Designing Data-Intensive Applications

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

Part 1. Foundation of Dada Systems

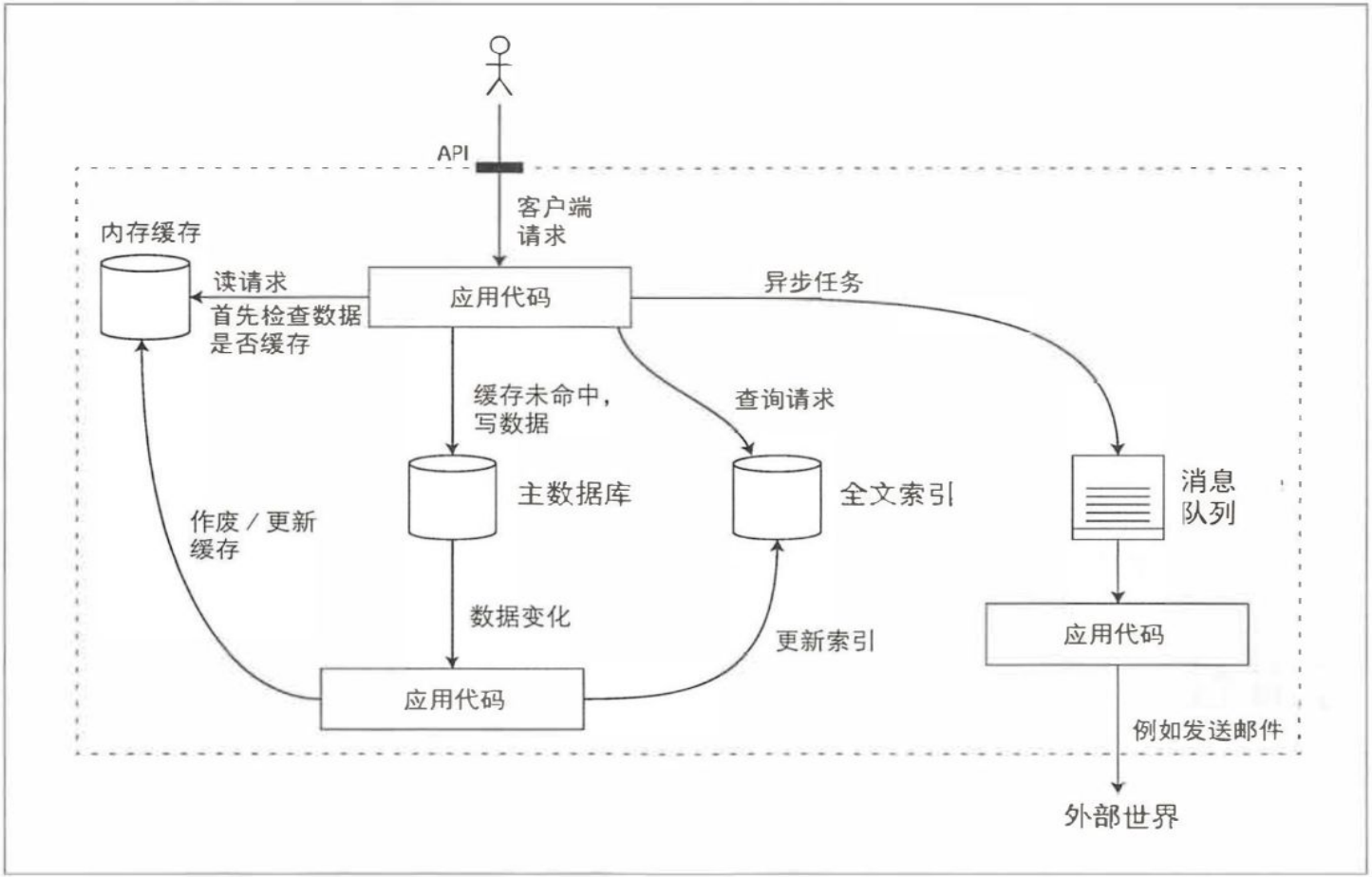
1: Reliable, Scalable and Maintainable Applications (07/27)

