

**Designing Data-Intensive**

**Applications**

*The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*

**数据密集应用系统设计**

*高可用，易扩展，好运维系统背后的思想*



Beijing  Boston  Farnham  Sebastopol  Tokyo

*Martin Kleppmann*

**Designing Data-Intensive Applications**

by Martin Kleppmann Copyright © 2017 Martin Kleppmann. All rights reserved. Printed in the United States of America. Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/insti‐ tutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editors:** Ann Spencer and Marie Beaugureau **Production Editor:** Kristen Brown **Copyeditor:** Rachel Head **Proofreader:** Amanda Kersey

March 2017: First Edition

**Revision History for the First Edition**

**Indexer:** Ellen Troutman-Zaig **Interior Designer:** David Futato **Cover Designer:** Karen Montgomery **Illustrator:** Rebecca Demarest

2017-03-01: First Release See *http://oreilly.com/catalog/errata.csp?isbn=9781449373320* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Designing Data-Intensive Applications*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-449-37332-0 [LSI]

*Technology is a powerful force in our society. Data, software, and communication can*

*be used for bad: to entrench unfair power structures, to undermine human rights, and to protect vested interests. But they can also be used for good: to make underrepresented people’s voices heard, to create opportunities for everyone, and to avert disasters. This book is dedicated to everyone working toward the good.*

*技术就是力量。技术可以用干坏事：加剧社会中的不公，妨碍一些人的权利，保护既得利益。同时，技术也可以干好事：让低微人的声音得以被倾听，为每个人创造机会，病免一些灾难。本书致力于让每个人都把技术用于好的方向。*

*Computing is pop culture. [...] Pop culture holds a disdain for history. Pop culture is all about identity and feeling like you’re participating. It has nothing to do with cooperation, the past or the future—it’s living in the present. I think the same is true of most people who write code for money. They have no idea where [their culture came from].*

—Alan Kay, in interview with *Dr Dobb’s Journal* (2012)

计算机是一种流行文化。流行文化曾经被人歧视。流行文化在于发现自我，享受参与的乐趣。它与过去、将来、他人无关，它活在当下。我相信，这和对于仅仅为了钱写代码的人一样，他们根本不懂背后的原理。（译者注：如果你是一个有追求的程序员，就应该探究其背后的工作原理。）

**Preface. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii**

**Part I. Foundations of Data Systems 1. Reliable, Scalable, and Maintainable Applications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3**

第一部分。数据系统基础。1. 高可用，易扩展，好运维应用

Thinking About Data Systems

关于数据系统的思考

4 6 7 8 9

Reliability（可靠性） Scalability（扩展性） 10 Describing Load （负载描述）11 Describing Performance（性能描述） 13 Approaches for Coping with Load （负载处理方法）17

Maintainability （运维性）18 Operability: Making Life Easy for Operations（可操作性：让生活更容易处理） 19 Simplicity: Managing Complexity （简洁性：管理复杂度）20 Evolvability: Making Change Easy（可扩展性：容易修改） 21

Summary 22

**2. Data Models and Query Languages. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27**

**数据模型和查询语言**

Relational Model Versus Document Model （关系模型vs文档模型）28 The Birth of NoSQL （NoSQL诞生）29 The Object-Relational Mismatch （对象-关系模型对比）29 Many-to-One and Many-to-Many Relationships （多对一和多对多模型）33 Are Document Databases Repeating History? （对象模型是历史的重现吗？）36

**Table of Contents**

**vii**

Relational Versus Document Databases Today（当前关系模型、对象模型对比） 38 Query Languages for Data （数据查询语言）42 Declarative Queries on the Web （Web上声明式查询）44 MapReduce Querying 46 Graph-Like Data Models （图模型）49 Property Graphs（属性图） 50 The Cypher Query Language（Cypher查询语言） 52 Graph Queries in SQL （用SQL实现图查询）53 Triple-Stores and SPARQL 55 The Foundation: Datalog 60 Summary 63

**3. Storage and Retrieval. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69**

70

72

76

79

83

85

90

91

93

95

97

99

Summary 103

**4. Encoding and Evolution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111** Formats for Encoding Data 112 Language-Specific Formats 113 JSON, XML, and Binary Variants 114 Thrift and Protocol Buffers 117 Avro 122 The Merits of Schemas 127 Modes of Dataflow 128 Dataflow Through Databases 129 Dataflow Through Services: REST and RPC 131 Message-Passing Dataflow 136 Summary 139

Data Structures That Power Your Database Hash Indexes SSTables and LSM-Trees B-Trees

Comparing B-Trees and LSM-Trees

Other Indexing Structures Transaction Processing or Analytics?

Data Warehousing

Stars and Snowflakes: Schemas for Analytics Column-Oriented Storage

Column Compression Sort Order in Column Storage Writing to Column-Oriented Storage 101 Aggregation: Data Cubes and Materialized Views 101



**viii | Table of Contents**



**Part II. Distributed Data**

1. **Replication. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151 Leaders and Followers 152 Synchronous Versus Asynchronous Replication 153 Setting Up New Followers 155 Handling Node Outages 156 Implementation of Replication Logs 158 Problems with Replication Lag 161 Reading Your Own Writes 162 Monotonic Reads 164 Consistent Prefix Reads 165 Solutions for Replication Lag 167 Multi-Leader Replication 168 Use Cases for Multi-Leader Replication 168 Handling Write Conflicts 171 Multi-Leader Replication Topologies 175 Leaderless Replication 177 Writing to the Database When a Node Is Down 177 Limitations of Quorum Consistency 181 Sloppy Quorums and Hinted Handoff 183 Detecting Concurrent Writes 184 Summary 192**
2. **Partitioning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 199 Partitioning and Replication 200 Partitioning of Key-Value Data 201  Partitioning by Key Range 202 Partitioning by Hash of Key 203 Skewed Workloads and Relieving Hot Spots 205  Partitioning and Secondary Indexes 206 Partitioning Secondary Indexes by Document 206 Partitioning Secondary Indexes by Term 208  Rebalancing Partitions 209 Strategies for Rebalancing 210 Operations: Automatic or Manual Rebalancing 213  Request Routing 214 Parallel Query Execution 216 Summary 216**
3. **Transactions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 221 The Slippery Concept of a Transaction 222**



**Table of Contents | ix**

The Meaning of ACID 223

Single-Object and Multi-Object Operations 228 Weak Isolation Levels 233 Read Committed 234 Snapshot Isolation and Repeatable Read 237 Preventing Lost Updates 242 Write Skew and Phantoms 246 Serializability 251 Actual Serial Execution 252 Two-Phase Locking (2PL) 257 Serializable Snapshot Isolation (SSI) 261 Summary 266

**8. The Trouble with Distributed Systems. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 273** Faults and Partial Failures 274 Cloud Computing and Supercomputing 275 Unreliable Networks 277 Network Faults in Practice 279 Detecting Faults 280 Timeouts and Unbounded Delays 281 Synchronous Versus Asynchronous Networks 284 Unreliable Clocks 287 Monotonic Versus Time-of-Day Clocks 288 Clock Synchronization and Accuracy 289 Relying on Synchronized Clocks 291 Process Pauses 295 Knowledge, Truth, and Lies 300 The Truth Is Defined by the Majority 300 Byzantine Faults 304 System Model and Reality 306 Summary 310

**9. Consistency and Consensus. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 321** Consistency Guarantees 322 Linearizability 324

What Makes a System Linearizable? 325 Relying on Linearizability 330 Implementing Linearizable Systems 332 The Cost of Linearizability 335

Ordering Guarantees 339 Ordering and Causality 339 Sequence Number Ordering 343



**x | Table of Contents**

Total Order Broadcast 348 Distributed Transactions and Consensus 352 Atomic Commit and Two-Phase Commit (2PC) 354 Distributed Transactions in Practice 360 Fault-Tolerant Consensus 364 Membership and Coordination Services 370 Summary 373

**Part III. Derived Data**

1. **Batch Processing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 389 Batch Processing with Unix Tools 391 Simple Log Analysis 391 The Unix Philosophy 394 MapReduce and Distributed Filesystems 397 MapReduce Job Execution 399 Reduce-Side Joins and Grouping 403 Map-Side Joins 408 The Output of Batch Workflows 411 Comparing Hadoop to Distributed Databases 414 Beyond MapReduce 419 Materialization of Intermediate State 419 Graphs and Iterative Processing 424 High-Level APIs and Languages 426 Summary 429**
2. **Stream Processing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 439 Transmitting Event Streams 440 Messaging Systems 441 Partitioned Logs 446 Databases and Streams 451 Keeping Systems in Sync 452 Change Data Capture 454 Event Sourcing 457 State, Streams, and Immutability 459 Processing Streams 464 Uses of Stream Processing 465 Reasoning About Time 468 Stream Joins 472 Fault Tolerance 476 Summary 479**

**Table of Contents | xi**

**12. The Future of Data Systems. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 489** Data Integration 490 Combining Specialized Tools by Deriving Data 490 Batch and Stream Processing 494 Unbundling Databases 499 Composing Data Storage Technologies 499 Designing Applications Around Dataflow 504 Observing Derived State 509 Aiming for Correctness 515 The End-to-End Argument for Databases 516 Enforcing Constraints 521 Timeliness and Integrity 524 Trust, but Verify 528 Doing the Right Thing 533 Predictive Analytics 533 Privacy and Tracking 536 Summary 543

**Glossary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 553 Index. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 559**



**xii | Table of Contents**

**Preface**

If you have worked in software engineering in recent years, especially in server-side and backend systems, you have probably been bombarded with a plethora of buzz‐ words relating to storage and processing of data. NoSQL! Big Data! Web-scale! Sharding! Eventual consistency! ACID! CAP theorem! Cloud services! MapReduce! Real-time!

近几年如果你的工作与软件相关，尤其是服务端和后端系统方面，你可能已经听到了太多的关于数据存储和处理的五花八门的词汇。NoSQL！大数据！Web-scale!分片！最终一致性！ACID！CAP理论！云服务！MapReduce!实时性！

In the last decade we have seen many interesting developments in databases, in distributed systems, and in the ways we build applications on top of them. There are various driving forces for these developments:

过去的几十年，我们见证了数据库、分布式系统领域以及构建在这些系统上应用系统的可喜进步。原因有以下几点：

* Internet companies such as Google, Yahoo!, Amazon, Facebook, LinkedIn, Microsoft, and Twitter are handling huge volumes of data and traffic, forcing them to create new tools that enable them to efficiently handle such scale.
* 例如Google, Yahoo!, Amazon, Facebook, LinkedIn, Microsoft, and Twitter这些公司需要处理大量的数据及其访问，迫使他们发明各种有效处理大规模数据的工具
* Businesses need to be agile, test hypotheses cheaply, and respond quickly to new market insights by keeping development cycles short and data models flexible.
* 现代商业模式需要敏捷开发， 迅速对市场变化做出相应，快速试错。这要求开发周期要短，数据模型必须灵活。
* Free and open source software has become very successful and is now preferred to commercial or bespoke in-house software in many environments.
* 免费和开源软件的大获成功。许多场景中需要商业化或者订制化的软件服务
* CPU clock speeds are barely increasing, but multi-core processors are standard, and networks are getting faster. This means parallelism is only going to increase.
* CPU时钟周期几乎不再提高，多核处理器已经普及，网络变得更快。这意味着并行化正在成为趋势。
* Even if you work on a small team, you can now build systems that are distributed across many machines and even multiple geographic regions, thanks to infra‐ structure as a service (IaaS) such as Amazon Web Services.
* 即使你在一个小团队，你也可以借助例如亚马逊的Web Services基础服务构建多机至多地域的系统。
* Many services are now expected to be highly available; extended downtime due to outages or maintenance is becoming increasingly unacceptable.  *Data-intensive applications* are pushing the boundaries of what is possible by making use of these technological developments. We call an application *data-intensive* if data is its primary challenge—the quantity of data, the complexity of data, or the speed at  which it is changing—as opposed to *compute-intensive*, where CPU cycles are the bottleneck.
* 许多服务需要是高可用的。因为电网或者运维原因导致的宕机变得越来越不可接受。数据密集型应用正在利用技术的发展拓展自己服务的边界。数据密集型应用的主要针对-大量数据，复杂数据，或者数据快速变化的场景。与其相对的是CPU为瓶颈的CPU密集型应用。

The tools and technologies that help data-intensive applications store and process data have been rapidly adapting to these changes. New types of database systems (“NoSQL”) have been getting lots of attention, but message queues, caches, search indexes, frameworks for batch and stream processing, and related technologies are very important too. Many applications use some combination of these.

针对数据密集型应用的数据存储和处理技术和工具正在快速发展。新型数据库系统（NoSQL）已经走进大众视线，同时消息队列，caches，检索索引，批处理和流处理架构及其相关技术也变得日益重要。很多应用都利用到以上多项技术。

The buzzwords that fill this space are a sign of enthusiasm for the new possibilities, which is a great thing. However, as software engineers and architects, we also need to have a technically accurate and precise understanding of the various technologies and their trade-offs if we want to build good applications. For that understanding, we have to dig deeper than buzzwords.

这些流行词汇预示着无限的可能性。同时这又程序员和架构师要有关于这些技术的理解、积累以及利用其构建应用要做出的取舍。从这个角度说，我们要深刻理解而不仅仅浮于词汇表面。

Fortunately, behind the rapid changes in technology, there are enduring principles that remain true, no matter which version of a particular tool you are using. If you understand those principles, you’re in a position to see where each tool fits in, how to make good use of it, and how to avoid its pitfalls. That’s where this book comes in.

幸运的是，在快速发展的技术、以及各个版本工具的背后，有一些共同的规律。如果你抓住了这些规律，你就能知道每个工具的适用场景，它擅长什么，怎么避免它的短板。这就是本书的目标。

The goal of this book is to help you navigate the diverse and fast-changing landscape of technologies for processing and storing data. This book is not a tutorial for one particular tool, nor is it a textbook full of dry theory. Instead, we will look at examples of successful data systems: technologies that form the foundation of many popular applications and that have to meet scalability, performance, and reliability require‐ ments in production every day.

本书目标就是帮忙你浏览纷乱而又快速发展的各种数据处理&存储技术的边界。本书不是一个各种工具的使用手册，也不是各种理论的干货。我们将会结合各种流行系统来介绍其底层技术如何满足高扩展，高性能，高可用的要求的。

We will dig into the internals of those systems, tease apart their key algorithms, discuss their principles and the trade-offs they have to make. On this journey, we will try to find useful ways of *thinking about* data systems—not just *how* they work, but also *why* they work that way, and what questions we need to ask.

