一、 可靠、可扩展与可维护的应用系统

可靠性：容忍硬件、软件失效，人为错误

可扩展性：评测负载与性能，延迟百分位数，吞吐量

可维护性：可运维，简单与可演化性

现在许多新型应用都属于数据密集型（data-intensive），而不是计算密集型（compute-intensive）。对于这些类型应用，CPU 的处理能力往往不是第一限制性因素，关键在于数据量、数据的复杂度及数据的快速多变性。

数据密集型应用通常也是基于标准模块构建而成，每个模块负责单一的常用功能。例如，许多应用系统都包括以下模块：

- 数据库：用以存储数据，这样之后应用可以再次访问
- 高速缓存：缓存那些复杂或操作代价昂贵的结果，以加快下一次访问
- 索引：用户可以按关键字搜索数据并支持各种过滤
- 流式处理：持续发送消息至另一个进程，处理采用异步方式
- 批处理：定期处理大量的累积数据

确实已有很多数据库系统， 但因为需求和设计目标的差异， 个中精妙都不尽相同。 缓存和索引方案与之类似。 因此在构建某个特定应用时， 我们总是需要弄清楚哪些组件、 哪些方法最适合自己， 并且当单个组件无法满足需求而必须组合使用时， 总要面临更多的技术挑战。

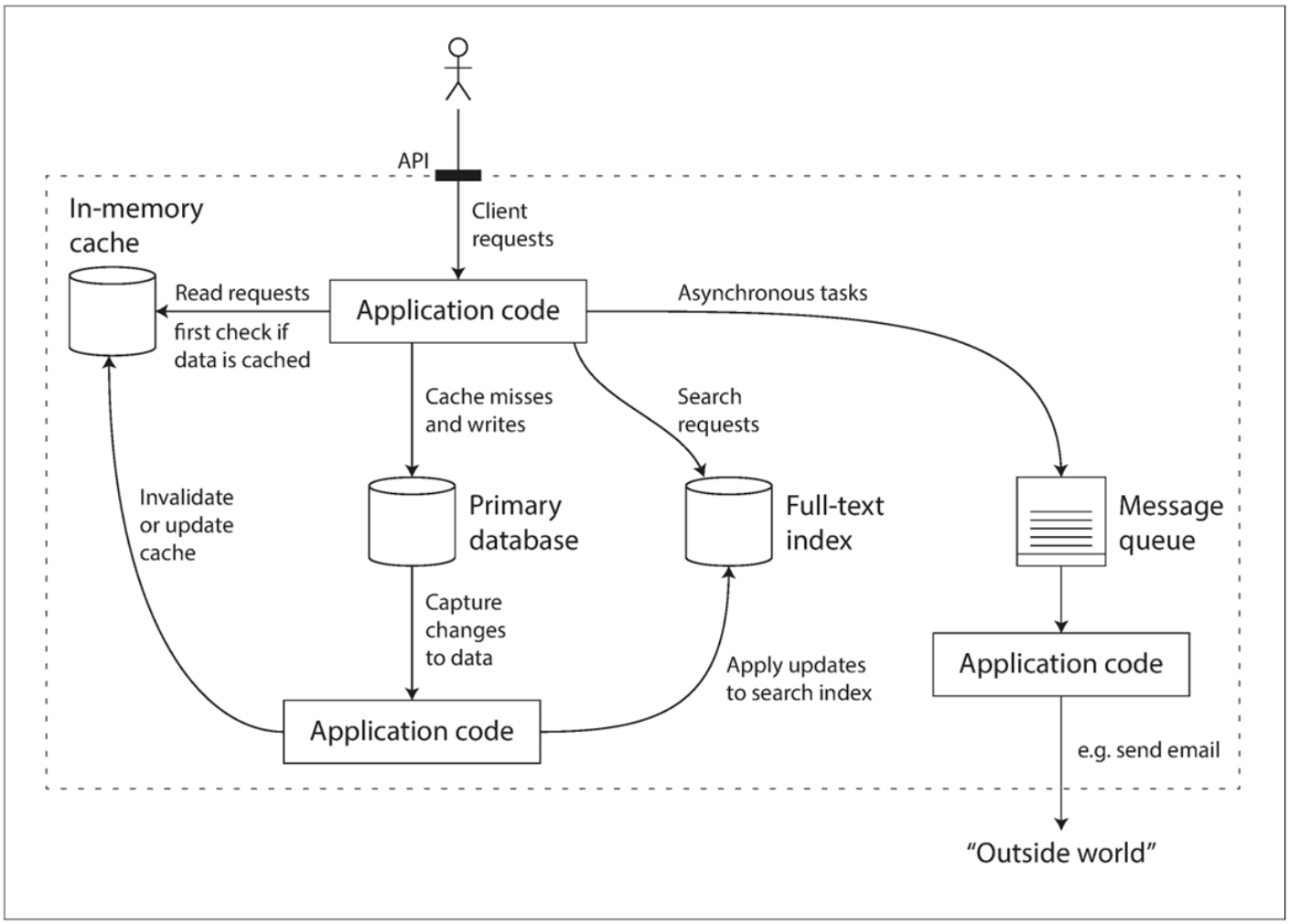
认识数据系统

通常将数据库、队列、高速缓存等视为不同类型的系统。 虽然数据库和消息队列存在某些相似性， 例如两者都会保存数据（至少一段时间），但他们却有着截然不同的访问模式，这就意味着不同的性能特征和设计实现。

近年来出现了许多用于数据存储和处理的新工具。 它们针对各种不同的应用场景进行优化，不适合再归为传统类型。例如，Redis既可以用于数据存储也适用于消息队列，Apache Kafka作为消息队列也具备了持久化存储保证。系统之间的界限正在变得模糊。