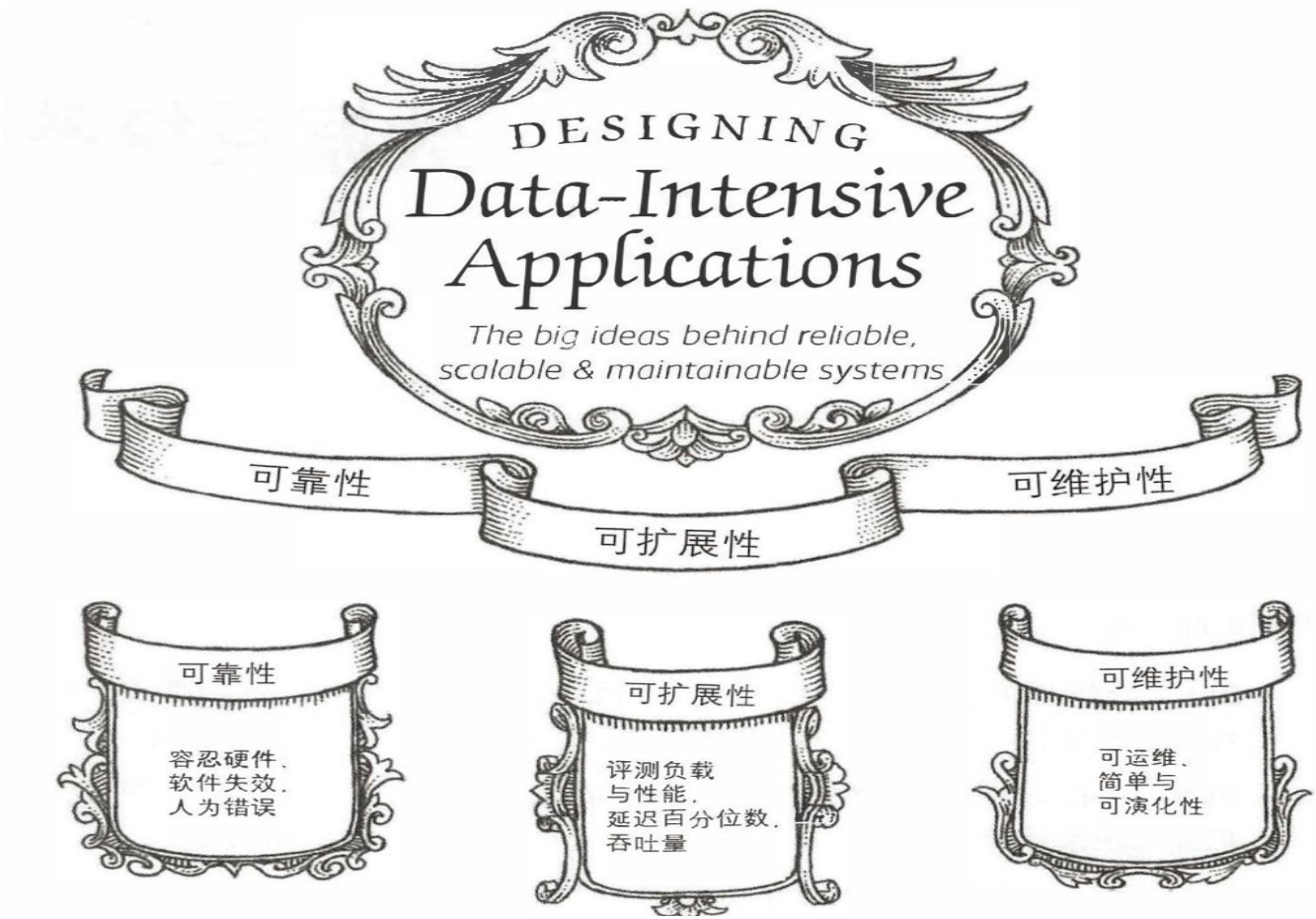
2: Data Models and Query Languages
3: Storage and Retrieval
4: Encoding and Evolution

当今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，许多应用系统都包含以下模块：

- 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。


API
客户端
请求
内存缓存
读请求
首先检查数据
是否缓存
应用代码
异步任务
缓存未命中，
写数据
查询请求
主数据库
全文索引
消息
队列
作废 / 更新
缓存
数据变化
更新索引
应用代码
应用代码
例如发送邮件
外部世界


