34 | 网站性能优化（上）

2019-11-27 四火

全栈工程师修炼指南 进入课程 >

讲述：四火

时长 21:17 大小 14.63M

你好，我是四火。

从今天开始我们进入“专题”这一章，本章的内容会涉及一些不适合单独归纳到前面任何一章的话题，比如这一讲和下一讲，我们来聊一聊网站的性能优化。

具体说，在这两讲性能专题中，第一讲我将介绍性能优化的基本知识，包括性能优化和软件设计的关系，性能指标和关注点，以及怎样去寻找性能瓶颈，这些是我们需要优先明确、分析和思考的内容，而不是没头没脑地直接跳进问题的坑去“优化”；而第二讲则要讲真正的“优化”了，我将从产品和架构调整，后端和持久层优化，以及前端和网络层优化这样三个方面结合例子来介绍具体的性能优化技术。

性能优化与软件设计

在最开始的部分，我想明确最重要的一件事——也许你已经听说过这句话——“**好的性能是设计出来的，而不是优化出来的**”。这就像是“高质量的软件是设计出来的，而不是测试出来的”一样，强调的都是软件设计的地位。因此，在设计阶段不考虑性能，而寄希望于在上线前后通过集中式的性能优化突击来将网站性能提高到另一个层次，往往是主次颠倒的。

我来举一个实际的例子说明下这一点吧。

在我参加工作不久后，曾经参与过的一个项目，用户的所有读写操作都要入库，当时的设计是把这样的数据放到关系数据库一个用户行为的表中，以供后续的查阅。上线以后，由于用户量大，操作频繁，很快这张表就变得巨大，读写的性能瓶颈都在这张表上，而且所有用户的操作都受到了影响，因为所有的操作都要写入这张表，在这时我被卷入这个项目，希望尽快解决这个性能问题。说实话，最初设计的时候没有考虑好，而这时候要做小修小改来解决这个功能的性能问题，是比较困难的。

具体来说，如果在设计阶段，这里其实是有很多可以考虑的因素的，它们都可以帮助提前介入这个性能隐患的分析和解决：

产品设计角度，用户的行为分为读操作和写操作，我们知道写操作通常更为重要，那用户的读操作是否有那么重要，可不可以只记录写操作？这样的数据有多重要，丢失的代价有多大，数据保密性是怎样的？这决定了一致性要求和数据存储在客户端是否可能。历史数据是怎样消费的，即数据访问模型是怎样的，那个时候 NoSQL 还没有兴起，但是如果不需要复杂的关系查询，这里还是可以有很大的优化空间的。

技术实现角度，能否以某种方式保留最近的若干条记录即可？这将允许某一种方式进行定期的数据淘汰、清理和归档。归档文件的组织形式又和数据访问模型有关，因为归档文件不支持任意的关系查询，那该怎样去支持特定的业务查询模型呢？比如将较老的记录归档到以用户 ID 和时间打散的历史文件中去，目的都是保证表的大小可控。哪怕是面对最糟糕的看似没有办法的“窘境”，只要预先考虑到了，我们还有对于数据库表进行 sharding 或者 partition 这样的看似“不优雅”，但是也能解决问题的通用方法。

上面我举了几个对于该性能问题的“优化”可以考虑的方面，但相信你也看到了，与其说这些手段是“优化”，倒不如说是“设计”。当时那个项目，我们被迫加班加点，周末时间顶

着客户不满赶工出一个线上的临时处理方案来缓解问题，并最终花了一周时间重构这部分设计，才算彻底地解决了这个问题。

这个教训是深刻的，**从工程上看，整个设计、开发、测试到运维一系列的步骤里面，我们总是不得不至少在某一个步骤做到足够细致和周全，这里的细致指的是深度，而周全则意味着广度。**

