第 1 章 绪　论

一般而言，人们评估一个 Web 站点的性能如何，通常先置身于用户的角度，来访问该站点的一系列页面，体验等待时间。

当用户输入页面地址后，浏览器获得了用户希望访问该地址的意图，便向站点服务器发起一系列的请求。注意，这些请求不仅包括对页面的请求，还包括对页面中许许多多组件的请求，比如图片、层叠样式表（CSS）、脚本（JavaScript）、内嵌页面（iframe）等。接下来的一段时间，浏览器等待服务器的响应以及返回的数据。待浏览器获得所有的返回数据后，经过本地的计算和渲染，最终一幅完整的页面才呈现于用户的眼前。

1.1　等待的真相

整个过程听起来好像并不复杂，也许你从来都没有考虑过在这段等待的时间里世界都发生了什么变化，也许你早已习惯了利用这段时间东张西望或者品尝零食，或者你根本没有来得及意识到这点，新的网页就已经闪亮登场。恭喜你，你很幸运！但是在这个世界上，幸运儿永远只占少数，大多数人的大脑处理速度已经让他们明显感觉到这段等待时间漫长无比，久经考验的他们可以随时身手敏捷地打开多个浏览器窗口与时间赛跑，并为此筋疲力尽。

另一方面，对于站点经营者来说，让用户等待的时间过长，也许会造成毁灭性的后果。笔者见过很多人为了享用某家特色小吃而在餐馆门口乐此不疲地排着长队，但没有听说有多少用户执著地等待着一个速度缓慢的站点而不去尝试别的站点。

在这段等待的时间里，到底发生了什么？事实上这并不简单，大概经历了以下几部分时间：

- 数据在网络上传输的时间。
- 站点服务器处理请求并生成回应数据的时间。
- 浏览器本地计算和渲染的时间。

数据在网络上传输的时间总的来说包括两部分，即浏览器端主机发出的请求数据经过网络

到达服务器的时间，以及服务器的回应数据经过网络回到浏览器端主机的时间。这两部分时间都可以视为某一大小的数据从某主机开始发送，直到另一端主机全部接收所消耗的总时间，我们称它为响应时间，它的决定因素主要包括发送的数据量和网络带宽。数据量容易计算，但是究竟什么是带宽呢？我们将在后续章节中详细介绍带宽的本质。

站点服务器处理请求并生成回应数据的时间主要消耗在服务器端，包括非常多的环节，我们一般用另一个指标来衡量这部分时间，即每秒处理请求数（也称吞吐率）。注意这里的吞吐率不是指单位时间内处理的数据量，而是请求数。影响服务器吞吐率的因素非常多，比如服务器的并发策略、I/O 模型、I/O 性能、CPU 核数等，当然也包括应用程序本身的逻辑复杂度等。这些将在后续章节中详细介绍。

浏览器本地计算和渲染的时间自然消耗在浏览器端，它依赖的因素包括浏览器采用的并发策略、样式渲染方式、脚本解释器的性能、页面大小、页面组件的数量、页面组件缓存状况、页面组件域名分布以及域名 DNS 解析等，并且其中一些因素随着各厂商浏览器版本的不同而略有变化。这部分内容在后续章节中也会适当提到。

可见，一个页面包含了若干个请求，每个请求都或多或少地涉及以上这些过程，假如有一处关键环节稍加拖延，整体的速度便可想而知。

现在，如果有用户向你抱怨在打开站点首页的时候等待了很久，你知道究竟慢在哪里了吗？

1.2 瓶颈在哪里

相信你一定知道赤壁之战，这是中国历史上一场著名的以少胜多的战役，东吴的任务是击退曹操的进攻，要完成这项任务，可谓“万事俱备，只欠东风”，这时东风便是决胜的瓶颈，所以很多系统论研究专家将其称为“东风效应”，也就是社会心理学里讲的“瓶颈效应”。

