**Chapter 1 Reliable, Scalable, and Maintainable Applications**

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?

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

因特网做得如此好以至于大多数人认为它是像大西洋一样的自然产物，而不是人类创造的。那么最近一次如此大规模的、完美的科学技术是什么时候呢？

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.

如今很多的应用程序都是数据密集型的， 而不是计算密集型的。CPU处理能力很少是这些应用程序的限制因素，通常，更大的问题是数据的量、数据的复杂性和数据的变化速度。

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

数据密集型的应用程序通常是基于标准的、可以提供通用功能的模块而构建的。比如，许多应用需要：

• Store data so that they, or another application, can find it again later (databases)

存储数据，这样之后，应用程序自己或者其他应用程序可以来获取数据 （数据库）

• Remember the result of an expensive operation, to speed up reads (caches)

 记住一次繁复操作的结果，这样可以加速之后的读 （缓存）

• Allow users to search data by keyword or filter it in various ways (search indexes)

 允许用户通过关键字进行搜索，或可以使用多种方法进行数据过滤 （搜索索引）

• Send a message to another process, to be handled asynchronously (stream processing)

 给另一个进程发送消息来做异步处理 （流处理）

• Periodically crunch a large amount of accumulated data (batch processing)

周期性地压缩大量累积数据（批处理）

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 building 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.

如果这听起来很痛苦，那仅仅是因为这些数据系统在抽象化上做的太成功：我们一直在使用却没有好好思考。当创建一个应用程序的时候，大多数工程师都不会想从头写一个新的数据存储引擎，因为数据库是一个非常好的现成工具。

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.

但是现实不是如此简单。不同的数据库有不同的特性，而不同的应用程序也有不同的需求。比如，我们有许多方法来做缓存，来创建搜索索引。当构建一个应用程序的时候，我们需要找出哪个工具，哪个方法是最适合我们手头的任务。另外，当单一工具无法满足我们的需求时，如何整合其他的工具来共同完成也是一个难题。

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.

这本书将会为你展示数据系统的原理和实例，并且如何使用它们来创建自己的数据密集型应用程序。我们将会探讨不同工具的共同性，它们的区别，以及它们如何实现它们各自的特点。

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.

在这一章节，我们将会先探讨可靠的、可伸缩的、可维护的数据系统的基本原理。我们会说明这些概念的含义，概述一些相关的思考方法，并为后续章节打一些基础。在后续的章节中，我们会层层深入，看一看在创建数据型密集型应用程序时需要考虑的不同的设计决策。

**Thinking About Data Systems**

想一想数据系统

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.

我们通常认为数据库、队列、缓存等是属于不同的工具。虽然数据库和消息队列从表面上看有一些共同性（都是存储数据一段时间），但是它们有着不同的访问模式，这也意味着它们的性能特性是不同的，因此有着非常不同的实现方式。

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

那么为什么我们要把它们统统放在数据系统这个范畴之下呢？

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.

最近几年，涌现了非常多的新工具，可以用来存储和处理数据。它们为不同的使用场景做了优化，也就是说，它们不再适合用传统范畴进行划分。比如，一些数据库也可以用来作消息队列（Redis），一些消息队列也可以有数据库一样的数据持久性。因此，划分边界正在变得模糊起来。

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.

其次，越来越多的应用程序不是单个工具就可以满足其数据存储和处理的需求。相反，工作被分解成可以使用单一工具就可以有效完成的子任务，当然我们需要使用代码将不同的工具整合在一起。

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).

举个例子，你有一个应用程序管理的缓存层（Memcached或类似的），或者从主数据库分离的全文索引服务器（Elasticsearch 或 Solr），那么通常需要代码来完成缓存和索引与主数据库的同步。图1-1展示了一种可能性（我们将在后续的章节中深入讨论）。

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.

当你结合几个工具来提供服务，那么这个服务的接口或应用接口（API）通常不会向用户展示实现细节。现在，你已经基本上创建了一个新的、特殊用途的、由小型通用组件组成的数据系统。你的数据系统可以提供这样的保证：比如，缓存被判失效或依据写来更新，这样用户可以看见一致的数据。你现在不仅仅是一个应用程序的开发人员，也是数据系统的设计者。

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?

如果你打算设计一个数据系统或服务，那么将会遇到很多棘手的问题。比如，当内部出现问题的时候，你怎么确保数据依然是正确完整的？当部分系统出现性能下降，你怎么为客户提供持续良好的性能？当负载变大的时候，你怎么扩展？一个好的服务API应该是什么样子的？

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.

有许多因素会影响数据系统的设计，比如，参与人员的技能和经验，对遗留系统的依赖性，交付所需的时间，你公司对于不同风险的承受程度，监管约束等等。这些因素依据情况会不同。

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

这本书中，我们会着重探讨对大多数软件系统都非常重要的3个关注点：

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 software faults, and even human error). See “Reliability” on page 6.