DESIGNING
Data-Intensive Applications
The big ideas behind reliable, scalable & maintainable systems
可靠性
可扩展性
可维护性
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

可靠性

每个人脑子里都有一个直观的认识，即什么意味着可靠或者不可靠。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。

如果所有上述目标都要支持才算“正常工作”，那么我们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。

Fault-tolerant System

Hardware Fault,

我们的第一个反应通常是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。

Software Errors,

软件系统问题有时没有快速解决办法，而只能仔细考虑很多细节，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[12]。

Human Errors

- 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，使“做正确的事情”很轻松，但搞坏很复杂。但是，如果限制过多，人们就会想法来绕过它，这会抵消其正面作用。因此解决之道在于很好的平衡。
- 想办法分离最容易出错的地方、容易引发故障的接口。特别是，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，万一出现问题，不会影响真实用户。
- 充分的测试：从各单元测试到全系统集成测试以及手动测试[3]。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。
- 当出现人为失误时，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（这样任何意外的错误仅会影响一小部分用户），并提供校验数据的工具（防止旧的计算方式不正确）。
- 设置详细而清晰的监控子系统，包括性能指标和错误率。在其他行业称为遥测（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[14]。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。这非常重要而且比较复杂，具体内容已超出本书范围。

可扩展性

即使系统现在工作可靠，并不意味着它将来一定能够可靠运转。发生退化的一个常见原因是负载增加：例如也许并发用户从最初的10 000个增长到100 000个，或从100万到1000万；又或者系统目前要处理的数据量超出之前很多倍。

可扩展性是用来描述系统应对负载增加能力的术语。但是请注意，它并不是衡量一个系统的一维指标，谈论“X是可扩展”或“Y不扩展”没有太大意义。相反，讨论可扩展性通常要考虑这类问题：“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”。

