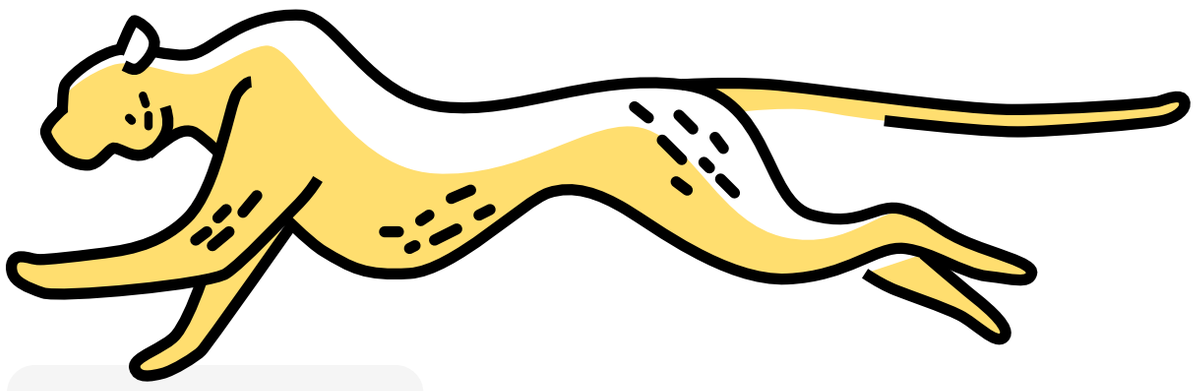
码农 the code maker

31

打破性能瓶颈



为性能做规划

快速拎清不同种类的性能测试

Web 性能核心优化点

Java 性能分析工具箱

好产品让用户为自己尖叫

算法之于性能

强迫性调优障碍

用 Numba 实现极速数据处理

iTuring.cn

目　录

书单

动手

成书手记

编者的话

编者 / 刘敏

相比早一代人考试驾照的苛刻条件，现在的考试要轻松得多，我们不再需要学习汽车的机械原理，达到对车辆简单维修和保养的水平，就能拿到驾照。

为了推动事物的普及，世界貌似正在朝着黑盒化的方向发展。如今的IT技术，同样变得更加容易使用，程序员甚至不需要知道内部细节，也能够开发出业务应用程序或是搭建基础设施。但是，性能调优需要理解系统内部的架构。对于程序员来说，在核心技术“黑盒化”的趋势下，性能是一块能永远发挥作用的天地。

性能调优问题不仅贯穿于开发的各个阶段，而且是用户体验和商业指标的重要影响因素。因此，在程序、系统、工程开发之前就应该为性能做规划，确定预期的性能指标。除了性能指标，算法的效率和外部系统的性能瓶颈也是影响性能的主要因素。在处理性能问题方面，虽然分析和调优是主角儿，但“性能测试”是不能缺少的“配角儿”。如果将调优的结果直接用于生产环境，会有一定风险，因此通常会在验证环境中验证调优的成果。

本期《码农》意在帮助那些对性能问题还不太关注，以及对性能一知半解的程序员快速了解性能调优方面的知识。“专题”一栏的文章包括了性能问题处理的各种环节及行家经验。基于这些理论，“践行”专栏给出了实操性的指导，帮助移动端应用的开发者应对网络条件较差、内存不足的情况，以提供流畅的用户体验。

谈到“如何打造良好的用户体验，培养持续忠诚的用户”，就必须提到O'Reilly出版社Head First系列图书策划人之一的Kathy Sierra。Head First系列图书的成功，无疑是Kathy推崇的“成就用户”思维的有力证据。所以，“人物”专栏特别提供“图灵访谈”对Kathy本人的访谈实录，帮助大家更好地理解“人们并不想因为擅长使用某个产品而变得了不起，他们想要那种因为使用了某个产品而带来的成就感”。

面对世界的冷漠和经济社会的薄情，我们当中的大部分人会无奈地麻痹自我，让自己像木头人一样，不去想，不去看。“成长手记”专栏的文章出自一位图灵的读者，他用自己的真实经历告诫每一位还未来过“此处”的追梦人，警示习惯了“此处”的木头人，纪念自己已逝的三年纯真。

Begin our story! ■

专题：打破“性能”瓶颈

为性能做规划

作者/Christian Antognini

资深数据库专家，从1995年就开始致力于探究Oracle数据库引擎的工作机制。长期关注逻辑与物理数据库的设计、数据库与Java应用程序的集成、查询优化器以及与性能管理和优化相关的各个方面。目前任瑞士苏黎世Trivadis公司首席顾问和性能教练，是OakTable网站核心成员。

如果仔细分析程序开发各阶段所需开展的工作，你也许会注意到每个阶段都有性能要求。即便如此，在实际开发过程中，开发团队还是会时常忘记性能要求，直到性能问题浮现出来。而那时也许为时已晚。因此，接下来的内容将从性能的角度出发，介绍在下一次开发应用程序时不应该再忽视的内容。

需求分析

简单来讲，需求分析（requirements analysis）就是确定应用程序的主要目标以及藉此期望达成的目的。进行需求分析之前，通常要对多个利益相关方进行调研。这一步十分必要，因为单独一方不太可能确定所有的业务需求和技术需求。由于需求的来源不一，因此必须对需求进行仔细分析，尤其需要找出不同需求间是否存在潜在冲突。在进行需求分析时，不仅要关注应用程序需要提供的功能，仔细确定这些功能的使用率也是至关重要的。对于每一个具体的功能，需要预估与之交互的用户数量、用户的使用频率以及每次使用时的预期响应时间。换句话说，你必须确定预期的性能指标。

这些性能需求不仅仅作为核心要素贯穿于应用程序开发的各个阶段，稍后也可将其作为定义服务级别协议以及制定容量规划的基础。

服务级别协议

服务级别协议（SLA）是用来明确服务提供商和用户之间关系的契约。它描述的内容包括服务项目，其在运行时间和停机时间的可用性、响应时间、客户支持水平，以及一旦服务提供商无法履行协议时相应的处理方式。

只有在能够验证响应时间的情况下，才能根据响应时间制定服务级别协议。这需要定义清晰的、可测量的性能数据以及与之相关的目标。这些性能数据通常被称作关键性能指标（Key Performance Indicator，KPI）。最理想的情况是使用一种监控工具收集、存储和评估这些指标数据。事实上，这样做的目的不仅是为了在某个目标没有达到时进行标识，还能为日后出具报告和制定容量规划而记录下依据。为了收集这些性能数据，可以采用两种主要的技术手段。第一种是利用监测代码（instrumentation code）的输出结果；第二种是使用响应时间监控工具。

分析与设计

架构设计师根据需求设计解决方案。开始的时候，为了定义架构，需要考虑所有的需求。事实上，对于一个需要承受高负载的应用程序，在设计之初就应考虑负载需求。当设计时用到诸如并行化、分布式计算或结果重用等技术时更应如此。例如，设计一个支持少数用户每分钟执行十几个事务的C/S应用程序，与设计一个支持成千上万用户每秒执行数百个事务的分布式系统完全是两码事。

有时需求也会通过在某一资源的使用上施加限制来影响架构。例如，如果一个应用程序用于通过低速网络连接到服务器的移动设备，那么其架构设计必须考虑能够支持较大延迟和较低吞吐量。通常，架构设计师不仅需要预见到一个方案可能出现的瓶颈，还要衡量这些瓶颈是否会危及需求的实现。如果架构设计师没有掌握足够的信息来进行这样的关键预先评估，就应该开发一个或甚至多个原型。在这方面，如果没有前一阶段收集的性能数据，将很难做出明智的决定。我所说的明智的决定是指那些能够实现以最小投资支撑预期负载的架构/设计的决定：简单方案处理简单问题，精简方案处理复杂问题。

编码和单元测试

专业开发人员编写的代码应具有下面这些特点。

稳定性：拥有应对意外情况的能力是所有软件都应具备的特性。为了达到预期质量，必须定期进行单元测试。这一点对于迭代型生命周期尤为重要。实际上，在这类模型中，快速重构现有代码的能力是不可或缺的。例如，在调用某个子程序时，如果传递的参数值超过指定范围，系统就必须能够做出相应处理而不至于崩溃。如有必要，应该同时生成有意义的错误信息。

可维护性：能够长期运行、结构良好、已文档化的可读代码比没有文档化的糟糕代码维护起来要容易得多（维护费用也更低）。例如，有人将多个操作写成单独一行晦涩的代码，这样的开发人员其实选错了展示才华的方式。

运行速度：代码应该进行优化，以期尽可能提高运行速度。在预期负载很高的情况下更应如此。代码应该具有可伸缩性，进而能够利用额外的硬件资源应对用户或事务的不断增加。例如，应该避免不必要的操作、串行程序，以及低效或不适合的算法。然而，一定不要掉进过早优化的陷阱。

精明的资源利用：代码应尽最大可能利用可访问资源。注意，这并不总是意味着使用最少的资源。比如，应用程序使用并行化操作比串行化操作要消耗更多的资源，但是有时候并行化也许是解决苛刻负载的唯一途径。

安全性：毋庸置疑，代码要拥有保证数据机密性和完整性，以及对用户进行验证和授权的能力。有时，不可抵赖性也是需要考虑的问题。例如，可能需要用到数字签名来防止终端用户否认通信或合同的有效性。

可检测性：检测的目的有两方面。其一，更易于分析出现的功能问题和性能问题（即使是精心设计的系统也无法避免这些问题的出现）；其二，有策略地添加代码以提供应用程序的性能信息。例如，通常情况下，添加用以获取某一操作所耗时间的代码非常简单。这是一个验证应用程序是否能够满足必要性能需求的简单有效的办法。

不仅这些特性彼此之间确实存在一些冲突，而且预算通常是有限的（有时甚至非常有限）。因此，我们通常有必要在这些特性之间做个优先级排序，在其中找到平衡点，以便在有限的预算下实现预期的需求。

集成和验收测试

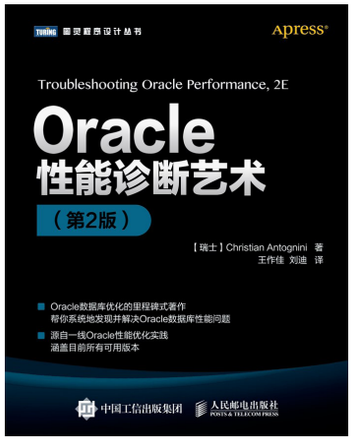
集成和验收测试的目的是验证应用程序的功能需求、性能需求以及系统稳定性。性能测试和功能测试同等重要，这一点无论如何强调都不过分。从各方面来看，一个性能差的应用程序和没有实现功能需求的应用程序一样糟糕。在这两种情况下，应用程序都是无用的。然而，只有明确定义过性能需求才有可能去验证它。

缺少正式的性能需求会导致两个主要问题。第一，极有可能在集成和验收测试阶段没有执行严格的、有条不紊的压力测试，这样，应用程序就会在不知道是否能够支持预期负载的情况下交付生产；第二，就性能而言，无法明确什么样的表现可以接受，什么样的表现不能接受。通常，只有在极端情况下（也就是说，性能非常好或非常糟糕），不同的人才会达成统一意见。如果无法达成共识，冗长、恼人、徒劳的会议以及人际冲突就会随之出现。

在实践中，设计、实现和执行良好的集成和验收测试来验证应用程序的性能表现并非易事。要想取得成功必须面对下面三个主要挑战。

设计压力测试时应该考虑能够产生典型的负载。对此主要有两种方法：一是让真实的用户做真实的工作；二是用工具来模拟用户。两种方法各有优缺点，应该根据具体情况进行具体分析。某些情况下，两种方法可同时用于对不同模块进行压力测试，或者用两种方法进行互补。

要产生典型的负载，就需要典型的测试数据。不仅数据行的数量和大小要符合预期的量，数据分布情况和数据内容也应与真实数据一致。例如，



《Oracle性能诊断艺术（第2版）》是Oracle数据库优化专家Christian Antognini的一部继往开来的里程碑式著作。书中的最佳实践和诸多建议全部来源于作者在实战一线的丰富积累，不仅简单实用，而且发人深省，堪称一座“宝库”，适合各层次读者研读和发掘。

如果属性中含有城市名称，那么用真实的城市名称就比用像Aaaacccc或Abcdefghij这样的字符串要好得多。这样做很重要，因为很多情况下应用程序和数据库都会因为不同的数据导致不同的表现（例如，索引或作用于数据的函数）。

测试的基础设施要尽可能与生产环境的基础设施保持一致。这对于高度分布的系统和需要与许多其他系统协作的系统来说尤其困难。

在顺序型生命周期模型中，项目开发接近尾声时才进入集成和验收测试阶段。如果导致性能问题的重大系统架构缺陷此时才被发现，问题会比较棘手。为避免这样的问题，在编码和单元测试阶段也应该进行压力测试。注意，迭代型生命周期模型不存在这种问题。事实上，根据“迭代型生命周期模型”的定义，每一次迭代都应该执行压力测试。■

快速拎清不同种类的性能测试

作者/小田圭二等

日本Oracle株式会社咨询部门经理。在解决性能问题方面有着丰富的经验，著有《图解OS、存储、网络：DB的内部机制》(絵で見てわかるOS/ストレージ/ネットワーク～データベースはこう使っている)、《图解Oracle的机制》(絵で見てわかるOracleの仕組み)等多部著作。

在处理性能问题方面，虽然调优是主角，但是“性能测试”是不可或缺的。要让调优顺利进行下去，最重要的工作就是通过测试来验证。不过，如果将调优结果直接在生产环境中验证，会有一定风险，因此通常是在验证环境中获得性能测试的结果，来验证调优的成果。在通往调优专家或性能专家的道路上，性能测试可以说是最重要的要素。

项目工程中的性能测试

在系统开发项目工程中，性能测试实际上主要在系统测试阶段执行。这是因为性能测试原则上是为了确认“系统在生产环境中运行时是否会有性能上的问题”，所以它是在完成集成测试后，确认系统能在生产环境中正常运行之后的阶段执行的。

即便如此，如果认为在进入系统测试阶段之前不需要考虑性能测试，那就错了。

测试的实施周期

在通常的开发、搭建项目的各个阶段，性能测试有几个变种。

作为判断性能的基准，最重要的测试就是（狭义的）“性能测试”。该测试也决定了系统能否发布。除此之外的其他测试则是为了更有效地运营项目或者其他目的而实施的。

“Rush Test”“压力测试”等不是根据测试目的来定义的，它们表示的只是测试的执行方式，因此在什么时间执行是由测试目的决定的。压力测试根据目的的不同，可以分为“性能测试”“临界测试”“耐久测试”等类型，具体做法都是在短时间内向系统发起大量的访问，以此来测量结果。一般会再现多个同时在线的用户的使用情况。在某些情况下，批处理时大量数据的流入也属于这一类。

下面将对各个种类的测试进行更详细的说明。

性能测试的种类

- 狭义的性能测试

这个是最为重要的测试，目的是判断是否能达到要求的性能。

实施时间　在系统测试阶段实施。前提是系统测试的功能部分已经全部通过测试，确定之后不再需要系统变更，并且已经做好了进行运用管理和批处理操作的准备，包含此动作的性能测试也已经准备好。如果没有满足这些前提条件，那么性能测试完成后，系统性能也有可能会发生变化，所以请尽量在上述前提下执行性能测试。

测量项目　性能测试需要确认以下 3 个性能指标是否均已达成。

1. 吞吐（处理条数/秒）

2. 响应时间（秒）

3. 同时使用数（用户数）

此外，确认服务器日志以及压力测试工具的记录，同时确认在测试中是否出现了错误信息。如果出现了错误信息，那么这个处理就有可能在执行过程中被跳过了，也就没有完成充分的性能验证。这种情况下就需要首先消除错误，然后再次执行测试。

如果已经定义了系统运行时资源使用率的上限（例如，遵守CPU使用率在50%以下等），则还需要一起确认资源使用率。

- 临界测试（临界性能、回退性能、故障测试）

前面提到过，性能测试用于判定系统能否发布，而除此之外还存在从其他角度进行判定的压力测试。

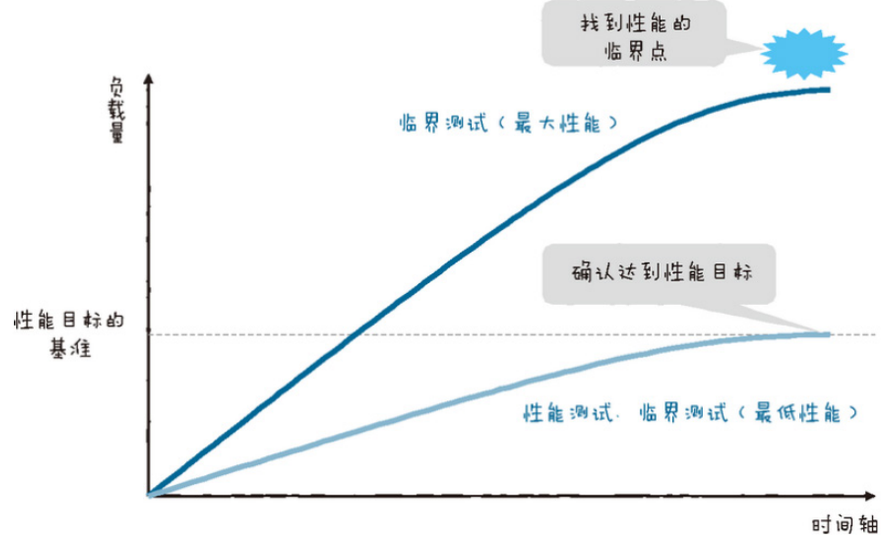


图　临界测试的种类

1. 临界测试（最低性能）

测试是否达到了性能目标的基准。临界测试作为性能测试实施之前的预实施，不用进行太严密的用户场景定义与资源统计，其目的仅仅在于对能否处理预计的处理条数进行简单的确认。另外，在实际执行性能测试的时候，可能会出现负载对象的性能不足、施加负载的一方性能不足、结构错误等情况，临界测试的另一个目的就是发现这些问题。

若无需进行大规模的准备工作就能立即执行的话，应该在系统完成集成测试之后或之前执行一下。

实施时间　在大规模地正式实施性能测试之前，需要与各相关部门进行协调。有时候性能测试的实施时间是有限制的，因此测试负责人或服务器管理者应该事先确认好自己责任范围内的测试是否能正常进行。

测量项目　不需要花费太多精力，只要进行最小限度的确认就可以了。

2. 临界测试（最大性能）

这个测试的目的在于，在负载超过性能目标的情况下，把握系统承受程度的上限，以及当时的情况和瓶颈。

如果需求定义中没有要求进行临界性能的测量，那就没必要实施这个测试了。不过，在实际向客户报告测试结果的时候，可能会被问到“超过这个负载的时候能正常运行吗”“验证过流量控制和超时功能了吗”等问题。系统发布后，在超负载的情况下，如果流量控制、超时、运维监控的阈值检测机制等不能按照预想的那样正常运行，就会出问题，有时甚

至会被当成残次品。为了消除这些隐患，我们要执行临界测试。要想通过验收，性能测试是必不可少的一项工作。而临界测试则非如此，而是项目方为了维护项目成果、避免风险而自发实施的测试。

实施时间　在系统测试阶段顺利通过性能测试后，如果有足够的时间就进行此项测试。这个时候可以进行两种测试。第一种是偏向基础设施的测试，在施加了与生产环境相似的流量控制的状态下，确认流量控制功能能否正常运行。另一种是在不进行流量控制的状态下，确认系统所能处理的上限以及这个时候的情况和瓶颈原因。在进行了这两个测试后，如果能基于明确的记录，对系统在超负载时的情况进行说明，客户一定能认可这个报告。在某些情况下，如果很好地执行了后面提到的“基础设施性能测试”，也可以不实施这里的第一种测试。

如果系统是横向扩展结构，那么也需要验证横向扩展结构下达到临界负载时的运行情况。理想情况下，在达到最大负载时，AP 服务器和 DB 服务器的 CPU 使用率会达到 100%，或者网络带宽的使用率会接近 100%，像这样施加负载让资源达到上限的话，那么作为临界性能测试的测试结果就可以说足够了。如果资源使用率没有达到 100% 就已经到了性能界线，吞吐也不能继续提升，或者增加负载也只是导致响应变差，那一定是哪里的设置存在瓶颈，必须搞清楚原因。

在从长远的角度计算系统使用人数的增加量以及针对这种情况的估算指标时，除了在纸上计算之外，还可以一并参考临界测试中阶段性的负载增加以及资源使用量。特别是与公司内部系统不同，在互联网系统中，可能会出现用户突然增加的情况，因此为了建立估算战略，事先进行测量是非常重要的。

测量项目　实施临界测试的方式是一直施加负载直到达到最大吞吐。使用压力测试工具增加并发度来生成负载的情况下，并发度增加到什么程度也是一个基准。此外，如前所述，为了判断资源是否用尽，也要一起参考服务器的CPU使用率。

3. 回退性能测试

回退性能测试也属于一种故障测试。在那些为了确保可用性而使用了冗余结构的系统中，我们需要验证当其中一部分处于停止状态时，是否能获得预期的性能。如果存在回退情况下的性能需求定义，就要进行这个测试。即使没有进行需求定义，如果在生产环境中运行时发生回退，导致没有获得预期的性能，也会很棘手，所以要尽可能地把这个测试加入到项目的验证计划中。

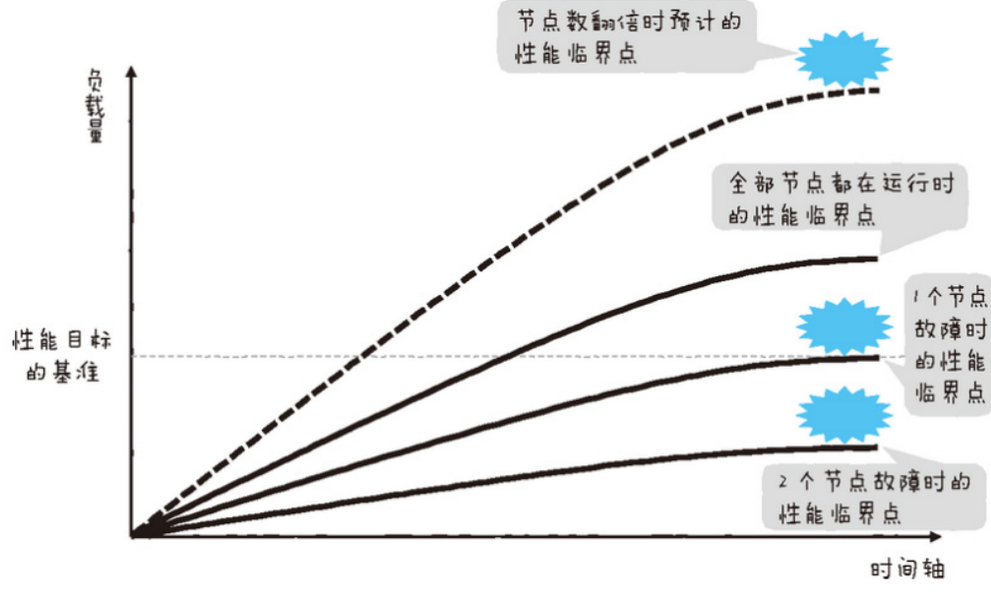


图　回退性能测试

实施时间　分为两种情况，一种是在系统测试阶段的性能测试结束后执行，另一种是在后面将会提到的基础设施性能测试的过程中执行。如果冗余结构以及可用性功能在系统基础设施中就完成了，并且能够与搭载的应用程序剥离开来，那么只需在基础设施性能测试中执行回退性能测试就可以了。其他情况下，由于要让应用程序在类似于生产环境的环境中运行来进行测试，因此就要在性能测试之后来执行了。

不仅要对一部分处于停止状态的结构进行性能测试，也要对运行过程中停止或者再次启动时响应时间的变化进行确认。

测量项目　通常的检验方法是，作为测量指标，通过吞吐来确认最大性能，以及通过响应时间和是否发生错误来确认行为的变化。

故障测试

故障测试实际上并不属于性能测试的种类，一般被归类到集成测试或系统测试中执行的故障测试。不过，在发生与性能相关的故障时，需要结合压力测试一起实施，而且故障测试与这里介绍的其他测试手法相近，所以笔者就在这里进行解说了。

故障测试的目的是触发高负载时会出现的故障，判断那种情况下系统的行为以及错误恢复是否与预计的一样。特别是那些会出现高负载但又追求高可用性的系统，故障测试是必需的。

实施时间　如果作为基础设施可以分离开来的话，可以在集成测试和系统测试的基础设施上进行故障测试。如果不能分离，则可以在性能测试

和临界测试等完成后，基于已经确立的性能测试和临界测试的手法来进行测试。

需要注意的是，如果在一般的性能测试和临界测试等场景中直接执行的话，有可能不能触发目标故障点，而是在别的地方出现瓶颈，导致不能触发希望出现的性能故障。这个时候，需要重新考虑负载场景、系统结构和设置，直到可以触发希望出现的性能故障。

测量项目　首先着眼于服务器以及负载终端的错误，确认那个时候的吞吐以及平均响应时间，将其作为参考指标。

- 基础设施性能测试

在最近的系统搭建中，大多会将应用程序和基础设施分离开来，分别制定搭建计划，然后在集成测试或系统测试中才将其汇合到一起。基础设施中包含中间件（DB 或 AP 服务器）的情况也很多。此外，基础设施作为基础，有专业负责人或供应商执行别的调度计划和检查，笔者认为，采用和应用程序不同的流程更容易推进。综合基础设施和私有云等一开始往往不能准备好应用程序，所以有时就需要在没有应用程序的情况下进行基础设施的发布及基础设施测试。

基础设施性能测试的目的与必要性

基础设施性能测试是与应用程序分离开，从基础设施的观点来进行的性能测试。基础设施性能测试的目的是防止在后面的系统测试阶段中基础设施出现性能问题导致返工或计划变更。基础设施搭建团队通过预先进行负载试验，来尽量规避风险。

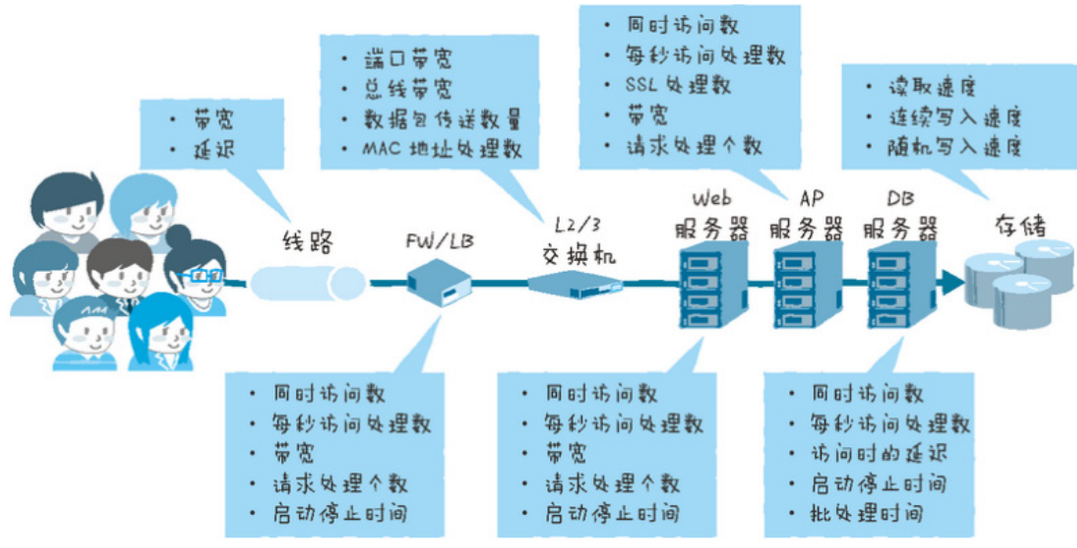


图　基础设施性能测试的主要验证项目

只要没有作为验收条件进行规定，基础设施性能测试就不是必需的。不过，如果在系统测试后的性能测试中才发现基础设施存在性能问题，返工成本就会很大，而且也有可能会影响到计划。为了不出现这样的情况，强烈建议在基础设施方面进行与实际生产环境相似的性能测试。

