第一部分 数据系统基础

前四章主要讲述数据系统的基础知识，适用于单机或分布式集群。

1. 第一章，主要介绍本书使用的术语和方法，阐述了可靠性、扩展性和易维护性等系统指标的含义，以及实现以上目标的方式方法。
2. 第二章，通过比较几种不同的数据模型和查询语言（不同数据库之间最为明显的区别）。我们可以得出不同场景下最合适的数据模型
3. 第三章，研究存储引擎内部原理，了解数据如何落地磁盘。不同的存储引擎适用于不同的服务负载，选择合适的存储引擎可以实现最佳性能。
4. 第四章，对比数据序列化的方式，特别强调为适应系统需求改变或计划调整，导致序列化方式高昂的代价。

# 第1章 系统的可靠性、扩展性和维护性

当今互联网时代，很多系统是数据密集型的，与之相对是计算密集型的。CPU很难成为这些系统的瓶颈，而数据的体量、复杂性以及变化频率却是非常棘手的问题。

通常，构建数据密集型应用，需要考虑以下几个方面：

1. 存储数据，便于查询（数据库）；
2. 热数据缓存化，加快查询速度（缓存）；
3. 提供搜索功能（检索）；
4. 异步化处理消息（流式处理）；
5. 周期性处理大量累计数据（批处理）；

如果系统具备以上功能，说明系统抽象非常完美。但是，工程师构建系统时，不会刻意设计一套新的存储引擎，因为主流的数据库已经可以完美的解决大部分基本需求。

实际上却不是这样的，不同的数据库有不同的特性，每个系统有各自的需求。拿构建缓存、索引来讲，有很多种有效方式。工程师的工作应该是评估哪些组件更适合当前的系统，但选型与集成这些组件来实现系统设计是比较困难的。

本书会结合理论和实践两方面，在构建数据密集型系统方面，给大家提供建议。我们会阐述组件之间的共性、特性以及其实现方式。

本章节，我们会了解数据系统可靠性、扩展性、维护性等方面的基本原理。我们将会阐明以上特性的含义，概括几种思考方式，以及复习基础知识。接下来的章节，我们会聚焦设计数据密集型系统的选型和决策。

## 数据系统的思考

通常，我们认为数据库、队列、缓存等组件是完全不同的组件。虽然数据库和队列有很多相似点——可以存储数据一段时间，但是二者有不同的访问模式，这就导致了不同的性能特性和实现方式。

但是，为什么我们应该把二者集成在数据系统内呢？

近几年，出现了很多数据存储与处理的组件，适用于很多使用场景，而不仅是针对特定场景。例如，有些数据存储可作为消息队里使用（Redis），同时有些消息队列具有数据库的持久化功能（Apache Kafka），中间件的边界正在变得模糊。

其次，越来越多的系统具备非常宽泛的功能，导致单一的中间件无法满足其数据存储与处理的需求。因此，系统被拆分为多个功能模块，多个中间件组合使用满足系统的需求。

例如，缓存系统（Memcached）或全文检索引擎（ElasticSearch或Solr）独立于数据库之外，我们需要通过代码的方式保证缓存、全文检索与数据库的一致性。如图1-1所示。