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。如果以图1-2的关系型数据模型，可以执行下述的SQL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
```

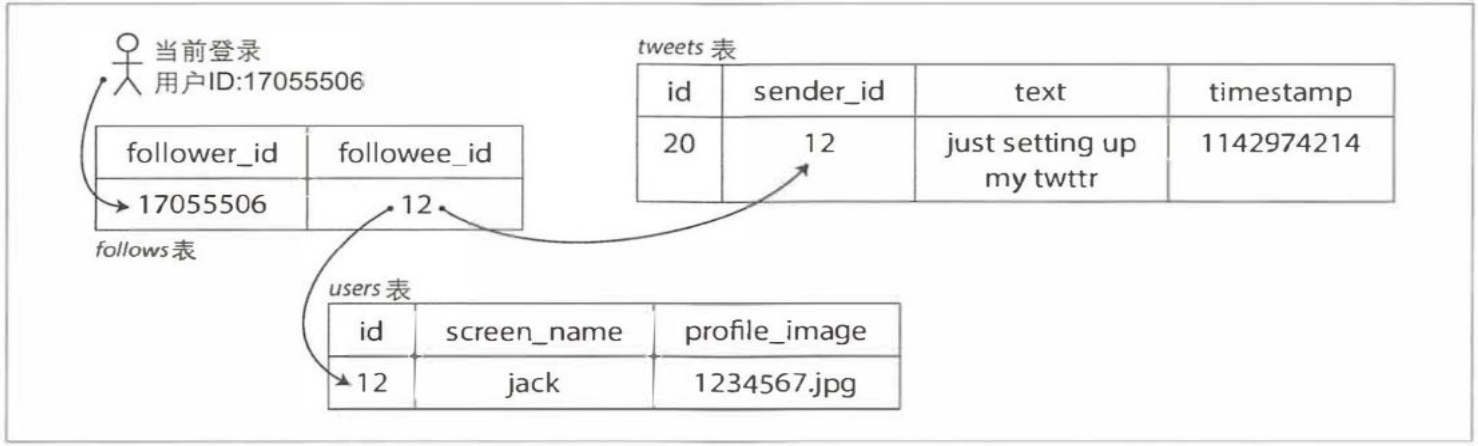


图1-2：采用关系型数据模型来支持时间线

2. 对每个用户的时间线维护一个缓存，如图1-3所示，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。

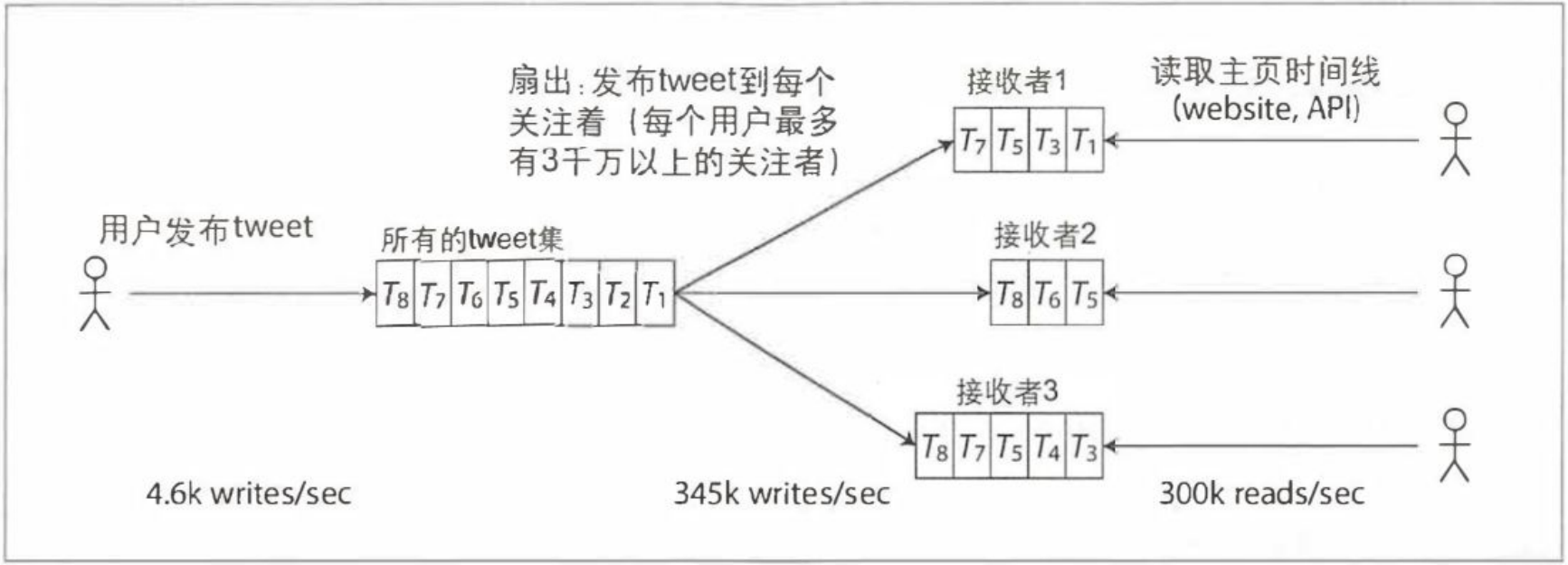


图1-3： Twitter的数据流水线方式来推送tweet，相关参数来自2012.11 [16]

描述性能

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？

Throughput, Response Time, Vertical Scaling, Horizontal Scaling

---

注3：理想情况下，批量作业的运行时间是数据集的总大小除以吞吐量。在实践中，由于倾斜（数据在多个工作进程中不均匀分布）问题，系统需要等待最慢的任务完成，所以运行时间往往更长。

可维护性

可运维性

方便运营团队来保持系统平稳运行。

简单性

简化系统复杂性，使新工程师能够轻松理解系统。注意这与用户界面的简单性并不一样。