其次，越来越多的应用系统需求广泛，单个组件往往无法满足所有数据处理与存储需求。因而需要将任务分解，每个组件负责高效完成其中一部分，多个组件依靠应用层代码驱动有机衔接起来。举个例子，假定某个应用包含缓存层（例如Memcached)与全文索引服务器（如 Elasticsearch或Solr) , 二者

与主数据库保持关联，通常由应用代码负责缓存、索引与主数据库之间的同步，如下图（一种数据系统架构，它包含了多个不同组件）



上面例子中，组合使用了多个组件来提供服务， 而对外提供服务的界面或者API会隐藏很多内部实现细节。 这样基本上我们基千一个个较小的、通用的组件，构建而成 一个全新的、专用的数据系统。这样的集成数据系统会提供某些技术保证，例如，缓存要正确刷新以保证外部客户端看到一致的结果。现在可以说，你既是一名应用开发者，也是一名数据系统设计师。

设计数据系统或数据服务时，一定会碰到很多棘手的问题。例如，当系统内出现了局部失效时，如何确保数据的正确性与完整性？当发生系统降级(degrade)时，该如何为客户提供一致的良好表现？负载增加时，系统如何扩展？友好的服务API该如何设计？

影响数据系统设计的因素有很多，其中包括相关人员技能和经验水平、遗留系统依赖性、交付周期、对不同风险因素的容忍度、监管合规等。这些因素往往因时因地而异。本书将专注于对大多数软件系统都极为重要的三个问题：

1. 可靠性（Reliability）：当出现意外情况如硬件故障、软件故障、人为错误等，系统应可以继续正常运转，虽然性能可能会降低，但确保功能正确。
2. 可扩展性（Scalability）：随着规模的增长，例如数据量、流量、复杂性，系统应以合理的方式来匹配这种增长、

3. 可维护性（Maintainability）：随着时间推移，许多新的人员（在不同的生命周期）参与到系统开发和运维，以维护现有功能或适配新场景等，系统都应高效运转。

可靠性

对于软件，典型的期望包括：

- 应用程序执行用户所期望的功能
- 可以容忍用户出现错误或者不正确的软件使用方法
- 性能可以应对典型场景、合理负载压力和数据量
- 系统可防止任何未经授权的访问和滥用

如果上述目标都要支持才算正常工作，那么可以认为可靠性大致意味着：即使发生了某些错误，系统仍可以继续正常工作。

可能出错的事情称为错误（faults）或故障，系统可应对错误则称为容错（fault - tolerant）或者弹性（resilient）。讨论容错时，只有讨论特定类型的错误。

故障（fault）不同于失效（failure）：故障指的是一部分状态偏离标准，而失效则是系统作为一个整体停止向用户提供服务。不太可能将故障概率降低到零，因此通常设计容错机制来避免从故障引发系统失效。将介绍在不可靠组件基础上构建可靠性系统的相关技术。

在容错系统中，用于测试目的，可以故意提高故障发生概率，例如通过随机杀死某个进程，来确保系统仍保持健壮。很多关键的bug实际上正是由于错误处理不当而造成的。通过这种故意引发故障的方式，来持续检验、测试系统的容错机制，增加对真实发生故障时应对的信心。Netflix的ChaosMonkey系统就是这种测试的典例。

通常倾向于容忍错误（而不是阻止错误），但也有预防胜于治疗的情况（比如安全问题）

硬件故障

考虑系统故障时，对于硬件故障容易想到：硬盘崩溃、内存故障、电网停电、误拔网线等等。

通常可以为硬件添加冗余来减少系统故障率。例如对磁盘配置 RAID, 服务器配备双电源，甚至热插拔CPU,数据中心添加备用电源、发电机等。当一个组件发生故障，冗余组件可以快速接管，之后再更换失效的组件。 这种方法可能并不能完全防止硬件故障所引发的失效，但还是被普遍采用，且在实际中也确实可以让系统不间断运行长达数年。采用硬件冗余方案对于大多数应用场景还是足够的，它使得单台机器完全失效的概率降为非常低的水平。只要可以将备份迅速恢复到新机器上，故障的停机时间在大多数应用中并不是灾难性的。 而多机冗余则只对少最的关键应用更有意义，对于这些应用，高可用性是绝对必要的。

随着数据量和应用计算需求的增加，更多的应用可以运行在大规模机器之上，随之而来的硬件故障率呈线性增长。例如，对于某些云平台（如Amazon Web Services, AWS) , 由于系统强调的是总体灵活性与弹性而非单台机器的可靠性，虚拟机实例经常会在事先无告警的情况下出现无法访问问题。

因此， 通过软件容错的方式来容忍多机失效成为新的手段，或者至少成为硬件容错的有力补充。这样的系统更具有操作便利性，例如当需要重启计算机时为操作系统打安全补丁， 可以每次给一个节点打补丁然后重启，而不需要同时下线整个系统（即滚动升级）

软件错误

硬件错误之间多是相互独立的：一台机器磁盘故障并不意味着另一台机器的磁盘也要失效。除非存在某种弱相关（例如一些共性原因，如服务器机架温度过高），否则通常不太可能出现大量硬件组件同时失效的情况。

另一类故障是系统内的软件问题，这些故障事先更加难以预料，而且因为节点之间是由软件关联的，因而往往会导致更多的系统故障。

- 由于软件错误，导致当输入特定值时应用服务器总是崩溃。 例如，2012年6月30日发生闰秒，由于Linux内核中的一个bug, 导致了很多应用程序在该时刻发生挂起
- 一个应用进程使用了某些共享资源如 CPU、内存、磁盘或网络带宽，但却不幸失控跑飞了
- 系统依赖于某些服务，但该服务突然变慢，甚至无响应或者开始返回异常的响应
- 级联故障，其中某个组件的小故障触发另一个组件故障，进而引发更多的系统问题