我们将会深入系统内部，抽丝剥茧，细究核心算法，详解他们做出各种取舍的准则。在这个过程中，我们将会形成自己对应数据系统的理解-不仅仅是它们的工作原理，还有它们为什么这样工作以及我们应该从什么角度去思考问题。

After reading this book, you will be in a great position to decide which kind of technology is appropriate for which purpose, and understand how tools can be combined to form the foundation of a good application architecture. You won’t be ready to build your own database storage engine from scratch, but fortunately that is rarely necessary. You will, however, develop a good intuition for what your systems are doing under the hood so that you can reason about their behavior, make good design decisions, and track down any problems that may arise.

读完本书后，你能知道针对于你的系统做出自己的技术选型，理解这些工具如何组成好的应用架构。大多数情况下，你可能不必真的从零到一的构建自己的数据存储引擎。但是你却能形成一个好的应用系统的直觉，它能帮助你理解系统背后的行为准则，做出好的设计判断，更方便地追踪系统异常。

**Who Should Read This Book?**

**谁应该读本书**

If you develop applications that have some kind of server/backend for storing or pro‐ cessing data, and your applications use the internet (e.g., web applications, mobile apps, or internet-connected sensors), then this book is for you.

如果你正在开发一个互联网应用的数据存储或者处理服务端/后端，这本书就是为你准备的。



**xiv | Preface**

This book is for software engineers, software architects, and technical managers who love to code. It is especially relevant if you need to make decisions about the architecture of the systems you work on—for example, if you need to choose tools for solving a given problem and figure out how best to apply them. But even if you have no choice over your tools, this book will help you better understand their strengths and weaknesses.

如果你是软件工程师，架构师或者对代码感兴趣的技术负责人都可以读此书。尤其是你需要对系统架构设计做决策时更需要这本书，例如，你需要为一个特定问题选择一个工具并衡量起效果。即使你还没想好选用哪个工具，本书也会帮你更好理解各个选择的利害。

You should have some experience building web-based applications or network services, and you should be familiar with relational databases and SQL. Any non- relational databases and other data-related tools you know are a bonus, but not required. A general understanding of common network protocols like TCP and HTTP is helpful. Your choice of programming language or framework makes no difference for this book.

你可能

If any of the following are true for you, you’ll find this book valuable:

如果你符合以下任意一点，你会发现本书对你来说很有价值：

* You want to learn how to make data systems scalable, for example, to support web or mobile apps with millions of users.
* 你想学习如何设计一个易扩展的支持数百万用户的web/移动app数据系统
* You need to make applications highly available (minimizing downtime) and operationally robust.
* 你需要设计一个高可用（宕机时间短）和易操作的系统。
* You are looking for ways of making systems easier to maintain in the long run, even as they grow and as requirements and technologies change.
* 你正在苦苦思索：从长远来看，随着需求和技术的不断变化，如何设计一个易维护的系统。
* You have a natural curiosity for the way things work and want to know what goes on inside major websites and online services. This book breaks down the internals of various databases and data processing systems, and it’s great fun to explore the bright thinking that went into their design.  Sometimes, when discussing scalable data systems, people make comments along the lines of, “You’re not Google or Amazon. Stop worrying about scale and just use a relational database.” There is truth in that statement: building for scale that you don’t need is wasted effort and may lock you into an inflexible design. In effect, it is a form of premature optimization. However, it’s also important to choose the right tool for the job, and different technologies each have their own strengths and weaknesses. As we shall see, relational databases are important but not the final word on dealing with data.
* 自然的好奇心驱使你去探究各种网络服务和在线应用的工作原理。本书将各种数据库系统和数据处理系统打散开来逐点分析，从设计者的角度来思考是一件很有趣的事情。当我们讨论大规模数据系统的时候经常有人泼冷水“你们公司又不是Google 或者Amazon，别杞人忧天，考虑什么扩展性，关系型数据库已经够用了。”这个假设有个前提：为了大规模做出的设计和妥协是以损失灵活性为代价的。实际上，这是某种程度上的过度设计。但是，用合适的工具解决合适的问题，每种技术都有其优略点。我们应该知道，虽然关系型数据库很重要，但它不是万能的。
* **Scope of This Book**
* **本书边界**
* This book does not attempt to give detailed instructions on how to install or use specific software packages or APIs, since there is already plenty of documentation for those things. Instead we discuss the various principles and trade-offs that are fundamental to data systems, and we explore the different design decisions taken by different products.
* 本书因为网络上各种软件包和api的安装使用很多了，本书不会涉及。相反，我们会详细分析数据系统背后的原理和取舍，以及不同系统做出的不同选择的原因。



**Preface | xv**

In the ebook editions we have included links to the full text of online resources. All links were verified at the time of publication, but unfortunately links tend to break frequently due to the nature of the web. If you come across a broken link, or if you are reading a print copy of this book, you can look up references using a search engine. For academic papers, you can search for the title in Google Scholar to find open-access PDF files. Alternatively, you can find all of the references at *https:// github.com/ept/ddia-references*, where we maintain up-to-date links.

在本书的电子版中我们有在线资源的所有链接。所有链接出版前都进行了校对，都是有效的。不过有些链接一定会随着时间流逝而失效。如果你遇到无效链接或者正在读纸质版图书，你可以搜索引擎自己查找文章。针对学术论文，你可以通过论文题目，在google学术搜索上找到pdf文件。你可以可以访问*https:// github.com/ept/ddia-references*，它会保证一直更新

We look primarily at the *architecture* of data systems and the ways they are integrated into data-intensive applications. This book doesn’t have space to cover deployment, operations, security, management, and other areas—those are complex and important topics, and we wouldn’t do them justice by making them superficial side notes in this book. They deserve books of their own.

我们主要研究数据系统的架构和他们在数据密集系统中的整合方式。本书无力涉及部署、操作、安全、管理和其它问题，它们各自都是既复杂又重要的专题 。肤浅的介绍不如不做。

Many of the technologies described in this book fall within the realm of the *Big Data* buzzword. However, the term “Big Data” is so overused and underdefined that it is not useful in a serious engineering discussion. This book uses less ambiguous terms, such as single-node versus distributed systems, or online/interactive versus offline/ batch processing systems.

本书描述的很多技术都是大数据领域的流行词汇。但是“大数据”这个词在通俗领域被用烂了，因此在严肃工程领域失去了精确的定义。本书会尽量少用这种模糊词汇，例如：“单点”对应“分布式系统”、“在线/交互”对应“离线/批量处理”系统

This book has a bias toward free and open source software (FOSS), because reading, modifying, and executing source code is a great way to understand how something works in detail. Open platforms also reduce the risk of vendor lock-in. However, where appropriate, we also discuss proprietary software (closed-source software, soft‐ ware as a service, or companies’ in-house software that is only described in literature but not released publicly).

本书倾向于免费开源软件，因为读、改、执行源码是最了解系统背后工作细节的最好的方法。开放平台中途被抛弃的风险也小一些。但是，如果有必要，我们也会讨论一些有版权系统（闭源软件，软件就是服务，或者没有公开发布只在文档中涉及的公司内部软件）

**Outline of This Book**

**本书结构**

This book is arranged into three parts:

本书分三个部分

1. In Part I, we discuss the fundamental ideas that underpin the design of data- intensive applications. We start in Chapter 1 by discussing what we’re actually trying to achieve: reliability, scalability, and maintainability; how we need to think about them; and how we can achieve them. In Chapter 2 we compare several different data models and query languages, and see how they are appropriate to different situations. In Chapter 3 we talk about storage engines: how databases arrange data on disk so that we can find it again efficiently. Chapter 4 turns to formats for data encoding (serialization) and evolution of schemas over time.

第一部分：我们讲述数据密集系统之下的基本思路和准则。第一章，我们介绍我们要达到的终极目标：高可用，易扩展，易运维；针对每个点我们如果去思考；如何才能达到各个目标。第二章,我们将通过不同的数据模型和查询语言对比来说明它们各种适用于何种场景。第三章，我们介绍存储引擎：数据库系统的数据在磁盘上的组织方式及如何才能高效的访问。第四章，说明一下数据编码方式（序列化）和随着时间推移，schemas的变化。

1. In Part II, we move from data stored on one machine to data that is distributed across multiple machines. This is often necessary for scalability, but brings with it a variety of unique challenges. We first discuss replication (Chapter 5), parti‐ tioning/sharding (Chapter 6), and transactions (Chapter 7). We then go into more detail on the problems with distributed systems (Chapter 8) and what it means to achieve consistency and consensus in a distributed system (Chapter 9).

第二部分：我们把数据载体从单机扩展到多机。这通常是为了满足扩展性的需求，但这也带来了很多前所未有的挑战。我们先在第五章讨论副本机制，第六章讨论分片机制，第七章讨论事务。然后我们在第八章探讨分布式系统中更细节信息。第九章会说明在分布式系统中的一贯性和一致性。

1. In Part III, we discuss systems that derive some datasets from other datasets. Derived data often occurs in heterogeneous systems: when there is no one database that can do everything well, applications need to integrate several different databases, caches, indexes, and so on. In Chapter 10 we start with a batch processing approach to derived data, and we build upon it with stream processing in Chapter 11. Finally, in Chapter 12 we put everything together and discuss approaches for building reliable, scalable, and maintainable applications in the future.

第三部分：我们介绍具有数据上下游关系的一系列数据系统。系统间数据交互经常发生：当一个数据库在某一方面有缺陷，同时应用需要把很多不同的数据库,caches,索引和其它系统整合在一起。第十章：我们介绍接收数据的批量处理系统，它的数据来源是第十一章介绍的流式处理系统。最后在十二章：我们把所有的系统放在一起讨论，将来如何才能构建一个高可用，易扩展，方便运维的应用。

**References and Further Reading**

参考文献和推荐阅读

Most of what we discuss in this book has already been said elsewhere in some form or another—in conference presentations, research papers, blog posts, code, bug trackers, mailing lists, and engineering folklore. This book summarizes the most important ideas from many different sources, and it includes pointers to the original literature throughout the text. The references at the end of each chapter are a great resource if you want to explore an area in more depth, and most of them are freely available online.

本书讨论的大部分内容在其他地方（一些会议，研究报告，blog，源码中，bug跟踪，邮件列表和程序员读物）已经有所介绍。本书只是将从各种来源的思路进行汇总，本文保留到原文的引用。每章尾的引用都是更深了解各个原理的很好的资源，一般都是免费在线阅读的。

**O’Reilly Safari**

Members have access to thousands of books, training videos, Learning Paths, interac‐ tive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Pro‐ fessional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

*Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

**Preface | xvii**

**How to Contact Us**

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 800-998-9938 (in the United States or Canada) 707-829-0515 (international or local) 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/designing-data-intensive-apps*.

To comment or ask technical questions about this book, send email to *bookques‐ tions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our web‐ site at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*Follow us on Twitter: *http://twitter.com/oreillymedia*Watch us on YouTube: *http://www.youtube.com/oreillymedia*

**Acknowledgments**

**鸣谢**

This book is an amalgamation and systematization of a large number of other people’s ideas and knowledge, combining experience from both academic research and industrial practice. In computing we tend to be attracted to things that are new and shiny, but I think we have a huge amount to learn from things that have been done before. This book has over 800 references to articles, blog posts, talks, documentation, and more, and they have been an invaluable learning resource for me. I am very grateful to the authors of this material for sharing their knowledge.

本书是大量学术研究和工业实践的思想系统化思路结晶。相对于提出新的思路，我们更乐于再前人经验上进行思考。本书引用了800多处文献，blog，访谈，文档，这都是无价的学习资源。非常感谢将这些知识分享的作者。

I have also learned a lot from personal conversations, thanks to a large number of people who have taken the time to discuss ideas or patiently explain things to me. In particular, I would like to thank Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Hel‐ land, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O’Neil, Onora O’Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, and Brett Wooldridge.

谢谢那些和我耐心讨论并解释给我听的人，这些会话也让我获益匪浅。尤其是以下各位：Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Hel‐ land, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O’Neil, Onora O’Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, and Brett Wooldridge.

Several more people have been invaluable to the writing of this book by reviewing drafts and providing feedback. For these contributions I am particularly indebted to Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, and Ben Summers. Of course, I take all responsibility for any remaining errors or unpalatable opinions in this book.

还有很多帮我校稿提供建议的人。我非常感谢以下人Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, and Ben Summers.当然，本书遗留的错误和纰漏仍是我的错误。

For helping this book become real, and for their patience with my slow writing and unusual requests, I am grateful to my editors Marie Beaugureau, Mike Loukides, Ann Spencer, and all the team at O’Reilly. For helping find the right words, I thank Rachel Head. For giving me the time and freedom to write in spite of other work commitments, I thank Alastair Beresford, Susan Goodhue, Neha Narkhede, and Kevin Scott.

非常感谢我的编辑谢谢Marie Beaugureau, Mike Loukides, Ann Spencer,和O’Reilly的整个团队，他们耐心等我写完此书，并不厌其烦的解答我的疑问。Rachel Head帮助我找到合适的词汇。感谢Alastair Beresford, Susan Goodhue, Neha Narkhede, and Kevin Scott，他们跟我写作的时间和无限制的自由。

Very special thanks are due to Shabbir Diwan and Edie Freedman, who illustrated with great care the maps that accompany the chapters. It’s wonderful that they took on the unconventional idea of creating maps, and made them so beautiful and compelling.

非常感谢Shabbir Diwan and Edie Freedman，他们为每章提供了插图。它们让本书变得有趣。

Finally, my love goes to my family and friends, without whom I would not have been able to get through this writing process that has taken almost four years. You’re the best.

最后，感谢我的家人和朋友，没有他们我不几乎不能为此书坚持四年。



**Preface | xix**

**PART I Foundations of Data Systems**

**第一部分：数据系统基础**



The first four chapters go through the fundamental ideas that apply to all data systems, whether running on a single machine or distributed across a cluster of machines:

前四章我们会简单介绍无论是单机还是分布式数据系统的基础知识。

1. Chapter 1 introduces the terminology and approach that we’re going to use throughout this book. It examines what we actually mean by words like *reliability*, *scalability*, and *maintainability*, and how we can try to achieve these goals.

