET 应用程序的代码执行，直到虚拟机管理程序提供资源（图 23）。

图 23



虽然症状是 .NET 应用程序的响应时间，但根本原因在于一个看似不相关的 Java 应用程序。然而，这两个应用程序是相关的，因为共享物理基础设施。关联分析使这个未知的关系变得清晰。

可见性和信任问题在虚拟化机器中很常见，因为您并不总是确切知道应用程序在哪里运行，以及是否实际上已经获得了承诺为他提供的资源。VMware 服务器通常过量使用，因为它们通常针对访客 VM 的平均资源需求进行配置。但是，当多个客户在意外的很短的时间内请求相同的资源时，就会触发*平均值的缺陷*。使用错误的工具，这些基础结构问题可能看起来像代码问题，您会无休止地跟踪一个永远发现不了的问题。

有几十个其他*“普通嫌疑犯”*，比如垃圾收集、饱和线程池和磁盘排队，和数百个*“不寻常的嫌疑犯”*可能很容易成为根本原因。这就是为什么对代码及其依赖关系（JVM、操作系统、虚拟机管理程序等）进行全面监控至关重要，因为这样无论什么根本原因，关联分析都能为您找到它。

对代码及其依赖关系（JVM、操作系统、虚拟机管理程序等）进行全面监控至关重要，这样无论根本原因是什么，关联分析都可以帮助您找到它。

下次您对问题进行故障排除，但找不到一致的根本原因时，您可能是找错了地方。后退一步，使用您所拥有的任何工具来深入了解底层基础架构。

隐藏在表面之下，第 4 部分：性能的三位一体

“性能”对您意味着什么？

大多数人对这个问题的回答会涉及响应时间，例如“我们的网站网页应在不到 2 秒内加载”或“我们需要确保我们的 SLA 达到 5 秒”。这些回答情有可原，因为 APM 的主要目标之一是保持客户满意，并且最终用户可以看到的 APM 的主要方面是应用程序需要响应多长时间。话虽如此，响应时间实际上更多是性能的副作用。它只是症状，而不是根本原因。

少数人的回答与吞吐量相关，例如“我们的应用程序需要能够每秒处理 1,000 个检出。” 吞吐量是每单位时间的操作在应用程序中完成多少工作的度量，是性能的重要指标。然而，在没有更大环境的情况下可能难以解释吞吐量。

关于性能，存在许多教科书定义，但在 APM 环境中最准确的是：

“机器、车辆或产品的能力，特别是在特定条件下观察时。”

性能有许多教科书定义，但在 APM 环境中最准确的是：“机器、车辆或产品的能力，特别是在特定条件下观察时。”

特别注意最后半句：*“……特别是在特定条件下观察时。”*了解深层的情况是能够解释您看到的效果的关键。不幸的是，很少有人在回答上述初始问题时考虑到这一点，人们往往忽略了答案中最重要的部分：负载。

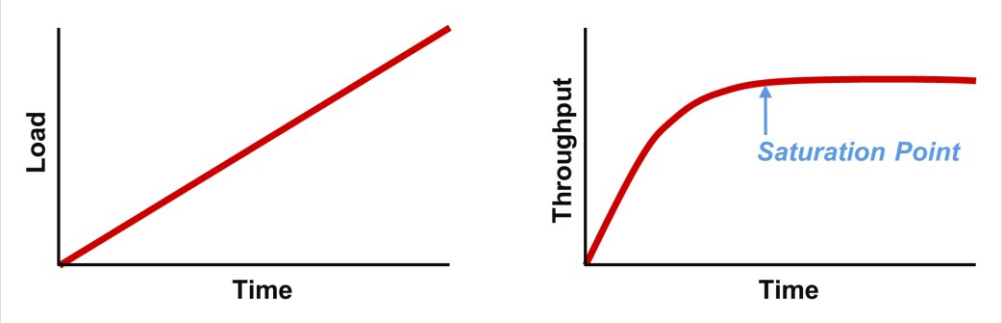
性能的三位一体

为了充分解决性能问题，您需要了解负载、吞吐量和响应时间之间的关系。

负载是所有工作开始的地方。正是它要求完成某项工作。没有它，您没有工作，也就没有响应时间。这也是导致性能最终下降的原因。在某些时候，对工作的需求比应用程序能够提供的需求更大，这时就会出现瓶颈。

图 24

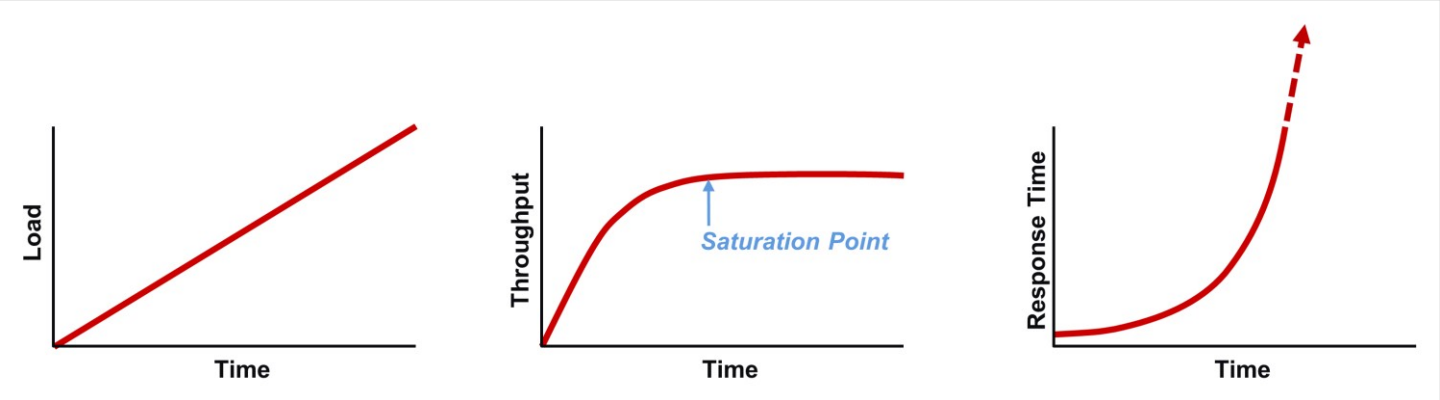
吞吐量是所需工作的执行速率。负载和吞吐量之间的关系是可预测的。随着负载的增加，吞吐量也会增加，直到一些资源饱和，之后，吞吐量将达到“平稳”状态。



当吞吐量达到“平稳”状态时，这是一个指标，此时您的应用程序不再扩展。

响应时间是吞吐量的副作用。当吞吐量与负载成比例地增加时，响应时间的增加可忽略，但是一旦达到吞吐量“平稳”状态，响应时间将随着排队的发生呈标志性“曲棍球棒”曲线式的指数增长：