最理想的情况当然是在产品设计阶段就意识到这个问题，但是这也是最难的；如果产品设计阶段做不到，那么开发实现阶段，开发人员通过深入的思考就能够意识到这个问题；如果开发阶段做不到，在测试阶段如果可以进行细致和周全的思考，也可以发现这个问题；如果测试阶段还没有做到，那就只好等着上线以后，问题发生来打脸了。当然，这里面的代价是随着工程流程向后进行而递增的。

到这里，我想你应该理解了为什么说“好的性能是设计出来的”，但是说说容易做却难。如果由于种种原因，我们还是遇到了需要在项目的中后期，来做性能优化的情况，那么这时候，我们可以遵循这样三条原则来进行：

问题定位，出现性能瓶颈，定位清楚问题的原因，是进行后面优化工作的前提；

问题解决，有时候问题比较棘手，我们可能会将解决方案划分为临时方案和长期方案，分别进行；

问题泛化，这指的是将解决一个问题泛化到更大的维度上，比如项目中还有没有类似的隐患？能不能总结经验以供未来的其他开发人员学习？这些是为了提高对未发生问题的预见性。

性能指标与关注点

在我们从产品经理手里拿到产品设计文档的时候，性能指标就应当有一定程度的明确了。性能指标基本上包含两个大类，一个是从业务角度定义的，另一个是从资源角度定义的。

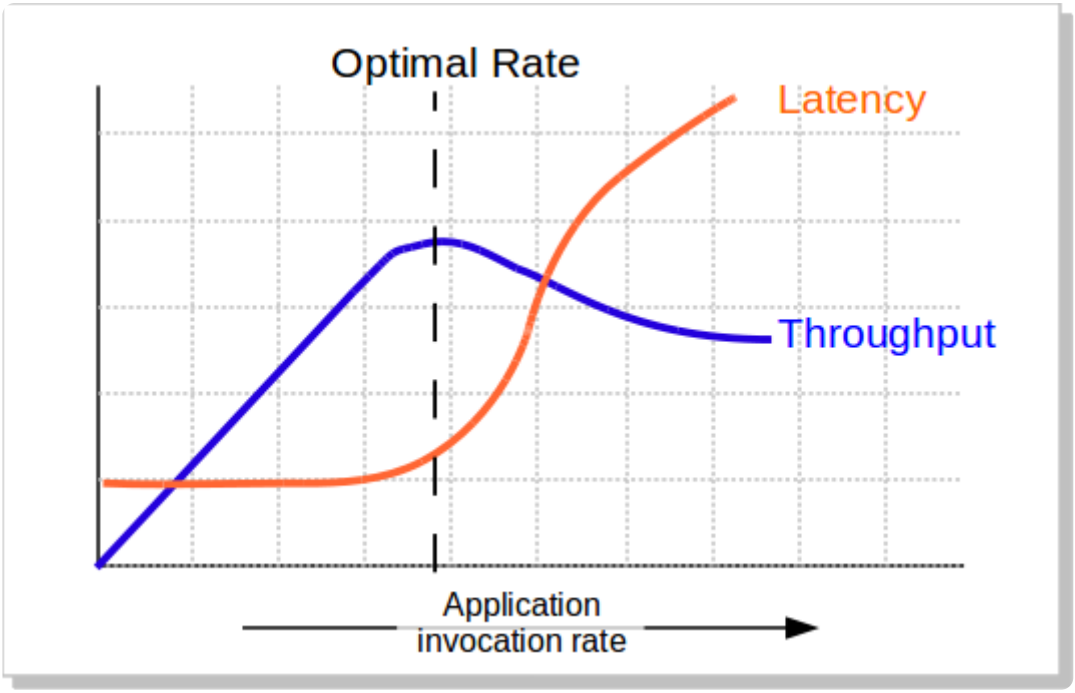
通常产品经理给的需求或设计文档中，就应当包含一些关键的业务角度定义的性能指标了，但是资源角度的性能指标，产品经理通常不怎么关心。一般说来，比如我们**最关心的性能指标，也往往就是从业务角度定义的，它可以说是“越高越好”；而资源角度，程序员有时考虑的却是“越低越好”，有时则是“达标就好”。**