可靠性

即使遇到故障（硬件或软件，甚至是人为失误），系统也应该可以继续正常工作（以期望的性能实现正确的功能）。详情见第6页的可靠性。

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.

可伸缩性

当系统在数据量，流量或复杂性上出现增长，那么系统应该有合理的方法来处理这种增长。详情见第10页的可伸缩性。

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 “Maintainability” on page 18.

可维护性

随着时间的推移，许多不同的人员会致力于这个系统（研发和运维人员都会维护目前的系统，并使系统适用于新的场景），他们应该是可以快速有效地完成这些工作。详情见第18页的可维护性.

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.

这些名词常常是在没有清楚地理解它们的意思的情况下出现的。出于对成熟的工程学的兴趣，我们将在本章的其余部分中探索思考可靠性、可伸缩性和可维护性的方法。然后在接下去的章节中，我们将对实现这些目标的不同技术、架构和算法一探究竟。

**Reliability**

Everybody has an intuitive idea of what it means for something to be reliable or unreliable.

可靠性

每个人都有一个直观的想法，那就是什么是可靠的或不可靠的。

For software, typical expectations include:

对于软件，典型的期望包括：

• The application performs the function that the user expected.

应用程序可以完成用户期望的功能。

• It can tolerate the user making mistakes or using the software in unexpected ways.

它可以允许用户犯错误或以没有想到的方式来使用。

• Its performance is good enough for the required use case, under the expected load and data volume.

在预期负载和数据量下，必须的场景的性能都是没有问题的。

• The system prevents any unauthorized access and abuse.

该系统防止任何未经授权的访问和滥用。

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.”

如果所有这些都意味着“运行正常”，那么我们可以把可靠性大致理解为“即使事情出了差错，也可以继续正确地工作。

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.

可能出错的东西称为故障，而预期故障并能够处理它们的系统称为容错或弹性系统。前一个词可能有点误导，因为这意味着我们可以让系统容忍所有可能的故障，这在现实中是不可行的。 如果整个地球（以及所有的服务器）被黑洞吞噬了，容忍这个错误需要把网络托管在太空（希望可以得到这样的预算批准）。所以说容忍某些类型的故障才是有意义的。

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.

请注意，故障与运行失败是不一样。故障通常定义为系统的一个组件偏离其应有的功能，而运行失败是指系统作为一个整体停止向用户提供所需的服务。不可能将故障率降低到零，因此通常，我们最好设计好容错机制来防止故障导致系统运行失败。在这本书中，我们介绍了几种技术来建立可靠的系统。

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.

在这种容错系统中，通过故意触发故障来增加故障率是有意义的（例如，没有预警地随机杀死某个进程），这似乎与我们的直觉恰恰相反。许多严重的bug实际上是由于错误处理不当造成的。你可以通过故意诱发故障，来确认容错机制是否可以持续地运行和被测试。这个方法可以确保当故障发生的时候，它们可以被准确地处理。Netflix Chaos Monkey就是其中一个方法。

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.

 虽然我们通常更倾向于容忍故障而不是预防故障，但有些情况下预防故障胜于从故障中恢复 （比如，不存在恢复方法）。安全问题就是这样。举个例子，如果攻击者破坏了系统并获得对敏感数据的访问，则我们无法取消该事件。但是，这本书主要涉及可以恢复的各种故障，将在接下来的章节中细述。

**Hardware Faults**

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.

 硬件故障

当我们想到系统故障的原因时，很快就会想到硬件故障： 硬盘崩溃，RAM故障，电网停电，有人拔出错误的网络电缆。任何有大型数据中心工作经验的人都可以告诉你，当你有很多机器时，这些事情总是会发生。

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.

硬盘一般具有大约10至50年的平均失效前时间（MTTF）。因此，在一个具有10000个磁盘的存储集群上，我们应该可以预料到平均每天都有一个磁盘坏掉。

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 component 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.

我们的第一反应通常是给各个硬件组件增加冗余，以降低系统的故障率。磁盘可以配置成RAID，服务器可以具有双电源和热插拔CPU，以及数据中心可能使用电池和柴油发电机来作为备用电源。当一个部件坏掉了，在其更换期间，冗余部件可以代替它工作。这种方法不能完全防止由硬件问题导致的系统运行失败，但是可以理解的是，这个方法通常可以保持机器运行多年不间断。

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.

过去，对大多数应用程序来说，硬件组件有冗余就足够用了，因为有冗余的存在，单个机器完全坏掉是相当罕见的。只要你能够相当快速地将备份恢复到新机器上，那么大多数应用程序在故障情况下的停机时间并不是灾难性的。因此，只有少数应用程序需要多机冗余，因为对这些应用程序来说，高可用性是绝对必要的。

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.

