**数据密集型系统开发的应对之法**

**——《设计数据密集型应用》读书笔记**

**文曼谕 2022141461053**

1. **概况**
2. **书籍信息**

原著：《Designing Data-Intensive Applications》

作者：Martin Kleppmann

领域：数据系统，应用开发

出版时间：2017年

1. **总体介绍**

《设计数据密集型应用》是一本深入探讨如何构建可扩展、可靠且可维护的数据密集型应用的书籍。作者从理论到实践，全面覆盖了数据密集型应用的设计原则与技术细节，不仅介绍了如何设计、实现和评估数据密集型应用程序，还重点强调了数据系统可靠性、可扩展性和可维护性的重要性。作者认为，在当今互联网时代，大多数应用都属于数据密集型，因此对数据处理和存储技术提出了高要求，这三个特性更是开发过程中的要点。

1. **章节结构**

本书分为三大部分：数据系统的基石、分布式数据、衍生数据。

图示

描述已自动生成

图1 本书总框架结构

第一部分主要介绍了系统设计的可靠性、可扩展性、可维护性的必要性以及实现的设计原则及技术。此外还介绍了多种数据模型与查询语言，讨论了不同存储引擎的工作原理以及索引技术，还介绍了数据的编码技术和保持数据向前向后的兼容性的方法。

第二部分则深入探讨了分布式数据的处理，包括复制、分区、事务和一致性等关键概念，以及分布式系统中的常见问题和解决方案。

第三部分讨论了从其他数据集中衍生数据的系统，包括批处理和流处理，以及相应的处理框架的工作原理。同时，还探讨了未来数据密集型应用的发展趋势和研究方向。

1. **第一部分：数据系统的基石**

****

图2 第一部分框架结构

1. **可靠性、可扩展性、可维护性**

**可靠性**是指系统能够在预期时间内完成其所需的任务，而不出现任何故障或错误，例如：硬件损坏、软件错误或人为失误等导致的系统失效。作者提出，可以复制关键组件或数据或定期备份数据，使得当某个组件发生故障时系统能够快速恢复；同时，还提到可以完善容错机制，实现自动化的故障检测和恢复机制，如心跳检测、超时机制、自愈算法等。

**可扩展性**是指系统能够适应不断增长的需求和负载，而不会影响其性能和可用性。其包括垂直扩展（增加单个服务器的资源，如CPU、内存）和水平扩展（添加更多的服务器节点）。在这些过程可以考虑将请求、数据（“Sharding”）分配给不同的服务器节点，并在内存中存储频繁访问的数据副本，以减少对后端数据库的压力。

**可维护性**是指系统能够方便地进行修改、更新和维护，而不会对现有的功能产生负面影响。为方便后续的维护，降低系统的复杂度、实现接口的抽象、进行模块化设计是非常有必要的。同时，开发人员还需要编写清晰易懂的文档，严格遵循编码规范，确保日志记录、跟踪分析等易于调试的技术支持。

这三个特性对于软件系统的成功至关重要：可靠性确保了系统的稳定性和可用性；可扩展性确保了系统能够满足不断变化的需求，从而保持竞争力；可维护性确保了系统能够长期维护和支持，从而降低了运营成本并提高了效率。因此，在设计和开发软件系统时，必须优先考虑这些特性，并采取相应的措施来确保它们得到充分的保障。

1. **数据模型与查询语言**
2. **数据模型的选择**

在选择数据模型时，要根据应用场景的特点，评估不同模型的优势和局限性；同时，随着业务需求和技术发展，原有数据模型可能需要调整或替换，因此也要考虑模型的灵活性和适应性。常见的数据模型有：

* **关系模型**，采用基于表格结构的数据组织方式，使用行和列来表示实体及其属性之间的关系。
* **文档模型**，以文档形式存储非结构化或半结构化数据，适合灵活多变的数据格式。
* **图模型**，用节点和边表示实体及它们之间的关联关系，适用于社交网络、推荐系统等领域。

此外，作者还提到了数据模型与编程语言的兼容性问题。如阻抗不匹配，即面向对象编程语言中的对象模型和关系数据库中的表格模型之间存在不兼容性，这通常借助ORM框架来桥接两者间的差距。

1. **查询语言的优化**

广泛运用的查询语言有如MySQL的关系型数据库查询语言SQL和如MongoDB的非关系型数据库查询语言NoSQL。不管选择何种查询语言，都应该学习如何编写高效的查询语句，避免不必要的搜索操作，充分利用索引和其他加速结构。