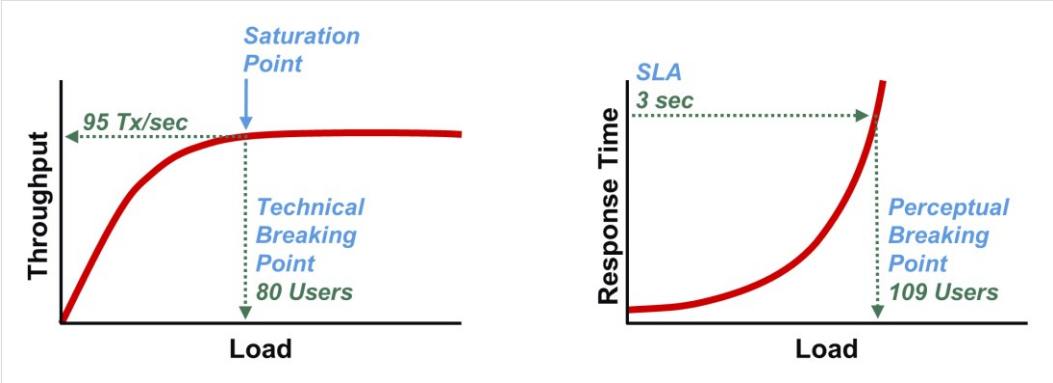
图 25



响应时间不是一切

从技术角度来看，响应时间实际上并不是性能瓶颈的指标。响应时间阈值和 SLA 是人为结构，用于指示对最终用户来说什么时候“太慢”。在以下示例中，SLA 为 3 秒，映射到 109 个用户；然而，饱和点仅映射到 80 个用户。

图 26

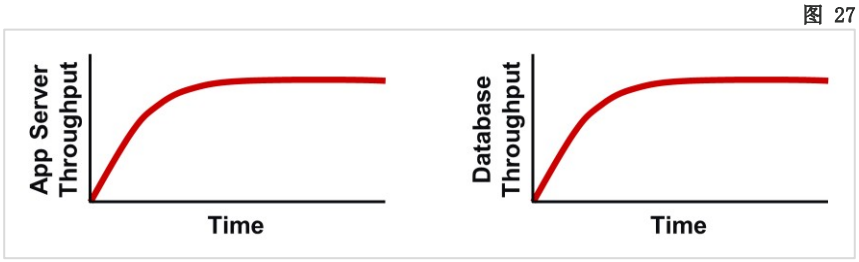


虽然对于超过 109 个用户来说，应用程序确实不满足他们 3 秒的 SLA 要求，但这只是一个业务指标。从技术上讲，说应用程序没有扩展超过 109 个用户是不正确的，因为它实际上在 80 个用户时停止了扩展。这一点非常重要，因为它决定了您如何对问题进行故障排除，以及是否会确定根本原因。

您需要确定为什么性能瓶颈出现在 80 个用户处（技术突破点）。如果您确定并解决这个问题，就可以解决 109 个用户的 SLA 问题（感知断点）。

比较和对比

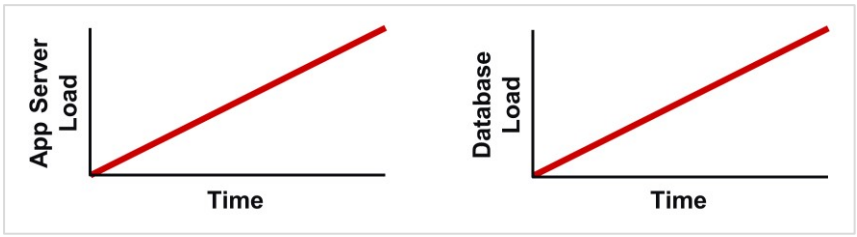
图 27



在当今的多层应用中，负载和吞吐量之间的关系变得越来越重要。当发生吞吐量“平稳”状态时，可以同时在多个层上见到（图 27）。

在这种情况下，应用程序团队会自然倾向于归咎下游项目，因为低性能的下游通常会影响上游。虽然这是对响应时间的真实评估，但并不总是适用于吞吐量。

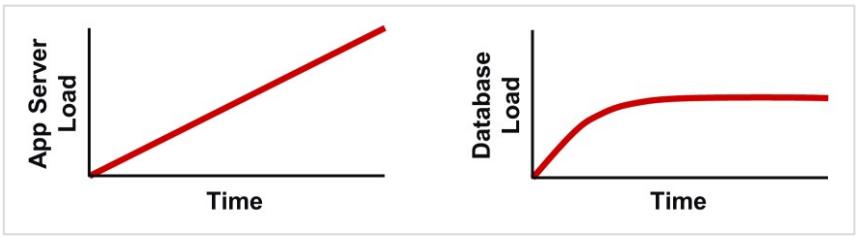
图 28



通过比较每层的吞吐量和负载，可以确定根本原因是什么。这里有一个对应于上述吞吐量的可能情况，其中负载在每一层呈线性增加（图 28）。

在这种情况下，由于负载在数据库层不断增加，可以确定瓶颈确实在下游数据库层上。

图 29



另一种可能的情况是应用服务器负载呈线性增加，但不会传播到数据库层（图 29）。

在这种情况下，瓶颈实际上是在应用程序服务器中，因此不会将负载传递到数据库服务器。记住负载是对工作的需求，在某些时候对数据库没有额外的负载要求。没有额外的负载，就不会有额外的吞吐量。

这两种情况说明，您不应该基于初始有限的数据集仓促下结论，而应该从其他角度交叉参考您的理论，来确保它准确无误。这将确保您不浪费宝贵的时间来尝试解决错误（察觉）的问题。“两次测量，一次切割”的木工技法同样适用于 APM。

客户示例：并非总是应用程序的故障

这是一个真实世界的场景，负载是解决问题的关键因素。Jon Hodgson 正在与一个购买 SteelCentral AppInternals 的新客户合作，帮助他们解决之前历经数月依然无法解决的性能问题。

客户按常规方法对其 Java 应用程序运行标准斜坡负载测试，但总是会失败，因为在负载适度增加之后响应时间会大大延长。

图 30 显示了客户从负载发电机的角度看到的情况：负载按照他们的预期线性增加，响应时间逐渐增加一段时间，直到发生“曲棍球棒”响应时间峰值。

然后，Hodgson 在五个前端应用程序服务器上安装了 SteelCentral AppInternals Java 监控代理程序，客户重新运行了它们的负载测试，再次出现同样的响应时间增加。但是，这次他们能够分别看到五个服务器的*内部*情况，发现了一个不同的问题（见下页的图 31）。