然而，随着数据量和应用程序计算需求的增加，越来越多的应用程序开始使用大量的机器，这相应地增加了硬件故障的发生率。此外，在某些云平台，如Amazon Web Services（AWS），虚拟机在没有警告的情况下变得不可用是相当常见的，因为该平台认为灵活性和弹性比单机可靠性更为重要。

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，除了硬件冗余之外，我们可以优先使用软件容错技术来处理整台机器的故障。这样的系统也具有操作优势：比如，一个使用单个服务器的系统需要停机时间来重新启动机器（如，操作系统安装安全补丁），而一个可以容忍停机的系统可以让一台机器一台机器地安装补丁，这样整个系统就不会暂停（滚动安装；参见第4章）。

**Software Errors**

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.

 软件错误

我们通常认为硬件故障是随机的并且彼此独立：一台机器的磁盘故障并不意味着另一台机器的磁盘也会故障。但也可能存在弱相关性（例如共同的原因，比如服务器机架中的温度），否则不太可能同时出现大量硬件组件故障。

 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:

 另一类故障是系统内的系统错误。这样的故障更难预测，并且由于它们是跨节点相关的，因此它们往往会比互不相关的硬件故障导致更多的系统问题。举例来说：

• 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].

 一个软件bug，它会导致应用服务器的每一个实例在给定一个特定的错误输入时全部崩溃。例如，2012年6月30日的闰秒，由于Linux内核中的bug，导致许多应用程序同时挂起。

• A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth.

 使用某些共享资源（CPU时间、内存、磁盘空间或网络带宽）的失控进程。

• A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.

 系统依赖的服务减慢，变得无响应，或开始返回损坏的结果。

• Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults [10].

 级联故障，其中一个组件中的小故障触发另一个组件中的故障，这反过来又触发其他故障。

 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].

导致这类软件故障的bug通常长时间处于休眠状态，直到它们被一些不寻常的情况触发。在那些情况下，软件会对其环境做出某种假设，尽管这种假设通常是正确的，但是由于某种原因，这种假设最终不再是正确的。

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].

软件中的系统故障没有快速解决的方法。但是许多小事都可能有所帮助：仔细考虑系统中的假设和交互；彻底的测试；进程隔离；允许进程崩溃和重新启动；测量、监控和分析生产环境中的系统行为。如果希望系统提供某种保证（例如，在消息队列中，传入消息的数量等于传出消息的数量），那么它可以在运行时不断检查自身，如果发现不一致则发出警报。

**Human Errors**

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].

人为错误

人类设计和构建软件系统，同时保持系统运行的运维人员也是人。即使他们都想做到最好，但人类依然被认为是不可靠的。例如，一项大型互联网服务的研究发现运维人员的配置错误是导致停机的主要原因，而硬件故障（服务器或网络）仅占停机原因的10%-25%。

How do we make our systems reliable, in spite of unreliable humans? The best systems combine several approaches:

尽管人类不可靠，那么我们如何使我们的系统可靠？ 最好的系统通常结合多种方法：

• 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.

设计系统时考虑如何最小化错误几率。例如，设计良好的抽象、API和管理接口使“正确的事情”变得容易，并阻止“错误的事情”。然而，如果接口过于严格，人们将绕过它们工作，从而它们的好处就大打折扣，因此这需要一个巧妙的平衡。

• 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.

去掉人们犯错误最多的地方和可能导致失败的地方。特别是，提供功能齐全的非生产的沙盒环境，人们就可以使用真实数据安全地探索和测试，而不影响真实用户。

• 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.

全面测试，从单元测试到整个系统集成测试和手工测试。自动化测试已经被广泛使用并且被掌握地很好，尤其适用于在正常操作中很少出现的个例情况。

• 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).

为了最小化故障造成的影响，需要允许快速和方便地从人为错误中恢复。例如，使其快速回滚配置的更改，逐步推出新代码（使任何意外的bug只影响用户中的一小部分），并提供重新计算数据的工具（以防旧计算不正确）。

• 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.

建立详细和清晰的监控，如性能指标和错误率。在其他工程学科中，这被称为遥测。（一旦火箭离开地面，遥测对于跟踪正在发生的事情和故障理解至关重要）。监视可以显示预警信号并允许我们检查是否有任何假设或约束被违反。当问题发生时，那些指标在诊断问题上可能是无价之宝。

• Implement good management practices and training—a complex and important aspect, and beyond the scope of this book.

 良好的管理和培训——一个复杂而重要的方面，不过超出了本书的范围。

**How Important Is Reliability?**

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.

可靠性有多么重要？

可靠性不仅仅适用于核电站和空中交通控制软件，更普通的应用程序也需要可靠地工作。商业应用程序中的bug导致生产无法正常运作（如果报告中的数据是错误的，那么还需要承担法律风险），并且电子商务网站的不可访问可能需要付出巨大的代价，比如收入的损失和声誉的损害。

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?

即使在“非关键”应用程序中，我们对用户也负有责任。比如，一个家长把他们孩子的照片和视频存储在你的照片应用程序中。如果数据库突然遭到破坏，他们会有什么感觉？他们知道如何从备份中恢复吗？

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.