1. **事务管理**

事务指一组数据库操作，要么全部成功执行，要么全部失败回滚。它具有ACID特性，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。为保障这四个特性，针对不同的场景和需求，有不同的隔离级别可以选择（读未提交、读已提交、可重复读、序列化）。其中，由于随着隔离级别的提高，事务的并发度也会降低，因此需要权衡利弊。此外，该章节还强调了并发控制的重要性，因为并发操作可能导致多种并发性错误，如脏读、脏写等，而事务隔离则是很好的解决办法。

而对于不同类型的数据库，可以采取不同的方式来实现事务控制。关系型数据库通常采用基于锁定的事务机制，即在事务执行过程中，会对相关的数据加锁，防止其他事务对其进行修改。而非关系型数据库则通常采用基于版本的事务机制，即每个数据都有一个版本号，每次修改都会增加版本号，只有在版本号匹配的情况下才会允许修改。此外，还有一些数据库采用了基于时间戳的事务机制，即每个事务都有一个时间戳，只有在同一时间戳下的事务才允许执行。

1. **系统的演进**

随着系统的完善和维护，数据模型必不可少的需要进行变更。为了保障数据的一致性和完整性，**版本控制**是重中之重，即记录数据模型的历史变更，允许回滚到之前的版本。而对于新版本的更新，必须要考虑向前兼容（新的版本可以读取旧版本的数据）和向后兼容（旧版本的应用程序能继续运行在新的数据模型上）。而为了保持系统的稳定性，可以考虑吧采取渐进式的演化，即逐步引入新特性，而非一次性大规模重构，从而减少风险。

1. **存储与检索**
2. **存储引擎**

在本小节，作者讨论了如下三种存储引擎，并指出不同的存储引擎适用于不同的场景和需求，没有绝对的优劣之分。在实际应用中，需要根据具体的业务需求和技术条件来选择合适的存储引擎。

* **日志结构存储引擎：**采用日志结构的方式来管理数据，具有较好的并发性和可靠性。代表性的存储引擎有 B+ 树、哈希表等。优点是可以高效地支持范围查询和排序操作，缺点是在更新操作方面效率较低。
* **面向页面的存储引擎：**将数据按照固定大小的页面进行划分，每个页面对应着磁盘上的一个物理块。代表性的存储引擎有哈夫曼树、位映射等。优点是可以在内存中快速访问数据，缺点是难以支持范围查询和排序操作。
* **列存储引擎**：将数据按列进行存储，相比于传统的按行存储方式，可以大幅降低存储空间的占用，并且能够更快地进行数据分析和查询。代表性的存储引擎有 Apache Cassandra、Google Bigtable 等。优点是能够高效地支持大规模数据分析和查询，缺点是对于一些小规模的事务性操作效率较低。

1. **索引技术**

本小节主要介绍了以下几种数据库中的索引技术，开发人员需要根据不同类型的查询模式，选择最适合的索引类型。

* **关键字索引**: 类似于关系模型中的主键索，用于唯一标识一条记录；
* **二级索引**: 辅助索引，用于有效地执行联接操作；
* **堆文件**：基于树形结构，适用于大量数据的排序和查找；
* **聚集索引**：将数据按照某个字段的值进行排序，并将数据存储在该字段的值上；
* **连接索引**：可以连接两个表格，可以快速找到符合条件的数据记录；
* **多维索引**：用于多维数据的索引方式；
* **模糊索引**：用于模糊查询，可以根据关键字的相似度来查找数据记录；
* **散列索引：**基于哈希表实现，适合精确匹配查询，但不支持范围查询；
* **B树索引：**支持范围查询和排序操作，常用于主键或唯一字段；
* **覆盖索引：**包含所有查询所需字段的索引，减少了额外的数据页访问次数。

然而，创建索引会占用额外的空间，并且会影响写入性能，因此在索引技术的采用上要权衡利弊。

1. **事务处理与分析**

事务处理（Online Transaction Processing**，**OLTP）强调实时处理大量短期交易，追求低延迟和高吞吐量, 主要用于保证业务正常运转；事务分析（Online Analytical Processing，OLAP）专注于复杂的数据分析任务，如聚合、汇总等，通常涉及大量的历史数据, 主要用于提高业务决策能力。其中，对于复杂的分析查询，可以采用预计算、物化视图、立方体等技术来加速响应时间。