图 30

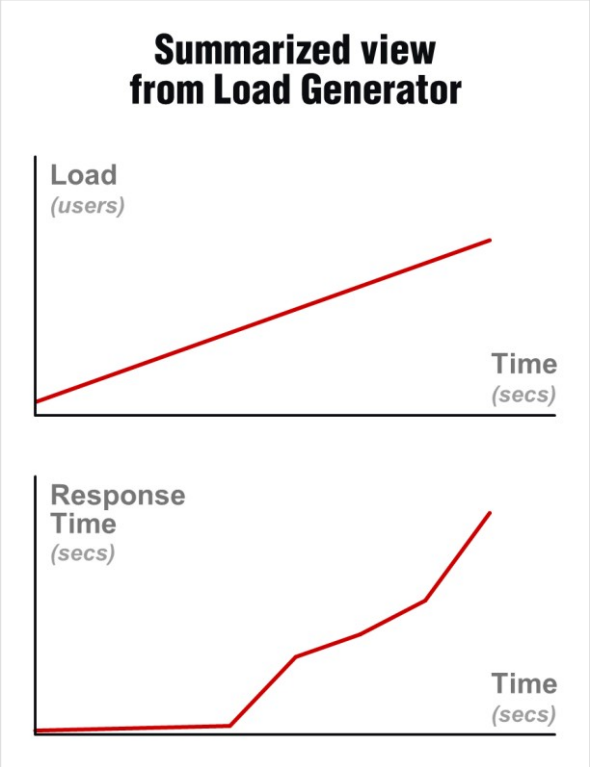
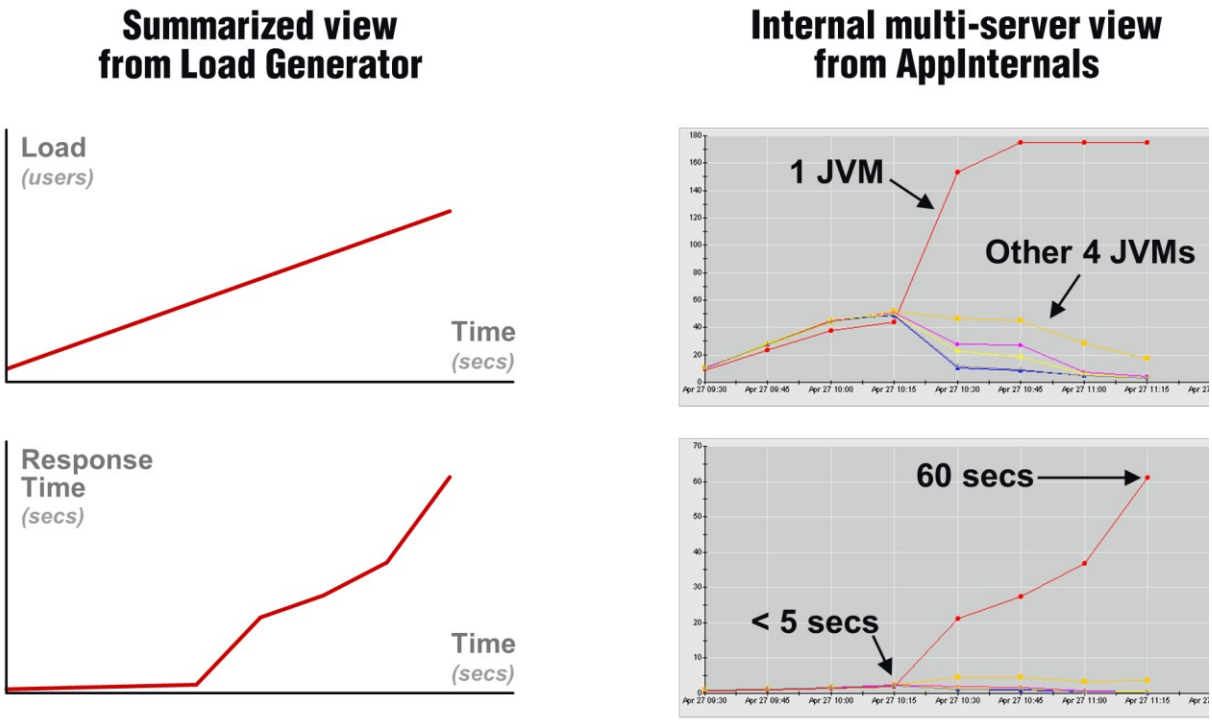


图 31



在 AppInternals 的右上图中出现了意想不到的情况。在测试的前半部分，负载在所有五个服务器之间均匀分布，然后发生了巨大的变化，几乎所有负载都被发送到一个服务器，此时，该服务器上的响应时间从 5 秒增加到 1 分钟，增加了 12 倍。在此期间，其他四个服务器具有极快的响应时间。

这个问题根本不是应用服务器性能的问题，而是负载均衡问题。负载均衡器被设置为一个简单的“循环”模式，众所周知这一模式是这种类型问题的罪魁祸首（虽然人们会认为它会均匀分配负载）。如果循环群集中的一台服务器稍弱或使用率稍多，则盲目地馈送到它的事务可能开始堆积，导致这里显示的不均衡和过载。

解决方案很简单。客户将负载均衡策略更改为“最少连接数”，这确保了在一个服务器落后的情况下，其他服务器会接收事务。再次进行负载测试时，运行要好得多。在性能开始降级之前，客户能够生成大约 4 倍的负载，他们不必优化单个代码或添加任何新硬件，即可体验这些改进。

客户之前专注于应用程序服务器及其下游依赖关系，从而忽略了在上游的根本原因。这是另一个很好的例子，用正确的工具和正确的方法找到问题的根本原因，就隐藏在表面下，只需要几个小时的努力就能解决。

隐藏在表面之下，第 5 部分：消除泄漏

如果您有应用程序性能分析的经验，那您遇到泄漏的情况可能比您意识到的更多。如果您的应用程序在运行一段时间后，就会出现性能下降或完全崩溃，需要定期重新启动才能恢复，则您可能存在应用程序泄漏。

大多数人都熟悉内存泄漏，但许多不同类型的资源都可能表现出类似泄漏的行为，需要您用各种方法来解决。

什么是泄漏？

当应用程序使用资源后没有返还资源，就会发生泄漏。可能的资源包括内存、文件句柄、数据库连接和许多其他资源。即使像 CPU 和 I/O 这样的资源也可能泄漏，如果调用代码遇到意外情况，导致形成无法突破的循环，就会随着时间的推移，堆积更多代码实例，处理工作量越积越多。

要理解一个关键概念，所有泄漏的发生都是因为一些有问题的代码块在运行。这个代码可以由最终用户 Web 请求、后台定时器、应用程序事件和许多其他事件触发。代码运行的频率越高，泄漏累积得越快。

图 32

可以想象一个简单的场景，就像一个牌堆，有人往上加牌，一次一张。尽管每张牌非常薄，但是时间久了，这叠牌也会非常高。牌加得越快，牌堆升高得越快。如果停止加牌，牌堆就不再升高，直到恢复加牌。

您是否存在泄漏问题？

以下是您的应用程序中可能存在泄漏或一些类似泄漏的行为的迹象：

- 您的应用程序会逐渐变慢，需要定期重新启动才能解决。
- 您的应用程序会逐渐变慢，但重新启动没有帮助。
- 您的应用程序会有几天或几周运行正常，然后突然启动失败，需要重新启动。
- 您看到一些资源的使用率随时间增长，需要重新启动才能解决问题。
- 您没有对应用程序的代码或环境进行任何更改，但其行为随着时间推移发生了变化。

这些都是严重的问题，可能会对应用程序的可用性、稳定性和性能产生负面影响。它们可能经常导致许多其他幻象问题，您可能浪费无数小时试图解决问题但总是失败。

为什么泄漏不好？

泄漏应该列为您要从应用程序中消除的头号问题。不同的资源泄漏对应用程序行为有不同的影响。连接池和文件句柄等资源的泄漏最初对应用程序没有负面影响，直到应用程序空间完全耗尽，此时应用程序只能等待无泄漏的实例变得可用，这时会发生崩溃或严重延迟。