导致软件故障的bug 通常潜伏很长时间，直到碰到特定的触发条件。这也意味着系统软件其实对使用环境存在某种假设，而这种假设多数情况都可以满足，但是在特定情况下，假设条件变得不再成立。

软件系统问题有时没有快速解决办法，而只能仔细考虑许多细节，包括认真检查依赖的假设条件与系统之间交互，进行全面的测试，进程隔离，允许进程崩溃并自动重启，反复评估，监控并分析生产环节的行为表现等。如果系统提供某些保证，例如，在消息队列中，输出消息的数量应等于输入消息的数量，则可以不断地检查确认，如发现差异则立即告警。

人为错误

一项针对大型互联网服务的调查发现，运维配置错误居然是系统下线的首要原因，而硬件问题（服务器或网络）仅在 10%~25% 的故障中有所影响。

如何保证系统的可靠性呢？可以尝试结合一下多种方法：

- 以最小出错的方式来设计系统。例如，精心设计的抽象层、API 以及管理界面。但是如果限制过多，人们就会想办法来绕过它，这会抵消其正面作用。因此解决之道在于很好的平衡。
- 分离最容易出错的地方、容易引发故障的接口。提供一个功能齐全但非生产用的沙箱环境，使人们可以放心的尝试、体验，包括导入真实的数据，出现问题不会影响到真实用户。

- 充分测试：从单元测试到全系统集成测试以及手动测试。自动化测试已被广泛使用，对于覆盖正常操作中很少出现的边界条件等尤为重要。
- 出现人为失误时，提供快速的恢复机制以尽量减少故障影响。例如，快速回滚配置改动，滚动发布新代码（这样意外错误仅会影响小部分用户），并提供校验数据的工具（防止旧的计算方式不正确）
- 设置详细清晰的监控室子系统，包括性能指标和错误率。其他行业称遥测（Telemetry），一旦火箭离开地面，遥测对于跟踪运行和了解故障至关重要。监控可以向我们发送告警信号，并检查是否存在假设不成立或违反约束条件等。这些检测指标对于诊断问题也非常有用。
- 推行管理流程并加以培训。这重要且复杂，超出本书范围。

可扩展性

可扩展性（Scalability）是用来描述系统应对负载增长能力的术语。

如果系统以某种方式增长，我们应对增长的措施有哪些?

我们该如何添加计算资源来处理额外的负载?

描述负载

需要简洁地描述系统当前的负载，这样才能更好地讨论后续增长问题（例如负载加倍会意味着什么）。负载可以用负载参数的若干数字来描述。参数的最佳选择取决于系统的体系结构，它可能是 Web 服务器的每秒请求处理次数，数据库写入的比例，聊天室同时活动用户数量，缓存命中率等。有时平均值很重要，有时系统瓶颈来自于少数峰值。

Twitter 的两个典型业务操作是：

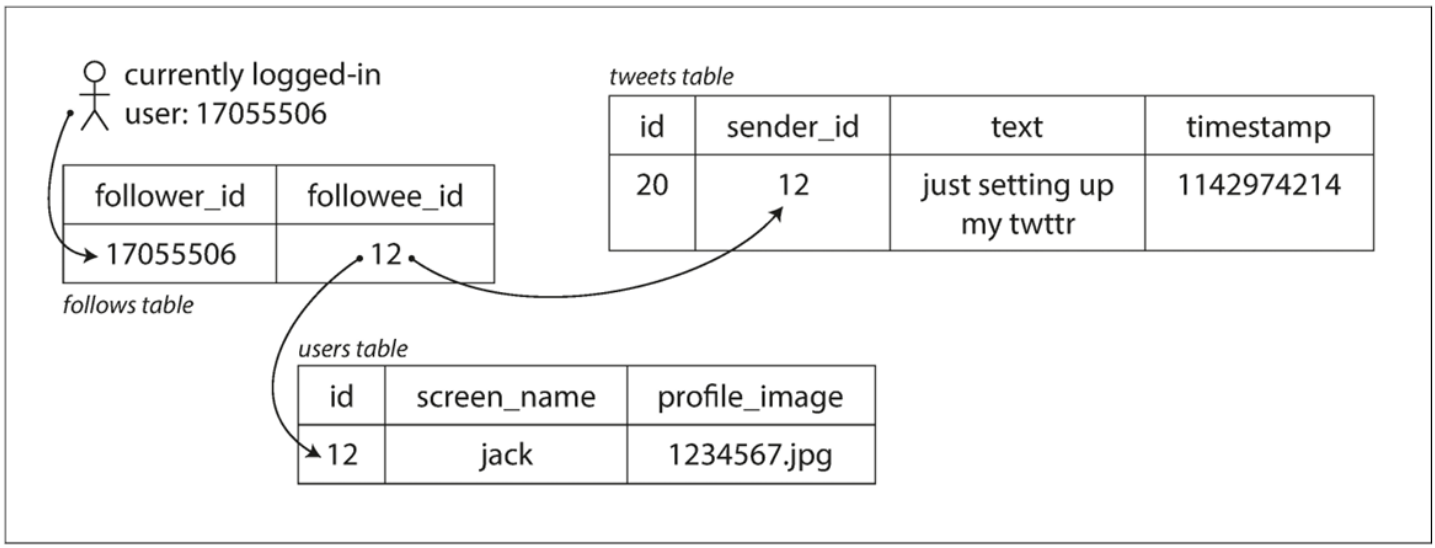
- 发布 tweet 消息：用户可以快速推送新消息到所有的关注者，平均大约 4.6k request/sec，峰值约 12k request/sec。
- 主页时间线（Home timeline）浏览：平均 300k request/sec 查看关注对象的最新消息。