作者提到，在实际开发中，需要考虑既能满足OLTP又能兼顾OLAP需求的系统架构，如HTAP（Hybrid Transaction/Analytical Processing），它允许在同一个系统中同时执行事务处理和分析处理，并且能够实现实时响应和批量处理的需求。同时，为了避免OLTP和OLAP互相干扰，还可以通过物理分离或逻辑隔离的方式来分配资源。

1. **编码与演化**
2. **数据编码**

数据的编码方式有如下两种，在选择时要权衡编码方式的性能、兼容性和易用性。同时，考虑到敏感信息的保护，可以利用部分编码格式提供的加密技术。

* 文本编码：主要用于人类可读的配置文件或API交互，如JSON、XML等；
* 二进制编码：旨在提高序列化和反序列化的效率，适用于内部通信或大数据传输，如Protocal Buffers、Thrift等。

1. **数据压缩**

数据压缩的核心是压缩算法，常见的压缩算法有哈夫曼编码、LZ77、LZW等。这些算法通过对数据进行编码或者删除冗余信息来实现压缩。在系统开发中，应当根据实际需求选择适当的压缩算法，比如在网络带宽有限的情况下优先考虑压缩率高的算法。通常情况下，数据压缩适用于需要传输或存储大量数据的场景，如云计算、大数据分析等领域。通过压缩数据，可以减少网络流量和存储空间，降低成本和提高效率，但也同时增加了CPU开销，因此需要综合考虑。

此外，作者还提出数据压缩的集成方案，即将压缩集成到整个数据生命周期中，从生成、传输到存储，形成一套完整的解决方案。

1. **第二部分：分布式数据**

**图示

描述已自动生成**

图3 第二部分框架结构

1. **复制**

在数据系统中，**复制**指在多个节点上保存相同数据的副本，以提高系统的容错能力、降低延迟和增加读取吞吐量。常见的复制算法有：

* **单领导复制**：指定一个节点作为领导者，所有写操作都必须先发送给领导者，然后由领导者同步或异步地传播给其他跟随者节点；
* **多领导复制**：允许多个节点同时接受写操作，再通过某种机制解决冲突；
* **无领导复制**：任何节点都可以接受写操作，客户端直接与最近的节点通信。

此外，复制操作还分为同步复制和异步复制，区别在于写操作后是否立即返回成功响应。前者数据安全性和一致性较高，但是性能较低、对网络延迟敏感；后者性能较好、减少了等待时间，但存在数据丢失的风险，网络故障可能会造成数据不一致。

1. **分区**

**分区**指将大型数据库拆分为较小的子集（称为分区或分片），以便不同的分区可以分配给不同的节点处理。分区操作可以提示系统的可扩展性，使得系统能够处理更大的数据量和更高的并发请求；同时也可以减少热点问题，即某个特定的数据项被频繁访问而导致的性能瓶颈。常见的分区策略如下：

* **哈希分区**：根据数据项的关键字段计算哈希值，并将该哈希值映射到某个分区，适用于随机访问模式；
* **范围分区**：按照关键字段的值范围划分数据，每个分区包含一定范围内的数据，适合顺序扫描和范围查询；
* **一致性哈希**：使用虚拟节点的概念，使新增或移除物理节点时影响最小化。

然而在进行数据分区时，需要考虑到数据分布不均可能导致某些节点负载过高而其他节点闲置的情况；此外，若分区键选择不当会影响查询效率，甚至导致跨分区的查询变得复杂。

1. **分布式事务**

相较于普通的事务，分布式事务会跨越多个节点的数据操作，通常涉及到多个数据源的操作，如数据库、消息队列等。常见的分布式事务模型包括强一致性的ACID模型和松散一致性的BASE模型，前者保证了事务的“ACID”四个特性，而后者则强调了基本的可用性和数据弹性。

基于ACID模型，有两种经典的分布式事务协议：

* 两阶段提交（Two-Phase Commit, 2PC）：分为准备阶段和提交阶段，确保多个数据库节点全部提交或全部中止事务；
* 三阶段提交（Three-Phase Commit, 3PC）：在2PC基础上增加了一个预提交阶段，旨在减少阻塞的可能性。