第一章：介绍本书所用的专业术语。包括高可用性，易扩展性，易运维性以及我们如何努力达成以上目标

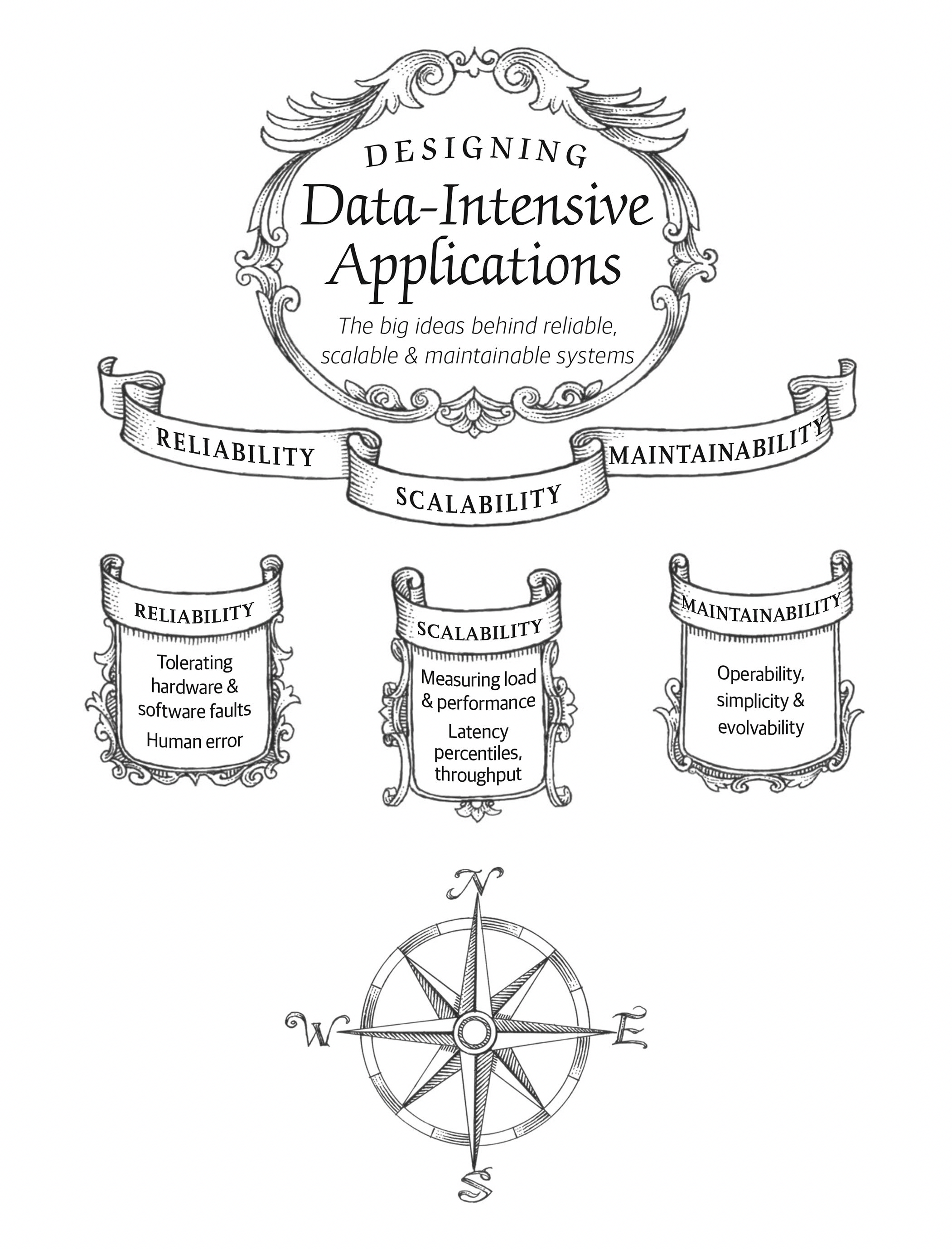
1. Chapter 2 compares several different data models and query languages—the most visible distinguishing factor between databases from a developer’s point of view. We will see how different models are appropriate to different situations.

第二章：以开发者视角，针对几种不通数据模型和查询语言的对比来说明他们的关键区别。我们会知道不同模型的适用场景。

3. Chapter 3 turns to the internals of storage engines and looks at how databases lay out data on disk. Different storage engines are optimized for different workloads, and choosing the right one can have a huge effect on performance.

4. Chapter 4 compares various formats for data encoding (serialization) and espe‐ cially examines how they fare in an environment where application requirements change and schemas need to adapt over time.

Later, Part II will turn to the particular issues of distributed data systems.



**CHAPTER 1**

**Reliable, Scalable, and Maintainable Applications**

**高可靠、易扩展、易运维应用**



*The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free?*

*Internet这么易用，已至很多人认为它是向太平洋一样的自然资源，从来不会想到它是一个人造物。上一次如此完美无瑕的大规模技术应用是什么时候？*

—Alan Kay, in interview with *Dr Dobb’s Journal* (2012)

Many applications today are *data-intensive*, as opposed to *compute-intensive*. Raw CPU power is rarely a limiting factor for these applications—bigger problems are usually the amount of data, the complexity of data, and the speed at which it is changing.

现今很多应用都是数据密集型应用（相对于计算密集型应用）。对这些应用来说，CPU能力很少成为瓶颈，多数情况下数据规模、数据复杂度和数据变化速度是我们面对的主要问题。

A data-intensive application is typically built from standard building blocks that provide commonly needed functionality. For example, many applications need to:

数据密集型应用一般来说都是由一些提供普通功能的标准模块组成。比如，很多应用需要一下功能：

* Store data so that they, or another application, can find it again later (*databases*)
* 存储数据，其它的应用其后可以去除这些数据（数据库）
* Remember the result of an expensive operation, to speed up reads (*caches*)
* 为了加速读，临时存储一些需要耗费大量操作资源的结果（caches）
* Allow users to search data by keyword or filter it in various ways (*search indexes*)
* 允许用户通过一些关键词检索或者过滤等各种操作（检索索引）
* Send a message to another process, to be handled asynchronously (*stream processing*)
* 将一条信息发送到另一个异步处理的流程（流式处理）
* Periodically crunch a large amount of accumulated data (*batch processing*)
* 周期性的处理积累的大量数据（批量处理）。

 If that sounds painfully obvious, that’s just because these *data systems* are such a successful abstraction: we use them all the time without thinking too much. When build‐ ing an application, most engineers wouldn’t dream of writing a new data storage engine from scratch, because databases are a perfectly good tool for the job.

为啥这么令人费解？因为数据系统这个曾经的抽象太成功了：我们一直都在使用，但是从来没必要细想。当构建一个应用的时候，有很多数据库能很好的提供存储服务，大数据工程师都不会想从头构建一个存储引擎。



**3**

But reality is not that simple. There are many database systems with different characteristics, because different applications have different requirements. There are various approaches to caching, several ways of building search indexes, and so on. When building an application, we still need to figure out which tools and which approaches are the most appropriate for the task at hand. And it can be hard to combine tools when you need to do something that a single tool cannot do alone.

但是实际情况没这么简单。不同的应用有不同的需求，因此也就存在很多具有不同特性的数据库。它们有各自不同cache、组织索引、及其它实现同一功能不同的方法。当构建一个应用系统时，我们仍然需要弄明白哪个工具的特点更切合应用的场景。当单独一个工具无法满足需求的时候，结合不同的工具解决问题就没那么容易了。

This book is a journey through both the principles and the practicalities of data systems, and how you can use them to build data-intensive applications. We will explore what different tools have in common, what distinguishes them, and how they achieve their characteristics.

本书涵盖数据系统的特性和提供服务的可能性，以及如何利用这些特性构建一个数据密集系统。我们将会说明不同工具的共性和特性以及他们是如何设计来实现特性的。

In this chapter, we will start by exploring the fundamentals of what we are trying to achieve: reliable, scalable, and maintainable data systems. We’ll clarify what those things mean, outline some ways of thinking about them, and go over the basics that we will need for later chapters. In the following chapters we will continue layer by layer, looking at different design decisions that need to be considered when working on a data-intensive application.

本章，我们开始接触我们终极系统（高可用，易扩展，易运维的数据系统）的基础要求。我们先解释这些词汇具体含义，从这些维度思考的基本思路，后续的章节我们会详细介绍每个特性。接下来的章节，我们一层层抽丝剥茧，说明设计一个数据密集系统要做出设计折衷时的思维方式。

**Thinking About Data Systems**

**数据系统思考**

We typically think of databases, queues, caches, etc. as being very different categories of tools. Although a database and a message queue have some superficial similarity— both store data for some time—they have very different access patterns, which means different performance characteristics, and thus very different implementations.

我们一般任务数据库，队列，caches是非常不同的工具。虽然数据库和消息队列有些相似之处-都能存储数据，他们却是完全不同的工组模式，这意味着不同的性能表现，实现也是完全不通。

So why should we lump them all together under an umbrella term like *data systems*?

那为什么我们还把他们都放在数据系统这个大概念下一起讨论呢？

Many new tools for data storage and processing have emerged in recent years. They are optimized for a variety of different use cases, and they no longer neatly fit into traditional categories [1]. For example, there are datastores that are also used as message queues (Redis), and there are message queues with database-like durability guarantees (Apache Kafka). The boundaries between the categories are becoming blurred.

今些年出现了很多数据存储和处理的新系统。他们都针对不同的应用场景做过优化，同时也不再适合传统的分类方法。例如：存在很多被用来做消息队列的存储系统（Redis），也存在具有数据库持久存储特性的消息队列（Apache Kafka）。不同种类系统间的边界变的越来越模糊。

Secondly, increasingly many applications now have such demanding or wide-ranging requirements that a single tool can no longer meet all of its data processing and storage needs. Instead, the work is broken down into tasks that *can* be performed efficiently on a single tool, and those different tools are stitched together using application code.

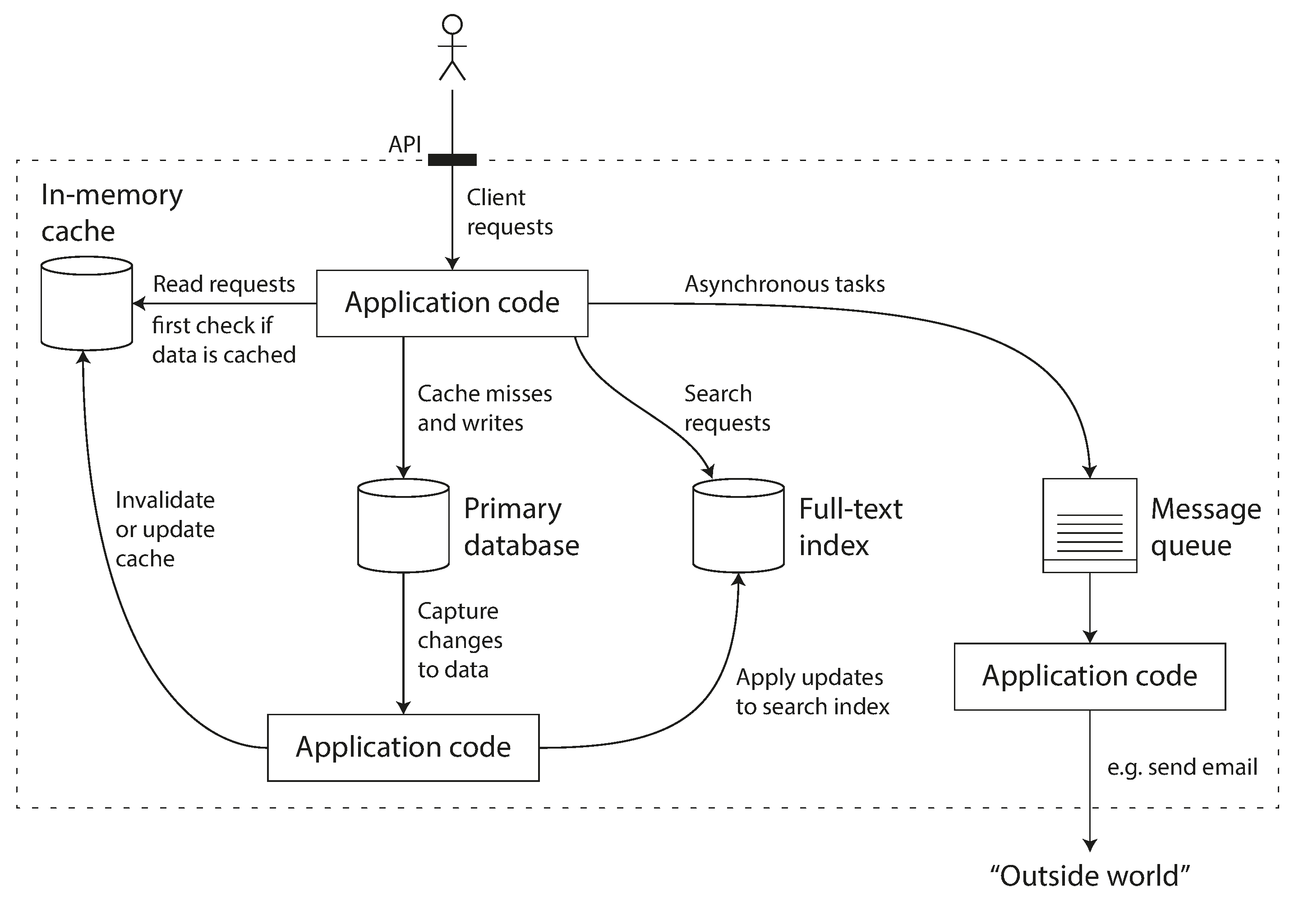
而且，越来越多的应用系统有了这些跨种类需求或者说，原有单一的工具已经不能满足所有的数据处理存储需求。相反，原有需求被分成多个任务的方式不如在一个单独工具内实现高效，同时多个工具的集成也需要应用层编码。

For example, if you have an application-managed caching layer (using Memcached or similar), or a full-text search server (such as Elasticsearch or Solr) separate from your main database, it is normally the application code’s responsibility to keep those caches and indexes in sync with the main database. Figure 1-1 gives a glimpse of what this may look like (we will go into detail in later chapters).

例如：你原来有一个应用层cache层（用Memcached或其类似组件），或者一个与主数据库分离的全文本检索服务（Elasticsearch或者Solr）。一般来说，应用层代码负责保持cache和索引与主数据库的同步。图1-1给出一种可能的实现方法（我们在后续章节给出实现细节）。



**4 | Chapter 1: Reliable, Scalable, and Maintainable Applications**



*Figure 1-1. One possible architecture for a data system that combines several components.*

*由多个模块构成的一个数据系统*

When you combine several tools in order to provide a service, the service’s interface or application programming interface (API) usually hides those implementation details from clients. Now you have essentially created a new, special-purpose data system from smaller, general-purpose components. Your composite data system may provide certain guarantees: e.g., that the cache will be correctly invalidated or updated on writes so that outside clients see consistent results. You are now not only an application developer, but also a data system designer.

当我们把几个工具集成成一个服务时，对应用者来说，工具的实现细节经常被服务或者应用的编程接口屏蔽。 现在，你从更小的通用组件开始构建一个全新，特定功能的数据系统。你组成的系统必须提供很多服务保证，比如，cache数据必须及时失效和更新，惟其如此，使用者才能看到前后一致的结果。你现在不只是一个应用开发者，也是一个系统设计者。

If you are designing a data system or service, a lot of tricky questions arise. How do you ensure that the data remains correct and complete, even when things go wrong internally? How do you provide consistently good performance to clients, even when parts of your system are degraded? How do you scale to handle an increase in load? What does a good API for the service look like?