1. 业务角度

你可以回想一下 [第 21 讲]，我们在那时就介绍了缓存的使用动机，就是节约开销。这和性能指标的关注点如出一辙，既包括延迟（latency）和吞吐量（throughput），这两个同时往往也是业务角度最重要的性能指标，又包括资源消耗，这也完全符合前面说的资源角度。

延迟和吞吐量，这两个指标往往是应用性能评估中最重要的两个。先来明确二者的定义：延迟，其实就是响应时间，指的是客户端发送请求以后，到收到该请求的响应，中间需要的时间消耗；而吞吐量，指的是系统处理请求的速率，反映的是单位时间内处理请求的能力。

一般说来，**延迟和吞吐量是既互斥、又相辅相成的关系。**一图胜千言，且看这个典型例子（下图来自这篇文章，这个图因很有典型意义而被到处转载）：



总体来说，随着压力的增大，同等时间内应用被调用的次数增加，我们可以发现这样几个规律：

单独观察 Latency 曲线，延迟总是非递减的。一开始延迟可以说几乎不变，到了某一个特定的值（转折点）以后，延迟突然迅速增大。

再来单独观察 Throughput 曲线，吞吐量总是一个先线性增大，后来增长减缓，到达峰值后逐渐减小的过程。

再把两者结合起来观察，延迟曲线前面提到的那个转折点，和吞吐量的最高点，通常在一个比较接近的位置上，并且**延迟的转折点往往要略先于吞吐量的最高点出现。**

一般说来，延迟会先上升，随着延迟的上升，吞吐量增大的斜率会越来越小，直到达到最高点，之后开始回落。请注意这个系统能达到的最佳吞吐量的点，这个点往往意味着延迟已经开始增大一小段时间了，因此对于一些延迟要求比较高的系统，我们是不允许它达到这个吞吐量的最高点的。

从这个图我们也可以看出，**如果我们不做流量控制，在系统压力大到一定程度的时候，系统并非只能做它“力所能及”的工作，而往往是“什么也做不成”了。**这也是一些不够健壮的系统，在压力较大的特殊业务场景，直接崩溃，对所有用户都拒绝服务的原因。

其它较常见的业务角度定义的性能指标还包括：

TPS：Transactions Per Second，每秒处理的事务数。相似的还有 QPS，Queries Per Second，每秒处理的查询数量等等。

并发用户数：同一时间有多少用户在使用系统。对于网站来说，由于 HTTP 连接的无状态性，往往会使用服务端的会话数量来定义有多少用户同时访问系统。

TP99：指的是请求中 99% 的请求能达到的性能，这里的 TP 是 Top Percentile 的缩写。比如，我们说某系统响应时间的 TP99 是 1 秒，指的就是 99% 的请求，都可以在 1 秒之内响应，至于剩下的 1% 就不一定了。

关于 TP99，为什么我们使用这种分布比例的方式，而不是使用简单的平均数来定义呢?

这是因为**性能平均数在真实的系统中往往不科学。**举例来说，在一个系统的 100 个请求中，99 个都在 1 秒左右返回了，还有一个 100 秒还不返回，那平均时间会大于 (99 x 1 + 1 x 100) / 100，大约等于 2 秒，这显然不能反映系统的真实情况。因为那一个耗时特别长的请求其实是一个异常，而非正常的请求，正常的请求平均时间就是 1 秒，而 TP99 就比较能反映真实情况了，因为 TP99 就可以达到 1 秒。

有了 TP99 这样的概念，就可以定义系统的 SLA 了，SLA 是 Service-Level Agreement，它是系统对于它的客户所承诺的可以提供的服务质量，既包括功能，也包括性能。在性能方面，SLA 的核心定义往往就是基于 Top Percentile 来进行的。比方说，一个网站的 SLA 要求它对于浏览器响应时间的 TP95 为 3 秒。