实施时间　基础设施搭建结束后，在基础设施的集成测试中故障测试完成之后实施。

在基础设施上进行性能测试，其最大课题就是在应用程序还没有完成的状态下如何预估出需要的性能，以及怎样使用作为样本运行的应用程序。

测量项目（样本应用程序）　评价的对象不同，测量的应用程序也不同。

1. 从网络到Web服务器的基础设施性能测试

在Web服务器上部署静态资源，然后对其发起大量访问就可以了。

2. 包含依赖于会话的处理在内的基础设施性能测试

使用依附于应用程序的样本程序。如果是WebLogic的话，就经常使用PetShop或MedRec等作为样本项目。这些程序会进行包含登录在内的会话管理，因此使用负载均衡器或Web服务器来进行会话和cookie的处理，然后分发，这样作为性能测试来说就足够了。

3. 使用数据库或缓存网格（Cache Grid）或KVS的情况

这些服务都不是直接从外部来访问，而大多是从AP服务器来访问的，因此很多时候不需要经过全部的基础设施。这种情况下，建议一个一个地单独验证。各种测试工具应该都已经准备好了。

在进行数据库的基础设施测试时，为了按照事先想好的处理流程编写脚本，或者预设好实际的运行步骤并添加负载，使用Oracle Real Application Testing或者Oracle Application Testing Suite的Load Testing Accelerator for ORACLE Database也很方便。

在基础设施性能测试中，特别是与存储和数据库相关的测试，需要准备好与生产环境相同的数据量来验证。另外，测量备份和恢复所需的时间以及运维批处理能否在一定时间内完成也应该包含在基础设施性能测试中。

基础设施性能测试的性能目标

在基础设施性能测试中，除了样本应用程序之外，另一个课题就是应该以什么样的性能目标作为基准。在讨论目标值时，一般按照下面的顺序进行。

① 提出性能目标信息

让应用程序开发部门提出严谨的性能目标信息。具体包括负载均衡器和Web服务器上必需的同时连接数、每秒的请求数、网络流量（bps）等。DB基础设施的情况下，只有简单的处理数和同时访问数是不够的，如果不能在应用程序这里更进一步，让其提示与业务相同级别的SQL的同时执行信息，就不能完成充分的基础设施测试。

如果没有很好地定义这些指标就进行应用程序设计，就会在开发时忽视性能，因此应该对应用程序开发部门明确地提出要求。

② 自己预估

如果不能获取上述信息，或者对应用程序开发部分预估的信息不够放心，就需要自己来预估，顺便进行验证。

下表汇总了预估目标信息所需的项目与知识。

表　计算基础设施性能测试的性能目标

项目	说明	常见范例	计算时是否需要			
			带宽	吞吐（请求/秒）	同时访问数（用户人数）	同时连接数
①顾客总数	能够想到的使用人数的最大值。内网的话就是公司员工数。互联网网站的话要根据市场来预测	3 万人（大公司的内网）；50 万人（中型规模的互联网站）	●	●	●	●
②高峰期 1 小时内的顾客集中率	在特定的 1 小时里，上述总数的顾客同时访问的比例最大是多少	70% = 内网的全体员工必须同时使用的处理；3%= 大型 EC 网站在促销期间会员集中访问	●	●	●	●
③顾客的平均思考时间	浏览了某个页面后，到浏览下一个页面前，中间的思考时间的平均值	5 秒 = 文章很短也不要输入的页面上，老用户访问下一个页面所需的时间； 120 秒 = 页面中有很多输入项目，也需要阅读说明的情况下所需的时间	●	●	●	●
④每个顾客浏览的页面数	从登录到浏览、提交、退出，中间一共有几步？互联网网站的话，要将那些浏览了 1 个页面就离开的用户也考虑在内，计算平均值	5 页 = 书面申请等； 7 页 =EC 网站的商品查找（有的用户会浏览超过 50 页，但大部分用户浏览 1~2 页就会离开）	●	●	●	●
⑤各个页面的平均内容数	每个页面使用的内容（图像、CSS、JS、XML）数的平均值	大部分是 4~50。根据页面的实际制作情况会有所不同	●	●		●
⑥平均内容大小	包含 HTML、图像和 PDF 等内容在内的平均文件大小	一般来说 HTML/ CSS/JS= 大概 10 KB； 图像文件 = 大概 50 KB； PDF 文件 =2MB	●			
⑦内容缓存比例	用户再次访问或者多次访问同一个页面时能够在缓存中获得内容的比例	内网 =70%； EC 网站 =10%	●			
⑧平均每个顾客的最大同时连接数	平均每个客户端终端设备（浏览器）等对服务器的最大连接数	普通的 IE11 用的网站=6 个；使用 WebSocket 的时候 = 稍微少于 6 个				●
⑨服务器的预估响应时间	服务器响应请求并向客户端返回内容所需的时间的平均值。将缓存响应也考虑在内	1.5 秒左右 = 公司内部的应用； 0.1 秒 = 调优做得很好的网站		●		

使用表中的各个项目如何计算出各个目标值呢？方法如下所示。

算式中带圆圈的数字就是表中项目的编号。

【带宽（bps）的计算方法】

（每小时的处理页面数）×每个页面的大小×没有命中缓存的概率（（①×②×④）×（⑤×⑥）×（1－⑦））÷3600s（1h）×8（bit）=带宽（bps）

【吞吐（请求/秒）的计算方法】

（每小时的处理页面数）×没有命中缓存的概率（（①×②×④）×（1－⑦）÷3600s（1h））=页面请求数/秒

【同时访问数（用户人数）的计算方法】

（每小时的处理页面数）×平均思考时间（①×②×④）×③÷3600s（1h）=同时访问数

【同时访问数（活跃连接数）的计算方法】

（（①×②×③×④）÷3600s（1h））×（⑧或者⑤中比较小的那个值）÷（HTTP KeepAlive的预计时间或者③中比较大的那个值）

“同时访问数”实际上会比这里的值更小。这是因为浏览器能够使用一个连接来处理多个内容。如果要进行更精确的计算，就需要考虑每个内容的平均响应与连接的整合度。

应用程序单元性能测试

应用程序的单元性能测试是在集成测试执行之前进行的测试。在集成之后发生性能问题的时候，如果不能简单地修复，就会导致集成之后的计划延期。为了防止这种事情的发生，应该提前进行应用程序单元性能测试，以防范于未然。

虽然这个测试并不是必需的，但是为了推进项目顺利进展，防止像前面那样在项目后期才发现问题导致返工，对项目计划和成本产生影响，就需要应用程序单元的开发方在确认性能之后再移交。

实施时间　在应用程序开发中进行单元测试时，建议同时进行单元性能测试。可以像单元测试一样以测试优先（Test First）的形式组合，在每天进行build时自动测试并检测出错误的机制中加入性能测试。Java的话也可以通过JUnitPerf等来实施。可以在代码中直接要求，若超过了响应时间的目标就报错。

- 耐久测试

耐久测试可以归类到故障测试这一大类中。但是，耐久测试可以沿用性能测试的方法，比起单独执行，与性能测试一起执行效率更高。因此，耐久测试作为性能测试的关联领域，多由性能测试的负责人来实施。

此项测试的目的在于确认系统长时间运作时是否会出现故障或报错、内存泄露、计划外的日志堆积，以及日志轮转（Logrotate）和每日的批处理是否可以正常运作等。建议在追求高可用性的系统中实施此项测试，

不过需要确定耐久测试所需的时间。若需要进行 1 周的连续作业测试，当然就要占用 1 周的系统，中间若出现失误需要重新实施，或在耐久性测试的结果中发现问题，就需要进行修正并重新测试。因此，在确定耐久测试所需的时间时，至少需要预留耐久测试实施时间的 3 倍的时间。

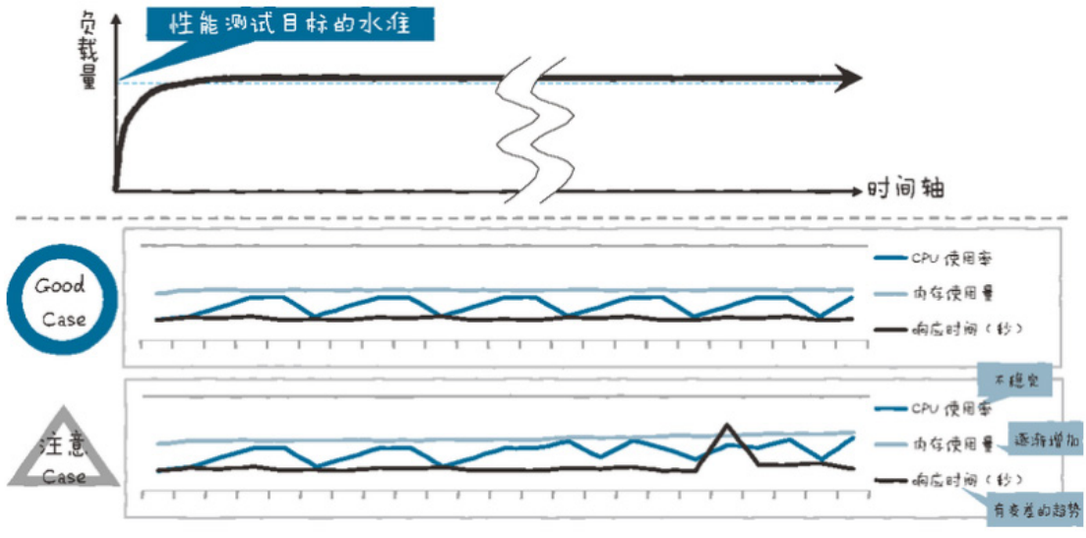


图　耐久测试

实施时间　如果在完成性能测试以及临界测试后，在项目上线前还有时间，并且有充足的时间可以占用系统的话，建议使用这个时间来进行耐久测试。

测量项目　耐久测试主要关注的是以下项目。

【响应时间】——需要观察平均响应时间是否有变差的趋势。该指标比 CPU 等的资源使用率更容易抓住问题。若有逐渐变差的趋势，就代表某个部分有可能发生了劣化

《图解性能优化》从性能的概念讲起，由浅入深，全面介绍了性能分析的基础知识、实际系统的性能分析、性能调优、性能测试、虚拟化环境下的性能分析、云计算环境下的性能分析等内容。

【内存使用量】——需观察进程中的内存使用量是否在逐步增加，据此可以检测出是否存在内存泄露。但是近年来OS中有缓冲和缓存的机制、堆内存（Heap Memory）和GC的机制，有时会先保证大量的内存，并在其中进行处理，所以需要在理解架构之后进行确认

【磁盘增加量】——在设计时应该有一个指标，即当访问数量是多少时日志的增加量是多少。然后再回过头来确认实际情况是否和设计时的预想一致，以及其他无关的目录下磁盘使用量是否有增加等

【其他参考指标】——CPU、线程数、系统内部内存（DB的缓存、Java VM的堆内部的动态）等

- 关联领域

如上所述，性能测试的关联领域有故障测试、耐久测试等。虽然这些测试单独实施起来难度较高，但由于可以利用性能测试的手法，而且大多数情况下都可以由性能测试的负责人来帮忙或者负责，因此在这里对其进行了介绍。要想成功地完成项目，项目组成员就不能持有“只要我负责的那部分没问题就OK了”这样的态度，而是应该所有成员一起努力协作，而项目经理则需要为大家创造更加容易一起协作的环境。

我们围绕着系统发布前的性能测试进行了解说，但在实际的开发现场，很多时候都是在项目发布后的打补丁以及库文件更新时进行的。为了使系统稳定运行，运维的时候也要能随时在测试环境中增加负载进行测试。

专题：打破“性能”瓶颈

Web性能核心优化点

作者/Ilya Grigorik

谷歌“Web加速”(Make The Web Fast)团队的性能工程师、开发大使。他每天的主要工作就是琢磨怎么让Web应用速度更快，总结并推广能够提升应用性能的最佳实践。

在任何复杂的系统中，性能优化的很大一部分工作就是把不同层之间的交互过程分解开来，弄清楚每一层次交互的约束和限制。优化不同层之间的交互与解一组方程没有什么不同，因为不同层之间总是相互依赖，但优化方式却有很多可能性。任何优化建议和最佳做法都不是一成不变的，涉及的每个要素都是动态发展的：浏览器越来越快、用户上网条件不断改善、Web应用的功能和复杂度也与日俱增。

因此，在讨论具体的最佳实践之前，一定要先明确真正的问题在哪里：什么是现代Web应用，我们手里有什么工具，如何测量Web性能，系统的哪些部分对优化有利或有碍？

超文本、网页和Web应用

互联网在过去几十年的发展过程中，至少带给了我们三种体验：超文本文档、富媒体网页和交互式Web应用。不可否认，后两种之间的界限有时候对用户来说很模糊，但从性能的角度看，这几种形式都要求我们在讨论问题、测量和定义性能时采取不同的手段。

- 超文本文档

万维网就起源于超文本文档，一种只有基本格式，但支持超链接的纯文本文档。按照现代眼光来看，这种文档或许没什么值得大惊小怪的，但它验证了万维网的假设、前景及巨大的实用价值。

- 富媒体网页

HTML工作组和早期的浏览器开发商扩展了超文本，使其支持更多的媒体，如图片和音频，同时也为丰富布局增加了很多手段。网页时代到来了，我们可以基于不同的媒体构建可见的页面布局了。但网页还只是看起来漂亮，很大程度上没有交互功能，与可打印的页面没有区别。

- Web应用

JavaScript及后来DHTML和Ajax的加入，再一次革命了Web，把简单的网页转换成了交互式Web应用。Web应用可以在浏览器中直接响应用户操作。于是，Outlook Web Access（IE5中的XMLHTTP就诞生于这个应用）等最早的、成熟的浏览器应用出现，也揭开了脚本、样式表和标记文档之间复杂依赖的新时代。

HTTP 0.9会话由一个文档请求构成，这对于取得超文本内容完全够用了：一个文档、一个TCP连接，然后关闭连接。因此，提升性能就是围绕短期TCP连接优化一次HTTP请求。

富媒体网页的出现改变了这个局面，因为一个简单的文档，变成了文档加依赖资源。因此，HTTP 1.0引入了HTTP元数据的表示法（首部），

HTTP 1.1 又加入了各种旨在提升性能的机制，如缓存、持久连接，等等。事实上，多TCP连接目前仍然存在，性能的关键指标已经从文档加载时间，变成了页面加载时间，常简写为PLT（Page Load Time）。

最后，Web应用把网页的简单依赖关系（在标签中使用媒体作为基本内容的补充）转换成了复杂的依赖关系：标记定义结构、样式表定义布局，而脚本构建最终的交互式应用，响应用户输入，并在交互期间创建样式表和标记。

结果，页面加载时间，这个一直以来衡量Web性能的事实标准，作为一个性能基础也越来越显得不够了。我们不再是构建网页，而是在构建一个动态、交互的Web应用。除了测量每个资源及整个页面的加载时间（PLT），还要回答有关应用的如下几个问题：

- 应用加载过程中的里程碑是什么？
- 用户第一次交互的时机何在？
- 什么交互应该吸引用户参与？
- 每个用户的参与及转化率如何？

性能好坏及优化策略成功与否，与你定义应用的特定基准和条件，并反复测试的效果直接相关。没有什么比得上应用特定的知识和测量，在关系到赢利目标和商业指标的情况下更是如此。

剖析现代Web应用

现代Web应用到底长啥样？HTTP Archive（http://httparchive.org/）可以回答这个问题。这个网站项目一直在抓取世界上是热门的网站（Alexa前100万名中的30多万名），记录、聚合并分析每个网站使用的资源、内容类型、首部及其他元数据的数量。

2013年初，一个普通的Web应用由下列内容构成。

- 90个请求，发送到15个主机，总下载量1311 KB HTML：10个请求，52 KB
  - 图片：55个请求，812 KB
  - JavaScript：15个请求，216 KB
  - CSS：5个请求，36 KB
  - 其他资源：5个请求，195 KB

在你读到这里的时候，前面所有数字都会变大，持续向上的趋势一直都很稳定，而且没有停止的迹象。不过，抛开实际的请求和字节数不提，更值得关注的还是个别组件的量级：现在，一个普通的Web应用大约就有1 MB，有100个左右的资源分散在15台不同的主机上！

与桌面应用相比，Web应用不需要单独安装，只要输入URL，按下回车键，就可以正常运行。可是，桌面应用只需要安装一次，而Web应用每次访问都需要走一遍“安装过程”——下载资源、构建DOM和CSSOM、运行JavaScript。正因为如此，Web性能研究迅速发展，成为人们热议

的话题也就不足为怪了。上百个资源、成兆字节的数据、数十个不同的主机，所有这些都必须在短短几百毫秒内亲密接触一次，才能带来即刻呈现的Web体验。

速度、性能与用户期望

速度和性能是两个相对的概念。每个应用都要满足自己特定的需求，因为商业条件、应用场景、用户期望，以及功能复杂性各不相同。尽管如此，如果应用必须对用户作出响应，那我们就必须从用户角度来考虑可感知的处理时间这个常量。事实上，虽然生活节奏越来越快——至少我们感觉如此，但人类的感知和反应时间则一直都没有变过（如下表所示），而且与应用的类型（在线或离线）、媒体的类型（笔记本、台式机或移动设备）无关。

表　时间和用户感觉

时间	感觉
0 ~100 ms	很快
100~300 ms	有一点点慢
300~1000 ms	机器在工作呢
> 1000 ms	先干点别的吧
> 10000 ms	不能用了

如果想让人感觉很快，就必须在几百毫秒内响应用户操作。超过1 s，用户的预期流程就会中断，心思就会向其他任务转移，而超过10秒钟，除非你有反馈，否则用户基本上就会终止任务！

现在，把DNS查询，随后的TCP握手，以及请求网页所需的几次往返时间都算上，光网络上的延迟就能轻易突破100~1000 ms的预算。难怪有那么多用户，特别是那些移动或无线用户，抱怨上网速度慢了！

资源瀑布

谈到Web性能，必然要谈资源瀑布。事实上，资源瀑布很可能是我们可以用来分析网络性能，诊断网络问题的一个最有价值的工具。很多浏览器都内置了一些手段，让我们能查看资源瀑布。此外，还有一些不错的在线工具，比如WebPageTest（http://www.webpagetest.org/），可以在不同的浏览器中呈现资源瀑布。

首先，必须知道每一个HTTP请求都由很多独立的阶段构成：DNS解析、TCP连接握手、TLS协商（必要时）、发送HTTP请求，然后下载内容。这些阶段的时长在不同的浏览器中会略有不同，但为了简单起见，这里就使用WebPageTest测试的结果。请大家提前熟悉每种颜色在自己浏览器中的含义。

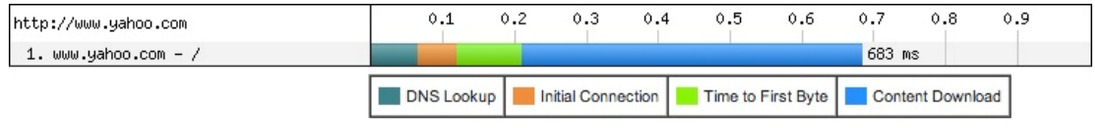


图　HTTP请求的构成（WebPageTest）

仔细看一下上图，打开Yahoo!主页花了683 ms，而其中有200多毫秒在等待网络就绪，占到了请求延迟的30%！然而，请求文档还只是开始，现代的Web应用还需要各种资源配合来生成最终结果。准确地说，要加载Yahoo!主页，浏览器要请求52个资源，从30个不同的主机获得，这些资源加起来总共486 KB。

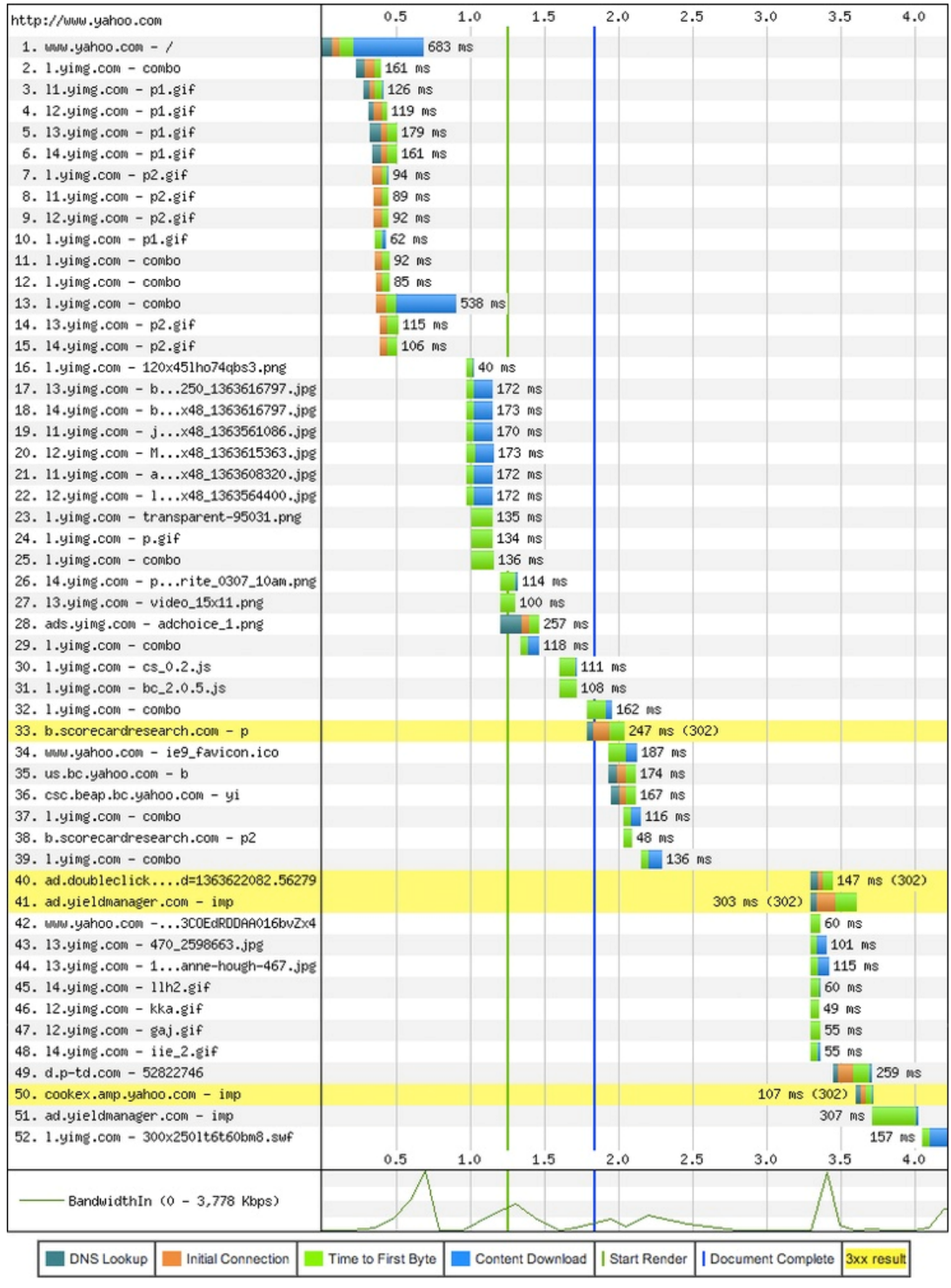


图　Yahoo.com资源瀑布图（WebPageTest，2013年3月）

资源的瀑布图能够揭示出页面的结构和浏览器处理顺序。首先，取得www.yahoo.com对应的文档，同时分派新的HTTP请求：HTTP解析是递增执行的，这样浏览器可以及早发现必要的资源，然后并行发送请求。实际上，何时获得什么资源很大程度上取决于标记结构。浏览器可以变更某些请求的优先顺序，但递增地发现文档中的每一个资源，最终造就了不同资源间的“瀑布效果”。

其次，“Start Render”（绿色的竖线）会在所有资源下载完成前开始，以便用户在页面构建期间就能与之交互。其实，“Document Complete”事件（蓝色的竖线）也会在剩余资源下载完成前触发。换句话说，浏览器的加载旋转图标此时停止旋转，用户可以继续与页面交互，但Yahoo!主页会渐进地在后台填充后续内容，比如广告和社交部件。

最早渲染时间、文档完成时间和最后资源获取时间，这三个时间说明我们讨论Web性能时有三个不同测量指标。我们应该关注哪一个时间呢？答案并不唯一，因应用而不同！ Yahoo!的工程师们选择了利用浏览器递增加载机制，让用户能够尽早与重要内容交互。而这样一来，他们必须根据应用不同，确定哪些内容重要（须先加载），哪些内容不重要（可以后填充）。

网络的瀑布图是个很强大的工具，有助于揭示任何页面或应用是否处于优化状态。前面分析和优化资源瀑布的过程，一般称为前端性能分析和优化。不过，这个称呼有误导性，好像所有性能瓶颈都在客户端似的。实际上，尽管JavaScript、CSS和渲染流水线很重要，而且资源对性能影响很大，但服务器响应时间和网络延迟（“后端性能”）对资源瀑布的影响也不容忽视。毕竟，如果网络被阻塞，也就谈不上什么解析或运行资源了！

为了说明这一点，只要切换到WebPageTest资源瀑布的连接视图即可。

图　Yahoo.com资源瀑布图的连接视图（WebPageTest，2013年3月）

资源瀑布图记录的是HTTP请求，而连接视图展示了每个TCP连接（这里共30个）的生命期，这些连接用于获取Yahoo!主页的资源。哪里比较突出呢？注意蓝色的下载时间，很短，在每个连接的总延迟里几乎微不足道。这里总共发生了15次DNS查询，30次TCP握手，还有很多等待接收每个响应第一个字节的网络延迟（绿色）。

为什么有些请求只显示绿色条（第一字节接收时间）？因为很多响应很小，相应的下载时间没有在图中体现。事实上，请求、响应的主要时间通常是往返延迟和服务器处理时间。

最后，也是最重要的，连接视图的底部显示了带宽利用率曲线。除了少量数据爆发外，可用连接的带宽利用率很低。这说明性能的限制并不在带宽！难道这是个异常现象，或者浏览器问题？都不是。对大多数应用而言，带宽的的确确不是性能的限制因素。限制 Web 性能的主要因素是客户端与服务器之间的网络往返延迟。

性能来源：计算、渲染和网络访问

Web 应用的执行主要涉及三个任务：取得资源、页面布局和渲染、JavaScript 执行。其中，渲染和脚本执行在一个线程上交错进行，不可能并发修改生成的 DOM。实际上，优化运行时的渲染和脚本执行是至关重要的。

可是，就算优化了 JavaScript 执行和渲染管道，如果浏览器因网络阻塞而等待资源到来，那结果也好不到哪里去。对运行在浏览器中的应用来说，迅速而有效地获取网络资源是第一要义。

有人可能会问，互联网今天的速度不是快多了吗，难道网络还会阻塞？是的，我们的应用也比以前大多了。然而，假如真像每个 ISP 和移动运营商鼓吹的那样，全球平均网速已经达到 3.1 Mbit/s，而且还在继续提速，Web 应用大一点又算得了什么呢？可惜的是，凭直觉以及前面展示的

Yahoo! 的例子，我们知道如果真是这样，那你就不用接着看这篇文章了。好，下面我们仔细谈一谈。

** 更多带宽其实不（太）重要 **

先别急，带宽当然重要！毕竟，所有 ISP 和移动运营商的广告，都意在提醒我们高带宽的好处：上传和下载加速、更流畅地欣赏视频，都有赖于 **[请读者在此插入最新数字]**Mbit/s 的速度！

能接入更高带宽固然好，特别是传输大块数据时更是如此，比如在线听音乐、看视频，或者下载大文件。可是，日常上网浏览需要的是从数十台主机获取较小的资源，这时候往返时间就成了瓶颈：

- 在 Yahoo! 主页上看视频受限于带宽；
- 加载和渲染 Yahoo! 主页受限于延迟。

根据在线视频品质及编码不同，需要的带宽从几百 Kbit/s 到几 Mbit/s 不等。比如，要流畅观看 1080P 的高清视频，需要 3 Mbit/s 以上的带宽。这个带宽是很多用户触手可及的，Netflix 等在线视频网站的流行也印证了这一点。那么，为什么下载比视频小得多得多 Web 应用，在能流畅观看高清视频的连接上却成了难题呢？