仅处理峰值约 12k 的消息发送听起来不难，但是，Twitter 扩展性的挑战重点不在于消息大小，而在于巨大的扇出（fan - out）结构：每个用户会关注很多人，也会被很多人圈粉。大概有两种处理方案：

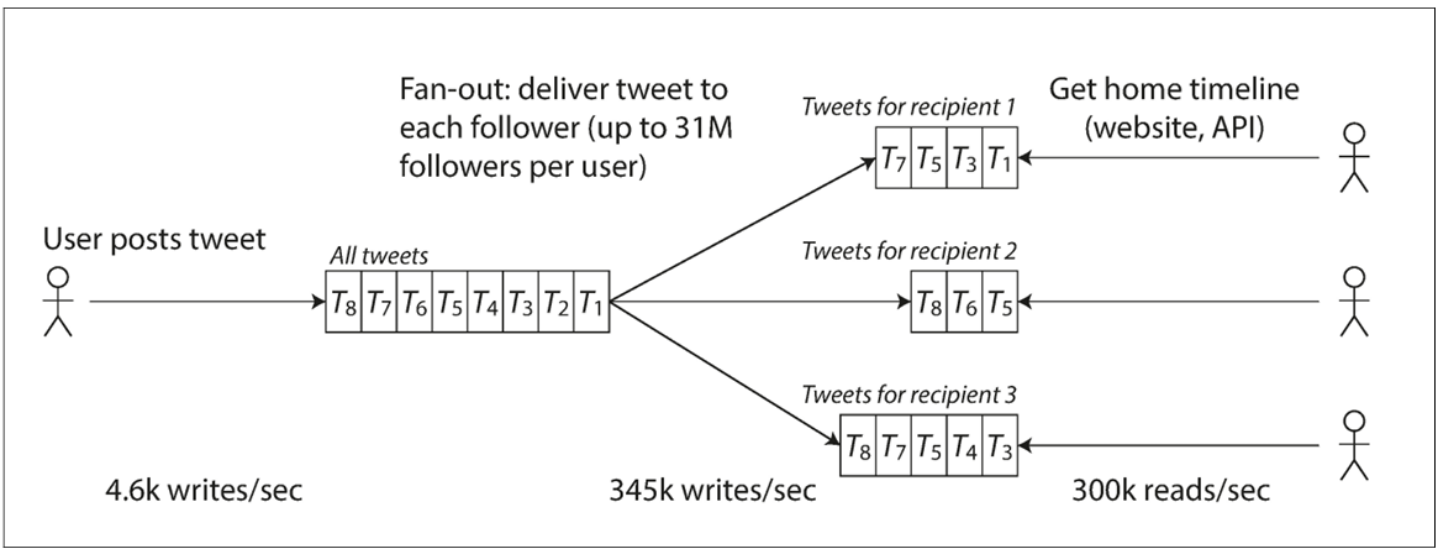
1. 将发送的新 tweet 插入到全局的 tweet 集合中。当用户查看时间线时，首先查找所有的关注对象，列出这些人的所有 tweet，最后以时间为序来排序合并。如下图（采用关系型数据模型来支持时间线），可以执行下述SQL查询语句：

```
1 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
```



2. 对每个用户的时间线维护一个缓存，如下图（以数据流水线方式来推送 tweet），类似每个用户一个 tweet 邮箱。当用户推送新 tweet 时，查询其关注者，将 tweet 插入到每个关注者的时间线缓存中。因为已经预先将结果取出，之后访问时间线性能非常快。



方法 1，发现主页时间线的读负载与日俱增，系统优化颇费周折，因此转而采用第二种方法。实践发现这样更好，因为时间线浏览 tweet 的压力几乎比发布 tweet 要高出两个数量级，基于此，在发布时多完成一些事情可以加速读性能。

然而方法 2 的缺点也很明显，在发布 tweet 时增加了大量额外的工作。考虑平均 75 个关注者和每秒 4.6k 的 tweet，则需要每秒 4.6 x 75 = 345k 速率写入缓存。但是，关注者其实偏差巨大，例如某些用户拥有超过 3000w 的 followers。这意味着峰值情况下一个 tweet 会导致 3000w 笔写入！而且要求尽量快，Twitter 的设计目标是 5s 内完成。

Twitter 的例子中，每个用户关注者的分布情况（还可以结合用户使用 Twitter 频率情况进行加权）是该案例可扩展的关键负载参数，因为它决定了扇出数。你的应用可能具有不同的特性，但可以采用类

似的原则来研究具体负载。

Twitter 故事最后的结局是：方法 2 已经得到了稳定实现，Twitter 正在转向结合两种方法。大多数用户的 tweet 在发布时继续以一对多写入时间线，但是少数具有超多关注者（例如名人）的用户除外，对这些用户采用类似方案 1，其推文被单独提取，在读取时才和用户的时间线主表合并。混合方法能够提高更好的表现。

描述性能

1. 当描述好负载以后，问题变成了：
   a. 增加负载参数并保持系统资源不变时，系统性能将受到什么影响?
   b. 增加负载参数并希望性能不变时，需要增加多少系统资源?
2. 批处理系统，通常关心吞吐量（throughput）；在线系统，通常更关心响应时间（response time）
3. 对于系统响应时间而言，最好用百分位点，比如中位数、p99 等标识。
4. 测量客户端的响应时间非常重要（而不是服务端），比如会出现头部阻塞、网络延迟等。
5. 实践中的百分位点，可以用一个滑动的时间窗口（比如 10 分钟）进行统计。可以对列表进行排序，效率低的话，考虑一下前向衰减，t-digest 等方法近似计算。

描述系统负载之后，接下来设想如果负载增加将会发生什么。有两种考虑方式：

- 负载增加，但系统资源（如CPU、内存、网络带宽等）保持不变，系统性能会发生什么变化?
- 负载增加，如果要保持性能不变，需要增加多少资源?