之所以称它为瓶颈，是因为尽管东吴做了很多战前准备，包括蒋干中计导致曹操错杀蔡瑁和张允、诸葛亮草船借箭、东吴苦练水军等，但是仅靠这些仍无法获得最终胜利，还需要最后的东南风才能一锤定音，完成火烧曹军战船的计划。不过之前的准备工作都是胜利的子因素，而东南风这个关键因素最终和其他子因素一起相互作用，将整个战斗的杀伤力无限放大。

曹操运气不好，遇上东南风，倒了大霉，曹军战船一片火海，这时候东吴需要派出勇猛的

陆军部队登岸攻下曹营，可是东吴向来精通水战，几乎没有强大的陆战部队，只有老将黄盖，这如何与曹操的精英骑兵抗衡呢？这个时候决胜的关键因素变成了刘备的盟军支援，五虎上将各个威猛无比，身怀必杀绝技，此时正是上岸一显身手的好机会，他们不费吹灰之力就将曹军打得落花流水，试想如果没有刘备的支援，赤壁之战胜败可能就扑朔迷离了。

可见，系统性能的瓶颈是指影响性能的关键因素，这个关键因素随着系统的运行又会发生不断的变化或迁移，比如，由于站点用户组成结构的多样性和习惯的差异，导致在不同时段系统的瓶颈各不相同，又如，站点在数据存储量或浏览量增长到不同级别时，系统瓶颈也会发生迁移。一旦找到真正影响系统性能的主要因素，也就是性能瓶颈，就要坚决对其进行调整或优化，因为你不得不这么做。

提示：

中医是一门关于生命的哲学，也是中国人智慧的结晶，它的光芒在于独到的思辨能力和系统性的分析方法，它认为世间万物都在不停地变化，并赋予它们阴阳状态，包括天地、季节、天气、心理、生理等，而患者的病理也在随之变化。所以，中医会对同一位患者在不同季节进行不同的诊断，找到不同的病因。

同时，在这些关键因素的背后，也存在很多不能忽略的子因素，构成了性能优化的“长尾效应”，也就是说，如果你对某个子因素背后的问题进行优化，可能会带来性能上的少许提升，也许不被察觉，但是多个子因素的优化结果也许会叠加在一起，带来性能上可观的提升。对于诸多子因素的优化，需要稍加谨慎，花点时间考虑这种优化是否值得，以及是否会带来潜在的副作用，还有其他依赖的非技术因素。

然而，不论是关键因素还是子因素，它们的背后都是影响系统性能的问题所在，问题本身并不涉及关键性，只有在不同的系统和应用场景下，才会显示出其是否关键。

本章其余部分将先列出我们经常遇到的一些问题，并简单介绍我们常用的优化方案，至于这些问题在什么时候是否关键，它们的本质是什么，以及如何调整或优化，在后续章节中将结合具体场景来详细探讨包括这些在内的更多主题，这也是本书贯穿始终的线索。

1.3 增加带宽

当 Web 站点的网页或组件的下载速度变慢时，一些架构师可能想到的最省事的办法就是增加服务器带宽，因为他们认为是服务器带宽不够用了，对于一些以提供下载服务为主的站点来说也许是这样的，但是对于其他服务的站点，你知道站点当前究竟使用了多少带宽

吗？这些带宽都用到哪里了呢？如何计算站点现在和可预见未来使用的带宽？带宽增加后下载速度就可以加快吗？使用独享带宽和共享带宽的本质区别是什么？如何节省带宽？还有，你可能会忍无可忍地问，究竟什么是带宽？

对于带宽的概念，如果你没有仔细阅读计算机网络教材中的描述，笔者敢肯定你一定是完全凭借自己的理解来认识它的，因为这个词实在是太有创意了，也实在太容易从字面理解了，但是这些认识从本质上讲完全是错误的，正是基于这种误解，很多人都无法完全解答上述那一连串问题，导致在所有涉及带宽的问题上，只能依靠经验和猜想。

在后续章节中，我们将通过介绍数据的网络传输原理，彻底揭开带宽的本质，以及数据传输响应时间的依赖因素和计算方法。搞清楚这些一点都不困难，它们是一个优秀架构师必须掌握的基础知识。