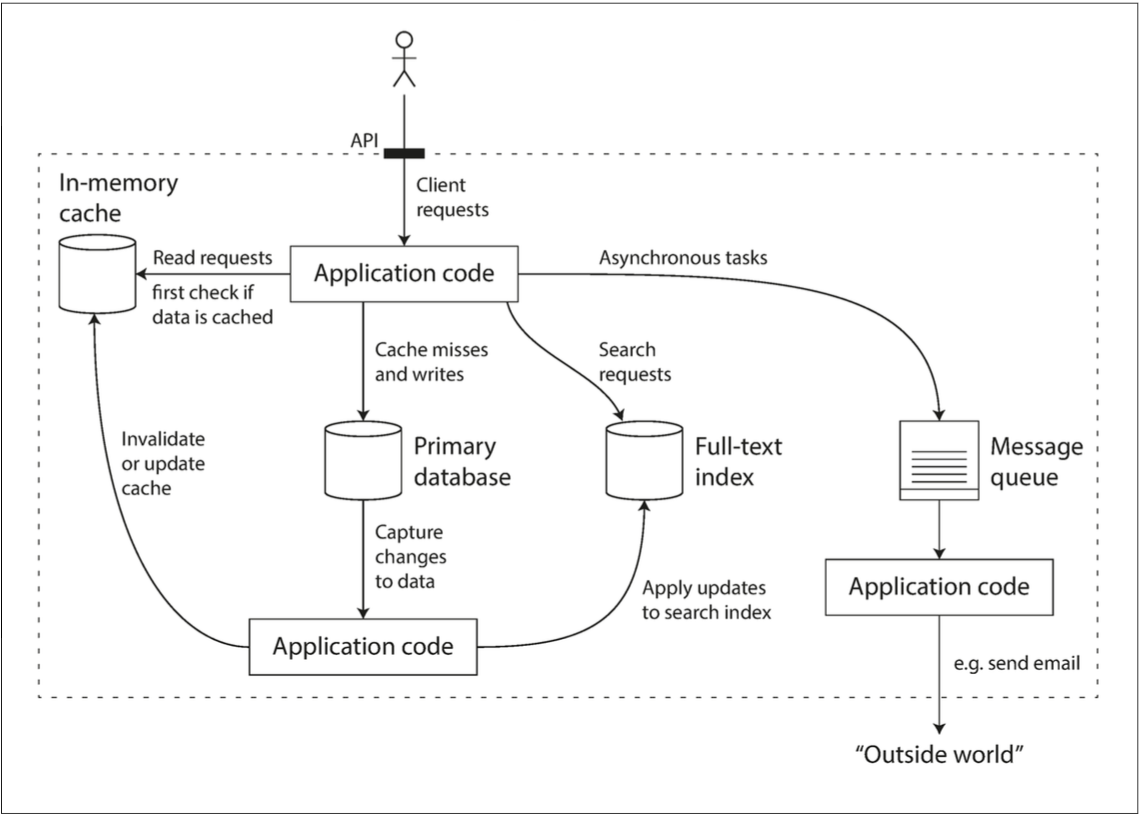


图1-1 数据系统的一种架构

当我们使用很多中间件实现服务时，API对客户端隐藏了实现细节。现在你已经利用某个中间件构建了一个特定功能数据系统，可能提供了某种特性：例如，客户端执行写操作，缓存能被正确的失效或更新，证明你不仅拥有系统开发能力，还具备数据系统设计能力。

设计数据系统或服务时，会遇到很多棘手的问题。如何保证数据的准确性和完整性，即使出现故障？当系统出现部分功能失效或降级时，如何保证可靠的服务？如何扩展系统来扛住更多的访问？优秀的API到底该如何设计？

设计数据系统有很多影响因子，包括参与者的技能和经验、系统依赖、持续交付、团队执行力等。

本书主要讨论设计软件的三个重要原则。

1. 可靠性

系统可持续性正常运行（以期望的方式提供正确的功能），即使遇到故障（软件、硬件故障或人为地错误操作）

1. 可扩展性

随着系统的成长（数据体量变大、访问量变大或复杂性提升），通过合理的方式快速扩容

1. 可维护性

许多不同的人会参与系统运行（工程师或业务人员，既要保障当前功能正常，也要保证系统可适应新的需求），并且可高效的维护系统运行。

这些概念常常不能被正确理解，基于对深度工程化的兴趣，在后续的章节中，我们会探索如何理解可靠性、可扩展性、可维护性。接着，我们会研究很多技术、架构和算法来实现以上目标。

## 1.2 可靠性

系统的可靠性与不可靠性定义，每个人有着自己的理解。对于软件开发，主要包括以下方面：

1. 可提供用户期望的功能；
2. 能容忍用户的误操作或系统异常；
3. 在容量范围内，系统高性能运行；
4. 拒绝非法访问与过载

如果综合以上因素称为“正确地运行”，那我们可粗略的理解可靠性为“系统能正确地运行，即使出现异常”。

系统异常可称之为故障，系统具备发现故障或处理故障称之为可容错性或可复原性。前人经验易误导读者：系统应该具备容忍所有可能的错误，事实上是不可实现的。假如地球被黑洞吞并，那系统需要建立在太空上。因此，容忍系统的某些错误才是有意义的。

请注意，错误不是故障。错误通常被定义为系统的某些组件脱离正常运行轨迹，而故障则是系统无法为用户提供服务。系统错误率为零是不可能的，因此系统通常被设计成具备容错能力，从而达到避免错误升级为故障的效果。在本书中，我们会利用几种技术，讨论系统不可靠的部分，来构建可靠的系统。

在容错系统中，可通过刻意、谨慎地增加错误率来验证，例如：随机杀死进程。通常，许多严重的漏洞来自错误的操作；通过谨慎的增加错误，我们可以验证机器的可容错性，当错误发生时，我们可自信的处理。NetFlix Chaos Monkey就是典型的例子。

虽然，相比于阻止错误发生，我们更倾向于容忍错误，但是有些场景阻止错误发生更重要，因为压根就没有容忍方案。我们来看一个安全方面的问题，如果攻击者入侵了系统或具有访问重要数据的权限，并且共计没法被消除。本书中，我们只讨论能被处理的错误。

### 1.2.1 硬件错误

### 1.2.2 软件错误

### 1.2.3 人为错误