两个问题都会关注性能指标，所以简要介绍一下如何描述系统性功能。

在批处理系统如 Hadoop 中，我们通常关心吞吐量（throughput），即每秒可处理的记录条数，或者在某指定数据集上运行作业所需的总时间；而在线系统通常更看重服务的响应时间（response time），即客户端从发送请求到接收响应之间的间隔。

延迟与响应时间

延迟（latency）和响应时间（response time）容易混淆，但它们并不完全一样。通常响应时间是客户端看到的：除了处理请求时间（服务时间，service time）外，还包括来回网络延迟和各种排队延迟。延迟则是请求花费在处理上的时间。

由于系统要处理各种不同的请求，响应时间可能变化很大。因此，最好不要将响应时间视为一个固定的数字，而是可度量的一种数值分布。

如图1-4所示，每个灰色条表示一个服务请求，高度表示该请求的响应时间。可以看到，大多数请求是相当快的，但偶尔会有异常表示需要更长的时间。也许这些异常请求确实代价很高，例如它们的数据大很多。但有时，即使所有请求都相同，也会由于其他变量因素而引入一些随机延迟抖动，这些因素

包括上下文切换和进程调度、网络数据包丢失和TCP重传、垃圾回收暂停、缺页中断和磁盘I/O，甚至服务器机架的机械振动。

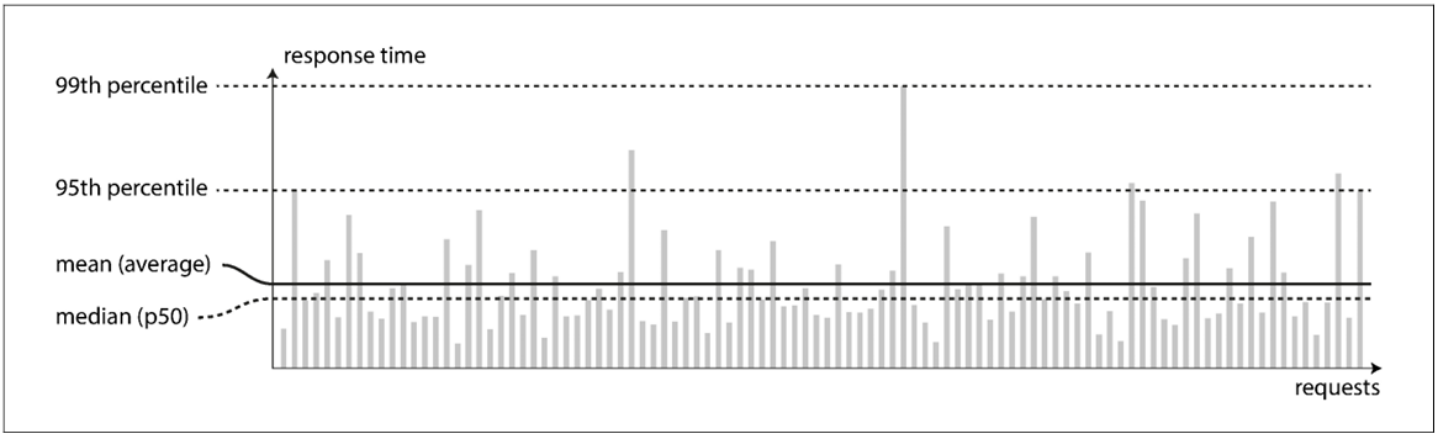


图1-4：对某服务采样100个服务请求，来说明响应按时间平均值和相关百分位数

经常考查的是服务请求的平均响应时间。然而，如果想知道更有代表性的响应时间，平均值并不是合适的指标，因为它掩盖了一些信息，无法告诉你有多少用户实际经历了多少延迟。

因此最好使用百分数（percentiles）。如果已经收集到了响应时间信息，将其从最快到最慢排序，中位数（median）就是列表中间的响应时间。例如，如果中位数响应时间为 200ms，意味着有一半的请求响应不到 200ms，而另一半请求则需要更长的时间。

中位数指标适合描述多少用户需要等待多长时间：一半用户请求的服务时间少于中位数响应时间，另一半则多于中位数的时间。因此中位数也称50百分位数，缩写p50。注意，中位数对应单个请求；意味着如果某用户发了多个请求（例如包含在一个完整会话过程中，或者某页面包含了多个资源），那么它们中至少一个比中位数慢的概率远大于50%。

为弄清异常值多糟糕，需要关注更大的百分位数如常见的第95、99和99.9（p95、p99和p999）值。作为典型的响应时间阈值，它们分别表示有95%、99%或99.9%的请求响应时间快于阈值。例如，95百分位数响应时间为1.5s，意味着100个请求中的95个请求快于1.5s，而5个请求则需要1.5s或更长时间，如图1-4所示。

采用较高的响应时间百分位数（tail latencies，尾部延迟或长尾效应）很重要，因为它们直接影响用户的总体服务体验。例如，亚马逊采用99.9百分位数来定义其内部服务的响应时间标准，或许它仅影响1000个请求中的1个。但是考虑到请求最慢的用户往往是购买了更多的商品，因此数据量更大。换言之，他们是最有价值的客户，让这些客户保持愉悦的购物体验非常重要：亚马逊还注意到，响应时间每增加100ms，销售额就会下降了约1%，其他研究则表明，1s的延迟增加等价于客户满意度下降16%。

但也有人说，优化这个99.99百分位数（10000个请求中最慢的1个）代价昂贵，并没有为亚马逊的商业目标带来足够的收益。进一步提高响应时间技术上代价更大，容易受到非可控因素如随机事件的影响，累积优势会减弱。