2. 资源角度

相应的，资源指标包括 CPU 占用、内存占用、磁盘 I/O、网络 I/O 等等。常规来讲，平均使用率和峰值都是我们比较关心的两个数据。一般我们可以把系统上限的一个比例，比如 60% 或者 80% 定义为一个安全阈值，一旦超过（比如 x 分钟内有 y 次检测到指标超过该阈值，这种复杂指标设置的目的在于考虑性能波动，减少假警报）就要告警。

寻找性能瓶颈

很多时候，如果我们能找到系统的性能瓶颈，其实就已经解决了一半问题。值得注意的是，性能的瓶颈并不是系统暴露在表面的问题，比如，TP99 的延迟不达标是表面问题，而不是系统瓶颈。

大致思路

在性能优化这个繁琐的过程当中，一要寻找问题，二才是解决问题，而寻找性能瓶颈就和寻找根本原因有关，也是一个很有意思的事儿。不同的人显然会有不同的看法，我在网站项目中进行性能瓶颈的定位时，常常会遵循下面几个步骤，换句话说，这个步骤来自于我的经验：

首先，我们应该对系统有一个大致的预期。举例来说，根据现有的架构和硬件，在当前的请求模型下，我们希望单台机器能够负载 100 TPS 的请求，这些是设计阶段就考虑清楚的。我看到有些项目中，这些没有在开始定义清楚，等系统大致做好了，再来定义性能目标，这时候把性能优化的工程师推上去优化，就缺乏一定的目的性，谁也不知道优化到什么程度是个头。比方说，当前性能是 80 TPS，那么要优化到 100 TPS，可能小修小补，很小的代价就可以做到；但是要优化到 500 TPS，那么很可能要在产品和架构上做调整，代价往往是很不一样的，没有预期而直接生米煮熟饭显然是不合工程性的。

其次，过一遍待优化业务的环境、配置、代码，对于整个请求的处理流程做到心中有数。说白了，这就是走读代码，这个代码既包括应用的源码，也要了解环境和配置，知道整个流程的处理一步一步是怎样进行的。比如说，浏览器上用户触发事件以后，服务端的应用从线程池里面取得了线程来处理请求，接着查询了几次数据库，每次都从连接池中取出或建立连接，进行若干的查询或事务读写操作，再对响应做序列化返回浏览器，而浏览器收到请求以后又进行解析并将执行渲染。值得一提的是，这个过程中，全栈工程师的价值就体现出来了，因为他能够有端到端的视野来看待这个问题，分析整个流程。

再次，开始压测，缓慢、逐步增大请求速率，直到预期的业务角度的性能指标无法再提升，待系统稳定。有些指标可能一开始就不满足要求，比如某系统的延迟，在一开始就不达标；也有时候是随着压力增大才发现不达标的，比如多数系统的吞吐量，随着压力增大，吞吐量上不去了。通常，我们可以给到系统一个足够的压力，或者等于、甚至高于系统最大的吞吐量，且待系统稳定后，再来分析，这种情况下我们可以把问题暴露得更彻底。

最后，才是根据业务角度的指标来进行特定的分析，我就以最常见的延迟和吞吐量这两个为例来说明。

1. 对于延迟一类的问题

从大的流程到小的环节，从操作系统到应用本身，逐步定位，寻找端到端的时间消耗在哪里。比方说，可以**优先考虑系统快照类的工具**，像是 jstack、kill -3 命令可以打印系统当前的线程执行堆栈，因为这一类工具往往执行比较简单，对系统的影响较小；如果还需要进一步明确，**其次再考虑侵入性较强的，运行时剖面分析（profile）类型的工具**，比如 JProfiler。如果发现运行的 50 个线程里面，有 48 个卡在等某一个锁的释放上面，这就值得怀疑该锁的占用是造成延迟问题的罪魁祸首。