当你设计数据系统或者服务的时候，会碰见各种奇怪的问题。你如何才能保证即使发生内部错误，数据依然是完整正确的呢？在系统的部分组件不能服务情况下，你如何才让终端用户享受到一致的高性能服务？你该怎么处理增长的请求？service什么样的API才是好的呢？

There are many factors that may influence the design of a data system, including the skills and experience of the people involved, legacy system dependencies, the timescale for delivery, your organization’s tolerance of different kinds of risk, regulatory constraints, etc. Those factors depend very much on the situation.

个人的技术积累，老系统遗留问题，开发周期，业务对于不通风险的容忍程度，监管限制等都会影响数据系统的设计。这些因素针对每个人大不相同。



**Thinking About Data Systems | 5**

In this book, we focus on three concerns that are important in most software systems:

本书，我们专注于软件系统中最重要的三个特性：

*Reliability*The system should continue to work *correctly* (performing the correct function at the desired level of performance) even in the face of *adversity* (hardware or soft‐ ware faults, and even human error). See “Reliability” on page 6.

可用性。即使遇到一些不正常的问题（软/硬件错误，人为错误），系统应该持续不断地正常工作（在一个预期性能指标下提供正确的功能）。See “Reliability” on page 6.

*Scalability*As the system *grows* (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth. See “Scalability” on page 10.

扩展性：随着系统规模（数据量，流量，复杂度）增加，应该很方便的处理增长带来的问题。See “Scalability” on page 10.

*Maintainability* Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it *productively*. See “Maintainabil‐ ity” on page 18.

运维性：在应用生命周期内，会有不同的人接触这个应用（程序员和应用使用者都会负责应用当前的正确运行，并为了满足新需求而进行改进），应用应该高效的满足这些需求。

These words are often cast around without a clear understanding of what they mean. In the interest of thoughtful engineering, we will spend the rest of this chapter exploring ways of thinking about reliability, scalability, and maintainability. Then, in the following chapters, we will look at various techniques, architectures, and algorithms that are used in order to achieve those goals.

这些词一般会直接使用。为了一些不知道这些概念的工程师，我们在本章剩下的部分将会尝试从多个角度思考以上三个特性。在后续章节，我们将讲述，哪些技术，架构，算法可以达到这些目标。

**Reliability**

Everybody has an intuitive idea of what it means for something to be reliable or unreliable. For software, typical expectations include:

每个人对于某物的可靠或者不可靠都有一个直觉的认识。对软件可靠性而言，经典的预期包括：

* The application performs the function that the user expected.
* 应用以用户预期的方式工作
* It can tolerate the user making mistakes or using the software in unexpected ways.
* 它能容忍用户的错误和一些非预期的操作。
* Its performance is good enough for the required use case, under the expected load and data volume.
* 在特定的负载和数据规模下，性能必须能满足应用需求。
* The system prevents any unauthorized access and abuse.  If all those things together mean “working correctly,” then we can understand *reliability* as meaning, roughly, “continuing to work correctly, even when things go wrong.”  The things that can go wrong are called *faults*, and systems that anticipate faults and can cope with them are called *fault-tolerant* or *resilient*. The former term is slightly misleading: it suggests that we could make a system tolerant of every possible kind of fault, which in reality is not feasible. If the entire planet Earth (and all servers on it) were swallowed by a black hole, tolerance of that fault would require web hosting in space—good luck getting that budget item approved. So it only makes sense to talk about tolerating *certain types* of faults.
* 系统必须访问未授权的操作。如果“正常工作”指的就是这些的话，可靠性大体可以理解为“即使有些东西不正常，但整个系统仍然能正常工作”。某些组件不能正常工作叫错误，系统能预料并处理可能的错误叫容错性。前面有个误导性的词语：它表名我们正在设计一个能处理任何可能出现错误的系统，其实这是不可能 的。如果整个地球（这个服务所有服务器都在地球上）都被黑洞吞了，针对这个错误的容错性需要将服务器部署到外太空（如果你能有足够的预算）。因此，只有针对特定类型的错误讲究容错性才是有意义的。

Note that a fault is not the same as a failure [2]. A fault is usually defined as one component of the system deviating from its spec, whereas a *failure* is when the system as a whole stops providing the required service to the user. It is impossible to reduce the probability of a fault to zero; therefore it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures. In this book we cover several techniques for building reliable systems from unreliable parts.

注意，错误和失败是两个概念。错误一般指系统某一组件不按照预期工作，失败指整个系统不能相应用户的请求。我们不可能把错误可能性减少为零，因此我们一般都尽量完善系统的容错机制，以防止因此导致的系统失败。本书我们将介绍一些根据不可靠组件构建可靠系统的技术方案。

Counterintuitively, in such fault-tolerant systems, it can make sense to *increase* the rate of faults by triggering them deliberately—for example, by randomly killing individual processes without warning. Many critical bugs are actually due to poor error handling [3]; by deliberately inducing faults, you ensure that the fault-tolerance machinery is continually exercised and tested, which can increase your confidence that faults will be handled correctly when they occur naturally. The Netflix *Chaos Monkey* [4] is an example of this approach.

与直觉不同，在容错系统中，通过主动触发增加错误出现频率是有意义的，比如无预警地随机杀掉一个进程。很多致命的bug都是因为针对某些错误的处理缺失引起的。手动触发错误能确保容错机制能确保不断的测试，你也会对系统更有信心，它能处理将来发生的错误。Netflix *Chaos Monkey系统就是一个这样的例子。*

Although we generally prefer tolerating faults over preventing faults, there are cases where prevention is better than cure (e.g., because no cure exists). This is the case with security matters, for example: if an attacker has compromised a system and gained access to sensitive data, that event cannot be undone. However, this book mostly deals with the kinds of faults that can be cured, as described in the following sections.

虽然相比于防止错误，我们更乐于纠正错误，可是，在某些场景下，防止错误比发生错误后再处理更好。在安全领域就是如此：如果针对一个系统的攻击已经发生了，并且窃取了一些敏感数据，这些事情是没办法回退的。因此，本书后续部分主要讨论可以被纠正的错误。

**Hardware Faults**

**硬件错误**

When we think of causes of system failure, hardware faults quickly come to mind. Hard disks crash, RAM becomes faulty, the power grid has a blackout, someone unplugs the wrong network cable. Anyone who has worked with large datacenters can tell you that these things happen *all the time* when you have a lot of machines.

当我们想到系统失败的原因时，第一个冒出的想法是硬件错误。硬件坏掉，RAM错误，电网断电，网络断线。在大型数据中心工作过的人都知道，当你有很多机器的时候，这些问题经常发生。

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years [5, 6]. Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

据说硬盘平局失效周期（MTTF）是10到50年[5,6]。因此，在一个超过10000块硬盘的集群中，几乎每天都有硬盘坏掉。

Our first response is usually to add redundancy to the individual hardware components in order to reduce the failure rate of the system. Disks may be set up in a RAID configuration, servers may have dual power supplies and hot-swappable CPUs, and datacenters may have batteries and diesel generators for backup power. When one component dies, the redundant component can take its place while the broken com‐ ponent is replaced. This approach cannot completely prevent hardware problems from causing failures, but it is well understood and can often keep a machine running uninterrupted for years.

我们想到的最直接的方法是给每个独立的硬件提供副本，这样就能减少整个系统失败的概率。硬盘可以配置为RAID模式，服务器可以配置双电源供电，可拔插CPUs，数据中心配备电池和产油发电机以备外部断电。这些措施并不能完全防止因硬件错误导致的系统失败，但这很容易理解，一般来说让一台机器不间断运行几年足够了。



**Reliability | 7**

Until recently, redundancy of hardware components was sufficient for most applications, since it makes total failure of a single machine fairly rare. As long as you can restore a backup onto a new machine fairly quickly, the downtime in case of failure is not catastrophic in most applications. Thus, multi-machine redundancy was only required by a small number of applications for which high availability was absolutely essential.

直到最近，硬件的冗余的方式可以使弹尽很少失败，针对大多数应用都是足够了。只要你能迅速的换掉损坏部件，针对大多数应用来说都不会有灾难性问题。因此，仅仅少数需要绝对高可用的系统需要多机冗余机制保证高可用性。

However, as data volumes and applications’ computing demands have increased, more applications have begun using larger numbers of machines, which proportionally increases the rate of hardware faults. Moreover, in some cloud platforms such as Amazon Web Services (AWS) it is fairly common for virtual machine instances to become unavailable without warning [7], as the platforms are designed to prioritize flexibility and elasticityi over single-machine reliability.

但是，随着数据规模和计算量增加，越来越多的应用需要大量的机器，相应地，这也增加了整个系统硬件错误的概率。此外，在一些例如：Amazon Web Services (AWS)这样的云平台，平台由于设计阶段就优先考虑灵活性和弹性而非单机的可靠性，因此虚拟机实例没有报警就宕机是家常便饭。

Hence there is a move toward systems that can tolerate the loss of entire machines, by using software fault-tolerance techniques in preference or in addition to hardware redundancy. Such systems also have operational advantages: a single-server system requires planned downtime if you need to reboot the machine (to apply operating system security patches, for example), whereas a system that can tolerate machine failure can be patched one node at a time, without downtime of the entire system (a *rolling upgrade*; see Chapter 4).

因此，优先通过软件容错或者硬件冗余的方式，系统正在朝着容忍单机整体宕机的方向发展。这种系统有很大的操作优势：如果你需要重启机器，对一个单机系统来说，整个系统必须停止服务（例如给操作系统打安全补丁），但是如果一个系统能容忍单机失败，它就能在某一时刻只给一个节点打补丁，而不会影响整个系统（滚动升级，见第四章）。

**Software Errors**

**软件错误**

We usually think of hardware faults as being random and independent from each other: one machine’s disk failing does not imply that another machine’s disk is going to fail. There may be weak correlations (for example due to a common cause, such as the temperature in the server rack), but otherwise it is unlikely that a large number of hardware components will fail at the same time.

我们一般认为多个部件间的硬件错误是独立、随机的：一台机器上的硬盘故障并不预示另一台机器上的硬盘也会故障。也许会有微弱的联系（例如一个由于共同的因素，比如服务器机架的温度），但是一般情况下，不会发生大量硬件同时故障的情况。

Another class of fault is a systematic error within the system [8]. Such faults are harder to anticipate, and because they are correlated across nodes, they tend to cause many more system failures than uncorrelated hardware faults [5]. Examples include:

系统中的另一类错误是系统性的错误。这类错误很难预料，因为它们是跨节点的，它们一般会引发系统级的错误，而不仅仅是硬件错误。这包括：

* A software bug that causes every instance of an application server to crash when given a particular bad input. For example, consider the leap second on June 30, 2012, that caused many applications to hang simultaneously due to a bug in the Linux kernel [9].
* 接收到错误输入引发的所有服务实例崩溃的软件bug。例如，一个Linux内核bug，导致时间跨过2012年6月30日时，许多应用会hang住。
* A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth.
* 一个失控进程用过了一些共享的资源-CPU时间 ，内存，硬盘空间，网络带宽



i.

Defined in “Approaches for Coping with Load” on page 17. **| Chapter 1: Reliable, Scalable, and Maintainable Applications**



**8**

* A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.
* 一个系统依赖的服务变慢了，无响应或者返回已经崩溃的相应。
* Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults [10].
* 雪崩式失败，一个小的硬件错误导致的系统错误会触发其它的错误，如此重复，触发出很多错误。

The bugs that cause these kinds of software faults often lie dormant for a long time until they are triggered by an unusual set of circumstances. In those circumstances, it is revealed that the software is making some kind of assumption about its environment—and while that assumption is usually true, it eventually stops being true for some reason [11].

这种能引发软件错误的bug经常潜伏在系统隐蔽处很久，直到再一些非常规环境下才会被触发。在这些场景中，软件会对它运行的环境进行一些一般情况下都成立的假设，但是由于某些原因这些假设不成立了。

There is no quick solution to the problem of systematic faults in software. Lots of small things can help: carefully thinking about assumptions and interactions in the system; thorough testing; process isolation; allowing processes to crash and restart; measuring, monitoring, and analyzing system behavior in production. If a system is expected to provide some guarantee (for example, in a message queue, that the number of incoming messages equals the number of outgoing messages), it can constantly check itself while it is running and raise an alert if a discrepancy is found [12].

没有针对软件中的系统级错误的特效药。以下准则有些帮助：认真推敲系统中的假设和相互依赖；完整的测试；处理隔离；运行进程的中断和重启；衡量、监控、分析系统中的某些表现。在运行过程中不断检查自己的状态并在非预期差异出现时发出报警能给系统提供更多保证（例如，在消息队列中，系统接受的消息条数必须等于输出的条数）。

* **Human Errors**
* **认为错误**

Humans design and build software systems, and the operators who keep the systems running are also human. Even when they have the best intentions, humans are known to be unreliable. For example, one study of large internet services found that configuration errors by operators were the leading cause of outages, whereas hardware faults (servers or network) played a role in only 10–25% of outages [13].  How do we make our systems reliable, in spite of unreliable humans? The best systems combine several approaches:

设计、实现系统是的人，操作系统运行的也是人。众所周知，无论多么专心， 人也总是不可靠的。例如一项研究表明操作人员的配置错误导致了绝大多数系统停服务，相对而言，硬件故障（服务器或网络）仅仅占到10-25%。人不可靠的话，怎么才能使系统可靠呢？设计一个好的系统有以下几个方法：

* Design systems in a way that minimizes opportunities for error. For example, well-designed abstractions, APIs, and admin interfaces make it easy to do “the right thing” and discourage “the wrong thing.” However, if the interfaces are too restrictive people will work around them, negating their benefit, so this is a tricky balance to get right.
* 采用减少人犯错可能性的设计方法。例如：良好抽象，API和管理接口使系统错误更少。但是，如果系统接口限制太多，会触犯将来使用它的人的利益，因此这需要好好权衡。
* Decouple the places where people make the most mistakes from the places where they can cause failures. In particular, provide fully featured non-production *sandbox* environments where people can explore and experiment safely, using real data, without affecting real users.
* 分割人犯错误和错误发生引起损失的场景。特别地，提供一个全功能的非生产沙河环境，可以在那里安全地用真实数据进行探索和实验，而不会影响到真实用户
* Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests [3]. Automated testing is widely used, well understood, and especially valuable for covering corner cases that rarely arise in normal operation.
* 提供各个级别的测试，从单测到系统级别的集成测试和人工测试。自动测试已经被广泛使用，也容易理解，对要覆盖很少触发的处于死角的场景尤其有用。
* Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure. For example, make it fast to roll back configuration changes, roll out new code gradually (so that any unexpected bugs affect only a small subset of users), and provide tools to recompute data (in case it turns out that the old computation was incorrect).
* 为减少系统停服务影响，要让系统能从人为错误中方便、快速的恢复。例如：配置修改的快速回滚，新代码的逐步上线（非预期的bug仅仅会影响一小部分用户），提供工具对数据进行重新计算（防止老的计算结果不正确）
* Set up detailed and clear monitoring, such as performance metrics and error rates. In other engineering disciplines this is referred to as *telemetry*. (Once a rocket has left the ground, telemetry is essential for tracking what is happening, and for understanding failures [14].) Monitoring can show us early warning signals and allow us to check whether any assumptions or constraints are being violated. When a problem occurs, metrics can be invaluable in diagnosing the issue.
* 设置详尽清晰的监控，例如性能和错误率监控。在另一个工程领域叫做遥测（当一个火箭升空后，遥测是追踪火箭正在如何运行以及理解错误的基本方法[14]）。监控能给我提早报警，允许我们确认是否某些假设和系统限制是有效的。问题发生后，监控数据对问题追查无比重要。
* Implement good management practices and training—a complex and important aspect, and beyond the scope of this book.
* 好的管理实践和训练-这是一个非常重要复杂的因素，已经超出本书讨论范围。

**How Important Is Reliability?**

**可靠性有多重要**

 Reliability is not just for nuclear power stations and air traffic control software— more mundane applications are also expected to work reliably. Bugs in business applications cause lost productivity (and legal risks if figures are reported incorrectly), and outages of ecommerce sites can have huge costs in terms of lost revenue and damage to reputation.  Even in “noncritical” applications we have a responsibility to our users. Consider a parent who stores all their pictures and videos of their children in your photo application [15]. How would they feel if that database was suddenly corrupted? Would they know how to restore it from a backup?  There are situations in which we may choose to sacrifice reliability in order to reduce development cost (e.g., when developing a prototype product for an unproven market) or operational cost (e.g., for a service with a very narrow profit margin)—but we should be very conscious of when we are cutting corners.