1. **分布式系统的基本问题**

在分布式系统中，存在一些常见的基本问题：

* **网络分区**：由于网络故障或其他原因，导致集群内的一部分节点无法与其他部分通信。这可能会破坏系统的可用性和一致性，需要采取措施如仲裁机制来应对。
* **时钟同步：**对于基于时间戳的操作，需要确保不同节点上的时钟尽可能接近；然而，由于硬件差异和网络延迟等因素，难以做到绝对同步。
* **故障检测与恢复：**由于分布式系统的多节点特性，需要及时发现并修复系统中的故障，以维持系统的稳定运行；常常采用心跳检测、超时机制等手段，并结合自动化的故障转移和自愈算法等。

对于诸如以上的难以避免的问题，在开发过程中需要考虑容错机制，来尽可能地保障系统的稳定性和可靠性，例如：冗余设计（数据复制、多路径传输等）、隔离、重试（但要避免无限循环重试）。

1. **一致性与共识**
2. **一致性模型**

一致性模型是指分布式系统中多个节点之间数据同步的方式和规则。常见的一致性模型包括：

* **强一致性模型**：所有节点看到的数据都是最新的版本，不存在陈旧数据
* **基于复制**的一致性模型：多个节点之间共享相同的数据副本，通过对数据的操作进行同步来保证一致性；
* **基于因果关系**的一致性模型：通过维护操作之间的因果关系来保证数据的一致性；
* **弱一致性模型**：不要求所有节点立即看到最新数据，允许有一定的延迟
* **基于分区容错性**的一致性模型：将数据划分为多个分区，并允许某些分区出现故障而不影响整个系统的运行；
* **基于时间旅行禁止**的一致性模型：不允许时间旅行，即不允许修改过去的状态，以此来保证数据的一致性；
* **最终一致性模型**：经过一段时间后，所有节点最终会收敛到相同的状态
* **基于线性一致性**的一致性模型：所有操作按照线性时间轴顺序执行，保证操作的原子性和有序性；
* **基于最终一致性**的一致性模型：最终达到一致性状态，但不一定实时同步，适用于大规模分布式系统。

1. **共识算法**

共识算法是指在一个分布式系统中，多个节点之间如何达成一致的方法，是实现一致性模型的具体方法。常见的共识算法包括：

* Paxos：一种基于消息传递的协议，通过多次提议和接受过程来达到一致性；
* Raft：在Paxos算法的基础上发展而来，引入了leader选举和日志复制等机制，更加易于理解和实现；
* ZAB：Zookeeper（一种开源的分布式应用程序协调服务）中采用的一致性协议，通过心跳检测和领导者选举等方式来保持一致性。

同时，作者提到了共识算法的“**终止属**性”，即算法必须取得进展，即使部分节点出现故障，其他节点也必须达成一项决定。此外，作者还提到了“**拜占庭式错误**”，即节点没有正确地遵循协议，从而破坏协议的安全属性。这写都需要一定的容错机制，作者指出共识算法需要至少占总体多数的节点正确工作，以确保终止属性得到满足。