其他泄漏资源（例如 Java 虚拟机 (JVM) 堆）在使用率增加时会对应用程序性能产生轻微影响，一旦应用程序的空间耗尽，就会完全崩溃。

当失控线程消耗 CPU 资源时，其他代码可用的循环会变少，当需要这些循环时，这些循环就会变慢。

磁盘和网络 I/O 也是如此。

为了尽可能高效地执行应用程序，您需要确保在应用程序需要的时候，都能始终拥有所需的资源。泄漏为应用程序故障排除人员提供了一个移动目标，因为性能问题可能仅仅是现有泄漏或一个全新的、不相关的问题的累积效应。

识别和消除泄漏至关重要，这样他们就不会对出现的新问题造成干扰。正常的应用程序在同一事务的第一百次和第一百万次执行之间的性能上应该没有差别。如果有差别，那就表示存在某种类似泄漏的行为。

要理解一个关键概念，所有泄漏的发生都是因为一些有问题的代码块在运行。这个代码可以由最终用户 Web 请求、后台定时器、应用程序事件和许多其他事件触发。代码运行的频率越高，泄漏累积得越快。

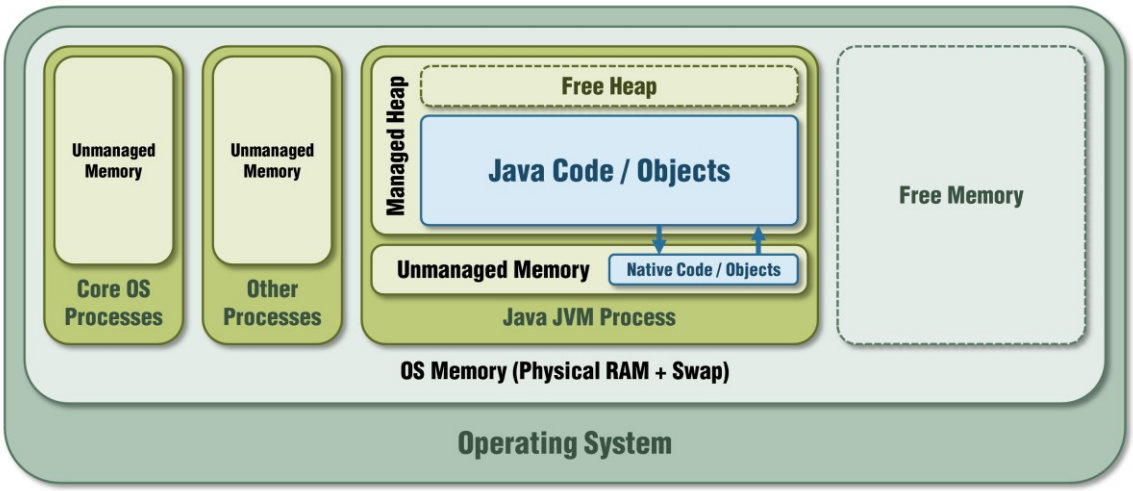
Java 和 .NET 内存泄漏

由于 Java 和 .NET 应用程序日益普及，与这些应用程序类型相关联的内存泄漏是最常见的。因此，对 Java 和 .NET 如何管理内存有基本的了解很重要。

为了简单起见，我们将以 Java 为例，.NET 的功能行为是一样的。

在像 C++ 这样的较老语言中，开发人员直接访问本机操作系统内存，并根据需要手动分配/取消分配内存。Java 仍然使用本机操作系统内存，但是大部分保留用于由 JVM 自动管理的托管“堆”。开发人员可以花更少的时间进行内存管理，花更多时间在代码功能上。虽然这使开发人员的工作更容易，但它在进行故障排除时会使问题更抽象。

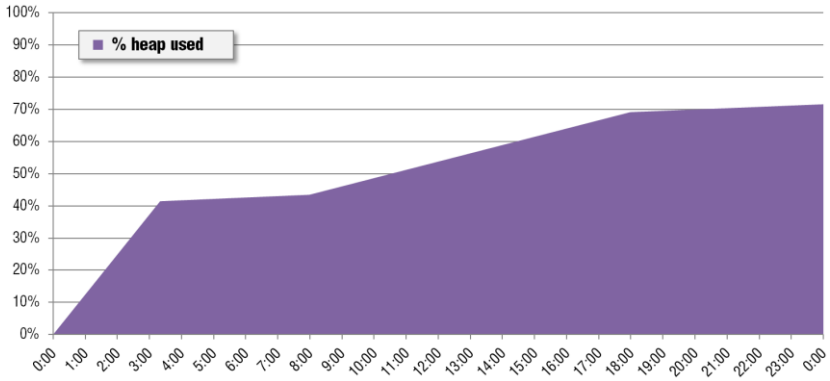
图 33



如果您使用传统的进程监控器*任务管理器*或 *top* 来查看您的 JVM 内存使用情况，列出的内存使用情况会包括 JVM 本身使用的非托管内存、本地库，以及为 Java 代码保留的整个托管堆，可以是 1% 使用率或 99% 使用率。这种模糊性使这些工具无法解决 Java 内存泄漏问题。查看内部堆使用情况的唯一方法是利用可以在 JVM 自身内部看到的专用工具。

大多数 Java 监控工具会显示堆使用率百分比。不同的工具将根据收集的方法以不同的方式报告这些数据，但重要的是要证实，在一段时间内，模式要么是平坦的，要么是上升和下降不断重复的。如果您看到堆使用率的趋势总体向上，则很可能存在泄漏，一旦达到 100%，应用程序就会崩溃。

图 34



知道存在泄漏只是解决问题的十分之一。如果您想确定泄漏的根本原因，则需要一个高级工具，可以看到在一段时间内，哪些对象占用堆和存在泄漏。

许多工具仅仅依靠堆“转储”来确定某个时间点的堆的内容。这些转储要求应用程序暂停几秒钟，有时几分钟，如果您无法将应用程序路由到能够继续其现有会话的其他系统，可能会导致用户中断。

转储通常会显示哪些对象正在使用最多的堆，如图 35 所示。

图 35

Object Name	Objects in Heap (22:00)
com.company.newsfeed.story	100,000
com.company.order.shippingDetails	67,679
com.company.health.pingResult	9,240

这可能会导致您认为 *newsfeed.story* 对象存在泄漏，因为它使用了最多的堆。然而，没有办法知道它的泄漏是不是存在另一个转储，来进行比较，所以这些工具通常会要求您在运行大约一个小时后进行另一个转储（另一个潜在的用户中断），来确定这段时间内变化最大的是什么。

图 36 显示了两个堆转储的比较，表明 *newsfeed.story* 可能只是干扰因素，但其他两个结果似乎也不是根本原因。虽然 *order.shippingDetails* 是堆的第二大消耗程序，但它的变化可以忽略不计。虽然 *health.pingResult* 的变化最大，但它是一个相对较小的堆。