** 延迟是性能瓶颈 **

关于为什么延迟会成为浏览网页的限制因素，我们就来通过两张图（如下所示），定量地感受一下不同的带宽和延迟时间对页面加载时间分别

有什么影响。这两张图来自SPDY协议的开发者之一Mike Belshe，测试的是互联网上最热门的一些站点。

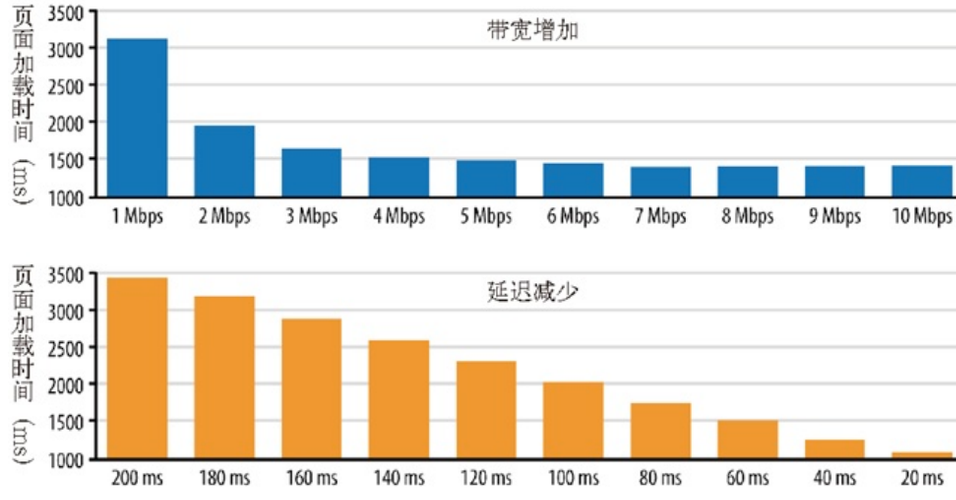


图　页面加载时间与带宽和延迟的关系

在第一个测试中，连接的延迟时间固定而带宽递增，从1 Mbit/s依次递增至10 Mbit/s。注意一开始，从1 Mbit/s升级到2 Mbit/s，页面加载时间几乎减少了一半，这也是我们希望看到的结果。可是，在此之后，带宽递增，加载时间减少得越来越不明显。而当带宽超过5 Mbit/s时，加载时间的减少比例只有几个百分点，从5 Mbit/s升级到10 Mbit/s，页面加载时间仅降低了5%！

Akamai公司的宽带速度报告显示，美国普通消费者上网带宽已经达到5 Mbit/s以上。很多其他国家很快也能达到这个数字，甚至有的已经超过了它。由此可知，靠提高带宽不会给美国人浏览网页带来多大的性能提升。或许美国人下载大文件、看视频的速度很快，但加载包含这些文件的页面的时间不会明显缩短。换句话说，增加带宽没有那么重要。

《Web性能权威指南》深入浅出地讲解并演示了针对TCP、UDP和TLS协议的性能优化最佳实践，以及面向无线和移动网络进行优化时的特殊要求。它还全面剖析了浏览器技术的几项重大革新，包括使用这些新技术时在性能方面需要的独到考量。革命性的HTTP 2.0、XHR客户端网络脚本、基于SSE及WebSocket的实时数据流，以及通过WebRTC实现P2P通信，对这些面向未来的重大浏览器技术，本书都从性能优化的角度给出了详尽的解读和分析。

然而，在延迟以20 ms递减的试验中，页面加载时间呈线性减少趋势。在选择ISP时，是不是应该把延迟时间而不是带宽放在首位呢？

> 要让互联网从整体上提速，就必须寻求降低RTT的方法。把跨大西洋的RTT从150 ms降低到100 ms，结果会怎么样？结果比把用户带宽从3.9 Mbit/s提高到10 Mbit/s甚至1 Gbit/s对速度的影响都大。减少页面加载时间的另一种方法，就是减少加载每个页面过程中的往返次数。今天，加载每个页面都需要客户端到服务器之间的数次往返。这些往返很大程度上是由客户端与服务器之间为建立连接（DNS、TCP、HTTP等）而进行握手所导致的，当然有的是由通信协议引发的（如TCP慢启动）。假如我们能够对协议加以改进，使得加载同样多的数据只需更少的往返，那应该也能够减少页面加载时间。这就是SPDY的目标之一。
>
> ——Mike Belshe，更多带宽其实不（太）重要

很多人可能都会惊讶于这两项试验的结果，而实际情况的确如此，这正是TCP握手机制、流量和拥塞控制、由丢包导致的队首拥塞等底层协议特点影响性能的直接后果。大多数HTTP数据流都是小型突发性数据流，而TCP则是为持久连接和大块数据传输而进行过优化的。网络往返时间在大多数情况下都是TCP吞吐量和性能的限制因素。于是，延迟自然也就成了HTTP及大多数基于HTTP交付的应用的性能瓶颈。

> 如果延迟对大多数有线连接是限制性能的因素，那可想而知，它对无线客户端将是更重要的性能瓶颈。事实上，无线延迟明显更高，因此网络优化是提升移动Web应用性能的关键。

专题：打破“性能”瓶颈

Java性能分析工具箱

性能分析器是性能分析师工具箱中最重要的工具。Java有许多性能分析器，各有优缺点。

性能分析经常需要使用各种工具——特别是那些采样分析器。即便是相同的应用，不同的分析器也能发现其他分析器所发现不了的问题。

几乎所有的Java性能分析工具都是用Java写的，并以“关联”（attaching）应用的方式进行性能分析——意思是性能分析器开启与目标应用之间的socket（或其他通信通道）。随后目标应用和性能分析工具交换应用的行为信息。

这意味着，就像调优任何其他Java应用一样，你必须注意性能分析工具自身性能的调优。尤其是如果应用很大，需要传递给分析工具的数据非常多，那么分析工具就必须得有足够大的堆内存来处理这些数据。性能分析工具采用并发GC算法运行，通常是个不错的主意。在性能分析过程中，不合时宜的Full GC停顿会导致性能分析工具缓冲区中的数据溢出。

作者/Scott Oaks

Oracle资深架构师，专注研究Oracle中间件软件的性能。加入Oracle之前，他曾于Sun Microsystem公司任职多年，在多个技术领域都有建树，包括SunOS的内核、网络程序设计、Windows系统的远程方法调用(RPC)以及OPEN LOOK虚拟窗口管理器。1996年，Scott成为Sun公司的Java布道师，并于2001年加入Sun公司的Java性能小组——从那时起他就一直专注于Java的性能提升。此外，Scott也在O'Reilly出版社出版了多部书籍，包括*Java Security*、*Java Threads*、*JXTA in a Nutshell*和*Jini in a Nutshell*。

采样分析器

性能分析有两种模式：数据采样或数据探查。数据采样是性能分析的基本模式，带来的开销最小，这点很重要。性能分析为人诟病的一点就是，对应用进行的测量会改变它的性能。(然而，你必须进行性能分析：还有什么方法能让你知道程序中的猫是否存活？）限制性使用性能分析可以使得测试结果更接近现实模型，即通常情况下应用的表现行为。

不幸的是，采样分析器可能会遇到各种错误。计时器定期触发采样分析器，然后采样分析器检查每个线程并判断正在执行的方法。

下图是采样中最常见的错误。线程交替执行`methodA`（见图中的阴影块）和`methodB`（见图中的白色块）。如果计时器只在线程执行`methodB`时触发，采样分析器就会报告成，在所有这段时间内线程都在执行`methodB`，而实际上更多时间在执行`methodA`。

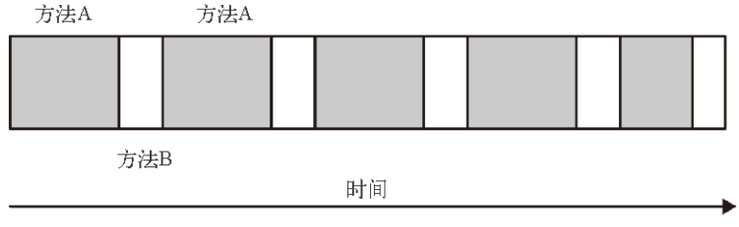


图　交替方法执行

常见的采样错误还不止这一个。减少这类错误的方法是，拉长分析的时间段，同时减少采样间隔。但是减少采样间隔和尽量减小性能分析影响应用的目的相违背，这里需要有平衡。不同的性能分析工具考虑的平衡点也不同，这就是为什么不同的性能分析工具所报告的数据有很大差别的原因。

下面的这个图是GlassFish应用服务器启动一个域时所测量出的基本采样分析。图中显示，大块时间（19%）都用在了defineClass1()上，接着是getPackageSourcesInternal()等方法。程序的启动性能是受JVM定义类过程的控制的，这并不奇怪。为了加快代码运行速度，必须改进类加载的性能。

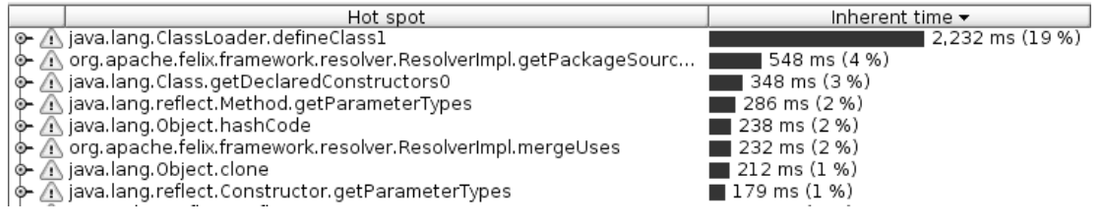


图　基于一个样例的测试

请仔细留意上面的最后一句话：必须改善类加载的性能，而不是改善defineClass1()的性能。改善性能的通常设想是，应该先优化性能分析结果中排在最上面的方法。然而，这种做法常常受限。这个案例中的defineClass1()是JDK的一部分，而且是本地方法（native method），除非改写JVM，否则不可能改善它的性能。而且，即便能改写JVM使得这个方法少花60%的时间，但换算成整体性能的改善，也不过10%——这无异于杯水车薪。

而更常见的情况是，排在性能分析结果顶上的方法只占了整体时间的2%或3%。即便将它所用的时间砍掉一半（通常极为困难），也只会使应用的性能提高1%。只盯着性能分析结果中最上头的方法，通常并不会让性能提高很多。

相反，你应该在最顶上那些方法所指引的区域中搜寻可优化的地方。GlassFish性能工程师不会试图让类定义更快，但通常他们会找出如何加速类加载——装载更少的类、并行加载类等。

快速小结

1. 采样分析器是最常用的分析器。
2. 因为采样分析器的采样频率相对较低，所以引入的测量失真也较小。
3. 不同的采样分析器各有千秋，针对不同应用各有所长。

探查分析器

探查分析器相比于采样分析器，侵入性更强，但它们可以给出关于程序内部所发生的更有价值的信息。下图探查的是与采样分析器部分介绍的相同的GlassFish域，采用的也是同一个性能分析工具，但这次使用的是探查模式。

Hot spot	Inherent time ▾	Average Time	Invocations
org.apache.felix.framework.resolver.ResolverImpl.getPackageSourcesInternal	11,934 ms (13 %)	356 µs	33,489
org.apache.felix.framework.util.ImmutableMap.get	11,651 ms (12 %)	2 µs	4,769,764
java.lang.Object.equals	5,797 ms (6 %)	0 µs	15,907,963
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.findClass	4,814 ms (5 %)	1,022 µs	4,710
java.util.Map.get	4,635 ms (5 %)	0 µs	6,783,508
java.util.Iterator.hasNext	4,123 ms (4 %)	0 µs	6,179,992
java.util.Map$Entry.getKey	4,013 ms (4 %)	0 µs	11,126,431
java.util.Iterator.next	3,951 ms (4 %)	0 µs	5,687,480
org.apache.felix.framework.util.ImmutableList$ListItr.hasNext	3,430 ms (3 %)	0 µs	4,591,013
org.apache.felix.framework.resolver.ResolverImpl.mergeUses	2,329 ms (2 %)	8 µs	265,690
java.lang.String.equals	1,988 ms (2 %)	0 µs	5,056,880

图　一个探查分析器

我们立即就能从这份性能分析结果中看出些问题。首先，现在最“热”的方法成了`getPackageSourcesInternal()`，因为占用时间达到了13%（而

不是前个例子中的4%)。性能分析结果中还有其他几个占用很多时间的方法，也都排到了前头，而defineClass1()已经完全消失了。这次工具还报告了每个方法被调用的次数，并且基于这个次数计算出了每次调用的平均时间。

难道这个性能分析结果比采样的好？这取决于能否有办法知道在给定的情况下哪种分析结果更精确。探查分析结果中的调用次数毫无疑问是精确的，而其他信息在判断哪段代码实际花费了更多时间，哪些有更多的优化空间时通常很有帮助。在这个例子中，虽然ImmutableMap.get()消耗了12%的整体时间，但它被调用了约470万次。减少这个方法的总调用次数比加快它的运行速度可以更显著地提升性能。

不过话说回来，探查分析器会在类加载时更改类的字节码（即插入统计调用次数的代码等）。相比采样分析器，探查分析器更可能会将性能偏差引入应用。比如，JVM会内联小方法使得执行时不会产生方法调用。编译器的这种判断是基于代码的大小，所以有可能使得这段代码不再被内联，这取决于如何探查代码。这会导致探查分析器高估某些方法对整体性能的影响。方法内联只是一个例子，说明编译器会基于代码在内存中的布局而做出决策。一般来讲，加入（更改）的探查代码越多，运行的性能分析结果就越有可能发生变更。

为何ImmutableMap.get()只在这里出现，而没在前面的采样分析结果中出现，还有一个很重要的技术原因。Java的采样分析器只能在线程位于安全点时采集线程样本——基本上只有在JVM分配内存的时候。get()方法可能永远都不会进入安全点，所以可能永远都不会被采样。

在这个例子中，探查分析器和采样分析器给出的结果都指向同样的代码区域：类装载和类解析。实际上，不同分析器的结果可能指向的是完全不同的代码区域。性能分析器可以用来很好地估计，但也仅仅是估计而已：某时某刻它们也会犯错。

快速小结

1. 探查分析器可以给出更多的应用信息，但相对采样分析器，它对应用的影响更大。
2. 探查分析器应该仅在小代码区域——一些类和包——中设置使用，以限制对应用性能的影响。

阻塞方法和线程时间线

接下来，我们用另一种探查分析工具（NetBeans性能分析器）来分析GlassFish启动。可以看出，此时的执行时间主要被`park()`和`parkNanos()`占用了（相对于占用更少的`read()`方法）。

Hot Spots - Method	Self time [%]	Self time	Invocations
java.util.concurrent.locks.LockSupport.**parkNanos** (Object, long)		632,104 ms (57.3%)	470
java.util.concurrent.locks.LockSupport.**park** (Object)		350,790 ms (31.8%)	356
java.net.SocketInputStream.**read** (byte[], int, int, int)		60,144 ms (5.5%)	21
org.apache.felix.framework.resolver.ResolverImpl.**getPackageSourcesInternal** (or...		1,562 ms (0.1%)	22,400
java.lang.ClassLoader.**defineClass** (String, byte[], int, int, java.security.ProtectionDo...		1,501 ms (0.1%)	2,447
java.lang.Class.**privateGetDeclaredConstructors** (boolean)		1,282 ms (0.1%)	10,657
java.util.HashMap.**getEntry** (Object)		1,015 ms (0.1%)	1,656,447
java.util.concurrent.LinkedTransferQueue.**awaitMatch** (java.util.concurrent.Linked...		1,007 ms (0.1%)	18
org.apache.felix.framework.resolver.ResolverImpl.**mergeUses** (org.osgi.framewor...		643 ms (0.1%)	182,836
org.apache.felix.framework.util.ImmutableMap.**get** (Object)		608 ms (0.1%)	3,067,340
org.apache.felix.framework.util.ImmutableList$ListItr.**hasNext** ()		605 ms (0.1%)	2,899,072
java.lang.ClassLoader.**getCallerClassLoader** ()		566 ms (0.1%)	18,506
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.**findClass** (String)		538 ms (0%)	2,980
org.apache.felix.framework.util.ImmutableList.**size** ()		524 ms (0%)	2,903,040
org.apache.felix.framework.BundleWiringImpl.**findClassOrResourceByDelegation** ...		455 ms (0%)	8,289
java.util.zip.Inflater.**inflate** (byte[], int, int)		413 ms (0%)	9,988
java.util.HashMap.**hash** (Object)		409 ms (0%)	2,259,411

图　一个带有阻塞方法的分析器

这些方法（以及类似的阻塞方法）并不消耗CPU时间，所以对应用的整体CPU使用率没有贡献。没有必要优化它们的执行。应用线程花费623秒不是在执行parkNanos()方法，而是等待别的事情发生（例如，等待其他线程调用notify()方法）。park()和read()方法同样如此。

因为这个原因，大多数分析器不会报告被阻塞的方法，相应的线程也被显示为空闲（这个例子，NetBeans被设置为显式包括这些方法和其他Java级别的方法）。在这个例子中，这是件好事：停止运行的是Java线程池中的线程，这些线程用来执行服务器所收到的servlet（和其他）请求。启动时没有请求发生，所以这些线程被阻塞，等待任务执行。这是正常状态。

在其他情况下，你总是希望能看到那些阻塞调用所花费的时间。线程在wait()——等待其他线程唤醒——中的用时决定了许多应用的整体执行时间。大多数基于Java的性能分析器可以通过设置过滤器和调整其他选项来显示或隐藏这些阻塞方法。

另一方面，审视线程的执行模式而不是分析器给阻塞方法所标记的用时，常常更有价值。下图是Oracle Solaris Studio性能分析工具中所显示的线程。

图　一个线程时间线分析器

每个横向区域都是一个线程（图中有两个线程：线程1.3和线程1.2）。颜色（或不同的灰色）条表示不同方法的执行过程，空白区域表示线程没有在执行。从更高角度来观察，线程1.2先是执行了大量代码，然后等待线程1.3，线程1.3之后稍稍等待线程1.2执行其他事情，等等。用工具进一步深入这些区域，可以让我们了解线程之间是如何相互影响的。

还可以注意到，有些空白区域没有任何线程在执行。这张图只显示了应用许多线程中的两个，所以这两个线程可能一起在等待其他线程，或者想成正在执行阻塞调用`read()`（或类似的）。

快速小结

1. 线程被阻塞可能是性能问题，也可能不是，有必要进一步调查它们为何被阻塞。
2. 通过正被阻塞的方法调用，或者分析线程的时间线，可以辨认出被阻塞的线程。

本地分析器

本地性能分析工具是指分析JVM自身性能的工具。这类工具可以看到JVM内部的工作原理，如果应用自身含有本地库，这类工具也能看到本地库代码的内部。所有本地分析工具都可以用来分析JVM的C代码（以及任何本地库），而有些本地工具则可以用来分析任何Java和C/C++应用。

下面的图是不是有点眼熟，它是Oracle Solaris Studio分析器分析GlassFish启动的结果。这是本地分析器，可以接受Java和C/C++代码。（虽然名字中有Solaris，但它也可以在Linux系统上运行。实际上，这

些图片都是Linux OS上截取的分析结果。不过，如果在Solaris的内核结构上运行，这个工具可以展示更多的应用信息。）

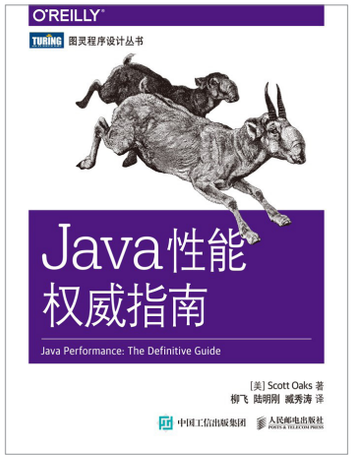
User CPU ▽ (sec.)	User CPU (sec.)	Name
25.105	25.105	<Total>
20.064	20.064	<JVM-System>
0.550	0.670	java.lang.ClassLoader.defineClass1(java.lang.String, byte[], int, int, java.sec
0.380	0.380	<no Java callstack recorded>
0.220	0.220	java.util.jar.Attributes.read(java.util.jar.Manifest$FastInputStream, byte[])
0.120	0.170	java.util.zip.Inflater.inflateBytes(long, byte[], int, int)
0.100	0.100	org.apache.felix.framework.util.ImmutableMap.get(java.lang.Object)

图　一个本地分析器

请注意，第一处与前面不同的是，记录下的应用总CPU时间为25.1秒，其中20秒是JVM-System所用。这是JVM自身代码的特性：JVM编译器线程和GC线程（加上其他一些辅助线程）。我们可以进一步了解到，实际上，这个例子的所有时间都花费在了编译器上（启动过程确实如此，期间有许多代码需要编译）。这个例子中，GC线程只花了很少量的时间。

除非是为了深入研究JVM自身，否则这些本地信息就足够了。如果我们愿意，可以进一步检查实际的JVM功能并优化它们。不过，这个工具所透露出来的关键信息是——基于Java的性能分析工具所不能提供的——应用花在GC上的时间。在基于Java的性能分析工具中，GC线程的影响几乎看不到。（除非测试所运行的机器是CPU受限，否则编译器线程占用大量时间并无大碍：虽然编译线程会消耗大量的CPU时间，但只要机器上有更多可用的CPU，应用自身就不受影响，因为编译是在后台发生。）

一旦检测出本地代码的影响，就可以过滤出来，以便重点考虑实际的启动过程。



《Java性能权威指南》对Java 7和Java 8中影响性能的因素展开了全面深入的介绍，讲解传统上影响应用性能的JVM特征，包括即时编译器、垃圾收集、语言特征等。内容包括：用G1垃圾收集器最大化应用的吞吐量；使用Java飞行记录器查看性能细节，而不必借助专业的分析工具；堆内存与原生内存最佳实践；线程与同步的性能，以及数据库性能最佳实践等。

User CPU ▽ (sec.)	User CPU (sec.)	Name
5.041	5.041	*<Total>*
0.550	0.670	java.lang.ClassLoader.defineClass1(java.lang.String, byte[], int, int, java.sec
0.380	0.380	*<no Java callstack recorded>*
0.220	0.220	java.util.jar.Attributes.read(java.util.jar.Manifest$FastInputStream, byte[])
0.120	0.170	java.util.zip.Inflater.inflateBytes(long, byte[], int, int)
0.100	0.100	org.apache.felix.framework.util.ImmutableMap.get(java.lang.Object)

图　一个过滤的本地分析器

采样分析器再次把`defineClass1()`指为了最热的方法，虽然这个方法及其子调用的实际用时——5.041秒中的0.67秒——占了约11%（不像前面采样分析器报告的那么显著）。分析结果还指出了其他一些需要调查的信息：读取和解压JAR文件。由于和类装载有关，这说明我们追踪的方向是正确——但在这个例子中，重要的是看到了实际读取（通过`inflateBytes()`方法）JAR文件I/O只占了极少的百分点。其他工具则不会告诉我们这些信息——部分是因为Java ZIP库中的本地代码被当作阻塞调用而被过滤掉了。

无论你用上述哪种性能分析工具，或更好的工具，至关重要的一点就是熟悉它们的特性。性能分析器是查找性能瓶颈最重要的工具，但你必须学会如何使用它们，然后找到需要优化的代码区域，而不是仅仅关注最上头的方法。

快速小结

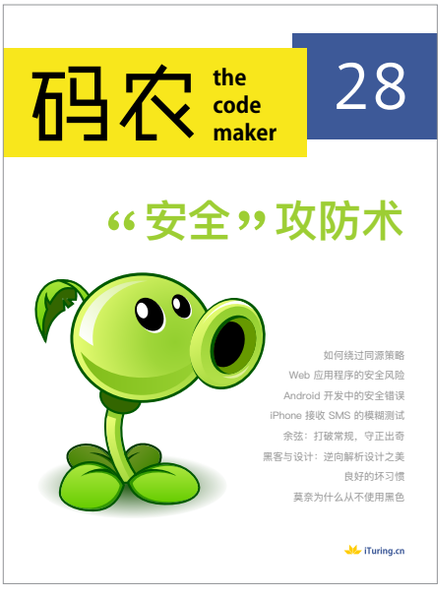
1. 本地性能分析器可以提供JVM和应用代码内部的信息。
2. 如果本地性能分析器显示GC占用了主要的CPU使用时间，优化垃圾收集器就是正确的做法。然而，如果显示编译线程占用了明显的时间，则说明通常对应用性能没什么影响。

读《码农》吐吐槽 还能赚银子！

《码农》电子刊如今慢慢悠悠已经出版了31期，从一开始的好评如潮，到现在的“怎么这么抽象啊”“赶脚内容没有以前好了？”“一下就翻完了，没什么内容啊”……小编表示压力很大，现在的码农读者太难伺候了，明明是本免费杂志，￥%#%&*%@# ￥*@ ￥#@！

但是，一看到好评，编者还是会热泪盈眶，热血沸腾，各种正能量啊“这么好质量的杂志还免费，天上掉馅饼呀”“希望一直出！每一版都读了，很值得推荐！！”……

所以《码农》还是会一直做下去，但是，是好是坏，您得给个话儿啊！



活动规则：在吐槽贴中留言，选出任何一期中你最喜欢的文章和最不喜欢的文章，即可获得图灵社区银子2两！凡吐槽吐得掷地有声者，加赠银子3两（共5两）！（怎样使用银子兑换图书）

活动时间：本活动长期有效。

Head First策划人Kathy Sierra：

好产品让用户为自己尖叫

Kathy Sierra，O'Reilly出版社Head First系列图书策划人之一，大型Java开发者社区JavaRanch.com创办人，多款教育类和娱乐类游戏主要开发人员。她深谙产品交互之道和认知科学理论，为加利福尼亚大

学洛杉矶分校创立了新媒体与交互设计课程。多年来，她一直帮助大公司、创业公司、非营利组织和教育者重新思考打造用户体验的方法，培养持续忠诚的用户。

- O'Reilly出版社Head First系列图书策划人之一

Head First（深入浅出）是O'Reilly Media出版社推出的一套入门指导型图书系列，由Kathy Sierra和Bert Bates策划创办。最初，该系列只涵盖编程和软件工程。由于该系列图书的成功，目前已经扩展至其他的主题，像科学、数学和商业等领域。Head First系列强调图书的“非正统性”，强化读者的视觉感受和读者的参与度。图书增添了很多互动性强的谜题、笑话，摒弃“标准”的设计和布局，推崇会话式的写作风格，让读者可以参与到给定的话题当中。（翻译自Wikipedia）

目前，图灵翻译出版了《嗨翻C语言》《Head First JavaScript程序设计》等。

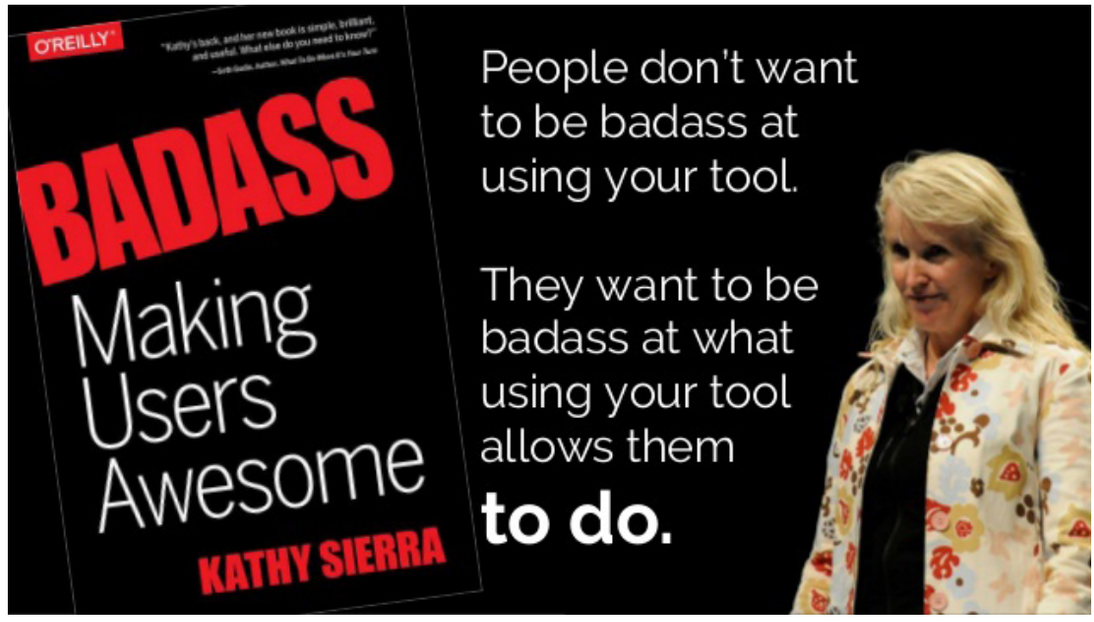
- 游戏开发人员

Kathy是电脑游戏Terratopia的主要开发人员，这是一款由Virgin Sound & Vision公司在1998年发行的儿童冒险游戏。另一款叫作“所有的狗狗都去了天堂“的游戏，是一款基于电影的游戏，素材由米高梅公司免费提供。

- 深谙产品交互之道和认知科学理论

“四岁，我的癫痫第一次发作的时候，我开始对大脑产生兴趣，”Kathy在个人博客上写道。这样的经历激发了Kathy对认知科学的探索。

多年来，她一直帮助大公司、创业公司、非营利组织和教育者重新思考打造用户体验的方法，培养持续忠诚的用户。Kathy 认为，“**人们并不想因为擅长使用某个产品而变得了不起，他们想要那种因为使用了某个产品而带来的成就感。**”



她的最近著作《用户思维 +：好产品让用户为自己尖叫》针对如何打造畅销产品的相关问题提出了新颖的观点：**用户并不关心产品本身有多棒，而是关心使用产品时自己有多棒。**作者利用其多年的交互设计经验，生动阐释了这一观点背后的科学。可贵的是，本书并不止步于解释“为什么”，还清晰呈现了“怎么做”。

访谈实录：

从你的博客上，我了解到你真的很喜欢马。因为只有很少数的人才有机会养马，所以想问下，相比人类，你能从马儿身上得到什么？

实际上，我们把badass的原则同样运用到了马匹身上，通过帮助“他们”变成更好的“马儿”，我们得到了非常棒的反馈和回应。传统意义上，大多数的马是为人类服务的，或是为了运动或是休闲消遣。训练的重点，通常是让马匹更擅长于服务“人类”的运动。我们的训练方式却很不一样……让马儿在“马匹”自己想要发展的方向上得到训练——敏捷、自豪、健壮、快乐、优秀。结果是，马儿在接触人类时的许多问题：恐惧、进攻、“懒散”、不听话，等等——这些都消失了。只有专注于马儿本身，我们才能得到一匹漂亮、健壮、快乐、有趣的马儿。

和马儿一起工作的经历就像是做实验（笑），我们可以测试许多关于学习和动机的科学理论，因为大部分的运动和动机科学研究同样适用于大多数的哺乳动物和人类。马儿能给我们带来非常快速和相当有价值的反馈。本书中的许多概念也都适用于马匹的训练。

根据你在游戏开发方面的经验，你是如何看待反沉迷系统的？事实上，腾讯开发的一款名为“王者荣耀”的网络游戏，就曾因为反沉迷系统的不到位，而饱受争议。

我不太了解中国游戏的反沉迷系统，但对游戏设计中的成瘾机制还是比较了解的。我反对在游戏设计过程中故意运用任何形式的“成瘾”技术。

“游戏”至少还是透明的，它无非是让你花时间玩游戏，我反而更担心那些运用行为科学原理，让你对他们的产品“上瘾”的技术。

我们的目标——已经在书中讨论过——并不是让人们更擅长“使用产品”，而是帮助他们在更大的情境下拥有更多的可能。因为使用了产品，他们的生活产生改变。一个最简单的例子就是，我们不希望人们“沉迷于”摄影app或是相机，而是希望他们在更大的摄影、艺术或是新闻传播等领域变得更有技巧。

作为产品和用户体验的设计师，我认为，我们有一种“道德上”的义务去思考我们的产品和产品体验是如何影响一个人的“一生”的，而不仅仅是他们使用产品的那段时间。我们对他们的生活有改善吗？还是正在伤害他们，强行剥夺他们的认知和时间，而这些本该是可以用于他们更感兴趣的地方的？

我相信，从很多方面来说，把产品或是商业的成功建立在从客户/用户那里“窃取”的注意力上，都是一个错误。最好的产品会鼓励人们去学习更多，做更多，积极地运用他们从产品中获得的东西，而不是沉迷于产品本身。作为一名专业的驯兽师，我也看到一些公司把人类客户当作动物一样，当他们完成某项表演后会给予一定的奖励。他们聘请行为科学方面的博士、使用大数据，为达目的，无所不用其极。我甚至都不会用这种方式对待我的“动物们”。

我们可以做得更好。我们可以设计出丰富用户生活的体验和产品，这不仅仅是让用户在产品上花费更多时间的问题。这才是我们设计出最成功的产品的方式。

什么原因让你和Bert Bates决定策划Head First系列图书的?