在某些情况下，我们可能需要选择牺牲可靠性来降低开发成本（例如，在为未经验证的市场开发原型产品时），或者运营成本（例如，利润率非常小的服务）。但是我们应该非常清楚什么时候需要精打细算。

**Scalability**

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.

可伸缩性

即使一个系统今天可以可靠地工作，但这并不意味着它将来一定会可靠地工作。性能下降的一个常见原因是负载增加：也许系统从10000个并发用户增长到100000个并发用户，或者从100万个增长到1000万个。也许它正在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?”

可扩展性是我们用来描述系统处理负载增加的能力的术语。然而，请注意，它不是一个描述系统的一维标签。“X是可伸缩的”或“Y不可伸缩的”是没有意义的。相反，讨论可伸缩性意味着要思考诸如“如果系统以特定的方式增长，我们如何应对？“和”我们如何添加计算资源来处理额外的负载？“之类的问题。

**Describing Load**

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.

描述负载

首先，我们需要简明地描述系统的当前负载；只有这样，我们才能讨论增长问题（如果负载加倍，会发生什么？）。负载可以用几个数字来描述，我们称之为负载参数。参数的最佳选择取决于系统的架构：可能是每秒向Web服务器的请求、数据库的读写比、聊天室中同时活跃的用户数、缓存命中率等等。也许平均情况对你来说是重要的，或者你的瓶颈是由少数极端情况所决定的。

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:

为了更具体地描述这个想法，我们使用Twitter 2012年11月发布的数据作为一个例子 。Twitter的两个主要业务是：

Post tweet

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

发布tweet

用户可以向他们的粉丝发布新消息（平均4.6K请求/秒，峰值超过12K请求/秒）。

 Home timeline

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

主页时间线

用户可以查看他们所粉的人发布的tweets（300 K请求/秒）

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-outii—each user follows many people, and each user is followed by many people. There are broadly two ways of implementing these two operations:

简单地处理每秒12000次写入（发布tweet的峰值速率）相当容易。然而，Twitter的扩展主要难题不是因为tweet量，而是因为粉丝扇出——每个用户粉了许多人，每个用户也被许多人粉了。大致有两种方法实现这两种操作：

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:

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

发布一条tweet只需将新的tweet插入到一个全局的tweet集合中。当用户请求查看他的主页时间线时，找到他粉的所有人和这些用户所有的tweet，然后合并它们（按时间排序）。在关系数据库中，如图1-2所示，你可以编写一个查询，例如：

2. 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.

为每个用户的主页时间线维护一个缓存，如每个接收者都有一个tweet邮箱（见图1-3）。当用户发布tweet时，查找粉该用户的所有人，并将新tweet插入到那些主页时间线缓存中。那么读取主页时间线的请求是快速的，因为它的结果已经提前计算出来了。

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的第一个版本使用方法1，但是系统很难跟上主页时间线查询的负载，所以公司切换到方法2。方法2工作得比较好，因为发布tweet的平均速率几乎比主页时间线读取速率低两个数量级，所以在这种情况下，最好在写时做更多的工作，在读时做更少的工作。

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.

然而，方法2的缺点是发布一条推特现在需要大量额外的工作。平均而言，一条tweet被传递给大约75个粉丝，所以每秒4.6k条tweet变成每秒345k条需要写到主页时间线的缓存。但是这个平均值忽略了一个事实，那就是每个用户的粉丝数量差别是很大的，一些用户拥有超过3000万的粉丝。这意味着单个tweet可能导致超过3000万主页时间线的写。所以，Twitter试图在五秒内向粉丝发送tweet是一个重大挑战。

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的示例中，每个用户的粉丝分布（可能需要使用那些用户发布的频率来加权）是讨论可伸缩性的关键负载参数，因为它决定了扇出负载。你的应用程序可能有着不同的特性，但你可以应用类似的方法来推理你自己的负载。

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例子的最后一个转折点：虽然方法2被很好地实现了，Twitter正在转向两种方法的混合使用。大多数用户的tweet在发布时继续被扇出到主页时间线上，但是少数拥有大量粉丝（如名人）的用户被排除在这个扇出之外。当需要读取用户粉的名人的tweet时，这些tweet都是独立获取的，然后与用户的主页时间线合并，就像在方法1中一样。这种混合方法能够始终如一地表现出良好的性能。在我们讨论了更多的技术基础之后，我们将在第12章重温这个例子。

**Describing Performance**

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:

描述性能

一旦你开始描述系统负载，你就可以研究当负载增加时会发生什么。你可以用两种方式：

• 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?

 当你增加负载参数但保持系统资源（CPU、内存、网络带宽等）不变时，那么对系统的性能有何影响？

• When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?

当你增加负载参数时，如果想要保持性能不变，需要增加多少资源？

Both questions require performance numbers, so let’s look briefly at describing the performance of a system.

这两个问题都需要性能参数，所以让我们简单地描述一下系统的性能。

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.