1.4 减少网页中的 HTTP 请求

我们知道，Web 站点中几乎任何一个网页都包含了多个组件，每个组件都需要下载、计算或渲染，毫无疑问，这些行为都会消耗时间。那么如果可以让网页减少这些行为，应该就可以加快网页的展示速度，这是毫无疑问的，但是往往我们需要在优雅的网页表现和性能之间权衡取舍，这也许是美和快之间的博弈，找到最优的均衡点至关重要，我们为此做了很多尝试和努力：

- 设计更加简单的网页，使其包含较少的图片和脚本，但是这可能牺牲了美观和用户交互。
- 将多个图片合并为一个文件，利用 CSS 背景图片的偏移技术呈现在网页中，避免了多个图片的下载。
- 合并 JavaScript 脚本或者 CSS 样式表。
- 充分利用 HTTP 中的浏览器端 Cache 策略，减少重复下载。

很显然，这些技巧都来自 Web 网页前端的优化，在后续章节中我们会有所涉及，但是不作为本书的重点来介绍，本书将更加偏重于站点服务器端的性能改善和规模扩展。

1.5 加快服务器脚本计算速度

笔者认为，大多数涉及性能问题的站点都会使用各种各样的服务器端脚本语言，比如主流

的 PHP、Ruby、Python、ASP.NET、JSP 等，这些脚本语言用来编写动态内容或者后台运行的小程序，已经成了几乎所有站点的首选。而曾经使用 C++编写动态内容的经历也让笔者记忆犹新，除了每天都在感叹 C++的严谨和优雅之外，笔者找不到其他任何好处。

我们知道，用这些脚本语言编写的程序文件需要通过相应的脚本解释器进行解释后生成中间代码，然后依托在解释器的运行环境中运行。所以，生成中间代码的这部分时间又成为大家为获取性能提升而瞄准的一个目标，对于一些拥有较强商业支持的脚本语言，比如 ASP.NET 和 JSP，均有内置的优化方案，比如解释器对某个脚本程序第一次解释的时候，将中间代码缓存起来，以供下次直接使用。

对于开源类的脚本语言，也有很多第三方组件来提供此类功能，比如 PHP 的 APC 组件等。使用这些组件进行脚本优化真的那么有用吗？不同的应用效果是否有所不同呢？在后续章节会详细探讨。

1.6 使用动态内容缓存

动态内容技术就像 Web 开发领域的一场工业革命，它带来了产业升级和 Web 开发者的地位提升，在过去相当长一段时间里，大家普遍认为一个站点的技术含量主要体现在后台的动态程序上，所以很多工程师都会带着虚荣心警告你：“请叫我后台开发工程师。”事实上，这种概念和偏见已经开始逐渐被历史抛弃，但这不是我们此刻讨论的重点。

自动态内容技术产生后，聪明的工程师们为了减少动态内容的重复计算，想到了截取动态内容的胜利果实，将动态内容的 HTML 输出结果缓存起来，在随后的一段时间内，当有用户访问时便跳过重复的动态内容计算而直接输出。

在实际应用中，动态内容缓存可能是大家使用得最多的技术，但是并不见得所有的动态内容都适合使用网页缓存，缓存带来的性能提升恰恰与有些动态数据实时交互的需求形成矛盾，这是非常尴尬的，而解决该问题的唯一途径不是技术本身，而是你如何权衡。

另一方面，缓存的实现还涉及了一系列非常现实的问题，即成千上万的缓存文件如何存储？缓存的命中率如何？缓存的过期策略如何设计？在拥有多台 Web 服务器的分布式站点上应用动态内容缓存需要考虑什么呢？

在后续章节将详细探讨这些问题。

1.7 使用数据缓存

动态内容缓存是将数据和表现整体打包，一步到位，但就像快餐店里的组合套餐一样，有时候未必完全合乎我们的口味。当我们意识到在自己的站点中，某些动态内容的计算时间其实主要消耗在一些烦人的特殊数据上，这些数据或者更新过于频繁，或者消耗大量的 I/O 等待时间，比如，对关系数据库中某字段的频繁更新和读取，这时我们为了提高缓存的灵活性和命中率，以及性能的要求，便开始考虑数据缓存。