我教Java编程已经有好几年了，所以我知道人们会在哪里痛苦挣扎。Java语言第一次出现的时候，我需要挑一本书学习Java。但是，我发现大多数编程书籍的设计方式会让编程学习变得非常低效。我也有些学习理论的背景知识，意识到我们可以做些改变让学习更有效、更“容易”。我们在编写Head First Java的时候，市面上已经有大约2000本Java相关的图书！当然，这对我们来说是个很大的挑战——怎样以一种有意义的方式把我们的书凸现出来？我们认为，目标必然不能是“写一本好书”，而应该是选择一个不同的目标——我们需要写一本能让读者“真正地”学习的书，这样他们才会告诉别人，“我从这本书中学到了知识！”

我们知道，人们只有真正地坚持阅读这本书的时候，这种情况才会发生。统计的数据也显示说，大多数的人并不能坚持看完任何一本编程书。所以，这就成了我们的目标——怎样才能帮助人们坚持看完书，让他们不断学习？这些想法使得Head First系列图书变得与众不同。从很多方面讲，这个选择也是《用户思维+》一书的基础。

在图书策划方面，编辑首先要考虑的方向是内容、作者还是读者？在每个方向上，你又会考虑哪些因素？

与大多数传统的出版商相比，我们有非常不同的观点（笑）。我们认为，图书就应该被看成是“产品”...…我们不把他们看成“读者”...…而是“用户”。图书一定要“很有用”并且“可以用”，所以它必须提供一种体验，以用户的思维方式为“用户”创造利益和带来改变。大多数技术书的最大的问题并不是质量不好，更多的是错误的思维方式。就像为做“好”

产品就强调质量一样，大多数技术书为做“好书”而做书，并没有考虑用户最想通过产品做什么事情。能做到“物美”的产品很多，却不一定被人们真正使用过。技术书也是如此。

“作者”应该被视为产品设计师或是体验设计师，而不是作者或作家。（除非是一位伟大的文学作家。）大多数的技术书作者并不能称为作家。如果我们把自己看作是用户体验设计师——我们为用户创造某种体验，从中用户（读者）可以按照他们（用户）的方式进行转换——那么，我们就获得了成功的最佳机会。

第一次看到你的书时，我就被badass这个字眼所吸引。在书中使用badass的用意是什么？

Badass也是导致很多人不喜欢这本书的一个字眼（撇嘴）。不过，我坚持使用badass是有一个很慎重的原因的。很长的一段时间里，我反复使用“创造热情的用户”这一说法，这也是《用户思维+》这本书以前的名字。从很多公司的合作中，我发现，这些公司希望他们的用户对他们（公司/品牌/产品）充满热情。“热情”这个词太容易被人理解成“公司的粉丝”这种意思，而这根本不是我的本意。我的意思是，“用户对你帮助他们做的事情饱含激情”。为此，我不得不反复解释这一点。我希望能有那么一个字眼，不太容易被大家误解。

Badass这个字，就不太容易解释为“对公司/品牌的热情”，并且可以把重点放在用户的能力上。

如果用户不想在某方面变得精通、擅长，根本没有动力从新手上升到高一点的水平呢?

我们在书中也谈到了这个问题。如果你在某个领域走得足够远，人们就会想要变得“更好”。任何一家公司也没有理由不去选择帮助用户在跟产品相关的事情上变得更好。这样做是为了帮助人们在某一方面变得更好……即使我们并不知道这个方面到底跟我们的产品如何相关。但是，它能帮助人们觉得自己更棒，因为我们帮助他们做更多的事情，继而连带着觉得我们也很棒。相比那些试图用尽所有方式来让人们认为我们了不起的做法，我们相信，这是一个更好利用营销预算的方式。

书中除了观点陈述之外，还提供了很多的认知理论支持。你是怎么得到这些知识的?

这些是我（很久以前）在加州大学洛杉矶分校研究和学习的东西。无论从实用性还是教学方面考虑，研究如何能够或是不能专注于有限的认知资源的问题，都是非常重要的。我们的工作就是——如果关心用户体验或是用户成长的话——帮助他们做出好的选择。

基于“操作性条件反射”（正强化或是间歇性可变奖励）的“成瘾”技术，并不是认知资源支持导向的而是“消耗”这些资源的。最好的解决办法并不是，“好吧，你的在线时间已经太长了，我们现在要关闭……”我们需要进一步提供一种体验，让他们可以转回到现实世界当中。这样，用户才能自己做出选择，心甘情愿地退出游戏。

很多时候，我们没有办法去决定产品的设计——产品就是产品——但是我们仍然可以通过用户手册、视频、社交媒体的宣传信息、用户群体等，在用户体验方面做很多事情。

最近，科学领域出现了一种可复制性危机。医学和心理学也有发现，原始实验不能在后续的研究中重现。所以，有些人开始认为自我损耗是一个错误。从这个意义上说，我们应该如何理解书中的观点“意志力和认知加工能力从同一个资源池中获取能量”？

认知资源方面的研究很多。尽管在复制“某些”特定的“自我损耗”的研究当中存在一些问题，但是毫无疑问，我们认为认知资源作为一种有限资源的观点得到了数百项不同研究的支持。当然，每一个单项的研究针对一个非常具体的问题，不同的研究会在甚至是完全相同的问题上得出互相矛盾或是完全相反的结论，但我们更关注大部分的研究，关注我们对有限的认知资源的了解。至于事情发生的真实机制是否完全被理解，（我们不太关注。）——嗯，我认为我们也永远不会了解那些真实的机制。科学研究越是深入到神经化学的兔子洞（这里将真理比喻成兔子洞），我们就越能意识到存在一种化学平衡，却不为我们知晓。支持某种学习行为的化学物质同样也会“抑制”某些学习行为，可能是因为量剂过高。比如，多巴胺在我们获取信息、知识、学习和体验的过程中发挥着极其强大的作用，但过量的多巴胺则会轻易地压倒或是“劫持”大脑，减少大脑的“创造性”和好奇心，等等。

我们希望——Bert 和我——从广泛的科学领域里，尽可能多地获取认知。我们把这种方式称为“冗余”的方法——这也是我们策划 Head First 系列图书时的原则。我们不是十分清楚哪些学习、心理、动机原则会起作用，但是我们会尽可能多地尝试不同的实践，希望至少能找到一些起作用的（笑）。

随着科学的发展，原有的理念很可能会受到挑战甚至是颠覆，我对这些都是十分欢迎的。一开始总是痛苦的，因为你发现一直以来那些奉行的东西也许并不是你当初认为的样子，你需要做出一些改变。但这始终是一次好机会，因为对我来说，当发现自己坚信并且奉行的东西可能是错误的时候，这就意味着还存在一个更好的方法（笑）。推到一切，重新再来。每一次的痛苦过后会紧随一段更激动的时刻，“哇——这个更好！”

根据成就用户（badass user）理论，如何策划出一本畅销书？

啊~~~这要弄清楚你真正想要回答什么样的问题。如果可能的话，提出一个比竞争对手“更棒”的问题。比如，竞争对手试图回答“如何策划出最好的书？”，这个问题在我们看来就不是核心问题，但大多数的图书都在试图回答这个问题。实际上，我们有很多更棒的问题值得回答。Head First系列图书首先回答，“我们如何确保人们可以从书中学习到东西？”即使是这样的问题，也不是真正的核心问题——因为它会引导我们进一步思考：人们实际上并没有认真地看完过一本技术书，他们又怎么能在不读完书的情况下学习到东西呢？因此，这让我们来到了中心问题，“怎样才能让人们真正地读完一本书？”我们发现，大多数的书在内容设计方面做得很棒，但读者就是不想读完。所以，我们就研究为什么人们不愿意读这些书。根据调查，阅读过程中的停滞不前，是因为读者来来回回翻看以前的内容，他们不是觉得错过了什么，就是觉得没理解或者是需要查看之前的图表、代码，或是其他一些东西。所以，我们认为，“如果他们反复地停下或是倒回，就不能继续前进”，也就找到了真正的问题，“我们怎样才能让人们坚持读完一本书？”

为了回答这个问题，我们做了其他人不曾做过的很多事情——一遍遍地重复示例，把话题情境不断推进，所以人们也就不用往回翻看了。我们

还尝试让他们明白，即使错过了什么内容，也没什么大不了的。大多数的人之所以选择不继续前进，是因为他们不确定自己是否进步了。他们不确定自己是否掌握了足够的知识以继续前进。我们就要不断地向他们保证，他们进步了；试着鼓励他们继续翻页，继续学习。为此，我们甚至学习了影视剧本写作方面的书（笑）。■

选自图灵访谈。

算法之于性能

作者 / 樽松谷仁等

日本 Oracle 株式会社技术顾问。曾在 Emprix 公司（美国本部）就职，为 SIer 和一般企业提供压力测试、性能管理等方面的咨询服务。之后就职于日本 Oracle 株式会社，还负责为使用 Java、WebLogic 、Exalogic 等中间件产品的客户提供咨询服务。

从技术视角严格来说，性能是非常模糊的术语。当一个人说“这是个高性能的应用”时，其实我们无从判断他说的是什么。他是说应用消耗的内存少？应用节约了网络流量？还是说应用使用起来非常流畅？总而言之，高性能有着多重的含义和丰富的解释方式。

不过，我们可以确认几种影响性能的主要因素。其中，算法的效率对性能有很大的影响。

算法

请想象一下，有一排连成长队的箱子，这些箱子中放着物品。那么，从这些箱子中找出目标物品的时间长短就代表性能的优劣。如果时间很短，就可以说性能很好。相反，如果时间很长，就称性能很差。

如果从队列的一头依次打开箱子，有 1000 个箱子，平均打开 500 个才能找到需要的物品。也就是说，我们需要从一头开始依次打开 500 个箱子。有没有效率更高的方法呢？如果把箱子里的物品标上序号排好的话，会更方便查找（图 1-1 上）。

比如说，我们要找标着数字“700”的物品。先打开最中间那个箱子，假设出来的是标着数字“500”的物品。因为要查找的数字比它更大，所以我们把目标转向这个箱子的右边部分。在右边的那一部分箱子之中，打开位于正中间的箱子，假设第 2 次找到的是标着“750”的物品。由于需要查找的数字比它更小，因此接下来就要查找这个箱子的左边。这样，通过有效地缩小查找范围，只需要很少的次数就能快速找到需要的数字（图 1-1 下）。

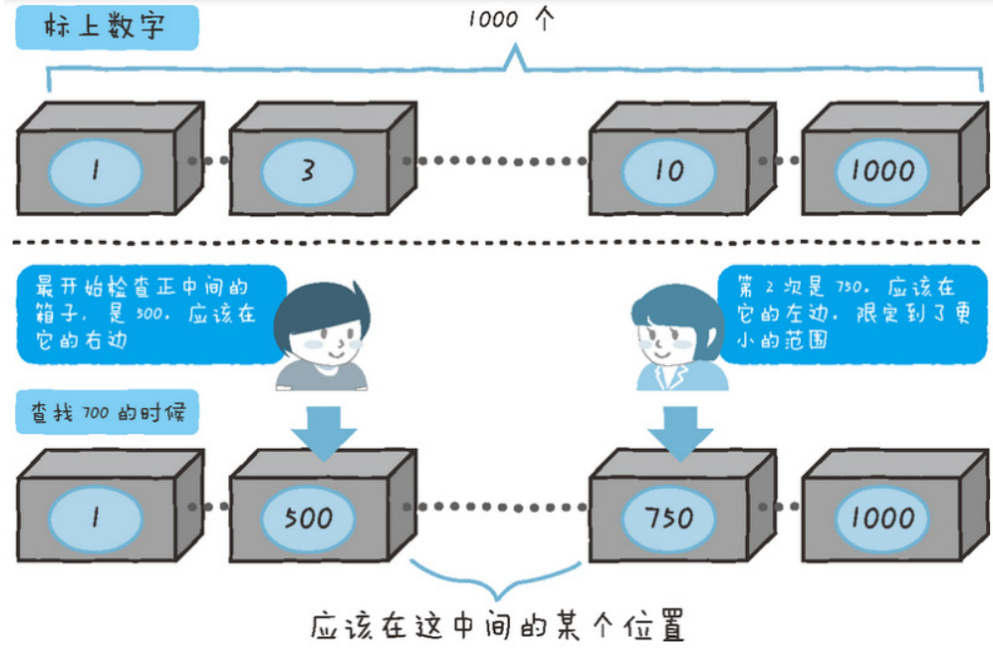


图 1-1　通过改进使查找起来更容易

与现实世界不同的是，这些箱子上都写着地址，如果知道了地址，就能立即打开那个箱子。例如，如果想要看第 100 号箱子的内容，不管自己在哪个位置，都能立即打开第 100 号箱子看里面的内容。

还可以在箱子的里面放入地址。例如，如果将第100号箱子的地址放入第99号箱子中，那么“100”这个数字就放入到了第99号箱子里（图1-2）。

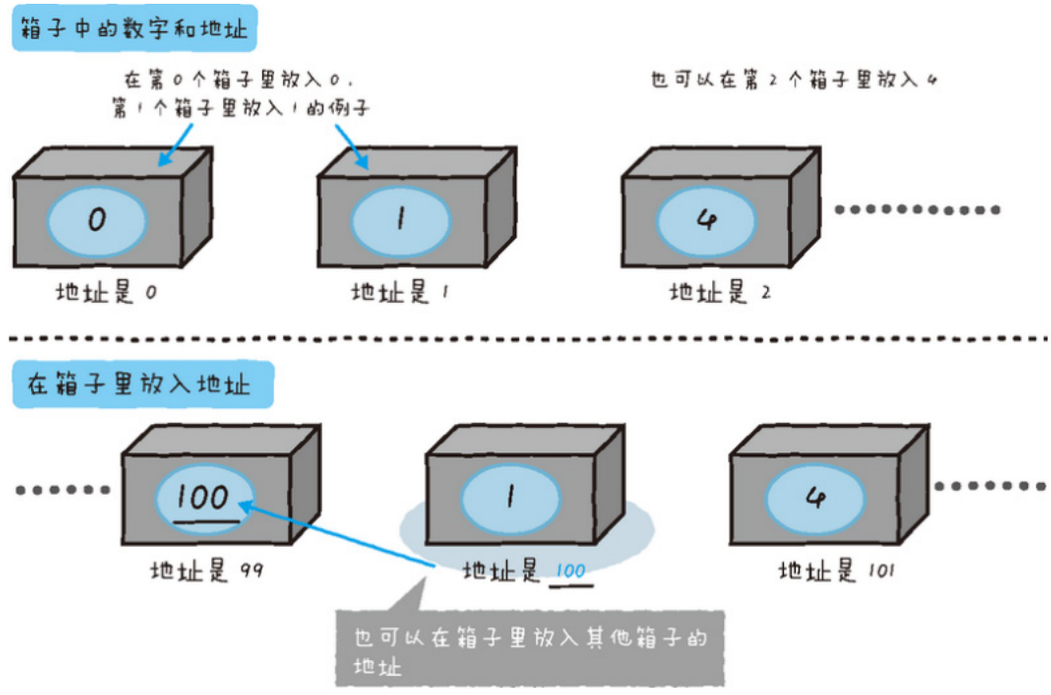


图1-2

“1. 箱子连成一排”“2. 地址”“3. 可以在箱子里面放地址”“4. 知道地址的话就能立即访问”，这4点就是算法的基础，也是计算机结构的基础。对此有些了解的人可能会从“可以在箱子里面放地址”这个比喻联想到C语言的“指针”。另外，也可能会有人从“知道地址的话就能立即访问”这个比喻联想到CPU处理的“物理内存”。有些人可能对指针有恐惧心理，觉得很难，其实理解指针的窍门就是区分“值”和“地址”。如果感到头脑混乱，请想一下这个原则。

接下来，让我们来考虑一下往箱子里放物品。假如要把从1到100的数字放到1000个箱子中。这里简单起见，假设从1开始按顺序放入数字。也就是说，在第1个箱子里放入1，第2个箱子里放入2，第100个箱子里放入100。这样一来，从101号箱子开始，之后的箱子都是空的（图1-3上）。接着，假设拿到了一个数字50.5，各位读者会怎么处理呢？这个问题其实没有正确答案。可以在101号箱子里放入50.5（图1-3中），也可以把51号之后的箱子里面的数字依次后移（图1-3下）。这里，我们把前者称为“方法1”，后者称为“方法2”。

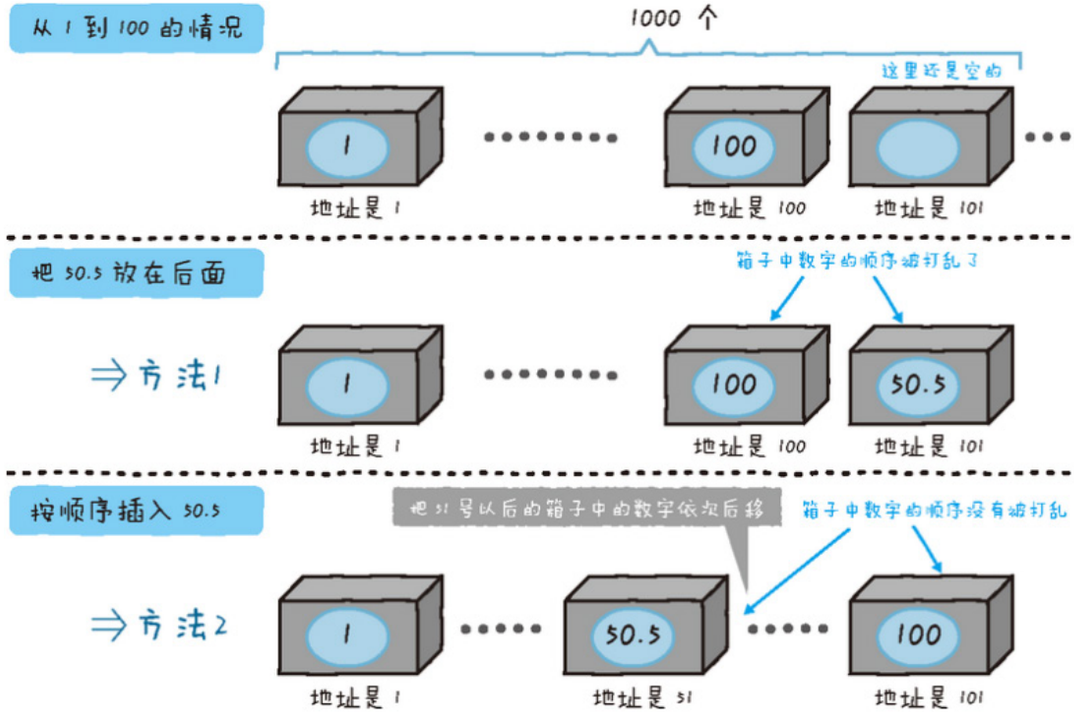


图1-3

学习算法的窍门

- 掌握优点和缺点

学习算法的窍门之一就是掌握算法的优点和缺点。那么前面列举的方法 1 和方法 2 各自的优缺点是什么呢？

方法 1 的优点是可以立即将数字放到箱子里。缺点是数字不再按照顺序排列，在查找数字的时候需要把所有的箱子都打开。

而方法 2 的优点是，由于数字是按照顺序排列的，因此可以使用前面介绍的从中间开始查找的方法。缺点就是在存放的时候把数字依次后移会很麻烦。

如各位所见，算法本身都有各自的优点和缺点。“折中”是一个很重要的思维。系统中的很多意外情况都是因为没有注意到这种折中所导致的。例如，由于将数据按照到手的顺序来存放，导致数据量增加时查找会耗费惊人的时间，等等。

另外，这个折中的思维不仅体现在算法上，在架构上也是一样的。

- 通过在图上推演来思考

接着介绍另一个学习的窍门。要理解性能，在图上推演是很重要的。如果读了文字说明后还是不理解，可以试着看一下图。可以的话，推荐自己画图，然后向别人说明。如果能够做到画图说明，可以说理解了算法（或运算指令）。实际去做一下就会知道，在自己尝试着去说明的过程中，会发现那些在理解上模棱两可的地方。

此外，不建议一上来就去掌握和说明一些异常操作，应该先从基本操作开始。之后，作为补充，再去掌握和说明更加详细的操作或异常操作等，这样就足够了。想要一下子理解细枝末节，只会事倍功半。笔者曾在某大型IT公司做了5年的培训老师，有很强的陈述和说明能力，不过还是强烈建议各位读者画图说明，以及在一开始先抛开细枝末节来理解。

对性能的影响程度

理解了算法的重要性后，接着我们来切身感受一下由于算法不同而导致的性能差别。我把Mega（M）作为标准单位来使用。Mega是表示100万的单位，然后把算法的好坏放在处理100万个数据的情况下来考虑，会更容易理解。

首先，我们考虑一下从100万个数据中找出某一个特定数据的情况。假设检查1个数据需要花费1毫秒。使用从一头开始逐个检查所有数据的算法的话，因为从概率论来说要检查一半数据后才能找到要找的数据，所以可以计算出需要花费50万次×1毫秒，也就是500秒。

接着，我们来考虑一下对于已经排好序的数据，一半一半地来查找的算法。首先，检查100万个数据的正中间的数据，假设数值是“50万”。如果要查找的数据比50万小，就查找左半部分；如果比50万大，就查找右半部分的数据集，以此类推。这样，每检查1次就可以把查找范围缩小一半。虽然一开始有100万个查找对象，但检查1次后，查找对象就变成了50万个，接着变成25万个，再接着变成12.5万个。那么，什么时候查找对象会变为1个呢？通过计算我们可以知道大概是在第20次的时候（图1-4）。检查1个数据花费1毫秒的话，总共就是20毫秒，与500秒相比就是一瞬间的事。读者可能会想：“当然不会使用从头开始

依次检查这样没有效率的方法了！”但是计算机经常会被迫进行这样没有效率的处理，只是使用的人没有意识到而已。

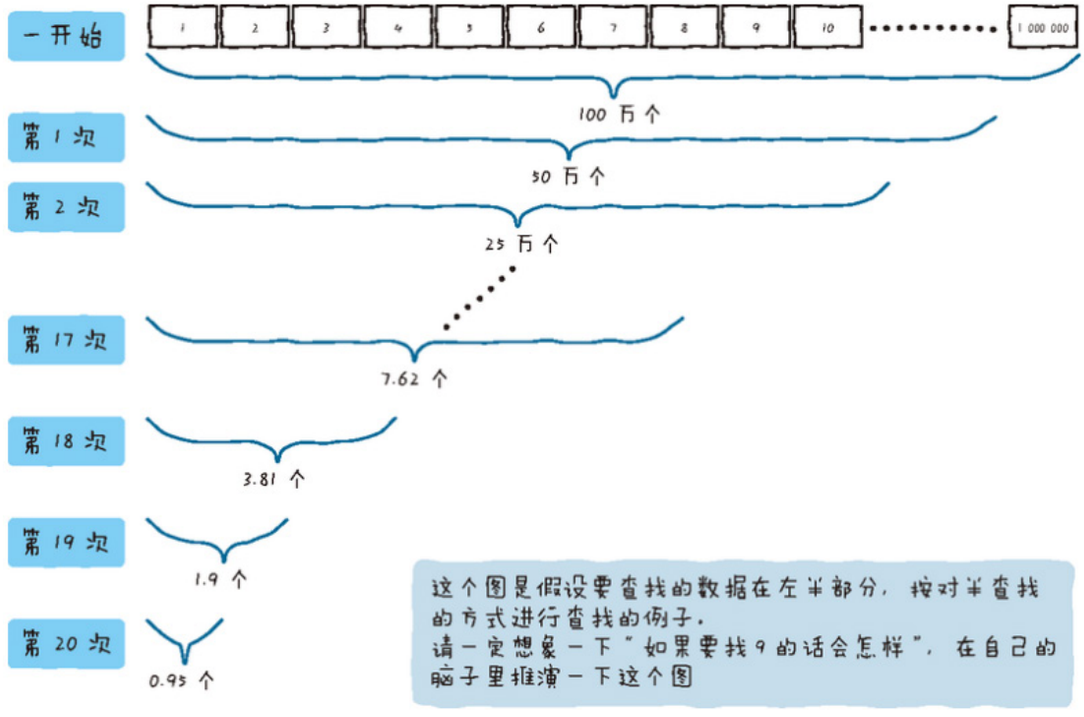


图 1-4

不过，一些一线程序员可能会想：“从头开始依次查找的方法和对半查找的方法相比，查找 1 个数据的时间应该是有差别的。从头开始查找的方法程序写起来比较简单。”的确，使用从头开始查找的方法，程序的编写比较轻松，处理也很简单，因此要查找 1 个数据花费的时间也很短。而使用对半查找的方法，需要记录自己检查到了哪个位置，还要计算下一步把哪部分数据对半，这是相对费事的。但其花费的时间是第一个方法的 1.5 倍？还是 2 倍？还是 3 倍？这里希望大家能有一种感觉，那就是这里所花费的时间从整体来看是很微小的，完全可以忽略。即使是 2 倍，