在Hadoop这样的批处理系统中，我们通常关心吞吐量——每秒可以处理的记录数量，或者在一定大小的数据集上运行作业所需的总时间。在线系统中，通常更重要的是服务的响应时间，即客户端发送请求和接收响应之间的时间。

**Latency and response time**

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

延迟和响应时间通常是同义的，但它们并不相同。响应时间是客户端所看到的：除了处理请求的实际时间（服务时间）之外，还包括网络延迟和队列延迟。延迟是请求等待处理的时间，在此期间它是休眠的，等待处理。

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.

 即使你一遍遍只做同样的请求，你会在每一次尝试得到一个略有不同的响应时间。在实际中，处理各种请求的系统响应时间可以是不一样的。因此，我们不能把响应时间看成是一个单一的数字，而是作为一个可以测量的值的分布。

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1-4中，每个灰色条表示对服务的请求，并且其高度表示请求花费的时间。大多数请求都是相当快的，但偶尔会出现需要更长时间的异常值。也许缓慢的请求本质上就更复杂，比如，它们要处理更多的数据。但是，即使在有些情况中，你认为所有请求都应该花费相同的时间，你也会得到不同的时间，比如原因有：上下文切换到后台进程可能会引入随机的附加延迟、网络分组的丢失和TCP重传、垃圾收集暂停、页面故障迫使从磁盘中读取，服务器机架的机械振动，或许多其他原因。

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.

我们经常可以看到报告中的服务平均响应时间。（严格地说，“平均值”并不指任何特定的公式，但在实践中通常被理解为算术平均值：给定n个值，将所有值加起来，除以n）但是，如果你想知道“具有代表性的”响应时间，平均值并不是一个很好的度量。因为它没有告诉你有多少用户实际经历过延迟。

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.

通常百分位数是更好的度量。如果将响应时间从最快到最慢进行排序，则中间的值是中位数：例如，如果响应时间中位数是200ms，则意味着一半的请求在不到200ms内返回，一半的请求花费的时间比这长。

 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%.

如果你想知道用户通常需要等待多长时间，那么中位数就是一个很好的度量：一半的用户请求是在低于响应时间中位数，而另一半需要比中位数更长的时间。中位数也被称为第五十百分位数，有时缩写为P50。注意，中位数是指单个请求。如果用户在会话的过程中发出多个请求，或者多个资源包含在单个页面中，那么其中至少有一个比中位数慢得多的概率远大于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.

 为了找出异常值有多大影响，你可以看看更高的百分位数：比如常见的95、99和99.9百分位数（缩写p95、p99和p999）。它们是响应时间的阈值，其中请求的95%、99%或99.9%比特定阈值更快。例如，如果第95个百分点的响应时间是1.5秒，这意味着100个请求中的95个请求花费不到1.5秒，而100个请求中的5个请求花费1.5秒或更多。如图1-4所示。

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].

高百分位数的响应时间（也称为尾部延迟）非常重要，因为它们直接影响到用户的服务体验。例如，亚马逊使用99.9百分位数来描述内部服务的响应时间需求，即使它只影响1000个请求中的1个。这是因为请求最慢的客户往往是那些拥有最多数据的客户，因为他们已经购买了很多东西，也就是说，他们是最有价值的客户。确保他们可以快速地访问网站是让这些客户满意的重要方法。亚马逊也观察到增加100ms的响应时间会减少销售额1%，而其他报告也显示，慢了1秒会降低顾客满意度16%。

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，优化99.99百分位数（10000个请求中最慢的一个）被认为太昂贵，并且不能为亚马逊提供足够的好处。在非常高的百分位数上减少响应时间是困难的，因为它们很容易受到你控制之外的随机事件的影响，好处也在减少。

 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，百分位经常用于服务水平目标（SLO）和服务水平协议（SLA）中，这些协议定义了期望的服务性能和可用性。SLA可以声明，如果服务的响应时间中位数小于200ms和第99个百分位小于1s（如果响应时间更长，那么它可能也会下降），则该服务被认为是可用的，并且该服务需要至少99.9%的时间是可用的。这些度量为客户设定期望，并允许客户在不满足SLA的情况下要求退款。

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.

队列延迟常常占了高百分位数的响应时间的很大一部分。由于服务器只能并行处理少量的事件（例如，受其CPU内核数量的限制），所以只需要少量的慢速请求就可以阻塞后续请求的处理，这种影响有时称为队头阻塞（head-of-line block）。即使这些后续请求在服务器上处理得很快，但是由于先前的等待时间，客户端将看到缓慢的总体响应时间。由于这种影响，我们有必要在客户端测量响应时间。

When generating load artificially in order to test the scalability of a system, the loadgenerating 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].

当为了测试系统的可伸缩性而人为生成负载时，生成负载的客户端需要持续发送与响应时间无关的请求。如果客户端在发送下一个请求之前需要等待前一个请求完成，那么该做法使得测试场景中的队列比在实际中的短，从而我们无法得到真实的测量数据。

**Percentiles in Practice**

 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 backend calls, and so a higher proportion of end-user requests end up being slow (an effect known as tail latency amplification  [24 ]).