1. **第三部分：衍生数据**

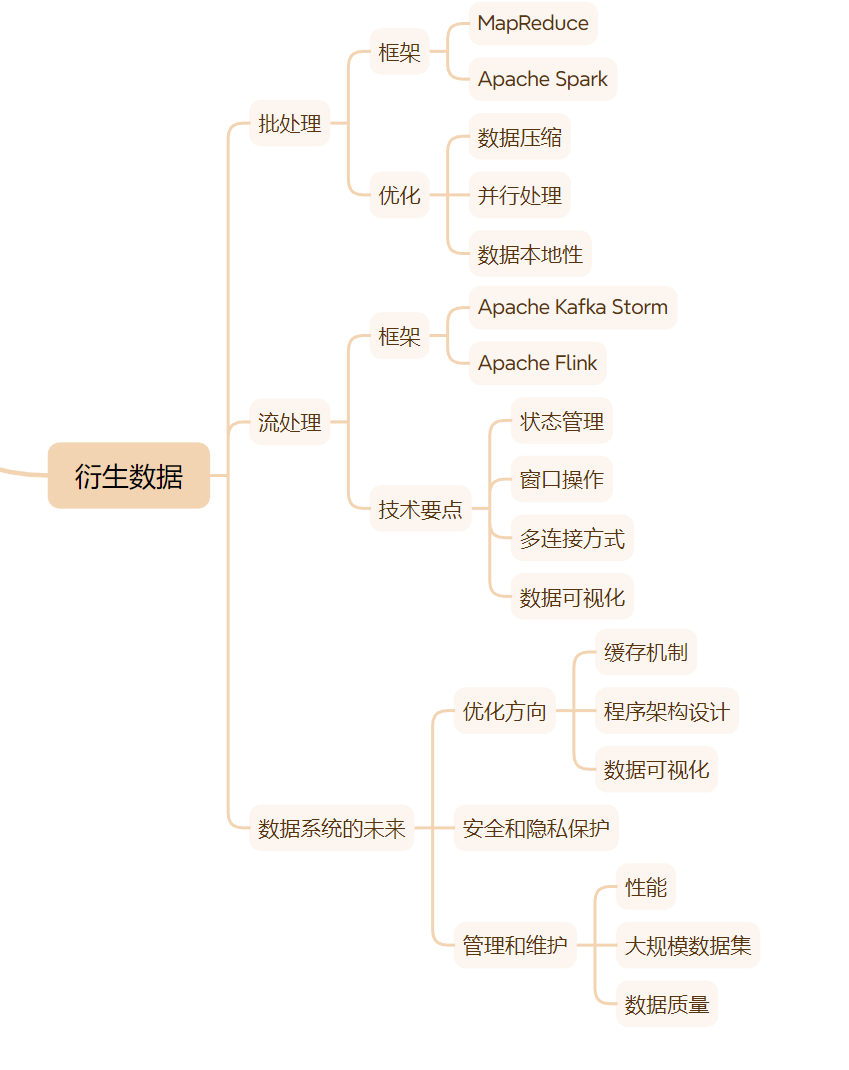
****

图4 第三部分框架结构

1. **批处理**

批处理（Batch Processing）指以批量方式处理大量数据，可以在后台运行，无需对用户请求进行即时响应；并且，批处理是资源密集型任务，可能涉及大量的计算资源，如CPU、内存和磁盘I/O。它常用于处理大规模的数据集，执行复杂的数据转换和聚合操作；也常用于定期更新统计数据或创建历史记录，如日志分析、用户行为跟踪等。

1. **批处理框架**

批处理可以通过多种技术手段来实现，常用的批处理框架包括：

* MapReduce：一种分布式计算框架，可以将大数据集分成若干个小块，分别在不同的计算机节点上进行并行计算和处理（Map阶段），最后将结果汇总起来（Reduce阶段）。它适用于处理海量数据，支持水平扩展；但不适合低延迟需求的应用场景
* Apache Spark：基于内存计算的分布式处理框架，提供比MapReduce更高的性能，同时支持流处理、机器学习等功能。它兼容Hadoop生态系统，API友好，支持多种编程语言，适合复杂的多步骤数据处理流程；但相比MapReduce，内存消耗较大，对硬件要求更高。

1. **批处理的优化**

对于“批处理”任务的优化，作者提到了以下三种概念：

* 数据压缩：（参考2.4.2）可以减少存储空间占用，加快数据传输速度；
* 并行处理：通过将数据分成多个部分并在不同的节点上并行处理来提高处理效率；同时需要考虑负载均衡、容错机制等问题；
* 数据本地性：尽可能让数据存储在离计算节点近的地方，以减少数据传输的时间和网络带宽的消耗；这可以通过调度器根据节点位置分配任务，优先选择拥有相关数据副本的节点执行计算来实现。

1. **流处理**

流处理（Stream Processing）指实时地处理连续到达的数据流，即对动态变化的数据即时进行不间断地分析和报告；不同于批处理的一次性作业，它是持续不断的过程，通常用于事件驱动架构中，如监控系统、推荐引擎等。

1. **流处理框架**

常见的流处理框架包括：

* Apache Kafka Storm: 一个轻量级的流处理框架，在Kafka集群上运行，它可以将数据流转换成一系列处理步骤，每个步骤都会产生一个新的数据流；它同时还支持窗口操作和状态管理，可以根据时间或者数量等方式对数据进行聚合，并维护数据的状态；
* Apache Flink: 支持流处理和批量处理，并且具有高可用性和容错能力；其核心思想是将所有数据看作一条无限长的流，对流进行切片，每个切片称为一个批次，对批次进行处理；同时采用了基于状态机的状态管理机制，将数据划分为多个状态，并根据状态更新的时间顺序进行处理。