那所花费的时间也只是40毫秒，3倍的话也只是60毫秒。与从头开始查找所要花费的500秒比起来，差别依旧是很大的。

基于以上原因，在比较算法优劣的时候，会忽略掉一些微小的系统开销。我们应该关注的是随着数据个数的变化，所花费的时间会以怎样的曲线发生变化。因为有一些算法，在数据只有一两个的时候性能很好，但是当数据个数达到几千到几万的时候，性能会急剧下降。

评价算法的指标

将数据的个数用变量n来表示，让我们来比较一下直线y=n和曲线$y=n^2$。可以发现$y=n^2$的值会急剧变大（图1-5）。即使是用y=2n这条直线来与曲线$y=n^2$作比较，差别也依然很大。在这里，2n的“2”对整体是不产生影响的，这就属于前面提到的“可以忽略的系统开销”。

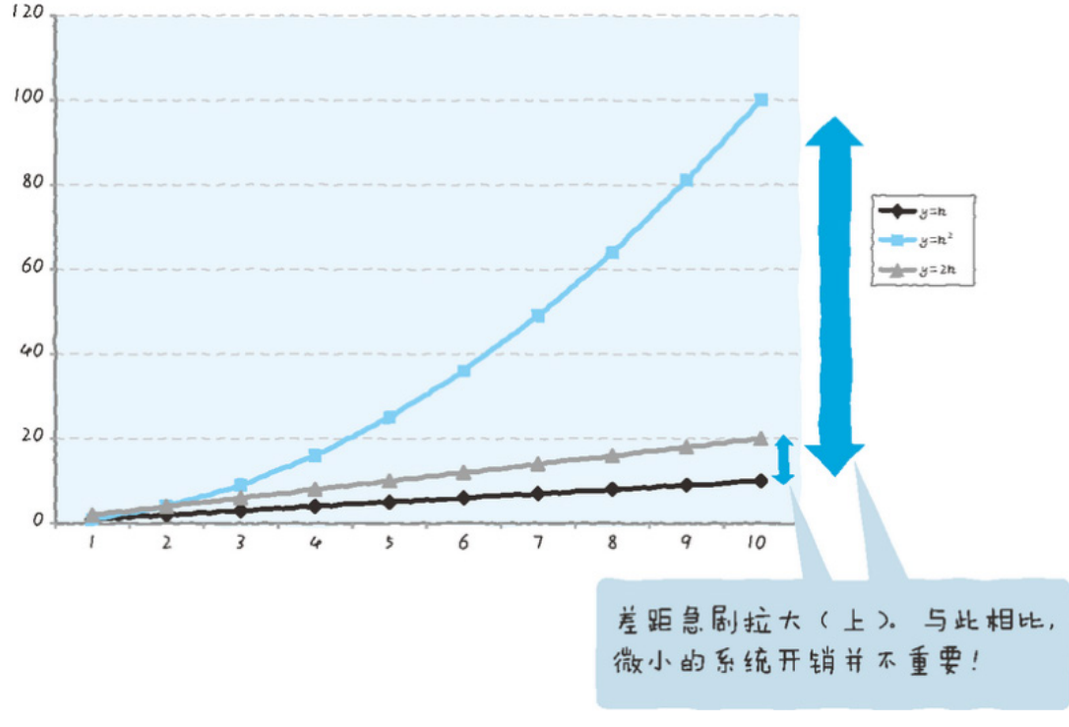


图1-5　数据个数与所需时间的关系

计算机原则上是处理大量数据的东西，所以我们只关心当数据量变大时决定性能优劣的关键（Key）。这个关键就是“复杂度”（Order）。在前面列举的y=n或y=2n中，其复杂度标记为O(n)。而O(1)则表示不会受到数据量增加的影响。作为算法来说，这就非常好了。

打个比方，查找最小值的情况下，如果要从分散的数据的一端开始搜索，就需要查看所有的数据。换句话说，复杂度是O(n)。但若数据是按顺序排列的，那第1个数据就是最小值，所以只要查看1次就完成了，复杂度就是O(1)。若数据个数是5个或10个这样的小数字，两者花费的时间差距可能只有几毫秒。但是若数据量达到了100万个会怎么样呢？这就会产生大约1000秒的差距。

接下来，我们就针对“查找数据”这种对计算机来说非常基础的操作，来通过复杂度判断一下算法的优劣。刚才已经介绍了将所有数据从头到尾查找一遍的方法。此外，还有一个非常有名的能提高数据查找效率的方法叫作“树”（图1-6）。

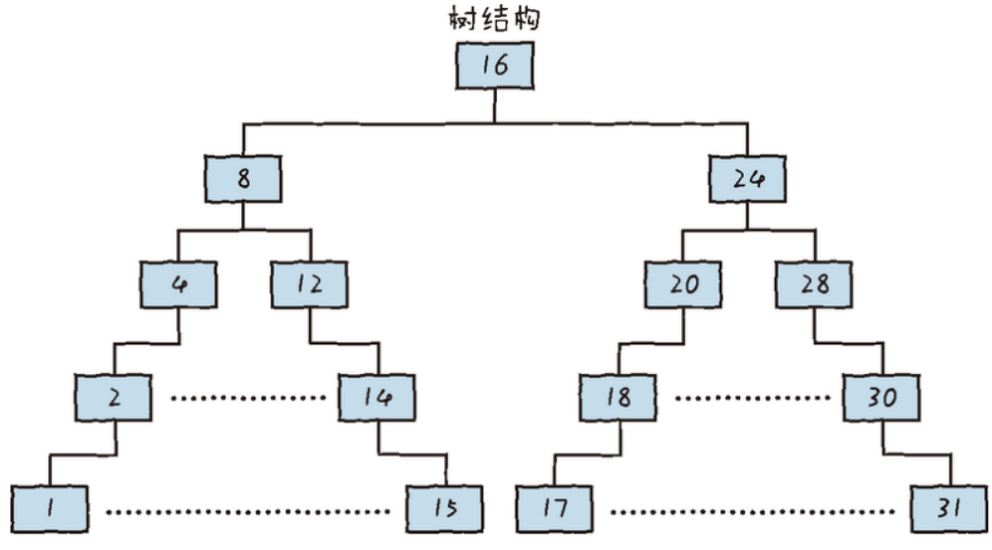


图1-6　树结构

《图解性能优化》列举了丰富的实例，并结合直观的插图，向读者传授了有用的实战技巧。另外，因为系统性能和系统架构密切相关，所以读者在学习系统性能的过程中还能有效地学到系统架构的相关知识。

在树的根节点放置 1 个数据，接着将比它大的数据放在右边，比它小的数据放在左边，以此方式进行分类。对分类到左右两边的数据，以同样的方法进行分类。这样就会生成一个像“树”一样的结构。从树的根节点开始查找，直到找到目标数据为止，这一过程就是把数据对半分的操作。这种操作的复杂度标记为 O(logn)。logn 指的是把 n 除以 2 多少次后会变为 1。大体来说，它的复杂度处于 O(1) 和 O(n) 之间，用图来表示的话，可以看到即便数据变大，O(logn) 曲线也只是缓缓地上升（图 1-7）。

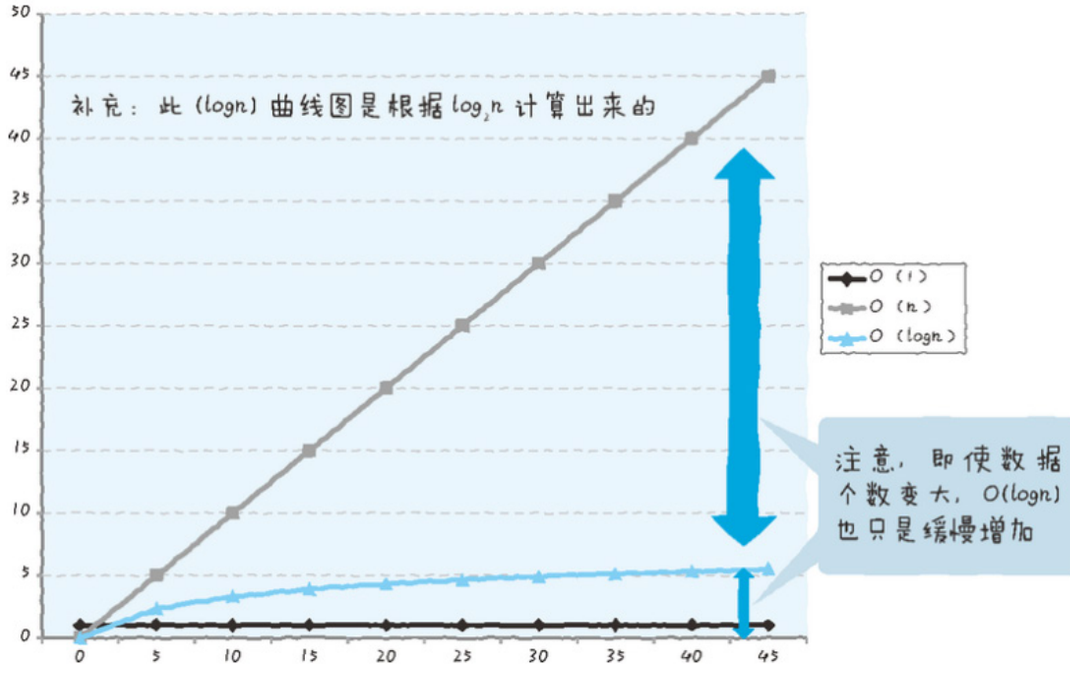


图 1-7　数据个数与所需时间的关系

在数据量小的时候，O(n) 的性能有时可能会超过 O(1)，但当数据量变大时，一定是 O(1) 胜出。在评价算法优劣的时候，首先要考虑到数据量很大的情况。■

强迫性调优障碍

作者/Christian Antognini

资深数据库专家，从1995年就开始致力于探究Oracle数据库引擎的工作机制。长期关注逻辑与物理数据库的设计、数据库与Java应用程序的集成、查询优化器以及与性能管理和优化相关的各个方面。目前任瑞士苏黎世Trivadis公司首席顾问和性能教练，是OakTable网站核心成员。

如果你曾仔细地定义过性能需求，那么很容易就可以确定应用程序是否正在遭遇性能问题。如果没有仔细定义过性能需求，那么答案很大程度上会取决于回答者的主观判断。

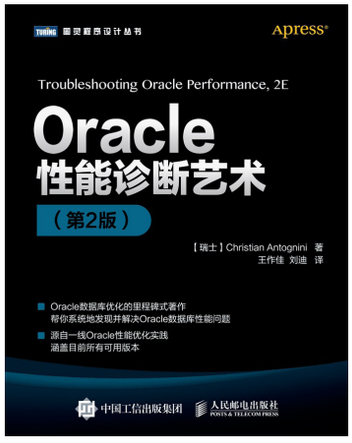
有趣的是，实际上导致应用程序性能被质疑的大部分情形都可以归纳为下面的少数几类。

- 用户不满意应用程序当前的性能表现。
- 系统监控工具警告某个基础组件正遭遇超时或不寻常的负载。
- 响应时间监控工具通知你某个服务级别协议没有得到满足。

第二点和第三点的区别尤为重要。基于此，接下来的内容将简要描述这些监控方案。之后，再来看一下某些看似有必要、实则没必要进行优化的情况。

系统监控

系统监控工具基于一般系统统计信息进行健康检查。其目的是识别出不寻常的负载模式以及故障。虽然这些工具可以同时监控整个基础设施，



《Oracle性能诊断艺术（第2版）》Oracle数据库优化的里程碑式著作，帮你系统地发现并解决Oracle数据库性能问题。

但这里需要强调的是，它们只监控单独的组件（例如主机、应用服务器、数据库或存储子系统），而不考虑组件间的作用关系。因此，对于拥有复杂基础设施的环境，在支撑基础结构的单个组件出现异常时，很难或者几乎不可能评估异常对系统响应时间的影响。其中一个例子就是高频度地使用某一资源。换句话说，系统监控工具发出警报只是说明应用程序或者基础设施中的某些组件可能出现了问题，而用户根本没有察觉到任何性能问题（称作误报）；反之，也可能出现用户正在遭遇性能问题，而系统监控工具并没有发现问题（称作漏报）。最常见、也是最简单的关于误报和漏报的例子，是监控一个拥有许多CPU的对称多处理系统的CPU负载情况。假设你有一个装有四颗四核CPU的系统。当看到使用率在75%左右时，你可能觉得这太高了，系统受到了CPU的限制。但是，如果执行任务的数量远大于处理核心数量，那么这样的负载就是很正常的。这便是一个误报。反之，当你看到CPU使用率大约为8%时，你可能觉得一切正常。但是如果系统正在执行一个没有并行的单任务，那对于这个任务来说可能瓶颈就是CPU。实际上，100%的1/16只有6.25%，因此单个任务的CPU使用率不能超过6.25%。这便是一个漏报。

响应时间监控

响应时间监控工具（也称为应用程序监控工具）基于由机器人产生的假想事务或者由终端用户产生的真实事务进行监控。这些工具测量应用程序处理关键事务的时间，如果时间超出预期阈值，它们就会发出警告。换句话说，它们和用户一样利用基础设施，也会像用户那样“抱怨”糟糕的性能。因为它们从用户的角度监控应用程序，所以这些工具不仅能检查单个组件，更重要的是，它们还能检查整个应用程序的基础设施。因此它们专门用于监控服务级别协议。

强迫性调优障碍

曾经有一段时间，大部分数据库管理员都患上一种叫作“强迫性调优障碍”[1]的病症。其症状是过多地检查性能相关的统计信息（大部分都是基于比率的），从而无法集中精力关注真正重要的事情。他们简单地以为应用某些“简单”规则，就能优化所管理的数据库。历史告诉我们，结果并不总是尽如人意。为什么会出现这样的情况？所有用来检查某个给定比率（或值）的规则都是独立于用户体验而制定的。也就是说，误报和漏报都是规则而非意外。更糟糕的是，大量的时间消耗在这些任务上。

举例来说，时不时会有数据库管理员向我提出这样的问题：“我注意到我们的一个数据库在某个闩（latch）上有大量的等待。怎么做才能减少或者最好能消除这些等待？”一般我会回答：“你的用户因为这个闩锁抱怨过么？肯定没有，所以不用担心。反倒应该问问他们认为应用程序的问题有哪些。然后分析这些问题，你就会知道这个闩锁上的等待到底有没有影响到用户。”

虽然我从未做过数据库管理员，但是必须承认我也患过“强迫性调优障碍”。现在我和其他大多数人一样，克服了这个病症。只不过与其他恶疾一样，彻底治愈“强迫性调优障碍”需要花很长时间。有些人根本没有意识到患了这种病症；有些人意识到了，但是因为多年的沉溺，总是很难去认识这样一个大错误并改掉陋习。■

[1] 这个绝妙的名词是由Gaya Krishna Vaidyanatha 发明的。你可以在 Oracle Insights: Tales of the Oak Table（Apress，2004）一书中找到它的相关讨论。

网络较差的情况下，怎样提供流畅的ios应用体验

作者/Gaurav Vaish

就职于雅虎公司的移动和新兴产品团队，为每月有数亿人使用的移动应用创建优雅的可重用方案。他曾是IIT全球指导计划的成员，还在印度班加罗尔创立了InColeg Learning 及 Edujini Labs有限公司。

在移动端应用中使用网络是必不可少的，但减少网络延迟的方法却是有限的，因此，你应该着手对网络条件进行最大程度的优化，并预先对不同的场景进行规划。

这篇文章将重点关注影响整体延迟的因素，并讨论如何充分利用现有的信息来最大程度地提高性能。

在网络中完成的大多数工作是你无法控制的，因此确定衡量的标准非常重要。我们将在这部分讨论一些较为重要的性能相关指标。请注意，此处并非要列举出所有的指标，只是挑出在性能优化相关的测量中更为重要的一些指标。

图1展示了典型的网络请求全景图。

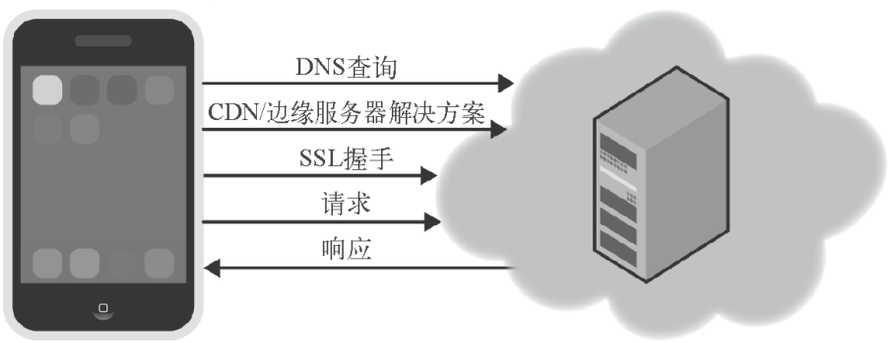


图1　网络——设备至服务器

后续讨论的大致结构是：先进行相关指标的说明，然后是一个或多个示例，最后是最佳实践。

DNS 查找时间

发起连接的第一步是 DNS 查找。如果你的应用严重依赖网络操作，DNS 的查找时间会使应用变慢。在一个关于两个位置的统计样本中，从加利福尼亚州的森尼韦尔市访问 www.google.com 时，DNS 查找时间是 2846 毫秒，而从印度的新德里访问时只花费了 34 毫秒（见图 2）。

查找时间与主 DNS 服务器的性能成函数关系。最终的连接时间与追踪到目的 IP 地址的路由成函数关系。

使用内容分发网络（content delivery network，CDN）将延迟最小化是一种常见的做法。在图 2 中，你应该注意到 www.google.com 在两个地点解析的 IP 地址是不同的。在森尼韦尔市，它被解析成了美国的一台服务器，而在新德里，它被解析成了印度的服务器。但因为 DNS 会为每一个独有的子域名进行查找，所以，拥有多个 CDN 主机名会导致应用的速度变慢。

为了最大限度地减少 DNS 查询时间所产生的延迟，你应该遵循以下的最佳实践。

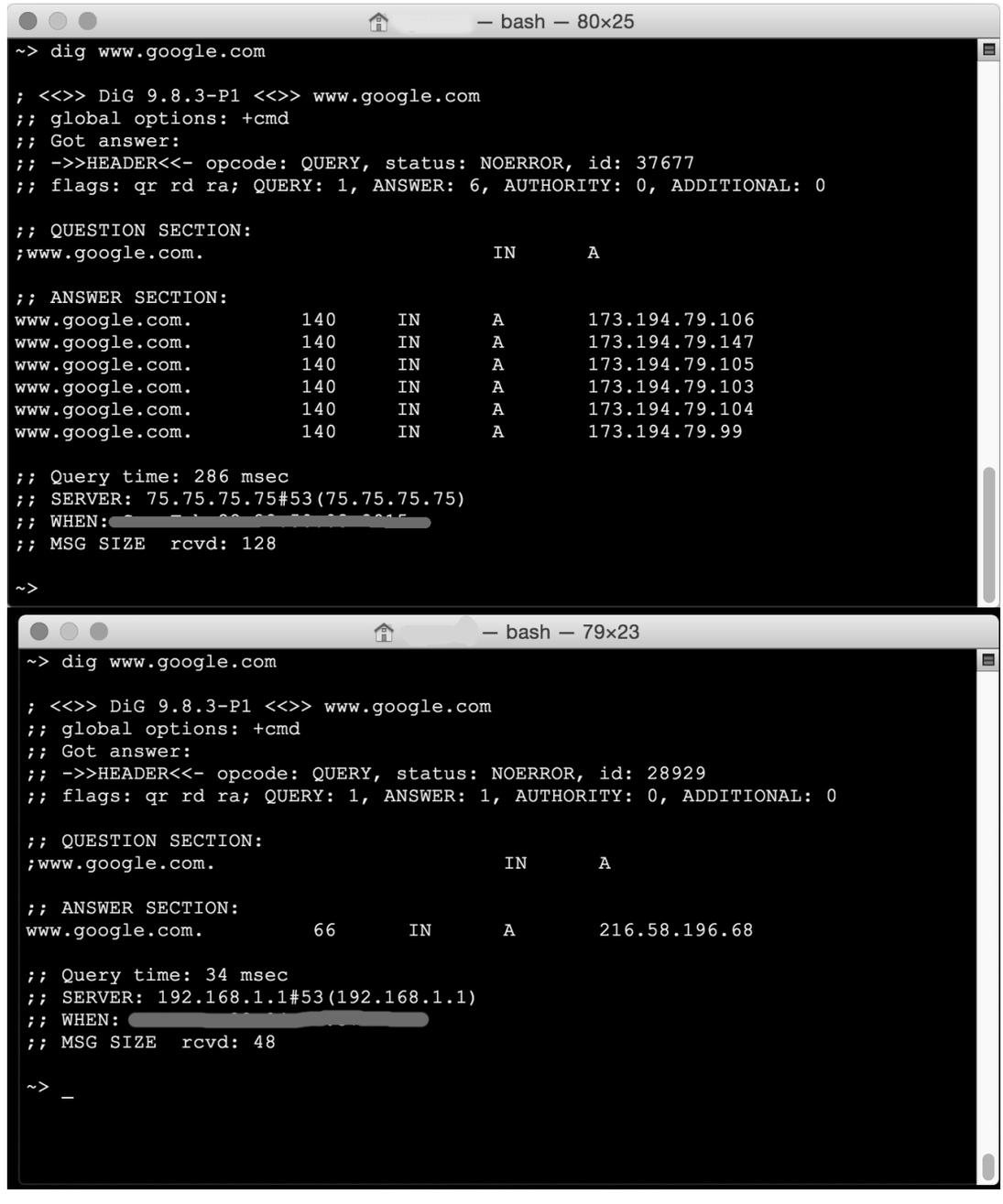


图2　www.google.com 的 DNS 查询时间——加州的森尼韦尔市（上）和印度的新德里（下）

- 最小化应用使用的专有域名的数量。按照路由的一般工作方式，多个域名是不可避免的。最好是能做到以下几点：

(1) 身份管理（登录、注销、配置文件）

(2) 数据服务（API 端点）

(3) CDN（图片和其他静态人工产品）

有可能需要其他域名（例如，用于提供视频、上传检测数据、具体的子数据服务、广告投放，甚至是国家特定的全球本地化）。如果子域名数量上升至两位数，那么势必会引发担忧。

- 在应用启动时不需要连接所有的域名，可能只需要身份管理和初始画面所需的数据。对于后续的子域名，尝试更早地进行 DNS 解析，也被称为 DNS 预先下载。为实现此操作，你可以参考以下两点。

如果子域名和主机在控制范围内，你可以配置一个预设的 URL，不返回任何数据，只返回 HTTP 204 的状态码，然后提前对该 URL 发起连接。

第二个方法是使用 gethostbyname 执行一个明确的 DNS 查找。然而，针对不同的协议，主机可能会解析至不同的 IP，例如，HTTP 请求可能会解析至一个地址，而 HTTPS 会解析至另一个地址。虽然不是很常见，但第 7 层的路由可以根据实际的请求解析 IP 地址，例如，图像是一个地址，视频是另外一个地址。鉴于这些因素，在连接之前解析 DNS 经常是无用的，对主机进行伪连接会更有效。

SSL握手时间

为了安全起见，可以假设应用中所有的连接均是通过TLS/SSL的（使用HTTPS）。HTTPS在连接开始时，先进行SSL握手，SSL握手主要是验证服务器证书，同时共享用于通信的随机密钥。这一操作听起来简单，但是却有很多步骤，还会耗费较多时间（见图3）。

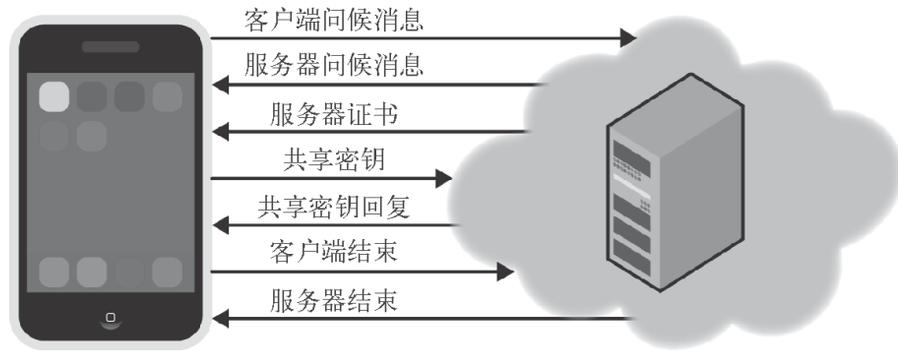


图3　SSL 握手

你应该遵循以下的最佳实践。

- 最大程度地减少应用发起的连接数。因此，也需要减少应用连接的独有域名的数量。
- 请求结束后不要关闭HTTP/S连接。

为所有的HTTPS请求添加头Connection: keep-alive。这确保了同样的连接在下一次请求时可以复用。

- 使用域分片。如此一来，虽然连接的是不同的主机名，你也可以使用同一个socket，只要它们解析为相同的IP，可以使用相同的证书（例如，在通配符域）就行了。

域分片在SPDY及其后续版本——HTTP/2（https://http2.github.io）——中是可用的。你需要一个支持上述任意一种格式的网络库。

网络类型

由于用户逐步丢弃了桌面设备，这也就放弃了总是处于连接状态的高速带宽网络，转而使用有同等质量的WiFi网络或使用间歇连接的带宽可变的移动网络。更有挑战的场景是，用户是移动的。当设备在移动信号塔之间切换时，网络和质量也会发生变化。一个设备可能在任何时刻从LTE网络切换到GPRS或进入无信号区域。对于这种情况，人们往往束手无策。

主机的可到达性

你可以使用苹果公司的可到达性库（https://developer.apple.com/library/ios/samplecode/Reachability/Introduction/Intro.html）或使用Tony Million对该库的简易替换（https://github.com/tonymillion/Reachability），主机的可到达性发生变化时，替换的库支持回调的调用。

如果设备闲置超过几秒（具体的值是不确定的，这里的几秒也可能变成几分钟），网络无线电可能已经关闭，这将导致无线资源控制器发生额外的几百或几千毫秒的延迟。

先确定主机的可到达性，从而确保应用具备处理此种场景的能力。

一般情况下，iPhone和iPad可以使用下列任何网络连接到互联网。

- WiFi

如果WiFi网络是私有网络（如家庭或办公室连接），那么你可以期望连接至互联网的网络是持续且质量较好的。

但是，处于WiFi网络中并不能保证一定连接上了互联网。例如，当设备连接到某一公共热点（例如，在旅馆或购物商场）时，如果用户没有成功地提供适当的凭证，那么将无法访问互联网。

即使设备已经连接至互联网，也可能有一些限制，比如可连接的域名或端口限制。举个例子，www.google.com或www.yahoo.com域可能是允许的，但mail.google.com或mail.yahoo.com却可能无法使用。

- 4G：LTE、HSPA+（高速的数据网络）

这些是最新一代的数据网络。在第一个真正的业务数据发送前，一般会有100~600毫秒的延迟。这些网络以亚毫秒的间隔动态地创建无线相关的资源，并且爆发性地发送数据。理论上来说，速度会从100Mbps浮动至1Gbps。对于高速率移动的通信，如在汽车或火车上，速度可能会在100Mbps；对于低速率移动的通信，如行人或静止的用户，速度可能会达到1Gbps。

- 3G：HSDPA、HSUPA、UMTS、DMA2000（中等速度的数据网络）

这些是上一代的数据网络，但使用频率可能比LTE更频繁。

3G的速度可能会从200Kbps变化至超过50Mbps。双向的传输速度可能并不对称。HSDPA具有较高的下行速度，HSUPA具有较高的上行速度。

- 2G：EDGE、GPRS（低速的数据网络）

20世纪90年代的网络仍然没有消亡。这些都是初始数字网络（第1代网络使用模拟信号），提供了较低的带宽。EDGE理论上具有500Kbps的极限速度，而GPRS最高只能达到50Kbps。