例如，百分位数通常用于描述、定义服务质量目标（Service Level Objectives，SLO）和服务质量协议（Service Level Agreements，SLA），这些是规定服务预期质量和可用性的合同。例如一份SLA合

约通常会声明，响应时间中位数小于200ms，99%请求的响应时间小于1s，且要求至少99.9%的时间都要达到上述服务指标。这些指标明确了服务质量预期，并允许客户在不符合SLA的情况下进行赔偿。

排队延迟往往在高百分位数响应时间中影响很大。由于服务器并行处理的请求有限（例如，CPU内核数的限制），正在处理的少数请求可能会阻挡后续请求，这种情况有时被称为对头阻塞。即使后续请求可能处理很简单，但它阻塞在等待先前请求的完成，客户端将会观察到极慢的响应时间。因此，很重要的一点是要在客户端来测量响应时间。

为测试系统的可扩展性而人为产生负载时，产生负载的客户端要独立于响应时间（不受响应时间的影响）持续发送请求。如果客户端在发送请求之前总是等待先前请求的完成，就会在测试中人为地缩短了服务器端的累计队列深度，这就带来了测试偏差。

应对负载的方法

1. 纵向扩展：转向更强大的机器
2. 横向扩展：将负载分布到多台小机器上
3. 弹性系统：检测到负载增加时自动增加计算资源
4. 跨多台机器部署无状态服务比较简单，但是把带状态的数据系统从单节点变成分布式配置则可能引入许多额外复杂度。因此，应该尽量将数据库放在单个节点上。

了解了描述负载的参数以及衡量性能的相关指标，接下来讨论可扩展性：即当负载参数增加时，应如何保持良好性能？

实践中的百分位数。对于后台服务，如果一次完整的服务里包含了多次请求调用，此时高百分位数指标尤为重要。即使这些子请求是并行发送、处理，但最终用户仍然需要等待最慢的那个调用完成才行。（比如下载视频，客户端多线程下载分片，如果没有边下边看功能，则需要等到视频完全下载完才能播放）

最好将响应时间百分位数添加到服务系统监控中，持续跟踪该目标。例如，设置一个10min的滑动窗口，监控其中的响应时间，滚动计算窗口中的中位数和各种百分位数，然后绘制性能图表。

一种简单的实现方案是在时间窗口内保留所有请求的响应时间列表，每分钟做一次排序。如果这种方式效率太低，可以采用一些近似算法（如正向衰减，t-digest 或 HdrHistogram）来计算百分位数，其CPU和内存开销很低。注意，降低采样时间精度或直接组合来自多台机器的数据，在数学上没有太大意义，聚合响应时间的正确方法是采用直方图。