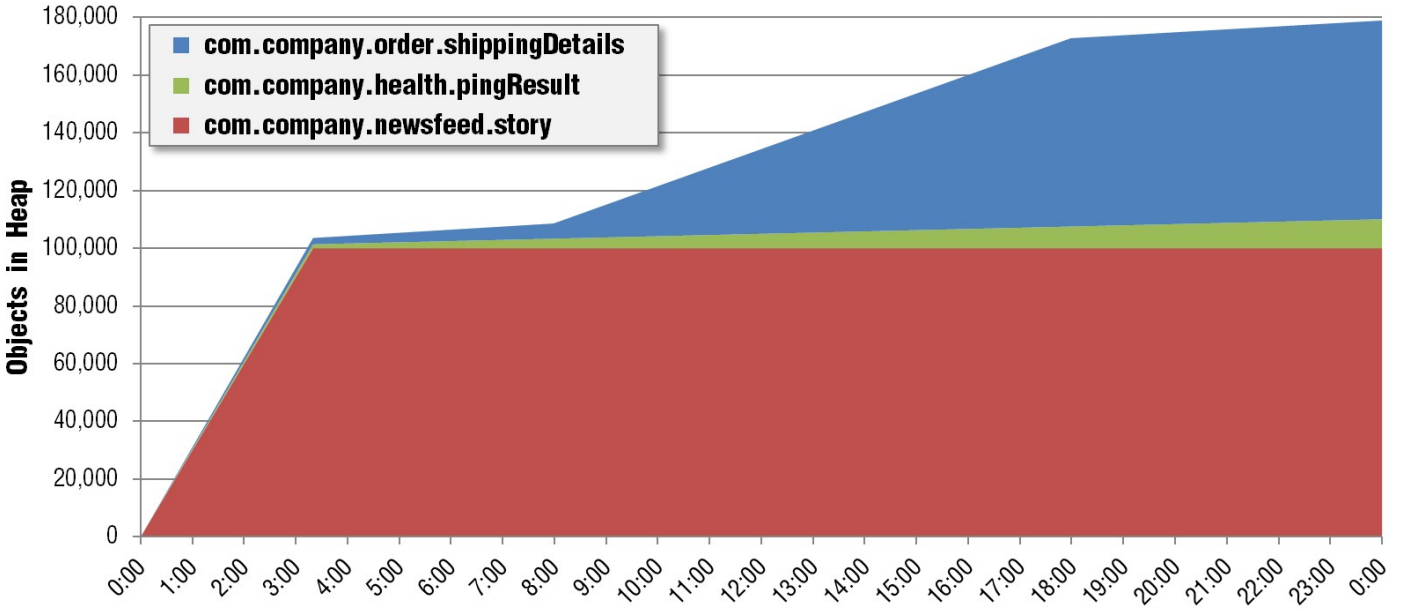
图 36

Object Name	Objects in Heap (22:00)	Objects in Heap (23:00)	% Change
com.company.newsfeed.story	100,000	100,000	-
com.company.order.shippingDetails	67,679	68,279	+0.89%
com.company.health.pingResult	9,240	9,660	+4.5%

为了真正了解该应用程序的堆使用模式，您需要在一天或一周的时间内拍摄多个快照，并对它们进行比较以确定趋势。这是一个麻烦的过程，可能导致难以解释的数据和进一步的用户中断。

更好的选择是使用更先进的 APM 产品，可以实时监控堆使用率并显示历史趋势。SteelCentral AppInternals 可以在生产中安全地做到这一点，并显示更清晰的情况。

图 37



此图表显示 24 小时前应用程序启动以来堆的前三大消耗程序的堆栈历史记录。*Newsfeed.story*（红色）似乎是某种缓存。它增长到有限的大小，然后从来不增加。所以即使它是堆的最大消耗程序，也是应用程序的有意行为，既不是问题也不是泄漏。

Order.shippingDetails（蓝色）显然是主要泄漏。在转储比较中它没有突出显示的原因是，转储被采用的夜间泄漏是最小的，但在工作时间，因为用户更频繁地执行代码，它增长更快。开发人员必须更正泄漏，否则应用程序每一两天就会耗尽内存。

Health.pingResult（绿色）在转储比较中似乎不是一个问题，但在这里我们可以看到它在一整天中稳定地泄漏。*order.shippingDetails* 泄漏被修复后，应用程序一次可以运行几个星期，*Health.pingResult* 泄漏将变得更明显，导致内存完全耗尽只要 15 天。这应该与主泄漏同时修复。

正如您所看到的，使用正确的工具，您不仅可以解决当前的问题，而且还可以快速解决尚未发现的问题。

使用正确的工具，您不仅可以解决当前的问题，而且还可以用最小的工作量迅速解决尚未发现的问题。

隐藏在表面之下，第 6 部分：排除类似泄漏的行为

如前所述，除 Java 和 .NET 堆内存之外的很多东西都可能泄漏。最后一节继续这一主题，讨论如何解决其他类型的类似泄漏的行为，这些是传统的泄漏分析工具和方法无法解决的问题。

深入了解泄漏

排除 Java 和 .NET 内存泄漏问题的工具会为您完成大部分重要工作，因此您不需要对泄漏行为有深入的了解，也能解决它们。但是，对于其他不那么传统的泄漏类型，您需要一些基本知识，才能设计和采用创新技术来解决它们。

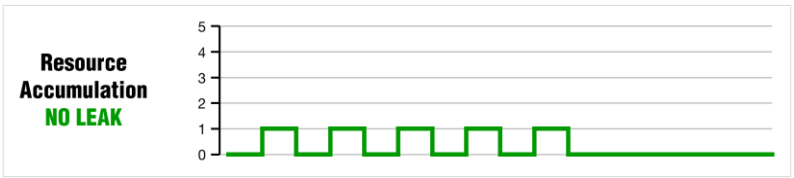
您应该还记得，当一些有问题的代码块运行时，就会发生泄漏。这种代码可能在用特定方式点击特定 URL 时运行，也可能是由特定用户访问时运行。泄漏代码可以由多个子应用程序中的多个函数共享，因此可能存在多个深层原因。它也可以在预定时间运行或在发生特定事件时响应。通常来说，部分区域的部分代码在以特定方式运行时，会加剧泄漏。