可到达性库可以给出访问主机的详细网络信息。使用此信息来确定传输内容的类型（例如，文本、图像或是视频），以及传输的各种项目的大小，等等。

0.2%的数据传输；46%的功耗！

2011年，密歇根大学和AT & T发布了“移动应用性能资源使用情况”（http://mobilityfirst.winlab.rutgers.edu/documents/mobisys11.pdf），这篇研究性论文分析了移动应用的网络使用和功耗效率。

论文探讨了潘多拉公司，将其在移动网络的间歇性网络传输的低效率问题作为经典案例研究。虽然问题已经修复了，但案例分析还是值得阅读的。

当播放一首歌曲时，应用会将这首歌全部下载，这是正确的行为：下载尽可能多的数据，让无线电关闭的时间尽可能长。

但是，在传送之后，应用会每60秒定期地发送检测事件。这些事件仅占传输总字节的0.2%，但却占了应用总功耗的46%。

事件的数据通常是比较小的，但因为无线电在较长的时间都会保持激活状态，所以它将应用的电量消耗增加了一倍。

通过将这些数据分发至一些请求之中，或在无线电已经处于激活状态时再发送数据，就可以消除不必要的能量拖尾，实现更高的电源效率。

为确保你的应用不会成为类似案件研究的一部分，在开发以网络为中心的应用时，你可以遵循以下的最佳实践。

- **设计时考虑不同的网络可用性。**在移动网络中，唯一不变的是，网络可用性是多变的。对于流媒体，最好选择HTTP实时流或任何可用的自适应比特率流媒体技术，这些技术可以在某一时刻针对可用带宽进行动态切换，切换至当前带宽的最佳流质量，从而提供流畅的视频播放。

对于非流媒体内容，你需要实现一些策略，确定在单次拉取时应该下载多少数据，并且数据量必须是自适应的。例如，你可能不希望在最新一次更新时，一次拉取所有的200封新邮件。你可以先下载前50封邮件，再逐步下载更多邮件。

同样，在低速网络时，不要打开视频自动播放功能，这可能会花费用户很多钱。

对于自定义的非流媒体数据拉取，要保持对服务器的关注。让客户端发送网络特征数，服务器决定返回的记录条数。这样一来，你可以在不发布应用新版本的情况下进行适应性改变。

- 出现失败时，**在随机的、以指数增长的延迟后进行重试。**

例如，第一次失败后，应用可能会在1秒后重试。第二次失败时，应用在2秒后重试，接着是4秒的延迟。不要忘记对每个会话设置最多的自动重试次数。

- **设立强制刷新之间的最短时间。**当用户明确要求刷新时，不要立即发出请求。相反，检查是否已经存在一个请求，或当前请求与上次请求的时间间隔是否小于阈值。如果满足上述条件，则不要发送此次请求。

- 使用**可到达性库**发现网络状态的变化。如图4所示，使用指示条向用户展示不可用的状态，毕竟设备没有网络连接并非你的错。通过让用户了解潜在的连接问题，可以避免你的应用受到指责。

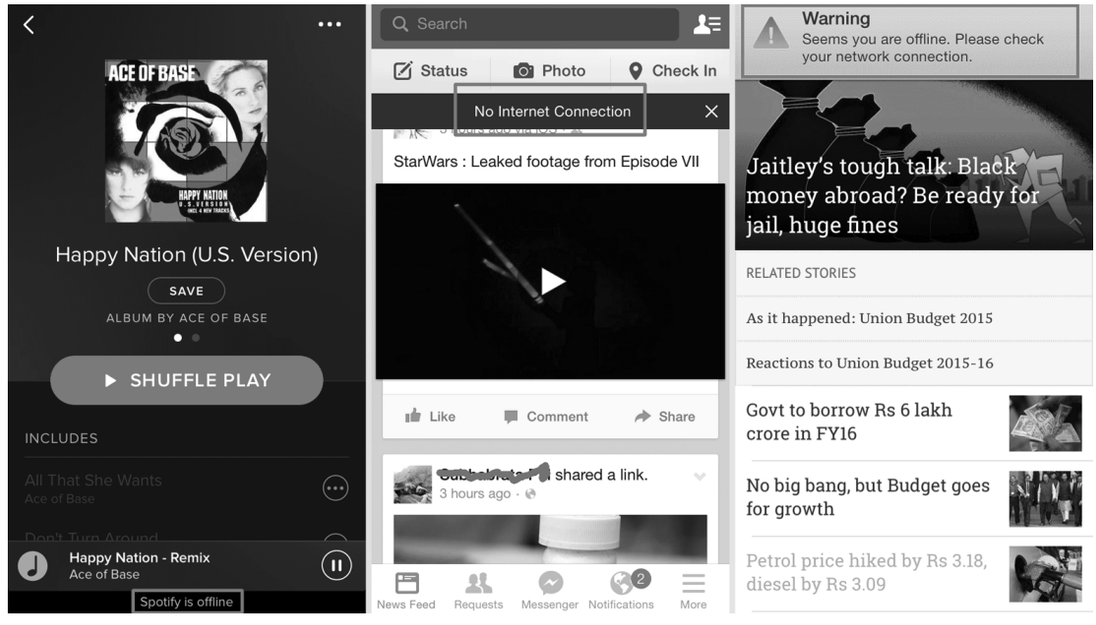


图4　Spotify、Facebook 和 TOI 针对离线网络状态的指示条

- **不要缓存网络状态。**不论是通过触发请求时的回调来获取状态，还是在发送请求之前显式地检查状态，要始终使用网络敏感度高的任务的最新值。

- **基于网络类型下载内容。**如果想要展示一个图像，不用总是下载原始的、高质量的图像。应该始终下载和设备适配的图像——iPhone 4S 所需的图像尺寸和第三代 iPad 所需的差别很大。

如果应用有视频内容，最好有一个与之关联的预览图像。如果应用支持自动播放功能，在非WiFi网络中只展示预览图像，因为自动播放会花费用户很多钱。

此外，针对图像、音频和视频等，提供一个关闭自动下载或关闭自动播放功能的选项。图5展示了WhatsApp应用中相关的设置。

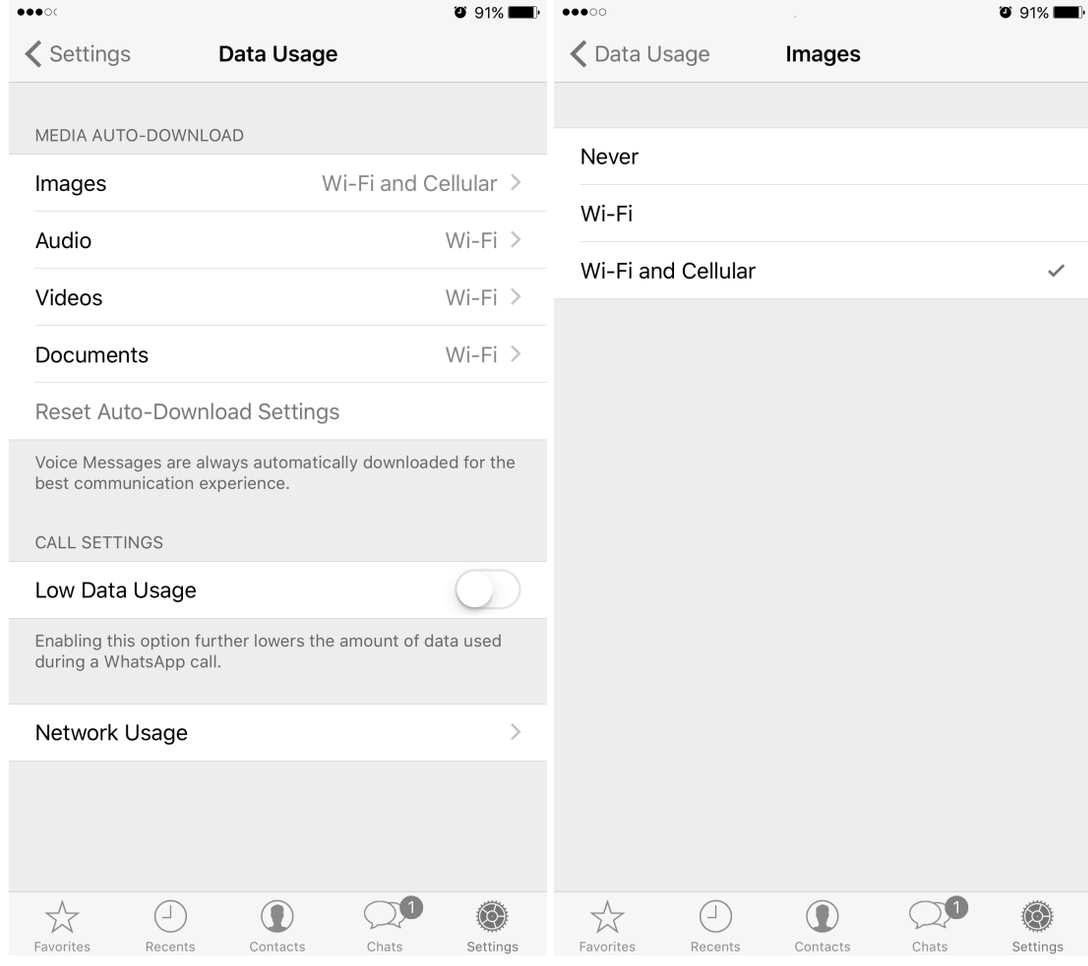


图5　WhatsApp 中对图像、音频和视频内容下载的可选设置

- **乐观地预先下载。**在 WiFi 网络中预先下载用户在后续时刻需要的内容。随后就可以使用缓存内容了。最好分次下载内容，在使用之后关掉网络连接，这有助于节省电量。

> 预先下载永远都是争论的焦点。在下载最少数据和获取最近可能需要使用的所有内容之间，人们总会存在激烈争执。
>
> 没有黄金法则可循。这在很大程度上取决于平均数据的大小、完成的下载数、预期的使用模式和网络条件。如果网络不断变化，而且你需要执行最小的数据传输，看看是否可以分批处理这些请求。

- 如果适用，当网络可用时，**支持同步的离线存储**。通常情况下，网络缓存就足够了。但如果需要更多的结构化数据，使用本地文件或 Core Data 会是一个较好的选择。

对游戏来说，缓存最近一级的详细信息。对邮件应用来说，存储一些带有附件的最新电子邮件是一个不错的选择。

根据不同的应用，你可能会允许用户创建新的离线内容，离线内容会在网络连接可用时和服务器进行同步。例如，在邮件应用中编写新邮件或回复某封邮件时；在社交应用中更新资料图片时；拍摄将要上传的照片或视频时。

总是将网络和通信与 UI 解耦。如果应用可以进行离线操作，那么就通知用户离线操作是可行的，否则就通知用户不能进行离线操作。不要让用户已经开始和应用进行交互之后，获取不到返回值。这会是一个糟糕的用户体验。

图6展示了离线模式下的Facebook和E*Trade应用。

Facebook应用通知用户网络不可用，但允许发布评论或状态更新。当网络可用时，上述内容会在后续进行同步。在E*Trade应用中，用户也可以与应用进行交互，但当查找某一股票的报价时，应用会进入死胡同，这会导致糟糕的用户体验。

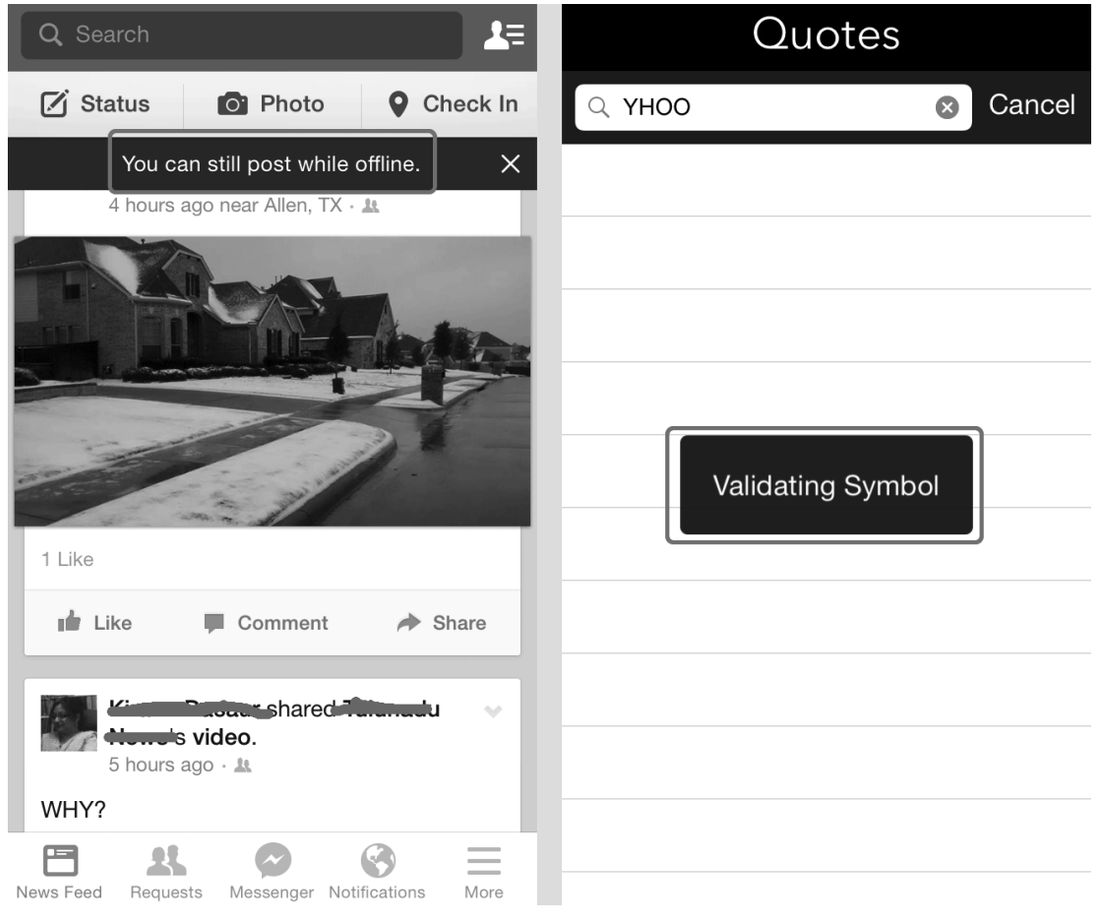


图6

需要注意的是，网络条件总是会超出应用的控制，但在这些限制范围内提供的用户体验却是受应用控制的。要将可用选项做到最好，这些选项包括离线存储、网络可到达性、网络类型、执行（或不执行）网络操作，通知（或不通知）用户相关的信息。

延迟

延迟是指从服务器请求资源时，在网络传输上花费的额外时间。设置用于测量网络延迟的系统是很重要的。

网络延迟可以通过使用请求过程中花费的总时间减去服务器上花费的时间（计算和服务响应）来测量：

```
Round-Trip Time = (Timestamp of Response - Timestamp of Request)
Network Latency = Round-Trip Time - Time Spent on Server
```

花费在服务器上的时间可以由服务器来计算。对客户端而言，往返的时间是准确可用的。服务器可以将花费的时间放在响应的自定义头部，然后客户端就可以用来计算延迟了。

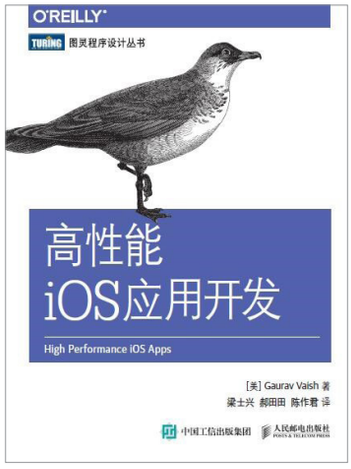
下面，我们看下计算延迟的示例代码。该代码假设响应包含了自定义头部X-Server-Time，这个时间以毫秒为单位，包含在服务器上花费的时间。

```
//server - NodeJS
app.post("/some/path", function(req, res) {
    var startTime = new Date().getTime();
    //处理
    var body = processRequest(req);
    var endTime = new Date().getTime();
```

```
    var serverTime = endTime - startTime;
    res.header("X-Server-Time", String(serverTime));
    res.send(body);
});

//client - iOS app
-(void)fireRequestWithLatency:(NSURLRequest *)request {

    NSDate *startTime = nil;
    AFHTTPRequestOperation *op =
        [[AFHTTPRequestOperation alloc] initWithRequest:request];
    [op setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *op, id res) {

        NSDate *endTime = [NSDate date];
        NSTimeInterval roundTrip = [endTime timeIntervalSinceDate:startTime];
        long roundTripMillis = (long)(roundTrip * 1000);

        NSHTTPURLResponse *res = op.response;
           NSString *serverTime = [res.allHeaderFields objectForKey:@"X-Server-
Time"];
        long serverTimeMillis = [serverTime longLongValue];

        long latencyMillis = roundTripMillis - serverTimeMillis;

    } failure:^(AFHTTPRequestOperation *op, NSError *error) {
        //处理错误。如果需要，向用户展示错误
    }];

    startTime = [NSDate date];
    [op start];
}
```

上面的代码关于网络延迟的表达基本准确，但它包括了服务端线路上刷新数据花费的时间，以及在客户端解析响应花费的时间。如果可以分离出来，它将提供真实的网络延迟时间，包括任何设备开销。

如果你有数据来分析任何模式下的延迟，还需跟踪下列数据。



《高性能iOS应用开发》共5个部分，主要从性能的衡量标准、对应用至关重要的核心优化点、iOS应用开发特有的性能优化技术以及性能的非代码方面，讲解了应用性能的优化问题。

- 连接超时

跟踪连接超时的次数是非常重要的。根据网络质量（较薄弱的基础设施或较低的容量），该指标会提供详细的地理区域分类，网络质量将反过来帮助规划同步时间的传输。例如，同步会在短时间间隔传输，比如几分钟，而不用在某一个特定时间跨时区同步。

- 响应超时

捕捉连接成功但响应超时的数量。这有助于根据地理位置和日期、年份的时间来规划数据中心的容量。

- 载荷大小

请求以及响应的大小完全可以在服务器端进行测量。使用此数据可以识别任何可能降低网络操作速度的峰值，并确定一些可用选项：通过选择合适的序列化格式（JSON、CSV、Protobuf等）减少数据占位，或者分割数据并使用增量同步（例如，通过使用小的批量大小或在多个块中发送部分数据）。

较差网络容量的最大化

我曾经开发过一个神奇的体育应用，当时的工程师团队渐渐注意到应用的延迟变得更长，超时（连接和响应超时）变得更多了。我们还发现，服务器通常会发送超过200KB的压缩JSON数据作为初始开销，因此我们必须在比赛开始前约20分钟内做这件事情。

在比赛当天晚上，现场有超过10 000个用户连接至一个基站，总共有50 000~80 000个用户，造成了移动数据网络的阻塞。

虽然不能做任何事情改善连接，但我们使用了一些技巧来改善体验。开始时，我们向设备发送了推送通知。在开始前几个小时发送出去的第一个推送通知用于询问用户是否将要去比赛场。并非所有用户都回应了，但相当多的用户回应了（我们采用了游戏化来激励）。这不仅提供了估算流量的数据，更重要的是明确了哪些用户需要通知。

第二个推送通知只发送给了表示将要前往比赛场的用户。这个推送通知是在比赛的前20分钟分批发送出去的。如果球场有1000个用户，其中的100个用户会在初始的两分钟收到通知，下100个用户会在接下来的2分钟收到通知，并依次类推。

通知将唤醒应用，通过使用地理位置来决定是否获取数据。现在不再是1000个人同时连接，连接将在以100人为一组的用户组中同时进行。

显而易见，你不能指望每个用户立即打开应用，但一个推送通知将有助于唤醒应用，并让它同步数据。与应用进一步交互时会变得更加顺畅。

追踪内存泄露，提升 Android 应用体验

作者 /Doug Sillars

是 AT&T 开发者计划中的性能推广领导者。他帮助了成千上万的移动开发人员将性能的最佳实践应用到 App 上。他开发的工具和总结的最佳实践，帮助开发人员使 App 运行得更快，同时使用了更少的数据和电量。他和妻子生活在华盛顿州的一个小岛上，并在家教育三个孩子。

为了诊断 App 在哪里出现了内存泄露，需要分析 App 内存中的所有文件。如果能找出需要释放的文件或者内存中重复的文件，那么就可以在代码中解决这些问题。这样就确保了对象能被正确地释放，或者说确保了内存中文件的复用（而不是内存中存储多个重复的实例）。

为了解析 App 内存中的文件，需要在电脑中保存一个内存堆转储。在监控器（装着一半绿色 Android 液体的圆柱体）中紧邻着 Heap Dump 的图标与之类似，但有一个向下的红色箭头。该图标可以为电脑保存堆转储信息以便进行进一步的解析。

> 已保存的堆转储是 Android 特有的格式文件。要用其他工具打开文件，必须要先进行文件转换。转换工具 hprof-conv 存储于 Android SDK 工具目录下的：
>
> ```
> hprof-conv _<existing_filename> <converted_filename>_
> ```
>
> 如果你是从 Android Studio 的 DDMS 中收集的堆转储，那么就不需要专门转换了，因为在 Android Studio 的 DDMS 下它是自动转换运行的。

当创建堆转储的时候，试着复现严重的内存问题。如果可以控制 App 的大小，或者模仿记录的任何行为，内存数据将会以 hprof 文件格式转

存。找出泄露可能会非常棘手，而且需要一直盯着工具，所以说泄露越大，反而越容易被找到。

为了解析堆转储，可以利用Eclipse的内存分析工具（MAT）。在2015年年初，Square公司发布了LeakCanary——一个使得MAT的解析更加自动化的开源库。当调试时，该工具就会记录App的内存泄露问题。首先，我们要了解如何运用MAT发现内存泄露，之后弄清LeakCanary是如何简化进程的。

MAT

Eclipse的内存分析工具（MAT）就像它的名字一样：对内存堆进行详细分析的工具。MAT是Eclipse IDE的一部分，但是如果Android的开发迁移到了Android Studio上，你可以从Eclipse.org（https://eclipse.org/mat/）上面下载一个独立的MAT App使用。

在MAT中打开hprof文件时，它的确对文件做了一些处理，并询问你是否需要一个自定义的报告。如果正在查询内存泄露，一般我会选择Leak Suspects的报告。它会显示使用内存最多的对象。一旦这些运行起来，打开的工具上方就会出现很多标签页。

MAT工具在不同的窗口上提供了大量的数据。下图展示的是主视图上的Overview的标签页。它显示的是内存主要消耗的饼图。饼图的每个区域代表一块被分配的内存，点击每个区域将会看到这块区域的详细信息。最大的内存块是灰色的，代表的是空闲内存。第二大的内存块是`Iceberg`类（包含2个字节数组的`ArrayList`），大概占用4MB的内存。

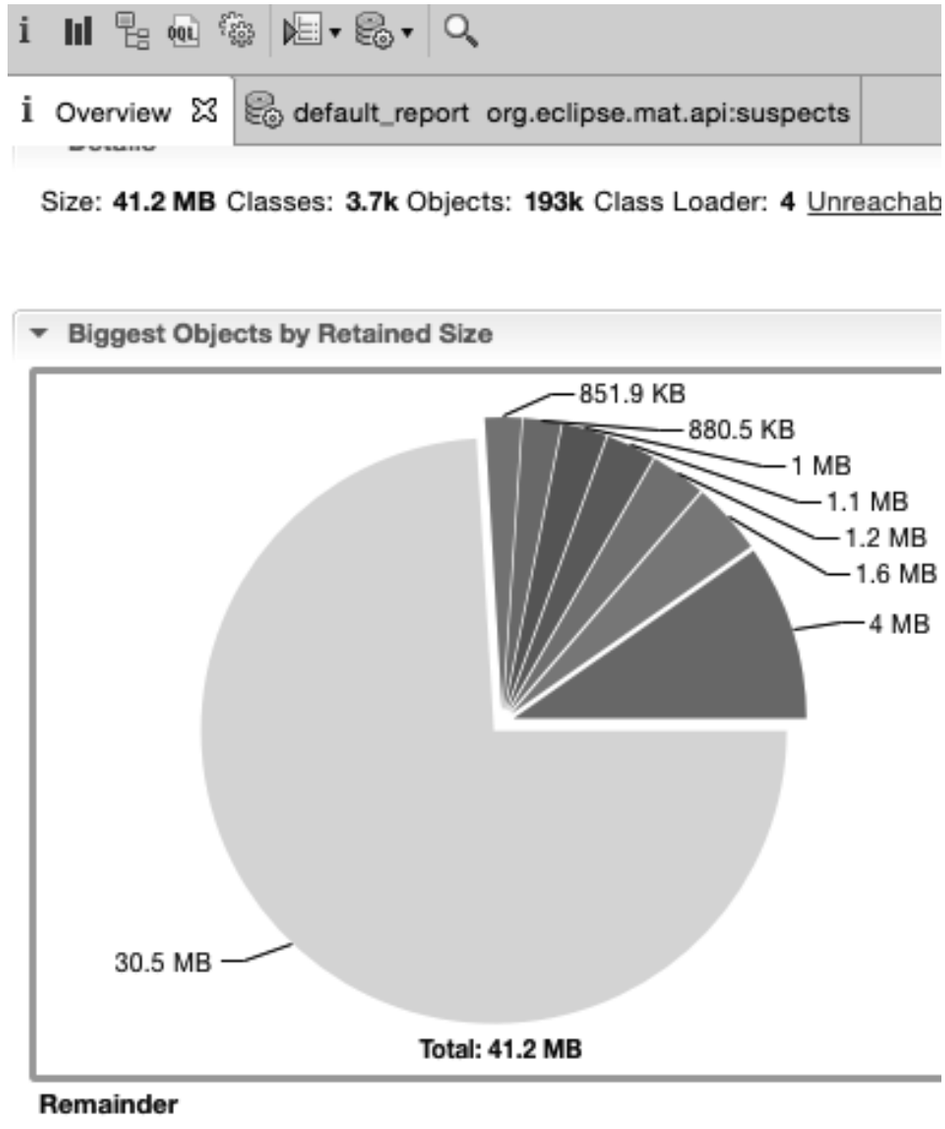


图　MAT 概况

正如在饼图中 Iceberg 类呈高亮状态，Inspector 窗口也提供了更多关于 Iceberg 类当前引用对象的信息。就像在代码中看到的一样，窗口同样展示了 iceSheet ArrayList。

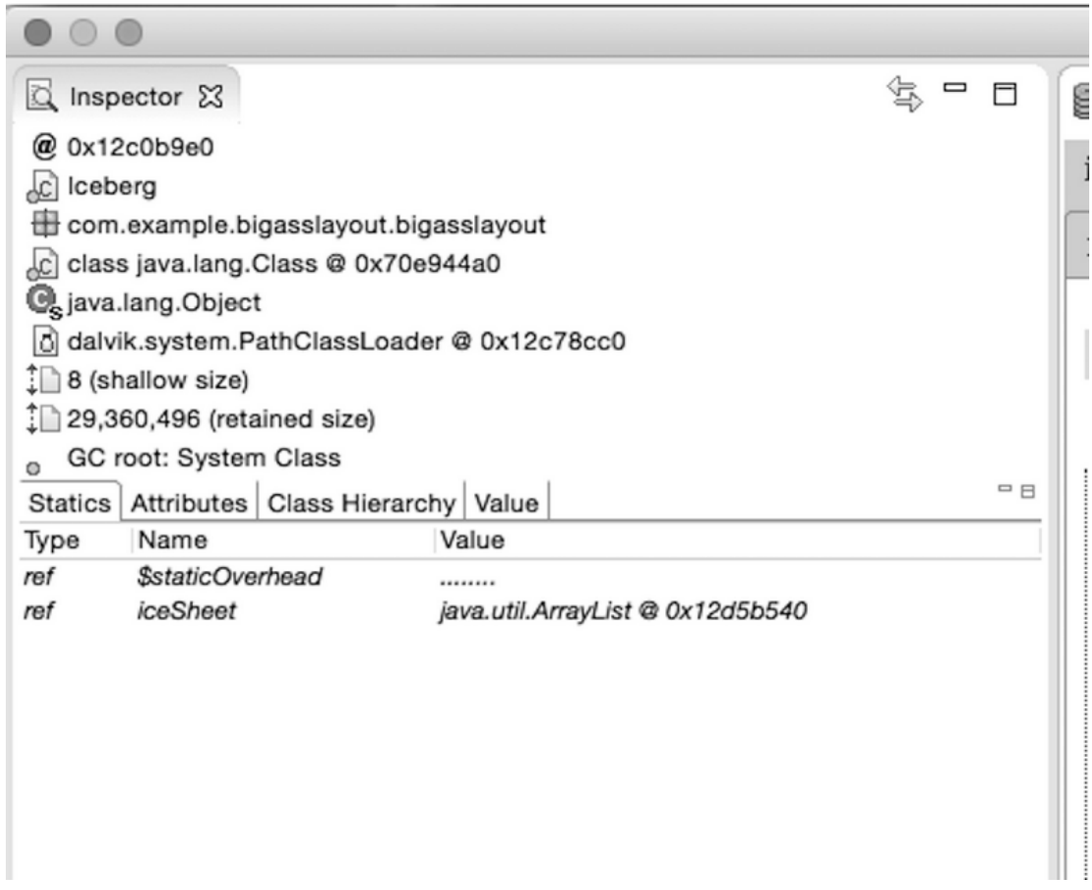


图　MAT 的 Inspector 窗口

将主视图从Overview切换到Leak Suspect报告，产生了另一个列出内存泄露猜想（基于使用的内存）的饼图。下面的图，显示了两个不同堆转储的饼图。左边的饼图是屏幕旋转两次之后的成果，有两个泄露预测，最大的部分大概占用了27MB（字节数组），Java类大概占用6.1MB（Java类）。右边的图表是屏幕旋转多次之后的结果，字节数组占用的内存大概是27MB，但是Java类的内存分配激增到了36MB。如果还不知道内存泄露的位置，这看起来是一个很好的发现机会。

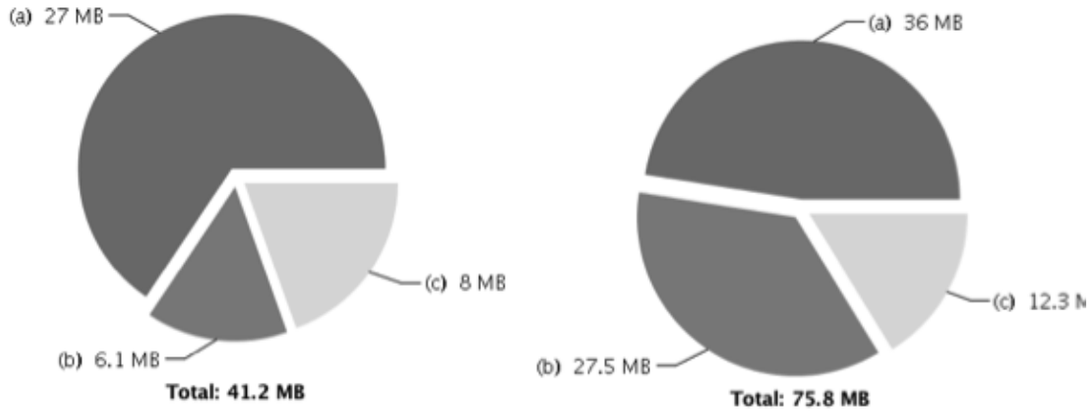


图　两个内存堆转储的 MAT 泄露猜想图

饼图的下方是一个描述了所有可疑信息的黄框。在这个图中，我们将会根据第二次追踪（多次屏幕旋转）继续分析。

Problem Suspect 1

The class **"com.example.bigasslayout.bigasslayout.Iceberg"**, loaded by **"dalvik.system.PathClassLoader @ 0x12c78cc0"**, occupies **37,749,168 (47.51%)** bytes. The memory is accumulated in one instance of **"java.lang.Object[]"** loaded by **"<system class loader>"**.