更加细粒度的数据缓存避免了过期时大量相关网页的整体更新，比如很多动态内容都包含了一段公用的数据，如果我们将整个页面全部缓存，那么假如这段数据频繁更新导致频繁过期，无疑会使所有的网页都要频繁地重建缓存，这对网页的其他部分内容似乎很不公平。此时如何协调网页缓存和数据缓存呢？是否能够将它们一起使用并各显其能呢？

另外，将数据缓存存储在哪里呢？这需要考虑多方面的因素。速度是一方面，如果无法提供高速的读写访问，那么这部分数据缓存可能不久便成为新的系统瓶颈。另外，数据缓存的共享也至关重要，如同一主机上不同进程间的共享、网络上不同主机间的共享等，一旦设计不当，将对站点未来的规模扩展带来致命的威胁。

在后续章节中也会详细探讨这些问题。

1.8 将动态内容静态化

在动态内容缓存技术的实现机制中，虽然避免了可观的重复计算，但是每次还都需要调用动态脚本解释器来判断缓存是否过期以及读取缓存，这似乎有些多余，而且关键是消耗了不少时间。直接让浏览器访问这些动态内容的缓存不是更好吗？在这种情况下，缓存成为直接暴露给前端的 HTML 网页，而整个缓存控制机制也发生了根本的变化，我们普遍称它为静态化，静态网页独立了，当家做主了，再也不用被脚本解释器呼来唤去。

独立意味着要承担更多的责任，原本动态内容缓存涉及的那些问题，在静态化实践中是否也会出现呢？让我们在后续章节中详细探讨。

1.9 更换 Web 服务器软件

从 20 世纪末开始影响全球经济的开源软件，不可否认给我们的生活带来了更多丰富的体

验和选择，但是更多的选择也代表着更多的结局，不论结局是好是坏，我们都需要为此承担责任。

在 Web 服务器软件的选择问题上，很多架构师依然很困惑，大量的压力测试对比数据蛊惑着激进的开发者和运维工程师，人们只关注所谓的并发量冠军，却忽视了更加本质性的东西，甚至不了解眼前测试数据的潜在前提。社会总是这样的，象牙塔式的精英教育和残酷的淘汰机制断送了无数人才的未来。而这一次，错误的选择将要付出沉痛的代价。有人拿着所谓的测试数据说 Apache 已经过时，你相信吗？也许下此结论为时尚早，尽管放弃它的人比比皆是，但是它的成功不是空穴来风，毕竟它已经活了很久了。

另一方面，你正在使用的 Web 服务器软件也许让你无比自豪，可是你知道那些复杂的参数配置背后的本质吗？你知道为什么它仅仅在处理你的站点请求时如此出色吗？如果让你自己编写 Web 服务器软件，你可以让它更快一些吗？

我们必须停止盲目的选择，停止对表面现象的崇拜，我们需要学习一些稍显底层的知识来武装自己。在后续章节中，我们将介绍 Web 服务器在并发策略方面的各种设计及其动机和本质，熟悉这些内容后，相信你可以解释和分析更多你所看到的现象。

1.10 页面组件分离

从某种角度看，中学校园里的快慢分班似乎合乎逻辑，虽然不一定合乎情理。快班的学生学习能力强，理解知识快，那么课程安排的节奏可以加快一些；慢班的学生则可以放缓课程安排的节奏，这样既互不影响，学校的升学率又可以得到保证，当然，假设的前提是学生之间互相帮助的效果不大。

在 Web 站点中，网页和各种各样的组件是否也需要“分班”呢？显然，它们的下载量和对服务器的能力要求不尽相同，如果由同一台物理服务器或者同一种并发策略的 Web 服务器软件来统一提供服务，那势必造成计算资源的浪费以及并发策略的低效。所以，分离带来的好处是显而易见的，那就是可以根据不同组件的需求，比如下载量、文件大小、对服务器各种资源的需求等，有针对性地采用不同的并发策略，并且提供最佳的物理资源。