图 38 显示了两个状态（运行或空闲）的代码块。

图 38

通常，代码在运行时会使用某些资源（RAM、连接、文件句柄、线程、CPU 等），然后在完成时将资源返回，如图 39 所示。

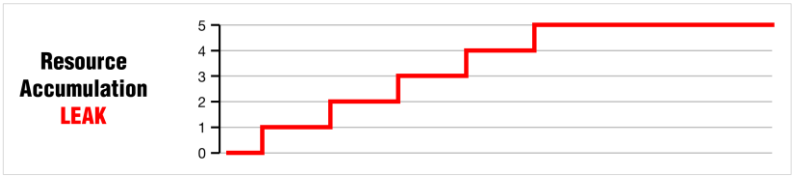
图 39



但是，如果代码存在泄漏，就不会返回所使用的资源，因此该代码的每个后续执行都会造成累积（图 40）。

寻找泄漏和类似泄漏的行为

图 40

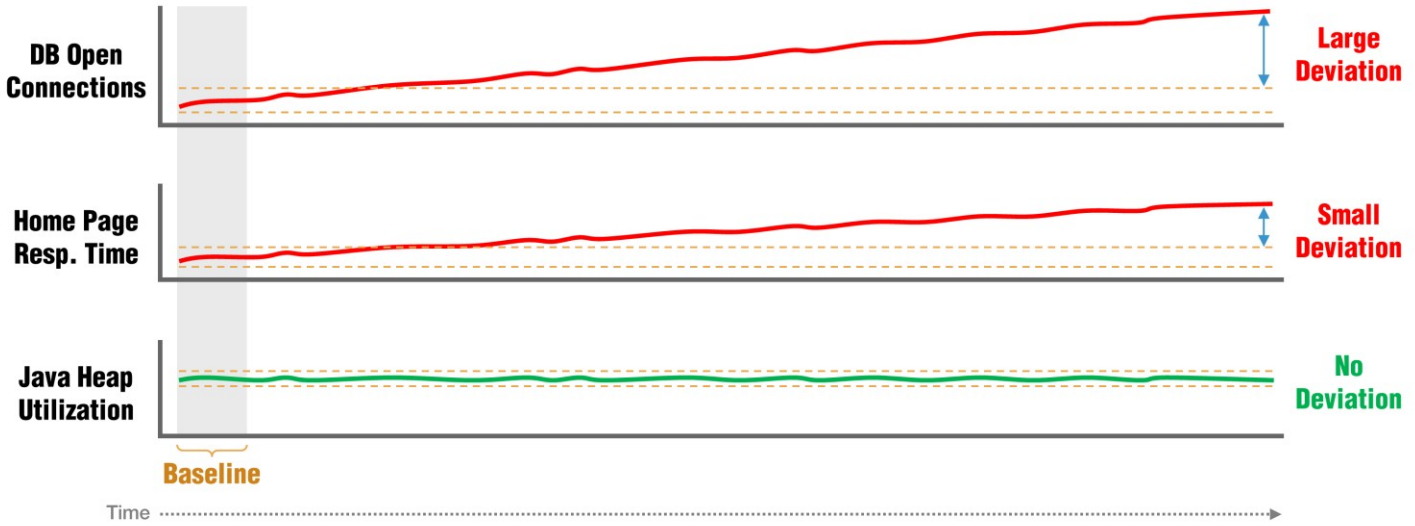


在 QA 中避免泄漏的最佳方法是让应用程序运行几个小时或几天的同一组操作的恒定负载。然后，您可以比较每个操作的性能，以及第一个小时和最后一个小时之间的应用程序和服务器资源使用率，以查看是否有任何显著变化。虽然这在概念上很简单，但有数百个可能泄漏的应用程序和服务器资源，以及可能导致泄漏的成千上万的代码。手动进行所有筛选将是一个痛苦而不切实际的任务。

SteelCentral AppInternals 提供了强大的分析功能，可以自动识别负载测试开始和结束之间数千个指标中哪些指标发生了最大变化。这将允许您快速识别哪些事务和操作出现性能下降，以及哪些资源在哪些服务器上显示泄漏行为。

在负载测试开始时，AppInternals 会观察数千个典型指标，以确定每个指标的“基线”。然后，您可以在负载测试结束时执行“偏差”分析，AppInternals 将快速显示哪些服务器对指标的更改最多。

图 41



在这里，我们可以看到主页响应时间不断变慢，这对应于不断增加的数据库连接数量。在实际应用中，您可能会看到其他偏差，这是更多形态的问题，如失控的数据库查询，这些查询会导致开放连接增加，并最终导致数据库减速。

另外，请注意，Java 堆使用率显示没有偏差，从而快速证明了此问题的根源不是 Java 内存泄漏。对问题进行故障排除时，排除组件同样重要，您不会再去寻找根本不存在的问题，这样能够节省无数小时。在实际应用中，AppInternals 会立即排除与泄漏无关的数千个潜在问题，只留下几十个相关因素作进一步调查。

在您的工具箱中拥有这样的解决方案可以为您节省数小时或数天的工作量，并有可能识别您不会想到去查找甚至不知道其存在的泄漏源。

水滴和洪水

上述工作流是找到所有泄漏的好方法，但它只会找到表现出相同偏差行为的根本原因。有时候，由于制表方法不同，根本原因具有不同的模式。找到这些泄漏的诀窍在于泄漏行为的微妙变化。这是一个特别具有挑战性的问题，所以我们需要更深入地了解它。

图 42

泄漏并不总是以恒定的速度增长，特别是当它们在生产中发生时。如果看一段时间内泄漏的趋势，通常会看到它的增长更快，或者根本不增长（图 42）。

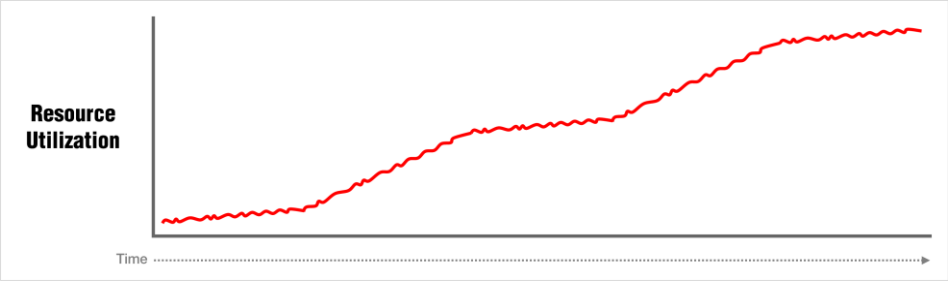
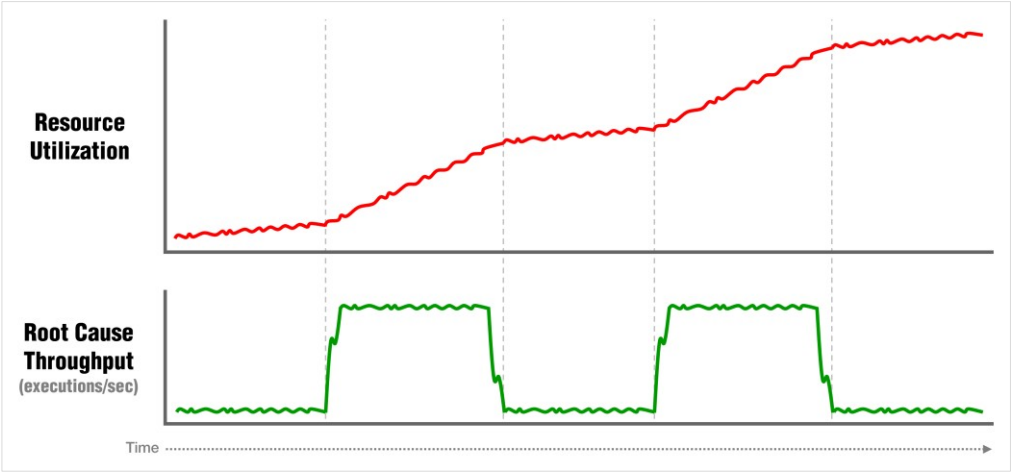


