TURING 图灵程序设计丛书

PEARSON

高扩展性网站的50条原则

Scalability Rules 50 Principles for Scaling Web Sites

[美] Martin L. Abbott Michael T. Fisher 著 张欣 杨海玲 译

- 网站运营的必备宝典
- 详细分析网站扩展性的通用原则
- 业内专家多年实战总结，亚马逊书店好评如潮

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

高扩展性网站的50条原则

Scalability Rules 50 Principles for Scaling Web Sites

[美] Martin L. Abbott
Michael T. Fisher 著
张欣 杨海玲 译

人民邮电出版社
北京

图书在版编目（CIP）数据

高扩展性网站的50条原则 / (美) 阿博特(Abbott,M.L.), (美) 费希尔 (Fisher,M.T.) 著 ; 张欣, 杨海玲译. -- 北京 : 人民邮电出版社, 2012.6
(图灵程序设计丛书)
书名原文: Scalability Rules: 50 Principles for Scaling Web Sites
ISBN 978-7-115-27572-1

Ⅰ. ①高… Ⅱ. ①阿… ②费… ③张… ④杨… Ⅲ. ①网站－设计 Ⅳ. ①TP393.092

中国版本图书馆CIP数据核字(2012)第027844号

内 容 提 要

本书给出了设计高扩展网站的50条原则，如不要过度设计、设计时就考虑扩展性、把方案简化3倍以上、减少DNS查找、尽可能减少对象等，每个原则都与不同的主题绑定在一起。大部分原则是面向技术的，只有少量原则解决的是与关键习惯和方法有关的问题，当然，每个原则都对构建可扩展的产品至关重要。

本书适合各层次Web开发人员阅读。

图灵程序设计丛书

高扩展性网站的50条原则

◆ 著 [美] Martin L. Abbott Michael T. Fisher
译 张 欣 杨海玲
责任编辑 卢秀丽
执行编辑 杨 爽 刘美英

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司

◆ 开本：880×1230 1/32
印张：7.875
字数：211千字 2012年6月第1版
印数：1－4 000册 2012年6月河北第1次印刷
著作权合同登记号 图字：01-2012-0973号
ISBN 978-7-115-27572-1

定价：35.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

致　谢

本书中介绍的原则不全是由我们的公司提出的，它们是近60年来将近200个公司、部门和组织的客户、同事以及合作伙伴一起工作的成果。他们每一位都在不同程度上对本书的全部或部分原则有所贡献。因此，我们要向过去几十年中与我们合作过的朋友、合作伙伴、客户、同事和老板表示感谢。

还要感谢本书的编辑，他们给我们提供了很多指导、反馈和项目管理经验。技术编辑 Geoffery Weber、Jeremy Wright 和 Robert Guild 与我们分享了他们几十年的技术经验，提供了非常宝贵的建议。Addison-Wesley 的编辑 Songlin Qiu 和 Trina MacDonald 在本书编写过程中为我们提供了文体和修辞方面的指导。感谢大家的帮助。

最后，要感谢我们的家人和朋友，感谢他们容忍我们缺席各种社交活动，只坐在计算机屏幕前敲个不停。著书非一人之力可以完成，没有家人、朋友的理解和支持，这个过程会相当艰苦。

前　言

感谢你对本书感兴趣！本书既可以作为初级读物，又可以作为复习资料和简单的参考手册，帮助每一位工程师、架构师、管理者开发和维护可扩展的互联网产品。本书设计了一系列原则，每个原则都与不同的主题绑定在一起。大部分原则是面向技术的，只有少量的原则解决的是与关键的习惯和方法有关的问题，当然，每个原则都对构建可扩展的产品至关重要。这些原则的深度和关注点各不相同。有些原则是通用的，如定义一个几乎适用于所有可扩展性问题的模型；而有些原则则是专用的，解释了某种技术，例如，如何修改文件头，从而让内容尽量适合缓存。

快速使用指南

对于经验丰富的工程师、架构师和管理者来说，都应该首先阅读所有原则开头的说明部分，它包含了原则的目的、适用情形、应用方式以及应用理由。读者可以按顺序浏览每一章，也可以直接跳到第 13 章，该章汇总了所有原则的开头说明部分。读过这些说明之后，再去阅读对你来说全新的章节或者你感兴趣的章节。