可靠性并不仅仅对发电站，空管系统来说很重要，愈来越多普通系统也需要可靠的工作。交易系统中的bug会引起重大损失（如果某些数据不正确也会有法律风险），商业网站的停服务会引起利润和商誉的损失。即使在“非关键”的应用中，我们也要对自己用户负责。设想一下，一个父母把他们所有孩子的照片和视频存储在你的应用中，数据库突然崩溃了，他们会作何感受？难道他们要知道怎么从一个备份中恢复数？有些场景中，我们为了减少开发成本（比如：给一个未经证实的市场开发一个原型时）或运营成本（一个只有很少盈利空间的服务）而牺牲稳定性，但是我们必须十分清楚自己选择的后果。

**Scalability**

可扩展性

Even if a system is working reliably today, that doesn’t mean it will necessarily work reliably in the future. One common reason for degradation is increased load: perhaps the system has grown from 10,000 concurrent users to 100,000 concurrent users, or from 1 million to 10 million. Perhaps it is processing much larger volumes of data than it did before.

即使今天系统是可靠的，也不意味着将来也是可靠的。一个服务降级的普遍的原因系统负载的增加：系统并发用户由10000上涨到100000，或者从1百万到1千万。也许它需要处理比以前更多的数据。

*Scalability* is the term we use to describe a system’s ability to cope with increased load. Note, however, that it is not a one-dimensional label that we can attach to a system: it is meaningless to say “X is scalable” or “Y doesn’t scale.” Rather, discussing scalability means considering questions like “If the system grows in a particular way, what are our options for coping with the growth?” and “How can we add computing resources to handle the additional load?”

可扩展性用来描述系统处理增长负载的能力。它不是一个非黑即白的描述系统的标签：说“X是扩展的”或者“Y是不可扩展的”是没意义的。相对来说，讨论可扩展性就像这些问题“假如系统按照一种特定的方式进行增长，我们应付这种增长的选择是什么？”和“为了应付额外的负载，我们改怎么增加计算资源？”。

**Describing Load**

**负载描述**

First, we need to succinctly describe the current load on the system; only then can we discuss growth questions (what happens if our load doubles?). Load can be described with a few numbers which we call *load parameters*. The best choice of parameters depends on the architecture of your system: it may be requests per second to a web server, the ratio of reads to writes in a database, the number of simultaneously active users in a chat room, the hit rate on a cache, or something else. Perhaps the average case is what matters for you, or perhaps your bottleneck is dominated by a small number of extreme cases.

首先，我们需要简要的描述系统中的当前负载；只有在此之后我们才能讨论增长的问题（假如我们的负载翻倍后会发生什么问题？）。我们通过负载参数的来描述负载。最好依据系统架构来选择参数：它可以是一个web服务器每秒接收请求数，数据库的读写比例，聊天室同时在线人数，cache命中比例，或者其它。通用原则是：用你关心的数据，或者很极端场景才会触发的系统瓶颈。

To make this idea more concrete, let’s consider Twitter as an example, using data published in November 2012 [16]. Two of Twitter’s main operations are:

为了使这个概念更具体，我们拿根据Twitter November 2012发布的数据来举例。Twitter的两个主要操作是：

*Post tweet*

*发tweet*

A user can publish a new message to their followers (4.6k requests/sec on average, over 12k requests/sec at peak).

一个用户向所有关注他的人发布一条讯息（平均有4600条请求每秒，峰值12000条每秒）

*Home timeline*

*主页时间线*

A user can view tweets posted by the people they follow (300k requests/sec).

一个用户可以查看他关注所有人发布的tweets。

Simply handling 12,000 writes per second (the peak rate for posting tweets) would be fairly easy. However, Twitter’s scaling challenge is not primarily due to tweet volume, but due to *fan-out*ii—each user follows many people, and each user is followed by many people. There are broadly two ways of implementing these two operations:

单单一秒处理12000条写请求（发tweets的峰值速度）就相当不容易。但是，twitter的可扩展性挑战不主要来自请求规模，而是扇出-每个人都有很多粉丝，每个人都被很多人关注。一般来说有两种方式来实现这两种操作：

1. Posting a tweet simply inserts the new tweet into a global collection of tweets. When a user requests their home timeline, look up all the people they follow, find all the tweets for each of those users, and merge them (sorted by time). In a relational database like in Figure 1-2, you could write a query such as:

发送请求动作会向全局的整体tweets数据集合插入一条新tweet。当一个人请求他的home timeline时，会查看他关注的所有人，然后找到每个人的更新，再把这些更新揉和在一起（以时间排序）。在Figure 1-2指示的关系型数据库中，你需要写一个这个的查询语句：

**SELECT** tweets.\*, users.\* **FROM** tweets **JOIN** users **ON** tweets.sender\_id = users.id **JOIN** follows **ON** follows.followee\_id = users.id **WHERE** follows.follower\_id = **current\_user**

ii. A term borrowed from electronic engineering, where it describes the number of logic gate inputs that are attached to another gate’s output. The output needs to supply enough current to drive all the attached inputs. In transaction processing systems, we use it to describe the number of requests to other services that we need to make in order to serve one incoming request.

从电气工程领域引入的一个词，它描述一个逻辑门输出到其他逻辑门输入的数目。输出必须提供足够的并发驱动所有的输入。在事务系统中，我们用它来描述为了处理一个请求，我们需要向其他服务发出多少个请求

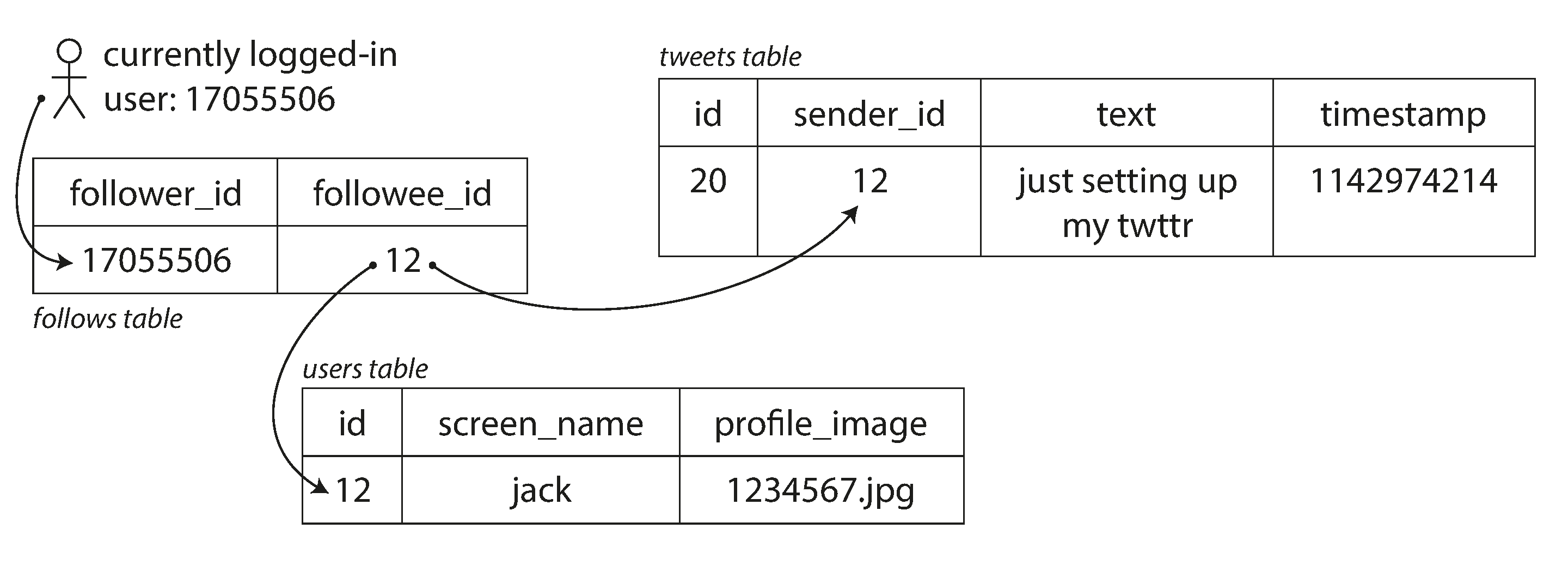
**Scalability | 11**

1. Maintain a cache for each user’s home timeline—like a mailbox of tweets for each recipient user (see Figure 1-3). When a user *posts a tweet*, look up all the people who follow that user, and insert the new tweet into each of their home timeline caches. The request to read the home timeline is then cheap, because its result has been computed ahead of time.

为每一个人的home timeline维护一个cache-就像每个接收者都有一个邮箱（Figure 1-3）。当一个人发送一条tweet时，会查找他的粉丝，然后把这条tweet放入他们的home timeline cache。因为每个人home timeline的都已经事先计算好了，请求这个数据时的读代价就很小了。

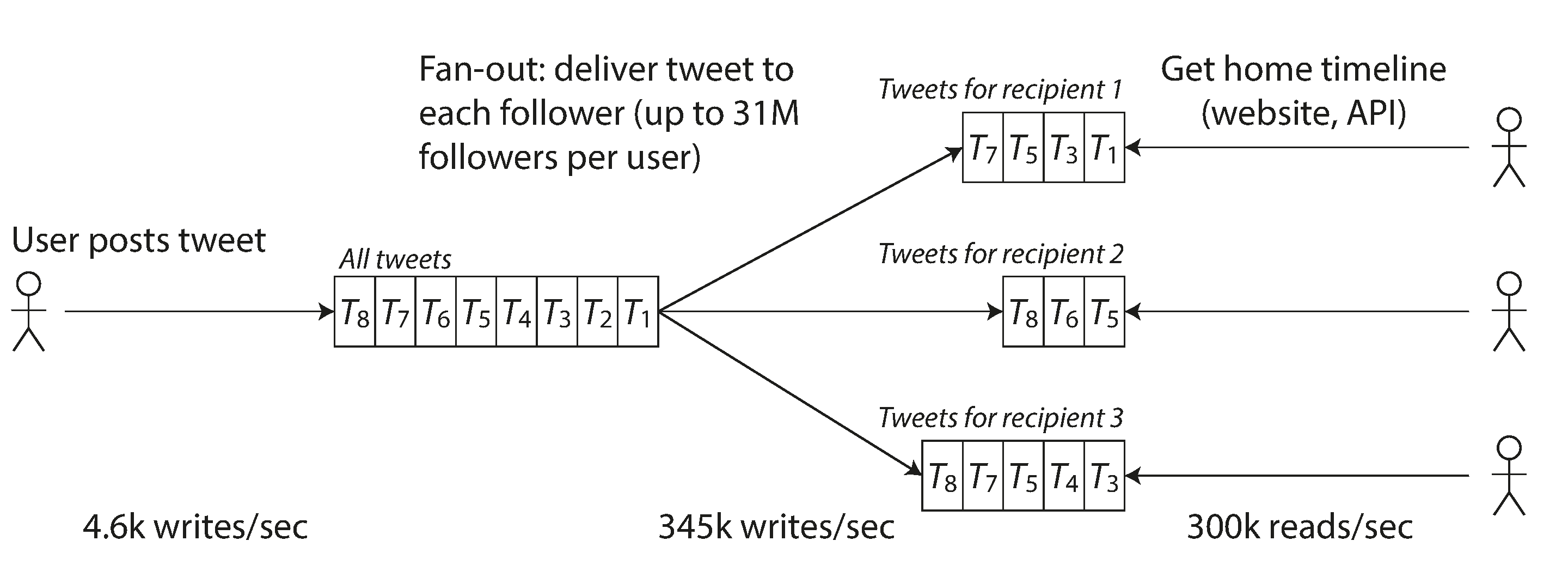
The first version of Twitter used approach 1, but the systems struggled to keep up with the load of home timeline queries, so the company switched to approach 2. This works better because the average rate of published tweets is almost two orders of magnitude lower than the rate of home timeline reads, and so in this case it’s preferable to do more work at write time and less at read time.

Twitter第一个实现版本采用方案一，但是系统随着home timeline请求的增长，变得很困难，因此他们切换到方案二。这个方案更好，因为发tweet的数目几乎是读home timeline数目的一半，同时这个方案在写的时候花费的时间多一些，在读的时候耗时少一些。



*Figure 1-2. Simple relational schema for implementing a Twitter home timeline.*

*实现Twitter home timeline简单的关系型数据库schema*



*Figure 1-3. Twitter’s data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].*

*Twitter用消息队列将tweets发送到给*粉丝

However, the downside of approach 2 is that posting a tweet now requires a lot of extra work. On average, a tweet is delivered to about 75 followers, so 4.6k tweets per second become 345k writes per second to the home timeline caches. But this average hides the fact that the number of followers per user varies wildly, and some users have over 30 million followers. This means that a single tweet may result in over 30 million writes to home timelines! Doing this in a timely manner— Twitter tries to deliver tweets to followers within five seconds—is a significant challenge.