图 43



泄漏随着导致它们的问题代码的执行速率加快而增加。代码运行越快，泄漏累积越快（图 43）。

这种关系最初可能会令人困惑，因为泄漏的量通常称为“计数”，但根本原因则称为“速率”。了解这种差异对于解决棘手的泄漏至关重要。

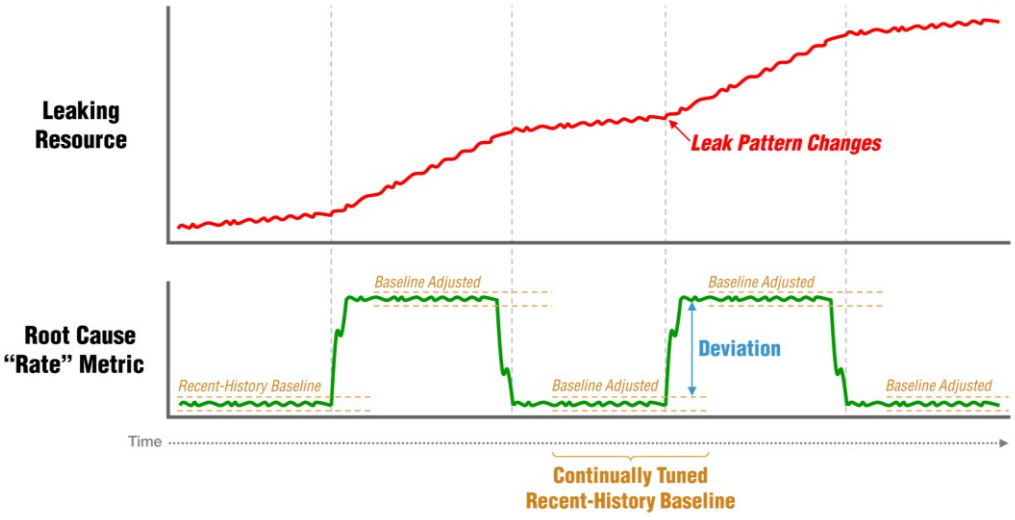
“计数”是某个时刻的瞬时值，例如“网站具有 200 个活动请求”或“DB 连接池具有 50 个活动连接”。

“速率”表示在一段时间范围内值的变化，例如“网站每秒获得 10 个请求”或“数据库每秒获取 10 个新连接”。

速率和计数通常是从相同的底层数据计算出来的，所以它们只是同一个硬币的不同侧面，但它们看起来完全不同。虽然泄漏看起来像一个斜坡，但它的根本原因通常不是这样，所以如果您希望确定导致泄漏的根本原因，就不要限制自己只寻找前者。

通过注意泄漏模式的变化并利用它们，您可以轻松识别这些棘手的基于速率的根本原因。

图 44



无需将特定的时间段指定为基线，只要利用 SteelCentral AppInternals 的默认行为，就能根据最近的历史记录持续调整其基线。然后，您需要做的就是在泄漏模式更改后立即进行有针对性的偏差分析。

通过使用这种有针对性的方法，导致泄漏行为发生变化的因素将立即变得明显。虽然这个问题非常复杂，但 SteelCentral AppInternals 的应用使解决方案变得很轻松（也就是，从“干草堆”中找到针）。

排除泄漏问题

现在，您已经很好地了解了什么是泄漏，并了解了一些找到泄漏的方法，让我们详细了解您的应用程序中存在泄漏或泄漏行为的每个常见迹象。

您的应用程序会逐渐变慢，需要定期重新启动才能解决。

在这种类型的泄漏中，随着泄漏的恶化，对应用程序的性能或行为的影响也会加剧，因为泄漏资源存在竞争现象。这可能是由于您认为已经完成的事务或操作实际上仍在后台运行。例如，未处理的异常可能会使事务陷入一个无止境的循环，因此即使最终用户可能看到错误并认为请求已经结束，一些代码实际上仍然在后台运行。

对于这种类型的泄漏，您通常会看到特定事务或事务子部分（例如特定的下游 Web 服务调用或数据库查询）的并发执行逐渐增加。您还会看到关键资源（如 CPU 或磁盘 I/O）的使用率增加，这是最终发生争用的地方。要解决这个问题，您需要增强代码以更好地处理异常，或检测失控事务并在超时后终止它们。

您的应用程序正常运行几天或几周，然后突然开始变慢，需要重新启动才能解决问题。

如果您没有主动找到问题根源，故障就可能来得猝不及防，因为泄漏开始时不会对应用程序的性能或行为造成负面影响，直到资源几乎（或完全）饱和。这种类型的泄漏通常存在于 Web 服务器线程、DB 连接池连接、文件句柄等资源上。当需要时，资源被打开，但是在完成后资源未关闭 — 有时是因为未处理的连接或意外的代码路径阻止了正常关闭。

这些资源没有被积极使用，因此对响应时间或 CPU 使用率没有影响，但一旦资源饱和，请求这些资源的事务就会停止或失败。要解决这个问题，您需要增强代码以更好地处理异常或意外的代码路径，以确保未使用的资源返回到原来的池中。另一个方案是例行检查背景内务管理，寻找这些泄漏，并将其返回到原来的池中。

您看到一些资源的使用率随时间增长，需要重新启动才能解决问题。

这可能是由于以前的泄漏，但是需要基于泄漏资源本身而不是泄漏的最终用户效应来识别。您可以使用相同的方法对其进行故障排除。

您没有对应用程序的代码或环境进行任何更改，但其行为随着时间推移发生了变化。

这是对任何类型的泄漏或类似泄漏行为的影响的高层次抽象。您的最终用户可能不会报告他们认为存在泄漏的应用程序，但他们会说：“最近似乎很慢”或“它崩溃的次数比以前更多了。” 这样的说法表明可能存在需要解决的泄漏。

您的应用程序会逐渐变慢，但重新启动没有帮助。

这种类型的问题是*“类似泄漏行为”*这个术语存在的原因。虽然*严格来说*，它们不是泄漏，但它们对性能有类似的影响。其典型原因是数据存储随时间而增长，但缺少针对该存储的后续查找的索引。

最常见的例子是数据库表不断增长，因此对“表扫描”的查询会逐渐变慢，并且由于缺少索引而需要更长时间。这通常会在恒定负载的长负载测试中显示出来，其中插入和读取都发生在同一个表上。它可以通过添加所需的索引来轻松修复。

一个更少见的例子是新文件不断添加到同一个文件夹，例如每天增长的大型 PDF 或图像存储库。一些文件系统没有文件夹中的文件的索引，因此它们必须执行相当于数据库表扫描的操作以查找您要查找的文件。这通常是一个即时操作，但是当单个文件夹中有成千上万个文件时，可能会导致延迟几秒，特别是在并发查找时。通过创建一个嵌套目录结构，将文件分成数百或数千个子文件夹来纠正这个问题。从概念上讲，您会将文件 “Bob’s Report.pdf” 存储在 \reports\b\bo\bobs\bobsr\ 文件夹中。在实际应用中，目录名称会基于散列算法，所以您可以确切知道在哪里查找它，而不必扫描成千上万的文件

将 SteelCentral AppInternals 添加到您的库中，以超越传统 Java 和 .NET 堆分析的限制，并识别应用程序中任何类似泄漏的行为，以便为您的用户确保一致的性能和可用性。

识别泄漏、消除问题并提供高性能应用程序

泄漏是非常常见的，由于种种原因和几乎可以在无限多的地方发生，可能很难发现。将 SteelCentral AppInternals 之类的解决方案添加到您的库中，您将可以超越传统 Java 和 .NET 堆分析的限制，识别应用程序中的任何类似泄漏的行为，从而确保为最终用户提供一致的性能和可用性。