对于经验较少的读者来说，一下子面对 50 条原则可能会令你不知所措。我们虽然确信你最终会了解所有原则，但是也充分理解你需要根据事情的轻重缓急来分配时间。正因为如此，我们专为管理者挑出了 5 章

内容（第1、2、4、7、12章），为软件开发人员挑出了5章内容（第1、2、5、10、11章），为技术运维人员挑出了5章内容（第2、3、6、8、9章），你应该优先阅读相应章节的内容，这样可以帮助你尽快掌握高扩展性知识。

无论你具体从事哪种工作，我们都建议你在有时间的时候阅读本书的所有原则，熟悉这些原则和概念。本书很薄，也许在某次短途航班上就能读完。

本书可以作为案头的参考资料。如果你想修正或者重新架构现有产品，那么第13章提供了一种方法，可以让你基于成本和预期收益的考虑将这些原则应用到现有平台上。如果你已经有了自己的分级方法，那么除非你认为我们的更好，否则还是不要改变它。如果你还没有分级方法，那么我们的方法则可以帮助你考虑首先应该应用哪条原则。

如果你刚开始开发一个新产品，那么这些原则可以算是开发高扩展性产品的最佳实践。在这种情况下，第13章提供的划分优先级的方法，可以在你设计产品时指导你最应该考虑哪些方面。你应该看看那些最可能使短期和长期需求具有高可扩展性的原则，然后将其付诸实践。

对于所有公司来说，这些原则有助于形成一套推动未来发展的架构理论。选用原则5、原则10或者原则15能使你的产品具有高扩展性，并可使用它们提升现有设计。工程师和架构师可以针对你选的每一条扩展原则提出一些问题，以确保新的设计能达到高扩展性标准。尽管这些原则是具体且固定的，但仍然有可修改的空间，具体要视系统的特殊要求而定。如果你和你的团队具有丰富的扩展经验，那么可以对这些原则进行必要的修改，使它们适用于特定场景。如果不具备这种经验，那就严格遵守这些原则吧，看看它们能实现多大的可扩展性。

最后要说的是，本书还可当做一本参考手册。第13章是一个快速索引，概述了所有原则。无论是遇到了问题，还是想开发一个更具备扩展

性的解决方案，都可以查阅第 13 章，以最快的速度找到能脱离困境的原则，或者在新的开发过程中发现最佳路径。除了把本书作为案头的参考手册，你还可以用各种方法把它融入到组织中，例如每周实践一两个原则并且在技术例会上讨论。

本书写作意图

目前市场上还没有什么关于高扩展性的好书。就讲述的方法而言，本书在市场上独一无二。它是第一本以面向原则的方式阐述高扩展性的书，又是第一本既提供了相关主题概览，又可以作为参考手册的书。对于想将本书应用于现有平台的读者，我们还特别准备了一章，对 50 条原则进行了总结和分级。

在我们的博客上，得到评论最多的一篇博文是关于将高扩展性作为一门学科的。我们和那些研究扩展性问题的专家都认为，当今的科技公司急需高扩展性架构师。在计算机系统发展的早期，几乎人人都是程序员，然后逐渐分化为运维人员、DBA、架构师等。现在的技术团队是由许多不同学科和专业的人才组成的，其中就缺少高扩展性架构师这类人才。

DBA 只需要把自己的事情做完，而不必再教其他人，除非正在带初级 DBA。而高扩展性架构师则与之不同，他们的一项主要职责就是培养技术人员。高扩展性架构师应该是老师与布道者，而不是知识的保密者。我们把 50 条原则集合起来，为这种教学奠定基础，相信这 50 条原则能够为扩展自己系统的公司提供指导性帮助。

如何决定选用哪 50 条原则

决定选用哪些原则并不容易。一本书可以轻松地写上 100 条甚至 200 条原则。我们选用原则的标准是看我们最常向客户提出哪些建议，以及

我们针对客户的产品最常推荐哪些变更、添加和修正。我们发现在前 50 条原则之后，推荐率大幅下降。这并不是说，前 50 条原则的推荐率是完全相等的，也不是说第 51 条就相当不常用。只是说，我们经常向客户推荐这 50 条。这些原则的介绍顺序也并不是根据它们的推荐频率来排的。在第 13 章中，我们对每条原则能降低多少风险，以及实施或采用起来的成本进行了评级，然后根据应用这些原则的好处和优先级对它们进行了分组。