当然，如果你的站点基本无人问津，而且服务器的各种资源大量闲置，那么自然不存在什么性能问题，也不需要什么组件分离。但是如果你的站点负载已经让你意识到组件分离是大势所趋，那还是趁早动手。

那么，什么组件需要分离？如何分离？幸运的是，这些并不困难，但是其涉及的知识绝对

不仅仅是组件分离本身，在后续章节中将会详细探讨。

1.11 合理部署服务器

真让人发疯，互联网为什么不是只有一个？你也许会说难道不是只有一个 Internet 吗？是的，但是 Internet 所特指的“互联网”是某种文化意义上的名词，同一个世界，同一个梦想，同一个互联网。而本书说的互联网，则是指由某互联网运营商负责搭建的一系列网络节点，它覆盖的地域有大有小，接入这些网络节点的局域网也可以相互通信，同时这些互联网之间也能够通过骨干线路互联互通。

世界上很多国家都有不止一个互联网运营商，中国的互联网运营商想必大家都非常熟悉，当你在家中安装宽带或者需要托管服务器的时候，都面临着运营商选择的问题，包括电信、网通、铁通、移动在内的几大国内运营商让你很头疼。特别是在部署 Web 站点各类服务器的时候，是否能够找到合理的位置部署服务器至关重要。

我们都知道，在基于 IP 寻址的互联网中，IP 地址相近的主机之间通信，数据经过少数的路由器即可到达，比如，同一局域网内通信或者接入同一个城市交换节点的局域网之间通信，在这种情况下，数据到达时间相对较短。

而如果通信的两端主机位于不同运营商的互联网中，那么数据必须流经两个互联网运营商的顶级交换节点和骨干线路，可想而知，在这个过程中的数据要经过更多次的存储转发，而且各互联网顶级交换节点之间又存在出口带宽的限制，如果互联网之间数据通信量比较大的话，那么这个顶级交换节点也就是“出口”，将会是瓶颈所在，就像连接两座城市之间的高速公路，当大量汽车需要频繁地往返于两座城市时，高速公路出现车流缓慢，那么汽车从一个城市到另一个城市的总时间加长了。

显而易见，我们当然希望 Web 站点的用户和服务器位于同一个互联网运营商的网络内，但如何实现呢？后续章节会涉及这方面的详细内容。

1.12 使用负载均衡

到此为止，我们已经最大程度地发挥了单台 Web 服务器的处理能力，但是，当它所承受的压力达到极限时，就需要有更多的服务器来分担工作，我们需要想办法将流量合理转移到更多的服务器上。

为此，我们需要通过各种不同的方法来实现 Web 负载均衡，可能是简单的 HTTP 重定向，或者是基于 DNS 的轮询解析，或者通过反向代理服务器来实现负载均衡调度，还可以通过 LVS 来组建服务器集群，它们有什么区别呢？无论如何，通过这些具体的实现方法，我们更加关心的是能否真正地均衡调度请求，以及是否具备高可用性，还有影响规模扩展的制约因素，这些内容都会在后续章节中详细介绍。

1.13 优化数据库

对于使用数据库的 Web 站点来说，你是否在性能优化时或多或少地忽视了数据库的存在？往往一些性能问题都发生在表现不佳的数据访问层面，这来源于不合理的应用程序数据访问组件设计、不合理的数据库表结构设计以及对于数据库内部构造缺乏深入的了解。毫不夸张地说，也许你之前的优化全都白干了。

Web 服务器与数据库服务器的数据通信一般基于标准的 TCP，即便它们位于同一台物理主机也是如此。其通信连接的建立和释放涉及代表一段内核高速缓冲区的文件描述符的创建和销毁，这需要不少的时间开销，包括系统调用导致的内核态切换以及某些异步阻塞 I/O 模型采用的文件描述符队列扫描机制。所以，频繁的数据库连接和释放无疑将导致数据访问等待时间的加长，这段时间浪费得毫无意义。