结语

本指南涵盖的场景可能听起来像难懂的极端情况，但在实际应用中，它们很常见。只是因为您没有看到，并不意味着它不存在；因此这个系列的标题称为：*隐藏在表面之下*。为了确保您不会忽略或误解问题，请遵循以下要点：

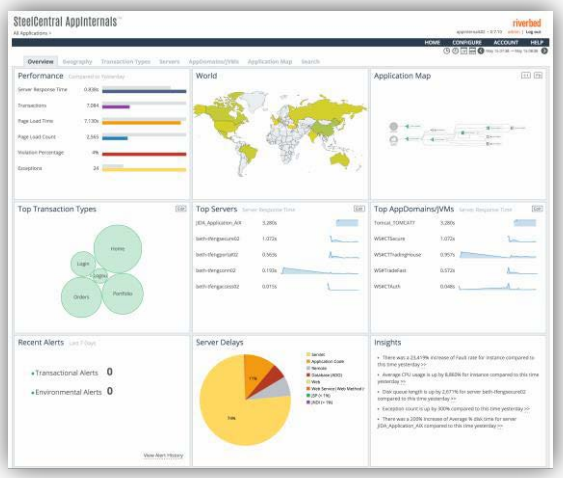
- 始终注意“平均值的缺陷”，它会掩盖问题。确保您具有足够的监控解决方案以解决该问题，并且您的故障排除工作流可以有效地利用该数据。
- “最慢的事情就是慢”的原因可能不是应用程序整体慢速的原因。对 APM 使用大数据方法，并整体分析您的事务以确定延迟的总体原因，首先针对这些领域进行优化。
- 运行缓慢的代码可能不是代码本身的原因。要了解您的代码运行的环境，并记住它实际上依赖于 JVM/CLR、操作系统和它所运行的虚拟机管理程序。如果任何这些依赖关系变得饱和，您的代码就会受影响。
- 学会区分症状和根本原因。不基于初步发现草率下结论。退一步，从不同的角度看问题，确认您的理论，以尽量减少浪费在干扰因素上的时间和工作量。
- 不要作出假设。验证一切。
- 确保您获得了预期的负载。确认负载是否按照您认为应采取的方式、按照设计的层级向下传播。在验证输入之前，您无法解释输出。在到达数据源的途中发生的停顿和从数据源返回时发生的延迟是不同的问题。
- 您需要获取全天候所有事务的详细信息，以确定延迟的总体原因，不要被所谓的“代表性”抽样方法误导。如果您只获取一部分事务，则将只解决一部分问题。

利用 SteelCentral AppInternals 掌控应用程序性能

持续监控应用程序。始终跟踪所有用户的所有事务，以获得最全面的端对端应用可视性和分析，从而实现快速解决问题和持续改进。提供全天候高性能应用。

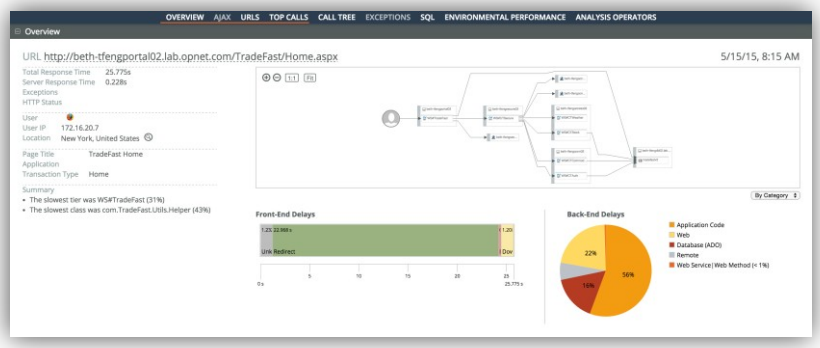
预先了解性能问题

您只需在回答中单击两次，即可了解性能问题。通过一个基于 Web 的互动仪表板来监控用户体验、应用程序、基础结构和关键业务事务。



重建及诊断所有事件

跟踪从用户设备或浏览器到后台应用程序的每项事务，并获取实时系统指标。详细地重建并分析事件，以修复代码、SQL、基础结构或远程调用。

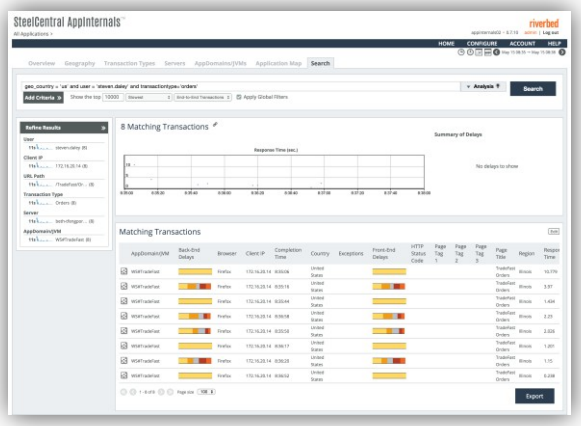


主动提高性能

使用简单的查询分析数十亿指标，以揭示错误、发现异常、挖掘业务见解或规划容量。探索数据集并持续提高性能。

在应用程序生命周期中协作

在开发、QA 和生产阶段衡量影响以及分享性能见解。详细的诊断可帮助开发和支持团队修复问题，而无需重现这些问题。



综观全局

与 Riverbed SteelCentral Portal 集成，获得各个应用、网络和架构的端对端可视性及分析。获得其他应用性能管理 (APM) 工具无法实现的全局。

免费试用 APP！

前往 www.appinternals.com 注册以免费试用 AppInternals。如需通

过电话联系我们，请拨打 1-87-RIVERBED。

作者：

Jon C. Hodgson 是 Riverbed Technology 的 APM 主题专家。十多年来，Jon 帮助世界各地的数百家组织优化了其关键任务应用程序的可靠性和性能。凭借系统管理、网络、编程和应用程序架构的背景，Jon 利用多学科故障排除方法分析和解决各种各样的问题，从典型的代码/sql 低效到更复杂的问题，涉及资源饱和、网络效应和低级别操作系统/虚拟化瓶颈。在痴迷于如何更快地解决问题的工作之余，他喜欢开着拖拉机在他密苏里州的家里挖东西。

现场指南编译：

Mason Coffman 是 Riverbed 平台营销团队的内容营销经理。在本职位的工作中，他开发了激励人心的内容，为计划和完成战略性 IT 和业务计划的 Riverbed 客户提供了有价值和一致的信息。Mason 拥有近十年的 B2B 营销经验，涵盖各种主题，包括数字学习内容、企业移动性、云计算、应用程序和网络性能管理等。

Riverbed公司简介

Riverbed 是应用性能基础设施的领导者，年收入超过 10 亿美元，为混合型企业提供最全面的平台，确保理想的应用性能，持续的数据可用性，并主动监测和解决性能问题，不会影响业务性能。Riverbed 助力混合型企业将应用性能转化为竞争优势，最大化员工生产率，借助 IT 创造新型运维灵活性。Riverbed 目前拥有 28000 余家客户，其中包括 97%的财富百强企业和 98%的福布斯全球百强企业。更多内容敬请浏览 www.riverbed.com。