 实践中的百分位数

在后端服务中（作为单个终端用户请求的一部分，后端服务被多次调用），高百分位数变得尤为重要。即使并行地进行调用，终端用户的请求仍然需要等待最慢的调用完成。如图1-5所示，只需要一个慢的调用就可以使整个终端用户请求慢下来。即使只有一小部分后端调用是缓慢的，但是如果终端用户请求需要多个后端调用，则遇上缓慢调用的可能性就会增加，因此大部分终端用户的请求最后都是缓慢的（称为尾部延迟放大效应）

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.

如果想要把响应时间百分位数添加到服务的监视仪表板上，则需要持续有效地计算它们。例如，你可能希望有一个窗口实时展现过去10分钟内的请求响应时间，那么每1分钟，你需要计算中位数和各种百分位数，并绘制出这些计算值。

The na.ve 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 ].

简单的实现是在时间窗口中保存所有请求的响应时间列表，并且每分钟对列表进行排序。如果这对你来说太低效，那么有一些算法可以以最小的CPU和内存成本来计算近似的百分位数，例如正向衰减、t-摘要或HdrHistogram。请注意，平均百分位数（例如，降低时间分辨率或整合来自几台机器的数据）在数学上是没有意义的——整合响应时间的正确方法是添加直方图。

**Approaches for Coping with Load**

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?

处理负载的方法

既然我们已经讨论了用于描述负载的参数和用于测量性能的度量，我们就可以认真地讨论可伸缩性了：即使负载参数增加了一些量，我们如何保持良好的性能？

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.

适合于一个负载级别的系统架构不太可能处理10倍的负载。如果你的服务正快速增长，那么每增加一个数量级的负载时，你可能需要重新考虑系统架构——或者可能比这更频繁。

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 powerfulmachines can still be simpler and cheaper than a large number of small virtual machines.

人们经常谈论垂直扩展（迁移到更强大的机器上）和水平扩展（跨多个较小的机器来分摊负载）的区别。跨多个机器分摊负载也称为无共享架构。可以在单台机器上运行的系统通常更简单，但是高端机器可能非常昂贵，因此非常高负载的工作常常无法避免横向扩展。实际上，好的架构通常包含一种实用的混合方法：例如，使用几个相当强大的机器仍然可以比大量小型虚拟机更简单和更便宜。

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).

有些系统是有弹性的，这意味着当检测到负载增加时，它们可以自动添加计算资源，而其他系统则会使用手动缩放（人工分析容量并决定向系统中添加更多机器）。如果负载是不太可能预测的，那么弹性系统就是有用的。但是手动缩放的系统更简单，并且可能有更少的操作意外（参见第209页的“重新平衡分区”）。

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.

虽然分发无状态服务到多台机器上相当简单，但是将使用有状态数据的系统从单个节点迁移到分布式环境可能带来许多额外的复杂性。因为这个原因，以前的通常的做法是将数据库保持在单个节点上（垂直扩展），直到扩展成本或高可用性迫使你使用分布式。

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.

随着用于分布式系统的工具和抽象变得越来越好，这种通常的做法可能会改变，至少对于某些类型的应用程序是这样。可以想象，将来会默认使用分布式数据系统，即使不处理大量数据或大量流量。在本书的其余部分中，我们将介绍多种分布式数据系统，讨论它们在可伸缩性方面的作用，以及在易用性和可维护性方面的作用。

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.

大规模系统的架构通常是为应用定制的——没有通用的、一刀切的可伸缩系统架构（非正式地称为魔术伸缩酱）。可能的问题是读取量、写入量、要存储的数据量、数据的复杂性、响应时间的需求、访问模式，或者（通常）所有这些的某种混合再加上其他各种问题。

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，设计成每秒处理100000个请求（每个请求的大小为1kB）的系统看起来与为每分钟3个请求（每个请求的大小为2GB）而设计的系统非常不同，即使两个系统具有相同的数据吞吐量。

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.

适用于特定应用程序而且伸缩良好的系统架构是基于以下假设构建的：哪些操作是常见的，哪些操作是罕见的（负载参数）。如果这些假设被证明是错误的，那么扩展的努力就最大程度地浪费了，最坏的情况将是适得其反。在早期或未经验证的产品中，能够快速地迭代产品特性通常比扩展到某个假设的未来负载更重要。

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.

尽管可伸缩系统架构是特定于应用程序的，但是它们通常由通用模块构建，然后以熟悉的模式排列。在本书中，我们将讨论这些模块和模式。

**Maintainability**

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.

 可维护性

众所周知，软件的大部分成本不在于它的初期开发，而在于它持续的维护——修复bug、保持系统运行、调查故障、使它适应新的平台、为新的使用场景修改它，偿还技术债务和增加新功能。

Yet, unfortunately, many people working on software systems dislike maintenance of so-called legacy systems—perhaps it involves fixing other people’s mistakes, or work‐ing 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.