2. 对于吞吐量一类的问题

这类问题 CPU 是核心。在请求速率逐步增大时，我们来看这样四种情况：

吞吐量逐步上升，而 CPU 变化很小。这种情况说明系统设计得比较好，能继续“扛”更高的负载，该阶段可以继续增大请求速率。

吞吐量逐步上升，而 CPU 也平稳上升。其实这还算是一种较为理想的系统，CPU 使用随着吞吐量的上升而平稳上升，直到 CPU 成为瓶颈——这里的原因很简单，用于业务处理的 CPU 是真正“干活的”，这部分的消耗显然是少不了的。如果这种情况下 CPU 已经很高了却还要优化，那就是说在不能动硬件的情况下，我们需要改进算法了，因为 CPU 已经在拼命干活了。就像程序员一周已经加班五十个小时了，还要再压迫他们，就要出事（系统崩溃）了。

吞吐量变化很小甚至回落，而 CPU 大幅上升。这种情况说明 CPU 的确在干活，却不是用于预期内的“业务处理”。最典型的例子就是，内存泄漏，系统在一个劲地 GC，CPU 看起来忙得要命，但是却不是忙于业务。

吞吐量上不去，CPU 也上不去。这种情况其实非常常见，而且随着压力的增大，它往往伴随着延迟的上升。这种情况是说，CPU 无法成为瓶颈，因为其它瓶颈先出现了，因此这种情况下就要检查磁盘 I/O、网络带宽、工作线程数等等。这就像是一个运动员的潜力（CPU）还很大，但是由于种种原因，比如技巧问题、心理原因，潜力发挥不出来，因此我们优化的目的，就是努力找到并移除当前的瓶颈，将 CPU 的这个潜力发挥出来。

工具的分类

在定位性能瓶颈的过程中，我们会和各种各样的工具打交道，在我看来，这些工具中最典型的有这样三种类型：

截取型：截取系统某个层面的一个快照加以分析。比如一些堆栈切面和分析的工具，jstack、jmap、kill -3、MAT、Heap Analyser 等，这类工具使用简单，对系统影响往往比较小。

监控型：监视、剖析系统变化，甚至数据流向。比如 JProfiler、JConsole、JStat、BTrace 等等，如前文所述，这类工具的侵入性往往更高，甚至本身就会大幅影响性能。

验尸型：系统已经宕机或完成运行任务了，但是留下了一些“罪证”，我们在事后来分析它们。最有名的就是 JVM 挂掉之后可能会留下的 hs_err_pid.log，或者是生成的 crash dump 文件。

总结思考

今天我们学习了性能优化的基本知识，包括性能优化和软件设计的关系，性能指标和关注点，以及怎样去寻找性能瓶颈，希望你能够理解，在跳进性能优化的“坑”之前先有了足够的认识、分析和思考。

现在，我来提一个问题吧：对于下面这些资源角度定义的性能指标，你能说说在 Linux 下，该用怎样的工具或命令来查看吗？

CPU 使用率、负载；

可用内存、换页；

磁盘 I/O；

网络 I/O；

应用进程、线程。

最后，欢迎你来分享你在性能优化中做测试和寻找性能瓶颈的故事，我相信这些经历，都会很有意思。

扩展阅读

关于性能指标，我提供两篇文章你可以做扩展，一篇是美团性能优化之路——性能指标体系，另一篇是性能测试中服务器关键性能指标浅析。

第二篇也相关，来自阿里云性能测试 PTS 的官方文档，测试指标，当然，对于文档中感兴趣的章节你也可以一并通读。

上一篇　33 | 特别放送：聊一聊程序员学英语

下一篇　35 | 网站性能优化（下）

精选留言 (3)

写留言


leslie

2019-11-27

打卡：这块自己是自己的专长。

展开

1

Mandalorian

2019-11-28

好想实战

展开

作者回复: 实战确实能够获得直接的知识，但是实战总是较为具体，不如这些更一般的经验和原则有普适性。因此我们二者都要学。

Paradise丶朽木

2019-11-27

“如果测试阶段还没有做到，那就只好等着上线以后，问题发生来打脸了。”
这个真的是印象深刻...

展开