但是，在方案二中，发送一个tweet需要很多额外的工作。平均一个tweet一般要发送给75个粉丝。但是平均值掩盖了每个人的粉丝个数范围很大这个问题，有些人有超过300万粉丝。这意味的一个tweet会产生300万的写操作！需要及时做完- Twitter会尽力在5s内将数据推送给所有粉丝-是一个非常大的挑战。

In the example of Twitter, the distribution of followers per user (maybe weighted by how often those users tweet) is a key load parameter for discussing scalability, since it determines the fan-out load. Your application may have very different characteristics, but you can apply similar principles to reasoning about its load.

在Twitter的例子中，用户粉丝的分布（可以通过用户发tweet频率衡量）是一个讨论可扩展性的核心负载参数，因为它决定扇出负载。你的系统可能有非常不同的特性，但是你能采取类似的准则推理它的负载。

The final twist of the Twitter anecdote: now that approach 2 is robustly implemented, Twitter is moving to a hybrid of both approaches. Most users’ tweets continue to be fanned out to home timelines at the time when they are posted, but a small number of users with a very large number of followers (i.e., celebrities) are excepted from this fan-out. Tweets from any celebrities that a user may follow are fetched separately and merged with that user’s home timeline when it is read, like in approach 1. This hybrid approach is able to deliver consistently good performance. We will revisit this example in Chapter 12 after we have covered some more technical ground.

Twitter例子的最终发展：现在，方案二确实被实现了，Twitter正在超两中方案和混合方案前进。大多数用户发tweet时，他们的tweet会被发送给粉丝的home timelines，但是少数人有大量粉丝的人（比如名人）从这种方案排除了。一个人关注的名人的tweets和自己的home timeline在读取时是分别请求的，就像方案一。这种混合结构能有一贯的好的性能表现。我们讲述更多的技术背景后，将在12章重新看到这个例子。

**Describing Performance**

**描述性能表现**

Once you have described the load on your system, you can investigate what happens when the load increases. You can look at it in two ways:

一旦有了系统的负载描述，你就能推测出负载增加后的系统表现。亦可以用下面两种方法：

* When you increase a load parameter and keep the system resources (CPU, memory, network bandwidth, etc.) unchanged, how is the performance of your system affected?
* 当你调高了复杂一个参数而又保持系统资源不变（CPU，内存，网络带宽等），系统的性能会怎样呢？
* When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?
* 当你调高一个负载参数，你需要增加多少资源产能保持性能不变呢？

Both questions require performance numbers, so let’s look briefly at describing the performance of a system.  In a batch processing system such as Hadoop, we usually care about *throughput*—the number of records we can process per second, or the total time it takes to run a job on a dataset of a certain size.iii In online systems, what’s usually more important is the service’s *response time*—that is, the time between a client sending a request and receiving a response.

这两个问题都需要性能数据，接下来我们简单看下如何描述一个系统的性能。在像Hadoop这样的批量处理系统中，我们经常关注吞吐-每秒能处理的记录条数，或者处理一个固定大小数据集作业的运行总时间。针对线上系统，服务的相应时间-客户端发送请求到接受到响应时间差-更重要。

iii. In an ideal world, the running time of a batch job is the size of the dataset divided by the throughput. In practice, the running time is often longer, due to skew (data not being spread evenly across worker processes) and needing to wait for the slowest task to complete.

理想状况下，一个批量作业的运行时间是数据集切分成单挑数据和和。实际上，运行总时间一般来说会更长，因为数据倾斜（数据并不总平均分布到所有节点中）和然后等待最慢任务完成

**Latency and response time**

**延迟和响应时间**

*Latency* and *response time* are often used synonymously, but they are not the same. The response time is what the client sees: besides the actual time to process the request (the *service time*), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled—during which it is *latent*, awaiting service [17].

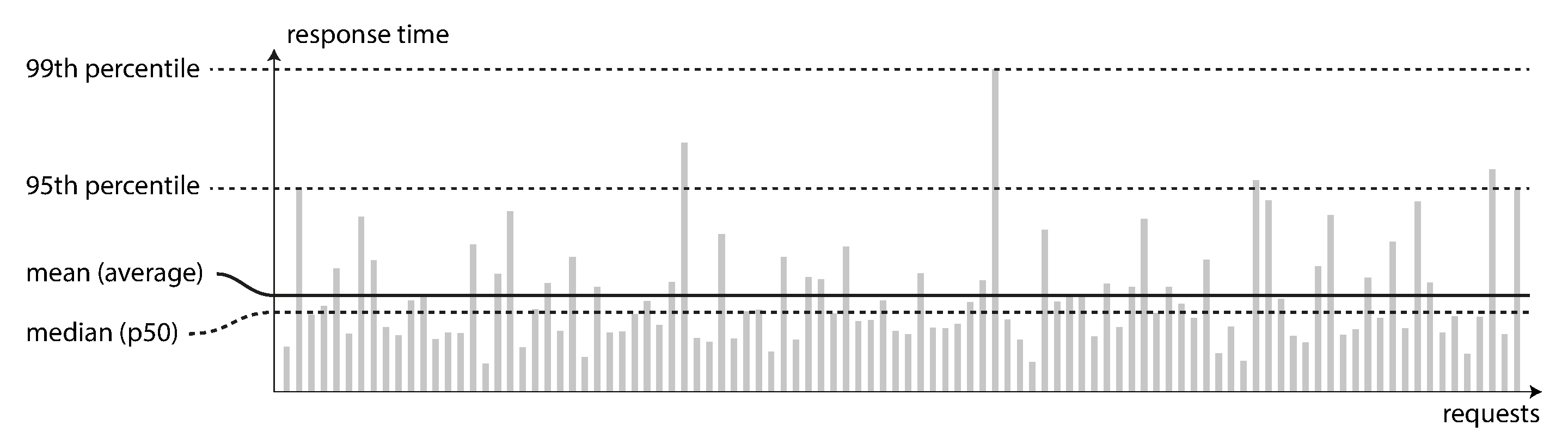
延迟和响应时间经常混用，但严格来讲他们并不同。响应时间是客户端视角：除去处理请求的时间（服务端花费时间），它还包括网络延迟和在队列中等待时间。延迟是请求等待被处理的时间-在这段时间内请求被放在一边，等待被处理。

Even if you only make the same request over and over again, you’ll get a slightly different response time on every try. In practice, in a system handling a variety of requests, the response time can vary a lot. We therefore need to think of response time not as a single number, but as a *distribution* of values that you can measure.

即使你重复发送同一请求，你将得到稍微不同的响应时间。实际上，在处理多个请求的系统中，响应时间差距很大。因此，我们不应该把响应时间当做一个数字，而是一个可以衡量的分布。

In Figure 1-4, each gray bar represents a request to a service, and its height shows how long that request took. Most requests are reasonably fast, but there are occasional *outliers* that take much longer. Perhaps the slow requests are intrinsically more expensive, e.g., because they process more data. But even in a scenario where you’d think all requests should take the same time, you get variation: random additional latency could be introduced by a context switch to a background process, the loss of a network packet and TCP retransmission, a garbage collection pause, a page fault forcing a read from disk, mechanical vibrations in the server rack [18], or many other causes.

在图Figure 1-4中，每个灰条代表一个请求，高度代表响应时间。大多数请求很快被响应，但是偶尔会花费很长时间。也许慢请求本质上更有价值，比如，因为这些请求需要处理更多的数据。即使在一个多有请求假设花费同样时间的场景下，你也会得到变化的值：由于进程调度切换引入的随机延迟，网络丢包和TCP重传，垃圾回收引起的中断，缺页中断导致的从磁盘读数据，服务器的机械振动[18],或者其他原因。

*Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.*

*100次请求的响应时间*

It’s common to see the *average* response time of a service reported. (Strictly speaking, the term “average” doesn’t refer to any particular formula, but in practice it is usually understood as the *arithmetic mean*: given *n* values, add up all the values, and divide by *n*.) However, the mean is not a very good metric if you want to know your “typical” response time, because it doesn’t tell you how many users actually experienced that delay.

一般关注服务的平均响应时间（严格来说，平均这个词并不指任何特定公式算法，在实际中它一般被理解为算数平均：给n个值，求和再除n）。但是，这并不是你想知道的“典型”的响应时间的好的维度，因为它并没有说明多少用户真正能感受到那种时延。

Usually it is better to use *percentiles*. If you take your list of response times and sort it from fastest to slowest, then the *median* is the halfway point: for example, if your median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

一般来说，百分比方式更好。如果你把响应时间列出来并从快到慢排序，中位数也就是中间的数：例如中位数是200ms，意味着一半请求在200ms内返回，一半请求花费更长的时间。

This makes the median a good metric if you want to know how long users typically have to wait: half of user requests are served in less than the median response time, and the other half take longer than the median. The median is also known as the *50th percentile*, and sometimes abbreviated as *p50*. Note that the median refers to a single request; if the user makes several requests (over the course of a session, or because several resources are included in a single page), the probability that at least one of them is slower than the median is much greater than 50%.

如果你想知道典型用户等待多久，中位数一个很好的维度：一半请求的响应时间比中位数短，另一半请求长。中位数也叫做第50百分数，有时也被缩写为p50.注意，中位数代指一个请求；如果一个用户发送多个请求（在一个会话内或者一个页内请求多个资源），至少有一个请求比中位数慢的概率远大于50%。

In order to figure out how bad your outliers are, you can look at higher percentiles: the *95th*, *99th*, and *99.9th* percentiles are common (abbreviated *p95*, *p99*, and *p999*). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more. This is illustrated in Figure 1-4.

为了衡量你的长尾情况，你可以更高的百分数：一般来说采用95分位，99分位和99.9分位值。他们是95%,99%,99.9%请求比分位值更快返回的阀值。例如：95非为值是1.5秒，那意味着100个请求中95个请求耗时不足1.5秒，5个请求超过1.5秒。如图Figure 1-4.

High percentiles of response times, also known as *tail latencies*, are important because they directly affect users’ experience of the service. For example, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases—that is, they’re the most valuable customers [19]. It’s important to keep those customers happy by ensuring the website is fast for them: Amazon has also observed that a 100 ms increase in response time reduces sales by 1% [20], and others report that a 1-second slowdown reduces a customer satisfaction metric by 16% [21, 22].

响应时间的高分位也被叫做长尾延时非常重要，这直接影响用户的服务体验。例如亚马逊描述内部服务响应时间采用99.9分位，虽然这在1000个客户中只会影响其中一个。这是因为最慢请求的客户，也是账号数据数据最多的用户，这些是最具价值用户[19]。确保访问迅速可以使他们购物愉快：Amazon发现响应时间每下降100ms，销售额增加1%[20]，其他报告说明每增加1s响应，客户满意度下降16%[21,22]

On the other hand, optimizing the 99.99th percentile (the slowest 1 in 10,000 requests) was deemed too expensive and to not yield enough benefit for Amazon’s purposes. Reducing response times at very high percentiles is difficult because they are easily affected by random events outside of your control, and the benefits are diminishing.

另一方面，最优的99.99分位（10000个请求中只有一个最慢）代价太昂贵，小于产生的利润。将服务保持在高百分位很困难，因为那种情况很容易受不可控随机事件的影响，并且产出的效益很小。

For example, percentiles are often used in *service level objectives* (SLOs) and *service level agreements* (SLAs), contracts that define the expected performance and availability of a service. An SLA may state that the service is considered to be up if it has a median response time of less than 200 ms and a 99th percentile under 1 s (if the response time is longer, it might as well be down), and the service may be required to be up at least 99.9% of the time. These metrics set expectations for clients of the service and allow customers to demand a refund if the SLA is not met.

例如：百分位经常用来描述服务水平目标（SLOs）和服务水平协议（SLAs），这些用来描述预期的服务性能和可用性。一个SLA可以表述成：服务响应时间中位数小于200ms，99分位小于1s的时候就认为服务是正常的，并且99.9% 的时候是正常的。这些维度为用户使用服务提供一个预期，如果没有达到SLA客户可以提出退款。

Queueing delays often account for a large part of the response time at high percentiles. As a server can only process a small number of things in parallel (limited, for example, by its number of CPU cores), it only takes a small number of slow requests to hold up the processing of subsequent requests—an effect sometimes known as *head-of-line blocking*. Even if those subsequent requests are fast to process on the server, the client will see a slow overall response time due to the time waiting for the prior request to complete. Due to this effect, it is important to measure response times on the client side.

排队时延经常是长尾的主要因素。一个服务器同时时刻只能处理几个请求（有限的，比如CPU的核数），几个慢请求就会阻挡住后续的请求-一种叫做队头阻塞的现象。即使后续的请求能被迅速处理，因为要等前面的请求处理完，client仍然感到响应时间很长。由于这种效应，衡量客户端端的响应时延非常重要。

When generating load artificially in order to test the scalability of a system, the load- generating client needs to keep sending requests independently of the response time. If the client waits for the previous request to complete before sending the next one, that behavior has the effect of artificially keeping the queues shorter in the test than they would be in reality, which skews the measurements [23].

当为了测试系统可扩展性手动发压力是，产生请求的客户端要保证互相独立地发送每个请求。如果客户端等到上一个请求返回再发送下一个请求，这人工地会使队列比实际情况中短，会歪曲测试结果。

**Percentiles in Practice**

**百分位实践**

High percentiles become especially important in backend services that are called multiple times as part of serving a single end-user request. Even if you make the calls in parallel, the end-user request still needs to wait for the slowest of the parallel calls to complete. It takes just one slow call to make the entire end-user request slow, as illustrated in Figure 1-5. Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple back‐ end calls, and so a higher proportion of end-user requests end up being slow (an effect known as *tail latency amplification* [24]).