Keywords
dalvik.system.PathClassLoader @ 0x12c78cc0
java.lang.Object[]
com.example.bigasslayout.bigasslayout.Iceberg

Details »

图　MAT 泄露猜想 1

猜想 1 是 Iceberg 类，在一个 Java 对象中使用了 37MB（全部内存的 47%）。点击详细信息的链接可以更进一步地研究猜想的内容。

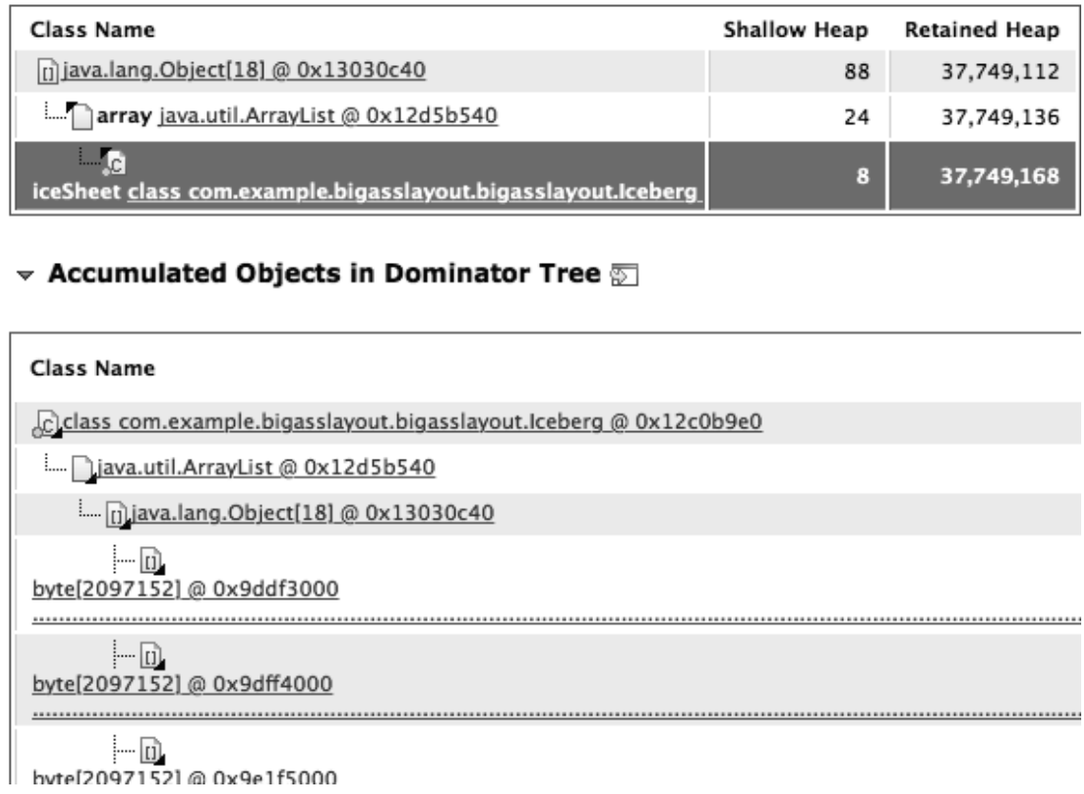


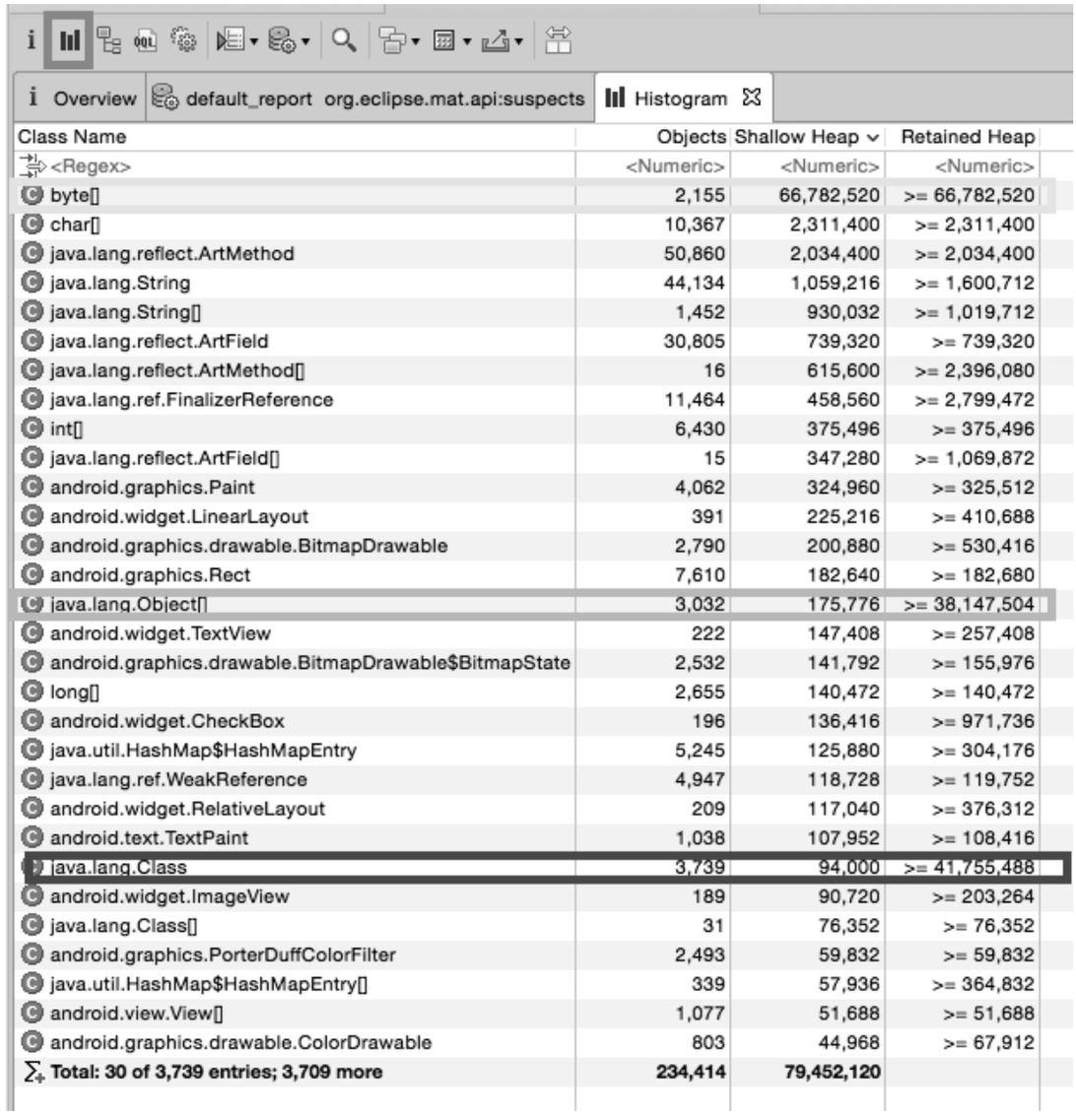
图　MAT 的泄露视图

在这种情况下，泄露猜测报告确定了问题所在。在The Shortest Path to Accumulation Point（在内存中对象引用的路径都保持如此）的视图直指`ArrayList iceSheet`。当然，在这个例子中，路径并不复杂，但它确实有作用。

当然也有一些巧妙的内存信息：`iceSheet`有一个8B的浅堆，同时有一个37MB的保留堆。浅堆是对象所用的内存，而保留堆是对象和所有对象引用对象的内存（在上面的例子中，是18个2.09MB字节的数组）。就像是树根在土壤中紧紧地抓住了树干部分，依旧在内存中的对象抓住了它们在内存中引用的所有其他对象。这很明显就是内存泄露。

很少有这么简单的例子。如果泄露不是非常明显，则需要更进一步的分析。让我们一起看看MAT中可以帮助隔离内存泄露的其他选项。

按下那个长得像条形图表（也就是，下图中的深色方框标记的地方）的图标，一个内存分布图就会被创建。



Overview | default_report org.eclipse.mat.api:suspects | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	2,155	66,782,520	>= 66,782,520
char[]	10,367	2,311,400	>= 2,311,400
java.lang.reflect.ArtMethod	50,860	2,034,400	>= 2,034,400
java.lang.String	44,134	1,059,216	>= 1,600,712
java.lang.String[]	1,452	930,032	>= 1,019,712
java.lang.reflect.ArtField	30,805	739,320	>= 739,320
java.lang.reflect.ArtMethod[]	16	615,600	>= 2,396,080
java.lang.ref.FinalizerReference	11,464	458,560	>= 2,799,472
int[]	6,430	375,496	>= 375,496
java.lang.reflect.ArtField[]	15	347,280	>= 1,069,872
android.graphics.Paint	4,062	324,960	>= 325,512
android.widget.LinearLayout	391	225,216	>= 410,688
android.graphics.drawable.BitmapDrawable	2,790	200,880	>= 530,416
android.graphics.Rect	7,610	182,640	>= 182,680
java.lang.Object[]	3,032	175,776	>= 38,147,504
android.widget.TextView	222	147,408	>= 257,408
android.graphics.drawable.BitmapDrawable$BitmapState	2,532	141,792	>= 155,976
long[]	2,655	140,472	>= 140,472
android.widget.CheckBox	196	136,416	>= 971,736
java.util.HashMap$HashMapEntry	5,245	125,880	>= 304,176
java.lang.ref.WeakReference	4,947	118,728	>= 119,752
android.widget.RelativeLayout	209	117,040	>= 376,312
android.text.TextPaint	1,038	107,952	>= 108,416
java.lang.Class	3,739	94,000	>= 41,755,488
android.widget.ImageView	189	90,720	>= 203,264
java.lang.Class[]	31	76,352	>= 76,352
android.graphics.PorterDuffColorFilter	2,493	59,832	>= 59,832
java.util.HashMap$HashMapEntry[]	339	57,936	>= 364,832
android.view.View[]	1,077	51,688	>= 51,688
android.graphics.drawable.ColorDrawable	803	44,968	>= 67,912
Total: 30 of 3,739 entries; 3,709 more	**234,414**	**79,452,120**	

图　MAT 分布图

这篇报告根据类分析了内存使用（再次以浅堆和保留堆进行区分）。在图“MAT分布图”中，有几个线索可以用于分析内存泄露。

- byte[]（第一个方框部分）包括了所有的图片（以及在iceSheet数组列表中的所有项）。66MB比预期的要多，这是因为在图“两个内存堆转储的MAT泄露猜想”中，只有27MB字节数组作为图片。
- java.lang.Object[]（第二个方框部分）有一个很小的浅堆，但是有大于38MB的保留堆。
- java.lang.Class（第三个方框部分）有一个类似的小浅堆，但是却有一个更大的保留堆。

这些线索都表示了小文件，但这些小文件却又对其他对象有庞大的引用。所以我们应该更进一步研究这些类。

为了更仔细地检查byte[]的对象列表，右击这一行，并且选择List Objects，然后选择“With Incoming references”。这样将会产生一个根据保留堆排序的新表格。

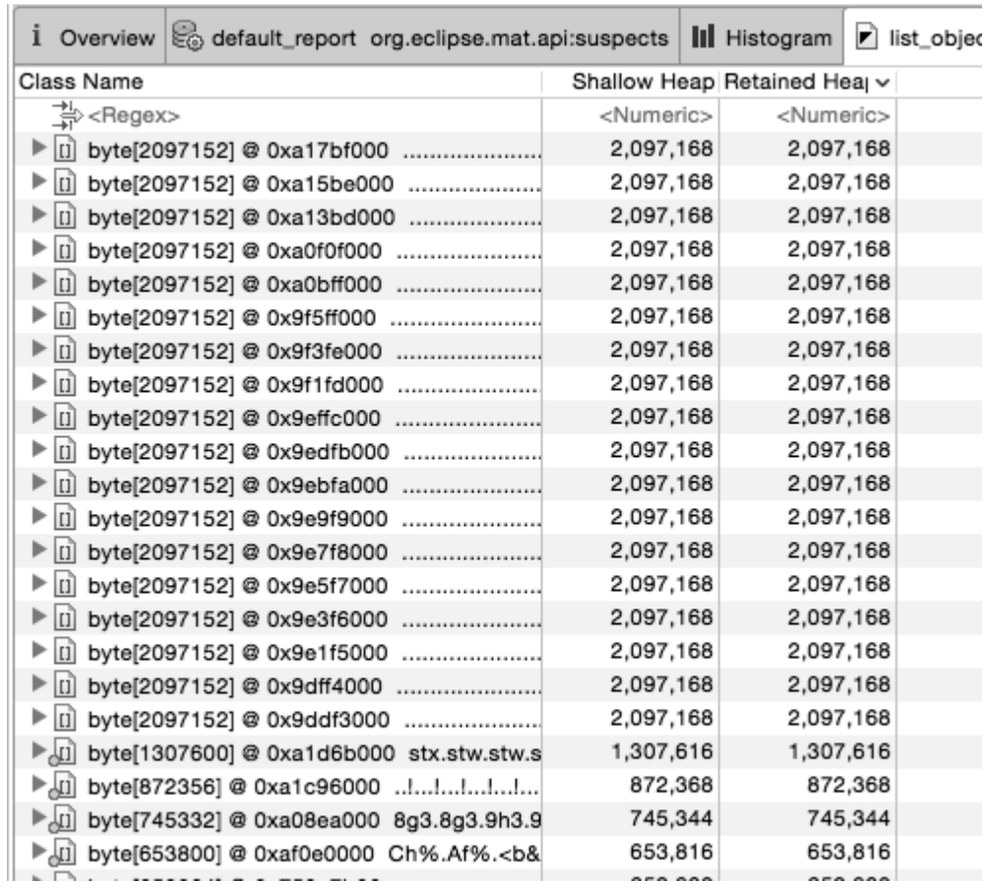


图　MAT 列出的 byte[] 的对象

在列表的顶部，可以看到由屏幕旋转得到的 18 个 2MB 的数组。如果想在垃圾回收的时候找到阻塞的根对象，对一个对象右击，并且选择“Path to GC Roots”→“excluding weak references”（在垃圾回收的过程中，弱引用不会阻塞对象）。这将会打开一个新的窗口，如下图所示。

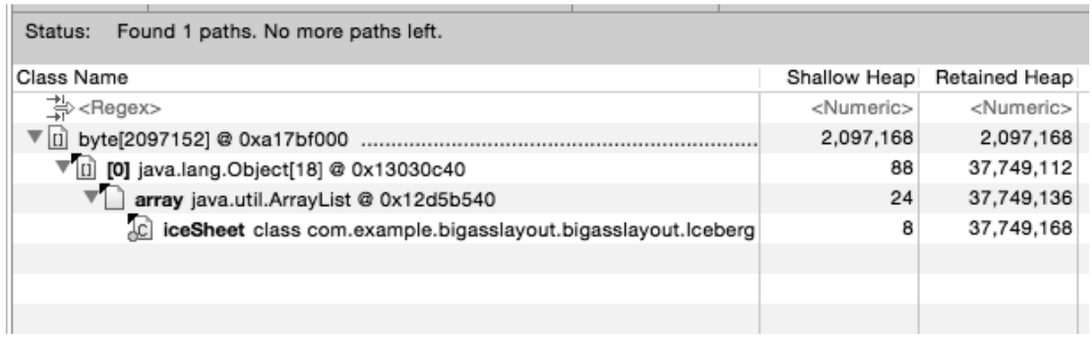


图　垃圾回收根部 2 字节数组

垃圾回收根部路径再次确认了 iceSheet 就是内存泄露的罪魁祸首。上图选择研究第一个字节数组，报告的第二行显示它在含有 18 个元素的数组列表的第 0 个位置。最后一行又出现了 iceSheet 的名字。我们再一次找到了内存泄露的原因。hprof 文件保存在本书的 GitHub 仓库里（https://github.com/dougsillars/HighPerformanceAndroidApps）。我将追踪 iceSheet 内存泄露的 java.lang.Object 和 java.lang.classes 留作一个练习，大家可以根据与上面相同的步骤找到同样的答案。

要想学习 Android 是如何处理内存分配，并找到方法优化 App 的内存分配，使用 Eclipse MAT 是一个不错的方法。但是在版本迭代快速的时代，你可能没有时间学习和调研一种新的工具来诊断内存泄露。好在 Square 团队开源了 LeakCanary，一款使 MAT 做的大部分事情能自动化输出的测试工具。

LeakCanary

Square公司开发的LeakCanary被用来减少App遇到的内存泄露错误。他们在不同的设备上发现了相同的崩溃，然后在MAT上做了一些实质性的实验来发掘是什么引发了泄露。这种方法比较慢，而他们想要在发布给最终的用户之前找到这个内存泄露，LeakCanary就诞生了。对于内存泄露，它就像是“煤矿中的金丝雀”：在内存溢出、崩溃之前，就可以嗅出内存泄露。自从使用了LeakCanary，Square公司报告（https://corner.squareup.com/2015/05/leak-canary.html）显示，OOM崩溃下降了94%。让我们一起看一下这个工具是如何运作的。

根据Square的说明（https://github.com/square/leakcanary），启动和运行LeakCanary非常简单。我在“Is this a goat?”App上运用了它，并且放到了GitHub上面。

在build.gradle文件上添加两个依赖：

```
debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3.1'
releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1'
```

在“Is it a goat?”App的应用类中，添加如下：

```
//LeakCanary引用监视器
public static RefWatcher getRefWatcher(Context context) {
    AmiAGoat application = (AmiAGoat) context.getApplicationContext();
    return application.refWatcher;
}
private RefWatcher refWatcher;


@Override public void onCreate() {
    super.onCreate();
    //在App创建上 - 打开LeakCanary
```

```
    refWatcher = LeakCanary.install(this);
}
```

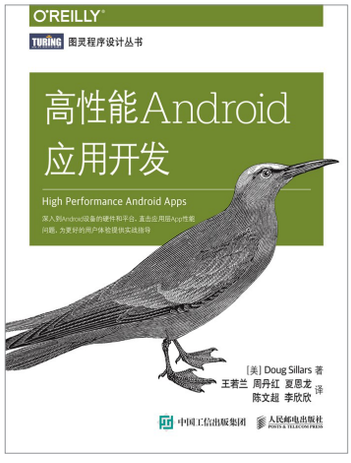
然后给CancelTheWatch类和Iceberg类添加了特定的引用watcher：

```
//LeakCanary监测变量
RefWatcher wishTheyHadAWatch = AmiAGoat.getRefWatcher(this);
wishTheyHadAWatch.watch(NoNeed);

RefWatcher icebergWatch = AmiAGoat.getRefWatcher(this);
icebergWatch.watch(theBigOne);
```

现在，当我运行“Is it a goat?”App，打开内存泄露，旋转屏幕时，事情发生了。在短暂的延迟之后，LeakCanary输出堆转储和相应的分析。写在日志上的报告如下：

```
05-25 15:43:28.283  17998-17998/<app>I/iceberg:
                    Captain, I think we might have hit something.
05-25 15:43:51.356  17998-18750/<app> D/LeakCanary: In <app>:1.0:1.
    * <app>.Iceberg has leaked:
    * GC ROOT static <app>.CancelTheWatch.iceberg
    * leaks <app>.Iceberg instance
    * Reference Key: 52614375-1531-47b1-96d7-4ec986861794
    * Device: motorola google Nexus 6 shamu
    * Android Version: 5.1 API: 22 LeakCanary: 1.3.1
    * Durations: watch=5443ms, gc=154ms, heap dump=2864ms, analysis=14302ms
    * Details:
    * Class <app>.CancelTheWatch
    |   static $staticOverhead = byte[] [id=0x12c9f9a1;length=8;size=24]
    |   static iceberg = <app>.Iceberg [id=0x1317e860]
    * Instance of <app>.Iceberg
    |   static $staticOverhead = byte[] [id=0x12c88e21;length=8;size=24]
    |   static iceSheet = java.util.ArrayList [id=0x12c267a0]
```

这个跟踪显示了关于设备的一切信息。据此追踪可知，Iceberg类有泄露，整个过程花费的时长（垃圾回收用了154毫秒，收集堆转储用了2秒，



《高性能Android应用开发》是Android性能方面的关键性指南。主要从电池、内存、CPU和网络方面讲解了电池管理、工作效率和速度这几个方面的性能优化问题，并介绍了一些有助于确定和定位性能问题所属类型的工具

分析用了14秒），以及哪些对象在类中引起了泄露。GitHub文档一步步地引导你向服务器机群上报这些泄露和堆转储。（注意，对于明显的延迟原因，内存泄露应该在调试版本就进行修复，这对内部测试具有重大的意义！）最终，这个报告将会在设备的通知栏进行提示，并以一个叫“Leaks”的新App在App列表中出现。

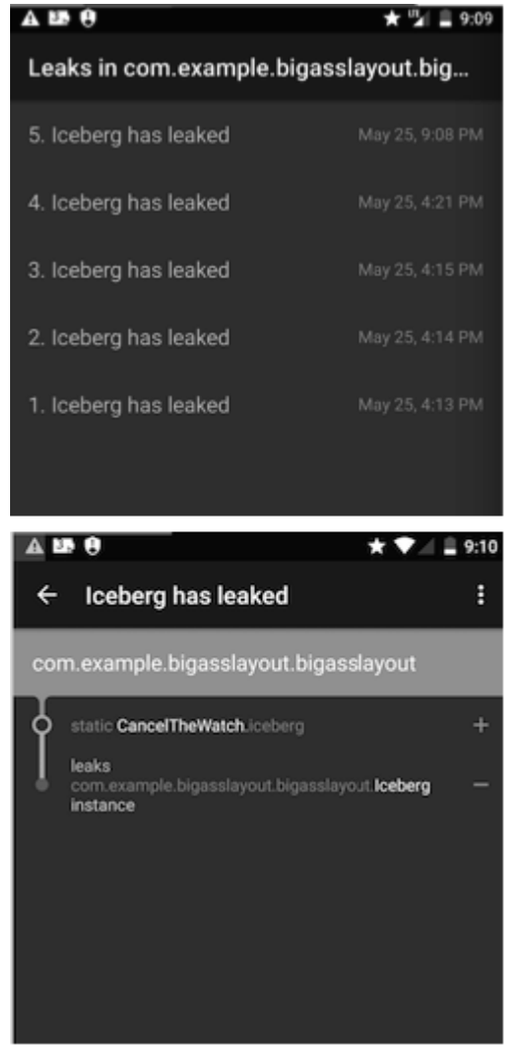


图　LeakCanary 截图：摘要（顶部）和详情（底部）

LeakCanary将存储设备最开始的7个泄露，并且会有一个菜单和其他人分享这个泄露和堆转储的信息。在内部测试中使用LeakCanary将帮助你发现MAT发现不了的内存泄露问题，快速定位App中的内存泄露，减少崩溃量，并提高App的性能。

小结

对于识别正在泄露内存的对象和类这个功能来说，MAT依旧是一个优秀的工具。理解MAT暴露的内存链接是非常重要的。然而，LeakCanary可以代替不断使用的MAT来测试内存问题。■

书　单

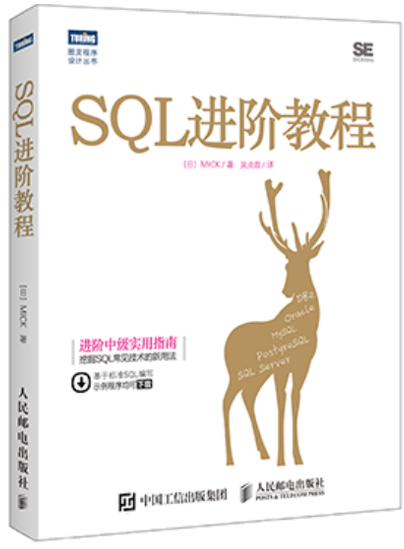
人工智能简史

作者：尼克

书号：978-7-115-47160-4

图灵社区推荐：9

全面讲述了人工智能的发展史，几乎覆盖人工智能学科的所有领域，以宏阔的视野和生动的语言，对人工智能进行了全面回顾和深度点评。



SQL进阶教程

作者：MICK

书号：978-7-115-47052-2

图灵社区推荐：10

本书是《SQL基础教程》作者MICK，为志在向中级进阶的数据库工程师编写的一本SQL技能提升指南。

陶哲轩教你学数学

作者：陶哲轩 (Terence Tao)

书号：978-7-115-46894-9

图灵社区推荐：6

国际知名数学家陶哲轩15岁时的著作，他从青少年的角度分析数学问题，主要是数学竞赛等智力谜题，用学生的语言解释思考过程，完整展现了少年陶哲轩的解题思路。启发性强，既能激发学生的数学兴趣、培养思维逻辑，又能充分展现数学的魅力。

修改软件的艺术

作者：David Scott Bernstein

书号：978-7-115-46776-8

图灵社区推荐：6

本书总结了9条构建易维护代码、解决遗留代码的最佳原则，是敏捷开发的具体实战指南。

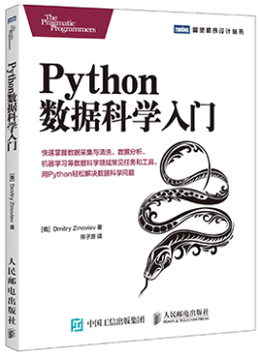
它不仅是关注如何构建更好的软件，更加注重如何构建更好的软件产业。书中囊括了作者身为专业开发者三十年所学的精华。如果你想要优化软件交付流程，但是感觉到裹足不前、无能为力，那么这本书正适合你。