可演化性

后续工程师能够轻松地对系统进行改进，并根据需求变化将其适配到非典型场景，也称为可延伸性、易修改性或可塑性。

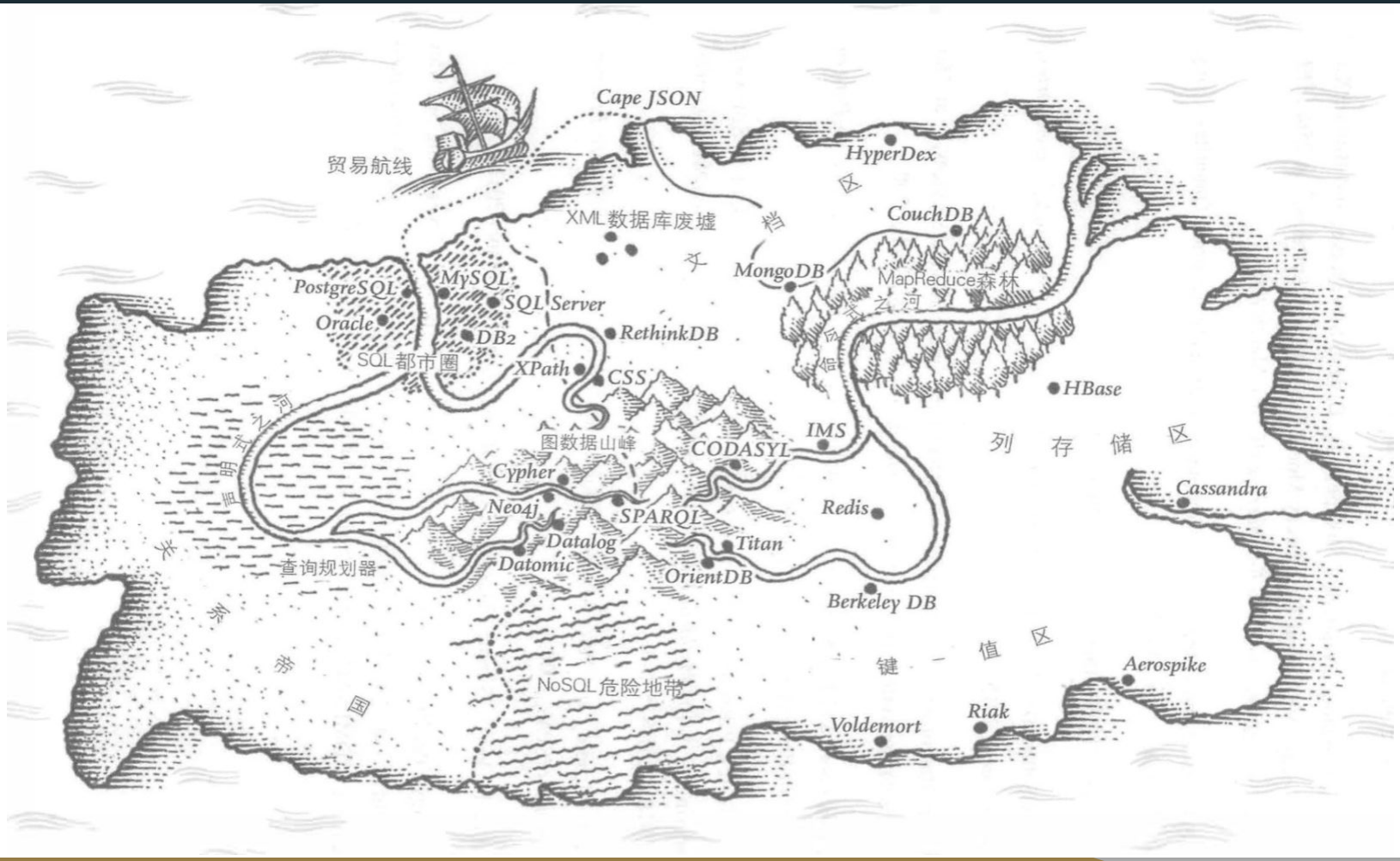
良好的可操作性意味着使日常工作变得简单，使运营团队能够专注于高附加值的任务。数据系统设计可以在这方面贡献很多，包括：

- 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，正如Moseley和Marks[32]把复杂性定义为一种“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。

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[34]。


贸易航线
Cape JSON
HyperDex
XML数据库废墟
文档区
CouchDB
MongoDB
MapReduce森林
PostgreSQL
MySQL
SQL Server
Oracle
DB2
RethinkDB
SQL都市圈
XPath
CSS
图数据山峰
CODASYL
IMS
HBase
列存储区
Cypher
Neo4j
SPARQL
Redis
Cassandra
Datalog
Titan
Datomic
查询规划器
OrientDB
Berkeley DB
键值区
NoSQL危险地带
Aerospike
Voldemort
Riak