1. **流处理的技术核心**

然而，作者还提到了流处理的一些技术核心：

* **状态管理**：即流处理需要管理和维护流处理过程中产生的中间状态，确保即使发生故障也能恢复到正确状态。通常，采用检查点机制定期保存状态快照，结合容错算法可以实现快速恢复。
* **窗口操作**：指流处理框架可以根据时间或数量等方式对数据进行聚合。这种方式可以帮助用户快速地分析数据，发现数据中的规律和趋势，从而做出更加准确的决策。此外，在进行延迟和吞吐量的权衡时，也可以通过微调窗口大小来找到最佳平衡点。
* **多连接方式**：支持多种连接方式，如流流连接、流表连接、表表连接等。不同的连接方式适用于不同的场景。
* **数据可视化**：流处理结果需要进行可视化展示，以便用户理解和决策。

1. **数据系统的未来**
2. **数据系统的未来发展方向**

* **更好的缓存机制**：作者探讨了如何通过更智能的缓存策略提高数据访问速度，减少延迟。例如，使用分布式缓存、内容分发网络（CDN）和边缘缓存等技术来优化用户体验。
* **更加灵活的应用程序设计**：作者强调了应用程序架构的灵活性和适应性，如微服务架构、Serverless架构（一种无服务器架构，开发者只需关注业务逻辑，无需关心底层基础设施的管理和维护）等，使应用能够快速响应变化的需求和技术进步。
* **更好的数据可视化**：作者分析了如何利用先进的可视化工具和技术，帮助用户更好地理解和分析数据。包括交互式仪表盘、实时数据流可视化和复杂关系图谱等。

1. **数据系统的安全和隐私保护**

* **确保数据的安全性和隐私性**：有多种技术和管理措施可以保护数据免受未经授权的访问和泄露，如加密存储、访问控制列表（ACL）、多因素认证等。
* **应对潜在的安全威胁**：常见的安全威胁类型及其防范方法有：SQL注入攻击、跨站脚本攻击（XSS）、分布式拒绝服务（DDoS）攻击等。同时作者还介绍了入侵检测系统（IDS）、防火墙和其他防护工具的作用。

1. **数据系统的管理和维护**

* **优化性能**：作者探讨了如何通过对硬件资源的有效分配、查询优化、索引策略调整等方式提升系统的整体性能，确保高效的数据处理能力。
* **管理大规模数据集**：常见的处理海量数据的技术有：数据分片、压缩、增量备份等，同时还需要选择合适的数据库管理系统（DBMS）来满足特定需求。
* **保证数据的质量**：作者强调了数据质量的重要性，并提出了具体的方法来提高数据的准确性、完整性和一致性，如数据清洗、验证规则设置、定期审计等。

1. **读后感**

阅读《Designing Data-Intensive Applications》这本书，不仅仅是一次技术知识的积累，更是一场对现代数据系统设计哲学的深刻思考。书中丰富的理论和技术细节固然重要，但真正打动我的是它所传递的设计理念和思维方式。

* **设计哲学：从复杂中寻求简单**

在当今这个数据爆炸的时代，构建一个高效、可靠且可扩展的数据密集型应用并非易事。书中强调了简化复杂性的必要性——通过抽象和模块化设计，将复杂的系统分解为易于理解和维护的小部分。这种“化繁为简”的设计理念让我深受启发。它提醒我们，在面对复杂问题时，不应被表面现象迷惑，而是要深入挖掘本质，寻找最简洁有效的解决方案。

* **技术选择：权衡与折衷的艺术**

书中多次提到，在设计数据系统时，往往需要在不同需求之间做出权衡，如一致性与可用性、性能与资源消耗等。这让我意识到，没有一种方案能够适用于所有情况，每个决策都伴随着一定的取舍。作为软件系统的开发者，需要根据具体应用场景灵活调整策略，找到最适合的平衡点。这种思维方式不仅适用于技术领域，也对我们日常生活中的决策有着重要的启示意义。

* **持续学习：适应快速变化的技术环境**

随着新技术不断涌现，保持学习的热情和能力变得尤为重要。书中介绍了许多前沿技术和未来发展趋势，激发了我对未知领域的探索欲望。同时，作者也指出，即使是最先进的技术也有其局限性，关键在于如何结合实际情况加以应用。这促使我反思自己的技术学习路径，认识到持续学习和自我更新的重要性，以适应快速变化的技术环境。