使用数据库持久连接有效地解决了这一难题，它包括不同程度上的持久化，本质的区别在于持久连接的应用范围和生命周期，比如某个进程内部的全局数据库连接，供进程内所有计算任务共享，在这个进程终止后便被释放；或者在某个动态内容的执行周期内，代码层面的持久连接对象，在动态内容计算结束后便不复存在；还有跨进程的数据库连接池，保存多个持久连接供应用程序重复使用。在这些采用数据库持久连接的应用设计中，同时还要注意保证数据访问的线程安全性。

与此同时，在设计关系数据库的表结构时，你是否合理使用了各种类型的索引呢？要做到这一点，你必须了解索引的有关知识，然而更重要的是如何根据 Web 站点变幻莫测的数据访问特点来有针对性地设计每个表的索引，这往往也是最有难度的，索引的合理使用对于依赖数据库访问的 Web 应用至关重要。

另外，你了解数据库存储引擎的特性吗？其实这并不困难，因为所有的主流数据库文档中都有详细介绍，但是究竟你的 Web 站点应该选择什么存储引擎呢？当然，没有绝对完美的方案，我们在这个世界上要做的唯一的事情就是不断进行取舍，像考虑索引一样去弄清楚存储引擎的本质，是绝对不会让你失望的。

随着时间的推移，你的 Web 站点可能逐渐被数据库绑架，单台数据库服务器再也无法应付整个站点的需要，这包括存储空间以及查询时间，人们开始抱怨数据库模型的不良设计制约了横向扩展以及负载均衡，这不是我们希望看到的结果。为此，我们将数据散列在多台主机，包括必要的冗余数据，以此来合理地分散数据库的密集访问，数据库扩展便成为我们考虑的方案。

1.14 考虑可扩展性

对于前面列举的诸多方案，本书不仅从其对性能影响的角度来深入探讨，还会适当地涉及开发、调试以及可扩展性。对于 Web 站点的可扩展性讨论已经屡见不鲜了，不论是代码层面的扩展，还是架构层面的扩展，涉及的内容非常多，究竟我们应该从何谈起呢？这是一个值得深思的问题。缺乏良好的可扩展性设计就像慢性自杀或者等待死亡，这甚至比 Web 站点所能遇到的其他一切困难更让人头疼，因为扩展对我们来说，就像在山穷水尽的时候被指引了一条星光大道，一旦扩展都无法进行，那真是死路一条。

的确，可扩展性并不是性能和速度的概念，它是指当系统负载增大时，通过增加资源来提高性能的能力。高性能往往需要通过这种能力来实现快速扩展，几乎没有多少团队可以在一个星期内通过增加服务器马上让服务能力扩容 100 倍。另一方面，可扩展性的目的在于适应负载的变化，从扩展的技术实现上来看，又包含了很多对局部性能的思考，以及了解何时需要扩展，这离不开对站点性能的把握。

然而，就像“人的病都是吃出来的”一样，Web 站点在成长的道路上不断吸收新的技术，但每一次技术的应用不当都可能引入一定程度上的不可扩展。现实往往是矛盾的，我们不得不使用一些技术来构建 Web 站点，同时又使用一些技术来提升站点性能，这些技术构成了我们理想架构的一部分，关键在于在这些技术和架构的应用中，我们是否意识到可扩展性，并且能否正确评估可扩展性的需求。

在本书后续的内容中，我们不打算用一个独立的章节来探讨可扩展性问题，而是将其渗透在所有需要考虑可扩展性的章节中，因为这样的组织方式可能更加适合读者阅读。

1.15 减少视觉等待

实在不行就给用户一些提示吧！最后笔者只能这么说了，事实上，这不是什么大不了的事情，即使认识到架构的瓶颈并投入大量人力来改善，也不是一天两天就可以完成的，要意

识到用户也许只是希望你不要不理他而已。

这部分显然已经超出了本书的讨论范围，它涉及人机交互的相关知识，并且充满着人文情怀，要真正做好它，恐怕要比本书中所有的问题都更有难度和挑战性，这毫不夸张，我们要承认这个现实，因为世界上最难的学问就是研究人，你觉得呢？