5 数学女孩3：哥德尔不完备定理

作者：结城浩
书号：978-7-115-46991-5
图灵社区推荐：4

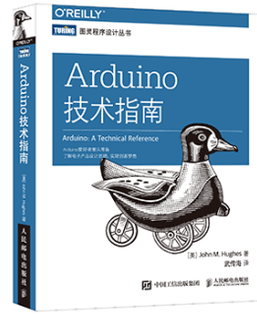
6 CSS重构：样式表性能调优

作者：Steve Lindstrom
书号：978-7-115-46978-6
图灵社区推荐：5



7 Python数据科学入门

作者：Dmitry Zinoviev
书号：978-7-115-47060-7
图灵社区推荐：4



8 Arduino技术指南

作者：John M. Hughes
书号：978-7-115-47105-5
图灵社区推荐：4

9 精通Metasploit渗透测试（第2版）

作者：Nipun Jaswal
书号：978-7-115-46940-3
图灵社区推荐：3

10 智能增长

作者：蒋凡
书号：978-7-115-47142-0
图灵社区推荐：2

用Numba实现极速数据处理

作者/Fernando Doglio

Globant公司软件架构师。过去十年一直从事Web开发工作，期间使用了大多数最前沿的技术，如PHP、Ruby on Rails、MySQL、Python、Node.js、AngularJS、REST API等。

Fernando喜欢钻研新事物，他的GitHub账户每个月也会因此获得回购。他还是开源拥护者，并通过网站lookingforpullrequests.com来获得人们的支持。Fernando另著有Pro REST API Development with Node.js。他的Twitter账号是@deleteman123。

数据处理（number crunching）是编程世界的一个主题。但是，由于Python经常用于解决科学研究和数据科学问题，所以数据处理成了Python世界的主流课题。

接下来，我会介绍一种方法Numba，它可以帮助我们写出快速高效的代码以解决科学计算问题。下面，我们从安装开始讲起，然后通过一些示例代码体现方法的优势。

Numba

Numba（http://numba.pydata.org/）是一个模块，让你能够（通过装饰器）控制Python解释器把函数转变成机器码。因此，Numba实现了与C和Cython同样的性能，但是不需要用新的解释器或者写C代码。

这个模块可以按需生成优化的机器码，甚至可以编译成CPU或GPU可执行代码。

下面的代码是官方网站上的示例，可以显示模块的使用方法。我们将在后面详细介绍：

```
from numba import jit
from numpy import arange

# jit装饰器告诉Numba编译函数
# 当函数被调用时，Numba会把参数类型引入
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```

虽然Numba看起来好像非常给力，但是它只是针对数组操作进行优化。它非常适合配合NumPy使用。因此，并非每个函数都可以用Numba优化，滥用Numba甚至会损害性能。

例如，让我们看一个类似的例子，不用Numba也可以完成类似的任务：

```
from numba import jit
from numpy import arange

# jit装饰器告诉Numba编译函数
# 当函数被调用时，Numba会把参数类型引入
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
```

```
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```

前面的代码使用和不使用 @jit 行的效果如下。

- 使用 @jit 行：0.3 秒
- 不使用 @jit 行：0.1 秒

安装

安装 Numba 有两种方法：可以用 Anaconda 出品的 conda 包管理器，也可以复制 GitHub 项目源代码进行编译。

如果你打算用 conda 方法安装，需要先安装 miniconda 命令行工具（可以从 http://conda.pydata.org/miniconda.html 下载）。安装之后，输入下面的命令：

```
$ conda install numba
```

命令输出结果如下图所示。所有要安装、升级的包都会显示出来，像 numpy 和 llvmlite 都是与 Numba 有直接依赖的包。

```
fernando@dune:~$ conda install numba
Fetching package metadata: ....
Solving package specifications: .
Package plan for installation in environment /home/fernando/miniconda:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    enum34-1.0.4               |           py27_0          48 KB
    funcsigs-0.4               |           py27_0          19 KB
    llvmlite-0.4.0             |           py27_0         7.3 MB
    numpy-1.9.2                |           py27_0         7.8 MB
    numba-0.18.2               |       np19py27_1         1.1 MB
    requests-2.7.0             |           py27_0         594 KB
    setuptools-16.0            |           py27_0         341 KB
    conda-3.12.0               |           py27_0         167 KB
    pip-6.1.1                  |           py27_0         1.4 MB
    ------------------------------------------------------------
                                           Total:        18.7 MB

The following NEW packages will be INSTALLED:

    enum34:     1.0.4-py27_0
    funcsigs:   0.4-py27_0
    llvmlite:   0.4.0-py27_0
    numba:      0.18.2-np19py27_1
    numpy:      1.9.2-py27_0
    pip:        6.1.1-py27_0
    setuptools: 16.0-py27_0

The following packages will be UPDATED:

    conda:      3.10.1-py27_0 --> 3.12.0-py27_0
    requests:   2.6.0-py27_0  --> 2.7.0-py27_0

Proceed ([y]/n)?
```

另外，如果想用源代码安装，你需要先用下面的命令复制源代码：

```
$ git clone git://github.com/numba/numba.git
```

当然 numpy 和 llvmlite 包也是需要提前安装好的。都准备好之后，用下面的命令进行安装：

```
$ python setup.py build_ext –inplace
```

需要注意的是，即使没有安装依赖包，上面的命令也可以成功运行。但是如果你没有安装依赖，Numba是没法儿用的。

要检查Numba是否可以正常使用，可以在Python的REPL里输入下面的命令：

```
>>> import numba
>>> numba.__version__
'0.18.2'
```

使用Numba

现在Numba已经安装好了，让我们看看如何使用它。这个模块提供的主要功能如下：

- 即时代码生成（On-the-fly code generation）
- CPU和GPU原生代码生成
- 与具有NumPy依赖的Python科学计算软件配合使用

1. Numba代码生成

Numba代码生成的主要方式是使用`@jit`装饰器。加上它就表示要用Numba的JIT编译器对函数进行优化。

让我们看看如何用`@jit`装饰器进行优化。

使用这个装饰器的方式有几种。默认的，也是官方推荐的方法，之前也已经介绍过：

延迟编译（Lazy compilation）

在下面的代码中，当函数被调用时，Numba将生成优化代码。它将引用属性类型和函数的返回类型：

```
from numba import jit

@jit
def sum2(a,b):
    return a + b
```

如果你用同样的函数调用其他类型，会生成并优化不同的代码路径。

(1) 及时编译

另一方面，如果你知道函数的接收类型（返回类型也可以），可以把这些类型传到`@jit`装饰器。之后，只有这种特殊情况会被优化。

下面代码中增加的部分会被传递到函数的签名里：

```
from numba import jit, int32

@jit(int32(int32, int32))
def sum2(a,b):
    return a + b
```

用于指定函数签名的常用类型如下。

- void：函数返回值类型，表示不返回任何结果。
- intp和uintp：指针大小的整数，分别表示签名和无签名类型。
- intc和uintc：相当于C语言的整型和无符号整型。
- int8、int16、int32和int64：固定宽度整型（无符号整型前面加u，比如uint8）。
- float32和float64：单精度和双精度浮点数类型。
- complex64和complex128：单精度和双精度复数类型。
- 数组可以用任何带索引的数值类型表示，比如float32[:]就是一维浮点数数组类型，int32[:,:]就是二维整型数组。

(2) 其他配置

除了及时编译，还有两个编译选项可以添加到@jit装饰器上。这两个选项将帮助我们完成Numba优化。选项具体描述如下。

(a) 没有GIL

无论何时，只要我们的代码用原始类型优化（不是用Python类型），GIL就不再必要了。

有一种方法可以禁止GIL。我们可以把nogil=True属性传到装饰器。这样我们就可以用多线程运行Python代码（或者说是Numba代码）了。

也就是说，只要不再受GIL的限制，你就可以处理多线程系统的常见问题了（一致性、数据同步、竞态条件等）。

(b) 无 Python 模式

这个选项可以让我们设置 Numba 的编译模式。默认设置时，它将在模式之间跳转。Numba 将针对需要优化的代码自动设置对应的优化模式。

一共有两种模式。一种是 object 模式。它产生的代码可以处理所有 Python 对象，并用 C API 完成 Python 对象上的操作。另一种是 nopython 模式，它可以不调用 C API 而生成更高效的代码。唯一的问题是，只有一部分函数和方法可以使用。

如果 Numba 不利用循环 JIT（loop-jitting）方法，object 模式就不会产生更快的代码（就是说循环可以被提取然后编译成 nopython 模式）。

我们可以用 Numba 把代码强制转换成 nopython 模式，如果转换失败就会产生错误。可以用下面的代码实现：

```
@jit(nopython=True)
def add2(a, b):
    return a + b
```

nopython 模式的问题在于它有一些限制，除了这种模式支持的 Python 子集范围有限之外，还有：

- 函数里表示数值的所有原生类型都可以被引用
- 函数里不可以分配新内存

另外，由于使用循环 JIT 方式，被优化的循环内部不能产生返回状态。否则，这种情况不适合优化。

下面，让我们用一段示例代码来演示优化的过程：

```
def sum(x, y):
    array = np.arange(x * y).reshape(x, y)
    sum = 0
    for i in range(x):
        for j in range(y):
            sum += array[i, j]
    return sum
```

上面的代码取自 Numba 网站。这个函数适合循环 JIT，也叫循环切换（loop-lifting）。为了让代码如预期运行，我们用 Python REPL 模式：

```
Python 2.7.9 |Continuum Analytics, Inc.| (default, Apr 14 2015, 12:54:25)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> from numba import jit
>>> import numpy as np
>>> @jit
... def sum_auto_jitting(x, y):
...     array = np.arange(x * y).reshape(x, y)
...     sum = 0
...     for i in range(x):
...         for j in range(y):
...             sum += array[i, j]
...     return sum
...
>>>
>>> sum_auto_jitting(2,65)
8385
>>> sum_auto_jitting.inspect_types()
```

另外，我们还直接使用了函数的 `inspect_types` 方法。这样做的好处是可以看到函数的源代码。这样在匹配 Numba 生成的机器码时，可以和源代码进行对照。

上面这个方法的输出结果可以帮助我们理解 Numba 优化背后的含义。

《Python性能分析与优化》对于Python程序员来说，仅仅知道如何写代码是不够的，还要能够充分利用关键代码的处理能力。本书将讨论如何对Python代码进行性能分析，找出性能瓶颈，并通过不同的性能优化技术消除瓶颈。

更具体地说，可以看到具体的引用类型，是否进行了自动优化，以及每行Python代码被翻译成多少行代码。

要理解`inspect_types`方法从代码中获得的输出结果的含义，需要注意看每一行源代码的编译过程都是由单独的行号开始的。后面跟着的是这一行生成的汇编指令，最后你可以看到不加注释的Python源代码。

注意看`LiftedLoop`行。在这一行你会看到Numba的优化代码。还需要注意在许多行最后引用的Numba类型。无论何时你看到一个`pyobject`类型，都不是原生类型，而是普通Python类型的封装版。

2. 在GPU上运行代码

前面已经提过，Numba代码可以运行在CPU和GPU上。其实，在GPU上运行并行程序可以进一步提升性能，比CPU上运行更快。

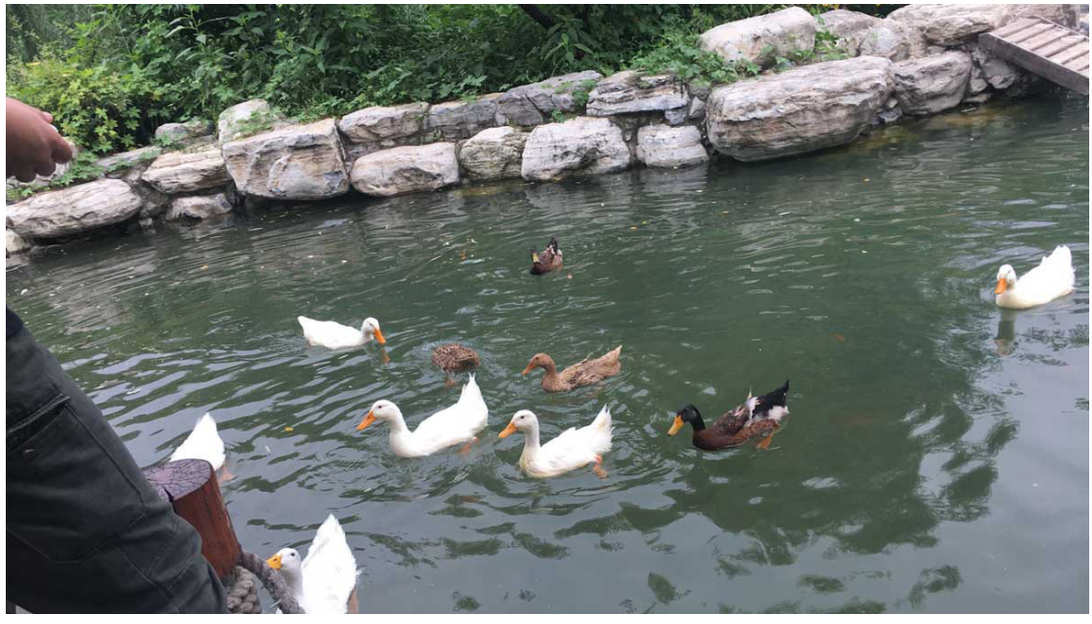
更具体地说就是，Numba支持CUDA编程（http://www.nvidia.com/object/cudahomenew.html），按照CUDA模式的规则把一部分Python代码翻译成CUDA核心与设备支持的语言形式。

CUDA是Nvidia开发的并行计算平台和编程模式。它可以利用GPU获得更大的速度提升。

因为GPU编程是个较大的主题，可以写一整本书，所以这里不再介绍更多细节。我们只说明Numba具有这种能力，通过装饰器`@cuda.jit`就可以实现。关于这个主题的具体文档，请参考http://numba.pydata.org/numba-doc/0.18.2/cuda/index.html的内容。■

木头人

作者/雨帆



“唉，你还真是一个不安分的家伙啊！”电话那头的老妈无可奈何地宠溺道。

此时还是八月初，火辣辣的太阳炙烤着大地，北京西小口软件园东边花坛里的各色花朵无精打采地垂下了平日高傲的头。池塘里的一群鸭子正在争抢着午间员工投喂的馒头，这给喘不过气的大热天增添了一抹生机。

此时的我，才和老妈通完电话。在短短的 30 分钟里，我做出了极为重要的决定——离开北京，回南京工作。

为何离职？为何离开？

很多人问过我，我也给出了不同的回答。对同事，我的说法是回家结婚生子。对朋友，我说北京空气太差，回南京会更健康。对家人，我说新房快交了，想回来准备装修……

从 2014 年 7 月大学毕业到决定离开北京，已经整整“北漂”了 3 年。这期间，北京于我，从来就不是能够长久生活下去的地方。当初毕业的时候，同学们要么出国，要么保研、考研、进公务体系，要么就回家乡发展。只有我一个人，从为数不多的 Offer 中挑了一份看起来还行的工作，来到了北京。记得还有同学笑道，“雨帆，你要住地下室了。”我还是义无反顾地放弃了父母早就准备好的“前程”，选择从来没有去过的“最北边”——北京。

在我看来，无论是吃穿住都不愁的厦门，十分稳定的济南校企，还是那各种意义上都十分繁华的“大上海”，都不是我向往的地方。我的骨子里向往冒险，向往诗意和远方。

我渴望离开父母，逃离烦闷无趣的校园，追求自我拼搏的生活。所以，这一次，我没有听任何人的劝诫。一个包，一把伞，一个箱子，一个人，坐上最便宜的绿皮火车，来到了北京。

你见过北京早上 5 点钟的太阳么？我见过。

那个时候的我，为了省钱，住在离公司 27 公里外的上地，公司却远在南三环的万通中心。每天天还没亮，我就得早早爬起来和 60 多个人争抢仅有的 2 个水龙头洗漱，然后在西二旗那一眼望不到边的地铁大军中蹒跚而行。下班到家，也已经是晚上 9 点。

在 A 公司工作的一年半里，我基本在公司睡觉。因为这样可以省下一天 5 元的地铁费用，同时还有公司最高 25 元的晚餐补助。这样，每天基本就没有开销。晚上加班的时间，就是看当天同事提交的代码，学习别人的设计文档。到了夜里 11 点，关掉所有的灯，在前台旁边铺好的地

铺上睡到第二天的 8 点，因为保洁阿姨这时候会来打扫卫生。简单洗漱后，开始新的一天工作。

那个时候的我，技术基础比别人差得太多，什么都不会，代码也没怎么写过。对我而言，每天除了学习、看书、工作之外，就是睡觉。工位后面柜子子上摆的书越来越多，一本、两本，十本、二十本，一书架……每本书都会基本看完。

记得有一天晚上 2 点，我还在看正则表达式，公司的 CEO 从我身边走过，关心地询问，“怎么这么晚了，还在加班”。一瞬间，我突然很想哭，

在离家 2000 多公里的土地上，我竟第一次感受到了来自别人的关怀。虽然只是简单的对话，但是对于那个时候的我来说，重过千金。

在 A 公司工作的那段期间，柏前辈教会了我很多东西，他是我至今都十分佩服和“嫉妒”的前辈。柏比我早 2 年入职，在各个方面都十分优秀，完美地简直不是“人”。在公司里面，加班最狠、最多的就是他，不论是周末还是节假日，你都能在工位上找到他。晚上，还会跟我争抢监控室的小床，逼我打地铺，哼！

时间很快到了 2015 年底，我所在的部门要拆分，基础架构部基本名存实亡，“老大”包也有意出去单干。当时，我是恐惧的！我那微乎其微的技术能力，真不知道还能干什么。但是，我还是选择了跳槽。

选择去面试 B 公司，是因为我经常在图灵社区上看到它的“广告”，另外在 Software Design 和 Web DB Press 上也能看到它的招聘信息。对我而言，它是陌生又神秘的。面试的前一天，北京才下了雪，从地铁站出来，扑面而来的除了冷还是冷，在公司窝了 1 个月的我不禁打了一个寒战。因为导航的错误，我在整个园区绕了一圈才找到 B 公司。我无暇顾及这银装素裹的园区美景，大地白茫茫一片，我看不到出路，一如当时迷茫的思路。

到了目的地后，发现 B 公司真的很小，小到面试只能在前台旁边的茶几上进行。然后，我见到了职业生涯中的另外两位导师，姜老师和光老师。当时的我并不认识姜老师，只是听人力一味地夸他。姜老师面试我的时候，只简单地考察了几个尾递归，询问我会不会写 Scala 和一些简单的技术问题。然后，是光老师的二面。至今，我都记得和他第一次见面的场景，

个子不高，戴着一副粗绿框眼镜，满脸的胡子像是几天不曾洗漱。疾步走来，一屁股坐下就开始审阅我的简历。那一瞬间，我突然有了一种被人看透的恐慌。

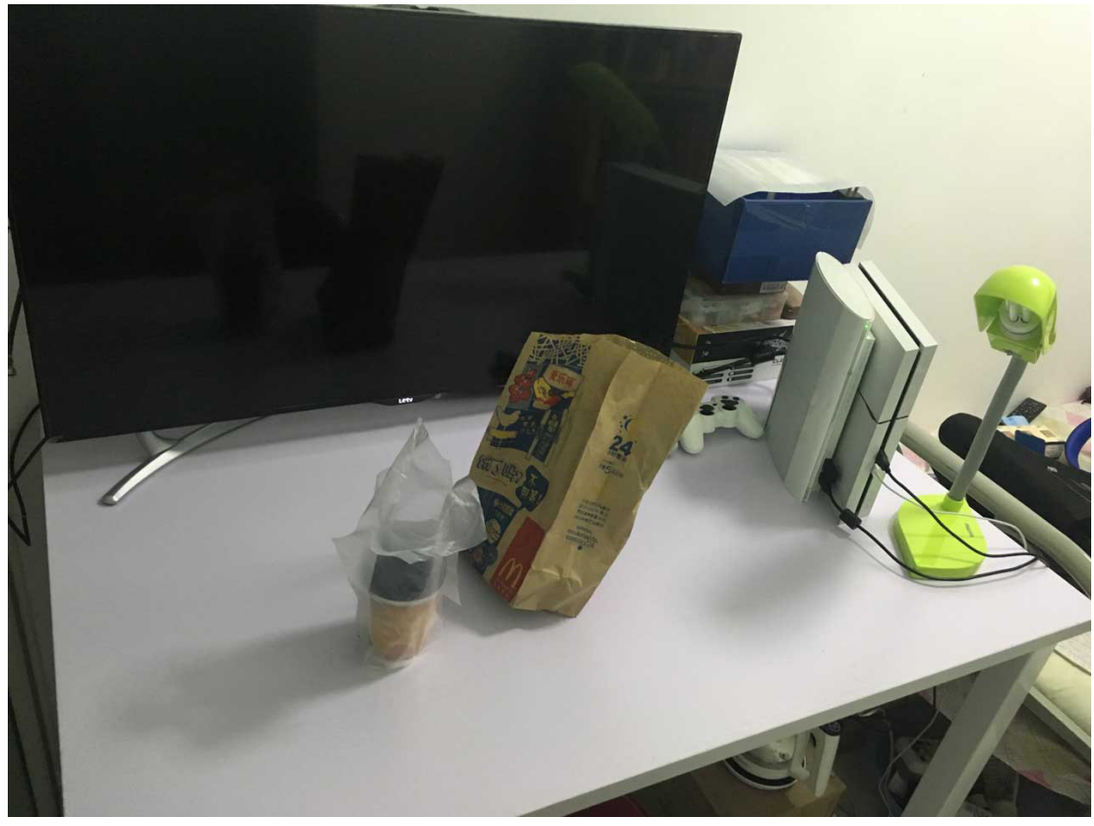
很快，我就入职了。后来才知道B的人员走了很多，我入职的部门有一大批去了杭州。招进来几个，也很快就走了。最后，连面试我的人力也离职了，这让我一度十分恐慌，心想是不是掉坑了。工作中接触到的用户系统写得十分可怕，我甚至对公司的技术研发产生了质疑。虽然每天都是在学习姜老师安排的新东西，心还是揣着的。直到友国入职，我才算是稍微安下了心。

友国入职B公司的原因很简单，他在这里可以写 Scala。在他眼里 Scala 天下第一，秒杀一切。一入职，他就和健开始一起写 Mock Agent，将那个我原本就看不大懂的项目，写得更加晦涩。

B公司既小又穷，公司的开支大头，就是人员的工资。即便这样，B公司还是愿意花费很多的资金让珊姐姐去参加国外的技术大会，将最新的 APM 技术带回来分享给大家。因为硬件设施的不足，CEO 都必须站到凳子上和大家讲话，才能确保每个人都能听到。

有一句话叫作“一群人的浪漫”，我在这里才真正体会到了其中的种种。姜老师扩宽了我的技术视野，光老师那强悍的编码设计能力让我折服，友国天天喊着的“Scala 大法好”的精神令我振奋，湘老师教会了我如何写测试、用好 IntelliJ IDEA。这些都是我职业生涯中难得的财富。

那么，为什么要离开呢？



这是我在延静里租的一间卧室，它是餐厅改造的。里面放了一张书桌外加一张床，仅此而已。为了好看，我没有拍床。

因为，北京从来没有给我安定的感觉。无论你是在繁华的三环内，还是充斥着互联网公司的海淀，抑或是百度总部的西二旗，等你忙完一天拖着疲惫的身体回到“家”，北京的恶意就会迎面袭来，让你无所适从。洗到一半的热水器会坏掉，忘记钥匙的你会被锁在门外一晚，高达 2 小时的通勤路程会腐蚀掉你的耐心……种种诸如此类，让我在北京找不到任何归属感。我只能在工作中麻木自己，让自己像木头人一样，不去想，不去看。

电话里，老妈常说的话就是，你又买了 XXX，到时候你是要搬家的，还不是要扔掉！

2016 年的 10 月份，纵然房价很高，我还是做出了人生中的重大决定——买房！交完订金的那晚，我和父母绕着即将建好的小区走了很久很久。谈不上高兴或者激动，只是心突然安定了下来，就好象是茫茫大海中有了一座灯塔，你能看到方向，并且知道该往哪里走。

今年 8 月底，在即将交房的前夕，我回到了南京开始了另一段旅程。

昨天，看了最新一期的暴走漫画，王尼玛对那个因为父母不让考研的小孩说，你要学会自己思考，自己去争取想要的东西来向父母证明他们错了。我想，我就是他口中的那个不安定又矛盾的小孩吧：一方面，想趁着年轻出去闯闯；一方面，又对各种琐碎觉得委屈要找人求安慰，容不得半粒沙子。

很多事情，也只有经历过，吃亏过，才知道对错，不是么？

谨以此文纪念我那已逝的三年岁月。■

原文链接：https://yufan.me/watashitachi-ha-mokuzaidesu/

投稿须知

码农 the code maker

《码农》(Code Maker) 是国内计算机图书高端品牌之一的“图灵教育”所推出的免费电子期刊。本着“分享知识、提升技术、塑造人生”的思想，向国内广大计算机技术从业人员及技术爱好者，传播国内外、世界前沿的技术知识。采用双月刊的形式，每两月推出一期主题鲜明的杂志。围绕主题内容，每期杂志分别设有相关的“人物”“鲜阅”“践行”“九卦”“书单”“妙评”等专栏内容，目的是摒弃刻板、陈旧的“正统式”陈述方式，帮助读者全方位、多角度地理解主题内容。

《码农》欢迎广大读者在这里投稿作品、交流学习。

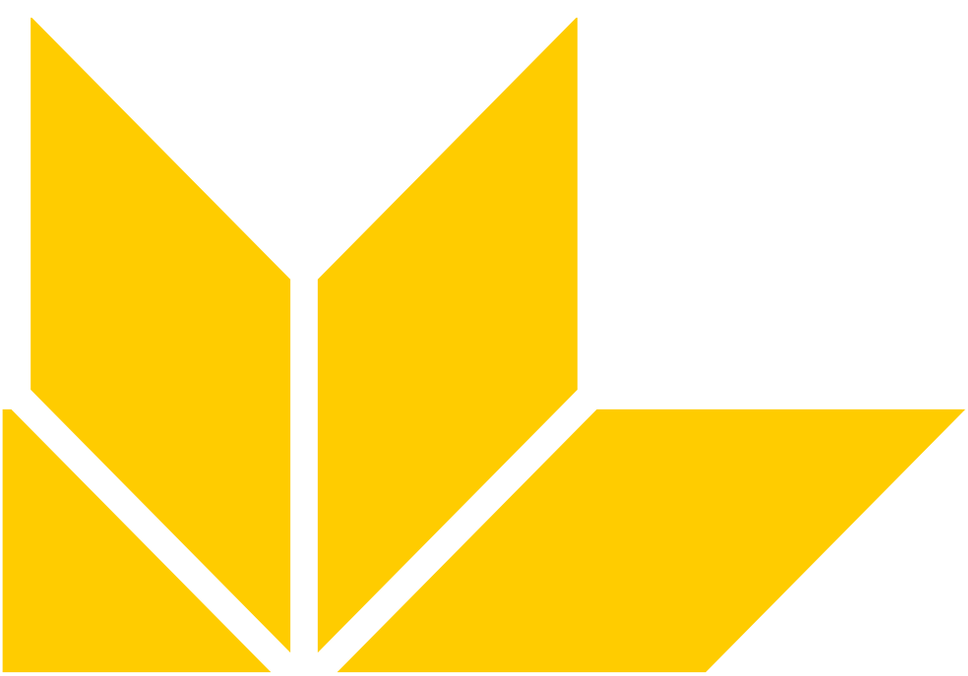
作品要求：

- 内容真实、准确、合法，并且不会侵害任何第三方的合法权益；
- 对于提交的作品在版权、邻接权、信息网络传播权、文字内容及转授权等方面与第三方发生纠纷或者引发其他法律责任的情况，《码农》不承担任何责任；
- 以电子版形式发送到邮箱 liumin@turingbook.com

对于采用的作品，我们会在杂志出版后，支付给作者一定数量的银子！银子可在图灵社区(www.ituring.cn) 上使用，兑换到“图灵教育”品牌下的纸质图书！

图灵社区 出品

出版人：武卫东
编辑：刘敏
设计：大胖



地址：北京市朝阳区北苑路13号院领地OFFICE C座603室
电话：010-51095181
微博：weibo.com/ituring
Email: ebook@turingbook.com