本书和*The Art of Scalability*[①]有什么不同

The Art of Scalability 是我们所著的有关该主题的第一本书，重点关注人员、方法和技术，而本书则专注于技术。不要误解了，我们始终认为人员和方法是构建高扩展性解决方案最重要的因素。毕竟，在制定可扩展的解决方案方面获得成功或遭遇失败的是公司，而公司则是由做出贡献的个体和管理制度构成的。扩展的失败不是技术的过失，它是人在构建、选择或集成时犯下的错误。不过我们认为，*The Art of Scalability* 一书已经足以解决与高扩展性相关的人员和方法的问题，而我们想要的是在技术方面更深入一层。

本书展开论述了上一本书的第三部分（技术部分）。本书中的内容比上一本书更新，讨论的方式也更技术化。上一本书中讨论过的内容，本书中都进行了扩展，或者用稍微不同的方式进行了定义，以便读者更好地理解这些概念。

① 该书中文版《可扩展的艺术》即将由人民邮电出版社出版。——编者注

目　录

第1章 化简方程

在我们的学习或工作中，都曾遇到过这样的情况：死盯着一个复杂的问题不放，以至于最后丧失了希望。我们应该从何处入手？如何在预定的时间内解决问题？或者说得极端一些，如何在有生之年解决这个问题？要做的事情太多了，这个问题太过棘手，是不能解决的，事实如此，就此罢手吧。游戏结束了……

等等，不要丧失希望。深呼吸，想想你的高中或者大学数学老师。就像解数学题，把大方程化简成易于计算的小方程一样，如果你遇到的是棘手的大型架构问题，那么可以把大问题分拆成小问题，把小问题分拆成更小的问题，直到这些问题可以轻易解决为止。

我们的观点是，任何大问题，只要分拆方法正确，都不过是一系列有待解决的小问题的集合。这一章介绍的就是如何把大的架构问题分拆成小问题，用较少的工作实现同样的结果。在很多案例中，该方法都减少了（而不是增加了）解决问题所必需的工作量，简化了架构和解决方案，最终得到了更具可扩展性的解决方案或平台。

本书各章所介绍的原则篇幅不一，复杂度也不同。有些普适的原则，适用于一个设计的多个方面。而有些原则只适合特定系统。

1.1　原则 1：不要过度设计

目的：防止设计中出现复杂的解决方案。

适用情形：适用于任何项目，所有大型的或复杂的系统和项目都应该采用该原则。

应用方式：让同行来检查解决方案是否好理解，抵制过度设计的强烈欲望。

应用理由：复杂的解决方案实施成本高，而且会产生大量长期成本。

要点：过度复杂的系统会限制扩展能力。简单的系统更容易维护和扩展，且成本更低。

维基百科解释说，过度设计分为两大类[1]。一类是指设计与实现超出了有用需求的产品。出于完整性的考虑，我们只简单地讨论一下这个问题。相对于第二类问题来说，这类问题对可扩展性的影响较小。过度设计的另一类问题指过于复杂的产品。如前所述，我们最关心的是第二类问题对可扩展性的影响。不过，还是先来了解一下第一个问题吧。

要解释过度设计的第一类问题，即超出产品有用需求的问题，就要先搞清楚“有用的”这个术语的含义，这个术语在这里表示的只是“能够使用”。例如，为家庭住房设计一种空调，能够在室外温度为 0 开时把整个房子的温度加热到 300 华氏度，这毫无意义，纯属浪费，我们只需要一个能够在室外温度为-20 华氏度时把房子加热到舒适温度的产品。这种过度设计会产生过度的成本，其中开发的成本会更高，实施该方案的硬件和软件成本也会更高。如果研发这种过度设计系统的时间比研发有用系统的时间更长，还可能拖延产品的发布，对公司造成进一步的影响。成本高，利润就低。研发时间长，收入或收益就会被延迟，所有这些成本都会影响到利益相关者。范围蔓延，或者最初的产品定义和最初