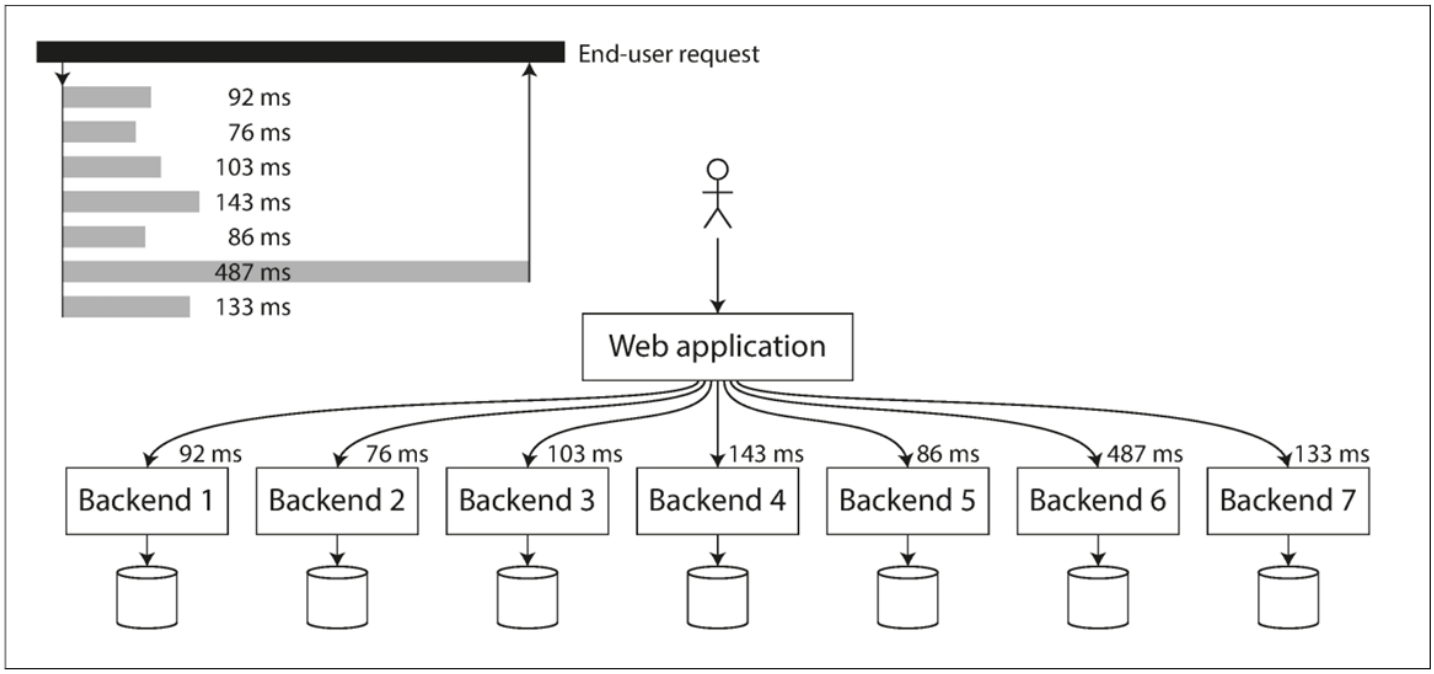


图1-5：一个服务涉及多个不同的后端调用，最慢的调用会拖累整个服务响应时间

针对特定级别负载设计的架构不太可能应付超出预设目标10倍的实际负载。如果目标服务处于快速增长阶段，需要认真考虑每增加一个数量级的负载，架构应如何设计。

现在谈论更多的是如何在垂直扩展（即升级到更强大的机器）和水平扩展（即将负载分布到多个更小的机器）之间作取舍。多台机器上分配负载也称为无共享体系结构。在单台机器上运行的系统通常更简单，但高端机器非常昂贵，且扩展水平有限，最终还是无法避免需要水平扩展。实际上，好的架构通常要做些实际取舍，例如，使用几个强悍的服务器可以比大量的小型虚拟机来的更简单、便宜。

某些系统具有弹性特征，它可以自动检测负载增加，然后自动添加更多计算资源，而其他系统则是手动扩展（人工分析性能表现，之后决定添加更多计算。例如针对相机启动时，提高CPU频率）。如果负载高度不可预测，则自动弹性系统会更加高效，但或许手动方式可以减少执行期间的意外情况（第6章 - 分区再平衡）。

把无状态服务分布然后扩展至多台机器相对比较容易，而又状态服务从单个节点扩展到分布式多级环境的复杂性会大大增加。因此，最近通常的做法是，将数据库运行在一个节点上（采用垂直扩展策略），知道高扩展性或高可用性的要求迫使不得不做水平扩展。

随着相关分布式向传统专门组件和编程接口的发展，对于某些应用类型来讲，上述做法或许会改变。即使应用可能并不会处理大量数据或流量，但未来分布式数据系统将成为标配。后续部分，将介绍多种分布式数据系统，不仅可以帮助提高可扩展性，也会提高易用性与可维护性。

超大规模的系统往往针对特定应用而高度定制，很难有一种通用的架构。背后取舍因素包括数据读取量、写入量、待存储的数据量、数据的复杂程度、响应时间要求、访问模式等，或者更多的是上述所有因素的叠加，再加上其他更复杂的问题。

例如，及时两个系统的数据吞吐量折算下来是一样的，但是为每秒处理100000次请求（每个大小为1KB）而设计的系统，与为每分钟3个请求（每个大小为2GB）设计的系统回答不相同。

对于特定应用来说，扩展能力好的架构通常会做出某些假设，然后有针对性地优化设计，如哪些操作是最频繁的，哪些负载是少数情况。如果这些假设最后发现是错误的，那么可扩展性的努力就白费

了，甚至会出现与设计预期完全相反的情况。对于初创公司或者尚未定性的产品，快速迭代推出产品功能往往比投入精力来应对不可知的扩展性更为重要。

可扩展架构通常都是从通用模块逐步构建而来的，背后往往有规律可循，所以本书会讨论这些通用模块和常见模式，希望对读者有所借鉴。

可维护性

1. 在设计之初就尽量考虑尽可能减少维护期间的痛苦，从而避免自己的软件系统变成遗留系统。
2. 三个设计原则：可操作性（Operability）便于运维团队保持系统平稳运行。简单性（Simplicity）从系统中消除尽可能多的复杂度（complexity），使新工程师也能轻松理解系统。（注意这和用户接口的简单性不一样。）可演化性（evolability）使工程师在未来能轻松地对系统进行更改，当需求变化时为新应用场景做适配。也称为可扩展性（extensibility），可修改性（modifiability）或可塑性（plasticity）。

软件的大部分成本不在最初的开发阶段，而是在于整个生命周期内持续的投入，包括维护与缺陷修复，监控系统来保持正常运行、故障排查、适配新平台、搭配新场景、技术缺陷的完善以及增加新功能等。

许多从业人不喜欢维护这些遗留系统，例如修复他人埋下的错误，或者使用过时的开发平台，或者被迫做不喜欢的工作。每个遗留系统总有其过期的理由，很难给出一个通用的建议该如何处理它们。

但是，我们可以从软件设计时开始考虑，尽可能减少维护期间的麻烦，避免造出容易淘汰的系统。为此，要特别关注软件系统的三个设计原则：

- 可运维性：方便运维团队保持系统平稳运行
- 简单性：简化系统复杂性，使新工程师能轻松理解系统。注意这与用户界面的简单性不一样
- 可演化性：后续工程师能够轻松对系统进行改进，并根据需求变化将其适配到非典型场景，也称为可延伸性、已修改性或可塑性

与可靠性和可扩展性类似，实现上述这些目标也没有简单的解决方案。下面建立对着三个特性的理解。

可运维性：运维更轻松

虽然某些操作可以而且应该是自动化的，但最终还是需要人来执行配置并确保正常工作。

运维团队对于保持软件系统顺利运行至关重要。一个优秀的运维团队至少负责以下内容：

- 监视系统的健康状况，并在服务出现异常状态时快速恢复服务
- 追踪问题的原因，例如系统故障或性能下降

- 保持软件和平台至最新状态，例如安全补丁方面
- 预测未来可能得问题，并在问题发生之前及时解决（例如容量规划）
- 建立用于部署、配置管理等良好的时间规范和工具包
- 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
- 当配置更改时，维护系统的安全稳健
- 制定流程来规范操作行为，并保持生产环境稳定
- 保持相关知识的传承（如对系统理解），例如发生团队人员离职或者新员工加入等