然而，不幸的是，许多从事软件系统工作的人不喜欢维护所谓的遗留系统——也许是因为它涉及修复其他人的错误，或者使用已经过时的平台，或者迫使他们做从未打算做的事情。每一个遗留系统本身都是有一些问题的，因此很难给出一般的建议来处理它们。

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:

然而，我们可以并且应该以这样的方式来设计软件：希望它能够最小化维护过程中的痛苦，从而避免自己创建遗留软件。为此，我们将特别关注软件系统的三个设计原则：

Operability

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

可操作性

使运维团队易于保持系统平稳运行。

Simplicity

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.)

简单性

新工程师可以从系统中去除尽可能多的复杂性来快速理解系统。（注意，这与用户界面的简单性不一样）。

Evolvability

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.

可进化性

工程师将来更容易对系统进行更改，在需求更改时将其适应于未想到的场景。也称为可扩展性、可修改性或可塑性。

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.

正如以前的可靠性和可扩展性，没有简单的解决方案来实现这些目标。相反，我们将尝试思考具有可操作性、简单性和可进化性的系统。

**Operability: Making Life Easy for Operations**

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.

 可操作性：让运维更轻松

有人建议“良好的运维通常可以解决不良（或不完整）软件所带来的限制，而不良的运维会使得好的软件无法可靠地运行”。虽然运维的某些方面可以而且应该是自动化的，但是首先要人类来设置自动化并确保其正常工作。

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

运维团队对于保持软件系统平稳运行至关重要。一个优秀的运维团队通常负责以下事项，甚至更多：

• Monitoring the health of the system and quickly restoring service if it goes into a bad state

 监控系统的运行状况，并在服务进入不良状态时快速恢复服务

• Tracking down the cause of problems, such as system failures or degraded performance

 追踪问题的原因，例如系统故障或性能下降

• Keeping software and platforms up to date, including security patches

 使软件和平台保持在最新版本，包括安全补丁

• Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage

 密切关注不同系统是如何相互影响的，以避免那些会造成破坏、有问题的变更

• Anticipating future problems and solving them before they occur (e.g., capacity planning)

 预测未来可能发生的问题并在问题发生之前解决（例如，容量规划）

• Establishing good practices and tools for deployment, configuration management, and more

 为部署，配置管理等建立良好的实践和工具

• Performing complex maintenance tasks, such as moving an application from one platform to another

 执行复杂的维护任务，例如将应用程序从一个平台迁移到另一个平台上

• Maintaining the security of the system as configuration changes are made

 当配置在更改时，保持系统的安全

• Defining processes that make operations predictable and help keep the production environment stable

 定义流程， 使得操作可预测并有助于保持生产环境的稳定

• Preserving the organization’s knowledge about the system, even as individual people come and go

 保持组织对系统的了解，即使有新人加入或有人离职

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:

 良好的可操作性意味着让日常工作变得容易，这样就允许运维团队集中精力在更重要的事情上。数据系统可以做各种事情让日常工作变得容易，包括：

• Providing visibility into the runtime behavior and internals of the system, with good monitoring

通过良好的监控，为系统的行为和系统的内部情况提供可视化

• Providing good support for automation and integration with standard tools

 为自动化和标准工具之间的集成提供良好的支持

• Avoiding dependency on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted)

 避免对单个机器的依赖（允许停机进行维护的同时系统可以继续不间断地运行）

• Providing good documentation and an easy-to-understand operational model (“If I do X, Y will happen”)

 提供良好的文档和一个易于理解的操作模型（“如果我做X，Y会发生”）

• Providing good default behavior, but also giving administrators the freedom to override defaults when needed

 提供良好可靠的默认值，但也允许管理员在需要时使用其他值来覆盖默认值。

• Self-healing where appropriate, but also giving administrators manual control over the system state when needed

 适当的自我恢复机制，但也允许管理员在有必要时手动控制系统。

• Exhibiting predictable behavior, minimizing surprises

 展现可预测的行为，尽量减少意外

**Simplicity: Managing Complexity**

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].

 简单性：管理复杂性

小型软件项目可以拥有非常简单且富有表现力的代码，但是随着项目越来越大，它们常常变得非常复杂和难以理解。这种复杂性减慢了所有系统的工作人员的工作进度，进一步增加了维护成本。一个复杂的软件项目有时被描述成一个巨大的泥球。

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].

 我们可以看到复杂性所表现出来的各种症状：状态空间的爆满、模块的紧密耦合、复杂难分的依赖关系、不一致的命名和术语、意在解决性能问题的高招、用于解决别处问题的特殊外壳等等。这个话题已经被讨论了很多很多。

When complexity makes maintenance hard, budgets and schedules are often overrun. 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.