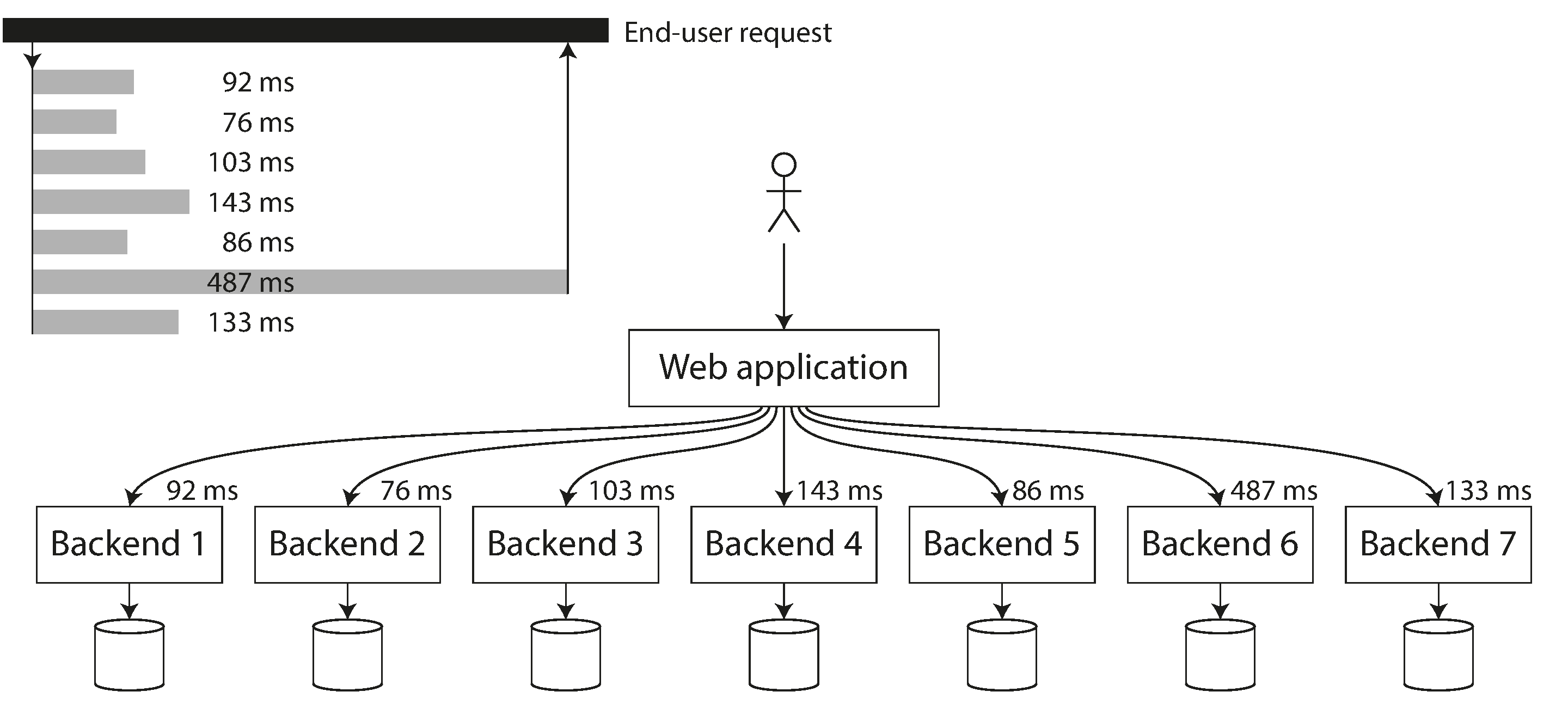
百分位的表现对于一次前端请求对应多次后端请求的后端服务尤其重要。即使你并行发送后端请求，前端用户请求依然需要等到并行中最慢的请求完成。一个慢的后端调用就让整个请求慢下来，就像Figure 1-5中展示的。即使只有一小部分后端请求很慢，如果用户请求被转发为后端的多个请求，用户请求时延增加的概率会增加很多。( 叫做长尾放大[24])

If you want to add response time percentiles to the monitoring dashboards for your services, you need to efficiently calculate them on an ongoing basis. For example, you may want to keep a rolling window of response times of requests in the last 10 minutes. Every minute, you calculate the median and various percentiles over the values in that window and plot those metrics on a graph.

如果你想把百分位响应时间作为服务监控页面的一个维度，你需要持续有效的计算这些值。每分钟，你都需要计算中位数和各个百分位并将它们展示在图表中。

The naïve implementation is to keep a list of response times for all requests within the time window and to sort that list every minute. If that is too inefficient for you, there are algorithms that can calculate a good approximation of percentiles at minimal CPU and memory cost, such as forward decay [25], t-digest [26], or HdrHistogram [27]. Beware that averaging percentiles, e.g., to reduce the time resolution or to combine data from several machines, is mathematically meaningless—the right way of aggregating response time data is to add the histograms [28].

最简单的实现就是维护一个时间窗口中的请求时延队列，并且每分钟排序一次。假如这种做法对你来说效率太低，有一些好的比较好的算法能用很少的CPU和内存资源来估算百分位，比如decay [25], t-digest [26], or HdrHistogram [27].百分位的平均值（比如., to reduce the time resolution或者取多个机器上百分位平均值在数学角度来说是没意义的），正确的聚合响应时间的方法是把这些数据放入矩阵图[28]



*Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.*

*当一个前端请求需要多个后端请求时，一个慢后端查询就会使整个前端查询变慢*

**Approaches for Coping with Load**

**处理负载问题方法**

Now that we have discussed the parameters for describing load and metrics for measuring performance, we can start discussing scalability in earnest: how do we maintain good performance even when our load parameters increase by some amount?

现在已经说明了描述负载的参数和衡量性能表现的维度，我们开始认真讨论一下可扩展性：当负载参数增加时，我们系统如何才能保持好的性能。

An architecture that is appropriate for one level of load is unlikely to cope with 10 times that load. If you are working on a fast-growing service, it is therefore likely that you will need to rethink your architecture on every order of magnitude load increase —or perhaps even more often than that.

一个适合处理特定负载的架构不太可能处理原有基础10倍的负载。如果你在负责一个快速增涨的服务，每次负载大的增长，你都需要重新思考你的系统架构，有时候频率需要更高。

People often talk of a dichotomy between *scaling up* (*vertical scaling*, moving to a more powerful machine) and *scaling out* (*horizontal scaling*, distributing the load across multiple smaller machines). Distributing load across multiple machines is also known as a *shared-nothing* architecture. A system that can run on a single machine is often simpler, but high-end machines can become very expensive, so very intensive workloads often can’t avoid scaling out. In reality, good architectures usually involve a pragmatic mixture of approaches: for example, using several fairly powerful machines can still be simpler and cheaper than a large number of small virtual machines.

人民一般分开讨论扩大（垂直扩展，迁移到能力更强机器上）和扩展（水平扩展，将负载分布到多台更小的机器上）。负载在多个机器间的分布也叫做“无共享”架构。运行在单机上的系统一般来说比较简单，高端机器又太贵，因此大量的负载经常不可避免的被扩展到多台机器上。实际上，好的架构经常是两种方式的混合体：例如利用少数的高端机器能保持结构简单，并且比大量的小型虚拟机要便宜。

Some systems are *elastic*, meaning that they can automatically add computing resources when they detect a load increase, whereas other systems are scaled manually (a human analyzes the capacity and decides to add more machines to the system). An elastic system can be useful if load is highly unpredictable, but manually scaled systems are simpler and may have fewer operational surprises (see “Rebalancing Partitions” on page 209).

一些系统是“弹性的”，意味着发现负责增加时，它们能自动地增加计算资源，其它的系统只能手动扩容（人工分析容量决定需要加多少台机器到系统中）。当负载不宜预判的时候，弹性系统非常有用。但是手工扩容系统更简单，也会有更少的运营意外。（see “Rebalancing Partitions” on page 209）

While distributing stateless services across multiple machines is fairly straightforward, taking stateful data systems from a single node to a distributed setup can introduce a lot of additional complexity. For this reason, common wisdom until recently was to keep your database on a single node (scale up) until scaling cost or high-availability requirements forced you to make it distributed.

相对于将无状态服务分布在多台机器上很简单，将有状态数据系统从单机分布到多机上会引入很多额外的复杂性。因此，直到最近，大家达成的共识一直都是将数据放在一个单独的节点（扩大）直到扩容代价或者高可用要求不能满足再使系统变成分布式。

As the tools and abstractions for distributed systems get better, this common wisdom may change, at least for some kinds of applications. It is conceivable that distributed data systems will become the default in the future, even for use cases that don’t handle large volumes of data or traffic. Over the course of the rest of this book we will cover many kinds of distributed data systems, and discuss how they fare not just in terms of scalability, but also ease of use and maintainability.

随着分布式系统的工具和抽象工作越来越好，至少针对某些种类的应用，共识可能变化了。可预见的是，将来分布式系统将会成为系统标配，即使不需要处理大量数据或者请求的场景。在本书剩下部分，我们将涉及很多分布式数据系统，不只在可扩展性方面进行讨论，也会包含如何更方便使用和维护。

The architecture of systems that operate at large scale is usually highly specific to the application—there is no such thing as a generic, one-size-fits-all scalable architecture (informally known as *magic scaling sauce*). The problem may be the volume of reads, the volume of writes, the volume of data to store, the complexity of the data, the response time requirements, the access patterns, or (usually) some mixture of all of these plus many more issues.

针对大规模系统的架构一般都是为应用高订制化的，没有通用的架构（*magic scaling sauce*）。问题可能是读容量，写容量，数据存储量，数据复杂度，响应时延，访问模式或者这些和其它问题的混合。

For example, a system that is designed to handle 100,000 requests per second, each 1 kB in size, looks very different from a system that is designed for 3 requests per minute, each 2 GB in size—even though the two systems have the same data throughput.

例如，一个系统设计为每秒处理100000个请求，每个请求长度在1k内。这个系统和设计为每分钟处理三个请求，每个请求2G的系统一定大不相同，虽然两个系统有相同的吞吐量。

An architecture that scales well for a particular application is built around assumptions of which operations will be common and which will be rare—the load parameters. If those assumptions turn out to be wrong, the engineering effort for scaling is at best wasted, and at worst counterproductive. In an early-stage startup or an unproven product it’s usually more important to be able to iterate quickly on product features than it is to scale to some hypothetical future load.

针对特定应用扩展性好的架构一定基于在负载参数范围内，某些操作频繁，某些操作少的假设。如果假设不成立，在此之上的为了扩展性的设计至少是没用的，搞不好还会适得其反。在项目启动阶段或者未证实的产品，在产品特性上的快速迭代比为了将来也许的容量扩容要重要的多。

Even though they are specific to a particular application, scalable architectures are nevertheless usually built from general-purpose building blocks, arranged in familiar patterns. In this book we discuss those building blocks and patterns.

虽然高扩展性的架构一般都是针对某一个特定应用设计的，但是它们经常由一些通用组件，按照一些通用模式组成的。本书我们将讨论这些通用组件和模式。

**Maintainability**

**可运维性**

It is well known that the majority of the cost of software is not in its initial development, but in its ongoing maintenance—fixing bugs, keeping its systems operational, investigating failures, adapting it to new platforms, modifying it for new use cases, repaying technical debt, and adding new features.

众所周知，软件的主要成本不是在初始开发阶段，而是在持续运维阶段-修复bug，保证系统容易操作，查找问题，新平台适配，新业务场景调整，偿还以前的技术性债务，增加一下儿新特性。

Yet, unfortunately, many people working on software systems dislike maintenance of so-called *legacy* systems—perhaps it involves fixing other people’s mistakes, or working with platforms that are now outdated, or systems that were forced to do things they were never intended for. Every legacy system is unpleasant in its own way, and so it is difficult to give general recommendations for dealing with them.

非常不幸的是，软件开发领域的许多人不喜欢运维所谓的遗留系统，可能是这需要修复别人遗留bug，或者需要折腾一个过时系统，或者需要增加一个系统非设计考虑到的一个新功能。每个遗留系统都有各自的讨人嫌的地方，很难给出通用建议。

However, we can and should design software in such a way that it will hopefully minimize pain during maintenance, and thus avoid creating legacy software ourselves. To this end, we will pay particular attention to three design principles for software systems:

但是应该并且能给出一些设计系统的原则，它们有望减少运维的代价，因此避免自己的系统成为遗留系统。最后，我们专门说一下设计软件系统的三个原则：

*Operability*

*可操作性*

Make it easy for operations teams to keep the system running smoothly.

为让系统 平滑运行，运营团队比如容易操作

*Simplicity*

*简洁性*

Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system. (Note this is not the same as simplicity of the user interface.)

通过尽量移除系统的复杂性，让新加入的工程师容易立即系统（和保持系统接口的简洁不同）

*Evolvability*

*可进化性*

Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as *extensibility*, *modifiability*, or *plasticity*.

让将来的系统改变比较容易，即使需要改变了，调整一些非预期的用户场景也不会太麻烦。也被叫做可扩展性，可修改性或者弹性。

As previously with reliability and scalability, there are no easy solutions for achieving these goals. Rather, we will try to think about systems with operability, simplicity, and evolvability in mind.

上面说到了可用性和扩展性，针对这些特性，没有什么灵丹妙药。我们应该把系统的易操作，简洁性和可进化性铭记于心。

**Operability: Making Life Easy for Operations**

**可操作性：让运营更容易**

It has been suggested that “good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations” [12]. While some aspects of operations can and should be automated, it is still up to humans to set up that automation in the first place and to make sure it’s working correctly.

大家常说“好的运营经常能处理坏系统的短板，但是一个好的软件也不能在一个坏运行人员手下好好工作”[12]。尽管有一些运维操作可以也应该被自动化，这也需要人来把自动化设置为最高优先级并且确保自动化机制正常工作。

Operations teams are vital to keeping a software system running smoothly. A good operations team typically is responsible for the following, and more [29]:

运营团队必须保证系统平滑运行。好的运营团队负责一下任务[29]：

* Monitoring the health of the system and quickly restoring service if it goes into a bad state
* 监控系统的运行状态，如果系统错误能迅速恢复
* Tracking down the cause of problems, such as system failures or degraded performance
* 快速定位问题，比如到底是系统故障还是性能下降
* Keeping software and platforms up to date, including security patches
* 保证软件和平台及时更新，包括一些安全补丁
* Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage
* 监视不同系统间的互相影响，保证有问题的改动在造成损失前发现
* Anticipating future problems and solving them before they occur (e.g., capacity planning)
* 对问题有预见性，防患于未然（比如扩容计划）
* Establishing good practices and tools for deployment, configuration management, and more
* 为系统部署，配置管理等操作提供良好的规范和工具
* Performing complex maintenance tasks, such as moving an application from one platform to another
* 能完成复杂的运维任务，比如将应用从一个平台迁移到另一个平台
* Maintaining the security of the system as configuration changes are made
* 随着配置修改，能保证系统的安全性
* Defining processes that make operations predictable and help keep the production environment stable
* 为使操作后果可预见和线上环境稳定制定流程
* Preserving the organization’s knowledge about the system, even as individual people come and go
* 虽然有人员流动，保证知识的传承

Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities. Data systems can do various things to make routine tasks easy, including:

好的运营让例行工作变得简单，让运营团队专注于更有价值的活动。数据系统让例行工作简单的几个准则如下：

* Providing visibility into the runtime behavior and internals of the system, with good monitoring
* 让系统的例行工作和内部状态可见，并配有完备监控
* Providing good support for automation and integration with standard tools
* 提供自动化和标准化的工具支持
* Avoiding dependency on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted)
* 避免依赖单独的机器（允许任何机器宕机运维而不影响系统整体运行）
* Providing good documentation and an easy-to-understand operational model (“If I do X, Y will happen”)
* 提供好的文档和操作模式（你进行操作X，会引起现象Y发生）
* Providing good default behavior, but also giving administrators the freedom to override defaults when needed
* 提供好的默认行为，同时也给管理员充分的自由去重新设置默认行为
* Self-healing where appropriate, but also giving administrators manual control over the system state when needed
* 在需要的地方进行自动恢复，也给管理员手动去设置系统状态的接口
* Exhibiting predictable behavior, minimizing surprises
* 列出可能的系统行为，减少突然性

**Simplicity: Managing Complexity**

简洁性：管理的复杂性

Small software projects can have delightfully simple and expressive code, but as projects get larger, they often become very complex and difficult to understand. This complexity slows down everyone who needs to work on the system, further increasing the cost of maintenance. A software project mired in complexity is sometimes described as a *big ball of mud* [30].