良好的可操作性意味着使日常工作变得简单，使运维团队能够专注于高附加值的任务。数据系统设计可以在这方面贡献很多，包括：

- 提供对系统运行时行为和内部的可观测性，方便监控
- 支持自动化，与标准工具集成
- 避免绑定特定的机器，这样在整个系统不间断运行的同时，允许机器停机维护
- 提供良好的文档和易于理解的操作模式，诸如 “如果我做了 X，会发生 Y”
- 提供良好的默认配置，且允许管理员在需要时方便地修改默认值
- 尝试自我修复，在需要时让管理员手动控制系统状态
- 行为可预测，减少意外发生

简单性：简化复杂度

小型软件项目通常可以写出简单而漂亮的代码，但随着项目越来越大，就会越来越复杂和难以理解。这种复杂性拖慢了开发效率，增加了维护成本。

复杂性有各种各样的表现方式：状态空间的膨胀，模块紧耦合，令人纠结的相互依赖关系，不一致的命名和术语，为了性能而采取的特殊处理，为解决特定问题引入的特殊框架等。

复杂性使得维护变得越来越困难，最终会导致预算超支和开发进度滞后。对于复杂的软件系统，变更而引入潜在错误的风险会显著加大，最终开发人员更加难以准确理解、评估或者更加容易忽略相关系统行为，包括背后的假设，潜在的后果，设计之外的模块交互等。相反，降低复杂性可以大大提高软件的可维护性，因此简单性应该是我们构建系统的关键目标之一。

简化系统设计并不意味着减少系统功能，而主要意味着消除意外方面的复杂性。复杂性定义为一种“意外”，即它并非软件固有、被用户所见或感知，而是实现本身所衍生出来的问题。

消除意外复杂性最好手段之一是抽象。一个好的设计抽象可以隐藏大量的实现细节，并对外提供干净、易懂的接口。一个好的设计抽象可用于各种不同的应用程序。这样，复用远比多次重复实现更有效率；另一方面，也带来更高质量的软件，而质量过硬的抽象组件所带来的好处，可以使运行其上的所有应用轻松获益。

例如，高级编程语言作为一种抽象，可以隐藏机器汇编代码、CPU寄存器和系统调用等细节和复杂性。SQL作为一种抽象，隐藏了内部复杂的磁盘和内存数据结构，以及来自多客户端的并发请求，系统崩溃滞后的不一致等问题。当然，使用高级编程语言最终并没有脱离机器汇编代码，只是并非直接使用，与汇编代码打交道的事情已经由编程语言抽象为高效接口代我们完成。

然而，设计好的抽象还是很有挑战性。在分布式系统领域中，虽然已有许多好的算法可供参考，但很多时候我们并不太清楚究竟该如何 利用他们，封装到抽象接口之中，最终帮助将系统的复杂性降低到可掌控的级别。

本书我们将广泛考察如何设计好的抽象，这样至少能够将大型系统的一部分抽象为定义明确、可重用的组件。

可演化性：易于改变

- 敏捷（agile） 工作模式为适应变化提供了一个框架
- 简单易懂的系统通常比复杂系统更容易修改，即可演化性（evolvability）

一成不变的系统需求几乎没有，想法和目标经常在不断变化：适配新的外部环境，新的用例，业务优先级的变化，用户要求的新功能，新平台取代旧平台，法律或监管要求的变化，业务增长促使架构的演变等。

组织流程方面，敏捷开发模式为适应变化提供了很好的参考。敏捷社区还发布了很多技术工具和模式，以帮助在频繁变化的环境中开发软件，例如测试驱动开发（TDD）和重构。

这些敏捷开发技术目前多数还只是针对小规模、本地模式（例如同一应用程序中的几个源代码文件）环境。本书将探索在更大的数据系统层面上提高敏捷性，系统由多个不同特性的应用或者服务协作而成。例如，对于 Twitter 的案例（描述负载），如何从方法1过渡到方法2，重构Twitter架构来实现主页时间线。

我们的目标是可以轻松地修改数据系统，使其适应不断变化的需求，这和简单性与抽象性密切相关：简单易懂的系统往往比复杂的系统更容易修改。这是一个非常重要的理念，我们将采用另一个不同的词来指代数据系统级的敏捷性，即可演化性。

小结

这一章探讨了一些关于数据密集型应用的基本原则，这些原则将指导如何阅读本书的其余部分。

一个应用必须完成预期的多种需求，主要包括功能性需求（应该做什么，比如各种存储、检索、搜索和处理数据）和一些非功能性需求（常规特性，例如安全性、可靠性、合规性、可伸缩性、兼容性或可维护性）。

可靠性意味着即使发生故障，系统也可以正常工作。故障包括硬件（通常是随机的，不相关的）、软件（缺陷通常是系统的，更加难以处理）以及认为（总是很难避免时不时会出错）方面。容错技术可

以很好地隐藏某种类型故障，避免影响最终用户。

可扩展性是指负载增加时，有效保持系统性能的相关技术策略。为了讨论可扩展性，首先探讨了如何定量描述负载和性能。简单以Twitter浏览时间线为例描述负载，并将响应时间百分位数作为衡量性能的有效方式。对于可扩展的系统，增加处理能力的同时，还可以在高负载情况下持续保持系统的高可靠性。

可维护性意味着许多方面，但本质是为了让工程和运维团队更为轻松。良好的抽象可以帮助降低复杂性，并使系统更易于修改和适配新场景。良好的可操作性意味着对系统健康状况有良好的可观测性和有效的管理办法。

然而知易行难，是应用程序可靠、可扩展和可维护并不容易。考虑到一些重要的模式和技术在很多不同应用中普遍适用，接下来几章，我们就一些数据密集系统例子，分析它们如何实现上述这些目标。

本书第三部分，将看到更多如图1-1所示的包含多个组件但更加复杂的例子。