 当复杂性让维护变得困难时，预算经常超支和任务经常不能按时完成。在复杂的软件中，进行更改而引入bug的风险也更大：当系统对于开发人员来说很难理解和推测时，隐藏的假设、意外的结果以及没有预想到的交互就往往容易被忽视。相反，降低复杂性极大地提高了软件的可维护性，因此简单化应该是我们建立的系统的重要目标。

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.

 让系统变简单并不一定意味着减少其功能，也可以意味着消除偶然的复杂性：如果复杂性不是软件所解决的问题（如用户所见）中所固有的，而是仅由实现过程引起的，Moseley和Mark认为这样的复杂性是偶然的。

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 facade. 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.

 我们用来消除种偶然复杂性的最好工具之一是抽象。一个好的抽象可以把大量实现细节隐藏在一个直接、简单易懂的接口背后。一个好的抽象也可以用于不同的应用程序。这种重用性不仅比多次重新实现类似的东西更有效，而且还会带来更高质量的软件，因为提升抽象组件可以让使用它的所有应用程序都受益。

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，高级编程语言是隐藏了机器代码、CPU寄存器和系统调用的抽象。SQL也是一个抽象，它隐藏了复杂的磁盘和内存数据结构、来自其他客户端的并发请求以及崩溃后的不一致性。当然，当我们在使用高级语言编程时，我们仍然在使用机器代码，只是不直接使用它，因为编程语言的抽象使我们不必考虑它。

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.

 然而，找到好的抽象是非常困难的。在分布式系统领域中，虽然有许多好的算法，但是如何将它们打包成抽象，以帮助我们将系统的复杂性保持在可管理的范围，还不太清楚。

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.

 贯穿本书，我们将睁大眼睛寻找那些好的抽象，那些可以帮助我们将大型系统的部分提取出来形成定义良好的、可重用的组件的抽象。

**Evolvability: Making Change Easy**

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.

 可进化性：让变更容易起来

 你的系统需求不可能永远不变。它们常常处于不断变化的过程中：你了解了新的事实，以前未预料的使用场景出现了，业务优先级改变了，用户提出了新功能，新平台替换旧平台，法律或法规要求的变更，系统的增长迫使系统架构的改变，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.

 在组织流程方面，敏捷工作模式提供了一个适应变化的框架。敏捷社区还开发了一些技术工具和模式，这些工具和模式对于在频繁变化的环境中开发软件很有帮助，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?

关于这些敏捷技术的大多数讨论都集中在相当小的本地范围中（比如，同一应用程序中的一些源代码文件）。在这本书中，我们将在一个更大的数据系统级别上寻找提高敏捷性的方法，它可能由几个具有不同特性的不同应用程序或服务组成。例如，你打算把方法1到方法2都结合到首页时间线上（第11页上的“Describing Load”），那么你将如何重构Twitter的架构？

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].

 你是否可以轻松地修改数据系统，并使其适应不断变化的需求，这与其简单性和抽象性密切相关：简单和易于理解的系统通常比复杂的系统更容易修改。但由于这是一个非常重要的想法，我们将使用一个不同的词来来指代数据系统级的敏捷性：可进化性。

**Summary**

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.

 总结

在本章中，我们探讨了一些关于数据密集型应用程序的基本概念。这些将指引我们学习本书的其余部分，在那里我们将深入了解这些技术细节。

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.

应用程序必须满足各种需求才能有价值。我们有功能需求（应该做什么，例如允许以各种方式存储、访问、搜索和处理数据）和一些非功能需求（一般属性如安全性、可靠性、合规性、可伸缩性、兼容性和可维护性）。在本章中，我们详细讨论了可靠性、可扩展性和可维护性。

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.

可靠性意味着即使故障发生，系统也能正常工作。故障可能存在于硬件（通常是随机的和独立的）、软件（bug通常是系统化的，很难处理）和人为（人们时常不可避免地会犯错误）。容错技术可以避开来自终端用户的某些类型的故障。

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.

可伸缩性意味着即使负载增加，我们也有办法保持性能良好。为了讨论可伸缩性，我们首先需要定量描述负载和性能。作为描述负载的示例，我们大致看了Twitter的主页时间线的问题，以及如何使用响应时间百分位数来度量性能。在可扩展系统中，你可以增加处理能力，这样在高负载下，依然可以保持系统的可靠性。

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.

可维护性包含许多方面，但本质上是为了让研发和运维团队更方便有效地工作。好的抽象可以帮助降低复杂性，使系统更易于变更和适应新的场景。良好的可操作性意味着可以把系统健康状况简单明了的可视化，并有办法进行有效地管理。

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，目前没有简单解决方案可以让应用程序可靠、可扩展和可维护。然而，有一些模式和技术在不同的应用程序中不断被使用。在接下来的几章中，我们将看看数据系统的一些例子，并分析它们是如何朝着这些目标努力的。

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.

在本书的第三部分中，我们将研究由几个组件构成的系统模式，例如图1-1中的组件。