小的软件项目有简洁明了的代码，但是随着项目越来越大，代码会越来越复杂，难以理解。这种复杂性拖慢了项目中的所有人，进一步增加了运维的代价。深陷于复杂性的系统有时候被叫做“大泥球”[30]

There are various possible symptoms of complexity: explosion of the state space, tight coupling of modules, tangled dependencies, inconsistent naming and terminology, hacks aimed at solving performance problems, special-casing to work around issues elsewhere, and many more. Much has been said on this topic already [31, 32, 33].

有很多可能的复杂度增加症状：状态空间的爆炸增长，模块间紧耦合，依赖紧密，命名和专业用语的不一致，解决性能问题的小后门，解决别处问题的特例还有其它等等。此处大多数都有讲述[31,32,33]

When complexity makes maintenance hard, budgets and schedules are often over‐ run. In complex software, there is also a greater risk of introducing bugs when making a change: when the system is harder for developers to understand and reason about, hidden assumptions, unintended consequences, and unexpected interactions are more easily overlooked. Conversely, reducing complexity greatly improves the maintainability of software, and thus simplicity should be a key goal for the systems we build.

当运维复杂度太大时，预算和排期一般也会超限。复杂的软件中，做出的一个改变引入bug的风险也变大：当开发者不易理解系统，弄不清楚隐藏的假设，非刻意的结果和非预期的交互更容易被忽略。相反，减少系统复杂度提高软件的可运维性，因此，简洁性是系统的一个核心目标。

Making a system simpler does not necessarily mean reducing its functionality; it can also mean removing *accidental* complexity. Moseley and Marks [32] define complexity as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.

让系统变得更简洁不一定意味着砍掉某些功能；它也可意味着移除很多意外引入的复杂度。Moseley and Marks [32]把意外引入的复杂度定义为：不是因为软件要解决的问题，而是因为实现而引入的复杂性。

One of the best tools we have for removing accidental complexity is *abstraction*. A good abstraction can hide a great deal of implementation detail behind a clean, simple-to-understand façade. A good abstraction can also be used for a wide range of different applications. Not only is this reuse more efficient than reimplementing a similar thing multiple times, but it also leads to higher-quality software, as quality improvements in the abstracted component benefit all applications that use it.

最好的防止引入意外复杂性的方法是抽象。好的抽象能把实现的细枝末节隐藏在简洁并容易理解的接口背后。好的抽象也能用在范围广大的多个应用。它不仅能有效复用避免重复实现一个同样的东西，它也会对高质量软件有很大贡献。组件好的抽象抽象，能使所有的应用都受益。

For example, high-level programming languages are abstractions that hide machine code, CPU registers, and syscalls. SQL is an abstraction that hides complex on-disk and in-memory data structures, concurrent requests from other clients, and inconsistencies after crashes. Of course, when programming in a high-level language, we are still using machine code; we are just not using it *directly*, because the programming language abstraction saves us from having to think about it.

例如：高级编程语言是机器代码，CPU寄存器，和系统调用的抽象。SQL语言的抽象隐藏了磁盘、内存中的复杂数据结构，其他client并发请求，软件崩溃后的一致性恢复。当然，用高级语言编程，我们仍然需要用到机器代码；我们只是不会直接使用而已，因为编程语言抽象层帮我们免去对机器语言的思考。

However, finding good abstractions is very hard. In the field of distributed systems, although there are many good algorithms, it is much less clear how we should be packaging them into abstractions that help us keep the complexity of the system at a manageable level.

但是，做一个好的抽象很不容易。在分布式领域，虽然有很多好的算法，但是如何把这些算法做抽象，保证整个系统复杂度在可管理范围仍然很模糊。

Throughout this book, we will keep our eyes open for good abstractions that allow us to extract parts of a large system into well-defined, reusable components.

通过这本书，我们将很关注允许我们把大系统分解成定义良好，可复用组件的好的抽象。

**Evolvability: Making Change Easy**

**可进化性：让改变更容易**

It’s extremely unlikely that your system’s requirements will remain unchanged forever. They are much more likely to be in constant flux: you learn new facts, previously unanticipated use cases emerge, business priorities change, users request new features, new platforms replace old platforms, legal or regulatory requirements change, growth of the system forces architectural changes, etc.

你的系统需求不大可能永远不变。不断变化很可能是常态：你发现新的问题，以前没想到过的场景出现了，交易优先级变化了，用户请求新的熟悉，新平台代替老平台，法律或者监管要求变化了，系统规模增长要求架构也随之变化等等。

In terms of organizational processes, *Agile* working patterns provide a framework for adapting to change. The Agile community has also developed technical tools and patterns that are helpful when developing software in a frequently changing environment, such as test-driven development (TDD) and refactoring.

在一系列有组织化的处理流程中，敏捷开发模式提供一个应对变化迅速调整的框架。敏捷开发社区已经发展出应对频繁变化的环境的技术工具和模式，比如测试驱动开发(TDD)和重构.

Most discussions of these Agile techniques focus on a fairly small, local scale (a couple of source code files within the same application). In this book, we search for ways of increasing agility on the level of a larger data system, perhaps consisting of several different applications or services with different characteristics. For example, how would you “refactor” Twitter’s architecture for assembling home timelines (“Describing Load” on page 11) from approach 1 to approach 2?

大多数关于敏捷开发技术讨论，专注于小，本地范围（在属于同一应用的几个源码文件）。本书，我们将探索大数据系统上的敏捷过程，可能由具有不同特征的多个应用组成。例如：你该如何从方案一到方案二重构Twitter的聚合home timelines的架构(page 11)。

The ease with which you can modify a data system, and adapt it to changing requirements, is closely linked to its simplicity and its abstractions: simple and easy-to- understand systems are usually easier to modify than complex ones. But since this is such an important idea, we will use a different word to refer to agility on a data system level: *evolvability* [34].

你为满足变化的需求，修改一个数据系统的难易程度，和系统的简洁程度和抽象密切相关：简洁易理解的系统比复杂的系统更容易修改。这个理念如此重要，我们用一个专有词汇:可进化性，来指代数据系统中的敏捷开发。

**Summary**

**汇总**

In this chapter, we have explored some fundamental ways of thinking about data-intensive applications. These principles will guide us through the rest of the book, where we dive into deep technical detail.

本章，我们涉及了数据密集系统的基础方法。这些准则将会在本书后续深挖技术细节的时候一直指导我们。

An application has to meet various requirements in order to be useful. There are *functional requirements* (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways), and some *nonfunctional requirements* (general properties like security, reliability, compliance, scalability, compatibility, and maintainability). In this chapter we discussed reliability, scalability, and maintainability in detail.

一个应用如果有用，必须满足很多要求。有很多功能性（它应该做的事情：比如允许数据存储，获取，检索，多种方式处理）和非功能性（一般就像安全，可靠性， compliance,可扩展性，兼容性和易运维性）需求。本章我们讨论了可靠性，可扩展性和可维护性方面的细节。

*Reliability* means making systems work correctly, even when faults occur. Faults can be in hardware (typically random and uncorrelated), software (bugs are typically systematic and hard to deal with), and humans (who inevitably make mistakes from time to time). Fault-tolerance techniques can hide certain types of faults from the end user.

可靠性意味着即使发生错误，系统也能正常的工作。错误可能来自硬件（典型来说随机不相关），软件（bugs一般来说是系统性的，很难处理），和人（一次次不可避免犯错误）。容错技术能将特定类型错误对用户屏蔽。

*Scalability* means having strategies for keeping performance good, even when load increases. In order to discuss scalability, we first need ways of describing load and performance quantitatively. We briefly looked at Twitter’s home timelines as an example of describing load, and response time percentiles as a way of measuring performance. In a scalable system, you can add processing capacity in order to remain reliable under high load.

可扩展性意味着即使负载增加，依然能保证系统良好性能的策略。为了讨论可扩展性，我们首先要能描述负载和性能指标。我们简单通过Twitter的home timelines例子来描述负载，响应时间百分位作为衡量性能的指标。在扩展性良好系统中，你能通过增加资源就能保持在高负载下的可用性。

*Maintainability* has many facets, but in essence it’s about making life better for the engineering and operations teams who need to work with the system. Good abstractions can help reduce complexity and make the system easier to modify and adapt for new use cases. Good operability means having good visibility into the system’s health, and having effective ways of managing it.

可运维性有很多方面，基本来说，它能让系统的工程师和运营团队生活更美好。好的抽象能减少复杂度，让系统更易修改调整以应对新的场景。好的运营意味着对系统健康程度有很好的认识，能有效管理系统。

There is unfortunately no easy fix for making applications reliable, scalable, or maintainable. However, there are certain patterns and techniques that keep reappearing in different kinds of applications. In the next few chapters we will take a look at some examples of data systems and analyze how they work toward those goals.

不幸的是，没有什么妙招让系统变的可靠，可扩展和易运维。但是针对不同的系统有一些重复出现的特定的模式和技术。在下几章，我们将看到一些数据系统的例子，分析他们为达成这些目标如何工作的。

Later in the book, in Part III, we will look at patterns for systems that consist of several components working together, such as the one in Figure 1-1.

本书后续的第三部分，我们将看到一些由不同组件构成系统中的模式，比如Figure 1-1.

**References**

[1] Michael Stonebraker and Uğur Çetintemel: “‘One Size Fits All’: An Idea Whose Time Has Come and Gone,” at *21st International Conference on Data Engineering* (ICDE), April 2005.

[2] Walter L. Heimerdinger and Charles B. Weinstock: “A Conceptual Framework for System Fault Tolerance,” Technical Report CMU/SEI-92-TR-033, Software Engi‐ neering Institute, Carnegie Mellon University, October 1992.

[3] Ding Yuan, Yu Luo, Xin Zhuang, et al.: “Simple Testing Can Prevent Most Criti‐ cal Failures: An Analysis of Production Failures in Distributed Data-Intensive Sys‐ tems,” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

[4] Yury Izrailevsky and Ariel Tseitlin: “The Netflix Simian Army,” *techblog.net‐ flix.com*, July 19, 2011.

[5] Daniel Ford, François Labelle, Florentina I. Popovici, et al.: “Availability in Glob‐ ally Distributed Storage Systems,” at *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2010.

[6] Brian Beach: “Hard Drive Reliability Update – Sep 2014,” *backblaze.com*, Septem‐ ber 23, 2014.

[7] Laurie Voss: “AWS: The Good, the Bad and the Ugly,” *blog.awe.sm*, December 18, 2012.

**Summary | 23**

[8] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: “What Bugs Live in the Cloud?,” at *5th ACM Symposium on Cloud Computing* (SoCC), November 2014. doi:10.1145/2670979.2670986

[9] Nelson Minar: “Leap Second Crashes Half the Internet,” *somebits.com*, July 3, 2012.

[10] Amazon Web Services: “Summary of the Amazon EC2 and Amazon RDS Ser‐ vice Disruption in the US East Region,” *aws.amazon.com*, April 29, 2011.

[11] Richard I. Cook: “How Complex Systems Fail,” Cognitive Technologies Labora‐ tory, April 2000.

[12] Jay Kreps: “Getting Real About Distributed System Reliability,” *blog.empathy‐ box.com*, March 19, 2012.

[13] David Oppenheimer, Archana Ganapathi, and David A. Patterson: “Why Do Internet Services Fail, and What Can Be Done About It?,” at *4th USENIX Symposium on Internet Technologies and Systems* (USITS), March 2003.

[14] Nathan Marz: “Principles of Software Engineering, Part 1,” *nathanmarz.com*, April 2, 2013.

[15] Michael Jurewitz: “The Human Impact of Bugs,” *jury.me*, March 15, 2013. [16] Raffi Krikorian: “Timelines at Scale,” at *QCon San Francisco*, November 2012.

[17] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN: 978-0-321-12742-6

[18] Kelly Sommers: “After all that run around, what caused 500ms disk latency even when we replaced physical server?” *twitter.com*, November 13, 2014.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: “Dynamo: Ama‐ zon’s Highly Available Key-Value Store,” at *21st ACM Symposium on Operating Sys‐ tems Principles* (SOSP), October 2007.

[20] Greg Linden: “Make Data Useful,” slides from presentation at Stanford Univer‐ sity Data Mining class (CS345), December 2006.

[21] Tammy Everts: “The Real Cost of Slow Time vs Downtime,” *webperformanceto‐ day.com*, November 12, 2014.

[22] Jake Brutlag: “Speed Matters for Google Web Search,” *googleresearch.blog‐ spot.co.uk*, June 22, 2009.

[23] Tyler Treat: “Everything You Know About Latency Is Wrong,” *bravenew‐ geek.com*, December 12, 2015.



**24 | Chapter 1: Reliable, Scalable, and Maintainable Applications**

[24] Jeffrey Dean and Luiz André Barroso: “The Tail at Scale,” *Communications of the ACM*, volume 56, number 2, pages 74–80, February 2013. doi: 10.1145/2408776.2408794

[25] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu: “Forward Decay: A Practical Time Decay Model for Streaming Systems,” at *25th IEEE International Conference on Data Engineering* (ICDE), March 2009.

[26] Ted Dunning and Otmar Ertl: “Computing Extremely Accurate Quantiles Using t-Digests,” *github.com*, March 2014.

[27] Gil Tene: “HdrHistogram,” *hdrhistogram.org*. [28] Baron Schwartz: “Why Percentiles Don’t Work the Way You Think,” *vividcor‐*

*tex.com*, December 7, 2015. [29] James Hamilton: “On Designing and Deploying Internet-Scale Services,” at *21st*

*Large Installation System Administration Conference* (LISA), November 2007. [30] Brian Foote and Joseph Yoder: “Big Ball of Mud,” at *4th Conference on Pattern*

*Languages of Programs* (PLoP), September 1997.

[31] Frederick P Brooks: “No Silver Bullet – Essence and Accident in Software Engi‐ neering,” in *The Mythical Man-Month*, Anniversary edition, Addison-Wesley, 1995. ISBN: 978-0-201-83595-3

[32] Ben Moseley and Peter Marks: “Out of the Tar Pit,” at *BCS Software Practice Advancement* (SPA), 2006.

[33] Rich Hickey: “Simple Made Easy,” at *Strange Loop*, September 2011.

[34] Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson: “Analyzing Software Evolvability,” at *32nd Annual IEEE International Computer Software and Applica‐ tions Conference* (COMPSAC), July 2008. doi:10.1109/COMPSAC.2008.50