的产品发布之间的范围差异，是过度设计的一种表现。

说个更接近我们工作的例子，是开发一个员工打卡系统，这个系统能够处理的员工数量是整个地球上人数的 100 倍。在这个软件的使用期限内，地球上的人口升至 100 倍的可能性是微乎其微的，而所有人都为一家公司工作的可能性则更小。我们当然想让构建的系统满足客户需求，但也不想浪费时间来实现和部署远远超出需求的系统。

过度设计的第二类表现是使系统过度复杂，或者用复杂的方式来实现它。简而言之，就是要花费过大的力气去完成一项工作，或者是让用户花费过大的力气去完成一项任务，或者是让程序员花费过大的力气去理解一个功能。让我们来逐一分析过度复杂的系统的这三种情况。

什么是花费过大的力气去完成一项工作呢？现实世界有最简单的例子。假设你让某人去杂货店买东西，你告诉他，店里面的所有商品都拿一个，排队结账时给你打电话。等他打电话给你时，你再告诉他到底想要哪几个，让他从所拿的无数篮商品中选出来，然后把其他商品都倒在地上。你一定会说："别开玩笑了。"可是，你在自己的代码中用过 `select (*) schema_name.table_name` 这样的 SQL 语句，只是为了从返回的集合中找出自己想要的结果吗（参见原则 35）？我们这个杂货店的例子，和上述的 `select(*)` 正是异曲同工。在你的代码中，有几个条件语句是处理个别情况的，它们是按照什么顺序执行的？是不是最可能发生的情况最先执行？你是不是经常刚查询完一个结果，又重复查询一次？是不是经常刚显示了一个 HTML 页面，又重新创建它？这种情况（反复做一项工作）随处可见，却又经常被忽视，我们将专门用一章（第 6 章）来讨论这个主题。

什么是让一位用户花费过大的力气去完成一项任务呢？答案非常简单。在许多情况下，少就是多。为追求系统的灵活性，我们总是想给它硬加上尽可能多的奇怪功能。但生活的情趣并不总在于多种多样。许多时候，用户只是想无干扰地尽可能快地从 A 到达 B。如果你的市场中有

99%的用户不需要把日志文件存成.pdf文件，那么就不要构建一个提示框询问他们是否想把日志文件保存成.pdf 文件。如果你的用户想把.wav 文件转换成 MP3 文件，那么他们已经不在乎损失精度了，所以不必再提示他们转换成无损压缩的 FLAC 文件，那样只会干扰他们。

最后一种情况，就是软件复杂得让其他程序员难以理解。创建复杂的代码让他人难以理解曾经非常流行（还有过比赛）。有时，代码写得复杂，是为了让它比一般程序员所开发的代码运行更快。而更多的情况是，代码的复杂度（就其理解的难度而言）成了程序员才华的象征，或者说是功夫高低的象征。那些开发的代码能让做代码检查的高级开发人员欲苦无泪的人反而颇受推崇。复杂度成了智慧的牢笼，编程极客们会在公司内部争强好胜。对于乐此不疲的人来说，这是很好的比赛，但对于公司和股东来说，则要为一场无人关心的牢笼大赛买单。对于那些仍然沉浸于这场极客盛宴的人，如果不想损害利益相关者的利益，又想真刀真枪地拼一场，那建议你参加国际混淆 C 代码竞赛，网址是 www0.us.ioccc.org/main.html。

我们都应该努力去写让每个人都能理解的代码。衡量一个伟大程序员的真正标准，是他能够多快把一个复杂的问题简化（见原则 3），多快能开发出一个既容易理解，又容易维护的解决方案。容易执行的解决方案意味着一般程序员就可以快速地掌握系统，为它提供支持。容易理解的解决方案则意味着在查找问题时能够更快地发现问题，从而以更快的方式把系统恢复到正常工作状态。容易执行的解决方案可以提高公司和解决方案的可扩展性。

要测试系统是否太复杂，一个很好的方法是让负责解决复杂问题的程序员把他的解决方案陈述给公司内的一组程序员。这组程序员应该代表公司内不同的编码水平，不同的工作年限（加入这一条，是因为可能有些有经验的程序员在公司的工作经验不多）。要通过这一测试，需要这组程序员中的每一位都能够轻松理解该解决方案，能够在无帮助的情况